CHAPTER 11
Go Tooling

A programming language doesn't exist in isolation. For it to be useful, tools must help the developer turn source code into an executable. Since Go is intended to address the problems that software engineers face today and to help them build quality software, careful thought has been put into tooling that simplifies tasks that are often difficult with other development platforms. This includes improvements in how you build, format, update, validate, and distribute, and even how your users will install your code.

I have already covered many of the bundled Go tools: `go vet`, `go fmt`, `go mod`, `go get`, `go list`, `go work`, `go doc`, and `go build`. The testing support provided by the `go test` tool is so extensive, it is covered by itself in Chapter 15. In this chapter, you will explore additional tools that make Go development great, both from the Go team and from third parties.

Using go run to Try Out Small Programs

Go is a compiled language, which means that before Go code is run, it must be converted into an executable file. This is in contrast to interpreted languages like Python or JavaScript, which allow you to write a quick script to test an idea and execute it immediately. Having that rapid feedback cycle is important, so Go provides similar functionality via the `go run` command. It builds and executes a program in one step. Let's go back to the first program from Chapter 1. Put it in a file called *hello.go*:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, world!")
}
```

(You can also find this code in the Chapter 11 repository in the *sample_code/gorun* directory.)

Once the file is saved, use the `go run` command to build and execute it:

```
go run hello.go
Hello, world!
```

If you look inside the directory after running the `go run` command, you see that no binary has been saved there; the only file in the directory is the *hello.go* file you just created. Where did the executable go (no pun intended)?

The `go run` command does, in fact, compile your code into a binary. However, the binary is built in a temporary directory. The `go run` command builds the binary, executes the binary from that temporary directory, and then deletes the binary after your program finishes. This makes the `go run` command useful for testing out small programs or using Go like a scripting language.

Use `go run` when you want to treat a Go program like a script and run the source code immediately.

Adding Third-Party Tools with go install

While some people choose to distribute their Go programs as precompiled binaries, tools written in Go can also be built from source and installed on your computer via the `go install` command.

As you saw in "Publishing Your Module" on page 251, Go modules are identified via their source code repositories. The `go install` command takes an argument, which is the path to the main package in a module's source code repository, followed by an `@` and the version of the tool you want (if you just want to get the latest version, use `@latest`). It then downloads, compiles, and installs the tool.

Always be sure to include the `@version` or `@latest` after the name of the package that you are installing! If you don't, it triggers a variety of confusing behaviors and is almost certainly not what you want to do. You either get an error message (if the current directory is not in a module, or if the current directory is a module, but the package isn't referenced in the module's *go.mod* file), or it installs the package version mentioned in *go.mod*.

By default, `go install` places binaries into the *go/bin* directory within your home directory. Set the *GOBIN* environment variable to change this location. It is strongly recommended that you add the `go install` directory to your executable search path (this is done by modifying the *PATH* environment variable on both Unix and Windows). For simplicity, all the examples in this chapter assume that you've added this directory.

Other environment variables are recognized by the `go` tool. You can get a complete list, along with a brief description of each variable, using the `go help environment` command. Many of them control low-level behavior that can be safely ignored. I'll point out the relevant ones as needed.

Some online resources tell you to set the `GOROOT` or `GOPATH` environment variables. `GOROOT` specifies the location where your Go development environment is installed, and `GOPATH` was used to store all Go source code, both your own and third-party dependencies. Setting these variables is no longer necessary; the `go` tool figures out `GOROOT` automatically, and `GOPATH`-based development has been superseded by modules.

Let's look at a quick example. Jaana Dogan created a great Go tool called `hey` that load tests HTTP servers. You can point it at the website of your choosing or an application that you've written. Here's how to install `hey` with the `go install` command:

```
$ go install github.com/rakyll/hey@latest
go: downloading github.com/rakyll/hey v0.1.4
go: downloading golang.org/x/net v0.0.0-20181017193950-04a2e542c03f
go: downloading golang.org/x/text v0.3.0
```

This downloads `hey` and all its dependencies, builds the program, and installs the binary in your Go binary directory.

As I covered in "Module Proxy Servers" on page 259, the contents of Go repositories are cached in proxy servers. Depending on the repository and the values in your `GOPROXY` environment variable, `go install` may download from a proxy or directly from a repository. If `go install` downloads directly from a repository, it relies on command-line tools being installed on your computer. For example, you must have Git installed to download from GitHub.

Now that you have built and installed `hey`, you can run it with the following:

```
$ hey https://go.dev

Summary:
  Total:          2.1272 secs
  Slowest:        1.4227 secs
  Fastest:        0.0573 secs
  Average:        0.3467 secs
  Requests/sec:   94.0181
```

If you have already installed a tool and want to update it to a newer version, rerun `go install` with the newer version specified or with `@latest`:

```
$ go install github.com/rakyll/hey@latest
```

Of course, you don't need to leave programs installed via `go install` in the *go/bin* directory; they are regular executable binaries and can be stored anywhere on your computer. Likewise, you don't have to distribute programs written in Go using `go install`; you can put a binary up for download. However, `go install` is convenient for Go developers, and it has become the method of choice for distributing third-party developer tools.

Improving Import Formatting with goimports

An enhanced version of `go fmt` called `goimports` also cleans up your import statements. It puts them in alphabetical order, removes unused imports, and attempts to guess any unspecified imports. Its guesses are sometimes inaccurate, so you should insert imports yourself.

You can download `goimports` with the command `go install golang.org/x/tools/cmd/goimports@latest`. You run it across your project with this command:

```
$ goimports -l -w .
```

The `-l` flag tells `goimports` to print the files with incorrect formatting to the console. The `-w` flag tells `goimports` to modify the files in place. The `.` specifies the files to be scanned: everything in the current directory and all its subdirectories.

The packages under `golang.org/x` are part of the Go Project but outside the main Go tree. While useful, they are developed under looser compatibility requirements than the Go standard library and may introduce backward-breaking changes. Some packages in the standard library, such as the context package that is covered in Chapter 14, started out in `golang.org/x`. The `pkgsite` tool that was covered in "Documenting Your Code with Go Doc Comments" on page 231 is also located there. You can see the other packages in the "Sub-repositories" section.

Using Code-Quality Scanners

Back in "go vet" on page 7, you looked at the built-in tool `go vet`, which scans source code for common programming errors. Many third-party tools can check code style and scan for potential bugs that are missed by `go vet`. These tools are often called *linters*.[1] In addition to likely programming errors, some of the changes suggested by these tools include properly naming variables, formatting error messages, and placing comments on public methods and types. These aren't errors since they don't keep your programs from compiling or make your program run incorrectly, but they do flag situations where you are writing nonidiomatic code.

When you add linters to your build process, follow the old maxim "trust, but verify." Since the kinds of issues that linters find are fuzzier, they sometimes have false positives and false negatives. This means you don't *have* to make the changes that they suggest, but you should take the suggestions seriously. Go developers expect code to look a certain way and follow certain rules, and if your code does not, it sticks out.

If you find a linter's suggestion to be unhelpful, each linting tool allows you to add a comment to your source code that blocks the errant result (the format of the comment varies from linter to linter; check each tool's documentation to learn what to write). Your comment should also include an explanation of why you are ignoring the linter's finding, so code reviewers (and future you, when you look back on your source code in six months) understand your reasoning.

1 The term "linter" comes from the original `lint` program written by Steve Johnson when he was on the Unix team at Bell Labs and described in his 1978 paper. The name comes from the tiny bits of fabric that come off clothes in a dryer and are captured by a filter. He saw his program as being like a filter capturing small errors.

staticcheck

If you had to pick one third-party scanner, use `staticcheck`. It is supported by many companies that are active in the Go community, includes more than 150 code-quality checks, and tries to produce few to no false positives. It is installed via `go install honnef.co/go/tools/cmd/staticcheck@latest`. Invoke it with `staticcheck ./...` to examine your module.

Here's an example of something that `staticcheck` finds that `go vet` does not:

```
package main

import "fmt"

func main() {
    s := fmt.Sprintf("Hello")
    fmt.Println(s)
}
```

(You can also find this code in the *sample_code/staticcheck_test* directory in the Chapter 11 repository.)

If you run `go vet` on this code, it doesn't find anything wrong. But, `staticcheck` notices a problem:

```
$ staticcheck ./...
main.go:6:7: unnecessary use of fmt.Sprintf (S1039)
```

Pass the code in parentheses to `staticcheck` with the `-explain` flag for an explanation of the issue:

```
$ staticcheck -explain S1039
Unnecessary use of fmt.Sprint

Calling fmt.Sprint with a single string argument is unnecessary
and identical to using the string directly.

Available since
    2020.1

Online documentation
    https://staticcheck.io/docs/checks#S1039
```

Another common issue that `staticcheck` finds is unused assignments to variables. While the Go compiler requires all variables to be read *once*, it doesn't check that *every* value assigned to a variable is read. It is a common practice to reuse an `err` variable when there are multiple function calls within a function. If you forget to write `if err != nil` after one of those function invocations, the compiler won't be able to help you. However, `staticcheck` can. This code compiles without a problem:

```
func main() {
        err := returnErr(false)
        if err != nil {
                fmt.Println(err)
        }
        err = returnErr(true)
        fmt.Println("end of program")
}
```

(This code is in the Chapter 11 repository in the *sample_code/check_err* directory.)

Running `staticcheck` finds the mistake:

```
$ staticcheck ./...
main.go:13:2: this value of err is never used (SA4006)
main.go:13:8: returnErr doesn't have side effects and its return value is
ignored (SA4017)
```

There are two related issues on line 13. The first is that the error returned by `returnErr` is never read. The second is that the `returnErr` function's output (the error) is being ignored.

revive

Another good linting option is `revive`. It is based on `golint`, a tool that used to be maintained by the Go team. Install `revive` with the command `go install github.com/mgechev/revive@latest`. By default, it enables only the rules that were present in `golint`. It can find style and code-quality issues like exported identifiers that don't have comments, variables that don't follow naming conventions, or error return values that aren't the last return value.

With a configuration file, you can turn on many more rules. For example, to enable a check for shadowing of universe block identifiers, create a file named *built_in.toml* with the following contents:

```
[rule.redefines-builtin-id]
```

If you scan the following code:

```
package main

import "fmt"

func main() {
    true := false
    fmt.Println(true)
}
```

you'll get this warning:

```
$ revive -config built_in.toml ./...
main.go:6:2: assignment creates a shadow of built-in identifier true
```

(You can also find this code in the Chapter 11 repository in the *sample_code/revive_test* directory.)

Other rules that can be enabled are focused on opinionated code organization, like limiting the number of lines in a function or number of public structs in a file. There are even rules for evaluating the complexity of the logic in a function. Check out the `revive` documentation and its supported rules.

golangci-lint

Finally, if you'd rather take the buffet approach to tool selection, there's `golangci-lint`. It is designed to make it as efficient as possible to configure and run over 50 code-quality tools, including `go vet`, `staticcheck`, and `revive`.

While you can use `go install` to install `golangci-lint`, it is recommended that you download a binary version instead. Follow the installation instructions on the website. Once it is installed, you run `golangci-lint`:

```
$ golangci-lint run
```

Back in "Unused Variables" on page 32, you looked at a program with variables set to values that were never read, and I mentioned that `go vet` and the go compiler were unable to detect these issues. Neither `staticcheck` nor `revive` catches this problem. However, one of the tools bundled with `golangci-lint` does:

```
$ golangci-lint run
main.go:6:2: ineffectual assignment to x (ineffassign)
    x := 10
    ^
main.go:9:2: ineffectual assignment to x (ineffassign)
    x = 30
    ^
```

You can also use `golangci-lint` to provide shadowing checks that go beyond what `revive` can do. Configure `golangci-lint` to detect shadowing of both universe block identifiers and identifiers within your own code by putting the following configuration into a file named *.golangci.yml* in the directory where you run `golangci-lint`:

```
linters:
  enable:
    - govet
    - predeclared

linters-settings:
  govet:
    check-shadowing: true
    settings:
```

```
      shadow:
        strict: true
    enable-all: true
```

With these settings, running `golangci-lint` on this code

```
package main

import "fmt"

var b = 20

func main() {
    true := false
    a := 10
    b := 30
    if true {
        a := 20
        fmt.Println(a)
    }
    fmt.Println(a, b)
}
```

detects the following issues:

```
$ golangci-lint run
main.go:5:5: var `b` is unused (unused)
var b = 20
    ^
main.go:10:2: shadow: declaration of "b" shadows declaration at line 5 (govet)
    b := 30
    ^
main.go:12:3: shadow: declaration of "a" shadows declaration at line 9 (govet)
        a := 20
        ^
main.go:8:2: variable true has same name as predeclared identifier (predeclared)
    true := false
    ^
```

(You can find both `golangci-lint` code samples in the Chapter 11 repository in the *sample_code/golangci-lint_test* directory.)

Because `golangci-lint` runs so many tools (as of this writing, it runs 7 different tools by default and allows you to enable more than 50 more), it's inevitable that your team may disagree with some of its suggestions. Review the documentation to understand what each tool can do. Once you come to agreement on which linters to enable, update the *.golangci.yml* file at the root of your module and commit it to source control. Check out the documentation for the file format.

While `golangci-lint` allows you to have a configuration file in your home directory, don't put one there if you are working with other developers. Unless you enjoy adding hours of silly arguments to your code reviews, you want to make sure that everyone is using the same code-quality tests and formatting rules.

I recommend that you start using `go vet` as a required part of your automated build process. Add `staticcheck` next since it produces few false positives. When you are interested in configuring tools and setting code-quality standards, look at `revive`, but be aware that it might have false positives and false negatives, so you can't require your team to fix every issue it reports. Once you are used to their recommendations, try out `golangci-lint` and tweak its settings until it works for your team.

Using govulncheck to Scan for Vulnerable Dependencies

One kind of code quality isn't enforced by the tools you've looked at so far: software vulnerabilities. Having a rich ecosystem of third-party modules is fantastic, but clever hackers find security vulnerabilities in libraries and exploit them. Developers patch these bugs when they are reported, but how do you ensure that the software that uses a vulnerable version of a library is updated to the fixed version?

The Go team has released a tool called `govulncheck` to address this situation. It scans through your dependencies and finds known vulnerabilities in both the standard library and in third-party libraries imported into your module. These vulnerabilities are reported in a public database maintained by the Go team. You can install it with this:

```
$ go install golang.org/x/vuln/cmd/govulncheck@latest
```

Let's take a look at a small program to see the vulnerability checker in action. First, download the repository. The source code in *main.go* is very simple. It imports a third-party YAML library and uses it to load a small YAML string into a struct:

```
func main() {
    info := Info{}

    err := yaml.Unmarshal([]byte(data), &info)
    if err != nil {
        fmt.Printf("error: %v\n", err)
        os.Exit(1)
    }
    fmt.Printf("%+v\n", info)
}
```

The *go.mod* file contains the required modules and their versions:

```
module github.com/learning-go-book-2e/vulnerable
```

```
go 1.20

require gopkg.in/yaml.v2 v2.2.7

require gopkg.in/check.v1 v1.0.0-20201130134442-10cb98267c6c // indirect
```

Let's see what happens when you run `govulncheck` on this project:

```
$ govulncheck ./...
Using go1.21 and govulncheck@v1.0.0 with vulnerability data from
    https://vuln.go.dev (last modified 2023-07-27 20:09:46 +0000 UTC).

Scanning your code and 49 packages across 1 dependent module
    for known vulnerabilities...

Vulnerability #1: GO-2020-0036
    Excessive resource consumption in YAML parsing in gopkg.in/yaml.v2
  More info: https://pkg.go.dev/vuln/GO-2020-0036
  Module: gopkg.in/yaml.v2
    Found in: gopkg.in/yaml.v2@v2.2.7
    Fixed in: gopkg.in/yaml.v2@v2.2.8
    Example traces found:
      #1: main.go:25:23: vulnerable.main calls yaml.Unmarshal

Your code is affected by 1 vulnerability from 1 module.
```

This module is using an old and vulnerable version of the YAML package. `govuln check` helpfully gives the exact line in the codebase that calls the problematic code.

If `govulncheck` knows there a vulnerability in a module that your code uses, but can't find an explicit call to the buggy part of the module, you'll get a less severe warning. The message informs you of the library's vulnerability and what version resolves the issue, but it will also let you know that your module is likely not affected.

Let's update to a fixed version and see if that solves the problem:

```
$ go get -u=patch gopkg.in/yaml.v2
go: downloading gopkg.in/yaml.v2 v2.2.8
go: upgraded gopkg.in/yaml.v2 v2.2.7 => v2.2.8
$ govulncheck ./...
Using go1.21and govulncheck@v1.0.0 with vulnerability data from
    https://vuln.go.dev (last modified 2023-07-27 20:09:46 +0000 UTC).

Scanning your code and 49 packages across 1 dependent module
    for known vulnerabilities...

No vulnerabilities found.
```

Remember, you should always strive for the smallest possible change to your project's dependencies since that makes it less likely that a change in a dependency breaks your code. For that reason, update to the most recent patch version for v2.2.*x*, which is v2.2.8. When `govulncheck` is run again, there are no known issues.

While `govulncheck` currently requires a `go install` to download it, it likely will be added to the standard toolset eventually. In the meantime, be sure to install and run it against your projects as a regular part of your builds. You can learn more about it in the blog post that announced it.

Embedding Content into Your Program

Many programs are distributed with directories of support files; you might have web page templates or some standard data that's loaded when a program starts. If a Go program needs support files, you could include a directory of files, but this takes away one of the advantages of Go: its ability to compile to a single binary that's easy to ship and distribute. However, there's another option. You can embed the contents of the files within your Go binary by using `go:embed` comments.

You can find a program demonstrating embedding on GitHub in the *sample_code/embed_passwords* directory in the Chapter 11 repository. It checks to see if a password is one of the 10,000 most commonly used passwords. Rather than write that list of passwords directly into the source code, you're going to embed it.

The code in *main.go* is straightforward:

```
package main

import (
    _ "embed"
    "fmt"
    "os"
    "strings"
)

//go:embed passwords.txt
var passwords string

func main() {
    pwds := strings.Split(passwords, "\n")
    if len(os.Args) > 1 {
        for _, v := range pwds {
            if v == os.Args[1] {
                fmt.Println("true")
                os.Exit(0)
            }
        }
        fmt.Println("false")
```

```
    }
}
```

You must do two things to enable embedding. First, the `embed` package must be imported. The Go compiler uses this import as a flag to indicate that embedding should be enabled. Because this sample code isn't referring to anything exported from the `embed` package, you use a blank import, which was discussed in "Avoiding the init Function if Possible" on page 239. The only symbol exported from `embed` is `FS`. You'll see it in the next example.

Next, you place a magic comment directly before each package-level variable that holds the contents of a file. This comment must start with `go:embed`, with no space between the slashes and `go:embed`. The comment must also be on the line directly before the variable. (Technically, it is legal to have blank lines or other, nonmagic comments between the embedding comment and the variable declaration, but don't do it.) In this sample, you are embedding the contents of *passwords.txt* into the package-level variable named `passwords`. It is idiomatic to treat a variable with an embedded value as immutable. As mentioned earlier, you can embed into only a package-level variable. The variable must be of type `string`, `[]byte`, or `embed.FS`. If you have a single file, it's simplest to use `string` or `[]byte`.

If you need to place one or more directories of files into your program, use a variable of type `embed.FS`. This type implements three interfaces defined in the `io/fs` package: `FS`, `ReadDirFS`, and `ReadFileFS`. This allows an instance of `embed.FS` to represent a virtual filesystem. The following program provides a simple command-line help system. If you don't provide a help file, it lists all available files. If you specify a file that's not present, it returns an error:

```
package main

import (
    "embed"
    "fmt"
    "io/fs"
    "os"
    "strings"
)

//go:embed help
var helpInfo embed.FS

func main() {
    if len(os.Args) == 1 {
        printHelpFiles()
        os.Exit(0)
    }
    data, err := helpInfo.ReadFile("help/" + os.Args[1])
    if err != nil {
```

```
        fmt.Println(err)
        os.Exit(1)
    }
    fmt.Println(string(data))
}
```

You can find this code, along with the sample help files, in the *sample_code/help_system* directory in the Chapter 11 repository.

Here's the output when you build and run this program:

```
$ go build
$ ./help_system
contents:
advanced/topic1.txt
advanced/topic2.txt
info.txt

$ ./help_system advanced/topic1.txt
This is advanced topic 1.

$ ./help_system advanced/topic3.txt
open help/advanced/topic3.txt: file does not exist
```

You should notice a couple of things. First, you no longer need to use a blank import for `embed`, since you are using `embed.FS`. Second, the directory name is part of the filesystem that's embedded. The users of this program don't enter the "help/" prefix, so you have to prepend it in the call to `ReadFile`.

The `printHelpFiles` function shows how you can treat an embedded virtual filesystem just like a real one:

```
func printHelpFiles() {
    fmt.Println("contents:")
    fs.WalkDir(helpInfo, "help",
        func(path string, d fs.DirEntry, err error) error {
            if !d.IsDir() {
                _, fileName, _ := strings.Cut(path, "/")
                fmt.Println(fileName)
            }
            return nil
        })
}
```

You use the `WalkDir` function in `io/fs` to walk through the embedded filesystem. `WalkDir` takes in an instance of `fs.FS`, a path to start at, and a function. This function is called for every file and directory in the filesystem, starting from the specified path. If the `fs.DirEntry` is not a directory, you print out its full pathname, removing the `help/` prefix by using `strings.Cut`.

There are a few more things to know about file embedding. While all the examples have been text files, you can embed binary files as well. You can also embed multiple files or directories into a single `embed.FS` variable by specifying their names, separated by spaces. When embedding a file or directory that has a space in its name, put the name in quotes.

In addition to exact file and directory names, you can use wildcards and ranges to specify the names of the files and directories you want to embed. The syntax is defined in the documentation for the Match function in the `path` package in the standard library, but it follows common conventions. For example, `*` matches 0 or more characters, and `?` matches a single character.

All embedding specifications, whether or not they use match patterns, are checked by the compiler. If they aren't valid, compilation fails. Here are the ways a pattern can be invalid:

- If the specified name or pattern doesn't match a file or directory
- If you specify multiple filenames or patterns for a `string` or `[]byte` variable
- If you specify a pattern for a `string` or `[]byte` variable and it matches more than one file

Embedding Hidden Files

Including files in a directory tree that start with `.` or `_` is a little complicated. Many operating systems consider these to be hidden files, so they are not included by default when a directory name is specified. However, you can override this behavior in two ways. The first is to put `/*` after the name of a directory you want to embed. This will include all hidden files within the root directory, but it will not include hidden files in its subdirectories. To include all hidden files in all subdirectories, put `all:` before the name of the directory.

This sample program (which you can find in the *sample_code/embed_hidden* directory in the Chapter 11 repository) makes this easier to understand. In the sample, the directory *parent_dir* contains two files, *.hidden* and *visible*, and one subdirectory, *child_dir*. The *child_dir* subdirectory contains two files, *.hidden* and *visible*.

Here is the code for the program:

```
//go:embed parent_dir
var noHidden embed.FS

//go:embed parent_dir/*
var parentHiddenOnly embed.FS

//go:embed all:parent_dir
var allHidden embed.FS
```

```
func main() {
    checkForHidden("noHidden", noHidden)
    checkForHidden("parentHiddenOnly", parentHiddenOnly)
    checkForHidden("allHidden", allHidden)
}

func checkForHidden(name string, dir embed.FS) {
    fmt.Println(name)
    allFileNames := []string{
        "parent_dir/.hidden",
        "parent_dir/child_dir/.hidden",
    }
    for _, v := range allFileNames {
        _, err := dir.Open(v)
        if err == nil {
            fmt.Println(v, "found")
        }
    }
    fmt.Println()
}
```

The output of the program is shown here:

```
noHidden

parentHiddenOnly
parent_dir/.hidden found

allHidden
parent_dir/.hidden found
parent_dir/child_dir/.hidden found
```

Using go generate

The `go generate` tool is a little different, because it doesn't do anything by itself. When you run `go generate`, it looks for specially formatted comments in your source code and runs programs specified in those comments. While you could use `go generate` to run anything at all, it is most commonly used by developers to run tools that (unsurprisingly, given the name) generate source code. This could be from analyzing existing code and adding functionality or processing schemas and making source code out of it.

A good example of something that can be automatically converted to code are Protocol Buffers, sometimes called *protobufs*. Protobuf is a popular binary format that is used by Google to store and transmit data. When working with protobufs, you write a *schema*, which is a language-independent description of the data structure. Developers who want to write programs to interact with data in protobuf format run

tools that process the schema and produce language-specific data structures to hold the data and language-specific functions to read and write data in protobuf format.

Let's see how this works in Go. You can find a sample module in the proto_generate repo. The module contains a protobuf schema file called *person.proto*:

```
syntax = "proto3";

message Person {
  string name = 1;
  int32 id = 2;
  string email = 3;
}
```

While making a struct that implements `Person` would be easy, writing the code to convert back and forth from the binary format is difficult. Let's use tools from Google to do the hard work and invoke them with `go generate`. You need to install two things. The first is the `protoc` binary for your computer (see the installation instructions. Next, use `go install` to install the Go protobuf plug-ins:

```
$ go install google.golang.org/protobuf/cmd/protoc-gen-go@v1.28
```

In *main.go*, there is the magic comment that's processed by `go generate`:

```
//go:generate protoc -I=. --go_out=.
  --go_opt=module=github.com/learning-go-book-2e/proto_generate
  --go_opt=Mperson.proto=github.com/learning-go-book-2e/proto_generate/data
  person.proto
```

(If you look at the source code in GitHub, you'll see this should be a single line. It's wrapped to fit the constraints of a printed page.)

Run `go generate` by typing the following:

```
$ go generate ./...
```

After running `go generate`, you'll see a new directory called *data* that contains a file named *person.pb.go*. It contains the source code for the `Person` struct, and some methods and functions that are used by the `Marshal` and `Unmarshal` functions in the `google.golang.org/protobuf/proto` module. You call these functions in your `main` function:

```
func main() {
    p := &data.Person{
        Name:  "Bob Bobson",
        Id:    20,
        Email: "bob@bobson.com",
    }
    fmt.Println(p)
    protoBytes, _ := proto.Marshal(p)
    fmt.Println(protoBytes)
    var p2 data.Person
```

```
    proto.Unmarshal(protoBytes, &p2)
    fmt.Println(&p2)
}
```

Build and run the program as usual:

```
$ go build
$ ./proto_generate
name:"Bob Bobson"  id:20  email:"bob@bobson.com"
[10 10 66 111 98 32 66 111 98 115 111 110 16 20 26 14 98
  111 98 64 98 111 98 115 111 110 46 99 111 109]
name:"Bob Bobson"  id:20  email:"bob@bobson.com"
```

Another tool commonly used with `go generate` is `stringer`. As I discussed in "iota Is for Enumerations—Sometimes" on page 152, enumerations in Go lack many of the features that are found in other languages with enumerations. One of those features is automatically generating a printable name for each value in the enumeration. The `stringer` tool is used with `go generate` to add a `String` method to your enumeration's values so they can be printed.

Install `stringer` with `go install golang.org/x/tools/cmd/stringer@latest`. The *sample_code/stringer_demo* directory in the Chapter 11 repository provides a very simple example of how to use `stringer`. Here's the source in *main.go*:

```
type Direction int

const (
    _ Direction = iota
    North
    South
    East
    West
)

//go:generate stringer -type=Direction

func main() {
    fmt.Println(North.String())
}
```

Run `go generate ./...` and you'll see a new file generated called *direction_string.go*. Use `go build` to build the `string_demo` binary and when you run it, you'll get the output:

```
North
```

You can configure `stringer` and its output in multiple ways. Arjun Mahishi has written a great blog post describing how to use `stringer` and customize its output.

Working with go generate and Makefiles

Since the job of `go generate` is to run other tools, you might wonder if it's worth using when you have a perfectly good Makefile in your project. The advantage of `go generate` is that it creates a separation of responsibilities. Use `go generate` commands to mechanically create source code, and use the Makefile to validate and compile source code.

It is a best practice to commit the source code created by `go generate` to version control. (The sample projects in the Chapter 11 repository don't include generated source code so you can see `go generate` work.) This allows people browsing your source code to see everything that's invoked, even the generated parts. It also means they don't need to have tools like `protoc` installed in order to build your code.

Checking in your generated source code technically means that you don't *need* to run `go generate` unless it will produce different output, such as processing a modified protobuf definition or an updated enumeration. However, it's still a good idea to automate calling `go generate` before `go build`. Relying on a manual process is asking for trouble. Some generator tools, like `stringer`, include clever tricks to block compilation if you forget to rerun `go generate`, but that's not universal. You'll inevitably waste time during testing trying to understand why a change didn't show up before realizing that you forgot to invoke `go generate`. (I made this mistake multiple times before I learned my lesson.) Given this, it is best to add a `generate` step to your Makefile and make it a dependency of your `build` step.

However, I would disregard this advice in two situations. The first is if invoking `go generate` on identical input produces source files with minor differences, such as a timestamp. A well-written `go generate` tool should produce identical output every time it's run on the same input, but there are no guarantees that every tool you need to use is well written. You don't want to keep on checking in new versions of files that are functionally identical, as they will clutter your version control system and make your code reviews noisier.

The second situation is if `go generate` takes a very long time to complete. Fast builds are a feature of Go, because they allow developers to stay focused and get rapid feedback. If you are noticeably slowing down a build to generate identical files, the loss in developer productivity is not worth it. In both cases, all you can do is leave lots of comments to remind people to rebuild when things change and hope that everyone on your team is diligent.

Reading the Build Info Inside a Go Binary

As companies develop more of their own software, it is becoming increasingly common for them to want to understand exactly what they have deployed to their data

centers and cloud environments, down to the version and the dependencies. You might wonder why you'd want to get this information from compiled code. After all, a company already has this information in version control.

Companies with mature development and deployment pipelines can capture this information right before deploying a program, allowing them to be sure that the information is accurate. However, many, if not most, companies don't track the exact version of internal software that's deployed. In some cases, software can be deployed for years without being replaced, and no one remembers much about it. If a vulnerability is reported in a version of a third-party library, you need to either find some way to scan your deployed software and figure out what versions of third-party libraries are deployed, or redeploy everything just to be safe. In the Java world, this exact problem happened when a serious vulnerability was discovered in the popular Log4j library.

Luckily, Go solves this problem for you. Every Go binary you create with `go build` automatically contains build information on what versions of what modules make up the binary, and also what build commands were used, what version control system was used, and what revision the code was at in your version control system. You can view this information with the `go version -m` command. The following shows the output for the `vulnerable` program when built on an Apple Silicon Mac:

```
$ go build
go: downloading gopkg.in/yaml.v2 v2.2.7
$ go version -m vulnerable
vulnerable: go1.20
    path     github.com/learning-go-book-2e/vulnerable
    mod      github.com/learning-go-book-2e/vulnerable    (devel)
    dep      gopkg.in/yaml.v2  v2.2.7  h1:VUgggvou5XRW9mHwD/yXxIYSMtY0zoKQf/v...
    build    -compiler=gc
    build    CGO_ENABLED=1
    build    CGO_CFLAGS=
    build    CGO_CPPFLAGS=
    build    CGO_CXXFLAGS=
    build    CGO_LDFLAGS=
    build    GOARCH=arm64
    build    GOOS=darwin
    build    vcs=git
    build    vcs.revision=623a65b94fd02ea6f18df53afaaea3510cd1e611
    build    vcs.time=2022-10-02T03:31:05Z
    build    vcs.modified=false
```

Because this information is embedded into every binary, `govulncheck` is capable of scanning Go programs to check for libraries with known vulnerabilities:

```
$ govulncheck -mode binary vulnerable
Using govulncheck@v1.0.0 with vulnerability data from
    https://vuln.go.dev (last modified 2023-07-27 20:09:46 +0000 UTC).
```

```
Scanning your binary for known vulnerabilities...

Vulnerability #1: GO-2020-0036
    Excessive resource consumption in YAML parsing in gopkg.in/yaml.v2
  More info: https://pkg.go.dev/vuln/GO-2020-0036
  Module: gopkg.in/yaml.v2
    Found in: gopkg.in/yaml.v2@v2.2.7
    Fixed in: gopkg.in/yaml.v2@v2.2.8
    Example traces found:
      #1: yaml.Unmarshal

Your code is affected by 1 vulnerability from 1 module.
```

Be aware that `govulncheck` can't track down exact lines of code when inspecting a binary. If `govulncheck` finds a problem in a binary, use `go version -m` to find out the exact deployed version, check the code out of version control, and run it again against the source code to pinpoint the issue.

If you want to build your own tools to read the build information, look at the `debug/buildinfo` package in the standard library.

Building Go Binaries for Other Platforms

One of the advantages of a VM-based language like Java, JavaScript, or Python is that you can take your code and get it to run on any computer where the virtual machine has been installed. This portability makes it easy for developers using these languages to build programs on a Windows or Mac computer and deploy it on a Linux server, even though the operating system and possibly the CPU architecture are different.

Go programs are compiled to native code, so the generated binary is compatible with only a single operating system and CPU architecture. However, that doesn't mean that Go developers need to maintain a menagerie of machines (virtual or otherwise) to release on multiple platforms. The `go build` command makes it easy to *cross-compile*, or create a binary for a different operating system and/or CPU. When `go build` is run, the target operating system is specified by the `GOOS` environment variable. Similarly, the `GOARCH` environment variable specifies the CPU architecture. If you don't set them explicitly, `go build` defaults to using the values for your current computer, which is why you've never had to worry about these variables before.

You can find the valid values and combinations for `GOOS` and `GOARCH` (sometimes pronounced "GOOSE" and "GORCH") in the installation documentation. Some of the supported operating systems and CPUs are a bit esoteric, and others might require some translation. For example, `darwin` refers to macOS (Darwin is the name of the macOS kernel), and `amd64` means 64-bit Intel-compatible CPUs.

Let's go back to the `vulnerable` program one last time. When using an Apple Silicon Mac (which has an ARM64 CPU), running `go build` defaults to `darwin` for `GOOS` and `arm64` for `GOARCH`. You can confirm this using the `file` command:

```
$ go build
$ file vulnerable
vulnerable: Mach-O 64-bit executable arm64
```

Here is how to build a binary for Linux on 64-bit Intel CPUs:

```
$ GOOS=linux GOARCH=amd64 go build
$ file vulnerable
vulnerable: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
    statically linked, Go BuildID=IDHVCE8XQPpWluGpMXpX/4VU3GpRZEifN
    8TzUrT_6/1c30VcDYNVPfSSN-zCkz/JsZSLAbWkxqIVhPkC5p5, with debug_info,
    not stripped
```

Using Build Tags

When writing programs that need to run on multiple operating systems or CPU architectures, you sometimes need different code for different platforms. You also might want to write a module that takes advantage of the latest Go features but is still backward compatible with older Go compilers.

You can create targeted code in two ways. The first is to use the name of the file to indicate when the file should be included in the build. You do this by adding the target *GOOS* and *GOARCH*, separated by `_`, to the filename before *.go*. For example, if you have a file that you want to be compiled only on Windows, you'd name the file *something_windows.go*, but if you wanted it to be compiled only when building for ARM64 Windows, name the file *something_windows_arm64.go*.

A *build tag* (also called a *build constraint*) is the other option you can use to specify when a file is compiled. Like embedding and generating, build tags take advantage of a magic comment. In this case, it's `//go:build`. This comment must be placed on the line before the package declaration in your file.

Build tags use boolean operators (`||`, `&&`, and `!`) and parentheses to specify exact build rules for architectures, operating systems, and Go versions. The build tag `//go:build (!darwin && !linux) || (darwin && !go1.12)`—which really appears in the Go standard library—specifies that the file should not be compiled on Linux or macOS, except it's OK to compile it on macOS if the Go version is 1.11 or earlier.

Some meta build constraints are also available. The constraint `unix` matches any Unix-ish platform, and `cgo` matches if `cgo` is supported by the current platform and is enabled. (I cover `cgo` in "Cgo Is for Integration, Not Performance" on page 433.)

The question becomes when you should use filenames to indicate where to run code and when you should use build tags. Because build tags allow binary operators, you

can specify a more specific set of platforms with them. The Go standard library sometimes takes a belt-and-suspenders approach. The package `internal/cpu` in the standard library has platform-specific source code for CPU feature detection. The file *internal/cpu/cpu_arm64_darwin.go* has a name that indicates that it is meant only for computers using Apple CPUs. It also has a `//go:build arm64 && darwin && !ios` line in the file to indicate that it should be compiled only when building for Apple Silicon Macs and not for iPhones or iPads. The build tags are able to specify the target platform with more detail, but following the filename convention makes it easy for a person to find the right file for a given platform.

In addition to the built-in build tags that represent Go versions, operating systems and CPU architectures, you can also use any string at all as a custom build tag. You can then control compilation of that file with the `-tags` command-line flag. For example, if you put `//go:build gopher` on the line before the package declaration in a file, it will not be compiled unless you include a `-tags gopher` flag as part of the `go build`, `go run`, or `go test` command.

Custom build tags are surprisingly handy. If you have a source file that you don't want to build right now (perhaps it doesn't compile yet, or it's an experiment that's not ready to be included), it is idiomatic to skip over the file by putting `//go:build ignore` on the line before the package declaration. You will see another use for custom build tags when looking at integration tests in "Using Integration Tests and Build Tags" on page 405.

When writing your build tags, make sure there isn't any whitespace between the `//` and `go:build`. If there is, Go will not consider it a build tag.

Testing Versions of Go

Despite Go's strong backward-compatibility guarantees, bugs do happen. It's natural to want to make sure that a new release doesn't break your programs. You also might get a bug report from a user of your library saying that your code doesn't work as expected on an older version of Go. One option is to install a secondary Go environment. For example, if you wanted to try out version 1.19.2, you would use the following commands:

```
$ go install golang.org/dl/go1.19.2@latest
$ go1.19.2 download
```

You can then use the command `go1.19.2` instead of the `go` command to see if version 1.19.2 works for your programs:

```
$ go1.19.2 build
```

Once you have validated that your code works, you can uninstall the secondary environment. Go stores secondary Go environments in the *sdk* directory within your home directory. To uninstall, delete the environment from the *sdk* directory and the binary from the *go/bin* directory. Here's how to do that on macOS, Linux, and BSD:

```
$ rm -rf ~/sdk/go.19.2
$ rm ~/go/bin/go1.19.2
```

Using go help to Learn More About Go Tooling

You can learn more about Go's tooling and runtime environment with the `go help` command. It contains exhaustive information about all the commands mentioned here, as well as things like modules, import path syntax, and working with nonpublic source code. For example, you can get information on import path syntax by typing `go help importpath`.

Exercises

These exercises cover some of the tools that you've learned about in this chapter. You can find the solutions in the *exercise_solutions* directory in the Chapter 11 repository.

1. Go to the UN's Universal Declaration of Human Rights (UDHR) page and copy the text of the UDHR into a text file called *english_rights.txt*. Click the Other Languages link and copy the document text in a few additional languages into files named *LANGUAGE_rights.txt*. Create a program that embeds these files into package-level variables. Your program should take in one command-line parameter, the name of a language. It should then print out the UDHR in that language.
2. Use `go install` to install `staticcheck`. Run it against your program and fix any problems it finds.
3. Cross-compile your program for ARM64 on Windows. If you are using an ARM64 Windows computer, cross-compile for AMD64 on Linux.

Wrapping Up

In this chapter, you learned about the tools that Go provides to improve software engineering practices and third-party code-quality tools. In the next chapter, you're going to explore one of the signature features in Go: concurrency.