Modules in Go

Session 14

Golang course by Exadel

28 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel

Agenda

- ▶️ GOPATH old days
- ▶️ Modules
- ▶️ Multi-module workspaces
- ▶️ Next time...

GOPATH old days

Go workspace in GOPATH era

▶️ Workspace looks like this:

```
$GOPATH/
    bin/fixhub                              # installed binary
    pkg/darwin_amd64/                       # compiled archives
        code.google.com/p/goauth2/oauth.a
        github.com/...
    src/                                    # source repositories
        code.google.com/p/goauth2/
            .hg
            oauth                           # used by package go-github
            ...
        github.com/
            golang/lint/...                 # used by package fixhub
                .git
            google/go-github/...            # used by package fixhub
                .git
            dsymonds/fixhub/
                .git
                client.go
                cmd/fixhub/fixhub.go        # package main
```

Why such file layout?

- Using file layout for builds means less configuration.
  - In fact, it means no configuration.
  - No Makefile, no build.xml.
  - Less time configuring.
- Everyone in the community uses the same layout. This makes it easier to share code.
- The Go tool helps build the Go community.

Source: "Organizing Go code" by David Crawshaw (https://go.dev/talks/2014/organizeio.slide#12)

GOPATH environment variable

▶️ Before go 1.11 all the source code shall be located under `$GOPATH/src` folder

- The Go path is used to resolve import statements.

▶️ The GOPATH environment variable lists places to look for Go code. *It might be a list of places, not just one place.*

- On **Unix/MacOS,** the value is a colon-separated (`:`) string.
- On **Windows**, the value is a semicolon-separated (`;`) string.

▶️ If the environment variable is unset, GOPATH defaults to a subdirectory named "go" in the user's home directory

- $HOME/go on Unix, %USERPROFILE%\go on Windows
- Run "go env GOPATH" to see the current GOPATH.

▶️ Interesting fact: in **Go 1.5** it was controlled by `GO15VENDOREXPERIMENT` env variable (now is deprecated) to enable/disable vendoring.

Dependencies Vendoring

▶️ Dependencies vendoring was (and is) a very useful technique:

```
/home/user/go/                  (GOPATH set to this path)
    src/
        crash/
            bang/               (Go code in package `bang`)
                b.go
        foo/                    (Go code in package `foo`)
            f.go
            bar/                (Go code in package `bar`)
                x.go
            vendor/             (Vendored folder)
                crash/
                    bang/       (code in `b.go` is imported as “crash/bang”, not as "foo/vendor/crash/bang")
                        b.go
                baz/            (code in `z.go` is imported as "baz", not as "foo/vendor/baz")
                    z.go
            quux/               (Go code in package `quux`)
                y.go
```

Reproducible build issue

▶️ Put simply, `go get` doesn't guarentee reproducible builds. Proposed solutions, tools like:

- godep
- gopkg.in
- govendor
- gb

▶️ Promoted the idea of a `vendor/` directory, a self-contained `$GOPATH` that could be checked in with the code so that your program had a copy of each of the dependencies it needed.

Solutions to help managing dependencies: variety of options

- Glide (https://github.com/Masterminds/glide) tool: combination of `glide.yaml` and `glide.lock` files
- DEP tool (https://github.com/golang/dep) was intended to be official solution.
  - Quote: "Hey **dep**, please make sure that *my project is in sync*: that **Gopkg.lock** satisfies all the imports in my project, and all the rules in **Gopkg.toml**, and that `vendor/` contains exactly what **Gopkg.lock** says it should."
- gopkg.in (https://labix.org/gopkg.in) provides versioned package paths:
  - `gopkg.in/user/pkg.v3 -> github.com/user/pkg (branch/tag v3, v3.N, or v.3.N.M)`
- Again, most of the tools are:
  - Not a complex solutions, like `maven`, `ant`, etc. => no plugins, programming
  - List of supported tools for migration to Go Modules: here (https://pkg.go.dev/cmd/go/internal/modconv@master#pkg-variables)

Solutions to help managing dependencies: under the hood

▶️ Many others: all of them shall control:

- dependency versioning based on git tags/branches/hashes
- vendor folder content population

▶️ Operates on two files:

- List of desired dependencies with versions (and some helper directives, probably)
- List of actual resolved dependency versions (example: `*.lock` file)

Modules

Why there is a need for Go Modules?

▶️ Prior to Go modules, `go get` only knew how to fetch whatever revision happened to be current in your repository at the time.

▶️ If you already had a copy of a package in your `$GOPATH` then `go get` would *skip it*, so you might end up building against a **really old version**.

▶️ If you used the `go get -u` flag to force it to download a fresh copy, you might find that you now had a **much newer version** of a package than the author.

▶️ `go get` does not provide reproducible builds.

▶️ In early 2018 the Go team (Russ Cox) proposed their own tool, at the time given the working title **vgo**, now known as **go modules**.

- Go modules are integrated into the Go tool. The notion of modules is baked in as a first class citizen.
- This makes it possible for Go developers to build their code anywhere they want.

What is a Go Module

▶️ **Go 1.11** introduced a new concept of Modules which brings first class support for managing dependency versions and enabling reproducible builds.

▶️ Go previously had no notion of dependency versions.

▶️ A **Module** is a way to **group together a set of packages** and give it a **version number** to mark it's existence (state) at a specific point in time.

- **Modules** have versions and the **version number is meaningful**.
- Go Modules use Semantic Versioning for their numbering scheme.

▶️ Modules record **precise dependency requirements** and create reproducible builds.

▶️ No more `GOPATH`

Source: "A gentle introduction to Golang Modules" by Ukiah Smith (https://ukiahsmith.com/blog/a-gentle-introduction-to-golang-modules/)

No more GOPATH

▶️ Modules allow for the deprecation of the `GOPATH`.

▶️ There is no longer a need to set it explicitly as a `go.mod` files defines the root of a Module, and allows the Go toolchain to know where everything is that it needs to work with.

- This was the purpose of GOPATH.

▶️ With modern Go toolchain, check the difference between command outputs:

- `$ go env GOPATH`
- `$ env | grep GOPATH`

Useful module-related commands:

```
11376 skovtunenko:graterm-example$ go mod help
Go mod provides access to operations on modules.

Note that support for modules is built into all the go commands,
not just 'go mod'. For example, day-to-day adding, removing, upgrading,
and downgrading of dependencies should be done using 'go get'.
See 'go help modules' for an overview of module functionality.

Usage:

        go mod <command> [arguments]

The commands are:

        download    download modules to local cache
        edit        edit go.mod from tools or scripts
        graph       print module requirement graph
        init        initialize new module in current directory
        tidy        add missing and remove unused modules
        vendor      make vendored copy of dependencies
        verify      verify dependencies have expected content
        why         explain why packages or modules are needed

Use "go help mod <command>" for more information about a command.
```

GOPATH and Modules: different operation modes

▶️ Environment variable `GO111MODULE` for backward compatibility. Possible values:

- If `GO111MODULE=off`, the go command ignores `go.mod` files and runs in GOPATH mode
- If `GO111MODULE=on` **or is unset**, the go command runs in module-aware mode, even when no `go.mod` file is present. Not all commands work without a `go.mod` file
- If `GO111MODULE=auto`, the go command runs in module-aware mode if a `go.mod` file is present in the current directory or any parent directory.

▶️ Try yourself: `go env GO111MODULE` => will output nothing on modern Go toolchain.

New place for documentation

- Was: godoc.org (godoc.org). Now it was migrated to PKG.GO.DEV (https://pkg.go.dev/)
- New site will search using proxy.golang.org (proxy.golang.org) database
- Nice-looking markdown-enabled parser to display repo documentation
- List of available versions, who is using the library?

Key Go Module commands

- For initialization: `go mod init`
- To download sources: `go mod download`
- To make sure that dependencies are up-to-date: `go mod tidy`
- To support vendoring: `go mod vendor`

Go Mod: init

- ▶️ **go mod init**
- ▶️ Support migrations from other tools
- ▶️ `go 1.x` directive specifies minimum toolchain version
- ▶️ You can initialize new projects everywhere!

```
module github.com/skovtunenko/test

go 1.15

require github.com/sirupsen/logrus v1.7.0
```

Go Mod: download

- ▶️ **go mod download**
- ▶️ Use to download dependencies
- ▶️ Automatically will be executed before `go build`
- ▶️ Integrated with `go mod tidy`

```
go: downloading github.com/sirupsen/logrus
v1.7.0
go: downloading golang.org/x/sys
v0.0.0-20191026070338-33540a1f6037
go: downloading github.com/stretchr/testify
v1.2.2
go: downloading github.com/pmezard/go-difflib
v1.0.0
go: downloading github.com/davecgh/go-spew
v1.1.1
```

Go Mod: get (1/2)

- **go get .....**
- Will download and add module as a dependency
- You can specify using @ symbol:
  - version
  - branch
  - commit hash
- Examples: `go get ...@latest` / `go get .....@none`

```
$ go get github.com/sirupsen/logrus@6699a89a232f3db797f2
e280639854bbc4b89725

go: downloading github.com/sirupsen/logrus
v1.7.0
go: github.com/sirupsen/logrus
6699a89a232f3db797f2e280639854bbc4b89725 => v1.7.0
go: downloading golang.org/x/sys
v0.0.0-20191026070338-33540a1f6037
```

Go Mod: get (2/2)

- Command **go get x.y.z@version** will update dependency for required version
- Using flag `-u` will update transitive dependencies
- Using flag `-u=patch` will take into account only patch-versions
- Using flag `-t` will download dependencies for testing
- Using flag `-d` will download, but ignore installation

```
$ go get -u

go: github.com/sirupsen/logrus upgrade => v1.7.0
go: golang.org/x/sys upgrade => v0.0.0-20201015000850-e3ed0017c211
```

Go Mod: tidy

- **go mod tidy**
- To make sure that everything downloaded, only necessary dependencies are there
- This is the main command to make sure that all is good
- Before:

```
require (
    github.com/golang/glog v0.0.0-20160126235308-23def4e6c14b // indirect
    github.com/sirupsen/logrus v1.7.0
    go.uber.org/zap v1.16.0 // indirect
)
```

- After:

```
require (
    github.com/sirupsen/logrus v1.7.0
)
```

Go Mod: vendor

- **go mod vendor**
- In use to download dependencies into `/vendor` directory
- Potential use-cases:
  - backward-compatibility with older approaches
  - CI/CD without (extra) network access
- Example:

```
$ cat vendor/modules.txt

github.com/sirupsen/logrus v1.7.0
 explicit
github.com/sirupsen/logrus
 golang.org/x/sys
v0.0.0-20191026070338-33540a1f6037
golang.org/x/sys/unix
golang.org/x/sys/windows
```

Semantic Versioning

Major version: increment for backwards-incompatible changes.

Minor version: increment for new features.

Patch version: increment for bug fixes.

v2.3.4 (semantic versioning)

```
import "my/thing/v2/sub/pkg"
```

(semantic import versioning)

Source: "Semantic Import Versioning" by Russ Cox (https://research.swtch.com/vgo-import)

Import compatibility rule for Go

▶️ Import compatibility rule for Go

```
If an old package and a new package have the same import path,
the new package must be backwards compatible with the old package. (c) Russ Cox
```

▶️ **Minimum Version Selection**: is a strategy of deciding which version of a library to use given the constraints that a developer has specified in the **go.mod** file, and the constraints of all the other dependent libraries used.

- If our project's **go.mod** file specifies that it needs **v1.2** of a library, and one of it's other dependencies specifies that it needs **v1.3** of the same library then Go will use **v1.3**, as it is the lowest version (oldest) that satifies all stated version needs.
- Avoid problems of accidentally bumping unrelated dependencies when new versions are released

Source: "A gentle introduction to Golang Modules" (https://ukiahsmith.com/blog/a-gentle-introduction-to-golang-modules/)

Advanced concepts: proxy & private repos

- Proxy using `GOPROXY` environment variable
  - `$ go env GOPROXY` => output is: **https://proxy.golang.org,direct**
- Different proxies:
  - Existing public ones
  - goproxy (https://github.com/elazarl/goproxy)
  - Athens (https://github.com/gomods/athens)
  - GitLab as a go proxy (https://docs.gitlab.com/ee/user/packages/go_proxy/#add-gitlab-as-a-go-proxy)
- Private repositories: `GOPRIVATE=gitlab.yourorganization.com`

Libraries with versions v2 +

- Example: Olivere ElasticSearch driver (https://olivere.github.io/elastic/)
- Module path needs to be changed to have: v2 / v3 / etc.
  - For module versions v0 - v1 there is no need to change the path!

Possible issues

▶️ Strictly follow semantic versioning! Watch for breaking API changes:

- Breaking API code change.
- No API change, but breaking change in the implementation logic.

▶️ Issues when some project want to go against **Go Modules** approach: chi@v1.5.x mod issues #561 (https://github.com/go-chi/chi/issues/561#issuecomment-739963585)

▶️ Do not fight against the **Go Modules** logic. Just obey - that's a reality now.

Multi-module workspaces

Multi-module workspaces

▶️ With *multi-module workspaces*, you can tell the Go command that you're writing code in multiple modules at the same time and easily build and run code in those modules.

▶️ Set of commands:

```
$ mkdir workspace
$ cd workspace

$ mkdir hello
$ cd hello
$ go mod init example.com/hello
go: creating new go.mod: module example.com/hello
```

▶️ The go work init command tells go to create a go.work file for a workspace:

```
$ go work init ./hello
```

▶️ clone some other repo: `git clone https://go.googlesource.com/example`. Add the module to the workspace:

```
$ go work use ./example
```

Multi-module workspaces: outcome

▶️ The go work use command adds a new module to the go.work file. It will now look like this:

```
go 1.18

use (
    ./hello
    ./example
)
```

▶️ This will allow us to use the new code we will write in our copy of the stringutil module instead of the version of the module in the module cache that we downloaded with the go get command.

▶️ Since the two modules are in the same workspace it's easy to make a change in one module and use it in another.

Next time...

▶️ Session15:

Goroutines, channels in Go

- Goroutines
- Channels: bi-directional and uni-directional, buffered and unbuffered
- Channels Axioms
- Select statement
- Blocking vs. non-blocking flows
- Rules for using channels
- Best practices

Thank you

Golang course by Exadel

28 Nov 2022

Sergio Kovtunenko
Lead backend developer, Exadel
skovtunenko@exadel.com (mailto:skovtunenko@exadel.com)
https://github.com/skovtunenko (https://github.com/skovtunenko)
@realSKovtunenko (http://twitter.com/realSKovtunenko)