Introduction to Programming in Go

13. Go Packages

Module Topics

1. Packages
2. Workspace Organization
3. Initialization
4. Using go get
5. Vendoring

Packages

Go Packages

1. Packages are located by their import path.

2. Location information is never "inside" a package file, packages files are located exclusively by the go tool.

3. The last part of a package import path is the local name for the package and used as a prefix .

4. Custom packages should use a naming protocol like that for XML namespaces to ensure unique names. For example

```
import "capitalone.com/devteam6/roberto/utilities"
```

5. Public symbols are capitalized, all others are private to the package.

Workspace Organization

The "Go" Way

1. All of a programmers's Go code is in a single workspace.
2. A workspace contains many version controlled repositories.
3. Each repository contains one or more packages.
4. A package has one or more Go source files in a single directory.
5. The path to a package's directory determines its import path.

Workspace Organization

1. A workspace is a directory hierarchy with three directories at its root:

   src: contains all of the Go source files,

   pkg: contains package objects, and

   bin: contains executable commands.

2. The pkg objects are compiled files.

3. The pkg directory contains subdirectories based on the compilation target architecture.

Adding a pkg to "Hello world"

```
// Example 13-01 Stringutils package

package stringutils

func Reverse(s string) string {
   r := []rune(s)
   for i,j := 0, len(r)-1; i < len(r)/2; i,j = i+1,j-1
{         r[i], r[j] = r[j], r[i]
   }
return string(r)}
}
```

Using the stringutil package

```
// Example 13-01 Hello.go

package main

import (
   "fmt"
   "stringutil"
)

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

The Project Structure - Build

```
src\
   hello\
      hello.go
   stringutil\
      stringutil.go
pkg\
bin\
```

```
[Module13]$ cd $GOPATH/src/hello
[stringutil] go build hello
[stringutil] ls
hello.go hello
```

The Project Structure - Install

```
src\
   hello\
      hello.go
   stringutil\
      stringutil.go
pkg\
   linux_amd64\
      stringutil.a
bin\
   hello
```

```
[Module13]$ cd $GOPATH/src/hello
[stringutil] go install hello
[stringutil] ls
hello.go hello
```

Package Aliases

1. Package prefixes are the last part of the import path.

2. This can lead to ambiguities.

3. We can define local aliases for the package prefixes.

4. Consider adding a new stringutils package with a different import path but identical local prefix to the Hello World example.

5. The new package has a different version of the Reverse() method which also converts a reversed string to uppercase.

Adding another pkg to "Hello world"

```
// Example 13-03 utils/stringutils package

package stringutils
import "strings"


func Reverse(s string) string {
   r := []rune(s)
   for i,j := 0, len(r)-1; i < len(r)/2; i,j = i+1,j-1
{         r[i], r[j] = r[j], r[i]
   }
return strings.Toupper(string(r))
}
```

Using Both the stringutils Packages

```
// Example 13-04 Hello.go with ambiguities

package main

import (
   "fmt"
   "stringutil"
   "util/stringutil"
)

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

```
[hello]$ go build hello
./hello.go:7: stringutil redeclared as imported package name
   previous declaration at ./hello.go:6
Error: process exited with code 2.
```

Using Both the stringutils Packages

```
// Example 13-05 Hello.go with local aliases

package main

import (
   "fmt"
   s1 "stringutil"
   s2 "util/stringutil"
)

func main() {
   fmt.Println(s1.Reverse("Hello World!"))
   fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
!dlroW olleH
!DLROW OLLEH
```

Package Initialization

Package Initialization

1. In each file there can be one or more functions named init().

2. All init() functions are executed after all of the package variable have been initialized; and

3. All of the init() functions from the imported packages have run.

4. The purpose of the init() functions verify and repair program state before execution.

Using Both the stringutils Packages

```
// Example 13-06 Hello.go with init()

import (
   "fmt"
   s1 "stringutil"
   s2 "util/stringutil"
)
func init() {
   fmt.Println("Main init 1")
}
func init() {
   fmt.Println("Main init 2")
}
func main() {
   fmt.Println(s1.Reverse("Hello World!"))
   fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

Init() in a Dependency

```
// Example 13-08 utils/stringutils package init()

package stringutils
import  "strings"
import  "fmt"

func init() {
   fmt.Println("util/stringutil init")
}


func Reverse(s string) string {
   r := []rune(s)
   for i,j := 0, len(r)-1; i < len(r)/2; i,j = i+1,j-1
{         r[i], r[j] = r[j], r[i]
   }
return strings.Toupper(string(r))
}
```

Init() in a Dependency

```
// Example 13-07 Hello.go with init()

import (
   "fmt"
   s1 "stringutil"
   s2 "util/stringutil"
)
func init() {
   fmt.Println("Main init 1")
}
func init() {
   fmt.Println("Main init 2")
}
func main() {
   fmt.Println(s1.Reverse("Hello World!"))
   fmt.Println(s2.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlroW olleH
!DLROW OLLEH
```

Blank Alias

1. If we want to force a dependency init() to execute but we don't want to import any symbols we use "_" for the local alias.

2. This disables the compiler from generating an error that we have a package import statement and no symbols imported from that package.

Blank Alias

```
// Example 13-08 Blank Alias

import (
   "fmt"
   "stringutil"
   _ "util/stringutil"
)
func init() {
   fmt.Println("Main init 1")
}
func init() {
   fmt.Println("Main init 2")
}
func main() {
   fmt.Println(stringutil.Reverse("Hello World!"))
}
```

```
[hello]$ go build hello
./hello
util/stringutil init
Main init 1
Main init 2
!dlroW olleH
```

Go Get

Getting Remote Packages

1. Allows Go to get packages from remote locations specific by a URL

2. Works with most repository and version control systems and has a rich range of options.

3. Copies the source code to the current workspace src directory, then does the equivalent of a go install on the downloaded source.

4. In the example, we get a utility called golint from a github repository starting with an empty workspace

Workspace and go get Command

```
src\
pkg\
bin\
```

```
[Module14]$ go get github.com/golang/lint/golint
```

```
bin\
   golint
pkg\
   linux_amd64\
      github.com\
         golang\
            lint.a
      golang.org\
         x\
            tools\
               go\
                  gcimporter15.a
```

Workspace and go get Command

```
src\
   github.com\
      golang\
         lint\
   golang.org\
         x\
```

Vendor Management

Vendoring

1. The drawback to the Go workspace organization is that there is only one version of a package available.

2. Problem 1: Two developers in the same workspace are using different versions of a package.

3. Problem 2: Two different projects requires different versions of a package.

4. Go is designed to work with local copies of remote packages so that changes or missing remote repositories do not break local builds.

5. This is called "vendoring."

Vendoring

1. If two different versions of a packate are needed, the import paths can be rewritten so that they are both installed but with different paths.

2. Not an effective solution since it involves a lot of low level work.

3. Too easy to make errors manually that will break a build.

Vendoring Third Party Tools

1. Third party tools like "godep" can be used to take snapshots of versions used in a build.

2. For example using "godep save" creates a json file of version information.

3. Then using "godep go build" feeds the correct version information into the build utility.

4. The use of third party tools though is not what was intended for Go.

A godep Snapshot

```
"ImportPath": "github.com/golang/prog",
    "GoVersion": "go1.6",
    "Deps": [
            {
            "ImportPath": "github.com/golang/examples/ex1",
            "Rev": "e0e1b550d545d9be0446ce324babcb16f09270f5"
            },
            {
             "ImportPath": "ithub.com/golang/examples/ex2",
             "Rev": "a1577bd3870218dc30725a7cf4655e9917e3751b"
                },
)
```

The Vendor Folder

1. The latest version of Go has introduced the idea of a vendor folder.

2. If there is a vendor folder in your src tree, then anything under the folder is an external vendor dependency.

3. When building the project, the go tools will check under the vendor folder first for the package rather than look outside your project.

4. Two different versions of a package can be kept in different projects by placing them in different vendor folders in each project.

5. Most of the third party vendoring tools support this feature.

The Vendor Folder

```
src\
   hello\
      vendor\
         stringutil\
            version 1 code
   bye\
      vendor\
         stringutil\
            version 2 code
   stringutil\
      version3 code
   whatever\
pkg\
bin\
```

The Vendor Folder

1. The hello project will use version 1 of stringutil package.

2. The bye project will use version 2 of stringutil package.

3. The whatever project, and any other projects, will use version 3 of stringutil package

Lab 13: Packages