Executing Other Programs in Go

CHRIS (MAC) MCENIRY

Chris (Mac) McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

If you have come to the Go world from bash or another shell language, one of the most critical tasks that you will be trying to replicate is calling out to other programs. Go has mechanisms in the standard library to accomplish this—the `os/exec` library.

When running an external program, you have to decide how to interact with this. These interactions tend to fall into several patterns:

1. Fire and Wait: Run another program, send its output to the terminal, and wait for it to finish.
2. Fire and Forget: Run another program, send its output to the terminal, and do not wait for it.
3. Pipe In: Feed data into the program.
4. Check Out: Check the output or exit code of the program.
5. Replace: Perform some setup, and then replace the current process with the other program.
6. Interact: Start another program and interact back and forth with it.

Each of these patterns is a combination of:

1. What to do with input for the other program?
2. What to do with the other program's output?
3. Do we need to block until the other program is done or not?

In this article, we're going to examine each of these interactions in turn with a focus on which patterns they use.

Note: These examples are very UNIX and bash focused. As such, the examples will only work on limited environments.

The code for these examples can be found at https://github.com/cmceniry/login/ in the exec directory. Each example is its own appropriately named subdirectory so that it can be executed directly with `go run $EXAMPLE`.

Fire and Wait

This is the simplest interaction with another process. In this pattern, the input and output are of little concern, but we do want to wait until the other program is complete. Its profile looks like:

1. Input: supply none (attaches automatically to `/dev/null` or equivalent)
2. Output: provide back to the attached terminal
3. Block till completion: yes

We begin much like any other Go program—the package declaration, imports, and our main func: the main library to include here is the standard library's `os` and `os/exec` components.

firenwait.go: setup.

```
package main

import (
    "os"
    "os/exec"
)

func main() {
```

To begin with the meat of our program, we first invoke the `exec.Command` func. This accepts the invocation of the other program as arguments. Go performs standard `PATH` resolution to find the program by name, but in our case, we're going to invoke the `/bin/ls` command. In addition, we pass `exec.Command` any arguments. For this example, we just want to list out the current directory's outputs.

As a result, we receive back an `*exec.Cmd` struct which will handle all interactions with our called program.

firenwait.go: command.

```
    c := exec.Command("/bin/ls", ".")
```

Since we want to display the output of the `ls` command, we need to connect the output of that command with our display. This is done by associating the `Stdout` member of our `*exec.Cmd` with the main `Stdout` from our current program. The main `Stdout` is available from the main `os` package.

Note: `Stdout`, and its accompanying `Stderr` for error output, is an `io.Writer` interface. Input is covered under `Stdin`, which is an `io.Reader` interface. If they are not specified by setting `Stdout` or `Stdin`, they default to `nil` and will be connected to the `/dev/null` equivalent. We'll explore using other items that satisfy the `Reader/Writer` interfaces later.

firenwait.go: connectoutput.

```
    c.Stdout = os.Stdout
```

With all of the initialization complete, we can `Run` our program. `Run` will block until the child process completes or fails. It returns an error if it is unable to run the other program or if the other program fails during execution (gets a non-zero exit code). For the example case, we `panic` for that, or exit normally otherwise.

firenwait.go: run.

```
    err := c.Run()
    if err != nil {
        panic(err)
    }
}
```

We can now run our example with `go run` and see the current directory. In this example, we are using `$GOPATH/src/github.com/cmceniry/login` as our starting point.

```
$ go run exec/firenwait/firenwait.go
README.md   exec      gofs      hardcode   useldap
```

Fire and Forget

The second example handles the case where we run a program but do not check for what happens to it. This follows the patterns for:

1. Input: supply none
2. Output: provide back to terminal
3. Block till completion: no

This is very similar to the first example. It includes the same libraries—plus `time` for the example. It creates the command the same way, and it associates the output in the same way. There are only two primary differences.

The first is the specific start of the command -- `c.Start()` instead of `c.Run()`. `Start` will begin the other process but will return as soon as it begins instead of waiting for it to complete. If there's an issue starting the other process—e.g., command is not found—then it will show up as the returned error to `Start`.

firenforget.go: start.

```
    err := c.Start()
```

The second is to reap the child when it exits. Although we're not doing anything with the output, we still need to handle the child when it exits. Otherwise, the child can hang around as a zombie process. It's not complete fire and forget—only mostly fire and forget.

firenforget.go: wait.

```
    go func() {
        err := c.Wait()
        if err != nil {
            panic(err)
        }
    }()
```

The last part is that we hold our program from finishing up for a couple of seconds. We want to make sure that our program exits after the other program exits. In most cases, there would be some other work that would be going on, so we simulate that with just a simple `Sleep`:

firenforget.go: work.

```
    // Do some other work...
    time.Sleep(2 * time.Second)
```

Pipe In

Our next example shows how to provide input to a program. As mentioned in the first example, `Stdin` is an `io.Reader`, so anything that satisfies that interface will work. In this example, we'll use the patterns from our first example—only "Input" is different:

1. Input: supplied
2. Output: provide back to terminal
3. Block till completion: yes

The goal of this example is to have the calculating program `dc` perform some arithmetic for us. We'll be using a `strings.Reader` to provide `dc` with data. With the following input, `dc` will calculate the sum of 1 plus 2, print the output, and quit.

```
1
2
+
p
q
```

The initialization is the same as previous programs, except for the addition of the `strings` package from the standard library.

As with the previous examples, we begin with getting an `exec.Cmd` struct. In this case, we invoke the `dc` command and supply no arguments.

pipein.go: command.

```
c := exec.Command("/usr/bin/dc")
```

Next, we connect the inputs and outputs. `strings.Reader` implements the `io.Reader` interface, so we can use it to send a static string in as our input. We connect this with the `Stdin` of our command. As before, we connect `Stdout` of our command with the existing terminal `Stdout`.

pipein.go: io.

```
c.Stdin = strings.NewReader("1\n2\n+\np\nq\n")
c.Stdout = os.Stdout
```

And now we can run `dc`.

pipein.go: run.

```
err := c.Run()
```

If all works out, we will see the sum as the result:

```
$ go run exec/pipein/pipein.go
3
```

Check Out

Normally, just running a command and expecting it to behave is wishful thinking. We can get some information if there's an issue starting the command, or with `Run` we can see whether the program exited with a non-zero exit code. However, sometimes it's important to know what that return code is or what the program returns as output.

In those cases, we need to check the `ProcessState` after our command runs. `ProcessState` is a very generic `struct` which mainly indicates whether the process is still running or not. For detailed information, it has a `Sys()` member method that returns an empty interface whose concrete implementation is very much operating system dependent. On UNIX, `Sys()` returns a `syscall.WaitStatus` that includes the detailed exit code that we're looking for.

In this example, we're going to run a command and check its exit code. It follows the pattern of:

1. Input: supply none
2. Output: discard except for the exit code
3. Block till completion: yes

The initialization is the same except that, in this case, we must include the `syscall` package of the standard library. We are even calling the command in the same way.

checkout.go: command.

```
c := exec.Command("/usr/bin/false")

err := c.Run()
```

Since we expect the failure to return an error, we must handle it. We check to see whether it is of the `exec.ExitError` type and handle that separately. Otherwise, we will panic on any other error, since that indicates something really unexpected happened, or exit normally on no error.

checkout.go: result.

```
switch err.(type) {
case *exec.ExitError:
   ws := c.ProcessState.Sys().(syscall.WaitStatus)
   fmt.Printf("Exited %d\n", ws.ExitStatus())
case nil:
   fmt.Printf("Exited normally\n")
default:
   panic(err)
}
```

If all goes well, we can see the expected result of an exit code of 1:

```
$ go run exec/checkout/checkout.go
Exited 1
```

You can see alternate behaviors by changing the command to execute. Try:

- `/usr/bin/true`
- `/usr/bin/notfound`

Replace

In the Replace interaction, we are largely using the Go program as a wrapper. The wrapper will perform some setup and then transfer control over to another program. Some examples of useful setups:

- Set environment variables—configuration parameters
- Set up file-system structures—working directory, lock files, etc.
- Check other dependencies—backend database or service—before starting up the application process

This follows the patterns:

1. Input: handed off
2. Output: handed off
3. Block till completion: no, handed off

Since process replacement is extremely operating system dependent, we're going to use the `syscall` package in the standard library—same as the previous example. This makes the program setup match the last exercise.

From there, we need to make any modifications as part of our wrapping action. In this example, we'll add a single environment variable.

replace.go: env.

```
env := append(
   os.Environ(),
   "USENIXLOGIN=true",
)
```

From there, instead of using the higher level `os/exec` package, we use the `syscall.Exec` function directly. For this example, we want to spawn a shell with the manipulated environment.

replace.go: handoff.

```
syscall.Exec("/bin/bash", []string{}, env)
```

For wrappers as simple as environment manipulations, that is the extent of it. We can now use the updated environment.

```
$ echo $USENIXLOGIN

$ go run exec/replace/replace.go
bash$ echo $USENIXLOGIN
true
```

Interact

The last example that we're going to take a look at involves interacting with the other program. This can be used if you need to programmatically interact with other command-line or terminal-based tools. Typically, you will be looking for data or errors and responding back into them.

Since this is before the process has exited, we're going to focus our time on manipulating the input and output of the process.

Specifically, in this example, we're going to:

- start with the letter "a",
- feed it into `cat`,
- read the output `cat` back out,
- append "b" to the output,
- feed that back into the same `cat` process, and
- repeat for "c", "d", and "e".

Each time through, we're going to build on the letters that have already been supplied, unless we're finally presented with the full string "abcde".

So far, we've been working with the `io.Reader` and `io.Writer` interfaces of `Stdin` and `Stdout`. To be able to provide the continuous feeds, Go provides a way to get pipes for each of these: `(*Cmd) StdinPipe()` and `(*Cmd) StdoutPipe()`. We're going to use these in this example to aid us.

For the start of our main section, we need to initialize our data and our command.

interact.go: vars.

```
feed := []string{"a", "b", "c", "d", "e", ""}
c := exec.Command("/bin/cat")
```

After that, we grab the pipes for `Stdin` and `Stdout`.

interact.go: stdin,stdout.

```
cin, err := c.StdinPipe()
cout, err := c.StdoutPipe()
```

We're going to rely on the `bufio` package of the standard library to more easily support the line and string manipulation that works well with `cat`. To do so, we need to wrap our `io.Reader` and `io.Writer` with `bufio.Scanner` and `bufio.Writer`, respectively.

interact.go: buffer.

```
bin := bufio.NewWriter(cin)
bout := bufio.NewScanner(cout)
```

With all of the prep work out of the way, we can get the ball rolling with `cat`. To do so, we need to prime the input with a newline and start `cat`.

interact.go: prime.

```
bin.WriteString("\n")
bin.Flush()
c.Start()
```

Next, we're going to iterate through our data. For each piece, we want to gather the `cat` output and then write back the output with our addition.

interact.go: addnprint.

```
for _, addition := range feed {
   if !bout.Scan() {
      panic("ended early")
   }
   if bout.Text() != "" {
      fmt.Printf("%s\n", bout.Text())
   }
   bin.WriteString(bout.Text() + addition + "\n")
   bin.Flush()
}
```

At the end, we want to clean up. Much like with the Fire and Forget example, we still need to wait for the other process to finish. However, since `cat` will not finish until its input is finished, we must first close that.

interact.go: cleanup.

```
cin.Close()
c.Wait()
```

Now, we can run our program much like the others, and we should see our five-letter output:

```
$ go run exec/interact/interact.go
a
ab
abc
abcd
abcde
```

Conclusion

One of the most basic functions of any script is to build on other programs. It is crucial to be able to both trigger other programs with various inputs and to respond to the results of those other programs. Although the invocation of these other programs has a few more steps in Go versus traditional scripting languages, Go allows you to more readily tap into a large corpus of software for processing inputs and outputs.

I hope this article has given you confidence to use Go when it is appropriate to handle these process interactions, and some ideas for how to readily do so.