100 Go Mistakes and How to Avoid Them

Teiva Harsanyi

MEAP

MANNING

100 Go Mistakes and How to Avoid Them

Teiva Harsanyi

MEAP

MANNING

1 Go: Simple to learn but hard to master

This chapter covers

- What makes Go an efficient, scalable, and productive language
- Exploring why Go is simple to learn but hard to master
- Presenting the common types of mistakes made by developers

Making mistakes is part of everyone's life. As quoted by Albert Einstein:

> Anyone who has never made a mistake has never tried anything new.
>
> -- Albert Einstein

What matters, in the end, isn't the number of mistakes we make but our capacity to learn from them. This assertion also applies to programming. The seniority we acquire in a language isn't a magic process; it involves making many mistakes and learning from them.

The purpose of this book is centered around this idea. It will aim to help you be a proficient Go developer by discussing 100 common mistakes in

many areas of the language.

In this chapter, we'll have a quick refresher as to why Go became mainstream over the years, understand why despite being considered simple to learn, mastering the nuances of Go can be challenging, and, finally, introduce the concepts we will cover in this book.

1.1 Go outline

If you read this book, it's likely that you're already convinced about Go. Therefore, let's be brief to remind us what makes Go a powerful language.

Software engineering has evolved heavily during the past decades. Most modern systems aren't written anymore by a single person but by teams consisting of multiple programmers, sometimes even hundreds, if not thousands. Nowadays, a code must be readable, expressive, and maintainable to guarantee a system's durability over the years. Meanwhile, in a fast-moving world, maximizing agility and reducing the time to market is critical for most organizations. Programming should also follow this trend and ensure that software engineers are as productive as possible when reading, writing, and maintaining code.

In response to these challenges, Google has conceived the Go programming language in 2007. Since then, many organizations have adopted the language to support various use cases: API, automation, databases, CLI (Command Line Interfaces), etc. Meanwhile, many today consider Go as the language of the cloud.

Feature-wise, Go has no type inheritance, no exceptions, no macros, no partial functions, no support for lazy variable evaluation or immutability,

no operator overloading, no pattern matching, etc. Why are these features missing in the language? The official Go FAQ gives us some insights:

> Why does Go not have feature X? Your favorite feature may be missing because it doesn't fit, because it affects compilation speed or clarity of design, or because it would make the fundamental system model too difficult.
>
> -- Go FAQ

Judging the quality of a programming language via its number of features is probably not an accurate metric. At least, it's not an objective of Go.

Instead, Go leverages a few essential characteristics when adopting a language at the scale of an organization, including:

- Stability: Even though Go receives frequent updates (including improvements and security patches), it has remained a very stable language over the past years. One may even consider it as one of the best *features* of the language.
- Expressivity: We can define expressivity in a programming language by how natural and intuitive we can write and read code. A reduced number of keywords and limited ways to solve common problems make Go an expressive language for large codebases.
- Compilation: As developers, what can be more exasperating than having to wait for a build to test our application, for example? Targeting fast compilation times has always been

a conscious goal from the language designers to enable productivity.
- Safety: Go is a strongly, statically typed language. Hence, it has strict compile-time rules that ensure the code write is type-safe in most cases.

Meanwhile, Go was built from the ground with solid features such as outstanding concurrency primitives with goroutines and channels. There's no strong need to rely on external libraries to build efficient concurrent applications. Observing how important concurrency is these days also demonstrates why Go is such strong language for the present and probably also for the future.

However, one may also consider Go as a simple language. And in a sense, this isn't necessarily wrong. For example, a newcomer can learn the language's main features in less than a day. So why read a book centered on the concepts of mistakes if Go is simple?

1.2 Simple doesn't mean easy

There is a subtle difference between simple and easy. Simple applied to a technology means not complicated to learn or to understand. However, easy means that we would achieve everything without much effort. Go is simple to learn but not necessarily easy to master.

Let's take concurrency, for example. In 2019, a study focusing on concurrency bugs was published: Understanding Real-World Concurrency Bugs in Go[1]. This study was the first systematic analysis of concurrency bugs. It focused on multiple popular Go repositories such as Docker, gRPC, or Kubernetes. One of the most important takeaways from this

study is that most of the blocking bugs were caused by an inaccurate usage of the message passing paradigm (via channels), despite the belief that message passing is easier to handle and less error-prone than sharing memory.

So what should be an appropriate reaction to such a takeaway? Should we consider that the language designers were wrong about message passing? Should we reconsider how we deal with concurrency in our project? Of course not.

It’s not a question to confront message passing vs. sharing memory and find who the winner is. However, it’s up for us, Go developers, to thoroughly understand how to leverage concurrency, the implications on modern processors, when to favor one approach over the other, and how to avoid common traps.

This example highlights that although a concept such as channels and goroutines can be simple to learn, it isn’t an easy topic in practice.

This leitmotif, simple doesn’t mean easy, can be generalized to many aspects of Go, not only concurrency. Hence, to be a proficient Go developer, we must have a thorough understanding of many aspects of the language, requiring time, effort, and mistakes.

This book aims to help you accelerate your journey towards proficiency in delving into 100 Go mistakes.

1.3 100 Go mistakes

Why should you read a book about common Go mistakes? Why not deepen our knowledge with an ordinary book that would dig into different topics?

Neuroscientists proved that facing mistakes are the best time for brain growth[2]. Haven't you already experienced the process of learning from a mistake and recalling months or even years after the context related to it? As presented in *Learning from Errors* by Janet Metcalfe, this characteristic is because mistakes have a facilitative effect. The main idea is that we can remember not only the error but also the context surrounding the mistake. This is one of the reasons why learning from mistakes is so efficient.

Therefore, to strengthen this facilitative effect, each mistake will, as much as possible, be accompanied by real-world examples. This book isn't only about theory; it aims to help you in getting better to avoid mistakes and make conscious decisions because you understand the rationale behind them.

> Tell me and I forget. Teach me and I remember. Involve me and I learn.
>
> -- Benjamin Franklin

Overall, all the mistakes can be classified into seven main categories:

- Bugs
- Needless complexity
- Weaker readability
- Lack of API convenience

- Sub-optimal organization
- Under-optimized code
- Lack of productivity

1.3.1 Bugs

The first type of mistake, and probably the most obvious, is software bugs. In 2003, a study conducted by Synopsys estimated the cost of software bugs in the US alone to be over $2 trillion. Furthermore, bugs can also lead to even more tragic impacts. We can, for example, mention cases such as the Therac-25, a radiation therapy machine produced by Atomic Energy of Canada Limited (AELC). Because of a race condition, the machine gave its patients radiation doses that were hundreds of times greater than expected, leading to the death of three patients. Hence, software bugs aren't only about money, and as developers, we should remember how impactful our jobs are.

This book will cover plenty of cases that could lead to various software bugs, including data races, leaks, logic errors, and other defects. Though accurate tests should be a way to discover such bugs as early as possible, we may sometimes miss some cases because of different factors such as time pressure or complexity. Therefore, it's essential as a Go developer to make sure we can avoid these common bugs.

1.3.2 Needless complexity

The next category of mistake is related to unnecessary complexity. A significant part of software complexity comes from the fact that, as developers, we strive to think about imaginary futures. Instead of solving concrete problems for right now, it can be tempting to build evolutionary

software that could tackle whatever future use cases. However, it leads to more drawbacks than benefits in most cases as it can make a codebase more complex to understand and reason about.

Coming back to Go, we can think about plenty of use cases where developers might be tempted to design abstractions for future needs, such as interfaces or generics. This book will discuss topics where we should remain careful not to harm a codebase with needless complexity.

1.3.3 Weaker readability

Another kind of mistake is to weaken readability. As Robert C. Martin wrote in *Clean Code: A Handbook of Agile Software Craftsmanship*, the ratio of time spent reading versus writing is well over 10 to 1. Most of us have started to program on solo projects where readability wasn't that important. However, software engineering is programming with a time dimension: making sure we can still work and maintain an application during months, years, or perhaps even decades.

When programming in Go, we can make many mistakes that can harm readability, such as nested code, data types representation, not using named result parameters in some cases. Throughout this book, we will understand how to write readable code and care for future readers (including future selves).

1.3.4 Unidiomatic organization

Be it while working on a new project or because we acquire inaccurate reflexes, another type of mistake is organizing our code and a project

suboptimally and unidiomatically. Such issues can make a project harder to reason about and maintain.

This book will cover some of these common mistakes in Go. For example, how to structure a project, deal with utility packages or `init` functions. All in all, it should help us to organize our code and projects efficiently and idiomatically.

1.3.5 Lack of API convenience

Making common mistakes that weaken how convenient an API is for our clients is another type of mistake. If an API isn't user-friendly, it will be less expressive, hence harder to understand and more error-prone.

We can think about many situations that would affect how convenient an API is, such as overusing `any` types, using the wrong creational pattern to deal with options, or applying blindly standard practices from object-oriented programming. Hence, we will cover common mistakes preventing us from exposing convenient API for our users.

1.3.6 Under-optimized code

Under-optimized code is another type of mistake made by developers. It can be because of various reasons, such as a lack of understanding about some language features or even fundamental knowledge. Performance is one of the most obvious impacts but not only. We can think about optimizing a code for other goals, such as accuracy.

For example, in this book, we will see some common techniques preventing us from making floating-point operations accurate. Meanwhile,

we will cover plenty of cases that can negatively impact performance code because of a poorly parallelized execution, not knowing how to reduce allocations, or the impacts of data alignment, for example. We will tackle optimization via different prisms.

1.3.7 Lack of productivity

In most cases, what's the best language we can choose when working on a new project? The one we're the most productive with. Being comfortable with how a language works and exploiting it to get the best out of it is crucial to reach proficiency.

In this book, we will cover many cases and concrete examples that will help be more productive while working in Go. For instance, writing efficient tests to ensure our code works, relying on the standard library to be more effective, or getting the best out of the profiling tools and linters.

Now, it's time to delve into these 100 common Go mistakes.

1.4 Chapter summary

- Go is a modern programming language that puts a lot of effort into developer productivity, which is crucial for most companies today.
- Go is simple to learn but not easy to master; this is why we need to deepen our knowledge to make the most effective use of the language.
- Learning via mistakes and concrete examples is a powerful way to be proficient in a language. This book aims to help you accelerate your path to proficiency via 100 common mistakes.

[1] Understanding Real-World Concurrency Bugs in Go by Tengfei Tu, Xiaoyu Liu, Linhai Song and Yiying Zhang

[2] J. S. Moser et al., “Mind Your Errors: Evidence for a Neural Mechanism Linking Growth Mind-set to Adaptive Posterror Adjustments,” Psychological Science 22/12 (2011)

2 Code and project organization

This chapter covers

- Organizing our code idiomatically
- Dealing efficiently with abstractions: interfaces and generics
- Best practices regarding how to structure a project

Organizing a Go codebase in a clean, idiomatic, and maintainable manner isn't an easy task. It requires experience and mistakes to understand all the best practices related to code and project organization. For example, what are the traps to avoid, such as variable shadowing and nested code abuse, how to structure packages, when and where to use interfaces or generics, init functions, utility packages? In this chapter, we will delve into common organizational mistakes.

2.1 Unintended variable shadowing

The scope of a variable refers to the places a variable can be referenced. In other words, the part of an application where a name binding is valid. In Go, a variable name declared in a block may be redeclared in an inner block. This principle, called variable shadowing, is prone to common mistakes.

In the following example, we will see an unintended side effect because of a shadowed variable. We will create an HTTP client in two different ways, depending on the value of a `tracing` boolean:

```go
1  var client *http.Client
2  if tracing {
3          client, err := createClientWithTracing()
4          if err != nil {
5                  return err
6          }
7          log.Println(client)
8  } else {
9          client, err := createDefaultClient()
10         if err != nil {
11                 return err
12         }
13         log.Println(client)
14 }
15 // Use client
```

Declare a client variable 1
Create an HTTP client with tracing enabled, the client variable is shadowed in this block 2
Create a default HTTP client, the client variable is also shadowed in this block 3

First, we declare a `client` variable. Then, we use the short variable declaration operator (`:=`) in both inner blocks to assign it to the inner `client` variables, not the outer one. As a result, following this example, the outer variable will always be nil.

> **Note**
>
> This code compiles because the inner `client` variables are used in the logging calls. If not, we would have compilation errors: `client declared and not used` .

How can we ensure assigning a value to the original `client` variable? There are two different options.

The first option is to use temporary variables in the inner blocks this way:

```go
var client *http.Client
if tracing {
        c, err := createClientWithTracing()
        if err != nil {
                return err
        }
        client = c
} else {
        // Same logic
}
```

Create a c temporary variable 1
Assign this temporary variable to client 2

Here, we assign the result to a temporary `c` variable whose scope is only within the `if` block. Then, we assign it back to the `client` variable. Meanwhile, we do the same for the `else` part.

The second option is to use the assignment operator (`=`) in the inner blocks to directly assign the function results to the `client` variable. However, it requires creating an `error` variable as the assignment operator works only if a variable name has already been declared:

```go
var client *http.Client
var err error
if tracing {
        client, err = createClientWithTracing()
        if err != nil {
                return err
        }
} else {
        // Same logic
}
```

Declare an err variable 1
Use the assignment operator to assign the *http.Client returned to the client variable directly 2

Instead of assigning first to a temporary variable, we can directly assign it to `client`.

Both options are perfectly valid. The main difference is that we perform only one assignment in the second option, which may be considered easier to read. Also, with the second option, error handling can be mutualized and implemented outside of the `if/else` statements:

```
if tracing {
        client, err = createClientWithTracing()
} else {
        client, err = createDefaultClient()
}
if err != nil {
        // Common error handling
}
```

Variable shadowing occurs when a variable name is redeclared in an inner block, and we've seen that this practice is prone to mistakes. Imposing a rule to forbid shadowed variables depends on our personal taste. For example, sometimes, it can be pretty convenient to reuse an existing variable name, like `err` for errors. Yet, in general, we should remain cautious as we have seen that we can face a scenario where a code compiles, but the variable that received the value was not the one expected. Later in this chapter, we will also see how to detect shadowed variables, which may help us spot possible bugs.

The following section will see why it is important to avoid abusing nested code.

2.2 Unnecessary nested code

A mental model applied to software is an internal representation of a system's behavior. While programming, we have to keep maintaining mental models, for example, about overall code interactions and function implementations. A code is qualified as readable based on multiple criteria such as naming, consistency, formatting, etc. A readable code will require less cognitive effort to maintain a mental model; hence it will be easier to read and maintain.

A critical aspect of readability is the number of nested levels. Let's do an exercise. Suppose that you work on a new project and have to understand what the following function `join` function does:

```go
func join(s1, s2 string, max int) (string, error) {
        if s1 == "" {
                return "", errors.New("s1 is empty")
        } else {
                if s2 == "" {
                        return "", errors.New("s2 is empty")
                } else {
                        concat, err := concatenate(s1, s2)
                        if err != nil {
                                return "", err
                        } else {
                                if len(concat) > max {
                                        return concat[:max], nil
                                } else {
                                        return concat, nil
                                }
                        }
                }
        }
}

func concatenate(s1 string, s2 string) (string, error) {
        // ...
}
```

We call a concatenate function that will perform some specific concatenation work but may return some errors 1

This function concatenates two strings and returns a substring if the length is greater than `max`. Meanwhile, it handles checks on `s1`, `s2` and if the call to `concatenate` returns an error. From an implementation perspective, this function is correct. However, building a mental model encompassing all the different cases is probably not a straightforward task. Why? Because of the number of nested levels.

Now, let's try doing this exercise again with the same function, implemented differently:

```go
func join(s1, s2 string, max int) (string, error) {
        if s1 == "" {
                return "", errors.New("s1 is empty")
        }
        if s2 == "" {
                return "", errors.New("s2 is empty")
        }
        concat, err := concatenate(s1, s2)
        if err != nil {
                return "", err
        }
        if len(concat) > max {
                return concat[:max], nil
        }
        return concat, nil
}

func concatenate(s1 string, s2 string) (string, error) {
        // ...
}
```

You should have probably noticed that building a mental model of this new version requires less cognitive load, despite doing the same job as before. Yet, here we maintain only two nested levels. As mentioned by Mat Ryer, a panelist on the Go Time podcast:

> Align the happy path to the left; you should quickly be able to scan down one column to see the expected execution flow.

It was difficult to distinguish the expected execution flow in the first version because of the nested `if` / `else` statements. Conversely, the second version requires scanning down one column to see the expected execution flow and the second column to see how the edge cases are handled as shown in the following figure:

Figure 2.1. To understand the expected execution flow, we just have to scan the happy path column.

```
func join(s1, s2 string, max int) (string, error) {
    if s1 == "" {
        return "", errors.New("s1 is empty")
    }
    if s2 == "" {
        return "", errors.New("s2 is empty")
    }
    concat, err := concatenate(s1, s2)
    if err != nil {
        return "", err
    }
    if len(concat) > max {
        return concat[:max], nil
    }
    return concat, nil
}
```

Happy path Error path & edge cases

In general, the more nested levels a function requires, the more complex it is to read and understand. Let’s see different applications of this rule to optimize our code for readability:

- When an `if` block returns in all cases, we should omit the `else` block. We shouldn't write this:

```
if foo() {
        // ...
        return true
} else {
        // ...
}
```

But instead, omit the `else` block like this:

```
if foo() {
        return true
}
// ...
```

With this new version, the code living previously in the `else` block is moved to the top level, making it easier to read.

- We can also follow this logic with a non-happy path:

```
if s != "" {
        // ...
} else {
        return errors.New("empty string")
}
```

Here, `s` being empty represents the non-happy path. Hence, we should flip the condition like so:

```
if s == "" { #1
        return errors.New("empty string")
}
// ...
```

This new version is easier to read as it kept the happy path on the left edge and reduced the number of blocks.

Writing readable code is an important challenge for every developer. Striving to reduce the number of nested blocks, align the happy path on the left, and return as early as possible are concrete means to improve a code's readability.

In the next section, we will discuss a common misuse in Go projects: init functions.

2.3 Misusing init functions

`init` functions are sometimes misused in Go applications. The potential consequences are poor error management or a code flow that is harder to understand. Let's refresh our minds about what is an init function. Then, we will see when it is or isn't recommended to use them.

2.3.1 Concepts

An init function is a function taking no arguments and returning no result (a `func()` function). When a package is initialized, all the constants and variables declarations in the package are evaluated. Then, the init functions are executed. Here is an example of a `main` package:

Listing 2.1. Package initialization

```go
package main

import "fmt"

var a = func() int {
        fmt.Println("var")
        return 0
}()

func init() {
        fmt.Println("init")
}

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

```
Executed first 1
Executed second 2
Executed last 3
```

Running this example would print the following output:

```
var
init
main
```

An init function is executed when a package is initialized. In the following example, we will define two packages, `main` and `redis` with `main` depending on `redis`:

Listing 2.2. main/main.go

```go
package main

```

```
3  import (
4          "fmt"
5
6          "redis"
7  )
8
9  func init() {
10         // ...
11 }
12
13 func main() {
14         err := redis.Store("foo", "bar")
15         // ...
16 }
```

Dependency on the redis package 1

Listing 2.3. redis/redis.go

```
1  package redis
2
3  // imports
4
5  func init() {
6          // ...
7  }
8
9  func Store(key, value string) error {
10         // ...
11 }
```

As `main` depends on `redis`, the `redis` package's init function will be executed first, followed by the init of the `main` package, then the `main` function itself, as shown in the following figure:

Figure 2.2. The init function of the `redis` package is executed first, then the init function of `main`, then the `main` function.

We can define multiple init functions per package. In this case, the execution order of the init function inside a package is based on the source files' alphabetical order. For example, if a package contains an `a.go` and a `b.go` file and both have an init function, the `a.go` init function will be executed first. We shouldn't rely on the ordering of init functions within a package. Indeed, it can be dangerous as source files can be renamed, potentially impacting the execution order.

We can also define multiple init functions within the same source file. For example, this code is perfectly valid:

Listing 2.4. Multiple init functions per source file

```
package main

import "fmt"

func init() {
        fmt.Println("init 1")
}

func init() {
        fmt.Println("init 2")
}
```

```

func main() {
}
```

First init function 1
Second init function 2

The first init function executed is the first one in the source order:

```
init 1
init 2
```

We can also use `init` functions for side effects. In the next example, we will define a `main` package that doesn't have a strong dependency on `foo` (e.g., no direct use of a public function). However, it requires the `foo` package to be initialized. We can do it using the `_` operator this way:

Listing 2.5. main/main.go

```go
package main

import (
        "fmt"

        _ "foo"
)

func main() {
        // ...
}
```

Imports foo for side effects 1

In this case, the `foo` package will be initialized before `main`. Hence, the init functions of `foo` will be executed.

Another aspect of an `init` function is that it can't be invoked directly:

Listing 2.6. An invalid example of an init function invoked from another function

```
1 package main
2
3 func init() {}
4
5 func main() {
6         init()
7 }
```

Invalid reference 1

This code produces the following compilation error:

```
1 $ go build .
2 ./main.go:6:2: undefined: init
```

Now that we refreshed our minds about how init functions work, let's see when we should use or not use them.

2.3.2 When to use init functions?

First, let's see an example where using an init function can be considered inappropriate: to hold a database connection pool.

In the init function, we will open a database using `sql.Open`. We will make this database a global variable that functions can later use.

```
1  var db *sql.DB
2
3  func init() {
4          dataSourceName := os.Getenv("MYSQL_DATA_SOURCE_NAME")
5          d, err := sql.Open("mysql", dataSourceName)
6          if err != nil {
7                  log.Panic(err)
8          }
9          err = d.Ping()
10         if err != nil {
11                 log.Panic(err)
12         }
13         db = d
14 }
```

Environement variable 1
Assign the DB connection to the global db variable 2

In this example, we open the database, check whether we can ping it, and then assign it to the global variable. What should we think about our implementation? Let's describe three main downsides.

First, error management in an init function is limited. Indeed, as an init function doesn't return any error, one of the only ways to signal an error is to panic, leading the application to be stopped. In our example, it might be ok to stop the application anyway if opening the database fails. However, it shouldn't be necessarily up to the package itself to decide whether to stop the application. Perhaps a caller might have preferred implementing a retry or using a fallback mechanism. Opening the database within an init function prevents client packages from implementing their error handling logic.

Another important downside is related to testing. If we add tests to this file, the init function will be executed before running the test cases, which isn't necessarily what we want. For example, if we add unit tests on a utility function that doesn't require this connection to be created. Therefore, it would complicate writing unit tests.

The last downside is that it requires assigning the database connection pool to a global variable. Global variables have some severe drawbacks, for example:

- Any functions can alter them within the package
- It can also make unit tests more complicated as a function that would depend on it wouldn't be isolated anymore

In most cases, we should favor encapsulating a variable rather than keeping it global.

For these reasons, the previous initialization should probably be handled as part of a plain old function like so:

```
func createClient(dataSourceName string) (*sql.DB, error) {
        db, err := sql.Open("mysql", dataSourceName)
        if err != nil {
                return nil, err
        }
        if err = db.Ping(); err != nil {
                return nil, err
        }
        return db, nil
}
```

Accept a data source name and return an *sql.DB and an error 1
Return an error 2

Using this function, we tackled the main downsides discussed:

- The responsibility of error handling is left up to the caller
- It's possible to create an integration test to check this function works
- The connection pool is encapsulated within the function

However, shall we avoid init functions at all costs? Not really. There are still use cases where init functions can be pretty useful. For example, the official Go blog (which is implemented in Go) uses an init function to set up the static HTTP configuration[3]:

```
func init() {
        redirect := func(w http.ResponseWriter, r
*http.Request) {
                http.Redirect(w, r, "/", http.StatusFound)
        }
        http.HandleFunc("/blog", redirect)
        http.HandleFunc("/blog/", redirect)

        static := http.FileServer(http.Dir("static"))
        http.Handle("/favicon.ico", static)
        http.Handle("/fonts.css", static)
        http.Handle("/fonts/", static)

        http.Handle("/lib/godoc/",
http.StripPrefix("/lib/godoc/",
                http.HandlerFunc(staticHandler)))
}
```

In this example, the init function cannot fail (`http.HandleFunc` can panic, but only if the handler is nil, which isn't the case here). Meanwhile, there's no need to create any global variable, and it will not impact possible unit tests. Therefore, this example is a good one where init functions can be helpful.

In summary, we have seen that init functions may lead to some issues:

- Limiting error management
- It can complicate how to implement tests (e.g., external dependency to be set up, which may not be necessary for the scope of unit tests)
- If the initialization requires to set a state, it has to be done through global variables

We should be cautious with `init` functions. They can be helpful in some situations, such as defining static configuration, as we have seen in this section. Otherwise, and in most cases, we should handle initializations through ad-hoc functions.

2.4 Overusing getters and setters

In programming, data encapsulation refers to hiding the values or state of an object. Getters and setters are means to enable encapsulation by providing exported methods on top of unexported object fields.

In Go, there is no automatic support, as we can see in some languages. It is also neither considered mandatory nor idiomatic to use getters and setters to access struct fields. For example, the standard library implements structs where some fields are accessible directly, such as the `time.Timer` struct:

```
timer := time.NewTimer(time.Second)
<-timer.C
```

C is a ←chan Time field 1

Though it's not recommended, we could even modify `C` directly (but we wouldn't receive events anymore). However, this example illustrates that even

the standard library doesn't enforce using getters and/or setters, even when we shouldn't modify a field.

On the other hand, using getters and setters present some advantages, including:

- It encapsulates a behavior associated with getting or setting a field, allowing new functionality to be added later. For example: validating a field, returning a computed value, or wrapping the access to a field around a mutex.
- It hides the internal representation, giving us more flexibility in what we expose.
- It provides a debugging interception point for when the property changes at runtime, making debugging easier.

If we fall into these cases or foresee a possible use case while guaranteeing forward compatibility, using getters and setters can bring some value. If we do use them, given a field called `balance`, the naming conventions we should follow are:

- The getter method should be named `Balance` (not `GetBalance`)
- The setter method should be named `SetBalance`

```
currentBalance := customer.Balance()
if currentBalance < 0 {
        customer.SetBalance(0)
}
```

Getter 1
Setter 2

In summary, we shouldn't overwhelm our code with getters and setters methods on structs if they don't bring any value. We should be pragmatic and strive to find the right balance between efficiency and following idioms that are sometimes considered indisputable in other programming paradigms. Remember that Go is a unique language designed for many characteristics, including simplicity. However, if we find a need for getters and setters or foresee a future need while guaranteeing forward compatibility, there's nothing wrong with using them.

The next section will discuss the problems of overusing interfaces.

2.5 Interface pollution

Interfaces are one of the cornerstones of the Go language when designing and structuring our code. However, like many tools or concepts, abusing it is generally not a good idea. Interface pollution is about overwhelming our code with unnecessary abstractions making it harder to understand. It's a common mistake made by developers coming from another language with different habits. Before delving into the topic, let's refresh our minds about Go's interfaces. Then, we will see when it's appropriate to use interfaces and when it may be considered pollution.

2.5.1 Concepts

An interface provides a way to specify the behavior of an object. Interfaces are used to create common abstractions that multiple objects can implement. What makes Go interfaces so different is that they are satisfied implicitly. There is no explicit keyword like `implements` to mark that an object X implements interface Y.

To understand what makes interfaces so powerful, we will dig into two popular ones from the standard library: `io.Reader` and `io.Writer`.

The `io` package provides abstractions for I/O primitives. Among these abstractions, `io.Reader` relates to reading data from a data source and `io.Writer` to writing data to a target as represented in the following figure:

Figure 2.3. `io.Reader` reads from a data source and fills a byte slice, whereas `io.Writer` writes to a target from a byte slice.

The `io.Reader` contains a single `Read` method:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

Custom implementations of the `io.Reader` interface should accept a slice of bytes, fill it with its data and return either the number of bytes read or an error.

On the other hand, `io.Writer` defines a single method: `Write`.

```
type Writer interface {
        Write(p []byte) (n int, err error)
}
```

Custom implementations of `io.Writer` should write to a target the data coming from a slice and return either the number of bytes written or an error.

Therefore, both interfaces provide fundamental abstractions:

- `io.Reader` to read data from a source
- `io.Writer` to write data to a target

What is the rationale for having these two interfaces in the language? What is the point of creating these abstractions?

Let's assume we must implement a function that should copy one file content to another. We could create a specific function that would take as input two `*os.File`. Or, we can choose to create a more generic function using `io.Reader` and `io.Writer` abstractions:

```
func copySourceToDest(source io.Reader, dest io.Writer) error {
        // ...
}
```

This function would work with `*os.File` parameters (as `*os.File` implements both `io.Reader` and `io.Writer`) and any other type that would implement these interfaces. For example, we could create our own `io.Writer` that would write to a database, and the code would remain the same. It increases the genericity of the function; hence, its reusability.

Furthermore, writing a unit test for this function is easier as instead of having to handle files, we can use the `strings` and `bytes` packages that provide implementations:

```go
func TestCopySourceToDest(t *testing.T) {
        const input = "foo"
        source := strings.NewReader(input)
        dest := bytes.NewBuffer(make([]byte, 0))

        err := copySourceToDest(source, dest)
        if err != nil {
                t.FailNow()
        }

        got := dest.String()
        if got != input {
                t.Errorf("expected: %s, got: %s", input, got)
        }
}
```

Create an io.Reader 1
Create an io.Writer 2
Call copySourceToDest from a *strings.Reader and a *bytes.Buffer 3

`source` is a `*strings.Reader` whereas `dest` a `*bytes.Buffer`. Here, we test the behavior of `copySourceToDest` without creating any files.

Moreover, while designing interfaces, the granularity (how many methods the interface contains) is something to keep in mind. A known proverb in Go

relates to how big an interface should be:

The bigger the interface, the weaker the abstraction.[4]

Indeed, adding methods to an interface can decrease its level of reusability. `io.Reader` and `io.Writer` are powerful abstractions because they cannot get any simpler. Furthermore, fine-grained interfaces can also be combined to create higher-level abstractions. This is the case with `io.ReadWriter` that combines the reader and writer behaviors:

```
type ReadWriter interface {
        Reader
        Writer
}
```

Note

As quoted by Einstein, "Everything should be made as simple as possible, but no simpler". Applied to interfaces, it denotes that finding the perfect granularity for an interface isn't necessarily a straightforward question.

Let's now discuss common cases when interfaces are recommended.

2.5.2 When to use interfaces?

When should we create interfaces in Go? Let's delve into three concrete use cases where interfaces are usually considered to bring value. Note that the goal isn't to be exhaustive as the more cases we add, the more they would depend on the context. However, these three cases should give us a general idea:

- Common behavior
- Decoupling
- Restricting behavior

Common behavior

The first option we will discuss is to use interfaces when multiple types implement a common behavior. In such a case, we can factor out the behavior inside an interface.

If we look at the standard library, we can find many examples of such a use case. For example, sorting a collection can be factored out via three methods:

- Retrieving the number of elements in the collection
- Reporting whether one element must be sorted before another
- Swapping two elements

Hence, the following interface was added to the `sort` package:

```
type Interface interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}
```

Number of elements 1
Check two elements 2
Swap two elements 3

This interface has a strong potential for reusability as it encompasses the common behavior to sort any collection that would be index-based.

Throughout the `sort` package, we can find dozens of implementations. If at some point we compute a collection of integers, for example, and we need to sort it, are we necessarily interested in the implementation type? Is it important whether the sorting algorithm is a merge sort or a quicksort, for example? In many cases, we don't care. Hence, the sorting behavior can be abstracted, and we could depend on the `sort.Interface`.

Finding the right abstraction to factor out a behavior can also bring many benefits. For example, the `sort` package provides utility functions that also rely on `sort.Interface` such as checking whether a collection is already sorted:

```
func IsSorted(data Interface) bool {
        n := data.Len()
        for i := n - 1; i > 0; i-- {
                if data.Less(i, i-1) {
                        return false
                }
        }
        return true
}
```

As `sort.Interface` is the right level of abstraction, it makes it highly valuable.

Let's now see another main use case when using interfaces.

Decoupling

Another important use case is about decoupling our code from an implementation. If we rely on an abstraction instead of a concrete implementation, the implementation itself can be replaceable with another without even having to change our code; this is the Liskov Substitition Principle (the L in SOLID).

One benefit of decoupling can be related to unit testing, for example. Let's assume we have to implement a `CreateNewCustomer` method that creates a new customer and stores it. We can decide to rely on the concrete implementation directly (let's say a `mysql.Store` struct):

```
type CustomerService struct {
        store mysql.Store
}

func (cs CustomerService) CreateNewCustomer(id string) error {
        customer := Customer{id: id}
        return cs.store.StoreCustomer(customer)
}
```

Depends on the concrete implementation 1

Now, what if we want to test this method? As `customerService` relies on the actual implementation to store a `Customer`, we are obliged to test it through integration tests which require spinning up a MySQL instance (unless we use an alternative technique such as `go-sqlmock` [5], but this isn't the scope of this section). Though integration tests are helpful, it's not always what we want to do. To give us more flexibility, we should decouple `CustomerService` from the actual implementation, which can be done via an interface like so:

```
type customerStorer interface {
        StoreCustomer(Customer) error
}
```

```go

type CustomerService struct {
        storer customerStorer
}

func (cs CustomerService) CreateNewCustomer(id string) error {
        customer := Customer{id: id}
        return cs.storer.StoreCustomer(customer)
}
```

Create a storage abstraction 1
Decouple Customer service from the actual implementation 2

As storing a customer is now done via an interface, it gives us more flexibility in how we want to test the method:

- Either with the concrete implementation via integration tests
- With a mock (or any kind of test double) via unit tests
- Or both

Let's now discuss another use case to restrict a behavior.

Restricting behavior

The last use case we will discuss can be pretty counter-intuitive at first sight. It's about restricting a type to a specific behavior.

Let's imagine we have implemented a custom configuration package to deal with dynamic configuration. We created a specific container for `int` configurations via an `IntConfig` struct that also exposes two methods, `Get` and `Set`:

```go
type IntConfig struct {
        // ...
}

```

```
func (c *IntConfig) Get() int {
        // Retrieve configuration
}

func (c *IntConfig) Set(value int) {
        // Update configuration
}
```

Now, suppose we receive an `IntConfig` that holds some specific configuration, such as a threshold. Yet, in our code, we are only interested in retrieving the config value, and we want to prevent updating it. How could we enforce that, semantically, this configuration is a read-only one if we don't want to change our configuration package? By creating an abstraction that would restrict the behavior to retrieving a config value solely:

```
type intConfigGetter interface {
        Get() int
}
```

Then, in our code, we could rely on `intConfigGetter` instead of the concrete implementation:

```
type Foo struct {
        threshold intConfigGetter
}

func NewFoo(threshold intConfigGetter) Foo {
        return Foo{threshold: threshold}
}

func (f Foo) Bar()  {
        threshold := f.threshold.Get()
        // ...
}
```

Inject the configuration getter 1

Read configuration 2

In this example, the configuration getter is injected into the `NewFoo` factory method. It doesn't even impact a client of this function as it can still pass an `IntConfig` struct because it implements `intConfigGetter`. Then, we can only read the configuration in the `Bar` method, not modify it.

Therefore, we can also use interfaces to restrict a type to a specific behavior for various reasons, such as semantics enforcement.

We have seen three potential use cases when interfaces are generally considered as bringing value: factoring out a common behavior, creating some decoupling, and restricting a type to a certain behavior. Again, this list isn't exhaustive, but it should give us a general understanding of when interfaces are helpful in Go.

Now, let's finish this section and discuss the problems with interface pollution.

2.5.3 Interface pollution

It's fairly common to see interfaces being overused in Go projects. Perhaps the developer's background was C# or Java, and he found it pretty natural to create interfaces before concrete types. However, this isn't how things should work in Go.

As we discussed, interfaces are made to create abstractions. And the main caveat when programming meets abstractions is remembering that abstractions **should be discovered, not created**. What does it mean?

It means we shouldn’t start by creating abstractions in our code if there is no immediate reason for it. We shouldn’t design with interfaces but wait for a concrete need. Said differently, we should create an interface when we need it, not when we foresee that we could need it.

What’s the main problem if we overuse interfaces? The answer is that it makes the code flow more complex. Adding a useless level of indirection doesn’t bring any value; it creates a useless abstraction making the code more difficult to read, understand, and reason about.

If we don’t have a strong reason for adding an interface and it’s unclear how an interface makes a code better, we should challenge this interface’s purpose. Why not call the implementation directly?

> **Note**
>
> We can also note a performance overhead when calling a method through an interface. It requires a lookup in a hashtable data structure to find the concrete type it’s pointing to. Yet, this isn’t an issue in many contexts as this overhead is minimal.

In summary, we should be cautious when creating abstractions in our code. Again, abstractions should be discovered, not created. It’s pretty frequent for us, software developers, to overengineer our code by trying to guess what would be the perfect level of abstraction based on what we think we might need later on. This process should be avoided as, in most cases, it pollutes our code with unnecessary abstractions, making it more complex to read. Let’s not try to solve a problem abstractly but solve what has to be solved now. Last but not least, if it’s unclear how an interface makes the code better, we should probably consider removing it to make our code simpler.

> Don’t design with interfaces, discover them.
>
> -- Rob Pike

The following section will discuss a common interface mistake, creating interfaces on the producer side.

2.6 Interface on the producer side

We have seen in the previous section when interfaces are considered as valuable. Yet, one question that Go developers quite often misunderstand is the following: where should an interface live?

Before delving into the topic, let’s make sure the terms that we will use throughout this section are clear:

- Producer side: an interface defined in the same package as the concrete implementation

Figure 2.4. The interface is defined alongside the concrete implementation.

- Consumer side: an interface defined in an external package, where it's used

Figure 2.5. The interface is defined where it's being used.

It's pretty common to see developers creating interfaces on the producer side, alongside the concrete implementation. This design is perhaps a habit from

developers having a C# or a Java background. However, in Go, this is, in most cases, not what we should do.

Let's discuss the following example. We create a specific package to store and retrieve customer data. Meanwhile, we decide, still in the same package, that all the calls will have to go through the following interface:

```
package store

type CustomerStorage interface {
        StoreCustomer(customer Customer) error
        GetCustomer(id string) (Customer, error)
        UpdateCustomer(customer Customer) error
        GetAllCustomers() ([]Customer, error)
        GetCustomersWithoutContract() ([]Customer, error)
        GetCustomersWithNegativeBalance() ([]Customer, error)
}
```

We might think we have some excellent reasons to create and expose this interface on the producer side. Perhaps it's a good way to decouple the client code from the actual implementation? Perhaps we foresee that it will help clients in creating test doubles? However, this isn't a best practice in Go.

As we mentioned, interfaces are satisfied implicitly in Go which tends to be a game-changer compared to languages with an explicit implementation. In most cases, the approach to follow is similar to what we described in the previous section: abstractions should be discovered, not created. It means that it's not up to the producer to force a given abstraction for all the clients. Instead, it's up to the client to decide whether he needs some form of abstraction and then determine what's the best abstraction level for his needs.

In the previous example, perhaps one client won't be interested in decoupling his code. Maybe another client wants to decouple his code but is only

interested in the `GetAllCustomers` method. In this case, it can create an interface with a single method, referencing the `Customer` struct from the external package:

```
package client

type customersGetter interface {
        GetAllCustomers() ([]store.Customer, error)
}
```

From a package organization, it would result in the following:

Figure 2.6. The `client` package defines the abstraction it needs by creating its own interface.

A couple of things to note:

- As the `customersGetter` interface is only used in the `client` package, it can remain unexported.
- Visually, it looks like circular dependencies. However, there's no dependency from `store` to `client` as the interface is

satisfied implicitly. This is why such an approach isn't always possible in languages with an explicit implementation.

The main point is that the `client` package can now define the most accurate abstraction for its need (here, only one method). It relates to the concept of the Interface-Segregation Principle (the I in SOLID), which states that no client should be forced to depend on methods it doesn't use.

Therefore, in this case, the best approach is to expose the concrete implementation on the producer side and let the clients decide how to use it and whether an abstraction is needed.

For the sake of completeness, let's mention that this approach, interfaces on the producer side, is sometimes used in the standard library. For example, the `encoding` package defines interfaces implemented by other sub-packages such as `encoding/json` or `encoding/binary`. So is the `encoding` package wrong about that? Definitely not. In this case, the abstractions defined in the `encoding` package are used across the standard library, and the language designers knew that creating this abstraction upfront was valuable. We are back to the discussion in the previous section: don't create an abstraction if you think it might be helpful in an imaginary future, or at least if you can't prove this abstraction is valid.

Therefore, an interface should live on the consumer side in most cases. However, in particular contexts, for example, when we know (not foresee) that an abstraction will be helpful for consumers, we may want to have it on the producer side. If we do, we should strive to keep it as minimal as possible, increasing its reusability potential and making it more easily composable.

Let's continue the discussion about interfaces in the context of functions signature.

2.7 Returning interfaces

While designing a function signature, we may have to either return an interface or a concrete implementation. Let's understand why returning an interface is, in many cases, considered a bad practice in Go.

We just presented why interfaces live in general on the consumer side. Let's see in Figure 2.7. There's a dependency from store package to client package.figure 2.7 what would happen dependency-wise if a function would return an interface instead of a struct. We will see that it leads to issues.

We will consider two packages:

- `client` containing a `Store` interface
- `store` containing an implementation of `Store`

Figure 2.7. There's a dependency from `store` package to `client` package.

In the `store` package, we define an `InMemoryStore` struct that implements the `Store` interface. Meanwhile, we create a `NewInMemoryStore` function to return a `Store` interface. There's a dependency from the implementation package to the client package in this design, and it may already sound a bit

odd. For example, the client package can't call the `NewInMemoryStore` function anymore; otherwise, there would be a cyclic dependency. A possible solution could be to call this function from another package and inject a `Store` implementation to `client`. However, being obliged to do that means that the design should be challenged.

Furthermore, what happens if another client uses the `InMemoryStore` struct? In that case, perhaps we would like to move the `Store` interface to another package? Or perhaps back to the implementation package, but we discussed why, in most cases, it isn't a best practice. So again, it looks like a code smell.

Hence, in general, returning an interface restricts flexibility as we force all the clients to use one particular type of abstraction.

In most cases, we can get inspiration from Postel's law:

Be conservative in what you do, be liberal in what you accept from others.[6]

If we apply this idiom to Go, it means:

- Returning structs instead of interfaces
- Accepting interfaces if possible

Of course, there are some exceptions. As software engineers, we are pretty familiar with the fact that rules are never really true in 100% of the cases. The most relevant one is the `error` type, an interface returned by many functions. We can also examine another exception in the standard library with the `io` package:

```
func LimitReader(r Reader, n int64) Reader {
        return &LimitedReader{r, n}
}
```

Here, the function returns an exported struct: `io.LimitedReader`. However, the function signature is an interface: `io.Reader`. So what's the rationale for breaking the rule we discussed so far? The `io.Reader` is an upfront abstraction. It's not one defined by clients, but it's a forced one as the language designers knew in advance that this level of abstraction would be helpful (for example, in terms of reusability and composability).

All in all, in most cases, we shouldn't return interfaces but concrete implementations. Otherwise, it can make our design more complex due to package dependencies and restrict flexibility as all the clients would have to rely on the same abstraction. Again, the conclusion is similar to the previous sections: if we know (not foresee) that an abstraction will be helpful for clients, we can consider returning an interface. Otherwise, we shouldn't force abstractions, they should be discovered by clients. If a client needs to abstract an implementation for whatever reason, it can still do it on his side.

In the next section, we will discuss a common mistake related to using `any`.

2.8 `any` says nothing[7]

In Go, an interface type that specifies zero methods is known as the empty interface: `interface{}`. From Go 1.18, the predeclared type `any` has became an alias for an empty interface. Hence, all the `interface{}` occurences can be replaced by `any`. However, in many cases, `any` can be considered as an over-generalization. Let's first remind us the core concepts, and then we will discuss the potential problems.

An `any` type can hold any value type:

```go
func main() {
        var i any

        i = 42
        i = "foo"
        i = struct {
                s string
        }{
                s: "bar",
        }
        i = f

        _ = i
}

func f() {}
```

An int 1
A string 2
A struct 3
A function 4
Assignment to the blank identifier so that the example compiles 5

In assigning a value to an `any` type, we lose all type info which requires a type assertion to get anything useful out of the `i` variable.

Let's now see a concrete example where using `any` isn't accurate. We will implement a `Store` struct and the skeleton of two methods: `Get` and `Set`. These methods will be used to store different struct types: `Customer` and `Contract`:

```go
package store

type Customer struct{
        // Some fields
}
```

```go
type Contract struct{
        // Some fields
}

type Store struct{}

func (s *Store) Get(id string) (any, error) {
        // ...
}

func (s *Store) Set(id string, v any) error {
        // ...
}
```

Return any 1
Accept any 2

Though there is nothing wrong with `Store` compilation-wise, we should take a minute to think about the method signatures. As we accept and return `any` arguments, the methods lack of expressiveness. If future developers have to use the `Store` struct, they will probably have to dig into the documentation or read the code to understand how to use these methods. Hence, accepting or returning an `any` type doesn't convey meaningful information.

Also, as there is no safeguard at compile-time, there is nothing that prevents a caller from calling these methods with whatever data type, such as an `int`:

```go
s := store.Store{}
s.Set("foo", 42)
```

By using `any`, we lose some of the benefits of Go being a statically-typed language. Instead, we should avoid `any` types and make our signatures explicit as much as possible. Regarding our example, it could mean duplicating the get and set methods per type:

```go
func (s *Store) GetContract(id string) (Contract, error) {
        // ...
}

func (s *Store) SetContract(id string, contract Contract)
error {
        // ...
}

func (s *Store) GetCustomer(id string) (Customer, error) {
        // ...
}

func (s *Store) SetCustomer(id string, customer Customer)
error {
        // ...
}
```

In this version, the methods are expressive, reducing the risk of incomprehension. Having more methods isn't necessarily a problem as clients can also create their own abstraction using an interface. For example, if a client is interested only in the `Contract` methods:

```go
type ContractStorer interface {
        GetContract(id string) (store.Contract, error)
        SetContract(id string, contract store.Contract) error
}
```

So what are the cases when `any` is helpful? Let's take a look at the standard library and see two examples where functions or methods accept `any` arguments.

The first example is in `encoding/json` package. As we can marshal any type, the `Marshal` function accepts an `any` argument:

```go
func Marshal(v any) ([]byte, error) {
        // ...
}
```

Another example in the `database/sql` package. If the query is parameterized (e.g., `SELECT * FROM FOO WHERE id = ?`), the parameters could be of any kind. Hence, it also uses `any` arguments:

```go
func (c *Conn) QueryContext(ctx context.Context, query string,
        args ...any) (*Rows, error) {
        // ...
}
```

In summary, `any` can be helpful if there is a genuine need for accepting or returning any possible type. For example, when it comes to marshaling or formatting. In general, we should avoid over-generalizing at all costs the code we write. Perhaps, a little bit of duplicated code can occasionally be better if it improves other aspects such as code expressiveness.

The following section will discuss another type of abstraction: generics.

2.9 Being confused about when to use generics

As of Go 1.18, generics were added to the language. In a nutshell, it allows writing code with types that can be specified later and instantiated when needed. However, it can be pretty easy to be confused about when to use generics and when not to. Throughout this section, we will describe the concept of generics in Go and then delve into common use and misuses.

2.9.1 Concepts

Consider the following function that extracts all the keys from a `map[string]int` type:

```
1 func getKeys(m map[string]int) []string {
2         var keys []string
3         for k := range m {
4                 keys = append(keys, k)
5         }
6         return keys
7 }
```

What if we would like to use a similar feature for another map type such as a `map[int]string`? Before generics, Go developers had a few options: using code generation, reflection, or duplicating code.

For example, we could write two functions, one for each map type, or even try to extend `getKeys` to accept different map types:

```
1  func getKeys(m any) ([]any, error) {
2          switch t := m.(type) {
3          default:
4                  return nil, fmt.Errorf("unknown type: %T", t)
5          case map[string]int:
6                  var keys []any
7                  for k := range t {
8                          keys = append(keys, k)
9                  }
10                 return keys, nil
11         case map[int]string:
12                 // ...
13         }
14 }
```

Accept and return any arguments 1
Handle runtime errors if a type isn't implemented yet 2

We can start noticing a few issues:

- First, it increases boilerplate code. Indeed, whenever we want to add a case, it will require duplicating the `range` loop.
- Meanwhile, the function now accepts an `any` type, which means we are losing some of the benefits of Go being a typed language. Indeed, checking whether a type is supported is done at runtime instead of compile-time. Hence, we also need to return an error if the provided type is unknown.
- Last but not least, as the key type can be either `int` or `string`, we are obliged to return a slice of `any` types to factor out key types. This approach increases the effort on the caller-side as the client may also have to perform a type check of the keys or extra conversion.

Thanks to generics, we can now refactor this code using type parameters.

Type parameters are generic types we can use with functions and types. For example, the following function accepts a type parameter:

```
func foo[T any](t T) {
        // ...
}
```

T is a type parameter 1

When calling `foo`, we will pass a type argument of any type. Supplying a type argument is called instantiation, and the work is done at compile time which keeps type safety as part of the core language features and avoids runtime overheads.

Let's get back to the `getKeys` function and use type parameters to write a generic version that would accept any kind of map:

```
1 func getKeys[K comparable, V any](m map[K]V) []K {
2         var keys []K
3         for k := range m {
4                 keys = append(keys, k)
5         }
6         return keys
7 }
```

The keys are comparable, whereas values are of any type 1
Create the keys slice 2

To handle the map, we defined two kinds of type parameters. First, the values can be of any type: `V any`. However, in Go, the map keys can't be of any type. For example, we cannot use slices:

```
var m map[[]byte]int
```

This code leads to a compilation error: `invalid map key type []byte`. Therefore, instead of accepting any key type, we are obliged to restrict type arguments so that the key type meets specific requirements. Here, being comparable (we can use `==` or `!=`). Hence, we defined `K` as `comparable` instead of `any`.

Restricting type arguments to match specific requirements is called a constraint. A constraint is an interface type that can contain:

- A set of behaviors (methods)

- But also arbitrary type

Let's see a concrete example for the latter. Imagine we don't want to accept any `comparable` type for map key type. For instance, we would like to restrict it to either `int` or `string` types. We can define a custom constraint this way:

```
type customConstraint interface {
        ~int | ~string
}

func getKeys[K customConstraint, V any](m map[K]V) []K {
        // Same implementation
}
```

Define a custom type that will restrict types to int and string 1
Change the type parameter K to be custom 2

First, we define a `customConstraint` interface to restrict the types to be either `int` or `string` using the union operator `|` (we will discuss the use of `~` a bit later). Then, `K` is now a `customConstraint` instead of a `comparable` as before.

Now, the signature of `getKeys` enforces that we can call it with a map of any value type, but the key type has to be an `int` or a `string`. For example, on the caller-side:

```
m = map[string]int{
        "one":   1,
        "two":   2,
        "three": 3,
}
keys := getKeys(m)
```

Note that Go can infer that `getKeys` is called with a `string` type argument. The call above was equivalent to this:

```
keys := getKeys[string](m)
```

Note

Using this constraint will restrict type arguments to custom types like this one:

```
type customInt int

func (i customInt) String() string {
        return strconv.Itoa(int(i))
}
```

As `customInt` is an `int` and implements the `String() string` method, the `customInt` type satisfies the constraint defined.

However, if we change the constraint to contain an `int` instead of an `~int`, using `customInt` would lead to a compilation error because the `int` type doesn't implement `String() string`.

What's the difference between a constraint using `~int` or `int`? Using `int` restricts it to that type, whereas `~int` restricts all the types whose underlying type is an `int`.

To illustrate it, let's imagine a constraint where we would like to restrict a type to any `int` type implementing the `String()`

`string` method:

```
type customConstraint interface {
        ~int
        String() string
}
```

Let's also note the `constraints` package contains a set of common constraints such as `Signed` that includes all the signed integer types. Let's ensure that a constraint doesn't already exist in this package before creating a new one.

So far, we have discussed examples using generics for functions. However, we can also use generics with data structures.

For example, we will create a linked list containing values of any type. Meanwhile, we will write an `Add` method to append a node:

```
type Node[T any] struct {
        Val  T
        next *Node[T]
}

func (n *Node[T]) Add(next *Node[T]) {
        n.next = next
}
```

Use type parameter 1
Instantiate type receiver 2

We use type parameters to define `T` and use both fields in `Node`. Regarding the method, the receiver is instantiated. Indeed, because `Node` is generic, it

has to follow also the type parameter defined.

One last thing to note about type parameters is that they can't be used with method arguments, only with function arguments or method receivers. For example, the following method wouldn't compile:

```
type Foo struct {}

func (Foo) bar[T any](t T) {}
```

```
./main.go:29:15: methods cannot have type parameters
```

If we want to use generics with methods, it's the receiver that needs to be a type parameter.

Now, let's delve into concrete cases where we should and shouldn't use generics.

2.9.2 Common uses and misuses

So when are generics useful? Let's discuss a couple of common uses where generics are recommended:

- Data structures. For example, we can use generics to factor out the element type if we implement a binary tree, a linked list, or a heap.

- Functions working with slices, maps, and channels of any type. For example, a function to merge two channels would

work with any channel type. Hence, we could use type parameters to factor out the channel type:

```
func merge[T any](ch1, ch2 <-chan T) <-chan T {
        // ...
}
```

- Factoring out behaviors instead of types. For example, the `sort` package contains a `sort.Interface` interface with three methods:

```
type Interface interface {
        Len() int
        Less(i, j int) bool
        Swap(i, j int)
}
```

This interface is used by different functions such as `sort.Ints` or `sort.Float64s`. Using type parameters, we could factor out the sorting behavior. For example, by defining a struct holding a slice and a comparison function:

```
type SliceFn[T any] struct { #1
        S       []T
        Compare func(T, T) bool #2
}

func (s SliceFn[T]) Len() int           { return len(s.S) }
func (s SliceFn[T]) Less(i, j int) bool { return
s.Compare(s.S[i], s.S[j]) }
func (s SliceFn[T]) Swap(i, j int)      { s.S[i], s.S[j] =
s.S[j], s.S[i] }
```

Then, as `SliceFn` struct implements `sort.Interface`, we can sort the provided slice using the `sort.Sort(sort.Interface)` function:

```
s := SliceFn[int]{
        S: []int{3, 2, 1},
        Compare: func(a, b int) bool {
                return a < b
        },
}
sort.Sort(s)
fmt.Println(s.S)
```

```
[1 2 3]
```

In this example, factoring out a behavior allowed us to avoid creating one function per type.

Conversely, when is it recommended not to use generics?

- When just calling a method of the type argument. For example, consider a function that receives an `io.Writer` and call the `Write` method:

```
func foo[T io.Writer](w T) {
        b := getBytes()
        _, _ = w.Write(b)
}
```

In this case, using generics wouldn't bring any value whatsoever, and we should make the `w` argument an

`io.Writer` directly.

- When it makes our code more complex. Generics are never mandatory, and as Go developers, we have been able to live without them for more than a decade. If writing generic functions or structures we figure out that it doesn't make our code clearer, we should probably reconsider our decision for this particular use case.

Though generics can be very helpful in particular conditions, we should be cautious about when to use them and not use them. In general, when we want to answer when not to use generics, we can find similarities with when not to use interfaces. Indeed, generics introduce a form of abstraction, and we have to remember that unnecessary abstractions introduce complexity. Again, let's not pollute our code with needless abstractions, and let's focus on solving concrete problems for now. It means that we shouldn't use type parameters prematurely. Let's wait until we are about to write boilerplate code to consider using generics.

In the following sections, we will discuss the possible problems while using type embedding.

2.10 Not being aware of the possible problems with type embedding

When creating a struct, Go offers the option to embed types. Yet, this may sometimes lead to unexpected behaviors if we don't understand all the implications of type embedding. Throughout this section, let's remind us how to embed types, what it brings, and the possible issues.

In Go, a struct field is called embedded if it's declared without a name. For example:

```
type Foo struct {
        Bar
}

type Bar struct {
        Baz int
}
```

Embedded field 1

In the `Foo` struct, the `Bar` type is declared without an associated name; hence it's an embedded field.

Embedding is used to *promote* the fields and methods of an embedded type. Here, as `Bar` contains a `Baz` field, it becomes promoted into `Foo`:

Figure 2.8. **`baz`** **is promoted, hence accessible directly from** **`S`**

Therefore, `Baz` becomes available from `Foo`:

```
foo := Foo{}
foo.Baz = 42
```

Note that `Baz` is available from two different paths. Either from the promoted one using `Foo.Baz` but also from the nominal one via `Bar`: `Foo.Bar.Baz`. Both relate to the same field.

> **Note**
>
> Yet, the scope of this section is related to embedded fields in structs solely.
>
> Embedding is also used within interfaces to compose an interface with others. For example, `io.ReadWriter`, is composed of an `io.Reader` and an `io.Writer`:
>
> ```
> type ReadWriter interface {
>  Reader
>  Writer
> }
> ```

Now that we remind ourselves what embedded types are, let's see an example of a wrong usage. We will implement a struct that will hold some in-memory data, and we want to protect it against concurrent accesses using a mutex:

```
type InMem struct {
        sync.Mutex
        m map[string]int
}

func New() *InMem {
        return &InMem{m: make(map[string]int)}
}
```

Embedded field 1

We have decided to make the map unexported so that clients can't interact with it directly but via exported methods. Meanwhile, the mutex field is embedded. Therefore, we can implement a `Get` method this way:

```
1 func (i *InMem) Get(key string) (int, bool) {
2         i.Lock()
3         v, contains := i.m[key]
4         i.Unlock()
5         return v, contains
6 }
```

Access the Lock directly 1
The same goes for the Unlock method 2

As the mutex is embedded, we can directly access the `Lock` and `Unlock` methods from the `i` receiver.

We mentioned at first that such an example is a wrong usage of type embedding. But what's the reason for that? As `sync.Mutex` is an embedded type, the `Lock` and `Unlock` methods will be promoted. Therefore, both methods will also become visible for external clients using `InMem`:

```
1 m := inmem.New()
2 m.Lock() // ??
```

This promotion is probably not desired. A mutex is, in most cases, something that we want to encapsulate within a struct and make invisible to external clients. Therefore, we shouldn't have made it an embedded field in this case:

```
1
```

```
2 type InMem struct {
3         mu sync.Mutex
4         m map[string]int
  }
```

Make the sync.Mutex field not embedded 1

As the mutex isn't embedded and unexported, it can't be accessed from external clients.

Let's now see another example, but this time, embedding can be considered a correct approach. We want to write a customer logger that would contain an `io.WriteCloser` and expose two methods: `Write` and `Close`. If `io.WriteCloser` isn't embedded, we would have to write it like so:

```
1  type Logger struct {
2          writeCloser io.WriteCloser
3  }
4
5  func (l Logger) Write(p []byte) (int, error) {
6          return l.writeCloser.Write(p)
7  }
8
9  func (l Logger) Close() error {
10         return l.writeCloser.Close()
11 }
12
13 func main() {
14         l := Logger{writeCloser: os.Stdout}
15         _, _ = l.Write([]byte("foo"))
16         _ = l.Close()
17 }
```

Forward the call to Writer 1
Forward the call to Writer 2

`Logger` would have to provide both a `Write` and a `Close` method that would *just* forward the call to `io.WriteCloser`. However, if the field now becomes embedded, we can remove these forwarding methods:

```
type Logger struct {
        io.WriteCloser
}

func main() {
        l := Logger{WriteCloser: os.Stdout}
        _, _ = l.Write([]byte("foo"))
        _ = l.Close()
}
```

Make io.Writer embedded 1

It remains the same for clients: two exported `Write` and `Close` methods. Yet, it prevents implementing these additional methods just to forward a call. Also, as `Write` and `Close` are promoted, it means that `Logger` satisfies the `io.WriteCloser` interface.

Note

Differentiating embedding from OOP subclassing can sometimes be confusing. The main difference is related to who is the receiver of a method.

Let's look at the following figure. The left-hand side represents a type `X` being embedded in `Y`, whereas on the right-hand side, `Y` extends `X`:

Figure 2.9. With embedding, the embedded type remains the receiver of a method. Conversely, with subclassing, the subclass becomes the receiver of a method.

With embedding, the receiver of `Foo` remains `X`. However, with subclassing, the receiver of `Foo` becomes the subclass, `Y`. Embedding is about composition, not inheritance.

So, what should we conclude about type embedding?

First, let's note that it's rarely a necessity, and it means that whatever the use case, we can probably solve it as well without type embedding. It's mainly used for convenience, in most cases to promote behaviors.

If we decide to use type embedding, we have to keep two main constraints in mind:

- It shouldn't be solely because of some syntactic sugar to simplify accessing a field (e.g., `Foo.Baz()` instead of `Foo.Bar.Baz()`). If it's the only rationale, let's not embed the inner type and use a field instead.
- It shouldn't promote data (fields) or a behavior (methods) we want to hide from outside. For example, if it allows clients to access a locking behavior that should have remained private to the struct.

> **Note**
>
> One may also argue that using type embedding could lead to extra efforts in terms of maintenance in the context of exported structs. Indeed, embedding a type inside an exported struct means remaining cautious when this type evolves. For example, if we add a new method in the inner type, we should ensure it doesn't break the latter constraint. Hence, to avoid this extra effort, teams can also prevent type embedding in public structs.

Using type embedding consciously by keeping these constraints in mind can help avoid boilerplate code with additional forwarding methods. However, let's make sure we don't do it solely for cosmetics and not promote elements that should have remained hidden.

In the next section, let's now discuss common patterns to deal with options configuration.

2.11 Not using the functional options pattern

When designing an API, one question may arise: how to deal with optional configuration? Solving this problem efficiently can improve how convenient our API will be. This section will go through a concrete example and cover different ways to handle it.

For this example, let's say we have to design a library that exposes a function to create an HTTP server. This function would accept different inputs: an address and a port. The skeleton of the function would be the following:

```
func NewServer(addr string, port int) (*http.Server, error) {
        // ...
}
```

The clients of our library have started to use this function, and everyone is happy. Yet, at some point in time, clients begin to complain that this function is somewhat limited and lacks other parameters (e.g., write timeout, connection context). However, we notice that adding new function parameters will break the compatibility, forcing the clients to modify the way they call `NewServer`.

In the meantime, we would like to enrich the logic related to port management this way:

- If the port isn't set, it uses the default one
- If the port is negative, it returns an error
- If the port is equal to 0, it uses a random port
- Otherwise, it uses the port provided by the client

Figure 2.10. Logic related to the port option.

How could we implement this function in an API-friendly way? Let's see different options.

2.11.1 Config struct

As Go doesn't support optional parameters in functions signature, the first possible approach is to use a config struct to convey what's mandatory and optional. For example, the mandatory parameters could live as function parameters, whereas the optional could be handled in the config struct:

```
type Config struct {
        Port        int
}


```

```
6 func NewServer(addr string, cfg Config) {
  }
```

This solution fixes the compatibility issue. Indeed, if we add new options, it will not break on the client-side. However, this approach doesn't solve our requirement related to port management. Indeed, we should bear in mind that if a struct field isn't provided, it will be initialized to its zero value:

- 0 for an integer
- 0.0 for a floating-point type
- `""` for a string
- nil for slices, maps, channels, pointers, interfaces, and functions

Therefore, in the following example, both structs are equal:

```
1 c1 := httplib.Config{
2         Port: 0,
3 }
4 c2 := httplib.Config{
5
6 }
```

Port is initialized to 0 1
Port is missing so initialized to 0 2

In our case, we need to find a way to distinguish a port purposely set to 0 and a missing port. Perhaps one option might be to handle all the parameters of the config struct as pointers this way:

```
type Config struct {
        Port        *int
}
```

Using a pointer of integer, semantically, we can make the difference between the value zero and a missing value (a nil pointer).

This option would work but has a couple of downsides. First, it's not handy for clients to provide an integer pointer. Clients have to create a variable and then pass a pointer this way:

```
port := 0
config := httplib.Config{
        Port: &port,
}
```

Provide integer pointer 1

It's not a showstopper as such, but the overall API becomes a bit less convenient to use. Also, the more options we will add, the more complex the code will be.

The second downside is that a client using our library with the default configuration will have to pass an empty struct this way:

```
httplib.NewServer("localhost", httplib.Config{})
```

This code doesn't look great. Readers will have to understand what is the meaning of this magic struct.

Another option would be to use the classic builder pattern as presented in the next section.

2.11.2 Builder pattern

Originally part of the Gang of Four design patterns, the builder pattern provides a flexible solution to various object creation problems. The construction of `Config` is separated from the struct itself. It requires an extra struct (`ConfigBuilder`) that will receive methods to configure and build a `Config`.

Let's see a concrete example and how it can help us in designing a friendly API that tackles all our requirements, including port management:

```
type Config struct {
        Port int
}

type ConfigBuilder struct {
        port *int
}

func (b *ConfigBuilder) Port(port int) *ConfigBuilder {
        b.port = &port
        return b
}

func (b *ConfigBuilder) Build() (Config, error) {
        cfg := Config{}

        if b.port == nil {
                cfg.Port = defaultHTTPPort
        } else {
                if *b.port == 0 {
                        cfg.Port = randomPort()
                } else if *b.port < 0 {
```

```
23                          return Config{}, errors.New("port
24 should be positive")
25                  } else {
26                          cfg.Port = *b.port
27                  }
28          }
29
30          return cfg, nil
31 }
32
33 func NewServer(addr string, config Config) (*http.Server,
34 error) {
           // ...
   }
```

Config struct 1
Config builder struct that contains an optional port 2
Public method to set up the port 3
Build method to create the config struct 4
Main logic related to port management 5

The `ConfigBuilder` struct holds the client configuration. It exposes a `Port` method to set up the port. Usually, such a configuration method returns the builder itself so that we can use method chaining (e.g., `builder.Foo("foo").Bar("bar")`). It also exposes a `Build` method that holds the logic on initializing the port value (whether the pointer was nil, etc.) and then returns a `Config` struct once created.

> **Note**
>
> There isn't a single possible implementation of the builder pattern. For example, one may favor an approach where the logic to define the final port value is inside the `Port` method instead of `Build`. Yet, the sections' scope is to present an overview of the builder pattern, not to delve into all the different possible variations.

Then, a client would use our builder-based API in the following manner (we assume that we have put our code in an `httplib` package):

```
1  builder := httplib.ConfigBuilder{}
2  builder.Port(8080)
3  cfg, err := builder.Build()
4  if err != nil {
5          return err
6  }
7
8  server, err := httplib.NewServer("localhost", cfg)
9  if err != nil {
10         return err
11 }
```

Creates a builder config 1
Set the port 2
Build the config struct 3
Pass the config struct 4

First, the client creates a `ConfigBuilder` and uses it to set up the optional field such as the port. Then, it calls the `Build` method and checks for errors. If ok, the config is passed to `NewServer`.

This approach makes the port management handier. It's not required to pass an integer pointer as the `Port` method accepts an integer. However, it's still needed to pass a config struct that might be empty if a client wants to use the default configuration:

```
server, err := httplib.NewServer("localhost", nil)
```

Another downside, in some situations, is related to error management. In programming languages where exceptions are thrown, the builder methods such as `Port` can raise exceptions if the input is invalid. If we want to keep the ability to chain the calls, the function can't return an error. Therefore, we have to delay the validation in the `Build` method. If a client can pass multiple options but want to handle precisely the case if a port is invalid, it makes error handling more complex.

Let's now see another approach called the functional options pattern that relies on variadic arguments.

2.11.3 Functional options pattern

The latest approach we will discuss is the functional options pattern. Though there are different implementations with minor variations, the main idea is the following:

- An unexported struct holds the configuration: `options`.
- Each option is a function that returns the same type: `type Option func(options *options) error`. For example, `WithPort` accepts a `int` argument representing the port and returns an `Option` type representing how to update the `options` struct.

Figure 2.11. The `WithPort` option updates the final `options` struct.

Here's the Go implementation for the `options` struct, the `Option` type, and the `WithPort` option.

```
1  type options struct {
2          port *int
3  }
4
5  type Option func(options *options) error
6
7  func WithPort(port int) Option {
8          return func(options *options) error {
9                  if port < 0 {
10                         return errors.New("port should be
11 positive")
12                 }
13                 options.port = &port
14                 return nil
15         }
   }
```

Configuration struct 1
Represent a function type that updates the configuration struct 2
An example of a configuration function updating the port 3

Here, `WithPort` returns a closure. A closure is an anonymous function that references variables from outside its body; here, the `port` variable. The closure respects the `Option` type and implements the port validation logic.

Each config field requires creating a public function (that starts with the `With` prefix for conventions) containing a similar logic: validating inputs if needed and updating the config struct.

Now, let's delve into the last part on the provider side, the `NewServer` implementation. The options will be passed as variadic arguments. Hence, we must iterate over these options to mutate the `options` config struct.

```go
func NewServer(addr string, opts ...Option) (*http.Server, error) {
	var options options
	for _, opt := range opts {
		err := opt(&options)
		if err != nil {
			return nil, err
		}
	}

	// At this stage, the options struct is built and contains the config
	// Therefore, we can implement our logic related to port configuration
	var port int
	if options.port == nil {
		port = defaultHTTPPort
	} else {
		if *options.port == 0 {
			port = randomPort()
		} else {
			port = *options.port
		}
	}

	// ...
}
```

Accept variadic Option arguments 1
Create an empty options struct 2
Iterate over all the input options 3
Call each option which will result in modifying the common options struct 4

We start by creating an empty `options` struct. Then, we iterate over each `Option` argument and execute them (bear in mind that the `Option` type is a function) to mutate the `options` struct. Then, once the `options` struct is built, we can implement the final logic regarding port management.

As `NewServer` accepts variadic `Option` arguments, a client can now call this API by passing multiple options following the mandatory address argument. For example:

```
server, err := httplib.NewServer("localhost",
                 httplib.WithPort(8080),
                 httplib.WithTimeout(time.Second))
```

However, in case the client needs the default configuration, it doesn't have to provide any argument such as an empty struct as we have seen with the previous approaches:

```
server, err := httplib.NewServer("localhost")
```

This pattern is the functional options pattern. It provides a handy and API-friendly way to handle options. Though the builder pattern can be a valid option, it has some minor downsides that tend to make the functional options

pattern the idiomatic way to deal with this problem in Go. Let's also note that this pattern is used in many different Go libraries, such as gRPC.

The next section will delve into a common mistake: Go projects misorganization.

2.12 Project misorganization

Organizing a Go project isn't an easy task. As the Go language gives a lot of freedom in designing packages and modules, the best practices are not quite as ubiquitous as they should be. This section will first delve into a common way of structuring a project and then discuss a few best practices showing how to improve how we organize a project.

2.12.1 Project structure

The language maintainer brings no strong convention about structuring a project in Go. However, one layout has emerged these years: github.com/golang-standards/project-layout.

If your project is small enough (only a few files), or if your organization has already created its standard, it may not be worth using or migrating to `project-layout`. Otherwise, it might be worth considering it. Let's delve into this layout and see what the main directories are:

- `/cmd`: Main source files. The `main.go` of a `foo` application should live in `/cmd/foo/main.go`.
- `/internal`: Private code that we don't want others importing in their applications or libraries.
- `/pkg`: Public code that we want to expose to others.

- `/test`: Additional external tests and test data. Unit tests in Go live in the same package as the source files. However, tests such as public API or integration tests should live in `/test`.
- `/configs`: Configuration files.
- `/docs`: Design and user documents.
- `/examples`: Examples for our application and/or public library.
- `/api`: API contract files (e.g., Swagger, Protocol Buffers, etc.).
- `/web`: Web application-specific asset (static files, etc.).
- `/build`: Packaging and CI files.
- `/scripts`: Scripts for analysis, installation, etc.
- `/vendor`: Application dependencies (e.g., Go modules dependencies).

There's no `/src` directory, just like in some other languages. The rationale is that `/src` is too generic; hence, this layout favors directories such as `/cmd`, `/internal` or `/pkg`.

> **Note**
>
> In 2021, Russ Cox, one of the core maintainers of Go, made some criticisms about this layout. Mainly because this project lives under the Github `golang-standards` organization despite not being an official standard. In any case, we must bear in mind that regarding project structure, there's no mandatory convention. This layout may be helpful for you or not, but what's important here is that indecision is the only wrong decision. Hence, agree on a layout to keep things consistent in your organization so that developers don't waste time switching from one repo to another.

Now, let's discuss how to organize the main logic of a Go repository.

2.12.2 Package organization

In Go, there is no concept of sub-packages. However, we can decide to organize packages within subdirectories. If we take a look at the standard library, the `net` directory is organized this way:

```
/net
    /http
        client.go
        ...
    /smtp
        auth.go
        ...
    addrselect.go
    ...
```

`net` acts both as a package and a directory that contains other packages. Yet, `net/http` doesn't inherit from `net` or have specific access rights to the `net` package. Elements inside of `net/http` can only see exported `net` elements. The main benefit of subdirectories is to keep packages in a place where they live with high cohesion.

Regarding the overall organization, there are different schools of thought. For example, shall we organize our application by context or by layer? It depends on our preferences. One may favor grouping code per context (e.g., the customer context, the contract context), one may favor following hexagonal architecture principles and group per technical layer. If the decision we take fits our use case, it cannot be a wrong decision as long as we remain consistent with it. Let's remember that the only wrong decision is no decision.

Regarding packages, there are multiple best practices that we should follow.

First, we should avoid premature packaging as it might cause us to overcomplicate a project. Sometimes, it's better to keep a simple organization and make our project evolve when we understand what it contains rather than forcing ourselves to have the perfect structure upfront.

Granularity is another essential thing to consider. We should avoid having dozens of nano package containing only one or two files. If we do, it's because we have probably missed some logical connections across these packages, making our project harder to understand for readers. Conversely, we should also avoid huge packages that dilute the meaning of a package name.

Package naming should also be taken with care. As we all know as developers, naming is hard. To help clients understand a Go project, we should name a package after what they provide, not what they contain. Also, naming should be meaningful. Therefore, a package name should be short, concise, and expressive. Also, by convention, lower case and a single-word name.

Regarding what to export, the rule is pretty straightforward. We should minimize what should be exported as much as possible to reduce the coupling between packages and keep unnecessary exported elements hidden. If we are unsure whether to export an element or not, we should default to not exporting it. Later, if we discover that we need to export it, we can adjust our code. Let's also keep in mind some exceptions, such as making fields exported so that a struct can be unmarshalled with `encoding/json`.

Organizing a project isn't straightforward, and if we follow these rules, it should help us make it easier to maintain. However, let's remember that consistency is also vital to ease maintainability. So, let's make sure that we keep things as consistent as possible within a codebase.

In the next section, we are going to tackle utility packages.

2.13 Creating utility packages

This section will discuss a common bad practice: creating shared packages such as `utils`, `common` or `base`. We will understand the problems with such an approach and how to improve our organization.

Let's see an example inspired by the official Go blog. It's about implementing a set data structure (a map where the value is ignored). The idiomatic way to do it in Go is to handle it via a `map[K]struct{}` type with `K` that can be of any allowed types in a map whereas the value is a `struct{}` type. Indeed, a map whose value type is `struct{}` conveys that we aren't interested in the value itself. Let's expose two methods in a `util` package:

```
1 package util
2
3 func NewStringSet(...string) map[string]struct{} { ... }
4 func SortStringSet(map[string]struct{}) []string { ... }
```

Creates a string set 1

Return a sorted list of keys 2

A client will use this package like this:

```
1 set := util.NewStringSet("c", "a", "b")
2 fmt.Println(util.SortStringSet(set))
```

The problem here is that `util` is meaningless. We could call it `common`, `shared`, or `base`, it remains a meaningless name that doesn't provide any insights about what the package provides.

Instead of a utility package, we should create an expressive package name such as `stringset`, for example:

```
package stringset

func New(...string) map[string]struct{} { ... }
func Sort(map[string]struct{}) []string { ... }
```

We removed the suffixes for `NewStringSet`, and `SortStringSet`, which respectively became `New` and `Sort`. On the client-side, it will now look like this:

```
set := stringset.New("c", "a", "b")
fmt.Println(stringset.Sort(set))
```

Note

In the previous section, we discussed the idea of nano packages. However, we mentioned how creating dozens of nano packages in an application can make the code path more complex to follow. However, the idea itself of a nano package isn't necessarily bad. If a small code group has high cohesion and doesn't really belong somewhere else, it's perfectly acceptable to organize it into a specific package. There isn't a strict rule to apply, and often, the challenge is finding the right balance.

We could even go a step further. Instead of exposing utility functions, we can even create a specific type and expose `Sort` as a method this way:

```
package stringset

type Set map[string]struct{}
func New(...string) Set { ... }
func (s Set) Sort() []string { ... }
```

This change would make the client even simpler. There would only be one reference to the `stringset` package:

```
set := stringset.New("c", "a", "b")
fmt.Println(set.Sort())
```

With this small refactoring, we got rid of a meaningless package name to expose an expressive API.

As Dave Cheney (a project member of Go) mentioned, it's reasonably frequent to find utility packages to handle common facilities. For example, if we decided to have a `client` and a `server` package, where should we put the common types? In this case, perhaps one solution is to combine the client, the server, and the common code into a single package.

Naming a package is a critical piece of application design, and we should be cautious about it. As a rule of thumb, creating shared packages without meaningful names isn't a good idea; this includes utility packages such as `utils`, `common` or `base`. Also, let's bear in mind that naming a package after what it provides and not what it contains can be an efficient way to increase its expressiveness.

In the next section, we will keep discussing packages with package collision.

2.14 Ignoring package name collisions

Package collision occurs when a variable name collides with an existing package name preventing the package from being reused. Let's see a concrete example with a library exposing a Redis client:

```
package redis

type Client struct { ... }

func NewClient() *Client { ... }

func (c *Client) Get(key string) (string, error) { ... }
```

Now let's jump on the client-side. Despite the package name is called `redis`, it's perfectly valid in Go to also create a variable named `redis`:

```
redis := redis.NewClient()
v, err := redis.Get("foo")
```

Call NewClient from the redis package 1
Use the redis variable 2

Here, the `redis` variable name collides with the `redis` package. Even though this is allowed, it should be avoided. Indeed, throughout the scope of the `redis` variable, the `redis` package isn't accessible anymore. Also, suppose a qualifier references both a variable and a package name throughout a function. In that case, it might be ambiguous for a code reader to know what a qualifier refers to. What are the options to avoid such a collision?

The first option is to find a different variable name. For example:

```
redisClient := redis.NewClient()
v, err := redisClient.Get("foo")
```

This is probably the most straightforward approach. However, if, for some reason, we prefer to keep naming our variable `redis`, we can play with package imports.

Using package imports, we can use an alias to change the qualifier to reference the `redis` package. For example:

```
import redisapi "mylib/redis"

// ...

redis := redisapi.NewClient()
v, err := redis.Get("foo")
```

Create an alias to the redis package 1
Access to the redis package via the redisapi alias 2

Here, we used the `redisapi` import alias to reference the `redis` package so that we can keep naming our variable `redis`.

Note

One option could also be to use dot imports to access all the public elements of a package without the package qualifier.

> However, this approach tends to increase the confusion and should, in most cases, be avoided.

Let's also note that we should also avoid naming collisions between a variable and a built-in function:

```
copy := copyFile(src, dst)
```

In this case, the `copy` built-in function wouldn't be accessible anymore as long as the `copy` variable lives.

In summary, we should prevent variable name collisions to avoid ambiguity. If we face a collision, we should either find another meaningful name or use an import alias.

In the next section, we will see a common mistake related to code documentation.

2.15 Missing code documentation

Documentation is an important aspect of coding. It simplifies how clients can consume an API but can also help in maintaining a project. In Go, we should follow some rules to make our code idiomatic. Let's delve into them.

First, every exported element must be documented. Be it a structure, an interface, a function, etc. If it's exported, it must be documented. The convention is to make the comments starting with the name of the exported element. For example:

```go
// Customer is a customer representation.
type Customer struct{}

// ID returns the customer identifier.
func (c Customer) ID() string { return "" }
```

As a convention, each comment should be a complete sentence that ends with punctuation. Let's also bear in mind that when we document a function (or a method), we should highlight what the function intends to do, not how it does it; this belongs to the core of a function and comments, not documentation. Furthermore, documentation should ideally provide enough information for a consumer not to delve into our code to understand how to use an exported element.

Note

Then, if a developer uses the `ComputePath` function, he should have a warning (most IDEs handle deprecated comments).

It's possible to deprecate an exported element using the `// Deprecated:` comment this way:

```go
// ComputePath returns the fastest path between
two points.
// Deprecated: This function uses a deprecated
way to compute
// the fastest path. Use ComputeFastestPath
instead.
func ComputePath() {}
```

When it comes to documenting a variable or a constant, we may be interested in conveying two aspects: its purpose and its content. The former should live as code documentation to be useful for external clients. The latter, though, shouldn't necessarily be public. For example:

```
// DefaultPermission is the default permission used by the
store engine.
const DefaultPermission = 0o644 // Need read and write
accesses.
```

This constant represents the default permission. The code documentation conveys its purpose, whereas the comment alongside the constant describes its actual content (read and write accesses).

To help clients and maintainers understand a package's scope, we should also document each package. The convention is to start the comment by `// Package `, followed by the package name this way:

```
// Package math provides basic constants and mathematical
functions.
//
// This package does not guarantee bit-identical results
// across architectures.
package math
```

The first line of a package comment should be concise as it will appear in the package list:

Figure 2.12. An example of the generated Go standard library.

Name	Synopsis
archive	
tar	Package tar implements access to tar archives.
zip	Package zip provides support for reading and writing ZIP archives.
bufio	Package bufio implements buffered I/O. It wraps an io.Reader or io.Writer object, creating another object (Reader or Writer) that also implements the interface but provides buffering and some help for textual I/O.
builtin	Package builtin provides documentation for Go's predeclared identifiers.
bytes	Package bytes implements functions for the manipulation of byte slices.

Then, we can provide all the information we need in the following lines.

Commenting a package can be done in any Go files of this package; there is no rule. In general, we should try to put it in a relevant file with the same name of the package or a specific file such as `doc.go`.

One last thing to mention regarding package documentation, the comments not adjacent to the declaration are omitted. For example, the following copyright comment will not be visible in the produced documentation:

```
1 // Copyright 2009 The Go Authors. All rights reserved.
2 // Use of this source code is governed by a BSD-style
3 // license that can be found in the LICENSE file.
4
5 // Package math provides basic constants and mathematical
6 functions.
7 //
8 // This package does not guarantee bit-identical results
9 // across architectures.
  package math
```

Empty line, therefore the above comment will not be included in the documentation 1

In summary, we should keep in mind that every exported element has to be documented. Documenting our code shouldn't be a constraint. We should take

the opportunity to make sure it will help clients and maintainers to understand the purpose of our code.

In the last section of this chapter, we will see a common mistake regarding tooling: not using linters.

2.16 Not using linters

A linter is an automatic tool to analyze code and catch errors. The scope of this section isn't to give an exhaustive list of the existing linters; otherwise, it will become deprecated pretty quickly. Yet, we should understand and remember why linters are essential for most Go projects.

To understand why linters are important, let's take one concrete example. In *Unintended variable shadowing*, we discussed potential errors related to variable shadowing. Using `vet`, a standard linter from the Go toolset, and `shadow`, we can detect shadowed variables:

```go
package main

import "fmt"

func main() {
        i := 0
        if true {
                i := 1
                fmt.Println(i)
        }
        fmt.Println(i)
}
```

Shadowed variable 1

`vet` being included alongside the Go binary, let's first install `shadow`, link it with Go `vet`, and then run it on the previous example.

```
1 $ go install
2 golang.org/x/tools/go/analysis/passes/shadow/cmd/shadow
3 $ go vet -vettool=$(which shadow)
  ./main.go:8:3: declaration of "i" shadows declaration at line 6
```

shadow installation 1

Linking to Go vet using the vettol argument 2

Go vet was able to detect the shadow variable 3

As we can notice, `vet` informs us that the variable `i` is shadowed in this example. Therefore, using appropriate linters can help make our code more robust and detect potential errors.

> **Note**
>
> Linters don't cover all the mistakes of this book. Therefore, the reader is recommended to continue his progression ;).

Again, this section's goal isn't to list all the available linters. However, if you're not a regular user of linters, here is a possible list that you may want to use daily:

- vet: standard Go analyzer
- kisielk/errcheck: errors checker
- fzipp/gocyclo: cyclomatic complexity analyzer
- goconst: repeated string constants analyzer

Besides linters, we should also use code formatters to fix code style:

- gofmt: standard Go code formatter
- goimports: standard Go imports formatter

Meanwhile, we should also look at golangci-lint. It's a linting tool that provides a facade on top of many useful linters and formatters. Also, it allows running the linters in parallel to improve analysis speed which is quite handy.

Linters and formatters are a powerful way to improve the quality and consistency of our codebase. Let's take the time to understand which one we should use and make sure we automate their execution (e.g., CI or Git pre-commit hook).

2.17 Chapter summary

- Avoiding shadowed variables can help keep us from making mistakes like referencing the wrong variable or confusing readers.
- Avoiding nested levels and keeping the happy path aligned on the left makes building a mental code model easier.
- In case we need to initialize some variables, let's remember that init functions have limited error handling and make state handling and testing more complex. In most cases, initializations should be handled as specific functions.
- Forcing to use getters and setters isn't idiomatic in Go. Being pragmatic and finding the right balance between efficiency and blindly following certain idioms should be the way to go.
- Abstractions should be discovered, not created. To prevent unnecessary complexity, we should create an interface when we need it, not when we foresee needing it. Or at least if we can prove this abstraction is a valid one.
- Keeping interfaces on the client side avoids unnecessary abstractions.
- To prevent being restricted in terms of flexibility, a function shouldn't return interfaces but concrete implementation in most cases. Conversely, a function should accept interfaces whenever possible.
- Only use `any` if you need to accept or return any possible type such as `json.Marshal`. Otherwise, `any` doesn't provide meaningful information and can lead to compile-time issues by allowing a caller to call methods with any data type.

- Relying on generics and type parameters can prevent us from writing boilerplate code to factor out elements or behaviors. However, let's not use type parameters prematurely but only when we see a concrete need for them. Otherwise, it would introduce unnecessary abstractions and complexity.
- Using type embedding can also help us avoid some boilerplate code; however, we should ensure it doesn't lead to visibility issues where some fields should have remained hidden.
- To handle options conveniently and in an API-friendly manner, we should use the function options pattern.
- Following a layout such as `project-layout` can be a good way to start structuring Go projects, especially if we are looking for existing conventions to standardize a new project.
- Naming is a critical piece of application design. Creating packages such `common`, `util`, or `shared` doesn't bring much value for the reader. We should refactor such packages into meaningful and specific packages.
- To avoid naming collisions between variables and packages leading to confusion or perhaps even bugs, use unique names for each. If this isn't feasible, we can use an import alias to change the qualifier to differentiate the package name from the variable name or think about a better name.
- To help clients and maintainers understand our code's purpose, we should document exported elements.
- To improve code quality and consistency, we should use linters and formatters.

[3] cs.opensource.google/go/x/website/+/e0d934b4:blog/blog.go;l=32

[4] www.youtube.com/watch?v=PAAkCSZUG1c&t=318s

[5] github.com/DATA-DOG/go-sqlmock

[6] datatracker.ietf.org/doc/html/rfc761

[7] www.youtube.com/watch?v=PAAkCSZUG1c&t=7m36s

3 Data types

This chapter covers

- Covering common mistakes related to basic types
- Understanding the fundamental concepts for slices and maps to prevent possible bugs, leaks, or inaccuracy
- Comparing values

Dealing with data types is a frequent operation for software engineers. This chapter will delve into the most common mistakes related to basic types, slices, and maps. The only data type that we will omit is strings, as it will be a chapter on its own (5 - Strings).

3.1 Creating confusion with octal literals

This chapter will discuss a common confusion with octal literal representation that can lead to confusion or even bugs.

What do you believe should be the output of the following code?

```
sum := 100 + 010
fmt.Println(sum)
```

At first sight, we may expect this code to print the result of 100 + 10 = 110. Yet, it will print 108 instead. How is it possible?

In Go, we must know that an integer literal starting with 0 is considered an octal integer (base 8). 10 in base 8 equals 8 in base 10. Thus, the sum is equal to 100 + 8 = 108. It's an important property of integer literals to keep in mind to avoid confusion while reading existing code, for example.

Octal integers are useful on different occasions. For instance, if we want to open a file using `os.OpenFile`. This function requires passing a permission as an `uint32`. If we want to match the Linux permissions, we can pass an octal number for readability instead of a base 10 number:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0644)
```

Also, we should know that it's possible to add an `o` character (the letter in lowercase) following the zero:

```
file, err := os.OpenFile("foo", os.O_RDONLY, 0o644)
```

Using `0o` as a prefix instead of only `0` means the same; however, it can help make the code clearer.

> **Note**
>
> We can also use an uppercase `O` character instead of a lowercase `o`. Yet, passing `0O644` can increase the confusion as depending on the character font, `0` can be closed to `O`.

We should also note the other integer literals representation:

- Binary: `0b` or `0B` prefix. For example, `0b100` is equal to 4 in base 10.
- Hexadecimal: `0x` or `0X` prefix. For example, `0xF` is equal to 15 in base 10.
- Imaginary: `i` suffix. For example, `3i`.

Finally, we can also use an underscore character as a separator for readability reasons. For example, we can write 1 billion this way: `1_000_000_000`. The underscore character can also be used with other representations (e.g., `0b00_00_01`).

In summary, Go handles binary, hexadecimal, imaginary, and octal numbers. Octal numbers start by 0. However, to improve readability and potential mistakes with future code readers, we should make octal numbers explicit using an `0o` prefix.

In the next section, we will dig into integers and discuss how overflows are handled in Go.

3.2 Neglecting integer overflows

Not understanding how integer overflows are handled in Go can lead to critical bugs. This section will delve into this question. But first, let's remind us of a few concepts related to integers.

3.2.1 Concepts

Go provides a total of ten integer types. There are four signed integer types:

- `int8`: 8 bits

- `int16` : 16 bits
- `int32` : 32 bits
- `int64` : 64 bits

And four unsigned integer types:

- `uint8` : 8 bits
- `uint16` : 16 bits
- `uint32` : 32 bits
- `uint64` : 64 bits

Furthermore, two integer types are the most commonly used:

- `int`
- `uint`

These two types have a size that depends on the system: 32 bits on 32-bit systems or 64 bits on 64-bit systems.

Let's now discuss overflow. We will use an `int32` that we will initialize to its maximum value and increment it. What should be the behavior of this code?

```
var counter int32 = math.MaxInt32
counter++
fmt.Printf("counter=%d\n", counter)
```

This code compiles and won't panic at runtime. However, the `counter++` statement will generate an integer overflow:

```
counter=-2147483648
```

An integer overflow occurs when an arithmetic operation creates a value outside of the range that can be represented with a given number of bytes.

An `int32` is represented using 32 bits. Here is the binary representation of the maximum `int32` value (`math.MaxInt32`):

```
1 01111111111111111111111111111111
2  |------31 bits set to 1-------|
```

As an `int32` is a signed integer, the bit on the left represents the integer's sign: 0 for positive, 1 for negative. If we increment this integer, there is no space left to represent the new value. Hence, it will lead to an integer overflow. Binary-wise, here's the new value:

```
1 10000000000000000000000000000000
2  |------31 bits set to 0-------|
```

As we can notice, the bit sign is now equal to 1, meaning negative. This value is the smallest possible value for a signed integer represented with 32 bits.

> **Note**
>
> The smallest possible negative value isn't `11111111111111111111111111111111` . Indeed, most systems rely

on the two's complement operation to represent binary numbers (invert every bit and add 1). The main goal of this operation is to make x - x equal zero regardless of x.

In Go, an integer overflow that can be detected at compile time will generate a compilation error. For example:

```
var counter int32 = math.MaxInt32 + 1
```

```
constant 2147483648 overflows int32
```

However, at runtime, an integer overflow and underflow are silent; it will not lead to an application panic. It is essential to have this behavior in mind as it may lead to sneaky bugs—for example, an integer increment or addition of positive integers that leads to a negative result.

Before delving into how to detect integer overflow with common operations, let's think about when to worry about it. In most contexts, like handling a counter of requests or basic additions/multiplications, we shouldn't worry too much if we use the right integer type. Yet, in some specific cases, such as memory-constrained projects using smaller integer types, dealing with large numbers, or when doing conversions, we may want to check possible overflows.

Note

The Ariane 5 launch failure is due to an overflow resulting from converting a 64-bit floating-point to a 16-bit signed integer.

3.2.2 Detect integer overflow during increment

If we want to detect an integer overflow during an increment operation with a type based on a defined size (`int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, or `uint64`), it can be achieved by checking the value against the `math` constants. For example, with an `int32`:

```
func Inc32(counter int32) int32 {
        if counter == math.MaxInt32 {
                panic("int32 overflow")
        }
        return counter + 1
}
```

1

This function checks whether the input is already equal to `math.MaxInt32`. We know the increment will lead to an overflow if it's the case.

What about `int` and `uint` types? Before Go 1.17, we had to build these constants manually. Now, they are part of the `math` package: `math.MaxInt`, `math.MinInt`, and `math.MaxUint`.

If we have to test an overflow on an `int` type, we can do it using `math.MaxInt`:

```
func IncInt(counter int) int {
        if counter == math.MaxInt {
                panic("int overflow")
        }
        return counter + 1
}
```

The logic is the same for an `uint`; we can use `math.MaxUint`:

```
func IncUint(counter uint) uint {
        if counter == math.MaxUint {
                panic("uint overflow")
        }
        return counter + 1
}
```

We have seen how to check integer overflows following an increment operation. Now, what about additions?

3.2.3 Detect integer overflow during addition

How can we detect an integer overflow during an addition? By reusing `math.MaxInt`:

```
func AddInt(a, b int) int {
        if a > math.MaxInt-b {
                panic("int overflow")
        }

        return a + b
}
```

1

`a` and `b` being the two operands; if `a` is greater than `math.MaxInt - b`, it means the operation will lead to an integer overflow.

Now, let's see with the latest operation: multiplication.

3.2.4 Detect integer overflow during multiplication

Multiplications are a bit more complex to handle. We have to perform checks against the minimal integer: `math.MinInt`:

The implementation is the following:

```go
func MultiplyInt(a, b int) int {
        if a == 0 || b == 0 {
                return 0
        }

        result := a * b
        if a == 1 || b == 1 {
                return result
        }
        if a == math.MinInt || b == math.MinInt {
                panic("integer overflow")
        }
        if result/b != a {
                panic("integer overflow")
        }
        return result
}
```

Checking an integer overflow with multiplication requires multiple steps. First, testing if one of the operands is equal to zero, one or `math.MinInt`. Then we divide the multiplication result by `b`. If the result isn't equal to the original factor (`a`), it means an integer overflow occurred.

In summary, an integer overflow (and underflow) are silent operations in Go. If we want to check whether an overflow leads to an overflow to avoid sneaky

errors, we can use the utility functions described in this section. Let's also remind that Go provides a package to deal with large numbers: `math/big`. It might also be an option if an `int` isn't enough.

Let's keep talking about basic Go types in the next section with floating-points.

3.3 Not understanding floating-points

In Go, there are two floating-point types (if we omit imaginary numbers): `float32` and `float64`. The concept of floating-point was invented to solve the major problem with integers: their inability to represent fractional values. To avoid bad surprises, we have to know that floating-point arithmetic is an approximation of real arithmetic. Let's discover the impacts of working with approximations and how to increase accuracy.

Let's take a look at a multiplication example:

```
var n float32 = 1.0001
fmt.Println(n * n)
```

We may expect this code to print the result of 1.0001 * 1.0001: 1.00020001, right? However, running this code on most x86 processors prints 1.0002 instead. How can we explain that? Let's understand the arithmetic of floating points.

If we take the `float64` type as an example, we should note that there's an infinite number of real values between `math.SmallestNonzeroFloat64` (the `float64` minimum) and `math MaxFloat64` (the `float64` maximum). Conversely, the `float64` type has a finite number of bits: 64. As making

infinite values fitting into a finite space isn't possible, we have to work with approximations. Hence, possibly losing precision. The same logic goes for the `float32` type.

Floating-points in Go follow the IEEE-754 standard, with some bits representing a mantissa and some other bits representing an exponent. A mantissa is a base value, whereas an exponent is a multiplier applied to the mantissa. In single-precision floating-point (`float32`), 8 bits represent the exponent, and 23 bits represent the mantissa. In double-precision floating-point (`float64`), it's 11 and 52 bits respectively for the exponent and the mantissa. The remaining bit is for the sign. To convert a floating-point into a decimal, we do the following calculation:

```
sign * 2^exponent * mantissa
```

Let's look at the representation of 1.0001 as a `float32` :

Figure 3.1. Representation of 1.0001 in `float32` .

The exponent uses the 8-bit excess/bias notation. Here, the 01111111 exponent value means 2^0, whereas the mantissa is equal to 1.000100016593933 (the scope of this section isn't to explain how conversions work). Hence, the decimal value equals 1 * 2^0 * 1.000100016593933. Thus, what we stored into a single-precision floating-

point isn't 1.0001 but 1.000100016593933. It's a lack of precision that affects the accuracy of the value stored.

Once we understand that `float32` and `float64` are approximations, what are the implications for us, developers?

The first implication is related to comparisons. Using the `==` operator to compare two floating-point numbers can lead to inaccuracies. Instead, we should compare that their difference is less than some small error value. For example, the `testify` [8] testing library has an `InDelta` function to assert that two values are within a given delta of each other.

Let's also bear in mind that the result of floating-point calculations depends on the actual processor. Most processors have a Floating-Point Unit (FPU) to deal with such calculations. There is no guarantee that one result executed on a machine will be the same on another machine with a different FPU. Again, comparing two values using a delta can be a solution for implementing valid tests across different machines.

Note

Here's an example that constructs these special kinds of numbers:

```
var a float64
positiveInf := 1 / a
negativeInf := -1 / a
nan := a / a
fmt.Println(positiveInf, negativeInf, nan)
```

This code prints the following:

```
+Inf -Inf NaN
```

We can check whether a floating-point number is infinite using `math.IsInf` and whether it is NaN using `math.IsNaN`.

Go also has three special kinds of floating-point numbers:

- Positive infinite.
- Negative infinite.
- NaN (Not-a-Number), the result of an undefined or unrepresentable operation. According to IEEE-754, the only floating-point number satisfying `f != f`.

We have seen that decimal to floating-point conversions can lead to losing accuracy. This is the error due to conversion. We also have to note that the error can accumulate in a sequence of floating-point operations.

Let's see an example with two functions that will perform the same sequence of operations in a different order. `f1` will start by initializing a `float64` to 10000 and then repeatedly adding 1.0001 to this result (`n` times). Conversely, `f2` will do the same operations but in the opposite order (add 10000 in the end).

```go
func f1(n int) float64 {
        result := 10_000.
        for i := 0; i < n; i++ {
                result += 1.0001
        }
        return result
}

func f2(n int) float64 {
        result := 0.
        for i := 0; i < n; i++ {
```

```
                result += 1.0001
        }
        return result + 10_000.
}
```

Let’s run these functions on an x86 processor while varying `n`:

Table 3.1. Varying *n* and executing `f1` and `f2`

`n`	Exact result	`f1`	`f2`
10	10010.001	10010.000999999993	10010.001
1k	11000.1	11000.099999999293	11000.099999999982
1m	1.0101e+06	1.0100999999761417e+06	1.0100999999766762e+06

We can notice that the bigger `n` is, the greater is the imprecision. However, we can also see that `f2` accuracy is better than `f1`. Indeed, we should keep in mind that the order of floating-point calculations can affect the accuracy of the result.

When performing a chain of additions and subtractions, we should group the operations to add or subtract values with a similar order of magnitude before adding or subtracting those with magnitudes that aren’t close. As `f2` performs the 10000 addition, in the end, it produces more accurate results than `f1`.

What about multiplications and divisions? Let’s imagine that we want to compute the following:

```
a × (b + c)
```

As we know, this calculation is equal to:

```
a × b + a × c
```

Let’s run these two calculations with `a` having a different order of magnitude than `b` and `c`:

```
a := 100000.001
b := 1.0001
c := 1.0002

fmt.Println(a * (b + c))
fmt.Println(a*b + a*c)
```

```
200030.00200030004
200030.0020003
```

The exact result is 200030.002. Hence, the first calculation has the worst accuracy. Indeed, when performing floating-point calculations involving additions, subtractions, multiplications, and divisions, we have to complete the multiplication and division operations first to have better accuracy. Sometimes, it may impact the execution time as in the previous example, it

requires three operations instead of two. In that case, it's a choice between accuracy and execution time.

Go's `float32`, and `float64` are approximations. Because of that, we have to bear a few rules in mind:

- When comparing two floating-point numbers, we should compare that their difference is within an acceptable range.
- We should group operations with a similar order of magnitude for better accuracy when performing additions or subtractions.
- Again, to favor accuracy, if a sequence of operations requires additions, subtractions, multiplications, and divisions, we should perform the multiplications and divisions first.

The following section will start delving into slices and discuss two crucial concepts: slice's length and capacity.

3.4 Not understanding slice length and capacity

It's pretty common for Go developers to mix slice length and capacity or not to understand them thoroughly. Assimilating these two concepts is essential for efficiently handling core operations, such as slice initialization and adding elements with `append`, copy, or slicing. Otherwise, it may lead to using slices suboptimally or even, as we will see in further sections, memory leaks.

In Go, a slice is backed by an array. It means the slice's data are stored contiguously in an array data structure. A slice also handles the logic of adding an element if the backing array is full or how to shrink the backing array if almost empty.

Internally, a slice holds a pointer towards the backing array plus a length and a capacity. The length is the number of elements the slice contains, whereas the capacity is the number of elements in the backing array. Let's go through a few examples to make things clear.

First, let's initialize a slice with a given length and capacity:

```
s := make([]int, 3, 6)
```

The first argument representing the length is mandatory. However, the second argument representing the capacity is optional.

The following figure shows the result of this code in memory:

Figure 3.2. A 3-length, 6-capacity slice.

`make` created an array of six elements (the capacity). Yet, as the length was set to 3, Go initialized only the first three elements. As the slice in an `[]int` type, the first three elements are initialized to the zeroed value of an `int`: 0. The greyed elements are allocated but unused yet.

If we print this slice, we will get the elements within the range of the length: `[0 0 0]`.

If we set `s[1]` to 1, it will update the second element of the slice without impacting its length or capacity, as shown in the next figure:

Figure 3.3. `s[1] = 1`.

However, accessing an element outside the length range is forbidden, even though it's already allocated in memory. For example, `s[4] = 0` would lead to a panic:

```
panic: runtime error: index out of range [4] with length 3
```

So, how can we use the remaining space of the slice? By using the `append` built-in function:

```
s = append(s, 2)
```

This code appended to the existing `s` slice a new element. It uses the first greyed element (which was allocated but unused yet) to store element `2` as shown in the following figure:

Figure 3.4. Appending an element to `s`.

The length of the slice was updated from three to four as the slice now contains four elements.

Now, what happens if we add three more elements so that the backing array isn't large enough?

```
s = append(s, 3, 4, 5)
fmt.Println(s)
```

If we run this code, we can notice that the slice was able to cope with our request:

```
[0 1 0 2 3 4 5]
```

As an array is a fixed-size structure, it could store the new elements until element `4`. Once we want to insert element `5`, as the array is already full, internally Go has created another array by doubling the capacity, copied all the elements, and then inserted element `5`:

Figure 3.5. As the initial backing array was full, it created another array and copied all the elements.

> **Note**
>
> In Go, a slice grows by doubling until it contains 1024 elements, after which it grows by 25% each time.

The slice now references the new backing array. What will happen to the previous backing array? If it's not referenced anymore, it will be freed eventually by the garbage collector (GC) if allocated on the heap (we will discuss heap memory in *Not understanding stack vs. heap* and how the GC works in *Not understanding how the GC works*).

Now, what happens with slicing? Slicing is an operation done on an array or a slice by providing a half-open range; the first index is included while the second is excluded. Let's now see the impacts:

```
s1 := make([]int, 3, 6)
s2 := s1[1:3]
```

```
1
2
```

The following figure shows the result in memory:

Figure 3.6. `s1` and `s2` reference the same backing array with different lengths and capacities.

First, `s1` is created as a 3-length, 6-capacity slice. When `s2` is created by slicing `s1`, both slices reference the same backing array. However, `s2` starts from a different index: 1. Therefore, its length and capacity are different from `s1`: 2-length, 5-capacity.

If we update `s1[1]` or `s2[0]`, the change will be done to the same array. Hence, it will become visible for both slices as shown in the next figure:

Figure 3.7. As `s1` and `s2` are backed by the same array, updating an element part of both slices will be visible in both slices.

Now, what happens if we append an element to `s2`? Would this code change `s1` as well?

```
s2 = append(s2, 2)
```

The shared backing array is modified, but only the length of `s2` will change:

Figure 3.8. Appending an element to `s2`.

`s1` remains a 3-length, 6-capacity slice. Therefore, if we print `s1` and `s2`, the added element is only visible for `s2`:

```
s1=[0 1 0], s2=[1 0 2]
```

It's important to understand this behavior not to make wrong assumptions while using `append`.

> **Note**
>
> In these examples, the backing array is internal and not available directly to the Go developer. The only exception is when a slice is created from slicing an existing array.

One last thing to note. What if we keep appending elements to `s2` until the backing array is full? What will be the state memory-wise? Let's add three more elements so that the backing array will not have enough capacity:

```
s2 = append(s2, 3)
s2 = append(s2, 4)
s2 = append(s2, 5)
```

1

This code will lead to creating another backing array:

Figure 3.9. Keep appending elements to `s2` until the backing array is full.

As we can notice, `s1` and `s2` are now referencing two different arrays. Indeed, as `s1` is still a 3-length, 6-capacity slice, it still has some available buffer, so it keeps referencing the initial array. Also, as we can notice, the new backing array was made by copying the initial one from the first index of `s2`. It's the reason why the new array starts by element `1`, not `0`.

To summarize, the slice length is the number of available elements in the slice, whereas the slice capacity is the number of elements in the backing array. Adding an element to a full slice (length == capacity) leads to creating a

new backing array with a new capacity, copying all the elements from the previous array, and updating the slice pointer to the new array.

In the next section, we will use the concepts of length and capacity with slice initialization.

3.5 Inefficient slice initialization

While initializing a slice using `make`, we've seen that we have to provide a length and an optional capacity. Forgetting to pass an appropriate value for both of these parameters when it makes sense is a widespread mistake. Let's see precisely when it's considered appropriate.

Let's take a look at the following example. We will have to implement a `convert` function that maps a slice of `Foo` into a slice of `Bar`. Both slices will have the same number of elements. Here is a first implementation:

```
func convert(foos []Foo) []Bar {
        bars := make([]Bar, 0)

        for _, foo := range foos {
                bars = append(bars, fooToBar(foo))
        }
        return bars
}
```

First, we initialized an empty slice of `Bar` elements using `make([]Bar, 0)`. Then, we use `append` to add the `Bar` elements.

At first, `bars` is empty, so adding the first element will allocate a backing array of size one. Then, every time the backing array is full, Go will recreate another array by doubling its capacity, as discussed in the previous section.

This logic of having to create another array because the current one is full will be repeated multiple times: when we add a 3rd element, a 5th, a 9th, etc. Assuming the input slice has 1000 elements, this algorithm requires allocating ten backing arrays and copying from one array to another more than 1000 elements in total. It will also lead to additional efforts for the GC to clean all these temporary backing arrays.

Performance-wise, there's no good reason not to give the Go runtime a helping hand. There are two different options for that.

The first option is to reuse the same code but to allocate the slice with a given capacity:

```
func convert(foos []Foo) []Bar {
        n := len(foos)
        bars := make([]Bar, 0, n)

        for _, foo := range foos {
                bars = append(bars, fooToBar(foo))
        }
        return bars
}
```

The only change is to create `bars` with a capacity equal to `n`, the length of `foos`. Internally, Go preallocates an array of *n* elements. Therefore, adding

up to *n* elements means reusing the same backing array; hence, reducing the number of allocations drastically.

The second option is to allocate `bars` with a given length:

```
func convert(foos []Foo) []Bar {
        n := len(foos)
        bars := make([]Bar, n)

        for i, foo := range foos {
                bars[i] = fooToBar(foo)
        }
        return bars
}
```

As we initialized the slice with a length, *n* elements were already allocated and initialized to the zero value of `Foo`. Hence, to set elements, we don't have to use `append` but `bars[i]`.

Which option is the best? Let's run a benchmark with the three solutions and an input slice of one million elements:

```
BenchmarkConvert_EmptySlice-4                 22     49739882 ns/op
BenchmarkConvert_GivenCapacity-4              86     13438544 ns/op
BenchmarkConvert_GivenLength-4                91     12800411 ns/op
```

As we can see, the first solution has a significant impact performance-wise. Keeping allocating array and copying elements makes the first benchmark almost 400% slower than the two other ones. Comparing the second and the third solutions, we can notice the third one is about 4% faster because we avoid repeated calls to the `append` built-in function, which has a small overhead compared to a direct assignment.

If setting a capacity and using `append` is less efficient than setting a length and assigning to a direct index, so why do we see it being used sometimes in Go projects? Let's look at a concrete example in Pebble, an open-source key-value store developed by CockroachDB.

A function called `collectAllUserKeys` needs to iterate over a slice of structs to format a particular byte slice. The resulting slice will be twice the length of the input slice:

```
func collectAllUserKeys(cmp Compare,
        tombstones []tombstoneWithLevel) [][]byte {
        keys := make([][]byte, 0, len(tombstones)*2)
        for _, t := range tombstones {
                keys = append(keys, t.Start.UserKey)
                keys = append(keys, t.End)
        }
        // ...
}
```

Here, the conscious choice was to use a given capacity and `append`. What's the rationale? If we had used a given length instead of a capacity, the code would have been the following:

```go
func collectAllUserKeys(cmp Compare,
        tombstones []tombstoneWithLevel) [][]byte {
        keys := make([][]byte, len(tombstones)*2)
        for i, t := range tombstones {
                keys[i*2] = t.Start.UserKey
                keys[i*2+1] = t.End
        }
        // ...
}
```

Notice how more complex the code looks to handle the slice index. Given that this function wasn't performance sensitive, it was decided to favor the easiest option to read.

Note

What if the future length of the slice isn't known precisely? For example, if the length of the output slice depends on a condition:

```go
func convert(foos []Foo) []Bar {
        // bars initialization

        for _, foo := range foos {
                if something(foo) {
                        continue
                }
                // Add a bar element
        }
        return bars
}
```

1

In that example, a `Foo` element is converted into a `Bar` and added to the slice only in a specific condition (`if`

> `something(foo)`). Should we initialize `bars` as an empty slice or with a given length or capacity? There's no strict rule there. It's a traditional software problem whether it's better to trade CPU or memory. Perhaps if `something(foo)` is true in 99% of the cases, it's worth initializing `bars` with a length or capacity? It depends on our use case.

Converting one slice type into another is a frequent operation for Go developers. As we have seen, if the length of the future slice is already known, there is no good reason to allocate at first an empty slice. The option is either to allocate a slice with a given capacity or with a given length. Between these two solutions, we have seen that the last one tends to be slightly faster. Yet, using a given capacity and `append` can be easier to implement and read in some contexts.

The next section will discuss the difference between nil and empty slices and why it matters for Go developers.

3.6 Being confused about nil vs. empty slice

It's fairly frequent for Go developers to mix nil and empty slices. We may want to use one over the other depending on the use case. Meanwhile, some libraries make the distinction between both. Hence, to be proficient with slices, we need to make sure we don't mix these concepts.

Before looking at an example, let's discuss definitions:

- A slice is empty if its length is equal to zero
- A slice is nil if it equals to `nil`

Let’s see different ways to initialize a slice. Can you guess what will be the output of this code? Each time, we will print whether the slice is empty or nil:

```
1  func main() {
2          var s []string
3          log(1, s)
4
5          s = []string(nil)
6          log(2, s)
7
8          s = []string{}
9          log(3, s)
10
11         s = make([]string, 0)
12         log(4, s)
13 }
14
15 func log(i int, s []string) {
16         fmt.Printf("%d: empty=%t\tnil=%t\n", i, len(s) == 0, s
17 == nil)
   }
```

1

This example will print the following:

```
1 1: empty=true   nil=true
2 2: empty=true   nil=true
3 3: empty=true   nil=false
4 4: empty=true   nil=false
```

All the slices are empty, meaning a length equals zero. Therefore, a nil slice is also an empty slice. However, only the first two are nil slices. So, if we have multiple ways to initialize a slice, which option should we favor?

Two things to note:

- First, one of the main differences between nil and an empty slice is regarding allocations. Indeed, initializing a nil slice doesn't require any allocation, which isn't the case for an empty slice.

- Second, regardless if a slice is nil, calling the `append` built-in function does work:

```
var s1 []string
fmt.Println(append(s1, "foo")) // [foo]
```

So, if a function returns a slice, for example, we shouldn't do as in other languages and return a non-nil collection for defensive reasons. As a nil slice doesn't require any allocation, we should favor returning a nil slice. Let's have a look at this function that returns a slice of strings:

```
func f() []string {
        var s []string
        if foo() {
                s = append(s, "foo")
        }
        if bar() {
                s = append(s, "bar")
        }
        return s
}
```

If both `foo` and `bar` are false, we will return an empty slice. Here, to prevent allocating an empty slice for no particular reason, we should favor option one (`var s []string`). One may use option four (`make([]string, 0)`) with a zero-length, but this doesn't bring any value compared to option one, and it requires an allocation which isn't the case here.

However, in the case we have to produce a slice with a known length, we should use option four (`s := make([]string, length[, capacity])`):

```
1 func intsToStrings(ints []int) []string {
2         s := make([]string, len(ints))
3         for i, v := range ints {
4                 s[i] = strconv.Itoa(v)
5         }
6         return s
7 }
```

Indeed, as discussed in *Inefficient slice initialization*, we should set the length (or capacity) in such a scenario to avoid extra allocations and copies.

Now, two different options remain:

- Option two: `s := []string(nil)`
- Option three: `s := []string{}`

This first option isn't the most widely used. Yet, it can be helpful for syntactic sugar as we can pass a nil slice in a single line. For example, using `append`:

```
s := append([]int(nil), 42)
```

If we had used option one (`var s []string`), it would have required two lines of code. This is probably not the most important readability optimization of all time; yet, it's still worth knowing.

Note

> We will see in *Not making slice copy correctly* one rationale to append to a nil slice.

Now, the latest option: option three with `s := []string{}`. This form is recommended to create a slice with initial elements. For example:

```
s := []string{"foo", "bar", "baz"}
```

However, if we don't need to create it with initial elements, we shouldn't use this option. Indeed, it brings the same benefits as option one (`var s []string`) except that the slice isn't nil; hence, it requires an allocation. Therefore, option three should be avoided without initial elements.

> **Note**
>
> Some linters can catch option three without initial values and recommend changing it in option one. However, let's note that it also changes the semantics from a non-nil to a nil slice.

We should also note that some libraries distinguish between nil and empty slices. This is the case, for example, with the `encoding/json` package. Let's have a look at the following examples where we will marshal two structs: one containing a nil slice, whereas the second one a non-nil empty slice:

```
var s1 []float32
        customer1 := customer{
        ID:          "foo",
        Operations: s1,
}
b, _ := json.Marshal(customer1)
```

```
fmt.Println(string(b))

s2 := make([]float32, 0)
        customer2 := customer{
        ID:         "bar",
        Operations: s2,
}
b, _ = json.Marshal(customer2)
fmt.Println(string(b))
```

```

```

Running this example, we can notice that the marshaling results for these two structs are different:

```
{"ID":"foo","Operations":null}
{"ID":"bar","Operations":[]}
```

A nil slice is marshaled as a `null` element, whereas a non-nil empty slice is marshaled as an empty array. If we work in the context of strict JSON clients that differentiate between `null` and `[]`, it's essential to keep this distinction in mind.

`encoding/json` isn't the only package from the standard library to make the distinction. For example, `reflect.DeepEqual` returns false if we compare a nil and a non-nil empty slice which is something to remember in the context of unit tests. In any case, while working with the standard library or external libraries, we should make sure that using one version or another doesn't lead to unexpected results.

To summarize, in Go, there is a distinction between nil and empty slices. A nil slice equals `nil`, whereas an empty slice has a zero length. A nil slice is

empty, but an empty slice isn't necessarily nil. Meanwhile, a nil slice doesn't require any allocation. We have seen throughout this section how to initialize a slice depending on the context:

- `var s []string` if we aren't sure about the final length and the slice can be empty
- `[]string(nil)` as syntactic sugar to create a nil and empty slice
- `make([]string, length[, capacity])` if the future length is known

The last option, `[]string{}`, should be avoided if we initialize the slice without elements.

Last but not least, let's make sure we check whether the libraries we use make the distinctions between nil and empty slices to prevent unexpected behaviors.

In the next section, we will continue this discussion and see the best way to check for an empty slice after having called a function.

3.7 Not properly checking if a slice is empty

We have seen in the previous section that there is a distinction between nil and empty slices. Having these notions in mind, what's the idiomatic way to check if a slice contains elements? Not having a clear answer can lead to subtle bugs.

Let's consider the following example. We will call a `getOperations` function that returns a slice of `float32`. Then, we want to call a `handle` function only if the slice contains elements. Here's a first (erroneous) version:

```go
func handleOperations(id string) {
	operations := getOperations(id)
	if operations != nil {
		handle(operations)
	}
}

func getOperations(id string) []float32 {
	operations := make([]float32, 0)

	if id == "" {
		return operations
	}

	// Add elements to operations

	return operations
}
```

1
2
3

In this example, checking if the slice has elements is done by checking if the `operations` slice isn't nil. Yet, there's a problem with this code. Indeed, `getOperations` never returns a nil slice; instead, it returns an empty slice. Therefore the `operations != nil` check will always be true.

So, what shall we do in this situation? One first approach might be to say that we need to modify `getOperations` to return a nil slice if `id` is empty:

```go
func getOperations(id string) []float32 {
	operations := make([]float32, 0)

	if id == "" {
		return nil
	}

	// Add elements to operations
```

```

        return operations
}
```

1

Instead of returning `operations` if `id` is empty, we return nil instead. This way, the check we had implemented about testing the slice nullity would match.

However, this approach isn't working in all situations. Indeed, we're not always in a context where we can change the callee. For example, if we use an external library, we won't create a PR just to change empty into nil slices.

So how could we check whether a slice is empty or nil? The solution is to check the length:

```go
func handleOperations(id string) {
        operations := getOperations(id)
        if len(operations) != 0 {
                handle(operations)
        }
}
```

1

We mentioned in the previous section that an empty slice has, by definition, a zero-length. Meanwhile, nil slices are always empty. Therefore, checking by checking the length of the slice, we cover all the scenarios:

- If the slice is nil: `len(operations) != 0` will be false

- If the slice isn't nil but empty: `len(operations) != 0` will also be false

Hence, checking the length is the best option to follow as we can't always control the approach taken by functions we call. Meanwhile, as the Go Wiki states, when designing interfaces, we should avoid distinguishing nil and empty slices, leading to subtle programming errors. Hence, when returning slices, we should make neither a semantic nor a technical difference if we return a nil or an empty slice. Both should mean the same thing for the callers.

This principle is the same with maps. To check if a map is empty, we should check its length, not whether it's nil.

In the next section, we will see how to make slice copies correctly.

3.8 Not making slice copy correctly

The `copy` built-in function allows copying elements from one source slice into a destination slice. Though it is a handy built-in function, Go developers can sometimes misunderstand it. Let's see a common mistake that results in copying the wrong number of elements.

In the following example, we will create a slice and copy its elements to another slice. What should be the output of this code?

```
1 src := []int{0, 1, 2}
2 var dst []int
3 copy(dst, src)
4 fmt.Println(dst)
```

If we run this example, it will print `[]`, not `[0 1 2]`. So what have we missed?

To use `copy` effectively, it's essential to understand that the number of elements copied to the destination slice corresponds to the minimum between:

- The source slice's length
- The destination slice's length

In the previous example, `src` is a 3-length slice but `dst` is a 0-length-slice as it was initialized to its zero value. Therefore, `copy` will copy the minimum between 3 and 0 elements: 0. The resulting slice is then empty.

If we want to perform a complete copy, the destination slice must have a length greater or equal to the source slice's length. Here, we will set up a length based on the source slice:

```
src := []int{0, 1, 2}
dst := make([]int, len(src))
copy(dst, src)
fmt.Println(dst)
```

```
1
```

As `dst` is now a slice initialized with a length equal to 3, it will copy three elements. This time, if we run this code, it will print `[0 1 2]`.

> **Note**
>
> Another common mistake is to invert the order of the arguments when calling `copy`. Remember that the destination is the former

argument, whereas the source is the latter.

Let's also mention that using the `copy` built-in function isn't the only way to copy slice elements. There are different alternatives, the most known being probably the following using `append`:

```
src := []int{0, 1, 2}
dst := append([]int(nil), src...)
```

We append the elements from the source slice to a nil slice. Hence, this code creates a 3-length, 3-capacity slice copy. This alternative has the advantage of being done in a single line. However, using `copy` is more idiomatic, hence easier to understand, even though it takes an extra line.

Copying elements from one slice to another is a reasonably frequent operation. When using `copy`, we must recall that the number of elements copied to the destination corresponds to the minimum between the two slice's lengths. Also, let's bear in mind that other alternatives exist to copy a slice, so we shouldn't be surprised if we find them in a codebase.

Let's keep discussing slices with a common mistake while using `append`.

3.9 Unexpected side-effects using slice append

This section will discuss a common mistake when using `append`, which may have unexpected side effects in some situations.

In the following example, we will initialize an `s1` slice, create `s2` by slicing `s1` and create `s3` by appending an element to `s2`:

```
1 s1 := []int{1, 2, 3}
2 s2 := s1[1:2]
3 s3 := append(s2, 10)
```

We initialized an `s1` slice containing three elements. `s2` is created from slicing `s1`. Then, we call `append` on `s3`. What should be the state of these three slices at the end of this code? Can you guess it?

Following the second line, right after `s2` is created, here's the state of both slices in memory:

Figure 3.10. Both slices are backed by the same array but with a different length and capacity.

`s1` is a 3-length, 3-capacity slice, while `s2` is a 1-length, 2-capacity slice, both backed by the same array as we already mentioned.

Adding an element using `append` checks whether the slice is full (length == capacity). If not full, the `append` function will add the element by updating

the backing array and returning a slice having a length incremented by one.

In this example, `s2` isn't full; it can accept one more element. Therefore, here's the final state of these three slices:

Figure 3.11. Again, all the slices are backed by the same array.

In the backing array, the last element was updated to store `10`. Thereby, if we print all the slices, here is the output:

```
s1=[1 2 10], s2=[2], s3=[2 10]
```

The `s1` slice's content was modified, even though we have updated neither `s1[2]` nor `s2[1]` directly. Hence, we should keep this in mind to avoid unintended consequences.

Let's see one impact of this principle while passing the result of a slicing operation to a function. We will initialize a slice with three elements and call a

function with only the first two elements:

```go
func main() {
        s := []int{1, 2, 3}

        f(s[:2])
        // Use s
}

func f(s []int) {
        // Update s
}
```

In this implementation, if `f` updates the first two elements, the changes will be visible to the slice in `main`. However, if `f` calls `append`, it will update the third element of the slice, even though we passed only two elements:

```go
func main() {
        s := []int{1, 2, 3}

        f(s[:2])
        fmt.Println(s) // [1 2 10]
}

func f(s []int) {
        _ = append(s, 10)
}
```

If we want to *protect* the third element for defensive reasons, meaning to make sure that `f` doesn't update it, we have two options.

The first option is to pass a copy of the slice and then construct the resulting slice:

```go
func main() {
        s := []int{1, 2, 3}
        sCopy := make([]int, 2)
        copy(sCopy, s)

        f(sCopy)
        result := append(sCopy, s[2])
        // Use result
}

func f(s []int) {
        // Update s
}
```

```
1
2
```

As we pass a copy to `f`, even if this function calls `append`, it will not lead to a side effect outside of the range of the first two elements. The downside of this solution is that it makes the code more complex to read and adds an extra copy which can be a problem if the slice is large.

Another solution can be used to limit the range of potential side-effects to the first two elements only. This solution involves the so-called *full slice expression*: `s[low:high:max]`. This statement creates a similar slice to one created with `s[low:high]` except that the resulting slice's capacity will be equal to `max - low`. Here's an example when calling `f`:

```go
func main() {
        s := []int{1, 2, 3}
        f(s[:2:2])
        // Use s
}

func f(s []int) {
        // Update s
}
```

```
1
```

Here, the slice passed to `f` isn't `s[:2]` but `s[:2:2]`. Hence, the slice's capacity is 2 - 0 = 2, as shown in the following figure:

Figure 3.12. `s[0:2]` **creates a 2-length, 3-capacity slice whereas** `s[0:2:2]` **creates a 2-length, 2-capacity slice.**

When passing `s[:2:2]`, we can solely limit the range of effects to the first two elements. Meanwhile, it also prevents from having to perform a slice copy.

When using slicing, we have to remember that we can face a situation leading to unintended side effects. Indeed, if the resulting slice has a length smaller than its capacity, `append` can mutate the original slice. If we want to restrict the range of possible side effects, we can either use a slice copy or the full slice expression, which prevents us from doing a copy.

In the next section, we will keep discussing slices but in the context of potential memory leaks.

3.10 Slice and memory leaks

This section will show that slicing an existing slice or array can lead to memory leaks in some conditions. We will discuss two cases: one case where the capacity is leaking and another related to pointers.

3.10.1 Capacity leak

For the first case, let's imagine implementing a custom binary protocol. A message can contain one million bytes, and the first five bytes represent the message type. In our code, we consume these messages, and for auditing purposes, we want to store the latest 1000 message types in memory. The skeleton of our function will be the following:

```
func consumeMessages() {
        for {
                msg := receiveMessage()
                // Do something with msg
                storeMessageType(getMessageType(msg))
        }
}

func getMessageType(msg []byte) []byte {
        return msg[:5]
}
```

The `getMessageType` function computes the message type by slicing the input slice. We test this implementation, and everything is fine. However, when we deploy our application, we notice that our application consumes about 1 GB of memory. How is it possible?

The slicing operation on `msg` using `msg[:5]` creates a 5-length slice. However, its capacity remains the same capacity as the initial slice. The remaining elements are still allocated in memory, even if eventually `msg` will not be referenced anymore. Let's see an example with a large message length of one million bytes:

Figure 3.13. After a new loop iteration, `msg` will not be used anymore. However, its backing array will still be used by `msg[:5]`.

As we can notice in this figure, the backing array of the slice still contains one million bytes after the slicing operation. Hence, if we keep in memory 1000 messages, instead of storing about 5 KB, we will hold about 1 GB.

So what can we do to solve this issue? By making a slice copy instead of slicing `msg`:

```
1 func getMessageType(msg []byte) []byte {
2         msgType := make([]byte, 5)
3         copy(msgType, msg)
4         return msgType
5 }
```

As we perform a copy, `msgType` is a 5-length, 5-capacity slice regardless of the size of the message received. Hence, we will store only 5 bytes per message type.

Note

Here, `getMessageType` would return a shrunk version of the initial slice: a 5-length, 5-capacity slice. Yet, would the GC be able to reclaim the inaccessible space from byte 5? The Go specification doesn't officially specify the behavior.

However, by using `runtime.Memstats`, we can records statistics about the memory allocator, such as the number of bytes allocated on the heap:

```
func printAlloc() {
        var m runtime.MemStats
        runtime.ReadMemStats(&m)
        fmt.Printf("%d KB\n", m.Alloc/1024)
}
```

If we call this function following a call to `getMessageType` and `runtime.GC()` to force running a garbage collection, we can notice that the inaccessible space isn't reclaimed. The whole backing array will still live in memory.

Therefore, using the full slice expression isn't a valid option (unless a future update of Go will tackle it).

What about using the full slice expression to solve this issue?

```
func getMessageType(msg []byte) []byte {
        return msg[:5:5]
}
```

As a rule of thumb, we have to remember that slicing a large slice or array can lead to potential high memory consumption. Indeed, the remaining space won't be reclaimed by the GC, and we can keep a large backing array, despite using only a few elements. Using slice copy is the solution to prevent such a case.

3.10.2 Slice and pointers

We have seen that slicing can cause a leak because of the slice capacity. Yet, what about the elements? Those are still part of the backing array but outside the length range. Does the GC collect them?

Let's delve into this question using a `Foo` struct containing a byte slice:

```
type Foo struct {
        v []byte
}
```

We want to check the memory allocations after each step:

- Allocate a slice of 1000 `Foo` elements
- Iterate over each `Foo` element, and allocate for each one 1 MB for the `v` slice

- Call a `keepFirstTwoElementsOnly` that will return only the first two elements using slicing, then call a GC

We want to see how memory behaves following the call to `keepFirstTwoElementsOnly` and a GC. Here's the scenario in Go (we will reuse the `printAlloc` function previously defined):

```go
func main() {
	foos := make([]Foo, 1_000)
	printAlloc()

	for i := 0; i < len(foos); i++ {
		foos[i] = Foo{
			v: make([]byte, 1024*1024),
		}
	}
	printAlloc()

	two := keepFirstTwoElementsOnly(foos)
	runtime.GC()
	printAlloc()
	runtime.KeepAlive(two)
}

func keepFirstTwoElementsOnly(foos []Foo) []Foo {
	return foos[:2]
}
```

1
2
3
4
5

We allocate the `foos` slice, allocate a slice of 1 MB for each element, then call `keepFirstTwoElementsOnly` and a GC. In the end, we use

`runtime.KeepAlive` to keep a reference to the `two` variable after the GC so that it won't be collected.

We may expect the GC to collect the 998 remaining `Foo` elements and the data allocated for the slice as these elements can't be accessed anymore. However, this isn't the case. For example, the code can print the following:

```
83 KB
1024072 KB
1024073 KB
```

1

The first step allocates about 83 KB of data. Indeed, we allocate 1000 zero values of `Foo`. The second step allocates 1 MB per slice, which increases memory. However, we can notice that the GC did not collect the remaining 998 elements after the last step. What's the reason?

The rule is the following, and it's essential to keep it in mind while working with slices: if the element is a pointer or a struct with pointer fields, the elements won't be reclaimed by the GC. Here, as `Foo` contains a slice (and a slice is a pointer on top of a backing array), the remaining 998 `Foo` elements and their slice won't be reclaimed. Therefore, even though these 998 elements can't be accessed anymore, they will stay in memory as long as the variable returned by `keepFirstTwoElementsOnly` is referenced.

So what are the options to ensure we don't leak the remaining `Foo` elements?

The first option, again, is to create a copy of the slice:

```go
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
        res := make([]Foo, 2)
        copy(res, foos)
        return res
}
```

As we copy the first two elements of the slice, the GC will know that the 998 elements won't be referenced anymore and can be collected by the GC.

There's a second option if we want to keep the underlying capacity of 1000 elements, for example, which is to mark the slices of the remaining elements to nil explicitly:

```go
func keepFirstTwoElementsOnly(foos []Foo) []Foo {
        for i := 2; i < len(foos); i++ {
                foos[i].v = nil
        }
        return foos[:2]
}
```

Here, we return a 2-length, 1000-capacity slice, but we set the slices of the remaining elements to nil. Hence, the GC will be able to collect the 998 backing arrays.

So, which option is the best? If we don't want to keep this capacity of 1000 elements, the first option is probably the best. However, the decision can also depend on the proportion of the elements. Let's see a visual example, assuming a slice containing `n` elements where we want to keep `i` elements:

Figure 3.14. Option one iterates until `i`, whereas option two iterates from `i`.

The first option creates a copy of `i` elements. Hence, it must iterate from element 0 to `i`. Meanwhile, the second option marks the remaining slices to nil, so it must iterate from element `i` to `n`. If performance is important and that `i` is closer to `n` than 0, we may consider the second option instead. It requires iterating over fewer elements (at least, it's probably worth benchmarking the two options).

In this section, we have seen two potential memory leak problems. The first one is about slicing an existing slice or array to preserve the capacity. If we handle large slices and reslice them to keep only a fraction of it, a lot of memory will remain allocated but unused. The second one is when we use the slicing operation with elements being a pointer or a struct with pointer fields, we should know that the GC won't reclaim these elements. In that case, the two options are to either perform a copy or mark the remaining elements or their fields to nil explicitly.

Now, let's start discussing maps in the context of initializations.

3.11 Inefficient map initialization

This section will discuss a similar issue that we already saw with slice initialization, yet using maps this time. But first, we need to know the basics regarding how maps are implemented in Go to understand why tweaking how to initialize a map is important.

3.11.1 Concepts

A map provides an unordered collection of key/value pairs in which all the keys are distinct.

In Go, a map is based on the hashtable data structure. Internally, a hashtable is an array of buckets, each bucket being a pointer to an array of key/value pairs as represented in the following figure:

Figure 3.15. A hashtable example with a focus on bucket 0.

An array of four elements backs this hashtable. If we delve into the array index, we notice one bucket consisting of a single key/value pair (element): "two": 2. Each bucket has a fixed size of eight elements.

Each operation (read, update, insert, delete) is done by associating a key to an array index. This step relies on a hash function. This function is stable as we want it to return the same bucket given the same key consistently. In the previous example, `hash("two")` returned 0; hence, the element was stored in the bucket referenced by the array index 0.

If we insert another element and hashing the key returns the same index, it will add another element to the same bucket:

Figure 3.16. `hash("six")` returned 0; hence, the element is stored to the same bucket

In case of insertion to a bucket already full (bucket overflow), Go will create another bucket of eight elements and link the previous bucket to it:

Figure 3.17. In case of a bucket overflow, Go will allocate a new bucket and link the previous bucket to it.

Regarding reads, updates, and deletes, it also requires calculating the corresponding array index. Then, Go will iterate sequentially over all the keys until it finds the provided one. Therefore, the worst-case time complexity for these three operations is *O(p)* with *p* the total number of elements in the buckets (one bucket by default, multiple buckets in case of overflows).

Let's now discuss why initializing a map efficiently is important.

3.11.2 Initialization

To understand the problems related to an inefficient map initialization, let's create a `map[string]int` type containing three elements:

```go
m := map[string]int{
        "1": 1,
        "2": 2,
        "3": 3,
}
```

Internally, the map will be backed by an array consisting of a single entry; hence, a single bucket. Now, what happens if we add one million elements? In that case, a single entry won't be enough as finding a key would mean, in the worst case, going over thousands of buckets. It's the reason why a map should be able to grow automatically to cope with the number of elements.

When a map grows, it will double its number of buckets. What are the conditions for a map to grow?

- If the average number of items in the buckets (called load factor) is greater than a constant value. This constant equals 6.5 (but may change in future versions as it's internal to Go).
- If there are too many buckets that are overflowed (containing more than eight elements).

When a map grows, all the keys will be dispatched again to all the buckets. It's the reason why in the worst-case scenario, inserting a key can be an *O(n)* operation, with *n* being the total number of elements in the map.

We have seen using slices that if we knew upfront the number of elements to be added in the slice, we could initialize it with a given size or capacity. It avoids having to keep repeating the costly slice growth operation. The idea is similar to maps. Indeed, we can use the `make` built-in function to provide an initial size while creating a map. For example, if we want to initialize a map that will contain one million elements, it can be done this way:

```
m := make(map[string]int, 1_000_000)
```

With a map, we can provide to `make` only an initial size, not a capacity as with slices; hence, a single argument.

By specifying a size, we give a hint as to the number of elements expected to go into the map. Internally, the map will be created with an appropriate number of buckets to store one million elements. Hence, it will save a lot of computation time as the map won't have to create buckets on the fly and handle buckets rebalancing.

Also, specifying a size *n* doesn't mean making a map with a maximum number of *n* elements: we can still add more than *n* elements if needed. Instead, it means asking the Go runtime to allocate a map with room for at least *n* elements, which is helpful if we already know the size upfront.

To understand why specifying a size is important, let's run two benchmarks. The former will insert one million elements in a map without setting an initial size, whereas the latter in a map initialized with a size.

```
BenchmarkMapWithoutSize-4              6    227413490 ns/op
BenchmarkMapWithSize-4                13     91174193 ns/op
```

The second version, with an initial size, is about 60% faster. Indeed, by providing a size, we prevent the map from keeping growing to cope with the inserted elements.

Therefore, just like with slices, if we know upfront the number of elements a map will contain, we should create it by providing an initial size. Doing this avoids potential map growth, which is quite heavy computation-wise as it requires reallocating enough space and rebalancing all the elements.

Let's keep discussing maps and see a common mistake leading to memory leaks.

3.12 Map and memory leaks

When working with maps in Go, we should understand some important characteristics about how a map grows and shrinks. Let's delve into these questions to prevent a common mistake leading to leaking memory.

First, to see a concrete example of this problem, let's design a scenario where we will work with the following map:

```
m := make(map[int][128]byte)
```

Each value of `m` will be an array of 128 bytes.

We will perform the following scenario:

- Allocate an empty map
- Add one million elements
- Remove all the elements and run a GC

After each step, we want to print the size of the heap (using MB this time) to see how this example behaves memory-wise:

```go
// Init
n := 1_000_000
m := make(map[int][128]byte)
printAlloc()

// Add elements
for i := 0; i < n; i++ {
        m[i] = randBytes()
}
printAlloc()

// Remove elements
for i := 0; i < n; i++ {
        delete(m, i)
}

// End
runtime.GC()
printAlloc()
runtime.KeepAlive(m)
```

First, we allocate an empty map, add one million elements, remove one million elements and then run a GC. We also make sure to keep a reference to the map using `runtime.KeepAlive` so that the map isn't collected as well. Let's run this example:

```
0 MB
461 MB
293 MB
```

```
3
```

What can we observe? At first, the heap size is minimal. Then it grows significantly after having added one million elements to the map. However, if we expected the heap size to decrease after removing all the elements, this isn't how maps work in Go. In the end, even though the GC has collected all the elements, the heap size is still 293MB large. So the memory shrunk, but not as we might have expected. What's the rationale?

We discussed in the previous section that a map was composed of 8-element buckets. Under the hood, a Go map is a pointer to a `runtime.hmap` struct. This struct contains multiple fields, including a `B` field giving the number of buckets in the map:

```
type hmap struct {
        B uint8 // log_2 of # of buckets
                        // (can hold up to loadFactor * 2^B
items)
        // ...
}
```

After adding one million elements, the value of `B` equals 18, which means 2^18 = 262,144 buckets. Then, when we remove one million elements, what's the value of `B`? Still 18. Hence, the map still contains the same number of buckets.

The reason is that the number of buckets in a map cannot shrink. Therefore, removing elements from a map doesn't impact the number of existing buckets; it just zeroes the slots in the buckets. A map can only grow and have more buckets; it will never shrink.

In the previous example, we went from 461 MB to 293 MB because the elements were collected but running the GC didn't impact the map itself. Even the number of extra buckets (the buckets created because of overflows) remains the same.

Let's take a step back and discuss when the fact that a map cannot shrink can be a problem.

Imagine building a cache using a `map[int][128]byte`. This map holds per customer ID (int) a sequence of 128 bytes. Now, suppose we want to save the last 1000 customers. The map size will remain constant, so we shouldn't worry that a map cannot shrink.

However, let's now say that we want to store one hour of data. Meanwhile, our company has decided to make a big promotion for Black Friday, and for one hour, we can have millions of customers connected to our system. In this case, even after a few days, once Black Friday is over, our map will contain the same number of buckets as during peak time. Hence, it explains why we can experience a high memory consumption that doesn't significantly decrease in such a scenario.

So, what are the solutions if we don't want to manually restart our service to clean the amount of memory consumed by the map?

One solution could be to recreate a copy of the current map at a regular pace. For example, every hour, we build a new map, copy all the elements, and release the previous one. The main drawback is that following the copy and until the next GC, we may consume twice the current memory for a short period.

Another solution can be to change the map type to store an array pointer: `map[int]*[128]byte`. It doesn't solve the fact that we will have an important number of buckets; however, each bucket entry will reserve the size of a pointer for the value instead of 128 bytes (hence, 8 bytes on 64-bit systems, 4 bytes on 32-bit systems). Coming back to the original scenario, let's compare the memory consumption for each map type following each step:

Table 3.2. Memory consumption in MB

Step	`map[int][128]byte`	map[int]*[128]byte
Allocate an empty map	0 MB	0 MB
Add one million elements	461 MB	182 MB
Remove all the elements and run a GC	293 MB	38 MB

As we can notice, after removing all the elements, the amount of required memory is significantly less important with a `map[int]*[128]byte` type. Also, in this case, during peak time, the amount of required memory is less significant due to some optimizations to reduce the memory consumed.

Note

> We should also note that if a key or a value is over 128 bytes, Go won't store it directly in the map bucket. Instead, Go will store a pointer referencing the key or the value.

As we have seen, adding *n* elements in a map and then deleting all the elements means keeping the same number of buckets in memory. Hence, we must remember that a Go map can only grow in size, so is its memory consumption. There is no automated strategy to shrink it. If this leads to high memory consumption, we can try different options such as forcing to recreate the map or using pointers to check if it can optimize it.

For the last section of this chapter, let's discuss comparing values in Go.

3.13 Comparing values incorrectly

Comparing values is a common operation in software development. Be it in a function to compare two objects or in testing to compare a value against an expectation, implementing comparisons is quite frequent. Our first instinct could be to use the `==` operator everywhere. Yet, as we will see in this section, this shouldn't always be the case. So when is it appropriate to use `==`, and what are the alternatives?

Let's start with a concrete example. We will create a basic `customer` struct and use `==` to compare two instances. What should be the output of this code, in your opinion?

```
type customer struct {
        id string
}

func main() {
        cust1 := customer{id: "x"}
        cust2 := customer{id: "x"}

```

```
        fmt.Println(cust1 == cust2)
}
```

Comparing these two `customer` structs is a valid operation in Go, and it will print true. Now, what happens if we make a slight modification to the `customer` struct to add a slice field:

```
type customer struct {
        id         string
        operations []float64
}

func main() {
        cust1 := customer{id: "x", operations: []float64{1.}}
        cust2 := customer{id: "x", operations: []float64{1.}}
        fmt.Println(cust1 == cust2)
}
```

```
1
```

We might expect this code to print true as well. However, it doesn't even compile:

```
invalid operation:
    cust1 == cust2 (struct containing []float64 cannot be compared)
```

The problem relates to how `==` and `!=` operators work. This operator doesn't work with slices or maps. Hence, as the `customer` struct contains a slice, it doesn't compile.

It’s essential to understand how to use `==` and `!=` to make comparisons effectively. We can use them on operands that are *comparable*:

- Booleans: Compare whether two booleans are equals
- Numerics (int, float, and complex types): Compare whether two numerics are equals
- Strings: Compare whether two strings are equals
- Channels: Compare whether two channels were created by the same call to `make` or if both are nil
- Interfaces: Compare whether two interfaces have identical dynamic types and equal dynamic values or if both have value nil
- Pointers: Compare whether two pointers point to the same value in memory or if both are nil
- Structs and arrays if they are composed of comparable types

> **Note**
>
> We can also use the `⇐`, `>=`, `<` and `>` operators with numeric types to compare the values and strings to compare their lexical order.

In the last example, our code failed to compile as the struct was composed on a non-comparable type (a slice).

We should also know the possible issues of using `==` and `!=` with `any` types. For example, comparing two integers assigned to `any` types is allowed:

```
var a any = 3
var b any = 3
```

```
fmt.Println(a == b)
```

This code prints:

```
true
```

However, what if we initialize two `customer` types (the latest version containing a slice field) and assign the values to `any` types?

```
var cust1 any = customer{id: "x", operations: []float64{1.}}
var cust2 any = customer{id: "x", operations: []float64{1.}}
fmt.Println(cust1 == cust2)
```

This code compiles. Yet, as both types can't be compared as the `customer` struct contains a slice field, it leads to an error at runtime, a panic:

```
panic: runtime error: comparing uncomparable type main.customer
```

Having these behaviors in mind, what are the options if we have to compare two slices, two maps, or two structs containing non-comparable types? If we stick with the standard library, the other option is to use runtime reflection with the `reflect` package.

Reflection is a form of metaprogramming, and it refers to the ability of an application to introspect and modify its structure and behavior.

For example, in Go, we can use `reflect.DeepEqual`. This function reports whether two elements are *deeply equal* by traversing recursively two values. The elements it accepts are basic types plus array, struct, slice, map, pointer, interface, and functions.

> **Note**
>
> The `reflect.DeepEqual` has a specific behavior depending on the type we provide. Before using it, let's read the documentation carefully.

Let's rerun the first example, using `reflect.DeepEqual:`

```
cust1 := customer{id: "x", operations: []float64{1.}}
cust2 := customer{id: "x", operations: []float64{1.}}
fmt.Println(reflect.DeepEqual(cust1, cust2))
```

Here, even though the `customer` struct contains non-comparable types (slice), it will operate as expected and print true.

However, there are two things to keep in mind while using `reflect.DeepEqual`. The first one is that it makes the distinction between an empty and a nil collection, as discussed in *Being confused about nil vs. empty slice*. Is this a problem? Not necessarily; it depends on our use case. For example, if we want to compare the result of two unmarshaling operations (e.g., from JSON to a Go struct), we may wish for this difference to be raised. Yet, it's worth having this behavior in mind to use effectively `reflect.DeepEqual`.

The other catch is something pretty standard in most of the languages. As this function uses reflection, which will introspect at runtime the values to discover how they are formed, it has a performance penalty. Doing a few benchmarks locally with structs of different sizes, on average, `reflect.DeepEqual` is about 100 times slower than `==`. This might be a reason to favor using it in the context of testing instead of runtime.

If performance is a crucial factor, another option might be to implement our own comparison method. Here's an example to compare two `customer` structs and return a boolean:

```
func (a customer) equal(b customer) bool {
        if a.id != b.id {
                return false
        }
        if len(a.operations) != len(b.operations) {
                return false
        }
        for i := 0; i < len(a.operations); i++ {
                if a.operations[i] != b.operations[i] {
                        return false
                }
        }
        return true
}
```

In this code, we built our comparison method with custom checks on the different fields of the `customer` struct. Running a local benchmark on a slice composed of 100 elements shows that our custom `equal` method is about 96 times faster than `reflect.DeepEqual`.

In general, we should remember that the `==` operator is pretty limited. For example, it doesn't work with slices and maps. In most cases, using `reflect.DeepEqual` is a solution, but the main catch is the performance penalty. In the context of unit tests, some other options are possible such as using external libraries with `go-cmp` [9] or `testify` [10]. However, if performance is crucial at runtime, implementing our custom method can be the best solution. One additional note: we should also remember that the standard library has some existing comparison methods. For example, we can use the optimized `bytes.Compare` function to compare two slices of bytes. Let's make sure we don't reinvent the wheel.

3.14 Chapter summary

- When reading an existing code, we should bear in mind that integer literals starting by 0 are octal numbers. Also, to improve readability, we should make octal integers explicit by prefixing them with `0o`.
- As integer overflows and underflows are handled silently in Go, we can implement our own functions to catch them.
- Making floating-point comparisons within a given delta can ensure that a code is portable.
- When performing additions or substractions, we should group the ones with a similar order of magnitude to favor accuracy. Also, we should perform multiplications and divisions before additions and substractions.
- Understanding the difference between slice length and capacity should be part of Go developers' core knowledge. The slice length is the number of available elements in the slice, whereas the slice capacity is the number of elements in the backing array.
- When creating a slice, we should initialize it with a given length or capacity if its length is already known. It will reduce the number of allocations and improve performance. The same logic goes for maps by initializing their size.
- Using copy or the full slice expression is a way to prevent `append` from creating conflicts if two different functions use slices backed by the same array. However, only a slice copy prevents memory leaks if we want to shrink a large slice.
- To copy one slice to another using the `copy` built-in function, we must recall that the number of copied elements corresponds

to the minimum between the two slice's lengths.

- Working with a slice of pointers or structs with pointer fields, we can avoid memory leaks by marking to nil the elements excluded by a slicing operation.
- To prevent common confusions such as when using the `encoding/json` or the `reflect` package, we must understand the difference between nil and empty slices. Both are 0-length, 0-capacity slices, but only a nil slice doesn't require allocations.
- To check if a slice doesn't contain any element, we should check its length. This check works regardless of whether the slice was nil or empty. The same goes for maps.
- To design unambiguous APIs, we shouldn't distinguish nil and empty slices.
- A map can always grow in memory, never shrink. Hence, if it leads to some memory issues, we can try different options such as forcing to recreate the map or using pointers.
- To compare types in Go, we can use the `==` and `!=` operators if two types are comparable: boolean, numerics, strings, pointers, channels, and structs composed of comparable types solely. Otherwise, we have to use `reflect.DeepEqual` and pay the price of reflection or use custom implementations/libraries.

[8] github.com/stretchr/testify

[9] github.com/google/go-cmp

[10] github.com/stretchr/testify

4 Control structures

This chapter covers

- Understanding how a `range` loop assigns the element values and evaluate the provided expression
- Dealing with `range` loop and pointers
- Preventing common map iteration and loop breaking mistakes
- Using `defer` inside a loop

Control structures in Go are similar to those in C or Java but differ in significant ways. For example, there is no `do` or `while` loop, only a generalized `for`. This chapter will delve into the most common mistakes related to control structures with a strong focus on the `range` loop, a common source of misunderstanding.

4.1 Ignoring that elements are copied in `range` loops

A `range` loop is a convenient way to iterate over various data structures. We don't have to handle an index and the termination state. Go developers may forget or be unaware of how a `range` loop assigns the values, leading to common mistakes. First, let's remind us how to use a `range` loop; next, we'll delve into how values are assigned.

4.1.1 Concepts

A `range` loop allows iterating over different data structures:

- String
- Array
- Pointer to an array
- Slice
- Map
- Receiving channel

Compared to a classic `for` loop, a `range` loop is a convenient way to iterate over all the elements of one of these data structures thanks to its concise syntax. Also, it's less error-prone as we don't have to handle manually the condition expression and the iteration variable, which may avoid errors such as off-by-one errors. Here is an example with an iteration over a slice of strings:

```
s := []string{"a", "b", "c"}
    for i, v := range s {
            fmt.Printf("index=%d, value=%s\n", i, v)
    }
```

This code loops over each element of the slice. In each iteration, as we iterate over a slice, `range` produces a pair of values: an index and an element value, respectively assigned to `i` and `v`. In general, `range` produces two values for each data structure except for a receiving channel that produces a single element, the value.

In some cases, we may be only interested in the element value, not in the index. As not using a local variable would lead to a compilation error, we can instead use the blank identifier to replace the index variable like so:

```
s := []string{"a", "b", "c"}
    for _, v := range s {
            fmt.Printf("value=%s\n", v)
    }
```

Thanks to the blank identifier, we iterate over each element by ignoring the index and assigning only the element value to `v`.

If we're not interested in the value, we can omit the second element like so:

```
for i := range s {}
```

Now that we refreshed our minds on using a `range` loop, let's see what kind of value is returned during an iteration.

4.1.2 Value copy

Understanding how the value is handled during each iteration is critical to effectively use a `range` loop. Let's see how it works with a concrete example.

We will create an `account` struct containing a single `balance` field:

```
type account struct {
            balance float32
    }
```

Next, we will create a slice of `account` structs and iterate over each element using a `range` loop. During each iteration, we will increment the `balance` of

each `account`:

```
accounts := []account{
            {balance: 100.},
            {balance: 200.},
            {balance: 300.},
    }
    for _, a := range accounts {
            a.balance += 1000
    }
```

Following this code, what should be the slice's content, in your opinion?

- `[{100} {200} {300}]`
- `[{1100} {1200} {1300}]`

The answer is the former: `[{100} {200} {300}]`. In this example, the `range` loop did not affect the slice's content. Let's understand why.

In Go, everything we assign is a copy. For example, if we assign the result of a function returning:

- A struct, it will perform a copy of this struct
- A pointer, it will perform a copy of the memory address (an address is 64-bit long on a 64-bit architecture)

It's crucial to have this in mind to avoid common mistakes, including those related to `range` loops. Indeed, when a `range` loop iterates over a data structure, it performs a copy of each element to the value variable (the second item).

Coming back to our example, when we iterated over each `account` element, it resulted in a struct copy assigned to the value variable `a`. Therefore, when we incremented the balance with `account.balance += 1000`, it mutated the value variable solely (`a`), not an element in the slice.

So what if we want to update the slice elements? There are two main options.

The first option is to access the element using the slice index. It can be achieved with both a classic `for` loop or a `range` loop using the index instead of the value variable:

```
1 for i := range accounts {
2             accounts[i].balance += 1000
3     }
4
5     for i := 0; i < len(accounts); i++ {
6             accounts[i].balance += 1000
7     }
```

Use the index variable to access the element of the slice 1
Use the traditional for loop 2

In the previous example, both iterations had the same effect: updating the elements inside of the `accounts` slice.

Which one should we favor? It depends on the context. If we want to go over each element, the first loop is shorter to write and read. Yet, if we need to control which element we want to update (e.g., one out of two), we should instead use the second loop.

Note

Another option could be to keep using the `range` loop and access the value but to modify the slice type to a slice of `account` pointers:

```
accounts := []*account{
            {balance: 100.},
            {balance: 200.},
            {balance: 300.},
    }
    for _, a := range accounts {
            a.balance += 1000
    }
```

Update the slice type to []*account 1
Update the slice elements directly 2

In this case, as we mentioned, the `a` variable is a copy of the `account` pointer stored in the slice. However, as both pointers reference the same struct, the `a.balance += 1000` statement will update the slice element itself.

However, such an option has two main downsides. First, it requires updating the slice type, which may not always be possible. Second, if performance is important, we should note that iterating over a slice of pointers may be less efficient for a CPU because of lack of predictability (we will delve into this point in *Not understanding CPU caches*).

In general, we should remember that the value element in a `range` loop is a copy. Therefore, if the value is a struct we need to mutate, we will only update the copy, not the element itself, unless the value or the field we modify is a pointer. The favored options are to access the element via the index using a `range` loop or a classic `for` loop.

In the next section, we will keep working with `range` loops and see how the provided expression is evaluated.

4.2 Ignoring how arguments are evaluated in `range` loops

The `range` loop syntax requires an expression. For example, in `for i, v := range exp`, `exp` is the expression. As we have seen, it can be a string, an array, a pointer to an array, a slice, a map, or a channel. Now, let's discuss the following question: how is this expression evaluated? It's an essential point of `range` loops to avoid common mistakes when using a `range` loop.

Let's look at the following example that appends an element to a slice we iterate over. Do you believe the loop will terminate?

```
s := []int{0, 1, 2}
    for range s {
            s = append(s, 10)
    }
```

To understand this question, we should know that when using a `range` loop, the provided expression is evaluated only once, before the beginning of the loop. In this context, evaluated means that the provided expression is copied to a temporary variable, and then `range` will iterate over this variable, not the original one.

In this example, when the `s` expression is evaluated, it will result in a slice copy:

Figure 4.1. `s` is copied to a temporary variable used by `range`.

The `range` loop will use this temporary variable. Also, the original slice `s` is updated during each iteration. Hence, after three iterations, the state is the following:

Figure 4.2. The temporary variable remains a 3-length slice, hence the iteration completes.

Each step resulted in appending a new element. However, after three steps, we went over all the elements. Indeed, the temporary slice used by `range` remains a 3-length slice. Hence, the loop completes after three iterations.

The behavior would have been different with a classic `for` loop:

```
s := []int{0, 1, 2}
    for i := 0; i < len(s); i++ {
            s = append(s, 10)
    }
```

In this example, the loop never ends. Indeed, the `len(s)` expression is evaluated during each iteration. As we keep adding elements, we will never reach a termination state. It's essential to keep this difference in mind to use Go loops accurately.

Coming back to the `range` operator, we should know that the behavior we described (expression evaluated only once) also applies to all the data types provided. As an example, we will see the implication of this behavior with two other types: channels and arrays.

4.2.1 Channels

Let's see a concrete example based on iterating on a channel using a `range` loop.

We will create two goroutines, both sending elements to two distinct channels. Then, in the parent goroutine, we will implement a consumer on one channel using a `range` loop that will try switching to the other channel during the iteration:

```
ch1 := make(chan int, 3)
    go func() {
            ch1 <- 0
            ch1 <- 1
            ch1 <- 2
            close(ch1)
```

```
7       }()
8
9       ch2 := make(chan int, 3)
10      go func() {
11              ch2 <- 10
12              ch2 <- 11
13              ch2 <- 12
14              close(ch2)
15      }()
16
17      ch := ch1
18      for v := range ch {
19              fmt.Println(v)
20              ch = ch2
21      }
```

Create a first channel that will contain elements 0, 1 and 2 1
Create a second channel that will contain elements 10, 11 and 12 2
Assign to ch the first channel 3
Create a channel consumer by iterating over ch 4
Assign to ch the second channel 5

In this example, the same logic regarding how the `range` expression is evaluated applies. The expression provided to `range` is a `ch` channel pointing to `ch1`. Hence, `range` evaluates `ch`, performs a copy to a temporary variable, and iterates over elements from this channel. Despite the `ch = ch2` statement, `range` will keep iterating over `ch1`, not `ch2`:

```
1 0
2     1
3     2
```

The `ch = ch2` statement isn't effectless, though. As we assigned `ch` to the second variable, if we call `close(ch)` following this code, it will close the second channel, not the first one.

Let's now see the impacts with an array.

4.2.2 Array

What's the impact of using a `range` loop with an array? As the `range` expression is evaluated before the beginning of the loop, what is assigned to the temporary loop variable is a copy of the array. Let's see this principle in action with the following example that updates a specific array index during the iteration:

```
1 a := [3]int{0, 1, 2}
2     for i, v := range a {
3             a[2] = 10
4             if i == 2 {
5                     fmt.Println(v)
6             }
7     }
```

Create an array of three elements 1
Iterate over the array 2
Update the last index 3
Print the content of the last index 4

This code updates the last index to 10. However, if we run this code, it will not print 10 but 2 as described in the following figure:

Figure 4.3. `range` iterates over the array copy (left) while the loop modifies `a` (right).

As we mentioned, the `range` operator creates a copy of the array. Meanwhile, the loop doesn't update the copy; it updates the original array: `a`. Therefore, the value of `v` during the last iteration is 2, not 10.

If we wanted to print the actual value of the last element, it can be achieved either:

- By accessing the element from its index:

```
a := [3]int{0, 1, 2}
    for i := range a {
            a[2] = 10
            if i == 2 {
                    fmt.Println(a[2])
            }
    }
```

Access a[2] instead of the range value variable As we access the original array, this code will print 2 instead of 10. 1

- Using an array pointer:

```
a := [3]int{0, 1, 2}
    for i, v := range &a {
            a[2] = 10
            if i == 2 {
                    fmt.Println(v)
            }
    }
```

Range over &a instead of a We assigned a copy of the array pointer to the temporary variable used by range. Yet, as both pointers reference the same array, accessing v also returns 10. 1

Both options are valid. However, the second option doesn't lead to copying the whole array, which may be something to keep in mind in case of the array is significantly large.

In summary, the `range` loop evaluates the provided expression only once, before the beginning of the loop, by doing a copy (regardless of the type). We should remember this behavior to avoid common mistakes that would lead us to access the wrong element, for example.

In the next section, we will see how to avoid common mistakes using `range` loops with pointers.

4.3 Ignoring the impacts of using pointer elements in `range` loops

This section will delve into a specific mistake when using a `range` loop with pointer elements. If we're not cautious enough, it can lead us to an issue

where we would reference the wrong elements. Let's understand this problem and how to fix it.

Before starting, let's clarify the rationale for using a slice or map of pointer elements? There are three main cases:

- In terms of semantics, storing data using pointer semantics implies sharing the element. For example, with the following method holding the logic to insert an element in a cache:

```
type Store struct {
            m map[string]*Foo
    }

    func (s Store) Put(id string, foo *Foo) {
            s.m[id] = foo
            // ...
    }
```

  Here, using the pointer semantics implies that the `Foo` element is shared by both the caller of `Put` and the `Store` struct.

- Sometimes, we already manipulate pointers. Hence, it can be handy to store directly in our collection pointers instead of values.

- Another case if we store large structs and these structs are frequently mutated, to avoid a copy and an insertion for each mutation, we can use pointers instead:

```
func updateMapValue(mapValue map[string]LargeStruct, id string) {
            value := mapValue[id] #1
```

```
            value.foo = "bar"
            mapValue[id] = value #2
    }

    func updateMapPointer(mapPointer map[string]*LargeStruct, id
string) {
            mapPointer[id].foo = "bar" #3
    }
```

As `updateMapPointer` accepts a map of pointers, the mutation of the `foo` field can be done in a single step.

Now, it's time to discuss the common mistake with pointer elements in `range` loops. We will consider the two following structs:

- A `Customer` struct representing a customer
- A `Store` that holds a map of `Customer` pointers

```
type Customer struct {
            ID      string
            Balance float64
    }

    type Store struct {
            m map[string]*Customer
    }
```

Meanwhile, we will consider the following method that iterates over a slice of `Customer` elements and store them into the `m` map:

```
func (s *Store) storeCustomers(customers []Customer) {
            for _, customer := range customers {
                    s.m[customer.ID] = &customer
            }
    }
```

Store customer pointer in the map 1

In this example, we iterated over the input slice using the `range` operator and stored `Customer` pointers into the map. However, does this method do what we expect?

Let's give it a try by calling it with a slice of three different `Customer` structs:

```
s.storeCustomers([]Customer{
            {ID: "1", Balance: 10},
            {ID: "2", Balance: -10},
            {ID: "3", Balance: 0},
    })
```

Here's the result of this code if we print the map:

```
key=1, value=&main.Customer{ID:"3", Balance:0}
    key=2, value=&main.Customer{ID:"3", Balance:0}
    key=3, value=&main.Customer{ID:"3", Balance:0}
```

As we can notice, instead of storing three different `Customer`, all the elements stored in the map references the same `Customer` struct: `3`. So what have we done wrong?

When we iterate over the `customers` slice using the `range` loop, regardless of the number of elements, it will create a single `customer` variable with a fixed address. We can verify this by printing the pointer address during each iteration:

```
```

```
func (s *Store) storeCustomers(customers []Customer) {
            for _, customer := range customers {
                    fmt.Printf("%p\n", &customer)
                            s.m[customer.ID] = &customer
            }
    }
```

Print the customer address 1

```
0xc000096020
    0xc000096020
    0xc000096020
```

Why is it important? Let's examine how during each iteration:

- During the first iteration, `customer` references the first element: `Customer 1`. We store a pointer of `customer`.
- During the second iteration, `customer` now references another element: `Customer 2`. We also store a pointer of `customer`.
- Finally, during the last iteration, `customer` references the last element: `Customer 3`. Again, the same pointer is stored in the map.

Figure 4.4. The `customer` variable has a constant address, so we store in the map the same pointer.

At the end of the iterations, we stored three times the same pointer in the map. This pointer's last assignment was a reference to the slice's last element: `Customer 3`. It's why all the elements of the map reference the same `Customer`.

So how do we fix this problem? There are two main solutions.

The first one is similar to what we saw in *Unintended variable shadowing*; it requires creating a local variable:

```
1 func (s *Store) storeCustomers(customers []Customer) {
2             for _, customer := range customers {
3                     current := customer
4                     s.m[current.ID] = &current
5             }
6     }
```

Create a local current variable 1
Store this pointer in the map 2

In this example, we don't store a pointer referencing `customer` but `current`. `current` is a variable referencing a unique `Customer` during each iteration. Therefore, following the loop, we have stored different pointers referencing different `Customer` structs in the map.

The other solution is to store a pointer referencing each element using the slice index:

```
1 func (s *Store) storeCustomers(customers []Customer) {
2             for i := range customers {
3                     customer := &customers[i]
4                             s.m[customer.ID] = customer
5             }
6     }
```

Assign to customer a pointer of the i element 1
Store the customer pointer 2

In this solution, `customer` is now a pointer. As it's initialized during each iteration, it has a unique address. Therefore, we will store in the maps different pointers.

When iterating over a data structure using a `range` loop, we must recall that all the values are assigned to a unique variable with a single unique address. Therefore, if we store a pointer referencing this variable during each iteration, we will end up in a situation where we store the same pointer referencing the same element: the latest one. We can overcome this issue by forcing the creation of a local variable in the loop's scope or creating a pointer referencing a slice element via its index. Both solutions are fine. Please also note that we took a slice data structure as an input, but the problem would be similar with a map.

In the next section, we will see common mistakes related to map iteration.

4.4 Making wrong assumptions during map iterations

Iterating over a map is a common source of misunderstanding and mistakes, mostly because developers take wrong assumptions. In this example, we will discuss two different cases:

- Ordering
- Map update during an iteration

This section will see two common mistakes based on wrong assumptions while iterating over a map.

4.4.1 Ordering

Regarding ordering, we should understand a few fundamental behaviors about the map data structure:

- It doesn't keep the data sorted by key (a map isn't based on a binary tree).
- It doesn't preserve the order the data was added in. For example, if we insert pair A before pair B, we shouldn't make any assumption based on this insertion order.

Furthermore, when iterating on a map, we shouldn't make any ordering assumption at all. Let's understand the implications of this statement.

We will consider the following map consisting of four buckets (the elements represent the key):

Figure 4.5. A map with four buckets.

Each index of the backing array points references a given bucket. Now, let's iterate over this map using a `range` loop and print all the keys:

```
for k := range m {
            fmt.Print(k)
    }
```

We mentioned that the keys aren't sorted by key. Hence, we can't expect this code to print `acdeyz`.

Meanwhile, we said the map doesn't preserve the insertion order. Hence, we also can't expect this code to print `ayzcde`.

Yet, can we at least expect this code to print the keys in the order they are currently stored in the map, `aczdey`? No, not even this. In Go, the iteration order over a map is *not specified*. Also, there is no guarantee that the order will be the same from one iteration to the next. We should keep these map's behaviors in mind to not base our code on wrong assumptions.

We can confirm all of these statements by running the previous loop twice:

```
zdyaec
    czyade
```

As we can notice, the order was different from one iteration to another.

Note

> Though there is no guarantee about the iteration order, the iteration distribution isn't uniform. It's the reason why the official Go specification states that the iteration is unspecified, not random.

So why does Go have such a surprising way to iterate over a map? It was a conscious choice from the language designers. They wanted to add some form of randomness to make sure developers never rely on any ordering assumption while working with maps[11].

Hence, as Go developers, we should never make assumptions regarding ordering while iterating on a map. However, let's note that using packages either from the standard library or external ones can lead to different behaviors. For example, when the `encoding/json` package marshals a map into JSON, it will reorder the data alphabetically by keys, regardless of the insertion order. Yet, this isn't a property of the Go map itself. If ordering is necessary, we should rely on other data structures such as a binary heap (the GoDS[12] library contains helpful data structures implementations).

Let's now move to the second mistake related to updating a map while iterating over it.

4.4.2 Map insert during iteration

In Go, updating a map (inserting or deleting an element) during an iteration is allowed; it doesn't lead to a compilation or a runtime error. However, there's another aspect that we should consider while adding an entry in a map during an iteration. Otherwise, it can lead to non-deterministic results.

Let's check the following example that iterates on a `map[int]bool`. If the pair value is true, we will add another element. Can you guess what will be the

output of this code?

```go
m := map[int]bool{
            0: true,
            1: false,
            2: true,
    }

    for k, v := range m {
            if v {
                    m[10+k] = true
            }
    }

    fmt.Println(m)
```

The result of this code is unpredictable. For example, if we run this code multiple times:

```
map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true]
    map[0:true 1:false 2:true 10:true 12:true 20:true 22:true 30:true 32:true]
    map[0:true 1:false 2:true 10:true 12:true 20:true]
```

To understand the reason, we have to read what the Go specification states in case of a new map entry during an iteration:

> If a map entry is created during iteration, it may be produced during the iteration or skipped. The choice may vary for each entry created and from one iteration to the next.
>
> -- Go spec

Hence, when an element is added to a map during an iteration, it may be produced during a follow-up iteration, or it may not. As Go developers, we don't have any way to enforce the behavior. Also, it may vary from one iteration to another, which is why we had three times a different result.

It's essential to have this behavior in mind to ensure our code doesn't produce unpredictable outputs. If we want to update a map while iterating over it and make sure the added entries aren't part of the iteration, one solution is to work on a copy of the map like so:

```
m := map[int]bool{
        0: true,
        1: false,
        2: true,
    }
    m2 := copyMap(m)

    for k, v := range m {
        m2[k] = v
        if v {
            m2[10+k] = true
        }
    }

    fmt.Println(m2)
```

Create a copy of the initial map 1
Update m2 instead of m 2

In this example, we dissociated the map being read from the map being updated. Indeed, we kept iterating over `m`, but the updates were done on `m2`. This new version creates a predictable and repeatable output:

```
map[0:true 1:false 2:true 10:true 12:true]
```

To summarize, when we work with a map, we shouldn't rely on:

- The ordering of the data by keys
- The preservation of the insertion order
- A deterministic iteration order
- The fact that an element added during an iteration will be produced during this iteration

Having these behaviors in mind should help us avoid common mistakes based on wrong assumptions.

In the next section, we will see a fairly frequent mistake made while breaking loops.

4.5 Ignoring how the `break` statement work

A `break` statement is commonly used to terminate the execution of a loop. When loops are used in conjunction with `switch` or `select`, it's frequent to see developers make the mistake of breaking the wrong statement.

Let's take a look at the following example. We will implement a `switch` inside a `for` loop. If the loop index has the value 2, we want to break the loop:

```
for i := 0; i < 5; i++ {
            fmt.Printf("%d ", i)

            switch i {
            default:
            case 2:
                    break
            }
    }
```

If i equals to 2, call break 1

This code may look right at first sight; however, it doesn't do what we expected. Indeed, the `break` statement doesn't terminate the `for` loop but the `switch` statement. Hence, this code iterates from 0 to 4, not 0 to 2: `0 1 2 3 4`.

One essential rule to keep in mind: a `break` statement terminates the execution of the innermost `for`, `switch`, or `select` statement. In the previous example: the `switch` statement.

So how can we write such a code that breaks the loop instead of the `switch` statement? The most idiomatic way is to use a label:

```
loop:
        for i := 0; i < 5; i++ {
                fmt.Printf("%d ", i)

                switch i {
                default:
                case 2:
                        break loop
                }
        }
```

Define a loop label 1
Terminate the loop attached to the loop label, not the switch 2

Here, we associate the `loop` label to the `for` loop. Then, as we provide to the `break` statement the `loop` label, it breaks the loop, not the switch. Therefore, this new version will print `0 1 2`, as we were expecting.

Note

One may challenge whether a break with a label is idiomatic and see it as a fancy goto statement. However, this isn't the case, and such a code is used in standard library. For example, in the `net/http` package while reading lines from a buffer:

```
readlines:
            for {
                    line, err :=
rw.Body.ReadString('\n')
                    switch {
                    case err == io.EOF:
                            break readlines
                    case err != nil:

t.Fatalf("unexpected error reading from CGI:
%v", err)
                    }
                    // ...
            }
```

This example uses an expressive label with `readlines` to emphasize the loop's goal.

Hence, we should consider breaking a statement using labels as an idiomatic approach in Go.

Breaking the wrong statement can also occur with a `select` inside a loop. In the following code, we want to use `select` on two cases and break the loop if the context cancels:

```
for {
        select {
        case <-ch:
                // Do something
        case <-ctx.Done():
                break
        }
    }
```

Break if the context cancels 1

Here the innermost `for`, `switch`, or `select` statement is the `select` statement, not the `for` loop. Hence, the loop will repeat. Again, to break the loop itself, we can do it using a label:

```
loop:
        for {
                select {
                case <-ch:
                        // Do something
                case <-ctx.Done():
                        break loop
                }
        }
```

Define a loop label 1
Terminate the loop attached to the loop label, not the select 2

Now, as expected, the `break` statement breaks the loop, not `select`.

> **Note**
>
> We can also use `continue` with a label to go to the next iteration of the labeled loop.

We should remain cautious while using a `switch` or `select` statement inside a loop. When using `break`, we should always make sure to know which statement it will affect. As we have seen, using labels is the idiomatic solution to enforce breaking a specific statement.

In the last section of this chapter, we will keep discussing loops, but this time in conjunction with the `defer` keyword.

4.6 Using `defer` inside a loop

The `defer` statement delays a call's execution until the surrounding function returns. It's mainly used to reduce boilerplate code. For example, if a resource has to be closed eventually, we can use `defer` to avoid repeating the closure calls before every single `return`. However, one common mistake is to be unaware of the consequences of using `defer` inside a loop. Let's look into this problem.

We will implement a function that opens a set of files where the file paths will be received via a channel. Hence, we will have to iterate over this channel, open the files and handle the closure. Here's our first version:

```
1  func readFiles(ch <-chan string) error {
2              for path := range ch {
3                      file, err := os.Open(path)
4                      if err != nil {
5                              return err
6                      }
7
8                      defer file.Close()
9
10                     // Do something with file
11             }
12             return nil
13     }
```

Iterate over the channel 1
Open the file 2
Defer the call to file.Close() 3

Note

We will discuss how to handle defer errors in *Not handling defer errors*.

There is a significant problem with this implementation. Indeed, we have to recall that `defer` schedules a function call when the **surrounding** function returns. In this case, the defer calls will be executed not during each loop iteration but when the `readFiles` function returns. If `readFiles` doesn't return, the file descriptors will be kept open forever, causing leaks.

What are the options to fix this problem?

One option can be to get rid of `defer` and handle the file closure manually. However, it means that we would have to abandon a convenient feature of the Go toolset just because we are in a loop. So what are the options if we want to keep using `defer`? We have to create another surrounding function around `defer` that would be called during each iteration.

For example, we can implement a `readFile` holding the logic per new file path received:

```
1  func readFiles(ch <-chan string) error {
2              for path := range ch {
3                      if err := readFile(path); err != nil {
4                              return err
5                      }
6              }
7              return nil
8      }
9
10     func readFile(path string) error {
11             file, err := os.Open(path)
12             if err != nil {
13                     return err
14             }
15
16             defer file.Close()
17
18             // Do something with file
19             return nil
20     }
```

Call the readFile function that contains the main logic 1
Keep the call to defer 2

In this implementation, the `defer` function is called when `readFile` returns, meaning at the end of each iteration. Therefore, we will not keep file descriptors opened until the parent `readFiles` function returns.

Another approach could be to make the `readFile` function a closure:

```
func readFiles(ch <-chan string) error {
        for path := range ch {
                err := func() error {
                        // ...
                        defer file.Close()
                        // ...
                }()
                if err != nil {
                        return err
                }
        }
        return nil
}
```

Run the provided closure 1

Yet, intrinsically, it remains the same solution: adding another surrounding function to execute the `defer` calls during each iteration. The plain old function has the advantage of being probably a bit clearer, and we can also write a specific unit test for it.

When using `defer`, we must remember that it schedules a function call when the surrounding function returns. Hence, calling `defer` within a loop will stack all the calls: they won't be executed during each iteration, which may cause memory leaks if the loop doesn't terminate, for example. The most convenient approach to solving this problem is introducing another function to be called during each iteration. Yet, if performance is crucial, one downside is the overhead added by the function call. If we are in this case and we want

to prevent such an overhead, we should get rid of `defer` and handle the defer call manually before looping.

Tour livebook

Take our tour and find out more about liveBook's features:

- Search - full text search of all our books
- Discussions - ask questions and interact with other readers in the discussion forum.
- Highlight, annotate, or bookmark.

take the tour

4.7 Chapter summary

- The value element in a `range` loop is a copy. Therefore, to mutate a struct, for example, we need to access it via its index or a classic `for` loop (unless the element or the field we want to modify is a pointer).
- Understanding that the expression passed to the `range` operator is evaluated only once before the beginning of the loop can help us avoid common mistakes such as inefficient assignment in channel or slice iteration.
- Using either a local variable or accessing an element using an index, we can prevent mistakes while copying pointers inside a loop.

- To ensure predictable outputs while using maps can be done by understanding that a map data structure:

    - Doesn't order the data by keys.
    - Doesn't preserve the insertion order.
    - Doesn't have a deterministic iteration order.
    - Doesn't guarantee that an element added during an iteration will be produced during this iteration.
- Using `break` or `continue` with a label enforces breaking a specific statement. It can be helpful with `switch` or `select` statements inside of loops.
- Extracting a loop logic inside a function will lead to a `defer` statement to be executed at the end of each iteration.

[11] github.com/golang/go/commit/d1f627f2f3f6fc22ed64e1cc7b17eefca952224b

[12] github.com/emirpasic/gods

5 Strings

This chapter covers

- Understanding the fundamental concept of the rune in Go
- Preventing common mistakes with string iteration and trimming
- Avoiding inefficient code because of string concatenations or useless conversions
- Avoiding memory leaks with substrings

In Go, a string is an immutable data structure representing:

- A pointer to an immutable byte slice
- The total number of bytes in this array

We will see in this chapter that Go has a pretty unique way to deal with strings and introduces a concept called runes. This concept is essential to understand and might confuse newcomers. Once we get how strings are managed, we can avoid common mistakes while iterating on a string. We will also see common mistakes made by Go developers while using or producing strings. In addition, we will see that sometimes we can work directly with `[]byte`, avoiding extra allocations. Lastly, we will see how to avoid a common mistake to create leaks from substrings. Helping you understand how strings work in Go by presenting common string mistakes is the primary goal of this chapter.

5.1 Not understanding the concept of rune

We couldn't start this chapter about strings without discussing the concept of the rune in Go. As you will see in the next sections of this chapter, this concept is key to thoroughly understanding how strings are handled and avoiding common mistakes. But first, before delving into Go runes, we need to make sure we are aligned about some fundamental programming concepts.

We should understand the distinction between charset and encoding:

- A charset, as the name suggests, is a set of characters. For example, the Unicode charset contains 2^21 characters.
- An encoding is the translation of a character's list in binary. For example, UTF-8 is an encoding standard capable of encoding all the Unicode characters in a variable number of bytes: from 1 to 4 bytes.

We mentioned characters to simplify the charset definition. Yet, in Unicode, we use the concept of *code point* to refer to an item represented by a single value. For example, the 汉 character is identified by the U+6C49 code point. Using UTF-8, 汉 is encoded using three bytes: 0xE6, 0xB1, and 0x89.

Why is it important? Because in Go, a rune *is* a Unicode code point.

Meanwhile, we mentioned that UTF-8 encodes characters from one to four bytes. Hence, up to 32 bits. It's the reason why in Go, a rune is an alias of `int32`:

```
type rune = int32
```

Something else to highlight about UTF-8: some people believe that Go strings are always UTF-8, yet this isn't true. Let's consider the following example:

```
s := "hello"
```

We assigned to `s` a string literal (a string constant). In Go, a source code is encoded in UTF-8. Hence, all the string literals will be encoded into a sequence of bytes using UTF-8. However, a string is a sequence of arbitrary bytes; it's not necessarily based on UTF-8. Hence, when we manipulate a variable that wasn't initialized from a string literal (e.g., reading from the filesystem), we can't necessarily expect it uses the UTF-8 encoding.

> **Note**
>
> `golang.org/x` , a repository that provides extensions to the standard library, contains packages to work with UTF-16 and UTF-32.

Let's get back to the `hello` example. We had a string composed of five characters: h, e, l, l, and o. These *simple* characters are encoded using a single byte each. It's the reason why getting the length of `s` returns five:

```
s := "hello"
    fmt.Println(len(s)) // 5
```

Yet, a character isn't always encoded into a single byte. Coming back to the 汉 character, we mentioned that with UTF-8, this character is encoded into three bytes. We can validate it with the following example:

```
s := "汉"
    fmt.Println(len(s)) // 3
```

Instead of printing 1, this example prints 3. Indeed, the `len` built-in function applied on a string doesn't return the number of characters; it returns the number of bytes.

Conversely, we can create a string from a list of bytes. We mentioned that the 汉 character was encoded using three bytes, 0xE6, 0xB1, and 0x89:

```
s := string([]byte{0xE6, 0xB1, 0x89})
    fmt.Printf("%s\n", s)
```

Here, we built a string composed of these 3 bytes. When we print the string, instead of printing three characters, it only prints a single one: 汉.

In summary:

- A charset is a set of characters, whereas an encoding describes how to translate a charset into binary.
- In Go, a string references an immutable slice of arbitrary bytes.
- A Go source code is encoded using UTF-8. Hence, all string literals are UTF-8 strings. Yet, as a string can contain arbitrary bytes, if it's obtained from another place (not the source code), it isn't guaranteed to be based on the UTF-8 encoding.
- A rune corresponds to a Unicode code point concept, meaning an item represented by a single value.
- Using UTF-8, a Unicode code point can be encoded from one to four bytes.
- Using `len` on a string in Go returns the number of bytes, not the number of runes.

Having these concepts in mind is essential as runes as everywhere in Go. Let's see a concrete application of this knowledge with a common mistake related to string iteration.

5.2 Inaccurate string iteration

Iterating on a string is a common operation for developers. Perhaps, we want to retrieve all the runes or implement a custom function to search for a specific substring. In both cases, we have to iterate on the different runes of a string. Yet, it's easy to get confused about how an iteration works.

Let's delve into a concrete example. Here, we want to print the different runes and their corresponding positions:

```
s := "hêllo"
    for i := range s {
            fmt.Printf("position %d: %c\n", i, s[i])
    }
    fmt.Printf("len=%d\n", len(s))
```

The string literal contains a special rune: ê 1

We use the `range` operator to iterate over `s`, and then, we want to print the rune using its index in the string:

```
position 0: h
    position 1: Ã
    position 3: l
    position 4: l
    position 5: o
    len=6
```

This code doesn't do what we wanted initially. Let's highlight three points:

- The second rune isn't `ê` but `Ã`.
- We went from position 1 to position 3, so what do we have at position 2?
- `len` returns a count of six, whereas `s` contains only five runes

Let's start with the latest observation. We already mentioned that `len` wasn't returning the number of runes in a string but the number of bytes. As we assigned to `s` a string literal, `s` is a UTF-8 string. Meanwhile, the

special character `ê` isn't encoded in a single byte; it requires two bytes. Therefore, calling `len(s)` returns six.

Note

What if we want to get the number of runes in a string, not the number of bytes? It depends on the encoding. In the previous example, as we assigned a string literal to `s`, it's a UTF-8 string. Hence, we can use the `unicode/utf8` package:

```
fmt.Println(utf8.RuneCountInString(s)) // 5
```

Let's get back to the iteration to understand the remaining surprises:

```
for i := range s {
            fmt.Printf("position %d: %c\n", i, s[i])
    }
```

We have to understand that in this example, we don't iterate over each rune; instead, we iterate over each starting index of a rune:

Figure 5.1. Printing `s[i]` prints the UTF-8 representation of each byte at index `i`

Printing `s[i]` doesn't print the `i` th rune; it prints the UTF-8 representation of the byte at index `i`. Hence, we printed `hÃllo` instead of `hêllo`. So how do we fix it if we want to print all the different runes? We have to use the value element of the `range` operator:

```
s := "hêllo"
    for i, r := range s {
            fmt.Printf("position %d: %c\n", i, r)
    }
```

Instead of printing the rune using `s[i]`, we use the `r` variable. Indeed, using a `range` loop on a string returns two variables: the starting index of a rune and the rune itself:

```
position 0: h
    position 1: ê
    position 3: l
    position 4: l
    position 5: o
```

The other approach is to convert the string into a slice of runes and iterate over it:

```
s := "hêllo"
    runes := []rune(s)
    for i, r := range runes {
            fmt.Printf("position %d: %c\n", i, r)
    }
```

```
position 0: h
    position 1: ê
    position 2: l
    position 3: l
    position 4: o
```

Here, we converted `s` into a slice of runes using `[]rune(s)`. Then, we iterate over this slice and use the value element of the `range` operator to print all the runes. The only difference is about the position: instead of printing the starting index of the rune's bytes sequence, it prints the rune's index directly.

Note that this solution introduces a runtime overhead compared to the previous one. Indeed, converting a string into a slice of rune requires allocating an additional slice and converting the bytes into runes; hence, an *O(n)* time complexity with `n` the number of bytes in the string. Therefore, if we want to iterate over all the runes, we should use the first solution.

However, if we want to access the `i` th rune of a string with the first option, we don't have access to the rune index but the starting index of a rune in the byte sequence. Hence, we should favor the second option in most cases:

```
s := "hêllo"
    r := []rune(s)[4]
    fmt.Printf("%c\n", r) // o
```

This code prints the fourth rune by first converting the string into a rune slice.

Note

One optimization if a string is composed of single-byte runes, for example containing letters `A` to `Z` and `a` to `z`. We can access the `i` th rune without converting the whole string into a slice of runes by accessing the byte directly using `s[i]`:

```
s := "hello"
    fmt.Printf("%c\n", rune(s[4])) // o
```

In summary, if we want to iterate over a string's runes, we can use the `range` loop on the string directly. Yet, we have to recall that the index doesn't correspond to the rune index but the starting index of the bytes sequence of the rune. As a rune can be composed of multiple bytes, if we want to access to the rune itself, we should use the value variable of `range`, not the index in the string. Meanwhile, if we are interested in getting the `i` th rune of a string, we should convert the string into a slice of runes in most cases.

In the next section, let's see a common confusion when using trim functions in the `strings` package.

5.3 Misusing trim functions

One common mistake made by Go developers when using the `strings` package is to mix `TrimRight` and `TrimSuffix`. Indeed, both functions serve a similar purpose, and it can be fairly easy to confuse them. Let's take a look at it.

In the following example, we will use `TrimRight`. What should be the output of this code?

```
fmt.Println(strings.TrimRight("123oxo", "xo"))
```

The answer is `123`. Was it what you were expecting? If not, you were probably expecting the result of `TrimSuffix` instead. Let's review both functions.

`TrimRight` removes all the trailing runes contained in a given set. In our example, we passed as a set `xo`, which contains two runes: `x` and `o`. Here's the logic:

Figure 5.2. Iterate backward until it finds a rune not part of the set.

- 123oxo
  - o is part of the set, remove 🗑
- 123ox
  - x is part of the set, remove 🗑
- 123o
  - o is part of the set, remove 🗑
- 123
  - 3 is **not** part of the set, stop 🛑

`TrimRight` iterates backward over each rune. If a rune is part of the provided set, it will remove it. If not, it stops the iteration and returns the remaining string. This is why this function returns `123`.

On the other side, `TrimSuffix` returns a string without a provided trailing suffix:

```
fmt.Println(strings.TrimSuffix("123oxo", "xo"))
```

As `123oxo` finishes by `xo`, this code prints `123o`. Also, removing the trailing suffix isn't a repeating operation. Hence, `TrimSuffix("123xoxo", "xo")` returns `123xo`

The principle is the same for the left-hand side of a string with `TrimLeft` and `TrimPrefix`:

```
fmt.Println(strings.TrimLeft("oxo123", "ox")) // 123
    fmt.Println(strings.TrimPrefix("oxo123", "ox")) /// o123
```

`string.TrimLeft` removes all the leading runes contained in a set, hence printing `123`. Moreover, `TrimPrefix` removes the provided leading prefix, hence printing `o123`.

One last thing to note related to this topic, `Trim` applies both `TrimLeft` and `TrimRight` on a string. Hence, it removes all the leading and trailing runes contained in a set:

```
fmt.Println(strings.Trim("oxo123oxo", "ox")) // 123
```

In summary, we have to make sure to understand the difference between `TrimRight` / `TrimLeft`, and `TrimSuffix` / `TrimPrefix`:

- `TrimRight` / `TrimLeft` removes the trailing/leading runes in a set
- `TrimSuffix` / `TrimPrefix` removes a given suffix/prefix

In the next section, we will delve into string concatenation.

5.4 Under-optimized strings concatenation

When it comes to concatenating strings, in Go, there are two main ways to do it, and one can be really inefficient in some conditions. Let's delve into this topic to understand the option we should favor and in which case.

Let’s write a `concat` function that will concatenate all the string elements of a slice using the `+=` operator.

```
1 func concat(values []string) string {
2             s := ""
3             for _, value := range values {
4                     s += value
5             }
6             return s
7     }
```

During each iteration, the `+=` operator concatenates `s` with the `value` string. At first sight, this function may not look wrong. Yet, with this implementation, we forget one of the core characteristics of a string: its immutability. Therefore, each iteration doesn’t update `s`; it reallocates a new string in memory, which significantly impacts the performance of this function.

Fortunately, there is a solution to deal with this problem, using the `strings` package and the `Builder` struct:

```
1 func concat(values []string) string {
2             sb := strings.Builder{}
3             for _, value := range values {
4                     _, _ = sb.WriteString(value)
5             }
6             return sb.String()
7     }
```

Creates a strings.Builder 1
Append a string 2
Returns the resulted string 3

First, we created a `string.Builder` struct using its zero value. During each iteration, we constructed the resulting string by calling the `WriteString` method that appended the content of `value` to its internal buffer; hence, minimizing memory copying.

Note that `WriteString` returns an error as the second output, but we purposely ignored it. Indeed, this method will never return a non-nil error. So what's the purpose for this method to return an error as part of its signature? `strings.Builder` implements the `io.StringWriter` interface, which contains a single method: `WriteString(s string) (n int, err error)`. Hence, to comply with this interface, `WriteString` had to return an error.

> **Note**
>
> We will discuss in more depth how to ignore errors idiomatically in *Ignoring an error*.

Using `string.Builder`, we can also append:

bookmark">
open discussion">
show text addons">
action">

- A byte slice using `Write`
- A single byte using `WriteByte`
- A single rune using `WriteRune`

Internally `strings.Builder` holds a byte slice. Each call to `WriteString` results in a call to `append` on this slice. There are two impacts. First, this struct shouldn't be used concurrently as the calls to `append` would lead to race conditions. The second impact is something that we already saw in *Inefficient slice initialization*: if the future length of a slice is already known, we should preallocate it. For that purpose, `strings.Builder` exposes a method: `Grow(n int)` to guarantee space for another n bytes.

Let's write another version of the `concat` method by calling `Grow` with the total number of bytes:

```
1  func concat(values []string) string {
2              total := 0
3              for i := 0; i < len(values); i++ {
4                      total += len(values[i])
5              }
6
7              sb := strings.Builder{}
8              sb.Grow(total)
9              for _, value := range values {
10                     _, _ = sb.WriteString(value)
11             }
12             return sb.String()
13     }
```

Iterate over each string to compute the total number of bytes 1
Call Grow with this total 2

Before the iteration, we compute the total number of bytes the final string will contain and assign the result to `total`. Note that we're not interested in the number of runes but the number of bytes so we use the `len` function. Then, we call `Grow` to guarantee space for `total` bytes before iterating over the strings.

Let's run a benchmark to compare the three versions (v1: using `+=`, v2: `strings.Builder{}` without preallocation, and v3: `strings.Builder{}` with preallocation). The input slice contains 1k strings, and each string contains 1k bytes:

```
BenchmarkConcatV1-4          16          72291485 ns/op
BenchmarkConcatV2-4        1188            878962 ns/op
BenchmarkConcatV3-4        5922            190340 ns/op
```

As we can see, the latest version is by far the most efficient: 99% faster than the v1 and 78% faster than the v2. We may ask ourselves: how iterating twice on the input slice could make the code faster? The answer lies in *Inefficient slice initialization*: if a slice isn't allocated with a given length or capacity, it will keep growing each time the slice is full, resulting in additional allocations and copies. Hence, it's the most efficient option in this case.

`strings.Builder` is the recommended solution to concatenate a list of strings. Usually, this solution should be used within a loop. Indeed, if we just have to concatenate a few strings (e.g., a name and a surname), it's not recommended to use it as it will make the code a bit less readable than using the `+=` operator or `fmt.Sprintf`.

As a general rule, we can remember that performance-wise, the `string.Builder` solution is faster from the moment we have to concatenate more than about five strings. Even though this exact number depends on many factors, such as the size of the concatenated strings and the machine, it can be a rule of thumb to remember to help us decide when to choose one solution over the other. Also, we shouldn't forget that if the

number of bytes of the future string is known in advance, we should use the `Grow` method to preallocate the internal byte slice.

Next, we will discuss the `bytes` package and why it may prevent useless string conversions.

5.5 Useless string conversion

When choosing to work with a string or with a `[]byte`, most programmers tend to favor strings for convenience. Yet, most I/O is actually done with `[]byte`. For example, `io.Reader`, `io.Writer`, and `io.ReadAll` work with `[]byte`, not strings. Hence, working with strings means extra conversions, although the `bytes` package contains many of the same operations as the `strings` package.

Let's see an example of what we shouldn't do. We will implement a `getBytes` function that takes an `io.Reader` as an input, reads from it, and calls a `sanitize` function. The sanitization will be done by trimming all the leading and trailing white spaces. Here's the skeleton of `getBytes`:

```
1 func getBytes(reader io.Reader) ([]byte, error) {
2             b, err := io.ReadAll(reader)
3             if err != nil {
4                     return nil, err
5             }
6             // Call sanitize
7     }
```

b is a []byte 1

We call `ReadAll` and assign the byte slice to `b`. Then, how could we implement the `sanitize` function? One option might be to create a `sanitize(string) string` function using the `strings` package:

```
func sanitize(s string) string {
            return strings.TrimSpace(s)
    }
```

Now, back to `getBytes`, as we manipulate a `[]byte`, we must convert it first to a string before calling `sanitize`. Then, we have to convert back the results into a `[]byte` as `getBytes` return a byte slice:

```
return []byte(sanitize(string(b))), nil
```

What's the problem with this implementation? We have to pay the extra price of converting a `[]byte` into a string and then convert a string into a `[]byte`. Memory-wise, each of these conversions requires an extra allocation. Indeed, even though a string is backed by a `[]byte`, converting a `[]byte` into a string requires a copy of the byte slice. It means a new memory allocation and a copy of all the bytes.

Note

We can test the fact that creating a string from a `[]byte` leads to a copy with the following code:

```
b := []byte{'a', 'b', 'c'}
    s := string(b)
    b[1] = 'x'
    fmt.Println(s)
```

Running this code doesn't print `axc` but `abc`. Indeed, in Go, a string is immutable.

So, how should we implement the `sanitize` function? Instead of accepting and returning a string, we should manipulate byte slice instead:

```
func sanitize(b []byte) []byte {
            return bytes.TrimSpace(b)
    }
```

The `bytes` package also has a `TrimSpace` function to trim all the leading and trailing white spaces. Now, on the caller side, it doesn't require any extra conversions:

```
return sanitize(b), nil
```

As we mentioned, most I/O is done with `[]byte`, not strings. When we're wondering whether we should work with strings or `[]byte`, let's recall that working with `[]byte` isn't necessarily less convenient. Indeed, all the exported functions of the `strings` package also have their alternative in the `bytes` package: `Split`, `Count`, `Contains`, `Index`, etc. Hence, be it while doing I/O or else, we should first check whether we couldn't

implement a whole workflow using bytes instead of strings and avoiding the price of additional conversions.

The last section of this chapter will discuss how the substring operation can sometimes lead to memory leak situations.

5.6 Substring and memory leaks

In *Slice and memory leaks*, we have seen how slicing a slice or array may lead to memory leak situations. This principle also applies to string and substring operations. First, we will see how substrings are handled in Go and prevent memory leaks.

To extract a subset of a string, we can use the following syntax:

```
s1 := "Hello, World!"
    s2 := s1[:5] // Hello
```

`s2` is constructed as a substring of `s1`. This example creates a string from the first five bytes, not the first five runes. Hence, we shouldn't use this syntax in the case of runes encoded with multiple bytes. Instead, we should convert the input string into a `[]rune` type first:

```
s1 := "Hêllo, World!"
    s2 := string([]rune(s1)[:5]) // Hêllo
```

Now that we refreshed our minds regarding the substring operation, let's delve into a concrete problem to illustrate possible memory leaks.

We will receive log messages as strings. Each log will be formatted with a UUID first (36 bytes) followed by the message itself. We want to store in memory these UUIDs, for example, to keep a cache of the latest *n* UUIDs. We should also note that these log messages can be potentially quite heavy (up to thousands of bytes). Here is our implementation:

```
func (s store) handleLog(log string) error {
            if len(log) < 36 {
                    return errors.New("log is not correctly formatted")
            }
            uuid := log[:36]
            s.store(uuid)
            // Do something
    }
```

To extract the UUID, we use a substring operation with `log[:36]` as we know that the UUID is encoded on 36 bytes. Then, we pass this `uuid` variable to the `store` method that will store it in memory. Is this solution problematic? Yes, it is.

When doing a substring operation, the Go specification doesn't specify whether the resulting string and the one involved in the substring operation should share the same data. However, the standard Go compiler does let them share the same backing array, which is probably the best solution memory and performance wise as it prevents a new allocation and a copy.

We mentioned that the log messages could be quite heavy. `log[:36]` will create a new string referencing the same backing array. Therefore, each `uuid` string that we will store in memory will not contain only 36 bytes but the number of bytes in the initial `log` string; hence potentially thousands of bytes.

How can we fix it? By making a deep copy of the substring so that the internal byte slice of `uuid` references a new backing array of 36 bytes solely:

```
1 func (s store) handleLog(log string) error {
2             if len(log) < 36 {
3                     return errors.New("log is not correctly
4 formatted")
5             }
6             uuid := string([]byte(log[:36]))
7             s.store(uuid)
8             // Do something
    }
```

Perform a []byte, then a string conversion 1

The copy is performed by converting the substring into a `[]byte` first, then into a string again. By doing this, we prevent a memory leak from occurring. The `uuid` string will be backed by an array consisting of only 36 bytes.

Let's note that some IDEs or linters may highlight that the `string([]byte(s))` conversions aren't necessary. For example, GoLand, the JetBrains IDE, warns about a redundant type conversion. This is true in the sense that we convert a string into a string, yet this operation has an actual effect. As discussed, it prevents the new string from being backed

by the same array as `uuid`. Let's be aware that the warnings raised by IDEs or linters might sometimes be inaccurate.

> **Note**
>
> As a string is mostly a pointer, calling a function to pass a string doesn't result in a deep copy of the bytes. The copied string will still reference the same backing array. Hence, we shouldn't consider the length of a string when calling a function. Furthermore, the way to do a deep copy is `string([]byte(s))`.

As of Go 1.18, the standard library also includes a solution with `strings.Clone` that returns a fresh copy of a string:

```
uuid := strings.Clone(log[:36])
```

Calling `strings.Clone` makes a copy of `log[:36]` into a new allocation, preventing any memory leak.

Two things to keep in mind while using substring in Go. First, the interval provided is based on the number of bytes, not the number of runes. Second, it may lead to a memory leak as the substring will share the same backing array as the initial string. The solutions to prevent this case from happening are to perform a string copy manually or to use `strings.Clone` from Go 1.18.

5.7 Chapter summary

- Understanding that a rune corresponds to the concept of Unicode code point and that it can be composed of multiple bytes should be part of the Go developer's core knowledge to work accurately with strings.
- Iterating on a string with the `range` operator iterates on the runes with the index corresponding to the starting index of the rune's bytes sequence. If we want to access a specific rune index (e.g., the third rune), we should convert the string into a `[]rune`.
- `strings.TrimRight` / `strings.TrimLeft` removes all the trailing/leading runes contained in a given set, whereas `strings.TrimSuffix` / `strings.TrimPrefix` returns a string without a provided suffix/prefix.
- Concatenating a list of strings should be made with `strings.Builder` to prevent allocating a new string during each iteration.
- Remembering that the `bytes` package offers the same operations as the `strings` package can help avoid extra bytes/string conversions.
- Using copies instead of substrings can prevent memory leaks as the string returned by a substring operation will be backed by the same byte array.

6 Functions and method

This chapter covers

- Discussing when to use value or pointer receivers
- Exploring when using named result parameter makes sense and being aware of potential side effects
- Avoiding a common mistake while returning a nil receiver in a function or method
- Discussing why functions accepting filename isn't a best practice
- Understanding how defer arguments are handled

A function wraps a sequence of statements into a unit that can be called elsewhere. It can take some input(s) and produces some output(s). On the other hand, a method is a function attached to a given type. The attached type is called a receiver and can be a pointer or a value. We will start this chapter by discussing how to choose one receiver type or the other as it's usually a source of debate. Then, we will discuss named parameters, when shall we use them, and why they may sometimes lead to mistakes. We will also discuss common mistakes when designing a function or returning specific values such as a nil receiver.

6.1 Not knowing which type of receiver to use

Choosing a receiver type for a method isn't always straightforward. When should we use value receivers? When should we use pointer receivers? We will see in this section what the conditions are to make the right decision.

In the optimizations chapter, we will thoroughly discuss values vs. pointers. Hence, this section will only scratch the surface performance-wise. Also, in many contexts, using a value or a pointer receiver shouldn't be dictated by performance but by other conditions that we will discuss. But first, let's refresh our minds about how receivers work.

In Go, we can attach either a value or a pointer receiver to a method. With a value receiver, Go makes a copy of the value and passes it to the method. Any changes to the object will remain local to the method. The original object will remain unchanged.

As an illustration, the following example mutates a value receiver:

```
1  type customer struct {
2              balance float64
3      }
4
5      func (c customer) add(v float64) {
6              c.balance += v
7      }
8
9      func main() {
10             c := customer{balance: 100.}
11             c.add(50.)
12             fmt.Printf("balance: %.2f\n", c.balance)
13     }
```

Value receiver 1

The customer balance will remain unchanged 2

As we used a value receiver, incrementing the balance in the `add` method didn't mutate the `balance` field of the original `customer` struct:

```
100.00
```

On the other side, with a pointer receiver, Go passes the address of an object to the method. Intrinsically, it remains a copy, but we will only copy a pointer, not the object itself (passing by reference doesn't exist in Go). Any modifications on the receiver will be done on the original object. Here is the same example, but now the receiver is a pointer:

```go
type customer struct {
        balance float64
    }

    func (c *customer) add(operation float64) {
        c.balance += operation
    }

    func main() {
        c := customer{balance: 100.0}
        c.add(50.0)
        fmt.Printf("balance: %.2f\n", c.balance)
    }
```

Pointer receiver 1

The customer balance will be updated 2

As we used a pointer receiver, incrementing the balance mutates the `balance` field of the original `customer` struct:

```
150.00
```

Choosing between value and pointer receiver isn't always straightforward. Let's now discuss most of the conditions to help us in making a choice:

A receiver *must* be a pointer:

- If the method needs to mutate the receiver. This rule is also valid if the receiver is a slice and a method needs to append elements:

```
type slice []int

    func (s *slice) add(element int) {
            *s = append(*s, element)
    }
```

- If the method receiver contains a field that cannot be copied; for example, a type part of the `sync` package (we will discuss this point in *Copying a `sync` type*).

A receiver *should* be a pointer:

- If the receiver is a large object. Indeed, using a pointer could make the call more efficient as it prevents making an extensive copy. In doubt about how large is large, benchmarking can be the solution. Indeed, it's pretty much impossible to state a specific size as it depends on many factors.

A receiver *must* be a value:

- If we have to enforce a receiver's immutability.
- If the receiver is a map, a function, or a channel; otherwise, it leads to a compilation error.

A receiver *should* be a value:

- If the receiver is a slice that doesn't have to be mutated.
- If the receiver is a small array or struct that is naturally a value type without mutable fields, for example: `time.Time`.
- If the receiver is a basic type such as `int`, `float64` or `string`.

There's one case that would need more discussion. Let's say that we design a different `customer` struct where its mutable fields aren't part of the struct directly but inside another one:

```go
type customer struct {
        data *data
}

type data struct {
        balance float64
}

func (c customer) add(operation float64) {
        c.data.balance += operation
}

func main() {
        c := customer{data: &data{
                balance: 100,
        }}
        c.add(50.)
```

```
18                fmt.Printf("balance: %.2f\n", c.data.balance)
19        }
```

The balance isn’t part of customer directly but into a struct referenced by a pointer field 1
Using a value receiver 2

Even though the receiver is a value, calling `add` does change the actual balance in the end:

```
150.00
```

In this case, we don’t need the receiver to be a pointer to mutate `balance`. However, for clarity, one may favor using a pointer receiver to highlight that `customer` as a whole object is mutable.

Note

Is it allowed to mix receiver types? For example, a struct containing multiple methods, some with pointer receivers, some with value receivers. The consensus seems to tend toward forbidding it. However, we can notice some counterexamples in the standard library; one example in particular: `time.Time`.

Indeed, the designers wanted to enforce that a `time.Time` is immutable. Hence, most methods such as `After`, `IsZero`, or `UTC` have a value receiver. Yet, to comply with existing interfaces such as `encoding.TextUnmarshaler`, `time.Time`

> has to implement the `UnmarshalBinary([]byte) error` method, which mutates the receiver given a byte slice. Thus, for this method, the receiver is a pointer one.
>
> Consequently, mixing receiver types should be avoided in general but not forbidden in 100% of the cases.

We should now have a good understanding of whether to use value or pointer receivers. Of course, it's impossible to be exhaustive as there will always be edge cases. Yet, this section's goal was to provide guidance to cover most cases. By default, we can choose to go with a value receiver unless there's a good reason not to do so. In doubt, we should use a pointer receiver.

In the next section, we will discuss named result parameters: reminding us what they are and when to use them.

6.2 Never using named result parameters

Named result parameters are an option in Go that isn't frequently used. This section will see when it's considered appropriate to use named result parameters to make our API more convenient. But first, let's refresh our memory about how it works.

When we return parameters in a function or a method, we can attach names to these parameters and use them as regular variables. When a result parameter is named, it's initialized to its zero value when the function/method begins. Also, with named result parameters, we can call a naked return statement (without arguments). In that case, the current values of the result parameters will be used as the returned values.

Here’s an example that uses a named result parameter `b`:

```
1 func f(a int) (b int) {
2             b = a
3             return
4     }
```

Name the int result parameters to b 1
Return the current value of b 2

We attached in this example the result parameter a name: `b`. When we call `return` without arguments, it will return the current value of `b` at this stage.

Yet, when is it recommended to use named result parameters?

First, let’s consider the following interface containing a method to get the coordinates from a given address:

```
1 type locator interface {
2             getCoordinates(address string) (float32,
3 float32, error)
    }
```

As this interface is unexported, documentation isn’t mandatory. Just by reading this code, can you guess what these two `float32` results are? Perhaps a latitude and longitude but in which order? Depending on the conventions, latitude isn’t always the first element. Therefore, we have to check the implementations to understand the results.

In that case, we should probably use named result parameters to ease code reading:

```
type locator interface {
            getCoordinates(address string) (lat, lng
float32, err error)
    }
```

With this new version, we can understand the meaning of the method signature by looking at the interface: latitude first, longitude second.

Now, let's pursue the question about when to use named result parameters with the method implementation. Should we also use named result parameters as part of the implementation itself?

```
func (l loc) getCoordinates(address string) (
            lat, lng float32, err error) {
            // ...
    }
```

In this very case, having an expressive method signature can also help code readers. Hence, we probably want to use named result parameters as well.

Note

If we need to return multiple results of the same type, we could also think about creating an ad-hoc struct with

> meaningful field names. However, this isn't always possible; for example, when satisfying an existing interface that we can't update.

Now, let's consider another function signature that allows storing a `Customer` type in a DB:

```
func StoreCustomer(customer Customer) (err error) {
            // ...
    }
```

Here, naming the `error` parameter `err` isn't helpful and doesn't help readers. In that case, we should favor not using named result parameters.

So answering about when to use named result parameters depends on the context. In most cases, if it's not clear whether it makes our code more readable, we shouldn't use named result parameters.

Also, we should note that having the result parameters already initialized can be quite handy in some contexts, even though they don't necessarily help readability. Let's see the following example proposed in Effective Go inspired by the `io.ReadFull` function:

```
func ReadFull(r io.Reader, buf []byte) (n int, err error) {
            for len(buf) > 0 && err == nil {
                    var nr int
                    nr, err = r.Read(buf)
                    n += nr
                    buf = buf[nr:]
            }
            return
    }
```

In this example, having named result parameters doesn't really increase readability. However, as both `n` and `err` are initialized to their zero value, the implementation is shorter. Yet, on the other side, this function can also be slightly confusing for readers at first sight. Again, it's a question of finding the right balance.

One note regarding naked returns (returns without arguments): they are considered acceptable in short functions; otherwise, they can harm readability as the reader must remember the outputs throughout the whole function. Meanwhile, we should be consistent within the scope of a function: either use only naked returns or only returns with arguments.

So what are the rules regarding named result parameters? In most cases, using named result parameters in the context of an interface definition can increase readability without leading to any side effects. Yet, there's no strict rule in the context of a method implementation. In some cases, they can also increase readability if two parameters have the same type, for example. In other cases, they can also be used for convenience. Therefore, we should use named result parameters sparingly when there's a clear benefit.

> **Note**
>
> In *Not handling defer errors*, we will discuss another use case for using named result parameters in the context of defer calls.

Furthermore, if we're not careful enough, they can lead to side effects and unintended consequences, as we will see in the next section.

6.3 Unintended side-effects with named result parameters

We mentioned why named result parameters could be useful in some situations. Yet, as these result parameters are initialized to their zero value, it can sometimes lead to subtle bugs if we're not careful enough. This section will illustrate such a case.

Let's enhance one previous example we discussed with a method returning the latitude and longitude from a given address. As we return two `float32`, we will decide to use named result parameters to make explicit the latitude and the longitude. This function will first validate the given address and then get the coordinates. In between, it will perform a check on the input context to make sure it wasn't canceled, or its deadline has passed.

> **Note**
>
> We will delve into how the concept of context in Go in *Misunderstanding Go contexts*. If you're not yet familiar with contexts, let's briefly say that a context can carry a cancellation signal or a deadline. We can check those by calling the `Err` method and checking that the returned error isn't nil.

Here's the new implementation of the `getCoordinates` method. Can you spot what's wrong with this code?

```
1  func (l loc) getCoordinates(ctx context.Context, address
2  string) (
3              lat, lng float32, err error) {
4              isValid := l.validateAddress(address)
5              if !isValid {
6                      return 0, 0, errors.New("invalid
7  address")
8              }
9
10             if ctx.Err() != nil {
11                     return 0, 0, err
12             }
13
               // Get and return coordinates
     }
```

Validate the address 1

Check whether the context was canceled or the deadline has passed 2

The error might not be that obvious at first glance. Here, the error returned in the `if ctx.Err() != nil` scope is `err`. Yet, we haven't assigned any value to it; it's still assigned to the zero value of an `error` type: nil. Hence, this code will always return a nil error.

Furthermore, this code compiles because `err` was initialized to its zero value because of named result parameters. Without attaching a name, we would have got the following compilation error:

```
Unresolved reference 'err'
```

One possible fix can be to assign `ctx.Err()` to `err` like so:

```
1 if err := ctx.Err(); err != nil {
2             return 0, 0, err
```

```
    }
```

We keep returning `err`, but we first assign it to the result of `ctx.Err()`. Note that `err` in this example will shadow the result variable.

> **Note**
>
> However, it would break the rule stating that we shouldn't mix naked return and return with arguments. In this case, we should probably stick with the first option. Remember that using named result parameters doesn't necessarily mean using naked returns. Sometimes, we can just use named result parameters to make a signature clearer.
>
> Another option could be to use a naked return statement:
>
> ```
> if err = ctx.Err(); err != nil {
>  return
> }
> ```

We can conclude with named result parameters by emphasizing that in some cases (e.g., returning multiple times the same type), it can improve code readability and be quite handy in other cases. Yet, we must recall that each parameter is initialized to its zero value. As we have seen in this section, it can lead to possible subtle bugs that aren't always straightforward to spot while reading code. Therefore, let's remain

cautious when using named result parameters to avoid potential side effects.

In the next section, we will discuss a common mistake made by Go developers when a function returns an interface.

6.4 Returning a nil receiver

We will discuss in this section the impact of returning an interface and why it may lead to errors in some conditions. This mistake is probably one of the most widespread in Go as it might be considered as counterintuitive, at least before we've made it.

Let's consider the following example. We will work on a `Customer` struct and implement a `Validate` method to perform sanity checks. Yet, we don't want to return the first error but possibly a list of errors. To do that, we will create a custom error type to convey multiple errors:

```
1  type MultiError struct {
2              errs []string
3      }
4
5      func (m *MultiError) Add(err error) {
6              m.errs = append(m.errs, err.Error())
7      }
8
9      func (m *MultiError) Error() string {
10             return strings.Join(m.errs, ";")
11     }
```

Add an error 1
Implement the error interface 2

`MultiError` satisfies the `error` interface as it implements `Error() string`. Meanwhile, it exposes an `Add` method to append an error. Using this struct, we can implement the `Customer.Validate` method in the following manner to check the customer's age and name. In case the sanity checks are ok, we want to return a nil error:

```
1  func (c Customer) Validate() error {
2              var m *MultiError
3
4              if c.Age < 0 {
5                      m = &MultiError{}
6                      m.Add(errors.New("age is negative"))
7              }
8              if c.Name == "" {
9                      if m == nil {
10                             m = &MultiError{}
11                     }
12                     m.Add(errors.New("name is nil"))
13             }
14
15             return m
16     }
```

Instantiate an empty *MultiError 1
Append an error if age is negative 2
Append an error if name is nil 3

In this implementation, `m` is initialized to the zero value of `*MultiError`; hence: nil. In case of a failing sanity check, we allocate a new `MultiError` if needed, then append an error. In the end, we return `m` that can be either a nil pointer or a pointer to a `MultiError` struct depending on the checks.

Now, let's test this implementation by running a case with a valid `Customer`:

```
customer := Customer{Age: 33, Name: "John"}
    if err := customer.Validate(); err != nil {
            log.Fatalf("customer is invalid: %v", err)
    }
```

Here will be the output:

```
2021/05/08 13:47:28 customer is invalid: <nil>
```

This result may be pretty surprising. Indeed, the `Customer` was valid, yet the `err != nil` condition was true, and logging the error printed `<nil>`. So, what's the issue?

In Go, we have to know that a pointer receiver can be nil. Let's experiment it by creating a dummy type and calling a method with a nil pointer receiver:

```
type Foo struct{}

    func (foo *Foo) Bar() string {
            return "bar"
    }

    func main() {
            var foo *Foo
            fmt.Println(foo.Bar())
    }
```

foo is nil 1

`foo` is initialized to the zero value of a pointer: nil. Yet, this code compiles, and it prints `bar` if we run it. A nil pointer is a valid receiver.

But why is this the case? In Go, a method is just some syntactic sugar for a function whose first parameter would be the receiver. Hence, the `Bar` method we've seen is similar to this function:

```
func Bar(foo *Foo) string {
            return "bar"
    }
```

We know that passing a nil pointer to a function is valid. Therefore, using a nil pointer as a receiver is also valid.

Let's get back to our initial example:

```
func (c Customer) Validate() error {
            var m *MultiError

            if c.Age < 0 {
                    // ...
            }
            if c.Name == "" {
                    // ...
            }

            return m
    }
```

`m` is initialized to the zero value of a pointer: nil. Then, if all the checks are valid, the argument provided to the return statement isn't `nil` directly but a nil pointer. As a nil pointer is a valid receiver, converting the result into an interface won't yield a nil value. Said differently, the caller of `Validate` will always get a non-nil error.

To make this point clear, let's remember that in Go, an interface is a dispatch wrapper. Here, the wrappee is nil (the `MultiError` pointer), whereas the wrapper isn't (the `error` interface):

Figure 6.1. The `error` wrapper isn't nil.

Therefore, regardless of the `Customer` provided, the caller of this function will always receive a non-nil error. Understanding this behavior is imperative as it's a widespread Go mistake.

So, what should we do to fix this example? The easiest solution, at the end of the function, is to return `m` only if it's not nil:

```
1  func (c Customer) Validate() error {
2              var m *MultiError
3
4              if c.Age < 0 {
5                      // ...
```

```
        }
        if c.Name == "" {
                // ...
        }

        if m != nil {
                return m
        }
        return nil
    }
```

Return m only if there was at least one error 1
Otherwise, return nil 2

At the end of the method, we check whether `m` isn't nil. If true, we return `m`; else, we return `nil` explicitly. Hence, in the case of a valid `Customer`, we return a nil interface, not a nil receiver converted into a non-nil interface.

We've seen in this section that in Go, having a nil receiver is allowed, and an interface converted from a nil pointer isn't a nil interface. For that reason, when we have to return an interface, we shouldn't return a nil pointer but a nil value directly. Generally, having a nil pointer isn't a desirable state and means a probable bug.

We saw an example with errors throughout this section as it's the most common case leading to this error. Yet, please note that this problem isn't only tied to errors. It can happen with any interface implemented using pointer receivers.

The next section will discuss a common design mistake when using a filename as a function input.

6.5 Using a filename as a function input

When creating a new function that needs to read a file, passing a filename isn't considered a best practice and can have negative impacts, such as making unit tests harder to write. Let's delve into this problem and understand how to overcome it.

We want to implement a function to count the number of empty lines in a file. One possible way to implement this function would be to accept a filename and use `bufio.NewScanner` to scan and check every line:

```
1  func countEmptyLinesInFile(filename string) (int, error) {
2              file, err := os.Open(filename)
3              if err != nil {
4                      return 0, err
5              }
6              // Handle file closure
7
8              scanner := bufio.NewScanner(file)
9              for scanner.Scan() {
10                     // ...
11             }
12     }
```

Open filename 1
Create a scanner from the *os.File variable that will split the input per line 2
Iterate over each line 3

We open a file from the filename. Then, we use `bufio.NewScanner` to scan every line (by default, it splits the input per line).

This function will do what we expect it to do. Indeed, as long as the provided filename is valid, we will read from it and return the number of empty lines. So what's the problem?

Let’s say that we want to implement unit tests to cover the following cases:

- A nominal case
- An empty file
- A file containing only empty lines

Each unit test will require creating a file within our Go project. The more complex the function is, the more cases we may want to add, and the more files we will create. We may even have to create dozens of files in some cases, which can quickly become unmanageable.

Furthermore, one may also point out that this function isn’t reusable. For example, if we had to implement the same logic but to count the number of empty lines with an HTTP request, we would have to duplicate the main logic:

```
func countEmptyLinesInHTTPRequest(request http.Request)
(int, error) {
            scanner := bufio.NewScanner(request.Body)
            // Copy the same logic
    }
```

One possible way to overcome these limitations might be to make the function accept a `*bufio.Scanner` (the output returned by `bufio.NewScanner`). Indeed, both functions have the same logic from the moment we created the `scanner` variable. It would work. Yet, in Go, the idiomatic way is to start from the reader’s abstraction.

Let's write a new version of the `countEmptyLines` function that would receive an `io.Reader` abstraction instead:

```
1 func countEmptyLines(reader io.Reader) (int, error) {
2             scanner := bufio.NewScanner(reader)
3             for scanner.Scan() {
4                     // ...
5             }
6     }
```

Accept an io.Reader as the input 1
Create a *bufio.Scanner from an io.Reader, not an *os.File 2

As `bufio.NewScanner` accepts an `io.Reader`, we can directly pass the `reader` variable.

So what are the benefits? First, this function abstracts the data source. Is it a file? An HTTP request? A socket input? It's not important for the function. As `*os.File` and the `Body` field of `http.Request` implement `io.Reader`, we can reuse the same function regardless of the input type.

Another benefit is related to testing. We mentioned that creating one file per test case could become cumbersome quite quickly. Now that `countEmptyLines` accepts an `io.Reader`, we can implement unit tests by creating an `io.Reader` from a string:

```
1 func TestCountEmptyLines(t *testing.T) {
2             emptyLines, err :=
3 countEmptyLines(strings.NewReader(
4                     `foo
5                             bar
6
```

```
                                baz
                                `))
            // Test logic
    }
```

Pass an io.Reader from a string 1

In this test, we created an `io.Reader` using `strings.NewReader` from a string literal directly. Therefore, we don't have to create one file per test case. Each test case can be self-contained, improving the test readability and maintainability as the reader doesn't have to open another file to see the content.

Accepting a filename as a function input to read from a file should, in most cases, be considered as a code smell (except in specific functions such as `os.Open`). As we've seen, it makes unit tests more complex as we may have to create multiple files. Also, it reduces the reusability of a function (though not all the functions are meant to be reused eventually). Using the `io.Reader` interface abstracts the data source. Regardless of whether the input is a file, a string, an HTTP, or gRPC request, the implementation can be reused and easily tested.

For the last section of the chapter, let's discuss a common mistake related to `defer`: how function/method arguments and method receivers are evaluated.

6.6 Ignoring how defer arguments and receivers are evaluated

We mentioned in a previous section that the `defer` statement delays a call's execution until the surrounding function returns. A common mistake

made by Go developers is not understanding how arguments are evaluated. We will delve into this problem with two subsections: one related to functions or methods arguments, the second related to method receivers.

6.6.1 Arguments evaluation

To illustrate how arguments are evaluated with `defer`, let's work on a concrete example. A function will have to call two functions `foo` and `bar`. Meanwhile, it has to handle a status regarding the execution:

- `StatusSuccess` if both `foo` and `bar` return no errors
- `StatusErrorFoo` if `foo` returns an error
- `StatusErrorBar` if `bar` returns an error

We will use this status for multiple actions. For example, to notify another goroutine and to increment counters. To avoid repeating these calls before every `return` statement, we will use `defer`. Here's our first implementation:

```
const (
        StatusSuccess  = "success"
        StatusErrorFoo = "error_foo"
        StatusErrorBar = "error_bar"
    )

    func f() error {
        var status string
        defer notify(status)
        defer incrementCounter(status)

        if err := foo(); err != nil {
            status = StatusErrorFoo
```

```
14                      return err
15              }
16
17              if err := bar(); err != nil {
18                      status = StatusErrorBar
19                      return err
20              }
21
22              status = StatusSuccess
23              return nil
24      }
```

Defer the call to notify 1
Defer the call to incrementCounter 2
Set the status to error foo 3
Set the status to error bar 4
Set the status to success 5

First, we declare a `status` variable. Then, we defer the calls to `notify` and `incrementCounter` using `defer`. Throughout this function and depending on the execution path, we update `status` accordingly.

However, if we give this function a try, we can notice that regardless of the execution path, `notify` and `incrementCounter` are always called with the same status: an empty string. How is it possible?

There's something crucial to understand with arguments evaluation in a defer function: the arguments are evaluated *right away*, not once the surrounding function returns. In our example, we called `notify(status)` and `incrementStatus(status)` as defer functions. Therefore, Go will delay these calls to be executed once `f` returns with the current value of `status` at the stage we used `defer`; hence, an empty string.

How can we solve this problem if we want to keep using `defer`? There are two leading solutions.

The first solution is to pass to the defer functions a string pointer:

```go
func f() error {
        var status string
        defer notify(&status)
        defer incrementCounter(&status)

        // The rest of the function unchanged
        if err := foo(); err != nil {
                status = StatusErrorFoo
                return err
        }

        if err := bar(); err != nil {
                status = StatusErrorBar
                return err
        }

        status = StatusSuccess
        return nil
    }
```

Pass to notify a string pointer 1
Pass to incrementCounter a string pointer 2

We keep updating `status` depending on the cases but now, `notify` and `incrementCounter` receive a string pointer. Why does it work?

Using `defer` evaluates right away the arguments; here, the address of `status`. Yes, `status` itself is modified throughout the function, but its address remains constant, regardless of the assignments. Hence, if `notify` or `incrementCounter` uses the value referenced by the string pointer, it will work as expected. However, this solution requires changing the signature of the two functions, which may not always be possible.

There’s another possible solution: calling as a defer statement a closure. As a reminder, a closure is an anonymous function value that references variables from outside its body. The arguments passed to a defer function are evaluated right away. Yet, we must know that the variables referenced by a defer closure are evaluated *during* the closure execution (hence, when the surrounding function returns).

Here is an example to clarify how defer closures work. A closure will reference two variables, one as a function argument, the second as a variable outside its body:

```
1 func main() {
2             i := 0
3             j := 0
4             defer func(i int) {
5                     fmt.Println(i, j)
6             }(i)
7             i++
8             j++
9     }
```

Call as a defer function a closure that accepts an integer as an input 1
i is the function input and j is an external variable 2
Pass i to the closure (evaluated right away) 3

Here, the closure uses `i` and `j` variables. `i` is passed as a function argument, so it’s evaluated immediately. Conversely, `j` references a variable outside of the closure body, so it’s evaluated when the closure is executed. If we run this example, it will print `0 1`.

Therefore, we can use a closure to implement a new version of our function:

```
1 func f() error {
2             var status string
3             defer func() {
4                     notify(status)
5                     incrementCounter(status)
6             }()
7
8             // The rest of the function unchanged
9     }
```

Call a closure as the defer function 1
Call notify within the closure and reference status 2
Call incrementCounter within the closure and reference status 3

Here, we wrapped the calls to both `notify` and `incrementCounter` within a closure. This closure references the `status` variable from outside its body. Therefore, `status` will be evaluated once the closure is executed, not once we call `defer`.

This solution will also work and doesn't require `notify` and `incrementCounter` changing their signature.

Now, what about using `defer` on a method with a pointer or value receiver? Let's delve into these questions.

6.6.2 Pointer and value receiver

In *Not knowing which type of receiver to use*, we discussed that a receiver could either be a value or a pointer. The same logic related to arguments evaluation applies when we use `defer` on a method: the receiver is also evaluated immediately. Let's understand the impacts with both receiver types:

First, here's an example that calls a method on a value receiver using defer but mutate this receiver afterward:

```
1  func main() {
2              s := Struct{id: "foo"}
3              defer s.print()
4              s.id = "bar"
5      }
6
7      type Struct struct {
8              id string
9      }
10
11     func (s Struct) print() {
12             fmt.Println(s.id)
13     }
```

s is evaluated immediately 1
Update s.id (not visible) 2
foo 3

We defer the call to the `print` method. Like arguments, calling `defer` makes the receiver be evaluated immediately. Hence, `defer` will delay the method's execution with a struct that contains an `id` field equal to `foo`. Therefore, this example prints `foo`.

Conversely, if the pointer is a receiver, the potential changes to the receiver after the call to `defer` will be visible:

```
func main() {
        s := &Struct{id: "foo"}
        defer s.print()
        s.id = "bar"
    }

    type Struct struct {
        id string
    }

    func (s *Struct) print() {
        fmt.Println(s.id)
    }
```

s being a pointer, it is evaluated immediately but may reference another variable when the defer method (visible) is executed

```
Defer method (visible) 2
bar 3
```

The `s` receiver is also evaluated immediately. However, it will lead to copying a pointer. Hence, the changes made to the struct referenced by the pointer are visible. This example prints `bar`.

In summary, we must remind that when calling `defer` on a function or method, the call's arguments are evaluated immediately. If we want to mutate the arguments provided to `defer` afterward, we can use pointers or closures. For a method, the receiver is also evaluated immediately; hence, the behavior depends on whether the receiver is a value or a pointer.

6.7 Chapter summary

- Deciding on using either value or pointer receiver should be made according to various factors such as the type, if it has to be mutated, if it contains a field that can't be copied, and how large the object is. In doubt, we should use a pointer receiver.
- Using named result parameters can be an efficient way to improve the readability of a function/method, especially if multiple result parameters have the same type. In some cases, it can also be convenient as they will be initialized to their zero value. Yet, we have to be cautious about potential side effects.
- When returning an interface, we have to be cautious about not returning a nil pointer but an explicit nil value. Otherwise, it may lead to unintended consequences as the caller will receive a non-nil value.
- Designing functions to receive `io.Reader` types instead of filenames improves the reusability of a function and makes testing easier.
- Passing a pointer to a defer function or wrapping a call inside a closure are two possible solutions to overcome arguments and receivers being evaluated immediately.

7 Error management

This chapter covers

- Understanding when to panic
- Knowing when to wrap an error or not
- Comparing an error type or an error value efficiently since Go 1.13
- Learning how to handle errors idiomatically
- Understanding how to ignore an error
- Handling errors in defer calls

Error management is a fundamental aspect of building robust and observable applications, and it should probably be as important as any other part of a codebase. In Go, error management in Go is quite singular. Indeed, it doesn't rely on the traditional try/catch mechanism as most programming languages. Instead, errors are returned as normal return values. This chapter will cover the most common mistakes related to errors.

7.1 Panicking

It's pretty common for Go newcomers to be somewhat confused about error handling. In Go, errors are usually managed by functions or methods, returning an `error` type as the last parameter. Yet, one may find this approach somewhat surprising and be tempted to reproduce exception handling in languages such as Java or Python using `panic` and `recover`. So, let's refresh our minds about the concept of panic and discuss when it's considered appropriate or not to panic.

In Go, `panic` is a built-in function that stops the ordinary flow:

```
1 func main() {
2             fmt.Println("a")
3             panic("foo")
4             fmt.Println("b")
5     }
```

This code prints `a` then stops before printing `b`:

```
1 a
2     panic: foo
3
4     goroutine 1 [running]:
5     main.main()
6             main.go:7 +0xb3
```

Once a panic is triggered, it continues up to the call stack either until the current goroutine has returned, or `panic` is caught with `recover`:

```
1  func main() {
2              defer func() {
3                      if r := recover(); r != nil {
4                              fmt.Println("recover", r)
5                      }
6              }()
7
8              f()
9      }
10
11     func f() {
12             fmt.Println("a")
13             panic("foo")
14             fmt.Println("b")
15     }
```

Call recover within a defer closure 1

```
Call f that will panic. This panic will be caught by the previous recover 2
```

In the `f` function, once `panic` is called, it stops the current execution of the function and goes up to the call stack: `main`. In `main`, as the panic is caught with `recover`, it doesn't stop the goroutine:

```
a
    recover foo
```

We should note that calling `recover()` to capture a goroutine panicking is only useful inside a defer function; otherwise, the function would return nil and have no other effect. This is because `defer` functions are also executed when the surrounding function panics.

As we have refreshed our minds about panic, let's tackle this question: when is it appropriate to panic?

In Go, panic is used to signal from genuinely exceptional conditions. For example, to signal a programmer error. If we delve into the `net/http` package, we can notice that in the `WriteHeader` method, there is a call to a `checkWriteHeaderCode` function to check whether the status code is valid:

```go
func checkWriteHeaderCode(code int) {
            if code < 100 || code > 999 {
                    panic(fmt.Sprintf("invalid WriteHeader code %v", code))
            }
    }
```

This function panics if the status code is invalid, which would be a pure programmer error.

Another example based on a programmer error could be found in the `database/sql` package while registering a database driver:

```
1  func Register(name string, driver driver.Driver) {
2              driversMu.Lock()
3              defer driversMu.Unlock()
4              if driver == nil {
5                      panic("sql: Register driver is nil")
6              }
7              if _, dup := drivers[name]; dup {
8                      panic("sql: Register called twice for
9  driver " + name)
10             }
11             drivers[name] = driver
       }
```

Panic if the driver is nil 1
Panic if the driver already registered 2

This function panics if the driver is nil (`driver.Driver` is an interface) or has already been registered. In both cases, it would be again considered as a programmer error. Also, in most cases (for example, with `go-sql-driver/mysql` [13] the most popular MySQL driver for Go), calling `Register` is done via an init function which limits error handling. For all these reasons, in case of an error, the designers made the function panic.

Another use case to panic is when our application requires a dependency, but it fails to initialize it. For example, let’s imagine that we expose a service to create new customer accounts. This service, at some stage, needs to validate the provided email address. To implement it, we can decide to use a regular expression.

In Go, the `regexp` package exposes two functions to create a regular expression from a string: `Compile` and `MustCompile`. The former returns a `*regexp.Regexp` and an error, whereas the latter returns only a `*regexp.Regexp` but panics in case of an error. In this case, the regular expression is a mandatory dependency. Indeed, if we fail to compile it, we will never be able to validate any email input. Hence, we may favor using `MustCompile` and panic in case of an error.

Panicking in Go should be used sparingly. We have seen two prominent cases, one to signal a programmer error and another where our application fails to create a mandatory dependency. Hence, exceptional conditions leading the application to be stopped. In most other cases, error management should be done with a function returning a proper `error` type as the last return argument.

Let's now start discussing errors and see when to wrap an error.

7.2 Ignoring when to wrap an error

Since Go 1.13, the `%w` directive allows us to wrap errors conveniently. Yet, some developers might be confused about when we should wrap an error or not. So, let's remind ourselves what error wrapping is and then when to use it.

Error wrapping is about wrapping or packing up an error inside a wrapper container that also make the source error available:

Figure 7.1. Wrap the error inside a wrapper.

In general, the two main use cases for error wrapping are the following:

- Adding additional context to an error
- Marking an error as a specific error

Regarding adding context, we can think about the following example. We receive a request from a specific user to access a DB resource but get a permission denied error during the query. For debugging purposes, if the error is eventually logged, we want to add extra context. In this case, we can wrap the error to indicate who is the user and what is the resource accessed:

Figure 7.2. Add additional context to the permission denied error.

Now let's say that instead of adding context, we want to mark the error. For example, we want to implement an HTTP handler that checks whether all the errors received while calling functions are of a `Forbidden` type to return a 403 status code. In that case, we can wrap this error inside `Forbidden`:

Figure 7.3. Mark the error as `Forbidden`.

In both cases, the source error remains available. Hence, a caller can also handle an error by unwrapping it and checking the source error. Let's also note that sometimes, we want to combine both approaches: adding context and marking an error.

Now that we clarified the main use cases to wrap an error, let's see different ways in Go to return an error we receive. We will consider the following piece of code and explore different options inside of the `if err != nil` block:

```
func Foo() error {
            err := bar()
            if err != nil {
                    // ?
            }
            // ...
    }
```

How to return the error? 1

The first option is to return this error directly. Indeed, if we don't want to mark the error or there's no helpful context we can think about adding, then this approach is perfectly fine:

```go
if err != nil {
        return err
    }
```

Here, we return the same error:

Figure 7.4. Return the error directly.

Before Go 1.13, to wrap an error, the only option without using any external library was to create a custom error type:

```go
type BarError struct {
        Err error
    }

    func (b BarError) Error() string {
        return "bar failed:" + b.Err.Error()
    }
```

Then, instead of returning `err` directly, we wrap the error into a `BarError`:

```go
if err != nil {
        return BarError{Err: err}
    }
```

Figure 7.5. Wrap the error inside `BarError`.

The benefit of this option is its flexibility. As `BarError` is a custom struct, we can add any additional context if needed. However, being obliged to create a specific error type can become quickly cumbersome if we want to repeat this operation.

To overcome this situation, Go 1.13 introduced the `%w` directive:

```
if err != nil {
            return fmt.Errorf("bar failed: %w", err)
    }
```

This code wraps the source error to add additional context without having to create another error type:

Figure 7.6. Wrap an error into a standard error.

As the source error remains available, a client can unwrap the parent error and then check whether the source error was of a specific type or value (we will discuss these points in the next sections).

The last option we will discuss is to use the `%v` directive instead:

```
if err != nil {
            return fmt.Errorf("bar failed: %v", err)
    }
```

The difference here is that the error itself isn't wrapped. We transform it into another error to add context but the source error itself isn't available anymore:

Figure 7.7. Convert the error.

The information about the source of the problem remains available. However, a caller can't unwrap this error and check whether the source was `bar error`. So, in a sense, this option is more restrictive than `%w`. However, should we prevent that since the `%w` directive has been released? Not necessarily.

Wrapping an error makes the source error available for callers. Hence, it means introducing potential coupling. For example, imagine if we use wrapping and the caller of `Foo` checks whether the source error is `bar error`. Now, what if we want to change our implementation and use another function that will return another type of error? It will break the error check made by the caller.

If we want to make sure our clients don't rely on something that we consider as implementation details, then the error returned shouldn't be wrapped but transformed. In such a case, using `%v` instead of `%w` can be the way to go.

So, let's summarize all the different options we tackled:

Option	Extra context	Marking an error	Source error available
Returning error directly	No	No	Yes
Custom error type	Possible (if the error type contains a string field, for example)	Yes	Possible (if the source error is exported or accessible via a method)
`fmt.Errorf` with `%w`	Yes	No	Yes
`fmt.Errorf` with `%v`	Yes	No	No

To summarize, when handling an error, we can decide to wrap it. Wrapping is about adding additional context to an error and/or marking an error as a specific type. If we need to mark an error, we should create a custom error

type. However, if we just want to add extra context, we should use `fmt.Errorf` with the `%w` directive as it doesn't require creating a new error type. However, error wrapping creates potential coupling as it makes the source error available for the caller. If we want to prevent it, we shouldn't use error wrapping but error transformation, for example, using `fmt.Errorf` with the `%v` directive.

This section has seen how to wrap an error with the `%w` directive. But once we start using it, what's the impact of comparing an error type?

7.3 Comparing an error type inaccurately

The previous section introduced a possible way to wrap errors using the `%w` directive. However, once we start using it, it's also essential to change our way to check for a specific error type; otherwise, we may handle errors inaccurately.

Let's discuss a concrete example. We will write an HTTP handler to return the transaction amount from an ID. Our handler will parse the request to get the ID and retrieves the amount from a DB. Our implementation can fail in two cases:

- If the ID is invalid (string length different from five characters)
- If querying the DB fails

In the former case, we want to return StatusBadRequest (400), whereas, in the latter, we want ServiceUnavailable (503). To do so, we will create a `transientError` type to mark that an error is temporary. The parent handler will check the error type. If the error is a `transientError` one, it will return a 503; otherwise, a 400.

Let’s first focus on the error type definition and the function the handler will call:

```
1  type transientError struct {
2              err error
3      }
4
5      func (t transientError) Error() string {
6              return fmt.Sprintf("transient error: %v", t.err)
7      }
8
9      func getTransactionAmount(transactionID string) (float32,
10 error) {
11             if len(transactionID) != 5 {
12                     return 0, fmt.Errorf("id is invalid: %s",
13 transactionID)
14             }
15
16             amount, err :=
17 getTransactionAmountFromDB(transactionID)
18             if err != nil {
19                     return 0, transientError{err: err}
               }
               return amount, nil
       }
```

Create a custom transientError 1
Return a simple error if the transaction ID is invalid 2
Return a transientError if we fail to query the DB 3

`getTransactionAmount` returns an error using `fmt.Errorf` if the identifier is invalid. However, if getting the transaction amount from the DB fails, it wraps the error into a `transientError` type.

Now, let’s write the HTTP handler that will check the error type to return the appropriate HTTP status code:

```go
func handler(w http.ResponseWriter, r *http.Request) {
            transactionID := r.URL.Query().Get("transaction")

            amount, err := getTransactionAmount(transactionID)
            if err != nil {
                    switch err := err.(type) {
                    case transientError:
                            http.Error(w, err.Error(),
http.StatusServiceUnavailable)
                    default:
                            http.Error(w, err.Error(),
http.StatusBadRequest)
                    }
                    return
            }

            // Write response
    }
```

Extract the transaction ID 1
Call getTransactionAmount that contains the whole logic 2
Check the error type and return a 503 if the error is a transient one, otherwise a 400 3

Using a `switch` on the error type, we return the appropriate HTTP status code: 400 in case of a bad request or 503 in case of a transient error.

This code is perfectly valid. However, let's now assume we want to perform a small refactoring of `getTransactionAmount`. The `transientError` won't be returned by `getTransactionAmount` but by `getTransactionAmountFromDB`. `getTransactionAmount` will now wrap this error using the `%w` directive:

```go
func getTransactionAmount(transactionID string) (float32,
error) {
            // Check transaction ID validity

            amount, err :=
getTransactionAmountFromDB(transactionID)
            if err != nil {
                    return 0, fmt.Errorf("failed to get
transaction %s: %w",
                            transactionID, err)
            }
```

```
12              return amount, nil
13      }
14
15      func getTransactionAmountFromDB(transactionID string)
16 (float32, error) {
17              // ...
18              if err != nil {
                        return 0, transientError{err: err}
                }
                // ...
        }
```

Instead of returning a transientError directly, it now wraps the error 1
The transientError is now returned by this function 2

If we run this code, we will notice that regardless of the error case, we will always return a 400. Meaning, the `case Transient` error will never be hit. How can we explain this behavior?

Before the refactoring, `transientError` was returned by `getTransactionAmount`:

Figure 7.8. As `getTransactionAmount` was returning a `transientError` if the DB fails, the case was true.

After the refactoring, `transientError` is now returned by `getTransactionAmountFromDB`:

Figure 7.9. Now, `getTransactionAmount` returns a wrapped error. Hence, the `case transientError` is now false.

What `getTransactionAmount` returns isn't a `transientError` directly: it's an error wrapping `transientError`. Therefore the `case transientError` is now false.

For that exact purpose, Go 1.13 also came with a directive to wrap an error and a way to check if the wrapped error is of a certain type with `errors.As`. This function recursively unwraps an error and returns true if an error in the chain matches the expected type.

Let's now rewrite our implementation of the caller using `errors.As`:

```
1  func handler(w http.ResponseWriter, r *http.Request) {
2              // Get transaction ID
3
4              amount, err := getTransactionAmount(transactionID)
5              if err != nil {
6                      if errors.As(err, &transientError{}) {
7                              http.Error(w, err.Error(),
8  http.StatusServiceUnavailable)
9                      } else {
10                             http.Error(w, err.Error(),
11 http.StatusBadRequest)
12                     }
13                     return
14             }
```

```
15
            // Write response
    }
```

Call errors.As by providing a pointer to transientError 1
Return a 503 if the error is transient 2
Else return a 400 3

We got rid of the switch case type in this new version, and we now use `errors.As`. This function requires the second argument (the target error) to be of a pointer. Otherwise, the function will compile but panic at runtime. Using `errors.As`, regardless if the runtime error is directly a `transientError` type or an error wrapping `transientError`, it will be able to match and then return a 503.

In summary, if we rely on Go 1.13 error wrapping, we must use `errors.As` to check whether an error is a specific type. This way, regardless of whether the error is returned directly by the function we call or wrapped inside an error, `error.As` will be able to recursively unwrap our main error and see if one of the errors is a specific type.

We have just seen how to compare an error type; now it's time to compare an error value.

7.4 Comparing an error value inaccurately

This section is similar to the one we just discussed but with sentinel errors (error values). First, we will define what a sentinel error conveys. Then, we will see how to compare an error to a value.

A sentinel error is an error defined as a global variable:

```
import "errors"

    var ErrFoo = errors.New("foo")
```

In general, the convention is to start with `Err` followed by the error type; here `ErrFoo`. A sentinel error conveys an *expected* error. But what do we mean by an expected error? Let's discuss it in the context of an SQL library.

We want to design a `Query` method that allows to execute a query to a database. This method returns a slice of rows. How should we handle the case when no rows are found? Here we have two options:

- Either to return a sentinel value; for example, a nil slice (think about `strings.Index` that returns the sentinel value -1 if a substring isn't present)
- Or to return a specific error that a client can check

Let's take the second approach. So, our method can return a specific error if no rows are found. We can classify this error as an expected one as passing a request that returns no rows is allowed. Conversely, in the event of errors such as a network issue, or a connection polling error, we face unexpected errors. It doesn't mean that we don't want to handle unexpected errors; it means that semantically, they convey a different meaning.

If we take a look at the standard library, we can find many examples of sentinel erros:

- `sql.ErrNoRows`: returned when a query doesn't return any row (which was exactly our case)

- `io.EOF`: returned by an `io.Reader` when no more input is available

That's the general principle behind sentinel errors. They are conveying an expected error that clients will expect to check. Therefore, as a general guideline:

- Expected errors should be designed as error values (sentinel errors): `var ErrFoo = errors.New("foo")`
- Unexpected errors should be designed as error types: `type BarError struct{ … }` with `BarError` implementing the `error` interface

Let's get back to the common mistake. How can we compare an error to a specific value? We can use the `==` operator:

```
err := query()
    if err != nil {
            if err == sql.ErrNoRows {
                    // ...
            } else {
                    // ...
            }
    }
```

Check error against the sql.ErrNoRows variable 1

Here, we called a `query` function and got an error. Checking if the error is an `sql.ErrNoRows` was done using the `==` operator.

However, in the same way we discussed in the previous section, a sentinel error can also be wrapped. If an `sql.ErrNoRows` is wrapped using

`fmt.Errorf` and the `%w` directive, the `err == sql.ErrNoRows` will always be false.

Again, Go from version 1.13 provides an answer. We have seen how `errors.As` is used to check an error against a type. With error values, we should use its counterpart: `errors.Is`. Let's rewrite the previous example:

```
err := query()
    if err != nil {
            if errors.Is(err, sql.ErrNoRows) {
                    // ...
            } else {
                    // ...
            }
    }
```

Using `errors.Is` instead of the `==` operator allows the comparison to work even if the error was wrapped using `%w`.

In summary, if we use in our application error wrapping using the `%w` directive and `fmt.Errorf`, checking an error against a specific value shouldn't be done using `==` but using `errors.Is`. Thus, even if the sentinel error is wrapped, `errors.Is` will be able to recursively unwrap it and compare each error in the chain against the provided value.

Now, it's time to discuss one of the most important aspects of error handling: not handling an error twice.

7.5 Handling an error twice

Handling an error multiple times is a frequent mistake made by developers, not specifically in Go. Let's understand why this is a problem and how to

handle errors efficiently.

To illustrate the problem, let's write a `GetRoute` function to get the route from a pair of sources to a pair of target coordinates. Let's assume this function will call an unexported `getRoute` function that contains the business logic to calculate the best route. Before calling `getRoute`, we have to validate the source and target coordinates using `validateCoordinates`. We also want the possible errors to be logged. Here's a possible implementation:

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route, error) {
            err := validateCoordinates(srcLat, srcLng)
            if err != nil {
                    log.Println("failed to validate source coordinates")
                    return Route{}, err
            }

            err = validateCoordinates(dstLat, dstLng)
            if err != nil {
                    log.Println("failed to validate target coordinates")
                    return Route{}, err
            }

            return getRoute(srcLat, srcLng, dstLat, dstLng)
    }

    func validateCoordinates(lat, lng float32) error {
            if lat > 90.0 || lat < -90.0 {
                    log.Printf("invalid latitude: %f", lat)
                    return fmt.Errorf("invalid latitude: %f", lat)
            }
            if lng > 180.0 || lng < -180.0 {
                    log.Printf("invalid longitude: %f", lng)
                    return fmt.Errorf("invalid longitude: %f", lng)
            }
            return nil
    }
```

Log and return the error 1

Log and return the error 2
Log and return the error 3
Log and return the error 4

What's the problem with that code? First, we can notice how cumbersome it is in `validateCoordinates` to repeat the `invalid latitude` or `invalid longitude` error messages in both logging and the error returned. Also, if we run it with an invalid latitude, for example, it will log the following lines:

```
2021/06/01 20:35:12 invalid latitude: 200.000000
    2021/06/01 20:35:12 failed to validate source coordinates
```

Hence, two log lines for a single error, which is a problem. Why? Because it makes debugging harder. For example, if this function is called multiple times concurrently, the two messages may not be one after the other in the logs, making the debugging process more complex.

As a rule of thumb, an error should be handled only once. Logging an error is handling an error, and so is returning an error. Hence, we should either log or return an error, never both.

Let's rewrite our implementation to handle errors only once:

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route,
error) {
            err := validateCoordinates(srcLat, srcLng)
            if err != nil {
                    return Route{}, err
            }

```

```
8               err = validateCoordinates(dstLat, dstLng)
9               if err != nil {
10                      return Route{}, err
11              }
12
13              return getRoute(srcLat, srcLng, dstLat, dstLng)
14      }
15
16      func validateCoordinates(lat, lng float32) error {
17              if lat > 90.0 || lat < -90.0 {
18                      return fmt.Errorf("invalid latitude: %f",
19 lat)
20              }
21              if lng > 180.0 || lng < -180.0 {
22                      return fmt.Errorf("invalid longitude: %f",
23 lng)
                }
                return nil
        }
```

Only return an error 1
Only return an error 2
Only return an error 3
Only return an error 4

In this version, each error is handled only once by being returned directly. Then, assuming the caller of `GetRoute` is handling the possible errors with logging, it would output the following message in case of an invalid latitude:

```
2021/06/01 20:35:12 invalid latitude: 200.000000
```

Is this new Go version of the code perfect? Not really. For example, the first implementation led to two logs in case of an invalid latitude. Still, we knew which call to `validateCoordinates` was failing: either the source or the target coordinates. Here, we lost this information. Hence, we need to add additional context to the error.

Let’s rewrite the latest version of our code using Go 1.13 error wrapping (we will omit `validateCoordinates` as it will remain unchanged):

```
func GetRoute(srcLat, srcLng, dstLat, dstLng float32) (Route, error) {
            err := validateCoordinates(srcLat, srcLng)
            if err != nil {
                    return Route{},
                            fmt.Errorf("failed to validate source coordinates: %w", err)
            }

            err = validateCoordinates(dstLat, dstLng)
            if err != nil {
                    return Route{},
                            fmt.Errorf("failed to validate target coordinates: %w", err)
            }

            return getRoute(srcLat, srcLng, dstLat, dstLng)
    }
```

Return a wrapper error 1
Return a wrapper error 2

Each error returned by `validateCoordinates` is now wrapped to provide additional contexts to the error, whether it’s related to the source or target coordinates. So if we run this new version, here’s what the caller would log in case of an invalid source latitude:

```
2021/06/01 20:35:12 failed to validate source coordinates:
            invalid latitude: 200.000000
```

With this version, we covered all the different cases: one single log, without losing any valuable information. In addition, each error is handled only once, which simplifies our code such as avoiding repeating error messages.

Handling an error should be done only once. As we have seen, logging an error is handling an error. Hence, we should either log or return an error. By doing this, we simplify our code and gain better insights into an error situation. Using error wrapping is the most convenient approach as it allows us to propagate the source error and add context to an error.

In the next section, we will see the appropriate way to ignore an error in Go.

7.6 Not handling an error

In some cases, we may want to ignore an error returned by a function. There should only be one way of doing it in Go if we face this case. Let's understand why.

We will consider the following example where we call a `notify` function that returns a single `error` argument. Yet, as we're not interested in this error, we will purposely omit any error handling:

```
func f() {
        // ...
        notify()
    }

    func notify() error {
        // ...
    }
```

Error handling is omitted 1

As we want to ignore the error, in this example, we just called `notify` without assigning its output to a classic `err` variable. There's nothing wrong

with this code from a functional standpoint: it will compile and run as expected.

However, from a maintainability perspective, it can lead to some issues. Let's consider a new reader arriving on this code. This reader notices that `notify` returns an error but it isn't handled by the parent function. How can he guess whether it was intentional or not? How can he know whether the previous developer forgot to handle it or did it purposely?

For these reasons, when we want to ignore an error in Go, there's only one way to write it:

```
_ = notify()
```

Instead of not assigning the error to a variable, we assigned it to the blank identifier. Compilation and runtime-wise, it doesn't change anything compared to the first piece of code. Yet, this new version makes explicit that we aren't interested in the error.

A comment can also accompany such a code. Not a comment to mention we ignored the error like so:

```
// Ignore the error
    _ = notify()
```

This comment duplicates what the code does and should be avoided. However, it might be a good idea to indicate the rationale why this error is ignored. For example:

```
// Notifications are sent in best effort.
    // Hence, it's accepted to miss some of them in case of errors.
    _ = notify()
```

Ignoring an error in Go should be exceptional. In many cases, we may still favor logging them, even at a low log level. Yet, if we are sure that an error can and should be ignored, we must do it explicitly by assigning it to the blank identifier. This way, a future reader will understand it was intentional.

Let's now discuss the last section of this chapter, how to handle errors returned by defer functions.

7.7 Not handling defer errors

Not handling errors in defer statements is a frequent mistake made by Go developers. Let's understand what the problem is and what are the possible solutions.

In the following example, we will implement a function to query a database to get the balance given a customer ID. We will use the `database/sql` and the `Query` method.

> **[Note] Note**
>
> Let's not delve too much into how this package works; we will do it in the standard library chapter in *SQL common mistakes*.

Here's a possible implementation (we will focus on the query itself, not the parsing of the results):

```go
const query = "..."

    func getBalance(db *sql.DB, clientID string) (
            float32, error) {
            rows, err := db.Query(query, clientID)
            if err != nil {
                    return 0, err
            }
            defer rows.Close()

            // Use rows
    }
```

Defer the call to rows.Close() 1

`rows` is a `*sql.Rows` type. It implements the `Closer` interface:

```go
type Closer interface {
            Close() error
    }
```

This interface contains a single `Close` method returning an error (we will also delve into this topic in *Not closing transient resources*). We mentioned in the previous section that errors should always be handled. Yet, in this case, the error returned by the defer call is ignored:

```go
defer rows.Close()
```

As discussed in the previous section, if we don't want to handle the error, we should ignore it explicitly using the blank identifier:

```go
defer func() { _ = rows.Close() }()
```

This version is more verbose but is better from a maintainability perspective as we explicitly marked that we ignore the error.

Yet, in such a case, instead of blindly ignoring all errors from defer calls, we should ask ourselves whether it's the best approach. In this case, calling `Close()` will return an error when it fails to free a database connection from the pool. Hence, ignoring this error is probably not what we want to do here. Most likely, a better option would be to log a message:

```
defer func() {
            err := rows.Close()
            if err != nil {
                    log.Printf("failed to close rows: %v", err)
            }
    }()
```

In this case, if closing `rows` fails, it will log a message so that we're aware of it.

Now, what if, instead of handling the error, we prefer to propagate it to the error of `getBalance` so that he can decide how to handle it? What about such an approach?

```
defer func() {
            err := rows.Close()
            if err != nil {
                    return err
            }
    }()
```

This implementation doesn't compile. Indeed, the `return` statement is associated to the anonymous `func()` function, not `getBalance`.

If we want to tie the error returned by `getBalance` to the error caught in the defer call, we must use named result parameters. Let's write the first version:

```
func getBalance(db *sql.DB, clientID string) (
            balance float32, err error) {
            rows, err := db.Query(query, clientID)
            if err != nil {
                    return 0, err
            }
            defer func() {
                    err = rows.Close()
            }()

            if rows.Next() {
                    err := rows.Scan(&balance)
                    if err != nil {
                            return 0, err
                    }
                    return balance, nil
            }
            // ...
    }
```

Assign the error to the output named parameter 1

Once the `rows` variable has been correctly created, we defer the call to `rows.Close()` in an anonymous function. This function assigns the error to the `err` variable, initialized using named result parameters.

This code may look okay; however, there's a problem with it. Indeed, if `rows.Scan` returns an error, `rows.Close` will be executed anyways. Yet, as this call overrides the error returned by `getBalance`, instead of returning an error, we may return a nil error if `rows.Close` returns successfully. Said differently, if the call to `db.Query` succeeds (the first line of the function), the

error returned by `getBalance` will always be the one returned by `rows.Close`, which isn't what we want.

The logic to implement isn't straightforward. If `rows.Scan` succeeds:

- If `rows.Close` succeeds: return no error
- If `rows.Close` fails: return this error

However, if `rows.Scan` fails, the logic is a bit more complex as we may have to handle two errors:

- If `rows.Close` succeeds: return the error from `rows.Scan`
- If `rows.Close` fails: ?

In both `rows.Scan` and `rows.Close` fail, what should we do? There are different options here. For example, returning a custom error that conveys two errors. Another one, which we will implement, could be to return the `rows.Scan` error but log the `rows.Close` one. Here's our final implementation of the anonymous function:

```
1  defer func() {
2              closeErr := rows.Close()
3              if err != nil {
4                      if closeErr != nil {
5                              log.Printf("failed to close rows:
6  %v", err)
7                      }
8                      return
9              }
10             err = closeErr
       }()
```

Assign the rows.Close error to another variable 1
If err was already not nil, we priorize it 2
Otherwise, we return closeErr 3

The `rows.Close` error is assigned to another variable: `closeErr`. Before assigning it to `err`, we check whether `err` is different from nil. If that's the case, an error was already returned by `getBalance`. In that case, we decide to log it and return the existing error.

As discussed, errors should always be handled. In the case of errors returned by defer calls, the very least we should do is to ignore it explicitly. If this isn't enough, we can decide to handle the error directly by logging it or propagating the error up to the caller, as illustrated in this section.

7.8 Chapter summary

- Using panic is an option to deal with errors in Go. However, it should only be used sparingly in unrecoverable conditions. For example, to signal a programmer error or when we fail to load a mandatory dependency.
- Wrapping an error allows to mark an error and/or provide additional context. However, error wrapping creates potential coupling as it makes the source error available for the caller. If we want to prevent it, we shouldn't use error wrapping.
- If we use Go 1.13 error wrapping with the `%w` directive and `fmt.Error`, comparing an error against a type or a value has to be done respectively using `errors.As` or `errors.Is`. Otherwise, if the returned error we want to check is wrapped, it will fail the checks.
- To convey an expected error, we should use error sentinels (error values), whereas an unexpected error should be a specific error type.
- In most situations, an error should be handled only once. Logging an error is handling an error. Therefore, we have to choose between logging or returning an error. In many cases, error wrapping is the solution as it allows us to provide additional context to an error and return the source error.
- Ignoring an error, be it during a function call or a defer function, should be done explicitly using the blank identifier. Otherwise, future readers may be confused about whether it was intentional or a miss.
- In many cases, we shouldn't ignore an error returned by a defer function. We can either handle it directly or propagate it to the

caller, depending on the context. If we want anyway to ignore it, we should also use the blank identifier.

[13] github.com/go-sql-driver/mysql/

8 Concurrency: Foundations

This chapter covers

- Understanding the differences between concurrency and parallelism
- Discussing why concurrency isn't always faster
- Discussing the impacts of CPU-bound and IO-bound workloads
- Deciding when to use channels vs. mutexes
- Differences between a data race and a race condition
- Delving into Go contexts

During the past decades, CPU vendors have somehow stopped this race to focus solely on clock speed[14]. Instead, modern CPUs are designed with multiple cores and hyperthreading (multiple logical cores on the same physical core). Therefore, to leverage these architectures, concurrency has become critical for software developers. Even though Go provides *simple* primitives, it doesn't necessarily mean that writing concurrent code has become easy. While the next chapter will focus on practice, this chapter will first go through the fundamental concepts related to concurrency.

8.1 Mixing concurrency and parallelism

Even after years of concurrent programming, developers may not clearly understand the differences between concurrency and parallelism. Before delving into Go-specific topics, it's first essential to get these concepts to

share a common vocabulary. We will illustrate this whole section with a real-life example: a coffee shop.

In this coffee shop, there's one waiter in charge of accepting the orders and preparing the coffee with a single coffee machine. Customers are passing orders and then waiting for their coffee:

Figure 8.1. A simple coffee shop

If the waiter is having a hard time serving all the customers and wants to speed the overall process, one idea might be to have a second waiter and a second coffee machine. A customer in the queue would wait for one waiter to be available:

Figure 8.2. Duplicate everything

In this new process, every part of the system is now independent. The coffee shop should serve consumers twice as fast. This is a *parallel* implementation of a coffee shop.

If we want to scale, we can keep duplicating waiters and coffee machines over and over. However, this isn't the only possible coffee shop design. Indeed, another possible design might be to split the work made by the waiters and have one in charge of accepting the orders and another one preparing the coffee. Also, instead of blocking the customer queue until a customer is served, we could introduce another queue for customers waiting for their coffee (think about Starbucks):

Figure 8.3. Split the role of the waiters

With this new design, we didn't make things parallel. Yet, what was impacted is the overall structure: we split a given role into two roles, and we introduced another queue. Unlike parallelism, which is about doing the same thing multiple times at once, *concurrency* is about structure.

Assuming one thread represents the waiter accepting orders, and another represents the coffee machine, we introduced another one to prepare the

coffee. Each thread is independent but has to coordinate with others. Here, the waiter thread accepting orders has to communicate which coffee to prepare. Meanwhile, the preparator threads have to communicate with the coffee machine thread:

What if we want to increase the throughput (service more customers during one hour, for example). As preparing a coffee is slower than accepting orders, a possible change could be to hire another waiter to prepare the coffee:

Figure 8.4. Hire another waiter to prepare the coffee

Here, the structure remains the same. Indeed, it remains a 3-step design: accept, prepare, make coffee. Hence, no changes in terms of concurrency. Yet, we are back to adding parallelism; here for one particular step: the coffee preparation.

Now, let’s assume that the part slowing down the whole process is the coffee machine. Indeed, having a single coffee machine introduces some contentions for the preparator threads as they both wait for a coffee machine thread to be available. What could be a solution? Adding more coffee machine threads:

Figure 8.5. Add more coffee machine

Instead of a single coffee machine, we have increased the level of parallelism by introducing more machines. Again, the structure hasn’t changed; it remains a 3-step design. Yet, we should have increased the throughput as the level of contention for the preparator thread should have decreased.

With this design, we can notice something important: *concurrency enables parallelism*. Indeed, concurrency provides a structure to solve a problem with

parts that may be parallelized.

> Concurrency is about dealing with lots of things at once.
> Parallelism is about doing lots of things at once.
>
> -- Rob Pike

In summary, concurrency and parallelism are different. Concurrency is about structure, and we can change a sequential implementation into a concurrent one by introducing different steps that separate concurrent threads can tackle. Meanwhile, parallelism is about execution, and we can leverage it at the level of a step by adding more parallel threads. Understanding these two concepts is fundamental to being a proficient Go developer.

The next section will discuss a prevalent mistake: believing that concurrency is always the way to go.

8.2 Concurrency isn’t always faster

A common misconception made by many developers is to believe that a concurrent solution should always be faster than a sequential one, and it couldn’t be more wrong. The overall performance of a solution depends on many factors, such as the efficiency of our structure (concurrency), which parts can be tackled in parallel, and the level of contention among the computation units. In this section, we will remind us of some of the fundamental knowledge of concurrency in Go; then we will see a concrete example where a concurrent solution isn’t necessarily faster.

8.2.1 Go scheduling

A thread is the smallest unit of processing that an OS can perform. If a process wants to execute multiple actions simultaneously, it will spin up multiple threads. These threads can be:

- Concurrent when two or more threads can start, run, and complete in overlapping time periods. For example, the waiter thread and the coffee machine thread in the previous section.
- Parallel when the same task can be executed multiple times at once. For example, multiple waiter threads.

The OS is responsible for scheduling the thread's process most optimally so that:

- All the threads can consume CPU cycles without being starved for too much time
- The workload is distributed as evenly as possible among the different CPU cores

> **Note**
>
> The word thread can also have a different meaning at a CPU level. Each physical core can be composed of multiple logical cores (the concept of hyperthreading), and a logical core is also called a thread. In this section, when we use the word thread, it doesn't refer to a logical core but the unit of processing concept.

A CPU core executes different threads. When it switches from one thread to another, it executes an operation called context switching. The active thread consuming CPU cycles was in an *executing* state and moved to a *runnable* state, meaning ready to be executed but pending an available core. Context switching is considered an expensive operation as the OS needs to save the

current execution state of a thread before the switch (e.g., the current register values).

As Go developers, we can't create threads directly, but we can create goroutines, which can be thought of as application-level threads. However, if an OS thread is context-switched on and off a CPU core by the OS, a goroutine is context-switched on and off an OS thread by the Go runtime. Also, compared to an OS thread, a goroutine has a smaller memory footprint: 2 KB for goroutines from Go 1.4, an OS thread depends on the OS, but for example, on Linux/x86-32, the default size is 2 MB[15]. Having a smaller size makes it also faster to context switch.

> **Note**
>
> Context switching a goroutine vs. a thread is about 80% to 90% faster depending on the architecture.

Let's now delve into how the Go scheduler works to overview how goroutines are handled.

Internally, the Go scheduler uses the following terminology[16]:

- G: goroutine
- M: OS thread (stands for machine)
- P: CPU core (stands for processor)

Each OS thread (M) is assigned to a CPU core (P) by the OS scheduler. Then, each goroutine (G) runs on an OS thread (M). The `GOMAXPROCS` variable defines the limit of OS threads (M) in charge of executing user-level code simultaneously. Yet, if a thread is blocked in a system call (e.g., I/O), the

scheduler can spin up more OS threads (M). As of Go 1.5, `GOMAXPROCS` is by default equal to the number of available CPU cores.

A goroutine has a simpler lifecycle than an OS thread. It can be either:

- Executing: the goroutine is scheduled on an M and executing its instructions
- Runnable: waiting for being in an executing state
- Waiting: stopped and pending for something to complete, such as a system call or a synchronization operation (e.g., mutex)

There's one last stage to understand about Go scheduling implementation: when a goroutine is created but cannot be executed yet. For example, all the other Ms are already executing a G. In this scenario, what will the Go runtime do about it? The answer is queuing. Indeed, the Go runtime handles two kinds of queues: one local queue per P and a global queue shared among all the Ps.

In figure Figure 8.6. An example of the current state of a Go application executed on a 4-core machine. Goroutines that aren't in an executing state are either runnable (pending to be executed) or waiting (pending a blocking operation).">figure 8.6, we will look at a given scheduling situation on a 4-core machine and `GOMAXPROCS` equal to 4. The different parts are the logical cores (P), the goroutines (G), the OS threads (M), the local queues, and the global queue:

Figure 8.6. An example of the current state of a Go application executed on a 4-core machine. Goroutines that aren't in an executing state are either runnable (pending to be executed) or waiting (pending a blocking operation).

First, we can notice five Ms, whereas `GOMAXPROCS` was set to 4. Yet, as we mentioned, if needed, the Go runtime can create more OS threads than the `GOMAXPROCS` value.

P0, P1 and P3 are currently busy executing Go runtime threads. Yet, P2 is presently idle as M3 is switched off P2, and there's no goroutine to be executed. This isn't a good situation as there are in total six runnable goroutines pending to be executed. Some in the global queue, some in other local queues. How will the Go runtime handle this situation? Here's in pseudo-code the scheduling implementation[17]:

```
runtime.schedule() {
        // Only 1/61 of the time, check the global runnable queue for a G.
        // If not found, check the local queue.
        // If not found,
        //     Try to steal from other Ps.
        //     If not, check the global runnable queue.
        //     If not found, poll network.
    }
```

Every 1/61 execution, the Go scheduler will check whether goroutines from the global queue are available. If not, it will check its local queue. Meanwhile, if both the global and the local queues are empty, it can pick up goroutines from other local queues. This principle in scheduling is called work-stealing, and it allows an underutilized processor to actively look for other processor's goroutine and *steal* some.

One last important thing to mention, prior to Go 1.14, the scheduler was cooperative, which meant that a goroutine could be context-switched off a thread only in specific blocking cases (e.g., channel send or receive, I/O, waiting for a mutex). Since Go 1.14, the Go scheduler is now preemptive. It means that when a goroutine is running for a specific amount of time (10 ms), it will be marked preemptible and can be context-switched off to be replaced by another goroutine. It allows a long-running job to be forced to share CPU time.

Now that we have understood the fundamentals of scheduling in Go, let's delve into a concrete example: implementing a merge sort in a parallel manner.

8.2.2 Parallel Merge Sort

First, let's briefly review how the merge sort algorithm works. Then, we will implement a parallel version. Please note that the scope isn't to implement the most efficient version but to support a concrete example showing why concurrency isn't always faster.

The merge sort algorithm works by breaking down a list repeatedly into two sublists until each sublist consists of a single element and then merges these sublists so that the result is a sorted list:

Figure 8.7. Applying the merge sort algorithm repeatedly breaks down each list into two sublists. Then, it operates a merge operation so that the resulting list is sorted.

Each split operation splits a list into two sublists, whereas a merge operation merges two sublists into a sorted list.

Here is the sequential implementation of this algorithm. We won't write the whole code as it's not the main point of this section:

```
1  func sequentialMergesort(s []int) {
2              if len(s) <= 1 {
```

```go
                return
        }

        middle := len(s) / 2
        sequentialMergesort(s[:middle])
        sequentialMergesort(s[middle:])
        merge(s, middle)
}

func merge(s []int, middle int) {
        // ...
}
```

First half 1
Second half 2
Merge two halves 3

This algorithm has a structure that makes it open to concurrency. Indeed, as each `sequentialMergesort` operation works on an independent set of data that doesn't need to be fully copied (here, an independent view of the underlying array using slicing), we could distribute this workload among the CPU cores by spinning up each `sequentialMergesort` operation in a different goroutine. Let's write a first parallel implementation:

```go
func parallelMergesortV1(s []int) {
        if len(s) <= 1 {
                return
        }

        middle := len(s) / 2

        var wg sync.WaitGroup
        wg.Add(2)

        go func() {
                defer wg.Done()
                parallelMergesortV1(s[:middle])
        }()

        go func() {
                defer wg.Done()
```

```
                        parallelMergesortV1(s[middle:])
                }()

                wg.Wait()
                merge(s, middle)
        }
```

Spin up the first half of the work in a goroutine 1
Spin up the second half of the work in a goroutine 2
Merge halves 3

In this version, each half of the workload is handled in a separate goroutine. The parent goroutine waits for both parts using `sync.WaitGroup` to be executed before the merge operation.

> **Note**
>
> If you're not yet familiar with `sync.WaitGroup`, we will delve more into it in *Misusing sync.WaitGroup*. In a nutshell, it allows waiting for *n* operations to complete; usually, goroutines as in the previous example.

We now have a parallel version of the merge sort algorithm. Therefore, if we run a benchmark to compare this version against the sequential one, the parallel version should be faster, correct? Let's run it on a 4-core machine with 10k elements:

```
Benchmark_sequentialMergesort-4       2278993555 ns/op
    Benchmark_parallelMergesortV1-4      17525998709 ns/op
```

Surprisingly, the parallel version is almost an order of magnitude slower. How can we explain this result? How is it possible that a parallel version that

distributes a workload across four cores is slower than a sequential version running on a single machine? Let’s analyze the problem.

If we have a slice of, say, 1024 elements, the parent goroutine will spin up two goroutines, each in charge of handling a half consisting of 512 elements. Then, each of these goroutines will spin up two new goroutines in charge to handle 256 elements. Then, 128, and so on until we spin up a goroutine to compute a single element.

If the workload that we want to parallelize is too small, meaning we’re going to compute it too fast, the benefit of distributing a job across cores will be destroyed. Indeed, the time it takes to create a goroutine and have the scheduler execute it is way too high compared to directly merging a tiny number of items in the current goroutine. Although goroutines are lightweight and faster to start than threads, we can still face cases where a workload is too small.

> **Note**
>
> We will discuss how to get more insights about when an execution is poorly parallelized in *Not using Go diagnostics tooling*.

So how could we overcome this result? Does it mean that the merge sort algorithm cannot be parallelized? Wait, not so fast.

Let’s try another approach. Because merging a tiny number of elements within a new goroutine isn’t efficient, let’s define a threshold. This threshold will represent how big a half should be to handle it in a parallel manner. If the half is below this value, we will handle it sequentially. Here’s a new version:

```go
const max = 2048

    func parallelMergesortV2(s []int) {
            if len(s) <= 1 {
                    return
            }

            if len(s) <= max {
                    sequentialMergesort(s)
            } else {
                    middle := len(s) / 2

                    var wg sync.WaitGroup
                    wg.Add(2)

                    go func() {
                            defer wg.Done()
                            parallelMergesortV2(s[:middle])
                    }()

                    go func() {
                            defer wg.Done()
                            parallelMergesortV2(s[middle:])
                    }()

                    wg.Wait()
                    merge(s, middle)
            }
    }
```

Define the threshold 1

Call our initial sequential version 2

If bigger than the threshold, keep the parallel version 3

If the value is smaller than `max`, we call the sequential version. Otherwise, we keep calling our parallel implementation. Does it impact the result? Yes, it does:

```
Benchmark_sequentialMergesort-4       2278993555 ns/op
    Benchmark_parallelMergesortV1-4      17525998709 ns/op
    Benchmark_parallelMergesortV2-4       1313010260 ns/op
```

Our v2 parallel implementation is more than 40% faster than the sequential one, thanks to this idea to define a threshold to indicate when parallel should be more efficient than sequential.

> **Note**
>
> Why have I set the threshold to 2048? It was the most optimal value for this specific workload on my machine. In general, such magic values should be defined carefully with benchmarks (running on a similar execution environment as production). Another thing, it's pretty interesting to note that running the same algorithm on a programming language that doesn't implement the concept of goroutines has an impact on the value. Indeed, running the same example in Java using threads means an optimal value closer to 8192. It tends to illustrate how goroutines are more efficient than threads.

We have seen throughout this chapter the fundamental concepts of scheduling in Go: the differences between a thread and a goroutine and how the Go runtime schedules goroutines. Meanwhile, using the parallel merge sort example, we illustrated that concurrency isn't always necessarily faster. As we have seen, as we were spinning up goroutines to handle minimal workload (merging only a small set of elements), the benefit we could get from parallelism was getting demolished.

So, where should we go from here? First, we must keep in mind that concurrency isn't always faster and shouldn't be considered the default way to go for all the problems. First, it makes things more complex. Also, modern CPUs have become incredibly efficient in executing sequential code and predictable code. For example, a superscalar processor can parallelize instruction execution over a single core with a high-efficiency level.

Does it mean that we shouldn't use concurrency? Of course, not. However, it's essential to keep these conclusions in mind. If we're not sure that a parallel version will be faster, perhaps the right approach is first to start with a simple and sequential one and build from here thanks to profiling (*Not using Go diagnostics tooling*) and benchmarks (*Writing inaccurate benchmarks*), for example. It can be the only way to ensure that a concurrency is worth it.

The following section will discuss a very frequent question: when to use channels or mutexes?

8.3 Being puzzled about when to use channels or mutexes

Given a concurrency problem, it may not always be straightforward whether we could implement a solution using channels or mutexes. As Go promotes sharing memory by communication, one mistake could be to always force using channels, regardless of the use case. However, we should see both options as being complementary. Let's clarify throughout this section when we should favor one over the other option. The goal won't be to discuss every possible use case we may face (it would probably take a complete book) but to give general guidelines that can help us decide.

First, a brief reminder about channels in Go: channels are a communication mechanism. Internally, a channel is a pipe in which we can send and receive values and allows to *connect* concurrent goroutines. A channel can either be:

- Unbuffered: the sender goroutine blocks until the receiver goroutine is ready
- Buffered: the sender goroutine blocks only when the buffer is full

Let’s get back to our initial problem. When should we use channels or mutexes? We will take the following example as a backbone with three different goroutines with specific relationships:

Figure 8.8. Goroutine G1 and G2 are parallel, whereas G2 and G3 are concurrent.

In this example:

- G1 and G2 are parallel goroutines. Perhaps two goroutines executing the same function that keeps receiving messages from a channel or perhaps, two goroutines executing at the same time the same HTTP handler.
- On the other side, G1 and G3 are concurrent goroutines, so are G2 and G3. All the goroutines are part of an overall concurrent structure, but G1 and G2 do the first step, whereas G3 does the next step.

In general, parallel goroutines have to *synchronize*. For example, when they need to access or mutate a shared resource such as a slice. Synchronization is

enforced with mutexes but not with any channel types (not with buffered channels). Hence, in general, synchronization between parallel goroutines should be achieved via mutexes.

Conversely, in general, concurrent goroutines have to *coordinate and orchestrate*. For example, if G3 needs to aggregate results from both G1 and G2. In that case, G1 and G2 need to signal to G3 that a new intermediate result is available. This coordination falls into the scope of communication, therefore, channels.

Regarding concurrent goroutines, there's also the case where we want to transfer the ownership of a resource from one step (G1 and G2) to another (G3). For example, if G1 and G2 are enriching a shared resource and at some point, we consider this job as complete. Here, we should use channels to signal a specific resource is ready and handle the ownership transfer.

Mutexes and channels have different semantics. Whenever we want to share a state or access a shared resource, mutexes ensure exclusive access to this resource. Conversely, channels are a mechanic for signaling with or without data (`chan struct{}` or not). Coordination or ownership transfer should be achieved via channels. It's important to know whether goroutines are parallel or concurrent as in general, we should rather need mutexes for parallel goroutines and channels for concurrent ones.

Let's now delve into a widespread topic regarding concurrency: race problems.

8.4 Not understanding race problems

Race problems can be among the hardest and the most insidious bugs a programmer can face. As Go developers, we must understand crucial aspects

such as the data races and race conditions, their possible impacts, and how to avoid them. We will go through these questions by first discussing data races vs. race conditions, and then we will delve into the Go memory model and explain why it matters.

8.4.1 Data races vs. race conditions

Let's first focus on data races. A data race occurs when two or more goroutines simultaneously access the same memory location, and at least one is writing. Here is an example where two goroutines increment a shared variable:

```
i := 0

    go func() {
            i++
    }()

    go func() {
            i++
    }()
```

Increment i 1

If we run this code using Go race detector (`-race` option), it will warn us that a data race has occurred:

```
==================
    WARNING: DATA RACE
    Write at 0x00c00008e000 by goroutine 7:
      main.main.func2()

    Previous write at 0x00c00008e000 by goroutine 6:
      main.main.func1()
    ==================
```

Also, the final value of `i` is unpredictable. Sometimes, it can be 1, sometimes 2.

What's the issue with this code? The `i++` statement can be decomposed into three operations:

- Read `i`
- Increment the value
- Write back to `i`

If the first goroutine executes and completes before the second one, here's what would happen:

Goroutine 1	Goroutine 2	Operation	`i`
			0
Read		←	0
Increment			0
Write back		→	1
	Read	←	1

Goroutine 1	Goroutine 2	Operation	`i`
	Increment		1
	Write back	→	2

The first goroutine reads, increments, and writes back to `i` the value 1. Then, the second goroutine performs the same set of actions but starts from 1. Hence, the final result written to `i` would be 2.

However, there's no guarantee that the first goroutine will either start or complete before the second one in the previous example. We can also face the case of an interleaved execution where both goroutines will run concurrently and compete for accessing `i`. Here's another possible scenario:

Goroutine 1	Goroutine 2	Operation	`i`
			0
Read		←	0
	Read	←	0
Increment			0

Goroutine 1	Goroutine 2	Operation	`i`
	Increment		0
Write back		→	1
	Write back	→	1

First, both goroutines read from `i` and get the value 0. Then, both increment it and write back their local result: 1, which isn't the expected result.

This is a possible impact of a data race. If two goroutines simultaneously access the same memory location with one at least writing to that memory location, it can lead to a hazard. Even worse, in some conditions, it may even be possible for the memory location to end up holding a value containing a meaningless combination of bits.

> **Note**
>
> We will see in *Not enabling the race flag* how Go can help us detect data races.

So how should we prevent a data race from happening? Let's see different techniques. The scope here isn't to present all the possible options (for example, we will omit `atomic.Value`) but to give the main ones.

The first option could be to make the increment operation atomic, meaning done in a single operation. This way, it would prevent entangled running operations:

Goroutine 1	Goroutine 2	Operation	`i`
			0
Read and increment		↔	1
	Read and increment	↔	2

Even if the second goroutine runs before the first one, the result will remain 2.

Atomic operations can be done in Go using the `atomic` package. Here's an example of how we could increment atomically an `int64`:

```
var i int64

    go func() {
            atomic.AddInt64(&i, 1)
    }()

    go func() {
            atomic.AddInt64(&i, 1)
    }()
```

Increment i atomically 1
Same 2

Both goroutines update `i` atomically. An atomic operation can't be interrupted. Thus, preventing two accesses at the same time. Regardless of the goroutines order execution, eventually, `i` will be equal to 2.

> **Note**
>
> The `atomic` package doesn't provide primitives for `int` but for `int32`, `int64`, `uint32`, and `uint64`. This is why `i` is an `int64` in this example.

Another option would be to synchronize the two goroutines with an ad-hoc data structure like a mutex. Mutex stands for mutual exclusion, and it ensures that at most one goroutine accesses a so-called critical section. In Go, the `sync` package provides a `Mutex` type:

```
i := 0
    mutex := sync.Mutex{}

    go func() {
            mutex.Lock()
            i++
            mutex.Unlock()
    }()

    go func() {
            mutex.Lock()
            i++
            mutex.Unlock()
    }()
```

```
Start of the critical section 1
Increment i 2
End of the critical section 3
```

In this example, incrementing `i` is the critical section. Regardless of the goroutines ordering, this example also produces a deterministic value for `i`: 2.

Which approach works best? The boundary is pretty straightforward. As we mentioned, the `atomic` package works only with specific types. If we want something else (for example, slices, maps, and structs), we can't rely on `atomic` anymore.

Another possible option is to prevent sharing the same memory location and instead to favor communication across the goroutines. For example, we could decide to create a channel used by each goroutine to produce the value of the increment:

```
i := 0
    ch := make(chan int)

    go func() {
            ch <- 1
    }()

    go func() {
            ch <- 1
    }()

    i += <-ch
    i += <-ch
```

Notify to increment by one 1
Increment i from what's received from the channel 2

Each goroutine notifies via the channel that we should increment `i` by one. The parent goroutine collects the notifications and increments `i`. As it's the only goroutine writing to `i`, this solution is also data race-free.

Let’s sum up what we have seen so far. Data races occur when multiple goroutines access the same memory location simultaneously (e.g., the same variable), and at least one of them is writing. We have also seen how to prevent it with three synchronization approaches:

- Using atomic operations
- Protecting a critical section with a mutex
- Using communication and channels to ensure a variable is updated only by a single goroutine

In these three approaches, eventually, `i` will have its value set to 2, regardless of the execution ordering of the two goroutines. Yet, regardless of the operation, is that always the case? Does a data race-free application necessarily mean a deterministic result? Let’s explore this question with another example.

Instead of having two goroutines incrementing a shared variable, now each one will make an assignment. We will use the approach using a mutex to prevent data races:

```
i := 0
    mutex := sync.Mutex{}

    go func() {
            mutex.Lock()
            defer mutex.Unlock()
            i = 1
    }()

    go func() {
            mutex.Lock()
            defer mutex.Unlock()
            i = 2
    }()
```

The first goroutine assigns 1 to i 1

```
The second goroutine assigns 2 to i 2
```

The first goroutine assigns 1 to `i`, whereas the second one assigns 2.

Is there a data race in this example? No, there isn't. Both goroutines access to the same variable but not at the same time as the mutex protects it. Yet, is this example deterministic? No, it isn't.

Depending on the ordering execution, `i` will eventually be equal to either 1 or 2. This example doesn't lead to a data race. Yet, it has a *race condition*. A race condition occurs when the behavior depends on the sequence or the timing of events that can't be controlled. Here, the timing of events is the goroutines execution order.

Ensuring a specific sequence of execution among goroutines is a question of coordination and orchestration. If we want to ensure that we first go from state 0 to state 1, then from state 1 to state 2, we should find a way to guarantee that goroutines are executed in order. Channels can be a way to solve this problem. Also, if we coordinate and orchestrate, it can also ensure that a particular section is accessed by only one goroutine, which can also mean removing the mutex in the previous example.

In summary, when we work in concurrent applications, it's essential to understand that a data race is different from a race condition. A data race occurs when multiple goroutines simultaneously access the same memory location, and at least one of them is writing. A data race means an unexpected behavior. However, a data race-free application doesn't necessarily mean deterministic results. Indeed, an application can be free of data races but can still have its behavior depending on uncontrolled events (e.g., goroutines execution, how fast will a message be published to a channel, how long will

last a call to a DB); this is a race condition. Understanding both aspects is crucial to becoming proficient in designing concurrent applications.

Let's now delve into the Go memory model and understand why it matters.

8.4.2 The Go memory model

The previous section discussed three main techniques to synchronize goroutines: atomic operations, mutexes, and channels. However, there are some core principles that we should be aware of as Go developers. For example, regarding channels: the guarantees between a buffered and unbuffered channel are different. To avoid unexpected races because of a lack of understanding of the core specifications of the language, we have to delve into the Go memory model.

The Go memory model[18] is a specification that defines the conditions under which a read from a variable in one goroutine can be guaranteed to happen after a write to the same variable in a different goroutine. Said differently, the guarantees that developers should have in mind to avoid data races and force a deterministic output.

Within a single goroutine, there's no chance to lead to unsynchronized accesses. Indeed, the happens-before order is guaranteed by the order expressed by our program.

However, within multiple goroutines, we should bear in mind some of these guarantees. We will use the notation A < B to denote that event A happened before event B. Let's delve into these guarantees (some copied from the Go memory model):

- Creating a goroutine happens before the goroutine's execution begins. Therefore, reading a variable and then spinning a new goroutine up that writes to this variable doesn't lead to a data race:

```
i := 0
    go func() {
            i++
    }()
```

- Conversely, the exit of a goroutine isn't guaranteed to happen before any event. Thus, the following example has a data race:

```
i := 0
    go func() {
            i++
    }()
    fmt.Println(i)
```

Again, if we want to prevent the data race from happening, we should synchronize these goroutines.

- A send on a channel happens before the corresponding receive from that channel completes. In the next example, a parent goroutine increments a variable before a send while another goroutine reads it after a channel read.

```
i := 0
    ch := make(chan struct{})
    go func() {
            <-ch
            fmt.Println(i)
    }()
```

```
    i++
    ch <- struct{}{}
```

The ordering is the following:

```
variable increment < channel send < channel receive < variable
read
```

By transitivity, we can ensure that accesses to `i` are synchronized. Hence, free from data races.

- Closing a channel happens before a receive of this closure. The next example is similar to the previous one except that instead of sending a message, we close the channel:

```
i := 0
    ch := make(chan struct{})
    go func() {
            <-ch
            fmt.Println(i)
    }()
    i++
    close(ch)
```

Therefore, this example is also free from data races.

- The last one regarding channel is perhaps counter-intuitive at first sight: a receive from an unbuffered channel happens *before* the send on that channel completes.

First, let's look at an example not with an unbuffered but with a buffered channel. We will have two goroutines with the parent sends a message and reads a variable whereas the child updates this variable and receives from the channel:

```
i := 0
    ch := make(chan struct{}, 1)
    go func() {
            i = 1
            <-ch
    }()
    ch <- struct{}{}
    fmt.Println(i)
```

This example leads to a data race. Let's have a look at this example visually:

Figure 8.9. If the channel is buffered, it leads to a data race.

We can observe that both the read and the write to `i` may occur simultaneously; therefore, `i` isn't synchronized. Now, let's change the channel to an unbuffered one to illustrate the memory model guarantee:

```
i := 0
    ch := make(chan struct{})
    go func() {
            i = 1
            <-ch
    }()
    ch <- struct{}{}
    fmt.Println(i)
```

Make the channel unbuffered 1

Changing the channel type make this example data race-free:

Figure 8.10. If the channel is unbuffered, it doesn't lead to a data race.

Here, we can see the main difference: the write is guaranteed to happen before the read. Note that the arrows don't represent causality (of course, a receive is caused by a send). It represents the ordering guarantees of the Go memory model. As a receive from an unbuffered channel happens before a send, then the write to `i` will always occur before the read.

Throughout this section, we have covered the main guarantees of the Go memory model. Understanding these guarantees should be part of our core knowledge when writing concurrent code. Indeed, it can prevent us from making wrong assumptions that can lead to data races and/or race conditions.

The following section will discuss why it's important to understand a workload type.

8.5 Not understanding the concurrency impacts of a workload type

This section will discuss the impacts of a workload type in a concurrent implementation. Indeed, if a workload is CPU or IO-bound, we may not have to tackle it in the same manner. Let's first define these concepts and then delve into the impacts.

In programming, the execution time of a workload is either limited by:

- The speed of the CPU. For example, running a merge sort algorithm. The workload is called CPU-bound.
- The speed of I/O. For example, making a REST call or a query in a DB. The workload is called I/O bound.
- The amount of available memory. The workload is called memory bound.

Note

The latter is probably the rarest nowadays, given that memory has become very cheap over the past decades. Hence, this section will focus on the two first workload types: CPU or I/O bound.

Why is it important to classify a workload in the context of a concurrent application? Let's understand it alongside one concurrency pattern: worker pooling.

Let's consider the following example. We will implement a `read` function that accepts an `io.Reader` and reads from it repeatedly 1024 bytes. We will pass these 1024 bytes to a `task` function that will perform some tasks (we will see what kind of task later on). This `task` function returns an integer, and we have to return the sum of all the results. Here's a sequential implementation:

```
1  func read(r io.Reader) (int, error) {
2              count := 0
3              for {
4                      b := make([]byte, 1024)
5                      _, err := r.Read(b)
6                      if err != nil {
7                              if err == io.EOF {
8                                      break
9                              }
10                             return 0, err
11                     }
12                     count += task(b)
13             }
14             return count, nil
15     }
```

Reads 1024 bytes 1

Stop the loop when we reach the end 2

Increment count based on the result of the task function 3

This function creates a `count` variable, reads from the `io.Reader` input, calls `task`, and increment `count`. Now, what if we want to run all the `task` functions in a parallel manner?

One option can be to use the so-called worker pooling pattern. It involves creating a fixed size of workers (goroutines) which are going to poll tasks work from a common channel:

Figure 8.11. Each goroutine from the fixed pool receives from a shared channel.

First, we spin up a fixed pool of goroutines (we'll discuss how many afterward). Then, we create a shared channel in which we will publish tasks to it after each read to the `io.Reader`. Each goroutine from the pool receives from this channel, performs their work, and then atomically updates a shared counter.

Let's see a possible way to write it in Go, with a pool size of ten goroutines. Each goroutine will atomically update a shared counter:

```
1  func read(r io.Reader) (int, error) {
2              var count int64
3              wg := sync.WaitGroup{}
4              var n = 10
5
6              ch := make(chan []byte, n)
7              wg.Add(n)
8              for i := 0; i < n; i++ {
9                      go func() {
10                             defer wg.Done()
11                             for b := range ch {
12                                     v := task(b)
13                                     atomic.AddInt64(&count,
14 int64(v))
15                             }
16                     }()
17             }
18
19             for {
20                     b := make([]byte, 1024)
21                     // Read from r to b
22                     ch <- b
23             }
24             close(ch)
25             wg.Wait()
26             return int(count), nil
       }
```

Create a channel with a capacity equal to the pool 1
Add n to the wait group 2
Create a pool of n goroutines 3
Call the Done method once the goroutine has received from the channel 4
Each goroutine will receive from the shared channel 5
Publish a new task to the channel after every read 6
Wait for the wait group to complete before returning 7

In this example, we use `n` to define the pool size. We create a channel with the pool's same capacity and a wait group with a delta of `n`. This way, we reduce potential contention in the parent goroutine while publishing messages. Then, we iterate `n` times to create a new goroutine that will receive from the shared channel. Each message received will be handled by

executing `task` and incrementing the shared counter atomically. After having read from the channel, each goroutine decrements the wait group.

In the parent goroutine, we keep reading from the `io.Reader` and publish each task to the channel. Last but not least, we close the channel and wait for the wait group to complete (meaning all the child goroutines have completed their job) before returning.

Having a fixed number of goroutines limits the downsides discussed; it narrows the resources' impact and prevents an external system from being flooded. Now, the golden question: what should be the pool size value? The answer depends on the workload type.

If the workload is IO-bound, the answer mainly depends on the external system. How many concurrent accesses will the system cope with if we want to maximize the throughput?

If the workload is CPU-bound, a best practice is to rely on `GOMAXPROCS`. `GOMAXPROCS` is a variable that sets the number of OS threads allocated to running goroutines. By default, this value is set to the number of logical CPUs.

Note

We can use the `runtime.GOMAXPROCS(int)` function to update the value of `GOMAXPROCS`. However, calling it with zero as an argument doesn't change the value; it just returns the current one:

```
n := runtime.GOMAXPROCS(0)
```

So, what's the rationale for mapping the size of the pool to `GOMAXPROCS`?

Let's take a concrete example and say we will run our application on a 4-core machine; hence Go will instantiate four OS threads where goroutines will be executed. At first, things may not be ideal, and we could face such a scenario with four CPU cores and four goroutines but only one being executed:

Figure 8.12. At most one goroutine is running.

M0 is currently running a goroutine of the worker pool. Hence, these goroutines start to receive messages from the channel and execute their job. However, the three other goroutines from the pool aren't yet assigned to an M; hence they are in a runnable state. M1, M2, and M3 don't have any goroutines to run, so they remain off a core. Thus, only one goroutine is running.

Eventually, with the work-stealing concept already described, P1 may steal goroutines from the local P0 queue:

Figure 8.13. At most two goroutines are running.

Here P1 stole the three goroutines from P0. Also, in this situation, eventually, all the goroutines may be assigned by the Go scheduler to a different OS thread. Yet, there's no guarantee about when this should occur exactly. Yet, as one of the main goals of the Go scheduler is to optimize resources (here, the distribution of the goroutines), we should end up in such a scenario given the nature of the workloads.

However, this scenario is still not optimal as at most two goroutines are running. Let's say the machine is running solely our application (besides the OS processes), so P2 and P3 will be free. Therefore, eventually, the OS may move M2 and M3 this way:

Figure 8.14. At most four goroutines are now running.

Here, the OS scheduler decided to move M2 to P2 and M3 to P3. Again, there is no guarantee about when this situation will happen. However, given a machine executing only our 4-thread application, this should be the final picture.

Now the situation has changed; it has become optimal. The four goroutines are running in separated threads, and each tread on separated cores. Therefore, it will reduce the amount of context-switching both at goroutines and threads levels.

This global picture cannot be designed and requested from us, Go developers. However, as we have seen, we can enable it with favorable conditions in the case of CPU-bound workloads: having a worker pool based on `GOMAXPROCS`.

> **Note**
>
> If, in particular conditions, we want to have the number of goroutines bound to the number of CPU cores, then why not rely on `runtime.NumCPU()`, which returns the number of logical CPU cores? As we mentioned, `GOMAXPROCS` can be changed and potentially be lower than the number of CPU cores. In the case of a CPU-bound workload, if the number of cores is four but we only have three threads, we shouldn't spin up four goroutines but three. Otherwise, a thread would share its execution time among two goroutines, increasing the number of context switches.

When implementing the worker pooling pattern, we have seen that the most optimal number of goroutines in the pool depends on the workload type. If the workload executed by the workers is IO-bound, the value mainly depends on

the external system. Conversely, if the workload is CPU-bound, the most optimal number of goroutines is close to the number of available threads. It's why knowing the workload type (either I/O or the CPU) is crucial when designing concurrent applications.

Last but not least, let's bear in mind that we should validate our assumptions via benchmarks in most cases. Indeed, concurrency isn't straightforward, and it can be pretty easy to draw hasty assumptions that may eventually not be valid.

For the last section of this chapter, let's now delve into a crucial aspect to understand to be proficient in Go: contexts.

8.6 Misunderstanding Go contexts

Developers sometimes misunderstand the `context.Context` type despite being one of the key concepts of the language and being one of the foundations of concurrent code in Go. Let's delve into this concept and make sure to understand why and how to use it efficiently.

According to the official documentation[19]:

A Context carries a deadline, a cancellation signal, and other values across API boundaries.

Let's delve into this definition and understand all the concepts related to a Go context.

8.6.1 Deadline

A deadline refers to a specific point in time determined either with:

- A `time.Duration` from now (e.g., in 250 ms)
- A `time.Time` (e.g., on *2023-02-07 00:00:00 UTC*)

The semantic of a deadline conveys that an ongoing activity should be stopped if this deadline is met. An activity is, for example, an I/O request, or a goroutine waiting to receive a message from a channel

For example, let's consider an application that receives flight positions from a radar every four seconds. Once we receive a position, we want to share it with other applications interested only in the latest position. We have at our disposal a `publisher` interface containing a single method:

```
type publisher interface {
            Publish(ctx context.Context, position
flight.Position) error
    }
```

This method accepts a context and a position. We will assume that the concrete implementation will call a function to publish a message to a broker (e.g., using Sarama to publish a Kafka message). This function will be *context-aware*, meaning it can cancel a request once the context is canceled.

Assuming we don't receive an existing context, what should we provide to the `Publish` method for the context argument?

We have mentioned that all the applications are interested only in the latest position. Hence, the context that we will build should convey that after four seconds, if we haven't been able to publish a flight position, we should stop it:

```
1 type publishHandler struct {
2             pub publisher
3     }
4
5     func (h publishHandler) publishPosition(position
6 flight.Position) error {
7             ctx, cancel :=
8 context.WithTimeout(context.Background(), 4*time.Second)
9             defer cancel()
              return h.pub.Publish(ctx, position)
    }
```

Create the context that will timeout after 4 seconds 1
Defer the cancelation 2
Pass the created context 3

This code creates a context using the `context.WithTimeout` function. This function accepts a timeout and a context. Here, as `publishPosition` doesn't receive an existing context, we create one from an empty context with `context.Background`. Meanwhile, `context.WithTimeout` returns two variables, the context created and a cancellation `func()` function that will cancel the context once called. Passing the context created to the `Publish` method should make it return in at most four seconds.

Also, what's the rationale for calling the `cancel` function as a defer one? Internally, `context.WithTimeout` creates a goroutine that will be retained in memory for four seconds or until `cancel` is called. Therefore, calling `cancel` as a defer function means that when we exit the parent function, the context will be canceled, and the goroutine created will be stopped. It's a safeguard to not return in leaving retained objects in memory.

Let's now move to the second aspect of Go contexts: cancellation signals.

8.6.2 Cancellation Signal

Another use case for Go contexts is to carry a cancellation signal.

Let’s imagine we want to create an application that calls within another goroutine a `CreateFileWatcher(ctx context.Context, filename string)`. This function will create a specific file watcher that will keep reading from a file and catch updates. When the provided context expires or is canceled, this function handles it to close the file descriptor.

Last but not least, when the main returns, we want things to be handled gracefully by closing this file descriptor. Therefore, we need to propagate a signal.

A possible approach can be to use `context.WithCancel` that returns a context (first return variable) that would cancel once the cancel function (second return variable) is called:

```
1  func main() {
2              ctx, cancel :=
3  context.WithCancel(context.Background())
4              defer cancel()
5
6              go func() {
7                      CreateFileWatcher(ctx, "foo.txt")
8              }()
9
10             // ...
       }
```

Create a cancellable context 1
Defer the call to cancel 2
Call the function using the created context 3

When the main returns, it will call the `cancel` function to cancel the context passed to `CreateFileWatcher` so that the file descriptor is closed gracefully.

Let's now delve into the latest aspects of Go contexts: values.

8.6.3 Context Values

The last use case for Go contexts is to carry a key/value list. Before understanding the rationale behind it, let's first see how to use it.

A context conveying values can be created this way:

```
ctx := context.WithValue(parentCtx, "key", "value")
```

Just like `context.WithTimeout`, `context.WithDeadline`, and `context.WithCancel`, `context.WithValue` is created from a parent context (here `parentCtx`). Here, we created a new `ctx` context containing the same characteristics as `parentCtx` but also conveying a key and a value.

We can access the value using the `Value` method:

```
ctx := context.WithValue(context.Background(), "key", "value")
    fmt.Println(ctx.Value("key"))
```

```
value
```

The key and values provided are `any` types. For the value, we want to pass whatever types. However, why should the key be an empty interface as well and not a string, for example? In that case, it could lead to collisions. Indeed,

two functions from different packages could use the same string value as a key. Hence, the latter would override the former value. Consequently, a best practice while handling context keys is to create an unexported custom type:

```
package provider

    type key string

    const myCustomKey key = "key"

    func f(ctx context.Context) {
            ctx = context.WithValue(ctx, myCustomKey, "foo")
            // ...
    }
```

Here, the `myCustomKey` constant is unexported. Hence, there's no risk that another package using the same context could override the value already set. Even if another package creates the same `myCustomKey` based on a `key` type as well, it will be a different key.

So what's the point of having a context carrying a key/value list? As Go contexts are generic and mainstream, there are infinite use cases.

For example, if we use tracing, we may want different subfunctions to share the same correlation ID. One may consider this ID too invasive to be part of the function signature. In this regard, we could also decide to include it as part of the provided context.

Another example is if we want to implement an HTTP middleware. If you're not familiar with such a concept, a middleware is an intermediate function executed before serving a request:

Figure 8.15. Before reaching the handler, a request goes through the middleware configured.

In this example, we configured two middlewares that must be executed before executing the handler itself. If we want middlewares to communicate together, it has to go through the context handled in the `*http.Request`.

Let's write an example of a middleware that marks whether the source host is a valid one:

```
1  type key string
2
3      const isValidHostKey key = "isValidHost"
4
5      func checkValid(next http.Handler) http.Handler {
6              return http.HandlerFunc(func(w
7  http.ResponseWriter, r *http.Request) {
8                      validHost := r.Host == "acme"
9                      ctx := context.WithValue(r.Context(),
10 isValidHostKey, validHost)
11
12                     next.ServeHTTP(w, r.WithContext(ctx))
               })
       }
```

Create the context key 1
Check whether the host is valid 2
Creates a new context with a value to convey whether the source host is valid 3
Call the next step with the new context 4

First, we define a specific context key called `isValidHostKey`. Then, the `checkValid` middleware checks whether the source host is valid. This information will be conveyed in a new context, passed to the next HTTP step using `next.ServeHTTP` (the next step can be another HTTP middleware or the final HTTP handler).

This was an example to show how context with values can be used in concrete Go applications.

We have seen in the past sections how to create a context to carry a deadline, a cancellation signal, and/or values. We can use this context and pass it to *context-aware* libraries, meaning libraries exposing functions accepting a context. But now, let's say that we move to the other side; we have to create a library, and we want external clients to provide a context that could be canceled.

8.6.4 Catching a context cancellation

The `context.Context` type exports a `Done` method that returns a receive-only notification channel: `<-chan struct{}`. This channel is closed when the work associated to the context should be canceled. For example, the `Done` channel related to a context created with:

- `context.WithCancel` is closed when the cancel function is called
- `context.WithDeadline` is closed when the deadline has expired

One thing to mention, why should the internal channel be closed when a context is canceled or has met a deadline instead of receiving a specific value? Because the closure of a channel is the only channel action that all the consumer goroutines will receive. This way, all the consumers will be notified once a context is canceled or a deadline is reached.

Furthermore, `context.Context` exports an `Err` method that returns nil if the `Done` channel isn't yet closed; otherwise, it returns a non-nil error explaining why the `Done` channel was closed, for example:

- A `context.Canceled` error if the channel was cancelled
- A `context.DeadlineExceeded` error if the context's deadline passed

Let's see a concrete example where we would like to keep receiving messages from a channel. Meanwhile, our implementation should be context-aware and return if the provided context is done:

```
1  func handler(ctx context.Context, ch chan Message) error {
2              for {
3                      select {
4                      case msg := <-ch:
5                              // Do something with msg
6                      case <-ctx.Done():
7                              return ctx.Err()
8                      }
9              }
10     }
```

Keep receiving messages from ch 1
If the context is done, return the error associated with it 2

We created a for loop, and we used `select` with two cases: either receiving messages from `ch` or receiving a signal that the context is done and that we have to stop our job. While dealing with channels, this is an example of how to make a function context-aware.

Note

Within a function that receives a context conveying a possible cancellation or timeout, the actions of receiving or sending a message to a channel shouldn't be done in a blocking way. For example, in the following function, we will receive from a channel and send a message to another:

```
func f(ctx context.Context) error {
            // ...
            ch1 <- struct{}{}

            v := <-ch2
            // ...
    }
```

Receive 1
Send 2

The problem with this function is that if the context is canceled or times out, we may have to wait until a message is sent or receive for nothing. Instead, we should use `select` to either wait for the channel actions to complete or to wait for the context cancellation:

```
```

```go
func f(ctx context.Context) error {
        // ...
        select {
        case <-ctx.Done():
                return ctx.Err()
        case ch1 <- struct{}{}:
        }

        select {
        case <-ctx.Done():
                return ctx.Err()
        case v := <-ch2:
                // ...
        }
}
```

Send a message to ch1 or wait for the context to be canceled 1
Receive a message from ch2 or wait for the context to be canceled 2

With this new version, if `ctx` is canceled or times out, we return immediately, without blocking the channel send or receive.

In summary, if we want to be a proficient Go developer, we have to understand what a context is and how to use it. Indeed, in Go, `context.Context` is everywhere, be it in the standard or external libraries. As we mentioned, a context allows carrying a deadline, a cancellation signal, and/or a list of key/values. In general, a function that users wait for should take a context, as it allows the upstream callers to decide when it should be aborted.

When in doubt about which context to use, we should use `context.TODO()` instead of passing an empty context with `context.Background`. Indeed, `context.TODO()` returns an empty context, but semantically, it conveys that the context to be used is either unclear or not yet available (because not yet propagated by a parent, for example).

Last but not least, let’s note that the available contexts in the standard library are all safe for concurrent use by multiple goroutines.

8.7 Chapter summary

- Understanding the fundamental differences between concurrency and parallelism is one of the cornerstones. Concurrency is about structure, whereas parallelism is about execution.
- To be a proficient developer, we have to acknowledge that concurrency isn't always faster. Solutions involving parallelization of minimal workload may not necessarily be faster than a sequential implementation. Benchmarking sequential vs. concurrent solutions should be the way to validate our assumptions.
- When creating a certain number of goroutines, we should take into consideration the workload type. Creating CPU-bound goroutines means bounding this number close to the `GOMAXPROCS` variable (based by default on the number of CPU cores on the host). Creating IO-bound goroutines depends on other factors, such as the external system.
- Being aware of goroutine interactions can also be helpful when deciding between channels and mutexes. In general, parallel goroutines require synchronization, hence mutex. Conversely, in general, concurrent goroutines require coordination and orchestration, hence channels.
- Being proficient in concurrency also means understanding that data races and race conditions are different concepts. Data races occur when multiple goroutines simultaneously access the same memory location, and at least one of them is writing. Meanwhile, data race-free doesn't necessarily mean a deterministic execution. When a behavior depends on the

sequence or the timing of events that can't be controlled, this is a race condition.

- Understanding the Go memory model and the underlying guarantees in terms of ordering and synchronization is essential to prevent us from possible data races and/or race conditions.
- Go contexts are also one of the cornerstones of concurrency in Go. A context allows carrying a deadline, a cancellation signal, and/or a list of key/values.

[14] www.gotw.ca/publications/concurrency-ddj.htm

[15] man7.org/linux/man-pages/man3/pthread_create.3.html

[16] github.com/golang/go/blob/go1.17.6/src/runtime/proc.go#L22

[17] github.com/golang/go/blob/go1.17.6/src/runtime/proc.go#L3291

[18] golang.org/ref/mem

[19] pkg.go.dev/context

9 Concurrency: Practice

This chapter covers

- Preventing common mistakes regarding the main Go concurrency primitives: goroutines and channels
- Understanding the impacts of using the standard data structures alongside concurrent code
- Delving into the standard library and some extensions
- Discussing subtle bugs that may lead to data races or even deadlock situations

In the previous chapter, we discussed the foundations regarding concurrency. Now, it's time to delve into practice and concrete mistakes made by Go developers when working with the concurrency primitives.

9.1 Propagating an inappropriate context

Contexts are omnipresent when working with concurrency in Go, and in many situations, it may be recommended to propagate them. However, sometimes context propagation can lead to subtle bugs, preventing subfunctions from being correctly executed.

Let's consider the following example. We expose an HTTP handler that performs some tasks and returns a response. However, just before returning the response, we want also to send it to a Kafka topic. As we don't want to penalize the HTTP consumer latency-wise, we want the publish action to be

handled asynchronously within a new goroutine. We assume that we have at our disposal a `publish` function, accepting a context so that the action of publishing a message can be interrupted if the context is canceled, for example.

Here would be a possible implementation:

```
1  func handler(w http.ResponseWriter, r *http.Request) {
2              response, err := doSomeTask(r.Context(), r)
3              if err != nil {
4                      http.Error(w, err.Error(),
5  http.StatusInternalServerError)
6                      return
7              }
8
9              go func() {
10                     err := publish(r.Context(), response)
11                     // Do something with err
12             }()
13
14             writeResponse(response)
       }
```

Perform some task to compute the HTTP response 1
Create a goroutine to publish the response to Kafka 2
Write the HTTP response 3

First, we call a `doSomeTask` function to get a `response` variable. It's used within the goroutine calling `publish` and to format the HTTP response. Also, when calling `publish`, we propagate the context attached to the HTTP request. Can you guess what's wrong with this piece of code?

We have to know that the context attached to an HTTP request can cancel in different conditions:

- When the client's connection closes
- In the case of an HTTP/2 request, when the request is canceled
- Or when the response has been written back to the client

In the first two cases, we probably handle things correctly. For example, if we just got a response from `doSomeTask`, but the client has closed the connection, it's probably OK to call `publish` with a context already canceled so that the message isn't published. But what about the last case?

When the response has been written to the client, the context associated with the request will be canceled. Therefore, we are facing a race condition:

- If writing the response is done after the Kafka publication, we both return a response and publish a message successfully
- However, if writing the response is done before or during the Kafka publication, the message may not be published

In the latter case, calling `publish` will return an error; just because we returned the HTTP response quickly.

So how can we fix this issue? One idea could be not to propagate the parent context. Instead, we would call `publish` with an empty context:

```
err := publish(context.Background(), response)
```

Here, that would work. Regardless of how long it takes to write back the HTTP response, we can call `publish`.

Yet, what if the context contained some useful values? For example, if the context contained a correlation ID used for distributed tracing, we can

correlate the HTTP request and the Kafka publication. Ideally, we would like to have a new context, detached from the potential parent cancellation, but that still conveys the values.

The standard package doesn't provide an immediate solution to this problem. Hence, a possible solution would be to implement our own Go context similar to the context provided, except that it wouldn't carry the cancellation signal.

A `context.Context` is an interface containing x methods:

```
type Context interface {
            Deadline() (deadline time.Time, ok bool)
            Done() <-chan struct{}
            Err() error
            Value(key any) any
    }
```

The context's deadline is managed by the `Deadline` method and the cancellation signal via the `Done` and the `Err` methods. When a deadline has passed, or the context has been canceled, `Done` should return a closed channel, whereas `Err` should return an error. Lastly, the values are carried via the `Value` method.

Let's create a custom context that would detach the cancelation signal from a parent context:

```
type detach struct {
            ctx context.Context
    }

    func (d detach) Deadline() (time.Time, bool) {
            return time.Time{}, false
    }
```

```go

func (d detach) Done() <-chan struct{} {
        return nil
}

func (d detach) Err() error {
        return nil
}

func (d detach) Value(key any) any {
        return d.ctx.Value(key)
}
```

Custom struct acting as a wrapper on top of the initial context 1
Delegate the get value call to the parent context 2

Except for the `Value` method that calls the parent context to retrieve a value, the other methods return a default value so that the context is never considered expired or canceled.

Thanks to our custom context, we can now call `publish` and detach the cancellation signal:

```go
err := publish(detach{ctx: r.Context()}, response)
```

Now, the context passed to `publish` is a context that would never expire nor get canceled but that would carry the parent context's values.

In summary, propagating a context should be done cautiously. We illustrated this section with an example of handling an asynchronous action based on a context associated with an HTTP request. As the context is canceled once we return the response, the asynchronous action can also get stopped unexpectedly. Let's bear in mind the impacts of propagating a given context

and if necessary, let's also keep in mind that it would always be possible to create our custom context for a specific action.

The following section will discuss a common concurrency mistake: starting a goroutine without plans to stop it.

9.2 Starting a goroutine without knowing when to stop it

Goroutines are easy and cheap to start. So easy and cheap that we may not necessarily have a plan about when to stop a new goroutine which can lead to leaks. Not knowing when to stop a goroutine is a design issue and a common concurrency mistake in Go. Let's understand why and how to prevent it.

First, let's quantify what a goroutine leak would mean. In terms of memory, a goroutine starts with a minimum of 2KB stack size, which can grow and shrink as needed (with a max stack size of 1 GB on 64-bit, 250 MB on 32-bit). Memory-wise, a goroutine can also hold variable references allocated to the heap. Meanwhile, a goroutine can hold resources such as HTTP or DB connections, open files, network sockets that should be closed gracefully eventually. If a goroutine is leaked, these kinds of resources will also be leaked.

Let's look at an example where the stop of a goroutine can be unclear. Here, a parent goroutine calls a function that returns a channel and then creates a new goroutine that will keep receiving messages from this channel:

```
ch := foo()
    go func() {
            for v := range ch {
                    // ...
            }
    }()
```

The created goroutine will exit when `ch` is closed. Yet, do we know exactly when this channel will be closed? It may not be evident as `ch` is created by the `foo` function. If the channel is never closed, then it's a leak. So we should always be cautious about the exit points of a goroutine and make sure one of those is eventually reached.

Now, let's discuss a concrete example. We will design an application that needs to watch some external configuration (e.g., using a DB connection). Here's a first implementation:

```
func main() {
        newWatcher()

        // Run the application
    }

    type watcher struct { /* Some resources */ }

    func newWatcher() {
        w := watcher{}
        go w.watch()
    }
```

Create a goroutine that watches some external configuration 1

We call `newWatcher`, which creates a `watcher` struct and spins up a goroutine in charge of watching the configuration. The problem with this code is that when the main goroutine exits (perhaps because of an OS signal or because it has a finite workload), the application will be stopped. Hence, the resources created by `watcher` won't be gracefully closed. How can we prevent this from happening?

One option could be to pass to `newWatcher` a context that will be cancelled when `main` returns:

```
1  func main() {
2              ctx, cancel :=
3  context.WithCancel(context.Background())
4              defer cancel()
5
6              newWatcher(ctx)
7
8              // Run the application
9      }
10
11     func newWatcher(ctx context.Context) {
12             w := watcher{}
13             go w.watch(ctx)
       }
```

Pass to newWatcher a context that will eventually cancel 1
Propagate this context 2

We propagate the context created to the `watch` method. When the context is canceled, the `watcher` struct should close its resources. However, can we guarantee that `watch` will have time to do so? Absolutely not, and that's a design flaw.

The problem here is that we used signaling to convey that a goroutine had to be stopped. We didn't block the parent goroutine until the resources had been closed. Let's make sure now that we do:

```
1  func main() {
2              w := newWatcher()
3              defer w.close()
4
5              // Run the application
6      }
7
```

```go
func newWatcher() watcher {
        w := watcher{}
        go w.watch()
        return w
}

func (w watcher) close() {
        // Close the resources
}
```

Defer the call the close method 1

`watcher` has a new method: `close`. Instead of signaling `watcher` that it's time to close its resources, we now call this `close` method using defer to guarantee the resources are closed before the application exits.

In summary, let's be mindful that a goroutine is a resource like any other which have to be eventually closed, be it to free memory or other resources. Starting a goroutine without knowing when to stop it is a design issue. Whenever a goroutine is started, we should have a clear plan about when it will stop. Last but not least, if a goroutine creates resources and its lifetime is bound to the lifetime of the application, it's probably safer to wait for it to be closed instead of notifying it. This way, we can ensure the resources are freed before exiting the application.

Let's now delve into one of the most common mistakes while working in Go: goroutines and loop variables.

9.3 Not being careful with goroutines and loop variables

Mishandling goroutines and loop variables is probably one of the most common ones made by Go developers when writing concurrent applications.

Let’s see a concrete example; then, we will define the conditions of such a bug and how to prevent it.

Let’s consider the following example. We will initialize a slice; then, within a closure executed as a new goroutine, we will access this element:

```
s := []int{1, 2, 3}

    for _, i := range s {
            go func() {
                    fmt.Print(i)
            }()
    }
```

Iterate over each element 1
Access the loop variable 2

One may expect this code to print `123` in no particular order (as there is no guarantee that a goroutine created first will complete first). However, the output of this code isn’t deterministic. For example, sometimes it can print `233`, sometimes `333`. What’s the reason?

In this example, we create new goroutines from a closure. As a reminder, a closure is a function value that references variables from outside its body; here the `i` variable. We have to know that when a closure goroutine is executed, it doesn’t capture the values when the goroutine is created. Instead, all the goroutines refer to the same exact variable. When a goroutine runs, it prints the value of `i` at the time `fmt.Println` is executed. Hence, `i` may have been modified since the goroutine was launched.

Let’s see a possible execution when the code prints `233`:

Figure 9.1. The goroutines access an `i` variable that isn't fixed but varies over time.

Over time, the value of `i` varies from 1, 2, then 3. In each iteration, we spin up a new goroutine. As there's no guarantee when each goroutine will start and complete, the result varies as well. In this example, the first goroutine prints `i` when it's equal to 2. Then, the other goroutines print it when the value is already equal to 3. Therefore, this example prints `233`. The behavior of this code isn't deterministic.

What are the solutions if we want each closure to access the value of `i` when the goroutine is created?

The first one, if we want to keep using a closure, involves creating a new variable:

```
for _, i := range s {
            val := i
            go func() {
                    fmt.Print(val)
            }()
    }
```

Create a variable local to each iteration 1

Why is this code working? In each iteration, we create a new local `val` variable. This variable captures the current value of `i` before creating the goroutine. Hence, when each closure goroutine executes the print statement, it does it with the expected value—this code prints `123` (again, in no particular order).

The second option is not to rely on a closure anymore but on an actual function:

```
for _, i := range s {
            go func(val int) {
                    fmt.Print(val)
            }(i)
    }
```

Execute a function that takes an integer as an argument 1
Call this function and passes the current value of i 2

We still execute as a goroutine an anonymous function (we don't run `go f(i)` for example), but this time it isn't a closure. The function doesn't reference `val` as a variable from outside its body; `val` is now part of the function input. By doing so, we fix `i` in each iteration and make our application work as expected.

We have to be cautious with goroutines and loop variables. If the goroutine is a closure that accesses an iteration variable declared from outside its body, it's a problem. We can fix it either by creating a local variable (for example, as we have seen using `val := i` before executing the goroutine) or by making the function no longer a closure. Both options work, and there's isn't one that we should favor over the other. One may find the closure approach handier, whereas one may find the function approach more expressive.

What happens with a `select` statement on multiple channels? Let's find it out.

9.4 Expecting a deterministic behavior using select and channels

One common mistake made by Go developers while working with channels is to make wrong assumptions about how `select` behaves with multiple channels. Such a false assumption can lead to subtle bugs that may be hard to identify and reproduce.

Let's imagine we want to implement a goroutine that needs to receive from two channels:

- `messageCh` for new messages to be processed.
- `disconnectedCh` to receive notifications conveying some disconnections. In that case, we want to return from the parent function.

Between those two channels, we want to prioritize `messageCh`. For example, if a disconnection occurs, we want to ensure that we have received all the messages before returning.

One may decide to handle the prioritization like so:

```
for {
            select {
            case v := <-messageCh:
                    fmt.Println(v)
            case <-disconnectCh:
                    fmt.Println("disconnection, return")
```

```
                        return
                }
        }
```

Use select statement to receive from multiple channels 1
Receive new messages 2
Receive disconnections 3

We use `select` to receive from multiple channels. As we want to prioritize `messageCh`, an assumption could be to write the `messageCh` case first and the `disconnectedCh` one next. But is this code even working? Let's give it a try by writing a dummy producer goroutine that will send ten messages and then send a disconnection notification:

```
for i := 0; i < 10; i++ {
                messageCh <- i
        }
        disconnectCh <- struct{}{}
```

If we run our previous example, here would be a possible output if `messageCh` is buffered:

```
0
        1
        2
        3
        4
        disconnection, return
```

So, instead of consuming the ten messages, we only received five of them; what's the reason? It lies in the specification of the `select` statement with multiple channels:

> If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection.
>
> -- Programming Language Specification

Unlike a `switch` statement where the first case with a match wins, the `select` statement will select one randomly if multiple options are possible.

This behavior might look odd at first, but there's a good reason for that: to prevent possible starvation. Indeed, suppose the first possible communication chosen is based on the source order. In that case, we may fall into the situation where we would solely receive from one single channel because of a fast sender, for example. To prevent this, the language designers have decided to use a random selection.

Coming back to our example, even though `case v := <-messageCh` was the first in source order, if there's a message in both `messageCh` and `disconnectCh`, there is no guarantee about which case will be chosen. For that reason, the behavior of the previous example isn't deterministic. We can either receive zero messages, or five, or even the ten messages: there's no guarantee about it.

So how can we try to overcome this situation? There are different options if we want to receive all the messages before returning in case of a disconnection.

If there's a single producer goroutine, we have two options:

- Either to make `messageCh` an unbuffered channel instead of a buffered one. As the sender goroutine blocks until the receiver goroutine is ready, it would guarantee that all the messages from `messageCh` are received before receiving the disconnection from `disconnectedCh`.
- Or to use a single channel instead of two. For example, we could define a `struct` that would either convey a new message or a disconnection. As channels guarantee that the order for the messages sent is the same as the messages received, we could ensure that the disconnection would be received in the end.

If we fall into the case where we have multiple producer goroutines, it may be impossible to guarantee which one writes first. Hence, be it with an unbuffered `messageCh` channel or a single channel, it will lead to a race condition among the producer goroutines. In that case, we could implement the following solution:

- Receive either from `messageCh` or `disconnectCh`
- If a disconnection is received:
    - Receive all the existing messages in `messageCh` if any
    - Then return

Here would be the solution:

```
1
```

```
2  for {
3                  select {
4                  case v := <-messageCh:
5                          fmt.Println(v)
6                  case <-disconnectCh:
7                          for {
8                                  select {
9                                  case v := <-messageCh:
10                                         fmt.Println(v)
11                                 default:
12
13 fmt.Println("disconnection, return")
14                                         return
15                                 }
16                         }
                   }
       }
```

Inner for/select 1
Receive the remaining messages 2
Then return 3

This solution uses an inner `for/select` with two cases, one on `messageCh` and a `default` case. Using `default` in a `select` statement is chosen only if none of the other cases match. In this case, it means we will return only once we have received all the remaining messages in `messageCh`.

Let's see how this code applies with a visual example. We will consider the case where we have two messages in `messageCh` and one disconnection in `disconnectCh`:

Figure 9.2. Initial state.

In this situation, as we have said, `select` will choose one or the other case randomly. Let's assume `select` chooses the second case:

Figure 9.3. Receive disconnection.

So, we receive the disconnection and now enter into the inner `select`:

Figure 9.4. Inner select.

Here, as long as messages remain in `messageCh`, `select` will always prioritize the first case over `default`:

Figure 9.5. Receive the remaining messages.

Then, once we have received all the messages from `messageCh`, `select` will not block but choose the `default` case:

Figure 9.6. Default case.

Hence, we will return and stop this goroutine.

This is a way to ensure we receive all the remaining messages from a channel with a receiver on multiple channels. Of course, if a `messageCh` is sent after the goroutine has returned (for example, if we have multiple producer goroutine), we will miss this message.

When using `select` with multiple channels, we must remember that if multiple options are possible, it's not the first case in the source order that will win. Instead, Go will select it randomly, so there's no guarantee about which one will be chosen. To overcome this behavior, in the case of a single producer goroutine, we can either use unbuffered channels or use a single channel instead. In the case of multiple producer goroutines, we can use inner selects and `default` to handle prioritizations.

The following section will discuss a common type of channel: notification channels.

9.5 Not using notification channels

Channels are a mechanism for communicating across goroutines via signaling. A signal can be either with or without data. However, it's not

always straightforward for Go programmers how the latter case should be tackled.

Let's delve into it with a concrete example. We will create a channel that would notify whenever a certain disconnection occurs. One idea could be to handle it as a `chan bool`:

```
disconnectCh := make(chan bool)
```

Now, let's say that you interact with an API that provides you with such a channel. As it's a channel of booleans, we can receive either true or false messages. It's probably clear what `true` conveys. However, what does false mean? Does it mean we haven't been disconnected, and in this case, how frequently are we going to receive such a signal? Does it mean that we have reconnected? Should we even expect to receive `false`? Perhaps, we should only expect to receive `true` messages.

If that's the case, meaning that we don't need a specific value to convey some information, we need a channel *without* data.

The idiomatic way to handle it is a channel of empty structs: `chan struct{}`.

In Go, an empty struct is a struct without any fields. Regardless of the architecture, it occupies zero bytes of storage as we can verify using `unsafe.Sizeof`:

```
var s struct{}
    fmt.Println(unsafe.Sizeof(s))
```

```
0
```

> **Note**
>
> We may wonder why not using an empty interface (`var i interface{}`). Yet, an empty interface isn't free; it occupies 8 bytes on 32-bit architecture and 16 bytes on 64-bit architecture.

An empty struct is a de-facto standard to convey an absence of meaning. For example, if we need a hash set structure (a collection of unique elements), we should use an empty struct as a value (e.g., `map[string]struct{}`).

Applied to channels, if we want to create a channel to send notification without data, the appropriate manner of doing it in Go is a `chan struct{}`. One of the most known utilization of a channel of empty structs comes with Go contexts, which we will delve into in this chapter.

A channel can be with or without data. If we want to design an idiomatic API in regards to Go standards, let's remember a channel without data should be expressed with a `chan struct{}` type. This way, it clarifies for receivers that they don't expect any meaning from a message content, only the fact that they have received a message. Such channels are called in Go notification channels.

For the next section, let's discuss how Go behaves with nil channels and what's the rationale for using them.

9.6 Not using nil channels

A mistake that is quite common while working in Go and channels is to forget that nil channels can sometimes be helpful. So what's a nil channel, and why should we care about them? That will be the scope of this section.

Let's start with a goroutine that will create a nil channel and wait for receiving a message. What should this code do?

```
var ch chan int
    <-ch
```

Nil channel 1

`ch` is a `chan int` type. The zero value of a channel being nil, `ch` is nil. Receiving a message on a nil channel is a valid operation. The goroutine won't panic; however, it will block forever.

This is the same principle if we send a message to a nil channel. This goroutine will also block forever:

```
var ch chan int
    ch <- 0
```

So what's the purpose to even allow receiving or sending messages to a nil channel then? We will discuss this question with a concrete example.

We will implement a `func merge(ch1, ch2 ←chan int) ←chan int` function to merge two channels into a single one. By merging them, we mean that each message received in either `ch1` or `ch2` will be sent to the channel returned:

Figure 9.7. Merging two channels into a single one.

So how can we do this in Go? Let's first write a naive implementation that spins up a goroutine and receives from both channels (the resulting channel will be a buffered channel of one element):

```go
func merge(ch1, ch2 <-chan int) <-chan int {
        ch := make(chan int, 1)

        go func() {
                for v := range ch1 {
                        ch <- v
                }
                for v := range ch2 {
                        ch <- v
                }
                close(ch)
        }()

        return ch
    }
```

Receive from ch1 and publish to the merged channel 1
Receive from ch2 and publish to the merged channel 2

Within another goroutine, we received from both channels, and each message ended up being published in `ch`.

The main issue with this first version is that we receive from `ch1` *then* we receive from `ch2`. It means that we won't receive from `ch2` until `ch1` is closed. Let's say that it doesn't fit our use case, as `ch1` may be open forever, so we want to receive from both channels simultaneously.

Let's write an improved version with concurrent receivers using `select`:

```go
func merge(ch1, ch2 <-chan int) <-chan int {
        ch := make(chan int, 1)

        go func() {
                for {
                        select {
                        case v := <-ch1:
                                ch <- v
                        case v := <-ch2:
                                ch <- v
                        }
                }
                close(ch)
        }()

        return ch
    }
```

Receive concurrently to both ch1 and ch2 1

The `select` statement lets a goroutine wait on multiple operations at the same time. As we wrap it inside of a `for` loop, we should repeatedly receive messages from one or the other channel, correct? But is this code even working?

One problem is that the `close(ch)` statement is unreachable. Indeed, looping over a channel using the `range` operator breaks when the channel is closed. However, the way we implemented a `for` / `select` doesn't catch when either `ch1` or `ch2` is closed. Even worse, if at some point `ch1` or `ch2` is closed,

here's what a receiver of the merged channel would receive when logging the value:

```
received: 0
    received: 0
    received: 0
    received: 0
    received: 0
    ...
```

So a receiver would repeatedly receive an integer equal to zero. What is the reason? Receiving from a closed channel is a non-blocking operation:

```
ch1 := make(chan int)
    close(ch1)
    fmt.Print(<-ch1, <-ch1)
```

Whereas one may expect this code to either panic or block, this code runs and prints `0 0`. Indeed, what we catch here is the closure event, not an actual message. To check whether we receive a message or a closure signal, we must do it this way:

```
ch1 := make(chan int)
    close(ch1)
    v, open := <-ch1
    fmt.Print(v, open)
```

Assign to open whether the channel is opened 1

Using the `open` boolean, we can now notice whether `ch1` is still opened:

```
0 false
```

Meanwhile, we also assigned 0 to `v` as it's the zero value of an integer.

Let's get back to our second solution. We said that it doesn't work very well as if `ch1` is closed; for example, because the `select` case was `case v := <-ch1`, we will keep entering this case and publishing a zero integer to the merged channel.

Let's take a step back and see what would be the best way to deal with this problem:

Figure 9.8. Handle different cases whether `ch1` or `ch2` is closed first.

We have to receive from both channels. Then, either:

- `ch1` is closed first, so we have to receive from `ch2` until it gets closed
- `ch2` is closed first, so we have to receive from `ch1` until it gets closed

How can we implement this in Go? Let's try to write a version that one may have done using a state machine approach and booleans:

```go
func merge(ch1, ch2 <-chan int) <-chan int {
            ch := make(chan int, 1)
            ch1Closed := false
            ch2Closed := false

            go func() {
                    for {
                            select {
                            case v, open := <-ch1:
                                    if !open {
                                            ch1Closed = true
                                            break
                                    }
                                    ch <- v
                            case v, open := <-ch2:
                                    if !open {
                                            ch2Closed = true
                                            break
                                    }
                                    ch <- v
                            }

                            if ch1Closed && ch2Closed {
                                    close(ch)
                                    return
                            }
                    }
            }()

            return ch
    }
```

Handle if ch1 is closed 1
Handle if ch2 is closed 2
Close and return if both channels are closed 3

We defined two booleans `ch1Closed` and `ch2Closed`. Once we receive a message from a channel, we check whether it's a closure signal. If yes, we

handle it by marking the channel as closed (e.g., `ch1Closed = true`). Once both channels are closed, we close the merged channel and stop the goroutine.

What is the problem with that code, apart from the fact that it's starting to get complex? There is one major issue: when one of the two channels is closed, the `for` loop will act as a busy-waiting one, meaning it will keep looping even though there's no new message received in the other channel. Indeed, we have to keep in mind the behavior of the `select` statement in our example. Let's say `ch1` was closed (so we won't receive any new message in here); once we reach `select` again, it will wait for one of these three conditions to happen:

- `ch1` is closed
- `ch2` has a new message
- `ch2` is closed

The first condition, `ch1` is closed, will always be valid. Therefore, as long as we don't receive a message in `ch2` or that it got closed, we will keep looping over the first case. This will lead to wasting CPU cycles and should be avoided. Therefore, our solution isn't viable.

We could try to enhance the state machine part and implement sub `for/select` loops within each case. But this would make our code even more complex and harder to understand.

It's the right time to come back to nil channels. As we mentioned, receiving from a nil channel is going to block forever. How about leveraging this idea in our solution? Instead of setting a boolean once a channel is closed, we will assign this channel to nil. Let's write the final version:

```go
func merge(ch1, ch2 <-chan int) <-chan int {
        ch := make(chan int, 1)

        go func() {
                for ch1 != nil || ch2 != nil {
                        select {
                        case v, open := <-ch1:
                                if !open {
                                        ch1 = nil
                                        break
                                }
                                ch <- v
                        case v, open := <-ch2:
                                if !open {
                                        ch2 = nil
                                        break
                                }
                                ch <- v
                        }
                }
                close(ch)
        }()

        return ch
    }
```

Continue if at least one channel isn't nil 1
Assign ch1 to a nil channel once closed 2
Assign ch2 to a nil channel once closed 3

First, we loop as long as at least one channel is still opened. Then, for example, if `ch1` is closed, we assign `ch1` to nil. Hence, during the next loop iteration, the `select` statement will now only wait for two conditions:

- `ch2` has a new message
- `ch2` is closed

`ch1` isn't even in the balance anymore as it's a nil channel. Meanwhile, we keep the same logic for `ch2` and assign it to nil once closed. Finally, when both channels are closed, we close the merged channel and return. Here's a model of this implementation:

Figure 9.9. Receive from both channels. If one is closed, assign it to nil so that we only receive from one channel.

This is the implementation we've been waiting for. We cover all the different cases, and it doesn't require a busy loop that will waste CPU cycles.

In summary, we have seen that waiting or sending to a nil channel is a blocking action, and this behavior isn't useless at all. As we have seen throughout the example of merging two channels, we can use nil channels to implement an elegant state machine that would *remove* one case from a `select` statement. Let's keep this idea in mind: nil channels are truly useful in some conditions and should be part of the Go developer toolset when it comes to dealing with concurrent code.

In the next section, we will discuss what size shall we set when creating a channel.

9.7 Being puzzled about a channel size

When creating a channel using the `make` built-in function, the channel can either be unbuffered or buffered. On this very topic, it's pretty frequent to find two common mistakes: being confused about when to use one or the other, and if we use a buffered channel, what size should we use? Let's delve into these points.

First, let's remember the core concepts. An unbuffered channel is a channel *without* any capacity. It can be created by either omitting the size or providing a zero size:

```
ch1 := make(chan int)
    ch2 := make(chan int, 0)
```

Using an unbuffered channel (also sometimes called synchronous channels), the sender will block until the receiver receives data from the channel.

Conversely, a buffered channel has a capacity, and it must be created with a size greater than or equal to one:

```
ch3 := make(chan int, 1)
```

With a buffered channel, a sender can send messages while the channel isn't full. Once the channel is full, it will block until a receiver goroutine receives a message. For example:

```
ch3 := make(chan int, 1)
    ch3 <-1
    ch3 <-2
```

```
Non-blocking 1
Blocking 2
```

The first send isn’t blocking, whereas the second one is as the channel is full at this stage.

Let’s take a step back and discuss the fundamental differences between these two channel types.

Channels are a concurrency abstraction to enable communication among goroutines. Yet, what about synchronization? In concurrency, synchronization means that we can guarantee that multiple goroutines will be in a known state at some point. For example, a mutex provides synchronization as it ensures that only one goroutine can be in a critical section at the same time. Regarding channels:

- An unbuffered channel enables synchronization. Indeed, we have the guarantee that two goroutines will be in a known state: one receiving and another one sending a message.
- Yet, a buffered channel doesn’t provide any strong synchronization. Indeed, a producer goroutine can send a message and then continue its execution if the channel isn’t full. The only guarantee is that a goroutine won’t receive a message before it is sent. Yet, this is only a guarantee because of causality (you don’t drink your coffee before having prepared it).

It’s essential to keep in mind this fundamental distinction. Both channel types enable communication, but only one provides synchronization. If we need synchronization, we must use unbuffered channels. Also, unbuffered channels

might be easier to reason about. Indeed, buffered channels can lead to obscure deadlocks that would have been immediately apparent with unbuffered channels.

There are other cases where unbuffered channels are preferable. For example, in the case of a notification channel where the notification is handled via a channel closure (`close(ch)`). Here, using a buffered channel wouldn't bring any benefits.

But what if we need a buffered channel? What size shall we provide?

The default value we should use for buffered channels is this minimum: one. So we may approach the problem from this standpoint: is there any good reason not to use a value of one? Here's a list of possible cases where we should use another size:

- While using a worker pooling-like pattern, meaning spinning a fixed number of goroutines that need to send data to a shared channel. In that case, we can tie the channel size to the number of goroutines created.
- When using channels for rate-limiting problems. For example, if we need to enforce resource utilization by bounding the number of requests, we should set up the channel size according to the limit.

If we are outside of these cases, using a different channel size should be done cautiously. Indeed, it's pretty frequent to see codebase using some magic numbers for setting a channel size:

```
ch := make(chan int, 40)
```

Why 40? What's the rationale for it? Why not 50? Or even 1000? Setting such a value should be done for a good reason. Perhaps, it was decided following a benchmark or performance tests. In many cases, it's probably a good idea to comment on the rationale for such a value.

Let's bear in mind that deciding about an accurate queue size isn't an easy problem. First, it's a balance between CPU and memory. The lesser the value, the more CPU contention we can face. Yet, the bigger the value, the more memory will need to be allocated.

Another point to consider is the one brought in the LMAX Disruptor white paper[20]:

> Queues are typically always close to full or close to empty due to the differences in pace between consumers and producers. They very rarely operate in a balanced middle ground where the rate of production and consumption is evenly matched.
>
> -- LMAX Disruptor

So, it's rare to find a channel size that will be steadily accurate, which means an accurate value that won't lead to too much contention nor a waste of memory allocation.

This is why, except for the cases described, it's usually best to start with a default channel size of one. When unsure, we could still measure it using benchmarks, for example.

As with almost any topic in programming, exceptions can be found. Therefore, the goal of this section wasn't to be exhaustive but to give

directions about what size we should use while creating channels. Synchronization is a guarantee brought with unbuffered channels, not buffered channels. Furthermore, if we need a buffered channel, we should remember that one should be the default value for the channel size. Having another value should be decided with care using an accurate process, and the rationale should probably be commented. Last but not least, let's remember that choosing buffered channels may also lead to potential obscure deadlocks that would have been easier to spot with unbuffered channels.

In the next section, we will discuss possible side-effects when dealing with string formatting.

9.8 Forgetting about possible side-effects with string formatting

Formatting strings is a common operation for developers, be it to return an error or log a message. However, it's pretty easy to forget the potential side-effects of string formatting while working in a concurrent application. This section will see two concrete examples: one taken from the etcd repository leading to a data race and another leading to a deadlock situation.

9.8.1 etcd data race

etcd is a distributed key-value store implemented in Go. It is used in many projects, including Kubernetes, to store all cluster data. It provides an API to interact with a cluster. For example, the `Watcher` interface is used to be notified upon data changes:

```
type Watcher interface {
            // Watch watches on a key or prefix. The watched
events will be returned
            // through the returned channel.
            // ...
```

```
            Watch(ctx context.Context, key string, opts
...OpOption) WatchChan
            Close() error
    }
```

The API relies on gRPC streaming. If you're not familiar with it, it's a technology to exchange data between a client and a server continuously. The server has to maintain a list of all the clients using this feature. Hence, the `Watcher` interface is implemented by a `watcher` struct containing all the active streams:

```
type watcher struct {
            // ...

            // streams hold all the active gRPC streams keyed
by ctx value.
            streams map[string]*watchGrpcStream
    }
```

The map's key is based on the context provided when calling the `Watch` method:

```
func (w *watcher) Watch(ctx context.Context, key string,
            opts ...OpOption) WatchChan {
            // ...
            ctxKey := fmt.Sprintf("%v", ctx)
            // ...
            wgs := w.streams[ctxKey]
            // ...
```

Format the map key depending on the provided context 1

`ctxKey` is the map's key, formatted from the context provided by the client. When formatting a string from a context created with values (`context.WithValue`), Go will try to access in reading all the values. In this case, the etcd developers have found that the context provided to `Watch` was a context containing mutable values (e.g., a pointer to a struct) in some conditions. They found a case where one goroutine was updating one of the context values, whereas another was executing `Watch`; hence accessing in reading to the context value. This led to a data race.

The fix[21] was to not rely on `fmt.Sprintf` to format the map's key to prevent traversing and reading the chain of wrapped values in the context. Instead, the solution was to implement a custom `streamKeyFromCtx` function to extract the key from a specific context value, which wasn't mutable.

> **Note**
>
> A potentially mutable value in a context can introduce additional complexity to prevent data races. This is probably a design decision to be considered with care.

This example illustrates that we have to be careful about the side effects of string formatting in concurrent applications, in this case, a data race. In the following example, we will see one leading to a deadlock situation.

9.8.2 Dead-lock

Let's say we have to deal with a `Customer` struct that can be accessed concurrently. We will use a `sync.RWMutex` to protect the accesses, whether reading or writing. We will implement an `UpdateAge` method to update the customer's age and check that the age is positive. Meanwhile, we will implement the `Stringer` interface.

Can you see what the problem is in this code with a `Customer` struct exposing an `UpdateAge` method and implementing the `fmt.Stringer` interface:

```
1  type Customer struct {
2              mutex sync.RWMutex
3              id    string
4              age   int
5      }
6
7      func (c *Customer) UpdateAge(age int) error {
8              c.mutex.Lock()
9              defer c.mutex.Unlock()
10
11             if age < 0 {
12                     return fmt.Errorf("age should be positive
13 for customer %v", c)
14             }
15
16             c.age = age
17             return nil
18     }
19
20     func (c *Customer) String() string {
21             c.mutex.RLock()
22             defer c.mutex.RUnlock()
23             return fmt.Sprintf("id %s, age %d", c.id, c.age)
       }
```

Use a sync.RWMutex to protect concurrent accesses 1
Lock and defer unlock as we update Customer 2
Return an error if age is negative 3
Lock and defer unlock as we read Customer 4

The problem here might not be that straightforward. If the provided `age` is negative, we return an error. As the error is formatted, using the `%s` directive on the receiver, it will call the `String` method to format `Customer`. Yet, as

`UpdateAge` already acquires the mutex lock, the `String` method won't be able to acquire it:

Figure 9.10. `UpdateAge` **execution if age is negative**

Hence, this situation leads to a deadlock situation. If all goroutines are also asleep, it leads to a panic:

```
fatal error: all goroutines are asleep - deadlock!

    goroutine 1 [semacquire]:

```

```
    sync.runtime_SemacquireMutex(0xc00009818c, 0x10b7d00, 0x0)
    ...
```

So how should we deal with this situation? First of all, it illustrates how unit testing is important. In that case, one may have argued that creating a test with a negative age isn't worth it as the logic is quite simple. However, without proper test coverage, we may have missed this issue.

One thing that could have been improved here is to restrict the scope of the mutex locking. Indeed, in the `UpdateAge`, we first acquire the lock and check whether the input is valid. We should have done the opposite: first checking the input, and if the input is valid, we should have acquired the lock. This has the benefits of reducing the potential side effects but can also have an impact performance-wise: a lock is acquired only when it's required, not before:

```
func (c *Customer) UpdateAge(age int) error {
            if age < 0 {
                    return fmt.Errorf("age should be positive
for customer %v", c)
            }

            c.mutex.Lock()
            defer c.mutex.Unlock()

            c.age = age
            return nil
    }
```

Lock the mutex only when input has been validated 1

In our case, locking the mutex only once the age has been checked avoids the deadlock situation. Indeed, if the age is negative, calling `String` is done without locking the mutex beforehand.

In some conditions, though, it’s not always straightforward nor possible to restrict the scope of a mutex lock. In these conditions, we have to be extremely careful with string formatting. Perhaps, we want to call another function that doesn’t try to acquire the mutex, or we want only to change our way to format the error so that it doesn’t call the `String` method. For example, the following code doesn’t lead to a deadlock as we only log the customer ID in accessing the `id` field directly:

```
func (c *Customer) UpdateAge(age int) error {
            c.mutex.Lock()
            defer c.mutex.Unlock()

            if age < 0 {
                    return fmt.Errorf("age should be positive
for customer id %s", c.id)
            }

            c.age = age
            return nil
    }
```

We have seen two concrete examples, one when formatting a key from a context and another when returning an error that will format a struct. In both cases, formatting a string led to a problem, a data race in the former example, and a deadlock situation in the latter. Therefore, in concurrent applications, we should remain cautious about the possible side effects of string formatting.

The following section will delve into the behavior of `append` while being called concurrently.

9.9 Creating data races with append

We mentioned in a previous section what a data race is and what the impacts are. Now, let’s delve into slices and discuss if adding an element to a slice

using `append` is data race-free. Spoiler? It depends.

In the following example, we will initialize a slice and create two goroutines that will use `append` to create a new slice with an additional element:

```
s := make([]int, 1)

    go func() {
            s1 := append(s, 1)
            fmt.Println(s1)
    }()

    go func() {
            s2 := append(s, 1)
            fmt.Println(s2)
    }()
```

In a new goroutine, append a new element on s 1
Same 2

Do you believe this example has a data race? The answer is no.

We have to recall some slice fundamentals that we described in the data types chapter. A slice being backed by an array, it has two properties: length and capacity. The length is the number of available elements in the slice, whereas the capacity is the total number of elements in the backing array. When we use `append`, the behavior depends on whether the slice is full (length == capacity). If yes, it creates a new backing array to add the new element; otherwise, it adds it to the existing backing array.

In this example, we created a slice with `make([]int, 1)`. This example creates a 1-length, 1-capacity slice. Thus, as the slice is full, using `append` is in each goroutine will return a slice backed by a new array. It doesn't mutate the existing array; hence it doesn't lead to a data race.

Now, let's run the same example with a slight change in how we initialized `s`. Instead of creating a slice with a length of one, we will create it with a length of zero but a capacity of one:

```
s := make([]int, 0, 1)

    // Same
```

Change the way the slice is initialized 1

How about this new example? Does it contain a data race? The answer is yes:

```
==================
    WARNING: DATA RACE
    Write at 0x00c00009e080 by goroutine 10:
      ...

    Previous write at 0x00c00009e080 by goroutine 9:
      ...
    ==================
```

Here, we created a slice with `make([]int, 0, 1)`. Therefore, the array isn't full. Consequently, both goroutines make an attempt to update the same index of the backing array (index 1), which is a data race.

So how can we prevent the data race from happening if we want both goroutines to work on a slice containing the initial elements of `s` plus an extra element? One solution is to create a copy of `s`:

```
 s := make([]int, 0, 1)

     go func() {
```

```
            sCopy := make([]int, len(s), cap(s))
            copy(sCopy, s)

            s1 := append(sCopy, 1)
            fmt.Println(s1)
    }()

    go func() {
            sCopy := make([]int, len(s), cap(s))
            copy(sCopy, s)

            s2 := append(sCopy, 1)
            fmt.Println(s2)
    }()
```

Make a copy and use append on the copied slice 1
Same 2

Both goroutines make first a copy of the slice. Then, they use `append` on the slice copy, not the original slice. Therefore, it prevents a data race from occurring as both goroutines work on isolated data.

Note

How much do data races impact slices and maps? Having multiple goroutines:

- Accessing the same slice index with at least one of them updating the value is a data race. Indeed, the goroutines are accessing the same memory location.
- Accessing different slice indices regardless of the operation isn't a data race; different indices mean different memory locations.
- Accessing the same map (regardless of whether it's the same or a different key) with at least one of them updating it is a data race. We may be wondering why

> it is different from a slice data structure? As we mentioned in the data types chapter, a map is an array of buckets, each bucket being a pointer to an array of key/value pairs. A hashing algorithm is used to determine the array index of the bucket. As this algorithm contains some randomness during the map initialization, one execution may lead to the same array index, whereas another execution may not. The race detector handles this case by raising a warning, regardless if an actual data race occurs or not.

While working with slices in concurrent contexts, we must recall that using `append` on slices isn't always race-free. Depending on the slice and whether it's full or not, the behavior will change. If the slice is full, `append` will be race-free. Otherwise, multiple goroutines may compete for updating the same array index, resulting in a data race.

In general, we shouldn't have a different implementation depending on whether the slice is full. We should consider that using `append` on a shared slice in concurrent applications may lead to a data race. Hence, it should be avoided.

Let's now discuss a common mistake with inaccurate mutex locks on top of slices and maps.

9.10 Using mutexes inaccurately with slices and maps

While working in concurrent contexts where data is both mutable and shared, we often have to implement protected accesses around data structures using mutexes. A common mistake is to use mutexes inaccurately when working

with slices and maps. Let's look at a concrete example and understand the potential problems.

We will implement a `Cache` struct used to handle caching for customer balances. This struct will contain a map of balances per customer ID and a mutex to protect concurrent accesses:

```
type Cache struct {
            mu       sync.RWMutex
            balances map[string]float64
    }
```

Note

This solution uses a `sync.RWMutex` to allow multiple readers as long as there are no writers.

Next, we will add an `AddBalance` method that will mutate the `balances` map. The mutation will be done in a critical section (within a mutex lock and a mutex unlock):

```
func (c *Cache) AddBalance(id string, balance float64) {
            c.mu.Lock()
            c.balances[id] = balance
            c.mu.Unlock()
    }
```

Meanwhile, we have to implement a method to calculate the average balance for all the customers. One idea that we may have is to handle a minimal critical section this way:

```go
func (c *Cache) AverageBalance() float64 {
        c.mu.RLock()
        balances := c.balances
        c.mu.RUnlock()

        sum := 0.
        for _, balance := range balances {
                sum += balance
        }
        return sum / float64(len(balances))
    }
```

Create a copy of the balances map 1
Iterate over the copy, outside of the critical section 2

First, we create a copy of the map to a local `balances` variable. Only the copy is done in the critical section to iterate over each balance and calculate the average outside of the critical section. However, is this solution working?

If we run a test using the `-race` flag with two concurrent goroutines, one calling `AddBalance` (hence mutating `balances`), and another calling `AverageBalance`, it does lead to a data race. So what's the problem here?

Internally, a map is a `runtime.hmap` struct containing mostly some metadata (e.g., a counter) and a pointer referencing data buckets. Hence, `balances := c.balances` doesn't copy the actual data. It's the same principle with a slice:

```go
s1 := []int{1, 2, 3}
    s2 := s1
    s2[0] = 42
    fmt.Println(s1)
```

Printing `s1` returns `[42 2 3]`, even though we modified `s2`. The reason is that `s2 := s1` creates a new slice: `s2` has the same length, the same

capacity, and is backed by the same array as `s1`.

Coming back to our example, we assigned to `balances` a new map referencing the same data buckets as `c.balances`. Meanwhile, the two goroutines will perform operations on the same data set, and one of them is mutating it. Hence, it's a data race.

How can we fix the data race? We have mainly two options.

If the iteration operation isn't heavy (it's the case here as we perform an increment operation), we should protect the whole function:

```
func (c *Cache) AverageBalance() float64 {
            c.mu.RLock()
            defer c.mu.RUnlock()

            sum := 0.
            for _, balance := range c.balances {
                    sum += balance
            }
            return sum / float64(len(c.balances))
    }
```

Unlock when the function returns 1

Here, the critical section now encompasses the whole function, which includes the iterations. This prevents data races.

Another option that can be used, if the iteration operation isn't a lightweight one, is to work on an actual copy of the data and to protect only the copy:

```
func (c *Cache) AverageBalance() float64 {
            c.mu.RLock()
            m := make(map[string]float64, len(c.balances))
```

```
        for k, v := range c.balances {
                m[k] = v
        }
        c.mu.RUnlock()

        sum := 0.
        for _, balance := range m {
                sum += balance
        }
        return sum / float64(len(m))
}
```

Copy the map 1

Here, once we have made a deep copy, we release the mutex. The iterations are done on the copy, outside of the critical section.

Let's think about this solution; here, we have to iterate twice on the map values: one to copy and one to perform the operations (here, the increments). Yet, the critical section is only the map copy. Therefore, this solution can be a good fit if and only if an operation isn't *fast*. For example, if an operation requires calling an external database, this solution will probably be more efficient. It's impossible to define a threshold when choosing one solution or the other as it depends on various factors such as the number of elements and the average size of the struct.

In summary, we have to be careful with the boundaries of a mutex lock. In this section, we have seen why assigning to a map an existing map (or an existing slice) isn't enough to protect data races. Indeed, the new variable, be it a map or a slice, is backed by the same data set. There are two leading solutions to prevent it, either to protect the whole function or to work on a copy of the actual data. In all cases, let's be cautious when designing critical sections and make sure the boundaries are accurately defined.

Let's now discuss a common mistake while using `sync.WaitGroup`.

9.11 Misusing `sync.WaitGroup`

`sync.WaitGroup` is a mechanism to wait for *n* operations to complete; usually, we use it to wait for *n* goroutines to complete. Let's first remind us of the public API; then, we will see a pretty frequent mistake leading to non-deterministic behavior.

A wait group can be created with the zero value of `sync.WaitGroup`:

```
wg := sync.WaitGroup{}
```

Internally, a `sync.WaitGroup` holds an internal counter initialized by default to zero. We can increment this counter using the `Add(int)` method and decrement it using `Done()` or `Add` with a negative value. Last but not least, if we want to wait for the counter to be equal to zero, we have to use the `Wait()` method that will be blocking.

> **Note**
>
> The counter cannot be negative; otherwise, the goroutine will panic.

In the following example, we will initialize a wait group, start three goroutines that will update a counter atomically, and then wait for them to complete. In the end, we want to wait for these three goroutines to print the value of the counter (which should be 3). Can you guess if there's an issue with this code?

```
1  wg := sync.WaitGroup{}
2      var v uint64
3
4      for i := 0; i < 3; i++ {
5              go func() {
6                      wg.Add(1)
7                      atomic.AddUint64(&v, 1)
8                      wg.Done()
9              }()
10     }
11
12     wg.Wait()
13     fmt.Println(v)
```

Create a goroutine 1
Increment the wait group counter 2
Atomically increment v 3
Decrement the wait group counter 4
Wait for all the goroutines to have incremented v before printing it 5

If we run this example, we will get a non-deterministic value, and it can print any value from 0 to 3. Also, we can note that if we enable the race flag, Go will even catch a data race. How is it even possible as we were using the `sync/atomic` package to update `v`? What's going wrong with this code?

The problem here is that `wg.Add(1)` is called within the newly created goroutine, not in the parent goroutine. Hence, there is no guarantee that we have indicated to the wait group that we wanted to wait for three goroutines before calling `wg.Wait()`.

Let's see a possible scenario when the code prints 2:

Figure 9.11. The last goroutine calls `wg.Add(1)` after the main goroutine is already unblocked.

In this scenario, the main goroutine spins up three goroutines. Yet, the last goroutine is executed after that the two first goroutines have already called `wg.Done()`. Hence, the parent goroutine is already unlocked. Therefore, in this scenario, when the main goroutine reads `v`, it's equal to 2. Also, the race detector can detect unsafe accesses to `v`.

When dealing with goroutines, it's crucial to remember that the execution isn't deterministic without synchronization. For example, the following code could print either `ab` or `ba`:

```
go func() {
        fmt.Print("a")
    }()
    go func() {
        fmt.Print("b")
    }()
```

Indeed, both goroutines can be assigned to different threads, and there's no guarantee about which thread will be executed first.

The CPU has to use a so-called memory fence (also called a memory barrier) to ensure order. Go provides different synchronization techniques for implementing memory fences such as`sync.WaitGroup` as it enables a happens-before relationship between `wg.Add` and `wg.Wait`.

Coming back to our example, there are two possible options to fix our issue. Either to call `wg.Add` before the loop with 3:

```
wg := sync.WaitGroup{}
    var v uint64

    wg.Add(3)
    for i := 0; i < 3; i++ {
            go func() {
                    // ...
            }()
    }

    // ...
```

Or to call `wg.Add` during each loop iteration before spinning up the child goroutines:

```
wg := sync.WaitGroup{}
    var v uint64

    for i := 0; i < 3; i++ {
            wg.Add(1)
            go func() {
                    // ...
            }()
    }

    // ...
```

Both solutions are fine. If the value that we want to set eventually to the wait group counter is known in advance, the first solution prevents us from having to call `wg.Add` multiple times. However, it requires making sure the same count is used in all the places to avoid subtle bugs.

Let's be cautious not to reproduce this common mistake made by Go developers. When using a `sync.WaitGroup`, the `Add` operation must be done before spinning up a goroutine in the parent one, whereas the `Done` operation must be done within the goroutine.

The following section will discuss another primitive of the `sync` package: `sync.Cond`.

9.12 Forgetting about `sync.Cond`

Among the synchronization primitives in the `sync` package, `sync.Cond` is probably the least used and understood. However, it brings features that we can't achieve with channels. This section will go through a concrete example to know when `sync.Cond` can be helpful and how to use it.

The example of this section aims to implement a donation goal mechanism, meaning an application that would raise alerts whenever specific goals are reached.

We will have one goroutine in charge of incrementing a balance (let's call it updater goroutine). In contrast, other goroutines will receive updates and print a message whenever a specific goal is reached (let's call it listener goroutines). For example, one goroutine is waiting for a $10 donation goal, whereas another is waiting for a $15 donation goal.

One first naive solution could be to use mutexes. The updater goroutine would increment the balance every second. On the other side, the listener goroutine would loop until their donation goal is met:

```go
type Donation struct {
        mu          sync.RWMutex
        balance int
    }
    donation := &Donation{}

    // Listener goroutines
    f := func(goal int) {
        donation.mu.RLock()
        for donation.balance < goal {
            donation.mu.RUnlock()
            donation.mu.RLock()
        }
        fmt.Printf("$%d goal reached\n", donation.balance)
        donation.mu.RUnlock()
    }
    go f(10)
    go f(15)

    // Updater goroutine
    go func() {
        for {
            time.Sleep(time.Second)
            donation.mu.Lock()
            donation.balance++
            donation.mu.Unlock()
        }
    }()
```

Create and instantiate a Donation struct containing the current balance and a mutex 1

Create a closure 2

Check if the goal is reached 3

Keep incrementing the balance 4

We protect the accesses to the shared `donation.balance` variable using the mutex. If we run this example, it would work as expected:

```
$10 goal reached
    $15 goal reached
```

The main issue and what makes this a terrible implementation is the busy loop. Indeed, each listener goroutine keeps looping until their donation goal is met, which will waste a lot of CPU cycles and make the CPU usage gigantic. We need to find a better solution.

Let's take a step back on what we need. We have to find a way to signal from the updater goroutine whenever the balance is updated. If we think about signaling in Go, we should think about channels. So let's try another version using the channel primitive:

```
type Donation struct {
            balance int
            ch      chan int
    }

    donation := &Donation{ch: make(chan int)}

    // Listener goroutines
    f := func(goal int) {
            for balance := range donation.ch {
                    if balance >= goal {
                            fmt.Printf("$%d goal reached\n",
balance)
                            return
                    }
            }
    }
    go f(10)
    go f(15)

    // Updater goroutine
    for {
            time.Sleep(time.Second)
            donation.balance++
            donation.ch <- donation.balance
    }
```

Update Donation so that it contains a channel 1
Receive channel updates 2
Send a message whenever the balance is updated 3

Each listener goroutine receives to a shared channel. Meanwhile, the updater goroutine sends messages whenever the balance is updated. However, if we give this solution a try, here would be a possible output:

```
$11 goal reached
    $15 goal reached
```

The first goroutine should have been notified when the balance is $10, not $11? So, what happened?

The reason is that a message sent to a channel is received by only one goroutine. In our example, if the first goroutine receives from the channel before the second one, here's what could happen:

Figure 9.12. The first goroutine receives the $1 message, then the second goroutine receives the $2 message, then the second goroutine receives the $3 message, and so forth.

The default distribution mode with multiple goroutines receiving from a shared channel is round-robin. It can change if one goroutine isn't ready yet to receive messages (not in a waiting state on the channel); in that case, Go would distribute the message to the next available goroutine.

In any case, each message is received by a single goroutine. Therefore, the first goroutine hasn't received the $10 message in this example, but the second one did. Only a channel closure is an event that can be broadcasted to multiple goroutines. Yet, here we don't want to close the channel as the updater goroutine couldn't send messages anymore.

Also, there's another issue using channels in this situation. The listener goroutines return whenever their donation goal is met. Hence, the updater goroutine has to know when all the listeners stop receiving messages to the channel. Otherwise, it will eventually become full and block the sender. A possible solution could be to add in the mix a `sync.WaitGroup` as well, but it would make the solution more complex.

Ideally, we need to find a way to repeatedly broadcast notifications whenever the balance is updated to multiple goroutines. Fortunately, Go has a solution for that: `sync.Cond`. Let's first discuss theory, and then we will see how to solve our problem using this primitive.

According to the official documentation[22]:

> Cond implements a condition variable, a rendezvous point for goroutines waiting for or announcing the occurrence of an event.
>
> -- sync package documentation

A condition variable is a container of threads (here, goroutines) waiting for a certain condition. In our example, the condition is a balance update. The updater goroutine will broadcast a notification whenever a balance is updated, whereas the listener goroutine will wait until an update. Furthermore,

`sync.Cond` relies on a `sync.Locker` (a `*sync.Mutex` or `*sync.RWMutex`) to prevent data races. Here will be a possible implementation:

```go
type Donation struct {
        cond    *sync.Cond
        balance int
    }

    donation := &Donation{
        cond: sync.NewCond(&sync.Mutex{}),
    }

    // Listener goroutines
    f := func(goal int) {
        donation.cond.L.Lock()
        for donation.balance < goal {
            donation.cond.Wait()
        }
        fmt.Printf("%d$ goal reached\n", donation.balance)
        donation.cond.L.Unlock()
    }
    go f(10)
    go f(15)

    // Updater goroutine
    for {
        time.Sleep(time.Second)
        donation.cond.L.Lock()
        donation.balance++
        donation.cond.L.Unlock()
        donation.cond.Broadcast()
    }
```

Add a *sync.Cond 1

The sync.Cond relies on a mutex 2

Wait for a condition (balance updated) within lock/unlock 3

Increment balance within lock/unlock 4

Broadcast the fact that a condition was met (balance updated) 5

First, we create a `*sync.Cond` using `sync.NewCond` and provide a `*sync.Mutex`. Then, what about the listener goroutines and the updater

goroutine?

The listener goroutines loop until the donation balance is met. Within the loop, we use the `Wait` method that will be blocking until the condition is met.

> **Note**
>
> Let's make sure the term condition is understood here. In this context, we're not talking about the donation goal condition but the balance being updated. So, it's a single condition variable shared by two listener goroutines.

The call to `Wait` has to be done within a critical section which may sound odd. Isn't the lock going to prevent other goroutines from also waiting for the same condition? Actually, the implementation of `Wait` is the following:

- Unlock the mutex
- Suspend the goroutine and wait for a notification
- Lock the mutex when the notification arrives

Hence, in the listener goroutines we will have two critical sections:

- When accessing to `donation.balance` in `for donation.balance < goal`
- When accessing to `donation.balance` in `fmt.Printf`

This way, all the accesses to the shared `donation.balance` variable are protected.

Now, what about the updater goroutine? The balance update is done within a critical section to prevent data races. Then, we call the `Broadcast` method,

which wakes all the goroutines waiting on the condition each time the balance is updated.

Hence, if we run this example, it will print what we expect:

```
10$ goal reached
    15$ goal reached
```

In our implementation, the condition variable is based on the balance being updated. Therefore, the listener variables wake each time a new donation is made to check whether their donation goal is met. This solution prevents us from having a busy loop that would burn CPU cycles in repeated checks.

Let's also note one possible drawback when using `sync.Cond`. When we send a notification, for example, to a `chan struct`, even if there's no active receiver, the message will be buffered, which guarantees that this notification will be received eventually. Using `sync.Cond` with the `Broadcast` method wakes all goroutines currently waiting on the condition, and if none, the notification will be missed. This is also an essential principle that we have to keep in mind.

Note

Here, this code doesn't block as there's no goroutine waiting to receive messages in this channel. This is the same principle for `Signal()`.

We should also note a way to wake only one goroutine using `Signal()` instead of `Broadcast`. In terms of semantics, it would

be the same as sending in a non-blocking fashion a message in a `chan struct`:

```
ch := make(chan struct{})
    select {
    case ch <- struct{}{}:
    default:
    }
```

Signaling in Go can be achieved with channels. However, we should be aware that only a channel closure is an event multiple goroutines can catch. Yet, this can happen only once. Therefore, if we repeatedly send notifications to multiple goroutines, `sync.Cond` is a solution. This primitive is based on condition variables that set up containers of threads waiting for a specific condition. Using `sync.Cond`, we can broadcast signals that will wake all the goroutines waiting on a condition.

Let's now extend our knowledge of concurrency primitives using `golang.org/x` and the `errgroup` package.

9.13 Not using `errgroup`

Regardless of the programming language, reinventing the wheel is rarely a good idea. Likewise, it's pretty frequent to find codebases reimplementing how to spin up multiple goroutines and aggregate the errors. Yet, a package in the Go ecosystem is designed to support this frequent use case. Let's delve into it and understand why it should be part of the toolset of Go developers.

`golang.org/x` is a repository providing extensions to the standard library. Among the different sub-repositories, there's a `sync` one that contains a handy package: `errgroup`.

Suppose we have to handle the following function. We receive as an argument some data that we want to use to call an external service. Yet, we can't make a single call but multiple calls with a different subset each time because of some constraints. Also, these calls will be made in parallel:

Figure 9.13. Each circle results in a parallel call.

In case of one error during a call, we want to return it. In case of multiple errors, we just want to return one of them. Let's try to write the skeleton of the implementation using the standard concurrency primitives solely:

```
1  func handler(ctx context.Context, circles []Circle) ([]Result,
2  error) {
3              results := make([]Result, len(circles))
4              wg := sync.WaitGroup{}
5              wg.Add(len(results))
6
7              for i, circle := range circles {
8                      i := i
9                      circle := circle
10                     go func() {
11                             defer wg.Done()
12
13                             result, err := foo(ctx, circle)
14                             if err != nil {
```

```
                                        // ?
                                }
                                results[i] = result
                        }()
                }

                wg.Wait()
                // ...
        }
```

Create a wait group to wait for all the goroutines that we spin up 1
Create a new i variable used in the goroutine (see Not being careful with goroutines and loop variables) 2
Same for circle 3
Trigger a goroutine per Circle 4
Indicate when the goroutine is complete 5
Aggregate the results 6

Here, we decided to use a `sync.WaitGroup` to wait until all the goroutines are completed and handle the aggregations in a slice. It's one way to do it; another could have been to send each partial result to a channel and aggregate them within another goroutine. The main challenge would have been to reorder the incoming messages if ordering is required. Therefore, we decided to go with the easiest approach and a shared slice.

> **Note**
>
> As each goroutine writes to a specific index, this implementation is data race-free.

However, there's one crucial case that we haven't tackled yet. What if `foo` (the call made within a new goroutine) returns an error? How should we handle it? There are different options, for example:

- Just like the `results` slice, we could have a slice of errors shared among the goroutines. Each one would write to this slice in case of an error. We would have to iterate over this slice in the parent goroutine to determine whether an error has occurred (O(n) time complexity).
- Another approach could be to have a single error variable accessed via a shared mutex by the goroutines.
- We could also think about sharing a channel of errors, and the parent goroutine would receive and handle these errors.

Regardless of the option chosen, it starts to make the solution pretty complex. For that reason, the `errgroup` package was designed and developed.

It exports a single `WithContext` function that returns a `*Group` struct given a context. This struct provides synchronization, error propagation, and context cancellation for a group of goroutines. It exports only two methods:

- `Go` to trigger a call in a new goroutine.
- `Wait` to block until all the goroutines have completed. It returns the first non-nil error if any.

Let's rewrite the solution using `errgroup`. First, we need to import the `errgroup` package:

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

And here's the implementation:

```go
func handler(ctx context.Context, circles []Circle) ([]Result, error) {
        results := make([]Result, len(circles))
        g, ctx := errgroup.WithContext(ctx)

        for i, circle := range circles {
                i := i
                circle := circle
                g.Go(func() error {
                        result, err := foo(ctx, circle)
                        if err != nil {
                                return err
                        }
                        results[i] = result
                        return nil
                })
        }

        if err := g.Wait(); err != nil {
                return nil, err
        }
        return results, nil
}
```

Create an *errgroup.Group given the parent context 1
Call Go to spin up the logic of handling the error and aggregating the results in a new goroutine 2
Call Wait to wait for all the goroutines 3

First, we create an `*errgroup.Group` by providing the parent context. In each iteration, we use `g.Go` to trigger a call in a new goroutine. This method takes a `func() error` as an input, a closure wrapping the call to `foo` and handling the result and error. As the main difference, if we get an error, we return it from this closure. Then, `g.Wait` allows us to wait for all the goroutines to complete.

This solution is inherently more straightforward than the first one (which was only partial as we didn't handle error yet). We don't have to rely on extra concurrency primitives, and the `errgroup.Group` was sufficient to tackle our use case.

Another benefit that we didn't tackle yet is the shared context. Let's imagine we have to trigger three parallel calls:

- The first one returns an error in one millisecond
- The second and third calls return a result or an error in five seconds

In our case, we want to return an error, if any. Hence, there's no point in waiting until the second and third calls are complete. Using `errgroup.WithContext` creates a shared context used in all the parallel calls. As the first call returns an error in one millisecond, it will cancel the context and thus the other goroutines. Hence, we won't have to wait five seconds to return an error. This is another benefit while using `errgroup`.

> **Note**
>
> The process invoked by `g.Go` has to be *context aware*; otherwise, canceling the context won't have any effect.

In summary, when we have to trigger multiple goroutines and handle errors plus context propagation, it may be worth considering if `errgroup` could be a solution for us. As we have seen, this package enables synchronization for a group of goroutines and provides an answer to deal with errors and shared contexts.

The last section of this chapter will discuss a common mistake made by Go developers when copying a sync type.

9.14 Copying a `sync` type

The `sync` package provides basic synchronization primitives such as mutexes, condition variables, and wait groups. For all these types, there's a hard rule to follow: they should never be copied. Let's understand the rationale and the possible problems with it.

We will create a thread-safe data structure to store counters. It will contain a `map[string]int` representing the current value for each counter. Also, we will use a `sync.Mutex` as the accesses have to be protected. Let's also add an `increment` method to increment a given counter name:

```
1  type Counter struct {
2              mu       sync.Mutex
3              counters map[string]int
4      }
5
6      func NewCounter() Counter {
7              return Counter{counters: map[string]int{}}
8      }
9
10     func (c Counter) Increment(name string) {
11             c.mu.Lock()
12             defer c.mu.Unlock()
13             c.counters[name]++
14     }
```

Factory function 1

Increment counter in a critical section 2

The increment logic is done within a critical section: between `c.mu.Lock()` and `c.mu.Unlock()`. Let's give our method a try by running using the `-race` option the following example that spins up two goroutines and increments their respective counter:

```
1 counter := NewCounter()
2
3     go func() {
```

```
            counter.Increment("foo")
    }()
    go func() {
            counter.Increment("bar")
    }()
```

If we run this example, it will raise a data race:

```
==================
    WARNING: DATA RACE
    ...
```

The problem in our `Counter` implementation is because the mutex is copied. Indeed, because the receiver of `Increment` is a value, whenever we call `Increment` it performs a copy of the `Counter` struct, which also copies the mutex.

`sync` types shouldn't be copied. This rule applies to the following types:

- `sync.Cond`
- `sync.Map`
- `sync.Mutex`
- `sync.RWMutex`
- `sync.Once`
- `sync.Pool`
- `sync.WaitGroup`

Therefore, the mutex shouldn't have been copied. What are the options, then?

The first one is to modify the receiver type for the `Increment` method:

```
1 func (c *Counter) Increment(name string) {
2             // Same code
3     }
```

Changing the receiver type avoids copying `Counter` when `Increment` is called. Therefore, the internal mutex isn't copied.

If we want to keep a value receiver, the second option is to change the type of the `mu` field in `Counter` to be a pointer:

```
1  type Counter struct {
2              mu       *sync.Mutex
3              counters map[string]int
4      }
5
6      func NewCounter() Counter {
7              return Counter{
8                      mu: &sync.Mutex{},
9                      counters: map[string]int{},
10             }
11     }
```

Change the type of mu 1
Change the way mu is initialized 2

If `Increment` has a value receiver, it will still copy the `Counter` struct. However, as `mu` is now a pointer, it will perform a pointer copy solely, not an actual copy of a `sync.Mutex`. Hence, this solution will also prevent data races.

Note

We also changed the way `mu` was initialized. Indeed, as `mu` is a pointer, if we omit it when creating `Counter` it will be initialized to the zero value of a pointer: nil. Hence, leading to the goroutine to panic when `c.mu.Lock()` is called.

We can face the issue of unintentionally copying a `sync` field in some of the following conditions:

- Calling a method with a value receiver (as we have seen)
- Calling a function with a `sync` argument
- Calling a function with an argument that contains a `sync` field

In each case, we should remain very cautious. Also, let's note that some linters can catch this issue. For example, using `go vet`:

```
$ go vet .
    ./main.go:19:9: Increment passes lock by value: Counter contains sync.Mutex
```

As a rule of thumb, whenever multiple goroutines have to access a common `sync` element, we must ensure they all rely on the same instance. This rule applies to all the types defined in the `sync` package. Using pointers is a way to solve this problem: either by having a pointer to a `sync` element or a pointer to a struct containing a `sync` element.

9.15 Chapter summary

- Understanding the conditions when a context can be canceled should matter when it comes to propagating it. For example, an HTTP handler canceling the context when the response has been sent.
- Avoiding leaks also means being mindful that whenever a goroutine is started, we should have a plan to stop it eventually.
- To avoid bugs with goroutines and loop variables, we should create local variables or call functions instead of closures.
- Understanding that `select` with multiple channels choose the case randomly if multiple options are possible prevents us from making wrong assumptions leading to subtle concurrency bugs.
- Sending notifications should be done using a `chan struct{}` type.
- Using nil channels should be part of the concurrency toolset of Go developers as it allows *removing* cases from `select` statements, for example.
- Using the right channel type given a problem should be a careful decision. Only unbuffered channels provide strong synchronization guarantees.
- Specifying a channel size different from one for buffered channels should be done for a good reason.
- Being aware that string formatting leads to calling existing functions means watching out for possible deadlocks and other data races.
- Calling `append` isn't always data race-free; hence, it shouldn't be used concurrently on a shared slice.

- Reminding us that slices and maps are pointers can prevent common data races.
- To accurately use `sync.WaitGroup`, we should call the `Add` method before spinning up goroutines.
- Sending repeated notifications to multiple goroutines can be done with `sync.Cond`.
- Synchronizing a group of goroutines and handling errors and contexts can be done with the `errgroup` package.
- `sync` types shouldn't be copied.

[20] lmax-exchange.github.io/disruptor/files/Disruptor-1.0.pdf

[21] github.com/etcd-io/etcd/pull/7816

[22] pkg.go.dev/sync

10 Standard library

This chapter covers

- Preventing from providing a wrong time duration
- Discussing potential memory leaks while using `time.After`
- Delving into the most common mistakes of JSON handling and SQL
- Remembering why transient resources have to be closed
- Avoiding unexpected behavior while forgetting the return statement in HTTP handlers
- Presenting why production-grade applications shouldn’t use default HTTP clients and servers

The Go standard library is a set of core packages that enhance and extend the language. For example, Go developers can write HTTP clients or servers, handle JSON data, or interact with SQL databases. All of these features are provided by the standard library. However, it can be easy to misuse it on many occasions or have a limited understanding of its behavior. Therefore, it can lead to both possible bugs and writing applications that shouldn’t be considered production-grade. Let’s delve into some of the most common mistakes while using this standard library.

10.1 Providing a wrong time duration

The standard library provides common functions or methods accepting a `time.Duration`. However, as `time.Duration` is an alias for `int64` type, newcomers to the language can get confused and provide a wrong duration. For example, developers with a Java or JavaScript background are used to passing numeric types.

To illustrate this common error, let's create a new `time.Ticker` that will deliver ticks of a clock at a regular interval, every second:

```
ticker := time.NewTicker(1000)
for {
        select {
        case <-ticker.C:
                // Do something
        }
}
```

If we run this code, we will notice that ticks aren't delivered every second; they are delivered every microsecond.

Indeed, as `time.Duration` is based on the `int64` type, the previous code was correct as `1000` is a valid `int64`. Yet, `time.Duration` represents the elapsed time between two instants in *nanoseconds*. Therefore, we provided to `NewTicker` a duration of 1000 nanoseconds = 1 microsecond.

This mistake isn't infrequent. Indeed, standard libraries in languages such as Java or JavaScript sometimes ask developers to provide durations in milliseconds.

Furthermore, if we wanted to consciously create a `time.Ticker` with an interval of one microsecond, we shouldn't pass an `int64` directly as we have

done. We should make sure always to use the `time.Duration` API to avoid possible confusion:

```
ticker = time.NewTicker(time.Microsecond)
// Or
ticker = time.NewTicker(1000 * time.Nanosecond)
```

This common mistake is also probably not the most complex of the book. However, developers with an existing background in other languages can easily fall into the trap of believing that milliseconds are expected for the functions and methods of the `time` package. Remember always to use the `time.Duration` API and provide an `int64` alongside a time unit.

Now, let's discuss a common mistake while using the `time` package with `time.After`.

10.2 `time.After` and memory leak

`time.After(time.Duration)` is a convenient function that waits for a provided duration to elapse and then sends a message to the channel returned. Usually, it's used in concurrent code; otherwise, if we want to sleep for a given duration, we can use `time.Sleep(time.Duration)`. The advantage of `time.After` is that it can be used to implement scenarios such as "if I don't receive any message in this channel for 5 seconds, then I will …". Yet, it's pretty frequent to find in codebases calls to `time.After` in a loop which, as we will describe in this section, may be a root cause for memory leaks.

Let's consider the following example. We will implement a function that repeatedly consumes messages from a channel. In the meantime, we want to

log a warning in case we haven't received any messages for more than one hour. Here is a possible implementation:

```
1  func consumer(ch <-chan Event) {
2          for {
3                  select {
4                  case event := <-ch:
5                          handle(event)
6                  case <-time.After(time.Hour):
7                          log.Println("warning: no messages
8  received")
9                  }
10         }
   }
```

Handle the event 1
Increment the idle counter 2

Here, we use `select` on two cases: receiving a message from `ch` or after one hour without messages (as `time.After` is evaluated during each iteration, the timeout is *reset* every time). At first sight, this code looks ok. However, it may lead to memory usage issues.

As we said, `time.After` returns a channel. We may expect this channel to be closed during each loop iteration; yet, this isn't the case. The resources created by `time.After` (including the channel) will be released once the timeout expires and will use some memory until it happens. How much memory? In Go 1.15, about 200 bytes of memory per call to `time.After`. If we receive an important volume of messages, such as 5 million per hour, our application will consume 1GB of memory to store the `time.After` resources.

Can we fix this issue by closing the channel programmatically during each iteration? It's not possible. The returned channel is a `<-chan time.Time` meaning a receive-only channel that can't be closed.

If we want to fix our example, there are different options.

The first option could be to use a context instead of `time.After`:

```
1  func consumer(ch <-chan Event) {
2          for {
3                  ctx, cancel :=
4  context.WithTimeout(context.Background(), time.Hour)
5                  select {
6                  case event := <-ch:
7                          cancel()
8                          handle(event)
9                  case <-ctx.Done():
10                         log.Println("warning: no messages
11 received")
12                 }
           }
   }
```

Main loop 1
Create a context with timeout 2
Cancel context if we receive a message 3
Context cancellation 4

This approach's downside is that we have to keep recreating a context during every single loop iteration. Creating a context isn't the most lightweight operation in Go as it requires creating a channel, for example. Can we do better?

The second option comes again from the `time` package: `time.NewTimer`. This function creates a `time.Timer` struct that exports:

- A `C` field, which is the internal timer channel
- A `Reset(time.Duration)` method to reset the duration

- A `Stop()` method to stop the timer

Note

We should note that `time.After` also relies on `time.Timer`. However, it only returns the `C` field. Hence, we don't have access to the `Reset` method:

```
1 package time
2
3 func After(d Duration) <-chan Time {
4         return NewTimer(d).C
5 }
```

Create a new time.Timer and return the channel field 1

Let's implement a new version using `time.NewTimer`:

```
1  func consumer(ch <-chan Event) {
2          timerDuration := 1 * time.Hour
3          timer := time.NewTimer(timerDuration)
4
5          for {
6                  timer.Reset(timerDuration)
7                  select {
8                  case event := <-ch:
9                          handle(event)
10                 case <-timer.C:
11                         log.Println("warning: no messages
12 received")
13                 }
14         }
   }
```

Creates a new timer 1

Main loop 2

```
Reset the duration 3
Timer expiration 4
```

In this implementation, we keep a recurring action during each loop iteration: calling the `Reset` method. However, calling `Reset` is less cumbersome than having to create a new context every time. It's faster and puts less pressure on the GC because it doesn't require any new heap allocation. Therefore, using `time.Timer` is the best possible solution for our initial problem.

> **Note**
>
> For the sake of simplicity, in the previous example, the previous goroutine doesn't stop. As we mentioned in *Starting a goroutine without knowing when to stop it*, this isn't a best practice. In production-grade code, we should find an exit condition such as a context that would cancel. In that case, we should also remember stopping the `time.Timer` created using `defer timer.Stop()` for example right after `timer` creation.

Using `time.After` in a loop isn't the only case that may lead to a possible peak of memory consumption. The problem relates to code that is repeatedly called. Loop is one case, but using `time.After` in an HTTP handler function can also lead to the same issues as this function will be called multiple times.

In general, we should be cautious when using `time.After`. Remember that the resources created will only be released when the timer expires. When the call to `time.After` is repeated (for example, in a loop, a Kafka consumer function, an HTTP handler), it may lead to a peak of memory consumption. In this case, we should favor `time.NewTimer`.

The following section will delve into the most common mistakes of JSON handling.

10.3 JSON handling common mistakes

Go has excellent support for JSON with the `encoding/json` package. This section will cover three common mistakes related to encoding (marshaling) and decoding (unmarshaling) JSON data.

10.3.1 Unexpected behavior because of type embedding

In *Not being aware of the possible problems with type embedding*, we explained the possible issues related to type embedding. In the context of JSON handling, let's discuss another potential impact that can lead to unexpected marshaling/unmarshaling results.

In the following example, we will create an `Event` struct containing an ID and an embedded timestamp:

```
type Event struct {
        ID int
        time.Time
}
```

Embedded field 1

Here, as `time.Time` is embedded, in the same way that we described previously, we can access the `time.Time` methods directly at the `Event` level; for example, `event.Second()`.

What are the possible impacts of embedded fields with JSON marshaling? Let's find it out in the following example. We will instantiate an `Event` and marshal it into JSON. What should be the output of this code?

```
event := Event{
        ID:   1234,
        Time: time.Now(),
}

b, err := json.Marshal(event)
if err != nil {
        return err
}

fmt.Println(string(b))
```

The field name of an anonymous field during a struct instantiation is the name of the struct (Time)

1

We may expect this code to print the following:

```
{"ID":1234,"Time":"2021-04-19T21:15:08.381652+02:00"}
```

Instead, it will print this:

```
"2020-12-21T00:08:22.81013+01:00"
```

How can we explain this output? What happened to the `ID` field and the 1234 value? As this field is exported, it should have been marshaled. To understand this problem, we have to highlight two points.

Firstly, as discussed in *Not being aware of the possible problems with type embedding*, if an embedded field type implements an interface, the struct containing the embedded field will also implement this interface.

Secondly, we can change the default marshaling behavior by making a type implement `json.Marshaler` interface. This interface contains a single `MarshalJSON` function:

```
1 type Marshaler interface {
2         MarshalJSON() ([]byte, error)
3 }
```

Here is an example with a custom marshaling:

```
1  type foo struct{}
2
3  func (foo) MarshalJSON() ([]byte, error) {
4          return []byte(`"foo"`), nil
5  }
6
7  func main() {
8          b, err := json.Marshal(foo{})
9          if err != nil {
10                 panic(err)
11         }
12         fmt.Println(string(b))
13 }
```

Define the struct 1
Implement the MarshalJSON method 2
Return a static response 3
json.Marshal will then rely on the custom MarshalJSON implementation 4

As we have changed the default JSON marshaling behavior by implementing the `Marshaler` interface, this code will print `"foo"`.

Having these two points clarified, let's get back to the initial problem with the `Event` struct:

```
type Event struct {
        ID int
        time.Time
}
```

1

We have to know that `time.Time` **implements** the `json.Marshaler` interface. As `time.Time` is an embedded field of `Event`, it promotes its methods. Therefore, `Event` also implement `json.Marshaler`.

Consequently, passing an `Event` to `json.Marshal` will not use the default marshaling behavior but the one provided by `time.Time`. It's the reason why marshaling an `Event` leads to ignoring the `ID` field.

> **Note**
>
> We would also face the issue the other way around if we were unmarshaling an `Event` using `json.Unmarshal`.

To fix this issue, there are two main possibilities.

First, we can make the `time.Time` field not embedded anymore by adding a name:

```
type Event struct {
        ID   int
        Time time.Time
}
```

time.Time isn't an embedded type anymore 1

This way, if we marshal a version of this `Event` struct, it will print something like:

```
{"ID":1234,"Time":"2020-12-21T00:30:41.413417+01:00"}
```

If we want to keep or have to keep the `time.Time` field embedded, the other option is to make `Event` implementing the `json.Marshaler` interface:

```
 func (e Event) MarshalJSON() ([]byte, error) {
        return json.Marshal(
                struct {
                        ID   int
                        Time time.Time
                }{
                        ID:   e.ID,
                        Time: e.Time,
                },
        )
}
```

Create an anonymous struct 1

In this solution, we implemented a custom `MarshalJSON` method in defining an anonymous struct reflecting the structure of `Event`. Yet, this solution is more cumbersome and requires ensuring that the `MarshalJSON` method is always up to date with the `Event` struct.

We should be careful with embedded fields. While promoting the fields and methods of an embedded field type can sometimes be convenient, it can also lead to subtle bugs as it can make the parent struct implement some interfaces without a clear signal. Again, when using embedded fields, we should clearly understand the possible side effects.

In the next section, we will see another common JSON mistake related to using `time.Time`.

10.3.2 JSON and monotonic clock

When marshaling or unmarshaling a struct that contains a `time.Time` type, we can sometimes face unexpected comparisons errors. It's helpful to delve into `time.Time` to refine our assumptions and prevent possible mistakes.

An OS handles two different clock types: wall and monotonic. This section will first delve into these two clock types, then see a possible impact while working with JSON and `time.Time`.

The wall clock is used to know the current time of the day. This clock is subject to potential variations. For example, if synchronized using NTP (Network Time Protocol), the clock can jump backward or forward in time. We shouldn't measure durations using the wall clock as we may face strange behavior such as negative durations. It's the reason why operating systems provide a second clock type: monotonic clocks. The monotonic clock guarantees that the time will always move forward and will not be impacted by jumps in time. It can be affected by potential frequency adjustments (if the server, for example, detects that the local quartz is moving at a different pace than the NTP server) but never by jumps in time.

In the following example, we will keep considering an `Event` struct but containing a single `time.Time` field (not embedded):

```
type Event struct {
        Time time.Time
}
```

We will instantiate an `Event`, marshal it into JSON and unmarshal it into another struct. Then, we will compare both structs. Let's find out if the marshaling/unmarshaling process is always symmetric:

```
t := time.Now()
event1 := Event{
        Time: t,
}

b, err := json.Marshal(event1)
if err != nil {
        return err
}

var event2 Event
err = json.Unmarshal(b, &event2)
if err != nil {
        return err
}

fmt.Println(event1 == event2)
```

Get the current local time 1
Instantiate an Event struct 2
Marshal into JSON 3
Unmarshal JSON 4

What should be the output of this code? It will print false, not true. How can we explain it?

First, let's print the content of `event1` and `event2`:

```
fmt.Println(event1.Time)
fmt.Println(event2.Time)
```

```
2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660
2021-01-10 17:13:08.852061 +0100 CET
```

So we can notice that it prints two different contents. `event1` is similar to `event2` except for the `m=+0.000338660` part. What is the meaning of it?

In Go, instead of splitting the two clocks into two different APIs, the `time.Time` may contain both a wall and monotonic time. When we get the local time using `time.Now()`, it returns a `time.Time` with both times:

```
2021-01-10 17:13:08.852061 +0100 CET m=+0.000338660
------------------------------------- --------------
             Wall time                Monotonic time
```

Conversely, when we unmarshal the JSON, the `time.Time` field doesn't contain the monotonic time, only the wall time. Therefore, when we compared both structs, the result was false because of a monotonic time difference, and it's also the reason why we noticed a difference while printing both structs.

How can we fix this problem? There are two main options.

When we used `==` operator to compare both `time.Time`, it compared all the struct fields, including the monotonic part. To avoid this, we can use the `Equal` method instead:

```
fmt.Println(event1.Time.Equal(event2.Time))
```

```
true
```

The `Equal` method doesn't consider monotonic time; therefore, this code prints true. However, in this case, we only compared the `time.Time` fields, not the parent `Event` structs.

The second option is to keep the `==` to compare the two structs but to strip away the monotonic time using the `Truncate` method. This method returns the result of rounding the `time.Time` value down to a multiple of a given duration. We can use it by providing a zero duration like so:

```
t := time.Now()
event1 := Event{
        Time: t.Truncate(0),
}

b, err := json.Marshal(event1)
if err != nil {
        return err
}

var event2 Event
err = json.Unmarshal(b, &event2)
if err != nil {
        return err
}
```

```
16
17 fmt.Println(event1 == event2)
```

Strip away the monotonic time 1
Perform the comparison using == operator 2

With this version, the two `time.Time` fields are now equal. Therefore, this code will now print true.

Note

Let's also note that each `time.Time` is associated with a `time.Location` that represents the time zone, for example:

```
t := time.Now() // 2021-01-10 17:13:08.852061
+0100 CET
```

Here, the location is set to CET as I used `time.Now()` that returns my current local time. We should note that the JSON marshaling result depends on the location. If we want to prevent it, we should either stick to a particular location, for example:

```
1 location, err :=
2 time.LoadLocation("America/New_York")
3 if err != nil {
4         return err
5 }
  t := time.Now().In(location) // 2021-05-18
  22:47:04.155755 -0500 EST
```

Get the current location for America/New_York 1

Or get the current time in UTC:

```
t := time.Now().UTC() // 2021-05-18
22:47:04.155755 +0000 UTC
```

In summary, the marshaling/unmarshaling process isn't always symmetric, and we faced this case with a struct containing a `time.Time`. We should keep this principle in mind so that we don't write erroneous tests, for example.

10.3.3 Map of `any`

When unmarshaling data, we can also provide a map instead of a struct. The rationale is when the keys and values are uncertain, passing a map will give us some flexibility instead of a static struct. However, there's a particular rule to bear in mind to avoid wrong assumptions and possibly goroutines panic.

Let's write an example that unmarshals a message into a map:

```
b := getMessage()
var m map[string]any
err := json.Unmarshal(b, &m)
if err != nil {
        return err
}
```

Provide a map pointer 1

If we provide the following JSON to the previous code:

```
{
        "id": 32,

```

```
4         "name": "foo"
  }
```

As we used a generic `map[string]any`, it would parse all the different fields automatically:

```
map[id:32 name:foo]
```

However, there's one important gotcha to remember if we use a map of `any`: any numeric value, regardless if it contains a decimal, is converted into a `float64` type. We can observe it by printing the type of `m["id"]`:

```
fmt.Printf("%T\n", m["id"])
```

```
float64
```

We should remember it to make sure we don't make the wrong assumption and expect numeric values without decimals to be converted into integers by default. Making incorrect assumptions with type conversions could lead, for example, to goroutine panics.

The following section will delve into the most common mistakes while writing applications that interact with SQL databases.

10.4 SQL common mistakes

The `database/sql` provides a generic interface around SQL (or SQL-like) databases. It's also fairly common to find common patterns or mistakes while using this package. Let's delve into five common mistakes.

10.4.1 Forgetting that `sql.Open` doesn't necessarily establish connections to a DB

When using `sql.Open`, one common misconception is to expect this function actually to establish connections to a DB:

```
db, err := sql.Open("mysql", dsn)
if err != nil {
        return err
}
```

Yet, this isn't necessarily the case. According to the documentation:

> Open may just validate its arguments without creating a connection to the database.
>
> -- sql.Open documentation

Actually, the behavior depends on the SQL driver used. For some drivers, `sql.Open` doesn't establish any connection; instead, it's only a preparation for later use (e.g., with `db.Query`). Therefore, the first connection to the DB may be established lazily.

Why does it matter to know about this behavior? For example, in some cases, we want to make a service ready only after knowing that all the dependencies

are correctly set up and reachable. If we don't, the service can potentially accept some traffic despite an erroneous configuration.

Therefore, if we want to ensure that the function that uses `sql.Open` also guarantees that the underlying DB is reachable, we should use the `Ping` method:

```
db, err := sql.Open("mysql", dsn)
if err != nil {
        return err
}
if err := db.Ping(); err != nil {
        return err
}
```

Call the Ping method following sql.Open 1

`Ping` forces establish a connection which ensures that the data source name is valid and the DB is reachable. Note that an alternative to ping is `PingContext` that asks for an additional context conveying when the ping should be canceled or times out.

Despite being perhaps counter-intuitive, let's remember that `sql.Open` doesn't necessarily establish any connection, and the first connection can be opened lazily. If we want to test our configuration and that a DB is reachable, we should follow `sql.Open` with a call to the `Ping` or `PingContext` method.

10.4.2 Forgetting about connections pooling

In the same way that the default HTTP client and server don't provide default behaviors that may not be effective in production, it's essential to understand how DB connections are handled in Go.

`sql.Open` returns an `*sql.DB` struct. This struct doesn't represent a single database connection; instead, it represents a pool of connections. It's worth noting so that we're not tempted to implement it manually. A connection in the pool can have two states:

- Already used (for example, by another goroutine that triggers a query)
- Idle (already created but not in use for the time being)

Furthermore, it's important to remember that creating a pool also means four available config parameters. Each of these parameters is an exported method of `*sql.DB`:

- `SetMaxOpenConns`: maximum number of open connections to the DB (default value: unlimited)
- `SetMaxIdleConns`: maximum number of idle connections (default value: 2)
- `SetConnMaxIdleTime`: maximum amount of time a connection can be idle before it's closed (default value: unlimited)
- `SetConnMaxLifetime`: maximum amount of time a connection can be held open before it's closed (default value: unlimited)

Here's an example with a maximum number of five connections:

Figure 10.1. A connection pool example with five connections.

This example has four ongoing connections: three idles and one already in use. Therefore, one available slot remains for an extra connection. If a new query comes in, it will pick one of the idle connections (if still available). If there are no more idle connections, the pool will create a new connection if there's an extra slot available; otherwise, it will wait until a connection is available.

So, why should we tweak these config parameters?

- Setting `SetMaxOpenConns` is important for production-grade applications. Indeed, as the default value is unlimited, we should rather set it to make sure it fits what the underlying DB can handle.
- Setting `SetMaxIdleConns` should be increased (default value of two) if our application generates a significant number of concurrent requests. Otherwise, it may lead to frequent reconnects.
- Setting `SetConnMaxIdleTime` is important if our application can face a burst of requests. Indeed, when the application is back to a more peaceful state, we want to make sure the connections created are eventually released.
- Last but not least, setting `SetConnMaxLifetime` can also be helpful if we connect to a load-balanced DB server, for

example. In that case, we want to ensure that our application never uses a connection for too long.

For production-grade applications, let's remember to consider these four parameters. Also, let's note that we can use multiple connection pools if one application faces different use cases.

10.4.3 Not using prepared statements

A prepared statement is a feature implemented by many SQL databases to execute a repeated SQL statement. Internally, the SQL statement is pre-compiled and separated from the data provided. There are two main benefits:

- Efficiency: the statement doesn't have to be re-compiled (compilation means parsing + optimization + translation)
- Security: it reduces the risks of SQL injection attacks

Therefore, if a statement is repeated, we should use prepared statements. Meanwhile, we should also use prepared statements in untrusted contexts (e.g., exposing an endpoint on the Internet where the request is mapped to an SQL statement).

To use prepared statements, instead of calling the `Query` method of `*sql.DB`, we have to call `Prepare`:

```
stmt, err := db.Prepare("SELECT * FROM ORDER WHERE ID = ?")
if err != nil {
        return err
}
rows, err := stmt.Query(id)
// ...
```

Prepare the statement 1

Execute the prepared query 2

First, we prepare the statement and then execute it while providing the arguments. The first output of the `Prepare` method is an `*sql.Stmt`, which can be reused and run concurrently. Let's also be aware that when the statement is no longer needed, it must be closed using the `Close()` method.

> **Note**
>
> The `Prepare` and `Query` methods also have alternatives to provide an additional context: `PrepareContext` and `QueryContext`.

For efficiency and security, let's remember to use prepared statements when it makes sense.

10.4.4 Mishandling null values

The next mistake is to mishandle null values with queries.

Let's write an example where we will retrieve the department and age of an employee:

```
rows, err := db.Query("SELECT DEP, AGE FROM EMP WHERE ID = ?",
id)
if err != nil {
        return err
}
// Defer closing rows

var (
        department string
```

```
        age int
)
for rows.Next() {
        err := rows.Scan(&department, &age)
        if err != nil {
                return err
        }
        // ...
}
```

Execute the query 1
Scan earch row 2

We used `Query` to execute a query. Then, we iterate over the rows and use `Scan` to copy the column into the values pointed by the `department` and `age` pointers. If we run this example, we may get the following error while calling `Scan`:

```
2021/10/29 17:58:05 sql: Scan error on column index 0, name
"DEPARTMENT":
converting NULL to string is unsupported
```

Here, the SQL driver raises an error because the department value was equal to `NULL`. If a column can be nullable, there are two options to prevent `Scan` from returning an error.

The first approach is to declare `department` as a string pointer:

```
var (
        department *string
        age        int
)
for rows.Next() {
        err := rows.Scan(&department, &age)
        // ...
}
```

Changing the type from string to *string 1

What we provide to `scan` is the address of a pointer, not the address of a string type directly. By doing so, if the value is NULL, `department` will be nil.

The other approach is to use one of the `sql.NullXXX` types. For example, `sql.NullString`:

```
var (
        department sql.NullString
        age        int
)
for rows.Next() {
        err := rows.Scan(&department, &age)
        // ...
}
```

Changing the type to sql.NullString 1

`sql.NullString` is a wrapper on top of a string. It contains two exported fields: `String` containing the string value and `Valid` conveying whether the string isn't NULL. The following wrappers are accessible:

- `sql.NullString`
- `sql.NullBool`
- `sql.NullInt32`
- `sql.NullInt64`
- `sql.NullFloat64`
- `sql.NullTime`

Both approaches work, with perhaps `sql.NullXXX` expressing the intent more clearly as mentioned by Russ Cox, a core Go maintainer:

> There's no effective difference. We thought people might want to use NullString because it is so common and perhaps expresses the intent more clearly than *string. But either will work.
>
> -- Russ Cox

So, let's remember that the best practice with nullable columns is to either handle it as a pointer or use an `sql.NullXXX` type.

10.4.5 Not handling rows iteration errors

Another common mistake is to miss possible errors from iterating over the rows. Let's have a look at this function where error handling is misused:

```
1  func get(ctx context.Context, db *sql.DB, id string) (string,
2  int, error) {
3          rows, err := db.QueryContext(ctx,
4                  "SELECT DEP, AGE FROM EMP WHERE ID = ?", id)
5          if err != nil {
6                  return "", 0, err
7          }
8          defer func() {
9                  err := rows.Close()
10                 if err != nil {
11                         log.Printf("failed to close rows:
12 %v\n", err)
13                 }
14         }()
15
16         var (
17                 department string
18                 age        int
```

```
        )
        for rows.Next() {
                err := rows.Scan(&department, &age)
                if err != nil {
                        return "", 0, err
                }
        }

        return department, age, nil
}
```

Handle errors while executing the query 1
Handle errors while closing the rows 2
Handle errors while scanning a row 3

In this function, we handled three errors: while executing the query, closing the rows, and scanning a row. Yet, this isn't enough. We have to know that the `for rows.Next() {}` loop can break either when there's no more row or when an error happened while preparing the next row. Following a rows iteration, we should call `rows.Err` to distinguish between the two cases:

```
func get(ctx context.Context, db *sql.DB, id string) (string,
int, error) {
        // ...
        for rows.Next() {
                // ...
        }

        if err := rows.Err(); err != nil {
                return "", 0, err
        }

        return department, age, nil
}
```

Check rows.Err to know whether the previous loop stopped because of an error 1

This is the best practice to keep in mind: because `rows.Next` can stop either when we iterated over all the rows or when an error happened while preparing

the next row, we should check `rows.Err` following the iteration.

Let's now discuss a frequent mistake: forgetting to close transient resources.

10.5 Not closing transient resources

It's pretty frequent for developers to work with transient (temporary) resources that must be closed at some point in the code to avoid leaks, for example, on disk or in memory. Structs can generally implement the `io.Closer` interface to convey that a transient resource has to be closed. Let's delve into three common examples of what happens when resources aren't correctly closed and how to handle them properly.

10.5.1 HTTP body

First, let's discuss this problem in the context of HTTP.

We will write a `getBody` method that makes an HTTP GET request and returns the HTTP body response. Here's a first implementation:

```go
type handler struct {
        client http.Client
        url    string
}

func (h handler) getBody() (string, error) {
        resp, err := h.client.Get(h.url)
        if err != nil {
                return "", err
        }

        body, err := io.ReadAll(resp.Body)
        if err != nil {
                return "", err
        }

```

```
17         return string(body), nil
18 }
```

Make an HTTP GET request 1
Read resp.Body and get a body as a []byte 2

We use `http.Get` and parse the response using `io.ReadAll`. This method looks ok, and it correctly returns the HTTP response body. However, there's a resource leak. Let's understand where.

`resp` is an `*http.Response` type. It contains a `Body io.ReadCloser` field (`io.ReadCloser` implements both `io.Reader` and `io.Closer`). This body must be closed if `http.Get` doesn't return an error; otherwise, it's a resource leak. It will keep some memory allocated, which is no longer needed but can't be reclaimed by the GC and may prevent clients from reusing the TCP connection in the worst cases.

The most convenient way to deal with body closure is to handle it as a defer statement this way:

```
1 defer func() {
2         err := resp.Body.Close()
3         if err != nil {
4                 log.Printf("failed to close response: %v\n",
5 err)
6         }
  }()
```

In this implementation, we properly handle the body resource closure as a defer function that will be executed once `getBody` returns.

Note

> On the server-side, while implementing an HTTP handler, it isn't required to close the request body as it's done automatically by the server.

We should also understand that a response body has to get closed regardless of we read it or not. For example, if we are only interested in the HTTP status code and not in the body, it has to be closed no matter what to avoid a leak:

```go
func (h handler) getStatusCode(body io.Reader) (int, error) {
        resp, err := h.client.Post(h.url, "application/json", body)
        if err != nil {
                return 0, err
        }

        defer func() {
                err := resp.Body.Close()
                if err != nil {
                        log.Printf("failed to close response: %v\n", err)
                }
        }()

        return resp.StatusCode, nil
}
```

Close response body even if we don't read it 1

This function closes the body even though we haven't read it.

Another essential thing to remember, there's a different behavior depending on whether we close the body with or without having read from it:

- If we close the body without a read, the default HTTP transport may close the connection

- If we close the body following a read, the default HTTP transport won't close the connection; hence, it may be reused

Therefore, if `getStatusCode` is called repeatedly and we want to leverage keep-alive connections, even though we aren't interested in the body, we should read it nonetheless:

```go
func (h handler) getStatusCode(body io.Reader) (int, error) {
        resp, err := h.client.Post(h.url, "application/json", body)
        if err != nil {
                return 0, err
        }

        // Close response body

        _, _ = io.Copy(io.Discard, resp.Body)

        return resp.StatusCode, nil
}
```

Read the response body 1

In this example, we read the body to keep the connection alive. Note that instead of using `io.ReadAll`, we used `io.Copy` to `io.Discard`, an `io.Writer` implementation. This code reads the body but discards it without any copy, making it more efficient than `io.ReadAll`.

Note

It's not that infrequent to see implementations that close the body if the response isn't empty, not if the error is nil:

```
resp, err := http.Get(url)
if resp != nil {
        defer resp.Body.Close()
}

if err != nil {
        return "", err
}
```

If response isn't nil 1
Close the response body as a defer function 2

This implementation isn't necessary. It's based on the fact that in some conditions (e.g., redirection failure), `resp` and `err` will both be not nil. Yet, according to the [official Go documentation]:

On error, any Response can be ignored. A non-nil Response with a non-nil error only occurs when CheckRedirect fails, and even then, the returned Response.Body is already closed.

Therefore, the `if resp != nil {}` check isn't necessary. We should stick with the initial solution that closes the body in a defer function only if there is no error.

Closing a resource to avoid leaks isn't solely related to HTTP body management. In general, all the struct implementing the `io.Closer` interface should be closed at some point. This interface contains a single `Close` method:

```
type Closer interface {
        Close() error
}
```

Let's now see the impacts with `sql.Rows`.

10.5.2 `sql.Rows`

`sql.Rows` is a struct used as a result of an SQL query. As this struct implements `io.Closer`, it has to be closed. The following example omits closing the rows:

```go
db, err := sql.Open("postgres", dataSourceName)
if err != nil {
        return err
}

rows, err := db.Query("SELECT * FROM CUSTOMERS")
if err != nil {
        return err
}

// Use rows

return nil
```

Perform the SQL query 1

Forgetting to close the rows means a connection leak, preventing the DB connection from being put back into the connection pool.

We can handle the closure as a defer function following the `if err != nil` block:

```go
// Open connection

rows, err := db.Query("SELECT * FROM CUSTOMERS")
if err != nil {
        return err
}
```

```
7
8  defer func() {
9          if err := rows.Close(); err != nil {
10                 log.Printf("failed to close rows: %v\n", err)
11         }
12 }()
13
14 // Use rows
```

Perform an SQL query 1
Close the rows 2

Following the `Query` call, we should eventually close `rows` to prevent a connection leak if it doesn't return an error.

> **Note**
>
> As discussed in the previous section, the `db` variable (`*sql.DB` type) represents a pool of connections. It also implements the `io.Closer` interface. Yet, as the documentation suggests, it is rare to close an `sql.DB` as it's meant to be long-lived and shared between many goroutines.

Let's now discuss closing resources while working with files.

10.5.3 `os.File`

`os.File` represents an open file descriptor. In the same way as `sql.Rows`, it has to be closed eventually:

```
1  f, err := os.OpenFile(filename, os.O_APPEND|os.O_WRONLY,
2  os.ModeAppend)
3  if err != nil {
4          return err
5  }
```

```

defer func() {
        if err := f.Close(); err != nil {
                log.Printf("failed to close file: %v\n", err)
        }
}()
```

Open file 1
Close the file descriptor 2

In this example, we use `defer` to defer the call to the `Close` method. If we don't eventually close an `os.File`, it will not lead to a leak per se. Indeed, the file will be closed automatically when `os.File` will be garbage collected. However, it's better to call `Close` explicitly as we don't know when the next GC will be triggered (unless we manually run it).

There's also another benefit of calling `Close` explicitly: when we want to monitor the error returned actively. This should be the case with writable files, for example.

We should know that writing to a file descriptor isn't a synchronous operation. For performance concerns, data is buffered. The BSD man page for `close(2)` mentions that closure can lead to an error in a previously-uncommitted write (still living in a buffer) encountered an I/O error. For that reason, if we want to write to a file, we should probably propagate any error occurring while closing the file:

```
func writeToFile(filename string, content []byte) (err error)
{
        // Open file

        defer func() {
                closeErr := f.Close()
                if err == nil {
                        err = closeErr
                }
        }()
```

```

        _, err = f.Write(content)
        return
}
```

Return the close error if the write succeeded 1

In this example, we used named arguments and set the error to the response of `f.Close` if the write succeeded. This way, clients will be aware that something went wrong with this function and can react accordingly.

Furthermore, having a success while closing a writable `os.File` doesn't guarantee the file to be written on disk. Indeed, the write can still live in a buffer on the filesystem and not be flushed on disk. If durability is a critical factor, we can use the `Sync()` method to commit a change. In that case, errors coming from `Close` can be safely ignored:

```
1  func writeToFile(filename string, content []byte) error {
2          // Open file
3
4          defer func() {
5                  _ = f.Close()
6          }()
7
8          _, err = f.Write(content)
9          if err != nil {
10                 return err
11         }
12
13         return f.Sync()
14 }
```

Ignore possible errors 1
Commit the write to the disk 2

This example is a synchronous write function. It ensures that the content is written to disk before returning. Yet, the counterpart is an impact on performance.

To summarize this section, we've seen how important it is to close ephemeral resources to not lead to leaks. Ephemeral resources must be closed at the right time and in specific situations. Sometimes, it's not always clear upfront about what has to be closed. We can only acquire this knowledge by carefully reading the API documentation and/or experience. Yet, we should remember that if a struct implements the `io.Closer` interface, we must eventually call the `Close` method. Last but not least, it's also essential to understand what shall we do if a closure fails: is it enough to log a message, or should we also propagate it? The appropriate action depends on the implementation, as seen with the three examples in this section.

Let's now switch to common mistakes related to HTTP handling: forgetting return statements.

10.6 Forgetting the return statement after replying to an HTTP request

While writing an HTTP handler, it's easy to forget the return statement after replying to an HTTP request. This may lead to an odd situation where we should have stopped a handler after an error, but in fact, we didn't.

We can observe this situation in the following example:

```
1 func handler(w http.ResponseWriter, req *http.Request) {
2         err := foo(req)
3         if err != nil {
4                 http.Error(w, "foo",
5 http.StatusInternalServerError)
```

```
        }

        // ...
}
```

Handle the error 1

If `foo` returns an error, we handle it using `http.Error` that replies to the request with the `foo` error message and a 500 Internal Server Error. The problem with this code is that if we enter into the `if err != nil` branch, the application will continue its execution as `http.Error` doesn't stop the handler's execution.

What's the real impact of such an error? First, let's discuss at the HTTP level. For example, if we had completed the previous HTTP handler by adding a step to write a successful HTTP response body and status code:

```
func handler(w http.ResponseWriter, req *http.Request) {
        err := foo(req)
        if err != nil {
                http.Error(w, "foo",
http.StatusInternalServerError)
        }

        _, _ = w.Write([]byte("all good"))
        w.WriteHeader(http.StatusCreated)
}
```

In the case `err != nil`, the HTTP response would be the following:

```
foo
all good
```

Therefore, the response would contain both the error and successful messages.

Regarding the HTTP status code, we would return only the first one. In the previous example: 500. However, Go would also log a warning:

```
2021/10/29 16:45:33 http: superfluous response.WriteHeader call
from main.handler (main.go:20)
```

This warning means that we tried to write the status code multiple times, and this was superfluous.

Execution-wise, the main impact would be to continue the execution of a function that should have been stopped. For example, if `foo` was returning a pointer beside the error, continuing the execution would mean using this pointer and perhaps leading to a nil pointer dereference (hence a goroutine panic).

The fix for this mistake is to keep thinking about adding the `return` statement following `http.Error`:

```
func handler(w http.ResponseWriter, req *http.Request) {
        err := foo(req)
        if err != nil {
                http.Error(w, "foo",
http.StatusInternalServerError)
                return
        }

        // ...
}
```

Add the return statement 1

Thanks to the `return` statement, the function would stop its execution if we ended into the `if err != nil` branch.

This error is probably not the most complex of this book. Yet, it's so easy to forget about it that this mistake is a reasonably frequent one. Let's always remember that `http.Error` doesn't stop a handler execution and has to be added manually. Last but not least, such an issue can and should probably be caught during testing with decent coverage.

The last section of this chapter will keep discussing HTTP. We will present why production-grade applications shouldn't rely on the default HTTP client and server implementations.

10.7 Using the default HTTP client and server

The `http` package provides HTTP client and server implementations. However, it's pretty easy and frequent for developers to make a common mistake: relying on the default implementations in the context of applications deployed eventually in production. Let's understand what the problems are and how to overcome them.

10.7.1 HTTP client

First, let's define what a default client means; we will take as an example a GET request. It means either using the zero value of an `http.Client` struct like so:

```
client := &http.Client{}
resp, err := client.Get("https://golang.org/")
```

Or, using the `http.Get` function:

```
resp, err := http.Get("https://golang.org/")
```

In the end, both approaches are the same as the `http.Get` function uses `http.DefaultClient`, which is also based on the zero value of `http.Client`:

```
// DefaultClient is the default Client and is used by Get,
Head, and Post.
var DefaultClient = &Client{}
```

So, what's the problem with using the default HTTP client?

First, the default client doesn't specify any timeouts. This absence of timeout is probably not something we want for production-grade systems, and it can lead to many issues such as never-ending requests that could exhaust system resources.

Before delving into the available timeouts while making a request, let's review the five steps involved in an HTTP request:

1. Dial to establish a TCP connection
2. TLS handshake (if enabled)
3. Send the request
4. Read the response headers
5. Read the response body

Here are how these steps relate to the main client timeouts:

Figure 10.2. The five steps when doing an HTTP request and the related timeouts.

The four main timeouts are the following:

- `net.Dialer.Timeout` : Specifies the maximum amount of time a dial will wait for a connection to complete.
- `http.Transport.TLSHandshakeTimeout` : Specifies the maximum amount of time to wait for the TLS handshake.
- `http.Transport.ResponseHeaderTimeout` : Specifies the amount of time to wait for a server's response headers.
- `http.Client.Timeout` : Specifies the time limit for a request. It includes all the steps, from step one (dial) to step five (read the response body).

Note

This error means the endpoint failed to respond on time. We get this error about headers because reading them is the first step while waiting for a response.

You may already have encountered the following error if specifying the `http.Client.Timeout` :

```
net/http: request canceled (Client.Timeout
exceeded while awaiting headers)
```

Here’s an example of an HTTP client that overrides these timeouts:

```
client := &http.Client{
        Timeout: 5 * time.Second,
        Transport: &http.Transport{
                DialContext: (&net.Dialer{
                        Timeout: time.Second,
                }).DialContext,
                TLSHandshakeTimeout:   time.Second,
                ResponseHeaderTimeout: time.Second,
        },
}
```

Global request timeout 1
Dial timeout 2
TLS handshake timeout 3
Response header timeout 4

We created a client with a one-second timeout for the dial, the TLS handshake, and reading the response header. Meanwhile, each request will have a global five seconds timeout.

The second aspect to bear in mind with the default HTTP client is how connections are handled.

By default, the HTTP client does connection pooling. Indeed, the default client reuses connections (it can be disabled by setting `http.Transport.DisableKeepAlives` to true). There’s an extra timeout to specify how long an idle connection is kept into the pool:

`http.Transport.IdleConnTimeout`. The default value is 90 seconds, which means that the connection can be reused for other requests during this time. After that time, if the connection hasn't been reused, it will be closed.

To configure the number of connections in the pool, we must override `http.Transport.MaxIdleConns`. This value is set by default to 100. Yet, there's something important to note: the existence of a limit per host with `http.Transport.MaxIdleConnsPerHost`, which is set by default to 2. So, for example, if we trigger 100 requests to the same host, only 2 connections will remain in the connection pool after that. Hence, if we trigger 100 requests again, we will have to reopen at least 98 connections. This is also an important configuration to note as it can impact the average latency if we have to deal with a significant number of parallel requests to the same host.

Let's remember that for production-grade systems, we probably want to override the default timeouts. Meanwhile, tweaking the parameters related to connection pooling can also have a significant impact on the latency.

10.7.2 HTTP Server

We should also be careful while implementing an HTTP server. Again, a default client can be created using either the zero value of `http.Server`:

```
server := &http.Server{}
server.Serve(listener)
```

Or, using some functions such as `http.Serve`, `http.ListenAndServe`, or `http.ListenAndServeTLS` as they also rely on the default `http.Server`.

Once a connection is accepted, an HTTP response is divided into five steps:

1. Wait for the client to send the request
2. TLS handshake (if enabled)
3. Read the request headers
4. Read the request body
5. Write the response

> **Note**
>
> The TLS handshake doesn't have to be repeated with an already established connection.

And here is how these steps relate to the main server timeouts:

Figure 10.3. The five steps of an HTTP response and the related timeouts.

The three main timeouts are the following:

- `http.Server.ReadHeaderTimeout` : A field to specify the maximum amount of time to read the request headers.
- `http.Server.ReadTimeout` : A field to specify the maximum amount of time to read the entire request.

- `http.TimeoutHandler`: A wrapper function to specify the maximum amount of time for a handler to complete.

The last parameter isn't a server parameter but a wrapper on top of a handler to limit its duration. If a handler fails to respond on time, the server will reply a *503 Service Unavailable* with a specific message. Meanwhile, the context passed to the handler will be canceled.

> **Note**
>
> We purposely omitted `http.Server.WriteTimeout`, which isn't necessary since `http.TimeoutHandler` has been released (Go 1.8). Indeed, `http.Server.WriteTimeout` has a few issues. First, its behavior depends on whether TLS is enabled, making it more complex to understand and use. Furthermore, it closes the TCP connection without returning a proper HTTP code if the timeout is reached. Meanwhile, it doesn't propagate the cancellation to the handler context. Therefore, a handler may continue its execution without knowing that the TCP connection is already closed.

While exposing our endpoint to untrusted clients, the best practice is probably to set at least the `http.Server.ReadHeaderTimeout` field and to use the `http.TimeoutHandler` wrapper function. Otherwise, clients might exploit it and create never-ending connections that can lead to exhaustion of system resources, for example.

Here's how to set up a server with these timeouts in place:

```
s := &http.Server{
        Addr:                ":8080",
```

```
        ReadHeaderTimeout: 500 * time.Millisecond,
        ReadTimeout:       500 * time.Millisecond,
        Handler:           http.TimeoutHandler(handler,
time.Second, "foo"),
}
```

Wrap the HTTP handler 1

`http.TimeoutHandler` wraps the provided handler. Here, if `handler` fails to respond in one second, it returns a 503 with `foo` as the HTTP response.

In the same way that we described regarding HTTP clients, we can also configure on the server-side the maximum amount of time for the next request when keep-alives are enabled; it can be done using `http.Server.IdleTimeout`:

```
s := &http.Server{
        // ...
        IdleTimeout: time.Second,
}
```

Note that if `http.Server.IdleTimeout` isn't set, the value of `http.Server.ReadTimeout` is used for the idle timeout. If both aren't set, there won't be any timeouts, and connections will remain open until they get closed by clients.

For production-grade applications, we need to make sure not to use default HTTP clients and servers. Otherwise, it can lead to requests being stuck forever due to an absence of timeouts or even malicious clients exploiting the fact that our server doesn't have any timeouts.

10.8 Chapter summary

- We should remain cautious with functions accepting a `time.Duration`. Even though passing an integer is allowed, we should strive to use the time API to prevent any possible confusion.
- Avoiding calls to `time.After` in repeated functions (e.g., loops or HTTP handlers) can help us avoid peak memory consumption. Indeed, the resources created by `time.After` are released only when the timer expires.
- Let's remain careful about using embedded fields in Go structs. It may lead to sneaky bugs like the one we saw with an embedded `time.Time` field implementing the `json.Marshaler` interface, hence overriding the default marshaling behavior.
- To compare two `time.Time` structs, we have to recall that `time.Time` contains both a wall and a monotonic clock and that the comparison using the `==` operator is done on both clocks.
- To not make wrong assumptions while providing a map while unmarshalling JSON data, let's remember that numerics are converted into `float64` by default.
- Calling `Ping` or `PingContext` methods should be done if we must test our configuration and make sure a DB is reachable.
- Configuring the DB connection parameters should be done for production-grade applications.
- Using SQL prepared statements is a way to make queries more efficient and more secure.

- Dealing with nullable columns in tables should be done with pointers or `sql.NullXXX` types.
- Calling the `Err` method of `*sql.Rows` after rows iteration ensures that we haven't missed an error while preparing the next row.
- We should eventually close all structs implementing `io.Closer` to avoid possible leaks.
- To avoid unexpected behaviors in HTTP handler implementations, let's make sure not to miss the return statement if we want a handler to stop after `http.Error`.
- For production-grade applications, we should also make sure not to use the default HTTP client and server implementations. Indeed, these implementations are missing timeouts and behaviors that should be mandatory in production.

11 Testing

This chapter covers

- Delving into the helpful Go options to categorize tests or make them more robust
- Making Go tests deterministic by seeing how to prevent using sleep calls and knowing how to test functions using the time API
- Delving into utility packages such as `httptest` and `iotest`
- Discussing common benchmark mistakes leading to taking wrong assumptions
- Presenting a set of tips to improve the testing process

Testing is a crucial aspect of a project's lifecycle. It has countless benefits, such as building confidence in an application, acting as code documentation, or making refactoring easier. Compared to some other languages, Go has strong primitives to write tests. Throughout this section, we will delve into common mistakes that make the testing process brittle, less effective, and less accurate.

11.1 Not categorizing tests

The testing pyramid is a model that groups tests into different categories. Unit tests occupy the ground of the pyramid. Most tests should be unit

tests; therefore, cheap to write, fast to execute, and highly deterministic. Usually, the more we go upwards in the pyramid, the more the tests are complex to write, slow to run, and harder to guarantee their determinism:

Figure 11.1. An example of the testing pyramid.

A common technique is to be explicit about which kind of tests to run. Indeed, depending on the project lifecycle stage, we may want, for instance, to run only unit tests or run all the tests in the project. Not categorizing tests means potential waste of time and effort and losing accuracy about the scope of a test. This section will delve into three main ways to categorize tests in Go.

11.1.1 Build tags

The most common technique to classify tests is using build tags. A build tag is a special comment at the beginning of a Go file, followed by an empty line.

Listing 11.1. bar.go

```
//go:build foo

package bar
```

This file contains the `foo` tag. Note that within one package, we may have multiple files with different build tags.

> **Note**
>
> As Go 1.17, the original syntax `// +build foo` got replaced by `//go:build foo`. For the time being (Go 1.18), `gofmt` synchronizes the two forms to help the migration.

Build tags are used for two primary use cases. First, as a conditional option to build an application; for example, if we want a source file to be included only if cgo is enabled (cgo is a way to let Go packages call C code), we can add the `//go:build cgo` build tag. The second use case is if we want to categorize a test as an integration test, we can also add a specific build flag, for example, `integration`:

Listing 11.2. db_test.go

```
//go:build integration

package db

import (
        "testing"
)

func TestInsert(t *testing.T) {
        // ...
}
```

Here, we added the `integration` build tag to categorize that this file contains integration tests. The benefit of using build tags is that we can select which kind of tests to execute. For example, let's assume a package contains two test files:

- The one we just created, `db_test.go`
- Plus another one that doesn't contain any build tag, `contract_test.go`

If we run `go test` inside this package without any options, it will run only the test files without build tags; hence `contract_test.go`:

```
$ go test -v .
=== RUN   TestContract
--- PASS: TestContract (0.01s)
PASS
```

However, if we now provide the `integration` tag, it will also include `db_test.go`:

```
$ go test --tags=integration -v .
=== RUN   TestInsert
--- PASS: TestInsert (0.01s)
=== RUN   TestContract
--- PASS: TestContract (2.89s)
PASS
```

So, running tests with a specific tag includes both the files without tags and the files matching this tag. Now, what if we want to run *only* integration tests? A possible way is to add a negation tag on the unit test files. For example, using `!integration` means including the test file only if the `integration` flag is *not* enabled:

Listing 11.3. contract_test.go

```
//go:build !integration

package db

import (
        "testing"
)

func TestContract(t *testing.T) {
        // ...
}
```

Therefore, running `go test`:

- With the `integration` flag will run only the integration tests
- Without the `integration` flag will run only the unit tests

Let's now discuss another option that works at the level of a single test, not a file.

11.1.2 Environment variable

As mentioned by Peter Bourgon, a member of the Go community, build tags have one main drawback: the absence of signals that a test has been ignored[23]. In the first example, when we executed `go test` without build flags, it has shown only the tests that were executed:

```
$ go test -v .
=== RUN   TestUnit
--- PASS: TestUnit (0.01s)
PASS
ok      db  0.319s
```

If we're not careful enough with the way tags are handled, we may forget about existing tests. For that reason, some projects tend to favor the approach to check the test category using environment variables.

For example, we can implement the `TestInsert` integration test by checking a specific environment variable and potentially skipping the test:

```
func TestInsert(t *testing.T) {
        if os.Getenv("INTEGRATION") != "true" {
                t.Skip("skipping integration test")
        }

        // ...
}
```

If the `INTEGRATION` environment variable isn't set to true, then the test is skipped with a message:

```
```

```
$ go test -v .
=== RUN   TestInsert
    db_integration_test.go:12: skipping integration test
--- SKIP: TestInsert (0.00s)
=== RUN   TestUnit
--- PASS: TestUnit (0.00s)
PASS
ok      db  0.319s
```

Showing skipped message 1

One benefit of using this approach is making explicit the skipped tests and the reason for it. This technique is probably less widely used than build tags, but it’s worth knowing as it presents some advantages, as we discussed.

Let’s now delve into another way to categorize tests: short mode.

11.1.3 Short mode

Another approach to categorize the tests is related to their speed. Indeed, we may have to dissociate short from long-running tests.

As an illustration, we can have a set of unit tests but only one that is notoriously slow. Hence, we would like to categorize it in a particular manner so that we don’t have to run it every single time (especially if the trigger is after saving a file, for example). For that purpose, the short mode allows us to make the distinction:

```go
func TestLongRunning(t *testing.T) {
        if testing.Short() {
                t.Skip("skipping long-running test")
        }

```

```
        // ...
}
```

Mark the test as a long-running one 1

Using `testing.Short`, we can retrieve whether the short mode was enabled while running the tests. Then, we use `Skip` to skip the test. To run tests using the short mode, we have to pass `-short`:

```
% go test -short -v .
=== RUN   TestLongRunning
    foo_test.go:9: skipping long-running test
--- SKIP: TestLongRunning (0.00s)
PASS
ok      foo  0.174s
```

`TestLongRunning` was excluded and skipped explicitly from tests executed. Note that compared to build tags, this option works per test, not per file.

In summary, categorizing tests is a best practice for a successful testing strategy. Throughout this section, we've seen three ways to categorize tests:

- Using build tags at the test file level
- Using environment variables to mark a specific test
- Based on the test pace using the short mode

We can also combine both approaches. For example, using build tags or environment variables to classify a test (e.g., unit or integration) and the short mode if our project contains long-running unit tests.

In the next section, we will discuss why enabling the `race` flag matters.

11.2 Not enabling the race flag

In *Not understanding race problems*, we defined a data race when two goroutines simultaneously access the same variable, with at least one writing to this variable. Besides that, we should know that in Go, a standard tool exists to help in detecting data races. One common mistake is forgetting how important this tool is and not enabling it. This section will delve into what the race detector catches, how to use it, and the limitations.

In Go, the race detector isn't a static analysis tool that would happen during the compilation; instead, it's a tool to find data races that occur at runtime.

To enable it, we have to set the `-race` command-line while compiling or running a test, for example:

```
$ go test -race ./...
```

Once enabled, the compiler will instrument the code to detect data races. Instrumentation refers to a compiler adding extra instructions. Here, track all the memory accesses and record when and how they occurred. Then, at runtime, the race detector will watch for data races. However, we should keep in mind the runtime overhead of enabling the race detector:

- Memory usage may increase by 5-10x
- Execution time may increase by 2-20x

Because of this overhead, it's generally recommended to enable the race detector only during local testing or CI. In production, we should avoid it (or only use it in the case of canary releases, for example).

However, we should keep in mind that regardless of the execution context, the Go race detector has a strong limit on the number of goroutines executed simultaneously: 8128. Beyond this threshold, the race detector will stop.

If a race is detected, Go will raise a warning. For example, this example contains a data race as `i` can be accessed at the same time for both a read and a write:

```go
package main

import (
        "fmt"
)

func main() {
        i := 0
        go func() { i++ }()
        fmt.Println(i)
}
```

Running this application with the `-race` flag will log the following data race warning:

```
1  ==================
2  WARNING: DATA RACE
3  Write at 0x00c000026078 by goroutine 7:
4    main.main.func1()
5        /tmp/app/main.go:9 +0x4e
6
7  Previous read at 0x00c000026078 by main goroutine:
8    main.main()
9        /tmp/app/main.go:10 +0x88
10
11 Goroutine 7 (running) created at:
12   main.main()
13       /tmp/app/main.go:9 +0x7a
14 ==================
```

Indicate that goroutine 7 was writing 1
Indicate that the main goroutine was reading 2
Indicate when the goroutine 7 was created 3

Let's make sure we are comfortable in reading these messages. Go always logs:

- What are the concurrent goroutine incriminated: here the main goroutine and goroutine 7
- Where the accesses occur in the code: lines 9 and 10
- When were these goroutines created: goroutine 7 was created into `main()`

Note

Internally, the race detector uses vector clocks, a data structure used to determine a partial ordering of events (also used in distributed systems such as databases). Each goroutine creation leads to creating a vector clock. Then, the instrumentation updates the vector clock at each memory

> access and synchronization event. Then, it compares the vector clocks to detect potential data races.

We should note that the race detector cannot catch a false positive (a data race that isn't a real one). Therefore, we can know that our code contains a data race if we get a warning.

Conversely, it can lead in some cases to false negatives: missing actual data races. Two things to note testing-wise. First, the race detector can only be as good as our tests. Thus, we should ensure that a concurrent code is tested thoroughly against data races. Furthermore, given the possible false negatives, if we do have a test to check data races, one option could be to put this logic inside a loop. This way, we could increase the chances to catch possible data races:

```
1 func TestDataRace(t *testing.T) {
2         for i := 0; i < 100; i++ {
3                 // Actual logic
4         }
5 }
```

One last thing to note regarding tests. If a specific file contains tests that lead to data races, we can exclude it from the race detection using the `!race` build tag:

```
1  //go:build !race
2
3  package main
4
5  import (
6          "testing"
7  )
8
```

```go
func TestFoo(t *testing.T) {
        // ...
}

func TestBar(t *testing.T) {
        // ...
}
```

This file will only be built if the race detector is disabled. Otherwise, the entire file won't be built; hence, the tests won't be executed.

In summary, we should bear in mind that running tests with the `-race` flag for applications leveraging concurrency is highly recommended, if not mandatory. It allows enabling the race detector, which instruments our code to catch potential data races. While enabled, it has an important impact on memory and performance; hence it has to be used in specific conditions such as local tests or CI.

The following section will discuss two flags related to execution mode: `parallel` and `shuffle`.

11.3 Not using test execution modes

While running tests, the `go` command can accept a set of flags to impact how tests are executed. A common mistake is not to be aware of these flags and miss opportunities that could bring faster execution or a better way to spot possible bugs. Let's delve into two of these flags: `parallel` and `shuffle`.

11.3.1 Parallel

The first execution mode we should be aware of is parallel. It allows running specific tests in parallel, which can be very useful, for example, to speed up long-running tests.

We can mark that a test has to be run in parallel by calling `t.Parallel`:

```
1 func TestFoo(t *testing.T) {
2         t.Parallel()
3         // ...
4 }
```

When marking a test using `t.Parallel`, it will be executed in parallel alongside all the other parallel tests. In terms of execution, though, Go runs first all the sequential tests one by one. Then, once the sequential tests are completed, it executes the parallel tests.

For example, the following code contains three tests, but only two of them are marked to be run in parallel:

```
1  func TestA(t *testing.T) {
2          t.Parallel()
3          // ...
4  }
5
6  func TestB(t *testing.T) {
7          t.Parallel()
8          // ...
9  }
10
11 func TestC(t *testing.T) {
12         // ...
13 }
```

Running the tests for this file will give us the following logs:

```
=== RUN   TestA
=== PAUSE TestA
=== RUN   TestB
=== PAUSE TestB
=== RUN   TestC
--- PASS: TestC (0.00s)
=== CONT  TestA
--- PASS: TestA (0.00s)
=== CONT  TestB
--- PASS: TestB (0.00s)
PASS
```

Pause TestA 1

Pause TestB 2

Run TestC 3

Resume TestA and TestB 4

`TestC` is the first one to be executed. `TestA` and `TestB` are logged first, but they are paused, waiting for `TestC` to complete. Then, both are resumed and executed in parallel.

By default, the maximum number of tests that can run simultaneously equals the `GOMAXPROCS` value. If we want to serialize the tests or rather increase this number in the context of long-running tests doing a lot of I/O, for example, we can change this value using the `-parallel` flag:

```
$ go test -parallel 16 .
```

Here, the maximum number of parallel tests is set to 16.

Let's now see another mode while running Go tests: shuffle.

11.3.2 Shuffle

As of Go 1.17, it's now possible to randomize the execution order of tests and benchmarks.

What's the rationale for that? A best practice while writing tests is to make them isolated. For example, they shouldn't depend on the execution order or shared variables. These hidden dependencies can mean a possible test error or, even worse, a bug that won't be caught in testing. To prevent that, we should use the `-shuffle` flag to randomize tests. We can either provide `on` or `off` to enable or disable tests shuffle (disabled by default):

```
$ go test -shuffle=on -v .
```

However, in some cases, we would like to run the tests again in the same order. For example, if tests fail during CI, we may want to reproduce the error locally. To do that, instead of passing `on` to the `shuffle` flag, we can pass the seed used to randomize the tests.

We can access this seed value when running shuffled tests by enabling the verbose mode (`-v`):

```
$ go test -shuffle=on -v .
-test.shuffle 1636399552801504000
```

```
=== RUN   TestBar
--- PASS: TestBar (0.00s)
=== RUN   TestFoo
--- PASS: TestFoo (0.00s)
PASS
ok      teivah  0.129s
```

Seed value 1

Here, we executed the tests randomly, but `go test` printed the seed value: 1636399552801504000. To force running the tests in the same order, we can provide this seed value to `shuffle`:

```
$ go test -shuffle=1636399552801504000 -v .
-test.shuffle 1636399552801504000
=== RUN   TestBar
--- PASS: TestBar (0.00s)
=== RUN   TestFoo
--- PASS: TestFoo (0.00s)
PASS
ok      teivah  0.129s
```

The tests were executed in the same order: `TestBar` first, then `TestFoo`.

In general, we should be cautious about existing test flags and keep ourselves informed about new features coming with recent Go releases. Running tests in parallel can be an excellent way to decrease the overall execution time of running all the tests. Furthermore, the `shuffle` mode can help us spot hidden dependencies that may mean testing errors or even invisible bugs while running tests in the same order.

11.4 Not using table-driven tests

Table-driven tests are an efficient technique to write condensed tests, which reduces boilerplate code to help us in getting more focused on what matters: the testing logic. This section will go through a concrete example to see why table-driven tests are worth knowing when working with Go.

Let's consider the following function that removes all the new line suffixes (`\n` or `\r\n`) from a string:

```go
func removeNewLineSuffixes(s string) string {
        if s == "" {
                return s
        }
        if strings.HasSuffix(s, "\r\n") {
                return removeNewLineSuffixes(s[:len(s)-2])
        }
        if strings.HasSuffix(s, "\n") {
                return removeNewLineSuffixes(s[:len(s)-1])
        }
        return s
}
```

This function removes all the leading `\r\n` or `\n` recursively. Now, let's say that we want to test this function extensively. We should at least cover the following cases:

- Input is empty
- Input ends with `\n`
- Input ends with `\r\n`
- Input ends with multiple `\n`
- Input ends without newlines

One possible way is to create one unit test per case:

```go
func TestRemoveNewLineSuffix_Empty(t *testing.T) {
	got := removeNewLineSuffixes("")
	expected := ""
	if got != expected {
		t.Errorf("got: %s", got)
	}
}

func
TestRemoveNewLineSuffix_EndingWithCarriageReturnNewLine(t
*testing.T) {
	got := removeNewLineSuffixes("a\r\n")
	expected := "a"
	if got != expected {
		t.Errorf("got: %s", got)
	}
}

func TestRemoveNewLineSuffix_EndingWithNewLine(t
*testing.T) {
	got := removeNewLineSuffixes("a\n")
	expected := "a"
	if got != expected {
		t.Errorf("got: %s", got)
	}
}

func TestRemoveNewLineSuffix_EndingWithMultipleNewLines(t
*testing.T) {
	got := removeNewLineSuffixes("a\n\n\n")
	expected := "a"
	if got != expected {
		t.Errorf("got: %s", got)
	}
}

func TestRemoveNewLineSuffix_EndingWithoutNewLine(t
*testing.T) {
	got := removeNewLineSuffixes("a\n")
	expected := "a"
	if got != expected {
		t.Errorf("got: %s", got)
	}
}
```

Each function represents a specific case that we wanted to cover. However, we can notice two main drawbacks. First, the function names become more complex (`TestRemoveNewLineSuffix_EndingWithCarriageReturnNewLine` is 55-character long), so it can quickly impact the clarity of what the function is supposed to test. The second drawback is the amount of duplication among these functions as the structure is always the same:

- Call `removeNewLineSuffixes`
- Define the expected value
- Compare the values
- Log an error message

If we want to change one of these steps, for example, include the expected value as part of the error message, we will have to repeat it in all the tests. And the more tests we write, the more difficult it becomes to maintain.

For these reasons, we can use table-driven tests so that we write the logic only once. Table-driven tests rely on subtests, meaning the option for a single test function to include multiple subtests. For example, the following test contains two subtests:

```
1  func TestFoo(t *testing.T) {
2          t.Run("subtest 1", func(t *testing.T) {
3                  if false {
4                          t.Error()
5                  }
6          })
7          t.Run("subtest 2", func(t *testing.T) {
8                  if 2 != 2 {
9                          t.Error()
10                 }
11
```

```
        })
    }
```

Run a first subtest called subtest 1 1
Run a second subtest called subtest 2 2

`TestFoo` function includes two subtests. If we run this test, it will show the result for both `subtest 1` and `subtest 2`:

```
--- PASS: TestFoo (0.00s)
    --- PASS: TestFoo/subtest_1 (0.00s)
    --- PASS: TestFoo/subtest_2 (0.00s)
PASS
```

We can also run a single test using the `-run` flag and concatenating the parent test name with the subtest. For example, to run only `subtest 1`:

```
$ go test -run=TestFoo/subtest_1 -v
=== RUN   TestFoo
=== RUN   TestFoo/subtest_1
--- PASS: TestFoo (0.00s)
    --- PASS: TestFoo/subtest_1 (0.00s)
```

Use the run flag to run only subtest 1 1

Let’s now come back to our example and see how to leverage subtests to prevent duplicating the testing logic. The main idea is to create one subtest per case. Different variations exist, but we will discuss a map data structure where the key represents the test name, and the value represents the test data (input, expected).

Using a data structure containing test data and leveraging subtests to avoid boilerplate code is the concept of table-driven tests. Here's a possible implementation using a map:

```
func TestRemoveNewLineSuffix(t *testing.T) {
        tests := map[string]struct {
                input    string
                expected string
        }{
                `empty`: {
                        input:    "",
                        expected: "",
                },
                `ending with \r\n`: {
                        input:    "a\r\n",
                        expected: "a",
                },
                `ending with \n`: {
                        input:    "a\n",
                        expected: "a",
                },
                `ending with multiple \n`: {
                        input:    "a\n\n\n",
                        expected: "a",
                },
                `ending without newline`: {
                        input:    "a",
                        expected: "a",
                },
        }
        for name, tt := range tests {
                t.Run(name, func(t *testing.T) {
                        got :=
removeNewLineSuffixes(tt.input)
                        if got != tt.expected {
                                t.Errorf("got: %s,
expected: %s", got, tt.expected)
                        }
                })
        }
}
```

Define the test data 1

Each entry in the map represents a subtest 2
Iterate over the map 3
Run a new subtest for each map entry 4

The `tests` variable is a map. The key is the test name, whereas the value represents test data; in our case: input and expected string. Each map entry is a new test case that we wanted to cover. Then, we run a new subtest for each map entry.

This test solves the two drawbacks we discussed:

- Each test name is now a string instead of a PascalCase function name, making it simpler to read.
- The logic is written only once and shared for all the different cases. Modifying the testing structure or adding a new test requires minimal effort.

One last thing to mention regarding table driver tests that can also be a source of mistakes: as we discussed previously, we can mark a test run in parallel by calling `t.Parallel`. We can also do it in subtests inside the closure provided to `t.Run`:

```
1 for name, tt := range tests {
2         t.Run(name, func(t *testing.T) {
3                 t.Parallel()
4                 // Use tt
5         })
6 }
```

Mark the subtest to be run in parallel 1

However, we should note that this closure uses a loop variable. Therefore, to prevent a similar mistake discussed in *Not being careful with goroutines and loop variables*, leading the closures to possibly use a wrong value of the `tt` variable, we should either create another variable or shadow it:

```
for name, tt := range tests {
        tt := tt
        t.Run(name, func(t *testing.T) {
                t.Parallel()
                // Use tt
        })
}
```

Shadow tt to make it local to the loop iteration 1

This way, each closure will access its own `tt` variable.

In summary, we should remember that if multiple unit tests have a similar structure, we can mutualize them using table-driven tests. As it prevents duplication, it makes it simple to change the testing logic and easier to add new use cases.

Let's start to discuss how to prevent flaky tests in Go.

11.5 Sleeping in unit tests

A flaky test is a test that may both pass or fail without any code change. Flaky tests are seen among the biggest hurdles in testing as they are expensive to debug and undermine our confidence in testing accuracy. In Go, calling `time.Sleep` in a test can be a good signal of possible flakiness. Indeed, it's fairly frequent to test concurrent code, for example,

using sleeps. Let's see throughout this section concrete techniques to remove sleeps from tests to prevent from writing flaky tests.

We will illustrate this section with a function that returns a value and spins up a goroutine that performs a job in the background. We will call a function to get a slice of `Foo` structs and return the first element. In the meantime, the other goroutine will be in charge of calling a `Publish` method with the first `n` `Foo` elements:

```
1  type Handler struct {
2          n         int
3          publisher publisher
4  }
5
6  type publisher interface {
7          Publish([]Foo)
8  }
9
10 func (h Handler) getBestFoo(someInputs int) Foo {
11         foos := getFoos(someInputs)
12         best := foos[0]
13
14         go func() {
15                 if len(foos) > h.n {
16                         foos = foos[:h.n]
17                 }
18                 h.publisher.Publish(foos)
19         }()
20
21         return best
22 }
```

Get a slice of Foo 1
Keep the first one (checking the length of foos is omitted for the sake of simplicity) 2
Keep only the first n Foo 3
Call the Publish method 4

The `Handler` struct contains two fields: a `n` field and `publisher` dependency used to publish the first `n` `Foo` structs. First, we get a slice of `Foo`, but before returning the first one, we spin up a new goroutine, filter the `foos` slice and call `Publish`.

How can we test this function? Writing the part to assert the response is straightforward. However, what if we also want to check what is passed to `Publish`?

First, we could mock the `publisher` interface to record the arguments passed while calling the `Publish` method. Then, one first approach could be to sleep for a few milliseconds before checking the arguments recorded:

```go
type publisherMock struct {
        mu  sync.RWMutex
        got []Foo
}

func (p *publisherMock) Publish(got []Foo) {
        p.mu.Lock()
        defer p.mu.Unlock()
        p.got = got
}

func (p *publisherMock) Get() []Foo {
        p.mu.RLock()
        defer p.mu.RUnlock()
        return p.got
}

func TestGetBestFoo(t *testing.T) {
        mock := publisherMock{}
        h := Handler{
                publisher: &mock,
                n:         2,
        }

        foo := h.getBestFoo(42)
        // Check foo

```

```
        time.Sleep(10 * time.Millisecond)
        published := mock.Get()
        // Check published
}
```

Sleep for ten milliseconds before checking the arguments passed to Publish 1

First, we wrote a mock of `publisher` that would rely on a mutex to protect access to the `published` field. In our unit test, we call sleep to leave some time before checking the arguments passed to `Publish`.

This test is inherently flaky. There is no strict guarantee that ten milliseconds here would be enough (in this example, it is likely but not guaranteed).

So, what are the options to improve this unit test?

The first option is to assert a given condition periodically using retries. For example, we could write a function that takes an assertion as an argument and a maximum number of retries plus a wait time that would be called periodically to avoid a busy loop:

```
func assert(t *testing.T, assertion func() bool,
        maxRetry int, waitTime time.Duration) {
        for i := 0; i < maxRetry; i++ {
                if assertion() {
                        return
                }
                time.Sleep(waitTime)
        }
        t.Fail()
}
```

Check the assertion 1

```
Sleep before to retry 2
Fail eventually if after a number of retries 3
```

This function checks the assertion provided and fails after a certain number of retries. We are also using `time.Sleep`, but we could use a lower duration with this code. For example, coming back to `TestGetBestFoo`:

```
assert(t, func() bool {
        return len(mock.Get()) == 2
}, 30, time.Millisecond)
```

Instead of sleeping for ten milliseconds, we sleep each millisecond and configure a max number of retries. Such an approach reduces the time execution if the test succeeds as it means reducing the waiting interval. Therefore, implementing a retry strategy is a better approach than using passive sleeps.

> **Note**
>
> Some testing libraries such as testify also offer retry features. For example, in `testify`, we can use the `Eventually` function that implements what was described with other features such as configuring the error message.

Another strategy could be to use channels to synchronize the goroutine, publishing the `Foo` structs and the testing goroutine. For example, in the

mock implementation, instead of copying the slice received into a field, we could send this value to a channel:

```go
type publisherMock struct {
        ch chan []Foo
}

func (p *publisherMock) Publish(got []Foo) {
        p.ch <- got
}

func TestGetBestFoo(t *testing.T) {
        mock := publisherMock{
                ch: make(chan []Foo),
        }
        defer close(mock.ch)

        h := Handler{
                publisher: &mock,
                n:         2,
        }
        foo := h.getBestFoo(42)
        // Check foo

        if v := len(<-mock.ch); v != 2 {
                t.Fatalf("expected 2, got %d", v)
        }
}
```

Send the arguments received 1
Compare these arguments 2

The publisher sends the argument received to a channel. Meanwhile, the testing goroutine sets up the mock and creates the assertion based on the value received. It would also be possible to implement a timeout strategy to make sure we don't wait for `mock.ch` possibly forever in something wrong occurs. We could do it, for example, using `select` with a `time.After` case.

What option should we favor: retry or synchronization? In general, if synchronization is possible, it's probably the default option to choose. Indeed, it reduces the waiting time to the bare minimum and makes it fully deterministic if well designed. If we can't apply synchronization, we should perhaps reconsider our design since we may have a problem. If synchronization is truly impossible, we should use the retry option, which will be, in any case, a better option than passive sleeps to eradicate non-determinism in tests.

Let's keep discussing how to prevent flakiness from testing, this time when using the time API.

11.6 Not dealing with the time API efficiently

Some functions have to rely on the time API, for example, to retrieve the current time. In such a case, it can be pretty easy to write brittle unit tests that may fail at some point. In this section, we will go through a concrete example and discuss the possible options. The goal isn't to cover every use case and technique but to give directions about writing more robust tests of functions using the time API.

Let's say an application receives events that we want to store in an in-memory cache. We will implement a `Cache` struct to hold the most recent events. This struct will expose three methods:

- Append events
- Get all the events
- Trim the events for a given duration (we will focus on this method)

Each of these methods will need to access the current time. Let's write a first implementation of the latter method using `time.Now()` (we will assume that all the events are sorted by time):

```go
type Cache struct {
        mu     sync.RWMutex
        events []Event
}

type Event struct {
        Timestamp time.Time
        Data string
}

func (c *Cache) TrimOlderThan(since time.Duration) {
        c.mu.RLock()
        defer c.mu.RUnlock()

        t := time.Now().Add(-since)
        for i := 0; i < len(c.events); i++ {
                if c.events[i].Timestamp.After(t) {
                        c.events = c.events[i:]
                        return
                }
        }
}
```

Subtract the given duration to the current time 1
Trim the events 2

First, we compute a `t` variable which is the current time minus the provided duration. Then, as the events are sorted by time, we update the internal `events` slice as soon as we reach an event whose time is after `t`.

Now, how could we test this method? We could rely on the current time using `time.Now` to create the events:

```go
func TestCache_TrimOlderThan(t *testing.T) {
        events := []Event{
                {Timestamp: time.Now().Add(-20 * time.Millisecond)},
                {Timestamp: time.Now().Add(-10 * time.Millisecond)},
                {Timestamp: time.Now().Add(10 * time.Millisecond)},
        }
        cache := &Cache{}
        cache.Add(events)
        cache.TrimOlderThan(15 * time.Millisecond)
        got := cache.GetAll()
        expected := 2
        if len(got) != expected {
                t.Fatalf("expected %d, got %d", expected, len(got))
        }
}
```

Create events using time.Now() 1
Add these events to the cache 2
Trim the events since 15 milliseconds ago 3
Retrieve all the events and compare 4

Here, we add a slice of events to the cache using `time.Now()` and adding or subtracting some small durations. Then, we trim these events for 15 milliseconds, and we perform the assertion.

Such an approach has one main drawback. If the machine executing the test is suddenly busy, we may trim fewer events than expected. It would still be possible to increase the duration provided to reduce the percentage of having a failing test; however, this isn't always possible. For example, what if the timestamp field was an unexported field generated while adding an event? In this case, it wouldn't be possible to pass a specific timestamp, and perhaps one may end up adding sleeps in the unit test.

The problem is related to the implementation of `TrimOlderThan`. As it calls `time.Now()`, it makes it harder to implement robust unit tests. Let's discuss two main approaches that would make our test less brittle.

The first approach is to make the way to retrieve the current time a dependency to the `Cache` struct. In production, we would inject the *real* implementation, whereas, in unit tests, we would pass a stub, for example.

To handle this dependency, there are different options, such as an interface or a function type. In our case, as we only rely on a single method (`time.Now()`), we can define a function type:

```go
type now func() time.Time

type Cache struct {
        mu     sync.RWMutex
        events []Event
        now    now
}
```

The `now` type is a function that returns a `time.Time`. In the factory function, we can pass the actual `time.Now` function this way:

```go
func NewCache() *Cache {
        return &Cache{
                events: make([]Event, 0),
                now:    time.Now,
        }
}
```

As the `now` dependency remains unexported, it wouldn't be accessible for external clients. Furthermore, in our unit test, we can create a `Cache` struct by injecting a *fake* implementation of `func() time.Time` based on a predefined time:

```go
func TestCache_TrimOlderThan(t *testing.T) {
        events := []Event{
                {Timestamp: parseTime(t, "2020-01-01T12:00:00.04Z")},
                {Timestamp: parseTime(t, "2020-01-01T12:00:00.05Z")},
                {Timestamp: parseTime(t, "2020-01-01T12:00:00.06Z")},
        }
        cache := &Cache{now: func() time.Time {
                return parseTime(t, "2020-01-01T12:00:00.06Z")
        }}
        cache.Add(events)
        cache.TrimOlderThan(15 * time.Millisecond)
        // ...
}

func parseTime(t *testing.T, timestamp string) time.Time {
        // ...
}
```

Create events based on specific timestamps 1
Inject a static function to fix the time 2

While creating a new `Cache` struct, we inject the `now` dependency based on a given time. Thanks to this approach, the test is now robust. Even in the worst conditions, the outcome of this test is deterministic.

Note

Instead of using a field, it's also possible to manage to retrieve the time via a global variable:

```
var now = time.Now
```

In general, we should tend to prevent having such a mutable shared state. In our case, it would lead to at least one concrete issue: tests wouldn't be isolated anymore as they would all depend on a shared variable. Therefore, the tests couldn't be run in parallel, for example. If possible, we should instead handle these cases as part of struct dependencies, fostering testing isolation.

This solution is also extensible. For example, what if the function calls `time.After`? We could either add another `after` dependency or create one interface grouping the two methods: `Now` and `After`. However, this approach has one main drawback. The `now` dependency isn't available if we create a unit test from an external package for example (we will explore it in *Not exploring all the Go testing features*).

In that case, another option is possible; instead of handling the time as an unexported dependency, we can also ask clients to provide the current time:

```
func (c *Cache) TrimOlderThan(now time.Time, since
time.Duration) {
        // ...
}
```

Yet, to go even further, we could also *merge* the two function arguments in a single `time.Time` that would represent a specific point in time until which we want to trim the events:

```
func (c *Cache) TrimOlderThan(t time.Time) {
        // ...
}
```

Hence, it would be up to the caller to calculate this point in time:

```
cache.TrimOlderThan(time.Now().Add(time.Second))
```

And it would be the same testing-wise:

```
func TestCache_TrimOlderThan(t *testing.T) {
        // ...
        cache.TrimOlderThan(parseTime(t, "2020-01-
01T12:00:00.06Z").
                Add(-15 * time.Millisecond))
        // ...
}
```

This option is probably the simplest as it doesn't require creating another type and a stub.

In general, we should be cautious about testing code that uses the `time` API. It can also be an open door for flaky tests. We have seen throughout this section two ways to deal with it. One option is to keep the `time`

interactions as part of a dependency that we can fake in unit tests. We can implement our own implementations or rely on external libraries for that. Another option, which is simpler but more limited, is to rework our API and ask the clients to provide us with the information we need, such as the current time.

Let's now delve into two helpful Go packages related to testing: `httptest` and `iotest`.

11.7 Not using testing utility packages

The standard library provides utility packages for testing. A common mistake is to be unaware of these packages and try to reinvent the wheel or rely on other solutions that wouldn't be that handy. This section will delve into two of these packages, one to help us when using HTTP and another one when doing I/O and using readers and writers.

11.7.1 `httptest`

The `httptest` package provides utilities for HTTP testing for both clients and servers. Let's delve into these two use cases.

First, let's see how `httptest` can help us while writing an HTTP server. We will implement a handler that performs some basic actions: writing a header, a body, and returning a specific status code. For the sake of clarity, we will omit error handling:

```
1 func Handler(w http.ResponseWriter, r *http.Request) {
2         w.Header().Add("X-API-VERSION", "1.0")
3         b, _ := io.ReadAll(r.Body)
4         _, _ = w.Write(append([]byte("hello "), b...))
```

```
5         w.WriteHeader(http.StatusCreated)
6 }
```

Concatenate `hello ` with the request body 1

An HTTP handler accepts two arguments: the request and a way to write the response. The `httptest` package provides utilities for both. For the request, we can use `httptest.NewRequest` to build a `*http.Request` using an HTTP method, an URL, and a body. Regarding the response, we can use `httptest.NewRecorder` to record the mutations made within the handler. Let's write a unit test of this handler:

```
1  func TestHandler(t *testing.T) {
2          req := httptest.NewRequest(http.MethodGet,
3  "http://localhost",
4                  strings.NewReader("foo"))
5          w := httptest.NewRecorder()
6          Handler(w, req)
7
8          if got := w.Result().Header.Get("X-API-VERSION");
9  got != "1.0" {
10                 t.Errorf("api version: expected 1.0, got
11 %s", got)
12         }
13
14         body, _ := ioutil.ReadAll(w.Body)
15         if got := string(body); got != "hello foo" {
16                 t.Errorf("body: expected hello foo, got
17 %s", got)
18         }
19
           if http.StatusOK != w.Result().StatusCode {
                   t.FailNow()
           }
   }
```

Build the request 1

Create the response recorder 2
Call the handler 3
Verify HTTP header 4
Verify HTTP body 5
Verify HTTP status code 6

Testing a handler using `httptest` doesn't test the transport (the HTTP part). The focus of the test is to call the handler directly with a request and a way to record the response. Then, using the response recorder, we wrote the assertions to verify the HTTP header, the body, and the status code.

Let's now see the other side of the coin when we want to test an HTTP client.

We will write a client in charge to query an HTTP endpoint that calculates how long it takes to drive from one coordinate to another. The client would look like this:

```
1  func (c DurationClient) GetDuration(url string,
2          lat1, lng1, lat2, lng2 float64) (
3          time.Duration, error) {
4          resp, err := c.client.Post(
5                  url, "application/json",
6                  buildRequestBody(lat1, lng1, lat2, lng2),
7          )
8          if err != nil {
9                  return 0, err
10         }
11
12         return parseResponseBody(resp.Body)
13 }
```

It performs an HTTP POST request to the provided URL and returns the parsed response (let's say, some JSON).

What if we want to test this client? One option could be to use Docker and spin up a mock server to return some pre-registered responses. However, this approach makes the test slow to execute. The other option is to use `httptest.NewServer` to create a local HTTP server based on a handler that we will provide. Once the server is up and running, we will use its URL and pass it to `GetDuration`:

```go
func TestDurationClientGet(t *testing.T) {
        srv := httptest.NewServer(
                http.HandlerFunc(
                        func(w http.ResponseWriter, r *http.Request) {
                                _, _ = w.Write([]byte(`{"duration": 314}`))
                        },
                ),
        )
        defer srv.Close()

        client := NewDurationClient()
        duration, err := client.GetDuration(srv.URL, 51.551261, -0.1221146, 51.57, -0.13)
        if err != nil {
                t.Fatal(err)
        }

        if duration != 314*time.Second {
                t.Errorf("expected 314 seconds, got %v", duration)
        }
}
```

Start the HTTP server 1

Register the handler to serve the response 2

Shut down the server 3
Provide the server URL 4
Verify the response 5

In this test, we create a server with a static handler returning `314` seconds. It would also be possible to make assertions based on the request sent. Furthermore, when we call `GetDuration`, we provide the URL of the server started. Compared to testing a handler, this test does perform an actual HTTP call, but it executes in only a few milliseconds.

It's also possible to start a new server using TLS with `httptest.NewTLSServer` and create an unstarted server with `httptest.NewUnstartedServer` so that we can start it lazily.

Let's remember how much `httptest` is helpful when working in the context of HTTP applications. Be it while writing a server or a client, `httptest` can help us create efficient tests.

11.7.2 `iotest`

The `iotest` package implements utilities for testing readers and writers. It's a convenient package that Go developers too often forget.

First, when implementing a custom `io.Reader`, we should remember to test it using `iotest.TestReader`. This utility function tests that a reader behaves correctly: it correctly returns the number of bytes read, fills the provided slice, etc. It also tests different behaviors if the provider reader implements interfaces such as `io.ReaderAt`.

Let's assume a custom `LowerCaseReader` that streams lower case letters from a given input `io.Reader`. Here's how to test this reader doesn't misbehave:

```
1 func TestLowerCaseReader(t *testing.T) {
2         err := iotest.TestReader(
3                 &LowerCaseReader{reader:
4 strings.NewReader("aBcDeFgHiJ")},
5                 []byte("acegi"),
6         )
7         if err != nil {
8                 t.Fatal(err)
9         }
  }
```

Provide an io.Reader 1
Expectation 2

We call `io.TestReader` by providing the custom `LowerCaseReader` and an expectation, the lowercase letters: `acegi`.

Another use case to use `iotest` package[24] is to make sure an application using readers and writers is tolerant to errors:

- `iotest.ErrReader`: creates an `io.Reader` that returns a provided error
- `iotest.HalfReader`: creates an `io.Reader` that reads only half as many requested bytes from an `io.Reader`
- `iotest.OneByteReader`: creates an `io.Reader` that reads a single byte for each non-empty read from an `io.Reader`

- `iotest.TimeoutReader`: creates an `io.Reader` that returns an error on the second read with no data, but subsequent calls will succeed
- `iotest.TruncateWriter`: creates an `io.Writer` that writes to an `io.Writer` but stops silently after n bytes

For example, let's assume we implement the following function that starts from reading all the bytes from a reader:

```
1 func foo(r io.Reader) error {
2         b, err := io.ReadAll(r)
3         if err != nil {
4                 return err
5         }
6
7         // ...
8 }
```

We want to make sure our function is resilient if, for example, the provided reader fails during a read (e.g., to simulate a network error):

```
1 func TestFoo(t *testing.T) {
2         err := foo(iotest.TimeoutReader(
3                 strings.NewReader(randomString(1024)),
4         ))
5         if err != nil {
6                 t.Fatal(err)
7         }
8 }
```

Wrap the provided io.Reader using io.TimeoutReader 1

Here, we wrapped an `io.Reader` using `io.TimeoutReader`. As we mentioned, the second read is going to fail. If we run this test to make sure our function is tolerant to error, we will get a test failure. Indeed, `io.ReadAll` returns any errors it found.

Knowing this, we could implement our custom `readAll` function that would tolerate up to n errors:

```go
func readAll(r io.Reader, retries int) ([]byte, error) {
	b := make([]byte, 0, 512)
	for {
		if len(b) == cap(b) {
			b = append(b, 0)[:len(b)]
		}
		n, err := r.Read(b[len(b):cap(b)])
		b = b[:len(b)+n]
		if err != nil {
			if err == io.EOF {
				return b, nil
			}
			retries--
			if retries < 0 {
				return b, err
			}
		}
	}
}
```

Tolerate retries 1

This implementation is similar to `io.ReadAll`, but it also handles configurable retries. If we change the implementation of our initial function to use our custom `readAll` instead of `io.ReadAll`, the test won't fail anymore:

```go
func foo(r io.Reader) error {
        b, err := readAll(r, 3)
        if err != nil {
                return err
        }

        // ...
}
```

Indicate up to three retries 1

We have seen an example of how to check that a function is tolerant to errors while reading from an `io.Reader`. We performed the test by relying on `iotest` package.

When doing I/O and working with `io.Reader` and `io.Writer`, let's remember how handy `iotest` package is. As we have seen, it provides utilities, including some to test the behavior of a custom `io.Reader` and test our application against errors occurring while reading or writing data.

The following section will discuss some of the common traps that can lead to writing inaccurate benchmarks.

11.8 Writing inaccurate benchmarks

In general, we should never guess about performance. When writing some optimizations, so many factors may come into play that even if we have a strong opinion about results, it's rarely a bad idea to test it. However, writing benchmarks isn't straightforward. Indeed, it can be pretty simple to write inaccurate benchmarks and take wrong assumptions because of that. The goal of this section is to delve into common and concrete traps leading to inaccuracy.

Before discussing these traps, let's briefly remember how benchmarks work in Go.

The skeleton of a benchmark is the following:

```
1 func BenchmarkFoo(b *testing.B) {
2         for i := 0; i < b.N; i++ {
3                 foo()
4         }
5 }
```

The function name starts with the `Benchmark` prefix. The function under test (`foo`) is called within the `for` loop. `b.N` represents a variable number of iterations. When running a benchmark, Go will try to make it match the requested benchmark time. The benchmark time is set by default to one second and can be changed with the `-benchtime` flag. `b.N` starts at one; if the benchmark completes in under one second, `b.N` is increased, and the benchmark runs again until `b.N` roughly match `benchtime`:

```
1 $ go test -bench=.
2 cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
3 BenchmarkFoo-4                73          16511228 ns/op
```

Here, the benchmark took about one second, and `foo` was executed 73 times for an average execution time of 16511228 nanoseconds. We can change the benchmark time using `-benchtime`:

```
$ go test -bench=. -benchtime=2s
BenchmarkFoo-4                150          15832169 ns/op
```

Now, `foo` has been executed roughly twice more than during the previous benchmark.

Let's now delve into these common traps.

11.8.1 Not resetting or pausing the timer

In some cases, we need to perform some operations before the benchmark loop. Sometimes, these operations can take quite a while (e.g., generating a large slice of data) and may have a significant impact on the benchmark results:

```
func BenchmarkFoo(b *testing.B) {
        expensiveSetup()
        for i := 0; i < b.N; i++ {
                functionUnderTest()
        }
}
```

In this case, what we can do is to use the `ResetTimer` method before entering the loop:

```
func BenchmarkFoo(b *testing.B) {
        expensiveSetup()
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                functionUnderTest()
```

```
        }
}
```

Reset the benchmark timer 1

Calling `ResetTimer` zeroes the elapsed benchmark time and memory allocation counters since the beginning of the test. This way, an expensive setup can be discarded from the test results.

Now, what if we have to perform some expensive setup not only once but within each loop iteration?

```
func BenchmarkFoo(b *testing.B) {
        for i := 0; i < b.N; i++ {
                expensiveSetup()
                functionUnderTest()
        }
}
```

Here, we can’t reset the timer as it would be executed during each loop iteration. Yet, it’s possible to stop and resume the benchmark timer to surround the call to `expensiveSetup`:

```
func BenchmarkFoo(b *testing.B) {
        for i := 0; i < b.N; i++ {
                b.StopTimer()
                expensiveSetup()
                b.StartTimer()
                functionUnderTest()
        }
}
```

Pause the benchmark timer 1

Resume the benchmark timer 2

Here, we pause the benchmark timer to perform the expensive setup and then resume it.

> **Note**
>
> There's one catch to remember about this approach: if the function under test is too fast to execute compared to the setup function, the benchmark may take too long to complete. The reason is that it would take way more than one user second to reach `benchtime`. Indeed, calculating the benchmark time is based solely on the execution time of `functionUnderTest`. So, if we wait for a significant time in each loop iteration, it will make the benchmark way slower than one second. If we want to keep the benchmark, one possible mitigation is to decrease `benchtime`.

Let's make sure to use the timer methods to preserve the accuracy of a benchmark.

11.8.2 Making wrong assumptions about micro-benchmarks

A micro-benchmark is about measuring a tiny computation unit, and it can sometimes be effortless to take wrong assumptions about it.

Let's say, for example, that we hesitate between using `atomic.StoreInt32` or `atomic.StoreInt64` (assuming that the values we

will handle will always fit in 32 bits). We want to write a benchmark to compare both functions:

```go
func BenchmarkAtomicStoreInt32(b *testing.B) {
        var v int32
        for i := 0; i < b.N; i++ {
                atomic.StoreInt32(&v, 1)
        }
}

func BenchmarkAtomicStoreInt64(b *testing.B) {
        var v int64
        for i := 0; i < b.N; i++ {
                atomic.StoreInt64(&v, 1)
        }
}
```

If we run this example, here's a possible output that we can get:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkAtomicStoreInt32
BenchmarkAtomicStoreInt32-4     197107742
5.682 ns/op
BenchmarkAtomicStoreInt64
BenchmarkAtomicStoreInt64-4     213917528
5.134 ns/op
```

We could easily take this benchmark for granted and decide to use `atomic.StoreInt64` as it looks faster. Now, we want for the sake of doing a *fair* benchmark to revert the order, so testing `atomic.StoreInt64` first, then `atomic.StoreInt32`. Here could be another possible output:

```
BenchmarkAtomicStoreInt64
BenchmarkAtomicStoreInt64-4     224900722
5.434 ns/op
```

```
4 BenchmarkAtomicStoreInt32
  BenchmarkAtomicStoreInt32-4     230253900
  5.159 ns/op
```

Now, `atomic.StoreInt32` has better results, so what happened?

In the case of micro-benchmarks, many factors can impact the results, such as the machine activity running the benchmarks, power management, thermal scaling, a better cache alignment of a sequence of instructions. We must remember that many factors, even outside of the scope of our Go project that can impact the results.

> **Note**
>
> We should make sure that the machine executing the benchmark is idle. However, external processes may run in the background, which may impact the benchmark results. For that reason, some tools such as `perflock` exist to limit how much CPU a benchmark can consume. For example, we can run a benchmark with 70% of the total available CPU. This way, it gives 30% room to the OS and other processes. Therefore, it would help to lower the impact of the machine activity factor on the results.

One first option can be to increase the benchmark time using the `-benchtime` option. In a similar fashion as the law of large numbers in the probability theory, running a benchmark a large number of times should tend to become closer to its expected value (assuming we omit the benefits of instructions caching and such mechanics).

Another option can be to use external tools on top of the classic benchmark tooling. One of them is part of the `golang.org/x` repository: `benchstat`. This tool allows us to compute and compare statistics about benchmark executions.

For example, let's run the benchmark ten times using the `-count` option and pipe the output to a specific file:

```
$ go test -bench=. -count=10 | tee stats.txt
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkAtomicStoreInt32-4     234935682
5.124 ns/op
BenchmarkAtomicStoreInt32-4     235307204
5.112 ns/op
// ...
BenchmarkAtomicStoreInt64-4     235548591
5.107 ns/op
BenchmarkAtomicStoreInt64-4     235210292
5.090 ns/op
// ...
```

We can then run `benchstat` on this file:

```
$ benchstat stats.txt
name                 time/op
AtomicStoreInt32-4   5.10ns ± 1%
AtomicStoreInt64-4   5.10ns ± 1%
```

Now, we observe that the results are the same: both functions take on average 5.10 ns to complete. It also shows the percent variation between the executions of a given benchmark: ± 1%. This metric tells us that both benchmarks are really stable, giving us more confidence in the computed

average results. Therefore, instead of concluding that `atomic.StoreInt32` is faster or slower, we can conclude that it has a similar execution time as `atomic.StoreInt64` for the usage we tested (in a specific Go version on a particular machine).

In general, we should be cautious about micro-benchmarks. Indeed, many factors can have a significant impact on the results and lead to potential wrong assumptions. Increasing the benchmark time or repeating the benchmark executions and computing stats with tools such as `benchstat` can be an efficient way to limit external factors and get more accurate results leading to better conclusions.

Also, let's highlight that we should take with care the results of a micro-benchmark executed on a given machine if the system that ends up running the application is different. Indeed, the production system may act quite differently from the one we ran the micro-benchmark.

11.8.3 Not being careful about compiler optimizations

Another common mistake related to writing benchmarks is getting fooled by some compiler optimizations, as it can also lead to wrong benchmark assumptions.

In this section, we will take the Go issue 14813[25], also highlighted by Dave Cheney, with a population count function (a function that counts the number of bits set to one):

```
const m1 = 0x5555555555555555
const m2 = 0x3333333333333333
const m4 = 0x0f0f0f0f0f0f0f0f
const h01 = 0x0101010101010101

```

```
func popcnt(x uint64) uint64 {
        x -= (x >> 1) & m1
        x = (x & m2) + ((x >> 2) & m2)
        x = (x + (x >> 4)) & m4
        return (x * h01) >> 56
}
```

This function takes and returns an `uint64`. To benchmark this function, one may write the following:

```
func BenchmarkPopcnt1(b *testing.B) {
        for i := 0; i < b.N; i++ {
                popcnt(uint64(i))
        }
}
```

However, if we execute this benchmark, we will get a result surprisingly low:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkPopcnt1-4      1000000000              0.2858 ns/op
```

A duration of 0.28 nanoseconds is roughly one clock cycle, so this number is unreasonably low. The problem here is that the developer wasn't careful enough about compiler optimizations. Indeed, in this case, the function under test is simple enough to be a candidate for inlining. Inlining is an optimization that replaces a function call with the body of the called function, and it allows preventing a function call, which has a small

footprint. Once the function is inlined, the compiler can notice the call has no side effects and replace it with the following benchmark:

```
1 func BenchmarkPopcnt1(b *testing.B) {
2         for i := 0; i < b.N; i++ {
3                 // Empty
4         }
5 }
```

The benchmark is now empty. Therefore, it’s the reason why we got a result close to one clock cycle. To prevent it from happening, a best practice is to follow this pattern:

- During each loop iteration, assign the result to a local variable (local in the context of the benchmark function)
- Assign the latest result to a global variable

In our case, it would mean writing the following benchmark:

```
1 var global uint64
2
3 func BenchmarkPopcnt2(b *testing.B) {
4         var v uint64
5         for i := 0; i < b.N; i++ {
6                 v = popcnt(uint64(i))
7         }
8         global = v
9 }
```

Define a global variable 1
Define a local variable 2
Assign the result to the local variable 3
Assign the result to the global variable 4

`global` is a global variable, whereas `v` is a local variable whose scope is the benchmark function. During each loop iteration, we assign the result of `popcnt` to the local variable. Then, we assign the latest result to the global variable.

> **Note**
>
> We may ask why not assign the result of the `popcnt` call directly to `global` so that it simplifies the test? Writing to a global variable is slower than a local variable (we will delve into these concepts in *Not understanding stack vs. heap*). Therefore, we should write each result to a local variable to limit the footprint during each loop iteration.

If we run these two benchmarks, we will now get a significant difference in the results:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkPopcnt1-4      1000000000               0.2858
ns/op
BenchmarkPopcnt2-4      606402058                1.993 ns/op
```

`BenchmarkPopcnt2` is the accurate version of the benchmark. It guarantees that we avoided the inlining optimizations, which can artificially lower the execution time or even remove the call to the function under test. Therefore, relying on the results of `BenchmarkPopcnt1` could have led to wrong assumptions.

Let’s remember the pattern to avoid compiler optimizations fooling the benchmark results: assigning the result of the function under test to a local variable and then assigning the latest result to a global variable. Such a best practice also prevents from making wrong assumptions.

11.8.4 Being fooled by the observer effect

In physics, the observer effect is the disturbance of an observed system by the act of observation. This effect can also be seen in benchmarks leading to making wrong assumptions about results. Let’s see a concrete example and then try to mitigate it.

We want to implement a function receiving a matrix of `int64` elements. This matrix has a fixed number of 512 columns, and we want to compute the total sum of the first eight columns.

Figure 11.2. Compute the sum of the first eight columns.

For the sake of optimizations, we also want to see whether varying the number of columns has an impact, so we will also implement a second function with 513 columns. The implementation is the following:

```
1  func calculateSum512(s [][512]int64) int64 {
2          var sum int64
3          for i := 0; i < len(s); i++ {
4                  for j := 0; j < 8; j++ {
5                          sum += s[i][j]
6                  }
7          }
8          return sum
9  }
10
11 func calculateSum513(s [][513]int64) int64 {
12         // Same implementation as calculateSum512
13 }
```

Iterate over each row 1
Iterate over the first eight columns 2
Increment sum 3

We iterate over each row, then over the first eight columns, and we increment a `sum` variable that we return. The implementation in `calculateSum513` remains the same.

We want to benchmark these functions to decide which one is the most performant given a fixed number of rows:

```
const rows = 1000

var res int64

func BenchmarkCalculateSum512(b *testing.B) {
        var sum int64
        s := createMatrix512(rows)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                sum = calculateSum512(s)
        }
        res = sum
}

func BenchmarkCalculateSum513(b *testing.B) {
        var sum int64
        s := createMatrix513(rows)
        b.ResetTimer()
        for i := 0; i < b.N; i++ {
                sum = calculateSum513(s)
        }
        res = sum
}
```

Create a matrix of 512 columns 1
Calculate the sum 2
Create a matrix of 513 columns 3

Calculate the sum 4

We want to create the matrix only once to limit the footprint on the results. Therefore, `createMatrix512` and `createMatrix513` are called outside of the loop. We may expect the results to be similar as again we only want to iterate on the first eight columns; however, this isn't the case (on my machine):

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkCalculateSum512-4        81854              15073
ns/op
BenchmarkCalculateSum513-4       161479               7358
ns/op
```

The second benchmark with 513 columns is about 50% faster. Again, as we iterate only over the first eight columns, this result can be quite surprising.

To understand this difference, we need to get the basics regarding CPU caches. In a nutshell, a CPU is composed of different caches (usually L1, L2, and L3). These caches are there to reduce the average cost to access data from the main memory. In some conditions, the CPU can fetch data from the main memory and copy it to L1. In this case, the CPU tries to fetch into L1 the matrix's subset `calculateSum` is interested in (the first eight columns of each row). However, it fits in memory in one case (513 columns), whereas in the other case (512 columns), it doesn't.

Note

> This isn't the scope of this chapter to explain why. However, we will delve into this problem in *Not understanding CPU caches*.

Coming back to the benchmark, the main issue is that we keep reusing the same matrix in both cases. As the function is repeated thousands of times, we don't measure the function execution receiving a plain new matrix. Instead, we measure a function that gets a matrix that already has a subset of the cells present in the cache. Therefore, as `calculateSum513` leads to fewer cache misses, it's has a better execution time.

This is an example of the observer effect. Because we keep observing a repeatedly called CPU-bound function, CPU caching may come into play and significantly affect the results.

In this example, to prevent this effect, we shouldn't reuse the matrix but create one during each test:

```
func BenchmarkCalculateSum512(b *testing.B) {
        var sum int64
        for i := 0; i < b.N; i++ {
                b.StopTimer()
                s := createMatrix512(rows)
                b.StartTimer()
                sum = calculateSum512(s)
        }
        res = sum
}
```

Create a new matrix during each loop iteration 1

A new matrix is now created during each loop iteration. If we run this benchmark again (and adjust `benchtime` otherwise, it takes too long to execute), the results are now closer to each other:

```
cpu: Intel(R) Core(TM) i5-7360U CPU @ 2.30GHz
BenchmarkCalculateSum512-4         1116              33547
ns/op
BenchmarkCalculateSum513-4          998              35507
ns/op
```

Therefore, instead of making wrong assumptions that `calculateSum513` is faster, we now see that both lead to similar results when receiving a new matrix.

As we have seen in this section, as we were reusing the same matrix, CPU caches significantly impacted the results. To prevent it, we had to create a new matrix during each loop iteration. In general, we should remember that observing a function under test may lead in some cases to significant differences in the results, especially in the context of micro-benchmarks of CPU-bound functions where low-level optimizations matter. Forcing a benchmark to recreate data during each iteration can be a good way to prevent this effect.

For the last section of this chapter, let’s delve into common tips regarding testing in Go.

11.9 Not exploring all the Go testing features

When it comes to writing tests, developers should know about Go’s specific testing features and options. Otherwise, the testing process can be

less accurate and even less efficient. The last section of this chapter will delve into different parts that can make us more comfortable while writing Go tests.

11.9.1 Code coverage

During the development process, it can be pretty handy to see visually which part of a code is covered by a test or not. We can have access to this information using the `coverprofile` flag:

```
$ go test -coverprofile=coverage.out ./...
```

This command creates a `coverage.out` file that we can then open using `go tool cover`:

```
$ go tool cover -html=coverage.out
```

This command will open the web browser and show the coverage for each line of code.

One thing to add, by default, the code coverage analyzes only the current package being tested. For example, if we have the following structure:

```
/myapp
  |_ foo
    |_ foo.go
    |_ foo_test.go
  |_ bar

```

```
7       |_ bar.go
        |_ bar_test.go
```

If some portion of `foo.go` is only tested in `bar_test.go`, by default, it won't be shown in the coverage report. To include it, we have to be in the `myapp` folder and use the `coverpkg` flag:

```
go test -coverpkg=./... -coverprofile=coverage.out ./...
```

Let's remember this feature to see the current code coverage and decide which parts would deserve more tests.

> **Note**
>
> We should remain cautious when it comes to chasing code coverage. Having 100% test coverage doesn't imply a bug-free application. Properly reasoning about what our tests cover is more important than any static threshold.

11.9.2 Testing from a different package

When writing unit tests, one approach can be to focus on behaviors instead of internals. Indeed, suppose we expose an API to clients. We may want our tests to focus on what's visible from the outside, not the implementation details. This way, if the implementation change (e.g., refactor a function into two), the tests will remain the same. Also, they can

be easier to understand as they show how our API is used. If we want to enforce this practice, we can do it using a different package.

In Go, all the files inside a folder should belong to the same package with only one exception: a test file can belong to a `_test` package.

For example, if the following source file belongs to the `counter` package:

Listing 11.4. counter.go

```
package counter

import "sync/atomic"

var count uint64

func Inc() uint64 {
        atomic.AddUint64(&count, 1)
        return count
}
```

The test files can either live in the same package or access internals such as the `count` variable. Or, it can live in a `counter_test` package like so:

Listing 11.5. counter_test.go

```
package counter_test

import (
        "testing"

        "myapp/counter"
)

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

```
        if counter.Inc() != 1 {
                t.Errorf("expected 1")
        }
}
```

In that case, the test is implemented in an external package and cannot access internals such as the `count` variable.

Using this practice, we can guarantee that a test won't use any unexported elements; hence, it will focus on testing the exposed behavior.

11.9.3 Utility functions

When writing tests, we can handle errors differently than our production code.

For example, let's say we want to test a function that takes as an argument a `Customer` struct. As the creation of a `Customer` will be reused, we will decide to create a specific `createCustomer` function for the sake of the tests. This function will return a possible error alongside a `Customer`:

```
func TestCustomer(t *testing.T) {
        customer, err := createCustomer("foo")
        if err != nil {
                t.Fatal(err)
        }
        // ...
}

func createCustomer(someArg string) (Customer, error) {
        // Create customer
        if err != nil {
                return Customer{}, err
        }
```

```
        return customer, nil
}
```

Create a customer and check for errors 1

We create a customer using the `createCustomer` utility function, and then we perform the rest of the test. However, in the context of testing functions, we can simplify error management. We can do it in passing the `*testing.T` variable to the utility function:

```
1  func TestCustomer(t *testing.T) {
2          customer := createCustomer(t, "foo")
3          // ...
4  }
5
6  func createCustomer(t *testing.T, someArg string) Customer
7  {
8          // Create customer
9          if err != nil {
10                 t.Fatal(err)
11         }
12         return customer
   }
```

Call the utility function and provide t 1
Fail the test directly if we can't create a customer 2

Instead of returning an error, `createCustomer` fails the test directly if it can't create a `Customer`. This way, it makes `TestCustomer` smaller to write and easier to read.

Let's remember this practice regarding error management and testing to improve our tests.

11.9.4 Setup and teardown

In some cases, we may have to prepare a testing environment. For example, in integration tests, spin up a specific Docker container and then stop it. Regarding setup and teardown functions, we may want to do it per test or per package. Fortunately, in Go, both are possible.

To do it per test, we can call the setup function as a preaction and the teardown function using defer:

```
func TestMySQLIntegration(t *testing.T) {
        setupMySQL()
        defer teardownMySQL()
        // ...
}
```

We should note that it's also possible to register a function to be executed at the end of a test. For example, let's assume the `TestMySQLIntegration` needs to call a `createConnection` to create the DB connection. If we want this function also to include the teardown part, we can use `t.Cleanup` to register a cleanup function:

```
func TestMySQLIntegration(t *testing.T) {
        // ...
        db := createConnection(t, "tcp(localhost:3306)/db")
        // ...
}

func createConnection(t *testing.T, dsn string) *sql.DB {
        db, err := sql.Open("mysql", dsn)
        if err != nil {
                t.FailNow()
        }
        t.Cleanup(
```

```
13                  func() {
14                          _ = db.Close()
15                  })
16          return db
17 }
```

Register a function to be executed at the end of the test 1

At the end of the test, the closure provided to `t.Cleanup` will be executed. It makes future unit tests easier to write as they won't be responsible for closing the `db` variable.

Note that we can register multiple cleanup functions. In that case, these functions will be executed just like if we were using `defer`: last in, first out.

Regarding setup and teardown per package, we have to use `TestMain` function. A simple implementation of `TestMain` is the following:

```
1 func TestMain(m *testing.M) {
2         os.Exit(m.Run())
3 }
```

This particular function accepts a `*testing.M` argument that exposes a single `Run` method to run all the tests. Therefore, we can surround this call with setup and teardown functions:

```
1 func TestMain(m *testing.M) {
2         setupMySQL()
3         code := m.Run()
```

```
        teardownMySQL()
        os.Exit(code)
}
```

Setup MySQL 1
Run the tests 2
Teardown MySQL 3

This code will spin up MySQL once before all the tests and then tear it down.

Using these practices to add setup and teardown functions, we can configure a complex environment for our tests.

11.10 Chapter summary

- Categorizing tests using build flags, environment variables, or the short mode makes the testing process more efficient. Indeed, we can create test categories using build flags or environment variables (e.g., unit vs. integration tests) and differentiate short from long-running tests to decide which kind of tests to execute.
- Enabling the `race` flag is highly recommended when writing concurrent applications. Indeed, it allows catching potential data races that can lead to software bugs.
- Using the `parallel` flag is an efficient way to speed up tests, especially long-running ones.
- Using the `shuffle` flag helps to ensure a test suite doesn't rely on wrong assumptions that could hide possible bugs.
- Table-driven tests are an efficient way to group a set of similar tests to prevent code duplication and make future updates easier to handle.
- Avoiding sleeps using synchronization makes a test less flaky and more robust. If synchronization isn't possible, we can consider a retry approach.
- Understanding how to deal with functions using the time API is another way to make a test less flaky. As we have seen, we can use standard techniques such as handling it as part of a hidden dependency or asking clients to provide it.
- Using the `httptest` package is a helpful way to deal with HTTP applications. Indeed, it provides a set of utilities to test both clients and servers.

- Using the `iotest` package can help us write `io.Reader` and test that our application is tolerant to errors.

- Regarding benchmarks:

    - Using the time methods allows preserving the accuracy of a benchmark.
    - Increasing `benchtime` or using tools such as `benchstat` can be helpful when dealing with micro-benchmarks.
    - The results of a micro-benchmark should be taken with care if the system that ends up running the application is different from the one running the micro-benchmark.
    - Making sure the function under test leads to a side effect prevents compiler optimizations from fooling the benchmark results.
    - Forcing a benchmark to recreate the data used by a CPU-bound function prevents the observer effect.
- Using code coverage with the `coverprofile` flag, we can quickly see which part of our code needs more attention.
- By placing unit tests in a different package, we can enforce writing tests that focus on an exposed behavior, not internals.
- Handling errors using the `*testing.T` variable instead of the classic `if err != nil` can help us make our code shorter and easier to read.

- Using setup and teardown functions allow us to configure a complex environment in the case of integration tests, for example.

[23] peter.bourgon.org/blog/2021/04/02/dont-use-build-tags-for-integration-tests.html

[24] pkg.go.dev/testing/iotest

[25] github.com/golang/go/issues/14813

12 Optimizations

This chapter covers

- Explaining the concept of mechanical sympathy and applying it to low levels such as CPU caches
- Understanding the core differences between heap and stack and how to reduce allocations
- Using standard Go diagnostics tooling
- Delving into how the Garbage Collector (GC) works
- Discussing the impacts of running Go inside of Docker and Kubernetes

Before delving into this chapter, one disclaimer; in most contexts, writing readable and clear code is better than an optimized but more complex code to understand. Indeed, optimization comes generally with a price, and we advocate the reader to follow this famous quote from Wes Dyer:

> Make it correct, make it clear, make it concise, make it fast, in that order.
>
> -- Wes Dyer

That being said, it doesn't necessarily mean that optimizing an application for speed and efficiency is necessarily something to prohibit. For example, we can try to identify code paths that need to be optimized because there's a need

for it such as making our customers happy or reducing our costs for example. Throughout this chapter, we will discuss common optimization techniques; some of them will be specific to Go, and some won't. Furthermore, we will also discuss methods to identify the bottlenecks so that we don't work blindly.

12.1 Not understanding CPU caches

Mechanical sympathy is a term coined by Jackie Steward, a three-time F1 world champion:

> You don't have to be an engineer to be a racing driver, but you do have to have mechanical sympathy.
>
> -- Sir Jackie Stewart

In a nutshell, when you understand how a system is designed to be used, be it an F1 car, an airplane, or a computer, you can align with the design to gain optimal performance. Throughout this section, we will delve into concrete examples of where a mechanical sympathy of how CPU caches work can help us in optimizing Go applications.

12.1.1 CPU architecture

First, let's understand the fundamentals of CPU architecture and why CPU caches are important. We will take as an example the Intel Core i5-7300.

Modern CPUs rely on caching to speed up memory access. In most cases, via three different caching levels: L1, L2, and L3. On the i5-7300, here's the size of these caches:

- L1: 64 KB
- L2: 256 KB
- L3: 4 MB

The i5-7300 has two physical cores but four logical cores in total (also called virtual cores or threads). Dividing a physical core into multiple logical cores is named in the Intel family as hyper-threading.

Here's an overview of the Intel Core i5-7300 (`Tn` stands for *thread n*):

Figure 12.1. Three level of caches, two physical cores, and four logical cores in total.

Each physical core (core 0 and core 1) is divided into two logical cores (thread 0 and thread 1). Regarding the L1 cache, it's split into two sub-caches: L1D for data and L1I for instructions (each of 32 KB). Indeed, caching isn't solely related to data. When a CPU executes an application, it can also cache some instructions with the same rationale: speed up the overall execution.

The closer a memory is to a logical core, the faster accesses will be[26]:

- L1: about one nanosecond
- L2: about four times slower than L1
- L3: about ten times slower than L1

The physical location of the CPU caches can also explain these differences. L1 and L2 are called on-die, meaning they belong to the same piece of silicon as the rest of the processor. Conversely, L3 is off-die, which partly explains the latency differences compared to L1 and L2.

Regarding main memory (or RAM), average accesses are between 50 and 100 times slower than L1. We can access up to 100 variables stored on L1 for the price of a single access to the main memory. Therefore, as Go developers, one avenue for improvement is to make sure our application leverage CPU caches.

12.1.2 Cache line

The concept of cache lines is crucial to understand. But before presenting what it is, let's understand why we need it.

When a specific memory location is accessed (e.g., reading a variable), it is likely that in the near future:

- The same location will be referenced again
- Nearby memory locations will be referenced

The former refers to temporal locality, and the latter refers to spatial locality. Both are part of a principle called locality of reference.

For example, let's have a look at the following function that computes the sum of an `int64` slice:

```
func sum(s []int64) int64 {
        var total int64
        length := len(s)
        for i := 0; i < length; i++ {
                total += s[i]
        }
        return total
}
```

In this example, temporal locality applies to multiple variables: `i`, `length`, and `total`. Indeed, throughout the iteration, we keep accessing these variables. Furthermore, spatial locality applies to code instructions and the slice, `s`. Indeed, as a slice is backed by an array allocated contiguously in memory, in this example, accessing `s[0]` means also accessing `s[1]`, `s[2]`, etc.

Temporal locality is part of why we need CPU caches: to speed up repeated accesses to the same variables. However, because of spatial locality, the CPU will copy what we call a cache line instead of copying a single variable from the main memory to a cache.

A cache line is a contiguous memory segment of a fixed size, usually 64 bytes (eight `int64` variables). Whenever a CPU decides to cache a memory block from the RAM, it will copy it to a cache line. Then, as memory is a hierarchy, when a CPU wants to access a specific memory location, it will first check in L1, then in L2, then in L3, and finally, if not present in these caches, in the main memory.

Let's illustrate the fetching of a memory block with a concrete example. We call the `sum` function with a slice of 16 `int64` elements for the first time. When `sum` accesses `s[0]`, this memory address won't be present in the cache yet. If the CPU decides to cache it (we will also discuss this decision in a future section), it will copy the whole memory block:

Figure 12.2. Accessing `s[0]` makes the CPU to copy the 0x000 memory block.

At first, accessing `s[0]` results in a cache miss because the address isn't present in the cache yet. This kind of miss is called a compulsory miss. However, if the CPU fetches the 0x000 memory block, accessing elements from 1 to 7 will result in a cache hit. Then, the same logic will apply when `sum` accesses `s[8]`:

Figure 12.3. Accessing `s[7]` makes the CPU to copy the 0x100 memory block.

Again, accessing `s8` results in a compulsory miss. Yet, if the `0x100` memory block is copied into a cache line, it will also speed up accesses for elements 9

to 15.

In the end, iterating over the 16 elements resulted in two compulsory cache misses and 14 cache hits.

> **Note**
>
> We may wonder about the exact strategy when a CPU copies a memory block. For example, will it copy a block to all the levels? To only L1? In this case, what about L2 and L3?
>
> We have to know that different strategies exist. Sometimes, caches are inclusive (e.g., L2 data are also present in L3), and sometimes caches are exclusive (e.g., L3 is called a victim cache as it contains only data evicted from L2).
>
> In general, these strategies are hidden by CPU vendors and not particularly useful to know. Therefore, let's not delve into these questions.

Let's discuss a concrete example to illustrate how fast CPU caches are. We will implement two functions that compute a total while iterating over a slice of `int64` elements. In one case, we will iterate over every two elements, whereas in the other case over every eight elements:

```
func sum2(s []int64) int64 {
        var total int64
        for i := 0; i < len(s); i+=2 {
                total += s[i]
        }
        return total
}

```

```
func sum8(s []int64) int64 {
        var total int64
        for i := 0; i < len(s); i += 8 {
                total += s[i]
        }
        return total
}
```

Iterate every two elements 1
Iterate every eight elements 2

Both functions are the same, except for the iteration. If we benchmark these two functions, our gut feeling may be that the second version will be about four times faster because we have to increment over four times fewer elements. However, running a benchmark shows us that `sum8` is only about 10% faster on my machine. Still faster, yes, but only 10%.

The reason is related to cache lines. We discussed that a cache line is usually 64 bytes, containing up to eight `int64` variables. Here, the running time of these loops is dominated by the memory accesses, not by the increment instruction. Three out of four accesses will result in a cache hit in the first case. Therefore, the execution time difference for these two functions isn't significant.

This example demonstrates why cache line matters and that it can be easy to get fooled by our gut feeling if we lack mechanical sympathy, in this case, how CPUs cache data.

Let's keep discussing locality of reference and see a concrete example of leveraging spatial locality.

12.1.3 Slice of structs vs. struct of slices

This section will discuss an example that will compare the execution time of two functions. The first one takes as an argument a slice of structs and sum all the `a` fields:

```
1  type Foo struct {
2          a int64
3          b int64
4  }
5
6  func sumFoo(foos []Foo) int64 {
7          var total int64
8          for i := 0; i < len(foos); i++ {
9                  total += foos[i].a
10         }
11         return total
12 }
```

Receive a slice of Foo 1
Iterate over each Foo and sum each a field 2

`sumFoo` receives a slice of `Foo` and increment `total` by reading each `a` field.

The second function will also compute a sum. Yet, this time the argument is a struct containing slices:

```
1  type Bar struct {
2          a []int64
3          b []int64
4  }
5
6  func sumBar(bar Bar) int64 {
7          var total int64
8          for i := 0; i < len(bar.a); i++ {
9                  total += bar.a[i]
```

```
        }
        return total
}
```

a and b are now slices 1
Receive a single struct 2
Iterate over each elements of a 3
Increment total 4

`sumBar` receives a single `Bar` struct that contains two slices: `a` and `b`. It iterates over each element of `a` to increment `total`.

Do we expect any difference in terms of speed for these two functions?

Before running a benchmark, let's visually see the differences in memory in [Figure 12.4. A struct of slices is more compact; therefore, requires less cache lines to iterate over.">figure 12.4](). In both cases, with the same amount of data: 16 `Foo` elements in the slice and 16 elements in the slices of `Bar`. Each black bar represents an `int64` that will be read to compute the sum, whereas each gray bar represents an `int64` that will be skipped.

Figure 12.4. A struct of slices is more compact; therefore, requires less cache lines to iterate over.

In the case of `sumFoo`, we receive a slice of structs containing two fields, `a` and `b`. Therefore, we will have a succession of `a` and `b` in memory. Conversely, in the case of `sumBar`, we receive a struct containing two slices, `a` and `b`. Therefore, all the elements of `a` will be allocated contiguously.

This difference doesn't lead to any memory compaction optimization. However, as the goal of both functions was to iterate over each `a`, in one case, it requires four cache lines and only two cache lines in the other case.

If we benchmark these two functions, `sumBar` will be faster (about 20% on my machine). The main reason is a better spatial locality that makes the CPU fetch fewer cache lines from memory.

This example demonstrates how spatial locality can have a substantial impact on performance. To optimize an application, we should organize data to get the most value out of each individual cache line.

However, is leveraging spatial locality enough to help the CPU out? We still miss one crucial characteristic: predictability.

12.1.4 Predictability

Predictability refers to the ability of a CPU to anticipate what the application will do to speed up its execution. Let's see a concrete example where a lack of predictability will negatively impact our application performance.

Again, let's look at two functions that sum a list of elements. The first one will iterate over a linked list and sum all the values:

```go
type node struct {
        value int64
        next  *node
}

func linkedList(n *node) int64 {
        var total int64
        for n != nil {
                total += n.value
                n = n.next
        }
        return total
}
```

Linked list data structure 1
Iterate over each node 2
Increment total 3

This function receives a linked list, iterates over it, and increments a total.

On the other side, let's take again the `sum2` function that we discussed previously that iterates over a slice, one element out of two:

```go
func sum2(s []int64) int64 {
        var total int64
        for i := 0; i < len(s); i+=2 {
                total += s[i]
        }
        return total
}
```

Iterate every two elements 1

Let's assume that the linked list is allocated contiguously, for example, by a single function, one time in a row. On a 64-bit architecture, a word is 64-bit long. Therefore, if we compare the two data structures the functions receive (linked list or slice), we would have the following (the darker bars represent the `int64` elements we use to increment the total):

Figure 12.5. Be it the linked list or a slice, in memory, both data structures are compacted in a similar manner.

In both examples, we face similar compaction. As a linked list is a succession of value and 64-bit pointer elements, we increment the sum using one element out of two. Meanwhile, the `sum2` example reads only one element out of two.

In both examples, the two data structures have the same spatial locality. Here, we may expect a similar execution time for these two functions. Yet, the second one iterating on the slice will be significantly faster (about 70% on my machine). So what's the reason?

To understand it, we have to delve into the concept of striding. Striding relates to how CPUs work through our data. There are three different types of strides:

- A unit stride is when all the values we want to access are allocated contiguously; for example, a slice of `int64` elements. This stride is predictable for a CPU and the most efficient as it requires a minimum number of cache lines to walk through the elements.
- A constant stride is still predictable for the CPU; for example, a slice that iterates over every two elements. However, it requires

more cache lines to walk through data, so it's less efficient than a unit stride.

- A non-unit stride is a stride the CPU can't predict to walk through data; for example, a linked list of a slice of pointers. As the CPU doesn't know whether data are allocated contiguously or not, it won't fetch any cache line.

Figure 12.6. The three types of stride.

For `sum2`, we face a constant stride. However, for the linked list, we face a non-unit stride. Even though we know ourselves that data were allocated contiguously, the CPU doesn't know about it. Therefore, it won't be able to predict how to walk through the linked list.

Because of a different stride and similar spatial locality, iterating over a linked list is significantly slower than a slice of values. We should generally favor unit strides over constant strides because of a better spatial locality. However, a non-unit stride, regardless of how data were allocated, cannot be predicted by the CPU, hence leading to negative performance impacts.

So far, we discussed that CPU caches were fast but significantly smaller than the main memory. Therefore, a CPU needs a strategy to fetch a memory block to a cache line. This policy is called cache placement policy and can significantly impact performance.

12.1.5 Cache placement policy

In *Writing inaccurate benchmarks*, we discussed an example with a matrix in which we had to compute the total sum of the first eight columns. Back then, we didn't explain why changing the overall number of columns impacted the benchmark results. It might sound counterintuitive: as we need to read only the first eight columns, why does changing the total number of columns impact the execution time? Let's understand it in this section.

As a reminder, the implementation was the following:

```go
func calculateSum512(s [][512]int64) int64 {
        var sum int64
        for i := 0; i < len(s); i++ {
                for j := 0; j < 8; j++ {
                        sum += s[i][j]
                }
        }
        return sum
}

func calculateSum513(s [][513]int64) int64 {
        // Same implementation as calculateSum512
}
```

Receive a matrix of 512 columns 1
Receive a matrix of 513 columns 2

We iterate over each row, summing the first eight columns each time. When these two functions are benchmarked each time a new matrix, we don't

observe any difference. However, if we keep reusing the same matrix, `calculateSum513` was about 50% faster on my machine. The reason lies in CPU caches and how a memory block is copied to a cache line. Let's delve into it to understand this difference.

When a CPU decides to copy a memory block and place it into the cache, it must follow a particular strategy. Assuming an L1D cache of 32KB and a cache line of 64 bytes, if a block was placed randomly into L1D, the CPU would have to iterate over 512 cache lines in the worst case to read a variable. This kind of cache is called fully associative.

To improve how fast an address can be accessed from a CPU cache, designers work on different policies regarding cache placement. Let's skip the history and discuss today's most widely used option: set-associative cache, which relies on cache partitioning.

For the sake of clarity with the following figures, we will work with a reduced version of the problem:

- We will assume an L1D cache of 512 bytes (8 cache lines)
- Meanwhile, the matrix will be composed of 4 rows and 32 columns on which we will keep reading only the first eight columns

Let's see how this matrix can be stored in memory. We will use the binary representation for the memory block addresses. Also, the gray blocks represent the first eight `int64` elements we want to iterate over. The remaining blocks will be the ones skipped during the iteration:

Figure 12.7. The matrix stored in memory, and an empty cache for the execution.

	Memory addresses
s[0][0] ... s[0][7]	0000000000000
s[0][8] ... s[0][15]	0001000000000
s[0][16] ... s[0][23]	0010000000000
s[0][24] ... s[0][31]	0011000000000
s[1][0] ... s[1][7]	0100000000000
s[1][8] ... s[1][15]	0101000000000
s[1][16] ... s[1][23]	0110000000000
s[1][24] ... s[1][31]	0111000000000
s[2][0] ... s[2][7]	1000000000000
s[2][8] ... s[2][15]	1001000000000
s[2][16] ... s[2][23]	1010000000000
s[2][24] ... s[2][31]	1011000000000
s[3][0] ... s[3][7]	1100000000000
s[3][8] ... s[3][15]	1101000000000
s[3][16] ... s[3][23]	1110000000000
s[3][24] ... s[3][31]	1111000000000

Cache

Each memory block contains 64 bytes; hence eight `int64` elements. The first memory block starts at 0x0000000000000, whereas the second one begins at 0001000000000 (512 in binary), and so on. We also represented the cache that can hold eight lines.

> **Note**
>
> We will see in *Not being aware of data alignment* that a slice doesn't necessarily start at the beginning of a block.

With the set-associate cache policy, a cache is partitioned into sets. We will assume the cache is 2-way set associative, meaning each set contains two lines. A memory block can belong to only one set, and the placement is

determined by its memory address. To understand it, we have to dissect the memory block address into three parts, block offset, set index, and tag bits:

- The block offset is based on the block size. Here a block size is 512 bytes, and 512 equals 2^9. Therefore the first 9 bits of the address represent the block offset (bo).
- The set index indicates to which set an address belongs. As the cache is 2-way set associative and contains eight lines, we have 8 / 2 = 4 sets. Furthermore, 4 equals 2^2; therefore, the two next bits represent the set index (si).
- The rest of the address are the tag bits (tb). In the previous figure, we represented an address using 13 bits for simplicity. To compute tb, we do 13 - bo - si. Here, it means the two remaining bits represent the tag bits.

Let's say the function starts and tries to read `s[0][0]`, which belongs to address 0000000000000. As this address isn't present in the cache yet, the CPU will calculate its set index and copy it to the corresponding cache set:

Figure 12.8. The memory address 0000000000000 is copied into set 0.

As discussed, nine bits represent the block offset: it's the minimum common prefix for each memory block address. Then, two bits represent the set index. With address 0000000000000, si equals 00. Hence, this memory block will be copied to set 0.

Then, when the function reads from `s[0][1]` to `s[0][7]`, the data will already be present in the cache. How will the CPU know about it? It will calculate the starting address of the memory block, compute the set index and the tag bits, then check whether 00 is already present in set 0.

Now, the function reads `s[0][8]`, and this address isn't cached yet. So the same operation occurs to copy the memory block 0100000000000:

Figure 12.9. The memory address 0100000000000 is copied into set 0.

This memory also has a set index equal to 00, so it also belongs to set 0. The cache line is copied to the next available line in set 0. Then, again, reading from `s[1][1]` to `s[1][7]` will result in cache hits.

Now things are starting to be interesting. The function reads `s[2][0]`, and this address isn't present in the cache. So, the same operation will be performed:

Figure 12.10. The memory address 1000000000000 replaces an existing cache line in set 0.

Here, the set index is again equal to 00. However, set 0 is already full, so what will the CPU do? Copy it to another set? No, it won't. The CPU will replace one of the existing cache lines to copy the memory block 1000000000000.

The cache replacement policy depends on the CPU, but usually, it's a pseudo-LRU policy (a real LRU would be too complex to handle). In this case, let's say it replaces our first cache line: 0000000000000.

This situation will be repeated when iterating on row three: the memory address 1100000000000 will also have a set index equal to 00, resulting in replacing an existing cache line again.

Now, let's say the benchmark executes again the function with a slice pointing to the same matrix starting at address 0000000000000. When the function

reads `s[0][0]`, the address won't be present in the cache. Indeed, this block was already replaced.

So instead of leveraging CPU caches from one execution to another, it will lead to more cache misses. This type of cache miss is called a conflict miss: a miss that wouldn't occur if the cache wasn't partitioned. All the variables we iterate belong to a memory block whose set index is 00. Therefore, we use only one cache set instead of having a decent distribution across all the cache.

Previously we discussed the concept of striding that we defined as how a CPU walks through our data. In this example, this stride is called a critical stride: one leading to accessing memory addresses with the same set index, hence stored to the same cache set.

Let's come back to our real-world example with the two functions `calculateSum512` and `calculateSum513`. The benchmark was executed on an 8-way set associative L1D cache of 32KB; hence 64 sets in total. As a cache line is 64 bytes, the critical stride equals 64 * 64 bytes = 4 KB. 4 KB of `int64` types represent 512 elements. Therefore, we reach a critical stride with a matrix of 512 columns, hence a poor caching distribution. Meanwhile, if the matrix contains 513 columns, it doesn't lead to a critical stride. This is why we observed such a massive difference between both benchmarks.

In summary, we have to be aware that modern caches are partitioned. Depending on the striding, we can reach a case where only one set is used, which may harm application performance and lead to many conflict misses. This particular kind of stride is called a critical stride. For performance-intensive applications, we should avoid them to get the most out of CPU caches.

Note

> Such an example can also highlight why the results of a micro-benchmark should be taken with care if executed on a different system than production. Indeed, if the production system has a different cache architecture, it may result in significant differences performance-wise.

Let's keep discussing the impact of CPU caching, but this time, we will see concrete impacts while writing concurrent code.

12.2 Writing concurrent code leading to false sharing

So far, we have discussed the fundamental concepts of CPU caching. Meanwhile, we have seen that some specific caches aren't shared among all the logical cores but are specific to a physical core (typically, L1 and L2). This specificity has some concrete impacts, such as concurrency and the concept of false sharing that can lead to a significant performance decrease. First, let's understand what false sharing is via a concrete example and then understand how to prevent it.

In this example, we will use two structs, `Input` and `Result`:

```
type Input struct {
        a int64
        b int64
}

type Result struct {
        sumA int64
        sumB int64
}
```

The goal will be to implement a `count` function that receives a slice of `Input` and compute:

- The sum of all the `Input.a` fields into `Result.sumA`
- The sum of all the `Input.b` fields into `Result.sumB`

For the sake of the example, we will implement a concurrent solution, one goroutine that computes `sumA` and another one `sumB`:

```
1  func count(inputs []Input) Result {
2          wg := sync.WaitGroup{}
3          wg.Add(2)
4
5          result := Result{}
6
7          go func() {
8                  for i := 0; i < len(inputs); i++ {
9                          result.sumA += inputs[i].a
10                 }
11                 wg.Done()
12         }()
13
14         go func() {
15                 for i := 0; i < len(inputs); i++ {
16                         result.sumB += inputs[i].b
17                 }
18                 wg.Done()
19         }()
20
21         wg.Wait()
22         return result
23 }
```

Initialize the result struct 1
Compute sumA 2
Compute sumB 3

We spin up two goroutines, one that iterates over each `a` field, whereas the other iterates over each `b` field. This example is perfectly fine from a concurrency perspective. For instance, it doesn't lead to any data race as each goroutine increments its own variable. However, this example illustrates the false sharing concept, degrading the expected performance.

Let's have a look at the main memory. As `sumA` and `sumB` are allocated contiguously, in most cases (7 out of 8), both variables will be allocated to the same memory block:

Figure 12.11. In this example, `sumA` and `sumB` are part of the same memory block.

Now, let's assume that the machine contains two cores. In most cases, eventually, we should have two threads scheduled on different cores. Therefore, if the CPU decides to copy this memory block to a cache line, it will be copied twice:

Figure 12.12. Each block will be copied to a cache line on both core 0 and core 1.

Both cache lines will be replicated as L1D (L1 data) is per core. Now, we recall that in our example, both goroutines will update their own variable, `sumA` on one side, and `sumB` on the other side:

Figure 12.13. Each goroutine update its own variable.

As these cache lines are replicated, one of the goals of the CPU is to guarantee cache coherency. For example, if one goroutine updates `sumA` and another reads `sumA` (after some synchronization), we expect our application to get the latest value.

However, our example isn't exactly this. Both goroutines are accessing their own variable, not a shared one. In this case, we might expect the CPU to know about it and understand that this isn't a conflict. Yet, this isn't the case.

The granularity that is tracked by the CPU when we write a variable present in a cache isn't the variable; it's the cache line.

When a cache line is shared across multiple cores with at least one goroutine being a writer, it will force to invalidate the whole cache line. Again, even if the updates are logically independent of each other (e.g., `sumA` and `sumB`). This is the problem of false sharing, and it leads to degrading performance.

> **Note**
>
> Internally, a CPU uses the MESI protocol to guarantee cache coherency. It allows tracking each cache line and marking it either Modified, Exclusive, Shared, Invalid (MESI).

One of the most important aspects to understand regarding memory and caching: sharing memory across cores doesn't exist; it's an illusion. Again, this comprehension derives from the fact we don't consider a machine as a black box, but instead, we try to have a mechanical sympathy with underlying levels.

So how do we solve false sharing? There are two main solutions.

The first solution is to keep the same approach but ensure that both `sumA` and `sumB` won't be part of the same cache line. For example, we can update the `Result` struct to add *padding* between both fields.

Padding is a technique to allocate extra memory. For example, as an `int64` requires an 8-byte allocation and that a cache line is 64-byte long, we need 64 - 8 = 56 bytes of padding:

```go
type Result struct {
        sumA int64
        _    [56]byte
        sumB int64
}
```

Padding 1

Here would be a possible allocation memory-wise:

Figure 12.14. `sumA` **and** `sumB` **are part of a different memory block.**

Using padding, `sumA` and `sumB` will always be part of a different memory block; hence, a different cache line.

If we benchmark both solutions (with and without padding), we will notice that the padding solution is significantly faster (about 40% on my machine). So, an important improvement because we have added padding between the two fields to prevent false sharing.

The second solution is to rework the complete structure of the algorithm. For example, instead of having both goroutines sharing the same struct, we could

make them communicate their local result via channels. Be it padding or communication, the result benchmark-wise is roughly the same.

In summary, we must remember that sharing memory across goroutines is an illusion in the lowest memory levels. We have seen that false sharing occurs when a cache line is shared across two cores, with at least one goroutine being a writer. If we need to optimize an application that relies on concurrency, we should check whether false sharing applies as it's a known application performance-degrading pattern. Meanwhile, let's remember that we can prevent false sharing with either padding or communication.

The following section will discuss how CPUs can execute instructions in parallel and how to leverage it.

12.3 Not taking into account instruction-level parallelism

Instruction-level parallelism is another factor that can lead to a significant impact on performance. Before defining this concept, let's delve into a concrete example and understand how to optimize it.

We will write a function receiving an array of two `int64` elements. This function will iterate a certain number of times (a constant), and during each iteration, it will:

- Increment the first element of the array
- Increment the second element of the array if the first element is even

Here's the Go version:

```go
const n = 1_000_000

func add(s [2]int64) [2]int64 {
        for i := 0; i < n; i++ {
                s[0]++
                if s[0]%2 == 0 {
                        s[1]++
                }
        }
        return s
}
```

Iterate n times 1
Increment s[0] 2
Increment s[1] if s[0] is even 3

The instructions executed within the loop are the following (let's remember that an increment requires both a read, then a write):

Figure 12.15. Three main steps: increment, check, increment.

The sequence is sequential. First, we increment `s[0]`, then before incrementing `s[1]` we need to read again `s[0]`.

Note

This sequence of instructions doesn't match the granularity of the assembly instructions. Yet, for clarity throughout this section, we

take a simplified view.

Let’s take a moment to discuss theory with the concept of instruction-level parallelism (ILP). Back a few decades ago, CPU designers stopped focusing solely on clock speed to improve CPU performance. They arrived with multiple optimizations, and one of them was the concept of ILP, which allows parallelizing the execution of a sequence of instructions. A processor that implements ILP within a single virtual core is called a superscalar processor.

For example, the following figure represents a CPU executing an application consisting of three instructions, `I1`, `I2`, and `I3`:

Figure 12.16. Despite being written sequentially, the three instructions are executed in parallel.

Executing a sequence of instructions requires different stages. In a nutshell, the CPU needs to decode the instructions and execute them. The execution is handled by the execution unit that will perform the different operations and calculations.

In this case, the CPU decided to execute the three instructions in parallel. Note that all the instructions don't necessarily complete in a single clock cycle. For example, an instruction that reads a value already present in a register will do; however, an instruction that reads an address that must be fetched from main memory may take dozens of clock cycles to complete.

Executed sequentially, this sequence of instructions would have taken the following time (the function `t(x)` denotes the time taken by the CPU to execute instruction `x`):

```
total time = t(I1) + t(I2) + t(I3)
```

Thanks to ILP, the total time is now the following:

```
total time = max(t(I1), t(I2), t(I3))
```

ILP looks magic theoretically. However, it leads to a few challenges called hazards.

For example, what if `I3` sets a variable to 42 but `I2` is a conditional instruction (e.g., `if foo == 1`)? In theory, it should prevent executing both `I2` and `I3` in parallel. This is called a control hazard (or branching hazard). In practice, CPU designers solved control hazards using branch prediction.

For example, a CPU can count that the condition was true 99 out of the last 100 times; therefore, it will execute both `I2` and `I3` in parallel. In case of a misprediction (`I2` happened to be false), the CPU will flush its current execution pipeline, ensuring it doesn't lead to any inconsistencies. This flush leads to a performance penalty of about 10 to 20 clock cycles.

Other types of hazards exist preventing from executing instructions in parallel. As Go developers, we can have an impact.

Let's consider the two following instructions that will update registers (temporary storage areas used to execute operations):

- `I1` adds the numbers in registers A and B to C
- `I2` adds the numbers in registers C and D to D

Because `I2` depends on the outcome of `I1` concerning the value of register C, both instructions cannot be executed simultaneously. `I1` must complete before `I2`. This is another kind of hazard called a data hazard.

In the previous figure, `I1`, `I2`, and `I3` were executed in parallel by the CPU. Therefore, there were no data hazards:

Figure 12.17. No data hazard between `I1`, `I2`, and `I3`.

Note

A couple of things to note regarding hazards:

- Another type of hazard exists called structural hazards. However, as Go developers, we can hardly impact it; hence, we will omit it in this section.
- Even for data hazard, CPU designers have imagined a trick called forwarding which basically bypass writing to a register. It doesn't aim to solve this problem but to alleviate the effects.

Now, we should have a decent understanding of ILP theory. Let's get back to our initial problem and focus on the content of the loop:

```
s[0]++
if s[0]%2 == 0 {
        s[1]++
}
```

As we discussed, data hazards prevent instructions from being executed simultaneously. Let's delve into the sequence of instructions, but this time, we will highlight the hazard between the instructions:

Figure 12.18. Showing the different hazard types between the instructions.

This sequence contains one control hazard because of the `if` statement. However, as discussed, it's the scope of the CPU to optimize it and predict what branch should be taken. Furthermore, we can notice multiple data hazards. As we discussed, data hazards prevent ILP from executing instructions in parallel. From an ILP standpoint, here will be the sequence of instructions:

Figure 12.19. Both increments are executed sequentially.

From an ILP standpoint, the only independent instructions are the `s[0]` check and the `s[1]` increment. Therefore, these two instruction sets can be executed in parallel thanks to branch prediction.

What about the increments? Can we improve our version to minimize the number of data hazards?

Let's write another version (`add2`) that introduces a temporary variable:

```
func add(s [2]int64) [2]int64 {
        for i := 0; i < n; i++ {
                s[0]++
                if s[0]%2 == 0 {
                        s[1]++
                }
        }
        return s
}

func add2(s [2]int64) [2]int64 {
        for i := 0; i < n; i++ {
                v := s[0]
                s[0] = v + 1
                if v%2 != 0 {
```

```
                        s[1]++
                }
        }
        return s
}
```

First version 1

Second version 2

Introduce a new variable to fix s[0] value 3

In this new version, we fix the value of `s[0]` to a new variable, `v`. Previously we were incrementing `s[0]` and checking whether it was even. To replicate this behavior, as `v` is based on `s[0]` before incrementing it, we now check whether `v` is odd.

Let's now compare the two versions in terms of hazards:

Figure 12.20. One main difference: the data hazard for the `v` check step.

We can notice the same number of steps. The significant difference is regarding the data hazards. Indeed the `s[0]` increment step and the check `v`

step now depend on the same instruction (`read s[0] into v`).

Why does it matter? Because it will allow the CPU to increase the level of parallelism:

Figure 12.21. In the second version, both increment steps can now be executed in parallel.

Despite the same number of steps, this second version increases how many steps can be executed in parallel: three parallel routes instead of two.

Meanwhile, the execution time should be optimized because the longest path has been reduced.

If we benchmark these two functions, we will notice a significant speedup improvement for the second version (about 20% on my machine), mainly because of ILP.

Let’s take a step back to the conclusion of this section. We discussed how modern CPUs leverage parallelism to optimize the execution time of a set of instructions. We also discussed one main hazard type that prevents executing instructions in parallel: data hazard. Furthermore, we have optimized a concrete Go example by reducing the number of data hazards to increase the number of instructions that can be executed in parallel.

Delving into how Go compiles our code into assembly and understanding how to leverage CPU optimizations such as ILP can be another avenue for improvement. Here, introducing a temporary variable resulted in a significant performance improvement. This example demonstrates how mechanical sympathy can help us optimize a Go application.

Let’s also remember to remain cautious about such micro-optimizations. Indeed, as the Go compiler keeps evolving throughout the versions, the generated assembly of an application can also change while bumping the Go version.

The following section will discuss the impacts of data alignment.

12.4 Not being aware of data alignment

Data alignment is a way to arrange how data are allocated to speed up memory accesses by the CPU. Not being aware of this concept can lead to

extra memory consumption and even degrading performance. This section will delve into this concept, describe where it applies, and the techniques to prevent under-optimized code.

To understand how data alignment work, let's first discuss what would happen without it.

Suppose we allocate two variables: an `int32` (32 bytes) and an `int64` (64 bytes).

```
var i int32
var j int64
```

Without data alignment, on a 64-bit architecture, these two variables could be allocated this way:

Figure 12.22. `j` is allocated on two words.

The `j` variable allocation could be spread in two words. If the CPU wanted to read `j`, it would lead to two memory accesses instead of a single one.

To prevent such a case, the memory address of a variable should be a multiple of its own size. This is the concept of data alignment. In Go, the guarantees in terms of alignment are the following:

- byte, uint8, int8: 1 byte
- uint16, int16: 2 bytes
- uint32, int32, float32: 4 bytes
- uint64, int64, floa64, complex64: 8 bytes
- complex128: 16 bytes

All these types are guaranteed to be aligned: an address is multiple of their size. For example, the address of any `int32` variable will be a multiple of 4.

Let's get back to the real world and see two different cases for `i` and `j` to be allocated in memory:

Figure 12.23. In both cases, `j` would be aligned to its own size.

In the first case, a 32-bit variable was allocated right before `i`. Therefore, `i` and `j` were allocated contiguously.

In the second case, this 32-bit variable wasn't allocated before `i` (e.g., it was a 64-bit variable); therefore, `i` was allocated at the beginning of a word. Regarding `j`, to respect data alignment (an address is a multiple of 64), it can be allocated alongside `i` but to the next multiple of 64. The gray box represents 32 bits of padding.

Let's now understand when padding can be an issue. We will consider the following struct containing three fields:

```
type Foo struct {
        b1 byte
        i  int64
        b2 byte
}
```

Considering that a `byte` type (one byte) and an `int64` (eight bytes), on a 64-bit architecture, the struct will be allocated in memory this way:

Figure 12.24. The struct occupies 24 bytes in total.

`b1` is allocated first. Then, as `i` is an `int64`, its address must be a multiple of eight. Therefore, it's impossible to allocate it alongside `b1` at 0x01. So, what's the next address being is a multiple of eight? 0x08. Regarding `b2`, the

variable is allocated to the next available address being a multiple on one: 0x10.

As a struct size has to be a multiple of the word size (8 bytes), its address isn't 17 bytes but 24 bytes in total. During the compilation, the Go compiler will add padding to guarantee data alignment:

```
type Foo struct {
        b1 byte
        _  [7]byte
        i  int64
        b2 byte
        _  [7]byte
}
```

Added by the compiler 1
Added by the compiler 2

Therefore, every time a `Foo` struct is created, it will require 24 bytes in memory, but only ten bytes will contain actual data. The remaining 14 bytes will be padding. Meanwhile, as a struct is an atomic unit, even after a GC, the struct won't be reorganized; it will always occupy 24 bytes in memory. Note that the compiler doesn't rearrange the fields; it only adds padding to guarantee data alignment.

So, how can we reduce the amount of memory allocated? The rule of thumb is to reorganize a struct so that its fields are sorted by type size in descending order. In our case, the `int64` type is the first followed by the two `byte` types:

```
type Foo struct {
        i  int64
        b1 byte

```

```
5         b2 byte
}
```

Now, here's how this new version of `Foo` will be allocated in memory:

Figure 12.25. The struct now occupies 16 bytes in memory.

`i` is allocated first and occupies a complete word. Then, the main difference is that now `b1` and `b2` can live alongside each other within the same word.

Again, the struct has to be a multiple of the word size, but instead of occupying 24 bytes in memory, it only occupies 16 bytes. Therefore, 33% of memory was saved just because we moved `i` at the first position.

What are the concrete impacts if we used the first version of the `Foo` struct (24 bytes) instead of the compacted one?

First, if the `Foo` are retained (e.g., an in-memory `Foo` cache), it would consume extra memory. However, even if the `Foo` aren't retained, it would also lead to other impacts. For example, if we frequently create `Foo` variables and if these variables are allocated to the heap (we will discuss this concept in the next section), it will lead to more frequent GCs, hence impacting the overall application performance.

Speaking about performance, there's another impact on spatial locality. For example, let's consider the `sum` function that would take a slice of `Foo` structs as an argument. This function would iterate over the slice and sum all the `i` fields (`int64`):

```
1 func sum(foos []Foo) int64 {
2         var s int64
3         for i := 0; i < len(foos); i++ {
4                 s += foos[i].i
5         }
6         return s
7 }
```

Sum all the i fields 1

As a slice is backed by an array, it means a contiguous allocation of `Foo` structs.

Let's delve into the backing array for the two versions of `Foo` and check two cache lines of data (128 bytes). Each gray bar represent 8 bytes of data, and the darker ones are the `i` variables (the fields we want to sum):

Figure 12.26. As each cache line contains more `i` variables, iterating over a slice of `Foo` will require fewer cache lines in total.

As we can see visually, with the latest version of `Foo`, each cache line will be more useful; in the sense that it will contain, on average, 33% more `i` variables. Therefore, iterating over a `Foo` slice to sum all the `int64` elements will be more efficient.

We can confirm this observation with a benchmark. If we run two benchmarks with the two versions of `Foo` using a slice of 10k elements, the version using the latest `Foo` struct is about 15% faster on my machine. 15% of speed improvement by changing the position of a single field in a struct.

Let's be mindful of data alignment. As we have seen throughout this section, reorganizing the fields of a Go struct to sort them per size in descending order is a way to prevent padding. Preventing padding means allocating more compact structs, leading to a few optimizations such as potentially reducing the frequency of GCs and a possible better spatial locality.

The following section will discuss the fundamental differences between stack and heap and understand why it matters.

12.5 Not understanding stack vs. heap

In Go, a variable can be allocated either on the stack or on the heap. These two types of memory are fundamentally different and can significantly impact data-intensive applications. Let's first delve into these concepts; then, we will see the rules followed by the compiler to decide where a variable should be allocated.

12.5.1 Stack vs. heap

First, let's discuss the differences between the stack and the heap.

The stack is the default memory, it's a LIFO data structure storing all the local variables for a specific goroutine. When a goroutine starts, it gets 2 KB of contiguous memory forming its stack space (this size has evolved in the past and could change again). However, this size isn't fixed at runtime and can grow and shrink as necessary (but it will always remain contiguous in memory, preserving data locality).

When Go enters a function, a stack frame is created, representing an interval in memory accessible only for the current function.

Let's delve into a concrete example to understand the concept of stack frames. Here, the `main` function will print the result of a `sumValue` function:

```
1  func main() {
2          a := 3
3          b := 2
4
5          c := sumValue(a, b)
6          println(c)
7  }
8
9  //go:noinline
10 func sumValue(x, y int) int {
11         z := x + y
12         return z
13 }
```

Call the sumValue function 1
Print the result 2
Disable inlining 3

Two things to note here. First, we use the `println` built-in function instead of `fmt.Println`; otherwise, it would force allocating the `c` variable on the heap. The second point, we disable inlining on the `sumValue` function;

otherwise, it wouldn't lead to a function call (we will discuss inlining in *Not relying on inlining*).

Let's have a look at the stack following `a` and `b` allocations:

Figure 12.27. `a` and `b` are allocated on the stack.

As we executed `main`, a stack frame for this function was created. The two variables `a` and `b` were allocated to the stack in this stack frame. All the variables stored are valid addresses, meaning they can be referenced and accessed.

Now, here's what happens if we enter into the `sumValue` function, up to the `return` statement:

Figure 12.28. Calling `sumValue` creates a new stack frame.

The Go runtime created a new stack frame as part of the current goroutine stack. `x` and `y` were allocated alongside `z` in the current stack frame.

The previous stack frame contains addresses that are still considered valid. We can't access `a` and `b` directly, but if we had a pointer on `a`, for example, it would be a valid one. We will discuss pointers in a short moment.

Let's move again until the last statement of the application, `println`. We exited the `sumValue` function, so what happens to its stack frame?

Figure 12.29. The `sumValue` stack frame was deleted and replaced by variables from `main`. In this example, `x` has been erased by `c` while `y` and `z` are still allocated in memory but unreachable.

The `sumValue` stack frame wasn't completely erased from memory. It means that when a function returns, Go doesn't take some time to deallocate the variables to reclaim free space. However, these previous variables can't be accessed anymore, and when new variables from the parent function get allocated to the stack, they will simply replace previous allocations. In a sense, a stack is self-cleaning; it doesn't require an additional mechanism such as a GC.

Now, let's make a slight change to understand the stack's limitations. Instead of returning an `int`, the function will return a pointer:

```
func main() {
        a := 3
        b := 2

        c := sumPtr(a, b)
        println(*c)
}

//go:noinline
func sumPtr(x, y int) *int {
        z := x + y
        return &z
}
```

Return a pointer 1

The `c` variable in `main` is now a `*int` type. Let's move directly to the last `println` statement, following the call to `sumPtr`. What would happen if `z` remained allocated on the stack (which, as we will understand, can't be the case):

Figure 12.30. The `c` variable references an address not valid anymore.

If `c` was referencing the address of the `z` variable and that `z` was allocated on the stack, it would be a major problem. This address wouldn't be valid anymore, but also, the stack frame of `main` keeps growing; it would erase the `z` variable. For that reason, the stack isn't enough, and we need another type of memory: the heap.

A memory heap is a pool of memory shared for all the goroutines:

Figure 12.31. Three goroutines having their own stack but sharing the heap.

Here, three goroutines `G1`, `G2`, and `G3` have their own stack. Meanwhile, they all share the same heap.

In the previous example, we have seen that the `z` variable couldn't live on the stack; therefore, it will be *escaped* to the heap. Indeed, if the compiler cannot prove that a variable *isn't* referenced after the function returns, the variable is allocated on the heap.

But why should we care in the end? What's the point of understanding the differences between stack and heap? Because there's a significant impact in terms of performance.

As we said, a stack is self-cleaning and is accessed by a single goroutine. Conversely, the heap has to be cleaned by an external system: the GC. Therefore, the more heap allocations are made, the more we pressure the GC. When the GC runs, it will use 25% of the available CPU capacity and potentially create milliseconds of "stop the world" latency.

Furthermore, we must understand that allocating on the stack is faster for the Go runtime as it's trivial: a pointer references the following memory address available. Conversely, allocating on the heap requires more effort to find the right place; hence, more time.

To illustrate these differences, let's benchmark `sumValue` and `sumPtr`:

```go
var globalValue int
var globalPtr *int

func BenchmarkSumValue(b *testing.B) {
        b.ReportAllocs()
        var local int
        for i := 0; i < b.N; i++ {
                local = sumValue(i, i)
        }
        globalValue = local
}

func BenchmarkSumPtr(b *testing.B) {
        b.ReportAllocs()
        var local *int
        for i := 0; i < b.N; i++ {
                local = sumPtr(i, i)
        }
        globalValue = *local
}
```

Report heap allocations 1
Sum by value 2
Report heap allocations 3
Sum by pointer 4

If we run these benchmarks (and still disable inlining), we will get the following results:

```
BenchmarkSumValue-4   992800992    1.261 ns/op   0 B/op   0 allocs/op
BenchmarkSumPtr-4     82829653     14.84 ns/op   8 B/op   1 allocs/op
```

As we can notice, `sumPtr` is about an order of magnitude slower than `sumValue`, which is the direct consequence of using the heap instead of the stack.

> **Note**
>
> This example shows that using pointers to avoid a copy isn't necessarily faster; it depends on the context.
>
> So far in this book, we only discussed values vs. pointers via the prism of semantics: using a pointer when a value has to be shared. In most cases, it should be the rule to follow.
>
> Bear also in mind that modern CPUs are extremely efficient in copying data, especially within the same cache line. Let's avoid premature optimization and focus on readability and semantics first.

Let's also note that in the previous benchmarks, we called `b.ReportAllocs()`, which highlights heap allocation (stack allocations aren't counted):

- `B/op: how many bytes per operation allocated
- `allocs/op: how many allocations per operation

Let’s now delve into the conditions for a variable to escape to the heap.

12.5.2 Escape analysis

Escape analysis refers to the work performed by the compiler to decide whether a variable should be allocated on the stack or the heap. Let’s understand the main rules.

First of all, when an allocation cannot be done on the stack, it will be done on the heap. Even though it sounds like a simplistic rule, it’s important to remember it. For example, if the compiler cannot prove that a variable isn’t referenced after a function returns, then this variable will be allocated on the heap. This was the case with the `sumPtr` function returning a pointer to a variable created in the function’s scope. In general, *sharing up* escapes to the heap.

But what about the opposite? What if we accept a pointer as in the following example:

```
1  func main() {
2          a := 3
3          b := 2
4          c := sum(&a, &b)
5          println(c)
6  }
7
8  //go:noinline
9  func sum(x, y *int) int {
10         return *x + *y
11 }
```

Accept pointers 1

`sum` accepts two pointers on variables created in the parent. If we move until the `return` statement in the `sum` function, here's the representation of the current stack:

Figure 12.32. The `x` and `y` variable reference valid addresses.

Despite being part of another stack frame, the `x` and `y` variables reference valid addresses. Therefore, `a` and `b` won't have to be escaped; they can stay on the stack. In general, *sharing down* stays on the stack.

What about other cases when a variable can be escaped to the heap?

- Global variables as multiple goroutines can access them.
- A pointer sent to a channel:

```
type Foo struct{ s string }
ch := make(chan *Foo, 1)
foo := &Foo{s: "x"}
ch <- foo
```

Here, `foo` will escape to the heap.

- A variable referenced by a value sent to a channel

```
type Foo struct{ s *string }
ch := make(chan Foo, 1)
s := "x"
bar := Foo{s: &s}
ch <- bar
```

As `s` is referenced by `Foo` via its address, it will escape to the heap.

- If a local variable is too large to fit on the stack.
- If the size of a local variable is unknown. For example, `s := make([]int, 10)` may not escape to heap but `s := make([]int, n)` will as its size is based on a variable.
- The backing array of a slice reallocated using `append`.

Though this list can give us ideas to understand compiler's decisions, it's not exhaustive and can also change in future Go versions. If we want to confirm an assumption, we can access compiler's decision using `-gcflags`:

```
$ go build -gcflags "-m=2"
...
./main.go:12:2: z escapes to heap:
```

Here, the compiler informs us that the `z` variable will escape to the heap.

Understanding the fundamental differences between heap and stack is crucial in optimizing a Go application. As we have seen, heap allocations are more complex to handle for the Go runtime and require an external system with the GC to deallocate data. Heap management can be responsible for up to 20 to 30% of the total CPU time consumed in some data-intensive applications. In the meantime, a stack is self-cleaning and local to a single goroutine making allocations faster. Therefore, optimizing memory allocation can have a great return on investment.

Furthermore, it's essential to understand the rules of escape analysis to write more efficient code. In general, sharing down stays on the stack, whereas sharing up escapes to the heap. This should prevent common mistakes such as premature optimizations where we may want to return pointers, for example, because "*it avoids a copy*". Let's focus on readability and semantics first and then optimize allocations if needed.

The following section will discuss how to reduce allocations.

12.6 Not knowing how to reduce allocations

Reducing allocations is a common optimization technique to speed up Go applications. Throughout this book, we already covered a few approaches that were reducing the number of heap allocations:

- Under-optimized strings concatenation section (*Under-optimized strings concatenation*): using `strings.Builder` instead of the `+` operator to concatenate strings.
- Useless string conversion (*Useless string conversion*): whenever possible, avoid having to convert `[]byte` into strings if possible

- Inefficient slice and map initialization (*Inefficient slice initialization* and *Inefficient map initialization*): preallocate slice and map if the length is already known
- Better data struct alignment to reduce struct size (*Not being aware of data alignment*)

As part of this section, we will delve into three common approaches:

- Changing our API
- Relying on compiler optimizations
- Using tools such as `sync.Pool`

12.6.1 API change

The first option is to work carefully on the API we provide. Let's take as a concrete example the `io.Reader` interface:

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
```

The `Read` method accepts a slice and returns the number of bytes read. Now, imagine if the `io.Reader` interface would have been designed the other way around: passing an `int` representing how many bytes have to be read and returning a slice:

```
type Reader interface {
        Read(n int) (p []byte, err error)
}
```

Semantically, there would be nothing wrong with it. However, the slice returned would automatically escape to the heap in this case. Indeed, we would be in the sharing up case described in the previous section.

The Go designers took the sharing down approach to prevent automatically escaping the slice to the heap. Therefore, it's up to the caller to provide a slice. It doesn't necessarily mean this slice won't be escaped. Indeed, the compiler may have decided that this slice cannot stay on the stack. However, it's up to the caller to handle it, not a constraint brought by calling the `Read` method.

Therefore, sometimes even a slight change on an API can positively impact allocations. When designing an API, let's remain aware of the escape analysis rules described in the previous section and use, if needed, `gcflags` to understand compiler's decisions.

12.6.2 Compiler optimizations

One of the goals of the Go compiler is to optimize our code if possible. Here's a concrete example regarding maps.

In Go, defining a map using a slice as a key type isn't possible. In some cases, especially in applications doing I/O, we can receive `[]byte` data that we would like to use as a key. Hence, we are obliged to transform it into a string first, and one could write the following code:

```
type cache struct {
        m map[string]int
}

func (c *cache) get(bytes []byte) (v int, contains bool) {
        key := string(bytes)
        v, contains = c.m[key]
```

```
        return
}
```

Holds a map of strings 1
Convert from []byte to string 2
Query the map using the string value 3

As the `get` function receives a `[]byte` slice, we convert it to a `key` string to query the map.

However, we should note that the Go compiler implements a specific optimization in the case we query the map using `string(bytes)`:

```
func (c *cache) get(bytes []byte) (v int, contains bool) {
        v, contains = c.m[string(bytes)]
        return
}
```

Query the map directly using string(bytes) 1

Despite being almost the same code (we call `string(bytes)` directly instead of passing a variable), in that case, the compiler will avoid doing this bytes to string conversion. Hence, the second version will be faster than the first one.

This example illustrates that two versions of a function that looks similar may result in different assembly code following the Go compiler's work. We should also be aware of the possible compiler's optimizations to optimize an application. Meanwhile, we should watch future Go releases to check whether new optimizations are added to the language.

12.6.3 `sync.Pool`

Another avenue for improvement if we want to tackle the number of allocations is to consider using `sync.Pool`.

First, we should understand that `sync.Pool` isn't a cache: there's no fixed size or maximum capacity that we can set. Instead, it's a pool to reuse common objects.

Let's imagine the following case. We want to implement a `write` function that receives an `io.Writer`, call a function to get a `[]byte` slice, and then write it to the `io.Writer`. Our code will look like this (we will omit error handling for the sake of clarity):

```
1 func write(w io.Writer) {
2         b := getResponse()
3         _, _ = w.Write(b)
4 }
```

Receive a []byte response 1
Write to the io.Writer 2

Here, `getResponse` returns a new `[]byte` slice upon each call. What if we would like to reduce the number of allocations by potentially reusing this slice? We will assume that all the responses have a max size of 1024 bytes. In this situation, we could use `sync.Pool`.

Creating a `sync.Pool` requires a `func() any` factory function; for example:

Figure 12.33. Define a factory function that will create a new object upon each call.

```
func factory() any {
    return 
}
```

`sync.Pool` exposes two methods:

- `Get() any`: to get an object from the pool
- `Put(any)`: to return an object to the pool

Using `Get` will either create a new object if the pool is empty or reuse one otherwise. Then, after having used the object, we can put it back to the pool using `Put`. Here's an example with the previously defined factory with a `Get` when the pool is empty, a `Put`, and a `Get` when the pool isn't empty:

Figure 12.34. `Get` either creates a new object or return one from the pool. `Put` returns the object back to the pool.

So one question, when will objects be drained from the pool? There's no specific method for it: it relies on the GC. Indeed, after each GC, objects from the pool will be destroyed.

If we come back to our example, assuming that we can update the `getResponse` function to write data to a given slice instead of creating one, we can now implement another version of the `write` method that would rely on a pool:

```
var pool = sync.Pool{
        New: func() any {
                return make([]byte, 1024)
        },
}

```

```go
func write(w io.Writer) {
        buffer := pool.Get().([]byte)
        buffer = buffer[:0]
        defer pool.Put(buffer)

        getResponse(buffer)
        _, _ = w.Write(buffer)
}
```

Create a pool and set the factory function 1
Get a []byte from the pool or create one 2
Reset the buffer 3
Put the buffer back to the pool 4
Write the response to the provided buffer 5

First, we define a new pool using the `sync.Pool` struct and set the factory function to create a new `[]byte` with a length of 1024 elements.

In the `write` function, we try to retrieve one buffer from the pool. If the pool is empty, it will create a new one; otherwise, it will select one arbitrary buffer from the pool and return it. One crucial step is to reset the buffer using `buffer[:0]` as this slice was potentially already used. Then, we defer the call to `Put` to put back the slice to the pool.

With this new version, calling `write` doesn't lead to creating a new `[]byte` slice for every single call. Instead, we can reuse existing allocated slices. In the worst-case scenario, for example, after a GC, it will create a new buffer; however, the amortized allocation cost is reduced.

In summary, if we frequently allocate many objects of the same type, we can consider using `sync.Pool`. Indeed, `sync.Pool` is a set of temporary objects that can help us prevent reallocating the same kind of data repeatedly. Also, `sync.Pool` is safe for use by multiple goroutines simultaneously.

Let's now discuss the concept of inlining to understand that it's not necessarily a compiler optimization we shouldn't care about.

12.7 Not relying on inlining

Inlining refers to replacing a function call by the body of this function. Nowadays, inlining is done automatically by compilers. Understanding the fundamentals of inlining can also be a way to optimize particular code paths of an application.

First, let's see a concrete example of inlining, with a simple `sum` function summing two `int` types:

```
1  func main() {
2          a := 3
3          b := 2
4          s := sum(a, b)
5          println(s)
6  }
7
8  func sum(a int, b int) int {
9          return a + b
10 }
```

Function will be inlined 1

If we run `go build` using `gcflags`, we will access the decision taken from the compiler regarding the `sum` function:

```
1 $ go build -gcflags "-m=2"
2 ./main.go:10:6: can inline sum with cost 4 as:
3     func(int, int) int { return a + b }
4 ...
5 ./main.go:6:10: inlining call to sum func(int, int) int {
  return a + b }
```

Here, the compiler decided to inline the call to `sum`. Hence, the previous code will be replaced by:

```
1 func main() {
2         a := 3
3         b := 2
4         s := a + b
5         println(s)
6 }
```

Replace the call to sum by its body 1

However, inlining only works for functions with a certain complexity, also known as inlining budget. Otherwise, the compiler would inform us that the function is too complex to be inlined:

```
1 ./main.go:10:6: cannot inline foo: function too complex:
2     cost 84 exceeds budget 80
```

Inlining has two main benefits. First, it removes the overhead of a function call (even though the overhead was mitigated since Go 1.17 and register-based calling conventions). Second, it allows the compiler to proceed to further optimizations. For example, following inlining a function, the compiler can decide that a variable supposed to escape initially on the heap may stay on the stack.

Now the question is: if this is an optimization applied automatically by the compiler, then why should we even care about it as Go developers? The answer lies in the concept of mid-stack inlining.

Mid stack inlining is about inlining functions that call other functions. Before Go 1.9, only leaf functions were considered for inlining. Now, thanks to mid stack inlining, the following `foo` function can also be inlined:

```
1 func main() {
2         foo()
3 }
4
5 func foo() {
6         x := 1
7         bar(x)
8 }
```

As the `foo` function isn't too complex, the compiler can inline its call:

```
1 func main() {
2         x := 1
3         bar(x)
4 }
```

Replace by the body of foo 1

Thanks to mid-stack inlining, as Go developers, we can now optimize an application using the concept of fast path inlining to distinguish between a fast and a slow path. Let's look at a concrete example released in the `sync.Mutex` implementation to understand how this concept works.

Before mid stack inlining, the implementation of the `Lock` method was the following:

```
1  func (m *Mutex) Lock() {
2          if atomic.CompareAndSwapInt32(&m.state, 0,
3  mutexLocked) {
```

```
4                   // Mutex isn't locked
5                   if race.Enabled {
6                           race.Acquire(unsafe.Pointer(m))
7                   }
8                   return
9           }
10
11          // Mutex is already locked
12          var waitStartTime int64
13          starving := false
14          awoke := false
15          iter := 0
16          old := m.state
17          for {
18                  // ...
19          }
20          if race.Enabled {
21                  race.Acquire(unsafe.Pointer(m))
22          }
    }
```

Complex logic 1

We can distinguish here two main paths:

- If the mutex isn't locked (`atomic.CompareAndSwapInt32` is true), fast path
- If the mutex is already locked (`atomic.CompareAndSwapInt32` is false), slow path

However, regardless of the path taken, the function cannot be inlined because of its complexity. Yet, to leverage mid stack inlining, the `Lock` method was refactored so that the slow path lives in a specific function:

```
1  func (m *Mutex) Lock() {
2          if atomic.CompareAndSwapInt32(&m.state, 0,
3  mutexLocked) {
4                  if race.Enabled {
5                          race.Acquire(unsafe.Pointer(m))
6                  }
```

```
7                   return
8           }
9           m.lockSlow()
10 }
11
12 func (m *Mutex) lockSlow() {
13          var waitStartTime int64
14          starving := false
15          awoke := false
16          iter := 0
17          old := m.state
18          for {
19                  // ...
20          }
21
22          if race.Enabled {
23                  race.Acquire(unsafe.Pointer(m))
24          }
   }
```

Mutex already locked path 1

Thanks to this change, the `Lock` method can now be inlined. The benefit of this approach is that locking a mutex that wasn't already locked is now done without paying the overhead of calling a function (it brings around 5% of speed improvement). Regarding the slow path, when the mutex was already locked, it didn't change. Previously it required one function call to execute this logic, and now it remains one function call, this time to `lockSlow`.

Therefore, this optimization technique is about distinguishing between a fast and a slow path. If a fast path can be inlined but not a slow one, we can extract the slow path inside a dedicated function. Hence, if the inlining budget isn't exceeded, our function will be a candidate for inlining.

Inlining isn't only an invisible compiler optimization we shouldn't care about. As seen in this section, understanding how inlining works and how to access the compiler's decision can be a road for optimization using the fast path

inlining technique. Indeed, extracting the slow path in a dedicated function prevents a function call if the fast path is executed.

The next section will delve into common diagnostics tooling to understand what should be optimized in our Go application.

12.8 Not using Go diagnostics tooling

Go offers a few excellent diagnostics tools to help us get insights into how an application performs. This section will focus on the most important ones: profiling and execution tracer. Both tools are so important that they should be part of any Go developer's core toolset interested in optimization.

Let's first delve into profiling.

12.8.1 Profiling

Profiling is the activity that provides insights into the execution of an application. It allows us to resolve performance issues, detect contention, locate memory leaks, and more. These insights can be collected via several profiles:

- CPU: determines where an application spends its time
- Goroutines: reports the stack traces of the ongoing goroutines
- Heap: reports heap memory allocation to monitor current memory usage and check for possible memory leaks
- Mutex: reports lock contentions to see the behaviors of the mutexes used in our code and whether an application spend too much time in locking calls
- Block: shows where goroutines block waiting on synchronization primitives

Profiling is achieved via instrumentation using a tool called a profiler; in Go: pprof. First, let's understand how and when to enable pprof; then, we will delve into the most critical profile types.

Enabling pprof

There are different ways to enable pprof. For example, we can use the `net/http/pprof` package to serve via HTTP the profiling data:

Listing 12.1. main.go

```
1  package main
2
3  import (
4          "fmt"
5          "log"
6          "net/http"
7          _ "net/http/pprof"
8  )
9
10 func main() {
11         http.HandleFunc("/", func(w http.ResponseWriter, r
12 *http.Request) {
13                 fmt.Fprintf(w, "")
14         })
15         log.Fatal(http.ListenAndServe(":80", nil))
   }
```

Blank import to pprof 1
Expose an HTTP endpoint 2

Importing `net/http/pprof` leads to a side effect allowing us to reach the pprof URL via *host/debug/pprof*. Note that enabling pprof is safe, even in production[27]. The profiles that impact performance, such as CPU profiling,

aren't enabled by default, nor do they run continuously: they are activated only for a specific period.

Now that we have seen how to expose a pprof endpoint, let's delve into the most common profiles.

CPU profiling

The CPU profiler relies on the OS and signaling. Indeed, when activated, the application will ask the OS to interrupt it every ten milliseconds by default via a `SIGPROF` signal. When the application receives a `SIGPROF`, it will suspend the current activity and transfer the execution to the profiler. The profiler will grab and collect data such as the current goroutines activity and aggregate execution statistics that we can retrieve. Then, it stops, and the execution resumes until the next `SIGPROF`.

We can access the `/debug/pprof/profile` endpoint to activate CPU profiling. Accessing this endpoint will execute CPU profiling for 30 seconds by default. It means that for 30 seconds, our application will be interrupted every ten milliseconds. Note that we can change these two default values. We can pass to the endpoint how long the profiling will last using the `second` parameter (e.g., `/debug/pprof/profile?seconds=15`). Meanwhile, we can also change the interruption rate, even under ten milliseconds. However, in most cases, ten milliseconds should be enough, and in decreasing this value (meaning increasing the rate), we should be careful not to harm performance. After 30 seconds, we will download the results of the CPU profiler.

> **Note**
>
> This command will produce the same type of file that can be downloaded via `/debug/pprof/profile`.

We can also enable the CPU profiler using the `-cpuprofile` flag, for example, when running a benchmark:

```
$ go test -bench=. -cpuprofile profile.out
```

From this file, we can navigate into the results using `go tool`:

```
$ go tool pprof -http=:8080 <file>
```

This command will open a web UI showing the call graph. Here's an example taken from one application to see how it looks like overall:

Figure 12.35. The call graph of an application during 30 seconds.

The more a box is red, the more it was a hot path. We can then navigate into this graph and get execution insights. For example:

Figure 12.36. Call graph example.

Here, this graph tells us that during 30 seconds, 0.06 seconds were spent in the `decode` method (`*FetchResponse` receiver). Among these 0.06 seconds, 0.02 where spent in `RecordBatch.decode` and 0.01 in `makemap` (create a map).

We can also access this kind of information from the web UI with different representations. For example, the `Top` view sorts the functions per execution time, and the `Flame graph` to visualize execution time hierarchy. The UI can even display the expensive parts of the source code line by line.

> **Note**
>
> We can also get profiling data via a command line. However, we will only focus on the web UI in this section.

Thanks to this data, we can also have a general idea about how an application behaves:

- If we observe too many calls to `runtime.mallogc`, it can mean an excessive amount of small heap allocation that we can try to minimize.
- If we observe too much time spent in channel operations or mutex locks, it can be an excessive contention harming application's performance.
- If we observe too much time spent `syscall.Read` or `syscall.Write`, the application spends a significant amount of time in Kernel mode. Perhaps, an avenue for improvement is to work on I/O buffering.

This is the kind of insight we can get from the CPU profiler. It brings a lot of value to understanding the hottest code path and identifying bottlenecks. However, note that it won't delve more than the rate configured as the CPU profiler is executed at a fixed pace (by default, ten milliseconds). To get finer-grained insights, we should use tracing in which we will delve into in a future section.

> **Note**
>
> We can also attach labels to the different functions. For example, imagine a common function called from different clients. If we want to track the time spent for both of these clients, we can use `pprof.Labels`.

Heap profiling

Heap profiling allows getting statistics about the current heap usage. Like CPU profiling, heap profiling is also sample-based. We can change this rate, but we should be careful about being too granular as the more we decrease the rate, the more effort the heap profiling will spend to collect data. By default, samples are profiled at one allocation for every 512 KB of heap allocation.

If we reach `/debug/pprof/heap/`, we will get raw data that can be hard to read. However, we can download a heap profile using `debug/pprof/heap/?debug=0` and then open it with `go tool` (same command) to navigate into the data using the web UI. For example:

Figure 12.37. A heap graph.

Here, we can observe an example of a heap graph. Calling the `MetadataResponse.decode` method leads to allocating 1536 KB of heap data (which represents 6.32% of the total heap). However, 0 out of these 1536 KB

was allocated by this function directly. So, we have to delve into the second call. Now, we can notice that the `TopicMetadata.decode` method allocated 512 KB out of the 1536 KB by itself. The rest, the 1024 KB, were allocated in another method.

This is how we can navigate the call chain to understand what part of an application is responsible for most of the heap allocations. We can also delve into different sample types:

- `alloc_objects`: the total amount of objects allocated
- `alloc_space`: the total amount of memory allocated
- `inuse_objects`: the amount of objects allocated and not yet released
- `inuse_space`: the amount of memory allocated and not yet released

Another part that is very helpful with heap profiling is to track memory leaks. With a GC-based language, the usual procedure is the following:

- Trigger a GC
- Download heap data
- Wait for a few seconds/minutes
- Trigger another GC
- Download another heap data
- Compare

Forcing a GC before downloading data is a way to prevent false assumptions. For example, if we see a peak of retained objects, without running a GC first, we cannot be sure whether it’s a leak or objects that the next GC will collect.

Using `pprof`, we can download a heap profile and force a GC in the meantime. Therefore, the procedure in Go is the following:

- Go to `/debug/pprof/heap?gc=1` (trigger GC and download heap profile)
- Wait for a few seconds/minutes
- Go to `/debug/pprof/heap?gc=1` again

- Using `go tool` to compare both heap profiles:

```
$ go tool pprof -http=:8080 -diff_base <file2> <file1>
```

Here's the kind of data that we can access:

Figure 12.38. Showing the differences between the two heap profiles.

For example, we can notice that the amount of heap memory held by the `newTopicProducer` method (top left) has decreased (-513 KB). In contrast, the amount held by `updateMetadata` (bottom right) has increased (+512 KB). Slow increases are normal. Indeed, the second heap profile can have been calculated in the middle of a service call, for example. However, we can repeat this process or wait for more time; the important part is to track steady increases regarding allocations of a specific object.

> **Note**
>
> There's another profiling related to the heap: `allocs` to report allocations. The heap profiling shows the current state of the heap memory. If we want to get insights about past memory allocations since the application has started, we can use allocations profiling. Note that, as discussed, because stack allocations are cheap, they aren't part of this profiling which only focuses on the heap.

Goroutines Profiling

The goroutine profile reports the stack trace of all the current goroutines in an application. We can download a file using `debug/pprof/goroutine/?debug=0` and use `go tool` again. Then, here's the kind of information we can get:

Figure 12.39. Goroutines graph.

We can see the current state of the application and how many goroutines were created per function. Here, `withRecover` has created 296 ongoing goroutines (63%), and 29 were related to a call to `responseFeeder`.

This kind of information is also beneficial if we suspect goroutine leaks. We can delve into goroutine profiler data to know which part of a system is the suspect.

Block profiling

The block profile reports where ongoing goroutines block waiting on synchronization primitives. This includes:

- Sending or receiving on an unbuffered channel
- Sending to a full channel
- Receiving from an empty channel
- Mutex contention
- Network or file system waits

It will also record the amount of time a goroutine has been waiting and is accessible via `debug/pprof/block`. This profile can be extremely helpful if we suspect that performance is harmed by blocking calls.

Note that the block profile isn't enabled by default, and we have to call `runtime.SetBlockProfileRate` to enable it. This function controls the fraction of goroutine blocking events reported. Once enabled, the profiler will keep collecting data in the background even if we don't call the `debug/pprof/block` endpoint. Let's be cautious if we want to set a high rate not to harm performance.

Mutex profiling

The last profile type is related to blocking but only regarding mutexes. If we suspect that our application spends a significant time waiting for locking mutexes, hence harming execution, we can use mutex profiling. It's accessible via `debug/pprof/mutex`.

Note this profile work in a similar manner as the blocking one. It's disabled by default, and we have to enable it using `runtime.SetMutexProfileFraction` that controls the fraction of mutex contention events reported.

A few additional notes about profiling:

- We haven't mentioned the threadcreate profile as it's been broken since 2013[28].
- Make sure to enable only one profiler at a time; for example, not CPU and heap profiling at the same time. Otherwise, it can lead to erroneous observations.

- pprof is extensible, and we can create our own custom profiles using `pprof.Profile`.

To conclude with profiling, we have seen the most important profiles that we can enable to help us understand how an application performs and the possible avenues for optimization. In general, it's recommended to enable pprof, even in production, as it offers an excellent balance in most cases. Some profiles, such as the CPU profile, lead to performance penalties but only during the time they are enabled.

Let's now delve into the execution tracer.

12.8.2 Execution tracer

The execution tracer is a tool capturing a wide range of runtime events to make them available for visualization with `go tool`. Tracer is helpful to:

- Understand some runtime events such as how GC performs
- Understand how goroutines execute
- Identify poorly parallelized execution

Let's try it with an example given in *Concurrency isn't always faster*. We discussed two parallel versions of the merge sort algorithm. The issue with the first version was a poor parallelization leading to creating too many goroutines. Let's see how tracer can help us in validating this statement.

Let's start by writing a benchmark on the first version and execute it with the `-trace` flag to enable the execution tracer:

```
$ go test -bench=. -v -trace=trace.out
```

> **Note**
>
> We can also download a remote trace file using the `debug/pprof/trace?debug=0` pprof endpoint.

This command creates a `trace.out` file that we can now open using `go tool`:

```
$ go tool trace trace.out
2021/11/26 21:36:03 Parsing trace...
2021/11/26 21:36:31 Splitting trace...
2021/11/26 21:37:00 Opening browser. Trace viewer is listening on
    http://127.0.0.1:54518
```

It opens the web browser, and now we can click on `View trace` to get all the traces during a specific timeframe:

Figure 12.40. Show goroutines activity and runtime events such as a GC phase.

Here, the figure represents about 150 milliseconds. We can observe multiple helpful metrics such as the goroutine count or the heap size. The heap size grows steadily until a point where a GC is triggered. We can also observe the

activity of the Go application per CPU core. The timeframe starts with user-level code, and then a stop-the-world is executed, which occupies the four CPU cores for approximately 40 milliseconds.

Regarding concurrency, we can observe that this version leverages all the available CPU cores on the machine. However, let's zoom inside a portion of one millisecond:

Figure 12.41. Too many small bars shows a poorly parallelized execution.

Each bar corresponds to a single goroutine execution. Having too many small bars doesn't look right. It means an execution that is poorly parallelized; we can zoom even closer to understand how these goroutines are orchestrated:

Figure 12.42. About 50% of the CPU time is spent to handle goroutine switches.

Here, we can observe that roughly 50% of the CPU time isn't spent executing the application code. The whitespaces represent the time taken by the Go runtime to spin up and orchestrate new goroutines.

Let’s now compare with the second parallel implementation that was about an order of magnitude faster, and zoom again to a one-millisecond timeframe:

Figure 12.43. The number of white spaces has been significantly reduced, proving that the CPU is more occupied.

We can observe that each goroutine takes more time to execute, and the number of white spaces has been significantly reduced. Hence, it means the CPU is way more occupied in executing application code than with the first version. Therefore, each millisecond of CPU time is spent more efficiently, explaining the benchmark differences.

Note that the granularity of the traces is per goroutine, not per function like the CPU profiling. However, it’s possible to define user-level tasks to get insights per function or group of functions using the `runtime/trace` package.

For example, imagine a function that computes a Fibonacci number and then writes it to a global variable using `atomic`; we could define two different tasks:

```
var v int64
ctx, fibTask := trace.NewTask(context.Background(),
```

```
"fibonacci")
trace.WithRegion(ctx, "main", func() {
        v = fibonacci(10)
})
fibTask.End()
ctx, fibStore := trace.NewTask(ctx, "store")
trace.WithRegion(ctx, "main", func() {
        atomic.StoreInt64(&result, v)
})
fibStore.End()
```

Create a fibonacci task 1

Create a store task 2

Using `go tool`, we can also get more precise information about how these two tasks perform. In the previous trace UI, we can observe the boundaries for each task per goroutine. Meanwhile, in `User-defined tasks`, we can follow the duration distribution:

Figure 12.44. Distribution of the user-level tasks.

Task type	Count	Duration distribution (complete tasks)	
fibonacci	10	10µs	1
		15µs	7
		25µs	0
		39µs	1
		63µs	1
		100µs	
store	10	2511ns	1
		3981ns	1
		6309ns	5
		10µs	0
		15µs	0
		25µs	1
		39µs	1
		63µs	1
		100µs	

Here, we can notice that in most cases, the `fibonacci` task was executed in less than 15 microseconds, whereas the `store` task in less than 6309 nanoseconds.

We already discussed in the previous section what kind of information we can get from CPU profiling. But what are the main differences with the data we can get from user-level traces?

- CPU profiling:
  - Sample-based
  - Per function
  - Doesn't go below the sampling rate (10 milliseconds by default)
- User-level traces:
  - Not sample-based
  - Per goroutine execution (unless we use the `runtime/trace` package)
  - Time executions aren't bound by any rate

In summary, the execution tracer is a powerful tool for understanding how an application performs. As we have seen with the merge sort example, we can identify poorly parallelized execution. However, its granularity remains per goroutine unless we manually use `runtime/trace` compared to a CPU profile, for example. We can use both profiling and the execution tracer to get the most out of the standard Go diagnostics tools when optimizing an application.

The next section will discuss how the GC works and how to tune it.

12.9 Not understanding how the GC works

The Garbage Collector (GC) is a critical piece of the Go language aiming to simplify the life of developers. It allows for tracking and free heap allocations that are no longer needed. As we can't replace every heap allocation with a stack allocation, understanding how the GC works should be part of the Go developer's toolset to optimize an application.

12.9.1 Concepts

A GC keeps a tree of the object references. The Go GC is based on the mark-and-sweep algorithm, which relies on two stages:

- Mark stage: traverse all the objects of the heap and mark them whether they are still in use
- Sweep stage: traverse the tree of references from the root and deallocate blocks of objects no longer referenced

When a GC runs, it first performs a set of different actions leading to *stopping the world* (two stop-the-world per GC, to be precise). Said differently, all the available CPU time will be used to perform the GC, putting our application code on hold. Following these steps, it starts the world again, resuming our application, but running in the meantime a concurrent phase. For that reason, the Go GC is called concurrent mark-and-sweep as it aims to reduce the amount of stop-the-world operations per GC cycle to mostly run concurrently alongside our application.

The Go GC also includes a way to free memory after a peak of consumption. Imagine our application is based on two phases:

- An init phase that would lead to frequent allocations and a large heap
- A runtime phase with moderate allocations and a small heap

How will Go tackle the fact that the large heap was only helpful when the application started but not after? It's handled as part of the GC with a so-called periodic scavenger. After a certain time, the GC will detect that such a large heap isn't necessary anymore. Hence, it will free some memory and return it to the OS.

> **Note**
>
> If the scavenger isn't quick enough, we can also manually force returning memory to the OS using `debug.FreeOSMemory()` .

So, the question that is important to understand is when will a GC cycle run? Indeed, compared to other languages such as Java, in Go, the configuration remains reasonably simple as it relies on a single environment variable: `GOGC` . This variable defines the percentage of the heap since the last GC before triggering another GC, by default: 100%.

One concrete example to make sure we understand it. Let's assume the current heap size is 128 MB. If `GOGC=100` , the next GC will be triggered when the heap size reaches 256 MB. Hence, a GC is triggered by default every time the heap size doubles. Also, if a GC hasn't been executed during the last two minutes, Go will force running one.

If we profile our application with production loads, we can try fine-tuning the `GOGC` :

- Reducing it will cause the heap to grow more slowly, increasing the pressure on the GC
- Conversely, bumping it will cause the heap to grow faster, reducing the pressure on the GC

Note

Enabling `gctrace` will write a trace to `stderr` each time the GC runs.

We can print the GC traces by setting the `GODEBUG` environment variable. For example, while running a benchmark:

```
$ GODEBUG=gctrace=1 go test -bench=. -v
```

Let's go through concrete examples to understand how the GC will behave in the event of a load increase.

12.9.2 Concrete examples

Let's imagine we expose some public services to users. During peak time at 12 AM, one million users will connect. However, the increase of the connected users will be done steadily.

In the following figure, we will represent the average heap size but also when will a GC be triggered if we keep `GOGC` to 100:

Figure 12.45. Steady increases of the connected users.

As `GOGC` is set to 100, the GC will be triggered every time the heap size doubles. In these conditions, as the number of users steadily increase, we should face an acceptable number of GC throughout the day:

Figure 12.46. GC frequency never reach a state above moderate.

We should have a moderate number of GC cycles at the beginning of the day. Until we reach 12 AM when the number of users starts to decrease, the number of GC cycles should also decrease steadily. In such a scenario, keeping `GOGC` to 100 should be perfectly fine.

Now, let's consider a second scenario where most of the one million users connect in less than an hour:

Figure 12.47. A sudden increase of users.

At 8 AM, the average heap size grows rapidly, reaching the peak in about one hour. In that situation, the frequency of the GC cycles should be heavily impacted during this hour:

Figure 12.48. During one hour, we observe a high frequency of GCs.

Because of the significant and sudden bump of the heap, we will face frequent GC cycles during a short period. Even though the Go GC is concurrent, it will lead to a significant number of stop-the-world periods during this period. And this situation can have some impacts, such as the average latency seen by the users.

In this situation, bumping `GOGC` to a higher value should probably be considered to reduce the pressure on the GC. Note that increasing `GOGC` doesn't lead to linear benefits. The bigger the heap, the longer it will take to clean it. Hence, using production load, configuring `GOGC` should be handled with care.

We can also note that in the case of exceptional conditions, with a bump that would be even more significant, tweaking `GOGC` may not be enough. For

example, let's say that instead of going from zero to one million users in one hour, we do it in a few seconds. During these seconds, the number of GC may reach a critical state leading the application to perform very poorly.

In such a condition, if the heap peak is known, there's one trick that we can apply. It involves forcing a large allocation of memory to improve the stability of the heap.

For example, we can force allocating 1 GB using a global variable in `main.go`:

```
var min = make([]byte, 1_000_000_000) // 1 GB
```

What's the point of such an allocation? If `GOGC` is kept to 100, instead of triggering a GC every time the heap doubles (which again will be extremely frequent during this small interval of a couple of seconds), Go will only trigger a GC when the heap reaches 2 GB. Therefore, it should reduce the number of GC cycles triggered when all the users connect, reducing the impact on the average latency during this period.

However, we could argue that when the heap size decreases, it will waste a lot of memory. Yet, this isn't the case. On most OS, the allocation of this `min` variable won't lead our application to necessarily consume 1 GB of memory. Indeed, calling `make` results in a system call to `mmap()`, which somehow leads to a lazy allocation. Indeed, for example, on Linux, memory is virtually addressed and mapped through page tables. Using `mmap()` will only allocate 1 GB of memory in the virtual address space, not the physical space. Only a read or a write will cause a page fault leading to an actual physical memory allocation.

> **Note**
>
> We can validate this behavior using tools such as `ps`.

Therefore, even if the application starts without any connected clients, it won't consume 1 GB of physical memory.

To conclude with GC, it's essential to understand how it behaves to optimize it. As Go developers, we can use `GOGC` to configure when the next GC cycle is triggered. In most cases, keeping it to 100 should be enough. However, if our application can face some peak of requests leading to frequent GC and latency impacts, we can increase this value. Last but not least, in the event of an exceptional peak of requests, using the trick to keep the virtual heap size to a minimum can be something to consider.

The last section of this chapter will discuss the impacts of running Go inside of Docker and Kubernetes.

12.10 Not understanding the impacts of running Go inside of Docker and Kubernetes

Using Go to writing services is the most common use according to the 2020 Go developer survey[29]. Meanwhile, Kubernetes is the most widely used platform to deploy these services. Hence, it's also important to understand the implications of running Go inside of Docker and Kubernetes to prevent common situations such as CPU throttling.

We mentioned in *Concurrency isn't always faster* that the `GOMAXPROCS` variable defines the limit of OS threads in charge of executing user-level code simultaneously. By default, it's set to the number of OS-apparent logical CPU cores. What does it mean in the context of Docker and Kubernetes?

Let’s assume that our Kubernetes cluster is composed of 8-core nodes. When a container is deployed in Kubernetes, we can define a CPU limit to make sure an application won’t consume all the host’s resources. For example, the following configuration limits the use of CPU to 4000 millicpu (or millicores), so four CPU cores:

```yaml
spec:
  containers:
  - name: myapp
    image: myapp
    resources:
      limits:
        cpu: 4000m
```

We may assume that when our application is deployed, `GOMAXPROCS` will be based on these limits; hence 4. Yet, it won’t be the case; it will be set to the number of logical cores on the host: 8. So, what’s the impact?

Kubernetes is using Completely Fair Scheduler (CFS) as a process scheduler. It’s also used to enforce CPU limits for Pod resources. When administrating a Kubernetes cluster, an administrator can configure these two parameters:

- `cpu.cfs_period_us` (global setting)
- `cpu.cfs_quota_us` (setting per Pod)

The former defines a period, whereas the latter a quota. By default, the period is set to 100 ms. Meanwhile, the default quota value will be how much CPU time the application can consume in 100 ms. The limit is set to four cores, which means 400 ms (4 * 100 ms).

Therefore, CFS will ensure that our application will never consume more than 400 ms of CPU time for a period of 100 ms.

Let's imagine a scenario where multiple goroutines are currently being executed on four different threads, and each thread is scheduled on a different core (1, 3, 4, and 8):

Figure 12.49. For every 100 ms period, the application consume less than 400 ms.

During the first period of 100 ms, four threads were busy, so we consumed 400 out of the 400 ms, hence 100% of the quota. Then, during the second period, 360 out of the 400 ms, etc. Everything is fine as the application consumes less than the quota.

However, let's remember that `GOMAXPROCS` is set to 8. Therefore, in the worst-case scenario, we can have eight threads and each of them scheduled on a different core:

Figure 12.50. Within each 100 ms period, the CPU is throttled after 50 ms.

Per portion of 100 ms, the quota is set to 400 ms. Therefore, if the eight threads are busy executing goroutines, after 50 ms, we will reach the quota of 400 ms (8 * 50 ms = 400 ms). What will be the consequence? CFS will throttle the CPU resource. Hence, no more CPU resources will be allocated until the start of another period. Said differently, our application will be on hold for 50 ms.

For example, a service with an average latency of 50 ms, can take up to 150 ms to complete. This is a possible 300% penalty on the latency.

So, what's the solution? First, keep an eye on the Go issue 33803[30]. Perhaps, in a future version of Go, `GOMAXPROCS` will be CFS-aware.

A solution for today is to rely on a library made by Uber called `automaxprocs`[31]. We can use this library by adding a blank import to `go.uber.org/automaxprocs` in `main.go`, and it will automatically set `GOMAXPROCS` to match Linux container CPU quota. In the previous example, `GOMAXPROCS` would have been set to 4 instead of 8; therefore, we wouldn't have been able to reach a state where the CPU is throttled.

In summary, let's remember that currently, Go isn't CFS-aware. Therefore, `GOMAXPROCS` isn't based on the CPU limits defined but on the host machine. Consequently, we can reach a state where the CPU is throttled, leading to long pauses and substantial impacts such as a significant latency increase. Until Go becomes CFS-aware, one solution can be to rely on `automaxprocs` to set `GOMAXPROCS` on the quota defined automatically.

12.11 Chapter summary

- Understanding how to leverage CPU caches is an important aspect to optimize CPU-bound applications as the L1 cache is about 50 to 100 times faster than the main memory.
- Being conscious of the cache line concept is critical to understanding how to organize data in data-intensive applications. A CPU doesn't fetch memory word by word; instead, it usually copies a memory block to a cache line of 64 bytes. To get the most out of each individual cache line, we should enforce spatial locality.
- Making our code predictable for the CPU can also be an efficient way to optimize certain functions. For example, a unit or constant stride is predictable for the CPU. Yet, a non-unit stride (e.g., linked list) isn't predictable.
- Being aware that caches are partitioned can help us avoid a critical stride; hence, utilizing only a tiny portion of the cache.
- Knowing that lower levels of CPU caches aren't shared across all the cores can allow us to avoid performance-degrading patterns such as false sharing while writing concurrency code. Sharing memory is an illusion.
- Leveraging instruction-level parallelism (ILP) can help us optimize specific parts of our code to allow a CPU to execute as many parallel instructions as possible. Identifying data hazards is one of the main steps.
- Understanding that in Go, basic types are aligned with their own size can help us to avoid common mistakes. For example, not knowing that reorganizing the fields of a struct per size in descending order can lead to more compact structs. Hence, less memory allocation and potentially a better spatial locality.

- Understanding the fundamental differences between heap and stack should also be part of our core knowledge when optimizing a Go application. Indeed, stack allocations are almost free, whereas heap allocations are slower and require an external system for cleaning with the GC.
- Reducing allocations is also an essential aspect of optimizing a Go application. It can be done via different axes. For example, designing API carefully to prevent sharing up, understanding the common Go compiler optimizations, or using `sync.Pool`.
- Using the fast path inlining technique can efficiently reduce the amortized time to call a function.
- Relying on profiling and the execution tracer are crucial steps to understand how an application performs and the parts to optimize.
- Understanding how to tune the GC can lead to multiple benefits such as handling more efficiently sudden load increases.
- Being aware that Go isn't CFS-aware can help us avoid CPU throttling when deployed in Docker and Kubernetes.

[26] colin-scott.github.io/personal_website/research/interactive_latency.html

[27] go.dev/doc/diagnostics#profiling

[28] github.com/golang/go/issues/6104

[29] go.dev/blog/survey2020-results

[30] github.com/golang/go/issues/33803

[31] github.com/uber-go/automaxprocs

Table of Contents

1. 1 Go: Simple to learn but hard to master
    1. This chapter covers
    2. 1.1 Go outline
    3. 1.2 Simple doesn’t mean easy
    4. 1.3 100 Go mistakes
        1. 1.3.1 Bugs
        2. 1.3.2 Needless complexity
        3. 1.3.3 Weaker readability
        4. 1.3.4 Unidiomatic organization
        5. 1.3.5 Lack of API convenience
        6. 1.3.6 Under-optimized code
        7. 1.3.7 Lack of productivity
    5. 1.4 Chapter summary
2. 2 Code and project organization
    1. This chapter covers
    2. 2.1 Unintended variable shadowing
    3. 2.2 Unnecessary nested code
    4. 2.3 Misusing init functions
        1. 2.3.1 Concepts
        2. 2.3.2 When to use init functions?
    5. 2.4 Overusing getters and setters
    6. 2.5 Interface pollution
        1. 2.5.1 Concepts
        2. 2.5.2 When to use interfaces?
        3. 2.5.3 Interface pollution
    7. 2.6 Interface on the producer side
    8. 2.7 Returning interfaces

9. 2.8 any says nothing[7]
10. 2.9 Being confused about when to use generics
    1. 2.9.1 Concepts
    2. 2.9.2 Common uses and misuses
11. 2.10 Not being aware of the possible problems with type embedding
12. 2.11 Not using the functional options pattern
    1. 2.11.1 Config struct
    2. 2.11.2 Builder pattern
    3. 2.11.3 Functional options pattern
13. 2.12 Project misorganization
    1. 2.12.1 Project structure
    2. 2.12.2 Package organization
14. 2.13 Creating utility packages
15. 2.14 Ignoring package name collisions
16. 2.15 Missing code documentation
17. 2.16 Not using linters
18. 2.17 Chapter summary

3. 3 Data types
    1. This chapter covers
    2. 3.1 Creating confusion with octal literals
    3. 3.2 Neglecting integer overflows
        1. 3.2.1 Concepts
        2. 3.2.2 Detect integer overflow during increment
        3. 3.2.3 Detect integer overflow during addition
        4. 3.2.4 Detect integer overflow during multiplication
    4. 3.3 Not understanding floating-points
    5. 3.4 Not understanding slice length and capacity
    6. 3.5 Inefficient slice initialization
    7. 3.6 Being confused about nil vs. empty slice

    8. 3.7 Not properly checking if a slice is empty
    9. 3.8 Not making slice copy correctly
    10. 3.9 Unexpected side-effects using slice append
    11. 3.10 Slice and memory leaks
        1. 3.10.1 Capacity leak
        2. 3.10.2 Slice and pointers
    12. 3.11 Inefficient map initialization
        1. 3.11.1 Concepts
        2. 3.11.2 Initialization
    13. 3.12 Map and memory leaks
    14. 3.13 Comparing values incorrectly
    15. 3.14 Chapter summary
4. 4 Control structures
    1. This chapter covers
    2. 4.1 Ignoring that elements are copied in range loops
        1. 4.1.1 Concepts
        2. 4.1.2 Value copy
    3. 4.2 Ignoring how arguments are evaluated in range loops
        1. 4.2.1 Channels
        2. 4.2.2 Array
    4. 4.3 Ignoring the impacts of using pointer elements in range loops
    5. 4.4 Making wrong assumptions during map iterations
        1. 4.4.1 Ordering
        2. 4.4.2 Map insert during iteration
    6. 4.5 Ignoring how the break statement work
    7. 4.6 Using defer inside a loop
    8. 4.7 Chapter summary
5. 5 Strings
    1. This chapter covers
    2. 5.1 Not understanding the concept of rune

    3. 5.2 Inaccurate string iteration
    4. 5.3 Misusing trim functions
    5. 5.4 Under-optimized strings concatenation
    6. 5.5 Useless string conversion
    7. 5.6 Substring and memory leaks
    8. 5.7 Chapter summary
6. 6 Functions and method
    1. This chapter covers
    2. 6.1 Not knowing which type of receiver to use
    3. 6.2 Never using named result parameters
    4. 6.3 Unintended side-effects with named result parameters
    5. 6.4 Returning a nil receiver
    6. 6.5 Using a filename as a function input
    7. 6.6 Ignoring how defer arguments and receivers are evaluated
        1. 6.6.1 Arguments evaluation
        2. 6.6.2 Pointer and value receiver
    8. 6.7 Chapter summary
7. 7 Error management
    1. This chapter covers
    2. 7.1 Panicking
    3. 7.2 Ignoring when to wrap an error
    4. 7.3 Comparing an error type inaccurately
    5. 7.4 Comparing an error value inaccurately
    6. 7.5 Handling an error twice
    7. 7.6 Not handling an error
    8. 7.7 Not handling defer errors
    9. 7.8 Chapter summary
8. 8 Concurrency: Foundations
    1. This chapter covers
    2. 8.1 Mixing concurrency and parallelism

    3. 8.2 Concurrency isn’t always faster
        1. 8.2.1 Go scheduling
        2. 8.2.2 Parallel Merge Sort
    4. 8.3 Being puzzled about when to use channels or mutexes
    5. 8.4 Not understanding race problems
        1. 8.4.1 Data races vs. race conditions
        2. 8.4.2 The Go memory model
    6. 8.5 Not understanding the concurrency impacts of a workload type
    7. 8.6 Misunderstanding Go contexts
        1. 8.6.1 Deadline
        2. 8.6.2 Cancellation Signal
        3. 8.6.3 Context Values
        4. 8.6.4 Catching a context cancellation
    8. 8.7 Chapter summary
9. 9 Concurrency: Practice
    1. This chapter covers
    2. 9.1 Propagating an inappropriate context
    3. 9.2 Starting a goroutine without knowing when to stop it
    4. 9.3 Not being careful with goroutines and loop variables
    5. 9.4 Expecting a deterministic behavior using select and channels
    6. 9.5 Not using notification channels
    7. 9.6 Not using nil channels
    8. 9.7 Being puzzled about a channel size
    9. 9.8 Forgetting about possible side-effects with string formatting
        1. 9.8.1 etcd data race
        2. 9.8.2 Dead-lock
    10. 9.9 Creating data races with append
    11. 9.10 Using mutexes inaccurately with slices and maps
    12. 9.11 Misusing sync.WaitGroup

    13. 9.12 Forgetting about sync.Cond
    14. 9.13 Not using errgroup
    15. 9.14 Copying a sync type
    16. 9.15 Chapter summary
10. 10 Standard library
    1. This chapter covers
    2. 10.1 Providing a wrong time duration
    3. 10.2 time.After and memory leak
    4. 10.3 JSON handling common mistakes
        1. 10.3.1 Unexpected behavior because of type embedding
        2. 10.3.2 JSON and monotonic clock
        3. 10.3.3 Map of any
    5. 10.4 SQL common mistakes
        1. 10.4.1 Forgetting that sql.Open doesn’t necessarily establish connections to a DB
        2. 10.4.2 Forgetting about connections pooling
        3. 10.4.3 Not using prepared statements
        4. 10.4.4 Mishandling null values
        5. 10.4.5 Not handling rows iteration errors
    6. 10.5 Not closing transient resources
        1. 10.5.1 HTTP body
        2. 10.5.2 sql.Rows
        3. 10.5.3 os.File
    7. 10.6 Forgetting the return statement after replying to an HTTP request
    8. 10.7 Using the default HTTP client and server
        1. 10.7.1 HTTP client
        2. 10.7.2 HTTP Server
    9. 10.8 Chapter summary
11. 11 Testing

1. This chapter covers
2. 11.1 Not categorizing tests
    1. 11.1.1 Build tags
    2. 11.1.2 Environment variable
    3. 11.1.3 Short mode
3. 11.2 Not enabling the race flag
4. 11.3 Not using test execution modes
    1. 11.3.1 Parallel
    2. 11.3.2 Shuffle
5. 11.4 Not using table-driven tests
6. 11.5 Sleeping in unit tests
7. 11.6 Not dealing with the time API efficiently
8. 11.7 Not using testing utility packages
    1. 11.7.1 httptest
    2. 11.7.2 iotest
9. 11.8 Writing inaccurate benchmarks
    1. 11.8.1 Not resetting or pausing the timer
    2. 11.8.2 Making wrong assumptions about micro-benchmarks
    3. 11.8.3 Not being careful about compiler optimizations
    4. 11.8.4 Being fooled by the observer effect
10. 11.9 Not exploring all the Go testing features
    1. 11.9.1 Code coverage
    2. 11.9.2 Testing from a different package
    3. 11.9.3 Utility functions
    4. 11.9.4 Setup and teardown
11. 11.10 Chapter summary

12. 12 Optimizations
    1. This chapter covers
    2. 12.1 Not understanding CPU caches
        1. 12.1.1 CPU architecture

    2. 12.1.2 Cache line
    3. 12.1.3 Slice of structs vs. struct of slices
    4. 12.1.4 Predictability
    5. 12.1.5 Cache placement policy
3. 12.2 Writing concurrent code leading to false sharing
4. 12.3 Not taking into account instruction-level parallelism
5. 12.4 Not being aware of data alignment
6. 12.5 Not understanding stack vs. heap
    1. 12.5.1 Stack vs. heap
    2. 12.5.2 Escape analysis
7. 12.6 Not knowing how to reduce allocations
    1. 12.6.1 API change
    2. 12.6.2 Compiler optimizations
    3. 12.6.3 sync.Pool
8. 12.7 Not relying on inlining
9. 12.8 Not using Go diagnostics tooling
    1. 12.8.1 Profiling
    2. 12.8.2 Execution tracer
10. 12.9 Not understanding how the GC works
    1. 12.9.1 Concepts
    2. 12.9.2 Concrete examples
11. 12.10 Not understanding the impacts of running Go inside of Docker and Kubernetes
12. 12.11 Chapter summary