第 5 章

企业级安全项目开发实践

5.1 从零快速掌握 Go 基础开发

5.1.1 Go 环境安装

(1) 下载 Go 的安装包,下载网址如下:

```
https://studyGo.com/dl
https://Go.org/dl/
```

其中第 1 个网址在国内访问快一些,打开时会出现类似如图 5-1 所示的页面,随着版本的更新,看到的版本信息可能不一样。选择一个稳定版本下载并安装,或者根据业务的需要,选择版本进行安装,除了最新版本,历史版本在页面上也是可以查看的。

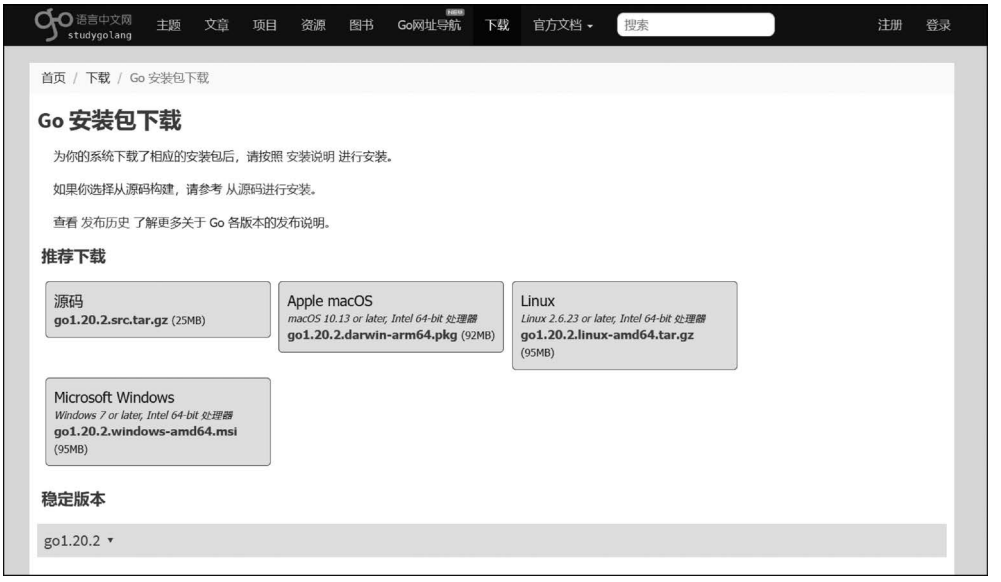


图 5-1 Go 安装包下载页面

（2）根据自己的计算机操作系统类型选择对应的安装包，下载安装包后双击进行安装，安装路径中不要出现中文或者空格。

如图 5-2 所示，Go 环境安装成功后，对应安装目录还有很多文件夹和文件，下面简要说明其中主要文件夹的功用。

名称	修改日期	类型	大小
api	2023/4/4 7:06	文件夹	
bin	2023/4/4 7:06	文件夹	
doc	2023/4/4 7:06	文件夹	
lib	2023/4/4 7:05	文件夹	
misc	2023/4/4 7:06	文件夹	
pkg	2023/4/4 7:05	文件夹	
src	2023/4/4 7:06	文件夹	
test	2023/4/4 7:06	文件夹	
codereview.cfg	2023/3/3 18:19	CFG 文件	1 KB
CONTRIBUTING.md	2023/3/3 18:19	Markdown File	2 KB
LICENSE	2023/3/3 18:19	文件	2 KB
PATENTS	2023/3/3 18:19	文件	2 KB
README.md	2023/3/3 18:19	Markdown File	2 KB
SECURITY.md	2023/3/3 18:19	Markdown File	1 KB
VERSION	2023/3/3 18:19	文件	1 KB

图 5-2　Go 安装目录文件

api 文件夹：用于存放依照 Go 版本顺序的 API 增量列表文件。这里所讲的 API 包含公开的变量、常量、函数等。这些 API 增量列表文件用于 Go 语言 API 检查。

bin 文件夹：用于存放主要的标准命令文件，包括 go、godoc 和 gofmt。

blog 文件夹：用于存放官方博客中的所有文章，这些文章都是 Markdown 格式的。

doc 文件夹：用于存放标准库的 HTML 格式的程序文档。可以通过 godoc 命令启动一个 Web 程序展现这些文档。

lib 文件夹：用于存放一些特殊的库文件。

misc 文件夹：用于存放一些辅助类的说明和工具。

pkg 文件夹：用于存放安装 Go 标准库后的所有归档文件。如果是 Windows 操作系统，则会发现其中有名称为 windows_amd64 的文件夹，称为平台相关目录。可以看到，这类文件夹的名称由对应的操作系统和计算架构的名称组合而成。通过 go install 命令，Go 程序（这里指标准库中的程序）会编译成平台相关的归档文件并存放到其中。另外，pkg/tool/windows_adm64 文件夹存放了使用 Go 制作软件时用到的很多强大命令和工具。

src 文件夹：用于存放 Go 自身，如 Go 标准工具及标准库的所有源码文件。深入探究 Go，就靠它了。

test 文件夹：存放用来测试和验证 Go 本身的所有相关文件。

（3）检查 Go 环境安装是否成功，在命令行中输入命令，命令如下：

```
go env
```

如果出现类似如图 5-3 所示的信息，则表示安装成功。

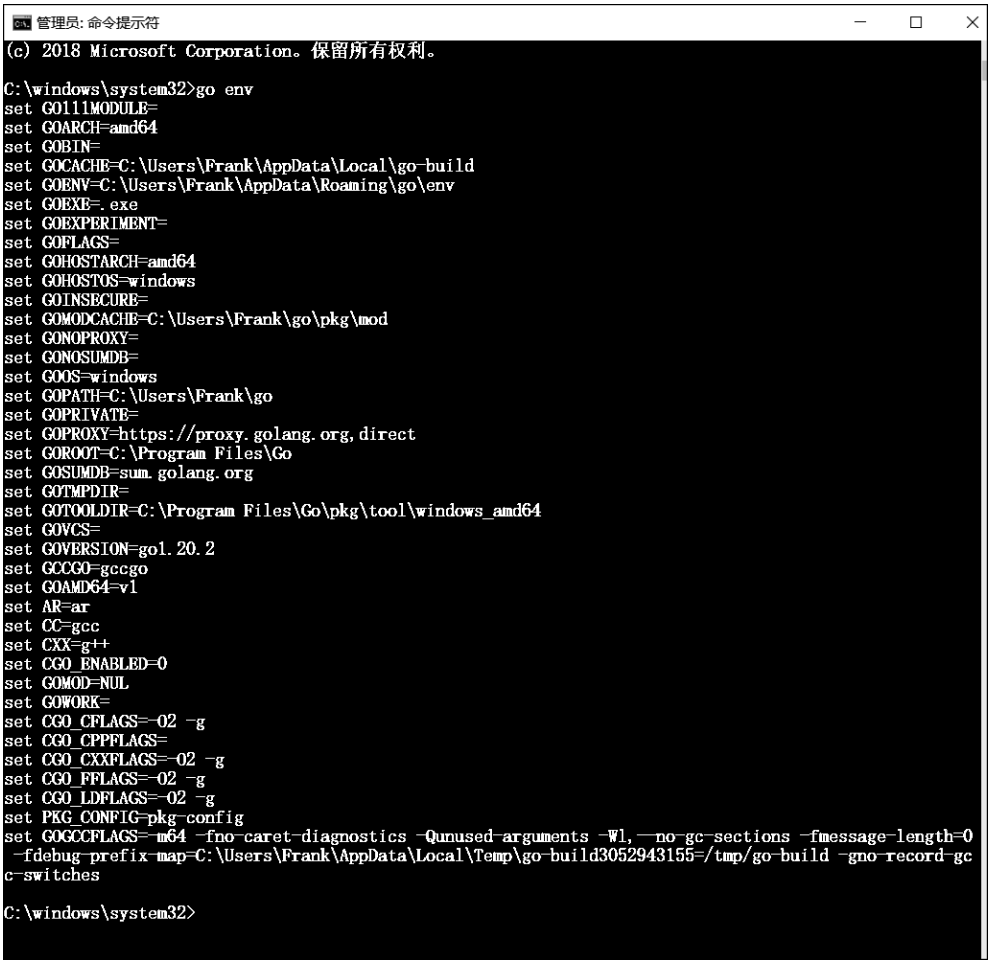


图 5-3　Go 环境检测

5.1.2　Go 开发环境安装

1. 选择 Go 的开发工具下载并安装

GoLand：GoLand 是 Jetbrains 家族的 Go 语言收费版的 IDE，有 30 天的免费试用期。

LiteIDE：LiteIDE 是一款开源、跨平台的轻量级 Go 语言集成开发环境(IDE)。

其他开发工具虽然集成了 Go 语言开发环境，但并没有以上两者专业，例如 Eclipse、VS Code 等。

2. 配置 Go 的工作区 GOPATH

GOROOT：GOROOT 的值应该是安装 Go 的根目录。

GOPATH：需要将工作区的目录路径添加到环境变量 GOPATH 中。否则即使处于

同一个工作区(事实上,未被加入 GOPATH 中的目录不应该称为工作区),代码之间也无法通过绝对代码包路径调用。

在实际开发环境中,工作区可以只有一个,也可以有多个,这些工作区的目的路径都需要添加到 GOPATH 中。与 GOROOT 意义相同,应该确保 GOPATH 一直有效。GOPATH 中不要包含 Go 语言的根目录(GOROOT),以便将 Go 语言本身的工作区与用户工作区严格分开。

通过 Go 工具中的代码获取命令 go get,可将指定项目的源码下载到在 GOPATH 中设定的第 1 个工作区中,并在其中完成编译和安装。

3. 以 GoLand 为例,创建工作区目录

需要注意的是,只有被加入 GOPATH 环境变量中的目录才能被称为 Go 的工作区目录,如图 5-4 所示,创建项目选择的工作目录需要和 GOPATH 系统配置的一致。一般情况下,Go 源码文件必须放在工作区中,但是对于命名源码文件来讲,这不是必需的。工作区其实就是一个对应于特定工厂的目录,它应该包含 3 个子目录,即 src 目录、pkg 目录和 bin 目录。

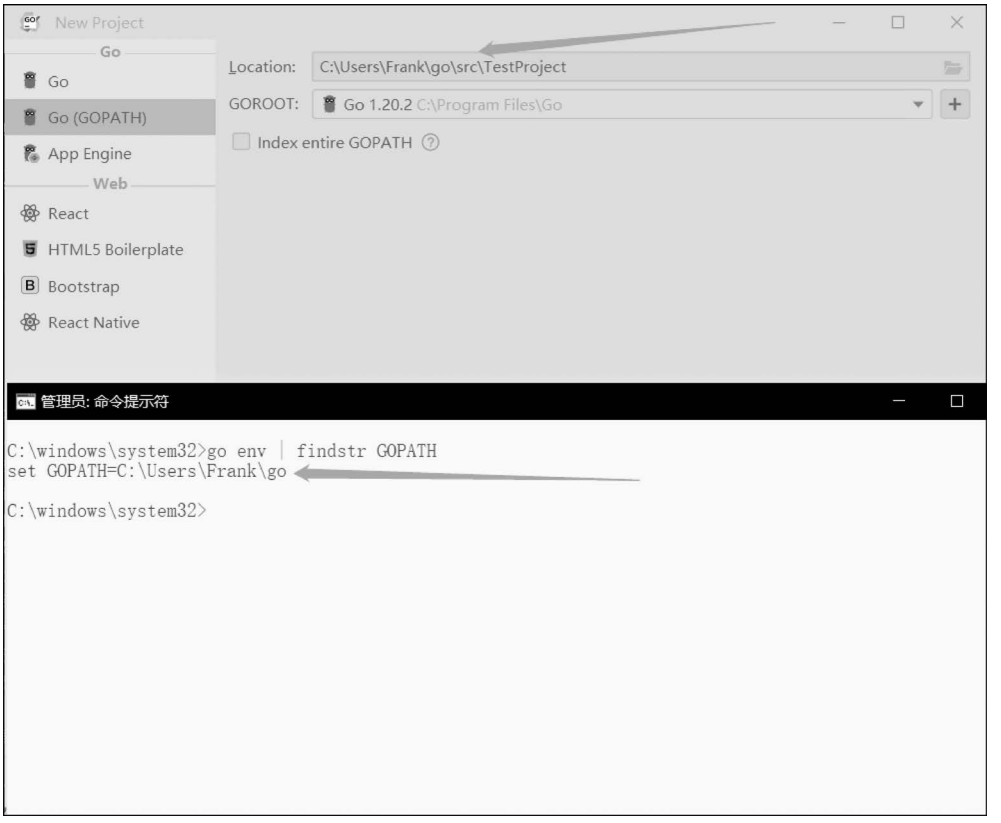


图 5-4　Go 工作区目录

接下来对 GOPATH 指定的工作目录的 3 个子目录功能分别进行说明。

src 目录：用于以代码包的形式组织并保存 Go 源码文件，这里的代码包与 src 下的子目录一一对应。例如，若一个源码文件被声明属于代码包 log，那么它就应当保存在 src/log 目录中。当然，也可以把 Go 源码文件直接存放在 src 目录下，但这样 Go 源码文件就只能被声明属于 main 代码包了。除非用于临时测试或演示，一般还是建议把 Go 源码文件放入特定的代码中。

pkg 目录：用于存放通过 go install 命令安装后的代码包的归档文件，前提是代码包中必须包含 Go 库源码文件。另外，归档文件是指那些名称以.a 结尾的文件。该目录与 GOROOT 目录下的 pkg 目录功能类似。区别在于，工作区中的 pkg 目录专门用来存放用户代码的归档文件。编译和安装用户代码的过程一般会以代码包为单位进行。例如 log 包被编译安装后，将生成一个名为 log.a 的归档文件，并存放在当前工作区的 pkg 目录下的平台相关目录中。

bin 目录：与 pkg 目录类似，在通过 go install 命令完成安装后，保存由 Go 命令源码文件生成的可执行文件。在类 UNIX 操作系统下，这个可执行文件一般来讲名称与源码文件的主文件名相同，而在 Windows 操作系统下，这个可执行文件的名称则是源码文件主文件名加.exe 后缀。

Go 语言的命令源码文件和库源码文件的区别如下。

命名源码文件：指的是声明属于 main 代码包并且包含无参数声明和结果声明的 main 函数的源码文件。这类源码文件是程序的入口，它们可以独立运行(使用 go run 命令)，也可以通过 go build 或 go install 命令得到相应的可执行文件。

库源码文件：指的是在某个代码包中的普通源码文件。

4. 编写第 1 个 Go 程序

程序代码如下：

```
//anonymous - link\example\chapter5\helloworld.go
package main                              //命令行源码文件必须在这里声明自己属于 main 包
/*
    使用 import 关键字导入包，建议每导入一个包占用一行，看起来比较美观
*/
import (
    "fmt"
)

func main() {
    fmt.Println("hello world")          //打印字符串并换行
}
```

运行程序，如图 5-5 所示，可打印出预期的内容。

5. 命令行运行 Go 程序

如图 5-6 所示，单击 Terminal 选项，可以在命令行中运行程序，命令如下：

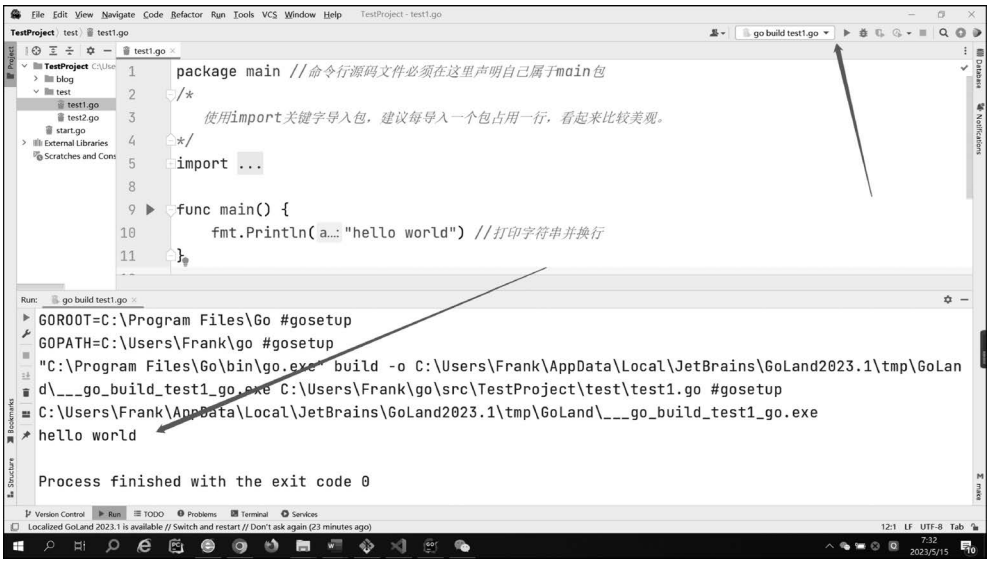


图 5-5　用 GoLand 编写第 1 个 Go 程序

```
go run .\test\test1.go
```

或者先编译，后运行，命令如下：

```
go build .\test\test1.go

.\test1.exe
```

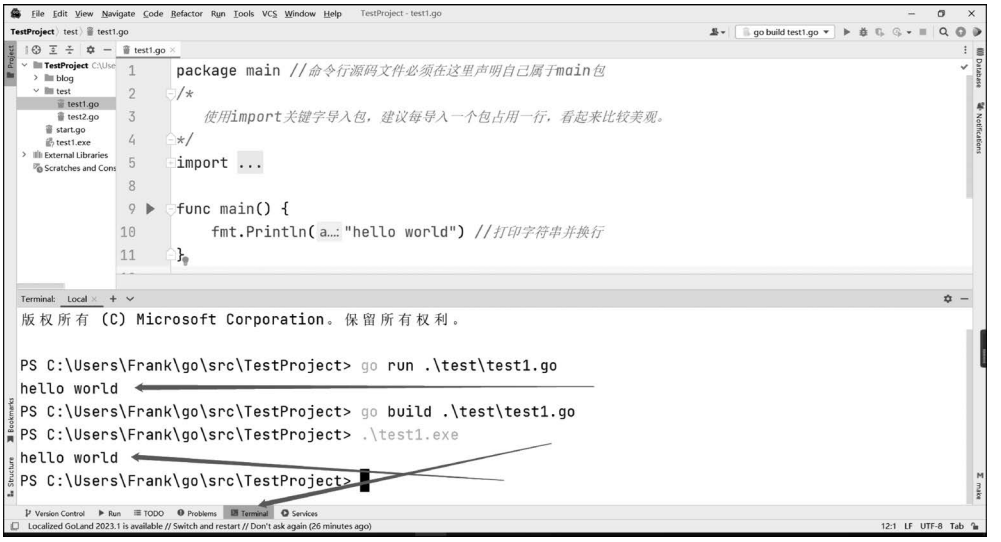


图 5-6　命令行运行 Go 程序

5.1.3 Go 常用的子命令

Go 本身包含了大量用于处理 Go 程序的命令和工具。go 命令就是其中最常见的一个，它有许多子命令。

1. go build

用于编译指定的代码，包括 Go 语言源码文件。命令源码文件会编译生成可执行文件，并存放在命令指令的目录或指定目录下，而库源码文件被编译后，则不会在非临时目录中留下任何文件。

查看帮助信息，命令如下：

```
go help build
```

使用场景主要用于将 go 源码文件编译为二进制可执行文件。

2. go clean

用于清理因执行其他 go 命令而一路留下来的临时目录和文件。

查看帮助信息，命令如下：

```
go help clean
```

使用场景主要用于清理以前编译生成的程序文件。

3. go doc

用于显示 Go 语言代码包及程序实体的文档。

查看帮助信息，命令如下：

```
go help doc
```

如查看 bufio 的使用文档，命令如下：

```
go doc bufio
```

4. go env

用于打印与 Go 语言相关的环境信息。

查看帮助信息，命令如下：

```
go help env
```

使用场景用于查看当前 Go 环境变量的一些配置。

5. go fix

用于修正指定代码包中的源码文件中包含的过时语法和代码调用。这使在升级 Go 语言版本时，可以非常方便地同步升级程序。

查看帮助信息，命令如下：

```
go help fix
```

6. go fmt

用于格式化指定代码包中的 Go 源码文件。实际上，它是通过执行 go fmt 命令实现功能的。

查看帮助信息，命令如下：

```
go help fmt
```

7. go generate

用于识别指定代码中源文件中的 go：generate 注解，并执行其携带的任意命令。该命令独立于 Go 语言标准的编译和安装体系。如果有需要解析的 go：generate 注解，就单独运行它。这个命令非常有用，可以用它自动生成或改动源码文件。

查看帮助信息，命令如下：

```
go help generate
```

test2.go 文件的内容如下：

```
//anonymous - link\example\chapter5\generate.go
package main

import (
    "fmt"
)

//go:generate go run test2.go
func main() {
    fmt.Println("博客地址：https://blog.csdn.net/u014374009/")
}
```

执行命令，结果如图 5-7 所示。

8. go get

用于下载，编译并安装指定的代码包及其依赖包。当需要从代码中转站或第三方代码库上自动拉取代码时，就全靠它了。

查看帮助信息，命令如下：

```
go help get
```

使用举例，将 beego 开源库下载到本地，使用该命令的前提是操作系统已安装好了 git 环境，命令如下：

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

图 5-7 go generate 使用举例

9. go install

用于编译并安装指定的代码包及其依赖包。安装包命令源码文件后，代码包所在(GOPATH 环境变量中定义)的工作区目录的 bin 子目录或者当前环境变量 GOBIN 指向的目录中会生成相应的可执行文件，而安装库源码文件后，会在代码包所在的工作目录的 pkg 子目录生成相应的归档文件。

查看帮助信息，命令如下：

```
go help install
```

10. go list

用于显示指定代码包的信息，它可谓是代码分析的一大便捷工具。利用 Go 语言标准代码库代码包 text/template 中规定的模板语法，可以非常灵活地控制输出信息。

查看帮助信息，命令如下：

```
go help list
```

11. go run

用于编译并允许指定命令源码文件。当不想生成可执行文件而直接运行命令源码文件时，就需要使用它。

查看帮助信息，命令如下：

```
go help run
```

12. go test

用于测试指定的代码包，前提是该代码包目录中必须存在测试源码文件。

查看帮助信息，命令如下：

```
go help test
```

13. go tool

用于运行一些特殊的 Go 语言工具，直接执行 go tool 命令，可以看到这些特殊工具。它们有的是其他 Go 标准命令的底层支持，有的是可以独当一面的利器，其中有两个值得特别介绍，即 pprof 和 trace。

pprof：用于以交互的方式访问一些性能概要文件。命令将会分析给定的概要文件，并根据要求提供高可读性的输出信息。这个工具可以分析概要文件，包括 CPU 概要文件、内存概要文件和程序阻塞概要文件。这些包含 Go 程序运行信息的概要文件，可以通过标准代码库代码包 runtime 和 runtime/pprof 中的程序来生成。

trace：用于读取 Go 程序的踪迹文件，并以图形化的方式展现出来。它能够让用户深入了解 Go 程序在运行过程中的内部情况。例如，当前进程中堆的大小及使用情况。再例如，程序的多个 goroutine 是怎样被调度的，以及它们在某个时刻被调度的原因。Go 程序踪迹文件可以通过标准库代码包 runtime/trace 和 net/http/pprof 中的程序来生成。

上述两个特殊工具对于 Go 程序调优非常有用，如果想要探究程序运行的过程，或者想要让程序运行得更快，更稳定，则这两个工具是必知必会的。另外，这两个工具都被 go test 命令直接支持，因此可以很方便地把它们融入程序测试环境中。

查看帮助信息，命令如下：

```
go help tool
```

14. go vet

用于检查指定代码包中的 Go 语言源码，并报告发现的可疑代码问题。该命令提供了除编译以外的一个程序检查方法，可用于找到程序中的潜在错误。

查看帮助信息，命令如下：

```
go help vet
```

使用案例，如检查 test2.go 文件代码是否有问题，命令如下：

```
go vet .\test\test1.go
```

15. go version

用于显示当前安装的 Go 语言的版本及计算环境。

查看帮助信息，命令如下：

```
go help version
```

16. go 命令通用的标记

当执行上述命令时，可以通过附加一些额外的标记来定制命令的执行过程。下面是比较通用的标记。

-a：用于强行重写编译所涉及的 Go 语言代码包，包括 Go 语言标准库中的代码包(即使它们已经是最新的了)。该标记可以让用户有机会通过改动更底层的代码包来做一些试验。

-n：使命令仅打印其在执行过程中用到的所有命令，而不真正执行它们。如果只想查看或验证命令的执行过程，而不想改变任何东西，则使用它正合适。

-race：用于检测并报告指定 Go 语言程序中存在的数据竞争问题。当用 Go 语言编写并发程序时，这是很重要的检测手段之一。

-v：用于打印命令在执行过程中涉及的代码包。这一定包含用户指定的目标代码包，并且有时还会包括该代码直接或间接依赖的那些代码包，也会知道哪些代码包被命令行处理过了。

-work：用于打印命令执行时生成和使用的临时工作目录的名字，并且命令执行完成后不删除它。这个目录下的文件可能有用，也可以从侧面了解命令的执行过程。如果不添加此标记，则临时工作目录会在命令执行完毕前删除。

-x：使命令打印其执行过程用到的所有命令，同时执行它们。

这些标记看作命令的特殊参数，它们都可以添加到命令名称和命名的真正参数中。用于编译、安装、运行和测试 Go 语言代码包或源码文件的命令都支持它们。

5.1.4 Go 的标识符命名规则

(1) Go 语言的 25 个关键字如下：

```
break, default, func, interface, select, case, defer, go, map, struct, chan, else, goto, package,
switch,const,fallthrough,if,range,type,continue,for,import,return,var
```

(2) Go 语言关键字的用途及解释如下。

var 和 const：用于变量和常量的声明。

package 和 import：用于包导入。

func：用于定义函数和方法。

return：用于从函数返回。

defer：用于在函数退出之前执行。

go：用于并行。

select：用于选择不同类型的通信。

interface：用于定义接口。

struct：用于定义抽象数据类型。

break、case、continue、for、fallthrough、else、if、switch、goto、default：用于流程控制。

chan：用于 channel 通信。

type：用于声明自定义类型。

map：用于声明 map 类型数据。

range：用于读取 slice、map、channel 数据。

(3) Go 语言有 36 个预定义的名字。

在 Go 语言中有很多预定义的名字，基本在内建的常量、类型和函数中，这些内部预定义的名字并不是关键字，它们是可以重新定义的。

Go 语言 36 个预定义的名字如下：

```
append,bool,byte,cap,close,complex,complex64,complex128,uintptr,copy,false,true,float32,
float64,imag,iota,int,uint,int8,uint8,int16,uint16,int32,uint32,int64,uint64,new,len,
make,panic,nil,print,println,real,recover,string
```

(4) Go 语言命名规则，标识符的命名规则如下：

① 允许使用字母、数字、下画线。

② 不允许使用 Go 语言关键字。

③ 不允许使用数字开头。

④ 区分大小写。

满足上面的 Go 编译器的要求后，生产环境中推荐的命名规则如下。

① 见名知义：自定义的变量名称最好能见名知义，增加代码的可读性，如果定义了一堆变量却不知道写的是什么意思，不方便调试，并且阅读非常困难。

② 驼峰命名法：

小驼峰式命名法(Lower Camel Case)：第 1 个单词以小写字母开始，从第 2 个单词开始首字母大写，例如 myNginxPort。

大驼峰式命名法(Upper Camel Case)：每个单字的首字母都采用大写字母，例如 FirstName、LastName。

③ 下画线命名法：每个单词都小写，各单词之间使用下画线进行分隔，例如 my_cluster。

命令规范案例，代码如下：

```
//anonymous - link\example\chapter5\naming.go
package main

import (
    "fmt"
)

func main() {

    /*
        标识符的命名规则如下：
            (1)允许使用字母、数字、下画线
            (2)不允许使用 Go 语言关键字
```

```
            (3)不允许使用数字开头
            (4)区分大小写

        满足上面的 Go 编译器的要求后,在生产环境中推荐的命名规则:
            (1)见名知义
            (2)驼峰命名法
                小驼峰式命名法:
                    第 1 个单词以小写字母开始,从第 2 个单词开始首字母大写,例如 myNginxPort
                大驼峰式命名法:
                    每个单字的首字母都采用大写字母,例如 FirstName、LastName
            (3)下画线命名法
                    每个单词都小写,各单词之间使用下画线进行分隔,例如 my_cluster
    */

    //小驼峰命名
    myNginxPort := "node101.test.org.cn:80"
    fmt.Println(myNginxPort)

    //大驼峰命名
    FirstName := "yang"
    LastName := ""
    fmt.Println(FirstName)
    fmt.Println(LastName)

    //下画线命名
    my_cluster := "yangyi_cluster"
    fmt.Println(my_cluster)

}
```

5.1.5 Go 编程的工程管理

1. 工作区概述

GOROOT：GOROOT 的值应该是安装 Go 的根目录。

GOPATH：需要将工作区的目录路径添加到环境变量 GOPATH 中。否则即使处于同一个工作区(事实上,未被加入 GOPATH 中的目录不应该称为工作区),代码之间也无法通过绝对代码包路径调用。

在实际开发环境中,工作区可以只有一个,也可以有多个,这些工作区的目的路径都需要添加到 GOPATH 中。与 GOROOT 一致,应该确保 GOPATH 一直有效。

GOPATH 中不要包含 Go 语言的根目录(GOROOT),以便将 Go 语言本身的工作区与用户工作区严格分开。

通过 Go 工具中的代码获取命令 go get,可将指定项目的源码下载到 GOPATH 中设定的第 1 个工作区中,并在其中完成编译和安装。

一般情况下,Go 源码文件必须放在工作区中,但是对于命名源码文件来讲,这不是必需的。工作区其实就是一个对应于特定工厂的目录,它应该包含 3 个子目录,即 src 目录、pkg

目录和 bin 目录。

接下来对 GOPATH 指定的工作目录的 3 个子目录的功能分别进行说明。

(1) src 目录：用于以代码包的形式组织并保存 Go 源码文件，这里的代码包与 src 下的子目录一一对应。例如，若一个源码文件被声明属于代码包 log，那么它就应当被保存在 src/log 目录中。

当然，也可以把 Go 源码文件直接放在 src 目录下，但这样 Go 源码文件就只能被声明属于 main 代码包了。除非用于临时测试或演示，一般还是建议把 Go 源码文件放入特定的代码中。

(2) pkg 目录：用于存放通过 go install 命令安装后的代码包的归档文件，前提是代码包中必须包含 Go 库源码文件。另外，归档文件是指那些名称以.a 结尾的文件。该目录与 GOROOT 目录下的 pkg 目录功能类似。区别在于，工作区中的 pkg 目录专门用来存放用户代码的归档文件。

编译和安装用户代码的过程一般会以代码包为单位。例如 log 包被编译安装后，将生成一个名为 log.a 的归档文件，并存放在当前工作区的 pkg 目录下与平台相关的目录中。

(3) bin 目录：与 pkg 目录类似，在通过 go install 命令完成安装后，保存由 Go 命令源码文件生成的可执行文件。在类 UNIX 操作系统下，这个可执行文件一般来讲名称与源码文件的主文件名相同，而在 Windows 操作系统下，这个可执行文件的名称则是源码文件主文件名加.exe 后缀。目录 src 用于包含所有的源代码，是 Go 命令行工具的一个强制的规则，而 pkg 和 bin 则无须手动创建，如果必要，则 Go 命令行工具在构建过程中会自动创建这些目录。

(4) 命名源码文件：指的是声明属于 main 代码包并且包含无参数声明和结果声明的 main 函数的源码文件。这类源码文件是程序的入口，它们可以独立运行(使用 go run 命令)，也可以通过 go build 或 go install 命令得到相应的可执行文件。

综上所述，可以总结为如果一个源码文件被声明属于 main 代码包，并且该文件中包含无参数声明和结果声明的 main 函数，则它就是命名源码文件。命名源码文件可通过 go run 命令直接运行。

(1) 库源码文件：指的是在某个代码包中的普通源码文件。

通常，库源码文件声明的包名会与它直接所述的代码包(目录)名一致，并且库源码文件中不包含无参数声明和无结果声明的 main 函数。

(2) 测试源码文件：测试源码文件是一种特殊的库文件，可以通过 go test 命令运行当前代码包下的所有测试源码文件。成为测试源码文件的充分条件有以下两个：

① 文件名需要以_test.go 结尾。

② 文件中需要至少包含一个名称以 Test 开头或 Benchmark 开头且拥有一种类型为 *testing.T 或 *testing.B 的参数的函数(testing.T 和 testing.B 是两种结构体类型，而 *testing.T 和 *testing.B 则分别为前两者的指针类型。它们分别是功能测试和基准测试所需的)；Go 代码的文本文件需要以 UTF-8 编码存储。如果源码文件中出现了非 UTF-8

编码的字符，则在运行、编译或安装时，Go 命令会抛出 illegal UTF-8 sequence 错误提示。

2. 多文件编程

blog 包中的 login.go 文件，代码如下：

```
//anonymous - link\example\chapter5\blog\login.go
package blog

import (
    "fmt"
)
/*
    函数名称首字母大写，可以被其他包访问
*/
func Login() {
    fmt.Println("login successful")
}
/*
函数名称首字母小写，不可以被其他包访问
*/
func sayHello() {
    fmt.Println("Hi")
}
/*
    函数名称首字母大写，可以被其他包访问 */
func SayHello() {
    fmt.Println("Hello")
}
```

start.go 文件中的代码如下：

```
//anonymous - link\example\chapter5\blog\login.go
package main

import (
    "blog"
)

func main() {
    blog.Login()
    blog.SayHello()
}
```

调用关系及运行效果如图 5-8 所示。

3. Go 函数-嵌套函数应用案例：递归函数

嵌套函数的定义，代码如下：

```
//anonymous - link\example\chapter5\nested.go
package main
```

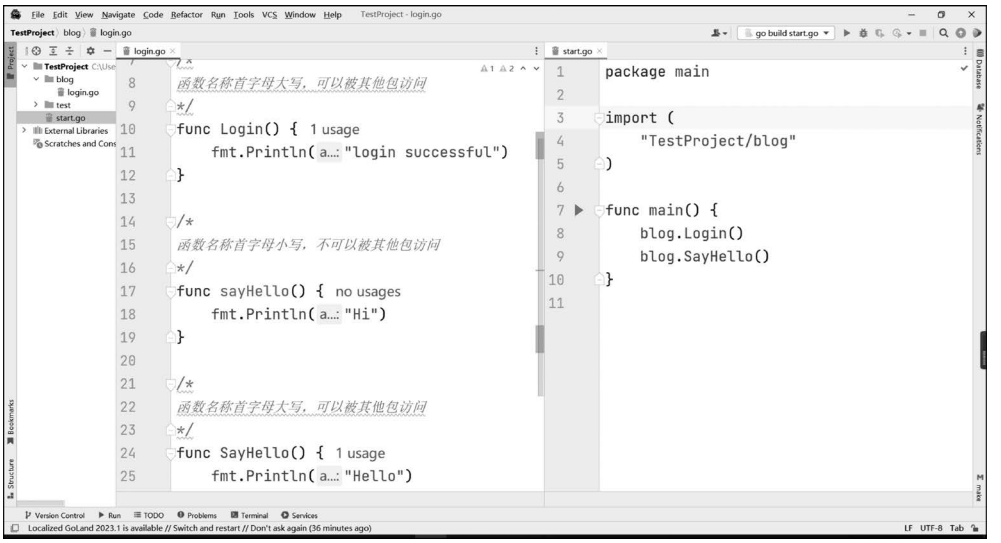


图 5-8　Go 多文件编程举例

```
import (
    "fmt"
)

func add1(x int, y int) int {
    fmt.Println("in add1...")
    return x + y
}
/*
    什么是嵌套函数：其实就是在一个函数中调用另外的函数
*/
func add2(x int, y int) int {
    fmt.Println("in add2...")
    return add1(x, y)
}

func main() {

    res := add2(100, 20)

    fmt.Println(res)
}
```

4. 嵌套函数的应用场景：递归函数

阶乘，代码如下：

```
//anonymous - link\example\chapter5\factorial.go
package main

import (
```

```
    "fmt"
)
/*
什么是递归函数：
如果一个函数在内部不调用其他函数，而是调用自己，则这个函数就是递归函数

递归函数的应用场景：
电商网站中的商品类别菜单的应用
查找某个目录下的文件

定义递归函数的注意事项：
(1)函数嵌套调用函数本身
(2)使用 return 指定函数出口
*/

var total = 1

func factorial(num int) {
    /*
递归函数需要定义递归函数的结束条件，否则会出现死递归的现象，如果出现死递归情况，程序就会
自动抛出"fatal error: stack overflow"异常
    */
    if num == 0 {
        return
    }
    total *= num

    /*
如果在函数内部自己调用自己，则这个函数就是递归函数
    */
    factorial(num - 1)
}

func main() {

    factorial(5)

    fmt.Printf("5 的阶乘是[%d]\n", total)
}
```

自我提升研究，上 100 层楼梯案例。

场景描述：一层楼有 100 个台阶，一个人上楼时他可以随机跨越 1～3 个台阶，那么问题来了，这个人从第 1 个台阶到第 100 个台阶总共有多少种走法？用递归方式实现。

还是基于上面的场景，假设这栋楼有 100 层，每层有 100 个台阶，这个人依旧只能随机跨越 1～3 个台阶，那么问题来了，这个人从第 1 层上到第 100 层楼共有多少种走法？用递归实现。

5.1.6　Go 函数：不定参数列表和多返回值函数

1. 不定参数列表

不定参数的产生背景是在定义函数时根据需求指定参数的个数和类型，但是有时如果无法确定参数的个数，此时就可以通过“不定参数列表”来解决这个问题，Go 语言的不定参数列表和 Python 中的 * args 有着异曲同工之妙。

Go 语言使用不定参数列表的语法格式如下：

```
func 函数名(数据集合 ...数据类型)
```

不定参数的案例，代码如下：

```
//anonymous - link\example\chapter5\indefinite.go
package main

import (
    "fmt"
)
/*
    不定参函数的定义：
    计算 N 个整型数据的和
*/
func sum(arr ...int) int {
    value := 0
    /*
        使用数组下标进行遍历
    */
    //for index := 0; index < len(arr); index++{
    //value += arr[index]
    //}

    /*
      使用 range 关键字进行范围遍历，range 会从集合中返回两个数：
          第 1 个是对应的坐标，赋值给了匿名变量"_"
          第 2 个对应的是值，赋值给了变量"data"
    */
    for _, data := range arr {
        value += data
    }
    return value
}

func main() {
    /*
        在调用函数时可以指定函数参数的个数不尽相同
    */
    fmt.Println(sum(1, 2, 3))
    fmt.Println(sum(1, 2, 3, 4, 5))
```

```
    fmt.Println(sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10))
}
```

2. 多返回值函数

函数返回多个值,代码如下:

```
//anonymous - link\example\chapter5\multiple.go
package main

import (
    "fmt"
)
/*
    函数的返回值是通过函数中的 return 语句获得的,return 后面的值也可以是一个表达式,只要返回值类型和定义的返回值列表所匹配即可
Go 语言支持多个返回值
*/
func test() (x int, y float64, z string) {

    return 18, 3.14, "Frank"
}

func main() {

    /*
        如果函数定义了多个返回值,就需要使用多个变量来接收这些返回值
        可以使用匿名变量("_")来接收不使用的变量的值,因此无法将匿名变量的值取出来
    */
    a, _, c := test()
    fmt.Println(a)
    fmt.Println(c)

}
```

5.1.7 Go 函数中的匿名函数应用案例:回调函数和闭包函数

1. 匿名函数

什么是匿名函数?顾名思义,就是没有函数名,而只有函数体的函数,函数可以作为一种类型被赋值给函数类型的变量,匿名函数往往以变量方式被传递。Go 语言支持匿名函数,即在需要使用函数时再定义函数。

Go 域名函数定义就是没有名字的普通函数,定义格式如下:

```
func(参数列表)(返回值列表){
    函数体
}
```

定义匿名函数时直接调用,示例代码如下:

```
//anonymous - link\example\chapter5\anonymous1.go
package main

import (
    "fmt"
)

func main() {

    /*
        定义匿名函数时直接调用
    */
    res := func(x int, y int) (z int) {
        z = x + y
        return z
    }(100, 20)

    fmt.Printf("res 的类型为[%T],res 的值为[%d]\n", res, res)
}
```

先声明匿名函数，再调用匿名函数，示例代码如下：

```
//anonymous - link\example\chapter5\anonymous2.go
package main

import (
    "fmt"
)

func main() {

    /*
        定义匿名函数，此时 add 是一个函数类型，只不过它是一个匿名函数
    */
    add := func(x int, y int) (z int) {
        z = x + y
        return z
    }
    fmt.Printf("add 的类型为[%T]\n", add)

    /*
        可以通过函数类型 add 多次调用匿名函数
    */
    res1 := add(100, 200)
    res2 := add(300, 500)
    fmt.Printf("res1 的类型为[%T],res1 的值为[%d]\n", res1, res1)
    fmt.Printf("res2 的类型为[%T],res2 的值为[%d]\n", res2, res2)
}
```

匿名函数可以作为返回值被多次调用，示例代码如下：

```
//anonymous - link\example\chapter5\anonymous3.go
package main

import (
    "fmt"
)
//使用 type 定义一个匿名函数类型
type FUNCTYPE func(int, int) int

func demo() FUNCTYPE {
    /*
        demo 的返回值为上面定义的匿名函数类型
    */
    return func(x int, y int) int {
        res := x + y
        return res
    }
}

func main() {
    /*

        add 的类型为(匿名)函数类型
    */
    add := demo()
    fmt.Printf("add 的类型为[%T],add 匿名函数的内存地址是[%X]\n", add, add)

    /*
        可以通过函数类型 add 多次调用匿名函数
    */
    res1 := add(100, 200)
    res2 := add(300, 500)
    fmt.Printf("res1 的类型为[%T],res1 的值为[%d]\n", res1, res1)
    fmt.Printf("res2 的类型为[%T],res2 的值为[%d]\n", res2, res2)
}
```

2. 匿名函数的应用场景

匿名函数经常被用于实现回调函数、闭包等。

(1) 回调函数的代码如下：

```
//anonymous - link\example\chapter5\anonymous4.go
package main

import (
    "fmt"
)
/*
    函数回调：
        简称回调,英文名为 Callback,即 call then back,被主函数调用运算后会返回主函数
        是指通过函数参数传递到其他代码的某一块可执行代码的引用
```

匿名函数作为回调函数的设计在 Go 语言的系统包中是很常见的，例如 strings 包中又有着实现，代码如下：

```
        func TrimFunc(s string, f func(rune) bool) string{
            return TrimRightFunc(TrimLeftFunc(s,f),f)
        }
*/
func callback(f func(int, int) int) int {
    return f(10, 20)
}

func add(x int, y int) int {
    return x + y
}

func main() {
    /*
        匿名函数(函数名本身是代码区的一个地址)的用途非常广泛，匿名函数本身是一种值，可以方便地保存在各种容器中实现回调函数和操作封装
    */
    fmt.Println(add)

    /*
        函数回调操作
    */
    fmt.Println(callback(add))
}
```

(2) 闭包函数的代码如下：

```
//anonymous - link\example\chapter5\anonymous5.go
package main

import (
    "fmt"
)
/*
    什么是闭包函数?
        闭包:闭是封闭(函数内部函数)，包是包含(该内部函数对外部作用域而非全局作用域的变量的引用)
        闭包指的是:函数内部函数对外部作用域而非全局作用域的引用

    Go 语言支持匿名函数作为闭包。匿名函数是一个内联语句或表达式
在下面的实例中，创建了函数 getSequence()，返回另外一个匿名函数 func() int。该函数的目的在闭包中递增 number 变量
*/
func getSequence() func() int {
    number := 100
    return func() int {
        /*
            匿名函数的优越性在于可以直接使用函数内的变量，而不必声明
```

```
            */
            number += 1
            return number
        }
    }

    func main() {
        /*
            f1 为一个空参匿名函数类型,number 变量的值依旧为 100
        */
        f1 := getSequence()

        /*
            调用 f1 函数,number 变量自增 1 并返回
        */
        fmt.Println(f1())
        fmt.Println(f1())
        fmt.Println(f1())

        fmt.Println("===== 分隔线 =====")
        /*
            创建新的匿名函数 f2,并查看结果
        */
        f2 := getSequence()
        fmt.Println(f2())
        fmt.Println(f2())
    }
```

5.1.8 Go 的面向对象编程

Go 的面向对象之所以与 C++、Java 及(较小程度上的)Python 这些语言不同,是因为它不支持继承,而仅支持聚合(也叫组合)和嵌入。接下来了解 Go 语言的面向对象编程。

1. 面向对象编程思想

面向对象编程刚流行时,继承是它首先被吹捧的最大优点之一,但是历经几十载的实践之后,事实证明该特性也有些明显的缺点,特别是当用于维护大系统时。Go 语言建议采用的是面向接口编程。

常见的编程方式:

(1) 面向过程(面向函数式编程):典型代表为 C 语言。

优点:流程清晰,代码易读。

缺点:耦合度太高,不利于项目迭代。

(2) 面向对象编程:典型代表为 C++、Java、Python、Go 等。

优点:解耦。

缺点:代码抽象度过高,不易读。

面向对象三要素:

（1）封装。

组装：将数据和操作组装到一起。

隐藏数据：对外只暴露一些接口，通过接口访问对象。例如驾驶员使用汽车，不需要了解汽车的构造细节，只需知道使用什么部件怎么驾驶就行，踩了油门就能跑，可以不了解其中的机动原理。

（2）继承。

多复用，继承来的就不用自己写了。

多继承少修改（Open-Closed Principle，OCP），使用继承来改变，来体现个性。

（3）多态。

面向对象编程最灵活的地方，即动态绑定。

与其他大部分使用聚合和继承的面向对象语言不同的是，Go 语言只支持聚合（也叫作组合）和嵌入。

（1）结构体的定义及初始化，示例代码如下：

```
//anonymous - link\example\chapter5\struct.go
package main

import (
    "fmt"
)

type Person struct {
    Name string
    Age int
    Gender string
}

type Student struct {
Person //通过匿名组合的方式嵌入了 Person 的属性
    Score float64
}

type Teacher struct {
    Person //通过匿名组合的方式嵌入了 Person 的属性
    Course string
}

type Schoolmaster struct {
    Person              //通过匿名组合的方式嵌入了 Person 的属性
    CarBrand string
}

func main() {
    /**
    第 1 种初始化方式：先定义后赋值
     */
```

```
    s1 := Student{}
    s1.Name = "Jason Yin"
    fmt.Println(s1)
    fmt.Printf("%+v\n\n", s1) //+v 表示打印结构体的各个字段

    /**
    第 2 种初始化方式:直接初始化
    */
    s2 := Teacher{Person{"张三", 18, "boy"}, "Go 并发编程"}
    fmt.Println(s2)
    fmt.Printf("%+v\n\n", s2)

    /**
    第 3 种赋值方式:初始化赋值部分字段
    */
    s3 := Schoolmaster{CarBrand: "丰田", Person: Person{Name: "JasonYin 最强王者"}}
    fmt.Println(s3)
    fmt.Printf("%+v\n", s3)
}
```

(2) 结构体的属性继承及变量赋值,示例代码如下:

```
//anonymous - link\example\chapter5\inherit.go
package main

import (
    "fmt"
)

type Animal struct {
    Age int
}

type People struct {
    Animal
    Name   string
    Age    int
    Gender string
}

type IdentityCard struct {
    IdCardNO     int
    Nationality  string
    Address      string
    Age          int
}
/*
    此时的 Students 采用了多重继承 */
type Students struct {
    IdentityCard
```

```
    People //多层继承
    Age int
    Score int
}

func main() {
    /**
    如果子类和父类存在同名的属性,则以就近原则为准
    */
    s1 := Students{
        Score: 150,
        IdentityCard: IdentityCard{
            IdCardNO: 110105199003072872,
            Nationality: "中华人民共和国",
            Address: "北京市朝阳区望京 SOHO",
            Age: 8,
        },
        People: People{Name: "Jason Yin", Age: 18, Animal: Animal{Age: 20}},
        Age: 27,
    }

    /**
    如果子类和父类存在同名的属性(如果父类还继承了其他类型,则称为多层继承),就以就近原则为准
    但是如果一个子类继承自多个父类(称为多重继承)且每个字段中都有相同的字段,则此时无法直接在子类调用该属性
    */
    fmt.Printf("学生的年龄是:[%d]\n", s1.Age)
    s1.Age = 21
    fmt.Printf("学生的年龄是:[%d]\n\n", s1.Age)

    //给 People 类的 Age 赋值
    fmt.Printf("People 的年龄是:[%d]\n", s1.People.Age)
    s1.People.Age = 5000
    fmt.Printf("People 的年龄是:[%d]\n\n", s1.People.Age)

    //给 IdentityCard 类的 Age 赋值
    fmt.Printf("IdentityCard 的年龄是:[%d]\n", s1.IdentityCard.Age)
    s1.IdentityCard.Age = 80
    fmt.Printf("IdentityCard 的年龄是:[%d]\n", s1.IdentityCard.Age)
}
```

(3) 匿名组合对象指针,示例代码如下:

```
//anonymous - link\example\chapter5\combination.go
package main

import (
    "fmt"
    "time"
```

```
)
type Vehicle struct {
    Brand string
    Wheel Byte
}

type Car struct {
    Vehicle
    Colour string
}

type Driver struct {
    *Car
    DrivingTime time.Time
}

func main() {
    /**
    对象指针匿名组合的第1种初始化方式:
        定义时直接初始化赋值
    */
    d1 := Driver{&Car{
        Vehicle: Vehicle{
            Brand: "丰田",
            Wheel: 4,
        },
        Colour: "红色",
    }, time.Now(),
    }
    //打印结构体的详细信息,注意观察指针对象
    fmt.Printf("%+v\n", d1)
    //可以直接调用对象的属性
    fmt.Printf("品牌:%s,颜色:%s\n", d1.Brand, d1.Colour)
    fmt.Printf("驾驶时间:%+v\n\n", d1.DrivingTime)
    time.Sleep(1000000000 * 3)

    /**
    对象指针匿名组合的第2种初始化方式:
        先声明,后赋值。遇到指针的情况一定要避免空(nil)指针,未初始化的指针的默认值为
nil,可以考虑使用new函数解决
    */
    var d2 Driver
    /**
    由于Driver结构体中有一个对象指针匿名组合Car,因此需要使用new函数申请空间
    */
    d2.Car = new(Car)
    d2.Brand = "奔驰"
    d2.Colour = "黄色"
    d2.DrivingTime = time.Now()
    fmt.Printf("%+v\n", d2)
    fmt.Printf("品牌:%s,颜色:%s\n", d2.Brand, d2.Colour)
    fmt.Printf("驾驶时间:%+v\n", d1.DrivingTime)
}
```

（4）结构体成员方法，示例代码如下：

```
//anonymous - link\example\chapter5\member.go
package main

import (
    "fmt"
)
//定义一个结构体
type Lecturer struct {
    Name string
    Age uint8
}
//为 Lecturer 结构体封装 Init 成员方法
func (l *Lecturer) Init() {
    l.Name = "Jason Yin"
    l.Age = 20
}
/**
为 Lecturer 结构体起一个别名
可以为 Instructor 类型添加成员方法
通过别名和成员方法为原有类型赋值新的操作
*/

type Instructor Lecturer

/**
    (1)为一个结构体创建成员方法时，如果成员方法有接收者，则需要考虑以下两种情况：
        如果这个接收者是对象，则是值传递；
        如果这个接收者是对象指针，则是引用传递。
    (2)只要函数接收者不同，哪怕函数名称相同，也不算同一个函数。
    (3)不管接收者变量名称是否相同，只要类型一致(包括对象和对象指针)，那么就认为接收者是
相同的，这时不允许出现相同名称函数。
    (4)给指针添加方法时，不允许给指针类型添加操作(因为 Go 语言中指针类型是只读的)
*/
func (i *Instructor) Init() {
    i.Name = "张三"
    i.Age = 18
}

func main() {

    var (
        l Lecturer
        i Instructor
    )

//可以使用对象调用成员方法
    i.Init()
    fmt.Printf("%+v\n\n", i)
```

```
    //可以用对象指针调用成员方法
    (&l).Init()
    fmt.Printf("%+v\n", l)
}
```

(5) 结构体的方法继承和重写,示例代码如下:

```
//anonymous-link\example\chapter5\overwrite.go
package main

import (
    "fmt"
)

type Father struct {
    Name string
    Age int
}

func (f *Father) Init() {
    f.Name = "成龙"
    f.Age = 66
}
//定义父类的 Eat 成员方法
func (f *Father) Eat() {
    fmt.Println("Jackie Chan is eating...")
}
//重写父类的 Eat 成员方法
func (s *Son) Eat() {
    fmt.Println("FangZuming is eating...")
}
//让 Son 类继承 Father 父类
type Son struct {
    Father //匿名组合能够继承父类的属性和方法
    Score int
}

func main() {
    var s Son
    s.Init()
    fmt.Printf("%+v\n", s)
    s.Eat()
    s.Name = "房祖名"
    s.Age = 38
    s.Score = 100
    fmt.Printf("%+v\n", s)
}
```

(6) 方法值和方法表达式,示例代码如下:

```
//anonymous-link\example\chapter5\expression.go
package main
```

```
import (
    "fmt"
)
/**
   定义函数,函数的返回值是函数类型
 */
func CallBack(a int) func(b int) int {
    return func(c int) int {
        fmt.Println("调用了 CallBack 回调函数...")
        return a + c
    }
}

type BigData struct {
    Name string
}

func (this *BigData) Init() {
    this.Name = "Hadoop"
}

func (this *BigData) PrinfInfo() {
    fmt.Printf("%v 是大数据生态圈的基石。\n", this.Name)
}

func (this BigData) SetInfoValue() {
    fmt.Printf("SetInfoValue : %p, %v\n", &this, this)
}

func (this *BigData) SetInfoPointer() {
    fmt.Printf("SetInfoPointer : %p, %v\n", this, this)
}

func main() {

    /**
      调用回调函数的返回值为函数类型
     */
    result := CallBack(10)
    fmt.Printf("result 的类型是:[%T],result 的值是:[%v]\n", result, result)
    res1 := result(20)          //对返回的函数再次进行调用
    fmt.Printf("res1 的类型是:[%T],res1 的值是:[%d]\n\n", res1, res1)

    var hadoop BigData
    hadoop.Init()               //调用 Hadoop 的初始化方法
    info := hadoop.PrinfInfo    //可以声明一个函数变量 info,称为方法表达式

    /**
      对 info 函数变量进行调用,这样可以起到隐藏调用者 hadoop 对象的效果(类似于回调函数的
      调用效果)
      方法值可以隐藏调用者,称为隐式调用
```

```
    */
    info()

    /**
       方法表达式可以显式调用,必须传递方法调用者对象,在实际开发中很少使用这种方式,了
       解即可
    */

    elk := BigData{"Elastic Stack"}
    fmt.Printf("main: %p, %v\n\n", &elk, elk)
    s1 := (*BigData).SetInfoPointer
    s1(&elk) //显式地把接收者传递过去
    s2 := (BigData).SetInfoValue
    s2(elk) //显式地把接收者传递过去
}
```

(7) 面向接口编程,举一个多态案例。

接口概述:Go 语言的接口类型用于定义一组行为,其中每个行为都由一种方法声明表示。接口类型中的方法声明只有方法签名而没有方法体,而方法签名包括且仅包括方法的名称、参数列表和结果返回列表。在 Go 语言中,接口是一个自定义类型,它声明了一种或者多种方法签名。接口是完全抽象的,因此不能将其实例化,然而,可以创建一个类型为接口的变量,它可以被赋值为任何满足该接口类型的实际类型的值。

计算器案例(多态案例),示例代码如下:

```
//anonymous - link\example\chapter5\polymorphic.go
package main

import (
    "fmt"
)
//实现面向对象版本包含加减法的计算器
type Parents struct {
    x int
    y int
}
//实现加法类
type Addition struct {
    Parents
}
//实现减法类
type Subtraction struct {
    Parents
}
//实现乘法类
type multiplication struct {
    Parents
}
```

```
//实现除法类
type Division struct {
    Parents
}

func (this *Addition) Operation() int {
    return this.x + this.y
}

func (this *Subtraction) Operation() int {
    return this.x - this.y
}

func (this *multiplication) Operation() int {
    return this.x * this.y
}

func (this *Division) Operation() int {
    return this.x / this.y
}
/**
实现接口版本包含加减法的计算器
接口就是一种规范标准,接口中不实现函数,只定义函数格式
面向接口编程(也称为面向协议编程)降低了代码的耦合度,方便后期代码的维护和扩充,这种实现
方法称为多态

多态三要素:
    (1)父类是接口。
    (2)子类实现所有接口中定义的函数。
    (3)有一个父类接口对应子类对象指针
*/
type MyCalculator interface {
Operation() int          //实现接口的结构体中必须包含 Operation 函数名且返回值为 int 类型
}

func Calculation(c MyCalculator) int {
    return c.Operation()
}

func main() {
    //调用加法
    a := Addition{Parents{100, 20}}
    sum := a.Operation()
    fmt.Println(sum)

    //调用减法
    b := Subtraction{Parents{100, 20}}
    sub := b.Operation()
    fmt.Println(sub)

    //调用乘法
```

```
    c := multiplication{Parents{100, 20}}
    mul := c.Operation()
    fmt.Println(mul)

    //调用除法
    d := Division{Parents{100, 20}}
    div := d.Operation()
    fmt.Println(div)

    fmt.Println("===== 分隔线 =====")

    //调用接口,需要传入对象指针,与上面面向对象的方法相比,接口表现了面向接口三要素中的
    //多态特征
    fmt.Println(Calculation(&a))
    fmt.Println(Calculation(&b))
    fmt.Println(Calculation(&c))
    fmt.Println(Calculation(&d))
}
```

空接口和类型断言,示例代码如下:

```
//anonymous-link\example\chapter5\assertion.go
package main

import (
    "fmt"
    "reflect"
)
/**
空接口(interface{})不包含任何方法,正因为如此,所有的类型都实现了空接口,因此空接口可以存
储任意类型的数值
如下所示,为空接口起了一个别名
*/
type MyInterface interface{}

func MyPrint(input MyInterface) {
    /**
    使用断言语法获取传输过来的数据类型,类似于类型强转
    断言语法格式如下:
        接口类型变量(断言的类型)
    如果不确定 interface 具体是什么类型,则在断言之前最好先进行判断
    */
    output, ok := input.(int)
    if ok {
        output = input.(int) + 100 //通过断言语法可以判断数据类型
        fmt.Println(output)
    } else {
        fmt.Println(input)
    }

    inputType := reflect.TypeOf(input) //通过反射也可以判断类型
```

```
    fmt.Printf("用户传入的是:[%v],其对应的类型是[%v]\n\n", input, inputType)
}

func main() {
    m1 := true
    MyPrint(m1)

    m2 := "Jason Yin"
    MyPrint(m2)

    m3 := 2020
    MyPrint(m3)

}
```

5.1.9　Go的高级数据类型实例：字典

Go中的字典(map)和数组与切片一样，都用来保存一组相同的数据类型。可以通过key键获取value值，map为映射关系容器，采用散列(hash)实现。

如果数据存在频繁删除操作，则应尽量不要使用切片，map删除数据效率要比切片高，如果数据需要排序，则切片和数组比map好，因为map是无序的。

(1) 字典的定义，示例代码如下：

```
//anonymous - link\example\chapter5\map_define.go
package main

import (
    "fmt"
)

func main() {
    /*
        声明字典结构语法如下：
            var 字典 map[键类型]值类型
        定义字典结构使用map关键字，"[]"中指定的是键(key)的类型，后面紧跟着的是值(value)的类型
        map中的key值除了切片、函数、复数(complex)及包含切片的结构体都可以，换句话说，使用这些类型会造成编译错误
     map在使用前也需要使用make函数进行初始化
     map没有容量属性，map只有长度属性，长度表示的是map中key和value有多少对
     map满足集合的特性，即key是不能重复的
    */

    //声明一个字典类型
    var m1 map[string]string
    //map在使用前必须初始化空间，和切片类似的是map自身也没有空间
    m1 = make(map[string]string)
    //注意，key和value都是字符串类型
```

```
    m1["Name"] = "Jason Yin"
    //注意,上一行已经定义"Name"这个 key 名称了,再次使用同名 key 会将上一个 key 对应的
    //value 覆盖
    m1["Name"] = "张三"
    fmt.Printf("m1 的数据类型是:%T,对应的长度是:%d\n", m1, len(m1))
    fmt.Println("m1 的数据是:", m1)

    //使用自动推导的类型并初始化空间
    m2 := make(map[string]int)
    //注意 key 是字符串类型,而 value 是 int 类型
    m2["Age"] = 18
    fmt.Printf("m2 的数据类型是:%T,对应的长度是:%d\n", m2, len(m2))
    fmt.Println("m2 的数据是:", m2)

    //直接初始化空间并赋初始值
    m3 := map[string]rune{"first": '周', "second": '杰', "third": '伦'}
    fmt.Printf("m3 的数据类型是:%T,对应的长度是:%d\n", m3, len(m3))
    fmt.Println("m3 的数据是:", m3)
}
```

(2) 字典的基本操作,主要包含以下几种操作。

字典的访问方式(查询),示例代码如下:

```
//anonymous - link\example\chapter5\map_find.go
package main

import (
    "fmt"
)

func main() {
    m1 := map[string]rune{"first": '周', "second": '杰', "third": '伦'}

    //第 1 种访问方式,可以通过 key 值访问
    fmt.Println("===== 第 1 种访问方式 =====")
    fmt.Println(m1["first"])

    //第 2 种访问方式,可以通过变量名访问所有数据
    fmt.Println("===== 第 2 种访问方式 =====")
    fmt.Println(m1)

    //第 3 种访问方式,同时获得 key 和 value
    fmt.Println("===== 第 3 种访问方式 =====")
    for key, value := range m1 {
        fmt.Println("key 值是:", key, ",value 值是:", value)
    }

    //第 4 种访问方式,只获得 key,基于 key 范围获取对应的 value
    fmt.Println("===== 第 4 种访问方式 =====")
    for key := range m1 {
```

```
        fmt.Println("key 值是:", key, ",value 值是:", m1[key])
    }

    //第 5 种访问方式,判断一个 map 是否有 key,基于返回的 bool 值执行相应的操作
    fmt.Println(" ===== 第 5 种访问方式 ===== ")
    value, flag := m1["first"]
    if flag {
        fmt.Println("key 的值为:", value)
    }
}
```

字典的增、删、改操作,示例代码如下:

```
//anonymous - link\example\chapter5\map_operator.go
package main

import (
    "fmt"
)

func main() {
    m1 := map[string]rune{"first": '周', "second": '杰', "third": '伦'}

    //增加 map 键值
    fmt.Println("增加 key 之前:", m1)
    m1["test"] = 666666
    fmt.Println("增加 key 之后:", m1)

    //更新键值
    m1["test"] = 88888888
    fmt.Println("更新 key 之后:", m1)

    //删除键值,Go 语言中 delete 函数只有删除 map 中元素的作用
    delete(m1, "test")
    fmt.Println("删除 key 之后:", m1)

}
```

字典的嵌套,示例代码如下:

```
//anonymous - link\example\chapter5\map_nest.go
package main

import (
    "fmt"
)

func main() {

    m1 := map[string]rune{"first": '广', "second": '东', "third": '省'}
```

```
    /*
        定义一个嵌套数据类型
    */
    m2 := make(map[string]map[string]int32)

    //可以为嵌套类型赋值
    m2["name"] = m1
    fmt.Println("m1 的数据为:", m1)
    fmt.Println("m2 的数据为:", m2)

}
```

(3) 字典作为函数参数,示例代码如下:

```
//anonymous - link\example\chapter5\map_args.go
package main

import (
    "fmt"
)

func Rename(m map[string]string) {
    //为传递进来的 map 增加一个 key
    m["name"] = "Jason Yin"
    fmt.Printf("Rename 函数中的 m 地址为:%p\n", m)
}

func main() {
    /*
        在 Go 语言中,数组作为参数进行传递是值传递,而切片作为参数进行传递是引用传递
        值传递:
            方法调用时,实参数把它的值传递给对应的形式参数,方法执行中形式参数值的改变不会影响实际参数的值
            引用传递(也称为传地址):
                函数调用时,实际参数的引用(地址,而不是参数的值)被传递给函数中相对应的形式参数(实际参数与形式参数指向了同一块存储区域)
                在函数执行时,对形式参数的操作实际上就是对实际参数的操作,方法执行中形式参数值的改变会影响实际参数的值
        map 作为函数参数传递实际上和切片传递一样,传递的是地址,也就是常说的引用传递
            (1)要先使用 make 进行初始化操作再使用类型,在函数传递时基本上是引用传递
            (2)在日常开发中,常见引用传递的高级数据类型有切片、字典和管道
    */
    m1 := make(map[string]string)

    fmt.Println("调用前的 m1 数据为:", m1)
    fmt.Printf("main 函数中的 m1 地址为:%p\n", m1)

    Rename(m1)

    fmt.Println("调用后的 m1 数据为:", m1)
}
```

5.1.10　Go的文本文件处理：文件操作常见的API

(1) 打开文件相关操作，主要包含如下方法。

如果文件不存在，就创建文件，如果文件存在，就清空文件内容并打开(Create方法)，示例代码如下：

```
//anonymous - link\example\chapter5\file_open1.go
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {

    /**
    创建文件的API函数，签名如下：
        func Create(name string) (*File, error)
    下面是对API参数的解释说明：
        name指的是文件名称，可以是相对路径，也可以是绝对路径
        *File指的是文件指针，使用完文件后要记得释放该文件指针资源
        error指的是创建文件的报错信息，例如，如果指定的文件父目录不存在就会报错"The system cannot find the path specified."
    注意事项：
        根据提供的文件名创建新的文件，返回一个文件对象，返回的文件对象是可读写的
        创建文件时，如果存在重名的文件就会覆盖原来的文件。换句话说，如果文件存在，就清空文件内容并打开新文件，如果文件不存在，则创建新文件并打开
    */
    f, err := os.Create("E:\\frank\\input\\kafka.txt")

    /**
    一旦文件报错就执行return语句，下面的defer语句就不会被执行
    注意事项：
        不要将下面的defer语句和判断错误的语句互换位置，因为判断错误的语句是确保文件是否创建成功的
        如果有错误，则意味着文件没有被成功创建，换句话说，如果文件创建失败，则文件指针为空，此时如果执行关闭文件操作，则会报错
        如果没有错误就意味着文件创建成功，即在执行关闭文件操作时可确保不会报错
    */
    if err != nil {
        fmt.Println(errors.New("报错提示：" + err.Error()))
        return
    } else {
        fmt.Println("文件创建成功...")
    }

    /**
```

```
    文件成功创建之后一直处于打开状态,因此使用完之后一定要关闭文件,当然关闭文件之前一
定要确保文件已经创建成功
    */
    defer f.Close()
}
```

以只读方式打开文件(Open 方法),示例代码如下:

```
//anonymous - link\example\chapter5\file_open2.go
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {

    /**
    打开文件的 API 函数,很明显它是基于 OpenFile 实现的
        func Open(name string) (*File, error) {
            return OpenFile(name, O_RDONLY, 0)
        }
    下面是对 API 参数的解释说明:
        name 指的是文件名称,可以是相对路径,也可以是绝对路径
        *File 指的是文件指针,使用完文件后要记得释放该文件指针资源
        error 指的是创建文件的报错信息,例如,如果指定的文件不存在就会报错"The system
cannot find the file specified."
    注意事项:
        Open()是以只读权限打开文件名为 name 的文件,得到的文件指针 file 只能用来对文件进
行读操作。
        如果有写文件的需求,就需要借助 Openfile 函数来打开了
*/

    f, err := os.Open("E:\\frank\\input\\kafka.txt")

    /**
    一旦文件报错就执行 return 语句,下面的 defer 语句就不会被执行
    不要将下面的 defer 语句和判断错误的语句互换位置,因为判断错误的语句用来判断文件是否
创建成功
    如果有错误,则意味着文件没有被成功创建,换句话说,如果文件创建失败,则文件指针为空,此
时如果执行关闭文件,则会报错
    如果没有错误就意味着文件创建成功,即在执行关闭文件的操作时不会报错
    */
    if err != nil {
        fmt.Println(errors.New("报错提示:" + err.Error()))
        return
    } else {
        fmt.Println("文件打开成功...")
    }
```

```
    /**
    文件成功创建之后一直处于打开状态,因此使用完之后一定要关闭文件,当然关闭文件之前一
定要确保文件已经创建成功
    */
    defer f.Close()
}
```

自定义文件的打开方式(OpenFile 方法),示例代码如下:

```
//anonymous - link\example\chapter5\file_open3.go
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {
    /**
      创建文件的 API 函数,签名如下:
          func OpenFile(name string, flag int, perm FileMode) (*File, error)
      下面是对 API 参数的解释说明:
          name 指的是文件名称,可以是相对路径,也可以是绝对路径
          flag 表示读写模式,常见的模式有:O_RDONLY(只读模式)、O_WRONLY(只写模式)和 O_RDWR
(可读可写模式)
          perm 表示打开权限。来源于 Linux 系统调用中的 open 函数,当参数为 O_CREATE 时,可创
建新文件
              权限取值是八进制,即 0~7
                  0:没有任何权限
                  1:执行权限(如果是可执行文件,则可以运行)
                  2:写权限
                  3:写权限与执行权限
                  4:读权限
                  5:读权限与执行权限
                  6:读权限与写权限
                  7:读权限、写权限与执行权限
          *File 指的是文件指针,使用完文件后要记得释放该文件指针资源
          error 指的是创建文件的报错信息,例如,如果指定的文件父目录不存在就会报错"The
system cannot find the path specified."

       使用 OpenFile 打开的文件,默认写时是从文件开头开始写入数据,这样会将原来的数据
覆盖
       如果不想以覆盖的方式写入,而想使用追加的方式写入,则具体的代码如下(当然也可以使
用 Seek 函数实现)
    */
    f, err := os.OpenFile("E:\\frank\\input\\kafka.txt", os.O_RDWR|os.O_APPEND, 0666)

    /**
      一旦文件报错就执行 return 语句,下面的 defer 语句就不会被执行
```

```
        注意事项：
            不要将下面的 defer 语句和判断错误的语句互换位置，因为判断错误的语句用来判断文件是否创建成功
            如果有错误，则意味着文件没有被成功创建，换句话说，如果文件创建失败，则文件指针为空，此时如果执行关闭文件操作，则会报错
            如果没有错误就意味着文件创建成功，即在执行关闭文件操作时不会报错
    */
    if err != nil {
        fmt.Println(errors.New("报错提示：" + err.Error()))
        return
    } else {
        fmt.Println("文件创建成功...")
    }

    /**
      文件成功创建之后一直处于打开状态，因此使用完之后一定要关闭文件，当然关闭文件之前一定要确保文件已经创建成功
    */
    defer f.Close()
}
```

(2) 写文件相关操作，主要包含以下几种方法。

Write 方法，示例代码如下：

```
//anonymous - link\example\chapter5\file_write1.go
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {

    f, err := os.Create("E:\\frank\\input\\kafka.txt")
    if err != nil {
        fmt.Println(errors.New("报错提示：" + err.Error()))
        return
    } else {
        fmt.Println("文件创建成功...")
    }

    /**
    文件成功创建之后一直处于打开状态，因此使用完之后一定要关闭文件，当然关闭文件之前一定要确保文件已经创建成功
    */
    defer f.Close()

    /**
```

```
    写入字节切片到文件中,Write 的函数签名如下:
        func (f *File) Write(b []Byte) (n int, err error)
    如上所示,传入需要写入的字节切片即可将内容写到操作系统对应的文件中
    */
    f.Write([]Byte("Kafka 是一个吞吐量高的消息队列\n"))

}
```

WriteString 方法,示例代码如下:

```
//anonymous - link\example\chapter5\file_write2.go
package main

import (
    "errors"
    "fmt"
    "os"
)

func main() {

    f, err := os.Create("E:\\frank\\input\\kafka.txt")
    if err != nil {
        fmt.Println(errors.New("报错提示:" + err.Error()))
        return
    } else {
        fmt.Println("文件创建成功...")
    }

    /**
    文件成功创建之后一直处于打开状态,因此使用完之后一定要关闭文件,当然关闭文件之前一
定要确保文件已经创建成功
    */
    defer f.Close()

    /**
    将字符串写入文件,其函数实现如下,很明显,WriteString 底层调用的依旧是 Write 方法
        func (f *File) WriteString(s string) (n int, err error) {
            return f.Write([]Byte(s))
        }
    如上所示,传入需要写入的字符串即可将内容写到操作系统对应的文件中
    */
    f.WriteString("Kafka 是一个消息队列")

}
```

WriteAt 方法,示例代码如下:

```
//anonymous - link\example\chapter5\file_write3.go
package main
```

```
import (
    "errors"
    "fmt"
    "os"
)

func main() {

    f, err := os.Create("E:\\frank\\input\\kafka.txt")
    if err != nil {
        fmt.Println(errors.New("报错提示:" + err.Error()))
        return
    } else {
        fmt.Println("文件创建成功...")
    }

    /**
    文件成功创建之后一直处于打开状态,因此使用完之后一定要关闭文件,当然关闭文件之前一
定要确保文件已经创建成功
    */
    defer f.Close()

    /**
    WriteAt 以带偏移量的方式写入数据,偏移量从文件起始位置开始,WriteAt 的函数签名如下:
        func (f *File) WriteAt(b []Byte, off int64) (n int, err error)
    以下是对函数签名相关参数的说明:
        b 表示待写入的数据内容
        off 表示偏移量(通常是 Seek 函数的返回值)
    */
    f.WriteAt([]Byte("Kafka"), 10)

}
```

Seek 方法,示例代码如下:

```
//anonymous - link\example\chapter5\file_write4.go
package main

import (
    "fmt"
    "os"
    "syscall"
)
const (
    //Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
    O_RDONLY int = syscall.O_RDONLY         //open the file read - only.
    O_WRONLY int = syscall.O_WRONLY         //open the file write - only.
    O_RDWR int = syscall.O_RDWR             //open the file read - write.
    //The remaining values may be or'ed in to control behavior.
    O_APPEND int = syscall.O_APPEND         //append data to the file when writing.
```

```
    O_CREATE int = syscall.O_CREAT          //create a new file if none exists.
    O_EXCEL int = syscall.O_EXCEL           //used with O_CREATE, file must not exist.
    O_SYNC int = syscall.O_SYNC             //open for synchronous I/O.
    O_TRUNC int = syscall.O_TRUNC           //truncate regular writable file when opened.
    )

func main() {
    f, err := os.OpenFile("E:\\frank\\input\\flume.txt", O_RDWR|O_CREATE|O_TRUNC, 0666)
    if err != nil {
        fmt.Println("err = ", err)
        return
    }
    defer f.Close()

    f.Write([]Byte("Flume 是一个文本日志收集工具。\n"))
    f.WriteString("Flume 是一种分布式、可靠且可用的服务，用于有效地收集、聚合和移动大量日
志数据。")
    f.WriteAt([]Byte("Flume 具有基于流数据流的简单灵活的体系结构。"), 60) //可以指定偏移
                                                                    //量写入

    /**
    Seek 的函数签名如下：
        func (f *File) Seek(offset int64, whence int) (ret int64, err error)
    以下是对函数签名相关参数的说明：
        offset 指定偏移量，如果是正数，则表示向文件尾偏移；如果是负数，则表示向文件头偏移
        whence 指定偏移量的起始位置
            io.SeekStart：文件起始位置，对应常量整型 0
            io.SeekCurrent：文件当前位置，对应常量整型 1
            io.SeekEnd：文件结尾位置，对应常量整型 2
        返回值：
            表示从文件起始位置到当前文件读写指针位置的偏移量
    */
    f.Seek(10, 2)

    f.WriteString("Flume 具有可调整的可靠性机制及许多故障转移和恢复机制，具有强大的功能
和容错能力。")
}
```

（3）读文件相关操作，主要包含以下几种方法。

Read 方法，示例代码如下：

```
//anonymous - link\example\chapter5\file_read1.go
package main

import (
    "fmt"
    "os"
)

func main() {
```

```
    f, err := os.OpenFile("E:\\frank\\input\\flume.txt", os.O_RDWR|os.O_APPEND, 0666)
    if err != nil {
        fmt.Println("文件打开失败:", err)
        return
    }
    defer f.Close()
    /**
    切片在使用前要申请内存空间,如果不知道文件的大小,则可尽量多给点空间,最好是4K的倍数
    如果在读取大文件的情况下,则应该循环读取
    */
    temp := make([]Byte, 1024*4)

    f.Read(temp)

    fmt.Println(string(temp))
}
```

ReadAt 方法,示例代码如下:

```
//anonymous-link\example\chapter5\file_read2.go
package main

import (
    "fmt"
    "os"
)

func main() {
    f, err := os.OpenFile("E:\\frank\\input\\flume.txt", os.O_RDWR|os.O_APPEND, 0666)
    if err != nil {
        fmt.Println("文件打开失败:", err)
        return
    }
    defer f.Close()

    temp := make([]Byte, 1024*4)

    f.ReadAt(temp, 5) //以带偏移量的方式获取数据,从文件头开始,当然使用Seek函数也能实现
                      //该功能
    fmt.Println(string(temp))
}
```

按行读取,示例代码如下:

```
//anonymous-link\example\chapter5\file_read3.go
package main

import (
    "bufio"
    "Bytes"
```

```
    "fmt"
    "os"
)

func main() {
    f, err := os.OpenFile("E:\\frank\\input\\flume.txt", os.O_RDWR|os.O_APPEND, 0666)
    if err != nil {
        fmt.Println("文件打开失败：", err)
        return
    }
    defer f.Close()

    buf := make([]Byte, 1024 * 4)
    f.Read(buf)   //一定要将数据读到切片中,否则在执行下面的操作时会读取不到数据
    reader := bufio.NewReader(Bytes.NewReader(buf))   //初始化一个阅读器
    line, _ := reader.ReadString('\n')
    //为阅读器将分隔符指定为换行符("\n"),即每次只读取一行
    fmt.Println(line)
}
```

（4）删除文件,示例代码如下：

```
//anonymous - link\example\chapter5\file_delete.go
package main

import (
    "os"
)

func main() {

    os.Remove("E:\\frank\\input\\kafka.txt")     //删除文件
}
```

（5）大文件复制,示例代码如下：

```
//anonymous - link\example\chapter5\file_copy.go
package main

import (
    "fmt"
    "io"
    "os"
)

func main() {
    args := os.Args                          //获取命令行参数，并判断输入是否合法

    if args == nil || len(args) != 3 {
```

```
        fmt.Println("useage : xxx srcFile dstFile")
        return
    }

    srcPath := args[1]                                  //获取源文件路径
    dstPath := args[2]                                  //获取目标文件路径
    fmt.Printf("srcPath = %s, dstPath = %s\n", srcPath, dstPath)

    if srcPath == dstPath {
        fmt.Println("error:源文件名与目的文件名相同")
        return
    }

    srcFile, err1 := os.Open(srcPath)                   //打开源文件
    if err1 != nil {
        fmt.Println(err1)
        return
    }

    dstFile, err2 := os.Create(dstPath)                 //创建目标文件
    if err2 != nil {
        fmt.Println(err2)
        return
    }

    buf := make([]Byte, 1024)                           //切片缓冲区
    for {
        //从源文件读取内容,n 为读取文件内容的长度
        n, err := srcFile.Read(buf)
        if err != nil && err != io.EOF {
            fmt.Println(err)
            break
        }

        if n == 0 {
            fmt.Println("文件处理完毕")
            break
        }

        //切片截取
        tmp := buf[:n]

        //把读取的内容写入目的文件
        dstFile.Write(tmp)
    }

    //关闭文件
    srcFile.Close()
    dstFile.Close()
}
```

5.1.11　Go 的文本文件处理：目录操作常见的 API

（1）读取目录内容，示例代码如下：

```
//anonymous - link\example\chapter5\dir_read.go
package main

import (
    "fmt"
    "os"
)

func main() {

    /**
      如下所示，打开目录和打开文件的函数是同一个函数：
          func OpenFile(name string, flag int, perm FileMode) (*File, error)

      函数签名各个参数的解释如下：
          name 表示要打开的目录名称。使用绝对路径较多
          flag 表示打开目录的读写模式，通常传 O_RDONLY(只读模式)
          perm 表示打开权限，但对于目录来讲略有不同，通常传 os.ModeDir
          返回值：
              由于是操作目录，所以 file 是指向目录的文件指针(*File)
              在 error 中保存错误信息
    */
    f, err := os.OpenFile("E:\\frank\\input", os.O_RDONLY, os.ModeDir)

    if err != nil {
        fmt.Println("目录打开失败：", err)
        return
    }
    defer f.Close()

    /**
        Readdir 函数的函数签名如下：
            func (f *File) Readdir(n int) ([]FileInfo, error)
        函数签名各个参数的解释如下。
            n：
              Readdir 读取与文件关联的目录的内容，并按目录顺序返回由 Lstat 返回的最多 n 个 FileInfo 值组成的片段。对同一文件的后续调用将产生更多的文件信息
              如果 n > 0，Readdir 最多返回 n 个 FileInfo 结构。在这种情况下，如果 Readdir 返回一个空片段，它将返回一个非 nil 错误来解释原因。在目录的末尾，错误是 io.EOF
              如果 n < = 0，Readdir 将在一个切片中返回目录中的所有文件信息。在这种情况下，如果 Readdir 成功(一直读取到目录的末尾)，则返回切片和 nil 错误。如果在目录结束之前遇到错误，Readdir 将返回在该点之前读取的 FileInfo 和非 nil 错误
          返回值：
              返回两个值，一个是读取的文件信息切片对象("[]FileInfo")，另一个是错误信息("error")
              FileInfo 中可以获取文件的名称、大小、权限、修改时间、是否是目录等。
    */
```

```
    files, err := f.Readdir(-100)
    if err != nil {
        fmt.Println("错误信息：", err)
        return
    }

    for _, file := range files {
        fmt.Printf("文件名称是：%s,文件大小是：%d,是否是目录：%t\n", file.Name(), file.
Size(), file.IsDir())
    }
}
```

(2) 修改当前工作目录,示例代码如下：

```
//anonymous-link\example\chapter5\dir_chg.go
package main

import (
    "fmt"
    "os"
)

func main() {

    /**
    修改当前工作目录,类似于 Linux 操作系统中的 cd 命令
    */
    os.Chdir("E:\\frank\\input")

    /**
    "."表示将路径指定为当前路径,".."表示将路径指定为当前路径的上级路径
    */
    f, err := os.OpenFile("..", os.O_RDONLY, os.ModeDir)

    if err != nil {
        fmt.Println("目录打开失败：", err)
        return
    }
    defer f.Close()

    files, err := f.Readdir(-100)
    if err != nil {
        fmt.Println("错误信息：", err)
        return
    }

    for _, file := range files {
        fmt.Printf("文件名称是：%s,文件大小是：%d,是否是目录：%t\n", file.Name(), file.
Size(), file.IsDir())
    }
}
```

（3）获取当前路径，示例代码如下：

```
//anonymous - link\example\chapter5\dir_cur.go
package main

import (
    "fmt"
    "os"
)

func main() {

    /**
    获取当前工作路径
    */
    pwd, _ := os.Getwd()
    fmt.Println(pwd)

    /**
    修改当前工作目录，类似于 Linux 操作系统中的 cd 命令
    */
    os.Chdir("E:\\frank\\input")

    pwd, _ = os.Getwd()
    fmt.Println(pwd)
}
```

（4）创建目录，示例代码如下：

```
//anonymous - link\example\chapter5\dir_create.go
package main

import (
    "os"
)

func main() {

    /**
    切换工作路径
    */
    os.Chdir("E:\\frank\\input")

    /**
    切换到指定的工作路径后创建 bigdata 目录
    */
    os.Mkdir("bigdata", 0755)

}
```

（5）其他常用目录操作推荐通过以下网站进行阅读：

```
https://studyGo.com/pkgdoc
https://studyGo.com/static/pkgdoc/pkg/os.htm
```

5.1.12 Go 并发编程实例：Goroutine

1. 并行和并发概述

(1) 什么是并行(Parallel)？并行如图 5-9 所示，指在同一时刻，有多条指令在多个处理器上同时执行。

(2) 什么是并发(Concurrency)？并发如图 5-10 所示，指在同一时刻只能有一条指令执行，但多个进程指令被快速地轮换执行，使在宏观上达到具有多个进程同时执行的效果，但在微观上并不是同时执行的，只是把时间分成若干段，通过 CPU 时间片轮转使多个进程快速交替地执行。

图 5-9 任务并行示例

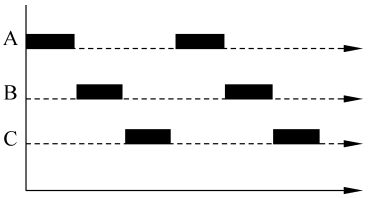


图 5-10 任务并发示例

(3) 并行和并发的区别：并行是两个队列同时使用两个 CPU 核(真正的多任务)，并发是两个队列交替使用 1 个 CPU 核(假的多任务)。

2. 常见的并发编程技术

(1) 进程并发的概念及特性主要如下。

程序：指编译好的二进制文件，保存在磁盘上，不占用系统资源。

进程：一个抽象的概念，与操作系统原理联系紧密。进程是活跃的程序，占用系统资源，在内存中执行。换句话说，程序运行起来，产生一个进程。

进程状态：进程基本的状态有 5 种。分别为初始态、就绪态(等待 CPU 分配时间片)、运行态(占用 CPU)、挂起态(等待除 CPU 以外的其他资源主动放弃 CPU)与终止态，其中初始态为进程准备阶段，常与就绪态结合来看。

在使用进程实现并发时会出现什么问题呢？

① 系统开销比较大，占用资源比较多，开启进程数量比较少。

② 在 UNIX/Linux 系统下，还会产生“孤儿进程”和“僵尸进程”。

孤儿进程：如果父进程先于子进程结束，则子进程称为孤儿进程，子进程的父进程称为 init 进程，init 进程领养孤儿进程。

僵尸进程：进程终止，父进程尚未回收，子进程残留资源(PCB)存放于内核中，变成僵尸(Zombie)进程。

在操作系统的运行过程中，可以产生很多进程。在 UNIX/Linux 系统中，正常情况下，

子进程是通过父进程 fork 创建的，子进程再创建新的进程，并且父进程永远无法预测子进程到底什么时候结束。当一个进程完成它的工作并终止之后，它的父进程需要调用系统，以便取得子进程的终止状态。

Windows 系统下的进程和 Linux 下的进程是不一样的，它比较懒惰，从来不执行任何任务，只是为线程提供执行环境，然后由线程负责执行包含在进程地址空间中的代码。当创建一个进程时，操作系统会自动创建这个进程的第 1 个线程，称为主线程。

(2) 线程并发的概念及特性主要如下。

线程：线程是轻量级的进程(Light Weight Process)本质上仍是进程(Linux 下)。

进程：独立地址空间，拥有 PCB(进程控制块)。

线程：有独立的 PCB(进程控制块)，但没有独立的地址空间(和其所在的进程共享用户空间)。

线程同步：指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其他线程为了保证数据的一致性，不能调用该功能。同步的目的是为了避免数据混乱，解决与时间有关的错误。实际上，不仅线程间需要同步，进程间、信号间等都需要同步机制，因此，所有多个控制流，共同操作一个共享资源的情况都需要同步。

常见锁的应用如下。

① 互斥量(mutex)：Linux 中提供了一把互斥锁 mutex(也称为互斥量)。每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束后解锁。资源还是共享的，线程间也还是竞争的，但通过锁就将资源的访问变成互斥操作，而后与时间有关的错误也不会产生了，但应注意同一时刻只能有一个线程持有该锁。

举个例子：当 A 线程对某个全局变量加锁访问时，如果 B 在访问前尝试加锁，则拿不到锁，B 阻塞。C 线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱问题。

综上所述，互斥锁实际上是操作系统提供的一把建议锁(又称协同锁)，建议程序中有多线程访问共享资源时使用该机制，但并没有强制限定，因此，即使有了 mutex，如果有线程不按规则访问数据，则依然会造成数据混乱问题。

② 读写锁：与互斥量类似，但读写锁允许更高的并行性。其特性为写独占，读共享。

读写锁状态：读写锁只有一把，但其具备两种状态，即读模式下加锁状态(读锁)和写模式下加锁状态(写锁)。

读写锁特性：当读写锁处于写模式加锁时，在解锁前所有对该锁加锁的线程都会被阻塞。当读写锁处于读模式加锁时，如果线程以读模式对其加锁，则会成功；如果线程以写模式加锁，则会阻塞。当读写锁处于读模式加锁时，既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。读锁、写锁并行阻塞，写锁优先级高。读写锁也叫共享独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。写独占、读共享。读写锁非常适合于对数据结构读的次数远大于写的情况。

(3) 进程和线程的区别及特性主要如下。

进程：并发执行的程序在执行过程中分配和管理资源的基本单位。

线程：进程的一个执行单元，是比进程还要小的独立运行的基本单位。一个程序至少有一个进程，一个进程至少有一个线程。

进程和线程的主要区别如下。

根本区别：进程是资源分配的最小单位，线程是程序执行的最小单位。计算机在执行程序时，会为程序创建相应的进程，在进行资源分配时，以进程为单位进行相应分配。每个进程都有相应的线程，在执行程序时，实际上执行的是相应的一系列线程。

地址空间：进程有自己独立的地址空间，每启动一个进程，系统都会为其分配地址空间，建立数据表来维护代码段、堆栈段和数据段；线程没有独立的地址空间，同一进程的线程共享本进程的地址空间。

资源拥有进程之间的资源是独立的；同一进程内的线程共享本进程的资源。

执行过程：每个独立的进程有一个程序运行的入口、顺序执行序列和程序入口，但是线程不能独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

调度单位：线程是处理器调度的基本单位，但是进程不是。由于程序执行的过程其实执行的是具体的线程，处理器处理的也是程序的相应线程，所以处理器调度的基本单位是线程。Windows 系统下，可以直接忽略进程的概念，只谈线程。因为线程是最小的执行单位，是被系统独立调度和分派的基本单位，而进程只是给线程提供执行环境。

系统开销：进程执行开销大，线程执行开销小。

(4) 协程并发的概念及主要特性如下。

协程：coroutine，也叫轻量级线程。与传统的系统级线程和进程相比，协程最大的优势在于轻量级。可以轻松创建上万个而不会导致系统资源衰竭，而线程和进程通常很难超过1万个。这也是协程被称为轻量级线程的原因。

一个线程中可以有任意多个协程，但某一时刻只能有一个协程在运行，多个协程分享该线程分配到的计算机资源。

协程不是被操作系统内核所管理的，而完全是由程序所控制的(也就是在用户态执行)，这样带来的好处就是性能可以得到很大提升，不会像线程切换那样消耗资源。

综上所述，协程是一种用户态的轻量级线程，协程的调度完全由用户控制。协程拥有自己的寄存器上下文和栈。协程在调度切换时，将寄存器上下文和栈保存到其他地方，再切回来时，恢复先前保存的寄存器上下文和栈，直接操作栈则基本没有内核切换的开销，可以用不加锁的方式访问全局变量，所以上下文的切换非常快。

子程序调用：或者称为函数，在所有语言中都是层级调用，例如 A 调用 B，B 在执行过程中又调用了 C，C 执行完毕后返回，B 执行完毕后返回，最后是 A 执行完毕，所以子程序调用是通过栈实现的，一个线程就执行一个子程序。子程序调用总是一个入口，一次返回，调用顺序是明确的，而协程的调用和子程序不同。

协程在子程序内部是可中断的，然后转而执行别的子程序，在适当时再返回来接着

执行。

多数语言在语法层面并不直接支持协程，而是通过库的方式支持，但用库的方式支持的功能也并不完整，例如，仅仅提供协程的创建、销毁与切换等能力。关于协程调度的实现理论上分为以下三类模型。

一对多：即用户态中的多个协程对应内核态的一个线程。如果在这样的轻量级线程中调用一个同步 IO 操作（例如网络通信、本地文件读写）都会阻塞其他的并发执行轻量级线程，从而无法真正达到轻量级线程本身期望达到的目标。

一对一：即用户态中的一个协程对应内核态的一个线程。虽然解决了一对多的阻塞问题，但是本质上还是线程之间的切换。

多对多：即用户态中的多个协程对应内核态的多个线程。相比一对多方案解决了阻塞问题，现在的协程调度器都使用类似的模型。

在协程中，调用一个任务就像调用一个函数一样，消耗的系统资源最少，但能达到进程、线程并发相同的效果。

在一次并发任务中，进程、线程、协程均可以实现。从系统资源消耗的角度来看，进程相当多，线程次之，协程最少。

（5）Go 并发的主要实现及特性如下：

Go 在语言级别支持协程，叫作 goroutine。Go 语言标准库提供的所有系统调用操作（包括所有同步 I/O 操作）都会将 CPU 出让给其他 goroutine。这让轻量级线程的切换管理不依赖于系统的线程和进程，也不需要依赖于 CPU 的核心数量。

有人把 Go 比作 21 世纪的 C 语言。第一是因为 Go 语言设计简单，第二是因为 21 世纪最重要的就是并行程序设计，而 Go 从语言层面就支持并发。同时，并发程序的内存管理有时是非常复杂的，而 Go 语言提供了自动垃圾回收机制。

Go 语言为并发编程而内置的上层 API 基于顺序通信进程（Communicating Sequential Processes，CSP）模型。这就意味着显式锁都是可以避免的，因为 Go 通过相对安全的通道发送和接收数据以实现同步，这大大地简化了并发程序的编写。

Go 语言中的并发程序主要使用两种手段实现，即 goroutine 和 channel。

goroutine 早期调度算法：

早期 goroutine 调度存在频繁加锁解锁问题，最好的情况就是哪个线程创建的协程就由哪个线程执行。

早期的协程调度存在资源复制的弊端，频繁地在线程间切换会增加系统开销。

goroutine 新版调度器算法（MPG）：

M：os 线程（操作系统内核提供的线程）。

G：goroutine，其包含了调度一个协程所需要的堆栈及 Instruction Pointer（IP 指令指针），以及其他一些重要的调度信息。

P：M 与 P 的中介，是实现 m：n 调度模型的关键，M 必须获得 P 才能对 G 进行调度，P 其实限定了 Go 调度的最大并发度。

P 默认和 CPU 核数相等，可按需设置。

M 要去抢占 P，如果抢到了 P，就去领取 G，如果没有任务就会从其他的 P 或者全局的任务队列获取 G。

3. goroutine 实战案例

(1) 什么是 goroutine? goroutine 是 Go 语言并行设计的核心，有人称为 go 程。goroutine 从量级上看很像协程，但它比线程更小，十几个 goroutine 可能体现在底层就是五六个线程，Go 语言内部实现了这些 goroutine 之间的内存共享。执行 goroutine 只需极少的栈内存(大概需要 4～5KB)，当然会根据相应的数据伸缩。也正因为如此，可同时运行成千上万个并发任务。goroutine 比 thread 更易用、更高效、更轻便。一般情况下，一个普通计算机运行几十个线程就有点负载过大了，但是同样的计算机却可以轻松地让成百上千个 goroutine 进行资源竞争。

(2) 创建 goroutine，代码如下：

```
//anonymous - link\example\chapter5\go_create.go
package main

import (
    "fmt"
    "time"
)

func Task(start int, end int, desc string) {
    for index := start; index <= end; index += 2 {
        fmt.Printf("%s %d\n", desc, index)
        time.Sleep(1 * time.Second)
    }
}

func main() {
    /**
    创建 goroutine:
        只需在函数调用语句前添加 Go 关键字，就可创建并发执行单元。开发人员无须了解任何执行细节，调度器会自动将其安排到合适的系统线程上执行
        在并发编程中，通常想将一个过程切分成几块，然后让每个 goroutine 各自负责一块工作，当一个程序启动时，主函数在一个单独的 goroutine 中运行，叫作 main goroutine。新的 goroutine 会用 Go 语句来创建，而 Go 语言的并发设计很轻松就可以达到这一目的

    goroutine 的特性:
        为了避免类似孤儿进程的存在，如果 main 协程挂掉，则所有协程都会挂掉
        换句话说，主 goroutine 退出后，其他的工作 goroutine 也会自动退出
    */
    go Task(10, 30, "Task Func Say: index =")

    Task(11, 30, "Main Say: index =")
}
```

(3) Goexit 函数的代码如下：

```
//anonymous - link\example\chapter5\go_exit.go
package main

import (
    "fmt"
    "runtime"
    "time"
)

func main() {
    go func() {
        defer fmt.Println("Goroutine 666666")

        func() {
            defer fmt.Println("Goroutine 88888888")

            /**
            return、Goexit() 和 os.Exit()的区别：
                return：
                    一般用于函数的返回，只能结束当前所在的函数
                Goexit()：
                    一般用于协程的退出
                  具有击穿特性，能结束当前所在的 goroutine，无论存在几层函数调用
                os.Exit()：
                  主动退出主 goroutine，换句话说，直接终止整个程序的运行
            */
            runtime.Goexit() //终止当前 goroutine
            //return
            //os.Exit(100)
            fmt.Println("AAAA")
        }()

        fmt.Println("CCCCC")
    }()

    //主 goroutine 会运行 15s，有充足的时间使上面的子 go 程代码执行完毕
    for index := 1; index <= 30; index += 2 {
        fmt.Printf("Main Say: index = %d\n", index)
        time.Sleep(1 * time.Second)
    }
}
```

5.1.13　Go 并发编程实例：channel

1. channel 的基本特性

(1) channel 是 Go 语言中的一个核心类型，可以把它看成管道，并发核心单元通过它就可以发送或者接收数据进行通信，这在一定程度上又进一步降低了编程的难度。channel 是一个数据类型，主要用来解决 go 程的同步问题及 go 程之间数据共享(数据传递)的问题。goroutine 运行在相同的地址空间，因此访问共享内存必须做好同步。goroutine 奉行通过

通信来共享内存，而不是以共享内存来通信，如图 5-11 所示。引用类型 channel 可用于多个 goroutine 通信。其内部实现了同步，确保并发安全。

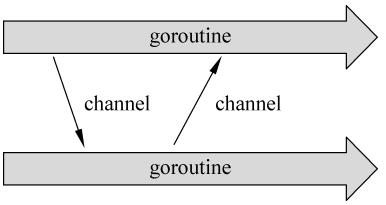


图 5-11 goroutine 通过 channel 通信示例

（2）无缓冲的 channel 是指在接收前没有能力保存任何数据值的通道。这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，这样才能完成发送和接收操作。否则通道会导致先执行发送或接收操作的 goroutine 阻塞等待。这种对通道进行发送和接收的交互行为本身就是同步的，其中任意一个操作都无法离开另一个操作单独存在。

阻塞：由于某种原因数据没有到达，当前 go 程（线程）持续处于等待状态，直到条件满足，才解除阻塞。

同步：在两个或多个 go 程（线程）间，保持数据内容一致性的机制。

图 5-12 展示了两个 goroutine 如何利用无缓冲的通道来共享一个值，步骤如下：

第 1 步，两个 goroutine 都到达通道，但哪个都没有开始执行发送或者接收操作。

第 2 步，左侧的 goroutine 将它的手伸进了通道，这模拟了向通道发送数据的行为。这时，这个 goroutine 会在通道中被锁住，直到交换完成。

第 3 步，右侧的 goroutine 将它的手放入通道，这模拟了从通道里接收数据的行为。这个 goroutine 也会在通道中被锁住，直到交换完成。

第 4 步和第 5 步，进行交换，并最终在第 6 步，两个 goroutine 都将它们的手从通道里拿出来，这模拟了被锁住的 goroutine 得到释放。两个 goroutine 现在都可以去做其他事情了。

无缓冲的 channel，创建格式如下：

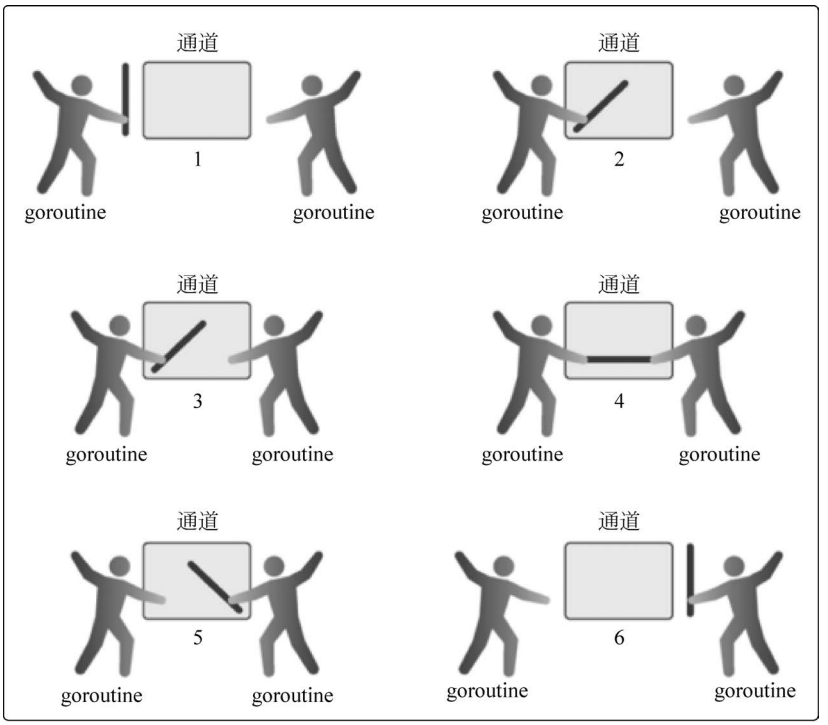
```
make(chan Type) //等价于"make(chan Type, 0)",如果没有指定缓冲区容量,则该通道就是同步的,
//因此会阻塞到发送者准备好发送和接收者准备好接收
```

（3）有缓冲的 channel 是一种在被接收前能存储一个或者多个数据值的通道。这种类型的通道并不强制要求 goroutine 之间必须同时完成发送和接收操作。通道阻塞发送和接收动作的条件也不同。只有通道中没有要接收的值时，接收动作才会被阻塞。只有通道没有可用缓冲区容纳被发送的值时，发送动作才会被阻塞。这导致有缓冲的通道和无缓冲的通道之间的一个很大的不同：无缓冲的通道保证进行发送和接收的 goroutine 会在同一时间进行数据交换；有缓冲的通道没有这种保证。示例如图 5-13 所示，步骤如下：

第 1 步，右侧的 goroutine 正在从通道接收一个值。

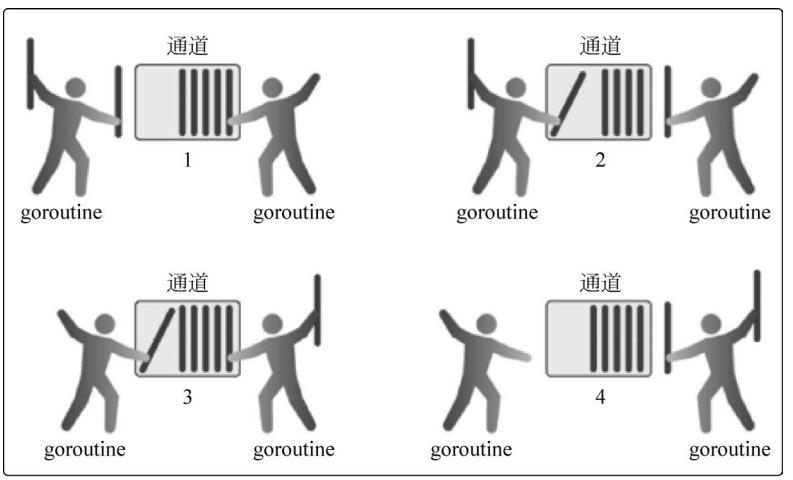
第 2 步，右侧的这个 goroutine 独立完成了接收值的动作，而左侧的 goroutine 正在将一个新值发送到通道里。

第 3 步，左侧的 goroutine 还在向通道发送新值，而右侧的 goroutine 正在从通道接收另外一个值。这个步骤里的两个操作既不是同步的，也不会互相阻塞。



使用无缓冲的通道在goroutine之间同步

图 5-12　无缓冲 channel 共享数据



使用有缓冲的通道在goroutine之间同步数据

图 5-13　有缓冲 channel 共享数据

在第 4 步，所有的发送和接收都完成，而通道里还有几个值，也有一些空间可以存更多的值。

有缓冲的 channel,创建格式如下:

```
make(chan Type, capacity) //如果给定了一个缓冲区容量,通道就是异步的。只要缓冲区有未使用
//空间用于发送数据,或还包含可以接收的数据,那么其通信就会无阻塞地进行。借助函数 len(ch)
//求取缓冲区中剩余元素的个数, cap(ch)可求取缓冲区元素容量的大小
```

2. channel 的基本使用

(1) 定义有缓冲区管道,代码如下:

```
//anonymous - link\example\chapter5\channel_buffer.go
package main

import "fmt"

func main() {
    /**
    和 map 类似,channel 也是一个对应 make 创建的底层数据结构的引用,创建 channel 的语法格式
如下:
        make(chan Type,capacity)

    以下是相关的参数说明:
        chan 是创建 channel 所需使用的关键字
        Type 是指定 channel 收发数据的类型
        capacity 是指定 channel 的大小
            当参数"capacity = 0"时,channel 是无缓冲阻塞读写的,即该 channel 无容量
            当参数"capacity > 0"时,channel 有缓冲,是非阻塞的,直到写满 capacity 个元素才阻
塞写入
    */
    s1 := make(chan int, 3) //定义一个有缓冲区的 channel

    //向 channel 写入数据
    s1 <- 110
    s1 <- 119
    s1 <- 120

    /**
    由于上面已经写了 3 个数据,此时 s1 这个 channel 的容量已经达到 3 个容量上限,即该 channel
已满
    如果 channel 已满,继续往该 channel 写入数据则会抛出异常:"fatal error: all goroutines are
asleep - deadlock!"
    */
    //s1 <- 114

    //从 channel 读取数据
    fmt.Println(<-s1)
    fmt.Println(<-s1, <-s1)

    /**
    上面的代码已经对 channel 各进行了三次读写,此时该 channel 中并没有数据
```

```
    如果 channel 无数据,从该 channel 读取数据时也会抛出异常:"fatal error: all goroutines are
asleep - deadlock!"
    */
    //fmt.Println(<-s1)
}
```

(2) 定义无缓冲区管道,代码如下:

```
//anonymous-link\example\chapter5\channel_unbuffer.go
package main

import (
    "fmt"
    "time"
)

func Read(s chan int) {
    defer fmt.Println("读端结束~~~")
    for index := 0; index < 3; index++{
        fmt.Printf("读取到 channel 的数据是: %d\n", <-s)
    }
}

func Write(s chan int, value int) {
    defer fmt.Println("写端结束...")
    for index := 100; index < value; index++{
        s <- index
    }
}

func main() {

    /**
    无缓冲区 channel 特性:
        只有在写端和读端同时准备就绪的情况下才能运行
    */
    s1 := make(chan int)              //定义一个无缓冲区的 channel
    //s1 := make(chan int, 10)        //定义一个有缓冲区的 channel,其容量为 10
    go Read(s1)

    go Write(s1, 110)

    /**
    为了让主 goroutine 阻塞,写个死循环即可
    */
    for {
        time.Sleep(1 * time.Second)
    }
}
```

(3) 关闭 channel,代码如下:

```
//anonymous - link\example\chapter5\channel_close.go
package main

import (
    "fmt"
    "runtime"
    "time"
)

func ReadChannel(s chan int) {
    defer fmt.Println("读端结束～～～")
    for index := 0; index < 5; index++{
        if index == 103 {
            /**
            程序结束时,清理掉 channel 占用的空间,只影响写入,而不影响读取
            */
            close(s)
        }

        /**
        从 channel 中接收数据,并赋值给 value,同时检查通道是否已关闭或者是否为空
        */
        value, ok := <- s
        if ! ok { //等效于"ok != true"
            fmt.Println("channel 已关闭或者管道中没有数据")
            runtime.Goexit()
        } else {
            fmt.Printf("读取到 channel 的数据是: %d\n", value)
        }
    }
}

func WriteChannel(s chan int, value int) {
    defer fmt.Println("写端结束...")
    for index := 100; index < value; index++{
        if index == 103 {
            /**
            程序结束时,清理掉 channel 占用的空间,只影响写入,而不影响读取
            */
            close(s)
        }
        s <- index
    }
}

func main() {

    s1 := make(chan int) //定义一个有缓冲区的 channel,其容量为 10
    go ReadChannel(s1)

    go WriteChannel(s1, 110)
    for {
```

```
        time.Sleep(1 * time.Second)
    }
}
```

3. 单向 channel

（1）单向 channel：在默认情况下，通道 channel 是双向的，也就是说，既可以往里面发送数据也可以从里面接收数据，但是，经常见到一个通道作为参数进行传递而只希望对方是单向使用的，要么只让它发送数据，要么只让它接收数据，这时可以指定通道的方向。

一般情况下，创建管道都是双向的，在向函数传入数据时，可以是单向的。只读的管道不能传递给只写的管道，同理，只写的管道也不能传递给只读的管道，但是双向的管道可以传递给任意单向的管道。

（2）单向管道，示例代码如下：

```
//anonymous - link\example\chapter5\channel_one.go
package main

import (
    "fmt"
    "time"
)
/**
s 的类型说明：
    "chan<- int"表示传入只写的管道
*/
func Send(s chan<- int, value int) {
    s <- value
}
/**
r 的类型说明：
    "<-chan int"表示传入只读的管道
*/
func Receive(r <-chan int) {
    fmt.Printf("管道中的数据为：%d\n", <-r)
}

func main() {

    //创建管道
    s1 := make(chan int, 5)

    go Receive(s1)

    go Send(s1, 110)

    for {
        time.Sleep(1 * time.Second)
    }
}
```

4. 单向 channel 应用案例：生产者消费者模型

(1) 什么是生产者消费者模型？单向 channel 最典型的应用是生产者消费者模型。

生产者消费者模型是指某个模块(函数等)负责产生数据，这些数据由另一个模块来负责处理(此处的模块是广义的，可以是类、函数、go 程、线程、进程等)。产生数据的模块就形象地被称为生产者，而处理数据的模块就被称为消费者。

仅仅抽象出生产者和消费者，还不是生产者/消费者模型。该模式还需要有一个缓冲区处于生产者和消费者之间，作为一个中介。生产者把数据放入缓冲区，而消费者从缓冲区取出数据。

举一个寄信的例子来辅助理解，假设要寄一封平信，大致过程为①把信写好——相当于生产者制造数据；②把信放入邮筒——相当于生产者把数据放入缓冲区；③邮递员把信从邮筒取出——相当于消费者把数据取出缓冲区；④邮递员把信拿去邮局做相应的处理——相当于消费者处理数据。

那么，这个缓冲区有什么用呢？为什么不让生产者直接调用消费者的某个函数，直接把数据传递过去，而画蛇添足般地设置一个缓冲区呢？缓冲区的好处大概如下。

解耦：

假设生产者和消费者分别是两个类。如果让生产者直接调用消费者的某种方法，则生产者对于消费者就会产生依赖(也就是耦合)。将来如果消费者的代码发生变化，则可能会直接影响到生产者，而如果两者都依赖于某个缓冲区，则两者之间不直接依赖，耦合度也就相应地降低了。

接着上述的例子，如果不使用邮筒(缓冲区)，需要把信直接交给邮递员。那就必须认识谁是邮递员。这就产生了和邮递员之间的依赖(相当于生产者和消费者的强耦合)。万一哪天邮递员换人了，还要重新认识下一个邮递员(相当于消费者变化导致修改生产者代码)，而邮筒相对来讲比较固定，依赖它的成本也比较低(相当于和缓冲区之间的弱耦合)。

处理并发：

生产者直接调用消费者的某种方法还有另一个弊端。由于函数调用是同步的(或者叫阻塞)，在消费者的方法没有返回之前，生产者只好一直等在那边。万一消费者处理数据很慢，生产者只能无端地浪费时间。

使用了生产者消费者模式之后，生产者和消费者可以是两个独立的并发主体。生产者把制造出来的数据往缓冲区一丢，就可以再去生产下一个数据了。基本上不用依赖消费者的处理速度。

其实当初这个生产者消费者模式主要用来处理并发问题。

从寄信的例子来看，如果没有邮筒，就得拿着信傻站在路口等邮递员过来收(相当于生产者阻塞)；又或者邮递员得挨家挨户问，谁要寄信(相当于消费者轮询)。

缓存(异步处理)：

如果生产者制造数据的速度时快时慢，缓冲区的好处就体现出来了。当数据制造得快时，消费者来不及处理，未处理的数据可以暂时存在缓冲区中。等生产者的制造速度慢下

来，消费者再慢慢处理掉。

假设邮递员一次只能带走 1000 封信。万一某次碰上情人节送贺卡，需要寄出去的信超过 1000 封，这时邮筒这个缓冲区就派上用场了。邮递员把来不及带走的信暂存在邮筒中，等下次过来时再拿走。

（2）生产者消费者模型的代码如下：

```
//anonymous - link\example\chapter5\channel_queue.go
package main

import (
    "fmt"
    "strconv"
    "time"
)
//定义生产者，假设生产者不消费
func Producer(p chan <- string) {
    defer fmt.Println("生产蛋糕结束")
    for index := 1; index <= 10; index++{
        p <- "生产了" + strconv.Itoa(index) + "个蛋糕。"
    }
}
//定义消费者，假设消费者不生产
func Consumer(c <- chan string) {
    defer fmt.Println("吃饱了")
    for index := 1; index <= 8; index++{
        fmt.Println(<- c)
    }
}

func main() {
    s1 := make(chan string, 10)

    go Producer(s1)

    go Consumer(s1)

    for {
        time.Sleep(1 * time.Second)
    }
}
```

5. channel 应用案例：定时器

Go 语言自带 time 包，包里面定义了定时器的结构，代码如下：

```
type Timer struct {
    C <- chan Time
    r runtimeTimer
}
```

在定时时间到达之前，没有数据会写入 C，如果这时读 C，则会一直阻塞，当时间到了以后系统会向 time.c 文件中写入当前时间，此时阻塞解除。

5.1.14 Go 并发编程实例：select

1. select 概述

Go 里面提供了一个关键字 select，通过 select 可以监听 channel 上的数据流动。有时希望能够借助 channel 发送或接收数据，并避免因为发送或者接收导致的阻塞，尤其是当 channel 没有准备好写或者读时，select 语句就可以实现这样的功能。select 的用法与 switch 语言非常类似，由 select 开始一个新的选择块，每个选择条件由 case 语句来描述。与 switch 语句相比，select 有比较多的限制，其中最大的一条限制就是每个 case 语句里必须有一个 I/O 操作。

2. select 的应用案例

(1) select 的示例代码如下：

```
//anonymous - link\example\chapter5\select_usage.go
package main

import (
    "fmt"
    "time"
)

func main() {

    s1 := make(chan int, 1)
    //s1 := make(chan int, 0) //无缓冲的 channel
    number := 1

    for {
        /**
        使用 select 关键字来监听指定 channel 的读写情况
        */
        select {
        case s1 <- number:
            fmt.Println("奇数：", number)
            number++
            time.Sleep(time.Second * 1)

        case <- s1:
            fmt.Println("偶数：", number)
            number++
            time.Sleep(time.Second * 1)

            /**
            当读取和写入(I/O 操作)都不满足的情况下，就会执行默认的条件，需要将 channel 的
容量设置为 0，这样就可以看到效率了
```

```
            */
        default:
            fmt.Println("======")
            time.Sleep(time.Second * 1)
        }
    }
}
```

（2）select 实现斐波那契数列，示例代码如下：

```
//anonymous - link\example\chapter5\select_fibo.go
package main

import (
    "fmt"
    "time"
)

func FibonacciSeriesWrite(fib chan int) {
    a, b := 1, 1
    fmt.Printf("%d\n%d\n", a, b)
    for {
        select {
        case fib <- a + b:              //写入数据
                a, b = b, a+b
        }
    }
}

func FibonacciSeriesRead(fib chan int) {
    for {
        fmt.Println(<-fib)              //读取数据
        time.Sleep(time.Second)
    }
}

func main() {
    //初始化 channel
    s1 := make(chan int)

    go FibonacciSeriesWrite(s1)

    go FibonacciSeriesRead(s1)

    for {
        time.Sleep(time.Second)
    }
}
```

（3）select 实现超时，示例代码如下：

```
//anonymous - link\example\chapter5\select_timeout.go
package main

import (
    "fmt"
    "os"
    "time"
)

func main() {

    s1 := make(chan int, 1)

    go func() {
        for {
            select {
            case s1 <- 110:
                fmt.Println("写入 channel 数据")
                fmt.Println("当前时间为:", time.Now())
            /**
            设置定时器,当 channel 中的数据在 30s 内没有被消费时就会退出程序
            */
            case <- time.After(time.Second * 30):
                fmt.Println("程序响应超过 30s,程序已退出!")
                fmt.Println("当前时间为:", time.Now())
                os.Exit(100)
            }
        }
    }()

    /**
    设置一次性定时器,仅消费一次数据
    */
    time.AfterFunc(
        time.Second * 3,
        func() {
            fmt.Printf("获取 channel 中的数据为: %d\n", <- s1)
        })

    for {
        time.Sleep(time.Second)
    }
}
```

5.1.15 Go 并发编程：传统的同步工具锁

1. 传统的同步工具锁

(1) 锁的作用：锁就是某个 go 程(线程)在访问某个资源时先锁住,防止其他 go 程的访问,等访问完毕解锁后其他 go 程再来加锁进行访问。这和生活中加锁使用公共资源相似,

例如公共卫生间。锁的作用是为了在并发编程时让数据一致。

（2）死锁问题：死锁是指两个或两个以上的进程在执行过程中，由于竞争资源或者由于彼此通信而造成的一种阻塞的现象，若无外力作用，则它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁。在使用锁的过程中，很容易造成死锁，在开发中应该尽量避免死锁，死锁的示例代码如下：

```
//anonymous - link\example\chapter5\deadlock_create.go
package main

import (
    "fmt"
)

func main() {

    //注意，无缓冲区 channel 在读端和写端都准备就绪时不阻塞
    s1 := make(chan int)

    /**
    主线程写入：
        主 go 程在写入数据时，但此时读端并没有准备就绪，因此代码会在该行阻塞，称之为死锁
        在开发中一定要使用锁机制时需要注意避免死锁现象
    */
    s1 <- 5

    /**
    子线程读取：
        通过上面的解释，相信大家心里也清楚，代码在上一行已经阻塞了，没有机会执行到当前
行，即没有开启子 go 程
    */
    go func() {
        fmt.Println(<- s1)
    }()
}
```

（3）死锁案例解决方案，示例代码如下：

```
//anonymous - link\example\chapter5\deadlock_solve.go
package main

import (
    "fmt"
    "time"
)

func main() {

    //注意，无缓冲区 channel 在读端和写端都准备就绪时不阻塞
    s1 := make(chan int)
```

```
    /**
    子线程读取：
        先开启一个子 go 程用于读取无缓冲 channel 中的数据，此时由于写端未就绪，因此子 go 程
会处于阻塞状态，但并不会影响主 go 程，因此代码可以继续向下执行
    */
    go func() {
        fmt.Println(<- s1)
    }()

    /**
    主线程写入：
        此时读端（子 go 程）处于阻塞状态并正在准备读取数据，主 go 程在写入数据时子 go 程会立
即执行
    */
    s1 <- 5

    for {
        time.Sleep(time.Second)
    }
}
```

2. 互斥锁

（1）什么是互斥锁？每个资源都对应于一个可称为互斥锁的标记，这个标记用来保证在任意时刻，只能有一个 go 程访问该资源。其他的 go 程只能等待。互斥锁是传统并发编程对共享资源进行访问控制的主要手段，它由标准库 sync 中的 Mutex 结构体类型表示。sync.Mutex 类型只有两个公开的指针方法，即 Lock 和 Unlock。Lock 用于锁定当前的共享资源，Unlock 用于进行解锁。在使用互斥锁时，一定要注意：对资源操作完成后，一定要解锁，否则会出现流程执行异常、死锁等问题。通常借助 defer 锁定后，立即使用 defer 语句保证互斥锁及时解锁。

（2）互斥锁的示例代码如下：

```
//anonymous - link\example\chapter5\lock_mutex.go
package main

import (
    "fmt"
    "sync"
    "time"
)
var mutex sync.Mutex                          //定义互斥锁
func MyPrint(data string) {
    mutex.Lock()                              //添加互斥锁
    defer mutex.Unlock()                      //使用结束时自动解锁

    for _, value := range data {              //迭代字符串的每个字符并打印
        fmt.Printf("%c", value)
        time.Sleep(time.Second)               //模拟 go 程在执行任务
    }
```

```
    fmt.Println()
}

func Show01(s1 string) {
    MyPrint(s1)
}

func Show02() {
    MyPrint("Jason Yin")
}

func main() {
    /**
    虽然在主 go 中开启了两个子 go 程,但由于两个子 go 程有互斥锁的存在,因此一次只能运行一
个 go 程
     */
    go Show01("张三")
    go Show02()
    //主 go 程设置充足的时间让所有子 go 程执行完毕,因为主 go 程结束会将所有的子 go 程杀死
    time.Sleep(time.Second * 30)
}
```

3. 读写锁

(1) 什么是读写锁？互斥锁的本质是当一个 goroutine 访问时,其他 goroutine 都不能访问。这样在资源同步及避免竞争的同时也降低了程序的并发性能。程序由原来的并行执行变成了串行执行。其实,当对一个不会变化的数据只做读操作时,是不存在资源竞争问题的。因为数据是不变的,不管怎么读取,多少 goroutine 同时读取都是可以的,所以问题不是出在“读”上,主要出在修改上,也就是“写”。修改的数据要同步,这样其他 goroutine 才可以感知到,所以真正的互斥应该是读取和修改、修改和修改之间,读和读之间是没有互斥操作的必要的,因此,衍生出另外一种锁,叫作读写锁。

读写锁可以让多个读操作并发,即同时读取,但是对于写操作是完全互斥的。也就是说,当一个 goroutine 进行写操作时,其他 goroutine 既不能进行读操作,也不能进行写操作。

Go 中的读写锁由结构体类型 sync.RWMutex 表示。在此类型的方法集合中包含两组方法：

一组是对写操作的锁定和解锁,简称写锁定和写解锁：

```
func (*RWMutex)Lock()
func (*RWMutex)Unlock()
```

另一组表示对读操作的锁定和解锁,简称读锁定与读解锁：

```
func (*RWMutex)RLock()
func (*RWMutex)RUnlock()
```

（2）读写锁的示例代码如下：

```
//anonymous - link\example\chapter5\lock_rwmutex.go
package main

import (
    "fmt"
    "math/rand"
    "sync"
    "time"
)

var (
    number int
    rwlock sync.RWMutex              //定义读写锁
)

func MyRead(n int) {
    rwlock.RLock()                   //添加读锁
    defer rwlock.RUnlock()           //使用结束时自动解锁
    fmt.Printf("[%d] Goroutine 读取的数据为：%d\n", n, number)
}

func MyWrite(n int) {
    rwlock.Lock()                    //添加写锁
    defer rwlock.Unlock()            //使用结束时自动解锁
    number = rand.Intn(100)
    fmt.Printf("%d Goroutine 写入的数据为：%d\n", n, number)
}

func main() {

    //创建写端
    for index := 201; index <= 205; index++{
        go MyWrite(index)
    }

    //创建读端
    for index := 110; index <= 130; index++{
        go MyRead(index)
    }

    for {
        time.Sleep(time.Second)
    }
}
```

4. 条件变量

（1）什么是条件变量？条件变量的作用并不能保证在同一时刻仅有一个 go 程访问某个共享的数据资源，而是在对应的共享数据的状态发生变化时，通知阻塞在某个条件上的

go 程。条件变量不是锁，在并发中不能达到同步的目的，因此条件变量总是与锁一起使用。

Go 标准库中的 sync.Cond 类型代表了条件变量。条件变量要与锁（互斥锁，或者读写锁）一起使用。成员变量 L 代表与条件变量搭配使用的锁。

（2）条件变量的案例，示例代码如下：

```
//anonymous - link\example\chapter5\cond.go
package main

import (
    "fmt"
    "runtime"

    "math/rand"
    "sync"
    "time"
)

/*
*
创建全局条件变量
*/
var cond sync.Cond

//生产者
func producer(out chan<- int, idx int) {
    for {
        /**
         条件变量对应互斥锁加锁，即在生产数据时得加锁
         */
        cond.L.Lock()

        /**
         产品区满 3 个就等待消费者消费
         */
        for len(out) == 3 {
            /**
             挂起当前 go 程，等待条件变量满足，被消费者唤醒，该函数的作用可归纳为以下三点：
             (1)阻塞等待条件变量满足
             (2)释放已掌握的互斥锁，相当于 cond.L.Unlock()。注意：两步为一个原子操作
             (3)当被唤醒，Wait()函数返回时，解除阻塞并重新获取互斥锁相当于 cond.L.Lock()
             */
            cond.Wait()
        }

        /**
         产生一个随机数，写入 channel 中（模拟生产者）
         */
        num := rand.Intn(1000)
        out <- num
```

```
            fmt.Printf("%dth 生产者,产生数据 %3d, 公共区剩余%d个数据\n", idx, num, len(out))

            /**
             单发通知,给一个正等待(阻塞)在该条件变量上的 goroutine(go 程)发送通知。换句话说,
唤醒阻塞的消费者
             */
            //cond.Signal()

            /**
             广播通知,给正在等待(阻塞)在该条件变量上的所有 goroutine(线程)发送通知
             */
            cond.Broadcast()

            /**
             生产结束,解锁互斥锁
             */
            cond.L.Unlock()

            /**
             生产完休息一会,给其他 go 程执行的机会
             */
            time.Sleep(time.Second)
        }
    }

    //消费者
    func consumer(in <- chan int, idx int) {
        for {
            /**
             条件变量对应互斥锁加锁(与生产者是同一个)
             */
            cond.L.Lock()

            /**
             产品区为空,等待生产者生产
             */
            for len(in) == 0 {
                /**
                 挂起当前 go 程,等待条件变量满足,被生产者唤醒
                 */
                cond.Wait()
            }

            /**
             将 channel 中的数据读走(模拟消费数据)
             */
            num := <- in
            fmt.Printf("[%dth] 消费者, 消费数据 %3d,公共区剩余%d个数据\n", idx, num, len(in))
            /**
             唤醒阻塞的生产者
             */
```

```
            cond.Signal()
            /**
             消费结束,解锁互斥锁
             */
            cond.L.Unlock()

            /**
             消费完休息一会,给其他 go 程执行的机会
             */
            time.Sleep(time.Millisecond * 500)
        }
    }

    func main() {

        /**
         设置随机数种子
         */
        rand.Seed(time.Now().UNIXNano())

        /**
         产品区(公共区)使用 channel 模拟
         */
        product := make(chan int, 3)

        /**
         创建互斥锁和条件变量(申请内存空间)
         */
        cond.L = new(sync.Mutex)

        /**
         创建 3 个生产者
         */
        for i := 101; i < 103; i++{
            go producer(product, i)
        }

        /**
         创建 5 个消费者
         */
        for i := 211; i < 215; i++{
            go consumer(product, i)
        }
        for {
            //主 go 程阻塞,不结束
            runtime.GC()
        }
    }
```

5. WaitGroup

(1) 什么是 WaitGroup? WaitGroup 用于等待一组 go 程的结束。父线程调用 Add 方法来设定应等待的 go 程的数量。每个被等待的 go 程在结束时应调用 Done 方法。同时，

主 go 程里可以调用 Wait 方法阻塞至所有 go 程结束。大致步骤如下。

① 创建 WaitGroup 对象,命令如下:

```
var wg sync.WaitGroup
```

② 添加主 go 程等待的子 go 程个数,命令如下:

```
wg.Add(数量)
```

③ 在各个子 go 程结束时,调用 defer wg.Done()。将主 go 等待的数量－1。注意:实名子 go 程需传地址。

④ 在主 go 程中等待,命令如下:

```
wg.wait()
```

(2) WaitGroup 的示例代码如下:

```
//anonymous - link\example\chapter5\waitgroup.go
package main

import (
    "fmt"
    "sync"
    "time"
)

func son1(group *sync.WaitGroup) {
    /**
    在各个子 go 程结束时一定要调用 Done 方法,它会通知 WaitGroup 该子 go 程执行完毕
    */
    defer group.Done()
    time.Sleep(time.Second * 3)
    fmt.Println("son1 子 go 程结束...")
}

func son2(group *sync.WaitGroup) {
    /**
    在各个子 go 程结束时一定要调用 Done 方法,它会通知 WaitGroup 该子 go 程执行完毕
    */
    defer group.Done()
    time.Sleep(time.Second * 5)
    fmt.Println("son2 子 go 程结束")
}

func son3(group *sync.WaitGroup) {
    /**
    在各个子 go 程结束时一定要调用 Done 方法,它会通知 WaitGroup 该子 go 程执行完毕
    */
```

```
    defer group.Done()
    time.Sleep(time.Second * 1)
    fmt.Println("son3 子 go 程结束～～～")
}

func main() {
    /**
    创建 WaitGroup 对象
    */
    var wg sync.WaitGroup

    /**
    添加主 go 程等待的子 go 程个数,该数量有 3 种情况:
    1.当主 go 程添加的子 go 程个数和实际子 go 程数量相等时,需要等待所有的子 go 程执行完毕后主 go 程才能正常退出。
    2.当主 go 程添加的子 go 程个数和实际子 go 程数量不等时有以下两种情况:
    (1)小于的情况:只需等待指定的子 go 程数量执行完毕后主 go 程就会退出,尽管还有其他的子 go 程没有运行完成。
    (2)大于的情况:最终会抛出异常 fatal error: all goroutines are asleep deadlock!
    */
    wg.Add(2)

    /**
    执行子 go 程
    */
    go son1(&wg)
    go son2(&wg)
    go son3(&wg)

    /**
    在主 go 程中等待,即主 go 程为阻塞状态
    */
    wg.Wait()
}
```

5.1.16　Go 网络编程：套接字

1. 套接字(Socket)网络概述

(1) 什么是协议？从应用的角度出发,协议可理解为规则,是数据传输和数据解释的规则。假设,A、B 双方欲传输文件。规定如下:

第 1 次,传输文件名,接收方接收到文件名,应答 OK 给传输方。

第 2 次,发送文件的尺寸,接收方接收到该数据再次应答一个 OK。

第 3 次,传输文件内容。同样,接收方完成接收数据后应答 OK 表示文件内容接收成功。

由此,无论 A、B 之间传递何种文件都是通过三次数据传输来完成的。A、B 之间形成了一个最简单的数据传输规则。双方都按此规则发送、接收数据。A、B 之间达成的这个相互

遵守的规则即为协议。

这种仅在A、B之间被遵守的协议称为原始协议。

此协议被更多的人采用，不断地增加、改进、维护、完善。最终形成一个稳定的、完整的文件传输协议，被广泛地应用于各种文件传输过程中。该协议就成为一个标准协议。最早的FTP就是由此衍生而来的。

典型协议：

应用层：常见的协议有HTTP和FTP。超文本传输协议(Hyper Text Transfer Protocol，HTTP)是互联网上应用最为广泛的一种网络协议。FTP为文件传输协议(File Transfer Protocol)。

传输层：常见协议有TCP/UDP。传输控制协议(Transmission Control Protocol，TCP)是一种面向连接的、可靠的、基于字节流的传输层通信协议。用户数据报文协议(User Datagram Protocol，UDP)是OSI参考模型中的一种无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。

网络层：常见协议有IP、ICMP、IGMP。IP是因特网互联协议(Internet Protocol)。ICMP是因特网控制报文协议(Internet Control Message Protocol)，它是TCP/IP协议簇的一个子协议，用于在IP主机、路由器之间传递控制消息。IGMP是因特网组管理协议(Internet Group Management Protocol)，是因特网协议家族中的一个组播协议。该协议运行在主机和组播路由器之间。

链路层：常见协议有ARP、RARP。ARP是正向地址解析协议(Address Resolution Protocol)，通过已知的IP，寻找对应主机的MAC地址。RARP是反向地址转换协议，通过MAC地址确定IP地址。

(2) 什么是Socket？Socket的英文含义是插座、插孔，一般称为套接字，用于描述IP地址和端口。可以实现不同程序间的数据通信。

Socket起源于UNIX，而UNIX的基本哲学之一就是“一切皆文件”，即文件都可以用“打开”→“读写”→“关闭”模式来操作。

Socket就是该模式的一个实现，网络的Socket数据传输是一种特殊的I/O，Socket也是一种文件描述符。Socket也具有一个类似于打开文件的函数调用：Socket()，该函数返回一个整型的Socket描述符，随后的连接建立、数据传输等操作都是通过该Socket实现的。

在TCP/IP中，“IP地址＋TCP或UDP端口号”唯一标识网络通信中的一个进程。“IP地址＋端口号”就对应一个Socket。欲建立连接的两个进程各自有一个Socket来标识，那么这两个Socket组成的Socket Pair就唯一标识一个连接，因此可以用Socket来描述网络连接的一对一关系。

常用的Socket类型有两种：

流式Socket(SOCK_STREAM)：流式是一种面向连接的Socket，针对面向连接的TCP服务应用。

数据报文式 Socket(SOCK_DGRAM)：数据报文式 Socket 是一种无连接的 Socket，对应无连接的 UDP 服务应用。

套接字的内核实现较为复杂，简化的结构如图 5-14 所示。

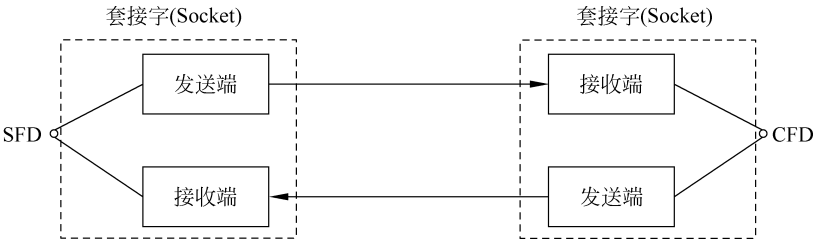


图 5-14　套接字简化的结构

(3) 网络应用程序设计模式及优缺点如下。

C/S 模式：传统的网络应用设计模式，客户端(Client)/服务器(Server)模式。需要在通信两端各自部署客户机和服务器来完成数据通信。

优点：①客户端位于目标主机上，可以保证性能，将数据缓存至客户端本地，从而提高数据传输效率；②一般来讲客户端和服务器端程序由一个开发团队创作，所以它们之间所采用的协议相对灵活。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯所采用的通信协议，即为 FTP 的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选 C/S 模式进行开发。如知名的网络游戏《魔兽世界》。三维画面，数据量庞大，使用 C/S 模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

缺点：①由于客户端和服务器端都需要有一个开发团队来完成开发，所以工作量将成倍提升，开发周期较长；②从用户角度出发，需要将客户端安装至用户主机上，对用户主机的安全性构成威胁。这也是很多用户不愿使用 C/S 模式应用程序的重要原因。

B/S 模式：浏览器(Browser)/服务器(Server)模式。只需在一端部署服务器，而另外一端使用每台计算机都默认配置的浏览器即可完成数据的传输。

优点：①B/S 模式相比 C/S 模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其开发工作量较小，只需开发服务器端；②另外由于其采用浏览器显示数据，因此移植性非常好，不受平台限制。如早期的偷菜游戏，在各个平台上都可以完美运行。

缺点：①B/S 模式的缺点也较明显。由于使用第三方浏览器，因此网络应用支持受限；②没有将客户端放到对方主机上，缓存数据不尽如人意，从而使传输数据量受到限制。应用的观感大打折扣；③必须与浏览器一样，采用标准 HTTP 进行通信，协议选择不灵活。

综上所述，在开发过程中，模式的选择由上述各自的特点决定。应根据实际需求选择应用程序设计模式。

2. TCP 的 Socket 编程实战案例

(1) 简单 C/S 模型通信，结构如图 5-15 所示。

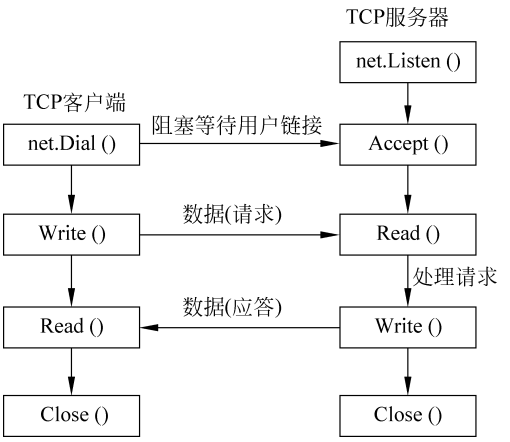


图 5-15　简单 C/S 模型通信结构

服务器端使用 Listen 函数创建监听 Socket，代码如下：

```
//anonymous - link\example\chapter5\socket_server.go
package main

import (
    "fmt"
    "net"
)

func main() {

    /**
    使用 Listen 函数创建监听 Socket，其函数签名如下：
        func Listen(network, address string) (Listener, error)
    以下是对函数签名的参数说明：
        network 用于指定服务器端 Socket 的协议，如 tcp/udp，注意此处是小写字母
        address 用于指定服务器端监听的 IP 地址和端口号，如果不指定地址，则默认监听当前服
务器的所有 IP 地址
    */
    socket, err := net.Listen("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("开启监听失败，错误原因：", err)
        return
    }
    defer socket.Close()
    fmt.Println("开启监听...")
    for {
        /**
        等待客户端连接请求
        */
        conn, err := socket.Accept()
        if err != nil {
            fmt.Println("建立连接失败，错误原因：", err)
```

```
            return
        }
        defer conn.Close()
        fmt.Println("建立连接成功,客户端地址是:", conn.RemoteAddr())

        /**
        接收客户端数据
        */
        buf := make([]Byte, 1024)
        conn.Read(buf)
        fmt.Printf("读取到客户端的数据为:%s\n", string(buf))

        /**
        将数据发送给客户端
        */
        tmp := "Blog 地址:https://blog.csdn.net/u014374009"
        conn.Write([]Byte(tmp))
    }
}
```

客户端使用 Dial 函数链接服务器端,代码如下:

```
//anonymous - link\example\chapter5\socket_client.go
package main

import (
    "fmt"
    "net"
)

func main() {

    /**
    使用 Dial 函数连接服务器端,其函数签名如下:
        func Dial(network, address string) (Conn, error)
    以下是对函数签名的各参数说明:
        network 用于指定客户端 Socket 的协议,如 tcp/udp,该协议应该和需要连接服务器端的协
议一致
        address 用于指定客户端需要连接服务器端的 Socket 信息,即指定服务器端的 IP 地址和
端口
    */
    conn, err := net.Dial("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("连接服务器端出错,错误原因:", err)
        return
    }
    defer conn.Close()
    fmt.Println("与服务器端连接建立成功...")
    /**
    给服务器端发送数据
```

```
    */
    conn.Write([]Byte("服务器端,请问博客地址的 URL 是多少呢?"))

    /**
    获取服务器的应答
    */
    var buf = make([]Byte, 1024)
    conn.Read(buf)
    fmt.Printf("从服务器端获取的数据为:%s\n", string(buf))
}
```

(2) 并发 C/S 模型通信,示例代码如下:

```
//anonymous-link\example\chapter5\socket_concurrency_server.go
package main

import (
    "fmt"
    "net"
    "strings"
)

func HandleConn(conn net.Conn) {
    //函数调用完毕,自动关闭 conn
    defer conn.Close()

    //获取客户端的网络地址信息
    addr := conn.RemoteAddr().String()
    fmt.Println(addr, " connect successful")

    buf := make([]Byte, 2048)

    for {
        //读取用户数据
        n, err := conn.Read(buf)
        if err != nil {
            fmt.Println("err = ", err)
            return
        }
        fmt.Printf("[%s]: %s\n", addr, string(buf[:n]))
        fmt.Println("len = ", len(string(buf[:n])))

        //if "exit" == string(buf[:n-1]) { //nc 测试,发送时只有 \n
        if "exit" == string(buf[:n-2]) {   //自己写的客户端测试,发送时多了两个字
                                           //符,即"\r\n"
            fmt.Println(addr, " exit")
            return
        }

        //把数据转换为大写,再给用户发送
```

```
        conn.Write([]Byte(strings.ToUpper(string(buf[:n]))))
    }
}

func main() {

    /**
     使用 Listen 函数创建监听 Socket,其函数签名如下:
        func Listen(network, address string) (Listener, error)
     以下是对函数签名的参数说明:
        network 用于指定服务器端 Socket 的协议,如 tcp/udp,注意此处是小写字母
        address 用于指定服务器端监听的 IP 地址和端口号,如果不指定地址,则默认监听当前服
务器的所有 IP 地址
     */
    socket, err := net.Listen("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("开启监听失败,错误原因:", err)
        return
    }
    defer socket.Close()
    fmt.Println("开启监听...")

    //接收多个用户
    for {
        /**
            等待客户端连接请求
         */
        conn, err := socket.Accept()
        if err != nil {
            fmt.Println("建立连接失败,错误原因:", err)
            return
        }

        //处理用户请求,新建一个 go 程
        go HandleConn(conn)
    }
}
```

客户端使用 Dial 函数连接服务器端,代码如下:

```
//anonymous - link\example\chapter5\socket_concurrency_client.go
package main

import (
    "fmt"
    "net"
    "strconv"
)

func main() {
```

```
    /**
    使用 Dial 函数连接服务器端,其函数签名如下:
        func Dial(network, address string) (Conn, error)
    以下是对函数签名的各参数说明:
        network 用于指定客户端 Socket 的协议,如 tcp/udp,该协议应该和需要连接服务器端的协
议一致
        address 用于指定客户端需要连接服务器端的 Socket 信息,即指定服务器端的 IP 地址和
端口
     */
    conn, err := net.Dial("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("连接服务器端出错,错误原因:", err)
        return
    }
    defer conn.Close()
    fmt.Println("与服务器端建立连接成功...")

    /**
    定义需要发送的数据,第 1 次给服务器端发送要发的长度
     */
    data := []Byte("服务器端,请问博客地址的 URL 是多少呢?")
    lenData := len(data)

    /**
    给服务器端发送数据的长度
     */
    conn.Write([]Byte(strconv.Itoa(lenData)))

    /**
    获取服务器的应答
     */
    var buf = make([]Byte, 1024)
    conn.Read(buf)
    fmt.Printf("从服务器端获取的数据为:%s\n", string(buf))

    /**
    第 2 次给服务器发送数据
     */
    conn.Write(data)
    conn.Read(buf)
    fmt.Printf("获取的数据为:%s\n", string(buf))
}
```

3. UDP 的 Socket 编程实战案例

(1) UDP 与 TCP 的主要区别为①TCP 是面向连接的,而 UDP 是面向无连接的,TCP 在建立端口连接时分别要进行三次握手和四次挥手,所以说 TCP 是可靠的连接,而 UDP 是不可靠的连接;②TCP 是流式传输,可能会出现粘包问题,UDP 是数据报文传输,UDP 可能会出现丢包问题。粘包问题可以通过发送数据包的长度解决,丢包问题可以通过为每个数据报文添加标识位解决;③TCP 要求系统资源较多,UDP 要求系统资源较少,TCP 需要

创建连接再进行通信，所以效率要比 UDP 低；④TCP 程序结构较复杂，UDP 程序结构较简单；⑤TCP 可以保证数据的准确性，而 UDP 则不能保证数据的准确性。

各自的应用场景如下。

TCP 的应用场景：例如文件传输、重要数据传输等。

UDP 的应用常见：例如打电话、直播等。

（2）简单 C/S 模型通信，示例代码如下：

```
//anonymous - link\example\chapter5\udp_server.go
package main

import (
    "fmt"
    "net"
)

func main() {
    /**
    创建监听的地址，并且指定 UDP
    */
    udp_addr, err := net.ResolveUDPAddr("udp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println("获取监听地址失败，错误原因：", err)
        return
    }

    /**
    创建数据通信 Socket
    */
    conn, err := net.ListenUDP("udp", udp_addr)
    if err != nil {
        fmt.Println("开启 UDP 监听失败，错误原因：", err)
        return
    }
    defer conn.Close()

    fmt.Println("开启监听...")

    buf := make([]Byte, 1024)

    /**
    通过 ReadFromUDP 可以读取数据，可以返回如下 3 个参数。
        dataLength:数据的长度
        raddr:远程的客户端地址
        err:错误信息
    */
    dataLength, raddr, err := conn.ReadFromUDP(buf)
    if err != nil {
        fmt.Println("获取客户端传递数据失败，错误原因：", err)
```

```
        return
    }
    fmt.Println("获取客户端的数据为：", string(buf[:dataLength]))

    /**
    写回数据
     */
    conn.WriteToUDP([]Byte("服务器端已经收到数据"), raddr)
}
```

客户端使用 Dial 函数连接服务器端，示例代码如下：

```
//anonymous - link\example\chapter5\udp_client.go
package main

import (
    "fmt"
    "net"
)

func main() {
    /**
    使用 Dial 函数连接服务器端，其函数签名如下：
        func Dial(network, address string) (Conn, error)
    以下是对函数签名的各参数说明：
        network 用于指定客户端 Socket 的协议，如 tcp/udp，该协议应该和需要连接服务器端的协
议一致
        address 用于指定客户端需要连接服务器端的 Socket 信息，即指定服务器端的 IP 地址和
端口
     */
    conn, err := net.Dial("udp", "127.0.0.1:9999")
    if err != nil {
        fmt.Println("连接服务器端出错，错误原因：", err)
        return
    }
    defer conn.Close()
    fmt.Println("与服务器端建立连接成功...")

    /**
    给服务器端发送数据
     */
    conn.Write([]Byte("Hi,My name is Jason Yin."))

    /**
    读取服务器端返回的数据
     */
    tmp := make([]Byte, 1024)
    n, _ := conn.Read(tmp)
    fmt.Println("获取服务器返回的数据为：", string(tmp[:n]))
}
```

5.1.17　Go 网络编程实例：HTTP 编程

1. 网络编程中的 HTTP 概述

一个 Web 服务器也被称为 HTTP 服务器，它通过超文本传输协议（Hyper Text Transfer Protocol，HTTP）与客户端通信。这个客户端通常指的是 Web 浏览器（其实手机端客户端内部也是由浏览器实现的）。

Web 服务器的工作原理可以简单地归纳为：

（1）客户机通过 TCP/IP 建立到服务器的 TCP 连接。

（2）客户端向服务器发送 HTTP 请求包，请求服务器里的资源文档。

（3）服务器向客户机发送 HTTP 应答包，如果请求的资源包含动态语言内容，则服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理后的数据返回客户端。

（4）客户机与服务器断开。由客户端解释 HTML 文档，在客户端屏幕上渲染图形结果。

2. 网络编程中的 HTTP

HTTP 是互联网上应用最为广泛的一种网络协议，它详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议。

HTTP 通常承载于 TCP 之上，有时也承载于 TLS 或 SSL 协议层之上，这时就成了常说的 HTTPS，如图 5-16 所示。

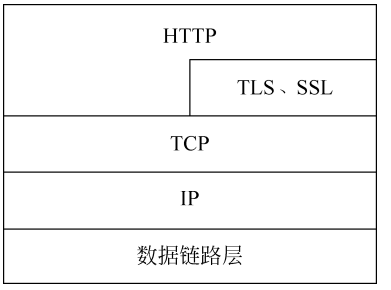


图 5-16　HTTP 层次结构

3. HTTP 请求报文格式

HTTP 请求报文由请求行、请求头部、空行、请求包体 4 部分组成，如图 5-17 所示。

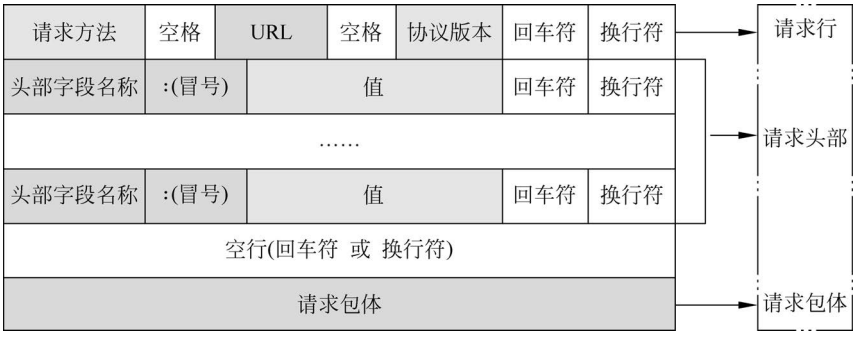


图 5-17　HTTP 请求报文格式

请求行：请求行由方法字段、URL 字段 和 HTTP 版本字段 3 部分组成，它们之间使用空格隔开。常用的 HTTP 请求方法有 GET、POST。

GET：当客户端要从服务器中读取某个资源时，使用 GET 方法。GET 方法要求服务器将 URL 定位的资源放在响应报文的数据部分，回送给客户端，即向服务器请求某个资

源。使用 GET 方法时，请求参数和对应的值附加在 URL 后面，利用一个问号(?)代表 URL 的结尾与请求参数的开始，传递参数的长度受限制，因此 GET 方法不适合用于上传数据。通过 GET 方法获取网页时，参数会显示在浏览器网址栏上，因此保密性很差。

POST：当客户端给服务器端提供信息较多时可以使用 POST 方法，POST 方法向服务器提交数据，例如完成表单数据的提交，将数据提交给服务器处理。

GET 一般用于获取/查询资源信息，POST 会附带用户数据，一般用于更新资源信息。POST 方法将请求参数封装在 HTTP 请求数据中，而且长度没有限制，因为 POST 携带的数据在 HTTP 的请求正文中，以名称/值的形式出现，可以传输大量数据。

请求头部：请求头部为请求报文添加了一些附加信息，由名称/值对组成，每行一对，名和值之间使用冒号分隔。请求头部通知服务器端有关于客户端请求的信息，典型的请求头如下。

User-Agent：请求的浏览器类型。

Accept：客户端可识别的响应内容类型列表，星号(*)用于按范围将类型分组，用 */* 指示可接受全部类型，用 type/* 指示可接受 type 类型的所有子类型。

Accept-Language：客户端可接受的自然语言。

Accept-Encoding：客户端可接受的编码压缩格式。

Accept-Charset：可接受的应答的字符集。

Host：请求的主机名，允许多个域名同处一个 IP 地址，即虚拟主机。

connection：连接方式(close 或 keepalive)。

Cookie：存储于客户端扩展字段，向同一域名的服务器端发送属于该域的 Cookie。

空行：最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

请求包体：请求包体不在 GET 方法中使用，而在 POST 方法中使用。POST 方法适用于需要客户填写表单的场合。与请求包体相关的最常使用的是包体类型 Content-Type 和包体长度 Content-Length。

4. HTTP 响应报文说明

HTTP 响应报文由状态行、响应头部、空行、响应包体 4 部分组成，如图 5-18 所示。

状态行：状态行由 HTTP 版本字段、状态码和状态码的描述文本 3 部分组成，它们之间使用空格隔开。

状态码：状态码由 3 位数字组成，第 1 位数字表示响应的类型，常用的状态码有 5 大类。

1xx：表示服务器已接收了客户端请求，客户端可继续发送请求。

2xx：表示服务器已成功接收到请求并进行处理。

3xx：表示服务器要求客户端重定向。

4xx：表示客户端的请求有非法内容。

5xx：表示服务器未能正常处理客户端的请求而出现意外错误。

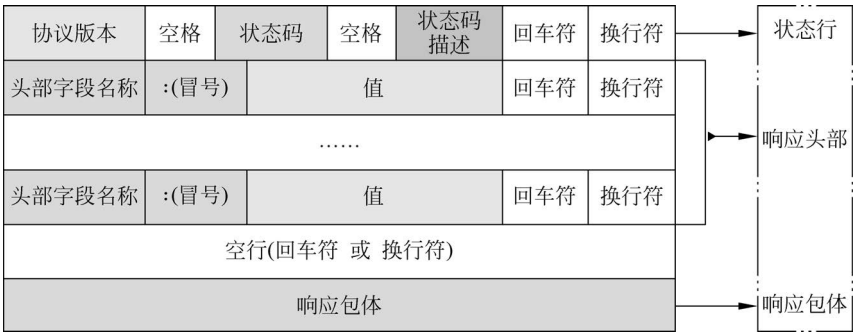


图 5-18　HTTP 响应报文格式

常见的状态码举例如下。

200：表示 OK，即客户端请求成功。

400：表示 Bad Request，即请求报文有语法错误。

401：表示 Unauthorized，即未授权。

403：表示 Forbidden，即服务器拒绝服务。

404：表示 Not Found，即请求的资源不存在。

500：表示 Internal Server Error，即服务器内部错误。

503：表示 Server Unavailable，即服务器临时不能处理客户端请求(稍后可能可以)。

响应头部，响应头可能包括以下几种信息。

Location：响应报头域用于将接受者重定向到一个新的位置。

Server：响应报头域包含了服务器用来处理请求的软件信息及其版本。

Vary：指示不可缓存的请求头列表。

Connection：连接方式。

空行：最后一个响应头部之后是一个空行，发送回车符和换行符，通知服务器以下不再有响应头部。

响应包体：服务器返回客户端的文本信息。

5. 编写简单的 Web 服务器

使用 Go 编写一个简单的 Web 服务器(生产环境建议初学者使用开源的 Web 框架，例如 Beego、Gin 等)，示例代码如下：

```
//anonymous - link\example\chapter5\web_server.go
package main

import (
    "fmt"
    "net/http"
)
```

```
func UserResp(resp http.ResponseWriter, req * http.Request) {
    fmt.Printf("请求方法：%s\n", req.Method)
    fmt.Printf("浏览器发送请求文件路径：%s\n", req.url)
    fmt.Printf("请求头：%s\n", req.Header)
    fmt.Printf("请求包体：%s\n", req.Body)
    fmt.Printf("客户端网络地址：%s\n", req.RemoteAddr)
    fmt.Printf("客户端 Agent：%s\n", req.UserAgent())

    /**
    给客户端回复数据
    */
    resp.Write([]Byte("User response"))
}

func IndexResp(resp http.ResponseWriter, req * http.Request) {
    resp.Write([]Byte("Index response"))
}

func main() {
    /**
    为不同的请求注册不同的函数
    */
    http.HandleFunc("/user", UserResp)
    http.HandleFunc("/index", IndexResp)

    //开启服务器,监听客户端的请求
    http.ListenAndServe("127.0.0.1:8080", nil)

}
```

6. 编写简单的客户端

使用 Go 发起 HTTP 请求,示例代码如下：

```
//anonymous - link\example\chapter5\web_client.go
package main

import (
    "fmt"
    "net/http"
)

func main() {

    /**
    该 URL 是刚刚编写的简单的 Web 服务器对应的资源
    */
    url := "http://127.0.0.1:8080/user"

    resp, err := http.Get(url)
    if err != nil {
```

```
        fmt.Println("获取数据失败,错误原因:", err)
        return
    }
    defer resp.Body.Close()

    /**
    获取从服务器端读到的数据
    */
    fmt.Printf("状态: %s\n", resp.Status)
    fmt.Printf("状态码: %v\n", resp.StatusCode)
    fmt.Printf("响应头部: %s\n", resp.Header)
    fmt.Println("响应包体:", resp.Body)

    /**
    定义切片缓冲区,临时存储读到的数据,并将每次读到的结果拼接到 data 中
    */
    buf := make([]Byte, 4096)
    var data string

    for {
        n, _ := resp.Body.Read(buf)
        if n == 0 {
            fmt.Println("数据读取完毕...")
            break
        }
        if err != nil {
            fmt.Println("数据读取失败,错误原因:", err)
            return
        }
        data += string(buf[:n])
    }

    fmt.Printf("从服务器端获取的内容是:[%s]\n", data)
}
```

5.1.18　Go 的序列化

1. 什么是序列化

数据在网络传输前后要进行序列化和反序列化。目的是将复杂的数据类型按照统一、简单且高效的形式转储,以达到网络传输的目的。除了在网络传输,有的数据存储到本地也是为了其他语言使用方便,通常也会使用相对较为通用的数据格式来存储,这就是常说的序列化,反序列化就是将数据按照规定的语法格式进行解析的过程。

2. 什么是 JSON

JSON 采用完全独立于语言的文本格式,但是也具有类似于 C 语言家族的特性(包括 C、C++、C#、Java、JavaScript、Perl、Python、Go 等)。这些特性使 JSON 成为理想的数据交换格式。易于人阅读和编写,同时也易于机器解析和生成(一般用于提升网络传输速率)。

目前，JSON 已经成为主流的数据格式。

JSON 的特性：①JSON 解析器和 JSON 库支持许多不同的编程语言；②JSON 文本格式在语法上与创建 JavaScript 对象的代码相同。由于这种相似性，无需解析器，JavaScript 程序能够使用内建的 eval()函数，用 JSON 数据来生成原生的 JavaScript 对象；③JSON 是存储和交换文本信息的语法，比 XML 更小、更快，更易解析；④JSON 具有自我描述性，语法简洁，易于理解；⑤JSON 数据主要有两种数据结构，一种是键/值，另一种是以数组的形式来表示。

3. JSON 序列化案例

(1) 结构体序列化通过 MarshalIndent 方法实现，示例代码如下：

```
//anonymous - link\example\chapter5\struct_json1.go
package main

import (
    "encoding/json"
    "fmt"
)
/**
定义需要的结构体
*/
type Teacher struct {
    Name string
    ID int
    Age int
    Address string
}

func main() {
    s1 := Teacher{
        Name: "Frank",
        ID: 1001,
        Age: 18,
        Address: "北京",
    }

    /**
    使用 encoding/json 包的 Marshal 函数进行序列化操作，其函数签名如下：
        func Marshal(v interface{}) ([]Byte, error)
    以下是对 Marshal 函数参数的相关说明
        v: 该参数是空接口类型。意味着任何数据类型(int、float、map 结构体等)都可以使用该函数进行序列化
        返回值：很明显返回值是字节切片和错误信息
    */
    //data, err := json.Marshal(&s1)

    /**
    Go 语言标准库的 encoding/json 包还提供了另外一种方法：MarshalIndent
```

```
    该方法的作用与 Marshal 的作用相同,只是可以通过方法参数设置前缀、缩进等,对 JSON 多了一
些格式处理,打印出来比较好看
    */
    data, err := json.MarshalIndent(s1, "\t", "")
    if err != nil {
        fmt.Println("序列化出错,错误原因:", err)
        return
    }

    /**
    查看序列化后的 JSON 字符串
    */
    fmt.Println("序列化之后的数据为:", string(data))
}
```

结构体序列化通过 Marshal 方法实现,示例代码如下:

```
//anonymous - link\example\chapter5\struct_json2.go
package main

import (
    "encoding/json"
    "fmt"
)
/**
定义需要的结构体
*/
type Teacher struct {
    Name string
    ID int
    Age int
    Address string
}

func main() {
    s1 := Teacher{
        Name: "Frank",
        ID: 1002,
        Age: 18,
        Address: "北京",
    }

    /**
    使用 encoding/json 包的 Marshal 函数进行序列化操作,其函数签名如下:
        func Marshal(v interface{}) ([]Byte, error)
    以下是对 Marshal 函数参数的相关说明
        v: 该参数是空接口类型。意味着任何数据类型(int、float、map 结构体等)都可以使用该函
数进行序列化
        返回值:很明显返回值是字节切片和错误信息
    */
```

```
    data, err := json.Marshal(&s1)     //注意,这里传递的是引用地址
    if err != nil {
        fmt.Println("序列化出错,错误原因:", err)
        return
    }

    /**
    查看序列化后的 JSON 字符串
    */
    fmt.Println("序列化之后的数据为:", string(data))
}
```

(2) map 序列化,示例代码如下:

```
//anonymous - link\example\chapter5\map_json.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {

    var s1 map[string]interface{}

    /**
    使用 make 函数初始化 map 以开辟内存空间
    */
    s1 = make(map[string]interface{})

    /**
    map 赋值操作
    */
    s1["name"] = "Jason Yin"
    s1["age"] = 20
    s1["address"] = [2]string{"北京", "陕西"}

    /**
    将 map 使用 Marshal()函数进行序列化
    */
    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("Marshal err:", err)
        return
    }

    /**
    查看序列化后的 JSON 字符串
    */
```

```
    fmt.Println("序列化之后的数据为：", string(data))

}
```

（3）切片序列化，示例代码如下：

```
//anonymous - link\example\chapter5\slice_json.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    /**
    创建一个类似于 map[string]interface{}的切片
    */
    var s1 []map[string]interface{}

    /**
    使用 make 函数初始化 map 以开辟内存空间
    */
    m1 := make(map[string]interface{})

    /**
    为 map 进行赋值操作
    */
    m1["name"] = "李白"
    m1["role"] = "打野"

    m2 := make(map[string]interface{})
    m2["name"] = "王昭君"
    m2["role"] = "中单"

    m3 := make(map[string]interface{})
    m3["name"] = "程咬金"
    m3["role"] = "上单"

    /**
    将 map 追加到切片中
    */
    s1 = append(s1, m3, m2, m1)

    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("序列化出错，错误原因：", err)
        return
    }
```

```
    /**
    查看序列化后的数据
    */
    fmt.Println(string(data))
}
```

(4) 数组序列化,示例代码如下:

```
//anonymous-link\example\chapter5\array_json.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    /**
    定义数组
    */
    var s1 = [5]int{9, 5, 2, 7, 5200}
    /**
    将数组使用 Marshal 函数进行序列化
    */
    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("序列化错误:", err)
        return
    }
    /**
    查看序列化后的 JSON 字符串
    */
    fmt.Println("数组序列化后的数据为:", string(data))
}
```

(5) 基础数据类型序列化,示例代码如下:

```
//anonymous-link\example\chapter5\basic_json.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    /**
    定义基础数据类型数据
    */
    var (
```

```
        Surname = '周'
        Name = "周杰伦"
        Age = 18
        Temperature = 35.6
        HubeiProvince = false
    )

    /**
    对基础数据类型进行序列化操作
    */
    surname, _ := json.Marshal(Surname)
    name, _ := json.Marshal(Name)
    age, _ := json.Marshal(Age)
    temperature, _ := json.Marshal(Temperature)
    hubeiProvince, _ := json.Marshal(HubeiProvince)

    /**
    查看序列化后的JSON字符串
    */
    fmt.Println("Surname序列化后的数据为：", string(surname))
    fmt.Println("Name序列化后的数据为：", string(name))
    fmt.Println("Age序列化后的数据为：", string(age))
    fmt.Println("Temperature序列化后的数据为：", string(temperature))
    fmt.Println("HubeiProvince序列化后的数据为：", string(hubeiProvince))
}
```

4. JSON反序列化案例

(1) 结构体反序列化,示例代码如下：

```
//anonymous - link\example\chapter5\json_struct.go
package main

import (
    "encoding/json"
    "fmt"
)

type People struct {
    Name string
    Age int
    Address string
}

func main() {
    /**
    以JSON数据为例,接下来要对该数据进行反序列化操作
    */
    p1 := `{"Name":"Jason Yin","Age":18,"Address":"北京"}`

    var s1 People
```

```
    fmt.Printf("反序列化之前：\n\ts1 =  %v \n\ts1.Name =  %s\n\n", s1, s1.Name)

    /**
    使用 encoding/json 包中的 Unmarshal()函数进行反序列化操作,其函数签名如下：
        func Unmarshal(data []Byte, v interface{}) error
    以下是对函数签名的参数说明
        data：待解析的 JSON 编码字符串
        v：解析后传出的结果,即用来容纳待解析的 JSON 数据容器
    */
    err := json.Unmarshal([]Byte(p1), &s1)
    if err != nil {
        fmt.Println("反序列化失败：", err)
        return
    }

    /**
    查看反序列化后的结果
    */
    fmt.Printf("反序列化之后：\n\ts1 =  %v \n\ts1.Name =  %s\n", s1, s1.Name)

}
```

(2) map 反序列化,示例代码如下：

```
//anonymous - link\example\chapter5\json_map.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {

    m1 := `{"address":["北京","陕西"],"age":20,"name":"Jason Yin"}`

    /**
    定义 map 变量,类型必须与之前序列化的类型完全一致
    */
    var s1 map[string]interface{}
    fmt.Println("反序列化之前:s1 =", s1)

    /**
    注意事项：
        不需要使用 make 函数初始化 m1,开辟空间。这是因为在反序列化函数 Unmarshal()中会判断传入的参数 2,如果是 map 类型数据,则会自动开辟空间。相当于 Unmarshal()函数可以帮助做 make 操作
        但传参时需要注意,Unmarshal 的第 2 个参数被用作传出,并返回结果,因此必须传 m1 的地址值
    */
```

```
    err := json.Unmarshal([]Byte(m1), &s1)
    if err != nil {
        fmt.Println("反序列化失败,错误原因:", err)
        return
    }

    fmt.Println("反序列化之后:s1 =", s1)
}
```

(3) 切片反序列化,示例代码如下:

```
//anonymous - link\example\chapter5\json_slice.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    s1 := `[{"name":"王昭君","role":"中单"},{"name":"李白","role":"打野"}]`

    var slice []map[string]interface{}
    fmt.Println("反序列化之前:slice =", slice)

    /**
    实现思路与前面两种完全一致,这里不再赘述
    注意事项:
        反序列化 JSON 字符串时,确保反序列化传出的数据类型与之前序列化的数据类型完全一致
    */
    err := json.Unmarshal([]Byte(s1), &slice)
    if err != nil {
        fmt.Println("反序列化失败,错误原因:", err)
        return
    }

    fmt.Println("反序列化之后:slice =", slice)
}
```

5. 结构体标签序列化

结构体标签序列化,示例代码如下:

```
//anonymous - link\example\chapter5\tag_json.go
package main

import (
    "encoding/json"
    "fmt"
)
/**
```

```
结构体的字段除了名字和类型外,还可以有一个可选的标签,它是一个附属于字段的字符串,可以是文档,也可以是其他的重要标记
例如在解析 JSON 或生成 JSON 文件时,常用到 encoding/json 包,它提供了一些默认标签
定义结构体时,可以通过这些默认标签来设定结构体成员变量,使之在序列化后得到特殊的输出
*/
type Student struct {
    /**
    "-"标签的作用是不进行序列化,效果和将结构体字段首字母写成小写一样
    */
    Name string `json:"-"`

    /**
    string 标签:在以这种方式生成的 JSON 对象中,ID 的类型转换为字符串
    */
    ID int `json:"id,string"`

    /**
    omitempty 标签:可以在序列化时忽略 0 值或者空值
    */
    Age int `json:"AGE,omitempty"`

    /**
    可以对字段名称进行重命名操作:例如下面的案例就是将"Address"字段重命名为"HomeAddress"
    */
    Address string `json:"HomeAddress"`

    /**
    由于该字段首字母是小写,因此该字段不参与序列化
    */
    score int
    Hobby string
}

func main() {
    s1 := Student{
        Name: "Jason Yin",
        ID: 001,
        //Age: 18,
        Address: "北京",
        score: 100,
        Hobby: "中国象棋",
    }

    data, err := json.Marshal(s1)
    if err != nil {
        fmt.Println("序列化出错,错误原因:", err)
        return
    }
    fmt.Println("序列化结果:", string(data))
}
```

6. gob 序列化和反序列化

和 Python 的 pickle 模块类似，Go 语言自带的序列化方式是 gob，一些 Go 语言自带的包使用的序列化方式也是 gob。接下来查看一下 gob 的使用方式。

（1）gob 序列化，示例代码如下：

```
//anonymous - link\example\chapter5\gob_serialize.go
package main

import (
    "Bytes"
    "encoding/gob"
    "fmt"
)

type People struct {
    Name string
    Age int
}

func main() {

    p := People{
        Name: "Jason Yin",
        Age: 18,
    }

    /**
    定义一字节容器，其结构体如下：
        type Buffer struct {
            buf []Byte          //contents are the Bytes buf[off : len(buf)]
            off int             //read at &buf[off], write at &buf[len(buf)]
            lastRead readOp     //last read operation, so that Unread* can work correctly
        }
    */
    buf := Bytes.Buffer{}

    /**
    初始化编码器，其函数签名如下：
        func NewEncoder(w io.Writer) *Encoder
    以下是函数签名的参数说明
        w:一个 io.Writer 对象，可以传递"Bytes.Buffer{}"的引用地址
        返回值:NewEncoder 返回将在 io.Writer 上传输的新编码器
    */
    encoder := gob.NewEncoder(&buf)

    /**
    编码操作
    */
    err := encoder.Encode(p)
    if err != nil {
```

```
        fmt.Println("编码失败,错误原因:", err)
        return
    }

    /**
    查看编码后的数据(gob 序列化其实是二进制数据)
     */
    fmt.Println(string(buf.Bytes()))
}
```

(2) gob 反序列化,示例代码如下:

```
//anonymous - link\example\chapter5\gob_deserialize.go
package main

import (
    "Bytes"
    "encoding/gob"
    "fmt"
)

type Student struct {
    Name string
    Age int
}

func main() {

    s1 := Student{
        Name: "Jason Yin",
        Age: 18,
    }

    buf := Bytes.Buffer{}

    /**
    初始化编码器
     */
    encoder := gob.NewEncoder(&buf)

    /**
    编码操作,相当于序列化过程
     */
    err := encoder.Encode(s1)
    if err != nil {
        fmt.Println("编码失败,错误原因:", err)
        return
    }

    /**
    查看编码后的数据(gob 序列化其实是二进制数据)
```

```
        */
        //fmt.Println(string(buf.Bytes()))

        /**
        初始化解码器,其函数签名如下:
            func NewDecoder(r io.Reader) *Decoder
        以下是函数签名的参数说明
            r:一个 io.Reader 对象,其函数签名如下
                    func NewReader(b []Byte) *Reader
                综上所述,可以将编码后的字节数组传递给该解码器
            返回值:new decoder 返回从 io.Reader 读取的新解码器
                如果 r 不同时实现 io.ByteReader,则将被包装在 bufio.Reader 中
        */
        decoder := gob.NewDecoder(Bytes.NewReader(buf.Bytes()))

        var s2 Student
        fmt.Println("解码之前 s2 = ", s2)

        /**
        进行解码操作,相当于反序列化过程
        */
        decoder.Decode(&s2)
        fmt.Println("解码之前 s2 = ", s2)
    }
```

5.1.19　Go 的序列化：ProtoBuf

1. ProtoBuf 概述

ProtoBuf 是 Protocol Buffers 的简称,它是谷歌公司用 C 语言(因此很多语法借鉴了 C 语法的特性)开发的一种数据描述语言,是一种轻便高效的结构化数据存储格式,可以用于结构化数据串行化,或者序列化。

它很适合做数据存储或 RPC 数据交换格式。

可用于通信协议、数据存储等与语言无关、与平台无关、可扩展的序列化结构数据格式。目前提供了 C++、Java、Python 这 3 种语言的 API,其他语言需要安装相关插件才能使用。

ProtoBuf 刚开源时的定位类似于 XML、JSON 等数据描述语言,通过附带工具生成代码并实现将结构化数据序列化的功能。

这里更关注的是 ProtoBuf 作为接口规范的描述语言,可以作为设计安全的跨语言 RPC 接口的基础工具,主要有以下特性:

(1) ProtoBuf 是类似于 JSON 的数据描述语言(数据格式)。

(2) ProtoBuf 非常适合 RPC 数据交换格式。

ProtoBuf 的优势和劣势。

优势:

(1) 序列化后体积比 JSON 和 XML 小,适合网络传输。

(2) 支持跨平台多语言。

(3) 消息格式升级和兼容性好。

(4) 序列化和反序列化的速度很快,快于 JSON 的处理速度。

劣势:

(1) 相比 XML 和 JSON,应用不够广。

(2) 二进制格式导致可读性差。

(3) 缺乏自描述。

更详细的特性和使用推荐阅读官方文档。

2. ProtoBuf 安装

(1) 下载 ProtoBuf 软件包,命令如下:

```
https://github.com/protocolbuffers/protobuf/releases
```

(2) 配置环境变量,即把下载后的文件解压后得到的目录路径追加到系统变量中,可通过如下命令进行验证:

```
protoc --version
```

(3) 安装 Go 的编译插件,执行以下命令安装插件:

```
go get -u github.com/Go/protobuf/protoc-gen-go
```

安装成功后会在%GOPATH%\bin 目录生成一个编译工具 protoc-gen-go.exe。

3. ProtoBuf 的简单语法

(1) 参考文档资料 https://developers.google.com/protocol-buffers/docs/reference/go-generated。

(2) 编写简单的 ProtoBuf 案例,需要注意的是,文件名后缀以.proto 结尾,示例代码如下:

```
//anonymous-link\example\chapter5\protobuf_example.proto

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号。如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";
//指定包名,package 关键字指明当前 mypb 包生成 Go 文件之后和 Go 的包名保持一致,但是如果定义
//了"option go_package"参数,则 package 的参数自动失效
package mypb;
//.proto 文件应包含一个 go_package 选项,用于指定包含所生成代码的 Go 软件包的完整导入路径
//(最后一次"bar"就是生成 Go 文件的包名),官方在未来的发行版本会支持
option go_package = "example.com/foo/bar";
/*
通过 message 关键定义传输数据的格式,类似于 Go 语言中的结构体,是包含一系列类型数据的集合
许多标准的简单数据类型可以作为字段类型,包括 bool、int32、float、double 和 string。也可以使
用其他 message 类型作为字段类型
```

```
 */
message People{
    /*
        这里的"1"表示字段是1,类似于数据库中表的主键 id 等于1,主键不能重复,标识位数据不能重复。该成员编码时用1代替名字
        在JSON中是通过成员的名字来绑定对应的数据,但是 ProtoBuf 编码通过成员的唯一编号来绑定对应的数据
        综上所述,ProtoBuf 编码后数据的体积会比较小,能够快速传输,但缺点是不利于阅读
     */
    string name = 1;

    //需要注意的是标识位不能使用 19 000～19 999(系统预留位)
    int32 age = 2;

    //结构体嵌套,例如嵌套一个 Student 结构体
    Student s = 3;

    //使用数组
    repeated string phone = 4;
}
/*
   message 的格式说明如下:
      消息由至少一个字段组合而成,类似于 Go 语言中的结构体,每个字段都有一定的格式
    (字段修饰符)数据类型 字段名称 = 唯一的编号标签值;
    唯一的编号标签:
      代表每个字段的一个唯一的编号标签,在同一条消息里不可以重复。这些编号标签用于在消息二进制格式中标识的字段,并且消息一旦定义就不能更改。需要说明的是,标签在1～15采用一字节进行编码,所以通常将标签1～15用于频繁发生的消息字段
      编号标签大小的范围是 1～2^29。19 000～19 999 是官方预留的值,不能使用
    注释格式:向.proto 文件添加注释,可以使用 C/C++/Java/Go 风格的双斜杠或者段落注释语法格式
    message 常见的数据类型与 Go 中类型对比:
        .proto 类型    Go 类型    介绍
        double         float64    64 位浮点数
        float          float32    32 位浮点数
        int32          int32      使用可变长度编码。编码负数效率低下,如果字段可能有负值,则应改用 sint32
        int64          int64      使用可变长度编码。编码负数效率低下,如果字段可能有负值,则应改用 sint64
        uint32         uint32     使用可变长度编码
        uint64         uint64     使用可变长度编码
        sint32         int32      使用可变长度编码。符号整型值。这些比常规 int32s 编码负数更有效
        sint64         int64      使用可变长度编码。符号整型值。这些比常规 int64s 编码负数更有效
        fixed32        uint32     总是 4 字节。如果值通常大于 228,则比 uint 32 更有效
        fixed64        uint64     总是 8 字节。如果值通常大于 256,则比 uint64 更有效
        sfixed32       int32      总是 4 字节
        sfixed64       int64      总是 8 字节
        bool           bool       布尔类型
        string         string     字符串必须始终包含 UTF－8 编码或 7 位 ASCII 文本
        Bytes          []Byte     可以包含任意字节序列
```

```
 * /
message Student{
    string name = 1;
    int32 age = 5;
}
```

(3) 基于 ProtoBuf 文件进行编译生成对应的 Go 文件,命令如下:

```
protoc --go_out=. demo.proto
```

生成文件的内容如下:

```
//anonymous-link\example\chapter5\protobuf_example.go

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
//Code generated by protoc-gen-go. DO NOT EDIT.
//versions:
//protoc-gen-go v1.21.0
//protoc          v3.11.4
//source: demo.proto
//指定包名,package 关键字指明当前 mypb 包生成 Go 文件之后和 Go 的包名保持一致,但是如果定义
//了"option go_package"参数,则 package 的参数失效

package bar

import (
    proto "github.com/Go/protobuf/proto"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)
const (
    //Verify that this generated code is sufficiently up-to-date.
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up-to-date.
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)
//This is a compile-time assertion that a sufficiently up-to-date version
//of the legacy proto package is being used.const _ = proto.ProtoPackageIsVersion4
//通过 message 关键字定义传输数据的格式,类似于 Go 语言中的结构体,是包含一系列类型数据的集合
//许多标准的简单数据类型可以作为字段类型,包括 bool、int32、float、double 和 string,也可以
//使用其他 message 类型作为字段类型
type People struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //注意,这里的"1"表示字段是 1,类似于数据库中表的主键 id 等于 1,主键不能重复,标识位数
```

```
    //据不能重复。该成员编码时用 1 代替名字
    //在 JSON 中通过成员的名字来绑定对应的数据,但是 ProtoBuf 编码却通过成员的唯一编号来
    //绑定对应的数据
    //综上所述,ProtoBuf 编码后数据的体积会比较小,能够快速传输,但缺点是不利于阅读
    Name string `protobuf:"Bytes,1,opt,name = name,proto3" json:"name,omitempty"`
    //需要注意的是标识位不能使用 19 000～19 999(系统预留位)
    Age int32 `protobuf:"varint,2,opt,name = age,proto3" json:"age,omitempty"`
    //结构体嵌套,例如嵌套一个 Student 结构体
    S *Student `protobuf:"Bytes,3,opt,name = s,proto3" json:"s,omitempty"`
    //使用数组
    Phone []string `protobuf:"Bytes,4,rep,name = phone,proto3" json:"phone,omitempty"`
}

func (x *People) Reset() {
    *x = People{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *People) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*People) ProtoMessage() {}

func (x *People) ProtoReflect() protoreflect.Message {
    mi := &file_demo_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
//Deprecated: Use People.ProtoReflect.Descriptor instead
func (*People) Descriptor() ([]byte, []int) {
    return file_demo_proto_rawDescGZIP(), []int{0}
}

func (x *People) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}
```

```
func (x *People) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *People) GetS() *Student {
    if x != nil {
        return x.S
    }
    return nil
}

func (x *People) GetPhone() []string {
    if x != nil {
        return x.Phone
    }
    return nil
}
//message 的格式说明如下
//消息由至少一个字段组合而成,类似于 Go 语言中的结构体,每个字段都有一定的格式
//(字段修饰符)数据类型 字段名称 = 唯一的编号标签值
//唯一的编号标签
//代表每个字段的一个唯一的编号标签,在同一条消息里不可以重复。这些编号标签用于在消息二
//进制格式中标识的字段,并且消息一旦定义就不能更改。需要说明的是标签在 1～15 采用一字节
//进行编码,所以通常将标签 1～15 用于频繁发生的消息字段。
//编号标签大小的范围是 1～2^29。19 000～19 999 是官方预留的值,不能使用
//注释格式
//向.proto 文件添加注释,可以使用 C/C++/Java/Go 风格的双斜杠或者段落注释语法格式
//message 常见的数据类型与 Go 中类型的对比
//.proto 类型    Go 类型    介绍
//double         float64    64 位浮点数
//float          float32    32 位浮点数
//int32          int32      使用可变长度编码。编码负数效率低下,如果字段可能有负值,则应改用
                            sint32
//int64          int64      使用可变长度编码。编码负数效率低下,如果字段可能有负值,则应改用
                            sint64
//uint32         uint32     使用可变长度编码
//uint64         uint64     使用可变长度编码
//sint32         int32      使用可变长度编码。符号整型值。这些比常规 int32s 编码负数更有效
//sint64         int64      使用可变长度编码。符号整型值。这些比常规 int64s 编码负数更有效
//fixed32        uint32     总是 4 字节。如果值通常大于 228,则比 uint 32 更有效
//fixed64        uint64     总是 8 字节。如果值通常大于 256,则比 uint64 更有效
//sfixed32       int32      总是 4 字节
//sfixed64       int64      总是 8 字节
//bool           bool       布尔类型
//string         string     字符串必须始终包含 UTF-8 编码或 7 位 ASCII 文本
//Bytes          []Byte     可以包含任意字节序列

type Student struct {
```

```
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Name string `protobuf:"Bytes,1,opt,name=name,proto3" json:"name,omitempty"`
    Age int32 `protobuf:"varint,5,opt,name=age,proto3" json:"age,omitempty"`
}

func (x *Student) Reset() {
    *x = Student{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Student) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Student) ProtoMessage() {}

func (x *Student) ProtoReflect() protoreflect.Message {
    mi := &file_demo_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
//Deprecated: Use Student.ProtoReflect.Descriptor instead
func (*Student) Descriptor() ([]byte, []int) {
    return file_demo_proto_rawDescGZIP(), []int{1}
}

func (x *Student) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *Student) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
```

```
}
var File_demo_proto protoreflect.FileDescriptor
var file_demo_proto_rawDesc = []Byte{
    0x0a, 0x0a, 0x64, 0x65, 0x6d, 0x6f, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x12, 0x04, 0x6d,
0x79,
    0x70, 0x62, 0x22, 0x61, 0x0a, 0x06, 0x50, 0x65, 0x6f, 0x70, 0x6c, 0x65, 0x12, 0x12, 0x0a,
0x04,
    0x6e, 0x61, 0x6d, 0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d,
0x65,
    0x12, 0x10, 0x0a, 0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03,
0x61,
    0x67, 0x65, 0x12, 0x1b, 0x0a, 0x01, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28, 0x0b, 0x32, 0x0d,
0x2e,
    0x6d, 0x79, 0x70, 0x62, 0x2e, 0x53, 0x74, 0x75, 0x64, 0x65, 0x6e, 0x74, 0x52, 0x01, 0x73,
0x12,
    0x14, 0x0a, 0x05, 0x70, 0x68, 0x6f, 0x6e, 0x65, 0x18, 0x04, 0x20, 0x03, 0x28, 0x09, 0x52,
0x05,
    0x70, 0x68, 0x6f, 0x6e, 0x65, 0x22, 0x2f, 0x0a, 0x07, 0x53, 0x74, 0x75, 0x64, 0x65, 0x6e,
0x74,
    0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52,
0x04,
    0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10, 0x0a, 0x03, 0x61, 0x67, 0x65, 0x18, 0x05, 0x20, 0x01,
0x28,
    0x05, 0x52, 0x03, 0x61, 0x67, 0x65, 0x42, 0x15, 0x5a, 0x13, 0x65, 0x78, 0x61, 0x6d, 0x70,
0x6c,
    0x65, 0x2e, 0x63, 0x6f, 0x6d, 0x2f, 0x66, 0x6f, 0x6f, 0x2f, 0x62, 0x61, 0x72, 0x62, 0x06,
0x70,
    0x72, 0x6f, 0x74, 0x6f, 0x33,
}
var (
    file_demo_proto_rawDescOnce sync.Once
    file_demo_proto_rawDescData = file_demo_proto_rawDesc
)

func file_demo_proto_rawDescGZIP() []Byte {
    file_demo_proto_rawDescOnce.Do(func() {
        file_demo_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo_proto_
rawDescData)
    })
    return file_demo_proto_rawDescData
}
var file_demo_proto_msgTypes = make([]protoimpl.MessageInfo, 2)var file_demo_proto_goTypes = []
interface{}{
    (*People)(nil), //0: mypb.People
    (*Student)(nil), //1: mypb.Student}var file_demo_proto_depIdxs = []int32{
    1, //0: mypb.People.s:type_name -> mypb.Student
    1, //[1:1] is the sub-list for method output_type
    1, //[1:1] is the sub-list for method input_type
    1, //[1:1] is the sub-list for extension type_name
    1, //[1:1] is the sub-list for extension extendee
    0, //[0:1] is the sub-list for field type_name}
```

```
func init() { file_demo_proto_init() }
func file_demo_proto_init() {
    if File_demo_proto != nil {
        return
    }
    if !protoimpl.UnsafeEnabled {
        file_demo_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*People); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_demo_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Student); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo_proto_rawDesc,
            NumEnums:    0,
            NumMessages: 2,
            NumExtensions: 0,
            NumServices: 0,
        },
        GoTypes:           file_demo_proto_goTypes,
        DependencyIndexes: file_demo_proto_depIdxs,
        MessageInfos:      file_demo_proto_msgTypes,
    }.Build()
    File_demo_proto = out.File
    file_demo_proto_rawDesc = nil
    file_demo_proto_goTypes = nil
    file_demo_proto_depIdxs = nil
}
```

4. ProtoBuf 的高级用法

(1) message 嵌套,示例代码如下:

```
//anonymous - link\example\chapter5\protobuf_message.proto

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";
//.proto 文件应包含一个 go_package 选项,用于指定包含所生成代码的 Go 软件包的完整导入路径
//(最后一次"bar"就是生成 go 文件的包名),官方在未来的发行版本会支持
option go_package = "example.com/foo/bar";

message Teacher{
    //姓名
    string name = 1;

    //年龄
    int32 age = 2 ;

    //地址
    string address = 3;

    //定义一个 message
    message PhoneNumber{
        string number = 1;
        int64 type = 2;
    }

    //使用定义的 message
    PhoneNumber phone = 4 ;
}
```

使用命令 protoc --go_out=. demo2.proto 生成对应的 Go 代码,代码如下:

```
//anonymous - link\example\chapter5\protobuf_message.go

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行

//Code generated by protoc - gen - go. DO NOT EDIT.
//versions:
//protoc - gen - go v1.21.0
//protoc          v3.11.4
//source: demo2.proto

package bar

import (
    proto "github.com/Go/protobuf/proto"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
```

```
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)

const (
    //Verify that this generated code is sufficiently up - to - date.
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up - to - date.
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)

//This is a compile - time assertion that a sufficiently up - to - date version
//of the legacy proto package is being used
const _ = proto.ProtoPackageIsVersion4

type Teacher struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //姓名
    Name string `protobuf:"Bytes,1,opt,name = name,proto3" json:"name,omitempty"`
    //年龄
    Age int32 `protobuf:"varint,2,opt,name = age,proto3" json:"age,omitempty"`
    //地址
    Address string `protobuf:"Bytes,3,opt,name = address,proto3" json:"address,omitempty"`
    //使用定义的 message
    Phone *Teacher_PhoneNumber `protobuf:"Bytes,4,opt,name = phone,proto3" json:"phone,
omitempty"`
}

func (x *Teacher) Reset() {
    *x = Teacher{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo2_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher) ProtoMessage() {}

func (x *Teacher) ProtoReflect() protoreflect.Message {
    mi := &file_demo2_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
```

```
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}

//Deprecated: Use Teacher.ProtoReflect.Descriptor instead
func (*Teacher) Descriptor() ([]Byte, []int) {
    return file_demo2_proto_rawDescGZIP(), []int{0}
}

func (x *Teacher) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *Teacher) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *Teacher) GetAddress() string {
    if x != nil {
        return x.Address
    }
    return ""
}

func (x *Teacher) GetPhone() *Teacher_PhoneNumber {
    if x != nil {
        return x.Phone
    }
    return nil
}

//定义一个 message
type Teacher_PhoneNumber struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Number string `protobuf:"Bytes,1,opt,name=number,proto3" json:"number,omitempty"`
    Type int64 `protobuf:"varint,2,opt,name=type,proto3" json:"type,omitempty"`
}
```

```
func (x *Teacher_PhoneNumber) Reset() {
    *x = Teacher_PhoneNumber{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo2_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher_PhoneNumber) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher_PhoneNumber) ProtoMessage() {}

func (x *Teacher_PhoneNumber) ProtoReflect() protoreflect.Message {
    mi := &file_demo2_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}

//Deprecated: Use Teacher_PhoneNumber.ProtoReflect.Descriptor instead
func (*Teacher_PhoneNumber) Descriptor() ([]byte, []int) {
    return file_demo2_proto_rawDescGZIP(), []int{0, 0}
}

func (x *Teacher_PhoneNumber) GetNumber() string {
    if x != nil {
        return x.Number
    }
    return ""
}

func (x *Teacher_PhoneNumber) GetType() int64 {
    if x != nil {
        return x.Type
    }
    return 0
}

var File_demo2_proto protoreflect.FileDescriptor

var file_demo2_proto_rawDesc = []byte{
    0x0a, 0x0b, 0x64, 0x65, 0x6d, 0x6f, 0x32, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x22, 0xb0,
0x01,
```

```
    0x0a, 0x07, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61,
0x6d,
    0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10,
0x0a,
    0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65,
0x12,
    0x18, 0x0a, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28,
0x09,
    0x52, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x12, 0x2a, 0x0a, 0x05, 0x70, 0x68,
0x6f,
    0x6e, 0x65, 0x18, 0x04, 0x20, 0x01, 0x28, 0x0b, 0x32, 0x14, 0x2e, 0x54, 0x65, 0x61, 0x63,
0x68,
    0x65, 0x72, 0x2e, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x52,
0x05,
    0x70, 0x68, 0x6f, 0x6e, 0x65, 0x1a, 0x39, 0x0a, 0x0b, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e,
0x75,
    0x6d, 0x62, 0x65, 0x72, 0x12, 0x16, 0x0a, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x18,
0x01,
    0x20, 0x01, 0x28, 0x09, 0x52, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x12, 0x12, 0x0a,
0x04,
    0x74, 0x79, 0x70, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x03, 0x52, 0x04, 0x74, 0x79, 0x70,
0x65,
    0x42, 0x15, 0x5a, 0x13, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x63, 0x6f, 0x6d,
0x2f,
    0x66, 0x6f, 0x6f, 0x2f, 0x62, 0x61, 0x72, 0x62, 0x06, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x33,
}

var (
    file_demo2_proto_rawDescOnce sync.Once
    file_demo2_proto_rawDescData = file_demo2_proto_rawDesc
)

func file_demo2_proto_rawDescGZIP() []Byte {
    file_demo2_proto_rawDescOnce.Do(func() {
         file_demo2_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo2_proto_
rawDescData)
    })
    return file_demo2_proto_rawDescData
}

var file_demo2_proto_msgTypes = make([]protoimpl.MessageInfo, 2)
var file_demo2_proto_goTypes = []interface{}{
    (*Teacher)(nil),                  //0: Teacher
    (*Teacher_PhoneNumber)(nil),      //1: Teacher.PhoneNumber
}
var file_demo2_proto_depIdxs = []int32{
    1, //0: Teacher.phone:type_name -> Teacher.PhoneNumber
    1, //[1:1] is the sub-list for method output_type
    1, //[1:1] is the sub-list for method input_type
    1, //[1:1] is the sub-list for extension type_name
    1, //[1:1] is the sub-list for extension extendee
    0, //[0:1] is the sub-list for field type_name
```

```
}

func init() { file_demo2_proto_init() }
func file_demo2_proto_init() {
    if File_demo2_proto != nil {
        return
    }
    if !protoimpl.UnsafeEnabled {
        file_demo2_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_demo2_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher_PhoneNumber); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo2_proto_rawDesc,
            NumEnums: 0,
            NumMessages: 2,
            NumExtensions: 0,
            NumServices: 0,
        },
        GoTypes:           file_demo2_proto_goTypes,
        DependencyIndexes: file_demo2_proto_depIdxs,
        MessageInfos:      file_demo2_proto_msgTypes,
    }.Build()
    File_demo2_proto = out.File
    file_demo2_proto_rawDesc = nil
    file_demo2_proto_goTypes = nil
    file_demo2_proto_depIdxs = nil
}
```

（2）repeated 关键字，示例代码如下：

```
//anonymous-link\example\chapter5\protobuf_repeated.proto

//ProtoBuf 默认支持的版本是 2.x，现在一般使用 3.x 版本，所以需要手动指定版本号，如果不这样
//做，则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";
//.proto 文件应包含一个 go_package 选项，用于指定包含所生成代码的 Go 软件包的完整导入路径
//（最后一次"bar"就是生成 go 文件的包名），官方在未来的发行版本会支持
option go_package ="example.com/foo/bar";

message Teacher{
    //姓名
    string name = 1;

    //年龄
    int32 age = 2 ;

    //地址
    string address = 3;

    //定义一个 message
    message PhoneNumber{
        string number = 1;
        int64 type = 2;
    }

    //repeated 关键字类似于 Go 中的切片，编译之后对应的也是 Go 的切片
    repeated PhoneNumber phone = 4 ;
}
```

使用命令 protoc --go_out=. demo3.proto 生成对应的 Go 代码，代码如下：

```
//anonymous-link\example\chapter5\protobuf_repeated.go
//ProtoBuf 默认支持的版本是 2.x，现在一般使用 3.x 版本，所以需要手动指定版本号，如果不这样
//做，则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行

//Code generated by protoc-gen-go. DO NOT EDIT.
//versions:
//protoc-gen-go v1.21.0
//protoc        v3.11.4
//source: demo3.proto

package bar

import (
    proto "github.com/Go/protobuf/proto"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)
```

```
const (
    //Verify that this generated code is sufficiently up-to-date.
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up-to-date.
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)

//This is a compile-time assertion that a sufficiently up-to-date version
//of the legacy proto package is being used
const _ = proto.ProtoPackageIsVersion4

type Teacher struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //姓名
    Name string `protobuf:"Bytes,1,opt,name=name,proto3" json:"name,omitempty"`
    //年龄
    Age int32 `protobuf:"varint,2,opt,name=age,proto3" json:"age,omitempty"`
    //地址
    Address string `protobuf:"Bytes,3,opt,name=address,proto3" json:"address,omitempty"`
    //repeated 关键字类似于 Go 中的切片,编译之后对应的也是 Go 的切片
    Phone []*Teacher_PhoneNumber `protobuf:"Bytes,4,rep,name=phone,proto3" json:"phone,
omitempty"`
}

func (x *Teacher) Reset() {
    *x = Teacher{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo3_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher) ProtoMessage() {}

func (x *Teacher) ProtoReflect() protoreflect.Message {
    mi := &file_demo3_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
```

```
    return mi.MessageOf(x)
}

//Deprecated: Use Teacher.ProtoReflect.Descriptor instead
func (*Teacher) Descriptor() ([]byte, []int) {
    return file_demo3_proto_rawDescGZIP(), []int{0}
}

func (x *Teacher) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *Teacher) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *Teacher) GetAddress() string {
    if x != nil {
        return x.Address
    }
    return ""
}

func (x *Teacher) GetPhone() []*Teacher_PhoneNumber {
    if x != nil {
        return x.Phone
    }
    return nil
}

//定义一个 message
type Teacher_PhoneNumber struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Number string `protobuf:"Bytes,1,opt,name=number,proto3" json:"number,omitempty"`
    Type int64 `protobuf:"varint,2,opt,name=type,proto3" json:"type,omitempty"`
}

func (x *Teacher_PhoneNumber) Reset() {
    *x = Teacher_PhoneNumber{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo3_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
```

```
    }
}

func (x * Teacher_PhoneNumber) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func ( * Teacher_PhoneNumber) ProtoMessage() {}

func (x * Teacher_PhoneNumber) ProtoReflect() protoreflect.Message {
    mi := &file_demo3_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}

//Deprecated: Use Teacher_PhoneNumber.ProtoReflect.Descriptor instead.
func ( * Teacher_PhoneNumber) Descriptor() ([]Byte, []int) {
    return file_demo3_proto_rawDescGZIP(), []int{0, 0}
}

func (x * Teacher_PhoneNumber) GetNumber() string {
    if x != nil {
        return x.Number
    }
    return ""
}

func (x * Teacher_PhoneNumber) GetType() int64 {
    if x != nil {
        return x.Type
    }
    return 0
}

var File_demo3_proto protoreflect.FileDescriptor

var file_demo3_proto_rawDesc = []Byte{
    0x0a, 0x0b, 0x64, 0x65, 0x6d, 0x6f, 0x33, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x22, 0xb0,
0x01,
    0x0a, 0x07, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61,
0x6d,
    0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10,
0x0a,
    0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65,
0x12,
```

```
    0x18, 0x0a, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28,
0x09,
    0x52, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x12, 0x2a, 0x0a, 0x05, 0x70, 0x68,
0x6f,
    0x6e, 0x65, 0x18, 0x04, 0x20, 0x03, 0x28, 0x0b, 0x32, 0x14, 0x2e, 0x54, 0x65, 0x61, 0x63,
0x68,
    0x65, 0x72, 0x2e, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x52,
0x05,
    0x70, 0x68, 0x6f, 0x6e, 0x65, 0x1a, 0x39, 0x0a, 0x0b, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e,
0x75,
    0x6d, 0x62, 0x65, 0x72, 0x12, 0x16, 0x0a, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x18,
0x01,
    0x20, 0x01, 0x28, 0x09, 0x52, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x12, 0x12, 0x0a,
0x04,
    0x74, 0x79, 0x70, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x03, 0x52, 0x04, 0x74, 0x79, 0x70,
0x65,
    0x42, 0x15, 0x5a, 0x13, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x63, 0x6f, 0x6d,
0x2f,
    0x66, 0x6f, 0x6f, 0x2f, 0x62, 0x61, 0x72, 0x62, 0x06, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x33,
}

var (
    file_demo3_proto_rawDescOnce sync.Once
    file_demo3_proto_rawDescData = file_demo3_proto_rawDesc
)

func file_demo3_proto_rawDescGZIP() []Byte {
    file_demo3_proto_rawDescOnce.Do(func() {
        file_demo3_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo3_proto_
rawDescData)
    })
    return file_demo3_proto_rawDescData
}

var file_demo3_proto_msgTypes = make([]protoimpl.MessageInfo, 2)
var file_demo3_proto_goTypes = []interface{}{
    (*Teacher)(nil),                    //0: Teacher
    (*Teacher_PhoneNumber)(nil),        //1: Teacher.PhoneNumber
}
var file_demo3_proto_depIdxs = []int32{
    1, //0: Teacher.phone:type_name -> Teacher.PhoneNumber
    1, //[1:1] is the sub-list for method output_type
    1, //[1:1] is the sub-list for method input_type
    1, //[1:1] is the sub-list for extension type_name
    1, //[1:1] is the sub-list for extension extendee
    0, //[0:1] is the sub-list for field type_name
}

func init() { file_demo3_proto_init() }
func file_demo3_proto_init() {
    if File_demo3_proto != nil {
```

```
        return
    }
    if ! protoimpl.UnsafeEnabled {
        file_demo3_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_demo3_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher_PhoneNumber); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo3_proto_rawDesc,
            NumEnums:      0,
            NumMessages:   2,
            NumExtensions: 0,
            NumServices:   0,
        },
        GoTypes:           file_demo3_proto_goTypes,
        DependencyIndexes: file_demo3_proto_depIdxs,
        MessageInfos:      file_demo3_proto_msgTypes,
    }.Build()
    File_demo3_proto = out.File
    file_demo3_proto_rawDesc = nil
    file_demo3_proto_goTypes = nil
    file_demo3_proto_depIdxs = nil
}
```

（3）enum 关键字，示例代码如下：

```
//anonymous - link\example\chapter5\protobuf_enum.proto
```

```
//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";
//.proto 文件应包含一个 go_package 选项,用于指定包含所生成代码的 Go 软件包的完整导入路径
//(最后一次"bar"就是生成 Go 文件的包名),官方在未来的发行版本会支持
option go_package = "example.com/foo/bar";

message Teacher{
    //姓名
    string name = 1;

    //年龄
    int32 age = 2 ;

    //地址
    string address = 3;

    //定义一个 message
    message PhoneNumber{
        string number = 1;
        PhoneType type = 2;
    }

    //repeated 关键字类似于 Go 中的切片,编译之后对应的也是 Go 的切片
    repeated PhoneNumber phone = 4 ;
}
//enum 为关键字,作用为定义一种枚举类型
    enum PhoneType {
    /*
        enum 还可以为不同的枚举常量指定相同的值来定义别名
        如果想要使用这个功能,则必须将 allow_alias 选项设置为 true,否则编译器将报错
    */
    option allow_alias = true;

    /*
        如下所示 enum 的第 1 个常量映射为 0,每个枚举定义必须包含一个映射到 0 的常量作为其
第 1 个元素
        这是因为必须有一个零值,以便可以使用 0 作为数字默认值
        零值必须是第 1 个元素,以便与 proto 2 语义兼容,其中第 1 个枚举值始终是默认值
        解析数据时,如果编码的消息不包含特定的单数元素,则解析对象中的相应字段将设置为
该字段的默认值
            不同类型的默认值不同,具体如下:
                对于字符串,默认值为空字符串
                对于字节,默认值为空字节
                对于 bools,默认值为 false
                对于数字类型,默认值为 0
                对于枚举,默认值为第 1 个定义的枚举值,该值必须为 0
                repeated 字段的默认值为空列表
                message 字段的默认值为空对象
    */
    MOBILE = 0;
```

```
    HOME = 1;
    WORK = 2;
    Personal = 2;
}
```

使用命令 protoc--go_out=. demo4.proto 生成对应的 Go 代码，代码如下：

```
//anonymous - link\example\chapter5\protobuf_enum.go

//ProtoBuf 默认支持的版本是 2.x，现在一般使用 3.x 版本，所以需要手动指定版本号，如果不这样
//做，则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
//Code generated by protoc - gen - go. DO NOT EDIT.//versions://protoc - gen - go v1.21.0//
protoc v3.11.4//source: demo4.proto
package bar

import (
    proto "github.com/Go/protobuf/proto"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)
const (
    //Verify that this generated code is sufficiently up - to - date
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up - to - date
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)
//This is a compile - time assertion that a sufficiently up - to - date version of the legacy
//proto package is being used.const _ = proto.ProtoPackageIsVersion4
//enum 为关键字，作用为定义一种枚举类型
type PhoneType int32 const (
    //如下所示，enum 的第 1 个常量映射为 0，每个枚举定义必须包含一个映射到 0 的常量作为其第
    //1 个元素
    //这是因为必须有一个 0 值，以便可以使用 0 作为数字默认值
    //0 值必须是第 1 个元素，以便与 proto 2 语义兼容，其中第 1 个枚举值始终是默认值
    //默认值
    //解析数据时，如果编码的消息不包含特定的单数元素，则解析对象中的相应字段将设置为该
    //字段的默认值
    //不同类型的默认值不同，具体如下
    //对于字符串，默认值为空字符串
    //对于字节，默认值为空字节
    //对于 bools，默认值为 false
    //对于数字类型，默认值为 0
    //对于枚举，默认值为第 1 个定义的枚举值，该值必须为 0
    //repeated 字段的默认值为空列表
    //message 字段的默认值为空对象
    PhoneType_MOBILE   PhoneType = 0
    PhoneType_HOME     PhoneType = 1
    PhoneType_WORK     PhoneType = 2
```

```
    PhoneType_Personal     PhoneType = 2
)
//Enum value maps for PhoneType
    var (
    PhoneType_name = map[int32]string{
        0: "MOBILE",
        1: "HOME",
        2: "WORK",
        //Duplicate value
    2: "Personal",
    }
    PhoneType_value = map[string]int32{
        "MOBILE": 0,
        "HOME": 1,
        "WORK": 2,
        "Personal": 2,
    }
)

func (x PhoneType) Enum() *PhoneType {
    p := new(PhoneType)
    *p = x
    return p
}

func (x PhoneType) String() string {
    return protoimpl.X.EnumStringOf(x.Descriptor(), protoreflect.EnumNumber(x))
}

func (PhoneType) Descriptor() protoreflect.EnumDescriptor {
    return file_demo4_proto_enumTypes[0].Descriptor()
}

func (PhoneType) Type() protoreflect.EnumType {
    return &file_demo4_proto_enumTypes[0]
}

func (x PhoneType) Number() protoreflect.EnumNumber {
    return protoreflect.EnumNumber(x)
}
//Deprecated: Use PhoneType.Descriptor instead
func (PhoneType) EnumDescriptor() ([]Byte, []int) {
    return file_demo4_proto_rawDescGZIP(), []int{0}
}

type Teacher struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //姓名
```

```
    Name string `protobuf:"Bytes,1,opt,name = name,proto3" json:"name,omitempty"`
    //年龄
    Age int32 `protobuf:"varint,2,opt,name = age,proto3" json:"age,omitempty"`
    //地址
    Address string `protobuf:"Bytes,3,opt,name = address,proto3" json:"address,omitempty"`
    //repeated 关键字类似于 Go 中的切片,编译之后对应的也是 Go 的切片
    Phone [] * Teacher_PhoneNumber `protobuf:"Bytes,4,rep,name = phone,proto3" json:"phone,
omitempty"`
}

func (x * Teacher) Reset() {
    * x = Teacher{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo4_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x * Teacher) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func ( * Teacher) ProtoMessage() {}

func (x * Teacher) ProtoReflect() protoreflect.Message {
    mi := &file_demo4_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
//Deprecated: Use Teacher.ProtoReflect.Descriptor instead
func ( * Teacher) Descriptor() ([]Byte, []int) {
    return file_demo4_proto_rawDescGZIP(), []int{0}
}

func (x * Teacher) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x * Teacher) GetAge() int32 {
    if x != nil {
        return x.Age
```

```
    }
    return 0
}

func (x *Teacher) GetAddress() string {
    if x != nil {
        return x.Address
    }
    return ""
}

func (x *Teacher) GetPhone() []*Teacher_PhoneNumber {
    if x != nil {
        return x.Phone
    }
    return nil
}
//定义一个 message
type Teacher_PhoneNumber struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Number string `protobuf:"Bytes,1,opt,name=number,proto3" json:"number,omitempty"`
    Type PhoneType `protobuf:"varint,2,opt,name=type,proto3,enum=PhoneType" json:"type,
omitempty"`
}

func (x *Teacher_PhoneNumber) Reset() {
    *x = Teacher_PhoneNumber{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo4_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher_PhoneNumber) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher_PhoneNumber) ProtoMessage() {}

func (x *Teacher_PhoneNumber) ProtoReflect() protoreflect.Message {
    mi := &file_demo4_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
```

```
    }
    return mi.MessageOf(x)
}
//Deprecated: Use Teacher_PhoneNumber.ProtoReflect.Descriptor instead
func (*Teacher_PhoneNumber) Descriptor() ([]Byte, []int) {
    return file_demo4_proto_rawDescGZIP(), []int{0, 0}
}

func (x *Teacher_PhoneNumber) GetNumber() string {
    if x != nil {
        return x.Number
    }
    return ""
}

func (x *Teacher_PhoneNumber) GetType() PhoneType {
    if x != nil {
        return x.Type
    }
    return PhoneType_MOBILE
}
var File_demo4_proto protoreflect.FileDescriptor
var file_demo4_proto_rawDesc = []Byte{
    0x0a, 0x0b, 0x64, 0x65, 0x6d, 0x6f, 0x34, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x22, 0xbc,
0x01,
    0x0a, 0x07, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61,
0x6d,
    0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10,
0x0a,
    0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65,
0x12,
    0x18, 0x0a, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28,
0x09,
    0x52, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x12, 0x2a, 0x0a, 0x05, 0x70, 0x68,
0x6f,
    0x6e, 0x65, 0x18, 0x04, 0x20, 0x03, 0x28, 0x0b, 0x32, 0x14, 0x2e, 0x54, 0x65, 0x61, 0x63,
0x68,
    0x65, 0x72, 0x2e, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x52,
0x05,
    0x70, 0x68, 0x6f, 0x6e, 0x65, 0x1a, 0x45, 0x0a, 0x0b, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e,
0x75,
    0x6d, 0x62, 0x65, 0x72, 0x12, 0x16, 0x0a, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x18,
0x01,
    0x20, 0x01, 0x28, 0x09, 0x52, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x12, 0x1e, 0x0a,
0x04,
    0x74, 0x79, 0x70, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x0e, 0x32, 0x0a, 0x2e, 0x50, 0x68,
0x6f,
    0x6e, 0x65, 0x54, 0x79, 0x70, 0x65, 0x52, 0x04, 0x74, 0x79, 0x70, 0x65, 0x2a, 0x3d, 0x0a,
0x09,
    0x50, 0x68, 0x6f, 0x6e, 0x65, 0x54, 0x79, 0x70, 0x65, 0x12, 0x0a, 0x0a, 0x06, 0x4d, 0x4f,
0x42,
```

```
    0x49, 0x4c, 0x45, 0x10, 0x00, 0x12, 0x08, 0x0a, 0x04, 0x48, 0x4f, 0x4d, 0x45, 0x10, 0x01,
0x12,
    0x08, 0x0a, 0x04, 0x57, 0x4f, 0x52, 0x4b, 0x10, 0x02, 0x12, 0x0c, 0x0a, 0x08, 0x50, 0x65,
0x72,
    0x73, 0x6f, 0x6e, 0x61, 0x6c, 0x10, 0x02, 0x1a, 0x02, 0x10, 0x01, 0x42, 0x15, 0x5a, 0x13,
0x65,
    0x78, 0x61, 0x6d, 0x70, 0x6c, 0x65, 0x2e, 0x63, 0x6f, 0x6d, 0x2f, 0x66, 0x6f, 0x6f, 0x2f,
0x62,
    0x61, 0x72, 0x62, 0x06, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x33,
}
var (
    file_demo4_proto_rawDescOnce sync.Once
    file_demo4_proto_rawDescData = file_demo4_proto_rawDesc
)

func file_demo4_proto_rawDescGZIP() []Byte {
    file_demo4_proto_rawDescOnce.Do(func() {
        file_demo4_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo4_proto_
rawDescData)
    })
    return file_demo4_proto_rawDescData
}
var file_demo4_proto_enumTypes = make([]protoimpl.EnumInfo, 1) var file_demo4_proto_
msgTypes = make([]protoimpl.MessageInfo, 2)var file_demo4_proto_goTypes = []interface{}{
    (PhoneType)(0),                 //0: PhoneType
    (*Teacher)(nil),                //1: Teacher
    (*Teacher_PhoneNumber)(nil),    //2: Teacher.PhoneNumber
}
var file_demo4_proto_depIdxs = []int32{
    2, //0: Teacher.phone:type_name -> Teacher.PhoneNumber
    0, //1: Teacher.PhoneNumber.type:type_name -> PhoneType
    2, //[2:2] is the sub-list for method output_type
    2, //[2:2] is the sub-list for method input_type
    2, //[2:2] is the sub-list for extension type_name
    2, //[2:2] is the sub-list for extension extendee
    0, //[0:2] is the sub-list for field type_name
}

func init() { file_demo4_proto_init() }
func file_demo4_proto_init() {
    if File_demo4_proto != nil {
        return
    }
    if !protoimpl.UnsafeEnabled {
        file_demo4_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
```

```
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_demo4_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher_PhoneNumber); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo4_proto_rawDesc,
            NumEnums:     1,
            NumMessages:  2,
            NumExtensions:0,
            NumServices:  0,
        },
        GoTypes:          file_demo4_proto_goTypes,
        DependencyIndexes:file_demo4_proto_depIdxs,
        EnumInfos:        file_demo4_proto_enumTypes,
        MessageInfos:     file_demo4_proto_msgTypes,
    }.Build()
    File_demo4_proto = out.File
    file_demo4_proto_rawDesc = nil
    file_demo4_proto_goTypes = nil
    file_demo4_proto_depIdxs = nil
}
```

(4) oneof 关键字(C 语言中的联合体),示例代码如下:

```
//anonymous - link\example\chapter5\protobuf_oneof.proto

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";
//.proto 文件应包含一个 go_package 选项,用于指定包含所生成代码的 Go 软件包的完整导入路径
//(最后一次"bar"就是生成 Go 文件的包名),官方在未来的发行版本会支持
option go_package = "example.com/foo/bar";
message Teacher{
```

```
    //姓名
    string name = 1;

    //年龄
    int32 age = 2 ;

    //地址
    string address = 3;

    //定义一个 message
    message PhoneNumber{
        string number = 1;
        PhoneType type = 2;
    }

    //repeated 关键字类似于 Go 中的切片,编译之后对应的也是 Go 的切片
    repeated PhoneNumber phone = 4 ;

    //如果有一个包含许多字段的消息,并且最多只能同时设置其中的一个字段,则可以使用 oneof 功能
    oneof data{
        string school = 5;
        int32 score = 6;
    }
}
//enum 为关键字,作用为定义一种枚举类型
    enum PhoneType {
    /*
        enum 还可以为不同的枚举常量指定相同的值来定义别名。
        如果想要使用这个功能必须将 allow_alias 选项设置为 true,负责编译器将报错
    */
    option allow_alias = true;

    /*
        如下所示,enum 的第 1 个常量映射为 0,每个枚举定义必须包含一个映射到 0 的常量作为其
第 1 个元素。
        这是因为必须有一个 0 值,以便可以使用 0 作为数字默认值
        0 值必须是第 1 个元素,以便与 proto 2 语义兼容,其中第 1 个枚举值始终是默认值

        默认值
            解析数据时,如果编码的消息不包含特定的单数元素,则解析对象中的相应字段将设
置为该字段的默认值
            不同类型的默认值不同,具体如下:
                对于字符串,默认值为空字符串
                对于字节,默认值为空字节
                对于 bools,默认值为 false
                对于数字类型,默认值为 0
                对于枚举,默认值为第 1 个定义的枚举值,该值必须为 0
                repeated 字段的默认值为空列表
                message 字段的默认值为空对象
    */
    MOBILE = 0;
    HOME = 1;
```

```
    WORK = 2;
    Personal = 2;
}
```

使用命令 protoc--go_out=. demo5.proto 生成对应的 Go 代码，代码如下：

```
//anonymous-link\example\chapter5\protobuf_oneof.go

//ProtoBuf 默认支持的版本是 2.x，现在一般使用 3.x 版本，所以需要手动指定版本号，如果不这样
//做，则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
//Code generated by protoc-gen-go. DO NOT EDIT
//versions:
//protoc-gen-go v1.21.0
//protoc          v3.11.4
//source: demo5.proto
package bar

import (
    proto "github.com/Go/protobuf/proto"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)
const (
    //Verify that this generated code is sufficiently up-to-date
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up-to-date
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)
//This is a compile-time assertion that a sufficiently up-to-date
//version of the legacy proto package is being used
const _ = proto.ProtoPackageIsVersion4
//enum 为关键字，作用为定义一种枚举类型
enumPhoneType {
/*
enum 还可以为不同的枚举常量指定相同的值来定义别名。
如果想要使用这个功能，则必须将 allow_alias 选项设置为 true，否则编译器将报错
*/
option allow_alias = true;
    //如下所示，enum 的第 1 个常量映射为 0，每个枚举定义必须包含一个映射到 0 的常量作为其第
    //1 个元素
    //这是因为必须有一个 0 值，以便可以使用 0 作为数字默认值
    //0 值必须是第 1 个元素，以便与 proto 2 语义兼容，其中第 1 个枚举值始终是默认值
    //
    //默认值
    //解析数据时，如果编码的消息不包含特定的单数元素，则解析对象中的相应字段将设置为该
    //字段的默认值
    //不同类型的默认值不同，具体如下
    //对于字符串，默认值为空字符串
```

```
        //对于字节,默认值为空字节
        //对于 bools,默认值为 false
        //对于数字类型,默认值为 0
        //对于枚举,默认值为第 1 个定义的枚举值,该值必须为 0
        //repeated 字段的默认值为空列表
        //message 字段的默认值为空对象
        PhoneType_MOBILE     PhoneType = 0
        PhoneType_HOME       PhoneType = 1
        PhoneType_WORK       PhoneType = 2
        PhoneType_Personal   PhoneType = 2
    )
    //Enum value maps for PhoneType
        var (
        PhoneType_name = map[int32]string{
            0: "MOBILE",
            1: "HOME",
            2: "WORK",
            //Duplicate value
        2: "Personal",
        }
        PhoneType_value = map[string]int32{
            "MOBILE":     0,
            "HOME":       1,
            "WORK":       2,
            "Personal":   2,
        }
    )

    func (x PhoneType) Enum() *PhoneType {
        p := new(PhoneType)
        *p = x
        return p
    }

    func (x PhoneType) String() string {
        return protoimpl.X.EnumStringOf(x.Descriptor(), protoreflect.EnumNumber(x))
    }

    func (PhoneType) Descriptor() protoreflect.EnumDescriptor {
        return file_demo5_proto_enumTypes[0].Descriptor()
    }

    func (PhoneType) Type() protoreflect.EnumType {
        return &file_demo5_proto_enumTypes[0]
    }

    func (x PhoneType) Number() protoreflect.EnumNumber {
        return protoreflect.EnumNumber(x)
    }
    //Deprecated: Use PhoneType.Descriptor instead
    func (PhoneType) EnumDescriptor() ([]Byte, []int) {
```

```
    return file_demo5_proto_rawDescGZIP(), []int{0}
}

type Teacher struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //姓名
    Name string `protobuf:"Bytes,1,opt,name=name,proto3" json:"name,omitempty"`
    //年龄
    Age int32 `protobuf:"varint,2,opt,name=age,proto3" json:"age,omitempty"`
    //地址
    Address string `protobuf:"Bytes,3,opt,name=address,proto3" json:"address,omitempty"`
    //repeated 关键字类似于 Go 中的切片,编译之后对应的也是 Go 的切片
    Phone []*Teacher_PhoneNumber `protobuf:"Bytes,4,rep,name=phone,proto3" json:"phone,
omitempty"`
    //如果有一个包含许多字段的消息,并且最多只能同时设置其中的一个字段,则可以使用 oneof
    //功能
    //
    //Types that are assignable to Data:
    //*Teacher_School
    //*Teacher_Score
    Data isTeacher_Data `protobuf_oneof:"data"`
}

func (x *Teacher) Reset() {
    *x = Teacher{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo5_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher) ProtoMessage() {}

func (x *Teacher) ProtoReflect() protoreflect.Message {
    mi := &file_demo5_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
```

```
}
//Deprecated: Use Teacher.ProtoReflect.Descriptor instead
func (*Teacher) Descriptor() ([]byte, []int) {
    return file_demo5_proto_rawDescGZIP(), []int{0}
}

func (x *Teacher) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *Teacher) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *Teacher) GetAddress() string {
    if x != nil {
        return x.Address
    }
    return ""
}

func (x *Teacher) GetPhone() []*Teacher_PhoneNumber {
    if x != nil {
        return x.Phone
    }
    return nil
}

func (m *Teacher) GetData() isTeacher_Data {
    if m != nil {
        return m.Data
    }
    return nil
}

func (x *Teacher) GetSchool() string {
    if x, ok := x.GetData().(*Teacher_School); ok {
        return x.School
    }
    return ""
}

func (x *Teacher) GetScore() int32 {
    if x, ok := x.GetData().(*Teacher_Score); ok {
        return x.Score
```

```
    }
    return 0
}

type isTeacher_Data interface {
    isTeacher_Data()
}

type Teacher_School struct {
    School string `protobuf:"Bytes,5,opt,name=school,proto3,oneof"`
}

type Teacher_Score struct {
    Score int32 `protobuf:"varint,6,opt,name=score,proto3,oneof"`
}

func (*Teacher_School) isTeacher_Data() {}

func (*Teacher_Score) isTeacher_Data() {}
//定义一个 message
type Teacher_PhoneNumber struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    Number string `protobuf:"Bytes,1,opt,name=number,proto3" json:"number,omitempty"`
    Type PhoneType `protobuf:"varint,2,opt,name=type,proto3,enum=PhoneType" json:"type,
omitempty"`
}

func (x *Teacher_PhoneNumber) Reset() {
    *x = Teacher_PhoneNumber{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo5_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher_PhoneNumber) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher_PhoneNumber) ProtoMessage() {}

func (x *Teacher_PhoneNumber) ProtoReflect() protoreflect.Message {
    mi := &file_demo5_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
```

```
            }
            return ms
        }
        return mi.MessageOf(x)
    }
    //Deprecated: Use Teacher_PhoneNumber.ProtoReflect.Descriptor instead
    func ( * Teacher_PhoneNumber) Descriptor() ([]Byte, []int) {
        return file_demo5_proto_rawDescGZIP(), []int{0, 0}
    }

    func (x * Teacher_PhoneNumber) GetNumber() string {
        if x ! = nil {
            return x.Number
        }
        return ""
    }

    func (x * Teacher_PhoneNumber) GetType() PhoneType {
        if x ! = nil {
            return x.Type
        }
        return PhoneType_MOBILE
    }
    var File_demo5_proto protoreflect.FileDescriptor
    var file_demo5_proto_rawDesc = []Byte{
        0x0a, 0x0b, 0x64, 0x65, 0x6d, 0x6f, 0x35, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x22, 0xf6,
    0x01,
        0x0a, 0x07, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61,
    0x6d,
        0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10,
    0x0a,
        0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65,
    0x12,
        0x18, 0x0a, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28,
    0x09,
        0x52, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x12, 0x2a, 0x0a, 0x05, 0x70, 0x68,
    0x6f,
        0x6e, 0x65, 0x18, 0x04, 0x20, 0x03, 0x28, 0x0b, 0x32, 0x14, 0x2e, 0x54, 0x65, 0x61, 0x63,
    0x68,
        0x65, 0x72, 0x2e, 0x50, 0x68, 0x6f, 0x6e, 0x65, 0x4e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x52,
    0x05,
        0x70, 0x68, 0x6f, 0x6e, 0x65, 0x12, 0x18, 0x0a, 0x06, 0x73, 0x63, 0x68, 0x6f, 0x6f, 0x6c,
    0x18,
        0x05, 0x20, 0x01, 0x28, 0x09, 0x48, 0x00, 0x52, 0x06, 0x73, 0x63, 0x68, 0x6f, 0x6f, 0x6c,
    0x12,
        0x16, 0x0a, 0x05, 0x73, 0x63, 0x6f, 0x72, 0x65, 0x18, 0x06, 0x20, 0x01, 0x28, 0x05, 0x48,
    0x00,
        0x52, 0x05, 0x73, 0x63, 0x6f, 0x72, 0x65, 0x1a, 0x45, 0x0a, 0x0b, 0x50, 0x68, 0x6f, 0x6e,
    0x65,
        0x4e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x12, 0x16, 0x0a, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65,
    0x72,
```

```
    0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x06, 0x6e, 0x75, 0x6d, 0x62, 0x65, 0x72, 0x12,
0x1e,
    0x0a, 0x04, 0x74, 0x79, 0x70, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x0e, 0x32, 0x0a, 0x2e,
0x50,
    0x68, 0x6f, 0x6e, 0x65, 0x54, 0x79, 0x70, 0x65, 0x52, 0x04, 0x74, 0x79, 0x70, 0x65, 0x42,
0x06,
    0x0a, 0x04, 0x64, 0x61, 0x74, 0x61, 0x2a, 0x3d, 0x0a, 0x09, 0x50, 0x68, 0x6f, 0x6e, 0x65,
0x54,
    0x79, 0x70, 0x65, 0x12, 0x0a, 0x0a, 0x06, 0x4d, 0x4f, 0x42, 0x49, 0x4c, 0x45, 0x10, 0x00,
0x12,
    0x08, 0x0a, 0x04, 0x48, 0x4f, 0x4d, 0x45, 0x10, 0x01, 0x12, 0x08, 0x0a, 0x04, 0x57, 0x4f,
0x52,
    0x4b, 0x10, 0x02, 0x12, 0x0c, 0x0a, 0x08, 0x50, 0x65, 0x72, 0x73, 0x6f, 0x6e, 0x61, 0x6c,
0x10,
    0x02, 0x1a, 0x02, 0x10, 0x01, 0x42, 0x15, 0x5a, 0x13, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c,
0x65,
    0x2e, 0x63, 0x6f, 0x6d, 0x2f, 0x66, 0x6f, 0x6f, 0x2f, 0x62, 0x61, 0x72, 0x62, 0x06, 0x70,
0x72,
    0x6f, 0x74, 0x6f, 0x33,
}
var (
    file_demo5_proto_rawDescOnce sync.Once
    file_demo5_proto_rawDescData = file_demo5_proto_rawDesc
)

func file_demo5_proto_rawDescGZIP() []Byte {
    file_demo5_proto_rawDescOnce.Do(func() {
        file_demo5_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo5_proto_
rawDescData)
    })
    return file_demo5_proto_rawDescData
}
var file_demo5_proto_enumTypes = make([]protoimpl.EnumInfo, 1) var file_demo5_proto_
msgTypes = make([]protoimpl.MessageInfo, 2)var file_demo5_proto_goTypes = []interface{}{
    (PhoneType)(0),               //0: PhoneType
    (*Teacher)(nil),              //1: Teacher
    (*Teacher_PhoneNumber)(nil), //2: Teacher.PhoneNumber
}
var file_demo5_proto_depIdxs = []int32{
    2, //0: Teacher.phone:type_name -> Teacher.PhoneNumber
    0, //1: Teacher.PhoneNumber.type:type_name -> PhoneType
    2, //[2:2] is the sub-list for method output_type
    2, //[2:2] is the sub-list for method input_type
    2, //[2:2] is the sub-list for extension type_name
    2, //[2:2] is the sub-list for extension extendee
    0, //[0:2] is the sub-list for field type_name
}

func init() { file_demo5_proto_init() }
func file_demo5_proto_init() {
    if File_demo5_proto != nil {
        return
```

```
    }
    if ! protoimpl.UnsafeEnabled {
        file_demo5_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_demo5_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher_PhoneNumber); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
    }
    file_demo5_proto_msgTypes[0].OneofWrappers = []interface{}{
        (*Teacher_School)(nil),
        (*Teacher_Score)(nil),
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo5_proto_rawDesc,
            NumEnums:      1,
            NumMessages:   2,
            NumExtensions: 0,
            NumServices:   0,
        },
        GoTypes:           file_demo5_proto_goTypes,
        DependencyIndexes: file_demo5_proto_depIdxs,
        EnumInfos:         file_demo5_proto_enumTypes,
        MessageInfos:      file_demo5_proto_msgTypes,
    }.Build()
    File_demo5_proto = out.File
    file_demo5_proto_rawDesc = nil
    file_demo5_proto_goTypes = nil
    file_demo5_proto_depIdxs = nil
}
```

(5) 定义 RPC 服务,示例代码如下:

```
//anonymous - link\example\chapter5\protobuf_rpc.proto

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行
syntax = "proto3";

//.proto 文件应包含一个 go_package 选项,用于指定包含所生成代码的 Go 软件包的完整导入路径
//(最后一次 bar 就是生成 Go 文件的包名),官方在未来的发行版本会支持
option go_package = "example.com/foo/bar";

message Teacher{
    //姓名
    string name = 1;
    //年龄
    int32 age = 2 ;
    //地址
    string address = 3;
}

/*
    如果需要将 message 与 RPC 一起使用,则可以在 .proto 文件中定义 RPC 服务接口,ProtoBuf 编
译器将根据选择的语言生成 RPC 接口代码
    通过定义服务,然后借助框架帮助实现部分的 RPC 代码
*/
service HelloService {
    //传入和传输的 Teacher 是上面定义的 message 对象
    rpc World (Teacher)returns (Teacher);
}
```

使用命令 protoc --go_out=plugins=grpc:. demo6.proto 生成对应的 Go 代码,代码如下:

```
//anonymous - link\example\chapter5\protobuf_rpc.go

//ProtoBuf 默认支持的版本是 2.x,现在一般使用 3.x 版本,所以需要手动指定版本号,如果不这样
//做,则协议缓冲区编译器将假定正在使用 proto 2。这也必须是文件的第 1 个非空的非注释行

//Code generated by protoc - gen - go. DO NOT EDIT.
//versions:
//protoc - gen - go v1.21.0
//protoc           v3.11.4
//source: demo6.proto

package bar

import (
    context "context"
    proto "github.com/Go/protobuf/proto"
```

```
    grpc "google.Go.org/grpc"
    codes "google.Go.org/grpc/codes"
    status "google.Go.org/grpc/status"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)

const (
    //Verify that this generated code is sufficiently up-to-date
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up-to-date
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)

//This is a compile-time assertion that a sufficiently up-to-date version
//of the legacy proto package is being used
const _ = proto.ProtoPackageIsVersion4

type Teacher struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

    //姓名
    Name string `protobuf:"Bytes,1,opt,name=name,proto3" json:"name,omitempty"`
    //年龄
    Age int32 `protobuf:"varint,2,opt,name=age,proto3" json:"age,omitempty"`
    //地址
    Address string `protobuf:"Bytes,3,opt,name=address,proto3" json:"address,omitempty"`
}

func (x *Teacher) Reset() {
    *x = Teacher{}
    if protoimpl.UnsafeEnabled {
        mi := &file_demo6_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *Teacher) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*Teacher) ProtoMessage() {}

func (x *Teacher) ProtoReflect() protoreflect.Message {
    mi := &file_demo6_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
```

```
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}

//Deprecated: Use Teacher.ProtoReflect.Descriptor instead
func (*Teacher) Descriptor() ([]Byte, []int) {
    return file_demo6_proto_rawDescGZIP(), []int{0}
}

func (x *Teacher) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *Teacher) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *Teacher) GetAddress() string {
    if x != nil {
        return x.Address
    }
    return ""
}

var File_demo6_proto protoreflect.FileDescriptor

var file_demo6_proto_rawDesc = []Byte{
    0x0a, 0x0b, 0x64, 0x65, 0x6d, 0x6f, 0x36, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x22, 0x49,
0x0a,
    0x07, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61, 0x6d,
0x65,
    0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10, 0x0a,
0x03,
    0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65, 0x12,
0x18,
    0x0a, 0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28, 0x09,
0x52,
    0x07, 0x61, 0x64, 0x64, 0x72, 0x65, 0x73, 0x73, 0x32, 0x2b, 0x0a, 0x0c, 0x48, 0x65, 0x6c,
0x6c,
    0x6f, 0x53, 0x65, 0x72, 0x76, 0x69, 0x63, 0x65, 0x12, 0x1b, 0x0a, 0x05, 0x57, 0x6f, 0x72,
0x6c,
```

```
    0x64, 0x12, 0x08, 0x2e, 0x54, 0x65, 0x61, 0x63, 0x68, 0x65, 0x72, 0x1a, 0x08, 0x2e, 0x54,
0x65,
    0x61, 0x63, 0x68, 0x65, 0x72, 0x42, 0x15, 0x5a, 0x13, 0x65, 0x78, 0x61, 0x6d, 0x70, 0x6c,
0x65,
    0x2e, 0x63, 0x6f, 0x6d, 0x2f, 0x66, 0x6f, 0x6f, 0x2f, 0x62, 0x61, 0x72, 0x62, 0x06, 0x70,
0x72,
    0x6f, 0x74, 0x6f, 0x33,
}

var (
    file_demo6_proto_rawDescOnce sync.Once
    file_demo6_proto_rawDescData = file_demo6_proto_rawDesc
)

func file_demo6_proto_rawDescGZIP() []Byte {
    file_demo6_proto_rawDescOnce.Do(func() {
         file_demo6_proto_rawDescData = protoimpl.X.CompressGZIP(file_demo6_proto_
rawDescData)
    })
    return file_demo6_proto_rawDescData
}

var file_demo6_proto_msgTypes = make([]protoimpl.MessageInfo, 1)
var file_demo6_proto_goTypes = []interface{}{
    (*Teacher)(nil), //0: Teacher
}
var file_demo6_proto_depIdxs = []int32{
    0, //0: HelloService.World:input_type -> Teacher
    0, //1: HelloService.World:output_type -> Teacher
    1, //[1:2] is the sub-list for method output_type
    0, //[0:1] is the sub-list for method input_type
    0, //[0:0] is the sub-list for extension type_name
    0, //[0:0] is the sub-list for extension extendee
    0, //[0:0] is the sub-list for field type_name
}

func init() { file_demo6_proto_init() }
func file_demo6_proto_init() {
    if File_demo6_proto != nil {
        return
    }
    if !protoimpl.UnsafeEnabled {
        file_demo6_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Teacher); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
```

```
            }
        }
    }
    type x struct{}
    out := protoimpl.TypeBuilder{
        File: protoimpl.DescBuilder{
            GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
            RawDescriptor: file_demo6_proto_rawDesc,
            NumEnums:      0,
            NumMessages:   1,
            NumExtensions: 0,
            NumServices:   1,
        },
        GoTypes:           file_demo6_proto_goTypes,
        DependencyIndexes: file_demo6_proto_depIdxs,
        MessageInfos:      file_demo6_proto_msgTypes,
    }.Build()
    File_demo6_proto = out.File
    file_demo6_proto_rawDesc = nil
    file_demo6_proto_goTypes = nil
    file_demo6_proto_depIdxs = nil
}

//Reference imports to suppress errors if they are not otherwise used
var _ context.Context
var _ grpc.ClientConnInterface

//This is a compile-time assertion to ensure that this generated file
//is compatible with the grpc package it is being compiled against
const _ = grpc.SupportPackageIsVersion6

//HelloServiceClient is the client API for HelloService service
//
//For semantics around ctx use and closing/ending streaming RPCs, please refer to https://godoc.
//org/google.Go.org/grpc#ClientConn.NewStream
type HelloServiceClient interface {
    World(ctx context.Context, in *Teacher, opts ...grpc.CallOption) (*Teacher, error)
}

type helloServiceClient struct {
    cc grpc.ClientConnInterface
}

func NewHelloServiceClient(cc grpc.ClientConnInterface) HelloServiceClient {
    return &helloServiceClient{cc}
}

func (c *helloServiceClient) World(ctx context.Context, in *Teacher, opts ...grpc.
CallOption) (*Teacher, error) {
    out := new(Teacher)
    err := c.cc.Invoke(ctx, "/HelloService/World", in, out, opts...)
    if err != nil {
```

```
        return nil, err
    }
    return out, nil
}

//HelloServiceServer is the server API for HelloService service
type HelloServiceServer interface {
    World(context.Context, *Teacher) (*Teacher, error)
}

//UnimplementedHelloServiceServer can be embedded to have forward compatible implementations
type UnimplementedHelloServiceServer struct {
}

func (*UnimplementedHelloServiceServer) World(context.Context, *Teacher) (*Teacher,
error) {
    return nil, status.Errorf(codes.Unimplemented, "method World not implemented")
}

func RegisterHelloServiceServer(s *grpc.Server, srv HelloServiceServer) {
    s.RegisterService(&_HelloService_serviceDesc, srv)
}

func _HelloService_World_Handler(srv interface{}, ctx context.Context, dec func(interface{})
error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
    in := new(Teacher)
    if err := dec(in); err != nil {
        return nil, err
    }
    if interceptor == nil {
        return srv.(HelloServiceServer).World(ctx, in)
    }
    info := &grpc.UnaryServerInfo{
        Server: srv,
        FullMethod: "/HelloService/World",
    }
    handler := func(ctx context.Context, req interface{}) (interface{}, error) {
        return srv.(HelloServiceServer).World(ctx, req.(*Teacher))
    }
    return interceptor(ctx, in, info, handler)
}

var _HelloService_serviceDesc = grpc.ServiceDesc{
    ServiceName: "HelloService",
    HandlerType: (*HelloServiceServer)(nil),
    Methods: []grpc.MethodDesc{
        {
            MethodName: "World",
            Handler: _HelloService_World_Handler,
        },
    },
```

```
        Streams: []grpc.StreamDesc{},
        Metadata: "demo6.proto",
    }
```

5. 报错问题记录

(1) Could not make proto path relative: protobuffer 案例.proto: No such file or directory。

现象描述：明明文件是存在的，但是在生成 Go 代码时总是提示找不到文件。

解决方案：这是由文件名称是中文导致的，将该文件名中的中文部分去掉就能解决该问题。

(2) --go_out: protoc-gen-go: Plugin failed with status code 1。

现象描述：提示 protoc-gen-go 不是内部命令，因此需要安装该工具。

解决方案：查看%GOPATH%\bin 目录是否有 protoc-gen-go 命令。如果没有，就直接执行以下命令。

```
go get -u github.com/Go/protobuf/protoc-gen-go
```

执行上述命令后会在%GOPATH%\bin 目录下生成一个 protoc-gen-go 命令。

(3) https fetch: Get https://google.Go.org/protobuf/types/descriptorpb? go-get=1: dial tcp 216.239.37.1: 443: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.

现象描述：提示远程连接失败，因为它访问的是谷歌公司的公网地址，由于国内政策原因，无法直接访问国外的一些特定的网站。

解决方案有两种方法：①自行 FQ；②配置代理。

推荐使用第 2 种解决方案，设置完下面几个环境变量后，go 命令将从公共代理镜像中快速拉取所需的依赖代码。设置代理后就可以下载所需要的工具，代码如下：

```
go env -w GO111MODULE=on
go env -w GOPROXY=https://goproxy.io,direct
#设置不从 proxy 的私有仓库拉取依赖，多个用逗号相隔(可选)
go env -w GOPRIVATE=*.corp.example.com
```

推荐阅读资料 https://goproxy.io/zh/和 https://goproxy.io/zh/docs/goproxyio-private.html。

5.1.20　Go 的序列化：RPC 和 GRPC

1. RPC 概述

(1) 什么是 RPC?RPC(Remote Procedure Call)是远程过程调用的缩写，通俗地说就是调用远处(一般指不同的主机)的一个函数。

（2）为什么微服务需要 RPC？使用微服务的一个好处就是，不限定服务的提供方使用什么技术选型，能够实现公司跨团队的技术解耦。

如果没有统一的服务框架、RPC 框架，各个团队的服务提供方就需要各自实现一套序列化、反序列化、网络框架、连接池、收发线程、超时处理、状态机等业务之外的重复技术劳动，造成整体低效，所以统一 RPC 框架把上述业务之外的技术劳动统一处理，是服务化首要解决的问题。

2. RPC 入门案例

在互联网时代，RPC 已经和 IPC（进程间通信）一样成为一个不可或缺的基础构件，因此 Go 语言的标准库也提供了一个简单的 RPC 实现，将以此为入口学习 RPC 的常见用法。

（1）RPC 的服务器端，示例代码如下：

```
//anonymous - link\example\chapter5\rpc_server.go
package main

import (
    "fmt"
    "net"
    "net/rpc"
)

type Zabbix struct{}
/**
定义成员方法：
    第 1 个参数是传入参数
    第 2 个参数必须是传出参数(引用类型)
Go 语言的 RPC 规则
方法只能有两个可序列化的参数，其中第 2 个参数是指针类型，并且返回一个 error 类型，同时必须是公开的方法
当调用远程函数之后，如果返回的错误不为空，则传出参数为空
*/
func (Zabbix) MonitorHosts(name string, response *string) error {
    *response = name + "主机监控中…"
    return nil
}

func main() {
    /**
    进程间交互有很多种方式，例如基于信号、共享内存、管道、套接字等方式

    (1)RPC 基于 TCP，因此需要先开启监听端口
    */
    listener, err := net.Listen("tcp", ":8888")
    if err != nil {
        fmt.Println("开启监听器失败，错误原因：", err)
        return
    }
    defer listener.Close()
```

```
    fmt.Println("服务启动成功...")

    /**
    (2)接受链接,即接受传输的数据
    */
    conn, err := listener.Accept()
    if err != nil {
        fmt.Println("建立连接失败...")
        return
    }
    defer conn.Close()
    fmt.Println("建立连接:", conn.RemoteAddr())
    /**
    (3)注册 RPC 服务,维护一个哈希表,key 值是服务名称,value 值是服务的地址。服务器有很多函数,希望被调用的函数需要注册到 RPC 上
    以下是 RegisterName 的函数签名:
        func RegisterName(name string, rcvr interface{}) error
    以下是对函数签名相关参数的说明:
        name 指的是服务名称
        rcvr 指的是结构体对象(这个结构体必须含有成员方法)
    */
    rpc.RegisterName("zabbix", new(Zabbix))

    /**
    (4)链接的处理交给 RCP 框架处理,即 RPC 调用,并返回执行后的数据,其工作原理大致分为 3 个步骤:①read,获取服务名称和方法名,获取请求数据;②调用对应服务里面的方法,获取传出数据;③write,把数据返回给 client
    */
    rpc.ServeConn(conn)
}
```

(2) RPC 的客户端,示例代码如下:

```
//anonymous - link\example\chapter5\rpc_client.go
package main

import (
    "fmt"
    "net"
    "net/rpc"
)

func main() {
    /**
    (1)首先通过 rpc.Dial 拨号 RPC 服务
默认数据传输过程中编码方式是 gob,可以选择 JSON
    */
    conn, err := net.Dial("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("链接服务器失败")
```

```
        return
    }
    defer conn.Close()
    /**
    (2)把 conn 和 rpc 进行绑定
    */
    client := rpc.NewClient(conn)

    /**
    (3)通过 client.Call 调用具体的 RPC 方法,其中 Call 函数的签名如下:
            func (client *Client) Call(serviceMethod string, args interface{}, reply
interface{}) error
        以下是对函数签名的相关参数进行补充说明
            serviceMethod: 用点号(.)链接的 RPC 服务名字和方法名字
            args: 指定输入参数
            reply: 指定输出参数
    */
    var data string
    err = client.Call("zabbix.MonitorHosts", "Nginx", &data)
    if err != nil {
        fmt.Println("远程接口调用失败,错误原因:", err)
        return
    }
    fmt.Println(data)
}
```

3. 跨语言的 RPC

标准库的 RPC 默认采用 Go 语言特有的 gob 编码,因此从其他语言调用 Go 语言实现的 RPC 服务将比较困难。跨语言是互联网时代 RPC 的一个首要条件,这里再实现一个跨语言的 RPC。得益于 RPC 的框架设计,Go 语言的 RPC 其实也是很容易实现跨语言支持的。这里将尝试通过官方自带的 net/rpc/jsonrpc 扩展实现一个跨语言 RPC。

(1) RPC 的服务器端,示例代码如下:

```
//anonymous - link\example\chapter5\cross_rpc_server.go
package main

import (
    "fmt"
    "net"
    "net/rpc"
    "net/rpc/jsonrpc"
)

type OpenFalcon struct{}
/**
定义成员方法:
    第 1 个参数是传入参数
    第 2 个参数必须是传出参数(引用类型)
```

```
Go 语言的 RPC 规则
    方法只能有两个可序列化的参数,其中第 2 个参数是指针类型,并且返回一个 error 类型,同时必须是公开的方法
当调用远程函数后,如果返回的错误不为空,则传出的参数为空
*/
func (OpenFalcon) MonitorHosts(name string, response *string) error {
    *response = name + "主机监控中..."
    return nil
}

func main() {
    /**
    进程间交互有很多种,例如基于信号、共享内存、管道、套接字等方式
    (1)RPC 基于是 TCP 的,因此需要先开启监听端口
    */
    listener, err := net.Listen("tcp", ":8888")
    if err != nil {
        fmt.Println("开启监听器失败,错误原因:", err)
        return
    }
    defer listener.Close()
    fmt.Println("服务启动成功...")

    /**
    (2)接受链接,即接受传输的数据
    */
    conn, err := listener.Accept()
    if err != nil {
        fmt.Println("建立连接失败...")
        return
    }
    defer conn.Close()
    fmt.Println("建立连接:", conn.RemoteAddr())
    /**
    (3)注册 RPC 服务,维护一个哈希表,key 值是服务名称,value 值是服务的地址。服务器有很多函数,希望被调用的函数需要注册到 RPC 上
    以下是 RegisterName 的函数签名:
        func RegisterName(name string, rcvr interface{}) error
    以下是对函数签名相关参数的说明:
        name 指的是服务名称
        rcvr 指的是结构体对象(这个结构体必须含有成员方法)
    */
    rpc.RegisterName("open_falcon", new(OpenFalcon))

    /**
    (4)链接的处理交给 RCP 框架处理,即 RPC 调用,并返回执行后的数据,其工作原理大致分为 3 个步骤:①read,获取服务名称和方法名,获取请求数据;②调用对应服务里面的方法,获取传出数据;③write,把数据返给 client
    */
    jsonrpc.ServeConn(conn)
}
```

（2）RPC 的客户端，示例代码如下：

```
//anonymous - link\example\chapter5\cross_rpc_client.go
package main

import (
    "fmt"
    "net/rpc/jsonrpc"
)

func main() {
    /**
    首先通过 rpc.Dial 拨号 RPC 服务
默认数据传输过程中的编码方式是 gob，可以选择 JSON，需要导入 net/rpc/jsonrpc 包
    */
    conn, err := jsonrpc.Dial("tcp", "127.0.0.1:8888")
    if err != nil {
        fmt.Println("链接服务器失败")
        return
    }
    defer conn.Close()

    var data string

    /**
    其中 Call 函数的签名如下：
        func (client *Client) Call(serviceMethod string, args interface{}, reply interface
{}) error
    以下对函数签名的相关参数进行补充说明：
        serviceMethod 表示用点号(.)链接的 RPC 服务名字和方法名字
        args 用于指定输入参数
        reply 用于指定输出参数
    */
    err = conn.Call("open_falcon.MonitorHosts", "Httpd", &data)
    if err != nil {
        fmt.Println("远程接口调用失败，错误原因：", err)
        return
    }
    fmt.Println(data)
}
```

4. GRPC 框架

（1）什么是 GRPC？GRPC 是谷歌公司基于 ProtoBuf 开发的跨语言的开源 RPC 框架。GRPC 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计。目前提供 C、Java 和 Go 语言版本，分别是 GRPC、GRPC-Java、GRPC-Go，其中 C 版本支持 C、C++、Node.js，Python、Ruby、Objective-C、PHP 和 C#。

GRPC 基于 HTTP/2 标准设计，带来诸如双向流、流控、头部压缩、单 TCP 连接上的多复用请求等特性。这些特性使其在移动设备上表现更好、更省电和更节省空间占用。

详细特性和使用推荐阅读 GRPC 官方文档中文版 http://doc.oschina.net/grpc?t=60133 和 GRPC 官网 https://grpc.io。

(2) 安装 GRPC 环境。

安装 GRPC 环境的命令如下：

```
go get -u -v google.Go.org/grpc
```

(3) 基于 ProtoBuf 编写 GRPC 服务，示例代码如下：

```
//anonymous-link\example\chapter5\protobuf_grpc.proto

//ProtoBuf 默认支持的版本是 2.0,现在一般使用 3.0 版本,所以需要手动指定版本号
//C 语言的编程风格
syntax = "proto3";
//指定包名 package pb;
//定义传输数据的格式
    message People{
    string name = 1; //1 表示数据库中表的主键 id 等于 1,主键不能重复,标示位数据不能重复
    //标示位不能使用 19 000 ~19 999(系统预留位)
    int32 age = 2;

    //结构体嵌套
    student s = 3;
    //使用数组/切片
    repeated string phone = 4;

    //oneof 的作用是多选一
    oneof data{
        int32 score = 5;
        string city = 6;
        bool good = 7;
    }
}
//oneof C 语言中的联合体
message student{
    string name = 1;
    int32 age = 6;
}
//通过先定义服务,然后借助框架,帮助实现部分的 RPC 代码
    service Hello{
    rpc World(student)returns(student);
}
```

命令行执行 protoc --go_out=plugins=grpc:. grpc.proto 生成 grpc.pb.go 文件，示例代码如下：

```
//anonymous-link\example\chapter5\protobuf_grpc.go

//ProtoBuf 默认支持的版本是 2.0,现在一般使用 3.0 版本,所以需要手动指定版本号
```

```
//C 语言的编程风格
//Code generated by protoc - gen - go. DO NOT EDIT
//versions:
//protoc - gen - go v1.21.0
//protoc           v3.11.4
//source: grpc.proto
//指定包名
package pb

import (
    context "context"
    proto "github.com/Go/protobuf/proto"
    grpc "google.Go.org/grpc"
    codes "google.Go.org/grpc/codes"
    status "google.Go.org/grpc/status"
    protoreflect "google.Go.org/protobuf/reflect/protoreflect"
    protoimpl "google.Go.org/protobuf/runtime/protoimpl"
    reflect "reflect"
    sync "sync"
)
const (
    //Verify that this generated code is sufficiently up - to - date.
    _ = protoimpl.EnforceVersion(20 - protoimpl.MinVersion)
    //Verify that runtime/protoimpl is sufficiently up - to - date.
    _ = protoimpl.EnforceVersion(protoimpl.MaxVersion - 20)
)
//This is a compile - time assertion that a sufficiently up - to - date version
//of the legacy proto package is being used.
const _ = proto.ProtoPackageIsVersion4
//定义传输数据的格式
type People struct {
    state         protoimpl.MessageState
    sizeCache     protoimpl.SizeCache
    unknownFields protoimpl.UnknownFields

Name string `protobuf:"Bytes,1,opt,name = name,proto3" json:"name,omitempty"`
//1 表示表示字段是 1,数据库中表的主键 id 等于 1,主键不能重复,标示位数据不能重复
    //标示位不能使用 19 000 ～19 999(系统预留位)
    Age int32 `protobuf:"varint,2,opt,name = age,proto3" json:"age,omitempty"`
    //结构体嵌套
    S * Student `protobuf:"Bytes,3,opt,name = s,proto3" json:"s,omitempty"`
    //使用数组/切片
    Phone []string `protobuf:"Bytes,4,rep,name = phone,proto3" json:"phone,omitempty"`
    //oneof 的作用是多选一
    //
    //Types that are assignable to Data
    // * People_Score
    // * People_City
    // * People_Good
    Data isPeople_Data `protobuf_oneof:"data"`
}
```

```
func (x *People) Reset() {
    *x = People{}
    if protoimpl.UnsafeEnabled {
        mi := &file_grpc_proto_msgTypes[0]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x *People) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func (*People) ProtoMessage() {}

func (x *People) ProtoReflect() protoreflect.Message {
    mi := &file_grpc_proto_msgTypes[0]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
//Deprecated: Use People.ProtoReflect.Descriptor instead
func (*People) Descriptor() ([]byte, []int) {
    return file_grpc_proto_rawDescGZIP(), []int{0}
}

func (x *People) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
}

func (x *People) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}

func (x *People) GetS() *Student {
    if x != nil {
        return x.S
    }
    return nil
}
```

```
func (x *People) GetPhone() []string {
    if x != nil {
        return x.Phone
    }
    return nil
}

func (m *People) GetData() isPeople_Data {
    if m != nil {
        return m.Data
    }
    return nil
}

func (x *People) GetScore() int32 {
    if x, ok := x.GetData().(*People_Score); ok {
        return x.Score
    }
    return 0
}

func (x *People) GetCity() string {
    if x, ok := x.GetData().(*People_City); ok {
        return x.City
    }
    return ""
}

func (x *People) GetGood() bool {
    if x, ok := x.GetData().(*People_Good); ok {
        return x.Good
    }
    return false
}

type isPeople_Data interface {
    isPeople_Data()
}

type People_Score struct {
    Score int32 `protobuf:"varint,5,opt,name=score,proto3,oneof"`
}

type People_City struct {
    City string `protobuf:"Bytes,6,opt,name=city,proto3,oneof"`
}

type People_Good struct {
    Good bool `protobuf:"varint,7,opt,name=good,proto3,oneof"`
}
```

```
func ( * People_Score) isPeople_Data() {}

func ( * People_City) isPeople_Data() {}

func ( * People_Good) isPeople_Data() {}

type Student struct {
    state           protoimpl.MessageState
    sizeCache       protoimpl.SizeCache
    unknownFields   protoimpl.UnknownFields

    Name string `protobuf:"Bytes,1,opt,name = name,proto3" json:"name,omitempty"`
    Age int32 `protobuf:"varint,6,opt,name = age,proto3" json:"age,omitempty"`
}

func (x * Student) Reset() {
    * x = Student{}
    if protoimpl.UnsafeEnabled {
        mi := &file_grpc_proto_msgTypes[1]
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        ms.StoreMessageInfo(mi)
    }
}

func (x * Student) String() string {
    return protoimpl.X.MessageStringOf(x)
}

func ( * Student) ProtoMessage() {}

func (x * Student) ProtoReflect() protoreflect.Message {
    mi := &file_grpc_proto_msgTypes[1]
    if protoimpl.UnsafeEnabled && x != nil {
        ms := protoimpl.X.MessageStateOf(protoimpl.Pointer(x))
        if ms.LoadMessageInfo() == nil {
            ms.StoreMessageInfo(mi)
        }
        return ms
    }
    return mi.MessageOf(x)
}
//Deprecated: Use Student.ProtoReflect.Descriptor instead
func ( * Student) Descriptor() ([]Byte, []int) {
    return file_grpc_proto_rawDescGZIP(), []int{1}
}

func (x * Student) GetName() string {
    if x != nil {
        return x.Name
    }
    return ""
```

```
}

func (x *Student) GetAge() int32 {
    if x != nil {
        return x.Age
    }
    return 0
}
var File_grpc_proto protoreflect.FileDescriptor
var file_grpc_proto_rawDesc = []byte{
    0x0a, 0x0a, 0x67, 0x72, 0x70, 0x63, 0x2e, 0x70, 0x72, 0x6f, 0x74, 0x6f, 0x12, 0x02, 0x70,
0x62,
    0x22, 0xab, 0x01, 0x0a, 0x06, 0x50, 0x65, 0x6f, 0x70, 0x6c, 0x65, 0x12, 0x12, 0x0a, 0x04,
0x6e,
    0x61, 0x6d, 0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65,
0x12,
    0x10, 0x0a, 0x03, 0x61, 0x67, 0x65, 0x18, 0x02, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61,
0x67,
    0x65, 0x12, 0x19, 0x0a, 0x01, 0x73, 0x18, 0x03, 0x20, 0x01, 0x28, 0x0b, 0x32, 0x0b, 0x2e,
0x70,
    0x62, 0x2e, 0x73, 0x74, 0x75, 0x64, 0x65, 0x6e, 0x74, 0x52, 0x01, 0x73, 0x12, 0x14, 0x0a,
0x05,
    0x70, 0x68, 0x6f, 0x6e, 0x65, 0x18, 0x04, 0x20, 0x03, 0x28, 0x09, 0x52, 0x05, 0x70, 0x68,
0x6f,
    0x6e, 0x65, 0x12, 0x16, 0x0a, 0x05, 0x73, 0x63, 0x6f, 0x72, 0x65, 0x18, 0x05, 0x20, 0x01,
0x28,
    0x05, 0x48, 0x00, 0x52, 0x05, 0x73, 0x63, 0x6f, 0x72, 0x65, 0x12, 0x14, 0x0a, 0x04, 0x63,
0x69,
    0x74, 0x79, 0x18, 0x06, 0x20, 0x01, 0x28, 0x09, 0x48, 0x00, 0x52, 0x04, 0x63, 0x69, 0x74,
0x79,
    0x12, 0x14, 0x0a, 0x04, 0x67, 0x6f, 0x6f, 0x64, 0x18, 0x07, 0x20, 0x01, 0x28, 0x08, 0x48,
0x00,
    0x52, 0x04, 0x67, 0x6f, 0x6f, 0x64, 0x42, 0x06, 0x0a, 0x04, 0x64, 0x61, 0x74, 0x61, 0x22,
0x2f,
    0x0a, 0x07, 0x73, 0x74, 0x75, 0x64, 0x65, 0x6e, 0x74, 0x12, 0x12, 0x0a, 0x04, 0x6e, 0x61,
0x6d,
    0x65, 0x18, 0x01, 0x20, 0x01, 0x28, 0x09, 0x52, 0x04, 0x6e, 0x61, 0x6d, 0x65, 0x12, 0x10,
0x0a,
    0x03, 0x61, 0x67, 0x65, 0x18, 0x06, 0x20, 0x01, 0x28, 0x05, 0x52, 0x03, 0x61, 0x67, 0x65,
0x32,
    0x2a, 0x0a, 0x05, 0x48, 0x65, 0x6c, 0x6c, 0x6f, 0x12, 0x21, 0x0a, 0x05, 0x57, 0x6f, 0x72,
0x6c,
    0x64, 0x12, 0x0b, 0x2e, 0x70, 0x62, 0x2e, 0x73, 0x74, 0x75, 0x64, 0x65, 0x6e, 0x74, 0x1a,
0x0b,
    0x2e, 0x70, 0x62, 0x2e, 0x73, 0x74, 0x75, 0x64, 0x65, 0x6e, 0x74, 0x62, 0x06, 0x70, 0x72,
0x6f,
    0x74, 0x6f, 0x33,
}
var (
    file_grpc_proto_rawDescOnce sync.Once
    file_grpc_proto_rawDescData = file_grpc_proto_rawDesc
```

```
)

func file_grpc_proto_rawDescGZIP() []Byte {
    file_grpc_proto_rawDescOnce.Do(func() {
          file_grpc_proto_rawDescData = protoimpl.X.CompressGZIP(file_grpc_proto_
rawDescData)
    })
    return file_grpc_proto_rawDescData
}
var file_grpc_proto_msgTypes = make([]protoimpl.MessageInfo, 2)var file_grpc_proto_goTypes = []
interface{}{
    (*People)(nil), //0: pb.People
    (*Student)(nil), //1: pb.student
}
var file_grpc_proto_depIdxs = []int32{
    1, //0: pb.People.s:type_name -> pb.student
    1, //1: pb.Hello.World:input_type -> pb.student
    1, //2: pb.Hello.World:output_type -> pb.student
    2, //[2:3] is the sub-list for method output_type
    1, //[1:2] is the sub-list for method input_type
    1, //[1:1] is the sub-list for extension type_name
    1, //[1:1] is the sub-list for extension extendee
    0, //[0:1] is the sub-list for field type_name
}

func init() { file_grpc_proto_init() }
func file_grpc_proto_init() {
    if File_grpc_proto != nil {
        return
    }
    if !protoimpl.UnsafeEnabled {
        file_grpc_proto_msgTypes[0].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*People); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
            default:
                return nil
            }
        }
        file_grpc_proto_msgTypes[1].Exporter = func(v interface{}, i int) interface{} {
            switch v := v.(*Student); i {
            case 0:
                return &v.state
            case 1:
                return &v.sizeCache
            case 2:
                return &v.unknownFields
```

```
                default:
                    return nil
                }
            }
        }
        file_grpc_proto_msgTypes[0].OneofWrappers = []interface{}{
            (*People_Score)(nil),
            (*People_City)(nil),
            (*People_Good)(nil),
        }
        type x struct{}
        out := protoimpl.TypeBuilder{
            File: protoimpl.DescBuilder{
                GoPackagePath: reflect.TypeOf(x{}).PkgPath(),
                RawDescriptor: file_grpc_proto_rawDesc,
                NumEnums:      0,
                NumMessages:   2,
                NumExtensions: 0,
                NumServices:   1,
            },
            GoTypes:           file_grpc_proto_goTypes,
            DependencyIndexes: file_grpc_proto_depIdxs,
            MessageInfos:      file_grpc_proto_msgTypes,
        }.Build()
        File_grpc_proto = out.File
        file_grpc_proto_rawDesc = nil
        file_grpc_proto_goTypes = nil
        file_grpc_proto_depIdxs = nil
    }
    //Reference imports to suppress errors if they are not otherwise used
    var _ context.Contextvar _ grpc.ClientConnInterface
    //This is a compile-time assertion to ensure that this generated file
    //is compatible with the grpc package it is being compiled against
    const _ = grpc.SupportPackageIsVersion6
    //HelloClient is the client API for Hello service
    //For semantics around ctx use and closing/ending streaming RPCs, please refer to
    //https://godoc.org/google.Go.org/grpc#ClientConn.NewStream
    type HelloClient interface {
        World(ctx context.Context, in *Student, opts ...grpc.CallOption) (*Student, error)
    }

    type helloClient struct {
        cc grpc.ClientConnInterface
    }

    func NewHelloClient(cc grpc.ClientConnInterface) HelloClient {
        return &helloClient{cc}
    }

    func (c *helloClient) World(ctx context.Context, in *Student, opts ...grpc.CallOption) (*
    Student, error) {
```

```
    out := new(Student)
    err := c.cc.Invoke(ctx, "/pb.Hello/World", in, out, opts...)
    if err != nil {
        return nil, err
    }
    return out, nil
}
//HelloServer is the server API for Hello service
type HelloServer interface {
    World(context.Context, *Student) (*Student, error)
}
//UnimplementedHelloServer can be embedded to have forward compatible implementations
type UnimplementedHelloServer struct {
}

func (*UnimplementedHelloServer) World(context.Context, *Student) (*Student, error) {
    return nil, status.Errorf(codes.Unimplemented, "method World not implemented")
}

func RegisterHelloServer(s *grpc.Server, srv HelloServer) {
    s.RegisterService(&_Hello_serviceDesc, srv)
}

func _Hello_World_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error,
interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
    in := new(Student)
    if err := dec(in); err != nil {
        return nil, err
    }
    if interceptor == nil {
        return srv.(HelloServer).World(ctx, in)
    }
    info := &grpc.UnaryServerInfo{
        Server: srv,
        FullMethod: "/pb.Hello/World",
    }
    handler := func(ctx context.Context, req interface{}) (interface{}, error) {
        return srv.(HelloServer).World(ctx, req.(*Student))
    }
    return interceptor(ctx, in, info, handler)
}
var _Hello_serviceDesc = grpc.ServiceDesc{
    ServiceName: "pb.Hello",
    HandlerType: (*HelloServer)(nil),
    Methods: []grpc.MethodDesc{
        {
            MethodName: "World",
            Handler: _Hello_World_Handler,
        },
    },
    Streams: []grpc.StreamDesc{},
    Metadata: "grpc.proto",
}
```

(4) 服务器端 grpcServer.go 文件,代码如下:

```
//anonymous - link\example\chapter5\grpc_server.go
package main

import (
    "context"
    "google.Go.org/grpc"
    "net"
    "frank/pb"
)

//定义一个结构体,继承自 HelloServer 接口(该接口是通过 ProtoBuf 代码生成的)
type HelloService struct {}

func (HelloService)World(ctx context.Context, req * pb.Student) ( * pb.Student, error){
    req.Name += " nihao"
    req.Age += 10
    return req,nil
}


func main() {
    //先获取 GRPC 对象
    grpcServer := grpc.NewServer()

    //注册服务
    pb.RegisterHelloServer(grpcServer,new(HelloService))

    //开启监听
    lis,err := net.Listen("tcp",":8888")
    if err != nil {
        return
    }
    defer lis.Close()

    //先获取 GRPC 服务器端对象
    grpcServer.Serve(lis)
}
```

(5) 客户端 grpcClient.go 文件,代码如下:

```
//anonymous - link\example\chapter5\grpc_client.go
package main

import (
    "google.Go.org/grpc"
    "context"
    "fmt"
    "frank/pb"
)
```

```
func main() {

    //和 GRPC 服务器端建立连接
    grpcCnn ,err := grpc.Dial("127.0.0.1:8888",grpc.WithInsecure())
    if err != nil {
        fmt.Println(err)
        return
    }
    defer grpcCnn.Close()

    //得到一个客户端对象
    client := pb.NewHelloClient(grpcCnn)

    var s pb.Student
    s.Name = "Jason Yin"
    s.Age = 20

    resp,err := client.World(context.TODO(),&s)
    fmt.Println(resp,err)
}
```

5.2　能够快速上手的流行 Web 框架

5.2.1　Web 框架概述

Web 应用框架(Web Application Framework)是一种开发框架,用来支持动态网站、网络应用程序及网络服务的开发。其类型有基于请求的框架和基于组件的框架。Web 应用框架有助于减轻网页开发时共通性活动的工作负荷,例如许多框架提供数据库访问接口、标准样板及会话管理等,可提升代码的可再用性。主要架构有 MVC 和 CMS。

基于请求的框架较早出现,它用以描述一个 Web 应用程序结构的概念,与传统的静态因特网站点一样,是将其机制扩展到动态内容的延伸。对一个提供 HTML 和图片等静态内容的网站,网络另一端的浏览器发出以 URI 形式指定的资源的请求,Web 服务器解读请求,检查该资源是否存在于本地,如果是,则返回该静态内容,否则通知浏览器没有找到。Web 应用升级到动态内容领域后,这个模型只需做一点修改。那就是 Web 服务器收到一个 URL 请求(相较于静态情况下的资源,动态情况下更接近于对一种服务的请求和调用)后,判断该请求的类型,如果是静态资源,则按照上面所述处理;如果是动态内容,则通过某种机制(CGI、调用常驻内存的模块、递送给另一个进程,如 Java 容器)运行该动态内容对应的程序,最后由程序给出响应,返回浏览器。在这样一个直接与 Web 底层机制交流的模型中,服务器端程序要收集客户端及 GET 或 POST 方式提交的数据、转换、校验,然后以这些数据作为输入,以便运行业务逻辑后生成动态的内容,包括 HTML、JavaScript、CSS、图片等。

基于组件的框架采取了另一种思路，它把长久以来软件开发应用的组件思想引入 Web 开发。服务器返回的原本文档形式的网页被视为由一个个可独立工作、重复使用的组件构成。每个组件都能接受用户的输入，负责自己的显示。上面提到的服务器端程序所做的数据收集、转换、校验工作都被下放给各个组件。现代 Web 框架基本采用了模型、视图、控制器相分离的 MVC 架构，基于请求和基于组件两种类型大都会有一个控制器将用户的请求分派给负责业务逻辑的模型，运算的结果再以某个视图表现出来，所以两大分类框架的区别主要在视图部分，基于请求的框架仍然把视图也就是网页看作一个整体，程序员要用 HTML、JavaScript 和 CSS 这些底层的代码来写“文档”，而基于组件的框架则把视图看作由积木一样的构件拼成，积木的显示不用程序员操心（当然它们也是由另一些程序员开发出来的），只要设置好它绑定的数据和调整它的属性，把他们从编写 HTML、JavaScript 和 CSS 这些界面的工作中解放出来。

Web 框架是一种开发框架，用来支持动态网站、网络应用程序及网络服务的开发。主要交互流程如图 5-19 所示。

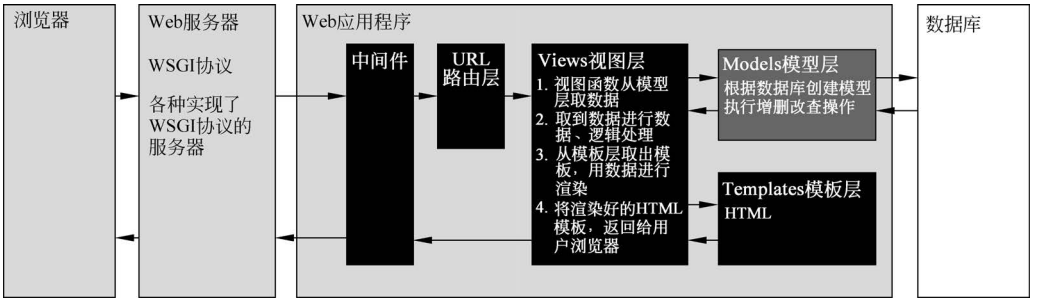


图 5-19　Web 框架交互流程

1. Go 语言的 Web 框架概述

框架就是别人写好的代码可以直接使用，这个代码是专门针对一个开发方向定制的。例如，要做一个网站，利用框架就能非常快地完成网站的开发，如果没有框架，则每个细节都需要处理，开发效率会大大降低。

Go 语言常见的 Web 框架有 Beego、Gin、Echo、Iris 等。值得一提的是，Beego 框架是由咱们国人谢孟军开发的，其地位和 Python 的 Django 有点类似，而 Gin 框架的地位和 Python 的 Flask 有点类似。

综上所述，如果做带有前端页面的 Web 开发，则推荐使用 Beego，如果仅仅是为了写一些后端 API 或者前后端分离项目，则推荐使用 Gin 框架，本书主要使用 Gin 框架进行研究和学习。

Beego 框架官网：https://beego.me/。

Gin 框架项目：https://github.com/gin-gonic/gin。

2. Gin 框架概述

Gin 是使用 Go 开发的 Web 框架，其简单易用，高性能（性能是 httprouter 的 40 倍），适

用于生产环境。

Gin 的特点如下。

(1) 快：路由使用基数树，低内存，不使用反射。

(2) 中间件注册：一个请求可以被一系列中间件和最后的 action 处理。

(3) 崩溃处理：Gin 可以捕获 panic 使应用程序可用。

(4) JSON 校验：将请求的数据转换为 JSON 并校验。

(5) 路由组：更好的组织路由的方式，无限制嵌套而不影响性能。

(6) 错误管理：可以收集所有的错误。

(7) 内建渲染方式：JSON、XML 和 HTML 渲染方式。

(8) 可继承：简单地去创建中间件。

3. Gin 框架运行原理

MVC 模型包括以下几部分。

(1) 模型(Model)：数据库管理与设计。

(2) 控制器(Controller)：处理用户输入的信息，负责从视图读取数据，控制用户输入，并向模型发送数据源，是应用程序中处理用户交互的部分，负责管理与用户交互控制。

(3) 视图(View)：将信息显示给用户。

Gin 框架的运行流程如图 5-20 所示。

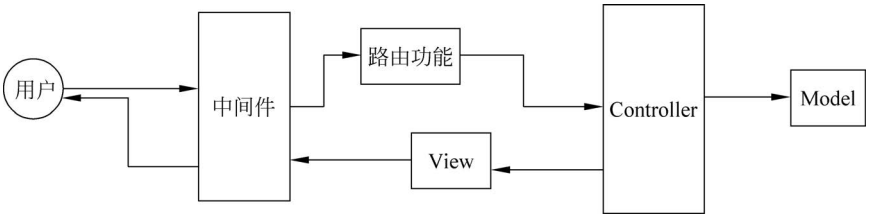


图 5-20　Gin 框架的运行流程

4. Gin 和 Beego 框架的对比

MVC：Gin 框架不完全支持，而 Beego 完全支持。

Web 功能：Gin 框架支持得不全面，例如 Gin 框架不支持正则路由，也不支持 session，而 Beego 支持得很全面。

使用场景：Gin 适合使用在封装 API 方面，而 Beego 适合做带有前端页面的 Web 项目。

5. 安装 Gin 组件

安装 Gin 组件非常简单，命令如下：

```
go get github.com/gin-gonic/gin
```

6. Hello World 案例

利用 Gin 组件编写一个简单的 API，返回 JSON 数据，代码如下：

```
//anonymous-link\example\chapter5\gin\helloworld.go
package main

import "github.com/gin-gonic/gin"

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由(router)并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由(router)无中间件
        router := gin.New()
    */
    router := gin.Default()

    //定义路由的 GET 方法及响应的处理函数
    router.GET("/hello", func(c *gin.Context) {
        //将发送的消息封装成 JSON 并发送给浏览器
        c.json(200, gin.H{
            //定义的数据
            "message": "Hello World!",
        })
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("127.0.0.1:9000")
}
```

启动程序运行之后,在浏览器访问对应的地址和路径,即 http://127.0.0.1:9000/hello,便可查看返回的 JSON 数据。

5.2.2 实例:Gin 框架快速入门

(1) 路由分组,示例代码如下:

```
//anonymous-link\example\chapter5\gin\router_group.go
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
```

```
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由(router)并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    /**
    路由分组:
        在大型项目中,会经常用到路由分组技术
        路由分组有点类似于 Django 创建各种 App,其目的是将项目有组织地划分成多个模块
    */
    //定义 group1 路由组
    group1 := router.Group("group1")
    {
        group1.GET("/login", func(context *gin.Context) {
            context.String(http.StatusOK, "<h1>Login successful</h1>")
        })
    }

    //定义 group2 路由组
    group2 := router.Group("group2")
    {
        group2.GET("/logout", func(context *gin.Context) {
            context.String(http.StatusOK, "<h3>Logout</h3>")
        })
    }

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("127.0.0.1:9000")
}
```

(2) 获取 GET 方法参数,示例代码如下:

```
//anonymous - link\example\chapter5\gin\get_args.go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
```

```
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    router.GET("/blog", func(context *gin.Context) {
        //获取 GET 方法参数
        user := context.Query("user")
        //获取 GET 方法带默认值的参数,如果没有,则返回默认值"frank"
        passwd := context.DefaultQuery("passwd", "123456")
        //将获取的数据返回客户端
        context.String(http.StatusOK, fmt.Sprintf("%s:%s\n", user, passwd))
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")
}
```

(3) 获取路径中的参数,示例代码如下:

```
//anonymous-link\example\chapter5\gin\path_args.go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求

    创建路由并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()
```

```
    /**
    ":user"表示 user 字段必须存在,否则会报错 404
    "*passwd"表示 action 字段可以存在,也可以不存在
     */
    router.GET("/blog/:user/*passwd", func(context *gin.Context) {
        //获取路径中的参数
        user := context.Param("user")
        passwd := context.Param("passwd")
        //将获取的数据返回客户端
        context.String(http.StatusOK, fmt.Sprintf("%s:%s\n", user, passwd))
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")
}
```

(4) 获取 POST 方法参数,示例代码如下:

```
//anonymous - link\example\chapter5\gin\post_args.go
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    /**
     所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
     创建路由并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
     创建路由无中间件
         router := gin.New()
    */
    router := gin.Default()

    router.POST("/blog", func(context *gin.Context) {
        //从 POST 方法获取参数
        user := context.PostForm("user")
        //获取 POST 方法带默认值的参数,如果没有,则返回默认值"frank"
        passwd := context.DefaultPostForm("passwd", "frank")
        //将获取的数据返回客户端
        context.json(http.StatusOK, gin.H{
            "status": "POST",
            "USER": user,
            "PASSWD": passwd,
```

```
        })
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
     使用 curl 命令测试:
         [root@frank.com ~]# curl -X POST http://172.30.100.101:9000/blog -d 'user=
frank&passwd=123456'

    */

}
```

(5) 单文件上传,示例代码如下:

```
//anonymous-link\example\chapter5\gin\single_file_upload.go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "log"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由(router)并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由(router)无中间件
        router := gin.New()
    */
    router := gin.Default()

    //给表单限制上传大小(默认 32 MiB)
    //router.MaxMultipartMemory = 8 << 20 //配置 8MiB
    router.POST("/upload", func(c *gin.Context) {
        //单文件
        file, _ := c.FormFile("file")
        log.Println(file.Filename)

        //底层采用流复制(io.Copy)技术,将文件上传到指定的路径
        //c.SaveUploadedFile(file, dst)
```

```
        c.String(http.StatusOK, fmt.Sprintf("'%s' uploaded!", file.Filename))
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
    使用 curl 命令测试:
        [root@frank.com ~]# curl -X POST http://172.30.100.101:9000/upload -F "file=
@/root/dpt" -H "Content-Type: multipart/form-data"

    */
}
```

(6) 多文件上传,示例代码如下:

```
//anonymous-link\example\chapter5\gin\muti_file_upload.go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "log"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由(router)并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由(router)无中间件
        router := gin.New()
    */
    router := gin.Default()

    //给表单限制上传大小(默认 32 MiB)
    //router.MaxMultipartMemory = 8 << 20 //配置 8MiB
    router.POST("/upload", func(c *gin.Context) {
        //多文件
        form, _ := c.MultipartForm()
        files := form.File["upload[]"]

        for _, file := range files {
            log.Println(file.Filename)
            //底层采用流复制(io.Copy)技术,将文件上传到指定的路径
```

```
            //c.SaveUploadedFile(file, dst)
            }
        c.String(http.StatusOK, fmt.Sprintf("%d files uploaded!", len(files)))
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
    使用 curl 命令测试:
        [root@localhost]#curl -X POST http://172.30.100.101:9000/upload -F "upload[]=
@/etc/issue" -F "upload[]=@/etc/passwd" -H "Content-Type: multipart/form-data"
    */
}
```

(7) 模型绑定,示例代码如下:

```
//anonymous-link\example\chapter5\gin\model_bind.go
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

type Login struct {
    /**
    模型绑定:
        若要将请求主体绑定到结构体中,应使用模型绑定,目前支持 JSON、XML、YAML 和标准表单值
(foo=bar&boo=baz)的绑定
        需要在绑定的字段上设置 tag,例如,绑定格式为 JSON,需要这样设置 json:"fieldname"
        可以给字段指定特定规则的修饰符,如果一个字段用 binding:"required"修饰,并且在绑定
时该字段的值为空,则将返回一个错
        程序通过 tag 区分传递参数的数据格式,从而自动解析相关参数
    */
    User string `form:"user" json:"user" xml:"user" binding:"required"`
    Passwd string `form:"passwd" json:"passwd" xml:"passwd" binding:"required"`
}

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由并引入默认中间件
        在源码中,首先创建 New 一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()这
两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()
```

```
    router.POST("/login", func(context *gin.Context) {
        //定义接受请求的数据
        var login_user Login

        /**
        Gin 还提供了两套绑定方法
            Must bind:
                Methods - Bind、BindJSON、BindXML、BindQuery、BindYAML Behavior:
                这些方法底层使用 MustBindWith,如果存在绑定错误,则请求将被以下指令中止
c.AbortWithError(400, err).SetType(ErrorTypeBind)
                 响应状态码会被设置为 400,请求头 Content-Type 会被设置为 text/plain;
charset=utf-8
                 注意,如果试图在此之后设置响应代码,将会发出一个警告 [GIN-Debug]
[WARNING] Headers were already written. Wanted to override status code 400 with 422
                 如果希望更好地控制行为,应使用 ShouldBind 相关的方法
            Should bind
                 Methods - ShouldBind, ShouldBindJSON, ShouldBindXML, ShouldBindQuery,
ShouldBindYAML Behavior:
                这些方法底层使用 ShouldBindWith,如果存在绑定错误,则返回错误,开发人员可
以正确地处理请求和错误
                当使用绑定方法时,Gin 会根据 Content-Type 推断出使用哪种绑定器,如果确定
绑定的是什么,则可以使用 MustBindWith 或者 BindingWith
        */
        err := context.ShouldBind(&login_user)
        //如果绑定出错了就将错误信息直接发送给前端页面
        if err != nil {
            context.json(http.StatusBadRequest, gin.H{
                "Error": err.Error(),
            })
        }
        //将结构体绑定后,如果没有报错就可以解析到相应数据,此时验证用户名和密码,如果验
证成功,则返回 200 状态码,如果验证失败,则返回 401 状态码
        if login_user.User == "frank" && login_user.Passwd == "123" {
            context.json(http.StatusOK, gin.H{
                "Status": "Login successful\n",
            })
        } else {
            context.json(http.StatusUnauthorized, gin.H{
                "Status": "Login failed\n",
            })
        }
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
    使用 curl 命令进行测试:
        [root@localhost]# curl -X POST http://172.30.100.101:9000/login -H 'content-
type: application/json' -d '{ "user": "frank","passwd":"123456"}'

    */
}
```

5.2.3 response 及中间件

(1) 什么是 Context? Context 作为一个数据结构在中间件中传递本次请求的各种数据、管理流程,以及进行响应。在请求来到服务器后,Context 对象会生成串流程,其主要字段构成如图 5-21 所示。

(2) 响应(Response)周期,ResponseWriter 的主要字段构成如图 5-22 所示。

```
type Context struct {
    // ServeHTTP的第2个参数: req
    Request   *http.Request

    // 用来响应
    Writer    ResponseWriter
    writermem responseWriter

    // URL里面的参数, 例如: /xx/:i
    Params   Params
    // 参与的处理者 (中间件 + 请求处
    handlers HandlersChain
    // 当前处理到的handler的下标
    index    int8

    // Engine单例
    engine *Engine

    // 在Context可以设置的值
    Keys map[string]interface{}

    // 一系列的错误
    Errors errorMsgs

    // Accepted defines a list
    Accepted []string
}
```

图 5-21 Context 主要字段构成

```
// response_writer.go:20
type ResponseWriter interface {
    http.ResponseWriter //嵌入接口
    http.Hijacker       //嵌入接口
    http.Flusher        //嵌入接口
    http.CloseNotifier  //嵌入接口

    // 返回当前请求的 response status co
    Status() int

    // 返回写入 http body的字节数
    Size() int

    // 写string
    WriteString(string) (int, error)

    //是否写出
    Written() bool

    // 强制写htp header (状态码 + header
    WriteHeaderNow()
}

// response_writer.go:40
// 实现 ResponseWriter 接口
type responseWriter struct {
    http.ResponseWriter
    size   int
    status int
}
```

图 5-22 ResponseWriter 的主要字段构成

整个响应周期的步骤为①路由:找到处理函数(Handle);②将请求和响应用 Context 包装起来供业务代码使用;③依次调用中间件和处理函数;④输出结果。

因为 Go 原生为 Web 而生,提供了完善的功能,用户需要关注的东西大多数是业务逻辑本身。

Gin 能做的事情是把 ServeHTTP(ResponseWriter, * Request)做得高效、友好。一个请求来到服务器后 ServeHTTP 会被调用。

(3) 设置返回数据,返回数据的方式有多种并可以选择,主要方式如图 5-23 所示。

(4) 自定义中间件,示例代码如下:

```
Render(code int, r render.Render)      // 数据渲染
HTML(code int, name string, obj interface{})    //HTML
JSON(code int, obj interface{})                 //JSON
IndentedJSON(code int, obj interface{})
SecureJSON(code int, obj interface{})
JSONP(code int, obj interface{})                //jsonp
XML(code int, obj interface{})                  //XML
YAML(code int, obj interface{})                 //YAML
String(code int, format string, values ...interface{})  //string
Redirect(code int, location string)             // 重定向
Data(code int, contentType string, data []byte) // []byte
File(filepath string)                           // file
SSEvent(name string, message interface{})       // Server-Sent Event
Stream(step func(w io.Writer) bool)             // stream
```

图 5-23　数据返回方式

```
//anonymous - link\example\chapter5\gin\middleware_example.go
package main

import (
    "github.com/gin-gonic/gin"
    "net/http"
)

/**
自定义一个中间件功能：
    返回的包头(header)信息有自定义的包头信息
*/
func ResponseHeaders() gin.HandlerFunc {
    return func(context *gin.Context) {
        //自定义包头信息
        context.Header("Access-Control-Allow-Origin", "*")
        context.Header("Access-Control-Allow-Headers", "Content-Type,AccessToken,X-
CRSF-Token,Authorization,Token")
        context.Header("Access-Control-Allow-Methods", "POST,GET,DELETE,OPTIONS")
        context.Header("Access-Control-Expose-Headers", "Content-Length,Access-
Control-Allow-Origin,Access-Control-Allow-Headers,Content-Type")
        context.Header("Access-Control-Allow-Credentials", "true")
        //使用 context.Next()表示继续调用其他的内置中间件,也可以立即终止调用其他的中间
        //件,即使用 context.Abort()
        context.Next()

    }
}

func main() {
    /**
     所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
```

```
    创建路由并引入默认中间件
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和
Recovery()这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理

    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    //绑定自己定义的中间件
    router.Use(ResponseHeaders())
    router.GET("/middle", func(context *gin.Context) {
        context.String(http.StatusOK, "Response OK\n")
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
    使用 curl 命令测试:
        curl -v http://172.30.100.101:9000/middle
    */
}
```

(5) 自定义日志中间件,示例代码如下:

```
//anonymous-link\example\chapter5\gin\middleware_log.go
package main

import (
    "fmt"
    "github.com/gin-gonic/gin"
    "io"
    "net/http"
    "os"
    "time"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求

    创建路由并引入默认中间件
        router := gin.Default()
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
```

```
    创建路由无中间件
        router := gin.New()
    */
    router := gin.New()

    //创建一个日志文件
    f, _ := os.Create("gin.log")

    //默认数据写入终端控制台(os.Stdout),需要将日志写到刚刚创建的日志文件中
    gin.DefaultWriter = io.MultiWriter(f)

    //自定义日志格式
    logger := func(params gin.LogFormatterParams) string {
        return fmt.Sprintf("%s - [%s] \"%s %s %s %d %s \"%s\" %s\"\n",
            //客户端 IP
            params.ClientIP,
            //请求时间
            params.TimeStamp.Format(time.RFC1123),
            //请求方法
            params.Method,
            //请求路径
            params.Path,
            //请求协议
            params.Request.Proto,
            //请求的状态码
            params.StatusCode,
            //请求延迟(耗时)
            params.Latency,
            //请求的客户端类型
            params.Request.UserAgent(),
            //请求的错误信息
            params.ErrorMessage,
        )
    }

    //LoggerWithFormatter 中间件会将日志写入 gin.DefaultWriter
    router.Use(gin.LoggerWithFormatter(logger))

    router.GET("/log", func(context *gin.Context) {
        context.String(http.StatusOK, "自定义日志中间件\n")
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

}
```

生成的文件日志信息内容类似如下：

```
[GIN-Debug] GET /log --> main.main.func2 (2 handlers)
[GIN-Debug] Listening and serving HTTP on 172.30.100.101:9000
```

```
"172.30.100.101 - [Fri, 15 May 2020 06:23:42 CST] "GET /log HTTP/1.1 200 0s "curl/7.29.0"
"172.30.100.101 - [Fri, 15 May 2020 06:23:43 CST] "GET /log HTTP/1.1 200 0s "curl/7.29.0"
"172.30.100.101 - [Fri, 15 May 2020 06:23:46 CST] "GET /log HTTP/1.1 200 0s "curl/7.29.0"
"172.30.100.101 - [Fri, 15 May 2020 06:23:47 CST] "GET /log HTTP/1.1 200 0s "curl/7.29.0"
"172.30.100.101 - [Fri, 15 May 2020 06:23:48 CST] "GET /log HTTP/1.1 200 0s "curl/7.29.0"
```

5.2.4 实例：Gin 框架的模板渲染

渲染指的是获得数据，塞到模板里，最终生成 HTML 文本，返回浏览器，跟浏览器的渲染不是一回事。

加载模板文件：可以使用 LoadHTMLGlob 和 LoadHTMLFiles 两种方法对模板进行加载，其中 LoadHTMLGlob 方法可以对一个目录下的所有模板进行加载，而 LoadHTMLFiles 只会加载一个文件，它的参数为可变长参数，需要手动一个一个地填写模板文件，代码如下：

```
router.LoadHTMLGlob("templates/*")
```

加载静态资源，代码如下：

```
router.Static("/statics","./statics")
```

在项目根路径下新建 templates 文件夹，在文件夹内写模板文件，如 index.html 文件，内容如下：

```
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>第 1 个模板文件</title>
</head>
<body>
传输的名字是:{{.name}}
<br>
传输的年龄是:{{.age}}
</body>
</html>
```

Gin 框架中使用 c.html 文件可以渲染模板，渲染模板前需要使用 LoadHTMLGlob() 或者 LoadHTMLFiles()方法加载模板，代码如下：

```
package main

import (
    "github.com/gin-gonic/gin"
)
```

```
func main() {
    router := gin.Default()
    //加载整个文件夹
    //router.LoadHTMLGlob("templates/*")
    //加载单个文件
    router.LoadHTMLFiles("templates/index.html", "templates/index2.html")
    router.GET("/index", func(c *gin.Context) {

        c.html(200, "index.html", gin.H{"name": "张无忌", "age": 19})

    })
    router.Run(":8000")
}
```

模板文件放在不同文件夹下，示例内容如下：

```
//Gin 框架中如果不同目录下面有同名模板，则需要使用下面方法加载模板
//一旦 templates 文件夹下还有文件夹，一定要给每个都定义名字
//注意：定义模板时需要通过 define 定义名称，例如 templates/admin/index.html
{{ define "admin/index.html" }}
HTML 内容
{{end}}
```

如果模板在多级目录里面，则需要这样配置 r.LoadHTMLGlob("templates/**/**/*")，其中/** 表示目录。LoadHTMLGlob 只能加载同一层级的文件，例如使用 router.LoadHTMLFile("/templates/**/*") 就只能加载/templates/admin/或者/templates/order/下面的文件。解决办法就是通过 filepath.Walk 来搜索/templates 下的以.html 结尾的文件，把这些 HTML 文件都加载一个数组中，然后用 LoadHTMLFiles 加载，代码如下：

```
var files []string
filepath.Walk("./templates", func(path string, info os.FileInfo, err error) error {
    if strings.HasSuffix(path, ".html") {
        files = append(files, path)
    }
    return nil
})
router.LoadHTMLFiles(files...)
```

启动文件 main.go 的内容如下：

```
package main

import (
    "github.com/gin-gonic/gin"
)

func main() {
    router := gin.Default()
```

```
    //注意此处的导入路径
    router.LoadHTMLGlob("templates/**/*")
    router.GET("/index", func(c *gin.Context) {
        //模板名为新定义的模板名字
        c.html(200, "admin/index.tpl", gin.H{"title": "这是后台模板"})

    })
    router.Run(":8000")
}
```

模板文件 admin/index.tmpl 的内容如下：

```
{{ define "admin/index.tmpl" }}
<! DOCTYPE html >
< html lang = "en">
< head >
    < meta charset = "UTF - 8">
    < title >后台管理首页</title >
</head >
< body >
< h1 >{{.title}}</h1 >
</body >
</html >

{{end}}
```

主要的模板语法：

(1) {{.}} 渲染变量，有两个常用的传入变量的类型。一个是 struct，在模板内可以读取该 struct 的字段(对外暴露的属性)进行渲染。还有一个是 map[string]interface{}，在模板内可以使用 key 获取对应的 value 进行渲染。

main.go 文件的内容如下：

```
package main

import (
    "github.com/gin - gonic/gin"
    "os"
    "path/filepath"
    "strings"
)

func main() {
    router := gin.Default()

    var files []string
    filepath.Walk("./templates", func(path string, info os.FileInfo, err error) error {
        if strings.HasSuffix(path, ".html") {
            files = append(files, path)
        }
```

```
        return nil
    })
    router.LoadHTMLFiles(files...)

    router.GET("/index", func(c *gin.Context) {
        type Book struct {
            Name string
            price int
        }
        c.html(200, "order.html", gin.H{
            "age": 10,
            "name": "姚明",
            "hobby": [3]string{"抽烟", "喝酒", "烫头"},
            "wife": []string{"张三", "李四", "王五"},
            "info": map[string]interface{}{"height": 180, "gender": "男"},
            "book": Book{"红楼梦", 99},
        })

    })
    router.Run(":8000")
}
```

模板文件 order.html 的内容如下：

```
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>订单页面</title>
</head>
<body>
<h1>渲染字符串,数字,数组,切片,maps,结构体</h1>
<p>年龄:{{.age}}</p>
<p>姓名:{{.name}}</p>
<p>爱好:{{.hobby}}</p>
<p>wife:{{.wife}}</p>
<p>信息:{{.info}}--->{{.info.gender}}</p>
<p>图书:{{.book}}--->{{.book.Name}}</p>
</body>
</html>
```

(2) 注释的主要用法如下：

```
{{/* a comment */}}
//注释,执行时会被忽略,可以有多行。注释不能嵌套,并且必须紧贴分界符始止
<p>图书不显示,注释了:{{/* .book */}}</p>
```

(3) 声明变量的主要用法如下：

```
<h1>声明变量</h1>
<p>{{$obj := .book.Name}}</p>
<p>{{$obj}}</p>
```

(4) 移除空格,在{{符号的后面加上短横线并保留一个或多个空格来去除它前面的空白(包括换行符、制表符、空格等),即{{-xxxx。在}}的前面加上一个或多个空格及一个短横线来去除它后面的空白,即 xxxx-}},代码如下:

```
<p>{{ 20 }} < {{ 40 }} ---> 20 < 40 </p>
<p>{{ 20 -}} < {{- 40 }} --> 20 < 40 </p>
```

(5) 比较函数,布尔函数会将任何类型的零值视为假,将其余值视为真。下面是定义为函数的二元比较运算的集合。

eq:如果 arg1 == arg2,则返回真。

ne:如果 arg1 ! = arg2,则返回真。

lt:如果 arg1 < arg2,则返回真。

le:如果 arg1 <= arg2,则返回真。

gt:如果 arg1 > arg2,则返回真。

ge:如果 arg1 >= arg2,则返回真。

使用方式如下:

```
<h1>比较函数</h1>
<p>{{gt 11 13}}</p>
<p>{{lt 11 13}}</p>
<p>{{eq 11 11}}</p>
```

(6) 条件判断的主要实现方式如下。

方式一:

```
{{if pipeline}} T1 {{end}}
```

方式二:

```
{{if pipeline}} T1 {{else}} T0 {{end}}
```

方式三:

```
{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
```

```
<h1>条件判断</h1>
<br>//案例一
{{if .show}}
展示信息
{{end}}

<br>//案例二
{{if gt .age 18}}
成年人
{{else}}
```

```
未成年人
{{end}}

<br>//案例三
{{if gt .score 90}}
优秀
{{else if gt .score 60}}
及格
{{else}}
不及格
{{end}}
```

（7）range 循环的主要实现方式如下。

方式一：

```
{{ range pipeline }} T1 {{ end }}
```

方式二：

```
//如果 pipeline 的长度为 0,则输出 else 中的内容
{{ range pipeline }} T1 {{ else }} T2 {{ end }}
```

range 可以遍历 slice、数组、map 或 channel。遍历时，会设置为当前正在遍历的元素。

对于第 1 个表达式，当遍历对象的值为 0 值时，range 直接跳过，就像 if 一样。对于第 2 个表达式，则在遍历到 0 值时执行 else。

range 的参数部分是 pipeline，所以在迭代的过程中是可以进行赋值的，但有两种赋值情况：

```
{{ range $value := pipeline }} T1 {{ end }}
{{ range $key, $value := pipeline }} T1 {{ end }}
```

如果 range 中只赋值给一个变量，则这个变量是当前正在遍历元素的值。如果赋值给两个变量，则第 1 个变量是索引值(array/slice 是数值，map 是 key)，第 2 个变量是当前正在遍历元素的值，代码如下：

```
<h1>range 循环</h1>
<h2>循环数组</h2>
{{range $index, $value:= .wife}}
<p>{{ $index}} --- {{ $value}}</p>
{{end}}
<h2>循环 map </h2>
{{range $key, $value:= .info}}
<p>{{ $key}} --- {{ $value}}</p>
{{end}}
<h2>循环空 -->"girls":map[string]interface{}{}</h2>
{{range $value:= .girls}}
```

```
<p>{{ $value}}</p>
{{else}}
没有女孩
{{end}}
```

(8) with…end 的主要实现方式如下。

方式一：

```
{{ with pipeline }} T1 {{ end }}
```

方式二：

```
{{ with pipeline }} T1 {{ else }} T0 {{ end }}
```

对于第 1 种格式，当 pipeline 不为 0 值时，将 T1 设置为 pipeline 运算的值，否则跳过。

对于第 2 种格式，当 pipeline 为 0 值时，执行 else 语句块 T0，否则设置为 pipeline 运算的值，并执行 T1。

示例代码如下：

```
<h1>with ... end</h1>
<h2>不使用 with</h2>
<p>{{.book.Name}}</p>
<p>{{.book.Price}}</p>
<h2>使用 with</h2>
{{with .book}}
<p>{{.Name}}</p>
<p>{{.Price}}</p>
{{end}}
```

(9) 函数：Go 的模板功能其实很有限，很多复杂的逻辑无法直接使用模板语法来表达，所以只能使用模板函数实现。

首先，template 包在创建新的模板时，支持.Funcs 方法来将自定义的函数集合导入该模板中，后续通过该模板渲染的文件均支持直接调用这些函数。

该函数集合的定义如下：

```
type FuncMap map[string]interface{}
```

key 为方法的名字，value 则为函数。这里函数的参数的个数没有限制，但是对于返回值有所限制。有两种选择，一种是只有一个返回值，还有一种是有两个返回值，但是第 2 个返回值必须是 error 类型的。这两种函数的区别是第 2 个函数在模板中被调用时，假设模板函数的第 2 个参数的返回不为空，则该渲染步骤将会被打断并报错。

内置函数，示例代码如下：

```
var builtins = FuncMap{
    //返回第 1 个为空的参数或最后一个参数。可以有任意多个参数
    //"and x y"等价于"if x then y else x"
    "and": and,
    //显式调用函数。第 1 个参数必须是函数类型,并且不是 template 中的函数,而是外部函数
    //例如一个 struct 中的某个字段是 func 类型的
    //"call .X.Y 1 2"表示调用 dot.X.Y(1, 2),Y 必须是 func 类型,函数参数是 1 和 2
    //函数必须只能有一个或两个返回值,如果有第 2 个返回值,则必须为 error 类型
    "call": call,
    //返回与其参数的文本表示形式等效地转义 HTML
    //这个函数在 html/template 中不可用
    "html": HTMLEscaper,
    //对可索引对象进行索引取值。第 1 个参数是索引对象,后面的参数是索引位
    //"index x 1 2 3"代表的是 x[1][2][3]。
    //可索引对象包括 map、slice、array
    "index": index,
    //返回与其参数的文本表示形式等效地转义 JavaScript
    "js": JSEscaper,
    //返回参数的 length
    "len": length,
    //布尔取反,只能有一个参数
    "not": not,
    //返回第 1 个不为空的参数或最后一个参数,可以有任意多个参数
    //"or x y"等价于"if x then x else y"
    "or": or,
    "print": fmt.Sprint,
    "printf": fmt.Sprintf,
    "println": fmt.Sprintln,
    //以适合嵌入网址查询中的形式返回其参数的文本表示的转义值
    //这个函数在 html/template 中不可用
    "urlquery": URLQueryEscaper,
}
<h1>内置函数</h1>
<p>{{len .name}} -->字节数</p>
```

自定义函数,示例代码如下:

```
//第 1 步:定义一个函数
func parserTime(t int64) string {
    return time.UNIX(t, 0).Format("2006 年 1 月 2 日 15 点 04 分 05s")
}
//第 2 步:在加载模板之前执行
router := gin.Default()
    router.SetFuncMap(template.FuncMap{
        "parserTime": parserTime,
    })
//第 3 步:在模板中使用 -->"date": time.Now().UNIX(),
<h1>自定义模板函数</h1>
<p>不使用自定义模板函数:{{.date}}</p>
<p>使用自定义模板函数:{{parserTime .date}}</p>
</body>
```

(10) 模板嵌套的主要实现方式如下。

define 可以直接在待解析内容中定义一个模板,代码如下:

```
//定义名称为 name 的 template
{{ define "name" }} T {{ end }}
```

使用 template 来执行模板,代码如下:

```
//执行名为 name 的 template
{{ template "name" }}      //不加点,不能使用当前页面的变量渲染 define 定义的模板
{{ template "name" . }}    //加入点,可以使用当前页面的变量渲染 define 定义的模板
```

完整的案例代码如下。

header.html 文件的内容如下:

```
{{define "header.html"}}
<style>
    h1{
        background: pink;
        color: aqua;
        text-align: center;
    }
</style>
<h1>这是一个头部 -- {{.header}}</h1>
{{end}}
```

footer.html 文件的内容如下:

```
{{define "footer.html"}}
<style>
    h1 {
        background: pink;
        color: aqua;
        text-align: center;
    }
</style>
<h1>这是一个尾部 -- {{.footer}}</h1>
{{end}}
```

index.html 文件的内容如下:

```
<! DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>第 1 个模板文件</title>
</head>
<body>
{{ template "header.html" .}}
```

```
<br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br><br>
<br><br><br><br><br><br><br>
{{ template "footer.html" .}}
</body>
</html>
```

main.go 文件的内容如下：

```
router.GET("/index", func(c *gin.Context) {
    c.html(200, "index.html", gin.H{
        "header": "头部头部",
        "footer": "尾部尾部",
    })
})
```

（11）模板继承的主要实现方式。

通过 block、define、template 实现模板继承。block 的使用方式如下：

```
{{ block "name" pipeline }} T {{ end }}
```

block 等价于 define 定义一个名为 name 的模板，并在“有需要”的地方执行这个模板，执行时将“.”设置为 pipeline 的值。等价于：先执行 {{ define "name" }} T {{ end }} 再执行 {{ template "name" pipeline }}。

完整案例的代码如下。

base.html 文件的内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .head {
            height: 50px;
            background-color: red;
            width: 100%;
            text-align: center;
        }
        .main {
            width: 100%;
        }
        .main .left {
            width: 30%;
            height: 1000px;
            float: left;
            background-color:violet;
            text-align: center;
```

```
        }
        .main .right {
            width: 70%;
            float: left;
            text-align: center;
            height: 1000px;
            background-color:yellowgreen;
        }
    </style>
</head>
<body>
<div class="head">
    <h1>顶部标题部分</h1>
</div>
<div class="main">
    <div class="left">
        <h1>左侧侧边栏</h1>
    </div>
    <div class="right">
        {{ block "content" . }}
        <h1>默认显示内容</h1>
        {{ end }}
    </div>
</div>
</body>
</html>
```

home.html 文件的内容如下：

```
{{ template "base.html" . }}

{{ define "content" }}
<h1>{{.s}}</h1>

{{ end }}
```

goods.html 文件的内容如下：

```
{{ template "base.html" . }}

{{ define "content" }}
<h1>{{.s}}</h1>
{{ end }}
```

main.go 文件的内容如下：

```
router.GET("/goods", func(c *gin.Context) {
    c.html(200, "goods.html", gin.H{
        "s": "这是商品 goods 页面",
```

```
    })
})
router.GET("/home", func(c *gin.Context) {
    c.html(200, "home.html", gin.H{
        "s": "这是首页,home",
    })
})
```

(12) 修改默认标识符：Go 标准库的模板引擎使用花括号{{和}}作为标识，而许多前端框架(如 Vue 和 AngularJS)也使用{{和}}作为标识符，所以当同时使用 Go 语言模板引擎和以上前端框架时就会出现冲突，这时需要修改标识符，即修改前端的或者修改 Go 语言的。这里演示如何修改 Go 语言模板引擎默认的标识符：

```
router.Delims("[[","]]")
```

(13) XSS 攻击，代码如下：

```
//定义函数
func safe (str string) template.html {
    return template.html(str)
}
//注册函数
router.SetFuncMap(template.FuncMap{
    "parserTime": parserTime,
    "safe": safe,
})

//模板中使用
<h1>XSS 攻击</h1>
<p>{{.str1}}</p>
<p>{{safe .str1 }}</p>
```

5.2.5　实例：Gin 框架的 Cookie 与 Session

1. Cookie 和 Session 的产生背景

由于 HTTP 是无状态的，服务器无法确定这次请求和上次请求是否来自同一个客户端。解决此问题给客户端颁发一个通行证，每个客户端拥有一个通行证，无论通过哪个客户端访问都必须携带自己的通行证。这样服务器就能从通行证上确认客户身份了。这就是 Cookie 的工作原理。

利用 Session 和 Cookie 可以让服务器知道不同的请求是否来自同一个客户端。

Cookie 与 Session 的区别：

(1) 什么是 Cookie?

Cookie 原意为甜饼，是由 W3C 组织提出的，是最早由 Netscape 社区发展的一种机制。目前 Cookie 已经成为标准，所有的主流浏览器(如 IE、Netscape、Firefox、Opera 等)都支持

Cookie。

Cookie 实际上是一小段文本信息。当客户端请求服务器时，如果服务器需要记录该用户的状态，就使用 response 向客户端浏览器颁发一个 Cookie。

客户端浏览器会把 Cookie 保存起来。当浏览器再请求该网站时，浏览器会把请求的网址连同该 Cookie 一同提交给服务器。服务器检查该 Cookie，以此来辨认用户状态。服务器还可以根据需要修改 Cookie 的内容。

(2) 什么是 Session?

除了使用 Cookie，Web 应用程序中还经常使用 Session 来记录客户端状态。Session 是服务器端使用的一种记录客户端状态的机制，使用上比 Cookie 简单一些，但相应地也增加了服务器的存储压力。

Session 是另一种记录客户状态的机制，不同的是 Cookie 保存在客户端浏览器中，而 Session 保存在服务器上。

客户端浏览器访问服务器时，服务器把客户端信息以某种形式记录在服务器上。这就是 Session。客户端浏览器再次访问时只需从该 Session 中查找该客户的状态就可以了。

(3) Cookie 和 Session 的区别：①Cookie 数据存放在客户的浏览器上，Session 数据放在服务器上(可以放在文件、数据库或者内存)；②Cookie 不是很安全，别人可以分析存放在本地的 Cookie 并进行 Cookie 欺骗，考虑到安全应当使用 Session；③两者最大的区别在于生存周期，一个是从 IE 启动到 IE 关闭(浏览器页面一关，Session 就消失了)，一个是预先设置的生存周期，或永久地保存于本地的文件(Cookie)；④Session 会在一定时间内保存在服务器上。当访问增多时会比较占用服务器的资源，考虑到减轻服务器压力，应当使用 Cookie；⑤单个 Cookie 保存的数据不能超过 4KB，很多浏览器限制一个站点最多保存 20 个 Cookie(Session 对象没有对存储的数据量进行限制，其中可以保存更为复杂的数据类型)。

综上所述，如果说 Cookie 机制是通过检查客户身上的“通行证”来确定客户身份，则 Session 机制就是通过检查服务器上的客户明细表来确认客户身份。Session 相当于程序在服务器上建立的一份客户档案，客户来访时只需查询客户档案表就可以了。

Session 信息存放在 Server 端，但 Session ID 存放在 Client Cookie。

2. 安装 Session 插件

安装 Session 组件的方式很简单，默认安装最新版本，代码如下：

```
go get github.com/gin-contrib/sessions
```

3. Cookie 与 Session 案例

(1) Cookie 案例，示例代码如下：

```
//anonymous-link\example\chapter5\gin\Cookie_example.go
package main

import (
    "fmt"
```

```
    "github.com/gin-gonic/gin"
)

func main() {
    /**
    所有的接口都要由路由进行管理。
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由并引入默认中间件
        router := gin.Default()
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理

    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    router.GET("/Cookie", func(context *gin.Context) {
        //获取 Cookie
        Cookie, err := context.cookie("gin_Cookie")
        if err != nil {
            Cookie = "NotSet"
            //设置 Cookie
            context.SetCookie("gin_Cookie", "test", 3600, "/", "localhost", false, true)
        }
        fmt.Println("Cookie value: ", Cookie)
    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")


    /**
    使用 curl 命令测试:
        curl -v http://172.30.100.101:9000/Cookie

    */
}
```

(2) Session 案例,示例代码如下:

```
//anonymous-link\example\chapter5\gin\session_example.go
package main

import (
    "github.com/gin-contrib/sessions"
    "github.com/gin-contrib/sessions/Cookie"
    "github.com/gin-gonic/gin"
```

```
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求
    创建路由并引入默认中间件
        router := gin.Default()
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理
    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    //定义加密
    store := Cookie.NewStore([]Byte("secret"))

    //绑定 Session 中间件
    router.Use(sessions.Sessions("mysession", store))

    //定义 GET 方法
    router.GET("/session", func(context *gin.Context) {
        //初始化 Session 对象
        session := sessions.Default(context)

        //如果浏览器第 1 次访问时返回状态码 401,则第 2 次访问时返回状态码 200
        if session.Get("user") != "frank" {
            session.Set("user", "frank")
            session.Save()
            context.json(http.StatusUnauthorized, gin.H{"user": session.Get("user")})
        } else {
            context.String(http.StatusOK, "Successful second visit")
        }

    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

    /**
        测试工具建议使用浏览器访问 http://172.30.100.101:9000/session,不推荐使用 curl
命令
        因为 curl 工具无法缓存,而浏览器有缓存,所以可以很明显地看到测试效果
    */
}
```

(3) 将 Session 存储在 Redis 服务器,示例代码如下:

```
//anonymous - link\example\chapter5\gin\session_redis.go
package main

import (
    "github.com/gin-contrib/sessions"
    "github.com/gin-contrib/sessions/redis"
    "github.com/gin-gonic/gin"
    "net/http"
)

func main() {
    /**
    所有的接口都要由路由进行管理
        Gin 的路由支持 GET、POST、PUT、DELETE、PATCH、HEAD、OPTIONS 等请求
        同时还有一个 Any 函数,可以同时支持以上所有请求

    创建路由并引入默认中间件
        router := gin.Default()
        在源码中,首先通过 New 创建一个 engine,紧接着通过 Use 方法传入了 Logger()和 Recovery()
这两个中间件
        其中 Logger 用于对日志进行记录,而 Recovery 用于对有 painc 时进行 500 的错误处理

    创建路由无中间件
        router := gin.New()
    */
    router := gin.Default()

    //定义加密(将 Session 信息存储在 Redis 服务器)
    store, _ := redis.NewStore(10, "tcp", "172.200.1.254:6379", "", []Byte("secret"))

    //绑定 Session 中间件
    router.Use(sessions.Sessions("mySession", store))

    //定义 GET 方法
    router.GET("/session", func(context *gin.Context) {
        //初始化 Session 对象
        session := sessions.Default(context)

        //如果浏览器第 1 次访问时返回状态码 401,则第 2 次访问时返回状态码 200
        if session.Get("user") != "frank" {
            session.Set("user", "frank")
            session.Save()
            context.json(http.StatusUnauthorized, gin.H{"user": session.Get("user")})
        } else {
            context.String(http.StatusOK, "Successful second visit")
        }

    })

    //启动路由并指定监听的地址及端口,若不指定,则默认监听 0.0.0.0:8080
    router.Run("172.30.100.101:9000")

}
```

5.2.6 Gin 框架的 JSON Web Token

1. JSON Web Token 概述

JSON Web Token(JWT)是为了在网络应用环境间传递声明而执行的一种基于 JSON 的开放标准(RFC 7519)。该 Token 被设计为紧凑且安全的,特别适用于分布式站点的单点登录(SSO)场景。

JWT 的声明一般被用来在身份提供者和服务提供者间传递被认证的用户身份信息,以便于从资源服务器获取资源,也可以增加一些额外的其他业务逻辑所必需的声明信息,该 Token 也可直接被用于认证,也可被加密。

2. JWT 的组成

Header:承载两部分信息。第 1 部分声明类型,这里是 JWT,第 2 部分声明加密的算法,通常直接使用 HMAC SHA256。

Playload:载荷就是存放有效信息的地方。iss:签发者;sub:面向的用户;aud:接收方;exp:过期时间;nbf:生效时间;iat:签发时间;jti:唯一身份标识。

Signature:签证信息,这个签证信息由 Header (base64 后的)、Payload (base64 后的)、Secret 三部分组成。

3. JWT 实现的单点登录流程

JWT 实现的单点登录流程如图 5-24 所示。

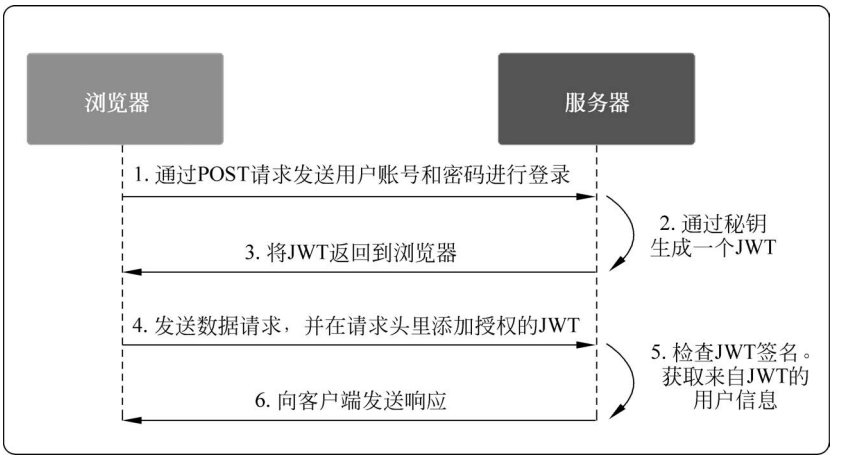


图 5-24 JWT 实现的单点登录流程

5.2.7 实例:Go 语言的 ORM 库 xorm

xorm 是一个简单而强大的 Go 语言 ORM 库,通过它可以使数据库操作变得非常简便。

1. 数据库及相关依赖组件安装

安装数据库服务，以 CentOS 7. x 系统为例，命令如下：

```
[root@localhost]# yum -y install mariadb-server
```

启动数据库服务，以 CentOS 7. x 系统为例，命令如下：

```
systemctl start mariadb
```

设置数据库服务开机自动启动，以 CentOS 7. x 系统为例，命令如下：

```
systemctl enable mariadb
Created symlink from /etc/systemd/system/multi-
user.target.wants/mariadb.service to
/usr/lib/systemd/system/mariadb.service.
```

进入 MySQL 并创建数据库，命令如下：

```
mysql
Welcome to the MariaDB monitor. Commands end with ; or \g
Your MariaDB connection id is 2
Server version: 5.5.65-MariaDB MariaDB Server

Copyright (c) 2000, 2018, Oracle, MariaDB Corporation Ab and others

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement

MariaDB [(none)]>
MariaDB [(none)]> create database Go CHARACTER SET utf8mb4;
Query OK, 1 row affected (0.03 sec)

MariaDB [(none)]>
MariaDB [(none)]> CREATE USER jason@'%' IDENTIFIED BY 'frank';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]>
MariaDB [(none)]> GRANT ALL ON Go.* TO jason@'%';
Query OK, 0 rows affected (0.00 sec)

MariaDB [(none)]>
MariaDB [(none)]> quit
Bye
```

官方 xorm 驱动，GitHub 地址如下：

```
https://github.com/go-xorm/xorm
```

安装 xorm 驱动，命令如下：

```
go get github.com/go-xorm/xorm
```

安装 Go 语言的 MySQL 驱动，命令如下：

```
go get github.com/go-sql-driver/mysql
```

2. CRUD 增、删、改、查操作

(1) 查询所有数据，代码如下：

```
//anonymous-link\example\chapter5\xorm\find1.go
package main

import (
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "github.com/go-xorm/xorm"
    "log"
)

//结构体字段对应数据库中的表字段
type User struct {
    Id      int64
    Name    string   `xorm:"name"`
    Age     int      `xorm:"age"`
    Phone   string   `xorm:"phone"`
    Address string   `xorm:"address"`
}

func main() {
    /**
     配置连接数据库信息，格式如下
        用户名:密码@tcp(数据库服务器地址:端口)/数据库名称? charset=字符集
     */
    cmd := fmt.Sprintf("jason:frank@tcp(172.200.1.254:3306)/Go? charset=utf8mb4")

    //使用上面的配置信息连接数据库，但要指定数据库的类型(这里指 MySQL)
    db_conn, err := xorm.NewEngine("mysql", cmd)
    if err != nil {
        log.Fatal(err)
    }

    //释放资源
    defer db_conn.Close()

    users := make([]User, 0) //等效于"var users []User"

    //获取所有资源
    err = db_conn.Find(&users)
    if err != nil {
        log.Println(err)
    } else {
        for _, user := range users {
```

```
            log.Println(user.Id, user.Name, user.Age, user.Address)
        }
    }
}
```

（2）过滤查询数据，代码如下：

```
//anonymous - link\example\chapter5\xorm\find2.go
package main

import (
    "fmt"
    _ "github.com/go - sql - driver/mysql"
    "github.com/go - xorm/xorm"
    "log"
)
//结构体字段对应数据库中的表字段
type User struct {
    Id       int64
    Name     string   `xorm:"name"`
    Age      int      `xorm:"age"`
    Phone    string   `xorm:"phone"`
    Address  string   `xorm:"address"`
}

func main() {
    /**
    配置连接数据库信息，格式如下
        用户名:密码@tcp(数据库服务器地址:端口)/数据库名称? charset = 字符集
    */
    cmd := fmt.Sprintf("jason:frank@tcp(172.200.1.254:3306)/Go? charset = utf8mb4")

    //使用上面的配置信息连接数据库，但要指定数据库的类型(这里指 MySQL)
    db_conn, err := xorm.NewEngine("mysql", cmd)
    if err != nil {
        log.Fatal(err)
    }

    //释放资源
    defer db_conn.Close()

    var users []User //等效于"users := make([]User, 0)"

    //过滤数据(年龄为 19～25 岁)
    err = db_conn.Where("age > ? and age < ?", 19, 25).Find(&users)
    if err != nil {
        log.Println(err)
    } else {
        for _, user := range users {
```

```
            log.Println(user.Id, user.Name, user.Age, user.Address)
        }
    }
}
```

(3) 插入操作,代码如下:

```
//anonymous - link\example\chapter5\xorm\insert.go
package main

import (
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "github.com/go-xorm/xorm"
    "log"
)
//结构体字段对应数据库中的表字段
type User struct {
    Id       int64
    Name     string   `xorm:"name"`
    Age      int      `xorm:"age"`
    Phone    string   `xorm:"phone"`
    Address  string   `xorm:"address"`
}

func main() {
    /**
    配置连接数据库信息,格式如下
        用户名:密码@tcp(数据库服务器地址:端口)/数据库名称? charset=字符集
    */
    cmd := fmt.Sprintf("jason:frank@tcp(172.200.1.254:3306)/Go? charset=utf8mb4")

    //使用上面的配置信息连接数据库,但要指定数据库的类型(这里指 MySQL)
    db_conn, err := xorm.NewEngine("mysql", cmd)
    if err != nil {
        log.Fatal(err)
    }

    //释放资源
    defer db_conn.Close()

    //定义待插入用户的数据
    new_user := User{
        Id:      3,
        Name:    "诡术妖姬",
        Age:     25,
        Phone:   "1000001",
        Address: "艾欧尼亚",
    }
```

```
    n, err := db_conn.Insert(new_user)
    fmt.Printf("成功插入了[%d]条数据! \n", n)
}
```

(4) 删除操作,代码如下:

```
//anonymous - link\example\chapter5\xorm\delete.go
package main

import (
    "fmt"
    _ "github.com/go - sql - driver/mysql"
    "github.com/go - xorm/xorm"
    "log"
)
//结构体字段对应数据库中的表字段
type User struct {
    Id       int64
    Name     string   `xorm:"name"`
    Age      int      `xorm:"age"`
    Phone    string   `xorm:"phone"`
    Address  string   `xorm:"address"`
}

func main() {
    /**
    配置连接数据库信息,格式如下
        用户名:密码@tcp(数据库服务器地址:端口)/数据库名称? charset = 字符集
    */
    cmd := fmt.Sprintf("jason:frank@tcp(172.200.1.254:3306)/Go? charset = utf8mb4")

    //使用上面的配置信息连接数据库,但要指定数据库的类型(这里指 MySQL)
    db_conn, err := xorm.NewEngine("mysql", cmd)
    if err != nil {
        log.Fatal(err)
    }

    //释放资源
    defer db_conn.Close()

    user := User{}

    //定义删除 name 字段为"诡术妖姬"的数据
    n, err := db_conn.Where("name = ?", "诡术妖姬").Delete(user)
    if err != nil {
        log.Println(err)
    }
    fmt.Printf("成功删除了[%d]条数据! \n", n)
}
```

(5) 更新操作,代码如下:

```
//anonymous - link\example\chapter5\xorm\update.go
package main

import (
    "fmt"
    "log"

    _ "github.com/go - sql - driver/mysql"
    "github.com/go - xorm/xorm"
)

//结构体字段对应数据库中的表字段
type User struct {
    Id       int64
    Name     string   `xorm:"name"`
    Age      int      `xorm:"age"`
    Phone    string   `xorm:"phone"`
    Address  string   `xorm:"address"`
}

func main() {
    /**
     配置连接数据库信息,格式如下
        用户名:密码@tcp(数据库服务器地址:端口)/数据库名称? charset = 字符集
     */
    cmd := fmt.Sprintf("jason:frank@tcp(172.200.1.254:3306)/Go? charset = utf8mb4")

    //使用上面的配置信息连接数据库,但要指定数据库的类型(这里指 MySQL)
    db_conn, err := xorm.NewEngine("mysql", cmd)
    if err != nil {
        log.Fatal(err)
    }

    //释放资源
    defer db_conn.Close()

    //定义待更新的字段内容
    user := User{Age: 27}

    //定义删除 name 字段为"诡术妖姬"的数据
    n, err := db_conn.Where("id = ?", 1).Update(user)
    if err != nil {
        log.Println(err)
    }
    fmt.Printf("成功更新了[%d]条数据! \n", n)
}
```

5.2.8 实例:Go 语言解析 YAML 配置文件

(1) YAML 配置文件,内容如下:

```
#anonymous - link\example\chapter5\analysis\config_file.yaml

##################Configuration Example ##################

# ====================== General ================================
general:
    #使用的 CPU 核数,默认使用操作系统的所有 CPU
    max_procs_enable: true
    #log 文件路径
    log_path: /etc/springboardMchine/seelog.xml
    #Debug 模式
    Debug: true
# ========================== HTTP API ========================
#用来以 HTTP Server 的形式提供对外的 API
api:
    host: 0.0.0.0
    port: 8080
# ========================== MySQL ================================
#MySQL 配置
mysql:
    host: 172.200.1.254
    port: 3306
    name: jumpserver
    user: jason
    password: frank

#Cache 配置
cache:
    host: 172.200.1.254
    port: 6379
    password: frank
    db: 0
# ========== RPC ============
#RPC 配置
rpc:
    host: 0.0.0.0
    port: 8888
```

(2) 自定义解析包,代码如下:

```
//anonymous - link\example\chapter5\analysis\manual_resolve.go
package config

import (
    "errors"
    "fmt"
    "github.com/toolkits/file"
    "gopkg.in/yaml.v1"
)

//根据配置文件定义与之对应的结构体字段
```

```
type GeneralConfig struct {
    LogPath         string   `yaml:"log_path"`
    Debug           bool     `yaml:"Debug"`
    MaxProcsEnable  bool     `yaml:max_procs_enable`
}

//根据定义的结构体,将配置文件的数据手动生成一个结构体对象
func newGeneralConfig() *GeneralConfig {
    return &GeneralConfig{
        LogPath:        "/etc/springboardMchine/seelog.xml",
        Debug:          true,
        MaxProcsEnable: true,
    }
}

type APIConfig struct {
    Host string   `yaml:"host"`
    Port int      `yaml:port`
}

func newAPIConfig() *APIConfig {
    return &APIConfig{
        Host: "0.0.0.0",
        Port: 8080,
    }
}

type MysqlConfig struct {
    Host      string   `yaml:"host"`
    Name      string   `yaml:"name"`
    Port      int      `yaml:"port"`
    Password  string   `yaml:"password"`
    User      string   `yaml:"user"`
}

func newMysqlConfig() *MysqlConfig {
    return &MysqlConfig{
        Host:     "172.200.1.254",
        Name:     "jumpserver",
        Port:     3306,
        User:     "jason",
        Password: "frank",
    }
}

type CacheConfig struct {
    Host     string   `yaml:""host`
    Port     int      `yaml:"port"`
    Password string   `yaml:"password"`
    Db       int      `yaml:"db"`
}
```

```
func newCacheConfig() *CacheConfig {
    return &CacheConfig{
        Host:     "172.200.1.254",
        Port:     6379,
        Db:       0,
        Password: "frank",
    }
}

type RpcConfig struct {
    Host string    `yaml:"host"`
    Port int       `yaml:"port"`
}

func newRpcConfig() *RpcConfig {
    return &RpcConfig{
        Host: "0.0.0.0",
        Port: 8888,
    }
}

//定义一个结构体,对上述定义的结构体进行封装
type ConfigYaml struct {
    Mysql      *MysqlConfig     `yaml:"mysql"`
    API        *APIConfig       `yaml:"api"`
    RpcClient  *RpcConfig       `yaml:"rpc"`
    Cache      *CacheConfig     `yaml:"cache"`
    General    *GeneralConfig   `yaml:"general"`
}

//使用封装后的结构体进行实例化
var (
    Config *ConfigYaml = &ConfigYaml{
        Mysql:     newMysqlConfig(),
        API:       newAPIConfig(),
        RpcClient: newRpcConfig(),
        Cache:     newCacheConfig(),
        General:   newGeneralConfig(),
    }
)

//定义连接数据库的函数
func DatabaseDialString() string {
    return fmt.Sprintf("%s:%s@%s(%s:%d)/%s?charset=%s",
        Config.Mysql.User,
        Config.Mysql.Password,
        "tcp",
        Config.Mysql.Host,
        Config.Mysql.Port,
        Config.Mysql.Name,
        "utf8mb4",
```

```
    )
}

func Parse(configFile string) error {
    //判断文件是否存在,如果不存在,就返回错误
    if ! file.IsExist(configFile) {
        return errors.New("config file " + configFile + " is not exist")
    }

    //读取配置文件信息(读取到的数据实际上是字符串),如果读取失败,则会返回错误
    configContent, err := file.ToTrimString(configFile)
    if err != nil {
        return err
    }

    //使用 YAML 格式对上一步读取到的字符串进行解析
    err = yaml.Unmarshal([]Byte(configContent), &Config)
    if err != nil {
        return err
    }
    return nil
}
```

(3) 调用解析包,代码如下:

```
//anonymous - link\example\chapter5\analysis\auto_resolve.go
package main

import (
    "config"
    "flag"
    "fmt"
    "log"
    "os"
)

func main() {
    /**
     flag 的 string 方法的源代码如下:
        func String(name string, value string, usage string) *string {
            return CommandLine.String(name, value, usage)
        }
     下面对 flag 的 string 方法进行解释说明
        name 用于指定自定义名称,即用来标识该 flag 是用来干什么的
        value 用于指定默认值
        usage 是当前的 flag 的一个描述信息
         *string 指返回值是存储标志值的字符串变量的地址(指针变量)
    */
    configFile := flag.String("c", "/etc/springboardMchine/seelog.xml", "yaml config file")
    /*
     flag 下面的这种写法和上面的作用是一样的,只不过上面的写法更简便,推荐使用上面的写法
```

```
    var mode string
    flag.StringVar(&mode,"m", "client", "mode[client|server|jump|audit]")
    version := flag.Bool("v", false, "show version")
    */
    //开始进行解析,会将解析的结果复制给上述的 configFile、version、mode 这 3 个指针变量
    flag.Parse()

    //判断配置文件的长度
    if len(*configFile) == 0 {
        log.Println("not have config file.")
        os.Exit(0)
    }

    //解析配置文件,利用手动解析封装的方法,如果在不同的包下,则注意引入包
    err := config.Parse(*configFile)
    if err != nil {
        log.Println("parse config file error:", err)
        os.Exit(0)
    }

    //打印解析的参数,利用手动解析封装的方法,如果在不同的包下,则注意引入包
    fmt.Println(config.Config.Mysql.Host, config.Config.Mysql.Name, config.Config.Mysql.
Port)
}
```

(4) 数据结构相互转换。

JSON 转 Map,示例代码如下:

```
//anonymous - link\example\chapter5\analysis\json_map.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    jsonStr := `
        {
                "name": "张三",
                "age": 18
        }
        `

    var mapResult map[string]interface{}
    err := json.Unmarshal([]Byte(jsonStr), &mapResult)
    if err != nil {
        fmt.Println(err)
        return
    }
```

```
    fmt.Println(mapResult)
}
```

JSON 转 Struct，示例代码如下：

```
//anonymous - link\example\chapter5\analysis\json_struct.go
package main

import (
    "encoding/json"
    "fmt"
)

type People1 struct {
    Name string   `json:"name"`
    Age int       `json:"age"`
}

func main() {
    jsonStr := `
        {
                "name": "张三",
                "age": 12
        }
        `
    var people People1
    err := json.Unmarshal([]Byte(jsonStr), &people)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(people)
}
```

Map 转 JSON，示例代码如下：

```
//anonymous - link\example\chapter5\analysis\map_json.go
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    var mapInstances []map[string]interface{}
    instance1 := map[string]interface{}{"name": "张三", "age": 18}
    instance2 := map[string]interface{}{"name": "李四", "age": 35}
    mapInstances = append(mapInstances, instance1, instance2)
```

```
    jsonStr, err := json.Marshal(mapInstances)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(jsonStr))
}
```

Map 转 Struct 的步骤如下。

安装插件，命令如下：

```
go get github.com/goinggo/mapstructure
```

使用示例，代码如下：

```
//anonymous - link\example\chapter5\analysis\map_struct.go
package main

import (
    "fmt"
    "github.com/goinggo/mapstructure"
)

type People3 struct {
    Name string   `json:"name"`
    Age int       `json:"age"`
}

//go get github.com/goinggo/mapstructure
func main() {
    mapInstance := make(map[string]interface{})
    mapInstance["Name"] = "张三"
    mapInstance["Age"] = 18

    var people People3
    err := mapstructure.Decode(mapInstance, &people)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(people)
}
```

Struct 转 JSON，示例代码如下：

```
//anonymous - link\example\chapter5\analysis\struct_json.go
package main
```

```
import (
    "encoding/json"
    "fmt"
)

type People2 struct {
    Name string   `json:"name"`
    Age int       `json:"age"`
}

func main() {
    p := People2{
        Name: "张三",
        Age: 18,
    }

    jsonBytes, err := json.Marshal(p)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(string(jsonBytes))
}
```

Struct 转 Map,示例代码如下:

```
//anonymous-link\example\chapter5\analysis\struct_map.go
package main

import (
    "fmt"
    "reflect"
)

type People4 struct {
    Name string   `json:"name"`
    Age int       `json:"age"`
}

func main() {
    people := People4{"张三", 18}

    obj1 := reflect.TypeOf(people)
    obj2 := reflect.ValueOf(people)

    var data = make(map[string]interface{})
    for i := 0; i < obj1.NumField(); i++{
        data[obj1.Field(i).Name] = obj2.Field(i).Interface()
    }

    fmt.Println(data)
}
```

5.2.9　实例：Go 使用 Gin 文件上传/下载及 swagger 配置

1. form 表单上传文件

(1) 单文件上传，前端示例代码如下：

```
<form action = "/upload2" method = "post" enctype = "multipart/form - data">
    < input type = "file" name = "file">
    < input type = "submit" value = "提交">
</form >
```

需要注意的是设置 enctype 属性参数。

后端代码如下：

```
func Upload2(context * gin.Context) {
    fmt.Println("+++++++++++++++ + ")
    file,_ : = context.FormFile("file")          //获取文件
    fmt.Println(file.Filename)

file_path : = "upload/" + file.Filename
//设置保存文件的路径，不要忘了后面的文件名

    context.SaveUploadedFile(file, file_path)    //保存文件

    context.String(http.StatusOK,"上传成功")
}
```

防止文件名冲突，使用时间戳命名，示例代码如下：

```
unix_int : = time.Now().UNIX()                    //时间戳，int 类型
time_unix_str : = strconv.FormatInt(unix_int,10)
//将 int 类型转换为 string 类型，方便拼接，使用 sprinf 也可以

file_path : = "upload/" + time_unix_str + file.Filename
//设置保存文件的路径，不要忘了后面的文件名
context.SaveUploadedFile(file, file_path)         //保存文件
```

(2) 多文件上传，前端代码如下：

```
<form action = "/upload2" method = "post" enctype = "multipart/form - data">
    < input type = "file" name = "file">
    < input type = "file" name = "file">
    < input type = "submit" value = "提交">
</form >
```

需要注意的是不要忘了 enctype 属性参数。

后端代码如下：

```
func Upload2(context * gin.Context) {
    form,_ := context.MultipartForm()
    files := form.File["file"]

    for _,file := range files {                          //循环
        fmt.Println(file.Filename)
        unix_int := time.Now().UNIX()                    //时间戳,int 类型
        time_unix_str := strconv.FormatInt(unix_int,10)
        //将 int 类型转换为 string 类型,方便拼接,使用 sprinf 也可以

        file_path := "upload/" + time_unix_str + file.Filename
        //设置保存文件的路径,不要忘了后面的文件名
        context.SaveUploadedFile(file, file_path)    //保存文件
    }

    context.String(http.StatusOK,"上传成功")
}
```

需要注意的是 form.File["file"] 使用的是方括号,而不是圆括号。

2. AJAX 方式上传文件

后端代码和上面使用的 form 表单方式是一样的。

(1) 单文件,前端代码如下:

```
<script src = "/static/js/jquery.min.js"></script>
<form>
    {{/ * <input type = "file" name = "file"> * /}}
    用户名:<input type = "test" id = "name"><br>
    <input type = "file" id = "file">
    <input type = "button" value = "提交" id = "btn_add">
</form>


<script>
    var btn_add = document.getElementById("btn_add");
    btn_add.onclick = function (ev) {
        var name = document.getElementById("name").value;
        var file = $("#file")[0].files[0];

        var form_data = new FormData();
        form_data.append("name",name);
        form_data.append("file",file);



        $.ajax({
            url:"/upload2",
            type:"POST",
            data:form_data,
            contentType:false,
```

```
            processData:false,
            success:function (data) {
                console.log(data);
            },
            fail:function (data) {
                console.log(data);
            }
        })

    }
</script>
```

需要注意的点：①引入 jquery. min. js 文件；②在 AJAX 中需要加两个参数：contentType：false 和 processData：false。processData：false 的默认值为 true，当设置为 true 时，jQuery AJAX 提交时不会序列化 data，而是直接使用 data。contentType：false 不使用默认的 application/x-www-form-urlencoded 这种 contentType。

分界符：目的是防止上传文件中出现分界符导致服务器无法正确识别文件的起始位置，在 AJAX 中将 contentType 设置为 false 是为了避免 jQuery 对其操作，从而失去分界符。

（2）多文件，需要理解的是如果 name 名称不相同，则表示个单文件上传，如果 name 名称相同，则表示多个文件上传。前端代码如下：

```
<script>
    var btn_add = document.getElementById("btn_add");
    btn_add.onclick = function (ev) {
        var name = document.getElementById("name").value;
        console.log( $ (".file"));
        var files_tag = $ (".file");
        var form_data = new FormData();

        for (var i = 0;i < files_tag.length;i++){
            var file = files_tag[i].files[0];
            form_data.append("file",file);

        }

        console.log(files);
        form_data.append("name",name);

        $ .ajax({
            url:"/upload2",
            type:"POST",
            data:form_data,
            contentType:false,
            processData:false,
            success:function (data) {
                console.log(data);
```

```
            },
            fail:function (data) {
                console.log(data);
            }
        })



    }
</script>
```

3. Go 文件上传和下载及 swagger 配置

文件上传,示例代码如下:

```
//@Summary 上传文件
//@Description
//@Tags file
//@Accept multipart/form-data
//@Param file formData file true "file"
//@Produce json
//@Success 200 {object} filters.Response {"code":200,"data":nil,"msg":""}
//@Router /upload [post]
func UploadFile(ctx *gin.Context) {
    file, header, err := ctx.Request.FormFile("file")
    if err != nil {
        returnMsg(ctx, configs.ERROR_PARAMS, "", err.Error())
        return
    }
    //获取文件名
    filename := header.Filename
    //写入文件
    out, err := os.Create("./static/" + filename)
    if err != nil {
        returnMsg(ctx, configs.ERROR_SERVERE, "", err.Error())
        return
    }
    defer out.Close()
    _, err = io.Copy(out, file)
    if err != nil {
        log.Fatal(err)

    }
    returnMsg(ctx, 200, "", "success")
}
```

文件下载,示例代码如下:

```
//@Summary 下载文件
//@Description
//@Tags file
```

```
//@Param filename query string true "file name"
//@Success 200 {object} gin.Context
//@Router /download [get]
func DownloadFile(ctx *gin.Context) {
    filename := ctx.DefaultQuery("filename", "")
    //fmt.Sprintf("attachment; filename= %s", filename)对下载的文件重命名
    ctx.Writer.Header().Add("Content-Disposition", fmt.Sprintf("attachment; filename= %s",
filename))
    ctx.Writer.Header().Add("Content-Type", "application/octet-stream")
    ctx.File("./static/a.txt")
}
```

5.3　理解并掌握 MVC 分层开发规范

MVC 实际是一种软件构件模式。它被设计的目的是降低程序开发中代码业务的耦合度，并且实现高重用性，以此增加代码复用率。部署快，并且生命周期内成本低，可维护性高也是 MVC 模式的特点。

开发 Web 应用程序主要使用 MVC 模式，使用分层开发模式能在大型项目中让开发人员更好地协同工作。MVC 是 Model(模型)、View(视图)、Controller(控制器)的简写，其交互流程如图 5-25 所示。

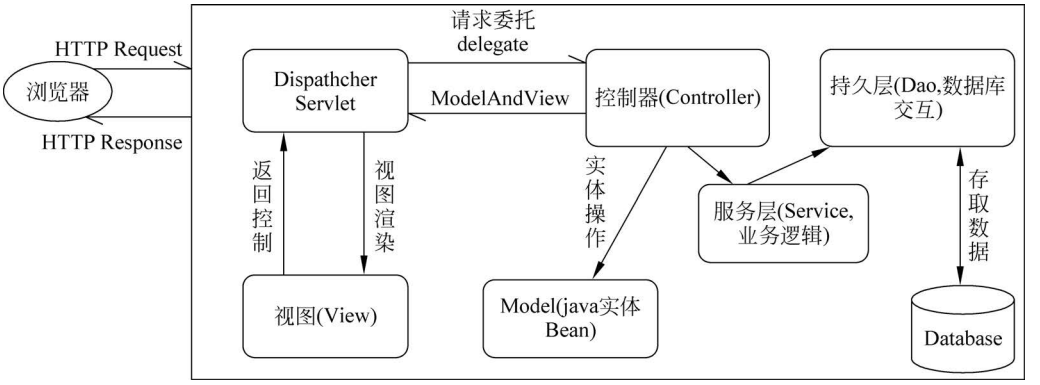


图 5-25　MVC 模式交互流程

Model：其作用是在内存中暂时存储数据，并在数据变化时更新控制器(如果要持久化，则需要把它写入数据库或者磁盘文件中)，负责数据库操作和业务逻辑操作。

View：主要用来解析、处理、显示内容，并进行模板的渲染。

Controller：主要用来处理视图中的响应。它执行如何调用 Model(模型)的实体 Bean、如何调用业务层的数据增加、删除、修改和查询等业务操作，以及如何将结果返给视图进行渲染。建议在控制器中尽量不要放逻辑代码。

这样分层的好处是：将应用程序的用户界面和业务逻辑分离，使代码具备良好的可扩展性、可复用性、可维护性和灵活性。

在实际的项目开发中，基本上采用前后端分离的方式进行项目开发，这里以后端开发为例进行说明，其主要分层结构和包目录如图 5-26 所示。

图 5-26　项目开发分层结构和包目录

以实际开发一个模块举例：

(1) 首先定义一个模型，即 Model，表明数据库和实体的映射关系，并封装相关的数据操作方法，示例如图 5-27 所示。

dhcp.go

```
package model

import ...

type DhcpModel struct {
    BaseModel
    Ip    string `json:"ip" gorm:"column:ip;not null" binding:"required" validate:"min=1,max=128"`
    Mac   string `json:"mac" gorm:"column:mac;null"`
    App   string `json:"app" gorm:"column:app;null"`
    State string `gorm:"column:state;null" json:"state"`
    Desc  string `json:"desc" gorm:"column:desc;null;type:text"`
}

func (c *DhcpModel) TableName() string {
    return "dhcp_info"
}

func (c *DhcpModel) Create() error {
    return DB.Create(&c).Error
}
```

图 5-27　MVC 模式下的 Model 示例

(2) 编写 Service 层，主要实现具体的执行业务的方法及调用 Model 层的数据操作，示例如图 5-28 所示。

(3) Handler 层主要实现对客户端请求进行处理，负责接收客户端请求参数，并调用 Service 层相关的方法进行数据处理，并把结果返回，示例如图 5-29 所示。

dhcp_manage.go

```go
package service

import ...

func ConnectIPs() (error, map[string]map[string]string) {
    shellPath := utils.GetCurrentAbPathByCaller() + "/../ansible/script/info/app_ips.sh"
    cmd := exec.Command( name: "sh",  arg...: "-c", shellPath)
    var stdout bytes.Buffer
    cmd.Stdout = &stdout
    err := cmd.Run()
    if err != nil { return err, nil }
    info := string(stdout.Bytes())
    infos := strings.Split(info,  sep: "\n")
    allIPs := make(map[string]map[string]string)
    for i := 0; i+2 < len(infos); i += 3 {
        ipInfo := map[string]string{"mac": infos[i+1], "state": infos[i+2]}
        allIPs[infos[i]] = ipInfo
    }
    for k, v := range allIPs {
        dhcpModel := model.DhcpModel{
            Ip:    k,
            Mac:   v["mac"],
            State: v["state"],
        }
        err := dhcpModel.Create()
        if err != nil { return err, nil }
    }
    return err, allIPs
}
```

图 5-28　MVC 模式下的 Service 示例

dhcp_manage.go

```go
package handler

import ...

// GetDhcpIps
// @Summary 查看分配IP
// @Description 查看分配对IP、Mac、状态
// @Tags DHCP服务
// @schemes http https
// @Accept json
// @Produce json
// @Security ApiKeyAuth
// @Response 200 {object} handler.Response
// @Router /dhcp/ips [get]
func GetDhcpIps(c *base.Context) {

    err, data := service.ConnectIPs()
    if err != nil {
        SendResponse(c, err,  data: nil)
        return
    } else {
        SendResponse(c,  err: nil, data)
        return
    }
}
```

图 5-29　MVC 模式下的 Handler 示例

Router 包主要负责把 Handler 对应的客户端处理方法映射到具体的路由和 API 请求路径及请求方法，提供对外的 HTTP 访问，示例如图 5-30 所示。

```
apiv1 := c.Group( relativePath: "/api/v1")
{
    //node
    apiv1.POST( relativePath: "/node/create/first", handler.CreateFirstNode)
    apiv1.POST( relativePath: "/node/create/other", handler.CreateOtherNode)
    apiv1.PUT( relativePath: "/node/update", handler.UpdateNode)
    apiv1.DELETE( relativePath: "/node/delete/:id", handler.DeleteNode)
    apiv1.PUT( relativePath: "/node/start/:id", handler.StartOpenvpn)
    apiv1.GET( relativePath: "/node/subnet", handler.GetSubnet)
    apiv1.GET( relativePath: "/node/list", handler.ListNode)

    //link
    apiv1.POST( relativePath: "/link/create", handler.CreateLink)
    apiv1.GET( relativePath: "/link/check/:id", handler.CheckLink)
    apiv1.DELETE( relativePath: "/link/delete/:id", handler.DeleteLink)
    apiv1.GET( relativePath: "/link/list", handler.ListLink)

    //account
    apiv1.GET( relativePath: "/account/vericode", handler.AccountVeriCode)
    apiv1.POST( relativePath: "/account/login", handler.AccountLogin)
    apiv1.POST( relativePath: "/account/create", handler.AccountCreate)
    apiv1.PUT( relativePath: "/account/update", handler.AccountUpdate)
    apiv1.DELETE( relativePath: "/account/delete/:id", handler.AccountDelete)
    apiv1.GET( relativePath: "/account/list", handler.AccountList)

    //dhcp
    apiv1.GET( relativePath: "/dhcp/ips", handler.GetDhcpIps)
    apiv1.GET( relativePath: "/ip/search/:ip", handler.GetIpInfo)

    //upgrade
    apiv1.POST( relativePath: "/upgrade/offline", handler.UpgradeOffline)
```

图 5-30　MVC 模式下的 Router 示例

5.4　省时省力的 API 智能文档生成工具

在前后端分离的项目开发过程中，如果后端开发人员能够提供一份清晰明了的接口文档，就能极大地提高大家的沟通效率和开发效率。如何维护接口文档历来都是令人头疼的问题，感觉很浪费精力，而且后续接口文档的维护也十分耗费精力。在很多年以前，也流行用 Word 等工具写接口文档，这里面的问题很多，如格式不统一、后端人员消费精力大、文档的时效性也无法保障。针对这类问题，最好是有一种方案能够既满足输出文档的需要又能随代码的变更自动更新，Swagger 正是能帮助解决接口文档问题的工具。

RESTful 是这些年的高频词汇，各大互联网公司也都纷纷推出了自己的 RESTful

API，其实 RESTful 和 Thrift、GRPC 类似，都是一种协议，但是这种协议有点特殊，即使用 HTTP 接口，返回的对象一般是 JSON 格式，这样有个好处，就是可以供前端的 JS 直接调用，使用非常方便，但 HTTP 本身并不是一个高效的协议，后端的内部通信还是使用 GRPC 或者 Thrift，这样可以获得更高的性能。其实如果只是要用 HTTP 返回 JSON 本身并不是一件很难的事情，不用任何框架，Go 本身也能很方便地做到，但是当有很多个 API 时，这些 API 的维护和管理就会变得很复杂，自己都无法记住这些 API 应该填什么参数，以及返回什么，当然花很多时间去维护一份接口文档，这样不仅耗时而且很难保证文档的即时性、准确性及一致性。

Swagger 有一整套规范来定义一个接口文件，类似于 Thrift 和 Proto 文件，定义了服务的请求内容和返回内容，同样也有工具可以生成各种不同语言的框架代码，在 Go 语言里使用 go-swagger 工具，这个工具还提供了额外的功能，可以可视化显示这个接口，方便阅读。Swagger 是基于标准的 OpenAPI 规范进行设计的，本质上是一种使用 JSON 表示 RESTful API 接口的描述语言，只要照着这套规范去编写注解或通过扫描代码去生成注解，就能生成统一标准的接口文档和一系列 Swagger 工具。Swagger 包括自动文档、代码生成和测试用例生成。

go-swagger 参考文档如下：

官方文档：https://swagger.io/docs/specification/about/。

使用指南：https://github.com/swaggo/swag/blob/master/README_zh-CN.md。

想要使用 go-swagger 为代码自动生成接口文档，一般需要下面几个步骤：

(1) 安装 swag 工具。

(2) 按照 Swagger 的要求给接口代码添加声明式注释，具体可参照声明式注释格式。

(3) 使用 swag 工具扫描代码自动生成 API 文档数据。

(4) 使用 gin-swagger 渲染在线接口文档页面。

安装 Go 对应的开源 Swagger 相关联的库，在项目的根目录下执行的安装命令如下：

```
$ go get -u github.com/swaggo/swag/cmd/swag
$ go get -u github.com/swaggo/gin-swagger
$ go get -u github.com/swaggo/files
$ go get -u github.com/alecthomas/template
```

验证是否安装成功，命令如下：

```
$ swag -v
swag version v1.6.5
```

go get 命令分两步：第 1 步如同 git clone 拉取 GitHub 上的依赖并下载，第 2 步就是会采用 go install 编译，这个 Swagger 包比较特殊，go install 编译会编译出可执行文件，然后放在 GOBIN，因此 GOBIN 的目录选择一定要选在可读可写权限目录下，如果放在只读文件夹下，则会安装不了 Swagger 的可执行文件，并且会报错：access denied 提醒权限不够。

写入注解，在完成了 Swagger 关联库的安装后，需要针对项目里的 API 接口进行注解的编写，以便于后续在进行生成时能够正确地运行，接下来将用到如下注解：

```
注解        描述
@Summary    摘要
@Produce    API 可以产生的 MIME 类型的列表，MIME 类型可以简单地理解为响应类型，例如 JSON、XML、HTML 等
@Param      参数格式，从左到右分别为参数名、入参类型、数据类型、是否必填、注释。入参类型，可以有的值是 formData、query、path、body、header，formData 表示是 post 请求的数据，query 表示带在 url 之后的参数，path 表示请求路径上得参数，例如上面例子里面的 key，body 表示是一个 raw 数据请求，header 表示带在 header 信息中获得参数。
@Success    响应成功，从左到右分别为状态码、参数类型、数据类型、注释
@Failure    响应失败，从左到右分别为状态码、参数类型、数据类型、注释
@Router     路由，从左到右分别为路由地址和 HTTP 方法
```

添加 API 注解，切换到项目目录下的 handler 目录，打开对应的 Go 文件，写入如下注解，如查询、新增、更新、删除，示例代码分别如下：

```
//@Summary 获取多个标签
//@Produce JSON
//@Param name query string false "标签名称" maxlength(100)
//@Param state query int false "状态" Enums(0, 1) default(1)
//@Param page query int false "页码"
//@Param page_size query int false "每页数量"
//@Success 200 {object} model.Tag "成功"
//@Failure 400 {object} errcode.Error "请求错误"
//@Failure 500 {object} errcode.Error "内部错误"
//@Router /api/v1/tags [get]
func (t Tag) List(c *gin.Context) {}
```

```
//@Summary 新增标签
//@Produce JSON
//@Param name body string true "标签名称" minlength(3) maxlength(100)
//@Param state body int false "状态" Enums(0, 1) default(1)
//@Param created_by body string true "创建者" minlength(3) maxlength(100)
//@Success 200 {object} model.Tag "成功"
//@Failure 400 {object} errcode.Error "请求错误"
//@Failure 500 {object} errcode.Error "内部错误"
//@Router /api/v1/tags [post]
func (t Tag) Create(c *gin.Context) {}
```

```
//@Summary 更新标签
//@Produce JSON
//@Param id path int true "标签 ID"
//@Param name body string false "标签名称" minlength(3) maxlength(100)
//@Param state body int false "状态" Enums(0, 1) default(1)
//@Param modified_by body string true "修改者" minlength(3) maxlength(100)
//@Success 200 {array} model.Tag "成功"
//@Failure 400 {object} errcode.Error "请求错误"
//@Failure 500 {object} errcode.Error "内部错误"
```

```
//@Router /api/v1/tags/{id} [put]
func (t Tag) Update(c *gin.Context) {}
```

```
//@Summary 删除标签
//@Produce JSON
//@Param id path int true "标签 ID"
//@Success 200 {string} string "成功"
//@Failure 400 {object} errcode.Error "请求错误"
//@Failure 500 {object} errcode.Error "内部错误"
//@Router /api/v1/tags/{id} [delete]
func (t Tag) Delete(c *gin.Context) {}
```

在这里只展示了标签模块的接口注解编写，接下来应当按照注解的含义和上述接口注解，完成文章模块接口注解的编写。

main 注解：在接口方法本身有了注解以后，针对这个项目能不能写注解呢？万一有很多个项目，又该怎么办呢？实际上是可以识别出来的，只要针对 main 方法写入如下注解：

```
//@title 匿名链路系统
//@version 1.0
//@description 网络攻防中的匿名链路系统，提供目标完整的匿名访问
//@termsOfService https://blog.csdn.net/u014374009
func main() {
    ...
}
```

生成文档，在完成了所有的注解编写后，回到项目根目录下，执行的命令如下：

```
swag init
```

在执行完命令后，会发现在 docs 文件夹生成了 docs.go、swagger.json、swagger.yaml 共 3 个文件。

路由：在注解编写完后通过 swag init 把 Swagger API 所需要的文件都生成了，那接下来怎么访问接口文档呢？其实很简单，只需在 routers 中进行默认初始化和注册对应的路由就可以了，打开项目目录下的 internal/routers 目录中的 router.go 文件，新增代码如下：

```
import (
    ...
    _ "github.com/go-programming-tour-book/blog-service/docs"
    //_ 表示执行 init 函数时调用该包，需要将这个替换为自己本地的 docs 目录路径。这个路径
    //是 GitHub 上别人的 docs，此处只是用来测试
    //此处应该这样写：_ "swagger_demo/docs"
    //上面的 swagger_demo 为本项目名称，docs 就是由 swag init 自动生成的目录，用于存放 docs.go、
    //swagger.json、swagger.yaml 文件
    ginSwagger "github.com/swaggo/gin-swagger"
    "github.com/swaggo/gin-swagger/swaggerFiles"
)
```

```
func NewRouter() *gin.Engine {
    r := gin.New()
    r.Use(gin.Logger())
    r.Use(gin.Recovery())
    r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
    ...
    return r
}
```

从表面上来看，主要做了两件事情，分别是初始化 docs 包和注册一个针对 Swagger 的路由，而在初始化 docs 包后，其 swagger.json 将会默认指向当前应用所启动的域名下的 swagger/doc.json 路径，如果有额外需求，则可进行手动指定，代码如下：

```
url := ginSwagger.url("http://127.0.0.1:8000/swagger/doc.json")
r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler, url))
```

查看接口文档，在完成了上述的设置后，重新启动服务器端，在浏览器中访问 Swagger 的地址 http://127.0.0.1:8000/swagger/index.html，这样就可以看到上述图片中的 Swagger 文档展示，其主要分为 3 部分，分别是项目主体信息、接口路由信息、模型信息，这 3 部分共同组成了主体内容。

对 Swagger 生成的 API 文档进行查看和举例，整体接口文档按照分组进行展示，如图 5-31 和图 5-32 所示。每个接口的请求参数都有相关的数据类型和说明。每个 API 都可以进行在线调试，如图 5-33 所示。

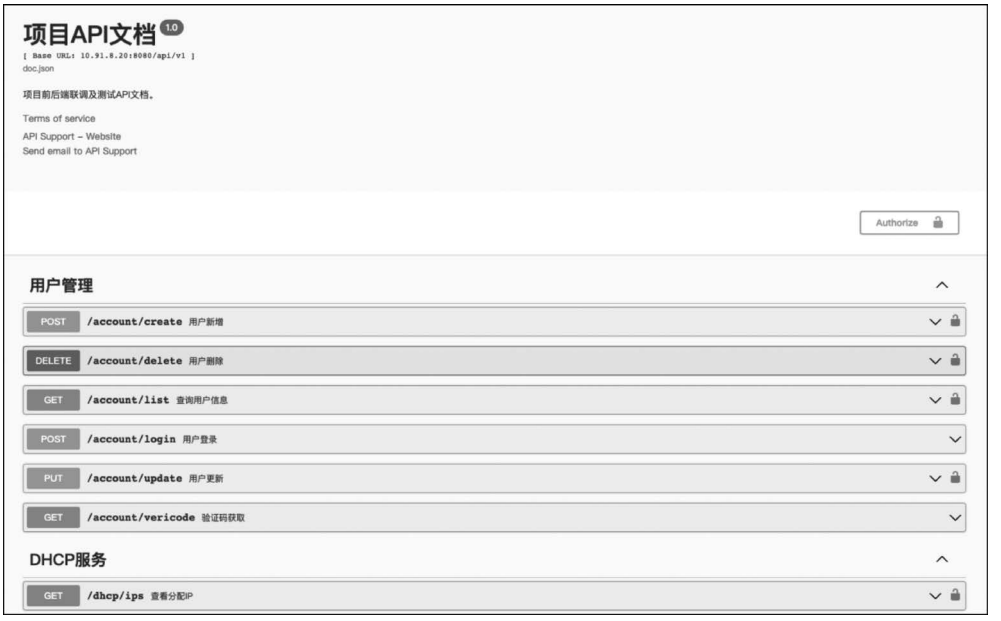


图 5-31　Swagger 生成的 API 文档示例 1

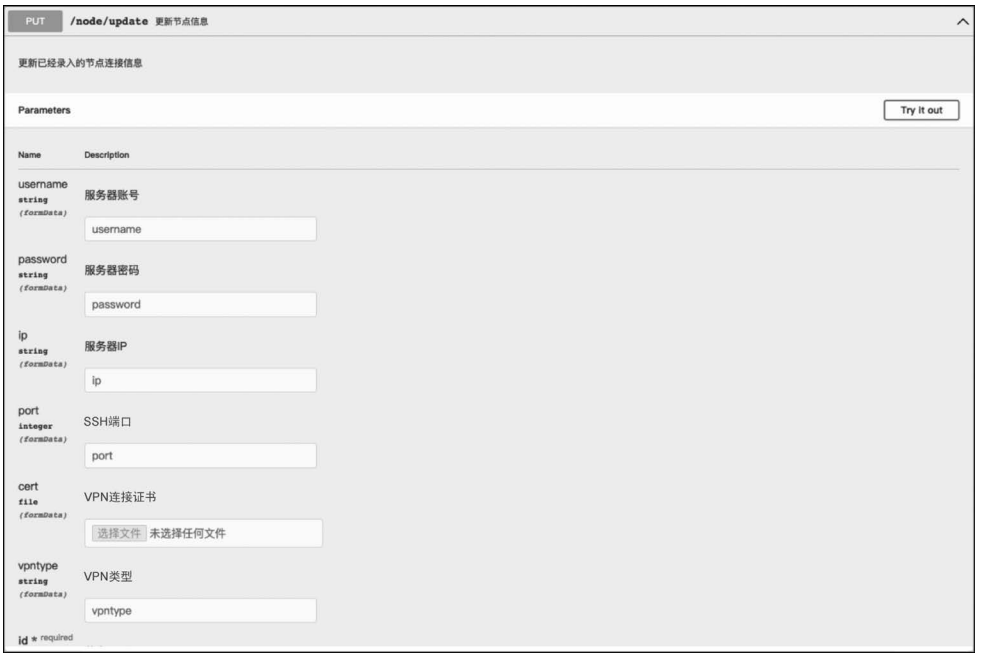


图 5-32　Swagger 生成的 API 文档示例 2

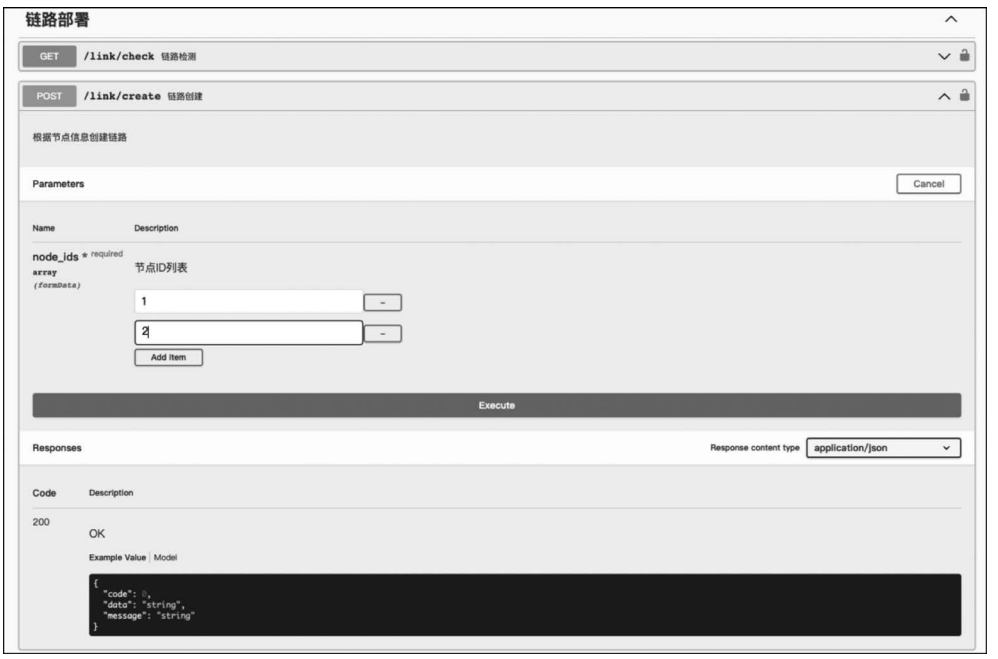


图 5-33　Swagger 在线 API 文档调试

5.5 Web 中间件及请求拦截器的使用

在 Web 应用服务中，完整的一个业务处理在技术上包含客户端操作、服务器端处理、将处理结果返回客户端 3 个步骤。

在实际的业务开发和处理中，会有更负责的业务和需求场景。一个完整的系统可能包含对鉴权认证、权限管理、安全检查、日志记录等多维度的系统支持。

鉴权认证、权限管理、安全检查、日志记录等这些保障和支持系统业务属于全系统的业务，和具体的系统业务没有关联，对于系统中的所有业务都适用。

由此，在业务开发过程中，为了更好地梳理系统架构，可以将上述描述所涉及的一些通用业务单独抽离出来并进行开发，然后以插件的形式进行对接。这种方式既保证了系统功能的完整，同时又有效地将具体业务和系统功能解耦，并且还可以达到灵活配置的目的。

这种通用业务独立开发并灵活配置使用的组件一般被称为“中间件”，因为其位于服务器和实际业务处理程序之间。其含义相当于在请求和具体的业务逻辑处理之间增加某些操作，这种以额外添加的方式不会影响编码效率，也不会侵入框架中。

中间件也叫拦截器或者过滤器，本质上都是在一个 HTTP 请求被处理之前执行的一段代码，Gin 的中间件是一个函数，函数签名和 Gin 的路由处理函数一致，即都是 func(* Context)类型。

一些比较流行的中间件框架如下：

```
+ [RestGate](https://github.com/pjebs/restgate) - Secure authentication for REST API endpoints

+ [staticbin](https://github.com/olebedev/staticbin) - middleware/handler for serving static files from binary data

+ [gin-cors](https://github.com/gin-contrib/cors) - Official CORS gin's middleware

+ [gin-csrf](https://github.com/utrack/gin-csrf) - CSRF protection

+ [gin-health](https://github.com/utrack/gin-health) - middleware that simplifies stat reporting via [gocraft/health](https://github.com/gocraft/health)

+ [gin-merry](https://github.com/utrack/gin-merry) - middleware for pretty-printing [merry](https://github.com/ansel1/merry) errors with context

+ [gin-revision](https://github.com/appleboy/gin-revision-middleware) - Revision middleware for Gin framework

+ [gin-jwt](https://github.com/appleboy/gin-jwt) - JWT Middleware for Gin Framework

+ [gin-sessions](https://github.com/kimiazhu/ginweb-contrib/tree/master/sessions) - session middleware based on MongoDB and mysql
```

```
+ [gin-location](https://github.com/drone/gin-location) - middleware for exposing the server's hostname and scheme

+ [gin-nice-recovery](https://github.com/ekyoung/gin-nice-recovery) - panic recovery middleware that lets you build a nicer user experience

+ [gin-limiter](https://github.com/davidleitw/gin-limiter) - A simple gin middleware for ip limiter based on redis.

+ [gin-limit](https://github.com/aviddiviner/gin-limit) - limits simultaneous requests; can help with high traffic load

+ [gin-limit-by-key](https://github.com/yangxikun/gin-limit-by-key) - An in-memory middleware to limit access rate by custom key and rate.

+ [ez-gin-template](https://github.com/michelloworld/ez-gin-template) - easy template wrap for gin

+ [gin-hydra](https://github.com/janekolszak/gin-hydra) - [Hydra](https://github.com/ory-am/hydra) middleware for Gin

+ [gin-glog](https://github.com/zalando/gin-glog) - meant as drop-in replacement for Gin's default logger

+ [gin-gomonitor](https://github.com/zalando/gin-gomonitor) - for exposing metrics with Go-Monitor

+ [gin-oauth2](https://github.com/zalando/gin-oauth2) - for working with OAuth2

+ [static](https://github.com/hyperboloide/static) An alternative static assets handler for the gin framework.

+ [xss-mw](https://github.com/dvwright/xss-mw) - XssMw is a middleware designed to "auto remove XSS" from user submitted input

+ [gin-helmet](https://github.com/danielkov/gin-helmet) - Collection of simple security middleware.

+ [gin-jwt-session](https://github.com/ScottHuangZL/gin-jwt-session) - middleware to provide JWT/Session/Flashes, easy to use while also provide options for adjust if necessary. Provide sample too.

+ [gin-template](https://github.com/foolin/gin-template) - Easy and simple to use html/template for gin framework.

+ [gin-redis-ip-limiter](https://github.com/Salvatore-Giordano/gin-redis-ip-limiter) - Request limiter based on ip address. It works with redis and with a sliding-window mechanism.

+ [gin-method-override](https://github.com/bu/gin-method-override) - Method override by POST form param `_method`, inspired by Ruby's same name rack
```

```
+ [gin-access-limit](https://github.com/bu/gin-access-limit) - An access-control middleware by specifying allowed source CIDR notations.

+ [gin-session](https://github.com/go-session/gin-session) - Session middleware for Gin

+ [gin-stats](https://github.com/semihalev/gin-stats) - Lightweight and useful request metrics middleware

+ [gin-statsd](https://github.com/amalfra/gin-statsd) - A Gin middleware for reporting to statsd deamon

+ [gin-health-check](https://github.com/RaMin0/gin-health-check) - A health check middleware for Gin

+ [gin-session-middleware](https://github.com/go-session/gin-session) - A efficient, safely and easy-to-use session library for Go.

+ [ginception](https://github.com/kubastick/ginception) - Nice looking exception page

+ [gin-inspector](https://github.com/fatihkahveci/gin-inspector) - Gin middleware for investigating http request.

+ [gin-dump](https://github.com/tpkeeper/gin-dump) - Gin middleware/handler to dump header/body of request and response. Very helpful for Debugging your applications.

+ [go-gin-prometheus](https://github.com/zsais/go-gin-prometheus) - Gin Prometheus metrics exporter

+ [ginprom](https://github.com/chenjiandongx/ginprom) - Prometheus metrics exporter for Gin

+ [gin-go-metrics](https://github.com/bmc-toolbox/gin-go-metrics) - Gin middleware to gather and store metrics using [rcrowley/go-metrics](https://github.com/rcrowley/go-metrics)

+ [ginrpc](https://github.com/xxjwxc/ginrpc) - Gin middleware/handler auto binding tools. support object register by annotated route like beego

+ [goscope](https://github.com/averageflow/goscope) - Watch incoming requests, outgoing responses and logs of your Gin application with this plug and play middleware inspired by Laravel Telescope.

+ [gin-nocache](https://github.com/alexander-melentyev/gin-nocache) - NoCache is a simple piece of middleware that sets a number of HTTP headers to prevent a router (or subrouter) from being cached by an upstream proxy and/or client.

+ [logging](https://github.com/axiaoxin-com/logging # gin-middleware-ginlogger) - logging provide GinLogger uses zap to log detailed access logs in JSON or text format with trace id, supports flexible and rich configuration, and supports automatic reporting of log events above error level to sentry
```

```
+ [ratelimiter](https://github.com/axiaoxin-com/ratelimiter) - Gin middleware for token bucket ratelimiter.

+ [servefiles](https://github.com/rickb777/servefiles) - serving static files with performance-enhancing cache control headers; also handles gzip & brotli compressed files
```

在 Gin 框架中，中间件(Middleware)指的是可以拦截 HTTP 请求-响应生命周期的特殊函数，在请求-响应生命周期中可以注册多个中间件，每个中间件提供不同的功能，一个中间执行完再轮到下一个中间件执行。

中间件的作用如下。

(1) 在请求到达 HTTP 请求处理方法之前拦截请求：①认证；②权限校验；③限流；④数据过滤 ；⑤IP 白名单。

(2) 处理完请求后，拦截响应，并进行相应处理：①统一添加响应头；②数据过滤；③中间件加的位置；④全局加；⑤路由组加；⑥路由明细加。

Gin 中的默认中间件；

(1) 默认使用了 Logger()和 Recovery()，全局作用了这两个中间件，方法如下：

```
r := gin.Default()

func Default() *Engine {
    DebugPrintWARNINGDefault()
    engine := New()
    engine.Use(Logger(), Recovery())
    return engine
}
```

(2) Gin 默认自带了一些中间件，函数如下：

```
func BasicAuth(accounts Accounts) HandlerFunc
func BasicAuthForRealm(accounts Accounts, realm string) HandlerFunc
func Bind(val interface{}) HandlerFunc          //拦截请求参数并进行绑定
func ErrorLogger() HandlerFunc                  //错误日志处理
func ErrorLoggerT(typ ErrorType) HandlerFunc    //自定义类型的错误日志处理
func Logger() HandlerFunc                       //日志记录
func LoggerWithConfig(conf LoggerConfig) HandlerFunc
func LoggerWithFormatter(f LogFormatter) HandlerFunc
func LoggerWithWriter(out io.Writer, notlogged ...string) HandlerFunc
func Recovery() HandlerFunc
func RecoveryWithWriter(out io.Writer) HandlerFunc
func WrapF(f http.HandlerFunc) HandlerFunc      //将 http.HandlerFunc 包装成中间件
func WrapH(h http.Handler) HandlerFunc          //将 http.Handler 包装成中间件
```

如何去除默认中间件，以及如何去除默认全局中间件，代码如下：

```
r := gin.New()//不带中间件
```

如何添加全局中间件，代码如下：

```
func main() {
    r := gin.Default()

    r.Use(func(c *gin.Context) {
        fmt.Println("hello start")
    })

    r.GET("/", func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })

    r.Run()
}
```

或者实现代码如下：

```
func M1(c *gin.Context) {
    fmt.Println("hello start")
}

func main() {
    r := gin.Default()

    r.Use(M1)

    r.GET("/", func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })

    r.Run()
}
```

如何在路由分组中使用中间件，代码如下：

```
func main() {
    r := gin.Default()
    v1 := r.Group("/v1", gin.Logger(), gin.Recovery())
    {
        v1.GET("/", func(c *gin.Context) {
            c.json(200, gin.H{"name": "m1"})
        })
        v1.GET("/test", func(c *gin.Context) {
            c.json(200, gin.H{"name": "m1 test"})
        })
    }

    r.Run()
}
```

单个路由使用中间件，代码如下：

```
func main() {
    r := gin.Default()
    r.GET("/", gin.Recovery(), gin.Logger(), func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })
    r.Run()
}
```

自定义中间件，代码如下：

```
func MyMiddleware(c *gin.Context) {
    //中间件逻辑
    fmt.Println("hello")
}

func main() {
    r := gin.Default()

    r.Use(MyMiddleware)

    r.GET("/", func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })
    r.Run()
}
```

拦截器主要实现逻辑，代码如下：

```
func MyMiddleware(c *gin.Context){
    //请求前逻辑
    c.Next()
    //请求后逻辑
}
```

Gin 内置的几个中断用户请求的方法：返回 200，但 body 里没有数据，函数如下：

```
func (c *Context) Abort()
func (c *Context) AbortWithError(code int, err error) *Error
func (c *Context) AbortWithStatus(code int)

func (c *Context) AbortWithStatusJSON(code int, jsonObj interface{}) //中断请求后，返回 JSON
//格式的数据
```

中断用户请求，示例代码如下：

```
func MyMiddleware(c *gin.Context) {
    c.Set("key", 1000)                          //请求前
    c.Next()
```

```
    c.json(http.StatusOK, c.GetInt("key"))            //请求后
}
func main() {
    r := gin.New()
    r.GET("test", MyMiddleware, func(c *gin.Context) {
        k := c.GetInt("key")
        c.Set("key", k + 2000)
    })
    r.Run()
}
```

实现自定义中间件的步骤如下。

方法 1：自定义中间件逻辑处理不返回数据，示例代码如下：

```
func MyMiddleware(c *gin.Context) {
    //中间件逻辑
    fmt.Println("hello")
}

func main() {
    r := gin.Default()

    r.Use(MyMiddleware)

    r.GET("/", func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })
    r.Run()
}
```

方法 2：返回一个中间件函数，示例代码如下：

```
//Gin 框架自带的中间件方法都返回 HandlerFunc 类型
type HandlerFunc func(*Context)


func MyMiddleware() func(c *gin.Context) {
    //自定义逻辑
    fmt.Println("requesting...")              //中间件不打印
    //返回中间件
    return func(c *gin.Context) {
        //中间件逻辑
        fmt.Println("test2")
    }
}


func main() {
    r := gin.Default()
```

```
    r.Use(MyMiddleware())                    //加括号

    r.GET("/", func(c *gin.Context) {
        c.json(200, gin.H{"name": "m1"})
    })
    r.Run()
}
```

中间件实现数据的传递，示例代码如下：

```
func MyMiddleware(c *gin.Context) {
    c.Set("mykey", 10)
    c.Set("mykey2", "m1")
}
func main() {
    //自定义中间件
    r := gin.New()
    r.GET("", MyMiddleware, func(c *gin.Context) {
        mykey := c.GetInt("mykey") //知道设置的是整型，所以使用 GetInt 方法获取
        mykey2 := c.GetString("mykey2")
        c.json(200, gin.H{
            "mykey": mykey,
            "mykey2": mykey2,
        })
    })
    r.Run()
}
```

Gin 框架中间件中 set 和 get 用于存取参数，主要的函数如下：

```
func (c *Context) Set(key string, value interface{})
//判断 key 是否存在 c.Get
func (c *Context) Get(key string) (value interface{}, exists bool)
func (c *Context) GetBool(key string) (b bool)
func (c *Context) GetDuration(key string) (d time.Duration)
func (c *Context) GetFloat64(key string) (f64 float64)
func (c *Context) GetInt(key string) (i int)
func (c *Context) GetInt64(key string) (i64 int64)
func (c *Context) GetString(key string) (s string)
func (c *Context) GetStringMap(key string) (sm map[string]interface{})
func (c *Context) GetStringMapString(key string) (sms map[string]string)
func (c *Context) GetStringMapStringSlice(key string) (smss map[string][]string)
func (c *Context) GetStringSlice(key string) (ss []string)
func (c *Context) GetTime(key string) (t time.Time)
func (c *Context) MustGet(key string) interface{} //必须有，否则会 panic
```

使用 gin.BasicAuth 中间件，示例代码如下：

```
//anonymous - link\example\chapter5\web\basic_auth.go
package main
```

```
import (
    "github.com/gin-gonic/gin"
    "net/http"
)

//type HandlerFunc func(*Context)

//模拟一些私人数据
var secrets = gin.H{
    "foo": gin.H{"email": "foo@bar.com", "phone": "123433"},
    "austin": gin.H{"email": "austin@example.com", "phone": "666"},
    "lena": gin.H{"email": "lena@guapa.com", "phone": "523443"},
}

func main() {

    r := gin.Default()

    r.GET("/", func(c *gin.Context) {
        c.json(200, secrets)
        //c.String(200, "index")
    })
    //为/admin 路由组设置 auth
    //路由组使用 gin.BasicAuth() 中间件
    //gin.Accounts 是 map[string]string 的一种快捷方式
    authorized := r.Group("/admin", gin.BasicAuth(gin.Accounts{
        "foo": "bar",
        "austin": "1234",
        "lena": "hello2",
        "manu": "4321",
    }))

    //admin/secrets 端点
    //触发 localhost:8080/admin/secrets
    authorized.GET("/secrets", func(c *gin.Context) {
        //获取用户,它是由 BasicAuth 中间件设置的
        user := c.MustGet(gin.AuthUserKey).(string)
        if secret, ok := secrets[user]; ok {
            c.json(http.StatusOK, gin.H{"user": user, "secret": secret})
        } else {
            c.json(http.StatusOK, gin.H{"user": user, "secret": "NO SECRET :("})
        }
    })

    //监听并在 0.0.0.0:8080 上启动服务
    r.Run(":8080")
}
```

5.6 快速实现应用及接口的请求鉴权

JWT 的原理和 Session 有点相像,其目的是解决 RESTful API 中无状态性。因为 RESTful 接口,需要权限校验,但是又不能让每个请求都把用户名和密码传入,因此产生了

这个 Token 的方法。

流程如下：

(1) 用户访问 auth 接口，获取 Token。服务器校验用户传入的用户名和密码等信息，确认无误后，产生一个 Token。这个 Token 其实类似于 map 的数据结构(JWT 数据结构)中的 key。

其本质就是 Token 中其实保存了用户的信息，只是被加密过了。就算服务器重启了 Token 还能使用，就是这个原因，因为数据被保存在 Token 这条长长的字符串中。

(2) 用户访问需要权限验证的接口，并传入 Token。

(3) 服务器验证 Token：根据自己的 Token 密钥判断 Token 是否正确(是否被别人篡改)，正确后才从 Token 中解析出 Token 中的信息，可能会把解析出的信息保存在 context 中。

基于证书和 JWT 验证示例如图 5-34 和图 5-35 所示。

```
var WhitelistAPI = map[string]bool{
    "/api/v1/account/login":    true,
    "/api/v1/account/vericode": true,
}

func Validate() gin.HandlerFunc {
    return func(c *gin.Context) {
        // 签名校验
        err, checkSign := utils.DecodeLicenseV1( license: "",  projectKey: "",  guid: "")
        if !checkSign {
            c.Abort()
            c.JSON(http.StatusInternalServerError, gin.H{
                "code":    2,
                "message": "无效的证书",
                "data":    err,
            })
            return
        }

        //Token校验
        if !WhitelistAPI[c.Request.RequestURI] {
            err := jwtVerify(c)
            if err != nil {
                c.Set( key: "account_info",  value: nil)
                c.Abort()
                c.JSON(http.StatusInternalServerError, gin.H{
                    "code":    3,
                    "message": "无效的Token",
                    "data":    err,
                })
                return
            }
        }
        c.Next()
    }
}
```

图 5-34　Gin 中证书和 Token 校验流程示例

```go
//验证Token
func jwtVerify(c *gin.Context) error {
    token := c.GetHeader( key: "token")
    if token == "" : fmt.Errorf("token not exist")

    //验证Token, 并存储在请求中
    return parseToken(token, c)
}

// 解析Token
func parseToken(token string, c *gin.Context) error {
    data, err := utils.DecryptByAes(token)
    if err != nil : fmt.Errorf("invalid token")
    var account model.AccountModel
    err = json.Unmarshal(data, &account)
    if err != nil : fmt.Errorf("invalid token")
    c.Set( key: "account_info", account)
    if time.Now().Unix()-account.UpdatedAt.Unix() > 30*60 {
        c.Header( key: "token",  value: "")
    } else if time.Now().Unix()-account.UpdatedAt.Unix() > 20*60 : refreshToken(c)
    return nil
}

// 更新Token
func refreshToken(c *gin.Context) error {
    user, exists := c.Get( key: "account_info")
    if !exists : fmt.Errorf("invalid token")
    account := user.(model.AccountModel)
    err, result := service.AccountLogin(account.Username, account.Password)
    if err != nil : fmt.Errorf("invalid token")
    c.Header( key: "token", result)
    return nil
}
```

图 5-35　Gin 中 Token 校验及自动更新

5.7　封装统一的参数传输及异常处理

Gin 框架中接受 Web 请求中的参数，有多种方式可以获取异常处理，为了统一格式和处理异常，基于 Gin 的 Context 方法属性扩展将函数请求参数的处理封装为一个通用的中间件，核心方法的示例代码如下：

```go
//anonymous - link\example\chapter5\web\args_deal.go
package base

import (
    "github.com/gin-gonic/gin"
)

func Args() gin.HandlerFunc {
    return func(c *gin.Context) {
```

```
var requestParams = make(map[string]interface{})
form, err := c.MultipartForm()
if err == nil {
    file := form.File
    for k, v := range file {
        var names []string
        for _, f := range v {
            names = append(names, f.Filename)
        }
        c.Set(k, names)
        requestParams[k] = names
    }
}
Bytes, err := ioutil.ReadAll(c.Request.Body)
if err == nil && Bytes != nil {
    maps := make(map[string]interface{})
    err = json.Unmarshal(Bytes, &maps)
    if err == nil {
        for k, v := range maps {
            c.Set(k, v)
            requestParams[k] = v
        }
    } else {
        params := strings.Split(string(Bytes), "&")
        for _, param := range params {
            if strings.Contains(param, "=") {
                arr := strings.Split(param, "=")
                key, _ := url.QueryUnescape(arr[0])
                val, _ := url.QueryUnescape(arr[1])
                c.Set(key, val)
                requestParams[key] = val
            }
        }
    }
}
query := c.Request.Form.Encode()
if len(query) > 1 {
    params := strings.Split(query, "&")
    for _, param := range params {
        if len(param) > 1 && strings.Contains(param, "=") {
            arr := strings.Split(param, "=")
            key, _ := url.QueryUnescape(arr[0])
            val, _ := url.QueryUnescape(arr[1])
            c.Set(key, val)
            requestParams[key] = val
        }
    }
}
pathArr := c.Params
for i := 0; i < len(pathArr); i++{
    c.Set(pathArr[i].Key, pathArr[i].Value)
```

```
            requestParams[pathArr[i].Key] = pathArr[i].Value
        }
        c.Request.ParseForm()
        for k, v := range c.Request.PostForm {
            val := strings.Join(v, ",")
            c.Set(k, val)
            requestParams[k] = val
        }

        if ! CheckPageParam(c) {
            c.Abort()
            c.json(http.StatusOK, gin.H{
                "code": 5,
                "message": "查询参数不合法",
                "data": "the query parameter is not legal",
            })
            return
        }

        c.Set("request_params", requestParams)
        c.Next()
    }
}

func CheckPageParam(c *gin.Context) bool {
    data, exists := c.Get("page_no")
    if exists && data != nil {
        result, e := strconv.ParseUint(fmt.Sprint(data), 10, 64)
        if e == nil {
            if result < 1 {
                return false
            }
        } else {
            return false
        }
    }
    data, exists = c.Get("page_size")
    if exists && data != nil {
        result, e := strconv.ParseUint(fmt.Sprint(data), 10, 64)
        if e == nil {
            if result < 1 || result > 1000 {
                return false
            }
        } else {
            return false
        }
    }
    return true
}
```

完整代码可查看以下目录文件：

```
anonymous - link\code\base\auth.go
anonymous - link\code\base\context.go
```

在 handler 中使用封装的中间件对参数进行获取，核心方法的示例代码如下：

```
//anonymous - link\example\chapter5\web\handler_demo.go
package handler

import (
    "base"
    "service"
    "time"
    "utils"
)

//AccountCreate
//@Summary 用户新增
//@Description 添加用户信息
//@Tags 用户管理
//@schemes http https
//@Accept json
//@Produce json
//@Security ApiKeyAuth
//@Response 200 {object} config.Response
//@Param username formData string true "用户账号"
//@Param nickname formData string true "用户名称"
//@Param password formData string true "用户密码"
//@Param status formData string true "用户状态"
//@Router /account/create [post]
func AccountCreate(c *base.Context) {
    username := c.ArgsString("username")
    password := c.ArgsString("password")
    nickname := c.ArgsString("nickname")
    status := c.ArgsUint("status")
    if err := service.CheckUsername(username); err != nil {
        SendResponse(c, err, nil)
        return
    }
    if err := service.CheckPassword(password); err != nil {
        SendResponse(c, err, nil)
        return
    }
    nickname, err := service.CheckNickname(nickname)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }
    account := model.AccountModel{
        Username: username,
        Nickname: nickname,
```

```
            Password: utils.String2Md5(password),
            Status: status,
            Role: 1,
            LastLogin: time.Now(),
        }
        account.CreatedAt = time.Now()
        account.UpdatedAt = time.Now()
        err, data := account.Create()
        if err != nil {
            SendResponse(c, err, nil)
            return
        }
        SendResponse(c, nil, data)
        return
    }
```

完整代码可查看以下目录及文件：

```
anonymous - link\code\handler\account_manage.go
anonymous - link\code\service\account_manage.go
anonymous - link\code\utils\
```

5.8 自定义中间件实现 AOP 式日志记录

为记录程序的操作记录及日志追踪，通过 Gin 的 Context 编写扩展中间件，做成一个 AOP 式的非侵入式的通用模块，结合并发异步任务通过 channel 传输数据，以及并发日志记录，核心方法的示例代码如下：

```
func LogAop() gin.HandlerFunc {
    return func(c *gin.Context) {
        startTime := time.Now()
        blw := &bodyLogWriter{body: Bytes.NewBufferString(""), ResponseWriter: c.Writer}
        c.Writer = blw
        c.Next()
        endTime := time.Now()

        var logInfo = make(map[string]interface{})
        logInfo["method"] = c.Request.Method
        logInfo["execute_time"] = time.Now()
        logInfo["content_length"] = c.Request.ContentLength
        logInfo["content_type"] = c.ContentType()
        logInfo["cost_time"] = endTime.Sub(startTime).Milliseconds()
        logInfo["request_url"] = c.Request.RequestURI
        logInfo["status_code"] = c.Writer.Status()
        logInfo["request_host"] = c.Request.Host
        logInfo["user_agent"] = c.Request.UserAgent()
        //ip, _ := c.RemoteIP()
```

```
            ip := c.Request.Header.Get("X-Real-IP")
            logInfo["remote_ip"] = ip
            logInfo["remote_addr"] = c.Request.RemoteAddr
            apipath := strings.Split(c.FullPath(), "/:")[0]
            logInfo["api_path"] = apipath
            logInfo["referer"] = c.Request.Referer()
            data := blw.body.String()
            if len(data) > 5000 {
                data = data[:5000]
            }
            logInfo["response_data"] = data
            accountInfo, _ := c.Get("account_info")
            account, _ := json.Marshal(accountInfo)
            logInfo["account_info"] = string(account)
            requestParams, _ := c.Get("request_params")
            params, _ := json.Marshal(requestParams)
            logInfo["request_params"] = string(params)
            logInfo["api_desc"] = ApiDesc[apipath]

            logData := config.LogData{
                LogInfo:       logInfo,
                RequestParams: requestParams,
                AccountInfo:   accountInfo,
            }
            config.LogSyncChan <- logData
        }
    }
```

以上日志数据没有被保存到数据，而是写入了 channel，这样可以再起一个协程操作，即把日志并发保存到数据，在 config 包下定义了一个全局的 channel，代码如下：

```
LogSyncChan = make(chan interface{}, 100)
```

为了便于阅读和展示，可以在保存日志的过程中，对日志的数据进行富化处理，例如对相关的操作状态和数据进行适当翻译，核心方法的示例代码如下：

```
func ApiLog(apipath string, params map[string]interface{}) map[string]string {
    accountMap := map[string]model.AccountModel{}
    whitelistMap := map[string]model.WhitelistModel{}
    nodeMap := map[string]model.NodeModel{}
    subnetMap := map[string]model.SubnetModel{}
    linkMap := map[string]model.LinkModel{}
    strategyMap := map[string]model.StrategyModel{}

    if strings.HasPrefix(apipath, "/api/v1/account") {
        accountList, _, _ := model.SearchAccount(0, 0, "", "id desc")
        for _, account := range accountList {
            accountMap[fmt.Sprint(account.Id)] = account
        }
    }
```

```
    if strings.HasPrefix(apipath, "/api/v1/node/subnet") {
        infoList, _, _ := model.SearchSubnet(0, 0, "", "id desc")
        for _, info := range infoList {
            subnetMap[fmt.Sprint(info.Id)] = info
        }
    } else if strings.HasPrefix(apipath, "/api/v1/node") {
        infoList, _, _ := model.SearchNode(0, 0, "", "id desc")
        for _, info := range infoList {
            nodeMap[fmt.Sprint(info.Id)] = info
        }
    }

    if strings.HasPrefix(apipath, "/api/v1/link") {
        infoList, _, _ := model.SearchLink(0, 0, "", "id desc")
        for _, info := range infoList {
            linkMap[fmt.Sprint(info.Id)] = info
        }
    }

    if strings.HasPrefix(apipath, "/api/v1/whitelist") {
        whiteList, _, _ := model.SearchWhitelist(0, 0, "", "id desc")
        for _, white := range whiteList {
            whitelistMap[fmt.Sprint(white.Id)] = white
        }
    }

    if strings.HasPrefix(apipath, "/api/v1/strategy") {
        strategyList, _ := model.SearchStrategy(0, 0, "")
        for _, strate := range strategyList {
            strategyMap[fmt.Sprint(strate.Id)] = strate
        }
    }

    apiFunc := map[string]map[string]string{
        //用户管理
        "/api/v1/account/create": {"page": "用户管理", "type": "创建", "remark": fmt.
Sprintf("创建了账号:%s", params["username"])},
        "/api/v1/account/vericode": {"page": "用户管理", "type": "查询", "remark": fmt.
Sprintf("查询了验证码")},
        "/api/v1/account/login": {"page": "用户管理", "type": "查询", "remark": fmt.
Sprintf("登录了账号,用户名:%s", params["username"])},
        "/api/v1/account/password": {"page": "用户管理", "type": "修改", "remark": fmt.
Sprintf("修改了密码")},
        "/api/v1/account/update": {"page": "用户管理", "type": "修改", "remark": fmt.
Sprintf("修改了用户信息,")},
        "/api/v1/account/generate/username": {"page": "用户管理", "type": "查询",
"remark": fmt.Sprintf("生成了随机用户账号")},
        "/api/v1/account/delete": {"page": "用户管理", "type": "删除", "remark": fmt.
Sprintf("删除了账号 ID 为%s,名称为%s", getString(params["id"]), accountMap[fmt.Sprint
(params["id"])].Username)},
        "/api/v1/account/list": {"page": "用户管理", "type": "查询", "remark": fmt.Sprintf
("查询了用户列表信息")},
```

```
        "/api/v1/account/generate/password": {"page": "用户管理", "type": "查询",
"remark": fmt.Sprintf("获得了随机用户密码")},
        "/api/v1/account/status": {"page": "用户管理", "type": "修改", "remark": fmt.
Sprintf("修改了用户状态,用户:%s", accountMap[fmt.Sprint(params["id"])].Username)},
        "/api/v1/account/logout": {"page": "用户管理", "type": "修改", "remark": fmt.
Sprintf("登出了账号")},
        "/api/v1/account/field": {"page": "用户管理", "type": "查询", "remark": fmt.
Sprintf("查询了表字段注释")},
    }
    return apiFunc[apipath]
}

type bodyLogWriter struct {
    gin.ResponseWriter
    body *bytes.Buffer
}

func (w bodyLogWriter) Write(b []byte) (int, error) {
    w.body.Write(b)
    return w.ResponseWriter.Write(b)
}

func getString(s interface{}) string {
    if s != nil {
        return fmt.Sprint(s)
    } else {
        return ""
    }
}

func getStatus(s interface{}) string {
    result, e := strconv.Atoi(fmt.Sprint(s))
    if e == nil {
        if result == 0 {
            return "停用"
        } else {
            return "启用"
        }
    }
    return "未知"
}

func getAnyString(desc string, keys ...interface{}) string {
    var result []string
    for _, key := range keys {
        if key != nil {
            result = append(result, fmt.Sprint(key))
        }
    }
    if len(result) > 0 {
        return fmt.Sprintf(",%s%s", desc, strings.Join(result, "、"))
```

```
        } else {
            return ""
        }
    }

    func getKeyword(keys ...interface{}) string {
        var result []string
        for _, key := range keys {
            if key != nil {
                result = append(result, fmt.Sprint(key))
            }
        }
        if len(result) > 0 {
            return fmt.Sprintf(",关键词为%s", strings.Join(result, "、"))
        } else {
            return ""
        }
    }

    func getJoinString(ss ...interface{}) string {
        var result []string
        for _, s := range ss {
            if s != nil {
                result = append(result, fmt.Sprint(s))
            }

        }
        if len(result) > 0 {
            return strings.Join(result, "、")
        } else {
            return "空"
        }
    }
```

在 main.go 启动文件中定义一个接受 channel 数据的函数，并保存到数据库，示例代码如下：

```
    func main() {
        go task.SyncLogData()

        gin.SetMode(mode)
        //Create the Gin engine.
        //g := gin.New()
        g := base.NewServer()
        g.Engine.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
        //Routes.
        router.Load(
            g,
            base.Cors(),
            base.Args(),
```

```
            base.Validate(),
            base.LogAop(),
        )
        go func() {
            zap.L().Info(http.ListenAndServe(addr, g).Error())
        }()
        <-exitChan
    }


    func SyncLogData() {
        for {
            select {
            case data, ok := <-config.LogSyncChan:
                if ok {
                    logData := data.(config.LogData)
                    base.SaveLog(logData.LogInfo, logData.RequestParams, logData.AccountInfo)
                }

            default:
                time.Sleep(1 * time.Second)
            }
        }
    }
```

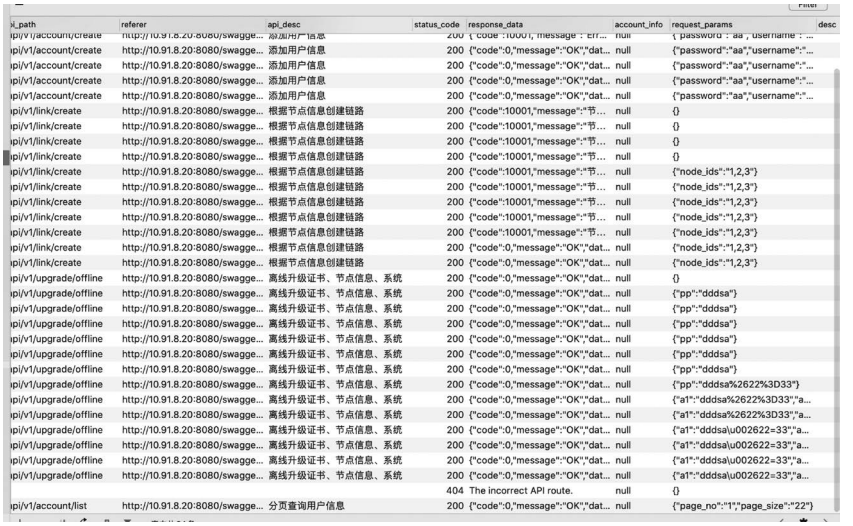
完整代码可查看以下目录及文件：

```
anonymous-link\code\base\log.go
anonymous-link\code\main.go
```

程序启动之后，随机地访问一些 API，打开数据库，发现已经自动记录了相关的请求及参数，如图 5-36 和图 5-37 所示。

id	is_del	created_at	updated_at	deleted_at	method	content_length	execute_time	content_type	cost_time	request_url
4	0	2022-07-14 10:23:57	2022-07-14 10:23:57	NULL	POST	26	2022-07-14 10:23:57	application/json	4	/api/v1/account/create
5	0	2022-07-14 10:28:23	2022-07-14 10:28:23	NULL	POST	26	2022-07-14 10:28:23	application/json	3	/api/v1/account/create
6	0	2022-07-14 10:36:20	2022-07-14 10:36:20	NULL	POST	26	2022-07-14 10:36:20	application/json	3	/api/v1/account/create
7	0	2022-07-14 10:40:09	2022-07-14 10:40:09	NULL	POST	26	2022-07-14 10:40:09	application/json	3	/api/v1/account/create
8	0	2022-07-14 10:53:40	2022-07-14 10:53:40	NULL	POST	14	2022-07-14 10:53:40	application/json	0	/api/v1/link/create
9	0	2022-07-14 10:55:14	2022-07-14 10:55:14	NULL	POST	14	2022-07-14 10:55:14	application/json	0	/api/v1/link/create
10	0	2022-07-14 10:59:46	2022-07-14 10:59:46	NULL	POST	14	2022-07-14 10:59:46	application/json	0	/api/v1/link/create
11	0	2022-07-14 11:02:24	2022-07-14 11:02:24	NULL	POST	14	2022-07-14 11:02:24	application/json	0	/api/v1/link/create
12	0	2022-07-14 11:06:05	2022-07-14 11:06:05	NULL	POST	14	2022-07-14 11:06:05	application/json	0	/api/v1/link/create
13	0	2022-07-14 11:08:27	2022-07-14 11:08:27	NULL	POST	14	2022-07-14 11:08:27	application/json	0	/api/v1/link/create
14	0	2022-07-14 11:26:22	2022-07-14 11:26:22	NULL	POST	14	2022-07-14 11:26:22	application/json	0	/api/v1/link/create
15	0	2022-07-14 11:27:48	2022-07-14 11:27:48	NULL	POST	14	2022-07-14 11:27:48	application/json	0	/api/v1/link/create
16	0	2022-07-14 15:59:00	2022-07-14 15:59:00	NULL	POST	14	2022-07-14 15:59:00	application/json	0	/api/v1/link/create
17	0	2022-07-14 16:30:37	2022-07-14 16:30:37	NULL	POST	14	2022-07-14 16:30:37	application/json	0	/api/v1/link/create
18	0	2022-07-14 16:34:25	2022-07-14 16:34:25	NULL	POST	14	2022-07-14 16:34:25	application/json	0	/api/v1/link/create
19	0	2022-07-14 17:32:39	2022-07-14 17:32:39	NULL	POST	94163	2022-07-14 17:32:39	multipart/form-data	1	/api/v1/upgrade/offline
20	0	2022-07-14 17:39:33	2022-07-14 17:39:33	NULL	POST	94257	2022-07-14 17:39:33	multipart/form-data	1	/api/v1/upgrade/offline
21	0	2022-07-14 17:42:15	2022-07-14 17:42:15	NULL	POST	94257	2022-07-14 17:42:15	multipart/form-data	0	/api/v1/upgrade/offline
22	0	2022-07-14 17:43:12	2022-07-14 17:43:12	NULL	POST	94257	2022-07-14 17:43:12	multipart/form-data	0	/api/v1/upgrade/offline
23	0	2022-07-14 18:09:03	2022-07-14 18:09:03	NULL	POST	94257	2022-07-14 18:09:03	multipart/form-data	0	/api/v1/upgrade/offline
24	0	2022-07-14 18:10:44	2022-07-14 18:10:44	NULL	POST	94257	2022-07-14 18:10:44	multipart/form-data	0	/api/v1/upgrade/offline
25	0	2022-07-15 10:29:48	2022-07-15 10:29:48	NULL	POST	94257	2022-07-15 10:29:48	multipart/form-data	1	/api/v1/upgrade/offline
26	0	2022-07-15 10:34:58	2022-07-15 10:34:58	NULL	POST	94263	2022-07-15 10:34:58	multipart/form-data	0	/api/v1/upgrade/offline
27	0	2022-07-15 10:43:42	2022-07-15 10:43:42	NULL	POST	878	2022-07-15 10:43:42	multipart/form-data	0	/api/v1/upgrade/offline
28	0	2022-07-15 10:53:46	2022-07-15 10:53:46	NULL	POST	878	2022-07-15 10:53:46	multipart/form-data	0	/api/v1/upgrade/offline
29	0	2022-07-15 11:08:50	2022-07-15 11:08:50	NULL	POST	878	2022-07-15 11:08:50	multipart/form-data	1	/api/v1/upgrade/offline
30	0	2022-07-15 11:10:39	2022-07-15 11:10:39	NULL	POST	878	2022-07-15 11:10:39	multipart/form-data	0	/api/v1/upgrade/offline
31	0	2022-07-15 11:13:18	2022-07-15 11:13:18	NULL	POST	972	2022-07-15 11:13:18	multipart/form-data	1	/api/v1/upgrade/offline
32	0	2022-07-15 11:14:51	2022-07-15 11:14:51	NULL	POST	972	2022-07-15 11:14:51	multipart/form-data	0	/api/v1/upgrade/offline
33	0	2022-07-15 15:06:18	2022-07-15 15:06:18	NULL	GET	0	2022-07-15 15:06:17		0	/composer.html
34	0	2022-07-19 08:50:11	2022-07-19 08:50:11	NULL	GET	0	2022-07-19 08:50:11		13	/api/v1/account/list?pa

图 5-36　API 请求日志记录示例 1



i_path	referer	api_desc	status_code	response_data	account_info	request_params	desc
pi/v1/account/create	http://10.91.8.20:8080/swagge...	添加用户信息	200	{"code":10001,"message":"Err...	null	{"password":"aa","username":"...	
pi/v1/account/create	http://10.91.8.20:8080/swagge...	添加用户信息	200	{"code":0,"message":"OK","dat...	null	{"password":"aa","username":"...	
pi/v1/account/create	http://10.91.8.20:8080/swagge...	添加用户信息	200	{"code":0,"message":"OK","dat...	null	{"password":"aa","username":"...	
pi/v1/account/create	http://10.91.8.20:8080/swagge...	添加用户信息	200	{"code":0,"message":"OK","dat...	null	{"password":"aa","username":"...	
pi/v1/account/create	http://10.91.8.20:8080/swagge...	添加用户信息	200	{"code":0,"message":"OK","dat...	null	{"password":"aa","username":"...	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":10001,"message":"节...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":0,"message":"OK","dat...	null	{"node_ids":"1,2,3"}	
pi/v1/link/create	http://10.91.8.20:8080/swagge...	根据节点信息创建链路	200	{"code":0,"message":"OK","dat...	null	{"node_ids":"1,2,3"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"pp":"dddsa%2622%3D33"}	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa%2622%3D33","a...	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa%2622%3D33","a...	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa\u002622=33","a...	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa\u002622=33","a...	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa\u002622=33","a...	
pi/v1/upgrade/offline	http://10.91.8.20:8080/swagge...	离线升级证书、节点信息、系统	200	{"code":0,"message":"OK","dat...	null	{"a1":"dddsa\u002622=33","a...	
			404	The incorrect API route.	null	{}	
pi/v1/account/list	http://10.91.8.20:8080/swagge...	分页查询用户信息	200	{"code":0,"message":"OK","dat...	null	{"page_no":"1","page_size":"22"}	

图 5-37　API 请求日志记录示例 2

5.9 使用 Go 调用外部命令的多种方式

在 Go 中用于执行命令的库是 os/exec，exec.Command 函数返回一个 Cmd 对象，根据不同的需求，可以将命令的执行分为以下 3 种情况：

（1）只执行命令，不获取结果。

（2）执行命令，并获取结果（不区分 stdout 和 stderr）。

（3）执行命令，并获取结果（区分 stdout 和 stderr）。

第 1 种：只执行命令，不获取结果。

直接调用 Cmd 对象的 Run 函数，返回的只有成功和失败，无法获取任何输出的结果，示例代码如下：

```
//anonymous - link\example\chapter5\cmd\cmd1.go
package main

import (
    "log"
    "os/exec"
)

func main() {
    cmd : = exec.Command("ls", " - l", "/var/log/")
    err : = cmd.Run()
    if err ! = nil {
        log.Fatalf("cmd.Run() failed with % sn", err)
    }
}
```

第 2 种：执行命令，并获取结果，有时执行一个命令就是想要获取输出结果，此时可以调用 Cmd 的 CombinedOutput 函数，示例代码如下：

```
//anonymous - link\example\chapter5\cmd\cmd2.go
package main

import (
"fmt"
"log"
"os/exec"
)

func main() {
    cmd := exec.Command("ls", "-l", "/var/log/")
    out, err := cmd.CombinedOutput()
    if err != nil {
        fmt.Printf("combined out:n%sn", string(out))
        log.Fatalf("cmd.Run() failed with %s", err)
    }
    fmt.Printf("combined out:n%sn", string(out))
}
```

CombinedOutput 函数，只返回 out，并不区分 stdout 和 stderr，结果示例如下：

```
$ go run demo.go

combined out:
total 11540876
-rw-r--r--    2 root   root   4096 Oct 29 2018 yum.log
drwx------    2 root   root   94 Nov 6 05:56 audit
-rw-r--r--    1 root   root   185249234 Nov 28 2019 message
-rw-r--r--    2 root   root   16374 Aug 28 10:13 boot.log
```

需要注意的是 Shell 命令能执行，并不代表 exec 也能执行。

例如想查看 /var/log/ 目录下带有 log 后缀名的文件，有点 Linux 基础的人都会尝试用下面这个命令进行查看：

```
$ ls -l /var/log/*.log
total 11540
-rw-r--r--    2 root   root     4096 Oct 29 2018 /var/log/yum.log
-rw-r--r--    2 root   root     16374 Aug 28 10:13 /var/log/boot.log
```

按照这个写法将它放入 exec.Command，示例命令如下：

```
//anonymous - link\example\chapter5\cmd\cmd3.go
package main

import (
"fmt"
```

```
"log"
"os/exec"
)

func main() {
    cmd := exec.Command("ls", "-l", "/var/log/*.log")
    out, err := cmd.CombinedOutput()
    if err != nil {
        fmt.Printf("combined out:n%sn", string(out))
        log.Fatalf("cmd.Run() failed with %sn", err)
    }
    fmt.Printf("combined out:n%sn", string(out))
}
```

运行时出现报错情况,结果类似如下:

```
$ go run demo.go
combined out:
ls: cannot access /var/log/*.log: No such file or directory

2020/11/11 19:46:00 cmd.Run() failed with exit status 2
exit status 1
```

为什么会报错呢? Shell 明明没有问题。其实很简单,原来 ls-l /var/log/*.log 并不等价于下面这段代码:

```
exec.Command("ls", "-l", "/var/log/*.log")
```

上面这段代码对应的 Shell 命令应该是下面这样,如果这样写,ls 就会把参数里的内容当成具体的文件名,而忽略通配符 *,对应的 Shell 代码如下:

```
$ ls -l "/var/log/*.log"
ls: cannot access /var/log/*.log: No such file or directory
```

第 3 种:执行命令,并区分 stdout 和 stderr,示例代码如下:

```
//anonymous-link\example\chapter5\cmd\cmd4.go
package main

import (
    "Bytes"
    "fmt"
    "log"
    "os/exec"
)

func main() {
    cmd := exec.Command("ls", "-l", "/var/log/*.log")
    var stdout, stderr Bytes.Buffer
    cmd.Stdout = &stdout      //标准输出
```

```
    cmd.Stderr = &stderr        //标准错误
    err := cmd.Run()
    outStr, errStr := string(stdout.Bytes()), string(stderr.Bytes())
    fmt.Printf("out:\n%s\nerr:\n%s\n", outStr, errStr)
    if err != nil {
        log.Fatalf("cmd.Run() failed with %s\n", err)
    }
}
```

运行之后可以看到前面的报错内容被归入标准错误里，结果如下：

```
$ go run demo.go
out:

err:
ls: cannot access /var/log/*.log: No such file or directory

2020/11/11 19:59:31 cmd.Run() failed with exit status 2
exit status 1
```

第 4 种：多条命令组合，使用管道。将上一条命令的执行结果，作为下一条命令的参数。在 Shell 中可以使用管道符实现。

例如统计 message 日志中 ERROR 日志的数量，Shell 代码如下：

```
$ grep ERROR /var/log/messages | wc -l
19
```

类似地，在 Go 中也有类似的实现，代码如下：

```
//anonymous-link\example\chapter5\cmd\cmd5.go
package main
import (
    "os"
    "os/exec"
)
func main() {
    c1 := exec.Command("grep", "ERROR", "/var/log/messages")
    c2 := exec.Command("wc", "-l")
    c2.Stdin, _ = c1.StdoutPipe()
    c2.Stdout = os.Stdout
    _ = c2.Start()
    _ = c1.Run()
    _ = c2.Wait()
}
```

输出如下：

```
$ go run demo.go
19
```

第 5 种：设置命令级别的环境变量。使用 os 库的 Setenv 函数设置的环境变量，其作用于整个进程的生命周期，代码如下：

```
//anonymous - link\example\chapter5\cmd\cmd6.go
package main
import (
    "fmt"
    "log"
    "os"
    "os/exec"
)
func main() {
    os.Setenv("NAME", "wangbm")
    cmd := exec.Command("echo", os.ExpandEnv("$NAME"))
    out, err := cmd.CombinedOutput()
    if err != nil {
        log.Fatalf("cmd.Run() failed with %s\n", err)
    }
    fmt.Printf("%s", out)
}
```

只要在这个进程里，NAME 变量的值都会是 wangbm，无论执行多少次命令，执行的结果都如下：

```
$ go run demo.go
wangbm
```

如果想把环境变量的作用范围再缩小到命令级别，则也是有办法的。

为了方便验证，新建个 Shell 脚本，内容如下：

```
$ cat /home/wangbm/demo.sh
echo $NAME

$ bash /home/wangbm/demo.sh # 由于全局环境变量中没有 NAME，所以无输出
```

另外，demo.go 文件里的代码如下：

```
//anonymous - link\example\chapter5\cmd\cmd7.go
package main
import (
    "fmt"
    "os"
    "os/exec"
)


func ChangeYourCmdEnvironment(cmd *exec.Cmd) error {
    env := os.Environ()
    cmdEnv := []string{}
```

```
    for _, e := range env {
        cmdEnv = append(cmdEnv, e)
    }
    cmdEnv = append(cmdEnv, "NAME=wangbm")
    cmd.Env = cmdEnv

    return nil
}

func main() {
    cmd1 := exec.Command("bash", "/home/wangbm/demo.sh")
    ChangeYourCmdEnvironment(cmd1) //将环境变量添加到 cmd1 命令：NAME=wangbm
    out1, _ := cmd1.CombinedOutput()
    fmt.Printf("output: %s", out1)

    cmd2 := exec.Command("bash", "/home/wangbm/demo.sh")
    out2, _ := cmd2.CombinedOutput()
    fmt.Printf("output: %s", out2)
}
```

执行后，可以看到第 2 次执行的命令却没有输出 NAME 的变量值，运行结果如下：

```
$ go run demo.go
output: wangbm
output:
```

5.10　打造高级路由器改写 DHCP 服务

DHCP 服务器是为客户端设备自动提供和分配 IP 地址、默认网关等网络参数的网络服务器，会自动发送客户端所需的网络参数，使客户端能够在网络中正常通信。

在路由硬件设备上需要安装 DHCP 服务，让其他的应用设备可以连接到路由设备进行流量数据转发，路由硬件上的 DHCP 服务需要实现以下功能：

（1）支持网线盲插，入网、出网支持随意插入网口。

（2）自动化网桥配置、网卡配置。

（3）自动 IP 分配，以及状态监测。

1. DHCP 服务安装

安装过程比较简单，以 CentOS 7 系统为例，安装命令如下：

```
yum install -y dhcp*
```

查看主机是否已安装 DHCP 包，命令如下：

```
rpm -qa | grep dhcpd
```

2. DHCP 服务配置

DHCP 服务配置的文件路径如下：

```
vim /etc/dhcp/dhcpd.conf
```

配置文件的内容如下：

```
#dhcpd.conf
option domain-name "example.com";
option domain-name-servers 114.114.114.114;
default-lease-time 600;   #默认租约时间,默认单位为秒
max-lease-time 7200;      #最大租约时间,客户端超过租约但尚未更新 IP 时,最长可以使用该
                          #IP 的时间
log-facility local7;      #本地日志设施
subnet 192.168.1.0 netmask 255.255.255.0 {   #IP 地址范围
    range 192.168.1.10 192.168.1.100;        #分配的 IP 地址范围
    option routers 192.168.1.2;              #配置网关
}
```

DHCP 在配置文件中的参数表明如何执行任务，以及是否执行任务，或将哪些网络配置选项发送给客户。主要参数及含义如下：

```
ddns-update-style          #配置 DHCP-DNS 互动更新模式

default-lease-time         #指定缺省租赁时间的长度,单位为秒

max-lease-time             #指定最大租赁时间长度,单位为秒

hardware                   #指定网卡接口类型和 MAC 地址

server-name                #通知 DHCP 客户服务器名称

get-lease-hostnames flag   #检查客户端使用的 IP 地址

fixed-address ip           #分配给客户端一个固定的地址

authritative               #拒绝不正确的 IP 地址的要求
```

DHCP 在配置文件中的声明用来描述网络布局、提供客户的 IP 地址等。主要声明及含义如下：

```
shared-network                  #用来告知是否一些子网络分享相同网络

subnet                          #描述一个 IP 地址是否属于该子网

range                           #为起始 IP 和终止 IP 提供动态分配 IP 的范围

host 主机名称                   #参考特别的主机
```

```
group                                          #为一组参数提供声明
allow unknown-clients ;  deny unknown-client     #是否将 IP 动态地分配给未知的使用者
allow bootp;deny bootp                         #是否响应激活查询
allow booting ;  deny booting                  #是否响应使用者查询
filename                                       #开始启动文件的名称,应用于无盘工作站
next-server                                   #设置服务器从引导文件中装主机名,应用于无盘工作站
```

DHCP 在配置文件中的选项用来配置 DHCP 可选参数,全部用 option 关键字作为开始。主要选项及含义如下:

```
subnet-mask                                    #为客户端设定子网掩码
domain-name                                    #为客户端指明 DNS 名字
domain-name-servers                            #为客户端指明 DNS 服务器 IP 地址
host-name                                      #为客户端指定主机名称
routers                                        #为客户端设定默认网关
broadcast-address                              #为客户端设定广播地址
ntp-server                                     #为客户端设定网络时间服务器 IP 地址
time-offset                                    #为客户端设定和格林尼治时间的偏移时间,单位为秒
# ---  default gateway 关于网关的配置
option routers 192.168.23.1;                   #设置客户端默认网关
option subnet-mask 255.255.255.0;              #设置客户端子网掩码
option domain-name "domain.org";               #设置域名
option domain-name-servers 192.168.23.128;     #设置网络内部 DNS 服务器的 IP 地址
option time-offset  -18000;                    #Eastern Standard Time
range dynamic-bootp 192.168.23.129 192.168.23.254; #定义 DHCP 地址池的服务范围,需排除静态地址
default-lease-time 21600;                      #设置默认租约时间
max-lease-time 43200;                          #设置最大租约时间
```

```
host ns { #设置静态 IP 地址,用于网络内固定服务器 IP,不要置于定义好的 DHCP 地址池范围内,否
          #则会引起 IP 冲突

hardware ethernet 00:0C:29:00:5B:78;          #设置静态主机的 MAC 地址,与 IP 进行绑定

fixed-address 192.168.23.128;                 #固定的地址
}
```

在配置文件规范中“#”号为注释,除括号一行外,每行都应以“;”结尾。DHCP 的 IP 分为静态 IP 和动态 IP,如果要设置静态 IP,则需要知道要设置主机的 MAC 地址。

3. DHCP 服务启动

配置文件修改完成后,启动 DHCPD 服务,命令如下:

```
systemctl start dhcpd              #启动 DHCPD 服务
systemctl enable dhcpd             #设置 DHCPD 服务开机自启动
```

如有错误,则会将错误信息显示在屏幕上。可以通过 netstat -unlt|grep 67 查看 DHCP 的信息,也可以通过/var/log.messages 查看 DHCP 的日志信息。

4. DHCP 客户端

配置网卡,设置为以 DHCP 方式获取 IP 地址,然后重启网卡并获取 IP 地址,在服务器端可以查看/var/log/messages 日志信息,以便确认客户端是否在向 DHCP 客户端申请 IP 地址,可以通过/var/db/dhcp.leases 查看租约申请记录。

/var/log/messages 服务器端日志查看 DHCP 客户端申请 IP 地址的过程。

DHCP 服务器和客户端租约建立的启动和到期时间的记录文件,路径如下:

```
/var/lib/dhcpd/dhcpd.leases
```

查看记录文件,命令如下:

```
cat /var/db/dhcpd.leases
```

DHCP 服务器和客户端租约建立的启动和到期时间的记录文件,仅在客户端申请 IP 地址之后才会有。lease 开始租约时间和 lease 结束租约时间是格林尼治标准时间(GMT),不是本地时间。

DHCP 客户端重新获取 IP 地址,命令如下:

```
dhclient -r            #终止旧客户端进程

dhclient eth0          #重新获取某块网卡的 IP

dhclient               #重新获取 IP
```

5. 网桥及网口配置

为了方便路由硬件设备上的其他网口可以连接到客户端,需要把除了出网的网口的其

他网口都聚合到一起，配置网关。这需要借助网桥实现，把所有物理网口挂载到虚拟网桥上。

创建网桥，命令如下：

```
brctl addbr $brName
```

将物理网口挂载到网桥，命令如下：

```
brctl addif $brName $devName
```

为网桥设置 IP，命令如下：

```
ifconfig $brName 192.168.10.1/24
```

6. 流量转发配置

配置流量转发，让网桥进来的流量都可以通过出网的网口出去，命令如下：

```
echo 1 > /proc/sys/net/ipv4/ip_forward
echo net.ipv4.ip_forward = 1 >> /etc/sysctl.conf
sysctl -p

iptables -A FORWARD -i $brName -o $outName -j ACCEPT
iptables -t nat -A POSTROUTING -o $outName -j MASQUERADE
```

7. 自动化配置脚本

完整的自动化配置脚本如下：

```
#anonymous-link\code\ansible\script\start\init_dhcp.sh

#! /bin/bash

destIP="8.8.8.8"
dhcpPath="/etc/dhcp/dhcpd.conf"

function init_dhcp() {
  receive=$(ping -n -c 5 $destIP | grep received | awk -F"," '{print $2}' | awk '{print
$1}')
  if [ $receive -gt 2 ]; then
#outName=$(ip route get $destIP | awk -F 'dev' '{print $NF}' | awk '{print $1}' | awk 'NR==
=1{print}')
#outIP=$(ip route get $destIP | awk -F 'via' '{print $NF}' | awk '{print $1}' | awk 'NR==
1{print}')
#ip route add $destIP dev $outName via $outIP
#echo $outName > /etc/dhcp/default.dev
#echo $outIP > /etc/dhcp/default.ip
    outName=$(cat /etc/dhcp/default.dev)
    outIP=$(cat /etc/dhcp/default.ip)
#ip route del default dev $outName
  else
```

```
    echo "There are currently no routes detected that are connected to the Internet."
    exit 2
  fi

  netFilePath="/etc/sysconfig/network-scripts/"
  number=0
  brName="br10"
  nicNames=($(brctl show $brName | awk 'NR>1 {print $NF}'))
  for nic in ${nicNames[@]}; do
    if [ -z $nic ]; then
      continue
    fi
    brctl delif $brName $nic
  done
  brctl delbr $brName
  brctl addbr $brName
  for devEnum in $(ip link show | grep ^[0-9]\\+: | awk -F '' {'print $2'}); do
    devName=${devEnum%:}
    devName=${devName%@*}
    nicPath=${netFilePath}ifcfg-$devName
    if [ $devName != "$outName" ] && [[ $devName == e* ]] && [ -f $nicPath ]; then
      ifconfig $devName 0
      brctl addif $brName $devName
      let number++
    fi

  done

  if [ $number -lt 1 ]; then
     echo "The network configuration is wrong, please check the system and network card
condition."
    exit 2
  fi

  ifconfig $brName 192.168.10.1/24

  cat >$dhcpPath <<EOF
#
# DHCP Server Configuration file
# see /usr/share/doc/dhcp*/dhcpd.conf.example
# see dhcpd.conf(5) man page
#
subnet 192.168.10.0 netmask 255.255.255.0 {
        range 192.168.10.10 192.168.10.200;
        option domain-name-servers 8.8.8.8;
# option domain-name "dns.mitu.cn";
        option routers 192.168.10.1;
        option broadcast-address 192.168.10.255;
        default-lease-time 6000;
        max-lease-time 72000;
}
```

```
EOF

  systemctl restart dhcpd

  rule_ids=($(iptables -L INPUT -nv --line-number | grep icmp-host-prohibited | awk
'{print $1}'))
  for((i = ${#rule_ids[@]} - 1; i >= 0; i--)); do
    iptables -D INPUT ${rule_ids[i]}
  done
  rule_ids=($(iptables -L FORWARD -nv --line-number | grep icmp-host-prohibited |
awk '{print $1}'))
  for((i = ${#rule_ids[@]} - 1; i >= 0; i--)); do
    iptables -D FORWARD ${rule_ids[i]}
  done

  count=$(iptables -L FORWARD -nv | awk '{print $6$7}' | grep "$brName$outName" | wc -l)
  if [ $count -lt 1 ]; then
    iptables -A FORWARD -i $brName -o $outName -j ACCEPT
  fi
  count=$(iptables -t nat -L POSTROUTING -nv | awk '{print $6$7}' | grep "*$outName" |
wc -l)
  if [ $count -lt 1 ]; then
    iptables -t nat -A POSTROUTING -o $outName -j MASQUERADE
  fi

  sysPath="/etc/sysctl.conf"
  count=$(cat $sysPath | grep -i "net.ipv4.ip_forward" | wc -l)
  if [ $count -gt 0 ]; then
    sed -i 's/net.ipv4.ip_forward.*/net.ipv4.ip_forward=1/g' $sysPath
  else
    echo "net.ipv4.ip_forward=1" >>$sysPath
  fi
  sysctl -p
  echo "dhcp success"
}

init_dhcp
```

思考：上面的配置可以为客户端分配 IP，并把客户端的流量通过路由设备的出网转发到公网，但是如果想访问路由设备本身的服务应该怎么做？

5.11　节点自动化部署

5.11.1　节点部署流程

为了实现节点自动化部署，主要包含以下步骤：

(1) 配置要出网的路由，包含节点的连接及配置。

(2) 获取目标服务器 root 账号及密码后,通过 ansible 把自动化部署脚本上传至目标服务器。

(3) 通过 ansible 远程传入参数并执行目标服务器上的脚本,完成节点服务器的部署,生成相关的连接证书。

(4) 通过 ansible 远程将目标服务器上的连接证书复制到本地保存。

(5) 通过 ansible 远程清理目标服务器上的使用痕迹和相关文件。

(6) 释放相关的节点连接,删除相关的出网路由。

5.11.2 实例:节点部署

以部署 openconnect 为例,openconnect 自动部署脚本的内容如下:

```
#anonymous - link\code\ansible\script\openconnect\install.sh

#! /bin/sh

destIP = "8.8.8.8"

check_os() {
  os_type = CentOS
  rh_file = "/etc/redhat - release"
  if grep - qs "Red Hat" " $ rh_file"; then
    os_type = rhel
  fi
  if grep - qs "release 7" " $ rh_file"; then
    os_ver = 7
  elif grep - qs "release 8" " $ rh_file"; then
    os_ver = 8
    grep - qi stream " $ rh_file" && os_ver = 8s
    grep - qi rocky " $ rh_file" && os_type = rocky
    grep - qi alma " $ rh_file" && os_type = alma
  elif grep - qs "Amazon Linux release 2" /etc/system - release; then
    os_type = amzn
    os_ver = 2
  else
    os_type = $(lsb_release - si 2 >/dev/null)
    [ - z " $ os_type" ] && [ - f /etc/os - release ] && os_type = $(. /etc/os - release &&
printf '% s' " $ ID")
    case $ os_type in
    [Uu]buntu
      os_type = Ubuntu
      ;;
    [Dd]ebian
      os_type = debian
      ;;
    [Rr]aspbian
```

```
        os_type=raspbian
        ;;
      *)
        exiterr "This script only supports Ubuntu, Debian, CentOS/RHEL 7/8 and Amazon Linux 2."
        ;;
      esac
      os_ver=$(sed 's/\..*//' /etc/debian_version | tr -dc 'A-Za-z0-9')
      if [ "$os_ver" = "8" ] || [ "$os_ver" = "jessiesid" ]; then
        exiterr "Debian 8 or Ubuntu < 16.04 is not supported."
      fi
      if { [ "$os_ver" = "10" ] || [ "$os_ver" = "11" ]; } && [ ! -e /dev/ppp ]; then
        exiterr "/dev/ppp is missing. Debian 11 or 10 users, see: https://git.io/vpndebian10"
      fi
    fi
  }
  install_vpn() {
    systemctl stop openvpn-server
    systemctl disable openvpn-server
    systemctl stop openvpn@server
    systemctl disable openvpn@server
    systemctl stop strongswan
    systemctl disable strongswan
    systemctl stop wg-quick@wg0
    systemctl disable wg-quick@wg0
    systemctl stop ocserv
    systemctl disable ocserv

    if [ "$os_type" = "debian" ] || [ "$os_type" = "raspbian" ]; then
      install_vpn_debian $1 $2
    elif [ "$os_type" = "Ubuntu" ]; then
      install_vpn_Ubuntu $1 $2
    else
      install_vpn_CentOS $1 $2
    fi
  }
  install_vpn_CentOS(){
    yum install -y epel-release
    yum install -y ocserv rsync
    yum install -y curl iptables-services
    local_ip=$2
    cd /etc/pki/ocserv
    rm -rf ca-key.pem
    rm -rf ca-cert.pem
    rm -rf server-key.pem
    rm -rf server-cert.pem
    rm -rf client-key.pem
    rm -rf client-cert.pem
    rm -rf cacerts/*
    rm -rf private/*
    rm -rf public/*
```

```
  systemctl enable iptables.service
  systemctl start iptables.service

  cat > ca.tmpl << EOF
  cn = "localhost CA"
  expiration_days = 9999
  serial = 1
  ca
  cert_signing_key
EOF


  cat > server.tmpl << EOF
  cn = " $ local_ip"
  serial = 2
  expiration_days = 9999
  signing_key
  encryption_key
EOF

  cat > client.tmpl << EOF
  dn = "cn = com,O = myvpn,UID = client"
  expiration_days = 3650
  signing_key
  tls_www_client
EOF

  certtool -- generate - privkey -- outfile ca - key.pem
  certtool -- generate - self - signed -- load - privkey ca - key.pem -- template ca.tmpl --
outfile ca - cert.pem
  certtool -- generate - privkey -- outfile server - key.pem
  certtool - - generate - certificate - - load - privkey server - key. pem - - load - ca -
certificate ca - cert. pem - - load - ca - privkey ca - key. pem - - template server. tmpl - -
outfile server - cert.pem
  certtool -- generate - privkey -- outfile client - key.pem
  certtool - - generate - certificate - - load - privkey client - key. pem - - load - ca -
certificate ca - cert. pem - - load - ca - privkey ca - key. pem - - template client. tmpl - -
outfile client - cert.pem

  cat >/etc/ocserv/ocserv.conf << EOF
auth = "certificate"
listen - host = 0.0.0.0
listen - host - is - dyndns = true
tcp - port = 1194
udp - port = 1194
run - as - user = ocserv
run - as - group = ocserv
socket - file = ocserv.sock
chroot - dir = /var/lib/ocserv
isolate - workers = false
max - clients = 16
```

```
max-same-clients = 2
rate-limit-ms = 100
keepalive = 32400
dpd = 90
mobile-dpd = 1800
switch-to-tcp-timeout = 25
try-mtu-discovery = false
server-cert = /etc/pki/ocserv/public/server-cert.pem
server-key = /etc/pki/ocserv/private/server-key.pem
ca-cert = /etc/pki/ocserv/cacerts/ca-cert.pem
cert-user-oid = 0.9.2342.19200300.100.1.1
tls-priorities = "NORMAL:%SERVER_PRECEDENCE:%COMPAT:-VERS-SSL3.0"
auth-timeout = 240
min-reauth-time = 300
max-ban-score = 50
ban-reset-time = 300
Cookie-timeout = 300
deny-roaming = false
rekey-time = 172800
rekey-method = ssl
use-occtl = true
pid-file = /var/run/ocserv.pid
device = vpns
predictable-ips = true
default-domain = example.com
ipv4-network = $1
ipv4-netmask = 255.255.255.0
ping-leases = false
route = $1/255.255.255.0
cisco-client-compat = true
dtls-legacy = true
user-profile = profile.xml
EOF


  iptables -F
  iptables -t nat -F
  count=$(cat /etc/sysctl.conf | grep "^net.ipv4.ip_forward" | wc -l)
  if [ $count -gt 0 ]; then
   sed -i 's/net.ipv4.ip_forward.*/net.ipv4.ip_forward=1/g' /etc/sysctl.conf
   sysctl -p
  else
   echo "net.ipv4.ip_forward=1">> /etc/sysctl.conf
   sysctl -p
  fi

  systemctl stop firewalld
  systemctl disable firewalld

  nicName=$(ip route get $destIP | awk -F 'dev' '{print $NF}' | awk '{print $1}' | awk 'NR==
1{print}')
```

```
    iptables -t nat -A POSTROUTING -s $1/24 -o $nicName -j MASQUERADE
    iptables -I INPUT -p tcp --dport 1194 -j ACCEPT
    iptables -I INPUT -p udp --dport 1194 -j ACCEPT
    iptables -I FORWARD -s $1/24 -j ACCEPT
    iptables -A OUTPUT -p all -j ACCEPT
    iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT
    current_path="/etc/pki/ocserv"
    client_info=${current_path}/client_info
    cd $current_path
    rm -rf client_info*
    cp ca-cert.pem cacerts
    cp ca-key.pem private
    cp server-key.pem private
    cp server-cert.pem public
    chmod -R 777 private
    chmod -R 777 public
    mkdir -p client_info
    cp client-* client_info
    cp ca-cert.pem client_info
    chmod -R 777 client_info/
    tar -zcvf $client_info.tar.gz -C $current_path client_info/
    mkdir -p /home/openconnect
    cp -rf client_info /home/openconnect
    chmod 777 /home/openconnect/*
    iptables-save
    service iptables save
    systemctl restart ocserv
    systemctl enable ocserv

    echo
"============================================================================
============================================================"
    echo 'The vpn server has been installed, you can view the vpn client connection information in
the "client_info" directory in the current directory.'
    echo
"============================================================================
============================================================"
}
install_vpn_Ubuntu(){
    apt-get install -y ocserv
    apt-get install -y gnutls-bin
    apt-get install -y curl rsync iptables

    local_ip=$2
    mkdir -p /etc/pki/ocserv
    cd /etc/pki/ocserv
    cat > ca.tmpl << EOF
    cn="localhost CA"
    expiration_days=9999
    serial=1
    ca
    cert_signing_key
EOF
```

```
    cat > server.tmpl << EOF
  cn = " $ local_ip"
  serial = 2
  expiration_days = 9999
  signing_key
  encryption_key
EOF
    cat > client.tmpl << EOF
  dn = "cn = com,O = myvpn,UID = client"
  expiration_days = 3650
  signing_key
  tls_www_client
EOF
  rm -rf ca-key.pem
  rm -rf ca-cert.pem
  rm -rf server-key.pem
  rm -rf server-cert.pem
  rm -rf client-key.pem
  rm -rf client-cert.pem
  rm -rf client_*
  rm -rf cacerts/*
  rm -rf private/*
  rm -rf public/*
  certtool --generate-privkey --outfile ca-key.pem
  certtool --generate-self-signed --load-privkey ca-key.pem --template ca.tmpl --
outfile ca-cert.pem
  certtool --generate-privkey --outfile server-key.pem
  certtool --generate-certificate --load-privkey server-key.pem --load-ca-
certificate ca-cert.pem --load-ca-privkey ca-key.pem --template server.tmpl --
outfile server-cert.pem
  certtool --generate-privkey --outfile client-key.pem
  certtool --generate-certificate --load-privkey client-key.pem --load-ca-
certificate ca-cert.pem --load-ca-privkey ca-key.pem --template client.tmpl --
outfile client-cert.pem
  cat >/etc/ocserv/ocserv.conf << EOF
auth = "certificate"
listen-host = 0.0.0.0
listen-host-is-dyndns = true
tcp-port = 1194
udp-port = 1194
socket-file = ocserv.sock
isolate-workers = false
max-clients = 16
max-same-clients = 2
rate-limit-ms = 100
keepalive = 32400
dpd = 90
mobile-dpd = 1800
switch-to-tcp-timeout = 25
try-mtu-discovery = false
server-cert = /etc/pki/ocserv/public/server-cert.pem
```

```
server-key = /etc/pki/ocserv/private/server-key.pem
ca-cert = /etc/pki/ocserv/cacerts/ca-cert.pem
cert-user-oid = 0.9.2342.19200300.100.1.1
tls-priorities = "NORMAL:%SERVER_PRECEDENCE:%COMPAT:-VERS-SSL3.0"
auth-timeout = 240
min-reauth-time = 300
max-ban-score = 50
ban-reset-time = 300
Cookie-timeout = 300
deny-roaming = false
rekey-time = 172800
rekey-method = ssl
use-occtl = true
pid-file = /var/run/ocserv.pid
device = vpns
predictable-ips = true
default-domain = example.com
ipv4-network = $1
ipv4-netmask = 255.255.255.0
ping-leases = false
route = $1/255.255.255.0
cisco-client-compat = true
dtls-legacy = true
EOF

  systemctl enable iptables.service
  systemctl start iptables.service
  systemctl stop firewalld
  systemctl disable firewalld
  iptables -F
  iptables -t nat -F
  count=$(cat /etc/sysctl.conf | grep "^net.ipv4.ip_forward" | wc -l)
  if [ $count -gt 0 ]; then
   sed -i 's/net.ipv4.ip_forward.*/net.ipv4.ip_forward=1/g' /etc/sysctl.conf
   sysctl -p
  else
   echo "net.ipv4.ip_forward=1">> /etc/sysctl.conf
   sysctl -p
  fi

  nicName=$(ip route get $destIP | awk -F 'dev' '{print $NF}' | awk '{print $1}' | awk 'NR==
1{print}')
  iptables -t nat -A POSTROUTING -s $1/24 -o $nicName -j MASQUERADE
  iptables -I INPUT -p tcp --dport 1194 -j ACCEPT
  iptables -I INPUT -p udp --dport 1194 -j ACCEPT
  iptables -I FORWARD -s $1/24 -j ACCEPT
  iptables -A OUTPUT -p all -j ACCEPT
  iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT


  mkdir /etc/iptables
```

```
  chmod 777 /etc/iptables/ *
  iptables - save > /etc/iptables/iptables.rules
  cat > /etc/iptables/startup.sh << EOF
#! /bin/sh
iptables - restore < /etc/iptables/iptables.rules
EOF
  cat > /etc/iptables/reload.sh << EOF
#! /bin/sh
iptables  - F
iptables - restore < /etc/iptables/iptables.rules
EOF
  cat > /etc/systemd/system/myiptables.service << EOF
[Unit]
Description = Reload iptables rules
[Service]
Type = oneshot
ExecStart = /bin/sh  - c /etc/iptables/startup.sh
ExecReload = /bin/sh  - c /etc/iptables/reload.sh
ExecStop = /bin/true
[Install]
WantedBy = multi - user.target
EOF

  chmod  - R 777 /etc/iptables
  systemctl daemon - reload
  systemctl enable myiptables.service


  current_path = "/etc/pki/ocserv"
  client_info =  $ {current_path}/client_info
  cd  $ current_path
  rm  - rf client_info *
  mkdir  - p client_info
  mkdir  - p cacerts
  mkdir  - p private
  mkdir  - p public
  chmod  - R 777 private
  chmod  - R 777 public
  chmod  - R 777 cacerts
  cp ca - cert.pem cacerts
  cp ca - key.pem private
  cp server - key.pem private
  cp server - cert.pem public
  cp client -  *  client_info
  cp ca - cert.pem client_info
  chmod  - R 777 client_info/
  tar  - zcvf  $ client_info.tar.gz  - C  $ current_path client_info/
  mkdir  - p /home/openconnect
  cp  - rf client_info /home/openconnect
  chmod 777 /home/openconnect/ *
```

```
    systemctl restart ocserv
    systemctl enable ocserv

    echo
  " ================================================================================
  ============================================================ "
    echo 'The vpn server has been installed, you can view the vpn client connection information in
  the "client_info" directory in the current directory. '
    echo
  " ================================================================================
  ============================================================ "
  }
  install_vpn_debian(){
    apt - get install  - y ocserv
    apt - get install  - y gnutls - bin
    apt - get install  - y curl rsync iptables
    local_ip = $ 2
    mkdir  - p /etc/pki/ocserv
    cd /etc/pki/ocserv
    cat > ca.tmpl << EOF
    cn = "localhost CA"
    expiration_days = 9999
    serial = 1
    ca
    cert_signing_key
  EOF
      cat > server.tmpl << EOF
    cn = " $ local_ip"
    serial = 2
    expiration_days = 9999
    signing_key
    encryption_key
  EOF
      cat > client.tmpl << EOF
    dn = "cn = com,O = myvpn,UID = client"
    expiration_days  =  3650
    signing_key
    tls_www_client
  EOF
    rm  - rf ca - key.pem
    rm  - rf ca - cert.pem
    rm  - rf server - key.pem
    rm  - rf server - cert.pem
    rm  - rf client - key.pem
    rm  - rf client - cert.pem
    rm  - rf client_ *
    rm  - rf cacerts/ *
    rm  - rf private/ *
    rm  - rf public/ *
    certtool  -- generate - privkey  -- outfile ca - key.pem
    certtool  -- generate - self - signed  -- load - privkey ca - key.pem  -- template ca.tmpl  --
  outfile ca - cert.pem
    certtool  -- generate - privkey  -- outfile server - key.pem
```

```
  certtool --generate-certificate --load-privkey server-key.pem --load-ca-
certificate ca-cert.pem --load-ca-privkey ca-key.pem --template server.tmpl --
outfile server-cert.pem
  certtool --generate-privkey --outfile client-key.pem
  certtool --generate-certificate --load-privkey client-key.pem --load-ca-
certificate ca-cert.pem --load-ca-privkey ca-key.pem --template client.tmpl --
outfile client-cert.pem
  cat >/etc/ocserv/ocserv.conf << EOF
auth = "certificate"
listen-host = 0.0.0.0
listen-host-is-dyndns = true
tcp-port = 1194
udp-port = 1194
socket-file = ocserv.sock
isolate-workers = false
max-clients = 16
max-same-clients = 2
rate-limit-ms = 100
keepalive = 32400
dpd = 90
mobile-dpd = 1800
switch-to-tcp-timeout = 25
try-mtu-discovery = false
server-cert = /etc/pki/ocserv/public/server-cert.pem
server-key = /etc/pki/ocserv/private/server-key.pem
ca-cert = /etc/pki/ocserv/cacerts/ca-cert.pem
cert-user-oid = 0.9.2342.19200300.100.1.1
tls-priorities = "NORMAL:%SERVER_PRECEDENCE:%COMPAT:-VERS-SSL3.0"
auth-timeout = 240
min-reauth-time = 300
max-ban-score = 50
ban-reset-time = 300
Cookie-timeout = 300
deny-roaming = false
rekey-time = 172800
rekey-method = ssl
use-occtl = true
pid-file = /var/run/ocserv.pid
device = vpns
predictable-ips = true
default-domain = example.com
ipv4-network = $1
ipv4-netmask = 255.255.255.0
ping-leases = false
route = $1/255.255.255.0
cisco-client-compat = true
dtls-legacy = true
EOF

  systemctl enable iptables.service
  systemctl start iptables.service
```

```
    systemctl stop firewalld
    systemctl disable firewalld
    iptables -F
    iptables -t nat -F
    count=$(cat /etc/sysctl.conf | grep "^net.ipv4.ip_forward" | wc -l)
    if [ $count -gt 0 ]; then
     sed -i 's/net.ipv4.ip_forward.*/net.ipv4.ip_forward=1/g' /etc/sysctl.conf
     sudo sysctl -p
    else
     echo "net.ipv4.ip_forward=1">> /etc/sysctl.conf
     sudo sysctl -p
    fi

    nicName=$(ip route get $destIP | awk -F 'dev' '{print $NF}' | awk '{print $1}' | awk 'NR==
1{print}')
    sudo iptables -t nat -A POSTROUTING -s $1/24 -o $nicName -j MASQUERADE
    sudo iptables -I INPUT -p tcp --dport 1194 -j ACCEPT
    sudo iptables -I INPUT -p udp --dport 1194 -j ACCEPT
    sudo iptables -I FORWARD -s $1/24 -j ACCEPT
    sudo iptables -A OUTPUT -p all -j ACCEPT
    sudo iptables -A INPUT -m state --state RELATED,ESTABLISHED -j ACCEPT

    mkdir /etc/iptables/
    iptables-save > /etc/iptables/iptables.rules
    cat > /etc/iptables/startup.sh << EOF
#! /bin/sh
iptables-restore < /etc/iptables/iptables.rules
EOF
    cat > /etc/iptables/reload.sh << EOF
#! /bin/sh
iptables -F
iptables-restore < /etc/iptables/iptables.rules
EOF
    cat > /etc/systemd/system/myiptables.service << EOF
[Unit]
Description=Reload iptables rules
[Service]
Type=oneshot
ExecStart=/bin/sh -c /etc/iptables/startup.sh
ExecReload=/bin/sh -c /etc/iptables/reload.sh
ExecStop=/bin/true
[Install]
WantedBy=multi-user.target
EOF

    chmod -R 777 /etc/iptables
    systemctl daemon-reload
    systemctl enable myiptables.service


    current_path="/etc/pki/ocserv"
```

```
  client_info = ${current_path}/client_info
  cd $current_path
  rm -rf client_info*
  mkdir -p client_info
  mkdir -p cacerts
  mkdir -p private
  mkdir -p public
  chmod -R 777 private
  chmod -R 777 public
  chmod -R 777 cacerts
  cp ca-cert.pem cacerts
  cp ca-key.pem private
  cp server-key.pem private
  cp server-cert.pem public
  cp client-* client_info
  cp ca-cert.pem client_info
  chmod -R 777 client_info/
  tar -zcvf $client_info.tar.gz -C $current_path client_info/
  mkdir -p /home/openconnect
  cp -rf client_info /home/openconnect
  chmod 777 /home/openconnect/*
  systemctl restart ocserv
  systemctl enable ocserv

  echo
"============================================================================
============================================================"
  echo 'The vpn server has been installed, you can view the vpn client connection information in
the "client_info" directory in the current directory.'
  echo
"============================================================================
============================================================"
}
check_os
install_vpn $1 $2
```

编写调用函数，集成 4 种 VPN 节点部署方案，Go 使用 ansible 自动部署节点通用函数的代码如下：

```
//anonymous-link\code\service\node_manage.go
func CreateFirstNode(username string, password string, ip string, port uint, netmask string,
vpnType string) (error, string) {
    cmd := exec.Command("ssh", username+"@"+ip, "-p", strconv.Itoa(int(port)))
    groupName := uuid.New().String()
    groupName = strings.ReplaceAll(groupName, "-", "")
    md5Value := md5.New()
    md5Value.Write([]Byte(ip))
    hostName := hex.EncodeToString(md5Value.Sum(nil))
    hostName = groupName
    hostPath := "/root/ansible/" + hostName

    conInfo := "echo \"[" + groupName + "]\n" + ip + " ansible_ssh_user=" + username +
" ansible_ssh_pass=" +
```

```
        password + " ansible_sudo_pass = " + password + " ansible_ssh_port = " + strconv.
Itoa(int(port)) + "\" >> " + hostPath
    cmd = exec.Command("sh", "-c", conInfo)
    cmd.Stdout = os.Stdout
    err := cmd.Run()
    if err != nil {
        return err, ""
    }

    parentPath := ""
    for j := 0; j < 5; j++{
        netmask = strings.Split(netmask, "/")[0]
        parentPath = "/home/"
        if vpnType == model.VpnType().OpenVpn {
            dirPath := "/root/ansible/script/openvpn"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m copy -a \"src = " + dirPath +
                " dest = " + parentPath + " force = yes backup = yes\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                return err, ""
            }
            parentPath += "openvpn/"
            shellPath := "install.sh"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m shell -a \"chdir = " +
                parentPath + " sh " + shellPath + " " + netmask + " > result.log\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                return err, ""
            }

        } else if vpnType == model.VpnType().Wireguard {
            dirPath := "/root/ansible/script/wireguard"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m copy -a \"src = " + dirPath +
                " dest = " + parentPath + " force = yes backup = yes\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                return err, ""
            }
            parentPath += "wireguard/"
            shellPath := "install.sh"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m shell -a \"chdir = " +
                parentPath + " sh " + shellPath + " " + netmask + " " + ip + " > result.log\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
```

```
            if err != nil {
                return err, ""
            }
        } else if vpnType == model.VpnType().StrongSwan {
            dirPath := "/root/ansible/script/strongswan"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m copy -a \"src=" + dirPath +
                " dest=" + parentPath + " force=yes backup=yes\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                continue
            }
            parentPath += "strongswan/"
            shellPath := "install.sh"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m shell -a \"chdir=" +
                parentPath + " sh " + shellPath + " " + netmask + " " + ip + " > result.log\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                continue
            }
        } else if vpnType == model.VpnType().OpenConnect {
            dirPath := "/root/ansible/script/openconnect"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m copy -a \"src=" + dirPath +
                " dest=" + parentPath + " force=yes backup=yes\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                continue
            }
            parentPath += "openconnect/"
            shellPath := "install.sh"
            cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + "
-m shell -a \"chdir=" +
                parentPath + " sh " + shellPath + " " + netmask + " " + ip + " > result.log\"")
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                continue
            }
        } else {
            err = fmt.Errorf("the vpn type '%s' is error", vpnType)
        }
        if err == nil {
            break
        }
    }
```

```
        if err != nil {
            return err, ""
        }

        key := uuid.New().String()
        key = strings.ReplaceAll(key, "-", "")
        certPath := "/root/cert/" + key + "/"
        err = os.MkdirAll(certPath, os.ModePerm)
        if err != nil {
            return err, ""
        }
        cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " +
            groupName + " -m synchronize -a \"mode=pull dest=" + certPath + " src=
" + parentPath + "client_info/\"")
        cmd.Stdout = os.Stdout
        err = cmd.Run()
        if err != nil {
            return err, ""
        }

        cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + " -m shell
-a \"rm -rf " + parentPath + "\"")
        cmd.Stdout = os.Stdout
        err = cmd.Run()
        if err != nil {
            return err, ""
        }

        cmd = exec.Command("sh", "-c", "ansible -i " + hostPath + " " + groupName + " -m shell
-a \"reboot\"")
        cmd.Stdout = os.Stdout
        cmd.Run()

        err = os.Remove(hostPath)
        if err != nil {
            return err, ""
        }

        return err, key
    }
```

调用函数，传入节点相关的信息，返回相关的节点连接凭证，即可完成自动化部署，一个新的节点就搭建完成了。

5.12 链路自动化搭建

5.12.1 链路部署流程

为了实现链路自动化部署，以部署 3 跳节点链路为例，主要包含以下步骤：

(1) 在路由设备上通过节点连接凭证启动第 1 跳连接，成功建立后会有新的虚拟网卡

和 IP。

（2）在路由设备上配置第 2 跳出网路由，使它通过第 1 跳虚拟网卡和 IP 出网。

（3）在路由设备上通过节点连接凭证启动第 2 跳连接，成功建立后会有新的虚拟网卡和 IP。

（4）在路由设备上配置第 3 跳出网路由，使它通过第 2 跳虚拟网卡和 IP 出网。

（5）在路由设备上通过节点连接凭证启动第 3 跳连接，成功建立后会有新的虚拟网卡和 IP。

（6）在路由设备上配置相关客户端流量的出网规则并通过第 3 跳虚拟网卡和 IP 出网。

通过叠加节点，建立虚拟链路，数据包通过加解密传输，最后让客户端数据流量包通过链路的第 1 跳节点入，通过链路的第 3 跳节点出。

5.12.2　实例：节点连接

以启动 openconnect 连接为例，openconnect 自动启动脚本的内容如下：

```
#anonymous - link\code\ansible\script\start\openconnect.sh

#! /bin/sh
function config_vpn_client() {
  openconnect -c $1 -k $2 --cafile= $3 https://$4:1194 --background --interface=
tun- $5

  for ((i = 0; i < 20; i++)); do
    vpn_ip= $(ip a | grep $6 | awk '{print $2}')
    if [ -z $vpn_ip ]; then
      sleep 2
    else
      break
    fi
  done

  if [ -z $vpn_ip ]; then
    exit 2
  fi
}
#1.服务器 IP 2.client - cert.pem 3.client - key.pem 4.ca - cert.pem 5.节点虚拟子网掩码 6.虚
#拟网卡的 UUID
config_vpn_client $2 $3 $4 $1 $6 $5
```

编写调用函数，调用脚本实现节点连接，代码如下：

```
//anonymous - link\code\service\node_manage.go
func StartOpenConnect(vpnServer string, netMask string, certName string) (error, string) {
    certPath := utils.GetCurrentAbPathByCaller() + "/../cert/" + certName + "/"
      shellPath := utils.GetCurrentAbPathByCaller() + "/../ansible/script/start/
openconnect.sh"
```

```
    clientName := uuid.New().String()
    clientName = strings.ReplaceAll(clientName, "-", "")[:10]
    netMask = netMask[:strings.LastIndex(netMask, ".") + 1]
    startVpn := fmt.Sprintf("sh %s %s %s %s %s %s %s", shellPath, vpnServer, certPath
+ "client-cert.pem",
        certPath + "client-key.pem", certPath + "ca-cert.pem", netMask, clientName)
    cmd := exec.Command("sh", "-c", startVpn)
    cmd.Stdout = os.Stdout
    err := cmd.Run()
    if err != nil {
        return err, ""
    }
    return nil, "tun-" + clientName
}
```

5.12.3 创建链路

通过多个节点创建一条链路,完整的创建链路函数的代码如下:

```
//anonymous-link\code\service\link_manage.go
func CreateLink(nodes []model.NodeModel, linkId uint64) (error, *model.LinkModel) {
    link, err := model.GetLink(linkId)
    if err != nil {
        return err, nil
    }
    linkLog := model.LinkLogModel{
        LinkId:      linkId,
        LinkName:    link.LinkName,
        LogType:     model.LinkLogType().Deploy,
        AccountId:   link.AccountId,
        AccountName: link.AccountName,
    }
    var nodeInfo []string
    for i, node := range nodes {
        nodeInfo = append(nodeInfo, fmt.Sprint(node.Id))
        var data string
        var err error
        if node.VpnType == model.VpnType().OpenVpn {
            err, data = StartOpenVPN(node.ServerIP, node.Netmask, node.CertName)
        } else if node.VpnType == model.VpnType().Wireguard {
            err, data = StartWireguard(node.ServerIP, node.Netmask, node.CertName)
        } else if node.VpnType == model.VpnType().StrongSwan {
            err, data = StartStrongSwan(node.ServerIP, node.Netmask, node.CertName)
        } else if node.VpnType == model.VpnType().OpenConnect {
            err, data = StartOpenConnect(node.ServerIP, node.Netmask, node.CertName)
        } else {
            return config.ErrVpnOrCertFile, nil
        }
        if err != nil {
            linkLog.Content = err.Error()
```

```
            linkLog.Create()
            link.Status = model.LinkStatus().DeployFailed
            model.UpdateLink(link)
            return err, nil
        }

        node.ClientName = data
        node.Status = model.NodeStatus().Inuse
        if i+1 < len(nodes) {
            node.NextNode = nodes[i+1].ServerIP
        } else {
            node.NextNode = ""
        }
        w := fmt.Sprintf("id=%d", node.Id)
        uds := map[string]interface{}{"client_name": node.ClientName, "status": node.
Status, "next_node": node.NextNode}
        if err = model.UpdateNodeByWhere(w, uds); err != nil {
            linkLog.Content = err.Error()
            linkLog.Create()
            link.Status = model.LinkStatus().DeployFailed
            model.UpdateLink(link)
            return err, nil
        }
        if len(node.NextNode) > 1 {
            if err = AddNextHopRoute(node.Netmask, node.NextNode); err != nil {
                linkLog.Content = err.Error()
                linkLog.Create()
                link.Status = model.LinkStatus().DeployFailed
                model.UpdateLink(link)
                return err, nil
            }
            for count := 5; count > 0; count-- {
                err = CheckNode(node.NextNode)
                if err != nil {
                    time.Sleep(5 * time.Second)
                }
            }
        }

    }

    data, _ := json.Marshal(nodeInfo)
    link.NodeInfo = string(data)
    link.Status = model.LinkStatus().Free
    err, _ = model.UpdateLink(link)
    if err != nil {
        return err, nil
    }
    return nil, link
}
```

传入相关的节点模型信息，调用相关的 VPN 服务创建链路，返回链路模型。

5.13 路由控制及实现

为了拥有灵活多变的路由和流量控制策略，可以通过以下方式实现：

(1) 利用 route 和 iptables 进行路由设置及转发。

(2) 白名单、入网、出网规则控制。

(3) VPN 多链路建立及数据流量转发。

5.13.1 配置默认链路出网策略

为路由设备配置默认链路出网，如果有其他终端连接到路由设备，则可以通过默认链路出网。默认路由转发规则的脚本内容如下：

```
#anonymous-link\code\ansible\script\forward\add-default.sh

#! /bin/sh

count=$(ip route | grep "default dev $1" | wc -l)
if [ $count -gt 0 ]; then
  ip route del default dev $1
fi

if [ -z $2 ]; then
  ip route add default dev $1
else
  ip route add default dev $1 metric $2
fi

count=$(iptables -t nat -L POSTROUTING -nv | grep $4 | wc -l)
if [ $count -gt 0 ]; then
  rule_ids=($(iptables -t nat -L POSTROUTING -nv --line-number | grep $4 | awk '{print
$1}'))
  for ((i = ${#rule_ids[@]} - 1; i >= 0; i--)); do
    iptables -t nat -D POSTROUTING ${rule_ids[i]}
  done
fi

count=$(iptables -L FORWARD -nv | grep $4 | wc -l)
if [ $count -gt 0 ]; then
  rule_ids=($(iptables -L FORWARD -nv --line-number | grep $4 | awk '{print $1}'))
  for ((i = ${#rule_ids[@]} - 1; i >= 0; i--)); do
    iptables -D FORWARD ${rule_ids[i]}
  done
fi

iptables -A FORWARD -s $3 -o $1 -j ACCEPT -m comment --comment $4
iptables -t nat -A POSTROUTING -s $3 -o $1 -j MASQUERADE -m comment --comment $4
```

通过封装函数进行调用，实现 API 的调用和前端界面操作的控制，实现函数如下：

```
func SetDefaultRouteLink(id uint64) error {
    link, err := model.GetLink(id)
    if err != nil {
        return err
    }
    var nodes []string
    err = json.Unmarshal([]Byte(link.NodeInfo), &nodes)
    if err != nil || len(nodes) < 1 {
        return err
    }
    nodeId, err := strconv.ParseUint(nodes[len(nodes) - 1], 10, 64)
    if err != nil {
        return err
    }
    node, err := model.GetNode(nodeId)
    if err != nil {
        return err
    }
     shellPath := utils.GetCurrentAbPathByCaller() + "/../ansible/script/forward/add -
default.sh"
    op := fmt.Sprintf("sh %s %s %d %s %s", shellPath, node.ClientName, config.
DefaultLinkRouteMetric,
        config.DefaultLinkSubnet, config.DefaultLinkUid)
    cmd := exec.Command("sh", "-c", op)
    cmd.Stdout = os.Stdout
    err = cmd.Run()
    return err
}
```

5.13.2　按源 IP 分流出网策略

根据连接到路由设备的终端 IP，为每个终端制定不同的出网策略。配置指定终端 IP 出网策略的脚本内容如下：

```
#anonymous - link\code\ansible\script\strategy\connect.sh

#! /bin/bash

iptables -A FORWARD -s $1 -o $2 -j ACCEPT -m comment --comment $3
iptables -t nat -A POSTROUTING -s $1 -o $2 -j MASQUERADE -m comment --comment $3
proxy_ip=$(ifconfig | grep $2 -A 2 | grep netmask |awk '{print $2}')
ip route add $6 via $proxy_ip dev $2 metric $5 table $4
if [ -z $7 ]; then
  ip rule add from $1 table $4
else
  ip rule add from $1 table $4 prio $7
fi
```

通过封装函数进行调用，实现 API 的调用和前端界面操作的控制，实现函数如下：

```
//anonymous - link\code\service\socket_manage.go

func AddStrategyRule(sourceIp, nicName, uniqueLabel, desIp string, routeTable, metric uint)
error {
    shellPath := utils.GetCurrentAbPathByCaller() + "/../ansible/script/strategy/connect.
sh"
    op := fmt.Sprintf("sh %s %s %s %s %d %d %s %d", shellPath, sourceIp, nicName,
uniqueLabel, routeTable,
        metric, desIp, config.DefaultStrategyPriority + metric)
    cmd := exec.Command("sh", "-c", op)
    cmd.Stdout = os.Stdout
    err := cmd.Run()
    return err
}
```

5.13.3 按源 IP 范围分流策略

根据连接到路由设备的终端 IP，为每个终端制定不同的出网策略。配置按源 IP 范围分流策略的脚本内容跟按源 IP 分流策略出网一样，不同的是传入的参数控制源 IP。

通过封装函数进行调用，实现 API 的调用和前端界面操作的控制，实现函数如下：

```
//anonymous - link\code\service\socket_manage.go
func AddStrategyRuleSourceIpRange(startIp, endIp string, linkId uint64, metric uint,
strategyId uint64) error {
    //暂时只支持前三位相同的 IP 范围
    IpSplitStart := strings.Split(startIp, ".")
    IpSplitEnd := strings.Split(endIp, ".")
    ipS := IpSplitStart[len(IpSplitStart) - 1]
    ipFront := startIp[:len(startIp) - len(ipS)]
    ipList := make([]string, 0)
    ipNumS, err := strconv.Atoi(ipS)
    if err != nil {
        panic(err)
    }
    ipE := IpSplitEnd[len(IpSplitEnd) - 1]
    ipNumE, err := strconv.Atoi(ipE)
    if err != nil {
        panic(err)
    }
    for ipNumS <= ipNumE {
        mip := ipFront + strconv.Itoa(ipNumS)
        ipList = append(ipList, mip)
        ipNumS++
    }
    //查出 db 中所有源 IP 类型的策略，排除 db 中已存在的
    sql := "SELECT DISTINCT device_ip as ip FROM strategy_info where s_type = 1"
    ipInfo, err := model.RawSql(sql)
```

```
    if err != nil {
        return err
    }
    ipDbMap := make(map[string]string, 0)
    for _, ip := range ipInfo {
        ipDbMap[ip["ip"].(string)] = ip["ip"].(string)
    }

    for _, ip := range ipList {
        if _, ok := ipDbMap[ip]; !ok {
            err = AddStrategyRuleNotRange(ip, "default", linkId, metric, strategyId)
            if err != nil {
                return err
            }
        }
    }
    return nil
}


func AddStrategyRuleNotRange(deviceIp, desIp string, linkId uint64, metric uint, strategyId
uint64) error {

    node, err := GetStrategyLinkNode(linkId)
    if err != nil {
        return err
    }
    err, routeTable := GetStrategyRouteTable()
    if err != nil {
        return err
    }

    key := uuid.New().String()
    key = strings.ReplaceAll(key, "-", "")

    err = AddStrategyRule(deviceIp, node.ClientName, key, desIp, routeTable, metric)
    if err != nil {
        return err
    }

    strategyUuid := model.StrategyUuidModel{
        StrategyId: strategyId,
        Uuid:       key,
        TableId:    routeTable,
    }
    err, _ = strategyUuid.Create()
    if err != nil {
        return err
    }

    return nil
}
```

5.13.4 按目标 IP 分流出网策略

根据连接到路由设备的终端 IP,为每个终端制定不同的出网策略。配置按目标 IP 分流出网策略的脚本内容跟按源 IP 分流策略出网一样,不同的是传入的参数控制目标 IP。

5.13.5 按目标网段分流出网策略

根据连接到路由设备的终端 IP,为每个终端制定不同的出网策略。配置按目标网段分流出网策略的脚本内容跟按源 IP 分流策略出网一样,不同的是传入的参数控制目标网段。

不同分流出网策略在调用时,传入的参数不一样,分流策略调用函数的内容如下:

```
//anonymous - link\code\handler\strategy_manage.go

//AddStrategy
//@Summary 新增分流策略
//@Description 新增分流策略
//@Tags 分流策略管理
//@schemes http https
//@Accept json
//@Produce json
//@Response 200 {object} config.Response
//@Param rule query string true "策略名称"
//@Param link_id query int true "链路 ID"
//@Param link_name query string true "链路名称"
//@Param strategy_type query int true "策略类型"
//@Param device_ip query string false "设备 IP"
//@Param start_ip query string false "起始 IP"
//@Param end_ip query string false "结束 IP"
//@Param des_ip query string false "目的 IP"
//@Param des_subnet query string false "目的网段"
//@Param priority query int true "优先级"
//@Param status query int true "策略状态"
//@Router /strategy/create [post]
func AddStrategy(c *base.Context) {
    //策略条件说明:
    //(1)按照源 IP 分流,下拉列表选择设备 IP
    //(2)按照源 IP 范围,两个输入框一排,分别代表开始和结束,校验输入的 IP
    //(3)按照目标 IP,下拉列表选择设备 IP,并且输入目标 IP 地址,校验公网 IP
    //(4)按照目标网段,下拉列表选择设备 IP,并且输入目标网段,校验网段
    rule := c.ArgsString("rule")
    linkId := c.ArgsUint("link_id")
    linkName := c.ArgsString("link_name")
    strategyType := c.ArgsUint("strategy_type")
    deviceIp := c.ArgsStringDefault("device_ip", "")
    startIp := c.ArgsStringDefault("start_ip", "")
    EndIp := c.ArgsStringDefault("end_ip", "")
    desIp := c.ArgsStringDefault("des_ip", "")
    desSubnet := c.ArgsStringDefault("des_subnet", "")
    priority := c.ArgsUint("priority")
```

```
status := c.ArgsUint("status")
user, _ := c.Get("account_info")
ac := user.(model.AccountModel)
if strategyType == model.StrategyType().SourceIpRange {
    deviceIp = "empty"
}
err := service.CheckStrategyInfo(rule, priority, status)
if err != nil {
    SendResponse(c, err, nil)
    return
}

strategy := model.StrategyModel{
    Rule:        rule,
    LinkId:      uint64(linkId),
    LinkName:    linkName,
    Priority:    priority,
    Status:      status,
    AccountId:   ac.Id,
    AccountName: ac.Username,
    SType:       strategyType,
}

switch strategyType {
case model.StrategyType().SourceIp:
    err = service.CheckSourceIpStrategy(deviceIp, linkId)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }
    strategy.DeviceIp = deviceIp
case model.StrategyType().SourceIpRange:
    err = service.CheckSourceIpRangeStrategy(startIp, EndIp, linkId)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }
    strategy.StartIp = startIp
    strategy.EndIp = EndIp
case model.StrategyType().DesIp:
    err = service.CheckDesIpStrategy(desIp, deviceIp, linkId)
    if err != nil {
        SendResponse(c, err, nil)
        return
    }
    strategy.DesIp = desIp
    strategy.DeviceIp = deviceIp
case model.StrategyType().DesSubnet:
    err = service.CheckDesSubnetStrategy(desSubnet, deviceIp, linkId)
    if err != nil {
        SendResponse(c, err, nil)
```

```
            return
        }
        strategy.DesSubnet = desSubnet
        strategy.DeviceIp = deviceIp
    }

    err, strateId := strategy.Create()

    if err != nil {
        SendResponse(c, err, nil)
        return
    }

    //如果创建时处于启用状态,则直接下发 ip route 命令
    if status == model.StrategyStatus().Inuse {
        switch strategyType {
        case model.StrategyType().SourceIp:
             err = service.AddStrategyRuleNotRange(deviceIp, "default", uint64(linkId),
priority, strateId)
            if err != nil {
                SendResponse(c, err, nil)
                return
            }
        case model.StrategyType().SourceIpRange:
             err = service.AddStrategyRuleSourceIpRange(startIp, EndIp, uint64(linkId),
priority, strateId)
            if err != nil {
                SendResponse(c, err, nil)
                return
            }
        case model.StrategyType().DesIp:
               err = service.AddStrategyRuleNotRange(deviceIp, desIp, uint64(linkId),
priority, strateId)
            if err != nil {
                SendResponse(c, err, nil)
                return
            }
        case model.StrategyType().DesSubnet:
             err = service.AddStrategyRuleNotRange(deviceIp, desSubnet, uint64(linkId),
priority, strateId)
            if err != nil {
                SendResponse(c, err, nil)
                return
            }
        }
    }

    SendResponse(c, nil, strateId)
    return
}
```

5.14　离线自动化升级

考虑到每个用户购买软件后的使用时间不一样，需要生成不一样的使用证书，而且每个用户内置的节点和链路不一样；也为了运营平台的匿名性和安全性，不采用集中在线升级的方式，而采用离线加密的方式进行升级。

1. 离线升级流程

离线升级流程如下：

(1) 解压升级文件。

(2) 用私钥解密节点连接凭证文件。

(3) 用私钥解密程序配置文件。

(4) 用私钥解密程序数据文件。

(5) 用私钥解密自动化脚本文件。

(6) 将相关文件复制并替换到程序设定的相关目录。

(7) 删除相关的文件。

(8) 重启服务，完成升级。

2. 代码实现

主要实现代码如下：

```
//anonymous - link\code\service\upgrade_offline.go
func UpgradeOffline(filePath, destDir, comFile string) error {
    err := DecompressFile(filePath, destDir)
    if err != nil {
        return err
    }
    fileName := CompressFileName(comFile)
    fileDir := destDir + fileName
    files, err := ioutil.ReadDir(fileDir)
    if err != nil {
        return err
    }
    for _, file := range files {
        if file.IsDir() {
            if file.Name() == "cert" {
                certDir := fmt.Sprintf("%s/%s/", fileDir, file.Name())
                fs, _ := ioutil.ReadDir(certDir)
                for _, f := range fs {
                    if f.IsDir() {
                        cfDir := fmt.Sprintf("%s%s/", certDir, f.Name())
                        cfs, _ := ioutil.ReadDir(cfDir)
```

```
                        for _, c := range cfs {
                            if ! c.IsDir() {
                                sf := cfDir + c.Name()
                                df := cfDir + c.Name() + ".bak"
                                err = utils.DecryptFile(sf, df)
                                if err != nil {
                                    return err
                                }
                                os.Remove(sf)
                                os.Rename(df, sf)
                            }
                        }
                    }
                }
            } else if file.Name() == "conf" || file.Name() == "scheme" {
                fDir := fmt.Sprintf("%s/%s/", fileDir, file.Name())
                fs, _ := ioutil.ReadDir(fDir)
                for _, f := range fs {
                    if ! f.IsDir() {
                        sf := fDir + f.Name()
                        df := fDir + f.Name() + ".bak"
                        err = utils.DecryptFile(sf, df)
                        if err != nil {
                            return err
                        }
                        os.Remove(sf)
                        os.Rename(df, sf)
                    }
                }
            } else if file.Name() == "ansible" {
                ansibleDir := fmt.Sprintf("%s/%s/", fileDir, file.Name())
                fs, _ := ioutil.ReadDir(ansibleDir)
                for _, f := range fs {
                    if f.IsDir() {
                        cfDir := fmt.Sprintf("%s%s/", ansibleDir, f.Name())
                        cfs, _ := ioutil.ReadDir(cfDir)
                        for _, c := range cfs {
                            if c.IsDir() {
                                scDir := fmt.Sprintf("%s%s/", cfDir, c.Name())
                                sfs, _ := ioutil.ReadDir(scDir)
                                for _, s := range sfs {
                                    if ! s.IsDir() {
sf := scDir + s.Name()
df := scDir + s.Name() + ".bak"
err = utils.DecryptFile(sf, df)
if err != nil {
    return err
}
os.Remove(sf)
os.Rename(df, sf)
                                    }
```

```
                                    }
                                }
                            }
                        }
                    }
                } else if file.Name() == "html" {
                    op := fmt.Sprintf("/bin/cp -rf %s/html/* %shtml/ && rm -rf %s/html",
fileDir, config.FrontendDir, fileDir)
                    cmd := exec.Command("sh", "-c", op)
                    cmd.Stdout = os.Stdout
                    err = cmd.Run()
                    if err != nil {
                        return err
                    }
                } else if file.Name() == "license" {
                    licenseDir := config.LicenseFile[:strings.LastIndex(config.LicenseFile,
"/") + 1]
                    op := fmt.Sprintf("mkdir -p %s;/bin/cp -rf %s/license/* %s && rm -rf
%s/license", licenseDir, fileDir, licenseDir, fileDir)
                    cmd := exec.Command("sh", "-c", op)
                    cmd.Stdout = os.Stdout
                    err = cmd.Run()
                    if err != nil {
                        return err
                    }
                }
            }
        }

        files, err = ioutil.ReadDir(fileDir)
        if err != nil {
            return err
        }

        if len(files) > 0 {
            op := fmt.Sprintf("/bin/cp -rf %s/* %s", fileDir, config.ProgramDir)
            cmd := exec.Command("sh", "-c", op)
            cmd.Stdout = os.Stdout
            err = cmd.Run()
            if err != nil {
                return err
            }
        }

        op := fmt.Sprintf("rm -rf %s", destDir)
        cmd := exec.Command("sh", "-c", op)
        cmd.Stdout = os.Stdout
        err = cmd.Run()
        return err
    }
```

5.15 IP 全球定位系统

IP 定位系统主要根据世界城市数据库、多语言数据库、IP 数据库、地图数据库等确定所在位置。IP 地理定位 API 提供 IP 地址的位置信息，如国家、地区、城市、邮政编码等。IP 地理位置是一种查找访问者地理位置信息的技术，例如国家、地区、城市、邮政编码、纬度、经度、域名、ISP、区号、移动数据、天气数据、使用类型、代理数据、海拔等，通过 IP 地址实现。该 IP 查找数据源可以以各种形式找到，例如，数据库、文件和网络服务，以供用户构建地理定位解决方案。该技术被广泛用于防火墙、域名服务器、广告服务器、路由、邮件系统、网站及地理定位等可能有用的其他自动化系统。

W3C 地理查询 API 是由 W3C 创新研究和研发的，用来探测客户端的地理位置信息，而最常见的来源不外是 IP 地址、WiFi 和蓝牙 MAC 地址、无线射频识别(RFID)、WiFi 连接的位置、全球定位系统(GPS)和 GSM / CDMA 蜂窝小区 ID。精确的数据回传有赖于最佳位置信息的位置来源。

地理编码与地理位置是息息相关的。查询过程当中需要通过其他地理数据，如城市或街道地址等，探讨相关联的地理坐标(纬度和经度)，再回传到地理数据系统。也可以通过相关的地理坐标而获取世界各大主要城市列表。

IP2Location Geolocation API 使用举例：

```
$ curl
"https://api.ip2location.com/v2/? ip = 220.181.41.72&key = {YOUR_API_KEY}&pack
age = WS25"
```

返回结果如下：

```
{
    "country_code":"CN",
    "country_name":"China",
    "region_name":"Beijing",
    "city_name":"Beijing",
    "latitude":"39.9075",
    "longitude":"116.39723",
    "zip_code":"100006",
    "time_zone":" + 08:00",
    "isp":"ChinaNet Beijing Province Network",
    "domain":"chinatelecom.com.cn",
    "net_speed":"DSL",
    "idd_code":"86",
    "area_code":"010",
    "weather_station_code":"CHXX0008",
    "weather_station_name":"Beijing",
    "mcc":"460",
    "mnc":"03/11",
    "mobile_brand":"China Telecom",
```

```
    "elevation":"49",
    "usage_type":"ISP/MOB",
    "address_type":"U",
    "category":"IAB19-18",
    "category_name":"Internet Technology",
    "credits_consumed":20
}
```

IP2Proxy Geolocation API 使用举例，代码如下：

```
$ curl
"https://api.ip2proxy.com/?key={YOUR_API_KEY}&ip=220.181.41.72&package=PX
11&format=json"
```

返回结果如下：

```
{
    "response":"OK",
    "countryCode":"CN",
    "countryName":"China",
    "regionName":"Beijing",
    "cityName":"Beijing",
    "isp":"ChinaNet Beijing Province Network",
    "domain":"chinatelecom.com.cn",
    "usageType":"ISP/MOB",
    "asn":"-",
    "lastSeen":"-",
    "proxyType":"-",
    "threat":"-",
    "isProxy":"NO",
    "provider":"-"
}
```

数据库文件下载网页网址为 http://dev.maxmind.com/geoip/geoip2/geolite2/。

打开下载页面，如图 5-38 所示，可以根据业务需要选择下载不同种类的数据库文件，结合代码进行使用。

Downloads

Database	MaxMind DB binary, gzipped	CSV format, zipped
GeoLite2 City	Download (md5 checksum)	Download (md5 checksum)
GeoLite2 Country	Download (md5 checksum)	Download (md5 checksum)
GeoLite2 ASN (Autonomous System Number)	Download (md5 checksum)	Download (md5 checksum)

The GeoLite2 databases may also be downloaded and updated with our GeoIP Update program.

图 5-38　IP 地理位置数据库下载页面

国家地理 IP 数据库的下载网址为 http://geolite.maxmind.com/download/geoip/database/GeoLite2-City.tar.gz。

城市地理 IP 数据库的下载网址为 http://geolite.maxmind.com/download/geoip/database/GeoLite2-Country.tar.gz。

服务运营商地理 IP 数据库的下载网址为 http://geolite.maxmind.com/download/geoip/database/GeoLite2-ASN.tar.gz。

自动更新数据库程序的网址为 https://dev.maxmind.com/geoip/geoipupdate/。

源码 GitHub 的网址为 https://github.com/maxmind/geoipupdate。

数据操作 API 库资源网址为 https://dev.maxmind.com/geoip/geoip2/downloadable/#MaxMind_APIs。

官方 C 操作库源码 GitHub 的网址为 https://github.com/maxmind/libmaxminddb。

文档参考网址为 http://maxmind.github.io/libmaxminddb/。

非官方 C++操作库下载网址为 https://www.ccoderun.ca/GeoLite2PP/download/?C=M;O=D。

API 文档参考网址为 https://www.ccoderun.ca/GeoLite2++/api/。

GeoIP 库在很多系统里支持以软件的形式安装，这里以 CentOS 系统为例，安装命令如下：

```
yum install -y GeoIP GeoIP-devel GeoIP-data
```

GeoIP 数据库文件一般保存在/usr/share/GeoIP/ 目录下，因为 IP 信息是经常变动的，所以需要经常更新数据库文件，查询 IP 所在位置，以 8.210.174.64 为例，命令如下：

```
geoiplookup -f /usr/share/GeoIP/GeoIP.dat 8.210.174.64
```

把 GeoIP 封装为函数，供系统调用查询，代码如下：

```
//anonymous-link\code\service\dhcp_manage.go
func SearchIPInfo(ip string) (error, map[string]map[string]string) {
    cmd := exec.Command("geoiplookup", ip)
    var stdout Bytes.Buffer
    cmd.Stdout = &stdout
    err := cmd.Run()
    if err != nil {
        return err, nil
    }
    info := string(stdout.Bytes())
    infos := strings.Split(info, "\n")
    if len(infos) < 3 {
        return fmt.Errorf(info), nil
    }
    allIPs := make(map[string]map[string]string)
    ipInfo := map[string]string{
        "country": strings.Split(infos[0], ":")[1],
        "city":    strings.Split(infos[1], ":")[1],
        "asnum":   strings.Split(infos[2], ":")[1],
```

```
        }
        for k, v := range ipInfo {
            if strings.Contains(v, "not found") || strings.Contains(v, "can't") {
                ipInfo[k] = ""
            }
        }

        allIPs[ip] = ipInfo
        return err, allIPs
    }
```

5.16　网络联通状态监测

因特网控制消息协议(Internet Control Message Protocol,ICMP)是 TCP/IP 协议簇的一个子协议,用于在 IP 主机、路由器之间传递控制消息。

ping(Packet Internet Groper)为因特网包探索器,是一种用于测试网络连接量的程序。ping 发送一个 ICMP;回声请求消息给目的地并报告是否收到所希望的 ICMP echo(ICMP 回声应答)。它是用来检查网络是否通畅或者检测网络连接速度的命令。

ping 是 Windows、UNIX 和 Linux 系统下的一个命令。ping 也属于一种通信协议,是 TCP/IP 的一部分。利用 ping 命令可以检查网络是否连通,可以很好地帮助用户分析和判定网络故障。基于 LCMP。

1. 命令格式

命令格式如下:

```
ping [参数] [主机名或 IP 地址]
```

2. 主要参数

常用参数-q 表示不显示任何传送封包的信息,只显示最后的结果。

-n:只输出数值。

-R:记录路由过程。

-c:在发送指定数目的包后停止。

-i:设定间隔几秒将一个网络封包发送给一台机器,预设值是一秒发送一次。

-t:设置存活数值 TTL 的大小。

-s:指定发送的数据字节数,预设值是 56。

加上 8 字节的 ICMP 头,一共是 64ICMP 数据字节。

3. 使用案例

(1) 百度网站联通测试,代码如下:

```
ping www.baidu.com
```

按快捷键 Ctrl+C 终止。

(2) 指定 ping 命令执行的次数,代码如下:

```
ping -c 5 www.baidu.com
```

发送完 5 个包后终止。

(3) 指定 0.5s 发一个包,代码如下:

```
ping -c 10 -i 0.5 www.baidu.com
```

(4) 指定包大小为 1024,代码如下:

```
ping -c 10 -i 0.5 -s 1024 www.baidu.com
```

在系统中利用 ping 检测网络联通性,需要考虑到 ping 有时会丢包,同时要考虑程序的通用性,把网络检测功能写成一个脚本,代码如下:

```
#anonymous-link\code\ansible\script\route\ping.sh

#! /bin/sh

function check_net() {
  receive=$(ping -n -c 5 $1 | grep received | awk -F"," '{print $2}' | awk '{print $1}')
  if [ $receive -gt 2 ]; then
    exit 0
  else
    exit 2
  fi
}

check_net $1
```

把脚本封装为函数,供系统调用,代码如下:

```
//anonymous-link\code\service\node_manage.go
func CheckNode(serverIp string) error {
    shellPath := utils.GetCurrentAbPathByCaller() + "/../ansible/script/route/ping.sh"
    op := fmt.Sprintf("sh %s %s", shellPath, serverIp)
    cmd := exec.Command("sh", "-c", op)
    cmd.Stdout = os.Stdout
    err := cmd.Run()
    return err
}
```

5.17 构建虚拟环境开发

因为系统深度依赖于 Linux 内核,所以在开发过程中需要有 Linux 系统环境,这里以 VMware 虚拟机安装开发环境为例,安装完 Linux 系统之后,把主机上的代码目录共享到虚

拟机内部进行运行，这样就实现了在主机上通过原有的代码编辑器进行系统开发，运行环境在虚拟机内部，可以查看运行结果。

配置虚拟机共享代码目录开发环境，步骤如下：

（1）单击“虚拟机”→“安装 VMware Tools”→“确定挂载对应的 CD/ROM 设备”。

（2）在虚拟机系统里查看对应的设备：lsblk。

（3）在虚拟机系统里挂载对应的设备：mkdir /root/tool && mount /dev/sr0 /root/tool。

（4）在虚拟机系统里解压对应的文件：cd /root/tool/ && tar-C /root/-zxvf VMwareTools-10.3.21-14772444.tar.gz。

（5）在虚拟机系统里安装程序依赖环境：yum-y install kernel-headers kernel-devel kernel gcc gcc-c++ perl make。

（6）在虚拟机系统里安装对应的程序：cd /root/vmware-tools-distrib/ && ./vmware-install.pl。

安装过程如下：

```
Do you still want to proceed with this installation? [no] yes

In which directory do you want to install the binary files?
[/usr/bin] 按 Enter 键

What is the directory that contains the init directories (rc0.d/ to rc6.d/)?
[/etc/rc.d] 按 Enter 键

What is the directory that contains the init scripts?
[/etc/rc.d/init.d] 按 Enter 键

In which directory do you want to install the daemon files?
[/usr/sbin] 按 Enter 键

In which directory do you want to install the library files?
[/usr/lib/vmware-tools] 按 Enter 键

The path "/usr/lib/vmware-tools" does not exist currently. This program is
going to create it, including needed parent directories. Is this what you want?
[yes] 按 Enter 键

In which directory do you want to install the common agent library files?
[/usr/lib] 按 Enter 键

In which directory do you want to install the common agent transient files?
[/var/lib] 按 Enter 键

In which directory do you want to install the documentation files?
```

```
[/usr/share/doc/vmware-tools] 按 Enter 键

The path "/usr/share/doc/vmware-tools" does not exist currently. This program
is going to create it, including needed parent directories. Is this what you
want? [yes] 按 Enter 键

Before running VMware Tools for the first time, you need to configure it by
invoking the following command: "/usr/bin/vmware-config-tools.pl". Do you want
this program to invoke the command for you now? [yes] 按 Enter 键

The VMware Host-Guest Filesystem allows for shared folders between the host OS
and the guest OS in a Fusion or Workstation virtual environment. Do you wish
to enable this feature? [yes] 按 Enter 键

The path "/bin/gcc" appears to be a valid path to the gcc binary.
Would you like to change it? [no] 按 Enter 键

The path "" is not a valid path to the 3.10.0-862.el7.x86_64 Kernel headers.
Would you like to change it? [yes] no

- This program could not find a valid path to the Kernel headers of the running
  Kernel. Please ensure that the header files for the running Kernel are
  installed on this system.
[ Press Enter key to continue ] 按 Enter 键

If you wish to have the shared folders feature, you can install the driver by
running vmware-config-tools.pl again after making sure that gcc, binutils, make
and the Kernel sources for your running Kernel are installed on your machine.
These packages are available on your distribution's installation CD.
[ Press Enter key to continue ] 按 Enter 键

The vmblock enables dragging or copying files between host and guest in a
Fusion or Workstation virtual environment. Do you wish to enable this feature?
[yes] 按 Enter 键

Do you want to enable Guest Authentication (vgauth)? Enabling vgauth is needed
if you want to enable Common Agent (caf). [yes] 按 Enter 键

Do you want to enable Common Agent (caf)? [no] 按 Enter 键
```

关闭虚拟机系统。

(7) 选择“虚拟机”→“设置”→“共享”,单击“+”按钮添加对应的文件夹路径,并勾选“启用共享文件夹”。

(8) 在虚拟机系统里进入项目目录,查看共享文件夹。

(9) 安装相关的系统开发依赖包,该项目运行安装依赖的命令如下:

```
go mod tidy
```

(10) 启动系统进行测试,命令如下:

```
sh start.sh
```

5.18　熟练使用 Linux 磁盘工具

Linux 系统下支持很多种磁盘格式，每种磁盘格式有不同的优势，也有不同的用法，常见的磁盘格式类型及参数如下。

5.18.1　ext4 磁盘格式

ext4 磁盘格式选项及属性如下：

```
mount -o options device directory
          Option:Description
          async:容许文件系统异步地输入与输出
          auto:Allows the file system to be mounted automatically using the mount -a command.
          defaults:Provides an alias for async,auto,dev,exec,nouser,rw,suid.
          exec:容许二进制文件执行
          loop:把镜像文件回环设备挂载
          noauto:Default behavior disallows the automatic mount of the file system using the
mount -a command.
          noexec:不容许二进制文件执行
          nouser:禁止普通用户 mount 与 umount
          remount:重新挂载
          ro:只读
          rw:容许读写
          user: 容许普通用户 mount 与 umount
          acl:访问控制列表
          commit = nsec 文件系统 Cache 刷新时间
          stripe = 条带大小(以 block 为单位)
          delalloc 开启延时块分配
          nodelalloc 禁止延时块分配
          barrier 开启 write barrier
          nobarrier 禁止 write barrier
          journal_dev = devnum (外部日志设备的设备号,由主次设备号组成)
```

模式如下：

```
data = writeback 性能,高; 写回模式,先写 metadata(表明日志),后写 data
data = ordered 性能,中; 命令模式,[先写 data,后写 metadata] == 事务,最后写 metadata journal
data = journal 性能,低: 日志模式, 先日志(metadata journal,data journal),后数据(metadata,
data)
```

挂载操作系统镜像举例，命令如下：

```
mount -o ro,loop Fedora-14-x86_64-Live-Desktop.iso /media/cdrom
```

磁盘操作属性如下：

```
mkfs.ext4 -b block-size 块大小(1k,2k,4k)
          -c 坏块测试
          -l filename 从文件读取坏块列表
          -C cluster-size 簇大小 (大块分配特性)
          -D 使用 direct I/O
          -E 扩展属性
               mmp_update_interval = MMP 更新时间间隔,必须小于 300s
               stride = 条块大小(RAID 组中每个条带单元 chunk 的大小)
               stripe_width = 条带大小 (单位为 block),在数据确定时,块分配器尽量地防止产
生 read-modify-write
               resize = 保留在线调整时的空间大小
               lazy_itable_init = 0/1 inode 表不彻底初始化 (挂载时由内核在后台初始化)
//40TG mount 后 50M 写初始化 55min (格式化时:20s,强制初始化时:7min) ( mkfs.ext4 -E lazy_
itable_init = 0,lazy_journal_init = 0 )
               lazy_journal_init = 0/1 日志 inode 表不彻底清零
               test_fs 设置文件系统体验标志

            -F(force 强制)
            -f fragment-size 指定片段大小
            -g blocks-per-group 指定每个块组内块的数量
            -G number-of-groups 指定块组数量(在元数据负载重时能够提升元数据性能)
            -i Bytes-per-inode 指定 Bytes/inode 比率
            -I inode-size 指定 inode 大小
            -j 建立一个 ext3 日志。默认建立合适大小的日志区
             -J 建立指定属性的日志。逗号分隔(size = 1024 块 内部日志大小,device = 外部日
志设备)
                    size = journal-size 内部日志大小,单位为 M,最小为 1024 个文件系统块,
最大为 10 240 000 个文件系统块或文件系统的一半
                    device = external-journal 外部日志块设备(设备名、标签、UUID)
                        外部日志必须先建立:mke2fs -b 4096 -O journal_dev external-
journal (/dev/ramhda)
                         mkfs.ext4 -J device = external-journal (/dev/ramhda) -F /dev/
mapper/vggxxxxxxx
            -L 设置 volume 标签,最长为 16 字节
            -m 指定保留空间百分比,为 root 用户
            -M 设置最后挂载目录
             -n 不真正建立文件系统,只是显示建立的信息
             -S 只写超级块和块组描述符(当超级块和备份超级块错误后,能够用来恢复数据,这
是由于它不会 touching inode 表和 bitmap)
              -O feature 指定建立文件系统时的特性(/etc/mke2fs.conf)
                    bigalloc 使能大块分配(cluster-size)
                    dir_index 使用哈希 B 树加速目录查找
                    extents 使用 extents 替代间接块
                    filetype 在目录项中存储文件类型信息
                    flex_bg 容许为每个块组元数据(分配 bitmap 和 inode 表)存放在任何位置
                    has_journal 建立 ext3 日志(-j)
                    journal_dev 在给定的设备上建立外部 ext3 日志
                    large_file 支持>2GB 的文件(现代内核会自动打开)
                    quota 建立 quota inodes(inode#3 为用户配额,inode#4 为组配额),并在
超级块中设置
                                        (挂载后本身启用 quota)
```

```
                    resize_inode 保留空间以便将来块组描述表增加,用于 resize2fs
                    sparse_super 建立少许的超缓块复制
                    uninit_bg 建立文件系统时不初始化全部的块组,加速建立时间,和 e2fsck
时间
                -O^has_journal 不启用日志
```

在/proc/fs/ext4/设备的/options 中查看已挂载文件系统的属性：

```
rw 文件系统挂载时的读写策略
    delalloc 开启延时块分配
    barrier 开启 write barrier(提供写顺序)
    user_xattr
    acl
    resuid=0 可使用保留块的用户 ID
    resgid=0 可使用保留块的组 ID
    errors=continue 文件系统出错时动作
    commit=5 文件系统刷 cache 的时间间隔
    max_batch_time=15000us 最大的等待合并一块儿提交的时间,默认为 15ms
    min_batch_time=0us 最小的等待合并一块儿提交的时间,0μs
    stripe=0 多块分配时和对齐的块数,对于 raid5/6,大小为数据磁盘数*chunk 大小
    data=ordered 文件系统挂载模式
    inode_readahead_blks=32 先行读入缓冲器缓存(buffer cache)的 inode 表块(table block)数
的最大值
    init_itable=10
    max_dir_size_kb=n 目录大小限制

    mb_order2_req=2 对于大于该值(2 的幂)的块,要求使用 Buddy 检索
    lifetime_write_kBytes 文件系统生成后写入的数据量(KB)
    mb_stats 指定收集(1)或不收集(0)多块分配的相关统计信息。统计信息在卸载时显示 0(禁用)
    max_writeback_mb_bump 进行下一次 inode 处理前尝试写入磁盘的数据量的最大值(MB) 128
    mb_stream_req=0 块数小于该值的文件群被集中写入磁盘上相近的区域
    mb_group_prealloc 未指定挂载选项的 stripe 参数时,以该值的倍数为单位确保块的分配(512)
    session_write_kBytes 挂载后写入文件系统的数据量(KB)
```

在/sys/fs/ext4/设备中查看：

```
    mb_stream_req=16 块数小于该值的文件群被集中写入磁盘上相邻的区域
    inode_readahead_blks=32 控制进行预读的 inode 表的数量
    inode_goal 下一个要分配的 inode 编号(调试用) 0(禁用)
    delayed_allocation_blocks 等待延迟分配的块数
    max_writeback_mb_bump=128 进行下一次 inode 处理前尝试写入磁盘数据量的最大值(MB)
    mb_group_prealloc=512 未指定 stripe 参数时,以该值的倍数为单位确保块的分配
    mb_max_to_scan=200 分配多块时为找出最佳 extent 而搜索的最大 extent 数
    mb_min_to_scan=10 分配多块时为找出最佳 extent 而搜索的最小 extent 数
    mb_order2_req=2 对于大于该值的块(2 的幂),要用 buddy 算法
    mb_stats=0 指定收集 1,与不收集 0 多块分配的相关统计信息,统计信息会在卸载时显示
    reserved_clusters
    lifetime_write_kBytes 只读,记录已经写入文件系统的数据量(KB)
    session_write_kBytes 只读,记录此记挂载以来已写入的数据(KB)
```

ext2/ext3/ext4 扩大文件系统：

```
resize2fs
```

使用 fsadm 检查或调整大小的文件系统工具。支持 ext2/ext3/ext4/ ReiserFS/XFS，命令如下：

```
fsadm [options] check device 检查设备
fsadm [options] resize device [new_size[BKMGTEP]]
    options 选项有
        -e 在调整大小前先卸载 ext2/ext3/ext4 文件系统
        -f force 绕过一些检查
        -h 显示帮助信息
        -n 只打印命令,不执行
        -v 打印更多信息
        -y yes 对任何提示均回答 yes
```

示例代码如下：

```
fsadm -e -y resize /dev/vg/test 1000M
```

5.18.2 ext4 外部日志设备

1. 日志设备丢失

移除不可用的日志,代码如下：

```
tune2fs -O ^has_journal /dev/ext4-device
```

检查修复文件系统,代码如下：

```
fsck/repair
```

建立内部日志,代码如下：

```
tune2fs -O has_journal /dev/ext4-device
```

2. 建立 ext4 外部日志

格式化日志设备,代码如下：

```
mke2fs -b 4096 -O journal_dev /dev/ext4-journal-device
```

建立一个新的文件系统,代码如下：

```
mkfs.ext4 -J device=/dev/ext4-journal-device /dev/ext4-device
```

添加给已存在的文件系统,代码如下：

```
tune2fs -O journal_dev -J device=/dev/ext4-journal-device /dev/ext4-device
```

外部日志设备大小，当日志太大时，会增长 crash 后文件系统检验 fsck 的时间。

文件系统的一般配置参数如下：

```
<32768 block logdev=1024 block 4k (<128M,4M)
<262144 block logdev=4096 block  (<1G,16M)
>262144 block logdev=8192 block  (>1G,32M )
```

3. fdtree：测试工具

问题 1：日志设备的永久性。当有多个硬盘设备时，Linux 是随机地指定名称的。不能确保/dev/sdb 重启后还会映射到同一设备。

除非加入一个自定义的 udev 规定。ext4 不理解 UUID，因此外部日志设备必须是一个持久性设备，而且重启后不会改变。

问题 2：当有外部日志设备时，默认日志挂载选项不支持，必须指定 journal_async_commit。不然操作一小时或有大量 IO 时很快就会变为只读。显然，外部日志不能使用同步更新，由于日志提交错误会提交或延后使 ext4 文件系统变为只读。

5.18.3　XFS 磁盘格式

XFS 格式化：块设备分割成 8 个或以上相等的线性区域(region 或块 chunk)称为分配组。分配组是唯一的，独立管理本身的 inode 节点和空闲空间(相似文件子系统，使用高效的 B+树来跟踪主要数据)，分配组机制给 XFS 提供了可伸缩和并行特性(多个线程和进程能够同时在同一个文件系统上执行 I/O 操作)。

XFS：数据段(数据，元数据)，日志段，实时段(默认在 mkfs.xfs 下：实时段不存在，日志段包含在数据段中)。

(1) 磁盘格式选项属性如下：

```
mkfs.xfs -b block_size(块大小) options
    -d data_section_options(数据属性)(sunit/swidth(单位为 512Byte)=su/sw 条带大小/宽度)
mkfs.xfs -d su=4k(条块 chunk 大小),sw=16(数据盘个数) /dev/sdb
mkfs.xfs -d sunit=128,swidth= sunit * 数据盘个数      /dev/sdd
 数据属性有
      agvount= value 指定分配组(并发小文件系统(16MB~1TB))
      agsize = value 与上条属性相似，指定分配组大小
      name= 指定文件系统内指定文件的名称。此时，日志段必须指定在内部(指定大小)
      file[=value] 指定上面要命名的是常规文件(默认为 1，能够为 0)
      size= value 指定数据段大小，需要 -d file =1
      sunit=value 指定条带单元大小(chunk，单位为 512)
      su=value 指定条带单元(chunk，单位为 Byte，如 64KB，必须为文件系统块大小的倍数)
      swidth=value 指定条带宽度(单位为 512，为 sunit 的数据盘个数的倍数)
      sw =value 条带宽度(一般为数据盘个数)
      noalign 忽略自动对齐(磁盘几何形状探测，文件不用几何对齐)
     -i inode_options 节点选项(xfs inode 包含两部分：固定部分和可变部分)
```

```
            这些选项会影响可变部分,包括目录数据、属性数据、符号链接数据,以及文件 extent 列
表和文件 extent 描述性根树
            选项有:
               size = value | log = value | perblock = value 指定 inode 大小(256～2048)
               maxpct = value 指定 inode 全部空间的百分比(默认为:< 1TB = 25 % ,< 50TB = 5 % >
50TB = 1 % )
               align [ = value] 指定分配 inode 时是否对齐。默认为 1,对齐
               attr = value 指定属性版本号,默认为 2
               projid32 位 [ = value] 是否使能 32 位配额项目标识符。默认为 1
            -f 强制(force)
            -l log_section_options (日志属性)(internal/logdev)
                  选项有
                     internal [ = value] 指定日志段是否作为数据段的一部分,默认为 1
                     logdev = device 指定日志位于一个独立的设备上(最小为 10MB,2560 个 4KB 块)
                      建立: mkfs.xfs -l logdev = /dev/ramhdb -f /dev/mapper/vggxxxxx
                      挂载: mount -o logdev = /dev/ramhdb /dev/mapper/vggxxxxx
                           size = value 指定日志段的大小
                           version = value 指定日志的版本,默认为 2
                           sunit = value 指定日志对齐,单位为 512
                           su = value 指定日志条带单元,单位为 Byte
                           lazy-count = value 是否延迟计数,默认为 1,更改超级块中各类连续
计数器的记录方法。在值为 1 时,不会在计数器每次变化时更新超级块
            -n naming_options 命名空间(目录参数)
                  选项有
                     size = value | log = value 块大小,不能小于文件系统 block,并且是 2 的幂
                     版本 2 默认为 4096(若是文件系统 block > 4096,则为 block)
                     version = value 命名空间的版本,默认为 2 或'ci'
                     ftype 和 value 容许 inode 类型存储在目录结构中,以便 readdir 和 getdents 不
需要查找 inode 就可知道 inode 类型。默认为 0,不存在目录结构中(使能 crc: -m crc = 1 时,此选
项会使能)
            -p protofile
            -r realtime_section_options (实时数据属性)(rtdev/size)
                实时段选项:
                     rtdev = device 指定外部实时设备名
                     extsize = value 指定实时段中块的大小,必须为文件系统块大小的倍数最小为
(max(文件系统块大小,4KB))。默认大小为条带宽度(条带卷),或 64KB(非条带卷),最大为 1GB
                     size = value 指定实时段的大小,noalign 此选项禁止条带大小探测,强制实时
设备没有几何条带
            -s sector size(扇区大小),最小为 512,最大为 32768 (32KB),不能大于文件系统块大小
            -L label 指定文件系统标签,最多为 12 个字符
            -q(quiet 不打印) -f(Force 强制)
            -N 只打印信息,不执行实际的建立
```

(2) 元数据日志能够独立存放 XFS 外部日志设备,命令如下:

```
mkfs.xfs -l logdev = /dev/sdb1,size = 10000b /dev/sda1
```

日志大小为 10 000block,存放在 sdb1 上。

外部日志设备的大小:与事务 transaction 的速率和大小相关,与文件系统的大小无关。大的 block size 会致使大的 transaction,日志事务 transaction 来源于目录更新(建立/删除/修改),如 mkdir、rmdir、create()、unlink()系统调用会产生日志数据。最小日志大小为最大

的 transaction 大小(取决于文件系统和目录块大小),最小为 10MB。目录块大小：mkfs.xfs-n；默认为 4KB,当文件系统 blocksize 大于 4KB 时,默认为 blocksize。提升大量小文件的性能,提升了目录查找的性能,由于树存储索引信息有较大的块和较小的深度,所以最大日志大小为 64k 个 blocks 和 128MB 中的最小值。日志太大会增长 crash 后文件系统的 mount 时间。

(3) mount-o 选项的用法如下：

```
allocsize= 延时分配时,预分配 buffered 的大小
    sunit= /swidth= 使用指定的条带单元与宽度(单位为 512Byte),优先级高于 mkfs 时指定的
barrier write barrier
    swalloc 根据条带宽度的边界调整数据分配
discard 块设备自动回收空间
    dmapi 使能 Data Management API 事件
mtpt=mountpoint
    inode64 建立 inode 节点位置不受限制
inode32inode 节点号不超过 32 位(为了兼容)
    largeio 大块分配,先 swidth,后 allocsize
nolargeio 尽可能小块分配
    noalign 数据分配时不用条带大小对齐
noatime 读取文件时不更新访问时间
    norecovery 挂载时不运行日志恢复(只读挂载)
logbufs= 在内存中的日志缓存区数量
     logbsize=内存中每个日志缓存区的大小
logdev=/rtdev=指定日志设备或实时设备,XFS 文件系统能够分为 3 部分:数据、日志和实时(可选)

sysctls:/proc/sys/fs/xfs/
          stats_clear: (Min: 0 Default: 0 Max: 1) 清除状态信息(/proc/fs/sys/xfs/stat)
          xfssyncd_centisecs:(Min: 100 Default: 3000 Max: 720000)xfssyncd 刷新元数据时间间
隔(写到磁盘,默认为 30s)
          xfsbufd_centisecs:(Min: 50 Default: 100Max: 3000)xfsbufd 扫描脏 buffer 的时间间隔
          age_buffer_centisecs:(Min: 100 Default: 1500 Max: 720000)xfsbufd 刷新脏 buffer 到
磁盘的时间
          irix_symlink_mode:(Min: 0 Default: 0 Max: 1)控制符号连接的模式是否是 0777
          inherit_nosymlinks:(Min: 0 Default: 1 Max: 1)xfs_io 下 chattr 命令设置 nosymlinks
标志
          inherit_sync: (Min: 0 Default: 1 Max: 1)xfs_io 下 chattr 命令设置 sync 标志
          inherit_nodump: (Min: 0 Default: 1 Max: 1)xfs_io 下 chattr 命令设置 nodump 标志
          inherit_noatime:(Min: 0 Default: 1 Max: 1)xfs_io 下 chattr 命令设置 noatime 标志
          rotorstep: (Min: 1 Default: 1 Max: 256)inode32 模式下
          error_level: (Min: 0 Default: 3 Max: 11)文件系统出错时会显示详细信息
                    XFS_ERRLEVEL_OFF:0
                    XFS_ERRLEVEL_LOW:1
                    XFS_ERRLEVEL_HIGH:5
          panic_mask:(Min: 0 Default: 0 Max: 127)遇到指定的错误时调用 Bug()(调试时用)
                    XFS_NO_PTAG                 0
                    XFS_PTAG_IFLUSH             0x00000001
                    XFS_PTAG_LOGRES             0x00000002
                    XFS_PTAG_AILDELETE          0x00000004
                    XFS_PTAG_ERROR_REPORT       0x00000008
```

```
XFS_PTAG_SHUTDOWN_CORRUPT          0x00000010
XFS_PTAG_SHUTDOWN_IOERROR          0x00000020
XFS_PTAG_SHUTDOWN_LOGERROR         0x00000040
```

5.18.4 XFS 工具

XFS 工具的命令如下：

```
mkfs.xfs：建立 XFS 文件系统
    xfs_admin：调整 XFS 文件系统的各类参数
    xfs_copy：将 XFS 文件系统的内容复制到一个或多个目标系统(并行方式)
    xfs_db：调试或检测 XFS 文件系统(查看文件系统碎片 xfs_db -c frs -r /dev/sdh 等)
    xfs_check：检测 XFS 文件系统的完整性
    xfs_bmap：查看一个文件的块映射 -->xfs_io -r -p xfs_bmap -c bmap "OPT" file
    xfs_repair：尝试修复受损的 XFS 文件系统 xfs_repair -n 仅报告问题,不修复
    xfs_fsr：碎片整理(xfs_fsr /dev/sdh)
    xfs_quota：管理 XFS 文件系统的磁盘配额
    xfs_metadump：将 XFS 文件系统的元数据 (metadata) 复制到一个文件中
    xfs_mdrestore：从一个文件中将元数据恢复到 XFS 文件系统
    xfsdump：增量备份 XFS 文件系统
    xfsrestore：恢复 XFS 文件系统
    xfs_growfs：调整一个 XFS 文件系统的大小(只能扩展)
    xfs_freeze：暂停(-f)和恢复(-u)XFS 文件系统
    xfs_info：查询 XFS 文件系统信息
    xfs_estimate：评估 XFS 文件系统的空间
    xfs_repair：修复 XFS 文件系统
    xfs_mkfile：建立 XFS 文件系统
    xfs_rtcp：XFS 实时复制命令
    xfs_ncheck：从 i 节点号生成路径
    xfs_io：调试 XFS I/O 路径
    xfs_logprint：打印 XFS 文件系统日志
```

检查文件系统：先确保 umount,命令如下：

```
xfs_check /dev/sdd(盘符); echo $?
```

如果返回 0,则表示正常。

修复文件系统,命令如下：

```
xfs_repair /dev/sdd (ext 系列工具为 fsck)
```

根据打印消息,修复失败时先执行 xfs_repair-L /dev/sdd(清空日志,会丢失文件),再执行 xfs_repair /dev/sdd,然后执行 xfs_check /dev/sdd 检查文件系统是否修复成功。

增大 XFS 文件系统：先用 lvextend 扩大 XFS 所在的 LUN,示例命令如下：

```
lvextend -L +5G /dev/mapper/lun5
xfs_growfs /demo (lun5 在扩大以前已经被格式化为 XFS 并挂载在/demo 下 )
df -h #查看文件系统的变化
```

在由 5 个盘组成的 raid5 下建立 lun；chunk＝64k；此时格式化命令如下：

```
mkfs.ext4 -E stride=16(64K/4k block) lun 设备
mount -o stripe=64 (16*4 个数据盘)

mkfs.xfs -d sunit=128 (64KB/扇区) swidth=512 (128*4 个数据盘)
mount -o sunit= swidth=
```

5.18.5　项目实践

以 CentOS 系统为例，需要将 U 盘挂载到系统进行文件操作，需要依赖于一些库，支持 exfat 格式的 U 盘需要安装 epel 库和 Nux Dextop 库，再安装 fuse-exfat 和 exfat-utils 包，即可识别 exfat 格式。

Nux Dextop 是类似 CentOS、RHEL、ScientificLinux 的第三方 RPM 仓库，例如 Ardour、Shutter 等。目前，Nux Dextop 对 CentOS/RHEL 6|7 可用。Nux Dextop 库依赖于 EPEL 库，所有要先安装 EPEL 库(需要管理员权限)。

安装 EPEL 库，命令如下：

```
yum -y install epel-release
```

对于 RHEL 6/CentOS 6，命令如下：

```
rpm -Uvh http://li.nux.ro/download/nux/dextop/el6/x86_64/nux-dextop-release-0-2.el6.
nux.noarch.rpm
```

对于 RHEL/CentOS 7，命令如下：

```
rpm -Uvh http://li.nux.ro/download/nux/dextop/el7/x86_64/nux-dextop-release-0-5.el7.
nux.noarch.rpm
```

检查 Nux Dextop 是否安装成功，命令如下：

```
yum repolist
```

如果仓库列表中有 Nux Dextop，则表示安装成功。

由于 Nux Dextop 仓库可能会与其他第三方库有冲突，例如(Repoforge 和 ATrpms)，所以建议在默认情况下不启用 Nux Dextop 仓库。

打开/etc/yum.repos.d/nux-dextop.repo，将 enabled＝1 修改为 enabled＝0：

```
sed -i 's/enabled=1/enabled=0/' /etc/yum.repos.d/nux-dextop.repo
```

安装 exfat 支持库文件，命令如下：

```
yum --enablerepo=nux-dextop install fuse-exfat exfat-utils
```

安装完成以后，将 exfat 格式的 U 盘挂载到系统，就可以识别对应的磁盘格式，然后便可以进行文件的读写操作。

5.19 离线打包外部应用依赖

项目依赖于很多第三方库或系统软件，如果要打包部署，则需要把相关的系统依赖包下载整理，这里以 CentOS 系统作为运行环境为例，举例说明如何下载相关的系统依赖包及关联包，以及下载之后的离线安装系统包的方法。

yum-downloadonly 用于下载所需要的软件包而并不真正地安装，下载好的软件包方便在没有网络的情况下使用。

方法一：downloadonly 插件。

有一个 yum 的插件叫作 downloadonly，顾名思义，就是只下载而不安装的意思。

(1) 安装插件，命令如下：

```
yum install yum - download
```

(2) 下载依赖包，以 httpd 为例，命令如下：

```
yum update httpd - y - downloadonly
```

这样 httpd 的 RPM 就被下载到/var/cache/yum/中去了。

也可以指定一个目录存放下载的文件，命令如下：

```
yum update httpd - y - downloadonly - downloaddir = /opt
```

值得注意的是，downloadonly 插件不但适用于 yum update，也适用于 yum install。

推荐方法二，有些系统版本用方法一安装不了。

方法二：yum-utils 中的 yumdownloader。

yum-utils 包含着一系列的 yum 的工具，例如 Debuginfo-install、package-cleanup、repoclosure、repodiff、repo-graph、repomanage、repoquery、repo-rss、reposync、repotrack、verifytree、yum-builddep、yum-complete-transaction、yumdownloader、yum-Debug-dump 和 yum-groups-manager。

(1) 安装 yum-utils，命令如下：

```
yum - y install yum - utils
```

(2) 使用 yumdownloader 下载依赖包，以 httpd 为例，命令如下：

```
yumdownloader httpd
```

方法三：利用 yum 的缓存功能。

用 yum 安装了某个工具后，想要这个工具的包，在 yum 安装的过程中其实已经把包下载了，只是没有保持而已，所以需要做的是将其缓存功能打开。

(1) 编辑 yum 配置文件，命令如下：

```
vi /etc/yum.conf
```

将其中的 keepcache=0 改为 keepcache=1，保存后退出。

(2) 重启对应的服务，命令如下：

```
/etc/init.d/yum-updatesd restart
```

(3) 安装 httpd 服务，命令如下：

```
yum install httpd
```

(4) 查看缓存文件保存的位置，命令如下：

```
cat /etc/yum.conf |grep cachedir
cachedir = /var/cache/yum
```

(5) 进入上述目录，命令如下：

```
cd cachedir = /var/cache/yum && tree ./
```

(6) 这时的目录树中应该可以找到需要的安装包了，命令如下：

```
ls -lh
```

查看 cat /etc/yum/pluginconf.d/downloadonly.conf，确保插件已被启用，代码如下：

```
[main]

enabled = 1
```

例如下载 Apache 软件包，并放在“/”下，命令如下：

```
yum install httpd -y --downloadonly --downloaddir = / usr/src
```

使用 rpm 命令进行安装，命令如下：

```
rpm -Uvh --force --nodeps /opt/soft/*.rpm
yum localinstall *.rpm -y
```

方法四：reposync 工具。

通过 yum 命令的 reposync 命令下载某个 repo 源的所有 RPM 软件包，命令如下：

```
reposync -r repo 源的名称 + -p + 指定下载的路径(可选)
```

默认会将软件包下载到当前目录下(自动生成 repo 源的目录及 Packages),命令如下:

```
mkdir repo_test

cd repo_rest

reposync -r base
```

也可以通过-p 来指定位置,软件包将被下载到此目录,命令如下:

```
reposync -r base -p root/repo2/
```

查看依赖包可以使用 yum deplist 命令进行查找,命令如下:

```
yum deplist nginx
```

使用 repotrack 命令下载所需依赖,命令如下:

```
yum -y install yum-utils
repotrack nginx
```

使用 yumdownloader 命令下载软件依赖,命令如下:

```
yumdownloader --resolve --destdir=/tmp ansible
```

只会下载当前系统环境下所需的依赖包。

使用 yum 自带的 downloadonly 插件下载 nginx,示例命令如下:

```
yum -y install nginx --downloadonly --
downloaddir=/tmp/
```

5.20 多功能的定时任务使用

在项目中,因为日志文件越累积越多,所以需要定期清理,并且节点状态和链路状态需要进行监测,流量也需要定时统计等。这就需要用到定时任务去完成,定时任务有很多实现方案,这里选用 goCron 框架。

goCron 是一个 Go 作业调度工具,可以使用简单的语法定期执行 go 函数。

goCron 支持 Cron 表达式,示例如下:

```
"0 0 0 * * *" 每天 0 点启动 * 通配符可以匹配任何数字

"*/5 * * * * *" 表示每隔 5s 执行一次

"*/1 * * * *" 表示每隔 1min 执行一次,比秒级别解析器少了一个 *
```

```
"30 * * * *" 分钟域为 30,其他域都用 * 表示任意。每 30 分触发

"30 3-6,20-23 * * *":分钟域为 30,小时域的 3-6 和 20-23 分别表示 3 点到 6 点和 20 点到
23 点。每小时的 30 分钟触发

"0 0 0 * * ?" 表示每天 0 点执行一次

"0 0 1 1 * ?" 表示每月 1 号凌晨 1 点执行一次

"0 1,2,3 * * * ?" 表示在 1 分、2 分和 3 分各执行一次

"0 0 0,1,2 * * ?" 表示每天的 0 点、1 点和 2 点各执行一次
```

1. 简单使用

首先,初始化 s 对象,然后直接配置定时任务,任务添加函数名 + 参数;最后,block 当前进程,观察任务执行情况,代码如下:

```
package main

import (
    "fmt"
    "time"

    "github.com/go-co-op/gocron"
)

func task(s string){
    fmt.Printf("I'm running, about %s. \n", s)
}

func main() {
    s := gocron.NewScheduler(time.UTC)
    s.Every(1).Seconds().Do(task, "1s")
    s.StartBlocking()
}
```

输出如下:

```
I'm running, about 1s.
I'm running, about 1s.
I'm running, about 1s.
```

2. 更多参考设置

针对定时任务,可以设置时、分、秒、天、周,也可以采用 crontab 字符串的格式进行设置,代码如下:

```
package main

import (
    "fmt"
    "time"

    "github.com/go-co-op/gocron"
)

func task(){
    fmt.Printf("I'm running.\n")
}

func main() {
    s := gocron.NewScheduler(time.UTC)

    //每隔多久
    s.Every(1).Seconds().Do(task)
    s.Every(1).Minutes().Do(task)
    s.Every(1).Hours().Do(task)
    s.Every(1).Days().Do(task)
    s.Every(1).Weeks().Do(task)

    //每周几
    s.Every(1).Monday().Do(task)
    s.Every(1).Thursday().Do(task)

    //每天固定时间
    s.Every(1).Days().At("10:30").Do(task)
    s.Every(1).Monday().At("18:30").Do(task)
    s.Every(1).Tuesday().At("18:30:59").Do(task)

    //设置 crontab 字符串格式
    s.Cron("*/1 * * * *").Do(task)

    s.StartBlocking()
}
```

3. 项目实践

在实际项目中，以检测节点部署状态和检测链路部署状态为例，代码如下：

```
//anonymous-link\code\task\periodic_tasks.go
package task

import (
    "encoding/json"
    "errors"
    "fmt"
    "github.com/jasonlvhit/gocron"
```

```
    "github.com/jinzhu/gorm"
    "manage/config"
    "manage/model"
    "manage/service"
    "manage/utils"
    "strings"
    "time"
)

func PeriodicTasks(logCycle string) {
    //可并发运行多个任务
    //gocron.Every(2).Minutes().Do(CheckNodeRunning)
    //gocron.Every(2).Minutes().Do(CheckLinkRunning)
    gocron.Every(3).Minute().Do(CheckNodeDeploying)
    gocron.Every(3).Minute().Do(CheckLinkDeploying)
    gocron.Every(15).Seconds().Do(CheckLinkStatus)
    gocron.Every(15).Seconds().Do(DelDefaultRoute)
    gocron.Every(15).Seconds().Do(AddWhitelistRoute)
    gocron.Every(3).Minutes().Do(SyncDeviceInfo)
    gocron.Every(1).Day().Do(ClearLogData, logCycle)
    gocron.Every(1).Minute().Do(ComputeLinkFlow)
    gocron.Every(3).Minutes().Do(CheckDeleteFailedLink)
    //gocron.Every(2).Minutes().Do(SyncDeleteNic)
    gocron.Every(1).Minute().Do(ComputeTerminalFlow)
    gocron.Every(1).Minute().Do(CheckTerminalConnect)

    <- gocron.Start()
}

func CheckNodeDeploying() {
    where := fmt.Sprintf("status = %d", model.NodeStatus().Deploying)
    nodeList, _, err := model.SearchNode(config.DefaultMaxLimit, 0, where, "")
    if err != nil {
        return
    }
    for _, node := range nodeList {
        if utils.SubMinuteByTime(node.UpdatedAt) > 20 {
            w := fmt.Sprintf("id = %d", node.Id)
            uds := map[string]interface{}{"status": model.NodeStatus().DeployFailed}
            err = model.UpdateNodeByWhere(w, uds)
            if len(node.PreNodes) > 1 {
                var ids []string
                json.Unmarshal([]Byte(node.PreNodes), &ids)
                if len(ids) > 0 {
                    w = fmt.Sprintf("id in ('%s')", strings.Join(ids, "','"))
                    nodes, count, err := model.SearchNode(0, 0, w, "")
                    if err == nil && count > 0 {
                        service.DeployFailedRecover(nodes)
                    }
                }
            }
```

```
                if err == nil {
                    SendMessage(node, config.MessageNodeUpdate)
                }
            }
        }
    }

    func CheckLinkDeploying() {
        where := fmt.Sprintf("status = %d", model.LinkStatus().Deploying)
        linkList, _, err := model.SearchLink(config.DefaultMaxLimit, 0, where, "")
        if err != nil {
            return
        }
        for _, link := range linkList {
            if utils.SubMinuteByTime(link.UpdatedAt) > 20 {
                w := fmt.Sprintf("id = %d", link.Id)
                uds := map[string]interface{}{"status": model.LinkStatus().DeployFailed}
                err = model.UpdateLinkByWhere(w, uds)
                if len(link.NodeInfo) > 1 {
                    var ids []string
                    json.Unmarshal([]Byte(link.NodeInfo), &ids)
                    service.StopNodes(ids)
                }
                if err == nil {
                    SendMessage(link, config.MessageLinkUpdate)
                }
            }
        }
    }
```

5.21 全自动智能化的测试框架

自动化测试主要基于 goconvey+httptest 进行单元和 API 测试，测试文件的编写规则如下：

(1) 测试用例文件名必须以_test.go 结尾，如 person_test.go。

(2) 测试用例函数必须以 Test 开头，例如 person_test.go 文件中的 TestReStore。

(3) 测试函数的形参必须是 t *testing.T。

(4) 在一个测试用例文件中，可以有多个测试用例函数。

如果要测试单个文件，则一定要带上被测试的源文件，命令如下：

```
go test -v person_test.go cal.go
```

测试单种方法，命令如下：

```
go test -v -test.run TestReStore
```

测试命令进入对应目录，输入 go test-v，命令如下：

```
go get github.com/smartystreets/goconvey
go get github.com/smartystreets/goconvey/convey@v1.7.2
go test -v
```

例如，对账号模块进行增、删、改、查测试，如图 5-39 所示，编写相关测试代码。

图 5-39　自动化测试用例

测试结果如图 5-40 所示。

覆盖率及覆盖程度，Web 页面如图 5-41 所示。

除了可以看到整个项目的总体测试覆盖率，还可以看到每个文件的测试覆盖率，如图 5-42 所示。

图 5-40 自动化测试结果

图 5-41 自动化测试覆盖代码

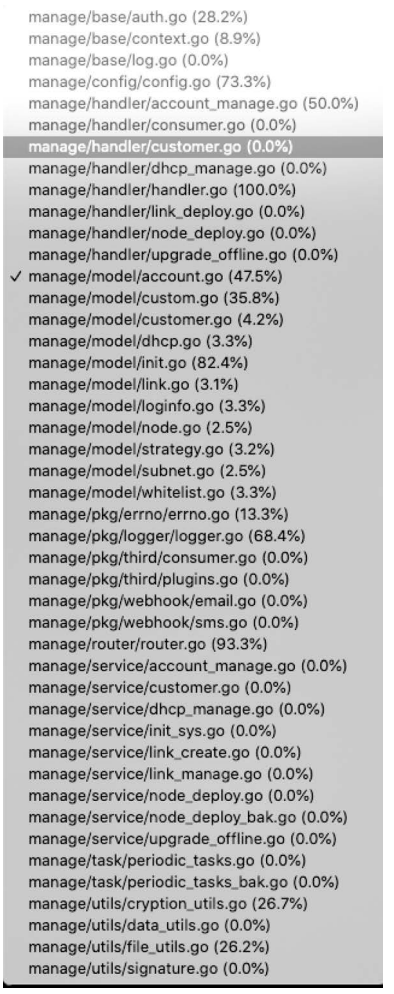


图 5-42　自动化测试覆盖率

5.22　完整项目的构建及介绍

项目的完整源码可访问网址 https://gitee.com/book-info/anonymous-link，整个项目的目录结构如图 5-43 所示。

主要文件的功能和目录的作用如下：

ansible 目录主要用于存放远程自动化部署的一些配置信息，还有远程自动化执行的一些封装的脚本模块。

base 目录主要用于存放项目的一些基础和抽象信息，例如权限检测模块、日志拦截模块、参数解析模块、自动化 API 文档信息等。

cert 目录主要用于存放每个节点的连接凭证，按照每个节点生成一个 UUID 文件夹的

图 5-43 项目开发目录结构

规则，方便在组件链路时进行使用。

conf 目录主要用于存放项目在不同运行环境中的配置信息，例如开发环境配置、测试环境配置、生产环境配置等，还有项目用到的文档。

config 目录主要用于对常量、消息及相应模板进行配置，以及错误码定义、通用数据结构组装和抽象等。

docker 目录主要用于存放 docker file 文件（自动化环境打包）、docker compose 自动化部署文件、k8s 自动化部署文件等。

docs 目录主要用于存放 Swagger 生产的 API 开发对接文档。

handler 目录主要用于存放接口逻辑处理模块，用于接受前端传入的参数，调用 service 进行逻辑处理，并返回相关数据。

license 目录主要用于存放软件使用证书，为每个硬件设备分配一个证书，防止软件被乱用或者盗用等。

mock 目录主要用于存放为自动化测试而不方便直接执行的方法，通过 mock 相应的方法可以让相关的测试流程和逻辑执行完成。

model 目录主要用于存放实体关系模型，以及相关的模型和数据库操作方法。

report 目录主要用于存放自动化测试报告，方便用户阅读和查看，提升代码测试覆盖度和项目软件的质量。

router 目录主要用于存放路由文件，结合 handler 定义接口访问路径。

rpm 目录主要用于存放程序依赖的第三方库或包，方便程序在启动时自动进行安装。

scheme 目录主要用于存放关系模型和数据库初始化创建的数据的 SQL 文件，以及后期生产环境中需要升级或者修改关系模型和数据库中的字段的 SQL 文件。

service 目录主要用于存放业务的逻辑处理模块，调用 model 相关方法对数据进行读写。

task 目录主要用于存放周期任务和定时任务，方便任务在入口处以协程方式进行调用启动。

test 目录主要用于存放 API 及相关模块方法的测试用例，需要注意的是文件名和方法名的定义有严格的命名规范。

utils 目录主要用于存放通用的工具类，把业务无关的方法抽象为公共方法，方便在其

他地方调用。

.gitignore 文件主要用于定义哪些文件或者目录忽略 git 的管理，例如有些文件不管是否被修改都不需要 git 进行跟踪或者提交上传。

check.sh 文件主要用于检测 git 管理中的哪一个分支有代码改动，方便后面的 CI/CD 流程执行相关的任务。

comment.sh 文件主要根据关系模型中的字段和注释自动生成相关字段的解释，并封装为 API，方便前后端进行对接联调。

go.mod 文件主要用于记录项目依赖的第三方库，方便进行版本和依赖管理。

Jenkinsfile 文件主要用于配置 CI/CD 自动化测试流程，方便代码提交后，借助于 Jenkins 的流水线功能，自动进行程序的环境依赖打包、自动化测试、自动化部署到不同的环境等。

main.go 文件是程序的启动入口，包含外部依赖库的安装、异步任务的启动、不同阶段的 API 启动服务等。

mock.sh 文件主要根据项目需要和开发方便，自定义了一套自动化对声明的方法进行 mock，方便自动化测试的执行及在非正式环境或没有相关的依赖服务的环境中运行，与之相对的是 recover.sh 文件，用于恢复被 mock 的方法。

package.sh 文件主要用于自动化打包脚本，自动对项目进行编译打包，并把运行程序放到相关的目录，需要注意的是，如果程序最终在 Linux 环境中运行，则需要把整个项目复制到 Linux 环境的/usr/local/目录中，并把项目文件夹名称重命名为 manage 执行打包操作，这样才能在最终的 Linux 路由设备上运行，主要原因是在后面的自动化安装系统及程序部署中使用了前面的绝对路径，这个路径可以修改，但是要全部修改。在 macOS 或者 Windows 系统上打包的程序无法在 Linux 系统上运行，反之亦然。

README.MD 文件主要用于介绍项目的一些基本情况，以及相关的使用说明等。

recover.sh 文件主要用于恢复通过被自定义方案 mock 的方法，保证程序的代码不被修改。

start.sh 文件主要用于非生产环境，例如开发环境，能够自动化识别服务器的 IP 地址，自动生成和更换 Swagger 文档中的 IP 地址和端口，方便前后端对接时查看 API 文档和进行在线接口测试。

swag.sh 文件主要用于非生产环境，以及用于没有网络或者没有完整安装 Swagger 相关依赖库的环境，可以基于存在的 API 文件修改 API 文档和进行在线接口测试。

test.sh 文件主要用于本地进行快速自动化测试，并生成相关的测试报告。

web.log 文件主要用于存放项目启动和运行的错误日志，方便排查和定位问题。

5.23　自定义封装服务和自启

程序开发完成后需要进行打包编译，然后放到硬件盒子路由器设备上运行，还需要把程序设置为自启动及服务自动恢复。目前主流实现方案有两种，一种是通过脚本写入 Linux

自启动文件目录；另一种是通过 system 服务将程序设置为自启动。两种方案各有优势，这里采用通过 system 服务的方式进行自启动，主要实现流程如下。

1. 打包编译程序

Go 开发的项目可以在 Linux 系统中进行编译打包成二进制文件，这种二进制文件不需要任何环境依赖便可以在绝大多数 Linux 系统环境中运行。编译打包命令如下：

```
go build -a -installsuffix cgo -o app .
```

2. 创建 service 服务

在 Linux 系统可以创建 system 管理的 service 服务，以 CentOS 系统为例，在/usr/lib/systemd/system/目录创建 manage.service 服务，内容如下：

```
#anonymous-link\iso\ventoy\source\init\manage.sh

[Unit]
Description=Manage Server Daemon
Wants=network-online.target
After=network-online.target

[Service]
Type=idle
PrivateTmp=true
WorkingDirectory=/usr/local/manage/
ExecStart=/usr/local/manage/app -c conf/config.yaml>/var/log/anonylink.log 2>&1
DeviceAllow=/dev/null rw
DeviceAllow=/dev/net/tun rw
KillMode=process
Restart=always
RestartSec=20
StartLimitInterval=0
#RestartSec=20s
#Restart=on-failure

[Install]
WantedBy=multi-user.target
```

3. 配置服务自启动

以 CentOS 系统为例，启动服务及设置服务自启动，命令如下：

```
systemctl start manage.service
systemctl enable manage.service
```