

Go语言四十二章经

书栈(BookStack.CN)

目 录

致谢

SUMMARY

前言

第一章 Go安装与运行

第二章 数据类型

第三章 变量

第四章 常量

第五章 作用域

第六章 约定和惯例

第七章 代码结构化

第八章 Go项目开发与编译

第九章 运算符

第十章 string

第十一章 数组(Array)

第十二章 切片(slice)

第十三章 字典(Map)

第十四章 流程控制

第十五章 错误处理

第十六章 函数

第十七章 Type关键字

第十八章 Struct 结构体

第十九章 接口

第二十章 方法

第二十一章 协程(goroutine)

第二十二章 通道(channel)

第二十三章 同步与锁

第二十四章 指针和内存

第二十五章 面向对象

第二十六章 测试

第二十七章 反射(reflect)

第二十八章 unsafe包

第二十九章 排序(sort)

第三十章 OS包

第三十一章 文件操作与IO

第三十二章 fmt包

第三十三章 Socket网络

第三十四章 命令行 flag 包
第三十五章 模板
第三十六章 net/http包
第三十七章 context包
第三十八章 Json数据格式
第三十九章 Mysql数据库
第四十章 LevelDB与BoltDB
第四十一章 网络爬虫
第四十二章 WEB框架(Gin)

致谢

当前文档《Go语言四十二章经》由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2018-10-16。

书栈(BookStack.CN) 仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈(BookStack.CN) 难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到 书栈(BookStack.CN) ，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到 书栈(BookStack.CN) 获取最新的文档，以跟上知识更新换代的步伐。

文档地址：<http://www.bookstack.cn/books/go42>

书栈官网：<http://www.bookstack.cn>

书栈开源：<https://github.com/TruthHun>

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

SUMMARY

- [前言](#)
- [第一章 Go安装与运行](#)
 - 1.1 Go安装
 - 1.2 Go 语言开发工具
- [第二章 数据类型](#)
 - 2.1 基本数据类型
 - 2.2 Unicode (UTF-8)
 - 2.3 复数
- [第三章 变量](#)
 - 3.1 变量以及声明
 - 3.2 零值nil
- [第四章 常量](#)
 - 4.1 常量以及iota
- [第五章 作用域](#)
 - 5.1 作用域
- [第六章 约定和惯例](#)
 - 6.1 可见性规则
 - 6.2 命名规范以及语法惯例
- [第七章 代码结构化](#)
 - 7.1 包的概念
 - 7.2 包的导入
 - 7.3 标准库
 - 7.4 从 GitHub 安装包
 - 7.5 导入外部安装包

- 7.6 包的分级声明和初始化

• 第八章 Go项目开发与编译

- 8.1 项目结构
- 8.2 使用 Godoc
- 8.3 Go程序的编译

• 第九章 运算符

- 9.1 内置运算符
- 9.2 运算符优先级
- 9.3 几个特殊运算符

• 第十章 string

- 10.1 有关string
- 10.2 字符串拼接

• 第十一章 数组(Array)

- 11.1 数组(Array)

• 第十二章 切片(slice)

- 12.1 切片(slice)
- 12.2 切片重组(reslice)
- 12.3 陈旧的(Stale)Slices

• 第十三章 字典(Map)

- 13.1 字典(Map)
- 13.2 “range”语句中更新引用元素的值

• 第十四章 流程控制

- 14.1 Switch 语句
- 14.2 Select控制
- 14.3 For循环
- 14.4 for-range 结构

• 第十五章 错误处理

- 15.1 错误类型
- 15.2 Panic
- 15.3 Recover: 从 panic 中恢复
- 15.4 有关于defer

• 第十六章 函数

- 16.1 函数分类
- 16.2 函数调用
- 16.3 内置函数
- 16.4 递归与回调
- 16.5 匿名函数
- 16.6 闭包函数
- 16.7 使用闭包调试
- 16.8 高阶函数

• 第十七章 Type关键字

- 17.1 Type

• 第十八章 Struct 结构体

- 18.1 结构体(struct)
- 18.2 结构体特性
- 18.3 匿名成员
- 18.4 内嵌(embedded)结构体
- 18.5 命名冲突

• 第十九章 接口

- 19.1 接口是什么
- 19.2 接口嵌套
- 19.3 类型断言
- 19.4 接口与动态类型
- 19.5 接口的提取
- 19.6 接口的继承

• 第二十章 方法

- 20.1 方法的定义
- 20.2 函数和方法的区别

- 20.3 指针或值方法
- 20.4 内嵌类型的方法提升

• 第二十一章 协程(goroutine)

- 21.1 并发
- 21.2 goroutine

• 第二十二章 通道(channel)

- 22.1 通道(channel)

• 第二十三章 同步与锁

- 23.1 同步锁
- 23.2 读写锁
- 23.3 sync.WaitGroup
- 23.4 sync.Once
- 23.5 sync.Map

• 第二十四章 指针和内存

- 24.1 指针
- 24.2 new() 和 make() 的区别
- 24.3 垃圾回收和 SetFinalizer

• 第二十五章 面向对象

- 25.1 Go 中的面向对象
- 25.2 多重继承

• 第二十六章 测试

- 26.1 单元测试
- 26.2 基准测试
- 26.3 分析并优化 Go 程序
- 26.4 用 pprof 调试

• 第二十七章 反射(reflect)

- 27.1 反射(reflect)
- 27.2 反射结构体

- [第二十八章 unsafe包](#)
 - 28.1 unsafe 包
 - 28.2 指针运算
- [第二十九章 排序\(sort\)](#)
 - 29.1 sort包介绍
 - 29.2 自定义sort.Interface排序
 - 29.3 sort.Slice
- [第三十章 OS包](#)
 - 30.1 启动外部命令和程序
 - 30.2 os/signal 信号处理
- [第三十一章 文件操作与IO](#)
 - 31.1 文件系统
 - 31.2 IO读写
 - 31.3 ioutil包
 - 31.4 bufio包
- [第三十二章 fmt包](#)
 - 32.1 fmt包格式化I/O
 - 32.2 格式化verb应用
- [第三十三章 Socket网络](#)
 - 33.1 Socket基础知识
 - 33.2 TCP 与 UDP
- [第三十四章 命令行 flag 包](#)
 - 34.1 命令行
 - 34.2 flag包
- [第三十五章 模板](#)
 - 35.1 text/template
 - 35.2 html/template

- 35.3 模板语法

• 第三十六章 net/http包

- 36.1 Request
- 36.2 Response
- 36.3 client
- 36.4 server
- 36.5 自定义处理器 (Custom Handlers)
- 36.6 将函数作为处理器
- 36.7 中间件Middleware
- 36.8 静态站点

• 第三十七章 context包

- 37.1 context包
- 37.2 context应用

• 第三十八章 Json数据格式

- 38.1 序列化与反序列化
- 38.2 Json格式处理
- 38.3 XML 数据格式
- 38.4 用 Gob 传输数据

• 第三十九章 Mysql数据库

- 39.1 database/sql包
- 39.2 Mysql数据库操作

• 第四十章 LevelDB与BoltDB

- 40.1 LevelDB
- 40.2 BoltDB

• 第四十一章 网络爬虫

- 41.1 go-colly
- 41.2 goquery

• 第四十二章 WEB框架(Gin)

- 42.1 有关于Gin
- 42.2 Gin实际应用

前言

《Go语言四十二章经》

作者：ffhelicopter（李骁） 时间：2018-04-15

起因

一直想写点什么但懒得动笔或者是不知写什么。而这次写《Go语言四十二章经》，纯粹是因为开发过程中碰到过的一些问题，踩到过的一些坑，感觉在Go语言学习使用过程中，有必要深刻理解这门语言的核心思维、清晰掌握语言的细节规范以及反复琢磨标准包代码设计模式，于是才有了这本书。

Go语言以语法简单、门槛低、上手快著称。但入门后很多人发现要写出地道的、遵循 Go语言思维的代码却是不易。

在刚开始学习中，我带着比较强的面向对象编程思维惯性来写代码。但后来发现，带着面向对象的思路来写Go 语言代码会很难继续写下去，或者说看了系统源代码或其他知名开源包源代码后，围绕着Struct和Interface来写代码会更高效，代码更美观。虽然有人认为，Go语言的Strcut 和 Interface 一起，配合方法，也可以理解为面向对象，这点我姑且认可，但开发中不要过意考虑这些。因为在Go 语言中，Interface接口的使用将更为灵活，刻意追求面向对象，会导致你很难理解接口在Go 语言中的妙处。

作为Go语言的爱好者，在阅读系统源代码或其他知名开源包源代码时，发现大牛对这门语言的了解之深入，代码实现之巧妙优美，所以我建议你花时间多多阅读这些代码。网上有说Go大神的标准是“能理解简洁和可组合性哲学”，的确Go语言追求代码简洁到极致，而组合思想可谓借助于struct和 interface两者而成为Go的灵魂。

Function，Method，Interface，Type等名词是程序员们接触比较多的关键字，但在Go语言中，你会发现，其有了更强大，更灵活的用法。当你彻底理解了Go语言相关基本概念，以及对其特点有深入的认知，当然这也这本书的目的，再假以时日多练习和实践，我相信你应该很快就能彻底掌握这门语言，成为一名出色的Gopher。

这本书适合Go语言新手来细细阅读，对于有一定经验的开发人员，也可以根据自己的情况，选择一些章节来看。

第一章到第二十六主要讲Go语言的基础知识，其中第十七章的type，第十八章的struct，第十九章的interface，以及第二十章的方法，都是Go语言中非常非常重要的部分。

而第二十一章的协程，第二十二章的通道以及第二十三章的同步与锁，这三章在并发处理中我们通常都需要用到，需要弄清楚他们的概念和彼此间联系。

从二十七章开始，到第三十八章，讲述了Go标准包中比较重要的几个包，可以仔细看源代码来学习大师们的编程风格。

从第三十九章开始到结尾，主要讲述了比较常用的第三方包，但由于篇幅有限，也就不展开来讲述，有兴趣的朋友可直接到相关开源项目详细了解。

最后，希望更多的人了解和使用Go语言，也希望阅读本书的朋友们多多交流。虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。联系邮

箱：roteman@163.com

祝各位Gopher们工作开心，愉快编码！

阅读

本书内容在github更

新：<https://github.com/ffhelicopter/Go42/blob/master/SUMMARY.md>

本书内容在简书更新：<https://www.jianshu.com/nb/29056963>

交流

虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。

联系邮箱：roteman@163.com

感谢以下网友对本书提出的修改建议： Joyboo 、 林远鹏

更新

本书会持续更新！为了更简单表述清楚，某些章节的内容我会根据情况随时更新；当然也会随Go语言版本的不断更新，不断修改完善相关章节的内容和代码。

第一章 Go安装与运行

《Go语言四十二章经》第一章 Go安装与运行

作者：李骁

Go语言是一门全新的静态类型开发语言，具有自动垃圾回收，丰富的内置类型，函数多返回值，错误处理，匿名函数，并发编程，反射等特性，并具有简洁、安全、并行、开源等特性。

在接下来26章中，主要讲解Go语言的基本语法和特性。在掌握基础知识的前提下，需要对struct、interface、方法、通道以及锁和goroutine有彻底理解。

1.1 Go安装

Go语言支持以下系统：

- Linux
- FreeBSD
- Mac OS
- Windows

安装包下载地址为：<https://golang.org/dl/>

国内可以正常下载地址：<https://golang.google.cn/dl/>

UNIX/Linux/Mac OS X, 和FreeBSD系统下使用源码安装方法：

1、下载源码包：go1.11.linux-amd64.tar.gz。

2、将下载的源码包解压至 /usr/local目录。

```
tar -C /usr/local -xzf go1.11.linux-amd64.tar.gz
```

3、将 /usr/local/go/bin 目录添加至PATH环境变量：

```
export PATH=$PATH:/usr/local/go/bin
```

注意：MAC系统下你可以使用 .pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

Windows系统下安装

你可以选择本地安装目录：D:\Go，把D:\Go\bin目录添加到 PATH 环境变量中。

设置\$GOPATH=D:\goproject以及 \$GOROOT=D:\Go\ 。

打开CMD，输入 go version，如下显示说明go运行环境已经安装成功：

```
D:\goproject\src\ind\study>go version
go version go1.11 windows/amd64
```

在本书中，所有代码和标准库的讲解都基于`go1.11`，还没有升级的用户请及时升级。

`$GOPATH`允许多个目录，当有多个目录时，请注意分隔符，多个目录的时候Windows是分号；当有多个`$GOPATH`时默认将`go get`获取的包存放在第一个目录下。

`$GOPATH`目录约定有三个子目录

- `src`存放源代码(比如：`.go .c .h .s`等) 按照Go 默认约定，`go run`，`go install`等命令的当前工作路径（即在此路径下执行上述命令）。
- `pkg`编译时生成的中间文件（比如：`.a`）
- `bin`编译后生成的可执行文件，接下来就可以试试代码编译运行了。

文件名：`test.go`，代码如下：

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     fmt.Println("Hello, World!")
7. }
```

使用`go`命令执行以上代码输出结果如下：

```
D:\goproject>go run test.go
```

```
Hello, World!
```

1.2 Go语言开发工具

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

支持的操作系统：

Windows x86 (32-bit or 64-bit)

Linux x86 (32-bit or 64-bit)

下载地址：<http://sourceforge.net/projects/liteide/files/>

源码地址：<https://github.com/visualfc/liteide>

本书《Go语言四十二章经》内容在github上同步地址：<https://github.com/ffhelicopter/Go42>

本书《Go语言四十二章经》内容在简书同步地址：<https://www.jianshu.com/nb/29056963>

虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。

联系邮箱：roteman@163.com

第二章 数据类型

《Go语言四十二章经》第二章 数据类型

作者：李骁

在 Go 语言中，数据类型可用于参数和变量声明。

2.1 基本数据类型

Go 语言按类别有以下几种数据类型：

- 布尔型：

布尔型的值只可以是常量 `true` 或者 `false`。一个简单的例子：`var b bool = true`。

- 数字类型：

整型 `int` 和浮点型 `float32`、`float64`，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。

- 字符串类型：

字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go 语言的字符串的字节使用UTF-8编码标识Unicode文本。

- 派生类型：

包括：

1. (a) 指针类型 (`Pointer`)
2. (b) 数组类型
3. (c) 结构类型(`struct`)
4. (d) `Channel` 类型
5. (e) 函数类型
6. (f) 切片类型
7. (g) 接口类型 (`interface`)
8. (h) `Map` 类型

数字类型：

Go 也有基于架构的类型，例如：int、uint 和 uintptr，这些类型的长度都是根据运行程序所在的操作系统类型所决定的。

类型	符号	长度范围
uint8	无符号	8位整型 (0 到 255)
uint16	无符号	16位整型 (0 到 65535)
uint32	无符号	32位整型 (0 到 4294967295)
uint64	无符号	64位整型 (0 到 18446744073709551615)
int8	有符号	8位整型 (-128 到 127)
int16	有符号	16位整型 (-32768 到 32767)
int32	有符号	32位整型 (-2147483648 到 2147483647)
int64	有符号	64位整型 (-9223372036854775808 到 9223372036854775807)

浮点型：

主要是为了表示小数，也可细分为float32和float64两种。浮点数能够表示的范围可以从很小到很巨大，这个极限值范围可以在math包中获取，math.MaxFloat32表示float32的最大值，大约是3.4e38，math.MaxFloat64大约是1.8e308，两个类型最小的非负值大约是1.4e-45和4.9e-324。

float32大约可以提供小数点后6位的精度，作为对比，float64可以提供小数点后15位的精度。通常情况应该优先选择float64，因此float32的精确度较低，在累积计算时误差扩散很快，而且float32能精确表达的最小正整数并不大，因为浮点数和整数的底层解释方式完全不同。

类型	长度
float32	IEEE-754 32位浮点型数
float64	IEEE-754 64位浮点型数

其他数字类型：

类型	长度
byte	类似 uint8
rune	类似 int32
uint32	或 64 位
int	与 uint 一样大小
uintptr	无符号整型，用于存放一个指针

字符串：

只读的Unicode字节序列，Go语言使用UTF-8格式编码Unicode字符，每个字符对应一个rune类型。一旦字符串变量赋值之后，内部的字符就不能修改，英文是一个字节，中文是三个字节。

```
1. string转int:      int, err := strconv.Atoi(string)
2. string转int64:   int64, err := strconv.ParseInt(string, 10, 64)
3. int转string:     string := strconv.Itoa(int)
4. int64转string:   string := strconv.FormatInt(int64, 10)
```

而一个range循环会在每次迭代时，解码一个UTF-8编码的符文。每次循环时，循环的索引是当前文字的起始位置，以字节为单位，代码点是它的值（rune）。

使用range迭代字符串时，需要注意的是range迭代的是Unicode而不是字节。返回的两个值，第一个是被迭代的字符的UTF-8编码的第一个字节在字符串中的索引，第二个值的为对应的字符且类型为rune（实际就是表示unicode值的整形数据）。

```
1. const s = "Go语言"
2. for i, r := range s {
3.     fmt.Printf("%#U : %d\n", r, i)
4. }
```

程序输出：

U+0047 ‘G’ : 0

U+006F ‘o’ : 1

U+8BED ‘语’ : 2

U+8A00 ‘言’ : 5

复数：

复数类型相对用的很少，主要是数学学科专业会用上。分为两种类型 complex64和complex128 前部分是实体后部分是虚体。

类型	长度
complex64	32位实数和虚数
complex128	64位实数和虚数

2.2 Unicode (UTF-8)

你可以通过增加前缀 `0` 来表示 8 进制数（如：`077`），增加前缀 `0x` 来表示 16 进制数（如：`0xFF`），以及使用 `e` 来表示 10 的连乘（如：`1e3 = 1000`，或者 `6.022e23 = 6.022 × 1e23`）

不过 Go 同样支持 Unicode (UTF-8)，因此字符同样称为 Unicode 代码点或者 runes，并在内存中使用 `int` 来表示。在文档中，一般使用格式 `U+hhhh` 来表示，其中 `h` 表示一个 16 进制数。其实 `rune` 也是 Go 其中的一个类型，并且是 `int32` 的别名。

在书写 Unicode 字符时，需要在 16 进制数之前加上前缀 `\u` 或者 `\U`。

因为 Unicode 至少占用 2 个字节，所以我们使用 `int16` 或者 `int` 类型来表示。如果需要使用到 4 字节，则会加上 `\U` 前缀；前缀 `\u` 则总是紧跟着长度为 4 的 16 进制数，前缀 `\U` 紧跟着长度为 8 的 16 进制数。

```
1. var ch int = '\u0041'
2. var ch2 int = '\u03B2'
3. var ch3 int = '\U00101234'
```

2.3 复数

Go 拥有以下复数类型：

`complex64` (32 位实数和虚数)

`complex128` (64 位实数和虚数)

复数使用 `re+imI` 来表示，其中 `re` 代表实数部分，`im` 代表虚数部分，`I` 代表根号负 1。
示例：

```
1. var c1 complex64 = 5 + 10i
2. fmt.Printf("The value is: %v", c1) // 输出: 5 + 10i
```

如果 `re` 和 `im` 的类型均为 `float32`，那么类型为 `complex64` 的复数 `c` 可以通过以下方式获得：

```
1. c = complex(re, im)
```

函数 `real(c)` 和 `imag(c)` 可以分别获得相应的实数和虚数部分。

第三章 变量

《Go语言四十二章经》第三章 变量

作者：李骁

3.1 变量以及声明

Go 语言变量名由字母、数字、下划线组成，其中首个字母不能为数字。

```
1. var (  
2.     a int  
3.     b bool  
4.     str string  
5. )
```

这种因式分解关键字的写法一般用于声明全局变量，一般在func 外定义。

当一个变量被var声明之后，系统自动赋予它该类型的零值：

- int 为 0
- float 为 0.0
- bool 为 false
- string 为空字符串""
- 指针为 nil

记住，这些变量在 Go 中都是经过初始化的。

多变量可以在同一行进行赋值，也称为 并行 或 同时 或 平行赋值。如：

```
1. a, b, c = 5, 7, "abc"
```

简式声明：

```
1. a, b, c := 5, 7, "abc" // 注意等号前的冒号
```

右边的这些值以相同的顺序赋值给左边的变量，所以 a 的值是 5， b 的值是 7，c 的值是 "abc"。

简式声明一般用在func内，要注意的是：全局变量和简式声明的变量尽量不要同名，否则很容易产生偶

然的变量隐藏Accidental Variable Shadowing。

即使对于经验丰富的Go开发者而言，这也是一个非常常见的陷阱。这个坑很容易挖，但又很难发现。

```

1. func main() {
2.     x := 1
3.     fmt.Println(x)    // prints 1
4.     {
5.         fmt.Println(x) // prints 1
6.         x := 2
7.         fmt.Println(x) // prints 2
8.     }
9.     fmt.Println(x)    // prints 1 (不是2)
10. }
```

如果你想要交换两个变量的值，则可以简单地使用：

```
1. a, b = b, a
```

(在 Go 语言中，这样省去了使用交换函数的必要)

空白标识符 `_` 也被用于抛弃值，如值 5 在：`_, b = 5, 7`` 中被抛弃。

```
1. _, b = 5, 7
```

`_` 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

由于Go语言有个强制规定，在函数内一定要使用声明的变量，但未使用的全局变量是没问题的。为了避免有未使用的变量，代码将编译失败，我们可以将该未使用的变量改为 `_`。

另外，在Go语言中，如果引入的包未使用，也不能通过编译。有时我们需要引入的包，比如需要 `init()`，或者调试代码时我们可能去掉了某些包的功能使用，你可以添加一个下划线标记符，`_`，来作为这个包的名字，从而避免编译失败。下滑线标记符用于引入，但不使用。

```

1. package main
2.
3. import (
4.     _ "fmt"
5.     "log"
6.     "time"
7. )
```

```

8.
9.  var _ = log.Println
10. func main() {
11.     _ = time.Now
12. }

```

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 `val` 和错误 `err` 是通过调用 `Func1` 函数同时得到：

```

1. val, err = Func1(var1)

```

对于布尔值的好的命名能够很好地提升代码的可读性，例如以 `is` 或者 `Is` 开头的 `isSorted`、`isFinished`、`isVisible`，使用这样的命名能够在阅读代码的获得阅读正常语句一样的良好体验，例如标准库中的 `unicode.IsDigit(ch)`。

在 Go 语言中，指针属于引用类型，其它的引用类型还包括 `slices`，`maps`和 `channel`。

注意，Go中的数组是数值，因此当你向函数中传递数组时，函数会得到原始数组数据的一份复制。如果你打算更新数组的数据，可以考虑使用数组指针类型。

```

1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     x := [3]int{1, 2, 3}
7.
8.     func(arr *[3]int) {
9.         (*arr)[0] = 7
10.         fmt.Println(arr) // prints &[7 2 3]
11.     }(&x)
12.
13.     fmt.Println(x) // prints [7 2 3]
14. }

```

被引用的变量会存储在堆中，以便进行垃圾回收，且比栈拥有更大的内存空间。

引申：

编译器会做逃逸分析，所以由Go的编译器决定在哪(堆or栈)分配内存，保证程序的正确性。

3.2 零值nil

`nil` 标志符用于表示interface、函数、maps、slices和channels的“零值”。如果你不指定变量的类型，编译器将无法编译你的代码，因为它猜不出具体的类型。

```
1. package main
2.
3. func main() {
4.     var x = nil // 错误
5.
6.     _ = x
7. }
```

在一个 `nil` 的slice中添加元素是没问题的，但对一个map做同样的事将会生成一个运行时的panic:

```
1. package main
2.
3. func main() {
4.     var m map[string]int
5.     m["one"] = 1 //error
6.
7. }
```

字符串不会为 `nil`

这对于经常使用 `nil` 分配字符串变量的开发者而言是个需要注意的地方。

```
1. var str string = "" // ""是字符串的零值
```

根据前面的介绍，其实这样写和上面的效果一样：

```
1. var str string
```


第四章 常量

《Go语言四十二章经》第四章 常量

作者：李骁

4.1 常量以及iota

常量使用关键字 `const` 定义，用于存储不会改变的数据。

存储在常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：`const identifier [type] = value`，例如：

```
1. const Pi = 3.14159
```

在 Go 语言中，你可以省略类型说明符 `[type]`，因为编译器可以根据变量（常量）的值来推断其类型。

1. 显式类型定义：`const b string = "abc"`
2. 隐式类型定义：`const b = "abc"`

一个没有指定类型的常量被使用时，会根据其使用环境而推断出它所需要具备的类型。换句话说，未定义类型的常量会在必要时刻根据上下文来获得相关类型。

常量的值必须是能够在编译时就能够确定的；你可以在其赋值表达式中涉及计算过程，但是所有用于计算的值必须在编译期间就能获得。

在这个例子中，`iota` 可以被用作枚举值：

```
1. const (  
2.     a = iota  
3.     b = iota  
4.     c = iota  
5. )
```

第一个 `iota` 等于 0，每当 `iota` 在新的一行被使用时，它的值都会自动加 1；所以 `a=0`，`b=1`，`c=2` 可以简写为如下形式：

```
1. const (  
2.     a = iota
```

```

3.      b
4.      c
5.  )

```

注意：

```

1.  const (
2.      a = iota
3.      b = 8
4.      c
5.  )

```

a, b, c分别为0, 8, 8, 新的常量b声明后, iota 不再向下赋值, 后面常量如果没有赋值, 则继承上一个常量值。

可以简单理解为在一个const块中, 每换一行定义个常量, iota 都会自动+1。

(关于 iota 的使用涉及到非常复杂多样的情况 , 这里不展开来讲了, 有兴趣可以查查资料研究)

iota 也可以用在表达式中, 如: iota + 50。在每遇到一个新的常量块或单个常量声明时, iota 都会重置为 0 (简单地讲, 每遇到一次 **const** 关键字, **iota** 就重置为 0)。

使用位左移与 iota 计数配合可优雅地实现存储单位的常量枚举：

```

1.  type ByteSize float64
2.  const (
3.      _ = iota // 通过赋值给空白标识符来忽略值
4.      KB ByteSize = 1<<(10*iota)
5.      MB
6.      GB
7.      TB
8.      PB
9.      EB
10.     ZB
11.     YB
12. )

```

当然, 常量之所以为常量就是恒定不变的量, 因此我们无法在程序运行过程中修改它的值; 如果你在代码中试图修改常量的值则会引发编译错误。同时, 在const 定义中, 对常量名没有强制要求全部大写, 不过我们一般都会全部字母大写, 以便阅读。

第五章 作用域

《Go语言四十二章经》第五章 作用域

作者：李骁

5.1 作用域

- 局部变量

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

- 全局变量

作用域都是全局的（在本包范围内）

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。全局变量可以在任何函数中使用。

- 简式变量

使用 `:=` 定义的变量，如果新变量 `p` 与那个同名已定义变量（这里就是那个全局变量 `p`）不在一个作用域中时，那么 Go 语言会新定义这个变量 `p`，遮盖住全局变量 `p`。刚开始很容易在此犯错而茫然，解决方法是局部变量尽量不同名。

注意，简式变量只能在函数内部声明使用，但是它可能会覆盖函数外全局同名变量。而且你不能在一个单独的声明中重复声明一个变量，但在多变量声明中这是允许的，而且其中至少要有一个新的声明变量。重复变量需要在相同的代码块内，否则你将得到一个隐藏变量。

如果你在代码块中犯了这个错误，将不会出现编译错误，但应用运行结果可能不是你所期望。所以尽可能避免和全局变量同名。

第六章 约定和惯例

《Go语言四十二章经》第六章 约定和惯例

作者：李骁

6.1 可见性规则

包通过下面这个被编译器强制执行的规则来决定是否将自身的代码对象暴露给外部文件：

当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 private）。

（大写字母可以使用任何 Unicode 编码的字符，比如希腊文，不仅仅是 ASCII 码中的大写字母）。

因此，在导入一个外部包后，能够且只能够访问该包中导出的对象。

假设在包 pack1 中我们有一个变量或函数叫做 Thing（以 T 开头，所以它能够被导出），那么在当前包中导入 pack1 包，Thing 就可以像面向对象语言那样使用点标记来调用：

```
1. pack1.Thing // (pack1 在这里是不可以省略的)
```

因此包也可以作为命名空间使用，帮助避免命名冲突（名称冲突）：两个包中的同名变量的区别在于他们的包名，例如 pack1.Thing 和 pack2.Thing。

注意事项：

如果你导入了一个包却没有使用它，则会在构建程序时引发错误，如 *imported and not used: os*，这正是遵循了 Go 的格言：“没有不必要的代码！”。

6.2 命名规范以及语法惯例

干净、可读的代码和简洁性是 Go 追求的主要目标。通过 Gofmt 来强制实现统一的代码风格。Go 语言中对象的命名也应该是简洁且有意义的。像 Java 和 Python 中那样使用混合着大小写和下划线的冗长的名称会严重降低代码的可读性。名称不需要指出自己所属的包，因为在调用的时候会使用包名作为限定符。返回某个对象的函数或方法的名称一般都是使用名词，没有 Get... 之类的字符，如果是用于修改某个对象，则使用 SetName。有必须要的话可以使用大小写混合的方式，如 MixedCaps 或 mixedCaps，而不是使用下划线来分割多个名称。

函数里的代码（函数体）使用大括号 `{}` 括起来。

左大括号 `{` 必须与方法的声明放在同一行，这是编译器的强制规定，否则你在使用 `Gofmt` 时就会出现错误提示：

Go 语言虽然看起来不使用分号作为语句的结束，但实际上这一过程是由编译器自动完成，因此才会引发像上面这样的错误。

右大括号 `}` 需要被放在紧接着函数体的下一行。如果你的函数非常简短，你也可以将它们放在同一行：

```
1. func Sum(a, b int) int { return a + b }
```

对于大括号 `{}` 的使用规则在任何时候都是相同的（如：`if` 语句等）。

因此符合规范的函数一般写成如下的形式：

```
1. func functionName(parameter_list) (return_value_list) {
2.     ...
3. }
```

只有当某个函数需要被外部包调用的时候才使用大写字母开头，并遵循 `Pascal` 命名法；否则就遵循骆驼命名法，即第一个单词的首字母小写，其余单词的首字母大写。

单字之间不以空格断开或连接号（`-`）、底线（`_`）连结，第一个单字首字母采用大写字母；后续单字的首字母亦用大写字母，例如：`FirstName`、`LastName`。每一个单字的首字母都采用大写字母的命名格式，被称为“`Pascal`命名法”，源自于`Pascal`语言的命名惯例，也有人称之为“大驼峰式命名法”（`Upper Camel Case`），为驼峰式大小写的子集。

帕斯卡命名法指当变量名和函数名称是由二个或二个以上单字连结在一起，而构成的唯一识别字时，用以增加变量和函数的可读性。

单行注释是最常见的注释形式，你可以在任何地方使用以 `//` 开头的单行注释。

多行注释也叫块注释，均以 `/*` 开头，并以 `*/` 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。

```
1. // Cap returns the capacity of the buffer's underlying byte slice,
2. // that is, the total space allocated for the buffer's data.
3.
4. /*
5.     Cap returns the capacity of the buffer's underlying byte slice,
6.     that is, the total space allocated for the buffer's data.
```

7. */

在Go标准库中，一般都采用单行注释，建议采用官方标准方式。

每一个包应该有相关注释，在 `package` 语句之前的块注释将被默认认为是这个包的文档说明，其中应该提供一些相关信息并对整体功能做简要的介绍。一个包可以分散在多个文件中，但是只需要在其中一个进行注释说明即可。当开发人员需要了解包的一些情况时，自然会用 `Godoc` 来显示包的文档说明，在首行的简要注释之后可以用成段的注释来进行更详细的说明，而不必拥挤在一起。

另外，在多段注释之间应以空行分隔加以区分，单行注释的 `//` 后面空一格，方便 `godoc` 生成标准文档。

```
1. // Copyright 2009 The Go Authors. All rights reserved.  
2. // Use of this source code is governed by a BSD-style  
3. // license that can be found in the LICENSE file.  
4.  
5. // Package hash provides interfaces for hash functions.  
6. package hash
```

几乎所有全局作用域的类型、常量、变量、函数和被导出的对象都应该有一个合理的注释。如果这种注释（称为文档注释）出现在函数前面，例如函数 `Abcd`，则要以 `“Abcd...”` 作为开头。

```
1. // enterOrbit causes Superman to fly into low Earth orbit, a position  
2. // that presents several possibilities for planet salvation.  
3. func enterOrbit() error {  
4.     ...  
5. }
```

第七章 代码结构化

《Go语言四十二章经》第七章 代码结构化

作者：李骁

7.1 包的概念

包是结构化代码的一种方式：每个程序都由包（通常简称为 `pkg`）的概念组成，可以使用自身的包或者从其它包中导入内容。

如同其它一些编程语言中的类库或命名空间的概念，每个 Go 文件都属于且仅属于一个包。一个包可以由许多以 `.go` 为扩展名的源文件组成，因此文件名和包名一般来说都是不相同的。

你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：`package main`。

`package main`表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 `main` 的包。

`package main`包下可以有多个文件，但所有文件中只能有一个`main()`方法，`main()`方法代表程序入口。

一个应用程序可以包含不同的包，而且即使你只使用 `main` 包也不必把所有的代码都写在一个巨大的文件里：你可以用一些较小的文件，并且在每个文件非注释的第一行都使用 `package main` 来指明这些文件都属于 `main` 包。如果你打算编译包名不是为 `main` 的源文件，如 `pack1`，编译后产生的对象文件将会是 `pack1.a` 而不是可执行程序。另外要注意的是，所有的包名都应该使用小写字母。当然，`main`包是不能在其他文档`import`的，编译器会报错：

```
1. import "xx/xx" is a program, not an importable package.
```

简单地说，在含有`mian`包的目录下，你可以写多个文件，每个文件非注释的第一行都使用 `package main` 来指明这些文件都属于这个应用的 `main` 包，只有一个文件能有`mian()`方法，也就是应用程序的入口。`main`包不是必须的，只有在可执行的应用程序中需要。

7.2 包的导入

一个 Go 程序是通过 `import` 关键字将一组包链接在一起。

`import "fmt"` 告诉 Go 编译器这个程序需要使用 `fmt` 包（的函数，或其他元素），`fmt` 包实现了格式化 IO（输入/输出）的函数。包名被封闭在半角双引号 `""` 中。如果你打算从已编译的包中导入并加载公开声明的方法，不需要插入已编译包的源代码。

import 其实是导入目录，而不是定义的package名字，虽然我们一般都会保持一致，但其实是可以随便定义目录名，只是使用时会很容易混乱，不建议这么做。

例如：package big，我们import “math/big”，其实是在src中的src/math目录。在代码中使用big.Int时，big指的才是Go文件中定义的package名字。

当你导入多个包时，最好按照字母顺序排列包名，这样做更加清晰易读。

如果包名不是以 ./，如 “fmt” 或者 “container/list”，则 Go 会在全局文件进行查找；如果包名以 ./ 开头，则 Go 会在相对目录中查找。

导入包即等同于包含了这个包的所有的代码对象。

除了符号 _，包中所有代码对象的标识符必须是唯一的，以避免名称冲突。但是相同的标识符可以在不同的包中使用，因为可以使用包名来区分它们。

导入包的路径的几种情况：

- 第一种方式相对路径

```
1. import  "./module"    //当前文件同一目录的module目录， 此方式没什么用容易出错，不建议用
```

- 第二种方式绝对路径

```
1. import  "LearnGo/init"    //加载Gopath/src/LearnGo/init模块，一般建议这样使用""
```

导入多个包的常见的方式是：

```
1. import (
2.  "fmt"
3.  "net/http"
4. )
```

调用导入的包函数的一般方式：

```
1. fmt.Println("Hello World!")
```

下面展示一些特殊的import方式

- 点操作

```
import( . "fmt" )
```

这个点操作的含义就是这个包导入之后在你调用这个包的函数时，你可以省略前缀的包名，如可以省略的写成Println(“hello world!”)

- 别名操作

别名操作顾名思义我们可以把包命名成另一个我们用起来容易记忆的名字。

```
1. import(
2.     f "fmt"
3. )
```

别名操作调用包函数时前缀变成了我们的前缀，即 `f.Println("hello world")`。在实际项目中有这样使用，但请谨慎使用，不要广泛采用这种形式。

- `_`操作

`_`操作其实是引入该包，而不直接使用包里面的函数，而是调用了该包里面的 `init` 函数。

```
1. import (
2.     _ "github.com/revel/modules/testrunner/app"
3.     _ "guild_website/app"
4. )
```

7.3 标准库

在 Go 的安装文件里包含了一些可以直接使用的包，即标准库。在 Windows 下，标准库的位置在 Go 根目录下的子目录 `pkg\windows_386` 中；在 Linux 下，标准库在 Go 根目录下的子目录 `pkg\linux_amd64` 中（如果是安装的是 32 位，则在 `linux_386` 目录中）。Go 的标准库包含了大量的包（如：`fmt` 和 `os`），在 `$GoROOT/src` 中可以看到源码，也可以根据情况自行重新编译。

完整列表可以在 Go Walker 查看。

1. `unsafe`：包含了一些打破 Go 语言“类型安全”的命令，一般的程序中不会被使用，可用在 C/C++ 程序的调用中。
2. `syscall-os-os/exec`：
3. `os`：提供给我们一个平台无关性的操作系统功能接口，采用类 Unix 设计，
4. 隐藏了不同操作系统间差异，让不同的文件系统和操作系统对象表现一致。
5. `os/exec`：提供我们运行外部操作系统命令和程序的方式。
6. `syscall`：底层的外部包，提供了操作系统底层调用的基本接口。
7. `archive/tar` 和 `/zip-compress`：压缩(解压缩)文件功能。
8. `fmt-io-bufio-path/filepath-flag`：
9. `fmt`：提供了格式化输入输出功能。
10. `io`：提供了基本输入输出功能，大多数是围绕系统功能的封装。
11. `bufio`：缓冲输入输出功能的封装。
12. `path/filepath`：用来操作在当前系统中的目标文件名路径。
13. `flag`：对命令行参数的操作。

```

14. strings-strconv-unicode-regex-bytes:
15.     strings: 提供对字符串的操作。
16.     strconv: 提供将字符串转换为基本类型的功能。
17.     unicode: 为 unicode 型的字符串提供特殊的功能。
18.     regexp: 正则表达式功能。
19.     bytes: 提供对字符型分片的操作。
20. math-math/cmath-math/big-math/rand-sort:
21.     math: 基本的数学函数。
22.     math/cmath: 对复数的操作。
23.     math/rand: 伪随机数生成。
24.     sort: 为数组排序和自定义集合。
25.     math/big: 大数的实现和计算。
26. container-/list-ring-heap: 实现对集合的操作。
27.     list: 双链表。
28.     ring: 环形链表。
29. time-log:
30.     time: 日期和时间的基本操作。
31.     log: 记录程序运行时产生的日志。
32. encoding/Json-encoding/xml-text/template:
33.     encoding/Json: 读取并解码和写入并编码 Json 数据。
34.     encoding/xml: 简单的 XML1.0 解析器。
35.     text/template: 生成像 HTML 一样的数据与文本混合的数据驱动模板。
36. net-net/http-html:
37.     net: 网络数据的基本操作。
38.     http: 提供了一个可扩展的 HTTP 服务器和客户端, 解析 HTTP 请求和回复。
39.     html: HTML5 解析器。
40. runtime: Go 程序运行时的交互操作, 例如垃圾回收和协程创建。
41. reflect: 实现通过程序运行时反射, 让程序操作任意类型的变量。

```

7.4 从 GitHub 安装包

如果有人想安装您的远端项目到本地机器, 打开终端并执行 (ffhelicopter是我在 GitHub 上的用户名):

```
1. go get -u github.com/ffhelicopter/tmm
```

这样现在这台机器上的其他 Go 应用程序也可以通过导入路

径: "github.com/ffhelicopter/tmm" 代替 "./ffhelicopter/tmm" 来使用。也可以将其缩写为: import ind "github.com/ffhelicopter/tmm"; 开发中一般这样操作:

```
1. import "github.com/ffhelicopter/tmm"
```

Go 对包的版本管理做的不是很友好，不过现在有些第三方项目做的不错，有兴趣的同学可以了解下（glide、godep、govendor）。

7.5 导入外部安装包

如果你要在你的应用中使用一个或多个外部包，你可以使用 `Go install` 在你的本地机器上安装它们。`Go install` 是 Go 中自动包安装工具：如需要将包安装到本地它会从远端仓库下载包：检出、编译和安装一气呵成。

在包安装前的先决条件是要自动处理包自身依赖关系的安装。被依赖的包也会安装到子目录下，但是没有文档和示例：可以到网上浏览。

Go install 使用了 **GoPATH** 变量

假设你想使用<https://github.com/gocolly/colly> 这种托管在 Google Code、GitHub 和 Launchpad 等代码网站上的包。

你可以通过如下命令安装：`Go install github.com/gocolly/colly` 将一个名为 `github.com/gocolly/colly` 安装在 `$GoPATH/pkg/` 目录下。

`Go install/build` 都是用来编译包和其依赖的包。

区别：`Go build` 只对 `main` 包有效，在当前目录编译生成一个可执行的二进制文件（依赖包生成的静态库文件放在 `$GoPATH/pkg`）。

`Go install` 一般生成静态库文件放在 `$GoPATH/pkg` 目录下，文件扩展名 `a`。

如果为 `main` 包，运行 `Go build` 则会在 `$GoPATH/bin` 生成一个可执行的二进制文件。

7.6 包的分级声明和初始化

你可以在使用 `import` 导入包之后定义或声明 0 个或多个常量（`const`）、变量（`var`）和类型（`type`），这些对象的作用域都是全局的（在本包范围内），所以可以被本包中所有的函数调用，然后声明一个或多个函数（`func`）。

如果存在 `init` 函数的话，则对该函数进行定义（这是一个特殊的函数，每个含有该函数的包都会首先执行这个函数）。

程序开始执行并完成初始化后，第一个调用（程序的入口点）的函数是 `main.main()`（如果有 `init()` 函数则会先执行该函数）。

如果你的 `main` 包的源代码没有包含 `main` 函数，则会引发构建错误 `undefined: main.main`。`main` 函数既没有参数，也没有返回类型，这一点上 `init` 函数和 `main` 函数一样。

main函数一旦返回就表示程序已成功执行并立即退出。

Go 程序的执行（程序启动）顺序如下：

程序的初始化和执行都起始于main包。如果main包还导入了其它的包，那么就会在编译时将它们依次导入。有时一个包会被多个包同时导入，那么它只会被导入一次（例如很多包可能都会用到fmt包，但它只会被导入一次，因为没有必要导入多次）。当一个包被导入时，如果该包还导入了其它的包，那么会先将其它包导入进来，然后再对这些包中的包级常量和变量进行初始化，接着执行init函数（如果有的话），依次类推。等所有被导入的包都加载完毕了，就会开始对main包中的包级常量和变量进行初始化，然后执行main包中的init函数（如果存在的话），最后执行main函数。

Go语言中init函数用于包(package)的初始化，该函数是Go语言的一个重要特性，有下面的特征：

- init函数是用于程序执行前做包的初始化的函数，比如初始化包里的变量等
- 每个包可以拥有多个init函数
- 包的每个源文件也可以拥有多个init函数
- 同一个包中多个init函数的执行顺序Go语言没有明确的定义(说明)
- 不同包的init函数按照包导入的依赖关系决定该初始化函数的执行顺序
- init函数不能被其他函数调用，而是在main函数执行之前，自动被调用

第八章 Go项目开发与编译

《Go语言四十二章经》第八章 Go项目开发与编译

作者：李骁

8.1 项目结构

Go的工程项目管理非常简单，使用目录结构和package名来确定工程结构和构建顺序。

环境变量GOPATH在项目管理中非常重要，想要构建一个项目，必须确保项目目录在GOPATH中。而GOPATH可以有多个项目用";"分隔。

Go 项目目录下一般有三个子目录：

- src存放源代码
- pkg编译后生成的文件
- bin编译后生成的可执行文件

我们重点要关注的其实就是src文件夹中的目录结构。

为了进行一个项目，我们会在GoPATH目录下的src目录中，新建立一个项目的主要目录，比如我写的一个WEB项目《使用gin快速搭建WEB站点以及提供RestFull接口》。

<https://github.com/ffhelicopter/tmm>

项目主要目录“tmm”： `$GOPATH/src/github.com/ffhelicopter/tmm`

在这个目录(tmm)下面还有其他目录，分别放置了其他代码，大概结构如下：

```
1.  src/github.com/ffhelicopter/tmm
2.                                     /api
3.                                     /handler
4.                                     /model
5.                                     /task
6.                                     /website
7.                                     main.go
```

main.go 文件中定义了package main 。同时也在文件中import了

```
1.  "github.com/ffhelicopter/tmm/api"
2.  "github.com/ffhelicopter/tmm/handler"
```

2个自定义包。

上面的目录结构是一般项目的目录结构，基本上可以满足单个项目开发的需要。如果需要构建多个项目，可按照类似的结构，分别建立不同项目目录。

当我们运行 `go install main.go` 会在GOPATH的bin 目录中生成可执行文件。

8.2 使用 Godoc

在程序中我们一般都会注释，如果我们按照一定规则，Godoc 工具会收集这些注释并产生一个技术文档。

Godoc工具在显示自定义包中的注释也有很好的效果：

注释必须以 `//` 开始并无空行放在声明（包，类型，函数）前。Godoc 会为每个文件生成一系列的网页。

```

1. // Copyright 2009 The Go Authors. All rights reserved.
2. // Use of this source code is governed by a BSD-style
3. // license that can be found in the LICENSE file.
4.
5. package zlib
6. ....
7.
8. // A Writer takes data written to it and writes the compressed
9. // form of that data to an underlying writer (see NewWriter).
10. type Writer struct {
11.     w          io.Writer
12.     level      int
13.     dict       []byte
14.     compressor * flate.Writer
15.     digest     hash.Hash32
16.     err        error
17.     scratch    [4]byte
18.     wroteHeader bool
19. }
20.
21. // NewWriter creates a new Writer.
22. // Writes to the returned Writer are compressed and written to w.
23. //
24. // It is the caller's responsibility to call Close on the WriteCloser when
25. // done.
26. // Writes may be buffered and not flushed until Close.
```

```

26. func NewWriter(w io.Writer) * Writer {
27.     z, _ := NewWriterLevelDict(w, DefaultCompression, nil)
28.     return z
29. }

```

一般采用行注释，每个// 后面采用一个空格隔开。注意函数或方法的整体注释以函数名或方法名开始，空换行也用//注释。整个注释和函数签名之间不要空行。

执行：

在 `doc_examples` 目录下我们有 `Go` 文件，文件中有一些注释（文件需要未编译） 命令行下进入目录下并输入命令：`Godoc -http=:6060 -goroot="."`

（. 是指当前目录，`-goroot` 参数可以是 `/path/to/my/package1` 这样的形式指出 `package1` 在你源码中的位置或接受用冒号形式分隔的路径，无根目录的路径为相对于当前目录的相对路径）

在浏览器打开地址：<http://localhost:6060>

然后你会看到本地的 `Godoc` 页面，从左到右一次显示出目录中的包。



8.3 Go程序的编译

如果想要构建一个程序，则包和包内的文件都必须以正确的顺序进行编译。包的依赖关系决定了其构建顺序。

属于同一个包的源文件必须全部被一起编译，一个包即是编译时的一个单元，因此根据惯例，每个目录都只包含一个包。

如果对一个包进行更改或重新编译，所有引用了这个包的客户端程序都必须全部重新编译。

Go 中的包模型采用了显式依赖关系的机制来达到快速编译的目的，编译器会从后缀名为 `.go` 的对象文件（需要且只需要这个文件）中提取传递依赖类型的信息。

如果 `A.go` 依赖 `B.go`，而 `B.go` 又依赖 `C.go`：

编译 `C.go`，`B.go`，然后是 `A.go`。

为了编译 `A.go`，编译器读取的是 `B.go` 而不是 `C.go`。

这种机制对于编译大型的项目时可以显著地提升编译速度，每一段代码只会被编译一次。

第九章 运算符

《Go语言四十二章经》第九章 运算符

作者：李骁

9.1 内置运算符

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

Go语言的算术运算符：

运算符	含义	示意
+	相加	A + B
-	相减	A - B
*	相乘	A * B
/	相除	B / A 结果还是整数 8/3=2
%	求余	B % A
++	自增	A++ 1
—	自减	A—

Go语言的关系运算符：

运算符	含义	示意
==	检查两个值是否相等。	(A == B) 为 False
!=	检查两个值是否不相等。	(A != B) 为 True
>	检查左边值是否大于右边值。	(A > B) 为 False
<	检查左边值是否小于右边值。	(A < B) 为 True
>=	检查左边值是否大于等于右边值。	(A >= B) 为 False
<=	检查左边值是否小于等于右边值。	(A <= B) 为 True

Go语言的逻辑运算符：

运算符	操作	含义
-----	----	----

&&	逻辑与	如果两边的操作数都是 True，则条件 True，否则为 False。
	逻辑或	如果两边的操作数有一个 True，则条件 True，否则为 False。
!	逻辑非	如果条件为 True，则逻辑 NOT 条件 False，否则为 True。

Go语言的位运算符：

位运算符对整数在内存中的二进制位进行操作。

下表列出了位运算符 &，|，和 ^ 的计算：

位	位	& 与	或	^ 异或
p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Go 语言支持的位运算符含义。

- & 按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。
- | 按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或。
- ^ 按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。
- << 左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。
- >> 右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。

Go语言的赋值运算符：

运算符	含义	示意
=	简单的赋值运算符	
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C = A 等于 C = C A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移后赋值	C <<= 2 等于 C = C << 2
>>=	右移后赋值	C >>= 2 等于 C = C >> 2

&=	按位与后赋值	C &= 2 等于 C = C & 2
^=	按位异或后赋值	C ^= 2 等于 C = C ^ 2
=	按位或后赋值	C = 2 等于 C = C 2

Go语言的其他运算符：

运算符	含义
&	返回变量存储地址 &a；将给出变量的实际地址。
*	指针变量。 *a；是一个指针变量

9.2 运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

优先级	运算符
7	^ !
6	* / % << >> & &^
5	+ - ^
4	== != < <= >= >
3	<-
2	&&
1	

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

9.3 几个特殊运算符

位清除 &^：

将指定位置上的值设置为 0。将运算符左边数据相异的位保留，相同位清零：

```
1. x=2
2. y=4
3. x&^y==x&(^y)
```

首先我们先换算成2进制 0000 0010 &^ 0000 0100 = 0000 0010 如果y bit位上的数是0则取x上对应位置的值， 如果y bit位上为1则结果位上取0

1、如果右侧是0，则左侧数保持不变

2、如果右侧是1，则左侧数一定清零

3、功能同 $a \& (\sim b)$ 相同

4、如果左侧是变量，也等同于：

```
1. var a int
2. a &^= b
```

\wedge (XOR) 在Go语言中XOR是作为二元运算符存在的：

但是如果是作为一元运算符出现，他的意思是按位取反。

如果作为二元运算符则是，XOR是不进位加法计算，也就是异或计算。 $0000\ 0100 + 0000\ 0010 = 0000\ 0110 = 6$

常见可用于整数和浮点数的二元运算符有 $+$ 、 $-$ 、 $*$ 和 $/$ 。

（相对于一般规则而言，Go 在进行字符串拼接时允许使用对运算符 $+$ 的重载，但 Go 本身不允许开发者进行自定义的运算符重载）

对于整数运算而言，结果依旧为整数，例如： $9 / 4 \rightarrow 2$ 。

取余运算符只能作用于整数： $9 \% 4 \rightarrow 1$ 。

浮点数除以 0.0 会返回一个无穷尽的结果，使用 $+\text{Inf}$ 表示。

你可以将语句 $b = b + a$ 简写为 $b+=a$ ，同样的写法也可用于 $-$ 、 $*$ 、 $/$ 、 $\%$ 。

对于整数和浮点数，你可以使用一元运算符 $++$ （递增）和 $--$ （递减），但只能用于后缀：

$i++ \rightarrow i += 1 \rightarrow i = i + 1$

$i-- \rightarrow i -= 1 \rightarrow i = i - 1$

同时，带有 $++$ 和 $--$ 的只能作为语句，而非表达式，因此 $n = i++$ 这种写法是无效的。

函数 `rand.Float32` 和 `rand.Float64` 返回介于 $[0.0, 1.0)$ 之间的伪随机数，其中包括 0.0 但不包括 1.0 。函数 `rand.Intn` 返回介于 $[0, n)$ 之间的伪随机数。

你可以使用 `Seed(value)` 函数来提供伪随机数的生成种子，一般情况下都会使用当前时间的纳秒级数字。

第十章 string

《Go语言四十二章经》第十章 string

作者：李骁

10.1 有关string

Go 语言中的string类型存储的字符串是不可变的，如果要修改string内容需要将string转换为[]byte或[]rune，并且修改后的string内容是重新分配的。

那么byte和rune的区别是什么(下面写法是type别名)：

```
1. type byte = uint8
2. type rune = int32
```

string 类型的零值为长度为零的字符串，即空字符串 ""。

一般的比较运算符(==、!=、<、<=、>=、>)通过在内存中按字节比较来实现字符串的对比。你可以通过函数 len() 来获取字符串所占的字节长度，例如：len(str)。

字符串的内容(纯字节)可以通过标准索引法来获取，在中括号 [] 内写入索引，索引从 0 开始计数：

```
1. 字符串 str 的第 1 个字节：str[0]
2. 第 i 个字节：str[i - 1]
3. 最后 1 个字节：str[len(str)-1]
```

需要注意的是，这种转换方案只对纯 ASCII 码的字符串有效。

如字符串含有中文等字符，我们可以看到每个中文字符的索引值相差3。下面代码同时说明了在for range循环处理字符时，不是按照字节的方式来处理的。v实际上是一个rune类型值。

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. func main() {
```

```

8.      s := "Go语言四十二章经"
9.      for k, v := range s {
10.         fmt.Printf("k:%d,v:%c == %d\n", k, v, v)
11.      }
12.  }
```

```

1.  程序输出：
2.  k:0,v:G == 71
3.  k:1,v:o == 111
4.  k:2,v:语 == 35821
5.  k:5,v:言 == 35328
6.  k:8,v:四 == 22235
7.  k:11,v:十 == 21313
8.  k:14,v:二 == 20108
9.  k:17,v:章 == 31456
10. k:20,v:经 == 32463
```

注意事项：

获取字符串中某个字节的地址的行为是非法的，例如：`&str[i]`。

10.2 字符串拼接

可以通过以下方式对代码中多行的字符串进行拼接。

- 直接使用运算符

```

1.  str := "Beginning of the string " +
2.  "second part of the string"
```

由于编译器行尾自动补全分号的缘故，加号 `+` 必须放在第一行。

拼接的简写形式 `+=` 也可以用于字符串：

```

1.  s := "hel" + "lo, "
2.  s += "world!"
3.  fmt.Println(s) // 输出 "hello, world!"
```

里面的字符串都是不可变的，每次运算都会产生一个新的字符串，所以会产生很多临时的无用的字符串，不仅没有用，还会给 gc 带来额外的负担，所以性能比较差。

- `fmt.Sprintf()`

```
1. fmt.Sprintf("%d:%s", 2018, "年")
```

内部使用 `[]byte` 实现，不像直接运算符这种会产生很多临时的字符串，但是内部的逻辑比较复杂，有很多额外的判断，还用到了 `interface`，所以性能一般。

- `strings.Join()`

```
1. strings.Join([]string{"hello", "world"}, ", ")
```

`Join`会先根据字符串数组的内容，计算出一个拼接之后的长度，然后申请对应大小的内存，一个一个字符串填入，在已有一个数组的情况下，这种效率会很高，但是本来没有，去构造这个数据的代价也不小。

- `bytes.Buffer`

```
1. var buffer bytes.Buffer
2. buffer.WriteString("hello")
3. buffer.WriteString(", ")
4. buffer.WriteString("world")
5.
6. fmt.Print(buffer.String())
```

这个比较理想，可以当成可变字符串使用，对内存的增长也有优化，如果能预估字符串的长度，还可以用 `buffer.Grow()` 接口来设置 `capacity`。

- `strings.Builder`

```
1. var b1 strings.Builder
2. b1.WriteString("ABC")
3. b1.WriteString("DEF")
4.
5. fmt.Print(b1.String())
```

`strings.Builder` 内部通过 `slice` 来保存和管理内容。`slice` 内部则是通过一个指针指向实际保存内容的数组。`strings.Builder` 同样也提供了 `Grow()` 来支持预定义容量。当我们可以预定义我们需要使用的容量时，`strings.Builder` 就能避免扩容而创建新的 `slice` 了。

`strings.Builder`是非线程安全，性能上和 `bytes.Buffer` 相差无几。

第十一章 数组(Array)

《Go语言四十二章经》第十一章 数组(Array)

作者：李骁

11.1 数组(Array)

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列（这是一种同构的数据结构）；这种类型可以是任意的原始类型例如整型、字符串或者自定义类型。数组长度必须是一个常量表达式，并且必须是一个非负整数。

数组长度也是数组类型的一部分，所以`[5]int`和`[10]int`是属于不同类型的。

注意事项：如果我们想让数组元素类型为任意类型的话可以使用空接口作为类型。当使用值时我们必须先做一个类型判断。

数组元素可以通过 索引（位置）来读取（或者修改），索引从 0 开始，第一个元素索引为 0，第二个索引为 1，以此类推。（数组以 0 开始在所有类 C 语言中是相似的）。元素的数目，也称为长度或者数组大小必须是固定的并且在声明该数组时就给出（编译时需要知道数组长度以便分配内存）；数组长度最大为 2Gb。

如果每个元素是一个整型值，当声明数组时所有的元素都会被自动初始化为默认值 0。

遍历数组的方法既可以for 条件循环，也可以使用 for-range。这两种 for 结构对于切片（slices）来说也同样适用。

Go 语言中的数组是一种值类型（不像 C/C++ 中是指向首元素的指针），所以可以通过 `new()` 来创建：

```
1. var arr1 = new([5]int)
```

那么这种方式 and `var arr2 [5]int` 的区别是什么呢？`arr1` 的类型是 `*[5]int`，而 `arr2` 的类型是 `[5]int`。

把一个大数组传递给函数会消耗很多内存。有两种方法可以避免这种现象：

1. 传递数组的指针
2. 使用数组的切片

以上我们通常使用切片。

```
1. var arrLazy = [...]int{5, 6, 7, 8, 22}
```

这里是不定长数组，其长度是根据初始化时指定的元素个数决定的。

几种赋值方式：

```
1. var arrAge = [5]int{18, 20, 15, 22, 16}
2. var arrLazy = [...]int{5, 6, 7, 8, 22}
3. var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
```

本书《Go语言四十二章经》内容在github上同步地址：<https://github.com/ffhelicopter/Go42>

本书《Go语言四十二章经》内容在简书同步地址：<https://www.jianshu.com/nb/29056963>

虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。

联系邮箱：roteman@163.com

第十二章 切片(slice)

《Go语言四十二章经》第十二章 切片(slice)

作者：李骁

12.1 切片(slice)

切片 (**slice**) 是对数组一个连续片段的引用 (该数组我们称之为相关数组, 通常是匿名的), 所以切片是一个引用类型 (和数组不一样)。这个片段可以是整个数组, 或者是由起始和终止索引标识的一些项的子集。需要注意的是, 终止索引标识的项不包括在切片内。切片提供了一个相关数组的动态窗口 (这里有关动态窗口的含义, 可参考数据库窗口函数的解释)。

切片是可索引的, 并且可以由 `len()` 函数获取长度。

给定项的切片索引可能比相关数组的相同元素的索引小。和数组不同的是, 切片的长度可以在运行时修改, 最小为 0 最大为相关数组的长度: 切片是一个 长度可变的数组。

切片提供了计算容量的函数 `cap()` 可以测量切片最长可以达到多少: 它等于切片的长度 + 数组除切片之外的长度。如果 `s` 是一个切片, `cap(s)` 就是从 `s[0]` 到数组末尾的数组长度。切片的长度永远不会超过它的容量, 所以对于切片 `s` 来说该不等式永远成立: $0 \leq \text{len}(s) \leq \text{cap}(s)$ 。

多个切片如果表示同一个数组的片段, 它们可以共享数据; 因此一个切片和相关数组的其他切片是共享存储的, 相反, 不同的数组总是代表不同的存储。数组实际上是切片的构建块。

优点

因为切片是引用, 所以它们不需要使用额外的内存并且比使用数组更有效率, 所以在 Go 代码中切片比数组更常用。

注意

绝对不要用指针指向 slice, 切片本身已经是一个引用类型, 所以它本身就是一个指针!

声明切片的格式是: `var identifier []type` (不需要说明长度)。

一个切片在未初始化之前默认为 `nil`, 长度为 0。

切片的初始化格式是:

```
1. var slice1 []type = arr1[start:end]
```

这表示 slice1 是由数组 arr1 从 start 索引到 end-1 索引之间的元素构成的子集（切分数组，start:end 被称为 slice 表达式）。

切片也可以用类似数组的方式初始化：

```
1. var x = []int{2, 3, 5, 7, 11}
```

这样就创建了一个长度为 5 的数组并且创建了一个相关切片。

当相关数组还没有定义时，我们可以使用 make() 函数来创建一个切片 同时创建好相关数组：

```
1. var slice1 []type = make([]type, len)
```

也可以简写为 slice1 := make([]type, len)，这里 len 是数组的长度并且也是 slice 的初始长度。

make 的使用方式是：func make([]T, len, cap)，其中 cap 是可选参数。

```
1. v := make([]int, 10, 50)
```

这样分配一个有 50 个 int 值的数组，并且创建了一个长度为 10，容量为 50 的切片 v，该切片指向数组的前 10 个元素。

以上我们列举了三种切片初始化方式，这三种方式都比较常用。

如果从数组或者切片中生成一个新的切片，我们可以使用下面的表达式：

```
1. a[low : high : max]
```

max-low的结果表示容量。

```
1. a := []int{1, 2, 3, 4, 5}
2. t := a[1:3:5]
```

这里t的容量 (capacity) 是5-1=4，长度是2。

12.2 切片重组(reslice)

```
1. slice1 := make([]type, start_length, capacity)
```

其中 start_length 作为切片初始长度而 capacity 作为相关数组的长度。

这么做的好处是我们的切片在达到容量上限后可以扩容。改变切片长度的过程称之为切片重组 reslicing，做法如下：`slice1 = slice1[0:end]`，其中 `end` 是新的末尾索引（即长度）。

当你重新划分一个slice时，新的slice将引用原有slice的数组。如果你忘了这个行为的话，在你的应用分配大量临时的slice用于创建新的slice来引用原有数据的一小部分时，会导致难以预期的内存使用。

```

1. package main
2.
3. import "fmt"
4.
5. func get() []byte {
6.     raw := make([]byte, 10000)
7.     fmt.Println(len(raw), cap(raw), &raw[0]) // prints: 10000 10000
8.     return raw[:3] // 10000个字节实际只需要引用3个，其他空间浪费
9. }
10.
11. func main() {
12.     data := get()
13.     fmt.Println(len(data), cap(data), &data[0]) // prints: 3 10000
14. }

```

为了避免这个陷阱，你需要从临时的slice中拷贝数据（而不是重新划分slice）。

```

1. package main
2.
3. import "fmt"
4.
5. func get() []byte {
6.     raw := make([]byte, 10000)
7.     fmt.Println(len(raw), cap(raw), &raw[0]) // prints: 10000 10000
8.     res := make([]byte, 3)
9.     copy(res, raw[:3]) // 利用copy 函数复制，raw 可被GC释放
10.    return res
11. }
12.
13. func main() {
14.     data := get()
15.     fmt.Println(len(data), cap(data), &data[0]) // prints: 3 3 <byte_addr_y>

```

```
16. }
```

`func append(s []T, x ...T) []T` 其中 `append` 方法将 0 个或多个具有相同类型 `s` 的元素追加到切片后面并且返回新的切片；追加的元素必须和原切片的元素同类型。如果 `s` 的容量不足以存储新增元素，`append` 会分配新的切片来保证已有切片元素和新增元素的存储。因此，返回的切片可能已经指向一个不同的相关数组了。`append` 方法总是返回成功，除非系统内存耗尽了。

`append`操作如果导致分配新的切片来保证已有切片元素和新增元素的存储，那么新的slice已经和原来slice没有任何关系，即使修改了数据也不会同步。`append`操作后，有没有生成新的slice需要看原有slice的容量是否足够，请见下面代码。

12.3 陈旧的(Stale)Slices

多个slice可以引用同一个底层数组。比如，当你从一个已有的slice创建一个新的slice时，这就会发生。如果你的应用功能需要这种行为，那么你将需要关注下“走味的”slice。

在某些情况下，在一个slice中添加新的数据，在原有数组无法保持更多新的数据时，将导致分配一个新的数组。而现在其他的slice还指向老的数组（和老的数据）。

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     s1 := []int{1, 2, 3}
7.     fmt.Println(len(s1), cap(s1), s1) // 输出 3 3 [1 2 3]
8.     s2 := s1[1:]
9.     fmt.Println(len(s2), cap(s2), s2) // 输出 2 2 [2 3]
10.    for i := range s2 {
11.        s2[i] += 20
12.    }
13.    // s2的修改会影响到数组数据，s1输出新数据
14.    fmt.Println(s1) // 输出 [1 22 23]
15.    fmt.Println(s2) // 输出 [22 23]
16.
17.    s2 = append(s2, 4) // append 导致了slice 扩容
18.
19.    for i := range s2 {
20.        s2[i] += 10
21.    }
22.    // s1 的数据现在是陈旧的老数据，而s2是新数据，他们的底层数组已经不是同一个了。
23.    fmt.Println(s1) // 输出[1 22 23]
```

```
24.         fmt.Println(s2) // 输出[32 33 14]
25.     }
```

1. 程序输出：
2. 3 3 [1 2 3]
3. 2 2 [2 3]
4. [1 22 23]
5. [22 23]
6. [1 22 23]
7. [32 33 14]

本书《Go语言四十二章经》内容在github上同步地址：<https://github.com/ffhelicopter/Go42>

本书《Go语言四十二章经》内容在简书同步地址：<https://www.jianshu.com/nb/29056963>

虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。

联系邮箱：roteman@163.com

第十三章 字典(Map)

《Go语言四十二章经》第十三章 字典(Map)

作者：李骁

13.1 字典(Map)

map 是引用类型，可以使用如下声明：

```
1. var map1 map[keytype]valuetype
2.
3. var map1 map[string]int
```

([keytype] 和 valuetype 之间允许有空格，但是 Gofmt 移除了空格)

在声明的时候不需要知道 map 的长度，map 是可以动态增长的。

未初始化的 map 的值是 nil。

key 可以是任意可以用 == 或者 != 操作符比较的类型，比如 string、int、float。所以数组、切片和结构体不能作为 key（译者注：含有数组切片的结构体不能作为 key，只包含内建类型的 struct 是可以作为 key 的），但是指针和接口类型可以。如果要用结构体作为 key 可以提供 Key() 和 Hash() 方法，这样可以通过结构体的域计算出唯一的数字或者字符串的 key。

value 可以是任意类型的；通过使用空接口类型，我们可以存储任意值，但是使用这种类型作为值时需要先做一次类型断言。map 也可以用函数作为自己的值，这样就可以用来做分支结构：key 用来选择要执行的函数。

map 传递给函数的代价很小：在 32 位机器上占 4 个字节，64 位机器上占 8 个字节，无论实际上存储了多少数据。

通过 **key** 在 **map** 中寻找值是很快的，比线性查找快得多，但是仍然比从数组和切片的索引中直接读取要慢 **100** 倍；所以如果你很在乎性能的话还是建议用切片来解决问题。

map 可以用 {key1: val1, key2: val2} 的描述方法来初始化，就像数组和结构体一样。

map 是 引用类型 的： 内存用 make 方法来分配。

map 的初始化：

```
1. var map1 = make(map[keytype]valuetype)
```

map 容量：

和数组不同，map 可以根据新增的 key-value 对动态的伸缩，因此它不存在固定长度或者最大限制。但是你也可以选择标明 map 的初始容量 capacity，就像这样：

make(map[keytype]valuetype, cap)。

例如：

```
1. map2 := make(map[string]float32, 100)
```

当 map 增长到容量上限的时候，如果再增加新的 key-value 对，map 的大小会自动加 1。所以出于性能的考虑，对于大的 map 或者会快速扩张的 map，即使只是大概知道容量，也最好先标明。

在一个 nil 的slice中添加元素是没问题的，但对一个map做同样的事将会生成一个运行时的panic。

```
1. Works:
2. package main
3. func main() {
4.     var s []int
5.     s = append(s, 1)
6. }
7.
8. Fails:
9. package main
10. func main() {
11.     var m map[string]int
12.     m["one"] = 1 // 错误
13.
14. }
```

map的key访问，val1, isPresent := map1[key1] 或者 val1 = map1[key1] 的方法获取 key1 对应的值 val1。

一般判断是否某个key存在，不使用值判断，而使用下面的方式：

```
1. if _, ok := x["two"]; !ok {
2.     fmt.Println("no entry")
3. }
```

用切片作为 map 的值

既然一个 key 只能对应一个 value，而 value 又是一个原始类型，那么如果一个 key 要对应多个值怎么办？例如，当我们要处理unix机器上的所有进程，以父进程（pid 为整形）作为 key，所有的子进程（以所有子进程的 pid 组成的切片）作为 value。通过将 value 定义为 []int 类型或者其他类型的切片，就可以优雅的解决这个问题。

这里有一些定义 map 的例子：

```
1. // 声明但未初始化map，此时是map的零值状态
2. map1 := make(map[string]string, 5)
3.
4. map2 := make(map[string]string)
5.
6. // 创建了初始化了一个空的map，这个时候没有任何元素
7. map3 := map[string]string{}
8.
9. // map中有三个值
10. map4 := map[string]string{"a": "1", "b": "2", "c": "3"}
```

从 map1 中删除 key1：

直接 delete(map1, key1) 就可以。如果 key1 不存在，该操作不会产生错误。

```
1. Delete(map4, "a")
```

map 默认是无序的，不管是按照 key 还是按照 value 默认都不排序。

如果你想为 map 排序，需要将 key（或者 value）拷贝到一个切片，再对切片排序（使用 sort 包）。

13.2 “range”语句中更新引用元素的值

在“range”语句中生成的数据的值是真实集合元素的拷贝。它们不是原有元素的引用。

这意味着更新这些值将不会修改原来的数据。同时也意味着使用这些值的地址将不会得到原有数据的指针。

```
1. package main
2. import "fmt"
3. func main() {
4.     data := []int{1, 2, 3}
5.     for _, v := range data {
6.         v *= 10 // 通常数据项不会改变
```



```
7.     }
8.     fmt.Println("data:", data) // 程序输出: [1 2 3]
9. }
```

如果你需要更新原有集合中的数据，使用索引操作符来获得数据。

```
1. package main
2. import "fmt"
3. func main() {
4.     data := []int{1, 2, 3}
5.     for i, _ := range data {
6.         data[i] *= 10
7.     }
8.
9.     fmt.Println("data:", data) // 程序输出 data: [10 20 30]
10. }
```

本书《Go语言四十二章经》内容在github上同步地址: <https://github.com/ffhelicopter/Go42>

本书《Go语言四十二章经》内容在简书同步地址: <https://www.jianshu.com/nb/29056963>

虽然本书中例子都经过实际运行，但难免出现错误和不足之处，烦请您指出；如有建议也欢迎交流。

联系邮箱: roteman@163.com

第十四章 流程控制

《Go语言四十二章经》第十四章 流程控制

作者：李骁

14.1 Switch 语句

```
1. switch var1 {  
2.     case val1:  
3.         ...  
4.     case val2:  
5.         ...  
6.     default:  
7.         ...  
8. }
```

```
1. switch {  
2.     case condition1:  
3.         ...  
4.     case condition2:  
5.         ...  
6.     default:  
7.         ...  
8. }
```

switch 语句的第二种形式是不提供任何被判断的值（实际上默认为判断是否为 true），然后在每个 case 分支中进行测试不同的条件。当任一分支的测试结果为 true 时，该分支的代码会被执行。

switch 语句的第三种形式是包含一个初始化语句：

```
1. switch initialization {  
2.     case val1:  
3.         ...  
4.     case val2:  
5.         ...  
6.     default:  
7.         ...  
8. }
```

```

1. switch result := calculate(); {
2.     case result < 0:
3.         ...
4.     case result > 0:
5.         ...
6.     default:
7.         // 0
8. }

```

变量 `var1` 可以是任何类型，而 `val1` 和 `val2` 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。前花括号 `{` 必须和 `switch` 关键字在同一行。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：`case val1, val2, val3`。一旦成功地匹配到某个分支，在执行完相应代码后就会退出整个 `switch` 代码块，也就是说您不需要特别使用 `break` 语句来表示结束。

如果在执行完每个分支的代码后，还希望继续执行后续分支的代码，可以使用 `fallthrough` 关键字来达到目的。

`fallthrough` 强制执行后面的 `case` 代码，`fallthrough` 不会判断下一条 `case` 的 `expr` 结果是否为 `true`。

```

1. package main
2.
3. import "fmt"
4.
5. func main() {
6.
7.     switch a := 1; {
8.     case a == 1:
9.         fmt.Println("The integer was == 1")
10.        fallthrough
11.     case a == 2:
12.         fmt.Println("The integer was == 2")
13.     case a == 3:
14.         fmt.Println("The integer was == 3")
15.        fallthrough
16.     case a == 4:
17.         fmt.Println("The integer was == 4")
18.     case a == 5:

```

```

19.         fmt.Println("The integer was == 5")
20.         fallthrough
21.     default:
22.         fmt.Println("default case")
23.     }
24. }

```

1. 程序输出：
2. The integer was == 1
3. The integer was == 2

14.2 Select控制

select是Go中的一个控制结构，类似于switch语句，用于处理异步IO操作。select会监听case语句中channel的读写操作，当case中channel读写操作为非阻塞状态（即能读写）时，将会触发相应的动作。

select中的case语句必须是一个channel操作

select中的default子句总是可运行的。

- 如果有多个case都可以运行，select会随机公平地选出一个执行，其他不会执行。
- 如果没有可运行的case语句，且有default语句，那么就会执行default的动作。
- 如果没有可运行的case语句，且没有default语句，select将阻塞，直到某个case通信可以运行。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {
9.     var c1, c2, c3 chan int
10.    var i1, i2 int
11.    select {
12.    case i1 = <-c1:
13.        fmt.Printf("received ", i1, " from c1\n")
14.    case c2 <- i2:
15.        fmt.Printf("sent ", i2, " to c2\n")
16.    case i3, ok := (<-c3):

```

```

17.         if ok {
18.             fmt.Printf("received ", i3, " from c3\n")
19.         } else {
20.             fmt.Printf("c3 is closed\n")
21.         }
22.         case <-time.After(time.Second * 3): //超时退出
23.             fmt.Println("request time out")
24.         }
25.     }
26.
27. // 输出:request time out

```

14.3 For循环

最简单的基于计数器的迭代，基本形式为：

```
1. for 初始化语句; 条件语句; 修饰语句 {}
```

这三部分组成的循环的头部，它们之间使用分号 ; 相隔，但并不需要括号 () 将它们括起来。

您还可以在循环中同时使用多个计数器：

```
1. for i, j := 0, N; i < j; i, j = i+1, j-1 {}
```

这得益于 Go 语言具有的平行赋值的特性，for 结构的第二种形式是没有头部的条件判断迭代（类似其它语言中的 while 循环），基本形式为：for 条件语句 {}。

您也可以认为这是没有初始化语句和修饰语句的 for 结构，因此 ;; 便是多余的了

条件语句是可以被省略的，如 i:=0; ; i++ 或 for { } 或 for ;; { } (;; 会在使用 Gofmt 时被移除)：这些循环的本质就是无限循环。

最后一个形式也可以被改写为 for true { }，但一般情况下都会直接写 for { }。

如果 for 循环的头部没有条件语句，那么就会认为条件永远为 true，因此循环体内必须有相关的条件判断以确保会在某个时刻退出循环。

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.

```

```

7. func main() {
8.     a := []int{1, 2, 3, 4, 5, 6}
9.     for i, j := 0, len(a)-1; i < j; i, j = i+1, j-1 {
10.         a[i], a[j] = a[j], a[i]
11.     }
12.
13.     for j := 0; j < 5; j++ {
14.         for i := 0; i < 10; i++ {
15.             if i > 5 {
16.                 break
17.             }
18.             fmt.Println(i)
19.         }
20.     }
21. }

```

14.4 for-range 结构

这是 Go 特有的一种的迭代结构，您会发现它在许多情况下都非常有用。它可以迭代任何一个集合（包括数组和 map），同时可以获得每次迭代所对应的索引。一般形式为：

```
1. for ix, val := range coll { }
```

要注意的是，val 始终为集合中对应索引的值拷贝，因此它一般只具有只读性质，对它所做的任何修改都不会影响到集合中原有的值（注：如果 val 为指针，则会产生指针的拷贝，依旧可以修改集合中的原值）。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. type field struct {
9.     name string
10. }
11.
12. func (p *field) print() {
13.     fmt.Println(p.name)
14. }

```

```

15.
16. func main() {
17.     data := []field{{"one"}, {"two"}, {"three"}}
18.
19.     for _, v := range data {
20.         go v.print()
21.     }
22.     time.Sleep(3 * time.Second)
23.     // goroutines (可能) 显示: three, three, three
24. }

```

当前的迭代变量作为匿名goroutine的参数。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {
9.     data := []string{"one", "two", "three"}
10.
11.     for _, v := range data {
12.         go func(in string) {
13.             fmt.Println(in)
14.         }(v)
15.     }
16.
17.     time.Sleep(3 * time.Second)
18.     // goroutines输出: one, two, three
19. }

```

一个字符串是 Unicode 编码的字符（或称之为 rune）集合，因此您也可以用它迭代字符串：

```

1. for pos, char := range str {
2.     ...
3. }

```

if

If语句由布尔表达式后紧跟一个或多个语句组成，注意布尔表达式不用()

```
1.  if 布尔表达式 {  
2.      /* 在布尔表达式为 true 时执行 */  
3.  }
```

break

一个 `break` 的作用范围为该语句出现后的最内部的结构，它可以被用于任何形式的 `for` 循环（计数器、条件判断等）。

但在 `switch` 或 `select` 语句中，`break` 语句的作用结果是跳过整个代码块，执行后续的代码。

continue

关键字 `continue` 忽略剩余的循环体而直接进入下一次循环的过程，但不是无条件执行下一次循环，执行之前依旧需要满足循环的判断条件。

关键字 `continue` 只能被用于 `for` 循环中。

label

`for`、`switch` 或 `select` 语句都可以配合标签（`label`）形式的标识符使用，即某一行第一个以冒号（`:`）结尾的单词（`Gofmt` 会将后续代码自动移至下一行）

（标签的名称是大小写敏感的，为了提升可读性，一般建议使用全部大写字母）

`continue` 语句指向 `LABEL1`，当执行到该语句的时候，就会跳转到 `LABEL1` 标签的位置。

使用标签和 `Goto` 语句是不被鼓励的：它们会很快导致非常糟糕的程序设计，而且总有更加可读的替代方案来实现相同的需求。

第十五章 错误处理

《Go语言四十二章经》第十五章 错误处理

作者：李骁

15.1 错误类型

任何时候当你需要一个新的错误类型，都可以用 `errors` (必须先 `import`) 包的 `errors.New` 函数接收合适的错误信息来创建，像下面这样：

```
1. err := errors.New("math - square root of negative number")
2. func Sqrt(f float64) (float64, error) {
3.     if f < 0 {
4.         return 0, errors.New("math - square root of negative number")
5.     }
6. }
```

用 `fmt` 创建错误对象：

通常你想要返回包含错误参数的更有信息量的字符串，例如：可以用 `fmt.Errorf()` 来实现：它和 `fmt.Printf()` 完全一样，接收有一个或多个格式占位符的格式化字符串和相应数量的占位变量。和打印信息不同的是它用信息生成错误对象。

比如在前面的平方根例子中使用：

```
1. if f < 0 {
2.     return 0, fmt.Errorf("square root of negative number %g", f)
3. }
```

15.2 Panic

在多层嵌套的函数调用中调用 `panic`，可以马上中止当前函数的执行，所有的 `defer` 语句都会保证执行并把控制权交还给接收到 `panic` 的函数调用者。这样向上冒泡直到最顶层，并执行（每层的）`defer`，在栈顶处程序崩溃，并在命令行中用传给 `panic` 的值报告错误情况：这个终止过程就是 `panicking`。

标准库中有许多包含 `Must` 前缀的函数，像 `regexp.MustCompile` 和 `template.Must`；当正则表达式或模板中转入的转换字符串导致错误时，这些函数会 `panic`。

不能随意地用 `panic` 中止程序，必须尽力补救错误让程序能继续执行。

自定义包中的错误处理和 `panicking`，这是所有自定义包实现者应该遵守的最佳实践：

- 1) 在包内部，总是应该从 `panic` 中 `recover`：不允许显式的超出包范围的 `panic()`
- 2) 向包的调用者返回错误值（而不是 `panic`）。

`recover()` 的调用仅当它在 `defer` 函数中被直接调用时才有效。

下面主函数 `recover` 了 `panic`：

```

1. func Parse(input string) (numbers []int, err error) {
2.     defer func() {
3.         if r := recover(); r != nil {
4.             var ok bool
5.             err, ok = r.(error)
6.             if !ok {
7.                 err = fmt.Errorf("pkg: %v", r)
8.             }
9.         }
10.    }()
11.
12.    fields := strings.Fields(input)
13.    numbers = fields2numbers(fields)
14.    return
15. }
16.
17. func fields2numbers(fields []string) (numbers []int) {
18.     if len(fields) == 0 {
19.         panic("no words to parse")
20.     }
21.     for idx, field := range fields {
22.         num, err := strconv.Atoi(field)
23.         if err != nil {
24.             panic(&ParseError{idx, field, err})
25.         }
26.         numbers = append(numbers, num)
27.     }
28.     return
29. }
```

15.3 Recover：从 `panic` 中恢复

正如名字一样，这个 (recover) 内建函数被用于从 panic 或 错误场景中恢复：让程序可以从 panicking 重新获得控制权，停止终止过程进而恢复正常执行。

recover 只能在 defer 修饰的函数中使用：用于取得 panic 调用中传递过来的错误值，如果是正常执行，调用 recover 会返回 nil，且没有其它效果。

总结：panic 会导致栈被展开直到 defer 修饰的 recover() 被调用或者程序中止。

```

1. func protect(g func()) {
2.     defer func() {
3.         log.Println("done")
4.         // 即使有panic, Println也正常执行。
5.         if err := recover(); err != nil {
6.             log.Printf("run time panic: %v", err)
7.         }
8.     }()
9.     log.Println("start")
10.    g() // 可能发生运行时错误的地方
11. }

```

15.4 有关于defer

说到错误处理，就不得不提defer。先说说它的规则：

- 规则一 当defer被声明时，其参数就会被实时解析
- 规则二 defer执行顺序为先进后出
- 规则三 defer可以读取有名返回值，也就是可以改变有名返回参数的值。

必须要先声明defer，否则不能捕获到panic异常。recover() 的调用仅当它在 defer 函数中被直接调用时才有效。

panic 是用来表示非常严重的不可恢复的错误的。在Go语言中这是一个内置函数，接收一个 interface{}类型的值（也就是任何值了）作为参数。

函数执行的时候panic了，函数不往下走，开始运行defer，defer处理完再返回。这时候（defer的时候），recover内置函数可以捕获到当前的panic（如果有的话），被捕获到的panic就不会向上传递了。

recover之后，逻辑并不会恢复到panic那个点去，函数还是会在defer之后返回。

大致过程：

Panic-->defer-->recover

```

1. // 规则一，当defer被声明时，其参数就会被实时解析
2. package main
3.
4. import "fmt"
5.
6. func main() {
7.     var i int = 1
8.
9.     defer fmt.Println("result =>", func() int { return i * 2 }())
10.    i++
11.    // 输出: result => 2 (而不是 4)
12. }

```

```

1. // 规则二 defer执行顺序为先进后出
2.
3. package main
4.
5. import "fmt"
6.
7. func main() {
8.
9.     defer fmt.Print(" !!! ")
10.    defer fmt.Print(" world ")
11.    fmt.Print(" hello ")
12.
13. }
14. //输出:  hello  world  !!!

```

上面讲了两条规则，第三条规则其实也不难理解，只要记住是可以改变有名返回值：

这是由于在Go语言中，return 语句不是原子操作，最先是所有结果值在进入函数时都会初始化为其类型的零值（姑且称为ret赋值），然后执行defer命令，最后才是return操作。如果是有名返回值，返回值变量其实可视为是引用赋值，可以被defer修改。而在匿名返回值时，给ret的值相当于拷贝赋值，defer命令时不能直接修改。

```

1. func fun1() (i int)

```

上面函数签名中的 i 就是有名返回值，如果fun1()中定义了 defer 代码块，是可以改变返回值 i 的，函数返回语句return i 可以简写为 return 。

这里综合了一下，在下面这个例子里列举了几种情况，可以好好琢磨下；

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. func main() {
8.     fmt.Println("=====")
9.     fmt.Println("return:", fun1())
10.
11.    fmt.Println("=====")
12.    fmt.Println("return:", fun2())
13.    fmt.Println("=====")
14.
15.    fmt.Println("return:", fun3())
16.    fmt.Println("=====")
17.
18.    fmt.Println("return:", fun4())
19. }
20.
21. func fun1() (i int) {
22.     defer func() {
23.         i++
24.         fmt.Println("defer2:", i) // 打印结果为 defer: 2
25.     }()
26.
27.     // 规则二 defer执行顺序为先进后出
28.
29.     defer func() {
30.         i++
31.         fmt.Println("defer1:", i) // 打印结果为 defer: 1
32.     }()
33.
34.     // 规则三 defer可以读取有名返回值（函数指定了返回参数名）
35.     return 0 //实际为2 。 换句话说其实怎么写都是直接 return 的效果
36. }
37.
38. func fun2() int {
39.     var i int
40.     defer func() {
41.         i++
42.         fmt.Println("defer2:", i) // 打印结果为 defer: 2

```

```

43.     }()
44.
45.     defer func() {
46.         i++
47.         fmt.Println("defer1:", i) // 打印结果为 defer: 1
48.     }()
49.     return i
50. }
51.
52. func fun3() (r int) {
53.     t := 5
54.     defer func() {
55.         t = t + 5
56.         fmt.Println(t)
57.     }()
58.     return t
59. }
60.
61. func fun4() int {
62.     i := 8
63.     // 规则一 当defer被声明时, 其参数就会被实时解析
64.     defer func(i int) {
65.         i = 99
66.         fmt.Println(i)
67.     }(i)
68.     i = 19
69.     return i
70. }

```

下面是输出, 在有名返回值情况下, return语句怎么写都改变不了最终返回的实际值, 在上面fun1() (i int) 中, return 100和return 0 没有任何作用, 返回的还是i的实际值, 所以我们一般直接写为return。这点要注意, 有时函数可能返回非我们希望的, 所以改为匿名返回也是一种办法。

```

1. 程序输出 :
2. =====
3. defer1: 1
4. defer2: 2
5. return: 2
6. =====
7. defer1: 1
8. defer2: 2
9. return: 0

```

```

10.  =====
11.  10
12.  return: 5
13.  =====
14.  99
15.  return: 19

```

使用defer计算函数执行时间

```

1.  package main
2.  import(
3.      "fmt"
4.      "time"
5.  )
6.
7.  func main(){
8.      defer timeCost(time.Now())
9.      fmt.Println("start program")
10.     time.Sleep(5*time.Second)
11.     fmt.Println("finish program")
12. }
13.
14. func timeCost(start time.Time){
15.     terminal:=time.Since(start)
16.     fmt.Println(terminal)
17. }

```

另外一种计算函数执行时间方法：

在对比和基准测试中，我们需要知道一个计算执行消耗的时间。最简单的一个办法就是在计算开始之前设置一个起始时候，再由计算结束时的结束时间，最后取出它们的差值，就是这个计算所消耗的时间。想要实现这样的做法，可以使用 `time` 包中的 `Now()` 和 `Sub` 函数：

```

1.  start := time.Now()
2.  longCalculation()
3.  end := time.Now()
4.  delta := end.Sub(start)
5.  fmt.Printf("longCalculation took this amount of time: %s\n", delta)

```

第十六章 函数

《Go语言四十二章经》第十六章 函数

作者：李骁

16.1 函数分类

Go 里面有三种类型的函数：

- 普通的带有名字的函数
- 匿名函数或者lambda函数
- 方法 (Methods)

除了main()、init()函数外，其它所有类型的函数都可以有参数与返回值。

函数参数、返回值以及它们的类型被统称为函数签名。

函数重载 (function overloading) 指的是可以编写多个同名函数，只要它们拥有不同的形参或者不同的返回值，在 Go 里面函数重载是不被允许的。

如果需要申明一个在外部定义的函数，你只需要给出函数名与函数签名，不需要给出函数体：

```
1. func flushICache(begin, end uintptr)
```

函数也可以以申明的方式被使用，作为一个函数类型，就像：

```
1. type binOp func(int, int) int
```

在这里，不需要函数体 {}。

函数是一等值 (first-class value)：它们可以赋值给变量，就像下面一样：

```
1. add := binOp
```

这个变量知道自己指向的函数的签名，所以给它赋一个具有不同签名的函数值是不可能的。

函数值 (functions value) 之间可以相互比较：如果它们引用的是相同的函数或者都是 nil 的话，则认为它们是相同的函数。函数不能在其它函数里面声明（不能嵌套），不过我们可以通过使用匿名函数来破除这个限制。

没有参数的函数通常被称为 无参数函数 (niladic function)，就像 `main.main()`

16.2 函数调用

- 按值传递 (call by value)
- 按引用传递 (call by reference)

Go 默认使用按值传递来传递参数，也就是传递参数的副本。函数接收参数副本之后，在使用变量的过程中可能对副本的值进行更改，但不会影响到原来的变量，比如 `Function(arg1)`。

如果你希望函数可以直接修改参数的值，而不是对参数的副本进行操作，你需要将参数的地址（变量名前面添加&符号，比如 `&variable`）传递给函数，这就是按引用传递，比如 `Function(&arg1)`，此时传递给函数的是一个指针。如果传递给函数的是一个指针，指针的值（一个地址）会被复制，但指针的值所指向的地址上的值不会被复制；我们可以通过这个指针的值来修改这个值所指向的地址上的值。

在函数调用时，像切片 (slice)、字典 (map)、接口 (interface)、通道 (channel) 这样的引用类型都是默认使用引用传递（即使没有显式的指出指针）

命名返回值作为结果形参 (result parameters) 被初始化为相应类型的零值，当需要返回的时候，我们只需要一条简单的不带参数的 `return` 语句。需要注意的是，即使只有一个命名返回值，也需要使用 `()` 括起来

如果函数的最后一个参数是采用 `...type` 的形式，那么这个函数就可以处理一个变长的参数，这个长度可以为 0，这样的函数称为变参函数。

这个函数接受一个类似某个类型的 slice 的参数，该参数可以通过 `for` 循环结构迭代。

```
1. func min(s ...int) int {
2.     if len(s)==0 {
3.         return 0
4.     }
5.     min := s[0]
6.     for _, v := range s {
7.         if v < min {
8.             min = v
9.         }
10.    }
11.    return min
12. }
```

16.3 内置函数

Go 语言拥有一些不需要进行导入操作就可以使用的内置函数。它们有时可以针对不同的类型进行操作：

名称	说明
close	用于通道通信
len、cap	len 用于返回某个类型的长度或数量（字符串、数组、切片、map 和通道）；cap 是容量的意思，用于返回某个类型的最大容量（只能用于切片和 map）
new、make	new 和 make 均是用于分配内存：new 用于值类型和用户定义的类型，如自定义结构，make 用于内置引用类型（切片、map 和通道）。它们的用法就像是函数，但是将类型作为参数：new(type)、make(type)。new(T) 分配类型 T 的零值并返回其地址，也就是指向类型 T 的指针。它也可以被用于基本类型：v := new(int)。make(T) 返回类型 T 的初始化之后的值，因此它比 new 进行更多的工作。new() 是一个函数，不要忘记它的括号。二者都是内存的分配（堆上），但是make只用于slice、map以及channel的初始化（非零值）；而new用于类型的内存分配，并且内存置为零。
copy、append	用于复制和连接切片
panic、recover	两者均用于错误处理机制

16.4 递归与回调

使用递归函数时经常会遇到的一个重要问题就是栈溢出：一般出现在大量的递归调用导致的程序栈内存分配耗尽。这个问题可以通过一个名为懒惰求值的技术解决，在 Go 语言中，我们可以使用通道（channel）和 goroutine。

Go 语言中也可以使用相互调用的递归函数：多个函数之间相互调用形成闭环。因为 Go 语言编译器的特殊性，这些函数的声明顺序可以是任意的。

函数可以作为其它函数的参数进行传递，然后在其它函数内调用执行，一般称之为回调。

```

1. package main
2. import (
3.     "fmt"
4. )
5. func main() {
6.     callback(1, Add)
7. }
8. func Add(a, b int) {
9.     fmt.Printf("The sum of %d and %d is: %d\n", a, b, a+b)
10. }
11. func callback(y int, f func(int, int)) {
12.     f(y, 2) // 实际上是 Add(1, 2)
13. }
```

16.5 匿名函数

当我们不希望给函数起名字的时候，可以使用匿名函数，例如：

```
1. func(x, y int) int { return x + y }
```

这样的函数不能够独立存在（编译器会返回错误：non-declaration statement outside function body），但可以被赋值于某个变量，即保存函数的地址到变量中：fplus := func(x, y int) int { return x + y }，然后通过变量名对函数进行调用：fplus(3, 4)。

当然，也可以直接对匿名函数进行调用：

```
1. func(x, y int) int { return x + y } (3, 4)
```

下面是一个计算从 1 到 1 百万整数的总和的匿名函数：

```
1. func() {  
2.     sum := 0  
3.     for i := 1; i <= 1e6; i++ {  
4.         sum += i  
5.     }  
6. }()
```

表示参数列表的第一对括号必须紧挨着关键字 func，因为匿名函数没有名称。花括号 {} 涵盖着函数体，最后的一对括号表示对该匿名函数的调用。

16.6 闭包函数

匿名函数同样被称之为闭包（函数式语言的术语）：它们被允许调用定义在其它环境下的变量。闭包可使得某个函数捕捉到一些外部状态，例如：函数被创建时的状态。另一种表示方式为：一个闭包继承了函数所声明时的作用域。这种状态（作用域内的变量）都被共享到闭包的环境中，因此这些变量可以在闭包中被操作，直到被销毁。闭包经常被用作包装函数：它们会预先定义好 1 个或多个参数以用于包装。另一个不错的应用就是使用闭包来完成更加简洁的错误检查。

仅仅从形式上将闭包简单理解为匿名函数是不够的，还需要理解闭包实质上的含义。

实质上看，闭包是由函数及其相关引用环境组合而成的实体（即：闭包=函数+引用环境）。闭包在运行时可以有多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。由闭包的实质含义，我们可以推论：闭包获取捕获变量相当于引用传递，而非值传递；对于闭包函数捕获的常量和变量，无论闭包何时何处被调用，闭包都可以使用这些常量和变量，而不用关心它们表面上的作用域。

应用闭包：将函数作为返回值，我们用一个例子来进行验证。

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. func addNumber(x int) func(int) {
8.     fmt.Printf("x: %d, addr of x:%p\n", x, &x)
9.     return func(y int) {
10.         k := x + y
11.         x = k
12.         y = k
13.         fmt.Printf("x: %d, addr of x:%p\n", x, &x)
14.         fmt.Printf("y: %d, addr of y:%p\n", y, &y)
15.     }
16. }
17.
18. func main() {
19.     addNum := addNumber(5)
20.     addNum(1)
21.     addNum(1)
22.     addNum(1)
23.
24.     fmt.Println("-----")
25.
26.     addNum1 := addNumber(5)
27.     addNum1(1)
28.     addNum1(1)
29.     addNum1(1)
30. }

```

```

1. 程序输出：
2. x: 5, addr of x:0xc042054080
3. x: 6, addr of x:0xc042054080
4. y: 6, addr of y:0xc042054098
5. x: 7, addr of x:0xc042054080
6. y: 7, addr of y:0xc0420540d0
7. x: 8, addr of x:0xc042054080
8. y: 8, addr of y:0xc0420540e8
9. -----

```

```

10. x: 5, addr of x:0xc042054100
11. x: 6, addr of x:0xc042054100
12. y: 6, addr of y:0xc042054110
13. x: 7, addr of x:0xc042054100
14. y: 7, addr of y:0xc042054128
15. x: 8, addr of x:0xc042054100
16. y: 8, addr of y:0xc042054140

```

首先强调一点，x是闭包中被捕获的变量，y只是闭包内部的局部变量，而非被捕获的变量。因此，对于每一次引用，x的地址都是固定的，是同一个引用变量；y的地址则是变化的。另外，闭包被引用了两次，由此产生了两个闭包实例，即`addNum := addNumber(5)`和`addNum1 := addNumber(5)`是两个不同实例，其中引用的两个x变量也来自两个不同的实例。

16.7 使用闭包调试

当您在分析和调试复杂的程序时，无数个函数在不同的代码文件中相互调用，如果这时候能够准确地知道哪个文件中的具体哪个函数正在执行，对于调试是十分有帮助的。您可以使用 `runtime` 或 `log` 包中的特殊函数来实现这样的功能。包 `runtime` 中的函数 `Caller()` 提供了相应的信息，因此可以在需要的时候实现一个 `where()` 闭包函数来打印函数执行的位置：

```

1. where := func() {
2.     _, file, line, _ := runtime.Caller(1)
3.     log.Printf("%s:%d", file, line)
4. }
5. where()
6. // some code
7. where()
8. // some more code
9. where()

```

或使用一个更加简短版本的 `where` 函数：

```

1. var where = log.Print
2. func func1() {
3.     where()
4.     ... some code
5.     where()
6.     ... some code
7.     where()
8. }

```

16.8 高阶函数

在定义所需功能时我们可以利用函数可以作为（其它函数的）参数的事实来使用高阶函数

定义一个通用的 `Process()` 函数，它接收一个作用于每一辆 `car` 的 `f` 函数作参数：

```
1. // Process all cars with the given function f:
2. func (cs Cars) Process(f func(car *Car)) {
3.     for _, c := range cs {
4.         f(c)
5.     }
6. }
```

第十七章 Type关键字

《Go语言四十二章经》第十七章 Type关键字

作者：李骁

17.1 Type

在Go 语言中，基础类型有下面几种：

```
1.      bool byte complex64 complex128 error float32 float64
2.      int  int8 int16 int32 int64 rune  string
3.      uint uint8 uint16 uint32 uint64 uintptr
```

使用 `type` 关键字可以定义你自己的类型，你可能想要定义一个结构体，但是也可以给一个已经存在的类型的新的名字，然后你就可以在你的代码中使用新的名字（用于简化名称或解决名称冲突），称为自定义类型，如：

```
1.  type IZ int
```

然后我们可以使用下面的方式声明变量：

```
1.  var a IZ = 5
```

这里我们可以看到 `int` 是变量 `a` 的底层类型，这也使得它们之间存在相互转换的可能。

如果你有多个类型需要定义，可以使用因式分解关键字的方式，例如：

```
1.  type (
2.      IZ int
3.      FZ float64
4.      STR string
5.  )
```

在 `type TZ int` 中，`TZ` 就是 `int` 类型的新名称（用于表示程序中的时区），称为自定义类型，然后就可以使用 `TZ` 来操作 `int` 类型的数据。使用这种方法定义之后的类型可以拥有更多的特性，且在类型转换时必须显式转换。

每个值都必须在经过编译后属于某个类型（编译器必须能够推断出所有值的类型），因为 Go 语言是一

种静态类型语言。

在必要以及可行的情况下，一个类型的值可以被转换成另一种类型的值。由于 Go 语言不存在隐式类型转换，因此所有的转换都必须显式说明，就像调用一个函数一样（类型在这里的作用可以看作是一种函数）：

```
1. valueOfTypeB = typeB(valueOfTypeA)
```

类型 B 的值 = 类型 B(类型 A 的值)

type TZ int 中，新类型不会拥有原类型所附带的方法；TZ 可以自定义一个方法用来输出更加人性化的时区信息。

```
1. type TZ = int
```

（这种写法应该才是真正的别名，type TZ int 其实是定义了新类型，这两种完全不是一个含义。自定义类型不会拥有原类型附带的方法，而别名是可以的。）类型别名在1.9中实现，可将别名类型和原类型这两个类型视为完全一致使用。下面举2个例子：

新类型不会拥有原类型所附带的方法：

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type A struct {
8.     Face int
9. }
10. type Aa A
11.
12. func (a A) f() {
13.     fmt.Println("hi ", a.Face)
14. }
15.
16. func main() {
17.     var s A = A{Face: 9}
18.     s.f()
19.
20.     var sa Aa = Aa{Face: 9}
21.     sa.f()
```



```
22. }
```

1. 编译错误信息: `sa.f undefined (type Aa has no field or method f)`

但如果是类型别名，完整拥有其方法：

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type A struct {
8.     Face int
9. }
10. type Aa=A
11.
12. func (a A) f() {
13.     fmt.Println("hi ", a.Face)
14. }
15.
16. func main() {
17.     var s A = A{Face: 9}
18.     s.f()
19.
20.     var sa Aa = Aa{Face: 9}
21.     sa.f()
22. }
```

1. 程序输出：

2. hi 9

3. hi 9

类型可以是基本类型，如：int、float、bool、string；

结构化的（复合的），如：struct、array、slice、map、channel；

只描述类型的行为的，如：interface。

结构化的类型没有真正的值，它使用 nil 作为默认值（在 Objective-C 中是 nil，在 Java 中是 null，在 C 和 C++ 中是 NULL 或 0）。值得注意的是，Go 语言中不存在类型继承。

函数也可以是一个确定的类型，就是以函数作为返回类型。这种类型的声明要写在函数名和可选的参数

列表之后，例如：

```
1. func FunctionName (a typea, b typeb) typeFunc
```

你可以在函数体中的某处返回使用类型为 `typeFunc` 的变量 `var`：

```
1. return var
```

一个函数可以拥有多个返回值，返回类型之间需要使用逗号分割，并使用小括号 `()` 将它们括起来，如：

```
1. func FunctionName (a typea, b typeb) (t1 type1, t2 type2)
```

自定义类型不会继承原有类型的方法，但接口方法或组合类型的元素则保留原有的方法。

```
1. // Mutex 用两种方法, Lock and Unlock。
2. type Mutex struct      { /* Mutex fields */ }
3. func (m *Mutex) Lock() { /* Lock implementation */ }
4. func (m *Mutex) Unlock() { /* Unlock implementation */ }
5.
6. // NewMutex和 Mutex 一样的数据结构，但是其方法是空的。
7. type NewMutex Mutex
8.
9. // PtrMutex 的方法也是空的
10. type PtrMutex *Mutex
11.
12. // *PrintableMutex 拥有Lock and Unlock 方法
13. type PrintableMutex struct {
14.     Mutex
15. }
```

第十八章 Struct 结构体

《Go语言四十二章经》第十八章 Struct 结构体

作者：李骁

18.1 结构体(struct)

Go 通过结构体的形式支持用户自定义类型，或者叫定制类型。

一个带属性的结构体试图表示一个现实世界中的实体。

结构体是复合类型 (*composite types*)，当需要定义一个类型，它由一系列属性组成，每个属性都有自己的类型和值的时候，就应该使用结构体，它把数据聚集在一起。

然后（方法）可以访问这些数据，就好像它们是一个独立实体的一部分。

结构体是值类型，因此可以通过 `new` 函数来创建。

组成结构体类型的那些数据称为字段 (*fields*)。每个字段都有一个类型和一个名字；在一个结构体中，字段名字必须是唯一的。

结构体定义的一般方式如下：

```
1. type identifier struct {  
2.     field1 type1  
3.     field2 type2  
4.     ...  
5. }
```

结构体里的字段都有 名字，像 `field1`、`field2` 等，如果字段在代码中从来也不会被用到，那么可以命名它为 `_`。

使用 `new`

使用 `new` 函数给一个新的结构体变量分配内存，它返回指向已分配内存的指针：`var t *T = new(T)`，如果需要可以把这条语句放在不同的行（比如定义是包范围的，但是分配却没有必要在开始就做）。

```
1. var t *T  
2. t = new(T)
```

写这条语句的惯用方法是：`t := new(T)`，变量 `t` 是一个指向 `T` 的指针，此时结构体字段的值是它们所属类型的零值。

声明 `var t T` 也会给 `t` 分配内存，并零值化内存，但是这个时候 `t` 是类型 `T`。在这两种方式中，`t` 通常被称做类型 `T` 的一个实例 (instance) 或对象 (object)。

同样的，使用点号符可以获取结构体字段的值：`structname.fieldname`。

在 Go 语言中这叫 选择器 (selector)。无论变量是一个结构体类型还是一个结构体类型指针，都使用同样的 选择器符 (selector-notation) 来引用结构体的字段：

```
1. type myStruct struct { i int }
2. var v myStruct      // v是结构体类型变量
3. var p *myStruct     // p是指向一个结构体类型变量的指针
4. v.i
5. p.i
6.
7. type Interval struct {
8.     start int
9.     end   int
10. }
```

初始化方式：

```
1. intr := Interval{0, 3}           (A)
2. intr := Interval{end:5, start:1} (B)
3. intr := Interval{end:5}          (C)
```

初始化一个结构体实例 (一个结构体字面量: struct-literal) 的更简短和惯用的方式如下：

```
1. ms := &struct1{10, 15.5, "Chris"}
2. // 此时ms的类型是 *struct1
```

或者：

```
1. var ms struct1
2. ms = struct1{10, 15.5, "Chris"}
```

混合字面量语法 (composite literal syntax)

`&struct1{a, b, c}` 是一种简写，底层仍然会调用 `new()`，这里值的顺序必须按照字段顺序来写。在下面的例子中能看到可以通过在值的前面放上字段名来初始化字段的方式。

表达式 `new(Type)` 和 `&Type{}` 是等价的。

结构体类型和字段的命名遵循可见性规则，一个导出的结构体类型中有些字段是导出的，另一些不可见。

18.2 结构体特性

- 结构体的内存布局

Go 语言中，结构体和它所包含的数据在内存中是以连续块的形式存在的，即使结构体中嵌套有其他的结构体，这在性能上带来了很大的优势。

- 递归结构体

结构体类型可以通过引用自身来定义。这在定义链表或二叉树的元素（通常叫节点）时特别有用，此时节点包含指向临近节点的链接（地址）。如下所示，链表中的 `su`，树中的 `ri` 和 `le` 分别是指向别的节点的指针。

- 链表

这块的 `data` 字段用于存放有效数据（比如 `float64`），`su` 指针指向后继节点。

Go 代码：

```
1. type Node struct {
2.     data    float64
3.     su      *Node
4. }
```

链表中的第一个元素叫 `head`，它指向第二个元素；最后一个元素叫 `tail`，它没有后继元素，所以它的 `su` 为 `nil` 值。当然真实的链接会有很多数据节点，并且链表可以动态增长或收缩。

同样地可以定义一个双向链表，它有一个前趋节点 `pr` 和一个后继节点 `su`：

```
1. type Node struct {
2.     pr      *Node
3.     data    float64
4.     su      *Node
5. }
```

- 二叉树

二叉树中每个节点最多能链接至两个节点：左节点（`le`）和右节点（`ri`），这两个节点本身又可以有左右节点，依次类推。树的顶层节点叫根节点（`root`），底层没有子节点的节点叫叶子节点（`leaves`），叶子节点的 `le` 和 `ri` 指针为 `nil` 值。在 Go 中可以如下定义二叉树：

```

1. type Tree struct {
2.     le      *Tree
3.     data    float64
4.     ri      *Tree
5. }

```

- 结构体工厂

Go 语言不支持面向对象编程语言中那样的构造子方法，但是可以很容易的在 Go 中实现 “构造子工厂”方法。为了方便通常会为类型定义一个工厂，按惯例，工厂的名字以 `new` 或 `New` 开头。假设定义了如下的 `File` 结构体类型：

```

1. type File struct {
2.     fd      int      // 文件描述符
3.     name    string   // 文件名
4. }

```

下面是这个结构体类型对应的工厂方法，它返回一个指向结构体实例的指针：

```

1. func NewFile(fd int, name string) *File {
2.     if fd < 0 {
3.         return nil
4.     }
5.
6.     return &File{fd, name}
7. }

```

然后这样调用它：

```

1. f := NewFile(10, "./test.txt")

```

在 Go 语言中常常像上面这样在工厂方法里使用初始化来简便的实现构造函数。

如果 `File` 是一个结构体类型，那么表达式 `new(File)` 和 `&File{}` 是等价的。

这可以和大多数面向对象编程语言中笨拙的初始化方式做个比较：`File f = new File(...)`。

我们可以说是工厂实例化了类型的一个对象，就像在基于类的OO语言中那样。

如果想知道结构体类型T的一个实例占用了多少内存，可以使用：`size := unsafe.Sizeof(T{})`。

- 如何强制使用工厂方法

通过应用可见性规则参考，就可以禁止使用 `new` 函数，强制用户使用工厂方法，从而使类型变成私有的，就像在面向对象语言中那样。

```

1. type matrix struct {
2.     ...
3. }
4.
5. func NewMatrix(params) *matrix {
6.     m := new(matrix) // 初始化 m
7.     return m
8. }

```

在包外，只有通过NewMatrix函数才可以初始化matrix 结构。

- 带标签的结构体

结构体中的字段除了有名字和类型外，还可以有一个可选的标签（tag）：它是一个附属于字段的字符串，可以是文档或其他的重要标记。标签的内容不可以在一般的编程中使用，只有包 reflect 能获取它。reflect包可以在运行时自省类型、属性和方法，比如：在一个变量上调用 reflect.TypeOf() 可以获取变量的正确类型，如果变量是一个结构体类型，就可以通过 Field 来索引结构体的字段，然后就可以使用 Tag 属性。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "reflect"
6. )
7.
8. type TagType struct { // 结构体标签
9.     field1 bool    "An important answer"
10.    field2 string "The name of the thing"
11.    field3 int     "How much there are"
12. }
13.
14. func main() {
15.     tt := TagType{true, "Barak Obama", 1}
16.     for i := 0; i < 3; i++ {
17.         refTag(tt, i)
18.     }
19. }
20.
21. func refTag(tt TagType, ix int) {
22.     ttType := reflect.TypeOf(tt)
23.     ixField := ttType.Field(ix)

```

```

24.         fmt.Printf("%v\n", ixField.Tag)
25.     }

```

1. 程序输出：
- 2.
3. An important answer
4. The name of the thing
5. How much there are

18.3 匿名成员

Go语言有一个特性让我们只声明一个成员对应的数据类型而不指名成员的名字；这类成员就叫匿名成员。匿名成员的数据类型必须是命名的类型或指向一个命名的类型的指针。

结构体可以包含一个或多个 匿名（或内嵌）字段，即这些字段没有显式的名字，只有字段的类型是必须的，此时类型就是字段的名字（这决定了在一个结构体中对于每一种数据类型只能有一个匿名字段。）。匿名字段本身可以是一个结构体类型，即 结构体可以包含内嵌结构体。

```

1. type Base struct {
2.     basename string
3. }
4.
5. type Derive struct { // 含内嵌结构体
6.     Base    // 匿名
7.     int
8. }

```

可以粗略地将这个和面向对象语言中的继承概念相比较，随后将会看到它被用来模拟类似继承的行为。Go 语言中的继承是通过内嵌或组合来实现的，所以可以说，在 Go 语言中，相比较于继承，组合更受青睐。

18.4 内嵌(embedded)结构体

内嵌与聚合：

外部类型只包含了内部类型的类型名， 而没有field 名， 则是内嵌。外部类型包含了内部类型的类型名，还有field名，则是聚合。聚合的在JAVA和C++都是常见的方式。而内嵌则是Go 的特有方式。

```

1. type Base struct {
2.     basename string
3. }

```



```

4.
5.  type Derive struct {           // 内嵌
6.     Base
7. }
8.
9.  type Derive struct {           // 内嵌， 这种内嵌与上面内嵌有差异
10.     *Base
11. }
12.
13. type Derive struct{            // 聚合
14.     base Base
15. }

```

内嵌的方式：

主要是通过结构体和接口的组合，有四种。

- 接口中内嵌接口：

这里的做为内嵌接口的含义实际上还是指的一个定义，而不是接口的一个实例，相当于合并了两个接口定义的函数，只有同时了Writer和 Reader 接口，是可以说是实现了WRer接口，即才可以作为WRer的实例。

```

1.  type Writer interface{
2.     Write()
3. }
4.
5.  type Reader interface{
6.     Read()
7. }
8.
9.  type WRer interface{
10.     Reader
11.     Writer
12. }

```

- 在接口中内嵌struct：

存在语法错误，并不具有实际的含义， 编译报错：

```

1.  interface contains embedded non-interface Person
2.
3.  Interface 不能嵌入非interface的类型。

```

- 在结构体 (struct) 中内嵌 接口 (interface)

- 1, 初始化的时候, 内嵌接口要用一个实现此接口的结构体赋值。
- 2, 外层结构体中, 只能调用内层接口定义的函数。 这是由于编译时决定。
- 3, 外层结构体, 可以作为receiver, 重新定义同名函数, 这样可以覆盖内层内嵌结构中定义的函数。
- 4, 如果上述第3条实现, 那么可以用外层结构体引用内嵌接口的实例, 并调用内嵌接口的函数。

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type Printer interface {
8.     Print()
9. }
10.
11. type CanonPrinter struct {
12.     Printname string
13. }
14.
15. func (printer CanonPrinter) Print() {
16.     fmt.Println("this is cannoprinter printing now ")
17. }
18.
19. type PrintWorker struct {
20.     Printer
21.     name string
22.     age  int
23. }
24.
25. // 如果没有下面实现, 则
26. func (printworker PrintWorker) Print() {
27.     fmt.Println("this is printing from PrintWorker ")
28.     printworker.Printer.Print()
29.     // 这里 printworker 首先引用内部嵌入Printer接口的实例,
30.     // 然后调用Printer 接口实例的Print()方法
31. }
32.
33. func main() {

```

```

34.     canon := CanonPrinter{"canoprint_num_1"}
35.     printworker := PrintWorker{Printer: canon, name: "ansendong", age: 34}
36.     printworker.Print()
37.     // 如果没有上述部分Func (printworker PrintWorker) Print()的实现,
38.     // 则这里只调用CanonPrinter实现的Print()方法。
39. }

```

- 结构体 (struct) 中内嵌 结构体 (struct)

1, 初始化, 内嵌结构体要进行赋值。

2, 外层结构自动获得内嵌结构体所有定义的field和实现的方法 (method)。

3, 同上述结构体中内嵌接口类似, 同样外层结构体可以定义同名方法, 这样覆盖内层结构体的定义的方法。 同样也可以定义同名变量, 覆盖内层结构体的变量。

4, 同样可以内层结构体引用, 内层结构体已经定义的方法和变量。

同样地结构体也是一种数据类型, 所以它也可以作为一个匿名字段来使用, 如同下面例子中那样。外层结构体通过 `outer.in1` 直接进入内层结构体的字段, 内嵌结构体甚至可以来自其他包。内层结构体被简单的插入或者内嵌进外层结构体。这个简单的“继承”机制提供了一种方式, 使得可以从另外一个或一些类型继承部分或全部实现。

```

1.  package main
2.
3.  import "fmt"
4.
5.  type innerS struct {
6.      in1 int
7.      in2 int
8.  }
9.
10. type outerS struct {
11.     b    int
12.     c    float32
13.     int  // anonymous field
14.     innerS //anonymous field
15. }
16.
17. func main() {
18.     outer := new(outerS)
19.     outer.b = 6
20.     outer.c = 7.5

```

```

21.     outer.int = 60
22.     outer.in1 = 5
23.     outer.in2 = 10
24.
25.     fmt.Printf("outer.b is: %d\n", outer.b)
26.     fmt.Printf("outer.c is: %f\n", outer.c)
27.     fmt.Printf("outer.int is: %d\n", outer.int)
28.     fmt.Printf("outer.in1 is: %d\n", outer.in1)
29.     fmt.Printf("outer.in2 is: %d\n", outer.in2)
30.
31.     // 使用结构体字面量
32.     outer2 := outerS{6, 7.5, 60, innerS{5, 10}}
33.     fmt.Println("outer2 is:", outer2)
34. }

```

```

1.  程序输出：
2.
3.  outer.b is: 6
4.  outer.c is: 7.500000
5.  outer.int is: 60
6.  outer.in1 is: 5
7.  outer.in2 is: 10
8.  outer2 is:{6 7.5 60 {5 10}}

```

18.5 命名冲突

当两个字段拥有相同的名字（可能是继承来的名字）时该怎么办呢？

外层名字会覆盖内层名字（但是两者的内存空间都保留），这提供了一种重载字段或方法的方式；如果相同的名字在同一级别出现了两次，如果这个名字被程序使用了，将会引发一个错误（不使用没关系）。没有办法来解决这种问题引起的二义性，必须由程序员自己修正。

使用 `c.a` 是错误的，到底是 `c.A.a` 还是 `c.B.a`。但可以完整写出来避免错误。

```

1.  type A struct {a int}
2.  type B struct {a, b int}
3.
4.  type C struct {A; B}
5.  var c C

```

第十九章 接口

《Go语言四十二章经》第十九章 接口

作者：李骁

19.1 接口是什么

Go 语言里有非常灵活的 接口 概念，通过它可以实现很多面向对象的特性。

接口定义了一组方法（方法集），但是这些方法不包含（实现）代码：它们没有被实现（它们是抽象的）。接口里也不能包含变量。

通过如下格式定义接口：

```
1. type Namer interface {  
2.     Method1(param_list) return_type  
3.     Method2(param_list) return_type  
4.     ...  
5. }
```

上面的 Namer 是一个 接口类型

（按照约定，只包含一个方法的）接口的名字由方法名加 [e]r 后缀组成，例如 Printer、Reader、Writer、Logger、Converter 等等。还有一些不常用的方式（当后缀 er 不合适时），比如 Recoverable，此时接口名以 able 结尾，或者以 I 开头。

Go 语言中的接口都很简短，通常它们会包含 0 个、最多 3 个方法。

注意：

类型不需要显式地声明它实现了某个接口，接口被隐式地实现，隐式接口解耦了实现接口的包和定义接口的包：互不依赖。

多个类型可以实现同一个接口，一个类型可以实现多个接口，实现了某个接口的类型，还可以有其它的方法。

接口类型是由一组方法定义的集合。接口类型的值可以存放实现这些方法的任何值。

类型（比如结构体）实现接口方法集中的所有方法，一定是接口方法集中所有方法。那么接口类型的值其实也可以存放该结构体的值。

如：

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type A struct {
8.     Face int
9. }
10.
11. type B interface {
12.     f()
13. }
14.
15. func (a A) f() {
16.     fmt.Println("hi ", a.Face)
17. }
18.
19. func main() {
20.     var s A = A{Face: 9}
21.     s.f()
22.
23.     var b B = A{Face: 9} //接口类型可接受结构体的值，因为结构体实现了接口
24.     b.f()
25. }

```

即使接口在类型之后才定义，二者处于不同的包中，被单独编译：只要类型实现了接口中的方法，它就实现了此接口。

所有这些特性使得接口具有很大的灵活性。

接口变量里包含了接收器实例的值和指向对应方法表的指针，也就是说接口实例上可以调用该实例方法，它使此方法更具有一般性。

注意：接口中的方法必须要全部实现，才能实现接口。

19.2 接口嵌套

一个接口可以包含一个或多个其他的接口，这相当于直接将这些内嵌接口的方法列举在外层接口中一样。但是在接口内不能内嵌结构体，编译会出错。

比如接口 `File` 包含了 `ReadWrite` 和 `Lock` 的所有方法，它还额外有一个 `Close()` 方法。和结

构体内嵌基本差不多。

```

1. type ReadWrite interface {
2.     Read(b Buffer) bool
3.     Write(b Buffer) bool
4. }
5.
6. type Lock interface {
7.     Lock()
8.     Unlock()
9. }
10.
11. type File interface {
12.     ReadWrite
13.     Lock
14.     Close()
15. }
```

19.3 类型断言

如何检测和转换接口变量的类型呢？这是类型断言（Type Assertion）就用上了。

一个接口类型的变量 `varI` 中可以包含任何类型的值，必须有一种方式来检测它的 动态 类型，即运行时在变量中存储的值的实际类型。在执行过程中动态类型可能会有所不同，但是它总是可以分配给接口变量本身的类型。通常我们可以使用 类型断言 来测试在某个时刻接口变量 `varI` 是否包含类型 `T` 的值：

```
1. v := varI.(T)           // unchecked type assertion
```

varI 必须是一个接口变量，否则编译器会报错：invalid type assertion: varI.(T) (non-interface type (type of varI) on left)。

Go 语言中的所有程序都实现了 `interface{}` 的接口，这意味着，所有的类型如 `string`，`int`，`int64` 甚至是自定义的 `struct` 类型都就此拥有了 `interface{}` 的接口，这种做法和 `java` 中的 `Object` 类型比较类似。那么在一个数据通过 `func funcName(interface{})` 的方式传进来的时候，也就意味着这个参数被自动的转为 `interface{}` 的类型。

```

1. func funcName(a interface{}) string {
2.     return string(a)
3. }
```

类型断言可能是无效的，虽然编译器会尽力检查转换是否有效，但是它不可能预见所有的可能性。如果转换在程序运行时失败会导致错误发生。更安全的方式是使用以下形式来进行类型断言：

```
1. if v, ok := varI.(T); ok { // 类型断言检查
2.     Process(v)
3.     return
4. }
```

如果转换合法，v 是 varI 转换到类型 T 的值，ok 会是 true；否则 v 是类型 T 的零值，ok 是 false，也没有运行时错误发生。

- 类型判断：type-switch

接口变量的类型也可以使用一种特殊形式的 switch 来检测：type-switch

```
1. switch t := areaIntf.(type) {
2. case *Square:
3.     fmt.Printf("Type Square %T with value %v\n", t, t)
4. case *Circle:
5.     fmt.Printf("Type Circle %T with value %v\n", t, t)
6. case nil:
7.     fmt.Printf("nil value: nothing to check?\n")
8. default:
9.     fmt.Printf("Unexpected type %T\n", t)
10. }
```

可以用 type-switch 进行运行时类型分析，但是在 type-switch 不允许有 fallthrough。

在处理来自于外部的、类型未知的数据时，比如解析诸如 Json 或 XML 编码的数据，类型测试和转换会非常有用。

- 测试一个值是否实现了某个接口

我们想测试它是否实现了 Stringer 接口，可以这样做：

```
1. type Stringer interface {
2.     String() string
3. }
4.
5. if sv, ok := v.(Stringer); ok {
6.     fmt.Printf("v implements String(): %s\n", sv.String())
7. }
```


`Print` 函数就是如此检测类型是否可以打印自身的。

接口是一种契约，实现类型必须满足它，它描述了类型的行为，规定类型可以做什么。接口彻底将类型能做什么，以及如何做分离开来，使得相同接口的变量在不同的时刻表现出不同的行为，这就是多态的本质。

编写参数是接口变量的函数，这使得它们更具有一般性。

使用接口使代码更具有普适性。

标准库里到处都使用了这个原则，如果对接口概念没有良好的把握，是不可能理解它是如何构建的。

19.4 接口与动态类型

Go 的动态类型

在经典的面向对象语言（像 C++，Java 和 C#）中数据和方法被封装为 **类** 的概念：类包含它们两者，并且不能剥离。

Go 没有类：数据（结构体或更一般的类型）和方法是一种松耦合的正交关系。

Go 中的接口跟 Java/C# 类似：都是必须提供一个指定方法集的实现。但是更加灵活通用：任何提供了接口方法实现代码的类型都隐式地实现了该接口，而不用显式地声明。

和其它语言相比，Go 是唯一结合了接口值，静态类型检查（是否该类型实现了某个接口），运行时动态转换的语言，并且不需要显式地声明类型是否满足某个接口。该特性允许我们在不改变已有的代码的情况下定义和使用新接口。

接收一个（或多个）接口类型作为参数的函数，其实参可以是任何实现了该接口的类型。实现了某个接口的类型可以被传给任何以此接口为参数的函数。

类似于 Python 和 Ruby 这类动态语言中的 **动态类型**（duck typing）；这意味着对象可以根据提供的方法被处理（例如，作为参数传递给函数），而忽略它们的实际类型：它们能做什么比它们是什么更重要。

动态方法调用

像 Python，Ruby 这类语言，动态类型是延迟绑定的（在运行时进行）：方法只是用参数和变量简单地调用，然后在运行时才解析（它们很可能有像 `responds_to` 这样的方法来检查对象是否可以响应某个方法，但是这也意味着更大的编码量和更多的测试工作）

Go 的实现与此相反，通常需要编译器静态检查的支持：当变量被赋值给一个接口类型的变量时，编译器会检查其是否实现了该接口的所有函数。如果方法调用作用于像 `interface{}` 这样的“泛型”上，你可以通过类型断言来检查变量是否实现了相应接口。

因此 Go 提供了动态语言的优点，却没有其他动态语言在运行时可能发生错误的缺点。

对于动态语言非常重要的单元测试来说，这样即可以减少单元测试的部分需求，又可以发挥相当大的作用。

Go 的接口提高了代码的分离度，改善了代码的复用性，使得代码开发过程中的设计模式更容易实现。用 Go 接口还能实现 依赖注入模式。

19.5 接口的提取

提取接口，是非常有用的设计模式，可以减少需要的类型和方法数量，而且不需要像传统的基于类的面向对象语言那样维护整个的类层次结构。

Go 接口可以让开发者找出自己写的程序中的类型。假设有一些拥有共同行为的对象，并且开发者想要抽象出这些行为，这时就可以创建一个接口来使用

所以你不用提前设计出所有的接口；整个设计可以持续演进，而不用废弃之前的决定。类型要实现某个接口，它本身不用改变，你只需要在这个类型上实现新的方法。

显式地指明类型实现了某个接口

如果你希望满足某个接口的类型显式地声明它们实现了这个接口，你可以向接口的方法集中添加一个具有描述性名字的方法。

大部分代码并不使用这样的约束，因为它限制了接口的实用性。

但是有些时候，这样的约束在大量相似的接口中被用来解决歧义。

19.6 接口的继承

当一个类型包含（内嵌）另一个类型（实现了一个或多个接口）的指针时，这个类型就可以使用（另一个类型）所有的接口方法。

类型可以通过继承多个接口来提供像 多重继承 一样的特性：

```
1. type ReaderWriter struct {  
2.     *io.Reader  
3.     *io.Writer  
4. }
```

上面概述的原理被应用于整个 Go 包，多态用得越多，代码就相对越少。这被认为是 Go 编程中的重要最佳实践。

第二十章 方法

《Go语言四十二章经》第二十章 方法

作者：李骁

在前面我们讲了结构（struct）和接口（interface），但关于这两种类型中非常重要的的方法以及方法调用一直没有具体讲解。那么在这一章里，我们来仔细看看方法有那些奇妙之处呢？

20.1 方法的定义

在 Go 语言中，结构体就像是类的一种简化形式，那么面向对象程序员可能会问：类的方法在哪里呢？在 Go 中有一个概念，它和方法有着同样的名字，并且大体上意思相近：

Go 方法是作用在接收器（receiver）上的一个函数，接收器是某种类型的变量。因此方法是一种特殊类型的函数。

定义方法的一般格式如下：

```
1. func (recv receiver_type) methodName(parameter_list) (return_value_list) { ...  
    }
```

在方法名之前，func 关键字之后的括号中指定接收器 receiver。

```
1. type A struct {  
2.     Face int  
3. }  
4.  
5. func (a A) f() {  
6.     fmt.Println("hi ", a.Face)  
7. }
```

上面代码中，我们定义了结构体 A，注意f()就是 A 的方法，(a A)表示接收器。

a 是 receiver 的实例，f()是它的方法名，那么方法调用遵循传统的 object.name 选择器符号：a.f()。

如果 recv 一个指针，Go 会自动解引用。如果方法不需要使用 recv 的值，可以用 _ 替换它，比如：

```
1. func (_ receiver_type) methodName(parameter_list) (return_value_list) { ... }
```

- 接收器类型可以是（几乎）任何类型，不仅仅是结构体类型：任何类型都可以有方法，甚至可以是函数类型，可以是 `int`、`bool`、`string` 或数组的别名类型。

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type MyInt int
8.
9. func (m MyInt) p() {
10.     fmt.Println("Now", m)
11. }
12.
13. func main() {
14.     var pp MyInt = 8
15.     pp.p()
16. }
```

```
1. 程序输出：
2. Now 8
```

- 接收器不能是一个接口类型，因为接口是一个抽象定义，但是方法却是具体实现；如果这样做会引发一个编译错误：`invalid receiver type...`。
- 接收器不能是一个指针类型，但是它可以是任何其他允许类型的指针。

关于接收器的命名

社区约定的接收器命名是类型的一个或两个字母的缩写(像 `c` 或者 `cl` 对于 `Client`)。不要使用泛指的名字像是 `me`, `this` 或者 `self`, 也不要使用过度描述的名字, 最后, 如果你在一个地方使用了 `c`, 那么就不要再别的地方使用 `cl`。

一个类型加上它的方法等价于面向对象中的一个类。一个重要的区别是：在 Go 中，类型的代码和绑定在它上面的方法的代码可以不放置在一起，它们可以存在在不同的源文件，唯一的要求是：它们必须是同一个包的。

类型 `T` (或 `T`) 上的所有方法的集合叫做类型 `T` (或 `T`) 的方法集。

因为方法是函数，所以同样的，不允许方法重载，即对于一个类型只能有一个给定名称的方法。但是如

果基于接收器类型，是有重载的：具有同样名字的方法可以在 2 个或多个不同的接收器类型上存在，比如在同一包里这么做是允许的：

```
1. func (a *denseMatrix) Add(b Matrix) Matrix
2. func (a *sparseMatrix) Add(b Matrix) Matrix
```

下面是非结构体类型上方法的例子：

```
1. type IntVector []int
2.
3. func (v IntVector) Sum() (s int) {
4.     for _, x := range v {
5.         s += x
6.     }
7.     return
8. }
```

类型和作用在它上面定义的方法必须在同一个包里定义，这就是为什么不能在 `int`、`float` 或类似这些的类型上定义方法。

类型在其他的，或是非本地的包里定义，在它上面定义方法都会得到和上面同样的错误。

- 但是有一个间接的方式：可以先定义该类型（比如：`int` 或 `float`）的新的自定义类型，然后再为自定义类型定义方法。

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type MyInt int
8. type HeInt MyInt
9.
10. func (m MyInt) p() {
11.     fmt.Println("Now", m)
12. }
13.
14. func main() {
15.     var pp MyInt = 8
16.     pp.p()
17.
```

```

18. hh := HeInt(pp)
19.     hh.p()
20. }

```

程序运行结果：hh.p undefined (type HeInt has no field or method p)
 因为hh 属于新的自定义类型 HeInt ， 它没有定义p()方法，需要另外定义这个方法。

如果我们采用别名，Go 1.9及以上版本编译通过：

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type MyInt int
8. type HeInt = MyInt
9.
10. func (m MyInt) p() {
11.     fmt.Println("Now", m)
12. }
13.
14. func main() {
15.     var pp MyInt = 8
16.     pp.p()
17.
18.     hh := HeInt(pp)
19.     hh.p()
20. }

```

1. 程序输出：
2. Now 8
3. Now 8

- 或者像下面这样将它作为匿名类型嵌入在一个新的结构体中。当然方法只在这个自定义类型上有效。

```

1. package main
2.
3. import (
4.     "fmt"

```

```

5.     "time"
6. )
7. type myTime struct {
8.     time.Time
9. }
10.
11. func (t myTime) first3Chars() string {
12.     return t.Time.String()[0:3]
13. }
14. func main() {
15.     m := myTime{time.Now()}
16.     // 调用匿名Time上的String方法
17.     fmt.Println("Full time now:", m.String())
18.     // 调用myTime.first3Chars
19.     fmt.Println("First 3 chars:", m.first3Chars())
20. }

```

```

1. 程序输出：
2.
3. Full time now: 2018-08-28 20:36:47.1135231 +0800 CST m=+0.002990901
4. First 3 chars: 201

```

20.2 函数和方法的区别

函数将变量作为参数：Function1(recv)

方法在变量上被调用：recv.Method1()

在接收器是指针时，方法可以改变接收器的值（或状态），这点函数也可以做到（当参数作为指针传递，即通过引用调用时，函数也可以改变参数的状态）。

接收器必须有一个显式的名字，这个名字必须在方法中被使用。

receiver_type 叫做（接收器）基本类型，这个类型必须在和方法同样的包中被声明。

在 Go 中，（接收器）类型关联的方法不写在类型结构里面，就像类那样；耦合更加宽松；类型和方法之间的关联由接收器来建立。

方法没有和数据定义（结构体）混在一起：

- 它们是正交的类型；
- 表示（数据）和行为（方法）是独立的。

20.3 指针或值方法

鉴于性能的原因，recv 最常见的是一个指向 receiver_type 的指针（因为我们不想要一个实例的拷贝，如果按值调用的话就会是这样），特别是在 receiver 类型是结构体时，就更是如此了。

如果想要方法改变接收器的数据，就在接收器的指针类型上定义该方法；否则，就在普通的值类型上定义方法；分别叫做指针方法，值方法。

下面声明一个 T 类型的变量，并调用其方法 M1() 和 M2()。

```
1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type T struct {
8.     Name string
9. }
10.
11. func (t T) M1() {
12.     t.Name = "name1"
13. }
14.
15. func (t *T) M2() {
16.     t.Name = "name2"
17. }
18. func main() {
19.
20.     t1 := T{"t1"}
21.
22.     fmt.Println("M1调用前:", t1.Name)
23.     t1.M1()
24.     fmt.Println("M1调用后:", t1.Name)
25.
26.     fmt.Println("M2调用前:", t1.Name)
27.     t1.M2()
28.     fmt.Println("M2调用后:", t1.Name)
29.
30. }
```

1. 程序输出：

2. M1调用前 : t1
3. M1调用后 : t1
4. M2调用前 : t1
5. M2调用后 : name2

可见, `t1.M2()` 修改了接收器数据。

分析:

我们姑且认为调用 `t1.M1()` 时相当于 `M1(t1)`, 实参和行参都是类型 `T`, 可以接受。此时在 `M1()` 中的 `t` 只是 `t1` 的值拷贝, 所以 `M1()` 的修改影响不到 `t1`。

同上, `t1.M2() => M2(t1)`, 这是将 `T` 类型传给了 `*T` 类型, `go` 可能会取 `t1` 的地址传进去: `M2(&t1)`。所以 `M2()` 的修改可以影响 `t1`。

上面的例子同时也说明了:

1. `T` 类型的变量可以调用这两个方法。

因为对于类型 `T`, 如果在 `*T` 上存在方法 `Meth()`, 并且 `t` 是这个类型的变量, 那么 `t.Meth()` 会被自动转换为 `(&t).Meth()`。

下面声明一个 `*T` 类型的变量, 并调用方法 `M1()` 和 `M2()`。

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type T struct {
8.     Name string
9. }
10.
11. func (t T) M1() {
12.     t.Name = "name1"
13. }
14.
15. func (t *T) M2() {
16.     t.Name = "name2"
17. }
18. func main() {
19.
20.     t2 := &T{"t2"}
```

```

21.
22.     fmt.Println("M1调用前:", t2.Name)
23.     t2.M1()
24.     fmt.Println("M1调用后:", t2.Name)
25.
26.     fmt.Println("M2调用前:", t2.Name)
27.     t2.M2()
28.     fmt.Println("M2调用后:", t2.Name)
29.
30. }
```

1. 程序输出：
2. M1调用前： t2
3. M1调用后： t2
4. M2调用前： t2
5. M2调用后： name2

分析：

`t2.M1() => M1(t2)`， `t2` 是指针类型， 取 `t2` 的值并拷贝一份传给 `M1`。

`t2.M2() => M2(t2)`，都是指针类型，不需要转换。

1. `*T` 类型的变量也可以调用这两个方法。

从上面我们可以得知：无论你声明方法的接收器是指针接收器还是值接收器，*Go*都可以帮你隐式转换为正确的方法使用。

但我们需要记住，值变量只拥有值方法，而指针变量则同时拥有值方法和指针方法。

无论是T类型变量还是*T类型变量，都可调用值方法或指针方法。但如果是接口变量呢，那么这两个方法都可以调用吗？

我们添加一个接口看看：

```

1. package main
2.
3. type T struct {
4.     Name string
5. }
6. type Intf interface {
7.     M1()
8.     M2()
9. }
```

```

10.
11. func (t T) M1() {
12.     t.Name = "name1"
13. }
14.
15. func (t *T) M2() {
16.     t.Name = "name2"
17. }
18. func main() {
19.     var t1 T = T{"t1"}
20.     t1.M1()
21.     t1.M2()
22.
23.     var t2 Intf = t1
24.     t2.M1()
25.     t2.M2()
26. }

```

编译不通过：

cannot use t1 (type T) as type Intf in assignment:

T does not implement Intf (M2 method has pointer receiver)

上面代码中我们看到，`var t2 Intf` 中，`t2`是`Intf`接口类型变量，`t1`是`T`类型值变量。上面错误信息中已经明确了`T`没有实现接口`Intf`，所以不能直接赋值。这是为什么呢？

首先这是Go语言的一种规则，具体如下：

- 规则一：如果使用指针方法来实现一个接口，那么只有指向那个类型的指针才能够实现对应的接口。
- 规则二：如果使用值方法来实现一个接口，那么那个类型的值和指针都能够实现对应的接口。

按照上面两条规则的规则一，我们稍微修改下代码：

```

1. package main
2.
3. type T struct {
4.     Name string
5. }
6. type Intf interface {
7.     M1()
8.     M2()
9. }

```

```

10.
11. func (t T) M1() {
12.     t.Name = "name1"
13. }
14.
15. func (t *T) M2() {
16.     t.Name = "name2"
17. }
18. func main() {
19.
20.     var t1 T = T{"t1"}
21.     t1.M1()
22.     t1.M2()
23.
24.     var t2 Intf = &t1
25.     t2.M1()
26.     t2.M2()
27. }

```

程序编译通过。

综合起来，接口类型的变量（实现了该接口）调用方法时，我们需要注意方法的接收器，是不是真正实现了接口。结合接口类型断言，我们做下测试：

```

1. package main
2.
3. import (
4.     "fmt"
5. )
6.
7. type T struct {
8.     Name string
9. }
10. type Intf interface {
11.     M1()
12.     M2()
13. }
14.
15. func (t T) M1() {
16.     t.Name = "name1"
17.     fmt.Println("M1")
18. }

```

```

19.
20. func (t *T) M2() {
21.     t.Name = "name2"
22.     fmt.Println("M2")
23. }
24. func main() {
25.
26.     var t1 T = T{"t1"}
27.
28.     // interface{}(t1) 先转为空接口, 再使用接口断言
29.     _, ok1 := interface{}(t1).(Intf)
30.     fmt.Println("t1 => Intf", ok1)
31.
32.     _, ok2 := interface{}(t1).(T)
33.     fmt.Println("t1 => T", ok2)
34.     t1.M1()
35.     t1.M2()
36.
37.     _, ok3 := interface{}(t1).(*T)
38.     fmt.Println("t1 => *T", ok3)
39.     t1.M1()
40.     t1.M2()
41.
42.     T, ok4 := interface{}(&t1).(Intf)
43.     fmt.Println("&t1 => Intf", ok4)
44.     t.M1()
45.     t.M2()
46.
47.     _, ok5 := interface{}(&t1).(T)
48.     fmt.Println("&t1 => T", ok5)
49.
50.     _, ok6 := interface{}(&t1).(*T)
51.     fmt.Println("&t1 => *T", ok6)
52.     t1.M1()
53.     t1.M2()
54.
55. }

```

1. 程序输出：
2. `t1 => Intf false`
3. `t1 => T true`
4. M1

```

5.  M2
6.  t1 => *T false
7.  M1
8.  M2
9.  &t1 => Intf true
10. M1
11. M2
12. &t1 => T false
13. &t1 => *T true
14. M1
15. M2

```

执行结果表明，`t1` 没有实现`Intf`方法集，不是`Intf`接口类型；而`&t1` 则实现了`Intf`方法集，是`Intf`接口类型，可以调用相应方法。而`t1` 这个结构体值变量本身则调用值方法或者指针方法都是可以的。

按照上面的两条规则，那究竟怎么选择是指针接收器还是值接收器呢？

- 何时使用值类型

1. 如果接收器是一个 `map`，`func` 或者 `chan`，使用值类型(因为它们本身就是引用类型)。
2. 如果接收器是一个 `slice`，并且方法不执行 `reslice` 操作，也不重新分配内存给 `slice`，使用值类型。
3. 如果接收器是一个小的数组或者原生的值类型结构体类型(比如 `time.Time` 类型)，而且没有可修改的字段和指针，又或者接收器是一个简单地基本类型像是 `int` 和 `string`，使用值类型就好了。

一个值类型的接收器可以减少一定数量的垃圾生成，如果一个值被传入一个值类型接收器的方法，一个栈上的拷贝会替代在堆上分配内存(但不是保证一定成功)，所以在没搞明白代码想干什么之前，别因为这个原因而选择值类型接收器。

- 何时使用指针类型

1. 如果方法需要修改接收器，接收器必须是指针类型。
2. 如果接收器是一个包含了 `sync.Mutex` 或者类似同步字段的结构体，接收器必须是指针，这样可以避免拷贝。
3. 如果接收器是一个大的结构体或者数组，那么指针类型接收器更有效率。(多大算大呢？假设把接收器的所有元素作为参数传给方法，如果你觉得参数有点多，那么它就是大)。
4. 从此方法中并发的调用函数和方法时，接收器可以被修改吗？一个值类型的接收器当方法调用时会创建一份拷贝，所以外部的修改不能作用到这个接收器上。如果修改必须被原始的接收器可见，那么接收

器必须是指针类型。

5. 如果接收器是一个结构体，数组或者 slice，它们中任意一个元素是指针类型而且可能被修改，建议使用指针类型接收器，这样会增加程序的可读性

当你看完这个还是有疑虑，还是不知道该使用哪种接收器，那么记住使用指针接收器。

20.4 内嵌类型的方法提升

当一个匿名类型被内嵌在结构体中时，匿名类型的可见方法也同样被内嵌，这在效果上等同于外层类型继承了这些方法：将父类型放在子类型中来实现。这个机制提供了一种简单的方式来模拟经典面向对象语言中的子类 and 继承相关的效果。

当我们嵌入一个类型，这个类型的方法就变成了外部类型的方法，但是当它被调用时，方法的接收器是内部类型(嵌入类型)，而非外部类型。

```

1.  type People struct {
2.     Age    int
3.     gender string
4.     Name   string
5. }
6.
7.  type OtherPeople struct {
8.     People
9. }
10.
11. func (p People) PeInfo() {
12.     fmt.Println("People ", p.Name, ": ", p.Age, "岁, 性别:", p.gender)
13. }
```

因此嵌入类型的名字充当着字段名，同时嵌入类型作为内部类型存在，我们可以使用下面的调用方法：

```
1. OtherPeople.People.PeInfo()
```

这儿我们通过类型名称来访问内部类型的字段和方法。然而，这些字段和方法也同样被提升到了外部类型：

```
1. OtherPeople.PeInfo()
```

下面是 Go 语言中内嵌类型方法集提升的规则：

给定一个结构体类型 S 和一个命名为 T 的类型，方法提升像下面规定的这样被包含在结构体方法集

中：

- 如果 **S** 包含一个匿名字段 **T**，**S** 和 ***S** 的方法集都包含接收器为 **T** 的方法提升

这条规则说的是当我们嵌入一个类型，嵌入类型的接收器为值类型的方法将被提升，可以被外部类型的值和指针调用。

- 如果 **S** 包含一个匿名字段 **T**，***S** 类型的方法集包含接收器为 ***T** 的方法提升

这条规则说的是当我们嵌入一个类型，可以被外部类型的指针调用的方法集只有嵌入类型的接收器为指针类型的方法集，也就是说，当外部类型使用指针调用内部类型的方法时，只有接收器为指针类型的内部类型方法集将被提升。

- 如果 **S** 包含一个匿名字段 ***T**，**S** 和 ***S** 的方法集都包含接收器为 **T** 或者 ***T** 的方法提升

这条规则说的是当我们嵌入一个类型的指针，嵌入类型的接收器为值类型或指针类型的方法将被提升，可以被外部类型的值或者指针调用。

这就是语言规范里方法提升中仅有的三条规则，根据这个推导出一条规则：

- 如果 **S** 包含一个匿名字段 **T**，**S** 的方法集不包含接收器为 ***T** 的方法提升。

这条规则说的是当我们嵌入一个类型，嵌入类型的接收器为指针的方法将不能被外部类型的值访问。这也是跟我们陈述的接口规则一致。

简单地说也是两条规则：

规则一：如果**S**包含嵌入字段**T**，则**S**和***S**的方法集都包括具有接收方**T**的提升方法。***S**的方法集还包括具有接收方***T**的提升方法。

规则二：如果**S**包含嵌入字段**T**，则**S**和**\S**的方法集都包括具有接收器**T**或***T**的提升方法。

注意：以上规则由于 `t.Meth()` 会被自动转换为 `(&t).Meth()` 这个语法糖，导致我们很容易误解上面的规则不起作用，而实际上规则是有效的。

我们通过下面代码验证如下：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "reflect"
6. )
7.
8. type People struct {
9.     Age    int
```



```

10.     gender string
11.     Name   string
12. }
13.
14. type OtherPeople struct {
15.     People
16. }
17.
18. type NewPeople People
19.
20. func (p *NewPeople) PeName(pname string) {
21.     fmt.Println("pold name:", p.Name)
22.     p.Name = pname
23.     fmt.Println("pnew name:", p.Name)
24. }
25.
26. func (p NewPeople) PeInfo() {
27.     fmt.Println("NewPeople ", p.Name, ": ", p.Age, "岁, 性别:", p.gender)
28. }
29.
30. func (p *People) PeName(pname string) {
31.     fmt.Println("old name:", p.Name)
32.     p.Name = pname
33.     fmt.Println("new name:", p.Name)
34. }
35.
36. func (p People) PeInfo() {
37.     fmt.Println("People ", p.Name, ": ", p.Age, "岁, 性别:", p.gender)
38. }
39.
40. func methodSet(a interface{}) {
41.     t := reflect.TypeOf(a)
42.     for i, n := 0, t.NumMethod(); i < n; i++ {
43.         m := t.Method(i)
44.         fmt.Println(m.Name, m.Type)
45.     }
46. }
47.
48. func main() {
49.     p := OtherPeople{People{26, "Male", "张三"}}
50.     p.PeInfo()
51.     p.PeName("Joke")

```

```

52.
53.     methodSet(p) // T方法提升
54.
55.     methodSet(&p) // *T和T方法提升
56.
57.     pp := NewPeople{42, "Male", "李四"}
58.     pp.PeInfo()
59.     pp.PeName("Haw")
60.
61.     methodSet(&pp)
62. }

```

```

1.  程序输出：
2.
3.  People 张三： 26 岁，性别：Male
4.  old name: 张三
5.  new name: Joke
6.  PeInfo func(main.OtherPeople)
7.  PeInfo func(*main.OtherPeople)
8.  PeName func(*main.OtherPeople, string)
9.  NewPeople 李四： 42 岁，性别：Male
10. pold name: 李四
11. pnew name: Haw
12. PeInfo func(*main.NewPeople)
13. PeName func(*main.NewPeople, string)

```

我们可以从上面输出看到，`*OtherPeople` 下有两个方法，而`OtherPeople`只有一个方法。

但是在Go中存在一个语法糖，比如上面代码：

```

1.     p.PeInfo()
2.     p.PeName("Joke")
3.
4.     methodSet(p) // T方法提升

```

虽然`P` 只有一个方法：`PeInfo func(main.OtherPeople)`，但我们依然可以调用 `p.PeName("Joke")`。

这里Go自动转为 `(&p).PeName("Joke")`，其调用后结果让我们以为`p`有两个方法，其实这里`p`只有一个方法。

有关于内嵌方法集的提升，初学者留意下这个规则。

结合前面的自定义类型赋值接口类型的规则，与内嵌类型的方法集提升规则这两个大规则一定要弄清楚，只有彻底弄清楚这些规则，我们在阅读和写代码时才能做到气闲神定。

第二十一章 协程(goroutine)

《Go语言四十二章经》第二十一章 协程(goroutine)

作者：李骁

Concurrency is about dealing with lots of things at once.

Parallelism is about doing lots of things at once.

并发：指的是程序的逻辑结构。如果程序代码结构中的某些函数逻辑上可以同时运行，但物理上未必会同时运行。

并行：并行是指程序的运行状态。并行则指的就是在物理层面也就是使用了不同CPU在执行不同或者相同的任务。

21.1 并发

并发是在同一时间处理（dealing with）多件事情。并行是在同一时间做（doing）多件事情。并发的目的在于把单个 CPU 的利用率使用到最高。并行则需要多核 CPU 的支持。

Go 语言从语言层面上就支持了并发，goroutine是Go语言提供的一种用户态线程，有时我们也称之为协程。所谓的协程，某种程度上也可以叫做轻量线程，它不由os，而由应用程序创建和管理，因此使用开销较低（一般为4K）。我们可以创建很多的goroutine，并且它们跑在同一个内核线程之上的时候，就需要一个调度器来维护这些goroutine，确保所有的goroutine都使用cpu，并且是尽可能公平的使用cpu资源。调度器的主要有4个重要部分，分别是M、G、P、Sched，前三个定义在runtime.h中，Sched定义在proc.c中。

- M（work thread）代表了系统线程OS Thread，由操作系统管理。
- P（processor）衔接M和G的调度上下文，它负责将等待执行的G与M对接。P的数量可以通过GOMAXPROCS()来设置，它其实也就代表了真正的并发度，即有多少个goroutine可以同时运行。
- G（goroutine）goroutine的实体，包括了调用栈，重要的调度信息，例如channel等。

在操作系统的OS Thread和编程语言的User Thread之间，实际上存在3中线程对应模型，也就是：1:1，1:N，M:N。

N:1 多个（N）用户线程始终在一个内核线程上跑，context上下文切换很快，但是无法真正的利用多核。

1:1 一个用户线程就只在一个内核线程上跑，这时可以利用多核，但是上下文切换很慢，切换效率很

低。

M:N 多个goroutine在多个内核线程上跑，这个可以集齐上面两者的优势，但是无疑增加了调度的难度。

M:N 综合两种方式 (N:1, 1:1) 的优势。多个 goroutines 可以在多个 OS threads 上处理。既能快速切换上下文，也能利用多核的优势，而Go正是选择这种实现方式。

Go 的goroutine是运行在虚拟CPU中的(通过runtime.GOMAXPROCS(1)所设定的虚拟CPU个数)。虚拟CPU个数未必会和实际CPU个数相吻合。

每个goroutine都会被一个特定的P(虚拟CPU)选定维护，而M(物理计算资源)每次挑选一个有效P，然后执行P中的goroutine。

每个P会将自己所维护的goroutine放到一个G队列中，其中就包括了goroutine堆栈信息，是否可执行信息等等。

默认情况下，P的数量与实际物理CPU的数量相等。当我们通过循环来创建goroutine时，goroutine会被分配到不同的G队列中。而M的数量又不是唯一的，当M随机挑选P时，也就等同随机挑选了goroutine。

所以，当我们碰到多个goroutine的执行顺序不是我们想象的顺序时就可以理解了，因为goroutine进入P管理的队列G是带有随机性的。

P的数量由runtime.GOMAXPROCS(1)所设定，通常来说它是和内核数对应，例如在4Core的服务器上会启动4个线程。G会有很多个，每个P会将goroutine从一个就绪的队列中做Pop操作，为了减小锁的竞争，通常情况下每个P会负责一个队列。

```
1. runtime.NumCPU()           // 返回当前CPU内核数
2. runtime.GOMAXPROCS(2)     // 设置运行时最大可执行CPU数
3. runtime.NumGoroutine()    // 当前正在运行的goroutine 数
```

P维护着这个队列(称之为runqueue)，Go语言里，启动一个goroutine很容易：go function 就行，所以每有一个go语句被执行，runqueue队列就在其末尾加入一个goroutine，在下一个调度点，就从runqueue中取出一个goroutine执行。

假如有两个M，即两个OS Thread线程，分别对应一个P，每一个P调度一个G队列。如此一来，就组成的goroutine运行时的基本结构：

- 当有一个M返回时，它必须尝试取得一个P来运行goroutine，一般情况下，它会从其他的OS Thread线程那里窃取一个P过来，如果没有拿到，它就把goroutine放在一个global runqueue里，然后自己进入线程缓存里。
- 如果某个P所分配的任务G很快就执行完了，这会导致多个队列存在不平衡，会从其他队列中截取

一部分goroutine到P上进行调度。一般来说，如果P从其他的P那里要取任务的话，一般就取run queue的一半，这就确保了每个OS线程都能充分的使用。

- 当一个OS Thread线程被阻塞时，P可以转而投奔另一个OS线程。

下面是G、 M、 P的具体结构，这不是Go代码：

```

1. struct G
2. {
3.     uintptr stackguard0; // 用于栈保护，但可以设置为StackPreempt，用于实现抢占式调度
4.     uintptr stackbase; // 栈顶
5.     Gobuf sched; // 执行上下文，G的暂停执行和恢复执行，都依靠它
6.     uintptr stackguard; // 跟stackguard0一样，但它不会被设置为StackPreempt
7.     uintptr stack0; // 栈底
8.     uintptr stacksize; // 栈的大小
9.     int16 status; // G的六个状态
10.    int64 goid; // G的标识id
11.    int8* waitreason; // 当status==Gwaiting有用，等待的原因，可能是调用time.Sleep
    之类
12.    G* schedlink; // 指向链表的下一个G
13.    uintptr gopc; // 创建此goroutine的Go语句的程序计数器PC，通过PC可以获得具体的
    函数和代码行数
14. };
15. struct P
16. {
17.     Lock; // plan9 C的扩展语法，相当于Lock lock;
18.     int32 id; // P的标识id
19.     uint32 status; // P的四个状态
20.     P* link; // 指向链表的下一个P
21.     M* m; // 它当前绑定的M，Pidle状态下，该值为nil
22.     MCache* mcache; // 内存池
23.     // Grunnable状态的G队列
24.     uint32 runqhead;
25.     uint32 runqtail;
26.     G* runq[256];
27.     // Gdead状态的G链表（通过G的schedlink）
28.     // gfreecnt是链表上节点的个数
29.     G* gfree;
30.     int32 gfreecnt;
31. };
32. struct M
33. {
34.     G* g0; // M默认执行G

```

```

35.     void    (*mstartfn)(void); // OS线程执行的函数指针
36.     G*   curg;           // 当前运行的G
37.     P*   p;             // 当前关联的P, 要是当前不执行G, 可以为nil
38.     P*   nextp; // 即将要关联的P
39.     int32 id; // M的标识id
40.     M*   alllink; // 加到allm, 使其不被垃圾回收(GC)
41.     M*   schedlink; // 指向链表的下一个M
42. };

```

21.2 goroutine

在Go中, goroutine的使用很简单, 直接在代码前加上关键字 `go` 即可。`go`关键字就是用来创建一个goroutine的, 后面的代码块就是这个goroutine需要执行的代码逻辑。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {
9.     for i := 1; i < 10; i++ {
10.         go func(i int) {
11.             fmt.Println(i)
12.         }(i)
13.     }
14.     // 暂停一会, 保证打印全部结束
15.     time.Sleep(1e9)
16. }

```

有关于goroutine 之间的通信以及goroutine与主线程的控制, 我们后续通过channel、context 以及锁来进一步说明。

第二十二章 通道(channel)

《Go语言四十二章经》第二十二章 通道(channel)

作者：李骁

22.1 通道(channel)

Go 奉行通过通信来共享内存，而不是共享内存来通信。所以，**channel** 是**goroutine**之间互相通信的通道，goroutine之间可以通过它发消息和接收消息。

channel是进程内的通信方式，因此通过channel传递对象的过程和调用函数时的参数传递行为比较一致，比如也可以传递指针等。

channel是类型相关的，一个channel只能传递（发送或接受 | send or receive）一种类型的值，这个类型需要在声明channel时指定。

默认的，信道的存消息和取消息都是阻塞的（叫做无缓冲的信道）

使用make来建立一个通道：

```
1. var channel chan int = make(chan int)
2. // 或
3. channel := make(chan int)
```

Go中channel可以是发送（send）、接收（receive）、同时发送（send）和接收（receive）。

```
1. // 定义接收的channel
2. receive_only := make (<-chan int)
3.
4. // 定义发送的channel
5. send_only := make (chan<- int)
6.
7. // 可同时发送接收
8. send_receive := make (chan int)
```

- chan<- 表示数据进入通道，要把数据写进通道，对于调用者就是发送。
- <-chan 表示数据从通道出来，对于调用者就是得到通道的数据，当然就是接收。

定义只发送或只接收的channel意义不大，一般用于在参数传递中：


```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {
9.     c := make(chan int) // 不使用带缓冲区的channel
10.    go send(c)
11.    go recv(c)
12.    time.Sleep(3 * time.Second)
13.    close(c)
14. }
15.
16. // 只能向chan里send数据
17. func send(c chan<- int) {
18.     for i := 0; i < 10; i++ {
19.
20.         fmt.Println("send ready ", i)
21.         c <- i
22.         fmt.Println("send ", i)
23.     }
24. }
25.
26. // 只能接收channel中的数据
27. func recv(c <-chan int) {
28.     for i := range c {
29.         fmt.Println("received ", i)
30.     }
31. }

```

```

1. 程序输出：
2.
3. send ready  0
4. send  0
5. send ready  1
6. received  0
7. received  1
8. send  1
9. send ready  2

```

```

10. send 2
11. send ready 3
12. received 2
13. received 3
14. send 3
15. send ready 4
16. send 4
17. send ready 5
18. received 4
19. received 5
20. send 5
21. send ready 6
22. send 6
23. send ready 7
24. received 6
25. received 7
26. send 7
27. send ready 8
28. send 8
29. send ready 9
30. received 8
31. received 9
32. send 9

```

运行结果上我们可以发现一个现象，往channel 发送数据后，这个数据如果没有取走，channel是阻塞的，也就是不能继续向channel 里面发送数据。因为上面代码中，我们没有指定channel 缓冲区的大小，默认是阻塞的。

我们可以建立带缓冲区的 channel：

```
1. c := make(chan int, 1024)
```

我们把前面的程序修改下：

```

1. package main
2.
3. import (
4.     "fmt"
5.     "time"
6. )
7.
8. func main() {

```

```

9.      c := make(chan int, 10) // 使用带缓冲区的channel
10.     go send(c)
11.     go recv(c)
12.     time.Sleep(3 * time.Second)
13.     close(c)
14. }
15.
16. // 只能向chan里send发送数据
17. func send(c chan<- int) {
18.     for i := 0; i < 10; i++ {
19.
20.         fmt.Println("send ready ", i)
21.         c <- i
22.         fmt.Println("send ", i)
23.     }
24. }
25.
26. // 只能接收channel中的数据
27. func recv(c <-chan int) {
28.     for i := range c {
29.         fmt.Println("received ", i)
30.     }
31. }

```

```

1. 程序输出：
2.
3. send ready  0
4. send  0
5. send ready  1
6. send  1
7. send ready  2
8. send  2
9. send ready  3
10. send  3
11. send ready  4
12. send  4
13. send ready  5
14. received  0
15. received  1
16. received  2
17. received  3
18. received  4

```

```

19. received 5
20. send 5
21. send ready 6
22. send 6
23. send ready 7
24. send 7
25. send ready 8
26. send 8
27. send ready 9
28. send 9
29. received 6
30. received 7
31. received 8
32. received 9

```

从运行结果我们可以看到（每次执行顺序不一定相同，goroutine 运行导致的原因），带有缓冲区的 channel，在缓冲区有数据而未填满前，读取不会出现阻塞的情况。

- 无缓冲的通道（unbuffered channel）是指在接收前没有能力保存任何值的通道。

这种类型的通道要求发送 goroutine 和接收 goroutine 同时准备好，才能完成发送和接收操作。如果两个goroutine没有同时准备好，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。

- 有缓冲的通道（buffered channel）是一种在被接收前能存储一个或者多个值的通道。

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。通道会阻塞发送和接收动作的条件也会不同。只有在通道中没有要接收的值时，接收动作才会阻塞。只有在通道没有可用缓冲区容纳被发送的值时，发送动作才会阻塞。

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。

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。

可以通过内置的close函数来关闭channel实现。

- channel不像文件一样需要经常去关闭，只有当你确实没有任何发送数据了，或者你想显式的结束range循环之类的，才去关闭channel；
- 关闭channel后，无法向channel 再发送数据(引发 panic 错误后导致接收立即返回零值)；
- 关闭channel后，可以继续向channel接收数据，不能继续发送数据；

- 对于nil channel，无论收发都会被阻塞。

第二十三章 同步与锁

《Go语言四十二章经》第二十三章 同步与锁

作者：李骁

23.1 同步锁

Go语言包中的sync包提供了两种锁类型：sync.Mutex和sync.RWMutex，前者是互斥锁，后者是读写锁。

互斥锁是传统的并发程序对共享资源进行访问控制的主要手段，在Go中，似乎更推崇由channel来实现资源共享和通信。它由标准库代码包sync中的Mutex结构体类型代表。只有两个公开方法：调用Lock()获得锁，调用unlock()释放锁。

- 使用Lock()加锁后，不能再继续对其加锁（同一个goroutine中，即：同步调用），否则会panic。只有在unlock()之后才能再次Lock()。异步调用Lock()，是正当的锁竞争，当然不会有panic了。适用于读写不确定场景，即读写次数没有明显的区别，并且只允许只有一个读或者写的场景，所以该锁也叫做全局锁。
- func (m *Mutex) Unlock()用于解锁m，如果在使用Unlock()前未加锁，就会引起一个运行错误。已经锁定的Mutex并不与特定的goroutine相关联，这样可以利用一个goroutine对其加锁，再利用其他goroutine对其解锁。

建议：同一个互斥锁的成对锁定和解锁操作放在同一层次的代码块中。

使用锁的经典模式：

```
1. var lck sync.Mutex
2. func foo() {
3.     lck.Lock()
4.     defer lck.Unlock()
5.     // ...
6. }
```

lck.Lock()会阻塞直到获取锁，然后利用defer语句在函数返回时自动释放锁。

下面代码通过3个goroutine来体现sync.Mutex 对资源的访问控制特征：

```
1. package main
2.
```

```

3. import (
4.     "fmt"
5.     "sync"
6.     "time"
7. )
8.
9. func main() {
10.    wg := sync.WaitGroup{}
11.
12.    var mutex sync.Mutex
13.    fmt.Println("Locking (G0)")
14.    mutex.Lock()
15.    fmt.Println("locked (G0)")
16.    wg.Add(3)
17.
18.    for i := 1; i < 4; i++ {
19.        go func(i int) {
20.            fmt.Printf("Locking (G%d)\n", i)
21.            mutex.Lock()
22.            fmt.Printf("locked (G%d)\n", i)
23.
24.            time.Sleep(time.Second * 2)
25.            mutex.Unlock()
26.            fmt.Printf("unlocked (G%d)\n", i)
27.            wg.Done()
28.        }(i)
29.    }
30.
31.    time.Sleep(time.Second * 5)
32.    fmt.Println("ready unlock (G0)")
33.    mutex.Unlock()
34.    fmt.Println("unlocked (G0)")
35.
36.    wg.Wait()
37. }

```

1. 程序输出：
2. Locking (G0)
3. locked (G0)
4. Locking (G1)
5. Locking (G3)
6. Locking (G2)

```

7.  ready unlock (G0)
8.  unlocked (G0)
9.  locked (G1)
10. unlocked (G1)
11. locked (G3)
12. locked (G2)
13. unlocked (G3)
14. unlocked (G2)

```

通过程序执行结果我们可以看到，当有锁释放时，才能进行lock动作，G0锁释放时，才有后续锁释放的可能，这里是G1抢到释放机会。

Mutex也可以作为struct的一部分，这样这个struct就会防止被多线程更改数据。

```

1.  package main
2.
3.  import (
4.      "fmt"
5.      "sync"
6.      "time"
7.  )
8.
9.  type Book struct {
10.     BookName string
11.     L         *sync.Mutex
12. }
13.
14. func (bk *Book) SetName(wg *sync.WaitGroup, name string) {
15.     defer func() {
16.         fmt.Println("Unlock set name:", name)
17.         bk.L.Unlock()
18.         wg.Done()
19.     }()
20.
21.     bk.L.Lock()
22.     fmt.Println("Lock set name:", name)
23.     time.Sleep(1 * time.Second)
24.     bk.BookName = name
25. }
26.
27. func main() {
28.     bk := Book{}

```



```

29.     bk.L = new(sync.Mutex)
30.     wg := &sync.WaitGroup{}
31.     books := []string{"《三国演义》", "《道德经》", "《西游记》"}
32.     for _, book := range books {
33.         wg.Add(1)
34.         go bk.SetName(wg, book)
35.     }
36.
37.     wg.Wait()
38. }

```

```

1.  程序输出：
2.
3.  Lock set name: 《西游记》
4.  Unlock set name: 《西游记》
5.  Lock set name: 《三国演义》
6.  Unlock set name: 《三国演义》
7.  Lock set name: 《道德经》
8.  Unlock set name: 《道德经》

```

23.2 读写锁

读写锁是分别针对读操作和写操作进行锁定和解锁操作的互斥锁。在Go语言中，读写锁由结构体类型 `sync.RWMutex` 代表。

基本遵循原则：

- 写锁定情况下，对读写锁进行读锁定或者写锁定，都将阻塞；而且读锁与写锁之间是互斥的；
- 读锁定情况下，对读写锁进行写锁定，将阻塞；加读锁时不会阻塞；
- 对未被写锁定的读写锁进行写解锁，会引发Panic；
- 对未被读锁定的读写锁进行读解锁的时候也会引发Panic；
- 写解锁在进行的同时会试图唤醒所有因进行读锁定而被阻塞的goroutine；
- 读解锁在进行的时候则会试图唤醒一个因进行写锁定而被阻塞的goroutine。

与互斥锁类似，`sync.RWMutex`类型的零值就已经是立即可用的读写锁了。在此类型的方法集合中包含了两对方法，即：

`RWMutex`提供四个方法：

```

1. func (*RWMutex) Lock // 写锁定
2. func (*RWMutex) Unlock // 写解锁
3.
4. func (*RWMutex) RLock // 读锁定
5. func (*RWMutex) RUnlock // 读解锁

```

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sync"
6.     "time"
7. )
8.
9. var m *sync.RWMutex
10.
11. func main() {
12.     wg := sync.WaitGroup{}
13.     wg.Add(20)
14.     var rwMutex sync.RWMutex
15.     Data := 0
16.     for i := 0; i < 10; i++ {
17.         go func(t int) {
18.             rwMutex.RLock()
19.             defer rwMutex.RUnlock()
20.             fmt.Printf("Read data: %v\n", Data)
21.             wg.Done()
22.             time.Sleep(2 * time.Second)
23.             // 这句代码第一次运行后，读解锁。
24.             // 循环到第二个时，读锁定后，这个goroutine就没有阻塞，同时读成功。
25.         }(i)
26.
27.         go func(t int) {
28.             rwMutex.Lock()
29.             defer rwMutex.Unlock()
30.             Data += t
31.             fmt.Printf("Write Data: %v %d \n", Data, t)
32.             wg.Done()
33.
34.             // 这句代码让写锁的效果显示出来，写锁定下是需要解锁后才能写的。
35.             time.Sleep(2 * time.Second)

```

```

36.         }(i)
37.     }
38.     time.Sleep(5 * time.Second)
39.     wg.Wait()
40. }

```

23.3 sync.WaitGroup

前面例子中我们有使用WaitGroup，它用于线程同步，WaitGroup等待一组线程集合完成，才会继续向下执行。主线程(goroutine)调用Add来设置等待的线程(goroutine)数量。然后每个线程(goroutine)运行，并在完成后调用Done。同时，Wait用来阻塞，直到所有线程(goroutine)完成才会向下执行。Add(-1)和Done()效果一致。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sync"
6. )
7.
8. func main() {
9.     var wg sync.WaitGroup
10.    for i := 0; i < 10; i++ {
11.        wg.Add(1)
12.        go func(t int) {
13.            defer wg.Done()
14.            fmt.Println(t)
15.        }(i)
16.    }
17.    wg.Wait()
18. }

```

23.4 sync.Once

sync.Once.Do(f func())能保证once只执行一次,这个sync.Once块只会执行一次。

```

1. package main
2.
3. import (
4.     "fmt"

```

```

5.     "sync"
6.     "time"
7. )
8.
9. var once sync.Once
10.
11. func main() {
12.
13.     for i, v := range make([]string, 10) {
14.         once.Do(onces)
15.         fmt.Println("v:", v, "---i:", i)
16.     }
17.
18.     for i := 0; i < 10; i++ {
19.
20.         go func(i int) {
21.             once.Do(once)
22.             fmt.Println(i)
23.         }(i)
24.     }
25.     time.Sleep(4000)
26. }
27. func onces() {
28.     fmt.Println("onces")
29. }
30. func once() {
31.     fmt.Println("once")
32. }

```

23.5 sync.Map

随着Go1.9的发布，有了一个新的特性，那就是sync.map，它是原生支持并发安全的map。虽然说普通map并不是线程安全（或者说并发安全），但一般情况下我们还是使用它，因为这足够了；只有在涉及到线程安全，再考虑sync.map。

但由于sync.Map的读取并不是类型安全的，所以我们在使用Load读取数据的时候我们需要做类型转换。

sync.Map的使用上和map有较大差异，详情见代码。

```

1. package main
2.

```

```
3. import (
4.     "fmt"
5.     "sync"
6. )
7.
8. func main() {
9.     var m sync.Map
10.
11.     //Store
12.     m.Store("name", "Joe")
13.     m.Store("gender", "Male")
14.
15.     //LoadOrStore
16.     //若key不存在, 则存入key和value, 返回false和输入的value
17.     v, ok := m.LoadOrStore("name1", "Jim")
18.     fmt.Println(ok, v) //false aaa
19.
20.     //若key已存在, 则返回true和key对应的value, 不会修改原来的value
21.     v, ok = m.LoadOrStore("name", "aaa")
22.     fmt.Println(ok, v) //false aaa
23.
24.     //Load
25.     v, ok = m.Load("name")
26.     if ok {
27.         fmt.Println("key存在, 值是: ", v)
28.     } else {
29.         fmt.Println("key不存在")
30.     }
31.
32.     //Range
33.     //遍历sync.Map
34.     f := func(k, v interface{}) bool {
35.         fmt.Println(k, v)
36.         return true
37.     }
38.     m.Range(f)
39.
40.     //Delete
41.     m.Delete("name1")
42.     fmt.Println(m.Load("name1"))
43.
44. }
```

第二十四章 指针和内存

《Go语言四十二章经》第二十四章 指针和内存

作者：李骁

24.1 指针

一个指针变量可以指向任何一个值的内存地址。它指向那个值的内存地址，在 32 位机器上占用 4 个字节，在 64 位机器上占用 8 个字节，并且与它所指向的值的大小无关。当然，可以声明指针指向任何类型的值来表明它的原始性或结构性；你可以在指针类型前面加上*号（前缀）来获取指针所指向的内容，这里的*号是一个类型更改器。使用一个指针引用一个值被称为间接引用。

当一个指针被定义后没有分配到任何变量时，它的值为 nil。

一个指针变量通常缩写为 ptr。

符号 “*” 可以放在一个指针前，如 “*intP”，那么它将得到这个指针指向地址上所存储的值；这被称为反引用（或者内容或者间接引用）操作符；另一种说法是指针转移。

对于任何一个变量 var，如下表达式都是正确的：var == *(&var)

注意事项：

你不能得到一个数字或常量的地址，下面的写法是错误的。

例如：

```
1. const i = 5
2. ptr := &i // error: cannot take the address of i
3. ptr2 := &10 // error: cannot take the address of 10
```

所以说，Go 语言和 C、C++ 以及 D 语言这些低级（系统）语言一样，都有指针的概念。

但是对于经常导致 C 语言内存泄漏继而程序崩溃的指针运算（所谓的指针算法，如：pointer+2，移动指针指向字符串的字节数或数组的某个位置）是不被允许的。

Go 语言中的指针保证了内存安全，更像是 Java、C# 和 VB.NET 中的引用。

因此 c = *p++ 在 Go 语言的代码中是不合法的。

指针的一个高级应用是你传递一个变量的引用（如函数的参数），这样不会传递变量的拷贝。指针

传递是很廉价的，只占用 4 个或 8 个字节。当程序在工作中需要占用大量的内存，或很多变量，或者两者都有，使用指针会减少内存占用和提高效率。被指向的变量也保存在内存中，直到没有任何指针指向它们，所以从它们被创建开始就具有相互独立的生命周期。

另一方面（虽然不太可能），由于一个指针导致的间接引用（一个进程执行了另一个地址），指针的过度频繁使用也会导致性能下降。

指针也可以指向另一个指针，并且可以进行任意深度的嵌套，导致你可以有多级的间接引用，但在大多数情况这会使你的代码结构不清晰。

如我们所见，在大多数情况下 Go 语言可以使程序员轻松创建指针，并且隐藏间接引用，如：自动反向引用。

对一个空指针的反向引用是不合法的，并且会使程序崩溃：

```
1. package main
2. func main() {
3.     var p *int = nil
4.     *p = 0
5. }
```

panic: runtime error: invalid memory address or nil pointer dereference

指针的使用方法：

- 定义指针变量；
- 为指针变量赋值；
- 访问指针变量中指向地址的值；
- 在指针类型前面加上*号来获取指针所指向的内容。

```
1. package main
2.
3. import "fmt"
4.
5. func main() {
6.     var a, b int = 20, 30 // 声明实际变量
7.     var ptra *int         // 声明指针变量
8.     var ptrb *int = &b
9.
10.    ptra = &a // 指针变量的存储地址
11. }
```

```

12.     fmt.Printf("a 变量的地址是: %x\n", &a)
13.     fmt.Printf("b 变量的地址是: %x\n", &b)
14.
15.     // 指针变量的存储地址
16.     fmt.Printf("ptr a 变量的存储地址: %x\n", ptr a)
17.     fmt.Printf("ptr b 变量的存储地址: %x\n", ptr b)
18.
19.     // 使用指针访问值
20.     fmt.Printf("*ptr a 变量的值: %d\n", *ptr a)
21.     fmt.Printf("*ptr b 变量的值: %d\n", *ptr b)
22. }

```

24.2 new() 和 make() 的区别

看起来二者没有什么区别，都在堆上分配内存，但是它们的行为不同，适用于不同的类型。

- new(T) 为每个新的类型T分配一片内存，初始化为 0 并且返回类型*T的内存地址：这种方法返回一个指向类型为 T，值为 0 的地址的指针，它适用于值类型如数组和结构体；它相当于 &T{ }。
- make(T) 返回一个类型为 T 的初始值，它只适用于3种内建的引用类型：切片、map 和 channel。

你并不总是知道变量是分配到栈还是堆上。在C++中，使用new创建的变量总是在堆上。在Go中，即使是使用 new() 或者 make() 函数来分配，变量的位置还是由编译器决定。编译器根据变量的大小和泄露分析的结果来决定其位置。这也意味着在局部变量上返回引用是没问题的，而这在C或者C++这样的语言中是不行的。

如果你想知道变量分配的位置，在“go build”或“go run”上传入“-m” “-gcflags”（即，go run -gcflags -m app.go）。

```

1. go run -gcflags -m main.go
2. # command-line-arguments
3. .\main.go:12:31: m.Alloc / 1024 escapes to heap
4. .\main.go:11:23: main &m does not escape
5. .\main.go:12:12: main ... argument does not escape

```

24.3 垃圾回收和 SetFinalizer

Go 开发者不需要写代码来释放程序中不再使用的变量和结构占用的内存，在 Go 运行时中有一个独立的进程，即垃圾收集器（GC），会处理这些事情，它搜索不再使用的变量然后释放它们的内存。可以通

过 `runtime` 包访问 GC 进程。

通过调用 `runtime.GC()` 函数可以显式的触发 GC，但这只在某些罕见的场景下才有用，比如当内存资源不足时调用 `runtime.GC()`，它会在此函数执行的点上立即释放一大片内存，此时程序可能会有短时的性能下降（因为 GC 进程在执行）。

如果想知道当前的内存状态，可以使用：

```
1. var m runtime.MemStats
2. runtime.ReadMemStats(&m)
3. fmt.Printf("%d Kb\n", m.Alloc / 1024)
```

上面的程序会给出已分配内存的总量，单位是 Kb。进一步的测量参考文档页面。

如果需要在对象 `obj` 被从内存移除前执行一些特殊操作，比如写到日志文件中，可以通过如下方式调用函数来实现：

```
1. runtime.SetFinalizer(obj, func(obj *typeObj))
```

`func(obj *typeObj)` 需要一个 `typeObj` 类型的指针参数 `obj`，特殊操作会在它上面执行。
`func` 也可以是一个匿名函数。

在对象被 GC 进程选中并从内存中移除以前，`SetFinalizer` 都不会执行，即使程序正常结束或者发生错误。

第二十五章 面向对象

《Go语言四十二章经》第二十五章 面向对象

作者：李骁

25.1 Go 中的面向对象

我们总结一下前面看到的：Go 没有类，而是松耦合的类型、方法对接口的实现。

OO 语言最重要的三个方面分别是：封装，继承和多态，在 Go 中它们是怎样表现的呢？

封装（数据隐藏）：

和别的 OO 语言有 4 个或更多的访问层次相比，Go 把它简化为了 2 层：

- 1) 包范围内的：通过标识符首字母小写，对象 只在它所在的包内可见
- 2) 可导出的：通过标识符首字母大写，对象 对所在包以外也可见类型只拥有自己所在包中定义的方法。

继承：

用组合实现，内嵌一个（或多个）包含想要的行为（字段和方法）的类型；多重继承可以通过内嵌多个类型实现。

多态：

用接口实现，某个类型的实例可以赋给它所实现的任意接口类型的变量。类型和接口是松耦合的，并且多重继承可以通过实现多个接口实现。Go 接口不是 Java 和 C# 接口的变体，而且：接口间是不相关的，并且是大规模编程和可适应的演进型设计的关键。

25.2 多重继承

多重继承指的是类型获得多个父类型行为的能力，它在传统的面向对象语言中通常是不被实现的（C++ 和 Python 例外）。因为在类继承层次中，多重继承会给编译器引入额外的复杂度。但是在 Go 语言中，通过在类型中嵌入所有必要的父类型，可以很简单的实现多重继承。

第二十六章 测试

《Go语言四十二章经》第二十六章 测试

作者：李骁

26.1 单元测试

首先所有的包都应该有一定的必要文档，然后同样重要的是对包的测试。

名为 `testing` 的包被专门用来进行自动化测试，日志和错误报告。并且还包含一些基准测试函数的功能。

对一个包做（单元）测试，需要写一些可以频繁（每次更新后）执行的小块测试单元来检查代码的正确性。于是我们必须写一些 Go 源文件来测试代码。测试程序必须属于被测试的包，并且文件名满足这种形式 `*_test.go`，所以测试代码和包中的业务代码是分开的。

`_test` 程序不会被普通的 Go 编译器编译，所以当放应用部署到生产环境时它们不会被部署；只有 `Gotest` 会编译所有的程序：普通程序和测试程序。

测试文件中必须导入“`testing`”包，并写一些名字以 `TestZzz` 打头的全局函数，这里的 `Zzz` 是被测试函数的字母描述，如 `TestFmtInterface`，`TestPayEmployees` 等。

测试函数必须有这种形式的头部：

```
1. func TestAbcde(t *testing.T)
```

`T` 是传给测试函数的结构类型，用来管理测试状态，支持格式化测试日志，如 `t.Log`，`t.Error`，`t.Errorf` 等。在函数的结尾把输出跟想要的结果对比，如果不等就打印一个错误。成功的测试则直接返回。

用下面这些函数来通知测试失败：

```
1) func (t *T) Fail()
```

```
1. 标记测试函数为失败，然后继续执行（剩下的测试）。
```

```
2) func (t *T) FailNow()
```

```
1. 标记测试函数为失败并中止执行；文件中别的测试也被略过，继续执行下一个文件。
```

```
3) func (t *T) Log(args ...interface{})
```

1. `args` 被用默认的格式格式化并打印到错误日志中。

```
4) func (t *T) Fatal(args ...interface{})
```

1. 结合 先执行 3)，然后执行 2) 的效果。

运行 `Go test` 来编译测试程序，并执行程序中的所有 `TestZZZ` 函数。如果所有的测试都通过会打印出 `PASS`。

对不能导出的函数不能进行单元或者基准测试。

`Gotest` 可以接收一个或多个函数程序作为参数，并指定一些选项。

在系统包中，有很多 `_test.go` 结尾的程序，大家可以用来测试，这里我就不写具体例子了。

26.2 基准测试

`testing` 包中有一些类型和函数可以用来做简单的基准测试；测试代码中必须包含以 `BenchmarkZzz` 打头的函数并接收一个 `*testing.B` 类型的参数，比如：

```
1. func BenchmarkReverse(b *testing.B) {
2.     ...
3. }
```

命令 `Go test -test.bench=.*` 会运行所有的基准测试函数；代码中的函数会被调用 `N` 次（`N` 是非常大的数，如 `N = 1000000`），并展示 `N` 的值和函数执行的平均时间，单位为 `ns`（纳秒，`ns/op`）。如果是用 `testing.Benchmark` 调用这些函数，直接运行程序即可。

测试的具体例子

```
1. package even
2. func Even(i int) bool {      // Exported function
3.     return i%2 == 0
4. }
5. func Odd(i int) bool {      // Exported function
6.     return i%2 != 0
7. }
```

在 `even` 包的路径下，我们创建一个名为 `oddeven_test.go` 的测试程序：

```

1. package even
2.
3. import "testing"
4.
5. func TestEven(t *testing.T) {
6.     if !Even(10) {
7.         t.Log(" 10 must be even!")
8.         t.Fail()
9.     }
10.    if Even(7) {
11.        t.Log(" 7 is not even!")
12.        t.Fail()
13.    }
14. }
15.
16. func BenchmarkOdd(b *testing.B) {
17.
18.     for i := 0; i < b.N; i++ {
19.         Odd(1155555555555555)
20.     }
21. }

```

现在我们可以用命令：Go test -test.bench=.* (或 make test) 来测试 even 包。

```

1. 输出：
2.
3. goos: windows
4. goarch: amd64
5. pkg: ind/even
6. BenchmarkOdd-4      2000000000      0.39 ns/op
7. PASS
8. ok      ind/even    4.785s

```

26.3 分析并优化 Go 程序

时间和内存消耗

如果代码使用了 Go 中 testing 包的基准测试功能，我们可以用 Gotest 标准的 -cpuprofile 和 -memprofile 标志向指定文件写入 CPU 或 内存使用情况报告。

使用方式：

```
1. go test -x -v -test.cpuprofile=pprof.out
```

执行上面代码，执行结果 pprof.out 文件中写入 cpu 性能分析信息。

26.4 用 pprof 调试

要监控Go程序的堆栈，cpu的耗时等性能信息，我们可以通过使用pprof包来实现。

pprof包有两种方式导入：

```
1. "net/http/pprof"
2. "runtime/pprof"
```

其实net/http/pprof中只是使用runtime/pprof包来进行封装了一下，并在http端口上暴露出来，让我们可以在浏览器查看程序的性能分析。我们可以自行查看net/http/pprof中代码，只有一个文件pprof.go。

下面我们具体说说怎么使用pprof，首先我们讲讲在开发中取得pprof信息的三种方式：

一：web 服务器程序

如果我们的Go程序是用http包启动的web服务器，你想查看自己的web服务器的状态。这个时候就可以选择net/http/pprof。你只需要引入包_ "net/http/pprof"，然后就可以在浏览器中使用<http://localhost:port/debug/pprof/>直接看到当前web服务的状态，包括CPU占用情况和内存使用情况等。

这里port是8080，也就是我们web服务器监听的端口。

```
1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6.     _ "net/http/pprof" // 为什么用_，在讲解http包时有解释。
7. )
8.
9. func myfunc(w http.ResponseWriter, r *http.Request) {
10.     fmt.Fprintf(w, "hi")
11. }
12.
13. func main() {
14.     http.HandleFunc("/", myfunc)
15.     http.ListenAndServe(":8080", nil)
```

```
16. }
```

二：服务进程

如果你的Go程序不是web服务器，而是一个服务进程，可以选择使用net/http/pprof包，然后开启一个goroutine来监听相应端口。

```
1. package main
2.
3. import (
4.     "fmt"
5.     "log"
6.     "net/http"
7.     _ "net/http/pprof"
8.
9.     "time"
10. )
11.
12. func main() {
13.     // 开启pprof
14.     go func() {
15.         log.Println(http.ListenAndServe("localhost:8080", nil))
16.     }()
17.     go hello()
18.     select {}
19. }
20. func hello() {
21.     for {
22.         go func() {
23.             fmt.Println("hello word")
24.         }()
25.         time.Sleep(time.Millisecond * 1)
26.     }
27. }
```

在前面这两种方式中，我们在命令行分别运行以下命令：

利用这个命令查看堆栈信息：

go tool pprof <http://localhost:8080/debug/pprof/heap>

利用这个命令可以查看程序CPU使用情况信息：

```
go tool pprof http://localhost:8080/debug/pprof/profile
```

使用这个命令可以查看block信息：

```
go tool pprof http://localhost:8080/debug/pprof/block
```



这里需要先安装graphviz, <http://www.graphviz.org/download/> , windows平台直接下载zip包, 解压缩后把bin目录放到\$path中。我们可以通过命令 png 产生图片, 还有svg, gif, pdf等命令, 生成的图片自动命名存放在当前目录下, 我们这里生成了png。其他命令使用可通过help查看。

三：应用程序

如果你的go程序只是一个应用程序, 那么你就不能使用net/http/pprof包了, 你就需要使用到runtime/pprof。比如下面的例子：

```
1. package main
2.
3. import (
4.     "flag"
5.     "fmt"
6.     "log"
7.
8.     "os"
9.     "runtime/pprof"
10.    "time"
11. )
12.
13. var cpuprofile = flag.String("cpuprofile", "", "write cpu profile to file")
14.
15. func main() {
16.     flag.Parse()
17.     if *cpuprofile != "" {
18.         f, err := os.Create(*cpuprofile)
19.         if err != nil {
20.             log.Fatal(err)
21.         }
22.         pprof.StartCPUProfile(f)
23.         defer pprof.StopCPUProfile()
24.     }
25.
```



```

26.     go hello()
27.     time.Sleep(10 * time.Second)
28. }
29. func hello() {
30.     for {
31.         go func() {
32.             fmt.Println("hello word")
33.         }()
34.         time.Sleep(time.Millisecond * 1)
35.     }
36. }

```

编译后运行：

```
1. study.exe --cpuprofile=cpu.prof
```

这里我们编译后可执行程序是study.exe，程序运行完后的cpu信息就会记录到cpu.prof中。

现在有了cpu.prof 文件，我们就可以通过go tool pprof 来看相应的信息了。在命令行运行：

```
1. go tool pprof study.exe cpu.prof
```

这里要注意的是需要带上可执行的程序名以及prof信息文件。

命令执行后会进入到：

界面和前面两种使用net/http/pprof包 一样。我们可以通过go tool pprof 生svg, png或者是pdf文件。

这是生成的png文件，和前面生成的png类似，前面我们生成的是block信息：

通过上面这三种情况的分析，我们可以知道，其实就是两种情况：go tool pprof 后面可以使用<http://localhost:8080/debug/pprof/profile> 这种url方式，也可以使用study.exe cpu.prof 这种文件方式来进行分析。可以根据你的项目情况灵活使用。

有关pprof，我们就讲这么多，在实际项目中，我们多使用就会发现这个工具还是蛮有用处的。

第二十七章 反射(reflect)

《Go语言四十二章经》第二十七章 反射(reflect)

作者：李骁

27.1 反射(reflect)

反射是用程序检查其所拥有的结构，尤其是类型的一种能力；这是元编程的一种形式。

反射可以在运行时检查类型和变量，例如它的大小、方法和 动态 的调用这些方法。这对于没有源代码的包尤其有用。

这是一个强大的工具，除非真得有必要，否则应当避免使用或小心使用。

变量的最基本信息就是类型和值。反射包的 `Type` 用来表示一个 Go 类型，反射包的 `Value` 为 Go 值提供了反射接口。

两个简单的函数，`reflect.TypeOf` 和 `reflect.ValueOf`，返回被检查对象的类型和值。

例如，`x` 被定义为：`var x float64 = 3.4`，那么 `reflect.TypeOf(x)` 返回 `float64`，`reflect.ValueOf(x)` 返回

实际上，反射是通过检查一个接口的值，变量首先被转换成空接口。这从下面两个函数签名能够很明显的看出来：

```
1. func TypeOf(i interface{}) Type
2. func ValueOf(i interface{}) Value
```

接口的值包含一个 `type` 和 `value`。

反射可以从接口值反射到对象，也可以从对象反射回接口值。

`reflect.Type` 和 `reflect.Value` 都有许多方法用于检查和操作它们。一个重要的例子是 `Value` 有一个 `Type` 方法返回 `reflect.Value` 的 `Type`。另一个是 `Type` 和 `Value` 都有 `Kind` 方法返回一个常量来表示类型：`Uint`、`Float64`、`Slice` 等等。同样 `Value` 有叫做 `Int` 和 `Float` 的方法可以获取存储在内部的值（跟 `int64` 和 `float64` 一样）

问题的原因是 `v` 不是可设置的（这里并不是说值不可寻址）。是否可设置是 `Value` 的一个属性，并且不是所有的反射值都有这个属性：可以使用 `CanSet()` 方法测试是否可设置。反射中有些内容是需要用地址去改变它的状态的。

当 `v := reflect.ValueOf(x)` 函数通过传递一个 `x` 拷贝创建了 `v`，那么 `v` 的改变并不能更改原始的 `x`。要想 `v` 的更改能作用到 `x`，那就必须传递 `x` 的地址 `v = reflect.ValueOf(&x)`。

通过 `Type()` 我们看到 `v` 现在的类型是 `*float64` 并且仍然是不可设置的。

要想让其可设置我们需要使用 `Elem()` 函数，这间接的使用指针：`v = v.Elem()`

现在 `v.CanSet()` 返回 `true` 并且 `v.SetFloat(3.1415)` 设置成功了！

27.2 反射结构体

有些时候需要反射一个结构类型。下面例子较为完整反射了一个结构体的字段和方法：

```

1. package main
2.
3. import (
4.     "fmt"
5.     "reflect"
6. )
7.
8. // 结构体
9. type ss struct {
10.     int
11.     string
12.     bool
13.     float64
14. }
15.
16. func (s ss) Method1(i int) string { return "结构体方法1" }
17. func (s *ss) Method2(i int) string { return "结构体方法2" }
18.
19. var (
20.     structValue = ss{ // 结构体
21.         20,
22.         "结构体",
23.         false,
24.         64.0,
25.     }
26. )
27.
28. // 复杂类型
29. var complexTypes = []interface{}{
30.     structValue, &structValue, // 结构体
31.     structValue.Method1, structValue.Method2, // 方法

```

```

32. }
33.
34. func main() {
35.     // 测试复杂类型
36.     for i := 0; i < len(complexTypes); i++ {
37.         PrintInfo(complexTypes[i])
38.     }
39. }
40.
41. func PrintInfo(i interface{}) {
42.     if i == nil {
43.         fmt.Println("-----")
44.         fmt.Printf("无效接口值:%v\n", i)
45.         fmt.Println("-----")
46.         return
47.     }
48.     v := reflect.ValueOf(i)
49.     PrintValue(v)
50. }
51.
52. func PrintValue(v reflect.Value) {
53.     fmt.Println("-----")
54.     // ----- 通用方法 -----
55.     fmt.Println("String           :", v.String()) // 反射值的字符串形式
56.     fmt.Println("Type           :", v.Type())    // 反射值的类型
57.     fmt.Println("Kind           :", v.Kind())    // 反射值的类别
58.     fmt.Println("CanAddr        :", v.CanAddr()) // 是否可以获取地址
59.     fmt.Println("CanSet         :", v.CanSet())  // 是否可以修改
60.     if v.CanAddr() {
61.         fmt.Println("Addr           :", v.Addr())      // 获取地址
62.         fmt.Println("UnsafeAddr      :", v.UnsafeAddr()) // 获取自由地址
63.     }
64.     // 获取方法数量
65.     fmt.Println("NumMethod        :", v.NumMethod())
66.     if v.NumMethod() > 0 {
67.         // 遍历方法
68.         i := 0
69.         for ; i < v.NumMethod()-1; i++ {
70.             fmt.Printf("    ┆ %v\n", v.Method(i).String())
71.             //                if i >= 4 { // 只列举 5 个
72.             //                fmt.Println("    ┆ ...")
73.             //                break

```

```

74.         //          }
75.     }
76.     fmt.Printf("    L %v\n", v.Method(i).String())
77.     // 通过名称获取方法
78.     fmt.Println("MethodByName          :", v.MethodByName("String").String())
79. }
80.
81. switch v.Kind() {
82. // 结构体 :
83. case reflect.Struct:
84.     fmt.Println("=== 结构体 ===")
85.     // 获取字段个数
86.     fmt.Println("NumField          :", v.NumField())
87.     if v.NumField() > 0 {
88.         var i int
89.         // 遍历结构体字段
90.         for i = 0; i < v.NumField()-1; i++ {
91.             field := v.Field(i) // 获取结构体字段
92.             fmt.Printf("    | %-8v %v\n", field.Type(), field.String())
93.         }
94.         field := v.Field(i) // 获取结构体字段
95.         fmt.Printf("    L %-8v %v\n", field.Type(), field.String())
96.         // 通过名称查找字段
97.         if v := v.FieldByName("ptr"); v.IsValid() {
98.             fmt.Println("FieldByName(ptr)  :", v.Type().Name())
99.         }
100.        // 通过函数查找字段
101.        v := v.FieldByNameFunc(func(s string) bool { return len(s) > 3 })
102.        if v.IsValid() {
103.            fmt.Println("FieldByNameFunc    :", v.Type().Name())
104.        }
105.    }
106. }
107. }

```

第二十八章 unsafe包

《Go语言四十二章经》第二十八章 unsafe包

作者：李骁

28.1 unsafe 包

```
1. func Alignof(x ArbitraryType) uintptr
2. func Offsetof(x ArbitraryType) uintptr
3. func Sizeof(x ArbitraryType) uintptr
4. type ArbitraryType int
5. type Pointer *ArbitraryType
```

这个包中，只提供了3个函数，两个类型。就这么少的量，却有着超级强悍的功能。学过C语言的都可能比较清楚，通过指针，知道变量在内存中占用的字节数，就可以通过指针加偏移量的操作，在地址中，修改，访问变量的值。在Go 语言中，怎么去实现这么疯狂的操作呢？就得靠unsafe包了。

ArbitraryType 是int的一个别名，但是Go 语言中，对ArbitraryType赋予了特殊的意义，千万不要死磕这个后边的int类型。通常，我们把interface{}看作是任意类型，那么ArbitraryType这个类型，在Go 语言系统中，比interface{}还要随意。

Pointer 是int指针类型的一个别名，在Go 语言系统中，可以把Pointer类型，理解成通用指针类型，用于转换不同类型指针。

Go 语言的指针类型长度与int类型长度，在内存中占用的字节数是一样的。ArbitraryType类型的变量也可以是指针。所以，千万不要死磕type后边的那个int。

```
1. func Alignof(x ArbitraryType) uintptr
2. func Offsetof(x ArbitraryType) uintptr
3. func Sizeof(x ArbitraryType) uintptr
```

通过分析发现，这三个函数的参数均是ArbitraryType类型，就是接受任何类型的变量。

1. Alignof返回变量对齐字节数量
2. Offsetof返回变量指定属性的偏移量，这个函数虽然接收的是任何类型的变量，但是这个又一个前提，就是变量要是个struct类型，且还不能直接将这个struct类型的变量当作参数，只能将这个struct类型变量的属性当作参数。
3. Sizeof 返回变量在内存中占用的字节数，切记，如果是slice，则不会返回这个slice在内存

中的实际占用长度。

unsafe中，通过这两个兼容万物的类型，将其他类型都转换过来，然后通过这三个函数，分别能取长度，偏移量，对齐字节数，就可以在内存地址映射中，来回游走。

28.2 指针运算

uintptr这个类型，在Go 语言中，字节长度也是与int一致。通常Pointer不能参与运算，比如你要在某个指针地址上加上一个偏移量，Pointer是不能做这个运算的，那么谁可以呢？

1. 就是uintptr类型了，只要将Pointer类型转换成uintptr类型，做完加减法后，转换成Pointer，通过*操作，取值，修改值。

unsafe.Pointer其实就是类似C的void *，在Go 语言中是用于各种指针相互转换的桥梁，可以让任意类型的指针实现相互转换，也可以将任意类型的指针转换为 uintptr 进行指针运算。

uintptr是Go 语言的内置类型，是能存储指针的整型， uintptr 的底层类型是int，它和unsafe.Pointer可相互转换。

uintptr和unsafe.Pointer的区别就是：

- unsafe.Pointer只是单纯的通用指针类型，用于转换不同类型指针，它不可以参与指针运算；
- 而uintptr是用于指针运算的，GC 不把 uintptr 当指针，也就是说 uintptr 无法持有对象， uintptr 类型的目标会被回收；
- unsafe.Pointer 可以和 普通指针 进行相互转换；
- unsafe.Pointer 可以和 uintptr 进行相互转换。

Go 语言的unsafe包很强大，基本上很少会去用它。它可以像C一样去操作内存，但由于Go 语言不支持直接进行指针运算，所以用起来稍显麻烦。

uintptr和intptr是无符号和有符号的指针类型，并且确保在64位平台上是8个字节，在32位平台上是4个字节，uintptr主要用于Go 语言中的指针运算。

通过unsafe包来实现对V的成员i和j赋值，然后通过PutI()和PutJ()来打印观察输出结果。

以下是main.Go源代码：

```
1. package main
2.
3. import (
4.     "fmt"
```

```

5.     "unsafe"
6. )
7.
8. type V struct {
9.     i int32
10.    j int64
11. }
12.
13. func (this V) PutI() {
14.    fmt.Printf("i=%d\n", this.i)
15. }
16. func (this V) PutJ() {
17.    fmt.Printf("j=%d\n", this.j)
18. }
19.
20. func main() {
21.    var v *V = new(V)
22.    var i *int32 = (*int32)(unsafe.Pointer(v))
23.    *i = int32(98)
24.    var j *int64 = (*int64)(unsafe.Pointer(uintptr(unsafe.Pointer(v)) +
    uintptr(unsafe.Sizeof(int32(0)))))
25.    *j = int64(763)
26.    v.PutI()
27.    v.PutJ()
28. }

```

1. 程序输出：
2. i=98
3. j=0

需要知道结构体V的成员布局，要修改的成员大小以及成员的偏移量。我们的核心思想就是：结构体的成员在内存中的分配是一段连续的内存，结构体中第一个成员的地址就是这个结构体的地址，您也可以认为是相对于这个结构体偏移了0。相同的，这个结构体中的任一成员都可以相对于这个结构体的偏移来计算出它在内存中的绝对地址。

具体来讲解下main方法的实现：

1. `var v *V = new(V)`

`new`是Go 语言的内置方法，用来分配一段内存(会按类型的零值来清零)，并返回一个指针。所以v就是类型为V的一个指针。


```
1. var i *int32 = (*int32)(unsafe.Pointer(v))
```

将指针v转成通用指针，再转成int32指针。这里就看到了unsafe.Pointer的作用了，您不能直接将v转成int32类型的指针，那样将会panic。刚才说了v的地址其实就是它的第一个成员的地址，所以这个i就很显然指向了v的成员i，通过给i赋值就相当于给v.i赋值了，但是别忘了i只是个指针，要赋值得解引用。

```
1. *i = int32(98)
```

现在已经成功的改变了v的私有成员i的值。

但是对于v.j来说，怎么来得到它在内存中的地址呢？其实我们可以获取它相对于v的偏移量(unsafe.Sizeof可以为我们做这个事)，但我上面的代码并没有这样去实现。各位别急，一步步来。

```
1. var j *int64 = (*int64)(unsafe.Pointer(uintptr(unsafe.Pointer(v)) +
    uintptr(unsafe.Sizeof(int32(0)))))
```

其实我们已经知道v是有两个成员的，包括i和j，并且在定义中，i位于j的前面，而i是int32类型，也就是说i占4个字节。所以j是相对于v偏移了4个字节。您可以用uintptr(4)或uintptr(unsafe.Sizeof(int32(0)))来做这个事。unsafe.Sizeof方法用来得到一个值应该占用多少个字节空间。注意这里跟C的用法不一样，C是直接传入类型，而Go语言是传入值。之所以转成uintptr类型是因为需要做指针运算。v的地址加上j相对于v的偏移地址，也就得到了v.j在内存中的绝对地址，别忘了j的类型是int64，所以现在的j就是一个指向v.j的指针，接下来给它赋值：

```
1. *j = int64(763)
```

上面结构体V中，定义了2个成员属性，如果我们定义一个byte类型的成员属性。我们来看下它的输出：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "unsafe"
6. )
7.
8. type V struct {
9.     b byte
10.    i int32
11.    j int64
12. }
```

```

13.
14. func (this V) PutI() {
15.     fmt.Printf("i=%d\n", this.i)
16. }
17. func (this V) PutJ() {
18.     fmt.Printf("j=%d\n", this.j)
19. }
20.
21. func main() {
22.     var v *V = new(V)
23.
24.     fmt.Printf("size=%d\n", unsafe.Sizeof(*v))
25.     var i *int32 = (*int32)(unsafe.Pointer(v))
26.     *i = int32(98)
27.     var j *int64 = (*int64)(unsafe.Pointer(uintptr(unsafe.Pointer(v)) +
    uintptr(unsafe.Sizeof(int32(0)))))
28.     *j = int64(763)
29.     v.PutI()
30.     v.PutJ()
31. }

```

1. 程序输出：
2. size=16
3. i=763
4. j=0

size=16，好像跟我们想像的不一致。来手动计算一下：b是byte类型，占1个字节；i是int32类型，占4个字节；j是int64类型，占8个字节，1+4+8=13。这是怎么回事呢？

这是因为发生了对齐。在struct中，它的对齐值是它的成员中的最大对齐值。

每个成员类型都有它的对齐值，可以用unsafe.Alignof方法来计算，比如unsafe.Alignof(v.b)就可以得到b的对齐值为1。但这个对齐值是其值类型的长度或引用的地址长度（32位或者64位），和其在结构体中的size不是简单相加的问题。经过在64位机器上测试，发现地址（uintptr）如下：

1. unsafe.Pointer(b): %s 825741058384
2. unsafe.Pointer(i): %s 825741058388
3. unsafe.Pointer(j): %s 825741058392

可以初步推断，也经过测试验证，第二个代码中，取i值使用uintptr(4)是准确的。至于size其实也和对齐值有关，也不是简单相加每个字段的长度。

`unsafe.Offsetof` 可以在实际中使用，如果改变私有的字段，需要程序员认真考虑后，按照上面的方法仔细确认好对齐值。

第二十九章 排序(sort)

《Go语言四十二章经》第二十九章 排序(sort)

作者：李骁

29.1 sort包介绍

Go语言标准库sort包中实现了3种基本的排序算法：插入排序、快排和堆排序。和其他语言中一样，这三种方式都是不公开的，他们只在sort包内部使用。所以用户在使用sort包进行排序时无需考虑使用那种排序方式，sort.Interface定义的三个方法：获取数据集合长度的Len()方法、比较两个元素大小的Less()方法和交换两个元素位置的Swap()方法，就可以顺利对数据集合进行排序。sort包会根据实际数据自动选择高效的排序算法。

```

1. type Interface
2. type Interface interface {
3.
4.     Len() int    // Len 为集合内元素的总数
5.     Less(i, j int) bool //如果index为i的元素小于index为j的元素，则返回true，否则
        false
6.     Swap(i, j int) // Swap 交换索引为 i 和 j 的元素
7. }
```

sort包里面已经实现了[]int, []float64, []string的排序。

任何实现了 sort.Interface 的类型（一般为集合），均可使用该包中的方法进行排序。这些方法要求集合内列出元素的索引为整数。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sort"
6. )
7.
8. func main() {
9.     a := []int{3, 5, 4, -1, 9, 11, -14}
10.    sort.Ints(a)
11.    fmt.Println(a)
```

```

12.     ss := []string{"surface", "ipad", "mac pro", "mac air", "think pad", "idea
      pad"}
13.     sort.Strings(ss)
14.     fmt.Println(ss)
15.     sort.Sort(sort.Reverse(sort.StringSlice(ss)))
16.     fmt.Printf("After reverse: %v\n", ss)
17. }

```

1. 程序输出：
2. [-14 -1 3 4 5 9 11]
3. [idea pad ipad mac air mac pro surface think pad]
4. After reverse: [think pad surface mac pro mac air ipad idea pad]

默认结果都是升序排列，如果我们想对一个 sortable object 进行逆序排序，可以自定义一个 type。但 sort.Reverse 帮你省掉了这些代码。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sort"
6. )
7.
8. func main() {
9.     a := []int{4, 3, 2, 1, 5, 9, 8, 7, 6}
10.    sort.Sort(sort.Reverse(sort.IntSlice(a)))
11.    fmt.Println("After reversed: ", a)
12. }

```

1. 程序输出：
2. After reversed: [9 8 7 6 5 4 3 2 1]

相关方法：

1. // 将类型为float64的slice以升序方式排序
2. func Float64s(a []float64)
- 3.
4. // 判定是否已经进行排序func Ints(a []int)
5. func Float64sAreSorted(a []float64) bool
- 6.
7. // Ints 以升序排列 int 切片。

```

8. func Ints(a []int)
9.
10. // 判断 int 切片是否已经按升序排列。
11. func IntsAreSorted(a []int) bool
12.
13. //IsSorted 判断数据是否已经排序。包括各种可sort的数据类型的判断。
14. func IsSorted(data Interface) bool
15.
16.
17. //Strings 以升序排列 string 切片。
18. func Strings(a []string)
19.
20. //判断 string 切片是否按升序排列
21. func StringsAreSorted(a []string) bool
22.
23. // search使用二分法进行查找，Search()方法回使用“二分查找”算法来搜索某指定切片[0:n]，
24. // 并返回能够使f(i)=true的最小的i (0<=i<n) 值，并且会假定，如果f(i)=true，则
    f(i+1)=true，
25. // 即对于切片[0:n]，i之前的切片元素会使f()函数返回false，i及i之后的元素会使f()
26. // 函数返回true。但是，当在切片中无法找到时f(i)=true的i时（此时切片元素都不能使f()
27. // 函数返回true），Search()方法会返回n（而不是返回-1）。
28. //
29. // Search 常用于在一个已排序的，可索引的数据结构中寻找索引为 i 的值 x，例如数组或切片。
30. // 这种情况下实参 f一般是一个闭包，会捕获所要搜索的值，以及索引并排序该数据结构的方式。
31. func Search(n int, f func(int) bool) int
32.
33. // SearchFloat64s 在float64s切片中搜索x并返回索引如Search函数所述。
34. // 返回可以插入x值的索引位置，如果x不存在，返回数组a的长度切片必须以升序排列
35. func SearchFloat64s(a []float64, x float64) int
36.
37. // SearchInts 在ints切片中搜索x并返回索引如Search函数所述。返回可以插入x值的
38. // 索引位置，如果x不存在，返回数组a的长度切片必须以升序排列
39. func SearchInts(a []int, x int) int
40.
41. // SearchFloat64s 在strings切片中搜索x并返回索引如Search函数所述。返回可以
42. // 插入x值的索引位置，如果x不存在，返回数组a的长度切片必须以升序排列
43. func SearchStrings(a []string, x string) int
44.
45. // 其中需要注意的是，以上三种search查找方法，其对应的slice必须按照升序进行排序，
46. // 否则会出现奇怪的结果。
47.
48. // Sort 对 data 进行排序。它调用一次 data.Len 来决定排序的长度 n，调用 data.Less

```

```

49. // 和 data.Swap 的开销为 $O(n \cdot \log(n))$ 。此排序为不稳定排序。他根据不同形式决定使用
50. // 不同的排序方式（插入排序，堆排序，快排）。
51. func Sort(data Interface)
52.
53. // Stable对data进行排序，不过排序过程中，如果data中存在相等的元素，则他们原来的
54. // 顺序不会改变，即如果有两个相等元素num，他们的初始index分别为i和j，并且 $i < j$ ，
55. // 则利用Stable对data进行排序后，i依然小于j。直接利用sort进行排序则不能够保证这一点。
56. func Stable(data Interface)

```

29.2 自定义sort.Interface排序

如果是具体的某个结构体的排序，就需要自己实现Interface了。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sort"
6. )
7.
8. type person struct {
9.     Name string
10.    Age  int
11. }
12.
13. type personSlice []person
14.
15. func (s personSlice) Len() int           { return len(s) }
16. func (s personSlice) Swap(i, j int)      { s[i], s[j] = s[j], s[i] }
17. func (s personSlice) Less(i, j int) bool { return s[i].Age < s[j].Age }
18.
19. func main() {
20.     a := personSlice{
21.         {
22.             Name: "AAA",
23.             Age:  55,
24.         },
25.         {
26.             Name: "BBB",
27.             Age:  22,
28.         },

```

```

29.     {
30.         Name: "CCC",
31.         Age:  0,
32.     },
33.     {
34.         Name: "DDD",
35.         Age:  22,
36.     },
37.     {
38.         Name: "EEE",
39.         Age:  11,
40.     },
41. }
42. sort.Sort(a)
43. fmt.Println("Sort:", a)
44.
45. sort.Stable(a)
46. fmt.Println("Stable:", a)
47.
48. }

```

1. 程序输出：
- 2.
3. Sort: [{CCC 0} {EEE 11} {BBB 22} {DDD 22} {AAA 55}]
4. Stable: [{CCC 0} {EEE 11} {BBB 22} {DDD 22} {AAA 55}]

29.3 sort.Slice

利用sort.Slice 函数，而不用提供一个特定的 sort.Interface 的实现，而是 Less(i, j int) 作为一个比较回调函数，可以简单地传递给 sort.Slice 进行排序。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "sort"
6. )
7.
8. type Peak struct {
9.     Name      string
10.    Elevation int // in feet

```



```

11. }
12.
13. func main() {
14.     peaks := []Peak{
15.         {"Aconcagua", 22838},
16.         {"Denali", 20322},
17.         {"Kilimanjaro", 19341},
18.         {"Mount Elbrus", 18510},
19.         {"Mount Everest", 29029},
20.         {"Mount Kosciuszko", 7310},
21.         {"Mount Vinson", 16050},
22.         {"Puncak Jaya", 16024},
23.     }
24.
25.     // does an in-place sort on the peaks slice, with tallest peak first
26.     sort.Slice(peaks, func(i, j int) bool {
27.         return peaks[i].Elevation >= peaks[j].Elevation
28.     })
29.     fmt.Println(peaks)
30.
31. }

```

1. 程序输出：
2. [{Mount Everest 29029} {Aconcagua 22838} {Denali 20322} {Kilimanjaro 19341}
{Mount Elbrus 18510} {Mount Vinson 16050} {Puncak Jaya 16024} {Mount Kosciuszko
7310}]

第三十章 OS包

《Go语言四十二章经》第三十章 OS包

作者：李骁

30.1 启动外部命令和程序

os 包有一个 `StartProcess` 函数可以调用或启动外部系统命令和二进制可执行文件；它的第一个参数是要运行的进程，第二个参数用来传递选项或参数，第三个参数是含有系统环境基本信息的结构体。

这个函数返回被启动进程的 `id (pid)`，或者启动失败返回错误。

exec 包中也有同样功能的更简单的结构体和函数；主要是 `exec.Command(name string, arg ... string)` 和 `Run()`。首先需要用系统命令或可执行文件的名字创建一个 `Command` 对象，然后用这个对象作为接收器调用 `Run()`。下面的程序（因为是执行 Linux 命令，只能在 Linux 下面运行）演示了它们的使用：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "os"
6. )
7.
8. func main() {
9.     // 1) os.StartProcess //
10.    /* Linux: */
11.    env := os.Environ()
12.    procAttr := &os.ProcAttr{
13.        Env: env,
14.        Files: []*os.File{
15.            os.Stdin,
16.            os.Stdout,
17.            os.Stderr,
18.        },
19.    }
20.    // 1st example: list files
21.    Pid, err := os.StartProcess("/bin/ls", []string{"ls", "-l"}, procAttr)
```

```

23.     if err != nil {
24.         fmt.Printf("Error %v starting process!", err) //
25.         os.Exit(1)
26.     }
27.     fmt.Printf("The process id is %v", pid)
28. }

```

30.2 os/signal 信号处理

信号处理在平滑重启，进程管理的中有着重要实际用途：

```

1.  package main
2.
3.  import (
4.      "fmt"
5.      "os"
6.      "os/signal"
7.      "syscall"
8.      "time"
9.  )
10.
11. func main() {
12.     go signalListen()
13.     for {
14.         time.Sleep(10 * time.Second)
15.     }
16. }
17.
18. func signalListen() {
19.     c := make(chan os.Signal)
20.     signal.Notify(c, syscall.SIGUSR2)
21.     for {
22.         s := <-c
23.         //收到信号后的处理，这里只是输出信号内容，可以做一些更有意思的事
24.         fmt.Println("get signal:", s)
25.     }
26. }

```

关于信号有关信息，有兴趣建议可以参考《Unix高级编程》。os包中其他的功能我就不一一介绍了。

第三十一章 文件操作与IO

《Go语言四十二章经》第三十一章 文件操作与IO

作者：李骁

31.1 文件系统

对于文件和目录的操作，Go主要在os 提供了的相应函数：

```

1. func Mkdir(name string, perm FileMode) error
2. func Chdir(dir string) error
3. func TempDir() string
4. func Rename(oldpath, newpath string) error
5. func Chmod(name string, mode FileMode) error
6. func Open(name string) (*File, error) {
7.     return OpenFile(name, O_RDONLY, 0)
8. }
9. func Create(name string) (*File, error) {
10.    return OpenFile(name, O_RDWR|O_CREATE|O_TRUNC, 0666)
11. }
12. func OpenFile(name string, flag int, perm FileMode) (*File, error) {
13.    testlog.Open(name)
14.    return openFileNoLog(name, flag, perm)
15. }
```

从上面函数定义中我们可以发现一个情况：那就是os中不同函数打开（创建）文件的操作，最终还是通过OpenFile来实现，而OpenFile由编译器根据系统的情况来选择不同的底层功能来实现，对这个实现细节有兴趣可以根据标准包来仔细了解，这里就不展开讲了。

1. os.Open(name string) 使用只读模式打开文件；
2. os.Create(name string) 创建新文件，如文件存在则原文件内容会丢失；
3. os.OpenFile(name string, flag int, perm FileMode) 这个函数可以指定flag和FileMode。这三个函数都会返回一个文件对象。

1. Flag :
- 2.
3. O_RDONLY int = syscall.O_RDONLY // 只读打开文件和os.Open()同义
4. O_WRONLY int = syscall.O_WRONLY // 只写打开文件

```

5.  O_RDWR   int = syscall.O_RDWR    // 读写方式打开文件
6.  O_APPEND  int = syscall.O_APPEND  // 当写的时候使用追加模式到文件末尾
7.  O_CREATE  int = syscall.O_CREAT   // 如果文件不存在，此案创建
8.  O_EXCL    int = syscall.O_EXCL    // 和O_CREATE一起使用，只有当文件不存在时才创建
9.  O_SYNC    int = syscall.O_SYNC    // 以同步I/O方式打开文件，直接写入硬盘
10. O_TRUNC   int = syscall.O_TRUNC   // 如果可以的话，当打开文件时先清空文件

```

在ioutil包中，也可以对文件操作，主要有下面三个函数：

```

1. func ReadFile(filename string) ([]byte, error) // f, err := os.Open(filename)
2. func WriteFile(filename string, data []byte, perm os.FileMode) error
   //os.OpenFile
3. func ReadDir(dirname string) ([]os.FileInfo, error) // f, err :=
   os.Open(dirname)

```

这三个函数涉及到了文件IO，而对文件的操作我们除了打开（创建），关闭外，更主要的是对内容的读写操作上，也即是文件IO处理上。在Go语言中，对于IO的操作在Go语言很多标准库中存在，很难完整地讲清楚。下面我就尝试结合io，ioutil，bufio这三个标准库，讲一讲这几个标准库在文件IO操作中的具体使用方法。

31.2 IO读写

Go语言中，为了方便开发者使用，将IO操作封装在了大概如下几个包中：

- io 为 IO 原语 (I/O primitives) 提供基本的接口
- io/ioutil 封装一些实用的 I/O 函数
- fmt 实现格式化 I/O，类似 C 语言中的 printf 和 scanf，后面会详细讲解
- bufio 实现带缓冲I/O

在 io 包中最重要的是两个接口：Reader 和 Writer 接口。

这两个接口是我们了解整个IO的关键，我们只要记住：实现了这两个接口，就有了 **IO** 的功能。

有关缓冲：

- 内核中的缓冲：无论进程是否提供缓冲，内核都是提供缓冲的，系统对磁盘的读写都会提供一个缓冲（内核高速缓冲），将数据写入到块缓冲进行排队，当块缓冲达到一定的量时，才把数据写入磁盘。
- 进程中的缓冲：是指对输入输出流进行了改进，提供了一个流缓冲，当调用一个函数向磁盘写数据时，先把数据写入缓冲区，当达到某个条件，如流缓冲满了，或刷新流缓冲，这时候才会把数据一次送往内核提供的块缓冲中，再经块化重写入磁盘。

Go 语言提供了很多读写文件的方式，一般来说常用的有三种。

一：os.File 实现了Reader 和 Writer 接口，所以在文件对象上，我们可以直接读写文件。

```
1. func (f *File) Read(b []byte) (n int, err error)
2. func (f *File) Write(b []byte) (n int, err error)
```

在使用File.Read读文件时，可考虑使用buffer：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "os"
6. )
7.
8. func main() {
9.     b := make([]byte, 1024)
10.    f, err := os.Open("./tt.txt")
11.    _, err = f.Read(b)
12.    f.Close()
13.
14.    if err != nil {
15.        fmt.Println(err)
16.    }
17.    fmt.Println(string(b))
18.
19. }
```

二：ioutil库，没有直接实现Reader 和 Writer 接口，但是通过内部调用，也可读写文件内容：

```
1. func ReadAll(r io.Reader) ([]byte, error)
2. func ReadFile(filename string) ([]byte, error) //os.Open
3. func WriteFile(filename string, data []byte, perm os.FileMode) error
   //os.OpenFile
4. func ReadDir(dirname string) ([]os.FileInfo, error) // os.Open
```

三：使用bufio库，这个库实现了IO的缓冲操作，通过内嵌io.Reader、io.Writer接口，新建了Reader，Writer 结构体。同时也实现了Reader 和 Writer 接口。

```
1. type Reader struct {
2.     buf          []byte
```

```

3.     rd          io.Reader // reader provided by the client
4.     r, w        int       // buf read and write positions
5.     err          error
6.     lastByte     int
7.     lastRuneSize int
8. }
9.
10. type Writer struct {
11.     err error
12.     buf []byte
13.     n    int
14.     wr   io.Writer
15. }
16.
17.
18. func (b *Reader) Read(p []byte) (n int, err error)
19. func (b *Writer) Write(p []byte) (nn int, err error)

```

这三种读方式的效率怎么样呢，我们可以看看：

```

1. package main
2.
3. import (
4.     "bufio"
5.     "fmt"
6.     "io"
7.     "io/ioutil"
8.     "os"
9.     "time"
10. )
11.
12. func read1(path string) {
13.     fi, err := os.Open(path)
14.     if err != nil {
15.         panic(err)
16.     }
17.     defer fi.Close()
18.     buf := make([]byte, 1024)
19.     for {
20.         n, err := fi.Read(buf)
21.         if err != nil && err != io.EOF {
22.             panic(err)

```

```
23.         }
24.         if 0 == n {
25.             break
26.         }
27.     }
28. }
29.
30. func read2(path string) {
31.     fi, err := os.Open(path)
32.     if err != nil {
33.         panic(err)
34.     }
35.     defer fi.Close()
36.     r := bufio.NewReader(fi)
37.     buf := make([]byte, 1024)
38.     for {
39.         n, err := r.Read(buf)
40.         if err != nil && err != io.EOF {
41.             panic(err)
42.         }
43.         if 0 == n {
44.             break
45.         }
46.     }
47. }
48.
49. func read3(path string) {
50.     fi, err := os.Open(path)
51.     if err != nil {
52.         panic(err)
53.     }
54.     defer fi.Close()
55.     _, err = ioutil.ReadAll(fi)
56. }
57.
58. func main() {
59.
60.     file := "" //找一个大的文件，如日志文件
61.     start := time.Now()
62.     read1(file)
63.     t1 := time.Now()
64.     fmt.Printf("Cost time %v\n", t1.Sub(start))
```



```

65.     read2(file)
66.     t2 := time.Now()
67.     fmt.Printf("Cost time %v\n", t2.Sub(t1))
68.     read3(file)
69.     t3 := time.Now()
70.     fmt.Printf("Cost time %v\n", t3.Sub(t2))
71. }

```

经过多次测试，基本上保持 `bufio < ioutil < file.Read` 这样的成绩，`bufio`读同一文件耗费时间最少，效果稳稳地保持在最佳。

31.3 ioutil包

下面代码使用*ioutil*包实现2种读文件，1种写文件的方法，其中 `ioutil.ReadAll` 可以读取所有 `io.Reader` 流。所以在网络连接中，也经常使用*ioutil.ReadAll* 来读取流，后面章节我们会讲到这块内容。

```

1. package main
2.
3. import (
4.     "fmt"
5.     "io/ioutil"
6.     "os"
7. )
8.
9. func main() {
10.     fileObj, err := os.Open("./tt.txt")
11.     defer fileObj.Close()
12.
13.     contents, _ := ioutil.ReadAll(fileObj)
14.     fmt.Println(string(contents))
15.
16.     if contents, _ := ioutil.ReadFile("./tt.txt"); err == nil {
17.         fmt.Println(string(contents))
18.     }
19.
20.     ioutil.WriteFile("./t3.txt", contents, 0666)
21.
22. }

```

31.4 bufio包

bufio 包通过 bufio.NewReader 和bufio.NewWriter 来创建IO 方法集，利用缓冲来处理流，后面章节我们也会讲到这块内容。

```

1. package main
2.
3. import (
4.     "bufio"
5.     "fmt"
6.     "os"
7. )
8.
9. func main() {
10.     fileObj, _ := os.OpenFile("./tt.txt", os.O_RDWR|os.O_CREATE, 0666)
11.     defer fileObj.Close()
12.
13.     Rd := bufio.NewReader(fileObj)
14.     cont, _ := Rd.ReadSlice('#')
15.     fmt.Println(string(cont))
16.
17.     Wr := bufio.NewWriter(fileObj)
18.     Wr.WriteString("WriteString writes a ## string.")
19.     Wr.Flush()
20. }
```

1. 程序输出：
2. WriteString writes a #

bufio包中，主要方法如下：

1. // NewReaderSize 将 rd 封装成一个带缓存的 bufio.Reader 对象，缓存大小由 size 指定（如果小于 16 则会被设置为 16）。
2. func NewReaderSize(rd io.Reader, size int) *Reader
- 3.
4. // NewReader 相当于 NewReaderSize(rd, 4096)
5. func NewReader(rd io.Reader) *Reader
- 6.
7. // Peek 返回缓存的一个切片，该切片引用缓存中前 n 个字节的数据。
8. // 如果 n 大于缓存的总大小，则返回 当前缓存中能读到的字节的数据。
9. func (b *Reader) Peek(n int) ([]byte, error)

```

10.
11.
12. // Read 从 b 中读出数据到 p 中，返回读出的字节数和遇到的错误。
13. // 如果缓存不为空，则只能读出缓存中的数据，不会从底层 io.Reader
14. // 中提取数据，如果缓存为空，则：
15. // 1、len(p) >= 缓存大小，则跳过缓存，直接从底层 io.Reader 中读出到 p 中。
16. // 2、len(p) < 缓存大小，则先将数据从底层 io.Reader 中读取到缓存中，
17. // 再从缓存读取到 p 中。
18. func (b *Reader) Read(p []byte) (n int, err error)
19.
20. // Buffered 该方法返回从当前缓存中能被读到的字节数。
21. func (b *Reader) Buffered() int
22.
23. // Discard 方法跳过后续的 n 个字节的数据，返回跳过的字节数。
24. func (b *Reader) Discard(n int) (discarded int, err error)
25.
26. // ReadSlice 在 b 中查找 delim 并返回 delim 及其之前的所有数据。
27. // 该操作会读出数据，返回的切片是已读出的数据的引用，切片中的数据在下一次
28. // 读取操作之前是有效的。
29. // 如果找到 delim，则返回查找结果，err 返回 nil。
30. // 如果未找到 delim，则：
31. // 1、缓存不满，则将缓存填满后再次查找。
32. // 2、缓存是满的，则返回整个缓存，err 返回 ErrBufferFull。
33. // 如果未找到 delim 且遇到错误（通常是 io.EOF），则返回缓存中的所有数据
34. // 和遇到的错误。
35. // 因为返回的数据有可能被下一次的读写操作修改，所以大多数操作应该使用
36. // ReadBytes 或 ReadString，它们返回的是数据的拷贝。
37. func (b *Reader) ReadSlice(delim byte) (line []byte, err error)
38.
39. // ReadLine 是一个低水平的行读取原语，大多数情况下，应该使用ReadBytes('\n')
40. // 或 ReadString('\n')，或者使用一个 Scanner。
41. // ReadLine 通过调用 ReadSlice 方法实现，返回的也是缓存的切片。
42. // 用于读取一行数据，不包括行尾标记（\n 或 \r\n）。
43. // 只要能读出数据，err 就为 nil。如果没有数据可读，则 isPrefix
44. // 返回 false，err 返回 io.EOF。
45. // 如果找到行尾标记，则返回查找结果，isPrefix 返回 false。
46. // 如果未找到行尾标记，则：
47. // 1、缓存不满，则将缓存填满后再次查找。
48. // 2、缓存是满的，则返回整个缓存，isPrefix 返回 true。
49. // 整个数据尾部“有一个换行标记”和“没有换行标记”的读取结果是一样。
50. // 如果 ReadLine 读取到换行标记，则调用 UnreadByte 撤销的是换行标记，
51. // 而不是返回的数据。

```

```

52. func (b *Reader) ReadLine() (line []byte, isPrefix bool, err error)
53.
54. // ReadBytes 功能同 ReadSlice, 只不过返回的是缓存的拷贝。
55. func (b *Reader) ReadBytes(delim byte) (line []byte, err error)
56.
57. // ReadString 功能同 ReadBytes, 只不过返回的是字符串。
58. func (b *Reader) ReadString(delim byte) (line string, err error)
59.
60. // Reset 将 b 的底层 Reader 重新指定为 r, 同时丢弃缓存中的所有数据,
61. // 复位所有标记和错误信息。 bufio.Reader。
62. func (b *Reader) Reset(r io.Reader)

```

下面一段代码是，里面有用到peek, Discard 等方法，可以修改方法参数值，仔细体会：

```

1. package main
2.
3. import (
4.     "bufio"
5.     "fmt"
6.     "strings"
7. )
8.
9. func main() {
10.     sr := strings.NewReader("ABCDEFGHJKLMNOPQRSTUVWXYZ1234567890")
11.     buf := bufio.NewReaderSize(sr, 0) //默认16
12.     b := make([]byte, 10)
13.
14.     fmt.Println("==", buf.Buffered()) // 0
15.     s, _ := buf.Peek(5)
16.     fmt.Printf("%d == %q\n", buf.Buffered(), s) //
17.     nn, er := buf.Discard(3)
18.     fmt.Println(nn, er)
19.
20.     for n, err := 0, error(nil); err == nil; {
21.         fmt.Printf("Buffered:%d ==Size:%d== n:%d== b[:n] %q == err:%v\n",
22.             buf.Buffered(), buf.Size(), n, b[:n], err)
23.         n, err = buf.Read(b)
24.         fmt.Printf("Buffered:%d ==Size:%d== n:%d== b[:n] %q == err: %v == s:
25.             %s\n", buf.Buffered(), buf.Size(), n, b[:n], err, s)
26.     }
27.
28.     fmt.Printf("%d == %q\n", buf.Buffered(), s)

```

```
27. }
```

有关IO 的处理，这里主要讲了针对文件的处理。后面在网络IO读写处理中，我们将会接触到更多的方式和方法。

第三十二章 fmt包

《Go语言四十二章经》第三十二章 fmt包

作者：李骁

32.1 fmt包格式化I/O

上一章我们有提到fmt格式化I/O，这一章我们就详细来说。在fmt包，有关格式化输入输出的方法就两大类：Scan 和 Print，分别在scan.go 和 print.go 文件中。

print.go文件中定义了如下函数：

```
1. func Printf(format string, a ...interface{}) (n int, err error)
2. func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
3. func Sprintf(format string, a ...interface{}) string
4.
5. func Print(a ...interface{}) (n int, err error)
6. func Fprint(w io.Writer, a ...interface{}) (n int, err error)
7. func Sprint(a ...interface{}) string
8.
9. func Println(a ...interface{}) (n int, err error)
10. func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
11. func Sprintln(a ...interface{}) string
```

这9个函数，按照两个维度来说明，基本上可以说明白了。当然这两个维度是我个人为了记忆而分，并不是官方的说法。

一：如果把“Print”理解为核心关键字，那么后面跟的后缀有“f”和“ln”以及“”，着重的是内容输出的结果；

如果后缀是“f”，则指定了format

如果后缀是“ln”，则有换行符

1. `Println`、`Fprintln`、`Sprintln` 输出内容时会加上换行符；
2. `Print`、`Fprint`、`Sprint` 输出内容时不加上换行符；
3. `Printf`、`Fprintf`、`Sprintf` 按照指定格式化文本输出内容。

二：如果把“Print”理解为核心关键字，那么前面的前缀有“F”和“S”以及“”，着重的是输出内容的来

源；

如果前缀是“F”，则指定了io.Writer

如果前缀是“S”，则是输出到字符串

1. `Print`、`Printf`、`Println` 输出内容到标准输出`os.Stdout`；
2. `Fprint`、`Fprintf`、`Fprintln` 输出内容到指定的`io.Writer`；
3. `Sprint`、`Sprintf`、`Sprintln` 输出内容到字符串。

`scan.go`文件中定义了如下函数：

1. `func Scanf(format string, a ...interface{}) (n int, err error)`
2. `func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)`
3. `func Sscanf(str string, format string, a ...interface{}) (n int, err error)`
- 4.
5. `func Scan(a ...interface{}) (n int, err error)`
6. `func Fscan(r io.Reader, a ...interface{}) (n int, err error)`
7. `func Sscan(str string, a ...interface{}) (n int, err error)`
- 8.
9. `func Scanln(a ...interface{}) (n int, err error)`
10. `func Fscanln(r io.Reader, a ...interface{}) (n int, err error)`
11. `func Sscanln(str string, a ...interface{}) (n int, err error)`

这9个函数可以扫描格式化文本以生成值。同样也可以按照两个维度来说明。

一：如果把“Scan”理解为核心关键字，那么后面跟的后缀有“f”和“ln”以及“”，着重的是输入内容的结果；

如果后缀是“f”，则指定了format

如果后缀是“ln”，则有换行符

1. `Scanln`、`Fscanln`、`Sscanln` 读取到换行时停止，并要求一次提供一行所有条目；
2. `Scan`、`Fscan`、`Sscan` 读取内容时不关注换行；
3. `Scanf`、`Fscanf`、`Sscanf` 根据格式化文本读取。

二：如果把“Scan”理解为核心关键字，那么前面的前缀有“F”和“S”以及“”，着重的是输入内容的来源；

如果前缀是“F”，则指定了io.Reader

如果前缀是“S”，则是从字符串读取

1. `Scan`、`Scanf`、`Scanln` 从标准输入`os.Stdin`读取文本；

2. `Fscan`、`Fscanf`、`Fscanln` 从指定的`io.Reader`接口读取文本；
3. `Sscan`、`Sscanf`、`Sscanln` 从一个参数字符串读取文本。

32.2 格式化verb应用

在应用上，我们主要讲讲格式化verb，fmt包中格式化的主要功能函数都在`format.go`文件中。

我们先来了解下有哪些verb：

1. 通用：
2. `%v` 值的默认格式表示。当输出结构体时，扩展标志（`%+v`）会添加字段名
3. `%#v` 值的Go语法表示
4. `%T` 值的类型的Go语法表示
5. `%%` 百分号
- 6.
7. 布尔值：
8. `%t` 单词`true`或`false`
- 9.
10. 整数：
11. `%b` 表示为二进制
12. `%c` 该值对应的unicode码值
13. `%d` 表示为十进制
14. `%o` 表示为八进制
15. `%q` 该值对应的单引号括起来的go语法字符面值，必要时会采用安全的转义表示
16. `%x` 表示为十六进制，使用`a-f`
17. `%X` 表示为十六进制，使用`A-F`
18. `%U` 表示为Unicode格式：`U+1234`，等价于"`U+%04X`"
- 19.
20. 浮点数、复数的两个组分：
21. `%b` 无小数部分、二进制指数的科学计数法，如`-123456p-78`；参见`strconv.FormatFloat`
22. `%e` 科学计数法，如`-1234.456e+78`
23. `%E` 科学计数法，如`-1234.456E+78`
24. `%f` 有小数部分 但无指数部分，如`123.456`
25. `%F` 等价于`%f`
26. `%g` 根据实际情况采用`%e`或`%f`格式（以获得 更简洁、准确的输出）
27. `%G` 根据实际情况采用`%E`或`%F`格式（以获得更简洁、准确的输出）
- 28.
29. 字符串和`[]byte`：
30. `%s` 直接输出字符串或者`[]byte`
31. `%q` 该值对应的双引号括起来的Go语法字符串面值，必要时会采用安全的转义表示
32. `%x` 每个字节用两字符十六进制数表示（使用`a-f`）
33. `%X` 每个字节用两字符十六进制数表示（使用`A-F`）


```

34.
35.  指针：
36.  %p      表示为十六进制，并加上前导的0x
37.
38.  宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度
    通过（可能有的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有
    跟数字，表示精度为0。举例如下：
39.
40.  %f:      默认宽度，默认精度
41.  %9f      宽度9，默认精度
42.  %.2f      默认宽度，精度2
43.  %9.2f     宽度9，精度2
44.  %9.f      宽度9，精度0
45.
46.  对于整数，宽度和精度都设置输出总长度。采用精度时表示右对齐并用0填充，而宽度默认表示用空格填
    充。
47.
48.  对于浮点数，宽度设置输出总长度；精度设置小数部分长度（如果有的话），除了%g/%G，此时精度设置总
    的数字个数。例如，对数字123.45，格式%6.2f 输出123.45；格式%.4g输出123.5。%e和%f的默认精
    度是6，%g的默认精度是可将该值区分出来需要的最小数字个数。
49.
50.  对复数，宽度和精度会分别用于实部和虚部，结果用小括号包裹。因此%f用于1.2+3.4i输出
    (1.200000+3.400000i)。
51.
52.  其它flag：
53.
54.  +      总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义）；
55.  -      在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；
56.  #      切换格式：
57.         八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X），指针去掉前面的
         0x（%#p）；
58.         对%q（%#q），如果strconv.CanBackquote返回真会输出反引号括起来的未转义字符串；
59.         对%U（%#U），如果字符是可打印的，会在输出Unicode格式、空格、单引号括起来的go字面
         值；
60.  ' '      对数值，正数前加空格而负数前加负号；
61.         对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格；
62.  0      使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面；

```

verb会忽略不支持的flag。

下面我们用一个程序来演示下：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "os"
6. )
7.
8. type User struct {
9.     name string
10.    age  int
11. }
12.
13. var valF float64 = 32.9983
14. var valI int = 89
15. var valS string = "Go is an open source programming language that makes it easy
    to build simple, reliable, and efficient software."
16. var valB bool = true
17.
18. func main() {
19.
20.    p := User{"John", 28}
21.
22.    fmt.Printf("Printf struct %%v : %v\n", p)
23.    fmt.Printf("Printf struct %%+v : %+v\n", p)
24.    fmt.Printf("Printf struct %%#v : %#v\n", p)
25.    fmt.Printf("Printf struct %%T : %T\n", p)
26.
27.    fmt.Printf("Printf struct %%p : %p\n", &p)
28.
29.    fmt.Printf("Printf float64 %%v : %v\n", valF)
30.    fmt.Printf("Printf float64 %%+v : %+v\n", valF)
31.    fmt.Printf("Printf float64 %%#v : %#v\n", valF)
32.    fmt.Printf("Printf float64 %%T : %T\n", valF)
33.    fmt.Printf("Printf float64 %%f : %f\n", valF)
34.    fmt.Printf("Printf float64 %%4.3f : %4.3f\n", valF)
35.    fmt.Printf("Printf float64 %%8.3f : %8.3f\n", valF)
36.    fmt.Printf("Printf float64 %%-8.3f : %-8.3f\n", valF)
37.    fmt.Printf("Printf float64 %%e : %e\n", valF)
38.    fmt.Printf("Printf float64 %%E : %E\n", valF)
39.
40.    fmt.Printf("Printf int %%v : %v\n", valI)
41.    fmt.Printf("Printf int %%+v : %+v\n", valI)
```

```

42.     fmt.Printf("Printf int %#v : %#v\n", valI)
43.     fmt.Printf("Printf int %T : %T\n", valI)
44.     fmt.Printf("Printf int %d : %d\n", valI)
45.     fmt.Printf("Printf int %8d : %8d\n", valI)
46.     fmt.Printf("Printf int %-8d : %-8d\n", valI)
47.     fmt.Printf("Printf int %b : %b\n", valI)
48.     fmt.Printf("Printf int %c : %c\n", valI)
49.     fmt.Printf("Printf int %o : %o\n", valI)
50.     fmt.Printf("Printf int %U : %U\n", valI)
51.     fmt.Printf("Printf int %q : %q\n", valI)
52.     fmt.Printf("Printf int %x : %x\n", valI)
53.
54.     fmt.Printf("Printf string %v : %v\n", valS)
55.     fmt.Printf("Printf string %+v : %+v\n", valS)
56.     fmt.Printf("Printf string %#v : %#v\n", valS)
57.     fmt.Printf("Printf string %T : %T\n", valS)
58.     fmt.Printf("Printf string %x : %x\n", valS)
59.     fmt.Printf("Printf string %X : %X\n", valS)
60.     fmt.Printf("Printf string %s : %s\n", valS)
61.     fmt.Printf("Printf string %200s : %200s\n", valS)
62.     fmt.Printf("Printf string %-200s : %-200s\n", valS)
63.     fmt.Printf("Printf string %q : %q\n", valS)
64.
65.     fmt.Printf("Printf bool %v : %v\n", valB)
66.     fmt.Printf("Printf bool %+v : %+v\n", valB)
67.     fmt.Printf("Printf bool %#v : %#v\n", valB)
68.     fmt.Printf("Printf bool %T : %T\n", valB)
69.     fmt.Printf("Printf bool %t : %t\n", valB)
70.
71.     s := fmt.Sprintf("a %s", "string")
72.     fmt.Println(s)
73.
74.     fmt.Fprintf(os.Stderr, "an %s\n", "error")
75. }

```

我们主要通过fmt.Printf来理解这些flag 的含义，这对我们今后的开发有较强的实际作用。至于其他函数，我就不一一举例，有兴趣可以进一步研究。

第三十三章 Socket网络

《Go语言四十二章经》第三十三章 Socket网络

作者：李骁

33.1 Socket基础知识

tcp/udp、ip构成了网络通信的基石，tcp/ip是面向连接的通信协议，要求建立连接时进行3次握手确保连接已被建立，关闭连接时需要4次通信来保证客户端和服务端都已经关闭，也就是我们常说的三次握手，四次挥手。在通信过程中还有保证数据不丢失，在连接不畅通时还需要进行超时重试等等。

socket就是封装了这一套基于tcp/udp/ip协议细节，提供了一系列套接字接口进行通信。

我们知道Socket有两种：TCP Socket和UDP Socket，TCP和UDP是协议，而要确定一个进程的需要三元组，还需要IP地址和端口。




- IPv4地址

目前的全球因特网所采用的协议族是TCP/IP协议。IP是TCP/IP协议中网络层的协议，是TCP/IP协议族的核心协议。目前主要采用的IP协议的版本号是4(简称为IPv4)，IPv4的地址位数为32位，也就是最多有 2^{32} 的网络设备可以联到Internet上。

地址格式类似这样：127.0.0.1

- IPv6地址

IPv6是新一版本的互联网协议，也可以说是新一代互联网的协议，它是为了解决IPv4在实施过程中遇到的各种问题而被提出的，IPv6采用128位地址长度，几乎可以不受限制地提供地址。在IPv6的设计过程中除了一劳永逸地解决了地址短缺问题以外，还考虑了在IPv4中解决不好的其它问题，主要有端到端IP连接、服务质量（QoS）、安全性、多播、移动性、即插即用等。

地址格式类似这样：2002  82e7   0 c0e8:82e7

33.2 TCP 与 UDP

Go是自带runtime的跨平台编程语言，Go中暴露给语言使用者的tcp socket api是建立OS原生tcp socket接口之上的，所以在使用上相对简单。

TCP Socket

建立网络连接过程：TCP连接的建立需要经历客户端和服务端的三次握手的过程。Go 语言net包封装了系列API，在TCP连接中，服务端是一个标准的Listen + Accept的结构，而在客户端Go语言使用net.Dial或DialTimeout进行连接建立：

在Go语言的net包中有一个类型TCPConn，这个类型可以用来作为客户端和服务端交互的通道，他有两个主要的函数：

```
1. func (c *TCPConn) Write(b []byte) (n int, err os.Error)
2. func (c *TCPConn) Read(b []byte) (n int, err os.Error)
```

TCPConn可以用在客户端和服务端来读写数据。

在Go语言中通过ResolveTCPAddr获取一个TCPAddr：

```
1. func ResolveTCPAddr(net, addr string) (*TCPAddr, os.Error)
```

net参数是"tcp4"、"tcp6"、"tcp"中的任意一个，分别表示TCP(IPv4-only)，TCP(IPv6-only)或者TCP(IPv4, IPv6的任意一个)。

addr表示域名或者IP地址，例如"www.google.com:80" 或者"127.0.0.1:22"。

我们来看一个TCP 连接建立的具体代码：

```
1. // tcp server 服务端代码
2.
3. package main
4.
5. import (
6.     "bufio"
7.     "fmt"
8.     "io"
9.     "net"
10.    "time"
11. )
12.
13. func main() {
14.
15.     var tcpAddr *net.TCPAddr
16.
17.     tcpAddr, _ = net.ResolveTCPAddr("tcp", "127.0.0.1:999")
18.
19.     tcpListener, _ := net.ListenTCP("tcp", tcpAddr)
20.
```

```

21.     defer tcpListener.Close()
22.
23.     fmt.Println("Server ready to read ...")
24.     for {
25.         tcpConn, err := tcpListener.AcceptTCP()
26.         if err != nil {
27.             fmt.Println("accept error:", err)
28.             continue
29.         }
30.         fmt.Println("A client connected : " + tcpConn.RemoteAddr().String())
31.         go tcpPipe(tcpConn)
32.     }
33.
34. }
35.
36. func tcpPipe(conn *net.TCPConn) {
37.     ipStr := conn.RemoteAddr().String()
38.
39.     defer func() {
40.         fmt.Println(" Disconnected : " + ipStr)
41.         conn.Close()
42.     }()
43.
44.     reader := bufio.NewReader(conn)
45.     i := 0
46.
47.     for {
48.         message, err := reader.ReadString('\n') //将数据按照换行符进行读取。
49.         if err != nil || err == io.EOF {
50.             break
51.         }
52.
53.         fmt.Println(string(message))
54.
55.         time.Sleep(time.Second * 3)
56.
57.         msg := time.Now().String() + conn.RemoteAddr().String() + " Server Say
hello! \n"
58.         b := []byte(msg)
59.
60.         conn.Write(b)
61.         i++

```

```

62.
63.         if i > 10 {
64.             break
65.         }
66.     }
67. }

```

服务端 `tcpListener.AcceptTCP()` 接受一个客户端连接请求，通过 `go tcpPipe(tcpConn)` 开启一个新协程来管理这连接。在 `func tcpPipe(conn *net.TCPConn)` 中，处理服务端和客户端数据的交换，在这段代码 `for` 中，通过 `bufio.NewReader` 读取客户端发送过来的数据。

客户端代码：

```

1.  // tcp client
2.
3.  package main
4.
5.  import (
6.      "bufio"
7.      "fmt"
8.      "io"
9.      "net"
10.     "time"
11. )
12.
13. func main() {
14.     var tcpAddr *net.TCPAddr
15.     tcpAddr, _ = net.ResolveTCPAddr("tcp", "127.0.0.1:999")
16.
17.     conn, err := net.DialTCP("tcp", nil, tcpAddr)
18.     if err != nil {
19.         fmt.Println("Client connect error ! " + err.Error())
20.         return
21.     }
22.
23.     defer conn.Close()
24.
25.     fmt.Println(conn.LocalAddr().String() + " : Client connected!")
26.
27.     onMessageRecived(conn)
28. }
29.

```

```

30. func onMessageRecived(conn *net.TCPConn) {
31.     reader := bufio.NewReader(conn)
32.     b := []byte(conn.LocalAddr().String() + " Say hello to Server... \n")
33.     conn.Write(b)
34.     for {
35.         msg, err := reader.ReadString('\n')
36.         fmt.Println("ReadString")
37.         fmt.Println(msg)
38.
39.         if err != nil || err == io.EOF {
40.             fmt.Println(err)
41.             break
42.         }
43.         time.Sleep(time.Second * 2)
44.
45.         fmt.Println("writing...")
46.
47.         b := []byte(conn.LocalAddr().String() + " write data to Server... \n")
48.         _, err = conn.Write(b)
49.
50.         if err != nil {
51.             fmt.Println(err)
52.             break
53.         }
54.     }
55. }

```

客户端`net.DialTCP("tcp", nil, tcpAddr)` 向服务端发起一个连接请求，调用 `onMessageRecived(conn)`，处理客户端和服务端数据的发送与接收。在`func onMessageRecived(conn *net.TCPConn)` 中，通过 `bufio.NewReader` 读取客户端发送过来的数据。

上面2个例子你可以试着运行一下，程序支持多个客户端同时运行。当然，这两个例子只是简单的tcp原始连接，在实际中，我们还可能需要定义协议。

用Socket进行通信，发送的数据包一定是有结构的，类似于：数据头+数据长度+数据内容+校验码+数据尾。而在TCP流传输的过程中，可能会出现分包与黏包的现象。我们为了解决这些问题，需要我们自定义通信协议进行封包与解包。对这方面内容如有兴趣可以去了解更多相关知识。

第三十四章 命令行 flag 包

《Go语言四十二章经》第三十四章 命令行 flag 包

作者：李骁

34.1 命令行

写命令行程序时需要对命令参数进行解析，这时我们可以使用os库。os可以通过变量Args来获取命令参数，os.Args返回一个字符串数组，其中第一个参数就是执行文件本身。

```
1. package main
2.
3. import (
4.     "fmt"
5.     "os"
6. )
7.
8. func main() {
9.     fmt.Println(os.Args)
10. }
```

编译执行后执行

```
1. $ ./cmd -user="root"
2. [./cmd -user=root]
```

这种方式操作起来要自己封装，比较费时费劲。

34.2 flag包

Go提供了flag库，可以很方便的操作命名行参数，下面介绍下flag的用法。

几个概念：

- 1) 命令行参数（或参数）：是指运行程序提供的参数
- 2) 已定义命令行参数：是指程序中通过flag.Xxx等这种形式定义了的参数
- 3) 非flag（non-flag）命令行参数（或保留的命令行参数）：先可以简单理解为flag包不能解析的

参数

```
1. package main
2.
3. import (
4.     "flag"
5.     "fmt"
6.     "os"
7. )
8.
9. var (
10.    h, H bool
11.
12.    v bool
13.    q *bool
14.
15.    D    string
16.    Conf string
17. )
18.
19. func init() {
20.    flag.BoolVar(&h, "h", false, "帮助信息")
21.    flag.BoolVar(&H, "H", false, "帮助信息")
22.
23.    flag.BoolVar(&v, "v", false, "显示版本号")
24.
25.    //
26.    flag.StringVar(&D, "D", "daemon", "set descripton ")
27.    flag.StringVar(&Conf, "Conf", "/dev/conf/cli.conf", "set Conf filename ")
28.
29.    // 另一种绑定方式
30.    q = flag.Bool("q", false, "退出程序")
31.
32.    // 像flag.Xxx函数格式都是一样的，第一个参数表示参数名称，
33.    // 第二个参数表示默认值，第三个参数表示使用说明和描述。
34.    // flag.XxxVar这样的函数第一个参数换成了变量地址，
35.    // 后面的参数和flag.Xxx是一样的。
36.
37.    // 改变默认的 Usage
38.
39.    flag.Usage = usage
40.}
```

```

41.     flag.Parse()
42.
43.     var cmd string = flag.Arg(0)
44.
45.     fmt.Printf("-----\n")
46.     fmt.Printf("cli non=flags      : %s\n", cmd)
47.
48.     fmt.Printf("q: %b\n", *q)
49.
50.     fmt.Printf("descripton: %s\n", D)
51.     fmt.Printf("Conf filename : %s\n", Conf)
52.
53.     fmt.Printf("-----\n")
54.     fmt.Printf("there are %d non-flag input param\n", flag.NArg())
55.     for i, param := range flag.Args() {
56.         fmt.Printf("#%d      :%s\n", i, param)
57.     }
58.
59. }
60.
61. func main() {
62.     flag.Parse()
63.
64.     if h || H {
65.         flag.Usage()
66.     }
67. }
68.
69. func usage() {
70.     fmt.Fprintf(os.Stderr, `CLI: 8.0
71. Usage: Cli [-hvq] [-D descripton] [-Conf filename]
72.
73. `)
74.     flag.PrintDefaults()
75. }

```

flag包实现了命令行参数的解析，大致需要几个步骤：

一：flag参数定义或绑定

定义flags有两种方式：

1) flag.Xxx(), 其中Xxx可以是Int、String等；返回一个相应类型的指针，如：

```
1. var ip = flag.Int("flagname", 1234, "help message for flagname")
```

2) flag.XxxVar(), 将flag绑定到一个变量上, 如:

```
1. var flagvar int
2. flag.IntVar(&flagvar, "flagname", 1234, "help message for flagname")
```

另外, 还可以创建自定义flag, 只要实现flag.Value接口即可 (要求receiver是指针), 这时候可以通过如下方式定义该flag:

```
flag.Var(&flagVal, "name", "help message for flagname")
```

命令行flag的语法有如下三种形式:

```
-flag // 只支持bool类型
-flag=x
-flag x // 只支持非bool类型
```

二: flag参数解析

在所有的flag定义完成之后, 可以通过调用flag.Parse()进行解析。

根据Parse()中for循环终止的条件, 当parseOne返回false, nil时, Parse解析终止。

```
1. s := f.args[0]
2. if len(s) == 0 || s[0] != '-' || len(s) == 1 {
3.     return false, nil
4. }
```

当遇到单独的一个“-”或不是“-”开始时, 会停止解析。比如: ./cli - -f 或 ./cli -f

这两种情况, -f都不会被正确解析。像这些参数, 我们称之为non-flag参数

parseOne方法中接下来是处理-flag=x, 然后是-flag (bool类型) (这里对bool进行了特殊处理), 接着是-flag x这种形式, 最后, 将解析成功的Flag实例存入FlagSet的actual map中。

Arg(i int)和Args()、NArg()、NFlag()

Arg(i int)和Args()这两个方法就是获取non-flag参数的; NArg()获得non-flag个数; NFlag()获得FlagSet中actual长度 (即被设置了的参数个数)。

flag解析遇到non-flag参数就停止了。所以如果我们将non-flag参数放在最前面, flag什么也不会解析, 因为flag遇到了这个就停止解析了。

三: 分支程序

根据参数值，代码进入分支程序，执行相关功能。上面代码提供了 `-h` 参数的功能执行。

```
1. if h || H {  
2.     flag.Usage()  
3. }
```

总体而言，从例子上看，`flag` package很有用，但是并没有强大到解析一切的程度。如果你的入参解析非常复杂，`flag`可能捉襟见肘。

Cobra是一个用来创建强大的现代CLI命令行的Go开源库。开源包可能比较合适构建更为复杂的命令行程序。开源地址：<https://github.com/spf13/cobra>

第三十五章 模板

《Go语言四十二章经》第三十五章 模板

作者：李骁

Printf也可以做到输出格式化，当然，对于简单的例子来说足够了，但是我们有时候还是需要复杂的输出格式，甚至需要将格式化代码分离开来。这时，可以使用text/template和html/template。

Go 官方库提供了两个模板库： text/template 和 html/template 。这两个库类似，当需要输出html格式的代码时需要使用 html/template。

35.1 text/template

所谓模板引擎，则将模板和数据进行渲染的输出格式化后的字符程序。对于Go，执行这个流程大概需要三步。

- 创建模板对象
- 加载模板
- 执行渲染模板

其中最后一步就是把加载的字符和数据进行格式化。

```
1. package main
2.
3. import (
4.     "log"
5.     "os"
6.     "text/template"
7. )
8.
9. const templ = `
10. {{range .}}-----
11. Name:   {{.Name}}
12. Price:  {{.Price | printf "%4s"}}
13. {{end}}`
14.
15. var report = template.Must(template.New("report").Parse(templ))
16.
17. type Book struct {
```

```

18.     Name  string
19.     Price float64
20. }
21.
22. func main() {
23.     Data := []Book{"《三国演义》", 19.82}, {"《儒林外史》", 99.09}, {"《史记》",
24.     26.89}}
25.     if err := report.Execute(os.Stdout, Data); err != nil {
26.         log.Fatal(err)
27.     }

```

```

1. 程序输出：
2. -----
3. Name:    《三国演义》
4. Price:   %!s(float64=19.82)
5. -----
6. Name:    《儒林外史》
7. Price:   %!s(float64=99.09)
8. -----
9. Name:    《史记》
10. Price:  %!s(float64=26.89)

```

如果把模板的内容存在一个文本文件里tmp.txt：

```

1. {{range .}}-----
2. Name:   {{.Name}}
3. Price:  {{.Price | printf "%4s"}}
4. {{end}}

```

我们可以这样处理：

```

1. package main
2.
3. import (
4.     "log"
5.     "os"
6.     "text/template"
7. )
8.
9. var report = template.Must(template.ParseFiles("tmp.txt"))

```

```

10.
11. type Book struct {
12.     Name string
13.     Price float64
14. }
15.
16. func main() {
17.     Data := []Book{{"《三国演义》", 19.82}, {"《儒林外史》", 99.09}, {"《史记》",
18.         26.89}}
19.     if err := report.Execute(os.Stdout, Data); err != nil {
20.         log.Fatal(err)
21.     }
22. }

```

```

1. 程序输出：
2.
3. -----
4. Name:    《三国演义》
5. Price:   %!s(float64=19.82)
6. -----
7. Name:    《儒林外史》
8. Price:   %!s(float64=99.09)
9. -----
10. Name:   《史记》
11. Price:  %!s(float64=26.89)

```

```

1. Tmpl, err := template.ParseFiles("tmp.txt")
2. //建立模板, 自动 new("name")

```

ParseFiles接受一个字符串，字符串的内容是一个模板文件的路径。

ParseGlob是用正则的方式匹配多个文件。

假设一个目录里有a.txt b.txt c.txt的话，用ParseFiles需要写3行对应3个文件，如果有更多文件，可以用ParseGlob。

写成template.ParseGlob("*.txt") 即可。

```

1. var report = template.Must(template.ParseFiles("tmp.txt"))

```

函数Must，它的作用是检测模板是否正确，例如大括号是否匹配，注释是否正确的关闭，变量是否正确的书写。

35.2 html/template

和text、template类似，html/template主要在提供支持HTML的功能，所以基本使用上和上面差不多，我们来看下面代码：

index.html:

```
1. <!doctype html>
2. <head>
3.   <meta charset="UTF-8">
4.   <meta name="Author" content="">
5.   <meta name="Keywords" content="">
6.   <meta name="Description" content="">
7.   <title>Go</title>
8. </head>
9. <body>
10.   {{ . }}
11. </body>
12. </html>
```

main.go:

```
1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6.     "text/template"
7. )
8.
9. func tHandler(w http.ResponseWriter, r *http.Request) {
10.     t, _ := template.ParseFiles("index.html")
11.     t.Execute(w, "Hello World!")
12. }
13.
14. func main() {
15.     http.HandleFunc("/", tHandler)
16.     http.ListenAndServe(":8080", nil)
17. }
```

运行程序，在浏览器打开：<http://localhost:8080/> 会看到页面显示Hello World!

```

1. func(t *Template) ParseFiles(filenames ...string) (*Template, error)
2. func(t *Template) ParseGlob(patternstring) (*Template, error)

```

从上面代码中我们可以看到，通过ParseFile加载了单个Html模板文件。但最终的页面很可能是多个模板文件的嵌套结果。

ParseFiles也支持加载多个模板文件，模板对象的名字则是第一个模板文件的文件名。

ExecuteTemplate方法，用于执行指定名字的模板，下面我们根据一段代码来看看：

Layout.html，注意在开头根据模板语法，定义了模板名字，define “layout”。

在结尾处，通过 {{ template “index” }}

注意：通过将模板应用于一个数据结构(即该数据结构作为模板的参数)来执行，来获得输出。模板执行时会遍历结构并将指针表示为.(称之为dot)，指向运行过程中数据结构的当前位置的值。

{{template “header” .}} 嵌套模板中，加入.dot 代表在该模板中也可以使用该数据结构，否则不能显示。

```

1.  {{ define "layout" }}
2.
3.  <!doctype html>
4.  <head>
5.    <meta charset="UTF-8">
6.    <meta name="Author" content="">
7.    <meta name="Keywords" content="">
8.    <meta name="Description" content="">
9.    <title>Go</title>
10. </head>
11. <body>
12.   {{ . }}
13.
14.   {{ template "index" }}
15. </body>
16. </html>
17.
18. {{ end }}

```

```

1. Index.html :
2.
3. {{ define "index" }}
4.

```

```

5. <div>
6. <b>Index</b>
7. </div>
8. {{ end }}

```

通过define定义模板，还可以通过template action引入模板，类似include。

```

1. package main
2.
3. import (
4.     "net/http"
5.     "text/template"
6. )
7.
8. func tHandler(w http.ResponseWriter, r *http.Request) {
9.     t, _ := template.ParseFiles("layout.html", "index.html")
10.    t.ExecuteTemplate(w, "layout", "Hello World!")
11. }
12.
13. func main() {
14.    http.HandleFunc("/", tHandler)
15.    http.ListenAndServe(":8080", nil)
16. }

```

运行程序，在浏览器打开：<http://localhost:8080/>

Hello World!

Index

```

1. 有关ParseGlob方法，则是通过glob通配符加载模板，例如 t, _ :=
   template.ParseGlob("*.html")

```

35.3 模板语法

```

1. 【模板标签】
2. 模板标签用"{{"和"}}"括起来
3.
4. 【注释】
5. {{/* a comment */}}
6. 使用"{{/*"和"*/}}"来包含注释内容
7.

```

8. **【变量】**
9. `{{.}}`
10. 此标签输出当前对象的值
11. `{{.Admpub}}`
12. 表示输出Struct对象中字段或方法名称为"Admpub"的值。
- 13.
14. 当"Admpub"是匿名字段时，可以访问其内部字段或方法，比如"Com"：`{{.Admpub.Com}}`，
15. 如果"Com"是一个方法并返回一个Struct对象，同样也可以访问其字段或方法：
`{{.Admpub.Com.Field1}}`
- 16.
17. `{{.Method1 "参数值1" "参数值2"}}`
18. 调用方法"Method1"，将后面的参数值依次传递给此方法，并输出其返回值。
- 19.
20. `{{$admpub}}`
21. 此标签用于输出在模板中定义的名称为"admpub"的变量。当\$admpub本身是一个Struct对象时，可访问其字段：`{{$admpub.Field1}}`
22. 在模板中定义变量：变量名称用字母和数字组成，并带上"\$"前缀，采用简式赋值。
23. 比如：`{{ $x := "OK" }}` 或 `{{ $x := pipeline }}`

1. **【通道函数】**
2. 用法1：
3. `{{FuncName1}}`
4. 此标签将调用名称为"FuncName1"的模板函数（等同于执行"`FuncName1()`"，不传递任何参数）并输出其返回值。
5. 用法2：
6. `{{FuncName1 "参数值1" "参数值2"}}`
7. 此标签将调用FuncName1("参数值1", "参数值2")，并输出其返回值
8. 用法3：
9. `{{.Admpub | FuncName1}}`
10. 此标签将调用名称为"FuncName1"的模板函数（等同于执行"`FuncName1(this.Admpub)`"，将竖线"|"左边的".Admpub"变量值作为函数参数传送）并输出其返回值。
- 11.
- 12.
13. **【条件判断】**
14. 用法1：
15. `{{if pipeline}} T1 {{end}}`
16. 标签结构：`{{if ...}} ... {{end}}`
17. 用法2：
18. `{{if pipeline}} T1 {{else}} T0 {{end}}`
19. 标签结构：`{{if ...}} ... {{else}} ... {{end}}`
20. 用法3：
21. `{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}`

22. 标签结构：`{{if ...}} ... {{else if ...}} ... {{end}}`
23. 其中`if`后面可以是一个条件表达式（包括通道函数表达式。`pipeline`即通道），也可以是一个字符窜变量或布尔值变量。当为字符窜变量时，如为空字符串则判断为`false`，否则判断为`true`。

1. 【遍历】
2. 用法1：
3. `{{range $k, $v := .Var}} {{$k}} => {{$v}} {{end}}`
4. `range...end`结构内部如要使用外部的变量，比如`.Var2`，需要这样写：`$.Var2`
5. （即：在外部变量名称前加符号`"$"`即可，单独的`"$"`意义等同于`global`）
6. 用法2：
7. `{{range .Var}} {{.}} {{end}}`
8. 用法3：
9. `{{range pipeline}} T1 {{else}} T0 {{end}}`
10. 当没有可遍历的值时，将执行`else`部分。
- 11.
- 12.
- 13.
14. 【嵌入子模板】
15. 用法1：
16. `{{template "name"}}`
17. 嵌入名称为`"name"`的子模板。使用前请确保已经用`{{define "name"}}`子模板内容`{{end}}`定义好了子模板内容。
18. 用法2：
19. `{{template "name" pipeline}}`
20. 将通道的值赋给子模板中的`"."`（即`"{{.}}"`）
- 21.
22. 【子模板嵌套】
23. `{{define "T1"}}ONE{{end}}`
24. `{{define "T2"}}TWO{{end}}`
25. `{{define "T3"}}{{template "T1"}} {{template "T2"}}{{end}}`
26. `{{template "T3"}}`
27. 输出：
28. ONE TWO
- 29.
- 30.
31. 【定义局部变量】
32. 用法1：
33. `{{with pipeline}} T1 {{end}}`
34. 通道的值将赋给该标签内部的`"."`。（注：这里的“内部”一词是指被`{{with pipeline}}...{{end}}`包围起来的部分，即`T1`所在位置）
35. 用法2：
36. `{{with pipeline}} T1 {{else}} T0 {{end}}`

```

37. 如果通道的值为空, "."不受影响并且执行T0, 否则, 将通道的值赋给"."并且执行T1。
38.
39.
40. 说明: {{end}}标签是if、with、range的结束标签。
41.
42.
43. 【例子: 输出字符窜】
44. {{"\output\""}}
45. 输出一个字符窜常量。
46.
47. {{`"output"`}}
48. 输出一个原始字符串常量
49.
50. {{printf "%q" "output"}}
51. 函数调用. (等同于: printf("%q", "output").)
52.
53. {{"output" | printf "%q"}}
54. 竖线"| "左边的结果作为函数最后一个参数. (等同于: printf("%q", "output").)
55.
56. {{printf "%q" (print "out" "put")}}
57. 圆括号中表达式的整体结果作为printf函数的参数. (等同于: printf("%q", print("out",
    "put")).)
58.
59. {{"put" | printf "%s%s" "out" | printf "%q"}}
60. 一个更复杂的调用. (等同于: printf("%q", printf("%s%s", "out", "put")).)
61.
62. {{"output" | printf "%s" | printf "%q"}}
63. 等同于: printf("%q", printf("%s", "output")).
64.
65. {{with "output"}}{{printf "%q" .}}{{end}}
66. 一个使用点号"."的with操作. (等同于: printf("%q", "output").)
67.
68. {{with $x := "output" | printf "%q"}}{{ $x }}{{end}}
69. with结构, 定义变量, 值为执行通道函数之后的结果 (等同于: $x := printf("%q", "output").)
70.
71. {{with $x := "output"}}{{printf "%q" $x}}{{end}}
72. with结构中, 在其它动作中使用定义的变量
73.
74. {{with $x := "output"}}{{ $x | printf "%q" }}{{end}}
75. 同上, 但使用了通道. (等同于: printf("%q", "output").)

```

2.

3. ===== 【预定义的模板全局函数】 =====

4. 【and】

5. {{and x y}}

6. 表示 : if x then y else x

7. 如果x为真, 返回y, 否则返回x。等同于Go中的 : x && y

8.

9. 【call】

10. {{call .X.Y 1 2}}

11. 表示 : dot.X.Y(1, 2)

12. call后面的第一个参数的结果必须是一个函数（即这是一个函数类型的值），其余参数作为该函数的参数。

13. 该函数必须返回一个或两个结果值，其中第二个结果值是error类型。

14. 如果传递的参数与函数定义的不匹配或返回的error值不为nil，则停止执行。

15.

16. 【html】

17. 转义文本中的html标签，如将 "<" 转义为 "<"， ">" 转义为 ">" 等

18.

19. 【index】

20. {{index x 1 2 3}}

21. 返回index后面的第一个参数的某个索引对应的元素值，其余的参数为索引值

22. 表示 : x[1][2][3]

23. x必须是一个map、slice或数组

24.

25. 【js】

26. 返回用JavaScript的escape处理后的文本

27.

28. 【len】

29. 返回参数的长度值（int类型）

30.

31. 【not】

32. 返回单一参数的布尔否定值。

33.

34. 【or】

35. {{or x y}}

36. 表示 : if x then x else y。等同于Go中的 : x || y

37. 如果x为真返回x, 否则返回y。

38.

39. 【print】

40. fmt.Sprintf的别名

41.

42. 【printf】

43. fmt.Sprintf的别名

```

44.
45.  【println】
46.  fmt.Sprintln的别名
47.
48.  【urlquery】
49.  返回适合在URL查询中嵌入到形参中的文本转义值。（类似于PHP的urlencode）

```

```

1.
2.  ===== 【布尔函数】 =====
3.
4.  布尔函数对于任何零值返回false，非零值返回true。
5.  这里定义了一组二进制比较操作符函数：
6.
7.  【eq】
8.  返回表达式"arg1 == arg2"的布尔值
9.
10. 【ne】
11. 返回表达式"arg1 != arg2"的布尔值
12.
13. 【lt】
14. 返回表达式"arg1 < arg2"的布尔值
15.
16. 【le】
17. 返回表达式"arg1 <= arg2"的布尔值
18.
19. 【gt】
20. 返回表达式"arg1 > arg2"的布尔值
21.
22. 【ge】
23. 返回表达式"arg1 >= arg2"的布尔值
24.
25. 对于简单的多路相等测试，eq只接受两个参数进行比较，后面其它的参数将分别依次与第一个参数进行比较，
26.  {{eq arg1 arg2 arg3 arg4}}
27.  即只能作如下比较：
28.  arg1==arg2 || arg1==arg3 || arg1==arg4 ...

```


第三十六章 net/http包

《Go语言四十二章经》第三十六章 net/http包

作者：李骁

在Go中，搭建一个http server简单到令人难以置信。只需要引入net/http包，写几行代码，一个http服务器就可以正常运行并接受访问请求。

下面就是Go最简单的http服务器：

```
1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6. )
7.
8. func myfunc(w http.ResponseWriter, r *http.Request) {
9.     fmt.Fprintf(w, "hi")
10. }
11.
12. func main() {
13.     http.HandleFunc("/", myfunc)
14.     http.ListenAndServe(":8080", nil)
15. }
```

访问 <http://localhost:8080/> ，我们可以看到网页输出“hi” ！

下面我们通过分析net/http的源代码，来深入理解这个包的实现原理。在net/http源代码中，我们可以深深体会到Go语言的接口设计哲学，这个包主要有4个文件，分别是：

client.go

server.go

request.go

response.go

我们知道有http Request 请求和 http Response 响应，以及client和server，我们先从这四个方面讲讲net/http包：

36.1 Request

http Request请求是由客户端发出的消息，用来使服务器执行动作。发出的消息包括起始行，Headers，Body。

在net/http包中，request.go文件定义了结构：

```
1.
2.  type Request struct
3.
4.  代表客户端给服务器端发送的一个请求或者是服务器端收到的一个请求，但是服务器端和客户端使用Request
   时语义区别很大。
5.
6.  // 利用指定的method, url以及可选的body返回一个新的请求.如果body参数实现了
7.  // io.Closer接口, Request返回值的Body 字段会被设置为body, 并会被Client
8.  // 类型的Do、Post和PostForm方法以及Transport.RoundTrip方法关闭。
9.  func NewRequest(method, urlStr string, body io.Reader) (*Request, error)
10.
11. // 从b中读取和解析一个请求。
12. func ReadRequest(b *bufio.Reader) (req *Request, err error)
13.
14. // 给request添加cookie, AddCookie向请求中添加一个cookie.按照RFC 6265
15. // section 5.4的规则, AddCookie不会添加超过一个Cookie头字段。
16. // 这表示所有的cookie都写在同一行, 用分号分隔 (cookie内部用逗号分隔属性)
17. func (r *Request) AddCookie(c *Cookie)
18.
19. // 返回request中指定名name的cookie, 如果没有发现, 返回ErrNoCookie
20. func (r *Request) Cookie(name string) (*Cookie, error)
21.
22. // 返回该请求的所有cookies
23. func (r *Request) Cookies() []*Cookie
24.
25. // 利用提供的用户名和密码给http基本权限提供具有一定权限的header。
26. // 当使用http基本授权时, 用户名和密码是不加密的
27. func (r *Request) SetBasicAuth(username, password string)
28.
29. // 如果在request中发送, 该函数返回客户端的user-Agent
30. func (r *Request) UserAgent() string
31.
32. // 对于指定格式的key, FormFile返回符合条件的第一个文件, 如果有必要的话,
33. // 该函数会调用ParseMultipartForm和ParseForm。
34. func (r *Request) FormFile(key string) (multipart.File, *multipart.FileHeader,
```

```

    error)
35.
36. // 返回key获取的队列中第一个值。在查询过程中post和put中的主题参数优先级
37. // 高于url中的value。为了访问相同key的多个值，调用ParseForm然后直接
38. // 检查RequestForm。
39. func (r *Request) FormValue(key string) string
40.
41. // 如果这是一个有多部分组成的post请求，该函数将会返回一个MIME 多部分reader，
42. // 否则的话将会返回一个nil和error。使用本函数代替ParseMultipartForm
43. // 可以将请求body当做流stream来处理。
44. func (r *Request) MultipartReader() (*multipart.Reader, error)
45.
46. // 解析URL中的查询字符串，并将解析结果更新到r.Form字段。对于POST或PUT
47. // 请求，ParseForm还会将body当作表单解析，并将结果既更新到r.PostForm也
48. // 更新到r.Form。解析结果中，POST或PUT请求主体要优先于URL查询字符串
49. // （同名变量，主体的值在查询字符串的值前面）。如果请求的主体的大小没有被
50. // MaxBytesReader函数设定限制，其大小默认限制为开头10MB。
51. // ParseMultipartForm会自动调用ParseForm。重复调用本方法是无意义的。
52. func (r *Request) ParseForm() error
53.
54. // ParseMultipartForm将请求的主体作为multipart/form-data解析。
55. // 请求的整个主体都会被解析，得到的文件记录最多 maxMemory字节保存在内存，
56. // 其余部分保存在硬盘的temp文件里。如果必要，ParseMultipartForm会
57. // 自行调用 ParseForm。重复调用本方法是无意义的。
58. func (r *Request) ParseMultipartForm(maxMemory int64) error
59.
60. // 返回post或者put请求body指定元素的第一个值，其中url中的参数被忽略。
61. func (r *Request) PostFormValue(key string) string
62.
63. // 检测在request中使用的http协议是否至少是major.minor
64. func (r *Request) ProtoAtLeast(major, minor int) bool
65.
66. // 如果request中有refer，那么refer返回相应的url。Referer在request
67. // 中是拼错的，这个错误从http初期就已经存在了。该值也可以从Headermap中
68. // 利用Header["Referer"]获取；在使用过程中利用Referer这个方法而
69. // 不是map的形式的好处是在编译过程中可以检查方法的错误，而无法检查map中
70. // key的错误。
71. func (r *Request) Referer() string
72.
73. // Write方法以有线格式将HTTP/1.1请求写入w（用于将请求写入下层TCPConn等）
74. // 。本方法会考虑请求的如下字段：Host URL Method (defaults to "GET")
75. // Header ContentLength TransferEncoding Body如果存在Body，

```

```

76. // ContentLength字段<= 0且TransferEncoding字段未显式设置为
77. // ["identity"], Write方法会显式添加"Transfer-Encoding: chunked"
78. // 到请求的头域。Body字段会在发送完请求后关闭。
79. func (r *Request) Write(w io.Writer) error
80.
81. // 该函数与Write方法类似，但是该方法写的request是按照http代理的格式去写。
82. // 尤其是，按照RFC 2616 Section 5.1.2, WriteProxy会使用绝对URI
83. // （包括协议和主机名）来初始化请求的第1行（Request-URI行）。无论何种情况，
84. // WriteProxy都会使用r.Host或r.URL.Host设置Host头。
85. func (r *Request) WriteProxy(w io.Writer) error

```

36.2 Response

指对于一个http请求的响应response，HTTP响应包括起始行，Headers，Body。

```

1. // 注意是在response.go中定义的，而在server.go有一个
2. // type response struct ，注意大小写。这个结构是体现在server端的功能。
3. type Response struct
4.
5. // ReadResponse从r读取并返回一个HTTP 回复。req参数是可选的，指定该回复
6. // 对应的请求（即是对该请求的回复）。如果是nil，将假设请求是GET请求。
7. // 客户端必须在结束resp.Body的读取后关闭它。读取完毕并关闭后，客户端可以
8. // 检查resp.Trailer字段获取回复的 trailer的键值对。
9. func ReadResponse(r *bufio.Reader, req *Request) (*Response, error)
10.
11. // 解析cookie并返回在header中利用set-Cookie设定的cookie值。
12. func (r *Response) Cookies() []*Cookie
13.
14. // 返回response中Location的header值的url。如果该值存在的话，则对于
15. // 请求问题可以解决相对重定向的问题，如果该值为nil，则返回ErrNoLocation。
16. func (r *Response) Location() (*url.URL, error)
17.
18. // 判定在response中使用的http协议是否至少是major.minor的形式。
19. func (r *Response) ProtoAtLeast(major, minor int) bool
20.
21. // 将response中信息按照线性格式写入w中。
22. func (r *Response) Write(w io.Writer) error

```

36.3 client

```

1.  // Client具有Do, Get, Head, Post以及PostForm等方法。 其中Do方法可以对
2.  // Request进行一系列的设定, 而其他的对request设定较少。如果Client使用默认的
3.  // Client, 则其中的Get, Head, Post以及PostForm方法相当于默认的http.Get,
4.  // http.Post, http.Head以及http.PostForm函数。
5.  type Client struct
6.
7.  // 利用GET方法对一个指定的URL进行请求, 如果response是如下重定向中的一个
8.  // 代码, 则Get之后将会调用重定向内容, 最多10次重定向。
9.  // 301 (永久重定向, 告诉客户端以后应该从新地址访问)
10. // 302 (暂时性重定向, 作为HTTP1.0的标准, PHP的默认Location重定向用到
11. // 也是302), 注: 303和307其实是对302的细化。
12. // 303 (对于Post请求, 它表示请求已经被处理, 客户端可以接着使用GET方法去
13. // 请求Location里的URL)
14. // 307 (临时重定向, 对于Post请求, 表示请求还没有被处理, 客户端应该向
15. // Location里的URL重新发起Post请求)
16. func Get(url string) (resp *Response, err error)
17.
18. // 该函数功能见net中Head方法功能。该方法与默认的defaultClient中
19. // Head方法一致。
20. func Head(url string) (resp *Response, err error)
21.
22. // 该方法与默认的defaultClient中Post方法一致。
23. func Post(url string, bodyType string, body io.Reader) (resp *Response, err
    error)
24.
25. // 该方法与默认的defaultClient中PostForm方法一致。
26. func PostForm(url string, data url.Values) (resp *Response, err error)
27.
28. // Do发送http请求并且返回一个http响应, 遵守client的策略, 如重定向,
29. // cookies以及auth等. 错误经常是由于策略引起的, 当err是nil时, resp
30. // 总会包含一个非nil的resp.body. 当调用者读完resp.body之后应该关闭它,
31. // 如果resp.body没有关闭, 则Client底层RoundTripper将无法重用存在的
32. // TCP连接去服务接下来的请求, 如果resp.body非nil, 则必须对其进行关闭.
33. // 通常来说, 经常使用Get, Post, 或者PostForm来替代Do.
34. func (c *Client) Do(req *Request) (resp *Response, err error)
35.
36. // 利用get方法请求指定的url.Get请求指定的页面信息, 并返回实体主体。
37. func (c *Client) Get(url string) (resp *Response, err error)
38.
39. // 利用head方法请求指定的url, Head只返回页面的首部。
40. func (c *Client) Head(url string) (resp *Response, err error)
41.

```

```

42. // post方法请求指定的URL, 如果body也是一个io.Closer, 则在请求之后关闭它
43. func (c *Client) Post(url string, bodyType string, body io.Reader) (resp
    *Response, err error)
44.
45. // 利用post方法请求指定的url, 利用data的key和value作为请求体.
46. func (c *Client) PostForm(url string, data url.Values) (resp *Response, err
    error)

```

Do方法可以灵活的对request进行配置, 然后进行请求。利用http.Client以及http.NewRequest可以模拟请求。下面模拟request中带有cookie的请求, 通过http.Client的Do方法发送这个请求。也就是说配置http.NewRequest, 我们通过http.Client的Do方法来发送任何http请求。示例如下:

- 模拟请求:

```

1. package main
2.
3. import (
4.     "fmt"
5.     "io/ioutil"
6.     "net/http"
7.     "strconv"
8. )
9.
10. func main() {
11.     client := &http.Client{}
12.     request, err := http.NewRequest("GET", "http://www.baidu.com", nil)
13.     if err != nil {
14.         fmt.Println(err)
15.     }
16.     cookie := &http.Cookie{Name: "userId", Value: strconv.Itoa(12345)}
17.
18.     //request中添加cookie
19.
20.     request.AddCookie(cookie)
21.
22.     //设置request的header
23.     request.Header.Set("Accept", "text/html, application/xhtml+xml,
        application/xml;q=0.9, */*;q=0.8")
24.     request.Header.Set("Accept-Charset", "GBK, utf-8;q=0.7, */*;q=0.3")
25.     request.Header.Set("Accept-Encoding", "gzip, deflate, sdch")
26.     request.Header.Set("Accept-Language", "zh-CN, zh;q=0.8")

```

```

27.     request.Header.Set("Cache-Control", "max-age=0")
28.     request.Header.Set("Connection", "keep-alive")
29.     response, err := client.Do(request)
30.     if err != nil {
31.         fmt.Println(err)
32.         return
33.     }
34.     defer response.Body.Close()
35.     fmt.Println(response.StatusCode)
36.     if response.StatusCode == 200 {
37.         r, err := ioutil.ReadAll(response.Body)
38.         if err != nil {
39.             fmt.Println(err)
40.         }
41.         fmt.Println(string(r))
42.     }
43. }

```

- 发送一个http Get请求:

```

1. package main
2. import (
3.     "fmt"
4.     "io/ioutil"
5.     "net/http"
6. )
7.
8. func main() {
9.     response, err := http.Get("http://www.baidu.com")
10.    if err != nil {
11.    }
12.
13.    //程序在使用完回复后必须关闭回复的主体。
14.
15.    defer response.Body.Close()
16.
17.    body, _ := ioutil.ReadAll(response.Body)
18.    fmt.Println(string(body))
19. }

```

使用http POST方法可以直接使用http.Post 或 http.PostForm

- http.Post请求:

```
1. package main
2.
3. import (
4.     "net/http"
5.     "strings"
6.     "fmt"
7.     "io/ioutil"
8. )
9.
10. func main() {
11.     resp, err := http.Post("http://www.xxx.com/loginRegister/login.do",
12.         "application/x-www-form-urlencoded",
13.         strings.NewReader("mobile=xxxxxxxxxx&isRemberPwd=1"))
14.     if err != nil {
15.         fmt.Println(err)
16.         return
17.     }
18.     defer resp.Body.Close()
19.     body, err := ioutil.ReadAll(resp.Body)
20.     if err != nil {
21.         fmt.Println(err)
22.         return
23.     }
24.     fmt.Println(string(body))
25. }
```

- http.PostForm请求:

```
1. package main
2.
3. import (
4.     "net/http"
5.     "fmt"
6.     "io/ioutil"
7.     "net/url"
8. )
9.
10. func main() {
11.     postParam := url.Values{
12.         "mobile": {"xxxxxxx"},
13.     }
```



```

13.         "isRemberPwd": {"1"},
14.     }
15.     resp, err := http.PostForm("http://www.xxx.com/loginRegister/login.do",
    postParam)
16.     if err != nil {
17.         fmt.Println(err)
18.         return
19.     }
20.     defer resp.Body.Close()
21.     body, err := ioutil.ReadAll(resp.Body)
22.     if err != nil {
23.         fmt.Println(err)
24.         return
25.     }
26.     fmt.Println(string(body))
27. }

```

上面列举了几种客户端发送http.request请求的方式：

1.http.Client与http.NewRequest模拟HTTP请求，方法可以是Get 、 Post 、 PostForm

2.http.Get 方式请求

3.http.Post 方式请求

4.http.PostForm 方式请求

一共这四种HTTP请求方式，这几种方式都和http.Client封装的函数（方法）有密切关系。

最基础的还是http.NewRequest，其他三种方法是在它基础上做了封装而已：

```

1. func NewRequest(method, url string, body io.Reader) (*Request, error)
2.
3. func (c *Client) Get(url string) (resp *Response, err error) {
4.     req, err := NewRequest("GET", url, nil)
5.     .....
6.
7. func (c *Client) Post(url string, contentType string, body io.Reader) (resp
    *Response, err error) {
8.     req, err := NewRequest("POST", url, body)
9.     .....

```

36.4 server

```

1. type Handler interface {
2.     ServeHTTP(ResponseWriter, *Request)
3. }
4.
5. type Server struct
6.
7. // 监听TCP网络地址srv.Addr然后调用Serve来处理接下来连接的请求。
8. // 如果srv.Addr是空的话，则使用":http"。
9. func (srv *Server) ListenAndServe() error
10.
11. // ListenAndServeTLS监听srv.Addr确定的TCP地址，并且会调用Serve
12. // 方法处理接收到的连接。必须提供证书文件和对应的私钥文件。如果证书是由
13. // 权威机构签发的，certFile参数必须是顺序串联的服务端证书和CA证书。
14. // 如果srv.Addr为空字符串，会使用":https"。
15. func (srv *Server) ListenAndServeTLS(certFile, keyFile string) error
16.
17. // 接受Listener l的连接，创建一个新的服务协程。该服务协程读取请求然后调用
18. // srv.Handler来应答。实际上就是实现了对某个端口进行监听，然后创建相应的连接。
19. func (srv *Server) Serve(l net.Listener) error
20.
21. // 该函数控制是否http的keep-alives能够使用，默认情况下，keep-alives总是可用的。
22. // 只有资源非常紧张的环境或者服务端在关闭进程中时，才应该关闭该功能。
23. func (s *Server) SetKeepAlivesEnabled(v bool)
24.
25. // 是一个http请求多路复用器，它将每一个请求的URL和
26. // 一个注册模式的列表进行匹配，然后调用和URL最匹配的模式的处理程序进行后续操作。
27. type ServeMux
28.
29. // 初始化一个新的ServeMux
30. func NewServeMux() *ServeMux
31.
32. // 将handler注册为指定的模式，如果该模式已经有了handler，则会出错panic。
33. func (mux *ServeMux) Handle(pattern string, handler Handler)
34.
35. // 将handler注册为指定的模式
36. func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
    *Request))
37.
38. // 根据指定的r.Method, r.Host以及r.URL.Path返回一个用来处理给定请求的handler。
39. // 该函数总是返回一个非nil的 handler，如果path不是一个规范格式，则handler会
40. // 重定向到其规范path。Handler总是返回匹配该请求的已注册模式；在内建重定向
41. // 处理器的情况下，pattern会在重定向后进行匹配。如果没有已注册模式可以应用于该请求，

```

```

42. // 本方法将返回一个内建的"404 page not found"处理器和一个空字符串模式。
43. func (mux *ServeMux) Handler(r *Request) (h Handler, pattern string)
44.
45. // 该函数用于将最接近请求url模式的handler分配给指定的请求。
46. func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request)

```

server.go中定义的接口：

```

1.
2. // A Handler responds to an HTTP request.
3. // ...
4. type Handler interface {
5.     ServeHTTP(ResponseWriter, *Request)
6. }

```

这个接口应该算是整个net/http中关键了，如果你仔细看这个包的源代码，你会发现很多结构体实现了这个接口的ServeHTTP方法。

注意这个接口的注释：Handler响应HTTP请求。没错，最终我们的http服务是通过实现ServeHTTP(ResponseWriter, *Request)来达到服务端接收客户端请求并响应。

理解 HTTP 构建的网络应用只要关注两个端——客户端（clinet）和服务端（server），两个端的交互来自 clinet 的 request，以及server端的response。

HTTP服务器，主要在于如何接受 clinet端 的 request，server端向client端返回response。

那这个过程是什么样的呢？有了前面request、response、client、server四个部分作为基础，我们要讲清楚这个过程，还需要回到开始的程序：

```

1. func main() {
2.     http.HandleFunc("/", myfunc)
3.     http.ListenAndServe(":8080", nil)
4. }

```

两行代码，成功启动了一个http服务器。我们通过net/http 包源代码分析，调用Http.HandleFunc，按顺序做了几件事：

1.Http.HandleFunc调用了DefaultServeMux的HandleFunc

```

1. func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
2.     DefaultServeMux.HandleFunc(pattern, handler)
3. }

```

2. `DefaultServeMux.HandleFunc`调用了`DefaultServeMux.Handle`，`DefaultServeMux`是一个`ServeMux` 指针变量。而`ServeMux` 是Go中`Multiplexer`（多路复用器），通过`Handle`匹配 `pattern` 和我们定义的`handler`。

```
1. var DefaultServeMux = &defaultServeMux
2. var defaultServeMux ServeMux
3.
4. func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter,
    *Request)) {
5.     mux.Handle(pattern, HandlerFunc(handler))
6. }
```

注意：

上面的方法命名`Handle`，`HandleFunc`和`HandlerFunc`，`Handler`（接口），他们很相似，容易混淆。记住`Handle`和`HandleFunc`和`pattern` 匹配有关，也即往`DefaultServeMux`的 `map[string]muxEntry`中增加对应的`handler`和路由规则。

接着我们看看`myfunc`的声明和定义：

```
1. func myfunc(w http.ResponseWriter, r *http.Request) {
2.     fmt.Fprintf(w, "hi")
3. }
```

而`type HandlerFunc func(ResponseWriter, *Request)` 是一个函数类型。而我们定义的`myfunc`的函数签名刚好符合这个函数类型。

所以`http.HandleFunc("/", myfunc)`，实际上是`mux.Handle("/", HandlerFunc(myfunc))`。

`HandlerFunc(myfunc)` 让`myfunc`成为了`HandlerFunc`类型，我们称`myfunc`为`handler`。而`HandlerFunc`类型是具有`ServeHTTP`方法的，有了`ServeHTTP`方法也就是实现了`Handler`接口。

现在`ServeMux`和`Handler`都和我们的`myfunc`联系上了，接下来和结构体`server`有关了。

从`http.ListenAndServe`的源码可以看出，它创建了一个`server`对象，并调用`server`对象的`ListenAndServe`方法：

```
1. func ListenAndServe(addr string, handler Handler) error {
2.     server := &Server{Addr: addr, Handler: handler}
3.     return server.ListenAndServe()
4. }
```

而我们HTTP服务器中第二行代码：

```
1. http.ListenAndServe(":8080", nil)
```

创建了一个server对象，并调用server对象的ListenAndServe方法，这里没有直接传递Handler，而是默认使用DefaultServeMux作为multiplexer，myfunc是存在于handler和路由规则中的。

Server的ListenAndServe方法中，会初始化监听地址Addr，同时调用Listen方法设置监听。

```
1. for {
2.     rw, e := l.Accept()
3.     ...
4.     c := srv.newConn(rw)
5.     c.setState(c.rwc, StateNew)
6.     go c.serve(ctx)
7. }
```

监听开启之后，一旦客户端请求过来，Go就开启一个协程go c.serve(ctx)处理请求，主要逻辑都在serve方法之中。

func (c *conn) serve(ctx context.Context)，这个方法很长，里面主要的一句：serverHandler{c.server}.ServeHTTP(w, w.req)。其中w由w, err := c.readRequest(ctx)得到，因为有传递context。

还是来看源代码：

```
1. type serverHandler struct {
2.     srv *Server
3. }
4.
5. func (sh serverHandler) ServeHTTP(rw ResponseWriter, req Request) {
6.     handler := sh.srv.Handler
7.     if handler == nil {
8.         handler = DefaultServeMux
9.     }
10.    if req.RequestURI == "" && req.Method == "OPTIONS" {
11.        handler = globalOptionsHandler{}
12.    }
13.    handler.ServeHTTP(rw, req)
14. }
```

从`http.ListenAndServe(":8080", nil)`开始, `handler`是`nil`, 所以最后实际`ServeHTTP`方法是`DefaultServeMux.ServeHTTP(rw, req)`。

```

1. func (mux *ServeMux) ServeHTTP(w ResponseWriter, r *Request) {
2.     if r.RequestURI == "*" {
3.         if r.ProtoAtLeast(1, 1) {
4.             w.Header().Set("Connection", "close")
5.         }
6.         w.WriteHeader(StatusBadRequest)
7.         return
8.     }
9.     h, _ := mux.Handler(r)
10.    h.ServeHTTP(w, r)
11. }

```

通过`func (mux ServeMux) Handler(r Request) (h Handler, pattern string)`, 我们得到`Handler h`, 然后执行`h.ServeHTTP(w, r)`方法, 也就是执行我们的`myfunc`函数 (别忘了`myfunc`是`HandlerFunc`类型, 而他的`ServeHTTP(w, r)`方法这里其实就是自己调用自己), 把`response`写到`http.ResponseWriter`对象返回给客户端, `fmt.Fprintf(w, "hi")`, 我们在客户端会接收到`hi`。至此整个HTTP服务执行完成。

总结下, HTTP服务整个过程大概是这样:

```
1. Request -> ServeMux(Multiplexer) -> handler-> Response
```

如果需要自定义 `http.Server`怎么办呢:

```

1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6. )
7.
8. func myfunc(w http.ResponseWriter, r *http.Request) {
9.     fmt.Fprintf(w, "hi")
10. }
11.
12. func main() {
13.     server := http.Server{
14.         Addr:      ":8080",
15.         ReadTimeout: 0,

```

```

16.         WriteTimeout: 0,
17.     }
18.     http.HandleFunc("/", myfunc)
19.     server.ListenAndServe()
20. }

```

这样服务也能跑起来！

指定Servemux的用法：

```

1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6. )
7.
8. func myfunc(w http.ResponseWriter, r *http.Request) {
9.     fmt.Fprintf(w, "hi")
10. }
11.
12. func main() {
13.     mux := http.NewServeMux()
14.
15.     mux.HandleFunc("/", myfunc)
16.     http.ListenAndServe(":8080", mux)
17. }

```

如果既指定Servemux又自定义 http.Server：

```

1. package main
2.
3. import (
4.     "fmt"
5.     "net/http"
6. )
7.
8. func myfunc(w http.ResponseWriter, r *http.Request) {
9.     fmt.Fprintf(w, "hi")
10. }
11.
12. func main() {

```

```
13.     server := http.Server{
14.         Addr:         ":8080",
15.         ReadTimeout:  0,
16.         WriteTimeout: 0,
17.     }
18.     mux := http.NewServeMux()
19.     server.Handler = mux
20.
21.     mux.HandleFunc("/", myfunc)
22.     server.ListenAndServe()
23. }
```

在前面pprof 包的内容中我们也用了本章开头这段代码，当我们访问

<http://localhost:8080/debug/pprof/> 时可以看到对应的性能分析报告。

因为我们这样导入 `_"net/http/pprof"` 包时，在文件 `pprof.go` 文件中`init` 函数已经定义好了 handler：

```
1. func init() {
2.     http.HandleFunc("/debug/pprof/", Index)
3.     http.HandleFunc("/debug/pprof/cmdline", Cmdline)
4.     http.HandleFunc("/debug/pprof/profile", Profile)
5.     http.HandleFunc("/debug/pprof/symbol", Symbol)
6.     http.HandleFunc("/debug/pprof/trace", Trace)
7. }
```

所以，我们就可以通过浏览器访问上面地址来看到报告。现在再来看这些代码，我们就明白怎么回事了！

36.5 自定义处理器（Custom Handlers）

自定义的Handler：

标准库http提供了Handler接口，用于开发者实现自己的handler。只要实现接口的ServeHTTP方法即可。

```
1. package main
2.
3. import (
4.     "log"
5.     "net/http"
6.     "time"
```



```

7.  )
8.
9.  type timeHandler struct {
10.      format string
11.  }
12.
13. func (th *timeHandler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
14.     tm := time.Now().Format(th.format)
15.     w.Write([]byte("The time is: " + tm))
16. }
17.
18. func main() {
19.     mux := http.NewServeMux()
20.
21.     th := &timeHandler{format: time.RFC1123}
22.     mux.Handle("/time", th)
23.
24.     log.Println("Listening...")
25.     http.ListenAndServe(":3000", mux)
26. }

```

我们知道，NewServeMux可以创建一个ServeMux实例，ServeMux同时也实现了ServeHTTP方法，因此代码中的mux也是一种handler。把它当成参数传给http.ListenAndServe方法，后者会把mux传给Server实例。因为指定了handler，因此整个http服务就不再是DefaultServeMux，而是mux，无论是在注册路由还是提供请求服务的时候。

任何有 func(http.ResponseWriter, *http.Request) 签名的函数都能转化为一个 HandlerFunc 类型。这很有用，因为 HandlerFunc 对象内置了 ServeHTTP 方法，后者可以聪明又方便的调用我们最初提供的函数内容。

36.6 将函数作为处理器

```

1. package main
2.
3. import (
4.     "log"
5.     "net/http"
6.     "time"
7. )
8.
9. func timeHandler(w http.ResponseWriter, r *http.Request) {

```

```

10.     tm := time.Now().Format(time.RFC1123)
11.     w.Write([]byte("The time is: " + tm))
12. }
13.
14. func main() {
15.     mux := http.NewServeMux()
16.
17.     // Convert the timeHandler function to a HandlerFunc type
18.     th := http.HandlerFunc(timeHandler)
19.     // And add it to the ServeMux
20.     mux.Handle("/time", th)
21.
22.     log.Println("Listening...")
23.     http.ListenAndServe(":3000", mux)
24. }

```

创建新的server:

```

1.  func index(w http.ResponseWriter, r *http.Request) {
2.      w.Header().Set("Content-Type", "text/html")
3.
4.      html := `<doctype html>
5.          <html>
6.          <head>
7.              <title>Hello World</title>
8.          </head>
9.          <body>
10.             <p>
11.                 Welcome
12.             </p>
13.          </body>
14. </html>`
15.     fmt.Fprintln(w, html)
16. }
17.
18. func main(){
19.     http.HandleFunc("/", index)
20.
21.     server := &http.Server{
22.         Addr: ":8000",
23.         ReadTimeout: 60 * time.Second,
24.         WriteTimeout: 60 * time.Second,

```

```

25.     }
26.     server.ListenAndServe()
27. }

```

36.7 中间件Middleware

所谓中间件，就是连接上下级不同功能的函数或者软件，通常进行一些包裹函数的行为，为被包裹函数提供添加一些功能或行为。前文的HandleFunc就能把签名为 `func(w http.ResponseWriter, r *http.Request)` 的函数包裹成handler。这个函数也算是中间件。

Go的http中间件很简单，只要实现一个函数签名为`func(http.Handler) http.Handler`的函数即可。`http.Handler`是一个接口，接口方法我们熟悉的为`serveHTTP`。返回也是一个handler。因为Go中的函数也可以当成变量传递或者或者返回，因此也可以在中间件函数中传递定义好的函数，只要这个函数是一个handler即可，即实现或者被handlerFunc包裹成为handler处理器。

```

1. func index(w http.ResponseWriter, r *http.Request) {
2.     w.Header().Set("Content-Type", "text/html")
3.
4.     html := `<doctype html>
5.         <html>
6.         <head>
7.             <title>Hello World</title>
8.         </head>
9.         <body>
10.            <p>
11.                Welcome
12.            </p>
13.        </body>
14.    </html>`
15.    fmt.Fprintln(w, html)
16. }
17.
18. func middlewareHandler(next http.Handler) http.Handler{
19.     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request){
20.         // 执行handler之前的逻辑
21.         next.ServeHTTP(w, r)
22.         // 执行完毕handler后的逻辑
23.     })
24. }
25.
26. func loggingHandler(next http.Handler) http.Handler {

```

```
27.     return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
28.         start := time.Now()
29.         log.Printf("Started %s %s", r.Method, r.URL.Path)
30.         next.ServeHTTP(w, r)
31.         log.Printf("Completed %s in %v", r.URL.Path, time.Since(start))
32.     })
33. }
34.
35. func main() {
36.     http.Handle("/", loggingHandler(http.HandlerFunc(index)))
37.
38.     http.ListenAndServe(":8000", nil)
39. }
```

36.8 静态站点

下面代码通过指定目录，作为静态站点：

```
1. package main
2.
3. import (
4.     "net/http"
5. )
6.
7. func main() {
8.     http.Handle("/", http.FileServer(http.Dir("D:/html/static/")))
9.     http.ListenAndServe(":8080", nil)
10. }
```

第三十七章 context包

《Go语言四十二章经》第三十七章 context包

作者：李骁

37.1 context包

在Go中，每个请求的request在单独的goroutine中进行，处理一个request也可能涉及多个goroutine之间的交互。一个请求衍生出的各个 goroutine 之间需要满足一定的约束关系，以实现一些诸如有效期，中止routine树，传递请求全局变量之类的功能。于是Go为我们提供一个解决方案，标准context包。使用context可以使开发者方便的在这些goroutine之间传递request相关的数据、取消goroutine的信号或截止时间等。

每个goroutine在执行之前，都要先知道程序当前的执行状态，通常将这些执行状态封装在一个Context变量中，传递给要执行的goroutine中。上下文则几乎已经成为传递与请求同生存周期变量的标准方法。在网络编程下，当接收到一个网络请求Request，处理Request时，我们可能需要开启不同的goroutine来获取数据与逻辑处理，即一个请求Request，会在多个goroutine中处理。而这些goroutine可能需要共享Request的一些信息；同时当Request被取消或者超时的时候，所有从这个Request创建的所有goroutine也应该被结束。

context包不仅实现了在程序单元之间共享状态变量的方法，同时能通过简单的方法，使我们在被调用程序单元的外部，通过设置ctx变量值，将过期或撤销这些信号传递给被调用的程序单元。若存在A调用B的API，B再调用C的API，若A调用B取消，那也要取消B调用C，通过在A，B，C的API调用之间传递Context，以及判断其状态。

Context结构

```
1. // Context包含过期，取消信号，request值传递等，方法在多个goroutine中协程安全
2. type Context interface {
3.     // Done 方法在context被取消或者超时返回一个close的channel
4.     Done() <-chan struct{}
5.
6.     Err() error
7.
8.     // Deadline 返回context超时时间
9.     Deadline() (deadline time.Time, ok bool)
10.
11.     // Value 返回context相关key对应的值
```

```

12.     Value(key interface{}) interface{}
13. }

```

- Deadline会返回一个超时时间，goroutine获得了超时时间后，例如可以对某些io操作设定超时时间。
- Done方法返回一个通道（channel），当Context被撤销或过期时，该通道关闭，即它是一个表示Context是否已关闭的信号。
- 当Done通道关闭后，Err方法表明Context被撤的原因。
- Value可以让goroutine共享一些数据，当然获得数据是协程安全的。但使用这些数据的时候要注意同步，比如返回了一个map，而这个map的读写则要加锁。

goroutine的创建和调用关系总是像层层调用进行的，就像人的辈分一样，而更靠顶部的goroutine应有办法主动关闭其下属的goroutine的执行（不然程序可能就失控了）。为了实现这种关系，Context结构也应该像一棵树，叶子节点须总是由根节点衍生出来的。

要创建Context树，第一步就是要得到根节点，context.Background函数的返回值就是根节点：

```

1. func Background() Context

```

该函数返回空的Context，该Context一般由接收请求的第一个goroutine创建，是与进入请求对应的Context根节点，它不能被取消、没有值、也没有过期时间。它常常作为处理Request的顶层context存在。

有了根节点，又该怎么创建其它的子节点，孙节点呢？context包为我们提供了多个函数来创建他们：

```

1. func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
2. func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
3. func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
4. func WithValue(parent Context, key interface{}, val interface{}) Context

```

函数都接收一个Context类型的参数parent，并返回一个Context类型的值，这样就层层创建出不同的节点。子节点是从复制父节点得到的，并且根据接收参数设定子节点的一些状态值，接着就可以将子节点传递给下层的goroutine了。

再回到之前的问题：该怎么通过Context传递改变后的状态呢？使用Context的goroutine无法取消某个操作，其实这也是符合常理的，因为这些goroutine是被某个父goroutine创建的，而理应只有父goroutine可以取消操作。在父goroutine中可以通过WithCancel方法获得一个cancel方法，从而获得cancel的权利。

第一个WithCancel函数，它是将父节点复制到子节点，并且还返回一个额外的CancelFunc函数类型

变量，该函数类型的定义为：

```
1. type CancelFunc func()
```

调用CancelFunc对象将撤销对应的Context对象，这就是主动撤销Context的方法。在父节点的Context所对应的环境中，通过WithCancel函数不仅可创建子节点的Context，同时也获得了该节点Context的控制权，一旦执行该函数，则该节点Context就结束了，则子节点需要类似如下代码来判断是否已结束，并退出该goroutine：

```
1. select {
2.     case <-cxt.Done():
3.         // do some clean...
4. }
```

WithDeadline函数的作用也差不多，它返回的Context类型值同样是parent的副本，但其过期时间由deadline和parent的过期时间共同决定。当parent的过期时间早于传入的deadline时间时，返回的过期时间应与parent相同。父节点过期时，其所有的子孙节点必须同时关闭；反之，返回的父节点的过期时间则为deadline。

WithTimeout函数与WithDeadline类似，只不过它传入的是从现在开始Context剩余的生命时长。他们都同样也都返回了所创建的子Context的控制权，一个CancelFunc类型的函数变量。

当顶层的Request请求函数结束后，我们就可以cancel掉某个context，从而层层goroutine根据判断cxt.Done()来结束。

WithValue函数，它返回parent的一个副本，调用该副本的Value(key)方法将得到val。这样我们不光将根节点原有的值保留了，还在子孙节点中加入了新的值，注意若存在Key相同，则会被覆盖。

context包通过构建树型关系的Context，来达到上一层goroutine能对传递给下一层goroutine的控制。对于处理一个Request请求操作，需要采用context来层层控制goroutine，以及传递一些变量来共享。

Context对象的生存周期一般仅为一个请求的处理周期。即针对一个请求创建一个Context变量（它为Context树结构的根）；在请求处理结束后，撤销此ctx变量，释放资源。

每次创建一个goroutine，要么将原有的Context传递给goroutine，要么创建一个子Context并传递给goroutine。

Context能灵活地存储不同类型、不同数目的值，并且使多个goroutine安全地读写其中的值。

当通过父Context对象创建子Context对象时，可同时获得子Context的一个撤销函数，这样父Context对象的创建环境就获得了对子Context将要被传递到的goroutine的撤销权。

注意：使用时遵循context规则

1. 不要把Context存在一个结构体当中，显式地传入函数。Context变量需要作为第一个参数使用，一般命名为ctx。
2. 即使方法允许，也不要传入一个nil的Context，如果你不确定你要用什么Context的时候传一个context.TODO。
3. 使用context的Value相关方法只应该用于在程序和接口中传递的和请求相关的元数据，不要用它来传递一些可选的参数。
4. 同样的Context可以用来传递到不同的goroutine中，Context在多个goroutine中是安全的。

在子Context被传递到的goroutine中，应该对该子Context的Done通道（channel）进行监控，一旦该通道被关闭（即上层运行环境撤销了本goroutine的执行），应主动终止对当前请求信息的处理，释放资源并返回。

37.2 context应用

```
1. package main
2.
3. import (
4.     "context"
5.     "log"
6.     "os"
7.     "time"
8. )
9.
10. var logg *log.Logger
11.
12. func someHandler() {
13.     // 新建一个ctx
14.     ctx, cancel := context.WithCancel(context.Background())
15.
16.     //传递ctx
17.     go doStuff(ctx)
18.
19.     //10秒后取消doStuff
20.     time.Sleep(10 * time.Second)
21.     log.Println("cancel")
22. }
```



```

23.      //调用cancel : context.WithCancel 返回的CancelFunc
24.      cancel()
25.
26.  }
27.
28.  func doStuff(ctx context.Context) {
29.
30.      // for 循环来每1秒work一下, 判断ctx是否被取消了, 如果是就退出
31.
32.      for {
33.          time.Sleep(1 * time.Second)
34.
35.          select {
36.          case <-ctx.Done():
37.              logg.Printf("done")
38.              return
39.          default:
40.              logg.Printf("work")
41.          }
42.      }
43.  }
44.
45.  func main() {
46.      logg = log.New(os.Stdout, "", log.Ltime)
47.      someHandler()
48.      logg.Printf("down")
49.  }

```

```

1.  程序输出 :
2.  16:28:21 work
3.  16:28:22 work
4.  16:28:23 work
5.  16:28:24 work
6.  16:28:25 work
7.  16:28:26 work
8.  16:28:27 work
9.  16:28:28 work
10. 16:28:29 work
11. 2018/08/22 16:28:30 cancel
12. 16:28:30 down

```

someHandler() 作为顶层的Request请求函数, 处理完主要任务后, 主动cancel掉context, 而子

层goroutine doStuff(ctx context.Context) 根据判断cxt.Done()来结束。

第三十八章 **Json**数据格式

第三十九章 Mysql数据库

《Go语言四十二章经》第三十九章 Mysql数据库

作者：李骁

39.1 database/sql包

Go 提供了database/sql包用于对SQL数据库的访问，作为操作数据库的入口对象sql.DB，主要为我们提供了两个重要的功能：

- sql.DB 通过数据库驱动为我们提供管理底层数据库连接的打开和关闭操作。
- sql.DB 为我们管理数据库连接池

需要注意的是，sql.DB表示操作数据库的抽象访问接口，而非一个数据库连接对象；它可以根据driver打开关闭数据库连接，管理连接池。正在使用的连接被标记为繁忙，用完后回到连接池等待下次使用。所以，如果你没有把连接释放回连接池，会导致过多连接使系统资源耗尽。

导入mysql数据库驱动

```
1. import (  
2.     "database/sql"  
3.     _ "github.com/go-sql-driver/mysql"  
4. )
```

通常来说，不应该直接使用驱动所提供的方法，而是应该使用 sql.DB，因此在导入 mysql 驱动时，这里使用了匿名导入的方式(在包路径前添加 _)，当导入了一个数据库驱动后，此驱动会自行初始化并注册自己到Go的database/sql上下文中，因此我们就可以通过 database/sql 包提供的方法访问数据库了。

39.2 Mysql数据库操作

我们先建立表结构：

```
1. CREATE TABLE t_article_cate (  
2.     `cid` int(10) NOT NULL AUTO_INCREMENT,  
3.     `cname` varchar(60) NOT NULL,  
4.     `ename` varchar(100),  
5.     `cateimg` varchar(255),
```

```

6.  `addtime` int(10) unsigned NOT NULL DEFAULT '0',
7.  `publishtime` int(10) unsigned NOT NULL DEFAULT '0',
8.  `scope` int(10) unsigned NOT NULL DEFAULT '10000',
9.  `status` tinyint(1) unsigned NOT NULL DEFAULT '0',
10. PRIMARY KEY (`cid`),
11. UNIQUE KEY catename (`cname`)
12. ) ENGINE=InnoDB AUTO_INCREMENT=99 DEFAULT CHARSET=utf8 COLLATE=utf8_general_ci;

```

下面代码使用预编译的方式，来进行增删改查的操作，并通过事务来批量提交一批数据。预编译语句(PreparedStatement)提供了诸多好处，可以实现自定义参数的查询，通常来说，比手动拼接字符串SQL 语句高效，可以防止SQL注入攻击。

```

1. package main
2.
3. import (
4.     "database/sql"
5.     "fmt"
6.     "strings"
7.     "time"
8.
9.     _ "github.com/go-sql-driver/mysql"
10. )
11.
12. type DbWorker struct {
13.     Dsn string
14.     Db  *sql.DB
15. }
16.
17. type Cate struct {
18.     cid      int
19.     cname    string
20.     addtime  int
21.     scope    int
22. }
23.
24. // sql.NullInt64 sql.NullString
25. // 因为Go是强类型语言，所以查询数据时先定义数据类型，
26. // 但是查询数据库中的数据存在三种可能：
27. // 存在值，存在零值，未赋值NULL 三种状态，因为可以将待查询的数据类型
28. // 定义为sql.Nullxxx类型，
29. // 可以通过判断Valid值来判断查询到的值是否为赋值状态还是未赋值NULL状态。
30.

```

```

31. func main() {
32.     dbw := DbWorker{Dsn: "root:123456@tcp(localhost:3306)/mydb?
        charset=utf8mb4"}
33.     // 支持下面几种DSN写法，具体看mysql服务端配置，常见为第2种
34.     // user@unix(/path/to/socket)/dbname?charset=utf8
35.     // user:password@tcp(localhost:5555)/dbname?charset=utf8
36.     // user:password@/dbname
37.     // user:password@tcp([de:ad:be:ef::ca:fe]:80)/dbname
38.
39.     dbtemp, err := sql.Open("mysql", dbw.Dsn)
40.     dbw.Db = dbtemp
41.
42.     if err != nil {
43.         panic(err)
44.         return
45.     }
46.     defer dbw.Db.Close()
47.
48.     // 插入数据测试
49.     dbw.insertData()
50.
51.     // 删除数据测试
52.     dbw.deleteData()
53.
54.     // 修改数据测试
55.     dbw.editData()
56.
57.     // 查询数据测试
58.     dbw.queryData()
59.
60.     // 事务操作测试
61.     dbw.transaction()
62. }

```

每次db.Query操作后，都建议调用rows.Close()。因为 db.Query() 会从数据库连接池中获取一个连接，这个底层连接在结果集(rows)未关闭前会被标记为处于繁忙状态。当遍历读到最后一条记录时，会发生一个内部EOF错误，自动调用rows.Close()，但如果提前退出循环，rows不会关闭，连接不会回到连接池中，连接也不会关闭，则此连接会一直被占用。因此通常我们使用 defer rows.Close() 来确保数据库连接可以正确放回到连接池中。

插入数据：

```

1. // 插入数据, sql预编译
2. func (dbw *DbWorker) insertData() {
3.     stmt, _ := dbw.Db.Prepare(`INSERT INTO t_article_cate (cname, addtime,
4.         scope) VALUES (?, ?, ?)`)
5.     defer stmt.Close()
6.
7.
8.     // 通过返回的ret可以进一步查询本次插入数据影响的行数
9.     // RowsAffected和最后插入的Id(如果数据库支持查询最后插入Id)
10.    if err != nil {
11.        fmt.Printf("insert data error: %v\n", err)
12.        return
13.    }
14.    if LastInsertId, err := ret.LastInsertId(); nil == err {
15.        fmt.Println("LastInsertId:", LastInsertId)
16.    }
17.    if RowsAffected, err := ret.RowsAffected(); nil == err {
18.        fmt.Println("RowsAffected:", RowsAffected)
19.    }
20. }

```

删除数据:

```

1. // 删除数据, 预编译
2. func (dbw *DbWorker) deleteData() {
3.     stmt, err := dbw.Db.Prepare(`DELETE FROM t_article_cate WHERE cid=?`)
4.     ret, err := stmt.Exec(122)
5.     // 通过返回的ret可以进一步查询本次插入数据影响的行数RowsAffected和
6.     // 最后插入的Id(如果数据库支持查询最后插入Id).
7.     if err != nil {
8.         fmt.Printf("insert data error: %v\n", err)
9.         return
10.    }
11.    if RowsAffected, err := ret.RowsAffected(); nil == err {
12.        fmt.Println("RowsAffected:", RowsAffected)
13.    }
14. }

```

修改数据:

```

1. // 修改数据, 预编译
2. func (dbw *DbWorker) editData() {
3.     stmt, err := dbw.Db.Prepare(`UPDATE t_article_cate SET scope=? WHERE cid=?`
4.     `)
5.     ret, err := stmt.Exec(111, 123)
6.     // 通过返回的ret可以进一步查询本次插入数据影响的行数RowsAffected和
7.     // 最后插入的Id(如果数据库支持查询最后插入Id).
8.     if err != nil {
9.         fmt.Printf("insert data error: %v\n", err)
10.        return
11.    }
12.    if RowsAffected, err := ret.RowsAffected(); nil == err {
13.        fmt.Println("RowsAffected:", RowsAffected)
14.    }
15. }

```

查询数据:

```

1. // 查询数据, 预编译
2. func (dbw *DbWorker) queryData() {
3.     // 如果方法包含Query, 那么这个方法是用用于查询并返回rows的。其他用Exec()
4.     // 另外一种写法
5.     // rows, err := db.Query("select id, name from users where id = ?", 1)
6.     stmt, _ := dbw.Db.Prepare(`SELECT cid, cname, addtime, scope From
7.     t_article_cate where status=?`)
8.     //err = db.QueryRow("select name from users where id = ?", 1).Scan(&name)
9.     // 单行查询, 直接处理
10.    defer stmt.Close()
11.
12.    rows, err := stmt.Query(0)
13.    defer rows.Close()
14.    if err != nil {
15.        fmt.Printf("insert data error: %v\n", err)
16.        return
17.    }
18.
19.    // 构造scanArgs、values两个slice,
20.    // scanArgs的每个值指向values相应值的地址
21.    columns, _ := rows.Columns()
22.    fmt.Println(columns)
23.    rowMaps := make([]map[string]string, 9)
24.    values := make([]sql.RawBytes, len(columns))

```



```

23.     scans := make([]interface{}, len(columns))
24.     for i := range values {
25.         scans[i] = &values[i]
26.         scans[i] = &values[i]
27.     }
28.     i := 0
29.     for rows.Next() {
30.         //将行数据保存到record字典
31.         err = rows.Scan(scans...)
32.
33.         each := make(map[string]string, 4)
34.         // 由于是map引用，放在上层for时，rowMaps最终返回值是最后一条。
35.         for i, col := range values {
36.             each[columns[i]] = string(col)
37.         }
38.
39.         // 切片追加数据，索引位置有意思。不这样写就不是希望的样子。
40.         rowMaps = append(rowMaps[:i], each)
41.         fmt.Println(each)
42.         i++
43.     }
44.     fmt.Println(rowMaps)
45.
46.     for i, col := range rowMaps {
47.         fmt.Println(i, col)
48.     }
49.
50.     err = rows.Err()
51.     if err != nil {
52.         fmt.Printf(err.Error())
53.     }
54. }

```

事务处理：

db.Begin()开始事务，Commit() 或 Rollback()关闭事务。Tx从连接池中取出一个连接，在关闭之前都使用这个连接。Tx不能和DB层的BEGIN，COMMIT混合使用。

```

1. func (dbw *DbWorker) transaction() {
2.     tx, err := dbw.Db.Begin()
3.     if err != nil {
4.
5.         fmt.Printf("insert data error: %v\n", err)

```

```
6.         return
7.     }
8.     defer tx.Rollback()
9.     stmt, err := tx.Prepare(`INSERT INTO t_article_cate (cname, addtime,
scope) VALUES (?, ?, ?)`)
10.    if err != nil {
11.
12.        fmt.Printf("insert data error: %v\n", err)
13.        return
14.    }
15.
16.    for i := 100; i < 110; i++ {
17.        cname := strings.Join([]string{"栏目-", string(i)}, "-")
18.        _, err = stmt.Exec(cname, time.Now().Unix(), i+20)
19.        if err != nil {
20.            fmt.Printf("insert data error: %v\n", err)
21.            return
22.        }
23.    }
24.    err = tx.Commit()
25.    if err != nil {
26.        fmt.Printf("insert data error: %v\n", err)
27.        return
28.    }
29.    stmt.Close()
30. }
```

第四十章 LevelDB与BoltDB

《Go语言四十二章经》第四十章 LevelDB与BoltDB

作者：李骁

LevelDB 和 BoltDB 都是k/v数据库。

但LevelDB没有事务，LevelDB实现了一个日志结构化的merge tree。它将有序的key/value存储在不同文件的之中，通过db, _ := leveldb.OpenFile("db", nil)，在db目录下有很多数据文件，并通过“层级”把它们分开，并且周期性地将小的文件merge为更大的文件。这让其在随机写的时候会很快，但是读的时候却很慢。

这也让LevelDB的性能不可预知：但数据量很小的时候，它可能性能很好，但是当随着数据量的增加，性能只会越来越糟糕。而且做merge的线程也会在服务器上出现问题。

LSM树而且通过批量存储技术规避磁盘随机写入问题。 LSM树的设计思想非常朴素，它的原理是把一颗大树拆分成N棵小树， 它首先写入到内存中（内存没有寻道速度的问题，随机写的性能得到大幅提升），在内存中构建一颗有序小树，随着小树越来越大，内存的小树会flush到磁盘上。磁盘中的树定期可以做merge操作，合并成一棵大树，以优化读性能。

BoltDB会在数据文件上获得一个文件锁，所以多个进程不能同时打开同一个数据库。BoltDB使用一个单独的内存映射的文件(.db)，实现一个写入时拷贝的B+树，这能让读取更快。而且，BoltDB的载入时间很快，特别是在从crash恢复的时候，因为它不需要去通过读log去找到上次成功的事务，它仅仅从两个B+树的根节点读取ID。

BoltDB支持完全可序列化的ACID事务，让应用程序可以更简单的处理复杂操作。

BoltDB设计源于LMDB，具有以下特点：

- 直接使用API存取数据，没有查询语句；
- 支持完全可序列化的ACID事务，这个特性比LevelDB强；
- 数据保存在内存映射的文件里。没有wal、线程压缩和垃圾回收；
- 通过COW技术，可实现无锁的读写并发，但是无法实现无锁的写写并发，这就注定了读性能超高，但写性能一般，适合与读多写少的场景。
- 最后，BoltDB使用Golang开发，而且被应用于influxDB项目作为底层存储。

LMDB的全称是Lightning Memory-Mapped Database(快如闪电的内存映射数据库)，它的文件结构简单，包含一个数据文件和一个锁文件。

LMDB文件可以同时由多个进程打开，具有极高的数据存取速度，访问简单，不需要运行单独的数据库管理进程，只要在访问数据的代码里引用LMDB库，访问时给文件路径即可。

让系统访问大量小文件的开销很大，而LMDB使用内存映射的方式访问文件，使得文件内寻址的开销非常小，使用指针运算就能实现。数据库单文件还能减少数据集复制/传输过程的开销。

40.1 LevelDB

```

1. package kvdb
2.
3. import (
4.     "fmt"
5.
6.     "github.com/syndtr/goleveldb/leveldb"
7.     "github.com/syndtr/goleveldb/leveldb/util"
8. )
9.
10. func Leveldb() {
11.     db, _ := leveldb.OpenFile("db", nil)
12.
13.     defer db.Close()
14.     //读写数据库:
15.     _ = db.Put([]byte("key1"), []byte("好好检查"), nil)
16.     _ = db.Put([]byte("key2"), []byte("天天向上"), nil)
17.     _ = db.Put([]byte("key:3"), []byte("就会一个本事"), nil)
18.
19.     data, _ := db.Get([]byte("key1"), nil)
20.     fmt.Println(string(data))
21.
22.     //迭代数据库内容:
23.     iter := db.NewIterator(nil, nil)
24.     for iter.Next() {
25.         key := iter.Key()
26.         value := iter.Value()
27.         fmt.Println(string(key), string(value))
28.
29.     }
30.     iter.Release()
31.     iter.Error()
32.
33.     //Seek-then-Iterate:
34.     iter = db.NewIterator(nil, nil)
35.     for ok := iter.Seek([]byte("key:")); ok; ok = iter.Next() {
36.         // Use key/value.

```

```

37.         fmt.Println("Seek-then-Iterate:")
38.         fmt.Println(string(iter.Value()))
39.     }
40.     iter.Release()
41.
42.     //Iterate over subset of database content:
43.     iter = db.NewIterator(&util.Range{Start: []byte("key:"), Limit:
[]byte("xoo")}, nil)
44.     for iter.Next() {
45.         // Use key/value.
46.
47.         fmt.Println("Iterate over subset of database content:")
48.         fmt.Println(string(iter.Value()))
49.     }
50.     iter.Release()
51.
52.     //Iterate over subset of database content with a particular prefix:
53.     iter = db.NewIterator(util.BytesPrefix([]byte("key")), nil)
54.     for iter.Next() {
55.         // Use key/value.
56.
57.         fmt.Println("Iterate over subset of database content with a particular
prefix:")
58.         fmt.Println(string(iter.Value()))
59.     }
60.     iter.Release()
61.
62.     _ = iter.Error()
63.
64.     //批量写:
65.     batch := new(leveldb.Batch)
66.     batch.Put([]byte("foo"), []byte("value"))
67.     batch.Put([]byte("bar"), []byte("another value"))
68.     batch.Delete([]byte("baz"))
69.     _ = db.Write(batch, nil)
70.
71.     _ = db.Delete([]byte("key"), nil)
72. }

```

40.2 BoltDB

```

1. package kvdb
2.
3. import (
4.     "fmt"
5.     "log"
6.     "time"
7.
8.     "github.com/boltdb/bolt"
9. )
10.
11. func Boltldb() error {
12.     // Bolt 会在数据文件上获得一个文件锁，所以多个进程不能同时打开同一个数据库。
13.     // 打开一个已经打开的 Bolt 数据库将导致它挂起，直到另一个进程关闭它。
14.     // 为防止无限期等待，您可以将超时选项传递给Open()函数：
15.     db, err := bolt.Open("my.db", 0600, &bolt.Options{Timeout: 1 *
        time.Second})
16.     defer db.Close()
17.     if err != nil {
18.         log.Fatal(err)
19.     }
20.
21.     // 两种处理方式：读-写和只读操作，读-写方式开始于db.Update方法：
22.     // Bolt 一次只允许一个读写事务，但是一次允许多个只读事务。
23.     // 每个事务处理都有一个始终如一的数据视图
24.     err = db.Update(func(tx *bolt.Tx) error {
25.
26.         // 这里还有另外一层：k-v存储在bucket中，
27.         // 可以将bucket当做一个key的集合或者是数据库中的表。
28.         // （顺便提一句，buckets中可以包含其他的buckets，这将会相当有用）
29.         // Buckets 是键值对在数据库中的集合。所有在bucket中的key必须唯一，
30.         // 使用DB.CreateBucket() 函数建立buket
31.         //Tx.DeleteBucket() 删除bucket
32.         //b := tx.Bucket([]byte("MyBucket"))
33.         b, err := tx.CreateBucketIfNotExists([]byte("MyBucket"))
34.
35.
36.         //要将 key/value 对保存到 bucket，请使用 Bucket.Put() 函数：
37.         //这将在 MyBucket 的 bucket 中将 "answer" key的值设置为"42"。
38.         err = b.Put([]byte("answer"), []byte("42"))
39.         return err
40.     })
41.

```

```

42.      // 可以看到, 传入db.update函数一个参数, 在函数内部你可以get/set数据和处理error。
43.      // 如返回为nil, 事务就会从数据库得到一个commit, 但如果返回一个实际的错误, 则会做回滚,
44.      // 你在函数中做的事情都不会commit。这很自然, 因为你不需要人为地去关心事务的回滚,
45.      // 只需要返回一个错误, 其他的由Bolt去帮你完成。
46.      // 只读事务 只读事务和读写事务不应该相互依赖, 一般不应该在同一个例程中同时打开。
47.      // 这可能会导致死锁, 因为读写事务需要定期重新映射数据文件,
48.  // 但只有在只读事务处于打开状态时才能这样做。
49.
50.      // 批量读写事务. 每一次新的事物都需要等待上一次事物的结束,
51.  // 可以通过DB.Batch()批处理来完
52.      err = db.Batch(func(tx *bolt.Tx) error {
53.
54.          return nil
55.      })
56.
57.      //只读事务在db.View函数之中: 在函数中可以读取, 但是不能做修改。
58.      db.View(func(tx *bolt.Tx) error {
59.          //要检索这个value, 我们可以使用 Bucket.Get() 函数:
60.          //由于Get是有安全保障的, 所有不会返回错误, 不存在的key返回nil
61.          b := tx.Bucket([]byte("MyBucket"))
62.  //tx.Bucket([]byte("MyBucket")).Cursor() 可这样写
63.          v := b.Get([]byte("answer"))
64.          id, _ := b.NextSequence()
65.          fmt.Printf("The answer is: %s %d \n", v, id)
66.
67.          //游标遍历key
68.          c := b.Cursor()
69.
70.          for k, v := c.First(); k != nil; k, v = c.Next() {
71.              fmt.Printf("key=%s, value=%s\n", k, v)
72.          }
73.
74.          //游标上有以下函数:
75.          //First() 移动到第一个键。
76.          //Last() 移动到最后一个键。
77.          //Seek() 移动到特定的一个键。
78.          //Next() 移动到下一个键。
79.          //Prev() 移动到上一个键。
80.
81.          //Prefix 前缀扫描
82.          prefix := []byte("1234")
83.          for k, v := c.Seek(prefix); k != nil && bytes.HasPrefix(k, prefix); k,

```

```

    v = c.Next() {
84.         fmt.Printf("key=%s, value=%s\n", k, v)
85.     }
86.     return nil
87. })
88.
89. //范围查找
90. //另一个常见的用例是扫描范围，例如时间范围。如果你使用一个合适的时间编码，如rfc3339然后
    可以查询特定日期范围的数据：
91. db.View(func(tx *bolt.Tx) error {
92.     // Assume our events bucket exists and has RFC3339 encoded time keys.
93.     c := tx.Bucket([]byte("Events")).Cursor()
94.
95.     // Our time range spans the 90's decade.
96.     min := []byte("1990-01-01T00:00:00Z")
97.     max := []byte("2000-01-01T00:00:00Z")
98.
99.     // Iterate over the 90's.
100.    for k, v := c.Seek(min); k != nil && bytes.Compare(k, max) <= 0; k, v =
    c.Next() {
101.        fmt.Printf("%s: %s\n", k, v)
102.    }
103.    return nil
104. })
105.
106. //如果你知道所在桶中拥有键，你也可以使用ForEach()来迭代：
107. db.View(func(tx *bolt.Tx) error {
108.     b := tx.Bucket([]byte("MyBucket"))
109.
110.     b.ForEach(func(k, v []byte) error {
111.         fmt.Printf("key=%s, value=%s\n", k, v)
112.         return nil
113.     })
114.     return nil
115. })
116.
117. //事务处理
118. // 开始事务
119. tx, err := db.Begin(true)
120. if err != nil {
121.     return err
122. }

```



```

123.     defer tx.Rollback()
124.
125.     // 使用事务...
126.     _, err = tx.CreateBucket([]byte("MyBucket"))
127.     if err != nil {
128.         return err
129.     }
130.
131.     // 事务提交
132.     if err = tx.Commit(); err != nil {
133.         return err
134.     }
135.     return err
136.
137.     //还可以在一个键中存储一个桶，以创建嵌套的桶：
138.     //func (*Bucket) CreateBucket(key []byte) (*Bucket, error)
139.     //func (*Bucket) CreateBucketIfNotExists(key []byte) (*Bucket, error)
140.     //func (*Bucket) DeleteBucket(key []byte) error
141. }
142.
143. //备份 curl http://localhost/backup > my.db
144. func BackupHandleFunc(w http.ResponseWriter, req *http.Request) {
145.     err := db.View(func(tx *bolt.Tx) error {
146.         w.Header().Set("Content-Type", "application/octet-stream")
147.         w.Header().Set("Content-Disposition", `attachment; filename="my.db"`)
148.         w.Header().Set("Content-Length", strconv.Itoa(int(tx.Size())))
149.         _, err := tx.WriteTo(w)
150.         return err
151.     })
152.     if err != nil {
153.         http.Error(w, err.Error(), http.StatusInternalServerError)
154.     }
155. }
156.
157. //桶的自增
158. //利用nextsequence()功能，你可以让boltdb生成序列作为你键值对的唯一标识。见下面的示例。
159. func (s *Store) CreateUser(u *User) error {
160.     return s.db.Update(func(tx *bolt.Tx) error {
161.         // 创建users桶
162.         b := tx.Bucket([]byte("users"))
163.
164.         // 生成自增序列

```

```
165.         id, _ = b.NextSequence()
166.         u.ID = int(id)
167.
168.         // Marshal user data into bytes.
169.         buf, err := Json.Marshal(u)
170.         if err != nil {
171.             return err
172.         }
173.         // Persist bytes to users bucket.
174.         return b.Put(itob(u.ID), buf)
175.     })
176. }
177.
178. // itob returns an 8-byte big endian representation of v.
179. func itob(v int) []byte {
180.     b := make([]byte, 8)
181.     binary.BigEndian.PutUint64(b, uint64(v))
182.     return b
183. }
184.
185. type User struct {
186.     ID int
187. }
```

第四十一章 网络爬虫

《Go语言四十二章经》第四十一章 网络爬虫

作者：李骁

41.1 go-colly

go-colly是用Go实现的网络爬虫框架。go-colly快速优雅，在单核上每秒可以发起1K以上请求；以回调函数的形式提供了一组接口，可以实现任意类型的爬虫。

Colly 特性：

清晰的API

快速（单个内核上的请求数大于1k）

管理每个域的请求延迟和最大并发数

自动cookie 和会话处理

同步/异步/并行抓取

高速缓存

自动处理非Unicode的编码

Robots.txt 支持

下面是官方提供的抓取例子：

```
1. package main
2.
3. import (
4.     "fmt"
5.
6.     "github.com/gocolly/colly"
7. )
8.
9. func main() {
10.     c := colly.NewCollector()
11.
12.     // Find and visit all links
13.     c.OnHTML("a[href]", func(e *colly.HTMLElement) {
14.         e.Request.Visit(e.Attr("href"))
15.     })
16.
```

```

17.     c.OnRequest(func(r *colly.Request) {
18.         fmt.Println("Visiting", r.URL)
19.     })
20.
21.     c.Visit("http://go-colly.org/")
22. }

```

程序输出：

```

1.  Visiting http://go-colly.org/
2.  Visiting http://go-colly.org/docs/
3.  Visiting http://go-colly.org/articles/
4.  Visiting http://go-colly.org/services/
5.  Visiting http://go-colly.org/datasets/
6.  .....

```

Colly大致的使用说明：

在代码中导入包：

```

1.  import "github.com/gocolly/colly"

```

colly的主体是Collector对象，管理网络通信和负责在作业运行时执行附加的回掉函数。使用colly需要先初始化Collector：

```

1.  c := colly.NewCollector()

```

可以向colly附加各种不同类型的回调函数，来控制收集作业或获取信息：

回调函数的调用顺序如下：

1. OnRequest
在发起请求前被调用
2. OnError
请求过程中如果发生错误被调用
3. OnResponse
收到回复后被调用
4. OnHTML
在OnResponse之后被调用，如果收到的内容是HTML
5. OnScraped

在OnHTML之后被调用

41.2 goquery

colly框架配合goquery库，功能更加强大。goquery将jQuery的语法和特性引入到了Go语言中，可以更灵活地选择采集内容的数据项。

```
1. package main
2.
3. import (
4.     "bytes"
5.     "fmt"
6.     "log"
7.     "net/url"
8.     "time"
9.
10.    "github.com/PuerkitoBio/goquery"
11.    "github.com/gocolly/colly"
12. )
13.
14. func main() {
15.     urlstr := "http://metalsucks.net"
16.     u, err := url.Parse(urlstr)
17.     if err != nil {
18.         log.Fatal(err)
19.     }
20.     c := colly.NewCollector()
21.     c.SetRequestTimeout(100 * time.Second)
22.     // 指定Agent信息
23.     c.UserAgent = "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/63.0.3239.108 Safari/537.36"
24.     c.OnRequest(func(r *colly.Request) {
25.         r.Headers.Set("Host", u.Host)
26.         r.Headers.Set("Connection", "keep-alive")
27.         r.Headers.Set("Accept", "*/*")
28.         r.Headers.Set("Origin", u.Host)
29.         r.Headers.Set("Referer", urlstr)
30.         r.Headers.Set("Accept-Encoding", "gzip, deflate")
31.         r.Headers.Set("Accept-Language", "zh-CN, zh;q=0.9")
32.     })
33.
34.     c.OnResponse(func(resp *colly.Response) {
35.         // 读取url内容 colly读取的内容传入给goquery
```

```

36.         htmlDoc, err :=
goquery.NewDocumentFromReader(bytes.NewReader(resp.Body))
37.         if err != nil {
38.             log.Fatal(err)
39.         }
40.
41.         // 找到抓取项
42.         htmlDoc.Find(".sidebar-reviews article .content-block").Each(func(i
int, s *goquery.Selection) {
43.             band := s.Find("a").Text()
44.             title := s.Find("i").Text()
45.             fmt.Printf("Review %d: %s - %s\n", i, band, title)
46.         })
47.     })
48.     c.OnError(func(resp *colly.Response, errHttp error) {
49.         err = errHttp
50.     })
51.     err = c.Visit(urlstr)
52. }

```

程序输出：

```

1. Review 0: Obscura - Diluvium
2. Review 1: Skeletonwitch - Devouring Radiant Light
3. Review 2: Deafheaven - Ordinary Corrupt Human Love
4. Review 3: Between the Buried and Me - Automata II
5. Review 4: Chelsea Grin - Eternal Nightmare

```

Colly + goquery 是抓取网络内容的利器，使用上极其方便。如今动态渲染的页面越来越多，爬虫们或多或少都需要用到headless browser来渲染待爬取的页面，这里推荐chromedp，开源网址：<https://github.com/chromedp/chromedp>

第四十二章 WEB框架(Gin)

《Go语言四十二章经》第四十二章 WEB框架(Gin)

作者：李骁

42.1 有关于Gin

Gin是Go语言写的一个web框架，API性能超强，运行速度号称较httprouter要快40x。开源网址：<https://github.com/gin-gonic/gin>

下载安装gin包：

```
1. go get -u github.com/gin-gonic/gin
```

一个简单的例子：

```
1. package main
2.
3. import "github.com/gin-gonic/gin"
4.
5. func main() {
6.     r := gin.Default()
7.     r.GET("/ping", func(c *gin.Context) {
8.         c.JSON(200, gin.H{
9.             "message": "pong",
10.        })
11.    })
12.    r.Run() // listen and serve on 0.0.0.0:8080
13. }
```

编译运行程序，打开浏览器，访问 <http://localhost:8080/ping>

页面显示：

```
1. {"message":"pong"}
```

以Json格式输出了数据。

gin的功能不只是简单输出Json数据。它是一个轻量级的WEB框架，支持RestFull风格API，支持

GET, POST, PUT, PATCH, DELETE, OPTIONS 等http方法, 支持文件上传, 分组路由, Multipart/Urlencoded FORM, 以及支持JsonP, 参数处理等等功能, 这些都和WEB紧密相关, 通过提供这些功能, 使开发人员更方便地处理WEB业务。

42.2 Gin实际应用

接下来使用Gin作为框架来搭建一个拥有静态资源站点, 动态WEB站点, 以及RESTFull API接口站点 (可专门作为手机APP应用提供服务使用) 组成的, 亦可根据情况分拆这套系统, 每种功能独立出来单独提供服务。

下面按照一套系统但采用分站点来说明, 首先是整个系统的目录结构, website目录下面static是资源类文件, 为静态资源站点专用; photo目录是UGC上传图片目录, tpl是动态站点的模板。当然这个目录结构是一种约定, 你可以根据情况来修改。整个项目已经开源, 你可以访问来详细了

解: <https://github.com/ffhelicopter/tmm>

具体每个站点的功能怎么实现呢? 请看下面有关每个功能的讲述:

一: 静态资源站点

一般网站开发中, 我们会考虑把js, css, 以及资源图片放在一起, 作为静态站点部署在CDN, 提升响应速度。采用Gin实现起来非常简单, 当然也可以使用net/http包轻松实现, 但使用Gin会更方便。

不管怎么样, 使用Go开发, 我们可以不用花太多时间在WEB服务环境搭建上, 程序启动就直接可以提供WEB服务了。

```
1. package main
2.
3. import (
4.     "net/http"
5.     "github.com/gin-gonic/gin"
6. )
7.
8. func main() {
9.     router := gin.Default()
10.
11.     // 静态资源加载, 本例为css, js以及资源图片
12.     router.StaticFS("/public",
13.         http.Dir("D:/goproject/src/github.com/ffhelicopter/tmm/website/static"))
14.     router.StaticFile("/favicon.ico", "./resources/favicon.ico")
15.
16.     // Listen and serve on 0.0.0.0:80
17.     router.Run(":80")
```



```
17. }
```

首先需要是生成一个Engine，这是gin的核心，默认带有Logger 和 Recovery 两个中间件。

```
1. router := gin.Default()
```

StaticFile 是加载单个文件，而StaticFS 是加载一个完整的目录资源：

```
1. func (group *RouterGroup) StaticFile(relativePath, filepath string) IRoutes
2. func (group *RouterGroup) StaticFS(relativePath string, fs http.FileSystem)
   IRoutes
```

这些目录下资源是可以随时更新，而不用重新启动程序。现在编译运行程序，静态站点就可以正常访问了。

访问<http://localhost/public/images/logo.jpg> 图片加载正常。每次请求响应都会在服务端有日志产生，包括响应时间，加载资源名称，响应状态值等等。

二：动态站点

如果需要动态交互的功能，比如发一段文字+图片上传。由于这些功能出来前端页面外，还需要服务端程序一起来实现，而且迭代需要经常需要修改代码和模板，所以把这些统一放在一个大目录下，姑且称动态站点。

tpl是动态站点所有模板的根目录，这些模板可调用静态资源站点的css，图片等；photo是图片上传后存放的目录。

```
1. package main
2.
3. import (
4.     "context"
5.     "log"
6.     "net/http"
7.     "os"
8.     "os/signal"
9.     "time"
10.
11.     "github.com/ffhelicopter/tmm/handler"
12.
13.     "github.com/gin-gonic/gin"
14. )
15.
```

```

16. func main() {
17.     router := gin.Default()
18.
19.     // 静态资源加载, 本例为css, js以及资源图片
20.     router.StaticFS("/public",
http.Dir("D:/goproject/src/github.com/ffhelicopter/tmm/website/static"))
21.     router.StaticFile("/favicon.ico", "./resources/favicon.ico")
22.
23.     // 导入所有模板, 多级目录结构需要这样写
24.     router.LoadHTMLGlob("website/tpl/*/*")
25.
26.     // website分组
27.     v := router.Group("/")
28.     {
29.
30.         v.GET("/index.html", handler.IndexHandler)
31.         v.GET("/add.html", handler.AddHandler)
32.         v.POST("/postme.html", handler.PostmeHandler)
33.     }
34.
35.     // router.Run(":80")
36.     // 这样写就可以了, 下面所有代码 (go1.8+) 是为了优雅处理重启等动作。
37.     srv := &http.Server{
38.         Addr:         ":80",
39.         Handler:       router,
40.         ReadTimeout:   30 * time.Second,
41.         WriteTimeout:  30 * time.Second,
42.     }
43.
44.     go func() {
45.         // 监听请求
46.         if err := srv.ListenAndServe(); err != nil && err !=
http.ErrServerClosed {
47.             log.Fatalf("listen: %s\n", err)
48.         }
49.     }()
50.
51.     // 优雅Shutdown (或重启) 服务
52.     quit := make(chan os.Signal)
53.     signal.Notify(quit, os.Interrupt) // syscall.SIGKILL
54.     <-quit
55.     log.Println("Shutdown Server ...")

```

```

56.
57.     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
58.     defer cancel()
59.     if err := srv.Shutdown(ctx); err != nil {
60.         log.Fatal("Server Shutdown:", err)
61.     }
62.     select {
63.     case <-ctx.Done():
64.     }
65.     log.Println("Server exiting")
66. }

```

在动态站点实现中，引入WEB分组以及优雅重启这两个功能。WEB分组功能可以通过不同的入口根路径来区别不同的模块，这里我们可以访问：<http://localhost/index.html>。如果新增一个分组，比如：

```
1. v := router.Group("/login")
```

我们可以访问：<http://localhost/login/xxxx>，xxx是我们在v.GET方法或v.POST方法中的路径。

```

1.     // 导入所有模板，多级目录结构需要这样写
2.     router.LoadHTMLGlob("website/tpl/*/*")
3.
4.     // website分组
5.     v := router.Group("/")
6.     {
7.
8.         v.GET("/index.html", handler.IndexHandler)
9.         v.GET("/add.html", handler.AddHandler)
10.        v.POST("/postme.html", handler.PostmeHandler)
11.    }

```

通过router.LoadHTMLGlob("website/tpl/*/*") 导入模板根目录下所有的文件。在前面有讲过html/template 包的使用，这里模板文件中的语法和前面一致。

```
1.     router.LoadHTMLGlob("website/tpl/*/*")
```

比如v.GET("/index.html", handler.IndexHandler)，通过访问<http://localhost/index.html> 这个URL，实际由handler.IndexHandler来处理。而在tmm目录下的handler存放了package handler 文件。在包里定义了IndexHandler函数，它使用了

index.html模板。

```

1. func IndexHandler(c *gin.Context) {
2.     c.HTML(http.StatusOK, "index.html", gin.H{
3.         "Title": "作品欣赏",
4.     })
5. }
```

index.html模板：

```

1. <!DOCTYPE html>
2. <html>
3. <head>
4. {{template "header" .}}
5. </head>
6. <body>
7.
8. <!--导航-->
9. <div class="feeds">
10.     <div class="top-nav">
11.         <a href="/index.html" class="active">欣赏</a>
12.         <a href="/add.html" class="add-btn">
13.             <svg class="icon" aria-hidden="true">
14.                 <use xlink:href="#icon-add"></use>
15.             </svg>
16.             发布
17.         </a>
18.     </div>
19.     <input type="hidden" id="showmore" value="{ $showmore }">
20.     <input type="hidden" id="page" value="{ $page }">
21.     <!--</div>-->
22. </div>
23. <script type="text/javascript">
24.     var done = true;
25.     $(window).scroll(function(){
26.         var scrollTop = $(window).scrollTop();
27.         var scrollHeight = $(document).height();
28.         var windowHeight = $(window).height();
29.         var showmore = $("#showmore").val();
30.         if(scrollTop + windowHeight + 300 >= scrollHeight && showmore == 1 &&
done){
31.             var page = $("#page").val();
```

```

32.         done = false;
33.         $.get("{:U('Product/listsAjax')}", { page : page }, function(json)
34.         {
35.             if (json.rs != "") {
36.                 $(".feeds").append(json.rs);
37.                 $("#showmore").val(json.showmore);
38.                 $("#page").val(json.page);
39.                 done = true;
40.             }
41.         }, 'json');
42.     });
43. </script>
44.     <script src="//at.alicdn.com/t/font_ttszo9nm0wwmi.js"></script>
45. </body>
46. </html>

```

在index.html模板中,通过`{{template "header" .}}`语句,嵌套了header.html模板。

header.html模板:

```

1.  {{ define "header" }}
2.     <meta charset="UTF-8">
3.     <meta name="viewport" content="width=device-width, initial-scale=1,
4.         maximum-scale=1, minimum-scale=1, user-scalable=no, minimal-ui">
5.     <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
6.     <meta name="format-detection" content="telephone=no,email=no">
7.     <title>{{ .Title }}</title>
8.     <link rel="stylesheet" href="/public/css/common.css">
9.     <script src="/public/lib/jquery-3.1.1.min.js"></script>
10.    <script src="/public/lib/jquery.cookie.js"></script>
11.    <link href="/public/css/font-awesome.css?v=4.4.0" rel="stylesheet">
12.  {{ end }}

```

`{{ define "header" }}` 让我们在模板嵌套时直接使用header名字,而在index.html中的`{{template "header" .}}` 注意“.”,可以使参数嵌套传递,否则不能传递,比如这里的Title。

现在我们访问 <http://localhost/index.html> ,可以看到浏览器显示Title 是“作品欣赏”,这个Title是通过IndexHandler来指定的。

接下来点击“发布”按钮,我们进入发布页面,上传图片,点击“完成”提交,会提示我们成功上传图片。可以在photo目录中看到刚才上传的图片。

注意：

由于在本人在发布到github的代码中，在处理图片上传的代码中，除了服务器存储外，还实现了IPFS发布存储，如果不需要IPFS，请注释相关代码。

有关IPFS：

IPFS本质上是一种内容可寻址、版本化、点对点超媒体的分布式存储、传输协议，目标是补充甚至取代过去20年里使用的超文本媒体传输协议（HTTP），希望构建更快、更安全、更自由的互联网时代。

IPFS 不算严格意义上区块链项目，是一个去中心化存储解决方案，但有些区块链项目通过它来做存储。

IPFS项目有在github上开源，Go语言实现哦，可以关注并了解。

优雅重启在迭代中有较好的实际意义，每次版本发布，如果直接停服务在部署重启，对业务还是有蛮大的影响，而通过优雅重启，这方面的体验可以做得更好些。这里ctrl + c 后过5秒服务停止。

三：中间件的使用，在API中可能使用限流，身份验证等

Go 语言中net/http设计的一大特点就是特别容易构建中间件。 gin也提供了类似的中间件。需要注意的是在gin里面中间件只对注册过的路由函数起作用。

而对于分组路由，嵌套使用中间件，可以限定中间件的作用范围。大致分为全局中间件，单个路由中间件和分组中间件。

即使是全局中间件，其使用前的代码不受影响。 也可在handler中局部使用，具体见api.GetUser。

在高并发场景中，有时候需要用到限流降速的功能，这里引入一个限流中间件。有关限流方法常见有两种，具体可自行研究，这里只讲使用。

导入 import "github.com/didip/tollbooth/limiter" 包，在上面代码基础上增加如下语句：

```
1.      //rate-limit 限流中间件
2.      lmt := tollbooth.NewLimiter(1, nil)
3.      lmt.SetMessage("服务繁忙，请稍后再试...")
```

并修改

```
1.  v.GET("/index.html", LimitHandler(lmt), handler.IndexHandler)
```

当F5刷新刷新<http://localhost/index.html> 页面时，浏览器会显示：服务繁忙，请稍后再试...

限流策略也可以为IP：

```
1. tollbooth.LimitByKey(lmt, []string{"127.0.0.1", "/"})
```

更多限流策略的配置，可以进一步github.com/didip/tollbooth/limiter 了解。

四：RestFull API接口

前面说了在gin里面可以采用分组来组织访问URL，这里RestFull API需要给出不同的访问URL来和动态站点区分，所以新建了一个分组v1。

在浏览器中访问<http://localhost/v1/user/1100000/>

这里对v1.GET("/user/:id/*action", LimitHandler(lmt), api.GetUser) 进行了限流控制，所以如果频繁访问上面地址也将会有限制，这在API接口中非常有作用。

通过 api这个包，来实现所有有关API的代码。在GetUser函数中，通过读取mysql数据库，查找到对应userid的用户信息，并通过Json格式返回给client。

在api.GetUser中，设置了一个局部中间件：

```
1. //CORS 局部CORS，可在路由中设置全局的CORS
2. c.Writer.Header().Add("Access-Control-Allow-Origin", "*")
```

gin关于参数的处理，api包中api.go文件中有简单说明，限于篇幅原因，就不在此展开。这个项目的详细情况，请访问 <https://github.com/ffhelicopter/tmm> 了解。有关gin的更多信息，请访问<https://github.com/gin-gonic/gin>，该开源项目比较活跃，可以关注。

完整mian.go代码：

```
1. package main
2.
3. import (
4.     "context"
5.     "log"
6.     "net/http"
7.     "os"
8.     "os/signal"
9.     "time"
10.
11.     "github.com/didip/tollbooth"
12.     "github.com/didip/tollbooth/limiter"
13.     "github.com/ffhelicopter/tmm/api"
14.     "github.com/ffhelicopter/tmm/handler"
15.
```

```

16.     "github.com/gin-gonic/gin"
17. )
18.
19. // 定义全局的CORS中间件
20. func Cors() gin.HandlerFunc {
21.     return func(c *gin.Context) {
22.         c.Writer.Header().Add("Access-Control-Allow-Origin", "")
23.         c.Next()
24.     }
25. }
26.
27. func LimitHandler(lmt *limiter.Limiter) gin.HandlerFunc {
28.     return func(c *gin.Context) {
29.         httpError := tollbooth.LimitByRequest(lmt, c.Writer, c.Request)
30.         if httpError != nil {
31.             c.Data(httpError.StatusCode, lmt.GetMessageContentType(),
32.                 []byte(httpError.Message))
33.             c.Abort()
34.         } else {
35.             c.Next()
36.         }
37.     }
38.
39. func main() {
40.     gin.SetMode(gin.ReleaseMode)
41.     router := gin.Default()
42.
43.     // 静态资源加载, 本例为css, js以及资源图片
44.     router.StaticFS("/public",
45.         http.Dir("D:/goproject/src/github.com/ffhelicopter/tmm/website/static"))
46.     router.StaticFile("/favicon.ico", "./resources/favicon.ico")
47.
48.     // 导入所有模板, 多级目录结构需要这样写
49.     router.LoadHTMLGlob("website/tpl/*/*")
50.     // 也可以根据handler, 实时导入模板。
51.
52.     // website分组
53.     v := router.Group("/")
54.     {
55.         v.GET("/index.html", handler.IndexHandler)

```



```

56.         v.GET("/add.html", handler.AddHandler)
57.         v.POST("/postme.html", handler.PostmeHandler)
58.     }
59.
60.     // 中间件 golang的net/http设计的一大特点就是特别容易构建中间件。
61.     // gin也提供了类似的中间件。需要注意的是中间件只对注册过的路由函数起作用。
62.     // 对于分组路由, 嵌套使用中间件, 可以限定中间件的作用范围。
63.     // 大致分为全局中间件, 单个路由中间件和群组中间件。
64.
65.     // 使用全局CORS中间件。
66.     // router.Use(Cors())
67.     // 即使是全局中间件, 在use前的代码不受影响
68.     // 也可在handler中局部使用, 见api.GetUser
69.
70.     //rate-limit 中间件
71.     lmt := tollbooth.NewLimiter(1, nil)
72.     lmt.SetMessage("服务繁忙, 请稍后再试...")
73.
74.     // API分组(RESTFULL)以及版本控制
75.     v1 := router.Group("/v1")
76.     {
77.         // 下面是群组中间的用法
78.         // v1.Use(Cors())
79.
80.         // 单个中间件的用法
81.         // v1.GET("/user/:id/*action", Cors(), api.GetUser)
82.
83.         // rate-limit
84.         v1.GET("/user/:id/*action", LimitHandler(lmt), api.GetUser)
85.
86.         //v1.GET("/user/:id/*action", Cors(), api.GetUser)
87.         // AJAX OPTIONS , 下面是有关OPTIONS用法的示例
88.         // v1.OPTIONS("/users", OptionsUser)      // POST
89.         // v1.OPTIONS("/users/:id", OptionsUser)  // PUT, DELETE
90.     }
91.
92.     srv := &http.Server{
93.         Addr:         ":80",
94.         Handler:      router,
95.         ReadTimeout:  30 * time.Second,
96.         WriteTimeout: 30 * time.Second,
97.     }

```

```
98.
99.     go func() {
100.         if err := srv.ListenAndServe(); err != nil && err !=
http.ErrServerClosed {
101.             log.Fatalf("listen: %s\n", err)
102.         }
103.     }()
104.
105.     // 优雅Shutdown（或重启）服务
106.     // 5秒后优雅Shutdown服务
107.     quit := make(chan os.Signal)
108.     signal.Notify(quit, os.Interrupt) //syscall.SIGKILL
109.     <-quit
110.     log.Println("Shutdown Server ...")
111.
112.     ctx, cancel := context.WithTimeout(context.Background(), 5*time.Second)
113.     defer cancel()
114.     if err := srv.Shutdown(ctx); err != nil {
115.         log.Fatal("Server Shutdown:", err)
116.     }
117.     select {
118.     case <-ctx.Done():
119.     }
120.     log.Println("Server exiting")
121. }
```