CHAPTER 12
Concurrency in Go

Concurrency is the computer science term for breaking up a single process into independent components and specifying how these components safely share data. Most languages provide concurrency via a library using operating system–level threads that share data by attempting to acquire locks. Go is different. Its main concurrency model, arguably Go's most famous feature, is based on Communicating Sequential Processes (CSP). This style for concurrency was described in 1978 in a paper by Tony Hoare, the man who invented the Quicksort algorithm. The patterns implemented with CSP are just as powerful as the standard ones but are far easier to understand.

In this chapter, you are going to quickly review the features that are the backbone of concurrency in Go: goroutines, channels, and the `select` keyword. Then you are going to look at some common Go concurrency patterns and learn about the situations where lower-level techniques are a better approach.

When to Use Concurrency

Let's start with a word of caution. Be sure that your program benefits from concurrency. When new Go developers start experimenting with concurrency, they tend to go through a series of stages:

1. This is *amazing*; I'm going to put everything in goroutines!
2. My program isn't any faster. I'm adding buffers to my channels.
3. My channels are blocking and I'm getting deadlocks. I'm going to use buffered channels with *really* big buffers.

4. My channels are still blocking. I'm going to use mutexes.
5. Forget it, I'm giving up on concurrency.

People are attracted to concurrency because they believe concurrent programs run faster. Unfortunately, that's not always the case. More concurrency doesn't automatically make things faster, and it can make code harder to understand. The key is understanding that *concurrency is not parallelism*. Concurrency is a tool to better structure the problem you are trying to solve.

Whether concurrent code runs in parallel (at the same time) depends on the hardware and whether the algorithm allows it. In 1967, Gene Amdahl, one of the pioneers of computer science, derived Amdahl's law. It is a formula for figuring out how much parallel processing can improve performance, given how much of the work must be performed sequentially. If you want to dive into the details on Amdahl's law, you can learn more in *The Art of Concurrency* by Clay Breshears (O'Reilly). For our purposes, all you need to understand is that more concurrency does not mean more speed.

Broadly speaking, all programs follow the same three-step process: they take data, transform it, and then output the result. Whether you should use concurrency in your program depends on how data flows through the steps in your program. Sometimes two steps can be concurrent because the data from one is not required for the other to proceed, and at other times two steps must happen in series because one depends on the other's output. Use concurrency when you want to combine data from multiple operations that can operate independently.

Another important thing to note is that concurrency isn't worth using if the process that's running concurrently doesn't take a lot of time. Concurrency isn't free; many common in-memory algorithms are so fast that the overhead of passing values via concurrency overwhelms any potential time savings you'd gain by running concurrent code in parallel. This is why concurrent operations are often used for I/O; reading or writing to a disk or network is thousands of times slower than all but the most complicated in-memory processes. If you are not sure if concurrency will help, first write your code serially and then write a benchmark to compare performance with a concurrent implementation. (See "Using Benchmarks" on page 393 for information on how to benchmark your code.)

Let's consider an example. Say you are writing a web service that calls three other web services. Your program sends data to two of those services, and then takes the results of those two calls and sends them to the third, returning the result. The entire process must take less than 50 milliseconds, or an error should be returned. This is a good use of concurrency, because there are parts of the code that need to perform I/O that can run without interacting with one another, there's a part where the results are combined, and there's a limit on how long the code needs to run. At the end of this chapter, you'll see how to implement this code.

Goroutines

The goroutine is the core concept in Go's concurrency model. To understand goroutines, let's define a couple of terms. The first is *process*. A process is an instance of a program that's being run by a computer's operating system. The operating system associates some resources, such as memory, with the process and makes sure that other processes can't access them. A process is composed of one or more *threads*. A thread is a unit of execution that is given some time to run by the operating system. Threads within a process share access to resources. A CPU can execute instructions from one or more threads at the same time, depending on the number of cores. One of the jobs of an operating system is to schedule threads on the CPU to make sure that every process (and every thread within a process) gets a chance to run.

Think of a goroutine as a lightweight thread, managed by the Go runtime. When a Go program starts, the Go runtime creates a number of threads and launches a single goroutine to run your program. All the goroutines created by your program, including the initial one, are assigned to these threads automatically by the Go runtime scheduler, just as the operating system schedules threads across CPU cores. This might seem like extra work, since the underlying operating system already includes a scheduler that manages threads and processes, but it has several benefits:

- Goroutine creation is faster than thread creation, because you aren't creating an operating system–level resource.
- Goroutine initial stack sizes are smaller than thread stack sizes and can grow as needed. This makes goroutines more memory efficient.
- Switching between goroutines is faster than switching between threads because it happens entirely within the process, avoiding operating system calls that are (relatively) slow.
- The goroutine scheduler is able to optimize its decisions because it is part of the Go process. The scheduler works with the network poller, detecting when a goroutine can be unscheduled because it is blocking on I/O. It also integrates with the garbage collector, making sure that work is properly balanced across all the operating system threads assigned to your Go process.

These advantages allow Go programs to spawn hundreds, thousands, even tens of thousands of simultaneous goroutines. If you try to launch thousands of threads in a language with native threading, your program will slow to a crawl.

If you are interested in learning more about how the scheduler does its work, watch the talk Kavya Joshi gave at GopherCon 2018 called "The Scheduler Saga".

A goroutine is launched by placing the `go` keyword before a function invocation. Just as with any other function, you can pass it parameters to initialize its state. However, any values returned by the function are ignored.

Any function can be launched as a goroutine. This is different from JavaScript, where a function runs asynchronously only if the author of the function declared it with the `async` keyword. However, it is customary in Go to launch goroutines with a closure that wraps business logic. The closure takes care of the concurrent bookkeeping. The following sample code demonstrates the concept:

```go
func process(val int) int {
    // do something with val
}

func processConcurrently(inVals []int) []int {
    // create the channels
    in := make(chan int, 5)
    out := make(chan int, 5)
    // launch processing goroutines
    for i := 0; i < 5; i++ {
        go func() {
            for val := range in {
                out <- process(val)
            }
        }()
    }
    // load the data into the in channel in another goroutine
    // read the data from the out channel
    // return the data
}
```

In this code, the `processConcurrently` function creates a closure, which reads values out of a channel and passes them to the business logic in the `process` function. The `process` function is completely unaware that it is running in a goroutine. The result of `process` is then written back to a different channel by the closure. (I'll do a brief overview of channels in the next section.) This separation of responsibility makes your programs modular and testable, and keeps concurrency out of your APIs. The decision to use a thread-like model for concurrency means that Go programs avoid the "function coloring" problem described by Bob Nystrom in his famous blog post "What Color Is Your Function?"

You can find a complete example on The Go Playground or in the *sample_code/goroutine* directory in the Chapter 12 repository.

Channels

Goroutines communicate using *channels*. Like slices and maps, channels are a built-in type created using the `make` function:

```
ch := make(chan int)
```

Like maps, channels are reference types. When you pass a channel to a function, you are really passing a pointer to the channel. Also like maps and slices, the zero value for a channel is `nil`.

Reading, Writing, and Buffering

Use the `<-` operator to interact with a channel. You read from a channel by placing the `<-` operator to the left of the channel variable, and you write to a channel by placing it to the right:

```
a := <-ch // reads a value from ch and assigns it to a
ch <- b   // write the value in b to ch
```

Each value written to a channel can be read only once. If multiple goroutines are reading from the same channel, a value written to the channel will be read by only one of them.

A single goroutine rarely reads and writes to the same channel. When assigning a channel to a variable or field, or passing it to a function, use an arrow before the `chan` keyword (`ch <-chan int`) to indicate that the goroutine only *reads* from the channel. Use an arrow after the `chan` keyword (`ch chan<- int`) to indicate that the goroutine only *writes* to the channel. Doing so allows the Go compiler to ensure that a channel is only read from or written to by a function.

By default, channels are *unbuffered*. Every write to an open, unbuffered channel causes the writing goroutine to pause until another goroutine reads from the same channel. Likewise, a read from an open, unbuffered channel causes the reading goroutine to pause until another goroutine writes to the same channel. This means you cannot write to or read from an unbuffered channel without at least two concurrently running goroutines.

Go also has *buffered* channels. These channels buffer a limited number of writes without blocking. If the buffer fills before there are any reads from the channel, a subsequent write to the channel pauses the writing goroutine until the channel is read. Just as writing to a channel with a full buffer blocks, reading from a channel with an empty buffer also blocks.

A buffered channel is created by specifying the capacity of the buffer when creating the channel:

```
ch := make(chan int, 10)
```

The built-in functions `len` and `cap` return information about a buffered channel. Use `len` to find out how many values are currently in the buffer and use `cap` to find out the maximum buffer size. The capacity of the buffer cannot be changed.

> Passing an unbuffered channel to both `len` and `cap` returns 0. This makes sense because, by definition, an unbuffered channel doesn't have a buffer to store values.

Most of the time, you should use unbuffered channels. In "Know When to Use Buffered and Unbuffered Channels" on page 301, I'll talk about the situations where buffered channels are useful.

Using for-range and Channels

You can also read from a channel by using a `for-range` loop:

```
for v := range ch {
    fmt.Println(v)
}
```

Unlike other `for-range` loops, there is only a single variable declared for the channel, which is the value. If the channel is open and a value is available on the channel, it is assigned to `v` and the body of the loop executes. If no value is available on the channel, the goroutine pauses until a value is available or the channel is closed. The loop continues until the channel is closed, or until a `break` or `return` statement is reached.

Closing a Channel

When you're done writing to a channel, you close it using the built-in `close` function:

```
close(ch)
```

Once a channel is closed, any attempts to write to it or close it again will panic. Interestingly, attempting to read from a closed channel always succeeds. If the channel is buffered and some values haven't been read yet, they will be returned in order. If the channel is unbuffered or the buffered channel has no more values, the zero value for the channel's type is returned.

This leads to a question that might sound familiar from your experience with maps: when your code reads from a channel, how do you tell the difference between a zero value that was written and a zero value that was returned because the channel is closed? Since Go tries to be a consistent language, there is a familiar answer—use the comma ok idiom to detect whether a channel has been closed:

```
v, ok := <-ch
```

If `ok` is set to `true`, the channel is open. If it is set to `false`, the channel is closed.

Anytime you are reading from a channel that might be closed, use the comma ok idiom to ensure that the channel is still open.

The responsibility for closing a channel lies with the goroutine that writes to the channel. Be aware that closing a channel is required only if a goroutine is waiting for the channel to close (such as one using a `for-range` loop to read from the channel). Since a channel is just another variable, Go's runtime can detect channels that are no longer referenced and garbage collect them.

Channels are one of the two things that set apart Go's concurrency model. They guide you into thinking about your code as a series of stages and making data dependencies clear, which makes it easier to reason about concurrency. Other languages rely on global shared state to communicate between threads. This mutable shared state makes it hard to understand how data flows through a program, which in turn makes it difficult to understand whether two threads are actually independent.

Understanding How Channels Behave

Channels have many states, each with a different behavior when reading, writing, or closing. Use Table 12-1 to keep them straight.

Table 12-1. How channels behave

	Unbuffered, open	Unbuffered, closed	Buffered, open	Buffered, closed	Nil
Read	Pause until something is written	Return zero value (use comma ok to see if closed)	Pause if buffer is empty	Return a remaining value in the buffer; if the buffer is empty, return zero value (use comma ok to see if closed)	Hang forever
Write	Pause until something is read	**PANIC**	Pause if buffer is full	**PANIC**	Hang forever
Close	Works	**PANIC**	Works, remaining values still there	**PANIC**	**PANIC**

You must avoid situations that cause Go programs to panic. As mentioned earlier, the standard pattern is to make the writing goroutine responsible for closing the channel when there's nothing left to write. When multiple goroutines are writing to the same channel, this becomes more complicated, as calling `close` twice on the same channel causes a panic. Furthermore, if you close a channel in one goroutine, a write to the channel in another goroutine triggers a panic as well. The way to address this is to use a `sync.WaitGroup`. You'll see an example in "Use WaitGroups" on page 306.

A `nil` channel can be dangerous as well, but it is useful in some cases. You'll learn more about them in "Turn Off a case in a select" on page 304.

select

The `select` statement is the other thing that sets apart Go's concurrency model. It is the control structure for concurrency in Go, and it elegantly solves a common problem: if you can perform two concurrent operations, which one do you do first? You can't favor one operation over others, or you'll never process some cases. This is called *starvation*.

The `select` keyword allows a goroutine to read from or write to one of a set of multiple channels. It looks a great deal like a blank `switch` statement:

```go
select {
case v := <-ch:
    fmt.Println(v)
case v := <-ch2:
    fmt.Println(v)
case ch3 <- x:
    fmt.Println("wrote", x)
case <-ch4:
    fmt.Println("got value on ch4, but ignored it")
}
```

Each `case` in a `select` is a read or a write to a channel. If a read or write is possible for a `case`, it is executed along with the body of the `case`. Like a `switch`, each `case` in a `select` creates its own block.

What happens if multiple cases have channels that can be read or written? The `select` algorithm is simple: it picks randomly from any of its cases that can go forward; order is unimportant. This is very different from a `switch` statement, which always chooses the first `case` that resolves to `true`. It also cleanly resolves the starvation problem, as no `case` is favored over another and all are checked at the same time.

Another advantage of `select` choosing at random is that it prevents one of the most common causes of deadlocks: acquiring locks in an inconsistent order. If you have two goroutines that both access the same two channels, they must be accessed in the same order in both goroutines, or they will *deadlock*. This means that neither one can proceed because they are waiting on each other. If every goroutine in your Go application is deadlocked, the Go runtime kills your program (see Example 12-1).

Example 12-1. Deadlocking goroutines

```go
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func() {
        inGoroutine := 1
        ch1 <- inGoroutine
        fromMain := <-ch2
        fmt.Println("goroutine:", inGoroutine, fromMain)
    }()
    inMain := 2
    ch2 <- inMain
    fromGoroutine := <-ch1
    fmt.Println("main:", inMain, fromGoroutine)
}
```

If you run this program on The Go Playground or in the *sample_code/deadlock* directory in the Chapter 12 repository, you'll see the following error:

```
fatal error: all goroutines are asleep - deadlock!
```

Remember that `main` is running on a goroutine that is launched at startup by the Go runtime. The goroutine that is explicitly launched cannot proceed until `ch1` is read, and the main goroutine cannot proceed until `ch2` is read.

If the channel read and the channel write in the main goroutine are wrapped in a `select`, deadlock is avoided (see Example 12-2).

Example 12-2. Using `select` to avoid deadlocks

```go
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    go func() {
        inGoroutine := 1
        ch1 <- inGoroutine
        fromMain := <-ch2
        fmt.Println("goroutine:", inGoroutine, fromMain)
    }()
    inMain := 2
    var fromGoroutine int
    select {
    case ch2 <- inMain:
    case fromGoroutine = <-ch1:
    }
    fmt.Println("main:", inMain, fromGoroutine)
}
```

If you run this program on The Go Playground or in the *sample_code/select* directory in the Chapter 12 repository, you'll get the output:

```
main: 2 1
```

Because a `select` checks whether any of its cases can proceed, the deadlock is avoided. The goroutine that is launched explicitly wrote the value 1 into `ch1`, so the read from `ch1` into `fromGoroutine` in the main goroutine is able to succeed.

Although this program doesn't deadlock, it still doesn't do the right thing. The `fmt.Println` statement in the launched goroutine never executes, because that goroutine is paused, waiting for a value to read from `ch2`. When the main goroutine exits, the program exits and kills any remaining goroutines, which does technically resolve the pause. However, you should make sure that all your goroutines exit properly so that you don't *leak* them. I talk about this in more detail in "Always Clean Up Your Goroutines" on page 299.

> Making this program behave properly requires a few techniques that you'll learn about later in the chapter. You can find a working solution on The Go Playground.

Since `select` is responsible for communicating over a number of channels, it is often embedded within a `for` loop:

```
for {
    select {
    case <-done:
        return
    case v := <-ch:
        fmt.Println(v)
    }
}
```

This is so common that the combination is often referred to as a `for-select` loop. When using a `for-select` loop, you must include a way to exit the loop. You'll see one way to do this in "Use the Context to Terminate Goroutines" on page 300.

Just like `switch` statements, a `select` statement can have a `default` clause. Also just like `switch`, `default` is selected when there are no cases with channels that can be read or written. If you want to implement a nonblocking read or write on a channel, use a `select` with a `default`. The following code does not wait if there's no value to read in `ch`; it immediately executes the body of the `default`:

```
select {
case v := <-ch:
    fmt.Println("read from ch:", v)
default:
    fmt.Println("no value written to ch")
}
```

You'll take a look at a use for `default` in "Implement Backpressure" on page 302.

Having a `default` case inside a `for-select` loop is almost always the wrong thing to do. It will be triggered every time through the loop when there's nothing to read or write for any of the cases. This makes your `for` loop run constantly, which uses a great deal of CPU.

Concurrency Practices and Patterns

Now that you've seen the basic tools that Go provides for concurrency, let's take a look at some concurrency best practices and patterns.

Keep Your APIs Concurrency-Free

Concurrency is an implementation detail, and good API design should hide implementation details as much as possible. This allows you to change how your code works without changing how your code is invoked.

Practically, this means that you should never expose channels or mutexes in your API's types, functions, and methods (I'll talk about mutexes in "When to Use Mutexes Instead of Channels" on page 313). If you expose a channel, you put the responsibility of channel management on the users of your API. The users then have to worry about concerns like whether a channel is buffered or closed or `nil`. They can also trigger deadlocks by accessing channels or mutexes in an unexpected order.

This doesn't mean that you shouldn't ever have channels as function parameters or struct fields. It means that they shouldn't be exported.

This rule has some exceptions. If your API is a library with a concurrency helper function, channels are going to be part of its API.

Goroutines, for Loops, and Varying Variables

Most of the time, the closure that you use to launch a goroutine has no parameters. Instead, it captures values from the environment where it was declared. Before Go 1.22, there was one common situation where this didn't work: when trying to capture the index or value of a `for` loop. As mentioned in "The for-range value is a copy" on page 81 and "Using go.mod" on page 224, a backward-breaking change was introduced in Go 1.22 that changed the behavior of a `for` loop so that it creates new variables for the index and value on each iteration instead of reusing a single variable.

The following code demonstrates the reason this change was worthwhile. You can find it in the `goroutine_for_loop` repository in the Learning Go 2nd Edition organization on GitHub.

If you run the following code on Go 1.21 or earlier (or on Go 1.22 or later with the Go version set to 1.21 or earlier in the `go` directive in the *go.mod* file), you'll see a subtle bug:

```go
func main() {
    a := []int{2, 4, 6, 8, 10}
    ch := make(chan int, len(a))
    for _, v := range a {
        go func() {
            ch <- v * 2
        }()
    }
    for i := 0; i < len(a); i++ {
        fmt.Println(<-ch)
    }
}
```

One goroutine is launched for each value in `a`. It looks like a different value is passed in to each goroutine, but running the code shows something different:

```
20
20
20
20
20
```

The reason every goroutine wrote `20` to `ch` on earlier versions of Go is that the closure for every goroutine captured the same variable. The index and value variables in a `for` loop were reused on each iteration. The last value assigned to `v` was `10`. When the goroutines run, that's the value that they see.

Upgrading to Go 1.22 or later and changing the value of the `go` directive in *go.mod* to 1.22 or later changes the behavior of `for` loops so they create a new index and value variable on each iteration. This gives you the expected result, with a different value passed to each goroutine:

```
20
8
4
12
16
```

If you cannot upgrade to Go 1.22, you can resolve this issue in two ways. The first is to make a copy of the value by shadowing the value within the loop:

```go
for _, v := range a {
    v := v
    go func() {
        ch <- v * 2
    }()
}
```

If you want to avoid shadowing and make the data flow more obvious, you can also pass the value as a parameter to the goroutine:

```go
for _, v := range a {
    go func(val int) {
        ch <- val * 2
    }(v)
}
```

While Go 1.22 prevents this issue for the index and value variables in `for` loops, you still need to be careful with other variables that are captured by closures. Anytime a closure depends on a variable whose value might change, whether or not it is used as a goroutine, you must pass the value into the closure or make sure a unique copy of the variable is created for each closure that refers to the variable.

Anytime a closure uses a variable whose value might change, use a parameter to pass a copy of the variable's current value into the closure.

Always Clean Up Your Goroutines

Whenever you launch a goroutine function, you must make sure that it will eventually exit. Unlike variables, the Go runtime can't detect that a goroutine will never be used again. If a goroutine doesn't exit, all the memory allocated for variables on its stack remains allocated and any memory on the heap that is rooted in the goroutine's stack variables cannot be garbage collected. This is called a *goroutine leak*.

It may not be obvious that a goroutine isn't guaranteed to exit. For example, say you used a goroutine as a generator:

```
func countTo(max int) <-chan int {
    ch := make(chan int)
    go func() {
        for i := 0; i < max; i++ {
            ch <- i
        }
        close(ch)
    }()
    return ch
}

func main() {
    for i := range countTo(10) {
        fmt.Println(i)
    }
}
```

This is just a short example; don't use a goroutine to generate a list of numbers. It's too simple of an operation, which violates one of our "when to use concurrency" guidelines.

In the common case, where you use all the values, the goroutine exits. However, if you exit the loop early, the goroutine blocks forever, waiting for a value to be read from the channel:

```
func main() {
    for i := range countTo(10) {
        if i > 5 {
            break
        }
        fmt.Println(i)
    }
}
```

Use the Context to Terminate Goroutines

To solve the `countTo` goroutine leak, you need a way to tell the goroutine that it's time to stop processing. You solve this in Go by using a *context*. Here's a rewrite of `countTo` to demonstrate this technique. You can find the code in the *sample_code/context_cancel* directory in the Chapter 12 repository:

```
func countTo(ctx context.Context, max int) <-chan int {
    ch := make(chan int)
    go func() {
        defer close(ch)
        for i := 0; i < max; i++ {
            select {
            case <-ctx.Done():
                return
            case ch <- i:
            }
        }
    }()
    return ch
}

func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()
    ch := countTo(ctx, 10)
    for i := range ch {
        if i > 5 {
            break
        }
        fmt.Println(i)
    }
}
```

The `countTo` function is modified to take a `context.Context` parameter in addition to `max`. The `for` loop in the goroutine is also changed. It is now a `for-select` loop with two cases. One tries to write to `ch`. The other case checks the channel returned by the `Done` method on the context. If it returns a value, you exit the `for-select` loop and the goroutine. Now, you have a way to prevent the goroutine from leaking when every value is read.

This leads to the question, how do you get the `Done` channel to return a value? It is triggered via *context cancellation*. In `main`, you create a context and a cancel function by using the `WithCancel` function in the `context` package. Next, you use `defer` to call `cancel` when the `main` function exits. This closes the channel returned by `Done`, and since a closed channel always returns a value, it ensures that the goroutine running `countTo` exits.

Using the context to terminate a goroutine is a very common pattern. It allows you to stop goroutines based on something from an earlier function in the call stack. In "Cancellation" on page 358, you'll learn in detail how to use the context to tell one or more goroutines that it is time to shut down.

Know When to Use Buffered and Unbuffered Channels

One of the most complicated techniques to master in Go concurrency is deciding when to use a buffered channel. By default, channels are unbuffered, and they are easy to understand: one goroutine writes and waits for another goroutine to pick up its work, like a baton in a relay race. Buffered channels are much more complicated. You have to pick a size, since buffered channels never have unlimited buffers. Proper use of a buffered channel means that you must handle the case where the buffer is full and your writing goroutine blocks waiting for a reading goroutine. So what is the proper use of a buffered channel?

The case for buffered channels is subtle. To sum it up in a single sentence: buffered channels are useful when you know how many goroutines you have launched, want to limit the number of goroutines you will launch, or want to limit the amount of work that is queued up.

Buffered channels work great when you either want to gather data back from a set of goroutines that you have launched or want to limit concurrent usage. They are also helpful for managing the amount of work a system has queued up, preventing your services from falling behind and becoming overwhelmed. Here are a couple of examples to show how they can be used.

In the first example, you are processing the first 10 results on a channel. To do this, you launch 10 goroutines, each of which writes its results to a buffered channel:

```go
func processChannel(ch chan int) []int {
    const conc = 10
    results := make(chan int, conc)
    for i := 0; i < conc; i++ {
        go func() {
            v := <- ch
            results <- process(v)
        }()
    }
    var out []int
    for i := 0; i < conc; i++ {
        out = append(out, <-results)
    }
    return out
}
```

You know exactly how many goroutines have been launched, and you want each goroutine to exit as soon as it finishes its work. This means you can create a buffered channel with one space for each launched goroutine, and have each goroutine write data to this goroutine without blocking. You can then loop over the buffered channel, reading out the values as they are written. When all the values have been read, you return the results, knowing that you aren't leaking any goroutines.

You can find this code in the *sample_code/buffered_channel_work* directory in the Chapter 12 repository.

Implement Backpressure

Another technique that can be implemented with a buffered channel is *backpressure*. It is counterintuitive, but systems perform better overall when their components limit the amount of work they are willing to perform. You can use a buffered channel and a `select` statement to limit the number of simultaneous requests in a system:

```
type PressureGauge struct {
    ch chan struct{}
}

func New(limit int) *PressureGauge {
    return &PressureGauge{
        ch: make(chan struct{}, limit),
    }
}

func (pg *PressureGauge) Process(f func()) error {
    select {
    case pg.ch <- struct{}{}:
        f()
        <-pg.ch
        return nil
    default:
        return errors.New("no more capacity")
    }
}
```

In this code, you create a struct containing a buffered channel that can hold a number of "tokens" and a function to run. Every time a goroutine wants to use the function, it calls `Process`. This is one of the rare examples of the same goroutine both reading and writing the same channel. The `select` tries to write a token to the channel. If it can, the function runs, and then a token is read to the buffered channel. If it can't write a token, the `default` case runs, and an error is returned instead. Here's a quick example that uses this code with the built-in HTTP server (you'll learn more about working with HTTP in "The Server" on page 337):

```
func doThingThatShouldBeLimited() string {
    time.Sleep(2 * time.Second)
    return "done"
}

func main() {
    pg := New(10)
    http.HandleFunc("/request", func(w http.ResponseWriter, r *http.Request) {
        err := pg.Process(func() {
            w.Write([]byte(doThingThatShouldBeLimited()))
        })
        if err != nil {
            w.WriteHeader(http.StatusTooManyRequests)
            w.Write([]byte("Too many requests"))
        }
    })
    http.ListenAndServe(":8080", nil)
}
```

You can find this code in the *sample_code/backpressure* directory in the Chapter 12 repository.

Turn Off a case in a select

When you need to combine data from multiple concurrent sources, the `select` keyword is great. However, you need to properly handle closed channels. If one of the cases in a `select` is reading a closed channel, it will always be successful, returning the zero value. Every time that case is selected, you need to check to make sure that the value is valid and skip the case. If reads are spaced out, your program is going to waste a lot of time reading junk values. Even if there is lots of activity on the nonclosed channels, your program will still spend some portion of its time reading from the closed channel, since `select` chooses a case at random.

When that happens, you rely on something that looks like an error: reading a `nil` channel. As you saw earlier, reading from or writing to a `nil` channel causes your code to hang forever. While that is bad if it is triggered by a bug, you can use a `nil` channel to disable a `case` in a `select`. When you detect that a channel has been closed, set the channel's variable to `nil`. The associated case will no longer run, because the read from the `nil` channel never returns a value. Here is a `for-select` loop that reads from two channels until both are closed:

```go
// in and in2 are channels
for count := 0; count < 2; {
    select {
    case v, ok := <-in:
        if !ok {
            in = nil // the case will never succeed again!
            count++
            continue
        }
        // process the v that was read from in
    case v, ok := <-in2:
        if !ok {
            in2 = nil // the case will never succeed again!
            count++
            continue
        }
        // process the v that was read from in2
    }
}
```

You can try out this code on The Go Playground or in the *sample_code/close_case* directory in the Chapter 12 repository.

Time Out Code

Most interactive programs have to return a response within a certain amount of time. One of the things that you can do with concurrency in Go is manage how much time a request (or a part of a request) has to run. Other languages introduce additional features on top of promises or futures to add this functionality, but Go's timeout idiom shows how you build complicated features from existing parts. Let's take a look:

```go
func timeLimit[T any](worker func() T, limit time.Duration) (T, error) {
    out := make(chan T, 1)
    ctx, cancel := context.WithTimeout(context.Background(), limit)
    defer cancel()
    go func() {
        out <- worker()
    }()
    select {
    case result := <-out:
        return result, nil
    case <-ctx.Done():
        var zero T
        return zero, errors.New("work timed out")
    }
}
```

Whenever you need to limit how long an operation takes in Go, you'll see a variation on this pattern. I talk about the context in Chapter 14 and cover using timeouts in detail in "Contexts with Deadlines" on page 363. For now, all you need to know is that reaching the timeout cancels the context. The `Done` method on the context returns a channel that returns a value when the context is canceled by either timing out or when the context's cancel method is called. You create a timed context by using the `WithTimeout` function in the `context` package and specify how long to wait by using constants from the `time` package (I'll talk more about the `time` package in "time" on page 324).

Once the context is set up, you run the worker in a goroutine and then use `select` to choose between two cases. The first case reads the value from the `out` channel when the work completes. The second case waits for the channel returned by the `Done` method to return a value, just as you saw in "Use the Context to Terminate Goroutines" on page 300. If it does, you return a timeout error. You write to a buffered channel of size 1 so that the channel write in the goroutine will complete even if `Done` is triggered first.

You can try out this code on The Go Playground or in the *sample_code/time_out* directory in the Chapter 12 repository.

If `timeLimit` exits before the goroutine finishes processing, the goroutine continues to run, eventually writing the returned value to the buffered channel and exiting. You just don't do anything with the result that is returned. If you want to stop work in a goroutine when you are no longer waiting for it to complete, use context cancellation, which I'll discuss in "Cancellation" on page 358.

Use WaitGroups

Sometimes one goroutine needs to wait for multiple goroutines to complete their work. If you are waiting for a single goroutine, you can use the context cancellation pattern that you saw earlier. But if you are waiting on several goroutines, you need to use a WaitGroup, which is found in the sync package in the standard library. Here is a simple example, which you can run on The Go Playground or in the *sample_code/waitgroup* directory in the Chapter 12 repository:

```go
func main() {
    var wg sync.WaitGroup
    wg.Add(3)
    go func() {
        defer wg.Done()
        doThing1()
    }()
    go func() {
        defer wg.Done()
        doThing2()
    }()
    go func() {
        defer wg.Done()
        doThing3()
    }()
    wg.Wait()
}
```

A sync.WaitGroup doesn't need to be initialized, just declared, as its zero value is useful. There are three methods on sync.WaitGroup: Add, which increments the counter of goroutines to wait for; Done, which decrements the counter and is called by a goroutine when it is finished; and Wait, which pauses its goroutine until the counter hits zero. Add is usually called once, with the number of goroutines that will be launched. Done is called within the goroutine. To ensure that it is called, even if the goroutine panics, you use a defer.

You'll notice that you don't explicitly pass the sync.WaitGroup. There are two reasons. The first is that you must ensure that every place that uses a sync.WaitGroup is using the same instance. If you pass the sync.WaitGroup to the goroutine function and don't use a pointer, then the function has a *copy* and the call to Done won't decrement the original sync.WaitGroup. By using a closure to capture the sync.WaitGroup, you are assured that every goroutine is referring to the same instance.

The second reason is design. Remember, you should keep concurrency out of your API. As you saw with channels earlier, the usual pattern is to launch a goroutine with a closure that wraps the business logic. The closure manages issues around concurrency, and the function provides the algorithm.

Let's take a look at a more realistic example. As I mentioned earlier, when you have multiple goroutines writing to the same channel, you need to make sure that the channel being written to is closed only once. A `sync.WaitGroup` is perfect for this. Let's see how it works in a function that processes the values in a channel concurrently, gathers the results into a slice, and returns the slice:

```
func processAndGather[T, R any](in <-chan T, processor func(T) R, num int) []R {
    out := make(chan R, num)
    var wg sync.WaitGroup
    wg.Add(num)
    for i := 0; i < num; i++ {
        go func() {
            defer wg.Done()
            for v := range in {
                out <- processor(v)
            }
        }()
    }
    go func() {
        wg.Wait()
        close(out)
    }()
    var result []R
    for v := range out {
        result = append(result, v)
    }
    return result
}
```

In this example, you launch a monitoring goroutine that waits until all the processing goroutines exit. When they do, the monitoring goroutine calls `close` on the output channel. The `for-range` channel loop exits when `out` is closed and the buffer is empty. Finally, the function returns the processed values. You can try out this code in the *sample_code/waitgroup_close_once* directory in the Chapter 12 repository.

While `WaitGroups` are handy, they shouldn't be your first choice when coordinating goroutines. Use them only when you have something to clean up (like closing a channel they all write to) after all your worker goroutines exit.

golang.org/x and errgroup.Group

The Go authors maintain a set of utilities that supplements the standard library. Collectively known as the `golang.org/x` packages, they include a package called `errgroup` that contains a type, `errgroup.Group`. It builds on top of `WaitGroup` to create a set of goroutines that stop processing when one of them returns an error. Read the `errgroup.Group` documentation to learn more.

Run Code Exactly Once

As I covered in "Avoiding the init Function if Possible" on page 239, `init` should be reserved for initialization of effectively immutable package-level state. However, sometimes you want to *lazy load*, or call some initialization code exactly once after program launch time. This is usually because the initialization is relatively slow and may not even be needed every time your program runs. The `sync` package includes a handy type called `Once` that enables this functionality. Let's take a quick look at how it works. Say you have some code that takes a long time to initialize:

```
type SlowComplicatedParser interface {
    Parse(string) string
}

func initParser() SlowComplicatedParser {
    // do all sorts of setup and loading here
}
```

Here's how you use `sync.Once` to delay initialization of a `SlowComplicatedParser`:

```
var parser SlowComplicatedParser
var once sync.Once

func Parse(dataToParse string) string {
    once.Do(func() {
        parser = initParser()
    })
    return parser.Parse(dataToParse)
}
```

There are two package-level variables: `parser`, which is of type `SlowComplicatedParser`, and `once`, which is of type `sync.Once`. As with `sync.WaitGroup`, you do not have to configure an instance of `sync.Once`. This is an example of *making the zero value useful*, which is a common pattern in Go.

As with `sync.WaitGroup`, you must make sure not to make a copy of an instance of `sync.Once`, because each copy has its own state to indicate whether it has already been used. Declaring a `sync.Once` instance inside a function is usually the wrong thing to do, as a new instance will be created on every function call and there will be no memory of previous invocations.

In the example, you want to make sure that `parser` is initialized only once, so you set the value of `parser` from within a closure that's passed to the `Do` method on `once`. If `Parse` is called more than once, `once.Do` will not execute the closure again.

You can try out this code on The Go Playground or in the *sample_code/sync_once* directory in the Chapter 12 repository.

Go 1.21 added helper functions that make it easier to run a function exactly once: `sync.OnceFunc`, `sync.OnceValue`, and `sync.OnceValues`. The only difference between the three functions is the number of return values of the passed-in function (zero, one, or two, respectively). The `sync.OnceValue` and `sync.OnceValues` functions are generic, so they adapt to the type of the original function's return values.

Using these functions is straightforward. You pass the original function to the helper function and get back a function that calls the original function only once. The values returned by the original function are cached. Here's how you can use `sync.OnceValue` to rewrite the `Parse` function in the previous example:

```
var initParserCached func() SlowComplicatedParser = sync.OnceValue(initParser)

func Parse(dataToParse string) string {
    parser := initParserCached()
    return parser.Parse(dataToParse)
}
```

The package-level `initParserCached` variable is assigned the function returned by `sync.OnceValue` when `initParser` is passed to it. The first time `initParserCached` is called, `initParser` is also called, and its return value is cached. Each subsequent time `initParserCached` is called, the cached value is returned. This means you can get rid of the `parser` package-level variable.

You can try out this code on The Go Playground or in the *sample_code/sync_value* directory in the Chapter 12 repository.

Put Your Concurrent Tools Together

Let's go back to the example from the first section in the chapter. You have a function that calls three web services. You send data to two of those services, and then take the results of those two calls and send them to the third, returning the result. The entire process must take less than 50 milliseconds, or an error is returned.

You'll start with the function you invoke:

```
func GatherAndProcess(ctx context.Context, data Input) (COut, error) {
    ctx, cancel := context.WithTimeout(ctx, 50*time.Millisecond)
    defer cancel()

    ab := newABProcessor()
    ab.start(ctx, data)
    inputC, err := ab.wait(ctx)
    if err != nil {
        return COut{}, err
    }

    c := newCProcessor()
    c.start(ctx, inputC)
```

```go
    out, err := c.wait(ctx)
    return out, err
}
```

The first thing to do is set up a `context.Context` that times out in 50 milliseconds, as you saw in "Time Out Code" on page 304.

After creating the context, use a `defer` to make sure the context's `cancel` function is called. As I'll discuss in "Cancellation" on page 358, you must call this function, or resources leak.

You are using `A` and `B` as the names of the two services that are called in parallel, so you'll make a new `abProcessor` to call them. You then start processing with a call to the `start` method, and then wait for your results with a call to the `wait` method.

When `wait` returns, you do a standard error check. If all is well, you call the third service, which you are calling `C`. The logic is the same as before. Processing is started with a call to the `start` method on the `cProcessor` and then you wait for the result with a call to the `wait` method on `cProcessor`. You then return the result of the `wait` method call.

This looks a lot like standard sequential code without concurrency. Let's look at the `abProcessor` and `cProcessor` to see how the concurrency happens:

```go
type abProcessor struct {
    outA chan aOut
    outB chan bOut
    errs chan error
}

func newABProcessor() *abProcessor {
    return &abProcessor{
        outA: make(chan aOut, 1),
        outB: make(chan bOut, 1),
        errs: make(chan error, 2),
    }
}
```

The `abProcessor` has three fields, all of which are channels. They are `outA`, `outB`, and `errs`. You'll see how you use all these channels next. Notice that every channel is buffered, so that the goroutines that write to them can exit after writing without waiting for a read to happen. The `errs` channel has a buffer size of `2`, because it could have up to two errors written to it.

Next is the implementation of the `start` method:

```go
func (p *abProcessor) start(ctx context.Context, data Input) {
    go func() {
        aOut, err := getResultA(ctx, data.A)
        if err != nil {
            p.errs <- err
            return
        }
        p.outA <- aOut
    }()
    go func() {
        bOut, err := getResultB(ctx, data.B)
        if err != nil {
            p.errs <- err
            return
        }
        p.outB <- bOut
    }()
}
```

The `start` method launches two goroutines. The first one calls `getResultA` to talk to the `A` service. If the call returns an error, you write to the `errs` channel. Otherwise, you write to the `outA` channel. Since these channels are buffered, the goroutine will not hang, no matter which channel is written to. Also notice that you are passing the context along to `getResultA`, which allows it to cancel processing if the timeout happens.

The second goroutine is just like the first, only it calls `getResultB` and writes to the `outB` channel on success.

Let's see what the `wait` method for `ABProcessor` looks like:

```
func (p *abProcessor) wait(ctx context.Context) (cIn, error) {
    var cData cIn
    for count := 0; count < 2; count++ {
        select {
        case a := <-p.outA:
            cData.a = a
        case b := <-p.outB:
            cData.b = b
        case err := <-p.errs:
            return cIn{}, err
        case <-ctx.Done():
            return cIn{}, ctx.Err()
        }
    }
    return cData, nil
}
```

The `wait` method on `abProcessor` is the most complicated method you need to implement. It populates a struct of type `cIn`, which holds the data returned from calling the `A` service and the `B` service. You define your output variable, `cData`, as type `cIn`. You then have a `for` loop that counts to two, since you need to read from two channels to finish successfully. Inside the loop, you have a `select` statement. If you read a value on `outA`, you set the `a` field on `cData`. If you read a value on `outB`, you set the b field on cData. If you read a value on the errs channel, you return immediately with the error. Finally, if the context times out, you return immediately with the error from the context's Err method.

Once you have read a value from both the p.outA channel and the p.outB channel, you exit the loop and return the input that you're going to use with the cProcessor.

The cProcessor looks like a simpler version of the abProcessor:

```go
type cProcessor struct {
    outC chan COut
    errs chan error
}

func newCProcessor() *cProcessor {
    return &cProcessor{
        outC: make(chan COut, 1),
        errs: make(chan error, 1),
    }
}

func (p *cProcessor) start(ctx context.Context, inputC cIn) {
    go func() {
        cOut, err := getResultC(ctx, inputC)
        if err != nil {
            p.errs <- err
            return
        }
        p.outC <- cOut
    }()
}

func (p *cProcessor) wait(ctx context.Context) (COut, error) {
    select {
    case out := <-p.outC:
        return out, nil
    case err := <-p.errs:
        return COut{}, err
    case <-ctx.Done():
        return COut{}, ctx.Err()
    }
}
```

The cProcessor struct has one out channel and one error channel.

The start method on the cProcessor looks like the start method on the abProcessor. It launches a goroutine that calls getResultC with your input data, writes to the errs channel on an error, and writes to the outC channel on success.

Finally, the `wait` method on the `cProcessor` is a simple `select` statement that checks whether there's a value to read from the `outC` channel, the `errs` channel, or the context's `Done` channel.

By structuring code with goroutines, channels, and `select` statements, you separate the individual steps, allow independent parts to run and complete in any order, and cleanly exchange data between the dependent parts. In addition, you make sure that no part of the program hangs, and you properly handle timeouts set both within this function and from earlier functions in the call history. If you are not convinced that this is a better method for implementing concurrency, try to implement this in another language. You might be surprised at how difficult it is.

You can find the code for this concurrent pipeline in the *sample_code/pipeline* directory in the Chapter 12 repository.

When to Use Mutexes Instead of Channels

If you've had to coordinate access to data across threads in other programming languages, you have probably used a *mutex*. This is short for *mutual exclusion*, and the job of a mutex is to limit the concurrent execution of some code or access to a shared piece of data. This protected part is called the *critical section*.

There are good reasons Go's creators designed channels and `select` to manage concurrency. The main problem with mutexes is that they obscure the flow of data through a program. When a value is passed from goroutine to goroutine over a series of channels, the data flow is clear. Access to the value is localized to a single goroutine at a time. When a mutex is used to protect a value, there is nothing to indicate which goroutine currently has ownership of the value, because access to the value is shared by all the concurrent processes. That makes it hard to understand the order of processing. There is a saying in the Go community to describe this philosophy: "Share memory by communicating; do not communicate by sharing memory."

That said, sometimes it is clearer to use a mutex, and the Go standard library includes mutex implementations for these situations. The most common case is when your goroutines read or write a shared value, but don't process the value. Let's use an in-memory scoreboard for a multiplayer game as an example. You'll first see how to implement this using channels. Here's a function that you can launch as a goroutine to manage the scoreboard:

```go
func scoreboardManager(ctx context.Context, in <-chan func(map[string]int)) {
    scoreboard := map[string]int{}
    for {
        select {
        case <-ctx.Done():
            return
        case f := <-in:
            f(scoreboard)
        }
    }
}
```

This function declares a map and then listens on one channel for a function that reads or modifies the map and on a context's Done channel to know when to shut down. Let's create a type with a method to write a value to the map:

```
type ChannelScoreboardManager chan func(map[string]int)

func NewChannelScoreboardManager(ctx context.Context) ChannelScoreboardManager {
    ch := make(ChannelScoreboardManager)
    go scoreboardManager(ctx, ch)
    return ch
}

func (csm ChannelScoreboardManager) Update(name string, val int) {
    csm <- func(m map[string]int) {
        m[name] = val
    }
}
```

The update method is very straightforward: just pass a function that puts a value into the map. But how about reading from the scoreboard? You need to return a value back. That means creating a channel that's written to within the passed-in function:

```
func (csm ChannelScoreboardManager) Read(name string) (int, bool) {
    type Result struct {
        out int
        ok  bool
    }
    resultCh := make(chan Result)
    csm <- func(m map[string]int) {
        out, ok := m[name]
        resultCh <- Result{out, ok}
    }
    result := <-resultCh
    return result.out, result.ok
}
```

While this code works, it's cumbersome and allows only a single reader at a time. A better approach is to use a mutex. The standard library has two mutex implementations, both in the `sync` package. The first, `Mutex`, has two methods, `Lock` and `Unlock`. Calling `Lock` causes the current goroutine to pause as long as another goroutine is currently in the critical section. When the critical section is clear, the lock is *acquired* by the current goroutine, and the code in the critical section is executed. A call to the `Unlock` method on the `Mutex` marks the end of the critical section.

The second mutex implementation, called `RWMutex`, allows you to have both reader locks and writer locks. While only one writer can be in the critical section at a time, reader locks are shared; multiple readers can be in the critical section at once. The writer lock is managed with the `Lock` and `Unlock` methods, while the reader lock is managed with `RLock` and `RUnlock` methods.

Anytime you acquire a mutex lock, you must make sure that you release the lock. Use a `defer` statement to call `Unlock` immediately after calling `Lock` or `RLock`:

```go
type MutexScoreboardManager struct {
    l          sync.RWMutex
    scoreboard map[string]int
}

func NewMutexScoreboardManager() *MutexScoreboardManager {
    return &MutexScoreboardManager{
        scoreboard: map[string]int{},
    }
}

func (msm *MutexScoreboardManager) Update(name string, val int) {
    msm.l.Lock()
    defer msm.l.Unlock()
    msm.scoreboard[name] = val
}

func (msm *MutexScoreboardManager) Read(name string) (int, bool) {
    msm.l.RLock()
    defer msm.l.RUnlock()
    val, ok := msm.scoreboard[name]
    return val, ok
}
```

You can find the example in the *sample_code/mutex* directory in the Chapter 12 repository.

Now that you've seen an implementation using mutexes, carefully consider your options before using one. Katherine Cox-Buday's excellent book *Concurrency in Go* (O'Reilly) includes a decision tree to help you decide whether to use channels or mutexes:

- If you are coordinating goroutines or tracking a value as it is transformed by a series of goroutines, use channels.
- If you are sharing access to a field in a struct, use mutexes.
- If you discover a critical performance issue when using channels (see "Using Benchmarks" on page 393 to learn how to do this), and you cannot find any other way to fix the issue, modify your code to use a mutex.

Since your scoreboard is a field in a struct and there's no transfer of the scoreboard, using a mutex makes sense. This is a good use for a mutex only because the data is stored in-memory. When data is stored in external services, like an HTTP server or a database, don't use a mutex to guard access to the system.

Mutexes require you to do more bookkeeping. For example, you must correctly pair locks and unlocks, or your programs will likely deadlock. The example both acquires and releases the locks within the same method. Another issue is that mutexes in Go aren't *reentrant*. If a goroutine tries to acquire the same lock twice, it deadlocks, waiting for itself to release the lock. This is different from languages like Java, where locks are reentrant.

Nonreentrant locks make it tricky to acquire a lock in a function that calls itself recursively. You must release the lock before the recursive function call. In general, be careful when holding a lock while making a function call, because you don't know what locks are going to be acquired in those calls. If your function calls another function that tries to acquire the same mutex lock, the goroutine deadlocks.

Like `sync.WaitGroup` and `sync.Once`, mutexes must never be copied. If they are passed to a function or accessed as a field on a struct, it must be via a pointer. If a mutex is copied, its lock won't be shared.

> Never try to access a variable from multiple goroutines unless you acquire a mutex for that variable first. It can cause odd errors that are hard to trace. See "Finding Concurrency Problems with the Data Race Detector" on page 406 to learn how to detect these problems.

sync.Map—This Is Not the Map You Are Looking For

When looking through the `sync` package, you'll find a type called `Map`. It provides a concurrency-safe version of Go's built-in `map`. Because of trade-offs in its implementation, `sync.Map` is appropriate only in very specific situations:

- When you have a shared map where key-value pairs are inserted once and read many times
- When goroutines share the map, but don't access each other's keys and values

Furthermore, because `sync.Map` was added to the standard library before the introduction of generics, `sync.Map` uses `any` as the type for its keys and values. This means the compiler cannot help you ensure that the right data types are used.

Given these limitations, in the rare situations where you need to share a map across multiple goroutines, use a built-in `map` protected by a `sync.RWMutex`.

Atomics—You Probably Don't Need These

In addition to mutexes, Go provides another way to keep data consistent across multiple threads. The `sync/atomic` package provides access to the *atomic variable* operations built into modern CPUs to add, swap, load, store, or compare and swap (CAS) a value that fits into a single register.

If you need to squeeze out every last bit of performance and are an expert on writing concurrent code, you'll be glad that Go includes atomic support. For everyone else, use goroutines and mutexes to manage your concurrency needs.

Where to Learn More About Concurrency

I've covered a few simple concurrency patterns here, but there are many more. In fact, you could write an entire book on how to properly implement various concurrency patterns in Go, and, luckily, Katherine Cox-Buday has. I've already mentioned *Concurrency in Go*, when discussing how to decide between mutexes or channels, but it's an excellent resource on all things involving Go and concurrency. Check out her book if you want to learn more.

Exercises

Using concurrency effectively is one of the most important skills for a Go developer. Work through these exercises to see if you have mastered them. The solutions are available in the *exercise_solutions* directory in the Chapter 12 repository.

1. Create a function that launches three goroutines that communicate using a channel. The first two goroutines each write 10 numbers to the channel. The third goroutine reads all the numbers from the channel and prints them out. The function should exit when all values have been printed out. Make sure that none of the goroutines leak. You can create additional goroutines if needed.
2. Create a function that launches two goroutines. Each goroutine writes 10 numbers to its own channel. Use a `for-select` loop to read from both channels, printing out the number and the goroutine that wrote the value. Make sure that your function exits after all values are read and that none of your goroutines leak.
3. Write a function that builds a `map[int]float64` where the keys are the numbers from 0 (inclusive) to 100,000 (exclusive) and the values are the square roots of those numbers (use the `math.Sqrt` function to calculate square roots). Use `sync.OnceValue` to generate a function that caches the `map` returned by this function and use the cached value to look up square roots for every 1,000th number from 0 to 100,000.

Wrapping Up

In this chapter, you've looked at concurrency and learned why Go's approach is simpler than more traditional concurrency mechanisms. In doing so, you've also learned when you should use concurrency as well as a few concurrency rules and patterns. In the next chapter, you're going to take a quick look at Go's standard library, which embraces a "batteries included" ethos for modern computing.