# Golang Libs Dep

## Use go.mod to manage project dependencies

- Why use go.mod

Before using go.mod, we used dep to manage golang project dependencies. When I added translation function to the project, I encountered a package google.golang.org/api could not be found using dep, but it can be found on GitHub. Using the get tool to get it, it's annoying to not use dep to manage it.

The following are the key pointsGo officially plans to open go mod by default in the future, and deprecate gopath in 1.13. [https://blog.golang.org/modules2019]

- How to use go.mod ?

1. Here we use a common service to give an example. This service is an infrastructure service of golang for file uploading and Google translation, and there are many third-party packages that depend on it.

Before using go.mod to manage the project, we first set the **export GO111MODULE=on**

- **use go mod init**

```
export GO111MODULE=on

cd $GOPATH/github.com/tinklabs/common_service

#go mod init module-name(general use project path), example:
go mod init github.com/tinklabs/common_service

#use the go build command to get the current dependency
go build
```

After getting all done, we can use go mod download to cache the dependencies locally. When we complete the first initialization dependency, we have two files, one go.mod, another go.sum, the go.mod file defines the module path and lists other specific versions of the modules that need to be imported at build time. , go.sum is a module download entry generated from go.mod.

- **use go mod download**

After using go mod for the first time, use go mod download to cache the dependency download. Building it later will greatly improve the build speed.

```
go help mod download
usage: go mod download [-json] [modules]

Download downloads the named modules, which can be module patterns
selecting
dependencies of the main module or module queries of the form
path@version.
With no arguments, download applies to all dependencies of the main
module.

The go command will automatically download modules as needed during
ordinary
execution. The "go mod download" command is useful mainly for
pre-filling
the local cache or to compute the answers for a Go module proxy.

By default, download reports errors to standard error but is otherwise
silent.
The -json flag causes download to print a sequence of JSON objects
to standard output, describing each downloaded module (or failure),
corresponding to this Go struct:

    type Module struct {
        Path     string // module path
        Version  string // module version
        Error    string // error loading module
        Info     string // absolute path to cached .info file
        GoMod    string // absolute path to cached .mod file
        Zip      string // absolute path to cached .zip file
        Dir      string // absolute path to cached source root directory
        Sum      string // checksum for path, version (as in go.sum)
        GoModSum string // checksum for go.mod (as in go.sum)
    }
```

**notice:** The dependency format must be **path@version**

2.How do we keep incremental updates to go.mod after each update of project dependencies?

- **use go mod tidy**

Execute the **go mod tidy** command, which adds missing modules and removes unwanted modules. After execution, the go.sum file (module download entry) is generated. Add the parameter -v, for example, go mod tidy -v to print the executed information, ie deleted and added packages, to the command line

```
go mod tidy -v
```

- **use go mod verify**

Execute the command go mod verify to check whether the dependencies of the current module are all downloaded, and whether they have been downloaded and modified. If all modules have not been modified, then all modules verified will be printed after executing this command.

```
go mod verify
```

- **use go mod vendor**

Run the go mod vendor command to generate the vendor folder. The folder will be placed with the dependencies described by your go.mod file. There is also a file modules.txt under the folder, which is all modules of your entire project. Before executing this command, **if you have a vendor directory before the project, you should delete it first.** Similarly, **go mod vendor -v** will print out the modules added to the vendor.

```
go mod vendor -v
```

**Features: backward compatible**

If there are two projects A, B and project B depends on project A.
When we use the go.mod tool to manage dependencies, we need to first go to the A project go mod init. After processing the A project, we will rely on the B project.

**Known issues:**

1.Environmental reasons such as the Go version may cause the go.sum to be calculated differently. It is necessary to observe the use of a unified goproxy to solve the problem.

2.Goproxy currently has some minor bugs, and some packages don't have problems.

```
go help goproxy

# ci or docker file add env
GOPROXY=https://example.com/proxy
```

**Makefile**

```
# simple example
GOCMD=go
DEPCMD=$(GOCMD) mod
GOBUILD=$(GOCMD) build
DEPENSURE=$(DEPCMD) tidy

all: test build
build:
	$(GOBUILD)
deps:
	$(DEPCMD) download && $(DEPENSURE) -v
```