SkyPenguinLabs

Samples

Go Modules In Depth For Beginners

SPL-PRGC1

Powered by SkyPenguinLabs

SPL-PRGC2 Introduction

> Go Modules in Depth for Beginners

Welcome to one of the first and official free courses released by SkyPenguinLabs!

Are you a beginner who is getting into Go yet having a hard time with its module system? No worries, because I can swear, even on my own personal experience, that this is not uncommon haha! Don't sweat it, but in this course, we are going to be covering these modules in depth and breaking it down to a science to get you familiar with how they may work.

In this module, we will explore various other topics which are described in the ToC (Table of Contents) displayed below.

Powered by SkyPenguinLabs

Table of Contents

> Course Module Outline

The table of contents is kept short, since courses by SkyPenguinLabs are more so individual modules or sets of modules making up sections of entire courses.

Sections start on the next page.

Powered by SkyPenguinLabs

Section 0x00

> Prerequisites

Before we jump into this snippet, and since this is a requirement for almost every course produced under SkyPenguinLabs, I must state the prerequisites to ensure you do not expect this to be within your current skill level and are surprised when it is not.

Generally speaking, this module is kept to a very bare minimum, is designed and written towards a beginner audience with minimal experience in the Go programming language, so minimal in fact, that the individual has merely started to explore how the language actually works to compile things together (e*.g: modules, libraries, code references, dynamic resolution*)

Simply put, that's it, enjoy the module :D

Powered by SkyPenguinLabs

Section 0x01

> The Science Behind Golang's Module Architecture

Now, getting straight into it, given that you may understand the basics of the Go programming language, it's about time to stretch into the wacky module system that Go has.

I say wacky for a few reasons, primarily because, right off the bat, unlike most programming languages, modules require an initial package, a binary, package. This package file, known as the `go.mod` file, contains metadata information about the project's current source directory, and other relevant references. You can also use this file to modify the way resources get compiled and interpreted by the go-compiler using directives such as `replace` which we will get into later.

Long story short, the code's non-standard references are included in this file along with the resources corresponding version/commit, which is used for resource versioning when compiling the application.

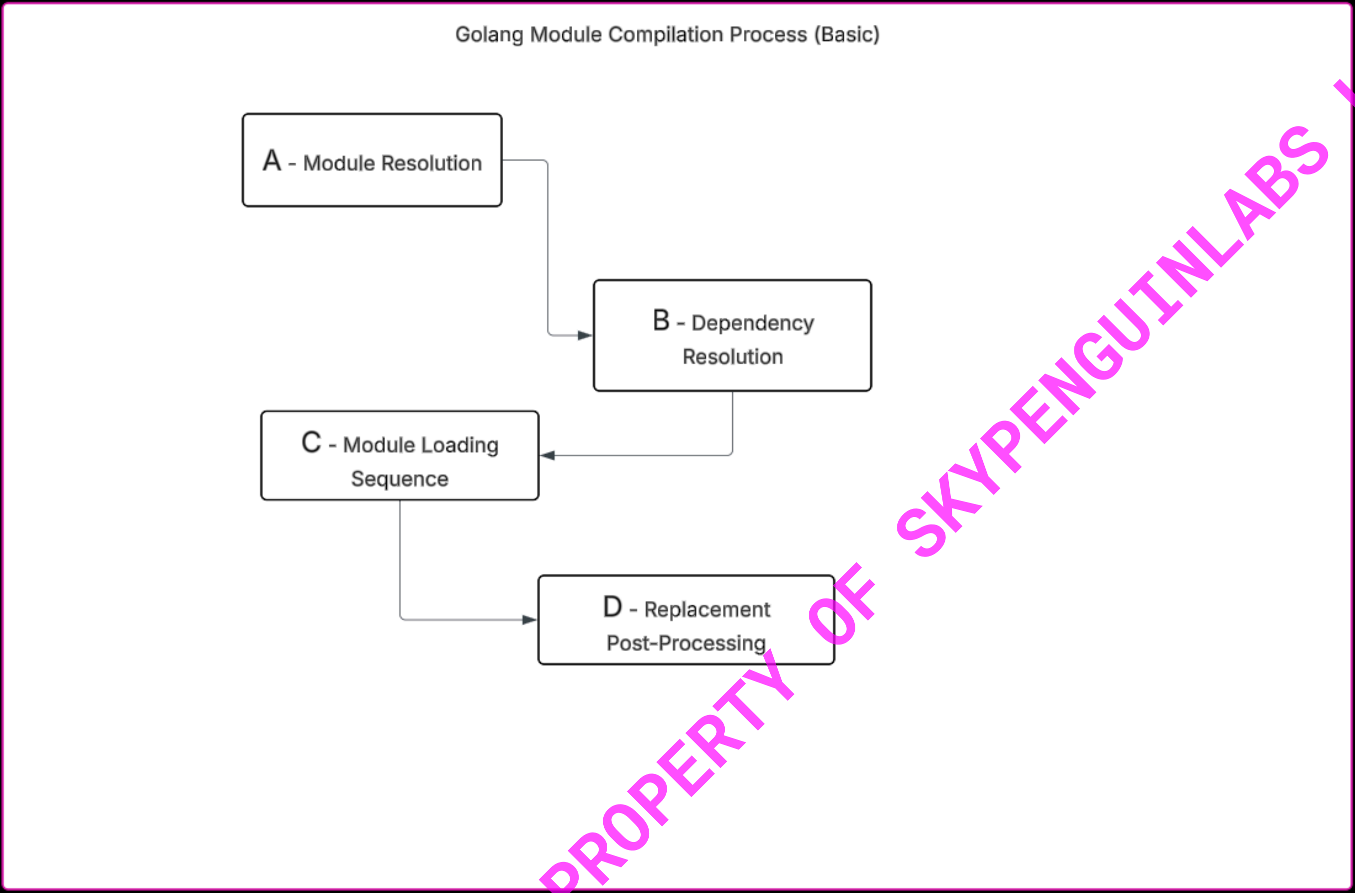
This go.mod file, references a go.sum file, to which summarizes the resources and their corresponding cryptographic hashes used for resource integrity when the compiler runs the modules.

INFORMATIONAL! 0x01

Many people forget that Golang was not designed like many other languages, solely with the idea to be a directly replaceable alternative to C or C++ to average developers. Instead, Go focuses on simplicity and developer productivity. This makes Go quite hard to grasp for many, and leads to people getting lost in the large library of features it contains

Directly, the package, referenced inside of the go.mod file, can contain many different modules which get required and compiled in that file.

In order to fully understand modules though, we kind of have to break down how the compiler sees them. This is visualized in the diagram below.


Golang Module Compilation Process (Basic)
A - Module Resolution
B - Dependency Resolution
C - Module Loading Sequence
D - Replacement Post-Processing

The following steps listed above are broken down in a list-style format below.

- **A - Module Resolution |** This step is the initial step, it looks for the `go.mod` file to reference the canonical import path of all modules, the modules corresponding go version being targeted, and pass them to the dependency resolution step.

- **B - Dependency Resolution |** This is the second step, which involves taking the results from module resolution and actually taking to versioning the dependencies based on relationships to the modules. Each node in this "graph" represents a module, which has an edge, which represents a dependency relationship where each relationship is simply version based around semantic versioning rules in the compiler. Many people forget this too, but

Go’s compiler uses a minimal version selection algorithm,which resolves to taking the most minimum version that satisfies all requirements across the dependency graph. This is actually a huge optimization scheme from package managers that choose the newest version that satisfies all constraints.

- C - Module Loading Sequence - This is the portion of the compilation step which involves verifying import paths to make sure it matches the current modules path (if not, handle accordingly, one example of the compiler doing so is by searching module cache, if its not there, downloading the specified version) and then verifying the integrity of the package hash represented in `go.sum`.

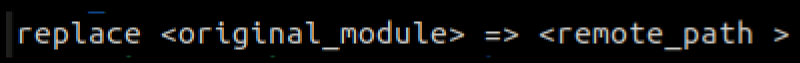
- D - Any Post-Processing Steps - Post-processing, the module system will look for replacement directives, which are directives that can be used in `go.mod` which tell the Go compiler to take ‘x’ package on the left side of the directive, and instead of using that reference, reference the source on the right when the source code itself implements the left side path. Woah, that was confusing, I understand. So let me break it down some more.

In order to use a module file, such as ‘fatih/color’ you need to take the package name of that module, which in the case of ‘fatih/color’ is ‘github.com/fatih/color’ and use that as the prefix before importing ANY files referencing that package. When we want to import a package apart of that module, we add the folders actual name. For example, if package `SuperSecretpackage` existed in `some_module` the compiler would see `github.com/fatih/color/some_module`

When we reference this, the Go compiler sees that we are trying to use the module ‘some_module’ and fetch all the code from that directory, so long as the package is installed on your system and does not conflict with your existing package.

We can use the ‘replace’ directive, in the go.mod file, to tell the Go compiler, when it sees that we have called to import ‘x’ go package or a specific module from that package, we tell the compiler to instead load source from ‘x’ directory/location.

Replacement call is shown below.



```
replace <original_module> => <remote_path >
```

As you can see, packages, which contain modules, which can contain more packages get kind of confusing, so try not to think too deeply into it because as deep and intricate as it may seem, it really in all honesty is not too terrible. It just takes a little timing to get adjusted to the weird kinks.

Now that we understand a little bit about how go modules function, let's conclude this section so we can hop into using them!

To conclude, Go modules operate under a pretty baseline consensus, but there are a few things that Go expects you to do as a programmer to prevent issues when working with them. This includes but is not limited to

- **Being responsible for knowing the package -** This includes checking the package name, knowing and keeping a direct order of your packages, and following all syntactic rules when naming packages to ensure things stay consistent which involves ensuring that the package names are lowercase, single worded if plausible, use no underscores or MixedCaps, and the naming should not conflict with standard libraries, nor include functions which conflict with standard libraries.

- **Understanding Resource Integrity -** Make sure before you start messing around with versioning in-depth with Go, and come across dependency errors such as this one, relevant to the integrity of the resource and ensure that the package you downloaded is correct.

- **Understanding how it works -** I think it would be really hard to solve module related errors with Go without understanding the structure above to the most minimalist amount plausible to the surface level scope.

Powered by SkyPenguinLabs

Section 0x02

> Basic Use of Go Modules

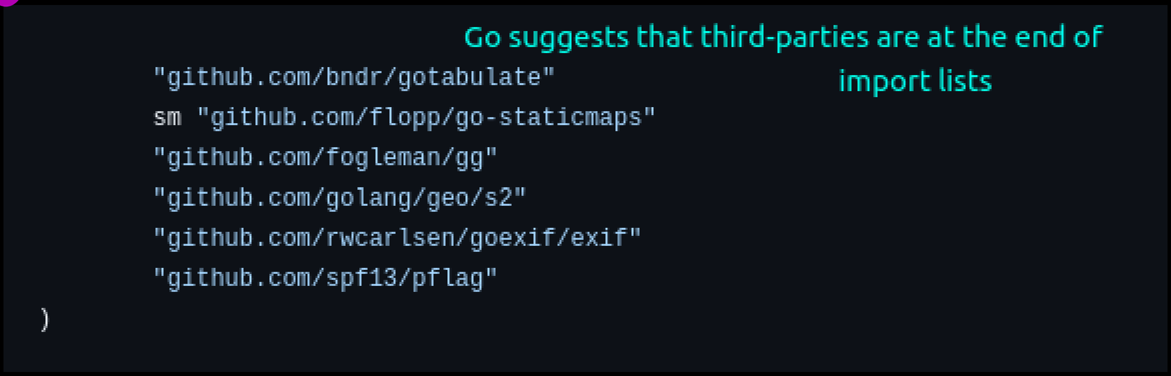
Using Go modules is actually not too terrible in itself. For a while, installing them and managing them was and still is considered a problem within Go's ecosystem. This is primarily due to the way Go chose to implement the 'go get'/'go install' tools where go install was meant to replace go get but in turn resulted in more errors for people when it first came out eventually Google fixing it and pushing out updates.

That being said, in this section, we will be using Golang version 1.24.2 linux/amd64 which is going to be using go install to install and work with Golang packages.

The following sections go through using Go modules properly in different scenarios.

Scenario 1 - Remote Packages (Baseline Use)

Everybody has probably seen the way packages get used here before. In fact, this is literally the front page of Golang applications. It's the standard way of importing remote modules. Simply just referencing the remote library, and a module identifier (*e.x: 'sm' referencing go-staticmaps*) which is used on one of the libraries. Then you can call types and functions from the code.



```
          "github.com/bndr/gotabulate"
          sm "github.com/flopp/go-staticmaps"
          "github.com/fogleman/gg"
          "github.com/golang/geo/s2"
          "github.com/rwcarlsen/goexif/exif"
          "github.com/spf13/pflag"
)
```

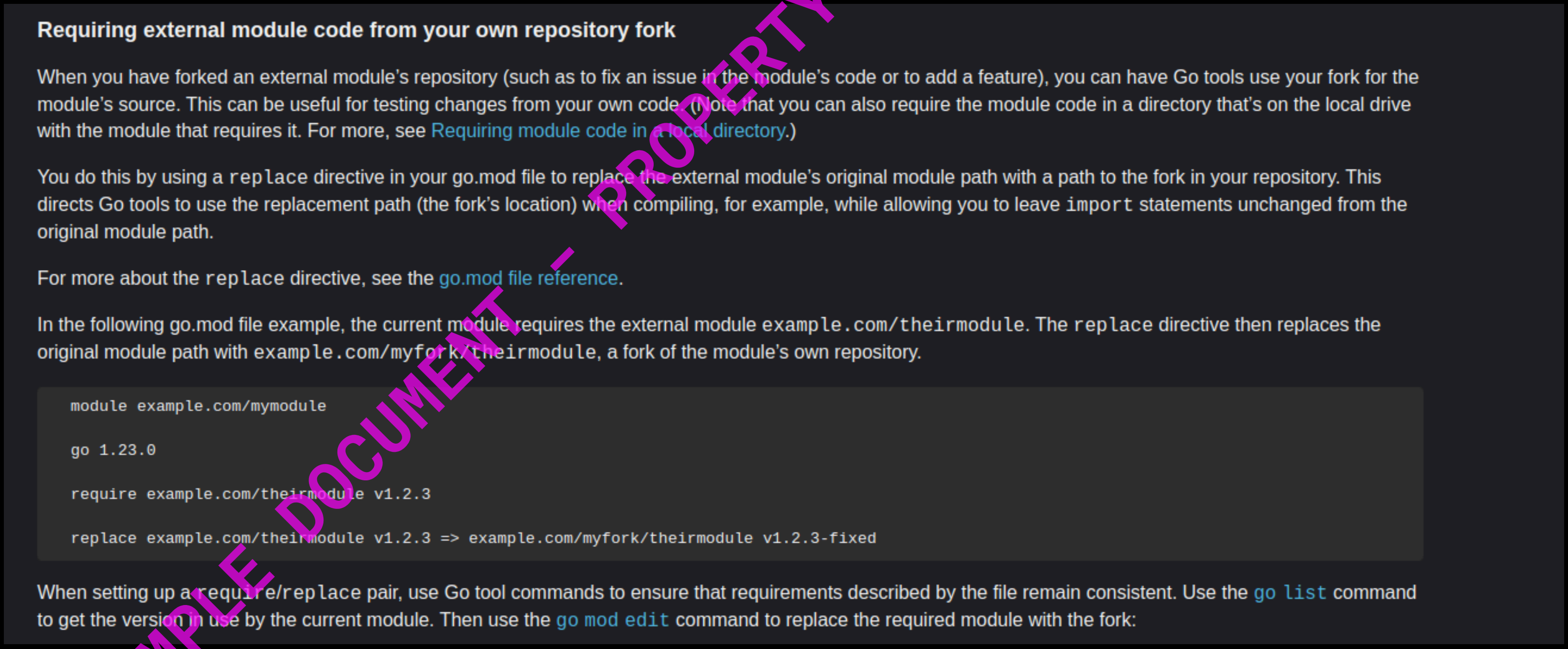
Go suggests that third-parties are at the end of import lists

Scenario 2- Using Custom Forks

Using the 'replace' call can be used to load your own forks. Unsure of if the reader caught this when I was referencing this, but if you really wanted to, and felt a package has a good version which would be useful to standardize across code-bases, whilst also not being a critical component, to escape breaking changes and versioning issues with auto-updates, you can easily localize a custom fork or version of the package using the replace call.

Custom forks are helpful for many reasons outside of versioning issues as well, such as wanting to remove bloat, or modify the library to your needs. It definitely becomes more handy in projects that require as precise optimization as possible towards resource requirements.

From Golang's official documentation specifically talking about managing the dependencies, they cover this exact concept, screenshotted below.



Requiring external module code from your own repository fork

When you have forked an external module's repository (such as to fix an issue in the module's code or to add a feature), you can have Go tools use your fork for the module's source. This can be useful for testing changes from your own code. (Note that you can also require the module code in a directory that's on the local drive with the module that requires it. For more, see Requiring module code in a local directory.)

You do this by using a `replace` directive in your go.mod file to replace the external module's original module path with a path to the fork in your repository. This directs Go tools to use the replacement path (the fork's location) when compiling, for example, while allowing you to leave `import` statements unchanged from the original module path.

For more about the `replace` directive, see the go.mod file reference.

In the following go.mod file example, the current module requires the external module `example.com/theirmodule`. The `replace` directive then replaces the original module path with `example.com/myfork/theirmodule`, a fork of the module's own repository.

```
module example.com/mymodule

go 1.23.0

require example.com/theirmodule v1.2.3

replace example.com/theirmodule v1.2.3 => example.com/myfork/theirmodule v1.2.3-fixed
```

When setting up a `require`/`replace` pair, use Go tool commands to ensure that requirements described by the file remain consistent. Use the `go list` command to get the version in use by the current module. Then use the `go mod edit` command to replace the required module with the fork: