



Go Modules

Dependency Management



kingkunte_

Posted on 12 Mar 2023 • Updated on 25 May



14



1



1



1



1

Go Modules: A Beginner's Guide.

#beginners #motivation #programming #go

Go Modules is a new approach to managing and organizing dependencies in Go projects introduced in Go 1.11 and became the default in Go 1.13. If you're new to Go development, a "**dependency**" is a piece of code your project requires to function, similar to a building block. Go Modules makes it simple to add new modules to your project, upgrade them to newer versions, and remove any that are no longer required. If you're new to Go Modules and want to understand the fundamentals, this beginner's tutorial will lead you through all the steps you need to know.

Go Modules is a tool for managing dependencies in Go projects that is crucial for any Go developer. Yet, if you're new to Go, it can be challenging to know where to begin. That is why I produced this Go Modules beginner's guide. In this post, we'll go through the fundamentals of:

1. Creating a new module:

A module is a collection of Go packages that are kept in a file tree that has a 'go.mod' file at its root. In this phase, you'll learn how to add a new module to your Go project.

2. Adding a dependency:

A dependency is a piece of code your project requires to function. In this step, we'll teach you how to use Go Modules to add a new dependency to your project.

3. Upgrading dependencies:

Dependencies may be updated to provide bug patches, performance enhancements, and new features. In this phase, we'll teach you how to use Go Modules to upgrade your dependencies to more recent versions.

4. Adding a dependency on a new major version:

A new dependency version may have significant modifications incompatible with the old version. In this step, we'll teach you how to upgrade a dependency to a new major version.

5. Upgrading a dependency to a new major version:

When you upgrade a dependency to a new major version, you may need to change your code to ensure compatibility. In this step, we'll teach you how to upgrade a dependency to a new major version.

6. Removing unused dependencies:

You might not need a dependency on your project in the long run. In this phase, we'll teach you how to use Go Modules to remove unused dependencies.

By the end of this beginner's guide, you'll grasp the fundamentals of Go Modules and how to use them to manage dependencies in your Go projects.

We'll now break everything down step by step with detailed explanations.

Creating and testing a new Go Module:

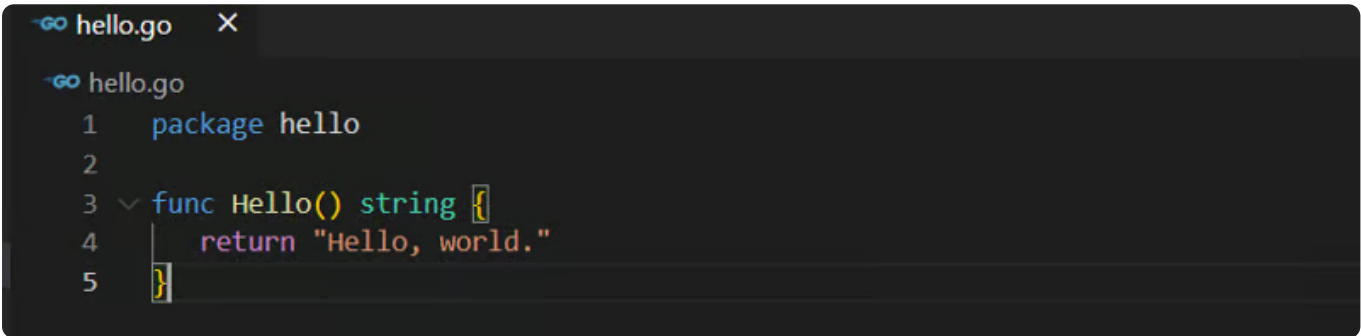
In the Go programming language, a module is a collection of related packages that can be versioned and shared with other developers. The modules' packages are organized in a directory hierarchy, and the module itself is defined by a file called `go.mod` in the module's root directory. Go modules improve the experience of using third-party code by allowing developers to specify the versions of the packages they depend on.

Now let's say you want to create a new module in the Go programming language. Here's a step-by-step guide to creating and testing a new module:

- Create a new directory on your computer outside the `$GOPATH/src` directory outside the `$GOPATH/src` directory on your computer `hello.go`
- In the command prompt or terminal, navigate to your new directory by running the following command:

```
$cd /path/to/your/new/directory
```

- Inside the new directory, create a new file called **hello.go** and add the following code:



```
hello.go X
hello.go
1 package hello
2
3 func Hello() string {
4     return "Hello, world."
5 }
```

_ This code creates a function that returns the string "Hello, world." when called._

- Now, we need to test the Hello function. Create a new file called `hello_test.go` in the same directory as `hello.go` and add the following code:



```
hello_test.go X
hello_test.go
1 package hello
2
3 import "testing"
4
5 func TestHello(t *testing.T) {
6     want := "Hello, world."
7     if got := Hello(); got != want {
8         t.Errorf("Hello() = %q, want %q", got, want)
9     }
10 }
```

This code tests the `Hello` function by comparing the actual output of the function with the expected output, which is the string `Hello, world.`

- At this point, we have created a package but not a module. To create a module, we need to run the `go mod init` command. Run the following command:

```
go mod init example.com/hello
```

This command creates a new module in the current directory and generates a `go.mod` file specifying the module's name and dependencies.

- Run the `go test` command to test the module:

go test

This command will run the test we created in step 4 and output a summary of the results. If the test passes, you should see a message that says `> "PASS"<`.

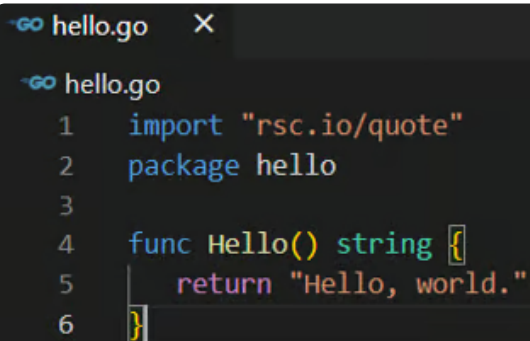
- If you open the `go.mod` file, you'll see that it contains the name of the module we just created:

```
module example.com/hello

go 1.16
```

This file also specifies the version of Go that we're using (in this case, version 1.16).

- Now, let's add a dependency to our module. We'll use the `rsc.io/quote` package, which provides a collection of famous quotes. Open the `hello.go` file and add the following line at the beginning of the file:



```
hello.go X
hello.go
1 import "rsc.io/quote"
2 package hello
3
4 func Hello() string {
5     return "Hello, world."
6 }
```

This line imports the quote package into our module.

- We can now modify the `Hello` function to return a quote instead of the string `"Hello, world."`. Change the `Hello` function to look like this:

```
hello.go X
hello.go
1  package main
2
3  import (
4      "rsc.io/quote"
5      "rsc.io/sampler"
6  )
7
8  func main() {
9      func Hello() string {
10         return quote.Hello()
11     }
12 }
```

This code returns a random quote from the quote package when the Hello function is called.

- Rerun the go test command to test the updated module:

go test

This command will download the `rsc.io/quote` package and its dependencies and then run the test we created in step 4 again. If the test passes, you should see a message that says "PASS"

And that's it! You've now created and tested a new module in Go. I hope you enjoyed that one. Coding is about having fun, and I want to make it as simple and easy to comprehend as possible. Let's have fun with the next part.

Adding and testing new dependency in Go.

When we use someone else's code in our program, we call it a dependency. The goal of Go modules is to make managing dependencies in our programs easy.

In Go, we can import other packages in our code. To add a new dependency, we need to import the package and then use it in our program. Here is a breakdown of the steps involved:

- Create a new Go project directory

```
$ mkdir hello && cd hello
```

- Initialize the project with Go modules

```
$ go mod init example.com/hello
```

- Create a new Go file called **hello.go** with the following contents:

```
hello.go X
hello.go
1 package main
2
3 import "rsc.io/quote"
4
5 func main() {
6     println(quote.Hello())
7 }
```

- Run the code to verify it works:

```
$ go run hello.go
```

```
Hello, world.
```

- Add a new dependency to the project

```
$ go get rsc.io/sampler
```

- Update hello.go to use the new dependency:

```
hello.go X
hello.go
1 package main
2
3 import (
4     "rsc.io/quote"
5     "rsc.io/sampler"
6 )
7 |
8 func main() {
9     println(quote.Hello())
10    println(sampler.Say())
11 }
```

- Run the code to verify it works with the new dependency:

```
$ go run hello.go
Hello, world.
This is the sampler.
```

Upgrading dependencies

In Go, a package can depend on other packages to work. These dependencies are managed using Go modules, which allow you to specify the versions of the dependencies your package needs.

To upgrade your dependencies, you need to follow these steps:

- Check the current dependency versions:

```
$ go list -m all
example.com/hello
golang.org/x/text
rsc.io/quote v1.5.2
rsc.io/sampler v1.3.0
```

-Upgrade a minor version of a dependency (`golang.org/x/text` in this example):

```
$ go get golang.org/x/text
```

- Check that the tests still pass

```
$ go test
PASS
```

-Check the new version in the go.mod file::

```
$ cat go.mod
module example.com/hello

go 1.16

require (
    golang.org/x/text v0.3.6 // indirect
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.3.0
)
```

- Upgrade another minor version of a dependency (`rsc.io/sampler` in this example):

```
$ go get rsc.io/sampler@v1.3.1
```

- Check that the tests still pass:

```
$ go test
PASS
```

- Check the new version in the go.mod file:

```
$ cat go.mod
module example.com/hello

go 1.16

require (
    golang.org/x/text v0.3.6 // indirect
    rsc.io/quote v1.5.2
    rsc.io/sampler v1.3.1
)
```

Please remember to change your version control system after upgrading dependencies.

Adding a dependency on a new major version

When adding a new function to a package that depends on a new major version of a module, the following steps can be followed:

- Update the import statement in the source code to import both the old version and the new major version of the module:



```
hello.go X
hello.go
1 package main
2
3 import (
4     import (
5         "rsc.io/quote"
6         quoteV3 "rsc.io/quote/v3"
7     )
8 )
9
10 func main() {
11     func Hello() string {
12         return quote.Hello()
13     }
14 }
```

- Add the new function to the package that depends on the latest major version of the module:


```
hello.go X
hello.go
1  package main
2
3  import (
4      import (
5          "rsc.io/quote"
6          quoteV3 "rsc.io/quote/v3"
7      )
8  )
9
10 func Proverb() string {
11     return quoteV3.Concurrency()
12 }
```

- Write a test function for the new function in a separate test file:

```
hello_test.go X
hello_test.go
1  func TestProverb(t *testing.T) {
2      want := "Concurrency is not parallelism."
3      if got := Proverb(); got != want {
4          t.Errorf("Proverb() = %q, want %q", got, want)
5      }
6  }
7
```

- Run the tests to ensure that the code works as expected:

```
$ go test
PASS
ok      example.com/hello    0.024s
```

- Use the go list command to check the module dependencies:

```
$ go list -m rsc.io/q...
rsc.io/quote v1.5.2
rsc.io/quote/v3 v3.1.0
```

- Update the source code to use the new function from the latest major version of the module:

```
hello.go X
hello.go
1 package main
2
3 import (
4     "rsc.io/quote"
5     "rsc.io/sampler"
6 )
7
8 func main() {
9     func Hello() string {
10         return quote.Hello()
11     }
12 }
```

- Rerun the tests to ensure that the code works as expected:

```
$ go test
PASS
ok      example.com/hello    0.014s
```

- Commit the changes to the version control system:

```
$ git add.
$ git commit -m "Add Proverb function using rsc.io/quote/v3."
```

Upgrading a dependency to a new major version:

Upgrading a dependency to a new major version can introduce breaking changes to your code. Therefore, it's essential to carefully review the release notes and documentation of the new major version before upgrading. Once you're confident that the upgrade won't cause any issues, you can follow these steps to upgrade the dependency:

- Adding a new function that requires a new major version of a dependency:

```
hello.go X
hello.go
1  package mypackage
2  import (
3      "rsc.io/quote"
4      quoteV3 "rsc.io/quote/v3"
5  )
6  func Hello() string {
7      return quote.Hello()
8  }
9  func Proverb() string {
10     return quoteV3.Concurrency()
11 }
```

Note that the `Proverb()` function requires a new major version of the `rsc.io/quote` dependency, identified by the module path `rsc.io/quote/v3`.

- Updating the tests:

```
hello_test.go X
hello_test.go
1  func TestProverb(t *testing.T) {
2      want := "Concurrency is not parallelism."
3      if got := Proverb(); got != want {
4          t.Errorf("Proverb() = %q, want %q", got, want)
5      }
6  }
```

- Running the tests:

```
$ go test
go: finding rsc.io/quote/v3 v3.1.0
go: downloading rsc.io/quote/v3 v3.1.0
go: extracting rsc.io/quote/v3 v3.1.0
PASS
ok      example.com/mypackage    0.021s
```

- Updating the code to use the new major version of the dependency:

```
hello.go X
hello.go
1  package mypackage
2
3  import "rsc.io/quote/v3"
4
5  func Hello() string {
6      return quote.HelloV3()
7  }
8
9  func Proverb() string {
10     return quote.Concurrency()
11 }
```

Note that we've updated the code to use the new major version of the dependency, which includes a new `HelloV3()` function.

- Updating the tests again:

```
hello_test.go X
hello_test.go
1  func TestProverb(t *testing.T) {
2      want := "Concurrency is not parallelism."
3      if got := Proverb(); got != want {
4          t.Errorf("Proverb() = %q, want %q", got, want)
5      }
6  }
7  func TestHello(t *testing.T) {
8      want := "Hello, world."
9      if got := Hello(); got != want {
10         t.Errorf("Hello() = %q, want %q", got, want)
11     }
12 }
```

- Rerunning the tests:

```
$ go test
PASS
ok      example.com/mypackage    0.020s
```

- Committing the changes to version control:

```
$ git add.
$ git commit -m "Add Proverb function that requires rsc.io/quote/v3, upgrade
```

Removing unused dependencies:

It's good practice to periodically review your project dependencies and remove any that are no longer being used. This can help reduce the size of your project and simplify maintenance.

To remove a dependency from your project, you'll need to take the following steps:

- Identify the dependency you want to remove

Look in your code and find any import statements that reference the dependency. Check you're `go.mod` file to see if the dependency is listed.

- Remove the import statement(s) from your code referencing the dependency.
- Use the `go mod tidy` command to remove the unused dependency from your `go.mod` file. This command will remove any modules from your `go.mod` file that is not required by your code.
- Commit the changes to your version control system.

Here's an example of how to remove an unused dependency from a project that was using the `rsc.io/quote/v3` package:

- Identify the dependency:

The `rsc.io/quote/v3` package is no longer being used in the project.

- Remove the import statement:

Remove `import quoteV3 "rsc.io/quote/v3"` from the `hello.go` file.

- Use the `go mod tidy` command:

Run `$go mod tidy` in the project directory to remove the unused dependency from the `go.mod` file.

```
$ go mod tidy
```

This will remove the following line from the `go.mod` file:

- Commit the changes:

Commit the changes to your version control system to save the changes to your project.

```
$ git add hello.go go.mod go.sum  
$ git commit -m "Remove unused dependency rsc.io/quote/v3."
```

Conclusion

Using Go modules is the recommended way of managing dependencies in Go. With the introduction of Go 1.11 and Go 1.12, modules are now available to all supported versions of Go.

This tutorial covered the essential workflows for using Go modules, including initializing a new module, adding and upgrading dependencies, and removing unused dependencies. We also saw how to work with module versions and how to handle dependencies with incompatible significant versions.

To start using modules in your local development, create a go.mod file for your project and add dependencies as needed using `$go get`. To keep your module's dependencies up-to-date and remove any unused dependencies, use `go mod tidy`.

We encourage you to try out modules in your Go projects and provide feedback to the Go community to help shape the future of dependency management in Go. With Go modules, managing dependencies in your Go projects has never been easier!

If you want to move on to my next article, here is the link:

Appreciation and Remarks.

Thank you for reading my article. Please leave your feedback below; it keeps me going and helps me know what I need to work on to improve the quality of the content I put out there.

Here's my Twitter: [Twitter](#)

Find all my links here: [linktr](#)

Cheers!



- Well Researched
- 100% Unique
- 0% Plagiarism
- 100% Human Written

SEO Article

&

Blog Posts Writer

Writing Engaging Content based on
Time-tested and proven on-page SEO techniques



Top-notch Quality content on your desired topic to boost your online presence.



[My Fiverr Gig](#)

👋 Before you go



Give your career some juice. **Join DEV.**

It takes **one minute** and is worth it for your career.

[Get started](#)

Top comments (18)