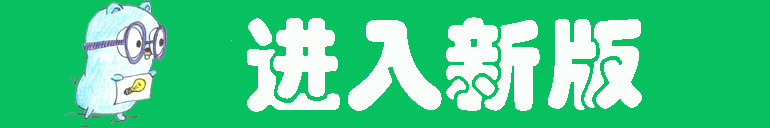
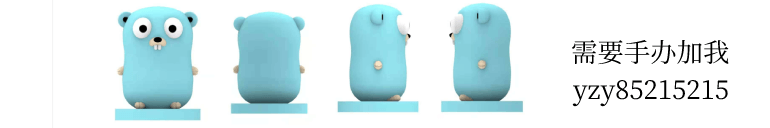
**一、前景**

**1. go语言介绍**

*欢迎大家加我微信大家一起学习****yzy85215215****(需要视频教程也可以加我)*

很多小伙伴不知道怎么学习go，需要掌握哪些知识点，据此我找了一个学习线路图**[go学习线路图](https://www.topgoer.com/%E5%BC%80%E6%BA%90/go%E5%AD%A6%E4%B9%A0%E7%BA%BF%E8%B7%AF%E5%9B%BE.html" \o "go学习线路图)**

[](http://www.topgoer.cn/)

[](https://www.topgoer.cn/blog-168.html)

**1.1.1. Go语言为并发而生**

go语言（或 Golang）是Google开发的开源编程语言，诞生于2006年1月2日下午15点4分5秒，于2009年11月开源，2012年发布go稳定版。Go语言在多核并发上拥有原生的设计优势，Go语言从底层原生支持并发，无须第三方库、开发者的编程技巧和开发经验。

go是非常年轻的一门语言，它的主要目标是“兼具Python 等动态语言的开发速度和C/C++等编译型语言的性能与安全性”

很多公司，特别是中国的互联网公司，即将或者已经完成了使用 Go 语言改造旧系统的过程。经过 Go 语言重构的系统能使用更少的硬件资源获得更高的并发和I/O吞吐表现。充分挖掘硬件设备的潜力也满足当前精细化运营的市场大环境。

Go语言的并发是基于 goroutine 的，goroutine 类似于线程，但并非线程。可以将 goroutine 理解为一种虚拟线程。Go 语言运行时会参与调度 goroutine，并将 goroutine 合理地分配到每个 CPU 中，最大限度地使用CPU性能。开启一个goroutine的消耗非常小（大约2KB的内存），你可以轻松创建数百万个goroutine。

goroutine的特点：

1.`goroutine`具有可增长的分段堆栈。这意味着它们只在需要时才会使用更多内存。

2.`goroutine`的启动时间比线程快。

3.`goroutine`原生支持利用channel安全地进行通信。

4.`goroutine`共享数据结构时无需使用互斥锁。

**1.1.2. Go语言简单易学**

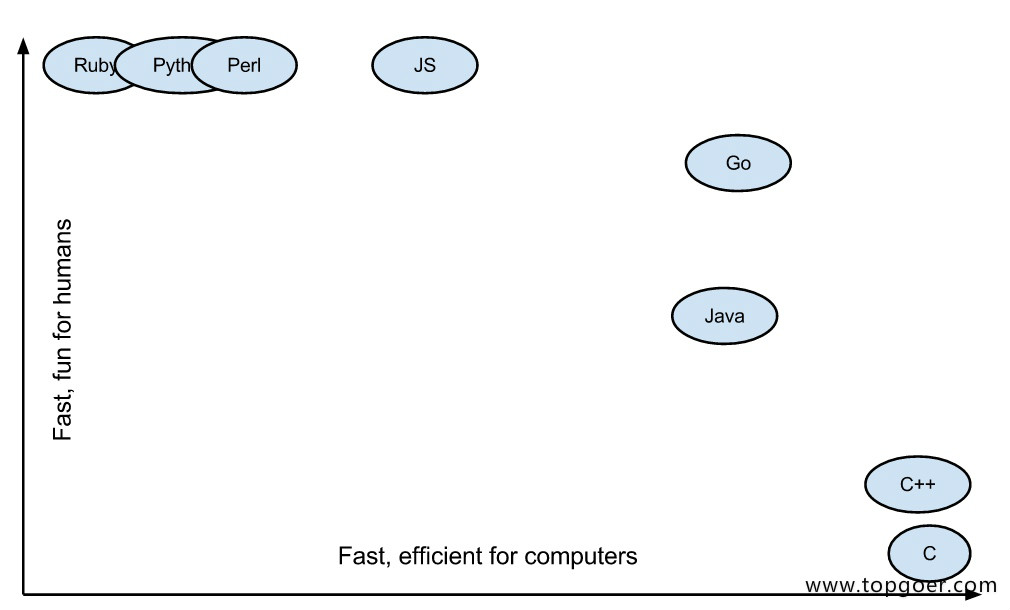
**语法简洁**

Go 语言简单易学，学习曲线平缓，不需要像 C/C++ 语言动辄需要两到三年的学习期。Go 语言被称为“互联网时代的C语言”。Go 语言的风格类似于C语言。其语法在C语言的基础上进行了大幅的简化，去掉了不需要的表达式括号，循环也只有 for 一种表示方法，就可以实现数值、键值等各种遍历。

**代码风格统一**

Go 语言提供了一套格式化工具——go fmt。一些 Go 语言的开发环境或者编辑器在保存时，都会使用格式化工具进行修改代码的格式化，这样就保证了不同开发者提交的代码都是统一的格式。(吐槽下：再也不用担心那些看不懂的黑魔法了…)

**开发效率高**



Go语言实现了开发效率与执行效率的完美结合，让你像写Python代码（效率）一样编写C代码（性能）。

**1.1.3. 使用go的公司**

* Google
  + <https://github.com/kubernetes/kubernetes>
* Facebook
  + <https://github.com/facebookgo>
* 腾讯
* 百度
* 360开源日志系统
  + <https://github.com/Qihoo360/poseidon>

**1.1.4. go适合做什么**

* 服务端开发
* 分布式系统，微服务
* 网络编程
* 区块链开发
* 内存KV数据库，例如boltDB、levelDB
* 云平台

**1.1.5. 学习Go语言的前景**

目前Go语言已经⼴泛应用于人工智能、云计算开发、容器虚拟化、⼤数据开发、数据分析及科学计算、运维开发、爬虫开发、游戏开发等领域。

Go语言简单易学，天生支持并发，完美契合当下高并发的互联网生态。Go语言的岗位需求持续高涨，目前的Go程序员数量少，待遇好。

抓住趋势，要学会做一个领跑者而不是跟随者。

国内Go语言的需求潜力巨大，目前无论是国内大厂还是新兴互联网公司基本上都会有Go语言的岗位需求。

**二、开发环境**

**1. Go的安装**

**1.1. 下载地址**

Go官网下载地址：[https://golang.org/dl/](https://golang.org/dl/" \t "_blank) (打开有点慢)

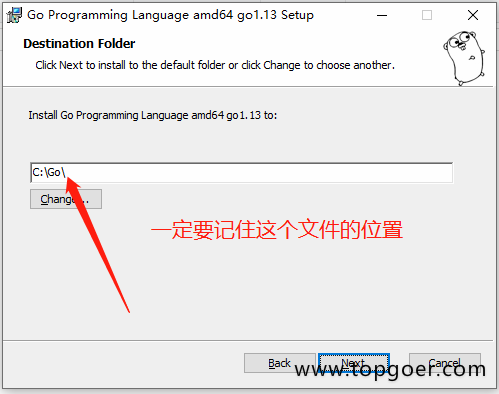
**1.2. Windows安装**



双击文件



一定要记住这个文件的位置后面还有用



**2.3. Linux下安装**

1.SSH远程登录你的linux服务器

2.安装 mercurial包

[root@localhost ~]# yum install mercurial

3.安装git包

[root@localhost ~]# yum install git

4.安装gcc

[root@localhost ~]# yum install gcc

5.下载Go的压缩包:(可选择最新的Go版本)

[root@localhost ~]# cd /usr/local/

[root@localhost local]# wget https://go.googlecode.com/files/go1.13.linux-amd64.tar.gz

注意：如果不能翻墙，去go语言资源站 下载相应的包。然后通过ftp上传到此目录。

6.下载完成 or ftp上传完成，用tar 命令来解压压缩包。

[root@localhost local]# tar -zxvf go1.13.linux-amd64.tar.gz

7.建立Go的工作空间（workspace，也就是GOPATH环境变量指向的目录）

GO代码必须在工作空间内。工作空间是一个目录，其中包含三个子目录：

src ---- 里面每一个子目录，就是一个包。包内是Go的源码文件

pkg ---- 编译后生成的，包的目标文件

bin ---- 生成的可执行文件

这里，我们在/home目录下, 建立一个名为go(可以不是go, 任意名字都可以)的文件夹， 然后再建立三个子文件夹(子文件夹名必须为src、pkg、bin)。

[root@localhost local]# cd /home/

[root@localhost home]# mkdir go

[root@localhost home]# cd go/

[root@localhost go]# mkdir bin

[root@localhost go]# mkdir src

[root@localhost go]# mkdir pkg

8.添加PATH环境变量and设置GOPATH环境变量

[root@localhost go]# vi /etc/profile

加入下面这三行:

export GOROOT=/usr/local/go ##Golang安装目录

export PATH=$GOROOT/bin:$PATH

export GOPATH=/home/go ##Golang项目目录

保存后，执行以下命令，使环境变量立即生效:

[root@localhost go]# source /etc/profile ##刷新环境变量

至此，Go语言的环境已经安装完毕。

9.验证一下是否安装成功，如果出现下面的信息说明安装成功了

[root@localhost go]# go version ##查看go版本

go version go1.13 linux/amd64

10.查看Go语言的环境信息

[root@localhost go]# go env

**2.4. Mac下安装**

没有Mac没有试过不知道怎么安装

**2.5. 检查**

上一步安装过程执行完毕后，可以打开终端窗口，输入go version命令，查看安装的Go版本。

## 2. 配置GOPATH

GOPATH是一个环境变量，用来表明你写的go项目的存放路径

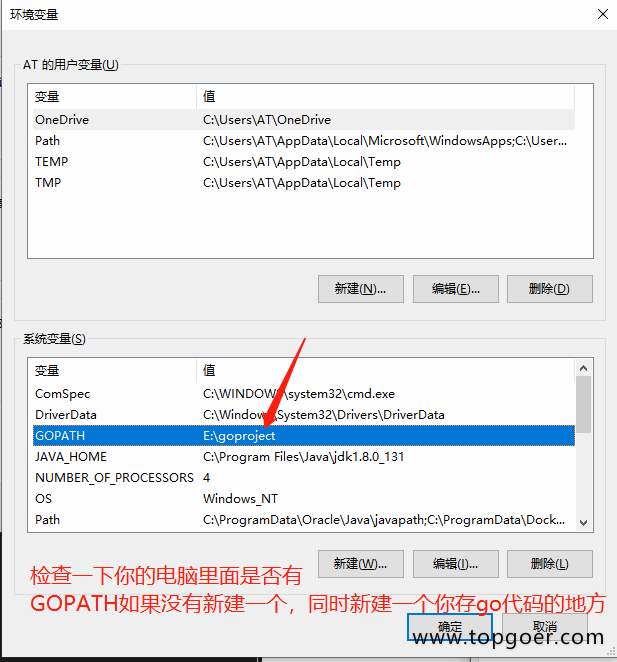
GOPATH路径最好只设置一个，所有的项目代码都放到GOPATH的src目录下。

Linux和Mac平台就参照上面配置环境变量的方式将自己的工作目录添加到环境变量中即可。 Windows平台按下面的步骤将（你的安装目录，例如：D:\go）添加到环境变量：

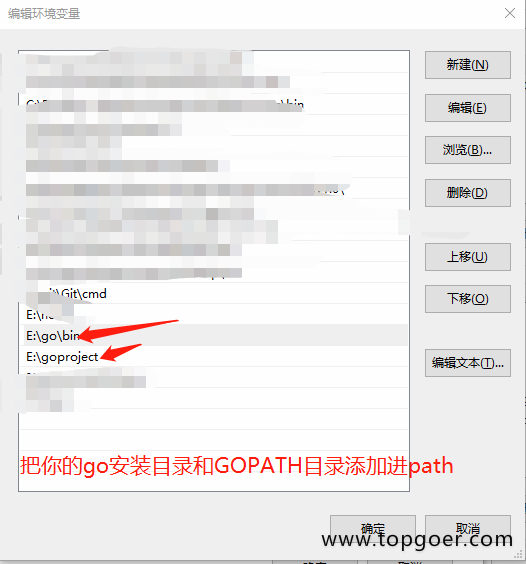
1.我的电脑->属性->高级系统设置

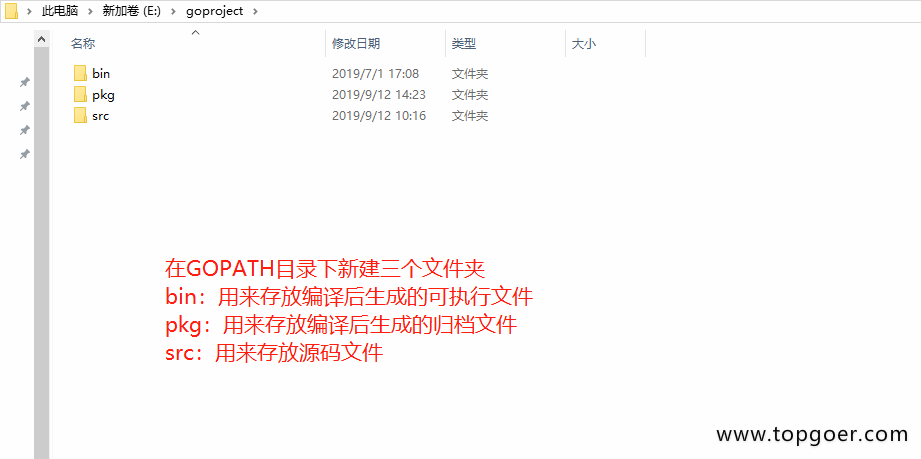


检查一下你的电脑里面是否存在GOPATH并且设置值为你要存go代码的目录



同时在path里面添加go的安装目录和GOPATH目录





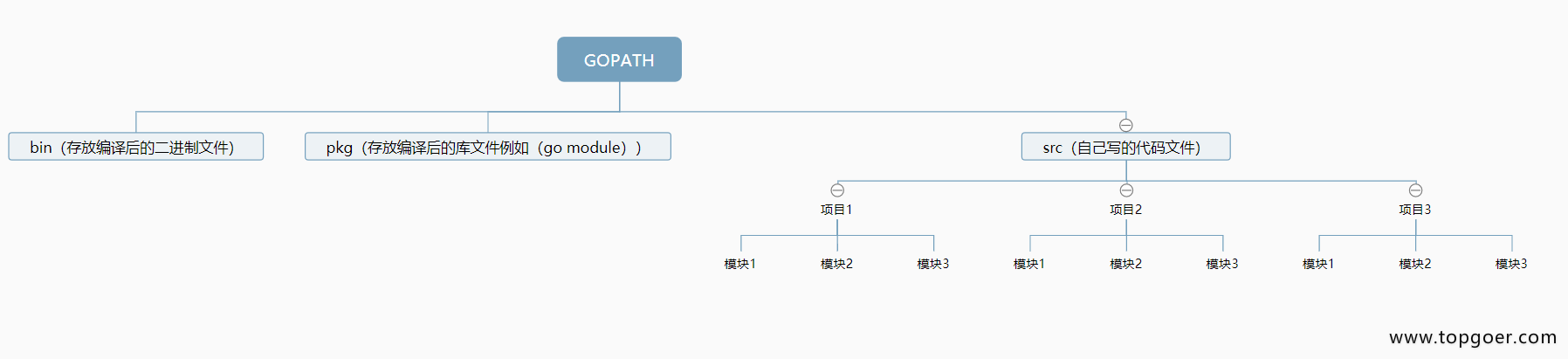
### 2.1. go的项目目录

在进行Go语言开发的时候，我们的代码总是会保存在$GOPATH/src目录下。在工程经过go build、go install或go get等指令后，会将下载的第三方包源代码文件放在$GOPATH/src目录下， 产生的二进制可执行文件放在 $GOPATH/bin目录下，生成的中间缓存文件会被保存在 $GOPATH/pkg 下。

如果我们使用版本管理工具（Version Control System，VCS。常用如Git）来管理我们的项目代码时，我们只需要添加$GOPATH/src目录的源代码即可。bin 和 pkg 目录的内容无需版本控制。

### 2.2. 适合个人开发者

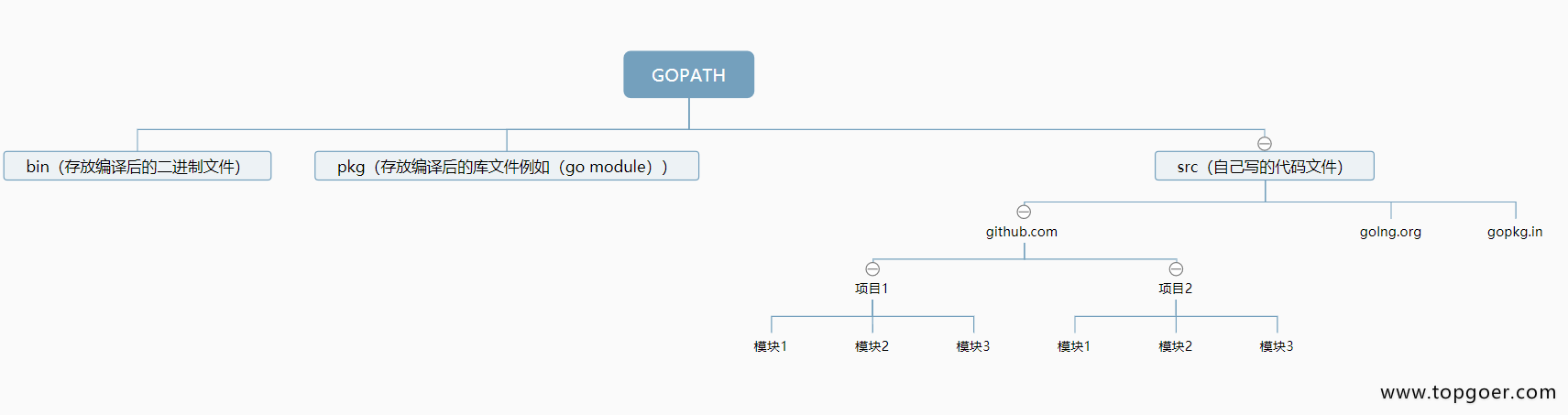
我们知道源代码都是存放在GOPATH的src目录下，那我们可以按照下图来组织我们的代码。



### 2.3. 目前流行的项目结构

Go语言中也是通过包来组织代码文件，我们可以引用别人的包也可以发布自己的包，但是为了防止不同包的项目名冲突，我们通常使用顶级域名来作为包名的前缀，这样就不担心项目名冲突的问题了。

因为不是每个个人开发者都拥有自己的顶级域名，所以目前流行的方式是使用个人的github用户名来区分不同的包。



举个例子：张三和李四都有一个名叫studygo的项目，那么这两个包的路径就会是：

import "github.com/zhangsan/studygo"

和

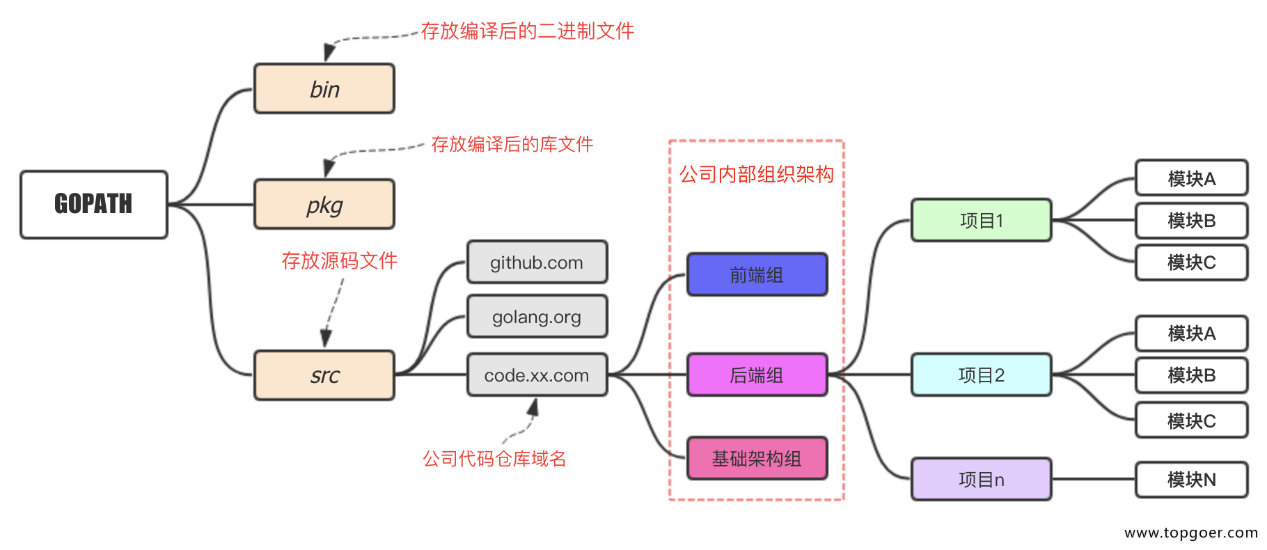
import "github.com/lisi/studygo"

以后我们从github上下载别人包的时候，如：

go get github.com/jmoiron/sqlx

那么，这个包会下载到我们本地GOPATH目录下的src/github.com/jmoiron/sqlx。

### 2.4. 适合企业开发者



## 3. 编辑器

### 3.1. Windows 安装vs code(mac版咱也没有电脑咱也不敢试)

Visual Studio Code，简称VS Code，它是目前使用人数最多的编辑器。尽管它由微软发布于2015年，与其他热门编辑器相比显得有些年轻，但它在过去几年中一直在不停的更新，它在最新的Stack Overflow调查中被选为Web开发人员中最受欢迎的文本编辑器。

VS Code不仅仅是一个基本的代码编辑器。有人说它更像是IDE而不是代码编辑器，因为它提供了许多通常只在IDE中才有的功能。主要功能包括内置调试工具，智能代码提示，集成终端以及对简易的Git操作（微软刚收购了GitHub）。作为初学者，您可以利用这些功能大大提高编程效率。

在 VS Code中找到的每个功能都完成一项出色的工作，构建了一些简单的功能集，包括语法高亮、智能补全、集成 git 和编辑器内置调试工具等，将使你开发更高效。

下载地址：<https://code.visualstudio.com/>

选择windows版本下载，vscode有新版本时候会自动更新，重启即可更新。

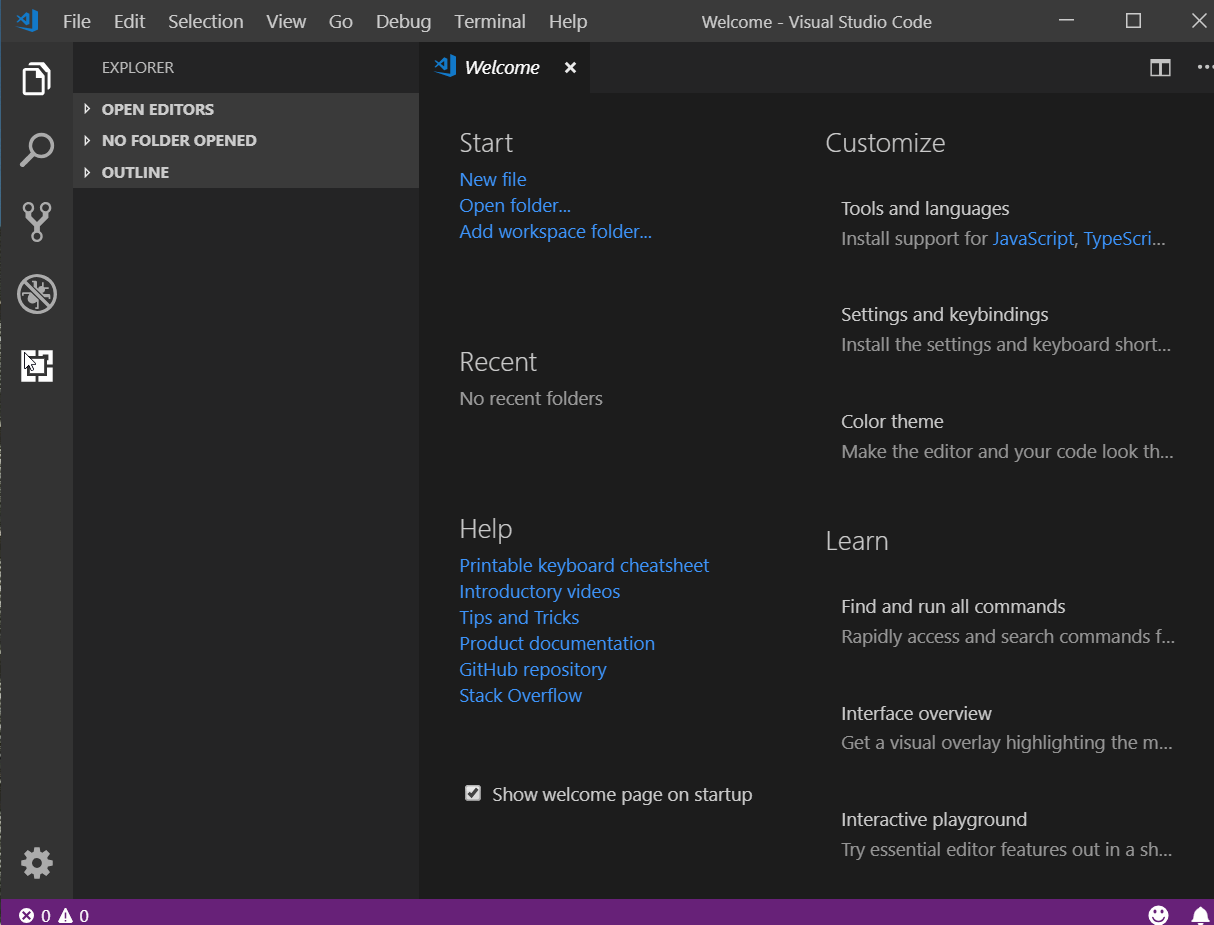
傻瓜式安装一直下一步就好了！

### 3.2. 配置

#### 3.2.1. 安装中文简体插件

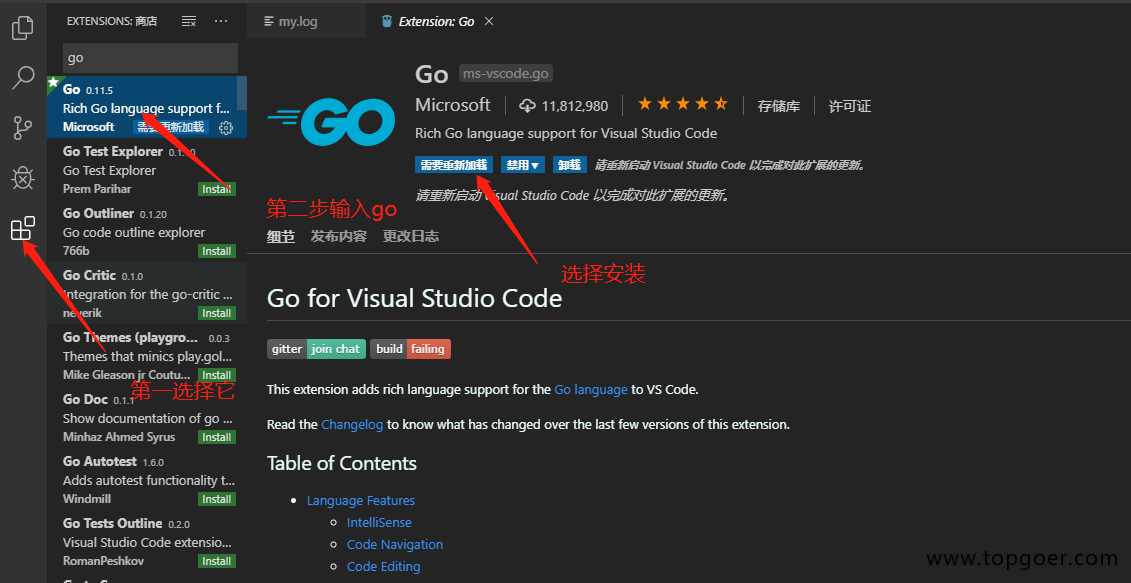
点击左侧菜单栏最后一项管理扩展，在搜索框中输入chinese，选中结果列表第一项，点击install安装。

安装完毕后右下角会提示重启VS Code，重启之后你的VS Code就显示中文啦！



#### 3.2.2. 安装go插件

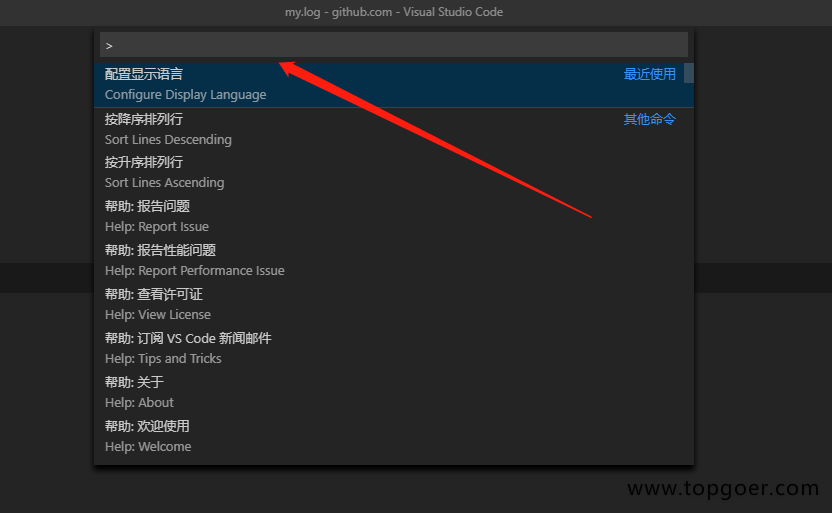
启动vscode选择插件->搜go选择Go for Visual Studio Code插件点击安装即可。如图：



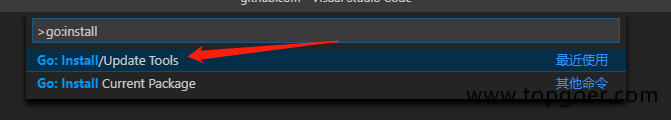
### 3.3. 安装Go语言开发工具包

在Go语言开发的时候为我们提供诸如代码提示、代码自动补全等功能。

Windows平台按下Ctrl+Shift+P，Mac平台按Command+Shift+P，这个时候VS Code界面会弹出一个输入框，如下图：



我们在这个输入框中输入>go:install，下面会自动搜索相关命令，我们选择Go:Install/Update Tools这个命令



选中并会回车执行该命令（或者使用鼠标点击该命令）



VS Code此时会下载并安装上图列出来的16个工具，但是由于国内的网络环境基本上都会出现安装失败

#### 3.3.1. 有两种方法解决这个问题：

#### 方法一：使用git下载源代码再安装

我们可以手动从github上下载工具，(执行此步骤前提需要你的电脑上已经安装了git)

第一步：现在自己的GOPATH的src目录下创建golang.org/x目录

第二步：在终端/cmd中cd到GOPATH/src/golang.org/x目录下

第三步：执行git clone https://github.com/golang/tools.git tools命令

第四步：执行git clone https://github.com/golang/lint.git命令

第五步：按下Ctrl/Command+Shift+P再次执行Go:Install/Update Tools命令，在弹出的窗口全选并点击确定，这一次的安装都会SUCCESSED了。

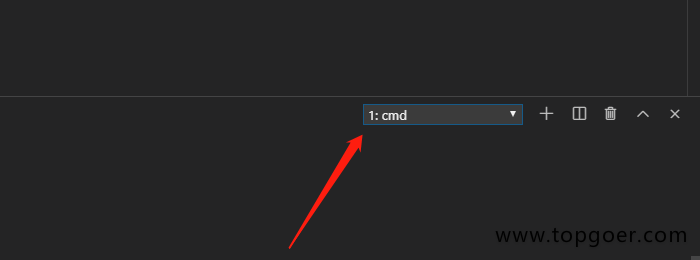
经过上面的步骤就可以安装成功了。 这个时候创建一个Go文件，就能正常使用代码提示、代码格式化等工具了。

方法二：下载已经编译好的可执行文件 如果上面的步骤执行失败了或者懒得一步一步执行，可以直接下载我已经编译好的可执行文件，拷贝到自己电脑上的 go/bin(GO的安装包目录不是代码包啊) 目录下。 https://pan.baidu.com/s/180J5j0n5Kt6wANjQyGdWDA，密码:dxti。

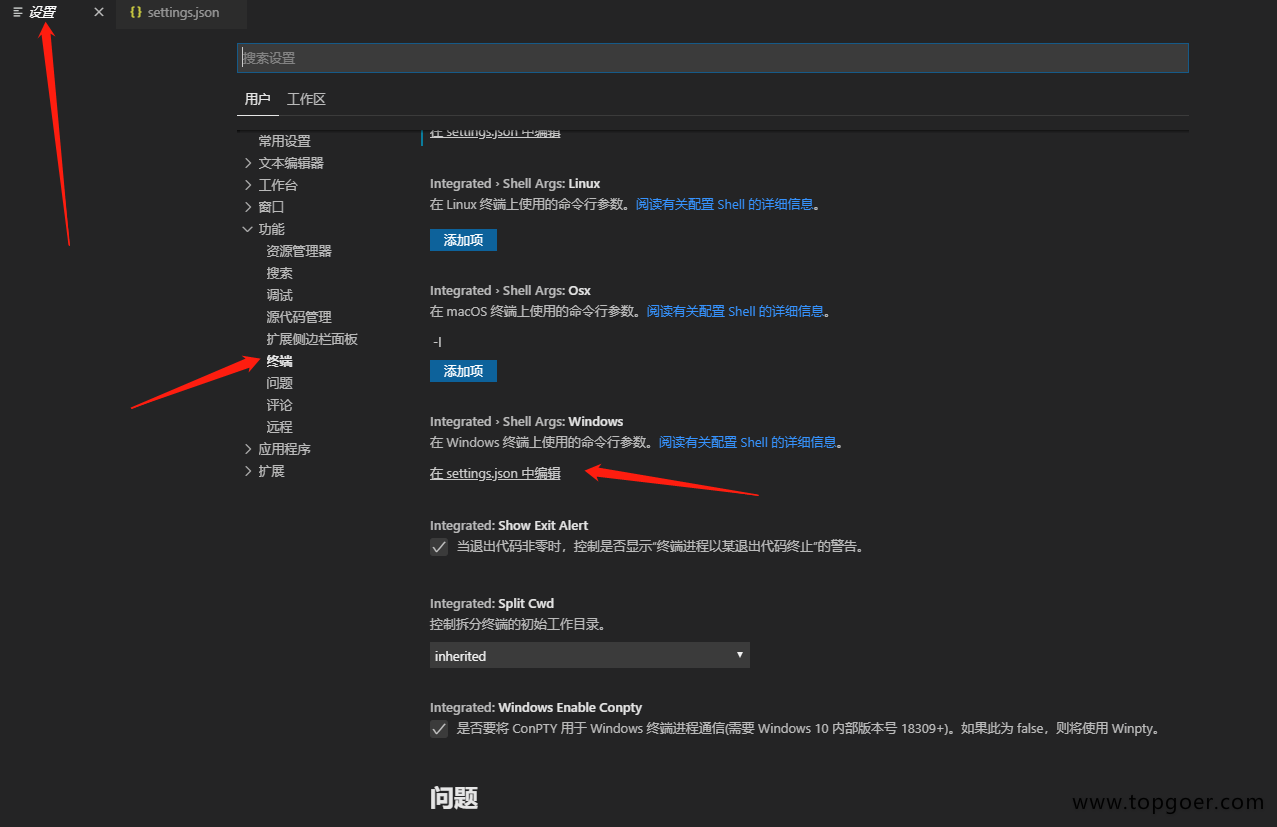
注意：特别是Mac下需要给拷贝的这些文件赋予可执行的权限。

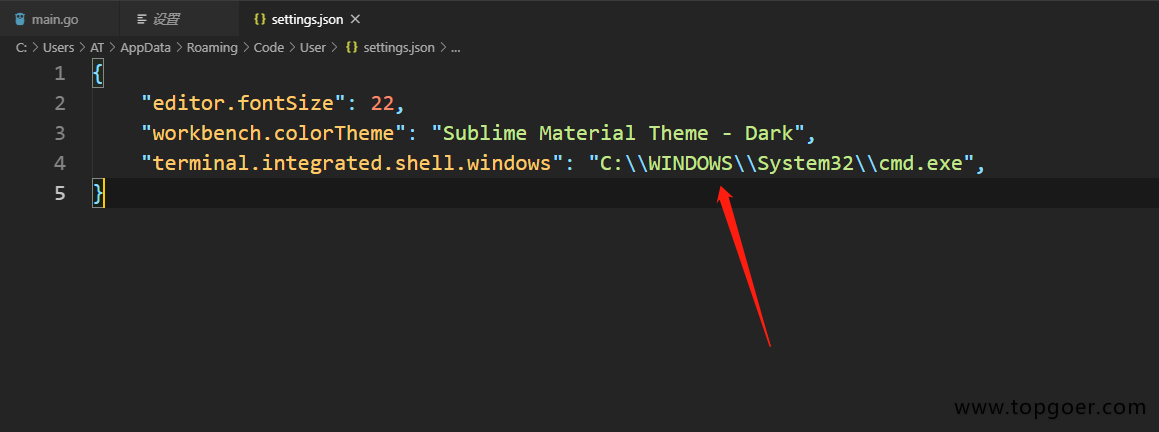
#### 3.3.2. 修改vscode终端cmd启动

在运行代码的时候需要终端运行，有的小伙伴终端默认的是powershell，有的直接默认是cmd，如果你的是powershell需要修改为cmd，如果默认的就是cmd直接放弃这块就好了



1.在文件 -> 首选项 -> 设置中打开settings页面, 搜索shell或则找到Terminal>Integrated>Shell:Windows,





添加"terminal.integrated.shell.windows": "C:\\WINDOWS\\System32\\cmd.exe", 后面的地址是你的cmd地址

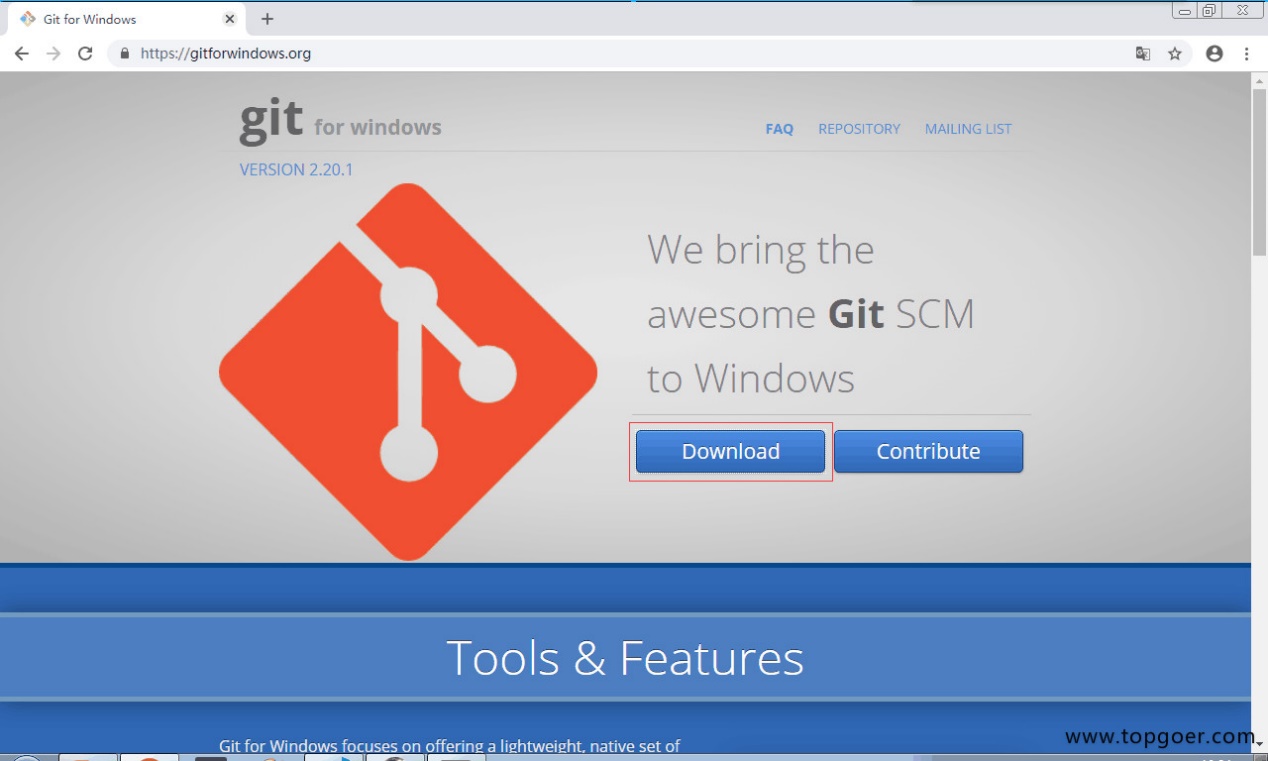
## 4. Git安装

### 4.1.1. 安装git

若你的Windows 系统已经装有git，跳过该安装。

下载地址：<http://msysgit.github.com/>

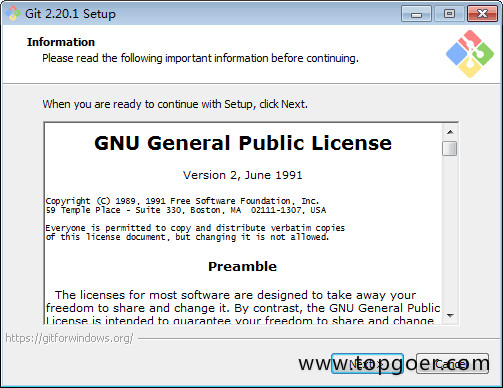
(1)进入管网，根据官网提示点击下载相应的版本。



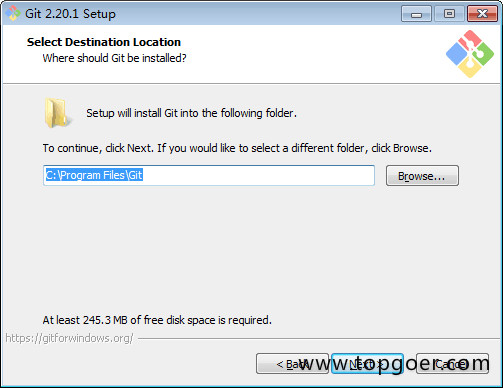
(2)git下载完成双击下载安装



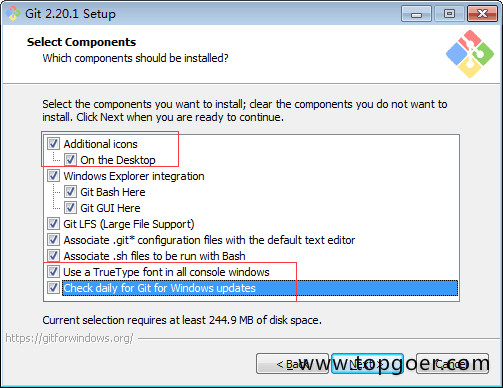
(3)选择next



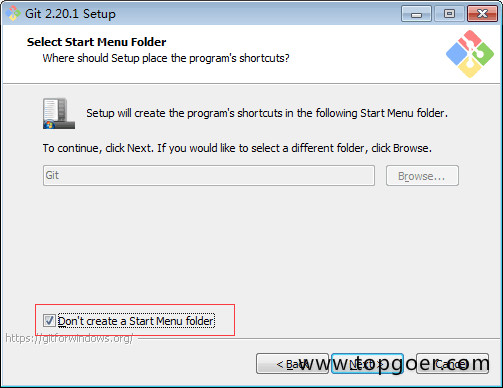
(4)文件位置存储，可根据自己盘的情况安装



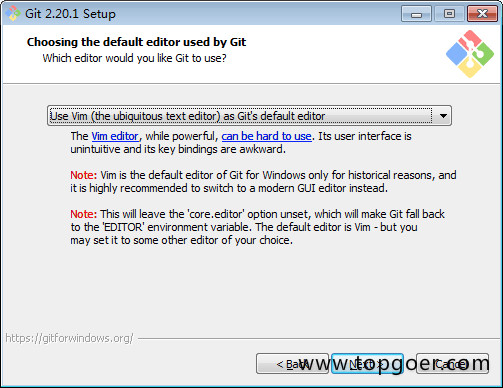
(5)安装配置文件，自己需要的都选上，下一步



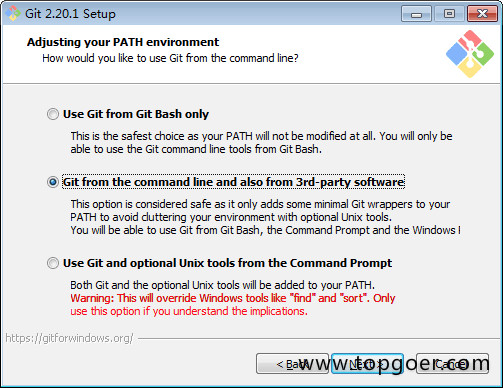
(6)不创建启动文件夹，下一步：



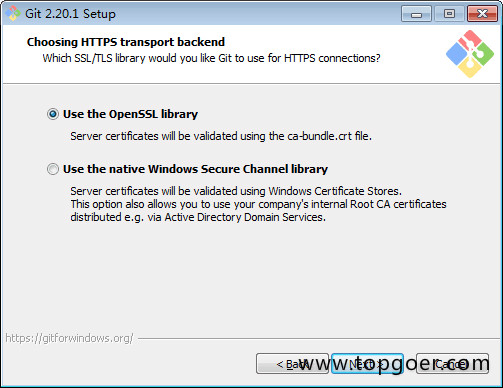
(7)选择默认的编辑器，我们直接用推荐的就行，下一步



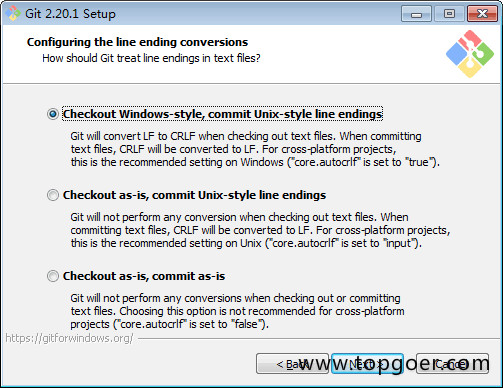
(8)勾上第二项，这样就可以在cmd中操作，下一步



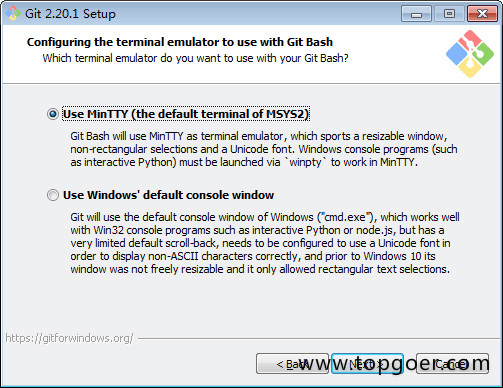
(9)使用默认设置就行，下一步：



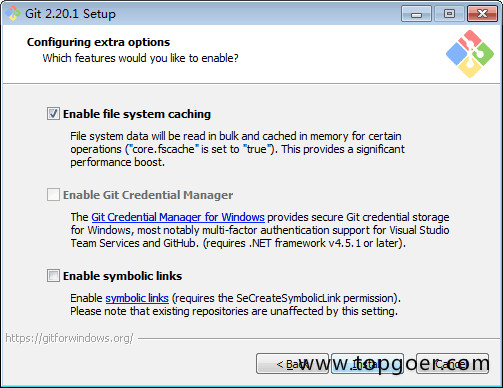
(10)配置行结束标记，保持默认“Checkout”



(11)在终端模拟器选择页面，默认即可，配置后Git



(12)最后配置Git额外选择默认即可，然后安装。



(13)安装完成



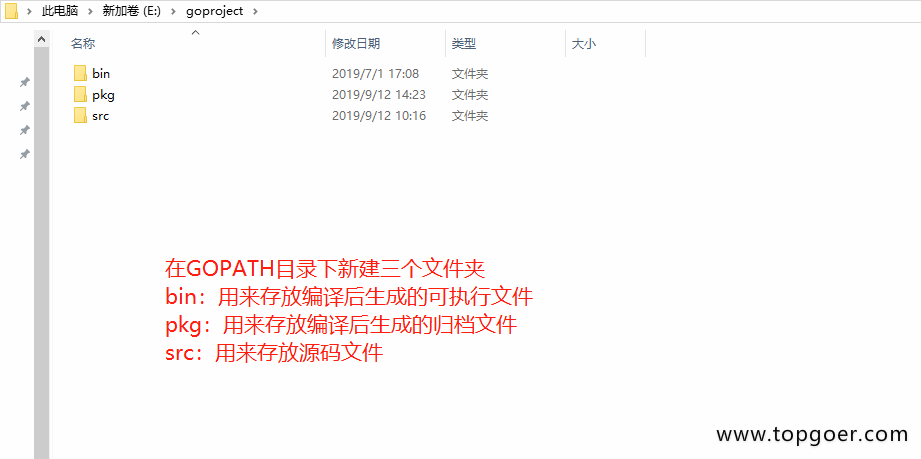
上述过程便是在windows环境下安装git的大致过程，对于不同的git版本安装过程可能会略有差异，但整体步骤均相同。

## 5. 第一个go程序

### 5.1. Hello World

学习语言的第一个程序肯定是hello word了

(1)进入前面创建的三个目录里面的src目录



(2)在src目录下创建一个hello目录，在hello目录中创建一个main.go文件：

package main // 声明 main 包，表明当前是一个可执行程序

import "fmt" // 导入内置 fmt

func main(){ // main函数，是程序执行的入口

fmt.Println("Hello World!") // 在终端打印 Hello World!

}

(3)在hello目录下执行：go build

go编译器会去 GOPATH的src目录下查找你要编译的hello项目

编译得到的可执行文件会保存在执行编译命令的当前目录下，如果是windows平台会在当前目录下找到hello.exe可执行文件。

(4)在终端直接执行该hello.exe文件：

d:\goproject\src\hello>hello.exe

Hello World!

我们还可以使用-o参数来指定编译后可执行文件的名字。

go build -o heiheihei.exe

# 三、GO基础

## 1. Go语言的主要特征

### 1.1. golang 简介

#### 1.1.1. 来历

很久以前，有一个IT公司，这公司有个传统，允许员工拥有20%自由时间来开发实验性项目。在2007的某一天，公司的几个大牛，正在用c++开发一些比较繁琐但是核心的工作，主要包括庞大的分布式集群，大牛觉得很闹心，后来c++委员会来他们公司演讲，说c++将要添加大概35种新特性。这几个大牛的其中一个人，名为：Rob Pike，听后心中一万个xxx飘过，“c++特性还不够多吗？简化c++应该更有成就感吧”。于是乎，Rob Pike和其他几个大牛讨论了一下，怎么解决这个问题，过了一会，Rob Pike说要不我们自己搞个语言吧，名字叫“go”，非常简短，容易拼写。其他几位大牛就说好啊，然后他们找了块白板，在上面写下希望能有哪些功能（详见文尾）。接下来的时间里，大牛们开心的讨论设计这门语言的特性，经过漫长的岁月，他们决定，以c语言为原型，以及借鉴其他语言的一些特性，来解放程序员，解放自己，然后在2009年，go语言诞生。

#### 1.1.2. 思想

Less can be more 大道至简,小而蕴真 让事情变得复杂很容易，让事情变得简单才难 深刻的工程文化

#### 1.1.3. 优点

自带gc。

静态编译，编译好后，扔服务器直接运行。

简单的思想，没有继承，多态，类等。

丰富的库和详细的开发文档。

语法层支持并发，和拥有同步并发的channel类型，使并发开发变得非常方便。

简洁的语法，提高开发效率，同时提高代码的阅读性和可维护性。

超级简单的交叉编译，仅需更改环境变量。

Go 语言是谷歌 2009 年首次推出并在 2012 年正式发布的一种全新的编程语言，可以在不损失应用程序性能的情况下降低代码的复杂性。谷歌首席软件工程师罗布派克(Rob Pike)说：我们之所以开发 Go，是因为过去10多年间软件开发的难度令人沮丧。Google 对 Go 寄予厚望，其设计是让软件充分发挥多核心处理器同步多工的优点，并可解决面向对象程序设计的麻烦。它具有现代的程序语言特色，如垃圾回收，帮助开发者处理琐碎但重要的内存管理问题。Go 的速度也非常快，几乎和 C 或 C++ 程序一样快，且能够快速开发应用程序。

#### 1.1.4. Go语言的主要特征：

1.自动立即回收。

2.更丰富的内置类型。

3.函数多返回值。

4.错误处理。

5.匿名函数和闭包。

6.类型和接口。

7.并发编程。

8.反射。

9.语言交互性。

#### 1.1.5. Golang文件名：

`所有的go源码都是以 ".go" 结尾。`

#### 1.1.6. Go语言命名：

1.Go的函数、变量、常量、自定义类型、包(package)的命名方式遵循以下规则：

1）首字符可以是任意的Unicode字符或者下划线

2）剩余字符可以是Unicode字符、下划线、数字

3）字符长度不限

2.Go只有25个关键字

break default func interface select

case defer go map struct

chan else goto package switch

const fallthrough if range type

continue for import return var

3.Go还有37个保留字

Constants: true false iota nil

Types: int int8 int16 int32 int64

uint uint8 uint16 uint32 uint64 uintptr

float32 float64 complex128 complex64

bool byte rune string error

Functions: make len cap new append copy close delete

complex real imag

panic recover

4.可见性：

1）声明在函数内部，是函数的本地值，类似private

2）声明在函数外部，是对当前包可见(包内所有.go文件都可见)的全局值，类似protect

3）声明在函数外部且首字母大写是所有包可见的全局值,类似public

#### 1.1.7. Go语言声明：

有四种主要声明方式：

var（声明变量）, const（声明常量）, type（声明类型） ,func（声明函数）。

Go的程序是保存在多个.go文件中，文件的第一行就是package XXX声明，用来说明该文件属于哪个包(package)，package声明下来就是import声明，再下来是类型，变量，常量，函数的声明。

#### 1.1.8. Go项目构建及编译

一个Go工程中主要包含以下三个目录：

src：源代码文件

pkg：包文件

bin：相关bin文件

1: 建立工程文件夹 goproject

2: 在工程文件夹中建立src,pkg,bin文件夹

3: 在GOPATH中添加projiect路径 例 e:/goproject

4: 如工程中有自己的包examplepackage，那在src文件夹下建立以包名命名的文件夹 例 examplepackage

5：在src文件夹下编写主程序代码代码 goproject.go

6：在examplepackage文件夹中编写 examplepackage.go 和 包测试文件 examplepackage\_test.go

7：编译调试包

go build examplepackage

go test examplepackage

go install examplepackage

这时在pkg文件夹中可以发现会有一个相应的操作系统文件夹如windows\_386z, 在这个文件夹中会有examplepackage文件夹，在该文件中有examplepackage.a文件

8：编译主程序

go build goproject.go

成功后会生成goproject.exe文件

至此一个Go工程编辑成功。

1.建立工程文件夹 go

$ pwd

/Users/\*\*\*/Desktop/go

2: 在工程文件夹中建立src,pkg,bin文件夹

$ ls

bin conf pkg src

3: 在GOPATH中添加projiect路径

$ go env

GOPATH="/Users/liupengjie/Desktop/go"

4: 那在src文件夹下建立以自己的包 example 文件夹

$ cd src/

$ mkdir example

5：在src文件夹下编写主程序代码代码 goproject.go

6：在example文件夹中编写 example.go 和 包测试文件 example\_test.go

example.go 写入如下代码：

package example

func add(a, b int) int {

return a + b

}

func sub(a, b int) int {

return a - b

}

example\_test.go 写入如下代码：

package example

import (

"testing"

)

func TestAdd(t \*testing.T) {

r := add(2, 4)

if r != 6 {

t.Fatalf("add(2, 4) error, expect:%d, actual:%d", 6, r)

}

t.Logf("test add succ")

}

7：编译调试包

$ go build example

$ go test example

ok example 0.013s

$ go install example

$ ls /Users/\*\*\*/Desktop/go/pkg/

darwin\_amd64

$ ls /Users/\*\*\*/Desktop/go/pkg/darwin\_amd64/

example.a

8：编译主程序

oproject.go 写入如下代码：

package main

import (

"fmt"

)

func main(){

fmt.Println("go project test")

}

$ go build goproject.go

$ ls

example goproject.go goproject

成功后会生成goproject文件

至此一个Go工程编辑成功。

运行该文件：

$ ./goproject

go project test

#### 1.1.9. go 编译问题

golang的编译使用命令 go build , go install;除非仅写一个main函数，否则还是准备好目录结构； GOPATH=工程根目录；其下应创建src，pkg，bin目录，bin目录中用于生成可执行文件，pkg目录中用于生成.a文件； golang中的import name，实际是到GOPATH中去寻找name.a, 使用时是该name.a的源码中生命的package 名字；这个在前面已经介绍过了。

注意点：

1.系统编译时 go install abc\_name时，系统会到GOPATH的src目录中寻找abc\_name目录，然后编译其下的go文件；

2.同一个目录中所有的go文件的package声明必须相同，所以main方法要单独放一个文件，否则在eclipse和liteide中都会报错；

编译报错如下：（假设test目录中有个main.go 和mymath.go,其中main.go声明package为main，mymath.go声明packag 为test);

$ go install test

can't load package: package test: found packages main (main.go) and test (mymath.go) in /home/wanjm/go/src/test

报错说 不能加载package test（这是命令行的参数），因为发现了两个package，分别时main.go 和 mymath.go;

3.对于main方法，只能在bin目录下运行 go build path\_tomain.go; 可以用-o参数指出输出文件名；

4.可以添加参数 go build -gcflags "-N -l" \*\*\*\*,可以更好的便于gdb；详细参见 http://golang.org/doc/gdb

5.gdb全局变量主一点。 如有全局变量 a；则应写为 p 'main.a'；注意但引号不可少；

## 2. Golang内置类型和函数

### 2.1. 内置类型

#### 2.1.1. 值类型：

bool

int(32 or 64), int8, int16, int32, int64

uint(32 or 64), uint8(byte), uint16, uint32, uint64

float32, float64

string

complex64, complex128

array -- 固定长度的数组

#### 2.1.2. 引用类型：(指针类型)

slice -- 序列数组(最常用)

map -- 映射

chan -- 管道

### 2.2. 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，例如：len、cap 和 append，或必须用于系统级的操作，例如：panic。因此，它们需要直接获得编译器的支持。

append -- 用来追加元素到数组、slice中,返回修改后的数组、slice

close -- 主要用来关闭channel

delete -- 从map中删除key对应的value

panic -- 停止常规的goroutine （panic和recover：用来做错误处理）

recover -- 允许程序定义goroutine的panic动作

real -- 返回complex的实部 （complex、real imag：用于创建和操作复数）

imag -- 返回complex的虚部

make -- 用来分配内存，返回Type本身(只能应用于slice, map, channel)

new -- 用来分配内存，主要用来分配值类型，比如int、struct。返回指向Type的指针

cap -- capacity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）

copy -- 用于复制和连接slice，返回复制的数目

len -- 来求长度，比如string、array、slice、map、channel ，返回长度

print、println -- 底层打印函数，在部署环境中建议使用 fmt 包

### 2.3. 内置接口error

type error interface { //只要实现了Error()函数，返回值为String的都实现了err接口

Error() String

}

## 3. Init函数和main函数

### 3.1. init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：

1 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

2 每个包可以拥有多个init函数

3 包的每个源文件也可以拥有多个init函数

4 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)

5 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

6 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

### 3.2. main函数

Go语言程序的默认入口函数(主函数)：func main()

函数体用｛｝一对括号包裹。

func main(){

//函数体

}

### 3.3. init函数和main函数的异同

相同点：

两个函数在定义时不能有任何的参数和返回值，且Go程序自动调用。

不同点：

init可以应用于任意包中，且可以重复定义多个。

main函数只能用于main包中，且只能定义一个。

两个函数的执行顺序：

对同一个go文件的init()调用顺序是从上到下的。

对同一个package中不同文件是按文件名字符串比较“从小到大”顺序调用各文件中的init()函数。

对于不同的package，如果不相互依赖的话，按照main包中"先import的后调用"的顺序调用其包中的init()，如果package存在依赖，则先调用最早被依赖的package中的init()，最后调用main函数。

如果init函数中使用了println()或者print()你会发现在执行过程中这两个不会按照你想象中的顺序执行。这两个函数官方只推荐在测试环境中使用，对于正式环境不要使用。

## 4. 命令

假如你已安装了golang环境，你可以在命令行执行go命令查看相关的Go语言命令：

$ go

Go is a tool for managing Go source code.

Usage:

go command [arguments]

The commands are:

build compile packages and dependencies

clean remove object files

doc show documentation for package or symbol

env print Go environment information

bug start a bug report

fix run go tool fix on packages

fmt run gofmt on package sources

generate generate Go files by processing source

get download and install packages and dependencies

install compile and install packages and dependencies

list list packages

run compile and run Go program

test test packages

tool run specified go tool

version print Go version

vet run go tool vet on packages

Use "go help [command]" for more information about a command.

Additional help topics:

c calling between Go and C

buildmode description of build modes

filetype file types

gopath GOPATH environment variable

environment environment variables

importpath import path syntax

packages description of package lists

testflag description of testing flags

testfunc description of testing functions

Use "go help [topic]" for more information about that topic.

go env用于打印Go语言的环境信息。

go run命令可以编译并运行命令源码文件。

go get可以根据要求和实际情况从互联网上下载或更新指定的代码包及其依赖包，并对它们进行编译和安装。

go build命令用于编译我们指定的源码文件或代码包以及它们的依赖包。

go install用于编译并安装指定的代码包及它们的依赖包。

go clean命令会删除掉执行其它命令时产生的一些文件和目录。

go doc命令可以打印附于Go语言程序实体上的文档。我们可以通过把程序实体的标识符作为该命令的参数来达到查看其文档的目的。

go test命令用于对Go语言编写的程序进行测试。

go list命令的作用是列出指定的代码包的信息。

go fix会把指定代码包的所有Go语言源码文件中的旧版本代码修正为新版本的代码。

go vet是一个用于检查Go语言源码中静态错误的简单工具。

go tool pprof命令来交互式的访问概要文件的内容。

## 5. 运算符

Go 语言内置的运算符有：

算术运算符

关系运算符

逻辑运算符

位运算符

赋值运算符

### 5.1.1. 算数运算符

| **运算符** | **描述** |
| --- | --- |
| + | 相加 |
| - | 相减 |
| \* | 相乘 |
| / | 相除 |
| % | 求余 |

注意： ++（自增）和--（自减）在Go语言中是单独的语句，并不是运算符。

### 5.1.2. 关系运算符

| **运算符** | **描述** |
| --- | --- |
| == | 检查两个值是否相等，如果相等返回 True 否则返回 False。 |
| != | 检查两个值是否不相等，如果不相等返回 True 否则返回 False。 |
| > | 检查左边值是否大于右边值，如果是返回 True 否则返回 False。 |
| >= | 检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。 |
| < | 检查左边值是否小于右边值，如果是返回 True 否则返回 False。 |
| <= | 检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。 |

### 5.1.3. 逻辑运算符

| **运算符** | **描述** |
| --- | --- |
| && | 逻辑 AND 运算符。 如果两边的操作数都是 True，则为 True，否则为 False。 |
| ll | 逻辑 OR 运算符。 如果两边的操作数有一个 True，则为 True，否则为 False。 |
| ! | 逻辑 NOT 运算符。 如果条件为 True，则为 False，否则为 True。 |

### 5.1.4. 位运算符

位运算符对整数在内存中的二进制位进行操作。

| **运算符** | **描述** |
| --- | --- |
| & | 参与运算的两数各对应的二进位相与。（两位均为1才为1） |
| l | 参与运算的两数各对应的二进位相或。（两位有一个为1就为1） |
| ^ | 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。（两位不一样则为1） |
| << | 左移n位就是乘以2的n次方。“a<<b”是把a的各二进位全部左移b位，高位丢弃，低位补0。 |
| >> | 右移n位就是除以2的n次方。“a>>b”是把a的各二进位全部右移b位。 |

### 5.1.5. 赋值运算符

| **运算符** | **描述** |
| --- | --- |
| = | 简单的赋值运算符，将一个表达式的值赋给一个左值 |
| += | 相加后再赋值 |
| -= | 相减后再赋值 |
| \*= | 相乘后再赋值 |
| /= | 相除后再赋值 |
| %= | 求余后再赋值 |
| <<= | 左移后赋值 |
| >>= | 右移后赋值 |
| &= | 按位与后赋值 |
| l= | 按位或后赋值 |
| ^= | 按位异或后赋值 |

## 6. 下划线

“\_”是特殊标识符，用来忽略结果。

### 6.1.1. 下划线在import中

在Golang里，import的作用是导入其他package。

　　 import 下划线（如：import hello/imp）的作用：当导入一个包时，该包下的文件里所有init()函数都会被执行，然而，有些时候我们并不需要把整个包都导入进来，仅仅是是希望它执行init()函数而已。这个时候就可以使用 import 引用该包。即使用【import \_ 包路径】只是引用该包，仅仅是为了调用init()函数，所以无法通过包名来调用包中的其他函数。 示例：

代码结构

src

|

+--- main.go

|

+--- hello

|

+--- hello.go

package main

import \_ "./hello"

func main() {

// hello.Print()

//编译报错：./main.go:6:5: undefined: hello

}

hello.go

package hello

import "fmt"

func init() {

fmt.Println("imp-init() come here.")

}

func Print() {

fmt.Println("Hello!")

}

输出结果：

imp-init() come here.

### 6.1.2. 下划线在代码中

package main

import (

"os"

)

func main() {

buf := make([]byte, 1024)

f, \_ := os.Open("/Users/\*\*\*/Desktop/text.txt")

defer f.Close()

for {

n, \_ := f.Read(buf)

if n == 0 {

break

}

os.Stdout.Write(buf[:n])

}

}

解释1：

下划线意思是忽略这个变量.

比如os.Open，返回值为\*os.File，error

普通写法是f,err := os.Open("xxxxxxx")

如果此时不需要知道返回的错误值

就可以用f, \_ := os.Open("xxxxxx")

如此则忽略了error变量

解释2：

占位符，意思是那个位置本应赋给某个值，但是咱们不需要这个值。

所以就把该值赋给下划线，意思是丢掉不要。

这样编译器可以更好的优化，任何类型的单个值都可以丢给下划线。

这种情况是占位用的，方法返回两个结果，而你只想要一个结果。

那另一个就用 "\_" 占位，而如果用变量的话，不使用，编译器是会报错的。

补充：

import "database/sql"

import \_ "github.com/go-sql-driver/mysql"

第二个import就是不直接使用mysql包，只是执行一下这个包的init函数，把mysql的驱动注册到sql包里，然后程序里就可以使用sql包来访问mysql数据库了。

## 7. 变量和常量

### 7.1. 变量

#### 7.1.1. 变量的来历

程序运行过程中的数据都是保存在内存中，我们想要在代码中操作某个数据时就需要去内存上找到这个变量，但是如果我们直接在代码中通过内存地址去操作变量的话，代码的可读性会非常差而且还容易出错，所以我们就利用变量将这个数据的内存地址保存起来，以后直接通过这个变量就能找到内存上对应的数据了。

#### 7.1.2. 变量类型

变量（Variable）的功能是存储数据。不同的变量保存的数据类型可能会不一样。经过半个多世纪的发展，编程语言已经基本形成了一套固定的类型，常见变量的数据类型有：整型、浮点型、布尔型等。

Go语言中的每一个变量都有自己的类型，并且变量必须经过声明才能开始使用。

#### 7.1.3. 变量声明

Go语言中的变量需要声明后才能使用，同一作用域内不支持重复声明。并且Go语言的变量声明后必须使用。

#### 7.1.4. 标准声明

Go语言的变量声明格式为：

var 变量名 变量类型

变量声明以关键字var开头，变量类型放在变量的后面，行尾无需分号。 举个例子：

var name string

var age int

var isOk bool

#### 7.1.5. 批量声明

每声明一个变量就需要写var关键字会比较繁琐，go语言中还支持批量变量声明：

var (

a string

b int

c bool

d float32

)

#### 7.1.6. 变量的初始化

Go语言在声明变量的时候，会自动对变量对应的内存区域进行初始化操作。每个变量会被初始化成其类型的默认值，例如： 整型和浮点型变量的默认值为0。 字符串变量的默认值为空字符串。 布尔型变量默认为false。 切片、函数、指针变量的默认为nil。

当然我们也可在声明变量的时候为其指定初始值。变量初始化的标准格式如下：

var 变量名 类型 = 表达式

举个例子：

var name string = "pprof.cn"

var sex int = 1

或者一次初始化多个变量

var name, sex = "pprof.cn", 1

##### 类型推导

有时候我们会将变量的类型省略，这个时候编译器会根据等号右边的值来推导变量的类型完成初始化。

var name = "pprof.cn"

var sex = 1

##### 短变量声明

在函数内部，可以使用更简略的 := 方式声明并初始化变量。

package main

import (

"fmt"

)

// 全局变量m

var m = 100

func main() {

n := 10

m := 200 // 此处声明局部变量m

fmt.Println(m, n)

}

##### 匿名变量

在使用多重赋值时，如果想要忽略某个值，可以使用匿名变量（anonymous variable）。 匿名变量用一个下划线\_表示，例如：

func foo() (int, string) {

return 10, "Q1mi"

}

func main() {

x, \_ := foo()

\_, y := foo()

fmt.Println("x=", x)

fmt.Println("y=", y)

}

匿名变量不占用命名空间，不会分配内存，所以匿名变量之间不存在重复声明。 (在Lua等编程语言里，匿名变量也被叫做哑元变量。)

注意事项：

函数外的每个语句都必须以关键字开始（var、const、func等）

:=不能使用在函数外。

\_多用于占位，表示忽略值。

### 7.2. 常量

相对于变量，常量是恒定不变的值，多用于定义程序运行期间不会改变的那些值。 常量的声明和变量声明非常类似，只是把var换成了const，常量在定义的时候必须赋值。

const pi = 3.1415

const e = 2.7182

声明了pi和e这两个常量之后，在整个程序运行期间它们的值都不能再发生变化了。

多个常量也可以一起声明：

const (

pi = 3.1415

e = 2.7182

)

const同时声明多个常量时，如果省略了值则表示和上面一行的值相同。 例如：

const (

n1 = 100

n2

n3

)

上面示例中，常量n1、n2、n3的值都是100。

#### 7.2.1. iota

iota是go语言的常量计数器，只能在常量的表达式中使用。 iota在const关键字出现时将被重置为0。const中每新增一行常量声明将使iota计数一次(iota可理解为const语句块中的行索引)。 使用iota能简化定义，在定义枚举时很有用。

举个例子：

const (

n1 = iota //0

n2 //1

n3 //2

n4 //3

)

#### 7.2.2. 几个常见的iota示例:

使用\_跳过某些值

const (

n1 = iota //0

n2 //1

\_

n4 //3

)

iota声明中间插队

const (

n1 = iota //0

n2 = 100 //100

n3 = iota //2

n4 //3

)

const n5 = iota //0

定义数量级 （这里的<<表示左移操作，1<<10表示将1的二进制表示向左移10位，也就是由1变成了10000000000，也就是十进制的1024。同理2<<2表示将2的二进制表示向左移2位，也就是由10变成了1000，也就是十进制的8。）

const (

\_ = iota

KB = 1 << (10 \* iota)

MB = 1 << (10 \* iota)

GB = 1 << (10 \* iota)

TB = 1 << (10 \* iota)

PB = 1 << (10 \* iota)

)

多个iota定义在一行

const (

a, b = iota + 1, iota + 2 //1,2

c, d //2,3

e, f //3,4

)

## 8. 基本类型

### 8.1. 基本类型介绍

Golang 更明确的数字类型命名，支持 Unicode，支持常用数据结构。

| **类型** | **长度(字节)** | **默认值** | **说明** |
| --- | --- | --- | --- |
| bool | 1 | false |  |
| byte | 1 | 0 | uint8 |
| rune | 4 | 0 | Unicode Code Point, int32 |
| int, uint | 4或8 | 0 | 32 或 64 位 |
| int8, uint8 | 1 | 0 | -128 ~ 127, 0 ~ 255，byte是uint8 的别名 |
| int16, uint16 | 2 | 0 | -32768 ~ 32767, 0 ~ 65535 |
| int32, uint32 | 4 | 0 | -21亿~ 21亿, 0 ~ 42亿，rune是int32 的别名 |
| int64, uint64 | 8 | 0 |  |
| float32 | 4 | 0.0 |  |
| float64 | 8 | 0.0 |  |
| complex64 | 8 |  |  |
| complex128 | 16 |  |  |
| uintptr | 4或8 |  | 以存储指针的 uint32 或 uint64 整数 |
| array |  |  | 值类型 |
| struct |  |  | 值类型 |
| string |  | "" | UTF-8 字符串 |
| slice |  | nil | 引用类型 |
| map |  | nil | 引用类型 |
| channel |  | nil | 引用类型 |
| interface |  | nil | 接口 |
| function |  | nil | 函数 |

支持八进制、 六进制，以及科学记数法。标准库 math 定义了各数字类型取值范围。

a, b, c, d := 071, 0x1F, 1e9, math.MinInt16

空指针值 nil，而非C/C++ NULL。

#### 8.1.1. 整型

整型分为以下两个大类： 按长度分为：int8、int16、int32、int64对应的无符号整型：uint8、uint16、uint32、uint64

其中，uint8就是我们熟知的byte型，int16对应C语言中的short型，int64对应C语言中的long型。

#### 8.1.2. 浮点型

Go语言支持两种浮点型数：float32和float64。这两种浮点型数据格式遵循IEEE 754标准： float32 的浮点数的最大范围约为3.4e38，可以使用常量定义：math.MaxFloat32。 float64 的浮点数的最大范围约为 1.8e308，可以使用一个常量定义：math.MaxFloat64。

#### 8.1.3. 复数

complex64和complex128

复数有实部和虚部，complex64的实部和虚部为32位，complex128的实部和虚部为64位。

#### 8.1.4. 布尔值

Go语言中以bool类型进行声明布尔型数据，布尔型数据只有true（真）和false（假）两个值。

注意：

布尔类型变量的默认值为false。

Go 语言中不允许将整型强制转换为布尔型.

布尔型无法参与数值运算，也无法与其他类型进行转换。

#### 8.1.5. 字符串

Go语言中的字符串以原生数据类型出现，使用字符串就像使用其他原生数据类型（int、bool、float32、float64 等）一样。 Go 语言里的字符串的内部实现使用UTF-8编码。 字符串的值为双引号(")中的内容，可以在Go语言的源码中直接添加非ASCII码字符，例如：

s1 := "hello"

s2 := "你好"

#### 8.1.6. 字符串转义符

Go 语言的字符串常见转义符包含回车、换行、单双引号、制表符等，如下表所示。

| **转义** | **含义** |
| --- | --- |
| \r | 回车符（返回行首） |
| \n | 换行符（直接跳到下一行的同列位置） |
| \t | 制表符 |
| \' | 单引号 |
| \" | 双引号 |
| \ | 反斜杠 |

举个例子，我们要打印一个Windows平台下的一个文件路径：

package main

import (

"fmt"

)

func main() {

fmt.Println("str := \"c:\\pprof\\main.exe\"")

}

#### 8.1.7. 多行字符串

Go语言中要定义一个多行字符串时，就必须使用反引号字符：

s1 := `第一行

第二行

第三行

`

fmt.Println(s1)

反引号间换行将被作为字符串中的换行，但是所有的转义字符均无效，文本将会原样输出。

#### 8.1.8. 字符串的常用操作

| **方法** | **介绍** |
| --- | --- |
| len(str) | 求长度 |
| +或fmt.Sprintf | 拼接字符串 |
| strings.Split | 分割 |
| strings.Contains | 判断是否包含 |
| strings.HasPrefix,strings.HasSuffix | 前缀/后缀判断 |
| strings.Index(),strings.LastIndex() | 子串出现的位置 |
| strings.Join(a[]string, sep string) | join操作 |

#### 8.1.9. byte和rune类型

组成每个字符串的元素叫做“字符”，可以通过遍历或者单个获取字符串元素获得字符。 字符用单引号（’）包裹起来，如：

var a := '中'

var b := 'x'

Go 语言的字符有以下两种：

uint8类型，或者叫 byte 型，代表了ASCII码的一个字符。

rune类型，代表一个 UTF-8字符。

当需要处理中文、日文或者其他复合字符时，则需要用到rune类型。rune类型实际是一个int32。 Go 使用了特殊的 rune 类型来处理 Unicode，让基于 Unicode的文本处理更为方便，也可以使用 byte 型进行默认字符串处理，性能和扩展性都有照顾

// 遍历字符串

func traversalString() {

s := "pprof.cn博客"

for i := 0; i < len(s); i++ { //byte

fmt.Printf("%v(%c) ", s[i], s[i])

}

fmt.Println()

for \_, r := range s { //rune

fmt.Printf("%v(%c) ", r, r)

}

fmt.Println()

}

输出：

112(p) 112(p) 114(r) 111(o) 102(f) 46(.) 99(c) 110(n) 229(å) 141() 154() 229(å) 174(®) 162(¢)

112(p) 112(p) 114(r) 111(o) 102(f) 46(.) 99(c) 110(n) 21338(博) 23458(客)

因为UTF8编码下一个中文汉字由3~4个字节组成，所以我们不能简单的按照字节去遍历一个包含中文的字符串，否则就会出现上面输出中第一行的结果。

字符串底层是一个byte数组，所以可以和[]byte类型相互转换。字符串是不能修改的 字符串是由byte字节组成，所以字符串的长度是byte字节的长度。 rune类型用来表示utf8字符，一个rune字符由一个或多个byte组成。

#### 8.1.10. 修改字符串

要修改字符串，需要先将其转换成[]rune或[]byte，完成后再转换为string。无论哪种转换，都会重新分配内存，并复制字节数组。

func changeString() {

s1 := "hello"

// 强制类型转换

byteS1 := []byte(s1)

byteS1[0] = 'H'

fmt.Println(string(byteS1))

s2 := "博客"

runeS2 := []rune(s2)

runeS2[0] = '狗'

fmt.Println(string(runeS2))

}

#### 8.1.11. 类型转换

Go语言中只有强制类型转换，没有隐式类型转换。该语法只能在两个类型之间支持相互转换的时候使用。

强制类型转换的基本语法如下：

T(表达式)

其中，T表示要转换的类型。表达式包括变量、复杂算子和函数返回值等.

比如计算直角三角形的斜边长时使用math包的Sqrt()函数，该函数接收的是float64类型的参数，而变量a和b都是int类型的，这个时候就需要将a和b强制类型转换为float64类型。

func sqrtDemo() {

var a, b = 3, 4

var c int

// math.Sqrt()接收的参数是float64类型，需要强制转换

c = int(math.Sqrt(float64(a\*a + b\*b)))

fmt.Println(c)

}

## 9. 数组Array

Golang Array和以往认知的数组有很大不同。

1. 数组：是同一种数据类型的固定长度的序列。

2. 数组定义：var a [len]int，比如：var a [5]int，数组长度必须是常量，且是类型的组成部分。一旦定义，长度不能变。

3. 长度是数组类型的一部分，因此，var a[5] int和var a[10]int是不同的类型。

4. 数组可以通过下标进行访问，下标是从0开始，最后一个元素下标是：len-1

for i := 0; i < len(a); i++ {

}

for index, v := range a {

}

5. 访问越界，如果下标在数组合法范围之外，则触发访问越界，会panic

6. 数组是值类型，赋值和传参会复制整个数组，而不是指针。因此改变副本的值，不会改变本身的值。

7.支持 "=="、"!=" 操作符，因为内存总是被初始化过的。

8.指针数组 [n]\*T，数组指针 \*[n]T。

### 9.1.1. 数组初始化：

#### 一维数组：

全局：

var arr0 [5]int = [5]int{1, 2, 3}

var arr1 = [5]int{1, 2, 3, 4, 5}

var arr2 = [...]int{1, 2, 3, 4, 5, 6}

var str = [5]string{3: "hello world", 4: "tom"}

局部：

a := [3]int{1, 2} // 未初始化元素值为 0。

b := [...]int{1, 2, 3, 4} // 通过初始化值确定数组长度。

c := [5]int{2: 100, 4: 200} // 使用索引号初始化元素。

d := [...]struct {

name string

age uint8

}{

{"user1", 10}, // 可省略元素类型。

{"user2", 20}, // 别忘了最后一行的逗号。

}

代码：

package main

import (

"fmt"

)

var arr0 [5]int = [5]int{1, 2, 3}

var arr1 = [5]int{1, 2, 3, 4, 5}

var arr2 = [...]int{1, 2, 3, 4, 5, 6}

var str = [5]string{3: "hello world", 4: "tom"}

func main() {

a := [3]int{1, 2} // 未初始化元素值为 0。

b := [...]int{1, 2, 3, 4} // 通过初始化值确定数组长度。

c := [5]int{2: 100, 4: 200} // 使用引号初始化元素。

d := [...]struct {

name string

age uint8

}{

{"user1", 10}, // 可省略元素类型。

{"user2", 20}, // 别忘了最后一行的逗号。

}

fmt.Println(arr0, arr1, arr2, str)

fmt.Println(a, b, c, d)

}

输出结果:

[1 2 3 0 0] [1 2 3 4 5] [1 2 3 4 5 6] [ hello world tom]

[1 2 0] [1 2 3 4] [0 0 100 0 200] [{user1 10} {user2 20}]

#### 多维数组

全局

var arr0 [5][3]int

var arr1 [2][3]int = [...][3]int{{1, 2, 3}, {7, 8, 9}}

局部：

a := [2][3]int{{1, 2, 3}, {4, 5, 6}}

b := [...][2]int{{1, 1}, {2, 2}, {3, 3}} // 第 2 纬度不能用 "..."。

代码：

package main

import (

"fmt"

)

var arr0 [5][3]int

var arr1 [2][3]int = [...][3]int{{1, 2, 3}, {7, 8, 9}}

func main() {

a := [2][3]int{{1, 2, 3}, {4, 5, 6}}

b := [...][2]int{{1, 1}, {2, 2}, {3, 3}} // 第 2 纬度不能用 "..."。

fmt.Println(arr0, arr1)

fmt.Println(a, b)

}

输出结果：

[[0 0 0] [0 0 0] [0 0 0] [0 0 0] [0 0 0]] [[1 2 3] [7 8 9]]

[[1 2 3] [4 5 6]] [[1 1] [2 2] [3 3]]

值拷贝行为会造成性能问题，通常会建议使用 slice，或数组指针。

package main

import (

"fmt"

)

func test(x [2]int) {

fmt.Printf("x: %p\n", &x)

x[1] = 1000

}

func main() {

a := [2]int{}

fmt.Printf("a: %p\n", &a)

test(a)

fmt.Println(a)

}

输出结果:

a: 0xc42007c010

x: 0xc42007c030

[0 0]

内置函数 len 和 cap 都返回数组长度 (元素数量)。

package main

func main() {

a := [2]int{}

println(len(a), cap(a))

}

输出结果：

2 2

#### 多维数组遍历：

package main

import (

"fmt"

)

func main() {

var f [2][3]int = [...][3]int{{1, 2, 3}, {7, 8, 9}}

for k1, v1 := range f {

for k2, v2 := range v1 {

fmt.Printf("(%d,%d)=%d ", k1, k2, v2)

}

fmt.Println()

}

}

输出结果：

(0,0)=1 (0,1)=2 (0,2)=3

(1,0)=7 (1,1)=8 (1,2)=9

### 9.1.2. 数组拷贝和传参

package main

import "fmt"

func printArr(arr \*[5]int) {

arr[0] = 10

for i, v := range arr {

fmt.Println(i, v)

}

}

func main() {

var arr1 [5]int

printArr(&arr1)

fmt.Println(arr1)

arr2 := [...]int{2, 4, 6, 8, 10}

printArr(&arr2)

fmt.Println(arr2)

}

### 9.1.3. 数组练习

#### 求数组所有元素之和

package main

import (

"fmt"

"math/rand"

"time"

)

// 求元素和

func sumArr(a [10]int) int {

var sum int = 0

for i := 0; i < len(a); i++ {

sum += a[i]

}

return sum

}

func main() {

// 若想做一个真正的随机数，要种子

// seed()种子默认是1

//rand.Seed(1)

rand.Seed(time.Now().Unix())

var b [10]int

for i := 0; i < len(b); i++ {

// 产生一个0到1000随机数

b[i] = rand.Intn(1000)

}

sum := sumArr(b)

fmt.Printf("sum=%d\n", sum)

}

#### 找出数组中和为给定值的两个元素的下标，例如数组[1,3,5,8,7]，找出两个元素之和等于8的下标分别是（0，4）和（1，2）

package main

import "fmt"

//λ 找出数组中和为给定值的两个元素的下标，例如数组[1,3,5,8,7]，

// 找出两个元素之和等于8的下标分别是（0，4）和（1，2）

// 求元素和，是给定的值

func myTest(a [5]int, target int) {

// 遍历数组

for i := 0; i < len(a); i++ {

other := target - a[i]

// 继续遍历

for j := i + 1; j < len(a); j++ {

if a[j] == other {

fmt.Printf("(%d,%d)\n", i, j)

}

}

}

}

func main() {

b := [5]int{1, 3, 5, 8, 7}

myTest(b, 8)

}

## 10. 切片Slice

需要说明，slice 并不是数组或数组指针。它通过内部指针和相关属性引用数组片段，以实现变长方案。

1. 切片：切片是数组的一个引用，因此切片是引用类型。但自身是结构体，值拷贝传递。

2. 切片的长度可以改变，因此，切片是一个可变的数组。

3. 切片遍历方式和数组一样，可以用len()求长度。表示可用元素数量，读写操作不能超过该限制。

4. cap可以求出slice最大扩张容量，不能超出数组限制。0 <= len(slice) <= len(array)，其中array是slice引用的数组。

5. 切片的定义：var 变量名 []类型，比如 var str []string var arr []int。

6. 如果 slice == nil，那么 len、cap 结果都等于 0。

### 10.1.1. 创建切片的各种方式

package main

import "fmt"

func main() {

//1.声明切片

var s1 []int

if s1 == nil {

fmt.Println("是空")

} else {

fmt.Println("不是空")

}

// 2.:=

s2 := []int{}

// 3.make()

var s3 []int = make([]int, 0)

fmt.Println(s1, s2, s3)

// 4.初始化赋值

var s4 []int = make([]int, 0, 0)

fmt.Println(s4)

s5 := []int{1, 2, 3}

fmt.Println(s5)

// 5.从数组切片

arr := [5]int{1, 2, 3, 4, 5}

var s6 []int

// 前包后不包

s6 = arr[1:4]

fmt.Println(s6)

}

### 10.1.2. 切片初始化

全局：

var arr = [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

var slice0 []int = arr[start:end]

var slice1 []int = arr[:end]

var slice2 []int = arr[start:]

var slice3 []int = arr[:]

var slice4 = arr[:len(arr)-1] //去掉切片的最后一个元素

局部：

arr2 := [...]int{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}

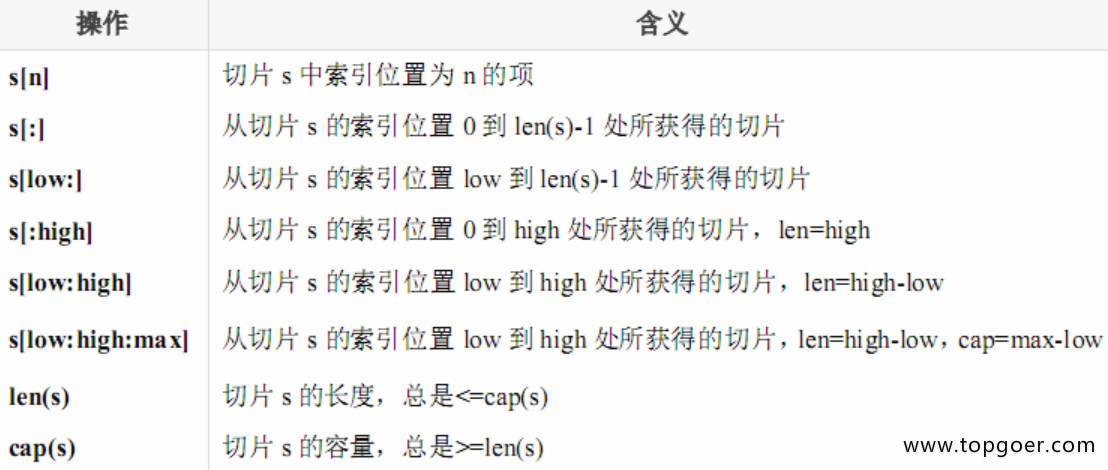
slice5 := arr[start:end]

slice6 := arr[:end]

slice7 := arr[start:]

slice8 := arr[:]

slice9 := arr[:len(arr)-1] //去掉切片的最后一个元素



代码：

package main

import (

"fmt"

)

var arr = [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

var slice0 []int = arr[2:8]

var slice1 []int = arr[0:6] //可以简写为 var slice []int = arr[:end]

var slice2 []int = arr[5:10] //可以简写为 var slice[]int = arr[start:]

var slice3 []int = arr[0:len(arr)] //var slice []int = arr[:]

var slice4 = arr[:len(arr)-1] //去掉切片的最后一个元素

func main() {

fmt.Printf("全局变量：arr %v\n", arr)

fmt.Printf("全局变量：slice0 %v\n", slice0)

fmt.Printf("全局变量：slice1 %v\n", slice1)

fmt.Printf("全局变量：slice2 %v\n", slice2)

fmt.Printf("全局变量：slice3 %v\n", slice3)

fmt.Printf("全局变量：slice4 %v\n", slice4)

fmt.Printf("-----------------------------------\n")

arr2 := [...]int{9, 8, 7, 6, 5, 4, 3, 2, 1, 0}

slice5 := arr[2:8]

slice6 := arr[0:6] //可以简写为 slice := arr[:end]

slice7 := arr[5:10] //可以简写为 slice := arr[start:]

slice8 := arr[0:len(arr)] //slice := arr[:]

slice9 := arr[:len(arr)-1] //去掉切片的最后一个元素

fmt.Printf("局部变量： arr2 %v\n", arr2)

fmt.Printf("局部变量： slice5 %v\n", slice5)

fmt.Printf("局部变量： slice6 %v\n", slice6)

fmt.Printf("局部变量： slice7 %v\n", slice7)

fmt.Printf("局部变量： slice8 %v\n", slice8)

fmt.Printf("局部变量： slice9 %v\n", slice9)

}

输出结果：

全局变量：arr [0 1 2 3 4 5 6 7 8 9]

全局变量：slice0 [2 3 4 5 6 7]

全局变量：slice1 [0 1 2 3 4 5]

全局变量：slice2 [5 6 7 8 9]

全局变量：slice3 [0 1 2 3 4 5 6 7 8 9]

全局变量：slice4 [0 1 2 3 4 5 6 7 8]

-----------------------------------

局部变量： arr2 [9 8 7 6 5 4 3 2 1 0]

局部变量： slice5 [2 3 4 5 6 7]

局部变量： slice6 [0 1 2 3 4 5]

局部变量： slice7 [5 6 7 8 9]

局部变量： slice8 [0 1 2 3 4 5 6 7 8 9]

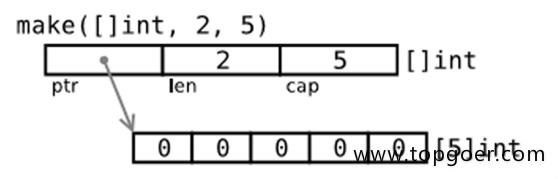
局部变量： slice9 [0 1 2 3 4 5 6 7 8]

### 10.1.3. 通过make来创建切片

var slice []type = make([]type, len)

slice := make([]type, len)

slice := make([]type, len, cap)



代码：

package main

import (

"fmt"

)

var slice0 []int = make([]int, 10)

var slice1 = make([]int, 10)

var slice2 = make([]int, 10, 10)

func main() {

fmt.Printf("make全局slice0 ：%v\n", slice0)

fmt.Printf("make全局slice1 ：%v\n", slice1)

fmt.Printf("make全局slice2 ：%v\n", slice2)

fmt.Println("--------------------------------------")

slice3 := make([]int, 10)

slice4 := make([]int, 10)

slice5 := make([]int, 10, 10)

fmt.Printf("make局部slice3 ：%v\n", slice3)

fmt.Printf("make局部slice4 ：%v\n", slice4)

fmt.Printf("make局部slice5 ：%v\n", slice5)

}

输出结果：

make全局slice0 ：[0 0 0 0 0 0 0 0 0 0]

make全局slice1 ：[0 0 0 0 0 0 0 0 0 0]

make全局slice2 ：[0 0 0 0 0 0 0 0 0 0]

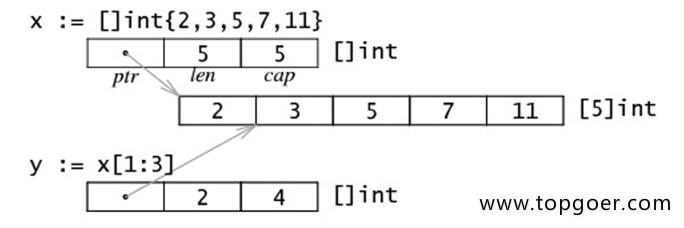
--------------------------------------

make局部slice3 ：[0 0 0 0 0 0 0 0 0 0]

make局部slice4 ：[0 0 0 0 0 0 0 0 0 0]

make局部slice5 ：[0 0 0 0 0 0 0 0 0 0]

切片的内存布局



读写操作实际目标是底层数组，只需注意索引号的差别。

package main

import (

"fmt"

)

func main() {

data := [...]int{0, 1, 2, 3, 4, 5}

s := data[2:4]

s[0] += 100

s[1] += 200

fmt.Println(s)

fmt.Println(data)

}

输出:

[102 203]

[0 1 102 203 4 5]

可直接创建 slice 对象，自动分配底层数组。

package main

import "fmt"

func main() {

s1 := []int{0, 1, 2, 3, 8: 100} // 通过初始化表达式构造，可使用索引号。

fmt.Println(s1, len(s1), cap(s1))

s2 := make([]int, 6, 8) // 使用 make 创建，指定 len 和 cap 值。

fmt.Println(s2, len(s2), cap(s2))

s3 := make([]int, 6) // 省略 cap，相当于 cap = len。

fmt.Println(s3, len(s3), cap(s3))

}

输出结果:

[0 1 2 3 0 0 0 0 100] 9 9

[0 0 0 0 0 0] 6 8

[0 0 0 0 0 0] 6 6

使用 make 动态创建slice，避免了数组必须用常量做长度的麻烦。还可用指针直接访问底层数组，退化成普通数组操作。

package main

import "fmt"

func main() {

s := []int{0, 1, 2, 3}

p := &s[2] // \*int, 获取底层数组元素指针。

\*p += 100

fmt.Println(s)

}

输出结果:

[0 1 102 3]

至于 [][]T，是指元素类型为 []T 。

package main

import (

"fmt"

)

func main() {

data := [][]int{

[]int{1, 2, 3},

[]int{100, 200},

[]int{11, 22, 33, 44},

}

fmt.Println(data)

}

输出结果：

[[1 2 3] [100 200] [11 22 33 44]]

可直接修改 struct array/slice 成员。

package main

import (

"fmt"

)

func main() {

d := [5]struct {

x int

}{}

s := d[:]

d[1].x = 10

s[2].x = 20

fmt.Println(d)

fmt.Printf("%p, %p\n", &d, &d[0])

}

输出结果:

[{0} {10} {20} {0} {0}]

0xc4200160f0, 0xc4200160f0

### 10.1.4. 用append内置函数操作切片（切片追加）

package main

import (

"fmt"

)

func main() {

var a = []int{1, 2, 3}

fmt.Printf("slice a : %v\n", a)

var b = []int{4, 5, 6}

fmt.Printf("slice b : %v\n", b)

c := append(a, b...)

fmt.Printf("slice c : %v\n", c)

d := append(c, 7)

fmt.Printf("slice d : %v\n", d)

e := append(d, 8, 9, 10)

fmt.Printf("slice e : %v\n", e)

}

输出结果：

slice a : [1 2 3]

slice b : [4 5 6]

slice c : [1 2 3 4 5 6]

slice d : [1 2 3 4 5 6 7]

slice e : [1 2 3 4 5 6 7 8 9 10]

append ：向 slice 尾部添加数据，返回新的 slice 对象。

package main

import (

"fmt"

)

func main() {

s1 := make([]int, 0, 5)

fmt.Printf("%p\n", &s1)

s2 := append(s1, 1)

fmt.Printf("%p\n", &s2)

fmt.Println(s1, s2)

}

输出结果：

0xc42000a060

0xc42000a080

[] [1]

### 10.1.5. 超出原 slice.cap 限制，就会重新分配底层数组，即便原数组并未填满。

package main

import (

"fmt"

)

func main() {

data := [...]int{0, 1, 2, 3, 4, 10: 0}

s := data[:2:3]

s = append(s, 100, 200) // 一次 append 两个值，超出 s.cap 限制。

fmt.Println(s, data) // 重新分配底层数组，与原数组无关。

fmt.Println(&s[0], &data[0]) // 比对底层数组起始指针。

}

输出结果:

[0 1 100 200] [0 1 2 3 4 0 0 0 0 0 0]

0xc4200160f0 0xc420070060

从输出结果可以看出，append 后的 s 重新分配了底层数组，并复制数据。如果只追加一个值，则不会超过 s.cap 限制，也就不会重新分配。 通常以 2 倍容量重新分配底层数组。在大批量添加数据时，建议一次性分配足够大的空间，以减少内存分配和数据复制开销。或初始化足够长的 len 属性，改用索引号进行操作。及时释放不再使用的 slice 对象，避免持有过期数组，造成 GC 无法回收。

### 10.1.6. slice中cap重新分配规律：

package main

import (

"fmt"

)

func main() {

s := make([]int, 0, 1)

c := cap(s)

for i := 0; i < 50; i++ {

s = append(s, i)

if n := cap(s); n > c {

fmt.Printf("cap: %d -> %d\n", c, n)

c = n

}

}

}

输出结果:

cap: 1 -> 2

cap: 2 -> 4

cap: 4 -> 8

cap: 8 -> 16

cap: 16 -> 32

cap: 32 -> 64

### 10.1.7. 切片拷贝

package main

import (

"fmt"

)

func main() {

s1 := []int{1, 2, 3, 4, 5}

fmt.Printf("slice s1 : %v\n", s1)

s2 := make([]int, 10)

fmt.Printf("slice s2 : %v\n", s2)

copy(s2, s1)

fmt.Printf("copied slice s1 : %v\n", s1)

fmt.Printf("copied slice s2 : %v\n", s2)

s3 := []int{1, 2, 3}

fmt.Printf("slice s3 : %v\n", s3)

s3 = append(s3, s2...)

fmt.Printf("appended slice s3 : %v\n", s3)

s3 = append(s3, 4, 5, 6)

fmt.Printf("last slice s3 : %v\n", s3)

}

输出结果：

slice s1 : [1 2 3 4 5]

slice s2 : [0 0 0 0 0 0 0 0 0 0]

copied slice s1 : [1 2 3 4 5]

copied slice s2 : [1 2 3 4 5 0 0 0 0 0]

slice s3 : [1 2 3]

appended slice s3 : [1 2 3 1 2 3 4 5 0 0 0 0 0]

last slice s3 : [1 2 3 1 2 3 4 5 0 0 0 0 0 4 5 6]

copy ：函数 copy 在两个 slice 间复制数据，复制长度以 len 小的为准。两个 slice 可指向同一底层数组，允许元素区间重叠。

package main

import (

"fmt"

)

func main() {

data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

fmt.Println("array data : ", data)

s1 := data[8:]

s2 := data[:5]

fmt.Printf("slice s1 : %v\n", s1)

fmt.Printf("slice s2 : %v\n", s2)

copy(s2, s1)

fmt.Printf("copied slice s1 : %v\n", s1)

fmt.Printf("copied slice s2 : %v\n", s2)

fmt.Println("last array data : ", data)

}

输出结果:

array data : [0 1 2 3 4 5 6 7 8 9]

slice s1 : [8 9]

slice s2 : [0 1 2 3 4]

copied slice s1 : [8 9]

copied slice s2 : [8 9 2 3 4]

last array data : [8 9 2 3 4 5 6 7 8 9]

应及时将所需数据 copy 到较小的 slice，以便释放超大号底层数组内存。

### 10.1.8. slice遍历：

package main

import (

"fmt"

)

func main() {

data := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

slice := data[:]

for index, value := range slice {

fmt.Printf("inde : %v , value : %v\n", index, value)

}

}

输出结果：

inde : 0 , value : 0

inde : 1 , value : 1

inde : 2 , value : 2

inde : 3 , value : 3

inde : 4 , value : 4

inde : 5 , value : 5

inde : 6 , value : 6

inde : 7 , value : 7

inde : 8 , value : 8

inde : 9 , value : 9

### 10.1.9. 切片resize（调整大小）

package main

import (

"fmt"

)

func main() {

var a = []int{1, 3, 4, 5}

fmt.Printf("slice a : %v , len(a) : %v\n", a, len(a))

b := a[1:2]

fmt.Printf("slice b : %v , len(b) : %v\n", b, len(b))

c := b[0:3]

fmt.Printf("slice c : %v , len(c) : %v\n", c, len(c))

}

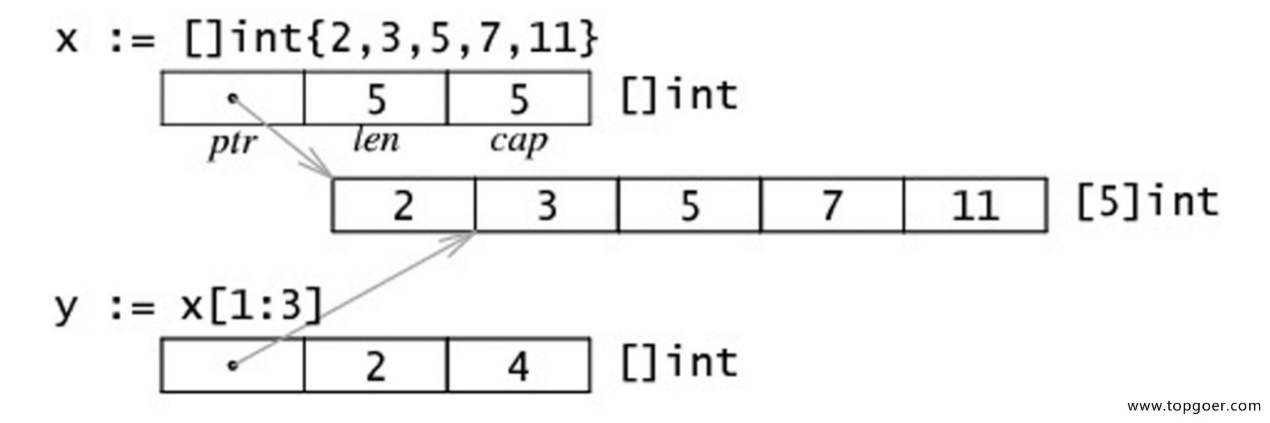
输出结果：

slice a : [1 3 4 5] , len(a) : 4

slice b : [3] , len(b) : 1

slice c : [3 4 5] , len(c) : 3

### 10.1.10. 数组和切片的内存布局



### 10.1.11. 字符串和切片（string and slice）

string底层就是一个byte的数组，因此，也可以进行切片操作。

package main

import (

"fmt"

)

func main() {

str := "hello world"

s1 := str[0:5]

fmt.Println(s1)

s2 := str[6:]

fmt.Println(s2)

}

输出结果：

hello

world

string本身是不可变的，因此要改变string中字符。需要如下操作： 英文字符串：

package main

import (

"fmt"

)

func main() {

str := "Hello world"

s := []byte(str) //中文字符需要用[]rune(str)

s[6] = 'G'

s = s[:8]

s = append(s, '!')

str = string(s)

fmt.Println(str)

}

输出结果：

Hello Go!

### 10.1.12. 含有中文字符串：

package main

import (

"fmt"

)

func main() {

str := "你好，世界！hello world！"

s := []rune(str)

s[3] = '够'

s[4] = '浪'

s[12] = 'g'

s = s[:14]

str = string(s)

fmt.Println(str)

}

输出结果：

你好，够浪！hello go

golang slice data[:6:8] 两个冒号的理解

常规slice , data[6:8]，从第6位到第8位（返回6， 7），长度len为2， 最大可扩充长度cap为4（6-9）

另一种写法： data[:6:8] 每个数字前都有个冒号， slice内容为data从0到第6位，长度len为6，最大扩充项cap设置为8

a[x:y:z] 切片内容 [x:y] 切片长度: y-x 切片容量:z-x

package main

import (

"fmt"

)

func main() {

slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

d1 := slice[6:8]

fmt.Println(d1, len(d1), cap(d1))

d2 := slice[:6:8]

fmt.Println(d2, len(d2), cap(d2))

}

数组or切片转字符串：

strings.Replace(strings.Trim(fmt.Sprint(array\_or\_slice), "[]"), " ", ",", -1)

## 11. Slice底层实现

**本章不属于基础部分但是面试经常会问到建议学学**

切片是 Go 中的一种基本的数据结构，使用这种结构可以用来管理数据集合。切片的设计想法是由动态数组概念而来，为了开发者可以更加方便的使一个数据结构可以自动增加和减少。但是切片本身并不是动态数据或者数组指针。切片常见的操作有 reslice、append、copy。与此同时，切片还具有可索引，可迭代的优秀特性。

### 11.1.1. 切片和数组

关于切片和数组怎么选择？接下来好好讨论讨论这个问题。

在 Go 中，与 C 数组变量隐式作为指针使用不同，Go 数组是值类型，赋值和函数传参操作都会复制整个数组数据。

func main() {

arrayA := [2]int{100, 200}

var arrayB [2]int

arrayB = arrayA

fmt.Printf("arrayA : %p , %v\n", &arrayA, arrayA)

fmt.Printf("arrayB : %p , %v\n", &arrayB, arrayB)

testArray(arrayA)

}

func testArray(x [2]int) {

fmt.Printf("func Array : %p , %v\n", &x, x)

}

打印结果：

arrayA : 0xc4200bebf0 , [100 200]

arrayB : 0xc4200bec00 , [100 200]

func Array : 0xc4200bec30 , [100 200]

可以看到，三个内存地址都不同，这也就验证了 Go 中数组赋值和函数传参都是值复制的。那这会导致什么问题呢？

假想每次传参都用数组，那么每次数组都要被复制一遍。如果数组大小有 100万，在64位机器上就需要花费大约 800W 字节，即 8MB 内存。这样会消耗掉大量的内存。于是乎有人想到，函数传参用数组的指针。

func main() {

arrayA := [2]int{100, 200}

testArrayPoint(&arrayA) // 1.传数组指针

arrayB := arrayA[:]

testArrayPoint(&arrayB) // 2.传切片

fmt.Printf("arrayA : %p , %v\n", &arrayA, arrayA)

}

func testArrayPoint(x \*[]int) {

fmt.Printf("func Array : %p , %v\n", x, \*x)

(\*x)[1] += 100

}

打印结果：

func Array : 0xc4200b0140 , [100 200]

func Array : 0xc4200b0180 , [100 300]

arrayA : 0xc4200b0140 , [100 400]

这也就证明了数组指针确实到达了我们想要的效果。现在就算是传入10亿的数组，也只需要再栈上分配一个8个字节的内存给指针就可以了。这样更加高效的利用内存，性能也比之前的好。

不过传指针会有一个弊端，从打印结果可以看到，第一行和第三行指针地址都是同一个，万一原数组的指针指向更改了，那么函数里面的指针指向都会跟着更改。

切片的优势也就表现出来了。用切片传数组参数，既可以达到节约内存的目的，也可以达到合理处理好共享内存的问题。打印结果第二行就是切片，切片的指针和原来数组的指针是不同的。

由此我们可以得出结论：

把第一个大数组传递给函数会消耗很多内存，采用切片的方式传参可以避免上述问题。切片是引用传递，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。

但是，依旧有反例。

package main

import "testing"

func array() [1024]int {

var x [1024]int

for i := 0; i < len(x); i++ {

x[i] = i

}

return x

}

func slice() []int {

x := make([]int, 1024)

for i := 0; i < len(x); i++ {

x[i] = i

}

return x

}

func BenchmarkArray(b \*testing.B) {

for i := 0; i < b.N; i++ {

array()

}

}

func BenchmarkSlice(b \*testing.B) {

for i := 0; i < b.N; i++ {

slice()

}

}

我们做一次性能测试，并且禁用内联和优化，来观察切片的堆上内存分配的情况。

go test -bench . -benchmem -gcflags "-N -l"

输出结果比较“令人意外”：

BenchmarkArray-4 500000 3637 ns/op 0 B/op 0 alloc s/op

BenchmarkSlice-4 300000 4055 ns/op 8192 B/op 1 alloc s/op

解释一下上述结果，在测试 Array 的时候，用的是4核，循环次数是500000，平均每次执行时间是3637 ns，每次执行堆上分配内存总量是0，分配次数也是0 。

而切片的结果就“差”一点，同样也是用的是4核，循环次数是300000，平均每次执行时间是4055 ns，但是每次执行一次，堆上分配内存总量是8192，分配次数也是1 。

这样对比看来，并非所有时候都适合用切片代替数组，因为切片底层数组可能会在堆上分配内存，而且小数组在栈上拷贝的消耗也未必比 make 消耗大。

### 11.1.2. 切片的数据结构

切片本身并不是动态数组或者数组指针。它内部实现的数据结构通过指针引用底层数组，设定相关属性将数据读写操作限定在指定的区域内。切片本身是一个只读对象，其工作机制类似数组指针的一种封装。

切片（slice）是对数组一个连续片段的引用，所以切片是一个引用类型（因此更类似于 C/C++ 中的数组类型，或者 Python 中的 list 类型）。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。切片提供了一个与指向数组的动态窗口。

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。

Slice 的数据结构定义如下:

type slice struct {

array unsafe.Pointer

len int

cap int

}

切片的结构体由3部分构成，Pointer 是指向一个数组的指针，len 代表当前切片的长度，cap 是当前切片的容量。cap 总是大于等于 len 的。

如果想从 slice 中得到一块内存地址，可以这样做：

s := make([]byte, 200)

ptr := unsafe.Pointer(&s[0])

如果反过来呢？从 Go 的内存地址中构造一个 slice。

var ptr unsafe.Pointer

var s1 = struct {

addr uintptr

len int

cap int

}{ptr, length, length}

s := \*(\*[]byte)(unsafe.Pointer(&s1))

构造一个虚拟的结构体，把 slice 的数据结构拼出来。

当然还有更加直接的方法，在 Go 的反射中就存在一个与之对应的数据结构 SliceHeader，我们可以用它来构造一个 slice

var o []byte

sliceHeader := (\*reflect.SliceHeader)((unsafe.Pointer(&o)))

sliceHeader.Cap = length

sliceHeader.Len = length

sliceHeader.Data = uintptr(ptr)

### 11.1.3. 创建切片

make 函数允许在运行期动态指定数组长度，绕开了数组类型必须使用编译期常量的限制。

创建切片有两种形式，make 创建切片，空切片。

#### make 和切片字面量

func makeslice(et \*\_type, len, cap int) slice {

// 根据切片的数据类型，获取切片的最大容量

maxElements := maxSliceCap(et.size)

// 比较切片的长度，长度值域应该在[0,maxElements]之间

if len < 0 || uintptr(len) > maxElements {

panic(errorString("makeslice: len out of range"))

}

// 比较切片的容量，容量值域应该在[len,maxElements]之间

if cap < len || uintptr(cap) > maxElements {

panic(errorString("makeslice: cap out of range"))

}

// 根据切片的容量申请内存

p := mallocgc(et.size\*uintptr(cap), et, true)

// 返回申请好内存的切片的首地址

return slice{p, len, cap}

}

还有一个 int64 的版本：

func makeslice64(et \*\_type, len64, cap64 int64) slice {

len := int(len64)

if int64(len) != len64 {

panic(errorString("makeslice: len out of range"))

}

cap := int(cap64)

if int64(cap) != cap64 {

panic(errorString("makeslice: cap out of range"))

}

return makeslice(et, len, cap)

}

实现原理和上面的是一样的，只不过多了把 int64 转换成 int 这一步罢了。

上图是用 make 函数创建的一个 len = 4， cap = 6 的切片。内存空间申请了6个 int 类型的内存大小。由于 len = 4，所以后面2个暂时访问不到，但是容量还是在的。这时候数组里面每个变量都是0 。

除了 make 函数可以创建切片以外，字面量也可以创建切片。

这里是用字面量创建的一个 len = 6，cap = 6 的切片，这时候数组里面每个元素的值都初始化完成了。需要注意的是 [ ] 里面不要写数组的容量，因为如果写了个数以后就是数组了，而不是切片了。

还有一种简单的字面量创建切片的方法。如上图。上图就 Slice A 创建出了一个 len = 3，cap = 3 的切片。从原数组的第二位元素(0是第一位)开始切，一直切到第四位为止(不包括第五位)。同理，Slice B 创建出了一个 len = 2，cap = 4 的切片。

#### nil 和空切片

nil 切片和空切片也是常用的。

var slice []int

nil 切片被用在很多标准库和内置函数中，描述一个不存在的切片的时候，就需要用到 nil 切片。比如函数在发生异常的时候，返回的切片就是 nil 切片。nil 切片的指针指向 nil。

空切片一般会用来表示一个空的集合。比如数据库查询，一条结果也没有查到，那么就可以返回一个空切片。

silce := make( []int , 0 )

slice := []int{ }

空切片和 nil 切片的区别在于，空切片指向的地址不是nil，指向的是一个内存地址，但是它没有分配任何内存空间，即底层元素包含0个元素。

最后需要说明的一点是。不管是使用 nil 切片还是空切片，对其调用内置函数 append，len 和 cap 的效果都是一样的。

### 11.1.4. 切片扩容

当一个切片的容量满了，就需要扩容了。怎么扩，策略是什么？

func growslice(et \*\_type, old slice, cap int) slice {

if raceenabled {

callerpc := getcallerpc(unsafe.Pointer(&et))

racereadrangepc(old.array, uintptr(old.len\*int(et.size)), callerpc, funcPC(growslice))

}

if msanenabled {

msanread(old.array, uintptr(old.len\*int(et.size)))

}

if et.size == 0 {

// 如果新要扩容的容量比原来的容量还要小，这代表要缩容了，那么可以直接报panic了。

if cap < old.cap {

panic(errorString("growslice: cap out of range"))

}

// 如果当前切片的大小为0，还调用了扩容方法，那么就新生成一个新的容量的切片返回。

return slice{unsafe.Pointer(&zerobase), old.len, cap}

}

// 这里就是扩容的策略

newcap := old.cap

doublecap := newcap + newcap

if cap > doublecap {

newcap = cap

} else {

if old.len < 1024 {

newcap = doublecap

} else {

for newcap < cap {

newcap += newcap / 4

}

}

}

// 计算新的切片的容量，长度。

var lenmem, newlenmem, capmem uintptr

const ptrSize = unsafe.Sizeof((\*byte)(nil))

switch et.size {

case 1:

lenmem = uintptr(old.len)

newlenmem = uintptr(cap)

capmem = roundupsize(uintptr(newcap))

newcap = int(capmem)

case ptrSize:

lenmem = uintptr(old.len) \* ptrSize

newlenmem = uintptr(cap) \* ptrSize

capmem = roundupsize(uintptr(newcap) \* ptrSize)

newcap = int(capmem / ptrSize)

default:

lenmem = uintptr(old.len) \* et.size

newlenmem = uintptr(cap) \* et.size

capmem = roundupsize(uintptr(newcap) \* et.size)

newcap = int(capmem / et.size)

}

// 判断非法的值，保证容量是在增加，并且容量不超过最大容量

if cap < old.cap || uintptr(newcap) > maxSliceCap(et.size) {

panic(errorString("growslice: cap out of range"))

}

var p unsafe.Pointer

if et.kind&kindNoPointers != 0 {

// 在老的切片后面继续扩充容量

p = mallocgc(capmem, nil, false)

// 将 lenmem 这个多个 bytes 从 old.array地址 拷贝到 p 的地址处

memmove(p, old.array, lenmem)

// 先将 P 地址加上新的容量得到新切片容量的地址，然后将新切片容量地址后面的 capmem-newlenmem 个 bytes 这块内存初始化。为之后继续 append() 操作腾出空间。

memclrNoHeapPointers(add(p, newlenmem), capmem-newlenmem)

} else {

// 重新申请新的数组给新切片

// 重新申请 capmen 这个大的内存地址，并且初始化为0值

p = mallocgc(capmem, et, true)

if !writeBarrier.enabled {

// 如果还不能打开写锁，那么只能把 lenmem 大小的 bytes 字节从 old.array 拷贝到 p 的地址处

memmove(p, old.array, lenmem)

} else {

// 循环拷贝老的切片的值

for i := uintptr(0); i < lenmem; i += et.size {

typedmemmove(et, add(p, i), add(old.array, i))

}

}

}

// 返回最终新切片，容量更新为最新扩容之后的容量

return slice{p, old.len, newcap}

}

上述就是扩容的实现。主要需要关注的有两点，一个是扩容时候的策略，还有一个就是扩容是生成全新的内存地址还是在原来的地址后追加。

#### 扩容策略

先看看扩容策略。

func main() {

slice := []int{10, 20, 30, 40}

newSlice := append(slice, 50)

fmt.Printf("Before slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("Before newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

newSlice[1] += 10

fmt.Printf("After slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("After newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

}

输出结果：

Before slice = [10 20 30 40], Pointer = 0xc4200b0140, len = 4, cap = 4

Before newSlice = [10 20 30 40 50], Pointer = 0xc4200b0180, len = 5, cap = 8

After slice = [10 20 30 40], Pointer = 0xc4200b0140, len = 4, cap = 4

After newSlice = [10 30 30 40 50], Pointer = 0xc4200b0180, len = 5, cap = 8

用图表示出上述过程。

从图上我们可以很容易的看出，新的切片和之前的切片已经不同了，因为新的切片更改了一个值，并没有影响到原来的数组，新切片指向的数组是一个全新的数组。并且 cap 容量也发生了变化。这之间究竟发生了什么呢？

Go 中切片扩容的策略是这样的：

如果切片的容量小于 1024 个元素，于是扩容的时候就翻倍增加容量。上面那个例子也验证了这一情况，总容量从原来的4个翻倍到现在的8个。

一旦元素个数超过 1024 个元素，那么增长因子就变成 1.25 ，即每次增加原来容量的四分之一。

注意：扩容扩大的容量都是针对原来的容量而言的，而不是针对原来数组的长度而言的。

#### 新数组 or 老数组 ？

再谈谈扩容之后的数组一定是新的么？这个不一定，分两种情况。

情况一：

func main() {

array := [4]int{10, 20, 30, 40}

slice := array[0:2]

newSlice := append(slice, 50)

fmt.Printf("Before slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("Before newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

newSlice[1] += 10

fmt.Printf("After slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &slice, len(slice), cap(slice))

fmt.Printf("After newSlice = %v, Pointer = %p, len = %d, cap = %d\n", newSlice, &newSlice, len(newSlice), cap(newSlice))

fmt.Printf("After array = %v\n", array)

}

打印输出：

Before slice = [10 20], Pointer = 0xc4200c0040, len = 2, cap = 4

Before newSlice = [10 20 50], Pointer = 0xc4200c0060, len = 3, cap = 4

After slice = [10 30], Pointer = 0xc4200c0040, len = 2, cap = 4

After newSlice = [10 30 50], Pointer = 0xc4200c0060, len = 3, cap = 4

After array = [10 30 50 40]

把上述过程用图表示出来，如下图。

通过打印的结果，我们可以看到，在这种情况下，扩容以后并没有新建一个新的数组，扩容前后的数组都是同一个，这也就导致了新的切片修改了一个值，也影响到了老的切片了。并且 append() 操作也改变了原来数组里面的值。一个 append() 操作影响了这么多地方，如果原数组上有多个切片，那么这些切片都会被影响！无意间就产生了莫名的 bug！

这种情况，由于原数组还有容量可以扩容，所以执行 append() 操作以后，会在原数组上直接操作，所以这种情况下，扩容以后的数组还是指向原来的数组。

这种情况也极容易出现在字面量创建切片时候，第三个参数 cap 传值的时候，如果用字面量创建切片，cap 并不等于指向数组的总容量，那么这种情况就会发生。

slice := array[1:2:3]

上面这种情况非常危险，极度容易产生 bug 。

建议用字面量创建切片的时候，cap 的值一定要保持清醒，避免共享原数组导致的 bug。

情况二：

情况二其实就是在扩容策略里面举的例子，在那个例子中之所以生成了新的切片，是因为原来数组的容量已经达到了最大值，再想扩容， Go 默认会先开一片内存区域，把原来的值拷贝过来，然后再执行 append() 操作。这种情况丝毫不影响原数组。

所以建议尽量避免情况一，尽量使用情况二，避免 bug 产生。

### 11.1.5. 切片拷贝

Slice 中拷贝方法有2个。

func slicecopy(to, fm slice, width uintptr) int {

// 如果源切片或者目标切片有一个长度为0，那么就不需要拷贝，直接 return

if fm.len == 0 || to.len == 0 {

return 0

}

// n 记录下源切片或者目标切片较短的那一个的长度

n := fm.len

if to.len < n {

n = to.len

}

// 如果入参 width = 0，也不需要拷贝了，返回较短的切片的长度

if width == 0 {

return n

}

// 如果开启了竞争检测

if raceenabled {

callerpc := getcallerpc(unsafe.Pointer(&to))

pc := funcPC(slicecopy)

racewriterangepc(to.array, uintptr(n\*int(width)), callerpc, pc)

racereadrangepc(fm.array, uintptr(n\*int(width)), callerpc, pc)

}

// 如果开启了 The memory sanitizer (msan)

if msanenabled {

msanwrite(to.array, uintptr(n\*int(width)))

msanread(fm.array, uintptr(n\*int(width)))

}

size := uintptr(n) \* width

if size == 1 {

// TODO: is this still worth it with new memmove impl?

// 如果只有一个元素，那么指针直接转换即可

\*(\*byte)(to.array) = \*(\*byte)(fm.array) // known to be a byte pointer

} else {

// 如果不止一个元素，那么就把 size 个 bytes 从 fm.array 地址开始，拷贝到 to.array 地址之后

memmove(to.array, fm.array, size)

}

return n

}

在这个方法中，slicecopy 方法会把源切片值(即 fm Slice )中的元素复制到目标切片(即 to Slice )中，并返回被复制的元素个数，copy 的两个类型必须一致。slicecopy 方法最终的复制结果取决于较短的那个切片，当较短的切片复制完成，整个复制过程就全部完成了。

举个例子，比如：

func main() {

array := []int{10, 20, 30, 40}

slice := make([]int, 6)

n := copy(slice, array)

fmt.Println(n,slice)

}

还有一个拷贝的方法，这个方法原理和 slicecopy 方法类似，不在赘述了，注释写在代码里面了。

func slicestringcopy(to []byte, fm string) int {

// 如果源切片或者目标切片有一个长度为0，那么就不需要拷贝，直接 return

if len(fm) == 0 || len(to) == 0 {

return 0

}

// n 记录下源切片或者目标切片较短的那一个的长度

n := len(fm)

if len(to) < n {

n = len(to)

}

// 如果开启了竞争检测

if raceenabled {

callerpc := getcallerpc(unsafe.Pointer(&to))

pc := funcPC(slicestringcopy)

racewriterangepc(unsafe.Pointer(&to[0]), uintptr(n), callerpc, pc)

}

// 如果开启了 The memory sanitizer (msan)

if msanenabled {

msanwrite(unsafe.Pointer(&to[0]), uintptr(n))

}

// 拷贝字符串至字节数组

memmove(unsafe.Pointer(&to[0]), stringStructOf(&fm).str, uintptr(n))

return n

}

再举个例子，比如：

func main() {

slice := make([]byte, 3)

n := copy(slice, "abcdef")

fmt.Println(n,slice)

}

输出：

3 [97,98,99]

说到拷贝，切片中有一个需要注意的问题。

func main() {

slice := []int{10, 20, 30, 40}

for index, value := range slice {

fmt.Printf("value = %d , value-addr = %x , slice-addr = %x\n", value, &value, &slice[index])

}

}

输出：

value = 10 , value-addr = c4200aedf8 , slice-addr = c4200b0320

value = 20 , value-addr = c4200aedf8 , slice-addr = c4200b0328

value = 30 , value-addr = c4200aedf8 , slice-addr = c4200b0330

value = 40 , value-addr = c4200aedf8 , slice-addr = c4200b0338

从上面结果我们可以看到，如果用 range 的方式去遍历一个切片，拿到的 Value 其实是切片里面的值拷贝。所以每次打印 Value 的地址都不变。

由于 Value 是值拷贝的，并非引用传递，所以直接改 Value 是达不到更改原切片值的目的的，需要通过 &slice[index] 获取真实的地址。

转自：<https://www.jianshu.com/p/030aba2bff41>

## 12. 指针

区别于C/C++中的指针，Go语言中的指针不能进行偏移和运算，是安全指针。

要搞明白Go语言中的指针需要先知道3个概念：指针地址、指针类型和指针取值。

### 12.1. Go语言中的指针

Go语言中的函数传参都是值拷贝，当我们想要修改某个变量的时候，我们可以创建一个指向该变量地址的指针变量。传递数据使用指针，而无须拷贝数据。类型指针不能进行偏移和运算。Go语言中的指针操作非常简单，只需要记住两个符号：&（取地址）和\*（根据地址取值）。

#### 12.1.1. 指针地址和指针类型

每个变量在运行时都拥有一个地址，这个地址代表变量在内存中的位置。Go语言中使用&字符放在变量前面对变量进行“取地址”操作。 Go语言中的值类型（int、float、bool、string、array、struct）都有对应的指针类型，如：\*int、\*int64、\*string等。

取变量指针的语法如下：

ptr := &v // v的类型为T

其中：

v:代表被取地址的变量，类型为T

ptr:用于接收地址的变量，ptr的类型就为\*T，称做T的指针类型。\*代表指针。

举个例子：

func main() {

a := 10

b := &a

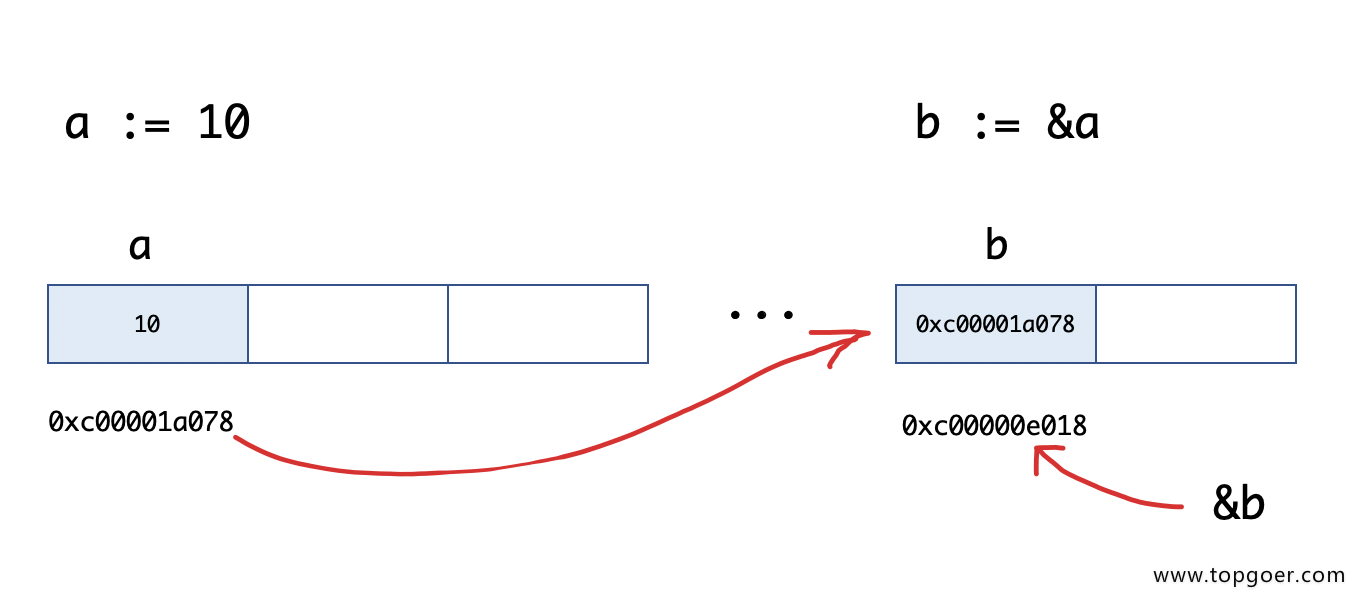
fmt.Printf("a:%d ptr:%p\n", a, &a) // a:10 ptr:0xc00001a078

fmt.Printf("b:%p type:%T\n", b, b) // b:0xc00001a078 type:\*int

fmt.Println(&b) // 0xc00000e018

}

我们来看一下b := &a的图示：



#### 12.1.2. 指针取值

在对普通变量使用&操作符取地址后会获得这个变量的指针，然后可以对指针使用\*操作，也就是指针取值，代码如下。

func main() {

//指针取值

a := 10

b := &a // 取变量a的地址，将指针保存到b中

fmt.Printf("type of b:%T\n", b)

c := \*b // 指针取值（根据指针去内存取值）

fmt.Printf("type of c:%T\n", c)

fmt.Printf("value of c:%v\n", c)

}

输出如下：

type of b:\*int

type of c:int

value of c:10

总结： 取地址操作符&和取值操作符\*是一对互补操作符，&取出地址，\*根据地址取出地址指向的值。

变量、指针地址、指针变量、取地址、取值的相互关系和特性如下：\

1.对变量进行取地址（&）操作，可以获得这个变量的指针变量。

2.指针变量的值是指针地址。

3.对指针变量进行取值（\*）操作，可以获得指针变量指向的原变量的值。

指针传值示例：

func modify1(x int) {

x = 100

}

func modify2(x \*int) {

\*x = 100

}

func main() {

a := 10

modify1(a)

fmt.Println(a) // 10

modify2(&a)

fmt.Println(a) // 100

}

#### 12.1.3. 空指针

* 当一个指针被定义后没有分配到任何变量时，它的值为 nil
* 空指针的判断

package main

import "fmt"

func main() {

var p \*string

fmt.Println(p)

fmt.Printf("p的值是%v\n", p)

if p != nil {

fmt.Println("非空")

} else {

fmt.Println("空值")

}

}

#### 12.1.4. new和make

我们先来看一个例子：

func main() {

var a \*int

\*a = 100

fmt.Println(\*a)

var b map[string]int

b["测试"] = 100

fmt.Println(b)

}

执行上面的代码会引发panic，为什么呢？ 在Go语言中对于引用类型的变量，我们在使用的时候不仅要声明它，还要为它分配内存空间，否则我们的值就没办法存储。而对于值类型的声明不需要分配内存空间，是因为它们在声明的时候已经默认分配好了内存空间。要分配内存，就引出来今天的new和make。 Go语言中new和make是内建的两个函数，主要用来分配内存

#### 12.1.5. new

new是一个内置的函数，它的函数签名如下：

func new(Type) \*Type

其中，

1.Type表示类型，new函数只接受一个参数，这个参数是一个类型

2.\*Type表示类型指针，new函数返回一个指向该类型内存地址的指针。

new函数不太常用，使用new函数得到的是一个类型的指针，并且该指针对应的值为该类型的零值。举个例子：

func main() {

a := new(int)

b := new(bool)

fmt.Printf("%T\n", a) // \*int

fmt.Printf("%T\n", b) // \*bool

fmt.Println(\*a) // 0

fmt.Println(\*b) // false

}

本节开始的示例代码中var a \*int只是声明了一个指针变量a但是没有初始化，指针作为引用类型需要初始化后才会拥有内存空间，才可以给它赋值。应该按照如下方式使用内置的new函数对a进行初始化之后就可以正常对其赋值了：

func main() {

var a \*int

a = new(int)

\*a = 10

fmt.Println(\*a)

}

#### 12.1.6. make

make也是用于内存分配的，区别于new，它只用于slice、map以及chan的内存创建，而且它返回的类型就是这三个类型本身，而不是他们的指针类型，因为这三种类型就是引用类型，所以就没有必要返回他们的指针了。make函数的函数签名如下：

func make(t Type, size ...IntegerType) Type

make函数是无可替代的，我们在使用slice、map以及channel的时候，都需要使用make进行初始化，然后才可以对它们进行操作。这个我们在上一章中都有说明，关于channel我们会在后续的章节详细说明。

本节开始的示例中var b map[string]int只是声明变量b是一个map类型的变量，需要像下面的示例代码一样使用make函数进行初始化操作之后，才能对其进行键值对赋值：

func main() {

var b map[string]int

b = make(map[string]int, 10)

b["测试"] = 100

fmt.Println(b)

}

#### 12.1.7. new与make的区别

1.二者都是用来做内存分配的。

2.make只用于slice、map以及channel的初始化，返回的还是这三个引用类型本身；

3.而new用于类型的内存分配，并且内存对应的值为类型零值，返回的是指向类型的指针。

#### 12.1.8. 指针小练习

* 程序定义一个int变量num的地址并打印
* 将num的地址赋给指针ptr，并通过ptr去修改num的值

package main

import "fmt"

func main() {

var a int

fmt.Println(&a)

var p \*int

p = &a

\*p = 20

fmt.Println(a)

}

## 13. Map

map是一种无序的基于key-value的数据结构，Go语言中的map是引用类型，必须初始化才能使用。

### 13.1.1. map定义

Go语言中 map的定义语法如下

map[KeyType]ValueType

其中，

KeyType:表示键的类型。

ValueType:表示键对应的值的类型。

map类型的变量默认初始值为nil，需要使用make()函数来分配内存。语法为：

make(map[KeyType]ValueType, [cap])

其中cap表示map的容量，该参数虽然不是必须的，但是我们应该在初始化map的时候就为其指定一个合适的容量。

### 13.1.2. map基本使用

map中的数据都是成对出现的，map的基本使用示例代码如下：

func main() {

scoreMap := make(map[string]int, 8)

scoreMap["张三"] = 90

scoreMap["小明"] = 100

fmt.Println(scoreMap)

fmt.Println(scoreMap["小明"])

fmt.Printf("type of a:%T\n", scoreMap)

}

输出：

map[小明:100 张三:90]

100

type of a:map[string]int

map也支持在声明的时候填充元素，例如：

func main() {

userInfo := map[string]string{

"username": "pprof.cn",

"password": "123456",

}

fmt.Println(userInfo) //

}

### 13.1.3. 判断某个键是否存在

Go语言中有个判断map中键是否存在的特殊写法，格式如下:

value, ok := map[key]

举个例子：

func main() {

scoreMap := make(map[string]int)

scoreMap["张三"] = 90

scoreMap["小明"] = 100

// 如果key存在ok为true,v为对应的值；不存在ok为false,v为值类型的零值

v, ok := scoreMap["张三"]

if ok {

fmt.Println(v)

} else {

fmt.Println("查无此人")

}

}

### 13.1.4. map的遍历

Go语言中使用for range遍历map。

func main() {

scoreMap := make(map[string]int)

scoreMap["张三"] = 90

scoreMap["小明"] = 100

scoreMap["王五"] = 60

for k, v := range scoreMap {

fmt.Println(k, v)

}

}

但我们只想遍历key的时候，可以按下面的写法：

func main() {

scoreMap := make(map[string]int)

scoreMap["张三"] = 90

scoreMap["小明"] = 100

scoreMap["王五"] = 60

for k := range scoreMap {

fmt.Println(k)

}

}

注意： 遍历map时的元素顺序与添加键值对的顺序无关。

### 13.1.5. 使用delete()函数删除键值对

使用delete()内建函数从map中删除一组键值对，delete()函数的格式如下：

delete(map, key)

其中，

map:表示要删除键值对的map

key:表示要删除的键值对的键

示例代码如下：

func main(){

scoreMap := make(map[string]int)

scoreMap["张三"] = 90

scoreMap["小明"] = 100

scoreMap["王五"] = 60

delete(scoreMap, "小明")//将小明:100从map中删除

for k,v := range scoreMap{

fmt.Println(k, v)

}

}

### 13.1.6. 按照指定顺序遍历map

func main() {

rand.Seed(time.Now().UnixNano()) //初始化随机数种子

var scoreMap = make(map[string]int, 200)

for i := 0; i < 100; i++ {

key := fmt.Sprintf("stu%02d", i) //生成stu开头的字符串

value := rand.Intn(100) //生成0~99的随机整数

scoreMap[key] = value

}

//取出map中的所有key存入切片keys

var keys = make([]string, 0, 200)

for key := range scoreMap {

keys = append(keys, key)

}

//对切片进行排序

sort.Strings(keys)

//按照排序后的key遍历map

for \_, key := range keys {

fmt.Println(key, scoreMap[key])

}

}

### 13.1.7. 元素为map类型的切片

下面的代码演示了切片中的元素为map类型时的操作：

func main() {

var mapSlice = make([]map[string]string, 3)

for index, value := range mapSlice {

fmt.Printf("index:%d value:%v\n", index, value)

}

fmt.Println("after init")

// 对切片中的map元素进行初始化

mapSlice[0] = make(map[string]string, 10)

mapSlice[0]["name"] = "王五"

mapSlice[0]["password"] = "123456"

mapSlice[0]["address"] = "红旗大街"

for index, value := range mapSlice {

fmt.Printf("index:%d value:%v\n", index, value)

}

}

### 13.1.8. 值为切片类型的map

下面的代码演示了map中值为切片类型的操作：

func main() {

var sliceMap = make(map[string][]string, 3)

fmt.Println(sliceMap)

fmt.Println("after init")

key := "中国"

value, ok := sliceMap[key]

if !ok {

value = make([]string, 0, 2)

}

value = append(value, "北京", "上海")

sliceMap[key] = value

fmt.Println(sliceMap)

}

## 14. Map实现原理

**本章不属于基础部分但是面试经常会问到建议学学**

### 14.1.1. 什么是Map

#### key，value存储

最通俗的话说Map是一种通过key来获取value的一个数据结构，其底层存储方式为数组，在存储时key不能重复，当key重复时，value进行覆盖，我们通过key进行hash运算（可以简单理解为把key转化为一个整形数字）然后对数组的长度取余，得到key存储在数组的哪个下标位置，最后将key和value组装为一个结构体，放入数组下标处，看下图：

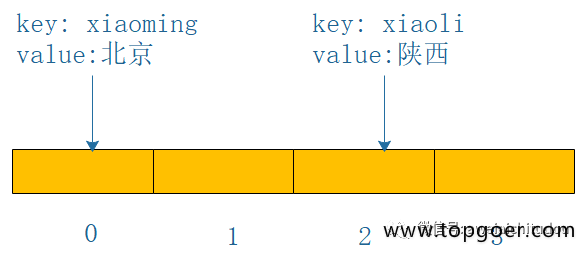
length = len(array) = 4

hashkey1 = hash(xiaoming) = 4

index1 = hashkey1% length= 0

hashkey2 = hash(xiaoli) = 6

index2 = hashkey2% length= 2

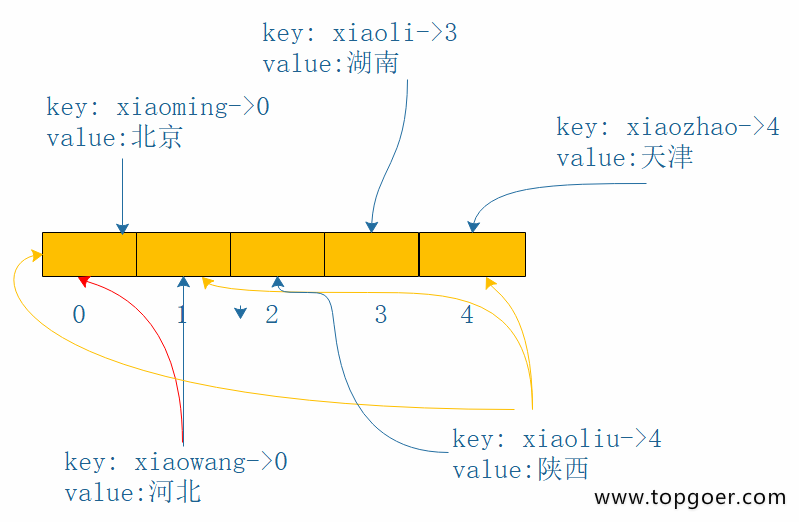


#### hash冲突

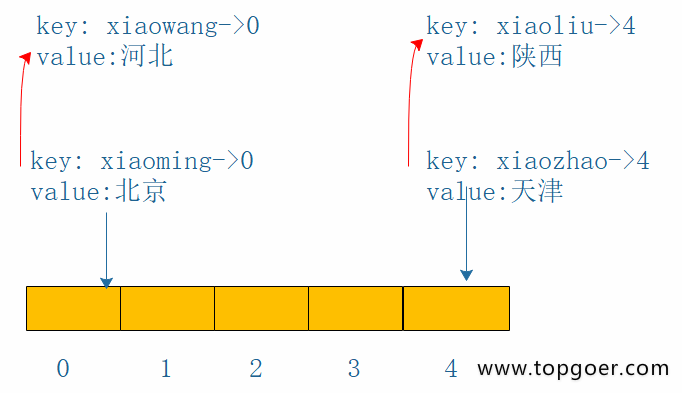
如上图所示，数组一个下标处只能存储一个元素，也就是说一个数组下标只能存储一对key，value, hashkey(xiaoming)=4占用了下标0的位置，假设我们遇到另一个key，hashkey(xiaowang)也是4，这就是hash冲突（不同的key经过hash之后得到的值一样），那么key=xiaowang的怎么存储？ hash冲突的常见解决方法

开放定址法：也就是说当我们存储一个key，value时，发现hashkey(key)的下标已经被别key占用，那我们在这个数组中空间中重新找一个没被占用的存储这个冲突的key，那么没被占用的有很多，找哪个好呢？常见的有线性探测法，线性补偿探测法，随机探测法，这里我们主要说一下线性探测法

线性探测，字面意思就是按照顺序来，从冲突的下标处开始往后探测，到达数组末尾时，从数组开始处探测，直到找到一个空位置存储这个key，当数组都找不到的情况下回扩容（事实上当数组容量快满的时候就会扩容了）；查找某一个key的时候，找到key对应的下标，比较key是否相等，如果相等直接取出来，否则按照顺寻探测直到碰到一个空位置，说明key不存在。如下图：首先存储key=xiaoming在下标0处，当存储key=xiaowang时，hash冲突了，按照线性探测，存储在下标1处，（红色的线是冲突或者下标已经被占用了） 再者key=xiaozhao存储在下标4处，当存储key=xiaoliu是，hash冲突了，按照线性探测，从头开始，存储在下标2处 （黄色的是冲突或者下标已经被占用了）



拉链法：何为拉链，简单理解为链表，当key的hash冲突时，我们在冲突位置的元素上形成一个链表，通过指针互连接，当查找时，发现key冲突，顺着链表一直往下找，直到链表的尾节点，找不到则返回空，如下图：



开放定址（线性探测）和拉链的优缺点

* 由上面可以看出拉链法比线性探测处理简单
* 线性探测查找是会被拉链法会更消耗时间
* 线性探测会更加容易导致扩容，而拉链不会
* 拉链存储了指针，所以空间上会比线性探测占用多一点
* 拉链是动态申请存储空间的，所以更适合链长不确定的

### 14.1.2. Go中Map的使用

直接用代码描述，直观，简单，易理解

//直接创建初始化一个mao

var mapInit = map[string]string {"xiaoli":"湖南", "xiaoliu":"天津"}

//声明一个map类型变量,

//map的key的类型是string，value的类型是string

var mapTemp map[string]string

//使用make函数初始化这个变量,并指定大小(也可以不指定)

mapTemp = make(map[string]string,10)

//存储key ，value

mapTemp["xiaoming"] = "北京"

mapTemp["xiaowang"]= "河北"

//根据key获取value,

//如果key存在，则ok是true，否则是flase

//v1用来接收key对应的value,当ok是false时，v1是nil

v1,ok := mapTemp["xiaoming"]

fmt.Println(ok,v1)

//当key=xiaowang存在时打印value

if v2,ok := mapTemp["xiaowang"]; ok{

fmt.Println(v2)

}

//遍历map,打印key和value

for k,v := range mapTemp{

fmt.Println(k,v)

}

//删除map中的key

delete(mapTemp,"xiaoming")

//获取map的大小

l := len(mapTemp)

fmt.Println(l)

看了上面的map创建，初始化，增删改查等操作，我们发现go的api其实挺简单易学的

### 14.1.3. Go中Map的实现原理

知其然，更得知其所以然，会使用map了，多问问为什么，go底层map到底怎么存储呢?接下来我们一探究竟。map的源码位于 src/runtime/map.go中 笔者go的版本是1.12在go中，map同样也是数组存储的的，每个数组下标处存储的是一个bucket,这个bucket的类型见下面代码，每个bucket中可以存储8个kv键值对，当每个bucket存储的kv对到达8个之后，会通过overflow指针指向一个新的bucket，从而形成一个链表,看bmap的结构，我想大家应该很纳闷，没看见kv的结构和overflow指针啊，事实上，这两个结构体并没有显示定义，是通过指针运算进行访问的。

//bucket结构体定义 b就是bucket

type bmap{

// tophash generally contains the top byte of the hash value

// for each key in this bucket. If tophash[0] < minTopHash,

// tophash[0] is a bucket evacuation state instead.

//翻译：top hash通常包含该bucket中每个键的hash值的高八位。

如果tophash[0]小于mintophash，则tophash[0]为桶疏散状态 //bucketCnt 的初始值是8

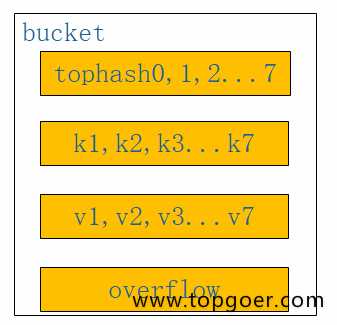
tophash [bucketCnt]uint8

// Followed by bucketCnt keys and then bucketCnt values.

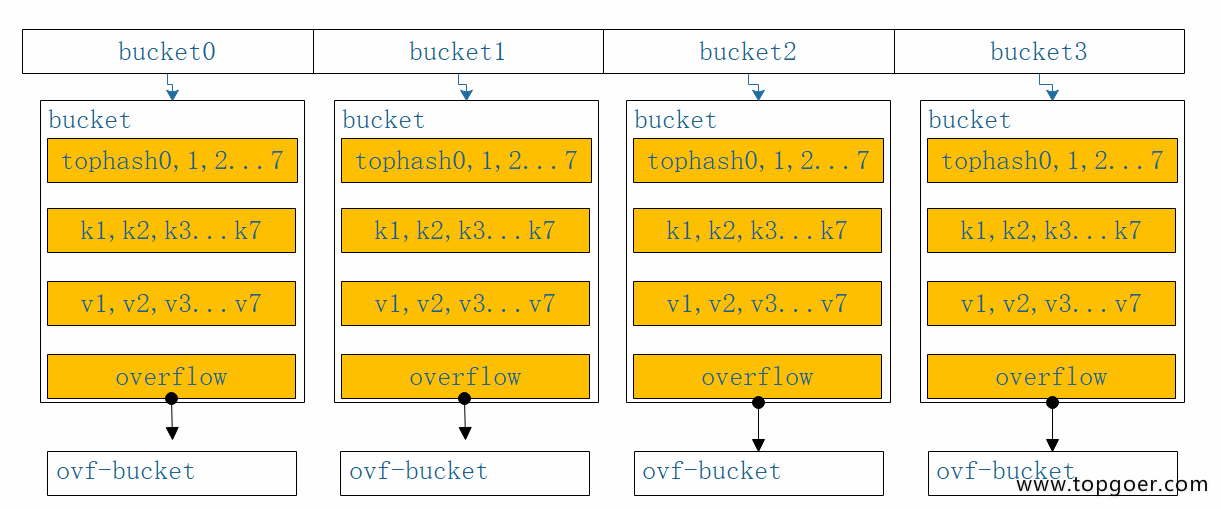
// NOTE: packing all the keys together and then all the values together makes the // code a bit more complicated than alternating key/value/key/value/... but it allows // us to eliminate padding which would be needed for, e.g., map[int64]int8.// Followed by an overflow pointer. //翻译：接下来是bucketcnt键，然后是bucketcnt值。

注意：将所有键打包在一起，然后将所有值打包在一起， 使得代码比交替键/值/键/值/更复杂。但它允许//我们消除可能需要的填充， 例如map[int64]int8./后面跟一个溢出指针}

看上面代码以及注释，我们能得到bucket中存储的kv是这样的，tophash用来快速查找key值是否在该bucket中，而不同每次都通过真值进行比较；还有kv的存放，为什么不是k1v1，k2v2..... 而是k1k2...v1v2...，我们看上面的注释说的 map[int64]int8,key是int64（8个字节），value是int8（一个字节），kv的长度不同，如果按照kv格式存放，则考虑内存对齐v也会占用int64，而按照后者存储时，8个v刚好占用一个int64,从这个就可以看出go的map设计之巧妙。



最后我们分析一下go的整体内存结构，阅读一下map存储的源码，如下图所示，当往map中存储一个kv对时，通过k获取hash值，hash值的低八位和bucket数组长度取余，定位到在数组中的那个下标，hash值的高八位存储在bucket中的tophash中，用来快速判断key是否存在，key和value的具体值则通过指针运算存储，当一个bucket满时，通过overfolw指针链接到下一个bucket。



go的map存储源码如下，省略了一些无关紧要的代码

func mapassign(t \*maptype, h \*hmap, key unsafe.Pointer) unsafe.Pointer {

//获取hash算法

alg := t.key.alg

//计算hash值

hash := alg.hash(key, uintptr(h.hash0))

//如果bucket数组一开始为空，则初始化

if h.buckets == nil {

h.buckets = newobject(t.bucket) // newarray(t.bucket, 1)

}

again:

// 定位存储在哪一个bucket中

bucket := hash & bucketMask(h.B)

//得到bucket的结构体

b := (\*bmap)(unsafe.Pointer(uintptr(h.buckets) +bucket\*uintptr(t.bucketsize)))

//获取高八位hash值

top := tophash(hash)

var inserti \*uint8

var insertk unsafe.Pointer

var val unsafe.Pointer

bucketloop:

//死循环

for {

//循环bucket中的tophash数组

for i := uintptr(0); i < bucketCnt; i++ {

//如果hash不相等

if b.tophash[i] != top {

//判断是否为空，为空则插入

if isEmpty(b.tophash[i]) && inserti == nil {

inserti = &b.tophash[i]

insertk = add(unsafe.Pointer(b), dataOffset+i\*uintptr(t.keysize))

val = add( unsafe.Pointer(b),

dataOffset+bucketCnt\*uintptr(t.keysize)+i\*uintptr(t.valuesize) )

}

//插入成功，终止最外层循环

if b.tophash[i] == emptyRest {

break bucketloop

}

continue

}

//到这里说明高八位hash一样，获取已存在的key

k := add(unsafe.Pointer(b), dataOffset+i\*uintptr(t.keysize))

if t.indirectkey() {

k = \*((\*unsafe.Pointer)(k))

}

//判断两个key是否相等，不相等就循环下一个

if !alg.equal(key, k) {

continue

}

// 如果相等则更新

if t.needkeyupdate() {

typedmemmove(t.key, k, key)

}

//获取已存在的value

val = add(unsafe.Pointer(b), dataOffset+bucketCnt\*uintptr(t.keysize)+i\*uintptr(t.valuesize))

goto done

}

//如果上一个bucket没能插入，则通过overflow获取链表上的下一个bucket

ovf := b.overflow(t)

if ovf == nil {

break

}

b = ovf

}

if inserti == nil {

// all current buckets are full, allocate a new one.

newb := h.newoverflow(t, b)

inserti = &newb.tophash[0]

insertk = add(unsafe.Pointer(newb), dataOffset)

val = add(insertk, bucketCnt\*uintptr(t.keysize))

}

// store new key/value at insert position

if t.indirectkey() {

kmem := newobject(t.key)

\*(\*unsafe.Pointer)(insertk) = kmem

insertk = kmem

}

if t.indirectvalue() {

vmem := newobject(t.elem)

\*(\*unsafe.Pointer)(val) = vmem

}

typedmemmove(t.key, insertk, key)

//将高八位hash值存储

\*inserti = top

h.count++

return val

}

转自：<https://cloud.tencent.com/developer/article/1468799>

## 15. 结构体

Go语言中没有“类”的概念，也不支持“类”的继承等面向对象的概念。Go语言中通过结构体的内嵌再配合接口比面向对象具有更高的扩展性和灵活性。

### 15.1. 类型别名和自定义类型

#### 15.1.1. 自定义类型

在Go语言中有一些基本的数据类型，如string、整型、浮点型、布尔等数据类型，Go语言中可以使用type关键字来定义自定义类型。

自定义类型是定义了一个全新的类型。我们可以基于内置的基本类型定义，也可以通过struct定义。例如：

//将MyInt定义为int类型

type MyInt int

通过Type关键字的定义，MyInt就是一种新的类型，它具有int的特性。

#### 15.1.2. 类型别名

类型别名是Go1.9版本添加的新功能。

类型别名规定：TypeAlias只是Type的别名，本质上TypeAlias与Type是同一个类型。就像一个孩子小时候有小名、乳名，上学后用学名，英语老师又会给他起英文名，但这些名字都指的是他本人。

type TypeAlias = Type

我们之前见过的rune和byte就是类型别名，他们的定义如下：

type byte = uint8

type rune = int32

#### 15.1.3. 类型定义和类型别名的区别

类型别名与类型定义表面上看只有一个等号的差异，我们通过下面的这段代码来理解它们之间的区别。

//类型定义

type NewInt int

//类型别名

type MyInt = int

func main() {

var a NewInt

var b MyInt

fmt.Printf("type of a:%T\n", a) //type of a:main.NewInt

fmt.Printf("type of b:%T\n", b) //type of b:int

}

结果显示a的类型是main.NewInt，表示main包下定义的NewInt类型。b的类型是int。MyInt类型只会在代码中存在，编译完成时并不会有MyInt类型。

### 15.2. 结构体

Go语言中的基础数据类型可以表示一些事物的基本属性，但是当我们想表达一个事物的全部或部分属性时，这时候再用单一的基本数据类型明显就无法满足需求了，Go语言提供了一种自定义数据类型，可以封装多个基本数据类型，这种数据类型叫结构体，英文名称struct。 也就是我们可以通过struct来定义自己的类型了。

Go语言中通过struct来实现面向对象。

#### 15.2.1. 结构体的定义

使用type和struct关键字来定义结构体，具体代码格式如下：

type 类型名 struct {

字段名 字段类型

字段名 字段类型

…

}

其中：

1.类型名：标识自定义结构体的名称，在同一个包内不能重复。

2.字段名：表示结构体字段名。结构体中的字段名必须唯一。

3.字段类型：表示结构体字段的具体类型。

举个例子，我们定义一个Person（人）结构体，代码如下：

type person struct {

name string

city string

age int8

}

同样类型的字段也可以写在一行，

type person1 struct {

name, city string

age int8

}

这样我们就拥有了一个person的自定义类型，它有name、city、age三个字段，分别表示姓名、城市和年龄。这样我们使用这个person结构体就能够很方便的在程序中表示和存储人信息了。

语言内置的基础数据类型是用来描述一个值的，而结构体是用来描述一组值的。比如一个人有名字、年龄和居住城市等，本质上是一种聚合型的数据类型

#### 15.2.2. 结构体实例化

只有当结构体实例化时，才会真正地分配内存。也就是必须实例化后才能使用结构体的字段。

结构体本身也是一种类型，我们可以像声明内置类型一样使用var关键字声明结构体类型。

var 结构体实例 结构体类型

#### 15.2.3. 基本实例化

type person struct {

name string

city string

age int8

}

func main() {

var p1 person

p1.name = "pprof.cn"

p1.city = "北京"

p1.age = 18

fmt.Printf("p1=%v\n", p1) //p1={pprof.cn 北京 18}

fmt.Printf("p1=%#v\n", p1) //p1=main.person{name:"pprof.cn", city:"北京", age:18}

}

我们通过.来访问结构体的字段（成员变量）,例如p1.name和p1.age等。

### 15.3. 匿名结构体

在定义一些临时数据结构等场景下还可以使用匿名结构体。

package main

import (

"fmt"

)

func main() {

var user struct{Name string; Age int}

user.Name = "pprof.cn"

user.Age = 18

fmt.Printf("%#v\n", user)

}

#### 15.3.1. 创建指针类型结构体

我们还可以通过使用new关键字对结构体进行实例化，得到的是结构体的地址。 格式如下：

var p2 = new(person)

fmt.Printf("%T\n", p2) //\*main.person

fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"", city:"", age:0}

从打印的结果中我们可以看出p2是一个结构体指针。

需要注意的是在Go语言中支持对结构体指针直接使用.来访问结构体的成员。

var p2 = new(person)

p2.name = "测试"

p2.age = 18

p2.city = "北京"

fmt.Printf("p2=%#v\n", p2) //p2=&main.person{name:"测试", city:"北京", age:18}

#### 15.3.2. 取结构体的地址实例化

使用&对结构体进行取地址操作相当于对该结构体类型进行了一次new实例化操作。

p3 := &person{}

fmt.Printf("%T\n", p3) //\*main.person

fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"", city:"", age:0}

p3.name = "博客"

p3.age = 30

p3.city = "成都"

fmt.Printf("p3=%#v\n", p3) //p3=&main.person{name:"博客", city:"成都", age:30}

p3.name = "博客"其实在底层是(\*p3).name = "博客"，这是Go语言帮我们实现的语法糖。

#### 15.3.3. 结构体初始化

type person struct {

name string

city string

age int8

}

func main() {

var p4 person

fmt.Printf("p4=%#v\n", p4) //p4=main.person{name:"", city:"", age:0}

}

#### 15.3.4. 使用键值对初始化

使用键值对对结构体进行初始化时，键对应结构体的字段，值对应该字段的初始值。

p5 := person{

name: "pprof.cn",

city: "北京",

age: 18,

}

fmt.Printf("p5=%#v\n", p5) //p5=main.person{name:"pprof.cn", city:"北京", age:18}

也可以对结构体指针进行键值对初始化，例如：

p6 := &person{

name: "pprof.cn",

city: "北京",

age: 18,

}

fmt.Printf("p6=%#v\n", p6) //p6=&main.person{name:"pprof.cn", city:"北京", age:18}

当某些字段没有初始值的时候，该字段可以不写。此时，没有指定初始值的字段的值就是该字段类型的零值。

p7 := &person{

city: "北京",

}

fmt.Printf("p7=%#v\n", p7) //p7=&main.person{name:"", city:"北京", age:0}

#### 15.3.5. 使用值的列表初始化

初始化结构体的时候可以简写，也就是初始化的时候不写键，直接写值：

p8 := &person{

"pprof.cn",

"北京",

18,

}

fmt.Printf("p8=%#v\n", p8) //p8=&main.person{name:"pprof.cn", city:"北京", age:18}

使用这种格式初始化时，需要注意：

1.必须初始化结构体的所有字段。

2.初始值的填充顺序必须与字段在结构体中的声明顺序一致。

3.该方式不能和键值初始化方式混用。

#### 15.3.6. 结构体内存布局

type test struct {

a int8

b int8

c int8

d int8

}

n := test{

1, 2, 3, 4,

}

fmt.Printf("n.a %p\n", &n.a)

fmt.Printf("n.b %p\n", &n.b)

fmt.Printf("n.c %p\n", &n.c)

fmt.Printf("n.d %p\n", &n.d)

输出：

n.a 0xc0000a0060

n.b 0xc0000a0061

n.c 0xc0000a0062

n.d 0xc0000a0063

#### 15.3.7. 面试题

type student struct {

name string

age int

}

func main() {

m := make(map[string]\*student)

stus := []student{

{name: "pprof.cn", age: 18},

{name: "测试", age: 23},

{name: "博客", age: 28},

}

for \_, stu := range stus {

m[stu.name] = &stu

}

for k, v := range m {

fmt.Println(k, "=>", v.name)

}

}

#### 15.3.8. 构造函数

Go语言的结构体没有构造函数，我们可以自己实现。 例如，下方的代码就实现了一个person的构造函数。 因为struct是值类型，如果结构体比较复杂的话，值拷贝性能开销会比较大，所以该构造函数返回的是结构体指针类型。

func newPerson(name, city string, age int8) \*person {

return &person{

name: name,

city: city,

age: age,

}

}

调用构造函数

p9 := newPerson("pprof.cn", "测试", 90)

fmt.Printf("%#v\n", p9)

#### 15.3.9. 方法和接收者

Go语言中的方法（Method）是一种作用于特定类型变量的函数。这种特定类型变量叫做接收者（Receiver）。接收者的概念就类似于其他语言中的this或者 self。

方法的定义格式如下：

func (接收者变量 接收者类型) 方法名(参数列表) (返回参数) {

函数体

}

其中，

1.接收者变量：接收者中的参数变量名在命名时，官方建议使用接收者类型名的第一个小写字母，而不是self、this之类的命名。例如，Person类型的接收者变量应该命名为 p，Connector类型的接收者变量应该命名为c等。

2.接收者类型：接收者类型和参数类似，可以是指针类型和非指针类型。

3.方法名、参数列表、返回参数：具体格式与函数定义相同。

举个例子：

//Person 结构体

type Person struct {

name string

age int8

}

//NewPerson 构造函数

func NewPerson(name string, age int8) \*Person {

return &Person{

name: name,

age: age,

}

}

//Dream Person做梦的方法

func (p Person) Dream() {

fmt.Printf("%s的梦想是学好Go语言！\n", p.name)

}

func main() {

p1 := NewPerson("测试", 25)

p1.Dream()

}

方法与函数的区别是，函数不属于任何类型，方法属于特定的类型。

#### 15.3.10. 指针类型的接收者

指针类型的接收者由一个结构体的指针组成，由于指针的特性，调用方法时修改接收者指针的任意成员变量，在方法结束后，修改都是有效的。这种方式就十分接近于其他语言中面向对象中的this或者self。 例如我们为Person添加一个SetAge方法，来修改实例变量的年龄。

// SetAge 设置p的年龄

// 使用指针接收者

func (p \*Person) SetAge(newAge int8) {

p.age = newAge

}

调用该方法：

func main() {

p1 := NewPerson("测试", 25)

fmt.Println(p1.age) // 25

p1.SetAge(30)

fmt.Println(p1.age) // 30

}

#### 15.3.11. 值类型的接收者

当方法作用于值类型接收者时，Go语言会在代码运行时将接收者的值复制一份。在值类型接收者的方法中可以获取接收者的成员值，但修改操作只是针对副本，无法修改接收者变量本身。

// SetAge2 设置p的年龄

// 使用值接收者

func (p Person) SetAge2(newAge int8) {

p.age = newAge

}

func main() {

p1 := NewPerson("测试", 25)

p1.Dream()

fmt.Println(p1.age) // 25

p1.SetAge2(30) // (\*p1).SetAge2(30)

fmt.Println(p1.age) // 25

}

#### 15.3.12. 什么时候应该使用指针类型接收者

1.需要修改接收者中的值

2.接收者是拷贝代价比较大的大对象

3.保证一致性，如果有某个方法使用了指针接收者，那么其他的方法也应该使用指针接收者。

#### 15.3.13. 任意类型添加方法

在Go语言中，接收者的类型可以是任何类型，不仅仅是结构体，任何类型都可以拥有方法。 举个例子，我们基于内置的int类型使用type关键字可以定义新的自定义类型，然后为我们的自定义类型添加方法。

//MyInt 将int定义为自定义MyInt类型

type MyInt int

//SayHello 为MyInt添加一个SayHello的方法

func (m MyInt) SayHello() {

fmt.Println("Hello, 我是一个int。")

}

func main() {

var m1 MyInt

m1.SayHello() //Hello, 我是一个int。

m1 = 100

fmt.Printf("%#v %T\n", m1, m1) //100 main.MyInt

}

注意事项： 非本地类型不能定义方法，也就是说我们不能给别的包的类型定义方法。

#### 15.3.14. 结构体的匿名字段

结构体允许其成员字段在声明时没有字段名而只有类型，这种没有名字的字段就称为匿名字段。

//Person 结构体Person类型

type Person struct {

string

int

}

func main() {

p1 := Person{

"pprof.cn",

18,

}

fmt.Printf("%#v\n", p1) //main.Person{string:"pprof.cn", int:18}

fmt.Println(p1.string, p1.int) //pprof.cn 18

}

匿名字段默认采用类型名作为字段名，结构体要求字段名称必须唯一，因此一个结构体中同种类型的匿名字段只能有一个。

#### 15.3.15. 嵌套结构体

一个结构体中可以嵌套包含另一个结构体或结构体指针。

//Address 地址结构体

type Address struct {

Province string

City string

}

//User 用户结构体

type User struct {

Name string

Gender string

Address Address

}

func main() {

user1 := User{

Name: "pprof",

Gender: "女",

Address: Address{

Province: "黑龙江",

City: "哈尔滨",

},

}

fmt.Printf("user1=%#v\n", user1)//user1=main.User{Name:"pprof", Gender:"女", Address:main.Address{Province:"黑龙江", City:"哈尔滨"}}

}

#### 15.3.16. 嵌套匿名结构体

//Address 地址结构体

type Address struct {

Province string

City string

}

//User 用户结构体

type User struct {

Name string

Gender string

Address //匿名结构体

}

func main() {

var user2 User

user2.Name = "pprof"

user2.Gender = "女"

user2.Address.Province = "黑龙江" //通过匿名结构体.字段名访问

user2.City = "哈尔滨" //直接访问匿名结构体的字段名

fmt.Printf("user2=%#v\n", user2) //user2=main.User{Name:"pprof", Gender:"女", Address:main.Address{Province:"黑龙江", City:"哈尔滨"}}

}

当访问结构体成员时会先在结构体中查找该字段，找不到再去匿名结构体中查找。

#### 15.3.17. 嵌套结构体的字段名冲突

嵌套结构体内部可能存在相同的字段名。这个时候为了避免歧义需要指定具体的内嵌结构体的字段。

//Address 地址结构体

type Address struct {

Province string

City string

CreateTime string

}

//Email 邮箱结构体

type Email struct {

Account string

CreateTime string

}

//User 用户结构体

type User struct {

Name string

Gender string

Address

Email

}

func main() {

var user3 User

user3.Name = "pprof"

user3.Gender = "女"

// user3.CreateTime = "2019" //ambiguous selector user3.CreateTime

user3.Address.CreateTime = "2000" //指定Address结构体中的CreateTime

user3.Email.CreateTime = "2000" //指定Email结构体中的CreateTime

}

#### 15.3.18. 结构体的“继承”

Go语言中使用结构体也可以实现其他编程语言中面向对象的继承。

//Animal 动物

type Animal struct {

name string

}

func (a \*Animal) move() {

fmt.Printf("%s会动！\n", a.name)

}

//Dog 狗

type Dog struct {

Feet int8

\*Animal //通过嵌套匿名结构体实现继承

}

func (d \*Dog) wang() {

fmt.Printf("%s会汪汪汪~\n", d.name)

}

func main() {

d1 := &Dog{

Feet: 4,

Animal: &Animal{ //注意嵌套的是结构体指针

name: "乐乐",

},

}

d1.wang() //乐乐会汪汪汪~

d1.move() //乐乐会动！

}

#### 15.3.19. 结构体字段的可见性

结构体中字段大写开头表示可公开访问，小写表示私有（仅在定义当前结构体的包中可访问）。

#### 15.3.20. 结构体与JSON序列化

JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式。易于人阅读和编写。同时也易于机器解析和生成。JSON键值对是用来保存JS对象的一种方式，键/值对组合中的键名写在前面并用双引号""包裹，使用冒号:分隔，然后紧接着值；多个键值之间使用英文,分隔。

//Student 学生

type Student struct {

ID int

Gender string

Name string

}

//Class 班级

type Class struct {

Title string

Students []\*Student

}

func main() {

c := &Class{

Title: "101",

Students: make([]\*Student, 0, 200),

}

for i := 0; i < 10; i++ {

stu := &Student{

Name: fmt.Sprintf("stu%02d", i),

Gender: "男",

ID: i,

}

c.Students = append(c.Students, stu)

}

//JSON序列化：结构体-->JSON格式的字符串

data, err := json.Marshal(c)

if err != nil {

fmt.Println("json marshal failed")

return

}

fmt.Printf("json:%s\n", data)

//JSON反序列化：JSON格式的字符串-->结构体

str := `{"Title":"101","Students":[{"ID":0,"Gender":"男","Name":"stu00"},{"ID":1,"Gender":"男","Name":"stu01"},{"ID":2,"Gender":"男","Name":"stu02"},{"ID":3,"Gender":"男","Name":"stu03"},{"ID":4,"Gender":"男","Name":"stu04"},{"ID":5,"Gender":"男","Name":"stu05"},{"ID":6,"Gender":"男","Name":"stu06"},{"ID":7,"Gender":"男","Name":"stu07"},{"ID":8,"Gender":"男","Name":"stu08"},{"ID":9,"Gender":"男","Name":"stu09"}]}`

c1 := &Class{}

err = json.Unmarshal([]byte(str), c1)

if err != nil {

fmt.Println("json unmarshal failed!")

return

}

fmt.Printf("%#v\n", c1)

}

#### 15.3.21. 结构体标签（Tag）

Tag是结构体的元信息，可以在运行的时候通过反射的机制读取出来。

Tag在结构体字段的后方定义，由一对反引号包裹起来，具体的格式如下：

`key1:"value1" key2:"value2"`

结构体标签由一个或多个键值对组成。键与值使用冒号分隔，值用双引号括起来。键值对之间使用一个空格分隔。 注意事项： 为结构体编写Tag时，必须严格遵守键值对的规则。结构体标签的解析代码的容错能力很差，一旦格式写错，编译和运行时都不会提示任何错误，通过反射也无法正确取值。例如不要在key和value之间添加空格。

例如我们为Student结构体的每个字段定义json序列化时使用的Tag：

//Student 学生

type Student struct {

ID int `json:"id"` //通过指定tag实现json序列化该字段时的key

Gender string //json序列化是默认使用字段名作为key

name string //私有不能被json包访问

}

func main() {

s1 := Student{

ID: 1,

Gender: "女",

name: "pprof",

}

data, err := json.Marshal(s1)

if err != nil {

fmt.Println("json marshal failed!")

return

}

fmt.Printf("json str:%s\n", data) //json str:{"id":1,"Gender":"女"}

}

#### 15.3.22. 小练习：

猜一下下列代码运行的结果是什么

package main

import "fmt"

type student struct {

id int

name string

age int

}

func demo(ce []student) {

//切片是引用传递，是可以改变值的

ce[1].age = 999

// ce = append(ce, student{3, "xiaowang", 56})

// return ce

}

func main() {

var ce []student //定义一个切片类型的结构体

ce = []student{

student{1, "xiaoming", 22},

student{2, "xiaozhang", 33},

}

fmt.Println(ce)

demo(ce)

fmt.Println(ce)

}

#### 15.3.23. 删除map类型的结构体

package main

import "fmt"

type student struct {

id int

name string

age int

}

func main() {

ce := make(map[int]student)

ce[1] = student{1, "xiaolizi", 22}

ce[2] = student{2, "wang", 23}

fmt.Println(ce)

delete(ce, 2)

fmt.Println(ce)

}

#### 15.3.24. 实现map有序输出(面试经常问到)

package main

import (

"fmt"

"sort"

)

func main() {

map1 := make(map[int]string, 5)

map1[1] = "www.topgoer.com"

map1[2] = "rpc.topgoer.com"

map1[5] = "ceshi"

map1[3] = "xiaohong"

map1[4] = "xiaohuang"

sli := []int{}

for k, \_ := range map1 {

sli = append(sli, k)

}

sort.Ints(sli)

for i := 0; i < len(map1); i++ {

fmt.Println(map1[sli[i]])

}

}

#### 15.3.25. 小案例

采用切片类型的结构体接受查询数据库信息返回的参数

地址：<https://github.com/lu569368/struct>

# 四、流程控制

## 1. 条件语句if

### 1.1.1. Go 语言条件语句：

条件语句需要开发者通过指定一个或多个条件，并通过测试条件是否为 true 来决定是否执行指定语句，并在条件为 false 的情况在执行另外的语句。

Go 语言提供了以下几种条件判断语句：

### 1.1.2. if 语句 if 语句 由一个布尔表达式后紧跟一个或多个语句组成。

Go 编程语言中 if 语句的语法如下：

• 可省略条件表达式括号。

• 持初始化语句，可定义代码块局部变量。

• 代码块左 括号必须在条件表达式尾部。

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则不执行。

x := 0

// if x > 10 // Error: missing condition in if statement

// {

// }

if n := "abc"; x > 0 { // 初始化语句未必就是定义变量， 如 println("init") 也是可以的。

println(n[2])

} else if x < 0 { // 注意 else if 和 else 左大括号位置。

println(n[1])

} else {

println(n[0])

}

\*不支持三元操作符(三目运算符) "a > b ? a : b"。

#### 实例:

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var a int = 10

/\* 使用 if 语句判断布尔表达式 \*/

if a < 20 {

/\* 如果条件为 true 则执行以下语句 \*/

fmt.Printf("a 小于 20\n" )

}

fmt.Printf("a 的值为 : %d\n", a)

}

以上代码执行结果为：

a 小于 20

a 的值为 : 10

### 1.1.3. if...else 语句 if 语句 后可以使用可选的 else 语句, else 语句中的表达式在布尔表达式为 false 时执行。

#### 语法

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 {

/\* 在布尔表达式为 true 时执行 \*/

} else {

/\* 在布尔表达式为 false 时执行 \*/

}

if 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则执行 else 语句块。

#### 实例:

package main

import "fmt"

func main() {

/\* 局部变量定义 \*/

var a int = 100

/\* 判断布尔表达式 \*/

if a < 20 {

/\* 如果条件为 true 则执行以下语句 \*/

fmt.Printf("a 小于 20\n" )

} else {

/\* 如果条件为 false 则执行以下语句 \*/

fmt.Printf("a 不小于 20\n" )

}

fmt.Printf("a 的值为 : %d\n", a)

}

以上代码执行结果为：

a 不小于 20

a 的值为 : 100

### 1.1.4. if 嵌套语句 你可以在 if 或 else if 语句中嵌入一个或多个 if 或 else if 语句。

#### 语法

Go 编程语言中 if...else 语句的语法如下：

if 布尔表达式 1 {

/\* 在布尔表达式 1 为 true 时执行 \*/

if 布尔表达式 2 {

/\* 在布尔表达式 2 为 true 时执行 \*/

}

}

你可以以同样的方式在 if 语句中嵌套 else if...else 语句

#### 实例

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var a int = 100

var b int = 200

/\* 判断条件 \*/

if a == 100 {

/\* if 条件语句为 true 执行 \*/

if b == 200 {

/\* if 条件语句为 true 执行 \*/

fmt.Printf("a 的值为 100 ， b 的值为 200\n" )

}

}

fmt.Printf("a 值为 : %d\n", a )

fmt.Printf("b 值为 : %d\n", b )

}

以上代码执行结果为

a 的值为 100 ， b 的值为 200

a 值为 : 100

b 值为 : 200

## 2. 条件语句switch

### 2.1.1. switch 语句

switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上直下逐一测试，直到匹配为止。 Golang switch 分支表达式可以是任意类型，不限于常量。可省略 break，默认自动终止。

#### 语法

Go 编程语言中 switch 语句的语法如下：

switch var1 {

case val1:

...

case val2:

...

default:

...

}

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。 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。

#### 实例:

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var grade string = "B"

var marks int = 90

switch marks {

case 90: grade = "A"

case 80: grade = "B"

case 50,60,70 : grade = "C"

default: grade = "D"

}

switch {

case grade == "A" :

fmt.Printf("优秀!\n" )

case grade == "B", grade == "C" :

fmt.Printf("良好\n" )

case grade == "D" :

fmt.Printf("及格\n" )

case grade == "F":

fmt.Printf("不及格\n" )

default:

fmt.Printf("差\n" )

}

fmt.Printf("你的等级是 %s\n", grade )

}

以上代码执行结果为：

优秀!

你的等级是 A

### 2.1.2. Type Switch

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

#### Type Switch 语法格式如下：

switch x.(type){

case type:

statement(s)

case type:

statement(s)

/\* 你可以定义任意个数的case \*/

default: /\* 可选 \*/

statement(s)

}

#### 实例：

package main

import "fmt"

func main() {

var x interface{}

//写法一：

switch i := x.(type) { // 带初始化语句

case nil:

fmt.Printf(" x 的类型 :%T\r\n", i)

case int:

fmt.Printf("x 是 int 型")

case float64:

fmt.Printf("x 是 float64 型")

case func(int) float64:

fmt.Printf("x 是 func(int) 型")

case bool, string:

fmt.Printf("x 是 bool 或 string 型")

default:

fmt.Printf("未知型")

}

//写法二

var j = 0

switch j {

case 0:

case 1:

fmt.Println("1")

case 2:

fmt.Println("2")

default:

fmt.Println("def")

}

//写法三

var k = 0

switch k {

case 0:

println("fallthrough")

fallthrough

/\*

Go的switch非常灵活，表达式不必是常量或整数，执行的过程从上至下，直到找到匹配项；

而如果switch没有表达式，它会匹配true。

Go里面switch默认相当于每个case最后带有break，

匹配成功后不会自动向下执行其他case，而是跳出整个switch,

但是可以使用fallthrough强制执行后面的case代码。

\*/

case 1:

fmt.Println("1")

case 2:

fmt.Println("2")

default:

fmt.Println("def")

}

//写法三

var m = 0

switch m {

case 0, 1:

fmt.Println("1")

case 2:

fmt.Println("2")

default:

fmt.Println("def")

}

//写法四

var n = 0

switch { //省略条件表达式，可当 if...else if...else

case n > 0 && n < 10:

fmt.Println("i > 0 and i < 10")

case n > 10 && n < 20:

fmt.Println("i > 10 and i < 20")

default:

fmt.Println("def")

}

}

以上代码执行结果为：

x 的类型 :<nil>

fallthrough

1

1

def

## 3. 条件语句select

### 3.1.1. select 语句

select 语句类似于 switch 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。

select 是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。 select 随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。

#### 语法

Go 编程语言中 select 语句的语法如下：

select {

case communication clause :

statement(s);

case communication clause :

statement(s);

/\* 你可以定义任意数量的 case \*/

default : /\* 可选 \*/

statement(s);

}

以下描述了 select 语句的语法：

每个case都必须是一个通信

所有channel表达式都会被求值

所有被发送的表达式都会被求值

如果任意某个通信可以进行，它就执行；其他被忽略。

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。

否则：

如果有default子句，则执行该语句。

如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

#### 实例：

package main

import "fmt"

func main() {

var c1, c2, c3 chan int

var i1, i2 int

select {

case i1 = <-c1:

fmt.Printf("received ", i1, " from c1\n")

case c2 <- i2:

fmt.Printf("sent ", i2, " to c2\n")

case i3, ok := (<-c3): // same as: i3, ok := <-c3

if ok {

fmt.Printf("received ", i3, " from c3\n")

} else {

fmt.Printf("c3 is closed\n")

}

default:

fmt.Printf("no communication\n")

}

}

以上代码执行结果为：

no communication

select可以监听channel的数据流动

select的用法与switch语法非常类似，由select开始的一个新的选择块，每个选择条件由case语句来描述

与switch语句可以选择任何使用相等比较的条件相比，select由比较多的限制，其中最大的一条限制就是每个case语句里必须是一个IO操作

select { //不停的在这里检测

case <-chanl : //检测有没有数据可以读

//如果chanl成功读取到数据，则进行该case处理语句

case chan2 <- 1 : //检测有没有可以写

//如果成功向chan2写入数据，则进行该case处理语句

//假如没有default，那么在以上两个条件都不成立的情况下，就会在此阻塞//一般default会不写在里面，select中的default子句总是可运行的，因为会很消耗CPU资源

default:

//如果以上都没有符合条件，那么则进行default处理流程

}

在一个select语句中，Go会按顺序从头到尾评估每一个发送和接收的语句。

如果其中的任意一个语句可以继续执行（即没有被阻塞），那么就从那些可以执行的语句中任意选择一条来使用。 如果没有任意一条语句可以执行（即所有的通道都被阻塞），那么有两种可能的情况： ①如果给出了default语句，那么就会执行default的流程，同时程序的执行会从select语句后的语句中恢复。 ②如果没有default语句，那么select语句将被阻塞，直到至少有一个case可以进行下去。

### 3.1.2. Golang select的使用及典型用法

#### 基本使用

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

例如：

package main

import "fmt"

func main() {

var c1, c2, c3 chan int

var i1, i2 int

select {

case i1 = <-c1:

fmt.Printf("received ", i1, " from c1\n")

case c2 <- i2:

fmt.Printf("sent ", i2, " to c2\n")

case i3, ok := (<-c3): // same as: i3, ok := <-c3

if ok {

fmt.Printf("received ", i3, " from c3\n")

} else {

fmt.Printf("c3 is closed\n")

}

default:

fmt.Printf("no communication\n")

}

}

//输出：no communication

### 3.1.3. 典型用法

#### 1.超时判断

//比如在下面的场景中，使用全局resChan来接受response，如果时间超过3S,resChan中还没有数据返回，则第二条case将执行

var resChan = make(chan int)

// do request

func test() {

select {

case data := <-resChan:

doData(data)

case <-time.After(time.Second \* 3):

fmt.Println("request time out")

}

}

func doData(data int) {

//...

}

#### 2.退出

//主线程（协程）中如下：

var shouldQuit=make(chan struct{})

fun main(){

{

//loop

}

//...out of the loop

select {

case <-c.shouldQuit:

cleanUp()

return

default:

}

//...

}

//再另外一个协程中，如果运行遇到非法操作或不可处理的错误，就向shouldQuit发送数据通知程序停止运行

close(shouldQuit)

#### 3.判断channel是否阻塞

//在某些情况下是存在不希望channel缓存满了的需求的，可以用如下方法判断

ch := make (chan int, 5)

//...

data：=0

select {

case ch <- data:

default:

//做相应操作，比如丢弃data。视需求而定

}

## 4. 循环语句for

### 4.1.1. Golang for支持三种循环方式，包括类似 while 的语法。

for循环是一个循环控制结构，可以执行指定次数的循环。

#### 语法

Go语言的For循环有3中形式，只有其中的一种使用分号。

for init; condition; post { }

for condition { }

for { }

init： 一般为赋值表达式，给控制变量赋初值；

condition： 关系表达式或逻辑表达式，循环控制条件；

post： 一般为赋值表达式，给控制变量增量或减量。

for语句执行过程如下：

①先对表达式 init 赋初值；

②判别赋值表达式 init 是否满足给定 condition 条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止for循环，执行循环体外语句。

s := "abc"

for i, n := 0, len(s); i < n; i++ { // 常见的 for 循环，支持初始化语句。

println(s[i])

}

n := len(s)

for n > 0 { // 替代 while (n > 0) {}

println(s[n]) // 替代 for (; n > 0;) {}

n--

}

for { // 替代 while (true) {}

println(s) // 替代 for (;;) {}

}

不要期望编译器能理解你的想法，在初始化语句中计算出全部结果是个好主意。

package main

func length(s string) int {

println("call length.")

return len(s)

}

func main() {

s := "abcd"

for i, n := 0, length(s); i < n; i++ { // 避免多次调用 length 函数。

println(i, s[i])

}

}

输出:

call length.

0 97

1 98

2 99

3 100

#### 实例：

package main

import "fmt"

func main() {

var b int = 15

var a int

numbers := [6]int{1, 2, 3, 5}

/\* for 循环 \*/

for a := 0; a < 10; a++ {

fmt.Printf("a 的值为: %d\n", a)

}

for a < b {

a++

fmt.Printf("a 的值为: %d\n", a)

}

for i,x:= range numbers {

fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)

}

}

以上实例运行输出结果为:

a 的值为: 0

a 的值为: 1

a 的值为: 2

a 的值为: 3

a 的值为: 4

a 的值为: 5

a 的值为: 6

a 的值为: 7

a 的值为: 8

a 的值为: 9

a 的值为: 1

a 的值为: 2

a 的值为: 3

a 的值为: 4

a 的值为: 5

a 的值为: 6

a 的值为: 7

a 的值为: 8

a 的值为: 9

a 的值为: 10

a 的值为: 11

a 的值为: 12

a 的值为: 13

a 的值为: 14

a 的值为: 15

第 0 位 x 的值 = 1

第 1 位 x 的值 = 2

第 2 位 x 的值 = 3

第 3 位 x 的值 = 5

第 4 位 x 的值 = 0

第 5 位 x 的值 = 0

### 4.1.2. 循环嵌套

在 for 循环中嵌套一个或多个 for 循环

#### 语法

以下为 Go 语言嵌套循环的格式：

for [condition | ( init; condition; increment ) | Range]

{

for [condition | ( init; condition; increment ) | Range]

{

statement(s)

}

statement(s)

}

#### 实例：

以下实例使用循环嵌套来输出 2 到 100 间的素数：

package main

import "fmt"

func main() {

/\* 定义局部变量 \*/

var i, j int

for i=2; i < 100; i++ {

for j=2; j <= (i/j); j++ {

if(i%j==0) {

break // 如果发现因子，则不是素数

}

}

if(j > (i/j)) {

fmt.Printf("%d 是素数\n", i)

}

}

}

以上实例运行输出结果为:

2 是素数

3 是素数

5 是素数

7 是素数

11 是素数

13 是素数

17 是素数

19 是素数

23 是素数

29 是素数

31 是素数

37 是素数

41 是素数

43 是素数

47 是素数

53 是素数

59 是素数

61 是素数

67 是素数

71 是素数

73 是素数

79 是素数

83 是素数

89 是素数

97 是素数

### 4.1.3. 无限循环

如过循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

package main

import "fmt"

func main() {

for true {

fmt.Printf("这是无限循环。\n");

}

}

**5. 循环语句range**

Golang range类似迭代器操作，返回 (索引, 值) 或 (键, 值)。

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

for key, value := range oldMap {

newMap[key] = value

}

|  | **1st value** | **2nd value** |  |
| --- | --- | --- | --- |
| string | index | s[index] | unicode, rune |
| array/slice | index | s[index] |  |
| map | key | m[key] |  |
| channel | element |  |  |

可忽略不想要的返回值，或 "\_" 这个特殊变量。

package main

func main() {

s := "abc"

// 忽略 2nd value，支持 string/array/slice/map。

for i := range s {

println(s[i])

}

// 忽略 index。

for \_, c := range s {

println(c)

}

// 忽略全部返回值，仅迭代。

for range s {

}

m := map[string]int{"a": 1, "b": 2}

// 返回 (key, value)。

for k, v := range m {

println(k, v)

}

}

输出结果：

97

98

99

97

98

99

a 1

b 2

\*注意，range 会复制对象。

package main

import "fmt"

func main() {

a := [3]int{0, 1, 2}

for i, v := range a { // index、value 都是从复制品中取出。

if i == 0 { // 在修改前，我们先修改原数组。

a[1], a[2] = 999, 999

fmt.Println(a) // 确认修改有效，输出 [0, 999, 999]。

}

a[i] = v + 100 // 使用复制品中取出的 value 修改原数组。

}

fmt.Println(a) // 输出 [100, 101, 102]。

}

输出结果：

[0 999 999]

[100 101 102]

建议改用引用类型，其底层数据不会被复制。

package main

func main() {

s := []int{1, 2, 3, 4, 5}

for i, v := range s { // 复制 struct slice { pointer, len, cap }。

if i == 0 {

s = s[:3] // 对 slice 的修改，不会影响 range。

s[2] = 100 // 对底层数据的修改。

}

println(i, v)

}

}

输出结果:

0 1

1 2

2 100

3 4

4 5

另外两种引用类型 map、channel 是指针包装，而不像 slice 是 struct。

for 和 for range有什么区别?

主要是使用场景不同

for可以

遍历array和slice

遍历key为整型递增的map

遍历string

for range可以完成所有for可以做的事情，却能做到for不能做的，包括

遍历key为string类型的map并同时获取key和value

遍历channel

## 6. 循环控制Goto、Break、Continue

循环控制语句

循环控制语句可以控制循环体内语句的执行过程。

GO 语言支持以下几种循环控制语句：

### 6.1.1. Goto、Break、Continue

1.三个语句都可以配合标签(label)使用

2.标签名区分大小写，定以后若不使用会造成编译错误

3.continue、break配合标签(label)可用于多层循环跳出

4.goto是调整执行位置，与continue、break配合标签(label)的结果并不相同

# 五、函数

## 1. 函数定义

### 1.1.1. golang函数特点：

• 无需声明原型。

• 支持不定 变参。

• 支持多返回值。

• 支持命名返回参数。

• 支持匿名函数和闭包。

• 函数也是一种类型，一个函数可以赋值给变量。

• 不支持 嵌套 (nested) 一个包不能有两个名字一样的函数。

• 不支持 重载 (overload)

• 不支持 默认参数 (default parameter)。

### 1.1.2. 函数声明：

函数声明包含一个函数名，参数列表， 返回值列表和函数体。如果函数没有返回值，则返回列表可以省略。函数从第一条语句开始执行，直到执行return语句或者执行函数的最后一条语句。

函数可以没有参数或接受多个参数。

注意类型在变量名之后 。

当两个或多个连续的函数命名参数是同一类型，则除了最后一个类型之外，其他都可以省略。

函数可以返回任意数量的返回值。

使用关键字 func 定义函数，左大括号依旧不能另起一行。

func test(x, y int, s string) (int, string) {

// 类型相同的相邻参数，参数类型可合并。 多返回值必须用括号。

n := x + y

return n, fmt.Sprintf(s, n)

}

函数是第一类对象，可作为参数传递。建议将复杂签名定义为函数类型，以便于阅读。

package main

import "fmt"

func test(fn func() int) int {

return fn()

}

// 定义函数类型。

type FormatFunc func(s string, x, y int) string

func format(fn FormatFunc, s string, x, y int) string {

return fn(s, x, y)

}

func main() {

s1 := test(func() int { return 100 }) // 直接将匿名函数当参数。

s2 := format(func(s string, x, y int) string {

return fmt.Sprintf(s, x, y)

}, "%d, %d", 10, 20)

println(s1, s2)

}

输出结果：

100 10, 20

有返回值的函数，必须有明确的终止语句，否则会引发编译错误。

你可能会偶尔遇到没有函数体的函数声明，这表示该函数不是以Go实现的。这样的声明定义了函数标识符。

package math

func Sin(x float64) float //implemented in assembly language

## 2. 参数

### 2.1.1. 函数参数

函数定义时指出，函数定义时有参数，该变量可称为函数的形参。形参就像定义在函数体内的局部变量。

但当调用函数，传递过来的变量就是函数的实参，函数可以通过两种方式来传递参数：

值传递：指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

func swap(x, y int) int {

... ...

}

引用传递：是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

package main

import (

"fmt"

)

/\* 定义相互交换值的函数 \*/

func swap(x, y \*int) {

var temp int

temp = \*x /\* 保存 x 的值 \*/

\*x = \*y /\* 将 y 值赋给 x \*/

\*y = temp /\* 将 temp 值赋给 y\*/

}

func main() {

var a, b int = 1, 2

/\*

调用 swap() 函数

&a 指向 a 指针，a 变量的地址

&b 指向 b 指针，b 变量的地址

\*/

swap(&a, &b)

fmt.Println(a, b)

}

输出结果：

2 1

在默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

注意1：无论是值传递，还是引用传递，传递给函数的都是变量的副本，不过，值传递是值的拷贝。引用传递是地址的拷贝，一般来说，地址拷贝更为高效。而值拷贝取决于拷贝的对象大小，对象越大，则性能越低。

注意2：map、slice、chan、指针、interface默认以引用的方式传递。

不定参数传值 就是函数的参数不是固定的，后面的类型是固定的。（可变参数）

Golang 可变参数本质上就是 slice。只能有一个，且必须是最后一个。

在参数赋值时可以不用用一个一个的赋值，可以直接传递一个数组或者切片，特别注意的是在参数后加上“…”即可。

func myfunc(args ...int) { //0个或多个参数

}

func add(a int, args…int) int { //1个或多个参数

}

func add(a int, b int, args…int) int { //2个或多个参数

}

注意：其中args是一个slice，我们可以通过arg[index]依次访问所有参数,通过len(arg)来判断传递参数的个数.

任意类型的不定参数： 就是函数的参数和每个参数的类型都不是固定的。

用interface{}传递任意类型数据是Go语言的惯例用法，而且interface{}是类型安全的。

func myfunc(args ...interface{}) {

}

代码：

package main

import (

"fmt"

)

func test(s string, n ...int) string {

var x int

for \_, i := range n {

x += i

}

return fmt.Sprintf(s, x)

}

func main() {

println(test("sum: %d", 1, 2, 3))

}

输出结果：

sum: 6

使用 slice 对象做变参时，必须展开。（slice...）

package main

import (

"fmt"

)

func test(s string, n ...int) string {

var x int

for \_, i := range n {

x += i

}

return fmt.Sprintf(s, x)

}

func main() {

s := []int{1, 2, 3}

res := test("sum: %d", s...) // slice... 展开slice

println(res)

}

## 3. 返回值

### 3.1.1. 函数返回值

"\_"标识符，用来忽略函数的某个返回值

Go 的返回值可以被命名，并且就像在函数体开头声明的变量那样使用。

返回值的名称应当具有一定的意义，可以作为文档使用。

没有参数的 return 语句返回各个返回变量的当前值。这种用法被称作“裸”返回。

直接返回语句仅应当用在像下面这样的短函数中。在长的函数中它们会影响代码的可读性。

package main

import (

"fmt"

)

func add(a, b int) (c int) {

c = a + b

return

}

func calc(a, b int) (sum int, avg int) {

sum = a + b

avg = (a + b) / 2

return

}

func main() {

var a, b int = 1, 2

c := add(a, b)

sum, avg := calc(a, b)

fmt.Println(a, b, c, sum, avg)

}

输出结果：

1 2 3 3 1

Golang返回值不能用容器对象接收多返回值。只能用多个变量，或 "\_" 忽略。

package main

func test() (int, int) {

return 1, 2

}

func main() {

// s := make([]int, 2)

// s = test() // Error: multiple-value test() in single-value context

x, \_ := test()

println(x)

}

输出结果：

1

多返回值可直接作为其他函数调用实参。

package main

func test() (int, int) {

return 1, 2

}

func add(x, y int) int {

return x + y

}

func sum(n ...int) int {

var x int

for \_, i := range n {

x += i

}

return x

}

func main() {

println(add(test()))

println(sum(test()))

}

输出结果：

3

3

命名返回参数可看做与形参类似的局部变量，最后由 return 隐式返回。

package main

func add(x, y int) (z int) {

z = x + y

return

}

func main() {

println(add(1, 2))

}

输出结果：

3

命名返回参数可被同名局部变量遮蔽，此时需要显式返回。

func add(x, y int) (z int) {

{ // 不能在一个级别，引发 "z redeclared in this block" 错误。

var z = x + y

// return // Error: z is shadowed during return

return z // 必须显式返回。

}

}

命名返回参数允许 defer 延迟调用通过闭包读取和修改。

package main

func add(x, y int) (z int) {

defer func() {

z += 100

}()

z = x + y

return

}

func main() {

println(add(1, 2))

}

输出结果：

103

显式 return 返回前，会先修改命名返回参数。

package main

func add(x, y int) (z int) {

defer func() {

println(z) // 输出: 203

}()

z = x + y

return z + 200 // 执行顺序: (z = z + 200) -> (call defer) -> (return)

}

func main() {

println(add(1, 2)) // 输出: 203

}

输出结果：

203

203

## 4. 匿名函数

### 4.1.1. 匿名函数

匿名函数是指不需要定义函数名的一种函数实现方式。1958年LISP首先采用匿名函数。

在Go里面，函数可以像普通变量一样被传递或使用，Go语言支持随时在代码里定义匿名函数。

匿名函数由一个不带函数名的函数声明和函数体组成。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

package main

import (

"fmt"

"math"

)

func main() {

getSqrt := func(a float64) float64 {

return math.Sqrt(a)

}

fmt.Println(getSqrt(4))

}

输出结果：

2

上面先定义了一个名为getSqrt 的变量，初始化该变量时和之前的变量初始化有些不同，使用了func，func是定义函数的，可是这个函数和上面说的函数最大不同就是没有函数名，也就是匿名函数。这里将一个函数当做一个变量一样的操作。

Golang匿名函数可赋值给变量，做为结构字段，或者在 channel 里传送。

package main

func main() {

// --- function variable ---

fn := func() { println("Hello, World!") }

fn()

// --- function collection ---

fns := [](func(x int) int){

func(x int) int { return x + 1 },

func(x int) int { return x + 2 },

}

println(fns[0](100))

// --- function as field ---

d := struct {

fn func() string

}{

fn: func() string { return "Hello, World!" },

}

println(d.fn())

// --- channel of function ---

fc := make(chan func() string, 2)

fc <- func() string { return "Hello, World!" }

println((<-fc)())

}

输出结果：

Hello, World!

101

Hello, World!

Hello, World!

## 5. 闭包、递归

### 5.1.1. 闭包详解

闭包的应该都听过，但到底什么是闭包呢？

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)。

“官方”的解释是：所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

维基百科讲，闭包（Closure），是引用了自由变量的函数。这个被引用的自由变量将和这个函数一同存在，即使已经离开了创造它的环境也不例外。所以，有另一种说法认为闭包是由函数和与其相关的引用环境组合而成的实体。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。

看着上面的描述，会发现闭包和匿名函数似乎有些像。可是可能还是有些云里雾里的。因为跳过闭包的创建过程直接理解闭包的定义是非常困难的。目前在JavaScript、Go、PHP、Scala、Scheme、Common Lisp、Smalltalk、Groovy、Ruby、 Python、Lua、objective c、Swift 以及Java8以上等语言中都能找到对闭包不同程度的支持。通过支持闭包的语法可以发现一个特点，他们都有垃圾回收(GC)机制。 javascript应该是普及度比较高的编程语言了，通过这个来举例应该好理解写。看下面的代码，只要关注script里方法的定义和调用就可以了。

<!DOCTYPE html>

<html lang="zh">

<head>

<title></title>

</head>

<body>

</body>

</html>

<script src="http://ajax.googleapis.com/ajax/libs/jquery/1.2.6/jquery.min.js" type="text/javascript"></script>

<script>

function a(){

var i=0;

function b(){

console.log(++i);

document.write("<h1>"+i+"</h1>");

}

return b;

}

$(function(){

var c=a();

c();

c();

c();

//a(); //不会有信息输出

document.write("<h1>=============</h1>");

var c2=a();

c2();

c2();

});

</script>

这段代码有两个特点：

函数b嵌套在函数a内部 函数a返回函数b 这样在执行完var c=a()后，变量c实际上是指向了函数b()，再执行函数c()后就会显示i的值，第一次为1，第二次为2，第三次为3，以此类推。 其实，这段代码就创建了一个闭包。因为函数a()外的变量c引用了函数a()内的函数b()，就是说：

当函数a()的内部函数b()被函数a()外的一个变量引用的时候，就创建了一个闭包。 在上面的例子中，由于闭包的存在使得函数a()返回后，a中的i始终存在，这样每次执行c()，i都是自加1后的值。 从上面可以看出闭包的作用就是在a()执行完并返回后，闭包使得Javascript的垃圾回收机制GC不会收回a()所占用的资源，因为a()的内部函数b()的执行需要依赖a()中的变量i。

在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。变量的作用域仅限于包含它们的函数，因此无法从其它程序代码部分进行访问。不过，变量的生存期是可以很长，在一次函数调用期间所创建所生成的值在下次函数调用时仍然存在。正因为这一特点，闭包可以用来完成信息隐藏，并进而应用于需要状态表达的某些编程范型中。 下面来想象另一种情况，如果a()返回的不是函数b()，情况就完全不同了。因为a()执行完后，b()没有被返回给a()的外界，只是被a()所引用，而此时a()也只会被b()引 用，因此函数a()和b()互相引用但又不被外界打扰（被外界引用），函数a和b就会被GC回收。所以直接调用a();是页面并没有信息输出。

下面来说闭包的另一要素引用环境。c()跟c2()引用的是不同的环境，在调用i++时修改的不是同一个i，因此两次的输出都是1。函数a()每进入一次，就形成了一个新的环境，对应的闭包中，函数都是同一个函数，环境却是引用不同的环境。这和c()和c()的调用顺序都是无关的。

### 5.1.2. Go的闭包

Go语言是支持闭包的，这里只是简单地讲一下在Go语言中闭包是如何实现的。 下面我来将之前的JavaScript的闭包例子用Go来实现。

package main

import (

"fmt"

)

func a() func() int {

i := 0

b := func() int {

i++

fmt.Println(i)

return i

}

return b

}

func main() {

c := a()

c()

c()

c()

a() //不会输出i

}

输出结果：

1

2

3

可以发现，输出和之前的JavaScript的代码是一致的。具体的原因和上面的也是一样的，这说明Go语言是支持闭包的。

闭包复制的是原对象指针，这就很容易解释延迟引用现象。

package main

import "fmt"

func test() func() {

x := 100

fmt.Printf("x (%p) = %d\n", &x, x)

return func() {

fmt.Printf("x (%p) = %d\n", &x, x)

}

}

func main() {

f := test()

f()

}

输出:

x (0xc42007c008) = 100

x (0xc42007c008) = 100

在汇编层 ，test 实际返回的是 FuncVal 对象，其中包含了匿名函数地址、闭包对象指针。当调 匿名函数时，只需以某个寄存器传递该对象即可。

FuncVal { func\_address, closure\_var\_pointer ... }

外部引用函数参数局部变量

package main

import "fmt"

// 外部引用函数参数局部变量

func add(base int) func(int) int {

return func(i int) int {

base += i

return base

}

}

func main() {

tmp1 := add(10)

fmt.Println(tmp1(1), tmp1(2))

// 此时tmp1和tmp2不是一个实体了

tmp2 := add(100)

fmt.Println(tmp2(1), tmp2(2))

}

返回2个闭包

package main

import "fmt"

// 返回2个函数类型的返回值

func test01(base int) (func(int) int, func(int) int) {

// 定义2个函数，并返回

// 相加

add := func(i int) int {

base += i

return base

}

// 相减

sub := func(i int) int {

base -= i

return base

}

// 返回

return add, sub

}

func main() {

f1, f2 := test01(10)

// base一直是没有消

fmt.Println(f1(1), f2(2))

// 此时base是9

fmt.Println(f1(3), f2(4))

}

### 5.1.3. Go 语言递归函数

递归，就是在运行的过程中调用自己。 一个函数调用自己，就叫做递归函数。

构成递归需具备的条件：

1.子问题须与原始问题为同样的事，且更为简单。

2.不能无限制地调用本身，须有个出口，化简为非递归状况处理。

#### 数字阶乘

一个正整数的阶乘（factorial）是所有小于及等于该数的正整数的积，并且0的阶乘为1。自然数n的阶乘写作n!。1808年，基斯顿·卡曼引进这个表示法。

package main

import "fmt"

func factorial(i int) int {

if i <= 1 {

return 1

}

return i \* factorial(i-1)

}

func main() {

var i int = 7

fmt.Printf("Factorial of %d is %d\n", i, factorial(i))

}

输出结果：

Factorial of 7 is 5040

### 5.1.4. 斐波那契数列(Fibonacci)

这个数列从第3项开始，每一项都等于前两项之和。

package main

import "fmt"

func fibonaci(i int) int {

if i == 0 {

return 0

}

if i == 1 {

return 1

}

return fibonaci(i-1) + fibonaci(i-2)

}

func main() {

var i int

for i = 0; i < 10; i++ {

fmt.Printf("%d\n", fibonaci(i))

}

}

输出结果：

0

1

1

2

3

5

8

13

21

34

## 6. 延迟调用（defer）

### 6.1.1. Golang延迟调用：

#### defer特性：

1. 关键字 defer 用于注册延迟调用。

2. 这些调用直到 return 前才被执。因此，可以用来做资源清理。

3. 多个defer语句，按先进后出的方式执行。

4. defer语句中的变量，在defer声明时就决定了。

#### defer用途：

1. 关闭文件句柄

2. 锁资源释放

3. 数据库连接释放

go语言 defer

go 语言的defer功能强大，对于资源管理非常方便，但是如果没用好，也会有陷阱。

defer 是先进后出

这个很自然,后面的语句会依赖前面的资源，因此如果先前面的资源先释放了，后面的语句就没法执行了。

package main

import "fmt"

func main() {

var whatever [5]struct{}

for i := range whatever {

defer fmt.Println(i)

}

}

输出结果：

4

3

2

1

0

#### defer 碰上闭包

package main

import "fmt"

func main() {

var whatever [5]struct{}

for i := range whatever {

defer func() { fmt.Println(i) }()

}

}

输出结果：

4

4

4

4

4

其实go说的很清楚,我们一起来看看go spec如何说的

Each time a "defer" statement executes, the function value and parameters to the call are evaluated as usualand saved anew but the actual function is not invoked.

也就是说函数正常执行,由于闭包用到的变量 i 在执行的时候已经变成4,所以输出全都是4.

#### defer f.Close

这个大家用的都很频繁,但是go语言编程举了一个可能一不小心会犯错的例子.

package main

import "fmt"

type Test struct {

name string

}

func (t \*Test) Close() {

fmt.Println(t.name, " closed")

}

func main() {

ts := []Test{{"a"}, {"b"}, {"c"}}

for \_, t := range ts {

defer t.Close()

}

}

输出结果：

c closed

c closed

c closed

这个输出并不会像我们预计的输出c b a,而是输出c c c

可是按照前面的go spec中的说明,应该输出c b a才对啊.

那我们换一种方式来调用一下.

package main

import "fmt"

type Test struct {

name string

}

func (t \*Test) Close() {

fmt.Println(t.name, " closed")

}

func Close(t Test) {

t.Close()

}

func main() {

ts := []Test{{"a"}, {"b"}, {"c"}}

for \_, t := range ts {

defer Close(t)

}

}

输出结果：

c closed

b closed

a closed

这个时候输出的就是c b a

当然,如果你不想多写一个函数,也很简单,可以像下面这样,同样会输出c b a

看似多此一举的声明

package main

import "fmt"

type Test struct {

name string

}

func (t \*Test) Close() {

fmt.Println(t.name, " closed")

}

func main() {

ts := []Test{{"a"}, {"b"}, {"c"}}

for \_, t := range ts {

t2 := t

defer t2.Close()

}

}

输出结果：

c closed

b closed

a closed

通过以上例子，结合

Each time a "defer" statement executes, the function value and parameters to the call are evaluated as usualand saved anew but the actual function is not invoked.

这句话。可以得出下面的结论：

defer后面的语句在执行的时候，函数调用的参数会被保存起来，但是不执行。也就是复制了一份。但是并没有说struct这里的this指针如何处理，通过这个例子可以看出go语言并没有把这个明确写出来的this指针当作参数来看待。

多个 defer 注册，按 FILO 次序执行 ( 先进后出 )。哪怕函数或某个延迟调用发生错误，这些调用依旧会被执行。

package main

func test(x int) {

defer println("a")

defer println("b")

defer func() {

println(100 / x) // div0 异常未被捕获，逐步往外传递，最终终止进程。

}()

defer println("c")

}

func main() {

test(0)

}

输出结果:

c

b

a

panic: runtime error: integer divide by zero

\*延迟调用参数在注册时求值或复制，可用指针或闭包 "延迟" 读取。

package main

func test() {

x, y := 10, 20

defer func(i int) {

println("defer:", i, y) // y 闭包引用

}(x) // x 被复制

x += 10

y += 100

println("x =", x, "y =", y)

}

func main() {

test()

}

输出结果:

x = 20 y = 120

defer: 10 120

\*滥用 defer 可能会导致性能问题，尤其是在一个 "大循环" 里。

package main

import (

"fmt"

"sync"

"time"

)

var lock sync.Mutex

func test() {

lock.Lock()

lock.Unlock()

}

func testdefer() {

lock.Lock()

defer lock.Unlock()

}

func main() {

func() {

t1 := time.Now()

for i := 0; i < 10000; i++ {

test()

}

elapsed := time.Since(t1)

fmt.Println("test elapsed: ", elapsed)

}()

func() {

t1 := time.Now()

for i := 0; i < 10000; i++ {

testdefer()

}

elapsed := time.Since(t1)

fmt.Println("testdefer elapsed: ", elapsed)

}()

}

输出结果:

test elapsed: 223.162µs

testdefer elapsed: 781.304µs

### 6.1.2. defer陷阱

#### defer 与 closure

package main

import (

"errors"

"fmt"

)

func foo(a, b int) (i int, err error) {

defer fmt.Printf("first defer err %v\n", err)

defer func(err error) { fmt.Printf("second defer err %v\n", err) }(err)

defer func() { fmt.Printf("third defer err %v\n", err) }()

if b == 0 {

err = errors.New("divided by zero!")

return

}

i = a / b

return

}

func main() {

foo(2, 0)

}

输出结果：

third defer err divided by zero!

second defer err <nil>

first defer err <nil>

解释：如果 defer 后面跟的不是一个 closure 最后执行的时候我们得到的并不是最新的值。

#### defer 与 return

package main

import "fmt"

func foo() (i int) {

i = 0

defer func() {

fmt.Println(i)

}()

return 2

}

func main() {

foo()

}

输出结果：

2

解释：在有具名返回值的函数中（这里具名返回值为 i），执行 return 2 的时候实际上已经将 i 的值重新赋值为 2。所以defer closure 输出结果为 2 而不是 1。

#### defer nil 函数

package main

import (

"fmt"

)

func test() {

var run func() = nil

defer run()

fmt.Println("runs")

}

func main() {

defer func() {

if err := recover(); err != nil {

fmt.Println(err)

}

}()

test()

}

输出结果：

runs

runtime error: invalid memory address or nil pointer dereference

解释：名为 test 的函数一直运行至结束，然后 defer 函数会被执行且会因为值为 nil 而产生 panic 异常。然而值得注意的是，run() 的声明是没有问题，因为在test函数运行完成后它才会被调用。

#### 在错误的位置使用 defer

当 http.Get 失败时会抛出异常。

package main

import "net/http"

func do() error {

res, err := http.Get("http://www.google.com")

defer res.Body.Close()

if err != nil {

return err

}

// ..code...

return nil

}

func main() {

do()

}

输出结果：

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

因为在这里我们并没有检查我们的请求是否成功执行，当它失败的时候，我们访问了 Body 中的空变量 res ，因此会抛出异常

#### 解决方案

总是在一次成功的资源分配下面使用 defer ，对于这种情况来说意味着：当且仅当 http.Get 成功执行时才使用 defer

package main

import "net/http"

func do() error {

res, err := http.Get("http://xxxxxxxxxx")

if res != nil {

defer res.Body.Close()

}

if err != nil {

return err

}

// ..code...

return nil

}

func main() {

do()

}

在上述的代码中，当有错误的时候，err 会被返回，否则当整个函数返回的时候，会关闭 res.Body 。

解释：在这里，你同样需要检查 res 的值是否为 nil ，这是 http.Get 中的一个警告。通常情况下，出错的时候，返回的内容应为空并且错误会被返回，可当你获得的是一个重定向 error 时， res 的值并不会为 nil ，但其又会将错误返回。上面的代码保证了无论如何 Body 都会被关闭，如果你没有打算使用其中的数据，那么你还需要丢弃已经接收的数据。

#### 不检查错误

在这里，f.Close() 可能会返回一个错误，可这个错误会被我们忽略掉

package main

import "os"

func do() error {

f, err := os.Open("book.txt")

if err != nil {

return err

}

if f != nil {

defer f.Close()

}

// ..code...

return nil

}

func main() {

do()

}

改进一下

package main

import "os"

func do() error {

f, err := os.Open("book.txt")

if err != nil {

return err

}

if f != nil {

defer func() {

if err := f.Close(); err != nil {

// log etc

}

}()

}

// ..code...

return nil

}

func main() {

do()

}

再改进一下

通过命名的返回变量来返回 defer 内的错误。

package main

import "os"

func do() (err error) {

f, err := os.Open("book.txt")

if err != nil {

return err

}

if f != nil {

defer func() {

if ferr := f.Close(); ferr != nil {

err = ferr

}

}()

}

// ..code...

return nil

}

func main() {

do()

}

释放相同的资源

如果你尝试使用相同的变量释放不同的资源，那么这个操作可能无法正常执行。

package main

import (

"fmt"

"os"

)

func do() error {

f, err := os.Open("book.txt")

if err != nil {

return err

}

if f != nil {

defer func() {

if err := f.Close(); err != nil {

fmt.Printf("defer close book.txt err %v\n", err)

}

}()

}

// ..code...

f, err = os.Open("another-book.txt")

if err != nil {

return err

}

if f != nil {

defer func() {

if err := f.Close(); err != nil {

fmt.Printf("defer close another-book.txt err %v\n", err)

}

}()

}

return nil

}

func main() {

do()

}

输出结果： defer close book.txt err close ./another-book.txt: file already closed

当延迟函数执行时，只有最后一个变量会被用到，因此，f 变量 会成为最后那个资源 (another-book.txt)。而且两个 defer 都会将这个资源作为最后的资源来关闭

解决方案：

package main

import (

"fmt"

"io"

"os"

)

func do() error {

f, err := os.Open("book.txt")

if err != nil {

return err

}

if f != nil {

defer func(f io.Closer) {

if err := f.Close(); err != nil {

fmt.Printf("defer close book.txt err %v\n", err)

}

}(f)

}

// ..code...

f, err = os.Open("another-book.txt")

if err != nil {

return err

}

if f != nil {

defer func(f io.Closer) {

if err := f.Close(); err != nil {

fmt.Printf("defer close another-book.txt err %v\n", err)

}

}(f)

}

return nil

}

func main() {

do()

}

## 7. 异常处理

Golang 没有结构化异常，使用 panic 抛出错误，recover 捕获错误。

异常的使用场景简单描述：Go中可以抛出一个panic的异常，然后在defer中通过recover捕获这个异常，然后正常处理。

panic：

1、内置函数

2、假如函数F中书写了panic语句，会终止其后要执行的代码，在panic所在函数F内如果存在要执行的defer函数列表，按照defer的逆序执行

3、返回函数F的调用者G，在G中，调用函数F语句之后的代码不会执行，假如函数G中存在要执行的defer函数列表，按照defer的逆序执行

4、直到goroutine整个退出，并报告错误

recover：

1、内置函数

2、用来控制一个goroutine的panicking行为，捕获panic，从而影响应用的行为

3、一般的调用建议

a). 在defer函数中，通过recever来终止一个goroutine的panicking过程，从而恢复正常代码的执行

b). 可以获取通过panic传递的error

注意:

1.利用recover处理panic指令，defer 必须放在 panic 之前定义，另外 recover 只有在 defer 调用的函数中才有效。否则当panic时，recover无法捕获到panic，无法防止panic扩散。

2.recover 处理异常后，逻辑并不会恢复到 panic 那个点去，函数跑到 defer 之后的那个点。

3.多个 defer 会形成 defer 栈，后定义的 defer 语句会被最先调用。

package main

func main() {

test()

}

func test() {

defer func() {

if err := recover(); err != nil {

println(err.(string)) // 将 interface{} 转型为具体类型。

}

}()

panic("panic error!")

}

输出结果：

panic error!

由于 panic、recover 参数类型为 interface{}，因此可抛出任何类型对象。

func panic(v interface{})

func recover() interface{}

向已关闭的通道发送数据会引发panic

package main

import (

"fmt"

)

func main() {

defer func() {

if err := recover(); err != nil {

fmt.Println(err)

}

}()

var ch chan int = make(chan int, 10)

close(ch)

ch <- 1

}

输出结果：

send on closed channel

延迟调用中引发的错误，可被后续延迟调用捕获，但仅最后一个错误可被捕获。

package main

import "fmt"

func test() {

defer func() {

fmt.Println(recover())

}()

defer func() {

panic("defer panic")

}()

panic("test panic")

}

func main() {

test()

}

输出:

defer panic

捕获函数 recover 只有在延迟调用内直接调用才会终止错误，否则总是返回 nil。任何未捕获的错误都会沿调用堆栈向外传递。

package main

import "fmt"

func test() {

defer func() {

fmt.Println(recover()) //有效

}()

defer recover() //无效！

defer fmt.Println(recover()) //无效！

defer func() {

func() {

println("defer inner")

recover() //无效！

}()

}()

panic("test panic")

}

func main() {

test()

}

输出:

defer inner

<nil>

test panic

使用延迟匿名函数或下面这样都是有效的。

package main

import (

"fmt"

)

func except() {

fmt.Println(recover())

}

func test() {

defer except()

panic("test panic")

}

func main() {

test()

}

输出结果：

test panic

如果需要保护代码 段，可将代码块重构成匿名函数，如此可确保后续代码被执 。

package main

import "fmt"

func test(x, y int) {

var z int

func() {

defer func() {

if recover() != nil {

z = 0

}

}()

panic("test panic")

z = x / y

return

}()

fmt.Printf("x / y = %d\n", z)

}

func main() {

test(2, 1)

}

输出结果：

x / y = 0

除用 panic 引发中断性错误外，还可返回 error 类型错误对象来表示函数调用状态。

type error interface {

Error() string

}

标准库 errors.New 和 fmt.Errorf 函数用于创建实现 error 接口的错误对象。通过判断错误对象实例来确定具体错误类型。

package main

import (

"errors"

"fmt"

)

var ErrDivByZero = errors.New("division by zero")

func div(x, y int) (int, error) {

if y == 0 {

return 0, ErrDivByZero

}

return x / y, nil

}

func main() {

defer func() {

fmt.Println(recover())

}()

switch z, err := div(10, 0); err {

case nil:

println(z)

case ErrDivByZero:

panic(err)

}

}

输出结果：

division by zero

Go实现类似 try catch 的异常处理

package main

import "fmt"

func Try(fun func(), handler func(interface{})) {

defer func() {

if err := recover(); err != nil {

handler(err)

}

}()

fun()

}

func main() {

Try(func() {

panic("test panic")

}, func(err interface{}) {

fmt.Println(err)

})

}

输出结果：

test panic

如何区别使用 panic 和 error 两种方式?

惯例是:导致关键流程出现不可修复性错误的使用 panic，其他使用 error。

## 8. 单元测试

不写测试的开发不是好程序员。我个人非常崇尚TDD（Test Driven Development）的，然而可惜的是国内的程序员都不太关注测试这一部分。 这篇文章主要介绍下在Go语言中如何做单元测试和基准测试。

### 8.1. go test工具

Go语言中的测试依赖go test命令。编写测试代码和编写普通的Go代码过程是类似的，并不需要学习新的语法、规则或工具。

go test命令是一个按照一定约定和组织的测试代码的驱动程序。在包目录内，所有以\_test.go为后缀名的源代码文件都是go test测试的一部分，不会被go build编译到最终的可执行文件中。

在\*\_test.go文件中有三种类型的函数，单元测试函数、基准测试函数和示例函数。

| **类型** | **格式** | **作用** |
| --- | --- | --- |
| 测试函数 | 函数名前缀为Test | 测试程序的一些逻辑行为是否正确 |
| 基准函数 | 函数名前缀为Benchmark | 测试函数的性能 |
| 示例函数 | 函数名前缀为Example | 为文档提供示例文档 |

go test命令会遍历所有的\*\_test.go文件中符合上述命名规则的函数，然后生成一个临时的main包用于调用相应的测试函数，然后构建并运行、报告测试结果，最后清理测试中生成的临时文件。

Golang单元测试对文件名和方法名，参数都有很严格的要求。

1、文件名必须以xx\_test.go命名

2、方法必须是Test[^a-z]开头

3、方法参数必须 t \*testing.T

4、使用go test执行单元测试

go test的参数解读：

go test是go语言自带的测试工具，其中包含的是两类，单元测试和性能测试

通过go help test可以看到go test的使用说明：

格式形如： go test [-c] [-i] [build flags] [packages] [flags for test binary]

参数解读：

-c : 编译go test成为可执行的二进制文件，但是不运行测试。

-i : 安装测试包依赖的package，但是不运行测试。

关于build flags，调用go help build，这些是编译运行过程中需要使用到的参数，一般设置为空

关于packages，调用go help packages，这些是关于包的管理，一般设置为空

关于flags for test binary，调用go help testflag，这些是go test过程中经常使用到的参数

-test.v : 是否输出全部的单元测试用例（不管成功或者失败），默认没有加上，所以只输出失败的单元测试用例。

-test.run pattern: 只跑哪些单元测试用例

-test.bench patten: 只跑那些性能测试用例

-test.benchmem : 是否在性能测试的时候输出内存情况

-test.benchtime t : 性能测试运行的时间，默认是1s

-test.cpuprofile cpu.out : 是否输出cpu性能分析文件

-test.memprofile mem.out : 是否输出内存性能分析文件

-test.blockprofile block.out : 是否输出内部goroutine阻塞的性能分析文件

-test.memprofilerate n : 内存性能分析的时候有一个分配了多少的时候才打点记录的问题。这个参数就是设置打点的内存分配间隔，也就是profile中一个sample代表的内存大小。默认是设置为512 \* 1024的。如果你将它设置为1，则每分配一个内存块就会在profile中有个打点，那么生成的profile的sample就会非常多。如果你设置为0，那就是不做打点了。

你可以通过设置memprofilerate=1和GOGC=off来关闭内存回收，并且对每个内存块的分配进行观察。

-test.blockprofilerate n: 基本同上，控制的是goroutine阻塞时候打点的纳秒数。默认不设置就相当于-test.blockprofilerate=1，每一纳秒都打点记录一下

-test.parallel n : 性能测试的程序并行cpu数，默认等于GOMAXPROCS。

-test.timeout t : 如果测试用例运行时间超过t，则抛出panic

-test.cpu 1,2,4 : 程序运行在哪些CPU上面，使用二进制的1所在位代表，和nginx的nginx\_worker\_cpu\_affinity是一个道理

-test.short : 将那些运行时间较长的测试用例运行时间缩短

目录结构：

test

|

—— calc.go

|

—— calc\_test.go

### 8.2. 测试函数

#### 8.2.1. 测试函数的格式

每个测试函数必须导入testing包，测试函数的基本格式（签名）如下：

func TestName(t \*testing.T){

// ...

}

测试函数的名字必须以Test开头，可选的后缀名必须以大写字母开头，举几个例子：\

func TestAdd(t \*testing.T){ ... }

func TestSum(t \*testing.T){ ... }

func TestLog(t \*testing.T){ ... }

其中参数t用于报告测试失败和附加的日志信息。 testing.T的拥有的方法如下：

func (c \*T) Error(args ...interface{})

func (c \*T) Errorf(format string, args ...interface{})

func (c \*T) Fail()

func (c \*T) FailNow()

func (c \*T) Failed() bool

func (c \*T) Fatal(args ...interface{})

func (c \*T) Fatalf(format string, args ...interface{})

func (c \*T) Log(args ...interface{})

func (c \*T) Logf(format string, args ...interface{})

func (c \*T) Name() string

func (t \*T) Parallel()

func (t \*T) Run(name string, f func(t \*T)) bool

func (c \*T) Skip(args ...interface{})

func (c \*T) SkipNow()

func (c \*T) Skipf(format string, args ...interface{})

func (c \*T) Skipped() bool

#### 8.2.2. 测试函数示例

就像细胞是构成我们身体的基本单位，一个软件程序也是由很多单元组件构成的。单元组件可以是函数、结构体、方法和最终用户可能依赖的任意东西。总之我们需要确保这些组件是能够正常运行的。单元测试是一些利用各种方法测试单元组件的程序，它会将结果与预期输出进行比较。

接下来，我们定义一个split的包，包中定义了一个Split函数，具体实现如下：

// split/split.go

package split

import "strings"

// split package with a single split function.

// Split slices s into all substrings separated by sep and

// returns a slice of the substrings between those separators.

func Split(s, sep string) (result []string) {

i := strings.Index(s, sep)

for i > -1 {

result = append(result, s[:i])

s = s[i+1:]

i = strings.Index(s, sep)

}

result = append(result, s)

return

}

在当前目录下，我们创建一个split\_test.go的测试文件，并定义一个测试函数如下：

// split/split\_test.go

package split

import (

"reflect"

"testing"

)

func TestSplit(t \*testing.T) { // 测试函数名必须以Test开头，必须接收一个\*testing.T类型参数

got := Split("a:b:c", ":") // 程序输出的结果

want := []string{"a", "b", "c"} // 期望的结果

if !reflect.DeepEqual(want, got) { // 因为slice不能比较直接，借助反射包中的方法比较

t.Errorf("excepted:%v, got:%v", want, got) // 测试失败输出错误提示

}

}

此时split这个包中的文件如下：

split $ ls -l

total 16

-rw-r--r-- 1 pprof staff 408 4 29 15:50 split.go

-rw-r--r-- 1 pprof staff 466 4 29 16:04 split\_test.go

在split包路径下，执行go test命令，可以看到输出结果如下：

split $ go test

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 0.005s

一个测试用例有点单薄，我们再编写一个测试使用多个字符切割字符串的例子，在split\_test.go中添加如下测试函数：

func TestMoreSplit(t \*testing.T) {

got := Split("abcd", "bc")

want := []string{"a", "d"}

if !reflect.DeepEqual(want, got) {

t.Errorf("excepted:%v, got:%v", want, got)

}

}

再次运行go test命令，输出结果如下：

split $ go test

--- FAIL: TestMultiSplit (0.00s)

split\_test.go:20: excepted:[a d], got:[a cd]

FAIL

exit status 1

FAIL github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

这一次，我们的测试失败了。我们可以为go test命令添加-v参数，查看测试函数名称和运行时间：

split $ go test -v

=== RUN TestSplit

--- PASS: TestSplit (0.00s)

=== RUN TestMoreSplit

--- FAIL: TestMoreSplit (0.00s)

split\_test.go:21: excepted:[a d], got:[a cd]

FAIL

exit status 1

FAIL github.com/pprof/studygo/code\_demo/test\_demo/split 0.005s

这一次我们能清楚的看到是TestMoreSplit这个测试没有成功。 还可以在go test命令后添加-run参数，它对应一个正则表达式，只有函数名匹配上的测试函数才会被go test命令执行。

split $ go test -v -run="More"

=== RUN TestMoreSplit

--- FAIL: TestMoreSplit (0.00s)

split\_test.go:21: excepted:[a d], got:[a cd]

FAIL

exit status 1

FAIL github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

现在我们回过头来解决我们程序中的问题。很显然我们最初的split函数并没有考虑到sep为多个字符的情况，我们来修复下这个Bug：

package split

import "strings"

// split package with a single split function.

// Split slices s into all substrings separated by sep and

// returns a slice of the substrings between those separators.

func Split(s, sep string) (result []string) {

i := strings.Index(s, sep)

for i > -1 {

result = append(result, s[:i])

s = s[i+len(sep):] // 这里使用len(sep)获取sep的长度

i = strings.Index(s, sep)

}

result = append(result, s)

return

}

这一次我们再来测试一下，我们的程序。注意，当我们修改了我们的代码之后不要仅仅执行那些失败的测试函数，我们应该完整的运行所有的测试，保证不会因为修改代码而引入了新的问题。

split $ go test -v

=== RUN TestSplit

--- PASS: TestSplit (0.00s)

=== RUN TestMoreSplit

--- PASS: TestMoreSplit (0.00s)

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

这一次我们的测试都通过了

### 8.3. 测试组

我们现在还想要测试一下split函数对中文字符串的支持，这个时候我们可以再编写一个TestChineseSplit测试函数，但是我们也可以使用如下更友好的一种方式来添加更多的测试用例。

func TestSplit(t \*testing.T) {

// 定义一个测试用例类型

type test struct {

input string

sep string

want []string

}

// 定义一个存储测试用例的切片

tests := []test{

{input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},

{input: "a:b:c", sep: ",", want: []string{"a:b:c"}},

{input: "abcd", sep: "bc", want: []string{"a", "d"}},

{input: "枯藤老树昏鸦", sep: "老", want: []string{"枯藤", "树昏鸦"}},

}

// 遍历切片，逐一执行测试用例

for \_, tc := range tests {

got := Split(tc.input, tc.sep)

if !reflect.DeepEqual(got, tc.want) {

t.Errorf("excepted:%v, got:%v", tc.want, got)

}

}

}

我们通过上面的代码把多个测试用例合到一起，再次执行go test命令。

split $ go test -v

=== RUN TestSplit

--- FAIL: TestSplit (0.00s)

split\_test.go:42: excepted:[枯藤 树昏鸦], got:[ 枯藤 树昏鸦]

FAIL

exit status 1

FAIL github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

我们的测试出现了问题，仔细看打印的测试失败提示信息：excepted:[枯藤 树昏鸦], got:[ 枯藤 树昏鸦]，你会发现[ 枯藤 树昏鸦]中有个不明显的空串，这种情况下十分推荐使用%#v的格式化方式。

我们修改下测试用例的格式化输出错误提示部分：

func TestSplit(t \*testing.T) {

...

for \_, tc := range tests {

got := Split(tc.input, tc.sep)

if !reflect.DeepEqual(got, tc.want) {

t.Errorf("excepted:%#v, got:%#v", tc.want, got)

}

}

}

此时运行go test命令后就能看到比较明显的提示信息了：

split $ go test -v

=== RUN TestSplit

--- FAIL: TestSplit (0.00s)

split\_test.go:42: excepted:[]string{"枯藤", "树昏鸦"}, got:[]string{"", "枯藤", "树昏鸦"}

FAIL

exit status 1

FAIL github.com/Q1mi/studygo/code\_demo/test\_demo/split 0.006s

### 8.4. 子测试

看起来都挺不错的，但是如果测试用例比较多的时候，我们是没办法一眼看出来具体是哪个测试用例失败了。我们可能会想到下面的解决办法

func TestSplit(t \*testing.T) {

type test struct { // 定义test结构体

input string

sep string

want []string

}

tests := map[string]test{ // 测试用例使用map存储

"simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},

"wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},

"more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},

"leading sep": {input: "枯藤老树昏鸦", sep: "老", want: []string{"枯藤", "树昏鸦"}},

}

for name, tc := range tests {

got := Split(tc.input, tc.sep)

if !reflect.DeepEqual(got, tc.want) {

t.Errorf("name:%s excepted:%#v, got:%#v", name, tc.want, got) // 将测试用例的name格式化输出

}

}

}

上面的做法是能够解决问题的。同时Go1.7+中新增了子测试，我们可以按照如下方式使用t.Run执行子测试：

func TestSplit(t \*testing.T) {

type test struct { // 定义test结构体

input string

sep string

want []string

}

tests := map[string]test{ // 测试用例使用map存储

"simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},

"wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},

"more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},

"leading sep": {input: "枯藤老树昏鸦", sep: "老", want: []string{"枯藤", "树昏鸦"}},

}

for name, tc := range tests {

t.Run(name, func(t \*testing.T) { // 使用t.Run()执行子测试

got := Split(tc.input, tc.sep)

if !reflect.DeepEqual(got, tc.want) {

t.Errorf("excepted:%#v, got:%#v", tc.want, got)

}

})

}

}

此时我们再执行go test命令就能够看到更清晰的输出内容了：

split $ go test -v

=== RUN TestSplit

=== RUN TestSplit/leading\_sep

=== RUN TestSplit/simple

=== RUN TestSplit/wrong\_sep

=== RUN TestSplit/more\_sep

--- FAIL: TestSplit (0.00s)

--- FAIL: TestSplit/leading\_sep (0.00s)

split\_test.go:83: excepted:[]string{"枯藤", "树昏鸦"}, got:[]string{"", "枯藤", "树昏鸦"}

--- PASS: TestSplit/simple (0.00s)

--- PASS: TestSplit/wrong\_sep (0.00s)

--- PASS: TestSplit/more\_sep (0.00s)

FAIL

exit status 1

FAIL github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

这个时候我们要把测试用例中的错误修改回来：

func TestSplit(t \*testing.T) {

...

tests := map[string]test{ // 测试用例使用map存储

"simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},

"wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},

"more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},

"leading sep": {input: "枯藤老树昏鸦", sep: "老", want: []string{"", "枯藤", "树昏鸦"}},

}

...

}

我们都知道可以通过-run=RegExp来指定运行的测试用例，还可以通过/来指定要运行的子测试用例，例如：go test -v -run=Split/simple只会运行simple对应的子测试用例。

### 8.5. 测试覆盖率

测试覆盖率是你的代码被测试套件覆盖的百分比。通常我们使用的都是语句的覆盖率，也就是在测试中至少被运行一次的代码占总代码的比例。

Go提供内置功能来检查你的代码覆盖率。我们可以使用go test -cover来查看测试覆盖率。例如：

split $ go test -cover

PASS

coverage: 100.0% of statements

ok github.com/pprof/studygo/code\_demo/test\_demo/split 0.005s

从上面的结果可以看到我们的测试用例覆盖了100%的代码。

Go还提供了一个额外的-coverprofile参数，用来将覆盖率相关的记录信息输出到一个文件。例如：

split $ go test -cover -coverprofile=c.out

PASS

coverage: 100.0% of statements

ok github.com/pprof/studygo/code\_demo/test\_demo/split 0.005s

上面的命令会将覆盖率相关的信息输出到当前文件夹下面的c.out文件中，然后我们执行go tool cover -html=c.out，使用cover工具来处理生成的记录信息，该命令会打开本地的浏览器窗口生成一个HTML报告。

### 8.6. 基准测试

#### 8.6.1. 基准测试函数格式

基准测试就是在一定的工作负载之下检测程序性能的一种方法。基准测试的基本格式如下：

func BenchmarkName(b \*testing.B){

// ...

}

基准测试以Benchmark为前缀，需要一个\*testing.B类型的参数b，基准测试必须要执行b.N次，这样的测试才有对照性，b.N的值是系统根据实际情况去调整的，从而保证测试的稳定性。 testing.B拥有的方法如下：

func (c \*B) Error(args ...interface{})

func (c \*B) Errorf(format string, args ...interface{})

func (c \*B) Fail()

func (c \*B) FailNow()

func (c \*B) Failed() bool

func (c \*B) Fatal(args ...interface{})

func (c \*B) Fatalf(format string, args ...interface{})

func (c \*B) Log(args ...interface{})

func (c \*B) Logf(format string, args ...interface{})

func (c \*B) Name() string

func (b \*B) ReportAllocs()

func (b \*B) ResetTimer()

func (b \*B) Run(name string, f func(b \*B)) bool

func (b \*B) RunParallel(body func(\*PB))

func (b \*B) SetBytes(n int64)

func (b \*B) SetParallelism(p int)

func (c \*B) Skip(args ...interface{})

func (c \*B) SkipNow()

func (c \*B) Skipf(format string, args ...interface{})

func (c \*B) Skipped() bool

func (b \*B) StartTimer()

func (b \*B) StopTimer()

#### 8.6.2. 基准测试示例

我们为split包中的Split函数编写基准测试如下：

func BenchmarkSplit(b \*testing.B) {

for i := 0; i < b.N; i++ {

Split("枯藤老树昏鸦", "老")

}

}

基准测试并不会默认执行，需要增加-bench参数，所以我们通过执行go test -bench=Split命令执行基准测试，输出结果如下：

split $ go test -bench=Split

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/split

BenchmarkSplit-8 10000000 203 ns/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 2.255s

其中BenchmarkSplit-8表示对Split函数进行基准测试，数字8表示GOMAXPROCS的值，这个对于并发基准测试很重要。10000000和203ns/op表示每次调用Split函数耗时203ns，这个结果是10000000次调用的平均值。

我们还可以为基准测试添加-benchmem参数，来获得内存分配的统计数据。

split $ go test -bench=Split -benchmem

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/split

BenchmarkSplit-8 10000000 215 ns/op 112 B/op 3 allocs/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 2.394s

其中，112 B/op表示每次操作内存分配了112字节，3 allocs/op则表示每次操作进行了3次内存分配。 我们将我们的Split函数优化如下：

func Split(s, sep string) (result []string) {

result = make([]string, 0, strings.Count(s, sep)+1)

i := strings.Index(s, sep)

for i > -1 {

result = append(result, s[:i])

s = s[i+len(sep):] // 这里使用len(sep)获取sep的长度

i = strings.Index(s, sep)

}

result = append(result, s)

return

}

这一次我们提前使用make函数将result初始化为一个容量足够大的切片，而不再像之前一样通过调用append函数来追加。我们来看一下这个改进会带来多大的性能提升：

split $ go test -bench=Split -benchmem

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/split

BenchmarkSplit-8 10000000 127 ns/op 48 B/op 1 allocs/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 1.423s

这个使用make函数提前分配内存的改动，减少了2/3的内存分配次数，并且减少了一半的内存分配。

#### 8.6.3. 性能比较函数

上面的基准测试只能得到给定操作的绝对耗时，但是在很多性能问题是发生在两个不同操作之间的相对耗时，比如同一个函数处理1000个元素的耗时与处理1万甚至100万个元素的耗时的差别是多少？再或者对于同一个任务究竟使用哪种算法性能最佳？我们通常需要对两个不同算法的实现使用相同的输入来进行基准比较测试。

性能比较函数通常是一个带有参数的函数，被多个不同的Benchmark函数传入不同的值来调用。举个例子如下：

func benchmark(b \*testing.B, size int){/\* ... \*/}

func Benchmark10(b \*testing.B){ benchmark(b, 10) }

func Benchmark100(b \*testing.B){ benchmark(b, 100) }

func Benchmark1000(b \*testing.B){ benchmark(b, 1000) }

例如我们编写了一个计算斐波那契数列的函数如下：

// fib.go

// Fib 是一个计算第n个斐波那契数的函数

func Fib(n int) int {

if n < 2 {

return n

}

return Fib(n-1) + Fib(n-2)

}

我们编写的性能比较函数如下：

// fib\_test.go

func benchmarkFib(b \*testing.B, n int) {

for i := 0; i < b.N; i++ {

Fib(n)

}

}

func BenchmarkFib1(b \*testing.B) { benchmarkFib(b, 1) }

func BenchmarkFib2(b \*testing.B) { benchmarkFib(b, 2) }

func BenchmarkFib3(b \*testing.B) { benchmarkFib(b, 3) }

func BenchmarkFib10(b \*testing.B) { benchmarkFib(b, 10) }

func BenchmarkFib20(b \*testing.B) { benchmarkFib(b, 20) }

func BenchmarkFib40(b \*testing.B) { benchmarkFib(b, 40) }

运行基准测试：

split $ go test -bench=.

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/fib

BenchmarkFib1-8 1000000000 2.03 ns/op

BenchmarkFib2-8 300000000 5.39 ns/op

BenchmarkFib3-8 200000000 9.71 ns/op

BenchmarkFib10-8 5000000 325 ns/op

BenchmarkFib20-8 30000 42460 ns/op

BenchmarkFib40-8 2 638524980 ns/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/fib 12.944s

这里需要注意的是，默认情况下，每个基准测试至少运行1秒。如果在Benchmark函数返回时没有到1秒，则b.N的值会按1,2,5,10,20,50，…增加，并且函数再次运行。

最终的BenchmarkFib40只运行了两次，每次运行的平均值只有不到一秒。像这种情况下我们应该可以使用-benchtime标志增加最小基准时间，以产生更准确的结果。例如：

split $ go test -bench=Fib40 -benchtime=20s

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/fib

BenchmarkFib40-8 50 663205114 ns/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/fib 33.849s

这一次BenchmarkFib40函数运行了50次，结果就会更准确一些了。

使用性能比较函数做测试的时候一个容易犯的错误就是把b.N作为输入的大小，例如以下两个例子都是错误的示范：

// 错误示范1

func BenchmarkFibWrong(b \*testing.B) {

for n := 0; n < b.N; n++ {

Fib(n)

}

}

// 错误示范2

func BenchmarkFibWrong2(b \*testing.B) {

Fib(b.N)

}

#### 8.6.4. 重置时间

b.ResetTimer之前的处理不会放到执行时间里，也不会输出到报告中，所以可以在之前做一些不计划作为测试报告的操作。例如：

func BenchmarkSplit(b \*testing.B) {

time.Sleep(5 \* time.Second) // 假设需要做一些耗时的无关操作

b.ResetTimer() // 重置计时器

for i := 0; i < b.N; i++ {

Split("枯藤老树昏鸦", "老")

}

}

#### 8.6.5. 并行测试

func (b B) RunParallel(body func(PB))会以并行的方式执行给定的基准测试。

RunParallel会创建出多个goroutine，并将b.N分配给这些goroutine执行， 其中goroutine数量的默认值为GOMAXPROCS。用户如果想要增加非CPU受限（non-CPU-bound）基准测试的并行性， 那么可以在RunParallel之前调用SetParallelism 。RunParallel通常会与-cpu标志一同使用。

func BenchmarkSplitParallel(b \*testing.B) {

// b.SetParallelism(1) // 设置使用的CPU数

b.RunParallel(func(pb \*testing.PB) {

for pb.Next() {

Split("枯藤老树昏鸦", "老")

}

})

}

执行一下基准测试：

split $ go test -bench=.

goos: darwin

goarch: amd64

pkg: github.com/pprof/studygo/code\_demo/test\_demo/split

BenchmarkSplit-8 10000000 131 ns/op

BenchmarkSplitParallel-8 50000000 36.1 ns/op

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 3.308s

还可以通过在测试命令后添加-cpu参数如go test -bench=. -cpu 1来指定使用的CPU数量。

### 8.7. Setup与TearDown

测试程序有时需要在测试之前进行额外的设置（setup）或在测试之后进行拆卸（teardown）。

#### 8.7.1. TestMain

通过在\*\_test.go文件中定义TestMain函数来可以在测试之前进行额外的设置（setup）或在测试之后进行拆卸（teardown）操作。

如果测试文件包含函数:func TestMain(m \*testing.M)那么生成的测试会先调用 TestMain(m)，然后再运行具体测试。TestMain运行在主goroutine中, 可以在调用 m.Run前后做任何设置（setup）和拆卸（teardown）。退出测试的时候应该使用m.Run的返回值作为参数调用os.Exit。

一个使用TestMain来设置Setup和TearDown的示例如下：

func TestMain(m \*testing.M) {

fmt.Println("write setup code here...") // 测试之前的做一些设置

// 如果 TestMain 使用了 flags，这里应该加上flag.Parse()

retCode := m.Run() // 执行测试

fmt.Println("write teardown code here...") // 测试之后做一些拆卸工作

os.Exit(retCode) // 退出测试

}

需要注意的是：在调用TestMain时, flag.Parse并没有被调用。所以如果TestMain 依赖于command-line标志 (包括 testing 包的标记), 则应该显示的调用flag.Parse。

#### 8.7.2. 子测试的Setup与Teardown

有时候我们可能需要为每个测试集设置Setup与Teardown，也有可能需要为每个子测试设置Setup与Teardown。下面我们定义两个函数工具函数如下：

// 测试集的Setup与Teardown

func setupTestCase(t \*testing.T) func(t \*testing.T) {

t.Log("如有需要在此执行:测试之前的setup")

return func(t \*testing.T) {

t.Log("如有需要在此执行:测试之后的teardown")

}

}

// 子测试的Setup与Teardown

func setupSubTest(t \*testing.T) func(t \*testing.T) {

t.Log("如有需要在此执行:子测试之前的setup")

return func(t \*testing.T) {

t.Log("如有需要在此执行:子测试之后的teardown")

}

}

使用方式如下：

func TestSplit(t \*testing.T) {

type test struct { // 定义test结构体

input string

sep string

want []string

}

tests := map[string]test{ // 测试用例使用map存储

"simple": {input: "a:b:c", sep: ":", want: []string{"a", "b", "c"}},

"wrong sep": {input: "a:b:c", sep: ",", want: []string{"a:b:c"}},

"more sep": {input: "abcd", sep: "bc", want: []string{"a", "d"}},

"leading sep": {input: "枯藤老树昏鸦", sep: "老", want: []string{"", "枯藤", "树昏鸦"}},

}

teardownTestCase := setupTestCase(t) // 测试之前执行setup操作

defer teardownTestCase(t) // 测试之后执行testdoen操作

for name, tc := range tests {

t.Run(name, func(t \*testing.T) { // 使用t.Run()执行子测试

teardownSubTest := setupSubTest(t) // 子测试之前执行setup操作

defer teardownSubTest(t) // 测试之后执行testdoen操作

got := Split(tc.input, tc.sep)

if !reflect.DeepEqual(got, tc.want) {

t.Errorf("excepted:%#v, got:%#v", tc.want, got)

}

})

}

}

测试结果如下：

split $ go test -v

=== RUN TestSplit

=== RUN TestSplit/simple

=== RUN TestSplit/wrong\_sep

=== RUN TestSplit/more\_sep

=== RUN TestSplit/leading\_sep

--- PASS: TestSplit (0.00s)

split\_test.go:71: 如有需要在此执行:测试之前的setup

--- PASS: TestSplit/simple (0.00s)

split\_test.go:79: 如有需要在此执行:子测试之前的setup

split\_test.go:81: 如有需要在此执行:子测试之后的teardown

--- PASS: TestSplit/wrong\_sep (0.00s)

split\_test.go:79: 如有需要在此执行:子测试之前的setup

split\_test.go:81: 如有需要在此执行:子测试之后的teardown

--- PASS: TestSplit/more\_sep (0.00s)

split\_test.go:79: 如有需要在此执行:子测试之前的setup

split\_test.go:81: 如有需要在此执行:子测试之后的teardown

--- PASS: TestSplit/leading\_sep (0.00s)

split\_test.go:79: 如有需要在此执行:子测试之前的setup

split\_test.go:81: 如有需要在此执行:子测试之后的teardown

split\_test.go:73: 如有需要在此执行:测试之后的teardown

=== RUN ExampleSplit

--- PASS: ExampleSplit (0.00s)

PASS

ok github.com/Q1mi/studygo/code\_demo/test\_demo/split 0.006s

### 8.8. 示例函数

#### 8.8.1. 示例函数的格式

被go test特殊对待的第三种函数就是示例函数，它们的函数名以Example为前缀。它们既没有参数也没有返回值。标准格式如下：

func ExampleName() {

// ...

}

#### 8.8.2. 示例函数示例

下面的代码是我们为Split函数编写的一个示例函数：

func ExampleSplit() {

fmt.Println(split.Split("a:b:c", ":"))

fmt.Println(split.Split("枯藤老树昏鸦", "老"))

// Output:

// [a b c]

// [ 枯藤 树昏鸦]

}

为你的代码编写示例代码有如下三个用处：

示例函数能够作为文档直接使用，例如基于web的godoc中能把示例函数与对应的函数或包相关联。

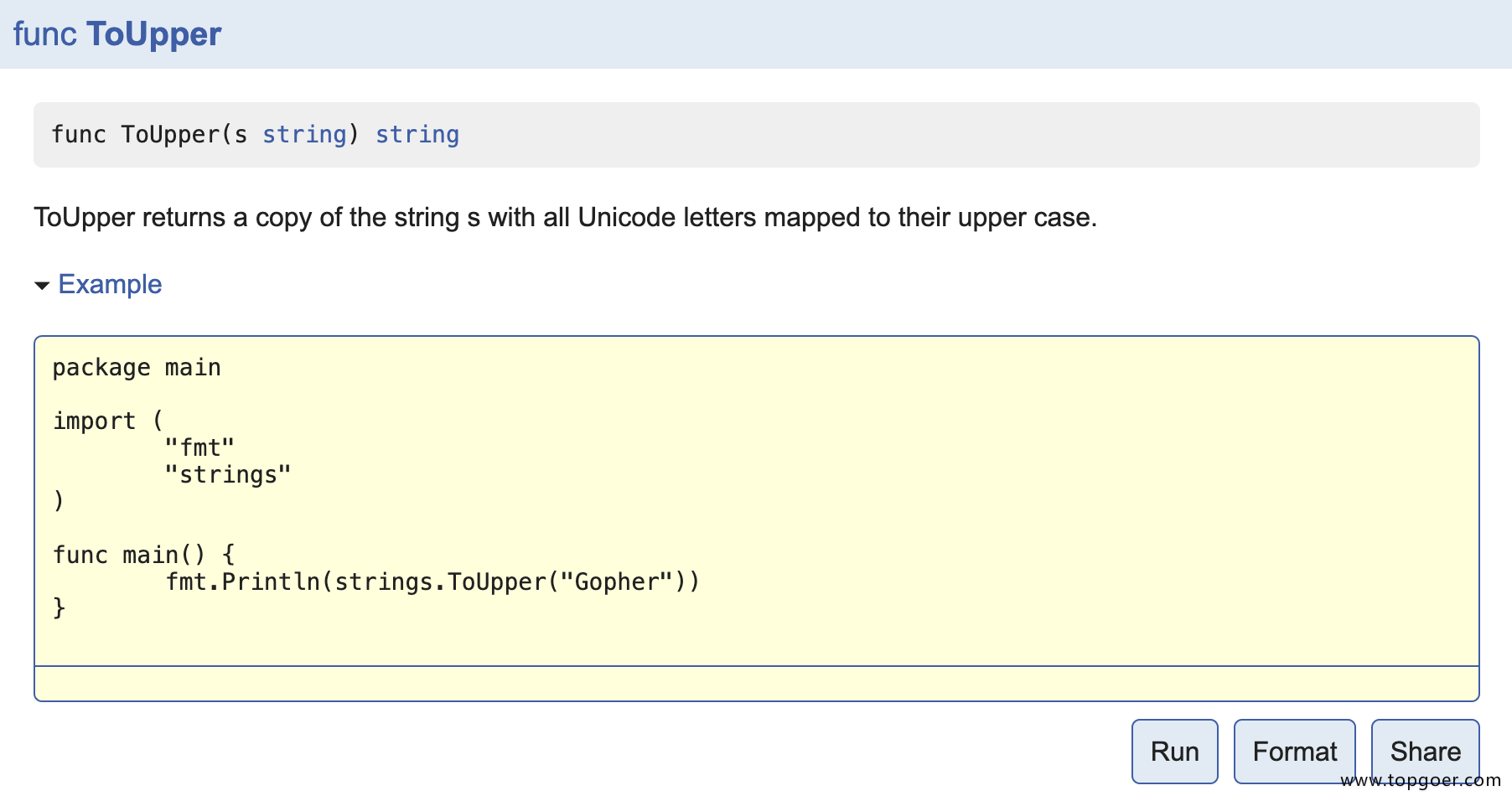
示例函数只要包含了// Output:也是可以通过go test运行的可执行测试。

split $ go test -run Example

PASS

ok github.com/pprof/studygo/code\_demo/test\_demo/split 0.006s

示例函数提供了可以直接运行的示例代码，可以直接在golang.org的godoc文档服务器上使用Go Playground运行示例代码。下图为strings.ToUpper函数在Playground的示例函数效果。



## 9. 压力测试

### 1.1.1. Go怎么写测试用例

开发程序其中很重要的一点是测试，我们如何保证代码的质量，如何保证每个函数是可运行，运行结果是正确的，又如何保证写出来的代码性能是好的，我们知道单元测试的重点在于发现程序设计或实现的逻辑错误，使问题及早暴露，便于问题的定位解决，而性能测试的重点在于发现程序设计上的一些问题，让线上的程序能够在高并发的情况下还能保持稳定。本小节将带着这一连串的问题来讲解Go语言中如何来实现单元测试和性能测试。

Go语言中自带有一个轻量级的测试框架testing和自带的go test命令来实现单元测试和性能测试，testing框架和其他语言中的测试框架类似，你可以基于这个框架写针对相应函数的测试用例，也可以基于该框架写相应的压力测试用例，那么接下来让我们一一来看一下怎么写。

另外建议安装gotests插件自动生成测试代码:

go get -u -v github.com/cweill/gotests/...

### 9.1.2. 如何编写测试用例

由于go test命令只能在一个相应的目录下执行所有文件，所以我们接下来新建一个项目目录gotest,这样我们所有的代码和测试代码都在这个目录下。

接下来我们在该目录下面创建两个文件：gotest.go和gotest\_test.go

gotest.go:这个文件里面我们是创建了一个包，里面有一个函数实现了除法运算:

package gotest

import (

"errors"

)

func Division(a, b float64) (float64, error) {

if b == 0 {

return 0, errors.New("除数不能为0")

}

return a / b, nil

}

gotest\_test.go:这是我们的单元测试文件，但是记住下面的这些原则：

文件名必须是\_test.go结尾的，这样在执行go test的时候才会执行到相应的代码

你必须import testing这个包

所有的测试用例函数必须是Test开头

测试用例会按照源代码中写的顺序依次执行

测试函数TestXxx()的参数是testing.T，我们可以使用该类型来记录错误或者是测试状态

测试格式：func TestXxx (t \*testing.T),Xxx部分可以为任意的字母数字的组合，但是首字母不能是小写字母[a-z]，例如Testintdiv是错误的函数名。

函数中通过调用testing.T的Error, Errorf, FailNow, Fatal, FatalIf方法，说明测试不通过，调用Log方法用来记录测试的信息。

下面是我们的测试用例的代码：

package gotest

import (

"testing"

)

func Test\_Division\_1(t \*testing.T) {

if i, e := Division(6, 2); i != 3 || e != nil { //try a unit test on function

t.Error("除法函数测试没通过") // 如果不是如预期的那么就报错

} else {

t.Log("第一个测试通过了") //记录一些你期望记录的信息

}

}

func Test\_Division\_2(t \*testing.T) {

t.Error("就是不通过")

}

我们在项目目录下面执行go test,就会显示如下信息：

--- FAIL: Test\_Division\_2 (0.00 seconds)

gotest\_test.go:16: 就是不通过

FAIL

exit status 1

FAIL gotest 0.013s

从这个结果显示测试没有通过，因为在第二个测试函数中我们写死了测试不通过的代码t.Error，那么我们的第一个函数执行的情况怎么样呢？默认情况下执行go test是不会显示测试通过的信息的，我们需要带上参数go test -v，这样就会显示如下信息：

=== RUN Test\_Division\_1

--- PASS: Test\_Division\_1 (0.00 seconds)

gotest\_test.go:11: 第一个测试通过了

=== RUN Test\_Division\_2

--- FAIL: Test\_Division\_2 (0.00 seconds)

gotest\_test.go:16: 就是不通过

FAIL

exit status 1

FAIL gotest 0.012s

上面的输出详细的展示了这个测试的过程，我们看到测试函数1Test\_Division\_1测试通过，而测试函数2Test\_Division\_2测试失败了，最后得出结论测试不通过。接下来我们把测试函数2修改成如下代码：

func Test\_Division\_2(t \*testing.T) {

if \_, e := Division(6, 0); e == nil { //try a unit test on function

t.Error("Division did not work as expected.") // 如果不是如预期的那么就报错

} else {

t.Log("one test passed.", e) //记录一些你期望记录的信息

}

}

然后我们执行go test -v，就显示如下信息，测试通过了：

=== RUN Test\_Division\_1

--- PASS: Test\_Division\_1 (0.00 seconds)

gotest\_test.go:11: 第一个测试通过了

=== RUN Test\_Division\_2

--- PASS: Test\_Division\_2 (0.00 seconds)

gotest\_test.go:20: one test passed. 除数不能为0

PASS

ok gotest 0.013s

### 9.1.3. 如何编写压力测试

压力测试用来检测函数(方法）的性能，和编写单元功能测试的方法类似,此处不再赘述，但需要注意以下几点：

压力测试用例必须遵循如下格式，其中XXX可以是任意字母数字的组合，但是首字母不能是小写字母

func BenchmarkXXX(b \*testing.B) { ... }

go test不会默认执行压力测试的函数，如果要执行压力测试需要带上参数-test.bench，语法:-test.bench="test\_name\_regex",例如go test -test.bench=".\*"表示测试全部的压力测试函数

在压力测试用例中,请记得在循环体内使用testing.B.N,以使测试可以正常的运行 文件名也必须以\_test.go结尾

下面我们新建一个压力测试文件webbench\_test.go，代码如下所示：

package gotest

import (

"testing"

)

func Benchmark\_Division(b \*testing.B) {

for i := 0; i < b.N; i++ { //use b.N for looping

Division(4, 5)

}

}

func Benchmark\_TimeConsumingFunction(b \*testing.B) {

b.StopTimer() //调用该函数停止压力测试的时间计数

//做一些初始化的工作,例如读取文件数据,数据库连接之类的,

//这样这些时间不影响我们测试函数本身的性能

b.StartTimer() //重新开始时间

for i := 0; i < b.N; i++ {

Division(4, 5)

}

}

我们执行命令go test webbench\_test.go -test.bench=".\*"，可以看到如下结果：

Benchmark\_Division-4 500000000 7.76 ns/op 456 B/op 14 allocs/op

Benchmark\_TimeConsumingFunction-4 500000000 7.80 ns/op 224 B/op 4 allocs/op

PASS

ok gotest 9.364s

上面的结果显示我们没有执行任何TestXXX的单元测试函数，显示的结果只执行了压力测试函数，第一条显示了Benchmark\_Division执行了500000000次，每次的执行平均时间是7.76纳秒，第二条显示了Benchmark\_TimeConsumingFunction执行了500000000，每次的平均执行时间是7.80纳秒。最后一条显示总共的执行时间。

### 9.1.4. 小结

通过上面对单元测试和压力测试的学习，我们可以看到testing包很轻量，编写单元测试和压力测试用例非常简单，配合内置的go test命令就可以非常方便的进行测试，这样在我们每次修改完代码,执行一下go test就可以简单的完成回归测试了。

# 方法

## 1. 方法定义

Golang 方法总是绑定对象实例，并隐式将实例作为第一实参 (receiver)。

• 只能为当前包内命名类型定义方法。

• 参数 receiver 可任意命名。如方法中未曾使用 ，可省略参数名。

• 参数 receiver 类型可以是 T 或 \*T。基类型 T 不能是接口或指针。

• 不支持方法重载，receiver 只是参数签名的组成部分。

• 可用实例 value 或 pointer 调用全部方法，编译器自动转换。

一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。

所有给定类型的方法属于该类型的方法集。

### 1.1.1. 方法定义：

func (recevier type) methodName(参数列表)(返回值列表){}

参数和返回值可以省略

package main

type Test struct{}

// 无参数、无返回值

func (t Test) method0() {

}

// 单参数、无返回值

func (t Test) method1(i int) {

}

// 多参数、无返回值

func (t Test) method2(x, y int) {

}

// 无参数、单返回值

func (t Test) method3() (i int) {

return

}

// 多参数、多返回值

func (t Test) method4(x, y int) (z int, err error) {

return

}

// 无参数、无返回值

func (t \*Test) method5() {

}

// 单参数、无返回值

func (t \*Test) method6(i int) {

}

// 多参数、无返回值

func (t \*Test) method7(x, y int) {

}

// 无参数、单返回值

func (t \*Test) method8() (i int) {

return

}

// 多参数、多返回值

func (t \*Test) method9(x, y int) (z int, err error) {

return

}

func main() {}

下面定义一个结构体类型和该类型的一个方法：

package main

import (

"fmt"

)

//结构体

type User struct {

Name string

Email string

}

//方法

func (u User) Notify() {

fmt.Printf("%v : %v \n", u.Name, u.Email)

}

func main() {

// 值类型调用方法

u1 := User{"golang", "golang@golang.com"}

u1.Notify()

// 指针类型调用方法

u2 := User{"go", "go@go.com"}

u3 := &u2

u3.Notify()

}

输出结果：

golang : golang@golang.com

go : go@go.com

解释： 首先我们定义了一个叫做 User 的结构体类型，然后定义了一个该类型的方法叫做 Notify，该方法的接受者是一个 User 类型的值。要调用 Notify 方法我们需要一个 User 类型的值或者指针。

在这个例子中当我们使用指针时，Go 调整和解引用指针使得调用可以被执行。注意，当接受者不是一个指针时，该方法操作对应接受者的值的副本(意思就是即使你使用了指针调用函数，但是函数的接受者是值类型，所以函数内部操作还是对副本的操作，而不是指针操作。

我们修改 Notify 方法，让它的接受者使用指针类型：

package main

import (

"fmt"

)

//结构体

type User struct {

Name string

Email string

}

//方法

func (u \*User) Notify() {

fmt.Printf("%v : %v \n", u.Name, u.Email)

}

func main() {

// 值类型调用方法

u1 := User{"golang", "golang@golang.com"}

u1.Notify()

// 指针类型调用方法

u2 := User{"go", "go@go.com"}

u3 := &u2

u3.Notify()

}

输出结果：

golang : golang@golang.com

go : go@go.com

注意：当接受者是指针时，即使用值类型调用那么函数内部也是对指针的操作。

方法不过是一种特殊的函数，只需将其还原，就知道 receiver T 和 \*T 的差别。

package main

import "fmt"

type Data struct {

x int

}

func (self Data) ValueTest() { // func ValueTest(self Data);

fmt.Printf("Value: %p\n", &self)

}

func (self \*Data) PointerTest() { // func PointerTest(self \*Data);

fmt.Printf("Pointer: %p\n", self)

}

func main() {

d := Data{}

p := &d

fmt.Printf("Data: %p\n", p)

d.ValueTest() // ValueTest(d)

d.PointerTest() // PointerTest(&d)

p.ValueTest() // ValueTest(\*p)

p.PointerTest() // PointerTest(p)

}

输出:

Data: 0xc42007c008

Value: 0xc42007c018

Pointer: 0xc42007c008

Value: 0xc42007c020

Pointer: 0xc42007c008

### 1.1.2. 普通函数与方法的区别

1.对于普通函数，接收者为值类型时，不能将指针类型的数据直接传递，反之亦然。

2.对于方法（如struct的方法），接收者为值类型时，可以直接用指针类型的变量调用方法，反过来同样也可以。

package main

//普通函数与方法的区别（在接收者分别为值类型和指针类型的时候）

import (

"fmt"

)

//1.普通函数

//接收值类型参数的函数

func valueIntTest(a int) int {

return a + 10

}

//接收指针类型参数的函数

func pointerIntTest(a \*int) int {

return \*a + 10

}

func structTestValue() {

a := 2

fmt.Println("valueIntTest:", valueIntTest(a))

//函数的参数为值类型，则不能直接将指针作为参数传递

//fmt.Println("valueIntTest:", valueIntTest(&a))

//compile error: cannot use &a (type \*int) as type int in function argument

b := 5

fmt.Println("pointerIntTest:", pointerIntTest(&b))

//同样，当函数的参数为指针类型时，也不能直接将值类型作为参数传递

//fmt.Println("pointerIntTest:", pointerIntTest(b))

//compile error:cannot use b (type int) as type \*int in function argument

}

//2.方法

type PersonD struct {

id int

name string

}

//接收者为值类型

func (p PersonD) valueShowName() {

fmt.Println(p.name)

}

//接收者为指针类型

func (p \*PersonD) pointShowName() {

fmt.Println(p.name)

}

func structTestFunc() {

//值类型调用方法

personValue := PersonD{101, "hello world"}

personValue.valueShowName()

personValue.pointShowName()

//指针类型调用方法

personPointer := &PersonD{102, "hello golang"}

personPointer.valueShowName()

personPointer.pointShowName()

//与普通函数不同，接收者为指针类型和值类型的方法，指针类型和值类型的变量均可相互调用

}

func main() {

structTestValue()

structTestFunc()

}

输出结果：

valueIntTest: 12

pointerIntTest: 15

hello world

hello world

hello golang

hello golang

## 2. 匿名字段

Golang匿名字段 ：可以像字段成员那样访问匿名字段方法，编译器负责查找。

package main

import "fmt"

type User struct {

id int

name string

}

type Manager struct {

User

}

func (self \*User) ToString() string { // receiver = &(Manager.User)

return fmt.Sprintf("User: %p, %v", self, self)

}

func main() {

m := Manager{User{1, "Tom"}}

fmt.Printf("Manager: %p\n", &m)

fmt.Println(m.ToString())

}

输出结果:

Manager: 0xc42000a060

User: 0xc42000a060, &{1 Tom}

通过匿名字段，可获得和继承类似的复用能力。依据编译器查找次序，只需在外层定义同名方法，就可以实现 "override"。

package main

import "fmt"

type User struct {

id int

name string

}

type Manager struct {

User

title string

}

func (self \*User) ToString() string {

return fmt.Sprintf("User: %p, %v", self, self)

}

func (self \*Manager) ToString() string {

return fmt.Sprintf("Manager: %p, %v", self, self)

}

func main() {

m := Manager{User{1, "Tom"}, "Administrator"}

fmt.Println(m.ToString())

fmt.Println(m.User.ToString())

}

输出结果:

Manager: 0xc420074180, &{{1 Tom} Administrator}

User: 0xc420074180, &{1 Tom}

## 3. 方法集

Golang方法集 ：每个类型都有与之关联的方法集，这会影响到接口实现规则。

• 类型 T 方法集包含全部 receiver T 方法。

• 类型 \*T 方法集包含全部 receiver T + \*T 方法。

• 如类型 S 包含匿名字段 T，则 S 和 \*S 方法集包含 T 方法。

• 如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 T + \*T 方法。

• 不管嵌入 T 或 \*T，\*S 方法集总是包含 T + \*T 方法。

用实例 value 和 pointer 调用方法 (含匿名字段) 不受方法集约束，编译器总是查找全部方法，并自动转换 receiver 实参。

Go 语言中内部类型方法集提升的规则：

类型 T 方法集包含全部 receiver T 方法。

package main

import (

"fmt"

)

type T struct {

int

}

func (t T) test() {

fmt.Println("类型 T 方法集包含全部 receiver T 方法。")

}

func main() {

t1 := T{1}

fmt.Printf("t1 is : %v\n", t1)

t1.test()

}

输出结果：

t1 is : {1}

类型 T 方法集包含全部 receiver T 方法。

类型 \*T 方法集包含全部 receiver T + \*T 方法。

package main

import (

"fmt"

)

type T struct {

int

}

func (t T) testT() {

fmt.Println("类型 \*T 方法集包含全部 receiver T 方法。")

}

func (t \*T) testP() {

fmt.Println("类型 \*T 方法集包含全部 receiver \*T 方法。")

}

func main() {

t1 := T{1}

t2 := &t1

fmt.Printf("t2 is : %v\n", t2)

t2.testT()

t2.testP()

}

输出结果：

t2 is : &{1}

类型 \*T 方法集包含全部 receiver T 方法。

类型 \*T 方法集包含全部 receiver \*T 方法。

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：

如类型 S 包含匿名字段 T，则 S 和 \*S 方法集包含 T 方法。

这条规则说的是当我们嵌入一个类型，嵌入类型的接受者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。

package main

import (

"fmt"

)

type S struct {

T

}

type T struct {

int

}

func (t T) testT() {

fmt.Println("如类型 S 包含匿名字段 T，则 S 和 \*S 方法集包含 T 方法。")

}

func main() {

s1 := S{T{1}}

s2 := &s1

fmt.Printf("s1 is : %v\n", s1)

s1.testT()

fmt.Printf("s2 is : %v\n", s2)

s2.testT()

}

输出结果：

s1 is : {{1}}

如类型 S 包含匿名字段 T，则 S 和 \*S 方法集包含 T 方法。

s2 is : &{{1}}

如类型 S 包含匿名字段 T，则 S 和 \*S 方法集包含 T 方法。

如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 T + \*T 方法。

这条规则说的是当我们嵌入一个类型的指针，嵌入类型的接受者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。

package main

import (

"fmt"

)

type S struct {

T

}

type T struct {

int

}

func (t T) testT() {

fmt.Println("如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 T 方法")

}

func (t \*T) testP() {

fmt.Println("如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 \*T 方法")

}

func main() {

s1 := S{T{1}}

s2 := &s1

fmt.Printf("s1 is : %v\n", s1)

s1.testT()

s1.testP()

fmt.Printf("s2 is : %v\n", s2)

s2.testT()

s2.testP()

}

输出结果：

s1 is : {{1}}

如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 T 方法

如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 \*T 方法

s2 is : &{{1}}

如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 T 方法

如类型 S 包含匿名字段 \*T，则 S 和 \*S 方法集包含 \*T 方法

## 4. 表达式

Golang 表达式 ：根据调用者不同，方法分为两种表现形式:

instance.method(args...) ---> <type>.func(instance, args...)

前者称为 method value，后者 method expression。

两者都可像普通函数那样赋值和传参，区别在于 method value 绑定实例，而 method expression 则须显式传参。

package main

import "fmt"

type User struct {

id int

name string

}

func (self \*User) Test() {

fmt.Printf("%p, %v\n", self, self)

}

func main() {

u := User{1, "Tom"}

u.Test()

mValue := u.Test

mValue() // 隐式传递 receiver

mExpression := (\*User).Test

mExpression(&u) // 显式传递 receiver

}

输出结果:

0xc42000a060, &{1 Tom}

0xc42000a060, &{1 Tom}

0xc42000a060, &{1 Tom}

需要注意，method value 会复制 receiver。

package main

import "fmt"

type User struct {

id int

name string

}

func (self User) Test() {

fmt.Println(self)

}

func main() {

u := User{1, "Tom"}

mValue := u.Test // 立即复制 receiver，因为不是指针类型，不受后续修改影响。

u.id, u.name = 2, "Jack"

u.Test()

mValue()

}

输出结果

{2 Jack}

{1 Tom}

在汇编层面，method value 和闭包的实现方式相同，实际返回 FuncVal 类型对象。

FuncVal { method\_address, receiver\_copy }

可依据方法集转换 method expression，注意 receiver 类型的差异。

package main

import "fmt"

type User struct {

id int

name string

}

func (self \*User) TestPointer() {

fmt.Printf("TestPointer: %p, %v\n", self, self)

}

func (self User) TestValue() {

fmt.Printf("TestValue: %p, %v\n", &self, self)

}

func main() {

u := User{1, "Tom"}

fmt.Printf("User: %p, %v\n", &u, u)

mv := User.TestValue

mv(u)

mp := (\*User).TestPointer

mp(&u)

mp2 := (\*User).TestValue // \*User 方法集包含 TestValue。签名变为 func TestValue(self \*User)。实际依然是 receiver value copy。

mp2(&u)

}

输出:

User: 0xc42000a060, {1 Tom}

TestValue: 0xc42000a0a0, {1 Tom}

TestPointer: 0xc42000a060, &{1 Tom}

TestValue: 0xc42000a100, {1 Tom}

将方法 "还原" 成函数，就容易理解下面的代码了。

package main

type Data struct{}

func (Data) TestValue() {}

func (\*Data) TestPointer() {}

func main() {

var p \*Data = nil

p.TestPointer()

(\*Data)(nil).TestPointer() // method value

(\*Data).TestPointer(nil) // method expression

// p.TestValue() // invalid memory address or nil pointer dereference

// (Data)(nil).TestValue() // cannot convert nil to type Data

// Data.TestValue(nil) // cannot use nil as type Data in function argument

}

## 5. 自定义error

### 5.1. 抛异常和处理异常

#### 5.1.1. 系统抛

package main

import "fmt"

// 系统抛

func test01() {

a := [5]int{0, 1, 2, 3, 4}

a[1] = 123

fmt.Println(a)

//a[10] = 11

index := 10

a[index] = 10

fmt.Println(a)

}

func getCircleArea(radius float32) (area float32) {

if radius < 0 {

// 自己抛

panic("半径不能为负")

}

return 3.14 \* radius \* radius

}

func test02() {

getCircleArea(-5)

}

//

func test03() {

// 延时执行匿名函数

// 延时到何时？（1）程序正常结束 （2）发生异常时

defer func() {

// recover() 复活 恢复

// 会返回程序为什么挂了

if err := recover(); err != nil {

fmt.Println(err)

}

}()

getCircleArea(-5)

fmt.Println("这里有没有执行")

}

func test04() {

test03()

fmt.Println("test04")

}

func main() {

test04()

}

#### 5.1.2. 返回异常

package main

import (

"errors"

"fmt"

)

func getCircleArea(radius float32) (area float32, err error) {

if radius < 0 {

// 构建个异常对象

err = errors.New("半径不能为负")

return

}

area = 3.14 \* radius \* radius

return

}

func main() {

area, err := getCircleArea(-5)

if err != nil {

fmt.Println(err)

} else {

fmt.Println(area)

}

}

#### 5.1.3. 自定义error：

package main

import (

"fmt"

"os"

"time"

)

type PathError struct {

path string

op string

createTime string

message string

}

func (p \*PathError) Error() string {

return fmt.Sprintf("path=%s \nop=%s \ncreateTime=%s \nmessage=%s", p.path,

p.op, p.createTime, p.message)

}

func Open(filename string) error {

file, err := os.Open(filename)

if err != nil {

return &PathError{

path: filename,

op: "read",

message: err.Error(),

createTime: fmt.Sprintf("%v", time.Now()),

}

}

defer file.Close()

return nil

}

func main() {

err := Open("/Users/5lmh/Desktop/go/src/test.txt")

switch v := err.(type) {

case \*PathError:

fmt.Println("get path error,", v)

default:

}

}

输出结果：

get path error, path=/Users/pprof/Desktop/go/src/test.txt

op=read

createTime=2018-04-05 11:25:17.331915 +0800 CST m=+0.000441790

message=open /Users/pprof/Desktop/go/src/test.txt: no such file or directory

# 面向对象

## 1. 匿名字段

**go支持只提供类型而不写字段名的方式，也就是匿名字段，也称为嵌入字段**

package main

import "fmt"

//λ go支持只提供类型而不写字段名的方式，也就是匿名字段，也称为嵌入字段

//人

type Person struct {

name string

sex string

age int

}

type Student struct {

Person

id int

addr string

}

func main() {

// 初始化

s1 := Student{Person{"5lmh", "man", 20}, 1, "bj"}

fmt.Println(s1)

s2 := Student{Person: Person{"5lmh", "man", 20}}

fmt.Println(s2)

s3 := Student{Person: Person{name: "5lmh"}}

fmt.Println(s3)

}

输出结果：

{{5lmh man 20} 1 bj}

{{5lmh man 20} 0 }

{{5lmh 0} 0 }

**同名字段的情况**

package main

import "fmt"

//人

type Person struct {

name string

sex string

age int

}

type Student struct {

Person

id int

addr string

//同名字段

name string

}

func main() {

var s Student

// 给自己字段赋值了

s.name = "5lmh"

fmt.Println(s)

// 若给父类同名字段赋值，如下

s.Person.name = "枯藤"

fmt.Println(s)

}

输出结果：

{{ 0} 0 5lmh}

{{枯藤 0} 0 5lmh}

**所有的内置类型和自定义类型都是可以作为匿名字段去使用**

package main

import "fmt"

//人

type Person struct {

name string

sex string

age int

}

// 自定义类型

type mystr string

// 学生

type Student struct {

Person

int

mystr

}

func main() {

s1 := Student{Person{"5lmh", "man", 18}, 1, "bj"}

fmt.Println(s1)

}

输出结果：

{{5lmh man 18} 1 bj}

**指针类型匿名字段**

package main

import "fmt"

//人

type Person struct {

name string

sex string

age int

}

// 学生

type Student struct {

\*Person

id int

addr string

}

func main() {

s1 := Student{&Person{"5lmh", "man", 18}, 1, "bj"}

fmt.Println(s1)

fmt.Println(s1.name)

fmt.Println(s1.Person.name)

}

输出结果：

{0xc00005c360 1 bj}

zs

zs

## 2. 接口

接口（interface）定义了一个对象的行为规范，只定义规范不实现，由具体的对象来实现规范的细节。

### 2.1. 接口

#### 2.1.1. 接口类型

在Go语言中接口（interface）是一种类型，一种抽象的类型。

interface是一组method的集合，是duck-type programming的一种体现。接口做的事情就像是定义一个协议（规则），只要一台机器有洗衣服和甩干的功能，我就称它为洗衣机。不关心属性（数据），只关心行为（方法）。

为了保护你的Go语言职业生涯，请牢记接口（interface）是一种类型。

#### 2.1.2. 为什么要使用接口

type Cat struct{}

func (c Cat) Say() string { return "喵喵喵" }

type Dog struct{}

func (d Dog) Say() string { return "汪汪汪" }

func main() {

c := Cat{}

fmt.Println("猫:", c.Say())

d := Dog{}

fmt.Println("狗:", d.Say())

}

上面的代码中定义了猫和狗，然后它们都会叫，你会发现main函数中明显有重复的代码，如果我们后续再加上猪、青蛙等动物的话，我们的代码还会一直重复下去。那我们能不能把它们当成“能叫的动物”来处理呢？

像类似的例子在我们编程过程中会经常遇到：

比如一个网上商城可能使用支付宝、微信、银联等方式去在线支付，我们能不能把它们当成“支付方式”来处理呢？

比如三角形，四边形，圆形都能计算周长和面积，我们能不能把它们当成“图形”来处理呢？

比如销售、行政、程序员都能计算月薪，我们能不能把他们当成“员工”来处理呢？

Go语言中为了解决类似上面的问题，就设计了接口这个概念。接口区别于我们之前所有的具体类型，接口是一种抽象的类型。当你看到一个接口类型的值时，你不知道它是什么，唯一知道的是通过它的方法能做什么。

#### 2.1.3. 接口的定义

Go语言提倡面向接口编程。

接口是一个或多个方法签名的集合。

任何类型的方法集中只要拥有该接口'对应的全部方法'签名。

就表示它 "实现" 了该接口，无须在该类型上显式声明实现了哪个接口。

这称为Structural Typing。

所谓对应方法，是指有相同名称、参数列表 (不包括参数名) 以及返回值。

当然，该类型还可以有其他方法。

接口只有方法声明，没有实现，没有数据字段。

接口可以匿名嵌入其他接口，或嵌入到结构中。

对象赋值给接口时，会发生拷贝，而接口内部存储的是指向这个复制品的指针，既无法修改复制品的状态，也无法获取指针。

只有当接口存储的类型和对象都为nil时，接口才等于nil。

接口调用不会做receiver的自动转换。

接口同样支持匿名字段方法。

接口也可实现类似OOP中的多态。

空接口可以作为任何类型数据的容器。

一个类型可实现多个接口。

接口命名习惯以 er 结尾。

每个接口由数个方法组成，接口的定义格式如下：

type 接口类型名 interface{

方法名1( 参数列表1 ) 返回值列表1

方法名2( 参数列表2 ) 返回值列表2

…

}

其中：

1.接口名：使用type将接口定义为自定义的类型名。Go语言的接口在命名时，一般会在单词后面添加er，如有写操作的接口叫Writer，有字符串功能的接口叫Stringer等。接口名最好要能突出该接口的类型含义。

2.方法名：当方法名首字母是大写且这个接口类型名首字母也是大写时，这个方法可以被接口所在的包（package）之外的代码访问。

3.参数列表、返回值列表：参数列表和返回值列表中的参数变量名可以省略。

举个例子：

type writer interface{

Write([]byte) error

}

当你看到这个接口类型的值时，你不知道它是什么，唯一知道的就是可以通过它的Write方法来做一些事情。

#### 2.1.4. 实现接口的条件

一个对象只要全部实现了接口中的方法，那么就实现了这个接口。换句话说，接口就是一个需要实现的方法列表。

我们来定义一个Sayer接口：

// Sayer 接口

type Sayer interface {

say()

}

定义dog和cat两个结构体：

type dog struct {}

type cat struct {}

因为Sayer接口里只有一个say方法，所以我们只需要给dog和cat 分别实现say方法就可以实现Sayer接口了。

// dog实现了Sayer接口

func (d dog) say() {

fmt.Println("汪汪汪")

}

// cat实现了Sayer接口

func (c cat) say() {

fmt.Println("喵喵喵")

}

接口的实现就是这么简单，只要实现了接口中的所有方法，就实现了这个接口。

#### 2.1.5. 接口类型变量

那实现了接口有什么用呢？

接口类型变量能够存储所有实现了该接口的实例。 例如上面的示例中，Sayer类型的变量能够存储dog和cat类型的变量。

func main() {

var x Sayer // 声明一个Sayer类型的变量x

a := cat{} // 实例化一个cat

b := dog{} // 实例化一个dog

x = a // 可以把cat实例直接赋值给x

x.say() // 喵喵喵

x = b // 可以把dog实例直接赋值给x

x.say() // 汪汪汪

}

#### 2.1.6. 值接收者和指针接收者实现接口的区别

使用值接收者实现接口和使用指针接收者实现接口有什么区别呢？接下来我们通过一个例子看一下其中的区别。

我们有一个Mover接口和一个dog结构体。

type Mover interface {

move()

}

type dog struct {}

#### 2.1.7. 值接收者实现接口

func (d dog) move() {

fmt.Println("狗会动")

}

此时实现接口的是dog类型：

func main() {

var x Mover

var wangcai = dog{} // 旺财是dog类型

x = wangcai // x可以接收dog类型

var fugui = &dog{} // 富贵是\*dog类型

x = fugui // x可以接收\*dog类型

x.move()

}

从上面的代码中我们可以发现，使用值接收者实现接口之后，不管是dog结构体还是结构体指针\*dog类型的变量都可以赋值给该接口变量。因为Go语言中有对指针类型变量求值的语法糖，dog指针fugui内部会自动求值\*fugui。

#### 2.1.8. 指针接收者实现接口

同样的代码我们再来测试一下使用指针接收者有什么区别：

func (d \*dog) move() {

fmt.Println("狗会动")

}

func main() {

var x Mover

var wangcai = dog{} // 旺财是dog类型

x = wangcai // x不可以接收dog类型

var fugui = &dog{} // 富贵是\*dog类型

x = fugui // x可以接收\*dog类型

}

此时实现Mover接口的是\*dog类型，所以不能给x传入dog类型的wangcai，此时x只能存储\*dog类型的值。

#### 2.1.9. 下面的代码是一个比较好的面试题

请问下面的代码是否能通过编译？

type People interface {

Speak(string) string

}

type Student struct{}

func (stu \*Stduent) Speak(think string) (talk string) {

if think == "sb" {

talk = "你是个大帅比"

} else {

talk = "您好"

}

return

}

func main() {

var peo People = Student{}

think := "bitch"

fmt.Println(peo.Speak(think))

}

### 2.2. 类型与接口的关系

#### 2.2.1. 一个类型实现多个接口

一个类型可以同时实现多个接口，而接口间彼此独立，不知道对方的实现。 例如，狗可以叫，也可以动。我们就分别定义Sayer接口和Mover接口，如下： Mover接口。

// Sayer 接口

type Sayer interface {

say()

}

// Mover 接口

type Mover interface {

move()

}

dog既可以实现Sayer接口，也可以实现Mover接口。

type dog struct {

name string

}

// 实现Sayer接口

func (d dog) say() {

fmt.Printf("%s会叫汪汪汪\n", d.name)

}

// 实现Mover接口

func (d dog) move() {

fmt.Printf("%s会动\n", d.name)

}

func main() {

var x Sayer

var y Mover

var a = dog{name: "旺财"}

x = a

y = a

x.say()

y.move()

}

#### 2.2.2. 多个类型实现同一接口

Go语言中不同的类型还可以实现同一接口 首先我们定义一个Mover接口，它要求必须由一个move方法。

// Mover 接口

type Mover interface {

move()

}

例如狗可以动，汽车也可以动，可以使用如下代码实现这个关系：

type dog struct {

name string

}

type car struct {

brand string

}

// dog类型实现Mover接口

func (d dog) move() {

fmt.Printf("%s会跑\n", d.name)

}

// car类型实现Mover接口

func (c car) move() {

fmt.Printf("%s速度70迈\n", c.brand)

}

这个时候我们在代码中就可以把狗和汽车当成一个会动的物体来处理了，不再需要关注它们具体是什么，只需要调用它们的move方法就可以了。

func main() {

var x Mover

var a = dog{name: "旺财"}

var b = car{brand: "保时捷"}

x = a

x.move()

x = b

x.move()

}

上面的代码执行结果如下：

旺财会跑

保时捷速度70迈

并且一个接口的方法，不一定需要由一个类型完全实现，接口的方法可以通过在类型中嵌入其他类型或者结构体来实现。

// WashingMachine 洗衣机

type WashingMachine interface {

wash()

dry()

}

// 甩干器

type dryer struct{}

// 实现WashingMachine接口的dry()方法

func (d dryer) dry() {

fmt.Println("甩一甩")

}

// 海尔洗衣机

type haier struct {

dryer //嵌入甩干器

}

// 实现WashingMachine接口的wash()方法

func (h haier) wash() {

fmt.Println("洗刷刷")

}

#### 2.2.3. 接口嵌套

接口与接口间可以通过嵌套创造出新的接口。

// Sayer 接口

type Sayer interface {

say()

}

// Mover 接口

type Mover interface {

move()

}

// 接口嵌套

type animal interface {

Sayer

Mover

}

嵌套得到的接口的使用与普通接口一样，这里我们让cat实现animal接口：

type cat struct {

name string

}

func (c cat) say() {

fmt.Println("喵喵喵")

}

func (c cat) move() {

fmt.Println("猫会动")

}

func main() {

var x animal

x = cat{name: "花花"}

x.move()

x.say()

}

### 2.3. 空接口

#### 2.3.1. 空接口的定义

空接口是指没有定义任何方法的接口。因此任何类型都实现了空接口。

空接口类型的变量可以存储任意类型的变量。

func main() {

// 定义一个空接口x

var x interface{}

s := "pprof.cn"

x = s

fmt.Printf("type:%T value:%v\n", x, x)

i := 100

x = i

fmt.Printf("type:%T value:%v\n", x, x)

b := true

x = b

fmt.Printf("type:%T value:%v\n", x, x)

}

#### 2.3.2. 空接口的应用

##### 空接口作为函数的参数

使用空接口实现可以接收任意类型的函数参数。

// 空接口作为函数参数

func show(a interface{}) {

fmt.Printf("type:%T value:%v\n", a, a)

}

##### 空接口作为map的值

使用空接口实现可以保存任意值的字典。

// 空接口作为map值

var studentInfo = make(map[string]interface{})

studentInfo["name"] = "李白"

studentInfo["age"] = 18

studentInfo["married"] = false

fmt.Println(studentInfo)

#### 2.3.3. 类型断言

空接口可以存储任意类型的值，那我们如何获取其存储的具体数据呢？

##### 接口值

一个接口的值（简称接口值）是由一个具体类型和具体类型的值两部分组成的。这两部分分别称为接口的动态类型和动态值。

我们来看一个具体的例子：

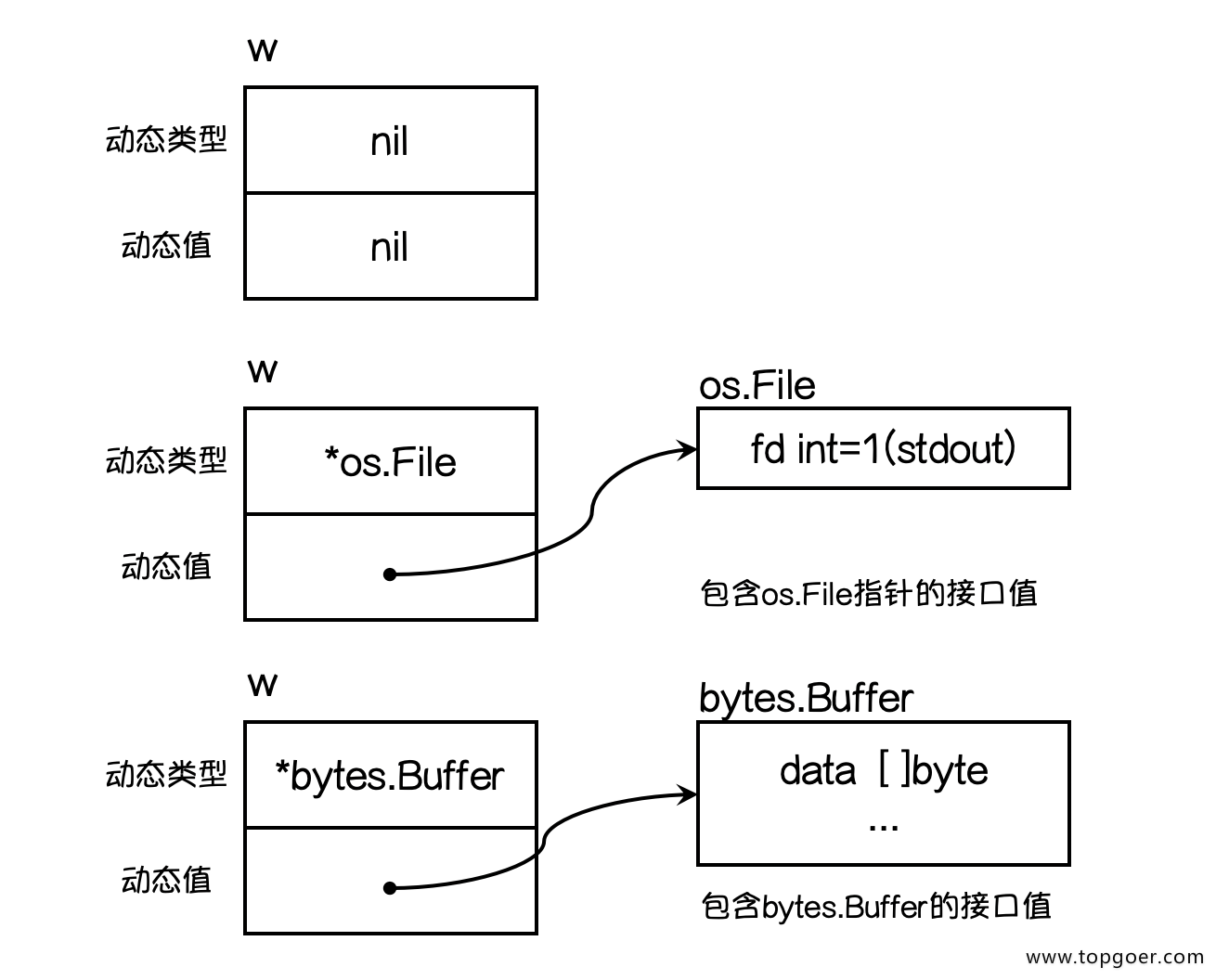
var w io.Writer

w = os.Stdout

w = new(bytes.Buffer)

w = nil

请看下图分解：



想要判断空接口中的值这个时候就可以使用类型断言，其语法格式：

x.(T)

其中：

x：表示类型为interface{}的变量

T：表示断言x可能是的类型。

该语法返回两个参数，第一个参数是x转化为T类型后的变量，第二个值是一个布尔值，若为true则表示断言成功，为false则表示断言失败。

举个例子：

func main() {

var x interface{}

x = "pprof.cn"

v, ok := x.(string)

if ok {

fmt.Println(v)

} else {

fmt.Println("类型断言失败")

}

}

上面的示例中如果要断言多次就需要写多个if判断，这个时候我们可以使用switch语句来实现：

func justifyType(x interface{}) {

switch v := x.(type) {

case string:

fmt.Printf("x is a string，value is %v\n", v)

case int:

fmt.Printf("x is a int is %v\n", v)

case bool:

fmt.Printf("x is a bool is %v\n", v)

default:

fmt.Println("unsupport type！")

}

}

因为空接口可以存储任意类型值的特点，所以空接口在Go语言中的使用十分广泛。

关于接口需要注意的是，只有当有两个或两个以上的具体类型必须以相同的方式进行处理时才需要定义接口。不要为了接口而写接口，那样只会增加不必要的抽象，导致不必要的运行时损耗。