

Packages and Modules

Overview



Packages

Modules

Creating Packages and Modules

PACKAGES



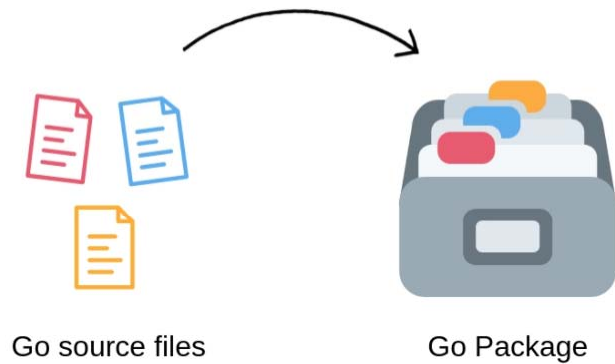
Introduction to Packages

Go was designed to encourage good software engineering practices. One of the guiding principles of high-quality software is the DRY principle - Don't Repeat Yourself, which basically means that you should never write the same code twice. You should reuse and build upon existing code as much as possible.

[Functions](#) are the most basic building blocks that allow code reuse. **Packages** are the next step into code reusability. They help you organize related Go source files together into a single unit, making them modular, reusable, and maintainable.

Go Package

In the most basic terms, A package is nothing but a directory inside your Go workspace containing one or more Go source files, or other Go packages.



Every Go source file belongs to a package. To declare a source file to be part of a package, we use the following syntax -

```
package <packagename>
```

The above package declaration must be the first line of code in your Go source file. All the functions, types, and variables defined in your Go source file become part of the declared package.

You can choose to export a member defined in your package to outside packages, or keep them private to the same package. Other packages can import and reuse the functions or types that are exported from your package.

Let's see an example

Almost all the code that we have seen so far

```
import "fmt"
```

`fmt` is a core library package that contains functionalities related to formatting and printing output or reading input from various I/O sources. It exports functions like `Println()`, `Printf()`, `Scanf()` etc, for other packages to reuse.

Packaging functionalities in this way has the following benefits -

- It reduces naming conflicts. You can have the same function names in different packages. This keeps our function names short and concise.
- It organizes related code together so that it is easier to find the code you want to reuse.
- It speeds up the compilation process by only requiring recompilation of smaller parts of the program that has actually changed. Although we use the `fmt` package, we don't need to recompile it every time we change our program.

The `main` package and `main()` function

Go programs start running in the `main` package. It is a special package that is used with programs that are meant to be executable.

By convention, Executable programs (the ones with the `main` package) are called *Commands*. Others are called simply *Packages*.

The `main()` function is a special function that is the entry point of an executable program. Let's see an example of an executable program in Go -

```
// Package declaration
package main
```

```
// Package declaration
package main

// Importing packages
import (
    "fmt"
    "time"
    "math"
    "math/rand"
)

func main() {
    // Finding the Max of two numbers
    fmt.Println(math.Max(73.15, 92.46))

    // Calculate the square root of a number
    fmt.Println(math.Sqrt(225))

    // Printing the value of  $\pi$ 
    fmt.Println(math.Pi)
```

```
// Epoch time in milliseconds
epoch := time.Now().Unix()
fmt.Println(epoch)

// Generating a random integer between 0 to 100
rand.Seed(epoch)
fmt.Println(rand.Intn(100))
}
```

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

```
# Output
92.46
15
3.141592653589793
1538045386
40
```

Go's convention is that - *the package name is the same as the last element of the import path*. For example, the name of the package imported as `math/rand` is `rand`. It is imported with path `math/rand` because it is nested inside the `math` package as a subdirectory.

Importing Packages

There are two ways to import packages in Go -

```
// Multiple import statements
import "fmt"
import "time"
import "math"
import "math/rand"
```

```
// Factored import statements
import (
    "fmt"
    "time"
    "math"
    "math/rand"
)
```

When you import a package, you can only access its exported names.

```
package main

import (
    "fmt"
    "math"
)

func main() {
    // MaxInt64 is an exported name
    fmt.Println("Max value of int64: ", int64(math.MaxInt64))

    // Phi is an exported name
    fmt.Println("Value of Phi ( $\phi$ ): ", math.Phi)

    // pi starts with a small letter, so it is not exported
    fmt.Println("Value of Pi ( $\pi$ ): ", math.pi)
}
```

Output

```
./exported_names.go:16:38: cannot refer to unexported name math.pi
./exported_names.go:16:38: undefined: math.pi
```

Creating Packages and Modules

MODULES



Introduction to Modules

A *module* is a collection of packages that are released, versioned, and distributed together. Modules may be downloaded directly from version control repositories or from module proxy servers.

A module is identified by a [module path](#), which is declared in a `go.mod` file, together with information about the module's dependencies. The *module root directory* is the directory that contains the `go.mod` file. The *main module* is the module containing the directory where the `go` command is invoked.

Each *package* within a module is a collection of source files in the same directory that are compiled together. A *package path* is the module path joined with the subdirectory containing the package (relative to the module root). For example, the module `"golang.org/x/net"` contains a package in the directory `"html"`. That package's path is `"golang.org/x/net/html"`.

Creating and managing custom Packages

Until now, We have only written code in the `main` package and used functionalities imported from Go's core library packages.

Let's create a sample Go project that has multiple custom packages with a bunch of source code files and see how the same concept of package declaration, imports, and exports apply to custom packages as well.

Fire up your terminal and create a directory for our Go project:

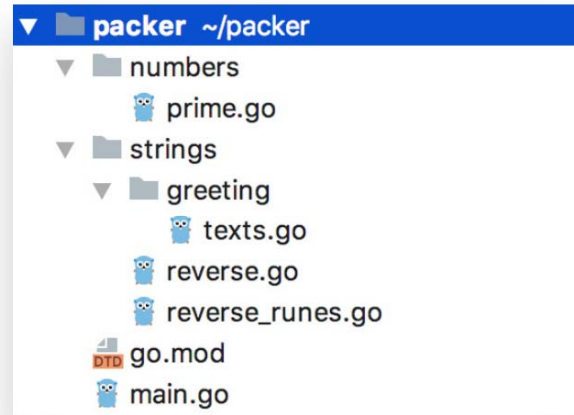
```
$ mkdir packer
```

Next, we'll create a [Go module](#) and make the project directory the root of the module.

Let's initialize a Go module by typing the following commands:

```
1 $ cd packer
2 $ go mod init github.com/harbudsan/packer
```

Let's now create some source files and place them in different packages inside our project. The following image displays all the packages and the source files:



numbers/prime.go

```
package numbers

import "math"

// Checks if a number is prime or not
func IsPrime(num int) bool {
    for i := 2; i <= int(math.Floor(math.Sqrt(float64(num)))); i++ {
        if num%i == 0 {
            return false
        }
    }
    return num > 1
}
```



```
strings > go reverse.go > ...
```

```
1 package strings
2
3 // Reverses a string
4 func Reverse(s string) string {
5     runes := []rune(s)
6     reversedRunes := reverseRunes(runes)
7     return string(reversedRunes)
8 }
```

strings/reverse_runes.go



```
package strings

// Reverses an array of runes
// This function is not exported (It is only visible inside the `strings` package)
func reverseRunes(r []rune) []rune {
    for i, j := 0, len(r)-1; i < j; i, j = i+1, j-1 {
        r[i], r[j] = r[j], r[i]
    }
    return r
}
```

strings > greeting >  texts.go >  WelcomeText

```
1 // Nested Package
2 package greeting
3
4 const (
5     WelcomeText = "Hello, World to Golang"
6     MorningText = "Good Morning"
7     EveningText = "Good Evening"
8 )
9
```

main.go (The main package: entry point of our program)

```
 main.go >  main
1 package main
2
3 import (
4     "fmt"
5     str "strings" // Package Alias
6
7     "github.com/harbudsan/packer/numbers"
8     "github.com/harbudsan/packer/strings"
9     "github.com/harbudsan/packer/strings/greeting" // Importing a nested package
10 )
11
12 func main() {
13     fmt.Println(numbers.IsPrime(19))
14
15     fmt.Println(greeting.WelcomeText)
16
17     fmt.Println(strings.Reverse("James Bond"))
18
19     fmt.Println(str.Count("Go is Awesome. I love Go", "Go"))
20 }
21
```

```
# Building the Go module
$ go build
```

The above command will produce an executable binary. Let's execute the binary file to run the program:

```
# Running the executable binary
$ ./packer
```

```
true
Hello, World to Golang
dnoB semaJ
2
```

Adding 3rd party Packages

Adding 3rd party packages to your project is very easy with Go modules. You can just import the package to any of the source files in your project, and the next time you build/run the project, Go automatically downloads it for you -

```
package main

import (
    "fmt"
    "rsc.io/quote"
)

func main() {
    fmt.Println(quote.Go())
}
```

```
$ go run main.go
go: finding rsc.io/quote v1.5.2
go: downloading rsc.io/quote v1.5.2
go: extracting rsc.io/quote v1.5.2
go: downloading rsc.io/sampler v1.3.0
go: extracting rsc.io/sampler v1.3.0
go: downloading golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go: extracting golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
go: finding rsc.io/sampler v1.3.0
go: finding golang.org/x/text v0.0.0-20170915032832-14c0d48ead0c
```

Don't communicate by sharing memory, share memory by communicating.

Go will also add this new dependency to the `go.mod` file.

Manually installing packages

You can use `go get` command to download 3rd party packages from remote repositories.

```
$ go get -u github.com/jinzhu/gorm
```

The above command fetches the `gorm` package from Github and adds it as a dependency to your `go.mod` file.

That's it. You can now import and use the above package in your program like this -

```
import "github.com/jinzhu/gorm"
```

Summary



Packages

Modules