CHAPTER 5

Concurrency at Scale

Now that you've learned some common patterns for utilizing concurrency within Go, let's turn our attention to composing these patterns into a series of practices that will enable you to write large, composable systems that scale.

In this chapter, we'll discuss ways to scale concurrent operations within a single process, and also begin looking at how concurrency comes into play when dealing with more than one process.

Error Propagation

With concurrent code, and especially distributed systems, it's both easy for something to go wrong in your system, and difficult to understand why it happened. You can save yourself, your team, and your users a whole lot of pain by carefully considering how issues propagate through your system, and how they end up being represented to the user. In the section "Error Handling" on page 97, we discussed *how* to propagate errors from goroutines, but we didn't spend any time discussing what those errors should look like, or how errors should flow through a large and complex system. Let's spend some time here discussing a philosophy of error propagation. What follows is an opinionated framework for handling errors in concurrent systems.

Many developers make the mistake of thinking of error propagation as secondary, or "other," to the flow of their system. Careful consideration is given to how data flows through the system, but errors are something that are tolerated and ferried up the stack without much thought, and ultimately dumped in front of the user. Go attempted to correct this bad practice by forcing users to handle errors at every frame in the call stack, but it's still common to see errors treated as second-class citizens to the system's control flow. With just a little forethought, and minimal overhead, you can make your error handling an asset to your system, and a delight to your users.

First let's examine what errors are. When do they occur, and what benefit do they provide?

Errors indicate that your system has entered a state in which it cannot fulfill an operation that a user either explicitly or implicitly requested. Because of this, it needs to relay a few pieces of critical information:

What happened.
: This is the part of the error that contains information about what happened, e.g., "disk full," "socket closed," or "credentials expired." This information is likely to be generated implicitly by whatever it was that generated the errors, although you can probably decorate this with some context that will help the user.

When and where it occurred.
: Errors should always contain a complete stack trace starting with how the call was initiated and ending with where the error was instantiated. The stack trace should *not* be contained in the error message (more on this in a bit), but should be easily accessible when handling the error up the stack.

  Further, the error should contain information regarding the context it's running within. For example, in a distributed system, it should have some way of identifying what machine the error occurred on. Later, when trying to understand what happened in your system, this information will be invaluable.

  In addition, the error should contain the time on the machine the error was instantiated on, in UTC.

A friendly user-facing message.
: The message that gets displayed to the user should be customized to suit your system and its users. It should only contain abbreviated and relevant information from the previous two points. A friendly message is human-centric, gives some indication of whether the issue is transitory, and should be about one line of text.

How the user can get more information.
: At some point, someone will likely want to know, in detail, what happened when the error occurred. Errors that are presented to users should provide an ID that can be cross-referenced to a corresponding log that displays the full information of the error: time the error occurred (not the time the error was logged), the stack trace—everything you stuffed into the error when it was created. It can also be helpful to include a hash of the stack trace to aid in aggregating like issues in bug trackers.

By default, no error will contain all of this information without your intervention. Therefore, you could take the stance that any error that is propagated to the user *without* this information is a mistake, and therefore a bug. This leads to a general

framework we can use to think about errors. It's possible to place all errors into one of two categories:

- Bugs
- Known edge cases (e.g., broken network connections, failed disk writes, etc.)

Bugs are errors that you have not customized to your system, or "raw" errors—your known edge cases. Sometimes this is intentional; you may be OK with letting errors from edge cases reach your users while you get the first few iterations of your system out the door. Sometimes this is by accident. But if you agree with the approach I've laid out, raw errors are always bugs. This distinction will prove useful when determining how to propagate errors, how your system grows over time, and what to ultimately display to the user.

Imagine a large system with multiple modules:

Let's say an error occurs in the "Low Level Component" and we've crafted a well-formed error there to be passed up the stack. Within the context of the "Low Level Component," this error might be considered well-formed, but within the context of our system, it may not be. Let's take the stance that at the boundaries of each component, all incoming errors must be wrapped in a well-formed error for the component our code is within. For example, if we were in "Intermediary Component," and we were calling code from "Low Level Component," which might error, we could have this:

```
func PostReport(id string) error {
    result, err := lowlevel.DoWork()
    if err != nil {
        if _, ok := err.(lowlevel.Error); ok { ❶
            err = WrapErr(err, "cannot post report with id %q", id) ❷
        }
        return err
    }
 // ...
}
```

❶ Here we check to ensure we're receiving a well-formed error. If we aren't, we'll simply ferry the malformed error up the stack to indicate a bug.

❷ Here we use a hypothetical function call to wrap the incoming error with pertinent information for our module, and to give it a new type. Note that wrapping

the error might involve *hiding* some low-level details that may not be important for the user within this context.

The low-level details of where the root of the error occurred (e.g., what goroutine, machine, stack trace, etc.) are still filled in when the error is initially instantiated, but our architecture dictates that at module boundaries we convert the error to our module's error type—potentially filling in pertinent information. Now, any error that escapes *our* module without our module's error type can be considered malformed, and a bug. Note that it is only necessary to wrap errors in this fashion at your *own* module boundaries—public functions/methods—or when your code can add valuable context. Usually this prevents the need for wrapping errors in most of the code.

Taking this stance allows our system to grow very organically. We can be sure that incoming errors are well-formed, and we in turn can ensure we are giving thought to how errors escape our module. Error correctness becomes an emergent property of our system. We also concede perfection from the start by explicitly handling malformed errors, and by doing so we have given ourselves a framework to take mistakes and correct them over time. Malformed errors are clearly delineated both by type and, as we'll see, by what is presented to the user.

As we established, all errors should be logged with as much information as is available. But when displaying errors to users, this is where the distinction between bugs and known edge cases comes in.

When our user-facing code receives a well-formed error, we can be confident that at all levels in our code, care was taken to craft the error message, and we can simply log it and print it out for the user to see. The confidence that we get from seeing an error with the correct type cannot be understated.

When malformed errors, or bugs, are propagated up to the user, we should also log the error, but then display a friendly message to the user stating something unexpected has happened. If we support automatic error reporting in our system, the error should be reported back as a bug. If we don't, we might suggest the user file a bug report. Note that the malformed error might actually contain useful information, but we cannot guarantee this, and so—since the only guarantee we do have is that the error is not customized—we should bluntly display a human-centric message about what happened.

Remember that in either case, with well- or malformed errors, we will have included a log ID in the message to give the user something to refer back to should the user want more information. Thus, even if bugs were to contain useful information, the curious user still has means to investigate.

Let's take a look at a complete example. This example won't be extremely robust (e.g., the error type is perhaps simplistic), and the call stack is linear, which obfuscates the

fact that it's only necessary to wrap errors at module boundaries. Also, it's difficult to represent functions in different packages in a book, and so we'll be pretending.

First, let's create an error type that can contain all of the aspects of a well-formed error we've discussed:

```
type MyError struct {
    Inner      error
    Message    string
    StackTrace string
    Misc       map[string]interface{}
}

func wrapError(err error, messagef string, msgArgs ...interface{}) MyError {
    return MyError{
        Inner:      err, ❶
        Message:    fmt.Sprintf(messagef, msgArgs...),
        StackTrace: string(debug.Stack()), ❷
        Misc:       make(map[string]interface{}), ❸
    }
}

func (err MyError) Error() string {
    return err.Message
}
```

❶ Here we store the error we're wrapping. We always want to be able to get back to the lowest-level error in case we need to investigate what happened.

❷ This line of code takes note of the stack trace when the error was created. A more sophisticated error type might elide the stack-frame from `wrapError`.

❸ Here we create a catch-all for storing miscellaneous information. This is where we might store the concurrent ID, a hash of the stack trace, or other contextual information that might help in diagnosing the error.

Next, let's create a module, `lowlevel`:

```
// "lowlevel" module

type LowLevelErr struct {
    error
}

func isGloballyExec(path string) (bool, error) {
    info, err := os.Stat(path)
    if err != nil {
        return false, LowLevelErr{(wrapError(err, err.Error()))} ❶
    }
    return info.Mode().Perm()&0100 == 0100, nil
}
```

❶ Here we wrap the raw error from calling `os.Stat` with a customized error. In this case we are OK with the message coming out of this error, and so we won't mask it.

Then, let's create another module, `intermediate`, which calls functions from the `lowlevel` package:

```
// "intermediate" module

type IntermediateErr struct {
    error
}

func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)
    if err != nil {
        return err ❶
    } else if isExecutable == false {
        return wrapError(nil, "job binary is not executable")
    }

    return exec.Command(jobBinPath, "--id="+id).Run() ❶
}
```

❶ Here we are passing on errors from the `lowlevel` module. Because of our architectural decision to consider errors passed on from other modules without wrapping them in our own type bugs, this will cause us issues later.

Finally, let's create a top-level `main` function that calls functions from the `intermediate` package. This is the user-facing portion of our program:

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
    log.Printf("%#v", err) ❸
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime|log.LUTC)

    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok { ❶
            msg = err.Error()
        }
        handleError(1, err, msg) ❷
    }
}
```

❶ Here we check to see if the error is of the expected type. If it is, we know it's a well-crafted error, and we can simply pass its message on to the user.

❷ On this line we bind the log and error message together with an ID of `1`. We could easily make this increase monotonically, or use a GUID to ensure a unique ID.

❸ Here we log out the full error in case someone needs to dig into what happened.

When we run this, we get a log message that contains:

```
[logID: 1]: 21:46:07 main.LowLevelErr{error:main.MyError{Inner:
(*os.PathError)(0xc4200123f0),
Message:"stat /bad/job/binary: no such file or directory",
StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0xc420012420, 0x2f, 0xc420045d80)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530200, 0xc4200123f0, 0xc420012420, 0x2f, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, ...)
    /tmp/babel-79540aE/go-src-7954NTK.go:22 +0x62
main.isGloballyExec(0x4d1313, 0xf, 0xc420045eb8, 0x487649, 0xc420056050)
    /tmp/babel-79540aE/go-src-7954NTK.go:37 +0xaa
main.runJob(0x4cfada, 0x1, 0x4d4c35, 0x22)
    /tmp/babel-79540aE/go-src-7954NTK.go:47 +0x48
main.main()
    /tmp/babel-79540aE/go-src-7954NTK.go:67 +0x63
", Misc:map[string]interface {}{}}}
```

And a message to `stdout` that contains:

```
[1] There was an unexpected issue; please report this as a bug.
```

We can see that somewhere along this error's path, it was not handled correctly, and because we cannot be sure the error message is fit for human consumption, we print a simple error out stating that something unexpected happened (true if we are following this methodology). If we look back up to our `intermediate` module, we recall why: we didn't wrap the errors from the `lowlevel` module. Let's correct that and see what happens:

```
// "intermediate" module

type IntermediateErr struct {
    error
}

func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)
    if err != nil {
        return IntermediateErr{wrapError(
            err,
```

```
            "cannot run job %q: requisite binaries not available",
            id,
        )} ❶
    } else if isExecutable == false {
        return wrapError(
            nil,
            "cannot run job %q: requisite binaries are not executable",
            id,
        )
    }

    return exec.Command(jobBinPath, "--id="+id).Run()
}
```

❶ Here we are now customizing the error with a crafted message. In this case, we want to obfuscate the low-level details of why the job isn't running because we feel it's not important information to consumers of our module.

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
    log.Printf("%#v", err)
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime|log.LUTC)

    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok {
            msg = err.Error()
        }
        handleError(1, err, msg)
    }
}
```

Now when we run the updated code, we get a similar log message:

```
[logID: 1]: 22:11:04 main.IntermediateErr{error:main.MyError
{Inner:main.LowLevelErr{error:main.MyError{Inner:(*os.PathError)
(0xc4200123f0), Message:"stat /bad/job/binary: no such file or directory",
StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0xc420012420, 0x2f, 0x0)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530200, 0xc4200123f0, 0xc420012420, 0x2f, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, ...)
    /tmp/babel-79540aE/go-src-7954DTN.go:22 +0xbb
main.isGloballyExec(0x4d1313, 0xf, 0x4daecc, 0x30, 0x4c5800)
    /tmp/babel-79540aE/go-src-7954DTN.go:39 +0xc5
main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
    /tmp/babel-79540aE/go-src-7954DTN.go:51 +0x4b
```

```
main.main()
    /tmp/babel-79540aE/go-src-7954DTN.go:71 +0x63
", Misc:map[string]interface {}{}}}, Message:"cannot run job \"1\":
requisite binaries not available", StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0x4d63f0, 0x33, 0xc420045e40)
    /home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79
main.wrapError(0x530380, 0xc42000a370, 0x4d63f0, 0x33,
0xc420045e40, 0x1, 0x1, 0x0, 0x0, 0x0, ...)
    /tmp/babel-79540aE/go-src-7954DTN.go:22 +0xbb
main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
    /tmp/babel-79540aE/go-src-7954DTN.go:53 +0x356
main.main()
    /tmp/babel-79540aE/go-src-7954DTN.go:71 +0x63
", Misc:map[string]interface {}{}}}
```

But our error message is now exactly what we want users to see:

```
[1] cannot run job "1": requisite binaries not available
```

There are error packages[1] that are compatible with this approach, but it will be up to you to implement this technique using whatever error package you decide to use. The good news is that this technique is organic; you can canvas your top-level error handling and delineate between bugs and well-crafted errors, and then progressively ensure that all the errors you create are considered well-crafted.

Timeouts and Cancellation

When working with concurrent code, timeouts and cancellations are going to turn up frequently. As we'll see in this section, among other things, timeouts are crucial to creating a system with behavior you can understand. Cancellation is one natural response to a timeout. We'll also explore other reasons a concurrent process might be canceled.

So what are the reasons we might want our concurrent processes to support timeouts? Here are a few:

System saturation
: As we discussed in the section "Queuing" on page 124, if our system is saturated (i.e., if its ability to process requests is at capacity), we may want requests at the edges of our system to time out rather than take a long time to field them. Which path you take depends on your problem space, but here are some general guidelines for when to time out:

  - If the request is unlikely to be repeated when it is timed out.

1 I recommend *http://github.com/pkg/errors*.

- If you don't have the resources to store the requests (e.g., memory for in-memory queues, disk space for persisted queues).
- If the need for the request, or the data it's sending, will go stale (we'll discuss this next). If a request is likely to be repeated, your system will develop an overhead from accepting and timing out requests. This can lead to a death-spiral if the overhead becomes greater than our system's capacity. However, this is a moot point if we lack the system resources required to store the request in a queue. And even if we meet these two guidelines, there is little point in enqueueing a request whose need will expire by the time we can process it. This brings us to our next reason to support timeouts.

Stale data

Sometimes data has a window within which it must be processed before more relevant data is available, or the need to process the data has expired. If a concurrent process takes longer to process the data than this window, we would want to time out and cancel the concurrent process. For instance, if our concurrent process is dequeing a request after a long wait, the request or its data might have become obsolete during the queuing process.

If this window is known beforehand, it would make sense to pass our concurrent process a `context.Context` created with `context.WithDeadline`, or `context.WithTimeout`. If the window is not known beforehand, we'd want the parent of the concurrent process to be able to cancel the concurrent process when the need for the request is no longer present. `context.WithCancel` is perfect for this purpose.

Attempting to prevent deadlocks

In a large system—especially distributed systems—it can sometimes be difficult to understand the way in which data might flow, or what edge cases might turn up. It is not unreasonable, and even recommended, to place timeouts on *all* of your concurrent operations to guarantee your system won't deadlock. The timeout period doesn't have to be close to the actual time it takes to perform your concurrent operation. The timeout period's purpose is only to prevent deadlock, and so it only needs to be short enough that a deadlocked system will unblock in a reasonable amount of time for your use case.

Remember from the section "Deadlocks, Livelocks, and Starvation" on page 10 that attempting to avoid a deadlock by setting a timeout can potentially transform your problem from a system that deadlocks to a system that livelocks. However, in large systems, because there are more moving parts, there is a higher probability that your system will experience a different timing profile than when you deadlocked last. Therefore, it is preferable to chance a livelock and fix that as

time permits, than for a deadlock to occur and have a system recoverable only by restart.

Note that this isn't a recommendation for how to build a system correctly; rather a suggestion for building a system that is tolerant to timing errors you may not have exercised during development and testing. I do recommend you keep the timeouts in place, but the goal should be to converge on a system without deadlocks where the timeouts are never triggered.

Now that we have a grasp on when to utilize timeouts, let's turn our attention to the causes of cancellation, and how to build a concurrent process to handle cancellation gracefully. There are a number of reasons why a concurrent process might be canceled:

Timeouts
: A timeout is an implicit cancellation.

User intervention
: For a good user experience, it's usually advisable to start long-running processes concurrently and then report status back to the user at a polling interval, or allow the users to query for status as they see fit. When there are user-facing concurrent operations, it is therefore also sometimes necessary to allow the users to cancel the operation they've started.

Parent cancellation
: For that matter, if any kind of parent of a concurrent operation—human or otherwise—stops, as a child of that parent, we will be canceled.

Replicated requests
: We may wish to send data to multiple concurrent processes in an attempt to get a faster response from one of them. When the first one comes back, we would want to cancel the rest of the processes. We'll discuss this in detail in the section "Replicated Requests" on page 172.

There are likely other possible reasons, too. However, the question "why" is not nearly as difficult or interesting as the question of "how." In Chapter 4 we explored two ways to cancel concurrent processes: a `done` channel, and the `context.Context` type. But that's the easy part; here we want to explore more complex questions: when a concurrent process is canceled, what does that mean for the algorithm that was executing, and its downstream consumers? When writing concurrent code that can be terminated at any time, what things do you need to take into account?

In order to answer those questions, the first thing we need to explore is the preemptability of a concurrent process. Take the following code, and assume it's running in its own goroutine:

```
var value interface{}
select {
case <-done:
    return
case value = <-valueStream:
}

result := reallyLongCalculation(value)

select {
case <-done:
    return
case resultStream<-result:
}
```

We've dutifully coupled the read from `valueStream` and the write to `resultStream` with a check against the `done` channel to see if the goroutine has been canceled, but we still have a problem. `reallyLongCalculation` doesn't look to be preemptable, and, according to the name, it looks like it might take a really long time! This means that if something attempts to cancel this goroutine while `reallyLongCalculation` is executing, it could be a very long time before we acknowledge the cancellation and halt. Let's try and make `reallyLongCaluclation` preemptable and see what happens:

```
reallyLongCalculation := func(
    done <-chan interface{},
    value interface{},
) interface{} {
    intermediateResult := longCalculation(value)
    select {
    case <-done:
        return nil
    default:
    }

    return longCaluclation(intermediateResult)
}
```

We've made some progress: `reallyLongCaluclation` is now preemptable, but we can see that we've only halved the problem: we can only preempt `reallyLongCalculation` in between calls to other, seemingly long-running, function calls. To solve this, we need to make `longCalculation` preemptable as well:

```
reallyLongCalculation := func(
    done <-chan interface{},
    value interface{},
) interface{} {
    intermediateResult := longCalculation(done, value)
    return longCaluclation(done, intermediateResult)
}
```

If you take this line of reasoning to its logical conclusion, we see that we must do two things: define the period within which our concurrent process is preemptable, and ensure that any functionality that takes more time than this period is itself preemptable. An easy way to do this is to break up the pieces of your goroutine into smaller pieces. You should aim for all *nonpreemptable* atomic operations to complete in less time than the period you've deemed acceptable.

there's another problem lurking here as well: if our goroutine happens to modify shared state—e.g., a database, a file, an in-memory data structure—what happens when the goroutine is canceled? Does your goroutine try and roll back the intermediary work it's done? How long does it have to do this work? Something has told the goroutine that it should halt, so the goroutine shouldn't take too long to roll back its work, right?

It's difficult to give general advice on how to handle this problem because the nature of your algorithm will dictate so much of how you handle this situation; however, if you keep your modifications to any shared state within a tight scope, and/or ensure those modifications are easily rolled back, you can usually handle cancellations pretty well. If possible, build up intermediate results in-memory and then modify state as quickly as possible. As an example, here is the *wrong* way to do it:

```
result := add(1, 2, 3)
writeTallyToState(result)
result = add(result, 4, 5, 6)
writeTallyToState(result)
result = add(result, 7, 8, 9)
writeTallyToState(result)
```

Here we write to state three times. If a goroutine running this code were canceled before the final write, we'd need to somehow roll back the previous two calls to `writeTallyToState`. Contrast that approach with this:

```
result := add(1, 2, 3, 4, 5, 6, 7, 8, 9)
writeTallyToState(result)
```

Here the surface area we have to worry about rolling back is much smaller. If the cancellation comes in after our call to `writeToState`, we still need a way to back out our changes, but the probability that this will happen is much smaller since we only modify state once.

Another issue you need to be concerned with is duplicated messages. Let's say you have a pipeline with three stages: a generator stage, stage A, and stage B. The generator stage monitors stage A by keeping track of how long it's been since it last read from its channel, and brings up a new instance, A2, if the current instance becomes nonperformant. If that were to happen, it is possible for stage B to receive duplicate messages (Figure 5-1).

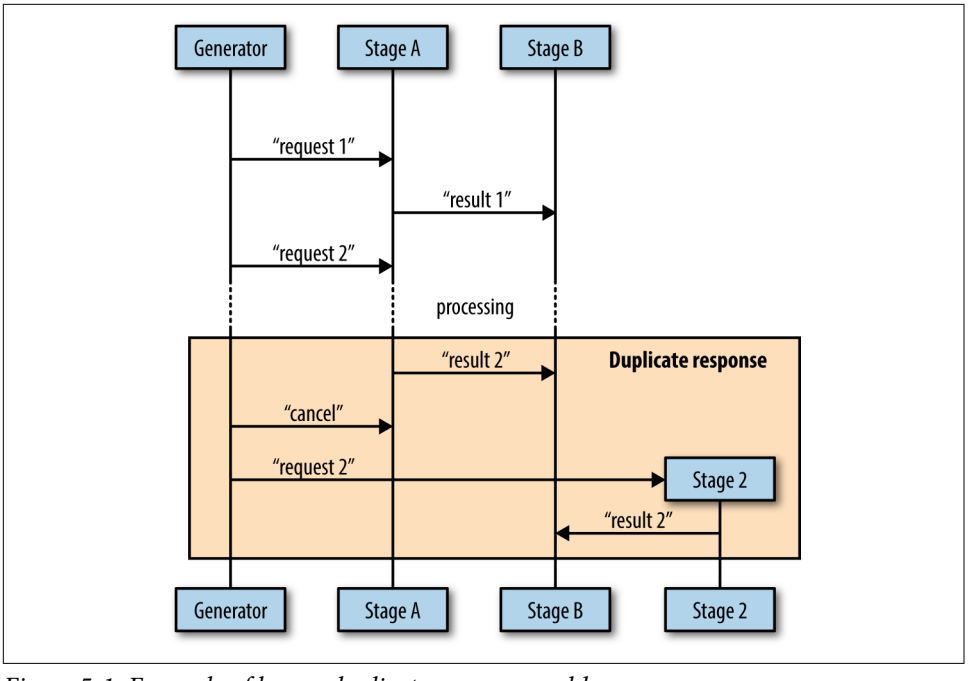

Figure 5-1. Example of how a duplicate message could occur

You can see here that it's possible for stage B to receive duplicate messages if the cancellation message comes in after stage A has already sent its result to stage B.

There are a few ways to avoid sending duplicate messages. The easiest (and the method I recommend) is to make it vanishingly unlikely that a parent goroutine will send a cancellation signal after a child goroutine has already reported a result. This requires bidirectional communication between the stages, and we'll cover this in detail in the section "Heartbeats" on page 161. Other approaches are:

Accept either the first or last result reported
: If your algorithm allows it, or your concurrent process is idempotent, you can simply allow for the possibility of duplicate messages in your downstream processes and choose whether to accept the first or last message you receive.

Poll the parent goroutine for permission
: You can use bidirectional communication with your parent to explicitly request permission to send your message. As we'll see, this approach is similar to heartbeats. It would look something like Figure 5-2.

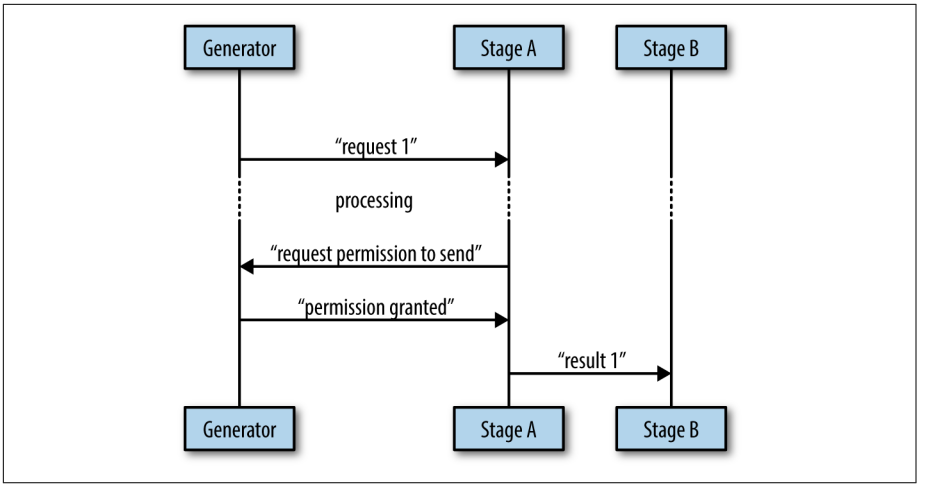

Figure 5-2. An example of polling the parent goroutine

Because we explicitly request permission to perform the write to B's channel, this is an even safer route than heartbeats; however, in practice, this is rarely necessary, and since it is more complicated than heartbeats, and heartbeats are more generally useful, I suggest you just use heartbeats.

When designing your concurrent processes, be sure to take into account timeouts and cancellation. Like many other topics in software engineering, neglecting timeouts and cancellation from the beginning and then attempting to put them in later is a bit like trying to add eggs to a cake after it has been baked.

Heartbeats

Heartbeats are a way for concurrent processes to signal life to outside parties. They get their name from human anatomy wherein a heartbeat signifies life to an observer. Heartbeats have been around since before Go, and remain useful within it.

There are a few different reasons heartbeats are interesting for concurrent code. They allow us insights into our system, and they can make testing the system deterministic when it might otherwise not be.

There are two different types of heartbeats we'll discuss in this section:

- Heartbeats that occur on a time interval.
- Heartbeats that occur at the beginning of a unit of work.

Heartbeats that occur on a time interval are useful for concurrent code that might be waiting for something else to happen for it to process a unit of work. Because you

don't know when that work might come in, your goroutine might be sitting around for a while waiting for something to happen. A heartbeat is a way to signal to its listeners that everything is well, and that the silence is expected.

The following code demonstrates a goroutine that exposes a heartbeat:

```
doWork := func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (<-chan interface{}, <-chan time.Time) {
    heartbeat := make(chan interface{}) ❶
    results := make(chan time.Time)
    go func() {
        defer close(heartbeat)
        defer close(results)

        pulse := time.Tick(pulseInterval) ❷
        workGen := time.Tick(2*pulseInterval) ❸

        sendPulse := func() {
            select {
            case heartbeat <-struct{}{}:
            default: ❹
            }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-done:
                    return
                case <-pulse: ❺
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for {
            select {
            case <-done:
                return
            case <-pulse: ❺
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
        }
    }()
    return heartbeat, results
}
```

❶ Here we set up a channel to send heartbeats on. We return this out of `doWork`.

❷ Here we set the heartbeat to pulse at the `pulseInterval` we were given. Every `pulseInterval` there will be something to read on this channel.

❸ This is just another ticker used to simulate work coming in. We choose a duration greater than the `pulseInterval` so that we can see some heartbeats coming out of the goroutine.

❹ Note that we include a `default` clause. We must always guard against the fact that no one may be listening to our heartbeat. The results emitted from the goroutine are critical, but the pulses are not.

❺ Just like with `done` channels, anytime you perform a send or receive, you also need to include a case for the heartbeat's pulse.

Notice that because we might be sending out multiple pulses while we wait for input, or multiple pulses while waiting to send results, all the `select` statements need to be within `for` loops. Looking good so far; how do we utilize this function and consume the events it emits? Let's take a look:

```
done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) }) ❶

const timeout = 2*time.Second ❷
heartbeat, results := doWork(done, timeout/2) ❸
for {
    select {
    case _, ok := <-heartbeat: ❹
        if ok == false {
            return
        }
        fmt.Println("pulse")
    case r, ok := <-results: ❺
        if ok == false {
            return
        }
        fmt.Printf("results %v\n", r.Second())
    case <-time.After(timeout): ❻
        return
    }
}
```

❶ We set up the standard `done` channel and close it after 10 seconds. This gives our goroutine time to do some work.

❷ Here we set our timeout period. We'll use this to couple our heartbeat interval to our timeout.

❸ We pass in `timeout/2` here. This gives our heartbeat an extra tick to respond so that our timeout isn't too sensitive.

❹ Here we select on the heartbeat. When there are no results, we are at least guaranteed a message from the `heartbeat` channel every `timeout/2`. If we don't receive it, we know there's something wrong with the goroutine itself.

❺ Here we select from the results channel; nothing fancy going on here.

❻ Here we time out if we haven't received either a heartbeat or a new result.

Running this code produces:

```
pulse
pulse
results 52
pulse
pulse
results 54
pulse
pulse
results 56
pulse
pulse
results 58
pulse
```

You can see that we receive about two pulses per result as we intended.

Now in a properly functioning system, heartbeats aren't that interesting. We might use them to gather statistics regarding idle time, but the utility for interval-based heartbeats really shines when your goroutine isn't behaving as expected.

Consider the next example. We'll simulate an incorrectly written goroutine with a panic by stopping the goroutine after only two iterations, and then not closing either of our channels. Let's have a look:

```
doWork := func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (<-chan interface{}, <-chan time.Time) {
    heartbeat := make(chan interface{})
    results := make(chan time.Time)
    go func() {
        pulse := time.Tick(pulseInterval)
        workGen := time.Tick(2*pulseInterval)
```

```
        sendPulse := func() {
            select {
            case heartbeat <-struct{}{}:
            default:
            }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-pulse:
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for i := 0; i < 2; i++ { ❶
            select {
            case <-done:
                return
            case <-pulse:
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
        }
    }()
    return heartbeat, results
}

done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) })

const timeout = 2 * time.Second
heartbeat, results := doWork(done, timeout/2)
for {
    select {
    case _, ok := <-heartbeat:
        if ok == false {
            return
        }
        fmt.Println("pulse")
    case r, ok := <-results:
        if ok == false {
            return
        }
        fmt.Printf("results %v\n", r)
    case <-time.After(timeout):
        fmt.Println("worker goroutine is not healthy!")
        return
```

```
    }
}
```

❶ Here is our simulated panic. Instead of infinitely looping until we're asked to stop, as in the previous example, we'll only loop twice.

Running this code produces:

```
pulse
pulse
worker goroutine is not healthy!
```

Beautiful! Within two seconds our system realizes something is amiss with our goroutine and breaks the for-select loop. By using a heartbeat, we have successfully avoided a deadlock, and we remain deterministic by not having to rely on a longer timeout. We'll discuss how we can take this concept even further in "Healing Unhealthy Goroutines" on page 188.

Also note that heartbeats help with the opposite case: they let us know that long-running goroutines remain up, but are just taking a while to produce a value to send on the values channel.

Now let's shift over to looking at heartbeats that occur at the beginning of a unit of work. These are extremely useful for tests. Here's an example that sends a pulse before every unit of work:

```
doWork := func(done <-chan interface{}) (<-chan interface{}, <-chan int) {
    heartbeatStream := make(chan interface{}, 1) ❶
    workStream := make(chan int)
    go func () {
        defer close(heartbeatStream)
        defer close(workStream)

        for i := 0; i < 10; i++ {
            select { ❷
            case heartbeatStream <- struct{}{}:
            default: ❸
            }

            select {
            case <-done:
                return
            case workStream <- rand.Intn(10):
            }
        }
    }()

    return heartbeatStream, workStream
}

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

```
    defer close(done)

    heartbeat, results := doWork(done)
    for {
        select {
        case _, ok := <-heartbeat:
            if ok {
                fmt.Println("pulse")
            } else {
                return
            }
        case r, ok := <-results:
            if ok {
                fmt.Printf("results %v\n", r)
            } else {
                return
            }
        }
    }
```

❶ Here we create the `heartbeat` channel with a buffer of one. This ensures that there's always at least one pulse sent out even if no one is listening in time for the send to occur.

❷ Here we set up a separate `select` block for the heartbeat. We don't want to include this in the same `select` block as the send on `results` because if the receiver isn't ready for the result, they'll receive a pulse instead, and the current value of the result will be lost. We also don't include a case statement for the `done` channel since we have a `default` case that will just fall through.

❸ Once again we guard against the fact that no one may be listening to our heartbeats. Because our `heartbeat` channel was created with a buffer of one, if someone *is* listening, but not in time for the first pulse, they'll still be notified of a pulse.

Running this code produces:

```
pulse
results 1
pulse
results 7
pulse
results 7
pulse
results 9
pulse
results 1
pulse
results 8
```

```
pulse
results 5
pulse
results 0
pulse
results 6
pulse
results 0
```

You can see in this example that we receive one pulse for every result, as intended.

Where this technique really shines is in writing tests. Interval-based heartbeats can be used in the same fashion, but if you only care that the goroutine has started doing its work, this style of heartbeat is simple. Consider the following snippet of code:

```
func DoWork(
    done <-chan interface{},
    nums ...int,
) (<-chan interface{}, <-chan int) {
    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)
    go func() {
        defer close(heartbeat)
        defer close(intStream)

        time.Sleep(2*time.Second) ❶

        for _, n := range nums {
            select {
            case heartbeat <- struct{}{}:
            default:
            }

            select {
            case <-done:
                return
            case intStream <- n:
            }
        }
    }()

    return heartbeat, intStream
}
```

❶ Here we simulate some kind of delay before the goroutine can begin working. In practice this can be all kinds of things and is nondeterministic. I've seen delays caused by CPU load, disk contention, network latency, and goblins.

The `DoWork` function is a pretty simple generator that converts the numbers we pass in to a stream on the channel it returns. Let's try testing this function. Here's an example of a *bad* test:

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    _, results := DoWork(done, intSlice...)

    for i, expected := range intSlice {
        select {
        case r := <-results:
            if r != expected {
                t.Errorf(
                  "index %v: expected %v, but received %v,",
                  i,
                  expected,
                  r,
                )
            }
        case <-time.After(1 * time.Second): ❶
            t.Fatal("test timed out")
        }
    }
}
```

❶ Here we time out after what we think is a reasonable duration to prevent a broken goroutine from deadlocking our test.

Running this test produces:

```
go test ./bad_concurrent_test.go
--- FAIL: TestDoWork_GeneratesAllNumbers (1.00s)
    bad_concurrent_test.go:46: test timed out
FAIL
FAIL    command-line-arguments  1.002s
```

This test is bad because it's nondeterministic. In our example function, I've ensured this test will always fail, but if I were to remove the `time.Sleep`, the situation actually gets worse: this test will pass at times, and fail at others.

We mentioned earlier how factors external to the process can cause the goroutine to take longer to get to its first iteration. Even whether or not the goroutine is scheduled in the first place is a concern. The point is that we can't be guaranteed that the first iteration of the goroutine will occur before our timeout is reached, and so we begin thinking in terms of probabilities: how likely is it that this timeout will be significant? We could increase the timeout, but that means failures will take a long time, thereby slowing down our test suite.

This is an awful, awful position to be in. The team no longer knows whether it can trust a test failure and begin ignoring failures—the whole endeavor begins to unravel.

Fortunately with a heartbeat this is easily solved. Here is a test that is deterministic:

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    heartbeat, results := DoWork(done, intSlice...)

    <-heartbeat ❶

    i := 0
    for r := range results {
        if expected := intSlice[i]; r != expected {
            t.Errorf("index %v: expected %v, but received %v,", i, expected, r)
        }
        i++
    }
}
```

❶ Here we wait for the goroutine to signal that it's beginning to process an iteration.

Running this test produces the following output:

```
ok      command-line-arguments                  2.002s
```

Because of the heartbeat, we can safely write our test without timeouts. The only risk we run is of one of our iterations taking an inordinate amount of time. If that's important to us, we can utilize the safer interval-based heartbeats and achieve perfect safety.

Here is an example of a test utilizing interval-based heartbeats:

```
func DoWork(
    done <-chan interface{},
    pulseInterval time.Duration,
    nums ...int,
) (<-chan interface{}, <-chan int) {
    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)
    go func() {
        defer close(heartbeat)
        defer close(intStream)

        time.Sleep(2*time.Second)

        pulse := time.Tick(pulseInterval)
        numLoop: ❷
        for _, n := range nums {
            for { ❶
                select {
                case <-done:
                    return
```

```
                case <-pulse:
                    select {
                    case heartbeat <- struct{}{}:
                    default:
                    }
                case intStream <- n:
                    continue numLoop ❸
                }
            }
        }
    }()

    return heartbeat, intStream
}

func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    const timeout = 2*time.Second
    heartbeat, results := DoWork(done, timeout/2, intSlice...)

    <-heartbeat ❹

    i := 0
    for {
        select {
        case r, ok := <-results:
            if ok == false {
                return
            } else if expected := intSlice[i]; r != expected {
                t.Errorf(
                    "index %v: expected %v, but received %v,",
                    i,
                    expected,
                    r,
                )
            }
            i++
        case <-heartbeat: ❺
        case <-time.After(timeout):
            t.Fatal("test timed out")
        }
    }
}
```

❶ We require two loops: one to range over our list of numbers, and this inner loop to run until the number is successfully sent on the `intStream`.

❷ We're using a label here to make continuing from the inner loop a little simpler.

❸ Here we continue executing the outer loop.

❹ We still wait for the first heartbeat to occur to indicate we've entered the goroutine's loop.

❺ We also select on the heartbeat here to keep the timeout from occuring.

Running this test produces:

```
ok      command-line-arguments                  3.002s
```

You've probably noticed that this version of the test is much less clear. The logic of what we're testing is a bit muddled. For this reason—if you're reasonably sure the goroutine's loop won't stop executing once it's started—I recommend only blocking on the first heartbeat and then falling into a simple `range` statement. You can write separate tests that specifically test for failing to close channels, loop iterations taking too long, and any other timing-related issues.

Heartbeats aren't strictly necessary when writing concurrent code, but this section demonstrates their utility. For any long-running goroutines, or goroutines that need to be tested, I highly recommend this pattern.

Replicated Requests

For some applications, receiving a response as quickly as possible is the top priority. For example, maybe the application is servicing a user's HTTP request, or retrieving a replicated blob of data. In these instances you can make a trade-off: you can replicate the request to multiple handlers (whether those be goroutines, processes, or servers), and one of them will return faster than the other ones; you can then immediately return the result. The downside is that you'll have to utilize resources to keep multiple copies of the handlers running.

If this replication is done in-memory, it might not be that costly, but if replicating the handlers requires replicating processes, servers, or even data centers, this can become quite costly. The decision you'll have to make is whether or not the cost is worth the benefit.

Let's look at how you can replicate requests within a single process. We'll use multiple goroutines to serve as request handlers, and the goroutines will sleep for a random amount of time between one and six nanoseconds to simulate load. This will give us handlers that return a result at various times and will allow us to see how this can lead to faster results.

Here's an example that replicates a simulated request over 10 handlers:

```
doWork := func(
    done <-chan interface{},
    id int,
    wg *sync.WaitGroup,
    result chan<- int,
) {
    started := time.Now()
    defer wg.Done()

    // Simulate random load
    simulatedLoadTime := time.Duration(1+rand.Intn(5))*time.Second
    select {
    case <-done:
    case <-time.After(simulatedLoadTime):
    }

    select {
    case <-done:
    case result <- id:
    }

    took := time.Since(started)
    // Display how long handlers would have taken
    if took < simulatedLoadTime {
        took = simulatedLoadTime
    }
    fmt.Printf("%v took %v\n", id, took)
}

done := make(chan interface{})
result := make(chan int)

var wg sync.WaitGroup
wg.Add(10)

for i:=0; i < 10; i++ { ❶
    go doWork(done, i, &wg, result)
}

firstReturned := <-result ❷
close(done) ❸
wg.Wait()

fmt.Printf("Received an answer from #%v\n", firstReturned)
```

❶ Here we start 10 handlers to handle our requests.

❷ This line grabs the first returned value from the group of handlers.

❸ Here we cancel all the remaining handlers. This ensures they don't continue to do unnecessary work.

Running this code produces:

```
8 took 1.000211046s
4 took 3s
9 took 2s
1 took 1.000568933s
7 took 2s
3 took 1.000590992s
5 took 5s
0 took 3s
6 took 4s
2 took 2s
Received an answer from #8
```

In this run, it looks like handler #8 returned fastest. Note that in the output we're displaying how long each handler *would* have taken so that you can get a sense of how much time this technique can save. Imagine if you only spun up one handler and it happened to be handler #5. Instead of waiting just over a second for the request to be handled, you would have had to wait for five seconds.

The only caveat to this approach is that all of your handlers need to have equal opportunity to service the request. In other words, you're not going to have a chance at receiving the fastest time from a handler that *can't* service the request. As I mentioned, whatever resources the handlers are using to do their job need to be replicated as well.

A different symptom of the same problem is uniformity. If your handlers are too much alike, the chances that any one will be an outlier is smaller. You should only replicate out requests like this to handlers that have different runtime conditions: different processes, machines, paths to a data store, or access to different data stores altogether.

Although this is can be expensive to set up and maintain, if speed is your goal, this is a valuable technique. In addition, this naturally provides fault tolerance and scalability.

Rate Limiting

If you've ever worked with an API for a service, you've likely had to contend with rate limiting, which constrains the number of times some kind of resource is accessed to some finite number per unit of time. The resource can be anything: API connections, disk reads/writes, network packets, errors.

Have you ever wondered why services put rate limits in place? Why not allow unfettered access to a system? The most obvious answer is that by rate limiting a system, you prevent entire classes of attack vectors against your system. If malicious users can access your system as quickly as their resources allow it, they can do all kinds of things.

For example, they could fill up your service's disk either with log messages or valid requests. If you've misconfigured your log rotation, they could even perform something malicious and then make enough requests that any record of the activity would be rotated out of the log and into `/dev/null`. They could attempt to brute-force access to a resource, or maybe they would just perform a distributed denial of service attack. The point is: if you don't rate limit requests to your system, you cannot easily secure it.

Malicious use isn't the only reason. In distributed systems, a legitimate user could degrade the performance of the system for other users if they're performing operations at a high enough volume, or if the code they're exercising is buggy. This can even cause the death-spirals we discussed earlier. From a product standpoint, this is awful! Usually you want to make some kind of guarantees to your users about what kind of performance they can expect on a consistent basis. If one user can affect that agreement, you're in for a bad time. A user's mental model is usually that their access to the system is sandboxed and can neither affect nor be affected by other users' activities. If you break that mental model, your system can feel like it's not well engineered, and even cause users to become angry or leave.

Even with only *one* user, rate limits can be advantageous. A lot of the time, systems have been developed to work well under the common use case, but may begin behaving differently under different circumstances. In complicated systems such as distributed systems, this effect can cascade through the system and have drastic, unintended consequences. Maybe under load you begin dropping packets, which causes your distributed database to lose its quorum and stop accepting writes, which causes your existing requests to fail, which causes… You can see how this can be a bad thing. It isn't unheard of for systems to perform a kind of DDoS attack on themselves in these instances!

A Story from the Field

I once worked on a distributed system that scaled work in parallel by starting new processes (this allowed it to scale horizontally to multiple machines). Each process would open a database connection, read some data, and do some calculations. For a time, we had great success in scaling the system in this manner to meet the needs of clients. However, after a while the system utilization grew to a point where reads from the database were timing out.

Our database administrators pored over logs to try and figure out what was going wrong. In the end, they discovered that because there were no rate limits set for anything on the system, processes were stomping all over each other. Disk contention would spike to 100% and remain there as different processes attempted to read data from different parts of the disk. This in turn led to a kind of sadistic round-robin timeout-retry loop. Jobs would never complete.

> A system was devised to place limits on the number of connections possible on the database, and rate limits were placed on bits per second a connection could read, and the problems went away. Clients had to wait longer for their jobs to complete, but they completed, and we were able to perform proper capacity planning to expand the capacity of the system in a structured way.

Rate limits allow you to reason about the performance and stability of your system by preventing it from falling outside the boundaries you've already investigated. If you need to expand those boundaries, you can do so in a controlled manner after lots of testing and coffee.

In scenarios where you're charging for access to your system, rate limits can maintain a healthy relationship with your clients. You can allow them to try the system out under heavily constrained rate limits. Google does this with its cloud offerings to great success.

After they've become paying customers, rate limits can even *protect* your users. Because most of the time access to the system is programmatic, it's very easy to introduce a bug that accesses your paid system in a runaway manner. This can be a *very* costly mistake and leaves both parties in the awkward situation of deciding what to do: does the service owner eat the cost and forgive the unintended access, or is the user forced to pay the bill, which might sour the relationship permanently?

Rate limits are often thought of from the perspective of people who *build* the resources being limited, but rate limiting can also be utilized by users. If I'm only just understanding how to utilize a service's API, it would be very comforting to be able to scale the rate limits way down so I know I won't shoot myself in the foot.

Hopefully I've given enough justification to convince you that rate limits are good even if you set limits that you think will never be reached. They're pretty simple to create, and they solve so many problems that it's hard to rationalize *not* using them.

So how do we go about implementing rate limits in Go?

Most rate limiting is done by utilizing an algorithm called the *token bucket*. It's very easy to understand, and relatively easy to implement as well. Let's take a look at the theory behind it.

Let's assume that to utilize a resource, you have to have an *access token* for the resource. Without the token, your request is denied. Now imagine these tokens are stored in a bucket waiting to be retrieved for usage. The bucket has a depth of `d`, which indicates it can hold `d` access tokens at a time. For example, if the bucket has a depth of five, it can hold five tokens.

Now, every time you need to access a resource, you reach into the bucket and remove a token. If your bucket contains five tokens, and you access the resource five times,

you'd be able to do so; but on the sixth try, no access token would be available. You either have to queue your request until a token becomes available, or deny the request.

Here's a time table to help visualize the concept. `time` represents the time-delta in seconds, `bucket` represents the number of request tokens in the bucket, and a `tok` in the `request` column denotes a successful request. (In this and future time tables, we'll assume the requests are instantaneous to simplify the visualization.)

time	bucket	request
0	5	tok
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	
1	0	
	0	

You can see that we're able to make all five requests before the first second, and then we are blocked as no more tokens are available for use.

So far, this is pretty straightforward. What about replenishing the tokens; do we ever get new ones? In the token bucket algorithm, we define `r` to be the rate at which tokens are added *back* to the bucket. It can be one a nanosecond, or one a minute. This becomes what we commonly think of as the rate limit: because we have to wait until new tokens become available, we limit our operations to that refresh rate.

Here's an example of a token bucket with a depth of one, and a rate of `1 tokens/second`:

time	bucket	request
0	1	
0	0	tok
1	0	
2	1	
2	0	tok
3	0	
4	1	
4	0	tok

You can see that we're immediately able to make a request, but we are then limited to one request every other second. Our rate limitation is working beautifully!

So we now have two settings we can fiddle with: how many tokens are available for immediate use—`d`, the depth of the bucket—and the rate at which they are replenished—`r`. Between these two we can control both the *burstiness* and overall rate limit. Burstiness simply means how many requests can be made when the bucket is full.

Here's an example of a token bucket with a depth of five, and a rate of `0.5 tokens/second`:

time	bucket	request
0	5	
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	tok
1	0 (0.5)	
2	1	
2	0	tok
3	0 (0.5)	
4	1	
4	0	tok

Here, we were able to immediately make five requests, after which point we were limited to a request every two seconds. Our *burst* was at the beginning.

Be aware that users may not consume the entire bucket of tokens in one long stream. The depth of the bucket only controls the bucket's capacity. Here's an example of a user who had a burst of two, and then four seconds later, had a burst of five:

time	bucket	request
0	5	
0	4	tok
0	3	tok
1	3	
2	4	
3	5	
4	5	
5	4	tok

time	bucket	request
5	3	tok
5	2	tok
5	1	tok
5	0	tok

While a user has tokens available, burstiness allows access to the system constrained only by the capabilities of the caller. For users who only access the system intermittently, but want to round-trip as quickly as possible when they do, bursts are nice to have. You just need to either ensure your system can handle all users bursting at once, or that it is statistically improbable that enough users will burst at the same time to affect your system. Either way, a rate limit allows you to take a calculated risk.

Let's put this algorithm to use and see how a Go program might behave when written against an implementation of the token bucket algorithm.

Let's pretend we have access to an API, and a Go client has been provided to utilize it. This API has two endpoints: one for reading a file, and one for resolving a domain name to an IP address. For simplicity's sake, I'm going to leave off any arguments and return values that would be needed to actually access a service. So here's our client:

```
func Open() *APIConnection {
    return &APIConnection{}
}

type APIConnection struct {}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    // Pretend we do work here
    return nil
}
```

Since in theory this request is going over the wire, we take a `context.Context` in as the first argument in case we need to cancel the request or pass values over to the server. Pretty standard stuff.

We'll now create a simple driver to access this API. The driver needs to read 10 files and resolve 10 addresses, but the files and addresses have no relation to each other and so the driver can make these API calls concurrent to one another. Later this will help stress our `APIClient` and exercise our rate limiter.

```
func main() {
    defer log.Printf("Done.")
```

```
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime | log.LUTC)

    apiConnection := Open()
    var wg sync.WaitGroup
    wg.Add(20)

    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            err := apiConnection.ReadFile(context.Background())
            if err != nil {
                log.Printf("cannot ReadFile: %v", err)
            }
            log.Printf("ReadFile")
        }()
    }

    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            err := apiConnection.ResolveAddress(context.Background())
            if err != nil {
                log.Printf("cannot ResolveAddress: %v", err)
            }
            log.Printf("ResolveAddress")
        }()
    }

    wg.Wait()
}
```

Running this code produces:

```
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
```

```
20:13:13 ReadFile
20:13:13 Done.
```

We can see that all API requests are fielded almost simultaneously. We have no rate limiting set up and so our clients are free to access the system as frequently as they like. Now is a good time to remind you that a bug could exist in our driver that could result in an infinite loop. Without rate limiting, I could be staring down a nasty bill.

OK, so let's introduce a rate limiter! I'm going to do so within the `APIConnection`, but normally a rate limiter would be running on a server so the users couldn't trivially bypass it. Production systems might *also* include a client-side rate limiter to help prevent the client from making unnecessary calls only to be denied, but that is an optimization. For our purposes, a client-side rate limiter keeps things simple.

We're going to be looking at examples that use an implementation of a token bucket rate limiter from the `golang.org/x/time/rate` package. I chose this package because this is as close to the standard library as I could get. There are certainly other packages out there that do the same thing with more bells and whistles, and those may serve you better for use in production systems. The `golang.org/x/time/rate` package is pretty simple, so it should work well for our purposes.

The first two ways we'll interact with this package are the `Limit` type and the `NewLimiter` function, defined here:

```
// Limit defines the maximum frequency of some events.  Limit is
// represented as number of events per second.  A zero Limit allows no
// events.
type Limit float64

// NewLimiter returns a new Limiter that allows events up to rate r
// and permits bursts of at most b tokens.
func NewLimiter(r Limit, b int) *Limiter
```

In `NewLimiter`, we see two familiar parameters: `r` and `b`. `r` is the rate we discussed previously, and `b` is the bucket depth we discussed.

The `rates` package also defines a helper method, `Every`, to assist in converting a `time.Duration` into a `Limit`:

```
// Every converts a minimum time interval between events to a Limit.
func Every(interval time.Duration) Limit
```

The `Every` function makes sense, but I want to discuss rate limits in terms of the number of operations per time measurement, not the interval between requests. We can express this as the following:

```
rate.Limit(events/timePeriod.Seconds())
```

But I don't want to type that every time, and the `Every` function has some special logic that will return `rate.Inf`—an indication that there is no limit—if the interval

provided is zero. Because of this, we'll express our helper function in terms of the `Every` function:

```
func Per(eventCount int, duration time.Duration) rate.Limit {
    return rate.Every(duration/time.Duration(eventCount))
}
```

After we create a `rate.Limiter`, we'll want to use it to block our requests until we're given an access token. We can do that with the `Wait` method, which simply calls `WaitN` with an argument of `1`:

```
// Wait is shorthand for WaitN(ctx, 1).
func (lim *Limiter) Wait(ctx context.Context)

// WaitN blocks until lim permits n events to happen.
// It returns an error if n exceeds the Limiter's burst size, the Context is
// canceled, or the expected wait time exceeds the Context's Deadline.
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)
```

We should now have all the ingredients we'll need to begin rate limiting our API requests. Let's modify our `APIConnection` type and give it a try!

```
func Open() *APIConnection {
    return &APIConnection{
        rateLimiter: rate.NewLimiter(rate.Limit(1), 1), ❶
    }
}

type APIConnection struct {
    rateLimiter *rate.Limiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { ❷
        return err
    }
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { ❷
        return err
    }
    // Pretend we do work here
    return nil
}
```

❶ Here we set the rate limit for all API connections to one event per second.

❷ Here we wait on the rate limiter to have enough access tokens for us to complete our request.

Running this code produces:

```
22:08:30 ResolveAddress
22:08:31 ReadFile
22:08:32 ReadFile
22:08:33 ReadFile
22:08:34 ResolveAddress
22:08:35 ResolveAddress
22:08:36 ResolveAddress
22:08:37 ResolveAddress
22:08:38 ResolveAddress
22:08:39 ReadFile
22:08:40 ResolveAddress
22:08:41 ResolveAddress
22:08:42 ResolveAddress
22:08:43 ResolveAddress
22:08:44 ReadFile
22:08:45 ReadFile
22:08:46 ReadFile
22:08:47 ReadFile
22:08:48 ReadFile
22:08:49 ReadFile
22:08:49 Done.
```

You can see that whereas before we were fielding all of our API requests simultaneously, we're now completing a request once a second. It looks like our rate limiter is working!

This gets us very basic rate limiting, but in production we're likely going to want something a little more complex. We will probably want to establish multiple tiers of limits: fine-grained controls to limit requests per second, and coarse-grained controls to limit requests per minute, hour, or day.

In certain instances, it's possible to do this with a single rate limiter; however, it's not possible in all cases, and by attempting to roll the semantics of limits per unit of time into a single layer, you lose a lot of information around the intent of the rate limiter. For these reasons, I find it easier to keep the limiters separate and then combine them into one rate limiter that manages the interaction for you. To this end I've created a simple aggregate rate limiter called `multiLimiter`. Here is the definition:

```
type RateLimiter interface { ❶
    Wait(context.Context) error
    Limit() rate.Limit
}

func MultiLimiter(limiters ...RateLimiter) *multiLimiter {
    byLimit := func(i, j int) bool {
        return limiters[i].Limit() < limiters[j].Limit()
    }
    sort.Slice(limiters, byLimit) ❷
    return &multiLimiter{limiters: limiters}
```

```
}

type multiLimiter struct {
    limiters []RateLimiter
}

func (l *multiLimiter) Wait(ctx context.Context) error {
    for _, l := range l.limiters {
        if err := l.Wait(ctx); err != nil {
            return err
        }
    }
    return nil
}

func (l *multiLimiter) Limit() rate.Limit {
    return l.limiters[0].Limit() ❸
}
```

❶ Here we define a `RateLimiter` interface so that a `MultiLimiter` can recursively define other `MultiLimiter` instances.

❷ Here we implement an optimization and sort by the `Limit()` of each `RateLimiter`.

❸ Because we sort the child `RateLimiter` instances when `multiLimiter` is instantiated, we can simply return the most restrictive limit, which will be the first element in the slice.

The `Wait` method loops through all the child rate limiters and calls `Wait` on each of them. These calls may or may not block, but we need to notify each rate limiter of the request so we can decrement our token bucket. By waiting for each limiter, we are guaranteed to wait for exactly the time of the longest wait. This is because if we perform smaller waits that only wait for segments of the longest wait and then hit the longest wait, the longest wait will be recalculated to only be the remaining time. This is because while the earlier waits were blocking, the latter waits were refilling their buckets; any waits after will be returned instantaneously.

Now that we have the means to express rate limits from multiple rate limits, let's take the opportunity to do so. Let's redefine our `APIConnection` to have limits both per second and per minute:

```
func Open() *APIConnection {
    secondLimit := rate.NewLimiter(Per(2, time.Second), 1) ❶
    minuteLimit := rate.NewLimiter(Per(10, time.Minute), 10) ❷
    return &APIConnection{
        rateLimiter: MultiLimiter(secondLimit, minuteLimit), ❸
    }
}
```

```
type APIConnection struct {
    rateLimiter RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}
```

❶ Here we define our limit per second with no burstiness.

❷ Here we define our limit per minute with a burstiness of 10 to give the users their initial pool. The limit per second will ensure we don't overload our system with requests.

❸ We then combine the two limits and set this as the master rate limiter for our `APIConnection`.

Running this code produces:

```
22:46:10 ResolveAddress
22:46:10 ReadFile
22:46:11 ReadFile
22:46:11 ReadFile
22:46:12 ReadFile
22:46:12 ReadFile
22:46:13 ReadFile
22:46:13 ReadFile
22:46:14 ReadFile
22:46:14 ReadFile
22:46:16 ResolveAddress
22:46:22 ResolveAddress
22:46:28 ReadFile
22:46:34 ResolveAddress
22:46:40 ResolveAddress
22:46:46 ResolveAddress
22:46:52 ResolveAddress
22:46:58 ResolveAddress
22:47:04 ResolveAddress
```

```
22:47:10 ResolveAddress
22:47:10 Done.
```

As you can see we make two requests per second up until request #11, at which point we begin making requests every six seconds. This is because we drained our available pool of per-minute request tokens, and become limited by this cap.

It might be slightly counterintuitive why request #11 occurs after only two seconds rather than six like the rest of the requests. Remember that although we limit our API requests to 10 a minute, that minute is a sliding window of time. By the time we reach the eleventh request, our per-minute rate limiter has accrued another token.

Defining limits like this allows us to express our coarse-grained limits plainly while still limiting the number of requests at a finer level of detail.

This technique also allows us to begin thinking across dimensions other than time. When you rate limit a system, you're probably going to limit more than one thing. You'll likely have some kind of limit on the number of API requests, but in addition, you'll probably also have limits on other resources like disk access, network access, etc. Let's flesh out our example a bit and set up rate limits for disk and network:

```
func Open() *APIConnection {
    return &APIConnection{
        apiLimit: MultiLimiter( ❶
            rate.NewLimiter(Per(2, time.Second), 2),
            rate.NewLimiter(Per(10, time.Minute), 10),
        ),
        diskLimit: MultiLimiter( ❷
            rate.NewLimiter(rate.Limit(1), 1),
        ),
        networkLimit: MultiLimiter( ❸
            rate.NewLimiter(Per(3, time.Second), 3),
        ),
    }
}

type APIConnection struct {
    networkLimit,
    diskLimit,
    apiLimit RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.diskLimit).Wait(ctx) ❹
    if err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}
```

```
func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.networkLimit).Wait(ctx) ❺
    if err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}
```

❶ Here we set up a rate limiter for API calls. There are limits for both requests per second and requests per minute.

❷ Here we set up a rate limiter for disk reads. We'll only limit this to one read per second.

❸ For networking, we'll set up a limit of three requests per second.

❹ When we go to read a file, we'll combine the limits from the API limiter and the disk limiter.

❺ When we require network access, we'll combine the limits from the API limiter and the network limiter.

Running this code produces:

```
01:40:15 ResolveAddress
01:40:15 ReadFile
01:40:16 ReadFile
01:40:17 ResolveAddress
01:40:17 ResolveAddress
01:40:17 ReadFile
01:40:18 ResolveAddress
01:40:18 ResolveAddress
01:40:19 ResolveAddress
01:40:19 ResolveAddress
01:40:21 ResolveAddress
01:40:27 ResolveAddress
01:40:33 ResolveAddress
01:40:39 ReadFile
01:40:45 ReadFile
01:40:51 ReadFile
01:40:57 ReadFile
01:41:03 ReadFile
01:41:09 ReadFile
01:41:15 ReadFile
01:41:15 Done.
```

I could build another time table here to break down why each call is happening where, but that would miss the point. Instead, let's focus on the fact that we're able to compose logical rate limiters into groups that make sense for each call, and the

`APIClient` does the correct thing. If we wanted to make a casual observation about how it's working, we could note that the API calls involving network access appear to happen with more regularity and finish in the first two-thirds of calls. This may have to do with when the goroutines are scheduled, but it's much more likely that our rate limiters are doing their jobs!

I should also mention that the `rate.Limiter` type has a few other tricks up its sleeve for optimizations and different use cases. I have only discussed its ability to wait until the token bucket receives another token, but if you're interested in using it, just know that it has a few other capabilities.

In this section, we've looked at the justification for utilizing rate limits, an algorithm for building one, a Go implementation of the token bucket algorithm, and how to compose token bucket limiters into larger, more complex rate limiters. This should give you a good overview of rate limits, and help you get started using them in the field.

Healing Unhealthy Goroutines

In long-lived processes such as daemons, it's very common to have a set of long-lived goroutines. These goroutines are usually blocked, waiting on data to come to them through some means, so that they can wake up, do their work, and then pass the data on. Sometimes the goroutines are dependent on a resource that you don't have very good control of. Maybe a goroutine receives a request to pull data from a web service, or maybe it's monitoring an ephemeral file. The point is that it can be very easy for a goroutine to become stuck in a bad state from which it cannot recover without external help. If you separate your concerns, you might even say that it shouldn't be the concern of a goroutine doing work to know how to heal itself from a bad state. In a long-running process, it can be useful to create a mechanism that ensures your goroutines remain healthy and restarts them if they become unhealthy. We'll refer to this process of restarting goroutines as "healing."[2]

To heal goroutines, we'll use our heartbeat pattern to check up on the liveliness of the goroutine we're monitoring. The type of heartbeat will be determined by what you're trying to monitor, but if your goroutine can become livelocked, make sure that the heartbeat contains some kind of information indicating that the goroutine is not only up, but doing useful work. In this section, for simplicity, we'll only consider whether goroutines are live or dead.

We'll call the logic that monitors a goroutine's health a *steward*, and the goroutine that it monitors a *ward*. Stewards will also be responsible for restarting a ward's goroutine

2 Those of you familiar with erlang may recognize this concept! Erlang's supervisors do much the same thing.

should it become unhealthy. To do so, it will need a reference to a function that can start the goroutine. Let's see what a steward might look like:

```
type startGoroutineFn func(
    done <-chan interface{},
    pulseInterval time.Duration,
) (heartbeat <-chan interface{}) ❶

newSteward := func(
    timeout time.Duration,
    startGoroutine startGoroutineFn,
) startGoroutineFn { ❷
    return func(
        done <-chan interface{},
        pulseInterval time.Duration,
    ) (<-chan interface{}) {
        heartbeat := make(chan interface{})
        go func() {
            defer close(heartbeat)

            var wardDone chan interface{}
            var wardHeartbeat <-chan interface{}
            startWard := func() { ❸
                wardDone = make(chan interface{}) ❹
                wardHeartbeat = startGoroutine(or(wardDone, done), timeout/2) ❺
            }
            startWard()
            pulse := time.Tick(pulseInterval)

        monitorLoop:
            for {
                timeoutSignal := time.After(timeout)

                for { ❻
                    select {
                    case <-pulse:
                        select {
                        case heartbeat <- struct{}{}:
                        default:
                        }
                    case <-wardHeartbeat: ❼
                        continue monitorLoop
                    case <-timeoutSignal: ❽
                        log.Println("steward: ward unhealthy; restarting")
                        close(wardDone)
                        startWard()
                        continue monitorLoop
                    case <-done:
                        return
                    }
                }
            }
```

```
        }()

        return heartbeat
    }
}
```

❶ Here we define the signature of a goroutine that can be monitored and restarted. We see the familiar `done` channel, and `pulseInterval` and `heartbeat` from the heartbeat pattern.

❷ On this line we see that a steward takes in a `timeout` for the goroutine it will be monitoring, and a function, `startGoroutine`, to start the goroutine it's monitoring. Interestingly, the steward itself returns a `startGoroutineFn` indicating that the steward itself is also monitorable.

❸ Here we define a closure that encodes a consistent way to start the goroutine we're monitoring.

❹ This is where we create a new channel that we'll pass into the ward goroutine in case we need to signal that it should halt.

❺ Here we start the goroutine we'll be monitoring. We want the ward goroutine to halt if either the steward is halted, or the steward wants to halt the ward goroutine, so we wrap both `done` channels in a logical-or. The `pulseInterval` we pass in is half of the timeout period, although as we discussed in "Heartbeats" on page 161, this can be tweaked.

❻ This is our inner loop, which ensures that the steward can send out pulses of its own.

❼ Here we see that if we receive the ward's pulse, we continue our monitoring loop.

❽ This line indicates that if we don't receive a pulse from the ward within our timeout period, we request that the ward halt and we begin a new ward goroutine. We then continue monitoring.

Our `for` loop is a *little* busy, but as long as you're familiar with the patterns involved, it's relatively straightforward to read through. Let's give our steward a test run. What happens if we monitor a goroutine that is misbehaving? Let's take a look:

```
log.SetOutput(os.Stdout)
log.SetFlags(log.Ltime | log.LUTC)

doWork := func(done <-chan interface{}, _ time.Duration) <-chan interface{} {
    log.Println("ward: Hello, I'm irresponsible!")
    go func() {
```

```
            <-done ❶
            log.Println("ward: I am halting.")
        }()
        return nil
    }
    doWorkWithSteward := newSteward(4*time.Second, doWork) ❷

    done := make(chan interface{})
    time.AfterFunc(9*time.Second, func() { ❸
        log.Println("main: halting steward and ward.")
        close(done)
    })

    for range doWorkWithSteward(done, 4*time.Second) {} ❹
    log.Println("Done")
```

❶ Here we see that this goroutine isn't doing anything but waiting to be canceled. It's also not sending out any pulses.

❷ This line creates a function that will create a steward for the goroutine `doWork` starts. We set the timeout for `doWork` at four seconds.

❸ Here we halt the steward and its ward after nine seconds so that our example will end.

❹ Finally, we start the steward and range over its pulses to prevent our example from halting.

This example produces the following output:

```
18:28:07 ward: Hello, I'm irresponsible!
18:28:11 steward: ward unhealthy; restarting
18:28:11 ward: Hello, I'm irresponsible!
18:28:11 ward: I am halting.
18:28:15 steward: ward unhealthy; restarting
18:28:15 ward: Hello, I'm irresponsible!
18:28:15 ward: I am halting.
18:28:16 main: halting steward and ward.
18:28:16 ward: I am halting.
18:28:16 Done
```

It looks like this is working quite nicely! Our ward is a little simplistic though: other than what's necessary for cancellation and heartbeats, it takes in no parameters and returns no arguments. How might we create a ward that has a shape that can be used with our steward? We could rewrite or generate the steward to fit our wards each time, but this is both cumbersome and unnecessary; instead, we'll use closures. Let's take a look at a ward that will generate an integer stream based on a discrete list of values:

```
doWorkFn := func(
    done <-chan interface{},
    intList ...int,
) (startGoroutineFn, <-chan interface{}) { ❶
    intChanStream := make(chan (<-chan interface{})) ❷
    intStream := bridge(done, intChanStream)
    doWork := func(
        done <-chan interface{},
        pulseInterval time.Duration,
    ) <-chan interface{} { ❸
        intStream := make(chan interface{}) ❹
        heartbeat := make(chan interface{})
        go func() {
            defer close(intStream)
            select {
            case intChanStream <- intStream: ❺
            case <-done:
                return
            }

            pulse := time.Tick(pulseInterval)

            for {
                valueLoop:
                for _, intVal := range intList {
                    if intVal < 0 {
                        log.Printf("negative value: %v\n", intVal) ❻
                        return
                    }

                    for {
                        select {
                        case <-pulse:
                            select {
                            case heartbeat <- struct{}{}:
                            default:
                            }
                        case intStream <- intVal:
                            continue valueLoop
                        case <-done:
                            return
                        }
                    }
                }
            }
        }()
        return heartbeat
    }
    return doWork, intStream
}
```

❶ Here we'll take in the values we want our ward to close over, and return any channels our ward will be using to communicate back on.

❷ This line creates our channel of channels as part of the bridge pattern.

❸ Here we create the closure that will be started and monitored by our steward.

❹ This is where we instantiate the channel we'll communicate on within this instance of our ward's goroutine.

❺ Here we let the bridge know about the new channel we'll be communicating on.

❻ This line simulates an unhealthy ward by logging an error when we encounter a negative number and returning from the goroutine.

You can see that since we'll potentially be starting multiple copies of our ward, we make use of bridge channels (see "The bridge-channel" on page 122) to help present a single uninterrupted channel to the consumer of `doWork`. Using these techniques, our wards can become arbitrarily complex simply by composing patterns. Let's see how utilizing this feels:

```
log.SetFlags(log.Ltime | log.LUTC)
log.SetOutput(os.Stdout)

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

doWork, intStream := doWorkFn(done, 1, 2, -1, 3, 4, 5) ❶
doWorkWithSteward := newSteward(1*time.Millisecond, doWork) ❷
doWorkWithSteward(done, 1*time.Hour) ❸

for intVal := range take(done, intStream, 6) { ❹
    fmt.Printf("Received: %v\n", intVal)
}
```

❶ This line creates our ward's function, allowing it to close over our variadic slice of integers, and return a stream that it will communicate back on.

❷ Here we create our steward that will monitor the `doWork` closure. Because we expect failures fairly quickly, we'll set the monitoring period at just one millisecond.

❸ Here we tell the steward to start the ward and begin monitoring.

❹ Finally, we use one of the pipeline stages we developed and take the first six values from our `intStream`.

Running this code produces:

```
Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting
Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting
Received: 1
23:25:33 negative value: -1
Received: 2
```

Interspersed with the values we receive, we see errors from the ward, and our steward detecting them and restarting the ward. You might also notice that we only ever receive values `1` and `2`. This is a symptom of our ward starting from scratch every time. When developing your wards, if your system is sensitive to duplicate values, be sure to take that into account. You might also consider writing a steward that exits after a certain number of failures. In this case, we could have simply made our generator stateful by updating the `intList` we are closed over in every iteration. Whereas before we had this:

```
valueLoop:
for _, intVal := range intList {
    // ...
}
```

We could instead write this:

```
valueLoop:
    for {
        intVal := intList[0]
        intList = intList[1:]
        // ...
    }
```

This would save our place between our ward's restarts, although we would remain stuck at our invalid negative number, and our ward would continue to fail.

Using this pattern can help ensure your long-lived goroutines stay up and healthy.

Summary

In this chapter, we've covered some ways to keep your systems stable and understandable as the problem domains they take on necessitate larger systems that are perhaps distributed. This chapter also demonstrated how Go's concurrency primitives scale as you create higher-order abstractions. Without the benefit of a language designed around concurrency, these patterns would likely be much more cumbersome, and much less robust.

In the final chapter, we're going to explore the internals of some of Go's runtime to help you develop a deep understanding of how things work. We'll also explore some useful tools that will make the job of developing and debugging Go software a bit easier.