CHAPTER 4

Concurrency Patterns in Go

We've explored the fundamentals of Go's concurrency primitives and discussed how to properly use these primitives. In this chapter, we'll do a deep-dive into how to compose these primitives into patterns that will help keep your system scalable and maintainable.

However, before we get started, we need to touch upon the format of some of the patterns contained in this chapter. In a lot of the examples, we'll be using channels that pass empty interfaces (`interface{}`) around. Usage of empty interfaces in Go is controversial; however, I've done this for a couple of reasons. The first is that it makes it easier to write concise examples in the remainder of the book. The second is that in some cases I believe this to be more representative of what the pattern is trying to accomplish. We'll discuss this point more directly in the section "Pipelines" on page 100.

If this is just too contentious to you, remember that you can always create Go generators for this code, and generate the patterns to utilize the type you're interested in.

With that said, let's dive in and learn about some patterns for concurrency in Go!

Confinement

When working with concurrent code, there are a few different options for safe operation. We've gone over two of them:

- Synchronization primitives for sharing memory (e.g., `sync.Mutex`)
- Synchronization via communicating (e.g., channels)

However, there are a couple of other options that are implicitly safe within multiple concurrent processes:

- Immutable data
- Data protected by confinement

In some sense, immutable data is ideal because it is implicitly concurrent-safe. Each concurrent process may operate on the same data, but it may not modify it. If it wants to create new data, it must create a new copy of the data with the desired modifications. This allows not only a lighter cognitive load on the developer, but can also lead to faster programs if it leads to smaller critical sections (or eliminates them altogether). In Go, you can achieve this by writing code that utilizes copies of values instead of pointers to values in memory. Some languages support utilization of pointers with explicitly immutable values; however, Go is not among these.

Confinement can also allow for a lighter cognitive load on the developer and smaller critical sections. The techniques to confine concurrent values are a bit more involved than simply passing copies of values, so in this chapter we'll explore these confinement techniques in depth.

Confinement is the simple yet powerful idea of ensuring information is only ever available from *one* concurrent process. When this is achieved, a concurrent program is implicitly safe and no synchronization is needed. There are two kinds of confinement possible: ad hoc and lexical.

Ad hoc confinement is when you achieve confinement through a convention—whether it be set by the languages community, the group you work within, or the codebase you work within. In my opinion, sticking to convention is difficult to achieve on projects of any size unless you have tools to perform static analysis on your code every time someone commits some code. Here's an example of ad hoc confinement that demonstrates why:

```
data := make([]int, 4)

loopData := func(handleData chan<- int) {
    defer close(handleData)
    for i := range data {
        handleData <- data[i]
    }
}

handleData := make(chan int)
go loopData(handleData)

for num := range handleData {
    fmt.Println(num)
}
```

We can see that the `data` slice of integers is available from both the `loopData` function and the loop over the `handleData` channel; however, by convention we're only accessing it from the `loopData` function. But as the code is touched by many people, and deadlines loom, mistakes might be made, and the confinement might break down and cause issues. As I mentioned, a static-analysis tool might catch these kinds of issues, but static analysis on a Go codebase suggests a level of maturity that not many teams achieve. This is why I prefer lexical confinement: it wields the compiler to enforce the confinement.

Lexical confinement involves using lexical scope to expose only the correct data and concurrency primitives for multiple concurrent processes to use. It makes it impossible to do the wrong thing. We've actually already touched on this topic in Chapter 3. Recall the section on channels, which discusses only exposing read or write aspects of a channel to the concurrent processes that need them. Let's take a look at that example again:

```
chanOwner := func() <-chan int {
    results := make(chan int, 5) ❶
    go func() {
        defer close(results)
        for i := 0; i <= 5; i++ {
            results <- i
        }
    }()
    return results
}

consumer := func(results <-chan int) { ❸
    for result := range results {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}

results := chanOwner()          ❷
consumer(results)
```

❶ Here we instantiate the channel within the lexical scope of the `chanOwner` function. This limits the scope of the write aspect of the `results` channel to the closure defined below it. In other words, it *confines* the write aspect of this channel to prevent other goroutines from writing to it.

❷ Here we receive the read aspect of the channel and we're able to pass it into the consumer, which can do nothing but read from it. Once again this confines the main goroutine to a read-only view of the channel.

❸ Here we receive a read-only copy of an `int` channel. By declaring that the only usage we require is read access, we confine usage of the channel within the `consume` function to only reads.

Set up this way, it is impossible to utilize the channels in this small example. This is a good lead-in to confinement, but probably not a very interesting example since channels are concurrent-safe. Let's take a look at an example of confinement that uses a data structure which is not concurrent-safe, an instance of `bytes.Buffer`:

```
printData := func(wg *sync.WaitGroup, data []byte) {
    defer wg.Done()

    var buff bytes.Buffer
    for _, b := range data {
        fmt.Fprintf(&buff, "%c", b)
    }
    fmt.Println(buff.String())
}

var wg sync.WaitGroup
wg.Add(2)
data := []byte("golang")
go printData(&wg, data[:3])     ❶
go printData(&wg, data[3:])     ❷

wg.Wait()
```

❶ Here we pass in a slice containing the first three bytes in the `data` structure.

❷ Here we pass in a slice containing the last three bytes in the `data` structure.

In this example, you can see that because `printData` doesn't close around the `data` slice, it cannot access it, and needs to take in a slice of `byte` to operate on. We pass in different subsets of the slice, thus constraining the goroutines we start to only the part of the slice we're passing in. Because of the lexical scope, we've made it impossible[1] to do the wrong thing, and so we don't need to synchronize memory access or share data through communication.

So what's the point? Why pursue confinement if we have synchronization available to us? The answer is improved performance and reduced cognitive load on developers. Synchronization comes with a cost, and if you can avoid it you won't have any critical sections, and therefore you won't have to pay the cost of synchronizing them. You also sidestep an entire class of issues possible with synchronization; developers simply

1 I'm ignoring the possibility of manually manipulating memory via the `unsafe` package. It's called `unsafe` for a reason!

don't have to worry about these issues. Concurrent code that utilizes lexical confinement also has the benefit of usually being simpler to understand than concurrent code without lexically confined variables. This is because within the context of your lexical scope you can write synchronous code.

Having said that, it can be difficult to establish confinement, and so sometimes we have to fall back to our wonderful Go concurrency primitives.

The for-select Loop

Something you'll see over and over again in Go programs is the for-select loop. It's nothing more than something like this:

```
for { // Either loop infinitely or range over something
    select {
    // Do some work with channels
    }
}
```

There are a couple of different scenarios where you'll see this pattern pop up.

Sending iteration variables out on a channel

Oftentimes you'll want to convert something that can be iterated over into values on a channel. This is nothing fancy, and usually looks something like this:

```
for _, s := range []string{"a", "b", "c"} {
    select {
    case <-done:
        return
    case stringStream <- s:
    }
}
```

Looping infinitely waiting to be stopped

It's very common to create goroutines that loop infinitely until they're stopped. There are a couple variations of this one. Which one you choose is purely a stylistic preference.

The first variation keeps the `select` statement as short as possible:

```
for {
    select {
    case <-done:
        return
    default:
    }

    // Do non-preemptable work
}
```

If the `done` channel isn't closed, we'll exit the `select` statement and continue on to the rest of our `for` loop's body.

The second variation embeds the work in a `default` clause of the `select` statement:

```
for {
    select {
    case <-done:
        return
    default:
        // Do non-preemptable work
    }
}
```

When we enter the `select` statement, if the `done` channel hasn't been closed, we'll execute the `default` clause instead.

There's nothing more to this pattern, but it shows up all over the place, and so it's worth mentioning.

Preventing Goroutine Leaks

As we covered in the section "Goroutines" on page 37, we know goroutines are cheap and easy to create; it's one of the things that makes Go such a productive language. The runtime handles multiplexing the goroutines onto any number of operating system threads so that we don't often have to worry about that level of abstraction. But they *do* cost resources, and goroutines are not garbage collected by the runtime, so regardless of how small their memory footprint is, we don't want to leave them lying about our process. So how do we go about ensuring they're cleaned up?

Let's start from the beginning and think about this step by step: why would a goroutine exist? In Chapter 2, we established that goroutines represent units of work that may or may not run in parallel with each other. The goroutine has a few paths to termination:

- When it has completed its work.
- When it cannot continue its work due to an unrecoverable error.
- When it's told to stop working.

We get the first two paths for free—these paths are your algorithm—but what about work cancellation? This turns out to be the most important bit because of the network effect: if you've begun a goroutine, it's most likely cooperating with several other goroutines in some sort of organized fashion. We could even represent this interconnectedness as a graph: whether or not a child goroutine should continue executing might be predicated on knowledge of the state of many *other* goroutines. The parent

goroutine (often the main goroutine) with this full contextual knowledge should be able to tell its child goroutines to terminate. We'll continue looking at large-scale goroutine interdependence in the next chapter, but for now let's consider how to ensure a single child goroutine is guaranteed to be cleaned up. Let's start with a simple example of a goroutine leak:

```
doWork := func(strings <-chan string) <-chan interface{} {
    completed := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(completed)
        for s := range strings {
            // Do something interesting
            fmt.Println(s)
        }
    }()
    return completed
}

doWork(nil)
// Perhaps more work is done here
fmt.Println("Done.")
```

Here we see that the main goroutine passes a nil channel into `doWork`. Therefore, the `strings` channel will never actually gets any strings written onto it, and the goroutine containing `doWork` will remain in memory for the lifetime of this process (we would even deadlock if we joined the goroutine within `doWork` and the main goroutine).

In this example, the lifetime of the process is very short, but in a real program, goroutines could easily be started at the beginning of a long-lived program. In the worst case, the main goroutine could *continue* to spin up goroutines throughout its life, causing creep in memory utilization.

The way to successfully mitigate this is to establish a signal between the parent goroutine and its children that allows the parent to signal cancellation to its children. By convention, this signal is usually a read-only channel named `done`. The parent goroutine passes this channel to the child goroutine and then closes the channel when it wants to cancel the child goroutine. Here's an example:

```
doWork := func(
  done <-chan interface{},
  strings <-chan string,
) <-chan interface{} { ❶
    terminated := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(terminated)
        for {
            select {
            case s := <-strings:
```

```
                // Do something interesting
                fmt.Println(s)
            case <-done: ❷
                return
            }
        }
    }()
    return terminated
}

done := make(chan interface{})
terminated := doWork(done, nil)

go func() { ❸
    // Cancel the operation after 1 second.
    time.Sleep(1 * time.Second)
    fmt.Println("Canceling doWork goroutine...")
    close(done)
}()

<-terminated ❹
fmt.Println("Done.")
```

❶ Here we pass the `done` channel to the `doWork` function. As a convention, this channel is the first parameter.

❷ On this line we see the ubiquitous for-select pattern in use. One of our case statements is checking whether our `done` channel has been signaled. If it has, we return from the goroutine.

❸ Here we create another goroutine that will cancel the goroutine spawned in `doWork` if more than one second passes.

❹ This is where we join the goroutine spawned from `doWork` with the main goroutine.

And the resulting output is:

```
Canceling doWork goroutine...
doWork exited.
Done.
```

You can see that despite passing in `nil` for our `strings` channel, our goroutine still exits successfully. Unlike the example before it, in this example we *do* join the two goroutines, and yet do not receive a deadlock. This is because before we join the two goroutines, we create a third goroutine to cancel the goroutine within `doWork` after a second. We have successfully eliminated our goroutine leak!

The previous example handles the case for goroutines receiving on a channel nicely, but what if we're dealing with the reverse situation: a goroutine blocked on attempting to write a value to a channel? Here's a quick example to demonstrate the issue:

```
newRandStream := func() <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited.") ❶
        defer close(randStream)
        for {
            randStream <- rand.Int()
        }
    }()

    return randStream
}

randStream := newRandStream()
fmt.Println("3 random ints:")
for i := 1; i <= 3; i++ {
    fmt.Printf("%d: %d\n", i, <-randStream)
}
```

❶ Here we print out a message when the goroutine successfully terminates.

Running this code produces:

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
```

You can see from the output that the deferred `fmt.Println` statement never gets run. After the third iteration of our loop, our goroutine blocks trying to send the next random integer to a channel that is no longer being read from. We have no way of telling the producer it can stop. The solution, just like for the receiving case, is to provide the producer goroutine with a channel informing it to exit:

```
newRandStream := func(done <-chan interface{}) <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited.")
        defer close(randStream)
        for {
            select {
            case randStream <- rand.Int():
            case <-done:
                return
            }
        }
    }()
```

```
    return randStream
}

done := make(chan interface{})
randStream := newRandStream(done)
fmt.Println("3 random ints:")
for i := 1; i <= 3; i++ {
    fmt.Printf("%d: %d\n", i, <-randStream)
}
close(done)

// Simulate ongoing work
time.Sleep(1 * time.Second)
```

This code produces:

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
newRandStream closure exited.
```

We see now that the goroutine is being properly cleaned up.

Now that we know how to ensure goroutines don't leak, we can stipulate a convention: *If a goroutine is responsible for creating a goroutine, it is also responsible for ensuring it can stop the goroutine.*

This convention will help ensure your programs are composable and scale as they grow. We'll revisit this technique and rule more in the sections "Pipelines" on page 100 and "The context Package" on page 131. How we ensure goroutines are able to be stopped can differ depending on the type and purpose of goroutine, but they all build on the foundation of passing in a `done` channel.

The or-channel

At times you may find yourself wanting to combine one or more `done` channels into a single `done` channel that closes if any of its component channels close. It is perfectly acceptable, albeit verbose, to write a `select` statement that performs this coupling; however, sometimes you can't know the number of `done` channels you're working with at runtime. In this case, or if you just prefer a one-liner, you can combine these channels together using the *or-channel* pattern.

This pattern creates a composite `done` channel through recursion and goroutines. Let's have a look:

```
var or func(channels ...<-chan interface{}) <-chan interface{}
or = func(channels ...<-chan interface{}) <-chan interface{} { ❶
    switch len(channels) {
    case 0: ❷
```

```
        return nil
    case 1: ❸
        return channels[0]
    }

    orDone := make(chan interface{})
    go func() { ❹
        defer close(orDone)

        switch len(channels) {
        case 2: ❺
            select {
            case <-channels[0]:
            case <-channels[1]:
            }
        default: ❻
            select {
            case <-channels[0]:
            case <-channels[1]:
            case <-channels[2]:
            case <-or(append(channels[3:], orDone)...): ❻
            }
        }
    }()
    return orDone
}
```

❶ Here we have our function, `or`, which takes in a variadic slice of channels and returns a single channel.

❷ Since this is a recursive function, we must set up termination criteria. The first is that if the variadic slice is empty, we simply return a nil channel. This is consistant with the idea of passing in no channels; we wouldn't expect a composite channel to do anything.

❸ Our second termination criteria states that if our variadic slice only contains one element, we just return that element.

❹ Here is the main body of the function, and where the recursion happens. We create a goroutine so that we can wait for messages on our channels without blocking.

❺ Because of how we're recursing, every recursive call to `or` will at least have two channels. As an optimization to keep the number of goroutines constrained, we place a special case here for calls to `or` with only two channels.

❻ Here we recursively create an or-channel from all the channels in our slice after the third index, and then select from this. This recurrence relation will destruc-

ture the rest of the slice into or-channels to form a tree from which the first signal will return. We also pass in the `orDone` channel so that when goroutines up the tree exit, goroutines down the tree also exit.

This is a fairly concise function that enables you to combine any number of channels together into a single channel that will close as soon as any of its component channels are closed, or written to. Let's take a look at how we can use this function. Here's a brief example that takes channels that close after a set duration, and uses the `or` function to combine these into a single channel that closes:

```
sig := func(after time.Duration) <-chan interface{}{ ❶
    c := make(chan interface{})
    go func() {
        defer close(c)
        time.Sleep(after)
    }()
    return c
}

start := time.Now() ❷
<-or(
    sig(2*time.Hour),
    sig(5*time.Minute),
    sig(1*time.Second),
    sig(1*time.Hour),
    sig(1*time.Minute),
)
fmt.Printf("done after %v", time.Since(start)) ❸
```

❶ This function simply creates a channel that will close when the time specified in the `after` elapses.

❷ Here we keep track of roughly when the channel from the `or` function begins to block.

❸ And here we print the time it took for the read to occur.

If you run this program you will get:

```
done after 1.000216772s
```

Notice that despite placing several channels in our call to `or` that take various times to close, our channel that closes after one second causes the entire channel created by the call to `or` to close. This is because—despite its place in the tree the `or` function builds—it will always close first and thus the channels that depend on its closure will close as well.

We achieve this terseness at the cost of additional goroutines—$f(x)=\lfloor x/2 \rfloor$ where `x` is the number of goroutines—but remember that one of Go's strengths is the ability to

quickly create, schedule, and run goroutines, and the language actively encourages using goroutines to model problems correctly. Worrying about the number of goroutines created here is probably a premature optimization. Further, if at compile time you don't know how many `done` channels you're working with, there isn't any other way to combine `done` channels.

This pattern is useful to employ at the intersection of modules in your system. At these intersections, you tend to have multiple conditions for canceling trees of goroutines through your call stack. Using the `or` function, you can simply combine these together and pass it down the stack. We'll take a look at another way of doing this in "The context Package" on page 131 that is also very nice, and perhaps a bit more descriptive.

We'll also look at how we can use a variation of this pattern to form a more complicated pattern in "Replicated Requests" on page 172.

Error Handling

In concurrent programs, error handling can be difficult to get right. Sometimes, we spend so much time thinking about how our various processes will be sharing information and coordinating, we forget to consider how they'll gracefully handle errored states. When Go eschewed the popular exception model of errors, it made a statement that error handling was important, and that as we develop our programs, we should give our error paths the same attention we give our algorithms. In that spirit, let's take a look at how we do that when working with multiple concurrent processes.

The most fundamental question when thinking about error handling is, "Who should be responsible for handling the error?" At some point, the program needs to stop ferrying the error up the stack and actually do something with it. What is responsible for this?

With concurrent processes, this question becomes a little more complex. Because a concurrent process is operating independently of its parent or siblings, it can be difficult for it to reason about what the right thing to do with the error is. Take a look at the following code for an example of this issue:

```
checkStatus := func(
  done <-chan interface{},
  urls ...string,
) <-chan *http.Response {
    responses := make(chan *http.Response)
    go func() {
        defer close(responses)
        for _, url := range urls {
            resp, err := http.Get(url)
            if err != nil {
                fmt.Println(err) ❶
```

```
                continue
            }
            select {
            case <-done:
                return
            case responses <- resp:
            }
        }
    }()
    return responses
}

done := make(chan interface{})
defer close(done)

urls := []string{"https://www.google.com", "https://badhost"}
for response := range checkStatus(done, urls...) {
    fmt.Printf("Response: %v\n", response.Status)
}
```

❶ Here we see the goroutine doing its best to signal that there's an error. What else can it do? It can't pass it back! How many errors is too many? Does it continue making requests?

Running this code produces:

```
Response: 200 OK
Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53: no such host
```

Here we see that the goroutine has been given no choice in the matter. It can't simply swallow the error, and so it does the only sensible thing: it prints the error and hopes something is paying attention. Don't put your goroutines in this awkward position. I suggest you separate your concerns: in general, your concurrent processes should send their errors to another part of your program that has complete information about the state of your program, and can make a more informed decision about what to do. The following example demonstrates a correct solution to this problem:

```
type Result struct { ❶
    Error error
    Response *http.Response
}
checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result { ❷
    results := make(chan Result)
    go func() {
        defer close(results)

        for _, url := range urls {
            var result Result
            resp, err := http.Get(url)
            result = Result{Error: err, Response: resp} ❸
            select {
```

```
            case <-done:
                return
            case results <- result: ❹
            }
        }
    }()
    return results
}

done := make(chan interface{})
defer close(done)

urls := []string{"https://www.google.com", "https://badhost"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil { ❺
        fmt.Printf("error: %v", result.Error)
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

❶ Here we create a type that encompasses both the `*http.Response` and the `error` possible from an iteration of the loop within our goroutine.

❷ This line returns a channel that can be read from to retrieve results of an iteration of our loop.

❸ Here we create a `Result` instance with the `Error` and `Response` fields set.

❹ This is where we write the `Result` to our channel.

❺ Here, in our main goroutine, we are able to deal with errors coming out of the goroutine started by `checkStatus` intelligently, and with the full context of the larger program.

This code produces:

```
Response: 200 OK
error: Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53:
no such host
```

The key thing to note here is how we've coupled the potential result with the potential error. This represents the complete set of possible outcomes created from the goroutine `checkStatus`, and allows our main goroutine to make decisions about what to do when errors occur. In broader terms, we've successfully separated the concerns of error handling from our producer goroutine. This is desirable because the goroutine that spawned the producer goroutine—in this case our main goroutine—has more context about the running program, and can make more intelligent decisions about what to do with errors.

In the previous example, we simply wrote errors out to `stdio`, but we could do something else. Let's alter our program slightly so that it stops trying to check for status if three or more errors occur:

```
done := make(chan interface{})
defer close(done)

errCount := 0
urls := []string{"a", "https://www.google.com", "b", "c", "d"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil {
        fmt.Printf("error: %v\n", result.Error)
        errCount++
        if errCount >= 3 {
            fmt.Println("Too many errors, breaking!")
            break
        }
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

This code produces this output:

```
error: Get a: unsupported protocol scheme ""
Response: 200 OK
error: Get b: unsupported protocol scheme ""
error: Get c: unsupported protocol scheme ""
Too many errors, breaking!
```

You can see that because errors are returned from `checkStatus` and not handled internally within the goroutine, error handling follows the familiar Go pattern. This is a simple example, but it's not hard to imagine situations where the main goroutine is coordinating results from multiple goroutines and building up more complex rules for continuing or canceling child goroutines. Again, the main takeaway here is that errors should be considered first-class citizens when constructing values to return from goroutines. If your goroutine can produce errors, those errors should be tightly coupled with your result type, and passed along through the same lines of communication—just like regular synchronous functions.

Pipelines

When you write a program, you probably don't sit down and write one long function—at least I hope you don't! You construct abstractions in the form of functions, structs, methods, etc. Why do we do this? Partly to abstract away details that don't matter to the greater flow, and partly so that we can work on one area of code without affecting other areas. Have you ever had to make a change to a system and found

yourself having to touch multiple areas just to make one logical change? It might be because that system suffers from poor abstraction.

A *pipeline* is just another tool you can use to form an abstraction in your system. In particular, it is a very powerful tool to use when your program needs to process streams, or batches of data. The word pipeline is believed to have first been used in 1856, and likely referred to a line of pipes that transported liquid from one place to another. We borrow this term in computer science because we're also transporting something from one place to another: data. A pipeline is nothing more than a series of things that take data in, perform an operation on it, and pass the data back out. We call each of these operations a *stage* of the pipeline.

By using a pipeline, you separate the concerns of each stage, which provides numerous benefits. You can modify stages independent of one another, you can mix and match how stages are combined independent of modifying the stages, you can process each stage concurrent to upstream or downstream stages, and you can *fan-out*, or *rate-limit* portions of your pipeline. We'll cover fan-out in the section "Fan-Out, Fan-In" on page 114, and we'll cover rate-limiting in Chapter 5. You don't have to worry about what these terms mean right now; let's start simple and just try and construct a pipeline stage.

As mentioned previously, a stage is just something that takes data in, performs a transformation on it, and sends the data back out. Here is a function that could be considered a pipeline stage:

```
multiply := func(values []int, multiplier int) []int {
    multipliedValues := make([]int, len(values))
    for i, v := range values {
        multipliedValues[i] = v * multiplier
    }
    return multipliedValues
}
```

This function takes a slice of integers in with a multiplier, loops through them multiplying as it goes, and returns a new transformed slice out. Looks like a boring function, right? Let's create another stage:

```
add := func(values []int, additive int) []int {
    addedValues := make([]int, len(values))
    for i, v := range values {
        addedValues[i] = v + additive
    }
    return addedValues
}
```

Another boring function! This one just creates a new slice and adds a value to each element. At this point, you might be wondering what makes these two functions pipeline stages and not just functions. Let's try combining them:

```
ints := []int{1, 2, 3, 4}
for _, v := range add(multiply(ints, 2), 1) {
    fmt.Println(v)
}
```

This code produces:

```
3
5
7
9
```

Look at how we combine `add` and `multiply` within the `range` clause. These are functions just like the ones you work with every day, but because we constructed them to have the properties of a pipeline stage, we're able to combine them to form a pipeline. That's interesting; what *are* the properties of a pipeline stage?

- A stage consumes and returns the same type.
- A stage must be reified[2] by the language so that it may be passed around. Functions in Go are reified and fit this purpose nicely.

Those of you familiar with functional programming may be nodding your head and thinking of terms like *higher order functions* and *monads*. Indeed, pipeline stages are very closely related to functional programming and can be considered a subset of monads. I won't go into monads or functional programming explicitly here, but they are interesting topics in their own right, and working knowledge of both topics is useful, although unnecessary, to draw on when trying to understand pipelines.

Here, our `add` and `multiply` stages satisfy all the properties of a pipeline stage: they both consume a slice of `int` and return a slice of `int`, and because Go has reified functions, we can pass `add` and `multiple` around. These properties give rise to the interesting properties of pipeline stages we mentioned earlier: namely it becomes very easy to combine our stages at a higher level without modifying the stages themselves.

For example, if we wanted to now add an additional stage to our pipeline to multiply by two, we'd simply wrap our previous pipeline in a new `multiply` stage, like so:

```
ints := []int{1, 2, 3, 4}
for _, v := range multiply(add(multiply(ints, 2), 1), 2) {
    fmt.Println(v)
}
```

2 Within the context of languages, reification means that the language exposes a concept to the developers so that they can work with it directly. Functions in Go are said to be reified because you can define variables that have a type of a function signature. This also means you can pass functions around your program.

Running this code produces:

```
6
10
14
18
```

Notice how we were able to do this without writing a new function, modifying any of the existing ones, or modifying what we do with the result of our pipeline. Maybe you're beginning to see the benefits of using the pipeline pattern. Of course we could write this code procedurally as well:

```
ints := []int{1, 2, 3, 4}
for _, v := range ints {
    fmt.Println(2*(v*2+1))
}
```

Initially, this looks much simpler, but as you'll see as we go along, the procedural code doesn't provide the same benefits a pipeline does when dealing with streams of data.

Notice how each stage is taking a slice of data and returning a slice of data? These stages are performing what we call *batch processing*. This just means that they operate on chunks of data all at once instead of one discrete value at a time. There is another type of pipeline stage that performs *stream processing*. This means that the stage receives and emits one element at a time.

There are pros and cons to batch processing versus stream processing, which we'll discuss in just a bit. For now, notice that for the original data to remain unaltered, each stage has to make a new slice of equal length to store the results of its calculations. That means that the memory footprint of our program at any one time is double the size of the slice we send into the start of our pipeline. Let's convert our stages to be stream oriented and see what that looks like:

```
multiply := func(value, multiplier int) int {
    return value * multiplier
}

add := func(value, additive int) int {
    return value + additive
}

ints := []int{1, 2, 3, 4}
for _, v := range ints {
    fmt.Println(multiply(add(multiply(v, 2), 1), 2))
}
```

This code produces:

```
6
10
14
18
```

Each stage is receiving and emitting a discrete value, and the memory footprint of our program is back down to only the size of the pipeline's input. But we had to pull the pipeline down into the body of the `for` loop and let the `range` do the heavy lifting of feeding our pipeline. Not only does this limit the reuse of how we feed the pipeline, but as we'll see later in this section, it also limits our ability to scale. We have other problems too. Effectively, we're instantiating our pipeline for every iteration of the loop. Though it's cheap to make function calls, we're making three function calls for each iteration of the loop. And what about concurrency? I stated earlier that one of the benefits of utilizing pipelines was the ability to process individual stages concurrently, and I mentioned something about *fan-out*. Where does all that come in?

I could probably extend our `multiply` and `add` functions a little more to introduce these concepts, but they've done their job of introducing the concept of a pipeline. It's time to begin learning what best practices exist for constructing pipelines in Go, and it begins with Go's *channel* primitive.

Best Practices for Constructing Pipelines

Channels are uniquely suited to constructing pipelines in Go because they fulfill all of our basic requirements. They can receive and emit values, they can safely be used concurrently, they can be ranged over, and they are reified by the language. Let's take a moment and convert the previous example to utilize channels instead:

```
generator := func(done <-chan interface{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

multiply := func(
  done <-chan interface{},
  intStream <-chan int,
  multiplier int,
) <-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
            case <-done:
```

```
                return
            case multipliedStream <- i*multiplier:
            }
        }
    }()
    return multipliedStream
}

add := func(
  done <-chan interface{},
  intStream <-chan int,
  additive int,
) <-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case addedStream <- i+additive:
            }
        }
    }()
    return addedStream
}

done := make(chan interface{})
defer close(done)

intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)

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

This code produces:

```
6
10
14
18
```

It looks like we've replicated the desired output, but at the cost of having a lot more code. What exactly have we gained? First, let's examine what we've written. We now have three functions instead of two. They all look like they start one goroutine inside their bodies, and use the pattern we established in "Preventing Goroutine Leaks" on page 90 of taking in a channel to signal that the goroutine should exit. They all look like they return channels, and some of them look like they take in an additional channel as well. Interesting! Let's start breaking this down further:

```
done := make(chan interface{})
defer close(done)
```

The first thing our program does is create a `done` channel and call `close` on it in a `defer` statement. As discussed previously, this ensures our program exits cleanly and never leaks goroutines. Nothing new there. Next, let's take a look at the function, `generator`:

```
generator := func(done <-chan interface{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

// ...

intStream := generator(done, 1, 2, 3, 4)
```

The `generator` function takes in a variadic slice of integers, constructs a buffered channel of integers with a length equal to the incoming integer slice, starts a goroutine, and returns the constructed channel. Then, on the goroutine that was created, `generator` ranges over the variadic slice that was passed in and sends the slices' values on the channel it created.

Note that the send on the channel shares a `select` statement with a selection on the `done` channel. Again, this is the pattern we established in "Preventing Goroutine Leaks" on page 90 to guard against leaking goroutines.

So in a nutshell, the `generator` function converts a discrete set of values into a stream of data on a channel. Aptly, this type of function is called a *generator*. You'll see this frequently when working with pipelines because at the beginning of the pipeline, you'll always have some batch of data that you need to convert to a channel. We'll go over a few examples of some fun generators in just a bit, but let's finish our analysis of this program first. Next, we construct our pipeline:

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

It's the same pipeline we've been working with all along: for a stream of numbers, we'll multiply them by two, add one, and then multiply the result by two. This pipeline is similar to our pipeline utilizing functions in the previous example, but it is different in very important ways.

First, we're using channels. This is obvious but significant because it allows two things: at the end of our pipeline, we can use a range statement to extract the values, and at each stage we can safely execute concurrently because our inputs and outputs are safe in concurrent contexts.

Which brings us to our second difference: each stage of the pipeline is executing concurrently. This means that any stage only need wait for its inputs, and to be able to send its outputs. This turns out to have massive ramifications as we'll discover in the section "Fan-Out, Fan-In" on page 114, but for now we can simply note that it allows our stages to execute independent of one another for some slice of time.

Finally, in our example, we range over this pipeline and values are pulled through the system:

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

Here is a table demonstrating how each of the values in the system will enter each channel, and when the channels will be closed. Iteration is the base-zero count of what iteration of the `for` loop we're on, and the value for each column is the value as it comes into the pipeline stage:

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
0	3		4		6
1		3		5	
1	4		6		10
2	(closed)	4		7	
2		(closed)	8		14
3			(closed)	9	
3				(closed)	18

Let's also examine more closely our use of the pattern to signal goroutines to exit. When we're dealing with multiple interdependent goroutines, how does this pattern end up working? What would happen if we called `close` on the `done` channel before the program was finished executing?

To answer these questions, let's take a look at our pipeline construction one more time:

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

The stages are interconnected in two ways: by the common `done` channel, and by the channels that are passed into subsequent stages of the pipeline. In other words, the channel created by the `multiply` function is passed into the `add` function, and so forth. Let's revisit the preceding table and, before allowing it to complete, call `close` on the `done` channel and see what happens:

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
1	3		4		6
close(done)	(closed)	3		5	
		(closed)	6		
			(closed)	7	
				(closed)	
					(exit range)

See how closing the `done` channel cascades through the pipeline? This is made possible by two things in each stage of the pipeline:

- Ranging over the incoming channel. When the incoming channel is closed, the range will exit.
- The send sharing a `select` statement with the `done` channel.

Regardless of what state the pipeline stage is in—waiting on the incoming channel, or waiting on the send—closing the `done` channel will force the pipeline stage to terminate.

There is a recurrence relation at play here. At the beginning of the pipeline, we've established that we must convert discrete values into a channel. There are two points in this process that *must* be preemptable:

- Creation of the discrete value that is not nearly instantaneous.
- Sending of the discrete value on its channel.

The first is up to you. In our example, in the `generator` function, the discrete values are generated by ranging over the variadic slice, which is instantaneous enough that it doesn't need to be preemptable. The second is handled via our `select` statement and `done` channel, which ensures that `generator` is preemptable even if it is blocked attempting to write to `intStream`.

On the other end of the pipeline, the final stage is ensured preemptability by induction. It is preemptable because the channel we're ranging over will be closed when preempted, and therefore our range will break when this occurs. The final stage is preemptable because the stream we rely on is preemptable.

In between the beginning of the pipeline and the end of the pipeline, the code is always ranging over a channel and sending on another channel within a `select` statement containing a `done` channel.

If a stage is blocked on retrieving a value from the incoming channel, it will become unblocked when that channel is closed. We know by induction that the channel will be closed because it is either a stage written like the stage we are within, or the beginning of the pipeline that we have established is preemptable. If a stage is blocked on sending a value, it is preemptable thanks to the `select` statement.

Thus, our entire pipeline is always preemptable by closing the `done` channel. Cool, right?

Some Handy Generators

I promised earlier I would talk about some fun generators that might be widely useful. As a reminder, a generator for a pipeline is any function that converts a set of discrete values into a stream of values on a channel. Let's take a look at a generator called `repeat`:

```
repeat := func(
    done <-chan interface{},
    values ...interface{},
) <-chan interface{} {
    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            for _, v := range values {
                select {
                case <-done:
                    return
                case valueStream <- v:
                }
            }
        }
    }()
    return valueStream
}
```

This function will repeat the values you pass to it infinitely until you tell it to stop. Let's take a look at another generic pipeline stage that is helpful when used in combination with `repeat`, `take`:

```
take := func(
    done <-chan interface{},
    valueStream <-chan interface{},
    num int,
) <-chan interface{} {
    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        for i := 0; i < num; i++ {
            select {
            case <-done:
                return
            case takeStream <- <- valueStream:
            }
        }
    }()
    return takeStream
}
```

This pipeline stage will only take the first `num` items off of its incoming `valueStream` and then exit. Together, the two can be very powerful:

```
done := make(chan interface{})
defer close(done)

for num := range take(done, repeat(done, 1), 10) {
    fmt.Printf("%v ", num)
}
```

Running this code produces:

```
1 1 1 1 1 1 1 1 1 1
```

In this basic example, we create a `repeat` generator to generate an infinite number of ones, but then only take the first 10. Because the `repeat` generator's send blocks on the `take` stage's receive, the `repeat` generator is very efficient. Although we have the capability of generating an infinite stream of ones, we only generate `N+1` instances where `N` is the number we pass into the `take` stage.

We can expand on this. Let's create another repeating generator, but this time, let's create one that repeatedly calls a function. Let's call it `repeatFn`:

```
repeatFn := func(
    done <-chan interface{},
    fn func() interface{},
) <-chan interface{} {
    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            select {
            case <-done:
```

```
                return
            case valueStream <- fn():
            }
        }
    }()
    return valueStream
}
```

Let's use it to generate 10 random numbers:

```
done := make(chan interface{})
defer close(done)

rand := func() interface{} { return rand.Int()}

for num := range take(done, repeatFn(done, rand), 10) {
    fmt.Println(num)
}
```

This produces:

```
5577006791947779410
8674665223082153551
6129484611666145821
4037200794235010051
3916589616287113937
6334824724549167320
605394647632969758
1443635317331776148
894385949183117216
2775422040480279449
```

That's pretty cool—an infinite channel of random integers generated on an as-needed basis!

You may be wondering why all of these generators and stages are receiving and sending on channels of `interface{}`. We could have just as easily written these functions to be specific to a type, or maybe written a Go generator.

Empty interfaces are a bit taboo in Go, but for pipeline stages it is my opinion that it's OK to deal in channels of `interface{}` so that you can use a standard library of pipeline patterns. As we discussed earlier, a lot of a pipeline's utility comes from reusable stages. This is best achieved when the stages operate at the level of specificity appropriate to itself. In the `repeat` and `repeatFn` generators, the concern is generating a stream of data by looping over a list or operator. With the `take` stage, the concern is limiting our pipeline. None of these operations require information about the types they're working on, but instead only require knowledge of the arity of their parameters.

When you need to deal in specific types, you can place a stage that performs the type assertion for you. The performance overhead of having an extra pipeline stage (and

thus goroutine) and the type assertion are negligible, as we'll see in just a bit. Here's a small example that introduces a `toString` pipeline stage:

```
toString := func(
    done <-chan interface{},
    valueStream <-chan interface{},
) <-chan string {
    stringStream := make(chan string)
    go func() {
        defer close(stringStream)
        for v := range valueStream {
            select {
            case <-done:
                return
            case stringStream <- v.(string):
            }
        }
    }()
    return stringStream
}
```

And an example of how to use it:

```
done := make(chan interface{})
defer close(done)

var message string
for token := range toString(done, take(done, repeat(done, "I", "am."), 5)) {
    message += token
}

fmt.Printf("message: %s...", message)
```

This code produces:

```
message: Iam.Iam.I...
```

So let's prove to ourselves that the performance cost of genericizing portions of our pipeline is negligible. We'll write two benchmarking functions: one to test the generic stages, and one to test the type-specific stages:

```
func BenchmarkGeneric(b *testing.B) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for range toString(done, take(done, repeat(done, "a"), b.N)) {
    }
}

func BenchmarkTyped(b *testing.B) {
    repeat := func(done <-chan interface{}, values ...string) <-chan string {
        valueStream := make(chan string)
        go func() {
```

```
                defer close(valueStream)
                for {
                    for _, v := range values {
                        select {
                        case <-done:
                            return
                        case valueStream <- v:
                        }
                    }
                }
            }()
            return valueStream
        }

        take := func(
            done <-chan interface{},
            valueStream <-chan string,
            num int,
        ) <-chan string {
            takeStream := make(chan string)
            go func() {
                defer close(takeStream)
                for i := num; i > 0 || i == -1; {
                    if i != -1 {
                        i--
                    }
                    select {
                    case <-done:
                        return
                    case takeStream <- <-valueStream:
                    }
                }
            }()
            return takeStream
        }

        done := make(chan interface{})
        defer close(done)

        b.ResetTimer()
        for range take(done, repeat(done, "a"), b.N) {
        }
    }
```

And the results from running this code are:

```
BenchmarkGeneric-4      1000000     2266    ns/op
BenchmarkTyped-4        1000000     1181    ns/op
PASS
ok      command-line-arguments      3.486s
```

You can see that the type-specific stages are twice as fast, but only marginally faster in magnitude. Generally, the limiting factor on your pipeline will either be your generator, or one of the stages that is computationally intensive. If the generator isn't creating a stream from memory as with the `repeat` and `repeatFn` generators, you'll probably be I/O bound. Reading from disk or the network will likely eclipse the meager performance overhead shown here.

If one of your stages is computationally expensive, this will *certainly* eclipse this performance overhead. If this technique still leaves a bad taste in your mouth, you can always write a Go generator for creating your generator stages. Speaking of one stage being computationally expensive, how can we help mitigate this? Won't it rate-limit the entire pipeline?

For ways to help mitigate this, let's discuss the fan-out, fan-in technique.

Fan-Out, Fan-In

So you've got a pipeline set up. Data is flowing through your system beautifully, transforming as it makes its way through the stages you've chained together. It's like a beautiful stream; a beautiful, slow stream, and oh my god why is this taking so long?

Sometimes, stages in your pipeline can be particularly computationally expensive. When this happens, upstream stages in your pipeline can become blocked while waiting for your expensive stages to complete. Not only that, but the pipeline itself can take a long time to execute as a whole. How can we address this?

One of the interesting properties of pipelines is the ability they give you to operate on the stream of data using a combination of separate, often reorderable stages. You can even reuse stages of the pipeline multiple times. Wouldn't it be interesting to reuse a single stage of our pipeline on multiple goroutines in an attempt to parallelize pulls from an upstream stage? Maybe that would help improve the performance of the pipeline.

In fact, it turns out it can, and this pattern has a name: *fan-out, fan-in*.

Fan-out is a term to describe the process of starting multiple goroutines to handle input from the pipeline, and fan-in is a term to describe the process of combining multiple results into one channel.

So what makes a stage of a pipeline suited for utilizing this pattern? You might consider fanning out one of your stages if both of the following apply:

- It doesn't rely on values that the stage had calculated before.
- It takes a long time to run.

The property of order-independence is important because you have no guarantee in what order concurrent copies of your stage will run, nor in what order they will return.

Let's take a look at an example. In the following example, I've constructed a very inefficient way to find primes. We'll use a lot of the stages we created in "Pipelines" on page 100:

```
rand := func() interface{} { return rand.Intn(50000000) }

done := make(chan interface{})
defer close(done)

start := time.Now()

randIntStream := toInt(done, repeatFn(done, rand))
fmt.Println("Primes:")
for prime := range take(done, primeFinder(done, randIntStream), 10) {
    fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

Here are the results of running this code:

```
Primes:
    24941317
    36122539
    6410693
    10128161
    25511527
    2107939
    14004383
    7190363
    45931967
    2393161
Search took: 23.437511647s
```

We're generating a stream of random numbers, capped at 50,000,000, converting the stream into an integer stream, and then passing that into our `primeFinder` stage. `primeFinder` naively begins to attempt to divide the number provided on the input stream by every number below it. If it's unsuccessful, it passes the value on to the next stage. Certainly, this is a horrible way to try and find prime numbers, but it fulfills our requirement of taking a *long* time.

In our `for` loop, we range over the found primes, print them out as they come in, and —thanks to our `take` stage—close the pipeline after 10 primes are found. We then print out how long the search took, and the `done` channel is closed by a `defer` statement and the pipeline is torn down.

To avoid duplicates in our results, we could introduce another stage in our pipeline to cache the primes that have been found in a set, but for simplicity, we'll just ignore these.

You can see it took roughly 23 seconds to find 10 primes. Not great. Normally we'd first look at the algorithm itself, maybe grab an algorithm cookbook, and see if we could improve things in each stage. But as the purpose of the stage here is to be slow, we'll instead look at how we can *fan-out* one or more of the stages to chew through slow operations more quickly.

This is a relatively simple example, so we only have two stages: random number generation and prime sieving. In a larger program, your pipeline might be composed of many more stages; how do we know which one to fan out? Remember our criteria from earlier: order-independence and duration. Our random integer generator is certainly order-independent, but it doesn't take a particularly long time to run. The `primeFinder` stage is also order-independent—numbers are either prime or not—and because of our naive algorithm, it certainly takes a long time to run. It looks like a good candidate for fanning out.

Fortunately the process of fanning out a stage in a pipeline is extraordinarily easy. All we have to do is start multiple versions of that stage. So instead of this:

```
primeStream := primeFinder(done, randIntStream)
```

We can do something like this:

```
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```

Here we're starting up as many copies of this stage as we have CPUs. On my computer, `runtime.NumCPU()` returns eight, so I'll continue to use this number in our discussion. In production, we would probably do a little empirical testing to determine the optimal number of CPUs, but here we'll stay simple and assume that a CPU will be kept busy by only one copy of the `findPrimes` stage.

And that's it! We now have eight goroutines pulling from the random number generator and attempting to determine whether the number is prime. Generating random numbers shouldn't take much time, and so each goroutine for the `findPrimes` stage should be able to determine whether its number is prime and then have another random number available to it immediately.

We still have a problem though: now that we have four goroutines, we also have four channels, but our range over primes is only expecting one channel. This brings us to the *fan-in* portion of the pattern.

As we discussed earlier, fanning in means *multiplexing* or joining together multiple streams of data into a single stream. The algorithm to do so is relatively simple:

```
fanIn := func(
    done <-chan interface{},
    channels ...<-chan interface{},
) <-chan interface{} { ❶
    var wg sync.WaitGroup ❷
    multiplexedStream := make(chan interface{})

    multiplex := func(c <-chan interface{}) { ❸
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }

    // Select from all the channels
    wg.Add(len(channels)) ❹
    for _, c := range channels {
        go multiplex(c)
    }

    // Wait for all the reads to complete
    go func() { ❺
        wg.Wait()
        close(multiplexedStream)
    }()

    return multiplexedStream
}
```

❶ Here we take in our standard `done` channel to allow our goroutines to be torn down, and then a variadic slice of `interface{}` channels to fan-in.

❷ On this line we create a `sync.WaitGroup` so that we can wait until all channels have been drained.

❸ Here we create a function, `multiplex`, which, when passed a channel, will read from the channel, and pass the value read onto the `multiplexedStream` channel.

❹ This line increments the `sync.WaitGroup` by the number of channels we're multiplexing.

❺ Here we create a goroutine to wait for all the channels we're multiplexing to be drained so that we can close the `multiplexedStream` channel.

In a nutshell, fanning in involves creating the multiplexed channel consumers will read from, and then spinning up one goroutine for each incoming channel, and one goroutine to close the multiplexed channel when the incoming channels have all been closed. Since we're going to be creating a goroutine that is waiting on `N` other goroutines to complete, it makes sense to create a `sync.WaitGroup` to coordinate things. The `multiplex` function also notifies the `WaitGroup` that it's done.

An Additional Reminder

A naive implementation of the fan-in, fan-out algorithm only works if the order in which results arrive is unimportant. We have done nothing to guarantee that the order in which items are read from the `randIntStream` is preserved as it makes its way through the sieve. Later, we'll look at an example of a way to maintain order.

Let's put all of this together and see if we get any decrease in runtime:

```
done := make(chan interface{})
defer close(done)

start := time.Now()

rand := func() interface{} { return rand.Intn(50000000) }

randIntStream := toInt(done, repeatFn(done, rand))

numFinders := runtime.NumCPU()
fmt.Printf("Spinning up %d prime finders.\n", numFinders)
finders := make([]<-chan interface{}, numFinders)
fmt.Println("Primes:")
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}

for prime := range take(done, fanIn(done, finders...), 10) {
    fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

Here are the results:

```
Spinning up 8 prime finders.
Primes:
    6410693
    24941317
```

```
    10128161
    36122539
    25511527
    2107939
    14004383
    7190363
    2393161
    45931967
Search took: 5.438491216s
```

So down from ~23 seconds to ~5 seconds, not bad! This clearly demonstrates the benefit of the fan-out, fan-in pattern, and it reiterates the utility of pipelines. We cut our execution time by ~78% without drastically altering the structure of our program.

The or-done-channel

At times you will be working with channels from disparate parts of your system. Unlike with pipelines, you can't make any assertions about how a channel will behave when code you're working with is canceled via its `done` channel. That is to say, you don't know if the fact that your goroutine was canceled means the channel you're reading from will have been canceled. For this reason, as we laid out in "Preventing Goroutine Leaks" on page 90, we need to wrap our read from the channel with a `select` statement that also selects from a `done` channel. This is perfectly fine, but doing so takes code that's easily read like this:

```
for val := range myChan {
    // Do something with val
}
```

And explodes it out into this:

```
loop:
for {
    select {
    case <-done:
        break loop
    case maybeVal, ok := <-myChan:
        if ok == false {
            return // or maybe break from for
        }
        // Do something with val
    }
}
```

This can get busy quite quickly—especially if you have nested loops. Continuing with the theme of utilizing goroutines to write clearer concurrent code, and not prematurely optimizing, we can fix this with a single goroutine. We encapsulate the verbosity so that others don't have to:

```
orDone := func(done, c <-chan interface{}) <-chan interface{} {
    valStream := make(chan interface{})
    go func() {
        defer close(valStream)
        for {
            select {
            case <-done:
                return
            case v, ok := <-c:
                if ok == false {
                    return
                }
                select {
                case valStream <- v:
                case <-done:
                }
            }
        }
    }()
    return valStream
}
```

Doing this allows us to get back to simple `for` loops, like so:

```
for val := range orDone(done, myChan) {
    // Do something with val
}
```

You may find edge cases in your code where you need a tight loop utilizing a series of `select` statements, but I would encourage you to try for readability first, and avoid premature optimization.

The tee-channel

Sometimes you may want to split values coming in from a channel so that you can send them off into two separate areas of your codebase. Imagine a channel of user commands: you might want to take in a stream of user commands on a channel, send them to something that executes them, and also send them to something that logs the commands for later auditing.

Taking its name from the `tee` command in Unix-like systems, the *tee-channel* does just this. You can pass it a channel to read from, and it will return two separate channels that will get the same value:

```
tee := func(
    done <-chan interface{},
    in <-chan interface{},
) (_, _ <-chan interface{}) { <-chan interface{}) {
    out1 := make(chan interface{})
    out2 := make(chan interface{})
    go func() {
```

```
        defer close(out1)
        defer close(out2)
        for val := range orDone(done, in) {
            var out1, out2 = out1, out2 ❶
            for i := 0; i < 2; i++ { ❷
                select {
                case <-done:
                case out1<-val:
                    out1 = nil ❸
                case out2<-val:
                    out2 = nil ❸
                }
            }
        }
    }()
    return out1, out2
}
```

❶ We will want to use local versions of `out1` and `out2`, so we shadow these variables.

❷ We're going to use one `select` statement so that writes to `out1` and `out2` don't block each other. To ensure both are written to, we'll perform two iterations of the `select` statement: one for each outbound channel.

❸ Once we've written to a channel, we set its shadowed copy to `nil` so that further writes will block and the other channel may continue.

Notice that writes to `out1` and `out2` are tightly coupled. The iteration over `in` cannot continue until both `out1` and `out2` have been written to. Usually this is not a problem as handling the throughput of the process reading from each channel should be a concern of something other than the tee command anyway, but it's worth noting. Here's a quick example to demonstrate:

```
done := make(chan interface{})
defer close(done)

out1, out2 := tee(done, take(done, repeat(done, 1, 2), 4))

for val1 := range out1 {
    fmt.Printf("out1: %v, out2: %v\n", val1, <-out2)
}
```

Utilizing this pattern, it's easy to continue using channels as the join points of your system.

The bridge-channel

In some circumstances, you may find yourself wanting to consume values from a sequence of channels:

```
<-chan <-chan interface{}
```

This is slightly different than coalescing a slice of channels into a single channel, as we saw in "The or-channel" on page 94 or "Fan-Out, Fan-In" on page 114. A sequence of channels suggests an ordered write, albeit from different sources. One example might be a pipeline stage whose lifetime is intermittent. If we follow the patterns we established in "Confinement" on page 85 and ensure channels are owned by the goroutines that write to them, every time a pipeline stage is restarted within a new goroutine, a new channel would be created. This means we'd effectively have a sequence of channels. We'll explore this scenario more in "Healing Unhealthy Goroutines" on page 188.

As a consumer, the code may not care about the fact that its values come from a sequence of channels. In that case, dealing with a channel of channels can be cumbersome. If we instead define a function that can destructure the channel of channels into a simple channel—a technique called *bridging* the channels—this will make it much easier for the consumer to focus on the problem at hand. Here's how we can achieve that:

```
bridge := func(
    done <-chan interface{},
    chanStream <-chan <-chan interface{},
) <-chan interface{} {
    valStream := make(chan interface{}) ❶
    go func() {
        defer close(valStream)
        for { ❷
            var stream <-chan interface{}
            select {
            case maybeStream, ok := <-chanStream:
                if ok == false {
                    return
                }
                stream = maybeStream
            case <-done:
                return
            }
            for val := range orDone(done, stream) { ❸
                select {
                case valStream <- val:
                case <-done:
                }
            }
        }
```

```
    }()
    return valStream
}
```

❶ This is the channel that will return all values from `bridge`.

❷ This loop is responsible for pulling channels off of `chanStream` and providing them to a nested loop for use.

❸ This loop is responsible for reading values off the channel it has been given and repeating those values onto `valStream`. When the stream we're currently looping over is closed, we break out of the loop performing the reads from this channel, and continue with the next iteration of the loop, selecting channels to read from. This provides us with an unbroken stream of values.

This is pretty straightforward code. Now we can use `bridge` to help present a single-channel facade over a channel of channels. Here's an example that creates a series of 10 channels, each with one element written to them, and passes these channels into the `bridge` function:

```
genVals := func() <-chan <-chan interface{} {
    chanStream := make(chan (<-chan interface{}))
    go func() {
        defer close(chanStream)
        for i := 0; i < 10; i++ {
            stream := make(chan interface{}, 1)
            stream <- i
            close(stream)
            chanStream <- stream
        }
    }()
    return chanStream
}

for v := range bridge(nil, genVals()) {
    fmt.Printf("%v ", v)
}
```

Running this produces:

```
0 1 2 3 4 5 6 7 8 9
```

Thanks to `bridge`, we can use the channel of channels from within a single range statement and focus on our loop's logic. Destructuring the channel of channels is left to code that is specific to this concern.

Queuing

Sometimes it's useful to begin accepting work for your pipeline even though the pipeline is not yet ready for more. This process is called *queuing*.

All this means is that once your stage has completed some work, it stores it in a temporary location in memory so that other stages can retrieve it later, and your stage doesn't need to hold a reference to it. In the section on "Channels" on page 64, we discussed *buffered channels*, a type of queue, but we haven't really made much use of them since—and for good reason.

While introducing queuing into your system is very useful, it's usually one of the last techniques you want to employ when optimizing your program. Adding queuing prematurely can hide synchronization issues such as deadlocks and livelocks, and further, as your program converges toward correctness, you may find that you need more or less queuing.

So what is queuing good for? Let's begin to answer that question by addressing one of the common mistakes people make when trying to tune the performance of a system: introducing queues to try and address performance concerns. Queuing will almost never speed up the total runtime of your program; it will only allow the program to behave differently.

To understand why, let's take a look at a simple pipeline:

```
done := make(chan interface{})
defer close(done)

zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
long := sleep(done, 4*time.Second, short)
pipeline := long
```

This pipeline chains together four stages:

1. A repeat stage that generates an endless stream of 0s.
2. A stage that cancels the previous stages after seeing three items.
3. A "short" stage that sleeps one second.
4. A "long" stage that sleeps four seconds.

For the purposes of this example, let's assume that stages 1 and 2 are instantaneous, and let's focus on how the stages that sleep affect the runtime of the pipeline.

Here's a table examining the time `t`, the iteration `i`, and how long the long and short stages have left to move to their next value.

Time(t)	i	Long stage	Short stage
0	0		1s
1	0	4s	1s
2	0	3s	(blocked)
3	0	2s	(blocked)
4	0	1s	(blocked)
5	1	4s	1s
6	1	3s	(blocked)
7	1	2s	(blocked)
8	1	1s	(blocked)
9	2	4s	(close)
10	2	3s	
11	2	2s	
12	2	1s	
13	3	(close)	

You can see that this pipeline takes roughly 13 seconds to run. The short stage takes about 9 seconds to complete.

What happens if we modify the pipeline to include a buffer? Let's examine the same pipeline with a buffer of 2 introduced between the long and short stages:

```
done := make(chan interface{})
defer close(done)

zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
buffer := buffer(done, 2, short)    // Buffers sends from short by 2
long := sleep(done, 4*time.Second, short)
pipeline := long
```

Here's the runtime:

Time(t)	i	Long stage	Buffer	Short stage
0	0		0/2	1s
1	0	4s	0/2	1s
2	0	3s	1/2	1s
3	0	2s	2/2	(close)
4	0	1s	2/2	
5	1	4s	1/2	
6	1	3s	1/2	
7	1	2s	1/2	
8	1	1s	1/2	

Time(t)	i	Long stage	Buffer	Short stage
9	2	4s	0/2	
10	2	3s	0/2	
11	2	2s	0/2	
12	2	1s	0/2	
13	3	(close)		

The entire pipeline still took 13 seconds! But look at the short stage's runtime. It's complete after only 3 seconds as opposed to the 9 seconds it took previously. We've cut this stage's runtime by two thirds! But if the entire pipeline still takes 13 seconds to execute, how does this help us?

Picture instead the following pipeline:

```
p := processRequest(done, acceptConnection(done, httpHandler))
```

Here the pipeline doesn't exit until it's canceled, and the stage that is accepting connections doesn't stop accepting connections until the pipeline is canceled. In this scenario, you wouldn't want connections to your program to begin timing out because your `processRequest` stage was blocking your `acceptConnection` stage. You want your `acceptConnection` stage to be unblocked as much as possible. Otherwise the users of your program might begin seeing their requests denied altogether.

So the answer to our question of the utility of introducing a queue isn't that the runtime of one of stages has been reduced, but rather that the time it's in a *blocking state* is reduced. This allows the stage to continue doing its job. In this example, users would likely experience lag in their requests, but they wouldn't be denied service altogether.

In this way, the true utility of queues is to *decouple stages* so that the runtime of one stage has no impact on the runtime of another. Decoupling stages in this manner then cascades to alter the runtime behavior of the system as a whole, which can be either good or bad depending on your system.

We then come to the question of tuning your queuing. Where should the queues be placed? What should the buffer size be? The answers to these questions depend on the nature of your pipeline.

Let's begin by analyzing situations in which queuing *can* increase the overall performance of your system. The only applicable situations are:

- If batching requests in a stage saves time.
- If delays in a stage produce a feedback loop into the system.

One example of the first situation is a stage that buffers input in something faster (e.g., memory) than it is designed to send to (e.g., disk). This is, of course, the entire

purpose of Go's `bufio` package. Here's an example that demonstrates a simple comparison of a buffered write to a queue versus an unbuffered write:

```
func BenchmarkUnbufferedWrite(b *testing.B) {
    performWrite(b, tmpFileOrFatal())
}

func BenchmarkBufferedWrite(b *testing.B) {
    bufferredFile := bufio.NewWriter(tmpFileOrFatal())
    performWrite(b, bufio.NewWriter(bufferredFile))
}

func tmpFileOrFatal() *os.File {
    file, err := ioutil.TempFile("", "tmp")
    if err != nil {
        log.Fatal("error: %v", err)
    }
    return file
}

func performWrite(b *testing.B, writer io.Writer) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for bt := range take(done, repeat(done, byte(0)), b.N) {
        writer.Write([]byte{bt.(byte)})
    }
}
```

```
go test -bench=. src/concurrency-patterns-in-go/queuing/buffering_test.go
```

And here are the results of running this benchmark:

```
BenchmarkUnbufferedWrite-8      500000          3969    ns/op
BenchmarkBufferedWrite-8        1000000         1356    ns/op
PASS
ok                              command-line-arguments  3.398s
```

As anticipated, the buffered write is faster than the unbuffered write. This is because in `bufio.Writer`, the writes are *queued* internally into a buffer until a sufficient chunk has been accumulated, and then the chunk is written out. This process is often called *chunking*, for obvious reasons.

Chunking is faster because `bytes.Buffer` must grow its allocated memory to accommodate the bytes it must store. For various reasons, growing memory is expensive; therefore, the less times we have to grow, the more efficient our system as a whole will perform. Thus, queuing has increased the performance of our system as a whole.

This is only a simple in-memory example of chunking, but you may encounter chunking frequently in the field. Usually anytime performing an operation requires an overhead, chunking may increase system performance. Some examples of this are opening database transactions, calculating message checksums, and allocating contiguous space.

Aside from chunking, queuing can also help if your algorithm can be optimized by supporting lookbehinds, or ordering.

The second scenario, where a delay in a stage causes more input into the pipeline, is a little more difficult to spot, but also more important because it can lead to a systemic collapse of your upstream systems.

This idea is often referred to as a *negative feedback loop*, downward-spiral, or even death-spiral. This is because a recurrent relation exists between the pipeline and its upstream systems; the rate at which upstream stages or systems submit new requests is somehow linked to how efficient the pipeline is.

If the efficiency of the pipeline drops below a certain critical threshold, the systems upstream from the pipeline begin increasing their inputs into the pipeline, which causes the pipeline to lose more efficiency, and the death-spiral begins. Without some sort of fail-safe, the system utilizing the pipeline will never recover.

By introducing a queue at the entrance to the pipeline, you can break the feedback loop at the cost of creating lag for requests. From the perspective of the caller into the pipeline, the request appears to be processing, but taking a very long time. As long as the caller doesn't time out, your pipeline will remain stable. If the caller does time out, you need to be sure you support some kind of check for readiness when dequeuing. If you don't, you can inadvertently create a feedback loop by processing dead requests thereby decreasing the efficiency of your pipeline.

Have You Ever Witnessed a Death-Spiral?

If you've ever attempted to access some hot new system when it first came online (e.g., new game servers, websites for product launches, etc.), and the site kept bouncing despite the developer's best efforts, congratulations! You've likely witnessed a negative feedback loop.

Invariably the development team tries different things until someone realizes they need a queue, and one is hastily implemented.

Then the customers begin complaining about queue times!

So from our examples we can begin to see a pattern emerge; queuing should be implemented either:

- At the entrance to your pipeline.
- In stages where batching will lead to higher efficiency.

You may be tempted to add queuing elsewhere—e.g., after a computationally expensive stage—but avoid that temptation! As we've learned, there are only a few situations where queuing will decrease the runtime of your pipeline, and peppering in queuing in an attempt to work around this can have disastrous consequences.

This is not intuitive at first; to understand why, we have to discuss throughput of the pipeline. Don't worry, it's not that difficult, and it will also help us answer the question of how to determine how large our queues should be.

In queuing theory, there is a law that—with enough sampling—predicts the throughput of your pipeline. It's called *Little's Law*, and you only need to know a few things to understand and make use of it.

Let's first define Little's Law algebraicly. It is commonly expressed as: `L=λW`, where:

- `L` = the average number of units in the system.
- `λ` = the average arrival rate of units.
- `W` = the average time a unit spends in the system.

This equation only applies to so-called *stable* systems. In a pipeline, a stable system is one in which the rate that work enters the pipeline, or *ingress*, is equal to the rate in which it exits the system, or *egress*. If the rate of ingress exceeds the rate of egress, your system is *unstable* and has entered a *death-spiral*. If the rate of ingress is less than the rate of egress, you still have an unstable system, but all that's happening is that your resources aren't being utilized completely. Not the worst situation in the world, but maybe you care about this if the underutilization is found on a vast scale (e.g., clusters or data centers).

So let's assume that our pipeline is stable. If we want to decrease `W`, the average time a unit spends in the system by a factor of `n`, we only have one option: to decrease the average number of units in the system: `L/n = λ * W/n`. And we can only decrease the average number of units in the system if we increase the rate of egress. Also notice that if we add queues to our stages, we're increasing `L`, which either increases the arrival rate of units (`nL = nλ * W`) or increases the average time a unit spends in the system (`nL = λ * nW`). Through Little's Law, we have proven that queuing will not help decrease the amount of time spent in a system.

Also notice that since we're observing our pipeline as a whole, reducing `W` by a factor of `n` is distributed throughout all stages of our pipeline. In our case, Little's Law should really be defined like this:

$$L = \lambda \Sigma_i W_i$$

That's another way of saying that your pipeline will only be as fast as your slowest stage. Optimize indiscriminately!

So Little's Law is neat! This simple equation opens up all kinds of ways to analyze our pipeline. Let's use it to ask some interesting questions. During our analysis, let's assume our pipeline has three stages.

Let's try and determine how many requests per second our pipeline can handle. Let's assume we enable sampling on our pipeline and find that 1 request (`r`) takes about 1 second to make it through the pipeline. Let's plug in those numbers!

```
3r = λr/s * 1s

3r/s = λr/s

λr/s = 3r/s
```

We set `L` to 3 because each stage in our pipeline is processing a request. We then set `W` to 1 second, do a little algebra, and voilà! In this pipeline, we can handle three requests per second.

What about determining how large our queue needs to be to handle a desired number of requests. Can Little's Law help us answer that?

Let's say our sampling indicates that a request takes 1 ms to process. What size would our queue have to be to handle 100,000 requests per second? Again, let's plug in the numbers!

```
Lr-3r = 100,000r/s * 0.0001s

Lr-3r = 10r

Lr = 7r
```

Again, our pipeline has three stages, so we'll decrement `L` by 3. We set `λ` to 100,000 r/s, and find that if we want to field that many requests, our queue should have a capacity of 7. Remember that as you increase the queue size, it takes your work longer to make it through the system! You're effectively trading system utilization for lag.

Something that Little's Law can't provide insight on is handling failure. Keep in mind that if for some reason your pipeline panics, you'll lose all the requests in your queue. This might be something to guard against if re-creating the requests is difficult or won't happen. To mitigate this, you can either stick to a queue size of zero, or you can

move to a *persistent queue*, which is simply a queue that is persisted somewhere that can be later read from should the need arise.

Queuing can be useful in your system, but because of its complexity, it's usually one of the last optimizations I would suggest implementing.

The context Package

As we've seen, in concurrent programs it's often necessary to preempt operations because of timeouts, cancellation, or failure of another portion of the system. We've looked at the idiom of creating a `done` channel, which flows through your program and cancels all blocking concurrent operations. This works well, but it's also somewhat limited.

It would be useful if we could communicate extra information alongside the simple notification to cancel: why the cancellation was occuring, or whether or not our function has a deadline by which it needs to complete.

It turns out that the need to wrap a `done` channel with this information is very common in systems of any size, and so the Go authors decided to create a standard pattern for doing so. It started out as an experiment that lived outside the standard library, but in Go 1.7, the `context` package was brought into the standard library, making this a standard Go idiom to consider when working with concurrent code.

If we take a peek into the `context` package, we see that it's very simple:

```
var Canceled = errors.New("context canceled")
var DeadlineExceeded error = deadlineExceededError{}

type CancelFunc
type Context

func Background() Context
func TODO() Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

We'll revisit these types and functions in a bit, but for now let's focus on the `Context` type. This is the type that will flow through your system much like a `done` channel does. If you use the `context` package, each function that is downstream from your top-level concurrent call would take in a `Context` as its first argument. The type looks like this:

```
type Context interface {

    // Deadline returns the time when work done on behalf of this
    // context should be canceled. Deadline returns ok==false when no
```

```
        // deadline is set. Successive calls to Deadline return the same
        // results.
        Deadline() (deadline time.Time, ok bool)

        // Done returns a channel that's closed when work done on behalf
        // of this context should be canceled. Done may return nil if this
        // context can never be canceled. Successive calls to Done return
        // the same value.
        Done() <-chan struct{}

        // Err returns a non-nil error value after Done is closed. Err
        // returns Canceled if the context was canceled or
        // DeadlineExceeded if the context's deadline passed. No other
        // values for Err are defined.  After Done is closed, successive
        // calls to Err return the same value.
        Err() error

        // Value returns the value associated with this context for key,
        // or nil if no value is associated with key. Successive calls to
        // Value with the same key returns the same result.
        Value(key interface{}) interface{}
    }
```

This also looks pretty simple. There's a `Done` method which returns a channel that's closed when our function is to be preempted. There's also some new, but easy to understand methods: a `Deadline` function to indicate if a goroutine will be canceled after a certain time, and an `Err` method that will return non-nil if the goroutine was canceled. But the `Value` method looks a little out of place. What's it for?

The Go authors noticed that one of the primary uses of goroutines was programs that serviced requests. Usually in these programs, request-specific information needs to be passed along in addition to information about preemption. This is the purpose of the `Value` function. We'll talk about this more in a bit, but for now we just need to know that the `context` package serves two primary purposes:

- To provide an API for canceling branches of your call-graph.
- To provide a data-bag for transporting request-scoped data through your call-graph.

Let's focus on the first aspect: cancellation.

As we learned in "Preventing Goroutine Leaks" on page 90, cancellation in a function has three aspects:

- A goroutine's parent may want to cancel it.
- A goroutine may want to cancel its children.

- Any blocking operations within a goroutine need to be preemptable so that it may be canceled.

The `context` package helps manage all three of these.

As we mentioned, the `Context` type will be the first argument to your function. If you look at the methods on the `Context` interface, you'll see that there's nothing present that can mutate the state of the underlying structure. Further, there's nothing that allows the function accepting the `Context` to cancel it. This protects functions up the call stack from children canceling the context. Combined with the `Done` method, which provides a `done` channel, this allows the `Context` type to safely manage cancellation from its antecedents.

This raises a question: if a `Context` is immutable, how do we affect the behavior of cancellations in functions below a current function in the call stack?

This is where the functions in the `context` package become important. Let's take a look at a few of them one more time to refresh our memory:

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

Notice that all these functions take in a `Context` and return one as well. Some of these also take in other arguments like `deadline` and `timeout`. The functions all generate new instances of a `Context` with the options relative to these functions.

`WithCancel` returns a new `Context` that closes its `done` channel when the returned `cancel` function is called. `WithDeadline` returns a new `Context` that closes its `done` channel when the machine's clock advances past the given `deadline`. `WithTimeout` returns a new `Context` that closes its `done` channel after the given `timeout` duration.

If your function needs to cancel functions below it in the call-graph in some manner, it will call one of these functions and pass in the `Context` it was given, and then pass the `Context` returned into its children. If your function doesn't need to modify the cancellation behavior, the function simply passes on the `Context` it was given.

In this way, successive layers of the call-graph can create a `Context` that adheres to their needs without affecting their parents. This provides a very composable, elegant solution for how to manage branches of your call-graph.

In this spirit, instances of a `Context` are meant to flow through your program's call-graph. In an object-oriented paradigm, it's common to store references to often-used data as member variables, but it's important to *not* do this with instances of `context.Context`. Instances of `context.Context` may look equivalent from the outside, but internally they may change at every stack-frame. For this reason, it's important to always pass instances of `Context` into your functions. This way functions have

the `Context` intended for it, and not the `Context` intended for a stack-frame `N` levels up the stack.

At the top of your asynchronous call-graph, your code probably won't have been passed a `Context`. To start the chain, the `context` package provides you with two functions to create empty instances of `Context`:

```
func Background() Context
func TODO() Context
```

`Background` simply returns an empty `Context`. `TODO` is not meant for use in production, but also returns an empty `Context`; `TODO`'s intended purpose is to serve as a placeholder for when you don't know which `Context` to utilize, or if you expect your code to be provided with a `Context`, but the upstream code hasn't yet furnished one.

So let's put all this to use. Let's look at an example that uses the `done` channel pattern, and see what benefits we might gain from switching to use of the `context` package. Here is a program that concurrently prints a greeting and a farewell:

```
func main() {
    var wg sync.WaitGroup
    done := make(chan interface{})
    defer close(done)

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printGreeting(done); err != nil {
            fmt.Printf("%v", err)
            return
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(done); err != nil {
            fmt.Printf("%v", err)
            return
        }
    }()

    wg.Wait()
}

func printGreeting(done <-chan interface{}) error {
    greeting, err := genGreeting(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
```

```
    return nil
}

func printFarewell(done <-chan interface{}) error {
    farewell, err := genFarewell(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(done <-chan interface{}) (string, error) {
    select {
    case <-done:
        return "", fmt.Errorf("canceled")
    case <-time.After(1*time.Minute):
    }
    return "EN/US", nil
}
```

Running this code produces:

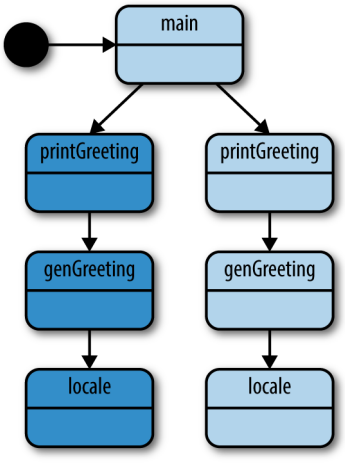
```
goodbye world!
hello world!
```

Ignoring the race condition (we could receive our farewell before we're greeted!), we can see that we have two branches of our program running concurrently. We've set up the standard preemption method by creating a `done` channel and passing it down through our call-graph. If we close the `done` channel at any point in `main`, both branches will be canceled.

By introducing goroutines in `main`, we've opened up the possibility of controlling this program in a few different and interesting ways. Maybe we want `genGreeting` to time out if it takes too long. Maybe we don't want `genFarewell` to invoke `locale` if we know its parent is going to be canceled soon. At each stack-frame, a function can affect the entirety of the call stack below it.

Using the `done` channel pattern, we could accomplish this by wrapping the incoming `done` channel in other `done` channels and then returning if any of them fire, but we wouldn't have the extra information about deadlines and errors a `Context` gives us.

To make comparing the `done` channel pattern to the use of the `context` package easier, let's represent this program as a tree. Each node in the tree represents an invocation of a function.



Let's modify our program to use the `context` package instead of a `done` channel. Because we now have the flexibility of a `context.Context`, we can introduce a fun scenario.

Let's say that `genGreeting` only wants to wait one second before abandoning the call to `locale`—a timeout of one second. We also want to build some smart logic into `main`. If `printGreeting` is unsuccessful, we also want to cancel our call to `printFarewell`. After all, it wouldn't make sense to say goodbye if we don't say hello!

Implementing this with the `context` package is trivial:

```
func main() {
    var wg sync.WaitGroup
    ctx, cancel := context.WithCancel(context.Background()) ❶
    defer cancel()
```

```
    wg.Add(1)
    go func() {
        defer wg.Done()

        if err := printGreeting(ctx); err != nil {
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel() ❷
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()

    wg.Wait()
}

func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(ctx context.Context) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second) ❸
    defer cancel()

    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}
```

```
func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(ctx context.Context) (string, error) {
    select {
    case <-ctx.Done():
        return "", ctx.Err() ❹
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

❶ Here `main` creates a new `Context` with `context.Background()` and wraps it with `context.WithCancel` to allow for cancellations.

❷ On this line, `main` will cancel the `Context` if there is an error returned from `printGreeting`.

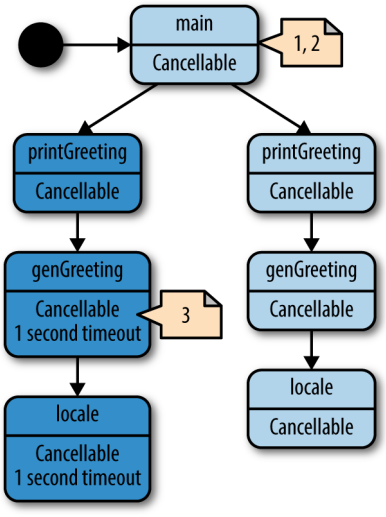
❸ Here `genGreeting` wraps its `Context` with `context.WithTimeout`. This will automatically cancel the returned `Context` after 1 second, thereby canceling any children it passes the `Context` into, namely `locale`.

❹ This line returns the reason why the `Context` was canceled. This error will bubble all the way up to `main`, which will cause the cancellation at ❷.

Here are the results of running this code:

```
cannot print greeting: context deadline exceeded
cannot print farewell: context canceled
```

Let's use our call-graph to understand what's going on. The numbers here correspond to the code callouts in the preceding example.



We can see from our output that the system works perfectly. Since we ensure `locale` takes at least one minute to run, our call in `genGreeting` will always time out, which means `main` will always cancel the call-graph below `printFarewell`.

Notice how `genGreeting` was able to build up a custom `context.Context` to meet its needs without having to affect its parent's `Context`. If `genGreeting` were to return successfully, and `printGreeting` needed to make another call, it could do so without leaking information about how `genGreeting` operated. This composability enables you to write large systems without mixing concerns throughout your call-graph.

We can make another improvement on this program: since we know `locale` takes roughly one minute to run, in `locale` we can check to see whether we were given a deadline, and if so, whether we'll meet it. This example demonstrates using the `context.Context`'s `Deadline` method to do so:

```
func main() {
    var wg sync.WaitGroup
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    wg.Add(1)
    go func() {
        defer wg.Done()

        if err := printGreeting(ctx); err != nil {
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel()
        }
    }()
```

```
    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()

    wg.Wait()
}

func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(ctx context.Context) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel()

    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}
```

```
func locale(ctx context.Context) (string, error) {
    if deadline, ok := ctx.Deadline(); ok { ❶
        if deadline.Sub(time.Now().Add(1*time.Minute)) <= 0 {
            return "", context.DeadlineExceeded
        }
    }

    select {
    case <-ctx.Done():
        return "", ctx.Err()
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

❶ Here we check to see whether our `Context` has provided a deadline. If it did, and our system's clock has advanced past the deadline, we simply return with a special error defined in the `context` package, `DeadlineExceeded`.

Although the difference in this iteration of the program is small, it allows the `locale` function to fail fast. In programs that may have a high cost for calling the next bit of functionality, this may save a significant amount of time, but at the very least it also allows the function to fail immediately instead of having to wait for the actual time-out to occur. The only catch is that you have to have some idea of how long your sub-ordinate call-graph will take—an exercise that can be very difficult.

This brings us to the other half of what the `context` package provides: a data-bag for a `Context` to store and retrieve request-scoped data. Remember that oftentimes when a function creates a goroutine and `Context`, it's starting a process that will service requests, and functions further down the stack may need information about the request. Here's an example of how to store data within the `Context`, and how to retrieve it:

```
func main() {
    ProcessRequest("jane", "abc123")
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), "userID", userID)
    ctx = context.WithValue(ctx, "authToken", authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (%v)",
        ctx.Value("userID"),
        ctx.Value("authToken"),
```

```
    )
}
```

This produces:

```
handling response for jane (abc123)
```

Pretty simple stuff. The only qualifications are that:

- The key you use must satisfy Go's notion of *comparability*; that is, the equality operators == and != need to return correct results when used.
- Values returned must be safe to access from multiple goroutines.

Since both the `Context`'s key and value are defined as `interface{}`, we lose Go's type-safety when attempting to retrieve values. The key could be a different type, or slightly different than the key we provide. The value could be a different type than we're expecting. For these reasons, the Go authors recommend you follow a few rules when storing and retrieving value from a `Context`.

First, they recommend you define a custom key-type in your package. As long as other packages do the same, this prevents collisions within the `Context`. As a reminder as to why, let's take a look at a short program that attempts to store keys in a map that have different types, but the same underlying value:

```
type foo int
type bar int

m := make(map[interface{}]int)
m[foo(1)] = 1
m[bar(1)] = 2

fmt.Printf("%v", m)
```

This produces:

```
map[1:1 1:2]
```

You can see that though the underlying values are the same, the different type information differentiates them within a map. Since the type you define for your package's keys is unexported, other packages cannot conflict with keys you generate within your package.

Since we don't export the keys we use to store the data, we must therefore export functions that retrieve the data for us. This works out nicely since it allows consumers of this data to use static, type-safe functions.

When you put all of this together, you get something like the following example:

```
func main() {
    ProcessRequest("jane", "abc123")
}
```

```
type ctxKey int

const (
    ctxUserID ctxKey = iota
    ctxAuthToken
)

func UserID(c context.Context) string {
    return c.Value(ctxUserID).(string)
}

func AuthToken(c context.Context) string {
    return c.Value(ctxAuthToken).(string)
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), ctxUserID, userID)
    ctx = context.WithValue(ctx, ctxAuthToken, authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (auth: %v)",
        UserID(ctx),
        AuthToken(ctx),
    )
}
```

Running this code produces:

```
handling response for jane (auth: abc123)
```

We now have a type-safe way to retrieve values from the `Context`, and—if the consumers were in a different package—they wouldn't know or care what keys were used to store the information. However, this technique does pose a problem.

In the previous example, let's say `HandleResponse` *did* live in another package named `response`, and let's say the package `ProcessRequest` lived in a package named `process`. The `process` package would have to import the `response` package to make the call to `HandleResponse`, but `HandleResponse` would have no way to access the accessor functions defined in the `process` package because importing `process` would form a circular dependency. Because the types used to store the keys in `Context` are private to the `process` package, the `response` package has no way to retrieve this data!

This coerces the architecture into creating packages centered around data types that are imported from multiple locations. This certainly isn't a bad thing, but it's something to be aware of.

The `context` package is pretty neat, but it hasn't been uniformly lauded. Within the Go community, the `context` package has been somewhat controversial. The cancellation aspect of the package has been pretty well received, but the ability to store arbitrary data in a `Context`, and the type-unsafe manner in which the data is stored, have caused some divisiveness. Although we have partially abated the lack of type-safety with our accessor functions, we could still introduce bugs by storing incorrect types. However, the larger issue is definitely the nature of *what* developers should store in instances of `Context`.

The most prevalent guidance on what's appropriate is this somewhat ambiguous comment in the `context` package:

```
Use context values only for request-scoped data that transits processes and
API boundaries, not for passing optional parameters to functions.
```

It's pretty clear what an optional parameter is (you shouldn't be using a `Context` to fulfill your secret desire for Go to support optional parameters), but what is "request-scoped data"? Supposedly it "transits processes and API boundaries," but that could describe lots of things. The best way I've found to define it is to come up with some heuristics with your team, and evaluate them in code reviews. Here are my heuristics:

1) The data should transit process or API boundaries.
: If you generate the data in your process' memory, it's probably not a good candidate to be request-scoped data unless you also pass it across an API boundary.

2) The data should be immutable.
: If it's not, then by definition what you're storing did not come from the request.

3) The data should trend toward simple types.
: If request-scoped data is meant to transit process and API boundaries, it's much easier for the other side to pull this data out if it doesn't also have to import a complex graph of packages.

4) The data should be data, not types with methods.
: Operations are logic and belong on the things consuming this data.

5) The data should help decorate operations, not drive them.
: If your algorithm behaves differently based on what is or isn't included in its `Context`, you have likely crossed over into the territory of optional parameters.

These aren't hard-and-fast rules; they're heuristics. However, if you find data you're storing in a `Context` violating all five of these guidelines, you might want to take a long look at what you're choosing to do.

Another dimension to consider is how many layers this data might need to traverse before utilization. If there are a few frameworks and tens of functions between where the data is accepted and where it is used, do you want to lean toward verbose, self-

documenting function signatures, and add the data as a parameter? Or would you rather place it in a `Context` and thereby create an invisible dependency? There are merits to each approach, and in the end it's a decision you and your team will have to make.

Even with these heuristics, whether or not a value is request-scoped data remains a difficult question to answer. Take a look at the following table. It lists my opinions on whether or not each type of data fulfills the five heuristics I've listed. Do you agree?

Data	1	2	3	4	5
Request ID	✓	✓	✓	✓	✓
User ID	✓	✓	✓	✓	
URL	✓	✓			
API Server Connection					
Authorization Token	✓	✓	✓	✓	
Request Token	✓	✓	✓		

Sometimes it's clear that something should not be stored in a context, as it is with API server connections, but sometimes it's not so clear. What about an authorization token? It's immutable, and it's likely a slice of bytes, but won't the receivers of this data use it to determine whether to field the request? Does this data belong in a context? To further muddy the waters, what is acceptable on one team may not be acceptable on another.

Ultimately there are no easy answers here. The package has been brought into the standard library, and so you must form *some* opinion on its use, but that opinion could (and probably should) change depending on what project you're touching. The final advice I'd leave you with is that the cancellation functionality provided by `Context` is very useful, and your feelings about the data-bag shouldn't deter you from using it.

Summary

We've covered a lot of ground in this chapter. We've combined Go's concurrency primitives to form patterns that help write maintainable concurrent code. Now that you're familiar with these patterns, we can discuss how we can incorporate these patterns into *other* patterns that will help you to write large systems. The next chapter will give you an overview of techniques for doing just that.