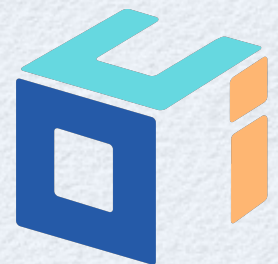


Go Modules

slides at <https://github.com/mvolkmann/talks>

R. Mark Volkmann
Object Computing, Inc.
<http://objectcomputing.com>
Email: mark@objectcomputing.com
Twitter: @mark_volkman
GitHub: mvolkmann



OCI | TRAINING

Past

- Lack of support for package versioning was a major issue before Go version 1.11
 - all projects under same `GOPATH` use same dependency versions
 - no record of dependency versions
 - hard for teammates to get same dependency versions
 - encourages use of a mono repo
- Leading contenders were
 - **vgo** from Russ Cox - <https://github.com/golang/go/wiki/vgo>
 - **dep** from Sam Boyer - <https://golang.github.io/dep/>



Present

- Go 1.11 includes experimental support for “modules”
- Mostly based on vgo
- Eliminates need to have code under `GOPATH`
- Adds support for dependency versioning
 - teammates can get same versions
 - repeatable builds



Modules

- A module is defined by a directory
 - referred to as “module root”
 - contains `go.mod` file and a set of source files
- `go.mod` can be created and updated manually
 - but there is no need to do either
- Easiest way to create `go.mod` for a new module
 - cd to its module root
 - run `go mod init module-name`
- ***module-name***
 - is the import path other modules will use to import this one
 - typically a version control path such as `github.com/mvolkmann/my-module`
 - can be a simple name for modules that will not be published

Simple Example ...

- Create new directory named `my-simple-module` not under `GOPATH`
- `cd` to directory
- `go mod init my-simple-module`
 - a GitHub path is not specified in this example because we are not planning to share this module with others
 - creates `go.mod` file containing `module my-simple-module`
- Create `main.go` with following content

later we will see other "directives" that can be in this file

```
package main

import (
    "fmt"
    "github.com/ttacon/chalk"
)

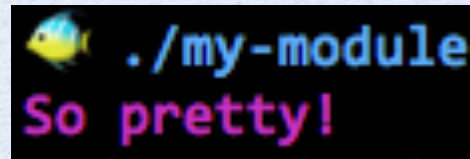
func main() {
    fmt.Printf("%s%s%s\n", chalk.Magenta, "So pretty!", chalk.Reset)
}
```

imports a single community package that is not yet listed in `go.mod`

... Simple Example

- To build this application and update `go.mod` with dependency information, enter `go build`

- creates executable file `my-simple-module`
- adds following to `go.mod`



```
require github.com/ttacon/chalk v0.0.0-20160626202418-22c06c80ed31
```

- version string that follows dependency path will be explained later
- creates `go.sum` file which stores checksum information for all dependencies
 - checksums are described later

Transition to Module Support

- In Go 1.11 it is possible to **use both** the **old** `GOPATH` approach **and** the **new** module approach
- Specified by environment variable `GO111MODULE`
 - when `off`, only `GOPATH` can be used
 - when `on`, only modules can be used
 - when `auto` or not set, either can be used and the **choice is based on** whether commands are run from a directory that contains a `go.mod` file or a descendant of such a directory
- If you have existing code that relies on `GOPATH` and you want to try modules in new or existing packages, not setting `GO111MODULE` is a good option

Adding Module Dependencies

- Easiest way
 - create source files that contain `import` statements for dependencies
 - run a command such as `go build`, `go test`, `go list`, or `go vet`
 - these commands trigger lookup of all dependencies
 - results are used to automatically update `go.mod`

Semantic Versioning

- Go modules prefer use of **semantic versioning** where version numbers have three parts referred to as major, minor, and patch
- For example, "1.2.3" represents
 - major version of 1
 - minor version of 2
 - patch version of 3
- **Patch** is incremented when backward-compatible **bug fixes** are made
- **Minor** is incremented when backward-compatible **new features** are added
- **Major** is incremented when **incompatible changes** are made

Module Code Layout

- No requirement to have a `src` directory in module root
- For simple modules
 - all source files can be in module root along with `go.mod`
- For modules defined by many source files
 - source files can be organized in subdirectories as desired, typically to indicate sets of related files

Dependency Source Code

- Source files for dependencies are not stored in the “module root” directories of modules that use them
- Instead they are stored in subdirectories of `$GOPATH/pkg/mod`
 - allows multiple modules to share them
 - when `GOPATH` is not set, `$HOME/go` is used
 - to see this, enter `go env GOPATH`
- Multiple versions of each dependency can be stored here
 - allows modules that depend on them to use different versions

Explicit Installs ...

- It is also possible to install dependencies with `go get`
 - updates `go.mod`
 - adds comment `// indirect` after path for new dependency
 - indicates that no code in current module has been seen yet that uses the dependency
- **indirect** comments
 - removed after uses of dependencies are added to module source files and a command such as `go build` is run
- **Not necessary** to use `go get` to install dependencies
 - since commands like `go build` add dependencies to `go.mod` on their own and they will be run eventually

... Explicit Installs

- **Primary reason** to use `go get` with modules
 - to **specify a specific version** to be installed
- Alternatively
 - once some version is installed, perhaps by running `go build`, the version to use can be modified by editing `go.mod` and re-running the command
- Changing version of a direct dependency
 - can change versions of its dependencies that are used
 - desirable since a specific version of a direct dependency may only work with specific versions of its dependencies

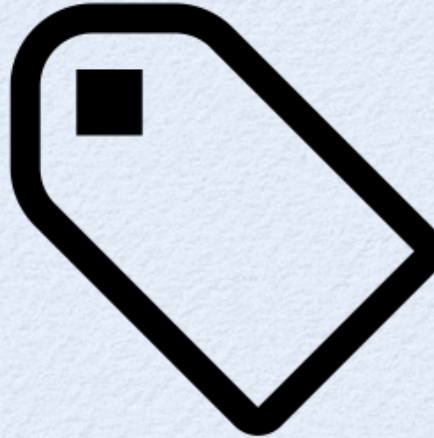
Releasing New Module Versions

- Major versions
 - 0 or 1 are considered unstable
 - 2 or higher are considered stable
- To release a new unstable version of a module
 - push changes
 - add repository tag following pattern `v{major}.{minor}.{patch}`
- To release a new stable version of a module
 - modify module name in `go.mod` to end with new major version
 - for example `module github.com/mvolkmann/my-module/v2`
 - push changes
 - add repository tag just like for unstable versions

seems odd to me
that major version 1
is considered unstable

Adding a Git Tag

- From command-line
 - `git tag tag-name`
 - `git push origin tag-name`
- From GitHub web UI
 - click "release(s)" tab
 - press "Draft a new release" button
 - enter tag name
 - optionally enter title and description
 - if not considered stable, check "This is a pre-release"
 - press "Publish release" button



Importing Versions

- Unstable versions
 - are imported without specifying major version
 - for example, `import github.com/mvolkmann/my-module`
 - will use latest version less than v2
- Stable versions
 - add major version to import paths in all source files that import it
 - for example, `import github.com/mvolkmann/my-module/v2`
 - only major version is specified, not minor or patch
 - referred to as “semantic import versioning”
- Presumably there will be tooling to automate this in the future
 - so it won't be necessary to manually update multiple source files

Ternary Example

- Create a reusable module
 - create GitHub repo at <https://github.com/mvolkmann/ternary>
 - clone the empty repo
 - create `main.go` (see next slide)
 - `go mod init github.com/mvolkmann/ternary` creates `go.mod`
 - add, commit, and push `go.mod` and `main.go`
 - add repository tag of `v0.1.0`
- Create an application that uses it
 - create directory `ternary-demo`
 - create `main.go` (ahead two slides)
 - `go mod init ternary-demo` creates `go.mod`
 - `go build` adds `require` to `go.mod`, installs `ternary` module, creates `go.sum`, and creates executable
 - `./ternary-demo` runs executable

ternary main.go

```
package ternary

// Any implements a ternary for any type.
// Typically a type assertion will be required.
// For example, color := ternary.Any(temperature >= 100, "red", "blue").(string)
func Any(cond bool, t interface{}, f interface{}) interface{} {
    if cond {
        return t
    }
    return f
}

// Int implements a ternary for int values.
func Int(cond bool, t int, f int) int {
    if cond {
        return t
    }
    return f
}

// String implements a ternary for string values.
func String(cond bool, t string, f string) string {
    if cond {
        return t
    }
    return f
}
```


ternary-demo main.go

```
package main

import (
    "fmt"

    t "github.com/mvolkmann/ternary"
)

func main() {
    temperature := 70
    color := t.String(temperature > 100, "red", "blue")
    fmt.Printf("color = %s\n", color)

    color = t.Any(temperature > 100, "red", "blue").(string)
    fmt.Printf("color = %s\n", color)
    fmt.Printf("color type = %T\n", color) // string
}
```

type assertion

Multiple Major Versions

- Since each version is stored separately and import paths specify the major version to be used, it is possible to use multiple major versions of a dependency in the same application
- But probably not a good idea

Versions Used

- Actual versions of dependencies that are used is determined by `go.mod`
- Several ways to specify a version, called “module queries”
- To add a module query to a `require` path, add a space and one of following

Module Query Type	Example	Notes
fully-specified	<code>v1.2.3</code>	
minor version prefix	<code>v1.2</code>	latest starting with <code>v1.2</code>
major version prefix	<code>v1</code>	latest starting with <code>v1</code>
version comparison	<code>>=v1.2.3</code>	can also use <code>></code> , <code><</code> , and <code><=</code>
latest	<code>latest</code>	
commit hash	<code>A1B2B3D</code>	
tag	<code>my-tag</code>	
branch name	<code>my-branch</code>	

haven't seen any examples using more than one in the same query

Version Details

- Running a command such as `go build` finds a version that matches each module query and updates `go.mod` with results
 - except when using a fully-specified version because then `go.mod` already has the version
- Rerunning commands like `go build` will not automatically get the latest matching versions since previously run commands will have caused **modules queries to be replaced by actual versions**
 - for example, a line like
`require github.com/mvolkmann/ternary latest`
could be replace by
`require github.com/mvolkmann/ternary v0.1.0`

Issue

- **Sam Boyer** is the lead maintainer of **Dep**, a competing dependency management tool for Go
- Sam feels that this feature of Go modules **loses information** about the minimum versions that were deemed compatible
- Further, he feels this makes it **hard to resolve diamond-shaped dependencies** where multiple modules needed by an app depend on different minor versions of other modules
 - ex. `foo` needs `baz/v2.1.0` and `bar` needs `baz/v2.2.0`
- See his talk “We need to talk about Go modules”
 - <https://www.youtube.com/watch?v=7GGr3S41gjM&feature=youtu.be&t=14m55s>

Versions Used

- **No automatic version updates** are ever performed
 - must modify module queries in `go.mod` or bump major version in imports
- Means checking `go.mod` files into a version control repository provides a way to produce **repeatable builds**
- “Version comparison” **gets nearest version** to what is requested, not latest version that matches
 - example, if query is `>=1.2.3` and existing versions include `1.2.2`, `1.2.3`, and `1.2.4`, this will use version `1.2.3`
 - differs from how npm works

Pseudo Versions

- For dependencies that do not currently use semantic versioning, an alternate way to determine whether one version is later than another is needed
- Pseudo versions provide this
- Pseudo versions are strings that have three parts
 - **first:** version in the form *vmajor.minor.something* ← seems to me that this is always *v0.0.0* when the repository is not tagged with a version
 - **second:** commit time in UTC
 - **third:** first 12 characters of commit hash
 - parts are separated by dashes
- *something* in first part is complicated
 - fortunately it's not necessary to think about this because pseudo version strings are automatically generated when dependencies that lack semantic versioning tags are installed

Other go.mod Directives

- Besides **module** and **require** directives, **go.mod** files can also contain **exclude** and **replace** directives
- **exclude** specifies versions of dependencies that cannot be used
- **replace** specifies versions of dependencies that should be replaced by another version
 - one use is to use a local version of a package
 - **replace** *import-path* => *local-directory-path*
- These can be used to avoid using versions that have known bugs or security issues

Using a Local Module

There is some **controversy** over whether packages not in the standard library can have **import paths** whose first part does not contain a **dot**. See <https://github.com/golang/go/issues/27503>.

- An app can import packages from modules that only reside in a local directory, not at a URL or under `GOPATH`
- Example
 - package to import defined in `~/foo/bar`
 - `go mod init foo/bar`
 - application defined in `~/demo`
 - `go mod init demo`
 - edit `go.mod` to add `require` and `replace` directives

```
~/demo/go.mod
module demo

require foo/bar v0.0.0

replace foo/bar => /Users/Mark/foo/bar
```

```
~/foo/bar/go.mod
module foo/bar
```

```
~/foo/bar/bar.go
package bar

import "fmt"

func Hello() {
    fmt.Println("Hello from bar!")
}
```

```
~/demo/main.go
package main

import "foo/bar"

func main() {
    bar.Hello() // Hello from bar!
}
```


Tidying `go.mod` Files

- Over time the list of dependencies in a `go.mod` file can become out of date
 - missing required dependencies How could this happen?
 - including dependencies (or major versions of them) that are no longer used
- **`go mod tidy`**
 - adds missing dependencies to `go.mod`
 - removes unused dependencies from `go.mod`
- Primary purpose is to remove unused dependencies
 - since many other commands such as `go build` also add missing dependencies

Checksums

- Go modules use checksums to verify that downloaded dependency code has not been modified
- Checksums for each dependency are stored in `go.sum`
 - one for the package as a whole and one for each source file
- `go mod verify` reports all directories that hold downloaded module code and contain files that have been modified

Proxies

- Can configure a **proxy server** that hosts Go modules so they are installed from there instead of connecting to public source control repositories
- One reason is to restrict access to only modules that have been vetted
- A Go proxy server is a web server that responds to GET requests for module URLs
- To use one, set **GOPROXY** environment variable to point to the server
- For more information, enter **go help goproxy**

Wrap Up

- Go try modules!

