启用 go mod 包管理工具

设置GO111MODULE

在Go语言 1.12 版本之前，要启用 go module 工具首先要设置环境变量 GO111MODULE，不过在Go语言 1.13 及以后的版本则不再需要设置环境变量。通过 GO111MODULE 可以开启或关闭 go module 工具。

- GO111MODULE=off 禁用 go module，编译时会从 GOPATH 和 vendor 文件夹中查找包；
- GO111MODULE=on 启用 go module，编译时会忽略 GOPATH 和 vendor 文件夹，只根据 go.mod下载依赖；
- GO111MODULE=auto（默认值），当项目在 GOPATH/src 目录之外，并且项目根目录有 go.mod 文件时，开启 go module

Windows 下开启 GO111MODULE 的命令为：

```
set GO111MODULE=on
# 或者
set GO111MODULE=auto
```

MacOS 或者 Linux 下开启 GO111MODULE 的命令为：

```
export GO111MODULE=on
# 或者
export GO111MODULE=auto
```

在开启 GO111MODULE 之后就可以使用 go module 工具了，也就是说在以后的开发中就没有必要在 GOPATH 中创建项目了，并且还能够很好的管理项目依赖的第三方包信息。

常用的`go mod`命令如下表所示：

命令	作用
go mod init	初始化当前文件夹，创建 go.mod 文件
go mod tidy	增加缺少的包，删除无用的包
go mod download	下载依赖包到本地（默认为 GOPATH/pkg/mod 目录）
go mod edit	编辑 go.mod 文件
go mod graph	打印模块依赖图
go mod vendor	将依赖复制到 vendor 目录下
go mod verify	校验依赖
go mod why	解释为什么需要依赖

GOPROXY

proxy 顾名思义就是代理服务器的意思。大家都知道，国内的网络有防火墙的存在，这导致有些Go语言的第三方包我们无法直接通过 `go get` 命令获取。GOPROXY 是Go语言官方提供的一种通过中间代理商来为用户提供包下载服务的方式。要使用 GOPROXY 只需要设置环境变量 GOPROXY 即可

目前公开的代理服务器的地址有：

goproxy	提供商
goproxy.io	微软
goproxy.cn	七牛云
mirrors.aliyun.com/goproxy	阿里云
gocenter.io	GoCenter

Windows 下设置 GOPROXY 的命令为：

```
go env -w GOPROXY=https://goproxy.cn,direct
```

MacOS 或 Linux 下设置 GOPROXY 的命令为：

```
export GOPROXY=https://goproxy.cn
```

> Go语言在 1.13 版本之后 GOPROXY 默认值为 https://proxy.golang.org，在国内可能会存在下载慢或者无法访问的情况，所以十分建议大家将 GOPROXY 设置为国内的 goproxy.cn

使用go get命令下载指定版本的依赖包

执行`go get`命令，在下载依赖包的同时还可以指定依赖包的版本。

- 运行`go get -u`命令会将项目中的包升级到最新的次要版本或者修订版本；
- 运行`go get -u=patch`命令会将项目中的包升级到最新的修订版本；
- 运行`go get [包名]@[版本号]`命令会下载对应包的指定版本或者将对应包升级到指定的版本。

提示：`go get [包名]@[版本号]`命令中版本号可以是 x.y.z 的形式，例如 go get foo@v1.2.3，也可以是 git 上的分支或 tag，例如 go get foo@master，还可以是 git 提交时的哈希值，例如 go get foo@e3702bed2。

在项目中应用

创建一个项目 “golang-examples”

1. 在 GOPATH 目录之外新建一个目录，并使用`go mod init`初始化生成 go.mod 文件

```
go mod init golang-examples
go: creating new go.mod: module golang-examples
```

初始化生成的 go.mod 文件如下所示：

```
module golang-examples

go 1.15
```

2. 新建一个 main.go 文件，写入以下代码：

```
package main

import "time"
import log "github.com/sirupsen/logrus"
import http "github.com/valyala/fasthttp"

const (
```

```
    imgList            = "https://k8s.gcr.io/v2/tags/list"
    DefaultHttpTimeout = 15 * time.Second
    repo               = "k8s.gcr.io/"
)

func main() {
    log.Info("get k8s.gcr.io public images...")
    status, body, errs := http.Get(nil, imgList)
    if errs != nil {
        log.Error("request error", errs)
    }
    if status != http.StatusOK {
        log.Warn("request faild")
    }
    log.Info(body)
}
```

执行`go run main.go`运行代码会发现 go mod 会自动查找依赖自动下载

现在查看 go.mod 内容：

```
module golang-examples

go 1.15

require (
    github.com/sirupsen/logrus v1.8.1
    github.com/valyala/fasthttp v1.28.0
)

```

go 会自动生成一个 go.sum 文件来记录 dependency tree
可以使用命令`go list -m -u all`来检查可以升级的 package，使用`go get -u need-upgrade-package`升级后会将新的依赖版本更新到 go.mod * 也可以使用`go get -u`升级所有依赖。

使用 replace 替换无法直接获取的 package

由于某些已知的原因，并不是所有的 package 都能成功下载，比如：golang.org 下的包。

modules 可以通过在 go.mod 文件中使用 replace 指令替换成 github 上对应的库，比如：

```
replace (
    golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a =>
github.com/golang/crypto v0.0.0-20190313024323-a1f597ede03a
)
```

或者

```
replace golang.org/x/crypto v0.0.0-20190313024323-a1f597ede03a =>
github.com/golang/crypto v0.0.0-20190313024323-a1f597ede03a
```

build

```
go build [-o 输出名] [-i] [编译标记] [包名]
```

> 如果参数为 *.go文件或文件列表，则编译为一个个单独的包。
> 当编译单个main包（文件），则生成可执行文件。
> 当编译单个或多个包非主包时，只构建编译包，**但丢弃生成的对象（.a）**，仅用作检查包可以构建。
> 当编译包时，会自动忽略'_test.go'的测试文件。

-o

> output 指定编译输出的名称，代替默认的包名。

-i

> install 安装作为目标的依赖关系的包(用于增量编译提速)。

以下 build 参数可用在 build, clean, get, install, list, run, test

```
-a
    完全编译，不理会-i产生的.a文件(文件会比不带-a的编译出来要大？)
-n
    仅打印输出build需要的命令，不执行build动作（少用）。
-p n
    开多少核cpu来并行编译，默认为本机CPU核数（少用）。
-race
    同时检测数据竞争状态，只支持 linux/amd64, freebsd/amd64,
darwin/amd64 和 windows/amd64.
-msan
    启用与内存消毒器的互操作。仅支持linux / amd64，并且只用Clang / LLVM
作为主机C编译器（少用）。
-v
    打印出被编译的包名（少用）.
-work
    打印临时工作目录的名称，并在退出时不删除它（少用）。
-x
    同时打印输出执行的命令名（-n）（少用）.
```

```
-asmflags 'flag list'
    传递每个go工具asm调用的参数（少用）
-buildmode mode
    编译模式（少用）
    'go help buildmode'
-compiler name
    使用的编译器 == runtime.Compiler
    (gccgo or gc)（少用）.
-gccgoflags 'arg list'
    gccgo 编译/链接器参数（少用）
-gcflags 'arg list'
    垃圾回收参数（少用）.
-installsuffix suffix
    a suffix to use in the name of the package installation directory,
    in order to keep output separate from default builds.
    If using the -race flag, the install suffix is automatically set to race
    or, if set explicitly, has _race appended to it.  Likewise for the -msan
    flag.  Using a -buildmode option that requires non-default compile flags
    has a similar effect.
-ldflags 'flag list'
    '-s -w': 压缩编译后的体积
    -s: 去掉符号表
    -w: 去掉调试信息，不能gdb调试了
-linkshared
    链接到以前使用创建的共享库
    -buildmode=shared.
-pkgdir dir
    从指定位置，而不是通常的位置安装和加载所有软件包。例如，当使用非标准配置构建时，使用-pkgdir将生成的包保留在单独的位置。
-tags 'tag list'
    构建出带tag的版本.
-toolexec 'cmd args'
    a program to use to invoke toolchain programs like vet and asm.
    For example, instead of running asm, the go command will run
    'cmd args /path/to/asm <arguments for asm>'.
```

以上命令，单引号/双引号均可。

对包的操作`'go help packages'`
对路径的描述`'go help gopath'`
对 C/C++ 的互操作`'go help c'`

构建遵守某些约定(`'go help gopath'`),但不是所有的项目都遵循这些约定，当使用自己的惯例或使用单独的软件构建系统时可以选择使用较低级别的调用`go tool compile`和`go tool link`来避免一些构建工具的开销和设计决策