GO 1.8 RELEASE PARTY

NEW FEATURE: PLUGINS

BY: DANIEL TOEBE

1 WHAT IS THE PLUGIN FEATURE?

1.1 THE PLUGIN FEATURE CREATES .SO FILES

- The new `Plugin` feature is a way to create .so (shared object) file for dynamically linking libs in Go

1.2 OK.. WHAT IS A SHARED OBJECT FILE?

- A .so file is a precompiled binary that allows you to call functions inside it dynamically at your executable's runtime

1.3 WHAT DOES THAT REALLY MEAN?

- Now instead of only having `go get` to download a package source on your build machine and compiling that package in your executable
  - You now can compile your package as a plugin binary distribute it separately and call it in your executable, and it will be loaded at runtime.

2 THAT SOUNDS NEAT, BUT WHAT IS THE BENEFIT?

2.1 FIRST

- Your executable may be smaller when you distribute it.

2.2 SECOND

- You can version the .so file. Making sure your executable calls the right version which can prevent runtime errors
  - Now you can create a `plugin.so.0.1` instead of a regular import and have many executables depend on that. What is you have added more features? Instead of updating all the dependent executables, you can just create a new `plugin.so.0.2`, call that in all the new executables you create and v0.1, and v0.2 can live side by side in your file system.

2.3 FINALLY

- Finally your .so files don't need to live in your $GOPATH at all! It can live anywhere in your file system and you can dynamically link in your executable.

2.4 WHAT DO I MEAN BY DYNAMICALLY LINKING?

- Basically you can place your .so file where ever you want, but if you have to make sure that it is places in the same place in the file system from server to server.
  - Examples: `/opt/libs` or something like `/usr/libs`
  - Or you can just have an environment variable created when you distribute your .so file and use that in calling your shared object in your executable using `os.Getenv()`
  - Another option is passing the path to the .so file through os.Args, or as a flag

3 HOW DO WE CREATE A SHARED OBJECT FILE?

3.1 FIST WE NEED TO HAVE THE PREREQUISITE IMPORT

```
import "C"
```

While we are not calling any C functions directly this is needed on all Plugins

3.2 NOW WE WRITE OUR FUNCTIONS

```go
// plugin.go
package main

import "C"

func Greet() string {
    return "Hello Phx Gophers"
}
```

3.3 NOW WE COMPILE IT

```
$ go build --buildmode=plugin plugin.go
```

- This produces a `plugin.so` file
- Note the use of `--buildmode=plugin`
  - The buildmode flag was introduced in Go 1.5, but the plugin option was introduced in Go 1.8
  - buildmode description

3.4 IF YOU WANT TO SEE WHAT THE FILE LOOKS LIKE...

```
$ file plugin.so
```

The output should look like:

```
plugin.so: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
dynamically linked, BuildID[sha1]=fc2ec03a4fca7e2b78aa1cb8bd82a9e1c99a0554,
not stripped
```

Here you can see it is an ELF type binary used by Linux, and importantly not stripped. The not stripped means it keeps all symbols in your .so file. To dive a little deep this is how your executable can call the functions in the .so file, by referencing the symbols.

4 NOW LETS WRITE OUR EXECUTABLE

4.1 FIST WE NEED TO IMPORT THE PLUGIN PACKAGE FROM THE 1.8 STANDARD LIB

```
import "plugin"
```

This is needed with any .go file that will call a .so file

4.2 NOW FOR THE WHOLE FILE

```go
package main

import (
    "fmt"
    "plugin"
)

func main() {
    p, err := plugin.Open("plugin.so")
    if err != nil {//Do something}
    g, err := p.Lookup("Greet")
    if err != nil {//Do something}
    greet := g.(func() string)
    fmt.Println(greet())
}
```

4.3 NOTICE THE P.LOOKUP()

- "Greet" is the name of the function we created in `plugin.so` file, and called with a string because that is the name of the symbol. An error returned on the function is because since the .so file is loaded at runtime there is no guarantee that the symbol exists

4.4 NOTICE THE ASSERTION WHEN CREATING GREET

- Since the .so file is loaded at runtime the compiler can't know what type you are trying to use

4.5 THE OUTPUT

We run...

```
$ go run main.go
```

... and we get

```
Hello Phx Gophers
```

4.6 THATS THE BASICS

5 THAT SOUNDS AWESOME BUT...

5.1 SOME CURRENT LIMITATIONS

- Right now you can only use .so files compiled from the go compiler. Meaning you can’t currently use .so files created with C
- You can only create and use the plugin system on Linux, so not on OSX or Windows. This is because the output of the `buildmode=plugin` creates an ELF binary and not compatible on OSX or Windows.
- Your executable must be compiled with the same version toolchain as the .so file. Right now that isn't an issue since all of this is in the 1.8 release, but may be an issue when 1.8.x is released