CHAPTER 13

The Standard Library

One of the best parts of developing with Go is being able to take advantage of its standard library. Like Python, it has a "batteries included" philosophy, providing many of the tools that you need to build an application. Since Go is a relatively new language, it ships with a library that is focused on problems faced in modern programming environments.

I can't cover all the standard library packages, and luckily, I don't have to, as there are many excellent sources of information on the standard library, starting with the documentation. Instead, I'll focus on several of the most important packages and how their design and use demonstrate the principles of idiomatic Go. Some packages (`errors`, `sync`, `context`, `testing`, `reflect`, and `unsafe`) are covered in their own chapters. In this chapter, you'll look at Go's built-in support for I/O, time, JSON, and HTTP.

io and Friends

For a program to be useful, it needs to read in and write out data. The heart of Go's input/output philosophy can be found in the `io` package. In particular, two interfaces defined in this package are probably the second and third most-used interfaces in Go: `io.Reader` and `io.Writer`.

What's number one? That'd be `error`, which you already looked at in Chapter 9.

Both `io.Reader` and `io.Writer` define a single method:

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

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

The `Write` method on the `io.Writer` interface takes in a slice of bytes, which are written to the interface implementation. It returns the number of bytes written and an error if something went wrong. The `Read` method on `io.Reader` is more interesting. Rather than return data through a return parameter, a slice is passed into the implementation and modified. Up to `len(p)` bytes will be written into the slice. The method returns the number of bytes written. This might seem a little strange. You might expect this:

```
type NotHowReaderIsDefined interface {
    Read() (p []byte, err error)
}
```

There's a very good reason `io.Reader` is defined the way it is. Let's write a function that's representative of how to work with an `io.Reader` to illustrate:

```
func countLetters(r io.Reader) (map[string]int, error) {
    buf := make([]byte, 2048)
    out := map[string]int{}
    for {
        n, err := r.Read(buf)
        for _, b := range buf[:n] {
            if (b >= 'A' && b <= 'Z') || (b >= 'a' && b <= 'z') {
                out[string(b)]++
            }
        }
        if err == io.EOF {
            return out, nil
        }
        if err != nil {
            return nil, err
        }
    }
}
```

There are three things to note. First, you create your buffer once and reuse it on every call to `r.Read`. This allows you to use a single memory allocation to read from a potentially large data source. If the `Read` method were written to return a `[]byte`, it would require a new allocation on every single call. Each allocation would end up on the heap, which would make quite a lot of work for the garbage collector.

If you want to reduce the allocations further, you could create a pool of buffers when the program launches. You would then take a buffer out of the pool when the function starts, and return it when it ends. By passing in a slice to `io.Reader`, memory allocation is under the control of the developer.

Second, you use the `n` value returned from `r.Read` to know how many bytes were written to the buffer and iterate over a subslice of your `buf` slice, processing the data that was read.

Finally, you know that you're done reading from `r` when the error returned from `r.Read` is `io.EOF`. This error is a bit odd, in that it isn't really an error. It indicates that there's nothing left to read from the `io.Reader`. When `io.EOF` is returned, you are finished processing and return your result.

The `Read` method in `io.Reader` has one unusual aspect. In most cases when a function or method has an error return value, you check the error before you try to process the nonerror return values. You do the opposite for `Read` because bytes might have been copied into the buffer before an error was triggered by the end of the data stream or by an unexpected condition.

If you get to the end of an `io.Reader` unexpectedly, a different sentinel error is returned (`io.ErrUnexpectedEOF`). Note that it starts with the string `Err` to indicate that it is an unexpected state.

Because `io.Reader` and `io.Writer` are such simple interfaces, they can be implemented many ways. You can create an `io.Reader` from a string by using the `strings.NewReader` function:

```
s := "The quick brown fox jumped over the lazy dog"
sr := strings.NewReader(s)
counts, err := countLetters(sr)
if err != nil {
    return err
}
fmt.Println(counts)
```

As I discussed in "Interfaces Are Type-Safe Duck Typing" on page 158, implementations of `io.Reader` and `io.Writer` are often chained together in a decorator pattern. Because `countLetters` depends on an `io.Reader`, you can use the exact same `countLetters` function to count English letters in a gzip-compressed file. First you write a function that, when given a filename, returns a `*gzip.Reader`:

```
func buildGZipReader(fileName string) (*gzip.Reader, func(), error) {
    r, err := os.Open(fileName)
    if err != nil {
```

```
        return nil, nil, err
    }
    gr, err := gzip.NewReader(r)
    if err != nil {
        return nil, nil, err
    }
    return gr, func() {
        gr.Close()
        r.Close()
    }, nil
}
```

This function demonstrates the way to properly wrap types that implement `io.Reader`. You create an `*os.File` (which meets the `io.Reader` interface), and after making sure it's valid, you pass it to the `gzip.NewReader` function, which returns a `*gzip.Reader` instance. If it is valid, you return the `*gzip.Reader` and a closure that properly cleans up your resources when it is invoked.

Since `*gzip.Reader` implements `io.Reader`, you can use it with `countLetters` just as you used the `*strings.Reader` previously:

```
r, closer, err := buildGZipReader("my_data.txt.gz")
if err != nil {
    return err
}
defer closer()
counts, err := countLetters(r)
if err != nil {
    return err
}
fmt.Println(counts)
```

You can find the code for `countLetters` and `buildGZipReader` in the *sample_code/io_friends* directory in the Chapter 13 repository.

Because there are standard interfaces for reading and writing, there's a standard function in the `io` package for copying from an `io.Reader` to an `io.Writer`, `io.Copy`. There are other standard functions for adding new functionality to existing `io.Reader` and `io.Writer` instances. These include the following:

`io.MultiReader`
: Returns an `io.Reader` that reads from multiple `io.Reader` instances, one after another

`io.LimitReader`
: Returns an `io.Reader` that reads only up to a specified number of bytes from the supplied `io.Reader`

`io.MultiWriter`
: Returns an `io.Writer` that writes to multiple `io.Writer` instances at the same time

Other packages in the standard library provide their own types and functions to work with `io.Reader` and `io.Writer`. You've seen a few of them already, but there are many more. These cover compression algorithms, archives, cryptography, buffers, byte slices, and strings.

Other one-method interfaces are defined in `io`, such as `io.Closer` and `io.Seeker`:

```
type Closer interface {
        Close() error
}

type Seeker interface {
        Seek(offset int64, whence int) (int64, error)
}
```

The `io.Closer` interface is implemented by types like `os.File` that need to do cleanup when reading or writing is complete. Usually, `Close` is called via a `defer`:

```
f, err := os.Open(fileName)
if err != nil {
    return nil, err
}
defer f.Close()
// use f
```

If you are opening the resource in a loop, do not use `defer`, as it will not run until the function exits. Instead, you should call `Close` before the end of the loop iteration. If there are errors that can lead to an exit, you must call `Close` there too.

The `io.Seeker` interface is used for random access to a resource. The valid values for `whence` are the constants `io.SeekStart`, `io.SeekCurrent`, and `io.SeekEnd`. This should have been made clearer by using a custom type, but in a surprising design oversight, `whence` is of type `int`.

The `io` package defines interfaces that combine these four interfaces in various ways. They include `io.ReadCloser`, `io.ReadSeeker`, `io.ReadWriteCloser`, `io.ReadWriteSeeker`, `io.ReadWriter`, `io.WriteCloser`, and `io.WriteSeeker`. Use these interfaces to specify what your functions expect to do with the data. For example, rather than just using an `os.File` as a parameter, use the interfaces to specify exactly what your function will do with the parameter. Not only does it make your functions more general-purpose, it also makes your intent clearer. Furthermore, make your code compatible with these interfaces if you are writing your own data sources and sinks.

In general, strive to create interfaces as simple and decoupled as the interfaces defined in `io`. They demonstrate the power of simple abstractions.

In addition to the interfaces in the `io` package, there are several helper functions for common operations. For example, the `io.ReadAll` function reads all the data from an `io.Reader` into a byte slice. One of the more clever functions in `io` demonstrates a pattern for adding a method to a Go type. If you have a type that implements `io.Reader` but not `io.Closer` (such as `strings.Reader`) and need to pass it to a function that expects an `io.ReadCloser`, pass your `io.Reader` into `io.NopCloser` and get back a type that implements `io.ReadCloser`. If you look at the implementation, it's very simple:

```
type nopCloser struct {
    io.Reader
}

func (nopCloser) Close() error { return nil }

func NopCloser(r io.Reader) io.ReadCloser {
    return nopCloser{r}
}
```

Anytime you need to add additional methods to a type so that it can meet an interface, use this embedded type pattern.

The `io.NopCloser` function violates the general rule of not returning an interface from a function, but it's a simple adapter for an interface that is guaranteed to stay the same because it is part of the standard library.

Among other things, the `os` package contains functions for interacting with files. The functions `os.ReadFile` and `os.WriteFile` read an entire file into a slice of bytes and write a slice of bytes into a file, respectively. These functions (and `io.ReadAll`) are fine for small amounts of data, but they are not appropriate for large data sources. When working with larger data sources, use the `Create`, `NewFile`, `Open`, and `OpenFile` functions in the `os` package. They return an `*os.File` instance, which implements the `io.Reader` and `io.Writer` interfaces. You can use an `*os.File` instance with the `Scanner` type in the `bufio` package.

time

Like most languages, Go's standard library includes time support, which is found, unsurprisingly, in the `time` package. Two main types are used to represent time: `time.Duration` and `time.Time`.

A period of time is represented with a `time.Duration`, a type based on an `int64`. The smallest amount of time that Go can represent is one nanosecond, but the `time` package defines constants of type `time.Duration` to represent a nanosecond, microsecond, millisecond, second, minute, and hour. For example, you represent a duration of 2 hours and 30 minutes as follows:

```
d := 2 * time.Hour + 30 * time.Minute // d is of type time.Duration
```

These constants make the use of a `time.Duration` both readable and type-safe. They demonstrate a good use of a typed constant.

Go defines a sensible string format, a series of numbers, that can be parsed into a `time.Duration` with the `time.ParseDuration` function. This format is described in the standard library documentation:

> A duration string is a possibly signed sequence of decimal numbers, each with optional fraction and a unit suffix, such as "300ms", "-1.5h" or "2h45m". Valid time units are "ns", "us" (or "µs"), "ms", "s", "m", "h".
>
> —Go Standard Library Documentation

Several methods are defined on `time.Duration`. It meets the `fmt.Stringer` interface and returns a formatted duration string via the `String` method. It also has methods to get the value as a number of hours, minutes, seconds, milliseconds, microseconds, or nanoseconds. The `Truncate` and `Round` methods truncate or round a `time.Duration` to the units of the specified `time.Duration`.

A moment of time is represented with the `time.Time` type, complete with a time zone. You acquire a reference to the current time with the function `time.Now`. This returns a `time.Time` instance set to the current local time.

The fact that a `time.Time` instance contains a time zone means that you should not use `==` to check whether two `time.Time` instances refer to the same moment in time. Instead, use the `Equal` method, which corrects for time zone.

The `time.Parse` function converts from a `string` to a `time.Time`, while the `Format` method converts a `time.Time` to a `string`. While Go usually adopts ideas that worked well in the past, it uses its own date and time formatting language. It relies on the idea of formatting the date and time January 2, 2006 at 3:04:05PM MST (Mountain Standard Time) to specify your format.

Why that date? Because each part of it represents one of the numbers from 1 to 7 in sequence, that is, 01/02 03:04:05PM '06 -0700 (MST is 7 hours before UTC).

For example, the following code

```
t, err := time.Parse("2006-01-02 15:04:05 -0700", "2023-03-13 00:00:00 +0000")
if err != nil {
    return err
}
fmt.Println(t.Format("January 2, 2006 at 3:04:05PM MST"))
```

prints out this output:

```
March 13, 2023 at 12:00:00AM UTC
```

While the date and time used for formatting is intended to be a clever mnemonic, I find it hard to remember and have to look it up each time I want to use it. Luckily, the most commonly used date and time formats have been given their own constants in the `time` package.

Just as there are methods on `time.Duration` to extract portions of it, there are methods defined on `time.Time` to do the same, including `Day`, `Month`, `Year`, `Hour`, `Minute`, `Second`, `Weekday`, `Clock` (which returns the time portion of a `time.Time` as separate hour, minute, and second `int` values), and `Date` (which returns the year, month, and day as separate `int` values). You can compare one `time.Time` instance against another with the `After`, `Before`, and `Equal` methods.

The `Sub` method returns a `time.Duration` that represents the elapsed time between two `time.Time` instances, while the `Add` method returns a `time.Time` that is `time.Duration` later, and the `AddDate` method returns a new `time.Time` instance that's incremented by the specified number of years, months, and days. As with `time.Duration`, there are `Truncate` and `Round` methods defined as well. All these methods are defined on a value receiver, so they do not modify the `time.Time` instance.

Monotonic Time

Most operating systems keep track of two sorts of time: the *wall clock*, which corresponds to the current time, and the *monotonic clock*, which counts up from the time the computer was booted. The reason for tracking two clocks is that the wall clock doesn't uniformly increase. Daylight Saving Time, leap seconds, and Network Time Protocol (NTP) updates can make the wall clock move unexpectedly forward or backward. This can cause problems when setting a timer or finding the amount of time that's elapsed.

To address this potential problem, Go uses monotonic time to track elapsed time whenever a timer is set or a `time.Time` instance is created with `time.Now`. This support is invisible; timers use it automatically. The `Sub` method uses the monotonic clock to calculate the `time.Duration` if both `time.Time` instances have it set. If they don't (because one or both of the instances was not created with `time.Now`), the `Sub` method uses the time specified in the instances to calculate the `time.Duration` instead.

If you want to understand the sorts of problems that can occur when not handling monotonic time correctly, take a look at the Cloudflare blog post that detailed a bug caused by the lack of monotonic time support in an earlier version of Go.

Timers and Timeouts

As I covered in "Time Out Code" on page 304, the `time` package includes functions that return channels that output values after a specified time. The `time.After` function returns a channel that outputs once, while the channel returned by `time.Tick` returns a new value every time the specified `time.Duration` elapses. These are used with Go's concurrency support to enable timeouts or recurring tasks.

You can also trigger a single function to run after a specified `time.Duration` with the `time.AfterFunc` function. Don't use `time.Tick` outside trivial programs, because the underlying `time.Ticker` cannot be shut down (and therefore cannot be garbage collected). Use the `time.NewTicker` function instead, which returns a `*time.Ticker` that has the channel to listen to, as well as methods to reset and stop the ticker.

encoding/json

REST APIs have enshrined JSON as the standard way to communicate between services, and Go's standard library includes support for converting Go data types to and from JSON. The word *marshaling* means converting from a Go data type to an encoding, and *unmarshaling* means converting to a Go data type.

Using Struct Tags to Add Metadata

Let's say that you are building an order management system and have to read and write the following JSON:

```
{
    "id":"12345",
    "date_ordered":"2020-05-01T13:01:02Z",
    "customer_id":"3",
```

```
    "items":[{"id":"xyz123","name":"Thing 1"},{"id":"abc789","name":"Thing 2"}]
}
```

You define types to map this data:

```
type Order struct {
    ID           string        `json:"id"`
    DateOrdered  time.Time     `json:"date_ordered"`
    CustomerID   string        `json:"customer_id"`
    Items        []Item        `json:"items"`
}

type Item struct {
    ID   string `json:"id"`
    Name string `json:"name"`
}
```

You specify the rules for processing your JSON with *struct tags*, strings that are written after the fields in a struct. Even though struct tags are strings marked with backticks, they cannot extend past a single line. Struct tags are composed of one or more tag/value pairs, written as *`tagName:"tagValue"`* and separated by spaces. Because they are just strings, the compiler cannot validate that they are formatted correctly, but `go vet` does. Also, note that all these fields are exported. Like any other package, the code in the `encoding/json` package cannot access an unexported field on a struct in another package.

For JSON processing, use the tag `json` to specify the name of the JSON field that should be associated with the struct field. If no `json` tag is provided, the default behavior is to assume that the name of the JSON object field matches the name of the Go struct field. Despite this default behavior, it's best to use the struct tag to specify the name of the field explicitly, even if the field names are identical.

When unmarshaling from JSON into a struct field with no `json` tag, the name match is case-insensitive. When marshaling a struct field with no `json` tag back to JSON, the JSON field will always have an uppercase first letter, because the field is exported.

If a field should be ignored when marshaling or unmarshaling, use a dash (`-`) for the name. If the field should be left out of the output when it is empty, add `,omitempty` after the name. For example, in the `Order` struct, if you didn't want to include `CustomerID` in the output if it was set to an empty string, the struct tag would be `json:"customer_id,omitempty"`.

Unfortunately, the definition of "empty" doesn't exactly align with the zero value, as you might expect. The zero value of a struct doesn't count as empty, but a zero-length slice or map does.

Struct tags allow you to use metadata to control how your program behaves. Other languages, most notably Java, encourage developers to place annotations on various program elements to describe *how* they should be processed, without explicitly specifying *what* is going to do the processing. While declarative programming allows for more concise programs, automatic processing of metadata makes it difficult to understand how a program behaves. Anyone who has worked on a large Java project with annotations has had a moment of panic when something goes wrong and they don't understand which code is processing a particular annotation and what changes it made. Go favors explicit code over short code. Struct tags are never evaluated automatically; they are processed when a struct instance is passed into a function.

Unmarshaling and Marshaling

The `Unmarshal` function in the `encoding/json` package is used to convert a slice of bytes into a struct. If you have a string named `data`, this is the code to convert `data` to a struct of type `Order`:

```
var o Order
err := json.Unmarshal([]byte(data), &o)
if err != nil {
    return err
}
```

The `json.Unmarshal` function populates data into an input parameter, just like the implementations of the `io.Reader` interface. As I discussed in "Pointers Are a Last Resort" on page 129, this allows for efficient reuse of the same struct over and over, giving you control over memory usage.

You use the `Marshal` function in the `encoding/json` package to write an `Order` instance back as JSON, stored in a slice of bytes:

```
out, err := json.Marshal(o)
```

This leads to the question: how are you able to evaluate struct tags? You might also be wondering how `json.Marshal` and `json.Unmarshal` are able to read and write a struct of any type. After all, every other method that you've written has worked only with types that were known when the program was compiled (even the types listed in a type switch are enumerated ahead of time). The answer to both questions is reflection. You can find out more about reflection in Chapter 16.

JSON, Readers, and Writers

The `json.Marshal` and `json.Unmarshal` functions work on slices of bytes. As you just saw, most data sources and sinks in Go implement the `io.Reader` and `io.Writer` interfaces. While you could use `io.ReadAll` to copy the entire contents of an `io.Reader` into a byte slice so it can be read by `json.Unmarshal`, this is inefficient. Similarly, you could write to an in-memory byte slice buffer using `json.Marshal` and then write that byte slice to the network or disk, but it'd be better if you could write to an `io.Writer` directly.

The `encoding/json` package includes two types that allow you to handle these situations. The `json.Decoder` and `json.Encoder` types read from and write to anything that meets the `io.Reader` and `io.Writer` interfaces, respectively. Let's take a quick look at how they work.

Start with your data in `toFile`, which implements a simple struct:

```
type Person struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}
toFile := Person {
    Name: "Fred",
    Age:  40,
}
```

The `os.File` type implements both the `io.Reader` and `io.Writer` interfaces, so it can be used to demonstrate `json.Decoder` and `json.Encoder`. First, you write `toFile` to a temp file by passing the temp file to `json.NewEncoder`, which returns a `json.Encoder` for the temp file. You then pass `toFile` to the `Encode` method:

```
tmpFile, err := os.CreateTemp(os.TempDir(), "sample-")
if err != nil {
    panic(err)
}
defer os.Remove(tmpFile.Name())
err = json.NewEncoder(tmpFile).Encode(toFile)
if err != nil {
    panic(err)
}
err = tmpFile.Close()
if err != nil {
    panic(err)
}
```

Once `toFile` is written, you can read the JSON back in by passing a reference to the temp file to `json.NewDecoder` and then calling the `Decode` method on the returned `json.Decoder` with a variable of type `Person`:

```
tmpFile2, err := os.Open(tmpFile.Name())
if err != nil {
    panic(err)
}
var fromFile Person
err = json.NewDecoder(tmpFile2).Decode(&fromFile)
if err != nil {
    panic(err)
}
err = tmpFile2.Close()
if err != nil {
    panic(err)
}
fmt.Printf("%+v\n", fromFile)
```

You can see a complete example on The Go Playground or in the *sample_code/json* directory in the Chapter 13 repository.

Encoding and Decoding JSON Streams

What do you do when you have multiple JSON structs to read or write at once? Our friends `json.Decoder` and `json.Encoder` can be used for these situations too.

Assume you have the following data:

```
{"name": "Fred", "age": 40}
{"name": "Mary", "age": 21}
{"name": "Pat", "age": 30}
```

For the sake of this example, assume it's stored in a string called `streamData`, but it could be in a file or even an incoming HTTP request (you'll see how HTTP servers work in just a bit).

You're going to store this data into your `t` variable, one JSON object at a time.

Just as before, you initialize your `json.Decoder` with the data source, but this time you use a `for` loop and run until you get an error. If the error is `io.EOF`, you have successfully read all the data. If not, there was a problem with the JSON stream. This lets you read and process the data, one JSON object at a time:

```
var t struct {
    Name string `json:"name"`
    Age  int    `json:"age"`
}

dec := json.NewDecoder(strings.NewReader(streamData))
for {
    err := dec.Decode(&t)
    if err != nil {
        if errors.Is(err, io.EOF) {
            break
        }
```

```
        panic(err)
    }
    // process t
}
```

Writing out multiple values with the `json.Encoder` works just like using it to write out a single value. In this example, you are writing to a `bytes.Buffer`, but any type that meets the `io.Writer` interface will work:

```
var b bytes.Buffer
enc := json.NewEncoder(&b)
for _, input := range allInputs {
    t := process(input)
    err = enc.Encode(t)
    if err != nil {
        panic(err)
    }
}
out := b.String()
```

You can run this example on The Go Playground or find it in the *sample_code/encode_decode* directory in the Chapter 13 repository.

This example has multiple JSON objects in the data stream that aren't wrapped in an array, but you can also use the `json.Decoder` to read a single object from an array without loading the entire array into memory at once. This can greatly increase performance and reduce memory usage. An example is in the Go documentation.

Custom JSON Parsing

While the default functionality is often sufficient, sometimes you need to override it. While `time.Time` supports JSON fields in RFC 3339 format out of the box, you might have to deal with other time formats. You can handle this by creating a new type that implements two interfaces, `json.Marshaler` and `json.Unmarshaler`:

```
type RFC822ZTime struct {
    time.Time
}

func (rt RFC822ZTime) MarshalJSON() ([]byte, error) {
    out := rt.Time.Format(time.RFC822Z)
    return []byte(`"` + out + `"`), nil
}

func (rt *RFC822ZTime) UnmarshalJSON(b []byte) error {
    if string(b) == "null" {
        return nil
    }
    t, err := time.Parse(`"`+time.RFC822Z+`"`, string(b))
    if err != nil {
        return err
```

```
    }
    *rt = RFC822ZTime{t}
    return nil
}
```

You embedded a `time.Time` instance into a new struct called `RFC822ZTime` so that you still have access to the other methods on `time.Time`. As was discussed in "Pointer Receivers and Value Receivers" on page 145, the method that reads the time value is declared on a value receiver, while the method that modifies the time value is declared on a pointer receiver.

You then change the type of your `DateOrdered` field and can work with RFC 822 formatted times instead:

```
type Order struct {
    ID          string      `json:"id"`
    DateOrdered RFC822ZTime `json:"date_ordered"`
    CustomerID  string      `json:"customer_id"`
    Items       []Item      `json:"items"`
}
```

You can run this code on The Go Playground or find it in the *sample_code/custom_json* directory in the Chapter 13 repository.

This approach has a philosophical drawback: the date format of the JSON determines the types of the fields in your data structure. This is a drawback to the `encoding/json` approach. You could have `Order` implement `json.Marshaler` and `json.Unmarshaler`, but that requires you to write code to handle all the fields, even the ones that don't require custom support. The struct tag format does not provide a way to specify a function to parse a particular field. That leaves you with creating a custom type for the field.

Another option is described in a blog post by Ukiah Smith. It allows you to redefine only the fields that don't match the default marshaling behavior by taking advantage of how struct embedding (which was covered in "Use Embedding for Composition" on page 154) interacts with JSON marshaling and unmarshaling. If a field on an embedded struct has the same name as the containing struct, that field is ignored when marshaling or unmarshaling JSON.

In this example, the fields for `Order` look like this:

```
type Order struct {
    ID          string    `json:"id"`
    Items       []Item    `json:"items"`
    DateOrdered time.Time `json:"date_ordered"`
    CustomerID  string    `json:"customer_id"`
}
```

The `MarshalJSON` method looks like this:

```
func (o Order) MarshalJSON() ([]byte, error) {
    type Dup Order

    tmp := struct {
        DateOrdered string `json:"date_ordered"`
        Dup
    }{
        Dup: (Dup)(o),
    }
    tmp.DateOrdered = o.DateOrdered.Format(time.RFC822Z)
    b, err := json.Marshal(tmp)
    return b, err
}
```

In the `MarshalJSON` method for `Order`, you define a type `Dup` whose underlying type is `Order`. The reason for creating `Dup` is that a type based on another type has the same fields as the underlying type, but not the methods. If you didn't have `Dup`, there would be an infinite loop of calls to `MarshalJSON` when you call `json.Marshal`, eventually resulting in a stack overflow.

You define an anonymous struct that has the `DateOrdered` field and an embedded `Dup`. You then assign the `Order` instance to the embedded field in `tmp`, assign the `DateOrdered` field in `tmp` the time formatted as RFC822Z, and call `json.Marshal` on `tmp`. This produces the desired JSON output.

There is similar logic in `UnmarshalJSON`:

```
func (o *Order) UnmarshalJSON(b []byte) error {
    type Dup Order

    tmp := struct {
        DateOrdered string `json:"date_ordered"`
        *Dup
    }{
        Dup: (*Dup)(o),
    }

    err := json.Unmarshal(b, &tmp)
    if err != nil {
        return err
    }

    o.DateOrdered, err = time.Parse(time.RFC822Z, tmp.DateOrdered)
    if err != nil {
        return err
    }
    return nil
}
```

In `UnmarshalJSON`, the call to `json.Unmarshal` populates the fields in `o` (except `DateOrdered`) because it's embedded into `tmp`. You then populate `DateOrdered` in `o` by using `time.Parse` to process the `DateOrdered` field in `tmp`.

You can run this code on The Go Playground or find it in the *sample_code/custom_json2* directory in the Chapter 13 repository.

While this does keep `Order` from having a field tied to the JSON format, the `MarshalJSON` and `UnmarshalJSON` methods on `Order` are coupled to the format of the time field in the JSON. You cannot reuse `Order` to support JSON that has the time formatted another way.

To limit the amount of code that cares about what your JSON looks like, define two structs. Use one for converting to and from JSON and the other for data processing. Read in JSON to your JSON-aware type, and then copy it to the other. When you want to write out JSON, do the reverse. This does create some duplication, but it keeps your business logic from depending on wire protocols.

You can pass a `map[string]any` to `json.Marshal` and `json.Unmarshal` to translate back and forth between JSON and Go, but save that for the exploratory phase of your coding and replace it with a concrete type when you understand what you are processing. Go uses types for a reason; they document the expected data and the types of the expected data.

While JSON is probably the most commonly used encoder in the standard library, Go ships with others, including XML and Base64. If you have a data format that you want to encode and you can't find support for it in the standard library or a third-party module, you can write one yourself. You'll learn how to implement our own encoder in "Use Reflection to Write a Data Marshaler" on page 417.

The standard library includes `encoding/gob`, which is a Go-specific binary representation that is a bit like serialization in Java. Just as Java serialization is the wire protocol for Enterprise Java Beans and Java RMI, the gob protocol is intended as the wire format for a Go-specific RPC (remote procedure call) implementation in the `net/rpc` package. Don't use either `encoding/gob` or `net/rpc`. If you want to do remote method invocation with Go, use a standard protocol like GRPC so that you aren't tied to a specific language. No matter how much you love Go, if you want your services to be useful, make them callable by developers using other languages.

net/http

Every language ships with a standard library, but the expectations of what a standard library should include have changed over time. As a language launched in the 2010s, Go's standard library includes something that other language distributions had considered the responsibility of a third party: a production-quality HTTP/2 client and server.

The Client

The `net/http` package defines a `Client` type to make HTTP requests and receive HTTP responses. A default client instance (cleverly named `DefaultClient`) is found in the `net/http` package, but you should avoid using it in production applications, because it defaults to having no timeout. Instead, instantiate your own. You need to create only a single `http.Client` for your entire program, as it properly handles multiple simultaneous requests across goroutines:

```
client := &http.Client{
    Timeout: 30 * time.Second,
}
```

When you want to make a request, you create a new `*http.Request` instance with the `http.NewRequestWithContext` function, passing it a context, the method, and URL that you are connecting to. If you are making a `PUT`, `POST`, or `PATCH` request, specify the body of the request with the last parameter as an `io.Reader`. If there is no body, use `nil`:

```
req, err := http.NewRequestWithContext(context.Background(),
    http.MethodGet, "https://jsonplaceholder.typicode.com/todos/1", nil)
if err != nil {
    panic(err)
}
```

I'll talk about what a context is in Chapter 14.

Once you have an `*http.Request` instance, you can set any headers via the `Headers` field of the instance. Call the `Do` method on the `http.Client` with your `http.Request`, and the result is returned in an `http.Response`:

```
req.Header.Add("X-My-Client", "Learning Go")
res, err := client.Do(req)
if err != nil {
```

```
    panic(err)
}
```

The response has several fields with information on the request. The numeric code of the response status is in the `StatusCode` field, the text of the response code is in the `Status` field, the response headers are in the `Header` field, and any returned content is in a `Body` field of type `io.ReadCloser`. This allows you to use it with `json.Decoder` to process REST API responses:

```
defer res.Body.Close()
if res.StatusCode != http.StatusOK {
    panic(fmt.Sprintf("unexpected status: got %v", res.Status))
}
fmt.Println(res.Header.Get("Content-Type"))
var data struct {
    UserID    int    `json:"userId"`
    ID        int    `json:"id"`
    Title     string `json:"title"`
    Completed bool   `json:"completed"`
}
err = json.NewDecoder(res.Body).Decode(&data)
if err != nil {
    panic(err)
}
fmt.Printf("%+v\n", data)
```

You can find this code in the *sample_code/client* directory in the Chapter 13 repository.

There are functions in the `net/http` package to make `GET`, `HEAD`, and `POST` calls. Avoid using these functions because they use the default client, which means they don't set a request timeout.

The Server

The HTTP Server is built around the concept of an `http.Server` and the `http.Handler` interface. Just as the `http.Client` sends HTTP requests, the `http.Server` is responsible for listening for HTTP requests. It is a performant HTTP/2 server that supports TLS.

A request to a server is handled by an implementation of the `http.Handler` interface that's assigned to the `Handler` field. This interface defines a single method:

```
type Handler interface {
    ServeHTTP(http.ResponseWriter, *http.Request)
}
```

The `*http.Request` should look familiar, as it's the exact same type that's used to send a request to an HTTP server. The `http.ResponseWriter` is an interface with three methods:

```
type ResponseWriter interface {
        Header() http.Header
        Write([]byte) (int, error)
        WriteHeader(statusCode int)
}
```

These methods must be called in a specific order. First, call `Header` to get an instance of `http.Header` and set any response headers you need. If you don't need to set any headers, you don't need to call it. Next, call `WriteHeader` with the HTTP status code for your response. (All the status codes are defined as constants in the `net/http` package. This would have been a good place to define a custom type, but that was not done; all status code constants are untyped integers.) If you are sending a response that has a 200 status code, you can skip `WriteHeader`. Finally, call the `Write` method to set the body for the response. Here's what a trivial handler looks like:

```
type HelloHandler struct{}

func (hh HelloHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
}
```

You instantiate a new `http.Server` just like any other struct:

```
s := http.Server{
    Addr:         ":8080",
    ReadTimeout:  30 * time.Second,
    WriteTimeout: 90 * time.Second,
    IdleTimeout:  120 * time.Second,
    Handler:      HelloHandler{},
}
err := s.ListenAndServe()
if err != nil {
    if err != http.ErrServerClosed {
        panic(err)
    }
}
```

The `Addr` field specifies the host and port the server listens on. If you don't specify them, your server defaults to listening on all hosts on the standard HTTP port, 80. You specify timeouts for the server's reads, writes, and idles by using `time.Duration` values. Be sure to set these to properly handle malicious or broken HTTP clients, as the default behavior is to not time out at all. Finally, you specify the `http.Handler` for your server with the `Handler` field.

You can find this code in the *sample_code/server* directory in the Chapter 13 repository.

A server that handles only a single request isn't terribly useful, so the Go standard library includes a request router, `*http.ServeMux`. You create an instance with the `http.NewServeMux` function. It meets the `http.Handler` interface, so it can be assigned to the `Handler` field in `http.Server`. It also includes two methods that allow it to dispatch requests. The first method is called `Handle` and takes in two parameters, a path and an `http.Handler`. If the path matches, the `http.Handler` is invoked.

While you could create implementations of `http.Handler`, the more common pattern is to use the `HandleFunc` method on `*http.ServeMux`:

```
mux.HandleFunc("/hello", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
})
```

This method takes in a function or closure and converts it to an `http.HandlerFunc`. You explored the `http.HandlerFunc` type in "Function Types Are a Bridge to Interfaces" on page 173. For simple handlers, a closure is sufficient. For more complicated handlers that depend on other business logic, use a method on a struct, as demonstrated in "Implicit Interfaces Make Dependency Injection Easier" on page 174.

Go 1.22 extends the path syntax to optionally allow HTTP verbs and path wildcard variables. The value of a wildcard variable is read using the `PathValue` method on `http.Request`:

```
mux.HandleFunc("GET /hello/{name}", func(w http.ResponseWriter,
                                         r *http.Request) {
    name := r.PathValue("name")
    w.Write([]byte(fmt.Sprintf("Hello, %s!\n", name)))
})
```

The package-level functions `http.Handle`, `http.HandleFunc`, `http.ListenAndServe`, and `http.ListenAndServeTLS` that work with a package-level instance of the `*http.ServeMux` called `http.DefaultServeMux`. Don't use them outside trivial test programs. The `http.Server` instance is created in the `http.ListenAndServe` and `http.ListenAndServeTLS` functions, so you are unable to configure server properties like timeouts. Furthermore, third-party libraries could have registered their own handlers with the `http.DefaultServeMux`, and there's no way to know without scanning through all your dependencies (both direct and indirect). Keep your application under control by avoiding shared state.

Because `*http.ServeMux` dispatches requests to `http.Handler` instances, and since `*http.ServeMux` implements `http.Handler`, you can create an `*http.ServeMux` instance with multiple related requests and register it with a parent `*http.ServeMux`:

```
person := http.NewServeMux()
person.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("greetings!\n"))
})
dog := http.NewServeMux()
dog.HandleFunc("/greet", func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("good puppy!\n"))
})
mux := http.NewServeMux()
mux.Handle("/person/", http.StripPrefix("/person", person))
mux.Handle("/dog/", http.StripPrefix("/dog", dog))
```

In this example, a request for `/person/greet` is handled by handlers attached to `person`, while `/dog/greet` is handled by handlers attached to `dog`. When you register `person` and `dog` with `mux`, you use the `http.StripPrefix` helper function to remove the part of the path that's already been processed by `mux`. You can find this code in the *sample_code/server_mux* directory in the Chapter 13 repository.

Middleware

One of the most common requirements of an HTTP server is to perform a set of actions across multiple handlers, such as checking whether a user is logged in, timing a request, or checking a request header. Go handles these cross-cutting concerns with the *middleware pattern*.

Rather than using a special type, the middleware pattern uses a function that takes in an `http.Handler` instance and returns an `http.Handler`. Usually, the returned `http.Handler` is a closure that is converted to an `http.HandlerFunc`. Here are two middleware generators, one that provides timing of requests and another that uses perhaps the worst access controls imaginable:

```
func RequestTimer(h http.Handler) http.Handler {
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        start := time.Now()
        h.ServeHTTP(w, r)
        dur := time.Since(start)
        slog.Info("request time",
            "path", r.URL.Path,
            "duration", dur)
    })
}

var securityMsg = []byte("You didn't give the secret password\n")

func TerribleSecurityProvider(password string) func(http.Handler) http.Handler {
    return func(h http.Handler) http.Handler {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if r.Header.Get("X-Secret-Password") != password {
                w.WriteHeader(http.StatusUnauthorized)
                w.Write(securityMsg)
```

```
                return
            }
            h.ServeHTTP(w, r)
        })
    }
}
```

These two middleware implementations demonstrate what middleware does. First, you do setup operations or checks. If the checks don't pass, you write the output in the middleware (usually with an error code) and return. If all is well, you call the handler's `ServeHTTP` method. When that returns, you run cleanup operations.

The `TerribleSecurityProvider` shows how to create configurable middleware. You pass in the configuration information (in this case, the password), and the function returns middleware that uses that configuration information. It is a bit of a mind bender, as it returns a closure that returns a closure.

You might be wondering how to pass values through the layers of middleware. This is done via the context, which you'll look at in Chapter 14.

You add middleware to your request handlers by chaining them:

```
terribleSecurity := TerribleSecurityProvider("GOPHER")

mux.Handle("/hello", terribleSecurity(RequestTimer(
    http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("Hello!\n"))
    }))))
```

We get back your middleware from `TerribleSecurityProvider` and then wrap your handler in a series of function calls. This calls the `terribleSecurity` closure first, then calls the `RequestTimer`, which then calls your actual request handler.

Because `*http.ServeMux` implements the `http.Handler` interface, you can apply a set of middleware to all the handlers registered with a single request router:

```
terribleSecurity := TerribleSecurityProvider("GOPHER")
wrappedMux := terribleSecurity(RequestTimer(mux))
s := http.Server{
    Addr:    ":8080",
    Handler: wrappedMux,
}
```

You can find this code in the *sample_code/middleware* directory in the Chapter 13 repository.

Use third-party modules to enhance the server

Just because the server is production quality doesn't mean that you shouldn't use third-party modules to improve its functionality. If you don't like the function chains for middleware, you can use a third-party module called `alice`, which allows you to use the following syntax:

```
helloHandler := func(w http.ResponseWriter, r *http.Request) {
    w.Write([]byte("Hello!\n"))
}
chain := alice.New(terribleSecurity, RequestTimer).ThenFunc(helloHandler)
mux.Handle("/hello", chain)
```

While `*http.ServeMux` gained some much-requested features in Go 1.22, its routing and variable support are still basic. Nesting `*http.ServeMux` instances is also a bit clunky. If you find yourself needing more advanced features, such as routing based on a header value, specifying a path variable using a regular expression, or better handler nesting, many third-party request routers are available. Two of the most popular ones are gorilla mux and chi. Both are considered idiomatic because they work with `http.Handler` and `http.HandlerFunc` instances, demonstrating the Go philosophy of using composable libraries that fit together with the standard library. They also work with idiomatic middleware, and both projects provide optional middleware implementations of common concerns.

Several popular web frameworks also implement their own handler and middleware patterns. Two of the most popular are Echo and Gin. They simplify web development by incorporating features like automating the binding of data in requests or responses to JSON. They also provide adapter functions that enable you to use `http.Handler` implementations, providing a migration path.

ResponseController

In "Accept Interfaces, Return Structs" on page 162, you learned that modifying interfaces breaks backward compatibility. You also learned that you can evolve an interface over time by defining new interfaces and using type switches and type assertions to see whether the new interfaces are implemented. The drawback to creating these additional interfaces is that it is difficult to know that they exist, and using type switches to check for them is verbose.

You can find an example of this in the `http` package. When the package was designed, the choice was made to make `http.ResponseWriter` an interface. This meant that additional methods could not be added to it in future releases, or the Go Compatibility Guarantee would be broken. To represent new optional functionality for an `http.ResponseWriter` instance, the `http` package contains a couple of interfaces that may be implemented by `http.ResponseWriter` implementations: `http.Flusher` and

`http.Hijacker`. The methods on these interfaces are used to control the output for a response.

In Go 1.20, a new concrete type was added to the `http` package, `http.ResponseController`. It demonstrates another way to expose methods that have been added to an existing API:

```
func handler(rw http.ResponseWriter, req *http.Request) {
    rc := http.NewResponseController(rw)
    for i := 0; i < 10; i++ {
        result := doStuff(i)
        _, err := rw.Write([]byte(result))
        if err != nil {
            slog.Error("error writing", "msg", err)
            return
        }
        err = rc.Flush()
        if err != nil && !errors.Is(err, http.ErrNotSupported) {
            slog.Error("error flushing", "msg", err)
            return
        }
    }
}
```

In this example, you want to return data to the client as it is calculated, if the `http.ResponseWriter` supports `Flush`. If not, you will return all the data after all of it has been calculated. The factory function `http.NewResponseController` takes in an `http.ResponseWriter` and returns a pointer to an `http.ResponseController`. This concrete type has methods for the optional functionality of an `http.ResponseWriter`. You check whether the optional method is implemented by the underlying `http.ResponseWriter` by comparing the returned error to `http.ErrNotSupported` using `errors.Is`. You can find this code in the *sample_code/response_controller* directory in the Chapter 13 repository.

Because `http.ResponseController` is a concrete type that wraps access to an `http.ResponseWriter` implementation, new methods can be added to it over time without breaking existing implementations. This makes new functionality discoverable and provides a way to check for the presence or absence of an optional method with a standard error check. This pattern is an interesting way to handle situations where an interface needs to evolve. In fact, `http.ResponseController` contains two methods that don't have corresponding interfaces: `SetReadDeadline` and `SetWriteDeadline`. Future optional methods on `http.ResponseWriter` will likely be added via this technique.

Structured Logging

Since its initial release, the Go standard library has included a simple logging package, `log`. While it's fine for small programs, it doesn't easily produce *structured* logs. Modern web services can have millions of simultaneous users, and at that scale you need software to process log output in order to understand what's going on. A structured log uses a documented format for each log entry, making it easier to write programs that process log output and discover patterns and anomalies.

JSON is commonly used for structured logs, but even whitespace separated key-value pairs are easier to process than unstructured logs that don't separate values into fields. While you certainly could write JSON by using the `log` package, it doesn't provide any support for simplifying structured log creation. The `log/slog` package resolves this problem.

Adding `log/slog` to the standard library demonstrated several good Go library design practices. The first good decision was to include structured logging in the standard library. Having a standard structured logger makes it easier to write modules that work together. Several third-party structured loggers have been released to address the shortcomings of `log`, including zap, logrus, go-kit log, and many others. The problem with a fragmented logging ecosystem is that you want control over where log output goes and what level of messages are logged. If your code depends on third-party modules that use different loggers, this becomes impossible. The usual advice to prevent logging fragmentation is to not log in a module intended as a library, but that's impossible to enforce and makes it harder to monitor what's going on in a third-party library. The `log/slog` package was new in Go 1.21, but the fact that it solves these inconsistencies makes it likely that within a few years, it will be used in the vast majority of Go programs.

The second good decision was to make structured logging its own package and not part of the `log` package. While both packages have similar purposes, they have very different design philosophies. Trying to add structured logging into an unstructured logging package would confuse the API. By making them separate packages, you know at a glance that `slog.Info` is a structured log and `log.Print` is unstructured, even if you don't remember whether `Info` is for structured or unstructured logging.

The next good decision was to make the `log/slog` API scalable. It starts simply, with a default logger available via functions:

```
func main() {
    slog.Debug("debug log message")
    slog.Info("info log message")
    slog.Warn("warning log message")
    slog.Error("error log message")
}
```

These functions allow you to log simple messages at various logging levels. The output looks like the following:

```
2023/04/20 23:13:31 INFO info log message
2023/04/20 23:13:31 WARN warning log message
2023/04/20 23:13:31 ERROR error log message
```

There are two things to notice. First, the default logger suppresses debug messages by default. You'll see how to control the logging level when I discuss creating your own logger in a bit.

The second point is a bit more subtle. While this is plain-text output, it uses white-space to make a structured log. The first column is the date in year/month/day format. The second column is the time in 24-hour time. The third column is the logging level. Finally, there is the message.

The power of structured logging comes from the ability to add fields with custom values. Update your logs with some custom fields:

```
userID := "fred"
loginCount := 20
slog.Info("user login",
    "id", userID,
    "login_count", loginCount)
```

You use the same function as before, but now you add optional arguments. Optional arguments come in pairs. The first part is the key, which should be a string. The second part is the value. This log line prints out the following:

```
2023/04/20 23:36:38 INFO user login id=fred login_count=20
```

After the message, you have key-value pairs, again space separated.

While this text format is far easier to parse than an unstructured log, you might want to use something like JSON instead. You also might want to customize where the log is written or the logging level. To do that, you create a structured logging instance:

```
options := &slog.HandlerOptions{Level: slog.LevelDebug}
handler := slog.NewJSONHandler(os.Stderr, options)
mySlog := slog.New(handler)
lastLogin := time.Date(2023, 01, 01, 11, 50, 00, 00, time.UTC)
mySlog.Debug("debug message",
    "id", userID,
    "last_login", lastLogin)
```

You are using the `slog.HandlerOptions` struct to define the minimum logging level for the new logger. You then use the `NewJSONHandler` method on `slog.HandlerOptions` to create a `slog.Handler` that writes logs using JSON to the specified `io.Writer`. In this case, you are using the standard error output. Finally, you use the `slog.New` function to create a `*slog.Logger` that wraps the

`slog.Handler`. You then create a `lastLogin` value to log, along with a user ID. This gives the following output:

```
{"time":"2023-04-22T23:30:01.170243-04:00","level":"DEBUG",
 "msg":"debug message","id":"fred","last_login":"2023-01-01T11:50:00Z"}
```

If JSON and text aren't sufficient for your output needs, you can define your own implementation of the `slog.Handler` interface and pass it to `slog.New`.

Finally, the `log/slog` package takes performance into consideration. If you aren't careful, your program might end up spending more time writing logs than doing the work it was designed to perform. You can choose to write data out to `log/slog` in a number of ways. You have already seen the simplest (but slowest) method, using alternating keys and values on the `Debug`, `Info`, `Warn`, and `Error` methods. For improved performance with fewer allocations, use the `LogAttrs` method instead:

```
mySlog.LogAttrs(ctx, slog.LevelInfo, "faster logging",
                slog.String("id", userID),
                slog.Time("last_login", lastLogin))
```

The first parameter is a `context.Context`, next comes the logging level, and then zero or more `slog.Attr` instances. There are factory functions for the most commonly used types, and you can use `slog.Any` for the ones that don't have functions already supplied.

Because of the Go Compatibility Promise, the `log` package isn't going away. Existing programs that use it will continue to work, as will programs that use third-party structured loggers. If you have code that uses a `log.Logger`, the `slog.NewLogLogger` function provides a bridge to the original `log` package. It creates a `log.Logger` instance that uses a `slog.Handler` to write its output:

```
myLog := slog.NewLogLogger(mySlog.Handler(), slog.LevelDebug)
myLog.Println("using the mySlog Handler")
```

This produces the following output:

```
{"time":"2023-04-22T23:30:01.170269-04:00","level":"DEBUG",
 "msg":"using the mySlog Handler"}
```

You can find all the coding examples for `log/slog` in the *sample_code/structured_logging* directory in the Chapter 13 repository.

The `log/slog` API includes more features, including dynamic logging-level support, context support (the context is covered in Chapter 14), grouping values, and creating a common header of values. You can learn more by looking at its API documentation. Most importantly, look at how `log/slog` was put together so you can learn how to construct APIs of your own.

Exercises

Now that you've learned more about the standard library, work through these exercises to reinforce what you've learned. Solutions are in the *exercise_solutions* directory of the Chapter 13 repository.

1. Write a small web server that returns the current time in RFC 3339 format when you send it a `GET` command. You can use a third-party module if you'd like.
2. Write a small middleware component that uses JSON structured logging to log the IP address of each incoming request to your web server.
3. Add the ability to return the time as JSON. Use the `Accept` header to control whether JSON or text is returned (default to text). The JSON should be structured as follows:

```
{
    "day_of_week": "Monday",
    "day_of_month": 10,
    "month": "April",
    "year": 2023,
    "hour": 20,
    "minute": 15,
    "second": 20
}
```

Wrapping Up

In this chapter, you looked at some of the most commonly used packages in the standard library and saw how they embody best practices that should be emulated in your code. You've also seen other sound software engineering principles: how some decisions might have been made differently given experience, and how to respect backward compatibility so you can build applications on a solid foundation.

In the next chapter, you're going to look at the context, a package and pattern for passing state and timers through Go code.