Building And Compiling Go Applications

Building And Compiling

Go is a compiled language. This means that before we can run a Go application it must be compiled to an executable capable of running on the current platform.

Running Go Code

When in development it is common to use `go run some-file.go` to run Go applications.

> *But you said we have to compile applications before we can run them?*

When we use `go run` to run a Go application it silently creates a binary, executes the binary, and then deletes the binary.

This makes the process of running code locally easy for developers.

Why Not Just Use `go Run`?

While using `go run` is great for local development, it is not ideal for applications that are to be given to other people.

`go run` has the following requirements:

- Needs Go installed, and potentially the same version
- Requires all dependencies be present
- Doesn't allow for custom hooks, as we'll see later

Building A Binary

Go ships with some pretty powerful build tools that can be accessed with the `go build` command.

When building a binary with the `go build` command, Go will build a fully bundled, statically linked, executable.

Building A Binary

```
$GOPATH/src/simple/main.go
```

```go
package main

import "fmt"

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

```
$ go build
```

```
-rw-r--r--@ 1 markbates  staff    74B May 12 15:04 main.go
-rwxr-xr-x@ 1 markbates  staff   1.6M May 12 15:06 simple*
```

Building A Binary

By default `go build` will name the executable after the current folder and place it in the current folder.

We can change this behavior with the `-o` flag.

```
$ go build -o bin/hello
```

```
drwxr-xr-x@ 3 markbates  staff   102B May 12 15:09 bin/
-rw-r--r--@ 1 markbates  staff    74B May 12 15:04 main.go

./bin:
-rwxr-xr-x@ 1 markbates  staff   1.6M May 12 15:09 hello*
```

Installing A Binary

When we setup our `GOPATH` we are told to add `$GOPATH/bin` to our `$PATH`. The reason for this is this is where the `go install` command places binaries.

We can use `go install` in exactly the same way as `go build`, the difference being that instead of placing a binary in the current directory it will build it and place it in the `$GOPATH/bin` directory.

Build Tags

Using tags at build time allows us to control how are binary is compiled, and what it does. For example, complile one binary for development and another for production; each with different levels of logging.

We can also control behavior based on platform, but we'll look at that later.

Custom Build Tags

Let's create a simple program that prints out a greeting.

```go
// main.go
package main

import "fmt"

func main() {
    fmt.Println(greeting)
}
```

Custom Build Tags

Now let's create two other files, `happy.go` and `sad.go`, that will contain the various greetings we wish to display.

```go
package main

import "fmt"

func init() {
    fmt.Println("happy.go")
}

var greeting = "It's so great to see you!!"
```

```go
package main

import "fmt"

func init() {
    fmt.Println("sad.go")
}

var greeting = "I'm so sorry to see you. :("
```

Custom Build Tags

Attempting to run `go build` on our application will result in an error.

```
./sad.go:10: greeting redeclared in this block
    previous declaration at ./happy.go:10
```

Using Build Tags

Using build tags we can turn on/off each file when building the binary.

Build flags are placed at the top of a `.go` file, above the `package` declaration.

```go
// +build tag1 tag2 tag3

package foo
```

Using Build Tags

We can update both the `happy.go` and the `sad.go` files to using the `happy` tag for `happy.go` or `!happy` for `sad.go`.

```go
// +build happy

// happy.go
package main
```

The `happy` build tag is equivalent of saying "only build this file when there is a `happy` tag".

```go
// +build !happy

// sad.go
package main
```

The `!happy` build tag is the equivalent of saying "don't build this file when there is a `happy` tag".

Using Build Tags

If we were to run the default `go build` command the application would compile. When run it would print out the following:

```
sad.go
I'm so sorry to see you. :(
```

Since we did not specify the `happy` flag the `happy.go` file was not compiled, but the `sad.go` one was.

Using Build Tags

To compile the application with the `happy` tag we need to change our `go build` command.

```
$ go build -tags happy
```

Now when run we get the happy greeting.

```
happy.go
It's so great to see you!!
```

Building Cross Platform Binaries

The Go build tools, by default, build binaries for the current platform you are on. However, it is possible to compile binaries for multiple different platforms using just the standard Go tools and a couple of environment variables and/or build tags.

Building With Environment Variables

The Go tools use two different environment variables to decide which binary to build.

$GOOS

The `$GOOS` variable controls which operating system to build for. Examples are `darwin`, `linux`, `windows`, etc..

$GOARCH

The `$GOARCH` variable controls which architecture to build for. Examples are `386`, `arm`, `amd64`, etc…

Available Platforms

```
$GOOS         $GOARCH       $GOOS       $GOARCH
android       arm           netbsd      386
darwin        386           netbsd      amd64
darwin        amd64         netbsd      arm
darwin        arm           openbsd     386
darwin        arm64         openbsd     arm64
dragonfly     amd64         openbsd     arm
freebsd       386           plan9       386
freebsd       amd64         plan9       amd64
freebsd       arm           solaris     amd64
linux         386           windows     386
linux         amd64         windows     amd64
linux         arm
linux         arm64
linux         ppc64
linux         ppc64le
linux         mips
linux         mipsle
linux         mips64
linux         mips64le
```

*As of Go 1.8

Building Cross Platform Binaries

```
$ GOOS=windows go build
```

```
-rw-r--r--@ 1 markbates  staff    74B May 12 15:04 main.go
-rwxr-xr-x@ 1 markbates  staff   1.6M May 12 16:18 simple.exe*
```

Cross Platform Build Tags

Using environment variables lets us decide which platform and architecture we should build a binary for. But what about compiling different code for different platforms?

We can accomplish this using the build tags that we already learned about.

Cross Platform Build Tags

Here we mark the `happy.go` file with `darwin` to indicate to the compiler that it should compile this file when compiling for OS X systems.

```
// +build darwin

// happy.go
package main
```

The `sad.go` file should be compiled when building for linux.

```
// +build linux

// sad.go
package main
```

Cross Platform Build Tags

The cross platform build tags are automatically linked to `$GOOS` and `$GOARCH`.

This means that, unlike our custom build tags, we *don't* need to pass them in when compiling the binary.

Cross Platform File Names

The Go tools also allow us to control which files we want to build for which platforms using file names instead of build tags.

```
-rw-r--r--@ 1 markbates  staff   136B May 12 17:36 greeting_darwin.go
-rw-r--r--@ 1 markbates  staff   134B May 12 17:35 greeting_linux.go
-rw-r--r--@ 1 markbates  staff    78B May 12 17:35 main.go
```

Modifying Built Source With `ldflags`

It is quite common for applications to have to information, such as a version number, or an "environment" such as `development` or `testing`.

When building a binary for distribution we can use the `-ldflags` flag to burn that information into the built binary.

Modifying Built Source With `ldflags`

```go
package main

import "fmt"

var version = "development"

func main() {
    fmt.Printf("Version: %s\n", version)
}
```

```
$ go build -ldflags "-X main.version=1.0.0" -o bin/hello
$ ./hello
Version: 1.0.0
```

Real Life Example

The Buffalo web framework, http://gobuffalo.io uses `-tags` and `-ldflags` when building binaries:

```
$ go build -tags nosqlite -o bin/web \
  -ldflags -X main.version=4b6c176 \
  -X main.buildTime="2017-05-12T17:59:34-04:00"
```

```
./bin/web version
web version 4b6c176 ("2017-05-12T17:59:34-04:00")
```

This application now has the Git SHA and the time it was built embedded in the binary, making it easy to diagose exactly what, and when, the binary was built.

Exercises

Play with build binaries.

- Compile for different platforms. What happens if you try to run the windows exe on a Mac?
- Insert information into a binary using `-ldflags`
- Turn on/off functionality using build tags