Appendix

As you set forth on your journey of writing concurrent code, you'll need the tools to write your program and analyze it for correctness, and a few helpful pointers to help you understand what's happening within your programs. Lucky for you, the Go ecosystem has a rich set of tooling both from the Go team and from the community! This appendix will discuss some of these tools and how they can aid you before, during, and after development. Since this book is focused on concurrency, I'm going to constrain the conversation to only topics that help you write or analyze concurrent code. We'll also briefly look at what happens when goroutines panic. It doesn't happen often, but the output can be a bit confusing the first time you see it.

Anatomy of a Goroutine Error

It happens to the best of us: sooner or later, your program will panic. If you're lucky, no humans or computers will be harmed in the process, and the worst that will happen is you'll be staring down the bad end of a stack trace.

Prior to Go 1.6, when a goroutine panicked, the runtime would print stack traces of all the currently executing goroutines. Sometimes this made it difficult (or at least time-consuming) to determine what had happened. At the time of this writing, Go 1.6 and greater greatly simplify things by printing only the stack trace of the panicking goroutine.

For example, when this simple program is executed:

```
1 package main
2
3 func main() {
4     waitForever := make(chan interface{})
5     go func() {
6         panic("test panic")
7     }()
8     <-waitForever
9 }
```

The following stack trace is produced:

```
panic: test panic

goroutine 4 [running]:
main.main.func1() ❸
    /tmp/babel-3271QbD/go-src-32713Rn.go:6 +0x65 ❶
created by main.main
    /tmp/babel-3271QbD/go-src-32713Rn.go:7 +0x4e ❷
exit status 2
```

❶ Refers to where the panic occurred.

❷ Refers to where the goroutine was started.

❸ Indicates the name of the function running as a goroutine. If it's an anonymous function as in this example, an automatic and unique identifier is assigned.

If you'd like to see the stack traces of all the goroutines that were executing when the program panicked, you can enable the old behavior by setting the `GOTRACEBACK` environmental variable to `all`.

Race Detection

In Go 1.1, a `-race` flag was added as a flag for most `go` commands:

```
$ go test -race mypkg    # test the package
$ go run -race mysrc.go  # compile and run the program
$ go build -race mycmd   # build the command
$ go install -race mypkg # install the package
```

If you're a developer and all you need is a more reliable way to detect race conditions, this is really all you need to know. One caveat of using the race detector is that the algorithm will only find races that are contained in code that is exercised. For this reason, the Go team recommends running a build of your application built with the race flag under real-world load. This increases the probability of finding races by virtue of increasing the probability that more code is exercised.

There are also some options you can specify via environmental variables to tweak the behavior of the race detector, although generally the defaults are sufficient:

LOG_PATH
: This tells the race detector to write reports to the *LOG_PATH.pid* file. You can also pass it special values: `stdout` and `stderr`. The default value is `stderr`.

STRIP_PATH_PREFIX
: This tells the race detector to strip the beginnings of file paths in reports to make them more concise.

HISTORY_SIZE
: This sets the per-goroutine history size, which controls how many previous memory accesses are remembered per goroutine. The valid range of values is [0, 7]. The memory allocated for goroutine history begins at 32 KB when `HISTORY_SIZE` is 0, and doubles with each subsequent value for a maximum of 4 MB at a `HISTORY_SIZE` of 7. When you see "failed to restore the stack" in reports, that's an indicator to increase this value; however, it can significantly increase memory consumption.

Given this simple program we first looked at in Chapter 1:

```
1 var data int
2 go func() { ❶
3     data++
4 }()
5 if data == 0 {
6     fmt.Printf("the value is %v.\n", data)
7 }
```

You would receive this error:

```
==================
WARNING: DATA RACE
Write by goroutine 6:
  main.main.func1()
      /tmp/babel-10285ejY/go-src-10285GUP.go:6 +0x44 ❶

Previous read by main goroutine:
  main.main()
      /tmp/babel-10285ejY/go-src-10285GUP.go:7 +0x8e ❷

Goroutine 6 (running) created at:
  main.main()
      /tmp/babel-10285ejY/go-src-10285GUP.go:6 +0x80
==================
Found 1 data race(s)
exit status 66
```

❶ Signifies a goroutine that is attempting to write unsynchronized memory access.

❷ Signifies a goroutine (in this case the main goroutine) trying to read this same memory.

The race detector is an extremely useful tool for automatically detecting race conditions in your code. I highly recommend integrating it as part of your continuous integration process. Again, because the race detection can only detect races that occur, and we covered how race conditions are sometime tricky to trigger, it should be continuously running real-world scenarios in an attempt to trigger one.

pprof

In large codebases, it can sometimes be difficult to ascertain how your program is performing at runtime. How many goroutines are running? Are your CPUs being fully utilized? How's memory usage doing? Profiling is a great way to answer these questions, and Go has a package in the standard library to support a profiler named "pprof."

pprof is a tool that was created at Google and can display profile data either while a program is running, or by consuming saved runtime statistics. The usage of the program is pretty well described by its help flag, so instead we'll stick to discussing the `runtime/pprof` package here—specifically as it pertains to concurrency.

The `runtime/pprof` package is pretty simple, and has predefined profiles to hook into and display:

```
goroutine    - stack traces of all current goroutines
heap         - a sampling of all heap allocations
threadcreate - stack traces that led to the creation of new OS threads
block        - stack traces that led to blocking on synchronization primitives
mutex        - stack traces of holders of contended mutexes
```

From the context of concurrency, most of these are useful for understanding what's happening within your running program. For example, here's a goroutine that can help you detect goroutine leaks:

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

// Every second, log how many goroutines are currently running.
go func() {
    goroutines := pprof.Lookup("goroutine")
    for range time.Tick(1*time.Second) {
        log.Printf("goroutine count: %d\n", goroutines.Count())
    }
}()

// Create some goroutines which will never exit.
var blockForever chan struct{}
for i := 0; i < 10; i++ {
    go func() { <-blockForever }()
    time.Sleep(500*time.Millisecond)
}
```

These built-in profiles can really help you profile and diagnose issues with your program, but of course you can write custom profiles tailored to help you monitor your programs:

```
func newProfIfNotDef(name string) *pprof.Profile {
    prof := pprof.Lookup(name)
    if prof == nil {
        prof = pprof.NewProfile(name)
    }
    return prof
}

prof := newProfIfNotDef("my_package_namespace")
```