目　录

前景
开发环境
- Go的安装
- 配置GOPATH
- 编辑器
- Git安装
- 第一个go程序

Go基础
- Go语言的主要特征
- Golang内置类型和函数
- Init函数和main函数
- 命令
- 运算符
- 下划线
- 变量和常量
- 基本类型
- 数组Array
- 切片Slice
- Slice底层实现
- 指针
- Map
- Map实现原理
- 结构体

流程控制
- 条件语句if
- 条件语句switch
- 条件语句select
- 循环语句for
- 循环语句range
- 循环控制Goto、Break、Continue

函数

- 函数定义
- 参数
- 返回值
- 匿名函数
- 闭包、递归
- 延迟调用（defer）
- 异常处理
- 单元测试
- 压力测试

方法
- 方法定义
- 匿名字段
- 方法集
- 表达式
- 自定义error

面向对象
- 匿名字段
- 接口

网络编程
- 互联网协议介绍
- socket编程
  - socket图解
  - TCP编程
  - UDP编程
  - TCP黏包
- http编程
- WebSocket编程

并发编程
- 并发介绍
- Goroutine
- runtime包
- Channel
- Goroutine池

- 定时器
- select
- 并发安全和锁
- Sync
- 原子操作(atomic包)
- GMP 原理与调度
- 爬虫小案例

数据操作

- go操作MySQL
  - go操作MySQL
  - Insert操作
  - Select操作
  - Update操作
  - Delete操作
  - MySQL事务
- go操作Redis
  - Redis介绍
  - 链接Redis
  - String类型Set、Get操作
  - String批量操作
  - 设置过期时间
  - List队列操作
  - Hash表
  - Redis连接池
- go操作ETCD
  - ETCD介绍
  - 操作ETCD
- zookeeper
  - 基本操作测试
  - 简单的分布式server
  - Zookeeper命令行使用
- go操作kafka
  - Kafka介绍

- Kafka深层介绍
- Kafka的安装
- 操作Kafka
- go操作RabbitMQ
  - RabbitMQ介绍
  - RabbitMQ安装
  - Simple模式
  - Work模式
  - Publish模式
  - Routing模式
  - Topic模式
- go操作ElasticSearch
  - ElasticSearch介绍
  - Elasticsearch安装
  - Kibana安装
  - 操作ElasticSearch
- NSQ
  - 安装
  - 生产者
  - 消费者
- GORM
- xorm
- go操作memcached

常用标准库
- fmt
- Time
- Flag
- Log
- IO操作
- Strconv
- Template
- Http
- Context

        - 数据格式
        - 反射
        - 文件操作
        - go module
        - String
    - beego框架
    - gin框架
    - Iris框架
    - Echo框架
    - 微服务
        - 认识微服务
        - 微服务生态
        - 微服务详解
        - RPC
        - RPC系统文档
        - Raft
        - gRPC
            - 安装
            - gRPC简介
            - Protobuf⇢Go转换
            - Protobuf语法
            - 小案例
            - OpenSSL安装
            - 认证
            - 拦截器
            - 内置Trace
            - HTTP网关
        - Go Micro入门
        - Go Micro接口详解
        - Go Micro文档1.x
        - Go Micro文档2.x
    - Go高级
    - 插件库

- 项目
  - github库地址
  - TCP扫描器
  - 定时任务
    - cron
    - gocron
  - 基于角色的访问控制框架
  - uuid
  - 支付宝支付
  - 微信支付
  - 微信公众号开发
  - 爬虫小案例
  - 千万数据过滤
    - 需求
    - 读入数据
    - 数据清洗
    - 省份划分
  - go-admin
  - go-vue-admin
  - go-admin
  - log
    - Logger
    - Zap Logger
    - 日志切割归档
  - 聊天室小案例
  - 性能压测工具wrk
  - gcc安装
  - gim即时通讯
- 开源仓库
- 其他
  - Golang 58个坑
  - 资料下载
  - 零碎知识点

学习路线图
面试题
go中文标准文档
关于

前景

*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**

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。
```

Go语言简单易学

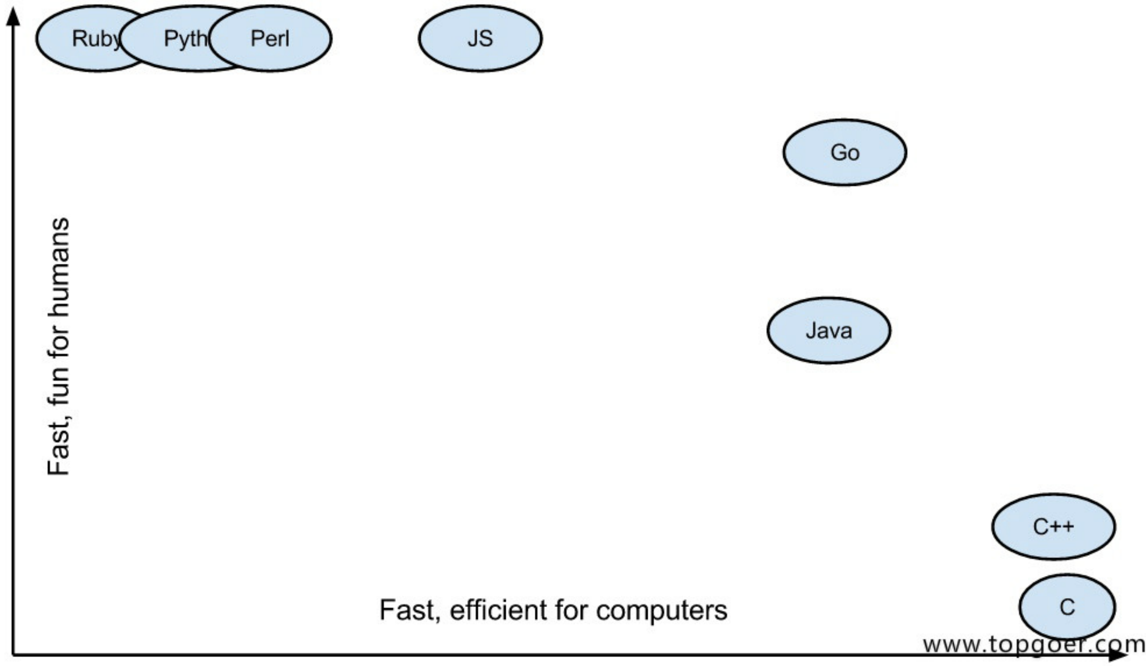
语法简洁

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）。

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

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

学习Go语言的前景

目前Go语言已经□泛应用于人工智能、云计算开发、容器虚拟化、□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。

开发环境

Go的安装

配置GOPATH

编辑器

Git安装

第一个go程序

Go的安装

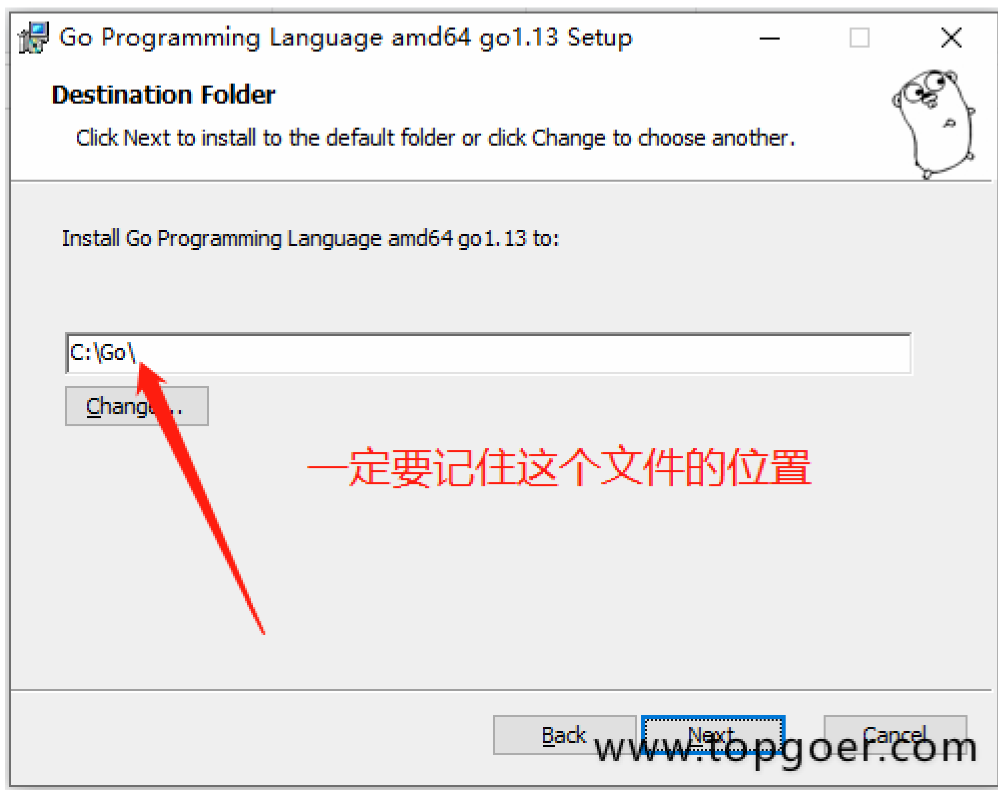
下载地址

Go官网下载地址：https://golang.org/dl/ (打开有点慢)

Windows安装

双击文件

一定要记住这个文件的位置后面还有用



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
```

Mac下安装

没有Mac没有试过不知道怎么安装

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。

配置GOPATH

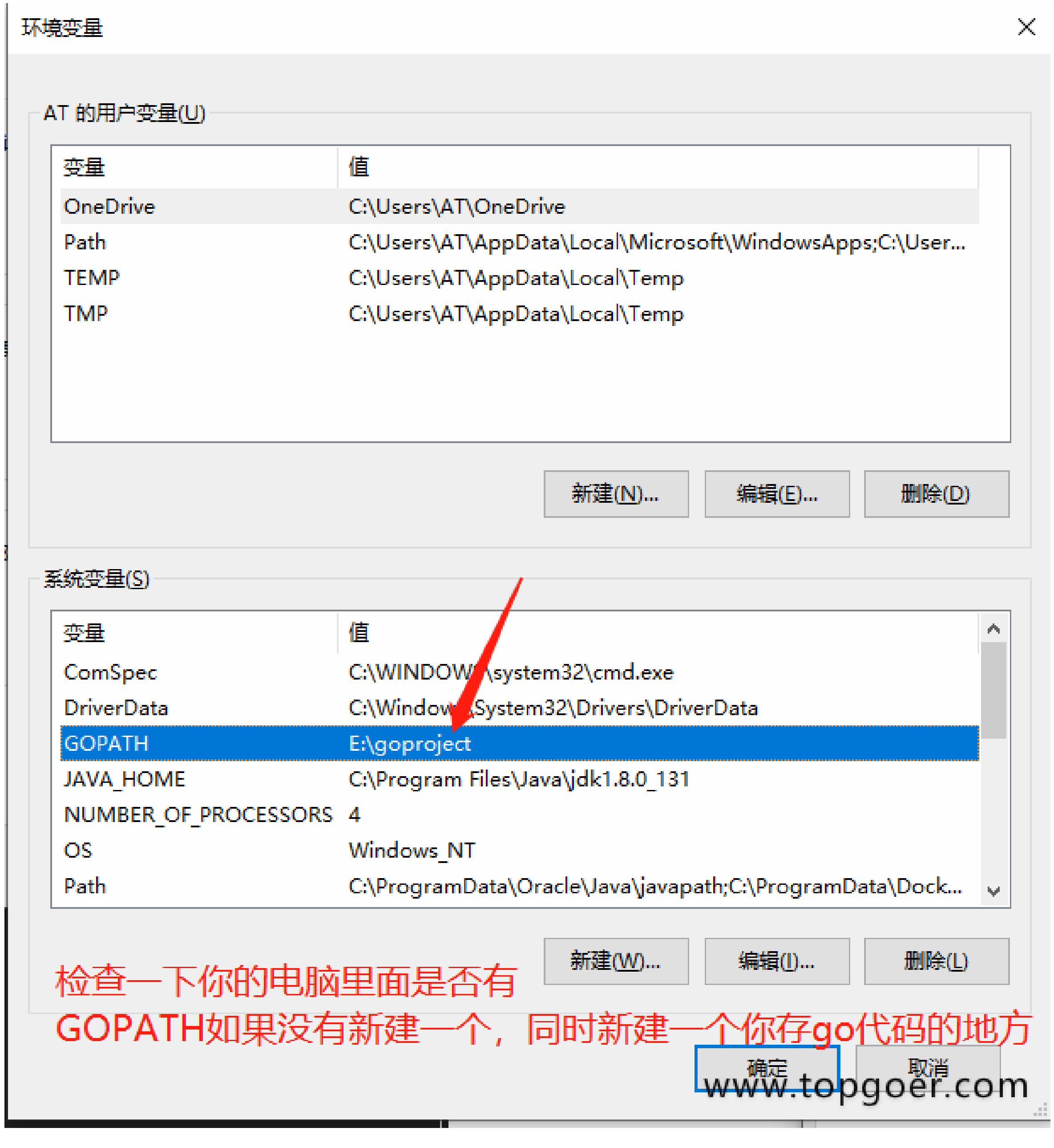
`GOPATH` 是一个环境变量，用来表明你写的 `go` 项目的存放路径

`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：

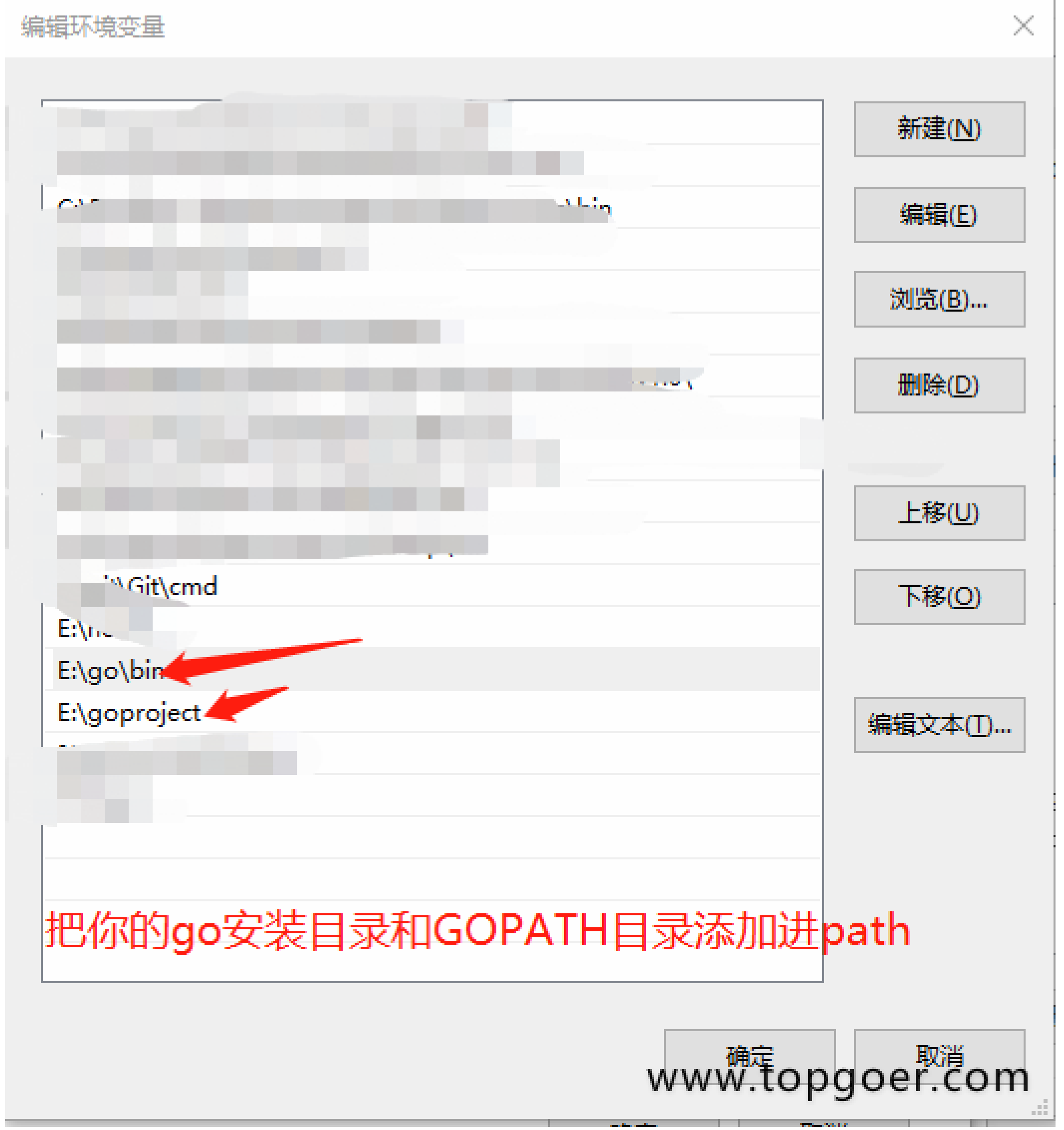
1.我的电脑->属性->高级系统设置

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

配置GOPATH



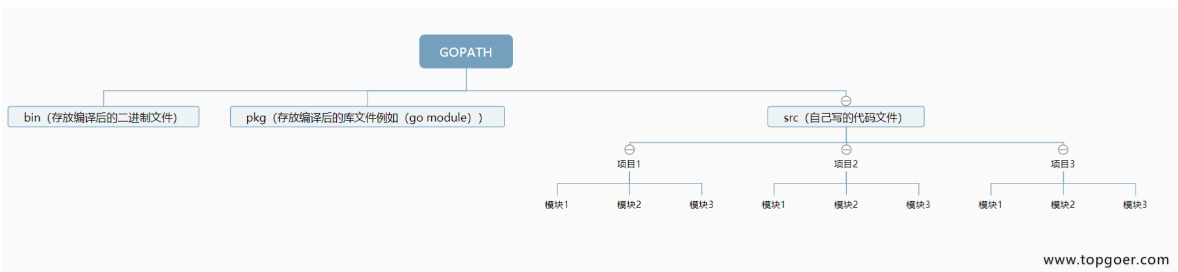
go的项目目录

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。

适合个人开发者

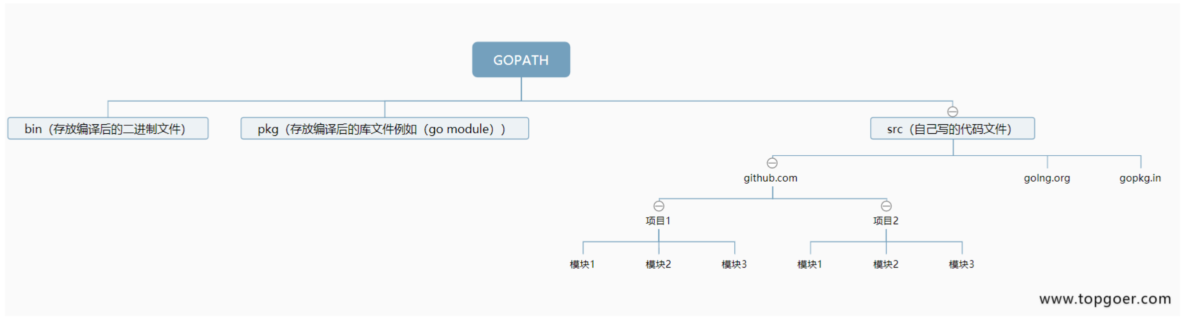
我们知道源代码都是存放在 `GOPATH` 的 `src` 目录下，那我们可以按照下图来组织我们的代码。



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

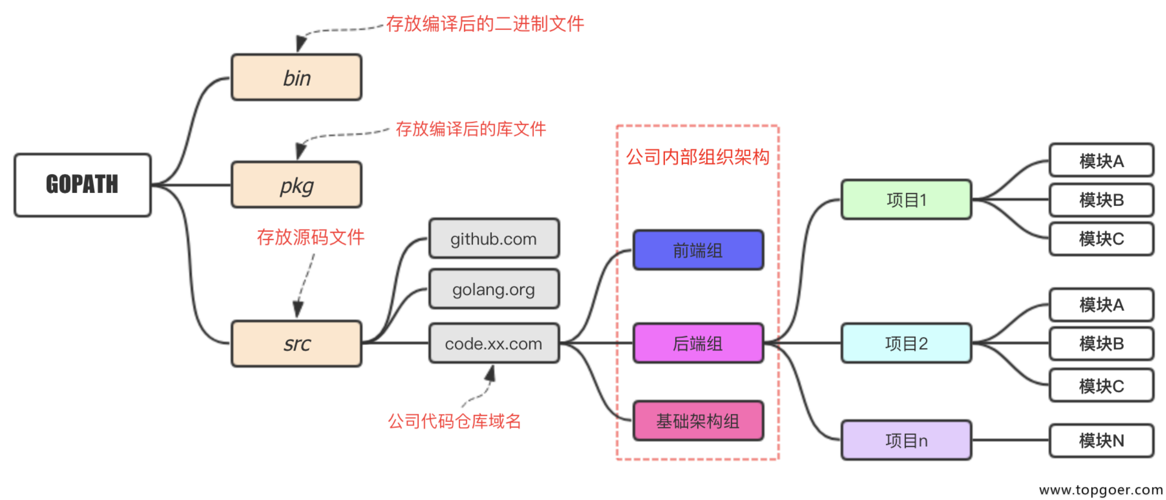
```
import "github.com/lisi/studygo"
```

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` 。

适合企业开发者



编辑器

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。

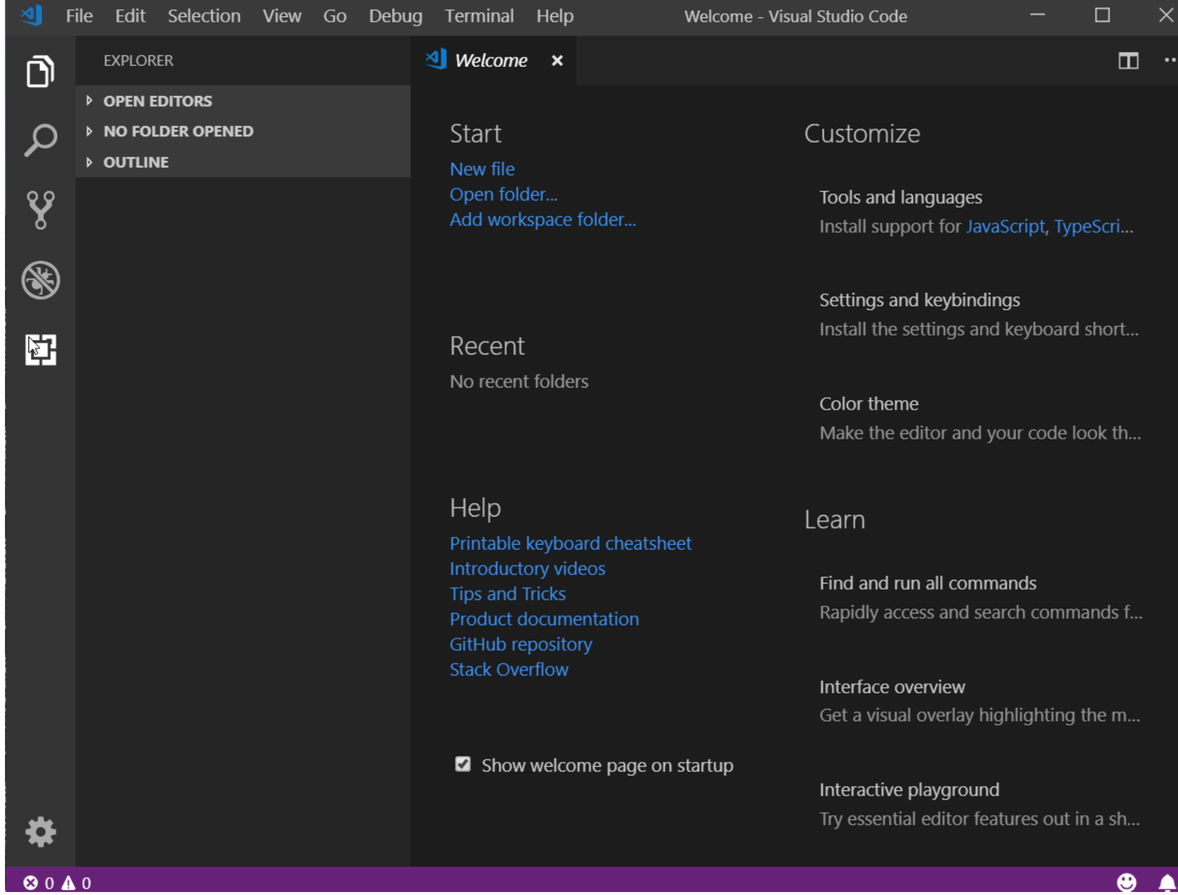
傻瓜式安装一直下一步就好了！

配置

安装中文简体插件

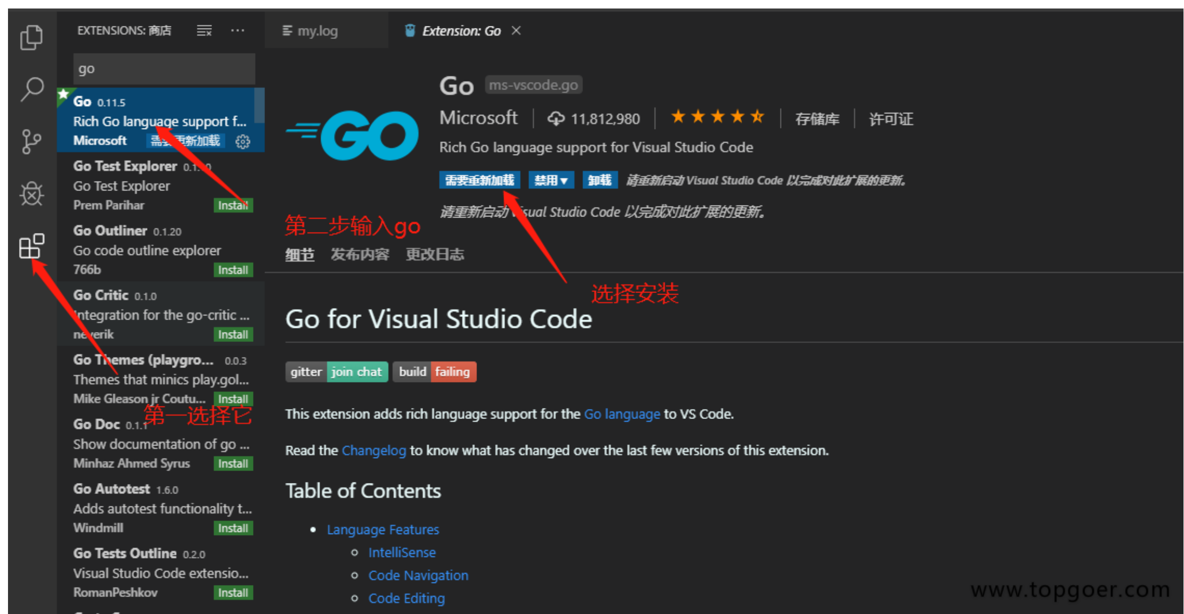
点击左侧菜单栏最后一项 `管理扩展` ，在搜索框中输入 `chinese` ，选中结果列表第一项，点击 `install` 安装。

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！



安装go插件

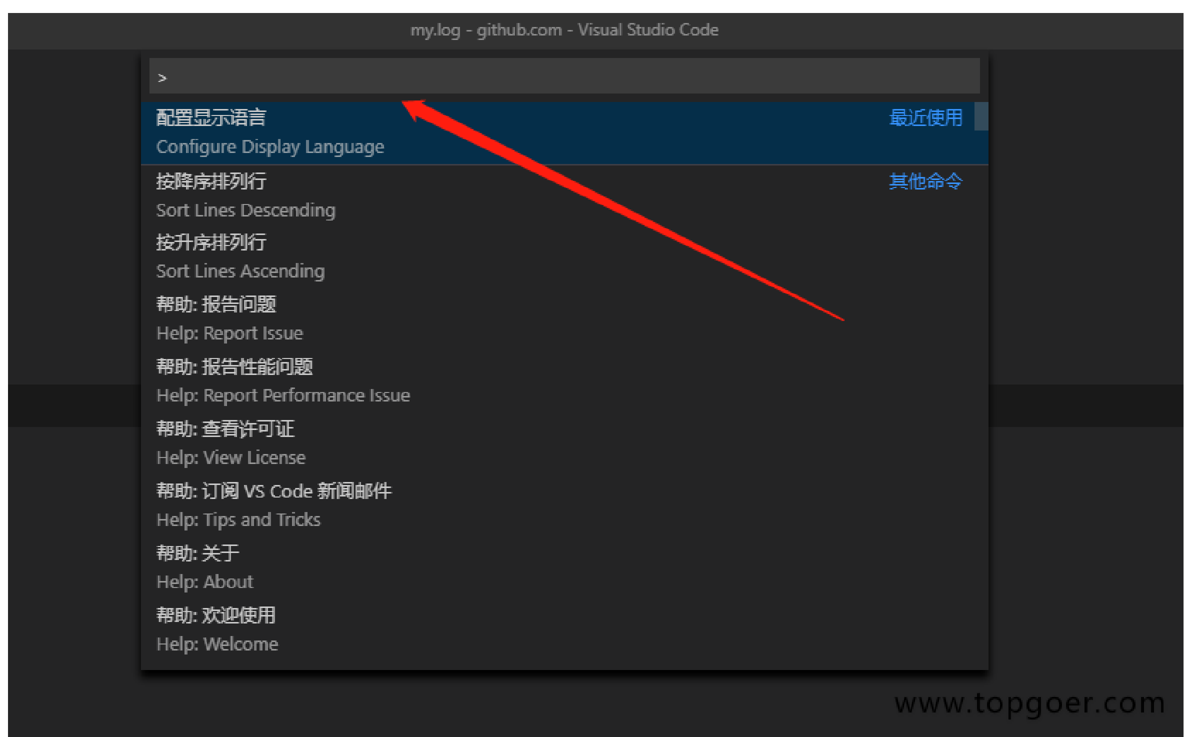
启动 `vscode` 选择插件->搜 `go` 选择 `Go for Visual Studio Code` 插件点击安装即可。如图：



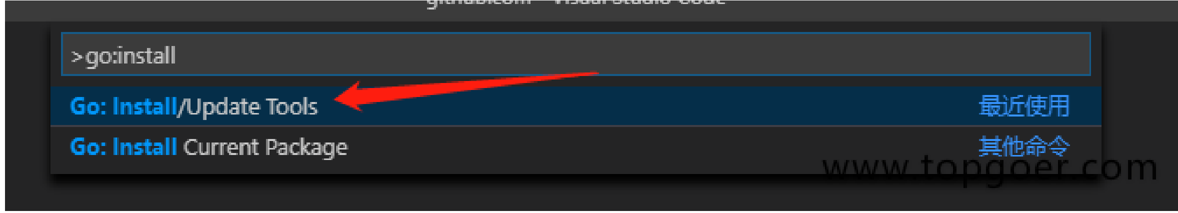
安装Go语言开发工具包

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。

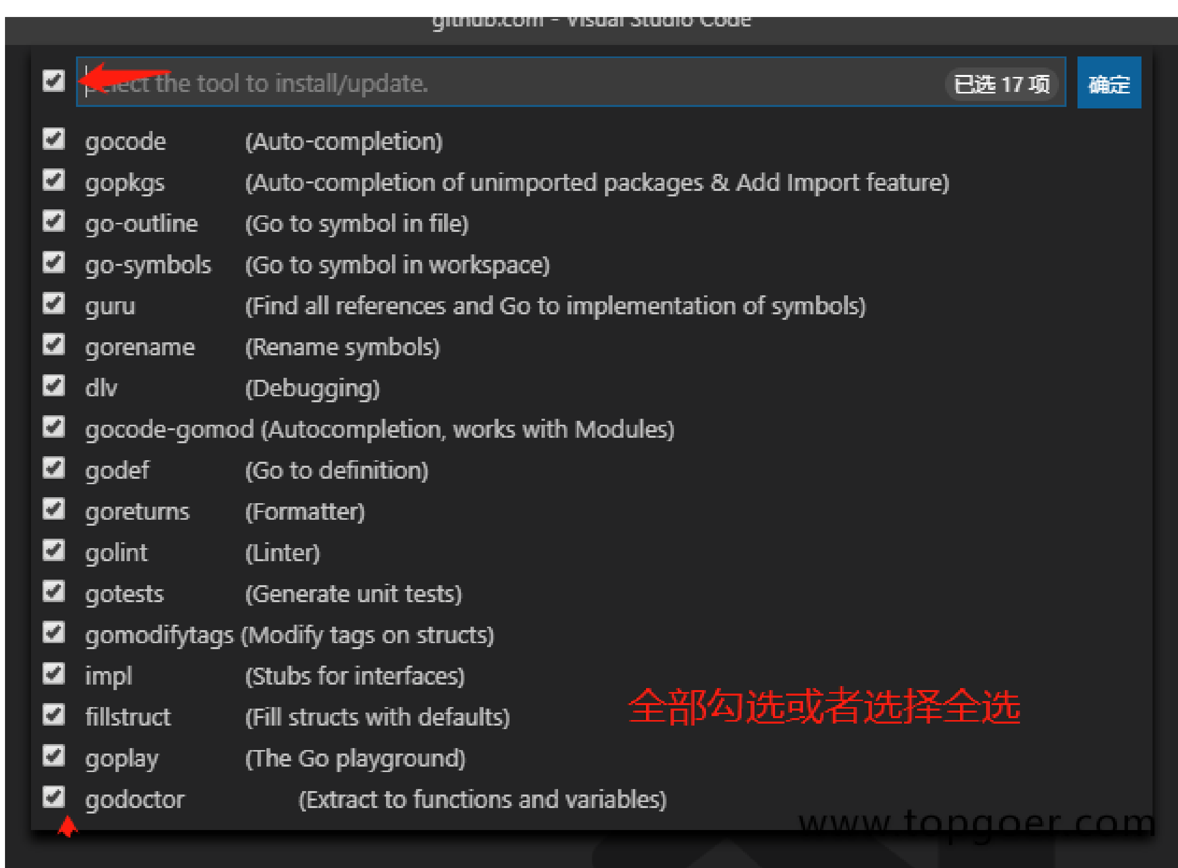
Windows平台按下 `Ctrl+Shift+P` ，Mac平台按 `Command+Shift+P` ，这个时候 `VS Code` 界面会弹出一个输入框，如下图：



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

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。

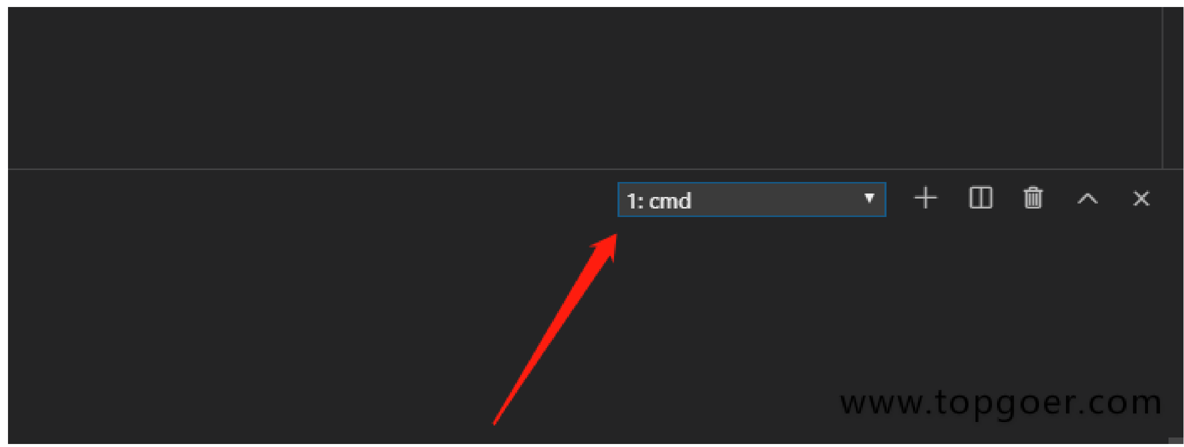
经过上面的步骤就可以安装成功了。 这个时候创建一个Go文件，就能正常使用代码提示、代码格式化等工具了。

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/18OJ5jOn5Kt6wANjQyGdWDA` ，密码:dxti。

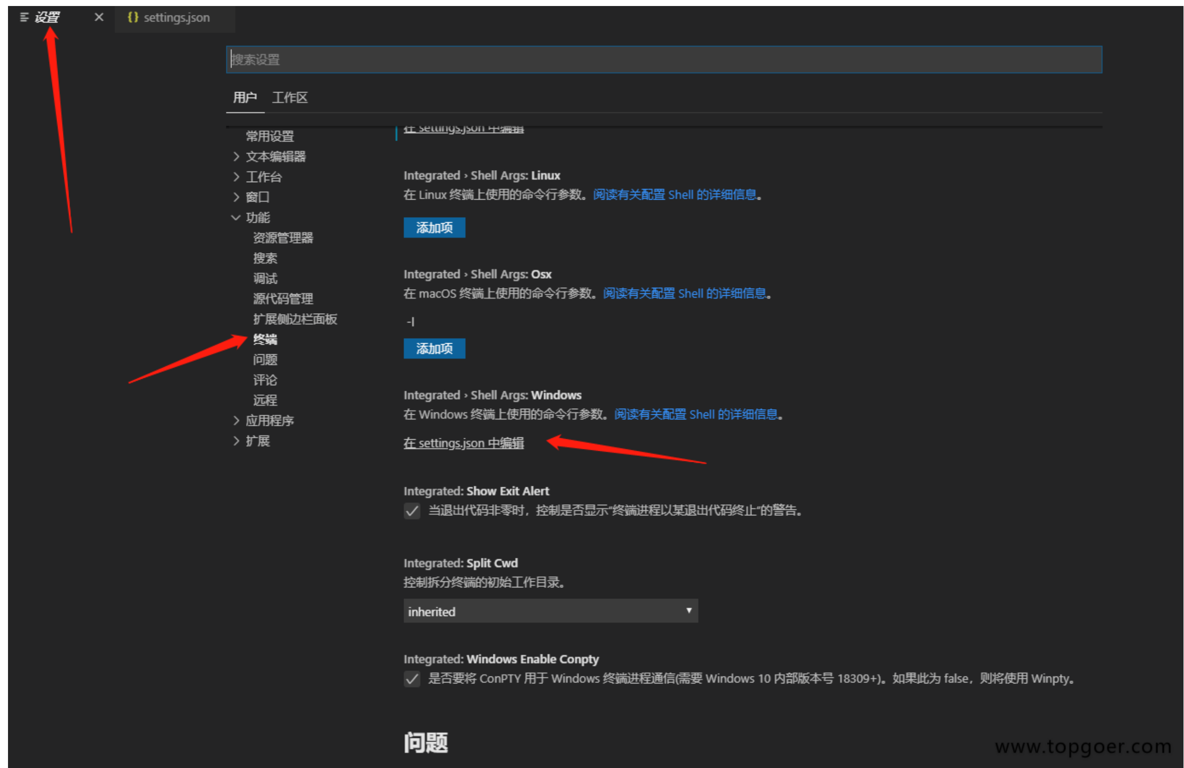
注意：特别是Mac下需要给拷贝的这些文件赋予可执行的权限。

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

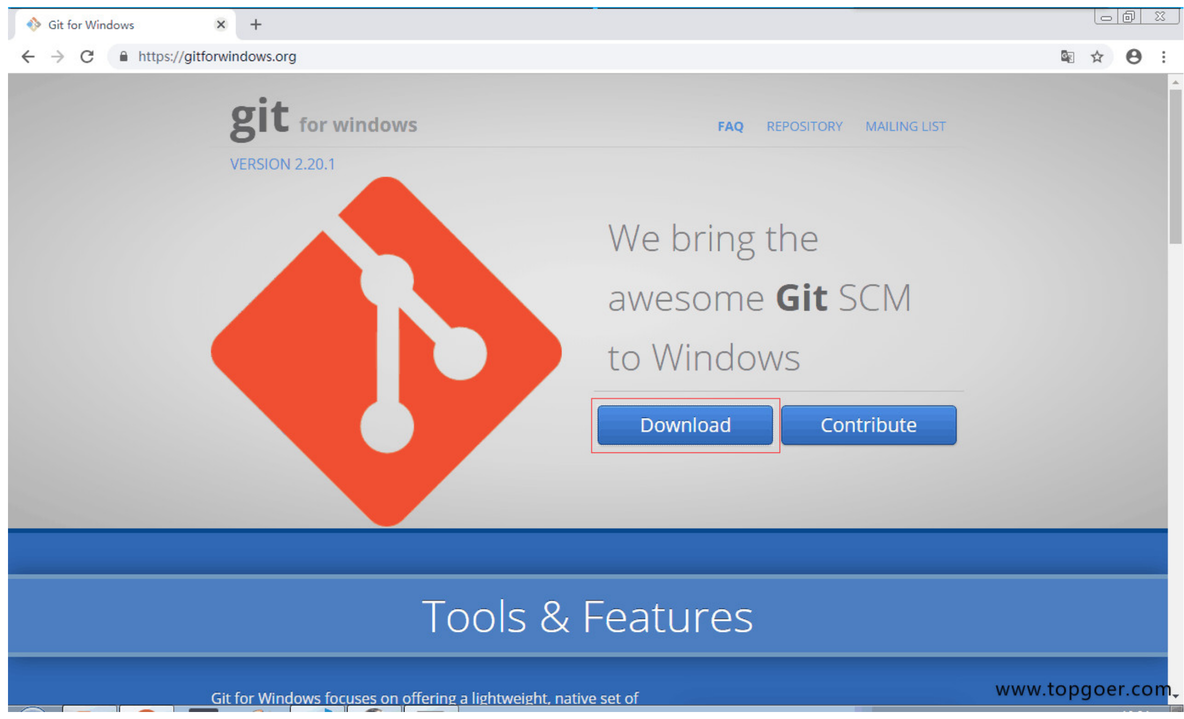
Git安装

安装git

若你的Windows 系统已经装有git，跳过该安装。

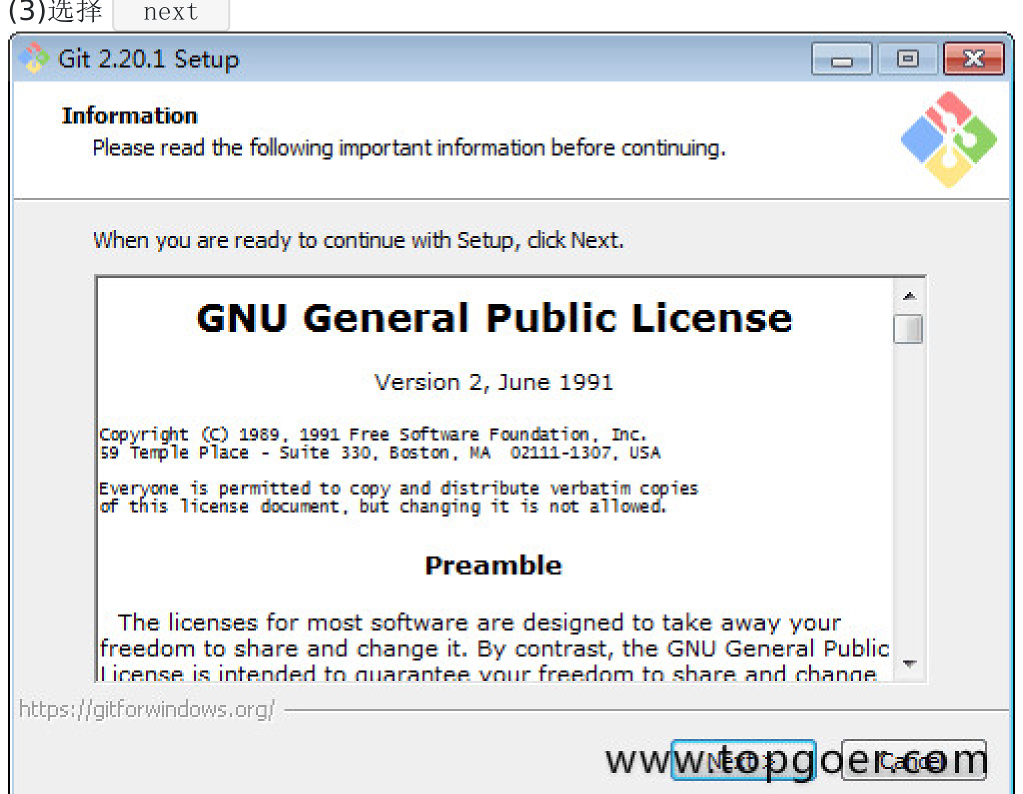
下载地址：http://msysgit.github.com/

(1)进入官网，根据官网提示点击下载相应的版本。

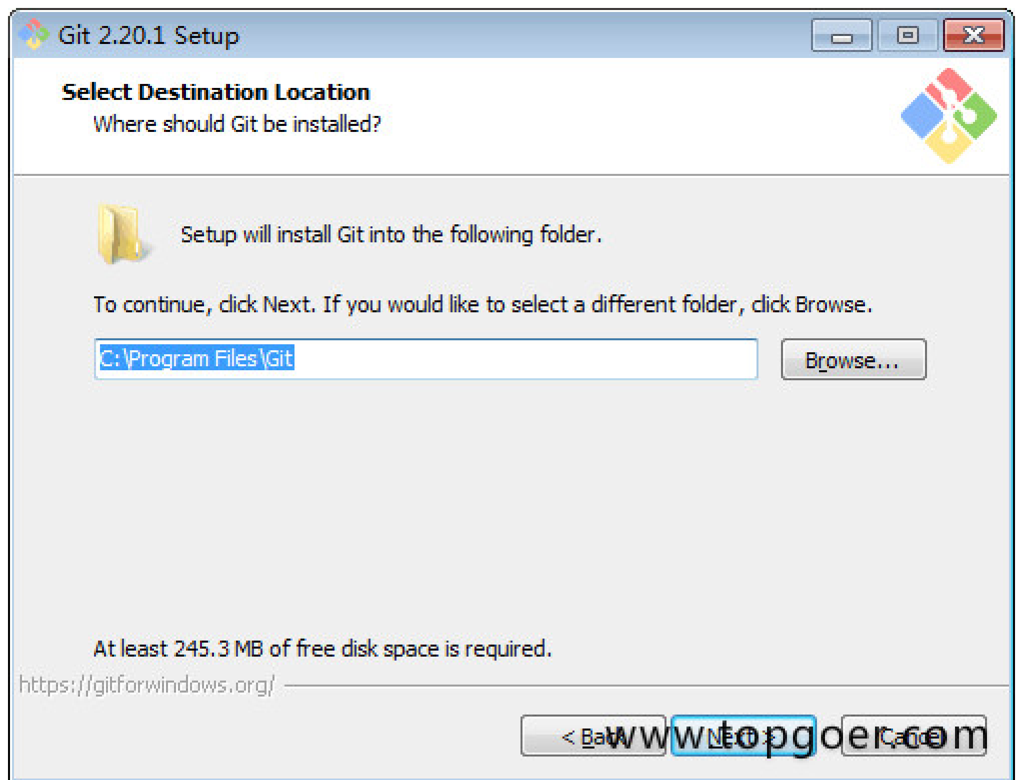


(2) `git` 下载完成双击下载安装

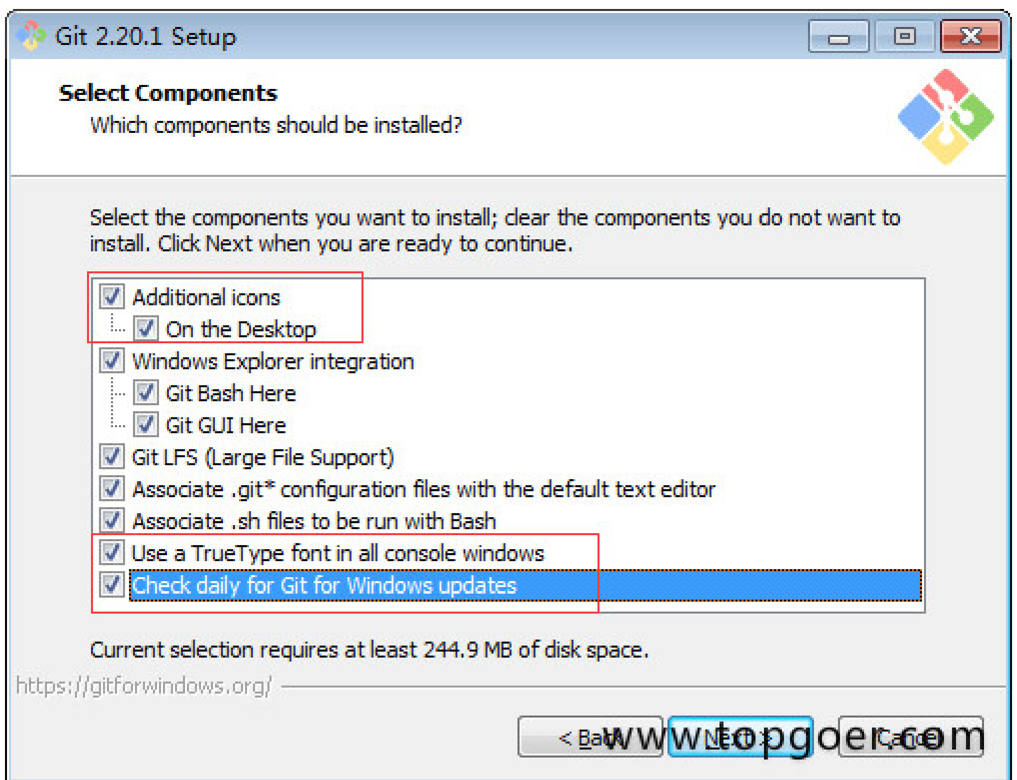
(3)选择 `next`



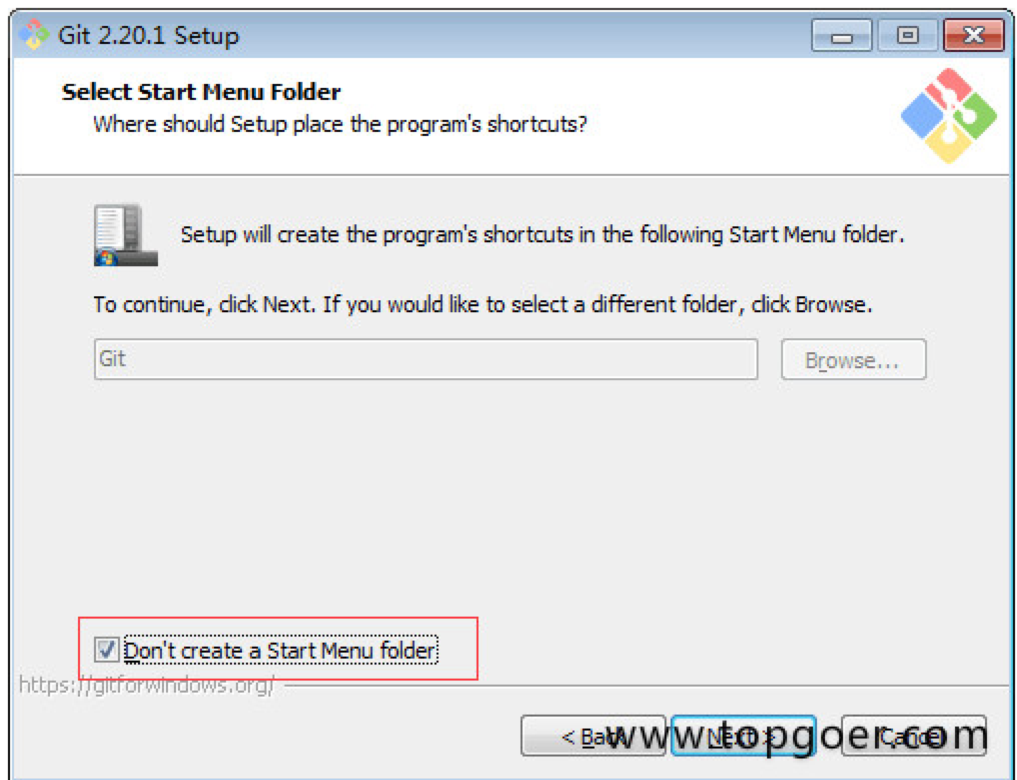
(4)文件位置存储，可根据自己盘的情况安装



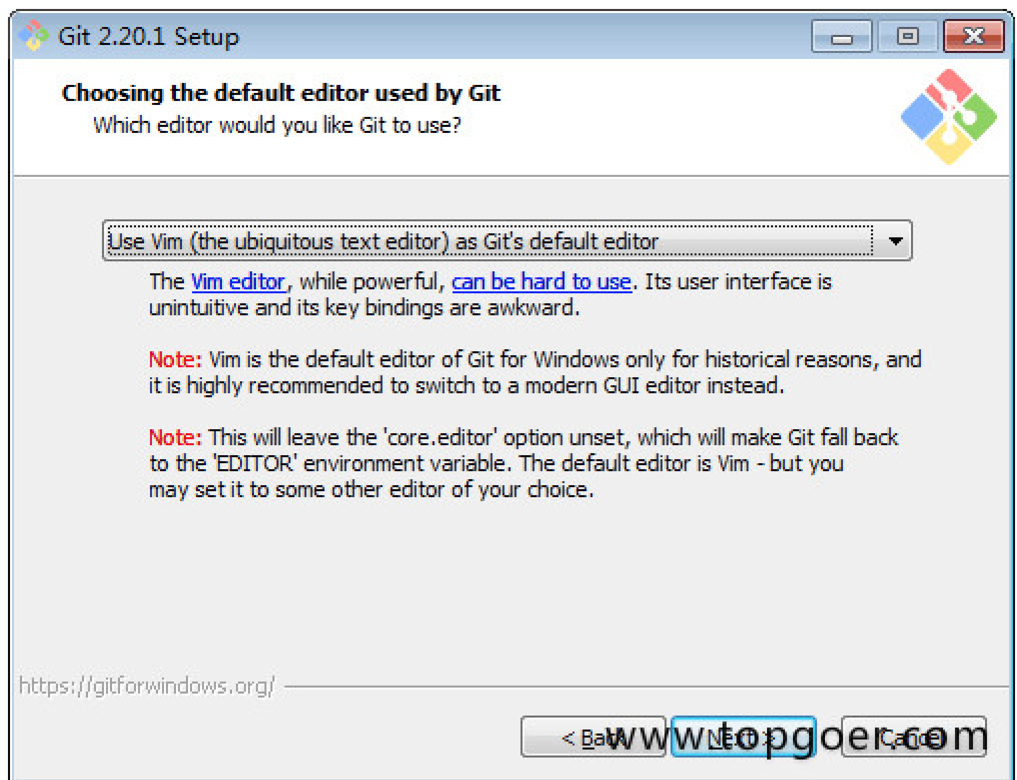
(5)安装配置文件，自己需要的都选上，下一步



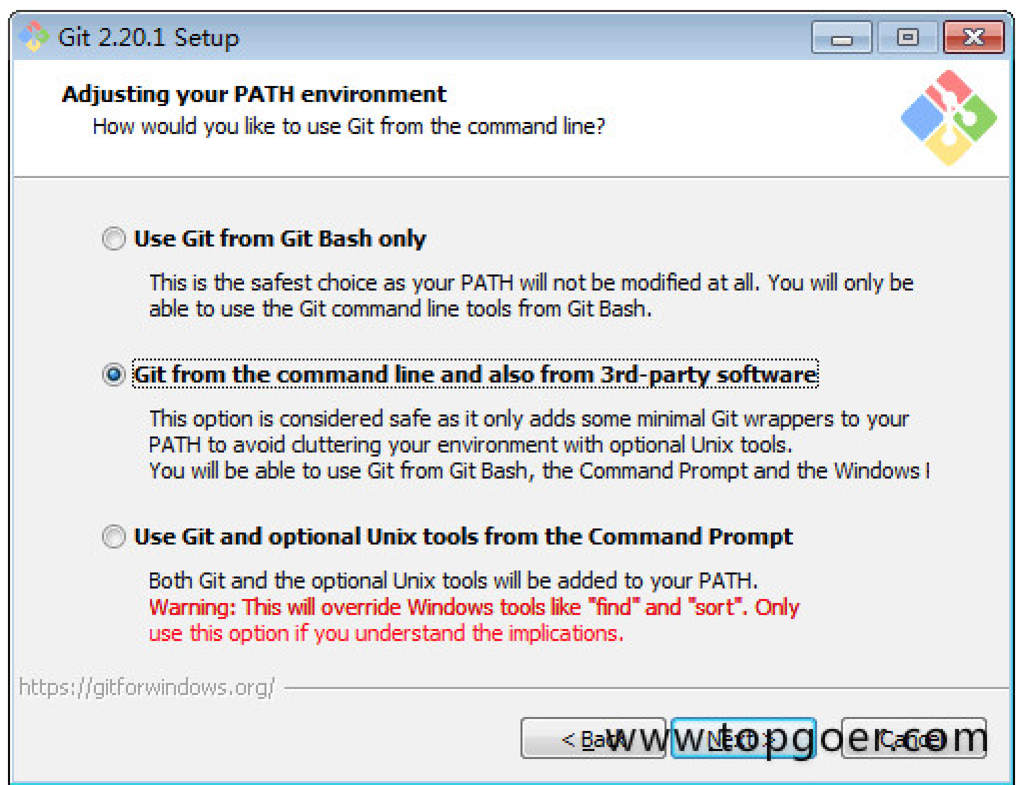
(6)不创建启动文件夹，下一步：



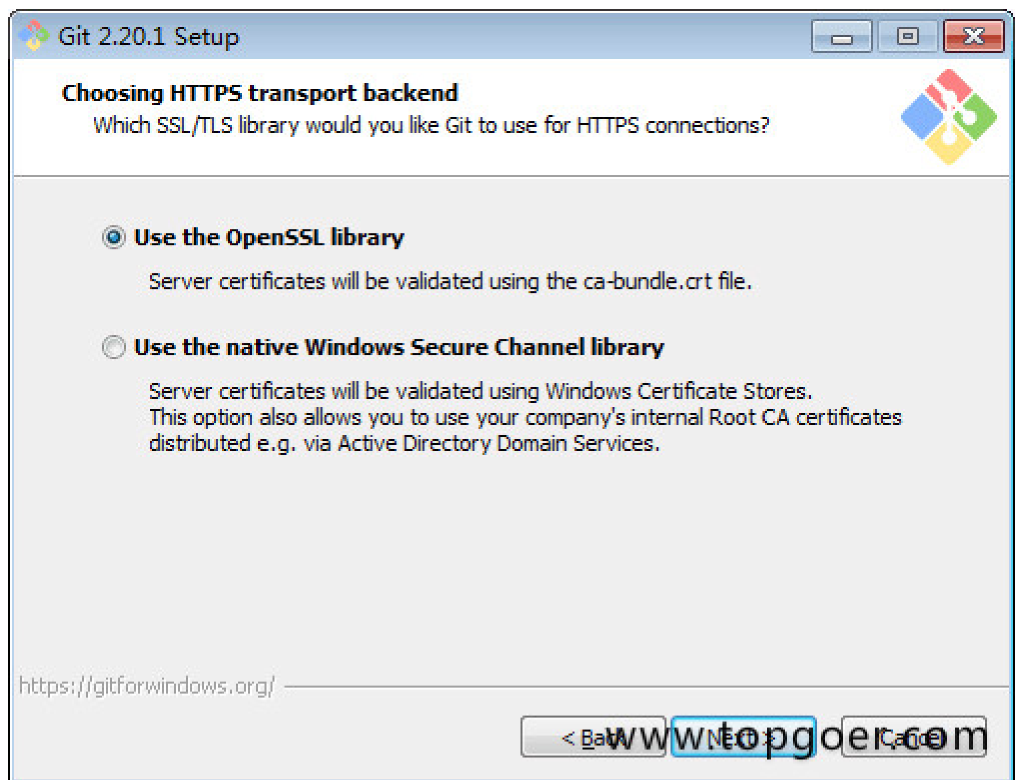
(7)选择默认的编辑器，我们直接用推荐的就行，下一步



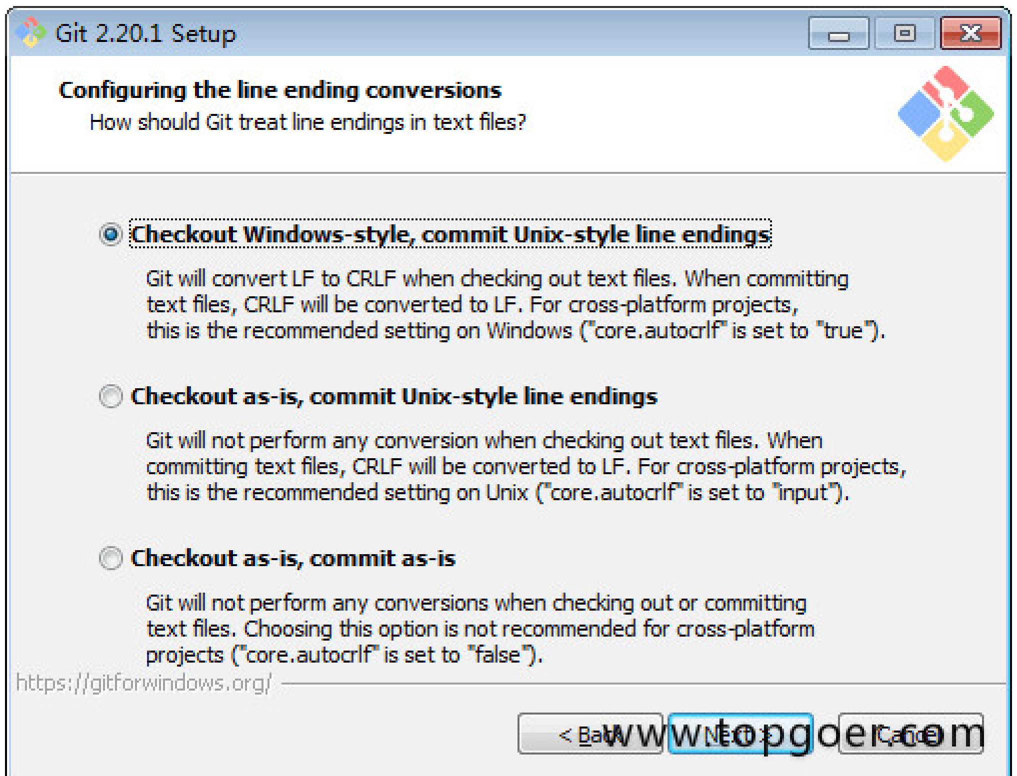
(8)勾上第二项，这样就可以在cmd中操作，下一步



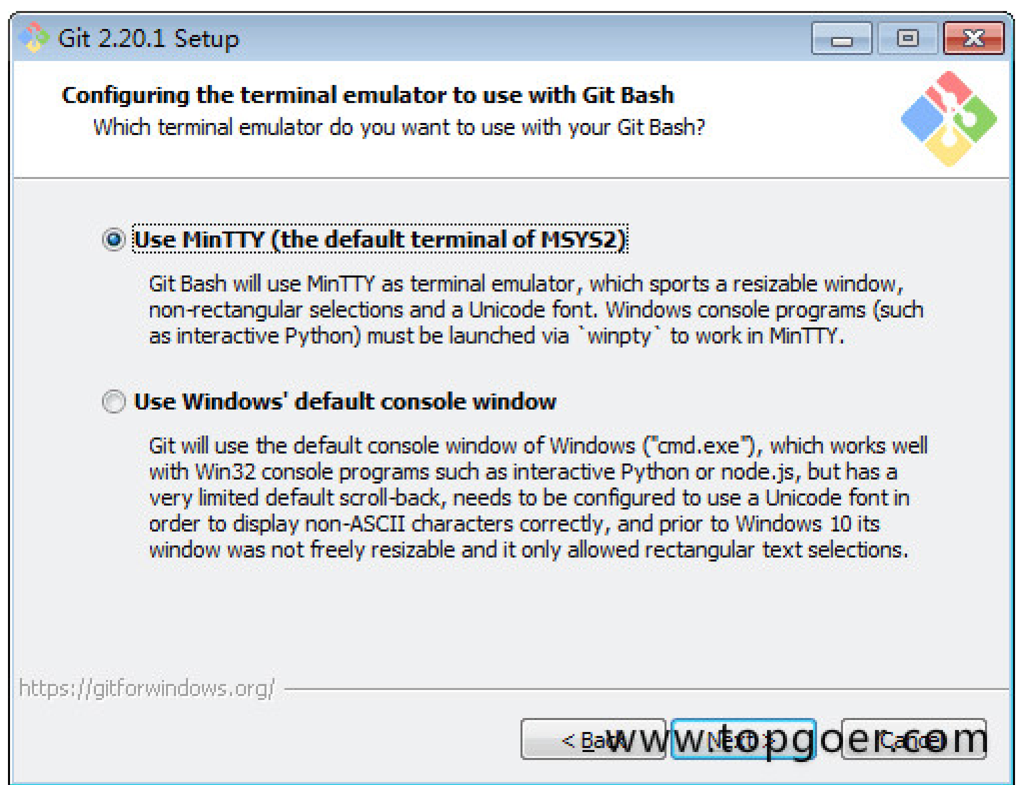
(9)使用默认设置就行，下一步：



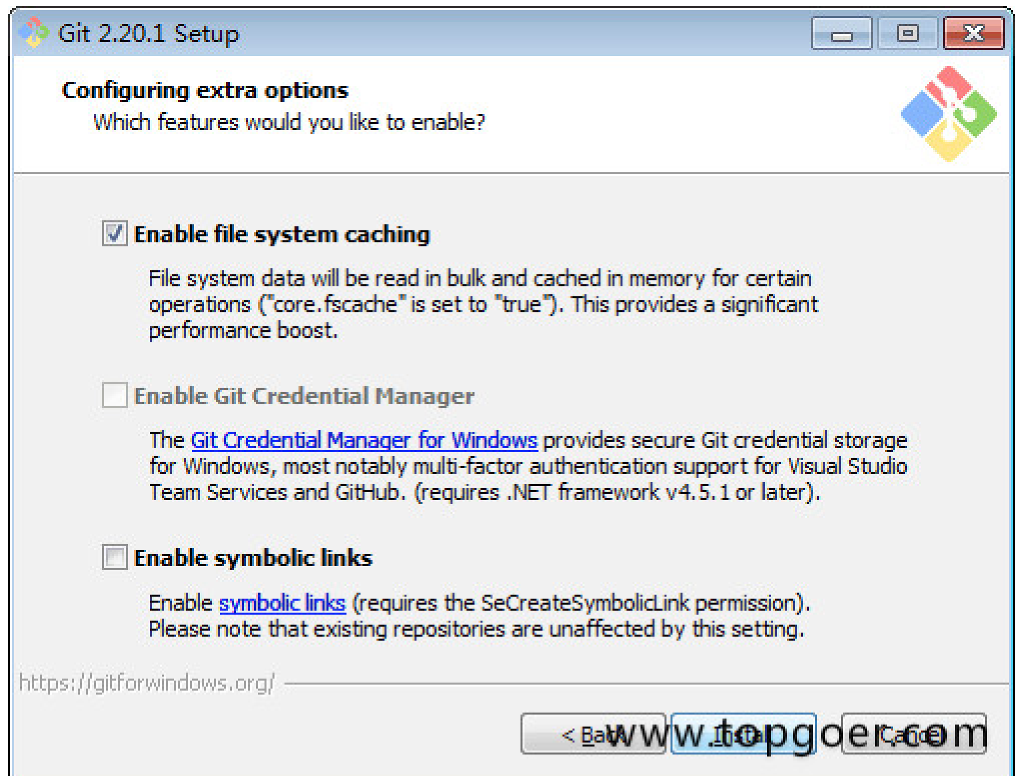
(10)配置行结束标记，保持默认“Checkout”



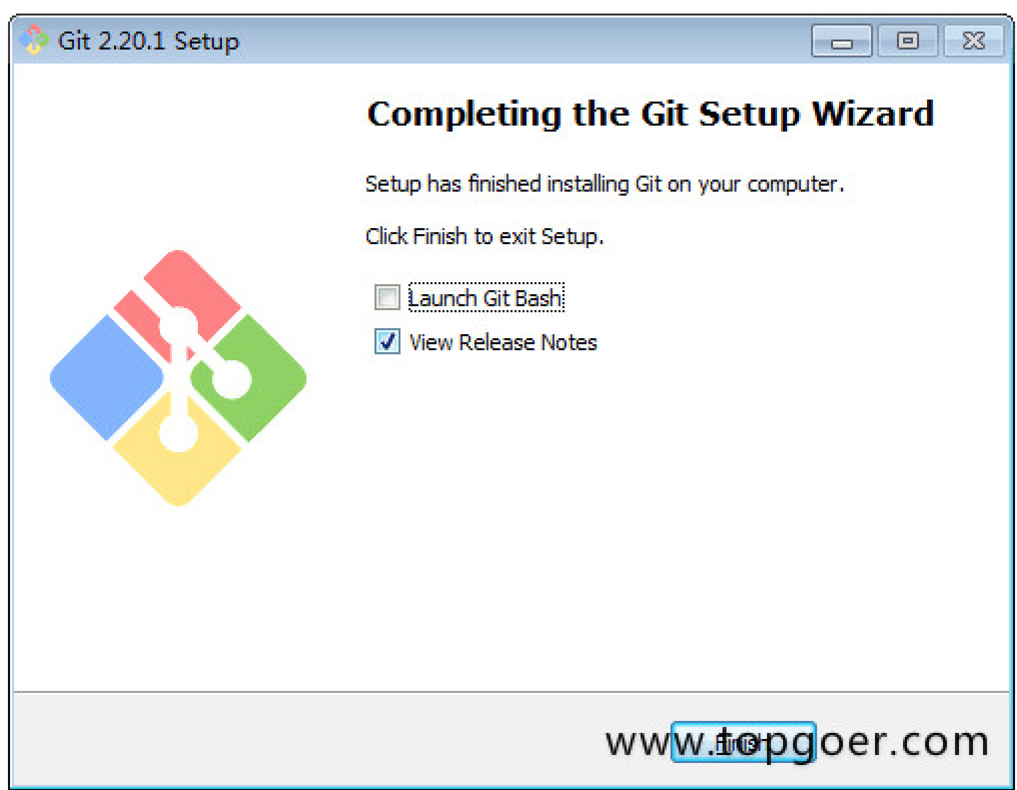
(11)在终端模拟器选择页面，默认即可，配置后Git



(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。

第一个go程序

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：

```go
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
```

Go基础

Go语言的主要特征

Golang内置类型和函数

Init函数和main函数

命令

运算符

下划线

变量和常量

基本类型

数组Array

切片Slice

Slice底层实现

指针

Map

Map实现原理

结构体

Go语言的主要特征

golang 简介

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。

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

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。

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。
```

Golang文件名：

```
`所有的go源码都是以".go"结尾。`
```

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protec
t
    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
```

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。

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
```

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；
```

Golang内置类型和函数

内置类型

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
```

引用类型：(指针类型)

```
    slice   -- 序列数组(最常用)
    map     -- 映射
    chan    -- 管道
```

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
```

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

    }
```

Init函数和main函数

`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
```

`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
}
```

`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。

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。

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
```

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。

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。

逻辑运算符

<table>
<tr><th>运算符</th><th>描述</th></tr>
<tr><td>&&</td><td>逻辑 AND 运算符。 如果两边的操作数都是 True，则为 True，否则为 False。</td></tr>
<tr><td>||</td><td>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。</td></tr>
<tr><td>!</td><td>逻辑 NOT 运算符。 如果条件为 True，则为 False，否则为 True。</td></tr>
</table>

位运算符

位运算符对整数在内存中的二进制位进行操作。

<table>
<tr><th>运算符</th><th>描述</th></tr>
<tr><td>&</td><td>参与运算的两数各对应的二进位相与。（两位均为1才为1）</td></tr>
<tr><td>|</td><td>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）</td></tr>
<tr><td>^</td><td>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）</td></tr>
<tr><td><<</td><td>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。</td></tr>
<tr><td>>></td><td>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。</td></tr>
</table>

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

运算符	描述
<<=	左移后赋值
>>=	右移后赋值
&=	按位与后赋值
|=	按位或后赋值
^=	按位异或后赋值

下划线

“_”是特殊标识符，用来忽略结果。

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.
```

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。

变量和常量

变量

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。

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。

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。

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
```

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
)
```

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。
```

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。

```go
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：

```go
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：

```go
const (
    n1 = 100
    n2
    n3
)
```

上面示例中，常量 `n1、n2、n3` 的值都是 `100` 。

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：

```go
const (
        n1 = iota //0
        n2        //1
        n3        //2
        n4        //3
    )
```

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
    )
```

基本类型

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

类型	长度(字节)	默认值	说明
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。

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。

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` 。

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。

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。
```

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"
```

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\"")
}
```

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。

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

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。

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))
    }
```

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)
    }
```

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。
```

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：

```go
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

```go
    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
```

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)
}
```

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"

//       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)
}
```

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。
```

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)
}
```

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
```

操作	含义
s[n]	切片 s 中索引位置为 n 的项
s[:]	从切片 s 的索引位置 0 到 len(s)-1 处所获得的切片
s[low:]	从切片 s 的索引位置 low 到 len(s)-1 处所获得的切片
s[:high]	从切片 s 的索引位置 0 到 high 处所获得的切片，len=high
s[low:high]	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low
s[low:high:max]	从切片 s 的索引位置 low 到 high 处所获得的切片，len=high-low，cap=max-low
len(s)	切片 s 的长度，总是<=cap(s)
cap(s)	切片 s 的容量，总是>=len(s)

www.topgoer.com

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]
```

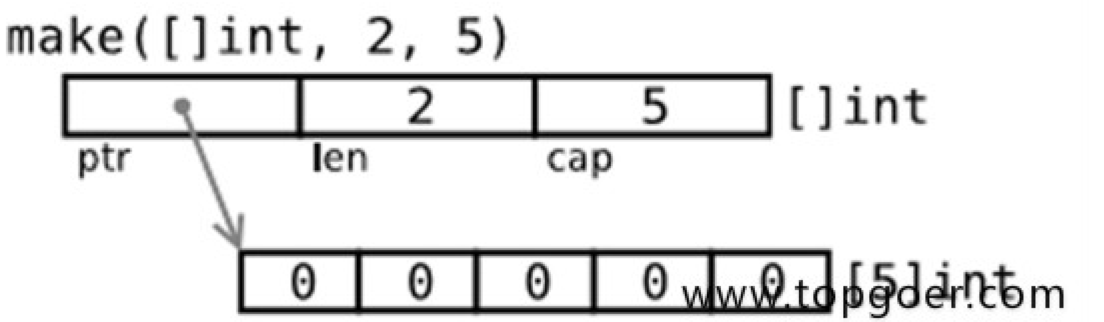
```
var slice1 []int = arr[0:6]        //可以简写为 var slice []int = arr[:end]
var slice2 []int = arr[5:10]       //可以简写为 var slice[]int = arr[start:]
var slice3 []int = arr[0:len(arr)] //var slice []int = arr[:]
var slice4 = arr[:len(arr)-1]      //去掉切片的最后一个元素
func main() {
    fmt.Printf("全局变量：arr %v\n", 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]
```

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：

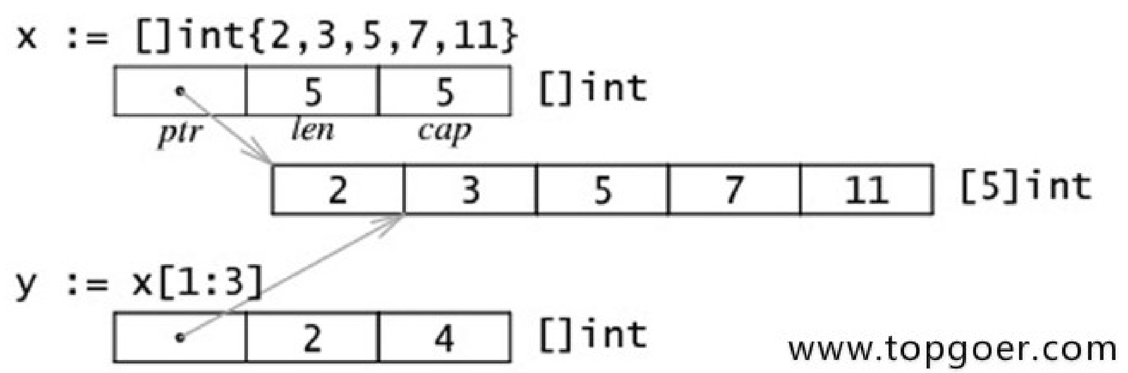
```
package main

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
```

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]
```

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。

slice中cap重新分配规律：

```go
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
```

切片拷贝

```go
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。

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
```

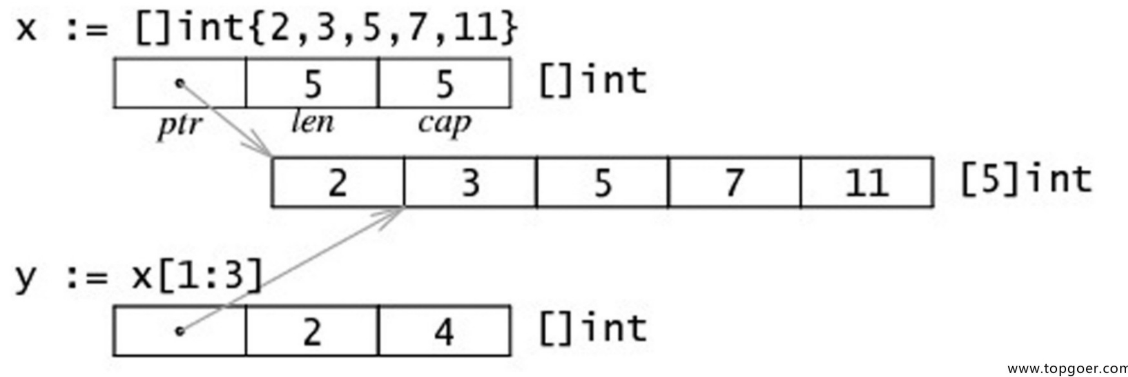
切片resize（调整大小）

```
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
```

数组和切片的内存布局



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!
```

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:

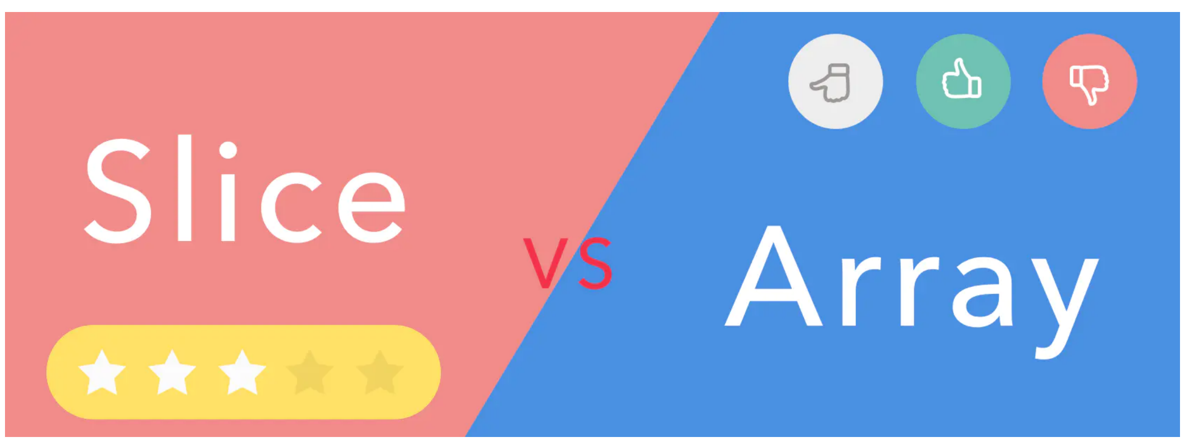
```
    strings.Replace(strings.Trim(fmt.Sprint(array_or_slice), "[]"), " ", ",", -1)
```

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。

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/o
p              0 alloc s/op
    BenchmarkSlice-4              300000                  4055 ns/op              8192 B/o
p              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。

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。

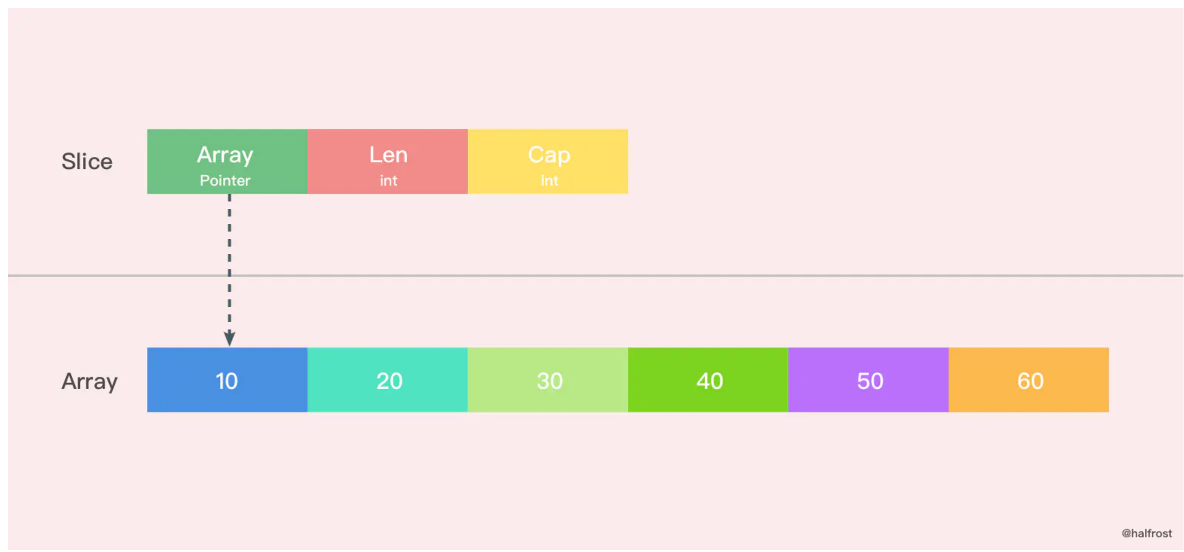
切片（slice）是对数组一个连续片段的引用，所以切片是一个引用类型（因此更类似于 C/C++ 中的数组类型，或者 Python 中的 list 类型）。这个片段可以是整个数组，或者是由起始和终止索引标识的一些项的子集。需要注意的是，终止索引标识的项不包括在切片内。切片提供了一个与指向数组的动态窗口。

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)
```

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

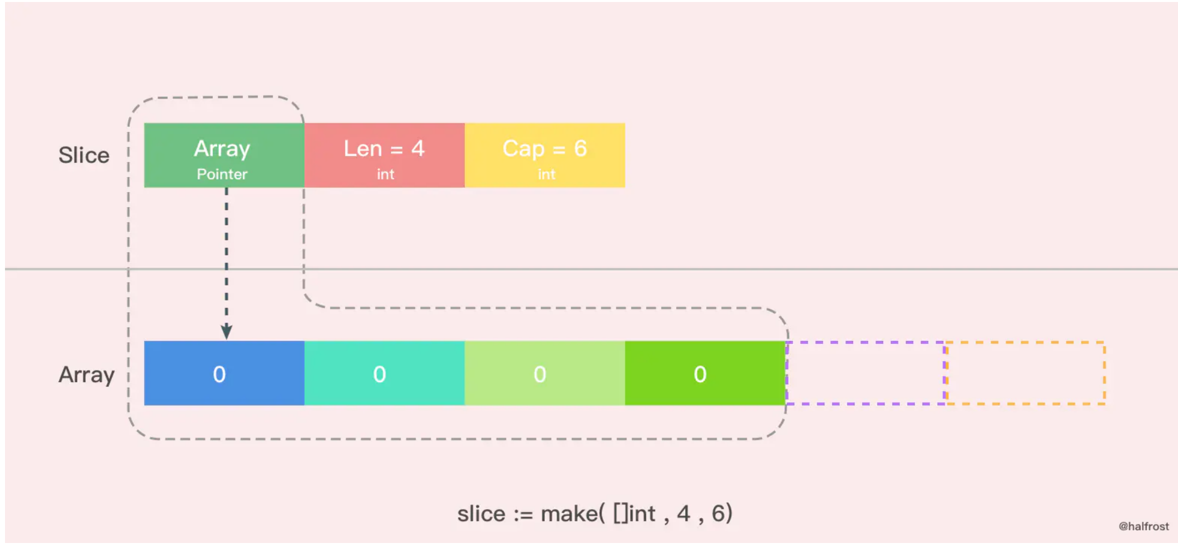
```
func makeslice(et *_type, len, cap int) slice {
    // 根据切片的数据类型，获取切片的最大容量
    maxElements := maxSliceCap(et.size)
    // 比较切片的长度，长度值域应该在[0,maxElements]之间
    if len < 0 || uintptr(len) > maxElements {
        panic(errorString("makeslice: len out of range"))
    }
    // 比较切片的容量，容量值域应该在[len,maxElements]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 {
```

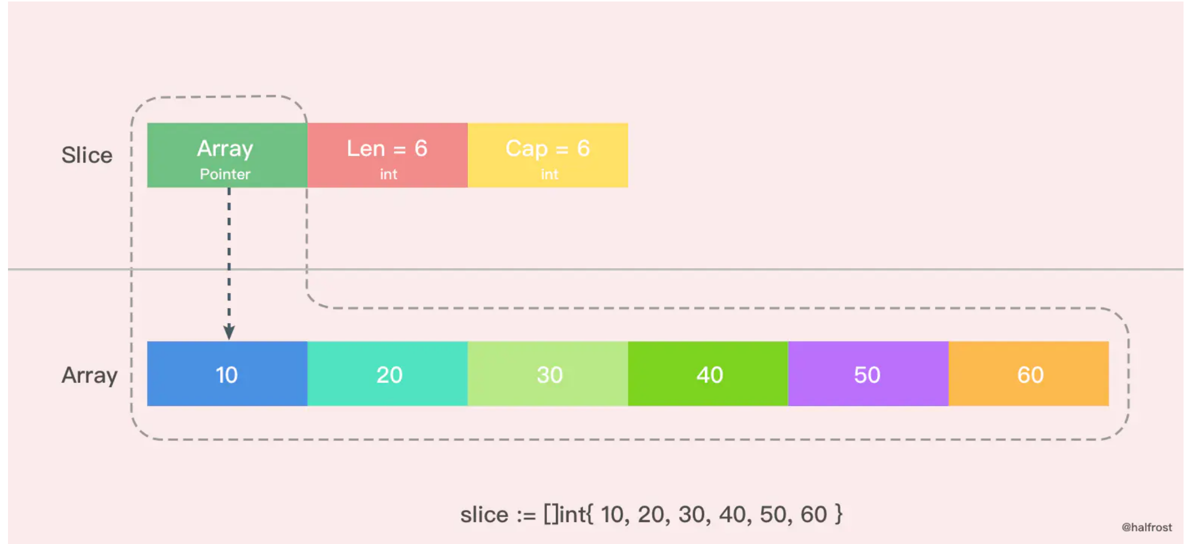
```
        panic(errorString("makeslice: len out of range"))
    }

    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。

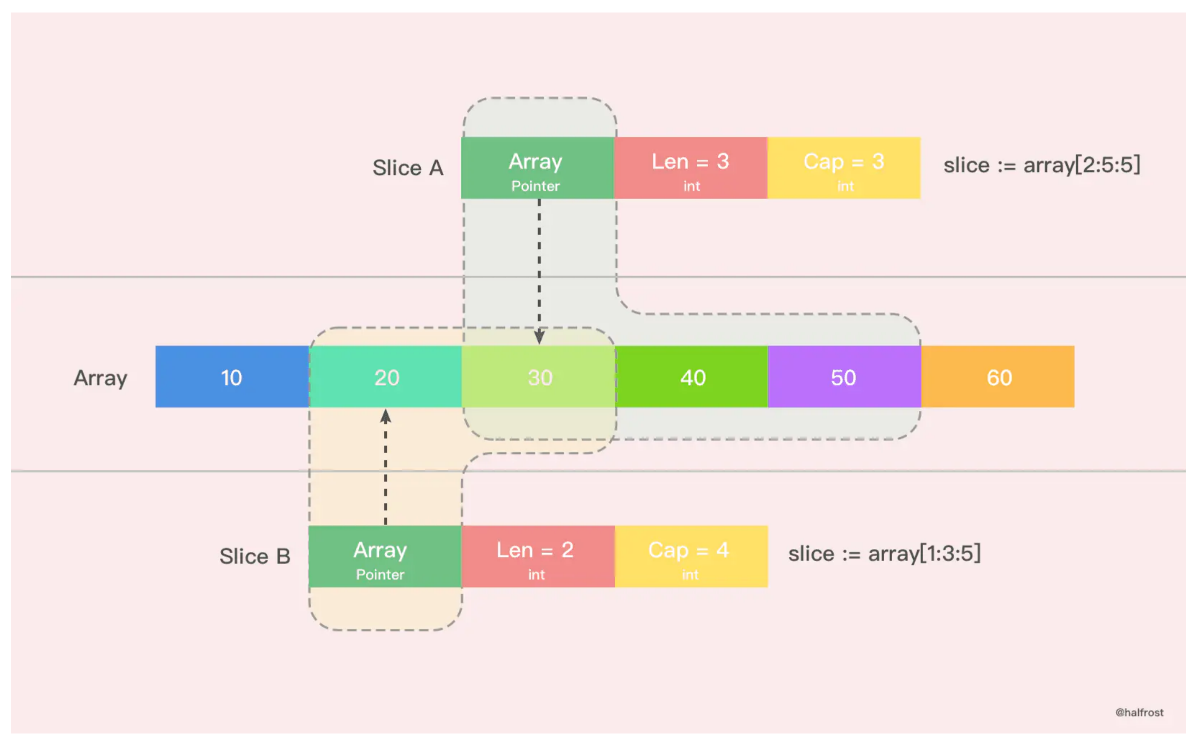


上图是用 make 函数创建的一个 len = 4， cap = 6 的切片。内存空间申请了6个 int 类型的内存大小。由于 len = 4，所以后面2个暂时访问不到，但是容量还是在的。这时候数组里面每个变量都是0 。

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

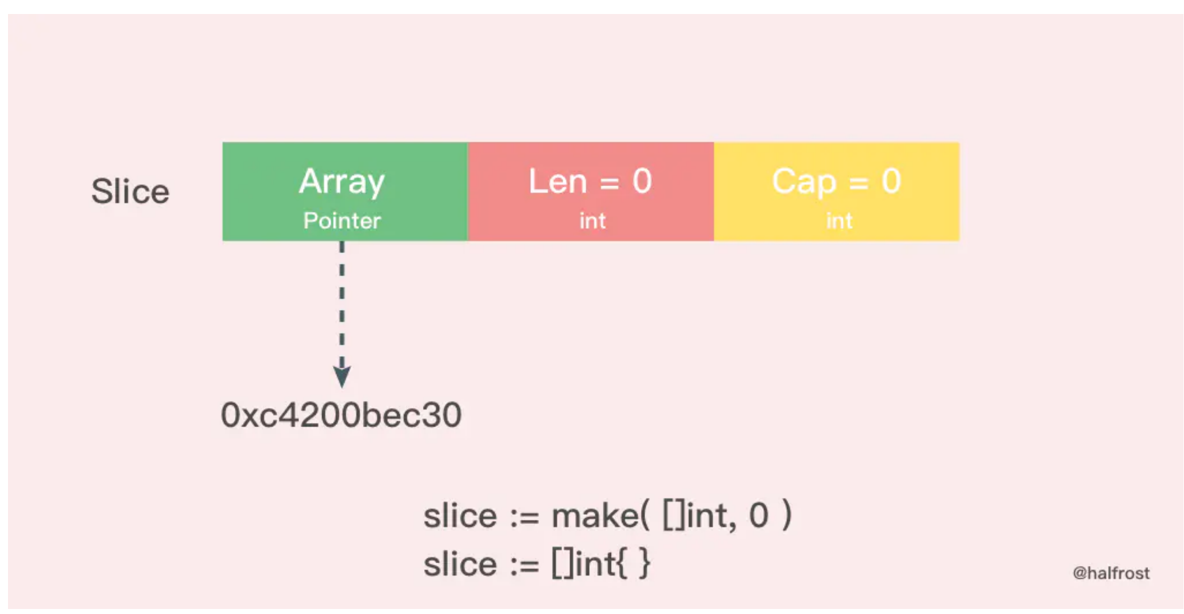
nil 切片和空切片也是常用的。

```
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。

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.arr
ay 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

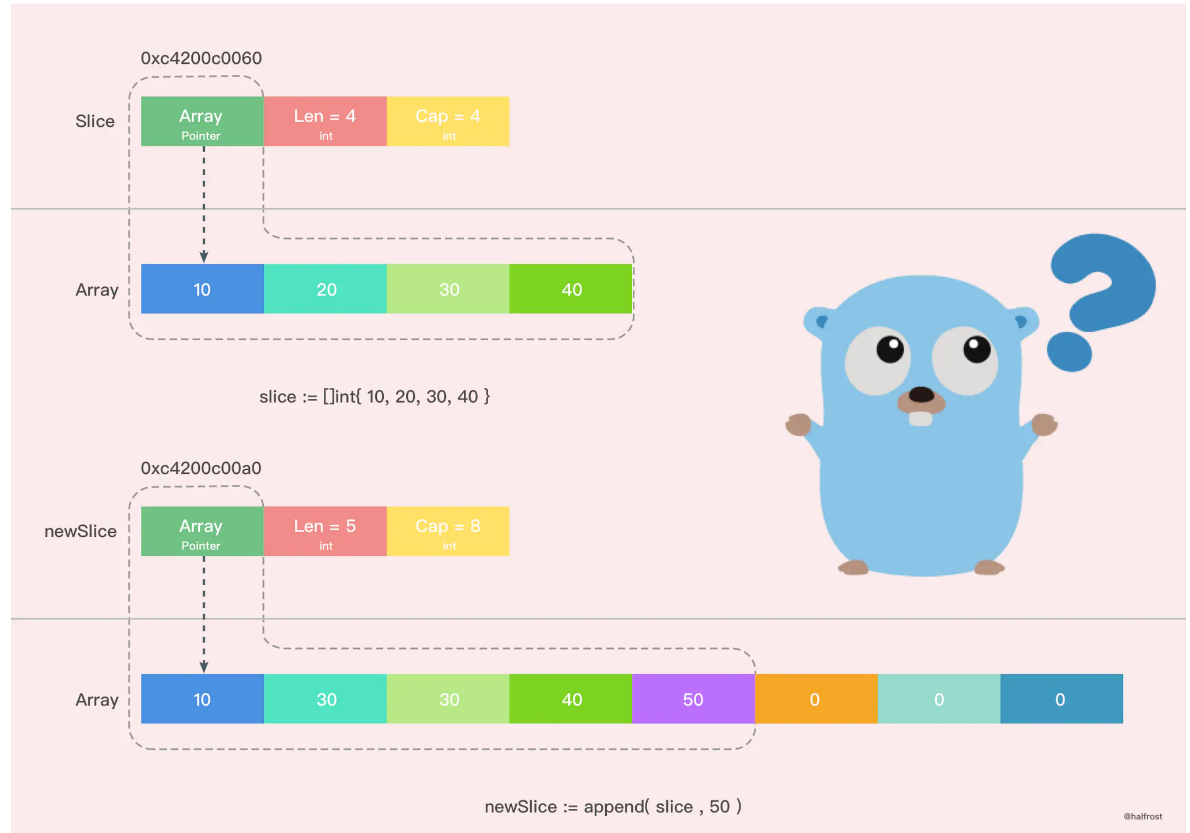
先看看扩容策略。

```
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：

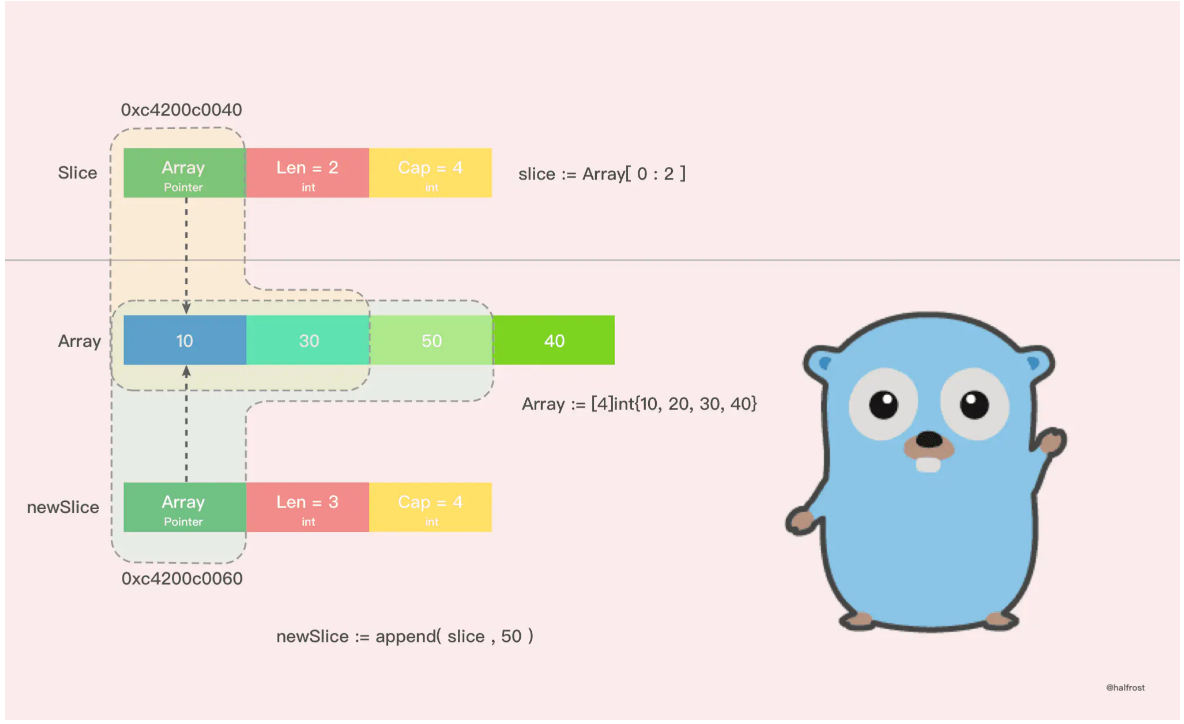
```
func main() {
    array := [4]int{10, 20, 30, 40}
    slice := array[0:2]
    newSlice := append(slice, 50)
    fmt.Printf("Before slice = %v, Pointer = %p, len = %d, cap = %d\n", slice, &
```

```
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。

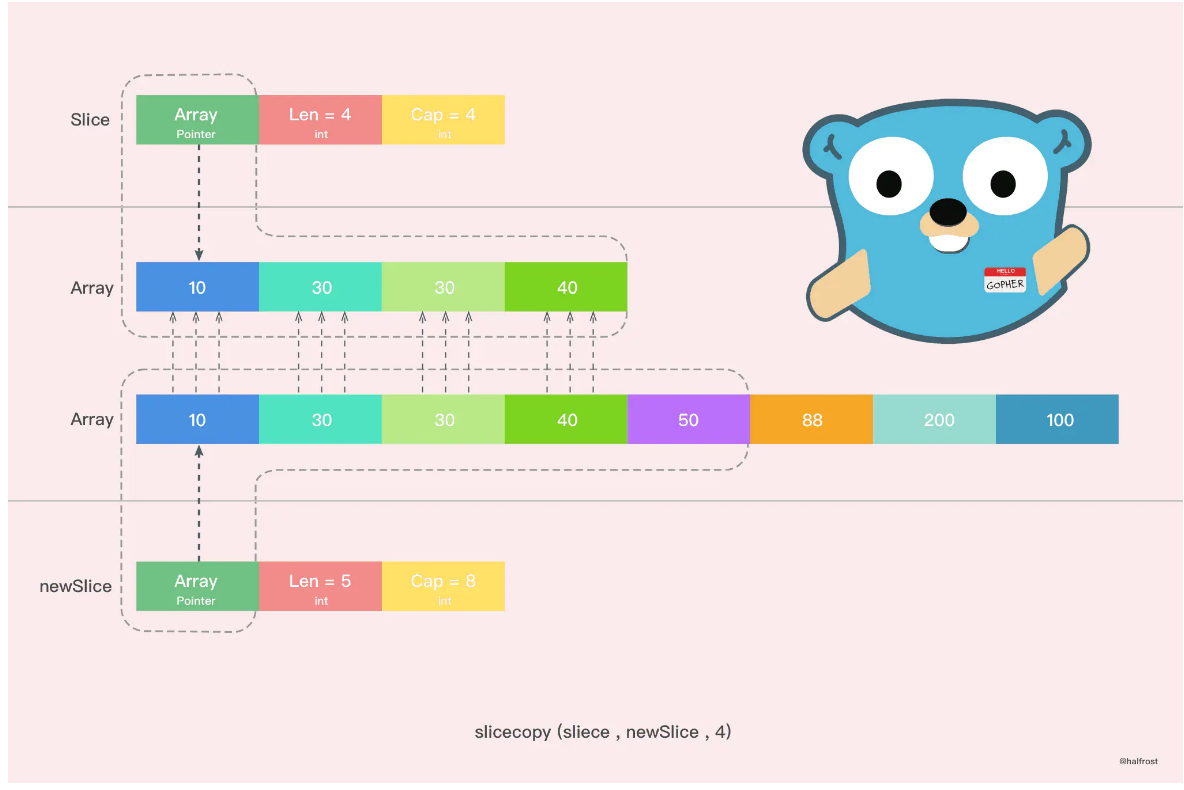
切片拷贝

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]
```

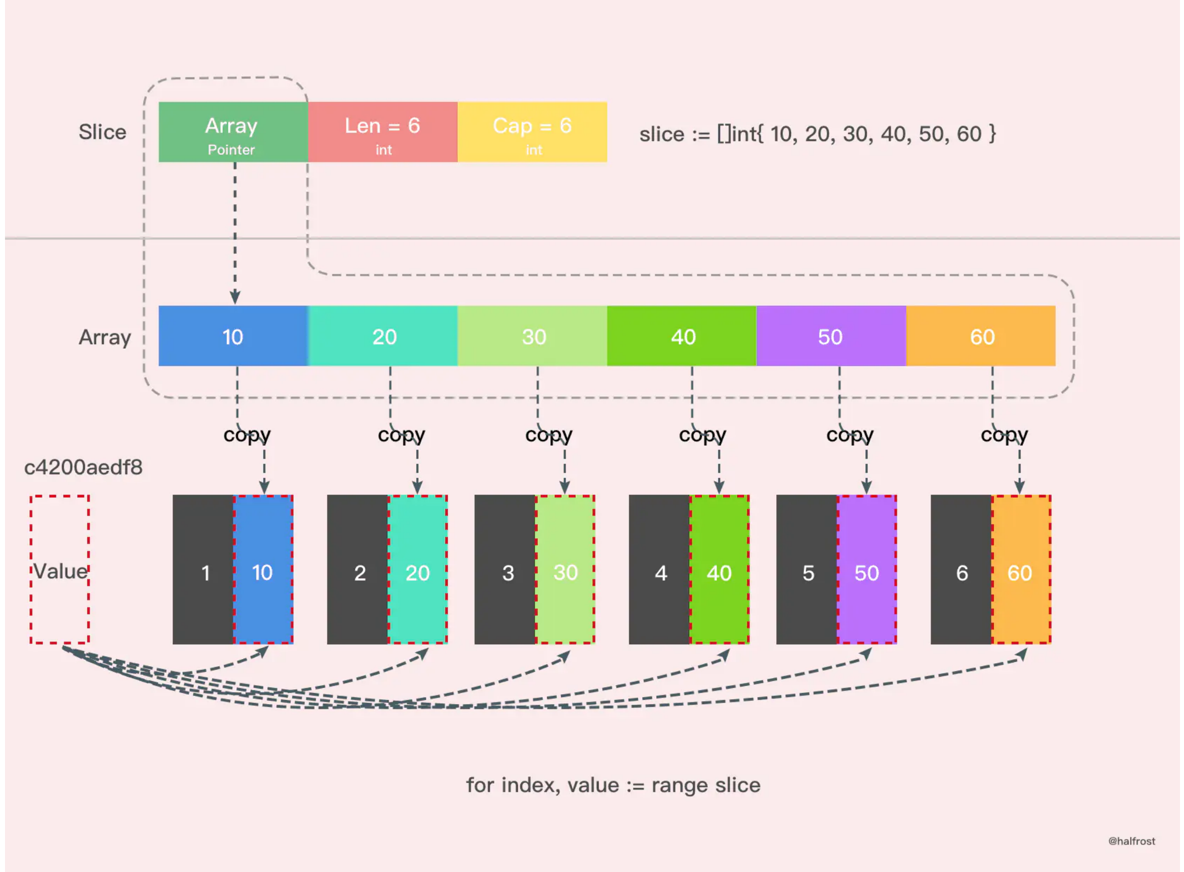
说到拷贝，切片中有一个需要注意的问题。

```
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, &v
alue, &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

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。

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）。

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
```

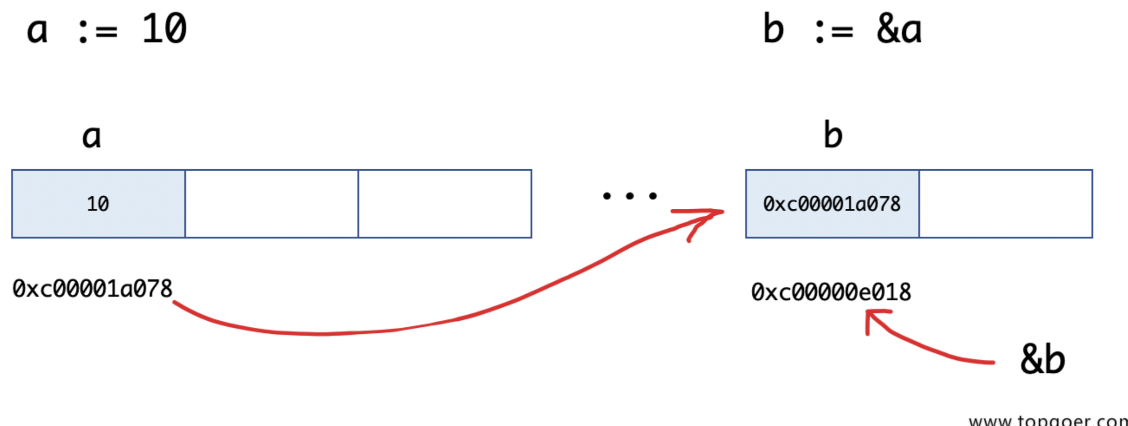
其中：

```
    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：



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
}
```

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")
    }
}
```

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

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：

```go
func main() {
	var a *int
	a = new(int)
	*a = 10
	fmt.Println(*a)
}
```

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：

```go
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：

```go
func main() {
	var b map[string]int
	b = make(map[string]int, 10)
	b["测试"] = 100
	fmt.Println(b)
}
```

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。
```

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

```go
package main

import "fmt"

func main() {
	var a int
	fmt.Println(&a)
	var p *int
	p = &a
	*p = 20
	fmt.Println(a)
}
```

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。

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。

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) //
}
```

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")
    }
}
```

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。

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)
    }
}
```

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])
	}
}
```

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)
```

```
    }
}
```

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)
}
```

Map实现原理

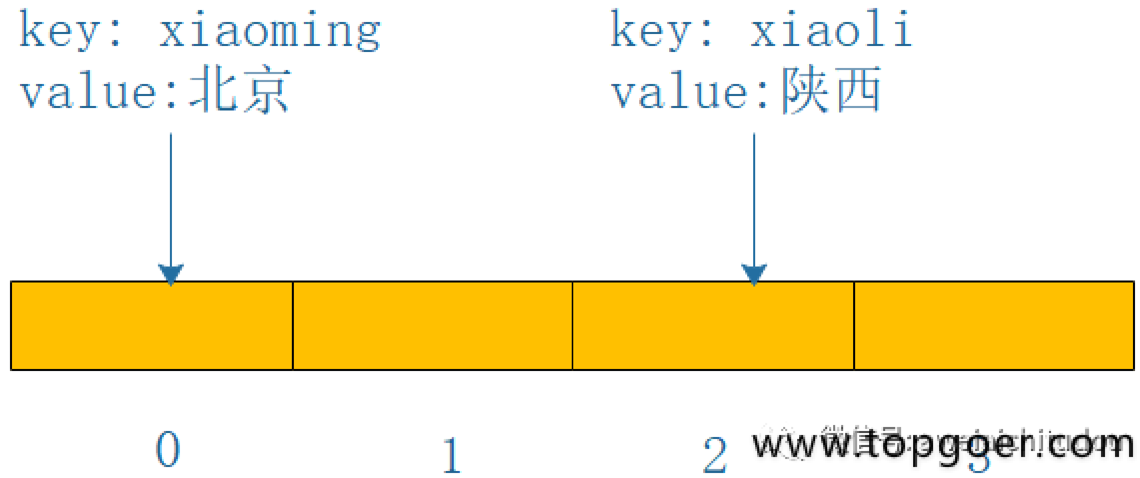
本章不属于基础部分但是面试经常会问到建议学学

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
```



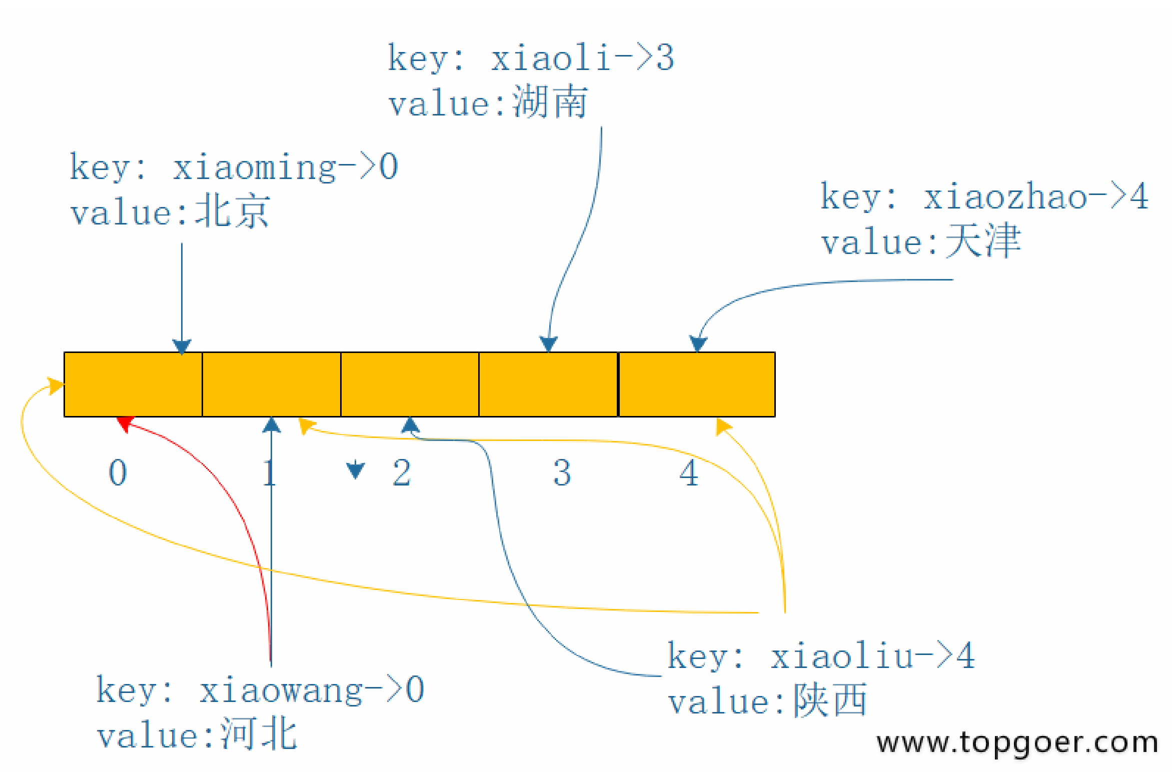
hash冲突

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

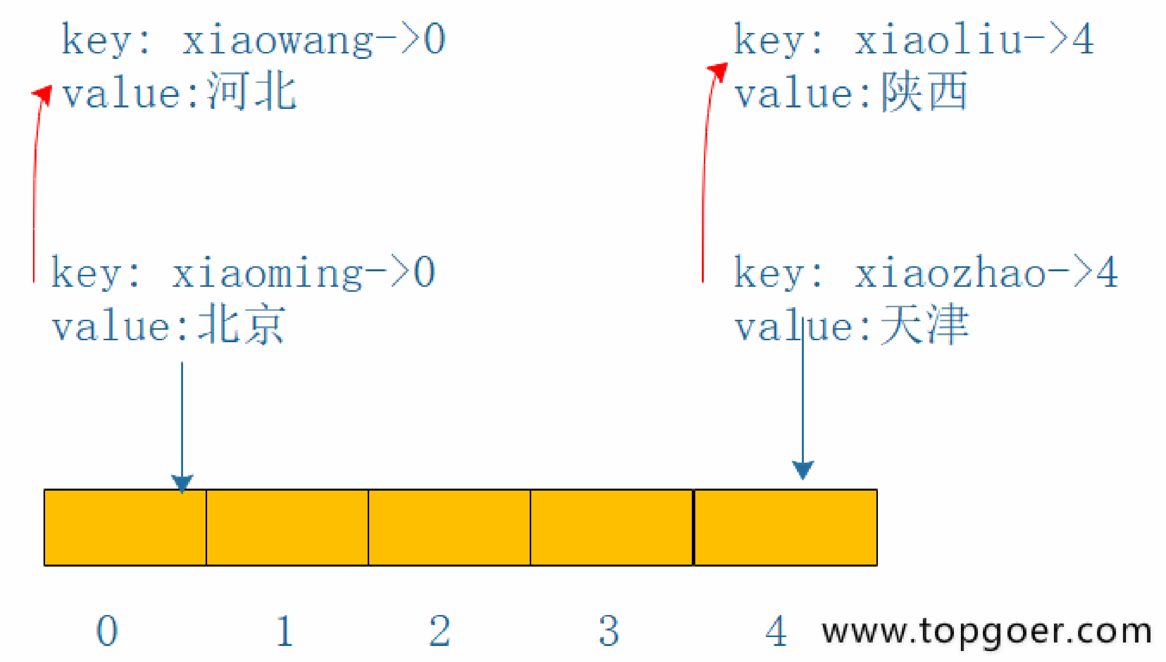
开放定址法：也就是说当我们存储一个key，value时，发现hashkey(key)的下标已经被别key占用，那我们在这个数组中空间中重新找一个没被占用的存储这个冲突的key，那么没被占用的

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

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

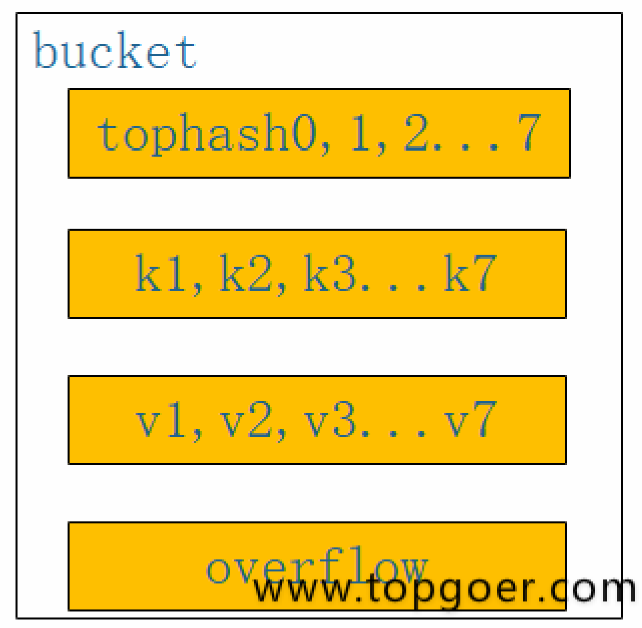
Go中Map的实现原理

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。
```

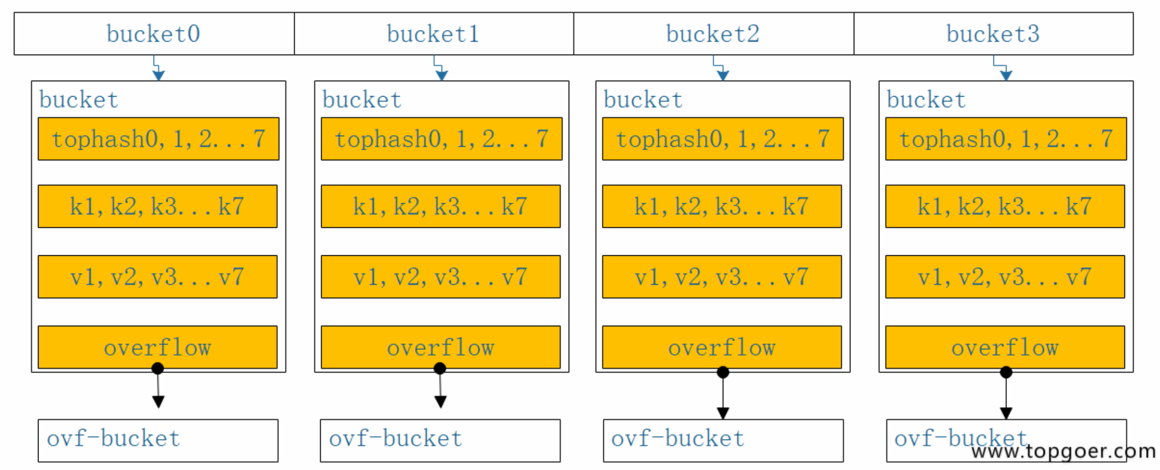
    注意：将所有键打包在一起，然后将所有值打包在一起，   使得代码比交替键/值/键/值/更复杂。但它允许*//我们消除可能需要的填充，   例如map[int64]int8./后面跟一个溢出指针}*

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。



www.topgoer.com

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.bucketsiz
e)))
    //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.keys
```

```
ize))
                    val = add( unsafe.Pointer(b),
                    dataOffset+bucketCnt*uintptr(t.keysize)+i*uintptr(t.valuesiz
e) )
                }
                //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

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。

类型别名和自定义类型

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。

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
```

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。

```go
//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。

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。

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：

```go
    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

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
```

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", ag
```

```
e: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。

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)
}
```

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}
```

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。

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}
}
```

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:1
8}
```

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}
```

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:1
8}
```

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。
```

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
```

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)
    }
}
```

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)
```

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。

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
}
```

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
}
```

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。
```

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。

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。

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"}}
}
```

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。

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
}
```

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)
```

```go
}

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!
}
```

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）。

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。

```go
//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)
}
```

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"}
}
```

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)
}
```

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)
}
```

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]])
    }
}
```

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

流程控制

条件语句if

条件语句switch

条件语句select

循环语句for

循环语句range

循环控制Goto、Break、Continue

条件语句if

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：

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
```

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
```

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
```

条件语句switch

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
```

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
```

条件语句select

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：

```go
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

```go
    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。

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
```

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
}
```

循环语句for

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) {}
    n--
    println(s[n])        // 替代 for (; n > 0;) {}
}

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
```

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
```

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");
    }
}
```

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

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：

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
```

函数

函数定义

参数

返回值

匿名函数

闭包、递归

延迟调用（defer）

异常处理

单元测试

压力测试

函数定义

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)。
```

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
```

参数

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)
}
```

返回值

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
```

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!
```

闭包、递归

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。

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))
}
```

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
```

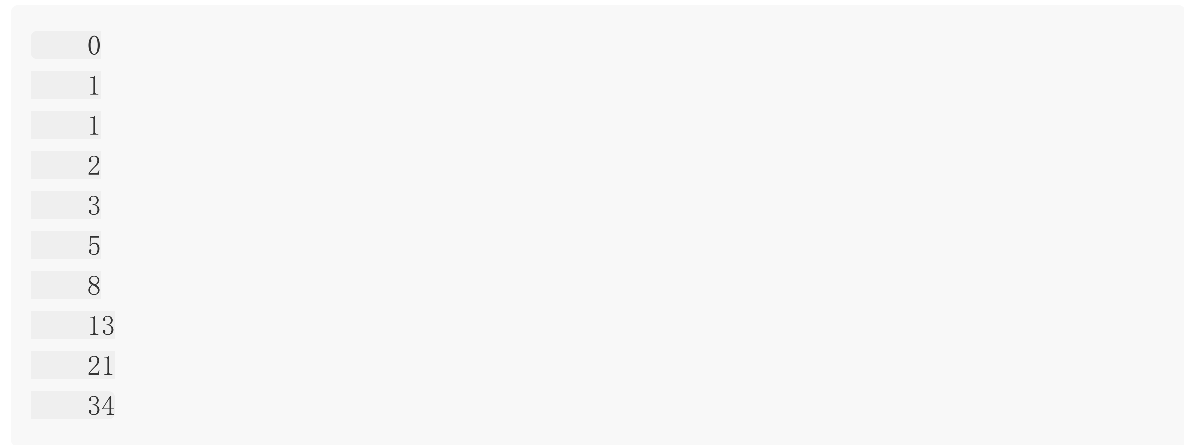
斐波那契数列(Fibonacci)

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
```

延迟调用（defer）

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
```

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()
}
```

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。

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。

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
```

测试函数

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()
```

```go
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
```

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：

```go
// 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：

```go
// 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

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
```

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。

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。

基准测试

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()
```

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/o
p           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/o
p           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。

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)
}
```

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")
    }
}
```

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。

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）。

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。

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) // 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
```

示例函数

###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() {
    // ...
}
```

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。
```

func **ToUpper**

```
func ToUpper(s string) string
```

ToUpper returns a copy of the string s with all Unicode letters mapped to their upper case.

▾ Example

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	fmt.Println(strings.ToUpper("Gopher"))
}
```

Run Format Share

www.topgoer.com

压力测试

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/...
```

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 fun
ction
        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
```

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/o
p          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。

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

方法定义

匿名字段

方法集

表达式

自定义error

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。

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
```

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
```

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}
```

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
```

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 derefer
ence

    // (Data)(nil).TestValue()  // cannot convert nil to type Data
    // Data.TestValue(nil)      // cannot use nil as type Data in function argum
ent
}
```

自定义error

抛异常和处理异常

###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()
}
```

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)
    }
}
```

自定义error:

```go
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

匿名字段

接口

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"

//      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：

```
    {0xc00006a360 1 bj}
    5lmh
    5lmh
```

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。

接口

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。

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。

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。

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。

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
}
```

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 {}
```

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` 。

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。

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 *Student) 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))
}
```

类型与接口的关系

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()
}
```

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")
}
```

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()
}
```

空接口

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)
}
```

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)
```

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？

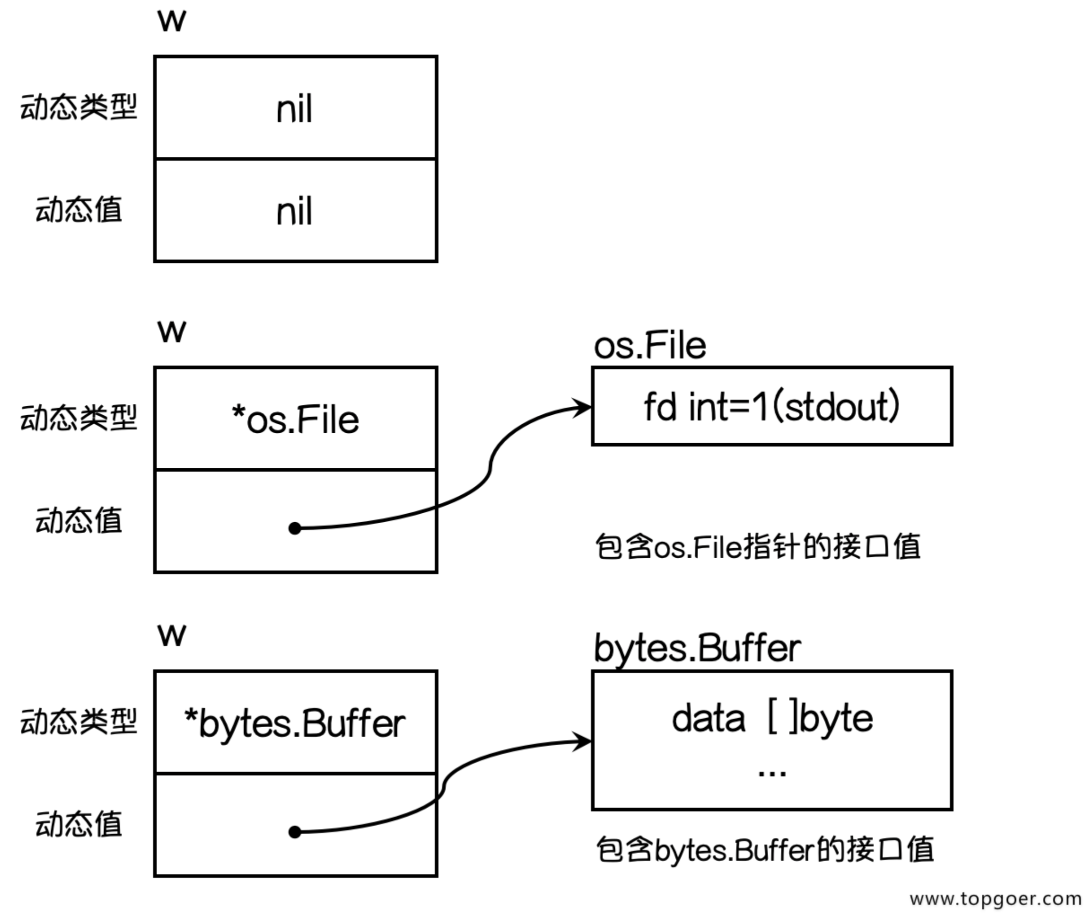
接口值

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。

网络编程

互联网协议介绍

socket编程

http编程

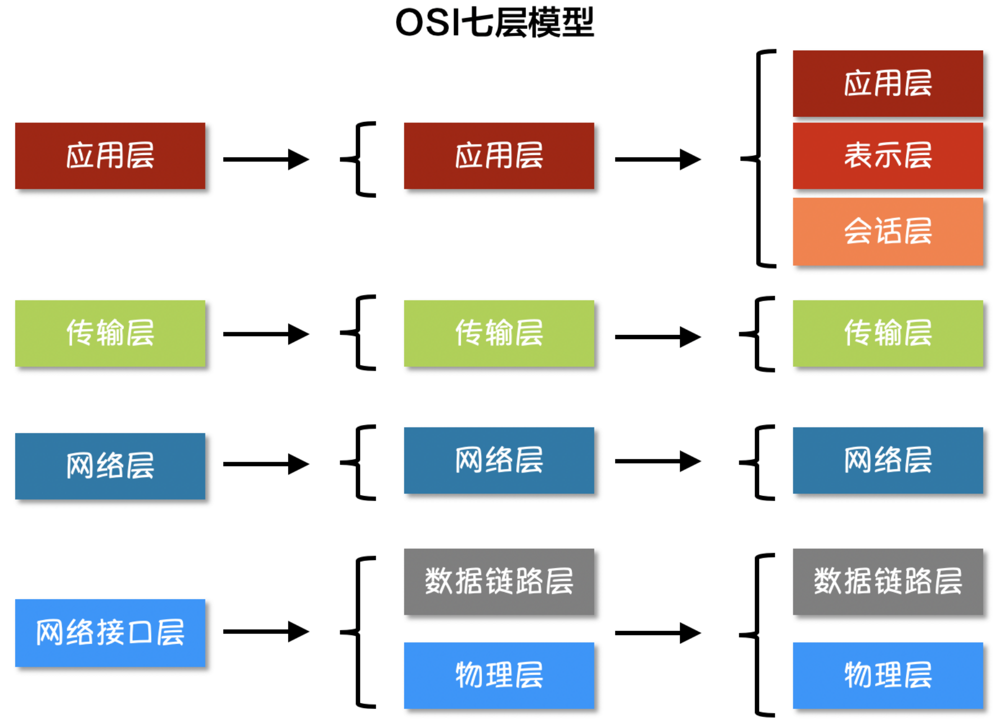
WebSocket编程

互联网协议介绍

互联网的核心是一系列协议，总称为"互联网协议"（Internet Protocol Suite），正是这一些协议规定了电脑如何连接和组网。我们理解了这些协议，就理解了互联网的原理。由于这些协议太过庞大和复杂，没有办法在这里一概而全，只能介绍一下我们日常开发中接触较多的几个协议。

互联网分层模型

互联网的逻辑实现被分为好几层。每一层都有自己的功能，就像建筑物一样，每一层都靠下一层支持。用户接触到的只是最上面的那一层，根本不会感觉到下面的几层。要理解互联网就需要自下而上理解每一层的实现的功能。



如上图所示，互联网按照不同的模型划分会有不用的分层，但是不论按照什么模型去划分，越往上的层越靠近用户，越往下的层越靠近硬件。在软件开发中我们使用最多的是上图中将互联网划分为五个分层的模型。

接下来我们一层一层的自底向上介绍一下每一层。

物理层

我们的电脑要与外界互联网通信，需要先把电脑连接网络，我们可以用双绞线、光纤、无线电波等方式。这就叫做"实物理层"，它就是把电脑连接起来的物理手段。它主要规定了网络的一

些电气特性，作用是负责传送0和1的电信号。

数据链路层

单纯的0和1没有任何意义，所以我们使用者会为其赋予一些特定的含义，规定解读电信号的方式：例如：多少个电信号算一组？每个信号位有何意义？这就是”数据链接层”的功能，它在”物理层”的上方，确定了物理层传输的0和1的分组方式及代表的意义。早期的时候，每家公司都有自己的电信号分组方式。逐渐地，一种叫做”以太网”（Ethernet）的协议，占据了主导地位。

以太网规定，一组电信号构成一个数据包，叫做”帧”（Frame）。每一帧分成两个部分：标头（Head）和数据（Data）。其中”标头”包含数据包的一些说明项，比如发送者、接受者、数据类型等等；”数据”则是数据包的具体内容。”标头”的长度，固定为18字节。”数据”的长度，最短为46字节，最长为1500字节。因此，整个”帧”最短为64字节，最长为1518字节。如果数据很长，就必须分割成多个帧进行发送。

那么，发送者和接受者是如何标识呢？以太网规定，连入网络的所有设备都必须具有”网卡”接口。数据包必须是从一块网卡，传送到另一块网卡。网卡的地址，就是数据包的发送地址和接收地址，这叫做MAC地址。每块网卡出厂的时候，都有一个全世界独一无二的MAC地址，长度是48个二进制位，通常用12个十六进制数表示。前6个十六进制数是厂商编号，后6个是该厂商的网卡流水号。有了MAC地址，就可以定位网卡和数据包的路径了。

我们会通过ARP协议来获取接受方的MAC地址，有了MAC地址之后，如何把数据准确的发送给接收方呢？其实这里以太网采用了一种很”原始”的方式，它不是把数据包准确送到接收方，而是向本网络内所有计算机都发送，让每台计算机读取这个包的”标头”，找到接收方的MAC地址，然后与自身的MAC地址相比较，如果两者相同，就接受这个包，做进一步处理，否则就丢弃这个包。这种发送方式就叫做”广播”（broadcasting）。

网络层

按照以太网协议的规则我们可以依靠MAC地址来向外发送数据。理论上依靠MAC地址，你电脑的网卡就可以找到身在世界另一个角落的某台电脑的网卡了，但是这种做法有一个重大缺陷就是以太网采用广播方式发送数据包，所有成员人手一”包”，不仅效率低，而且发送的数据只能局限在发送者所在的子网络。也就是说如果两台计算机不在同一个子网络，广播是传不过去的。这种设计是合理且必要的，因为如果互联网上每一台计算机都会收到互联网上收发的所有数据包，那是不现实的。

因此，必须找到一种方法区分哪些MAC地址属于同一个子网络，哪些不是。如果是同一个子网络，就采用广播方式发送，否则就采用”路由”方式发送。这就导致了”网络层”的诞生。它的作用是引进一套新的地址，使得我们能够区分不同的计算机是否属于同一个子网络。这套地址就叫做”网络地址”，简称”网址”。

“网络层”出现以后，每台计算机有了两种地址，一种是MAC地址，另一种是网络地址。两种地址之间没有任何联系，MAC地址是绑定在网卡上的，网络地址则是网络管理员分配的。网络地址帮助我们确定计算机所在的子网络，MAC地址则将数据包送到该子网络中的目标网卡。因此，从逻辑上可以推断，必定是先处理网络地址，然后再处理MAC地址。

规定网络地址的协议，叫做IP协议。它所定义的地址，就被称为IP地址。目前，广泛采用的是IP协议第四版，简称IPv4。IPv4这个版本规定，网络地址由32个二进制位组成，我们通常习惯用分成四段的十进制数表示IP地址，从0.0.0.0一直到255.255.255.255。

根据IP协议发送的数据，就叫做IP数据包。IP数据包也分为”标头”和”数据”两个部分：”标头”部分主要包括版本、长度、IP地址等信息，”数据”部分则是IP数据包的具体内容。IP数据包的”标头”部分的长度为20到60字节，整个数据包的总长度最大为65535字节。

传输层

有了MAC地址和IP地址，我们已经可以在互联网上任意两台主机上建立通信。但问题是同一台主机上会有许多程序都需要用网络收发数据，比如QQ和浏览器这两个程序都需要连接互联网并收发数据，我们如何区分某个数据包到底是归哪个程序的呢？也就是说，我们还需要一个参数，表示这个数据包到底供哪个程序（进程）使用。这个参数就叫做”端口”（port），它其实是每一个使用网卡的程序的编号。每个数据包都发到主机的特定端口，所以不同的程序就能取到自己所需要的数据。

“端口”是0到65535之间的一个整数，正好16个二进制位。0到1023的端口被系统占用，用户只能选用大于1023的端口。有了IP和端口我们就能实现唯一确定互联网上一个程序，进而实现网络间的程序通信。

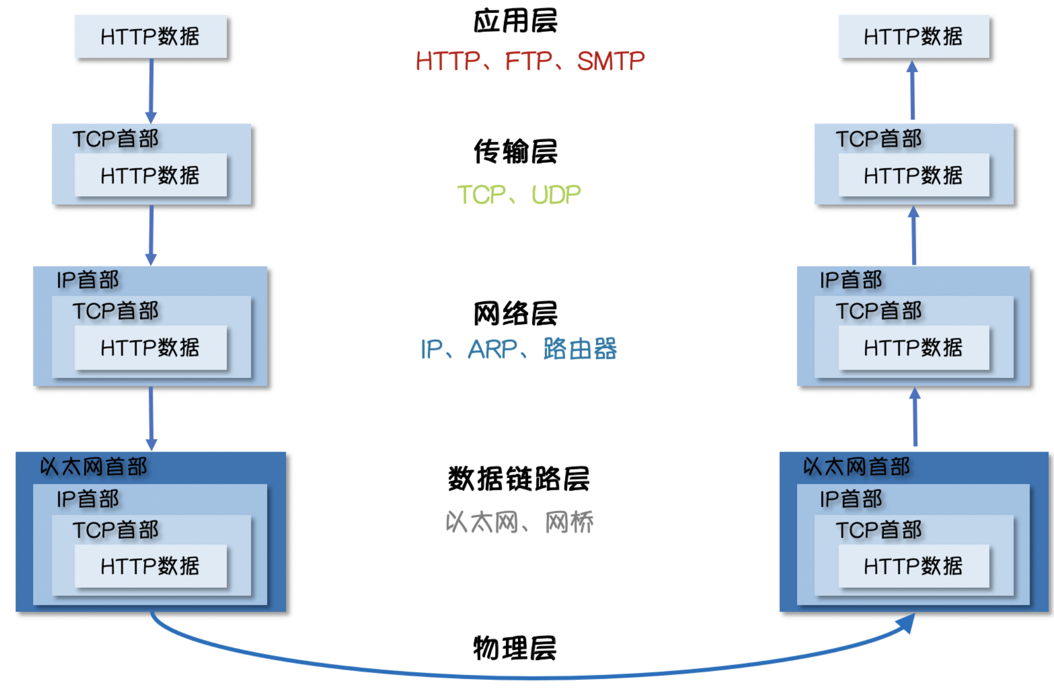
我们必须在数据包中加入端口信息，这就需要新的协议。最简单的实现叫做UDP协议，它的格式几乎就是在数据前面，加上端口号。UDP数据包，也是由”标头”和”数据”两部分组成：”标头”部分主要定义了发出端口和接收端口，”数据”部分就是具体的内容。UDP数据包非常简单，”标头”部分一共只有8个字节，总长度不超过65,535字节，正好放进一个IP数据包。

UDP协议的优点是比较简单，容易实现，但是缺点是可靠性较差，一旦数据包发出，无法知道对方是否收到。为了解决这个问题，提高网络可靠性，TCP协议就诞生了。TCP协议能够确保数据不会遗失。它的缺点是过程复杂、实现困难、消耗较多的资源。TCP数据包没有长度限制，理论上可以无限长，但是为了保证网络的效率，通常TCP数据包的长度不会超过IP数据包的长度，以确保单个TCP数据包不必再分割。

应用层

应用程序收到”传输层”的数据，接下来就要对数据进行解包。由于互联网是开放架构，数据来源五花八门，必须事先规定好通信的数据格式，否则接收方根本无法获得真正发送的数据内容。”应用层”的作用就是规定应用程序使用的数据格式，例如我们TCP协议之上常见的Email、HTTP、FTP等协议，这些协议就组成了互联网协议的应用层。

如下图所示，发送方的HTTP数据经过互联网的传输过程中会依次添加各层协议的标头信息，接收方收到数据包之后再依次根据协议解包得到数据。


HTTP数据
应用层
HTTP、FTP、SMTP
HTTP数据
TCP首部
HTTP数据
传输层
TCP、UDP
TCP首部
HTTP数据
IP首部
TCP首部
HTTP数据
网络层
IP、ARP、路由器
IP首部
TCP首部
HTTP数据
以太网首部
IP首部
TCP首部
HTTP数据
数据链路层
以太网、网桥
以太网首部
IP首部
TCP首部
HTTP数据
物理层
双绞线、光纤、无线电波
www.topgoer.com

socket编程

socket图解

TCP编程

UDP编程

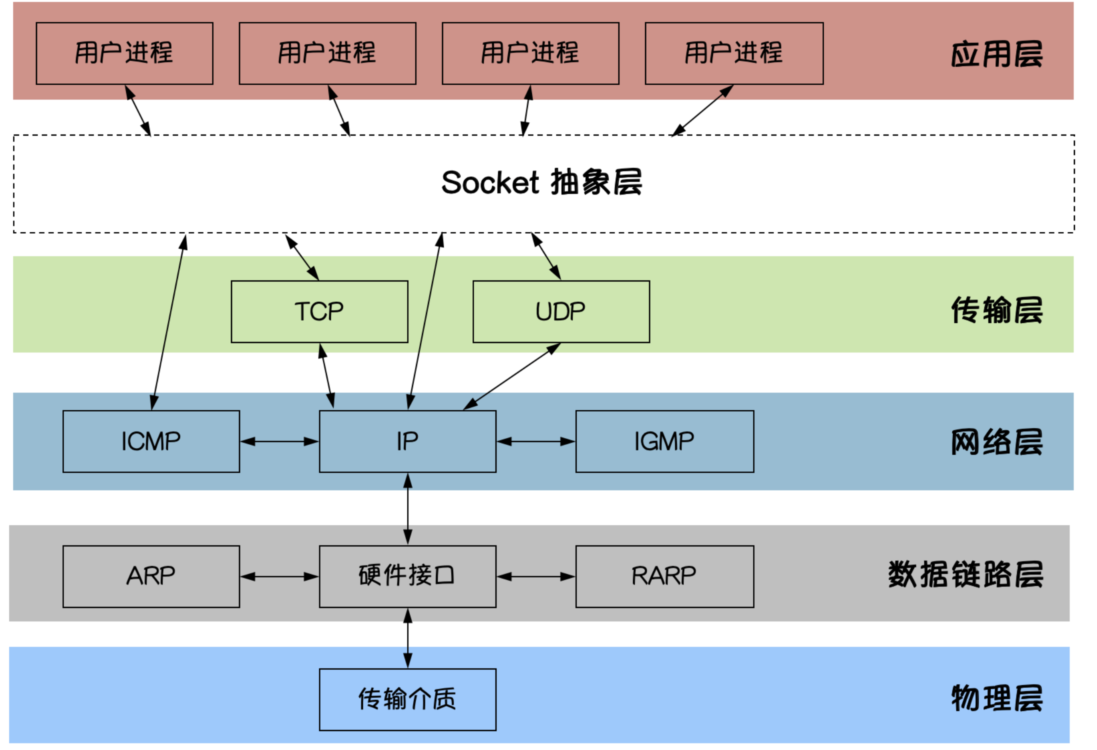
TCP黏包

socket图解

Socket是BSD UNIX的进程通信机制，通常也称作"套接字"，用于描述IP地址和端口，是一个通信链的句柄。Socket可以理解为TCP/IP网络的API，它定义了许多函数或例程，程序员可以用它们来开发TCP/IP网络上的应用程序。电脑上运行的应用程序通常通过"套接字"向网络发出请求或者应答网络请求。

socket图解

Socket是应用层与TCP/IP协议族通信的中间软件抽象层。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket后面，对用户来说只需要调用Socket规定的相关函数，让Socket去组织符合指定的协议数据然后进行通信。



- Socket又称“套接字”，应用程序通常通过“套接字”向网络发出请求或者应答网络请求
- 常用的Socket类型有两种：流式Socket和数据报式Socket，流式是一种面向连接的Socket，针对于面向连接的TCP服务应用，数据报式Socket是一种无连接的Socket，针对于无连接的UDP服务应用
- TCP：比较靠谱，面向连接，比较慢
- UDP：不是太靠谱，比较快

举个例子：TCP就像货到付款的快递，送到家还必须见到你人才算一整套流程。UDP就像某快递快递柜一扔就走管你收到收不到，一般直播用UDP。

TCP编程

Go语言实现TCP通信

TCP协议

TCP/IP(Transmission Control Protocol/Internet Protocol) 即传输控制协议/网间协议，是一种面向连接（连接导向）的、可靠的、基于字节流的传输层（Transport layer）通信协议，因为是面向连接的协议，数据像水流一样传输，会存在黏包问题。

TCP服务端

一个TCP服务端可以同时连接很多个客户端，例如世界各地的用户使用自己电脑上的浏览器访问淘宝网。因为Go语言中创建多个goroutine实现并发非常方便和高效，所以我们可以每建立一次链接就创建一个goroutine去处理。

TCP服务端程序的处理流程:

```
    1.监听端口
    2.接收客户端请求建立链接
    3.创建goroutine处理链接。
```

我们使用Go语言的net包实现的TCP服务端代码如下：

```
// tcp/server/main.go

// TCP server端

// 处理函数
func process(conn net.Conn) {
    defer conn.Close() // 关闭连接
    for {
        reader := bufio.NewReader(conn)
        var buf [128]byte
        n, err := reader.Read(buf[:]) // 读取数据
        if err != nil {
            fmt.Println("read from client failed, err:", err)
            break
        }
        recvStr := string(buf[:n])
        fmt.Println("收到client端发来的数据：", recvStr)
        conn.Write([]byte(recvStr)) // 发送数据
```

```
	}
}

func main() {
	listen, err := net.Listen("tcp", "127.0.0.1:20000")
	if err != nil {
		fmt.Println("listen failed, err:", err)
		return
	}
	for {
		conn, err := listen.Accept() // 建立连接
		if err != nil {
			fmt.Println("accept failed, err:", err)
			continue
		}
		go process(conn) // 启动一个goroutine处理连接
	}
}
```

将上面的代码保存之后编译成server或server.exe可执行文件。

TCP客户端

一个TCP客户端进行TCP通信的流程如下：

```
	1.建立与服务端的链接
	2.进行数据收发
	3.关闭链接
```

使用Go语言的net包实现的TCP客户端代码如下：

```
// tcp/client/main.go

// 客户端
func main() {
	conn, err := net.Dial("tcp", "127.0.0.1:20000")
	if err != nil {
		fmt.Println("err :", err)
		return
	}
	defer conn.Close() // 关闭连接
	inputReader := bufio.NewReader(os.Stdin)
	for {
		input, _ := inputReader.ReadString('\n') // 读取用户输入
		inputInfo := strings.Trim(input, "\r\n")
```

```go
		if strings.ToUpper(inputInfo) == "Q" { // 如果输入q就退出
			return
		}
		_, err = conn.Write([]byte(inputInfo)) // 发送数据
		if err != nil {
			return
		}
		buf := [512]byte{}
		n, err := conn.Read(buf[:])
		if err != nil {
			fmt.Println("recv failed, err:", err)
			return
		}
		fmt.Println(string(buf[:n]))
	}
}
```

将上面的代码编译成client或client.exe可执行文件，先启动server端再启动client端，在client端输入任意内容回车之后就能够在server端看到client端发送的数据，从而实现TCP通信。

UDP编程

Go语言实现UDP通信

UDP协议

UDP协议（User Datagram Protocol）中文名称是用户数据报协议，是OSI（Open System Interconnection，开放式系统互联）参考模型中一种无连接的传输层协议，不需要建立连接就能直接进行数据发送和接收，属于不可靠的、没有时序的通信，但是UDP协议的实时性比较好，通常用于视频直播相关领域。

UDP服务端

使用Go语言的net包实现的UDP服务端代码如下：

```
// UDP/server/main.go

// UDP server端
func main() {
	listen, err := net.ListenUDP("udp", &net.UDPAddr{
		IP:   net.IPv4(0, 0, 0, 0),
		Port: 30000,
	})
	if err != nil {
		fmt.Println("listen failed, err:", err)
		return
	}
	defer listen.Close()
	for {
		var data [1024]byte
		n, addr, err := listen.ReadFromUDP(data[:]) // 接收数据
		if err != nil {
			fmt.Println("read udp failed, err:", err)
			continue
		}
		fmt.Printf("data:%v addr:%v count:%v\n", string(data[:n]), addr, n)
		_, err = listen.WriteToUDP(data[:n], addr) // 发送数据
		if err != nil {
			fmt.Println("write to udp failed, err:", err)
			continue
		}
	}
}
```

UDP客户端

使用Go语言的net包实现的UDP客户端代码如下：

```go
// UDP 客户端
func main() {
	socket, err := net.DialUDP("udp", nil, &net.UDPAddr{
		IP:   net.IPv4(0, 0, 0, 0),
		Port: 30000,
	})
	if err != nil {
		fmt.Println("连接服务端失败，err:", err)
		return
	}
	defer socket.Close()
	sendData := []byte("Hello server")
	_, err = socket.Write(sendData) // 发送数据
	if err != nil {
		fmt.Println("发送数据失败，err:", err)
		return
	}
	data := make([]byte, 4096)
	n, remoteAddr, err := socket.ReadFromUDP(data) // 接收数据
	if err != nil {
		fmt.Println("接收数据失败，err:", err)
		return
	}
	fmt.Printf("recv:%v addr:%v count:%v\n", string(data[:n]), remoteAddr, n)
}
```

TCP黏包

服务端代码如下：

```
// socket_stick/server/main.go

func process(conn net.Conn) {
	defer conn.Close()
	reader := bufio.NewReader(conn)
	var buf [1024]byte
	for {
		n, err := reader.Read(buf[:])
		if err == io.EOF {
			break
		}
		if err != nil {
			fmt.Println("read from client failed, err:", err)
			break
		}
		recvStr := string(buf[:n])
		fmt.Println("收到client发来的数据：", recvStr)
	}
}

func main() {

	listen, err := net.Listen("tcp", "127.0.0.1:30000")
	if err != nil {
		fmt.Println("listen failed, err:", err)
		return
	}
	defer listen.Close()
	for {
		conn, err := listen.Accept()
		if err != nil {
			fmt.Println("accept failed, err:", err)
			continue
		}
		go process(conn)
	}
}
```

客户端代码如下：

```
// socket_stick/client/main.go

func main() {
	conn, err := net.Dial("tcp", "127.0.0.1:30000")
	if err != nil {
		fmt.Println("dial failed, err", err)
		return
	}
	defer conn.Close()
	for i := 0; i < 20; i++ {
		msg := `Hello, Hello. How are you?`
		conn.Write([]byte(msg))
	}
}
```

将上面的代码保存后，分别编译。先启动服务端再启动客户端，可以看到服务端输出结果如下：

```
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?Hello, Hello. How are you?
收到client发来的数据： Hello, Hello. How are you?Hello, Hello. How are you?
```

客户端分10次发送的数据，在服务端并没有成功的输出10次，而是多条数据“粘”到了一起。

为什么会出现粘包

主要原因就是tcp数据传递模式是流模式，在保持长连接的时候可以进行多次的收和发。

“粘包”可发生在发送端也可发生在接收端：

```
	1.由Nagle算法造成的发送端的粘包：Nagle算法是一种改善网络传输效率的算法。简单来说就是当我们提交一段数据给TCP发送时，TCP并不立刻发送此段数据，而是等待一小段时间看看在等待期间是否还有要发送的数据，若有则会一次把这两段数据发送出去。
	2.接收端接收不及时造成的接收端粘包：TCP会把接收到的数据存在自己的缓冲区中，然后通知应用层取数据。当应用层由于某些原因不能及时的把TCP的数据取出来，就会造成TCP缓冲区中存放了几段数据。
```

解决办法

出现"粘包"的关键在于接收方不确定将要传输的数据包的大小，因此我们可以对数据包进行封包和拆包的操作。

封包：封包就是给一段数据加上包头，这样一来数据包就分为包头和包体两部分内容了(过滤非法包时封包会加入"包尾"内容)。包头部分的长度是固定的，并且它存储了包体的长度，根据包头长度固定以及包头中含有包体长度的变量就能正确的拆分出一个完整的数据包。

我们可以自己定义一个协议，比如数据包的前4个字节为包头，里面存储的是发送的数据的长度。

```
// socket_stick/proto/proto.go
package proto

import (
	"bufio"
	"bytes"
	"encoding/binary"
)

// Encode 将消息编码
func Encode(message string) ([]byte, error) {
	// 读取消息的长度，转换成int32类型（占4个字节）
	var length = int32(len(message))
	var pkg = new(bytes.Buffer)
	// 写入消息头
	err := binary.Write(pkg, binary.LittleEndian, length)
	if err != nil {
		return nil, err
	}
	// 写入消息实体
	err = binary.Write(pkg, binary.LittleEndian, []byte(message))
	if err != nil {
		return nil, err
	}
	return pkg.Bytes(), nil
}

// Decode 解码消息
func Decode(reader *bufio.Reader) (string, error) {
	// 读取消息的长度
	lengthByte, _ := reader.Peek(4) // 读取前4个字节的数据
	lengthBuff := bytes.NewBuffer(lengthByte)
	var length int32
	err := binary.Read(lengthBuff, binary.LittleEndian, &length)
```

```go
    if err != nil {
        return "", err
    }
    // Buffered返回缓冲中现有的可读取的字节数。
    if int32(reader.Buffered()) < length+4 {
        return "", err
    }

    // 读取真正的消息数据
    pack := make([]byte, int(4+length))
    _, err = reader.Read(pack)
    if err != nil {
        return "", err
    }
    return string(pack[4:]), nil
}
```

接下来在服务端和客户端分别使用上面定义的proto包的Decode和Encode函数处理数据。

服务端代码如下：

```go
// socket_stick/server2/main.go

func process(conn net.Conn) {
    defer conn.Close()
    reader := bufio.NewReader(conn)
    for {
        msg, err := proto.Decode(reader)
        if err == io.EOF {
            return
        }
        if err != nil {
            fmt.Println("decode msg failed, err:", err)
            return
        }
        fmt.Println("收到client发来的数据：", msg)
    }
}

func main() {

    listen, err := net.Listen("tcp", "127.0.0.1:30000")
    if err != nil {
        fmt.Println("listen failed, err:", err)
        return
```

```go
	}
	defer listen.Close()
	for {
		conn, err := listen.Accept()
		if err != nil {
			fmt.Println("accept failed, err:", err)
			continue
		}
		go process(conn)
	}
}
```

客户端代码如下：

```go
// socket_stick/client2/main.go

func main() {
	conn, err := net.Dial("tcp", "127.0.0.1:30000")
	if err != nil {
		fmt.Println("dial failed, err", err)
		return
	}
	defer conn.Close()
	for i := 0; i < 20; i++ {
		msg := `Hello, Hello. How are you?`
		data, err := proto.Encode(msg)
		if err != nil {
			fmt.Println("encode msg failed, err:", err)
			return
		}
		conn.Write(data)
	}
}
```

http编程

web工作流程

- Web服务器的工作原理可以简单地归纳为
  - 客户机通过TCP/IP协议建立到服务器的TCP连接
  - 客户端向服务器发送HTTP协议请求包，请求服务器里的资源文档
  - 服务器向客户机发送HTTP协议应答包，如果请求的资源包含有动态语言的内容，那么服务器会调用动态语言的解释引擎负责处理“动态内容”，并将处理得到的数据返回给客户端
  - 客户机与服务器断开。由客户端解释HTML文档，在客户端屏幕上渲染图形结果

HTTP协议

- 超文本传输协议(HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络协议，它详细规定了浏览器和万维网服务器之间互相通信的规则，通过因特网传送万维网文档的数据传送协议
- HTTP协议通常承载于TCP协议之上

HTTP服务端

```go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    //http://127.0.0.1:8000/go
    // 单独写回调函数
    http.HandleFunc("/go", myHandler)
    //http.HandleFunc("/ungo",myHandler2 )
    // addr：监听的地址
    // handler：回调函数
    http.ListenAndServe("127.0.0.1:8000", nil)
}
```

```
// handler函数
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Println(r.RemoteAddr, "连接成功")
    // 请求方式: GET POST DELETE PUT UPDATE
    fmt.Println("method:", r.Method)
    // /go
    fmt.Println("url:", r.URL.Path)
    fmt.Println("header:", r.Header)
    fmt.Println("body:", r.Body)
    // 回复
    w.Write([]byte("www.51mh.com"))
}
```

HTTP客户端

```
package main

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

func main() {
    //resp, _ := http.Get("http://www.baidu.com")
    //fmt.Println(resp)
    resp, _ := http.Get("http://127.0.0.1:8000/go")
    defer resp.Body.Close()
    // 200 OK
    fmt.Println(resp.Status)
    fmt.Println(resp.Header)

    buf := make([]byte, 1024)
    for {
        // 接收服务端信息
        n, err := resp.Body.Read(buf)
        if err != nil && err != io.EOF {
            fmt.Println(err)
            return
        } else {
            fmt.Println("读取完毕")
            res := string(buf[:n])
            fmt.Println(res)
            break
        }
```

```
        }
}
```

WebSocket编程

webSocket是什么

- WebSocket是一种在单个TCP连接上进行全双工通信的协议
- WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据
- 在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输
- 需要安装第三方包：
    - cmd中：go get -u -v github.com/gorilla/websocket

举个聊天室的小例子

在同一级目录下新建四个go文件 `connection.go|data.go|hub.go|server.go`

运行

```
go run server.go hub.go data.go connection.go
```

运行之后执行local.html文件

server.go文件代码

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func main() {
    router := mux.NewRouter()
    go h.run()
    router.HandleFunc("/ws", myws)
    if err := http.ListenAndServe("127.0.0.1:8080", router); err != nil {
        fmt.Println("err:", err)
```

```
    }
}
```

hub.go文件代码

```
package main

import "encoding/json"

var h = hub{
    c: make(map[*connection]bool),
    u: make(chan *connection),
    b: make(chan []byte),
    r: make(chan *connection),
}

type hub struct {
    c map[*connection]bool
    b chan []byte
    r chan *connection
    u chan *connection
}

func (h *hub) run() {
    for {
        select {
        case c := <-h.r:
            h.c[c] = true
            c.data.Ip = c.ws.RemoteAddr().String()
            c.data.Type = "handshake"
            c.data.UserList = user_list
            data_b, _ := json.Marshal(c.data)
            c.sc <- data_b
        case c := <-h.u:
            if _, ok := h.c[c]; ok {
                delete(h.c, c)
                close(c.sc)
            }
        case data := <-h.b:
            for c := range h.c {
                select {
                case c.sc <- data:
                default:
                    delete(h.c, c)
                    close(c.sc)
```

```
                }
            }
        }
    }
}
```

data.go文件代码

```
package main

type Data struct {
    Ip       string   `json:"ip"`
    User     string   `json:"user"`
    From     string   `json:"from"`
    Type     string   `json:"type"`
    Content  string   `json:"content"`
    UserList []string `json:"user_list"`
}
```

connection.go文件代码

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/gorilla/websocket"
)

type connection struct {
    ws   *websocket.Conn
    sc   chan []byte
    data *Data
}

var wu = &websocket.Upgrader{ReadBufferSize: 512,
    WriteBufferSize: 512, CheckOrigin: func(r *http.Request) bool { return true
 }}

func myws(w http.ResponseWriter, r *http.Request) {
    ws, err := wu.Upgrade(w, r, nil)
    if err != nil {
```

```
        return
    }
    c := &connection{sc: make(chan []byte, 256), ws: ws, data: &Data{}}
    h.r <- c
    go c.writer()
    c.reader()
    defer func() {
        c.data.Type = "logout"
        user_list = del(user_list, c.data.User)
        c.data.UserList = user_list
        c.data.Content = c.data.User
        data_b, _ := json.Marshal(c.data)
        h.b <- data_b
        h.r <- c
    }()
}

func (c *connection) writer() {
    for message := range c.sc {
        c.ws.WriteMessage(websocket.TextMessage, message)
    }
    c.ws.Close()
}

var user_list = []string{}

func (c *connection) reader() {
    for {
        _, message, err := c.ws.ReadMessage()
        if err != nil {
            h.r <- c
            break
        }
        json.Unmarshal(message, &c.data)
        switch c.data.Type {
        case "login":
            c.data.User = c.data.Content
            c.data.From = c.data.User
            user_list = append(user_list, c.data.User)
            c.data.UserList = user_list
            data_b, _ := json.Marshal(c.data)
            h.b <- data_b
        case "user":
            c.data.Type = "user"
            data_b, _ := json.Marshal(c.data)
            h.b <- data_b
```

```
        case "logout":
            c.data.Type = "logout"
            user_list = del(user_list, c.data.User)
            data_b, _ := json.Marshal(c.data)
            h.b <- data_b
            h.r <- c
        default:
            fmt.Print("========default================")
        }
    }
}

func del(slice []string, user string) []string {
    count := len(slice)
    if count == 0 {
        return slice
    }
    if count == 1 && slice[0] == user {
        return []string{}
    }
    var n_slice = []string{}
    for i := range slice {
        if slice[i] == user && i == count {
            return slice[:count]
        } else if slice[i] == user {
            n_slice = append(slice[:i], slice[i+1:]...)
            break
        }
    }
    fmt.Println(n_slice)
    return n_slice
}
```

local.html文件代码

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <style>
        p {
            text-align: left;
            padding-left: 20px;
        }
```

```
    </style>
</head>
<body>
<div style="width: 800px;height: 600px;margin: 30px auto;text-align: center">
    <h1>www.51mh.comy演示聊天室</h1>
    <div style="width: 800px;border: 1px solid gray;height: 300px;">
        <div style="width: 200px;height: 300px;float: left;text-align: left;">
            <p><span>当前在线:</span><span id="user_num">0</span></p>
            <div id="user_list" style="overflow: auto;">
            </div>
        </div>
        <div id="msg_list" style="width: 598px;border:  1px solid gray; height:
 300px;overflow: scroll;float: left;">
        </div>
    </div>
    <br>
    <textarea id="msg_box" rows="6" cols="50" onkeydown="confirm(event)"></texta
rea><br>
    <input type="button" value="发送" onclick="send()">
</div>
</body>
</html>
<script type="text/javascript">
    var uname = prompt('请输入用户名', 'user' + uuid(8, 16));
    var ws = new WebSocket("ws://127.0.0.1:8080/ws");
    ws.onopen = function () {
        var data = "系统消息：建立连接成功";
        listMsg(data);
    };
    ws.onmessage = function (e) {
        var msg = JSON.parse(e.data);
        var sender, user_name, name_list, change_type;
        switch (msg.type) {
            case 'system':
                sender = '系统消息: ';
                break;
            case 'user':
                sender = msg.from + ': ';
                break;
            case 'handshake':
                var user_info = {'type': 'login', 'content': uname};
                sendMsg(user_info);
                return;
            case 'login':
            case 'logout':
                user_name = msg.content;
```

```
                name_list = msg.user_list;
                change_type = msg.type;
                dealUser(user_name, change_type, name_list);
                return;
        }
        var data = sender + msg.content;
        listMsg(data);
    };
    ws.onerror = function () {
        var data = "系统消息 : 出错了,请退出重试.";
        listMsg(data);
    };
    function confirm(event) {
        var key_num = event.keyCode;
        if (13 == key_num) {
            send();
        } else {
            return false;
        }
    }
    function send() {
        var msg_box = document.getElementById("msg_box");
        var content = msg_box.value;
        var reg = new RegExp("\r\n", "g");
        content = content.replace(reg, "");
        var msg = {'content': content.trim(), 'type': 'user'};
        sendMsg(msg);
        msg_box.value = '';
    }
    function listMsg(data) {
        var msg_list = document.getElementById("msg_list");
        var msg = document.createElement("p");
        msg.innerHTML = data;
        msg_list.appendChild(msg);
        msg_list.scrollTop = msg_list.scrollHeight;
    }
    function dealUser(user_name, type, name_list) {
        var user_list = document.getElementById("user_list");
        var user_num = document.getElementById("user_num");
        while(user_list.hasChildNodes()) {
            user_list.removeChild(user_list.firstChild);
        }
        for (var index in name_list) {
            var user = document.createElement("p");
            user.innerHTML = name_list[index];
            user_list.appendChild(user);
```

```
        }
        user_num.innerHTML = name_list.length;
        user_list.scrollTop = user_list.scrollHeight;
        var change = type == 'login' ? '上线' : '下线';
        var data = '系统消息: ' + user_name + ' 已' + change;
        listMsg(data);
    }
    function sendMsg(msg) {
        var data = JSON.stringify(msg);
        ws.send(data);
    }
    function uuid(len, radix) {
        var chars = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
xyz'.split('');
        var uuid = [], i;
        radix = radix || chars.length;
        if (len) {
            for (i = 0; i < len; i++) uuid[i] = chars[0 | Math.random() * radi
x];
        } else {
            var r;
            uuid[8] = uuid[13] = uuid[18] = uuid[23] = '-';
            uuid[14] = '4';
            for (i = 0; i < 36; i++) {
                if (!uuid[i]) {
                    r = 0 | Math.random() * 16;
                    uuid[i] = chars[(i == 19) ? (r & 0x3) | 0x8 : r];
                }
            }
        }
        return uuid.join('');
    }
</script>
```

并发编程

并发介绍

Goroutine

runtime包

Channel

Goroutine池

定时器

select

并发安全和锁

Sync

原子操作(atomic包)

GMP 原理与调度

爬虫小案例

并发介绍

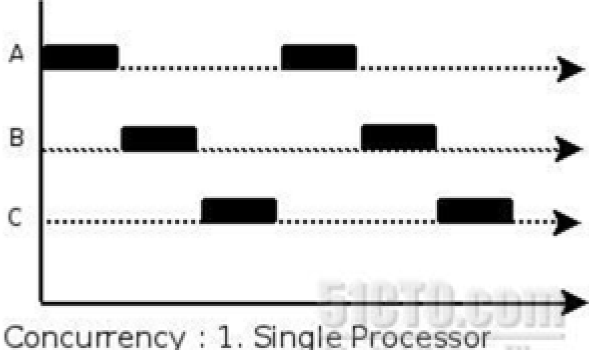
进程和线程

```
    A. 进程是程序在操作系统中的一次执行过程，系统进行资源分配和调度的一个独立单位。
    B. 线程是进程的一个执行实体,是CPU调度和分派的基本单位,它是比进程更小的能独立运行的基本单位。
    C.一个进程可以创建和撤销多个线程;同一个进程中的多个线程之间可以并发执行。
```

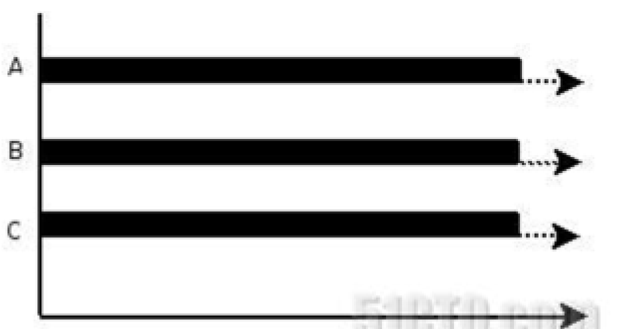
并发和并行

```
    A. 多线程程序在一个核的cpu上运行，就是并发。
    B. 多线程程序在多个核的cpu上运行，就是并行。
```

并发



并行



协程和线程

```
协程：独立的栈空间，共享堆空间，调度由用户自己控制，本质上有点类似于用户级线程，这些用户级线程的调度也是自己实现的。
线程：一个线程上可以跑多个协程，协程是轻量级的线程。
```

goroutine 只是由官方实现的超级"线程池"。

每个实力 `4~5KB` 的栈内存占用和由于实现机制而大幅减少的创建和销毁开销是go高并发的根本原因。

并发不是并行：

并发主要由切换时间片来实现"同时"运行，并行则是直接利用多核实现多线程的运行，go可以设置使用核数，以发挥多核计算机的能力。

goroutine 奉行通过通信来共享内存，而不是共享内存来通信。

Goroutine

在java/c++中我们要实现并发编程的时候，我们通常需要自己维护一个线程池，并且需要自己去包装一个又一个的任务，同时需要自己去调度线程执行任务并维护上下文切换，这一切通常会耗费程序员大量的心智。那么能不能有一种机制，程序员只需要定义很多个任务，让系统去帮助我们把这些任务分配到CPU上实现并发执行呢？

Go语言中的goroutine就是这样一种机制，goroutine的概念类似于线程，但 goroutine是由Go的运行时（runtime）调度和管理的。Go程序会智能地将 goroutine 中的任务合理地分配给每个CPU。Go语言之所以被称为现代化的编程语言，就是因为它在语言层面已经内置了调度和上下文切换的机制。

在Go语言编程中你不需要去自己写进程、线程、协程，你的技能包里只有一个技能-goroutine，当你需要让某个任务并发执行的时候，你只需要把这个任务包装成一个函数，开启一个goroutine去执行这个函数就可以了，就是这么简单粗暴。

使用goroutine

Go语言中使用goroutine非常简单，只需要在调用函数的时候在前面加上go关键字，就可以为一个函数创建一个goroutine。

一个goroutine必定对应一个函数，可以创建多个goroutine去执行相同的函数。

启动单个goroutine

启动goroutine的方式非常简单，只需要在调用的函数（普通函数和匿名函数）前面加上一个go关键字。

举个例子如下：

```
func hello() {
    fmt.Println("Hello Goroutine!")
}
func main() {
    hello()
    fmt.Println("main goroutine done!")
}
```

这个示例中hello函数和下面的语句是串行的，执行的结果是打印完Hello Goroutine!后打印main goroutine done!。

接下来我们在调用hello函数前面加上关键字go，也就是启动一个goroutine去执行hello这个函数。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
}
```

这一次的执行结果只打印了main goroutine done!，并没有打印Hello Goroutine!。为什么呢？

在程序启动时，Go程序就会为main()函数创建一个默认的goroutine。

当main()函数返回的时候该goroutine就结束了，所有在main()函数中启动的goroutine会一同结束，main函数所在的goroutine就像是权利的游戏中的夜王，其他的goroutine都是异鬼，夜王一死它转化的那些异鬼也就全部GG了。

所以我们要想办法让main函数等一等hello函数，最简单粗暴的方式就是time.Sleep了。

```
func main() {
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    time.Sleep(time.Second)
}
```

执行上面的代码你会发现，这一次先打印main goroutine done!，然后紧接着打印Hello Goroutine!。

首先为什么会先打印main goroutine done!是因为我们在创建新的goroutine的时候需要花费一些时间，而此时main函数所在的goroutine是继续执行的。

启动多个goroutine

在Go语言中实现并发就是这样简单，我们还可以启动多个goroutine。让我们再来一个例子：（这里使用了sync.WaitGroup来实现goroutine的同步）

```
var wg sync.WaitGroup

func hello(i int) {
    defer wg.Done() // goroutine结束就登记-1
    fmt.Println("Hello Goroutine!", i)
}
func main() {

    for i := 0; i < 10; i++ {
        wg.Add(1) // 启动一个goroutine就登记+1
        go hello(i)
```

```
    }
    wg.Wait() // 等待所有登记的goroutine都结束
}
```

多次执行上面的代码，会发现每次打印的数字的顺序都不一致。这是因为10个goroutine是并发执行的，而goroutine的调度是随机的。

注意

- 如果主协程退出了，其他任务还执行吗（运行下面的代码测试一下吧）

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 合起来写
    go func() {
        i := 0
        for {
            i++
            fmt.Printf("new goroutine: i = %d\n", i)
            time.Sleep(time.Second)
        }
    }()
    i := 0
    for {
        i++
        fmt.Printf("main goroutine: i = %d\n", i)
        time.Sleep(time.Second)
        if i == 2 {
            break
        }
    }
}
```

goroutine与线程

可增长的栈

OS线程（操作系统线程）一般都有固定的栈内存（通常为2MB）,一个goroutine的栈在其生命周期开始时只有很小的栈（典型情况下2KB），goroutine的栈不是固定的，他可以按需增大和缩小，goroutine的栈大小限制可以达到1GB，虽然极少会用到这个大。所以在Go语言中一次创建十万左右的goroutine也是可以的。

goroutine调度

GPM是Go语言运行时（runtime）层面的实现，是go语言自己实现的一套调度系统。区别于操作系统调度OS线程。

- 1.G很好理解，就是个goroutine的，里面除了存放本goroutine信息外 还有与所在P的绑定等信息。
- 2.P管理着一组goroutine队列，P里面会存储当前goroutine运行的上下文环境（函数指针，堆栈地址及地址边界），P会对自己管理的goroutine队列做一些调度（比如把占用CPU时间较长的goroutine暂停、运行后续的goroutine等等）当自己的队列消费完了就去全局队列里取，如果全局队列里也消费完了会去其他P的队列里抢任务。
- 3.M（machine）是Go运行时（runtime）对操作系统内核线程的虚拟， M与内核线程一般是一一映射的关系， 一个groutine最终是要放到M上执行的；

P与M一般也是一一对应的。他们关系是： P管理着一组G挂载在M上运行。当一个G长久阻塞在一个M上时，runtime会新建一个M，阻塞G所在的P会把其他的G 挂载在新建的M上。当旧的G阻塞完成或者认为其已经死掉时 回收旧的M。

P的个数是通过runtime.GOMAXPROCS设定（最大256），Go1.5版本之后默认为物理线程数。 在并发量大的时候会增加一些P和M，但不会太多，切换太频繁的话得不偿失。

单从线程调度讲，Go语言相比起其他语言的优势在于OS线程是由OS内核来调度的，goroutine则是由Go运行时（runtime）自己的调度器调度的，这个调度器使用一个称为m:n调度的技术（复用/调度m个goroutine到n个OS线程）。 其一大特点是goroutine的调度是在用户态下完成的， 不涉及内核态与用户态之间的频繁切换，包括内存的分配与释放，都是在用户态维护着一块大的内存池， 不直接调用系统的malloc函数（除非内存池需要改变），成本比调度OS线程低很多。 另一方面充分利用了多核的硬件资源，近似的把若干goroutine均分在物理线程上， 再加上本身goroutine的超轻量，以上种种保证了go调度方面的性能。

runtime包

runtime.Gosched()

让出CPU时间片，重新等待安排任务(大概意思就是本来计划的好好的周末出去烧烤，但是你妈让你去相亲,两种情况第一就是你相亲速度非常快，见面就黄不耽误你继续烧烤，第二种情况就是你相亲速度特别慢，见面就是你侬我侬的，耽误了烧烤，但是还馋就是耽误了烧烤你还得去烧烤)

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    go func(s string) {
        for i := 0; i < 2; i++ {
            fmt.Println(s)
        }
    }("world")
    // 主协程
    for i := 0; i < 2; i++ {
        // 切一下，再次分配任务
        runtime.Gosched()
        fmt.Println("hello")
    }
}
```

runtime.Goexit()

退出当前协程(一边烧烤一边相亲，突然发现相亲对象太丑影响烧烤，果断让她滚蛋，然后也就没有然后了)

```
package main

import (
    "fmt"
    "runtime"
)
```

```go
func main() {
	go func() {
		defer fmt.Println("A.defer")
		func() {
			defer fmt.Println("B.defer")
			// 结束协程
			runtime.Goexit()
			defer fmt.Println("C.defer")
			fmt.Println("B")
		}()
		fmt.Println("A")
	}()
	for {
	}
}
```

runtime.GOMAXPROCS

Go运行时的调度器使用GOMAXPROCS参数来确定需要使用多少个OS线程来同时执行Go代码。默认值是机器上的CPU核心数。例如在一个8核心的机器上，调度器会把Go代码同时调度到8个OS线程上（GOMAXPROCS是m:n调度中的n）。

Go语言中可以通过runtime.GOMAXPROCS()函数设置当前程序并发时占用的CPU逻辑核心数。

Go1.5版本之前，默认使用的是单核心执行。Go1.5版本之后，默认使用全部的CPU逻辑核心数。

我们可以通过将任务分配到不同的CPU逻辑核心上实现并行的效果，这里举个例子：

```go
func a() {
	for i := 1; i < 10; i++ {
		fmt.Println("A:", i)
	}
}

func b() {
	for i := 1; i < 10; i++ {
		fmt.Println("B:", i)
	}
}

func main() {
	runtime.GOMAXPROCS(1)
	go a()
```

```
    go b()
    time.Sleep(time.Second)
}
```

两个任务只有一个逻辑核心，此时是做完一个任务再做另一个任务。 将逻辑核心数设为2，此时两个任务并行执行，代码如下。

```
func a() {
    for i := 1; i < 10; i++ {
        fmt.Println("A:", i)
    }
}

func b() {
    for i := 1; i < 10; i++ {
        fmt.Println("B:", i)
    }
}

func main() {
    runtime.GOMAXPROCS(2)
    go a()
    go b()
    time.Sleep(time.Second)
}
```

Go语言中的操作系统线程和goroutine的关系：

- 1.一个操作系统线程对应用户态多个goroutine。
- 2.go程序可以同时使用多个操作系统线程。
- 3.goroutine和OS线程是多对多的关系，即m:n。

Channel

channel

单纯地将函数并发执行是没有意义的。函数与函数间需要交换数据才能体现并发执行函数的意义。

虽然可以使用共享内存进行数据交换，但是共享内存在不同的goroutine中容易发生竞态问题。为了保证数据交换的正确性，必须使用互斥量对内存进行加锁，这种做法势必造成性能问题。

Go语言的并发模型是CSP（Communicating Sequential Processes），提倡通过通信共享内存而不是通过共享内存而实现通信。

如果说goroutine是Go程序并发的执行体，channel就是它们之间的连接。channel是可以让一个goroutine发送特定值到另一个goroutine的通信机制。

Go 语言中的通道（channel）是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出（First In First Out）的规则，保证收发数据的顺序。每一个通道都是一个具体类型的导管，也就是声明channel的时候需要为其指定元素类型。

channel类型

channel是一种类型，一种引用类型。声明通道类型的格式如下：

```
    var 变量 chan 元素类型
```

举几个例子：

```
    var ch1 chan int   // 声明一个传递整型的通道
    var ch2 chan bool  // 声明一个传递布尔型的通道
    var ch3 chan []int // 声明一个传递int切片的通道
```

创建channel

通道是引用类型，通道类型的空值是nil。

```
var ch chan int
fmt.Println(ch) // <nil>
```

声明的通道后需要使用make函数初始化之后才能使用。

创建channel的格式如下：

```
    make(chan 元素类型, [缓冲大小])
```

channel的缓冲大小是可选的。

举几个例子：

```
ch4 := make(chan int)
ch5 := make(chan bool)
ch6 := make(chan []int)
```

channel操作

通道有发送（send）、接收(receive）和关闭（close）三种操作。

发送和接收都使用<-符号。

现在我们先使用以下语句定义一个通道：

```
ch := make(chan int)
```

发送

将一个值发送到通道中。

```
ch <- 10 // 把10发送到ch中
```

接收

从一个通道中接收值。

```
x := <- ch // 从ch中接收值并赋值给变量x
<-ch       // 从ch中接收值，忽略结果
```

关闭

我们通过调用内置的close函数来关闭通道。

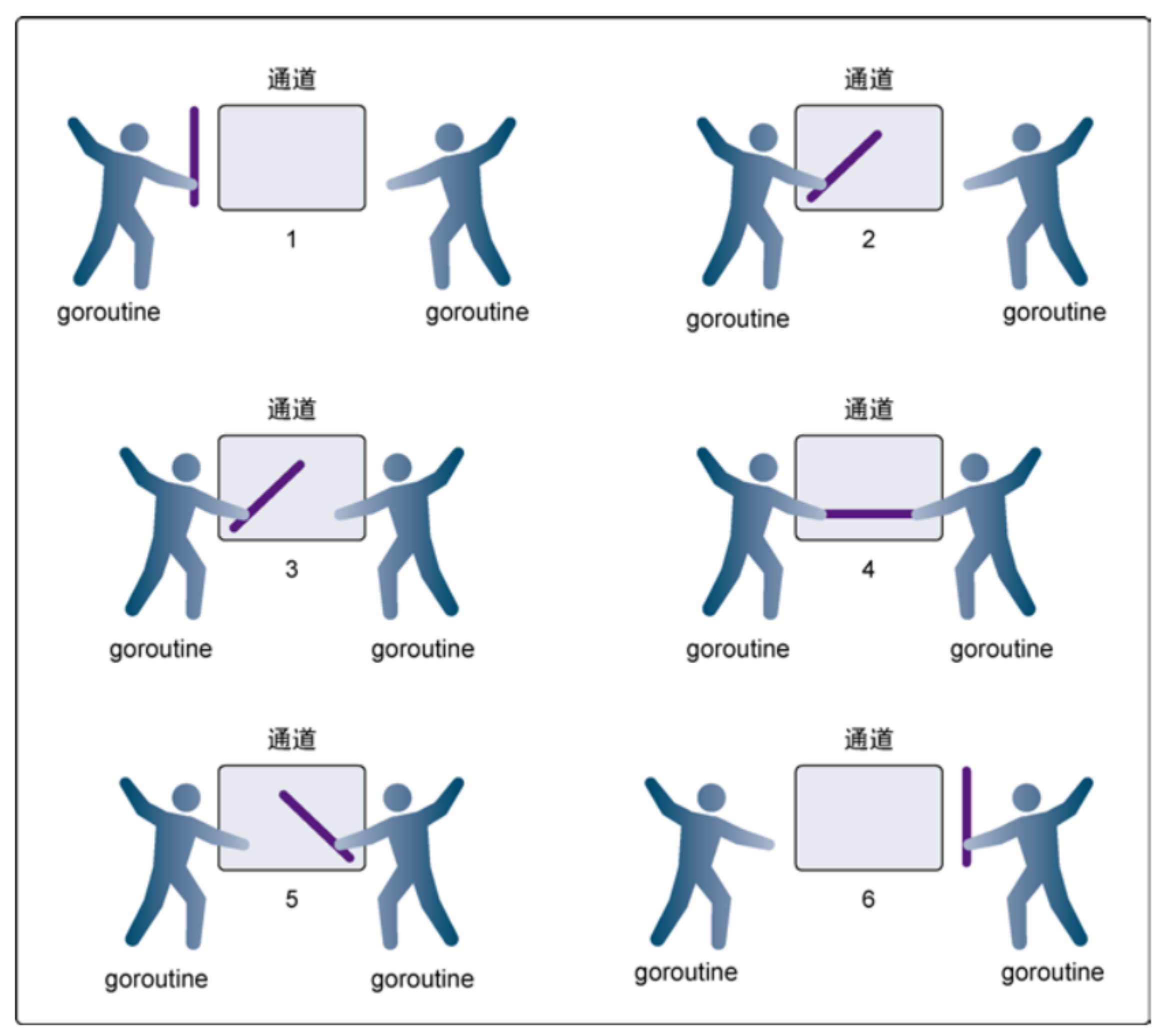
```
    close(ch)
```

关于关闭通道需要注意的事情是，只有在通知接收方goroutine所有的数据都发送完毕的时候才需要关闭通道。通道是可以被垃圾回收机制回收的，它和关闭文件是不一样的，在结束操作之后关闭文件是必须要做的，但关闭通道不是必须的。

关闭后的通道有以下特点：

```
1.对一个关闭的通道再发送值就会导致panic。
2.对一个关闭的通道进行接收会一直获取值直到通道为空。
3.对一个关闭的并且没有值的通道执行接收操作会得到对应类型的零值。
4.关闭一个已经关闭的通道会导致panic。
```

无缓冲的通道



使用无缓冲的通道在 goroutine 之间同步

无缓冲的通道又称为阻塞的通道。我们来看一下下面的代码：

```
func main() {
    ch := make(chan int)
```

```
    ch <- 10
    fmt.Println("发送成功")
}
```

上面这段代码能够通过编译，但是执行的时候会出现以下错误：

```
    fatal error: all goroutines are asleep - deadlock!

    goroutine 1 [chan send]:
    main.main()
            .../src/github.com/pprof/studygo/day06/channel02/main.go:8 +0x54
```

为什么会出现deadlock错误呢？

因为我们使用ch := make(chan int)创建的是无缓冲的通道，无缓冲的通道只有在有人接收值的时候才能发送值。就像你住的小区没有快递柜和代收点，快递员给你打电话必须要把这个物品送到你的手中，简单来说就是无缓冲的通道必须有接收才能发送。

上面的代码会阻塞在ch <- 10这一行代码形成死锁，那如何解决这个问题呢？

一种方法是启用一个goroutine去接收值，例如：

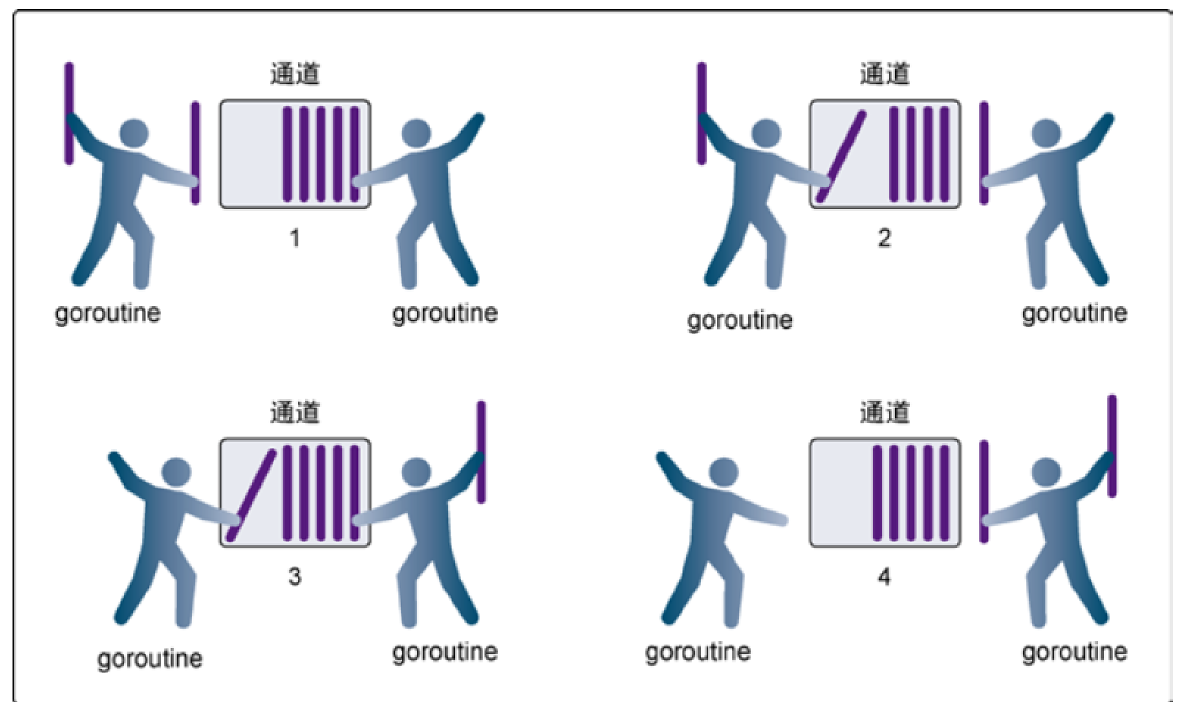
```
func recv(c chan int) {
    ret := <-c
    fmt.Println("接收成功", ret)
}
func main() {
    ch := make(chan int)
    go recv(ch) // 启用goroutine从通道接收值
    ch <- 10
    fmt.Println("发送成功")
}
```

无缓冲通道上的发送操作会阻塞，直到另一个goroutine在该通道上执行接收操作，这时值才能发送成功，两个goroutine将继续执行。相反，如果接收操作先执行，接收方的goroutine将阻塞，直到另一个goroutine在该通道上发送一个值。

使用无缓冲通道进行通信将导致发送和接收的goroutine同步化。因此，无缓冲通道也被称为同步通道。

有缓冲的通道

解决上面问题的方法还有一种就是使用有缓冲区的通道。



使用有缓冲的通道在 goroutine 之间同步数据

我们可以在使用make函数初始化通道的时候为其指定通道的容量，例如：

```
func main() {
    ch := make(chan int, 1) // 创建一个容量为1的有缓冲区通道
    ch <- 10
    fmt.Println("发送成功")
}
```

只要通道的容量大于零，那么该通道就是有缓冲的通道，通道的容量表示通道中能存放元素的数量。就像你小区的快递柜只有那么个多格子，格子满了就装不下了，就阻塞了，等到别人取走一个快递员就能往里面放一个。

我们可以使用内置的len函数获取通道内元素的数量，使用cap函数获取通道的容量，虽然我们很少会这么做。

close()

可以通过内置的close()函数关闭channel（如果你的管道不往里存值或者取值的时候一定记得关闭管道）

```
package main

import "fmt"

```

```
func main() {
    c := make(chan int)
    go func() {
        for i := 0; i < 5; i++ {
            c <- i
        }
        close(c)
    }()
    for {
        if data, ok := <-c; ok {
            fmt.Println(data)
        } else {
            break
        }
    }
    fmt.Println("main结束")
}
```

如何优雅的从通道循环取值

当通过通道发送有限的数据时，我们可以通过close函数关闭通道来告知从该通道接收值的goroutine停止等待。当通道被关闭时，往该通道发送值会引发panic，从该通道里接收的值一直都是类型零值。那如何判断一个通道是否被关闭了呢？

我们来看下面这个例子：

```
// channel 练习
func main() {
    ch1 := make(chan int)
    ch2 := make(chan int)
    // 开启goroutine将0~100的数发送到ch1中
    go func() {
        for i := 0; i < 100; i++ {
            ch1 <- i
        }
        close(ch1)
    }()
    // 开启goroutine从ch1中接收值，并将该值的平方发送到ch2中
    go func() {
        for {
            i, ok := <-ch1 // 通道关闭后再取值ok=false
            if !ok {
                break
            }
            ch2 <- i * i
```

```
		}
		close(ch2)
	}()
	// 在主goroutine中从ch2中接收值打印
	for i := range ch2 { // 通道关闭后会退出for range循环
		fmt.Println(i)
	}
}
```

从上面的例子中我们看到有两种方式在接收值的时候判断通道是否被关闭，我们通常使用的是for range的方式。

单向通道

有的时候我们会将通道作为参数在多个任务函数间传递，很多时候我们在不同的任务函数中使用通道都会对其进行限制，比如限制通道在函数中只能发送或只能接收。

Go语言中提供了单向通道来处理这种情况。例如，我们把上面的例子改造如下：

```
func counter(out chan<- int) {
	for i := 0; i < 100; i++ {
		out <- i
	}
	close(out)
}

func squarer(out chan<- int, in <-chan int) {
	for i := range in {
		out <- i * i
	}
	close(out)
}
func printer(in <-chan int) {
	for i := range in {
		fmt.Println(i)
	}
}

func main() {
	ch1 := make(chan int)
	ch2 := make(chan int)
	go counter(ch1)
	go squarer(ch2, ch1)
	printer(ch2)
}
```

其中,

```
1. chan<- int是一个只能发送的通道,可以发送但是不能接收;
2. <-chan int是一个只能接收的通道,可以接收但是不能发送。
```

在函数传参及任何赋值操作中将双向通道转换为单向通道是可以的,但反过来是不可以的。

通道总结

channel常见的异常总结,如下图:

channel异常情况总结					
channel	nil	非空	空的	满了	没满
接收	阻塞	接收值	阻塞	接收值	接收值
发送	阻塞	发送值	发送值	阻塞	发送值
关闭	panic	关闭成功,读完数据后返回零值	关闭成功,返回零值	关闭成功,读完数据后返回零值	关闭成功,读完数据后返回零值

www.topgoer.com

注意:关闭已经关闭的channel也会引发panic。

Goroutine池

worker pool（goroutine池）

- 本质上是生产者消费者模型
- 可以有效控制goroutine数量，防止暴涨
- 需求：
    - 计算一个数字的各个位数之和，例如数字123，结果为1+2+3=6
    - 随机生成数字进行计算
- 控制台输出结果如下：

```
job id:483015 randnum:7441912810059464404 result:73
job id:483016 randnum:5802028204449355485 result:78
job id:483017 randnum:2108302484068381386 result:75
job id:483018 randnum:1520684906012643161 result:65
job id:483019 randnum:4362171100320566475 result:63
job id:483020 randnum:7856703083866169273 result:95
job id:483021 randnum:7944042142785246759 result:90
job id:483022 randnum:1124753523380975949 result:87
job id:483023 randnum:6514065607240209399 result:78
job id:483024 randnum:6971005295395104852 result:81
job id:483025 randnum:6734009815004836451 result:74
job id:483026 randnum:5773076917115205882 result:84
```

```
package main

import (
    "fmt"
    "math/rand"
)

type Job struct {
    // id
```

```
    Id int
    // 需要计算的随机数
    RandNum int
}

type Result struct {
    // 这里必须传对象实例
    job *Job
    // 求和
    sum int
}

func main() {
    // 需要2个管道
    // 1.job管道
    jobChan := make(chan *Job, 128)
    // 2.结果管道
    resultChan := make(chan *Result, 128)
    // 3.创建工作池
    createPool(64, jobChan, resultChan)
    // 4.开个打印的协程
    go func(resultChan chan *Result) {
        // 遍历结果管道打印
        for result := range resultChan {
            fmt.Printf("job id:%v randnum:%v result:%d\n", result.job.Id,
                result.job.RandNum, result.sum)
        }
    }(resultChan)
    var id int
    // 循环创建job，输入到管道
    for {
        id++
        // 生成随机数
        r_num := rand.Int()
        job := &Job{
            Id:      id,
            RandNum: r_num,
        }
        jobChan <- job
    }
}

// 创建工作池
// 参数1：开几个协程
func createPool(num int, jobChan chan *Job, resultChan chan *Result) {
    // 根据开协程个数，去跑运行
```

```
    for i := 0; i < num; i++ {
        go func(jobChan chan *Job, resultChan chan *Result) {
            // 执行运算
            // 遍历job管道所有数据，进行相加
            for job := range jobChan {
                // 随机数接过来
                r_num := job.RandNum
                // 随机数每一位相加
                // 定义返回值
                var sum int
                for r_num != 0 {
                    tmp := r_num % 10
                    sum += tmp
                    r_num /= 10
                }
                // 想要的结果是Result
                r := &Result{
                    job: job,
                    sum: sum,
                }
                //运算结果扔到管道
                resultChan <- r
            }
        }(jobChan, resultChan)
    }
}
```

定时器

定时器

- Timer：时间到了，执行只执行1次

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 1.timer基本使用
    //timer1 := time.NewTimer(2 * time.Second)
    //t1 := time.Now()
    //fmt.Printf("t1:%v\n", t1)
    //t2 := <-timer1.C
    //fmt.Printf("t2:%v\n", t2)

    // 2.验证timer只能响应1次
    //timer2 := time.NewTimer(time.Second)
    //for {
    // <-timer2.C
    // fmt.Println("时间到")
    //}

    // 3.timer实现延时的功能
    //(1)
    //time.Sleep(time.Second)
    //(2)
    //timer3 := time.NewTimer(2 * time.Second)
    //<-timer3.C
    //fmt.Println("2秒到")
    //(3)
    //<-time.After(2*time.Second)
    //fmt.Println("2秒到")

    // 4.停止定时器
    //timer4 := time.NewTimer(2 * time.Second)
    //go func() {
    // <-timer4.C
```

```
    // fmt.Println("定时器执行了")
    //}()
    //b := timer4.Stop()
    //if b {
    // fmt.Println("timer4已经关闭")
    //}

    // 5.重置定时器
    timer5 := time.NewTimer(3 * time.Second)
    timer5.Reset(1 * time.Second)
    fmt.Println(time.Now())
    fmt.Println(<-timer5.C)

    for {
    }
}
```

- Ticker：时间到了，多次执行

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // 1.获取ticker对象
    ticker := time.NewTicker(1 * time.Second)
    i := 0
    // 子协程
    go func() {
        for {
            //<-ticker.C
            i++
            fmt.Println(<-ticker.C)
            if i == 5 {
                //停止
                ticker.Stop()
            }
        }
    }()
    for {
    }
}
```

select

select多路复用

在某些场景下我们需要同时从多个通道接收数据。通道在接收数据时，如果没有数据可以接收将会发生阻塞。你也许会写出如下代码使用遍历的方式来实现：

```
for{
    // 尝试从ch1接收值
    data, ok := <-ch1
    // 尝试从ch2接收值
    data, ok := <-ch2
    …
}
```

这种方式虽然可以实现从多个通道接收值的需求，但是运行性能会差很多。为了应对这种场景，Go内置了select关键字，可以同时响应多个通道的操作。

select的使用类似于switch语句，它有一系列case分支和一个默认的分支。每个case会对应一个通道的通信（接收或发送）过程。select会一直等待，直到某个case的通信操作完成时，就会执行case分支对应的语句。具体格式如下：

```
    select {
    case <-chan1:
        // 如果chan1成功读到数据，则进行该case处理语句
    case chan2 <- 1:
        // 如果成功向chan2写入数据，则进行该case处理语句
    default:
        // 如果上面都没有成功，则进入default处理流程
    }
```

- select可以同时监听一个或多个channel，直到其中一个channel ready

```
package main

import (
   "fmt"
   "time"
)

func test1(ch chan string) {
```

```
    time.Sleep(time.Second * 5)
    ch <- "test1"
}
func test2(ch chan string) {
    time.Sleep(time.Second * 2)
    ch <- "test2"
}

func main() {
    // 2个管道
    output1 := make(chan string)
    output2 := make(chan string)
    // 跑2个子协程，写数据
    go test1(output1)
    go test2(output2)
    // 用select监控
    select {
    case s1 := <-output1:
        fmt.Println("s1=", s1)
    case s2 := <-output2:
        fmt.Println("s2=", s2)
    }
}
```

- 如果多个channel同时ready，则随机选择一个执行

```
package main

import (
    "fmt"
)

func main() {
    // 创建2个管道
    int_chan := make(chan int, 1)
    string_chan := make(chan string, 1)
    go func() {
        //time.Sleep(2 * time.Second)
        int_chan <- 1
    }()
    go func() {
        string_chan <- "hello"
    }()
    select {
    case value := <-int_chan:
```

```
        fmt.Println("int:", value)
    case value := <-string_chan:
        fmt.Println("string:", value)
    }
    fmt.Println("main结束")
}
```

- 可以用于判断管道是否存满

```
package main

import (
    "fmt"
    "time"
)

// 判断管道有没有存满
func main() {
    // 创建管道
    output1 := make(chan string, 10)
    // 子协程写数据
    go write(output1)
    // 取数据
    for s := range output1 {
        fmt.Println("res:", s)
        time.Sleep(time.Second)
    }
}

func write(ch chan string) {
    for {
        select {
        // 写数据
        case ch <- "hello":
            fmt.Println("write hello")
        default:
            fmt.Println("channel full")
        }
        time.Sleep(time.Millisecond * 500)
    }
}
```

并发安全和锁

有时候在Go代码中可能会存在多个goroutine同时操作一个资源（临界区），这种情况会发生竞态问题（数据竞态）。类比现实生活中的例子有十字路口被各个方向的的汽车竞争；还有火车上的卫生间被车厢里的人竞争。

举个例子：

```
var x int64
var wg sync.WaitGroup

func add() {
    for i := 0; i < 5000; i++ {
        x = x + 1
    }
    wg.Done()
}
func main() {
    wg.Add(2)
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}
```

上面的代码中我们开启了两个goroutine去累加变量x的值，这两个goroutine在访问和修改x变量的时候就会存在数据竞争，导致最后的结果与期待的不符。

互斥锁

互斥锁是一种常用的控制共享资源访问的方法，它能够保证同时只有一个goroutine可以访问共享资源。Go语言中使用sync包的Mutex类型来实现互斥锁。 使用互斥锁来修复上面代码的问题：

```
var x int64
var wg sync.WaitGroup
var lock sync.Mutex

func add() {
    for i := 0; i < 5000; i++ {
        lock.Lock() // 加锁
        x = x + 1
        lock.Unlock() // 解锁
```

```
    }
    wg.Done()
}
func main() {
    wg.Add(2)
    go add()
    go add()
    wg.Wait()
    fmt.Println(x)
}
```

使用互斥锁能够保证同一时间有且只有一个goroutine进入临界区，其他的goroutine则在等待锁；当互斥锁释放后，等待的goroutine才可以获取锁进入临界区，多个goroutine同时等待一个锁时，唤醒的策略是随机的。

读写互斥锁

互斥锁是完全互斥的，但是有很多实际的场景下是读多写少的，当我们并发的去读取一个资源不涉及资源修改的时候是没有必要加锁的，这种场景下使用读写锁是更好的一种选择。读写锁在Go语言中使用sync包中的RWMutex类型。

读写锁分为两种：读锁和写锁。当一个goroutine获取读锁之后，其他的goroutine如果是获取读锁会继续获得锁，如果是获取写锁就会等待；当一个goroutine获取写锁之后，其他的goroutine无论是获取读锁还是写锁都会等待。

读写锁示例：

```
var (
    x      int64
    wg     sync.WaitGroup
    lock   sync.Mutex
    rwlock sync.RWMutex
)

func write() {
    // lock.Lock()   // 加互斥锁
    rwlock.Lock() // 加写锁
    x = x + 1
    time.Sleep(10 * time.Millisecond) // 假设读操作耗时10毫秒
    rwlock.Unlock()                   // 解写锁
    // lock.Unlock()                     // 解互斥锁
    wg.Done()
}

func read() {
```

```go
    // lock.Lock()                      // 加互斥锁
    rwlock.RLock()                    // 加读锁
    time.Sleep(time.Millisecond) // 假设读操作耗时1毫秒
    rwlock.RUnlock()                  // 解读锁
    // lock.Unlock()                    // 解互斥锁
    wg.Done()
}

func main() {
    start := time.Now()
    for i := 0; i < 10; i++ {
        wg.Add(1)
        go write()
    }

    for i := 0; i < 1000; i++ {
        wg.Add(1)
        go read()
    }

    wg.Wait()
    end := time.Now()
    fmt.Println(end.Sub(start))
}
```

需要注意的是读写锁非常适合读多写少的场景，如果读和写的操作差别不大，读写锁的优势就发挥不出来。

Sync

sync.WaitGroup

在代码中生硬的使用time.Sleep肯定是不合适的，Go语言中可以使用sync.WaitGroup来实现并发任务的同步。 sync.WaitGroup有以下几个方法：

方法名	功能
(wg * WaitGroup) Add(delta int)	计数器+delta
(wg *WaitGroup) Done()	计数器-1
(wg *WaitGroup) Wait()	阻塞直到计数器变为0

sync.WaitGroup内部维护着一个计数器，计数器的值可以增加和减少。例如当我们启动了N 个并发任务时，就将计数器值增加N。每个任务完成时通过调用Done()方法将计数器减1。通过调用Wait()来等待并发任务执行完，当计数器值为0时，表示所有并发任务已经完成。

我们利用sync.WaitGroup将上面的代码优化一下：

```
var wg sync.WaitGroup

func hello() {
    defer wg.Done()
    fmt.Println("Hello Goroutine!")
}
func main() {
    wg.Add(1)
    go hello() // 启动另外一个goroutine去执行hello函数
    fmt.Println("main goroutine done!")
    wg.Wait()
}
```

需要注意sync.WaitGroup是一个结构体，传递的时候要传递指针。

sync.Once

说在前面的话：这是一个进阶知识点。

在编程的很多场景下我们需要确保某些操作在高并发的场景下只执行一次，例如只加载一次配置文件、只关闭一次通道等。

Go语言中的sync包中提供了一个针对只执行一次场景的解决方案–sync.Once。

sync.Once只有一个Do方法，其签名如下：

```
func (o *Once) Do(f func()) {}
```

注意：如果要执行的函数f需要传递参数就需要搭配闭包来使用。

加载配置文件示例

延迟一个开销很大的初始化操作到真正用到它的时候再执行是一个很好的实践。因为预先初始化一个变量（比如在init函数中完成初始化）会增加程序的启动耗时，而且有可能实际执行过程中这个变量没有用上，那么这个初始化操作就不是必须要做的。我们来看一个例子：

```
var icons map[string]image.Image

func loadIcons() {
	icons = map[string]image.Image{
		"left":  loadIcon("left.png"),
		"up":    loadIcon("up.png"),
		"right": loadIcon("right.png"),
		"down":  loadIcon("down.png"),
	}
}

// Icon 被多个goroutine调用时不是并发安全的
func Icon(name string) image.Image {
	if icons == nil {
		loadIcons()
	}
	return icons[name]
}
```

多个goroutine并发调用Icon函数时不是并发安全的，现代的编译器和CPU可能会在保证每个goroutine都满足串行一致的基础上自由地重排访问内存的顺序。loadIcons函数可能会被重排为以下结果：

```
func loadIcons() {
	icons = make(map[string]image.Image)
	icons["left"] = loadIcon("left.png")
	icons["up"] = loadIcon("up.png")
	icons["right"] = loadIcon("right.png")
	icons["down"] = loadIcon("down.png")
}
```

在这种情况下就会出现即使判断了icons不是nil也不意味着变量初始化完成了。考虑到这种情况，我们能想到的办法就是添加互斥锁，保证初始化icons的时候不会被其他的goroutine操作，但是这样做又会引发性能问题。

使用sync.Once改造的示例代码如下：

```
var icons map[string]image.Image

var loadIconsOnce sync.Once

func loadIcons() {
	icons = map[string]image.Image{
		"left":  loadIcon("left.png"),
		"up":    loadIcon("up.png"),
		"right": loadIcon("right.png"),
		"down":  loadIcon("down.png"),
	}
}

// Icon 是并发安全的
func Icon(name string) image.Image {
	loadIconsOnce.Do(loadIcons)
	return icons[name]
}
```

sync.Once其实内部包含一个互斥锁和一个布尔值，互斥锁保证布尔值和数据的安全，而布尔值用来记录初始化是否完成。这样设计就能保证初始化操作的时候是并发安全的并且初始化操作也不会被执行多次。

sync.Map

Go语言中内置的map不是并发安全的。请看下面的示例：

```
var m = make(map[string]int)

func get(key string) int {
	return m[key]
}

func set(key string, value int) {
	m[key] = value
}

func main() {
	wg := sync.WaitGroup{}
```

```
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            set(key, n)
            fmt.Printf("k=:%v,v:=%v\n", key, get(key))
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

上面的代码开启少量几个goroutine的时候可能没什么问题，当并发多了之后执行上面的代码就会报fatal error: concurrent map writes错误。

像这种场景下就需要为map加锁来保证并发的安全性了，Go语言的sync包中提供了一个开箱即用的并发安全版map–sync.Map。开箱即用表示不用像内置的map一样使用make函数初始化就能直接使用。同时sync.Map内置了诸如Store、Load、LoadOrStore、Delete、Range等操作方法。

```
var m = sync.Map{}

func main() {
    wg := sync.WaitGroup{}
    for i := 0; i < 20; i++ {
        wg.Add(1)
        go func(n int) {
            key := strconv.Itoa(n)
            m.Store(key, n)
            value, _ := m.Load(key)
            fmt.Printf("k=:%v,v:=%v\n", key, value)
            wg.Done()
        }(i)
    }
    wg.Wait()
}
```

原子操作(atomic包)

原子操作

代码中的加锁操作因为涉及内核态的上下文切换会比较耗时、代价比较高。针对基本数据类型我们还可以使用原子操作来保证并发安全，因为原子操作是Go语言提供的方法它在用户态就可以完成，因此性能比加锁操作更好。Go语言中原子操作由内置的标准库sync/atomic提供。

atomic包

方法	解释
func LoadInt32(addr int32) (val int32) func LoadInt64(addr `int64 ) (val int64) func LoadUint32(addr uint32 ) (val uint32) func LoadUint64(addr uint64 ) (val uint64) func LoadUintptr(addr uintptr ) (val uintptr) func LoadPointer(addr unsafe.Pointer`) (val unsafe.Pointer)	读取操作
func StoreInt32(addr `*int32`, val int32) func StoreInt64(addr `*int64`, val int64) func StoreUint32(addr `*uint32`, val uint32) func StoreUint64(addr `*uint64`, val uint64) func StoreUintptr(addr `*uintptr`, val uintptr) func StorePointer(addr `*unsafe.Pointer`, val unsafe.Pointer)	写入操作
func AddInt32(addr `*int32`, delta int32) (new int32) func AddInt64(addr `*int64`, delta int64) (new int64) func AddUint32(addr `*uint32`, delta uint32) (new uint32) func AddUint64(addr `*uint64`, delta uint64) (new uint64) func AddUintptr(addr `*uintptr`, delta uintptr) (new uintptr)	修改操作
func SwapInt32(addr `*int32`, new int32) (old int32) func SwapInt64(addr `*int64`, new int64) (old int64) func SwapUint32(addr `*uint32`, new uint32) (old uint32) func SwapUint64(addr `*uint64`, new uint64) (old uint64) func SwapUintptr(addr `*uintptr`, new uintptr) (old uintptr) func SwapPointer(addr `*unsafe.Pointer`, new unsafe.Pointer) (old unsafe.Pointer)	交换操作

方法	解释
func CompareAndSwapInt32(addr `*int32`, old, new int32) (swapped bool) func CompareAndSwapInt64(addr `*int64`, old, new int64) (swapped bool) func CompareAndSwapUint32(addr `*uint32`, old, new uint32) (swapped bool) func CompareAndSwapUint64(addr `*uint64`, old, new uint64) (swapped bool) func CompareAndSwapUintptr(addr `*uintptr`, old, new uintptr) (swapped bool) func CompareAndSwapPointer(addr `*unsafe.Pointer`, old, new unsafe.Pointer) (swapped bool)	比较并交换操作

示例

我们填写一个示例来比较下互斥锁和原子操作的性能。

```
var x int64
var l sync.Mutex
var wg sync.WaitGroup

// 普通版加函数
func add() {
	// x = x + 1
	x++ // 等价于上面的操作
	wg.Done()
}

// 互斥锁版加函数
func mutexAdd() {
	l.Lock()
	x++
	l.Unlock()
	wg.Done()
}

// 原子操作版加函数
func atomicAdd() {
	atomic.AddInt64(&x, 1)
	wg.Done()
}
```

```go
func main() {
	start := time.Now()
	for i := 0; i < 10000; i++ {
		wg.Add(1)
		// go add()       // 普通版add函数 不是并发安全的
		// go mutexAdd()  // 加锁版add函数 是并发安全的，但是加锁性能开销大
		go atomicAdd() // 原子操作版add函数 是并发安全，性能优于加锁版
	}
	wg.Wait()
	end := time.Now()
	fmt.Println(x)
	fmt.Println(end.Sub(start))
}
```

atomic包提供了底层的原子级内存操作，对于同步算法的实现很有用。这些函数必须谨慎地保证正确使用。除了某些特殊的底层应用，使用通道或者sync包的函数/类型实现同步更好。

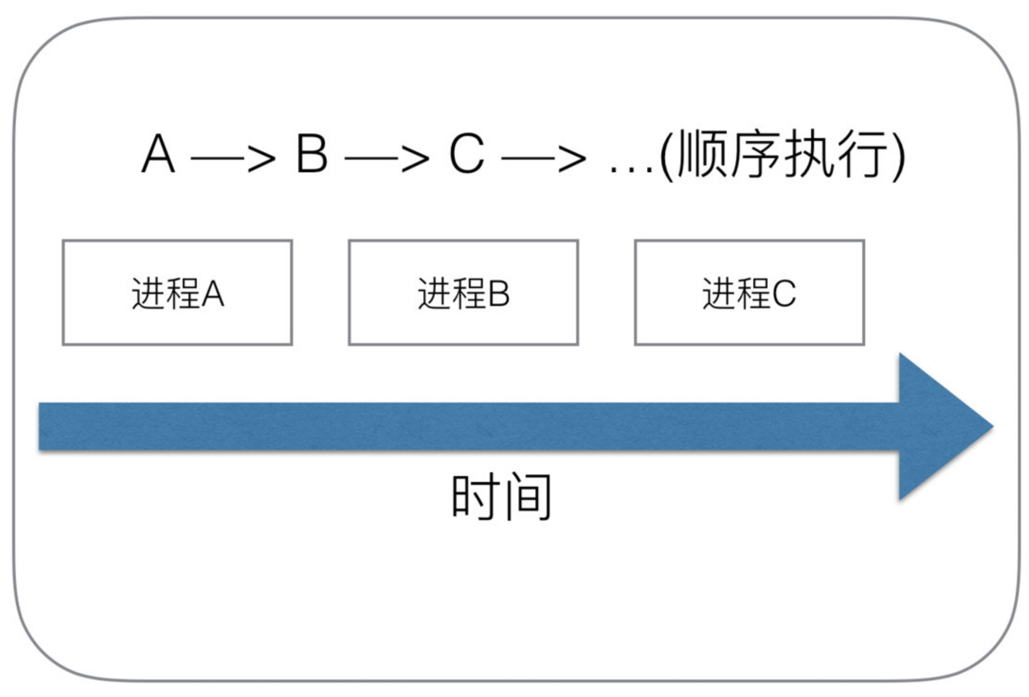
GMP 原理与调度

一、Golang “调度器” 的由来?

(1) 单进程时代不需要调度器

我们知道，一切的软件都是跑在操作系统上，真正用来干活 (计算) 的是 CPU。早期的操作系统每个程序就是一个进程，知道一个程序运行完，才能进行下一个进程，就是 “单进程时代”

一切的程序只能串行发生。



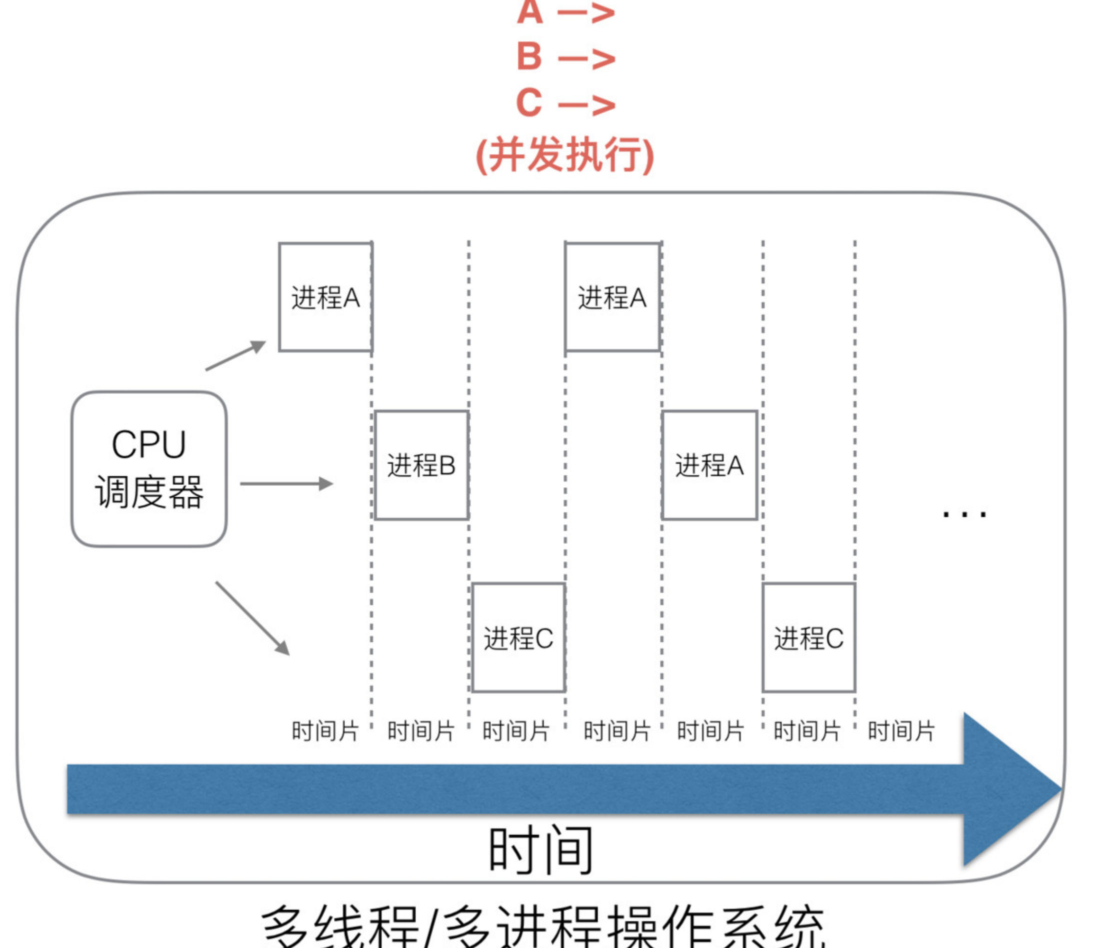
早期的单进程操作系统，面临 2 个问题：

1. 单一的执行流程，计算机只能一个任务一个任务处理。
2. 进程阻塞所带来的 CPU 时间浪费。

那么能不能有多个进程来宏观一起来执行多个任务呢？

后来操作系统就具有了最早的并发能力：多进程并发，当一个进程阻塞的时候，切换到另外等待执行的进程，这样就能尽量把 CPU 利用起来，CPU 就不浪费了。

(2) 多进程 / 线程时代有了调度器需求

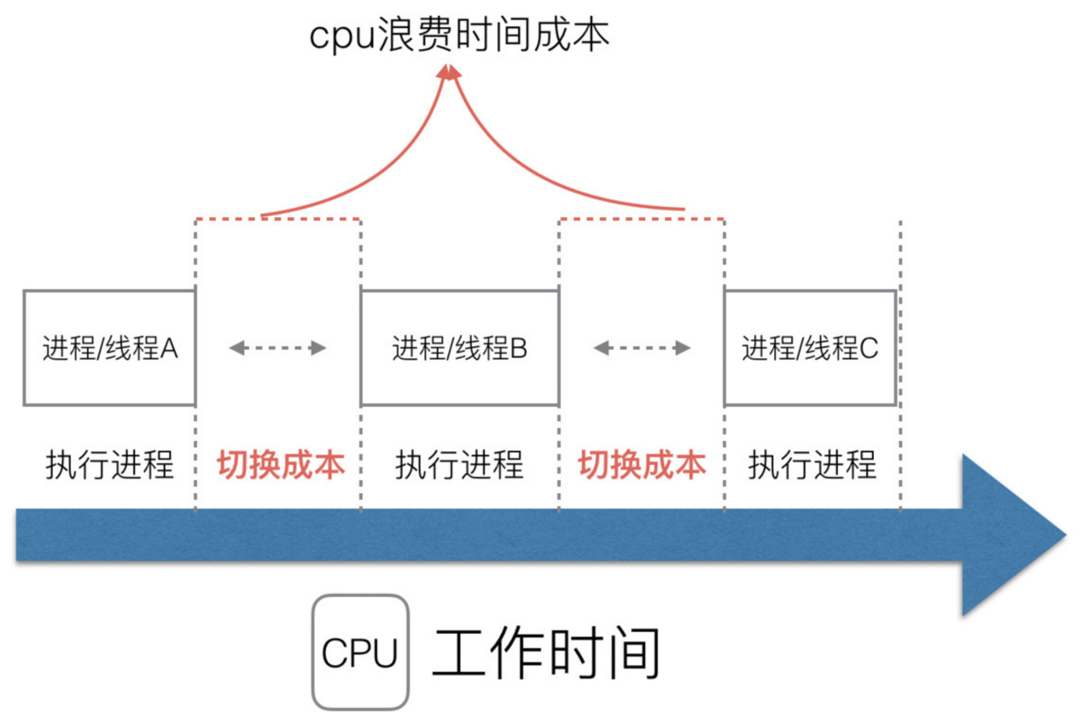


在多进程 / 多线程的操作系统中，就解决了阻塞的问题，因为一个进程阻塞 cpu 可以立刻切换到其他进程中去执行，而且调度 cpu 的算法可以保证在运行的进程都可以被分配到 cpu 的运行时间片。这样从宏观来看，似乎多个进程是在同时被运行。

但新的问题就又出现了，进程拥有太多的资源，进程的创建、切换、销毁，都会占用很长的时间，CPU 虽然利用起来了，但如果进程过多，CPU 有很大的一部分都被用来进行进程调度了。

怎么才能提高 CPU 的利用率呢？

但是对于 Linux 操作系统来讲，cpu 对进程的态度和线程的态度是一样的。



很明显，CPU 调度切换的是进程和线程。尽管线程看起来很美好，但实际上多线程开发设计会变得更加复杂，要考虑很多同步竞争等问题，如锁、竞争冲突等。

(3) 协程来提高 CPU 利用率

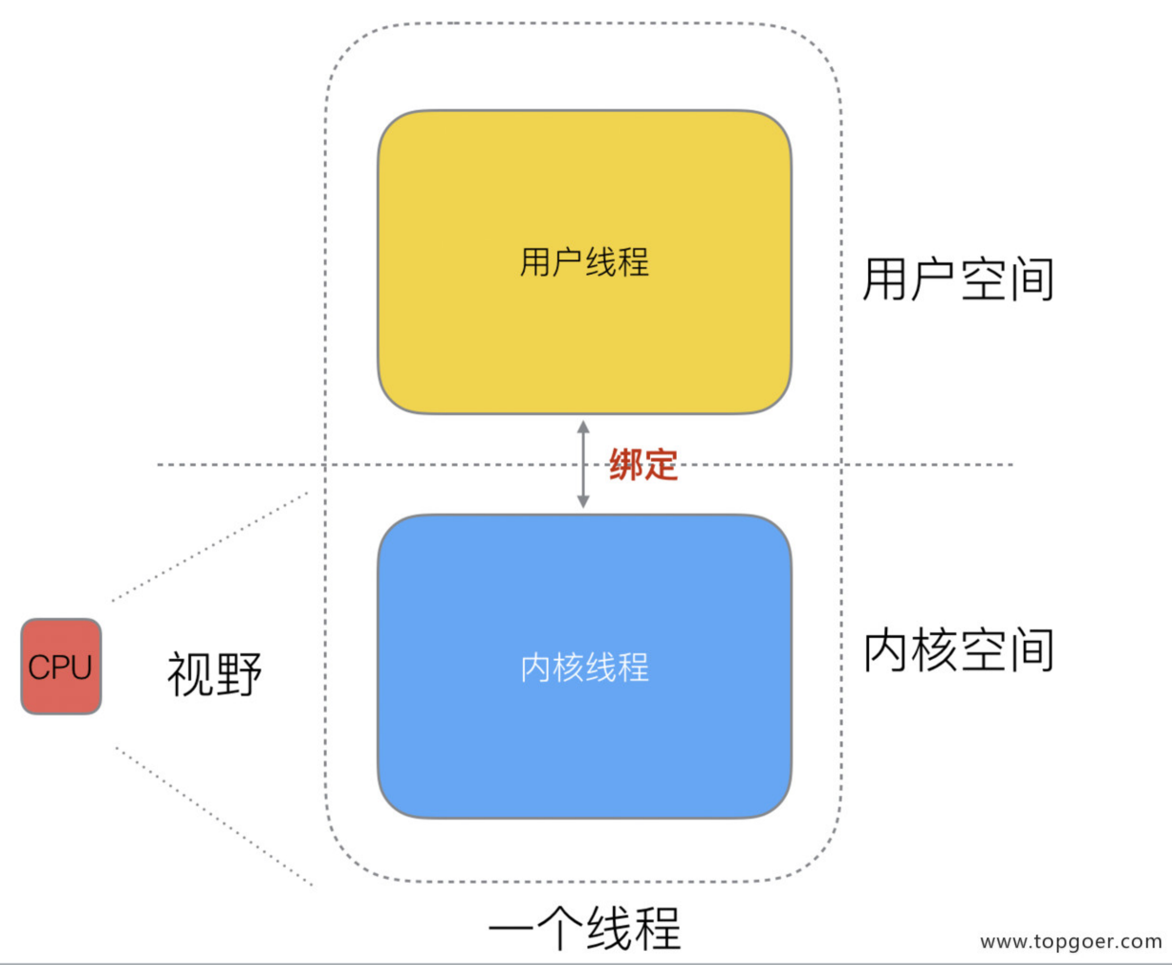
多进程、多线程已经提高了系统的并发能力，但是在当今互联网高并发场景下，为每个任务都创建一个线程是不现实的，因为会消耗大量的内存 (进程虚拟内存会占用 4GB [32 位操作系统], 而线程也要大约 4MB)。

大量的进程 / 线程出现了新的问题

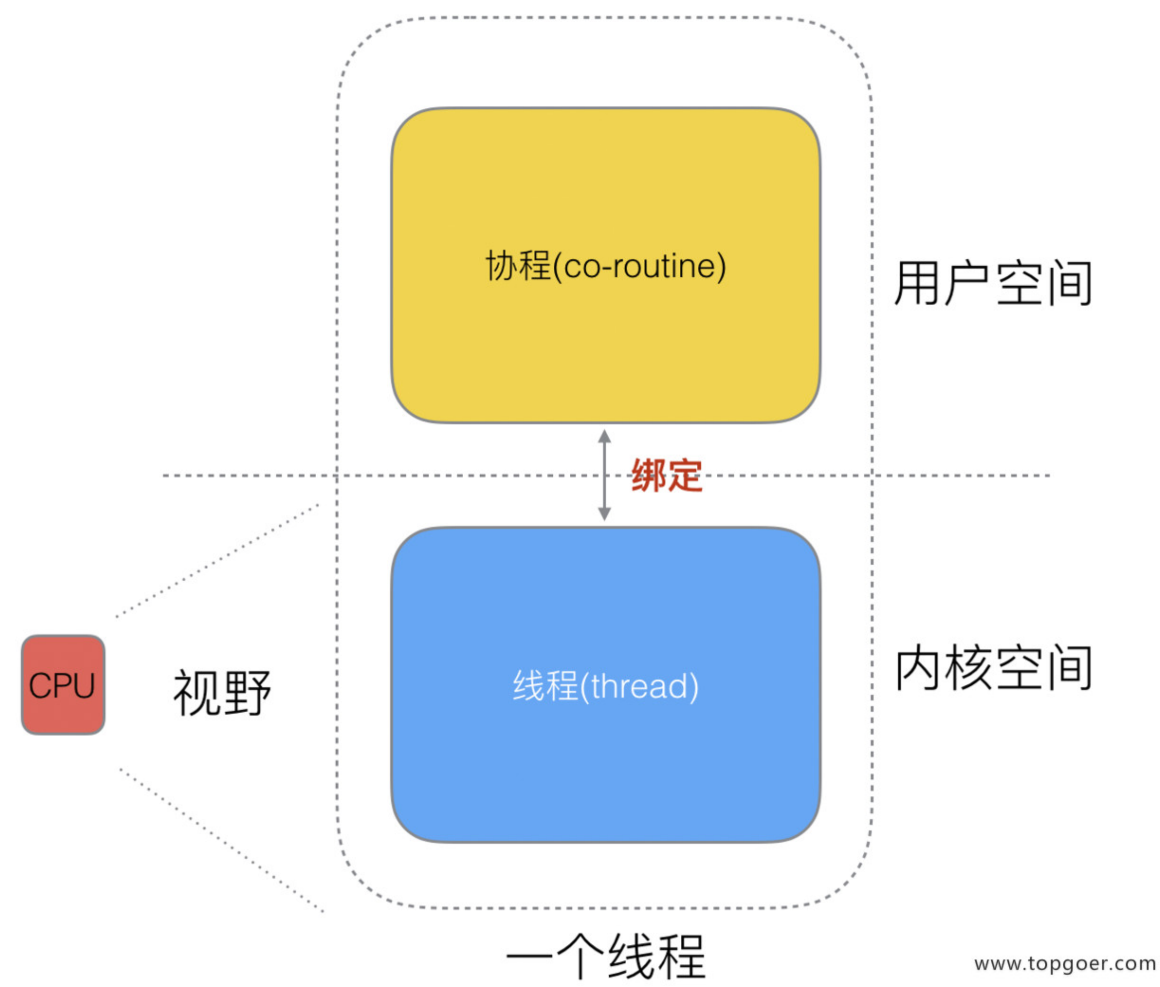
- 高内存占用
- 调度的高消耗 CPU

好了，然后工程师们就发现，其实一个线程分为 “内核态 “线程和” 用户态 “线程。

一个 “用户态线程” 必须要绑定一个 “内核态线程”，但是 CPU 并不知道有 “用户态线程” 的存在，它只知道它运行的是一个 “内核态线程”(Linux 的 PCB 进程控制块)。



这样，我们再去细化去分类一下，内核线程依然叫 “线程 (thread)”，用户线程叫 “协程 (co-routine)”.



看到这里，我们就要开脑洞了，既然一个协程 (co-routine) 可以绑定一个线程 (thread)，那么能不能多个协程 (co-routine) 绑定一个或者多个线程 (thread) 上呢。

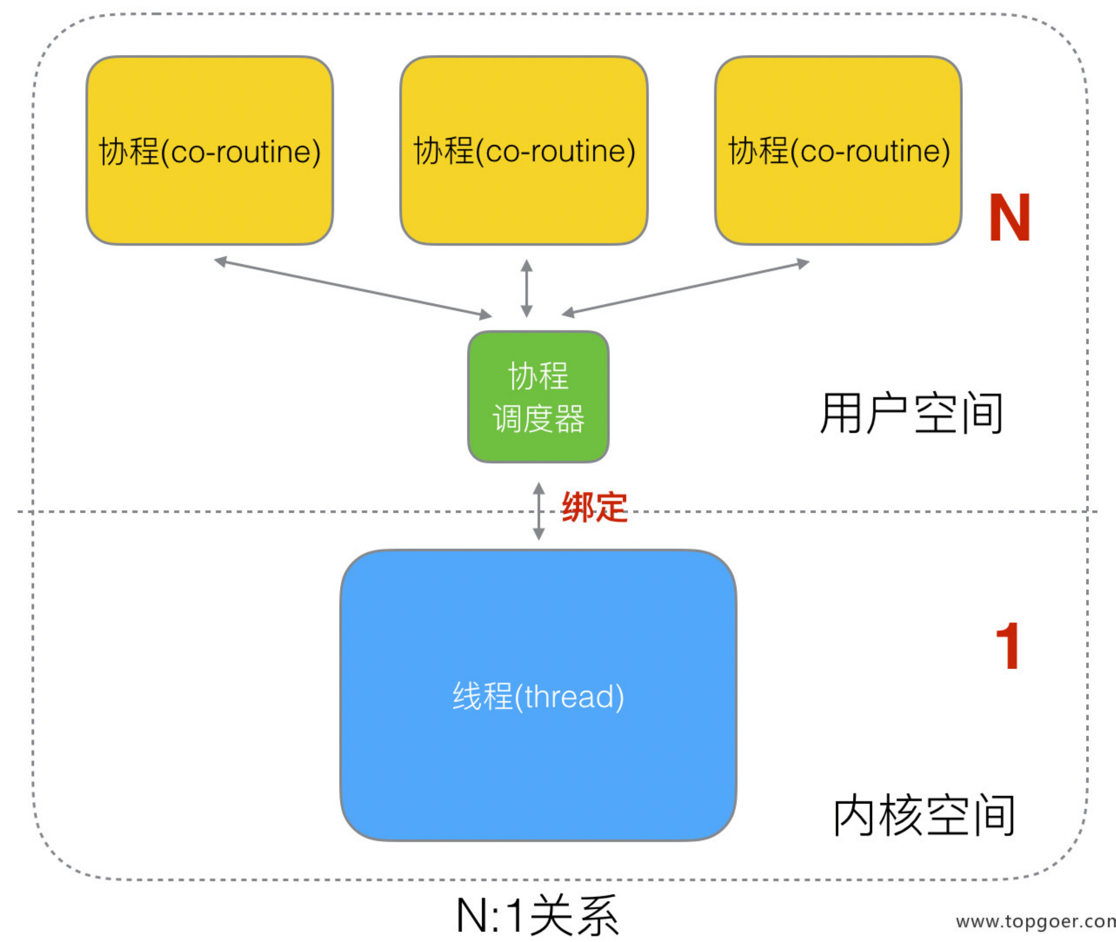
之后，我们就看到了有 3 中协程和线程的映射关系：

> s="default">
>
> N:1 关系

N 个协程绑定 1 个线程，优点就是协程在用户态线程即完成切换，不会陷入到内核态，这种切换非常的轻量快速。但也有很大的缺点，1 个进程的所有协程都绑定在 1 个线程上

缺点：

- 某个程序用不了硬件的多核加速能力
- 一旦某协程阻塞，造成线程阻塞，本进程的其他协程都无法执行了，根本就没有并发的能力了。

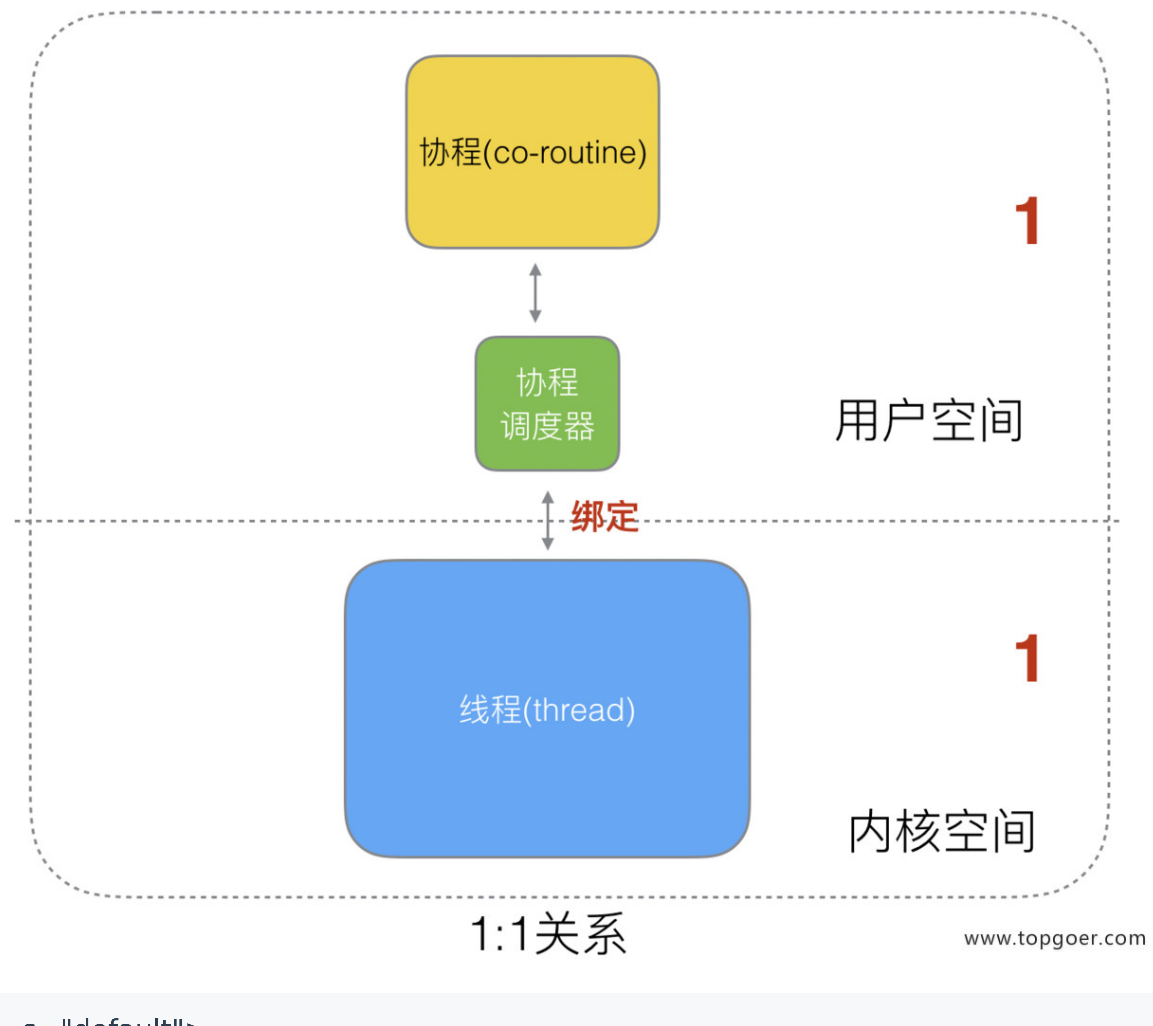


> s="default">
>
> 1:1 关系

1 个协程绑定 1 个线程，这种最容易实现。协程的调度都由 CPU 完成了，不存在 N:1 缺点，

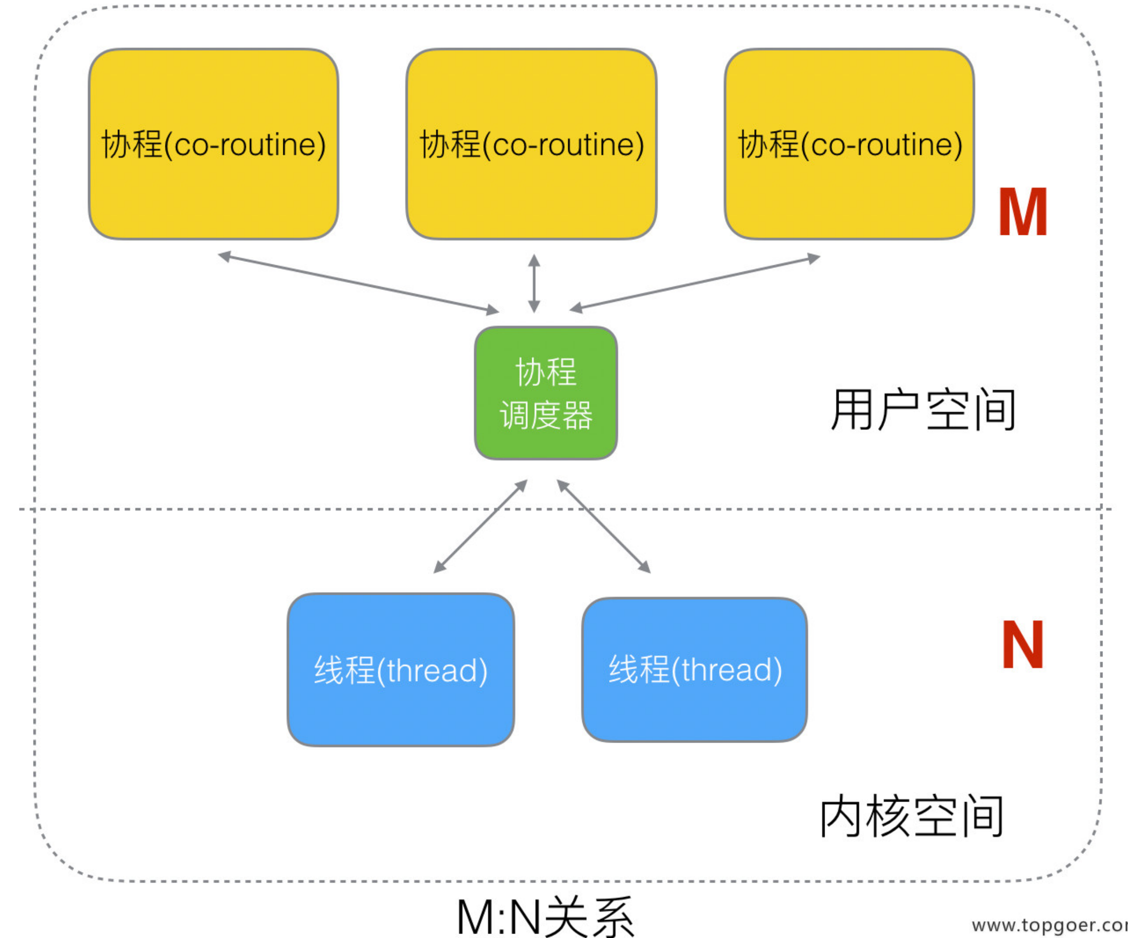
缺点：

- 协程的创建、删除和切换的代价都由 CPU 完成，有点略显昂贵了。



> s="default">
>
> M:N 关系

M 个协程绑定 1 个线程，是 N:1 和 1:1 类型的结合，克服了以上 2 种模型的缺点，但实现起来最为复杂。



协程跟线程是有区别的，线程由 CPU 调度是抢占式的，协程由用户态调度是协作式的，一个协程让出 CPU 后，才执行下一个协程。

(4) Go 语言的协程 goroutine

Go 为了提供更容易使用的并发方法，使用了 goroutine 和 channel。goroutine 来自协程的概念，让一组可复用的函数运行在一组线程之上，即使有协程阻塞，该线程的其他协程也可以被 runtime 调度，转移到其他可运行的线程上。最关键的是，程序员看不到这些底层的细节，这就降低了编程的难度，提供了更容易的并发。

Go 中，协程被称为 goroutine，它非常轻量，一个 goroutine 只占几 KB，并且这几 KB 就足够 goroutine 运行完，这就能在有限的内存空间内支持大量 goroutine，支持了更多的并发。虽然一个 goroutine 的栈只占几 KB，但实际是可伸缩的，如果需要更多内容，runtime 会自动为 goroutine 分配。

Goroutine 特点：

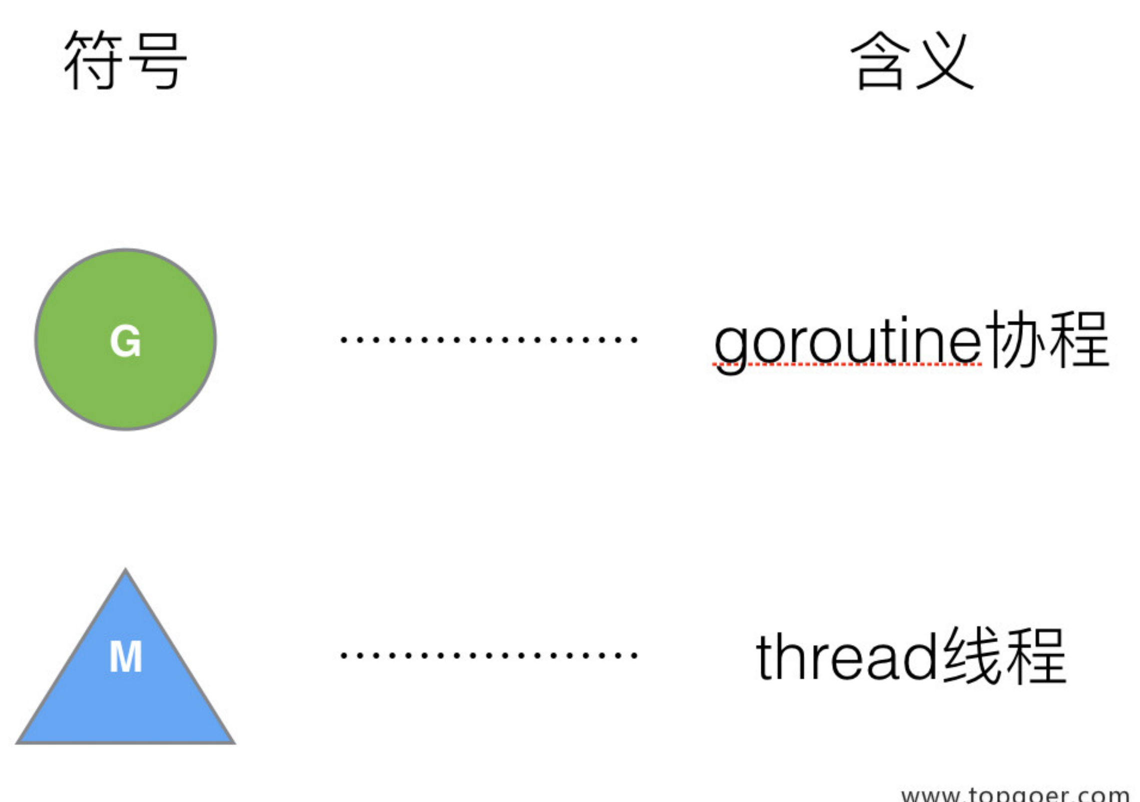
- 占用内存更小（几 kb）
- 调度更灵活 (runtime 调度)

(5) 被废弃的 goroutine 调度器

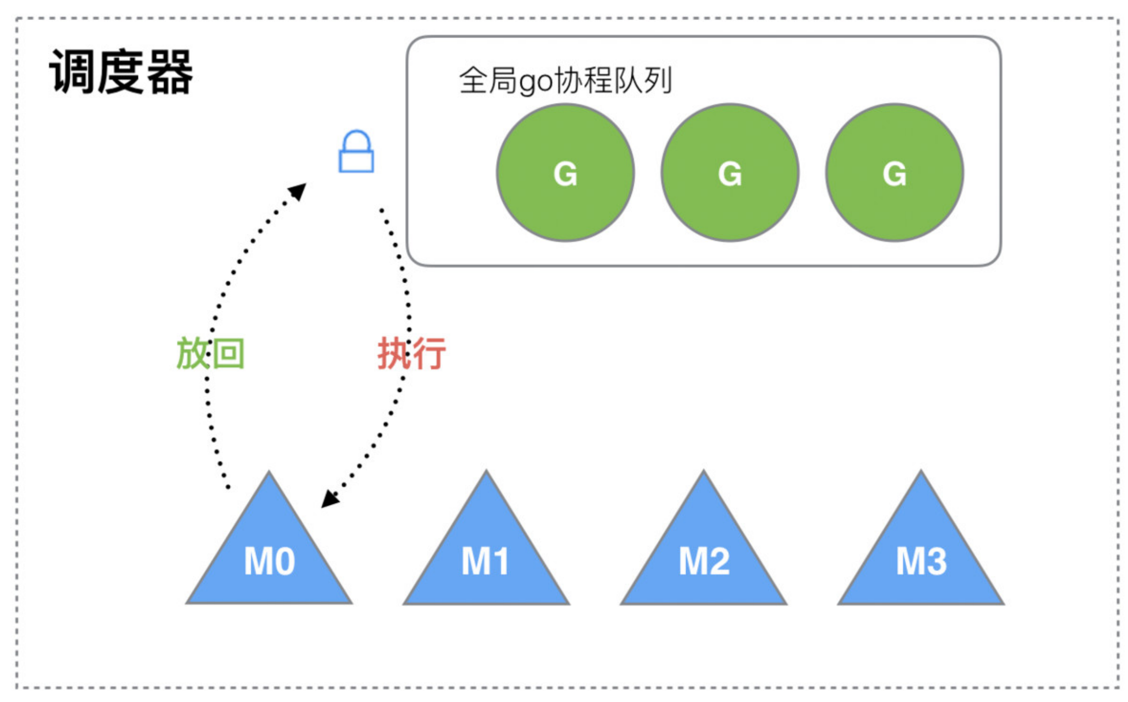
好了，既然我们知道了协程和线程的关系，那么最关键的一点就是调度协程的调度器的实现了。

Go 目前使用的调度器是 2012 年重新设计的，因为之前的调度器性能存在问题，所以使用 4 年就被废弃了，那么我们先来分析一下被废弃的调度器是如何运作的？

> 大部分文章都是会用 G 来表示 Goroutine，用 M 来表示线程，那么我们也会用这种表达的对应关系。



下面我们来看看被废弃的 golang 调度器是如何实现的？



M 想要执行、放回 G 都必须访问全局 G 队列，并且 M 有多个，即多线程访问同一资源需要加锁进行保证互斥 / 同步，所以全局 G 队列是有互斥锁进行保护的。

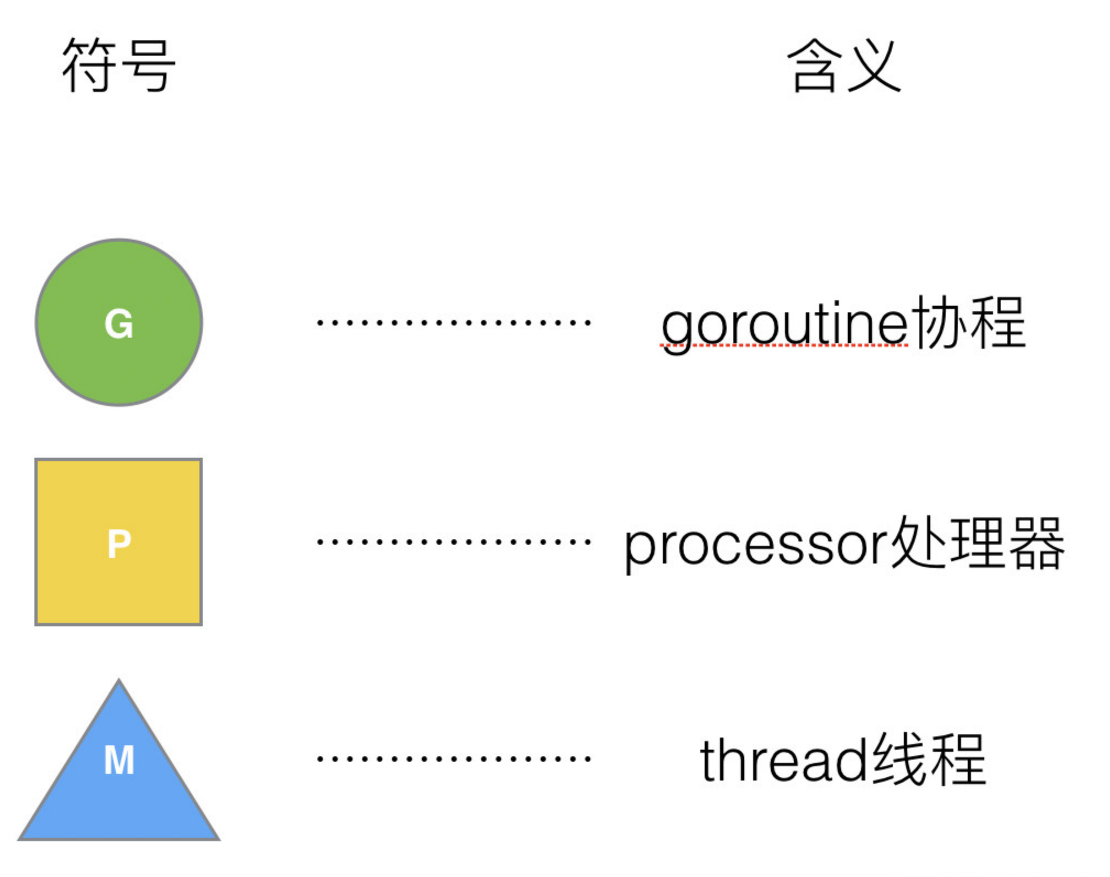
老调度器有几个缺点：

- 创建、销毁、调度 G 都需要每个 M 获取锁，这就形成了激烈的锁竞争。
- M 转移 G 会造成延迟和额外的系统负载。比如当 G 中包含创建新协程的时候，M 创建了 G'，为了继续执行 G，需要把 G'交给 M'执行，也造成了很差的局部性，因为 G'和 G 是相关的，最好放在 M 上执行，而不是其他 M'。
- 系统调用 (CPU 在 M 之间的切换) 导致频繁的线程阻塞和取消阻塞操作增加了系统开销。

二、Goroutine 调度器的 GMP 模型的设计思想

面对之前调度器的问题，Go 设计了新的调度器。

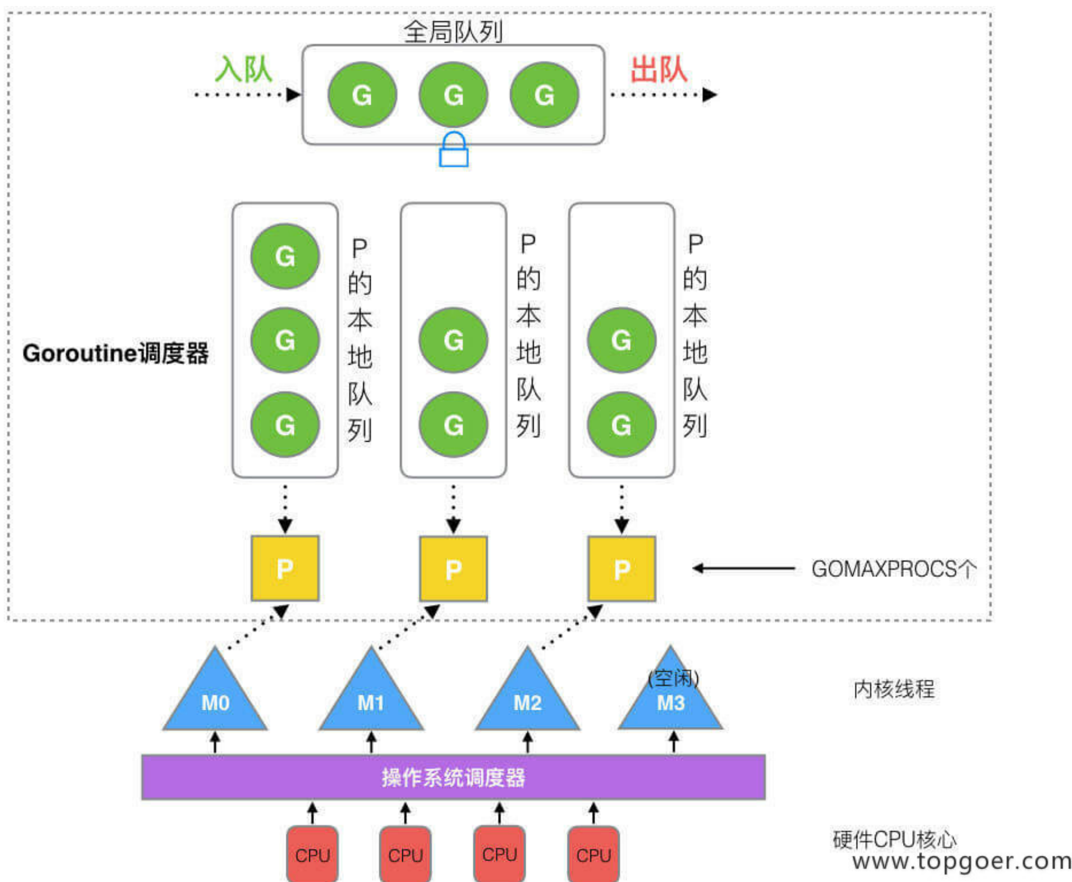
在新调度器中，出列 M (thread) 和 G (goroutine)，又引进了 P (Processor)。



Processor，它包含了运行 goroutine 的资源，如果线程想运行 goroutine，必须先获取 P，P 中还包含了可运行的 G 队列。

(1) GMP 模型

在 Go 中，线程是运行 goroutine 的实体，调度器的功能是把可运行的 goroutine 分配到工作线程上。



- 全局队列（Global Queue）：存放等待运行的 G。
- P 的本地队列：同全局队列类似，存放的也是等待运行的 G，存的数量有限，不超过 256 个。新建 G’时，G’优先加入到 P 的本地队列，如果队列满了，则会把本地队列中一半的 G 移动到全局队列。
- P 列表：所有的 P 都在程序启动时创建，并保存在数组中，最多有 GOMAXPROCS(可配置) 个。
- M：线程想运行任务就得获取 P，从 P 的本地队列获取 G，P 队列为空时，M 也会尝试从全局队列拿一批 G 放到 P 的本地队列，或从其他 P 的本地队列偷一半放到自己 P 的本地队列。M 运行 G，G 执行之后，M 会从 P 获取下一个 G，不断重复下去。

Goroutine 调度器和 OS 调度器是通过 M 结合起来的，每个 M 都代表了 1 个内核线程，OS 调度器负责把内核线程分配到 CPU 的核上执行。

> 有关 P 和 M 的个数问题

1、P 的数量：

- 由启动时环境变量 $GOMAXPROCS 或者是由 runtime 的方法 GOMAXPROCS() 决定。这意味着在程序执行的任意时刻都只有 $GOMAXPROCS 个 goroutine 在同时运行。

2、M 的数量:

- go 语言本身的限制：go 程序启动时，会设置 M 的最大数量，默认 10000. 但是内核很难支持这么多的线程数，所以这个限制可以忽略。
- runtime/debug 中的 SetMaxThreads 函数，设置 M 的最大数量
- 一个 M 阻塞了，会创建新的 M。

M 与 P 的数量没有绝对关系，一个 M 阻塞，P 就会去创建或者切换另一个 M，所以，即使 P 的默认数量是 1，也有可能会创建很多个 M 出来。

> P 和 M 何时会被创建

1、P 何时创建：在确定了 P 的最大数量 n 后，运行时系统会根据这个数量创建 n 个 P。

2、M 何时创建：没有足够的 M 来关联 P 并运行其中的可运行的 G。比如所有的 M 此时都阻塞住了，而 P 中还有很多就绪任务，就会去寻找空闲的 M，而没有空闲的，就会去创建新的 M。

(2) 调度器的设计策略

复用线程：避免频繁的创建、销毁线程，而是对线程的复用。

1）work stealing 机制

当本线程无可运行的 G 时，尝试从其他线程绑定的 P 偷取 G，而不是销毁线程。

2）hand off 机制

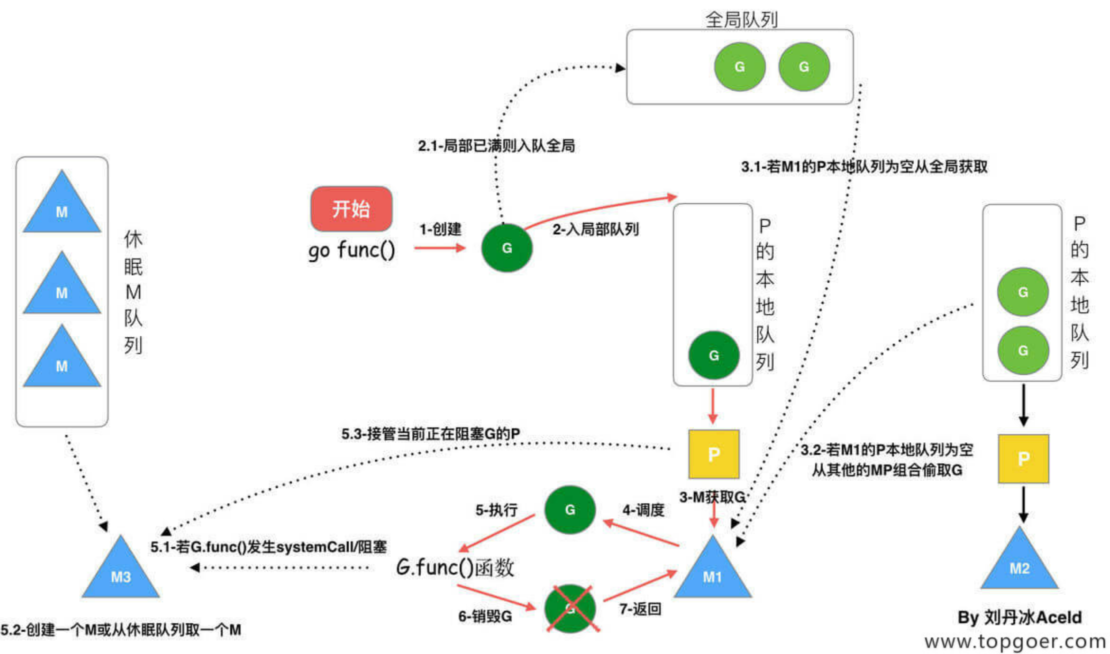
当本线程因为 G 进行系统调用阻塞时，线程释放绑定的 P，把 P 转移给其他空闲的线程执行。

利用并行：GOMAXPROCS 设置 P 的数量，最多有 GOMAXPROCS 个线程分布在多个 CPU 上同时运行。GOMAXPROCS 也限制了并发的程度，比如 GOMAXPROCS = 核数/2，则最多利用了一半的 CPU 核进行并行。

抢占：在 coroutine 中要等待一个协程主动让出 CPU 才执行下一个协程，在 Go 中，一个 goroutine 最多占用 CPU 10ms，防止其他 goroutine 被饿死，这就是 goroutine 不同于 coroutine 的一个地方。

全局 G 队列：在新的调度器中依然有全局 G 队列，但功能已经被弱化了，当 M 执行 work stealing 从其他 P 偷不到 G 时，它可以从全局 G 队列获取 G。

(3) go func () 调度流程



从上图我们可以分析出几个结论：

1、我们通过 go func () 来创建一个 goroutine；

2、有两个存储 G 的队列，一个是局部调度器 P 的本地队列、一个是全局 G 队列。新创建的 G 会先保存在 P 的本地队列中，如果 P 的本地队列已经满了就会保存在全局的队列中；

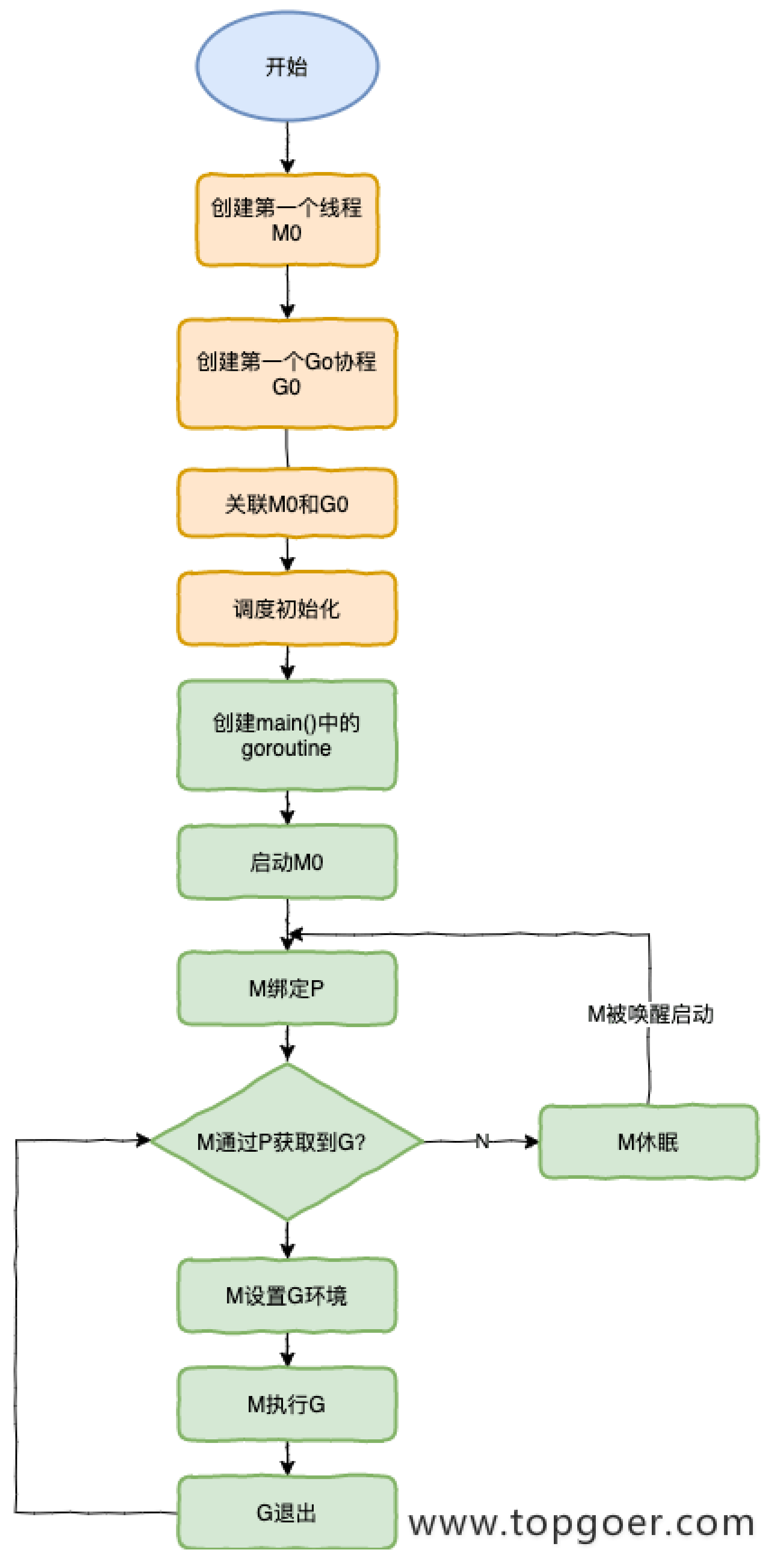
3、G 只能运行在 M 中，一个 M 必须持有一个 P，M 与 P 是 1：1 的关系。M 会从 P 的本地队列弹出一个可执行状态的 G 来执行，如果 P 的本地队列为空，就会想其他的 MP 组合偷取一个可执行的 G 来执行；

4、一个 M 调度 G 执行的过程是一个循环机制；

5、当 M 执行某一个 G 时候如果发生了 syscall 或则其余阻塞操作，M 会阻塞，如果当前有一些 G 在执行，runtime 会把这个线程 M 从 P 中摘除 (detach)，然后再创建一个新的操作系统的线程 (如果有空闲的线程可用就复用空闲线程) 来服务于这个 P；

6、当 M 系统调用结束时候，这个 G 会尝试获取一个空闲的 P 执行，并放入到这个 P 的本地队列。如果获取不到 P，那么这个线程 M 变成休眠状态， 加入到空闲线程中，然后这个 G 会被放入全局队列中。

(4) 调度器的生命周期



特殊的 M0 和 G0

M0

M0 是启动程序后的编号为 0 的主线程，这个 M 对应的实例会在全局变量 runtime.m0 中，不需要在 heap 上分配，M0 负责执行初始化操作和启动第一个 G， 在之后 M0 就和其他的 M 一样了。

G0

G0 是每次启动一个 M 都会第一个创建的 goroutine，G0 仅用于负责调度的 G，G0 不指向任何可执行的函数，每个 M 都会有一个自己的 G0。在调度或系统调用时会使用 G0 的栈空间，全局变量的 G0 是 M0 的 G0。

我们来跟踪一段代码

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world")
}
```

接下来我们来针对上面的代码对调度器里面的结构做一个分析。

也会经历如上图所示的过程：

- 1.runtime 创建最初的线程 m0 和 goroutine g0，并把 2 者关联。
- 2.调度器初始化：初始化 m0、栈、垃圾回收，以及创建和初始化由 GOMAXPROCS 个 P 构成的 P 列表。
- 3.示例代码中的 main 函数是 main.main，runtime 中也有 1 个 main 函数 —— runtime.main，代码经过编译后，runtime.main 会调用 main.main，程序启动时会为 runtime.main 创建 goroutine，称它为 main goroutine 吧，然后把 main goroutine 加入到 P 的本地队列。
- 4.启动 m0，m0 已经绑定了 P，会从 P 的本地队列获取 G，获取到 main goroutine。
- 5.G 拥有栈，M 根据 G 中的栈信息和调度信息设置运行环境
- 6.M 运行 G
- 7.G 退出，再次回到 M 获取可运行的 G，这样重复下去，直到 main.main 退出，runtime.main 执行 Defer 和 Panic 处理，或调用 runtime.exit 退出程序。

调度器的生命周期几乎占满了一个 Go 程序的一生，runtime.main 的 goroutine 执行之前都是为调度器做准备工作，runtime.main 的 goroutine 运行，才是调度器的真正开始，直到

runtime.main 结束而结束。

(5) 可视化 GMP 编程

有 2 种方式可以查看一个程序的 GMP 的数据。

方式 1：go tool trace

trace 记录了运行时的信息，能提供可视化的 Web 页面。

简单测试代码：main 函数创建 trace，trace 会运行在单独的 goroutine 中，然后 main 打印"Hello World" 退出。

trace.go

```go
package main

import (
    "os"
    "fmt"
    "runtime/trace"
)

func main() {

    //创建trace文件
    f, err := os.Create("trace.out")
    if err != nil {
        panic(err)
    }

    defer f.Close()

    //启动trace goroutine
    err = trace.Start(f)
    if err != nil {
        panic(err)
    }
    defer trace.Stop()

    //main
    fmt.Println("Hello World")
}
```

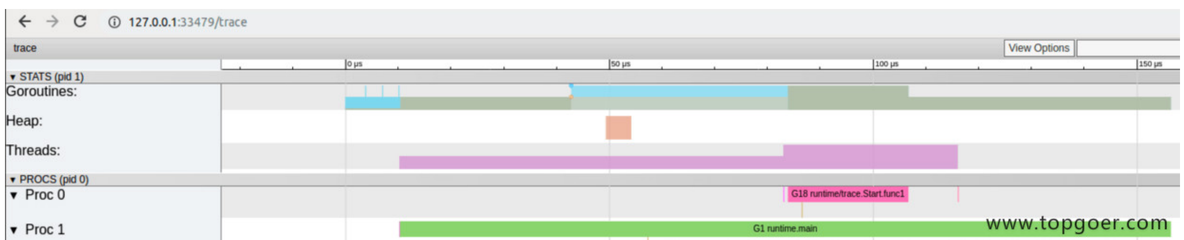
运行程序

```
$ go run trace.go
Hello World
```

会得到一个 trace.out 文件，然后我们可以用一个工具打开，来分析这个文件。

```
$ go tool trace trace.out
2020/02/23 10:44:11 Parsing trace...
2020/02/23 10:44:11 Splitting trace...
2020/02/23 10:44:11 Opening browser. Trace viewer is listening on http://127.0.0.1:33479
```

我们可以通过浏览器打开 http://127.0.0.1:33479 网址，点击 view trace 能够看见可视化的调度流程。



www.topgoer.com

G 信息

点击 Goroutines 那一行可视化的数据条，我们会看到一些详细的信息。

3 items selected. Counter Samples (3)

Counter	Series	Time	Value
Goroutines	GCWaiting	0.042808	0
Goroutines	Runnable	0.042808	1
Goroutines	Running	0.042808	1

G0

G1

www.topgoer.com

> 一共有两个G在程序中，一个是特殊的G0，是每个M必须有的一个初始化的G，这个我们不必讨论。

其中 G1 应该就是 main goroutine (执行 main 函数的协程)，在一段时间内处于可运行和运行的状态。

M 信息

点击 Threads 那一行可视化的数据条，我们会看到一些详细的信息。

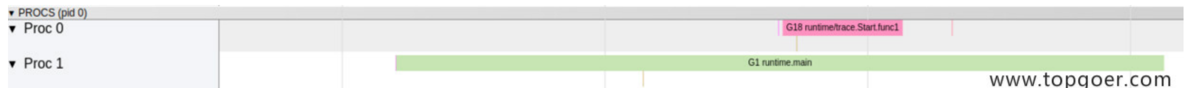
2 items selected. Counter Samples (2)

Counter	Series	Time	Value
Threads	InSyscall	0.010201	0
Threads	Running	0.010201	1

www.topgoer.com

一共有两个 M 在程序中，一个是特殊的 M0，用于初始化使用，这个我们不必讨论。

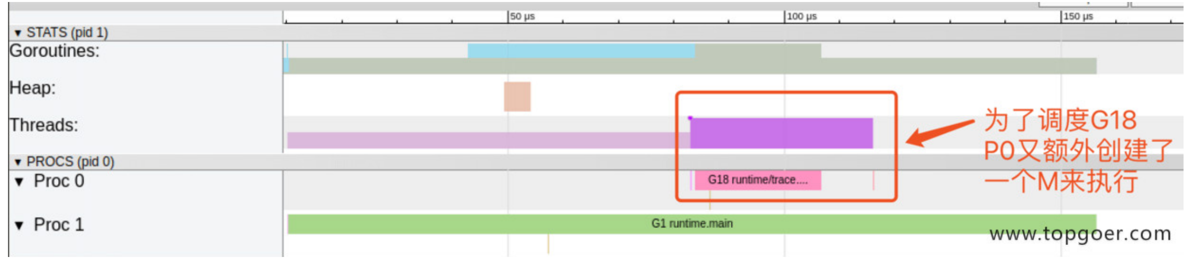
P 信息



G1 中调用了 main.main，创建了 trace goroutine g18。G1 运行在 P1 上，G18 运行在 P0 上。

这里有两个 P，我们知道，一个 P 必须绑定一个 M 才能调度 G。

我们在来看看上面的 M 信息。



我们会发现，确实 G18 在 P0 上被运行的时候，确实在 Threads 行多了一个 M 的数据，点击查看如下：

2 items selected. Counter Samples (2)

Counter	Series	Time	Value
Threads	InSyscall	0.083032	0
Threads	Running	0.083032	2

www.topgoer.com

多了一个 M2 应该就是 P0 为了执行 G18 而动态创建的 M2.

方式 2：Debug trace

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 5; i++ {
        time.Sleep(time.Second)
        fmt.Println("Hello World")
    }
}
```

编译

```
$ go build trace2.go
```

通过 Debug 方式运行

```
$ GODEBUG=schedtrace=1000 ./trace2
SCHED 0ms: gomaxprocs=2 idleprocs=0 threads=4 spinningthreads=1 idlethreads=1 ru
nqueue=0 [0 0]
Hello World
SCHED 1003ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2
runqueue=0 [0 0]
Hello World
SCHED 2014ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2
runqueue=0 [0 0]
Hello World
SCHED 3015ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2
runqueue=0 [0 0]
Hello World
SCHED 4023ms: gomaxprocs=2 idleprocs=2 threads=4 spinningthreads=0 idlethreads=2
runqueue=0 [0 0]
Hello World
```

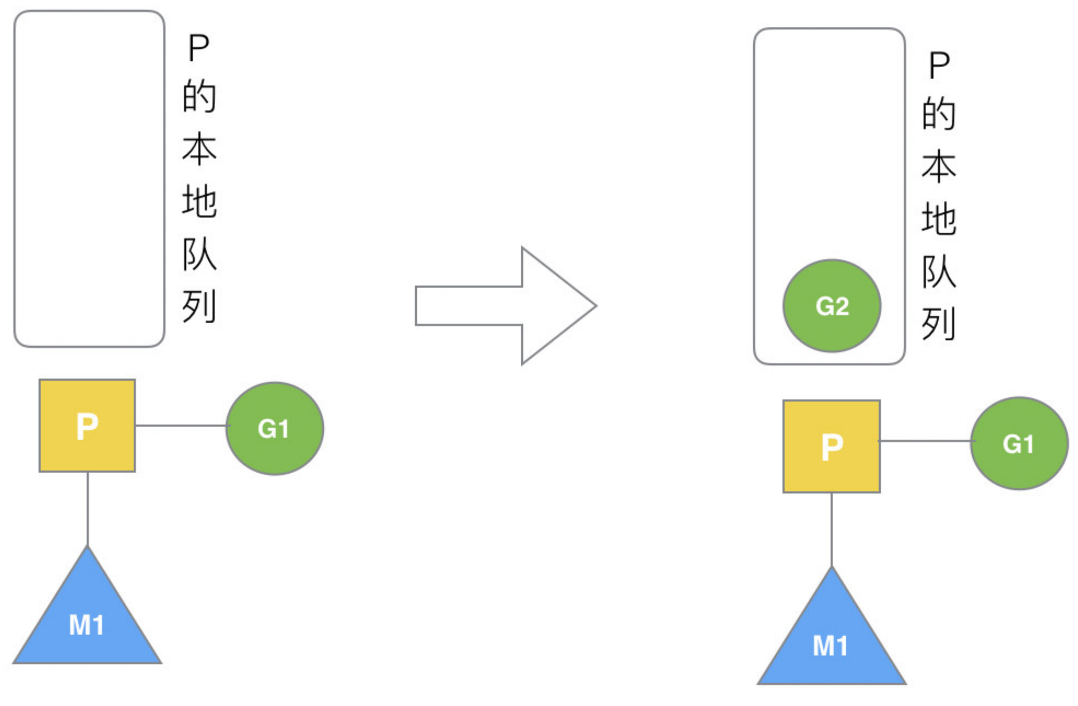
- `SCHED`：调试信息输出标志字符串，代表本行是 goroutine 调度器的输出；
- `0ms`：即从程序启动到输出这行日志的时间；

- `gomaxprocs`: P 的数量，本例有 2 个 P, 因为默认的 P 的属性是和 cpu 核心数量默认一致，当然也可以通过 GOMAXPROCS 来设置；
- `idleprocs`: 处于 idle 状态的 P 的数量；通过 gomaxprocs 和 idleprocs 的差值，我们就可知道执行 go 代码的 P 的数量；
- `threads`: os threads/M 的数量，包含 scheduler 使用的 m 数量，加上 runtime 自用的类似 sysmon 这样的 thread 的数量；
- `spinningthreads`: 处于自旋状态的 os thread 数量；
- `idlethread`: 处于 idle 状态的 os thread 的数量；
- `runqueue=0`： Scheduler 全局队列中 G 的数量；
- `[0 0]`: 分别为 2 个 P 的 local queue 中的 G 的数量。

三、Go 调度器调度场景过程全解析

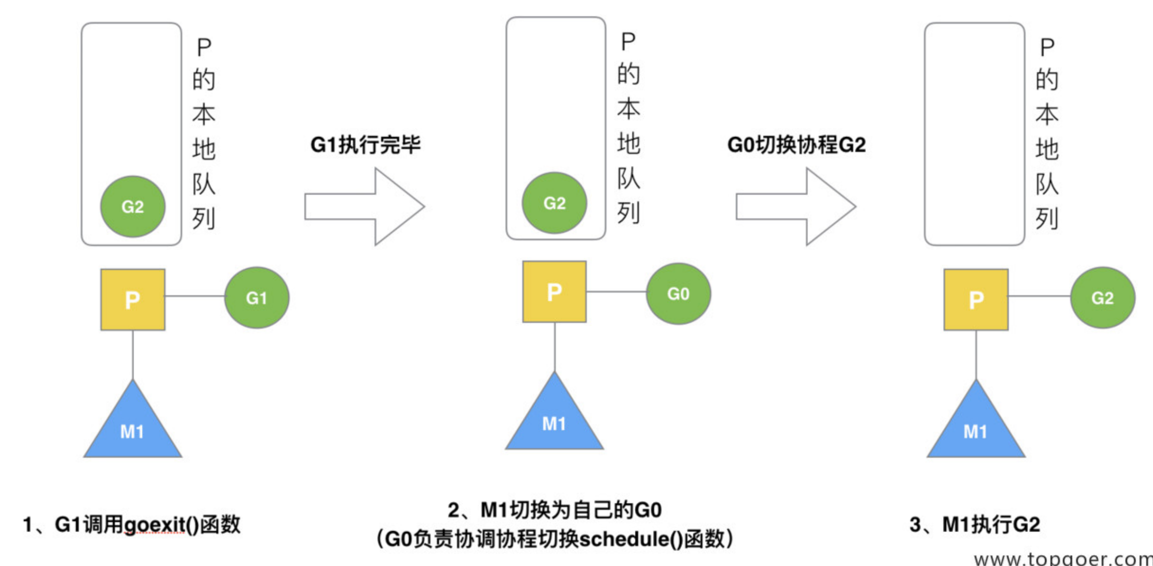
(1) 场景 1

P 拥有 G1，M1 获取 P 后开始运行 G1，G1 使用 go func() 创建了 G2，为了局部性 G2 优先加入到 P1 的本地队列。



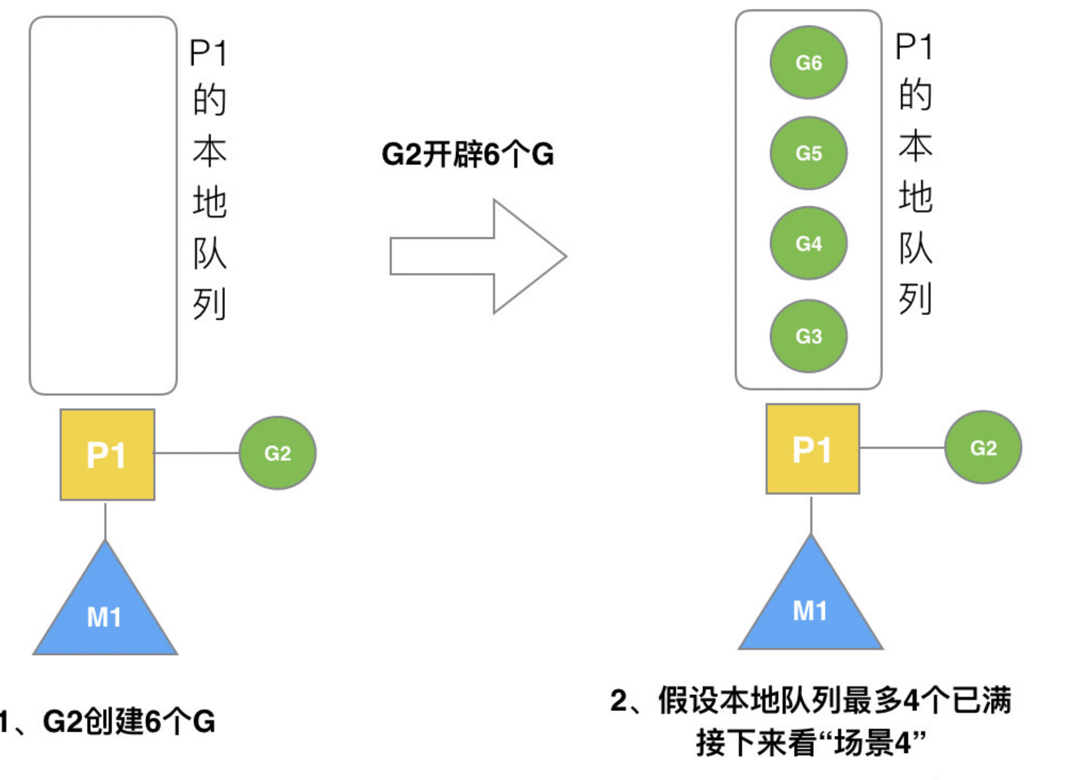
(2) 场景 2

G1 运行完成后 (函数：goexit)，M 上运行的 goroutine 切换为 G0，G0 负责调度时协程的切换（函数：schedule）。从 P 的本地队列取 G2，从 G0 切换到 G2，并开始运行 G2 (函数：execute)。实现了线程 M1 的复用。



(3) 场景 3

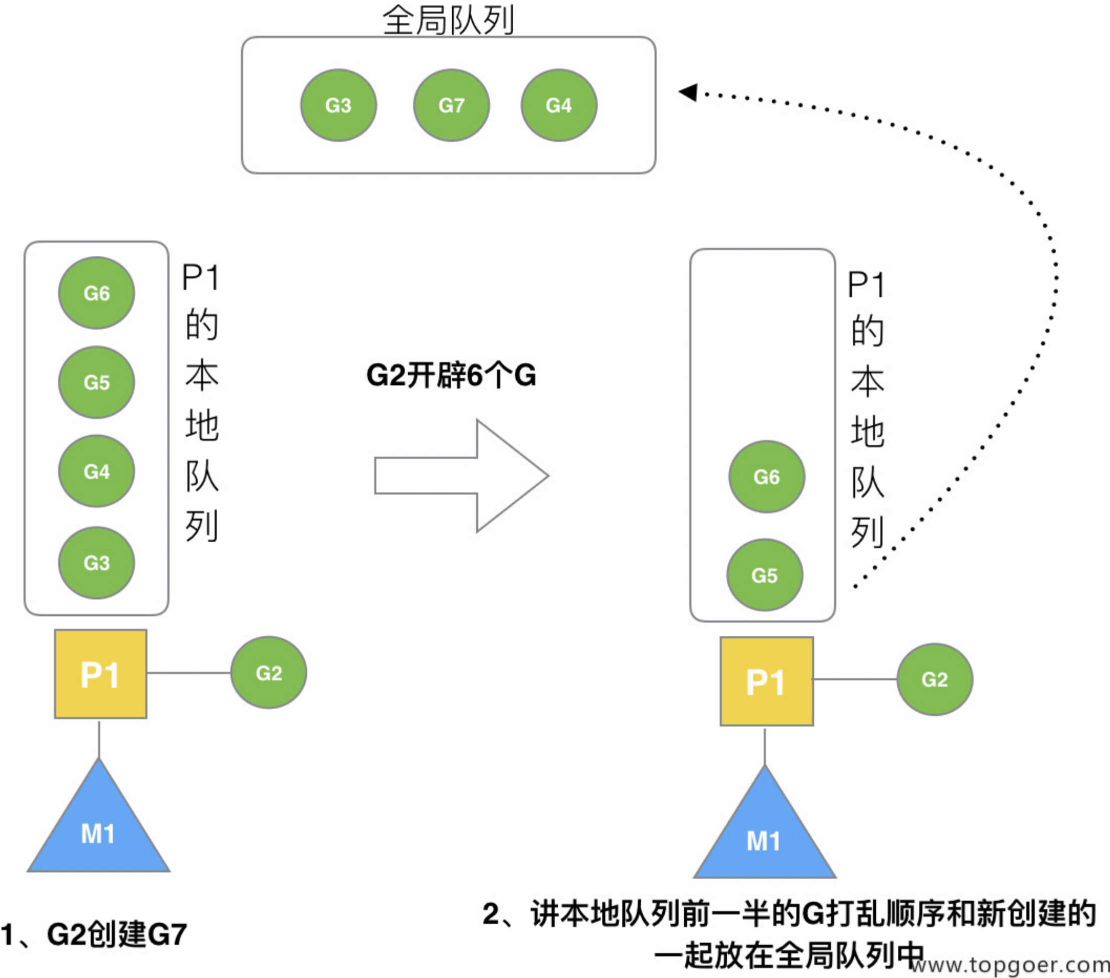
假设每个 P 的本地队列只能存 3 个 G。G2 要创建了 6 个 G，前 3 个 G（G3, G4, G5）已经加入 p1 的本地队列，p1 本地队列满了。



(4) 场景 4

G2 在创建 G7 的时候，发现 P1 的本地队列已满，需要执行负载均衡 (把 P1 中本地队列中前一半的 G，还有新创建 G 转移到全局队列)

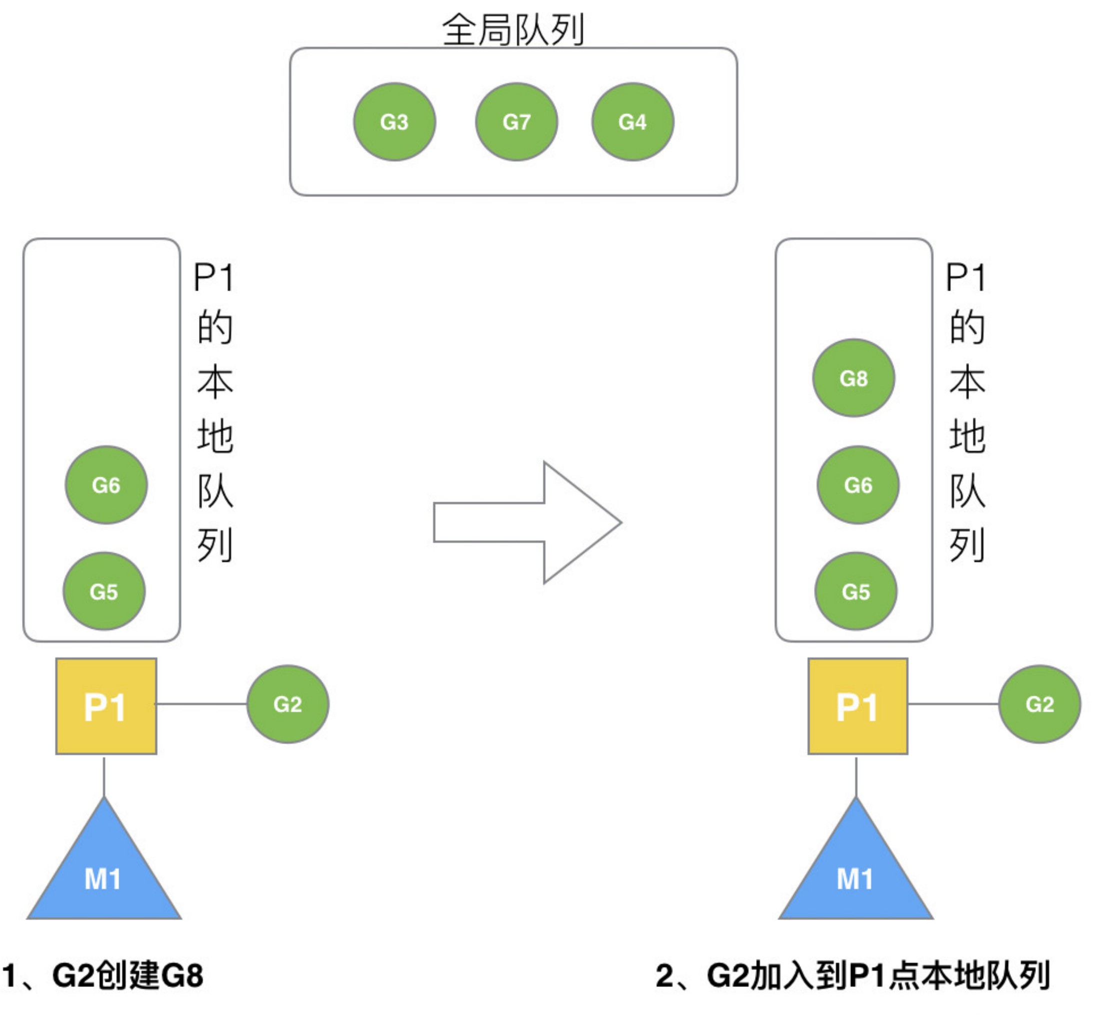
> （实现中并不一定是新的 G，如果 G 是 G2 之后就执行的，会被保存在本地队列，利用某个老的 G 替换新 G 加入全局队列）



这些 G 被转移到全局队列时，会被打乱顺序。所以 G3,G4,G7 被转移到全局队列。

(5) 场景 5

G2 创建 G8 时，P1 的本地队列未满，所以 G8 会被加入到 P1 的本地队列。



G8 加入到 P1 点本地队列的原因还是因为 P1 此时在与 M1 绑定，而 G2 此时是 M1 在执行。所以 G2 创建的新的 G 会优先放置到自己的 M 绑定的 P 上。

(6) 场景 6

规定：在创建 G 时，运行的 G 会尝试唤醒其他空闲的 P 和 M 组合去执行。

场景6：唤醒正在休眠的M

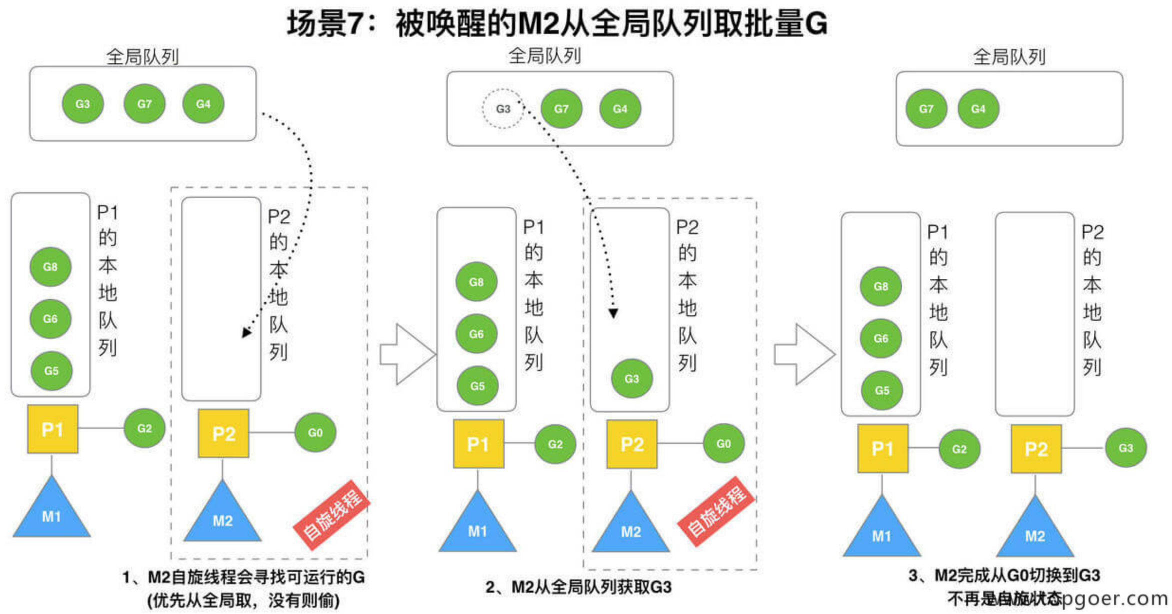
假定 G2 唤醒了 M2，M2 绑定了 P2，并运行 G0，但 P2 本地队列没有 G，M2 此时为自旋线程（没有 G 但为运行状态的线程，不断寻找 G）。

(7) 场景 7

M2 尝试从全局队列 (简称 “GQ”) 取一批 G 放到 P2 的本地队列（函数：findrunnable()）。M2 从全局队列取的 G 数量符合下面的公式：

```
n = min(len(GQ)/GOMAXPROCS + 1, len(GQ/2))
```

至少从全局队列取 1 个 g，但每次不要从全局队列移动太多的 g 到 p 本地队列，给其他 p 留点。这是从全局队列到 P 本地队列的负载均衡。



假定我们场景中一共有 4 个 P（GOMAXPROCS 设置为 4，那么我们允许最多就能用 4 个 P 来供 M 使用）。所以 M2 只从能从全局队列取 1 个 G（即 G3）移动 P2 本地队列，然后完成从 G0 到 G3 的切换，运行 G3。

(8) 场景 8

假设 G2 一直在 M1 上运行，经过 2 轮后，M2 已经把 G7、G4 从全局队列获取到了 P2 的本地队列并完成运行，全局队列和 P2 的本地队列都空了，如场景 8 图的左半部分。

场景8：M2从M1中偷取G

1、M2自旋线程会寻找可运行的G
(全局为空，则向其他的MP组合偷取)

2、M2从M1偷去到G8来执行

www.topgoer.com

全局队列已经没有 G，那 m 就要执行 work stealing (偷取)：从其他有 G 的 P 哪里偷取一半 G 过来，放到自己的 P 本地队列。P2 从 P1 的本地队列尾部取一半的 G，本例中一半则只有 1 个 G8，放到 P2 的本地队列并执行。

(9) 场景 9

G1 本地队列 G5、G6 已经被其他 M 偷走并运行完成，当前 M1 和 M2 分别在运行 G2 和 G8，M3 和 M4 没有 goroutine 可以运行，M3 和 M4 处于自旋状态，它们不断寻找 goroutine。

场景9：自旋线程的最大限制

全局队列

GOMAXPROCS=4

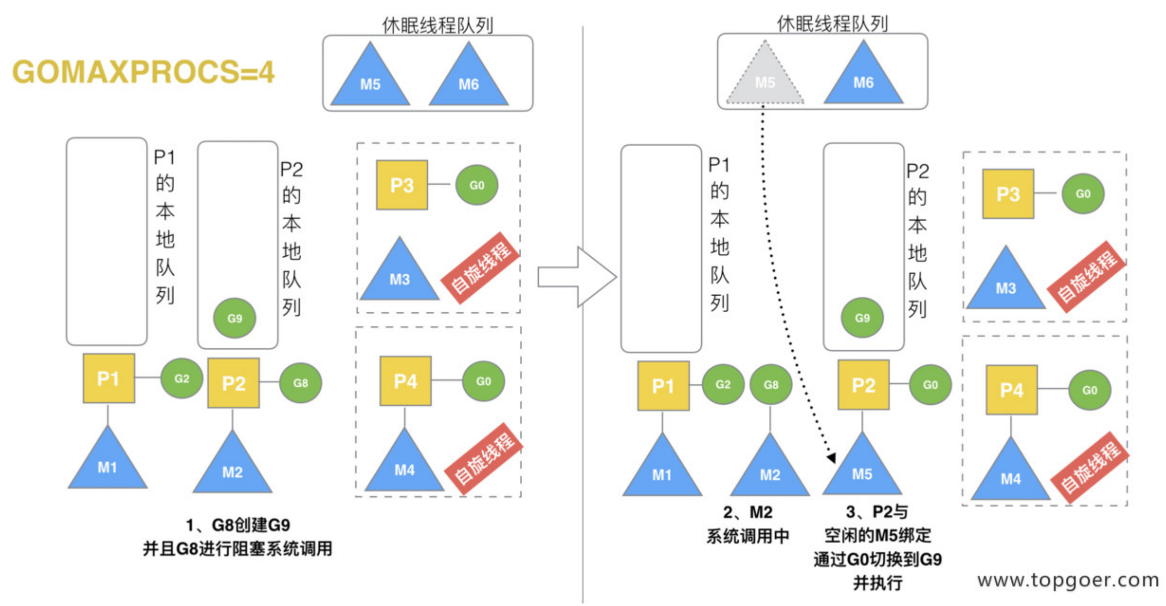
1、M1，M2正在执行G2、G8　　2、M3偷走了P1的G5，并执行完　　3、M4偷走了P1的G6，并执行完

www.topgoer.com

为什么要让 m3 和 m4 自旋，自旋本质是在运行，线程在运行却没有执行 G，就变成了浪费 CPU. 为什么不销毁现场，来节约 CPU 资源。因为创建和销毁 CPU 也会浪费时间，我们希望当有新 goroutine 创建时，立刻能有 M 运行它，如果销毁再新建就增加了时延，降低了效率。当然也考虑了过多的自旋线程是浪费 CPU，所以系统中最多有 GOMAXPROCS 个自旋的线程 (当前例子中的 GOMAXPROCS=4，所以一共 4 个 P)，多余的没事做线程会让他们休眠。

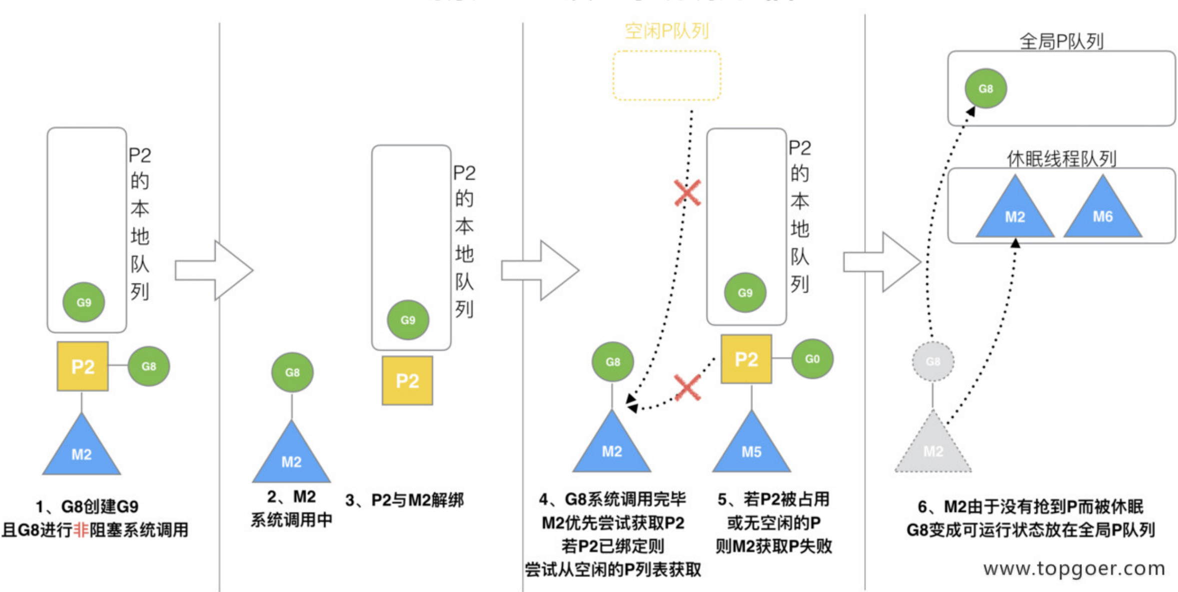
(10) 场景 10

假定当前除了 M3 和 M4 为自旋线程，还有 M5 和 M6 为空闲的线程 (没有得到 P 的绑定，注意我们这里最多就只能够存在 4 个 P，所以 P 的数量应该永远是 M>=P, 大部分都是 M 在抢占需要运行的 P)，G8 创建了 G9，G8 进行了阻塞的系统调用，M2 和 P2 立即解绑，P2 会执行以下判断：如果 P2 本地队列有 G、全局队列有 G 或有空闲的 M，P2 都会立马唤醒 1 个 M 和它绑定，否则 P2 则会加入到空闲 P 列表，等待 M 来获取可用的 p。本场景中，P2 本地队列有 G9，可以和其他空闲的线程 M5 绑定。



(11) 场景 11

G8 创建了 G9，假如 G8 进行了非阻塞系统调用。



M2 和 P2 会解绑，但 M2 会记住 P2，然后 G8 和 M2 进入系统调用状态。当 G8 和 M2 退出系统调用时，会尝试获取 P2，如果无法获取，则获取空闲的 P，如果依然没有，G8 会被记为可运行状态，并加入到全局队列，M2 因为没有 P 的绑定而变成休眠状态 (长时间休眠等待 GC 回收销毁)。

四、小结

总结，Go 调度器很轻量也很简单，足以撑起 goroutine 的调度工作，并且让 Go 具有了原生（强大）并发的能力。Go 调度本质是把大量的 goroutine 分配到少量线程上去执行，并利用多核并行，实现更强大的并发。

转自公众号：刘丹冰AceId

爬虫小案例

爬虫步骤

- 明确目标（确定在哪个网站搜索）
- 爬（爬下内容）
- 取（筛选想要的）
- 处理数据（按照你的想法去处理）

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
)

//这个只是一个简单的版本只是获取QQ邮箱并且没有进行封装操作，另外爬出来的数据也没有进行去重操作
var (
    // \d是数字
    reQQEmail = `(\d+)@qq.com`
)

// 爬邮箱
func GetEmail() {
    // 1.去网站拿数据
    resp, err := http.Get("https://tieba.baidu.com/p/6051076813?red_tag=1573533731")
    HandleError(err, "http.Get url")
    defer resp.Body.Close()
    // 2.读取页面内容
    pageBytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "ioutil.ReadAll")
    // 字节转字符串
    pageStr := string(pageBytes)
    //fmt.Println(pageStr)
    // 3.过滤数据，过滤qq邮箱
    re := regexp.MustCompile(reQQEmail)
    // -1代表取全部
```

```
    results := re.FindAllStringSubmatch(pageStr, -1)
    //fmt.Println(results)

    // 遍历结果
    for _, result := range results {
        fmt.Println("email:", result[0])
        fmt.Println("qq:", result[1])
    }
}

// 处理异常
func HandleError(err error, why string) {
    if err != nil {
        fmt.Println(why, err)
    }
}
func main() {
    GetEmail()
}
```

正则表达式

- 文档：https://studygolang.com/pkgdoc
- API
    - re := regexp.MustCompile(reStr)，传入正则表达式，得到正则表达式对象
    - ret := re.FindAllStringSubmatch(srcStr,-1)：用正则对象，获取页面页面，srcStr是页面内容，-1代表取全部
- 爬邮箱
- 方法抽取
- 爬超链接
- 爬手机号
    - http://www.zhaohaowang.com/ 如果连接失效了自己找一个有手机号的就好了
- 爬身份证号
    - http://henan.qq.com/a/20171107/069413.htm 如果连接失效了自己找一个就好了
- 爬图片链接

```
package main
```

```
import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
)

var (
    // w代表大小写字母+数字+下划线
    reEmail = `\w+@\w+\.\w+`
    // s?有或者没有s
    // +代表出1次或多次
    //\s\S各种字符
    // +?代表贪婪模式
    reLinke  = `href="(https?://[\s\S]+?)"`
    rePhone  = `1[3456789]\d\s?\d{4}\s?\d{4}`
    reIdcard = `[123456789]\d{5}((19\d{2})|(20[01]\d))((0[1-9])|(1[012]))((0[1-9
])|([12]\d)|(3[01]))\d{3}[\dXx]`
    reImg    = `https?://[^"]+?(\.((jpg)|(png)|(jpeg)|(gif)|(bmp)))`
)

// 处理异常
func HandleError(err error, why string) {
    if err != nil {
        fmt.Println(why, err)
    }
}

func GetEmail2(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reEmail)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result)
    }
}

// 抽取根据url获取内容
func GetPageStr(url string) (pageStr string) {
    resp, err := http.Get(url)
    HandleError(err, "http.Get url")
    defer resp.Body.Close()
    // 2.读取页面内容
    pageBytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "ioutil.ReadAll")
    // 字节转字符串
```

```
    pageStr = string(pageBytes)
    return pageStr
}

func main() {
    // 2.抽取的爬邮箱
    // GetEmail2("https://tieba.baidu.com/p/6051076813?red_tag=1573533731")
    // 3.爬链接
    //GetLink("http://www.baidu.com/s?wd=%E8%B4%B4%E5%90%A7%20%E7%95%99%E4%B8%8B%E9%82%AE%E7%AE%B1&rsv_spt=1&rsv_iqid=0x98ace53400003985&issp=1&f=8&rsv_bp=1&rsv_idx=2&ie=utf-8&tn=baiduhome_pg&rsv_enter=1&rsv_dl=ib&rsv_sug2=0&inputT=5197&rsv_sug4=6345")
    // 4.爬手机号
    //GetPhone("https://www.zhaohaowang.com/")
    // 5.爬身份证号
    //GetIdCard("https://henan.qq.com/a/20171107/069413.htm")
    // 6.爬图片
    // GetImg("http://image.baidu.com/search/index?tn=baiduimage&ps=1&ct=201326592&lm=-1&cl=2&nc=1&ie=utf-8&word=%E7%BE%8E%E5%A5%B3")
}

func GetIdCard(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reIdcard)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result)
    }
}

// 爬链接
func GetLink(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reLinke)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result[1])
    }
}

//爬手机号
func GetPhone(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(rePhone)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
```

```
        fmt.Println(result)
    }
}

func GetImg(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reImg)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result[0])
    }
}
```

并发爬取美图

下面的两个是即将要爬的网站，如果网址失效自己换一个就好了

- https://www.bizhizu.cn/shouji/tag-%E5%8F%AF%E7%88%B1/1.html

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
    "strconv"
    "strings"
    "sync"
    "time"
)

func HandleError(err error, why string) {
    if err != nil {
        fmt.Println(why, err)
    }
}

// 下载图片，传入的是图片叫什么
func DownloadFile(url string, filename string) (ok bool) {
    resp, err := http.Get(url)
    HandleError(err, "http.get.url")
    defer resp.Body.Close()
    bytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "resp.body")
```

```
    filename = "E:/topgoer.com/src/github.com/student/3.0/img/" + filename
    // 写出数据
    err = ioutil.WriteFile(filename, bytes, 0666)
    if err != nil {
        return false
    } else {
        return true
    }
}

// 并发爬思路:
// 1.初始化数据管道
// 2.爬虫写出：26个协程向管道中添加图片链接
// 3.任务统计协程：检查26个任务是否都完成，完成则关闭数据管道
// 4.下载协程：从管道里读取链接并下载

var (
    // 存放图片链接的数据管道
    chanImageUrls chan string
    waitGroup     sync.WaitGroup
    // 用于监控协程
    chanTask chan string
    reImg    = `https?://[^"]+?(\.((jpg)|(png)|(jpeg)|(gif)|(bmp)))`
)

func main() {
    // myTest()
    // DownloadFile("http://i1.shaodiyejin.com/uploads/tu/201909/10242/e5794daf58_4.jpg", "1.jpg")

    // 1.初始化管道
    chanImageUrls = make(chan string, 1000000)
    chanTask = make(chan string, 26)
    // 2.爬虫协程
    for i := 1; i < 27; i++ {
        waitGroup.Add(1)
        go getImgUrls("https://www.bizhizu.cn/shouji/tag-%E5%8F%AF%E7%88%B1/" + strconv.Itoa(i) + ".html")
    }
    // 3.任务统计协程，统计26个任务是否都完成，完成则关闭管道
    waitGroup.Add(1)
    go CheckOK()
    // 4.下载协程：从管道中读取链接并下载
    for i := 0; i < 5; i++ {
        waitGroup.Add(1)
        go DownloadImg()
```

```
    }
    waitGroup.Wait()
}

// 下载图片
func DownloadImg() {
    for url := range chanImageUrls {
        filename := GetFilenameFromUrl(url)
        ok := DownloadFile(url, filename)
        if ok {
            fmt.Printf("%s 下载成功\n", filename)
        } else {
            fmt.Printf("%s 下载失败\n", filename)
        }
    }
    waitGroup.Done()
}

// 截取url名字
func GetFilenameFromUrl(url string) (filename string) {
    // 返回最后一个/的位置
    lastIndex := strings.LastIndex(url, "/")
    // 切出来
    filename = url[lastIndex+1:]
    // 时间戳解决重名
    timePrefix := strconv.Itoa(int(time.Now().UnixNano()))
    filename = timePrefix + "_" + filename
    return
}

// 任务统计协程
func CheckOK() {
    var count int
    for {
        url := <-chanTask
        fmt.Printf("%s 完成了爬取任务\n", url)
        count++
        if count == 26 {
            close(chanImageUrls)
            break
        }
    }
    waitGroup.Done()
}

// 爬图片链接到管道
```

```go
// url是传的整页链接
func getImgUrls(url string) {
	urls := getImgs(url)
	// 遍历切片里所有链接，存入数据管道
	for _, url := range urls {
		chanImageUrls <- url
	}
	// 标识当前协程完成
	// 每完成一个任务，写一条数据
	// 用于监控协程知道已经完成了几个任务
	chanTask <- url
	waitGroup.Done()
}

// 获取当前页图片链接
func getImgs(url string) (urls []string) {
	pageStr := GetPageStr(url)
	re := regexp.MustCompile(reImg)
	results := re.FindAllStringSubmatch(pageStr, -1)
	fmt.Printf("共找到%d条结果\n", len(results))
	for _, result := range results {
		url := result[0]
		urls = append(urls, url)
	}
	return
}

// 抽取根据url获取内容
func GetPageStr(url string) (pageStr string) {
	resp, err := http.Get(url)
	HandleError(err, "http.Get url")
	defer resp.Body.Close()
	// 2.读取页面内容
	pageBytes, err := ioutil.ReadAll(resp.Body)
	HandleError(err, "ioutil.ReadAll")
	// 字节转字符串
	pageStr = string(pageBytes)
	return pageStr
}
```

数据操作

go操作MySQL

go操作Redis

go操作ETCD

zookeeper

go操作kafka

go操作RabbitMQ

go操作ElasticSearch

NSQ

GORM

xorm

go操作memcached

go操作MySQL

go操作MySQL

Insert操作

Select操作

Update操作

Delete操作

MySQL事务

go操作MySQL

新建test数据库，person、place 表

```
CREATE TABLE `person` (
    `user_id` int(11) NOT NULL AUTO_INCREMENT,
    `username` varchar(260) DEFAULT NULL,
    `sex` varchar(260) DEFAULT NULL,
    `email` varchar(260) DEFAULT NULL,
    PRIMARY KEY (`user_id`)
  ) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;

CREATE TABLE place (
    country varchar(200),
    city varchar(200),
    telcode int
)ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

```
    mysql> desc person;
    +----------+--------------+------+-----+---------+----------------+
    | Field    | Type         | Null | Key | Default | Extra          |
    +----------+--------------+------+-----+---------+----------------+
    | user_id  | int(11)      | NO   | PRI | NULL    | auto_increment |
    | username | varchar(260) | YES  |     | NULL    |                |
    | sex      | varchar(260) | YES  |     | NULL    |                |
    | email    | varchar(260) | YES  |     | NULL    |                |
    +----------+--------------+------+-----+---------+----------------+
    4 rows in set (0.00 sec)

    mysql> desc place;
    +---------+--------------+------+-----+---------+-------+
    | Field   | Type         | Null | Key | Default | Extra |
    +---------+--------------+------+-----+---------+-------+
    | country | varchar(200) | YES  |     | NULL    |       |
    | city    | varchar(200) | YES  |     | NULL    |       |
    | telcode | int(11)      | YES  |     | NULL    |       |
    +---------+--------------+------+-----+---------+-------+
    3 rows in set (0.01 sec)
```

mysql使用

使用第三方开源的mysql库: github.com/go-sql-driver/mysql （mysql驱动） github.com/jmoiron/sqlx （基于mysql驱动的封装）

命令行输入：

```
go get github.com/go-sql-driver/mysql
go get github.com/jmoiron/sqlx
```

链接mysql

```
database, err := sqlx.Open("mysql", "root:XXXX@tcp(127.0.0.1:3306)/test")
//database, err := sqlx.Open("数据库类型", "用户名:密码@tcp(地址:端口)/数据库名")
```

Insert操作

```
package main

import (
    "fmt"
    _ "github.com/go-sql-driver/mysql"
    "github.com/jmoiron/sqlx"
)

type Person struct {
    UserId   int    `db:"user_id"`
    Username string `db:"username"`
    Sex      string `db:"sex"`
    Email    string `db:"email"`
}

type Place struct {
    Country string `db:"country"`
    City    string `db:"city"`
    TelCode int    `db:"telcode"`
}

var Db *sqlx.DB

func init() {
    database, err := sqlx.Open("mysql", "root:root@tcp(127.0.0.1:3306)/test")
    if err != nil {
        fmt.Println("open mysql failed,", err)
        return
    }
    Db = database
    defer db.Close()  // 注意这行代码要写在上面err判断的下面
}

func main() {
    r, err := Db.Exec("insert into person(username, sex, email)values(?, ?, ?)",
"stu001", "man", "stu01@qq.com")
    if err != nil {
        fmt.Println("exec failed, ", err)
        return
    }
    id, err := r.LastInsertId()
    if err != nil {
```

```
        fmt.Println("exec failed, ", err)
        return
    }

    fmt.Println("insert succ:", id)
}
```

Select操作

```
package main

import (
    "fmt"

    _ "github.com/go-sql-driver/mysql"
    "github.com/jmoiron/sqlx"
)

type Person struct {
    UserId   int    `db:"user_id"`
    Username string `db:"username"`
    Sex      string `db:"sex"`
    Email    string `db:"email"`
}

type Place struct {
    Country string `db:"country"`
    City    string `db:"city"`
    TelCode int    `db:"telcode"`
}

var Db *sqlx.DB

func init() {

    database, err := sqlx.Open("mysql", "root:root@tcp(127.0.0.1:3306)/test")
    if err != nil {
        fmt.Println("open mysql failed,", err)
        return
    }

    Db = database
    defer db.Close()  // 注意这行代码要写在上面err判断的下面
}

func main() {

    var person []Person
    err := Db.Select(&person, "select user_id, username, sex, email from person
 where user_id=?", 1)
    if err != nil {
```

```
        fmt.Println("exec failed, ", err)
        return
    }

    fmt.Println("select succ:", person)
}
```

输出结果:

```
    select succ: [{1 stu001 man stu01@qq.com}]
```

Update操作

```
package main

import (
    "fmt"

    _ "github.com/go-sql-driver/mysql"
    "github.com/jmoiron/sqlx"
)

type Person struct {
    UserId   int    `db:"user_id"`
    Username string `db:"username"`
    Sex      string `db:"sex"`
    Email    string `db:"email"`
}

type Place struct {
    Country string `db:"country"`
    City    string `db:"city"`
    TelCode int    `db:"telcode"`
}

var Db *sqlx.DB

func init() {

    database, err := sqlx.Open("mysql", "root:root@tcp(127.0.0.1:3306)/test")
    if err != nil {
        fmt.Println("open mysql failed,", err)
        return
    }

    Db = database
    defer db.Close()  // 注意这行代码要写在上面err判断的下面
}

func main() {

    res, err := Db.Exec("update person set username=? where user_id=?", "stu000
3", 1)
    if err != nil {
        fmt.Println("exec failed, ", err)
```

```
        return
    }
    row, err := res.RowsAffected()
    if err != nil {
        fmt.Println("rows failed, ",err)
    }
    fmt.Println("update succ:",row)

}
```

运行结果：

第一次运行：

```
    update succ: 1
    #受影响的行数 1
```

第二次运行

```
    update succ: 0
    #受影响的行数 0
```

Delete操作

```
package main

import (
	"fmt"

	_ "github.com/go-sql-driver/mysql"
	"github.com/jmoiron/sqlx"
)

type Person struct {
	UserId   int    `db:"user_id"`
	Username string `db:"username"`
	Sex      string `db:"sex"`
	Email    string `db:"email"`
}

type Place struct {
	Country string `db:"country"`
	City    string `db:"city"`
	TelCode int    `db:"telcode"`
}

var Db *sqlx.DB

func init() {

	database, err := sqlx.Open("mysql", "root:root@tcp(127.0.0.1:3306)/test")
	if err != nil {
		fmt.Println("open mysql failed,", err)
		return
	}

	Db = database
	defer db.Close()  // 注意这行代码要写在上面err判断的下面
}

func main() {

	/*
	_, err := Db.Exec("delete from person where user_id=?", 1)
	if err != nil {
		fmt.Println("exec failed, ", err)
```

```
        return
    }
    */

    res, err := Db.Exec("delete from person where user_id=?", 1)
    if err != nil {
        fmt.Println("exec failed, ", err)
        return
    }

    row,err := res.RowsAffected()
    if err != nil {
        fmt.Println("rows failed, ",err)
    }

    fmt.Println("delete succ: ",row)
}
```

运行结果：

第一次运行：

```
    delete succ: 1
    #受影响的行数 1
```

第二次运行：

```
    delete succ: 0
    #受影响的行数 0
```

MySQL事务

mysql事务特性：

```
    1) 原子性
    2) 一致性
    3) 隔离性
    4) 持久性
```

golang MySQL事务应用：

```
    1)  import ( “github.com/jmoiron/sqlx")
    2)  Db.Begin()          开始事务
    3)  Db.Commit()          提交事务
    4)  Db.Rollback()       回滚事务
```

```
package main

    import (
        "fmt"

        _ "github.com/go-sql-driver/mysql"
        "github.com/jmoiron/sqlx"
    )

    type Person struct {
        UserId   int    `db:"user_id"`
        Username string `db:"username"`
        Sex      string `db:"sex"`
        Email    string `db:"email"`
    }

    type Place struct {
        Country string `db:"country"`
        City    string `db:"city"`
        TelCode int    `db:"telcode"`
    }

    var Db *sqlx.DB

    func init() {
        database, err := sqlx.Open("mysql", "root:root@tcp(127.0.0.1:3306)/test"
)
```

```
        if err != nil {
            fmt.Println("open mysql failed,", err)
            return
        }
        Db = database
    }

    func main() {
        conn, err := Db.Begin()
        if err != nil {
            fmt.Println("begin failed :", err)
            return
        }

        r, err := conn.Exec("insert into person(username, sex, email)values(?, ?, ?)", "stu001", "man", "stu01@qq.com")
        if err != nil {
            fmt.Println("exec failed, ", err)
            conn.Rollback()
            return
        }
        id, err := r.LastInsertId()
        if err != nil {
            fmt.Println("exec failed, ", err)
            conn.Rollback()
            return
        }
        fmt.Println("insert succ:", id)

        r, err = conn.Exec("insert into person(username, sex, email)values(?, ?, ?)", "stu001", "man", "stu01@qq.com")
        if err != nil {
            fmt.Println("exec failed, ", err)
            conn.Rollback()
            return
        }
        id, err = r.LastInsertId()
        if err != nil {
            fmt.Println("exec failed, ", err)
            conn.Rollback()
            return
        }
        fmt.Println("insert succ:", id)

        conn.Commit()
    }
```

输出结果：

```
insert succ: 2
insert succ: 3
```

查看MySQL：

```
mysql> select * from person;
+---------+----------+------+-------------+
| user_id | username | sex  | email       |
+---------+----------+------+-------------+
|       2 | stu001   | man  | stu01@qq.com |
|       3 | stu001   | man  | stu01@qq.com |
+---------+----------+------+-------------+
2 rows in set (0.00 sec)
```

go操作Redis

Redis介绍

链接Redis

String类型Set、Get操作

String批量操作

设置过期时间

List队列操作

Hash表

Redis连接池

Redis介绍

Redis 简介

Redis 是完全开源免费的，遵守BSD协议，是一个高性能的key-value数据库。

Redis 与其他 key - value 缓存产品有以下三个特点：

```
    Redis支持数据的持久化，可以将内存中的数据保存在磁盘中，重启的时候可以再次加载进行使用。
    Redis不仅仅支持简单的key-value类型的数据，同时还提供string、list（链表）、set（集合）、hash表等数据结构的存储。
    Redis支持数据的备份，即master-slave模式的数据备份。
```

Redis 优势

性能极高 – Redis能读的速度是110000次/s,写的速度是81000次/s ，单机能够达到15w qps，通常适合做缓存。

丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hashes, Sets 及 Ordered Sets 数据类型操作。

原子 – Redis的所有操作都是原子性的，意思就是要么成功执行要么失败完全不执行。单个操作是原子性的。多个操作也支持事务，即原子性，通过MULTI和EXEC指令包起来。

丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

Redis与其他key-value存储有什么不同？

Redis有着更为复杂的数据结构并且提供对他们的原子性操作，这是一个不同于其他数据库的进化路径。Redis的数据类型都是基于基本数据结构的同时对程序员透明，无需进行额外的抽象。

Redis运行在内存中但是可以持久化到磁盘，所以在对不同数据集进行高速读写时需要权衡内存，因为数据量不能大于硬件内存。在内存数据库方面的另一个优点是，相比在磁盘上相同的复杂的数据结构，在内存中操作起来非常简单，这样Redis可以做很多内部复杂性很强的事情。同时，在磁盘格式方面他们是紧凑的以追加的方式产生的，因为他们并不需要进行随机访问。

redis使用

使用第三方开源的redis库: github.com/garyburd/redigo/redis

命令行输入 ：

```
go get github.com/garyburd/redigo/redis
```

链接Redis

```go
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

func main() {
    c, err := redis.Dial("tcp", "localhost:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    fmt.Println("redis conn success")

    defer c.Close()
}
```

String类型Set、Get操作

```
package main

import (
	"fmt"
	"github.com/garyburd/redigo/redis"
)

func main() {
	c, err := redis.Dial("tcp", "localhost:6379")
	if err != nil {
		fmt.Println("conn redis failed,", err)
		return
	}

	defer c.Close()
	_, err = c.Do("Set", "abc", 100)
	if err != nil {
		fmt.Println(err)
		return
	}

	r, err := redis.Int(c.Do("Get", "abc"))
	if err != nil {
		fmt.Println("get abc failed,", err)
		return
	}

	fmt.Println(r)
}
```

运行结果：

```
	MISCONF Redis is configured to save RDB snapshots, but is currently not able to persist on disk. Commands that may modify the data set are disabled. Please check Redis logs for details about the error.
```

Redis被配置为保存数据库快照，但它目前不能持久化到硬盘。用来修改集合数据的命令不能用。请查看Redis日志的详细错误信息。

原因：

强制关闭Redis快照导致不能持久化。

解决方案：

运行config set stop-writes-on-bgsave-error no　命令后，关闭配置项stop-writes-on-bgsave-error解决该问题。

开启命令行新窗口 2：

链接Redis：

```
redis-cli -h 127.0.0.1 -p 6379
127.0.0.1:6379> config set stop-writes-on-bgsave-error no
OK
```

返回命令行窗口 1 运行程序：

```
go run main.go
```

输出结果：

```
100
```

命令行窗口 2：

```
127.0.0.1:6379> get abc
"100"
```

String批量操作

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

func main() {
    c, err := redis.Dial("tcp", "localhost:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    defer c.Close()
    _, err = c.Do("MSet", "abc", 100, "efg", 300)
    if err != nil {
        fmt.Println(err)
        return
    }

    r, err := redis.Ints(c.Do("MGet", "abc", "efg"))
    if err != nil {
        fmt.Println("get abc failed,", err)
        return
    }

    for _, v := range r {
        fmt.Println(v)
    }
}
```

运行结果：

```
    100
    300
```

redis窗口：

```
    127.0.0.1:6379> mget abc efg
    1) "100"
```

```
    2) "300"
```

设置过期时间

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

func main() {
    c, err := redis.Dial("tcp", "localhost:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    defer c.Close()
    _, err = c.Do("expire", "abc", 10)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

命令行运行：

```
    go run main.go
```

Redis命令行窗口：

```
    127.0.0.1:6379> get abc
    "100"

    # 10秒后过期
    127.0.0.1:6379> get abc
    (nil)
```

List队列操作

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

func main() {
    c, err := redis.Dial("tcp", "localhost:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    defer c.Close()
    _, err = c.Do("lpush", "book_list", "abc", "ceg", 300)
    if err != nil {
        fmt.Println(err)
        return
    }

    r, err := redis.String(c.Do("lpop", "book_list"))
    if err != nil {
        fmt.Println("get abc failed,", err)
        return
    }

    fmt.Println(r)
}
```

运行结果：

```
    300
```

Redis命令行：

```
    127.0.0.1:6379> lpop book_list
    "ceg"
    127.0.0.1:6379> lpop book_list
    "abc"
```

```
127.0.0.1:6379> lpop book_list
(nil)
```

Hash表

```
package main

import (
    "fmt"
    "github.com/garyburd/redigo/redis"
)

func main() {
    c, err := redis.Dial("tcp", "localhost:6379")
    if err != nil {
        fmt.Println("conn redis failed,", err)
        return
    }

    defer c.Close()
    _, err = c.Do("HSet", "books", "abc", 100)
    if err != nil {
        fmt.Println(err)
        return
    }

    r, err := redis.Int(c.Do("HGet", "books", "abc"))
    if err != nil {
        fmt.Println("get abc failed,", err)
        return
    }

    fmt.Println(r)
}
```

运行结果：

```
    100
```

Redis命令行：

```
    127.0.0.1:6379> hget books abc
    "100"
```

Redis连接池

```
package main
import(
    "fmt"
    "github.com/garyburd/redigo/redis"
)

var pool *redis.Pool  //创建redis连接池

func init(){
    pool = &redis.Pool{     //实例化一个连接池
        MaxIdle:16,    //最初的连接数量
        // MaxActive:1000000,    //最大连接数量
        MaxActive:0,    //连接池最大连接数量,不确定可以用0（0表示自动定义），按需分配
        IdleTimeout:300,    //连接关闭时间 300秒 （300秒不使用自动关闭）
        Dial: func() (redis.Conn ,error){     //要连接的redis数据库
            return redis.Dial("tcp","localhost:6379")
        },
    }
}

func main(){
    c := pool.Get() //从连接池，取一个链接
    defer c.Close() //函数运行结束 ，把连接放回连接池

        _,err := c.Do("Set","abc",200)
        if err != nil {
            fmt.Println(err)
            return
        }

        r,err := redis.Int(c.Do("Get","abc"))
        if err != nil {
            fmt.Println("get abc faild :",err)
            return
        }
        fmt.Println(r)
        pool.Close() //关闭连接池
}
```

运行结果：

```
200
```

Redis命令行：

```
127.0.0.1:6379> get abc
"200"
```

go操作ETCD

ETCD介绍

操作ETCD

ETCD介绍

etcd介绍

etcd是使用Go语言开发的一个开源的、高可用的分布式key-value存储系统，可以用于配置共享和服务的注册和发现。

类似项目有zookeeper和consul。

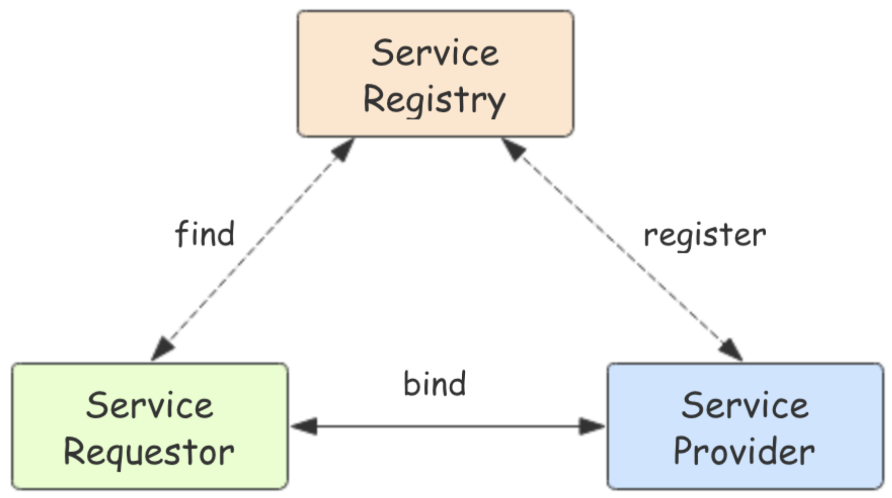
etcd具有以下特点：

- 完全复制：集群中的每个节点都可以使用完整的存档
- 高可用性：Etcd可用于避免硬件的单点故障或网络问题
- 一致性：每次读取都会返回跨多主机的最新写入
- 简单：包括一个定义良好、面向用户的API（gRPC）
- 安全：实现了带有可选的客户端证书身份验证的自动化TLS
- 快速：每秒10000次写入的基准速度
- 可靠：使用Raft算法实现了强一致、高可用的服务存储目录

etcd应用场景

服务发现

服务发现要解决的也是分布式系统中最常见的问题之一，即在同一个分布式集群中的进程或服务，要如何才能找到对方并建立连接。本质上来说，服务发现就是想要了解集群中是否有进程在监听 udp 或 tcp 端口，并且通过名字就可以查找和连接。



配置中心

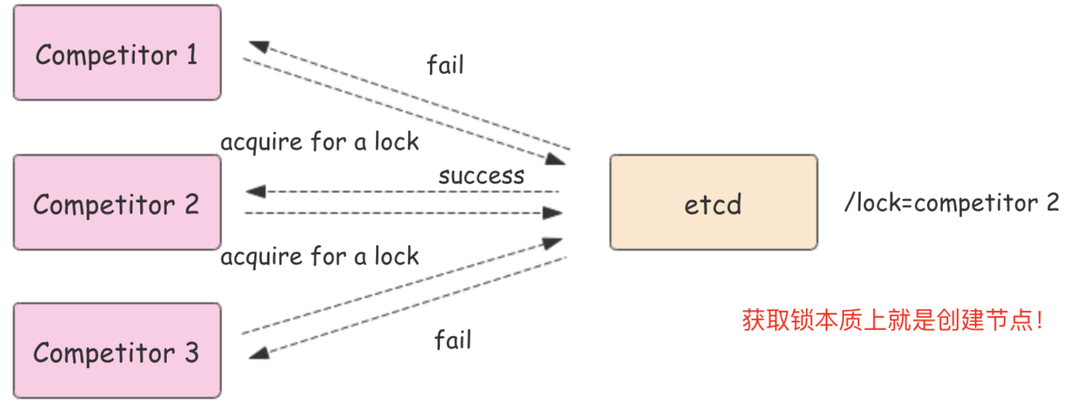
将一些配置信息放到 etcd 上进行集中管理。

这类场景的使用方式通常是这样：应用在启动的时候主动从 etcd 获取一次配置信息，同时，在 etcd 节点上注册一个 Watcher 并等待，以后每次配置有更新的时候，etcd 都会实时通知订阅者，以此达到获取最新配置信息的目的。

分布式锁

因为 etcd 使用 Raft 算法保持了数据的强一致性，某次操作存储到集群中的值必然是全局一致的，所以很容易实现分布式锁。锁服务有两种使用方式，一是保持独占，二是控制时序。

- 保持独占即所有获取锁的用户最终只有一个可以得到。etcd 为此提供了一套实现分布式锁原子操作 CAS（CompareAndSwap）的 API。通过设置prevExist值，可以保证在多个节点同时去创建某个目录时，只有一个成功。而创建成功的用户就可以认为是获得了锁。
- 控制时序，即所有想要获得锁的用户都会被安排执行，但是获得锁的顺序也是全局唯一的，同时决定了执行顺序。etcd 为此也提供了一套 API（自动创建有序键），对一个目录建值时指定为POST动作，这样 etcd 会自动在目录下生成一个当前最大的值为键，存储这个新的值（客户端编号）。同时还可以使用 API 按顺序列出所有当前目录下的键值。此时这些键的值就是客户端的时序，而这些键中存储的值可以是代表客户端的编号。



为什么用 etcd 而不用ZooKeeper？

etcd 实现的这些功能，ZooKeeper都能实现。那么为什么要用 etcd 而非直接使用 ZooKeeper呢？

为什么不选择ZooKeeper？

- 部署维护复杂，其使用的Paxos强一致性算法复杂难懂。官方只提供了Java和C两种语言的接口。
- 使用Java编写引入大量的依赖。运维人员维护起来比较麻烦。
- 最近几年发展缓慢，不如etcd和consul等后起之秀。

为什么选择etcd？

- 简单。使用 Go 语言编写部署简单；支持HTTP/JSON API,使用简单；使用 Raft 算法保证强一致性让用户易于理解。
- etcd 默认数据一更新就进行持久化。
- etcd 支持 SSL 客户端安全认证。

最后，etcd 作为一个年轻的项目，正在高速迭代和开发中，这既是一个优点，也是一个缺点。优点是它的未来具有无限的可能性，缺点是无法得到大项目长时间使用的检验。然而，目前 CoreOS、Kubernetes和CloudFoundry等知名项目均在生产环境中使用了etcd，所以总的来说，etcd值得你去尝试。

etcd集群

etcd 作为一个高可用键值存储系统，天生就是为集群化而设计的。由于 Raft 算法在做决策时需要多数节点的投票，所以 etcd 一般部署集群推荐奇数个节点，推荐的数量为 3、5 或者 7

个节点构成一个集群。

搭建一个3节点集群示例：

在每个etcd节点指定集群成员，为了区分不同的集群最好同时配置一个独一无二的token。

下面是提前定义好的集群信息，其中n1、n2和n3表示3个不同的etcd节点。

```
TOKEN=token-01
CLUSTER_STATE=new
CLUSTER=n1=http://10.240.0.17:2380,n2=http://10.240.0.18:2380,n3=http://10.240.0.19:2380
```

在n1这台机器上执行以下命令来启动etcd：

```
etcd --data-dir=data.etcd --name n1 \
    --initial-advertise-peer-urls http://10.240.0.17:2380 --listen-peer-urls http://10.240.0.17:2380 \
    --advertise-client-urls http://10.240.0.17:2379 --listen-client-urls http://10.240.0.17:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

在n2这台机器上执行以下命令启动etcd：

```
etcd --data-dir=data.etcd --name n2 \
    --initial-advertise-peer-urls http://10.240.0.18:2380 --listen-peer-urls http://10.240.0.18:2380 \
    --advertise-client-urls http://10.240.0.18:2379 --listen-client-urls http://10.240.0.18:2379 \
    --initial-cluster ${CLUSTER} \
    --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKEN}
```

在n3这台机器上执行以下命令启动etcd：

```
etcd --data-dir=data.etcd --name n3 \
    --initial-advertise-peer-urls http://10.240.0.19:2380 --listen-peer-urls http://10.240.0.19:2380 \
    --advertise-client-urls http://10.240.0.19:2379 --listen-client-urls http://10.240.0.19:2379 \
    --initial-cluster ${CLUSTER} \
```

```
        --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKE
N}
```

etcd 官网提供了一个可以公网访问的 etcd 存储地址。你可以通过如下命令得到 etcd 服务的目录，并把它作为-discovery参数使用。

```
    curl https://discovery.etcd.io/new?size=3
    https://discovery.etcd.io/a81b5818e67a6ea83e9d4daea5ecbc92

    # grab this token
    TOKEN=token-01
    CLUSTER_STATE=new
    DISCOVERY=https://discovery.etcd.io/a81b5818e67a6ea83e9d4daea5ecbc92


    etcd --data-dir=data.etcd --name n1 \
        --initial-advertise-peer-urls http://10.240.0.17:2380 --listen-peer-urls
http://10.240.0.17:2380 \
        --advertise-client-urls http://10.240.0.17:2379 --listen-client-urls htt
p://10.240.0.17:2379 \
        --discovery ${DISCOVERY} \
        --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKE
N}


    etcd --data-dir=data.etcd --name n2 \
        --initial-advertise-peer-urls http://10.240.0.18:2380 --listen-peer-urls
http://10.240.0.18:2380 \
        --advertise-client-urls http://10.240.0.18:2379 --listen-client-urls htt
p://10.240.0.18:2379 \
        --discovery ${DISCOVERY} \
        --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKE
N}


    etcd --data-dir=data.etcd --name n3 \
        --initial-advertise-peer-urls http://10.240.0.19:2380 --listen-peer-urls
http://10.240.0.19:2380 \
        --advertise-client-urls http://10.240.0.19:2379 --listen-client-urls htt
p:/10.240.0.19:2379 \
        --discovery ${DISCOVERY} \
        --initial-cluster-state ${CLUSTER_STATE} --initial-cluster-token ${TOKE
N}
```

到此etcd集群就搭建起来了，可以使用etcdctl来连接etcd。

```
export ETCDCTL_API=3
HOST_1=10.240.0.17
HOST_2=10.240.0.18
HOST_3=10.240.0.19
ENDPOINTS=$HOST_1:2379,$HOST_2:2379,$HOST_3:2379

etcdctl --endpoints=$ENDPOINTS member list
```

操作ETCD

这里使用官方的 `etcd/clientv3` 包来连接etcd并进行相关操作。

安装

```
go get go.etcd.io/etcd/clientv3
```

put和get操作

put命令用来设置键值对数据，get命令用来根据key获取值。

```
package main

import (
	"context"
	"fmt"
	"time"

	"go.etcd.io/etcd/clientv3"
)

// etcd client put/get demo
// use etcd/clientv3

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"127.0.0.1:2379"},
		DialTimeout: 5 * time.Second,
	})
	if err != nil {
		// handle error!
		fmt.Printf("connect to etcd failed, err:%v\n", err)
		return
	}
	fmt.Println("connect to etcd success")
	defer cli.Close()
	// put
	ctx, cancel := context.WithTimeout(context.Background(), time.Second)
	_, err = cli.Put(ctx, "lmh", "lmh")
	cancel()
	if err != nil {
		fmt.Printf("put to etcd failed, err:%v\n", err)
```

```
        return
    }
    // get
    ctx, cancel = context.WithTimeout(context.Background(), time.Second)
    resp, err := cli.Get(ctx, "lmh")
    cancel()
    if err != nil {
        fmt.Printf("get from etcd failed, err:%v\n", err)
        return
    }
    for _, ev := range resp.Kvs {
        fmt.Printf("%s:%s\n", ev.Key, ev.Value)
    }
}
```

watch操作

watch用来获取未来更改的通知。

```
package main

import (
    "context"
    "fmt"
    "time"

    "go.etcd.io/etcd/clientv3"
)

// watch demo

func main() {
    cli, err := clientv3.New(clientv3.Config{
        Endpoints:   []string{"127.0.0.1:2379"},
        DialTimeout: 5 * time.Second,
    })
    if err != nil {
        fmt.Printf("connect to etcd failed, err:%v\n", err)
        return
    }
    fmt.Println("connect to etcd success")
    defer cli.Close()
    // watch key:lmh change
    rch := cli.Watch(context.Background(), "lmh") // <-chan WatchResponse
    for wresp := range rch {
```

```
        for _, ev := range wresp.Events {
            fmt.Printf("Type: %s Key:%s Value:%s\n", ev.Type, ev.Kv.Key, ev.Kv.V
alue)
        }
    }
}
```

将上面的代码保存编译执行，此时程序就会等待etcd中lmh这个key的变化。

例如：我们打开终端执行以下命令修改、删除、设置lmh这个key。

```
    etcd> etcdctl.exe --endpoints=http://127.0.0.1:2379 put lmh "lmh1"
    OK

    etcd> etcdctl.exe --endpoints=http://127.0.0.1:2379 del lmh
    1

    etcd> etcdctl.exe --endpoints=http://127.0.0.1:2379 put lmh "lmh2"
    OK
```

上面的程序都能收到如下通知。

```
    watch>watch.exe
    connect to etcd success
    Type: PUT Key:lmh Value:lmh1
    Type: DELETE Key:lmh Value:
    Type: PUT Key:lmh Value:lmh2
```

lease租约

```
package main

import (
    "fmt"
    "time"
)

// etcd lease

import (
    "context"
    "log"

    "go.etcd.io/etcd/clientv3"
```

```
)

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"127.0.0.1:2379"},
		DialTimeout: time.Second * 5,
	})
	if err != nil {
		log.Fatal(err)
	}
	fmt.Println("connect to etcd success.")
	defer cli.Close()

	// 创建一个5秒的租约
	resp, err := cli.Grant(context.TODO(), 5)
	if err != nil {
		log.Fatal(err)
	}

	// 5秒钟之后, /lmh/ 这个key就会被移除
	_, err = cli.Put(context.TODO(), "/lmh/", "lmh", clientv3.WithLease(resp.I
D))
	if err != nil {
		log.Fatal(err)
	}
}
```

keepAlive

```
package main

import (
	"context"
	"fmt"
	"log"
	"time"

	"go.etcd.io/etcd/clientv3"
)

// etcd keepAlive

func main() {
	cli, err := clientv3.New(clientv3.Config{
		Endpoints:   []string{"127.0.0.1:2379"},
```

```
        DialTimeout: time.Second * 5,
    })
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("connect to etcd success.")
    defer cli.Close()

    resp, err := cli.Grant(context.TODO(), 5)
    if err != nil {
        log.Fatal(err)
    }

    _, err = cli.Put(context.TODO(), "/lmh/", "lmh", clientv3.WithLease(resp.I
D))
    if err != nil {
        log.Fatal(err)
    }

    // the key 'foo' will be kept forever
    ch, kaerr := cli.KeepAlive(context.TODO(), resp.ID)
    if kaerr != nil {
        log.Fatal(kaerr)
    }
    for {
        ka := <-ch
        fmt.Println("ttl:", ka.TTL)
    }
}
```

基于etcd实现分布式锁

go.etcd.io/etcd/clientv3/concurrency在etcd之上实现并发操作，如分布式锁、屏障和选举。

导入该包：

```
    import "go.etcd.io/etcd/clientv3/concurrency"
```

基于etcd实现的分布式锁示例：

```
cli, err := clientv3.New(clientv3.Config{Endpoints: endpoints})
if err != nil {
    log.Fatal(err)
```

```
}
defer cli.Close()

// 创建两个单独的会话用来演示锁竞争
s1, err := concurrency.NewSession(cli)
if err != nil {
    log.Fatal(err)
}
defer s1.Close()
m1 := concurrency.NewMutex(s1, "/my-lock/")

s2, err := concurrency.NewSession(cli)
if err != nil {
    log.Fatal(err)
}
defer s2.Close()
m2 := concurrency.NewMutex(s2, "/my-lock/")

// 会话s1获取锁
if err := m1.Lock(context.TODO()); err != nil {
    log.Fatal(err)
}
fmt.Println("acquired lock for s1")

m2Locked := make(chan struct{})
go func() {
    defer close(m2Locked)
    // 等待直到会话s1释放了/my-lock/的锁
    if err := m2.Lock(context.TODO()); err != nil {
        log.Fatal(err)
    }
}()

if err := m1.Unlock(context.TODO()); err != nil {
    log.Fatal(err)
}
fmt.Println("released lock for s1")

<-m2Locked
fmt.Println("acquired lock for s2")
```

输出:

```
    acquired lock for s1
    released lock for s1
```

```
    acquired lock for s2
```

查看文档了解更多

zookeeper

基本操作测试

简单的分布式server

Zookeeper命令行使用

基本操作测试

简单的例子来测试下基本的操作：

```
package main

/**
客户端doc地址：github.com/samuel/go-zookeeper/zk
**/
import (
    "fmt"
    "time"

    zk "github.com/samuel/go-zookeeper/zk"
)

/**
 * 获取一个zk连接
 * @return {[type]}
 */
func getConnect(zkList []string) (conn *zk.Conn) {
    conn, _, err := zk.Connect(zkList, 10*time.Second)
    if err != nil {
        fmt.Println(err)
    }
    return
}

/**
 * 测试连接
 * @return
 */
func test1() {
    zkList := []string{"localhost:2181"}
    conn := getConnect(zkList)

    defer conn.Close()
    var flags int32 = 0
    //flags有4种取值：
    //0:永久，除非手动删除
    //zk.FlagEphemeral  = 1:短暂，session断开则改节点也被删除
    //zk.FlagSequence   = 2:会自动在节点后面添加序号
    //3:Ephemeral和Sequence，即，短暂且自动添加序号
    conn.Create("/go_servers", nil, flags, zk.WorldACL(zk.PermAll)) // zk.WorldA
```

```
CL(zk.PermAll)控制访问权限模式

    time.Sleep(20 * time.Second)
}

/*
删改与增不同在于其函数中的version参数,其中version是用于 CAS支持
func (c *Conn) Set(path string, data []byte, version int32) (*Stat, error)
func (c *Conn) Delete(path string, version int32) error

demo:
if err = conn.Delete(migrateLockPath, -1); err != nil {
    log.Error("conn.Delete(\"%s\") error(%v)", migrateLockPath, err)
}
*/

/**
 * 测试临时节点
 * @return {[type]}
 */
func test2() {
    zkList := []string{"localhost:2181"}
    conn := getConnect(zkList)

    defer conn.Close()
    conn.Create("/testadaadsasdsaw", nil, zk.FlagEphemeral, zk.WorldACL(zk.PermA
ll))

    time.Sleep(20 * time.Second)
}

/**
 * 获取所有节点
 */
func test3() {
    zkList := []string{"localhost:2181"}
    conn := getConnect(zkList)

    defer conn.Close()

    children, _, err := conn.Children("/go_servers")
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%v \n", children)
}
```

```
func main() {
    test3()
}
```

简单的分布式server

简单的分布式server

目前分布式系统已经很流行了，一些开源框架也被广泛应用，如dubbo、Motan等。对于一个分布式服务，最基本的一项功能就是服务的注册和发现，而利用zk的EPHEMERAL节点则可以很方便的实现该功能。EPHEMERAL节点正如其名，是临时性的，其生命周期是和客户端会话绑定的，当会话连接断开时，节点也会被删除。下边我们就来实现一个简单的分布式server：

server：

服务启动时，创建zk连接，并在go_servers节点下创建一个新节点，节点名为"ip:port"，完成服务注册
服务结束时，由于连接断开，创建的节点会被删除，这样client就不会连到该节点

client：

先从zk获取go_servers节点下所有子节点，这样就拿到了所有注册的server
从server列表中选中一个节点（这里只是随机选取，实际服务一般会提供多种策略），创建连接进行通信
这里为了演示，我们每次client连接server，获取server发送的时间后就断开。主要代码如下：

server.go

```
package main

import (
    "fmt"
    "net"
    "os"
    "time"

    "github.com/samuel/go-zookeeper/zk"
)

func main() {
    go starServer("127.0.0.1:8897")
    go starServer("127.0.0.1:8898")
    go starServer("127.0.0.1:8899")

    a := make(chan bool, 1)
    <-a
```

```
}

func checkError(err error) {
	if err != nil {
		fmt.Println(err)
	}
}

func starServer(port string) {
	tcpAddr, err := net.ResolveTCPAddr("tcp4", port)
	fmt.Println(tcpAddr)
	checkError(err)

	listener, err := net.ListenTCP("tcp", tcpAddr)
	checkError(err)

	//注册zk节点q
	// 链接zk
	conn, err := GetConnect()
	if err != nil {
		fmt.Printf(" connect zk error: %s ", err)
	}
	defer conn.Close()
	// zk节点注册
	err = RegistServer(conn, port)
	if err != nil {
		fmt.Printf(" regist node error: %s ", err)
	}

	for {
		conn, err := listener.Accept()
		if err != nil {
			fmt.Fprintf(os.Stderr, "Error: %s", err)
			continue
		}
		go handleCient(conn, port)
	}

	fmt.Println("aaaaaa")
}

func handleCient(conn net.Conn, port string) {
	defer conn.Close()

	daytime := time.Now().String()
	conn.Write([]byte(port + ": " + daytime))
```

```
}
func GetConnect() (conn *zk.Conn, err error) {
    zkList := []string{"localhost:2181"}
    conn, _, err = zk.Connect(zkList, 10*time.Second)
    if err != nil {
        fmt.Println(err)
    }
    return
}

func RegistServer(conn *zk.Conn, host string) (err error) {
    _, err = conn.Create("/go_servers/"+host, nil, zk.FlagEphemeral, zk.WorldACL
(zk.PermAll))
    return
}

func GetServerList(conn *zk.Conn) (list []string, err error) {
    list, _, err = conn.Children("/go_servers")
    return
}
```

client.go

```
package main

import (
    "errors"
    "fmt"
    "io/ioutil"
    "math/rand"
    "net"
    "time"

    "github.com/samuel/go-zookeeper/zk"
)

func checkError(err error) {
    if err != nil {
        fmt.Println(err)
    }
}
func main() {
    for i := 0; i < 100; i++ {
        startClient()
```

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

func startClient() {
    // service := "127.0.0.1:8899"
    //获取地址
    serverHost, err := getServerHost()
    if err != nil {
        fmt.Printf("get server host fail: %s \n", err)
        return
    }

    fmt.Println("connect host: " + serverHost)
    tcpAddr, err := net.ResolveTCPAddr("tcp4", serverHost)
    checkError(err)
    conn, err := net.DialTCP("tcp", nil, tcpAddr)
    checkError(err)
    defer conn.Close()

    _, err = conn.Write([]byte("timestamp"))
    checkError(err)

    result, err := ioutil.ReadAll(conn)
    checkError(err)
    fmt.Println(string(result))

    return
}

func getServerHost() (host string, err error) {
    conn, err := GetConnect()
    if err != nil {
        fmt.Printf(" connect zk error: %s \n ", err)
        return
    }
    defer conn.Close()
    serverList, err := GetServerList(conn)
    if err != nil {
        fmt.Printf(" get server list error: %s \n", err)
        return
    }

    count := len(serverList)
    if count == 0 {
        err = errors.New("server list is empty \n")
```

```
        return
    }

    //随机选中一个返回
    r := rand.New(rand.NewSource(time.Now().UnixNano()))
    host = serverList[r.Intn(3)]
    return
}
func GetConnect() (conn *zk.Conn, err error) {
    zkList := []string{"localhost:2181"}
    conn, _, err = zk.Connect(zkList, 10*time.Second)
    if err != nil {
        fmt.Println(err)
    }
    return
}
func GetServerList(conn *zk.Conn) (list []string, err error) {
    list, _, err = conn.Children("/go_servers")
    return
}
```

先启动server，可以看到有三个节点注册到zk：

```
    127.0.0.1:8897
    127.0.0.1:8899
    127.0.0.1:8898
    2018/08/27 14:04:58 Connected to 127.0.0.1:2181
    2018/08/27 14:04:58 Connected to 127.0.0.1:2181
    2018/08/27 14:04:58 Connected to 127.0.0.1:2181
    2018/08/27 14:04:58 Authenticated: id=100619932030205976, timeout=10000
    2018/08/27 14:04:58 Re-submitting `0` credentials after reconnect
    2018/08/27 14:04:58 Authenticated: id=100619932030205977, timeout=10000
    2018/08/27 14:04:58 Re-submitting `0` credentials after reconnect
    2018/08/27 14:04:58 Authenticated: id=100619932030205978, timeout=10000
    2018/08/27 14:04:58 Re-submitting `0` credentials after reconnect
```

启动client，可以看到每次client都会随机连接到一个节点进行通信：

```
    2018/08/27 14:05:21 Connected to 127.0.0.1:2181
    2018/08/27 14:05:21 Authenticated: id=100619932030205979, timeout=10000
    2018/08/27 14:05:21 Re-submitting `0` credentials after reconnect
    2018/08/27 14:05:21 Recv loop terminated: err=EOF
    connect host: 127.0.0.1:8899
    2018/08/27 14:05:21 Send loop terminated: err=<nil>
    read tcp 127.0.0.1:54062->127.0.0.1:8899: read: connection reset by peer
```

```
127.0.0.1:8899: 2018-08-27 14:05:21.291641 +0800 CST m=+22.480149656
2018/08/27 14:05:22 Connected to [::1]:2181
2018/08/27 14:05:22 Authenticated: id=100619932030205980, timeout=10000
2018/08/27 14:05:22 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:22 Recv loop terminated: err=EOF
2018/08/27 14:05:22 Send loop terminated: err=<nil>
connect host: 127.0.0.1:8897
read tcp 127.0.0.1:54064->127.0.0.1:8897: read: connection reset by peer
127.0.0.1:8897: 2018-08-27 14:05:22.302322 +0800 CST m=+23.490801385
2018/08/27 14:05:23 Connected to 127.0.0.1:2181
2018/08/27 14:05:23 Authenticated: id=100619932030205981, timeout=10000
2018/08/27 14:05:23 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:23 Recv loop terminated: err=EOF
2018/08/27 14:05:23 Send loop terminated: err=<nil>
connect host: 127.0.0.1:8897
read tcp 127.0.0.1:54070->127.0.0.1:8897: read: connection reset by peer
127.0.0.1:8897: 2018-08-27 14:05:23.312873 +0800 CST m=+24.501324228
2018/08/27 14:05:24 Connected to 127.0.0.1:2181
2018/08/27 14:05:24 Authenticated: id=100619932030205982, timeout=10000
2018/08/27 14:05:24 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:24 Recv loop terminated: err=EOF
connect host: 127.0.0.1:8899
2018/08/27 14:05:24 Send loop terminated: err=<nil>
read tcp 127.0.0.1:54072->127.0.0.1:8899: read: connection reset by peer
127.0.0.1:8899: 2018-08-27 14:05:24.323668 +0800 CST m=+25.512090155
2018/08/27 14:05:25 Connected to 127.0.0.1:2181
2018/08/27 14:05:25 Authenticated: id=100619932030205983, timeout=10000
2018/08/27 14:05:25 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:25 Recv loop terminated: err=EOF
2018/08/27 14:05:25 Send loop terminated: err=<nil>
connect host: 127.0.0.1:8897
read tcp 127.0.0.1:54074->127.0.0.1:8897: read: connection reset by peer
127.0.0.1:8897: 2018-08-27 14:05:25.330257 +0800 CST m=+26.518650566
2018/08/27 14:05:26 Connected to [::1]:2181
2018/08/27 14:05:26 Authenticated: id=100619932030205984, timeout=10000
2018/08/27 14:05:26 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:26 Recv loop terminated: err=EOF
2018/08/27 14:05:26 Send loop terminated: err=<nil>
connect host: 127.0.0.1:8897
read tcp 127.0.0.1:54080->127.0.0.1:8897: read: connection reset by peer
127.0.0.1:8897: 2018-08-27 14:05:26.357251 +0800 CST m=+27.545614616
2018/08/27 14:05:27 Connected to 127.0.0.1:2181
2018/08/27 14:05:27 Authenticated: id=100619932030205985, timeout=10000
2018/08/27 14:05:27 Re-submitting `0` credentials after reconnect
connect host: 127.0.0.1:8899
2018/08/27 14:05:27 Recv loop terminated: err=EOF
```

```
2018/08/27 14:05:27 Send loop terminated: err=<nil>
read tcp 127.0.0.1:54082->127.0.0.1:8899: read: connection reset by peer
127.0.0.1:8899: 2018-08-27 14:05:27.369096 +0800 CST m=+28.557430764
2018/08/27 14:05:28 Connected to [::1]:2181
2018/08/27 14:05:28 Authenticated: id=100619932030205986, timeout=10000
2018/08/27 14:05:28 Re-submitting `0` credentials after reconnect
2018/08/27 14:05:28 Recv loop terminated: err=EOF
2018/08/27 14:05:28 Send loop terminated: err=<nil>
connect host: 127.0.0.1:8898
read tcp 127.0.0.1:54084->127.0.0.1:8898: read: connection reset by peer
127.0.0.1:8898: 2018-08-27 14:05:28.380455 +0800 CST m=+29.568760988
......
```

至此，我们的分布式server就实现了

Zookeeper命令行使用

Zookeeper部署

Zookeeper的部署相对来说还是比较简单，读者可以在网上找到相应的教程。Zookeeper有三种运行形式：集群模式、单机模式、伪集群模式。

以下实验都是在单机模式下进行。

服务端使用

zookeeper下bin目录下常用的脚本解释：

1.zkCleanup 清理Zookeeper历史数据，包括事物日志文件和快照数据文件

2.zkCli Zookeeper的一个简易客户端

3.zkEnv 设置Zookeeper的环境变量

4.zkServer Zookeeper服务器的启动、停止、和重启脚本

1.运行服务

进入bin目录，使用zkServer.sh start启动服务

使用jps命令查看，存在QuorumPeerMain进程，表示Zookeeper已经启动

2.停止服务

在bin目录下，使用zkServer.sh stop停止服务

使用jps命令查看，QuorumPeerMain进程已不存在，表示Zookeeper已经关闭

客户端使用

1.打开客户端

在服务端开启的情况下，运行客户端，使用如下命令：./zkCli.sh

连接服务端成功。若连接不同的主机，可使用如下命令：./zkCli.sh -server ip:port（当然可以使用配置文件）。

可以使用帮助命令help来查看客户端的操作

2.创建节点

使用create命令，可以创建一个Zookeeper节点，如下：

create [-s] [-e] path data acl

其中，-s或-e分别指定节点特性，顺序或临时节点，若不指定，则表示持久节点；acl用来进行权限控制。

① 创建顺序节点

使用 create -s /zk-test 123 命令创建zk-test顺序节点

可以看到创建的zk-test节点后面添加了一串数字以示区别。

② 创建临时节点

使用 create -e /zk-temp 123 命令创建zk-temp临时节点

临时节点在客户端会话结束后，就会自动删除，下面使用quit命令退出客户端

再次使用客户端连接服务端，并使用ls / 命令查看根目录下的节点

可以看到根目录下已经不存在zk-temp临时节点了。

③ 创建永久节点

使用 create /zk-permanent 123 命令创建zk-permanent永久节点

可以看到永久节点不同于顺序节点，不会自动在后面添加一串数字。

3.读取节点

与读取相关的命令有ls 命令和get 命令，ls命令可以列出Zookeeper指定节点下的所有子节点，只能查看指定节点下的第一级的所有子节点；get命令可以获取Zookeeper指定节点的数据内容和属性信息。其用法分别如下

ls path [watch]

get path [watch]

ls2 path [watch]

若获取根节点下面的所有子节点，使用ls / 命令即可

若想获取根节点数据内容和属性信息，使用get / 命令即可

也可以使用ls2 / 命令查看

可以看到其子节点数量为8。

若想获取/zk-permanent的数据内容和属性，可使用如下命令：get /zk-permanent

可以看到其数据内容为123，还有其他的属性，之后会详细介绍。

4.更新节点

使用set命令，可以更新指定节点的数据内容，用法如下

set path data [version]

其中，data就是要更新的新内容，version表示数据版本，如将/zk-permanent节点的数据更新为456，可以使用如下命令：set /zk-permanent 456

现在dataVersion已经变为1了，表示进行了更新。

5.删除节点

使用delete命令可以删除Zookeeper上的指定节点，用法如下

delete path [version]

其中version也是表示数据版本，使用delete /zk-permanent 命令即可删除/zk-permanent节点

可以看到，已经成功删除/zk-permanent节点。值得注意的是，若删除节点存在子节点，那么无法删除该节点，必须先删除子节点，再删除父节点。

go操作kafka

Kafka介绍

Kafka深层介绍

Kafka的安装

操作Kafka

Kafka介绍

Kafka是什么

```
    kafka使用scala开发，支持多语言客户端（c++、java、python、go等）
    Kafka最先由LinkedIn公司开发，之后成为Apache的顶级项目。
    Kafka是一个分布式的、分区化、可复制提交的日志服务
    LinkedIn使用Kafka实现了公司不同应用程序之间的松耦和，那么作为一个可扩展、高
可靠的消息系统
    支持高Throughput的应用
    scale out：无需停机即可扩展机器
    持久化：通过将数据持久化到硬盘以及replication防止数据丢失
    支持online和offline的场景
```

Kafka的特点

Kafka是分布式的，其所有的构件borker(服务端集群)、producer(消息生产)、consumer(消息消费者)都可以是分布式的。

在消息的生产时可以使用一个标识topic来区分，且可以进行分区；每一个分区都是一个顺序的、不可变的消息队列， 并且可以持续的添加。

同时为发布和订阅提供高吞吐量。据了解，Kafka每秒可以生产约25万消息（50 MB），每秒处理55万消息（110 MB）。

消息被处理的状态是在consumer端维护，而不是由server端维护。当失败时能自动平衡

常用的场景

监控：主机通过Kafka发送与系统和应用程序健康相关的指标，然后这些信息会被收集和处理从而创建监控仪表盘并发送警告。

消息队列： 应用程度使用Kafka作为传统的消息系统实现标准的队列和消息的发布—订阅，例如搜索和内容提要（Content Feed）。比起大多数的消息系统来说，Kafka有更好的吞吐量，内置的分区，冗余及容错性，这让Kafka成为了一个很好的大规模消息处理应用的解决方案。消息系统 一般吞吐量相对较低，但是需要更小的端到端延时，并尝尝依赖于Kafka提供的强大的持久性保障。在这个领域，Kafka足以媲美传统消息系统，如ActiveMR或RabbitMQ

站点的用户活动追踪: 为了更好地理解用户行为，改善用户体验，将用户查看了哪个页面、点击了哪些内容等信息发送到每个数据中心的Kafka集群上，并通过Hadoop进行分析、生成日常报告。

流处理：保存收集流数据，以提供之后对接的Storm或其他流式计算框架进行处理。很多用户会将那些从原始topic来的数据进行 阶段性处理，汇总，扩充或者以其他的方式转换到新的topic下再继续后面的处理。例如一个文章推荐的处理流程，可能是先从RSS数据源中抓取文章的内 容，然后将其丢入一个叫做“文章”的topic中；后续操作可能是需要对这个内容进行清理，比如回复正常数据或者删除重复数据，最后再将内容匹配的结果返 还给用户。这就在一个独立的topic之外，产生了一系列的实时数据处理的流程。

日志聚合:使用Kafka代替日志聚合（log aggregation）。日志聚合一般来说是从服务器上收集日志文件，然后放到一个集中的位置（文件服务器或HDFS）进行处理。然而Kafka忽略掉 文件的细节，将其更清晰地抽象成一个个日志或事件的消息流。这就让Kafka处理过程延迟更低，更容易支持多数据源和分布式数据处理。比起以日志为中心的 系统比如Scribe或者Flume来说，Kafka提供同样高效的性能和因为复制导致的更高的耐用性保证，以及更低的端到端延迟

持久性日志：Kafka可以为一种外部的持久性日志的分布式系统提供服务。这种日志可以在节点间备份数据，并为故障节点数据回复提供一种重新同步的机制。Kafka中日志压缩功能为这种用法提供了条件。在这种用法中，Kafka类似于Apache BookKeeper项目。

Kafka中包含以下基础概念

```
    1.Topic(话题)：Kafka中用于区分不同类别信息的类别名称。由producer指定
    2.Producer(生产者)：将消息发布到Kafka特定的Topic的对象(过程)
    3.Consumers(消费者)：订阅并处理特定的Topic中的消息的对象(过程)
    4.Broker(Kafka服务集群)：已发布的消息保存在一组服务器中，称之为Kafka集群。集群中的每一个服务器都是一个代理(Broker). 消费者可以订阅一个或多个话题，并从Broker拉数据，从而消费这些已发布的消息。
    5.Partition(分区)：Topic物理上的分组，一个topic可以分为多个partition，每个partition是一个有序的队列。partition中的每条消息都会被分配一个有序的id（offset）
    Message：消息，是通信的基本单位，每个producer可以向一个topic（主题）发布一些消息。
```

消息

消息由一个固定大小的报头和可变长度但不透明的字节阵列负载。报头包含格式版本和CRC32效验和以检测损坏或截断

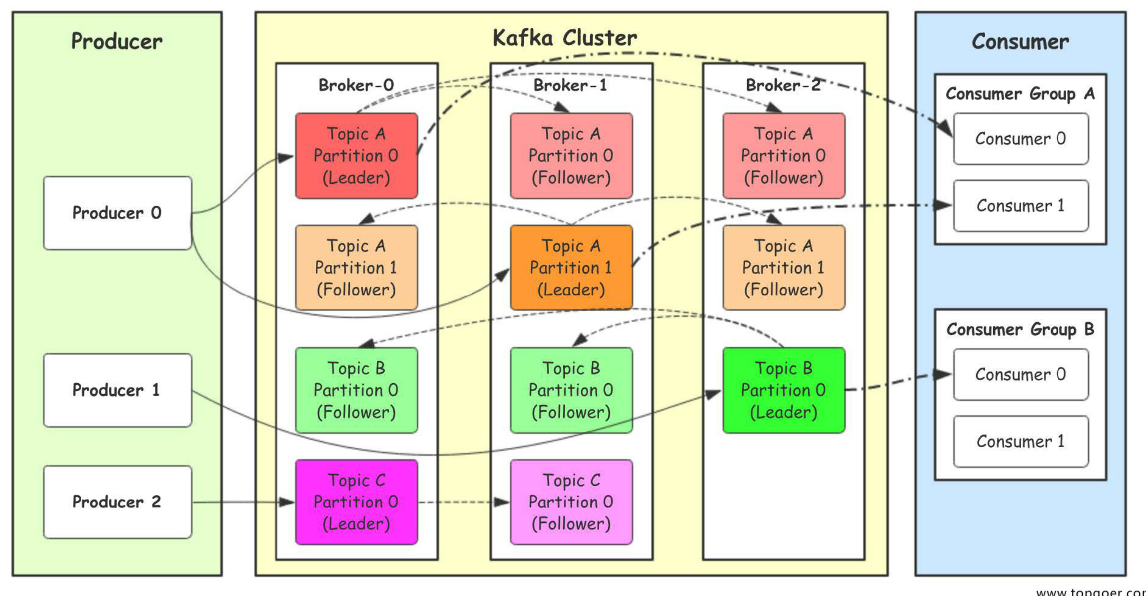
消息格式

```
    1. 4 byte CRC32 of the message
    2. 1 byte "magic" identifier to allow format changes, value is 0 or 1
    3. 1 byte "attributes" identifier to allow annotations on the message independent of the version
       bit 0 ~ 2 : Compression codec
           0 : no compression
```

```
            1 : gzip
            2 : snappy
            3 : lz4
        bit 3 : Timestamp type
            0 : create time
            1 : log append time
        bit 4 ~ 7 : reserved
    4. (可选) 8 byte timestamp only if "magic" identifier is greater than 0
    5. 4 byte key length, containing length K
    6. K byte key
    7. 4 byte payload length, containing length V
    8. V byte payload
```

Kafka深层介绍

架构介绍

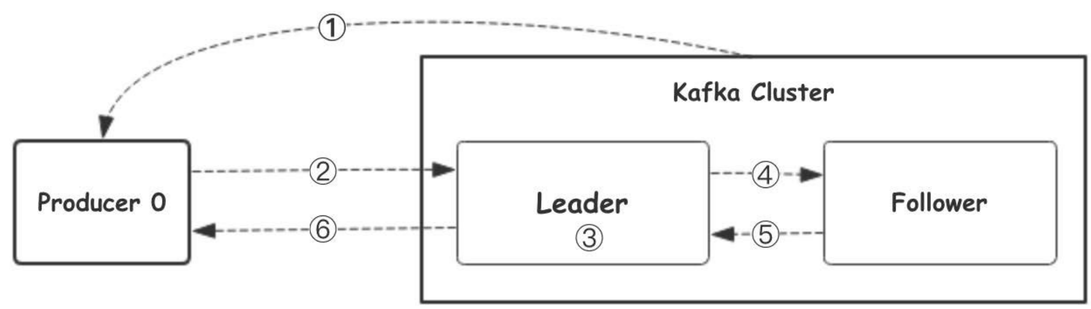


- Producer：Producer即生产者，消息的产生者，是消息的□口。
- kafka cluster：kafka集群，一台或多台服务□组成
  - Broker：Broker是指部署□Kafka实□的服务□节点。每个服务□上有一个或多个kafka 的实 □□我们姑且认为每个broker对应一台服务□□每个kafka集群内的broker都有一个□重复的 编号，如图中的broker-0、broker-1等……
  - Topic：消息的主题，可以理解为消息的分类，kafka的数据就保存在topic。在每个broker上 都可以创建多个topic。实际应用中通常是一个业务线建一个topic。
  - Partition：Topic的分区，每个topic可以有多个分区，分区的作用是做负载，提高kafka的吞 吐□□同一个topic在□同的分区的数据是□重复的，partition的表现形式就是一个一个的□件夹！
  - Replication:每一个分区都有多个副本，副本的作用是做备胎。当主分区（Leader）故障的 时候会选择一个备胎（Follower）上位，成为Leader。在kafka中默认副本的最大数□是10 个，且副本的数□□能大于Broker的数□□follower和leader绝对是在□同的机器，同一机
    □对同一个分区也只可能存放一个副本（包括自己）。
- Consumer：消费者，即消息的消费方，是消息的出口。

- Consumer Group：我们可以将多个消费组组成一个消费者组，在kafka的设计中同一个分 区的数据只能被消费者组中的某一个消费者消费。同一个消费者组的消费者可以消费同一个 topic的不同分区的数据，这也是为提高kafka的吞吐

作流程

我们看上的架构图中，producer就是生产者，是数据的入口。Producer在写入数据的时候会把数据 写入到leader中，会直接将数据写入follower！那leader怎么找呢？写入的流程又是么样的呢？我 们看下图：




```
    1.   产者从Kafka集群获取分区leader信息
    2.   产者将消息发送给leader
    3.leader将消息写入本地磁盘
    4.follower从leader拉取消息数据
    5.follower将消息写入本地磁盘后向leader发送ACK
    6.leader收到所有的follower的ACK之后向生产者发送ACK
```

选择partition的原则

那在kafka中，如果某个topic有多个partition，producer怎么知道该将数据发往哪个partition呢？ kafka中有几个原则：

```
    1.partition在写入的时候可以指定需要写入的partition，如果有指定，则写入对应的partition。
    2.如果没有指定partition，但是设置了数据的key，则会根据key的值hash出一个partition。
    3.如果既没指定partition，又没有设置key，则会采用轮询  式，即每次取一小段时间的数据写入某
    个partition，下一小段的时间写入下一个partition
```

ACK应答机制

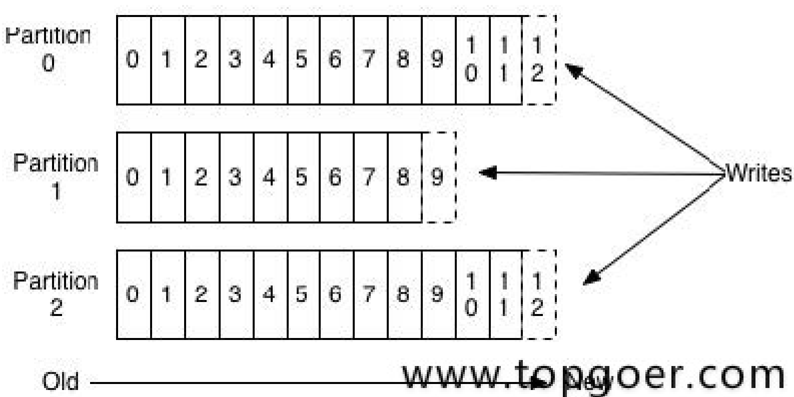
producer在向kafka写入消息的时候，可以设置参数来确定是否确认kafka接收到数据，这个参数可设置 的值为 0,1,all

- 0代表producer往集群发送数据需要等到集群的返回，确保消息发送成功。安全性最低但是效 率最高。
- 1代表producer往集群发送数据只要leader应答就可以发送下一条，只确保leader发送成功。
- all代表producer往集群发送数据需要所有的follower都完成从leader的同步才会发送下一条，确保 leader发送成功和所有的副本都完成备份。安全性最高，但是效率最低。

最后要注意的是，如果往不存在的topic写数据，kafka会动创建topic，partition和replication的数量
默认配置都是1。

Topic和数据志

topic 是同类别的消息记录（record）的集合。在Kafka中，个主题通常有多个订阅者。对于每个
主题，Kafka集群维护了个分区数据志件结构如下：



每个partition都是个有序并且不可变的消息记录集合。当新的数据写时，就被追加到partition的末
尾。在每个partition中，每条消息都会被分配个顺序的唯标识，这个标识被称为offset，即偏移
量。注意，Kafka只保证在同个partition内部消息是有序的，在不同partition之间，并不能保证消息
有序。

Kafka可以配置一个保留期限，用来标识日志会在Kafka集群内保留多长时间。Kafka集群会保留在保留
期限内所有被发布的消息，不管这些消息是否被消费过。比如保留期限设置为两天，那么数据被发布到
Kafka集群的两天以内，所有的这些数据都可以被消费。当超过两天，这些数据将会被清空，以便为后
续的数据腾出空间。由于Kafka会将数据进行持久化存储（即写入到硬盘上），所以保留的数据大小可
以设置为一个比较大的值。

Partition结构

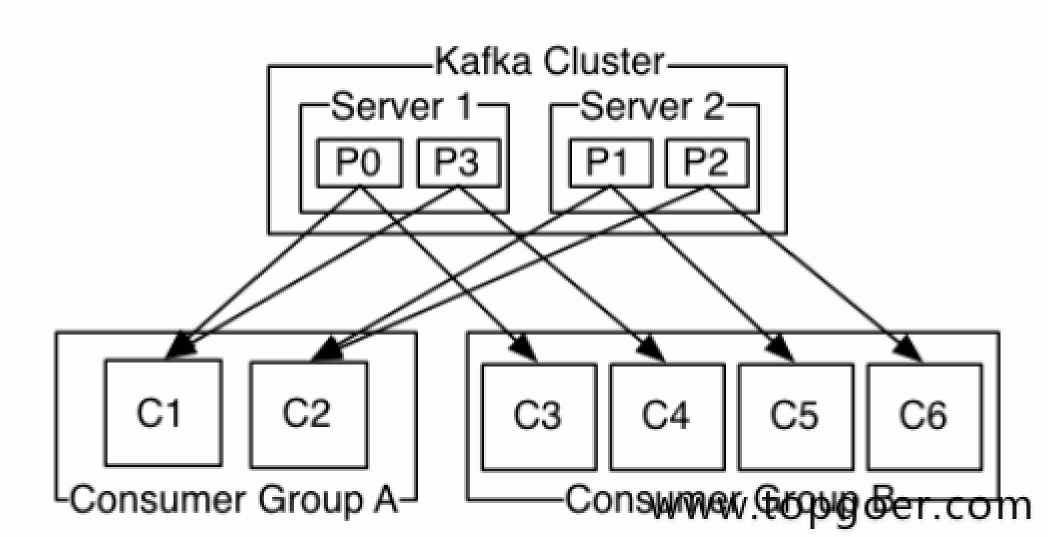
Partition在服务器上的表现形式就是一个一个的文件夹，每个partition的文件夹下面会有多组segment
文件，每组segment文件又包含 .index 文件、 .log 文件、 .timeindex 文件三个文件，其中 .log 文
件就是实际存储message的地方，而 .index 和 .timeindex 文件为索引文件，用于检索消息。

消费数据

多个消费者实例可以组成一个消费者组，并用一个标签来标识这个消费者组。一个消费者组中的不同消
费者实例可以运行在不同的进程甚至不同的服务器上。

如果所有的消费者实例都在同一个消费者组中，那么消息记录会被很好的均衡的发送到每个消费者实
例。

如果所有的消费者实例都在不同的消费者组，那么每一条消息记录会被广播到每一个消费者实例。



举个例□□如上图所示□个两个节点的Kafka集群上拥有□个四个partition（P0-P3）的topic。有两个
消费者组都在消费这个topic中的数据，消费者组A有两个消费者实例，消费者组B有四个消费者实例。
从图中我们可以看到，在同□个消费者组中，每个消费者实例可以消费多个分区，但是每个分区最多只
能被消费者组中的□个实例消费。也就是说，如果有□个4个分区的主题，那么消费者组中最多只能有4
个消费者实例去消费，多出来的都不会被分配到分区。其实这也很好理解，如果允许两个消费者实例同
时消费同□个分区，那么就□法记录这个分区被这个消费者组消费的offset了。如果在消费者组中动态
的上线或下线消费者，那么Kafka集群会□动调整分区与消费者实例间的对应关系。

Kafka的安装

kafka搭建

注意:我的电脑是win10的以下都是基于win10的安装

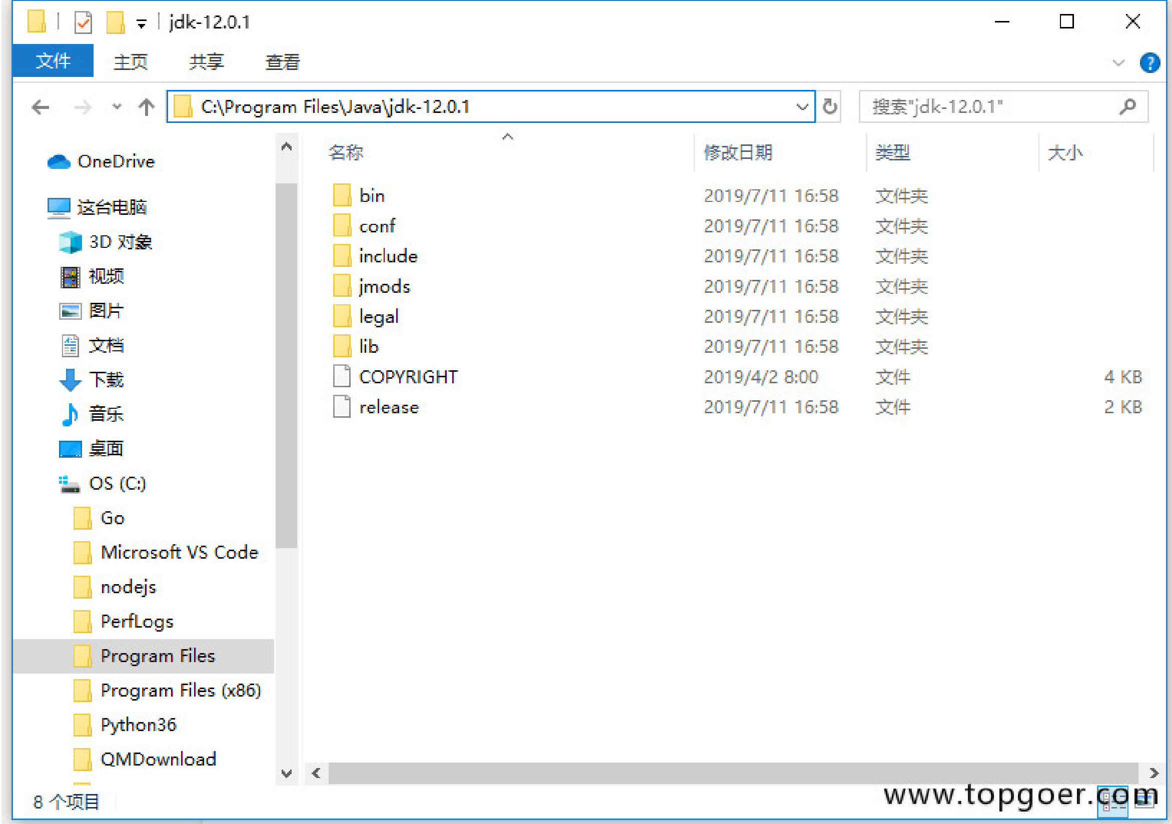
kafka环境基于zookeeper,zookeeper环境基于JAVA-JDK。

安装JAVA-JDK

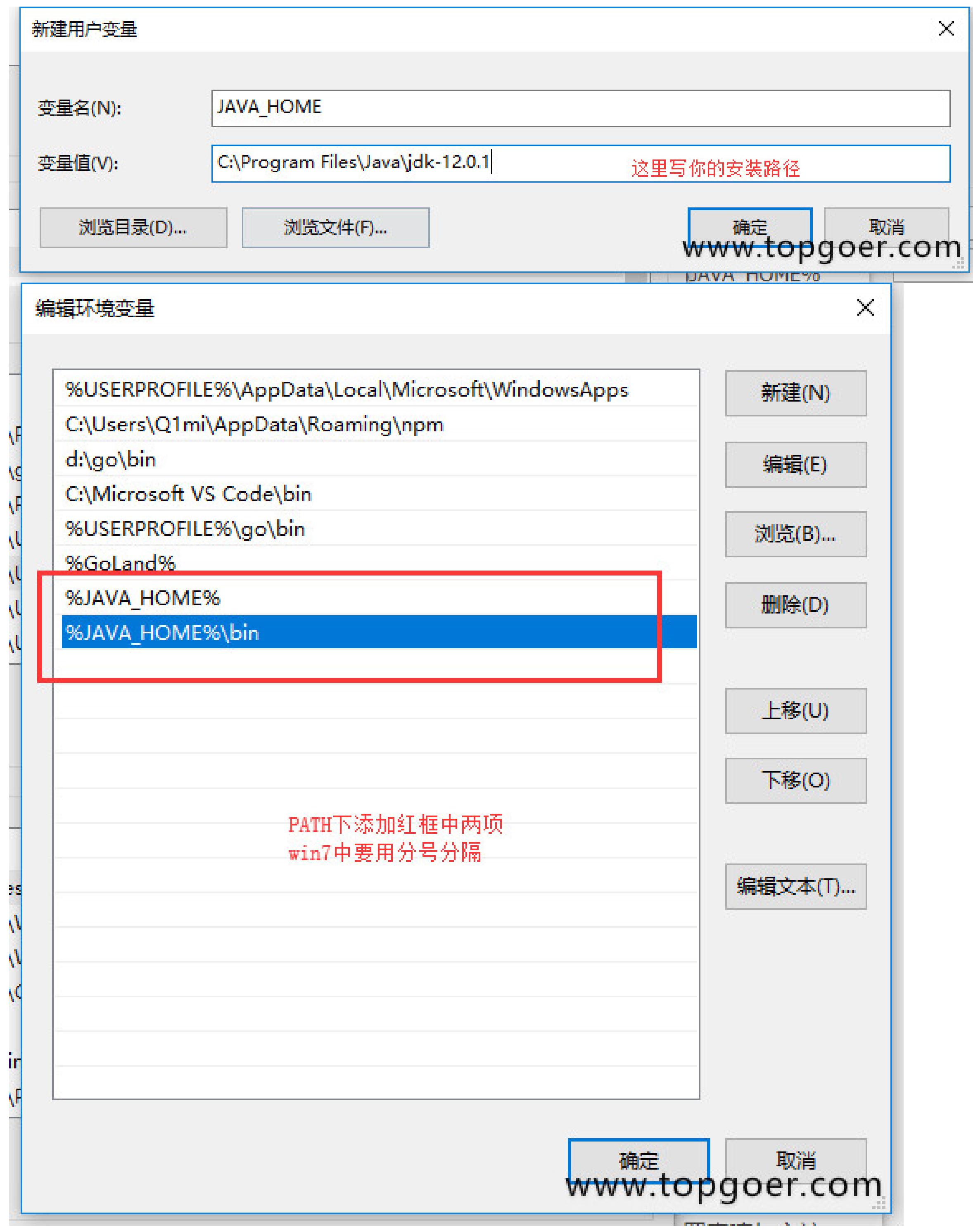
下载地址：https://www.oracle.com/technetwork/java/javase/downloads/jdk12-downloads-5295953.html

安装JDK

Java(TM) SE Development Kit 12.0.1 (64-bit) - 目标文件夹
Java
ORACLE
Java(TM) SE Development Kit 12.0.1 (64-bit)，包括专用 JRE 和 src.zip。它要求硬盘驱动器上有 420MB 空间。单击"更改"按钮可更改安装文件夹。
将 Java(TM) SE Development Kit 12.0.1 (64-bit) 安装到：
C:\Program Files\Java\jdk-12.0.1\
更改...
上一步(B)
下一步(N)
取消
www.topgoer.com
Java(TM) SE Development Kit 12.0.1 (64-bit) - 进度
Java
ORACLE
状态: 正在更新组件注册
www.topgoer.com



添加环境变量



注意：第二部本来需要安装zookeeper的但是现在kafka上面自带这个了！可以安装可以不安装！我不找刺激就不安装了！

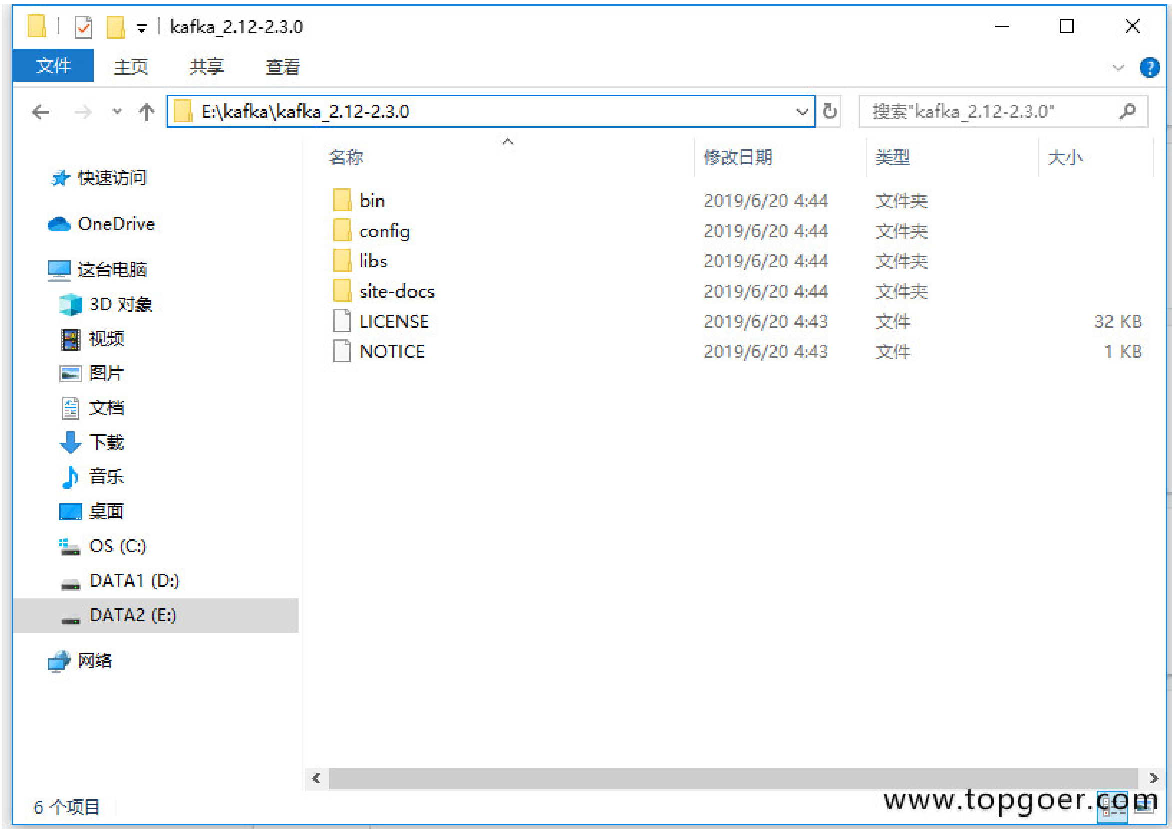
安装kafka

下载

下载地址：http://kafka.apache.org/downloads 下载kafka_2.12-2.3.0.tgz。

安装

将下载好的压缩包解压到本地即可。

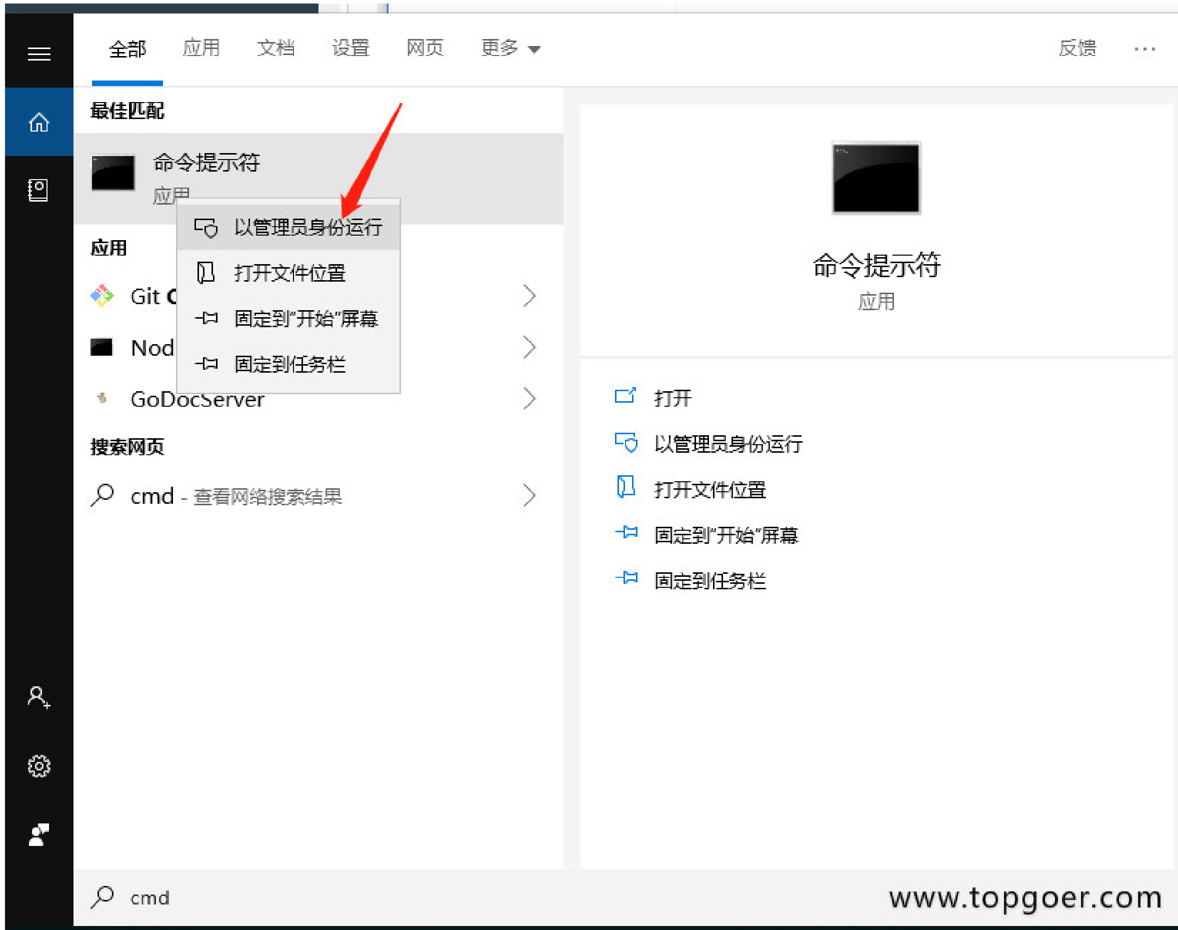


配置

```
1. 打开config目录下的server.properties文件
2. 修改log.dirs=E:\\kafkalogs
3. 打开config目录下的zookeeper.properties文件
4. 修改dataDir=E:\\kafka\\zookeeper
```

启动

注意：记得啊执行cmd记得以管理员的身份启动不然启动不了！ 两次都需要管理员身份（另外记得走到kafka你解压的目录执行下面的命令啊




```
先执行：bin\windows\zookeeper-server-start.bat config\zookeeper.properties

再执行：bin\windows\kafka-server-start.bat config\server.properties
```

进行单机实例测试简单使用

windows使用的是路径E:\Kafka\kafka_2.12-2.0.0\bin\windows下批处理命令，如有问题，参见

步骤：

1）启动kafka内置的zookeeper

.\bin\windows\zookeeper-server-start.bat .\config\zookeeper.properties

出现binding to port …表示zookeeper启动成功，不关闭页面

2）kafka服务启动，成功不关闭页面

.\bin\windows\kafka-server-start.bat .\config\server.properties

3）创建topic测试主题kafka，成功不关闭页面

.\bin\windows\kafka-topics.bat –create –zookeeper localhost:2181 –replication-factor 1 –partitions 1 –topic test

4）创建生产者产生消息，不关闭页面

.\bin\windows\kafka-console-producer.bat –broker-list localhost:9092 –topic test

```
C:\WINDOWS\system32\cmd.exe - .\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic kafka
Microsoft Windows [版本 10.0.17134.228]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\admin>cd E:\Kafka\kafka_2.12-2.0.0

C:\Users\admin>e:

E:\Kafka\kafka_2.12-2.0.0>.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic kafka
>this is producer message
>hello,consumer
>
```

5）创建消费者接收消息，不关闭页面

.\bin\windows\kafka-console-consumer.bat –bootstrap-server localhost:9092 –topic test –from-beginning

```
C:\WINDOWS\system32\cmd.exe - .\bin\windows\kafka-console-consumer.bat  --bootstrap-server localhost:9092 --topic kafka --from-b...
Microsoft Windows [版本 10.0.17134.228]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\admin>cd E:\Kafka\kafka_2.12-2.0.0

C:\Users\admin>e:

E:\Kafka\kafka_2.12-2.0.0>.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic kafka --from-beginning
this is producer message
hello,consumer
```

https://blog.csdn.net/github_38482082

```
C:\WINDOWS\system32\cmd.exe - .\bin\windows\kafka-console-producer.bat  --broker-list localhost:9092 --topic kafka
Microsoft Windows [版本 10.0.17134.228]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\admin>cd E:\Kafka\kafka_2.12-2.0.0

C:\Users\admin>e:

E:\Kafka\kafka_2.12-2.0.0>.\bin\windows\kafka-console-producer.bat --broker-list localhost:9092 --topic kafka
>this is producer message
>hello,consumer
>你好
>have you receive?
>receive message
>great
>中文乱码
>
```

生产发送方

```
C:\WINDOWS\system32\cmd.exe - .\bin\windows\kafka-console-consumer.bat  --bootstrap-server localhost:9092 --topic kafka --from-b...
Microsoft Windows [版本 10.0.17134.228]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\admin>cd E:\Kafka\kafka_2.12-2.0.0

C:\Users\admin>e:

E:\Kafka\kafka_2.12-2.0.0>.\bin\windows\kafka-console-consumer.bat --bootstrap-server localhost:9092 --topic kafka --from-beginning
this is producer message
hello,consumer
锟斤拷锟
have you receive?
receive message
great
锟斤拷锟斤拷锟斤拷锟斤拷锟斤拷
```

接收消费方

中文乱码

https://blog.csdn.net/github_38482082

操作Kafka

sarama

Go语言中连接kafka使用第三方库: github.com/Shopify/sarama。

下载及安装

```
go get github.com/Shopify/sarama
```

注意事项:
sarama v1.20之后的版本加入了zstd压缩算法，需要用到cgo，在Windows平台编译时会提示类似如下错误：
github.com/DataDog/zstd
exec: “gcc”:executable file not found in %PATH%
所以在Windows平台请使用v1.19版本的sarama。(如果不会版本控制请查看博客里面的go module章节)

连接kafka发送消息

```
package main

import (
    "fmt"

    "github.com/Shopify/sarama"
)

// 基于sarama第三方库开发的kafka client

func main() {
    config := sarama.NewConfig()
    config.Producer.RequiredAcks = sarama.WaitForAll          // 发送完数据需要leader和follow都确认
    config.Producer.Partitioner = sarama.NewRandomPartitioner // 新选出一个partition
    config.Producer.Return.Successes = true                   // 成功交付的消息将在success channel返回

    // 构造一个消息
    msg := &sarama.ProducerMessage{}
```

```
    msg.Topic = "web_log"
    msg.Value = sarama.StringEncoder("this is a test log")
    // 连接kafka
    client, err := sarama.NewSyncProducer([]string{"127.0.0.1:9092"}, config)
    if err != nil {
        fmt.Println("producer closed, err:", err)
        return
    }
    defer client.Close()
    // 发送消息
    pid, offset, err := client.SendMessage(msg)
    if err != nil {
        fmt.Println("send msg failed, err:", err)
        return
    }
    fmt.Printf("pid:%v offset:%v\n", pid, offset)
}
```

连接kafka消费消息

```
package main

import (
    "fmt"

    "github.com/Shopify/sarama"
)

// kafka consumer

func main() {
    consumer, err := sarama.NewConsumer([]string{"127.0.0.1:9092"}, nil)
    if err != nil {
        fmt.Printf("fail to start consumer, err:%v\n", err)
        return
    }
    partitionList, err := consumer.Partitions("web_log") // 根据topic取到所有的分区
    if err != nil {
        fmt.Printf("fail to get list of partition:err%v\n", err)
        return
    }
    fmt.Println(partitionList)
    for partition := range partitionList { // 遍历所有的分区
        // 针对每个分区创建一个对应的分区消费者
```

```
        pc, err := consumer.ConsumePartition("web_log", int32(partition), sarama.OffsetNewest)
        if err != nil {
            fmt.Printf("failed to start consumer for partition %d,err:%v\n", partition, err)
            return
        }
        defer pc.AsyncClose()
        // 异步从每个分区消费信息
        go func(sarama.PartitionConsumer) {
            for msg := range pc.Messages() {
                fmt.Printf("Partition:%d Offset:%d Key:%v Value:%v", msg.Partition, msg.Offset, msg.Key, msg.Value)
            }
        }(pc)
    }
}
```

go操作RabbitMQ

RabbitMQ介绍

RabbitMQ安装

Simple模式

Work模式

Publish模式

Routing模式

Topic模式

RabbitMQ介绍

MQ简介

简单释义

消息总线(Message Queue)，是一种跨进程、异步的通信机制，用于上下游传递消息。由消息系统来确保消息的可靠传递。

背景描述

当前市面上mq的产品很多，比如RabbitMQ、Kafka、ActiveMQ、ZeroMQ和阿里巴巴捐献给Apache的RocketMQ。甚至连redis这种NoSQL都支持MQ的功能。

适用场景

- 上下游逻辑解耦&&物理解耦
- 保证数据最终一致性
- 广播
- 错峰流控等等

RabbitMQ的特点

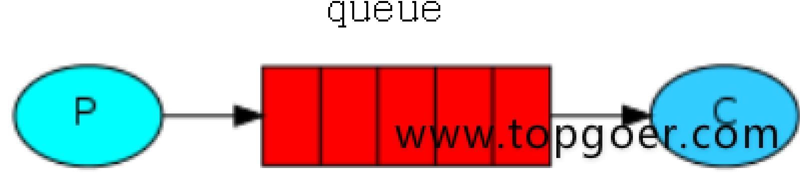
> RabbitMQ是由Erlang语言开发的AMQP的开源实现。

AMQP：Advanced Message Queue，高级消息队列协议。它是应用层协议的一个开放标准，为面向消息的中间件设计，基于此协议的客户端与消息中间件可传递消息，并不受产品、开发语言灯条件的限制。

- `可靠性(Reliablity)：`使用了一些机制来保证可靠性，比如持久化、传输确认、发布确认。
- `灵活的路由(Flexible Routing)：`在消息进入队列之前，通过Exchange来路由消息。对于典型的路由功能，Rabbit已经提供了一些内置的Exchange来实现。针对更复杂的路由功能，可以将多个Exchange绑定在一起，也通过插件机制实现自己的Exchange。
- `消息集群(Clustering)：`多个RabbitMQ服务器可以组成一个集群，形成一个逻辑Broker。
- `高可用(Highly Avaliable Queues)`：队列可以在集群中的机器上进行镜像，使得在部分节点出问题的情况下队列仍然可用。

- `多种协议(Multi-protocol):` 支持多种消息队列协议，如STOMP、MQTT等。
- `多种语言客户端(Many Clients):` 几乎支持所有常用语言，比如Java、.NET、Ruby等。
- `管理界面(Management UI)` ：提供了易用的用户界面，使得用户可以监控和管理消息Broker的许多方面。
- `跟踪机制(Tracing)` ：如果消息异常，RabbitMQ提供了消息的跟踪机制，使用者可以找出发生了什么。
- `插件机制(Plugin System)` ：提供了许多插件，来从多方面进行扩展，也可以编辑自己的插件。

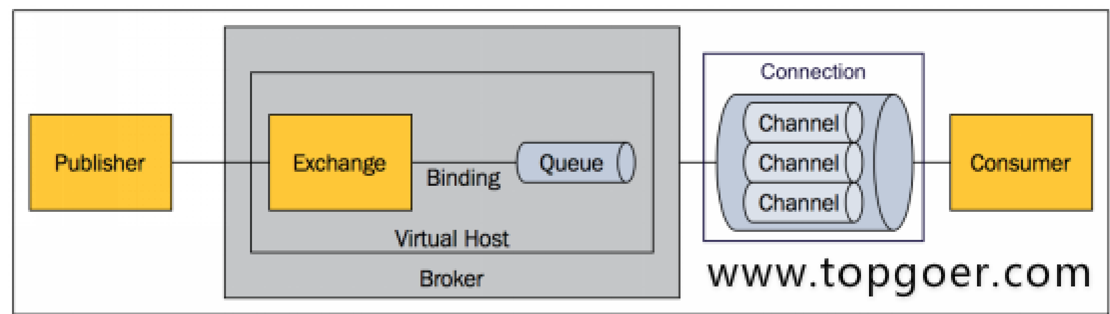
rabbitmq简单使用



所有MQ产品从模型抽象来说，都是一样的过程：

- 消费者(consumer)订阅某个队列。
- 生产者(product)创建消息，然后发布到队列中(queue)，最终将消息发送到监听的消费者。

> 这只是最简单抽象的描述，具体到RabbitMQ则由更详细的概念需要解释。



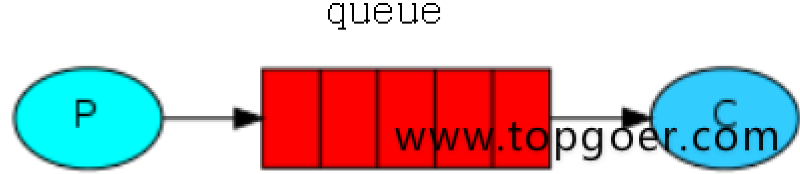
- `Broker:` 标识消息队列服务器实体.
- `Virtual Host:` 虚拟主机。标识一批交换机、消息队列和相关对象。虚拟主机是共享相同的身份认证和加密环境的独立服务器域。每个vhost本质上就是一个mini版的

RabbitMQ服务器，拥有自己的队列、交换器、绑定和权限机制。vhost是AMQP概念的基础，必须在链接时指定，RabbitMQ默认的vhost是 /。

- `Exchange:` 交换器，用来接收生产者发送的消息并将这些消息路由给服务器中的队列。
- `Queue:` 消息队列，用来保存消息直到发送给消费者。它是消息的容器，也是消息的终点。一个消息可投入一个或多个队列。消息一直在队列里面，等待消费者连接到这个队列将其取走。
- `Banding:` 绑定，用于消息队列和交换机之间的关联。一个绑定就是基于路由键将交换机和消息队列连接起来的路由规则，所以可以将交换器理解成一个由绑定构成的路由表。
- `Channel:` 信道，多路复用连接中的一条独立的双向数据流通道。新到是建立在真实的TCP连接内地虚拟链接，AMQP命令都是通过新到发出去的，不管是发布消息、订阅队列还是接收消息，这些动作都是通过信道完成。因为对于操作系统来说，建立和销毁TCP都是非常昂贵的开销，所以引入了信道的概念，以复用一条TCP连接。
- `Connection:` 网络连接，比如一个TCP连接。
- `Publisher:` 消息的生产者，也是一个向交换器发布消息的客户端应用程序。
- `Consumer:` 消息的消费者，表示一个从一个消息队列中取得消息的客户端应用程序。
- `Message:` 消息，消息是不具名的，它是由消息头和消息体组成。消息体是不透明的，而消息头则是由一系列的可选属性组成，这些属性包括routing-key(路由键)、priority(优先级)、delivery-mode(消息可能需要持久性存储[消息的路由模式])等。

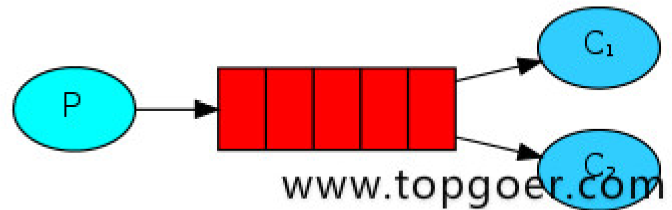
RabbitMQ的六种工作模式

simple简单模式



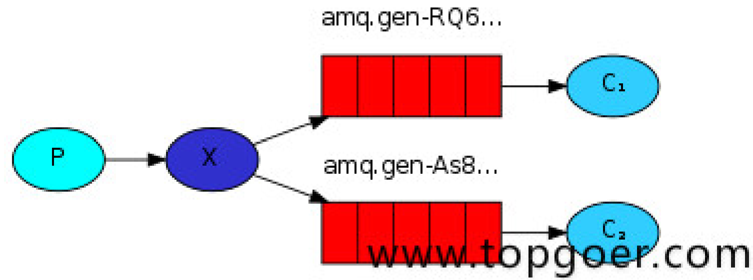
- 消息产生着§将消息放入队列
- 消息的消费者(consumer) 监听(while) 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失)应用场景:聊天(中间有一个过度的服务器;p端,c端)

work工作模式(资源的竞争)



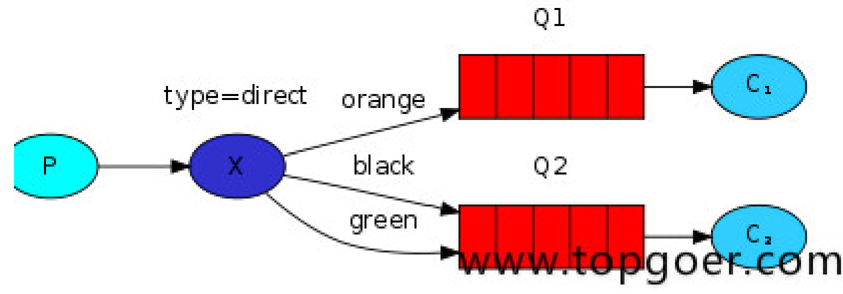
- 消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2,同时监听同一个队列,消息被消费?C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患,高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize,与同步锁的性能不一样) 保证一条消息只能被一个消费者使用)
- 应用场景:红包;大项目中的资源调度(任务分配系统不需知道哪一个任务执行系统在空闲,直接将任务扔到消息队列中,空闲的系统自动争抢)

publish/subscribe发布订阅(共享资源)



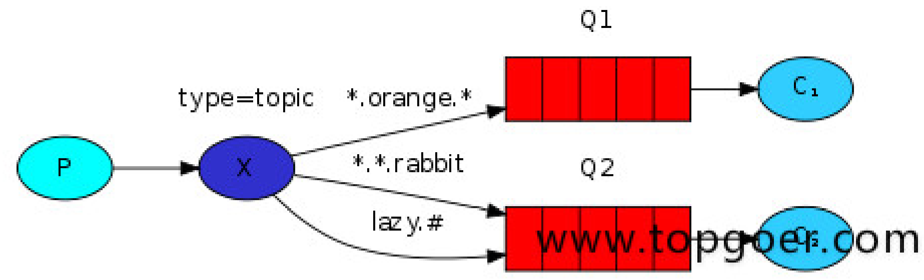
- X代表交换机rabbitMQ内部组件,erlang 消息产生者是代码完成,代码的执行效率不高,消息产生者将消息放入交换机,交换机发布订阅把消息发送到所有消息队列中,对应消息队列的消费者拿到消息进行消费
- 相关场景:邮件群发,群聊天,广播(广告)

routing路由模式



- 消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;
- 根据业务功能定义路由字符串
- 从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可以自定义消费者,实时接收错误;

topic 主题模式(路由模式的一种)



- 星号井号代表通配符
- 星号代表多个单词,井号代表一个单词
- 路由功能添加模糊匹配
- 消息产生者产生消息,把消息交给交换机
- 交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消费

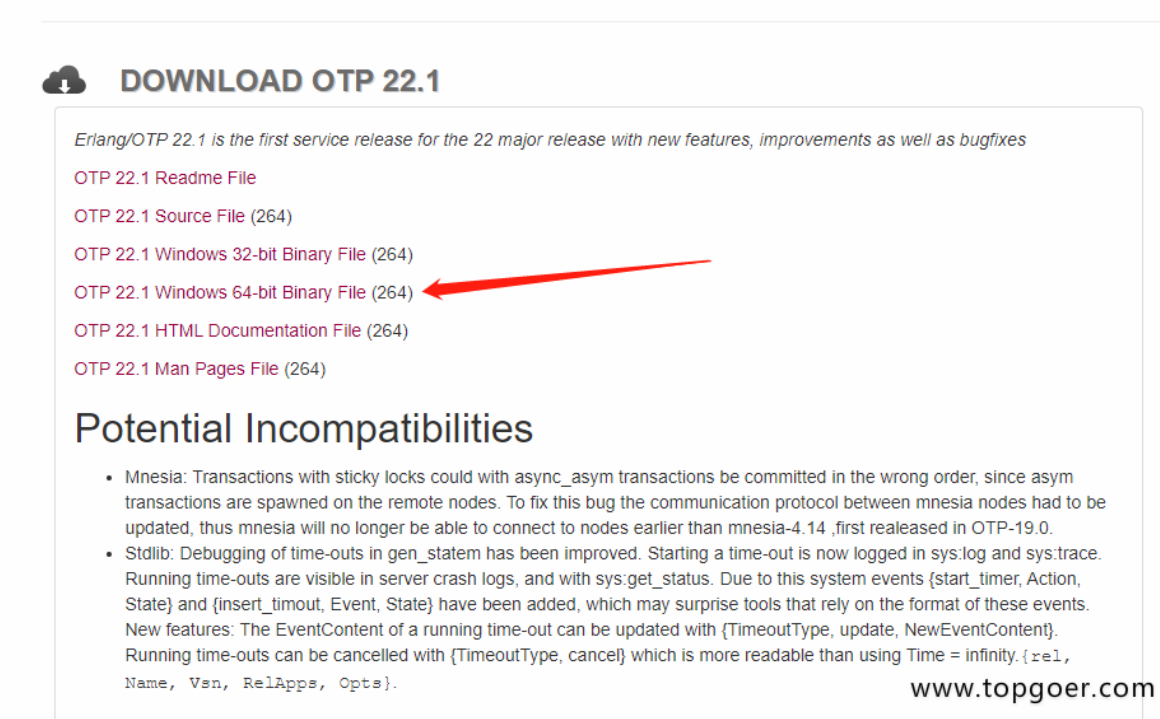
RPC (先不做解释后续补充)

RabbitMQ安装

win下安装

第一步：下载并安装erlang

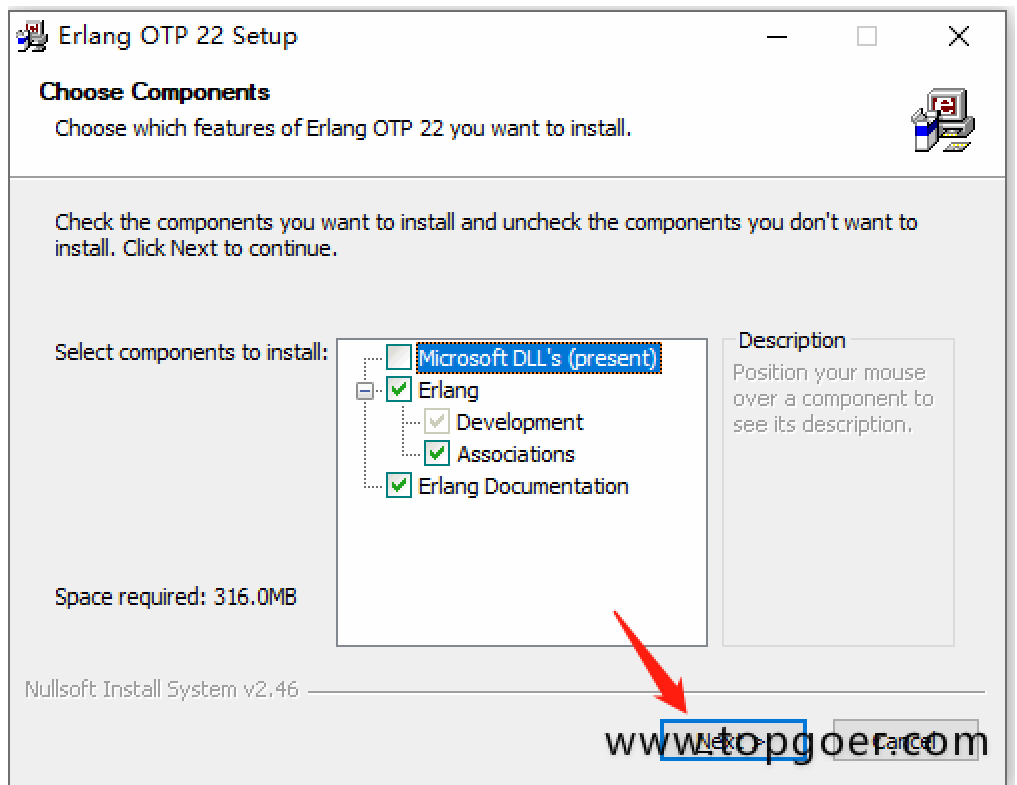
- 原因：RabbitMQ服务端代码是使用并发式语言Erlang编写的，安装Rabbit MQ的前提是安装Erlang。
- 下载地址：http://www.erlang.org/downloads



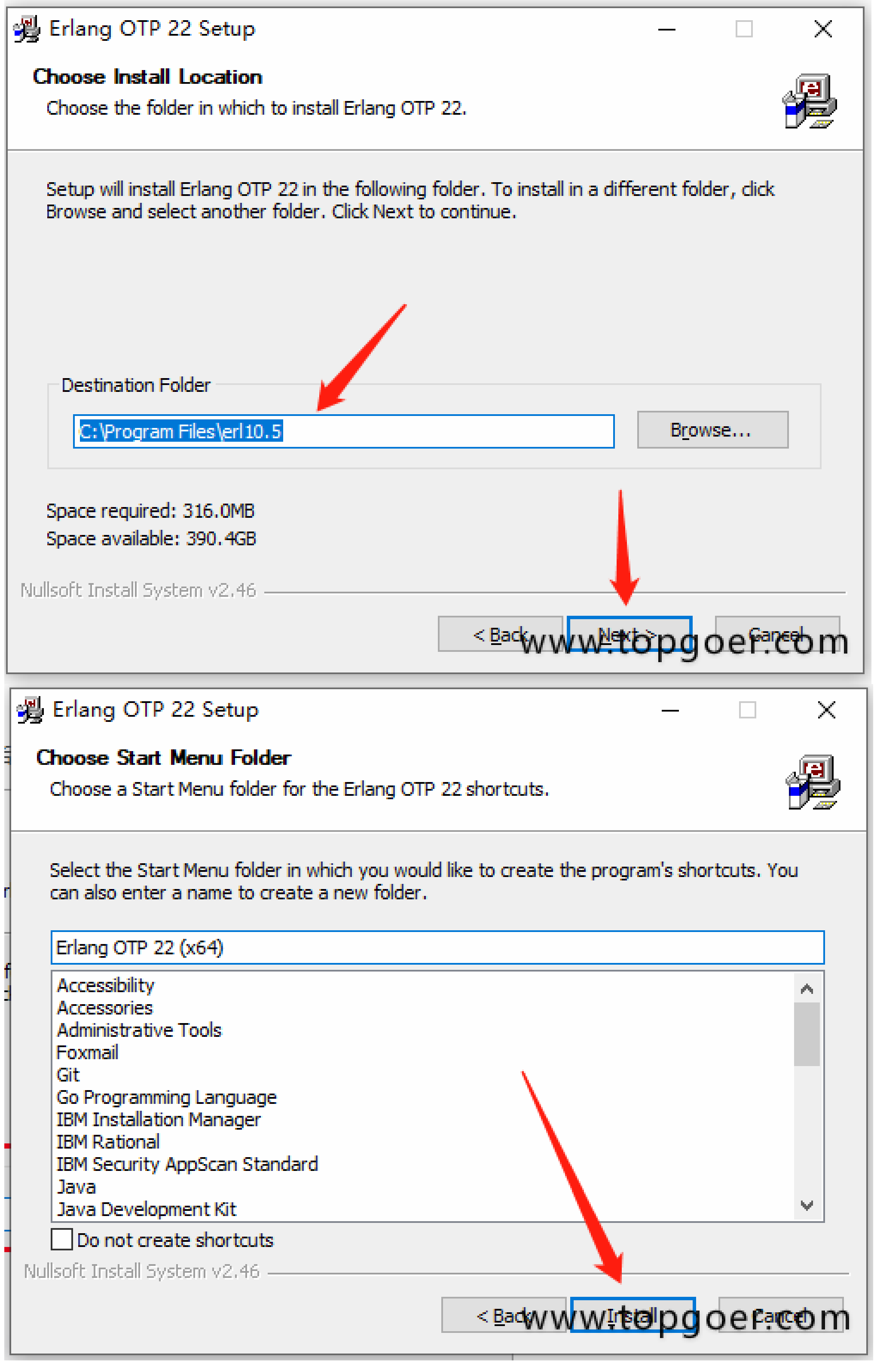
根据本机位数选择erlang下载版本。

- 下载完是这么个东西：

- 双击，点next就可以。



- 选择一个自己想保存的地方，然后next、finish就可以。



- 安装完事儿后要记得配置一下系统的环境变量。

  此电脑->鼠标右键“属性”->高级系统设置->环境变量->“新建”系统环境变量

新建系统变量

变量名(N): ERLANG_HOME

变量值(V): E:\RabbitMQ\Erlang\erl10.5

浏览目录(D)... 浏览文件(F)... 确定 取消

www.topgoer.com

变量名：ERLANG_HOME

变量值就是刚才erlang的安装地址，点击确定。

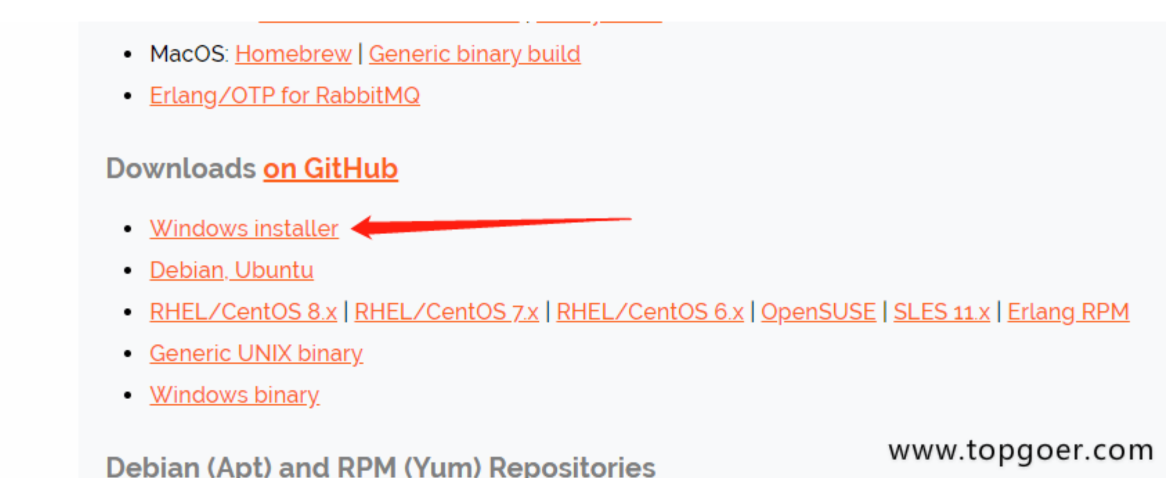
- 然后双击系统变量path

点击“新建”，将%ERLANG_HOME%\bin加入到path中。

- 最后windows键+R键，输入cmd，再输入erl，看到版本号就说明erlang安装成功了。

第二步：下载并安装RabbitMQ

- 下载地址：http://www.rabbitmq.com/download.html

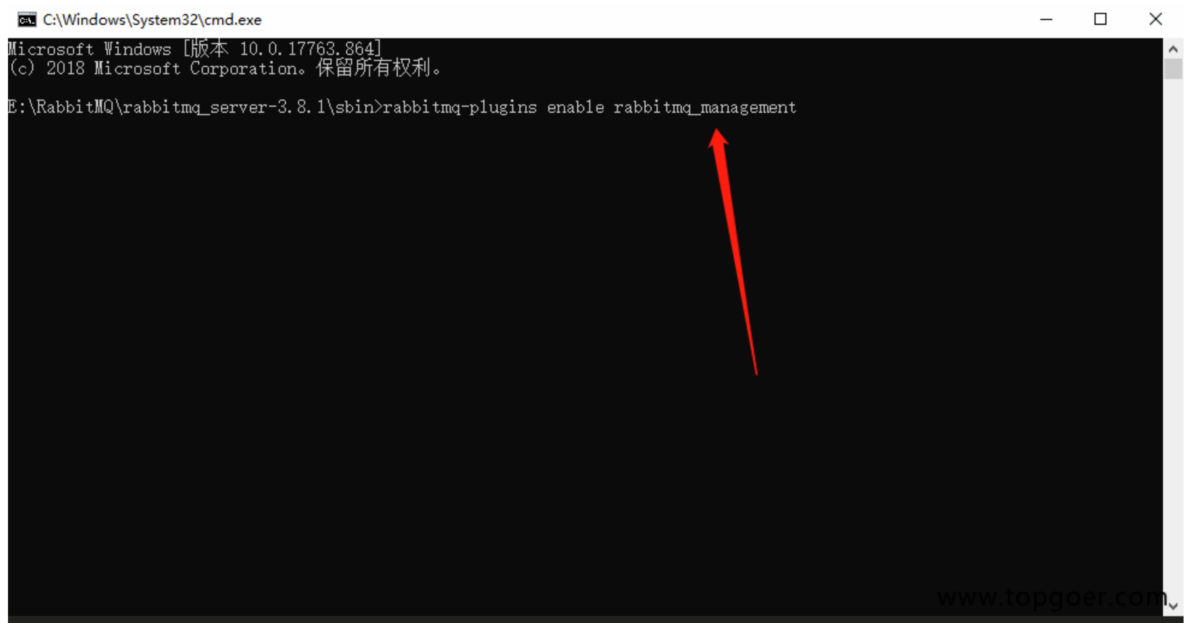


- 双击下载后的.exe文件，安装过程与erlang的安装过程相同。

- RabbitMQ安装好后接下来安装RabbitMQ-Plugins。打开命令行cd，输入RabbitMQ的sbin目录。

我的目录是：E:\RabbitMQ\rabbitmq_server-3.8.1\sbin

然后在后面输入rabbitmq-plugins enable rabbitmq_management命令进行安装



如果出现下面的提示表示运行成功

```
E:\RabbitMQ\rabbitmq_server-3.8.1\sbin>rabbitmq-plugins.bat enable rabbitmq_management
Enabling plugins on node rabbit@DESKTOP-VMKOFN7:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DESKTOP-VMKOFN7...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.
```
www.topgoer.com

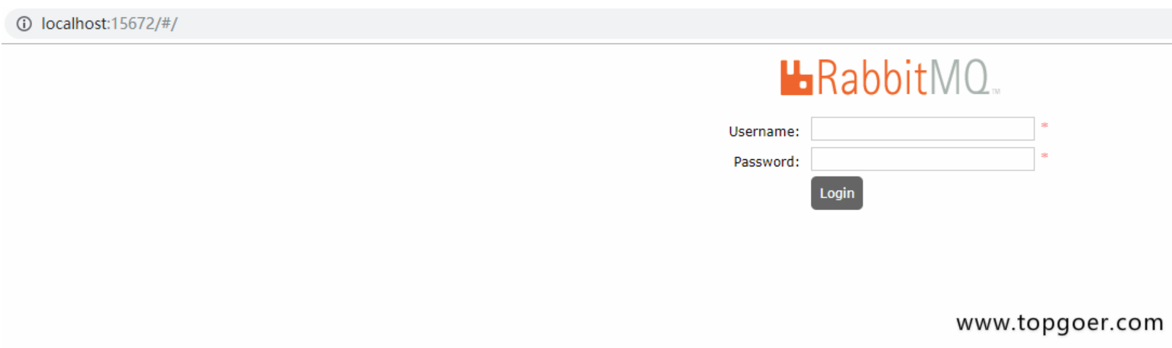
输入命令：rabbitmq-server.bat

如果出现下面的提示表示启动成功

```
E:\RabbitMQ\rabbitmq_server-3.8.1\sbin>rabbitmq-server.bat

  ##  ##      RabbitMQ 3.8.1
  ##  ##
  ##########  Copyright (c) 2007-2019 Pivotal Software, Inc.
  ######  ##
  ##########  Licensed under the MPL 1.1. Website: https://rabbitmq.com

  Doc guides: https://rabbitmq.com/documentation.html
  Support:    https://rabbitmq.com/contact.html
  Tutorials:  https://rabbitmq.com/getstarted.html
  Monitoring: https://rabbitmq.com/monitoring.html

  Logs: C:/Users/AT/AppData/Roaming/RabbitMQ/log/rabbit@DESKTOP-VMKOFN7.log
        C:/Users/AT/AppData/Roaming/RabbitMQ/log/rabbit@DESKTOP-VMKOFN7_upgrade.log

  Config file(s): (none)

  Starting broker... completed with 3 plugins.
```

www.topgoer.com

rabbitmq启动成功，浏览器中http://localhost:15672，



输入guest,guest进入rabbitMQ管理控制台：

localhost:15672/#/
RabbitMQ 3.8.1 Erlang 22.1
Overview Connections Channels Exchanges Queues Admin
Overview
Totals
Queued messages last minute ?
Currently idle
Message rates last minute ?
Currently idle
Global counts ?
Connections: 0 Channels: 0 Exchanges: 7 Queues: 0 Consumers: 0
Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats
rabbit@DESKTOP-VMKOFN7	0 65536 available	0 58893 available	434 1048576 available	89MiB 3.2GiB high watermark	392GiB 48MiB low watermark	2h 8m	basic disc 1 rss	This node All nodes

Churn statistics
Ports and contexts
Export definitions
Import definitions
HTTP API Server Docs Tutorials Community Support Community Slack Commercial Support Plugins GitHub Changelog

www.topgoer.com

注意

上面的安装步骤如果出现下面错误

- 输入rabbitmq-plugins enable rabbitmq_management命令出现

```
Microsoft Windows [版本 10.0.17763.864]
(c) 2018 Microsoft Corporation。保留所有权利。

E:\RabbitMQ\rabbitmq_server-3.8.1\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@DESKTOP-VMKOFN7:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DESKTOP-VMKOFN7...
Plugin configuration unchanged.
```

www.topgoer.com

这样的提示解决办法

解决方法：

将 `C:\Users\Administrator\.erlang.cookie`

同步至 `C:\Windows\System32\config\systemprofile\.erlang.cookie`

同时删除： `C:\Users\Administrator\AppData\Roaming\RabbitMQ目录`

重新输入rabbitmq-plugins enable rabbitmq_management命令

- 运行命令：rabbitmq-server.bat

如果出现下面的提示：

```
E:\RabbitMQ\rabbitmq_server-3.8.1\sbin>rabbitmq-server.bat
ERROR: node with name "rabbit" already running on "DESKTOP-VMKOFN7"
```

这个是因为rabbit已经启动了，不能再次启动，通过tasklist指令，发现进程是存在的：

```
tasklist | find /i "erl"
```

```
E:\RabbitMQ\rabbitmq_server-3.8.1\sbin>tasklist | find /i "erl"
erlsrv.exe                    3844 Services                   0      3,240 K
erl.exe                       3084 Services                   0     78,352 K
```

www.topgoer.com

如果有结果，那么说明已经启动了，通过任务管理器kill掉进程再次启动即可。

`ctrl+alt+delete` 进入 `任务管理器`

dllhost.exe	11652	正在运行	AT	00	768 K	已禁用
dwm.exe	1164	正在运行	DWM-1	00	42,516 K	已禁用
epmd.exe	5068	正在运行	SYSTEM	00	448 K	不允许
erl.exe	7308	正在运行	AT	00	32,292 K	已禁用
explorer.exe	6268	正在运行	AT	00	40,228 K	已禁用
FNPLicensingServic...	3276	正在运行	SYSTEM	00	1,068 K	不允许
fontdrvhost.exe	620	正在运行	UMFD-0	00	20 K	已禁用
fontdrvhost.exe	628	正在运行	UMFD-1	00	1,376 K	已禁用

www.topgoer.com

右键->结束任务就好了

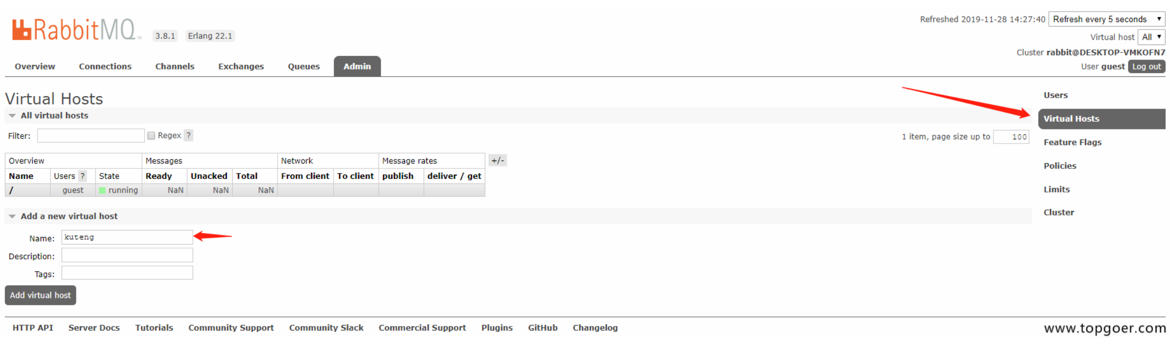
Simple模式

Simple模式

- 消息产生着§将消息放入队列
- 消息的消费者(consumer) 监听(while) 消息队列,如果队列中有消息,就消费掉,消息被拿走后,自动从队列中删除(隐患 消息可能没有被消费者正确处理,已经从队列中消失了,造成消息的丢失)应用场景:聊天(中间有一个过度的服务器;p端,c端)

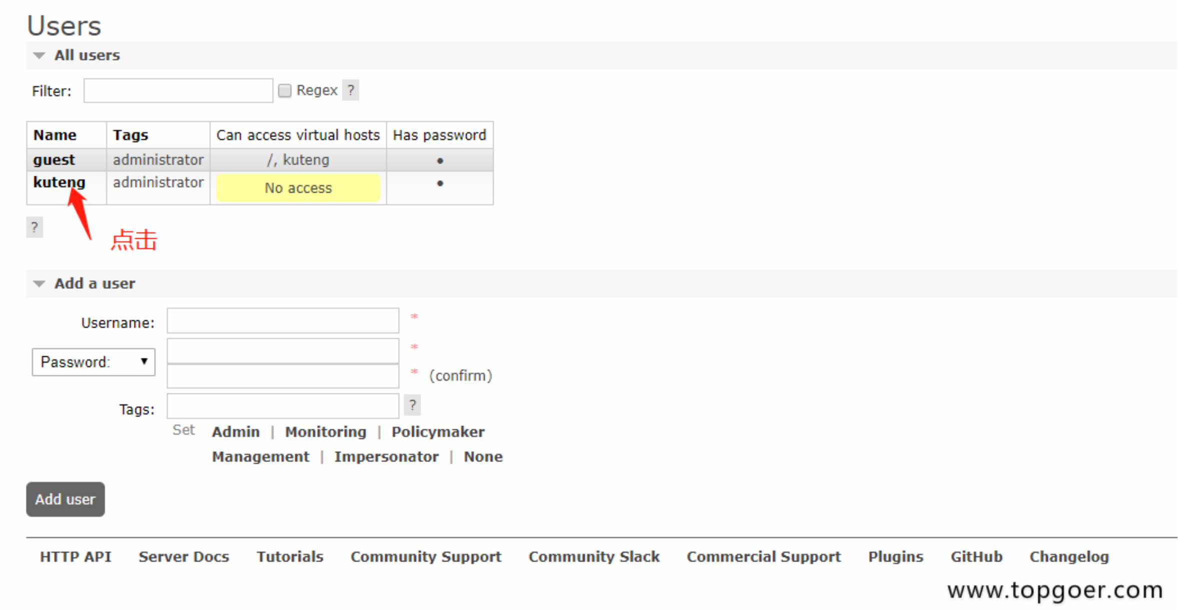
做simple简单模式之前首先我们新建一个 `Virtual Host` 并且给他分配一个用户名，用来隔离数据，根据自己需要自行创建

第一步



第二步

第三步



第四步

▼ Overview

Tags administrator

Can log in with password •

▼ Permissions

Current permissions

... no permissions ...

Set permission

Virtual Host: kuteng

Configure regexp: .*

Write regexp: .*

Read regexp: .*

Set permission

▼ Topic permissions

Current topic permissions

... no topic permissions ...

Set topic permission

Virtual Host: /

Exchange: (AMQP default)

Write regexp: .*

Read regexp: .*

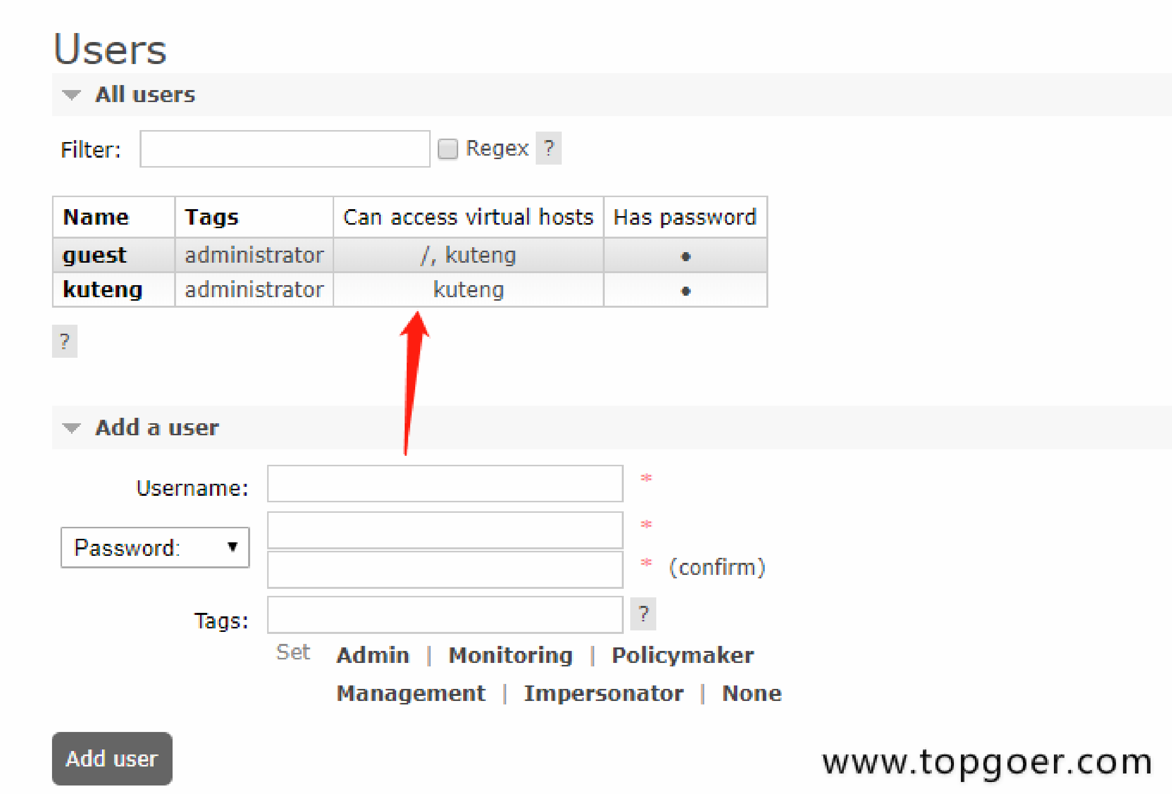
Set topic permission

▶ Update this user

▶ Delete this user

www.topgoer.com

第五步



代码层面

目录结构

kuteng-RabbitMQ

-RabbitMQ

–rabitmq.go //这个是RabbitMQ的封装

-SimlpePublish

–mainSimlpePublish.go //Publish 先启动

-SimpleRecieve

–mainSimpleRecieve.go

rabitmq.go代码

```
package RabbitMQ

import (
	"fmt"
	"log"

	"github.com/streadway/amqp"
)

//连接信息amqp://kuteng:kuteng@127.0.0.1:5672/kuteng这个信息是固定不变的amqp://
事固定参数后面两个是用户名密码ip地址端口号Virtual Host
const MQURL = "amqp://kuteng:kuteng@127.0.0.1:5672/kuteng"

//rabbitMQ结构体
type RabbitMQ struct {
	conn    *amqp.Connection
	channel *amqp.Channel
	//队列名称
	QueueName string
	//交换机名称
	Exchange string
	//bind Key 名称
	Key string
	//连接信息
	Mqurl string
}

//创建结构体实例
func NewRabbitMQ(queueName string, exchange string, key string) *RabbitMQ {
	return &RabbitMQ{QueueName: queueName, Exchange: exchange, Key: key, Mqurl:
 MQURL}
}

//断开channel 和 connection
func (r *RabbitMQ) Destory() {
	r.channel.Close()
	r.conn.Close()
}

//错误处理函数
func (r *RabbitMQ) failOnErr(err error, message string) {
	if err != nil {
```

```
        log.Fatalf("%s:%s", message, err)
        panic(fmt.Sprintf("%s:%s", message, err))
    }
}

//创建简单模式下RabbitMQ实例
func NewRabbitMQSimple(queueName string) *RabbitMQ {
    //创建RabbitMQ实例
    rabbitmq := NewRabbitMQ(queueName, "", "")
    var err error
    //获取connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.Mqurl)
    rabbitmq.failOnErr(err, "failed to connect rabb"+
        "itmq!")
    //获取channel
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "failed to open a channel")
    return rabbitmq
}

//直接模式队列生产
func (r *RabbitMQ) PublishSimple(message string) {
    //1.申请队列，如果队列不存在会自动创建，存在则跳过创建
    _, err := r.channel.QueueDeclare(
        r.QueueName,
        //是否持久化
        false,
        //是否自动删除
        false,
        //是否具有排他性
        false,
        //是否阻塞处理
        false,
        //额外的属性
        nil,
    )
    if err != nil {
        fmt.Println(err)
    }
    //调用channel 发送消息到队列中
    r.channel.Publish(
        r.Exchange,
        r.QueueName,
        //如果为true，根据自身exchange类型和routekey规则无法找到符合条件的队列会把消息返还给发送者
        false,
```

```
		//如果为true，当exchange发送消息到队列后发现队列上没有消费者，则会把消息返还给发送者
		false,
		amqp.Publishing{
			ContentType: "text/plain",
			Body:        []byte(message),
		})
}

//simple 模式下消费者
func (r *RabbitMQ) ConsumeSimple() {
	//1.申请队列，如果队列不存在会自动创建，存在则跳过创建
	q, err := r.channel.QueueDeclare(
		r.QueueName,
		//是否持久化
		false,
		//是否自动删除
		false,
		//是否具有排他性
		false,
		//是否阻塞处理
		false,
		//额外的属性
		nil,
	)
	if err != nil {
		fmt.Println(err)
	}

	//接收消息
	msgs, err := r.channel.Consume(
		q.Name, // queue
		//用来区分多个消费者
		"", // consumer
		//是否自动应答
		true, // auto-ack
		//是否独有
		false, // exclusive
		//设置为true，表示 不能将同一个Conenction中生产者发送的消息传递给这个Connection中 的消费者
		false, // no-local
		//列是否阻塞
		false, // no-wait
		nil,   // args
	)
	if err != nil {
```

```
        fmt.Println(err)
    }

    forever := make(chan bool)
    //启用协程处理消息
    go func() {
        for d := range msgs {
            //消息逻辑处理，可以自行设计逻辑
            log.Printf("Received a message: %s", d.Body)

        }
    }()

    log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
    <-forever

}
```

mainSimlpePublish.go代码

```
package main

import (
    "fmt"

    "github.com/student/kuteng-RabbitMQ/RabbitMQ"
)

func main() {
    rabbitmq := RabbitMQ.NewRabbitMQSimple("" +
        "kuteng")
    rabbitmq.PublishSimple("Hello kuteng222!")
    fmt.Println("发送成功！")
}
```

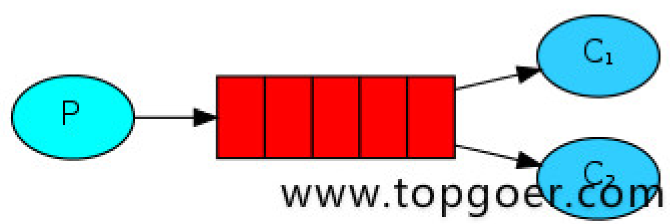
mainSimpleRecieve.go代码

```
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
    rabbitmq := RabbitMQ.NewRabbitMQSimple("" +
        "kuteng")
```

```
    rabbitmq.ConsumeSimple()
}
```

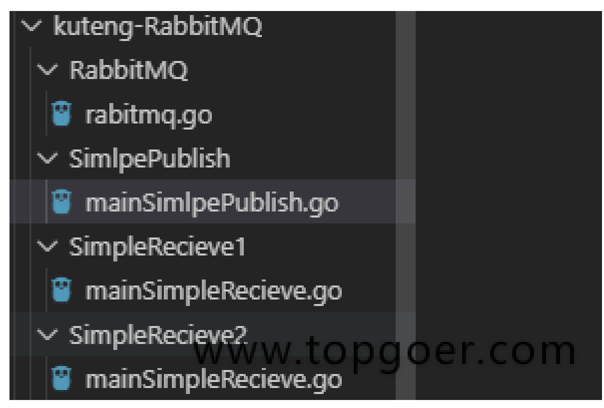
Work模式

Work模式（工作模式，一个消息只能被一个消费者获取）



- 消息产生者将消息放入队列消费者可以有多个,消费者1,消费者2,同时监听同一个队列,消息被消费?C1 C2共同争抢当前的消息队列内容,谁先拿到谁负责消费消息(隐患,高并发情况下,默认会产生某一个消息被多个消费者共同使用,可以设置一个开关(syncronize,与同步锁的性能不一样) 保证一条消息只能被一个消费者使用)
- 应用场景:红包;大项目中的资源调度(任务分配系统不需知道哪一个任务执行系统在空闲,直接将任务扔到消息队列中,空闲的系统自动争抢)

目录结构



kuteng-RabbitMQ

-RabbitMQ

–rabitmq.go //这个是RabbitMQ的封装和Simple模式代码一样

-SimlpePublish

–mainSimlpePublish.go //Publish 先启动

-SimpleRecieve1

–mainSimpleRecieve.go

-SimpleRecieve2

–mainSimpleRecieve.go

注意

Work模式和Simple模式相比代码并没有发生变化只是多了一个消费者

rabitmq.go代码

```
package RabbitMQ

import (
	"fmt"
	"log"

	"github.com/streadway/amqp"
)

//连接信息amqp://kuteng:kuteng@127.0.0.1:5672/kuteng这个信息是固定不变的amqp://事固定参数后面两个是用户名密码ip地址端口号Virtual Host
const MQURL = "amqp://kuteng:kuteng@127.0.0.1:5672/kuteng"

//rabbitMQ结构体
type RabbitMQ struct {
	conn    *amqp.Connection
	channel *amqp.Channel
	//队列名称
	QueueName string
	//交换机名称
	Exchange string
	//bind Key 名称
	Key string
	//连接信息
	Mqurl string
}

//创建结构体实例
func NewRabbitMQ(queueName string, exchange string, key string) *RabbitMQ {
	return &RabbitMQ{QueueName: queueName, Exchange: exchange, Key: key, Mqurl: MQURL}
```

```
}

//断开channel 和 connection
func (r *RabbitMQ) Destory() {
    r.channel.Close()
    r.conn.Close()
}

//错误处理函数
func (r *RabbitMQ) failOnErr(err error, message string) {
    if err != nil {
        log.Fatalf("%s:%s", message, err)
        panic(fmt.Sprintf("%s:%s", message, err))
    }
}

//创建简单模式下RabbitMQ实例
func NewRabbitMQSimple(queueName string) *RabbitMQ {
    //创建RabbitMQ实例
    rabbitmq := NewRabbitMQ(queueName, "", "")
    var err error
    //获取connection
    rabbitmq.conn, err = amqp.Dial(rabbitmq.Mqurl)
    rabbitmq.failOnErr(err, "failed to connect rabb"+
        "itmq!")
    //获取channel
    rabbitmq.channel, err = rabbitmq.conn.Channel()
    rabbitmq.failOnErr(err, "failed to open a channel")
    return rabbitmq
}

//直接模式队列生产
func (r *RabbitMQ) PublishSimple(message string) {
    //1.申请队列，如果队列不存在会自动创建，存在则跳过创建
    _, err := r.channel.QueueDeclare(
        r.QueueName,
        //是否持久化
        false,
        //是否自动删除
        false,
        //是否具有排他性
        false,
        //是否阻塞处理
        false,
        //额外的属性
        nil,
```

```
	)
	if err != nil {
		fmt.Println(err)
	}
	//调用channel 发送消息到队列中
	r.channel.Publish(
		r.Exchange,
		r.QueueName,
		//如果为true，根据自身exchange类型和routekey规则无法找到符合条件的队列会
把消息返还给发送者
		false,
		//如果为true，当exchange发送消息到队列后发现队列上没有消费者，则会把消息
返还给发送者
		false,
		amqp.Publishing{
			ContentType: "text/plain",
			Body:        []byte(message),
		})
}

//simple 模式下消费者
func (r *RabbitMQ) ConsumeSimple() {
	//1.申请队列，如果队列不存在会自动创建，存在则跳过创建
	q, err := r.channel.QueueDeclare(
		r.QueueName,
		//是否持久化
		false,
		//是否自动删除
		false,
		//是否具有排他性
		false,
		//是否阻塞处理
		false,
		//额外的属性
		nil,
	)
	if err != nil {
		fmt.Println(err)
	}

	//接收消息
	msgs, err := r.channel.Consume(
		q.Name, // queue
		//用来区分多个消费者
		"", // consumer
		//是否自动应答
```

```
		true,  // auto-ack
		//是否独有
		false, // exclusive
		//设置为true，表示 不能将同一个Conenction中生产者发送的消息传递给这个Connection中 的消费者
		false, // no-local
		//列是否阻塞
		false, // no-wait
		nil,   // args
	)
	if err != nil {
		fmt.Println(err)
	}

	forever := make(chan bool)
	//启用协程处理消息
	go func() {
		for d := range msgs {
			//消息逻辑处理，可以自行设计逻辑
			log.Printf("Received a message: %s", d.Body)

		}
	}()

	log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
	<-forever

}
```

mainSimlpePublish.go代码

```
package main

import (
	"fmt"
	"strconv"
	"time"

	"github.com/student/kuteng-RabbitMQ/RabbitMQ"
)

func main() {
	rabbitmq := RabbitMQ.NewRabbitMQSimple("" +
		"kuteng")
```

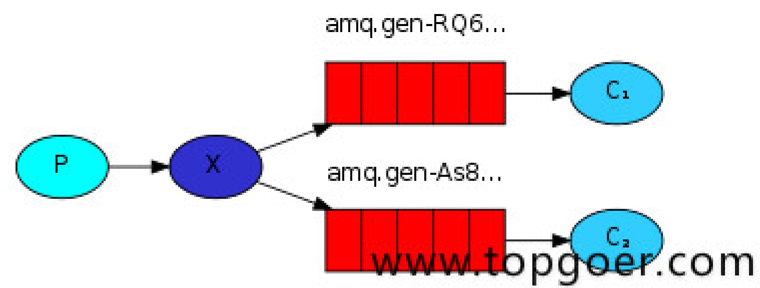
```
    for i := 0; i <= 100; i++ {
        rabbitmq.PublishSimple("Hello kuteng!" + strconv.Itoa(i))
        time.Sleep(1 * time.Second)
        fmt.Println(i)
    }
}
```

mainSimpleRecieve.go代码(两个消费端的代码是一样的)

```
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
    rabbitmq := RabbitMQ.NewRabbitMQSimple("" +
        "kuteng")
    rabbitmq.ConsumeSimple()
}
```

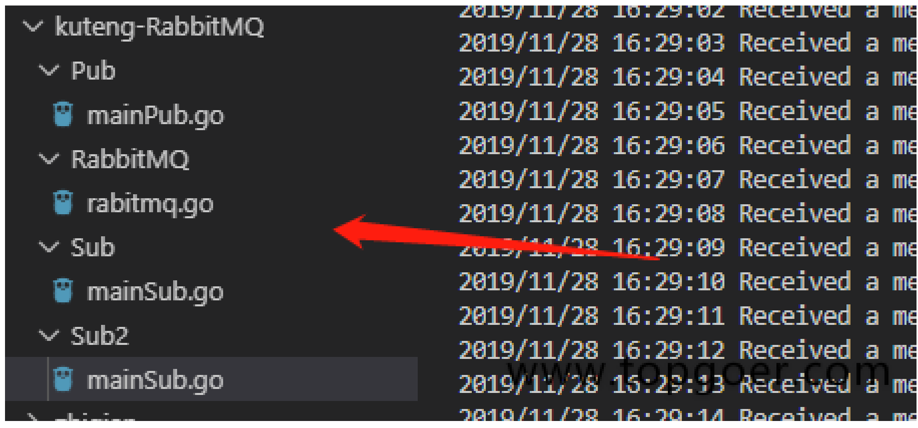
Publish模式

Publish模式（订阅模式，消息被路由投递给多个队列，一个消息被多个消费者获取）



- X代表交换机rabbitMQ内部组件,erlang 消息产生者是代码完成,代码的执行效率不高,消息产生者将消息放入交换机,交换机发布订阅把消息发送到所有消息队列中,对应消息队列的消费者拿到消息进行消费
- 相关场景:邮件群发,群聊天,广播(广告)

目录结构



kuteng-RabbitMQ

-RabbitMQ

–rabitmq.go //这个是RabbitMQ的封装

-Pub

–mainPub.go //Publish 先启动

-Sub

–mainSub.go

-Sub2

–mainSub.go

rabitmq.go代码

```
package RabbitMQ

import (
	"fmt"
	"log"

	"github.com/streadway/amqp"
)

//连接信息
const MQURL = "amqp://kuteng:kuteng@127.0.0.1:5672/kuteng"

//rabbitMQ结构体
type RabbitMQ struct {
	conn    *amqp.Connection
	channel *amqp.Channel
	//队列名称
	QueueName string
	//交换机名称
	Exchange string
	//bind Key 名称
	Key string
	//连接信息
	Mqurl string
}

//创建结构体实例
func NewRabbitMQ(queueName string, exchange string, key string) *RabbitMQ {
	return &RabbitMQ{QueueName: queueName, Exchange: exchange, Key: key, Mqurl: MQURL}
}

//断开channel 和 connection
func (r *RabbitMQ) Destory() {
	r.channel.Close()
	r.conn.Close()
```

```
}

//错误处理函数
func (r *RabbitMQ) failOnErr(err error, message string) {
	if err != nil {
		log.Fatalf("%s:%s", message, err)
		panic(fmt.Sprintf("%s:%s", message, err))
	}
}

//订阅模式创建RabbitMQ实例
func NewRabbitMQPubSub(exchangeName string) *RabbitMQ {
	//创建RabbitMQ实例
	rabbitmq := NewRabbitMQ("", exchangeName, "")
	var err error
	//获取connection
	rabbitmq.conn, err = amqp.Dial(rabbitmq.Mqurl)
	rabbitmq.failOnErr(err, "failed to connect rabbitmq!")
	//获取channel
	rabbitmq.channel, err = rabbitmq.conn.Channel()
	rabbitmq.failOnErr(err, "failed to open a channel")
	return rabbitmq
}

//订阅模式生产
func (r *RabbitMQ) PublishPub(message string) {
	//1.尝试创建交换机
	err := r.channel.ExchangeDeclare(
		r.Exchange,
		"fanout",
		true,
		false,
		//true表示这个exchange不可以被client用来推送消息，仅用来进行exchange和exchange之间的绑定
		false,
		false,
		nil,
	)

	r.failOnErr(err, "Failed to declare an excha"+
		"nge")

	//2.发送消息
	err = r.channel.Publish(
		r.Exchange,
		"",
```

```
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//订阅模式消费端代码
func (r *RabbitMQ) RecieveSub() {
    //1.试探性创建交换机
    err := r.channel.ExchangeDeclare(
        r.Exchange,
        //交换机类型
        "fanout",
        true,
        false,
        //YES表示这个exchange不可以被client用来推送消息，仅用来进行exchange和exc
hange之间的绑定
        false,
        false,
        nil,
    )
    r.failOnErr(err, "Failed to declare an exch"+
        "ange")
    //2.试探性创建队列，这里注意队列名称不要写
    q, err := r.channel.QueueDeclare(
        "", //随机生产队列名称
        false,
        false,
        true,
        false,
        nil,
    )
    r.failOnErr(err, "Failed to declare a queue")

    //绑定队列到 exchange 中
    err = r.channel.QueueBind(
        q.Name,
        //在pub/sub模式下，这里的key要为空
        "",
        r.Exchange,
        false,
        nil)

    //消费消息
```

```go
	messges, err := r.channel.Consume(
		q.Name,
		"",
		true,
		false,
		false,
		false,
		nil,
	)

	forever := make(chan bool)

	go func() {
		for d := range messges {
			log.Printf("Received a message: %s", d.Body)
		}
	}()
	fmt.Println("退出请按 CTRL+C\n")
	<-forever
}
```

mainPub.go代码

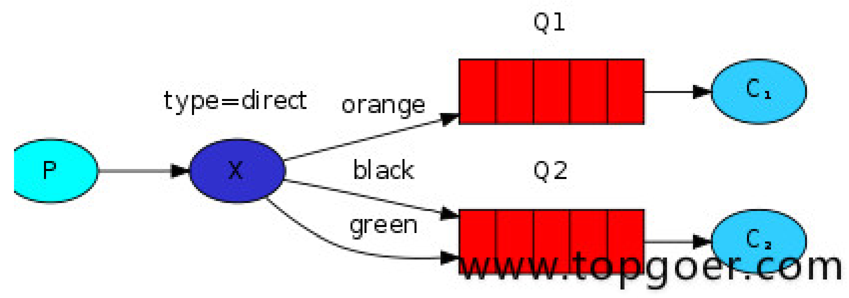
```go
package main

import (
	"fmt"
	"strconv"
	"time"

	"github.com/student/kuteng-RabbitMQ/RabbitMQ"
)

func main() {
	rabbitmq := RabbitMQ.NewRabbitMQPubSub("" +
		"newProduct")
	for i := 0; i < 100; i++ {
		rabbitmq.PublishPub("订阅模式生产第" +
			strconv.Itoa(i) + "条" + "数据")
		fmt.Println("订阅模式生产第" +
			strconv.Itoa(i) + "条" + "数据")
		time.Sleep(1 * time.Second)
	}

}
```

mainSub.go代码(两个消费者代码是一样的)

```
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
    rabbitmq := RabbitMQ.NewRabbitMQPubSub("" +
        "newProduct")
    rabbitmq.RecieveSub()
}
```

Routing模式

Routing模式(路由模式，一个消息被多个消费者获取，并且消息的目标队列可被生产者指定)



- 消息生产者将消息发送给交换机按照路由判断,路由是字符串(info) 当前产生的消息携带路由字符(对象的方法),交换机根据路由的key,只能匹配上路由key对应的消息队列,对应的消费者才能消费消息;
- 根据业务功能定义路由字符串
- 从系统的代码逻辑中获取对应的功能字符串,将消息任务扔到对应的队列中业务场景:error 通知;EXCEPTION;错误通知的功能;传统意义的错误通知;客户通知;利用key路由,可以将程序中的错误封装成消息传入到消息队列中,开发者可以自定义消费者,实时接收错误;

目录结构

kuteng-RabbitMQ

-RabbitMQ

–rabitmq.go //这个是RabbitMQ的封装

-publish

–mainpublish.go //Publish 先启动

-recieve1

–mainrecieve.go

-recieve2

–mainrecieve.go

rabitmq.go代码

```
package RabbitMQ

import (
	"fmt"
	"log"

	"github.com/streadway/amqp"
)

//连接信息
const MQURL = "amqp://kuteng:kuteng@127.0.0.1:5672/kuteng"

//rabbitMQ结构体
type RabbitMQ struct {
	conn    *amqp.Connection
	channel *amqp.Channel
	//队列名称
	QueueName string
	//交换机名称
	Exchange string
	//bind Key 名称
	Key string
	//连接信息
	Mqurl string
}

//创建结构体实例
func NewRabbitMQ(queueName string, exchange string, key string) *RabbitMQ {
	return &RabbitMQ{QueueName: queueName, Exchange: exchange, Key: key, Mqurl: MQURL}
}

//断开channel 和 connection
```

```
func (r *RabbitMQ) Destory() {
	r.channel.Close()
	r.conn.Close()
}

//错误处理函数
func (r *RabbitMQ) failOnErr(err error, message string) {
	if err != nil {
		log.Fatalf("%s:%s", message, err)
		panic(fmt.Sprintf("%s:%s", message, err))
	}
}

//路由模式
//创建RabbitMQ实例
func NewRabbitMQRouting(exchangeName string, routingKey string) *RabbitMQ {
	//创建RabbitMQ实例
	rabbitmq := NewRabbitMQ("", exchangeName, routingKey)
	var err error
	//获取connection
	rabbitmq.conn, err = amqp.Dial(rabbitmq.Mqurl)
	rabbitmq.failOnErr(err, "failed to connect rabbitmq!")
	//获取channel
	rabbitmq.channel, err = rabbitmq.conn.Channel()
	rabbitmq.failOnErr(err, "failed to open a channel")
	return rabbitmq
}

//路由模式发送消息
func (r *RabbitMQ) PublishRouting(message string) {
	//1.尝试创建交换机
	err := r.channel.ExchangeDeclare(
		r.Exchange,
		//要改成direct
		"direct",
		true,
		false,
		false,
		false,
		nil,
	)

	r.failOnErr(err, "Failed to declare an excha"+
		"nge")

	//2.发送消息
```

```
    err = r.channel.Publish(
        r.Exchange,
        //要设置
        r.Key,
        false,
        false,
        amqp.Publishing{
            ContentType: "text/plain",
            Body:        []byte(message),
        })
}

//路由模式接受消息
func (r *RabbitMQ) RecieveRouting() {
    //1.试探性创建交换机
    err := r.channel.ExchangeDeclare(
        r.Exchange,
        //交换机类型
        "direct",
        true,
        false,
        false,
        false,
        nil,
    )
    r.failOnErr(err, "Failed to declare an exch"+
        "ange")
    //2.试探性创建队列，这里注意队列名称不要写
    q, err := r.channel.QueueDeclare(
        "", //随机生产队列名称
        false,
        false,
        true,
        false,
        nil,
    )
    r.failOnErr(err, "Failed to declare a queue")

    //绑定队列到 exchange 中
    err = r.channel.QueueBind(
        q.Name,
        //需要绑定key
        r.Key,
        r.Exchange,
        false,
        nil)
```

```
    //消费消息
    messges, err := r.channel.Consume(
        q.Name,
        "",
        true,
        false,
        false,
        false,
        nil,
    )

    forever := make(chan bool)

    go func() {
        for d := range messges {
            log.Printf("Received a message: %s", d.Body)
        }
    }()

    fmt.Println("退出请按 CTRL+C\n")
    <-forever
}
```

mainpublish.go代码

```
package main

import (
    "fmt"
    "strconv"
    "time"

    "github.com/student/kuteng-RabbitMQ/RabbitMQ"
)

func main() {
    kutengone := RabbitMQ.NewRabbitMQRouting("kuteng", "kuteng_one")
    kutengtwo := RabbitMQ.NewRabbitMQRouting("kuteng", "kuteng_two")
    for i := 0; i <= 100; i++ {
        kutengone.PublishRouting("Hello kuteng one!" + strconv.Itoa(i))
        kutengtwo.PublishRouting("Hello kuteng Two!" + strconv.Itoa(i))
        time.Sleep(1 * time.Second)
        fmt.Println(i)
    }
```

```
}
```

recieve1/mainrecieve.go代码

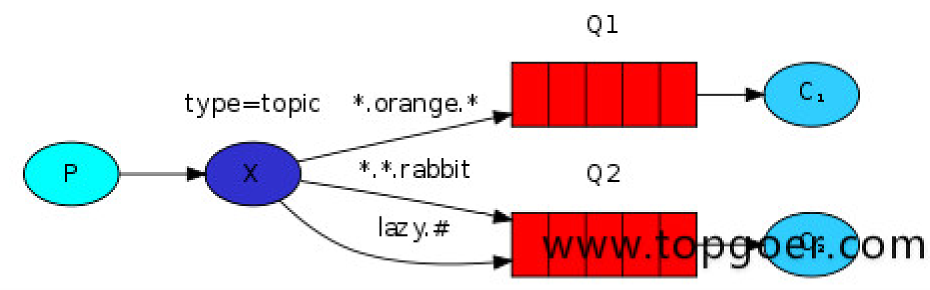
```go
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
	kutengone := RabbitMQ.NewRabbitMQRouting("kuteng", "kuteng_one")
	kutengone.RecieveRouting()
}
```

recieve2/mainrecieve.go代码

```go
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
	kutengtwo := RabbitMQ.NewRabbitMQRouting("kuteng", "kuteng_two")
	kutengtwo.RecieveRouting()
}
```

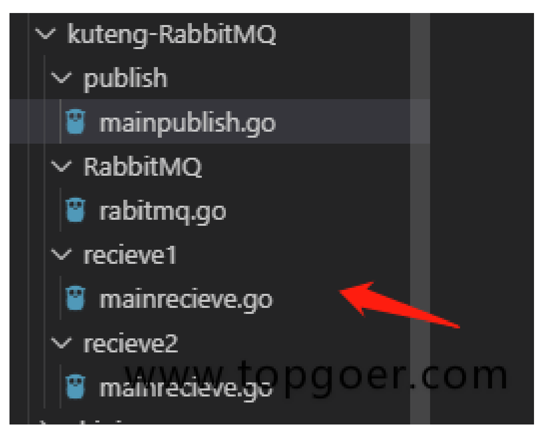
Topic模式

Topic模式（话题模式，一个消息被多个消费者获取，消息的目标queue可用BindingKey以通配符，（#：一个或多个词，`*`：一个词）的方式指定



- 星号井号代表通配符
- 星号代表多个单词,井号代表一个单词
- 路由功能添加模糊匹配
- 消息产生者产生消息,把消息交给交换机
- 交换机根据key的规则模糊匹配到对应的队列,由队列的监听消费者接收消息消费

目录结构



kuteng-RabbitMQ

-RabbitMQ

–rabitmq.go //这个是RabbitMQ的封装

-publish

–mainpublish.go //Publish 先启动

-recieve1

–mainrecieve.go

-recieve2

–mainrecieve.go

rabitmq.go代码

```
package RabbitMQ

import (
	"fmt"
	"log"

	"github.com/streadway/amqp"
)

//连接信息
const MQURL = "amqp://kuteng:kuteng@127.0.0.1:5672/kuteng"

//rabbitMQ结构体
type RabbitMQ struct {
	conn    *amqp.Connection
	channel *amqp.Channel
	//队列名称
	QueueName string
	//交换机名称
	Exchange string
	//bind Key 名称
	Key string
	//连接信息
	Mqurl string
}

//创建结构体实例
func NewRabbitMQ(queueName string, exchange string, key string) *RabbitMQ {
	return &RabbitMQ{QueueName: queueName, Exchange: exchange, Key: key, Mqurl:
 MQURL}
}
```

```
//断开channel 和 connection
func (r *RabbitMQ) Destory() {
	r.channel.Close()
	r.conn.Close()
}

//错误处理函数
func (r *RabbitMQ) failOnErr(err error, message string) {
	if err != nil {
		log.Fatalf("%s:%s", message, err)
		panic(fmt.Sprintf("%s:%s", message, err))
	}
}

//话题模式
//创建RabbitMQ实例
func NewRabbitMQTopic(exchangeName string, routingKey string) *RabbitMQ {
	//创建RabbitMQ实例
	rabbitmq := NewRabbitMQ("", exchangeName, routingKey)
	var err error
	//获取connection
	rabbitmq.conn, err = amqp.Dial(rabbitmq.Mqurl)
	rabbitmq.failOnErr(err, "failed to connect rabbitmq!")
	//获取channel
	rabbitmq.channel, err = rabbitmq.conn.Channel()
	rabbitmq.failOnErr(err, "failed to open a channel")
	return rabbitmq
}

//话题模式发送消息
func (r *RabbitMQ) PublishTopic(message string) {
	//1.尝试创建交换机
	err := r.channel.ExchangeDeclare(
		r.Exchange,
		//要改成topic
		"topic",
		true,
		false,
		false,
		false,
		nil,
	)

	r.failOnErr(err, "Failed to declare an excha"+
		"nge")
```

```
	//2.发送消息
	err = r.channel.Publish(
		r.Exchange,
		//要设置
		r.Key,
		false,
		false,
		amqp.Publishing{
			ContentType: "text/plain",
			Body:        []byte(message),
		})
}

//话题模式接受消息
//要注意key,规则
//其中“*”用于匹配一个单词，“#”用于匹配多个单词（可以是零个）
//匹配 kuteng.* 表示匹配 kuteng.hello, kuteng.hello.one需要用kuteng.#才能匹配到
func (r *RabbitMQ) RecieveTopic() {
	//1.试探性创建交换机
	err := r.channel.ExchangeDeclare(
		r.Exchange,
		//交换机类型
		"topic",
		true,
		false,
		false,
		false,
		nil,
	)
	r.failOnErr(err, "Failed to declare an exch"+
		"ange")
	//2.试探性创建队列，这里注意队列名称不要写
	q, err := r.channel.QueueDeclare(
		"", //随机生产队列名称
		false,
		false,
		true,
		false,
		nil,
	)
	r.failOnErr(err, "Failed to declare a queue")

	//绑定队列到 exchange 中
	err = r.channel.QueueBind(
		q.Name,
```

```
		//在pub/sub模式下，这里的key要为空
		r.Key,
		r.Exchange,
		false,
		nil)

	//消费消息
	messges, err := r.channel.Consume(
		q.Name,
		"",
		true,
		false,
		false,
		false,
		nil,
	)

	forever := make(chan bool)

	go func() {
		for d := range messges {
			log.Printf("Received a message: %s", d.Body)
		}
	}()

	fmt.Println("退出请按 CTRL+C\n")
	<-forever
}
```

mainpublish.go代码

```
package main

import (
	"fmt"
	"strconv"
	"time"

	"github.com/student/kuteng-RabbitMQ/RabbitMQ"
)

func main() {
	kutengOne := RabbitMQ.NewRabbitMQTopic("exKutengTopic", "kuteng.topic.one")
	kutengTwo := RabbitMQ.NewRabbitMQTopic("exKutengTopic", "kuteng.topic.two")
	for i := 0; i <= 100; i++ {
```

```
        kutengOne.PublishTopic("Hello kuteng topic one!" + strconv.Itoa(i))
        kutengTwo.PublishTopic("Hello kuteng topic Two!" + strconv.Itoa(i))
        time.Sleep(1 * time.Second)
        fmt.Println(i)
    }

}
```

recieve1/mainrecieve.go代码

```
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
    kutengOne := RabbitMQ.NewRabbitMQTopic("exKutengTopic", "#")
    kutengOne.RecieveTopic()
}
```

recieve2/mainrecieve.go代码

```
package main

import "github.com/student/kuteng-RabbitMQ/RabbitMQ"

func main() {
    kutengOne := RabbitMQ.NewRabbitMQTopic("exKutengTopic", "kuteng.*.two")
    kutengOne.RecieveTopic()
}
```

go操作ElasticSearch

ElasticSearch介绍

Elasticsearch安装

Kibana安装

操作ElasticSearch

ElasticSearch介绍

介绍

Elasticsearch（ES）是一个基于Lucene构建的开源、分布式、RESTful接口的全文搜索引擎。Elasticsearch还是一个分布式文档数据库，其中每个字段均可被索引，而且每个字段的数据均可被搜索，ES能够横向扩展至数以百计的服务器存储以及处理PB级的数据。可以在极短的时间内存储、搜索和分析大量的数据。通常作为具有复杂搜索场景情况下的核心发动机。

Elasticsearch能做什么

- 当你经营一家网上商店，你可以让你的客户搜索你卖的商品。在这种情况下，你可以使用ElasticSearch来存储你的整个产品目录和库存信息，为客户提供精准搜索，可以为客户推荐相关商品。

- 当你想收集日志或者交易数据的时候，需要分析和挖掘这些数据，寻找趋势，进行统计，总结，或发现异常。在这种情况下，你可以使用Logstash或者其他工具来进行收集数据，当这引起数据存储到ElasticsSearch中。你可以搜索和汇总这些数据，找到任何你感兴趣的信息。

- 对于程序员来说，比较有名的案例是GitHub，GitHub的搜索是基于ElasticSearch构建的，在github.com `/search` 页面，你可以搜索项目、用户、issue、pull request，还有代码。共有 `40~50` 个索引库，分别用于索引网站需要跟踪的各种数据。虽然只索引项目的主分支（master），但这个数据量依然巨大，包括20亿个索引文档，30TB的索引文件。

Elasticsearch基本概念

Near Realtime(NRT) 几乎实时

Elasticsearch是一个几乎实时的搜索平台。意思是，从索引一个文档到这个文档可被搜索只需要一点点的延迟，这个时间一般为毫秒级。

Cluster 集群

群集是一个或多个节点（服务器）的集合， 这些节点共同保存整个数据，并在所有节点上提供联合索引和搜索功能。一个集群由一个唯一集群ID确定，并指定一个集群名（默认为

“elasticsearch”）。该集群名非常重要，因为节点可以通过这个集群名加入群集，一个节点只能是群集的一部分。

确保在不同的环境中不要使用相同的群集名称，否则可能会导致连接错误的群集节点。例如，你可以使用logging-dev、logging-stage、logging-prod分别为开发、阶段产品、生产集群做记录。

Node节点

节点是单个服务器实例，它是群集的一部分，可以存储数据，并参与群集的索引和搜索功能。就像一个集群，节点的名称默认为一个随机的通用唯一标识符（UUID），确定在启动时分配给该节点。如果不希望默认，可以定义任何节点名。这个名字对管理很重要，目的是要确定你的网络服务器对应于你的ElasticSearch群集节点。

我们可以通过群集名配置节点以连接特定的群集。默认情况下，每个节点设置加入名为“elasticSearch”的集群。这意味着如果你启动多个节点在网络上，假设他们能发现彼此都会自动形成和加入一个名为“elasticsearch”的集群。

在单个群集中，你可以拥有尽可能多的节点。此外，如果“elasticsearch”在同一个网络中，没有其他节点正在运行，从单个节点的默认情况下会形成一个新的单节点名为”elasticsearch”的集群。

Index索引

索引是具有相似特性的文档集合。例如，可以为客户数据提供索引，为产品目录建立另一个索引，以及为订单数据建立另一个索引。索引由名称（必须全部为小写）标识，该名称用于在对其中的文档执行索引、搜索、更新和删除操作时引用索引。在单个群集中，你可以定义尽可能多的索引。

Type类型

在索引中，可以定义一个或多个类型。类型是索引的逻辑类别/分区，其语义完全取决于你。一般来说，类型定义为具有公共字段集的文档。例如，假设你运行一个博客平台，并将所有数据存储在一个索引中。在这个索引中，你可以为用户数据定义一种类型，为博客数据定义另一种类型，以及为注释数据定义另一类型。

Document文档

文档是可以被索引的信息的基本单位。例如，你可以为单个客户提供一个文档，单个产品提供另一个文档，以及单个订单提供另一个文档。本文件的表示形式为JSON（JavaScript Object Notation）格式，这是一种非常普遍的互联网数据交换格式。

在索引/类型中，你可以存储尽可能多的文档。请注意，尽管文档物理驻留在索引中，文档实际上必须索引或分配到索引中的类型。

Shards & Replicas分片与副本

索引可以存储大量的数据，这些数据可能超过单个节点的硬件限制。例如，十亿个文件占用磁盘空间1TB的单指标可能不适合对单个节点的磁盘或可能太慢服务仅从单个节点的搜索请求。

为了解决这一问题，Elasticsearch提供细分你的指标分成多个块称为分片的能力。当你创建一个索引，你可以简单地定义你想要的分片数量。每个分片本身是一个全功能的、独立的“指数”，可以托管在集群中的任何节点。

Shards分片的重要性主要体现在以下两个特征：

1.副本为分片或节点失败提供了高可用性。为此，需要注意的是，一个副本的分片不会分配在同一个节点作为原始的或主分片，副本是从主分片那里复制过来的。

2.副本允许用户扩展你的搜索量或吞吐量，因为搜索可以在所有副本上并行执行。

ES基本概念与关系型数据库的比较

ES概念	关系型数据库
Index（索引）支持全文检索	Database（数据库）
Type（类型）	Table（表）
Document（文档），不同文档可以有不同的字段集合	Row（数据行）
Field（字段）	Column（数据列）
Mapping（映射）	Schema（模式）

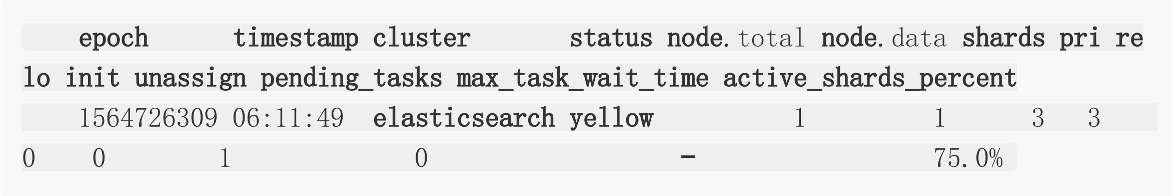
ES API

以下示例使用curl演示。

查看健康状态

```
curl -X GET 127.0.0.1:9200/_cat/health?v
```

输出：



```
epoch      timestamp cluster       status node.total node.data shards pri relo init unassign pending_tasks max_task_wait_time active_shards_percent
1564726309 06:11:49  elasticsearch yellow          1         1      3   3    0    0        1             0                  -                 75.0%
```

查询当前es集群中所有的indices

```
curl -X GET 127.0.0.1:9200/_cat/indices?v
```

输出：

```
health status index                uuid                   pri rep docs.count docs.deleted store.size pri.store.size
green  open   .kibana_task_manager LUo-IxjDQdWeAbR-SYuYvQ   1   0          2            0     45.5kb         45.5kb
green  open   .kibana_1            PLvyZV1bRDWex05xkOrNNg   1   0          4            1     23.9kb         23.9kb
yellow open   user                 o42mIpDeSgSWZ6eARWUfKw   1   1          0            0       283b           283b
```

创建索引

```
curl -X PUT 127.0.0.1:9200/www
```

输出：

```
{"acknowledged":true,"shards_acknowledged":true,"index":"www"}
```

删除索引

```
curl -X DELETE 127.0.0.1:9200/www
```

输出：

```
{"acknowledged":true}
```

插入记录

```
curl -H "ContentType:application/json" -X POST 127.0.0.1:9200/user/person -d '
{
	"name": "LMH",
	"age": 18,
	"married": true
}'
```

输出：

```
{
    "_index": "user",
    "_type": "person",
    "_id": "MLcwUWwBvEa8j5UrLZj4",
    "_version": 1,
    "result": "created",
    "_shards": {
        "total": 2,
        "successful": 1,
        "failed": 0
    },
    "_seq_no": 3,
    "_primary_term": 1
}
```

也可以使用PUT方法，但是需要传入id

```
    curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "name": "LMH",
        "age": 18,
        "married": false
    }'
```

检索

Elasticsearch的检索语法比较特别，使用GET方法携带JSON格式的查询条件。

全检索：

```
    curl -X GET 127.0.0.1:9200/user/person/_search
```

按条件检索：

```
    curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "query":{
            "match": {"name": "LMH"}
        }
    }'
```

ElasticSearch默认一次最多返回10条结果，可以像下面的示例通过size字段来设置返回结果的数目。

```
    curl -H "ContentType:application/json" -X PUT 127.0.0.1:9200/user/person/4 -d '
    {
        "query":{
            "match": {"name": "LMH"},
            "size": 2
        }
    }'
```

Elasticsearch安装

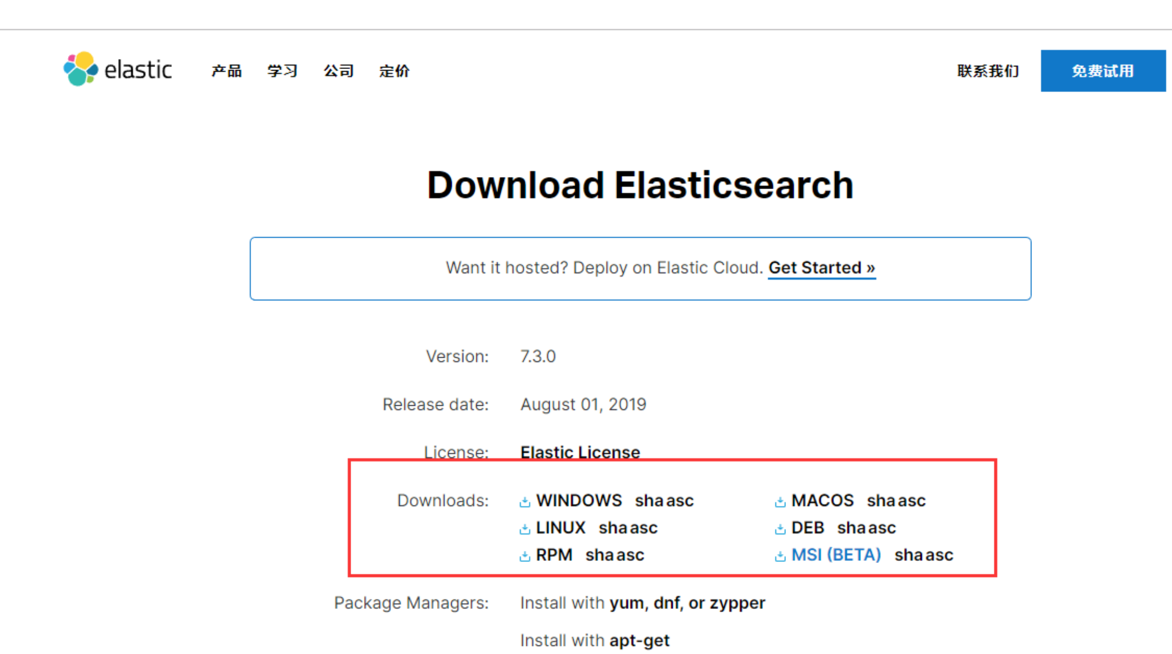
Elasticsearch官网：https://www.elastic.co/cn/products/elasticsearch

Elasticsearch介绍

Elasticsearch（ES）是一个基于Lucene构建的开源、分布式、RESTful接口的全文搜索引擎。Elasticsearch还是一个分布式文档数据库，其中每个字段均可被索引，而且每个字段的数据均可被搜索，ES能够横向扩展至数以百计的服务器存储以及处理PB级的数据。可以在极短的时间内存储、搜索和分析大量的数据。通常作为具有复杂搜索场景情况下的核心发动机。

下载

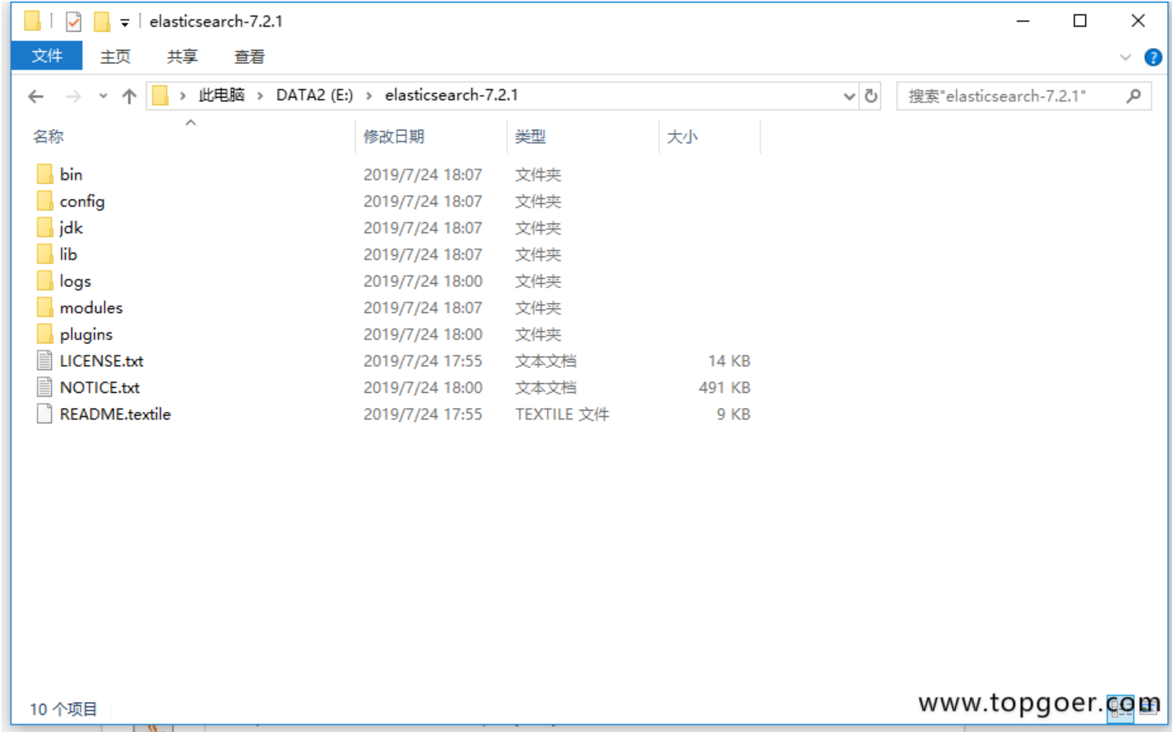
官方网站下载链接：https://www.elastic.co/cn/downloads/elasticsearch



请根据自己的需求下载对应的版本。

安装

将上一步下载的压缩包解压，下图以Windows为例。



启动

执行bin\elasticsearch.bat启动，默认在本机的9200端口启动服务。

使用浏览器访问elasticsearch服务，可以看到类似下面的信息。

Kibana安装

Kibana介绍

官网链接：https://www.elastic.co/cn/products/kibana

Kibana是一个开源的分析和可视化平台，设计用于和Elasticsearch一起工作。

你可以使用Kibana来搜索、查看、并和存储在Elasticsearch索引中的数据进行交互。

你可以轻松地执行高级数据分析，并且以各种图标、表格和地图的形式可视化数据。

Kibana使得理解大量数据变得很容易。它简单的、基于浏览器的界面使你能够快速创建和共享动态仪表板，实时显示Elasticsearch查询的变化。

下载

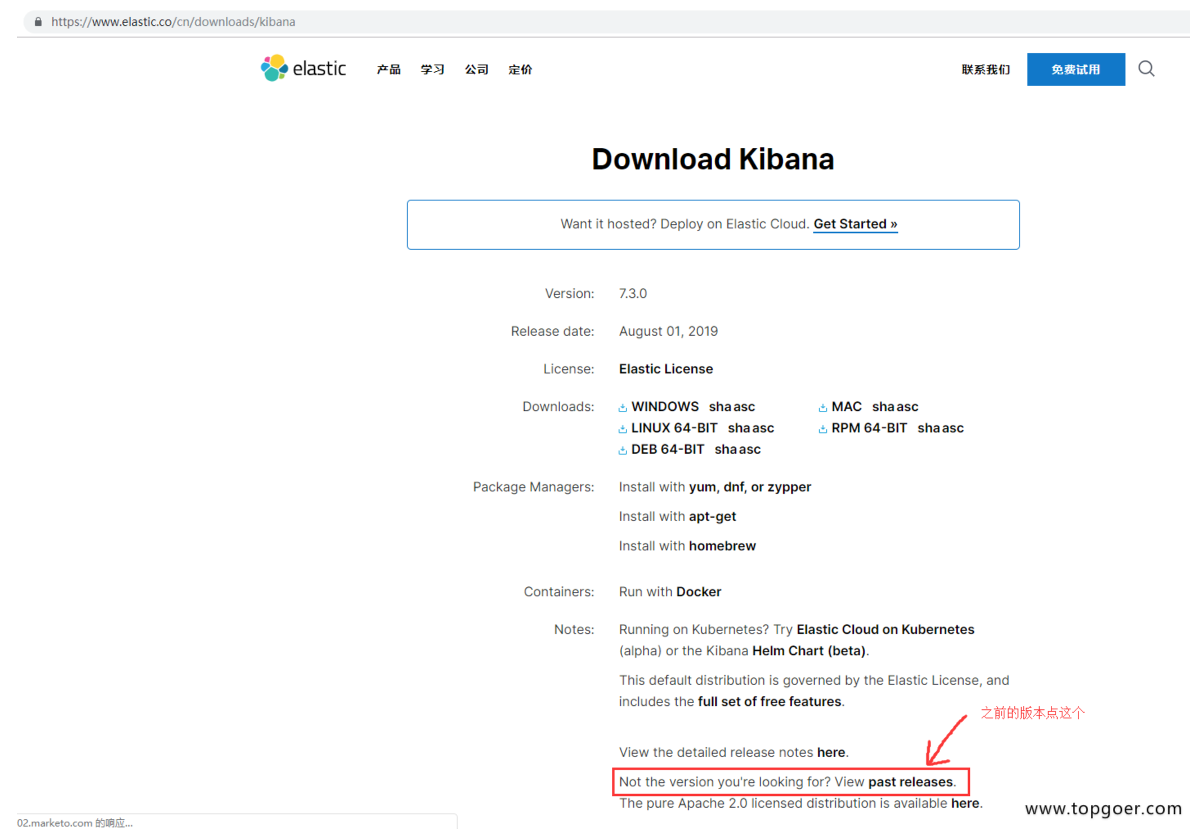
官方下载链接：https://www.elastic.co/cn/downloads/kibana

请根据需求下载对应的版本。

注意：

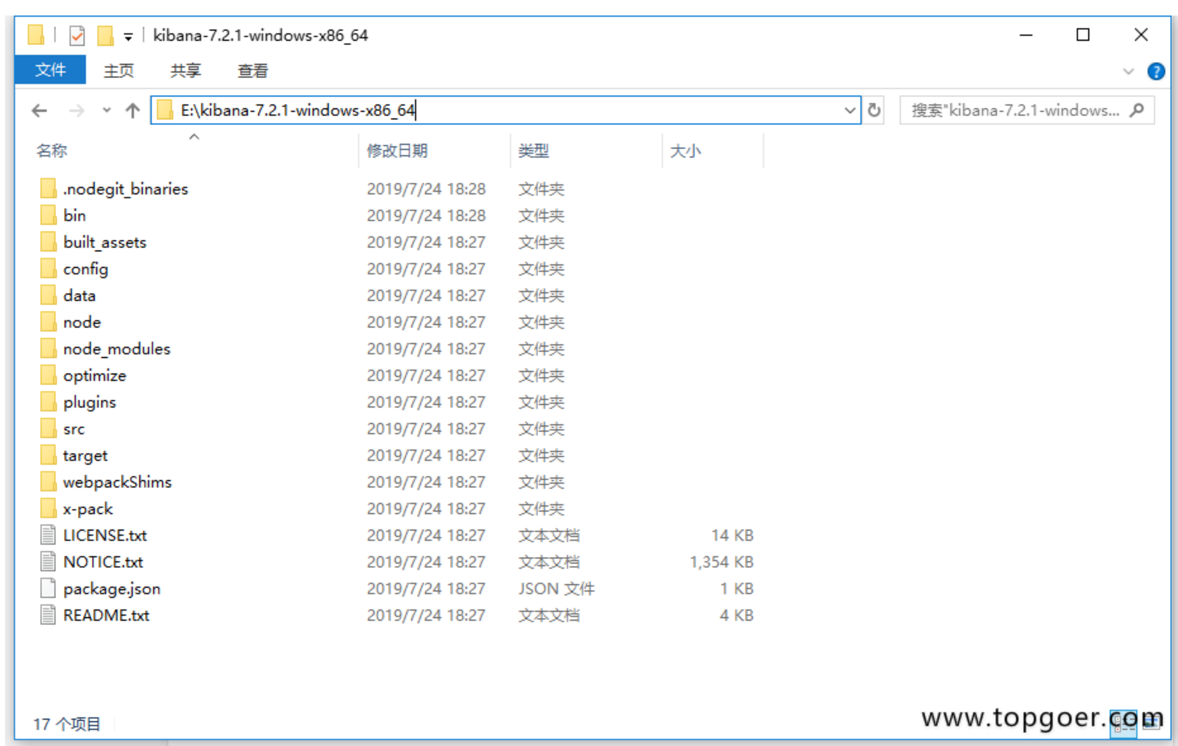
Kibana与Elasticsearch的版本要相互对应，否则可能不兼容！！！

例如：Elasticsearch是7.2.1的版本，那么你的Kibana也要下载7.2.1的版本。



安装

将上一步下载得到的文件解压。



修改config目录下的配置文件kibana.yml（如你是本机没有发生改变可以省略这一步）

将配置文件中 elasticsearch.hosts设置为你elasticseatch的地址，例如：

```
6
7 # The URLs of the Elasticsearch instances to use for
8 elasticsearch.hosts: ["http://localhost:9200"]
9
0 # When this setting's value is true Kibana uses the h
1 # setting. When the value of this setting is false, K
2 # that connects to this Kibana instance.
3 #elasticsearch.preserveHost: true
```
www.topgoer.com

（找不到直接Ctrl+F搜索‘url’）

然后翻到最后修改一下语言，配置成简体中文。

```
112  # Specifies locale to be used for all localizable strings, dates and number formats.
113  i18n.locale: "zh-CN"
114
```
www.topgoer.com

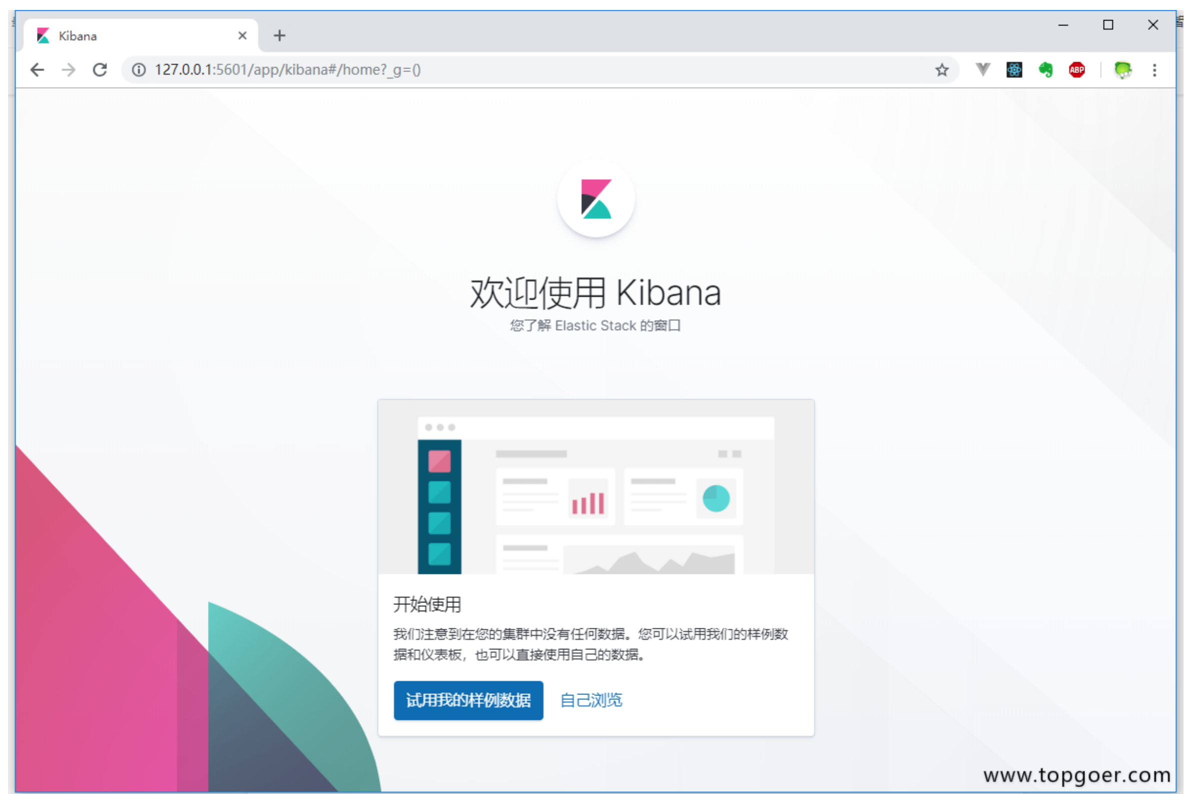
启动

执行bin\kibana.bat启动

启动过程比较慢，请耐心等待出现类似下图界面，就表示启动成功。

```
C:\Windows\System32\cmd.exe - bin\kibana.bat
  log   [03:24:04.286] [info][status][plugin:tilemap@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.287] [info][status][plugin:watcher@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.289] [info][status][plugin:grokdebugger@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.290] [info][status][plugin:logstash@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.292] [info][status][plugin:beats_management@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.293] [info][status][plugin:index_management@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.294] [info][status][plugin:index_lifecycle_management@7.2.1] Status changed from yellow to green - Rea
dy
  log   [03:24:04.296] [info][status][plugin:rollup@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.296] [info][status][plugin:remote_clusters@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.297] [info][status][plugin:cross_cluster_replication@7.2.1] Status changed from yellow to green - Read
y
  log   [03:24:04.298] [info][status][plugin:snapshot_restore@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:04.299] [info][kibana-monitoring][monitoring] Starting monitoring stats collection
  log   [03:24:04.338] [info][status][plugin:maps@7.2.1] Status changed from yellow to green - Ready
  log   [03:24:05.823] [warning][reporting] Generating a random key for xpack.reporting.encryptionKey. To prevent pendin
g reports from failing on restart, please set xpack.reporting.encryptionKey in kibana.yml
  log   [03:24:05.835] [info][status][plugin:reporting@7.2.1] Status changed from uninitialized to green - Ready
  log   [03:24:05.882] [warning][task_manager] This Kibana instance defines an older template version (7020199) than is
currently in Elasticsearch (7030099). Because of the potential for non-backwards compatible changes, this Kibana instanc
e will only be able to claim scheduled tasks with "kibana.apiVersion" <= 1 in the task metadata.
  log   [03:24:05.892] [warning][task_manager] This Kibana instance defines an older template version (7020199) than is
currently in Elasticsearch (7030099). Because of the potential for non-backwards compatible changes, this Kibana instanc
e will only be able to claim scheduled tasks with "kibana.apiVersion" <= 1 in the task metadata.
  log   [03:24:06.828] [warning][task_manager] This Kibana instance defines an older template version (7020199) than is
currently in Elasticsearch (7030099). Because of the potential for non-backwards compatible changes, this Kibana instanc
e will only be able to claim scheduled tasks with "kibana.apiVersion" <= 1 in the task metadata.
  log   [03:24:06.991] [info][listening] Server running at http://localhost:5601
  log   [03:24:07.011] [info][status][plugin:spaces@7.2.1] Status changed from yellow to green - Ready
```

使用浏览器访问本机的5601端口即可看到类似下面的界面：


Kibana
127.0.0.1:5601/app/kibana#/home?_g=()
欢迎使用 Kibana
您了解 Elastic Stack 的窗口
开始使用
我们注意到在您的集群中没有任何数据。您可以试用我们的样例数据和仪表板，也可以直接使用自己的数据。
试用我的样例数据
自己浏览
www.topgoer.com

操作ElasticSearch

elastic client

我们使用第三方库https://github.com/olivere/elastic 来连接ES并进行操作。

注意下载与你的ES相同版本的client，例如我们这里使用的ES是7.2.1的版本，那么我们下载的client也要与之对应为github.com/olivere/elastic/v7。

使用go.mod来管理依赖：

```
    require (
        github.com/olivere/elastic/v7 v7.0.4
    )
```

简单示例：

```
package main

import (
    "context"
    "fmt"

    "github.com/olivere/elastic/v7"
)

// Elasticsearch demo

type Person struct {
    Name    string `json:"name"`
    Age     int    `json:"age"`
    Married bool   `json:"married"`
}

func main() {
    client, err := elastic.NewClient(elastic.SetURL("http://127.0.0.1:9200"))
    if err != nil {
        // Handle error
        panic(err)
    }

    fmt.Println("connect to es success")
    p1 := Person{Name: "lmh", Age: 18, Married: false}
```

```
    put1, err := client.Index().
        Index("user").
        BodyJson(p1).
        Do(context.Background())
    if err != nil {
        // Handle error
        panic(err)
    }
    fmt.Printf("Indexed user %s to index %s, type %s\n", put1.Id, put1.Index, pu
t1.Type)
}
```

示例2：

```
package main

import (
    "context"
    "fmt"
    "log"
    "os"
    "reflect"

    "gopkg.in/olivere/elastic.v7" //这里使用的是版本5，最新的是6，有改动
)

var client *elastic.Client
var host = "http://127.0.0.1:9200/"

type Employee struct {
    FirstName string   `json:"first_name"`
    LastName  string   `json:"last_name"`
    Age       int      `json:"age"`
    About     string   `json:"about"`
    Interests []string `json:"interests"`
}

//初始化
func init() {
    errorlog := log.New(os.Stdout, "APP", log.LstdFlags)
    var err error
    client, err = elastic.NewClient(elastic.SetErrorLog(errorlog), elastic.SetUR
L(host))
    if err != nil {
        panic(err)
```

```
    }
    info, code, err := client.Ping(host).Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Elasticsearch returned with code %d and version %s\n", code, inf
o.Version.Number)

    esversion, err := client.ElasticsearchVersion(host)
    if err != nil {
        panic(err)
    }
    fmt.Printf("Elasticsearch version %s\n", esversion)

}

/*下面是简单的CURD*/

//创建
func create() {

    //使用结构体
    e1 := Employee{"Jane", "Smith", 32, "I like to collect rock albums", []strin
g{"music"}}
    put1, err := client.Index().
        Index("megacorp").
        Type("employee").
        Id("1").
        BodyJson(e1).
        Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Indexed tweet %s to index s%s, type %s\n", put1.Id, put1.Index,
 put1.Type)

    //使用字符串
    e2 := `{"first_name":"John","last_name":"Smith","age":25,"about":"I love to
 go rock climbing","interests":["sports","music"]}`
    put2, err := client.Index().
        Index("megacorp").
        Type("employee").
        Id("2").
        BodyJson(e2).
        Do(context.Background())
    if err != nil {
```

```
        panic(err)
    }
    fmt.Printf("Indexed tweet %s to index s%s, type %s\n", put2.Id, put2.Index,
 put2.Type)

    e3 := `{"first_name":"Douglas","last_name":"Fir","age":35,"about":"I like to
build cabinets","interests":["forestry"]}`
    put3, err := client.Index().
        Index("megacorp").
        Type("employee").
        Id("3").
        BodyJson(e3).
        Do(context.Background())
    if err != nil {
        panic(err)
    }
    fmt.Printf("Indexed tweet %s to index s%s, type %s\n", put3.Id, put3.Index,
 put3.Type)

}

//删除
func delete() {

    res, err := client.Delete().Index("megacorp").
        Type("employee").
        Id("1").
        Do(context.Background())
    if err != nil {
        println(err.Error())
        return
    }
    fmt.Printf("delete result %s\n", res.Result)
}

//修改
func update() {
    res, err := client.Update().
        Index("megacorp").
        Type("employee").
        Id("2").
        Doc(map[string]interface{}{"age": 88}).
        Do(context.Background())
    if err != nil {
        println(err.Error())
    }
```

```
    fmt.Printf("update age %s\n", res.Result)

}

//查找
func gets() {
    //通过id查找
    get1, err := client.Get().Index("megacorp").Type("employee").Id("2").Do(context.Background())
    if err != nil {
        panic(err)
    }
    if get1.Found {
        fmt.Printf("Got document %s in version %d from index %s, type %s\n", get1.Id, get1.Version, get1.Index, get1.Type)
    }
}

//搜索
func query() {
    var res *elastic.SearchResult
    var err error
    //取所有
    res, err = client.Search("megacorp").Type("employee").Do(context.Background())
    printEmployee(res, err)

    //字段相等
    q := elastic.NewQueryStringQuery("last_name:Smith")
    res, err = client.Search("megacorp").Type("employee").Query(q).Do(context.Background())
    if err != nil {
        println(err.Error())
    }
    printEmployee(res, err)

    //条件查询
    //年龄大于30岁的
    boolQ := elastic.NewBoolQuery()
    boolQ.Must(elastic.NewMatchQuery("last_name", "smith"))
    boolQ.Filter(elastic.NewRangeQuery("age").Gt(30))
    res, err = client.Search("megacorp").Type("employee").Query(q).Do(context.Background())
    printEmployee(res, err)

    //短语搜索 搜索about字段中有 rock climbing
```

```
    matchPhraseQuery := elastic.NewMatchPhraseQuery("about", "rock climbing")
    res, err = client.Search("megacorp").Type("employee").Query(matchPhraseQuer
y).Do(context.Background())
    printEmployee(res, err)

    //分析 interests
    aggs := elastic.NewTermsAggregation().Field("interests")
    res, err = client.Search("megacorp").Type("employee").Aggregation("all_inter
ests", aggs).Do(context.Background())
    printEmployee(res, err)

}

//简单分页
func list(size, page int) {
    if size < 0 || page < 1 {
        fmt.Printf("param error")
        return
    }
    res, err := client.Search("megacorp").
        Type("employee").
        Size(size).
        From((page - 1) * size).
        Do(context.Background())
    printEmployee(res, err)

}

//打印查询到的Employee
func printEmployee(res *elastic.SearchResult, err error) {
    if err != nil {
        print(err.Error())
        return
    }
    var typ Employee
    for _, item := range res.Each(reflect.TypeOf(typ)) { //从搜索结果中取数据的
方法
        t := item.(Employee)
        fmt.Printf("%#v\n", t)
    }
}

func main() {
    create()
    delete()
    update()
```

```
    gets()
    query()
    list(1, 3)
}
```

更多使用详见文档：https://godoc.org/github.com/olivere/elastic

NSQ

安装

生产者

消费者

安装

Mac安装nsq：

按照安装文档中的说明进行操作。

打开终端：

执行：$ brew install nsq

若：-bash: brew: command not found

执行：$ ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"

然后执行：brew install nsq

在一个shell中，开始nsqlookupd：

$ nsqlookupd

在另一个shell中，开始nsqd：

$ nsqd –lookupd-tcp-address=127.0.0.1:4160

在另一个shell中，开始nsqadmin：

$ nsqadmin –lookupd-http-address=127.0.0.1:4161

发布初始消息（也在集群中创建主题）：

$ curl -d 'hello world 1' 'http://127.0.0.1:4151/pub?topic=test'

最后，在另一个shell中，开始nsq_to_file：

$ nsq_to_file –topic=test –output-dir=/tmp –lookupd-http-address=127.0.0.1:4161

发布更多消息nsqd：

$ curl -d 'hello world 2' 'http://127.0.0.1:4151/pub?topic=test'

$ curl -d 'hello world 3' 'http://127.0.0.1:4151/pub?topic=test'

验证事物按预期工作，在Web浏览器中打开http://127.0.0.1:4171/ 以查看nsqadminUI并查看统计信息。另外，检查 `test.*.log` 写入的日志文件（）的内容/tmp。

这里的重要教训是nsq_to_file（客户端）未明确告知test 主题产生的位置，它从中检索此信息，nsqlookupd并且尽管有连接的时间，但不会丢失任何消息。

生产者

运行Nsq服务集群

首先启动nsqlookud，在一个shell中，开始nsqlookupd：

$ nsqlookupd

在另一个shell中，开始nsqd：

$ nsqd –lookupd-tcp-address=127.0.0.1:4160

在另一个shell中，开始nsqadmin：

$ nsqadmin –lookupd-http-address=127.0.0.1:4161

发布初始消息（也在集群中创建主题）：

$ curl -d ‘hello world 1’ ‘http://127.0.0.1:4151/pub?topic=test'

最后，在另一个shell中，开始nsq_to_file：

$ nsq_to_file –topic=test –output-dir=/tmp –lookupd-http-address=127.0.0.1:4161

验证事物按预期工作，在Web浏览器中打开http://127.0.0.1:4171/ 以查看nsqadminUI并查看统计信息。另外，检查 `test.*.log` 写入的日志文件（）的内容/tmp。

链接nsq 并创建生产者：

```
package main

import (
    "fmt"

    nsq "github.com/nsqio/go-nsq"
)

func main() {
    // 定义nsq生产者
    var producer *nsq.Producer
    // 初始化生产者
    // producer, err := nsq.NewProducer("地址:端口", nsq.*Config )
    producer, err := nsq.NewProducer("127.0.0.1:4150", nsq.NewConfig())
    if err != nil {
        panic(err)
    }
```

```
    err = producer.Ping()
    if nil != err {
        // 关闭生产者
        producer.Stop()
        producer = nil
    }

    fmt.Println("ping nsq success")
}
```

生产者创建topic并写入nsq：

```
package main

import (
    "fmt"

    nsq "github.com/nsqio/go-nsq"
)

func main() {
    // 定义nsq生产者
    var producer *nsq.Producer
    // 初始化生产者
    // producer, err := nsq.NewProducer("地址:端口", nsq.*Config )
    producer, err := nsq.NewProducer("127.0.0.1:4150", nsq.NewConfig())
    if err != nil {
        panic(err)
    }

    err = producer.Ping()
    if nil != err {
        // 关闭生产者
        producer.Stop()
        producer = nil
    }

    // 生产者写入nsq,10条消息, topic = "test"
    topic := "test"
    for i := 0; i < 10; i++ {
        message := fmt.Sprintf("message:%d", i)
        if producer != nil && message != "" { //不能发布空串，否则会导致error
            err = producer.Publish(topic, []byte(message)) // 发布消息
            if err != nil {
```

```
                fmt.Printf("producer.Publish,err : %v", err)
            }
            fmt.Println(message)
        }
    }

    fmt.Println("producer.Publish success")

}
```

消费者

```go
//Nsq接收测试
package main

import (
	"fmt"
	"time"

	"github.com/nsqio/go-nsq"
)

// 消费者
type ConsumerT struct{}

// 主函数
func main() {
	InitConsumer("test", "test-channel", "127.0.0.1:4161")
	for {
		time.Sleep(time.Second * 10)
	}
}

//处理消息
func (*ConsumerT) HandleMessage(msg *nsq.Message) error {
	fmt.Println("receive", msg.NSQDAddress, "message:", string(msg.Body))
	return nil
}

//初始化消费者
func InitConsumer(topic string, channel string, address string) {
	cfg := nsq.NewConfig()
	cfg.LookupdPollInterval = time.Second          //设置重连时间
	c, err := nsq.NewConsumer(topic, channel, cfg) // 新建一个消费者
	if err != nil {
		panic(err)
	}
	c.SetLogger(nil, 0)        //屏蔽系统日志
	c.AddHandler(&ConsumerT{}) // 添加消费者接口

	//建立NSQLookupd连接
	if err := c.ConnectToNSQLookupd(address); err != nil {
		panic(err)
	}
```

```
    //建立多个nsqd连接
    // if err := c.ConnectToNSQDs([]string{"127.0.0.1:4150", "127.0.0.1:4152"}); err != nil {
    //  panic(err)
    // }

    // 建立一个nsqd连接
    // if err := c.ConnectToNSQD("127.0.0.1:4150"); err != nil {
    //  panic(err)
    // }
}
```

GORM

请跳转 http://wen.topgoer.com/docs/gorm/gorm-1c54sbcda16o6

xorm

请跳转 http://wen.topgoer.com/docs/xorm/xorm-1c5ch983t3sso

go操作memcached

go使用memcached需要第三方的驱动库，这里有一个库是memcached作者亲自实现的，代码质量效率肯定会有保障

安装

```
    go get github.com/bradfitz/gomemcache/memcache
```

使用

```
    import "github.com/bradfitz/gomemcache/memcache"
```

栗子(吃的那种)

```
package main

import (
    "fmt"
    "github.com/bradfitz/gomemcache/memcache"
)

var (
    server = "127.0.0.1:11211"
)

func main() {
    //create a handle
    mc := memcache.New(server)
    if mc == nil {
        fmt.Println("memcache New failed")
    }

    //set key-value
    mc.Set(&memcache.Item{Key: "foo", Value: []byte("my value")})

    //get key's value
    it, _ := mc.Get("foo")
    if string(it.Key) == "foo" {
        fmt.Println("value is ", string(it.Value))
    } else {
```

```
        fmt.Println("Get failed")
    }
    ///Add a new key-value
    mc.Add(&memcache.Item{Key: "foo", Value: []byte("bluegogo")})
    it, err := mc.Get("foo")
    if err != nil {
        fmt.Println("Add failed")
    } else {
        if string(it.Key) == "foo" {
            fmt.Println("Add value is ", string(it.Value))
        } else {
            fmt.Println("Get failed")
        }
    }
    //replace a key's value
    mc.Replace(&memcache.Item{Key: "foo", Value: []byte("mobike")})
    it, err = mc.Get("foo")
    if err != nil {
        fmt.Println("Replace failed")
    } else {
        if string(it.Key) == "foo" {
            fmt.Println("Replace value is ", string(it.Value))
        } else {
            fmt.Println("Replace failed")
        }
    }
    //delete an exist key
    err = mc.Delete("foo")
    if err != nil {
        fmt.Println("Delete failed:", err.Error())
    }
    //incrby
    err = mc.Set(&memcache.Item{Key: "aaa", Value: []byte("1")})
    if err != nil {
        fmt.Println("Set failed :", err.Error())
    }
    it, err = mc.Get("foo")
    if err != nil {
        fmt.Println("Get failed ", err.Error())
    } else {
        fmt.Println("src value is:", it.Value)
    }
    value, err := mc.Increment("aaa", 7)
    if err != nil {
        fmt.Println("Increment failed")
    } else {
```

```
        fmt.Println("after increment the value is :", value)
    }
    //decrby
    value, err = mc.Decrement("aaa", 4)
    if err != nil {
        fmt.Println("Decrement failed", err.Error())
    } else {
        fmt.Println("after decrement the value is ", value)
    }
}
```

常用标准库

fmt

Time

Flag

Log

IO操作

Strconv

Template

Http

Context

数据格式

反射

文件操作

go module

fmt

fmt包实现了类似C语言printf和scanf的格式化I/O。主要分为向外输出内容和获取输入内容两大部分。

向外输出

标准库fmt提供了以下几种输出相关函数。

Print

Print系列函数会将内容输出到系统的标准输出，区别在于Print函数直接输出内容，Printf函数支持格式化输出字符串，Println函数会在输出内容的结尾添加一个换行符。

```
func Print(a ...interface{}) (n int, err error)
func Printf(format string, a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
```

举个简单的例子：

```
func main() {
    fmt.Print("在终端打印该信息。")
    name := "枯藤"
    fmt.Printf("我是：%s\n", name)
    fmt.Println("在终端打印单独一行显示")
}
```

执行上面的代码输出：

```
    在终端打印该信息。我是：枯藤
    在终端打印单独一行显示
```

Fprint

Fprint系列函数会将内容输出到一个io.Writer接口类型的变量w中，我们通常用这个函数往文件中写入内容。

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error)
func Fprintln(w io.Writer, a ...interface{}) (n int, err error)
```

举个例子：

```
// 向标准输出写入内容
fmt.Fprintln(os.Stdout, "向标准输出写入内容")
fileObj, err := os.OpenFile("./xx.txt", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 064
4)
if err != nil {
	fmt.Println("打开文件出错，err:", err)
	return
}
name := "枯藤"
// 向打开的文件句柄中写入内容
fmt.Fprintf(fileObj, "往文件中写如信息：%s", name)
```

注意，只要满足io.Writer接口的类型都支持写入。

Sprint

Sprint系列函数会把传入的数据生成并返回一个字符串。

```
func Sprint(a ...interface{}) string
func Sprintf(format string, a ...interface{}) string
func Sprintln(a ...interface{}) string
```

简单的示例代码如下：

```
s1 := fmt.Sprint("枯藤")
name := "枯藤"
age := 18
s2 := fmt.Sprintf("name:%s,age:%d", name, age)
s3 := fmt.Sprintln("枯藤")
fmt.Println(s1, s2, s3)
```

Errorf

Errorf函数根据format参数生成格式化字符串并返回一个包含该字符串的错误。

```
func Errorf(format string, a ...interface{}) error
```

通常使用这种方式来自定义错误类型，例如：

```
err := fmt.Errorf("这是一个错误")
```

格式化占位符

`*printf` 系列函数都支持format格式化参数，在这里我们按照占位符将被替换的变量类型划分，方便查询和记忆。

通用占位符

占位符	说明
%v	值的默认格式表示
%+v	类似%v，但输出结构体时会添加字段名
%#v	值的Go语法表示
%T	打印值的类型
%%	百分号

示例代码如下：

```
fmt.Printf("%v\n", 100)
fmt.Printf("%v\n", false)
o := struct{ name string }{"枯藤"}
fmt.Printf("%v\n", o)
fmt.Printf("%#v\n", o)
fmt.Printf("%T\n", o)
fmt.Printf("100%%\n")
```

输出结果如下：

```
100
false
{枯藤}
struct { name string }{name:"枯藤"}
struct { name string }
100%
```

布尔型

占位符	说明
%t	true或false

整型

占位符	说明
%b	表示为二进制
%c	该值对应的unicode码值
%d	表示为十进制
%o	表示为八进制
%x	表示为十六进制，使用a-f
%X	表示为十六进制，使用A-F
%U	表示为Unicode格式：U+1234，等价于"U+%04X"
%q	该值对应的单引号括起来的go语法字符字面值，必要时会采用安全的转义表示

示例代码如下：

```
n := 65
fmt.Printf("%b\n", n)
fmt.Printf("%c\n", n)
fmt.Printf("%d\n", n)
fmt.Printf("%o\n", n)
fmt.Printf("%x\n", n)
fmt.Printf("%X\n", n)
```

输出结果如下：

```
1000001
A
65
101
41
41
```

浮点数与复数

占位符	说明

占位符	说明
%b	无小数部分、二进制指数的科学计数法，如-123456p-78
%e	科学计数法，如-1234.456e+78
%E	科学计数法，如-1234.456E+78
%f	有小数部分但无指数部分，如123.456
%F	等价于%f
%g	根据实际情况采用%e或%f格式（以获得更简洁、准确的输出）
%G	根据实际情况采用%E或%F格式（以获得更简洁、准确的输出）

示例代码如下：

```
f := 12.34
fmt.Printf("%b\n", f)
fmt.Printf("%e\n", f)
fmt.Printf("%E\n", f)
fmt.Printf("%f\n", f)
fmt.Printf("%g\n", f)
fmt.Printf("%G\n", f)
```

输出结果如下：

```
    6946802425218990p-49
    1.234000e+01
    1.234000E+01
    12.340000
    12.34
    12.34
```

字符串和[]byte

占位符	说明
%s	直接输出字符串或者[]byte
%q	该值对应的双引号括起来的go语法字符串字面值， 必要时会采用安全的转义表示

占位符	说明
%x	每个字节用两字符十六进制数表示（使用a-f
%X	每个字节用两字符十六进制数表示（使用A-F）

示例代码如下：

```
    s := "枯藤"
    fmt.Printf("%s\n", s)
    fmt.Printf("%q\n", s)
    fmt.Printf("%x\n", s)
    fmt.Printf("%X\n", s)
```

输出结果如下：

```
    枯藤
    "枯藤"
    e69eafe897a4
    E69EAFE897A4
```

指针

占位符	说明
%p	表示为十六进制，并加上前导的0x

示例代码如下：

```
a := 18
fmt.Printf("%p\n", &a)
fmt.Printf("%#p\n", &a)
```

输出结果如下：

```
    0xc000054058
    c000054058
```

宽度标识符

宽度通过一个紧跟在百分号后面的十进制数指定，如果未指定宽度，则表示值时除必需之外不作填充。精度通过（可选的）宽度后跟点号后跟的十进制数指定。如果未指定精度，会使用默认精度；如果点号后没有跟数字，表示精度为0。举例如下

占位符	说明
%f	默认宽度，默认精度
%9f	宽度9，默认精度
%.2f	默认宽度，精度2
%9.2f	宽度9，精度2
%9.f	宽度9，精度0

示例代码如下：

```
n := 88.88
fmt.Printf("%f\n", n)
fmt.Printf("%9f\n", n)
fmt.Printf("%.2f\n", n)
fmt.Printf("%9.2f\n", n)
fmt.Printf("%9.f\n", n)
```

输出结果如下：

```
88.880000
88.880000
88.88
    88.88
       89
```

其他falg

占位符	说明
'+'	总是输出数值的正负号；对%q（%+q）会生成全部是ASCII字符的输出（通过转义）；
' '	对数值，正数前加空格而负数前加负号；对字符串采用%x或%X时（% x或% X）会给各打印的字节之间加空格

占位符	说明
'-'	在输出右边填充空白而不是默认的左边（即从默认的右对齐切换为左对齐）；
'#'	八进制数前加0（%#o），十六进制数前加0x（%#x）或0X（%#X）， 指针去掉前面的0x（%#p）对%q（%#q），对%U（%#U） 会输出空格和单引号括起来的go字面值；
'0'	使用0而不是空格填充，对于数值类型会把填充的0放在正负号后面；

举个例子：

```
s := "枯藤"
fmt.Printf("%s\n", s)
fmt.Printf("%5s\n", s)
fmt.Printf("%-5s\n", s)
fmt.Printf("%5.7s\n", s)
fmt.Printf("%-5.7s\n", s)
fmt.Printf("%5.2s\n", s)
fmt.Printf("%05s\n", s)
```

输出结果如下：

```
   枯藤
      枯藤
   枯藤
      枯藤
   枯藤
      枯藤
   000枯藤
```

获取输入

Go语言fmt包下有fmt.Scan、fmt.Scanf、fmt.Scanln三个函数，可以在程序运行过程中从标准输入获取用户的输入。

fmt.Scan

函数定签名如下：

```
func Scan(a ...interface{}) (n int, err error)
```

- Scan从标准输入扫描文本，读取由空白符分隔的值保存到传递给本函数的参数中，换行符视为空白符。
- 本函数返回成功扫描的数据个数和遇到的任何错误。如果读取的数据个数比提供的参数少，会返回一个错误报告原因。

具体代码示例如下：

```
func main() {
	var (
		name    string
		age     int
		married bool
	)
	fmt.Scan(&name, &age, &married)
	fmt.Printf("扫描结果 name:%s age:%d married:%t \n", name, age, married)
}
```

将上面的代码编译后在终端执行，在终端依次输入枯藤、18和false使用空格分隔。

```
	$ ./scan_demo
	枯藤 18 false
	扫描结果 name:枯藤 age:18 married:false
```

fmt.Scan从标准输入中扫描用户输入的数据，将以空白符分隔的数据分别存入指定的参数。

fmt.Scanf

函数签名如下：

```
func Scanf(format string, a ...interface{}) (n int, err error)
```

- Scanf从标准输入扫描文本，根据format参数指定的格式去读取由空白符分隔的值保存到传递给本函数的参数中。
- 本函数返回成功扫描的数据个数和遇到的任何错误。

代码示例如下：

```
func main() {
	var (
		name    string
		age     int
```

```
        married bool
    )
    fmt.Scanf("1:%s 2:%d 3:%t", &name, &age, &married)
    fmt.Printf("扫描结果 name:%s age:%d married:%t \n", name, age, married)
}
```

将上面的代码编译后在终端执行，在终端按照指定的格式依次输入枯藤、18和false。

```
    $ ./scan_demo
    1:枯藤 2:18 3:false
    扫描结果 name:枯藤 age:18 married:false
```

fmt.Scanf不同于fmt.Scan简单的以空格作为输入数据的分隔符，fmt.Scanf为输入数据指定了具体的输入内容格式，只有按照格式输入数据才会被扫描并存入对应变量。

例如，我们还是按照上个示例中以空格分隔的方式输入，fmt.Scanf就不能正确扫描到输入的数据。

```
    $ ./scan_demo
    枯藤 18 false
    扫描结果 name: age:0 married:false
```

fmt.Scanln

函数签名如下：

```
func Scanln(a ...interface{}) (n int, err error)
```

- Scanln类似Scan，它在遇到换行时才停止扫描。最后一个数据后面必须有换行或者到达结束位置。
- 本函数返回成功扫描的数据个数和遇到的任何错误。

具体代码示例如下：

```
    func main() {
        var (
            name    string
            age     int
            married bool
        )
        fmt.Scanln(&name, &age, &married)
```

```
        fmt.Printf("扫描结果 name:%s age:%d married:%t \n", name, age, married)
    }
```

将上面的代码编译后在终端执行，在终端依次输入枯藤、18和false使用空格分隔。

```
    $ ./scan_demo
    枯藤 18 false
    扫描结果 name:枯藤 age:18 married:false
```

fmt.Scanln遇到回车就结束扫描了，这个比较常用。

bufio.NewReader

有时候我们想完整获取输入的内容，而输入的内容可能包含空格，这种情况下可以使用bufio包来实现。示例代码如下：

```
func bufioDemo() {
    reader := bufio.NewReader(os.Stdin) // 从标准输入生成读对象
    fmt.Print("请输入内容：")
    text, _ := reader.ReadString('\n') // 读到换行
    text = strings.TrimSpace(text)
    fmt.Printf("%#v\n", text)
}
```

Fscan系列

这几个函数功能分别类似于fmt.Scan、fmt.Scanf、fmt.Scanln三个函数，只不过它们不是从标准输入中读取数据而是从io.Reader中读取数据。

```
func Fscan(r io.Reader, a ...interface{}) (n int, err error)
func Fscanln(r io.Reader, a ...interface{}) (n int, err error)
func Fscanf(r io.Reader, format string, a ...interface{}) (n int, err error)
```

Sscan系列

这几个函数功能分别类似于fmt.Scan、fmt.Scanf、fmt.Scanln三个函数，只不过它们不是从标准输入中读取数据而是从指定字符串中读取数据。

```
func Sscan(str string, a ...interface{}) (n int, err error)
func Sscanln(str string, a ...interface{}) (n int, err error)
func Sscanf(str string, format string, a ...interface{}) (n int, err error)
```

Time

时间和日期是我们编程中经常会用到的，本文主要介绍了Go语言内置的time包的基本用法。

time包

time包提供了时间的显示和测量用的函数。日历的计算采用的是公历。

时间类型

time.Time类型表示时间。我们可以通过time.Now()函数获取当前的时间对象，然后获取时间对象的年月日时分秒等信息。示例代码如下：

```
func timeDemo() {
	now := time.Now() //获取当前时间
	fmt.Printf("current time:%v\n", now)

	year := now.Year()     //年
	month := now.Month()   //月
	day := now.Day()       //日
	hour := now.Hour()     //小时
	minute := now.Minute() //分钟
	second := now.Second() //秒
	fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute, second)
}
```

时间戳

时间戳是自1970年1月1日（08:00:00GMT）至当前时间的总毫秒数。它也被称为Unix时间戳（UnixTimestamp）。

基于时间对象获取时间戳的示例代码如下：

```
func timestampDemo() {
	now := time.Now()            //获取当前时间
	timestamp1 := now.Unix()     //时间戳
	timestamp2 := now.UnixNano() //纳秒时间戳
	fmt.Printf("current timestamp1:%v\n", timestamp1)
	fmt.Printf("current timestamp2:%v\n", timestamp2)
}
```

使用time.Unix()函数可以将时间戳转为时间格式。

```
func timestampDemo2(timestamp int64) {
    timeObj := time.Unix(timestamp, 0) //将时间戳转为时间格式
    fmt.Println(timeObj)
    year := timeObj.Year()     //年
    month := timeObj.Month()   //月
    day := timeObj.Day()       //日
    hour := timeObj.Hour()     //小时
    minute := timeObj.Minute() //分钟
    second := timeObj.Second() //秒
    fmt.Printf("%d-%02d-%02d %02d:%02d:%02d\n", year, month, day, hour, minute,
 second)
}
```

时间间隔

time.Duration是time包定义的一个类型，它代表两个时间点之间经过的时间，以纳秒为单位。time.Duration表示一段时间间隔，可表示的最长时间段大约290年。

time包中定义的时间间隔类型的常量如下：

```
const (
    Nanosecond  Duration = 1
    Microsecond          = 1000 * Nanosecond
    Millisecond          = 1000 * Microsecond
    Second               = 1000 * Millisecond
    Minute               = 60 * Second
    Hour                 = 60 * Minute
)
```

例如：time.Duration表示1纳秒，time.Second表示1秒。

时间操作

Add

我们在日常的编码过程中可能会遇到要求时间+时间间隔的需求，Go语言的时间对象有提供Add方法如下：

```
    func (t Time) Add(d Duration) Time
```

举个例子，求一个小时之后的时间：

```
func main() {
    now := time.Now()
    later := now.Add(time.Hour) // 当前时间加1小时后的时间
    fmt.Println(later)
}
```

Sub

求两个时间之间的差值：

```
    func (t Time) Sub(u Time) Duration
```

返回一个时间段t-u。如果结果超出了Duration可以表示的最大值/最小值，将返回最大值/最小值。要获取时间点t-d（d为Duration），可以使用t.Add(-d)。

Equal

```
    func (t Time) Equal(u Time) bool
```

判断两个时间是否相同，会考虑时区的影响，因此不同时区标准的时间也可以正确比较。本方法和用t==u不同，这种方法还会比较地点和时区信息。

Before

```
    func (t Time) Before(u Time) bool
```

如果t代表的时间点在u之前，返回真；否则返回假。

After

```
    func (t Time) After(u Time) bool
```

如果t代表的时间点在u之后，返回真；否则返回假。

定时器

使用time.Tick(时间间隔)来设置定时器，定时器的本质上是一个通道（channel）。

```
func tickDemo() {
    ticker := time.Tick(time.Second) //定义一个1秒间隔的定时器
    for i := range ticker {
```

```
        fmt.Println(i)//每秒都会执行的任务
    }
}
```

时间格式化

时间类型有一个自带的方法Format进行格式化，需要注意的是Go语言中格式化时间模板不是常见的Y-m-d H:M:S而是使用Go的诞生时间2006年1月2号15点04分（记忆口诀为2006 1 2 3 4）。也许这就是技术人员的浪漫吧。

补充：如果想格式化为12小时方式，需指定PM。

```
func formatDemo() {
    now := time.Now()
    // 格式化的模板为Go的出生时间2006年1月2号15点04分 Mon Jan
    // 24小时制
    fmt.Println(now.Format("2006-01-02 15:04:05.000 Mon Jan"))
    // 12小时制
    fmt.Println(now.Format("2006-01-02 03:04:05.000 PM Mon Jan"))
    fmt.Println(now.Format("2006/01/02 15:04"))
    fmt.Println(now.Format("15:04 2006/01/02"))
    fmt.Println(now.Format("2006/01/02"))
}
```

解析字符串格式的时间

```
now := time.Now()
fmt.Println(now)
// 加载时区
loc, err := time.LoadLocation("Asia/Shanghai")
if err != nil {
    fmt.Println(err)
    return
}
// 按照指定时区和指定格式解析字符串时间
timeObj, err := time.ParseInLocation("2006/01/02 15:04:05", "2019/08/04 14:15:20", loc)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println(timeObj)
fmt.Println(timeObj.Sub(now))
```

Flag

Go语言内置的flag包实现了命令行参数的解析，flag包使得开发命令行工具更为简单。

os.Args

如果你只是简单的想要获取命令行参数，可以像下面的代码示例一样使用os.Args来获取命令行参数。

```
package main

import (
    "fmt"
    "os"
)

//os.Args demo
func main() {
    //os.Args是一个[]string
    if len(os.Args) > 0 {
        for index, arg := range os.Args {
            fmt.Printf("args[%d]=%v\n", index, arg)
        }
    }
}
```

将上面的代码执行go build -o "args_demo"编译之后，执行：

```
    $ ./args_demo a b c d
    args[0]=./args_demo
    args[1]=a
    args[2]=b
    args[3]=c
    args[4]=d
```

os.Args是一个存储命令行参数的字符串切片，它的第一个元素是执行文件的名称。

flag包基本使用

本文介绍了flag包的常用函数和基本用法，更详细的内容请查看官方文档。

导入flag包

```
import flag
```

flag参数类型

flag包支持的命令行参数类型有bool、int、int64、uint、uint64、float float64、string、duration。

flag参数	有效值
字符串flag	合法字符串
整数flag	1234、0664、0x1234等类型，也可以是负数。
浮点数flag	合法浮点数
bool类型flag	1, 0, t, f, T, F, true, false, TRUE, FALSE, True, False。
时间段flag	任何合法的时间段字符串。如”300ms”、”-1.5h”、”2h45m”。合法的单位有”ns”、”us” /“µs”、”ms”、”s”、”m”、”h”。

定义命令行flag参数

有以下两种常用的定义命令行flag参数的方法。

flag.Type()

基本格式如下：

`flag.Type(flag名, 默认值, 帮助信息)*Type` 例如我们要定义姓名、年龄、婚否三个命令行参数，我们可以按如下方式定义：

```
name := flag.String("name", "张三", "姓名")
age := flag.Int("age", 18, "年龄")
married := flag.Bool("married", false, "婚否")
delay := flag.Duration("d", 0, "时间间隔")
```

需要注意的是，此时name、age、married、delay均为对应类型的指针。

flag.TypeVar()

基本格式如下： flag.TypeVar(Type指针, flag名, 默认值, 帮助信息) 例如我们要定义姓名、年龄、婚否三个命令行参数，我们可以按如下方式定义：

```
var name string
var age int
var married bool
var delay time.Duration
flag.StringVar(&name, "name", "张三", "姓名")
flag.IntVar(&age, "age", 18, "年龄")
flag.BoolVar(&married, "married", false, "婚否")
flag.DurationVar(&delay, "d", 0, "时间间隔")
```

flag.Parse()

通过以上两种方法定义好命令行flag参数后，需要通过调用flag.Parse()来对命令行参数进行解析。

支持的命令行参数格式有以下几种：

- -flag xxx （使用空格，一个-符号）
- –flag xxx （使用空格，两个-符号）
- -flag=xxx （使用等号，一个-符号）
- –flag=xxx （使用等号，两个-符号）

其中，布尔类型的参数必须使用等号的方式指定。

Flag解析在第一个非flag参数（单个"-"不是flag参数）之前停止，或者在终止符"–"之后停止。

flag其他函数

- flag.Args() ////返回命令行参数后的其他参数，以[]string类型
- flag.NArg() //返回命令行参数后的其他参数个数
- flag.NFlag() //返回使用的命令行参数个数

完整示例

定义

```
func main() {
    //定义命令行参数方式1
    var name string
    var age int
    var married bool
    var delay time.Duration
    flag.StringVar(&name, "name", "张三", "姓名")
```

```
    flag.IntVar(&age, "age", 18, "年龄")
    flag.BoolVar(&married, "married", false, "婚否")
    flag.DurationVar(&delay, "d", 0, "延迟的时间间隔")

    //解析命令行参数
    flag.Parse()
    fmt.Println(name, age, married, delay)
    //返回命令行参数后的其他参数
    fmt.Println(flag.Args())
    //返回命令行参数后的其他参数个数
    fmt.Println(flag.NArg())
    //返回使用的命令行参数个数
    fmt.Println(flag.NFlag())
}
```

使用

命令行参数使用提示：

```
    $ ./flag_demo -help
    Usage of ./flag_demo:
      -age int
            年龄 (default 18)
      -d duration
            时间间隔
      -married
            婚否
      -name string
            姓名 (default "张三")
```

正常使用命令行flag参数：

```
    $ ./flag_demo -name pprof --age 28 -married=false -d=1h30m
    pprof 28 false 1h30m0s
    []
    0
    4
```

使用非flag命令行参数：

```
    $ ./flag_demo a b c
    张三 18 false 0s
    [a b c]
```

```
3
0
```

Log

Go语言内置的log包实现了简单的日志服务。本文介绍了标准库log的基本使用。

使用Logger

log包定义了Logger类型，该类型提供了一些格式化输出的方法。本包也提供了一个预定义的“标准”logger，可以通过调用函数Print系列(Print|Printf|Println）、Fatal系列（Fatal|Fatalf|Fatalln）、和Panic系列（Panic|Panicf|Panicln）来使用，比自行创建一个logger对象更容易使用。

例如，我们可以像下面的代码一样直接通过log包来调用上面提到的方法，默认它们会将日志信息打印到终端界面：

```
package main

import (
    "log"
)

func main() {
    log.Println("这是一条很普通的日志。")
    v := "很普通的"
    log.Printf("这是一条%s日志。\n", v)
    log.Fatalln("这是一条会触发fatal的日志。")
    log.Panicln("这是一条会触发panic的日志。")
}
```

编译并执行上面的代码会得到如下输出：

```
    2019/10/11 14:04:17 这是一条很普通的日志。
    2019/10/11 14:04:17 这是一条很普通的日志。
    2019/10/11 14:04:17 这是一条会触发fatal的日志。
```

logger会打印每条日志信息的日期、时间，默认输出到系统的标准错误。Fatal系列函数会在写入日志信息后调用os.Exit(1)。Panic系列函数会在写入日志信息后panic。

配置logger

默认情况下的logger只会提供日志的时间信息，但是很多情况下我们希望得到更多信息，比如记录该日志的文件名和行号等。log标准库中为我们提供了定制这些设置的方法。

log标准库中的Flags函数会返回标准logger的输出配置，而SetFlags函数用来设置标准logger的输出配置。

```
    func Flags() int
    func SetFlags(flag int)
```

flag选项

log标准库提供了如下的flag选项，它们是一系列定义好的常量。

```
const (
    // 控制输出日志信息的细节，不能控制输出的顺序和格式。
    // 输出的日志在每一项后会有一个冒号分隔：例如2009/01/23 01:23:23.123123 /a/b/c/d.go:23: message
    Ldate         = 1 << iota     // 日期：2009/01/23
    Ltime                         // 时间：01:23:23
    Lmicroseconds                 // 微秒级别的时间：01:23:23.123123（用于增强Ltime位）
    Llongfile                     // 文件全路径名+行号： /a/b/c/d.go:23
    Lshortfile                    // 文件名+行号：d.go:23（会覆盖掉Llongfile）
    LUTC                          // 使用UTC时间
    LstdFlags     = Ldate | Ltime // 标准logger的初始值
)
```

下面我们在记录日志之前先设置一下标准logger的输出选项如下：

```
func main() {
    log.SetFlags(log.Llongfile | log.Lmicroseconds | log.Ldate)
    log.Println("这是一条很普通的日志。")
}
```

编译执行后得到的输出结果如下：

```
    2019/10/11 14:05:17.494943 .../log_demo/main.go:11: 这是一条很普通的日志。
```

配置日志前缀

log标准库中还提供了关于日志信息前缀的两个方法：

```
    func Prefix() string
    func SetPrefix(prefix string)
```

其中Prefix函数用来查看标准logger的输出前缀，SetPrefix函数用来设置输出前缀。

```
func main() {
	log.SetFlags(log.Llongfile | log.Lmicroseconds | log.Ldate)
	log.Println("这是一条很普通的日志。")
	log.SetPrefix("[pprof]")
	log.Println("这是一条很普通的日志。")
}
```

上面的代码输出如下：

```
	[pprof]2019/10/11 14:05:57.940542 .../log_demo/main.go:13: 这是一条很普通的日志。
```

这样我们就能够在代码中为我们的日志信息添加指定的前缀，方便之后对日志信息进行检索和处理。

配置日志输出位置

```
	func SetOutput(w io.Writer)
```

SetOutput函数用来设置标准logger的输出目的地，默认是标准错误输出。

例如，下面的代码会把日志输出到同目录下的xx.log文件中。

```
func main() {
	logFile, err := os.OpenFile("./xx.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0644)
	if err != nil {
		fmt.Println("open log file failed, err:", err)
		return
	}
	log.SetOutput(logFile)
	log.SetFlags(log.Llongfile | log.Lmicroseconds | log.Ldate)
	log.Println("这是一条很普通的日志。")
	log.SetPrefix("[小王子]")
	log.Println("这是一条很普通的日志。")
}
```

如果你要使用标准的logger，我们通常会把上面的配置操作写到init函数中。

```
func init() {
	logFile, err := os.OpenFile("./xx.log", os.O_CREATE|os.O_WRONLY|os.O_APPEND,
```

```
0644)
    if err != nil {
        fmt.Println("open log file failed, err:", err)
        return
    }
    log.SetOutput(logFile)
    log.SetFlags(log.Llongfile | log.Lmicroseconds | log.Ldate)
}
```

创建logger

log标准库中还提供了一个创建新logger对象的构造函数–New，支持我们创建自己的logger示例。New函数的签名如下：

```
    func New(out io.Writer, prefix string, flag int) *Logger
```

New创建一个Logger对象。其中，参数out设置日志信息写入的目的地。参数prefix会添加到生成的每一条日志前面。参数flag定义日志的属性（时间、文件等等）。

举个例子：

```
func main() {
    logger := log.New(os.Stdout, "<New>", log.Lshortfile|log.Ldate|log.Ltime)
    logger.Println("这是自定义的logger记录的日志。")
}
```

将上面的代码编译执行之后，得到结果如下：

```
    <New>2019/10/11 14:06:51 main.go:34: 这是自定义的logger记录的日志。
```

总结 :
Go内置的log库功能有限，例如无法满足记录不同级别日志的情况，我们在实际的项目中根据自己的需要选择使用第三方的日志库，如logrus、zap等。

IO操作

输入输出的底层原理

- 终端其实是一个文件，相关实例如下：
  - `os.Stdin` ：标准输入的文件实例，类型为 `*File`
  - `os.Stdout` ：标准输出的文件实例，类型为 `*File`
  - `os.Stderr` ：标准错误输出的文件实例，类型为 `*File`

以文件的方式操作终端:

```
package main

import "os"

func main() {
    var buf [16]byte
    os.Stdin.Read(buf[:])
    os.Stdin.WriteString(string(buf[:]))
}
```

文件操作相关API

- `func Create(name string) (file *File, err Error)`
  - 根据提供的文件名创建新的文件，返回一个文件对象，默认权限是0666
- `func NewFile(fd uintptr, name string) *File`
  - 根据文件描述符创建相应的文件，返回一个文件对象
- `func Open(name string) (file *File, err Error)`
  - 只读方式打开一个名称为name的文件
- `func OpenFile(name string, flag int, perm uint32) (file *File, err Error)`
  - 打开名称为name的文件，flag是打开的方式，只读、读写等，perm是权限
- `func (file *File) Write(b []byte) (n int, err Error)`
  - 写入byte类型的信息到文件
- `func (file *File) WriteAt(b []byte, off int64) (n int, err Error)`
  - 在指定位置开始写入byte类型的信息
- `func (file *File) WriteString(s string) (ret int, err Error)`

  - 写入string信息到文件
- `func (file *File) Read(b []byte) (n int, err Error)`
  - 读取数据到b中
- `func (file *File) ReadAt(b []byte, off int64) (n int, err Error)`
  - 从off开始读取数据到b中
- `func Remove(name string) Error`
  - 删除文件名为name的文件

打开和关闭文件

`os.Open()` 函数能够打开一个文件，返回一个 `*File` 和一个 `err` 。对得到的文件实例调用close()方法能够关闭文件。

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 只读方式打开当前目录下的main.go文件
    file, err := os.Open("./main.go")
    if err != nil {
        fmt.Println("open file failed!, err:", err)
        return
    }
    // 关闭文件
    file.Close()
}
```

写文件

```
package main

import (
    "fmt"
    "os"
)

func main() {
```

```
    // 新建文件
    file, err := os.Create("./xxx.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()
    for i := 0; i < 5; i++ {
        file.WriteString("ab\n")
        file.Write([]byte("cd\n"))
    }
}
```

读文件

文件读取可以用file.Read()和file.ReadAt()，读到文件末尾会返回io.EOF的错误

```
package main

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

func main() {
    // 打开文件
    file, err := os.Open("./xxx.txt")
    if err != nil {
        fmt.Println("open file err :", err)
        return
    }
    defer file.Close()
    // 定义接收文件读取的字节数组
    var buf [128]byte
    var content []byte
    for {
        n, err := file.Read(buf[:])
        if err == io.EOF {
            // 读取结束
            break
        }
        if err != nil {
            fmt.Println("read file err ", err)
            return
```

```
        }
        content = append(content, buf[:n]...)
    }
    fmt.Println(string(content))
}
```

拷贝文件

```
package main

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

func main() {
    // 打开源文件
    srcFile, err := os.Open("./xxx.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    // 创建新文件
    dstFile, err2 := os.Create("./abc2.txt")
    if err2 != nil {
        fmt.Println(err2)
        return
    }
    // 缓冲读取
    buf := make([]byte, 1024)
    for {
        // 从源文件读数据
        n, err := srcFile.Read(buf)
        if err == io.EOF {
            fmt.Println("读取完毕")
            break
        }
        if err != nil {
            fmt.Println(err)
            break
        }
        //写出去
        dstFile.Write(buf[:n])
    }
```

```
    srcFile.Close()
    dstFile.Close()
}
```

bufio

- bufio包实现了带缓冲区的读写，是对文件读写的封装
- bufio缓冲写数据

模式	含义
os.O_WRONLY	只写
os.O_CREATE	创建文件
os.O_RDONLY	只读
os.O_RDWR	读写
os.O_TRUNC	清空
os.O_APPEND	追加

- bufio读数据

```
package main

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

func wr() {
    // 参数2：打开模式，所有模式d都在上面
    // 参数3是权限控制
    // w写 r读 x执行   w  2   r  4   x  1
    file, err := os.OpenFile("./xxx.txt", os.O_CREATE|os.O_WRONLY, 0666)
    if err != nil {
        return
    }
    defer file.Close()
```

```
    // 获取writer对象
    writer := bufio.NewWriter(file)
    for i := 0; i < 10; i++ {
        writer.WriteString("hello\n")
    }
    // 刷新缓冲区，强制写出
    writer.Flush()
}

func re() {
    file, err := os.Open("./xxx.txt")
    if err != nil {
        return
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    for {
        line, _, err := reader.ReadLine()
        if err == io.EOF {
            break
        }
        if err != nil {
            return
        }
        fmt.Println(string(line))
    }

}

func main() {
    re()
}
```

ioutil工具包

- 工具包写文件
- 工具包读取文件

```
package main

import (
   "fmt"
   "io/ioutil"
)
```

```go
func wr() {
    err := ioutil.WriteFile("./yyy.txt", []byte("www.51mh.com"), 0666)
    if err != nil {
        fmt.Println(err)
        return
    }
}

func re() {
    content, err := ioutil.ReadFile("./yyy.txt")
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(content))
}

func main() {
    re()
}
```

例子

实现一个cat命令

使用文件操作相关知识，模拟实现linux平台cat命令的功能。

```go
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)

// cat命令实现
func cat(r *bufio.Reader) {
    for {
        buf, err := r.ReadBytes('\n') //注意是字符
        if err == io.EOF {
            break
        }
```

```
        fmt.Fprintf(os.Stdout, "%s", buf)
    }
}

func main() {
    flag.Parse() // 解析命令行参数
    if flag.NArg() == 0 {
        // 如果没有参数默认从标准输入读取内容
        cat(bufio.NewReader(os.Stdin))
    }
    // 依次读取每个指定文件的内容并打印到终端
    for i := 0; i < flag.NArg(); i++ {
        f, err := os.Open(flag.Arg(i))
        if err != nil {
            fmt.Fprintf(os.Stdout, "reading from %s failed, err:%v\n", flag.Arg
(i), err)
            continue
        }
        cat(bufio.NewReader(f))
    }
}
```

Strconv

strconv包

strconv包实现了基本数据类型与其字符串表示的转换，主要有以下常用函数： Atoi()、Itia()、parse系列、format系列、append系列。

更多函数请查看官方文档。

string与int类型转换

这一组函数是我们平时编程中用的最多的。

Atoi()

Atoi()函数用于将字符串类型的整数转换为int类型，函数签名如下。

```
func Atoi(s string) (i int, err error)
```

如果传入的字符串参数无法转换为int类型，就会返回错误。

```
s1 := "100"
i1, err := strconv.Atoi(s1)
if err != nil {
    fmt.Println("can't convert to int")
} else {
    fmt.Printf("type:%T value:%#v\n", i1, i1) //type:int value:100
}
```

Itoa()

Itoa()函数用于将int类型数据转换为对应的字符串表示，具体的函数签名如下。

```
func Itoa(i int) string
```

示例代码如下：

```
i2 := 200
s2 := strconv.Itoa(i2)
fmt.Printf("type:%T value:%#v\n", s2, s2) //type:string value:"200"
```

a的典故

【扩展阅读】这是C语言遗留下的典故。C语言中没有string类型而是用字符数组(array)表示字符串，所以Itoa对很多C系的程序员很好理解。

Parse系列函数

Parse类函数用于转换字符串为给定类型的值：ParseBool()、ParseFloat()、ParseInt()、ParseUint()。

ParseBool()

```
func ParseBool(str string) (value bool, err error)
```

返回字符串表示的bool值。它接受1、0、t、f、T、F、true、false、True、False、TRUE、FALSE；否则返回错误。

ParseInt()

```
func ParseInt(s string, base int, bitSize int) (i int64, err error)
```

返回字符串表示的整数值，接受正负号。

base指定进制（2到36），如果base为0，则会从字符串前置判断，”0x”是16进制，”0”是8进制，否则是10进制；

bitSize指定结果必须能无溢出赋值的整数类型，0、8、16、32、64 分别代表 int、int8、int16、int32、int64；

返回的err是 `*NumErr` 类型的，如果语法有误，err.Error = ErrSyntax；如果结果超出类型范围err.Error = ErrRange。

ParseUnit()

```
func ParseUint(s string, base int, bitSize int) (n uint64, err error)
```

ParseUint类似ParseInt但不接受正负号，用于无符号整型。

ParseFloat()

```
func ParseFloat(s string, bitSize int) (f float64, err error)
```

解析一个表示浮点数的字符串并返回其值。

如果s合乎语法规则，函数会返回最为接近s表示值的一个浮点数（使用IEEE754规范舍入）。

bitSize指定了期望的接收类型，32是float32（返回值可以不改变精确值的赋值给float32），64是float64；

返回值err是 `*NumErr` 类型的，语法有误的，err.Error=ErrSyntax；结果超出表示范围的，返回值f为±Inf，err.Error= ErrRange。

代码示例

```
b, err := strconv.ParseBool("true")
f, err := strconv.ParseFloat("3.1415", 64)
i, err := strconv.ParseInt("-2", 10, 64)
u, err := strconv.ParseUint("2", 10, 64)
```

这些函数都有两个返回值，第一个返回值是转换后的值，第二个返回值为转化失败的错误信息。

Format系列函数

Format系列函数实现了将给定类型数据格式化为string类型数据的功能。

FormatBool()

```
func FormatBool(b bool) string
```

根据b的值返回"true"或"false"。

FormatInt()

```
func FormatInt(i int64, base int) string
```

返回i的base进制的字符串表示。base 必须在2到36之间，结果中会使用小写字母'a'到'z'表示大于10的数字。

FormatUint()

```
func FormatUint(i uint64, base int) string
```

是FormatInt的无符号整数版本。

FormatFloat()

```
func FormatFloat(f float64, fmt byte, prec, bitSize int) string
```

函数将浮点数表示为字符串并返回。

bitSize表示f的来源类型（32：float32、64：float64），会据此进行舍入。

fmt表示格式：’f’（-ddd.dddd）、’b’（-ddddp±ddd，指数为二进制）、’e’（-d.dddde±dd，十进制指数）、’E’（-d.ddddE±dd，十进制指数）、’g’（指数很大时用’e’格式，否则’f’格式）、’G’（指数很大时用’E’格式，否则’f’格式）。

prec控制精度（排除指数部分）：对’f’、’e’、’E’，它表示小数点后的数字个数；对’g’、’G’，它控制总的数字个数。如果prec 为-1，则代表使用最少数量的、但又必需的数字来表示f。

代码示例

```
s1 := strconv.FormatBool(true)
s2 := strconv.FormatFloat(3.1415, 'E', -1, 64)
s3 := strconv.FormatInt(-2, 16)
s4 := strconv.FormatUint(2, 16)
```

其他

isPrint()

```
func IsPrint(r rune) bool
```

返回一个字符是否是可打印的，和unicode.IsPrint一样，r必须是：字母（广义）、数字、标点、符号、ASCII空格。

CanBackquote()

```
func CanBackquote(s string) bool
```

返回字符串s是否可以不被修改的表示为一个单行的、没有空格和tab之外控制字符的反引号字符串。

其他

除上文列出的函数外，strconv包中还有Append系列、Quote系列等函数。具体用法可查看官方文档。

Template

html/template包实现了数据驱动的模板，用于生成可对抗代码注入的安全HTML输出。它提供了和text/template包相同的接口，Go语言中输出HTML的场景都应使用text/template包。

模板

在基于MVC的Web架构中，我们通常需要在后端渲染一些数据到HTML文件中，从而实现动态的网页效果。

模板示例

通过将模板应用于一个数据结构（即该数据结构作为模板的参数）来执行，来获得输出。模板中的注释引用数据接口的元素（一般如结构体的字段或者字典的键）来控制执行过程和获取需要呈现的值。模板执行时会遍历结构并将指针表示为'.'（称之为"dot"）指向运行过程中数据结构的当前位置的值。

用作模板的输入文本必须是utf-8编码的文本。"Action"—数据运算和控制单位—由"{{"和"}}"界定；在Action之外的所有文本都不做修改的拷贝到输出中。Action内部不能有换行，但注释可以有换行。

HTML文件代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>Hello {{.}}</p>
</body>
</html>
```

我们的HTTP server端代码如下：

```
// main.go

func sayHello(w http.ResponseWriter, r *http.Request) {
    // 解析指定文件生成模板对象
```

```
    tmpl, err := template.ParseFiles("./hello.html")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    // 利用给定数据渲染模板，并将结果写入w
    tmpl.Execute(w, "5lmh.com")
}
func main() {
    http.HandleFunc("/", sayHello)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        fmt.Println("HTTP server failed,err:", err)
        return
    }
}
```

模板语法

`{{.}}`

模板语法都包含在 `{{和}}` 中间，其中 `{{.}}` 中的点表示当前对象。

当我们传入一个结构体对象时，我们可以根据 `.` 来访问结构体的对应字段。例如：

```
// main.go

type UserInfo struct {
    Name   string
    Gender string
    Age    int
}

func sayHello(w http.ResponseWriter, r *http.Request) {
    // 解析指定文件生成模板对象
    tmpl, err := template.ParseFiles("./hello.html")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    // 利用给定数据渲染模板，并将结果写入w
    user := UserInfo{
        Name:   "枯藤",
        Gender: "男",
        Age:    18,
```

```
    }
    tmpl.Execute(w, user)
}
```

HTML文件代码如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Hello</title>
</head>
<body>
    <p>Hello {{.Name}}</p>
    <p>性别：{{.Gender}}</p>
    <p>年龄：{{.Name}}</p>
</body>
</html>
```

同理，当我们传入的变量是map时，也可以在模板文件中通过.根据key来取值。

注释

```
    {{/* a comment */}}
    注释，执行时会忽略。可以多行。注释不能嵌套，并且必须紧贴分界符始止。
```

pipeline

pipeline是指产生数据的操作。比如 `{{.}}` 、 `{{.Name}}` 等。Go的模板语法中支持使用管道符号|链接多个命令，用法和unix下的管道类似：|前面的命令会将运算结果(或返回值)传递给后一个命令的最后一个位置。

注意：并不是只有使用了|才是pipeline。Go的模板语法中，pipeline的概念是传递数据，只要能产生数据的，都是pipeline。

变量

Action里可以初始化一个变量来捕获管道的执行结果。初始化语法如下：

```
    $variable := pipeline
```

其中$variable是变量的名字。声明变量的action不会产生任何输出。

条件判断

Go模板语法中的条件判断有以下几种:

```
{{if pipeline}} T1 {{end}}

{{if pipeline}} T1 {{else}} T0 {{end}}

{{if pipeline}} T1 {{else if pipeline}} T0 {{end}}
```

range

Go的模板语法中使用range关键字进行遍历，有以下两种写法，其中pipeline的值必须是数组、切片、字典或者通道。

```
{{range pipeline}} T1 {{end}}
如果pipeline的值其长度为0，不会有任何输出

{{range pipeline}} T1 {{else}} T0 {{end}}
如果pipeline的值其长度为0，则会执行T0。
```

with

```
{{with pipeline}} T1 {{end}}
如果pipeline为empty不产生输出，否则将dot设为pipeline的值并执行T1。不修改外面的dot。

{{with pipeline}} T1 {{else}} T0 {{end}}
如果pipeline为empty，不改变dot并执行T0，否则dot设为pipeline的值并执行T1。
```

预定义函数

执行模板时，函数从两个函数字典中查找：首先是模板函数字典，然后是全局函数字典。一般不在模板内定义函数，而是使用Funcs方法添加函数到模板里。

预定义的全局函数如下：

```
and
    函数返回它的第一个empty参数或者最后一个参数；
    就是说"and x y"等价于"if x then y else x"；所有参数都会执行；
or
```

```
    返回第一个非empty参数或者最后一个参数；
    亦即"or x y"等价于"if x then x else y"；所有参数都会执行；
not
    返回它的单个参数的布尔值的否定
len
    返回它的参数的整数类型长度
index
    执行结果为第一个参数以剩下的参数为索引/键指向的值；
    如"index x 1 2 3"返回x[1][2][3]的值；每个被索引的主体必须是数组、切片或者字典。
print
    即fmt.Sprint
printf
    即fmt.Sprintf
println
    即fmt.Sprintln
html
    返回其参数文本表示的HTML逸码等价表示。
urlquery
    返回其参数文本表示的可嵌入URL查询的逸码等价表示。
js
    返回其参数文本表示的JavaScript逸码等价表示。
call
    执行结果是调用第一个参数的返回值，该参数必须是函数类型，其余参数作为调用该函数的参数；
    如"call .X.Y 1 2"等价于go语言里的dot.X.Y(1, 2)；
    其中Y是函数类型的字段或者字典的值，或者其他类似情况；
    call的第一个参数的执行结果必须是函数类型的值（和预定义函数如print明显不同）；
    该函数类型值必须有1到2个返回值，如果有2个则后一个必须是error接口类型；
    如果有2个返回值的方法返回的error非nil，模板执行会中断并返回给调用模板执行者该错误；
```

比较函数

布尔函数会将任何类型的零值视为假，其余视为真。

下面是定义为函数的二元比较运算的集合：

```
    eq      如果arg1 == arg2则返回真
    ne      如果arg1 != arg2则返回真
    lt      如果arg1 < arg2则返回真
    le      如果arg1 <= arg2则返回真
    gt      如果arg1 > arg2则返回真
    ge      如果arg1 >= arg2则返回真
```

为了简化多参数相等检测，eq（只有eq）可以接受2个或更多个参数，它会将第一个参数和其余参数依次比较，返回下式的结果：

```
{{eq arg1 arg2 arg3}}
```

比较函数只适用于基本类型（或重定义的基本类型，如”type Celsius float32”）。但是，整数和浮点数不能互相比较。

自定义函数

Go的模板支持自定义函数。

```
func sayHello(w http.ResponseWriter, r *http.Request) {
	htmlByte, err := ioutil.ReadFile("./hello.html")
	if err != nil {
		fmt.Println("read html failed, err:", err)
		return
	}
	// 自定义一个夸人的模板函数
	kua := func(arg string) (string, error) {
		return arg + "真帅", nil
	}
	// 采用链式操作在Parse之前调用Funcs添加自定义的kua函数
	tmpl, err := template.New("hello").Funcs(template.FuncMap{"kua": kua}).Parse
(string(htmlByte))
	if err != nil {
		fmt.Println("create template failed, err:", err)
		return
	}

	user := UserInfo{
		Name:   "枯藤",
		Gender: "男",
		Age:    18,
	}
	// 使用user渲染模板，并将结果写入w
	tmpl.Execute(w, user)
}
```

我们可以在模板文件hello.html中使用我们自定义的kua函数了。

```
{{kua .Name}}
```

嵌套template

我们可以在template中嵌套其他的template。这个template可以是单独的文件，也可以是通过define定义的template。

举个例子： t.html文件内容如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>tmpl test</title>
</head>
<body>

    <h1>测试嵌套template语法</h1>
    <hr>
    {{template "ul.html"}}
    <hr>
    {{template "ol.html"}}
</body>
</html>

{{ define "ol.html"}}
<h1>这是ol.html</h1>
<ol>
    <li>吃饭</li>
    <li>睡觉</li>
    <li>打豆豆</li>
</ol>
{{end}}
```

ul.html文件内容如下：

```
<ul>
    <li>注释</li>
    <li>日志</li>
    <li>测试</li>
</ul>
```

我们注册一个templDemo路由处理函数.

```
http.HandleFunc("/tmpl", tmplDemo)
```

tmplDemo函数的具体内容如下：

```
func tmplDemo(w http.ResponseWriter, r *http.Request) {
    tmpl, err := template.ParseFiles("./t.html", "./ul.html")
    if err != nil {
        fmt.Println("create template failed, err:", err)
        return
    }
    user := UserInfo{
        Name:   "枯藤",
        Gender: "男",
        Age:    18,
    }
    tmpl.Execute(w, user)
}
```

Http

Go语言内置的net/http包十分的优秀，提供了HTTP客户端和服务端的实现。

net/http介绍

Go语言内置的net/http包提供了HTTP客户端和服务端的实现。

HTTP协议

超文本传输协议（HTTP，HyperText Transfer Protocol)是互联网上应用最为广泛的一种网络传输协议，所有的WWW文件都必须遵守这个标准。设计HTTP最初的目的是为了提供一种发布和接收HTML页面的方法。

HTTP客户端

基本的HTTP/HTTPS请求
Get、Head、Post和PostForm函数发出HTTP/HTTPS请求。

```
resp, err := http.Get("http://5lmh.com/")
...
resp, err := http.Post("http://5lmh.com/upload", "image/jpeg", &buf)
...
resp, err := http.PostForm("http://5lmh.com/form",
    url.Values{"key": {"Value"}, "id": {"123"}})
```

程序在使用完response后必须关闭回复的主体。

```
resp, err := http.Get("http://5lmh.com/")
if err != nil {
    // handle error
}
defer resp.Body.Close()
body, err := ioutil.ReadAll(resp.Body)
// ...
```

GET请求示例

使用net/http包编写一个简单的发送HTTP请求的Client端，代码如下：

```
package main
```

```go
import (
    "fmt"
    "io/ioutil"
    "net/http"
)

func main() {
    resp, err := http.Get("https://www.5lmh.com/")
    if err != nil {
        fmt.Println("get failed, err:", err)
        return
    }
    defer resp.Body.Close()
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("read from resp.Body failed,err:", err)
        return
    }
    fmt.Print(string(body))
}
```

将上面的代码保存之后编译成可执行文件，执行之后就能在终端打印liwenzhou.com网站首页的内容了，我们的浏览器其实就是一个发送和接收HTTP协议数据的客户端，我们平时通过浏览器访问网页其实就是从网站的服务器接收HTTP数据，然后浏览器会按照HTML、CSS等规则将网页渲染展示出来。

带参数的GET请求示例

关于GET请求的参数需要使用Go语言内置的net/url这个标准库来处理。

```go
func main() {
    apiUrl := "http://127.0.0.1:9090/get"
    // URL param
    data := url.Values{}
    data.Set("name", "枯藤")
    data.Set("age", "18")
    u, err := url.ParseRequestURI(apiUrl)
    if err != nil {
        fmt.Printf("parse url requestUrl failed,err:%v\n", err)
    }
    u.RawQuery = data.Encode() // URL encode
    fmt.Println(u.String())
    resp, err := http.Get(u.String())
    if err != nil {
        fmt.Println("post failed, err:%v\n", err)
```

```
        return
    }
    defer resp.Body.Close()
    b, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("get resp failed,err:%v\n", err)
        return
    }
    fmt.Println(string(b))
}
```

对应的Server端HandlerFunc如下：

```
func getHandler(w http.ResponseWriter, r *http.Request) {
    defer r.Body.Close()
    data := r.URL.Query()
    fmt.Println(data.Get("name"))
    fmt.Println(data.Get("age"))
    answer := `{"status": "ok"}`
    w.Write([]byte(answer))
}
```

Post请求示例

上面演示了使用net/http包发送GET请求的示例，发送POST请求的示例代码如下：

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "strings"
)

// net/http post demo

func main() {
    url := "http://127.0.0.1:9090/post"
    // 表单数据
    //contentType := "application/x-www-form-urlencoded"
    //data := "name=枯藤&age=18"
    // json
    contentType := "application/json"
```

```
	data := `{"name":"枯藤","age":18}`
	resp, err := http.Post(url, contentType, strings.NewReader(data))
	if err != nil {
		fmt.Println("post failed, err:%v\n", err)
		return
	}
	defer resp.Body.Close()
	b, err := ioutil.ReadAll(resp.Body)
	if err != nil {
		fmt.Println("get resp failed,err:%v\n", err)
		return
	}
	fmt.Println(string(b))
}
```

对应的Server端HandlerFunc如下：

```
func postHandler(w http.ResponseWriter, r *http.Request) {
	defer r.Body.Close()
	// 1. 请求类型是application/x-www-form-urlencoded时解析form数据
	r.ParseForm()
	fmt.Println(r.PostForm) // 打印form数据
	fmt.Println(r.PostForm.Get("name"), r.PostForm.Get("age"))
	// 2. 请求类型是application/json时从r.Body读取数据
	b, err := ioutil.ReadAll(r.Body)
	if err != nil {
		fmt.Println("read request.Body failed, err:%v\n", err)
		return
	}
	fmt.Println(string(b))
	answer := `{"status": "ok"}`
	w.Write([]byte(answer))
}
```

自定义Client

要管理HTTP客户端的头域、重定向策略和其他设置，创建一个Client：

```
client := &http.Client{
	CheckRedirect: redirectPolicyFunc,
}
resp, err := client.Get("http://5lmh.com")
// ...
req, err := http.NewRequest("GET", "http://5lmh.com", nil)
```

```
// ...
req.Header.Add("If-None-Match", `W/"wyzzy"`)
resp, err := client.Do(req)
// ...
```

自定义Transport

要管理代理、TLS配置、keep-alive、压缩和其他设置，创建一个Transport：

```
tr := &http.Transport{
    TLSClientConfig:    &tls.Config{RootCAs: pool},
    DisableCompression: true,
}
client := &http.Client{Transport: tr}
resp, err := client.Get("https://51mh.com")
```

Client和Transport类型都可以安全的被多个go程同时使用。出于效率考虑，应该一次建立、尽量重用。

服务端

默认的Server

ListenAndServe使用指定的监听地址和处理器启动一个HTTP服务端。处理器参数通常是nil，这表示采用包变量DefaultServeMux作为处理器。

Handle和HandleFunc函数可以向DefaultServeMux添加处理器。

```
http.Handle("/foo", fooHandler)
http.HandleFunc("/bar", func(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %q", html.EscapeString(r.URL.Path))
})
log.Fatal(http.ListenAndServe(":8080", nil))
```

默认的Server示例

使用Go语言中的net/http包来编写一个简单的接收HTTP请求的Server端示例，net/http包是对net包的进一步封装，专门用来处理HTTP协议的数据。具体的代码如下：

```
// http server

func sayHello(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintln(w, "Hello 枯藤！")
```

```
}

func main() {
    http.HandleFunc("/", sayHello)
    err := http.ListenAndServe(":9090", nil)
    if err != nil {
        fmt.Printf("http server failed, err:%v\n", err)
        return
    }
}
```

将上面的代码编译之后执行，打开你电脑上的浏览器在地址栏输入127.0.0.1:9090回车，此时就能够看到 `Hello 枯藤！`

自定义Server

要管理服务端的行为，可以创建一个自定义的Server：

```
s := &http.Server{
    Addr:           ":8080",
    Handler:        myHandler,
    ReadTimeout:    10 * time.Second,
    WriteTimeout:   10 * time.Second,
    MaxHeaderBytes: 1 << 20,
}
log.Fatal(s.ListenAndServe())
```

Context

在 Go http包的Server中，每一个请求在都有一个对应的 goroutine 去处理。请求处理函数通常会启动额外的 goroutine 用来访问后端服务，比如数据库和RPC服务。用来处理一个请求的 goroutine 通常需要访问一些与请求特定的数据，比如终端用户的身份认证信息、验证相关的token、请求的截止时间。 当一个请求被取消或超时时，所有用来处理该请求的 goroutine 都应该迅速退出，然后系统才能释放这些 goroutine 占用的资源。

为什么需要Context

基本示例

```
package main

import (
	"fmt"
	"sync"

	"time"
)

var wg sync.WaitGroup

// 初始的例子

func worker() {
	for {
		fmt.Println("worker")
		time.Sleep(time.Second)
	}
	// 如何接收外部命令实现退出
	wg.Done()
}

func main() {
	wg.Add(1)
	go worker()
	// 如何优雅的实现结束子goroutine
	wg.Wait()
	fmt.Println("over")
}
```

全局变量方式

```go
package main

import (
    "fmt"
    "sync"

    "time"
)

var wg sync.WaitGroup
var exit bool

// 全局变量方式存在的问题:
// 1. 使用全局变量在跨包调用时不容易统一
// 2. 如果worker中再启动goroutine，就不太好控制了。

func worker() {
    for {
        fmt.Println("worker")
        time.Sleep(time.Second)
        if exit {
            break
        }
    }
    wg.Done()
}

func main() {
    wg.Add(1)
    go worker()
    time.Sleep(time.Second * 3) // sleep3秒以免程序过快退出
    exit = true                 // 修改全局变量实现子goroutine的退出
    wg.Wait()
    fmt.Println("over")
}
```

通道方式

```go
package main

import (
    "fmt"
    "sync"
```

```
	"time"
)

var wg sync.WaitGroup

// 管道方式存在的问题：
// 1. 使用全局变量在跨包调用时不容易实现规范和统一，需要维护一个共用的channel

func worker(exitChan chan struct{}) {
LOOP:
	for {
		fmt.Println("worker")
		time.Sleep(time.Second)
		select {
		case <-exitChan: // 等待接收上级通知
			break LOOP
		default:
		}
	}
	wg.Done()
}

func main() {
	var exitChan = make(chan struct{})
	wg.Add(1)
	go worker(exitChan)
	time.Sleep(time.Second * 3) // sleep3秒以免程序过快退出
	exitChan <- struct{}{}      // 给子goroutine发送退出信号
	close(exitChan)
	wg.Wait()
	fmt.Println("over")
}
```

官方版的方案

```
package main

import (
	"context"
	"fmt"
	"sync"

	"time"
)
```

```go
var wg sync.WaitGroup

func worker(ctx context.Context) {
LOOP:
	for {
		fmt.Println("worker")
		time.Sleep(time.Second)
		select {
		case <-ctx.Done(): // 等待上级通知
			break LOOP
		default:
		}
	}
	wg.Done()
}

func main() {
	ctx, cancel := context.WithCancel(context.Background())
	wg.Add(1)
	go worker(ctx)
	time.Sleep(time.Second * 3)
	cancel() // 通知子goroutine结束
	wg.Wait()
	fmt.Println("over")
}
```

当子goroutine又开启另外一个goroutine时，只需要将ctx传入即可：

```go
package main

import (
	"context"
	"fmt"
	"sync"

	"time"
)

var wg sync.WaitGroup

func worker(ctx context.Context) {
	go worker2(ctx)
LOOP:
	for {
		fmt.Println("worker")
```

```
        time.Sleep(time.Second)
        select {
        case <-ctx.Done(): // 等待上级通知
            break LOOP
        default:
        }
    }
    wg.Done()
}

func worker2(ctx context.Context) {
LOOP:
    for {
        fmt.Println("worker2")
        time.Sleep(time.Second)
        select {
        case <-ctx.Done(): // 等待上级通知
            break LOOP
        default:
        }
    }
}
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    wg.Add(1)
    go worker(ctx)
    time.Sleep(time.Second * 3)
    cancel() // 通知子goroutine结束
    wg.Wait()
    fmt.Println("over")
}
```

Context初识

Go1.7加入了一个新的标准库context，它定义了Context类型，专门用来简化 对于处理单个请求的多个 goroutine 之间与请求域的数据、取消信号、截止时间等相关操作，这些操作可能涉及多个 API 调用。

对服务器传入的请求应该创建上下文，而对服务器的传出调用应该接受上下文。它们之间的函数调用链必须传递上下文，或者可以使用WithCancel、WithDeadline、WithTimeout或WithValue创建的派生上下文。当一个上下文被取消时，它派生的所有上下文也被取消。

Context接口

context.Context是一个接口，该接口定义了四个需要实现的方法。具体签名如下：

```go
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

其中：

- Deadline方法需要返回当前Context被取消的时间，也就是完成工作的截止时间（deadline）；
- Done方法需要返回一个Channel，这个Channel会在当前工作完成或者上下文被取消之后关闭，多次调用Done方法会返回同一个Channel；
- Err方法会返回当前Context结束的原因，它只会在Done返回的Channel被关闭时才会返回非空的值；
    - 如果当前Context被取消就会返回Canceled错误；
    - 如果当前Context超时就会返回DeadlineExceeded错误；
- Value方法会从Context中返回键对应的值，对于同一个上下文来说，多次调用Value 并传入相同的Key会返回相同的结果，该方法仅用于传递跨API和进程间跟请求域的数据；

Background()和TODO()

Go内置两个函数：Background()和TODO()，这两个函数分别返回一个实现了Context接口的background和todo。我们代码中最开始都是以这两个内置的上下文对象作为最顶层的partent context，衍生出更多的子上下文对象。

Background()主要用于main函数、初始化以及测试代码中，作为Context这个树结构的最顶层的Context，也就是根Context。

TODO()，它目前还不知道具体的使用场景，如果我们不知道该使用什么Context的时候，可以使用这个。

background和todo本质上都是emptyCtx结构体类型，是一个不可取消，没有设置截止时间，没有携带任何值的Context。

With系列函数

此外，context包中还定义了四个With系列函数。

WithCancel

WithCancel的函数签名如下：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
```

WithCancel返回带有新Done通道的父节点的副本。当调用返回的cancel函数或当关闭父上下文的Done通道时，将关闭返回上下文的Done通道，无论先发生什么情况。

取消此上下文将释放与其关联的资源，因此代码应该在此上下文中运行的操作完成后立即调用cancel。

```
func gen(ctx context.Context) <-chan int {
        dst := make(chan int)
        n := 1
        go func() {
            for {
                select {
                case <-ctx.Done():
                    return // return结束该goroutine，防止泄露
                case dst <- n:
                    n++
                }
            }
        }()
        return dst
    }
func main() {
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel() // 当我们取完需要的整数后调用cancel

    for n := range gen(ctx) {
        fmt.Println(n)
        if n == 5 {
            break
        }
    }
}
```

上面的示例代码中，gen函数在单独的goroutine中生成整数并将它们发送到返回的通道。 gen的调用者在使用生成的整数之后需要取消上下文，以免gen启动的内部goroutine发生泄漏。

WithDeadline

WithDeadline的函数签名如下：

```
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
```

返回父上下文的副本，并将deadline调整为不迟于d。如果父上下文的deadline已经早于d，则WithDeadline(parent, d)在语义上等同于父上下文。当截止日过期时，当调用返回的cancel函数时，或者当父上下文的Done通道关闭时，返回上下文的Done通道将被关闭，以最先发生的情况为准。

取消此上下文将释放与其关联的资源，因此代码应该在此上下文中运行的操作完成后立即调用cancel。

```
func main() {
	d := time.Now().Add(50 * time.Millisecond)
	ctx, cancel := context.WithDeadline(context.Background(), d)

	// 尽管ctx会过期，但在任何情况下调用它的cancel函数都是很好的实践。
	// 如果不这样做，可能会使上下文及其父类存活的时间超过必要的时间。
	defer cancel()

	select {
	case <-time.After(1 * time.Second):
		fmt.Println("overslept")
	case <-ctx.Done():
		fmt.Println(ctx.Err())
	}
}
```

上面的代码中，定义了一个50毫秒之后过期的deadline，然后我们调用context.WithDeadline(context.Background(), d)得到一个上下文（ctx）和一个取消函数（cancel），然后使用一个select让主程序陷入等待：等待1秒后打印overslept退出或者等待ctx过期后退出。 因为ctx50秒后就过期，所以ctx.Done()会先接收到值，上面的代码会打印ctx.Err()取消原因。

WithTimeout

WithTimeout的函数签名如下：

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

WithTimeout返回WithDeadline(parent, time.Now().Add(timeout))。

取消此上下文将释放与其相关的资源，因此代码应该在此上下文中运行的操作完成后立即调用cancel，通常用于数据库或者网络连接的超时控制。具体示例如下：

```go
package main

import (
	"context"
	"fmt"
	"sync"

	"time"
)

// context.WithTimeout

var wg sync.WaitGroup

func worker(ctx context.Context) {
LOOP:
	for {
		fmt.Println("db connecting ...")
		time.Sleep(time.Millisecond * 10) // 假设正常连接数据库耗时10毫秒
		select {
		case <-ctx.Done(): // 50毫秒后自动调用
			break LOOP
		default:
		}
	}
	fmt.Println("worker done!")
	wg.Done()
}

func main() {
	// 设置一个50毫秒的超时
	ctx, cancel := context.WithTimeout(context.Background(), time.Millisecond*50
)
	wg.Add(1)
	go worker(ctx)
	time.Sleep(time.Second * 5)
	cancel() // 通知子goroutine结束
	wg.Wait()
	fmt.Println("over")
}
```

WithValue

WithValue函数能够将请求作用域的数据与 Context 对象建立关系。声明如下：

```
    func WithValue(parent Context, key, val interface{}) Context
```

WithValue返回父节点的副本，其中与key关联的值为val。

仅对API和进程间传递请求域的数据使用上下文值，而不是使用它来传递可选参数给函数。

所提供的键必须是可比较的，并且不应该是string类型或任何其他内置类型，以避免使用上下文在包之间发生冲突。WithValue的用户应该为键定义自己的类型。为了避免在分配给interface{}时进行分配，上下文键通常具有具体类型struct{}。或者，导出的上下文关键变量的静态类型应该是指针或接口。

```
package main

import (
    "context"
    "fmt"
    "sync"

    "time"
)

// context.WithValue

type TraceCode string

var wg sync.WaitGroup

func worker(ctx context.Context) {
    key := TraceCode("TRACE_CODE")
    traceCode, ok := ctx.Value(key).(string) // 在子goroutine中获取trace code
    if !ok {
        fmt.Println("invalid trace code")
    }
LOOP:
    for {
        fmt.Printf("worker, trace code:%s\n", traceCode)
        time.Sleep(time.Millisecond * 10) // 假设正常连接数据库耗时10毫秒
        select {
        case <-ctx.Done(): // 50毫秒后自动调用
            break LOOP
        default:
        }
    }
    fmt.Println("worker done!")
    wg.Done()
```

```go
}

func main() {
	// 设置一个50毫秒的超时
	ctx, cancel := context.WithTimeout(context.Background(), time.Millisecond*50
)
	// 在系统的入口中设置trace code传递给后续启动的goroutine实现日志数据聚合
	ctx = context.WithValue(ctx, TraceCode("TRACE_CODE"), "12512312234")
	wg.Add(1)
	go worker(ctx)
	time.Sleep(time.Second * 5)
	cancel() // 通知子goroutine结束
	wg.Wait()
	fmt.Println("over")
}
```

使用Context的注意事项

- 推荐以参数的方式显示传递Context
- 以Context作为参数的函数方法，应该把Context作为第一个参数。
- 给一个函数方法传递Context的时候，不要传递nil，如果不知道传递什么，就使用context.TODO()
- Context的Value相关方法应该传递请求域的必要数据，不应该用于传递可选参数
- Context是线程安全的，可以放心的在多个goroutine中传递

客户端超时取消示例

调用服务端API时如何在客户端实现超时控制？

server端

```go
// context_timeout/server/main.go
package main

import (
	"fmt"
	"math/rand"
	"net/http"

	"time"
)
```

```go
// server端，随机出现慢响应

func indexHandler(w http.ResponseWriter, r *http.Request) {
	number := rand.Intn(2)
	if number == 0 {
		time.Sleep(time.Second * 10) // 耗时10秒的慢响应
		fmt.Fprintf(w, "slow response")
		return
	}
	fmt.Fprint(w, "quick response")
}

func main() {
	http.HandleFunc("/", indexHandler)
	err := http.ListenAndServe(":8000", nil)
	if err != nil {
		panic(err)
	}
}
```

client端

```go
// context_timeout/client/main.go
package main

import (
	"context"
	"fmt"
	"io/ioutil"
	"net/http"
	"sync"
	"time"
)

// 客户端

type respData struct {
	resp *http.Response
	err  error
}

func doCall(ctx context.Context) {
	transport := http.Transport{
	   // 请求频繁可定义全局的client对象并启用长链接
	   // 请求不频繁使用短链接
```

```
        DisableKeepAlives: true,        }
    client := http.Client{
        Transport: &transport,
    }

    respChan := make(chan *respData, 1)
    req, err := http.NewRequest("GET", "http://127.0.0.1:8000/", nil)
    if err != nil {
        fmt.Printf("new requestg failed, err:%v\n", err)
        return
    }
    req = req.WithContext(ctx) // 使用带超时的ctx创建一个新的client request
    var wg sync.WaitGroup
    wg.Add(1)
    defer wg.Wait()
    go func() {
        resp, err := client.Do(req)
        fmt.Printf("client.do resp:%v, err:%v\n", resp, err)
        rd := &respData{
            resp: resp,
            err:  err,
        }
        respChan <- rd
        wg.Done()
    }()

    select {
    case <-ctx.Done():
        //transport.CancelRequest(req)
        fmt.Println("call api timeout")
    case result := <-respChan:
        fmt.Println("call server api success")
        if result.err != nil {
            fmt.Printf("call server api failed, err:%v\n", result.err)
            return
        }
        defer result.resp.Body.Close()
        data, _ := ioutil.ReadAll(result.resp.Body)
        fmt.Printf("resp:%v\n", string(data))
    }
}

func main() {
    // 定义一个100毫秒的超时
    ctx, cancel := context.WithTimeout(context.Background(), time.Millisecond*10
0)
```

```
    defer cancel() // 调用cancel释放子goroutine资源
    doCall(ctx)
}
```

数据格式

数据格式介绍

- 是系统中数据交互不可缺少的内容
- 这里主要介绍JSON、XML、MSGPack

JSON

- json是完全独立于语言的文本格式，是k-v的形式 name:zs
- 应用场景：前后端交互，系统间数据交互

```
{
  "dates": {
    "date": [
      {
        "id": "1",
        "name": "JSON",
        "abb": "JavaScript Object Notation"
      },
      {
        "id": "2",
        "name": "XML",
        "abb": "eXtensible Markup Language"
      },
      {
        "id": "3",
        "name": "YAML",
        "abb": "Yet Another Markup Language"
      }
    ]
  }
}
```

www.topgoer.com

- json使用go语言内置的encoding/json 标准库

- 编码json使用json.Marshal()函数可以对一组数据进行JSON格式的编码

```
    func Marshal(v interface{}) ([]byte, error)
```

示例过结构体生成json

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Name  string
    Hobby string
}

func main() {
    p := Person{"5lmh.com", "女"}
    // 编码json
    b, err := json.Marshal(p)
    if err != nil {
        fmt.Println("json err ", err)
    }
    fmt.Println(string(b))

    // 格式化输出
    b, err = json.MarshalIndent(p, "", "     ")
    if err != nil {
        fmt.Println("json err ", err)
    }
    fmt.Println(string(b))
}
```

struct tag

```
    type Person struct {
        //"-"是忽略的意思
        Name  string `json:"-"`
        Hobby string `json:"hobby" `
    }
```

示例通过map生成json

```
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    student := make(map[string]interface{})
    student["name"] = "51mh.com"
    student["age"] = 18
    student["sex"] = "man"
    b, err := json.Marshal(student)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(b)
}
```

- 解码json使用json.Unmarshal()函数可以对一组数据进行JSON格式的解码

```
    func Unmarshal(data []byte, v interface{}) error
```

示例解析到结构体

```
package main

import (
    "encoding/json"
    "fmt"
)

type Person struct {
    Age       int    `json:"age,string"`
    Name      string `json:"name"`
    Niubility bool   `json:"niubility"`
}

func main() {
    // 假数据
    b := []byte(`{"age":"18","name":"51mh.com","marry":false}`)
    var p Person
```

```
    err := json.Unmarshal(b, &p)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Println(p)
}
```

示例解析到interface

```
package main

import (
    "encoding/json"
    "fmt"
)

func main() {
    // 假数据
    // int和float64都当float64
    b := []byte(`{"age":1.3,"name":"5lmh.com","marry":false}`)

    // 声明接口
    var i interface{}
    err := json.Unmarshal(b, &i)
    if err != nil {
        fmt.Println(err)
    }
    // 自动转到map
    fmt.Println(i)
    // 可以判断类型
    m := i.(map[string]interface{})
    for k, v := range m {
        switch vv := v.(type) {
        case float64:
            fmt.Println(k, "是float64类型", vv)
        case string:
            fmt.Println(k, "是string类型", vv)
        default:
            fmt.Println("其他")
        }
    }
}
```

XML

- 是可扩展标记语言，包含声明、根标签、子元素和属性
- 应用场景：配置文件以及webService

示例：

```
    <?xml version="1.0" encoding="UTF-8" ?>
    <servers version="1">
        <server>
            <serverName>Shanghai_VPN</serverName>
            <serverIP>127.0.0.1</serverIP>
        </server>
        <server>
            <serverName>Beijing_VPN</serverName>
            <serverIP>127.0.0.2</serverIP>
        </server>
    </servers>
```

```
package main

import (
   "encoding/xml"
   "fmt"
   "io/ioutil"
)

// 抽取单个server对象
type Server struct {
   ServerName string `xml:"serverName"`
   ServerIP   string `xml:"serverIP"`
}

type Servers struct {
   Name    xml.Name `xml:"servers"`
   Version int     `xml:"version"`
   Servers []Server `xml:"server"`
}

func main() {
   data, err := ioutil.ReadFile("D:/my.xml")
   if err != nil {
      fmt.Println(err)
      return
   }
   var servers Servers
```

```
    err = xml.Unmarshal(data, &servers)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("xml: %#v\n", servers)
}
```

MSGPack

- MSGPack是二进制的json，性能更快，更省空间
- 需要安装第三方包：go get -u github.com/vmihailenco/msgpack

```
package main

import (
    "fmt"
    "github.com/vmihailenco/msgpack"
    "io/ioutil"
    "math/rand"
)

type Person struct {
    Name string
    Age  int
    Sex  string
}

// 二进制写出
func writerJson(filename string) (err error) {
    var persons []*Person
    // 假数据
    for i := 0; i < 10; i++ {
        p := &Person{
            Name: fmt.Sprintf("name%d", i),
            Age:  rand.Intn(100),
            Sex:  "male",
        }
        persons = append(persons, p)
    }
    // 二进制json序列化
    data, err := msgpack.Marshal(persons)
    if err != nil {
        fmt.Println(err)
```

```
        return
    }
    err = ioutil.WriteFile(filename, data, 0666)
    if err != nil {
        fmt.Println(err)
        return
    }
    return
}

// 二进制读取
func readJson(filename string) (err error) {
    var persons []*Person
    // 读文件
    data, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    // 反序列化
    err = msgpack.Unmarshal(data, &persons)
    if err != nil {
        fmt.Println(err)
        return
    }
    for _, v := range persons {
        fmt.Printf("%#v\n", v)
    }
    return
}

func main() {
    //err := writerJson("D:/person.dat")
    //if err != nil {
    // fmt.Println(err)
    // return
    //}
    err := readJson("D:/person.dat")
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

反射

反射是指在程序运行期对程序本身进行访问和修改的能力

变量的内在机制

- 变量包含类型信息和值信息 var arr [10]int arr[0] = 10
- 类型信息：是静态的元信息，是预先定义好的
- 值信息：是程序运行过程中动态改变的

反射的使用

- reflect包封装了反射相关的方法
- 获取类型信息：reflect.TypeOf，是静态的
- 获取值信息：reflect.ValueOf，是动态的

空接口与反射

- 反射可以在运行时动态获取程序的各种详细信息
- 反射获取interface类型信息

```
package main

import (
    "fmt"
    "reflect"
)

//反射获取interface类型信息

func reflect_type(a interface{}) {
    t := reflect.TypeOf(a)
    fmt.Println("类型是：", t)
    // kind()可以获取具体类型
    k := t.Kind()
    fmt.Println(k)
    switch k {
    case reflect.Float64:
        fmt.Printf("a is float64\n")
    case reflect.String:
```

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

func main() {
    var x float64 = 3.4
    reflect_type(x)
}
```

- 反射获取interface值信息

```
package main

import (
    "fmt"
    "reflect"
)

//反射获取interface值信息

func reflect_value(a interface{}) {
    v := reflect.ValueOf(a)
    fmt.Println(v)
    k := v.Kind()
    fmt.Println(k)
    switch k {
    case reflect.Float64:
        fmt.Println("a是：", v.Float())
    }
}

func main() {
    var x float64 = 3.4
    reflect_value(x)
}
```

- 反射修改值信息

```
package main

import (
    "fmt"
    "reflect"
)
```

```
//反射修改值
func reflect_set_value(a interface{}) {
    v := reflect.ValueOf(a)
    k := v.Kind()
    switch k {
    case reflect.Float64:
        // 反射修改值
        v.SetFloat(6.9)
        fmt.Println("a is ", v.Float())
    case reflect.Ptr:
        // Elem()获取地址指向的值
        v.Elem().SetFloat(7.9)
        fmt.Println("case:", v.Elem().Float())
        // 地址
        fmt.Println(v.Pointer())
    }
}

func main() {
    var x float64 = 3.4
    // 反射认为下面是指针类型，不是float类型
    reflect_set_value(&x)
    fmt.Println("main:", x)
}
```

结构体与反射

查看类型、字段和方法

```
package main

import (
    "fmt"
    "reflect"
)

// 定义结构体
type User struct {
    Id   int
    Name string
    Age  int
}

// 绑方法
```

```
func (u User) Hello() {
	fmt.Println("Hello")
}

// 传入interface{}
func Poni(o interface{}) {
	t := reflect.TypeOf(o)
	fmt.Println("类型：", t)
	fmt.Println("字符串类型：", t.Name())
	// 获取值
	v := reflect.ValueOf(o)
	fmt.Println(v)
	// 可以获取所有属性
	// 获取结构体字段个数：t.NumField()
	for i := 0; i < t.NumField(); i++ {
		// 取每个字段
		f := t.Field(i)
		fmt.Printf("%s : %v", f.Name, f.Type)
		// 获取字段的值信息
		// Interface()：获取字段对应的值
		val := v.Field(i).Interface()
		fmt.Println("val :", val)
	}
	fmt.Println("=================方法====================")
	for i := 0; i < t.NumMethod(); i++ {
		m := t.Method(i)
		fmt.Println(m.Name)
		fmt.Println(m.Type)
	}

}

func main() {
	u := User{1, "zs", 20}
	Poni(u)
}
```

查看匿名字段

```
package main

import (
	"fmt"
	"reflect"
)
```

```go
// 定义结构体
type User struct {
    Id   int
    Name string
    Age  int
}

// 匿名字段
type Boy struct {
    User
    Addr string
}

func main() {
    m := Boy{User{1, "zs", 20}, "bj"}
    t := reflect.TypeOf(m)
    fmt.Println(t)
    // Anonymous：匿名
    fmt.Printf("%#v\n", t.Field(0))
    // 值信息
    fmt.Printf("%#v\n", reflect.ValueOf(m).Field(0))
}
```

修改结构体的值

```go
package main

import (
    "fmt"
    "reflect"
)

// 定义结构体
type User struct {
    Id   int
    Name string
    Age  int
}

// 修改结构体值
func SetValue(o interface{}) {
    v := reflect.ValueOf(o)
    // 获取指针指向的元素
    v = v.Elem()
```

```
    // 取字段
    f := v.FieldByName("Name")
    if f.Kind() == reflect.String {
        f.SetString("kuteng")
    }
}

func main() {
    u := User{1, "5lmh.com", 20}
    SetValue(&u)
    fmt.Println(u)
}
```

调用方法

```
package main

import (
    "fmt"
    "reflect"
)

// 定义结构体
type User struct {
    Id   int
    Name string
    Age  int
}

func (u User) Hello(name string) {
    fmt.Println("Hello: ", name)
}

func main() {
    u := User{1, "5lmh.com", 20}
    v := reflect.ValueOf(u)
    // 获取方法
    m := v.MethodByName("Hello")
    // 构建一些参数
    args := []reflect.Value{reflect.ValueOf("6666")}
    // 没参数的情况下: var args2 []reflect.Value
    // 调用方法，需要传入方法的参数
    m.Call(args)
}
```

获取字段的tag

```
package main

import (
    "fmt"
    "reflect"
)

type Student struct {
    Name string `json:"name1" db:"name2"`
}

func main() {
    var s Student
    v := reflect.ValueOf(&s)
    // 类型
    t := v.Type()
    // 获取字段
    f := t.Elem().Field(0)
    fmt.Println(f.Tag.Get("json"))
    fmt.Println(f.Tag.Get("db"))
}
```

反射练习

- 任务：解析如下配置文件
  - 序列化：将结构体序列化为配置文件数据并保存到硬盘
  - 反序列化：将配置文件内容反序列化到程序的结构体
- 配置文件有server和mysql相关配置

```
#this is comment
;this a comment
;[]表示一个section
[server]
ip = 10.238.2.2
port = 8080

[mysql]
username = root
passwd = admin
database = test
host = 192.168.10.10
```

```
port = 8000
timeout = 1.2
```

代码地址：https://github.com/lu569368/Practise_reflex.git

文件操作

打开和关闭文件

os.Open()函数能够打开一个文件，返回一个 `*File` 和一个err。对得到的文件实例调用close()方法能够关闭文件。

```
package main

import (
    "fmt"
    "os"
)

func main() {
    // 只读方式打开当前目录下的main.go文件
    file, err := os.Open("./main.go")
    if err != nil {
        fmt.Println("open file failed!, err:", err)
        return
    }
    // 关闭文件
    file.Close()
}
```

为了防止文件忘记关闭，我们通常使用defer注册文件关闭语句。

读取文件

file.Read()

基本使用

Read方法定义如下：

```
    func (f *File) Read(b []byte) (n int, err error)
```

它接收一个字节切片，返回读取的字节数和可能的具体错误，读到文件末尾时会返回0和io.EOF。 举个例子：

```
func main() {
    // 只读方式打开当前目录下的main.go文件
    file, err := os.Open("./main.go")
    if err != nil {
        fmt.Println("open file failed!, err:", err)
        return
    }
    defer file.Close()
    // 使用Read方法读取数据
    var tmp = make([]byte, 128)
    n, err := file.Read(tmp)
    if err == io.EOF {
        fmt.Println("文件读完了")
        return
    }
    if err != nil {
        fmt.Println("read file failed, err:", err)
        return
    }
    fmt.Printf("读取了%d字节数据\n", n)
    fmt.Println(string(tmp[:n]))
}
```

循环读取

使用for循环读取文件中的所有数据。

```
func main() {
    // 只读方式打开当前目录下的main.go文件
    file, err := os.Open("./main.go")
    if err != nil {
        fmt.Println("open file failed!, err:", err)
        return
    }
    defer file.Close()
    // 循环读取文件
    var content []byte
    var tmp = make([]byte, 128)
    for {
        n, err := file.Read(tmp)
        if err == io.EOF {
            fmt.Println("文件读完了")
            break
        }
        if err != nil {
```

```
            fmt.Println("read file failed, err:", err)
            return
        }
        content = append(content, tmp[:n]...)
    }
    fmt.Println(string(content))
}
```

bufio读取文件

bufio是在file的基础上封装了一层API，支持更多的功能。

```
package main

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

// bufio按行读取示例
func main() {
    file, err := os.Open("./main.go")
    if err != nil {
        fmt.Println("open file failed, err:", err)
        return
    }
    defer file.Close()
    reader := bufio.NewReader(file)
    for {
        line, err := reader.ReadString('\n') //注意是字符
        if err == io.EOF {
            fmt.Println("文件读完了")
            break
        }
        if err != nil {
            fmt.Println("read file failed, err:", err)
            return
        }
        fmt.Print(line)
    }
}
```

ioutil读取整个文件

io/ioutil包的ReadFile方法能够读取完整的文件，只需要将文件名作为参数传入。

```
package main

import (
    "fmt"
    "io/ioutil"
)

// ioutil.ReadFile读取整个文件
func main() {
    content, err := ioutil.ReadFile("./main.go")
    if err != nil {
        fmt.Println("read file failed, err:", err)
        return
    }
    fmt.Println(string(content))
}
```

文件写入操作

os.OpenFile()函数能够以指定模式打开文件，从而实现文件写入相关功能。

```
    func OpenFile(name string, flag int, perm FileMode) (*File, error) {
        ...
    }
```

其中：

name：要打开的文件名 flag：打开文件的模式。 模式有以下几种：

模式	含义
os.O_WRONLY	只写
os.O_CREATE	创建文件
os.O_RDONLY	只读
os.O_RDWR	读写
os.O_TRUNC	清空

模式	含义
os.O_APPEND	追加

perm：文件权限，一个八进制数。r（读）04，w（写）02，x（执行）01。

Write和WriteString

```
func main() {
	file, err := os.OpenFile("xx.txt", os.O_CREATE|os.O_TRUNC|os.O_WRONLY, 0666)
	if err != nil {
		fmt.Println("open file failed, err:", err)
		return
	}
	defer file.Close()
	str := "hello go"
	file.Write([]byte(str))       //写入字节切片数据
	file.WriteString("hello go") //直接写入字符串数据
}
```

bufio.NewWriter

```
func main() {
	file, err := os.OpenFile("xx.txt", os.O_CREATE|os.O_TRUNC|os.O_WRONLY, 0666)
	if err != nil {
		fmt.Println("open file failed, err:", err)
		return
	}
	defer file.Close()
	writer := bufio.NewWriter(file)
	for i := 0; i < 10; i++ {
		writer.WriteString("hello word\n") //将数据先写入缓存
	}
	writer.Flush() //将缓存中的内容写入文件
}
```

ioutil.WriteFile

```
func main() {
	str := "hello word"
	err := ioutil.WriteFile("./xx.txt", []byte(str), 0666)
	if err != nil {
```

```go
		fmt.Println("write file failed, err:", err)
		return
	}
}
```

例子

copyFile

借助io.Copy()实现一个拷贝文件函数。

```go
// CopyFile 拷贝文件函数
func CopyFile(dstName, srcName string) (written int64, err error) {
	// 以读方式打开源文件
	src, err := os.Open(srcName)
	if err != nil {
		fmt.Printf("open %s failed, err:%v.\n", srcName, err)
		return
	}
	defer src.Close()
	// 以写|创建的方式打开目标文件
	dst, err := os.OpenFile(dstName, os.O_WRONLY|os.O_CREATE, 0644)
	if err != nil {
		fmt.Printf("open %s failed, err:%v.\n", dstName, err)
		return
	}
	defer dst.Close()
	return io.Copy(dst, src) //调用io.Copy()拷贝内容
}
func main() {
	_, err := CopyFile("dst.txt", "src.txt")
	if err != nil {
		fmt.Println("copy file failed, err:", err)
		return
	}
	fmt.Println("copy done!")
}
```

实现一个cat命令

使用文件操作相关知识，模拟实现linux平台cat命令的功能。

```go
package main

import (
	"bufio"
	"flag"
	"fmt"
	"io"
	"os"
)

// cat命令实现
func cat(r *bufio.Reader) {
	for {
		buf, err := r.ReadBytes('\n') //注意是字符
		if err == io.EOF {
			break
		}
		fmt.Fprintf(os.Stdout, "%s", buf)
	}
}

func main() {
	flag.Parse() // 解析命令行参数
	if flag.NArg() == 0 {
		// 如果没有参数默认从标准输入读取内容
		cat(bufio.NewReader(os.Stdin))
	}
	// 依次读取每个指定文件的内容并打印到终端
	for i := 0; i < flag.NArg(); i++ {
		f, err := os.Open(flag.Arg(i))
		if err != nil {
			fmt.Fprintf(os.Stdout, "reading from %s failed, err:%v\n", flag.Arg
(i), err)
			continue
		}
		cat(bufio.NewReader(f))
	}
}
```

go module

为什么需要依赖管理

- 最早的时候，Go所依赖的所有的第三方库都放在GOPATH这个目录下面。这就导致了同一个库只能保存一个版本的代码。如果不同的项目依赖同一个第三方的库的不同版本，应该怎么解决？

go module

go module是Go1.11版本之后官方推出的版本管理工具，并且从Go1.13版本开始，go module将是Go语言默认的依赖管理工具。

GO111MODULE

要启用go module支持首先要设置环境变量GO111MODULE，通过它可以开启或关闭模块支持，它有三个可选值：off、on、auto，默认值是auto。

- GO111MODULE=off禁用模块支持，编译时会从GOPATH和vendor文件夹中查找包。
- GO111MODULE=on启用模块支持，编译时会忽略GOPATH和vendor文件夹，只根据go.mod下载依赖。
- GO111MODULE=auto，当项目在$GOPATH/src外且项目根目录有go.mod文件时，开启模块支持。

简单来说，设置GO111MODULE=on之后就可以使用go module了，以后就没有必要在GOPATH中创建项目了，并且还能够很好的管理项目依赖的第三方包信息。

使用 go module 管理依赖后会在项目根目录下生成两个文件go.mod和go.sum。

GOPROXY

Go1.11之后设置GOPROXY命令为：

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

Go1.13之后GOPROXY默认值为 https://proxy.golang.org ，在国内是无法访问的，所以十分建议大家设置GOPROXY，这里我推荐使用goproxy.cn。

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

go mod命令

常用的go mod命令如下：

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

go get

在项目中执行go get命令可以下载依赖包，并且还可以指定下载的版本。

- 运行go get -u将会升级到最新的次要版本或者修订版本(x.y.z, z是修订版本号， y是次要版本号)
- 运行go get -u=patch将会升级到最新的修订版本
- 运行go get package@version将会升级到指定的版本号version

  如果下载所有依赖可以使用go mod download命令。

简单粗暴的使用go module

1.set GO111MODULE=on

2.SET GOPROXY=https://goproxy.cn (这是win环境下的) export GOPROXY=https://goproxy.cn (这是mac环境下的)

3.go mod init [包名] // 初始化项目(如果你是初始化项目直接 `go mod init` 就好了)

4.在你的目录文件下会生成go.mod和go.sum文件 go.mod 里面包含了你的所有的包！

5.在文件里面引入包名的时候有的编辑器会报错但是是可以正常编译的

6.下载包使用go get

7.修改包的版本号直接去go.mod文件修改然后go mod download

String

Golang提供了许多内置的字符串函数，这些函数可在处理字符串数据时帮助执行一些操作。Golang字符串函数是核心部分。使用此功能无需安装，仅需要导入“字符串”包。重要的Golang字符串函数列表如下：

1)Golang字符串包含功能[区分大小写]

您可以使用Contains（）在字符串中搜索特定的文本/字符串/字符。它返回true或false的输出。如果在字符串2中找到字符串1，则返回true。如果在字符串2中找不到字符串1，则返回false。

语法：

```
func Contains(s, substr string) bool
```

DEMO

```
package main
import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Contains("我是中国人", "中国"))    //true
    fmt.Println(strings.Contains("I like golang", "like"))    //true
    fmt.Println(strings.Contains("www.topgoer.com", "topgoer"))  //true
    fmt.Println(strings.Contains("www.TopgoEr.com", "topgoer"))  //false
    fmt.Println(strings.Contains("www.TopgoEr com", " "))    //true
}
```

输出结果

```
true
true
true
false
true
```

2)Golang ContainsAny()[区分大小写]

您可以使用ContainsAny（）在字符串中搜索特定的文本/字符串/字符。它返回true或false的输出。如果在字符串中找到字符的unicode，则它返回true，否则输出将为false。您可以在下面的程序中看到ContainsAny与Contains的比较。

句法：

```
func ContainsAny（s, chars string）bool
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ContainsAny("Golang", "g"))     //true
    fmt.Println(strings.ContainsAny("Golang", "l & a"))   //true
    fmt.Println(strings.ContainsAny("GolAng", "a"))       // false
    fmt.Println(strings.ContainsAny("Golang", "G"))       //true
    fmt.Println(strings.ContainsAny("GOLANG", "GOLANG"))   //true
    fmt.Println(strings.ContainsAny("GOLANG", "golang"))   // false
    fmt.Println(strings.ContainsAny("Shell-12541", "1"))  //true
    //  Contains vs ContainsAny
    fmt.Println(strings.ContainsAny("Shell-12541", "1-2")) // true
    fmt.Println(strings.Contains("Shell-12541", "1-2"))    // false
}
```

输出结果

```
true
true
false
true
true
false
true
true
false
```

3)Golang Count() [区分大小写]

此函数计算字符串中字符/字符串/文本的不重叠实例的数量。

语法:

```
func Count(s, sep string) int
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Count("topgoer", "t"))   //1
    fmt.Println(strings.Count("Topgoer", "T"))   //1
    fmt.Println(strings.Count("Topgoer", "M"))   //0
    fmt.Println(strings.Count("Topgoer", "goer"))   // 1
    fmt.Println(strings.Count("Topgoer", "wwwTopgoercom"))   // 0
    fmt.Println(strings.Count("Shell-25152", "-25"))   //1
    fmt.Println(strings.Count("Shell-25152", "-21"))   //0
    fmt.Println(strings.Count("test", ""))      // length of string + 1   5
    fmt.Println(strings.Count("test", " "))    //0
}
```

输出结果

```
1
1
0
1
0
1
0
5
0
```

4)Golang EqualFold() [不区分大小写]

使用EqualFold，您可以检查两个字符串是否相等。如果两个字符串相等，则返回输出true，如果两个字符串都不相等，则返回false。

语法

```
func EqualFold(s, t string) bool
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.EqualFold("Topgoer", "TOPGOER"))   //true
    fmt.Println(strings.EqualFold("Topgoer", "topgoer"))   //true
    fmt.Println(strings.EqualFold("Topgoer", "Topgoercom"))  //false
    fmt.Println(strings.EqualFold("Topgoer", "goer"))      //false
    fmt.Println(strings.EqualFold("Topgoer", "Topgoer & goer"))   //false
    fmt.Println(strings.EqualFold("Topgoer-1254", "topgoer-1254"))  //true
    fmt.Println(strings.EqualFold(" ", " "))  // single space both side   //true
    fmt.Println(strings.EqualFold(" ", "  "))  // double space right side  //fal
se
}
```

输出结果

```
true
true
false
false
false
true
true
false
```

5) Golang Fields()

Fields函数将一个或多个连续空白字符的每个实例周围的字符串分解为一个数组。

语法

```
func Fields(s string) []string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    testString := "I love my country"
    testArray := strings.Fields(testString)
    for _, v := range testArray {
        fmt.Println(v)
    }
}
```

输出结果

```
I
love
my
country
```

6）Golang FieldsFunc()

FieldsFunc函数在每次运行满足f（c）的Unicode代码点c时都将字符串s断开，并返回s的切片数组。您可以使用此功能按数字或特殊字符的每个点分割字符串。检查以下两个FieldsFunc示例：

语法

```
func FieldsFunc (s string, f func (rune bool) [] string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
    "unicode"
)

func main() {

    x := func(c rune) bool {
        return !unicode.IsLetter(c)
    }
    strArray := strings.FieldsFunc(`I love my country  - I,love?my!country
                                 I, love, my - country`,x)
    for _, v := range strArray {
        fmt.Println(v)
    }

    fmt.Println("\n*****************Split by number******************\n")

    y := func(c rune) bool {
        return unicode.IsNumber(c)
    }
    testArray := strings.FieldsFunc(`1 I love my country.2 I love my,country.3 I
-love my country.4 I love my?country`,y)
    for _, w := range testArray {
        fmt.Println(w)
    }
}
```

输出结果

```
I
love
my
country
I
love
my
country
I
love
my
```

```
country

*****************Split by number******************

 I love my country.
 I love my,country.
 I-love my country.
 I love my?country
```

7）Golang HasPrefix()

HasPrefix函数检查字符串s是否以指定的字符串开头。如果字符串S以前缀字符串开头，则返回true，否则返回false。

语法

```
func HasPrefix(s, prefix string) bool
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.HasPrefix("Topgoer", "Top"))    //true
    fmt.Println(strings.HasPrefix("Topgoer", "top"))    //false
    fmt.Println(strings.HasPrefix("Topgoer", "ccc"))    //false
    fmt.Println(strings.HasPrefix("Topgoer", ""))       //true
}
```

输出结果

```
true
false
false
true
```

8）Golang HasSuffix()

HasSuffix函数检查字符串s是否以指定的字符串结尾。如果字符串S以后缀字符串结尾，则返回true，否则返回false。

语法

```
func HasSuffix(s, prefix string) bool
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.HasSuffix("Topgoer", "goer"))    //true
    fmt.Println(strings.HasSuffix("Topgoer", "R"))       //false
    fmt.Println(strings.HasSuffix("Topgoer", "GOER"))    //false
    fmt.Println(strings.HasSuffix("123456", "456"))    //true
    fmt.Println(strings.HasSuffix("Topgoer", ""))      //true
}
```

输出结果

```
true
false
false
true
true
```

9）Golang Index()

Index功能可以搜索字符串中的特定文本。它仅通过匹配字符串中的特定文本即可工作。如果找到，则返回以0开头的特定位置。如果找不到，则返回-1。

语法

```
func Index(s, sep string) int
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Index("Topgoer", "goer"))    //true
    fmt.Println(strings.Index("Topgoer", "R"))       //false
    fmt.Println(strings.Index("Topgoer", "GOER"))    //false
    fmt.Println(strings.Index("123-456", "-"))    //true
    fmt.Println(strings.Index("Topgoer", ""))      //true
}
```

输出结果

```
3
-1
-1
3
0
```

10)Golang IndexAny()

IndexAny函数从string [left]中的chars [right]返回任何Unicode代码点的第一个实例的索引。它仅通过匹配字符串中的特定文本即可工作。如果找到，则返回以0开头的特定位置。如果找不到，则返回-1。

语法：

```
func IndexAny (s, chars string) int
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)
```

```
func main() {
	fmt.Println(strings.IndexAny("topgoer", "www"))
	fmt.Println(strings.IndexAny("topgoer", "ggg"))
	fmt.Println(strings.IndexAny("mobile", "one"))
	fmt.Println(strings.IndexAny("123456789", "4"))
	fmt.Println(strings.IndexAny("123456789", "0"))
}
```

输出结果

```
-1
3
1
3
-1
```

11）Golang IndexByte()

IndexByte函数返回字符串中第一个字符实例的索引。如果找到，则返回以0开头的特定位置。如果找不到，则返回-1。

语法：

```
func IndexByte(s string, c byte) int
```

DEMO

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	var s,t,u byte
	t = 't'
	fmt.Println(strings.IndexByte("Topgoer",t))
	fmt.Println(strings.IndexByte("topgoer",t))
	fmt.Println(strings.IndexByte("ogoer",t))
	s = 1
	fmt.Println(strings.IndexByte("5221-topgoer",s))
	u = '1'
```

```
    fmt.Println(strings.IndexByte("5221-topgoer",u))
}
```

输出结果

```
-1
0
-1
-1
3
```

12）Golang IndexRune()

IndexRune函数以字符串形式返回Unicode代码点r的第一个实例的索引。如果找到，则返回以0开头的特定位置。如果找不到，则返回-1。在下面的示例中，s，t和u变量类型声明为符文。

语法：

```
func IndexRune(s string, r rune) int
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var s,t,u rune
    t = 'T'
    fmt.Println(strings.IndexRune("Topgoer",t))
    fmt.Println(strings.IndexRune("topgoer",t))
    fmt.Println(strings.IndexRune("opgoer",t))
    s = 1
    fmt.Println(strings.IndexRune("5221-JAPAN",s))
    u = '1'
    fmt.Println(strings.IndexRune("5221-JAPAN",u))
}
```

输出结果

```
0
-1
-1
-1
3
```

13）Golang Join() [concatenate]

Join（）函数从切片的元素返回字符串。Join将字符串Slice的元素连接起来以创建单个字符串。分隔符字符串sep指定在结果字符串中的切片元素之间放置的内容。

语法

```
func Join(stringSlice []string, sep string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    // Slice of strings
    textString := []string{"wen", "topgoer", "com"}
    fmt.Println(strings.Join(textString, "-"))

    // Slice of strings
    textNum := []string{"1", "2", "3", "4", "5"}
    fmt.Println(strings.Join(textNum,""))
}
```

输出结果

```
wen-topgoer-com
12345
```

14）Golang LastIndex() [区分大小写]

LastIndex函数可在字符串中搜索特定的特定文本/字符/字符串。它返回字符串中最后一个实例text / character / strin的索引。如果找到，则返回以0开头的特定位置。如果找不到，则返回-1。

语法

```
func LastIndex(s, sep string) int
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.LastIndex("topgoer", "o")) // position j=0,a=1,p=2,a=3
    fmt.Println(strings.LastIndex("topgoer", "G"))
    fmt.Println(strings.LastIndex("Topgoer", "go"))
    fmt.Println(strings.LastIndex("TOPGOER TOPGOER", "go"))
    fmt.Println(strings.LastIndex("1234567890 1234567890", "0"))
    fmt.Println(strings.LastIndex("1234567890 1234567890", "00"))
    fmt.Println(strings.LastIndex("1234567890 1234567890", "123"))
}
```

输出结果

```
4
-1
3
-1
20
-1
11
```

15）Golang Repeat()

Repeat函数将字符串重复指定的次数，并返回一个新字符串，该字符串由字符串s的计数副本组成。Count指定将重复字符串的次数。必须大于或等于0。

语法

```
func Repeat(s string, count int) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    textString := "China"
    repString := strings.Repeat(textString, 5)
    fmt.Println(repString)

    textString = " A " // char with space on both side
    repString = strings.Repeat(textString,5)
    fmt.Println(repString)        // Repeat space also

    fmt.Println("ba" + strings.Repeat("na", 2))
    fmt.Println("111" + strings.Repeat("22", 2))
    fmt.Println("111" + strings.Repeat(" ",2))
}
```

输出结果

```
ChinaChinaChinaChinaChina
 A  A  A  A  A
banana
1112222
111
```

16)Golang Replace() [区分大小写]

替换功能用字符串中的某些其他字符替换某些字符。n指定要在字符串中替换的字符数。如果n小于0，则替换次数没有限制。

语法

```
func Replace(s, old, new string, n int) string
```

DEMO

```
package main

import (
	"fmt"
	"strings"
)

func main() {

	fmt.Println(strings.Replace("Australia Japan Canada Indiana", "an", "anese", 0))
	fmt.Println(strings.Replace("Australia Japan Canada Indiana", "an", "anese", 1))
	fmt.Println(strings.Replace("Australia Japan Canada Indiana", "an", "anese", 2))
	fmt.Println(strings.Replace("Australia Japan Canada Indiana", "an", "anese", -1))
	fmt.Println(strings.Replace("1234534232132", "1", "0", -1))
	// case-sensitive
	fmt.Println(strings.Replace("Australia Japan Canada Indiana", "AN", "anese", -1))
}
```

输出结果

```
Australia Japan Canada Indiana
Australia Japanese Canada Indiana
Australia Japanese Caneseada Indiana
Australia Japanese Caneseada Indianesea
```

17) Golang Split() [区分大小写]

拆分功能将字符串拆分为一个切片。将s字符串拆分为用sep分隔的所有子字符串，并返回这些分隔符之间的子字符串的一部分。

语法

```
func Split(S string, sep string) []string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    strSlice := strings.Split("a,b,c", ",")
    fmt.Println(strSlice,"\n")

    strSlice = strings.Split("I love my country", " ")
    for _, v := range strSlice {
        fmt.Println(v)
    }

    strSlice = strings.Split("abacadaeaf","a")
    fmt.Println("\n",strSlice)

    strSlice = strings.Split("abacadaeaf","A")
    fmt.Println("\n",strSlice)

    strSlice = strings.Split("123023403450456056706780789","0")
    fmt.Println("\n",strSlice)

    strSlice = strings.Split("123023403450456056706780789",",")
    fmt.Println("\n",strSlice)
}
```

输出结果

```
[a b c]

I
love
my
country

 [ b c d e f]

 [abacadaeaf]

 [123 234 345 456 567 678 789]
```

```
[1230234034504560567067807089]
```

18）Golang SplitN() [区分大小写]

SplitN函数将字符串分成片。SplitN将s字符串拆分为所有由sep分隔的子字符串，并返回这些分隔符之间的子字符串的一部分。n确定要返回的子字符串数。
n小于0：最多n个子字符串；最后一个子字符串将是未拆分的余数。
n等于0：结果为nil（零子字符串）
n大于0：所有子字符串

语法

```
func SplitN(s, sep string, n int) []string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    strSlice := strings.SplitN("a,b,c", ",",0)
    fmt.Println(strSlice,"\n")

    strSlice = strings.SplitN("a,b,c", ",",1)
    fmt.Println(strSlice,"\n")

    strSlice = strings.SplitN("a,b,c", ",",2)
    fmt.Println(strSlice,"\n")

    strSlice = strings.SplitN("a,b,c", ",",3)
    fmt.Println(strSlice,"\n")

    strSlice = strings.SplitN("I love my country", " ",-1)
    for _, v := range strSlice {
        fmt.Println(v)
    }
    strSlice = strings.SplitN("1230234034504560567067807089","0",5)
    fmt.Println("\n",strSlice)
}
```

输出结果

```
[]

[a, b, c]

[a  b, c]

[a  b  c]

I
love
my
country

 [123 234 345 456 56706780789]
```

19）Golang SplitAfter() [区分大小写]

SplitAfter函数将字符串分成片。在Sep的每个实例之后，SplitAfter将S切片为所有子字符串，并返回这些子字符串的切片。

语法

```
func SplitAfter(S String,  sep string) []string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    strSlice := strings.SplitAfter("a,b,c", ",")
    fmt.Println(strSlice,"\n")

    strSlice = strings.SplitAfter("I love my country", " ")
    for _, v := range strSlice {
```

```
        fmt.Println(v)
    }

    strSlice = strings.SplitAfter("abacadaeaf","a")
    fmt.Println("\n",strSlice)

    strSlice = strings.SplitAfter("abacadaeaf","A")
    fmt.Println("\n",strSlice)

    strSlice = strings.SplitAfter("123023403450456056706780789","0")
    fmt.Println("\n",strSlice)

    strSlice = strings.SplitAfter("123023403450456056706780789",",")
    fmt.Println("\n",strSlice)
}
```

输出结果

```
[a, b, c]

I
love
my
country

 [a ba ca da ea f]

 [abacadaeaf]

 [1230 2340 3450 4560 5670 6780 789]

 [123023403450456056706780789]
```

20）Golang SplitAfterN()[区分大小写]

SplitAfterN函数将字符串分成片。SplitAfterN在sep的每个实例之后将String切片为子字符串，并返回这些子字符串的切片。n确定要返回的子字符串数。
n小于0：最多n个子字符串；最后一个子字符串将是未拆分的余数。
n等于0：结果为nil（零子字符串）
n大于0：所有子字符串

语法

```
func SplitAfterN(string s, sep string, n int) []string
```

DEMO

```
package main

import (
	"fmt"
	"strings"
)

func main() {
	strSlice := strings.SplitAfterN("a,b,c", ",",0)
	fmt.Println(strSlice,"\n")

	strSlice = strings.SplitAfterN("a,b,c", ",",1)
	fmt.Println(strSlice,"\n")

	strSlice = strings.SplitAfterN("a,b,c", ",",2)
	fmt.Println(strSlice,"\n")

	strSlice = strings.SplitAfterN("a,b,c", ",",3)
	fmt.Println(strSlice,"\n")

	strSlice = strings.SplitAfterN("I love my country", " ",-1)
	for _, v := range strSlice {
		fmt.Println(v)
	}
	strSlice = strings.SplitAfterN("123023403450456056706780789","0",5)
	fmt.Println("\n",strSlice)
}
```

输出结果

```
[]

[a,b,c]

[a, b,c]

[a, b, c]

I
```

```
love
my
country

[1230 2340 3450 4560 56706780789]
```

21）Golang Title()

Title函数将每个单词的第一个字符转换为大写。

语法

```
func Title(s string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Title("i like golang"))
    fmt.Println(strings.Title("i love my country"))
    fmt.Println(strings.Title("topgoer com"))
}
```

输出结果

```
I Like Golang
I Love My Country
Topgoer Com
```

22）Golang ToTitle()

ToTitle函数将每个单词的所有字符转换为大写。

语法

```
func ToTitle(s string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ToTitle("i like golang"))
    fmt.Println(strings.ToTitle("i love my country"))
    fmt.Println(strings.ToTitle("topgoer com"))
}
```

输出结果

```
I LIKE GOLANG
I LOVE MY COUNTRY
TOPGOER COM
```

23）Golang ToLower()

ToTitle函数将每个单词的所有字符转换为小写。

语法

```
func ToLower(s string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ToLower("I Like Golang"))
    fmt.Println(strings.ToLower("I Love My Country"))
```

```
    fmt.Println(strings.ToLower("topgoer Com"))
}
```

输出结果

```
i like golang
i love my country
topgoer com
```

24）Golang ToUpper()

ToUpper函数将每个单词的所有字符转换为大写。

语法

```
func ToUpper(s string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.ToUpper("I Like Golang"))
    fmt.Println(strings.ToUpper("I Love My Country"))
    fmt.Println(strings.ToUpper("topgoer Com"))
}
```

输出结果

```
I LIKE GOLANG
I LOVE MY COUNTRY
TOPGOER COM
```

25)Golang Trim() [区分大小写]

Trim函数从字符串的两边删除预定义的字符cutset。

语法

```
func Trim(s string, cutset string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.Trim("0120 2510", "0"))
    fmt.Println(strings.Trim("abcd axyz", "a"))
    fmt.Println(strings.Trim("abcd axyz", "A"))
    fmt.Println(strings.Trim("! Abcd dcbA !", "A"))
    fmt.Println(strings.Trim("! Abcd dcbA !", "!"))
    fmt.Println(strings.Trim(" Abcd dcbA ", " "))
}
```

输出结果

```
120 251
bcd axyz
abcd axyz
! Abcd dcbA !
 Abcd dcbA
Abcd dcbA
```

26)Golang TrimLeft() [区分大小写]

trimlefit函数只从字符串s的左侧删除预定义字符cutset。

语法

```
func TrimLeft(s string, cutset string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    fmt.Println(strings.TrimLeft("0120 2510", "0"))
    fmt.Println(strings.TrimLeft("abcd axyz", "a"))
    fmt.Println(strings.TrimLeft("abcd axyz", "A"))
    fmt.Println(strings.TrimLeft("! Abcd dcbA !", "A"))
    fmt.Println(strings.TrimLeft("! Abcd dcbA !", "!"))
    fmt.Println(strings.TrimLeft(" Abcd dcbA ", " "))
}
```

输出结果

```
120 2510
bcd axyz
abcd axyz
! Abcd dcbA !
 Abcd dcbA !
Abcd dcbA
```

27）Golang TrimRight() [区分大小写]

TrimRight函数仅从字符串s的右侧删除预定义字符cutset。

语法

```
func TrimRight(s string, cutset string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)
```

```
func main() {
    fmt.Println(strings.TrimRight("0120 2510", "0"))
    fmt.Println(strings.TrimRight("abcd axyz", "a"))
    fmt.Println(strings.TrimRight("abcd axyz", "A"))
    fmt.Println(strings.TrimRight("! Abcd dcbA !", "A"))
    fmt.Println(strings.TrimRight("! Abcd dcbA !", "!"))
    fmt.Println(strings.TrimRight(" Abcd dcbA ", " "))
}
```

输出结果

```
0120 251
abcd axyz
abcd axyz
! Abcd dcbA !
! Abcd dcbA
 Abcd dcbA
```

28)Golang TrimSpace()

TrimSpace函数从字符串的两侧删除空白和其他预定义字符。

“\t”-选项卡

“\n”-新行

“\x0B”-垂直选项卡

“\r”-回车

“”-普通空白

语法

```
func TrimSpace(s string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)
```

```
func main() {
    fmt.Println(strings.TrimSpace(" I love my country "))
    fmt.Println(strings.TrimSpace(" \t\n  I love my country \t\n "))
    fmt.Println(strings.TrimSpace(" \t\n\r\x0BI love my country\t\n "))
}
```

输出结果

```
I love my country
I love my country
I love my country
```

29）Golang TrimPrefi() [区分大小写]

TrimPrefix函数从S字符串的开头删除前缀字符串。如果S不以前缀开头，则S将原封不动地返回。

语法

```
func TrimPrefix(S string, prefix string) string
```

DEMO

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    var s string
    s = "I love my country"
    s = strings.TrimPrefix(s, "I")
    s = strings.TrimSpace(s)
    fmt.Println(s)

    s = "I love my country"
    s = strings.TrimPrefix(s, "i")
    fmt.Println(s)

    s = "\nI-love-my-country"
```

```
    s = strings.TrimPrefix(s, "\n")
    fmt.Println(s)

    s = "\tI-love-my-country"
    s = strings.TrimPrefix(s, "\t")
    fmt.Println(s)
}
```

输出结果

```
love my country
I love my country
I-love-my-country
I-love-my-country
```

beego框架

请跳转 http://www.topgoer.cn/docs/beegozhongwenwendang/beegozhongwenwendang-1c5087bb5qpst

gin框架

请跳转 http://www.topgoer.cn/docs/ginkuangjia/ginkuangjia-1c50hfaag99k2

Iris框架

请跳转 http://www.topgoer.cn/docs/Iris/Iris-1cao1k3mdlsbj

Echo框架

请跳转 http://www.topgoer.cn/docs/echo/echo-1c8r7vdjvmp71

微服务

认识微服务

微服务生态

微服务详解

RPC

RPC系统文档

Raft

gRPC

Go Micro入门

Go Micro接口详解

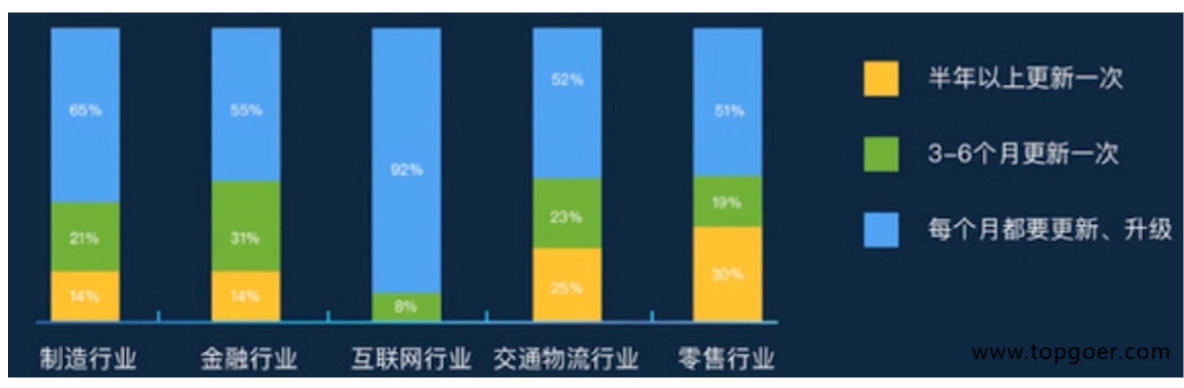
Go Micro文档1.x

Go Micro文档2.x

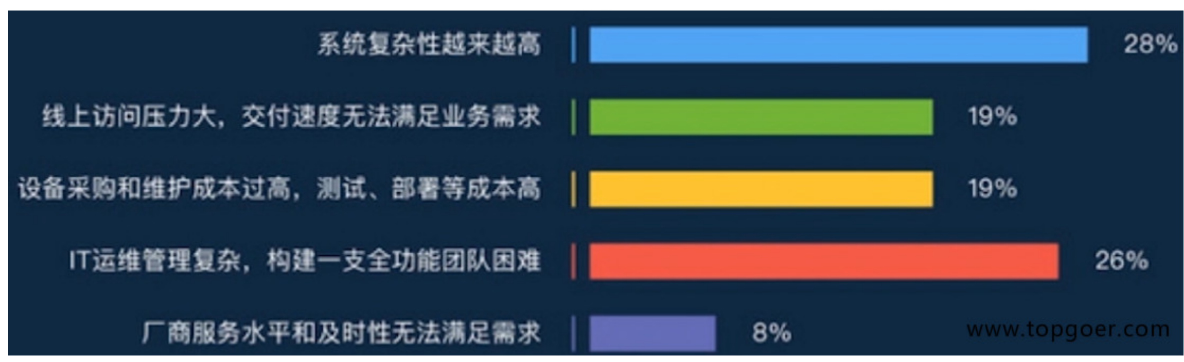
认识微服务

行业背景

不同行业IT系统更新频率

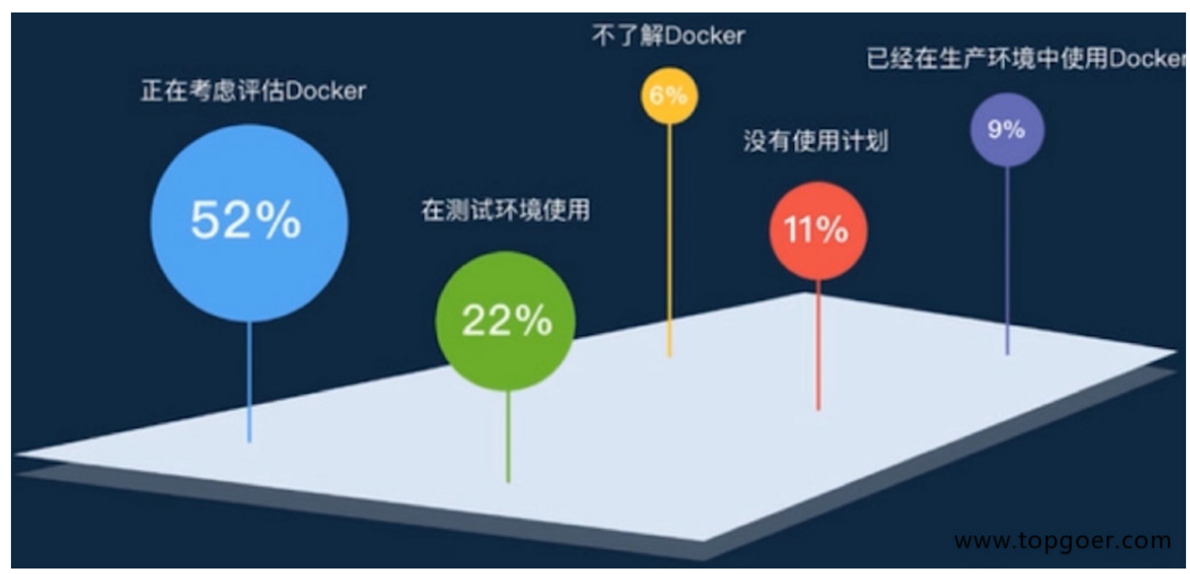


IT系统存在的问题



微服务架构在企业中应用情况

docker在企业中的使用情况



什么是微服务

- 使用一套小服务来开发单个应用的方式，每个服务运行在独立的进程里，一般采用轻量级的通讯机制互联，并且它们可以通过自动化的方式部署
- 什么叫微？
    - 单一功能
    - 代码少，不是，而且代码多
    - 架构变的复杂了
    - 微服务是设计思想，不是量的体现

微服务的特点

- 单一职责，此时项目专注于登录和注册
- 轻量级的通信，通信与平台和语言无关，http是轻量的，例如java的RMI属于重量的
- 隔离性，数据隔离
- 有自己的数据
- 技术多样性

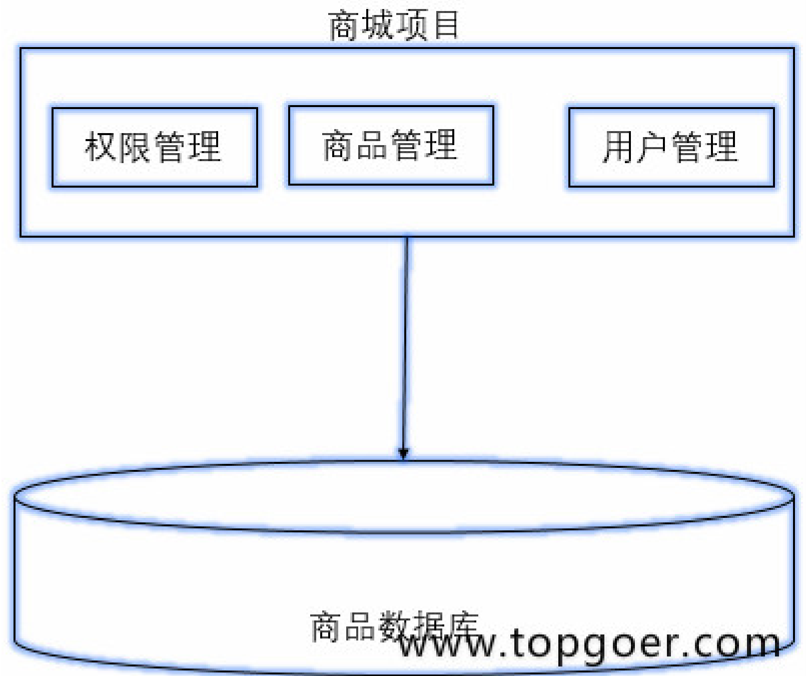
微服务诞生背景

- 互联网行业的快速发展，需求变化快，用户数量变化快
- 敏捷开发深入人心，用最小的代价，做最快的迭代，频繁修改、测试、上线
- 容器技术的成熟，是微服务的技术基础

互联网架构演进之路

单体架构

- 所有功能放一个项目里
- 应用和数据库服务器可能部在一起，分开部
- 优点：
    - 简单，高效，小型项目
- 缺点：
    - 扛不住
    - 技术栈受限



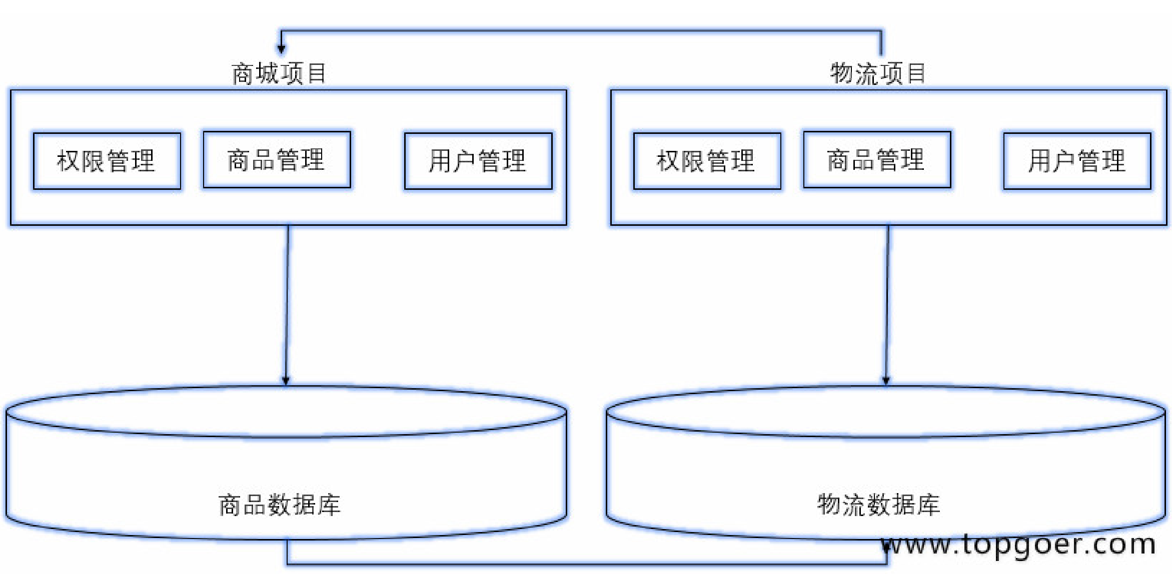
垂直架构

- 将大项目架构拆分成一个一个单体架构
- 优点：
    - 不至于像单体无限扩大
- 缺点：
    - 有瓶颈

- 成本高

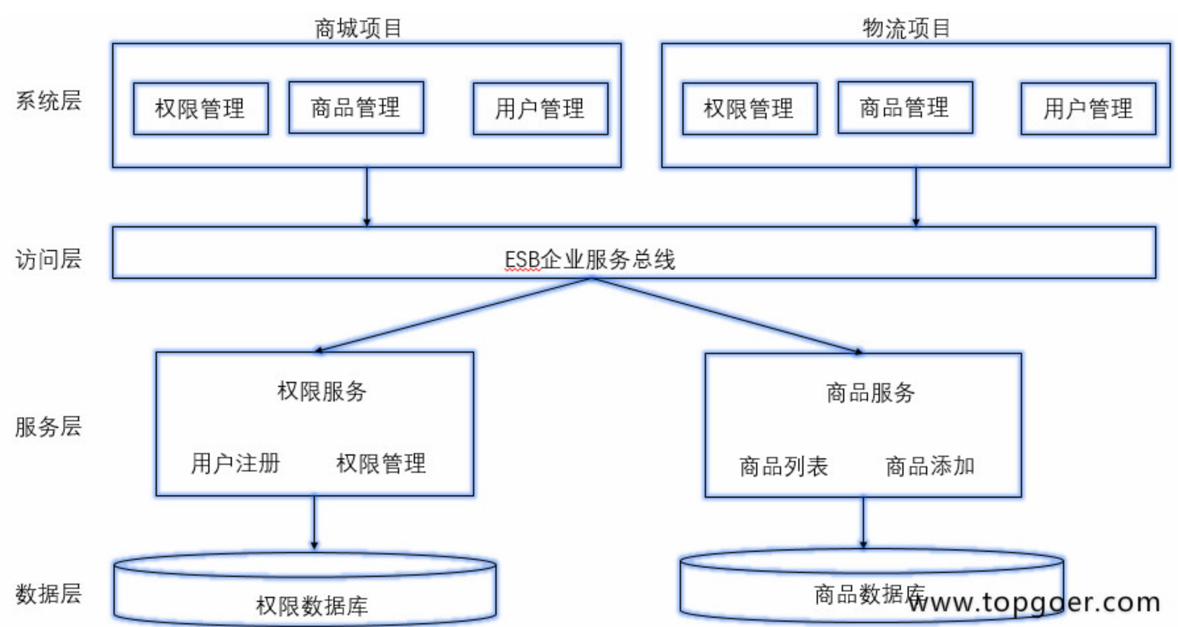
SOA架构，面向服务的编程

- ESB，比较传统的中间件技术
- 优点：
  - 代码提高重用性，ESB接口解耦
  - 针对不同服务，做不同数据层和部署
- 缺点：
  - ESB比较重量级
  - 对于开发人员来说，系统层和服务层界限模糊



微服务架构

- 每个功能抽取成一个一个的服务
- 微服务之间访问是轻量级的，RPC



微服务架构的优势

- 独立性
- 使用者容易理解
- 技术栈灵活
- 高效团队

微服务架构的不足

- 额外的工作，服务的拆分
- 保证数据一致性
- 增加了沟通成本

微服务生态

硬件层

- 用docker+k8s去解决

通信层

- 网络传输，用RPC（远程过程调用）
  - HTTP传输，GET POST PUT DELETE
  - 基于TCP，更靠底层，RPC基于TCP，Dubbo（18年底改成支持各种语言），Grpc，Thrift
- 需要知道调用谁，用服务注册和发现
  - 需要分布式数据同步：etcd，consul，zk

- 数据传递这里面可能是各种语言，各种技术，各种传递
- 效率对比：https://tech.meituan.com/2015/02/26/serialization-vs-deserialization.html

应用平台层

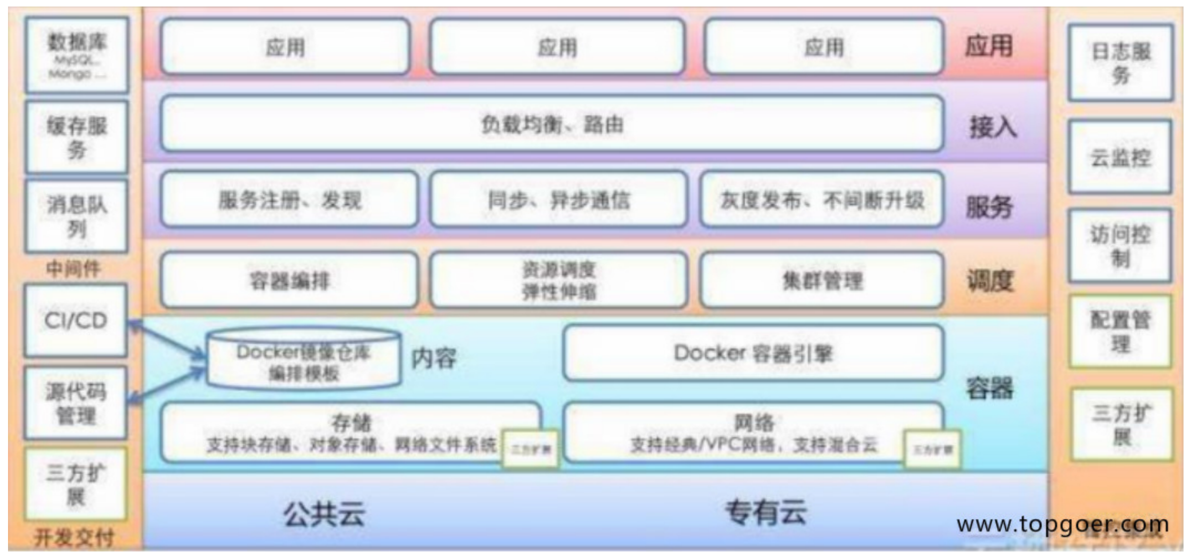
- 云管理平台、监控平台、日志管理平台，需要他们支持
- 服务管理平台，测试发布平台
- 服务治理平台

微服务层

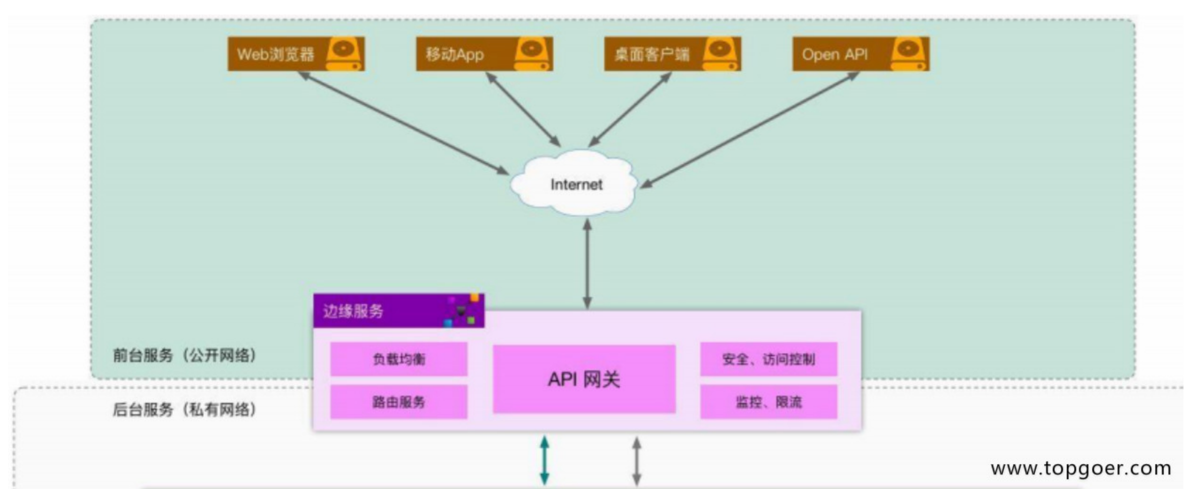
- 用微服务框架实现业务逻辑

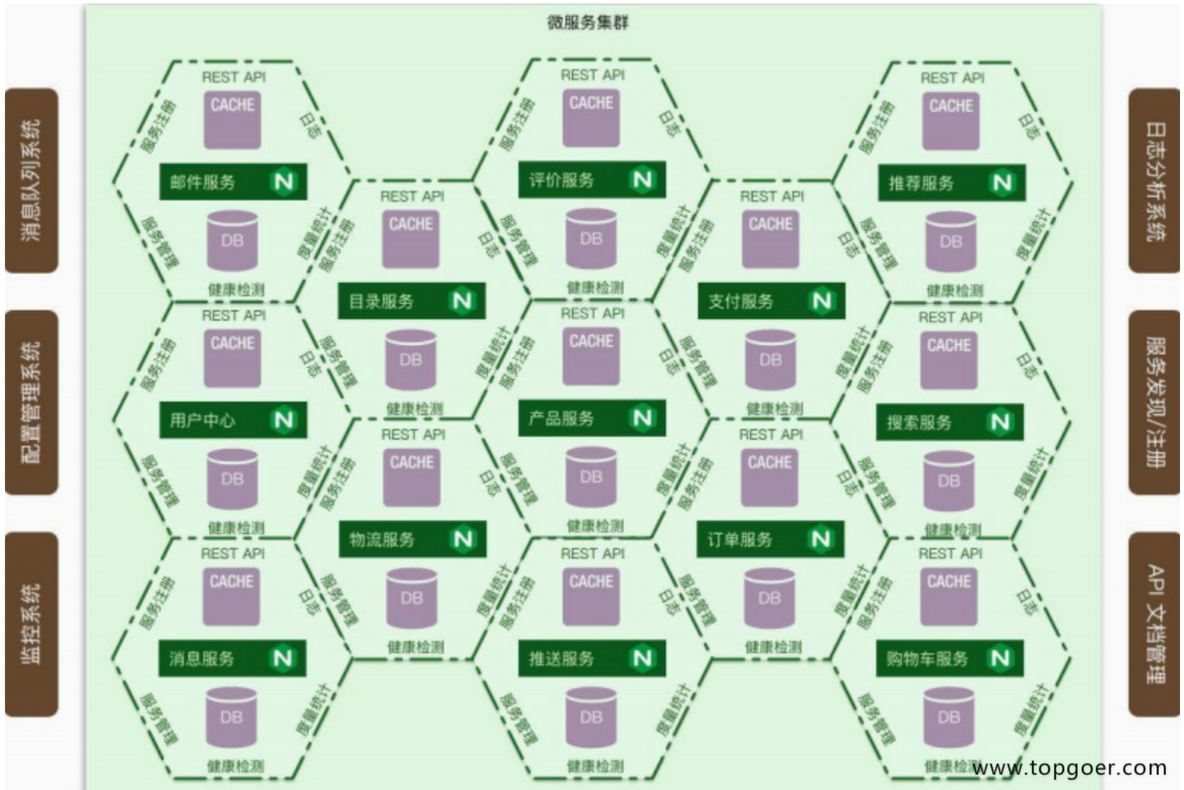
微服务详解

微服务架构



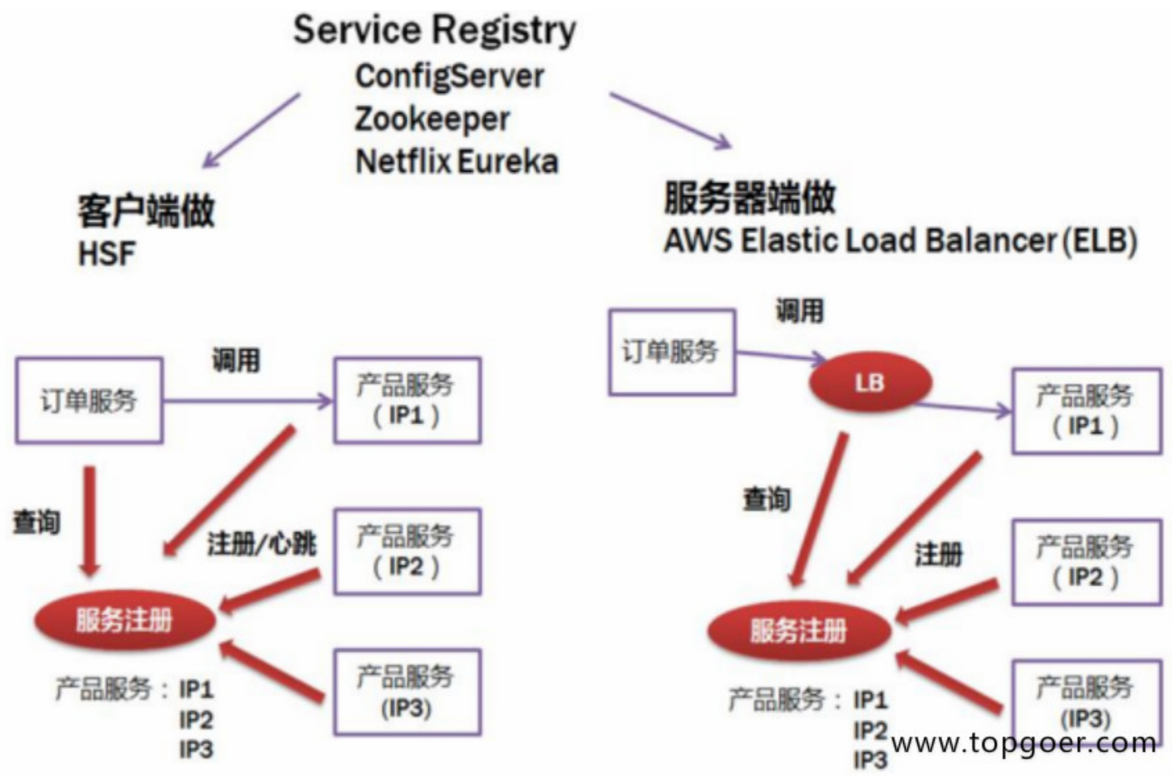
- 从程序架构来看如下





服务注册和发现

- 客户端做，需要实现一套注册中心，记录服务地址，知道具体访问哪个，轮询算法去做，加权轮询
- 服务端做，比较简单，服务端启动，自动注册即可，AWS的ELB去访问



- 微服务一般不用LVS负载，扩展实例需要改配置，不符合微服务弹性扩展思想
- 更多公司倾向于客户端做注册发现
- etcd解决分布式一致性，raft
- etcd使用场景：
  - 注册发现
  - 共享配置
  - 分布式锁
  - leader选举

rpc调用和服务监控

- RPC相关内容
  - 数据传输：JSON Protobuf thrift
  - 负载：随机算法 轮询 一致性hash 加权
  - 异常容错：健康检测 熔断 限流
- 服务监控
  - 日志收集
  - 打点采样

RPC

RPC简介

- 远程过程调用（Remote Procedure Call，RPC）是一个计算机通信协议
- 该协议允许运行于一台计算机的程序调用另一台计算机的子程序，而程序员无需额外地为这个交互作用编程
- 如果涉及的软件采用面向对象编程，那么远程过程调用亦可称作远程调用或远程方法调用

流行RPC框架的对比

RPC对比	Dubbo	Motan	Thrift	Grpc
开发语言	Java	Java	跨语言	跨语言
服务治理	√	√	×	×
多种序列化	√	√	只支持thrift	只支持protobuf
多种注册中心	√	√	×	×
管理中心	√	√	×	×
跨语言通讯	×	×	√	√

www.topgoer.com

golang中如何实现RPC

- golang中实现RPC非常简单，官方提供了封装好的库，还有一些第三方的库
- golang官方的net/rpc库使用encoding/gob进行编解码，支持tcp和http数据传输方式，由于其他语言不支持gob编解码方式，所以golang的RPC只支持golang开发的服务器与客户端之间的交互
- 官方还提供了net/rpc/jsonrpc库实现RPC方法，jsonrpc采用JSON进行数据编解码，因而支持跨语言调用，目前jsonrpc库是基于tcp协议实现的，暂不支持http传输方式
- 例题：golang实现RPC程序，实现求矩形面积和周长

服务端

```go
package main

import (
    "log"
    "net/http"
    "net/rpc"
)

//      例题：golang实现RPC程序，实现求矩形面积和周长

type Params struct {
    Width, Height int
}

type Rect struct{}

// RPC服务端方法，求矩形面积
func (r *Rect) Area(p Params, ret *int) error {
    *ret = p.Height * p.Width
    return nil
}

// 周长
func (r *Rect) Perimeter(p Params, ret *int) error {
    *ret = (p.Height + p.Width) * 2
    return nil
}

// 主函数
func main() {
    // 1.注册服务
    rect := new(Rect)
    // 注册一个rect的服务
    rpc.Register(rect)
    // 2.服务处理绑定到http协议上
    rpc.HandleHTTP()
    // 3.监听服务
    err := http.ListenAndServe(":8000", nil)
    if err != nil {
        log.Panicln(err)
    }
}
```

客户端

```go
package main

import (
    "fmt"
    "log"
    "net/rpc"
)

// 传的参数
type Params struct {
    Width, Height int
}

// 主函数
func main() {
    // 1.连接远程rpc服务
    conn, err := rpc.DialHTTP("tcp", ":8000")
    if err != nil {
        log.Fatal(err)
    }
    // 2.调用方法
    // 面积
    ret := 0
    err2 := conn.Call("Rect.Area", Params{50, 100}, &ret)
    if err2 != nil {
        log.Fatal(err2)
    }
    fmt.Println("面积：", ret)
    // 周长
    err3 := conn.Call("Rect.Perimeter", Params{50, 100}, &ret)
    if err3 != nil {
        log.Fatal(err3)
    }
    fmt.Println("周长：", ret)
}
```

- golang写RPC程序，必须符合4个基本条件，不然RPC用不了
    - 结构体字段首字母要大写，可以别人调用
    - 函数名必须首字母大写
    - 函数第一参数是接收参数，第二个参数是返回给客户端的参数，必须是指针类型

    - 函数还必须有一个返回值error

- 练习：模仿前面例题，自己实现RPC程序，服务端接收2个参数，可以做乘法运算，也可以做商和余数的运算，客户端进行传参和访问，得到结果如下：

服务端代码：

```
package main

import (
    "errors"
    "log"
    "net/http"
    "net/rpc"
)

// 结构体，用于注册的
type Arith struct{}

// 声明参数结构体
type ArithRequest struct {
    A, B int
}

// 返回给客户端的结果
type ArithResponse struct {
    // 乘积
    Pro int
    // 商
    Quo int
    // 余数
    Rem int
}

// 乘法
func (this *Arith) Multiply(req ArithRequest, res *ArithResponse) error {
    res.Pro = req.A * req.B
    return nil
}

// 商和余数
func (this *Arith) Divide(req ArithRequest, res *ArithResponse) error {
    if req.B == 0 {
        return errors.New("除数不能为0")
    }
```

```
    // 除
    res.Quo = req.A / req.B
    // 取模
    res.Rem = req.A % req.B
    return nil
}

// 主函数
func main() {
    // 1.注册服务
    rect := new(Arith)
    // 注册一个rect的服务
    rpc.Register(rect)
    // 2.服务处理绑定到http协议上
    rpc.HandleHTTP()
    // 3.监听服务
    err := http.ListenAndServe(":8000", nil)
    if err != nil {
        log.Fatal(err)
    }
}
```

客户端代码:

```
package main

import (
    "fmt"
    "log"
    "net/rpc"
)

type ArithRequest struct {
    A, B int
}

// 返回给客户端的结果
type ArithResponse struct {
    // 乘积
    Pro int
    // 商
    Quo int
    // 余数
    Rem int
}
```

```
func main() {
    conn, err := rpc.DialHTTP("tcp", ":8000")
    if err != nil {
        log.Fatal(err)
    }
    req := ArithRequest{9, 2}
    var res ArithResponse
    err2 := conn.Call("Arith.Multiply", req, &res)
    if err2 != nil {
        log.Fatal(err2)
    }
    fmt.Printf("%d * %d = %d\n", req.A, req.B, res.Pro)
    err3 := conn.Call("Arith.Divide", req, &res)
    if err3 != nil {
        log.Fatal(err3)
    }
    fmt.Printf("%d / %d 商 %d，余数 = %d\n", req.A, req.B, res.Quo, res.Rem)
}
```

> s="default">
>
> s="default">
>
> 另外，net/rpc/jsonrpc库通过json格式编解码，支持跨语言调用

服务端代码：

```
package main

import (
    "fmt"
    "log"
    "net"
    "net/rpc"
    "net/rpc/jsonrpc"
)

type Params struct {
    Width, Height int
}
type Rect struct {
}

func (r *Rect) Area(p Params, ret *int) error {
```

```
    *ret = p.Width * p.Height
    return nil
}
func (r *Rect) Perimeter(p Params, ret *int) error {
    *ret = (p.Height + p.Width) * 2
    return nil
}
func main() {
    rpc.Register(new(Rect))
    lis, err := net.Listen("tcp", ":8080")
    if err != nil {
        log.Panicln(err)
    }
    for {
        conn, err := lis.Accept()
        if err != nil {
            continue
        }
        go func(conn net.Conn) {
            fmt.Println("new client")
            jsonrpc.ServeConn(conn)
        }(conn)
    }
}
```

客户端代码:

```
package main

import (
    "fmt"
    "log"
    "net/rpc/jsonrpc"
)

type Params struct {
    Width, Height int
}

func main() {
    conn, err := jsonrpc.Dial("tcp", ":8080")
    if err != nil {
        log.Panicln(err)
    }
    ret := 0
```

```
    err2 := conn.Call("Rect.Area", Params{50, 100}, &ret)
    if err2 != nil {
        log.Panicln(err2)
    }
    fmt.Println("面积：", ret)
    err3 := conn.Call("Rect.Perimeter", Params{50, 100}, &ret)
    if err3 != nil {
        log.Panicln(err3)
    }
    fmt.Println("周长：", ret)
}
```

RPC调用流程

- 微服务架构下数据交互一般是对内 RPC，对外 REST
- 将业务按功能模块拆分到各个微服务，具有提高项目协作效率、降低模块耦合度、提高系统可用性等优点，但是开发门槛比较高，比如 RPC 框架的使用、后期的服务监控等工作
- 一般情况下，我们会将功能代码在本地直接调用，微服务架构下，我们需要将这个函数作为单独的服务运行，客户端通过网络调用

网络传输数据格式

- 两端要约定好数据包的格式
- 成熟的RPC框架会有自定义传输协议，这里网络传输格式定义如下，前面是固定长度消息头，后面是变长消息体

网络字节流：

header uint32	data []byte

www.topgoer.com

- 自己定义数据格式的读写

```
package rpc

import (
    "encoding/binary"
    "io"
    "net"
)

// 测试网络中读写数据的情况
```

```
// 会话连接的结构体
type Session struct {
    conn net.Conn
}

// 构造方法
func NewSession(conn net.Conn) *Session {
    return &Session{conn: conn}
}

// 向连接中去写数据
func (s *Session) Write(data []byte) error {
    // 定义写数据的格式
    // 4字节头部 + 可变体的长度
    buf := make([]byte, 4+len(data))
    // 写入头部，记录数据长度
    binary.BigEndian.PutUint32(buf[:4], uint32(len(data)))
    // 将整个数据，放到4后边
    copy(buf[4:], data)
    _, err := s.conn.Write(buf)
    if err != nil {
        return err
    }
    return nil
}

// 从连接读数据
func (s *Session) Read() ([]byte, error) {
    // 读取头部记录的长度
    header := make([]byte, 4)
    // 按长度读取消息
    _, err := io.ReadFull(s.conn, header)
    if err != nil {
        return nil, err
    }
    // 读取数据
    dataLen := binary.BigEndian.Uint32(header)
    data := make([]byte, dataLen)
    _, err = io.ReadFull(s.conn, data)
    if err != nil {
        return nil, err
    }
    return data, nil
}
```

测试类

```go
package rpc

import (
	"fmt"
	"net"
	"sync"
	"testing"
)

func TestSession_ReadWriter(t *testing.T) {
	// 定义地址
	addr := "127.0.0.1:8000"
	my_data := "hello"
	// 等待组定义
	wg := sync.WaitGroup{}
	wg.Add(2)
	// 写数据的协程
	go func() {
		defer wg.Done()
		lis, err := net.Listen("tcp", addr)
		if err != nil {
			t.Fatal(err)
		}
		conn, _ := lis.Accept()
		s := Session{conn: conn}
		err = s.Write([]byte(my_data))
		if err != nil {
			t.Fatal(err)
		}
	}()

	// 读数据的协程
	go func() {
		defer wg.Done()
		conn, err := net.Dial("tcp", addr)
		if err != nil {
			t.Fatal(err)
		}
		s := Session{conn: conn}
		data, err := s.Read()
		if err != nil {
			t.Fatal(err)
		}
		// 最后一层校验
```

```
        if string(data) != my_data {
            t.Fatal(err)
        }
        fmt.Println(string(data))
    }()
    wg.Wait()
}
```

编码解码

```
package rpc

import (
    "bytes"
    "encoding/gob"
)

// 定义RPC交互的数据结构
type RPCData struct {
    // 访问的函数
    Name string
    // 访问时的参数
    Args []interface{}
}

// 编码
func encode(data RPCData) ([]byte, error) {
    //得到字节数组的编码器
    var buf bytes.Buffer
    bufEnc := gob.NewEncoder(&buf)
    // 编码器对数据编码
    if err := bufEnc.Encode(data); err != nil {
        return nil, err
    }
    return buf.Bytes(), nil
}

// 解码
func decode(b []byte) (RPCData, error) {
    buf := bytes.NewBuffer(b)
    // 得到字节数组解码器
    bufDec := gob.NewDecoder(buf)
    // 解码器对数据节码
    var data RPCData
    if err := bufDec.Decode(&data); err != nil {
```

```
        return data, err
    }
    return data, nil
}
```

实现RPC服务端

- 服务端接收到的数据需要包括什么？
    - 调用的函数名、参数列表，还有一个返回值error类型
- 服务端需要解决的问题是什么？
    - Map维护客户端传来调用函数，服务端知道去调谁
- 服务端的核心功能有哪些？
    - 维护函数map
    - 客户端传来的东西进行解析
    - 函数的返回值打包，传给客户端

```
package rpc

import (
    "fmt"
    "net"
    "reflect"
)

// 声明服务端
type Server struct {
    // 地址
    addr string
    // map 用于维护关系的
    funcs map[string]reflect.Value
}

// 构造方法
func NewServer(addr string) *Server {
    return &Server{addr: addr, funcs: make(map[string]reflect.Value)}
}

// 服务端需要一个注册Register
```

```
// 第一个参数函数名，第二个传入真正的函数
func (s *Server) Register(rpcName string, f interface{}) {
	// 维护一个map
	// 若map已经有键了
	if _, ok := s.funcs[rpcName]; ok {
		return
	}
	// 若map中没值，则将映射加入map，用于调用
	fVal := reflect.ValueOf(f)
	s.funcs[rpcName] = fVal
}

// 服务端等待调用的方法
func (s *Server) Run() {
	// 监听
	lis, err := net.Listen("tcp", s.addr)
	if err != nil {
		fmt.Printf("监听 %s err :%v", s.addr, err)
		return
	}
	for {
		// 服务端循环等待调用
		conn, err := lis.Accept()
		if err != nil {
			return
		}
		serSession := NewSession(conn)
		// 使用RPC方式读取数据
		b, err := serSession.Read()
		if err != nil {
			return
		}
		// 数据解码
		rpcData, err := decode(b)
		if err != nil {
			return
		}
		// 根据读到的name，得到要调用的函数
		f, ok := s.funcs[rpcData.Name]
		if !ok {
			fmt.Println("函数 %s 不存在", rpcData.Name)
			return
		}
		// 遍历解析客户端传来的参数,放切片里
		inArgs := make([]reflect.Value, 0, len(rpcData.Args))
		for _, arg := range rpcData.Args {
```

```
            inArgs = append(inArgs, reflect.ValueOf(arg))
        }
        // 反射调用方法
        // 返回Value类型，用于给客户端传递返回结果,out是所有的返回结果
        out := f.Call(inArgs)
        // 遍历out ，用于返回给客户端，存到一个切片里
        outArgs := make([]interface{}, 0, len(out))
        for _, o := range out {
            outArgs = append(outArgs, o.Interface())
        }
        // 数据编码，返回给客户端
        respRPCData := RPCData{rpcData.Name, outArgs}
        bytes, err := encode(respRPCData)
        if err != nil {
            return
        }
        // 将服务端编码后的数据，写出到客户端
        err = serSession.Write(bytes)
        if err != nil {
            return
        }
    }
}
```

实现RPC客户端

- 客户端只有函数原型，使用reflect.MakeFunc() 可以完成原型到函数的调用
- reflect.MakeFunc()是Client从函数原型到网络调用的关键

```
package rpc

import (
    "net"
    "reflect"
)

// 声明服务端
type Client struct {
    conn net.Conn
}

// 构造方法
func NewClient(conn net.Conn) *Client {
```

```
    return &Client{conn: conn}
}

// 实现通用的RPC客户端
// 传入访问的函数名
// fPtr指向的是函数原型
//var select fun xx(User)
//cli.callRPC("selectUser",&select)
func (c *Client) callRPC(rpcName string, fPtr interface{}) {
    // 通过反射，获取fPtr未初始化的函数原型
    fn := reflect.ValueOf(fPtr).Elem()
    // 需要另一个函数，作用是对第一个函数参数操作
    f := func(args []reflect.Value) []reflect.Value {
        // 处理参数
        inArgs := make([]interface{}, 0, len(args))
        for _, arg := range args {
            inArgs = append(inArgs, arg.Interface())
        }
        // 连接
        cliSession := NewSession(c.conn)
        // 编码数据
        reqRPC := RPCData{Name: rpcName, Args: inArgs}
        b, err := encode(reqRPC)
        if err != nil {
            panic(err)
        }
        // 写数据
        err = cliSession.Write(b)
        if err != nil {
            panic(err)
        }
        // 服务端发过来返回值，此时应该读取和解析
        respBytes, err := cliSession.Read()
        if err != nil {
            panic(err)
        }
        // 解码
        respRPC, err := decode(respBytes)
        if err != nil {
            panic(err)
        }
        // 处理服务端返回的数据
        outArgs := make([]reflect.Value, 0, len(respRPC.Args))
        for i, arg := range respRPC.Args {
            // 必须进行nil转换
            if arg == nil {
```

```
                // reflect.Zero()会返回类型的零值的value
                // .out()会返回函数输出的参数类型
                outArgs = append(outArgs, reflect.Zero(fn.Type().Out(i)))
                continue
            }
            outArgs = append(outArgs, reflect.ValueOf(arg))
        }
        return outArgs
    }
    // 完成原型到函数调用的内部转换
    // 参数1是reflect.Type
    // 参数2 f是函数类型，是对于参数1 fn函数的操作
    // fn是定义，f是具体操作
    v := reflect.MakeFunc(fn.Type(), f)
    // 为函数fPtr赋值，过程
    fn.Set(v)
}
```

实现RPC通信测试

- 给服务端注册一个查询用户的方法，客户端使用RPC方式调用

```
package rpc

import (
    "encoding/gob"
    "fmt"
    "net"
    "testing"
)

//      给服务端注册一个查询用户的方法，客户端使用RPC方式调用

// 定义用户对象
type User struct {
    Name string
    Age  int
}

// 用于测试用户查询的方法
func queryUser(uid int) (User, error) {
    user := make(map[int]User)
    // 假数据
    user[0] = User{"zs", 20}
    user[1] = User{"ls", 21}
```

```
    user[2] = User{"ww", 22}
    // 模拟查询用户
    if u, ok := user[uid]; ok {
        return u, nil
    }
    return User{}, fmt.Errorf("%d err", uid)
}

func TestRPC(t *testing.T) {
    // 编码中有一个字段是interface{}时，要注册一下
    gob.Register(User{})
    addr := "127.0.0.1:8000"
    // 创建服务端
    srv := NewServer(addr)
    // 将服务端方法，注册一下
    srv.Register("queryUser", queryUser)
    // 服务端等待调用
    go srv.Run()
    // 客户端获取连接
    conn, err := net.Dial("tcp", addr)
    if err != nil {
        fmt.Println("err")
    }
    // 创建客户端对象
    cli := NewClient(conn)
    // 需要声明函数原型
    var query func(int) (User, error)
    cli.callRPC("queryUser", &query)
    // 得到查询结果
    u, err := query(1)
    if err != nil {
        fmt.Println("err")
    }
    fmt.Println(u)
}
```

RPC系统文档

请跳转 http://wen.topgoer.com/docs/rpc/rpc-1c5ck8luqjf3l

Raft

Raft是consoul和etcd的核心算法

Raft介绍

- Raft提供了一种在计算系统集群中分布状态机的通用方法，确保集群中的每个节点都同意一系列相同的状态转换
- 它有许多开源参考实现，具有Go，C ++，Java和Scala中的完整规范实现
- 一个Raft集群包含若干个服务器节点，通常是5个，这允许整个系统容忍2个节点的失效，每个节点处于以下三种状态之一
  - follower（跟随者） ：所有节点都以 follower 的状态开始。如果没收到 leader消息则会变成 candidate状态
  - candidate（候选人）：会向其他节点“拉选票”，如果得到大部分的票则成为leader，这个过程就叫做Leader选举(Leader Election)
  - leader（领导者）：所有对系统的修改都会先经过leader

Raft一致性算法

- Raft通过选出一个leader来简化日志副本的管理，例如，日志项(log entry)只允许从leader流向follower
- 基于leader的方法，Raft算法可以分解成三个子问题
  - Leader election (领导选举)：原来的leader挂掉后，必须选出一个新的leader
  - Log replication (日志复制)：leader从客户端接收日志，并复制到整个集群中
  - Safety (安全性)：如果有任意的server将日志项回放到状态机中了，那么其他的server只会回放相同的日志项

Raft动画演示

- 地址：http://thesecretlivesofdata.com/raft/
- 动画主要包含三部分：
  - 第一部分介绍简单版的领导者选举和日志复制的过程
  - 第二部分介绍详细版的领导者选举和日志复制的过程
  - 第三部分介绍如果遇到网络分区（脑裂），raft算法是如何恢复网络一致的

Leader election (领导选举)

- Raft 使用一种心跳机制来触发领导人选举
- 当服务器程序启动时，节点都是 follower(跟随者) 身份
- 如果一个跟随者在一段时间里没有接收到任何消息，也就是选举超时，然后他就会认为系统中没有可用的领导者然后开始进行选举以选出新的领导者
- 要开始一次选举过程，follower 会给当前term加1并且转换成candidate状态，然后它会并行的向集群中的其他服务器节点发送请求投票的 RPCs 来给自己投票。
- 候选人的状态维持直到发生以下任何一个条件发生的时候
    - 他自己赢得了这次的选举
    - 其他的服务器成为领导者
    - 一段时间之后没有任何一个获胜的人

Log replication (日志复制)

- 当选出 leader 后，它会开始接收客户端请求，每个请求会带有一个指令，可以被回放到状态机中
- leader 把指令追加成一个log entry，然后通过AppendEntries RPC并行地发送给其他的server，当该entry被多数server复制后，leader 会把该entry回放到状态机中，然后把结果返回给客户端
- 当 follower 宕机或者运行较慢时，leader 会无限地重发AppendEntries给这些follower，直到所有的follower都复制了该log entry
- raft的log replication要保证如果两个log entry有相同的index和term，那么它们存储相同的指令
- leader在一个特定的term和index下，只会创建一个log entry

gRPC

安装

gRPC简介

Protobuf⇢Go转换

Protobuf语法

小案例

OpenSSL安装

认证

拦截器

内置Trace

HTTP网关

安装

gRPC简介

- gRPC由google开发，是一款语言中立、平台中立、开源的远程过程调用系统
- gRPC客户端和服务端可以在多种环境中运行和交互，例如用java写一个服务端，可以用go语言写客户端调用

gRPC与Protobuf介绍

- 微服务架构中，由于每个服务对应的代码库是独立运行的，无法直接调用，彼此间的通信就是个大问题
- gRPC可以实现微服务，将大的项目拆分为多个小且独立的业务模块，也就是服务，各服务间使用高效的protobuf协议进行RPC调用，gRPC默认使用protocol buffers，这是google开源的一套成熟的结构数据序列化机制（当然也可以使用其他数据格式如JSON）
- 可以用proto files创建gRPC服务，用message类型来定义方法参数和返回类型

安装gRPC和Protobuf

- go get github.com/golang/protobuf/proto
- go get google.golang.org/grpc（无法使用，用如下命令代替）
    - git clone https://github.com/grpc/grpc-go.git $GOPATH/src/google.golang.org/grpc
    - git clone https://github.com/golang/net.git $GOPATH/src/golang.org/x/net
    - git clone https://github.com/golang/text.git $GOPATH/src/golang.org/x/text
    - go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
    - git clone https://github.com/google/go-genproto.git $GOPATH/src/google.golang.org/genproto
    - cd $GOPATH/src/
    - go install google.golang.org/grpc
- go get github.com/golang/protobuf/protoc-gen-go
- 上面安装好后，会在GOPATH/bin下生成protoc-gen-go.exe

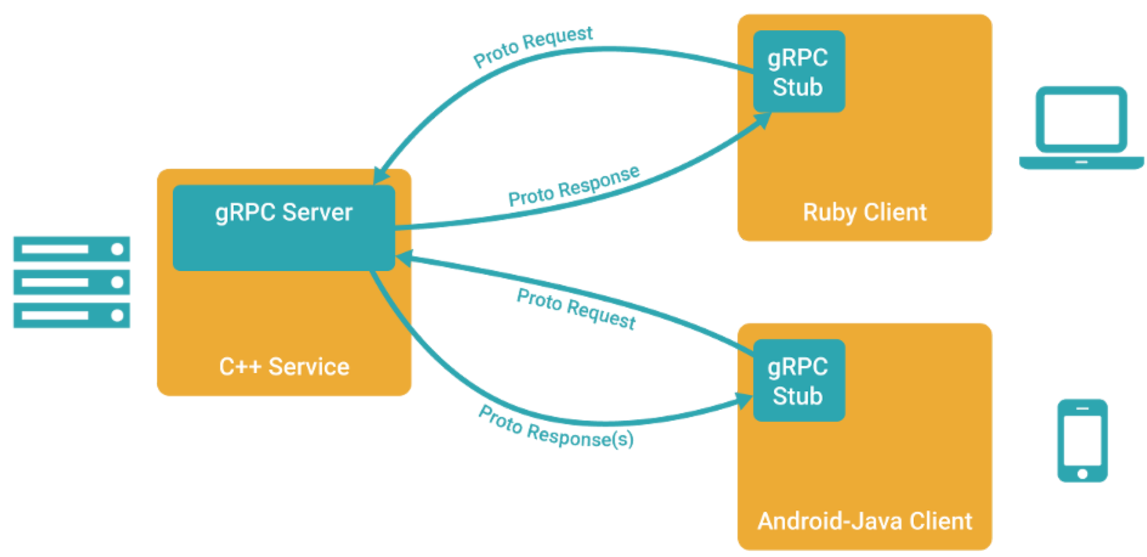
- 但还需要一个protoc.exe，windows平台编译受限，很难自己手动编译，直接去网站下载一个，地址：https://github.com/protocolbuffers/protobuf/releases/tag/v3.9.0 ，同样放在GOPATH/bin下

注意：这里面好多都是需要vpn才能下载好的！分享一个下载好的 https://pan.baidu.com/s/1T8eJkHib2uPL3gNMCRdZmQ 提取码 s42z

gRPC简介

gRPC 是一个高性能、开源、通用的RPC框架，由Google推出，基于HTTP2协议标准设计开发，默认采用Protocol Buffers数据序列化协议，支持多种开发语言。gRPC提供了一种简单的方法来精确的定义服务，并且为客户端和服务端自动生成可靠的功能库。

在gRPC客户端可以直接调用不同服务器上的远程程序，使用姿势看起来就像调用本地程序一样，很容易去构建分布式应用和服务。和很多RPC系统一样，服务端负责实现定义好的接口并处理客户端的请求，客户端根据接口描述直接调用需要的服务。客户端和服务端可以分别使用gRPC支持的不同语言实现。



主要特性

- 强大的IDL

  gRPC使用ProtoBuf来定义服务，ProtoBuf是由Google开发的一种数据序列化协议（类似于XML、JSON、hessian）。ProtoBuf能够将数据进行序列化，并广泛应用在数据存储、通信协议等方面。

- 多语言支持

gRPC支持多种语言，并能够基于语言自动生成客户端和服务端功能库。目前已提供了C版本grpc、Java版本grpc-java 和 Go版本grpc-go，其它语言的版本正在积极开发中，其中，grpc支持C、C++、Node.js、Python、Ruby、Objective-C、PHP和C#等语言，grpc-java已经支持Android开发。

- HTTP2

gRPC基于HTTP2标准设计，所以相对于其他RPC框架，gRPC带来了更多强大功能，如双向流、头部压缩、多复用请求等。这些功能给移动设备带来重大益处，如节省带宽、降低TCP链接次数、节省CPU使用和延长电池寿命等。同时，gRPC还能够提高了云端服务和Web应用的性能。gRPC既能够在客户端应用，也能够在服务器端应用，从而以透明的方式实现客户端和服务器端的通信和简化通信系统的构建。

更多介绍请查看官方网站

Protobuf⇢Go转换

这里使用一个测试文件对照说明常用结构的protobuf到golang的转换。只说明关键部分代码，详细内容请查看完整文件。示例文件在 `proto/test` 目录下。

Package

在proto文件中使用 `package` 关键字声明包名，默认转换成go中的包名与此一致，如果需要指定不一样的包名，可以使用 `go_package` 选项：

```
package test;
option go_package="test";
```

Message

proto中的 `message` 对应go中的 `struct` ，全部使用驼峰命名规则。嵌套定义的 `message` ， `enum` 转换为go之后，名称变为 `Parent_Child` 结构。

示例proto：

```
// Test 测试
message Test {
    int32 age = 1;
    int64 count = 2;
    double money = 3;
    float score = 4;
    string name = 5;
    bool fat = 6;
    bytes char = 7;
    // Status 枚举状态
    enum Status {
        OK = 0;
        FAIL = 1;
    }
    Status status = 8;
    // Child 子结构
    message Child {
        string sex = 1;
    }
    Child child = 9;
    map<string, string> dict = 10;
}
```

转换结果：

```
// Status 枚举状态
type Test_Status int32

const (
    Test_OK   Test_Status = 0
    Test_FAIL Test_Status = 1
)

// Test 测试
type Test struct {
    Age    int32          `protobuf:"varint,1,opt,name=age" json:"age,omitempty"`
    Count  int64          `protobuf:"varint,2,opt,name=count" json:"count,omitempty"`
    Money  float64        `protobuf:"fixed64,3,opt,name=money" json:"money,omitempty"`
    Score  float32        `protobuf:"fixed32,4,opt,name=score" json:"score,omitempty"`
    Name   string         `protobuf:"bytes,5,opt,name=name" json:"name,omitempty"`
    Fat    bool           `protobuf:"varint,6,opt,name=fat" json:"fat,omitempty"`
    Char   []byte         `protobuf:"bytes,7,opt,name=char,proto3" json:"char,omitempty"`
    Status Test_Status `protobuf:"varint,8,opt,name=status,enum=test.Test_Status" json:"status,omitempty"`
    Child  *Test_Child `protobuf:"bytes,9,opt,name=child" json:"child,omitempty"`
    Dict   map[string]string `protobuf:"bytes,10,rep,name=dict" json:"dict,omitempty" protobuf_key:"bytes,1,opt,name=key" protobuf_val:"bytes,2,opt,name=value"`
}

// Child 子结构
type Test_Child struct {
    Sex string `protobuf:"bytes,1,opt,name=sex" json:"sex,omitempty"`
}
```

除了会生成对应的结构外，还会有些工具方法，如字段的getter:

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

枚举类型会生成对应名称的常量，同时会有两个map方便使用：

```
var Test_Status_name = map[int32]string{
    0: "OK",
    1: "FAIL",
}
var Test_Status_value = map[string]int32{
    "OK":   0,
    "FAIL": 1,
}
```

Service

定义一个简单的Service， `TestService` 有一个方法 `Test` ，接收一个 `Request` 参数，返回 `Response` ：

```
// TestService 测试服务
service TestService {
    // Test 测试方法
    rpc Test(Request) returns (Response) {};
}

// Request 请求结构
message Request {
    string name = 1;
}

// Response 响应结构
message Response {
    string message = 1;
}
```

转换结果：

```
// 客户端接口
type TestServiceClient interface {
    // Test 测试方法
    Test(ctx context.Context, in *Request, opts ...grpc.CallOption) (*Response, error)
}

// 服务端接口
type TestServiceServer interface {
    // Test 测试方法
```

```
    Test(context.Context, *Request) (*Response, error)
}
```

生成的go代码中包含该Service定义的接口，客户端接口已经自动实现了，直接供客户端使用者调用，服务端接口需要由服务提供方实现。

Protobuf语法

基本规范

- 文件以.proto做为文件后缀，除结构定义外的语句以分号结尾
- 结构定义可以包含：message、service、enum
- rpc方法定义结尾的分号可有可无
- Message命名采用驼峰命名方式，字段命名采用小写字母加下划线分隔方式

```
message SongServerRequest {
    required string song_name = 1;
}
```

- Enums类型名采用驼峰命名方式，字段命名采用大写字母加下划线分隔方式

```
enum Foo {
    FIRST_VALUE = 1;
    SECOND_VALUE = 2;
}
```

- Service与rpc方法名统一采用驼峰式命名

字段规则

- 字段格式： `限定修饰符 | 数据类型 | 字段名称 | = | 字段编码值 | [字段默认值]`
- 限定修饰符包含 required\optional\repeated
    - Required: 表示是一个必须字段，必须相对于发送方，在发送消息之前必须设置该字段的值，对于接收方，必须能够识别该字段的意思。发送之前没有设置required字段或者无法识别required字段都会引发编解码异常，导致消息被丢弃
    - Optional：表示是一个可选字段，可选对于发送方，在发送消息时，可以有选择性的设置或者不设置该字段的值。对于接收方，如果能够识别可选字段就进行相应的处理，如果无法识别，则忽略该字段，消息中的其它字段正常处理。—因为optional字段的特性，很多接口在升级版本中都把后来添加的字段都统一的设置为optional字段，这样老的版本无需升级程序也可以正常的与新的软件进行通信，只不过新的字段无法识别而已，因为并不是每个节点都需要新的功能，因此可以做到按需升级和平滑过渡

    - Repeated：表示该字段可以包含0~N个元素。其特性和optional一样，但是每一次可以包含多个值。可以看作是在传递一个数组的值
- 数据类型
    - Protobuf定义了一套基本数据类型。几乎都可以映射到C++\Java等语言的基础数据类型

.proto	C++	Java	Python	Go	Ruby
double	double	double	float	float64	Float
float	float	float	float	float32	Float
int32	int32	int	int	int32	Fixnum or Bignu
int64	int64	long	ing/long[3]	int64	Bignum
uint32	uint32	int[1]	int/long[3]	uint32	Fixnum or Bignu
uint64	uint64	long[1]	int/long[3]	uint64	Bignum
sint32	int32	int	intj	int32	Fixnum or Bignu
sint64	int64	long	int/long[3]	int64	Bignum
fixed32	uint32	int[1]	int	uint32	Fixnum or Bignu
fixed64	uint64	long[1]	int/long[3]	uint64	Bignum
sfixed32	int32	int	int	int32	Fixnum or Bignu
sfixed64	int64	long	int/long[3]	int64	Bignum
bool	bool	boolean	boolean	bool	TrueClass/False
string	string	String	str/unicode[4]	string	String(UTF-8)
bytes	string	ByteString	str	[]byte	String(ASCII-8B

```
+ N 表示打包的字节并不是固定。而是根据数据的大小或者长度
+ 关于 fixed32 和int32的区别。fixed32的打包效率比int32的效率高，但是使用的空间一般比int32多。因此一个属于时间效率高，一个属于空间效率高
```

- 字段名称
  - 字段名称的命名与C、C++、Java等语言的变量命名方式几乎是相同的
  - protobuf建议字段的命名采用以下划线分割的驼峰式。例如 first_name 而不是 firstName
- 字段编码值
  - 有了该值，通信双方才能互相识别对方的字段，相同的编码值，其限定修饰符和数据类型必须相同，编码值的取值范围为 `1~2^32` （4294967296）
  - 其中 1~15的编码时间和空间效率都是最高的，编码值越大，其编码的时间和空间效率就越低，所以建议把经常要传递的值把其字段编码设置为1-15之间的值
  - 1900~2000编码值为Google protobuf 系统内部保留值，建议不要在自己的项目中使用
- 字段默认值
  - 当在传递数据时，对于required数据类型，如果用户没有设置值，则使用默认值传递到对端

service如何定义

- 如果想要将消息类型用在RPC系统中，可以在.proto文件中定义一个RPC服务接口，protocol buffer编译器会根据所选择的不同语言生成服务接口代码
- 例如，想要定义一个RPC服务并具有一个方法，该方法接收SearchRequest并返回一个SearchResponse，此时可以在.proto文件中进行如下定义：

```
service SearchService {
    rpc Search (SearchRequest) returns (SearchResponse) {}
}
```

- 生成的接口代码作为客户端与服务端的约定，服务端必须实现定义的所有接口方法，客户端直接调用同名方法向服务端发起请求，比较麻烦的是，即便业务上不需要参数也必须指定一个请求消息，一般会定义一个空message

Message如何定义

- 一个message类型定义描述了一个请求或响应的消息格式，可以包含多种类型字段
- 例如定义一个搜索请求的消息格式，每个请求包含查询字符串、页码、每页数目
- 字段名用小写，转为go文件后自动变为大写，message就相当于结构体

```
syntax = "proto3";

message SearchRequest {
    string query = 1;                   // 查询字符串
    int32  page_number = 2;           // 页码
    int32  result_per_page = 3;     // 每页条数
}
```

- 首行声明使用的protobuf版本为proto3
- SearchRequest 定义了三个字段，每个字段声明以分号结尾，.proto文件支持双斜线 // 添加单行注释

添加更多Message类型

- 一个.proto文件中可以定义多个消息类型，一般用于同时定义多个相关的消息，例如在同一个.proto文件中同时定义搜索请求和响应消息

```
syntax = "proto3";

// SearchRequest 搜索请求
message SearchRequest {
    string query = 1;                   // 查询字符串
    int32  page_number = 2;           // 页码
    int32  result_per_page = 3;     // 每页条数
}

// SearchResponse 搜索响应
message SearchResponse {
    ...
}
```

如何使用其他Message

- message支持嵌套使用，作为另一message中的字段类型

```
message SearchResponse {
    repeated Result results = 1;
}

message Result {
    string url = 1;
```

```
        string title = 2;
        repeated string snippets = 3;
    }
```

Message嵌套的使用

- 支持嵌套消息，消息可以包含另一个消息作为其字段。也可以在消息内定义一个新的消息
- 内部声明的message类型名称只可在内部直接使用

```
    message SearchResponse {
        message Result {
            string url = 1;
            string title = 2;
            repeated string snippets = 3;
        }
        repeated Result results = 1;
    }
```

- 另外，还可以多层嵌套

```
    message Outer {                  // Level 0
        message MiddleAA {          // Level 1
            message Inner {         // Level 2
                int64 ival = 1;
                bool  booly = 2;
            }
        }
        message MiddleBB {           // Level 1
            message Inner {           // Level 2
                int32 ival = 1;
                bool  booly = 2;
            }
        }
    }
```

proto3的Map类型

- proto3支持map类型声明

```
    map<key_type, value_type> map_field = N;
```

```
    message Project {...}
    map<string, Project> projects = 1;
```

- 键、值类型可以是内置的类型，也可以是自定义message类型
- 字段不支持repeated属性

.proto文件编译

- 通过定义好的.proto文件生成Java, Python, C++, Go, Ruby, JavaNano, Objective-C, or C# 代码，需要安装编译器protoc
- 当使用protocol buffer编译器运行.proto文件时，编译器将生成所选语言的代码，用于使用在.proto文件中定义的消息类型、服务接口约定等。不同语言生成的代码格式不同：
  - C++: 每个.proto文件生成一个.h文件和一个.cc文件，每个消息类型对应一个类
  - Java: 生成一个.java文件，同样每个消息对应一个类，同时还有一个特殊的Builder类用于创建消息接口
  - Python: 姿势不太一样，每个.proto文件中的消息类型生成一个含有静态描述符的模块，该模块与一个元类metaclass在运行时创建需要的Python数据访问类
  - Go: 生成一个.pb.go文件，每个消息类型对应一个结构体
  - Ruby: 生成一个.rb文件的Ruby模块，包含所有消息类型
  - JavaNano: 类似Java，但不包含Builder类
  - Objective-C: 每个.proto文件生成一个pbobjc.h和一个pbobjc.m文件
  - C#: 生成.cs文件包含，每个消息类型对应一个类

import导入定义

- 可以使用import语句导入使用其它描述文件中声明的类型
- protobuf 接口文件可以像C语言的h文件一个，分离为多个，在需要的时候通过 import导入需要对文件。其行为和C语言的#include或者java的import的行为大致相同，例如import “others.proto”;
- protocol buffer编译器会在 -I / –proto_path参数指定的目录中查找导入的文件，如果没有指定该参数，默认在当前目录中查找

包的使用

- 在.proto文件中使用package声明包名，避免命名冲突

```
syntax = "proto3";
package foo.bar;
message Open {...}
```

- 在其他的消息格式定义中可以使用包名+消息名的方式来使用类型，如

```
message Foo {
    ...
    foo.bar.Open open = 1;
    ...
}
```

- 在不同的语言中，包名定义对编译后生成的代码的影响不同
  - C++ 中：对应C++命名空间，例如Open会在命名空间foo::bar中
  - Java 中：package会作为Java包名，除非指定了option jave_package选项
  - Python 中：package被忽略
  - Go 中：默认使用package名作为包名，除非指定了option go_package选项
  - JavaNano 中：同Java
  - C# 中：package会转换为驼峰式命名空间，如Foo.Bar,除非指定了option csharp_namespace选项

小案例

按照惯例，这里也从一个Hello项目开始，本项目定义了一个Hello Service，客户端发送包含字符串名字的请求，服务端返回Hello消息。

流程：

1. 编写`.proto`描述文件
2. 编译生成`.pb.go`文件
3. 服务端实现约定的接口并提供服务
4. 客户端按照约定调用`.pb.go`文件中的方法请求服务

项目结构：

```
|—— hello/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // 服务端
|—— proto/
    |—— hello/
        |—— hello.proto    // proto描述文件
        |—— hello.pb.go    // proto编译后文件
```

Step1：编写描述文件：hello.proto

```
syntax = "proto3"; // 指定proto版本
package hello;     // 指定默认包名

// 指定golang包名
option go_package = "hello";

// 定义Hello服务
service Hello {
    // 定义SayHello方法
    rpc SayHello(HelloRequest) returns (HelloResponse) {}
}

// HelloRequest 请求结构
message HelloRequest {
    string name = 1;
}

// HelloResponse 响应结构
```

```
message HelloResponse {
    string message = 1;
}
```

`hello.proto` 文件中定义了一个Hello Service，该服务包含一个 `SayHello` 方法，同时声明了 `HelloRequest` 和 `HelloResponse` 消息结构用于请求和响应。客户端使用 `HelloRequest` 参数调用 `SayHello` 方法请求服务端，服务端响应 `HelloResponse` 消息。一个最简单的服务就定义好了。

Step2：编译生成 `.pb.go` 文件

```
$ cd proto/hello

# 编译hello.proto
$ protoc -I . --go_out=plugins=grpc:. ./hello.proto
```

在当前目录内生成的 `hello.pb.go` 文件，按照 `.proto` 文件中的说明，包含服务端接口 `HelloServer` 描述，客户端接口及实现 `HelloClient` ，及 `HelloRequest` 、 `HelloResponse` 结构体。

> 注意：不要手动编辑该文件

Step3：实现服务端接口 `server/main.go`

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}
```

```go
// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloResponse, error) {
	resp := new(pb.HelloResponse)
	resp.Message = fmt.Sprintf("Hello %s.", in.Name)

	return resp, nil
}

func main() {
	listen, err := net.Listen("tcp", Address)
	if err != nil {
		grpclog.Fatalf("Failed to listen: %v", err)
	}

	// 实例化grpc Server
	s := grpc.NewServer()

	// 注册HelloService
	pb.RegisterHelloServer(s, HelloService)

	grpclog.Println("Listen on " + Address)
	s.Serve(listen)
}
```

服务端引入编译后的 `proto` 包，定义一个空结构用于实现约定的接口，接口描述可以查看 `hello.pb.go` 文件中的 `HelloServer` 接口描述。实例化grpc Server并注册HelloService，开始提供服务。

运行：

```
$ go run main.go
Listen on 127.0.0.1:50052  //服务端已开启并监听50052端口
```

Step4：实现客户端调用 `client/main.go`

```go
package main

import (
	pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包
	"golang.org/x/net/context"
	"google.golang.org/grpc"
```

```
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

func main() {
    // 连接
    conn, err := grpc.Dial(Address, grpc.WithInsecure())
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)

    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}
```

客户端初始化连接后直接调用 `hello.pb.go` 中实现的 `SayHello` 方法，即可向服务端发起请求，使用姿势就像调用本地方法一样。

运行：

```
$ go run main.go
Hello gRPC.    // 接收到服务端响应
```

如果你收到了"Hello gRPC"的回复，恭喜你已经会使用github.com/jergoo/go-grpc-example/proto/hello了。

> 建议到这里仔细看一看hello.pb.go文件中的内容，对比hello.proto文件，理解protobuf中的定义转换为golang后的结构。

OpenSSL安装

OpenSSL官网

官方下载地址： https://www.openssl.org/source/

Windows安装方法

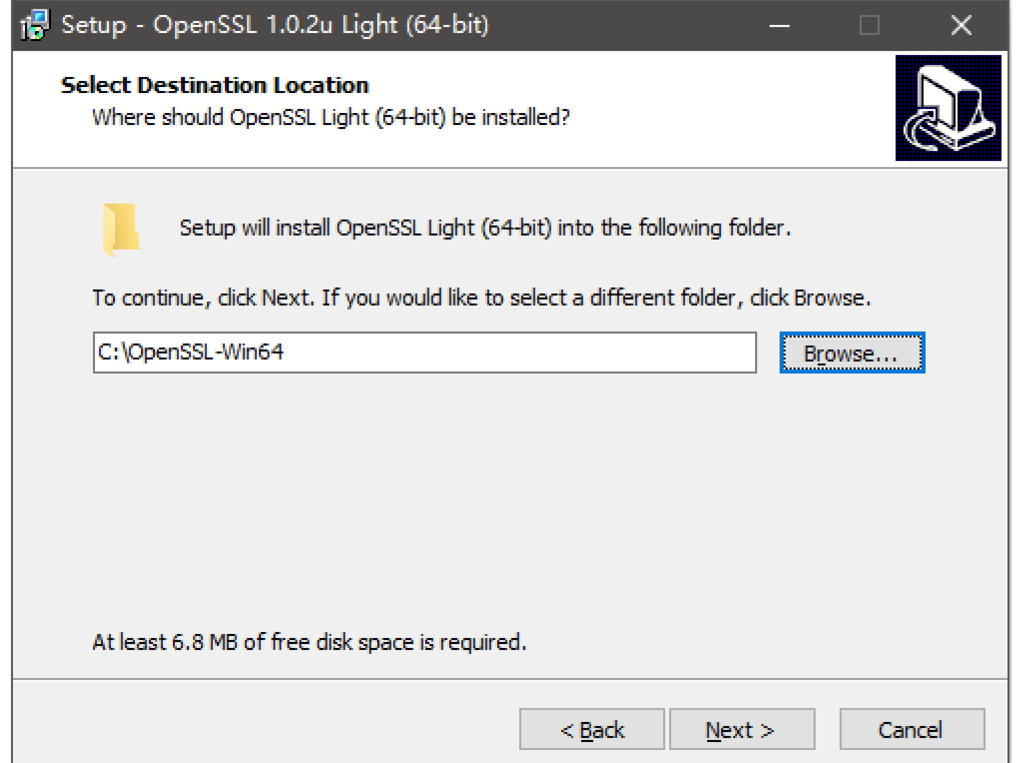
OpenSSL官网没有提供windows版本的安装包，可以选择其他开源平台提供的工具。例如

http://slproweb.com/products/Win32OpenSSL.html

以该工具为例，安装步骤和使用方法如下：

进入下载页面选择下载的版本

选择安装的位置



剩下的都是下一步

设置环境变量，例如工具安装在C:\OpenSSL-Win64，则将C:\OpenSSL-Win64\bin；复制到Path中

打开命令行程序cmd（以管理员身份运行），运行以下命令:

```
利用  openssl  生成公钥私钥
生成公钥:  openssl genrsa -out rsa_private_key.pem 1024
生成私钥: openssl rsa -in rsa_private_key.pem -pubout -out rsa_public_key.pem
```

认证

gRPC默认内置了两种认证方式：

- SSL/TLS认证方式
- 基于Token的认证方式

同时，gRPC提供了接口用于扩展自定义认证方式

TLS认证示例

这里直接扩展hello项目，实现TLS认证机制

首先需要准备证书，在hello目录新建keys目录用于存放证书文件。

证书制作

制作私钥 (.key)

```
# Key considerations for algorithm "RSA" ≥ 2048-bit
$ openssl genrsa -out server.key 2048

# Key considerations for algorithm "ECDSA" ≥ secp384r1
# List ECDSA the supported curves (openssl ecparam -list_curves)
$ openssl ecparam -genkey -name secp384r1 -out server.key
```

自签名公钥(x509) (PEM-encodings `.pem` | `.crt`)

```
$ openssl req -new -x509 -sha256 -key server.key -out server.pem -days 3650
```

自定义信息

```
-----
Country Name (2 letter code) [AU]:CN  //这个是中国的缩写
State or Province Name (full name) [Some-State]:XxXx  //省份
Locality Name (eg, city) []:XxXx  //城市
Organization Name (eg, company) [Internet Widgits Pty Ltd]:XX Co. Ltd  //公司名称
Organizational Unit Name (eg, section) []:Dev   //部门名称
Common Name (e.g. server FQDN or YOUR name) []:server name   //服务名称 例如www.
```

```
topgoer.com
Email Address []:xxx@xxx.com  //邮箱地址
```

目录结构

```
|—— hello-tls/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // 服务端
|—— keys/                 // 证书目录
    |—— server.key
    |—— server.pem
|—— proto/
    |—— hello/
        |—— hello.proto   // proto描述文件
        |—— hello.pb.go   // proto编译后文件
```

示例代码

`proto/helloworld.proto` 及 `proto/hello.pb.go` 文件不需要改动

修改服务端代码：server/main.go

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
```

```
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloResponse, error) {
    resp := new(pb.HelloResponse)
    resp.Message = fmt.Sprintf("Hello %s.", in.Name)

    return resp, nil
}

func main() {
    listen, err := net.Listen("tcp", Address)
    if err != nil {
        grpclog.Fatalf("Failed to listen: %v", err)
    }

    // TLS认证
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem", "../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to generate credentials %v", err)
    }

    // 实例化grpc Server, 并开启TLS认证
    s := grpc.NewServer(grpc.Creds(creds))

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address + " with TLS")

    s.Serve(listen)
}
```

运行：

```
$ go run main.go

Listen on 127.0.0.1:50052 with TLS
```

服务端在实例化grpc Server时，可配置多种选项，TLS认证是其中之一

客户端添加TLS认证：client/main.go

```
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

func main() {
    // TLS连接  记得把server name改成你写的服务器地址
    creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem", "server name")
    if err != nil {
        grpclog.Fatalf("Failed to create TLS credentials %v", err)
    }

    conn, err := grpc.Dial(Address, grpc.WithTransportCredentials(creds))
    if err != nil {
        grpclog.Fatalln(err)
    }
    defer conn.Close()

    // 初始化客户端
    c := pb.NewHelloClient(conn)

    // 调用方法
    req := &pb.HelloRequest{Name: "gRPC"}
    res, err := c.SayHello(context.Background(), req)
    if err != nil {
        grpclog.Fatalln(err)
    }

    grpclog.Println(res.Message)
}
```

运行：

```
$ go run main.go

Hello gRPC
```

客户端添加TLS认证的方式和服务端类似，在创建连接 `Dial` 时，同样可以配置多种选项，后面的示例中会看到更多的选项。

Token认证示例

再进一步，继续扩展hello-tls项目，实现TLS + Token认证机制

目录结构

```
|—— hello_token/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // 服务端
|—— keys/                 // 证书目录
    |—— server.key
    |—— server.pem
|—— proto/
    |—— hello/
        |—— hello.proto   // proto描述文件
        |—— hello.pb.go   // proto编译后文件
```

示例代码

先修改客户端实现：client/main.go

```
package main

import (
    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)
```

```go
const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"

    // OpenTLS 是否开启TLS认证
    OpenTLS = true
)

// customCredential 自定义认证
type customCredential struct{}

// GetRequestMetadata 实现自定义认证接口
func (c customCredential) GetRequestMetadata(ctx context.Context, uri ...string)
(map[string]string, error) {
    return map[string]string{
        "appid":  "101010",
        "appkey": "i am key",
    }, nil
}

// RequireTransportSecurity 自定义认证是否开启TLS
func (c customCredential) RequireTransportSecurity() bool {
    return OpenTLS
}

func main() {
    var err error
    var opts []grpc.DialOption

    if OpenTLS {
        // TLS连接
        creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem",
"server name")
        if err != nil {
            grpclog.Fatalf("Failed to create TLS credentials %v", err)
        }
        opts = append(opts, grpc.WithTransportCredentials(creds))
    } else {
        opts = append(opts, grpc.WithInsecure())
    }

    // 使用自定义认证
    opts = append(opts, grpc.WithPerRPCCredentials(new(customCredential)))

    conn, err := grpc.Dial(Address, opts...)
```

```
	if err != nil {
		grpclog.Fatalln(err)
	}

	defer conn.Close()

	// 初始化客户端
	c := pb.NewHelloClient(conn)

	// 调用方法
	req := &pb.HelloRequest{Name: "gRPC"}
	res, err := c.SayHello(context.Background(), req)
	if err != nil {
		grpclog.Fatalln(err)
	}

	grpclog.Println(res.Message)
}
```

这里我们定义了一个 `customCredential` 结构，并实现了两个方法 `GetRequestMetadata` 和 `RequireTransportSecurity` 。这是gRPC提供的自定义认证方式，每次RPC调用都会传输认证信息。 `customCredential` 其实是实现了 `grpc/credential` 包内的 `PerRPCCredentials` 接口。每次调用，token信息会通过请求的metadata传输到服务端。下面具体看一下服务端如何获取metadata中的信息。

修改server/main.go中的SayHello方法：

```
package main

import (
	"fmt"
	"net"

	pb "github.com/jergoo/go-grpc-example/proto/hello"

	"golang.org/x/net/context"
	"google.golang.org/grpc"
	"google.golang.org/grpc/codes"
	"google.golang.org/grpc/credentials" // 引入grpc认证包
	"google.golang.org/grpc/grpclog"
	"google.golang.org/grpc/metadata" // 引入grpc meta包
)

const (
	// Address gRPC服务地址
```

```go
	Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService ...
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
	// 解析metada中的信息并验证
	md, ok := metadata.FromContext(ctx)
	if !ok {
		return nil, grpc.Errorf(codes.Unauthenticated, "无Token认证信息")
	}

	var (
		appid  string
		appkey string
	)

	if val, ok := md["appid"]; ok {
		appid = val[0]
	}

	if val, ok := md["appkey"]; ok {
		appkey = val[0]
	}

	if appid != "101010" || appkey != "i am key" {
		return nil, grpc.Errorf(codes.Unauthenticated, "Token认证信息无效: appid
=%s, appkey=%s", appid, appkey)
	}

	resp := new(pb.HelloResponse)
	resp.Message = fmt.Sprintf("Hello %s.\nToken info: appid=%s,appkey=%s", in.N
ame, appid, appkey)

	return resp, nil
}

func main() {
	listen, err := net.Listen("tcp", Address)
	if err != nil {
```

```
        grpclog.Fatalf("failed to listen: %v", err)
    }

    // TLS认证
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to generate credentials %v", err)
    }

    // 实例化grpc Server, 并开启TLS认证
    s := grpc.NewServer(grpc.Creds(creds))

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address + " with TLS + Token")

    s.Serve(listen)
}
```

服务端可以从 `context` 中获取每次请求的metadata，从中读取客户端发送的token信息并验证有效性。

运行：

```
$ go run main.go

Listen on 50052 with TLS + Token
```

运行客户端程序 client/main.go：

```
$ go run main.go

// 认证成功结果
Hello gRPC
Token info: appid=101010,appkey=i am key

// 修改key验证认证失败结果：
rpc error: code = 16 desc = Token认证信息无效: appID=101010, appKey=i am not key
```

`google.golang.org/grpc/credentials/oauth` 包已实现了用于Google API的oauth和jwt验证的方法，使用方法可以参考官方文档。在实际应用中，我们可以根据自己的业务需求实现合适的验证方式。

拦截器

grpc服务端和客户端都提供了interceptor功能，功能类似middleware，很适合在这里处理验证、日志等流程。

在自定义Token认证的示例中，认证信息是由每个服务中的方法处理并认证的，如果有大量的接口方法，这种姿势就太不优雅了，每个接口实现都要先处理认证信息。这个时候interceptor就可以用来解决了这个问题，在请求被转到具体接口之前处理认证信息，一处认证，到处无忧。在客户端，我们增加一个请求日志，记录请求相关的参数和耗时等等。修改hello_token项目实现：

目录结构

```
|—— hello_interceptor/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // 服务端
|—— keys/             // 证书目录
    |—— server.key
    |—— server.pem
|—— proto/
    |—— hello/
        |—— hello.proto   // proto描述文件
        |—— hello.pb.go   // proto编译后文件
```

示例代码

Step 1. 服务端interceptor：

> hello_interceptor/server/main.go

```
package main

import (
    "fmt"
    "net"

    pb "github.com/jergoo/go-grpc-example/proto/hello"

    "golang.org/x/net/context"
```

```
	"google.golang.org/grpc"
	"google.golang.org/grpc/codes"       // grpc 响应状态码
	"google.golang.org/grpc/credentials" // grpc认证包
	"google.golang.org/grpc/grpclog"
	"google.golang.org/grpc/metadata" // grpc metadata包
)

const (
	// Address gRPC服务地址
	Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.He
lloResponse, error) {
	resp := new(pb.HelloResponse)
	resp.Message = fmt.Sprintf("Hello %s.", in.Name)

	return resp, nil
}

func main() {
	listen, err := net.Listen("tcp", Address)
	if err != nil {
		grpclog.Fatalf("Failed to listen: %v", err)
	}

	var opts []grpc.ServerOption

	// TLS认证
	creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
	if err != nil {
		grpclog.Fatalf("Failed to generate credentials %v", err)
	}

	opts = append(opts, grpc.Creds(creds))

	// 注册interceptor
	opts = append(opts, grpc.UnaryInterceptor(interceptor))
```

```
    // 实例化grpc Server
    s := grpc.NewServer(opts...)

    // 注册HelloService
    pb.RegisterHelloServer(s, HelloService)

    grpclog.Println("Listen on " + Address + " with TLS + Token + Interceptor")

    s.Serve(listen)
}

// auth 验证Token
func auth(ctx context.Context) error {
    md, ok := metadata.FromContext(ctx)
    if !ok {
        return grpc.Errorf(codes.Unauthenticated, "无Token认证信息")
    }

    var (
        appid  string
        appkey string
    )

    if val, ok := md["appid"]; ok {
        appid = val[0]
    }

    if val, ok := md["appkey"]; ok {
        appkey = val[0]
    }

    if appid != "101010" || appkey != "i am key" {
        return grpc.Errorf(codes.Unauthenticated, "Token认证信息无效: appid=%s, appkey=%s", appid, appkey)
    }

    return nil
}

// interceptor 拦截器
func interceptor(ctx context.Context, req interface{}, info *grpc.UnaryServerInfo, handler grpc.UnaryHandler) (interface{}, error) {
    err := auth(ctx)
    if err != nil {
        return nil, err
```

```
    }
    // 继续处理请求
    return handler(ctx, req)
}
```

Step 2. 实现客户端interceptor：

> hello_interceptor/client/main.go

```
package main

import (
    "time"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入proto包

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials" // 引入grpc认证包
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"

    // OpenTLS 是否开启TLS认证
    OpenTLS = true
)

// customCredential 自定义认证
type customCredential struct{}

// GetRequestMetadata 实现自定义认证接口
func (c customCredential) GetRequestMetadata(ctx context.Context, uri ...string)
(map[string]string, error) {
    return map[string]string{
        "appid":  "101010",
        "appkey": "i am key",
    }, nil
}

// RequireTransportSecurity 自定义认证是否开启TLS
func (c customCredential) RequireTransportSecurity() bool {
    return OpenTLS
```

```
}

func main() {
	var err error
	var opts []grpc.DialOption

	if OpenTLS {
		// TLS连接
		creds, err := credentials.NewClientTLSFromFile("../../keys/server.pem", "server name")
		if err != nil {
			grpclog.Fatalf("Failed to create TLS credentials %v", err)
		}
		opts = append(opts, grpc.WithTransportCredentials(creds))
	} else {
		opts = append(opts, grpc.WithInsecure())
	}

	// 指定自定义认证
	opts = append(opts, grpc.WithPerRPCCredentials(new(customCredential)))
	// 指定客户端interceptor
	opts = append(opts, grpc.WithUnaryInterceptor(interceptor))

	conn, err := grpc.Dial(Address, opts...)
	if err != nil {
		grpclog.Fatalln(err)
	}
	defer conn.Close()

	// 初始化客户端
	c := pb.NewHelloClient(conn)

	// 调用方法
	req := &pb.HelloRequest{Name: "gRPC"}
	res, err := c.SayHello(context.Background(), req)
	if err != nil {
		grpclog.Fatalln(err)
	}

	grpclog.Println(res.Message)
}

// interceptor 客户端拦截器
func interceptor(ctx context.Context, method string, req, reply interface{}, cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
	start := time.Now()
```

```
    err := invoker(ctx, method, req, reply, cc, opts...)
    grpclog.Printf("method=%s req=%v rep=%v duration=%s error=%v\n", method, req, reply, time.Since(start), err)
    return err
}
```

运行结果

```
$ cd hello_inteceptor/server && go run main.go
Listen on 127.0.0.1:50052 with TLS + Token + Interceptor
```

```
$ cd hello_inteceptor/client && go run main.go
method=/hello.Hello/SayHello req=name:"gRPC"  rep=message:"Hello gRPC."  duration=33.879699ms error=<nil>

Hello gRPC.
```

项目推荐： go-grpc-middleware

这个项目对interceptor进行了封装，支持多个拦截器的链式组装，对于需要多种处理的地方使用起来会更方便些。

内置Trace

grpc内置了客户端和服务端的请求追踪，基于 `golang.org/x/net/trace` 包实现，默认是开启状态，可以查看事件和请求日志，对于基本的请求状态查看调试也是很有帮助的，客户端与服务端基本一致，这里以服务端开启trace server为例，修改hello项目服务端代码：

目录结构

```
|—— hello_trace/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // 服务端
|—— proto/
    |—— hello/
        |—— hello.proto   // proto描述文件
        |—— hello.pb.go   // proto编译后文件
```

示例代码

```
package main

import (
    "fmt"
    "net"
    "net/http"

    pb "github.com/jergoo/go-grpc-example/proto/hello" // 引入编译生成的包

    "golang.org/x/net/context"
    "golang.org/x/net/trace"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
)

const (
    // Address gRPC服务地址
    Address = "127.0.0.1:50052"
)

// 定义helloService并实现约定的接口
```

```go
type helloService struct{}

// HelloService Hello服务
var HelloService = helloService{}

// SayHello 实现Hello服务接口
func (h helloService) SayHello(ctx context.Context, in *pb.HelloRequest) (*pb.HelloResponse, error) {
	resp := new(pb.HelloResponse)
	resp.Message = fmt.Sprintf("Hello %s.", in.Name)

	return resp, nil
}

func main() {
	listen, err := net.Listen("tcp", Address)
	if err != nil {
		grpclog.Fatalf("failed to listen: %v", err)
	}

	// 实例化grpc Server
	s := grpc.NewServer()

	// 注册HelloService
	pb.RegisterHelloServer(s, HelloService)

	// 开启trace
	go startTrace()

	grpclog.Println("Listen on " + Address)
	s.Serve(listen)
}

func startTrace() {
	trace.AuthRequest = func(req *http.Request) (any, sensitive bool) {
		return true, true
	}
	go http.ListenAndServe(":50051", nil)
	grpclog.Println("Trace listen on 50051")
}
```

这里我们开启一个http服务监听50051端口，用来查看grpc请求的trace信息

运行：

```
$ go run main.go

Listen on 127.0.0.1:50052
Trace listen on 50051

# 进入client目录执行一次客户端请求
```

服务端事件查看

访问：localhost:50051/debug/events，结果如图：

可以看到服务端注册的服务和服务正常启动的事件信息。

请求日志信息查看

访问：localhost:50051/debug/requests，结果如图：

/debug/requests

grpc.Recv.Hello [0 active] [≥0s] [≥0.05s] [≥0.1s] [≥0.2s] [≥0.5s] [≥1s] [≥10s] [≥100s] [errors] [minute] [hour] [total]

Family: grpc.Recv.Hello

[Normal/Summary] [Normal/Expanded] [Traced/Summary] [Traced/Expanded]

Completed Requests

```
When                         Elapsed (s)
2017/08/07 09:59:30.624530     0.003282 /hello.Hello/SayHello
           09:59:30.624936     .    406 ... RPC: from 127.0.0.1:51794 deadline:none
           09:59:30.627410     .   2474 ... recv: name:"gRPC"
           09:59:30.627419     .      9 ... OK
           09:59:30.627476     .     57 ... sent: message:"Hello gRPC."
```

www.topgoer.com

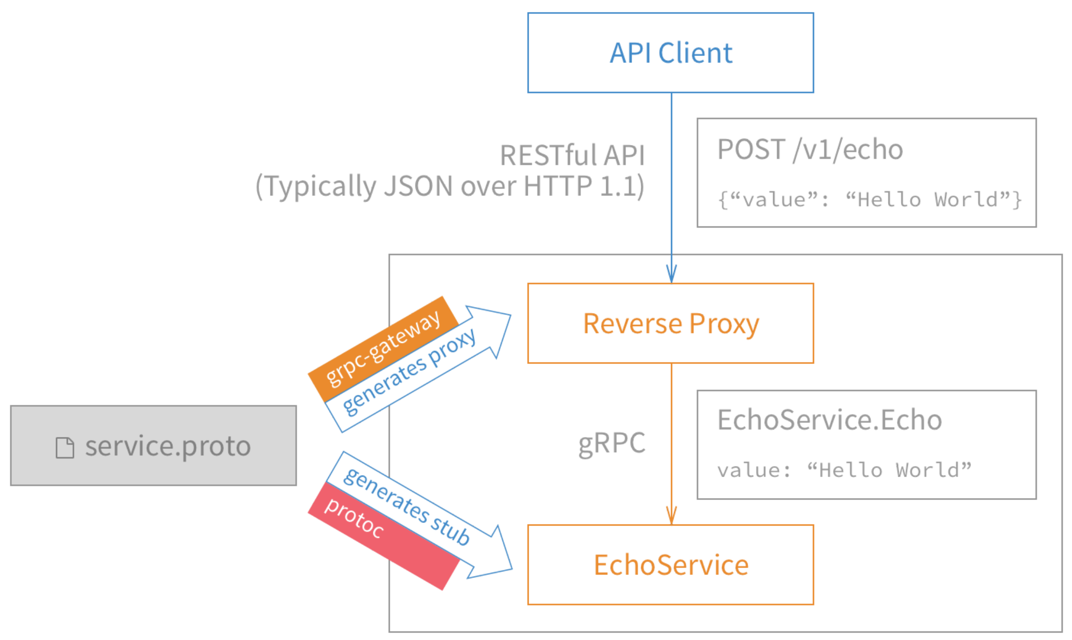
这里可以显示最近的请求状态，包括请求的服务、参数、耗时、响应，对于简单的状态查看还是很方便的，默认值显示最近10条记录。

HTTP网关

源自coreos的一篇博客 Take a REST with HTTP/2, Protobufs, and Swagger。

etcd3 API全面升级为gRPC后，同时要提供REST API服务，维护两个版本的服务显然不太合理，所以grpc-gateway诞生了。通过protobuf的自定义option实现了一个网关，服务端同时开启gRPC和HTTP服务，HTTP服务接收客户端请求后转换为grpc请求数据，获取响应后转为json数据返回给客户端。

结构如图：



安装grpc-gateway

```
$ go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
```

目录结构

```
|—— hello_http/
    |—— client/
        |—— main.go   // 客户端
    |—— server/
        |—— main.go   // GRPC服务端
```

```
    |—— server_http/
        |—— main.go     // HTTP服务端
|—— proto/
    |—— google          // googleApi http-proto定义
        |—— api
            |—— annotations.proto
            |—— annotations.pb.go
            |—— http.proto
            |—— http.pb.go
    |—— hello_http/
        |—— hello_http.proto    // proto描述文件
        |—— hello_http.pb.go    // proto编译后文件
        |—— hello_http_pb.gw.go // gateway编译后文件
```

这里用到了google官方Api中的两个proto描述文件，直接拷贝不要做修改，里面定义了protocol buffer扩展的HTTP option，为grpc的http转换提供支持。

示例代码

Step 1. 编写proto描述文件：proto/hello_http.proto

```
syntax = "proto3";

package hello_http;
option go_package = "hello_http";

import "google/api/annotations.proto";

// 定义Hello服务
service HelloHTTP {
    // 定义SayHello方法
    rpc SayHello(HelloHTTPRequest) returns (HelloHTTPResponse) {
        // http option
        option (google.api.http) = {
            post: "/example/echo"
            body: "*"
        };
    }
}

// HelloRequest 请求结构
message HelloHTTPRequest {
    string name = 1;
}
```

```
// HelloResponse 响应结构
message HelloHTTPResponse {
    string message = 1;
}
```

这里在原来的 `SayHello` 方法定义中增加了http option, POST方式，路由为"/example/echo"。

Step 2. 编译proto

```
$ cd proto

# 编译google.api
$ protoc -I . --go_out=plugins=grpc,Mgoogle/protobuf/descriptor.proto=github.com/golang/protobuf/protoc-gen-go/descriptor:. google/api/*.proto

# 编译hello_http.proto
$ protoc -I . --go_out=plugins=grpc,Mgoogle/api/annotations.proto=github.com/jergoo/go-grpc-example/proto/google/api:. hello_http/*.proto

# 编译hello_http.proto gateway
$ protoc --grpc-gateway_out=logtostderr=true:. hello_http/hello_http.proto
```

注意这里需要编译google/api中的两个proto文件，同时在编译hello_http.proto时使用 `M` 参数指定引入包名，最后使用grpc-gateway编译生成 `hello_http_pb.gw.go` 文件，这个文件就是用来做协议转换的，查看文件可以看到里面生成的http handler，处理proto文件中定义的路由"example/echo"接收POST参数，调用HelloHTTP服务的客户端请求grpc服务并响应结果。

Step 3: 实现服务端和客户端

server/main.go和client/main.go的实现与hello项目一致，这里不再说明。

> server_http/main.go

```
package main

import (
    "net/http"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/grpclog"
```

```
    gw "github.com/jergoo/go-grpc-example/proto/hello_http"
)

func main() {
    ctx := context.Background()
    ctx, cancel := context.WithCancel(ctx)
    defer cancel()

    // grpc服务地址
    endpoint := "127.0.0.1:50052"
    mux := runtime.NewServeMux()
    opts := []grpc.DialOption{grpc.WithInsecure()}

    // HTTP转grpc
    err := gw.RegisterHelloHTTPHandlerFromEndpoint(ctx, mux, endpoint, opts)
    if err != nil {
        grpclog.Fatalf("Register handler err:%v\n", err)
    }

    grpclog.Println("HTTP Listen on 8080")
    http.ListenAndServe(":8080", mux)
}
```

就是这么简单。开启了一个http server，收到请求后根据路由转发请求到对应的RPC接口获得结果。grpc-gateway做的事情就是帮我们自动生成了转换过程的实现。

运行结果

依次开启gRPC服务和HTTP服务端：

```
$ cd hello_http/server && go run main.go
Listen on 127.0.0.1:50052
```

```
$ cd hello_http/server_http && go run main.go
HTTP Listen on 8080
```

调用grpc客户端：

```
$ cd hello_http/client && go run main.go
Hello gRPC.

# HTTP 请求
```

```
$ curl -X POST -k http://localhost:8080/example/echo -d '{"name": "gRPC-HTTP is working!"}'
{"message":"Hello gRPC-HTTP is working!."}
```

升级版服务端

上面的使用方式已经实现了我们最初的需求，grpc-gateway项目中提供的示例也是这种使用方式，这样后台需要开启两个服务两个端口。其实我们也可以只开启一个服务，同时提供http和gRPC调用方式。

新建一个项目 `hello_http_2` ，基于 `hello_tls` 项目改造。客户端只要修改调用的proto包地址就可以了，这里我们看服务端的实现：

> hello_http_2/server/main.go

```go
package main

import (
    "crypto/tls"
    "io/ioutil"
    "net"
    "net/http"
    "strings"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    pb "github.com/jergoo/go-grpc-example/proto/hello_http"
    "golang.org/x/net/context"
    "golang.org/x/net/http2"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/grpclog"
)

// 定义helloHTTPService并实现约定的接口
type helloHTTPService struct{}

// HelloHTTPService Hello HTTP服务
var HelloHTTPService = helloHTTPService{}

// SayHello 实现Hello服务接口
func (h helloHTTPService) SayHello(ctx context.Context, in *pb.HelloHTTPRequest) (*pb.HelloHTTPResponse, error) {
    resp := new(pb.HelloHTTPResponse)
    resp.Message = "Hello " + in.Name + "."
```

```
    return resp, nil
}

func main() {
    endpoint := "127.0.0.1:50052"
    conn, err := net.Listen("tcp", endpoint)
    if err != nil {
        grpclog.Fatalf("TCP Listen err:%v\n", err)
    }

    // grpc tls server
    creds, err := credentials.NewServerTLSFromFile("../../keys/server.pem",
"../../keys/server.key")
    if err != nil {
        grpclog.Fatalf("Failed to create server TLS credentials %v", err)
    }
    grpcServer := grpc.NewServer(grpc.Creds(creds))
    pb.RegisterHelloHTTPServer(grpcServer, HelloHTTPService)

    // gw server
    ctx := context.Background()
    dcreds, err := credentials.NewClientTLSFromFile("../../keys/server.pem", "se
rver name")
    if err != nil {
        grpclog.Fatalf("Failed to create client TLS credentials %v", err)
    }
    dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}
    gwmux := runtime.NewServeMux()
    if err = pb.RegisterHelloHTTPHandlerFromEndpoint(ctx, gwmux, endpoint, dopt
s); err != nil {
        grpclog.Fatalf("Failed to register gw server: %v\n", err)
    }

    // http服务
    mux := http.NewServeMux()
    mux.Handle("/", gwmux)

    srv := &http.Server{
        Addr:      endpoint,
        Handler:   grpcHandlerFunc(grpcServer, mux),
        TLSConfig: getTLSConfig(),
    }

    grpclog.Infof("gRPC and https listen on: %s\n", endpoint)
```

```
    if err = srv.Serve(tls.NewListener(conn, srv.TLSConfig)); err != nil {
        grpclog.Fatal("ListenAndServe: ", err)
    }

    return
}

func getTLSConfig() *tls.Config {
    cert, _ := ioutil.ReadFile("../../keys/server.pem")
    key, _ := ioutil.ReadFile("../../keys/server.key")
    var demoKeyPair *tls.Certificate
    pair, err := tls.X509KeyPair(cert, key)
    if err != nil {
        grpclog.Fatalf("TLS KeyPair err: %v\n", err)
    }
    demoKeyPair = &pair
    return &tls.Config{
        Certificates: []tls.Certificate{*demoKeyPair},
        NextProtos:   []string{http2.NextProtoTLS}, // HTTP2 TLS支持
    }
}

// grpcHandlerFunc returns an http.Handler that delegates to grpcServer on incom
ing gRPC
// connections or otherHandler otherwise. Copied from cockroachdb.
func grpcHandlerFunc(grpcServer *grpc.Server, otherHandler http.Handler) http.Ha
ndler {
    if otherHandler == nil {
        return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            grpcServer.ServeHTTP(w, r)
        })
    }
    return http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
        if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"),
"application/grpc") {
            grpcServer.ServeHTTP(w, r)
        } else {
            otherHandler.ServeHTTP(w, r)
        }
    })
}
```

gRPC服务端接口的实现没有区别，重点在于HTTP服务的实现。gRPC是基于http2实现的， `net/http` 包也实现了http2，所以我们可以开启一个HTTP服务同时服务两个版本的协议，在注册http handler的时候，在方法 `grpcHandlerFunc` 中检测请求头信息，决定是

直接调用gRPC服务，还是使用gateway的HTTP服务。 `net/http` 中对http2的支持要求开启https，所以这里要求使用https服务。

步骤

- 注册开启TLS的grpc服务
- 注册开启TLS的gateway服务，地址指向grpc服务
- 开启HTTP server

运行结果

```
$ cd hello_http_2/server && go run main.go
gRPC and https listen on: 127.0.0.1:50052
```

```
$ cd hello_http_2/client && go run main.go
Hello gRPC.

# HTTP 请求
$ curl -X POST -k https://localhost:50052/example/echo -d '{"name": "gRPC-HTTP is working!"}'
{"message":"Hello gRPC-HTTP is working!."}
```

Go Micro入门

go-micro简介

- Go Micro是一个插件化的基础框架，基于此可以构建微服务，Micro的设计哲学是可插拔的插件化架构
- 在架构之外，它默认实现了consul作为服务发现（2019年源码修改了默认使用mdns），通过http进行通信，通过protobuf和json进行编解码

go-micro的主要功能

- 服务发现：自动服务注册和名称解析。服务发现是微服务开发的核心。当服务A需要与服务B通话时，它需要该服务的位置。默认发现机制是多播DNS（mdns），一种零配置系统。您可以选择使用SWIM协议为p2p网络设置八卦，或者为弹性云原生设置设置consul
- 负载均衡：基于服务发现构建的客户端负载均衡。一旦我们获得了服务的任意数量实例的地址，我们现在需要一种方法来决定要路由到哪个节点。我们使用随机散列负载均衡来提供跨服务的均匀分布，并在出现问题时重试不同的节点
- 消息编码：基于内容类型的动态消息编码。客户端和服务器将使用编解码器和内容类型为您无缝编码和解码Go类型。可以编码任何种类的消息并从不同的客户端发送。客户端和服务器默认处理此问题。这包括默认的protobuf和json
- 请求/响应：基于RPC的请求/响应，支持双向流。我们提供了同步通信的抽象。对服务的请求将自动解决，负载平衡，拨号和流式传输。启用tls时，默认传输为http / 1.1或http2
- Async Messaging：PubSub是异步通信和事件驱动架构的一流公民。事件通知是微服务开发的核心模式。启用tls时，默认消息传递是点对点http / 1.1或http2
- 可插拔接口：Go Micro为每个分布式系统抽象使用Go接口，因此，这些接口是可插拔的，并允许Go Micro与运行时无关，可以插入任何基础技术
    - 插件地址：https://github.com/micro/go-plugins

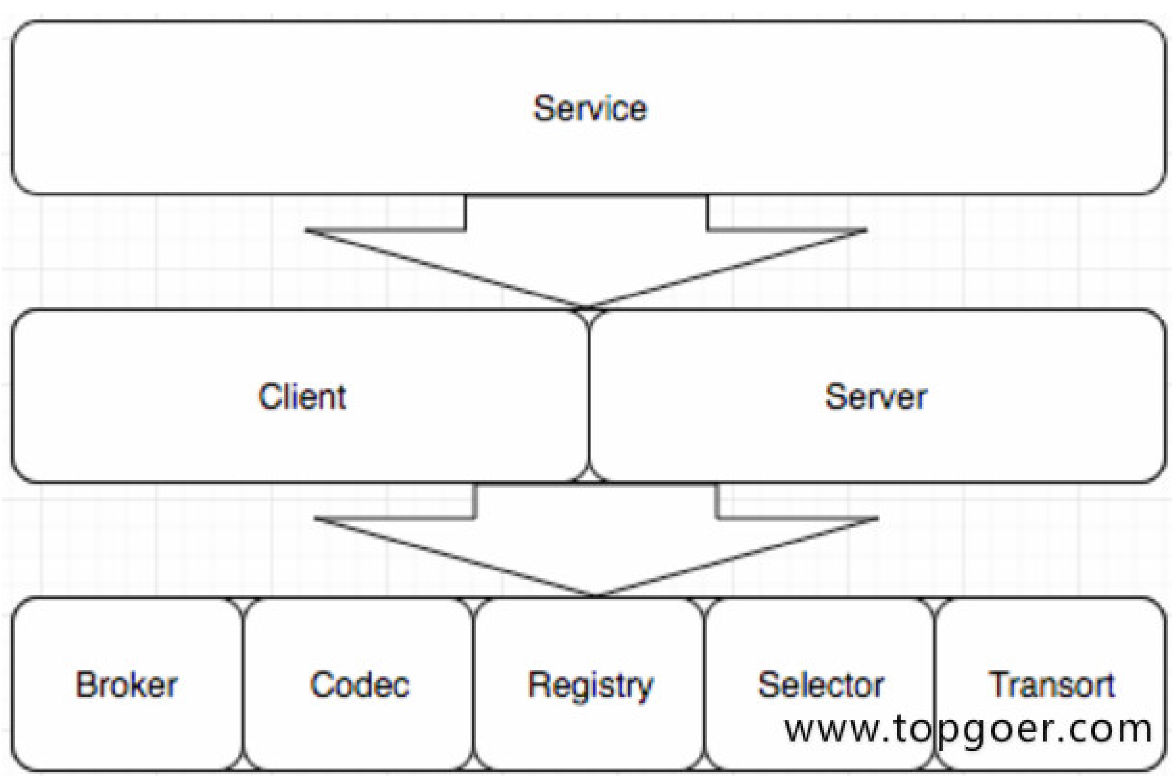
go-micro通信流程

- Server监听客户端的调用，和Brocker推送过来的信息进行处理。并且Server端需要向Register注册自己的存在或消亡，这样Client才能知道自己的状态

- Register服务的注册的发现，Client端从Register中得到Server的信息，然后每次调用都根据算法选择一个的Server进行通信，当然通信是要经过编码/解码，选择传输协议等一系列过程的
- 如果有需要通知所有的Server端可以使用Brocker进行信息的推送，Brocker 信息队列进行信息的接收和发布

go-micro核心接口

- go-micro之所以可以高度订制和他的框架结构是分不开的，go-micro由8个关键的interface组成，每一个interface都可以根据自己的需求重新实现，这8个主要的inteface也构成了go-micro的框架结构



Go Micro接口详解

Transort通信接口

通信相关接口

```
type Socket interface {
    Recv(*Message) error
    Send(*Message) error
    Close() error
}

type Client interface {
    Socket
}

type Listener interface {
    Addr() string
    Close() error
    Accept(func(Socket)) error
}

type Transport interface {
    Dial(addr string, opts ...DialOption) (Client, error)
    Listen(addr string, opts ...ListenOption) (Listener, error)
    String() string
}
```

Codec编码接口

编解码，底层也是protobuf

```
type Codec interface {
    ReadHeader(*Message, MessageType) error
    ReadBody(interface{}) error
    Write(*Message, interface{}) error
    Close() error
    String() string
}
```

Registry注册接口

服务注册发现的实现：etcd、consul、mdns、kube-DNS、zk

```
type Registry interface {
    Register(*Service, ...RegisterOption) error
    Deregister(*Service) error
    GetService(string) ([]*Service, error)
    ListServices() ([]*Service, error)
    Watch(...WatchOption) (Watcher, error)
    String() string
    Options() Options
}
```

Selector负载均衡

根据不同算法请求主机列表

```
type Selector interface {
    Init(opts ...Option) error
    Options() Options
    // Select returns a function which should return the next node
    Select(service string, opts ...SelectOption) (Next, error)
    // Mark sets the success/error against a node
    Mark(service string, node *registry.Node, err error)
    // Reset returns state back to zero for a service
    Reset(service string)
    // Close renders the selector unusable
    Close() error
    // Name of the selector
    String() string
}
```

Broker发布订阅接口

pull push watch

```
type Broker interface {
    Options() Options
    Address() string
    Connect() error
    Disconnect() error
    Init(...Option) error
    Publish(string, *Message, ...PublishOption) error
    Subscribe(string, Handler, ...SubscribeOption) (Subscriber, error)
```

```
    String() string
}
```

Client客户端接口

```
type Client interface {
    Init(...Option) error
    Options() Options
    NewMessage(topic string, msg interface{}, opts ...MessageOption) Message
    NewRequest(service, method string, req interface{}, reqOpts ...RequestOption) 
Request
    Call(ctx context.Context, req Request, rsp interface{}, opts ...CallOption) e
rror
    Stream(ctx context.Context, req Request, opts ...CallOption) (Stream, error)
    Publish(ctx context.Context, msg Message, opts ...PublishOption) error
    String() string
}
```

Server服务端接口

```
type Server interface {
    Options() Options
    Init(...Option) error
    Handle(Handler) error
    NewHandler(interface{}, ...HandlerOption) Handler
    NewSubscriber(string, interface{}, ...SubscriberOption) Subscriber
    Subscribe(Subscriber) error
    Register() error
    Deregister() error
    Start() error
    Stop() error
    String() string
}
```

Serveice接口

```
type Service interface {
    Init(...Option)
    Options() Options
    Client() client.Client
    Server() server.Server
    Run() error
```

```
    String() string
}
```

Go Micro文档1.x

请跳转 http://wen.topgoer.com/docs/go-Micro1x/gomocro1qianjing

Go Micro文档2.x

请跳转 http://wen.topgoer.com/docs/mindoc/jieshao

Go高级

请跳转 http://www.topgoer.cn/docs/gosuanfa/gosuanfa-1c906k4cpjfnp

插件库

请跳转 http://www.topgoer.cn/docs/gochajian/gochajian-1c8r6fd1vfk9o

项目

github库地址

TCP扫描器

定时任务

基于角色的访问控制框架

uuid

支付宝支付

微信支付

微信公众号开发

爬虫小案例

千万数据过滤

go-admin

go-vue-admin

go-admin

log

聊天室小案例

性能压测工具wrk

gcc安装

gim即时通讯

github库地址

分享好的github库地址

插件

- QQ、微信（WeChat）、支付宝（AliPay）的Go版本SDK https://github.com/iGoogle-ink/gopay
- 发送邮件库 https://github.com/go-gomail/gomail
- 读写Microsoft Excel https://github.com/360EntSecGroup-Skylar/excelize 详细资料
- 生成uuid https://github.com/satori/go.uuid
- 开源，分布式，简单高效的搜索引擎 https://github.com/go-ego/riot
- 基于 Go 的高性能 MySQL Proxy https://github.com/flike/kingshard
- yaml对Go语言的支持 https://github.com/go-yaml/yaml/tree/v2.2.7
- Codis是一个分布式Redis解决方案数据库代理 https://github.com/CodisLabs/codis
- 用Go语言编写的markdown解析器 https://github.com/yuin/goldmark

项目

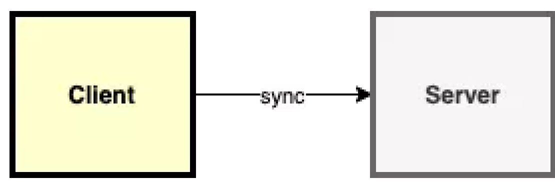
- 基于beego框架的接口在线文档管理系统 https://github.com/lifei6671/mindoc
- 开源文库系统 https://github.com/truthhun/DocHub
- Go常用规范定义案例 https://github.com/zc2638/go-standard
- 开源监控度量的看板系统 https://github.com/zc2638/go-standard
- go应用开发的调试工具 https://github.com/derekparker/delve
- 高并发、重量级爬虫软件 https://github.com/henrylee2cn/pholcus
- Web分析 https://github.com/matomo-org/matomo

TCP扫描器

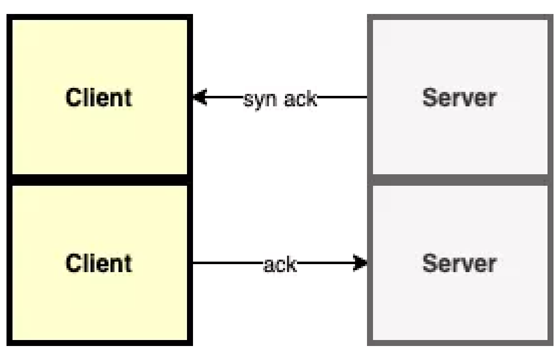
Go在网络应用编程方面堪称完美。它自带的标准库也很优秀，在开发过程中可以给予我们很多帮助。

在本文中，我们将会用Go写一个简单的TCP扫描器。整个程序的代码在50行以内。在我们开始动手之前，先介绍一些理论知识。

不得不说，TCP是比我们介绍的要复杂的多，但是我们只介绍一点基础知识。TCP的握手有三个过程。首先，客户端发送一个 syn 的包，表示建立回话的开始。如果客户端收到超时，说明端口可能在防火墙后面，



第二，如果服务端应答 syn-ack 包，意味着这个端口是打开的，否则会返回 rst 包。最后，客户端需要另外发送一个 ack 包。从这时起，连接就已经建立。



我们TCP扫描器第一步先实现单个端口的测试。使用标准库中的 net.Dial 函数，该函数接收两个参数：协议和测试地址（带端口号）。

```
package main

import (
    "fmt"
    "net"
)

func main() {
    _, err := net.Dial("tcp", "google.com:80")
    if err == nil {
        fmt.Println("Connection successful")
    } else {
```

```
        fmt.Println(err)
    }
}
```

为了不一个一个地测试每个端口，我们将添加一个简单的循环来简化整个测试过程。

```
package main

import (
    "fmt"
    "net"
)

func main() {
    for port := 80; port < 100; port++ {
        conn, err := net.Dial("tcp", fmt.Sprintf("google.com:%d", port))
        if err == nil {
            conn.Close()
            fmt.Println("Connection successful")
        } else {
            fmt.Println(err)
        }
    }
}
这种处理方式有个很大的问题，极度的慢。我们可以通过两个操作来处理一下：并行的执行
及为每个连接添加超时控制。

我们来看下如何实现并行。第一步先把扫描功能拆分为一个独立函数。这样会使我们的代码
看起来清晰。

func isOpen(host string, port int) bool {
  time.Sleep(time.Millisecond * 1)
  conn, err := net.Dial("tcp", fmt.Sprintf("%s:%d", host, port))
  if err == nil {
     _ = conn.Close()
     return true
  }

  return false
}
```

我们会引入一个新的方法 WaitGroup ，详细用法信息可以参考标准库文档。在主函数中，我们可以拆分为协程去执行，然后等待执行结束。

```
func main() {
  ports := []int{}

  wg := &sync.WaitGroup{}
  for port := 1; port < 100; port++ {
      wg.Add(1)
      go func() {
          opened := isOpen("google.com", port)
          if opened {
              ports = append(ports, port)
          }
          wg.Done()
      }()
  }

  wg.Wait()
  fmt.Printf("opened ports: %v\n", ports)
}
```

我们的代码已经执行的很快了，但是由于超时的原因，我们需要等待很久才能收到返回的错误信息。我们可以假设如果我们200毫秒内没有收到服务器的回应，就不再继续等待。

```
func isOpen(host string, port int, timeout time.Duration) bool {
    time.Sleep(time.Millisecond * 1)
    conn, err := net.DialTimeout("tcp", fmt.Sprintf("%s:%d", host, port), timeou
t)
    if err == nil {
        _ = conn.Close()
        return true
    }

    return false
}

func main() {
    ports := []int{}

    wg := &sync.WaitGroup{}
    timeout := time.Millisecond * 200
    for port := 1; port < 100; port++ {
        wg.Add(1)
        go func(p int) {
            opened := isOpen("google.com", p, timeout)
            if opened {
                ports = append(ports, p)
```

```
            }
            wg.Done()
        }(port)
    }

    wg.Wait()
    fmt.Printf("opened ports: %v\n", ports)
}
```

至此，我们就得到了一个简单的端口扫描器。但有些不好的是，不能很方便的修改域名地址以及端口号范围，我们必须要重新编译代码才可以。Go还有一个很不错的包叫做 flag 。

flag 包可以帮助我们编写命令行程序。我们可以配置每个字符串或数字。我们为主机名及要测试的端口范围和连接超时添加参数。

```
func main() {
    hostname := flag.String("hostname", "", "hostname to test")
    startPort := flag.Int("start-port", 80, "the port on which the scanning star
ts")
    endPort := flag.Int("end-port", 100, "the port from which the scanning ends"
)
    timeout := flag.Duration("timeout", time.Millisecond * 200, "timeout")
    flag.Parse()

    ports := []int{}

    wg := &sync.WaitGroup{}
    for port := *startPort; port <= *endPort; port++ {
        wg.Add(1)
        go func(p int) {
            opened := isOpen(*hostname, p, *timeout)
            if opened {
                ports = append(ports, p)
            }
            wg.Done()
        }(port)
    }

    wg.Wait()
    fmt.Printf("opened ports: %v\n", ports)
}
```

如果我们想要显示如何使用，我们可以添加一个 -h 参数，来显示使用说明。整个项目不到50行的代码，我们使用到了并行、flag 及 net 包。

唯一的问题就是，现在这个程序会有竞争条件。在只扫描少数端口时，速度比较慢，可能不会出现，但确实存在这个问题。所以我们需要使用 mutex 来修复它。

```
wg := &sync.WaitGroup{}
mutex := &sync.Mutex{}
for port := *startPort; port <= *endPort; port++ {
    wg.Add(1)
    go func(p int) {
        opened := isOpen(*hostname, p, *timeout)
        if opened {
            mutex.Lock()
            ports = append(ports, p)
            mutex.Unlock()
        }
        wg.Done()
    }(port)
}
```

我们本次只是简单的实现端口扫描的功能。如果大家喜欢编写这种工具，可以加入自己的理解或特性。参照 nmap 等著名扫描器的实现思路，用Go来打造自己的扫描器，从而加深对网络编程的理解。

本文转自：https://mp.weixin.qq.com/s/OhS_RQZojJbkenOSS_tEng

定时任务

cron

gocron

cron

cron 表达式的基本格式

用过 linux 的应该对 cron 有所了解。linux 中可以通过 crontab -e 来配置定时任务。不过，linux 中的 cron 只能精确到分钟。而我们这里要讨论的 Go 实现的 cron 可以精确到秒，除了这点比较大的区别外，cron 表达式的基本语法是类似的。（如果使用过 Java 中的 Quartz，对 cron 表达式应该比较了解，而且它和这里我们将要讨论的 Go 版 cron 很像，也都精确到秒）cron(计划任务)，顾名思义，按照约定的时间，定时的执行特定的任务（job）。cron 表达式表达了这种约定。 cron 表达式代表了一个时间集合，使用 6 个空格分隔的字段表示。

字段名	是否必须	允许的值	允许的特定字符
秒(Seconds)	是	0-59	* / , -
分(Minutes)	是	0-59	* / , -
时(Hours)	是	0-23	* / , -
日(Day of month)	是	1-31	* / , – ?
月(Month)	是	1-12 or JAN-DEC	* / , -
星期(Day of week)	否	0-6 or SUM-SAT	* / , – ?

注：

1）月(Month)和星期(Day of week)字段的值不区分大小写，如：SUN、Sun 和 sun 是一样的。
2）星期 (Day of week)字段如果没提供，相当于是 *

特殊字符说明

1）星号 `(*)`
表示 cron 表达式能匹配该字段的所有值。如在第5个字段使用星号(month)，表示每个月

2）斜线(/)
表示增长间隔，如第1个字段(minutes) 值是 3-59/15，表示每小时的第3分钟开始执行一次，之后每隔 15 分钟执行一次（即 3、18、33、48 这些时间点执行），这里也可以表示为：3/15

3）逗号(,)
用于枚举值，如第6个字段值是 MON,WED,FRI，表示 星期一、三、五 执行

4）连字号(-)
表示一个范围，如第3个字段的值为 9-17 表示 9am 到 5pm 直接每个小时（包括9和17）

5）问号(?)
只用于日(Day of month)和星期(Day of week)，\表示不指定值，可以用于代替 *

cron举例说明

- 每隔5秒执行一次： `*/5 * * * * ?`
- 每隔1分钟执行一次： `0 */1 * * * ?`
- 每天23点执行一次： `0 0 23 * * ?`
- 每天凌晨1点执行一次： `0 0 1 * * ?`
- 每月1号凌晨1点执行一次： `0 0 1 1 * ?`
- 在26分、29分、33分执行一次： `0 26,29,33 * * * ?`
- 每天的0点、13点、18点、21点都执行一次： `0 0 0,13,18,21 * * ?`

示例

最简单crontab任务

根据官网手册写法

```
package main

import (
    "github.com/robfig/cron"
    "log"
)

func main() {
    i := 0
    c := cron.New()
    spec := "*/5 * * * * ?"
    _,err:=c.AddFunc(spec, func() {
        i++
        log.Println("cron running:", i)
    })
    fmt.Println(err)
```

```
    c.Start()

    select{}
}
```

会报一个错误

```
    expected exactly 5 fields, found 6: [*/5 * * * * ?]
```

这就尴尬了，查询大量资料，把代码修改为一下代码就可以完美运行了

```
package main

import (
    "fmt"

    "github.com/robfig/cron"
)

// 返回一个支持至 秒 级别的 cron
func newWithSeconds() *cron.Cron {
    secondParser := cron.NewParser(cron.Second | cron.Minute |
        cron.Hour | cron.Dom | cron.Month | cron.DowOptional | cron.Descriptor)
    return cron.New(cron.WithParser(secondParser), cron.WithChain())
}
func main() {
    i := 0
    c := newWithSeconds()
    spec := "0 */1 * * * ?"  //一分钟运行一次
    c.AddFunc(spec, func() {
        i++
        fmt.Println("cron running:", i)
    })
    c.Start()

    select {}
}
```

输出结果：

```
    cron running: 1
    cron running: 2
    cron running: 3
```

多个定时crontab任务

```
package main

import (
	"fmt"
	"log"

	"github.com/robfig/cron"
)

type TestJob struct {
}

func (this TestJob) Run() {
	fmt.Println("testJob1...")
}

type Test2Job struct {
}

func (this Test2Job) Run() {
	fmt.Println("testJob2...")
}

// 返回一个支持至 秒 级别的 cron
func newWithSeconds() *cron.Cron {
	secondParser := cron.NewParser(cron.Second | cron.Minute |
		cron.Hour | cron.Dom | cron.Month | cron.DowOptional | cron.Descriptor)
	return cron.New(cron.WithParser(secondParser), cron.WithChain())
}
func main() {
	i := 0
	c := newWithSeconds()
	//AddFunc
	spec := "*/5 * * * * ?"
	c.AddFunc(spec, func() {
		i++
		log.Println("cron running:", i)
	})

	//AddJob方法
	c.AddJob(spec, TestJob{})
	c.AddJob(spec, Test2Job{})

	//启动计划任务
```

```
    c.Start()

    //关闭着计划任务, 但是不能关闭已经在执行中的任务.
    defer c.Stop()

    select {}
}
```

输出结果:

```
    2019/11/14 17:23:10 cron running: 1
    testJob2...
    testJob1...
    2019/11/14 17:23:15 cron running: 2
    testJob2...
    testJob1...
```

gocron

jasonlvhit/gocron

安装:

```
go get -u github.com/jasonlvhit/gocron
```

每隔1秒执行一个任务，每隔4秒执行另一个任务：

```
package main

import (
    "fmt"
    "time"

    "github.com/jasonlvhit/gocron"
)

func task() {
    fmt.Println("I am runnning task.", time.Now())
}
func superWang() {
    fmt.Println("I am runnning superWang.", time.Now())
}

func main() {
    s := gocron.NewScheduler()
    s.Every(1).Seconds().Do(task)
    s.Every(4).Seconds().Do(superWang)

    sc := s.Start() // keep the channel
    go test(s, sc)   // wait
    <-sc             // it will happens if the channel is closed
}

func test(s *gocron.Scheduler, sc chan bool) {
    time.Sleep(8 * time.Second)
    s.Remove(task) //remove task
    time.Sleep(8 * time.Second)
    s.Clear()
    fmt.Println("All task removed")
```

```
    close(sc) // close the channel
}
```

输出结果：

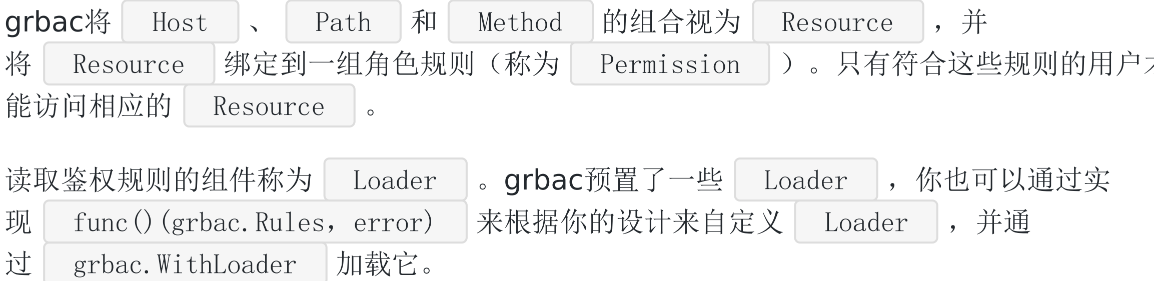
```
    I am runnning task. 2019-11-15 09:37:07.218128145 +0800 CST m=+1.000414542
    I am runnning task. 2019-11-15 09:37:08.218062127 +0800 CST m=+2.000348513
    I am runnning task. 2019-11-15 09:37:09.218047138 +0800 CST m=+3.000333512
    I am runnning superWang. 2019-11-15 09:37:10.218058701 +0800 CST m=+4.000345
070
    I am runnning task. 2019-11-15 09:37:10.218100706 +0800 CST m=+4.000387079
    I am runnning task. 2019-11-15 09:37:11.218033085 +0800 CST m=+5.000319455
    I am runnning task. 2019-11-15 09:37:12.21804561 +0800 CST m=+6.000331978
    I am runnning task. 2019-11-15 09:37:13.218053083 +0800 CST m=+7.000339456
    I am runnning superWang. 2019-11-15 09:37:14.218038711 +0800 CST m=+8.000325
086
    I am runnning superWang. 2019-11-15 09:37:18.218061128 +0800 CST m=+12.00034
7516
    I am runnning superWang. 2019-11-15 09:37:22.218054256 +0800 CST m=+16.00034
0629
    All task removed
```

基于角色的访问控制框架

Grbac是一个快速，优雅和简洁的RBAC框架。它支持增强的通配符并使用Radix树匹配HTTP请求。令人惊奇的是，您可以在任何现有的数据库和数据结构中轻松使用它。

项目地址：https://github.com/storyicon/grbac

grbac的作用是确保指定的资源只能由指定的角色访问。请注意，grbac不负责存储鉴权规则和分辨“当前请求发起者具有哪些角色”，更不负责角色的创建、分配等。这意味着您应该首先配置规则信息，并提供每个请求的发起者具有的角色。



grbac将 `Host` 、 `Path` 和 `Method` 的组合视为 `Resource` ，并将 `Resource` 绑定到一组角色规则（称为 `Permission` ）。只有符合这些规则的用户才能访问相应的 `Resource` 。

读取鉴权规则的组件称为 `Loader` 。grbac预置了一些 `Loader` ，你也可以通过实现 `func()(grbac.Rules, error)` 来根据你的设计来自定义 `Loader` ，并通过 `grbac.WithLoader` 加载它。

- 1. 最常见的用例
- 2. 概念
  - 2.1. Rule
  - 2.2. Resource
  - 2.3. Permission
  - 2.4. Loader
- 3. 其他例子
  - 3.1. gin && grbac.WithJSON
  - 3.2. echo && grbac.WithYaml
  - 3.3. iris && grbac.WithRules
  - 3.4. ace && grbac.WithAdvancedRules
  - 3.5. gin && grbac.WithLoader
- 4. 增强的通配符
- 5. 运行效率
- 6. 生产环境

1. 最常见的用例

下面是最常见的用例，它使用 `gin` ，并将 `grbac` 包装成了一个中间件。通过这个例子，你可以很容易地知道如何在其他http框架中使用 `grbac` （比如 `echo` ， `iris` ， `ace` 等）：

```
package main

import (
    "github.com/gin-gonic/gin"
    "github.com/storyicon/grbac"
    "net/http"
    "time"
)

func LoadAuthorizationRules() (rules grbac.Rules, err error) {
    // 在这里实现你的逻辑
    // ...
    // 你可以从数据库或文件加载授权规则
    // 但是你需要以 grbac.Rules 的格式返回你的身份验证规则
    // 提示：你还可以将此函数绑定到golang结构体
    return
}

func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

func Authorization() gin.HandlerFunc {
    // 在这里，我们通过“grbac.WithLoader”接口使用自定义Loader功能
    // 并指定应每分钟调用一次LoadAuthorizationRules函数以获取最新的身份验证规则。
    // Grbac还提供一些现成的Loader：
    // grbac.WithYAML
    // grbac.WithRules
    // grbac.WithJSON
    // ...
    rbac, err := grbac.New(grbac.WithLoader(LoadAuthorizationRules, time.Minute))
    if err != nil {
        panic(err)
    }
    return func(c *gin.Context) {
        roles, err := QueryRolesByHeaders(c.Request.Header)
```

```
		if err != nil {
			c.AbortWithError(http.StatusInternalServerError, err)
			return
		}
		state, _ := rbac.IsRequestGranted(c.Request, roles)
		if !state.IsGranted() {
			c.AbortWithStatus(http.StatusUnauthorized)
			return
		}
	}
}

func main(){
	c := gin.New()
	c.Use(Authorization())

	// 在这里通过c.Get、c.Post等函数绑定你的API
	// ...

	c.Run(":8080")
}
```

2. 概念

这里有一些关于 `grbac` 的概念。这很简单，你可能只需要三分钟就能理解。

2.1. Rule

```
// Rule即规则，用于定义Resource和Permission之间的关系
type Rule struct {
	// ID决定了Rule的优先级。
	// ID值越大意味着Rule的优先级越高。
	// 当请求被多个规则同时匹配时，grbac将仅使用具有最高ID值的规则。
	// 如果有多个规则同时具有最大的ID，则将随机使用其中一个规则。
	ID int `json:"id"`
	*Resource
	*Permission
}
```

如你所见， `Rule` 由三部分组成： `ID` ， `Resource` 和 `Permission` 。
“ID”确定规则的优先级。
当请求同时满足多个规则时（例如在通配符中），

`grbac` 将选择具有最高ID的那个，然后使用其权限定义进行身份验证。
如果有多个规则同时具有最大的ID，则将随机使用其中一个规则（所以请避免这种情况）。

下面有一个非常简单的例子：

```yaml
#Rule
- id: 0
  # Resource
  host: "*"
  path: "**"
  method: "*"
  # Permission
  authorized_roles:
  - "*"
  forbidden_roles: []
  allow_anyone: false

#Rule
- id: 1
  # Resource
  host: domain.com
  path: "/article"
  method: "{DELETE,POST,PUT}"
  # Permission
  authorized_roles:
  - editor
  forbidden_roles: []
  allow_anyone: false
```

在以yaml格式编写的此配置文件中，ID=0 的规则表明任何具有任何角色的人都可以访问所有资源。
但是ID=1的规则表明只有 `editor` 可以对文章进行增删改操作。
这样，除了文章的操作只能由 `editor` 访问之外，任何具有任何角色的人都可以访问所有其他资源。

2.2. Resource

```go
type Resource struct {
    // Host 定义资源的Host，允许使用增强的通配符。
    Host string `json:"host"`
    // Path 定义资源的Path，允许使用增强的通配符。
    Path string `json:"path"`
    // Method 定义资源的Method，允许使用增强的通配符。
```

```
    Method string `json:"method"`
}
```

Resource用于描述Rule适用的资源。
当执行 `IsRequestGranted(c.Request, roles)` 时，grbac首先将当前的 `Request` 与所有 `Rule` 中的 `Resources` 匹配。

Resource的每个字段都支持增强的通配符

2.3. Permission

```
// Permission用于定义权限控制信息
type Permission struct {
    // AuthorizedRoles定义允许访问资源的角色
    // 支持的类型: 非空字符串，*
    //      *: 意味着任何角色，但访问者应该至少有一个角色，
    //      非空字符串：指定的角色
    AuthorizedRoles []string `json:"authorized_roles"`
    // ForbiddenRoles 定义不允许访问指定资源的角色
    // ForbiddenRoles 优先级高于AuthorizedRoles
    // 支持的类型：非空字符串，*
    //      *: 意味着任何角色，但访问者应该至少有一个角色，
    //      非空字符串：指定的角色
    //
    ForbiddenRoles []string `json:"forbidden_roles"`
    // AllowAnyone的优先级高于 ForbiddenRoles、AuthorizedRoles
    // 如果设置为true，任何人都可以通过验证。
    // 请注意，这将包括“没有角色的人”。
    AllowAnyone bool `json:"allow_anyone"`
}
```

“Permission”用于定义绑定到的“Resource”的授权规则。
这是易于理解的，当请求者的角色符合“Permission”的定义时，他将被允许访问Resource，否则他将被拒绝访问。

为了加快验证的速度， `Permission` 中的字段不支持“增强的通配符”。
在 `AuthorizedRoles` 和 `ForbiddenRoles` 中只允许 `*` 表示所有。

2.4. Loader

Loader用于加载Rule。 grbac预置了一些加载器，你也可以通过实现 `func()(grbac.Rules, error)` 来自定义加载器并通过 `grbac.WithLoader` 加载它。

method	description

method	description
WithJSON(path, interval)	定期从 `json` 文件加载规则配置
WithYaml(path, interval)	定期从 `yaml` 文件加载规则配置
WithRules(Rules)	从 `grbac.Rules` 加载规则配置
WithAdvancedRules(loader.AdvancedRules)	以一种更紧凑的方式定义Rule，并使用 `loader.AdvancedRules` 加载
WithLoader(loader func()(Rules, error), interval)	使用自定义函数定期加载规则

`interval` 定义了Rules的重载周期。
当 `interval <0` 时， `grbac` 会放弃周期加载Rules配置;
当 `interval∈[0,1s)` 时， `grbac` 会自动将 `interval` 设置为 `5s` ；

3. 其他例子

这里有一些简单的例子，可以让你更容易理解 `grbac` 的工作原理。
虽然 `grbac` 在大多数http框架中运行良好，但很抱歉我现在只使用gin，所以如果下面的例子中有一些缺陷，请告诉我。

3.1. gin && grbac.WithJSON

如果你想在 `JSON` 文件中编写配置文件，你可以通过 `grbac.WithJSON(filepath, interval)` 加载它， `filepath` 是你的json文件路径，并且grbac将每隔interval重新加载一次文件。 。

```
[
    {
        "id": 0,
        "host": "*",
        "path": "**",
        "method": "*",
        "authorized_roles": [
            "*"
        ],
        "forbidden_roles": [
            "black_user"
        ],
        "allow_anyone": false
```

```
    },
    {
        "id":1,
        "host": "domain.com",
        "path": "/article",
        "method": "{DELETE,POST,PUT}",
        "authorized_roles": ["editor"],
        "forbidden_roles": [],
        "allow_anyone": false
    }
]
```

以上是“JSON”格式的身份验证规则示例。它的结构基于grbac.Rules。

```
func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

func Authentication() gin.HandlerFunc {
    rbac, err := grbac.New(grbac.WithJSON("config.json", time.Minute * 10))
    if err != nil {
        panic(err)
    }
    return func(c *gin.Context) {
        roles, err := QueryRolesByHeaders(c.Request.Header)
        if err != nil {
            c.AbortWithError(http.StatusInternalServerError, err)
            return
        }

        state, err := rbac.IsRequestGranted(c.Request, roles)
        if err != nil {
            c.AbortWithStatus(http.StatusInternalServerError)
            return
        }

        if !state.IsGranted() {
            c.AbortWithStatus(http.StatusUnauthorized)
            return
        }
```

```
    }
}

func main(){
    c := gin.New()
    c.Use(Authentication())

    // 在这里通过c.Get、c.Post等函数绑定你的API
    // ...

    c.Run(":8080")
}
```

3.2. echo && grbac.WithYaml

如果你想在 `YAML` 文件中编写配置文件，你可以通过 `grbac.WithYAML(file, interval)` 加载它， `file` 是你的yaml文件路径，并且grbac将每隔一个interval重新加载一次文件。

```
#Rule
- id: 0
  # Resource
  host: "*"
  path: "**"
  method: "*"
  # Permission
  authorized_roles:
  - "*"
  forbidden_roles: []
  allow_anyone: false

#Rule
- id: 1
  # Resource
  host: domain.com
  path: "/article"
  method: "{DELETE,POST,PUT}"
  # Permission
  authorized_roles:
  - editor
  forbidden_roles: []
  allow_anyone: false
```

以上是“YAML”格式的认证规则的示例。它的结构基于grbac.Rules。

```
func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

func Authentication() echo.MiddlewareFunc {
    rbac, err := grbac.New(grbac.WithYAML("config.yaml", time.Minute * 10))
    if err != nil {
            panic(err)
    }
    return func(echo.HandlerFunc) echo.HandlerFunc {
        return func(c echo.Context) error {
            roles, err := QueryRolesByHeaders(c.Request().Header)
            if err != nil {
                c.NoContent(http.StatusInternalServerError)
                return nil
            }
            state, err := rbac.IsRequestGranted(c.Request(), roles)
            if err != nil {
                c.NoContent(http.StatusInternalServerError)
                return nil
            }
            if state.IsGranted() {
                return nil
            }
            c.NoContent(http.StatusUnauthorized)
            return nil
        }
    }
}

func main(){
    c := echo.New()
    c.Use(Authentication())

    // 在这里通过c.Get、c.Post等函数绑定你的API
    // ...

}
```

3.3. iris && grbac.WithRules

如果你想直接在代码中编写认证规则， `grbac.WithRules(rules)` 提供了这种方式，你可以像这样使用它：

```
func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

func Authentication() iris.Handler {
    var rules = grbac.Rules{
        {
            ID: 0,
            Resource: &grbac.Resource{
                Host: "*",
                Path: "**",
                Method: "*",
            },
            Permission: &grbac.Permission{
                AuthorizedRoles: []string{"*"},
                ForbiddenRoles: []string{"black_user"},
                AllowAnyone: false,
            },
        },
        {
            ID: 1,
            Resource: &grbac.Resource{
                Host: "domain.com",
                Path: "/article",
                Method: "{DELETE,POST,PUT}",
            },
            Permission: &grbac.Permission{
                AuthorizedRoles: []string{"editor"},
                ForbiddenRoles: []string{},
                AllowAnyone: false,
            },
        },
    }
    rbac, err := grbac.New(grbac.WithRules(rules))
    if err != nil {
        panic(err)
    }
    return func(c context.Context) {
```

```
        roles, err := QueryRolesByHeaders(c.Request().Header)
        if err != nil {
            c.StatusCode(http.StatusInternalServerError)
            c.StopExecution()
            return
        }
        state, err := rbac.IsRequestGranted(c.Request(), roles)
        if err != nil {
            c.StatusCode(http.StatusInternalServerError)
            c.StopExecution()
            return
        }
        if !state.IsGranted() {
            c.StatusCode(http.StatusUnauthorized)
            c.StopExecution()
            return
        }
    }
}

func main(){
    c := iris.New()
    c.Use(Authentication())

    // 在这里通过c.Get、c.Post等函数绑定你的API
    // ...

}
```

3.4. ace && grbac.WithAdvancedRules

如果你想直接在代码中编写认证规则， `grbac.WithAdvancedRules(rules)` 提供了这种方式，你可以像这样使用它：

```
func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

func Authentication() ace.HandlerFunc {
    var advancedRules = loader.AdvancedRules{
```

```
        {
            Host: []string{"*"},
            Path: []string{"**"},
            Method: []string{"*"},
            Permission: &grbac.Permission{
                AuthorizedRoles: []string{},
                ForbiddenRoles: []string{"black_user"},
                AllowAnyone: false,
            },
        },
        {
            Host: []string{"domain.com"},
            Path: []string{"/article"},
            Method: []string{"PUT","DELETE","POST"},
            Permission: &grbac.Permission{
                AuthorizedRoles: []string{"editor"},
                ForbiddenRoles: []string{},
                AllowAnyone: false,
            },
        },
    }
    auth, err := grbac.New(grbac.WithAdvancedRules(advancedRules))
    if err != nil {
        panic(err)
    }
    return func(c *ace.C) {
        roles, err := QueryRolesByHeaders(c.Request.Header)
        if err != nil {
        c.AbortWithStatus(http.StatusInternalServerError)
            return
        }
        state, err := auth.IsRequestGranted(c.Request, roles)
        if err != nil {
            c.AbortWithStatus(http.StatusInternalServerError)
            return
        }
        if !state.IsGranted() {
            c.AbortWithStatus(http.StatusUnauthorized)
            return
        }
    }
}

func main(){
    c := ace.New()
    c.Use(Authentication())
```

```
    // 在这里通过c.Get、c.Post等函数绑定你的API
    // ...

}
```

`loader.AdvancedRules` 试图提供一种比 `grbac.Rules` 更紧凑的定义鉴权规则的方法。

3.5. gin && grbac.WithLoader

```
func QueryRolesByHeaders(header http.Header) (roles []string,err error){
    // 在这里实现你的逻辑
    // ...
    // 这个逻辑可能是从请求的Headers中获取token，并且根据token从数据库中查询用户的相应角色。
    return roles, err
}

type MySQLLoader struct {
    session *gorm.DB
}

func NewMySQLLoader(dsn string) (*MySQLLoader, error) {
    loader := &MySQLLoader{}
    db, err := gorm.Open("mysql", dsn)
    if err != nil {
        return nil, err
    }
    loader.session = db
    return loader, nil
}

func (loader *MySQLLoader) LoadRules() (rules grbac.Rules, err error) {
    // 在这里实现你的逻辑
    // ...
    // 你可以从数据库或文件加载授权规则
    // 但是你需要以 grbac.Rules 的格式返回你的身份验证规则
    // 提示：你还可以将此函数绑定到golang结构体
    return
}

func Authentication() gin.HandlerFunc {
    loader, err := NewMySQLLoader("user:password@/dbname?charset=utf8&parseTime=
```

```
True&loc=Local")
    if err != nil {
        panic(err)
    }
    rbac, err := grbac.New(grbac.WithLoader(loader.LoadRules, time.Second * 5))
    if err != nil {
        panic(err)
    }
    return func(c *gin.Context) {
        roles, err := QueryRolesByHeaders(c.Request.Header)
        if err != nil {
            c.AbortWithStatus(http.StatusInternalServerError)
            return
        }

        state, err := rbac.IsRequestGranted(c.Request, roles)
        if err != nil {
            c.AbortWithStatus(http.StatusInternalServerError)
            return
        }
        if !state.IsGranted() {
            c.AbortWithStatus(http.StatusUnauthorized)
            return
        }
    }
}

func main(){
    c := gin.New()
    c.Use(Authorization())

    // 在这里通过c.Get、c.Post等函数绑定你的API
    // ...

    c.Run(":8080")
}
```

4. 增强的通配符

`Wildcard` 支持的语法：

```
pattern:
  { term }
term:
```

```
    '*'         匹配任何非路径分隔符的字符串
    '**'        匹配任何字符串，包括路径分隔符.
    '?'         匹配任何单个非路径分隔符
    '[' [ '^' ] { character-range } ']'
          character class (must be non-empty)
    '{' { term } [ ',' { term } ... ] '}'
    c           匹配字符 c (c != '*', '?', '\\', '[')
    '\\' c      匹配字符 c

character-range:
    c           匹配字符 c (c != '\\', '-', ']')
    '\\' c      匹配字符 c
    lo '-' hi   匹配字符 c for lo <= c <= hi
```

5. 运行效率

```
➜ gos test -bench=.
goos: linux
goarch: amd64
pkg: github.com/storyicon/grbac/pkg/tree
BenchmarkTree_Query                   2000           541397 ns/op
BenchmarkTree_Foreach_Query           2000           1360719 ns/op
PASS
ok      github.com/storyicon/grbac/pkg/tree     13.182s
```

测试用例包含1000个随机规则，“BenchmarkTree_Query”和“BenchmarkTree_Foreach_Query”函数分别测试四个请求：

```
541397/(4*1e9)=0.0001s
```

当有1000条规则时，每个请求的平均验证时间为“0.0001s”，这很快（大多数时间在通配符的匹配上）。

6. 生产环境

`grbac` 已经被以下企业用于生产环境:

uuid

###什么是uuid?

uuid是Universally Unique Identifier的缩写，即通用唯一识别码。

uuid的目的是让分布式系统中的所有元素，都能有唯一的辨识资讯，而不需要透过中央控制端来做辨识资讯的指定。如此一来，每个人都可以建立不与其它人冲突的 uuid。

A universally unique identifier (UUID) is a 128-bit number used to identify information in computer systems.

例如java中生成uuid：

```
package com.mytest;
import java.util.UUID;
public class UTest {
    public static void main(String[] args) {
        UUID uuid = UUID.randomUUID();
        System.out.println(uuid);
}}
```

c++中生成uuid：

```
#pragma comment(lib, "rpcrt4.lib")
#include <windows.h>
#include <iostream>

using namespace std;

int main()
{
    UUID uuid;
    UuidCreate(&uuid);
    char *str;
    UuidToStringA(&uuid, (RPC_CSTR*)&str);
    cout<<str<<endl;
    RpcStringFreeA((RPC_CSTR*)&str);
    return 0;
}
```

go生成uuid：

目前，golang中的uuid还没有纳入标准库，我们使用github上的开源库：

```
    go get -u github.com/satori/go.uuid
```

使用：

```
package main

import (
    "fmt"

    uuid "github.com/satori/go.uuid"
)

func main() {
    // 创建
    u1, _ := uuid.NewV4()
    fmt.Printf("UUIDv4: %s\n", u1)

    // 解析
    u2, err := uuid.FromString("f5394eef-e576-4709-9e4b-a7c231bd34a4")
    if err != nil {
        fmt.Printf("Something gone wrong: %s", err)
        return
    }
    fmt.Printf("Successfully parsed: %s", u2)
}
```

uuid在websocket中使用

这里就是一个简单的使用而已，在websocket中为每一个连接的客户端分配一个uuid。

golang中可以使用github.com/gorilla/websocket为我们提供的websocket开发包。

声明一个客户端结构体：

```
    type Client struct {
        id      string
        socket *websocket.Conn
        send    chan []byte
    }
```

使用：

```
    client := &Client{id: uuid.NewV4().String(), socket: conn, send: make(chan
 []byte)}
```

支付宝支付

本文采用沙箱环境

开启沙箱

文档：https://docs.open.alipay.com/200/105311/
沙箱地址：https://openhome.alipay.com/platform/appDaily.htm

生成秘钥（已弃用，请直接使用步骤 3）

本文中的签名方法默认为 RSA2，采用支付宝提供的 RSA 签名 & 验签工具 生成秘钥时，秘钥的格式必须为 PKCS1，秘钥长度推荐 2048。所以在支付宝管理后台请注意配置 RSA2 (SHA256) 密钥。
生成秘钥对之后，将公钥提供给支付宝（通过支付宝后台上传）对我们请求的数据进行签名验证，我们的代码中将使用私钥对请求数据签名。

RSA 签名和验证工具下载：https://docs.open.alipay.com/291/105971

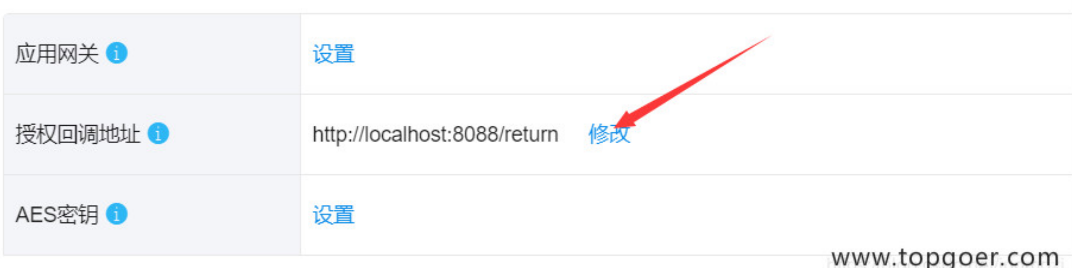
- 下载之后解压
- 双击 RSA签名验签工具.bat
- 秘钥格式选择 PKCS1
- 点击生成秘钥
- 复制公钥
- 回到沙箱中，点击查看应用公钥，然后点击修改

- 保存好私钥，我们一会需要在代码中用到
- 复制支付宝公钥，代码中验证需要用到

APPID	2016091800540000
支付宝网关	https://openapi.alipaydev.com/gateway.do
RSA2(SHA256)密钥(推荐)	查看应用公钥 | 查看支付宝公钥
RSA(SHA1)密钥	查看应用公钥 | 查看支付宝公钥

www.topgoer.com

- 配置支付成功后的回调地址



证书认证

目前新创建的支付宝应用只支持证书方式认证，已经弃用之前的公钥和私钥的方式

公钥秘钥说明

我们生成秘钥对之后，将公钥提供给支付宝（通过支付宝后台上传）对我们请求的数据进行签名验证，我们的代码中使用私钥对请求数据签名。

- 证书签名请求文件（用来提交给支付宝后台生成证书的）
- 应用私钥（调用支付宝接口的时候，我们需要使用该私钥对参数进行签名）
- 支付宝公钥证书（用来验证我们的签名的，现在已经被支付宝公钥证书取代）

下载生成工具

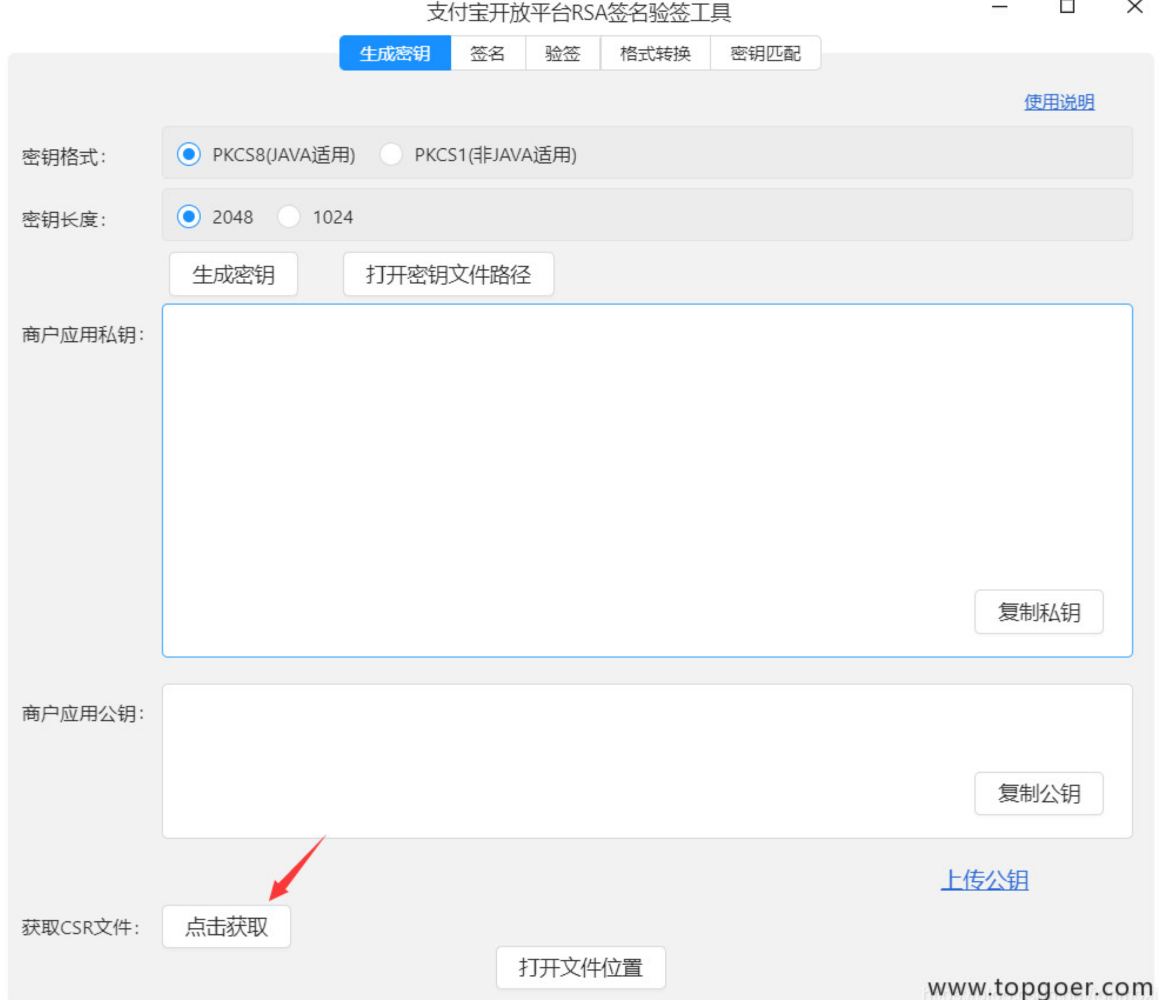
用。加密的过程为系统使用公钥（public key）进行加密，并将密文发送到解密者，解密者用私钥
开发者要保证接口中使用的私钥与此处的公钥匹配，否则无法调用接口。开发者可通过下方链接
WINDOWS（**windows版本工具请不要安装在含有空格的目录路径下**）
MAC_OSX

WINDOWS老版本下载地址：WINDOWS
MAC OS老版本下载地址：MAC_OSX
注意：
密钥和应用（APPID）一一对应，即开发者需要为名下的每个应用分别设置密钥，且不同应用的

www.topgoer.com

生成 csr 证书签名请求文件

工具安装好之后打开，点击获取



输入信息

主要是组织/公司这块一定要写的和你支付宝中应用的名一样，不然不会通过的，填写完毕之后点击生成CSR文件，点击页面的打开文件位置，就可以看到三个文件了，分别是证书签名请求文件，应用公钥，应用私钥

上传 CSR 证书签名请求文件

回到支付宝后台，点击 接口加签方式 设置，选择公钥证书，点击上次 CSR 生成证书，把我们刚才生成的那个证书 (.csr) 上传进去

下载证书

上传好之后，会弹出让你下载证书的页面，把那三个证书都下载下来，分别是：应用公钥证书，支付宝公钥证书，支付宝根证书

代码部分

下载第三方库

```
go get github.com/smartwalle/alipay/v3
```

网页扫码支付

```
package main

import (
	"fmt"
	"github.com/smartwalle/alipay"
	"net/http"
	"os/exec"
	"strings"
	"time"
)

var (
	// appId
	appId = ""
	// 应用公钥（跟csr文件同级目录）
	aliPublicKey = ""
	// 应用私钥（跟csr文件同级目录）
	privateKey = ""
	client, _ = alipay.New(appId, aliPublicKey, privateKey, false)
)

func init() {
	client.LoadAppPublicCert("应用公钥证书")
	client.LoadAliPayPublicCert("支付宝公钥证书")
	client.LoadAliPayRootCert("支付宝根证书")
}

//网站扫码支付
func WebPageAlipay() {
	pay := alipay.AliPayTradePagePay{}
	// 支付成功之后，支付宝将会重定向到该 URL
	pay.ReturnURL = "http://localhost:8088/return"
	//支付标题
	pay.Subject = "支付宝支付测试"
	//订单号，一个订单号只能支付一次
	pay.OutTradeNo = time.Now().String()
	//销售产品码，与支付宝签约的产品码名称,目前仅支持FAST_INSTANT_TRADE_PAY
	pay.ProductCode = "FAST_INSTANT_TRADE_PAY"
	//金额
	pay.TotalAmount = "0.01"

	url, err := client.TradePagePay(pay)
	if err != nil {
		fmt.Println(err)
	}
```

```
    payURL := url.String()
    //这个 payURL 即是用于支付的 URL，可将输出的内容复制，到浏览器中访问该 URL
即可打开支付页面。
    fmt.Println(payURL)

    //打开默认浏览器
    payURL = strings.Replace(payURL,"&","^&",-1)
    exec.Command("cmd", "/c", "start",payURL).Start()
}

//手机客户端支付
func WapAlipay() {
    pay := alipay.AliPayTradeWapPay{}
    // 支付成功之后，支付宝将会重定向到该 URL
    pay.ReturnURL = "http://localhost:8088/return"
    //支付标题
    pay.Subject = "支付宝支付测试"
    //订单号，一个订单号只能支付一次
    pay.OutTradeNo = time.Now().String()
    //商品code
    pay.ProductCode = time.Now().String()
    //金额
    pay.TotalAmount = "0.01"

    url, err := client.TradeWapPay(pay)
    if err != nil {
        fmt.Println(err)
    }
    payURL := url.String()
    //这个 payURL 即是用于支付的 URL，可将输出的内容复制，到浏览器中访问该 URL
即可打开支付页面。
    fmt.Println(payURL)
    //打开默认浏览器
    payURL = strings.Replace(payURL,"&","^&",-1)
    exec.Command("cmd", "/c", "start",payURL).Start()
}

func main() {
    //生成支付URL
    WapAlipay()
    // 支付成功之后的返回URL页面
    http.HandleFunc("/return", func(rep http.ResponseWriter, req *http.Request)
{
        req.ParseForm()
        ok, err := client.VerifySign(req.Form)
        if err == nil && ok {
```

```
                rep.Write([]byte("支付成功"))
            }
        })
        fmt.Println("server start....")
        http.ListenAndServe(":8088", nil)
}
```

转自：https://learnku.com/go/t/31515

微信支付

这是用Golang封装了微信的所有API接口的SDK，并自动生成和解析XML数据，包括微信支付、公众号、小程序、移动端的工具函数。

- 支持境内普通商户和境内服务商(境外和银行服务商没有条件测试)。
- 支持全局配置应用ID、商家ID等信息。
- 全部参数和返回值均使用 `struct` 类型传递，而不是 `map` 类型。

安装

```
go get -u gitee.com/cuckoopark/wechat
```

初始化

```
const (
    isProd       = true                                // 生产环境或沙盒环境
    isMch        = false                               // 是否是企业模式，仅当调用企业付款时为true
    serviceType  = wechat.ServiceTypeNormalDomestic // 普通商户或服务商等类型
    apiKey       = "xxxxxxxx"                          // 微信支付上设置的API Key
    certFilepath = "/xxx/yyy/apiclient_cert.p12"       // 微信证书文件的本地路径，仅部分接口使用，如果不使用这些接口，可以传递空值
)
config := wechat.Config{
    AppId: AppID,
    MchId: MchID,
    SubAppId: SubAppId, // 仅服务商模式有效
    SubMchId: SubMchID, // 仅服务商模式有效
}
client := wechat.NewClient(isProd, isMch, serviceType, apiKey, certFilepath, config)
```

使用

以下是通用的接口，使用上面初始化时生成的实例 `client` 进行相应函数的调用。其中带有 `(*Client)` 字样的接口，需要使用 `wechat.NewClient` 创建的实例对象来调用，而不带的接口，则可以直接使用 `wechat.XXX` 调用。

使用样例：

```go
func Test() {
    // 初始化参数
    body := wechat.QueryOrderBody{}
    body.OutTradeNo = "YgENQFTovdeJdFouNyy3nFVOhGD6ZvPH"
    // 请求订单查询
    wxRsp, err := client.QueryOrder(body)
    if err != nil {
        return
    }
    fmt.Printf("返回值: %+v\n", wxRsp)
}
```

注意事项：

- 参数或返回值的类型，请查看接口对应的文件，里面有 `XXXBody` 和 `XXXResponse` 与之对应。
- 参数或返回值中的常量，请参照constant.go文件。
- 具体使用方法，请参照接口对应的测试文件。

微信支付

对应文件： `wx_pay_xxxxxx.go`

- 提交付款码支付： `(*Client) Micropay` 。
- 统一下单： `(*Client) UnifiedOrder` 。
- 查询订单： `(*Client) QueryOrder` 。
- 关闭订单： `(*Client) CloseOrder` 。
- 撤销订单： `(*Client) Reverse` 。
- 申请退款： `(*Client) Refund` 。
- 查询退款： `(*Client) QueryRefund` 。
- 下载对账单： `(*Client) DownloadBill` 。
- 交易保障(JSAPI)： `(*Client) ReportJsApi` 。
- 交易保障(MICROPAY)： `(*Client) ReportMicropay` 。
- 下载资金账单：TODO，client.DownloadFundFlow()。
- 拉取订单评价数据：TODO，client.BatchQueryComment()。
- 企业付款到零钱： `(*Client) Change` 。
- 查询企业付款到零钱： `(*Client) QueryChange` 。

微信支付回调

对应文件： `wx_notify_xxxxxx.go`

- 支付回调： `(*Client) NotifyPay` 。
- 退款回调： `(*Client) NotifyRefund` 。

微信公众号

对应文件： `wx_service_xxxxxx.go`

- 授权码查询OpenId： `(*Client) OpenIdByAuthCode` 。
- 获取基础支持的AccessToken： `GetBasicAccessToken` 。
- 获取用户基本信息(UnionId机制)： `GetUserInfo` 。
- 获取H5支付签名： `GetH5PaySign` 。

微信小程序

对应文件： `wx_applet_xxxxxx.go`

- 获取小程序支付签名： `GetAppletPaySign` 。
- 获取小程序码： `GetAppletUnlimitQrcode` 。

移动端

对应文件： `wx_app_xxxxxx.go`

- 获取APP支付签名： `GetAppPaySign` 。

文档

- 微信支付文档：https://pay.weixin.qq.com/wiki/doc/api/index.html
- 随机数生成算法：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_3
- 签名生成算法：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_3
- 交易金额：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 交易类型：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 货币类型：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2

- 时间规则：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 时间戳：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 商户订单号：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 银行类型：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=4_2
- 单品优惠功能字段：https://pay.weixin.qq.com/wiki/doc/api/danpin.php?chapter=9_101&index=1
- 代金券或立减优惠：https://pay.weixin.qq.com/wiki/doc/api/micropay.php?chapter=12_1
- 最新县及县以上行政区划代码：https://pay.weixin.qq.com/wiki/doc/api/download/store_adress.csv

开发进度

- 境内普通商户
  - 付款码支付
  - JSAPI支付
  - Native支付
  - APP支付
  - H5支付
  - 小程序支付
  - (TODO) 代金券或立减优惠
  - (TODO) 现金红包
  - 企业付款
- 境内服务商
  - 付款码支付
  - JSAPI支付
  - Native支付
  - APP支付
  - H5支付
  - 小程序支付
  - 现金红包

测试方法

修改 `client_test.go` 中的生成测试Client的代码，调整沙盒/生产环境、普通商户/服务商等选项，或者修改环境变量，来调整商户参数。

环境变量的脚本在 `env` 文件中，修改后加载环境变量：

```
source env
go test
```

TODO

- 测试改为不同情境使用不同的用例。
- 继续调试境内普通商户和境内服务商的其他模块API文档。
- 选择性调试境外接口。
- 继续增加公众号和小程序相关接口。
- 移除 `service` 开头的文件。

微信公众号开发

请跳转

http://wen.topgoer.com/docs/gowechat/gowechat-1cb49i4ees248

爬虫小案例

爬虫步骤

- 明确目标（确定在哪个网站搜索）
- 爬（爬下内容）
- 取（筛选想要的）
- 处理数据（按照你的想法去处理）

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
)

//这个只是一个简单的版本只是获取QQ邮箱并且没有进行封装操作，另外爬出来的数据也没有进行去重操作
var (
    // \d是数字
    reQQEmail = `(\d+)@qq.com`
)

// 爬邮箱
func GetEmail() {
    // 1.去网站拿数据
    resp, err := http.Get("https://tieba.baidu.com/p/6051076813?red_tag=1573533731")
    HandleError(err, "http.Get url")
    defer resp.Body.Close()
    // 2.读取页面内容
    pageBytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "ioutil.ReadAll")
    // 字节转字符串
    pageStr := string(pageBytes)
    //fmt.Println(pageStr)
    // 3.过滤数据，过滤qq邮箱
    re := regexp.MustCompile(reQQEmail)
    // -1代表取全部
```

```
    results := re.FindAllStringSubmatch(pageStr, -1)
    //fmt.Println(results)

    // 遍历结果
    for _, result := range results {
        fmt.Println("email:", result[0])
        fmt.Println("qq:", result[1])
    }
}

// 处理异常
func HandleError(err error, why string) {
    if err != nil {
        fmt.Println(why, err)
    }
}
func main() {
    GetEmail()
}
```

正则表达式

- 文档：https://studygolang.com/pkgdoc
- API
  - re := regexp.MustCompile(reStr)，传入正则表达式，得到正则表达式对象
  - ret := re.FindAllStringSubmatch(srcStr,-1)：用正则对象，获取页面页面，srcStr是页面内容，-1代表取全部
- 爬邮箱
- 方法抽取
- 爬超链接
- 爬手机号
  - http://www.zhaohaowang.com/ 如果连接失效了自己找一个有手机号的就好了
- 爬身份证号
  - http://henan.qq.com/a/20171107/069413.htm 如果连接失效了自己找一个就好了
- 爬图片链接

```
package main
```

```go
import (
	"fmt"
	"io/ioutil"
	"net/http"
	"regexp"
)

var (
	// w代表大小写字母+数字+下划线
	reEmail = `\w+@\w+\.\w+`
	// s?有或者没有s
	// +代表出1次或多次
	//\s\S各种字符
	// +?代表贪婪模式
	reLinke  = `href="(https?://[\s\S]+?)"`
	rePhone  = `1[3456789]\d\s?\d{4}\s?\d{4}`
	reIdcard = `[123456789]\d{5}((19\d{2})|(20[01]\d))((0[1-9])|(1[012]))((0[1-9
])|([12]\d)|(3[01]))\d{3}[\dXx]`
	reImg    = `https?://[^"]+?(\.((jpg)|(png)|(jpeg)|(gif)|(bmp)))`
)

// 处理异常
func HandleError(err error, why string) {
	if err != nil {
		fmt.Println(why, err)
	}
}

func GetEmail2(url string) {
	pageStr := GetPageStr(url)
	re := regexp.MustCompile(reEmail)
	results := re.FindAllStringSubmatch(pageStr, -1)
	for _, result := range results {
		fmt.Println(result)
	}
}

// 抽取根据url获取内容
func GetPageStr(url string) (pageStr string) {
	resp, err := http.Get(url)
	HandleError(err, "http.Get url")
	defer resp.Body.Close()
	// 2.读取页面内容
	pageBytes, err := ioutil.ReadAll(resp.Body)
	HandleError(err, "ioutil.ReadAll")
	// 字节转字符串
```

```
    pageStr = string(pageBytes)
    return pageStr
}

func main() {
    // 2.抽取的爬邮箱
    // GetEmail2("https://tieba.baidu.com/p/6051076813?red_tag=1573533731")
    // 3.爬链接
    //GetLink("http://www.baidu.com/s?wd=%E8%B4%B4%E5%90%A7%20%E7%95%99%E4%B8%8B%E9%82%AE%E7%AE%B1&rsv_spt=1&rsv_iqid=0x98ace53400003985&issp=1&f=8&rsv_bp=1&rsv_idx=2&ie=utf-8&tn=baiduhome_pg&rsv_enter=1&rsv_dl=ib&rsv_sug2=0&inputT=5197&rsv_sug4=6345")
    // 4.爬手机号
    //GetPhone("https://www.zhaohaowang.com/")
    // 5.爬身份证号
    //GetIdCard("https://henan.qq.com/a/20171107/069413.htm")
    // 6.爬图片
    // GetImg("http://image.baidu.com/search/index?tn=baiduimage&ps=1&ct=201326592&lm=-1&cl=2&nc=1&ie=utf-8&word=%E7%BE%8E%E5%A5%B3")
}

func GetIdCard(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reIdcard)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result)
    }
}

// 爬链接
func GetLink(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reLinke)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result[1])
    }
}

//爬手机号
func GetPhone(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(rePhone)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
```

```
        fmt.Println(result)
    }
}

func GetImg(url string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reImg)
    results := re.FindAllStringSubmatch(pageStr, -1)
    for _, result := range results {
        fmt.Println(result[0])
    }
}
```

并发爬取美图

下面的两个是即将要爬的网站，如果网址失效自己换一个就好了

- https://www.bizhizu.cn/shouji/tag-%E5%8F%AF%E7%88%B1/1.html

```
package main

import (
    "fmt"
    "io/ioutil"
    "net/http"
    "regexp"
    "strconv"
    "strings"
    "sync"
    "time"
)

func HandleError(err error, why string) {
    if err != nil {
        fmt.Println(why, err)
    }
}

// 下载图片，传入的是图片叫什么
func DownloadFile(url string, filename string) (ok bool) {
    resp, err := http.Get(url)
    HandleError(err, "http.get.url")
    defer resp.Body.Close()
    bytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "resp.body")
```

```
    filename = "E:/topgoer.com/src/github.com/student/3.0/img/" + filename
    // 写出数据
    err = ioutil.WriteFile(filename, bytes, 0666)
    if err != nil {
        return false
    } else {
        return true
    }
}

// 并发爬思路:
// 1.初始化数据管道
// 2.爬虫写出:26个协程向管道中添加图片链接
// 3.任务统计协程:检查26个任务是否都完成,完成则关闭数据管道
// 4.下载协程:从管道里读取链接并下载

var (
    // 存放图片链接的数据管道
    chanImageUrls chan string
    waitGroup     sync.WaitGroup
    // 用于监控协程
    chanTask chan string
    reImg    = `https?://[^"]+?(\.((jpg)|(png)|(jpeg)|(gif)|(bmp)))`
)

func main() {
    // myTest()
    // DownloadFile("http://i1.shaodiyejin.com/uploads/tu/201909/10242/e5794daf5
8_4.jpg", "1.jpg")

    // 1.初始化管道
    chanImageUrls = make(chan string, 1000000)
    chanTask = make(chan string, 26)
    // 2.爬虫协程
    for i := 1; i < 27; i++ {
        waitGroup.Add(1)
        go getImgUrls("https://www.bizhizu.cn/shouji/tag-%E5%8F%AF%E7%88%B1/" +
 strconv.Itoa(i) + ".html")
    }
    // 3.任务统计协程,统计26个任务是否都完成,完成则关闭管道
    waitGroup.Add(1)
    go CheckOK()
    // 4.下载协程:从管道中读取链接并下载
    for i := 0; i < 5; i++ {
        waitGroup.Add(1)
        go DownloadImg()
```

```go
	}
	waitGroup.Wait()
}

// 下载图片
func DownloadImg() {
	for url := range chanImageUrls {
		filename := GetFilenameFromUrl(url)
		ok := DownloadFile(url, filename)
		if ok {
			fmt.Printf("%s 下载成功\n", filename)
		} else {
			fmt.Printf("%s 下载失败\n", filename)
		}
	}
	waitGroup.Done()
}

// 截取url名字
func GetFilenameFromUrl(url string) (filename string) {
	// 返回最后一个/的位置
	lastIndex := strings.LastIndex(url, "/")
	// 切出来
	filename = url[lastIndex+1:]
	// 时间戳解决重名
	timePrefix := strconv.Itoa(int(time.Now().UnixNano()))
	filename = timePrefix + "_" + filename
	return
}

// 任务统计协程
func CheckOK() {
	var count int
	for {
		url := <-chanTask
		fmt.Printf("%s 完成了爬取任务\n", url)
		count++
		if count == 26 {
			close(chanImageUrls)
			break
		}
	}
	waitGroup.Done()
}

// 爬图片链接到管道
```

```
// url是传的整页链接
func getImgUrls(url string) {
    urls := getImgs(url)
    // 遍历切片里所有链接，存入数据管道
    for _, url := range urls {
        chanImageUrls <- url
    }
    // 标识当前协程完成
    // 每完成一个任务，写一条数据
    // 用于监控协程知道已经完成了几个任务
    chanTask <- url
    waitGroup.Done()
}

// 获取当前页图片链接
func getImgs(url string) (urls []string) {
    pageStr := GetPageStr(url)
    re := regexp.MustCompile(reImg)
    results := re.FindAllStringSubmatch(pageStr, -1)
    fmt.Printf("共找到%d条结果\n", len(results))
    for _, result := range results {
        url := result[0]
        urls = append(urls, url)
    }
    return
}

// 抽取根据url获取内容
func GetPageStr(url string) (pageStr string) {
    resp, err := http.Get(url)
    HandleError(err, "http.Get url")
    defer resp.Body.Close()
    // 2.读取页面内容
    pageBytes, err := ioutil.ReadAll(resp.Body)
    HandleError(err, "ioutil.ReadAll")
    // 字节转字符串
    pageStr = string(pageBytes)
    return pageStr
}
```

千万数据过滤

需求

读入数据

数据清洗

省份划分

需求

某公司希望对自己公司的开放数据进行分类，过滤掉不完善的信息！信息的内容如下格式大概有1000万条1.27G

为了隐私安全做了图片马赛克下面是数据样式（主要学技术）

```
[illegible],3101151[illegible]02231,M,19881130,-,-,1391[illegible]524,-,-,www.venus394@sohu.com,
[illegible]宸寰,1101[illegible]7191116,M,19710719,-,-,13[illegible]253,-,-,13501165253@139.com,0
[illegible]国,310[illegible]2140434,M,19560214,-,20033[illegible]5235470,-,-,shen0214@tom.com,
[illegible]杰,460[illegible]7191026,M,19820719,-,5702[illegible]1352064,0898-66820283,0898-66820270,hecj2698@sina.com,0
[illegible]鹏,1201[illegible]52039,M,19700126,-,-,1392[illegible]7503,-,-,hp7454@sina.com,
[illegible]彦彬,310[illegible]1240433,M,19801124,-,-,13[illegible]321534,-,-,bg1omd@gmail.com,
[illegible]磊,650202[illegible]2009X,M,19730302,-,-,1389[illegible]31,-,-,xjlvlei@163.com,
[illegible]毅,620202[illegible]50632,M,19840606,-,214026,1[illegible]103556,-,-,yjcylove@yahoo.com.cn,0
[illegible]锐,51112[illegible]39058,M,19811113,上海,-,13[illegible]142265,-,-,undeadpoll@163.com,0
[illegible]思磊,310[illegible]7275027,F,19850727,长宁区
[illegible]炜,43010[illegible]60513,M,19761116,-,-,1356[illegible]126,-,-,weihe.sh@gmail.com,
[illegible]轩,32110[illegible]91043,F,19801129,-,-,1351[illegible]-,-,jfy333@163.com,
[illegible]李超,330[illegible]91060042,F,19810106,-,-,1386[illegible]ovespice@126.com,
[illegible],37100[illegible]214019,M,19850621,山东青岛[illegible]17,0532-82241070,-,jiangmiao500@126.com,
[illegible]安,12010[illegible]5193327,M,19790414,-,-,134[illegible],-,-,anan2631@163.com,0
[illegible]晓伟,320[illegible]10831019X,M,19810831,南京市鼓楼区国庐路19号405室,-,1[illegible]80516,1[illegible]80516,
[illegible]玮,142636[illegible]08020034,M,19820802,山西省太原市杏花岭区南华门东二条7号[illegible]计署太原[illegible],
```

读入数据

```
package main

import (
	"bufio"
	"fmt"
	"io"
	"io/ioutil"
	"os"
	"strings"

	//防止中文乱码的一个库
	"github.com/axgle/mahonia"
)

// 读取数据
func read() {
	contentBytes, err := ioutil.ReadFile("./kaifang.txt")
	if err != nil {
		fmt.Println("读入失败，", err)
	}
	contentStr := string(contentBytes)
	// 逐行打印，并处理乱码
	lineStrs := strings.Split(contentStr, "\n\r")
	for _, lineStr := range lineStrs {
		//fmt.Println(lineStr)
		newStr := ConvertEncoding(lineStr, "GBK")
		fmt.Println(newStr)
	}
}

// 方法2：缓冲读取（如果文件比较大的情况下建议是缓冲读取）
func read2() {
	file, _ := os.Open("./kaifang.txt")
	defer file.Close()
	// 建缓冲区
	reader := bufio.NewReader(file)
	for {
		lineBytes, _, err := reader.ReadLine()
		if err == io.EOF {
			break
		}
		gbkStr := string(lineBytes)
```

```go
        utfStr := ConvertEncoding(gbkStr, "GBK")
        fmt.Println(utfStr)
    }
}

// 处理乱码
// 参数1：处理的数据
// 参数2：数据目前的编码
// 参数3：返回的正常数据
func ConvertEncoding(srcStr string, encoding string) (dstStr string) {
    // 创建编码处理器
    enc := mahonia.NewDecoder(encoding)
    // 编码器处理字符串为utf8的字符串
    utfStr := enc.ConvertString(srcStr)
    dstStr = utfStr
    return
}

func main() {
    read2()
}
```

数据清洗

读取清洗的效率还是挺快的

```
package main

import (
	"bufio"
	"fmt"
	"io"
	"os"
	"strings"

	"github.com/axgle/mahonia"
)

func main() {
	// 1.打开文件
	file, _ := os.Open("./kaifang.txt")
	defer file.Close()
	// 创建优质文件
	goodFile, _ := os.OpenFile("./kaifang_good.txt", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
	defer goodFile.Close()
	// 创建劣质文件
	badFile, _ := os.OpenFile("./kaifang_bad.txt", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
	defer badFile.Close()
	// 2.缓冲读取
	reader := bufio.NewReader(file)
	for {
		lineBytes, _, err := reader.ReadLine()
		if err == io.EOF {
			break
		}
		gbkStr := string(lineBytes)
		lineStr := ConvertEncoding(gbkStr, "GBK")
		// 3.根据行数据,取身份证
		fields := strings.Split(lineStr, ",")
		// 判断长度大于等于2,下标1的位置长度=18
		if len(fields) >= 2 && len(fields[1]) == 18 {
			goodFile.WriteString(lineStr + "\n")
			fmt.Println("Good:", lineStr)
		} else {
```

```
            badFile.WriteString(lineStr + "\n")
            fmt.Println("Bad:", lineStr)
        }
    }
}

func ConvertEncoding(srcStr string, encoding string) (dstStr string) {
    // 创建编码处理器
    enc := mahonia.NewDecoder(encoding)
    // 编码器处理字符串为utf8的字符串
    utfStr := enc.ConvertString(srcStr)
    dstStr = utfStr
    return
}
```

省份划分

根据身份证号的前2位把数据划分到不同的省份文件里面

```
package main

import (
	"bufio"
	"fmt"
	"io"
	"os"
	"strings"
	"sync"
)

// 按照34个省划分数据
// 1.创建34个省份，创建34个数据管道
// 2.读优质数据，写入对应省份管道
// 3.把省份管道写道对应文件，开34个协程

// 抽象出一个省份对象
type Province struct {
	// Id 身份证前2位
	Id string
	// 省份名
	Name string
	// 该省对应的文件，例如 北京.txt
	File *os.File
	// 本省文件的数据管道
	chanData chan string
}

// 声明等待组
var wg sync.WaitGroup

func main() {
	// 声明个map，存放所有省市的
	pMap := make(map[string]*Province)
	ps := []string{"北京市11", "天津市12", "河北省13",
		"山西省14", "内蒙古自治区15", "辽宁省21", "吉林省22",
		"黑龙江省23", "上海市31", "江苏省32", "浙江省33", "安徽省34",
		"福建省35", "江西省36", "山东省37", "河南省41", "湖北省42",
		"湖南省43", "广东省44", "广西壮族自治区45", "海南省46",
		"重庆市50", "四川省51", "贵州省52", "云南省53", "西藏自治区54",
```

```
        "陕西省61", "甘肃省62", "青海省63", "宁夏回族自治区64", "新疆维吾尔自治区65",
        "香港特别行政区81", "澳门特别行政区82", "台湾省83"}
    // 遍历所有省市，创建实例，省份管道创建
    for _, p := range ps {
        name := p[:len(p)-2]
        id := p[len(p)-2:]
        // 创建对象
        province := Province{Id: id, Name: name}
        // 添加进map
        pMap[id] = &province
        // 为当前省份开一个文件
        file, _ := os.OpenFile("./"+province.Name+".txt", os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0666)
        province.File = file
        defer file.Close()
        // 创建当前省份管道
        province.chanData = make(chan string, 1024)
        fmt.Println(name, "管道已创建")
    }
    // 遍历34个省份，开34个对应文件写数据
    for _, province := range pMap {
        wg.Add(1)
        // 写入数据，这里是map中的地址
        go writeFile(province)
    }
    // 读优质文件，写入对应的省份管道
    file, _ := os.Open("./kaifang_good.txt")
    defer file.Close()
    // 缓冲读取
    reader := bufio.NewReader(file)
    // 逐行读
    for {
        lineBytes, _, err := reader.ReadLine()
        if err == io.EOF {
            for _, province := range pMap {
                close(province.chanData)
                fmt.Println(province.Name, "管道已经关闭")
            }
            break
        }
        // 转str，转utf
        lineStr := string(lineBytes)
        // 逗号切分
        fieldsSlice := strings.Split(lineStr, ",")
        id := fieldsSlice[1][0:2]
```

```
        // 对号入座，写入对应管道
        if provice, ok := pMap[id]; ok {
            provice.chanData <- (lineStr + "\n")
        } else {
            fmt.Println("未知的省", id)
        }
    }
    wg.Wait()
}

// 向文件写数据
func writeFile(province *Province) {
    for lineStr := range province.chanData {
        province.File.WriteString(lineStr)
        fmt.Print(province.Name, "写入", lineStr)
    }
    wg.Done()
}
```

go-admin

前言

GoAdmin 可以帮助你的golang应用快速实现数据可视化，搭建一个数据管理平台。

demo: https://demo.go-admin.cn
账号：admin 密码：admin

代码地址： https://github.com/GoAdminGroup/go-admin

特征

- □ **高生产效率**: 10分钟内做一个好看的管理后台
- □ **主题**: 默认为adminlte，更多好看的主题正在制作中，欢迎给我们留言
- □ **插件化**: 提供插件使用，真正实现一个插件解决不了问题，那就两个
- □ **认证**: 开箱即用的rbac认证系统
- ⚙ **框架支持**: 支持大部分框架接入，让你更容易去上手和扩展

使用

通过以下三步运行：

第一步：导入 sql

mysql
postgresql
sqlite

第二步：创建 main.go

```
package main

import (
    "github.com/gin-gonic/gin"
    _ "github.com/GoAdminGroup/go-admin/adapter/gin"
    _ "github.com/GoAdminGroup/go-admin/modules/db/drivers/mysql"
    "github.com/GoAdminGroup/go-admin/engine"
    "github.com/GoAdminGroup/go-admin/plugins/admin"
    "github.com/GoAdminGroup/themes/adminlte"
    "github.com/GoAdminGroup/go-admin/modules/config"
    "github.com/GoAdminGroup/go-admin/template"
        "github.com/GoAdminGroup/go-admin/template/chartjs"
        "github.com/GoAdminGroup/go-admin/template/types"
    "github.com/GoAdminGroup/go-admin/examples/datamodel"
    "github.com/GoAdminGroup/go-admin/modules/language"
)

func main() {
    r := gin.Default()

    eng := engine.Default()

    // global config
    cfg := config.Config{
        Databases: config.DatabaseList{
            "default": {
                Host:       "127.0.0.1",
                Port:       "3306",
                User:       "root",
                Pwd:        "root",
                Name:       "godmin",
                MaxIdleCon: 50,
                MaxOpenCon: 150,
                Driver:     "mysql",
            },
        },
        UrlPrefix: "admin",
```

```
        // STORE 必须设置且保证有写权限，否则增加不了新的管理员用户
        Store: config.Store{
            Path:   "./uploads",
            Prefix: "uploads",
        },
        Language: language.CN,
        // 开发模式
                Debug: true,
                // 日志文件位置，需为绝对路径
                InfoLogPath: "/var/logs/info.log",
                AccessLogPath: "/var/logs/access.log",
                ErrorLogPath: "/var/logs/error.log",
                ColorScheme: adminlte.ColorschemeSkinBlack,
    }

        // Generators： 详见 https://github.com/GoAdminGroup/go-admin/blob/maste
r/examples/datamodel/tables.go
    adminPlugin := admin.NewAdmin(datamodel.Generators)

    // 增加 chartjs 组件
    template.AddComp(chartjs.NewChart())

    // 增加 generator, 第一个参数是对应的访问路由前缀
    // 例子:
    //
    // "user" => http://localhost:9033/admin/info/user
    //
    // adminPlugin.AddGenerator("user", datamodel.GetUserTable)

    // 自定义首页

            r.GET("/admin", func(ctx *gin.Context) {
                engine.Content(ctx, func(ctx interface{}) (types.Panel, error) {
                    return datamodel.GetContent()
                })
            })

    _ = eng.AddConfig(cfg).AddPlugins(adminPlugin).Use(r)

    _ = r.Run(":9033")
}
```

其他框架的例子: https://github.com/GoAdminGroup/go-admin/tree/master/examples

第三步：运行

```
GO111MODULE=on go run main.go
```

访问：http://localhost:9033/admin

更多细节详见 文档说明

这里一个超级简单上手的例子

go-vue-admin

基本介绍

项目介绍

在线预览

> Gin-vue-admin是一个基于vue和gin开发的全栈前后端分离的后台管理系统，集成jwt鉴权，动态路由，动态菜单，casbin鉴权，表单生成器，代码生成器等功能，提供多种示例文件，让您把更多时间专注在业务开发上。

贡献指南

Hi! 首先感谢你使用 gin-vue-admin。

Gin-vue-admin 是一套为后台管理平台准备的一整套前后端分离架构式的开源框架，旨在快速搭建后台管理系统。

Gin-vue-admin 的成长离不开大家的支持，如果你愿意为 gin-vue-admin 贡献代码或提供建议，请阅读以下内容。

1.2.1 Issue 规范

- issue 仅用于提交 Bug 或 Feature 以及设计相关的内容，其它内容可能会被直接关闭。如果你在使用时产生了疑问，请到 Slack 或 Gitter 里咨询。
- 在提交 issue 之前，请搜索相关内容是否已被提出。

1.2.2 Pull Request 规范

- 请先 fork 一份到自己的项目下，不要直接在仓库下建分支。
- commit 信息要以 `[文件名]: 描述信息` 的形式填写，例如 `README.md: fix xxx bug` 。
- 确保 PR 是提交到 `develop` 分支，而不是 `master` 分支。
- 如果是修复 bug，请在 PR 中给出描述信息。

- 合并代码需要两名维护人员参与：一人进行 review 后 approve，另一人再次 review，通过后即可合并。

版本列表

- master: 2.0, 用于生产环境
- develop: 2.0, 用于测试环境
- gin-vue-admin_v2.0_dev （v2.0 不再兼容 v1.0）
- gin-vue-admin_v1.0_stable （v1.0 稳定版，会持续更新和维护）
- gin-vue-admin_v1.0_dev （v1.0 稳定版，会持续更新和维护）

使用说明

```
- node版本 > v8.6.0
- golang版本 >= v1.11
- IDE推荐：Goland
- 各位在clone项目以后，把db文件导入自己创建的库后，最好前往七牛云申请自己的空间地址。
- 替换掉项目中的七牛云公钥，私钥，仓名和默认url地址，以免发生测试文件数据错乱
```

web端

```
# clone the project
git clone https://github.com/piexlmax/gin-vue-admin.git

# enter the project directory
cd web

# install dependency
npm install

# develop
npm run serve
```

server端

```
# 使用 go.mod

# 安装go依赖包
go list (go mod tidy)

# 编译
go build
```

swagger自动化API文档

2.3.1 安装 swagger

（1）可以翻墙

```
go get -u github.com/swaggo/swag/cmd/swag
```

（2）无法翻墙

由于国内没法安装 go.org/x 包下面的东西，需要先安装 `gopm`

```
# 下载gopm包
go get -v -u github.com/gpmgo/gopm

# 执行
gopm get -g -v github.com/swaggo/swag/cmd/swag

# 到GOPATH的/src/github.com/swaggo/swag/cmd/swag路径下执行
go install
```

2.3.2 生成API文档

```
cd server
swag init
```

执行上面的命令后，server目录下会出现docs文件夹，登录 http://localhost:8888/swagger/index.html，即可查看swagger文档

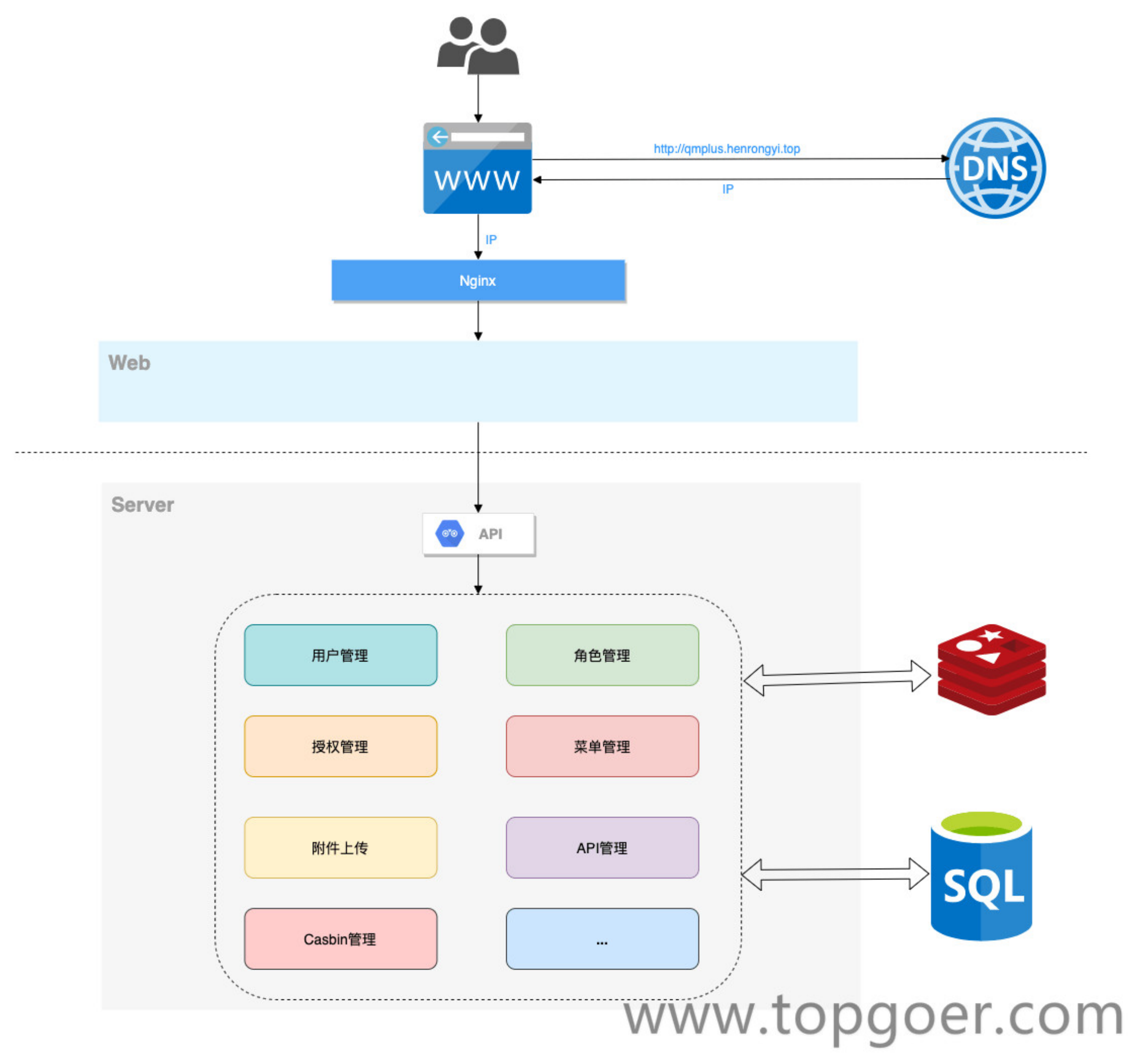
技术选型

- 前端：用基于 `vue` 的 `Element-UI` 构建基础页面。
- 后端：用 `Gin` 快速搭建基础restful风格API， `Gin` 是一个go语言编写的Web框架。

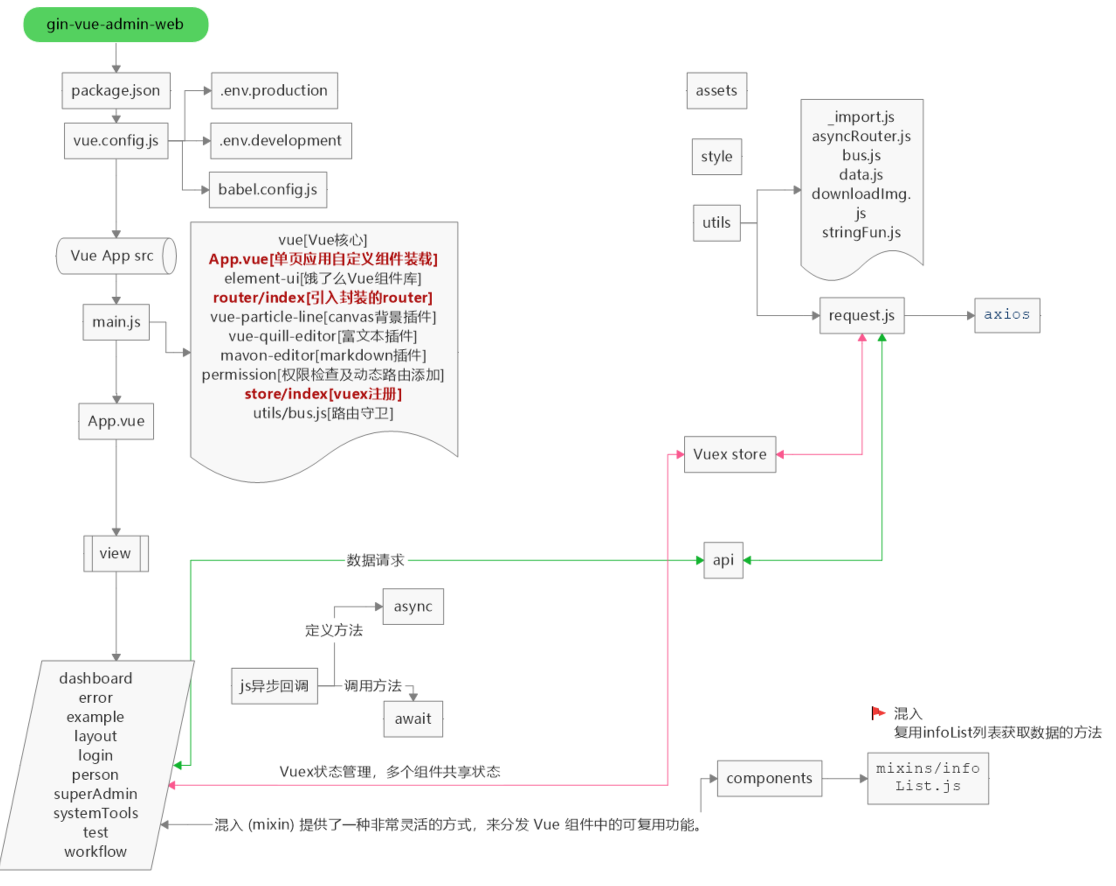
- 数据库：采用`MySql`(5.6.44)版本，使用`gorm`实现对数据库的基本操作,已添加对sqlite数据库的支持。
- 缓存：使用`Redis`实现记录当前活跃用户的`jwt`令牌并实现多点登录限制。
- API文档：使用`Swagger`构建自动化文档。
- 配置文件：使用`fsnotify`和`viper`实现`yaml`格式的配置文件。
- 日志：使用`go-logging`实现日志记录。

项目架构

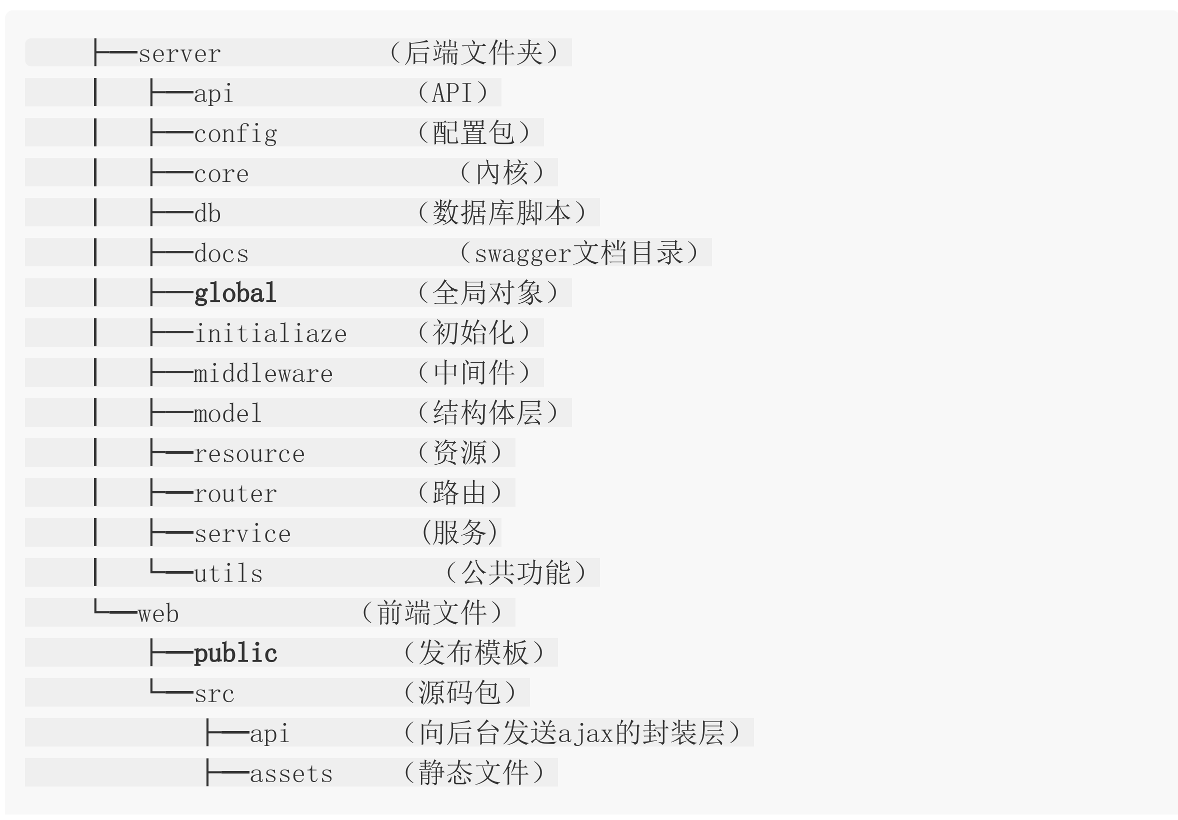
系统架构图



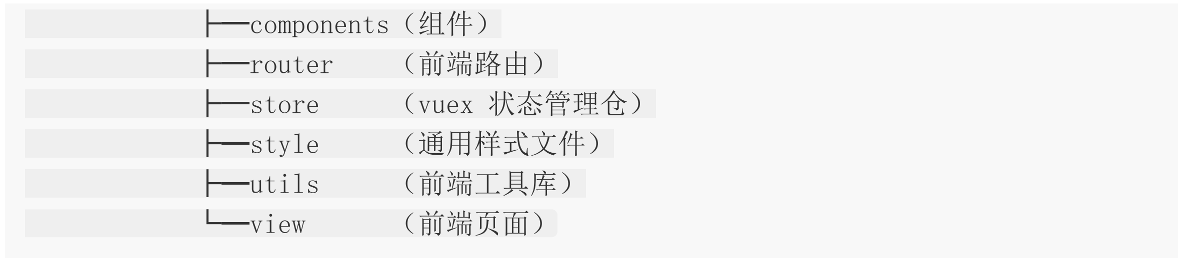
前端详细设计图 （提供者:baobeisuper）



目录结构



```
    ├──server                  （后端文件夹）
    │   ├──api                  （API）
    │   ├──config               （配置包）
    │   ├──core                    （内核）
    │   ├──db                   （数据库脚本）
    │   ├──docs                    （swagger文档目录）
    │   ├──global               （全局对象）
    │   ├──initializaze         （初始化）
    │   ├──middleware           （中间件）
    │   ├──model                （结构体层）
    │   ├──resource             （资源）
    │   ├──router               （路由）
    │   ├──service              （服务）
    │   └──utils                  （公共功能）
    └──web                   （前端文件）
        ├──public              （发布模板）
        └──src                 （源码包）
            ├──api             （向后台发送ajax的封装层）
            ├──assets          （静态文件）
```



```
            ├─components (组件)
            ├─router     (前端路由)
            ├─store      (vuex 状态管理仓)
            ├─style      (通用样式文件)
            ├─utils      (前端工具库)
            └─view       (前端页面)
```

主要功能

- 权限管理：基于 `jwt` 和 `casbin` 实现的权限管理
- 文件上传下载：实现基于七牛云的文件上传操作（为了方便大家测试，我公开了自己的七牛测试号的各种重要token，恳请大家不要乱传东西）
- 分页封装：前端使用mixins封装分页，分页方法调用mixins即可
- 用户管理：系统管理员分配用户角色和角色权限。
- 角色管理：创建权限控制的主要对象，可以给角色分配不同api权限和菜单权限。
- 菜单管理：实现用户动态菜单配置，实现不同角色不同菜单。
- api管理：不同用户可调用的api接口的权限不同。
- 配置管理：配置文件可前台修改（测试环境不开放此功能）。
- 富文本编辑器：MarkDown编辑器功能嵌入。
- 条件搜索：增加条件搜索示例。
- restful示例：可以参考用户管理模块中的示例API。

  ```
  前端文件参考: src\view\superAdmin\api\api.vue
  后台文件参考: model\dnModel\api.go
  ```

- 多点登录限制：需要在 `config.yaml` 中把 `system` 中的 `useMultipoint` 修改为true(需要自行配置Redis和Config中的Redis参数，测试阶段，有bug请及时反馈)。
- 分片长传：提供文件分片上传和大文件分片上传功能示例。
- 表单生成器：表单生成器借助 @form-generator。
- 代码生成器：后台基础逻辑以及简单curd的代码生成器。

计划任务

- 导入，导出Excel
- Echart图表支持
- 工作流，任务交接功能开发

- 单独前端使用模式以及数据模拟

go-admin

基于Gin + Vue + Element UI的前后端分离权限管理系统

系统初始化极度简单，只需要配置文件中，修改数据库连接，系统启动后会自动初始化数据库信息以及必须的基础数据

在线文档

视频教程

特性

- 遵循 RESTful API 设计规范
- 基于 GIN WEB API 框架，提供了丰富的中间件支持（用户认证、跨域、访问日志、追踪ID等）
- 基于Casbin的 RBAC 访问控制模型
- JWT 认证
- 支持 Swagger 文档(基于swaggo)
- 基于 GORM 的数据库存储，可扩展多种类型数据库
- 配置文件简单的模型映射，快速能够得到想要的配置
- 代码生成工具
- 表单构建工具
- 多命令模式
- TODO: 单元测试

内置

1. 用户管理：用户是系统操作者，该功能主要完成系统用户配置。
2. 部门管理：配置系统组织机构（公司、部门、小组），树结构展现支持数据权限。

3. 岗位管理：配置系统用户所属担任职务。
4. 菜单管理：配置系统菜单，操作权限，按钮权限标识等。
5. 角色管理：角色菜单权限分配、设置角色按机构进行数据范围权限划分。
6. 字典管理：对系统中经常使用的一些较为固定的数据进行维护。
7. 参数管理：对系统动态配置常用参数。
8. 操作日志：系统正常操作日志记录和查询；系统异常信息日志记录和查询。
9. 登录日志：系统登录日志记录查询包含登录异常。
10. 系统接口：根据业务代码自动生成相关的api接口文档。
11. 代码生成：根据数据表结构生成对应的增删改查相对应业务，全部可视化编程，基本业务可以0代码实现。
12. 表单构建：自定义页面样式，拖拉拽实现页面布局。
13. 服务监控：查看一些服务器的基本信息。

准备工作

你需要在本地安装 [go] [gin] node 和 git

同时配套了系列教程包含视频和文档，如何从下载完成到熟练使用，强烈建议大家先看完这些教程再来实践本项目！！！

轻松实现go-admin写出第一个应用 - 文档教程

步骤一 - 基础内容介绍

步骤二 - 实际应用 - 编写增删改查

手把手教你从入门到放弃 - 视频教程

如何启动go-admin

使用生成工具轻松实现业务

go-admin菜单的配置说明

多命令启动方式讲解以及IDE配置

如有问题请先看上述使用文档和文章，若不能满足，欢迎 issue 和 pr ，视频教程和文档持续更新中

本地开发

首次启动说明

```
# 获取代码
git clone https://github.com/wenjianzhang/go-admin.git

# 进入工作路径
cd ./go-admin

# 编译项目
go build

# 修改配置
# 文件路径  go-admin/config/settings.yml
vi ./config/setting.yml

# 1. 配置文件中修改数据库信息
# 注意: settings.database 下对应的配置数据
# 2. 确认log路径
```

初始化数据库，以及服务启动

```
# 首次配置需要初始化数据库资源信息
./go-admin init -c config/settings.yml -m dev


# 启动项目，也可以用IDE进行调试
./go-admin server -c config/settings.yml -p 8000 -m dev
```

文档生成

```
swag init

# 如果没有swag命令 go get安装一下即可
go get -u github.com/swaggo/swag/cmd/swag
```

交叉编译

```
env GOOS=windows GOARCH=amd64 go build main.go

# or

env GOOS=linux GOARCH=amd64 go build main.go
```

🎉 在线体验

> admin / 123456

演示地址：http://www.zhangwj.com

log

Logger

Zap Logger

日志切割归档

Logger

介绍

在许多Go语言项目中，我们需要一个好的日志记录器能够提供下面这些功能：

- 能够将事件记录到文件中，而不是应用程序控制台。
- 日志切割-能够根据文件大小、时间或间隔等来切割日志文件。
- 支持不同的日志级别。例如INFO，DEBUG，ERROR等。
- 能够打印基本信息，如调用文件/函数名和行号，日志时间等。

默认的Go Logger

在介绍Uber-go的zap包之前，让我们先看看Go语言提供的基本日志功能。Go语言提供的默认日志包是 https://golang.org/pkg/log/ 。

实现Go Logger

实现一个Go语言中的日志记录器非常简单——创建一个新的日志文件，然后设置它为日志的输出位置。

```
package main

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

func main() {
    //创建输出日志文件
    logFile, err := os.Create("./" + time.Now().Format("20060102") + ".txt")
    if err != nil {
        fmt.Println(err)
    }
    //创建一个Logger
    //参数1：日志写入目的地
    //参数2：每条日志的前缀
    //参数3：日志属性
    loger := log.New(logFile, "test_", log.Ldate|log.Ltime|log.Lshortfile)
```

```
    //Flags返回Logger的输出选项
    fmt.Println(loger.Flags())

    //SetFlags设置输出选项
    loger.SetFlags(log.Ldate | log.Ltime | log.Lshortfile)

    //返回输出前缀
    fmt.Println(loger.Prefix())

    //设置输出前缀
    loger.SetPrefix("test_")

    //输出一条日志
    loger.Output(2, "打印一条日志信息")

    //格式化输出日志
    // loger.Printf("第%d行 内容:%s", 11, "我是错误")

    //等价于print();os.Exit(1);
    // loger.Fatal("我是错误")

    //等价于print();panic();
    // loger.Panic("我是错误33333333")

    //log的导出函数
    //导出函数基于std,std是标准错误输出
    //var std = New(os.Stderr, "", LstdFlags)

    //获取输出项
    fmt.Println(log.Flags())
    //获取前缀
    fmt.Printf(log.Prefix())
}
```

Go Logger的优势和劣势

优势

它最大的优点是使用非常简单。我们可以设置任何io.Writer作为日志记录输出并向其发送要写入的日志。

劣势

- 仅限基本的日志级别

    - 只有一个Print选项。不支持INFO/DEBUG等多个级别。
- 对于错误日志，它有Fatal和Panic
    - Fatal日志通过调用os.Exit(1)来结束程序
    - Panic日志在写入日志消息之后抛出一个panic
    - 但是它缺少一个ERROR日志级别，这个级别可以在不抛出panic或退出程序的情况下记录错误
- 缺乏日志格式化的能力——例如记录调用者的函数名和行号，格式化日期和时间格式。等等。
- 不提供日志切割的能力。

Zap Logger

Uber-go Zap

Zap是非常快的、结构化的，分日志级别的Go日志库。

为什么选择Uber-go zap

- 它同时提供了结构化日志记录和printf风格的日志记录
- 它非常的快

根据Uber-go Zap的文档，它的性能比类似的结构化日志包更好——也比标准库更快。 以下是Zap发布的基准测试信息

记录一条消息和10个字段:

Package	Time	Objects Allocated
⚡ zap	3131 ns/op	5 allocs/op
⚡ zap (sugared)	4173 ns/op	21 allocs/op
zerolog	16154 ns/op	90 allocs/op
lion	16341 ns/op	111 allocs/op
go-kit	17049 ns/op	126 allocs/op
logrus	23662 ns/op	142 allocs/op

www.topgoer.com

记录一个静态字符串，没有任何上下文或printf风格的模板：

Package	Time	Objects Allocated
⚡ zap	361 ns/op	0 allocs/op
⚡ zap (sugared)	534 ns/op	2 allocs/op
zerolog	323 ns/op	0 allocs/op
standard library	575 ns/op	2 allocs/op
go-kit	922 ns/op	13 allocs/op
lion	1413 ns/op	10 allocs/op
logrus	2291 ns/op	27 allocs/op

www.topgoer.com

安装

运行下面的命令安装zap

```
go get -u go.uber.org/zap
```

配置Zap Logger

Zap提供了两种类型的日志记录器—Sugared Logger和Logger。

在性能很好但不是很关键的上下文中，使用SugaredLogger。它比其他结构化日志记录包快4-10倍，并且支持结构化和printf风格的日志记录。

在每一微秒和每一次内存分配都很重要的上下文中，使用Logger。它甚至比SugaredLogger更快，内存分配次数也更少，但它只支持强类型的结构化日志记录。

Logger

- 通过调用zap.NewProduction()/zap.NewDevelopment()或者zap.Example()创建一个Logger。
- 上面的每一个函数都将创建一个logger。唯一的区别在于它将记录的信息不同。例如production logger默认记录调用函数信息、日期和时间等。
- 通过Logger调用Info/Error等。
- 默认情况下日志都会打印到应用程序的console界面。

```go
package main

import (
	"net/http"

	"go.uber.org/zap"
)

var logger *zap.Logger

func main() {
	InitLogger()
	defer logger.Sync()
	simpleHttpGet("www.5lmh.com")
	simpleHttpGet("http://www.google.com")
}

func InitLogger() {
	logger, _ = zap.NewProduction()
}

func simpleHttpGet(url string) {
	resp, err := http.Get(url)
	if err != nil {
		logger.Error(
			"Error fetching url..",
			zap.String("url", url),
			zap.Error(err))
	} else {
		logger.Info("Success..",
			zap.String("statusCode", resp.Status),
			zap.String("url", url))
		resp.Body.Close()
	}
}
```

在上面的代码中，我们首先创建了一个Logger，然后使用Info/ Error等Logger方法记录消息。

日志记录器方法的语法是这样的：

```go
	func (log *Logger) MethodXXX(msg string, fields ...Field)
```

其中MethodXXX是一个可变参数函数，可以是Info / Error/ Debug / Panic等。每个方法都接受一个消息字符串和任意数量的zapcore.Field场参数。

每个zapcore.Field其实就是一组键值对参数。

我们执行上面的代码会得到如下输出结果：

```
{"level":"error","ts":1573180648.858149,"caller":"ce2/main.go:25","msg":"Error fetching url..","url":"www.5lmh.com","error":"Get www.5lmh.com: unsupported protocol scheme \"\"","stacktrace":"main.simpleHttpGet\n\te:/goproject/src/github.com/student/log/ce2/main.go:25\nmain.main\n\te:/goproject/src/github.com/student/log/ce2/main.go:14\nruntime.main\n\tE:/go/src/runtime/proc.go:200"}

{"level":"error","ts":1573180669.9273467,"caller":"ce2/main.go:25","msg":"Error fetching url..","url":"http://www.google.com","error":"Get http://www.google.com: dial tcp 31.13.72.54:80: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.","stacktrace":"main.simpleHttpGet\n\te:/goproject/src/github.com/student/log/ce2/main.go:25\nmain.main\n\te:/goproject/src/github.com/student/log/ce2/main.go:15\nruntime.main\n\tE:/go/src/runtime/proc.go:200"}
```

Sugared Logger

现在让我们使用Sugared Logger来实现相同的功能。

- 大部分的实现基本都相同。
- 惟一的区别是，我们通过调用主logger的. Sugar()方法来获取一个SugaredLogger。
- 然后使用SugaredLogger以printf格式记录语句

下面是修改过后使用SugaredLogger代替Logger的代码：

```
package main

import (
    "net/http"

    "go.uber.org/zap"
)

var sugarLogger *zap.SugaredLogger

func main() {
    InitLogger()
    defer sugarLogger.Sync()
    simpleHttpGet("www.5lmh.com")
    simpleHttpGet("http://www.google.com")
```

```
}

func InitLogger() {
    logger, _ := zap.NewProduction()
    sugarLogger = logger.Sugar()
}

func simpleHttpGet(url string) {
    sugarLogger.Debugf("Trying to hit GET request for %s", url)
    resp, err := http.Get(url)
    if err != nil {
        sugarLogger.Errorf("Error fetching URL %s : Error = %s", url, err)
    } else {
        sugarLogger.Infof("Success! statusCode = %s for URL %s", resp.Status, ur
l)
        resp.Body.Close()
    }
}
```

```
{"level":"error","ts":1573180998.3522997,"caller":"ce3/main.go:27","msg":"Error
 fetching URL www.5lmh.com : Error = Get www.5lmh.com: unsupported protocol sche
me \"\"","stacktrace":"main.simpleHttpGet\n\te:/goproject/src/github.com/studen
t/log/ce3/main.go:27\nmain.main\n\te:/goproject/src/github.com/student/log/ce3/m
ain.go:14\nruntime.main\n\tE:/go/src/runtime/proc.go:200"}

{"level":"error","ts":1573181019.3677258,"caller":"ce3/main.go:27","msg":"Error
 fetching URL http://www.google.com : Error = Get http://www.google.com: dial tc
p 67.228.37.26:80: connectex: A connection attempt failed because the connected
 party did not properly respond after a period of time, or established connectio
n failed because connected host has failed to respond.","stacktrace":"main.simpl
eHttpGet\n\te:/goproject/src/github.com/student/log/ce3/main.go:27\nmain.main\n
\te:/goproject/src/github.com/student/log/ce3/main.go:15\nruntime.main\n\tE:/go/
src/runtime/proc.go:200"}
```

你应该注意到的了，到目前为止这两个logger都打印输出JSON结构格式。

在本博客的后面部分，我们将更详细地讨论SugaredLogger，并了解如何进一步配置它。

定制logger

将日志写入文件而不是终端

- 我们要做的第一个更改是把日志写入文件，而不是打印到应用程序控制台。

```
func New(core zapcore.Core, options ...Option) *Logger
```

zapcore.Core需要三个配置——Encoder，WriteSyncer，LogLevel。

1.Encoder:编码器(如何写入日志)。我们将使用开箱即用的NewJSONEncoder()，并使用预先设置的ProductionEncoderConfig()。

```
zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig())
```

2.WriterSyncer ：指定日志将写到哪里去。我们使用zapcore.AddSync()函数并且将打开的文件句柄传进去。

```
file, _ := os.Create("./test.log")
writeSyncer := zapcore.AddSync(file)
```

3.Log Level：哪种级别的日志将被写入。

我们将修改上述部分中的Logger代码，并重写InitLogger()方法。其余方法—main() /SimpleHttpGet()保持不变。

```
package main

import (
    "net/http"
    "os"

    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

var sugarLogger *zap.SugaredLogger

func main() {
    InitLogger()
    defer sugarLogger.Sync()
    simpleHttpGet("www.51mh.com")
    simpleHttpGet("http://www.google.com")
}

func InitLogger() {
    writeSyncer := getLogWriter()
    encoder := getEncoder()
    core := zapcore.NewCore(encoder, writeSyncer, zapcore.DebugLevel)
```

```
    logger := zap.New(core)
    sugarLogger = logger.Sugar()
}

func getEncoder() zapcore.Encoder {
    return zapcore.NewJSONEncoder(zap.NewProductionEncoderConfig())
}

func getLogWriter() zapcore.WriteSyncer {
    //如果想要追加写入可以查看我的博客文件操作那一章
    file, _ := os.Create("./test.log")
    return zapcore.AddSync(file)
}

func simpleHttpGet(url string) {
    sugarLogger.Debugf("Trying to hit GET request for %s", url)
    resp, err := http.Get(url)
    if err != nil {
        sugarLogger.Errorf("Error fetching URL %s : Error = %s", url, err)
    } else {
        sugarLogger.Infof("Success! statusCode = %s for URL %s", resp.Status, url)
        resp.Body.Close()
    }
}
```

当使用这些修改过的logger配置调用上述部分的main()函数时，以下输出将打印在文件——test.log中。

```
{"level":"debug","ts":1573181732.5292294,"msg":"Trying to hit GET request for www.51mh.com"}
{"level":"error","ts":1573181732.5292294,"msg":"Error fetching URL www.51mh.com : Error = Get www.51mh.com: unsupported protocol scheme \"\""}
{"level":"debug","ts":1573181732.5292294,"msg":"Trying to hit GET request for http://www.google.com"}
{"level":"error","ts":1573181753.564804,"msg":"Error fetching URL http://www.google.com : Error = Get http://www.google.com: dial tcp 66.220.149.32:80: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond."}
```

将JSON Encoder更改为普通的Log Encoder

现在，我们希望将编码器从JSON Encoder更改为普通Encoder。为此，我们需要将NewJSONEncoder()更改为NewConsoleEncoder()。

```
    return zapcore.NewConsoleEncoder(zap.NewProductionEncoderConfig())
```

当使用这些修改过的logger配置调用上述部分的main()函数时，以下输出将打印在文件——test.log中。

```
1.573181811861697e+09    debug    Trying to hit GET request for www.5lmh.com
1.5731818118626883e+09    error    Error fetching URL www.5lmh.com : Error = Get www.5lmh.com: unsupported protocol scheme ""
1.5731818118626883e+09    debug    Trying to hit GET request for http://www.google.com
1.5731818329012108e+09    error    Error fetching URL http://www.google.com : Error = Get http://www.google.com: dial tcp 216.58.200.228:80: connectex: A connection attempt failed because the connected party did not properly respond after a period of time, or established connection failed because connected host has failed to respond.
```

更改时间编码并添加调用者详细信息

鉴于我们对配置所做的更改，有下面两个问题：

- 时间是以非人类可读的方式展示，例如1.572161051846623e+09
- 调用方函数的详细信息没有显示在日志中

  我们要做的第一件事是覆盖默认的ProductionConfig()，并进行以下更改:

修改时间编码器

- 在日志文件中使用大写字母记录日志级别

```
func getEncoder() zapcore.Encoder {
    encoderConfig := zap.NewProductionEncoderConfig()
    encoderConfig.EncodeTime = zapcore.ISO8601TimeEncoder
    encoderConfig.EncodeLevel = zapcore.CapitalLevelEncoder
    return zapcore.NewConsoleEncoder(encoderConfig)
}
```

接下来，我们将修改zap logger代码，添加将调用函数信息记录到日志中的功能。为此，我们将在zap.New(..)函数中添加一个Option。

```
    logger := zap.New(core, zap.AddCaller())
```

日志切割归档

使用Lumberjack进行日志切割归档

这个日志程序中唯一缺少的就是日志切割归档功能。

> s="default">
>
> s="default">
>
> Zap本身不支持切割归档日志文件

为了添加日志切割归档功能，我们将使用第三方库Lumberjack来实现。

安装

执行下面的命令安装Lumberjack

```
    go get -u github.com/natefinch/lumberjack
```

zap logger中加入Lumberjack

要在zap中加入Lumberjack支持，我们需要修改WriteSyncer代码。我们将按照下面的代码修改getLogWriter()函数：

```
func getLogWriter() zapcore.WriteSyncer {
    lumberJackLogger := &lumberjack.Logger{
        Filename:   "./test.log",
        MaxSize:    10,
        MaxBackups: 5,
        MaxAge:     30,
        Compress:   false,
    }
    return zapcore.AddSync(lumberJackLogger)
}
```

Lumberjack Logger采用以下属性作为输入:

- Filename: 日志文件的位置
- MaxSize：在进行切割之前，日志文件的最大大小（以MB为单位）

- MaxBackups：保留旧文件的最大个数
- MaxAges：保留旧文件的最大天数
- Compress：是否压缩/归档旧文件

测试所有功能

最终，使用Zap/Lumberjack logger的完整示例代码如下：

```
package main

import (
    "net/http"

    "github.com/natefinch/lumberjack"
    "go.uber.org/zap"
    "go.uber.org/zap/zapcore"
)

var sugarLogger *zap.SugaredLogger

func main() {
    InitLogger()
    defer sugarLogger.Sync()
    simpleHttpGet("www.topgoer.com")
    simpleHttpGet("http://www.topgoer.com")
}

func InitLogger() {
    writeSyncer := getLogWriter()
    encoder := getEncoder()
    core := zapcore.NewCore(encoder, writeSyncer, zapcore.DebugLevel)

    logger := zap.New(core, zap.AddCaller())
    sugarLogger = logger.Sugar()
}

func getEncoder() zapcore.Encoder {
    encoderConfig := zap.NewProductionEncoderConfig()
    encoderConfig.EncodeTime = zapcore.ISO8601TimeEncoder
    encoderConfig.EncodeLevel = zapcore.CapitalLevelEncoder
    return zapcore.NewConsoleEncoder(encoderConfig)
}

func getLogWriter() zapcore.WriteSyncer {
```

```
    lumberJackLogger := &lumberjack.Logger{
        Filename:   "./test.log",
        MaxSize:    1,
        MaxBackups: 5,
        MaxAge:     30,
        Compress:   false,
    }
    return zapcore.AddSync(lumberJackLogger)
}

func simpleHttpGet(url string) {
    sugarLogger.Debugf("Trying to hit GET request for %s", url)
    resp, err := http.Get(url)
    if err != nil {
        sugarLogger.Errorf("Error fetching URL %s : Error = %s", url, err)
    } else {
        sugarLogger.Infof("Success! statusCode = %s for URL %s", resp.Status, ur
l)
        resp.Body.Close()
    }
}
```

聊天室小案例

webSocket是什么

- WebSocket是一种在单个TCP连接上进行全双工通信的协议
- WebSocket使得客户端和服务器之间的数据交换变得更加简单，允许服务端主动向客户端推送数据
- 在WebSocket API中，浏览器和服务器只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输
- 需要安装第三方包：
    - cmd中：go get -u -v github.com/gorilla/websocket

举个聊天室的小例子

在同一级目录下新建四个go文件 `connection.go|data.go|hub.go|server.go`

运行

```
    go run server.go hub.go data.go connection.go
```

运行之后执行local.html文件

server.go文件代码

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gorilla/mux"
)

func main() {
    router := mux.NewRouter()
    go h.run()
    router.HandleFunc("/ws", myws)
    if err := http.ListenAndServe("127.0.0.1:8080", router); err != nil {
        fmt.Println("err:", err)
```

```
    }
}
```

hub.go文件代码

```
package main

import "encoding/json"

var h = hub{
    c: make(map[*connection]bool),
    u: make(chan *connection),
    b: make(chan []byte),
    r: make(chan *connection),
}

type hub struct {
    c map[*connection]bool
    b chan []byte
    r chan *connection
    u chan *connection
}

func (h *hub) run() {
    for {
        select {
        case c := <-h.r:
            h.c[c] = true
            c.data.Ip = c.ws.RemoteAddr().String()
            c.data.Type = "handshake"
            c.data.UserList = user_list
            data_b, _ := json.Marshal(c.data)
            c.sc <- data_b
        case c := <-h.u:
            if _, ok := h.c[c]; ok {
                delete(h.c, c)
                close(c.sc)
            }
        case data := <-h.b:
            for c := range h.c {
                select {
                case c.sc <- data:
                default:
                    delete(h.c, c)
                    close(c.sc)
```

```
                }
            }
        }
    }
}
```

data.go文件代码

```
package main

type Data struct {
    Ip       string   `json:"ip"`
    User     string   `json:"user"`
    From     string   `json:"from"`
    Type     string   `json:"type"`
    Content  string   `json:"content"`
    UserList []string `json:"user_list"`
}
```

connection.go文件代码

```
package main

import (
    "encoding/json"
    "fmt"
    "net/http"

    "github.com/gorilla/websocket"
)

type connection struct {
    ws   *websocket.Conn
    sc   chan []byte
    data *Data
}

var wu = &websocket.Upgrader{ReadBufferSize: 512,
    WriteBufferSize: 512, CheckOrigin: func(r *http.Request) bool { return true
 }}

func myws(w http.ResponseWriter, r *http.Request) {
    ws, err := wu.Upgrade(w, r, nil)
    if err != nil {
```

```
        return
    }
    c := &connection{sc: make(chan []byte, 256), ws: ws, data: &Data{}}
    h.r <- c
    go c.writer()
    c.reader()
    defer func() {
        c.data.Type = "logout"
        user_list = del(user_list, c.data.User)
        c.data.UserList = user_list
        c.data.Content = c.data.User
        data_b, _ := json.Marshal(c.data)
        h.b <- data_b
        h.r <- c
    }()
}

func (c *connection) writer() {
    for message := range c.sc {
        c.ws.WriteMessage(websocket.TextMessage, message)
    }
    c.ws.Close()
}

var user_list = []string{}

func (c *connection) reader() {
    for {
        _, message, err := c.ws.ReadMessage()
        if err != nil {
            h.r <- c
            break
        }
        json.Unmarshal(message, &c.data)
        switch c.data.Type {
        case "login":
            c.data.User = c.data.Content
            c.data.From = c.data.User
            user_list = append(user_list, c.data.User)
            c.data.UserList = user_list
            data_b, _ := json.Marshal(c.data)
            h.b <- data_b
        case "user":
            c.data.Type = "user"
            data_b, _ := json.Marshal(c.data)
            h.b <- data_b
```

```
            case "logout":
                c.data.Type = "logout"
                user_list = del(user_list, c.data.User)
                data_b, _ := json.Marshal(c.data)
                h.b <- data_b
                h.r <- c
            default:
                fmt.Print("========default================")
            }
        }
}

func del(slice []string, user string) []string {
    count := len(slice)
    if count == 0 {
        return slice
    }
    if count == 1 && slice[0] == user {
        return []string{}
    }
    var n_slice = []string{}
    for i := range slice {
        if slice[i] == user && i == count {
            return slice[:count]
        } else if slice[i] == user {
            n_slice = append(slice[:i], slice[i+1:]...)
            break
        }
    }
    fmt.Println(n_slice)
    return n_slice
}
```

local.html文件代码

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
    <meta http-equiv="content-type" content="text/html;charset=utf-8">
    <style>
        p {
            text-align: left;
            padding-left: 20px;
        }
```

```
    </style>
</head>
<body>
<div style="width: 800px;height: 600px;margin: 30px auto;text-align: center">
    <h1>www.51mh.comy演示聊天室</h1>
    <div style="width: 800px;border: 1px solid gray;height: 300px;">
        <div style="width: 200px;height: 300px;float: left;text-align: left;">
            <p><span>当前在线:</span><span id="user_num">0</span></p>
            <div id="user_list" style="overflow: auto;">
            </div>
        </div>
        <div id="msg_list" style="width: 598px;border:  1px solid gray; height:
 300px;overflow: scroll;float: left;">
        </div>
    </div>
    <br>
    <textarea id="msg_box" rows="6" cols="50" onkeydown="confirm(event)"></texta
rea><br>
    <input type="button" value="发送" onclick="send()">
</div>
</body>
</html>
<script type="text/javascript">
    var uname = prompt('请输入用户名', 'user' + uuid(8, 16));
    var ws = new WebSocket("ws://127.0.0.1:8080/ws");
    ws.onopen = function () {
        var data = "系统消息：建立连接成功";
        listMsg(data);
    };
    ws.onmessage = function (e) {
        var msg = JSON.parse(e.data);
        var sender, user_name, name_list, change_type;
        switch (msg.type) {
            case 'system':
                sender = '系统消息: ';
                break;
            case 'user':
                sender = msg.from + ': ';
                break;
            case 'handshake':
                var user_info = {'type': 'login', 'content': uname};
                sendMsg(user_info);
                return;
            case 'login':
            case 'logout':
                user_name = msg.content;
```

```
                name_list = msg.user_list;
                change_type = msg.type;
                dealUser(user_name, change_type, name_list);
                return;
            }
            var data = sender + msg.content;
            listMsg(data);
        };
        ws.onerror = function () {
            var data = "系统消息 : 出错了,请退出重试.";
            listMsg(data);
        };
        function confirm(event) {
            var key_num = event.keyCode;
            if (13 == key_num) {
                send();
            } else {
                return false;
            }
        }
        function send() {
            var msg_box = document.getElementById("msg_box");
            var content = msg_box.value;
            var reg = new RegExp("\r\n", "g");
            content = content.replace(reg, "");
            var msg = {'content': content.trim(), 'type': 'user'};
            sendMsg(msg);
            msg_box.value = '';
        }
        function listMsg(data) {
            var msg_list = document.getElementById("msg_list");
            var msg = document.createElement("p");
            msg.innerHTML = data;
            msg_list.appendChild(msg);
            msg_list.scrollTop = msg_list.scrollHeight;
        }
        function dealUser(user_name, type, name_list) {
            var user_list = document.getElementById("user_list");
            var user_num = document.getElementById("user_num");
            while(user_list.hasChildNodes()) {
                user_list.removeChild(user_list.firstChild);
            }
            for (var index in name_list) {
                var user = document.createElement("p");
                user.innerHTML = name_list[index];
                user_list.appendChild(user);
```

```
        }
        user_num.innerHTML = name_list.length;
        user_list.scrollTop = user_list.scrollHeight;
        var change = type == 'login' ? '上线' : '下线';
        var data = '系统消息: ' + user_name + ' 已' + change;
        listMsg(data);
    }
    function sendMsg(msg) {
        var data = JSON.stringify(msg);
        ws.send(data);
    }
    function uuid(len, radix) {
        var chars = '0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvw
xyz'.split('');
        var uuid = [], i;
        radix = radix || chars.length;
        if (len) {
            for (i = 0; i < len; i++) uuid[i] = chars[0 | Math.random() * radi
x];
        } else {
            var r;
            uuid[8] = uuid[13] = uuid[18] = uuid[23] = '-';
            uuid[14] = '4';
            for (i = 0; i < 36; i++) {
                if (!uuid[i]) {
                    r = 0 | Math.random() * 16;
                    uuid[i] = chars[(i == 19) ? (r & 0x3) | 0x8 : r];
                }
            }
        }
        return uuid.join('');
    }
</script>
```

性能压测工具wrk

安装命令

```
    git clone https://github.com/wg/wrk
     ll
    cd wrk/
     ll
     make
     ll
    cp wrk /usr/local/sbin/
```

帮助 wrk

```
使用方法: wrk <选项> <被测HTTP服务的URL>
  Options:
    -c, --connections <N>  跟服务器建立并保持的TCP连接数量
    -d, --duration    <T>  压测时间
    -t, --threads     <N>  使用多少个线程进行压测

    -s, --script      <S>  指定Lua脚本路径
    -H, --header      <H>  为每一个HTTP请求添加HTTP头
        --latency          在压测结束后，打印延迟统计信息
        --timeout     <T>  超时时间
    -v, --version          打印正在使用的wrk的详细版本信息

  <N>代表数字参数，支持国际单位 (1k, 1M, 1G)
  <T>代表时间参数，支持时间单位 (2s, 2m, 2h)
```

结果显示

```
Running 30s test @ http://www.51mh.com （压测时间30s）
  8 threads and 200 connections （共8个测试线程，200个连接）
  Thread Stats   Avg      Stdev     Max   +/- Stdev
              （平均值） （标准差）（最大值）（正负一个标准差所占比例）
    Latency    46.67ms  215.38ms   1.67s    95.59%
     （延迟）
    Req/Sec     7.91k     1.15k   10.26k    70.77%
     （处理中的请求数）
  Latency Distribution （延迟分布）
```

```
     50%     2.93ms
     75%     3.78ms
     90%     4.73ms
     99%     1.35s  （99分位的延迟）
  1790465 requests in 30.01s, 684.08MB read  （30.01秒内共处理完成了1790465个请求，读取了684.08MB数据）
Requests/sec:  59658.29  （平均每秒处理完成59658.29个请求）
Transfer/sec:     22.79MB  （平均每秒读取数据22.79MB）
```

示例图片：

Wrk 压测命令

◆ wrk –t4 –c300 –d30s --latency "http://www.baidu.com"

解释：使用4个线程300个连接，对百度首页进行了30秒的压测，并要求在压测结果中输出响应延迟信息

```
[root@imooc-tool wrk]# ./wrk -t 4 -c300 -d30s --latency  "http://www.baidu.c
om/"
Running 30s test @ http://www.baidu.com/        ← (对百度首页 压测30s)
  4 threads and 300 connections                 ← (共四个线程，300个连接)
  Thread Stats   Avg      Stdev     Max   +/- Stdev   ← +/-标准差占比
(延迟) Latency   261.28ms  352.12ms   1.97s   82.79%
    Req/Sec   214.99    129.35     2.38k    97.42%
  (Avg: 平均值  Stdev: 标准差  Max: 最大值)
  Latency Distribution
     50%   93.36ms
     75%  310.60ms                              ← 延迟分布
     90%  730.81ms
     99%     1.50s
  25702 requests in 30.02s, 379.03MB read       ← 30.02秒内供处理了25702个请求，读取了379.03MB数据
  Socket errors: connect 0, read 141, write 0, timeout 486
Requests/sec:    856.12                         ← 平均每秒处理 856.12 个请求，通常说的QPS
Transfer/sec:     12.63MB
```

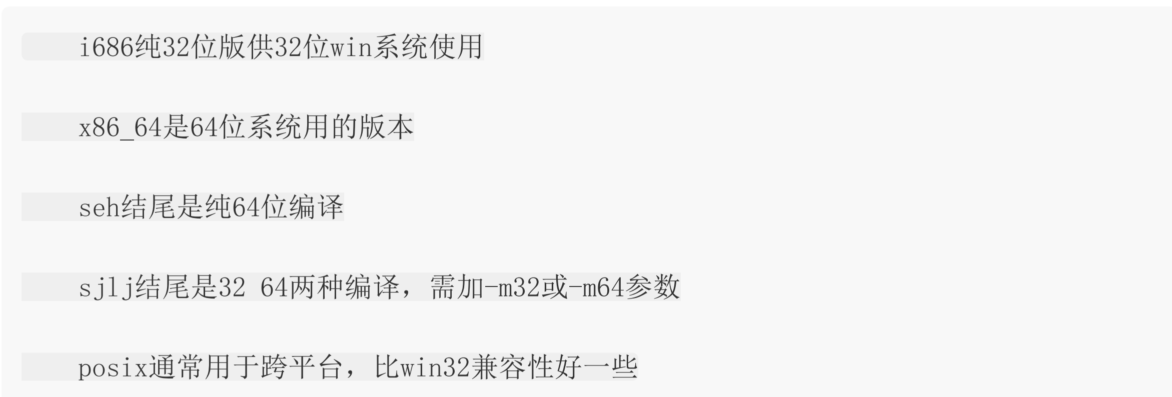
gcc安装

GCC原名为GNU C语言编译器（GNU C Compiler），只能处理C语言。但其很快扩展，变得可处理C++，后来又扩展为能够支持更多编程语言，如Fortran、Pascal、Objective -C、Java、Ada、Go以及各类处理器架构上的汇编语言等，所以改名GNU编译器套件（GNU Compiler Collection）

MinGW-W64 GCC安装与配置

MinGW-w64下载地址：https://sourceforge.net/projects/mingw-w64/files/

选择合适的版本



```
    i686纯32位版供32位win系统使用

    x86_64是64位系统用的版本

    seh结尾是纯64位编译

    sjlj结尾是32 64两种编译，需加-m32或-m64参数

    posix通常用于跨平台，比win32兼容性好一些
```

我下载的是：x86_64-posix-sjlj(这个已经编译好了解压完就能用）

配置过程：

- 下载压缩包
- 将压缩包解压到硬盘
- 配置编译环境

假设我这里将下载的压缩包解压到：E:\mingw 目录下

配置环境变量：

右击“计算机” ->属性 ->高级系统设置 ->环境环境 ->系统变量 ->“Path”变量 ->编辑，追加 ;E:\mingw\bin

验证环境是否安装

开始菜单 -> 附件 -> “命令行提示符” 或 “Win键+R”组合键，输入：cmd

在命令行下输入：gcc -v

如有输出GCC信息则配置成功，配置成功如图：

```
Microsoft Windows [版本 10.0.17763.1098]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\AT>gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=E:/mingw/bin/../libexec/gcc/x86_64-w64-mingw32/8.1.0/lto-wrapper.exe
Target: x86_64-w64-mingw32
Configured with: ../../../src/gcc-8.1.0/configure --host=x86_64-w64-mingw32 --build=x86_64-w64-mingw32 --target=x86_64-w
64-mingw32 --prefix=/mingw64 --with-sysroot=/c/mingw810/x86_64-810-posix-sjlj-rt_v6-rev0/mingw64 --enable-shared --enabl
e-static --enable-targets=all --enable-multilib --enable-languages=c,c++,fortran,lto --enable-libstdcxx-time=yes --enabl
e-threads=posix --enable-libgomp --enable-libatomic --enable-lto --enable-graphite --enable-checking=release --enable-fu
lly-dynamic-string --enable-version-specific-runtime-libs --enable-sjlj-exceptions --disable-libstdcxx-pch --disable-lib
stdcxx-debug --enable-bootstrap --disable-rpath --disable-win32-registry --disable-nls --disable-werror --disable-symver
s --with-gnu-as --with-gnu-ld --with-arch-32=i686 --with-arch-64=nocona --with-tune-32=generic --with-tune-64=core2 --wi
th-libiconv --with-system-zlib --with-gmp=/c/mingw810/prerequisites/x86_64-w64-mingw32-static --with-mpfr=/c/mingw810/pr
erequisites/x86_64-w64-mingw32-static --with-mpc=/c/mingw810/prerequisites/x86_64-w64-mingw32-static --with-isl=/c/mingw
810/prerequisites/x86_64-w64-mingw32-static --with-pkgversion='x86_64-posix-sjlj-rev0, Built by MinGW-W64 project' --wit
h-bugurl=https://sourceforge.net/projects/mingw-w64 CFLAGS='-O2 -pipe -fno-ident -I/c/mingw810/x86_64-810-posix-sjlj-rt_
v6-rev0/mingw64/opt/include -I/c/mingw810/prerequisites/x86_64-zlib-static/include -I/c/mingw810/prerequisites/x86_64-w6
4-mingw32-static/include' CXXFLAGS='-O2 -pipe -fno-ident -I/c/mingw810/x86_64-810-posix-sjlj-rt_v6-rev0/mingw64/opt/incl
ude -I/c/mingw810/prerequisites/x86_64-zlib-static/include -I/c/mingw810/prerequisites/x86_64-w64-mingw32-static/include
' CPPFLAGS=' -I/c/mingw810/x86_64-810-posix-sjlj-rt_v6-rev0/mingw64/opt/include -I/c/mingw810/prerequisites/x86_64-zlib-
static/include -I/c/mingw810/prerequisites/x86_64-w64-mingw32-static/include' LDFLAGS='-pipe -fno-ident -L/c/mingw810/x8
6_64-810-posix-sjlj-rt_v6-rev0/mingw64/opt/lib -L/c/mingw810/prerequisites/x86_64-zlib-static/lib -L/c/mingw810/prerequi
sites/x86_64-w64-mingw32-static/lib '
Thread model: posix
gcc version 8.1.0 (x86_64-posix-sjlj-rev0, Built by MinGW-W64 project)

C:\Users\AT>
```

gim即时通讯

简要介绍

gim是一个即时通讯服务器，代码全部使用golang完成。主要功能
1.支持tcp，websocket接入
2.离线消息同步
3.多业务接入
4.单用户多设备同时在线
5.单聊，群聊，以及超大群聊天场景
6.支持服务水平扩展

使用技术：

数据库：Mysql+Redis
通讯框架：GRPC
长连接通讯协议：Protocol Buffers
日志框架：Zap

安装部署

1.首先安装MySQL，Redis
2.创建数据库gim，执行sql/create_table.sql，完成初始化表的创建（数据库包含提供测试的一些初始数据）
3.修改config下配置文件，使之和你本地配置一致
4.分别切换到cmd的tcp_conn,ws_conn,logic目录下，执行go run main.go,启动TCP连接层服务器,WebSocket连接层服务器,逻辑层服务器
注意：tcp_conn使用了linux的系统调用，所以只能在linux环境下启动，如果是其他环境，可以在安装docker的前提下，
执行run.sh启动

迅速跑通本地测试

1.在test目录下，tcp_conn或者ws_conn目录下，执行go run main,启动测试脚本
2.根据提示,依次填入app_id,user_id,device_id,sync_sequence(中间空格空开)，进行长连接登录；数据库device表中已经初始化了一些设备信息，用作测试
3.执行api/logic/logic_client_ext_test.go下的TestLogicExtServer_SendMessage函数，发送消息

业务服务器如何接入

1.首先生成私钥和公钥
2.在app表里根据你的私钥添加一条app记录
3.将app_id和公钥保存到业务服务器
4.将用户通过LogicClientExtServer.AddUser接口添加到IM服务器
5.通过LogicClientExtServer.RegisterDevice接口注册设备，获取设备id(device_id)
6.将app_id，user_id,device_id用公钥通过公钥加密，生成token,相应库的代码在pkg/util/aes.go
7.接下来使用这个token，app就可以和IM服务器交互

rpc接口简介

项目所有的proto协议在gim/public/proto/目录下
1.tcp.proto
长连接通讯协议
2.logic_client.ext.proto
对客户端（Android设备，IOS设备）提供的rpc协议
3.logic_server.ext.proto
对业务服务器提供的rpc协议
4.logic.int.proto
对conn服务层提供的rpc协议
5.conn.int.proto
对logic服务层提供的rpc协议

项目目录简介

项目结构遵循 https://github.com/golang-standards/project-layout

```
api:          服务对外提供的grpc接口
cmd:          服务启动入口
config:       服务配置
internal:     每个服务私有代码
pkg:          服务共有代码
sql:          项目sql文件
test:         长连接测试脚本
```

服务简介

1.tcp_conn
维持与客户端的TCP长连接，心跳，以及TCP拆包粘包，消息编解码
2.ws_conn
维持与客户端的WebSocket长连接，心跳，消息编解码
3.logic
设备信息，用户信息，群组信息管理，消息转发逻辑

TCP拆包粘包

遵循LV的协议格式，一个消息包分为两部分，消息字节长度以及消息内容。
这里为了减少内存分配，拆出来的包的内存复用读缓存区内存。
拆包流程：
1.首先从系统缓存区读取字节流到buffer
2.根据包头的length字段，检查报的value字段的长度是否大于等于length
3.如果大于，返回一个完整包（此包内存复用），重复步骤2
4.如果小于，将buffer的有效字节前移，重复步骤1

单用户多设备支持，离线消息同步

每个用户都会维护一个自增的序列号，当用户A给用户B发送消息是，首先会获取A的最大序列号，设置为这条消息的seq，持久化到用户A的消息列表，
再通过长连接下发到用户A账号登录的所有设备，再获取用户B的最大序列号，设置为这条消息的seq，持久化到用户B的消息列表，再通过长连接下发
到用户B账号登录的所有设备。
假如用户的某个设备不在线，在设备长连接登录时，用本地收到消息的最大序列号，到服务器做消息同步，这样就可以保证离线消息不丢失。

读扩散和写扩散

首先解释一下，什么是读扩散，什么是写扩散

读扩散

简介：群组成员发送消息时，先建立一个会话，都将这个消息写入这个会话中，同步离线消息时，需要同步这个会话的未同步消息
优点：每个消息只需要写入数据库一次就行，减少数据库访问次数，节省数据库空间
缺点：一个用户有n个群组，客户端每次同步消息时，要上传n个序列号，服务器要对这n个群组分别做消息同步

写扩散

简介：在群组中，每个用户维持一个自己的消息列表，当群组中有人发送消息时，给群组的每个用户的消息列表插入一条消息即可
优点：每个用户只需要维护一个序列号和消息列表
缺点：一个群组有多少人，就要插入多少条消息，当群组成员很多时，DB的压力会增大

消息转发逻辑选型以及特点

普通群组：

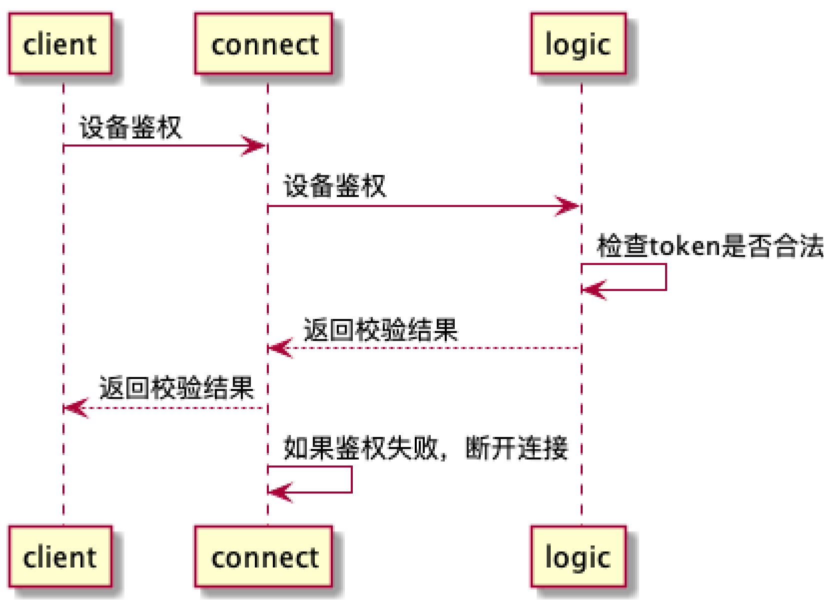
采用写扩散，群组成员信息持久化到数据库保存。支持消息离线同步。

超大群组：

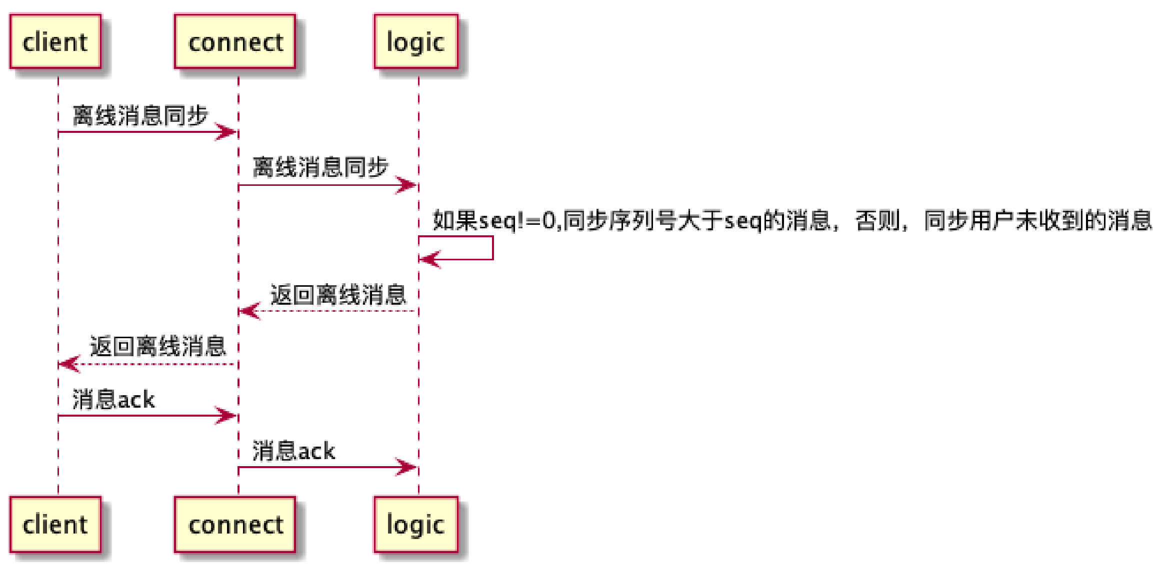
采用读扩散，群组成员信息保存到redis,不支持离线消息同步。

核心流程时序图

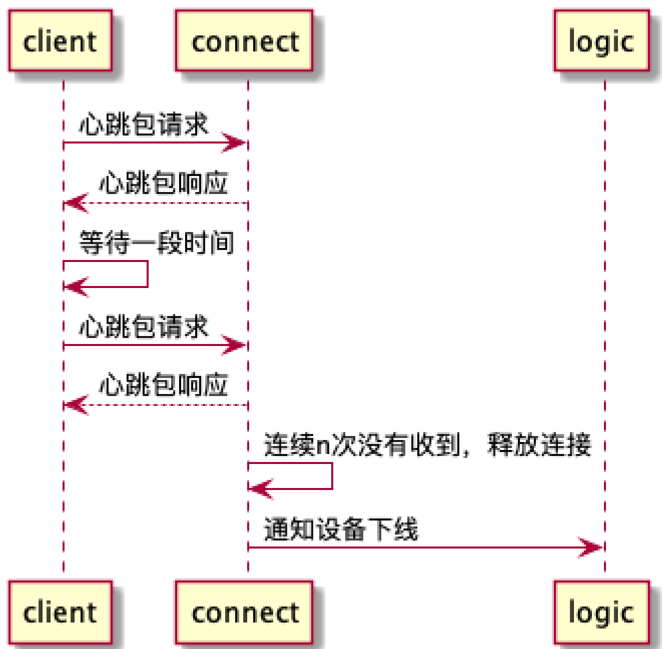
长连接登录



离线消息同步

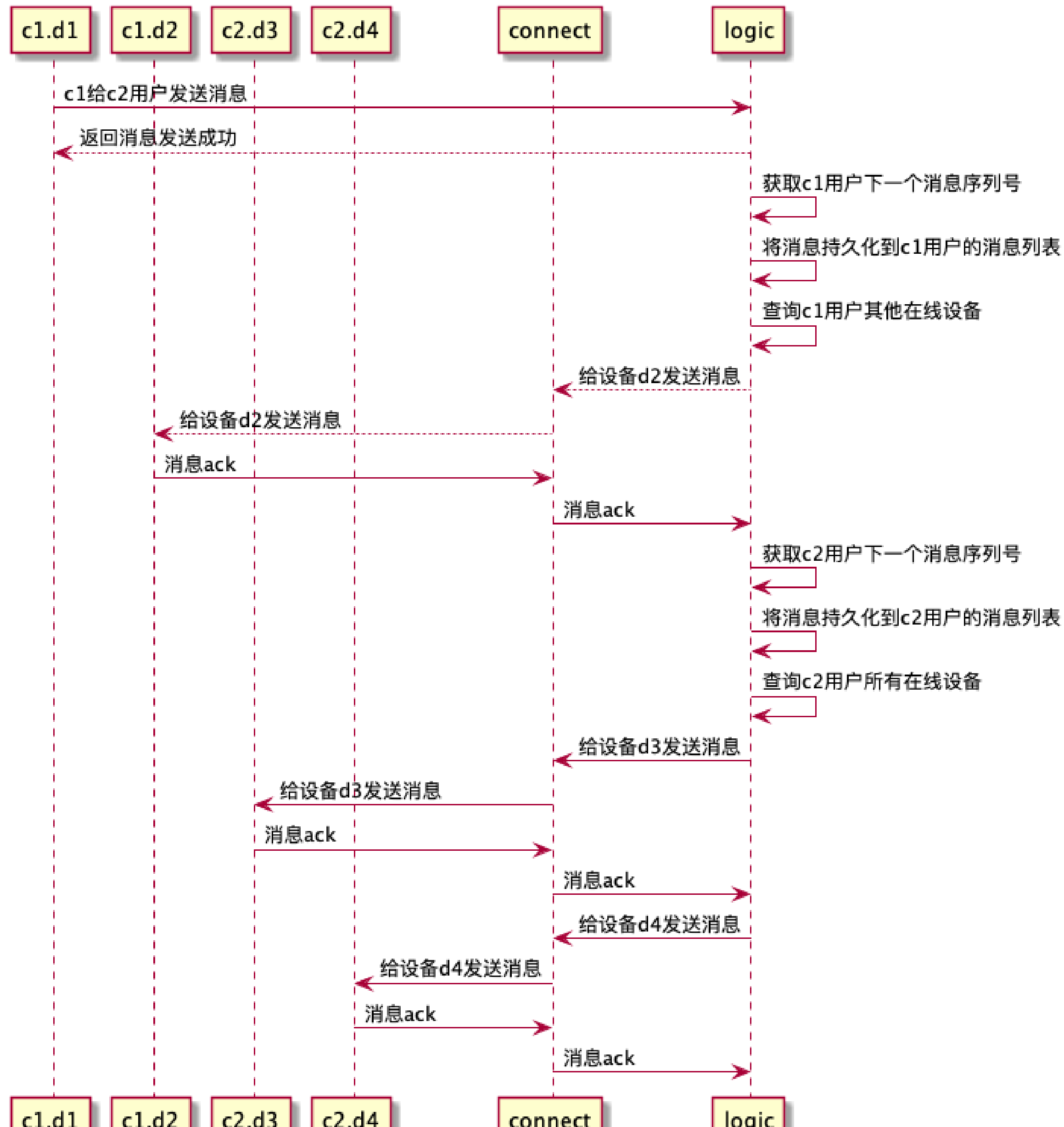


心跳



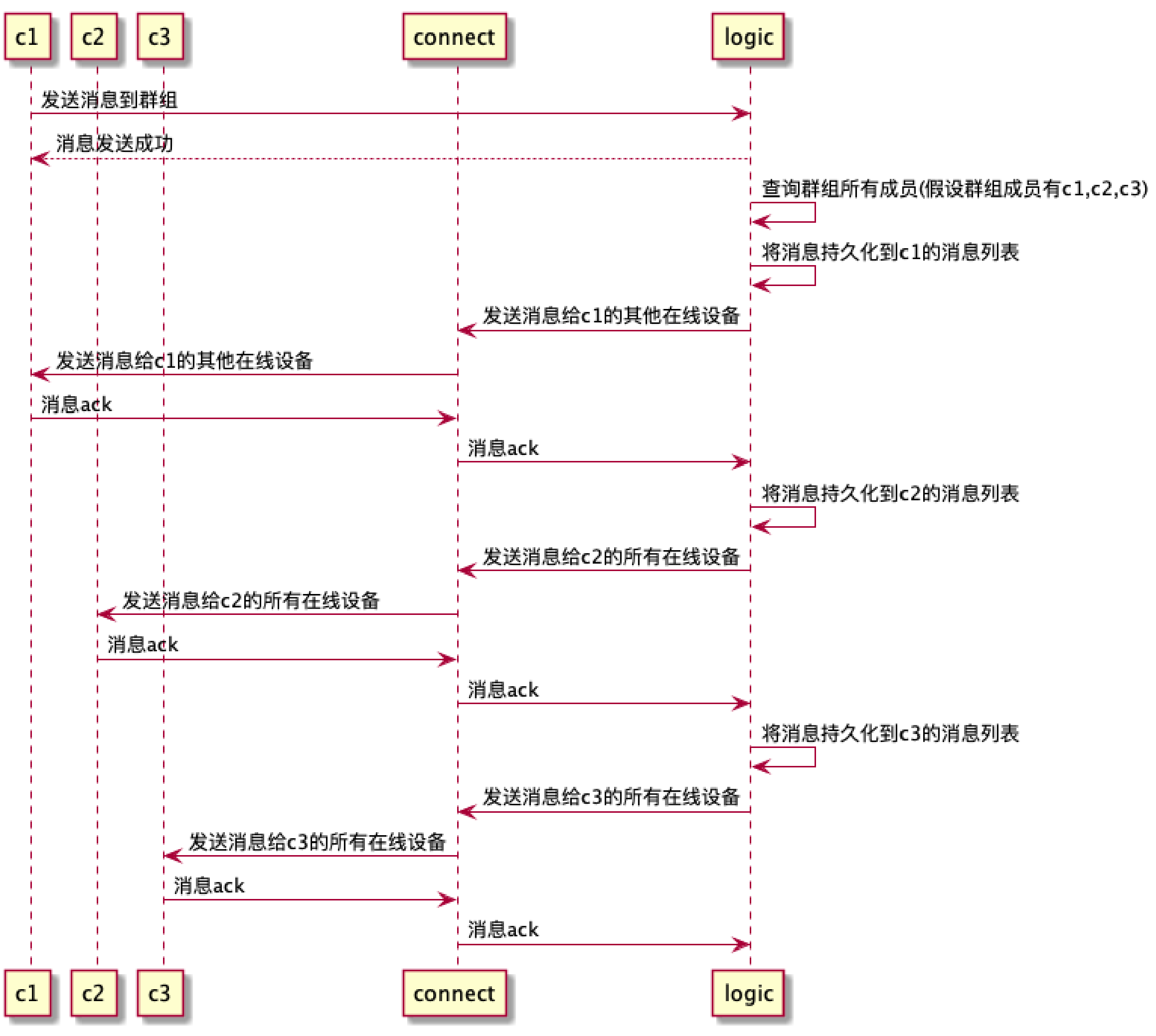
消息单发

c1.d1和c1.d2分别表示c1用户的两个设备d1和d2,c2.d3和c2.d4同理

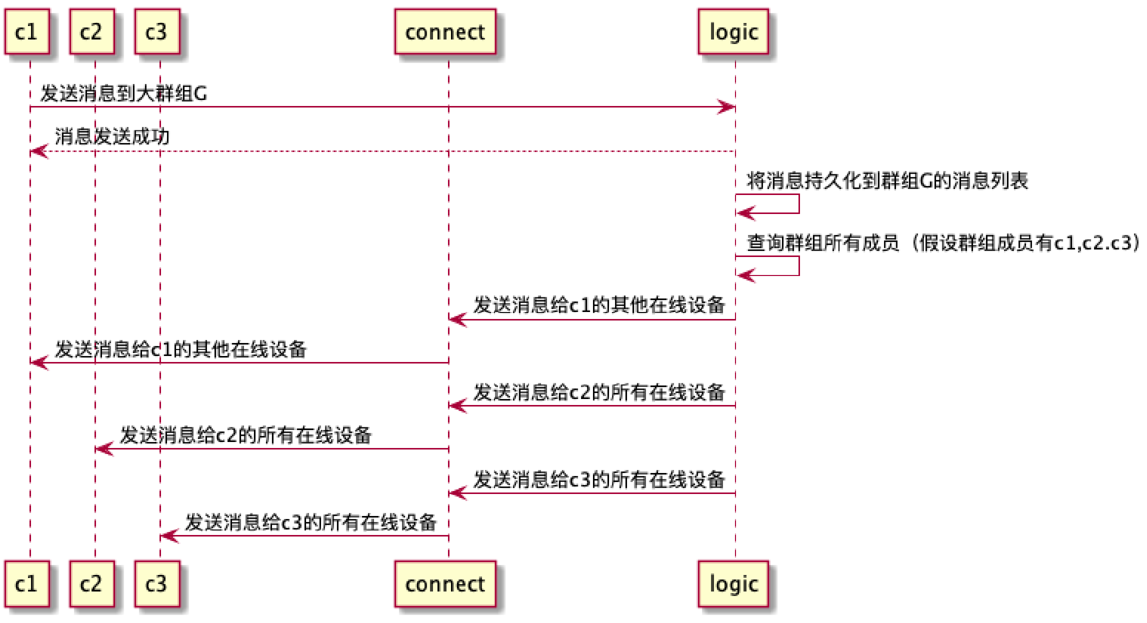


小群消息群发

c1,c2.c3表示一个群组中的三个用户



大群消息群发



错误处理,链路追踪,日志打印

系统中的错误一般可以归类为两种，一种是业务定义的错误，一种就是未知的错误，在业务正式上线的时候，业务定义的错误的属于正常业务逻辑，不需要打印出来，
但是未知的错误，我们就需要打印出来，我们不仅要知道是什么错误，还要知道错误的调用堆栈，所以这里我对GRPC的错误进行了一些封装，使之包含调用堆栈。

```
func WrapError(err error) error {
    if err == nil {
        return nil
    }

    s := &spb.Status{
        Code:    int32(codes.Unknown),
        Message: err.Error(),
        Details: []*any.Any{
            {
                TypeUrl: TypeUrlStack,
                Value:   util.Str2bytes(stack()),
            },
        },
    }
    return status.FromProto(s).Err()
}
// Stack 获取堆栈信息
func stack() string {
    var pc = make([]uintptr, 20)
    n := runtime.Callers(3, pc)

    var build strings.Builder
    for i := 0; i < n; i++ {
        f := runtime.FuncForPC(pc[i] - 1)
        file, line := f.FileLine(pc[i] - 1)
        n := strings.Index(file, name)
        if n != -1 {
            s := fmt.Sprintf(" %s:%d \n", file[n:], line)
            build.WriteString(s)
        }
    }
    return build.String()
}
```

这样，不仅可以拿到错误的堆栈，错误的堆栈也可以跨RPC传输，但是，这样你只能拿到当前服务的堆栈，却不能拿到调用方的堆栈，就比如说，A服务调用
B服务，当B服务发生错误时，在A服务通过日志打印错误的时候，我们只打印了B服务的调用堆

栈，怎样可以把A服务的堆栈打印出来。我们在A服务调用的地方也获取一次堆栈。

```
func WrapRPCError(err error) error {
    if err == nil {
        return nil
    }
    e, _ := status.FromError(err)
    s := &spb.Status{
        Code:    int32(e.Code()),
        Message: e.Message(),
        Details: []*any.Any{
            {
                TypeUrl: TypeUrlStack,
                Value:   util.Str2bytes(GetErrorStack(e) + " --grpc-- \n" + stack()),
            },
        },
    }
    return status.FromProto(s).Err()
}

func interceptor(ctx context.Context, method string, req, reply interface{}, cc *grpc.ClientConn, invoker grpc.UnaryInvoker, opts ...grpc.CallOption) error {
    err := invoker(ctx, method, req, reply, cc, opts...)
    return gerrors.WrapRPCError(err)
}

var LogicIntClient   pb.LogicIntClient

func InitLogicIntClient(addr string) {
    conn, err := grpc.DialContext(context.TODO(), addr, grpc.WithInsecure(), grpc.WithUnaryInterceptor(interceptor))
    if err != nil {
        logger.Sugar.Error(err)
        panic(err)
    }

    LogicIntClient = pb.NewLogicIntClient(conn)
}
```

像这样，就可以获取完整一次调用堆栈。
错误打印也没有必要在函数返回错误的时候，每次都去打印。因为错误已经包含了堆栈信息

```
// 错误的方式
if err != nil {
    logger.Sugar.Error(err)
    return err
}

// 正确的方式
if err != nil {
    return err
}
```

然后，我们在上层统一打印就可以

```
func startServer {
    extListen, err := net.Listen("tcp", conf.LogicConf.ClientRPCExtListenAddr)
    if err != nil {
        panic(err)
    }
    extServer := grpc.NewServer(grpc.UnaryInterceptor(LogicClientExtIntercepto
r))
    pb.RegisterLogicClientExtServer(extServer, &LogicClientExtServer{})
    err = extServer.Serve(extListen)
    if err != nil {
        panic(err)
    }
}

func LogicClientExtInterceptor(ctx context.Context, req interface{}, info *grpc.
UnaryServerInfo, handler grpc.UnaryHandler) (resp interface{}, err error) {
    defer func() {
        logPanic("logic_client_ext_interceptor", ctx, req, info, &err)
    }()

    resp, err = handler(ctx, req)
    logger.Logger.Debug("logic_client_ext_interceptor", zap.Any("info", info), z
ap.Any("ctx", ctx), zap.Any("req", req),
        zap.Any("resp", resp), zap.Error(err))

    s, _ := status.FromError(err)
    if s.Code() != 0 && s.Code() < 1000 {
        md, _ := metadata.FromIncomingContext(ctx)
        logger.Logger.Error("logic_client_ext_interceptor", zap.String("method",
info.FullMethod), zap.Any("md", md), zap.Any("req", req),
            zap.Any("resp", resp), zap.Error(err), zap.String("stack", gerrors.G
etErrorStack(s)))
```

```
    }
    return
}
```

这样做的前提就是，在业务代码中透传context,golang不像其他语言，可以在线程本地保存变量，像Java的ThreadLocal，所以只能通过函数参数的形式进行传递，gim中，service层函数的第一个参数
都是context，但是dao层和cache层就不需要了，不然，显得代码臃肿。
最后可以在客户端的每次请求添加一个随机的request_id，这样客户端到服务的每次请求都可以串起来了。

```go
func getCtx() context.Context {
    token, _ := util.GetToken(1, 2, 3, time.Now().Add(1*time.Hour).Unix(), util.PublicKey)
    return metadata.NewOutgoingContext(context.TODO(), metadata.Pairs(
        "app_id", "1",
        "user_id", "2",
        "device_id", "3",
        "token", token,
        "request_id", strconv.FormatInt(time.Now().UnixNano(), 10)))
}
```

github

https://github.com/alberliu/gim

开源仓库

请跳转
http://www.topgoer.cn/docs/gokaiyuancangku/gokaiyuancangku-1c8rasb5s2caj

其他

Golang 58个坑

资料下载

零碎知识点

学习路线图

Golang 58个坑

前言

Go 是一门简单有趣的编程语言，与其他语言一样，在使用时不免会遇到很多坑，不过它们大多不是 Go 本身的设计缺陷。如果你刚从其他语言转到 Go，那这篇文章里的坑多半会踩到。

如果花时间学习官方 doc、wiki、讨论邮件列表、 Rob Pike 的大量文章以及 Go 的源码，会发现这篇文章中的坑是很常见的，新手跳过这些坑，能减少大量调试代码的时间。

初级篇：1-34

1.左大括号 { 不能单独放一行

在其他大多数语言中，{ 的位置你自行决定。Go比较特别，遵守分号注入规则（automatic semicolon injection）：编译器会在每行代码尾部特定分隔符后加;来分隔多条语句，比如会在) 后加分号：

```
// 错误示例
func main()
{
    println("www.topgoer.com是个不错的go语言中文文档")
}

// 等效于
func main();    // 无函数体
{
    println("hello world")
}
```

```
    ./main.go: missing function body
    ./main.go: syntax error: unexpected semicolon or newline before {
```

```
// 正确示例
func main() {
    println("www.topgoer.com是个不错的go语言中文文档")
}
```

2.未使用的变量

如果在函数体代码中有未使用的变量，则无法通过编译，不过全局变量声明但不使用是可以的。即使变量声明后为变量赋值，依旧无法通过编译，需在某处使用它：

```
// 错误示例
var gvar int      // 全局变量，声明不使用也可以

func main() {
    var one int      // error: one declared and not used
    two := 2      // error: two declared and not used
    var three int     // error: three declared and not used
    three = 3
}


// 正确示例
// 可以直接注释或移除未使用的变量
func main() {
    var one int
    _ = one

    two := 2
    println(two)

    var three int
    one = three

    var four int
    four = four
}
```

3.未使用的 import

如果你 import一个包，但包中的变量、函数、接口和结构体一个都没有用到的话，将编译失败。可以使用 `_` 下划线符号作为别名来忽略导入的包，从而避免编译错误，这只会执行 package 的 init()

```
// 错误示例
import (
    "fmt"    // imported and not used: "fmt"
    "log"    // imported and not used: "log"
    "time"    // imported and not used: "time"
)

func main() {
}
```

```go
// 正确示例
// 可以使用 goimports 工具来注释或移除未使用到的包
import (
	_ "fmt"
	"log"
	"time"
)

func main() {
	_ = log.Println
	_ = time.Now
}
```

4.简短声明的变量只能在函数内部使用

```go
// 错误示例
myvar := 1	// syntax error: non-declaration statement outside function body
func main() {
}


// 正确示例
var  myvar = 1
func main() {
}
```

5.使用简短声明来重复声明变量

不能用简短声明方式来单独为一个变量重复声明，:=左侧至少有一个新变量，才允许多变量的重复声明：

```go
// 错误示例
func main() {
	one := 0
	one := 1 // error: no new variables on left side of :=
}


// 正确示例
func main() {
	one := 0
	one, two := 1, 2	// two 是新变量，允许 one 的重复声明。比如 error 处理经
```

```
常用同名变量 err
    one, two = two, one    // 交换两个变量值的简写
}
```

6.不能使用简短声明来设置字段的值

struct 的变量字段不能使用 := 来赋值以使用预定义的变量来避免解决：

```
// 错误示例
type info struct {
    result int
}

func work() (int, error) {
    return 3, nil
}

func main() {
    var data info
    data.result, err := work()    // error: non-name data.result on left side of
:=
    fmt.Printf("info: %+v\n", data)
}


// 正确示例
func main() {
    var data info
    var err error    // err 需要预声明

    data.result, err = work()
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("info: %+v\n", data)
}
```

7.不小心覆盖了变量

对从动态语言转过来的开发者来说，简短声明很好用，这可能会让人误会 := 是一个赋值操作符。如果你在新的代码块中像下边这样误用了 :=，编译不会报错，但是变量不会按你的预期工作：

```go
func main() {
    x := 1
    println(x)        // 1
    {
        println(x)    // 1
        x := 2
        println(x)    // 2    // 新的 x 变量的作用域只在代码块内部
    }
    println(x)        // 1
}
```

这是 Go 开发者常犯的错，而且不易被发现。可使用 vet工具来诊断这种变量覆盖，Go 默认不做覆盖检查，添加 -shadow 选项来启用：

```
> go tool vet -shadow main.go
main.go:9: declaration of "x" shadows declaration at main.go:5
```

注意 vet 不会报告全部被覆盖的变量，可以使用 go-nyet 来做进一步的检测：

```
> $GOPATH/bin/go-nyet main.go
main.go:10:3:Shadowing variable `x`
```

8.显式类型的变量无法使用 nil 来初始化

nil 是 interface、function、pointer、map、slice 和 channel 类型变量的默认初始值。但声明时不指定类型，编译器也无法推断出变量的具体类型。

```go
// 错误示例
func main() {
    var x = nil    // error: use of untyped nil
    _ = x
}


// 正确示例
func main() {
    var x interface{} = nil
    _ = x
}
```

9.直接使用值为 nil 的 slice、map

允许对值为 nil 的 slice 添加元素，但对值为 nil 的 map添加元素则会造成运行时 panic

```
// map 错误示例
func main() {
    var m map[string]int
    m["one"] = 1        // error: panic: assignment to entry in nil map
    // m := make(map[string]int)// map 的正确声明，分配了实际的内存
}


// slice 正确示例
func main() {
    var s []int
    s = append(s, 1)
}
```

10.map 容量

在创建 map 类型的变量时可以指定容量，但不能像 slice 一样使用 cap() 来检测分配空间的大小：

```
// 错误示例
func main() {
    m := make(map[string]int, 99)
    println(cap(m))     // error: invalid argument m1 (type map[string]int) for
 cap
}
```

11.string 类型的变量值不能为 nil

对那些喜欢用 nil 初始化字符串的人来说，这就是坑：

```
// 错误示例
func main() {
    var s string = nil    // cannot use nil as type string in assignment
    if s == nil {    // invalid operation: s == nil (mismatched types string and
nil)
        s = "default"
    }
}


// 正确示例
func main() {
    var s string    // 字符串类型的零值是空串 ""
    if s == "" {
```

```
        s = "default"
    }
}
```

12.Array 类型的值作为函数参数

在 C/C++ 中，数组（名）是指针。将数组作为参数传进函数时，相当于传递了数组内存地址的引用，在函数内部会改变该数组的值。

在 Go 中，数组是值。作为参数传进函数时，传递的是数组的原始值拷贝，此时在函数内部是无法更新该数组的：

```
// 数组使用值拷贝传参
func main() {
    x := [3]int{1,2,3}

    func(arr [3]int) {
        arr[0] = 7
        fmt.Println(arr)    // [7 2 3]
    }(x)
    fmt.Println(x)              // [1 2 3]    // 并不是你以为的 [7 2 3]
}
```

如果想修改参数数组：

- 直接传递指向这个数组的指针类型：

```
// 传址会修改原数据
func main() {
    x := [3]int{1,2,3}

    func(arr *[3]int) {
        (*arr)[0] = 7
        fmt.Println(arr)    // &[7 2 3]
    }(&x)
    fmt.Println(x)      // [7 2 3]
}
```

- 直接使用 slice：即使函数内部得到的是 slice 的值拷贝，但依旧会更新 slice 的原始数据（底层 array）

```
// 会修改 slice 的底层 array，从而修改 slice
func main() {
```

```
    x := []int{1, 2, 3}
    func(arr []int) {
        arr[0] = 7
        fmt.Println(x)    // [7 2 3]
    }(x)
    fmt.Println(x)    // [7 2 3]
}
```

13.range 遍历 slice 和 array 时混淆了返回值

与其他编程语言中的 for-in 、foreach 遍历语句不同，Go 中的 range 在遍历时会生成 2 个值，第一个是元素索引，第二个是元素的值：

```
// 错误示例
func main() {
    x := []string{"a", "b", "c"}
    for v := range x {
        fmt.Println(v)    // 1 2 3
    }
}


// 正确示例
func main() {
    x := []string{"a", "b", "c"}
    for _, v := range x {    // 使用 _ 丢弃索引
        fmt.Println(v)
    }
}
```

14.slice 和 array 其实是一维数据

看起来 Go 支持多维的 array 和 slice，可以创建数组的数组、切片的切片，但其实并不是。

对依赖动态计算多维数组值的应用来说，就性能和复杂度而言，用 Go 实现的效果并不理想。

可以使用原始的一维数组、"独立" 的切片、"共享底层数组"的切片来创建动态的多维数组。

1.使用原始的一维数组：要做好索引检查、溢出检测、以及当数组满时再添加值时要重新做内存分配。

2.使用"独立"的切片分两步：

- 创建外部 slice

- 对每个内部 slice 进行内存分配

  注意内部的 slice 相互独立，使得任一内部 slice 增缩都不会影响到其他的 slice

```
// 使用各自独立的 6 个 slice 来创建 [2][3] 的动态多维数组
func main() {
    x := 2
    y := 4

    table := make([][]int, x)
    for i  := range table {
        table[i] = make([]int, y)
    }
}
```

1.使用“共享底层数组”的切片

- 创建一个存放原始数据的容器 slice
- 创建其他的 slice
- 切割原始 slice 来初始化其他的 slice

```
func main() {
    h, w := 2, 4
    raw := make([]int, h*w)

    for i := range raw {
        raw[i] = i
    }

    // 初始化原始 slice
    fmt.Println(raw, &raw[4])    // [0 1 2 3 4 5 6 7] 0xc420012120

    table := make([][]int, h)
    for i := range table {

        // 等间距切割原始 slice，创建动态多维数组 table
        // 0: raw[0*4: 0*4 + 4]
        // 1: raw[1*4: 1*4 + 4]
        table[i] = raw[i*w : i*w + w]
    }

    fmt.Println(table, &table[1][0])    // [[0 1 2 3] [4 5 6 7]] 0xc420012120
}
```

更多关于多维数组的参考

go-how-is-two-dimensional-arrays-memory-representation

what-is-a-concise-way-to-create-a-2d-slice-in-go

15.访问 map 中不存在的 key

和其他编程语言类似，如果访问了 map 中不存在的 key 则希望能返回 nil，比如在 PHP 中：

```
> php -r '$v = ["x"=>1, "y"=>2]; @var_dump($v["z"]);'
NULL
```

Go 则会返回元素对应数据类型的零值，比如 nil、'' 、false 和 0，取值操作总有值返回，故不能通过取出来的值来判断 key 是不是在 map 中。

检查 key 是否存在可以用 map 直接访问，检查返回的第二个参数即可：

```
// 错误的 key 检测方式
func main() {
    x := map[string]string{"one": "2", "two": "", "three": "3"}
    if v := x["two"]; v == "" {
        fmt.Println("key two is no entry")    // 键 two 存不存在都会返回的空字符串
    }
}

// 正确示例
func main() {
    x := map[string]string{"one": "2", "two": "", "three": "3"}
    if _, ok := x["two"]; !ok {
        fmt.Println("key two is no entry")
    }
}
```

16.string 类型的值是常量，不可更改

尝试使用索引遍历字符串，来更新字符串中的个别字符，是不允许的。

string 类型的值是只读的二进制 byte slice，如果真要修改字符串中的字符，将 string 转为 []byte 修改后，再转为 string 即可：

```
// 修改字符串的错误示例
func main() {
    x := "text"
```

```
    x[0] = "T"          // error: cannot assign to x[0]
    fmt.Println(x)
}


// 修改示例
func main() {
    x := "text"
    xBytes := []byte(x)
    xBytes[0] = 'T'     // 注意此时的 T 是 rune 类型
    x = string(xBytes)
    fmt.Println(x)      // Text
}
```

注意： 上边的示例并不是更新字符串的正确姿势，因为一个 UTF8 编码的字符可能会占多个字节，比如汉字就需要 `3~4` 个字节来存储，此时更新其中的一个字节是错误的。

更新字串的正确姿势：将 string 转为 rune slice（此时 1 个 rune 可能占多个 byte），直接更新 rune 中的字符

```
func main() {
    x := "text"
    xRunes := []rune(x)
    xRunes[0] = '我'
    x = string(xRunes)
    fmt.Println(x)      // 我ext
}
```

17.string 与 byte slice 之间的转换

当进行 string 和 byte slice 相互转换时，参与转换的是拷贝的原始值。这种转换的过程，与其他编程语的强制类型转换操作不同，也和新 slice 与旧 slice 共享底层数组不同。

Go 在 string 与 byte slice 相互转换上优化了两点，避免了额外的内存分配：

- 在 map[string] 中查找 key 时，使用了对应的 []byte，避免做 m[string(key)] 的内存分配
- 使用 for range 迭代 string 转换为 []byte 的迭代：for i,v := range []byte(str) {...}

18.string 与索引操作符

对字符串用索引访问返回的不是字符，而是一个 byte 值。

这种处理方式和其他语言一样，比如 PHP 中：

```
> php -r '$name="中文"; var_dump($name);'     # "中文" 占用 6 个字节
string(6) "中文"

> php -r '$name="中文"; var_dump($name[0]);' # 把第一个字节当做 Unicode 字符读取，显示 U+FFFD
string(1) "�"

> php -r '$name="中文"; var_dump($name[0].$name[1].$name[2]);'
string(3) "中"
```

```
func main() {
    x := "ascii"
    fmt.Println(x[0])         // 97
    fmt.Printf("%T\n", x[0])// uint8
}
```

如果需要使用 `for range` 迭代访问字符串中的字符（ `unicode code point / rune` ），标准库中有 `"unicode/utf8"` 包来做 `UTF8` 的相关解码编码。另外 `utf8string` 也有像 `func (s *String) At(i int) rune` 等很方便的库函数。

19.字符串并不都是 UTF8 文本

string 的值不必是 UTF8 文本，可以包含任意的值。只有字符串是文字字面值时才是 UTF8 文本，字串可以通过转义来包含其他数据。

判断字符串是否是 UTF8 文本，可使用 “unicode/utf8” 包中的 ValidString() 函数：

```
func main() {
    str1 := "ABC"
    fmt.Println(utf8.ValidString(str1))    // true

    str2 := "A\xfeC"
    fmt.Println(utf8.ValidString(str2))    // false

    str3 := "A\\xfeC"
    fmt.Println(utf8.ValidString(str3))    // true    // 把转义字符转义成字面值
}
```

20.字符串的长度

在 Python 中：

```
    data = u'♥'
    print(len(data)) # 1
```

然而在 Go 中：

```
func main() {
    char := "♥"
    fmt.Println(len(char))    // 3
}
```

Go 的内建函数 len() 返回的是字符串的 byte 数量，而不是像 Python 中那样是计算 Unicode 字符数。

如果要得到字符串的字符数，可使用 “unicode/utf8” 包中的 RuneCountInString(str string) (n int)

```
func main() {
    char := "♥"
    fmt.Println(utf8.RuneCountInString(char))    // 1
}
```

注意： RuneCountInString 并不总是返回我们看到的字符数，因为有的字符会占用 2 个 rune：

```
func main() {
    char := "é"
    fmt.Println(len(char))    // 3
    fmt.Println(utf8.RuneCountInString(char))    // 2
    fmt.Println("cafe\u0301")    // café    // 法文的 cafe，实际上是两个 rune 的组合
}
```

21.在多行 array、slice、map 语句中缺少 , 号

```
func main() {
    x := []int {
        1,
        2    // syntax error: unexpected newline, expecting comma or }
    }
    y := []int{1,2,}
    z := []int{1,2}
```

```
    // ...
}
```

声明语句中 } 折叠到单行后，尾部的 , 不是必需的。

22.log.Fatal 和 log.Panic 不只是 log

log 标准库提供了不同的日志记录等级，与其他语言的日志库不同，Go 的 log 包在调用 `Fatal*()` 、 `Panic*()` 时能做更多日志外的事，如中断程序的执行等：

```
func main() {
    log.Fatal("Fatal level log: log entry")        // 输出信息后，程序终止执行
    log.Println("Nomal level log: log entry")
}
```

23.对内建数据结构的操作并不是同步的

尽管 Go 本身有大量的特性来支持并发，但并不保证并发的数据安全，用户需自己保证变量等数据以原子操作更新。

goroutine 和 channel 是进行原子操作的好方法，或使用 “sync” 包中的锁。

24.range 迭代 string 得到的值

range 得到的索引是字符值（Unicode point / rune）第一个字节的位置，与其他编程语言不同，这个索引并不直接是字符在字符串中的位置。

注意一个字符可能占多个 rune，比如法文单词 café 中的 é。操作特殊字符可使用norm 包。

for range 迭代会尝试将 string 翻译为 UTF8 文本，对任何无效的码点都直接使用 0XFFFD rune（�）UNicode 替代字符来表示。如果 string 中有任何非 UTF8 的数据，应将 string 保存为 byte slice 再进行操作。

```
func main() {
    data := "A\xfe\x02\xff\x04"
    for _, v := range data {
        fmt.Printf("%#x ", v)    // 0x41 0xfffd 0x2 0xfffd 0x4    // 错误
    }

    for _, v := range []byte(data) {
        fmt.Printf("%#x ", v)    // 0x41 0xfe 0x2 0xff 0x4    // 正确
    }
}
```

25.range 迭代 map

如果你希望以特定的顺序（如按 key 排序）来迭代 map，要注意每次迭代都可能产生不一样的结果。

Go 的运行时是有意打乱迭代顺序的，所以你得到的迭代结果可能不一致。但也并不总会打乱，得到连续相同的 5 个迭代结果也是可能的，如：

```
func main() {
    m := map[string]int{"one": 1, "two": 2, "three": 3, "four": 4}
    for k, v := range m {
        fmt.Println(k, v)
    }
}
```

如果你去 Go Playground 重复运行上边的代码，输出是不会变的，只有你更新代码它才会重新编译。重新编译后迭代顺序是被打乱的：

```
➜ blog go run main.go
one 1
two 2
three 3
four 4
➜ blog go run main.go
two 2
three 3
four 4
one 1
```

26.switch 中的 fallthrough 语句

switch 语句中的 case 代码块会默认带上 break，但可以使用 fallthrough 来强制执行下一个 case 代码块。

```
func main() {
    isSpace := func(char byte) bool {
        switch char {
        case ' ':    // 空格符会直接 break，返回 false // 和其他语言不一样
        // fallthrough    // 返回 true
        case '\t':
            return true
        }
        return false
```

```
    }
    fmt.Println(isSpace('\t'))     // true
    fmt.Println(isSpace(' '))      // false
}
```

不过你可以在 case 代码块末尾使用 fallthrough，强制执行下一个 case 代码块。

也可以改写 case 为多条件判断：

```
func main() {
    isSpace := func(char byte) bool {
        switch char {
        case ' ', '\t':
            return true
        }
        return false
    }
    fmt.Println(isSpace('\t'))     // true
    fmt.Println(isSpace(' '))      // true
}
```

27.自增和自减运算

很多编程语言都自带前置后置的 ++、- 运算。但 Go 特立独行，去掉了前置操作，同时 ++、— 只作为运算符而非表达式。

```
// 错误示例
func main() {
    data := []int{1, 2, 3}
    i := 0
    ++i              // syntax error: unexpected ++, expecting }
    fmt.Println(data[i++])    // syntax error: unexpected ++, expecting :
}


// 正确示例
func main() {
    data := []int{1, 2, 3}
    i := 0
    i++
    fmt.Println(data[i])    // 2
}
```

28.按位取反

很多编程语言使用 ~ 作为一元按位取反（NOT）操作符，Go 重用 ^ XOR 操作符来按位取反：

```
// 错误的取反操作
func main() {
    fmt.Println(~2)        // bitwise complement operator is ~
}


// 正确示例
func main() {
    var d uint8 = 2
    fmt.Printf("%08b\n", d)        // 00000010
    fmt.Printf("%08b\n", ^d)    // 11111101
}
```

同时 ^ 也是按位异或（XOR）操作符。

一个操作符能重用两次，是因为一元的 NOT 操作 NOT 0x02，与二元的 XOR 操作 0x22 XOR 0xff 是一致的。

Go 也有特殊的操作符 AND NOT &^ 操作符，不同位才取1。

```
func main() {
    var a uint8 = 0x82
    var b uint8 = 0x02
    fmt.Printf("%08b [A]\n", a)
    fmt.Printf("%08b [B]\n", b)

    fmt.Printf("%08b (NOT B)\n", ^b)
    fmt.Printf("%08b ^ %08b = %08b [B XOR 0xff]\n", b, 0xff, b^0xff)

    fmt.Printf("%08b ^ %08b = %08b [A XOR B]\n", a, b, a^b)
    fmt.Printf("%08b & %08b = %08b [A AND B]\n", a, b, a&b)
    fmt.Printf("%08b &^%08b = %08b [A 'AND NOT' B]\n", a, b, a&^b)
    fmt.Printf("%08b&(^%08b)= %08b [A AND (NOT B)]\n", a, b, a&(^b))
}
```

```
    10000010 [A]
    00000010 [B]
    11111101 (NOT B)
    00000010 ^ 11111111 = 11111101 [B XOR 0xff]
    10000010 ^ 00000010 = 10000000 [A XOR B]
    10000010 & 00000010 = 00000010 [A AND B]
```

```
    10000010 &^00000010 = 10000000 [A 'AND NOT' B]
    10000010&(^00000010)= 10000000 [A AND (NOT B)]
```

29.运算符的优先级

除了位清除（bit clear）操作符，Go 也有很多和其他语言一样的位操作符，但优先级另当别论。

```
func main() {
    fmt.Printf("0x2 & 0x2 + 0x4 -> %#x\n", 0x2&0x2+0x4)     // & 优先 +
    //prints: 0x2 & 0x2 + 0x4 -> 0x6
    //Go:     (0x2 & 0x2) + 0x4
    //C++:     0x2 & (0x2 + 0x4) -> 0x2

    fmt.Printf("0x2 + 0x2 << 0x1 -> %#x\n", 0x2+0x2<<0x1)     // << 优先 +
    //prints: 0x2 + 0x2 << 0x1 -> 0x6
    //Go:      0x2 + (0x2 << 0x1)
    //C++:    (0x2 + 0x2) << 0x1 -> 0x8

    fmt.Printf("0xf | 0x2 ^ 0x2 -> %#x\n", 0xf|0x2^0x2)     // | 优先 ^
    //prints: 0xf | 0x2 ^ 0x2 -> 0xd
    //Go:     (0xf | 0x2) ^ 0x2
    //C++:     0xf | (0x2 ^ 0x2) -> 0xf
}
```

优先级列表：

```
    Precedence      Operator
        5               *  /  %  <<  >>  &  &^
        4               +  -  |  ^
        3               ==  !=  <  <=  >  >=
        2               &&
        1               ||
```

30.不导出的 struct 字段无法被 encode

以小写字母开头的字段成员是无法被外部直接访问的，所以 struct 在进行 json、xml、gob 等格式的 encode 操作时，这些私有字段会被忽略，导出时得到零值：

```
func main() {
    in := MyData{1, "two"}
    fmt.Printf("%#v\n", in)     // main.MyData{One:1, two:"two"}

    encoded, _ := json.Marshal(in)
```

```
    fmt.Println(string(encoded))    // {"One":1}    // 私有字段 two 被忽略了

    var out MyData
    json.Unmarshal(encoded, &out)
    fmt.Printf("%#v\n", out)     // main.MyData{One:1, two:""}
}
```

31.程序退出时还有 goroutine 在执行

程序默认不等所有 goroutine 都执行完才退出，这点需要特别注意：

```
// 主程序会直接退出
func main() {
    workerCount := 2
    for i := 0; i < workerCount; i++ {
        go doIt(i)
    }
    time.Sleep(1 * time.Second)
    fmt.Println("all done!")
}

func doIt(workerID int) {
    fmt.Printf("[%v] is running\n", workerID)
    time.Sleep(3 * time.Second)        // 模拟 goroutine 正在执行
    fmt.Printf("[%v] is done\n", workerID)
}
```

如下，main() 主程序不等两个 goroutine 执行完就直接退出了：

```
➜ blog go run main.go
[0] is running
[1] is running
all done!
➜ blog _
```

常用解决办法：使用 “WaitGroup” 变量，它会让主程序等待所有 goroutine 执行完毕再退出。

如果你的 goroutine 要做消息的循环处理等耗时操作，可以向它们发送一条 kill 消息来关闭它们。或直接关闭一个它们都等待接收数据的 channel：

```
// www.topgoer.com go语言中文文档
// 等待所有 goroutine 执行完毕
// 进入死锁
```

```go
func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})

    workerCount := 2
    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doIt(i, done, wg)
    }

    close(done)
    wg.Wait()
    fmt.Println("all done!")
}

func doIt(workerID int, done <-chan struct{}, wg sync.WaitGroup) {
    fmt.Printf("[%v] is running\n", workerID)
    defer wg.Done()
    <-done
    fmt.Printf("[%v] is done\n", workerID)
}
```

执行结果：

```
➜ blog go run main.go
[0] is running
[0] is done
[1] is running
[1] is done
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200160ac)
        /usr/local/go/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200160a0)
        /usr/local/go/src/sync/waitgroup.go:131 +0x72
main.main()
        /Users/wuyin/Go/src/blog/main.go:19 +0xe2
exit status 2
➜ blog _
```

看起来好像 goroutine 都执行完了，然而报错：

> fatal error: all goroutines are asleep - deadlock!

为什么会发生死锁？goroutine 在退出前调用了 wg.Done() ，程序应该正常退出的。

原因是 goroutine 得到的 “WaitGroup” 变量是 var wg WaitGroup 的一份拷贝值，即 doIt() 传参只传值。所以哪怕在每个 goroutine 中都调用了 wg.Done()， 主程序中的 wg 变量并不会受到影响。

```
// www.topgoer.com go语言中文文档
// 等待所有 goroutine 执行完毕
// 使用传址方式为 WaitGroup 变量传参
// 使用 channel 关闭 goroutine

func main() {
    var wg sync.WaitGroup
    done := make(chan struct{})
    ch := make(chan interface{})

    workerCount := 2
    for i := 0; i < workerCount; i++ {
        wg.Add(1)
        go doIt(i, ch, done, &wg)    // wg 传指针，doIt() 内部会改变 wg 的值
    }

    for i := 0; i < workerCount; i++ {    // 向 ch 中发送数据，关闭 goroutine
        ch <- i
    }

    close(done)
    wg.Wait()
    close(ch)
    fmt.Println("all done!")
}

func doIt(workerID int, ch <-chan interface{}, done <-chan struct{}, wg *sync.WaitGroup) {
    fmt.Printf("[%v] is running\n", workerID)
    defer wg.Done()
    for {
        select {
        case m := <-ch:
            fmt.Printf("[%v] m => %v\n", workerID, m)
        case <-done:
            fmt.Printf("[%v] is done\n", workerID)
            return
```

```
        }
    }
}
```

运行效果：

```
➜ blog go run main.go
[1] is running
[1] m => 0
[0] is running
[0] is done
[1] m => 1
[1] is done
all done!
➜ blog _
```

32.向无缓冲的 channel 发送数据，只要 receiver 准备好了就会立刻返回

只有在数据被 receiver 处理时，sender 才会阻塞。因运行环境而异，在 sender 发送完数据后，receiver 的 goroutine 可能没有足够的时间处理下一个数据。如：

```
func main() {
    ch := make(chan string)

    go func() {
        for m := range ch {
            fmt.Println("Processed:", m)
            time.Sleep(1 * time.Second)    // 模拟需要长时间运行的操作
        }
    }()

    ch <- "cmd.1"
    ch <- "cmd.2" // 不会被接收处理
}
```

运行效果：

```
➜ blog go run main.go
Processed: cmd.1
➜ blog _
```

33.向已关闭的 channel 发送数据会造成 panic

从已关闭的 channel 接收数据是安全的：

接收状态值 ok 是 false 时表明 channel 中已没有数据可以接收了。类似的，从有缓冲的 channel 中接收数据，缓存的数据获取完再没有数据可取时，状态值也是 false

向已关闭的 channel 中发送数据会造成 panic：

```
➜ blog go run main.go
2
panic: send on closed channel

goroutine 6 [running]:
main.main.func1(0xc420070060, 0x1)
        /Users/wuyin/Go/src/blog/main.go:12 +0x3f
created by main.main
        /Users/wuyin/Go/src/blog/main.go:11 +0x6f
exit status 2
```

针对上边有 bug 的这个例子，可使用一个废弃 channel done 来告诉剩余的 goroutine 无需再向 ch 发送数据。此时 <- done 的结果是 {}：

```go
func main() {
    ch := make(chan int)
    done := make(chan struct{})

    for i := 0; i < 3; i++ {
        go func(idx int) {
            select {
            case ch <- (idx + 1) * 2:
                fmt.Println(idx, "Send result")
            case <-done:
                fmt.Println(idx, "Exiting")
            }
        }(i)
    }

    fmt.Println("Result: ", <-ch)
    close(done)
    time.Sleep(3 * time.Second)
}
```

运行效果：

```
➜ blog go run main.go
Result:  6
2 Send result
1 Exiting
0 Exiting
➜ blog
```

34.使用了值为 nil 的 channel

在一个值为 nil 的 channel 上发送和接收数据将永久阻塞：

```go
func main() {
    var ch chan int // 未初始化，值为 nil
    for i := 0; i < 3; i++ {
        go func(i int) {
            ch <- i
        }(i)
    }

    fmt.Println("Result: ", <-ch)
    time.Sleep(2 * time.Second)
}
```

runtime 死锁错误：

> fatal error: all goroutines are asleep - deadlock!
> goroutine 1 [chan receive (nil chan)]

利用这个死锁的特性，可以用在 select 中动态的打开和关闭 case 语句块：

```go
func main() {
    inCh := make(chan int)
    outCh := make(chan int)

    go func() {
        var in <-chan int = inCh
        var out chan<- int
        var val int

        for {
            select {
            case out <- val:
                println("--------")
```

```
                out = nil
                in = inCh
            case val = <-in:
                println("++++++++++")
                out = outCh
                in = nil
            }
        }
    }()

    go func() {
        for r := range outCh {
            fmt.Println("Result: ", r)
        }
    }()

    time.Sleep(0)
    inCh <- 1
    inCh <- 2
    time.Sleep(3 * time.Second)
}
```

运行效果：

```
➜  blog go run main.go
++++++++++
--------
++++++++++
Result:  1
Result:  2
--------
➜  blog
```

35.若函数 receiver 传参是传值方式，则无法修改参数的原有值

方法 receiver 的参数与一般函数的参数类似：如果声明为值，那方法体得到的是一份参数的值拷贝，此时对参数的任何修改都不会对原有值产生影响。

除非 receiver 参数是 map 或 slice 类型的变量，并且是以指针方式更新 map 中的字段、slice 中的元素的，才会更新原有值:

```
type data struct {
    num   int
    key   *string
```

```go
	items map[string]bool
}

func (this *data) pointerFunc() {
	this.num = 7
}

func (this data) valueFunc() {
	this.num = 8
	*this.key = "valueFunc.key"
	this.items["valueFunc"] = true
}

func main() {
	key := "key1"

	d := data{1, &key, make(map[string]bool)}
	fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)

	d.pointerFunc()	// 修改 num 的值为 7
	fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)

	d.valueFunc()	// 修改 key 和 items 的值
	fmt.Printf("num=%v  key=%v  items=%v\n", d.num, *d.key, d.items)
}
```

运行结果：

```
➜ blog go run main.go
num=1  key=key1  items=map[]
num=7  key=key1  items=map[]
num=7  key=valueFunc.key  items=map[valueFunc:true]
➜ blog _
```

中级篇：36-51

36.关闭 HTTP 的响应体

使用 HTTP 标准库发起请求、获取响应时，即使你不从响应中读取任何数据或响应为空，都需要手动关闭响应体。新手很容易忘记手动关闭，或者写在了错误的位置：

```go
// 请求失败造成 panic
func main() {
	resp, err := http.Get("https://api.ipify.org?format=json")
```

```
    defer resp.Body.Close()    // resp 可能为 nil，不能读取 Body
    if err != nil {
        fmt.Println(err)
        return
    }

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

    fmt.Println(string(body))
}

func checkError(err error) {
    if err != nil{
        log.Fatalln(err)
    }
}
```

上边的代码能正确发起请求，但是一旦请求失败，变量 resp 值为 nil，造成 panic：

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

应该先检查 HTTP 响应错误为 nil，再调用 resp.Body.Close() 来关闭响应体：

```
// 大多数情况正确的示例
func main() {
    resp, err := http.Get("https://api.ipify.org?format=json")
    checkError(err)

    defer resp.Body.Close()    // 绝大多数情况下的正确关闭方式
    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

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

输出：

```
Get https://api.ipify.org?format=...: x509: certificate signed by unknown author
ity
```

绝大多数请求失败的情况下，resp 的值为 nil 且 err 为 non-nil。但如果你得到的是重定向错误，那它俩的值都是 non-nil，最后依旧可能发生内存泄露。2 个解决办法：

- 可以直接在处理 HTTP 响应错误的代码块中，直接关闭非 nil 的响应体。
- 手动调用 defer 来关闭响应体：

```
// 正确示例
func main() {
    resp, err := http.Get("http://www.baidu.com")

    // 关闭 resp.Body 的正确姿势
    if resp != nil {
        defer resp.Body.Close()
    }

    checkError(err)
    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)
    checkError(err)

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

resp.Body.Close() 早先版本的实现是读取响应体的数据之后丢弃，保证了 keep-alive 的 HTTP 连接能重用处理不止一个请求。但 Go 的最新版本将读取并丢弃数据的任务交给了用户，如果你不处理，HTTP 连接可能会直接关闭而非重用，参考在 Go 1.5 版本文档。

如果程序大量重用 HTTP 长连接，你可能要在处理响应的逻辑代码中加入：

```
    _, err = io.Copy(ioutil.Discard, resp.Body) // 手动丢弃读取完毕的数据
```

如果你需要完整读取响应，上边的代码是需要写的。比如在解码 API 的 JSON 响应数据：

```
    json.NewDecoder(resp.Body).Decode(&data)
```

37.关闭 HTTP 连接

一些支持 HTTP1.1 或 HTTP1.0 配置了 connection: keep-alive 选项的服务器会保持一段时间的长连接。但标准库 “net/http” 的连接默认只在服务器主动要求关闭时才断开，所以你的程序可能会消耗完 socket 描述符。解决办法有 2 个，请求结束后：

- 直接设置请求变量的 Close 字段值为 true，每次请求结束后就会主动关闭连接。
- 设置 Header 请求头部选项 Connection: close，然后服务器返回的响应头部也会有这个选项，此时 HTTP 标准库会主动断开连接。

```go
// 主动关闭连接
func main() {
	req, err := http.NewRequest("GET", "http://golang.org", nil)
	checkError(err)

	req.Close = true
	//req.Header.Add("Connection", "close")	// 等效的关闭方式

	resp, err := http.DefaultClient.Do(req)
	if resp != nil {
		defer resp.Body.Close()
	}
	checkError(err)

	body, err := ioutil.ReadAll(resp.Body)
	checkError(err)

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

你可以创建一个自定义配置的 HTTP transport 客户端，用来取消 HTTP 全局的复用连接：

```go
func main() {
	tr := http.Transport{DisableKeepAlives: true}
	client := http.Client{Transport: &tr}

	resp, err := client.Get("https://golang.google.cn/")
	if resp != nil {
		defer resp.Body.Close()
	}
	checkError(err)

	fmt.Println(resp.StatusCode)	// 200

	body, err := ioutil.ReadAll(resp.Body)
	checkError(err)

	fmt.Println(len(string(body)))
}
```

根据需求选择使用场景：

- 若你的程序要向同一服务器发大量请求，使用默认的保持长连接。

- 若你的程序要连接大量的服务器，且每台服务器只请求一两次，那收到请求后直接关闭连接。或增加最大文件打开数 fs.file-max 的值。

38.将 JSON 中的数字解码为 interface 类型

在 encode/decode JSON 数据时，Go 默认会将数值当做 float64 处理，比如下边的代码会造成 panic：

```
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    if err := json.Unmarshal(data, &result); err != nil {
        log.Fatalln(err)
    }

    fmt.Printf("%T\n", result["status"])    // float64
    var status = result["status"].(int)    // 类型断言错误
    fmt.Println("Status value: ", status)
}
```

> panic: interface conversion: interface {} is float64, not int

如果你尝试 decode 的 JSON 字段是整型，你可以：

- 将 int 值转为 float 统一使用
- 将 decode 后需要的 float 值转为 int 使用

```
// 将 decode 的值转为 int 使用
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    if err := json.Unmarshal(data, &result); err != nil {
        log.Fatalln(err)
    }

    var status = uint64(result["status"].(float64))
    fmt.Println("Status value: ", status)
}
```

- 使用 Decoder 类型来 decode JSON 数据，明确表示字段的值类型

```
// 指定字段类型
func main() {
    var data = []byte(`{"status": 200}`)
    var result map[string]interface{}

    var decoder = json.NewDecoder(bytes.NewReader(data))
    decoder.UseNumber()

    if err := decoder.Decode(&result); err != nil {
        log.Fatalln(err)
    }

    var status, _ = result["status"].(json.Number).Int64()
    fmt.Println("Status value: ", status)
}

 // 你可以使用 string 来存储数值数据，在 decode 时再决定按 int 还是 float 使用
 // 将数据转为 decode 为 string
 func main() {
     var data = []byte({"status": 200})
      var result map[string]interface{}
      var decoder = json.NewDecoder(bytes.NewReader(data))
      decoder.UseNumber()
      if err := decoder.Decode(&result); err != nil {
          log.Fatalln(err)
      }
    var status uint64
      err := json.Unmarshal([]byte(result["status"].(json.Number).String()), &st
atus);
    checkError(err)
       fmt.Println("Status value: ", status)
}
```

使用 struct 类型将你需要的数据映射为数值型

```
// struct 中指定字段类型
func main() {
      var data = []byte(`{"status": 200}`)
      var result struct {
          Status uint64 `json:"status"`
      }

      err := json.NewDecoder(bytes.NewReader(data)).Decode(&result)
      checkError(err)
```

```
    fmt.Printf("Result: %+v", result)
}
```

- 可以使用 struct 将数值类型映射为 json.RawMessage 原生数据类型
  适用于如果 JSON 数据不着急 decode 或 JSON 某个字段的值类型不固定等情况：

```
// 状态名称可能是 int 也可能是 string，指定为 json.RawMessage 类型
func main() {
    records := [][]byte{
        []byte(`{"status":200, "tag":"one"}`),
        []byte(`{"status":"ok", "tag":"two"}`),
    }

    for idx, record := range records {
        var result struct {
            StatusCode uint64
            StatusName string
            Status     json.RawMessage `json:"status"`
            Tag        string          `json:"tag"`
        }

        err := json.NewDecoder(bytes.NewReader(record)).Decode(&result)
        checkError(err)

        var name string
        err = json.Unmarshal(result.Status, &name)
        if err == nil {
            result.StatusName = name
        }

        var code uint64
        err = json.Unmarshal(result.Status, &code)
        if err == nil {
            result.StatusCode = code
        }

        fmt.Printf("[%v] result => %+v\n", idx, result)
    }
```

39.struct、array、slice 和 map 的值比较

可以使用相等运算符 == 来比较结构体变量，前提是两个结构体的成员都是可比较的类型：

```
type data struct {
    num     int
    fp      float32
    complex complex64
    str     string
    char    rune
    yes     bool
    events  <-chan string
    handler interface{}
    ref     *byte
    raw     [10]byte
}

func main() {
    v1 := data{}
    v2 := data{}
    fmt.Println("v1 == v2: ", v1 == v2)    // true
}
```

如果两个结构体中有任意成员是不可比较的，将会造成编译错误。注意数组成员只有在数组元素可比较时候才可比较。

```
type data struct {
    num    int
    checks [10]func() bool        // 无法比较
    doIt   func() bool        // 无法比较
    m      map[string]string    // 无法比较
    bytes  []byte            // 无法比较
}

func main() {
    v1 := data{}
    v2 := data{}

    fmt.Println("v1 == v2: ", v1 == v2)
}
```

> invalid operation: v1 == v2 (struct containing [10]func() bool cannot be compared)

Go 提供了一些库函数来比较那些无法使用 == 比较的变量，比如使用 “reflect” 包的 DeepEqual() ：

```go
// 比较相等运算符无法比较的元素
func main() {
	v1 := data{}
	v2 := data{}
	fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))	// true

	m1 := map[string]string{"one": "a", "two": "b"}
	m2 := map[string]string{"two": "b", "one": "a"}
	fmt.Println("v1 == v2: ", reflect.DeepEqual(m1, m2))	// true

	s1 := []int{1, 2, 3}
	s2 := []int{1, 2, 3}
		// 注意两个 slice 相等，值和顺序必须一致
	fmt.Println("v1 == v2: ", reflect.DeepEqual(s1, s2))	// true
}
```

这种比较方式可能比较慢，根据你的程序需求来使用。DeepEqual() 还有其他用法：

```go
func main() {
	var b1 []byte = nil
	b2 := []byte{}
	fmt.Println("b1 == b2: ", reflect.DeepEqual(b1, b2))	// false
}
```

注意：

- DeepEqual() 并不总适合于比较 slice

```go
func main() {
	var str = "one"
	var in interface{} = "one"
	fmt.Println("str == in: ", reflect.DeepEqual(str, in))	// true

	v1 := []string{"one", "two"}
	v2 := []string{"two", "one"}
	fmt.Println("v1 == v2: ", reflect.DeepEqual(v1, v2))	// false

	data := map[string]interface{}{
		"code":  200,
		"value": []string{"one", "two"},
	}
	encoded, _ := json.Marshal(data)
	var decoded map[string]interface{}
	json.Unmarshal(encoded, &decoded)
```

```go
    fmt.Println("data == decoded: ", reflect.DeepEqual(data, decoded))    // false
}
```

如果要大小写不敏感来比较 byte 或 string 中的英文文本，可以使用 “bytes” 或 “strings” 包的 ToUpper() 和 ToLower() 函数。比较其他语言的 byte 或 string，应使用 bytes.EqualFold() 和 strings.EqualFold()

如果 byte slice 中含有验证用户身份的数据（密文哈希、token 等），不应再使用 reflect.DeepEqual()、bytes.Equal()、 bytes.Compare()。这三个函数容易对程序造成 timing attacks，此时应使用 “crypto/subtle” 包中的 subtle.ConstantTimeCompare() 等函数

- reflect.DeepEqual() 认为空 slice 与 nil slice 并不相等，但注意 byte.Equal() 会认为二者相等：

```go
func main() {
    var b1 []byte = nil
    b2 := []byte{}

    // b1 与 b2 长度相等、有相同的字节序
    // nil 与 slice 在字节上是相同的
    fmt.Println("b1 == b2: ", bytes.Equal(b1, b2))    // true
}
```

40.从 panic 中恢复

在一个 defer 延迟执行的函数中调用 recover() ，它便能捕捉 / 中断 panic

```go
// 错误的 recover 调用示例
func main() {
    recover()    // 什么都不会捕捉
    panic("not good")    // 发生 panic，主程序退出
    recover()    // 不会被执行
    println("ok")
}

// 正确的 recover 调用示例
func main() {
    defer func() {
        fmt.Println("recovered: ", recover())
    }()
    panic("not good")
}
```

从上边可以看出，recover() 仅在 defer 执行的函数中调用才会生效。

```
// 错误的调用示例
func main() {
    defer func() {
        doRecover()
    }()
    panic("not good")
}

func doRecover() {
    fmt.Println("recobered: ", recover())
}
```

> recobered: panic: not good

41.在 range 迭代 slice、array、map 时通过更新引用来更新元素

在 range 迭代中，得到的值其实是元素的一份值拷贝，更新拷贝并不会更改原来的元素，即是拷贝的地址并不是原有元素的地址：

```
func main() {
    data := []int{1, 2, 3}
    for _, v := range data {
        v *= 10        // data 中原有元素是不会被修改的
    }
    fmt.Println("data: ", data)    // data:  [1 2 3]
}
```

如果要修改原有元素的值，应该使用索引直接访问：

```
func main() {
    data := []int{1, 2, 3}
    for i, v := range data {
        data[i] = v * 10
    }
    fmt.Println("data: ", data)    // data:  [10 20 30]
}
```

如果你的集合保存的是指向值的指针，需稍作修改。依旧需要使用索引访问元素，不过可以使用 range 出来的元素直接更新原有值：

```
func main() {
    data := []*struct{ num int }{{1}, {2}, {3},}
    for _, v := range data {
        v.num *= 10    // 直接使用指针更新
    }
    fmt.Println(data[0], data[1], data[2])    // &{10} &{20} &{30}
}
```

42.slice 中隐藏的数据

从 slice 中重新切出新 slice 时，新 slice 会引用原 slice 的底层数组。如果跳了这个坑，程序可能会分配大量的临时 slice 来指向原底层数组的部分数据，将导致难以预料的内存使用。

```
func get() []byte {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    return raw[:3]    // 重新分配容量为 10000 的 slice
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 10000 0xc420080000
}
```

可以通过拷贝临时 slice 的数据，而不是重新切片来解决：

```
func get() (res []byte) {
    raw := make([]byte, 10000)
    fmt.Println(len(raw), cap(raw), &raw[0])    // 10000 10000 0xc420080000
    res = make([]byte, 3)
    copy(res, raw[:3])
    return
}

func main() {
    data := get()
    fmt.Println(len(data), cap(data), &data[0])    // 3 3 0xc4200160b8
}
```

43.Slice 中数据的误用

举个简单例子，重写文件路径（存储在 slice 中）

分割路径来指向每个不同级的目录，修改第一个目录名再重组子目录名，创建新路径：

```
// 错误使用 slice 的拼接示例
func main() {
    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    println(sepIndex)

    dir1 := path[:sepIndex]
    dir2 := path[sepIndex+1:]
    println("dir1: ", string(dir1))        // AAAA
    println("dir2: ", string(dir2))        // BBBBBBBBB

    dir1 = append(dir1, "suffix"...)
        println("current path: ", string(path))    // AAAAsuffixBBBB

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // uffixBBBB

    println("new path: ", string(path))    // AAAAsuffix/uffixBBBB    // 错误结
果
}
```

拼接的结果不是正确的 AAAAsuffix/BBBBBBBBB，因为 dir1、 dir2 两个 slice 引用的数据都是 path 的底层数组，第 13 行修改 dir1 同时也修改了 path，也导致了 dir2 的修改

解决方法：

- 重新分配新的 slice 并拷贝你需要的数据
- 使用完整的 slice 表达式：input[low:high:max]，容量便调整为 max - low

```
// 使用 full slice expression
func main() {

    path := []byte("AAAA/BBBBBBBBB")
    sepIndex := bytes.IndexByte(path, '/') // 4
    dir1 := path[:sepIndex:sepIndex]        // 此时 cap(dir1) 指定为4， 而不是先
前的 16
    dir2 := path[sepIndex+1:]
    dir1 = append(dir1, "suffix"...)

    path = bytes.Join([][]byte{dir1, dir2}, []byte{'/'})
    println("dir1: ", string(dir1))        // AAAAsuffix
    println("dir2: ", string(dir2))        // BBBBBBBBB
```

```
	println("new path: ", string(path))	// AAAAsuffix/BBBBBBBBB
}
```

第 6 行中第三个参数是用来控制 dir1 的新容量，再往 dir1 中 append 超额元素时，将分配新的 buffer 来保存。而不是覆盖原来的 path 底层数组

44.旧 slice

当你从一个已存在的 slice 创建新 slice 时，二者的数据指向相同的底层数组。如果你的程序使用这个特性，那需要注意“旧”（stale） slice 问题。

某些情况下，向一个 slice 中追加元素而它指向的底层数组容量不足时，将会重新分配一个新数组来存储数据。而其他 slice 还指向原来的旧底层数组。

```
// 超过容量将重新分配数组来拷贝值、重新存储
func main() {
	s1 := []int{1, 2, 3}
	fmt.Println(len(s1), cap(s1), s1)	// 3 3 [1 2 3 ]

	s2 := s1[1:]
	fmt.Println(len(s2), cap(s2), s2)	// 2 2 [2 3]

	for i := range s2 {
		s2[i] += 20
	}
	// 此时的 s1 与 s2 是指向同一个底层数组的
	fmt.Println(s1)		// [1 22 23]
	fmt.Println(s2)		// [22 23]

	s2 = append(s2, 4)	// 向容量为 2 的 s2 中再追加元素，此时将分配新数组来存

	for i := range s2 {
		s2[i] += 10
	}
	fmt.Println(s1)		// [1 22 23]	// 此时的 s1 不再更新，为旧数据
	fmt.Println(s2)		// [32 33 14]
}
```

45.类型声明与方法

从一个现有的非 interface 类型创建新类型时，并不会继承原有的方法：

```
// 定义 Mutex 的自定义类型
type myMutex sync.Mutex
```

```
func main() {
    var mtx myMutex
    mtx.Lock()
    mtx.UnLock()
}
```

> mtx.Lock undefined (type myMutex has no field or method Lock)...

如果你需要使用原类型的方法，可将原类型以匿名字段的形式嵌到你定义的新 struct 中：

```
// 类型以字段形式直接嵌入
type myLocker struct {
    sync.Mutex
}

func main() {
    var locker myLocker
    locker.Lock()
    locker.Unlock()
}
```

interface 类型声明也保留它的方法集：

```
type myLocker sync.Locker

func main() {
    var locker myLocker
    locker.Lock()
    locker.Unlock()
}
```

46.跳出 for-switch 和 for-select 代码块

没有指定标签的 break 只会跳出 switch/select 语句，若不能使用 return 语句跳出的话，可为 break 跳出标签指定的代码块：

```
// break 配合 label 跳出指定代码块
func main() {
loop:
    for {
        switch {
        case true:
```

```
            fmt.Println("breaking out...")
            //break    // 死循环，一直打印 breaking out...
            break loop
        }
    }
    fmt.Println("out...")
}
```

goto 虽然也能跳转到指定位置，但依旧会再次进入 for-switch，死循环。

47.for 语句中的迭代变量与闭包函数

for 语句中的迭代变量在每次迭代中都会重用，即 for 中创建的闭包函数接收到的参数始终是同一个变量，在 goroutine 开始执行时都会得到同一个迭代值：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        go func() {
            fmt.Println(v)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 three three three
}
```

最简单的解决方法：无需修改 goroutine 函数，在 for 内部使用局部变量保存迭代值，再传参：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        vCopy := v
        go func() {
            fmt.Println(vCopy)
        }()
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

另一个解决方法：直接将当前的迭代值以参数形式传递给匿名函数：

```
func main() {
    data := []string{"one", "two", "three"}

    for _, v := range data {
        go func(in string) {
            fmt.Println(in)
        }(v)
    }

    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

注意下边这个稍复杂的 3 个示例区别：

```
type field struct {
    name string
}

func (p *field) print() {
    fmt.Println(p.name)
}

// 错误示例
func main() {
    data := []field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 three three three
}


// 正确示例
func main() {
    data := []field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {
        v := v
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 one two three
```

```
}

// 正确示例
func main() {
    data := []*field{{"one"}, {"two"}, {"three"}}
    for _, v := range data {    // 此时迭代值 v 是三个元素值的地址，每次 v 指向的值不同
        go v.print()
    }
    time.Sleep(3 * time.Second)
    // 输出 one two three
}
```

48.defer 函数的参数值

对 defer 延迟执行的函数，它的参数会在声明时候就会求出具体值，而不是在执行时才求值：

```
// 在 defer 函数中参数会提前求值
func main() {
    var i = 1
    defer fmt.Println("result: ", func() int { return i * 2 }())
    i++
}
```

> result: 2

49.defer 函数的执行时机

对 defer 延迟执行的函数，会在调用它的函数结束时执行，而不是在调用它的语句块结束时执行，注意区分开。

比如在一个长时间执行的函数里，内部 for 循环中使用 defer 来清理每次迭代产生的资源调用，就会出现问题：

```
// www.topgoer.com go语言中文文档
// 命令行参数指定目录名
// 遍历读取目录下的文件
func main() {

    if len(os.Args) != 2 {
        os.Exit(1)
    }

    dir := os.Args[1]
```

```
    start, err := os.Stat(dir)
    if err != nil || !start.IsDir() {
        os.Exit(2)
    }

    var targets []string
    filepath.Walk(dir, func(fPath string, fInfo os.FileInfo, err error) error {
        if err != nil {
            return err
        }

        if !fInfo.Mode().IsRegular() {
            return nil
        }

        targets = append(targets, fPath)
        return nil
    })

    for _, target := range targets {
        f, err := os.Open(target)
        if err != nil {
            fmt.Println("bad target:", target, "error:", err)    //error:too man
y open files
            break
        }
        defer f.Close()    // 在每次 for 语句块结束时，不会关闭文件资源

        // 使用 f 资源
    }
}
```

先创建 10000 个文件：

```
#!/bin/bash
for n in {1..10000}; do
    echo content > "file${n}.txt"
done
```

运行效果：

```
➜ blog go run main.go /Users/wuyin/Desktop/files
bad target: /Users/wuyin/Desktop/files/file5371.txt error:
open /Users/wuyin/Desktop/files/file5371.txt: too many open
 files
➜ blog _
```

解决办法：defer 延迟执行的函数写入匿名函数中：

```
// 目录遍历正常
func main() {
    // ...

    for _, target := range targets {
        func() {
            f, err := os.Open(target)
            if err != nil {
                fmt.Println("bad target:", target, "error:", err)
                return    // 在匿名函数内使用 return 代替 break 即可
            }
            defer f.Close()    // 匿名函数执行结束，调用关闭文件资源

            // 使用 f 资源
        }()
    }
}
```

当然你也可以去掉 defer，在文件资源使用完毕后，直接调用 f.Close() 来关闭。

50.失败的类型断言

在类型断言语句中，断言失败则会返回目标类型的“零值”，断言变量与原来变量混用可能出现异常情况：

```
// 错误示例
func main() {
    var data interface{} = "great"

    // data 混用
    if data, ok := data.(int); ok {
        fmt.Println("[is an int], data: ", data)
    } else {
        fmt.Println("[not an int], data: ", data)    // [isn't a int], data:  0
    }
```

```
}


// 正确示例
func main() {
    var data interface{} = "great"

    if res, ok := data.(int); ok {
        fmt.Println("[is an int], data: ", res)
    } else {
        fmt.Println("[not an int], data: ", data)    // [not an int], data:  gre
at
    }
}
```

51.阻塞的 gorutinue 与资源泄露

在 2012 年 Google I/O 大会上，Rob Pike 的 Go Concurrency Patterns 演讲讨论 Go 的几种基本并发模式，如 完整代码 中从数据集中获取第一条数据的函数：

```
func First(query string, replicas []Search) Result {
    c := make(chan Result)
    replicaSearch := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
        go replicaSearch(i)
    }
    return <-c
}
```

在搜索重复时依旧每次都起一个 goroutine 去处理，每个 goroutine 都把它的搜索结果发送到结果 channel 中，channel 中收到的第一条数据会直接返回。

返回完第一条数据后，其他 goroutine 的搜索结果怎么处理？他们自己的协程如何处理？

在 First() 中的结果 channel 是无缓冲的，这意味着只有第一个 goroutine 能返回，由于没有 receiver，其他的 goroutine 会在发送上一直阻塞。如果你大量调用，则可能造成资源泄露。

为避免泄露，你应该确保所有的 goroutine 都能正确退出，有 2 个解决方法：

- 使用带缓冲的 channel，确保能接收全部 goroutine 的返回结果：

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,len(replicas))
    searchReplica := func(i int) { c <- replicas[i](query) }
    for i := range replicas {
```

```
        go searchReplica(i)
    }
    return <-c
}
```

- 使用 select 语句，配合能保存一个缓冲值的 channel default 语句：
  default 的缓冲 channel 保证了即使结果 channel 收不到数据，也不会阻塞 goroutine

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result,1)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        default:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }
    return <-c
}
```

- 使用特殊的废弃（cancellation） channel 来中断剩余 goroutine 的执行：

```
func First(query string, replicas ...Search) Result {
    c := make(chan Result)
    done := make(chan struct{})
    defer close(done)
    searchReplica := func(i int) {
        select {
        case c <- replicas[i](query):
        case <- done:
        }
    }
    for i := range replicas {
        go searchReplica(i)
    }

    return <-c
}
```

Rob Pike 为了简化演示，没有提及演讲代码中存在的这些问题。不过对于新手来说，可能会不加思考直接使用。

高级篇：52-58

52.使用指针作为方法的 receiver

只要值是可寻址的，就可以在值上直接调用指针方法。即是对一个方法，它的 receiver 是指针就足矣。

但不是所有值都是可寻址的，比如 map 类型的元素、通过 interface 引用的变量：

```
type data struct {
    name string
}

type printer interface {
    print()
}

func (p *data) print() {
    fmt.Println("name: ", p.name)
}

func main() {
    d1 := data{"one"}
    d1.print()    // d1 变量可寻址，可直接调用指针 receiver 的方法

    var in printer = data{"two"}
    in.print()    // 类型不匹配

    m := map[string]data{
        "x": data{"three"},
    }
    m["x"].print()    // m["x"] 是不可寻址的    // 变动频繁
}
```

> cannot use data literal (type data) as type printer in assignment:
> data does not implement printer (print method has pointer receiver)

> cannot call pointer method on m["x"]
> cannot take the address of m["x"]

53.更新 map 字段的值

如果 map 一个字段的值是 struct 类型，则无法直接更新该 struct 的单个字段：

```go
// 无法直接更新 struct 的字段值
type data struct {
    name string
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    m["x"].name = "Jerry"
}
```

> cannot assign to struct field m["x"].name in map

因为 map 中的元素是不可寻址的。需区分开的是，slice 的元素可寻址：

```go
type data struct {
    name string
}

func main() {
    s := []data{{"Tom"}}
    s[0].name = "Jerry"
    fmt.Println(s)    // [{Jerry}]
}
```

注意：不久前 gccgo 编译器可更新 map struct 元素的字段值，不过很快便修复了，官方认为是 Go1.3 的潜在特性，无需及时实现，依旧在 todo list 中。

更新 map 中 struct 元素的字段值，有 2 个方法：

- 使用局部变量

```go
// 提取整个 struct 到局部变量中，修改字段值后再整个赋值
type data struct {
    name string
}

func main() {
    m := map[string]data{
        "x": {"Tom"},
    }
    r := m["x"]
    r.name = "Jerry"
```

```
    m["x"] = r
    fmt.Println(m)      // map[x:{Jerry}]
}
```

- 使用指向元素的 map 指针

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }

    m["x"].name = "Jerry"     // 直接修改 m["x"] 中的字段
    fmt.Println(m["x"])     // &{Jerry}
}
```

但是要注意下边这种误用：

```
func main() {
    m := map[string]*data{
        "x": {"Tom"},
    }
    m["z"].name = "what???"
    fmt.Println(m["x"])
}
```

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

54.nil interface 和 nil interface 值

虽然 interface 看起来像指针类型，但它不是。interface 类型的变量只有在类型和值均为 nil 时才为 nil

如果你的 interface 变量的值是跟随其他变量变化的（雾），与 nil 比较相等时小心：

```
func main() {
    var data *byte
    var in interface{}

    fmt.Println(data, data == nil)    // <nil> true
    fmt.Println(in, in == nil)    // <nil> true

    in = data
    fmt.Println(in, in == nil)    // <nil> false    // data 值为 nil，但 in 值不
```

```
为 nil
}
```

如果你的函数返回值类型是 interface，更要小心这个坑：

```
// 错误示例
func main() {
    doIt := func(arg int) interface{} {
        var result *struct{} = nil
        if arg > 0 {
            result = &struct{}{}
        }
        return result
    }

    if res := doIt(-1); res != nil {
        fmt.Println("Good result: ", res)    // Good result:  <nil>
        fmt.Printf("%T\n", res)            // *struct {}    // res 不是 nil，它
的值为 nil
        fmt.Printf("%v\n", res)            // <nil>
    }
}


// 正确示例
func main() {
    doIt := func(arg int) interface{} {
        var result *struct{} = nil
        if arg > 0 {
            result = &struct{}{}
        } else {
            return nil    // 明确指明返回 nil
        }
        return result
    }

    if res := doIt(-1); res != nil {
        fmt.Println("Good result: ", res)
    } else {
        fmt.Println("Bad result: ", res)    // Bad result:  <nil>
    }
}
```

55.堆栈变量

你并不总是清楚你的变量是分配到了堆还是栈。

在 C++ 中使用 new 创建的变量总是分配到堆内存上的，但在 Go 中即使使用 new()、make() 来创建变量，变量为内存分配位置依旧归 Go 编译器管。

Go 编译器会根据变量的大小及其 “escape analysis” 的结果来决定变量的存储位置，故能准确返回本地变量的地址，这在 C/C++ 中是不行的。

在 go build 或 go run 时，加入 -m 参数，能准确分析程序的变量分配位置：

```
➜ blog go run -gcflags -m main.go
# command-line-arguments
./main.go:6:10: can inline main.func1
./main.go:17:15: "Good result: " escapes to heap
./main.go:6:10: func literal escapes to heap
./main.go:19:15: "Bad result: " escapes to heap
./main.go:13:3: result escapes to heap
./main.go:9:23: &struct {} literal escapes to heap
```

56.GOMAXPROCS、Concurrency（并发）and Parallelism（并行）

Go 1.4 及以下版本，程序只会使用 1 个执行上下文 / OS 线程，即任何时间都最多只有 1 个 goroutine 在执行。

Go 1.5 版本将可执行上下文的数量设置为 runtime.NumCPU() 返回的逻辑 CPU 核心数，这个数与系统实际总的 CPU 逻辑核心数是否一致，取决于你的 CPU 分配给程序的核心数，可以使用 GOMAXPROCS 环境变量或者动态的使用 runtime.GOMAXPROCS() 来调整。

误区：GOMAXPROCS 表示执行 goroutine 的 CPU 核心数，参考文档

GOMAXPROCS 的值是可以超过 CPU 的实际数量的，在 1.5 中最大为 256

```go
func main() {
    fmt.Println(runtime.GOMAXPROCS(-1))    // 4
    fmt.Println(runtime.NumCPU())    // 4
    runtime.GOMAXPROCS(20)
    fmt.Println(runtime.GOMAXPROCS(-1))    // 20
    runtime.GOMAXPROCS(300)
    fmt.Println(runtime.GOMAXPROCS(-1))    // Go 1.9.2 // 300
}
```

57.读写操作的重新排序

Go 可能会重排一些操作的执行顺序，可以保证在一个 goroutine 中操作是顺序执行的，但不保证多 goroutine 的执行顺序：

```
var _ = runtime.GOMAXPROCS(3)

var a, b int

func u1() {
    a = 1
    b = 2
}

func u2() {
    a = 3
    b = 4
}

func p() {
    println(a)
    println(b)
}

func main() {
    go u1()    // 多个 goroutine 的执行顺序不定
    go u2()
    go p()
    time.Sleep(1 * time.Second)
}
```

运行效果：

```
➜  blog go run main.go
1
2
➜  blog go run main.go
3
4
➜  blog go run main.go
1
2
```

如果你想保持多 goroutine 像代码中的那样顺序执行，可以使用 channel 或 sync 包中的锁机制等。

58.优先调度

你的程序可能出现一个 goroutine 在运行时阻止了其他 goroutine 的运行，比如程序中有一个不让调度器运行的 for 循环：

```
func main() {
    done := false

    go func() {
        done = true
    }()

    for !done {
    }

    println("done !")
}
```

for 的循环体不必为空，但如果代码不会触发调度器执行，将出现问题。

调度器会在 GC、Go 声明、阻塞 channel、阻塞系统调用和锁操作后再执行，也会在非内联函数调用时执行：

```
func main() {
    done := false

    go func() {
        done = true
    }()

    for !done {
        println("not done !")    // 并不内联执行
    }

    println("done !")
}
```

可以添加 -m 参数来分析 for 代码块中调用的内联函数：

```
➜ blog go run -gcflags -m main.go
# command-line-arguments
./main.go:6:5: can inline main.func1
./main.go:6:5: func literal escapes to heap
./main.go:6:5: func literal escapes to heap
./main.go:7:3: &done escapes to heap
./main.go:4:10: moved to heap: done
not done !
not done !
not done !
done !
➜ blog
```

你也可以使用 runtime 包中的 Gosched() 来 手动启动调度器：

```
func main() {
    done := false

    go func() {
        done = true
    }()

    for !done {
        runtime.Gosched()
    }

    println("done !")
}
```

转自：http://devs.cloudimmunity.com/gotchas-and-common-mistakes-in-go-golang/index.html#string_byte_slice_conv

资料下载

链接：https://pan.baidu.com/s/1s2_CsfkvIrG96WR3dSJo4w

提取码：ejs1

零碎知识点

1.new() 与 make() 的区别

new(T) 和 make(T,args) 是 Go 语言内建函数，用来分配内存，但适用的类型不同。

new(T) 会为 T 类型的新值分配已置零的内存空间，并返回地址（指针），即类型为 `*T` 的值。换句话说就是，返回一个指针，该指针指向新分配的、类型为 T 的零值。适用于值类型，如数组、结构体等。

make(T,args) 返回初始化之后的 T 类型的值，这个值并不是 T 类型的零值，也不是指针 `*T` ，是经过初始化之后的 T 的引用。make() 只适用于 slice、map 和 channel.

2. `defer func() { recover() }()` 可以拦截panic错误

3.转换小写字母

```
//转换成小写字母
func toLowerCase(str string) string {
    rune_arr := []rune(str)
    for i, _ := range rune_arr {
        if rune_arr[i] >= 65 && rune_arr[i] <= 90 {
            rune_arr[i] += 32
        }
    }
    return string(rune_arr)
}
```

4.Golang 解决 golang.org/x/ 下包下载不下来的问题

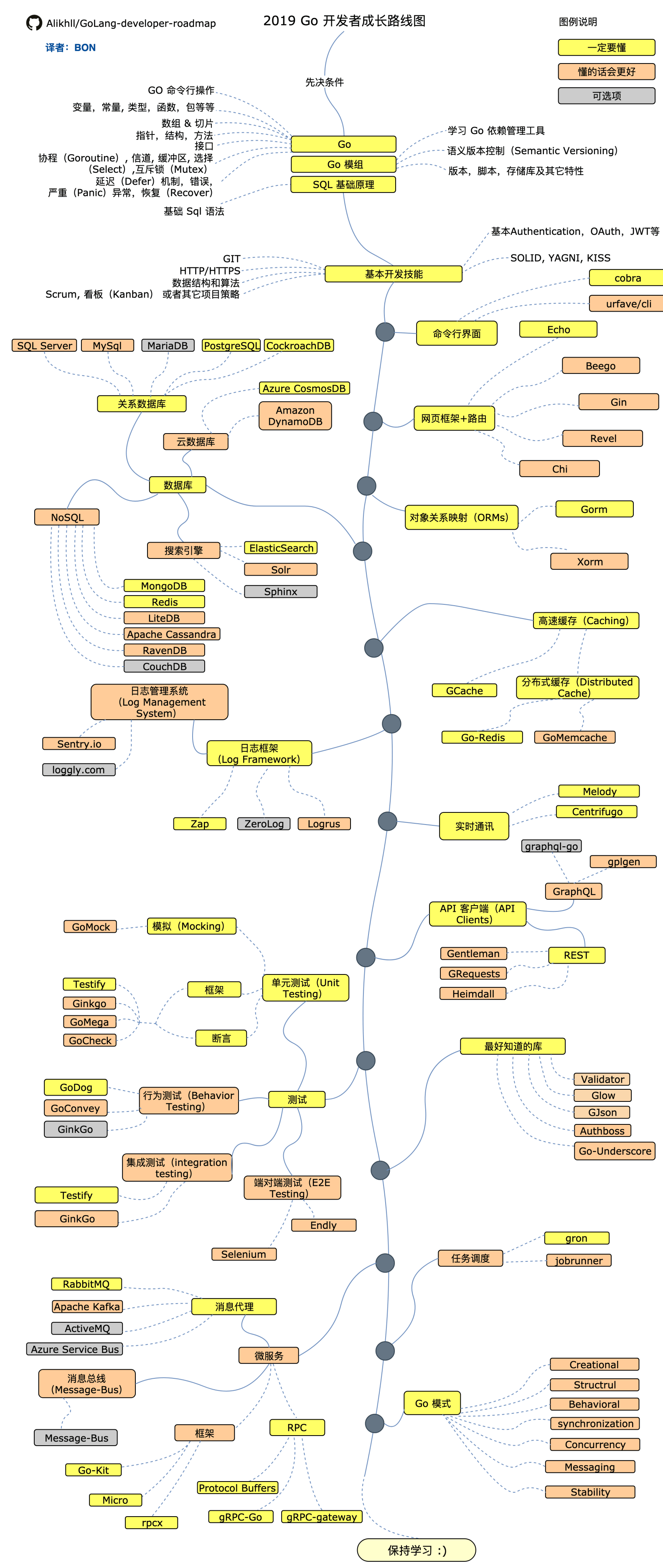
由于众所周知的原因，golang在下载golang.org的包时会出现访问不了的情况。尤其是x包，很多库都依赖于它。由于x包在github上都有镜像，我们可以使用从github.com上把代码clone到创建的golang。org/x目录上就OK了

- git clone https://github.com/golang/sys.git
- git clone https://github.com/golang/net.git
- git clone https://github.com/golang/text.git
- git clone https://github.com/golang/lint.git

- git clone https://github.com/golang/tools.git
- git clone https://github.com/golang/crypto.git

还有其他的包可以直接进github包里面查找

学习路线图



Alikhll/GoLang-developer-roadmap
译者：BON
2019 Go 开发者成长路线图
图例说明
一定要懂
懂的话会更好
可选项
先决条件
GO 命令行操作
变量，常量，类型，函数，包等等
数组 & 切片
指针，结构，方法
接口
协程（Goroutine），信道，缓冲区，选择（Select），互斥锁（Mutex）
延迟（Defer）机制，错误，严重（Panic）异常，恢复（Recover）
基础 Sql 语法
Go
Go 模组
SQL 基础原理
学习 Go 依赖管理工具
语义版本控制（Semantic Versioning）
版本，脚本，存储库及其它特性
基本Authentication，OAuth，JWT等
SOLID, YAGNI, KISS
GIT
HTTP/HTTPS
数据结构和算法
Scrum，看板（Kanban）或者其它项目策略
基本开发技能
cobra
urfave/cli
命令行界面
Echo
Beego
Gin
Revel
Chi
网页框架+路由
SQL Server
MySql
MariaDB
PostgreSQL
CockroachDB
关系数据库
Azure CosmosDB
Amazon DynamoDB
云数据库
数据库
NoSQL
搜索引擎
ElasticSearch
Solr
Sphinx
MongoDB
Redis
LiteDB
Apache Cassandra
RavenDB
CouchDB
对象关系映射（ORMs）
Gorm
Xorm
高速缓存（Caching）
GCache
分布式缓存（Distributed Cache）
Go-Redis
GoMemcache
日志管理系统（Log Management System）
Sentry.io
loggly.com
日志框架（Log Framework）
Zap
ZeroLog
Logrus
实时通讯
Melody
Centrifugo
graphql-go
gplgen
GraphQL
API 客户端（API Clients）
Gentleman
GRequests
Heimdall
REST
GoMock
模拟（Mocking）
单元测试（Unit Testing）
框架
断言
Testify
Ginkgo
GoMega
GoCheck
GoDog
GoConvey
GinkGo
行为测试（Behavior Testing）
测试
集成测试（integration testing）
端对端测试（E2E Testing）
Testify
GinkGo
Endly
Selenium
最好知道的库
Validator
Glow
GJson
Authboss
Go-Underscore
任务调度
gron
jobrunner
RabbitMQ
Apache Kafka
ActiveMQ
Azure Service Bus
消息代理
微服务
消息总线（Message-Bus）
Message-Bus
框架
RPC
Go-Kit
Micro
rpcx
Protocol Buffers
gRPC-Go
gRPC-gateway
Go 模式
Creational
Structrul
Behavioral
synchronization
Concurrency
Messaging
Stability
保持学习 :)

面试题

请跳转

http://www.topgoer.cn/docs/gomianshiti/mianshiti

go中文标准文档

请跳转 http://word.topgoer.com

关于

我是谁

枯藤 一个苦逼的程序员单纯的喜欢代码

联系我

- 欢迎有好文章的小伙伴加我分享文章
- 欢迎找到博客错误的地方加我
- 欢迎学习交流的小伙伴加我
- 微信：yzy85215215

优秀网站

- go语言中文网
- Go编程时光
- 全栈直通车
- 友情链接以及优秀网站推荐请联系我

版权

- 文章部分内容来自优秀的go语言讲师李文周老师 博客地址: www.liwenzhou.com