CHAPTER 14

The Context

Servers need a way to handle metadata on individual requests. This metadata falls into two general categories: metadata that is required to correctly process the request, and metadata on when to stop processing the request. For example, an HTTP server might want to use a tracking ID to identify a chain of requests through a set of microservices. It also might want to set a timer that ends requests to other microservices if they take too long.

Many languages use *threadlocal* variables to store this kind of information, associating data to a specific operating system thread of execution. This doesn't work in Go because goroutines don't have unique identities that can be used to look up values. More importantly, threadlocals feel like magic; values go in one place and pop up somewhere else.

Go solves the request metadata problem with a construct called the *context*. Let's see how to use it correctly.

What Is the Context?

Rather than add a new feature to the language, a context is simply an instance that meets the `Context` interface defined in the `context` package. As you know, idiomatic Go encourages explicit data passing via function parameters. The same is true for the context. It is just another parameter to your function.

Just as Go has a convention that the last return value from a function is an `error`, Go has another convention that the context is explicitly passed through your program as the first parameter of a function. The usual name for the context parameter is `ctx`:

```
func logic(ctx context.Context, info string) (string, error) {
    // do some interesting stuff here
    return "", nil
}
```

In addition to defining the `Context` interface, the `context` package contains several factory functions for creating and wrapping contexts. When you don't have an existing context, such as at the entry point to a command-line program, create an empty initial context with the function `context.Background`. This returns a variable of type `context.Context`. (Yes, this is an exception to the usual pattern of returning a concrete type from a function call.)

An empty context is a starting point; each time you add metadata to the context, you do so by *wrapping* the existing context by using one of the factory functions in the `context` package.

Another function, `context.TODO`, also creates an empty `context.Context`. It is intended for temporary use during development. If you aren't sure where the context is going to come from or how it's going to be used, use `context.TODO` to put a placeholder in your code. Production code shouldn't include `context.TODO`.

When writing an HTTP server, you use a slightly different pattern for acquiring and passing the context through layers of middleware to the top-level `http.Handler`. Unfortunately, context was added to the Go APIs long after the `net/http` package was created. Because of the compatibility promise, there was no way to change the `http.Handler` interface to add a `context.Context` parameter.

The compatibility promise does allow new methods to be added to existing types, and that's what the Go team did. Two context-related methods are on `http.Request`:

- `Context` returns the `context.Context` associated with the request.
- `WithContext` takes in a `context.Context` and returns a new `http.Request` with the old request's state combined with the supplied `context.Context`.

Here's the general pattern:

```
func Middleware(handler http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        // wrap the context with stuff -- you'll see how soon!
        req = req.WithContext(ctx)
        handler.ServeHTTP(rw, req)
    })
}
```

The first thing you do in your middleware is extract the existing context from the request by using the `Context` method. (If you want to skip ahead, you can see how to put values into the context in "Values" on page 352.) After you put values into the context, you create a new request based on the old request and the now-populated context by using the `WithContext` method. Finally, you call the `handler` and pass it your new request and the existing `http.ResponseWriter`.

When you implement the handler, extract the context from the request by using the `Context` method and call your business logic with the context as the first parameter, just as you saw previously:

```
func handler(rw http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    err := req.ParseForm()
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    data := req.FormValue("data")
    result, err := logic(ctx, data)
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    rw.Write([]byte(result))
}
```

When making an HTTP call from your application to another HTTP service, use the `NewRequestWithContext` function in the `net/http` package to construct a request that includes existing context information:

```
type ServiceCaller struct {
    client *http.Client
}

func (sc ServiceCaller) callAnotherService(ctx context.Context, data string)
                                    (string, error) {
    req, err := http.NewRequestWithContext(ctx, http.MethodGet,
                "http://example.com?data="+data, nil)
    if err != nil {
        return "", err
    }
    resp, err := sc.client.Do(req)
    if err != nil {
        return "", err
    }
    defer resp.Body.Close()
    if resp.StatusCode != http.StatusOK {
        return "", fmt.Errorf("Unexpected status code %d",
                                 resp.StatusCode)
    }
    // do the rest of the stuff to process the response
    id, err := processResponse(resp.Body)
    return id, err
}
```

You can find these code samples in the *sample_code/context_patterns* directory in the Chapter 14 repository.

Now that you know how to acquire and pass a context, let's start making them useful. You'll begin with passing values.

Values

By default, you should prefer to pass data through explicit parameters. As has been mentioned before, idiomatic Go favors the explicit over the implicit, and this includes explicit data passing. If a function depends on some data, it should be clear what data it needs and where that data came from.

However, in some cases you cannot pass data explicitly. The most common situation is an HTTP request handler and its associated middleware. As you have seen, all HTTP request handlers have two parameters, one for the request and one for the response. If you want to make a value available to your handler in middleware, you need to store it in the context. Possible situations include extracting a user from a JWT (JSON Web Token) or creating a per-request GUID that is passed through multiple layers of middleware and into your handler and business logic.

There is a factory method for putting values into the context, `context.WithValue`. It takes in three values: a context, a key to look up the value, and the value itself. The key and the value parameters are declared to be of type `any`. The `context.WithValue` function returns a context, but it is not the same context that was passed into the function. Instead, it is a *child* context that contains the key-value pair and *wraps* the passed-in *parent* `context.Context`.

You'll see this wrapping pattern several times. A context is treated as an immutable instance. Whenever you add information to a context, you do so by wrapping an existing parent context with a child context. This allows you to use contexts to pass information into deeper layers of the code. The context is never used to pass information out of deeper layers to higher layers.

The `Value` method on `context.Context` checks whether a value is in a context or any of its parent contexts. This method takes in a key and returns the value associated with the key. Again, both the key parameter and the value result are declared to be of type `any`. If no value is found for the supplied key, `nil` is returned. Use the comma ok idiom to type-assert the returned value to the correct type:

```
ctx := context.Background()
if myVal, ok := ctx.Value(myKey).(int); !ok {
    fmt.Println("no value")
} else {
    fmt.Println("value:", myVal)
}
```

If you are familiar with data structures, you might recognize that searching for values stored in the context chain is a *linear* search. This has no serious performance implications when there are only a few values, but it would perform poorly if you stored dozens of values in the context during a request. That said, if your program is creating a context chain with dozens of values, your program probably needs some refactoring.

The value stored in the context can be of any type, but picking the correct key is important. Like the key for a `map`, the key for a context value must be comparable. Don't just use a `string` like `"id"`. If you use `string` or another predefined or exported type for the type of the key, different packages could create identical keys, resulting in collisions. This causes problems that are hard to debug, such as one package writing data to the context that masks the data written by another package, or reading data from the context that was written by another package.

Two patterns are used to guarantee that a key is unique and comparable. The first creates a new, unexported type for the key, based on an `int`:

```
type userKey int
```

After declaring your unexported key type, you then declare an unexported constant of that type:

```
const (
    _ userKey = iota
    key
)
```

With both the type and the typed constant of the key being unexported, no code from outside your package can put data into the context that would cause a collision. If your package needs to put multiple values into the context, define a different key of the same type for each value, using the `iota` pattern you looked at in "iota Is for Enumerations—Sometimes" on page 152. Since you care about the constant's value only as a way to differentiate multiple keys, this is a perfect use for `iota`.

Next, build an API to place a value into the context and to read the value from the context. Make these functions public only if code outside your package should be able to read and write your context values. The name of the function that creates a context with the value should start with `ContextWith`. The function that returns the value from the context should have a name that ends with `FromContext`. Here are the implementations of functions to set and read the user from the context:

```go
func ContextWithUser(ctx context.Context, user string) context.Context {
    return context.WithValue(ctx, key, user)
}

func UserFromContext(ctx context.Context) (string, bool) {
    user, ok := ctx.Value(key).(string)
    return user, ok
}
```

Another option is to define the unexported key type by using an empty struct:

```go
type userKey struct{}
```

The functions for managing access to the context value are then changed:

```go
func ContextWithUser(ctx context.Context, user string) context.Context {
    return context.WithValue(ctx, userKey{}, user)
}

func UserFromContext(ctx context.Context) (string, bool) {
    user, ok := ctx.Value(userKey{}).(string)
    return user, ok
}
```

How do you know which key style to use? If you have a set of related keys for storing different values in the context, use the `int` and `iota` technique. If you have only a single key, either is fine. The important thing is that you want to make it impossible for context keys to collide.

Now that you've written your user-management code, let's see how to use it. You're going to write middleware that extracts a user ID from a cookie:

```go
// a real implementation would be signed to make sure
// the user didn't spoof their identity
func extractUser(req *http.Request) (string, error) {
    userCookie, err := req.Cookie("identity")
    if err != nil {
        return "", err
    }
    return userCookie.Value, nil
}

func Middleware(h http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        user, err := extractUser(req)
        if err != nil {
            rw.WriteHeader(http.StatusUnauthorized)
            rw.Write([]byte("unauthorized"))
            return
        }
        ctx := req.Context()
        ctx = ContextWithUser(ctx, user)
        req = req.WithContext(ctx)
        h.ServeHTTP(rw, req)
    })
}
```

In the middleware, you first get your user value. Next, you extract the context from the request with the `Context` method and create a new context that contains the user with your `ContextWithUser` function. It is idiomatic to reuse the `ctx` variable name when you wrap a context. You then make a new request from the old request and the new context by using the `WithContext` method. Finally, you call the next function in your handler chain with our new request and the supplied `http.ResponseWriter`.

In most cases, you want to extract the value from the context in your request handler and pass it in to your business logic explicitly. Go functions have explicit parameters, and you shouldn't use the context as a way to sneak values past the API:

```
func (c Controller) DoLogic(rw http.ResponseWriter, req *http.Request) {
    ctx := req.Context()
    user, ok := identity.UserFromContext(ctx)
    if !ok {
        rw.WriteHeader(http.StatusInternalServerError)
        return
    }
    data := req.URL.Query().Get("data")
    result, err := c.Logic.BusinessLogic(ctx, user, data)
    if err != nil {
        rw.WriteHeader(http.StatusInternalServerError)
        rw.Write([]byte(err.Error()))
        return
    }
    rw.Write([]byte(result))
}
```

Your handler gets the context by using the `Context` method on the request, extracts the user from the context by using the `UserFromContext` function, and then calls the business logic. This code shows the value of separation of concerns; how the user is loaded is unknown to the `Controller`. A real user-management system could be implemented in middleware and swapped in without changing any of the controller's code.

The complete code for this example is in the *sample_code/context_user* directory in the Chapter 14 repository.

In some situations, it's better to keep a value in the context. The tracking GUID that was mentioned earlier is one. This information is meant for management of your application; it is not part of your business state. Passing it explicitly through your code adds additional parameters and prevents integration with third-party libraries that do not know about your metainformation. By leaving a tracking GUID in the context, it passes invisibly through business logic that doesn't need to know about tracking and is available when your program writes a log message or connects to another server.

Here is a simple context-aware GUID implementation that tracks from service to service and creates logs with the GUID included:

```go
package tracker

import (
    "context"
    "fmt"
    "net/http"

    "github.com/google/uuid"
)

type guidKey int

const key guidKey = 1

func contextWithGUID(ctx context.Context, guid string) context.Context {
    return context.WithValue(ctx, key, guid)
}

func guidFromContext(ctx context.Context) (string, bool) {
    g, ok := ctx.Value(key).(string)
    return g, ok
}

func Middleware(h http.Handler) http.Handler {
    return http.HandlerFunc(func(rw http.ResponseWriter, req *http.Request) {
        ctx := req.Context()
        if guid := req.Header.Get("X-GUID"); guid != "" {
            ctx = contextWithGUID(ctx, guid)
        } else {
            ctx = contextWithGUID(ctx, uuid.New().String())
        }
        req = req.WithContext(ctx)
        h.ServeHTTP(rw, req)
    })
}

type Logger struct{}

func (Logger) Log(ctx context.Context, message string) {
    if guid, ok := guidFromContext(ctx); ok {
        message = fmt.Sprintf("GUID: %s - %s", guid, message)
    }
    // do logging
    fmt.Println(message)
}

func Request(req *http.Request) *http.Request {
    ctx := req.Context()
    if guid, ok := guidFromContext(ctx); ok {
        req.Header.Add("X-GUID", guid)
    }
    return req
}
```

The `Middleware` function either extracts the GUID from the incoming request or generates a new GUID. In both cases, it places the GUID into the context, creates a new request with the updated context, and continues the call chain.

Next you see how this GUID is used. The `Logger` struct provides a generic logging method that takes in a context and a string. If there's a GUID in the context, it appends it to the beginning of the log message and outputs it. The `Request` function is used when this service makes a call to another service. It takes in an `*http.Request`, adds a header with the GUID if it exists in the context, and returns the `*http.Request`.

Once you have this package, you can use the dependency injection techniques that I discussed in "Implicit Interfaces Make Dependency Injection Easier" on page 174 to create business logic that is completely unaware of any tracking information. First, you declare an interface to represent your logger, a function type to represent a request decorator, and a business logic struct that depends on them:

```
type Logger interface {
    Log(context.Context, string)
}

type RequestDecorator func(*http.Request) *http.Request

type LogicImpl struct {
    RequestDecorator RequestDecorator
    Logger           Logger
    Remote           string
}
```

Next, you implement your business logic:

```
func (l LogicImpl) Process(ctx context.Context, data string) (string, error) {
    l.Logger.Log(ctx, "starting Process with "+data)
    req, err := http.NewRequestWithContext(ctx,
        http.MethodGet, l.Remote+"/second?query="+data, nil)
    if err != nil {
        l.Logger.Log(ctx, "error building remote request:"+err.Error())
        return "", err
    }
    req = l.RequestDecorator(req)
    resp, err := http.DefaultClient.Do(req)
    // process the response...
}
```

The GUID is passed through to the logger and the request decorator without the business logic being aware of it, separating the data needed for program logic from the data needed for program management. The only place that's aware of the association is the code in `main` that wires up your dependencies:

```
controller := Controller{
    Logic: LogicImpl{
        RequestDecorator: tracker.Request,
        Logger:           tracker.Logger{},
        Remote:           "http://localhost:4000",
    },
}
```

You can find the complete code for the GUID tracker in the *sample_code/context_guid* directory in the Chapter 14 repository.

Use the context to pass values through standard APIs. Copy values from the context into explicit parameters when they are needed for processing business logic. System maintenance information can be accessed directly from the context.

Cancellation

While context values are useful for passing metadata and working around the limitations of Go's HTTP API, the context has a second use. The context also allows you to control the responsiveness of your application and coordinate concurrent goroutines. Let's see how.

I discussed this briefly in "Use the Context to Terminate Goroutines" on page 300. Imagine that you have a request that launches several goroutines, each one calling a different HTTP service. If one service returns an error that prevents you from returning a valid result, there is no point in continuing to process the other goroutines. In Go, this is called *cancellation*, and the context provides the mechanism for its implementation.

To create a cancellable context, use the `context.WithCancel` function. It takes in a `context.Context` as a parameter and returns a `context.Context` and a `context.CancelFunc`. Just like `context.WithValue`, the returned `context.Context` is a child context of the context that was passed into the function. A `context.CancelFunc` is a function that takes no parameters and *cancels* the context, telling all the code that's listening for potential cancellation that it's time to stop processing.

Anytime you create a context that has an associated cancel function, you *must* call that cancel function when you are done processing, whether or not your processing ends in an error. If you do not, your program will leak resources (memory and goroutines) and eventually slow down or crash. No error occurs if you call the cancel function more than once; any invocation after the first does nothing.

The easiest way to make sure you call the cancel function is to use `defer` to invoke it right after the cancel function is returned:

```go
ctx, cancelFunc := context.WithCancel(context.Background())
defer cancelFunc()
```

This leads to the question, how do you detect cancellation? The `context.Context` interface has a method called `Done`. It returns a channel of type `struct{}`. (The reason this is the chosen return type is that an empty struct uses no memory.) This channel is closed when the `cancel` function is invoked. Remember, a closed channel always immediately returns its zero value when you attempt to read it.

If you call `Done` on a context that isn't cancellable, it returns `nil`. As was covered in "Turn Off a case in a select" on page 304, a read from a `nil` channel never returns. If this is not done inside a `case` in a `select` statement, your program will hang.

Let's take a look at how this works. Let's say you have a program that's gathering data from a number of HTTP endpoints. If any one of them fails, you want to end processing across all of them. Context cancellation allows you to do that.

In this example, you are going to take advantage of a great service called httpbin.org. You send it HTTP or HTTPS requests to test how your application responds to a variety of situations. You'll use two of its endpoints: one that delays response for a specified number of seconds, and one that will return one of the status codes that you send it.

First, create your cancellable context, a channel to get back data from your goroutines, and a `sync.WaitGroup` to allow you to wait until all goroutines have completed:

```go
ctx, cancelFunc := context.WithCancel(context.Background())
defer cancelFunc()
ch := make(chan string)
var wg sync.WaitGroup
wg.Add(2)
```

Next, you launch two goroutines, one that calls a URL that randomly returns a bad status, and the other that sends a canned JSON response after a delay. First the random status goroutine:

```
go func() {
    defer wg.Done()
    for {
        // return one of these status code at random
        resp, err := makeRequest(ctx,
            "http://httpbin.org/status/200,200,200,500")
        if err != nil {
            fmt.Println("error in status goroutine:", err)
            cancelFunc()
            return
        }
        if resp.StatusCode == http.StatusInternalServerError {
            fmt.Println("bad status, exiting")
            cancelFunc()
            return
        }
        select {
        case ch <- "success from status":
        case <-ctx.Done():
        }
        time.Sleep(1 * time.Second)
    }
}()
```

The `makeRequest` function is a helper to make an HTTP request using the supplied context and URL. If you get back an OK status, you write a message to the channel and sleep for a second. When an error occurs or you get back a bad status code, you call `cancelFunc` and exit the goroutine.

The delay goroutine is similar:

```
go func() {
    defer wg.Done()
    for {
        // return after a 1 second delay
        resp, err := makeRequest(ctx, "http://httpbin.org/delay/1")
        if err != nil {
            fmt.Println("error in delay goroutine:", err)
            cancelFunc()
            return
        }
        select {
        case ch <- "success from delay: " + resp.Header.Get("date"):
        case <-ctx.Done():
        }
    }
}()
```

Finally, you use the for/select pattern to read data from the channel written to by the goroutines and wait for cancellation to happen:

```
loop:
    for {
        select {
        case s := <-ch:
            fmt.Println("in main:", s)
        case <-ctx.Done():
            fmt.Println("in main: cancelled!")
            break loop
        }
    }
    wg.Wait()
```

In your `select` statement, you have two cases. One reads from the message channel, and the other waits for the done channel to be closed. When it closes, you exit the loop and wait for the goroutines to exit. You can find this program in the *sample_code/cancel_http* directory in the Chapter 14 repository.

Here's what happens when you run the code (the results are random, so go ahead and run it a few times to see different results):

```
in main: success from status
in main: success from delay: Thu, 16 Feb 2023 03:53:57 GMT
in main: success from status
in main: success from delay: Thu, 16 Feb 2023 03:53:58 GMT
bad status, exiting
in main: cancelled!
error in delay goroutine: Get "http://httpbin.org/delay/1": context canceled
```

There are some interesting things to note. First, you are calling `cancelFunc` multiple times. As mentioned earlier, this is perfectly fine and causes no problems. Next, notice that you got an error from the delay goroutine after cancellation was triggered. This is because the built-in HTTP client in the Go standard library respects cancellation. You created the request using the cancellable context, and when it was cancelled, the request ended. This triggers the error path in the goroutine and makes sure that it does not leak.

You might wonder about the error that caused the cancellation and how you can report it. An alternate version of `WithCancel`, called `WithCancelCause`, returns a cancellation function that takes in an error as a parameter. The `Cause` function in the `context` package returns the error that was passed into the first invocation of the cancellation function.

Cause is a function in the context package instead of a method on context.Context because the ability to return errors via cancellation was added to the context package in Go 1.20, long after the context was initially introduced. If a new method was added to the context.Context interface, this would have broken any third-party code that implemented it. Another option would be to define a new interface that included this method, but existing code already passes context.Context everywhere, and converting to a new interface with a Cause method would require type assertions or type switches. Adding a function is the simplest approach. There are several ways to evolve your APIs over time. You should pick the one that has the least impact on your users.

Let's rewrite the program to capture the error. First, you change your context creation:

```go
ctx, cancelFunc := context.WithCancelCause(context.Background())
defer cancelFunc(nil)
```

Next, you are going to make slight modifications to your two goroutines. The body of the for loop in the status goroutine now looks like this:

```go
resp, err := makeRequest(ctx, "http://httpbin.org/status/200,200,200,500")
if err != nil {
    cancelFunc(fmt.Errorf("in status goroutine: %w", err))
    return
}
if resp.StatusCode == http.StatusInternalServerError {
    cancelFunc(errors.New("bad status"))
    return
}
ch <- "success from status"
time.Sleep(1 * time.Second)
```

You've removed the fmt.Println statements and pass errors to cancelFunc. The body of the for loop in the delay goroutine now looks like this:

```go
resp, err := makeRequest(ctx, "http://httpbin.org/delay/1")
if err != nil {
    fmt.Println("in delay goroutine:", err)
    cancelFunc(fmt.Errorf("in delay goroutine: %w", err))
    return
}
ch <- "success from delay: " + resp.Header.Get("date")
```

The fmt.Println is still there so you can show that the error is still generated and passed to cancelFunc.

Finally, you use `context.Cause` to print out the error both on initial cancellation and after you finish waiting for your goroutines to complete:

```
loop:
    for {
        select {
        case s := <-ch:
            fmt.Println("in main:", s)
        case <-ctx.Done():
            fmt.Println("in main: cancelled with error", context.Cause(ctx))
            break loop
        }
    }
    wg.Wait()
    fmt.Println("context cause:", context.Cause(ctx))
```

You can find this code in the *sample_code/cancel_error_http* directory in the Chapter 14 repository.

Running your new program produces this output:

```
in main: success from status
in main: success from delay: Thu, 16 Feb 2023 04:11:49 GMT
in main: cancelled with error bad status
in delay goroutine: Get "http://httpbin.org/delay/1": context canceled
context cause: bad status
```

You see that the error from the status goroutine is printed out both when cancellation is initially detected in the `switch` statement and after you finish waiting for the delay goroutine to complete. Notice that the delay goroutine called `cancelFunc` with an error, but that error doesn't overwrite the initial cancellation error.

Manual cancellation is very useful when your code reaches a logical state that ends processing. Sometimes you want to cancel because a task is taking too long. In that case, you use a timer.

Contexts with Deadlines

One of the most important jobs for a server is managing requests. A novice programmer often thinks that a server should take as many requests as it possibly can and work on them for as long as it can until it returns a result for each client.

The problem is that this approach does not scale. A server is a shared resource. Like all shared resources, each user wants to get as much as they can out of it and isn't terribly concerned with the needs of other users. It's the responsibility of the shared resource to manage itself so that it provides a fair amount of time to all its users.

Generally, a server can do four things to manage its load:

- Limit simultaneous requests
- Limit the number of queued requests waiting to run
- Limit the amount of time a request can run
- Limit the resources a request can use (such as memory or disk space)

Go provides tools to handle the first three. You saw how to handle the first two when learning about concurrency in Chapter 12. By limiting the number of goroutines, a server manages simultaneous load. The size of the waiting queue is handled via buffered channels.

The context provides a way to control how long a request runs. When building an application, you should have an idea of your performance envelope: how long you have for your request to complete before the user has an unsatisfactory experience. If you know the maximum amount of time that a request can run, you can enforce it using the context.

> While `GOMEMLIMIT` provides a soft way to limit the amount of memory a Go program uses, if you want to enforce constraints on memory or disk space that a single request uses, you'll have to write the code to manage that yourself. Discussion of this topic is beyond the scope of this book.

You can use one of two functions to create a time-limited context. The first is `context.WithTimeout`. It takes two parameters: an existing context and `time.Duration` that specifies the duration until the context automatically cancels. It returns a context that automatically triggers a cancellation after the specified duration as well as a cancellation function that is invoked to cancel the context immediately.

The second function is `context.WithDeadline`. This function takes in an existing context and a `time.Time` that specifies the time when the context is automatically canceled. Like `context.WithTimeout`, it returns a context that automatically triggers a cancellation after the specified time has elapsed as well as a cancellation function.

> If you pass a time in the past to `context.WithDeadline`, the context is created already canceled.

Just as with the cancellation function returned from `context.WithCancel` and `context.WithCancelCause`, you must make sure that that the cancellation function returned by context.WithTimeout and context.WithDeadline is invoked at least once.

If you want to find out when a context will automatically cancel, use the Deadline method on context.Context. It returns a time.Time that indicates the time and a bool that indicates if a timeout was set. This mirrors the comma ok idiom you use when reading from maps or channels.

When you set a time limit for the overall duration of the request, you might want to subdivide that time. And if you call another service from your service, you might want to limit how long you allow the network call to run, reserving some time for the rest of your processing or for other network calls. You control how long an individual call takes by creating a child context that wraps a parent context by using context.WithTimeout or context.WithDeadline.

Any timeout that you set on the child context is bounded by the timeout set on the parent context; if a parent context times out in 2 seconds, you can declare that a child context times out in 3 seconds, but when the parent context times out after 2 seconds, so will the child.

You can see this with a simple program:

```
ctx := context.Background()
parent, cancel := context.WithTimeout(ctx, 2*time.Second)
defer cancel()
child, cancel2 := context.WithTimeout(parent, 3*time.Second)
defer cancel2()
start := time.Now()
<-child.Done()
end := time.Now()
fmt.Println(end.Sub(start).Truncate(time.Second))
```

In this sample, you specify a 2-second timeout on the parent context and a 3-second timeout on the child context. You then wait for the child context to complete by waiting on the channel returned from the Done method on the child context.Context. I'll talk more about the Done method in the next section.

You can find this code in the *sample_code/nested_timers* directory in the Chapter 14 repository or run this code on The Go Playground. You'll see the following result:

```
2s
```

Since contexts with timers can cancel because of a timeout or an explicit call to the cancellation function, the context API provides a way to tell what caused cancellation. The Err method returns nil if the context is still active, or it returns one of two sentinel errors if the context has been canceled: context.Canceled or context.DeadlineExceeded. The first is returned after explicit cancellation, and the second is returned when a timeout triggered cancellation.

Let's see them in use. You're going to make one more change to your httpbin program. This time, you're adding a timeout to the context that's used to control how long the goroutines run:

```
ctx, cancelFuncParent := context.WithTimeout(context.Background(), 3*time.Second)
defer cancelFuncParent()
ctx, cancelFunc := context.WithCancelCause(ctx)
defer cancelFunc(nil)
```

If you want the option of returning the error for the cancellation cause, you need to wrap a context created by `WithTimeout` or `WithDeadline` in a context created by `WithCancelCause`. You must `defer` both cancellation functions to keep resources from being leaked. If you want to return a custom sentinel error when a context times out, use the `context.WithTimeoutCause` or `context.WithDeadlineCause` functions instead.

Now your program will exit if a 500 status code is returned or if you don't get a 500 status code within 3 seconds. The only other change you are making to the program is to print out the value returned by `Err` when cancellation happens:

```
fmt.Println("in main: cancelled with cause:", context.Cause(ctx),
    "err:", ctx.Err())
```

You can find the code in the *sample_code/timeout_error_http* directory in the Chapter 14 repository.

The results are random, so run the program multiple times to see different results. If you run the program and hit the timeout, you'll get output like this:

```
in main: success from status
in main: success from delay: Sun, 19 Feb 2023 04:36:44 GMT
in main: success from status
in main: success from status
in main: success from delay: Sun, 19 Feb 2023 04:36:45 GMT
in main: cancelled with cause: context deadline exceeded
    err: context deadline exceeded
in delay goroutine: Get "http://httpbin.org/delay/1":
    context deadline exceeded
context cause: context deadline exceeded
```

Notice that the error returned by `context.Cause` is the same error that's returned by the `Err` method: `context.DeadlineExceeded`.

If the status error happens within 3 seconds, you'll get output like this:

```
in main: success from status
in main: success from status
in main: success from delay: Sun, 19 Feb 2023 04:37:14 GMT
in main: cancelled with cause: bad status err: context canceled
in delay goroutine: Get "http://httpbin.org/delay/1": context canceled
context cause: bad status
```

Now the error returned by `context.Cause` is `bad status`, but `Err` returns a `context.Canceled` error.

Context Cancellation in Your Own Code

Most of the time, you don't need to worry about timeouts or cancellation within your own code; it simply doesn't run for long enough. Whenever you call another HTTP service or the database, you should pass along the context; those libraries properly handle cancellation via the context.

You should think about handling cancellation for two situations. The first is when you have a function that reads or writes channels by using a `select` statement. As shown in "Cancellation" on page 358, include a `case` that checks the channel returned by the `Done` method on the context. This allows your function to exit upon context cancellation, even if the goroutines do not handle cancellation properly.

The second situation is when you write code that runs long enough that it should be interrupted by a context cancellation. In that case, check the status of the context periodically using `context.Cause`. The `context.Cause` function returns an error if the context has been cancelled.

Here's the pattern for supporting context cancellation in your code:

```go
func longRunningComputation(ctx context.Context, data string) (string, error) {
    for {
        // do some processing
        // insert this if statement periodically
        // to check if the context has been cancelled
        if err := context.Cause(ctx); err != nil {
            // return a partial value if it makes sense,
            // or a default one if it doesn't
            return "", err
        }
        // do some more processing and loop again
    }
}
```

Here's an example loop from a function that calculates π by using the inefficient Leibniz algorithm. Using context cancellation allows you to control how long it can run:

```go
i := 0
for {
    if err := context.Cause(ctx); err != nil {
        fmt.Println("cancelled after", i, "iterations")
        return sum.Text('g', 100), err
    }
```

```
        var diff big.Float
        diff.SetInt64(4)
        diff.Quo(&diff, &d)
        if i%2 == 0 {
            sum.Add(&sum, &diff)
        } else {
            sum.Sub(&sum, &diff)
        }
        d.Add(&d, two)
        i++
    }
```

You can see the complete sample program that demonstrates this pattern in the *sample_code/own_cancellation* directory in the Chapter 14 repository.

Exercises

Now that you've seen how to use the context, try to implement these exercises. All the answers are found in the Chapter 14 repository.

1. Create a middleware-generating function that creates a context with a timeout. The function should have one parameter, which is the number of milliseconds that a request is allowed to run. It should return a `func(http.Handler) http.Handler`.

2. Write a program that adds randomly generated numbers between 0 (inclusive) and 100,000,000 (exclusive) together until one of two things happen: the number 1234 is generated or 2 seconds has passed. Print out the sum, the number of iterations, and the reason for ending (timeout or number reached).

3. Assume you have a simple logging function that looks like this:

   ```
   func Log(ctx context.Context, level Level, message string) {
       var inLevel Level
       // TODO get a logging level out of the context and assign it to inLevel
       if level == Debug && inLevel == Debug {
           fmt.Println(message)
       }
       if level == Info && (inLevel == Debug || inLevel == Info) {
           fmt.Println(message)
       }
   }
   ```

   Define a type called `Level` whose underlying type is `string`. Define two constants of this type, `Debug` and `Info`, and set them to `"debug"` and `"info"`, respectively.

   Create functions to store the log level in the context and to extract it.

   Create a middleware function to get the logging level from a query parameter called `log_level`. The valid values for `log_level` are `debug` and `info`.

Finally, fill in the `TODO` in `Log` to properly extract the log level from the context. If the log level is not assigned or is not a valid value, nothing should be printed.

Wrapping Up

In this chapter, you learned how to manage request metadata by using the context. You can now set timeouts, perform explicit cancellation, pass values through the context, and know when you should do each of these things. In the next chapter, you're going to see Go's built-in testing framework and learn how to use it to find bugs and diagnose performance problems in your programs.