Лабораторная работа №1.5
Многомодульные программы на Go

Содержание

1 Теоретические сведения

1.1 Модули и пакеты

Код приложений, написанных на Go, группируется в пакеты (packages), а пакеты группируются в модули (modules). Разработчик определяет для модуля зависимости, то есть информацию о том, что необходимо для его запуска (версию Go и, возможно, набор других модулей, требуемых для работы данного).

По мере работы над модулем и добавления новой функциональности разработчик публикует новые версии модуля. Если существуют другие разработчики, код которых вызывает функции из данного модуля — они могут импортировать обновлённые пакеты модуля.

Рассмотрим в качестве примера создание модуля `greetings` из официальной документации Go, задача которого — вывод приветствия для заданного пользователя.

Для написания такого важного приложения необходимо создать директорию `greetings`, в которую будет помещён исходный код первого модуля. Инициализация, как и в предыдущих работах, выполняется с помощью команды `go mod init`, с аргументом — адресом в интернете, с которого различные инструменты Go смогут загружать модуль. Для простоты мы используем ненастоящий адрес `example.com/greetings`, а в реальном проекте это должен быть адрес репозитория, содержащего код модуля.

```
mkdir greetings
cd greetings
go mod init example.com/greetings
```

В результате команда создаст файл `go.mod` для отслеживания зависимостей кода. В данном случае этот файл будет включать только имя модуля `greetings` и версию Go, которую поддерживает его код. Но по мере добавления зависимостей файл `go.mod` будет перечислять их версии, необходимые для работы модуля. Это обеспечивает воспроизводимость сборок и дает разработчику прямой контроль над тем, какие версии модулей использовать.

Код модуля `greetings` должен быть помещён в файл `greetings.go`, и в документации предлагается следующее содержание:

```
package greetings

import "fmt"

func Hello(name string) string {
    message := fmt.Sprintf("Hi, %v. Welcome!", name)
    return message
}
```

Как можно заметить, этот код содержит функцию `Hello`, которая принимает в качестве аргумента имя и возвращает строку с приветствием. Модуль не содержит функцию `main`, потому что он является вспомогательным модулем, а не самостоятельной программой.

Для модуля, использующего данный, создадим директорию `hello`, находящуюся на том же уровне вложенности, что и директория `greetings`:

```
cd ..
mkdir hello
cd hello
go mod init example.com/hello
```

Код модуля `Hello`, будет обращаться к модулю `greetings`. Он может выглядеть следующим образом:

```
package main

import (
    "fmt"

    "example.com/greetings"
)

func main() {
    message := greetings.Hello("Gladys")
    fmt.Println(message)
}
```

Если бы адрес example.com/greetings был настоящим, он мог бы быть загружен из интернета при сборке проекта `Hello`. Однако, поскольку в реальности такого адреса не существует, мы можем использовать вместо него локальный путь. Для этого требуется *адаптировать* модуль `example.com/hello`, чтобы он мог найти код `example.com/greetings` в локальной файловой системе.

Команда `go mod edit` позволяет отредактировать модуль и перенаправить инструменты Go с его пути (где никакого модуля на самом деле нет) в локальный каталог:

```
go mod edit -replace example.com/greetings=../greetings
```

Разумеется, команду нужно выполнять, когда текущим является каталог редактируемого модуля, то есть директория `hello`. После запуска команды файл `go.mod` в этом каталоге будет включать директиву `replace`:

```
module example.com/hello

go 1.16

replace example.com/greetings => ../greetings
```

Далее необходимо выполнить команду `go mod tidy`, которая синхронизирует зависимости модуля `example.com/hello`, добавив те, которые затребованы в его исходном коде, но еще не отслеживаются непосредственно в модуле.

```
go mod tidy
```

Команда должна выдать сообщение следующего вида:

```
go: found example.com/greetings in example.com/greetings v0.0.0-00010101000000-0000000000000
```

В результате её выполнения файл `go.mod` модуля `hello` должен принять следующий вид:

```
module example.com/hello

go 1.16

replace example.com/greetings => ../greetings

require example.com/greetings v0.0.0-00010101000000-0000000000000
```

Номер, следующий за путем модуля, является *псевдономером версии* — сгенерированным номером, используемым вместо настоящего номера версии (которого у модуля еще нет).

ПРИМЕЧАНИЕ 1: Для ссылки на опубликованный модуль в файле go.mod обычно пропускается директива `replace` и используется директива `require` с помеченным номером версии в конце: например, `require example.com/greetings v1.1.0`.

Для сборки многомодульного приложения в исполняемый файл необходимо выполнить уже знакомую команду `go build` из командной строки, когда текущим является каталог `hello`.

```
go build
```

ПРИМЕЧАНИЕ 2: Если директория, в которой находится модуль, загружена в реальный репозиторий — например, на GitHub по адресу github.com/vasya/greetings, строка модуля в файле `go.mod` должна содержать `module github.com/vasya/greetings`.

Код в `greetings.go` объявляет пакет с помощью `package greetings`, а пользователи могут затем использовать этот пакет, импортировав его в свой код с помощью `import "github.com/someuser/modname"`.

Кроме того, пользователь может установить приложение на свой компьютер из сетевого репозитория командой наподобие `go install github.com/vasya/hello@latest`.

1.2 Пакеты из нескольких файлов

Пакет Go можно разделить на несколько файлов, все из которых находятся в одном каталоге, например:

```
greetings/
  go.mod
  greeting1.go
  greeting2.go
  greeting3.go
```

Все файлы в этой директории должны объявлять свою принадлежность пакету командой `package greetings`.

Базовой исполняемой командой, то есть тем, станет исполняемым файлом, запускаемым пользователем из командной строки, становится файл Go, в котором определена функция `main`. Однако все файлы проекта объявляют `package main`.

1.3 Рабочие пространства

Работа с несколькими модулями в Go может быть достаточно трудоёмкой, если требуется редактировать одновременно несколько модулей, когда один из них зависит от другого.

После редактирования родительского модуля, его необходимо отправить в репозиторий, а затем выполнить обновление в зависящем от него модуле, чтобы загрузить новую версию.

До Go 1.18 это выполнялось с помощью директивы `replace`, использованием локальных версий модулей из файловой системы. Однако в версии 1.18 для более удобной работы над несколькими модулями были введены *рабочие пространства* (workspace).

При этом используется специальный файл `go.work`. Он во многом похож на файл `go.mod`. Назначение файла `go.work` — указать локальное «рабочее пространство» модулей, то есть сказать: «Я работаю над этими наборами модулей вместе, локально».

Рассмотрим создание рабочего пространства на примере структуры проекта, соответствующей примеру выше:

```
greetings/
  go.mod
hello/
  go.mod
```

Рабочее пространство необходимо создать в *родительской* (так проще) директории по отношению к этим двум поддиректориям, выполнив следующую команду:

```
go work init ./greetings ./hello .
```

В результате будет создан файл `go.work` следующего содержания:

```
go 1.18
use (
    ./greetings/
    ./hello/
)
```

Сборка проекта, а также его запуск, должны выполняться с указанием директории, в которой находится рабочее пространство. Например, если это директория `myworkspace`, в которой находятся поддиректории `greetings` и `hello`, команды `go build` и `go run` могут выполняться из директории, родительской по отношению к ней, так:

```
go run ./myworkspace/hello.go
```

2 Задания для самостоятельного выполнения

В ходе выполнения работы требуется:

1. Модифицировать программу с иерархической классификацией объектов реального мира, разработанной в лабораторной работе **0.5** так, чтобы «классы» находились в отдельных модулях, и импортировались в демонстрационную программу, создающую экземпляры этих классов и вызывающую их методы для вывода на экран характеристик объектов. Использовать директиву `replace` для локальной работы с модулями на файловой системе. Привести в отчете файлый `go.mod` для созданных модулей.

2. Модифицировать программу так, чтобы её код размещался в онлайн-репозитории (например, на GitHub). Привести измененные файлы.

3. Выполнить модификацию проекта так, чтобы он использовал рабочее пространство, которому принадлежат все разработанные модули. Привести результирующий файл рабочего пространства, а также команды, используемые для сборки, установки и запуска.