GO-03 02: Модули и пакеты. Работа с зависимостями, go mod

Описание:

Как и во многих других языках, в Go нам приходится часто использовать сторонние библиотеки. В прошлом задании для работы с такими библиотеками мы использовали утилиту go get. Тогда мы подтянули в наш проект два сторонних пакета.

```
$ go get github.com/fatih/color
$ go get github.com/huandu/xstrings
```

Но мы сделали это в ручном режиме, загрузили в директорию $GOPATH/src/github.com/ каждую библиотеку отдельной командой. Что достаточно неудобно, особенно если в нашем проекте будут десятки сторонних библиотек. Также мы скачивали всегда последнюю версию библиотеки и не имели возможность управлять версиями зависимостей в нашем проекте.
Эти и другие проблемы решают менеджеры зависимостей. В Go за все время его существования использовались разные менеджеры зависимостей, с их списком можно ознакомиться по ссылке PackageManagementTools. Но с выходом версии golang 1.13 основным (и, откровенно говоря, единственным) менеджером зависимостей стал go mod. Он умеет собирать все зависимости в сборку, контролировать версии, использовать прокси репозитории, позволяет очень просто разрабатывать и поддерживать собственные библиотеки в виде модулей и т.д. Все его плюсы и особенности мы рассмотрим в этой части практикума. И конечно же, научимся с ним работать.
Прошлый раз мы запускали свой проект из $GOPATH, то есть проект лежал в директории $GOPATH/src/.

```
└── $GOPATH
    └── src
        └── github.com
            └── fatih
                └── color
                    └── color.go
                    └── ...
        └── gopackages
            └── main.go
            └── wordz
                └── wordz.go
            └── color
                └── color.go
```

Как мы помним, при импорте Go смотрит в директорию $GOPATH/src/ и подключает оттуда пакеты, в том числе и локальные. Так было, например, с импортом нашего тестового локального пакета color.

```
// фрагмент файла main.go
import (
  ...
  ...
  newcolor "gopackages/color"
)
```

Пакет color был импортирован, в итоге, по этому пути - $GOPATH/src/gopackages/color. И именно по этой причине ваш проект должен был лежать строго в базовой директории Go $GOPATH/src/ - чтобы все зависимости и внутренние пакеты могли быть проимпортированы.
Привязываться к определенному рабочему пространству в виде директории $GOPATH достаточно неудобно, поэтому go mod как раз решает и эту проблему. С помощью него код проекта можно будет хранить в любой директории. И кстати, по умолчанию go mod не работает с проектами, которые лежат в стандартном рабочем пространстве $GOPATH.
Давайте создадим новую директорию ~/mygo, которая не находится в $GOPATH.

```
$ mkdir ~/mygo
```

Переместим туда наш пакет gopackages:

```
$ mv gopackages/ ~/mygo/
```

Получаем:

```
~/mygo
    └── gopackages
        ├── main.go
        ├── wordz
        │   └── wordz.go
        └── color
            └── color.go
```

Для наглядной демонстрации проблемы попробуем сейчас запустить наш проект:

```
$ go run main.go
main.go:7:2: cannot find package "gopackages/color" in any of:
     /usr/local/Cellar/go/1.14.2_1/libexec/src/gopackages/color (from
$GOROOT)
     /Users/dhnikolas/go/src/gopackages/color (from $GOPATH)
```

Видим, что Go не может найти пакет color ни в $GOROOT, ни в $GOPATH и ожидаемо выдает панику.
Давайте добавим в проект go mod и тем самым исправим ошибку. Для этого необходимо изменить все относительные пути импорта на абсолютные, так как в go mod относительные пути не поддерживаются! Изменим “./wordz” на “gopackages/wordz” и т.д.
Далее выполним следующие команды:

```
$ cd ~/mygo/gopackages
$ go mod init gopackages
go: creating new go.mod: module gopackages
```

Мы добавили в проект поддержку Go модулей. И собственно, превратили проект в модуль под названием gopackages. В корне появился новый файл go.mod. Давайте рассмотрим его подробнее:
файл go.mod

```
module gopackages

go 1.14
```

- директива module - указывает на название модуля;
- директива go - указывает на версию языка, которую используют файлы в этом модуле.

В дальнейшем мы еще вернемся к этому файлу. А сейчас давайте попробуем выполнить сборку проекта, который только что стал модулем!

```
$ go build
go: finding module for package github.com/fatih/color
go: finding module for package github.com/huandu/xstrings
go: downloading github.com/huandu/xstrings v1.3.2
go: found github.com/fatih/color in github.com/fatih/color v1.9.0
go: found github.com/huandu/xstrings in github.com/huandu/xstrings
v1.3.2
go: downloading github.com/mattn/go-isatty v0.0.11
```

Мы видим интересный вывод. go mod сам нашел все сторонние библиотеки, которые использует программа в директивах import, и скачал их со всеми зависимостями! С включением модулей в проекте у нас изменилась логика работы директивы import. Теперь название модуля является корневой директорией в импорте. То есть, "gopackages/wordz” нужно понимать так: gopackages/ - это корневая директория нашего модуля вне зависимости от того, где расположен сам модуль (в нашем случае это ~/my/gopackages), а wordz - директория пакета внутри модуля. go mod ищет подключаемый пакет по названию модуля в локальной директории. В директории с нативными библиотеками $GOROOT, если нужного пакета там нет, go mod пытается скачать его последнюю версию из удаленного источника, например, github.com/huandu/xstrings. После этого кеширует

скачанную библиотеку в директорию $GOPATH/pkg/mod/ и при следующей сборке использует библиотеку уже из кеша. Теперь вернемся к файлу go.mod.

```
module gopackages

go 1.14

require (
  github.com/fatih/color v1.9.0
  github.com/huandu/xstrings v1.3.2
)
```

В файле появилась директива require, которая говорит о том, какие библиотеки и версии этих библиотек необходимы для сборки проекта. Мы можем контролировать версии, изменяя их в этом файле.
После сборки проекта появился еще один файл go.sum.
Файл go.sum:

```
github.com/fatih/color v1.9.0
h1:8xPHl4/q1VyqGIPif1F+1V3Y3lSmrq01EabUW3CoW5s=
github.com/fatih/color v1.9.0/go.mod
h1:eQcE1qtQxscV5RaZvpXrrb8Drkc3/DdQ+uUYCNjL+zU=
github.com/huandu/xstrings v1.3.2
h1:L18LIDzqlW6xN2rEkpdV8+oL/IXWJ1APd+vsdYy4Wdw=
github.com/huandu/xstrings v1.3.2/go.mod
h1:y5/lhBue+AyNmUVz9RLU9xbLR0o4KIIExikq4ovT0aE=
github.com/mattn/go-colorable v0.1.4
h1:snbPLB8fVfU9iwbbo30TPtbLRzwWu6aJS6Xh4eaaviA=
github.com/mattn/go-colorable v0.1.4/go.mod
h1:U0ppj6V5qS13XJ6of8GYAs25YV2eR4EVcfRqFIhoBtE=
github.com/mattn/go-isatty v0.0.8/go.mod
h1:Iq45c/XA43vh69/j3iqttzPXn0bhXyGjM0Hdxcsrc5s=
github.com/mattn/go-isatty v0.0.11
h1:FxPOTFNqGkuDUGi3H/qkUbQO4ZiBa2brKq5r0l8TGeM=
github.com/mattn/go-isatty v0.0.11/go.mod
h1:PhnuNfih5lzO57/f3n+odYbM4JtupLOxQOAqxQCu2WE=
golang.org/x/sys v0.0.0-20190222072716-a9d3bda3a223/go.mod
h1:STP8DvDyc/dI5b8T5hshtkjS+E42TnysNCUPdjciGhY=
golang.org/x/sys v0.0.0-20191026070338-33540a1f6037/go.mod
h1:h1NjWce9XRLGQEsW7wpKNCjG9DtNlClVuFLEZdDNbEs=
```

В этом файле хранятся хеши модулей, которые использует наш проект. Они нужны, чтобы гарантировать, что состояние библиотеки при первой сборке такое же, как и при последующих. Это необходимо в целях безопасности или во избежание ошибок в версионировании модулей. Оба файла go.mod и go.sum должны быть всегда закоммитчены в репозитории.
Мы рассмотрели основной кейс работы go mod с точки зрения потребителя библиотек) Теперь мы можем его использовать и не задумываться лишний раз о том, какие библиотеки мы скачали, а какие - нет. Go mod сам разберется с этим. Он добавляет все скачанные библиотеки в определенную директорию, и затем все проекты при сборке используют одно общее хранилище этих библиотек. Получается, что код библиотеки скачан один раз, а используется при любой сборке разных проектов. Тем самым нам не приходится держать копию одной и той же библиотеки для каждого проекта.
Дальше рассмотрим еще несколько полезных возможностей, которые предоставляет go mod.
go mod tidy
Эта команда нужна для того, чтобы актуализировать файлы go.mod и go.sum в соответствии с реальными импортами проекта. Например, мы перестанем использовать библиотеку github.com/fatih/color, удалим ее использование из кода, но при этом в файле go.mod ничего не изменится и пакет github.com/fatih/color будет зафиксирован как обязательная зависимость. Чтобы это исправить, нужно запустить команду go mod tidy и go удалит неиспользуемые библиотеки из файлов go.mod и go.sum. Также добавит недостающие библиотеки в эти файлы.
go mod download
Команда позволяет скачать все необходимые зависимости без сборки проекта. Это может быть полезно, если нужно посмотреть код библиотеки. Или для того, чтобы ваша IDE могла подсвечивать синтаксис и показывать подсказки из сторонних библиотек до того момента, как вы соберете проект.
go mod vendor
Во многих других языках менеджеры зависимостей используют директорию vendor для сторонних библиотек. И это иногда может быть нужным. Хотя идеология go mod как раз в обратном, чтобы не копировать код зависимостей из проекта в проект, а хранить его в одном месте и в одном экземпляре. Но при этом есть возможность использовать такую же логику. Команда go mod vendor копирует необходимые зависимости из общего хранилища (по умолчанию - $GOPATH/pkg/mod/) в директорию проекта, в папку vendor. И в таком случае при сборке необходимо будет указать режим vendor (go build -mod vendor), чтобы go искал зависимости в директории vendor, а не в директории $GOPATH/pkg/mod/.
Итак, подведем итог:

- go mod init добавляет систему модулей в проект (файлы go.mod go.sum);
- автоматически скачивает библиотеки, указанные в директиве import;
- изменяет логику работы импорта;
- можно управлять зависимостями и версиями с помощью файла go.mod;
- отвязывает проект от обязательного использования $GOPATH;
- может использоваться в режиме vendor.

Полезные ссылки:

- https://blog.golang.org/using-go-modules
- https://разработка-программ.рф/заметки/система-пакаджей-go/
- https://golang-blog.blogspot.com/2020/02/go-1-14-go-command.html

Задание:

1. Переведите проект из предыдущего задания на Go mod (если вы этого не сделали по ходу этого задания).
2. Уберите использование библиотеки github.com/fatih/color и приведите в соответствие файлы go.mod и go.sum с помощью команды go mod tidy.
3. Измените версию библиотеки github.com/huandu/xstrings на 1.2.1.
4. Соберите проект в режиме vendor, директорию vendor добавьте в .gitignore
5. Выполненное задание поместите в ветку task_02 вашего репозитория.
6. В ответе пришлите ссылку на merge request в ветку master своего проекта ветки 02_task.