Go: How Does a Goroutine Start and Exit?

Vincent Blanchon Follow
Apr 1 · 3 min read ★

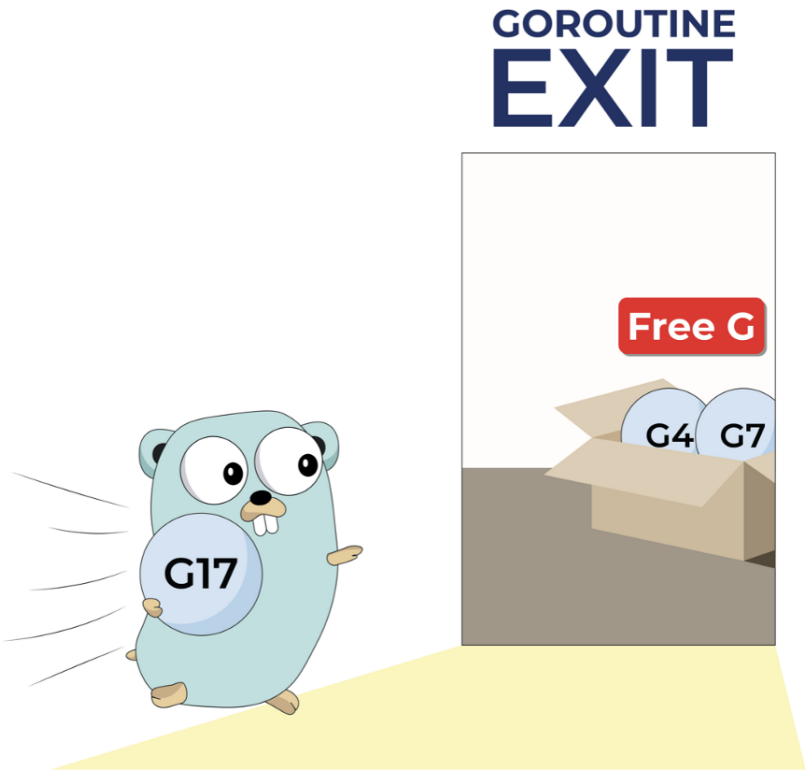


Illustration created for "A Journey With Go", made from the original Go Gopher, created by Renee French.

ℹ️ *This article is based on Go 1.14.*

In Go, a goroutine is nothing but a Go structure containing information regarding the running program, such as stack, program counter, or its current OS thread. The Go scheduler deals with that information to give them running time. The scheduler also has to pay attention at the start and the exit of the goroutines, two phases that need to be managed carefully.

For more information about the stack and the program counter, I suggest you read my article "Go: What a Goroutine Switch Actually Involve?"

Start

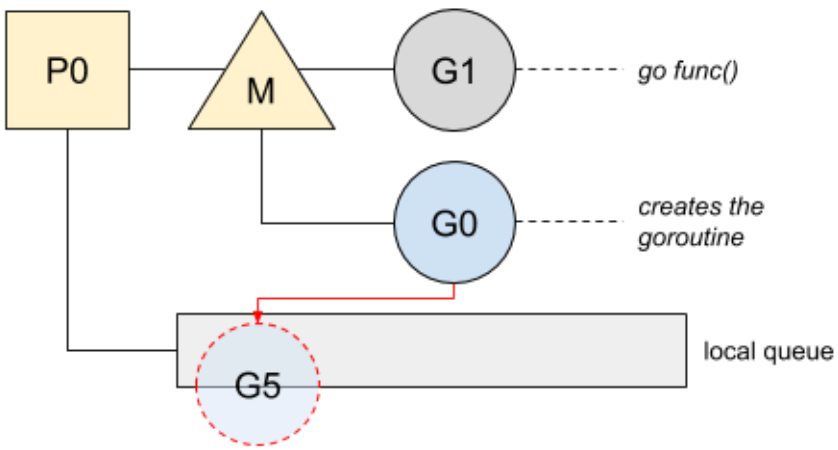
The process of starting a goroutine is quite simple. Let's use a program as an example:

```
func main() {
    var wg sync.WaitGroup
    wg.Add( delta: 1)

    go func() {
        println( args...: "goroutine is running...")
        wg.Done()
    }()

    println( args...: "main is running...")

    wg.Wait()
}
```

The `main` function starts a goroutine before printing a message. Since the goroutine will have its own running time, Go notifies the runtime to set up a new goroutine, meaning:

- Creating the stack.
- Collecting information about the current program counter or caller's data.
- Updating internal data of the goroutine such as ID or status.

However, the goroutine does not get any runtime immediately. The newly created goroutine will be enqueued at the beginning of the local queue and will run at the next round of the Go scheduler. Here is a diagram of the current state:



Putting the goroutine at the head of the queue makes it the first to run after the current goroutine. It will run either on the same thread or on another one if any work-stealing happens.

For more information about the work-stealing, I suggest you read my article "Go: Work-Stealing in Go Scheduler."

The goroutine creation also can be seen in the assembly instructions:

```
                                          Generated name for the anonymous function
0x0054 00084 (main.go:11)  LEAQ   "".main.func1·f(SB), AX
0x005b 00091 (main.go:11)  PCDATA $0, $0
0x005b 00091 (main.go:11)  MOVQ   AX, 8(SP)  ──→  Address of the func is pushed
                                                   to the stack
0x0060 00096 (main.go:11)  PCDATA $0, $1
0x0060 00096 (main.go:11)  MOVQ   "".&wg+24(SP), AX
0x0065 00101 (main.go:11)  PCDATA $0, $0
0x0065 00101 (main.go:11)  MOVQ   AX, 16(SP)
0x006a 00106 (main.go:11)  CALL   runtime.newproc(SB)   Goroutine creation
```

Once the goroutine is created and pushed onto the local queue of goroutines, it goes directly to the next instructions of the main function.

Exit

When a goroutine ends, Go must schedule another goroutine to not waste the CPU time. It will also keep the goroutine to reuse it later.

You can find more information about the recycling of the goroutine in my article "Go: How Does Go Recycle Goroutines?"

However, Go needs a way to be aware of the end of the goroutine. This control is during the creation of the goroutine. While creating the goroutine, Go sets the stack to a function named `goexit` before setting the program counter to the real function called by the goroutine. This trick forces the goroutine to call the function `goexit` after ending its work. The following program allows us to visualize it:

```go
func main() {
    var wg sync.WaitGroup
    wg.Add( delta: 1)

    go func() {
        var skip int
        for {
            _, file, line, ok := runtime.Caller(skip)
            if !ok {
                break
            }
            fmt.Printf( format: "%s:%d\n", file, line)
            skip++
        }

        wg.Done()
    }()
    wg.Wait()
}
```

The output will complete the stack trace:

```
/path/to/src/main.go:16
/usr/local/go/src/runtime/asm_amd64.s:1373
```

The file `asm_amd64` written in assembly contains this function:

```
1372    TEXT runtime·goexit(SB),NOSPLIT,$0-0
1373 ✓      BYTE    $0x90   // NOP
1374        CALL    runtime·goexit1(SB) // does not return
1375        // traceback from goexit1 must hit code range of goexit
1376        BYTE    $0x90   // NOP
```

Then, Go will switch to `g0` to schedule another goroutine.

It is also possible to stop the goroutine manually by calling `runtime.Goexit()`:

```
func main() {
    var wg sync.WaitGroup
    wg.Add( delta: 1)

    go func() {
        defer wg.Done()

        runtime.Goexit() // goroutine exits here

        println( args...: "never executed")
    }()
    wg.Wait()
}
```

This function will run the deferred functions first, then will call the same function seen previously when a goroutine exits.

Golang Go Goroutines Internals