模块化

用任何语言来开发，如果软件规模扩大，会编写大量的函数、结构体、接口等代码。这些代码不可能写在一个文件中，这就会产生大量的文件。如果这些文件杂乱无章，就会造成名称冲突、重复定义、难以检索、无法引用、共享不便、版本管理等一系列问题。有一些功能模块如何复用，如何共享方便其他项目使用。所以，一定要有模块化管理，解决以上诸多问题。

包

- 包由多个文件和目录组成
- 使用`package <包名>`来定义包名
- 包名一般都采用小写，符合标识符要求
- 当前目录名和`package <包名>`中的包名不需要一致，但最好保持一致
- **同级文件归属一个包**，就是说每个包目录的当前目录中，只能统一使用同一个package的包名，否则编译出错

一般来说，开发项目时，可以把相关功能的代码集中放在某个包里面。例如在main包目录中新建一个calc包，将所有计算函数都放在其中，以供别的代码调用。

同一个目录就是同一个包，该包内go文件里的变量、函数、结构体互相可见，可以直接使用。

跨目录就是跨包，使用时需要导入别的包，导入需要指定该包的路径。

包管理

GOPATH

Go 1.11版本之前，项目依赖包存于GOPATH。GOPATH是一个环境变量，指向一个目录，其中存放项目依赖包的源码。

GOPATH默认值是`家目录/go`。

开发的代码放在`GOPATH/src`目录中，编译这个目录的代码，生成的二进制文件放到`GOPATH/bin`目录下。

这会有以下问题

- GOPATH不区分项目，代码中任何import的路径均从GOPATH作为根目录开始。如果有多个项目，不同项目依赖不同库的不同版本，这就很难解决了
- 所有项目的依赖都放在GOPATH中，很难知道当前项目的依赖项是哪些

GOPATH + vendor机制

Go 1.5引入vendor机制。

vendor：将项目依赖包复制到项目下的vendor目录，在编译时使用项目下的vendor目录的包进行编译。

依然不能解决不同项目依赖不同包版本问题

包搜索顺序

- 在当前包vendor目录查找
- 向上级目录查找，直到GOPATH/src/vendor目录
- 在GOPATH目录查找
- 在GOROOT目录查找标准库

Go Modules

Go Modules是从Go 1.11版本引入，到1.13版本之后已经成熟，Go Modules成为官方的依赖包管理解决方案。

优势：

- 不受GOPATH限制，代码可放在任意目录
- 自动管理和**下载**依赖，且可以控制使用版本
- 不允许使用相对导入

GO111MODULE配置

GO111MODULE控制Go Module模式是否开启，有off、on、auto（默认）三个值，auto是默认值。

- `GO111MODULE=on`，支持模块，Go会忽略GOPATH和vendor目录，只根据go.mod下载依赖，在**$GOPATH/pkg/mod**目录搜索依赖包。
  - Go 1.13后默认开启
  - 目前开发已经使用了1.17+版本，可以不配置，默认直接开启
- `GO111MODULE=off`，不支持模块，Go会从GOPATH和vendor目录寻找包
- `GO111MODULE=auto`，在 `$GOPATH/src` 外面构建项目且根目录有go.mod文件时，开启模块支持。否则使用GOPATH和vendor机制

GOPROXY环境变量可以指定包下载镜像

- GOPROXY=https://goproxy.cn,direct
- GOPROXY=https://mirrors.aliyun.com/goproxy/
- GOPROXY=https://mirrors.cloud.tencent.com/go/
- GOPROXY=https://repo.huaweicloud.com/repository/goproxy/

Module模式

go mod命令

在Go 1.11开始引入，可以在任何目录使用go.mod创建项目

- init 当前文件夹下初始化一个新的module, 创建go.mod文件
- tidy 自动分析依赖，下载缺失的模块，移除未使用的模块，并更新go.mod文件

构建Module模式项目

构建项目根目录magtools，并初始化模块`go mod init magedu.com/tools`，会产生go.mod文件，内容如下

```
module magedu.com/tools

go 1.19
```

- module 指定模块名称
- go 指定当前模块使用的Go版本

根目录下新建main.go，内容如下

```
package main

import "fmt"

func main() {
    fmt.Println("hello magedu.com")
}
```

- package 指定包名，同一个目录包名必须相同
- import 导入包。这里是绝对导入，且fmt是标准库中的包
    - 标准库搜索`$GOROOT/src`
- main函数，必须在main包中，且只能有一个main函数。如果要编译成可执行文件，必须要有main函数

导入子包

构建本地子包calc，其下创建calc.go

```
package calc

import "fmt"

// Add首字母大写导出
func Add(x, y int) int {
    fmt.Printf("x and y in calc.Add: %d, %d\n", x, y)
    return x + y
}
```

如何在main.go中使用子包的函数Add呢?

```
import "./calc" // 相对导入，不推荐，会发生错误
import "magedu.com/tools/calc" // 正确，Local Package本地包需要使用Module名/子包路径
// 在VSCode中，代码中输入calc会有导入提示
```

如果在calc下再创建minus/minus.go

```go
package minus

import "fmt"

func Minus(x, y int) int {
    fmt.Printf("x and y in calc/minus.Minus: %d, %d\n", x, y)
    return x - y
}
```

那么main.go中就要如下导入

```go
package main

import (
    "fmt"

    "magedu.com/tools/calc"
    "magedu.com/tools/calc/minus"
    // m "magedu.com/tools/calc/minus"
)

func main() {
    fmt.Println(calc.Add(1, 100))
    fmt.Println(minus.Minus(22, 33))
    // fmt.Println(m.Minus(22, 33))
    fmt.Println("hello magedu.com")
}
```

项目目录结构

```
magtools
├ calc
│ ├ minus
│ │ └ minus.go
│ └ calc.go
├ go.mod
└ main.go
```

import关键字

1、绝对导入

```
import (
    "fmt"

    "magedu.com/tools/calc"
    "magedu.com/tools/calc/minus"
)

// 使用举例
calc.Add(4, 10)
minus.Minus(10, 20)
```

2、别名导入

如果有两个导入的包冲突时，可以重命名包来避免冲突

```
import m "magedu.com/tools/calc/minus"

// 使用举例
m.Minus()
```

3、相对导入

不建议使用

```
import "./calc"
```

4、点导入

不推荐使用。

把包内所有导出成员直接导入到本地。很少使用，很有可能导入的标识符导致冲突。

```
import . "magedu.com/tools/calc/minus"

// 使用举例
Minus()
```

go-staticcheck对于点导入会有警告，`should not use dot imports (ST1001)go-staticcheck`。参看 should not use dot imports (ST1001)go-staticcheck。

5、匿名导入

```
import _ "magedu.com/tools/calc/minus"
```

使用下划线作为别名，就意味着无法使用了，那其目的何在?

这种情况下，只能执行导入的包内的所有init函数了。主要作用是做包的初始化用。

导入本地其它项目

把calc包挪到本地其它目录中，如何导入呢？例如把calc包挪到o:/calc，同时在calc目录使用`go mod init ccc`，打开**增加的go.mod**，内容如下

```
module ccc

go 1.19
```

main.go中的导入和使用，如下

```
package main

import (
    "fmt"

    c "tools/ttt" // 故意随便写了一个包路径
    // 由于包路径的最后一段是ttt，而calc/calc.go里面是package calc，路径和包名不一样，所以要用别名
    "tools/ttt/minus" // 上面的子包
)

func main() {
    fmt.Println(c.Add(111, 1000))
    fmt.Println(minus.Minus(200, 300))
    fmt.Println("hello magedu.com")
}
```

还需要手动在go.mod中增加

```
module magedu.com/tools

go 1.19

require (
    tools/ttt v0.0.0 // 指定伪版本号，满足格式要求
)
replace  tools/ttt => "o:/calc" // replace指令指定包搜索路径，而不是去GOPATH/pkg/mod
```

- 参考 https://golang.google.cn/ref/mod#go-mod-file-require
- require：用于设置一个特定的模块版本
    - // indirect：该注释表示该模块为间接依赖，也就是在当前应用程序中的 import 语句中，并没有发现这个模块的明确引用，有可能是你先手动 go get 拉取下来的，也有可能是你所依赖的模块所依赖的
- exclude：用于从使用中排除一个特定的模块版本
- replace：用于将一个模块版本替换为另外一个模块版本

导入第三方包

在 https://pkg.go.dev/ 查找包Beego

下载

```
go get -u github.com/astaxie/beego

go mod tidy
```

main.go

```
package main

import (
    "fmt"

    c "tools/ttt" // 故意随便写了一个包路径
    // 由于包路径的最后一段是ttt，而calc/calc.go里面是package calc，这里要用别名
    "tools/ttt/minus" // 上面的子包

    "github.com/astaxie/beego"
)

func main() {
    fmt.Println(c.Add(111, 1000))
    fmt.Println(minus.Minus(200, 300))
    fmt.Println("hello magedu.com")
    beego.Run()
}
```

第三方依赖包搜索 `$GOPATH/pkg/mod`

go.mod中

```
require github.com/astaxie/beego v1.12.3

省略很多的间接依赖
```

拉取模块依赖后，会发现多出了一个 go.sum 文件，其详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

init函数

- init函数，无参无返回值，不能被其他函数调用
- 包中的init函数将在main函数之前自动执行
- 每个包中init函数可以有多个，且可以位于不同的文件中
- 一个文件中至多有一个init函数
- 同一个包中的init函数没有明确的执行顺序，不可预期
- 不同包的init函数的执行顺序由导入顺序决定

init函数主要是做一些初始化工作。但是由于同一个包里面init函数执行顺序不可预期，所以，除非有必要，不要在同一个包里面定义多个init。init和main函数不一定在同一个文件中。

`import _ "xxx"`作用是什么？只执行该包的init函数，无法使用包内资源。

`import "xxx"`作用是什么？也会执行该包的init函数，也可以使用包内资源。