目　录

Go 的 fake-useragent 了解一下
用 Go 来了解一下 Redis 通讯协议
使用 Gomock 进行单元测试
在 Go 中恰到好处的内存对齐
来，控制一下 goroutine 的并发数量
for-loop 与 json.Unmarshal 性能分析概要
简单围观一下有趣的 //go: 指令
我要在栈上。不，你应该在堆上
Go1.12 defer 会有性能损耗，尽量不要用？
从实践到原理，带你参透 gRPC
Go1.13 defer 的性能是如何提高的
Go 应用内存占用太多，让排查？（VSZ篇）

介绍

煎鱼的迷之博客

写写代码，喝喝茶，搞搞 Go，一起吧，这是我的项目地址：
https://github.com/eddycjy/blog

在线阅读

- https://eddycjy.com/

我的公众号

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见

如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进

HTTP 应用

Go 介绍与环境安装

准备环节

安装 Go

Centos

首先，根据对应的操作系统选择安装包 下载，在这里我使用的是 Centos 64 位系统，如下：

```
$ wget https://studygolang.com/dl/golang/go1.13.1.linux-amd64.tar.gz

$ tar -zxvf go1.13.1.linux-amd64.tar.gz

$ mv go/ /usr/local/
```

配置 /etc/profile

```
vi /etc/profile
```

添加环境变量 GOROOT 和将 GOBIN 添加到 PATH 中

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

配置完毕后，执行命令令其生效

```
source /etc/profile
```

在控制台输入 `go version` ，若输出版本号则**安装成功**，如下：

```
$ go version
go version go1.13.1 linux/amd64
```

MacOS

在 MacOS 上安装 Go 最方便的办法就是使用 brew，安装如下：

```
$ brew install go
```

升级命令如下：

```
$ brew upgrade go
```

注：升级命令你不需要执行，但我想未来你有一天会用到的。

同样在控制台输入 `go version` ，若输出版本号则**安装成功**。

了解 Go

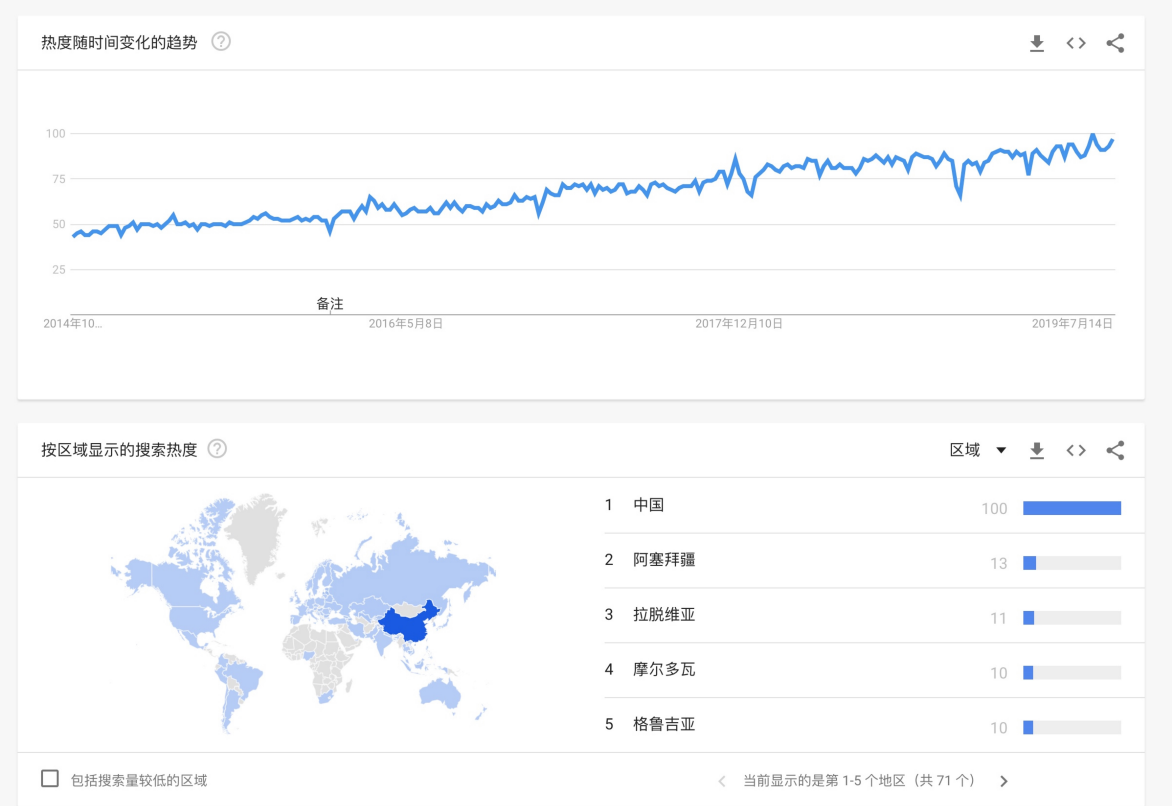
是什么

> Go is an open source programming language that makes it easy to build simple, reliable, and efficient software.

上述为官方说明，如果简单来讲，大致为如下几点：

- Go 是编程语言。
- 谷歌爸爸撑腰。
- 语言级高并发。
- 上手快，入门简单。
- 简洁，很有特色。
- 国内使用人群逐年增多。

谁在用



有什么

那么大家会有些疑问，纠结 `Go` 本身有什么东西，我们刚刚设置的环境变量又有什么用呢，甚至作为一名老粉，你会纠结 GOPATH 去哪里了，我们一起接着往下看。

目录结构

首先，我们在解压的时候会得到一个名为 `go` 的文件夹，其中包括了所有 `Go` 语言相关的一些文件，如下：

```
$ tree -L 1 go
go
├── api
├── bin
├── doc
├── lib
├── misc
├── pkg
├── src
├── test
└── ...
```

在这之中包含了很多文件夹和文件，我们来简单说明其中主要文件夹的作用：

- api：用于存放依照 `Go` 版本顺序的 API 增量列表文件。这里所说的 API 包含公开的变量、常量、函数等。这些 API 增量列表文件用于 `Go` 语言 API 检查
- bin：用于存放主要的标准命令文件（可执行文件），包含 `go`、`godoc`、`gofmt`
- blog：用于存放官方博客中的所有文章
- doc：用于存放标准库的 HTML 格式的程序文档。我们可以通过 `godoc` 命令启动一个 Web 程序展示这些文档
- lib：用于存放一些特殊的库文件
- misc：用于存放一些辅助类的说明和工具
- pkg：用于存放安装 `Go` 标准库后的所有归档文件（以 `.a` 结尾的文件）。注意，你会发现其中有名称为 `linux_amd64` 的文件夹，我们称为平台相关目录。这类文件夹的名称由对应的操作系统和计算架构的名称组合而成。通过 `go install` 命令，`Go` 程序会被编译成平台相关的归档文件存放到其中
- src：用于存放 `Go` 自身、`Go` 标准工具以及标准库的所有源码文件
- test：存放用来测试和验证 `Go` 本身的所有相关文件

环境变量

你可能会疑惑刚刚设置的环境变量是什么，如下：

- GOROOT：`Go` 的根目录。
- PATH 下增加 `$GOROOT/bin`：`Go` 的 `bin` 下会存放可执行文件，我们把他加入 `$PATH` 后，未来拉下来并编译后的二进制文件就可以直接在命令行使用。

那在什么东西都不下载的情况下，`$GOBIN` 下面有什么呢，如下：

```
bin/ $ls
go  gofmt
```

- go：`Go` 二进制本身。
- gofmt：代码格式化工具。

因此我们刚刚把 `$GOBIN` 加入到 `$PATH` 后，你执行 `go version` 命令后就可以查看到对应的输出结果。

注：MacOS 用 brew 安装的话就不需要。

放在哪

你现在知道 Go 是什么了，也知道 Go 的源码摆在哪了，你肯定会想，那我应用代码放哪呢，答案是在 **Go1.11+ 和开启 Go Modules 的情况下摆哪都行**。

了解 Go Modules

了解历史

在过去，Go 的依赖包管理在工具上混乱且不统一，有 dep，有 glide，有 govendor...甚至还有因为外网的问题，频频导致拉不下来包，很多人苦不堪言，盼着官方给出一个大一统做出表率。

而在 Go modules 正式出来之前还有一个叫 dep 的项目，我们在上面有提到，它是 Go 的一个官方实验性项目，目的也是为了解决 Go 在依赖管理方面的问题，当时社区里面几乎所有的人都认为 dep 肯定就是未来 Go 官方的依赖管理解决方案了。

但是万万没想到，半路杀出个程咬金，Russ Cox 义无反顾地推出了 Go modules，这瞬间导致一石激起千层浪，让社区炸了锅。大家一致认为 Go team 实在是太霸道、太独裁了，连个招呼都不打一声。我记得当时有很多人在网上跟 Russ Cox 口水战，各种依赖管理解决方案的专家都冒出来发表意见，讨论范围甚至一度超出了 Go 语言的圈子触及到了其他语言的领域。

当然，最后，推成功了，Go modules 已经进入官方工具链中，与 Go 深深结合，以前常说的 GOPATH 终将会失去它原有的作用，而且它还提供了 GOPROXY 间接解决了国内访问外网的问题。

了解 Russ Cox

在上文中提到的 Russ Cox 是谁呢，他是 Go 这个项目目前代码提交量最多的人，甚至是第二名的两倍还要多（从 2019 年 09 月 30 日前来看）。

Russ Cox 还是 Go 现在的掌舵人（大家应该知道之前 Go 的掌舵人是 Rob Pike，但是听说由于他本人不喜欢特朗普执政所以离开了美国，然后他岁数也挺大的了，所以也正在逐渐交权，不过现在还是在参与 Go 的发展）。

Russ Cox 的个人能力相当强，看问题的角度也很独特，这也就是为什么他刚一提出 Go modules 的概念就能引起那么大范围的响应。虽然是被强推的，但事实也证明当下的 Go modules 表现得确实很优秀，所以这表明一定程度上的“独裁”还是可以接受的，至少可以保证一个项目能更加专一地朝着一个方向发展。

初始化行为

在前面我们已经了解到 Go 依赖包管理的历史情况，接下来我们将正式的进入使用，首先你需要有一个你喜欢的目录，例如：`$ mkdir ~/go-application && cd ~/go-application`，然后执行如下命令：

```
$ mkdir go-gin-example && cd go-gin-example

$ go env -w GO111MODULE=on

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

$ go mod init github.com/EDDYCJY/go-gin-example
go: creating new go.mod: module github.com/EDDYCJY/go-gin-example

$ ls
go.mod
```

- `mkdir xxx && cd xxx`：创建并切换到项目目录里去。
- `go env -w GO111MODULE=on`：打开 Go modules 开关（目前在 Go1.13 中默认值为 `auto`）。
- `go env -w GOPROXY=...`：设置 GOPROXY 代理，这里主要涉及到两个值，第一个是 `https://goproxy.cn`，它是由七牛云背书的一个强大稳定的 Go 模块代理，可以有效地解决你的外网问题；第二个是 `direct`，它是一个特殊的 fallback 选项，它的作用是用于指示 Go 在拉取模块时遇到错误会回源到模块版本的源地址去抓取（比如 GitHub 等）。
- `go mod init [MODULE_PATH]`：初始化 Go modules，它将会生成 go.mod 文件，需要注意的是 `MODULE_PATH` 填写的是模块引入路径，你可以根据自己的情况修改路径。

在执行了上述步骤后，初始化工作已完成，我们打开 `go.mod` 文件看看，如下：

```
module github.com/EDDYCJY/go-gin-example

go 1.13
```

默认的 `go.mod` 文件里主要是两块内容，一个是当前的模块路径和预期的 Go 语言版本。

基础使用

- 用 `go get` 拉取新的依赖
  - 拉取最新的版本(优先择取 tag)：`go get golang.org/x/text@latest`
  - 拉取 `master` 分支的最新 commit：`go get golang.org/x/text@master`
  - 拉取 tag 为 v0.3.2 的 commit：`go get golang.org/x/text@v0.3.2`

    - 拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2：`go get golang.org/x/text@342b2e`
    - 用 `go get -u` 更新现有的依赖
    - 用 `go mod download` 下载 go.mod 文件中指明的所有依赖
    - 用 `go mod tidy` 整理现有的依赖
    - 用 `go mod graph` 查看现有的依赖结构
    - 用 `go mod init` 生成 go.mod 文件 (Go 1.13 中唯一一个可以生成 go.mod 文件的子命令)
- 用 `go mod edit` 编辑 go.mod 文件
- 用 `go mod vendor` 导出现有的所有依赖 (事实上 Go modules 正在淡化 Vendor 的概念)
- 用 `go mod verify` 校验一个模块是否被篡改过

这一小节主要是针对 Go modules 的基础使用讲解，还没具体的使用，是希望你能够留个印象，因为在后面章节会不断夹杂 Go modules 的知识点。

注：建议阅读官方文档 wiki/Modules。

开始 Gin 之旅

是什么

> Gin is a HTTP web framework written in Go (Golang). It features a Martini-like API with much better performance – up to 40 times faster. If you need smashing performance, get yourself some Gin.

Gin 是用 Go 开发的一个微框架，类似 Martinier 的 API，重点是小巧、易用、性能好很多，也因为 httprouter 的性能提高了 40 倍。

安装

我们回到刚刚创建的 `go-gin-example` 目录下，在命令行下执行如下命令：

```
$ go get -u github.com/gin-gonic/gin
go: downloading golang.org/x/sys v0.0.0-20190222072716-a9d3bda3a223
go: extracting golang.org/x/sys v0.0.0-20190222072716-a9d3bda3a223
go: finding github.com/gin-contrib/sse v0.1.0
go: finding github.com/ugorji/go v1.1.7
go: finding gopkg.in/yaml.v2 v2.2.3
```

```
go: finding golang.org/x/sys latest
go: finding github.com/mattn/go-isatty v0.0.9
go: finding github.com/modern-go/concurrent latest
...
```

go.sum

这时候你再检查一下该目录下，会发现多个了个 `go.sum` 文件，如下：

```
github.com/davecgh/go-spew v1.1.0/go.mod h1:J7Y8YcW...
github.com/davecgh/go-spew v1.1.1/go.mod h1:J7Y8YcW...
github.com/gin-contrib/sse v0.0.0-20190301062529-5545eab6dad3 h1:t8FVkw33L+wilf2
QiWkwOUV77qRpcH/JHPKGpKa2E8g=
github.com/gin-contrib/sse v0.0.0-20190301062529-5545eab6dad3/go.mod h1:VJ0WA
2...
github.com/gin-contrib/sse v0.1.0 h1:Y/yl/+YNO...
...
```

`go.sum` 文件详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

go.mod

既然我们下载了依赖包，`go.mod` 文件会不会有所改变呢，我们再去看看，如下：

```
module github.com/EDDYCJY/go-gin-example

go 1.13

require (
        github.com/gin-contrib/sse v0.1.0 // indirect
        github.com/gin-gonic/gin v1.4.0 // indirect
        github.com/golang/protobuf v1.3.2 // indirect
        github.com/json-iterator/go v1.1.7 // indirect
        github.com/mattn/go-isatty v0.0.9 // indirect
        github.com/ugorji/go v1.1.7 // indirect
        golang.org/x/sys v0.0.0-20190927073244-c990c680b611 // indirect
        gopkg.in/yaml.v2 v2.2.3 // indirect
)
```

确确实实发生了改变，那多出来的东西又是什么呢，`go.mod` 文件又保存了什么信息呢，实际上 `go.mod` 文件是启用了 Go modules 的项目所必须的最重要的文件，因为它描述了当前项目（也就是当前模块）的元信息，每一行都以一个动词开头，目前有以下 5 个动词:

- module：用于定义当前项目的模块路径。
- go：用于设置预期的 Go 版本。
- require：用于设置一个特定的模块版本。
- exclude：用于从使用中排除一个特定的模块版本。
- replace：用于将一个模块版本替换为另外一个模块版本。

你可能还会疑惑 `indirect` 是什么东西， `indirect` 的意思是传递依赖，也就是非直接依赖。

测试

编写一个 `test.go` 文件

```
package main

import "github.com/gin-gonic/gin"

func main() {
  r := gin.Default()
  r.GET("/ping", func(c *gin.Context) {
    c.JSON(200, gin.H{
      "message": "pong",
    })
  })
  r.Run() // listen and serve on 0.0.0.0:8080
}
```

执行 `test.go`

```
$ go run test.go
...
[GIN-debug] GET    /ping                     --> main.main.func1 (3 handlers)
[GIN-debug] Environment variable PORT is undefined. Using port :8080 by default
[GIN-debug] Listening and serving HTTP on :8080
```

访问 `$HOST:8080/ping` ，若返回 `{"message":"pong"}` 则正确

```
curl 127.0.0.1:8080/ping
```

至此，我们的环境安装和初步运行都基本完成了。

再想一想

刚刚在执行了命令 `$ go get -u github.com/gin-gonic/gin` 后，我们查看了 `go.mod` 文件，如下：

```
...
require (
        github.com/gin-contrib/sse v0.1.0 // indirect
        github.com/gin-gonic/gin v1.4.0 // indirect
        ...
)
```

你会发现 `go.mod` 里的 `github.com/gin-gonic/gin` 是 `indirect` 模式，这显然不对啊，因为我们的应用程序已经实际的编写了 gin server 代码了，我就想把它调对，怎么办呢，在应用根目录下执行如下命令：

```
$ go mod tidy
```

该命令主要的作用是整理现有的依赖，非常的常用，执行后 `go.mod` 文件内容为：

```
...
require (
        github.com/gin-contrib/sse v0.1.0 // indirect
        github.com/gin-gonic/gin v1.4.0
        ...
)
```

可以看到 `github.com/gin-gonic/gin` 已经变成了直接依赖，调整完毕。

参考

本系列示例代码

- go-gin-example

相关文档

- Gin
- Gin Web Framework
- 干货满满的 Go Modules 和 goproxy.cn

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

Gin搭建Blog API's

思考

首先，在一个初始项目开始前，大家都要思考一下

- 程序的文本配置写在代码中，好吗？
- API 的错误码硬编码在程序中，合适吗？
- db 句柄谁都去 `Open` ，没有统一管理，好吗？
- 获取分页等公共参数，谁都自己写一套逻辑，好吗？

显然在较正规的项目中，这些问题的答案都是**不可以**，为了解决这些问题，我们挑选一款读写配置文件的库，目前比较火的有 viper，有兴趣你未来可以简单了解一下，没兴趣的话等以后接触到再说。

但是本系列选用 go-ini/ini ，它的 中文文档。大家是必须需要要简单阅读它的文档，再接着完成后面的内容。

本文目标

- 编写一个简单的 API 错误码包。
- 完成一个 Demo 示例。
- 讲解 Demo 所涉及的知识点。

介绍和初始化项目

初始化项目目录

在前一章节中，我们初始化了一个 `go-gin-example` 项目，接下来我们需要继续新增如下目录结构：

```
go-gin-example/
├─── conf
├─── middleware
├─── models
├─── pkg
```

```
├─── routers
└─── runtime
```

- conf：用于存储配置文件
- middleware：应用中间件
- models：应用数据库模型
- pkg：第三方包
- routers 路由逻辑处理
- runtime：应用运行时数据

添加 Go Modules Replace

打开 `go.mod` 文件，新增 `replace` 配置项，如下：

```
module github.com/EDDYCJY/go-gin-example

go 1.13

require (...)

replace (
        github.com/EDDYCJY/go-gin-example/pkg/setting => ~/go-application/go-gin-example/pkg/setting
        github.com/EDDYCJY/go-gin-example/conf          => ~/go-application/go-gin-example/pkg/conf
        github.com/EDDYCJY/go-gin-example/middleware  => ~/go-application/go-gin-example/middleware
        github.com/EDDYCJY/go-gin-example/models       => ~/go-application/go-gin-example/models
        github.com/EDDYCJY/go-gin-example/routers       => ~/go-application/go-gin-example/routers
)
```

可能你会不理解为什么要特意跑来加 `replace` 配置项，首先你要看到我们使用的是完整的外部模块引用路径（ `github.com/EDDYCJY/go-gin-example/xxx` ），而这个模块还没推送到远程，是没有办法下载下来的，因此需要用 `replace` 将其指定读取本地的模块路径，这样子就可以解决本地模块读取的问题。

注：后续每新增一个本地应用目录，你都需要主动去 go.mod 文件里新增一条 replace（我不会提醒你），如果你漏了，那么编译时会出现报错，找不到那个模块。

初始项目数据库

新建 `blog` 数据库，编码为 `utf8_general_ci` ，在 `blog` 数据库下，新建以下表

1、 标签表

```
CREATE TABLE `blog_tag` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `name` varchar(100) DEFAULT '' COMMENT '标签名称',
  `created_on` int(10) unsigned DEFAULT '0' COMMENT '创建时间',
  `created_by` varchar(100) DEFAULT '' COMMENT '创建人',
  `modified_on` int(10) unsigned DEFAULT '0' COMMENT '修改时间',
  `modified_by` varchar(100) DEFAULT '' COMMENT '修改人',
  `deleted_on` int(10) unsigned DEFAULT '0',
  `state` tinyint(3) unsigned DEFAULT '1' COMMENT '状态 0为禁用、1为启用',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='文章标签管理';
```

2、 文章表

```
CREATE TABLE `blog_article` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `tag_id` int(10) unsigned DEFAULT '0' COMMENT '标签ID',
  `title` varchar(100) DEFAULT '' COMMENT '文章标题',
  `desc` varchar(255) DEFAULT '' COMMENT '简述',
  `content` text,
  `created_on` int(11) DEFAULT NULL,
  `created_by` varchar(100) DEFAULT '' COMMENT '创建人',
  `modified_on` int(10) unsigned DEFAULT '0' COMMENT '修改时间',
  `modified_by` varchar(255) DEFAULT '' COMMENT '修改人',
  `deleted_on` int(10) unsigned DEFAULT '0',
  `state` tinyint(3) unsigned DEFAULT '1' COMMENT '状态 0为禁用1为启用',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COMMENT='文章管理';
```

3、 认证表

```
CREATE TABLE `blog_auth` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT '' COMMENT '账号',
  `password` varchar(50) DEFAULT '' COMMENT '密码',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

```
INSERT INTO `blog`.`blog_auth` (`id`, `username`, `password`) VALUES (null, 'test', 'test123456');
```

编写项目配置包

在 `go-gin-example` 应用目录下，拉取 `go-ini/ini` 的依赖包，如下：

```
$ go get -u github.com/go-ini/ini
go: finding github.com/go-ini/ini v1.48.0
go: downloading github.com/go-ini/ini v1.48.0
go: extracting github.com/go-ini/ini v1.48.0
```

接下来我们需要编写基础的应用配置文件，在 `go-gin-example` 的 `conf` 目录下新建 `app.ini` 文件，写入内容：

```
#debug or release
RUN_MODE = debug

[app]
PAGE_SIZE = 10
JWT_SECRET = 23347$040412

[server]
HTTP_PORT = 8000
READ_TIMEOUT = 60
WRITE_TIMEOUT = 60

[database]
TYPE = mysql
USER = 数据库账号
PASSWORD = 数据库密码
#127.0.0.1:3306
HOST = 数据库IP:数据库端口号
NAME = blog
TABLE_PREFIX = blog_
```

建立调用配置的 `setting` 模块，在 `go-gin-example` 的 `pkg` 目录下新建 `setting` 目录（注意新增 replace 配置），新建 `setting.go` 文件，写入内容：

```
package setting

import (
    "log"
```

```
    "time"

    "github.com/go-ini/ini"
)

var (
    Cfg *ini.File

    RunMode string

    HTTPPort int
    ReadTimeout time.Duration
    WriteTimeout time.Duration

    PageSize int
    JwtSecret string
)

func init() {
    var err error
    Cfg, err = ini.Load("conf/app.ini")
    if err != nil {
        log.Fatalf("Fail to parse 'conf/app.ini': %v", err)
    }

    LoadBase()
    LoadServer()
    LoadApp()
}

func LoadBase() {
    RunMode = Cfg.Section("").Key("RUN_MODE").MustString("debug")
}

func LoadServer() {
    sec, err := Cfg.GetSection("server")
    if err != nil {
        log.Fatalf("Fail to get section 'server': %v", err)
    }

    HTTPPort = sec.Key("HTTP_PORT").MustInt(8000)
    ReadTimeout = time.Duration(sec.Key("READ_TIMEOUT").MustInt(60)) * time.Seco
nd
    WriteTimeout =  time.Duration(sec.Key("WRITE_TIMEOUT").MustInt(60)) * time.S
econd
}
```

```
func LoadApp() {
    sec, err := Cfg.GetSection("app")
    if err != nil {
        log.Fatalf("Fail to get section 'app': %v", err)
    }

    JwtSecret = sec.Key("JWT_SECRET").MustString("!@)*#)!@U#@*!@!)")
    PageSize = sec.Key("PAGE_SIZE").MustInt(10)
}
```

当前的目录结构：

```
go-gin-example
├──── conf
|    └──── app.ini
├──── go.mod
├──── go.sum
├──── middleware
├──── models
├──── pkg
|    └──── setting.go
├──── routers
└──── runtime
```

编写 API 错误码包

建立错误码的 `e` 模块，在 `go-gin-example` 的 `pkg` 目录下新建 `e` 目录（注意新增 replace 配置），新建 `code.go` 和 `msg.go` 文件，写入内容：

1、code.go：

```
package e

const (
    SUCCESS = 200
    ERROR = 500
    INVALID_PARAMS = 400

    ERROR_EXIST_TAG = 10001
    ERROR_NOT_EXIST_TAG = 10002
    ERROR_NOT_EXIST_ARTICLE = 10003

    ERROR_AUTH_CHECK_TOKEN_FAIL = 20001
```

```
    ERROR_AUTH_CHECK_TOKEN_TIMEOUT = 20002
    ERROR_AUTH_TOKEN = 20003
    ERROR_AUTH = 20004
)
```

2、 msg.go：

```
package e

var MsgFlags = map[int]string {
    SUCCESS : "ok",
    ERROR : "fail",
    INVALID_PARAMS : "请求参数错误",
    ERROR_EXIST_TAG : "已存在该标签名称",
    ERROR_NOT_EXIST_TAG : "该标签不存在",
    ERROR_NOT_EXIST_ARTICLE : "该文章不存在",
    ERROR_AUTH_CHECK_TOKEN_FAIL : "Token鉴权失败",
    ERROR_AUTH_CHECK_TOKEN_TIMEOUT : "Token已超时",
    ERROR_AUTH_TOKEN : "Token生成失败",
    ERROR_AUTH : "Token错误",
}

func GetMsg(code int) string {
    msg, ok := MsgFlags[code]
    if ok {
        return msg
    }

    return MsgFlags[ERROR]
}
```

编写工具包

在 `go-gin-example` 的 `pkg` 目录下新建 `util` 目录（注意新增 replace 配置），并拉取 `com` 的依赖包，如下：

```
$ go get -u github.com/unknwon/com
```

编写分页页码的获取方法

在 `util` 目录下新建 `pagination.go` ，写入内容：

```
package util

import (
    "github.com/gin-gonic/gin"
    "github.com/unknwon/com"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

func GetPage(c *gin.Context) int {
    result := 0
    page, _ := com.StrTo(c.Query("page")).Int()
    if page > 0 {
        result = (page - 1) * setting.PageSize
    }

    return result
}
```

编写 models init

拉取 `gorm` 的依赖包，如下：

```
$ go get -u github.com/jinzhu/gorm
```

拉取 `mysql` 驱动的依赖包，如下：

```
$ go get -u github.com/go-sql-driver/mysql
```

完成后，在 `go-gin-example` 的 `models` 目录下新建 `models.go` ，用于 `models` 的初始化使用

```
package models

import (
    "log"
    "fmt"

    "github.com/jinzhu/gorm"
    _ "github.com/jinzhu/gorm/dialects/mysql"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
```

```
)

var db *gorm.DB

type Model struct {
    ID int `gorm:"primary_key" json:"id"`
    CreatedOn int `json:"created_on"`
    ModifiedOn int `json:"modified_on"`
}

func init() {
    var (
        err error
        dbType, dbName, user, password, host, tablePrefix string
    )

    sec, err := setting.Cfg.GetSection("database")
    if err != nil {
        log.Fatal(2, "Fail to get section 'database': %v", err)
    }

    dbType = sec.Key("TYPE").String()
    dbName = sec.Key("NAME").String()
    user = sec.Key("USER").String()
    password = sec.Key("PASSWORD").String()
    host = sec.Key("HOST").String()
    tablePrefix = sec.Key("TABLE_PREFIX").String()

    db, err = gorm.Open(dbType, fmt.Sprintf("%s:%s@tcp(%s)/%s?charset=utf8&parse
Time=True&loc=Local",
        user,
        password,
        host,
        dbName))

    if err != nil {
        log.Println(err)
    }

    gorm.DefaultTableNameHandler = func (db *gorm.DB, defaultTableName string) s
tring  {
        return tablePrefix + defaultTableName;
    }

    db.SingularTable(true)
    db.LogMode(true)
```

```
    db.DB().SetMaxIdleConns(10)
    db.DB().SetMaxOpenConns(100)
}

func CloseDB() {
    defer db.Close()
}
```

编写项目启动、路由文件

最基础的准备工作完成啦，让我们开始编写 Demo 吧！

编写 Demo

在 `go-gin-example` 下建立 `main.go` 作为启动文件（也就是 `main` 包），我们先写个**Demo**，帮助大家理解，写入文件内容：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

func main() {
    router := gin.Default()
    router.GET("/test", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "test",
        })
    })

    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", setting.HTTPPort),
        Handler:        router,
        ReadTimeout:    setting.ReadTimeout,
        WriteTimeout:   setting.WriteTimeout,
        MaxHeaderBytes: 1 << 20,
    }
```

```
    s.ListenAndServe()
}
```

执行 `go run main.go` ，查看命令行是否显示

```
[GIN-debug] [WARNING] Creating an Engine instance with the Logger and Recovery m
iddleware already attached.

[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:   gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /test                     --> main.main.func1 (3 handlers)
```

在本机执行 `curl 127.0.0.1:8000/test` ，检查是否返回 `{"message":"test"}` 。

知识点

那么，我们来延伸一下 Demo 所涉及的知识点！

标准库

- fmt：实现了类似 C 语言 printf 和 scanf 的格式化 I/O。格式化动作（'verb'）源自 C 语言但更简单
- net/http：提供了 HTTP 客户端和服务端的实现

Gin

- gin.Default()：返回 Gin 的 `type Engine struct{...}` ，里面包含 `RouterGroup` ，相当于创建一个路由 `Handlers` ，可以后期绑定各类的路由规则和函数、中间件等
- router.GET(...){...}：创建不同的 HTTP 方法绑定到 `Handlers` 中，也支持 POST、PUT、DELETE、PATCH、OPTIONS、HEAD 等常用的 Restful 方法
- gin.H{...}：就是一个 `map[string]interface{}`
- gin.Context： `Context` 是 `gin` 中的上下文，它允许我们在中间件之间传递变量、管理流、验证 JSON 请求、响应 JSON 请求等，在 `gin` 中包含大量 `Context` 的方法，例如我们常用的 `DefaultQuery` 、 `Query` 、 `DefaultPostForm` 、 `PostForm` 等等

&http.Server 和 ListenAndServe？

1、http.Server：

```
type Server struct {
    Addr    string
    Handler Handler
    TLSConfig *tls.Config
    ReadTimeout time.Duration
    ReadHeaderTimeout time.Duration
    WriteTimeout time.Duration
    IdleTimeout time.Duration
    MaxHeaderBytes int
    ConnState func(net.Conn, ConnState)
    ErrorLog *log.Logger
}
```

- Addr：监听的 TCP 地址，格式为`:8000`
- Handler：http 句柄，实质为`ServeHTTP`，用于处理程序响应 HTTP 请求
- TLSConfig：安全传输层协议（TLS）的配置
- ReadTimeout：允许读取的最大时间
- ReadHeaderTimeout：允许读取请求头的最大时间
- WriteTimeout：允许写入的最大时间
- IdleTimeout：等待的最大时间
- MaxHeaderBytes：请求头的最大字节数
- ConnState：指定一个可选的回调函数，当客户端连接发生变化时调用
- ErrorLog：指定一个可选的日志记录器，用于接收程序的意外行为和底层系统错误；如果未设置或为`nil`则默认以日志包的标准日志记录器完成（也就是在控制台输出）

2、 ListenAndServe：

```
func (srv *Server) ListenAndServe() error {
    addr := srv.Addr
    if addr == "" {
        addr = ":http"
    }
    ln, err := net.Listen("tcp", addr)
    if err != nil {
        return err
    }
    return srv.Serve(tcpKeepAliveListener{ln.(*net.TCPListener)})
}
```

开始监听服务，监听 TCP 网络地址，Addr 和调用应用程序处理连接上的请求。

我们在源码中看到 `Addr` 是调用我们在 `&http.Server` 中设置的参数，因此我们在设置时要用 `&` ，我们要改变参数的值，因为我们 `ListenAndServe` 和其他一些方法需要用到 `&http.Server` 中的参数，他们是相互影响的。

3、 `http.ListenAndServe` 和 连载一 的 `r.Run()` 有区别吗？

我们看看 `r.Run` 的实现：

```
func (engine *Engine) Run(addr ...string) (err error) {
    defer func() { debugPrintError(err) }()

    address := resolveAddress(addr)
    debugPrint("Listening and serving HTTP on %s\n", address)
    err = http.ListenAndServe(address, engine)
    return
}
```

通过分析源码，得知**本质上没有区别**，同时也得知了启动 `gin` 时的监听 debug 信息在这里输出。

4、 为什么 Demo 里会有 `WARNING` ？

首先我们可以看下 `Default()` 的实现

```
// Default returns an Engine instance with the Logger and Recovery middleware al
ready attached.
func Default() *Engine {
    debugPrintWARNINGDefault()
    engine := New()
    engine.Use(Logger(), Recovery())
    return engine
}
```

大家可以看到默认情况下，已经附加了日志、恢复中间件的引擎实例。并且在开头调用了 `debugPrintWARNINGDefault()` ，而它的实现就是输出该行日志

```
func debugPrintWARNINGDefault() {
    debugPrint(`[WARNING] Creating an Engine instance with the Logger and Recove
ry middleware already attached.
`)
}
```

而另外一个 `Running in "debug" mode. Switch to "release" mode in production.` ，是运行模式原因，并不难理解，已在配置文件的管控下 :-)，运维人员随时就可以修改它的配置。

5、 Demo 的 `router.GET` 等路由规则可以不写在 `main` 包中吗？

我们发现 `router.GET` 等路由规则，在 Demo 中被编写在了 `main` 包中，感觉很奇怪，我们去抽离这部分逻辑！

在 `go-gin-example` 下 `routers` 目录新建 `router.go` 文件，写入内容：

```
package routers

import (
    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

func InitRouter() *gin.Engine {
    r := gin.New()

    r.Use(gin.Logger())

    r.Use(gin.Recovery())

    gin.SetMode(setting.RunMode)

    r.GET("/test", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "test",
        })
    })

    return r
}
```

修改 `main.go` 的文件内容：

```
package main

import (
    "fmt"
    "net/http"

    "github.com/EDDYCJY/go-gin-example/routers"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)
```

```
func main() {
    router := routers.InitRouter()

    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", setting.HTTPPort),
        Handler:        router,
        ReadTimeout:    setting.ReadTimeout,
        WriteTimeout:   setting.WriteTimeout,
        MaxHeaderBytes: 1 << 20,
    }

    s.ListenAndServe()
}
```

当前目录结构：

```
go-gin-example/
├──── conf
|    └──── app.ini
├──── main.go
├──── middleware
├──── models
|    └──── models.go
├──── pkg
|    ├──── e
|    |    ├──── code.go
|    |    └──── msg.go
|    ├──── setting
|    |    └──── setting.go
|    └──── util
|         └──── pagination.go
├──── routers
|    └──── router.go
├──── runtime
```

重启服务，执行 `curl 127.0.0.1:8000/test` 查看是否正确返回。

下一节，我们将以我们的 Demo 为起点进行修改，开始编码！

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

Gin搭建Blog API's

涉及知识点

- Gin：Golang 的一个微框架，性能极佳。
- beego-validation：本节采用的 beego 的表单验证库，中文文档。
- gorm，对开发人员友好的 ORM 框架，英文文档
- com，一个小而美的工具包。

本文目标

- 完成博客的标签类接口定义和编写

定义接口2020-12-27 13:26:30 星期日

本节正是编写标签的逻辑，我们想一想，一般接口为增删改查是基础的，那么我们定义一下接口吧！

- 获取标签列表：GET(“/tags”)
- 新建标签：POST(“/tags”)
- 更新指定标签：PUT(“/tags/:id”)
- 删除指定标签：DELETE(“/tags/:id”)

编写路由空壳

开始编写路由文件逻辑，在 `routers` 下新建 `api` 目录，我们当前是第一个 API 大版本，因此在 `api` 下新建 `v1` 目录，再新建 `tag.go` 文件，写入内容：

```
package v1

import (
    "github.com/gin-gonic/gin"
)

//获取多个文章标签
func GetTags(c *gin.Context) {
```

```
}

//新增文章标签
func AddTag(c *gin.Context) {
}

//修改文章标签
func EditTag(c *gin.Context) {
}

//删除文章标签
func DeleteTag(c *gin.Context) {
}
```

注册路由

我们打开 `routers` 下的 `router.go` 文件，修改文件内容为：

```
package routers

import (
    "github.com/gin-gonic/gin"

    "gin-blog/routers/api/v1"
    "gin-blog/pkg/setting"
)

func InitRouter() *gin.Engine {
    r := gin.New()

    r.Use(gin.Logger())

    r.Use(gin.Recovery())

    gin.SetMode(setting.RunMode)

    apiv1 := r.Group("/api/v1")
    {
        //获取标签列表
        apiv1.GET("/tags", v1.GetTags)
        //新建标签
        apiv1.POST("/tags", v1.AddTag)
        //更新指定标签
        apiv1.PUT("/tags/:id", v1.EditTag)
```

```
        //删除指定标签
        apiv1.DELETE("/tags/:id", v1.DeleteTag)
    }

    return r
}
```

当前目录结构：

```
gin-blog/
├──── conf
|    └──── app.ini
├──── main.go
├──── middleware
├──── models
|    └──── models.go
├──── pkg
|    ├──── e
|    |    ├──── code.go
|    |    └──── msg.go
|    ├──── setting
|    |    └──── setting.go
|    └──── util
|         └──── pagination.go
├──── routers
|    ├──── api
|    |    └──── v1
|    |         └──── tag.go
|    └──── router.go
├──── runtime
```

检验路由是否注册成功

回到命令行，执行 `go run main.go` ，检查路由规则是否注册成功。

```
$ go run main.go
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /api/v1/tags              --> gin-blog/routers/api/v1.GetTags
(3 handlers)
[GIN-debug] POST   /api/v1/tags              --> gin-blog/routers/api/v1.AddTag
```

```
 (3 handlers)
[GIN-debug] PUT    /api/v1/tags/:id          --> gin-blog/routers/api/v1.EditTag
(3 handlers)
[GIN-debug] DELETE /api/v1/tags/:id          --> gin-blog/routers/api/v1.DeleteT
ag (3 handlers)
```

运行成功，那么我们愉快的**开始编写我们的接口**吧！

下载依赖包

首先我们要拉取 `validation` 的依赖包，在后面的接口里会使用到表单验证

```
$ go get -u github.com/astaxie/beego/validation
```

编写标签列表的 models 逻辑

创建 `models` 目录下的 `tag.go` ，写入文件内容：

```
package models

type Tag struct {
    Model

    Name string `json:"name"`
    CreatedBy string `json:"created_by"`
    ModifiedBy string `json:"modified_by"`
    State int `json:"state"`
}

func GetTags(pageNum int, pageSize int, maps interface {}) (tags []Tag) {
    db.Where(maps).Offset(pageNum).Limit(pageSize).Find(&tags)

    return
}

func GetTagTotal(maps interface {}) (count int){
    db.Model(&Tag{}).Where(maps).Count(&count)

    return
}
```

1. 我们创建了一个 `Tag struct{}` ，用于 `Gorm` 的使用。并给予了附属属性 `json` ，这样子在 `c.JSON` 的时候就会自动转换格式，非常的便利
2. 可能会有的初学者看到 `return` ，而后面没有跟着变量，会不理解；其实你可以看到在函数末端，我们已经显示声明了返回值，这个变量在函数体内也可以直接使用，因为他在一开始就被声明了
3. 有人会疑惑 `db` 是哪里来的；因为在同个 `models` 包下，因此 `db *gorm.DB` 是可以直接使用的

编写标签列表的路由逻辑

打开 `routers` 目录下 v1 版本的 `tag.go` ，第一我们先编写**获取标签列表的接口**

修改文件内容：

```
package v1

import (
    "net/http"

    "github.com/gin-gonic/gin"
    //"github.com/astaxie/beego/validation"
    "github.com/Unknwon/com"

    "gin-blog/pkg/e"
    "gin-blog/models"
    "gin-blog/pkg/util"
    "gin-blog/pkg/setting"
)

//获取多个文章标签
func GetTags(c *gin.Context) {
    name := c.Query("name")

    maps := make(map[string]interface{})
    data := make(map[string]interface{})

    if name != "" {
        maps["name"] = name
    }

    var state int = -1
    if arg := c.Query("state"); arg != "" {
        state = com.StrTo(arg).MustInt()
```

```
        maps["state"] = state
    }

    code := e.SUCCESS

    data["lists"] = models.GetTags(util.GetPage(c), setting.PageSize, maps)
    data["total"] = models.GetTagTotal(maps)

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : data,
    })
}

//新增文章标签
func AddTag(c *gin.Context) {
}

//修改文章标签
func EditTag(c *gin.Context) {
}

//删除文章标签
func DeleteTag(c *gin.Context) {
}
```

1. `c.Query`可用于获取`?name=test&state=1`这类 URL 参数，而`c.DefaultQuery`则支持设置一个默认值
2. `code`变量使用了`e`模块的错误编码，这正是先前规划好的错误码，方便排错和识别记录
3. `util.GetPage`保证了各接口的`page`处理是一致的
4. `c *gin.Context`是`Gin`很重要的组成部分，可以理解为上下文，它允许我们在中间件之间传递变量、管理流、验证请求的 JSON 和呈现 JSON 响应

在本机执行`curl 127.0.0.1:8000/api/v1/tags`，正确的返回值为`{"code":200,"data":{"lists":[],"total":0},"msg":"ok"}`，若存在问题请结合 gin 结果进行拍错。

在获取标签列表接口中，我们可以根据`name`、`state`、`page`来筛选查询条件，分页的步长可通过`app.ini`进行配置，以`lists`、`total`的组合返回达到分页效果。

编写新增标签的 models 逻辑

接下来我们编写**新增标签**的接口

打开 `models` 目录下的 `tag.go` ，修改文件（增加 2 个方法）：

```
...
func ExistTagByName(name string) bool {
    var tag Tag
    db.Select("id").Where("name = ?", name).First(&tag)
    if tag.ID > 0 {
        return true
    }

    return false
}

func AddTag(name string, state int, createdBy string) bool{
    db.Create(&Tag {
        Name : name,
        State : state,
        CreatedBy : createdBy,
    })

    return true
}
...
```

编写新增标签的路由逻辑

打开 `routers` 目录下的 `tag.go` ，修改文件（变动 AddTag 方法）：

```
package v1

import (
    "log"
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/astaxie/beego/validation"
    "github.com/Unknwon/com"

    "gin-blog/pkg/e"
    "gin-blog/models"
    "gin-blog/pkg/util"
    "gin-blog/pkg/setting"
```

```
)

...

//新增文章标签
func AddTag(c *gin.Context) {
    name := c.Query("name")
    state := com.StrTo(c.DefaultQuery("state", "0")).MustInt()
    createdBy := c.Query("created_by")

    valid := validation.Validation{}
    valid.Required(name, "name").Message("名称不能为空")
    valid.MaxSize(name, 100, "name").Message("名称最长为100字符")
    valid.Required(createdBy, "created_by").Message("创建人不能为空")
    valid.MaxSize(createdBy, 100, "created_by").Message("创建人最长为100字符")
    valid.Range(state, 0, 1, "state").Message("状态只允许0或1")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        if ! models.ExistTagByName(name) {
            code = e.SUCCESS
            models.AddTag(name, state, createdBy)
        } else {
            code = e.ERROR_EXIST_TAG
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : make(map[string]string),
    })
}
...
```

用 `Postman` 用 POST 访问 `http://127.0.0.1:8000/api/v1/tags?name=1&state=1&created_by=test` ，查看 `code` 是否返回 `200` 及 `blog_tag` 表中是否有值，有值则正确。

编写 models callbacks

但是这个时候大家会发现，我明明新增了标签，但 `created_on` 居然没有值，那做修改标签的时候 `modified_on` 会不会也存在这个问题？

为了解决这个问题，我们需要打开 `models` 目录下的 `tag.go` 文件，修改文件内容（修改包引用和增加 2 个方法）：

```
package models

import (
    "time"

    "github.com/jinzhu/gorm"
)

...

func (tag *Tag) BeforeCreate(scope *gorm.Scope) error {
    scope.SetColumn("CreatedOn", time.Now().Unix())

    return nil
}

func (tag *Tag) BeforeUpdate(scope *gorm.Scope) error {
    scope.SetColumn("ModifiedOn", time.Now().Unix())

    return nil
}
```

重启服务，再在用 `Postman` 用 POST 访问 `http://127.0.0.1:8000/api/v1/tags?name=2&state=1&created_by=test` ，发现 `created_on` 已经有值了！

在这几段代码中，涉及到知识点：

这属于 `gorm` 的 `Callbacks` ，可以将回调方法定义为模型结构的指针，在创建、更新、查询、删除时将被调用，如果任何回调返回错误，gorm 将停止未来操作并回滚所有更改。

`gorm` 所支持的回调方法：

- 创建：BeforeSave、BeforeCreate、AfterCreate、AfterSave
- 更新：BeforeSave、BeforeUpdate、AfterUpdate、AfterSave
- 删除：BeforeDelete、AfterDelete
- 查询：AfterFind

编写其余接口的路由逻辑

接下来，我们一口气把剩余的两个接口（EditTag、DeleteTag）完成吧

打开 `routers` 目录下 v1 版本的 `tag.go` 文件，修改内容：

```
...

//修改文章标签
func EditTag(c *gin.Context) {
    id := com.StrTo(c.Param("id")).MustInt()
    name := c.Query("name")
    modifiedBy := c.Query("modified_by")

    valid := validation.Validation{}

    var state int = -1
    if arg := c.Query("state"); arg != "" {
        state = com.StrTo(arg).MustInt()
        valid.Range(state, 0, 1, "state").Message("状态只允许0或1")
    }

    valid.Required(id, "id").Message("ID不能为空")
    valid.Required(modifiedBy, "modified_by").Message("修改人不能为空")
    valid.MaxSize(modifiedBy, 100, "modified_by").Message("修改人最长为100字符")
    valid.MaxSize(name, 100, "name").Message("名称最长为100字符")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        code = e.SUCCESS
        if models.ExistTagByID(id) {
            data := make(map[string]interface{})
            data["modified_by"] = modifiedBy
            if name != "" {
                data["name"] = name
            }
            if state != -1 {
                data["state"] = state
            }

            models.EditTag(id, data)
        } else {
            code = e.ERROR_NOT_EXIST_TAG
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
```

```
        "msg" : e.GetMsg(code),
        "data" : make(map[string]string),
    })
}

//删除文章标签
func DeleteTag(c *gin.Context) {
    id := com.StrTo(c.Param("id")).MustInt()

    valid := validation.Validation{}
    valid.Min(id, 1, "id").Message("ID必须大于0")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        code = e.SUCCESS
        if models.ExistTagByID(id) {
            models.DeleteTag(id)
        } else {
            code = e.ERROR_NOT_EXIST_TAG
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : make(map[string]string),
    })
}
```

编写其余接口的 models 逻辑

打开 `models` 下的 `tag.go` ，修改文件内容：

```
...

func ExistTagByID(id int) bool {
    var tag Tag
    db.Select("id").Where("id = ?", id).First(&tag)
    if tag.ID > 0 {
        return true
    }

    return false
}
```

```
func DeleteTag(id int) bool {
    db.Where("id = ?", id).Delete(&Tag{})

    return true
}

func EditTag(id int, data interface {}) bool {
    db.Model(&Tag{}).Where("id = ?", id).Updates(data)

    return true
}
...
```

验证功能

重启服务，用 Postman

- PUT 访问 http://127.0.0.1:8000/api/v1/tags/1?name=edit1&state=0&modified_by=edit1 ， 查看 code 是否返回 200
- DELETE 访问 http://127.0.0.1:8000/api/v1/tags/1 ， 查看 code 是否返回 200

至此，Tag 的 API’s 完成，下一节我们将开始 Article 的 API’s 编写！

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

Gin搭建Blog API's

涉及知识点

- Gin：Golang 的一个微框架，性能极佳。
- beego-validation：本节采用的 beego 的表单验证库，中文文档。
- gorm，对开发人员友好的 ORM 框架，英文文档
- com，一个小而美的工具包。

本文目标

- 完成博客的文章类接口定义和编写

定义接口

本节编写文章的逻辑，我们定义一下接口吧！

- 获取文章列表：GET(“/articles”)
- 获取指定文章：POST(“/articles/:id”)
- 新建文章：POST(“/articles”)
- 更新指定文章：PUT(“/articles/:id”)
- 删除指定文章：DELETE(“/articles/:id”)

编写路由逻辑

在 `routers` 的 v1 版本下，新建 `article.go` 文件，写入内容：

```
package v1

import (
    "github.com/gin-gonic/gin"
)

//获取单个文章
func GetArticle(c *gin.Context) {
}
```

```
//获取多个文章
func GetArticles(c *gin.Context) {
}

//新增文章
func AddArticle(c *gin.Context) {
}

//修改文章
func EditArticle(c *gin.Context) {
}

//删除文章
func DeleteArticle(c *gin.Context) {
}
```

我们打开 `routers` 下的 `router.go` 文件，修改文件内容为：

```
package routers

import (
    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/routers/api/v1"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

func InitRouter() *gin.Engine {
    ...
    apiv1 := r.Group("/api/v1")
    {
        ...
        //获取文章列表
        apiv1.GET("/articles", v1.GetArticles)
        //获取指定文章
        apiv1.GET("/articles/:id", v1.GetArticle)
        //新建文章
        apiv1.POST("/articles", v1.AddArticle)
        //更新指定文章
        apiv1.PUT("/articles/:id", v1.EditArticle)
        //删除指定文章
        apiv1.DELETE("/articles/:id", v1.DeleteArticle)
    }
```

```
    return r
}
```

当前目录结构：

```
go-gin-example/
├──── conf
|    └──── app.ini
├──── main.go
├──── middleware
├──── models
|    ├──── models.go
|    └──── tag.go
├──── pkg
|    ├──── e
|    |    ├──── code.go
|    |    └──── msg.go
|    ├──── setting
|    |    └──── setting.go
|    └──── util
|         └──── pagination.go
├──── routers
|    ├──── api
|    |    └──── v1
|    |         ├──── article.go
|    |         └──── tag.go
|    └──── router.go
├──── runtime
```

在基础的路由规则配置结束后，我们**开始编写我们的接口**吧！

##编写 models 逻辑

创建 `models` 目录下的 `article.go` ，写入文件内容：

```
package models

import (
    "github.com/jinzhu/gorm"

    "time"
)

type Article struct {
    Model
```

```
    TagID int `json:"tag_id" gorm:"index"`
    Tag   Tag `json:"tag"`

    Title string `json:"title"`
    Desc string `json:"desc"`
    Content string `json:"content"`
    CreatedBy string `json:"created_by"`
    ModifiedBy string `json:"modified_by"`
    State int `json:"state"`
}


func (article *Article) BeforeCreate(scope *gorm.Scope) error {
    scope.SetColumn("CreatedOn", time.Now().Unix())

    return nil
}

func (article *Article) BeforeUpdate(scope *gorm.Scope) error {
    scope.SetColumn("ModifiedOn", time.Now().Unix())

    return nil
}
```

我们创建了一个 `Article struct {}` ，与 `Tag` 不同的是， `Article` 多了几项，如下：

1. `gorm:index` ，用于声明这个字段为索引，如果你使用了自动迁移功能则会有所影响，在不使用则无影响
2. `Tag` 字段，实际是一个嵌套的 `struct` ，它利用 `TagID` 与 `Tag` 模型相互关联，在执行查询的时候，能够达到 `Article` 、 `Tag` 关联查询的功能
3. `time.Now().Unix()` 返回当前的时间戳

接下来，请确保已对上一章节的内容通读且了解，由于逻辑偏差不会太远，我们本节直接编写这五个接口

打开 `models` 目录下的 `article.go` ，修改文件内容：

```
package models

import (
    "time"

    "github.com/jinzhu/gorm"
```

```
)

type Article struct {
    Model

    TagID int `json:"tag_id" gorm:"index"`
    Tag   Tag `json:"tag"`

    Title string `json:"title"`
    Desc string `json:"desc"`
    Content string `json:"content"`
    CreatedBy string `json:"created_by"`
    ModifiedBy string `json:"modified_by"`
    State int `json:"state"`
}


func ExistArticleByID(id int) bool {
    var article Article
    db.Select("id").Where("id = ?", id).First(&article)

    if article.ID > 0 {
        return true
    }

    return false
}

func GetArticleTotal(maps interface {}) (count int){
    db.Model(&Article{}).Where(maps).Count(&count)

    return
}

func GetArticles(pageNum int, pageSize int, maps interface {}) (articles []Artic
le) {
    db.Preload("Tag").Where(maps).Offset(pageNum).Limit(pageSize).Find(&article
s)

    return
}

func GetArticle(id int) (article Article) {
    db.Where("id = ?", id).First(&article)
    db.Model(&article).Related(&article.Tag)
```

```
    return
}

func EditArticle(id int, data interface {}) bool {
    db.Model(&Article{}).Where("id = ?", id).Updates(data)

    return true
}

func AddArticle(data map[string]interface {}) bool {
    db.Create(&Article {
        TagID : data["tag_id"].(int),
        Title : data["title"].(string),
        Desc : data["desc"].(string),
        Content : data["content"].(string),
        CreatedBy : data["created_by"].(string),
        State : data["state"].(int),
    })

    return true
}

func DeleteArticle(id int) bool {
    db.Where("id = ?", id).Delete(Article{})

    return true
}

func (article *Article) BeforeCreate(scope *gorm.Scope) error {
    scope.SetColumn("CreatedOn", time.Now().Unix())

    return nil
}

func (article *Article) BeforeUpdate(scope *gorm.Scope) error {
    scope.SetColumn("ModifiedOn", time.Now().Unix())

    return nil
}
```

在这里，我们拿出三点不同来讲，如下：

1、 我们的 `Article` 是如何关联到 `Tag` ？

```
func GetArticle(id int) (article Article) {
    db.Where("id = ?", id).First(&article)
    db.Model(&article).Related(&article.Tag)

    return
}
```

能够达到关联，首先是 `gorm` 本身做了大量的约定俗成

- `Article` 有一个结构体成员是 `TagID` ，就是外键。 `gorm` 会通过类名+ID 的方式去找到这两个类之间的关联关系
- `Article` 有一个结构体成员是 `Tag` ，就是我们嵌套在 `Article` 里的 `Tag` 结构体，我们可以通过 `Related` 进行关联查询

2、 `Preload` 是什么东西，为什么查询可以得出每一项的关联 `Tag` ？

```
func GetArticles(pageNum int, pageSize int, maps interface {}) (articles []Artic
le) {
    db.Preload("Tag").Where(maps).Offset(pageNum).Limit(pageSize).Find(&article
s)

    return
}
```

`Preload` 就是一个预加载器，它会执行两条 SQL，分别是 `SELECT * FROM blog_articles;` 和 `SELECT * FROM blog_tag WHERE id IN (1,2,3,4);` ，那么在查询出结构后， `gorm` 内部处理对应的映射逻辑，将其填充到 `Article` 的 `Tag` 中，会特别方便，并且避免了循环查询

那么有没有别的办法呢，大致是两种

- `gorm` 的 `Join`
- 循环 `Related`

综合之下，还是 `Preload` 更好，如果你有更优的方案，欢迎说一下 :)

3、 `v, (I)` 是什么？

`v` 表示一个接口值， `I` 表示接口类型。这个实际就是 Golang 中的**类型断言**，用于判断一个接口值的实际类型是否为某个类型，或一个非接口值的类型是否实现了某个接口类型

打开 `routers` 目录下 v1 版本的 `article.go` 文件，修改文件内容：

```
package v1

import (
    "net/http"
    "log"

    "github.com/gin-gonic/gin"
    "github.com/astaxie/beego/validation"
    "github.com/unknwon/com"

    "github.com/EDDYCJY/go-gin-example/models"
    "github.com/EDDYCJY/go-gin-example/pkg/e"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
    "github.com/EDDYCJY/go-gin-example/pkg/util"
)

//获取单个文章
func GetArticle(c *gin.Context) {
    id := com.StrTo(c.Param("id")).MustInt()

    valid := validation.Validation{}
    valid.Min(id, 1, "id").Message("ID必须大于0")

    code := e.INVALID_PARAMS
    var data interface {}
    if ! valid.HasErrors() {
        if models.ExistArticleByID(id) {
            data = models.GetArticle(id)
            code = e.SUCCESS
        } else {
            code = e.ERROR_NOT_EXIST_ARTICLE
        }
    } else {
        for _, err := range valid.Errors {
            log.Printf("err.key: %s, err.message: %s", err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : data,
    })
}
```

```
//获取多个文章
func GetArticles(c *gin.Context) {
    data := make(map[string]interface{})
    maps := make(map[string]interface{})
    valid := validation.Validation{}

    var state int = -1
    if arg := c.Query("state"); arg != "" {
        state = com.StrTo(arg).MustInt()
        maps["state"] = state

        valid.Range(state, 0, 1, "state").Message("状态只允许0或1")
    }

    var tagId int = -1
    if arg := c.Query("tag_id"); arg != "" {
        tagId = com.StrTo(arg).MustInt()
        maps["tag_id"] = tagId

        valid.Min(tagId, 1, "tag_id").Message("标签ID必须大于0")
    }

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        code = e.SUCCESS

        data["lists"] = models.GetArticles(util.GetPage(c), setting.PageSize, ma
ps)
        data["total"] = models.GetArticleTotal(maps)

    } else {
        for _, err := range valid.Errors {
            log.Printf("err.key: %s, err.message: %s", err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : data,
    })
}

//新增文章
func AddArticle(c *gin.Context) {
    tagId := com.StrTo(c.Query("tag_id")).MustInt()
```

```
    title := c.Query("title")
    desc := c.Query("desc")
    content := c.Query("content")
    createdBy := c.Query("created_by")
    state := com.StrTo(c.DefaultQuery("state", "0")).MustInt()

    valid := validation.Validation{}
    valid.Min(tagId, 1, "tag_id").Message("标签ID必须大于0")
    valid.Required(title, "title").Message("标题不能为空")
    valid.Required(desc, "desc").Message("简述不能为空")
    valid.Required(content, "content").Message("内容不能为空")
    valid.Required(createdBy, "created_by").Message("创建人不能为空")
    valid.Range(state, 0, 1, "state").Message("状态只允许0或1")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        if models.ExistTagByID(tagId) {
            data := make(map[string]interface {})
            data["tag_id"] = tagId
            data["title"] = title
            data["desc"] = desc
            data["content"] = content
            data["created_by"] = createdBy
            data["state"] = state

            models.AddArticle(data)
            code = e.SUCCESS
        } else {
            code = e.ERROR_NOT_EXIST_TAG
        }
    } else {
        for _, err := range valid.Errors {
            log.Printf("err.key: %s, err.message: %s", err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : make(map[string]interface{}),
    })
}

//修改文章
func EditArticle(c *gin.Context) {
    valid := validation.Validation{}
```

```
    id := com.StrTo(c.Param("id")).MustInt()
    tagId := com.StrTo(c.Query("tag_id")).MustInt()
    title := c.Query("title")
    desc := c.Query("desc")
    content := c.Query("content")
    modifiedBy := c.Query("modified_by")

    var state int = -1
    if arg := c.Query("state"); arg != "" {
        state = com.StrTo(arg).MustInt()
        valid.Range(state, 0, 1, "state").Message("状态只允许0或1")
    }

    valid.Min(id, 1, "id").Message("ID必须大于0")
    valid.MaxSize(title, 100, "title").Message("标题最长为100字符")
    valid.MaxSize(desc, 255, "desc").Message("简述最长为255字符")
    valid.MaxSize(content, 65535, "content").Message("内容最长为65535字符")
    valid.Required(modifiedBy, "modified_by").Message("修改人不能为空")
    valid.MaxSize(modifiedBy, 100, "modified_by").Message("修改人最长为100字符")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        if models.ExistArticleByID(id) {
            if models.ExistTagByID(tagId) {
                data := make(map[string]interface {})
                if tagId > 0 {
                    data["tag_id"] = tagId
                }
                if title != "" {
                    data["title"] = title
                }
                if desc != "" {
                    data["desc"] = desc
                }
                if content != "" {
                    data["content"] = content
                }

                data["modified_by"] = modifiedBy

                models.EditArticle(id, data)
                code = e.SUCCESS
            } else {
                code = e.ERROR_NOT_EXIST_TAG
            }
```

```
        } else {
            code = e.ERROR_NOT_EXIST_ARTICLE
        }
    } else {
        for _, err := range valid.Errors {
            log.Printf("err.key: %s, err.message: %s", err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : make(map[string]string),
    })
}

//删除文章
func DeleteArticle(c *gin.Context) {
    id := com.StrTo(c.Param("id")).MustInt()

    valid := validation.Validation{}
    valid.Min(id, 1, "id").Message("ID必须大于0")

    code := e.INVALID_PARAMS
    if ! valid.HasErrors() {
        if models.ExistArticleByID(id) {
            models.DeleteArticle(id)
            code = e.SUCCESS
        } else {
            code = e.ERROR_NOT_EXIST_ARTICLE
        }
    } else {
        for _, err := range valid.Errors {
            log.Printf("err.key: %s, err.message: %s", err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : make(map[string]string),
    })
}
```

当前目录结构：

```
go-gin-example/
├──── conf
|    └──── app.ini
├──── main.go
├──── middleware
├──── models
|    ├──── article.go
|    ├──── models.go
|    └──── tag.go
├──── pkg
|    ├──── e
|    |    ├──── code.go
|    |    └──── msg.go
|    ├──── setting
|    |    └──── setting.go
|    └──── util
|         └──── pagination.go
├──── routers
|    ├──── api
|    |    └──── v1
|    |         ├──── article.go
|    |         └──── tag.go
|    └──── router.go
├──── runtime
```

验证功能

我们重启服务，执行 `go run main.go` ，检查控制台输出结果

```
$ go run main.go
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /api/v1/tags              --> gin-blog/routers/api/v1.GetTags
(3 handlers)
[GIN-debug] POST   /api/v1/tags              --> gin-blog/routers/api/v1.AddTag
 (3 handlers)
[GIN-debug] PUT    /api/v1/tags/:id          --> gin-blog/routers/api/v1.EditTag
(3 handlers)
[GIN-debug] DELETE /api/v1/tags/:id          --> gin-blog/routers/api/v1.DeleteT
ag (3 handlers)
[GIN-debug] GET    /api/v1/articles          --> gin-blog/routers/api/v1.GetArti
```

```
cles (3 handlers)
[GIN-debug] GET    /api/v1/articles/:id      --> gin-blog/routers/api/v1.GetArti
cle (3 handlers)
[GIN-debug] POST   /api/v1/articles          --> gin-blog/routers/api/v1.AddArti
cle (3 handlers)
[GIN-debug] PUT    /api/v1/articles/:id      --> gin-blog/routers/api/v1.EditArt
icle (3 handlers)
[GIN-debug] DELETE /api/v1/articles/:id      --> gin-blog/routers/api/v1.DeleteA
rticle (3 handlers)
```

使用 `Postman` 检验接口是否正常，在这里大家可以选用合适的参数传递方式，此处为了方便展示我选用了 GET/Param 传参的方式，而后期会改为 POST。

- POST：http://127.0.0.1:8000/api/v1/articles?tag_id=1&title=test1&desc=test-desc&content=test-content&created_by=test-created&state=1
- GET：http://127.0.0.1:8000/api/v1/articles
- GET：http://127.0.0.1:8000/api/v1/articles/1
- PUT：http://127.0.0.1:8000/api/v1/articles/1?tag_id=1&title=test-edit1&desc=test-desc-edit&content=test-content-edit&modified_by=test-created-edit&state=0
- DELETE：http://127.0.0.1:8000/api/v1/articles/1

至此，我们的 API's 编写就到这里，下一节我们将介绍另外的一些技巧！

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

使用 JWT 进行身份校验

涉及知识点

- JWT

本文目标

在前面几节中，我们已经基本的完成了 API's 的编写，但是，还存在一些非常严重的问题，例如，我们现在的 API 是可以随意调用的，这显然还不安全全，在本文中我们通过 jwt-go（GoDoc）的方式来简单解决这个问题。

下载依赖包

首先，我们下载 jwt-go 的依赖包，如下：

```
go get -u github.com/dgrijalva/jwt-go
```

编写 jwt 工具包

我们需要编写一个 `jwt` 的工具包，我们在 `pkg` 下的 `util` 目录新建 `jwt.go` ，写入文件内容：

```
package util

import (
    "time"

    jwt "github.com/dgrijalva/jwt-go"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

var jwtSecret = []byte(setting.JwtSecret)

type Claims struct {
    Username string `json:"username"`
    Password string `json:"password"`
    jwt.StandardClaims
```

```
}

func GenerateToken(username, password string) (string, error) {
    nowTime := time.Now()
    expireTime := nowTime.Add(3 * time.Hour)

    claims := Claims{
        username,
        password,
        jwt.StandardClaims {
            ExpiresAt : expireTime.Unix(),
            Issuer : "gin-blog",
        },
    }

    tokenClaims := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    token, err := tokenClaims.SignedString(jwtSecret)

    return token, err
}

func ParseToken(token string) (*Claims, error) {
    tokenClaims, err := jwt.ParseWithClaims(token, &Claims{}, func(token *jwt.To
ken) (interface{}, error) {
        return jwtSecret, nil
    })

    if tokenClaims != nil {
        if claims, ok := tokenClaims.Claims.(*Claims); ok && tokenClaims.Valid {
            return claims, nil
        }
    }

    return nil, err
}
```

在这个工具包，我们涉及到

- `NewWithClaims(method SigningMethod, claims Claims)`，`method`对应着`SigningMethodHMAC struct{}`，其包含`SigningMethodHS256`、`SigningMethodHS384`、`SigningMethodHS512`三种`crypto.Hash`方案

- `func (t *Token) SignedString(key interface{})` 该方法内部生成签名字符串，再用于获取完整、已签名的 `token`
- `func (p *Parser) ParseWithClaims` 用于解析鉴权的声明，方法内部主要是具体的解码和校验的过程，最终返回 `*Token`
- `func (m MapClaims) Valid()` 验证基于时间的声明 `exp, iat, nbf`，注意如果没有任何声明在令牌中，仍然会被认为是有效的。并且对于时区偏差没有计算方法

有了 `jwt` 工具包，接下来我们要编写要用于 `Gin` 的中间件，我们在 `middleware` 下新建 `jwt` 目录，新建 `jwt.go` 文件，写入内容：

```
package jwt

import (
    "time"
    "net/http"

    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/pkg/util"
    "github.com/EDDYCJY/go-gin-example/pkg/e"
)

func JWT() gin.HandlerFunc {
    return func(c *gin.Context) {
        var code int
        var data interface{}

        code = e.SUCCESS
        token := c.Query("token")
        if token == "" {
            code = e.INVALID_PARAMS
        } else {
            claims, err := util.ParseToken(token)
            if err != nil {
                code = e.ERROR_AUTH_CHECK_TOKEN_FAIL
            } else if time.Now().Unix() > claims.ExpiresAt {
                code = e.ERROR_AUTH_CHECK_TOKEN_TIMEOUT
            }
        }

        if code != e.SUCCESS {
            c.JSON(http.StatusUnauthorized, gin.H{
                "code" : code,
```

```
                "msg" : e.GetMsg(code),
                "data" : data,
            })

            c.Abort()
            return
        }

        c.Next()
    }
}
```

如何获取 Token

那么我们如何调用它呢，我们还要获取 Token 呢？

1、 我们要新增一个获取 Token 的 API

在 models 下新建 auth.go 文件，写入内容：

```
package models

type Auth struct {
    ID int `gorm:"primary_key" json:"id"`
    Username string `json:"username"`
    Password string `json:"password"`
}

func CheckAuth(username, password string) bool {
    var auth Auth
    db.Select("id").Where(Auth{Username : username, Password : password}).First
(&auth)
    if auth.ID > 0 {
        return true
    }

    return false
}
```

在 routers 下的 api 目录新建 auth.go 文件，写入内容：

```
package api

import (
```

```
    "log"
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/astaxie/beego/validation"

    "github.com/EDDYCJY/go-gin-example/pkg/e"
    "github.com/EDDYCJY/go-gin-example/pkg/util"
    "github.com/EDDYCJY/go-gin-example/models"
)

type auth struct {
    Username string `valid:"Required; MaxSize(50)"`
    Password string `valid:"Required; MaxSize(50)"`
}

func GetAuth(c *gin.Context) {
    username := c.Query("username")
    password := c.Query("password")

    valid := validation.Validation{}
    a := auth{Username: username, Password: password}
    ok, _ := valid.Valid(&a)

    data := make(map[string]interface{})
    code := e.INVALID_PARAMS
    if ok {
        isExist := models.CheckAuth(username, password)
        if isExist {
            token, err := util.GenerateToken(username, password)
            if err != nil {
                code = e.ERROR_AUTH_TOKEN
            } else {
                data["token"] = token

                code = e.SUCCESS
            }

        } else {
            code = e.ERROR_AUTH
        }
    } else {
        for _, err := range valid.Errors {
            log.Println(err.Key, err.Message)
        }
    }
```

```
    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : data,
    })
}
```

我们打开 `routers` 目录下的 `router.go` 文件，修改文件内容（新增获取 token 的方法）：

```
package routers

import (
    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/routers/api"
    "github.com/EDDYCJY/go-gin-example/routers/api/v1"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

func InitRouter() *gin.Engine {
    r := gin.New()

    r.Use(gin.Logger())

    r.Use(gin.Recovery())

    gin.SetMode(setting.RunMode)

    r.GET("/auth", api.GetAuth)

    apiv1 := r.Group("/api/v1")
    {
        ...
    }

    return r
}
```

验证 Token

获取 `token` 的 API 方法就到这里啦，让我们来测试下是否可以正常使用吧！

重启服务后，用 `GET` 方式访问 `http://127.0.0.1:8000/auth?username=test&password=test123456` ，查看返回值是否正确


```
{
  "code": 200,
  "data": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3QiLCJwYXNzd29yZCI6InRlc3QxMjM0NTYiLCJleHAiOjE1MTg3MjAwMzcsImlzcyI6Imdpbi1ibG9nIn0.-kKOV9EO6qTHOzupQM_gHXAGDB3EJtJS4H5TTCyWwW8"
  },
  "msg": "ok"
}
```


我们有了 `token` 的 API，也调用成功了

将中间件接入 `Gin`

2、 接下来我们将中间件接入到 `Gin` 的访问流程中

我们打开 `routers` 目录下的 `router.go` 文件，修改文件内容（新增引用包和中间件引用）

```
package routers

import (
    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/routers/api"
    "github.com/EDDYCJY/go-gin-example/routers/api/v1"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
    "github.com/EDDYCJY/go-gin-example/middleware/jwt"
)

func InitRouter() *gin.Engine {
    r := gin.New()

    r.Use(gin.Logger())

    r.Use(gin.Recovery())

    gin.SetMode(setting.RunMode)

    r.GET("/auth", api.GetAuth)
```

```
    apiv1 := r.Group("/api/v1")
    apiv1.Use(jwt.JWT())
    {
        ...
    }

    return r
}
```

当前目录结构：

```
go-gin-example/
├──── conf
|    └──── app.ini
├──── main.go
├──── middleware
|    └──── jwt
|         └──── jwt.go
├──── models
|    ├──── article.go
|    ├──── auth.go
|    ├──── models.go
|    └──── tag.go
├──── pkg
|    ├──── e
|    |    ├──── code.go
|    |    └──── msg.go
|    ├──── setting
|    |    └──── setting.go
|    └──── util
|         ├──── jwt.go
|         └──── pagination.go
├──── routers
|    ├──── api
|    |    ├──── auth.go
|    |    └──── v1
|    |         ├──── article.go
|    |         └──── tag.go
|    └──── router.go
├──── runtime
```

到这里，我们的 `JWT` 编写就完成啦！

验证功能

我们来测试一下，再次访问

- http://127.0.0.1:8000/api/v1/articles
- http://127.0.0.1:8000/api/v1/articles?token=23131

正确的反馈应该是

```
{
  "code": 400,
  "data": null,
  "msg": "请求参数错误"
}

{
  "code": 20001,
  "data": null,
  "msg": "Token鉴权失败"
}
```

我们需要访问 `http://127.0.0.1:8000/auth?username=test&password=test123456` ，得到 `token`

```
{
  "code": 200,
  "data": {
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6InRlc3QiLCJwYXNzd29yZCI6InRlc3QxMjM0NTYiLCJleHAiOjE1MTg3MjQ2OTMsImlzcyI6Imdpbi1ibG9nIn0.KSBY6TeavV_30kfmP7HWLRYKP5TPEDgHtABe9HCsic4"
  },
  "msg": "ok"
}
```

再用包含 `token` 的 URL 参数去访问我们的应用 API，

访问 `http://127.0.0.1:8000/api/v1/articles?token=eyJhbGci...` ，检查接口返回值

```
{
  "code": 200,
  "data": {
    "lists": [
      {
        "id": 2,
        "created_on": 1518700920,
        "modified_on": 0,
```

```
        "tag_id": 1,
        "tag": {
          "id": 1,
          "created_on": 1518684200,
          "modified_on": 0,
          "name": "tag1",
          "created_by": "",
          "modified_by": "",
          "state": 0
        },
        "content": "test-content",
        "created_by": "test-created",
        "modified_by": "",
        "state": 0
      }
    ],
    "total": 1
  },
  "msg": "ok"
}
```


返回正确，至此我们的 `jwt-go` 在 `Gin` 中的验证就完成了！

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

编写一个简单的文件日志

涉及知识点

- 自定义 log。

本文目标

在上一节中，我们解决了 API's 可以任意访问的问题，那么我们现在还有一个问题，就是我们的日志，都是输出到控制台上的，这显然对于一个项目来说是不合理的，因此我们这一节简单封装 `log` 库，使其支持简单的文件日志！

新建 `logging` 包

我们在 `pkg` 下新建 `logging` 目录，新建 `file.go` 和 `log.go` 文件，写入内容：

编写 `file` 文件

1、 file.go：

```
package logging

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

var (
    LogSavePath = "runtime/logs/"
    LogSaveName = "log"
    LogFileExt = "log"
    TimeFormat = "20060102"
)

func getLogFilePath() string {
    return fmt.Sprintf("%s", LogSavePath)
}
```

```
func getLogFileFullPath() string {
    prefixPath := getLogFilePath()
    suffixPath := fmt.Sprintf("%s%s.%s", LogSaveName, time.Now().Format(TimeForm
at), LogFileExt)

    return fmt.Sprintf("%s%s", prefixPath, suffixPath)
}

func openLogFile(filePath string) *os.File {
    _, err := os.Stat(filePath)
    switch {
        case os.IsNotExist(err):
            mkDir()
        case os.IsPermission(err):
            log.Fatalf("Permission :%v", err)
    }

    handle, err := os.OpenFile(filePath, os.O_APPEND | os.O_CREATE | os.O_WRONL
Y, 0644)
    if err != nil {
        log.Fatalf("Fail to OpenFile :%v", err)
    }

    return handle
}

func mkDir() {
    dir, _ := os.Getwd()
    err := os.MkdirAll(dir + "/" + getLogFilePath(), os.ModePerm)
    if err != nil {
        panic(err)
    }
}
```

- `os.Stat`：返回文件信息结构描述文件。如果出现错误，会返回`*PathError`

```
type PathError struct {
    Op   string
    Path string
    Err  error
}
```

- `os.IsNotExist`：能够接受`ErrNotExist`、`syscall`的一些错误，它会返回一个布尔值，能够得知文件不存在或目录不存在
- `os.IsPermission`：能够接受`ErrPermission`、`syscall`的一些错误，它会返回一个布尔值，能够得知权限是否满足
- `os.OpenFile`：调用文件，支持传入文件名称、指定的模式调用文件、文件权限，返回的文件的方法可以用于 I/O。如果出现错误，则为`*PathError`。

```
const (
    // Exactly one of O_RDONLY, O_WRONLY, or O_RDWR must be specified.
    O_RDONLY int = syscall.O_RDONLY // 以只读模式打开文件
    O_WRONLY int = syscall.O_WRONLY // 以只写模式打开文件
    O_RDWR   int = syscall.O_RDWR   // 以读写模式打开文件
    // The remaining values may be or'ed in to control behavior.
    O_APPEND int = syscall.O_APPEND // 在写入时将数据追加到文件中
    O_CREATE int = syscall.O_CREAT  // 如果不存在，则创建一个新文件
    O_EXCL   int = syscall.O_EXCL   // 使用O_CREATE时，文件必须不存在
    O_SYNC   int = syscall.O_SYNC   // 同步IO
    O_TRUNC  int = syscall.O_TRUNC  // 如果可以，打开时
)
```

- `os.Getwd`：返回与当前目录对应的根路径名
- `os.MkdirAll`：创建对应的目录以及所需的子目录，若成功则返回`nil`，否则返回`error`
- `os.ModePerm`：`const`定义`ModePerm FileMode = 0777`

编写`log`文件

2、log.go

```
package logging

import (
    "log"
    "os"
    "runtime"
    "path/filepath"
    "fmt"
)

type Level int
```

```
var (
    F *os.File

    DefaultPrefix = ""
    DefaultCallerDepth = 2

    logger *log.Logger
    logPrefix = ""
    levelFlags = []string{"DEBUG", "INFO", "WARN", "ERROR", "FATAL"}
)

const (
    DEBUG Level = iota
    INFO
    WARNING
    ERROR
    FATAL
)

func init() {
    filePath := getLogFileFullPath()
    F = openLogFile(filePath)

    logger = log.New(F, DefaultPrefix, log.LstdFlags)
}

func Debug(v ...interface{}) {
    setPrefix(DEBUG)
    logger.Println(v)
}

func Info(v ...interface{}) {
    setPrefix(INFO)
    logger.Println(v)
}

func Warn(v ...interface{}) {
    setPrefix(WARNING)
    logger.Println(v)
}

func Error(v ...interface{}) {
    setPrefix(ERROR)
    logger.Println(v)
}
```

```
func Fatal(v ...interface{}) {
    setPrefix(FATAL)
    logger.Fatalln(v)
}

func setPrefix(level Level) {
    _, file, line, ok := runtime.Caller(DefaultCallerDepth)
    if ok {
        logPrefix = fmt.Sprintf("[%s][%s:%d]", levelFlags[level], filepath.Base
(file), line)
    } else {
        logPrefix = fmt.Sprintf("[%s]", levelFlags[level])
    }

    logger.SetPrefix(logPrefix)
}
```

- `log.New`：创建一个新的日志记录器。`out`定义要写入日志数据的`IO`句柄。`prefix`定义每个生成的日志行的开头。`flag`定义了日志记录属性

```
func New(out io.Writer, prefix string, flag int) *Logger {
    return &Logger{out: out, prefix: prefix, flag: flag}
}
```

- `log.LstdFlags`：日志记录的格式属性之一，其余的选项如下

```
const (
    Ldate         = 1 << iota     // the date in the local time zone: 2009/01/23
    Ltime                         // the time in the local time zone: 01:23:23
    Lmicroseconds                 // microsecond resolution: 01:23:23.123123.  a
ssumes Ltime.
    Llongfile                     // full file name and line number: /a/b/c/d.g
o:23
    Lshortfile                    // final file name element and line number: d.
go:23. overrides Llongfile
    LUTC                          // if Ldate or Ltime is set, use UTC rather th
an the local time zone
    LstdFlags     = Ldate | Ltime // initial values for the standard logger
)
```

当前目录结构：

```
gin-blog/
├──── conf
│    └──── app.ini
├──── main.go
├──── middleware
│    └──── jwt
│         └──── jwt.go
├──── models
│    ├──── article.go
│    ├──── auth.go
│    ├──── models.go
│    └──── tag.go
├──── pkg
│    ├──── e
│    │    ├──── code.go
│    │    └──── msg.go
│    ├──── logging
│    │    ├──── file.go
│    │    └──── log.go
│    ├──── setting
│    │    └──── setting.go
│    └──── util
│         ├──── jwt.go
│         └──── pagination.go
├──── routers
│    ├──── api
│    │    ├──── auth.go
│    │    └──── v1
│    │         ├──── article.go
│    │         └──── tag.go
│    └──── router.go
├──── runtime
```

我们自定义的 `logging` 包，已经基本完成了，接下来让它接入到我们的项目之中吧。我们打开先前包含 `log` 包的代码，如下：

1. 打开 `routers` 目录下的 `article.go` 、 `tag.go` 、 `auth.go` 。
2. 将 `log` 包的引用删除，修改引用我们自己的日志包为 `github.com/EDDYCJY/go-gin-example/pkg/logging` 。
3. 将原本的 `log.Println(...)` 改为 `logging.Info(...)` 。

例如 `auth.go` 文件的修改内容：

```
package api
```

```
import (
    "net/http"

    "github.com/gin-gonic/gin"
    "github.com/astaxie/beego/validation"

    "github.com/EDDYCJY/go-gin-example/pkg/e"
    "github.com/EDDYCJY/go-gin-example/pkg/util"
    "github.com/EDDYCJY/go-gin-example/models"
    "github.com/EDDYCJY/go-gin-example/pkg/logging"
)
...
func GetAuth(c *gin.Context) {
    ...
    code := e.INVALID_PARAMS
    if ok {
        ...
    } else {
        for _, err := range valid.Errors {
            logging.Info(err.Key, err.Message)
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code" : code,
        "msg" : e.GetMsg(code),
        "data" : data,
    })
}
```

验证功能

修改文件后，重启服务，我们来试试吧！

获取到 API 的 Token 后，我们故意传错误 URL 参数给接口，
如：`http://127.0.0.1:8000/api/v1/articles?tag_id=0&state=9999999&token=eyJhbG..`

然后我们到 `$GOPATH/gin-blog/runtime/logs` 查看日志：

```
$ tail -f log20180216.log
[INFO][article.go:79]2018/02/16 18:33:12 [state 状态只允许0或1]
[INFO][article.go:79]2018/02/16 18:33:42 [state 状态只允许0或1]
[INFO][article.go:79]2018/02/16 18:33:42 [tag_id 标签ID必须大于0]
[INFO][article.go:79]2018/02/16 18:38:39 [state 状态只允许0或1]
[INFO][article.go:79]2018/02/16 18:38:39 [tag_id 标签ID必须大于0]
```

日志结构一切正常，我们的记录模式都为 `Info` ，因此前缀是对的，并且我们是入参有问题，也把错误记录下来了，这样排错就很方便了！

至此，本节就完成了，这只是一个简单的扩展，实际上我们线上项目要使用的文件日志，是更复杂一些，开动你的大脑 举一反三吧！

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

优雅的重启服务

知识点

- 信号量的了解。
- 应用热更新。

本文目标

在前面编写案例代码时，我相信你会想到，每次更新完代码，更新完配置文件后，就直接这么 `ctrl+c` 真的没问题吗， `ctrl+c` 到底做了些什么事情呢？

在这一节中我们简单讲述 `ctrl+c` 背后的**信号**以及如何在 `Gin` 中**优雅的重启服务**，也就是对 `HTTP` 服务进行热更新。

ctrl + c

> 内核在某些情况下发送信号，比如在进程往一个已经关闭的管道写数据时会产生 `SIGPIPE` 信号

在终端执行特定的组合键可以使系统发送特定的信号给此进程，完成一系列的动作

命令	信号	含义
ctrl + c	SIGINT	强制进程结束
ctrl + z	SIGTSTP	任务中断，进程挂起
ctrl + \	SIGQUIT	进程结束 和 `dump core`
ctrl + d		EOF
	SIGHUP	终止收到该信号的进程。若程序中没有捕捉该信号， 当收到该信号时，进程就会退出（常用于 重启、重新加载进程）

因此在我们执行 `ctrl + c` 关闭 `gin` 服务端时，**会强制进程结束，导致正在访问的用户等出现问题**

常见的 `kill -9 pid` 会发送 `SIGKILL` 信号给进程，也是类似的结果

信号

本段中反复出现**信号**是什么呢？

信号是 `Unix` 、类 `Unix` 以及其他 `POSIX` 兼容的操作系统中进程间通讯的一种有限制的方式

它是一种异步的通知机制，用来提醒进程一个事件（硬件异常、程序执行异常、外部发出信号）已经发生。当一个信号发送给一个进程，操作系统中断了进程正常的控制流程。此时，任何非原子操作都将被中断。如果进程定义了信号的处理函数，那么它将被执行，否则就执行默认的处理函数

所有信号

```
$ kill -l
 1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF 28) SIGWINCH    29) SIGIO   30) SIGPWR
31) SIGSYS  34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

怎样算优雅

目的

- 不关闭现有连接（正在运行中的程序）
- 新的进程启动并替代旧进程
- 新的进程接管新的连接

- 连接要随时响应用户的请求，当用户仍在请求旧进程时要保持连接，新用户应请求新进程，不可以出现拒绝请求的情况

流程

1、替换可执行文件或修改配置文件

2、发送信号量 `SIGHUP`

3、拒绝新连接请求旧进程，但要保证已有连接正常

4、启动新的子进程

5、新的子进程开始 `Accet`

6、系统将新的请求转交新的子进程

7、旧进程处理完所有旧连接后正常结束

实现优雅重启

endless

> Zero downtime restarts for golang HTTP and HTTPS servers. (for golang 1.3+)

我们借助 fvbock/endless 来实现 `Golang HTTP/HTTPS` 服务重新启动的零停机

`endless server` 监听以下几种信号量：

- syscall.SIGHUP：触发 `fork` 子进程和重新启动
- syscall.SIGUSR1/syscall.SIGTSTP：被监听，但不会触发任何动作
- syscall.SIGUSR2：触发 `hammerTime`
- syscall.SIGINT/syscall.SIGTERM：触发服务器关闭（会完成正在运行的请求）

`endless` 正正是依靠监听这些**信号量**，完成管控的一系列动作

安装

```
go get -u github.com/fvbock/endless
```

编写

打开 gin-blog 的 `main.go` 文件，修改文件：

```
package main

import (
    "fmt"
    "log"
    "syscall"

    "github.com/fvbock/endless"

    "gin-blog/routers"
    "gin-blog/pkg/setting"
)

func main() {
    endless.DefaultReadTimeOut = setting.ReadTimeout
    endless.DefaultWriteTimeOut = setting.WriteTimeout
    endless.DefaultMaxHeaderBytes = 1 << 20
    endPoint := fmt.Sprintf(":%d", setting.HTTPPort)

    server := endless.NewServer(endPoint, routers.InitRouter())
    server.BeforeBegin = func(add string) {
        log.Printf("Actual pid is %d", syscall.Getpid())
    }

    err := server.ListenAndServe()
    if err != nil {
        log.Printf("Server err: %v", err)
    }
}
```

`endless.NewServer` 返回一个初始化的 `endlessServer` 对象，在 `BeforeBegin` 时输出当前进程的 `pid`，调用 `ListenAndServe` 将实际“启动”服务

验证

编译

```
$ go build main.go
```

执行

```
$ ./main
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
...
Actual pid is 48601
```

启动成功后，输出了 `pid` 为 48601；在另外一个终端执行 `kill -1 48601` ，检验先前服务的终端效果

```
[root@localhost go-gin-example]# ./main
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:  gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /auth                     --> ...
[GIN-debug] GET    /api/v1/tags              --> ...
...

Actual pid is 48601

...

Actual pid is 48755
48601 Received SIGTERM.
48601 [::]:8000 Listener closed.
48601 Waiting for connections to finish...
48601 Serve() returning...
Server err: accept tcp [::]:8000: use of closed network connection
```

可以看到该命令已经挂起，并且 `fork` 了新的子进程 `pid` 为 `48755`

```
48601 Received SIGTERM.
48601 [::]:8000 Listener closed.
48601 Waiting for connections to finish...
48601 Serve() returning...
Server err: accept tcp [::]:8000: use of closed network connection
```

大致意思为主进程（ `pid` 为 48601）接受到 `SIGTERM` 信号量，关闭主进程的监听并且等待正在执行的请求完成；这与我们先前的描述一致

唤醒

这时候在 `postman` 上再次访问我们的接口，你可以惊喜的发现，他“复活”了！

```
Actual pid is 48755
48601 Received SIGTERM.
48601 [::]:8000 Listener closed.
48601 Waiting for connections to finish...
48601 Serve() returning...
Server err: accept tcp [::]:8000: use of closed network connection


$ [GIN] 2018/03/15 - 13:00:16 | 200 |      188.096µs |    192.168.111.1 | GET
/api/v1/tags...
```

这就完成了一次正向的流转了

你想想，每次更新发布、或者修改配置文件等，只需要给该进程发送**SIGTERM 信号**，而不需要强制结束应用，是多么便捷又安全的事！

问题

`endless` 热更新是采取创建子进程后，将原进程退出的方式，这点不符合守护进程的要求

http.Server - Shutdown()

如果你的 `Golang >= 1.8` ，也可以考虑使用 `http.Server` 的 Shutdown 方法

```
package main

import (
    "fmt"
    "net/http"
    "context"
    "log"
    "os"
    "os/signal"
    "time"


    "gin-blog/routers"
    "gin-blog/pkg/setting"
)

func main() {
    router := routers.InitRouter()

    s := &http.Server{
        Addr:           fmt.Sprintf(":%d", setting.HTTPPort),
```

```
        Handler:        router,
        ReadTimeout:    setting.ReadTimeout,
        WriteTimeout:   setting.WriteTimeout,
        MaxHeaderBytes: 1 << 20,
    }

    go func() {
        if err := s.ListenAndServe(); err != nil {
            log.Printf("Listen: %s\n", err)
        }
    }()

    quit := make(chan os.Signal)
    signal.Notify(quit, os.Interrupt)
    <- quit

    log.Println("Shutdown Server ...")

    ctx, cancel := context.WithTimeout(context.Background(), 5 * time.Second)
    defer cancel()
    if err := s.Shutdown(ctx); err != nil {
        log.Fatal("Server Shutdown:", err)
    }

    log.Println("Server exiting")
}
```

小结

在日常的服务中，优雅的重启（热更新）是非常重要的一环。而 `Golang` 在 `HTTP` 服务方面的热更新也有不少方案了，我们应该根据实际应用场景挑选最合适的

参考

本系列示例代码

- go-gin-example

拓展阅读

- manners
- graceful

- grace
- plugin: new package for loading plugins · golang/go@0cbb12f · GitHub

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

为它加上Swagger

涉及知识点

- Swagger

本文目标

一个好的 `API's` ，必然离不开一个好的 `API` 文档，如果要开发纯手写 `API` 文档，不存在的（很难持续维护），因此我们要自动生成接口文档。

安装 swag

```
$ go get -u github.com/swaggo/swag/cmd/swag@v1.6.5
```

若 `$GOROOT/bin` 没有加入 `$PATH` 中，你需要执行将其可执行文件移动到 `$GOBIN` 下

```
mv $GOPATH/bin/swag /usr/local/go/bin
```

验证是否安装成功

检查 $GOBIN 下是否有 swag 文件，如下：

```
$ swag -v
swag version v1.6.5
```

安装 gin-swagger

```
$ go get -u github.com/swaggo/gin-swagger@v1.2.0
$ go get -u github.com/swaggo/files
$ go get -u github.com/alecthomas/template
```

注：若无科学上网，请务必配置 Go modules proxy。

初始化

编写 API 注释

`Swagger` 中需要将相应的注释或注解编写到方法上，再利用生成器自动生成说明文件

`gin-swagger` 给出的范例：

```
// @Summary Add a new pet to the store
// @Description get string by ID
// @Accept  json
// @Produce  json
// @Param   some_id     path    int     true        "Some ID"
// @Success 200 {string} string    "ok"
// @Failure 400 {object} web.APIError "We need ID!!"
// @Failure 404 {object} web.APIError "Can not find ID"
// @Router /testapi/get-string-by-int/{some_id} [get]
```

我们可以参照 `Swagger` 的注解规范和范例去编写

```
// @Summary 新增文章标签
// @Produce  json
// @Param name query string true "Name"
// @Param state query int false "State"
// @Param created_by query int false "CreatedBy"
// @Success 200 {string} json "{"code":200,"data":{},"msg":"ok"}"
// @Router /api/v1/tags [post]
func AddTag(c *gin.Context) {
```

```
// @Summary 修改文章标签
// @Produce  json
// @Param id path int true "ID"
// @Param name query string true "ID"
// @Param state query int false "State"
// @Param modified_by query string true "ModifiedBy"
// @Success 200 {string} json "{"code":200,"data":{},"msg":"ok"}"
// @Router /api/v1/tags/{id} [put]
func EditTag(c *gin.Context) {
```

参考的注解请参见 go-gin-example。以确保获取最新的 swag 语法

路由

在完成了注解的编写后，我们需要针对 swagger 新增初始化动作和对应的路由规则，才可以使用。打开 routers/router.go 文件，新增内容如下：

```
package routers

import (
    ...

    _ "github.com/EDDYCJY/go-gin-example/docs"

    ...
)

// InitRouter initialize routing information
func InitRouter() *gin.Engine {
    ...
    r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
    ...
    apiv1 := r.Group("/api/v1")
    apiv1.Use(jwt.JWT())
    {
        ...
    }

    return r
}
```

生成

我们进入到 `gin-blog` 的项目根目录中，执行初始化命令

```
[$ gin-blog]# swag init
2018/03/13 23:32:10 Generate swagger docs....
2018/03/13 23:32:10 Generate general API Info
2018/03/13 23:32:10 create docs.go at  docs/docs.go
```

完毕后会在项目根目录下生成 `docs`

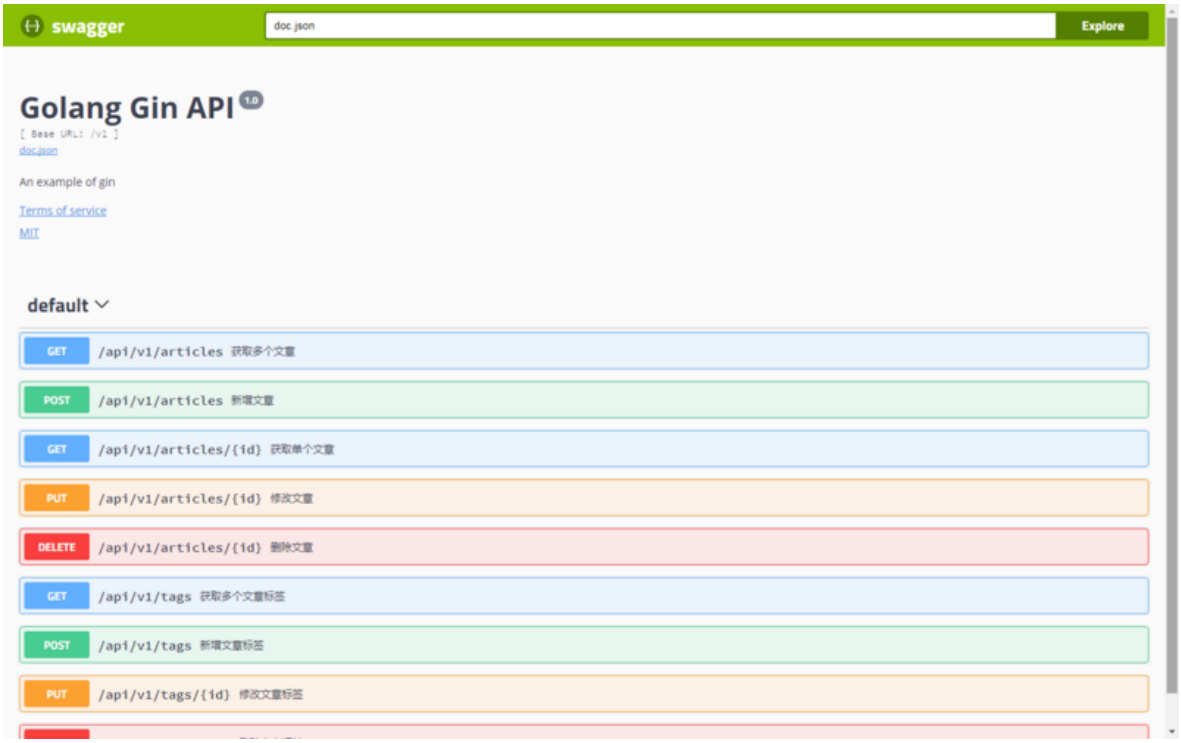
```
docs/
├─── docs.go
└─── swagger
    ├─── swagger.json
    └─── swagger.yaml
```

我们可以检查 `docs.go` 文件中的 `doc` 变量，详细记载中我们文件中所编写的注解和说明

```
// GENERATED BY THE COMMAND ABOVE; DO NOT EDIT
// This file was generated by swaggo/swag at
// 2018-03-17 21:41:17.911363936 +0800 CST m=+0.156842263

package docs

import (
    "github.com/swaggo/swag"
)

var doc = `{
    "swagger": "2.0",
    "info": {
        "description": "An example of gin",
        "title": "Golang Gin API",
        "termsOfService": "https://github.com/EDDYCJY/go-gin-example",
        "contact": {},
        "license": {
            "name": "MIT",
            "url": "https://github.com/EDDYCJY/go-gin-example/blob/master/LICENSE"
        },
        "version": "1.0"
    },
    "basePath": "/v1",
    "paths": {
        "/api/v1/articles": {
            "get": {
                "produces": [
                    "application/json"
                ],
                "summary": "获取多个文章",
                "parameters": [
                    {
                        "type": "integer",
                        "description": "TagID",
                        "name": "tag_id",
                        "in": "query"
                    },
                    {
                        "type": "integer",
                        "description": "State",
                        "name": "state",
                        "in": "query"
                    },
                    {
                        "type": "integer",
                        "description": "CreatedBy",
                        "name": "created_by",
                        "in": "query"
                    }
                ],
                "responses": {
```

验证

大功告成，访问一下 `http://127.0.0.1:8000/swagger/index.html` ， 查看 `API` 文档生成是否正确



参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

将Golang应用部署到Docker

涉及知识点

- Go + Docker

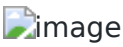
本文目标

将我们的 `go-gin-example` 应用部署到一个 Docker 里，你需要先准备好如下东西：

- 你需要安装好 `docker` 。
- 如果上外网比较吃力，需要配好镜像源。

Docker

在这里简单介绍下 Docker，建议深入学习

 image

Docker 是一个开源的轻量级容器技术，让开发者可以打包他们的应用以及应用运行的上下文环境到一个可移植的镜像中，然后发布到任何支持 Docker 的系统上运行。 通过容器技术，在几乎没有性能开销的情况下，Docker 为应用提供了一个隔离运行环境

- 简化配置
- 代码流水线管理
- 提高开发效率
- 隔离应用
- 快速、持续部署

接下来我们正式开始对项目进行 `docker` 的所需处理和编写，每一个大标题为步骤大纲

Golang

一、编写 Dockerfile

在 `go-gin-example` 项目根目录创建 Dockerfile 文件，写入内容

```
FROM golang:latest

ENV GOPROXY https://goproxy.cn,direct
WORKDIR $GOPATH/src/github.com/EDDYCJY/go-gin-example
COPY . $GOPATH/src/github.com/EDDYCJY/go-gin-example
RUN go build .

EXPOSE 8000
ENTRYPOINT ["./go-gin-example"]
```

作用

`golang:latest` 镜像为基础镜像，将工作目录设置为 `$GOPATH/src/go-gin-example` ，并将当前上下文目录的内容复制到 `$GOPATH/src/go-gin-example` 中

在进行 `go build` 编译完毕后，将容器启动程序设置为 `./go-gin-example` ，也就是我们所编译的可执行文件

注意 `go-gin-example` 在 `docker` 容器里编译，并没有在宿主机现场编译

说明

Dockerfile 文件是用于定义 Docker 镜像生成流程的配置文件，文件内容是一条条指令，每一条指令构建一层，因此每一条指令的内容，就是描述该层应当如何构建；这些指令应用于基础镜像并最终创建一个新的镜像

你可以认为用于快速创建自定义的 Docker 镜像

1、 FROM

指定基础镜像（必须有的指令，并且必须是第一条指令）

2、 WORKDIR

格式为 `WORKDIR` <工作目录路径>

使用 `WORKDIR` 指令可以来**指定工作目录**（或者称为当前目录），以后各层的当前目录就被改为指定的目录，如果目录不存在， `WORKDIR` 会帮你建立目录

3、COPY

格式：

```
COPY <源路径>... <目标路径>
COPY ["<源路径1>",... "<目标路径>"]
```

`COPY` 指令将从构建上下文目录中 <源路径> 的文件/目录**复制**到新的一层的镜像内的 <目标路径> 位置

4、RUN

用于执行命令行命令

格式：`RUN` <命令>

5、EXPOSE

格式为 `EXPOSE` <端口 1> [<端口 2>...]

`EXPOSE` 指令是**声明运行时容器提供服务端口，这只是一个声明**，在运行时并不会因为这个声明应用就会开启这个端口的服务

在 Dockerfile 中写入这样的声明有两个好处

- 帮助镜像使用者理解这个镜像服务的守护端口，以方便配置映射
- 运行时使用随机端口映射时，也就是 `docker run -P` 时，会自动随机映射 `EXPOSE` 的端口

6、ENTRYPOINT

`ENTRYPOINT` 的格式和 `RUN` 指令格式一样，分为两种格式

- `exec` 格式：

```
<ENTRYPOINT> "<CMD>"
```

- `shell` 格式：

```
ENTRYPOINT [ "curl", "-s", "http://ip.cn" ]
```

`ENTRYPOINT` 指令是**指定容器启动程序及参数**

二、构建镜像

`go-gin-example` 的项目根目录下**执行** `docker build -t gin-blog-docker .`

该命令作用是创建/构建镜像，`-t` 指定名称为 `gin-blog-docker`，`.` 构建内容为当前上下文目录

```
$ docker build -t gin-blog-docker .
Sending build context to Docker daemon 96.39 MB
Step 1/6 : FROM golang:latest
 ---> d632bbfe5767
Step 2/6 : WORKDIR $GOPATH/src/github.com/EDDYCJY/go-gin-example
 ---> 56294f978c5d
Removing intermediate container e112997b995d
Step 3/6 : COPY . $GOPATH/src/github.com/EDDYCJY/go-gin-example
 ---> 3b60960120cf
Removing intermediate container 63e310b3f60c
Step 4/6 : RUN go build .
 ---> Running in 52648a431450
go: downloading github.com/gin-gonic/gin v1.3.0
go: downloading github.com/go-ini/ini v1.32.1-0.20180214101753-32e4be5f41bb
go: downloading github.com/swaggo/gin-swagger v1.0.1-0.20190110070702-0c6fcfd3c7
f3
...
 ---> 7bfbeb301fea
Removing intermediate container 52648a431450
Step 5/6 : EXPOSE 8000
 ---> Running in 98f5b387d1bb
 ---> b65bd4076c65
Removing intermediate container 98f5b387d1bb
Step 6/6 : ENTRYPOINT ./go-gin-example
 ---> Running in c4f6cdeb667b
 ---> d8a109c7697c
Removing intermediate container c4f6cdeb667b
Successfully built d8a109c7697c
```

三、验证镜像

查看所有的镜像，确定刚刚构建的 `gin-blog-docker` 镜像是否存在

```
$ docker images
REPOSITORY              TAG                 IMAGE ID            CREATED
SIZE
gin-blog-docker         latest              d8a109c7697c        About a minute a
go    946 MB
docker.io/golang        latest              d632bbfe5767        8 days ago
779 MB
...
```

四、创建并运行一个新容器

执行命令 `docker run -p 8000:8000 gin-blog-docker`

```
$ docker run -p 8000:8000 gin-blog-docker
dial tcp 127.0.0.1:3306: connect: connection refused
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:   gin.SetMode(gin.ReleaseMode)

...
Actual pid is 1
```

运行成功，你以为大功告成了吗？

你想太多了，仔细看看控制台的输出了一条错误 `dial tcp 127.0.0.1:3306: connect: connection refused`

我们研判一下，发现是 `Mysql` 的问题，接下来第二项我们将解决这个问题

Mysql

一、拉取镜像

从 `Docker` 的公共仓库 `Dockerhub` 下载 `MySQL` 镜像（国内建议配个镜像）

```
$ docker pull mysql
```

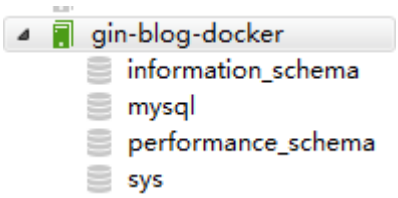
二、创建并运行一个新容器

运行 `Mysql` 容器，并设置执行成功后返回容器 ID

```
$ docker run --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=rootroot -d mysql
8c86ac986da4922492934b6fe074254c9165b8ee3e184d29865921b0fef29e64
```

连接 Mysql

初始化的 `Mysql` 应该如图



Golang + Mysql

一、删除镜像

由于原本的镜像存在问题，我们需要删除它，此处有几种做法

- 删除原本有问题的镜像，重新构建一个新镜像
- 重新构建一个不同 `name` 、 `tag` 的新镜像

删除原本的有问题的镜像， `-f` 是强制删除及其关联状态

若不执行 `-f` ，你需要执行 `docker ps -a` 查到所关联的容器，将其 `rm` 解除两者依赖关系

```
$ docker rmi -f gin-blog-docker
Untagged: gin-blog-docker:latest
Deleted: sha256:d8a109c7697c3c2d9b4de7dbb49669d10106902122817b6467a031706bc52ab4
Deleted: sha256:b65bd4076c65a3c24029ca4def3b3f37001ff7c9eca09e2590c4d29e1e23dce5
Deleted: sha256:7bfbeb301fea9d8912a4b7c43e4bb8b69bdc57f0b416b372bfb6510e476a7dee
Deleted: sha256:3b60960120cf619181c1762cdc1b8ce318b8c815e056659809252dd321bcb642
Deleted: sha256:56294f978c5dfcfa4afa8ad033fd76b755b7ecb5237c6829550741a4d2ce10bc
```

二、修改配置文件

将项目的配置文件 `conf/app.ini` ，内容修改为

```
#debug or release
RUN_MODE = debug

[app]
PAGE_SIZE = 10
JWT_SECRET = 233

[server]
HTTP_PORT = 8000
READ_TIMEOUT = 60
WRITE_TIMEOUT = 60

[database]
TYPE = mysql
USER = root
PASSWORD = rootroot
HOST = mysql:3306
```

```
NAME = blog
TABLE_PREFIX = blog_
```

三、重新构建镜像

重复先前的步骤，回到 `gin-blog` 的项目根目录下**执行** `docker build -t gin-blog-docker .`

四、创建并运行一个新容器

关联

Q：我们需要将 `Golang` 容器和 `Mysql` 容器关联起来，那么我们需要怎么做呢？

A：增加命令 `--link mysql:mysql` 让 `Golang` 容器与 `Mysql` 容器互联；通过 `--link` ，**可以在容器内直接使用其关联的容器别名进行访问**，而不通过 IP，但是 `--link` 只能解决单机容器间的关联，在分布式多机的情况下，需要通过别的方式进行连接

运行

执行命令 `docker run --link mysql:mysql -p 8000:8000 gin-blog-docker`

```
$ docker run --link mysql:mysql -p 8000:8000 gin-blog-docker
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:    export GIN_MODE=release
 - using code:    gin.SetMode(gin.ReleaseMode)
...
Actual pid is 1
```

结果

检查启动输出、接口测试、数据库内数据，均正常；我们的 `Golang` 容器和 `Mysql` 容器成功关联运行，大功告成 :)

Review

思考

虽然应用已经能够跑起来了

但如果对 `Golang` 和 `Docker` 有一定的了解，我希望你能够想到至少 2 个问题

- 为什么 `gin-blog-docker` 占用空间这么大？（可用 `docker ps -as | grep gin-blog-docker` 查看）
- `Mysql` 容器直接这么使用，数据存储到哪里去了？

创建超小的 Golang 镜像

Q：第一个问题，为什么这么镜像体积这么大？

A：`FROM golang:latest` 拉取的是官方 `golang` 镜像，包含 Golang 的编译和运行环境，外加一堆 GCC、build 工具，相当齐全

这是有问题的，**我们可以不在 Golang 容器中现场编译的**，压根用不到那些东西，我们只需要一个能够运行可执行文件的环境即可

构建 Scratch 镜像

Scratch 镜像，简洁、小巧，基本是个空镜像

一、修改 Dockerfile

```
FROM scratch

WORKDIR $GOPATH/src/github.com/EDDYCJY/go-gin-example
COPY . $GOPATH/src/github.com/EDDYCJY/go-gin-example

EXPOSE 8000
CMD ["./go-gin-example"]
```

二、编译可执行文件

```
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o go-gin-example .
```

编译所生成的可执行文件会依赖一些库，并且是动态链接。在这里因为使用的是 `scratch` 镜像，它是空镜像，因此我们需要将生成的可执行文件静态链接所依赖的库

三、构建镜像

```
$ docker build -t gin-blog-docker-scratch .
Sending build context to Docker daemon 133.1 MB
Step 1/5 : FROM scratch
 --->
Step 2/5 : WORKDIR $GOPATH/src/github.com/EDDYCJY/go-gin-example
```

```
 ---> Using cache
 ---> ee07e166a638
Step 3/5 : COPY . $GOPATH/src/github.com/EDDYCJY/go-gin-example
 ---> 1489a0693d51
Removing intermediate container e3e9efc0fe4d
Step 4/5 : EXPOSE 8000
 ---> Running in b5630de5544a
 ---> 6993e9f8c944
Removing intermediate container b5630de5544a
Step 5/5 : CMD ./go-gin-example
 ---> Running in eebc0d8628ae
 ---> 5310bebeb86a
Removing intermediate container eebc0d8628ae
Successfully built 5310bebeb86a
```

注意，假设你的 Golang 应用没有依赖任何的配置等文件，是可以直接把可执行文件给拷贝进去即可，其他都不必关心

这里可以有好几种解决方案

- 依赖文件统一管理挂载
- go-bindata 一下

...

因此这里如果**解决了文件依赖的问题**后，就不需要把目录给 `COPY` 进去了

四、运行

```
$ docker run --link mysql:mysql -p 8000:8000 gin-blog-docker-scratch
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:   export GIN_MODE=release
 - using code:   gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /auth                     --> github.com/EDDYCJY/go-gin-examp
le/routers/api.GetAuth (3 handlers)
...
```

成功运行，程序也正常接收请求

接下来我们再看看占用大小，执行 `docker ps -as` 命令

```
$ docker ps -as
CONTAINER ID        IMAGE                     COMMAND                  ...
```

```
SIZE
9ebdba5a8445        gin-blog-docker-scratch    "./go-gin-example"        ...      0
B (virtual 132 MB)
427ee79e6857        gin-blog-docker            "./go-gin-example"        ...      0
B (virtual 946 MB)
```

从结果而言，占用大小以 `Scratch` 镜像为基础的容器完胜，完成目标

Mysql 挂载数据卷

倘若不做任何干涉，在每次启动一个 `Mysql` 容器时，数据库都是空的。另外容器删除之后，数据就丢失了（还有各类意外情况），非常糟糕！

数据卷

数据卷 是被设计用来持久化数据的，它的生命周期独立于容器，Docker 不会在容器被删除后自动删除 数据卷，并且也不存在垃圾回收这样的机制来处理没有任何容器引用的 数据卷。如果需要在删除容器的同时移除数据卷。可以在删除容器的时候使用 `docker rm -v` 这个命令

数据卷 是一个可供一个或多个容器使用的特殊目录，它绕过 UFS，可以提供很多有用的特性：

- 数据卷 可以在容器之间共享和重用
- 对 数据卷 的修改会立马生效
- 对 数据卷 的更新，不会影响镜像
- 数据卷 默认会一直存在，即使容器被删除

> 注意：数据卷 的使用，类似于 Linux 下对目录或文件进行 mount，镜像中的被指定为挂载点的目录中的文件会隐藏掉，能显示看的是挂载的 数据卷。

如何挂载

首先创建一个目录用于存放数据卷；示例目录 `/data/docker-mysql` ，注意 `--name` 原本名称为 `mysql` 的容器，需要将其删除 `docker rm`

```
$ docker run --name mysql -p 3306:3306 -e MYSQL_ROOT_PASSWORD=rootroot -v /data/
docker-mysql:/var/lib/mysql -d mysql
54611dbcd62eca33fb320f3f624c7941f15697d998f40b24ee535a1acf93ae72
```

创建成功，检查目录 `/data/docker-mysql` ，下面多了不少数据库文件

验证

接下来交由你进行验证，目标是创建一些测试表和数据，然后删除当前容器，重新创建的容器，数据库数据也依然存在（当然了数据卷指向要一致）

我已验证完毕，你呢？

参考

本系列示例代码

- go-gin-example

书籍

- Docker —— 从入门到实践

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

定制 GORM Callbacks

涉及知识点

- GORM

本文目标

> GORM itself is powered by Callbacks, so you could fully customize GORM as you want

GORM 本身是由回调驱动的，所以我们可以根据需要完全定制 GORM，以此达到我们的目的，如下：

- 注册一个新的回调
- 删除现有的回调
- 替换现有的回调
- 注册回调的顺序

在 GORM 中包含以上四类 Callbacks，我们结合项目选用“替换现有的回调”来解决一个小痛点。

问题

在 models 目录下，我们包含 tag.go 和 article.go 两个文件，他们有一个问题，就是 BeforeCreate、BeforeUpdate 重复出现了，那难道 100 个文件，就要写一百次吗？

1、tag.go

```
func (tag *Tag) BeforeCreate(scope *gorm.Scope) error {
    scope.SetColumn( column: "CreatedOn", time.Now().Unix())

    return nil
}

func (tag *Tag) BeforeUpdate(scope *gorm.Scope) error {
    scope.SetColumn( column: "ModifiedOn", time.Now().Unix())

    return nil
}
```

2、article.go

```
func (article *Article) BeforeCreate(scope *gorm.Scope) error {
    scope.SetColumn( column: "CreatedOn", time.Now().Unix())

    return nil
}

func (article *Article) BeforeUpdate(scope *gorm.Scope) error {
    scope.SetColumn( column: "ModifiedOn", time.Now().Unix())

    return nil
}
```

显然这是不可能的，如果先前你已经意识到这个问题，那挺 OK，但没有的话，现在开始就要改

解决

在这里我们通过 Callbacks 来实现功能，不需要一个个文件去编写

实现 Callbacks

打开 models 目录下的 models.go 文件，实现以下两个方法：

1、updateTimeStampForCreateCallback

```
// updateTimeStampForCreateCallback will set `CreatedOn`, `ModifiedOn` when crea
ting
func updateTimeStampForCreateCallback(scope *gorm.Scope) {
    if !scope.HasError() {
        nowTime := time.Now().Unix()
        if createTimeField, ok := scope.FieldByName("CreatedOn"); ok {
            if createTimeField.IsBlank {
                createTimeField.Set(nowTime)
```

```
            }
        }

        if modifyTimeField, ok := scope.FieldByName("ModifiedOn"); ok {
            if modifyTimeField.IsBlank {
                modifyTimeField.Set(nowTime)
            }
        }
    }
}
```

在这段方法中，会完成以下功能

- 检查是否有含有错误（db.Error）
- `scope.FieldByName` 通过 `scope.Fields()` 获取所有字段，判断当前是否包含所需字段

```
for _, field := range scope.Fields() {
    if field.Name == name || field.DBName == name {
        return field, true
    }
    if field.DBName == dbName {
        mostMatchedField = field
    }
}
```

- `field.IsBlank` 可判断该字段的值是否为空

```
func isBlank(value reflect.Value) bool {
    switch value.Kind() {
    case reflect.String:
        return value.Len() == 0
    case reflect.Bool:
        return !value.Bool()
    case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
        return value.Int() == 0
    case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Ui
nt64, reflect.Uintptr:
        return value.Uint() == 0
    case reflect.Float32, reflect.Float64:
        return value.Float() == 0
    case reflect.Interface, reflect.Ptr:
        return value.IsNil()
```

```
    }

    return reflect.DeepEqual(value.Interface(), reflect.Zero(value.Type()).Interface())
}
```

- 若为空则 `field.Set` 用于给该字段设置值，参数为 `interface{}`

2、updateTimeStampForUpdateCallback

```
// updateTimeStampForUpdateCallback will set `ModifyTime` when updating
func updateTimeStampForUpdateCallback(scope *gorm.Scope) {
    if _, ok := scope.Get("gorm:update_column"); !ok {
        scope.SetColumn("ModifiedOn", time.Now().Unix())
    }
}
```

- `scope.Get(...)` 根据入参获取设置了字面值的参数，例如本文中是 `gorm:update_column` ，它会去查找含这个字面值的字段属性
- `scope.SetColumn(...)` 假设没有指定 `update_column` 的字段，我们默认在更新回调设置 `ModifiedOn` 的值

注册 Callbacks

在上面小节我已经把回调方法编写好了，接下来需要将其注册进 GORM 的钩子里，但其本身自带 Create 和 Update 回调，因此调用替换即可

在 models.go 的 init 函数中，增加以下语句

```
db.Callback().Create().Replace("gorm:update_time_stamp", updateTimeStampForCreateCallback)
db.Callback().Update().Replace("gorm:update_time_stamp", updateTimeStampForUpdateCallback)
```

验证

访问 AddTag 接口，成功后检查数据库，可发现 `created_on` 和 `modified_on` 字段都为当前执行时间

访问 EditTag 接口，可发现 `modified_on` 为最后一次执行更新的时间

拓展

我们想到，在实际项目中硬删除是较少存在的，那么是否可以通过 Callbacks 来完成这个功能呢？

答案是可以的，我们在先前 `Model struct` 增加 `DeletedOn` 变量

```
type Model struct {
    ID int `gorm:"primary_key" json:"id"`
    CreatedOn int `json:"created_on"`
    ModifiedOn int `json:"modified_on"`
    DeletedOn int `json:"deleted_on"`
}
```

实现 Callbacks

打开 models 目录下的 models.go 文件，实现以下方法：

```
func deleteCallback(scope *gorm.Scope) {
    if !scope.HasError() {
        var extraOption string
        if str, ok := scope.Get("gorm:delete_option"); ok {
            extraOption = fmt.Sprint(str)
        }

        deletedOnField, hasDeletedOnField := scope.FieldByName("DeletedOn")

        if !scope.Search.Unscoped && hasDeletedOnField {
            scope.Raw(fmt.Sprintf(
                "UPDATE %v SET %v=%v%v%v",
                scope.QuotedTableName(),
                scope.Quote(deletedOnField.DBName),
                scope.AddToVars(time.Now().Unix()),
                addExtraSpaceIfExist(scope.CombinedConditionSql()),
                addExtraSpaceIfExist(extraOption),
            )).Exec()
        } else {
            scope.Raw(fmt.Sprintf(
                "DELETE FROM %v%v%v",
                scope.QuotedTableName(),
                addExtraSpaceIfExist(scope.CombinedConditionSql()),
                addExtraSpaceIfExist(extraOption),
            )).Exec()
        }
    }
```

```
}

func addExtraSpaceIfExist(str string) string {
    if str != "" {
        return " " + str
    }
    return ""
}
```

- `scope.Get("gorm:delete_option")` 检查是否手动指定了 delete_option
- `scope.FieldByName("DeletedOn")` 获取我们约定的删除字段，若存在则 `UPDATE` 软删除，若不存在则 `DELETE` 硬删除
- `scope.QuotedTableName()` 返回引用的表名，这个方法 GORM 会根据自身逻辑对表名进行一些处理
- `scope.CombinedConditionSql()` 返回组合好的条件 SQL，看一下方法原型很明了

```
func (scope *Scope) CombinedConditionSql() string {
    joinSQL := scope.joinsSQL()
    whereSQL := scope.whereSQL()
    if scope.Search.raw {
        whereSQL = strings.TrimSuffix(strings.TrimPrefix(whereSQL, "WHERE ("),
")")
    }
    return joinSQL + whereSQL + scope.groupSQL() +
        scope.havingSQL() + scope.orderSQL() + scope.limitAndOffsetSQL()
}
```

- `scope.AddToVars` 该方法可以添加值作为 SQL 的参数，也可用于防范 SQL 注入

```
func (scope *Scope) AddToVars(value interface{}) string {
    _, skipBindVar := scope.InstanceGet("skip_bindvar")

    if expr, ok := value.(*expr); ok {
        exp := expr.expr
        for _, arg := range expr.args {
            if skipBindVar {
                scope.AddToVars(arg)
            } else {
                exp = strings.Replace(exp, "?", scope.AddToVars(arg), 1)
            }
        }
        return exp
```

```
    }

    scope.SQLVars = append(scope.SQLVars, value)

    if skipBindVar {
        return "?"
    }
    return scope.Dialect().BindVar(len(scope.SQLVars))
}
```

注册 Callbacks

在 models.go 的 init 函数中，增加以下删除的回调

```
db.Callback().Delete().Replace("gorm:delete", deleteCallback)
```

验证

重启服务，访问 DeleteTag 接口，成功后即可发现 deleted_on 字段有值

小结

在这一章节中，我们结合 GORM 完成了新增、更新、查询的 Callbacks，在实际项目中常常也是这么使用

毕竟，一个钩子的事，就没有必要自己手写过多不必要的代码了

（注意，增加了软删除后，先前的代码需要增加 `deleted_on` 的判断）

参考

本系列示例代码

- go-gin-example

文档

- gorm

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

Cron定时任务

知识点

- 完成定时任务的功能

本文目标

在实际的应用项目中，定时任务的使用是很常见的。你是否有过 Golang 如何做定时任务的疑问，莫非是轮询，在本文中我们将结合我们的项目讲述 Cron。

介绍

我们将使用 cron 这个包，它实现了 cron 规范解析器和任务运行器，简单来讲就是包含了定时任务所需的功能

Cron 表达式格式

字段名	是否必填	允许的值	允许的特殊字符
秒（Seconds）	Yes	0-59	* / , -
分（Minutes）	Yes	0-59	* / , -
时（Hours）	Yes	0-23	* / , -
一个月中的某天（Day of month）	Yes	1-31	* / , - ?
月（Month）	Yes	1-12 or JAN-DEC	* / , -
星期几（Day of week）	Yes	0-6 or SUN-SAT	* / , - ?

Cron 表达式表示一组时间，使用 6 个空格分隔的字段

可以留意到 Golang 的 Cron 比 Crontab 多了一个秒级，以后遇到秒级要求的时候就省事了

Cron 特殊字符

1、星号（*）

星号表示将匹配字段的所有值

2、斜线（/）

斜线用户 描述范围的增量，表现为“N-MAX/x”，first-last/x 的形式，例如 3-59/15 表示此时的第三分钟和此后的每 15 分钟，到 59 分钟为止。即从 N 开始，使用增量直到该特定范围结束。它不会重复

3、逗号（,）

逗号用于分隔列表中的项目。例如，在 Day of week 使用“MON，WED，FRI”将意味着星期一，星期三和星期五

4、连字符（-）

连字符用于定义范围。例如，9 - 17 表示从上午 9 点到下午 5 点的每个小时

5、问号（?）

不指定值，用于代替“*”，类似“_”的存在，不难理解

预定义的 Cron 时间表

输入	简述	相当于
@yearly (or @annually)	1 月 1 日午夜运行一次	0 0 0 1 1 *
@monthly	每个月的午夜，每个月的第一个月运行一次	0 0 0 1 * *
@weekly	每周一次，周日午夜运行一次	0 0 0 * * 0
@daily (or @midnight)	每天午夜运行一次	0 0 0 * * *
@hourly	每小时运行一次	0 0 * * * *

安装

```
$ go get -u github.com/robfig/cron
```

实践

在上一章节 Gin 实践 连载十 定制 GORM Callbacks 中，我们使用了 GORM 的回调实现了软删除，同时也引入了另外一个问题

就是我怎么硬删除，我什么时候硬删除？这个往往与业务场景有关系，大致为

- 另外有一套硬删除接口
- 定时任务清理（或转移、backup）无效数据

在这里我们选用第二种解决方案来进行实践

编写硬删除代码

打开 models 目录下的 tag.go、article.go 文件，分别添加以下代码

1、tag.go

```
func CleanAllTag() bool {
    db.Unscoped().Where("deleted_on != ? ", 0).Delete(&Tag{})

    return true
}
```

2、article.go

```
func CleanAllArticle() bool {
    db.Unscoped().Where("deleted_on != ? ", 0).Delete(&Article{})

    return true
}
```

注意硬删除要使用 `Unscoped()` ，这是 GORM 的约定

编写 Cron

在 项目根目录下新建 cron.go 文件，用于编写定时任务的代码，写入文件内容

```
package main

import (
    "time"
    "log"

    "github.com/robfig/cron"
```

```
    "github.com/EDDYCJY/go-gin-example/models"
)

func main() {
    log.Println("Starting...")

    c := cron.New()
    c.AddFunc("* * * * * *", func() {
        log.Println("Run models.CleanAllTag...")
        models.CleanAllTag()
    })
    c.AddFunc("* * * * * *", func() {
        log.Println("Run models.CleanAllArticle...")
        models.CleanAllArticle()
    })

    c.Start()

    t1 := time.NewTimer(time.Second * 10)
    for {
        select {
        case <-t1.C:
            t1.Reset(time.Second * 10)
        }
    }
}
```

在这段程序中，我们做了如下的事情

cron.New()

会根据本地时间创建一个新（空白）的 Cron job runner

```
func New() *Cron {
    return NewWithLocation(time.Now().Location())
}

// NewWithLocation returns a new Cron job runner.
func NewWithLocation(location *time.Location) *Cron {
    return &Cron{
        entries:  nil,
        add:      make(chan *Entry),
        stop:     make(chan struct{}),
        snapshot: make(chan []*Entry),
        running:  false,
```

```
        ErrorLog: nil,
        location: location,
    }
}
```

c.AddFunc()

AddFunc 会向 Cron job runner 添加一个 func ，以按给定的时间表运行

```
func (c *Cron) AddJob(spec string, cmd Job) error {
    schedule, err := Parse(spec)
    if err != nil {
        return err
    }
    c.Schedule(schedule, cmd)
    return nil
}
```

会首先解析时间表，如果填写有问题会直接 err，无误则将 func 添加到 Schedule 队列中等待执行

```
func (c *Cron) Schedule(schedule Schedule, cmd Job) {
    entry := &Entry{
        Schedule: schedule,
        Job:      cmd,
    }
    if !c.running {
        c.entries = append(c.entries, entry)
        return
    }

    c.add <- entry
}
```

3、c.Start()

在当前执行的程序中启动 Cron 调度程序。其实这里的主体是 goroutine + for + select + timer 的调度控制哦

```
func (c *Cron) Run() {
    if c.running {
        return
    }
    c.running = true
```

```
    c.run()
}
```

time.NewTimer + for + select + t1.Reset

如果你是初学者，大概会有疑问，这是干嘛用的？

*（1）time.NewTimer *

会创建一个新的定时器，持续你设定的时间 d 后发送一个 channel 消息

（2）for + select

阻塞 select 等待 channel

（3）t1.Reset

会重置定时器，让它重新开始计时

注：本文适用于 “t.C 已经取走，可直接使用 Reset”。

总的来说，这段程序是为了阻塞主程序而编写的，希望你带着疑问来想，有没有别的办法呢？

有的，你直接 `select{}` 也可以完成这个需求 :)

验证

```
$ go run cron.go
2018/04/29 17:03:34 [info] replacing callback `gorm:update_time_stamp` from /Use
rs/eddycjy/go/src/github.com/EDDYCJY/go-gin-example/models/models.go:56
2018/04/29 17:03:34 [info] replacing callback `gorm:update_time_stamp` from /Use
rs/eddycjy/go/src/github.com/EDDYCJY/go-gin-example/models/models.go:57
2018/04/29 17:03:34 [info] replacing callback `gorm:delete` from /Users/eddycjy/
go/src/github.com/EDDYCJY/go-gin-example/models/models.go:58
2018/04/29 17:03:34 Starting...
2018/04/29 17:03:35 Run models.CleanAllArticle...
2018/04/29 17:03:35 Run models.CleanAllTag...
2018/04/29 17:03:36 Run models.CleanAllArticle...
2018/04/29 17:03:36 Run models.CleanAllTag...
2018/04/29 17:03:37 Run models.CleanAllTag...
2018/04/29 17:03:37 Run models.CleanAllArticle...
```

检查输出日志正常，模拟已软删除的数据，定时任务工作 OK

小结

定时任务很常见，希望你通过本文能够熟知 Golang 怎么实现一个简单的定时任务调度管理

可以不依赖系统的 Crontab 设置，指不定哪一天就用上了呢

问题

如果你手动修改计算机的系统时间，是会导致定时任务错乱的，所以一般不要乱来。

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 02 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

优化配置结构及实现图片上传

知识点

- 重构、调整结构

本文目标

这个应用程序跑了那么久了，越来越大，越来越壮，仿佛我们的产品一样，现在它需要进行小范围重构了，以便于后续的使用，这非常重要。

前言

一天，产品经理突然跟你说文章列表，没有封面图，不够美观，！）&￥！&）#&￥！加一个吧，几分钟的事

你打开你的程序，分析了一波写了个清单：

- 优化配置结构（因为配置项越来越多）
- 抽离 原 logging 的 File 便于公用（logging、upload 各保有一份并不合适）
- 实现上传图片接口（需限制文件格式、大小）
- 修改文章接口（需支持封面地址参数）
- 增加 blog_article （文章）的数据库字段
- 实现 http.FileServer

嗯，你发现要较优的话，需要调整部分的应用程序结构，因为功能越来越多，原本的设计也要跟上节奏

也就是在适当的时候，及时优化

优化配置结构

一、讲解

在先前章节中，采用了直接读取 KEY 的方式去存储配置项，而本次需求中，需要增加图片的配置项，总体就有些冗余了

我们采用以下解决方法：

- 映射结构体：使用 MapTo 来设置配置参数
- 配置统管：所有的配置项统管到 setting 中

映射结构体（示例）

在 go-ini 中可以采用 MapTo 的方式来映射结构体，例如：

```
type Server struct {
    RunMode string
    HttpPort int
    ReadTimeout time.Duration
    WriteTimeout time.Duration
}

var ServerSetting = &Server{}

func main() {
    Cfg, err := ini.Load("conf/app.ini")
    if err != nil {
        log.Fatalf("Fail to parse 'conf/app.ini': %v", err)
    }

    err = Cfg.Section("server").MapTo(ServerSetting)
    if err != nil {
        log.Fatalf("Cfg.MapTo ServerSetting err: %v", err)
    }
}
```

在这段代码中，可以注意 ServerSetting 取了地址，为什么 MapTo 必须地址入参呢？

```
// MapTo maps section to given struct.
func (s *Section) MapTo(v interface{}) error {
    typ := reflect.TypeOf(v)
    val := reflect.ValueOf(v)
    if typ.Kind() == reflect.Ptr {
        typ = typ.Elem()
        val = val.Elem()
    } else {
        return errors.New("cannot map to non-pointer struct")
    }

    return s.mapTo(val, false)
}
```

在 MapTo 中 `typ.Kind() == reflect.Ptr` 约束了必须使用指针，否则会返回 `cannot map to non-pointer struct` 的错误。这个是表面原因

更往内探究，可以认为是 `field.Set` 的原因，当执行 `val := reflect.ValueOf(v)`，函数通过传递 `v` 拷贝创建了 `val`，但是 `val` 的改变并不能更改原始的 `v`，要想 `val` 的更改能作用到 `v`，则必须传递 `v` 的地址

显然 go-ini 里也是包含修改原始值这一项功能的，你觉得是什么原因呢？

配置统管

在先前的版本中，models 和 file 的配置是在自己的文件中解析的，而其他在 setting.go 中，因此我们需要将其在 setting 中统一接管

你可能会想，直接把两者的配置项复制粘贴到 setting.go 的 init 中，一下子就完事了，搞那么麻烦？

但你在想想，先前的代码中存在多个 init 函数，执行顺序存在问题，无法达到我们的要求，你可以试试

（此处是一个基础知识点）

在 Go 中，当存在多个 init 函数时，执行顺序为：

- 相同包下的 init 函数：按照源文件编译顺序决定执行顺序（默认按文件名排序）
- 不同包下的 init 函数：按照包导入的依赖关系决定先后顺序

所以要避免多 init 的情况，**尽量由程序把控初始化的先后顺序**

二、落实

修改配置文件

打开 conf/app.ini 将配置文件修改为大驼峰命名，另外我们增加了 5 个配置项用于上传图片的功能，4 个文件日志方面的配置项

```
[app]
PageSize = 10
JwtSecret = 233

RuntimeRootPath = runtime/

ImagePrefixUrl = http://127.0.0.1:8000
ImageSavePath = upload/images/
# MB
```

```
ImageMaxSize = 5
ImageAllowExts = .jpg,.jpeg,.png

LogSavePath = logs/
LogSaveName = log
LogFileExt = log
TimeFormat = 20060102

[server]
#debug or release
RunMode = debug
HttpPort = 8000
ReadTimeout = 60
WriteTimeout = 60

[database]
Type = mysql
User = root
Password = rootroot
Host = 127.0.0.1:3306
Name = blog
TablePrefix = blog_
```

优化配置读取及设置初始化顺序

第一步

将散落在其他文件里的配置都删掉，**统一在 setting 中处理**以及**修改 init 函数为 Setup 方法**

打开 pkg/setting/setting.go 文件，修改如下：

```
package setting

import (
    "log"
    "time"

    "github.com/go-ini/ini"
)

type App struct {
    JwtSecret string
    PageSize int
    RuntimeRootPath string
```

```
    ImagePrefixUrl string
    ImageSavePath string
    ImageMaxSize int
    ImageAllowExts []string

    LogSavePath string
    LogSaveName string
    LogFileExt string
    TimeFormat string
}

var AppSetting = &App{}

type Server struct {
    RunMode string
    HttpPort int
    ReadTimeout time.Duration
    WriteTimeout time.Duration
}

var ServerSetting = &Server{}

type Database struct {
    Type string
    User string
    Password string
    Host string
    Name string
    TablePrefix string
}

var DatabaseSetting = &Database{}

func Setup() {
    Cfg, err := ini.Load("conf/app.ini")
    if err != nil {
        log.Fatalf("Fail to parse 'conf/app.ini': %v", err)
    }

    err = Cfg.Section("app").MapTo(AppSetting)
    if err != nil {
        log.Fatalf("Cfg.MapTo AppSetting err: %v", err)
    }

    AppSetting.ImageMaxSize = AppSetting.ImageMaxSize * 1024 * 1024
```

```
    err = Cfg.Section("server").MapTo(ServerSetting)
    if err != nil {
        log.Fatalf("Cfg.MapTo ServerSetting err: %v", err)
    }

    ServerSetting.ReadTimeout = ServerSetting.ReadTimeout * time.Second
    ServerSetting.WriteTimeout = ServerSetting.WriteTimeout * time.Second

    err = Cfg.Section("database").MapTo(DatabaseSetting)
    if err != nil {
        log.Fatalf("Cfg.MapTo DatabaseSetting err: %v", err)
    }
}
```

在这里，我们做了如下几件事：

- 编写与配置项保持一致的结构体（App、Server、Database）
- 使用 MapTo 将配置项映射到结构体上
- 对一些需特殊设置的配置项进行再赋值

需要你去做的事：

- 将 models.go、setting.go、pkg/logging/log.go 的 init 函数修改为 Setup 方法
- 将 models/models.go 独立读取的 DB 配置项删除，改为统一读取 setting
- 将 pkg/logging/file 独立的 LOG 配置项删除，改为统一读取 setting

这几项比较基础，并没有贴出来，我希望你可以自己动手，有问题的话可右拐 项目地址

第二步

在这一步我们要设置初始化的流程，打开 main.go 文件，修改内容：

```
func main() {
    setting.Setup()
    models.Setup()
    logging.Setup()

    endless.DefaultReadTimeOut = setting.ServerSetting.ReadTimeout
    endless.DefaultWriteTimeOut = setting.ServerSetting.WriteTimeout
    endless.DefaultMaxHeaderBytes = 1 << 20
    endPoint := fmt.Sprintf(":%d", setting.ServerSetting.HttpPort)

    server := endless.NewServer(endPoint, routers.InitRouter())
    server.BeforeBegin = func(add string) {
```

```
        log.Printf("Actual pid is %d", syscall.Getpid())
    }

    err := server.ListenAndServe()
    if err != nil {
        log.Printf("Server err: %v", err)
    }
}
```

修改完毕后，就成功将多模块的初始化函数放到启动流程中了（先后顺序也可以控制）

验证

在这里为止，针对本需求的配置优化就完毕了，你需要执行 `go run main.go` 验证一下你的功能是否正常哦

顺带留个基础问题，大家可以思考下

```
ServerSetting.ReadTimeout = ServerSetting.ReadTimeout * time.Second
ServerSetting.WriteTimeout = ServerSetting.ReadTimeout * time.Second
```

若将 setting.go 文件中的这两行删除，会出现什么问题，为什么呢？

抽离 File

在先前版本中，在 logging/file.go 中使用到了 os 的一些方法，我们通过前期规划发现，这部分在上传图片功能中可以复用

第一步

在 pkg 目录下新建 file/file.go ，写入文件内容如下：

```
package file

import (
    "os"
    "path"
    "mime/multipart"
    "io/ioutil"
)

func GetSize(f multipart.File) (int, error) {
    content, err := ioutil.ReadAll(f)
```

```
    return len(content), err
}

func GetExt(fileName string) string {
    return path.Ext(fileName)
}

func CheckNotExist(src string) bool {
    _, err := os.Stat(src)

    return os.IsNotExist(err)
}

func CheckPermission(src string) bool {
    _, err := os.Stat(src)

    return os.IsPermission(err)
}

func IsNotExistMkDir(src string) error {
    if notExist := CheckNotExist(src); notExist == true {
        if err := MkDir(src); err != nil {
            return err
        }
    }

    return nil
}

func MkDir(src string) error {
    err := os.MkdirAll(src, os.ModePerm)
    if err != nil {
        return err
    }

    return nil
}

func Open(name string, flag int, perm os.FileMode) (*os.File, error) {
    f, err := os.OpenFile(name, flag, perm)
    if err != nil {
        return nil, err
    }

    return f, nil
}
```

在这里我们一共封装了 7 个 方法

- GetSize：获取文件大小
- GetExt：获取文件后缀
- CheckNotExist：检查文件是否存在
- CheckPermission：检查文件权限
- IsNotExistMkDir：如果不存在则新建文件夹
- MkDir：新建文件夹
- Open：打开文件

在这里我们用到了 `mime/multipart` 包，它主要实现了 MIME 的 multipart 解析，主要适用于 HTTP 和常见浏览器生成的 multipart 主体

multipart 又是什么，rfc2388 的 multipart/form-data 了解一下

第二步

我们在第一步已经将 file 重新封装了一层，在这一步我们将原先 logging 包的方法都修改掉

1、打开 pkg/logging/file.go 文件，修改文件内容：

```
package logging

import (
    "fmt"
    "os"
    "time"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
    "github.com/EDDYCJY/go-gin-example/pkg/file"
)

func getLogFilePath() string {
    return fmt.Sprintf("%s%s", setting.AppSetting.RuntimeRootPath, setting.AppSe
tting.LogSavePath)
}

func getLogFileName() string {
    return fmt.Sprintf("%s%s.%s",
        setting.AppSetting.LogSaveName,
        time.Now().Format(setting.AppSetting.TimeFormat),
        setting.AppSetting.LogFileExt,
```

```
    )
}

func openLogFile(fileName, filePath string) (*os.File, error) {
    dir, err := os.Getwd()
    if err != nil {
        return nil, fmt.Errorf("os.Getwd err: %v", err)
    }

    src := dir + "/" + filePath
    perm := file.CheckPermission(src)
    if perm == true {
        return nil, fmt.Errorf("file.CheckPermission Permission denied src: %s",
src)
    }

    err = file.IsNotExistMkDir(src)
    if err != nil {
        return nil, fmt.Errorf("file.IsNotExistMkDir src: %s, err: %v", src, er
r)
    }

    f, err := file.Open(src + fileName, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 064
4)
    if err != nil {
        return nil, fmt.Errorf("Fail to OpenFile :%v", err)
    }

    return f, nil
}
```

我们将引用都改为了 file/file.go 包里的方法

2、打开 pkg/logging/log.go 文件，修改文件内容:

```
package logging

...

func Setup() {
    var err error
    filePath := getLogFilePath()
    fileName := getLogFileName()
    F, err = openLogFile(fileName, filePath)
    if err != nil {
```

```
        log.Fatalln(err)
    }

    logger = log.New(F, DefaultPrefix, log.LstdFlags)
}

...
```

由于原方法形参改变了，因此 openLogFile 也需要调整

实现上传图片接口

这一小节，我们开始实现上次图片相关的一些方法和功能

首先需要在 blog_article 中增加字段 `cover_image_url`，格式为 `varchar(255) DEFAULT '' COMMENT '封面图片地址'`

第零步

一般不会直接将上传的图片名暴露出来，因此我们对图片名进行 MD5 来达到这个效果

在 util 目录下新建 md5.go，写入文件内容：

```
package util

import (
    "crypto/md5"
    "encoding/hex"
)

func EncodeMD5(value string) string {
    m := md5.New()
    m.Write([]byte(value))

    return hex.EncodeToString(m.Sum(nil))
}
```

第一步

在先前我们已经把底层方法给封装好了，实质这一步为封装 image 的处理逻辑

在 pkg 目录下新建 upload/image.go 文件，写入文件内容：

```
package upload

import (
    "os"
    "path"
    "log"
    "fmt"
    "strings"
    "mime/multipart"

    "github.com/EDDYCJY/go-gin-example/pkg/file"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
    "github.com/EDDYCJY/go-gin-example/pkg/logging"
    "github.com/EDDYCJY/go-gin-example/pkg/util"
)

func GetImageFullUrl(name string) string {
    return setting.AppSetting.ImagePrefixUrl + "/" + GetImagePath() + name
}

func GetImageName(name string) string {
    ext := path.Ext(name)
    fileName := strings.TrimSuffix(name, ext)
    fileName = util.EncodeMD5(fileName)

    return fileName + ext
}

func GetImagePath() string {
    return setting.AppSetting.ImageSavePath
}

func GetImageFullPath() string {
    return setting.AppSetting.RuntimeRootPath + GetImagePath()
}

func CheckImageExt(fileName string) bool {
    ext := file.GetExt(fileName)
    for _, allowExt := range setting.AppSetting.ImageAllowExts {
        if strings.ToUpper(allowExt) == strings.ToUpper(ext) {
            return true
        }
    }

    return false
```

```
}

func CheckImageSize(f multipart.File) bool {
    size, err := file.GetSize(f)
    if err != nil {
        log.Println(err)
        logging.Warn(err)
        return false
    }

    return size <= setting.AppSetting.ImageMaxSize
}

func CheckImage(src string) error {
    dir, err := os.Getwd()
    if err != nil {
        return fmt.Errorf("os.Getwd err: %v", err)
    }

    err = file.IsNotExistMkDir(dir + "/" + src)
    if err != nil {
        return fmt.Errorf("file.IsNotExistMkDir err: %v", err)
    }

    perm := file.CheckPermission(src)
    if perm == true {
        return fmt.Errorf("file.CheckPermission Permission denied src: %s", src)
    }

    return nil
}
```

在这里我们实现了 7 个方法，如下：

- GetImageFullUrl：获取图片完整访问 URL
- GetImageName：获取图片名称
- GetImagePath：获取图片路径
- GetImageFullPath：获取图片完整路径
- CheckImageExt：检查图片后缀
- CheckImageSize：检查图片大小
- CheckImage：检查图片

这里基本是对底层代码的二次封装，为了更灵活的处理一些图片特有的逻辑，并且方便修改，不直接对外暴露下层

第二步

这一步将编写上传图片的业务逻辑，在 routers/api 目录下 新建 upload.go 文件，写入文件内容:

```
package api

import (
    "net/http"

    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/pkg/e"
    "github.com/EDDYCJY/go-gin-example/pkg/logging"
    "github.com/EDDYCJY/go-gin-example/pkg/upload"
)

func UploadImage(c *gin.Context) {
    code := e.SUCCESS
    data := make(map[string]string)

    file, image, err := c.Request.FormFile("image")
    if err != nil {
        logging.Warn(err)
        code = e.ERROR
        c.JSON(http.StatusOK, gin.H{
            "code": code,
            "msg":  e.GetMsg(code),
            "data": data,
        })
    }

    if image == nil {
        code = e.INVALID_PARAMS
    } else {
        imageName := upload.GetImageName(image.Filename)
        fullPath := upload.GetImageFullPath()
        savePath := upload.GetImagePath()

        src := fullPath + imageName
        if ! upload.CheckImageExt(imageName) || ! upload.CheckImageSize(file) {
            code = e.ERROR_UPLOAD_CHECK_IMAGE_FORMAT
```

```
        } else {
            err := upload.CheckImage(fullPath)
            if err != nil {
                logging.Warn(err)
                code = e.ERROR_UPLOAD_CHECK_IMAGE_FAIL
            } else if err := c.SaveUploadedFile(image, src); err != nil {
                logging.Warn(err)
                code = e.ERROR_UPLOAD_SAVE_IMAGE_FAIL
            } else {
                data["image_url"] = upload.GetImageFullUrl(imageName)
                data["image_save_url"] = savePath + imageName
            }
        }
    }

    c.JSON(http.StatusOK, gin.H{
        "code": code,
        "msg":  e.GetMsg(code),
        "data": data,
    })
}
```

所涉及的错误码（需在 pkg/e/code.go、msg.go 添加）：

```
// 保存图片失败
ERROR_UPLOAD_SAVE_IMAGE_FAIL = 30001
// 检查图片失败
ERROR_UPLOAD_CHECK_IMAGE_FAIL = 30002
// 校验图片错误，图片格式或大小有问题
ERROR_UPLOAD_CHECK_IMAGE_FORMAT = 30003
```

在这一大段的业务逻辑中，我们做了如下事情：

- c.Request.FormFile：获取上传的图片（返回提供的表单键的第一个文件）
- CheckImageExt、CheckImageSize 检查图片大小，检查图片后缀
- CheckImage：检查上传图片所需（权限、文件夹）
- SaveUploadedFile：保存图片

总的来说，就是 入参 -> 检查 -》 保存 的应用流程

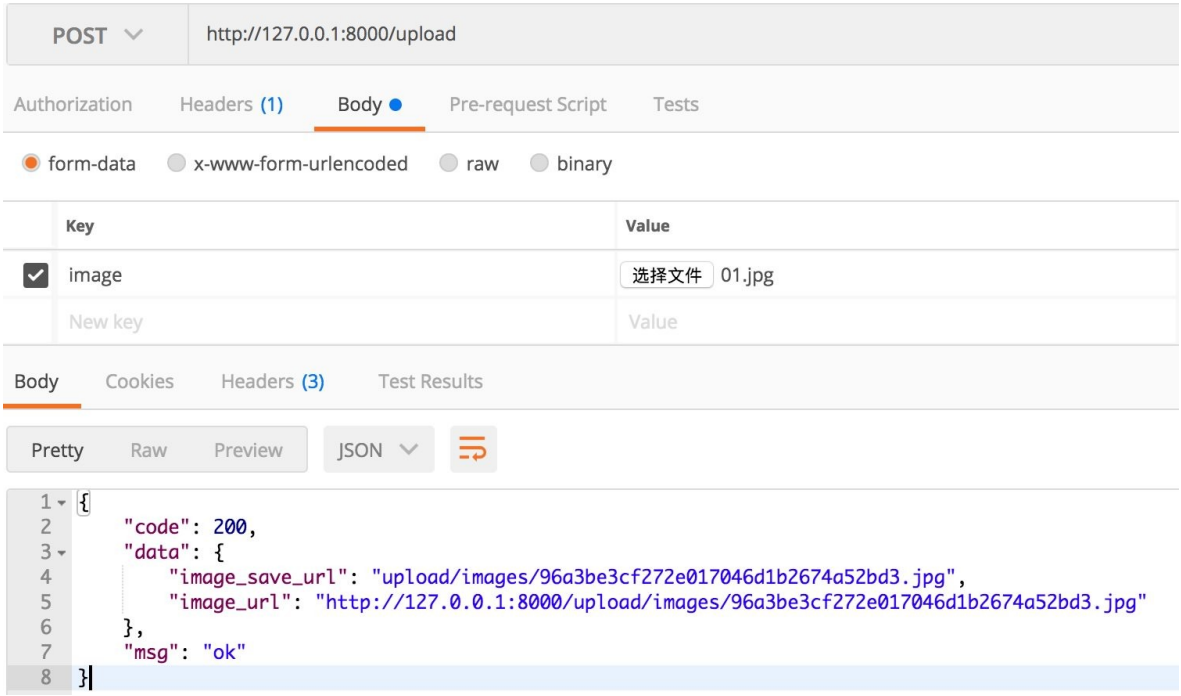
第三步

打开 routers/router.go 文件，增加路由 `r.POST("/upload", api.UploadImage)` ，如：

```
func InitRouter() *gin.Engine {
    r := gin.New()
    ...
    r.GET("/auth", api.GetAuth)
    r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
    r.POST("/upload", api.UploadImage)

    apiv1 := r.Group("/api/v1")
    apiv1.Use(jwt.JWT())
    {
        ...
    }

    return r
}
```

验证

最后我们请求一下上传图片的接口，测试所编写的功能



检查目录下是否含文件（注意权限问题）

```
$ pwd
$GOPATH/src/github.com/EDDYCJY/go-gin-example/runtime/upload/images

$ ll
```

```
... 96a3be3cf272e017046d1b2674a52bd3.jpg
... c39fa784216313cf2faa7c98739fc367.jpeg
```

在这里我们一共返回了 2 个参数，一个是完整的访问 URL，另一个为保存路径

实现 http.FileServer

在完成了上一小节后，我们还需要让前端能够访问到图片，一般是如下：

- CDN
- http.FileSystem

在公司的话，CDN 或自建分布式文件系统居多，也不需要过多关注。而在实践里的话肯定是本地搭建了，Go 本身对此就有很好的支持，而 Gin 更是再封装了一层，只需要在路由增加一行代码即可

r.StaticFS

打开 routers/router.go 文件，增加路由 `r.StaticFS("/upload/images", http.Dir(upload.GetImageFullPath()))` ，如：

```
func InitRouter() *gin.Engine {
    ...
    r.StaticFS("/upload/images", http.Dir(upload.GetImageFullPath()))

    r.GET("/auth", api.GetAuth)
    r.GET("/swagger/*any", ginSwagger.WrapHandler(swaggerFiles.Handler))
    r.POST("/upload", api.UploadImage)
    ...
}
```

它做了什么

当访问 $HOST/upload/images 时，将会读取到 $GOPATH/src/github.com/EDDYCJY/go-gin-example/runtime/upload/images 下的文件

而这行代码又做了什么事呢，我们来看看方法原型

```
// StaticFS works just like `Static()` but a custom `http.FileSystem` can be used instead.
// Gin by default user: gin.Dir()
func (group *RouterGroup) StaticFS(relativePath string, fs http.FileSystem) IRoutes {
```

```
    if strings.Contains(relativePath, ":") || strings.Contains(relativePath, "*"
) {
        panic("URL parameters can not be used when serving a static folder")
    }
    handler := group.createStaticHandler(relativePath, fs)
    urlPattern := path.Join(relativePath, "/*filepath")

    // Register GET and HEAD handlers
    group.GET(urlPattern, handler)
    group.HEAD(urlPattern, handler)
    return group.returnObj()
}
```

首先在暴露的 URL 中禁止了 * 和 : 符号的使用，通过 `createStaticHandler` 创建了静态文件服务，实质最终调用的还是 `fileServer.ServeHTTP` 和一些处理逻辑了

```
func (group *RouterGroup) createStaticHandler(relativePath string, fs http.FileS
ystem) HandlerFunc {
    absolutePath := group.calculateAbsolutePath(relativePath)
    fileServer := http.StripPrefix(absolutePath, http.FileServer(fs))
    _, nolisting := fs.(*onlyfilesFS)
    return func(c *Context) {
        if nolisting {
            c.Writer.WriteHeader(404)
        }
        fileServer.ServeHTTP(c.Writer, c.Request)
    }
}
```

http.StripPrefix

我们可以留意下 `fileServer := http.StripPrefix(absolutePath, http.FileServer(fs))` 这段语句，在静态文件服务中很常见，它有什么作用呢？

`http.StripPrefix` 主要作用是从请求 URL 的路径中删除给定的前缀，最终返回一个 `Handler`

通常 http.FileServer 要与 http.StripPrefix 相结合使用，否则当你运行：

```
http.Handle("/upload/images", http.FileServer(http.Dir("upload/images")))
```

会无法正确的访问到文件目录，因为 `/upload/images` 也包含在了 URL 路径中，必须使用：

```
http.Handle("/upload/images", http.StripPrefix("upload/images", http.FileServer(http.Dir("upload/images"))))
```

/*filepath

到下面可以看到 `urlPattern := path.Join(relativePath, "/*filepath")` ， `/*filepath` 你是谁，你在这里有什么用，你是 Gin 的产物吗?

通过语义可得知是路由的处理逻辑，而 Gin 的路由是基于 httprouter 的，通过查阅文档可得到以下信息

```
Pattern: /src/*filepath

 /src/                     match
 /src/somefile.go          match
 /src/subdir/somefile.go   match
```

`*filepath` 将匹配所有文件路径，并且 `*filepath` 必须在 Pattern 的最后

验证

重新执行 `go run main.go` ，去访问刚刚在 upload 接口得到的图片地址，检查 http.FileSystem 是否正常

修改文章接口

接下来，需要你修改 routers/api/v1/article.go 的 AddArticle、EditArticle 两个接口

- 新增、更新文章接口：支持入参 cover_image_url
- 新增、更新文章接口：增加对 cover_image_url 的非空、最长长度校验

这块前面文章讲过，如果有问题可以参考项目的代码

总结

在这章节中，我们简单的分析了下需求，对应用做出了一个小规划并实施

完成了清单中的功能点和优化，在实际项目中也是常见的场景，希望你能够细细品尝并针对一些点进行深入学习

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 02 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

优化你的应用结构和实现Redis缓存

前言

之前就在想，不少教程或示例的代码设计都是一步到位的（也没问题）

但实际操作的读者真的能够理解透彻为什么吗？左思右想，有了今天这一章的内容，我认为实际经历过一遍印象会更加深刻

本文目标

在本章节，将介绍以下功能的整理：

- 抽离、分层业务逻辑：减轻 routers.go 内的 api 方法的逻辑（但本文暂不分层 repository，这块逻辑还不重）。
- 增加容错性：对 gorm 的错误进行判断。
- Redis 缓存：对获取数据类的接口增加缓存设置。
- 减少重复冗余代码。

问题在哪？

在规划阶段我们发现了一个问题，这是目前的伪代码：

```
if ! HasErrors() {
    if ExistArticleByID(id) {
        DeleteArticle(id)
        code = e.SUCCESS
    } else {
        code = e.ERROR_NOT_EXIST_ARTICLE
    }
} else {
    for _, err := range valid.Errors {
        logging.Info(err.Key, err.Message)
    }
}

c.JSON(http.StatusOK, gin.H{
    "code": code,
    "msg":  e.GetMsg(code),
```

```
    "data": make(map[string]string),
})
```

如果加上规划内的功能逻辑呢，伪代码会变成：

```
if ! HasErrors() {
    exists, err := ExistArticleByID(id)
    if err == nil {
        if exists {
            err = DeleteArticle(id)
            if err == nil {
                code = e.SUCCESS
            } else {
                code = e.ERROR_XXX
            }
        } else {
            code = e.ERROR_NOT_EXIST_ARTICLE
        }
    } else {
        code = e.ERROR_XXX
    }
} else {
    for _, err := range valid.Errors {
        logging.Info(err.Key, err.Message)
    }
}

c.JSON(http.StatusOK, gin.H{
    "code": code,
    "msg":  e.GetMsg(code),
    "data": make(map[string]string),
})
```

如果缓存的逻辑也加进来，后面慢慢不断的迭代，岂不是会变成如下图一样？

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

现在我们发现了问题，应及时解决这个代码结构问题，同时把代码写的清晰、漂亮、易读易改也是一个重要指标

如何改？

在左耳朵耗子的文章中，这类代码被称为“箭头型”代码，有如下几个问题：

1、我的显示器不够宽，箭头型代码缩进太狠了，需要我来回拉水平滚动条，这让我在读代码的时候，相当的不舒服

2、除了宽度外还有长度，有的代码的 if-else 里的 if-else 里的 if-else 的代码太多，读到中间你都不知道中间的代码是经过了什么样的层层检查才来到这里的

总而言之，“箭头型代码”如果嵌套太多，代码太长的话，会相当容易让维护代码的人（包括自己）迷失在代码中，因为看到最内层的代码时，你已经不知道前面的那一层一层的条件判断是什么样的，代码是怎么运行到这里的，所以，箭头型代码是非常难以维护和 Debug 的。

简单的来说，就是**让出错的代码先返回，前面把所有的错误判断全判断掉，然后就剩下的就是正常的代码了**

（注意：本段引用自耗子哥的 如何重构“箭头型”代码，建议细细品尝）

落实

本项目将对既有代码进行优化和实现缓存，希望你习得方法并对其他地方也进行优化

第一步：完成 Redis 的基础设施建设（需要你先装好 Redis）

第二步：对现有代码进行拆解、分层（不会贴上具体步骤的代码，希望你能够实操一波，加深理解 ）

Redis

一、配置

打开 conf/app.ini 文件，新增配置：

```
...
[redis]
Host = 127.0.0.1:6379
Password =
MaxIdle = 30
MaxActive = 30
IdleTimeout = 200
```

二、缓存 Prefix

打开 pkg/e 目录，新建 cache.go，写入内容：

```
package e

const (
    CACHE_ARTICLE = "ARTICLE"
    CACHE_TAG     = "TAG"
)
```

三、缓存 Key

（1）、打开 service 目录，新建 cache_service/article.go

写入内容：传送门

（2）、打开 service 目录，新建 cache_service/tag.go

写入内容：传送门

这一部分主要是编写获取缓存 KEY 的方法，直接参考传送门即可

四、Redis 工具包

打开 pkg 目录，新建 gredis/redis.go，写入内容：

```
package gredis

import (
    "encoding/json"
    "time"

    "github.com/gomodule/redigo/redis"

    "github.com/EDDYCJY/go-gin-example/pkg/setting"
)

var RedisConn *redis.Pool

func Setup() error {
    RedisConn = &redis.Pool{
        MaxIdle:     setting.RedisSetting.MaxIdle,
        MaxActive:   setting.RedisSetting.MaxActive,
        IdleTimeout: setting.RedisSetting.IdleTimeout,
        Dial: func() (redis.Conn, error) {
            c, err := redis.Dial("tcp", setting.RedisSetting.Host)
            if err != nil {
                return nil, err
            }
            if setting.RedisSetting.Password != "" {
                if _, err := c.Do("AUTH", setting.RedisSetting.Password); err !=
nil {
                    c.Close()
                    return nil, err
                }
            }
            return c, err
        },
        TestOnBorrow: func(c redis.Conn, t time.Time) error {
            _, err := c.Do("PING")
            return err
        },
    }

    return nil
}

func Set(key string, data interface{}, time int) error {
    conn := RedisConn.Get()
    defer conn.Close()
```

```
    value, err := json.Marshal(data)
    if err != nil {
        return err
    }

    _, err = conn.Do("SET", key, value)
    if err != nil {
        return err
    }

    _, err = conn.Do("EXPIRE", key, time)
    if err != nil {
        return err
    }

    return nil
}

func Exists(key string) bool {
    conn := RedisConn.Get()
    defer conn.Close()

    exists, err := redis.Bool(conn.Do("EXISTS", key))
    if err != nil {
        return false
    }

    return exists
}

func Get(key string) ([]byte, error) {
    conn := RedisConn.Get()
    defer conn.Close()

    reply, err := redis.Bytes(conn.Do("GET", key))
    if err != nil {
        return nil, err
    }

    return reply, nil
}

func Delete(key string) (bool, error) {
    conn := RedisConn.Get()
    defer conn.Close()
```

```
    return redis.Bool(conn.Do("DEL", key))
}

func LikeDeletes(key string) error {
    conn := RedisConn.Get()
    defer conn.Close()

    keys, err := redis.Strings(conn.Do("KEYS", "*"+key+"*"))
    if err != nil {
        return err
    }

    for _, key := range keys {
        _, err = Delete(key)
        if err != nil {
            return err
        }
    }

    return nil
}
```

在这里我们做了一些基础功能封装

1、设置 RedisConn 为 redis.Pool（连接池）并配置了它的一些参数：

- Dial：提供创建和配置应用程序连接的一个函数
- TestOnBorrow：可选的应用程序检查健康功能
- MaxIdle：最大空闲连接数
- MaxActive：在给定时间内，允许分配的最大连接数（当为零时，没有限制）
- IdleTimeout：在给定时间内将会保持空闲状态，若到达时间限制则关闭连接（当为零时，没有限制）

2、封装基础方法

文件内包含 Set、Exists、Get、Delete、LikeDeletes 用于支撑目前的业务逻辑，而在里面涉及到了如方法：

（1） `RedisConn.Get()` ：在连接池中获取一个活跃连接

（2） `conn.Do(commandName string, args ...interface{})` ：向 Redis 服务器发送命令并返回收到的答复

（3） `redis.Bool(reply interface{}, err error)` ：将命令返回转为布尔值

（4） `redis.Bytes(reply interface{}, err error)` ：将命令返回转为 Bytes

（5） `redis.Strings(reply interface{}, err error)` ：将命令返回转为 []string

在 redigo 中包含大量类似的方法，万变不离其宗，建议熟悉其使用规则和 Redis 命令 即可

到这里为止，Redis 就可以愉快的调用啦。另外受篇幅限制，这块的深入讲解会另外开设！

拆解、分层

在先前规划中，引出几个方法去优化我们的应用结构

- 错误提前返回
- 统一返回方法
- 抽离 Service，减轻 routers/api 的逻辑，进行分层
- 增加 gorm 错误判断，让错误提示更明确（增加内部错误码）

编写返回方法

要让错误提前返回，c.JSON 的侵入是不可避免的，但是可以让其更具可变性，指不定哪天就变 XML 了呢？

1、打开 pkg 目录，新建 app/request.go，写入文件内容：

```
package app

import (
    "github.com/astaxie/beego/validation"

    "github.com/EDDYCJY/go-gin-example/pkg/logging"
)

func MarkErrors(errors []*validation.Error) {
    for _, err := range errors {
        logging.Info(err.Key, err.Message)
    }

    return
}
```

2、打开 pkg 目录，新建 app/response.go，写入文件内容：

```
package app

import (
    "github.com/gin-gonic/gin"

    "github.com/EDDYCJY/go-gin-example/pkg/e"
)

type Gin struct {
    C *gin.Context
}

func (g *Gin) Response(httpCode, errCode int, data interface{}) {
    g.C.JSON(httpCode, gin.H{
        "code": errCode,
        "msg":  e.GetMsg(errCode),
        "data": data,
    })

    return
}
```

这样子以后如果要变动，直接改动 app 包内的方法即可

修改既有逻辑

打开 routers/api/v1/article.go，查看修改 GetArticle 方法后的代码为：

```
func GetArticle(c *gin.Context) {
    appG := app.Gin{c}
    id := com.StrTo(c.Param("id")).MustInt()
    valid := validation.Validation{}
    valid.Min(id, 1, "id").Message("ID必须大于0")

    if valid.HasErrors() {
        app.MarkErrors(valid.Errors)
        appG.Response(http.StatusOK, e.INVALID_PARAMS, nil)
        return
    }

    articleService := article_service.Article{ID: id}
    exists, err := articleService.ExistByID()
    if err != nil {
```

```
        appG.Response(http.StatusOK, e.ERROR_CHECK_EXIST_ARTICLE_FAIL, nil)
        return
    }
    if !exists {
        appG.Response(http.StatusOK, e.ERROR_NOT_EXIST_ARTICLE, nil)
        return
    }

    article, err := articleService.Get()
    if err != nil {
        appG.Response(http.StatusOK, e.ERROR_GET_ARTICLE_FAIL, nil)
        return
    }

    appG.Response(http.StatusOK, e.SUCCESS, article)
}
```

这里有几个值得变动点，主要是在内部增加了错误返回，如果存在错误则直接返回。另外进行了分层，业务逻辑内聚到了 service 层中去，而 routers/api（controller）显著减轻，代码会更加的直观

例如 service/article_service 下的 `articleService.Get()` 方法：

```
func (a *Article) Get() (*models.Article, error) {
    var cacheArticle *models.Article

    cache := cache_service.Article{ID: a.ID}
    key := cache.GetArticleKey()
    if gredis.Exists(key) {
        data, err := gredis.Get(key)
        if err != nil {
            logging.Info(err)
        } else {
            json.Unmarshal(data, &cacheArticle)
            return cacheArticle, nil
        }
    }

    article, err := models.GetArticle(a.ID)
    if err != nil {
        return nil, err
    }

    gredis.Set(key, article, 3600)
```

```
    return article, nil
}
```

而对于 gorm 的 错误返回设置，只需要修改 models/article.go 如下:

```
func GetArticle(id int) (*Article, error) {
    var article Article
    err := db.Where("id = ? AND deleted_on = ? ", id, 0).First(&article).Related
(&article.Tag).Error
    if err != nil && err != gorm.ErrRecordNotFound {
        return nil, err
    }

    return &article, nil
}
```

习惯性增加 .Error，把控绝大部分的错误。另外需要注意一点，在 gorm 中，查找不到记录也算一种“错误”哦

最后

显然，本章节并不是你跟着我敲系列。我给你的课题是“实现 Redis 缓存并优化既有的业务逻辑代码”

让其能够不断地适应业务的发展，让代码更清晰易读，且呈层级和结构性

如果有疑惑，可以到 go-gin-example 看看我是怎么写的，你是怎么写的，又分别有什么优势、劣势，取长补短一波？

参考

本系列示例代码

- go-gin-example

推荐阅读

- 如何重构“箭头型”代码

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

实现导出、导入 Excel

知识点

- 导出功能的实现

本文目标

在本节，我们将实现对标签信息的导出、导入功能，这是很标配功能了，希望你掌握基础的使用方式。

另外在本文我们使用了 2 个 Excel 的包，excelize 最初的 XML 格式文件的一些结构，是通过 tealeg/xlsx 格式文件结构演化而来的，因此特意在此都展示了，你可以根据自己的场景和喜爱去使用。

配置

首先要指定导出的 Excel 文件的存储路径，在 app.ini 中增加配置：

```
[app]
...

ExportSavePath = export/
```

修改 setting.go 的 App struct：

```
type App struct {
    JwtSecret       string
    PageSize        int
    PrefixUrl       string

    RuntimeRootPath string

    ImageSavePath  string
    ImageMaxSize   int
    ImageAllowExts []string

    ExportSavePath string

    LogSavePath string
```

```
    LogSaveName string
    LogFileExt  string
    TimeFormat  string
}
```

在这里需增加 ExportSavePath 配置项，另外将先前 ImagePrefixUrl 改为 PrefixUrl 用于支撑两者的 HOST 获取

（注意修改 image.go 的 GetImageFullUrl 方法）

pkg

新建 pkg/export/excel.go 文件，如下：

```
package export

import "github.com/EDDYCJY/go-gin-example/pkg/setting"

func GetExcelFullUrl(name string) string {
    return setting.AppSetting.PrefixUrl + "/" + GetExcelPath() + name
}

func GetExcelPath() string {
    return setting.AppSetting.ExportSavePath
}

func GetExcelFullPath() string {
    return setting.AppSetting.RuntimeRootPath + GetExcelPath()
}
```

这里编写了一些常用的方法，以后取值方式如果有变动，直接改内部代码即可，对外不可见

尝试一下标准库

```
f, err := os.Create(export.GetExcelFullPath() + "test.csv")
if err != nil {
    panic(err)
}
defer f.Close()

f.WriteString("\xEF\xBB\xBF")

w := csv.NewWriter(f)
```

```
data := [][]string{
    {"1", "test1", "test1-1"},
    {"2", "test2", "test2-1"},
    {"3", "test3", "test3-1"},
}

w.WriteAll(data)
```

在 Go 提供的标准库 encoding/csv 中，天然的支持 csv 文件的读取和处理，在本段代码中，做了如下工作：

1、os.Create：

创建了一个 test.csv 文件

2、f.WriteString("\xEF\xBB\xBF")：

`\xEF\xBB\xBF` 是 UTF-8 BOM 的 16 进制格式，在这里的用处是标识文件的编码格式，通常会出现在文件的开头，因此第一步就要将其写入。如果不标识 UTF-8 的编码格式的话，写入的汉字会显示为乱码

3、csv.NewWriter：

```
func NewWriter(w io.Writer) *Writer {
    return &Writer{
        Comma: ',',
        w:     bufio.NewWriter(w),
    }
}
```

4、w.WriteAll：

```
func (w *Writer) WriteAll(records [][]string) error {
    for _, record := range records {
        err := w.Write(record)
        if err != nil {
            return err
        }
    }
    return w.w.Flush()
}
```

WriteAll 实际是对 Write 的封装，需要注意在最后调用了 `w.w.Flush()` ，这充分了说明了 WriteAll 的使用场景，你可以想想作者的设计用意

导出

Service 方法

打开 service/tag.go，增加 Export 方法，如下：

```
func (t *Tag) Export() (string, error) {
    tags, err := t.GetAll()
    if err != nil {
        return "", err
    }

    file := xlsx.NewFile()
    sheet, err := file.AddSheet("标签信息")
    if err != nil {
        return "", err
    }

    titles := []string{"ID", "名称", "创建人", "创建时间", "修改人", "修改时间"}
    row := sheet.AddRow()

    var cell *xlsx.Cell
    for _, title := range titles {
        cell = row.AddCell()
        cell.Value = title
    }

    for _, v := range tags {
        values := []string{
            strconv.Itoa(v.ID),
            v.Name,
            v.CreatedBy,
            strconv.Itoa(v.CreatedOn),
            v.ModifiedBy,
            strconv.Itoa(v.ModifiedOn),
        }

        row = sheet.AddRow()
        for _, value := range values {
            cell = row.AddCell()
            cell.Value = value
        }
    }

    time := strconv.Itoa(int(time.Now().Unix()))
```

```
    filename := "tags-" + time + ".xlsx"

    fullPath := export.GetExcelFullPath() + filename
    err = file.Save(fullPath)
    if err != nil {
        return "", err
    }

    return filename, nil
}
```

routers 入口

打开 routers/api/v1/tag.go，增加如下方法：

```
func ExportTag(c *gin.Context) {
    appG := app.Gin{C: c}
    name := c.PostForm("name")
    state := -1
    if arg := c.PostForm("state"); arg != "" {
        state = com.StrTo(arg).MustInt()
    }

    tagService := tag_service.Tag{
        Name:  name,
        State: state,
    }

    filename, err := tagService.Export()
    if err != nil {
        appG.Response(http.StatusOK, e.ERROR_EXPORT_TAG_FAIL, nil)
        return
    }

    appG.Response(http.StatusOK, e.SUCCESS, map[string]string{
        "export_url":      export.GetExcelFullUrl(filename),
        "export_save_url": export.GetExcelPath() + filename,
    })
}
```

路由

在 routers/router.go 文件中增加路由方法，如下

```
apiv1 := r.Group("/api/v1")
apiv1.Use(jwt.JWT())
{
    ...
    //导出标签
    r.POST("/tags/export", v1.ExportTag)
}
```

验证接口

访问 `http://127.0.0.1:8000/tags/export` ，结果如下：

```
{
  "code": 200,
  "data": {
    "export_save_url": "export/tags-1528903393.xlsx",
    "export_url": "http://127.0.0.1:8000/export/tags-1528903393.xlsx"
  },
  "msg": "ok"
}
```

最终通过接口返回了导出文件的地址和保存地址

StaticFS

那你想想，现在直接访问地址肯定是无法下载文件的，那么该如何做呢？

打开 router.go 文件，增加代码如下：

```
r.StaticFS("/export", http.Dir(export.GetExcelFullPath()))
```

若你不理解，强烈建议温习下前面的章节，举一反三

验证下载

再次访问上面的 export_url ，如： `http://127.0.0.1:8000/export/tags-1528903393.xlsx` ，是不是成功了呢？

导入

Service 方法

打开 service/tag.go，增加 Import 方法，如下：

```
func (t *Tag) Import(r io.Reader) error {
    xlsx, err := excelize.OpenReader(r)
    if err != nil {
        return err
    }

    rows := xlsx.GetRows("标签信息")
    for irow, row := range rows {
        if irow > 0 {
            var data []string
            for _, cell := range row {
                data = append(data, cell)
            }

            models.AddTag(data[1], 1, data[2])
        }
    }

    return nil
}
```

routers 入口

打开 routers/api/v1/tag.go，增加如下方法：

```
func ImportTag(c *gin.Context) {
    appG := app.Gin{C: c}

    file, _, err := c.Request.FormFile("file")
    if err != nil {
        logging.Warn(err)
        appG.Response(http.StatusOK, e.ERROR, nil)
        return
    }

    tagService := tag_service.Tag{}
    err = tagService.Import(file)
    if err != nil {
        logging.Warn(err)
        appG.Response(http.StatusOK, e.ERROR_IMPORT_TAG_FAIL, nil)
        return
    }
```

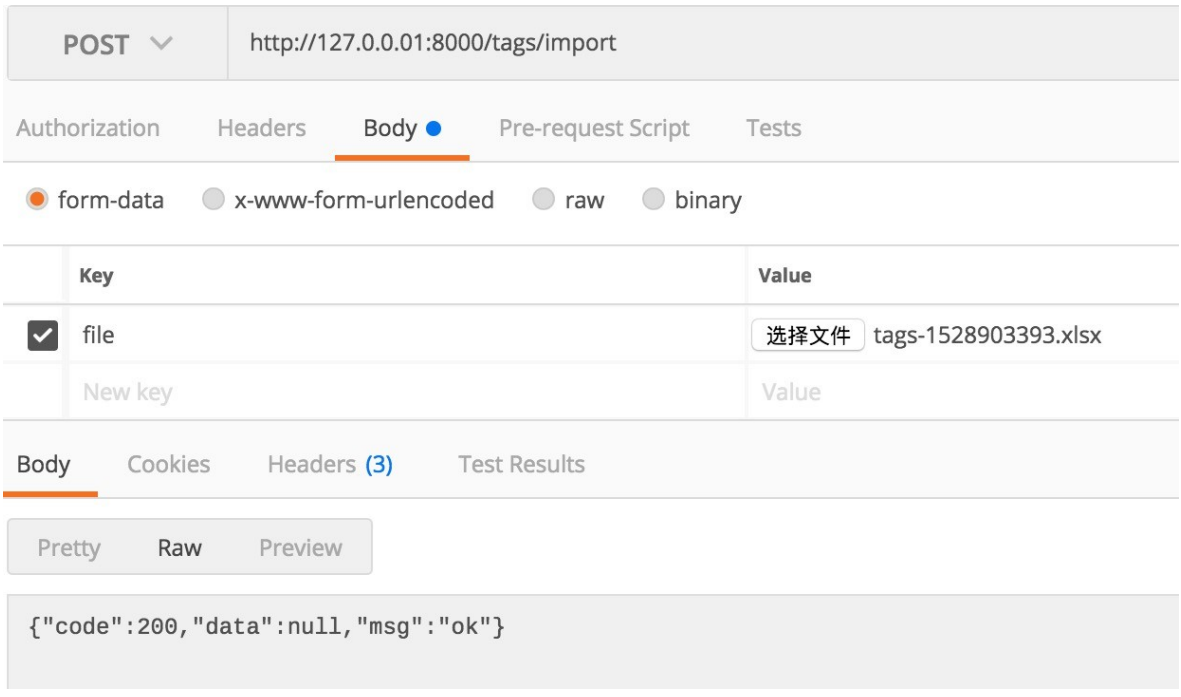
```
    appG.Response(http.StatusOK, e.SUCCESS, nil)
}
```

路由

在 routers/router.go 文件中增加路由方法，如下

```
apiv1 := r.Group("/api/v1")
apiv1.Use(jwt.JWT())
{
    ...
    //导入标签
    r.POST("/tags/import", v1.ImportTag)
}
```

验证



在这里我们将先前导出的 Excel 文件作为入参，访问 `http://127.0.0.01:8000/tags/import` ，检查返回和数据是否正确入库

总结

在本文中，简单介绍了 Excel 的导入、导出的使用方式，使用了以下 2 个包：

- tealeg/xlsx
- 360EntSecGroup-Skylar/excelize

你可以细细阅读一下它的实现和使用方式，对你的把控更有帮助

课外

- tag：导出使用 excelize 的方式去实现（可能你会发现更简单哦）
- tag：导入去重功能实现
- artice ：导入、导出功能实现

也不失为你很好的练手机会，如果有兴趣，可以试试

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 02 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

生成二维码、合并海报

知识点

- 图片生成
- 二维码生成

本文目标

在文章的详情页中，我们常常会需要去宣传它，而目前最常见的就是发海报了，今天我们将实现如下功能：

- 生成二维码
- 合并海报（背景图 + 二维码）

实现

首先，你需要在 App 配置项中增加二维码及其海报的存储路径，我们约定配置项名称为 `QrCodeSavePath`，值为 `qrcode/`，经过多节连载的你应该能够完成，若有不懂可参照 go-gin-example。

生成二维码

安装

```
$ go get -u github.com/boombuler/barcode
```

工具包

考虑生成二维码这一动作贴合工具包的定义，且有公用的可能性，新建 pkg/qrcode/qrcode.go 文件，写入内容：

```
package qrcode

import (
    "image/jpeg"
```

```
    "github.com/boombuler/barcode"
    "github.com/boombuler/barcode/qr"

    "github.com/EDDYCJY/go-gin-example/pkg/file"
    "github.com/EDDYCJY/go-gin-example/pkg/setting"
    "github.com/EDDYCJY/go-gin-example/pkg/util"
)

type QrCode struct {
    URL    string
    Width  int
    Height int
    Ext    string
    Level  qr.ErrorCorrectionLevel
    Mode   qr.Encoding
}

const (
    EXT_JPG = ".jpg"
)

func NewQrCode(url string, width, height int, level qr.ErrorCorrectionLevel, mod
e qr.Encoding) *QrCode {
    return &QrCode{
        URL:    url,
        Width:  width,
        Height: height,
        Level:  level,
        Mode:   mode,
        Ext:    EXT_JPG,
    }
}

func GetQrCodePath() string {
    return setting.AppSetting.QrCodeSavePath
}

func GetQrCodeFullPath() string {
    return setting.AppSetting.RuntimeRootPath + setting.AppSetting.QrCodeSavePat
h
}

func GetQrCodeFullUrl(name string) string {
    return setting.AppSetting.PrefixUrl + "/" + GetQrCodePath() + name
}
```

```
func GetQrCodeFileName(value string) string {
    return util.EncodeMD5(value)
}

func (q *QrCode) GetQrCodeExt() string {
    return q.Ext
}

func (q *QrCode) CheckEncode(path string) bool {
    src := path + GetQrCodeFileName(q.URL) + q.GetQrCodeExt()
    if file.CheckNotExist(src) == true {
        return false
    }

    return true
}

func (q *QrCode) Encode(path string) (string, string, error) {
    name := GetQrCodeFileName(q.URL) + q.GetQrCodeExt()
    src := path + name
    if file.CheckNotExist(src) == true {
        code, err := qr.Encode(q.URL, q.Level, q.Mode)
        if err != nil {
            return "", "", err
        }

        code, err = barcode.Scale(code, q.Width, q.Height)
        if err != nil {
            return "", "", err
        }

        f, err := file.MustOpen(name, path)
        if err != nil {
            return "", "", err
        }
        defer f.Close()

        err = jpeg.Encode(f, code, nil)
        if err != nil {
            return "", "", err
        }
    }

    return name, path, nil
}
```

这里主要聚焦 `func (q *QrCode) Encode` 方法，做了如下事情：

- 获取二维码生成路径
- 创建二维码
- 缩放二维码到指定大小
- 新建存放二维码图片的文件
- 将图像（二维码）以 JPEG 4：2：0 基线格式写入文件

另外在 `jpeg.Encode(f, code, nil)` 中，第三个参数可设置其图像质量，默认值为 75

```
// DefaultQuality is the default quality encoding parameter.
const DefaultQuality = 75

// Options are the encoding parameters.
// Quality ranges from 1 to 100 inclusive, higher is better.
type Options struct {
    Quality int
}
```

路由方法

1、第一步

在 routers/api/v1/article.go 新增 GenerateArticlePoster 方法用于接口开发

2、第二步

在 routers/router.go 的 apiv1 中新增 `apiv1.POST("/articles/poster/generate", v1.GenerateArticlePoster)` 路由

3、第三步

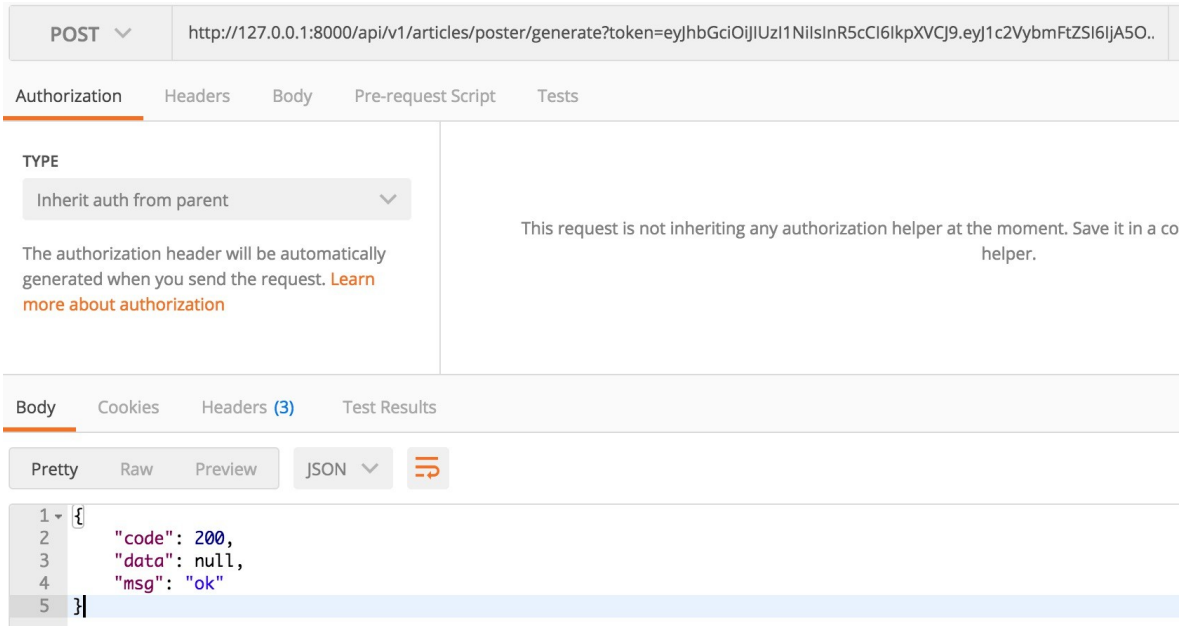
修改 GenerateArticlePoster 方法，编写对应的生成逻辑，如下：

```
const (
    QRCODE_URL = "https://github.com/EDDYCJY/blog#gin%E7%B3%BB%E5%88%97%E7%9B%AE%E5%BD%95"
)

func GenerateArticlePoster(c *gin.Context) {
    appG := app.Gin{c}
    qrc := qrcode.NewQrCode(QRCODE_URL, 300, 300, qr.M, qr.Auto)
    path := qrcode.GetQrCodeFullPath()
```

```
    _, _, err := qrc.Encode(path)
    if err != nil {
        appG.Response(http.StatusOK, e.ERROR, nil)
        return
    }

    appG.Response(http.StatusOK, e.SUCCESS, nil)
}
```

验证

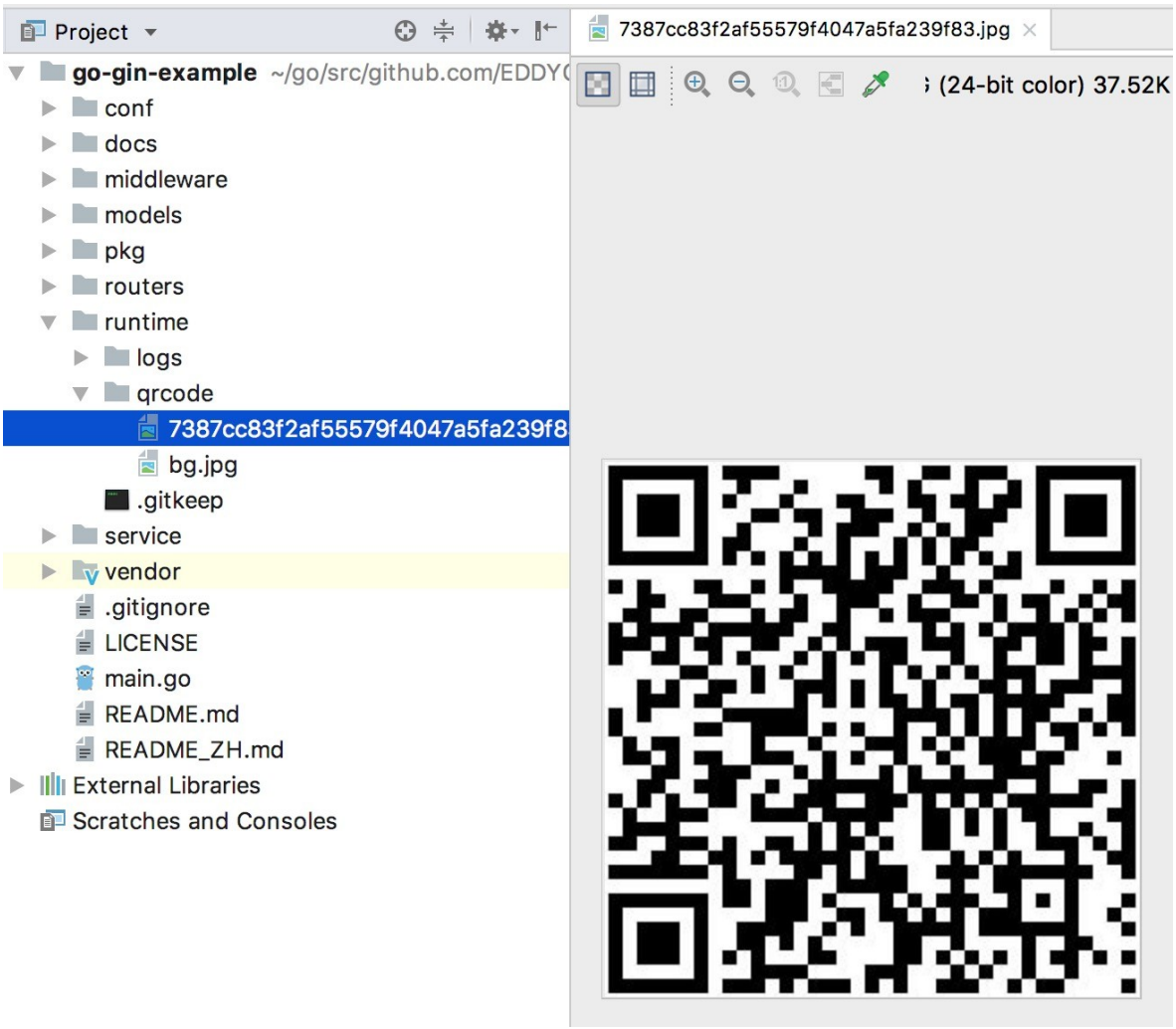
通过 POST 方法访问 `http://127.0.0.1:8000/api/v1/articles/poster/generate?token=$token` （注意 $token）



通过检查两个点确定功能是否正常，如下：

1、访问结果是否 200

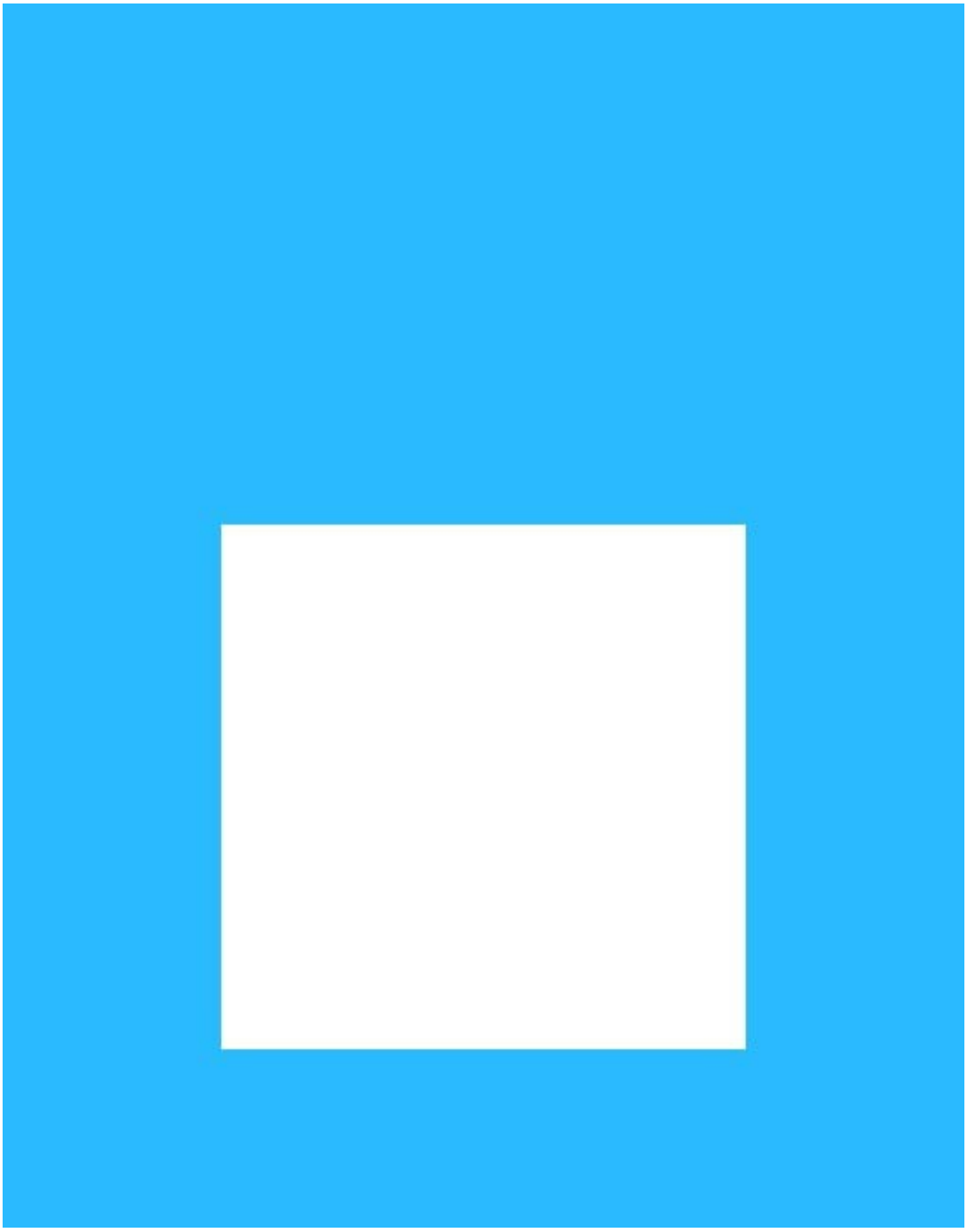
2、本地目录是否成功生成二维码图片



合并海报

在这一节，将实现二维码图片与背景图合并成新的一张图，可用于常见的宣传海报等业务场景

背景图



将背景图另存为 runtime/qrcode/bg.jpg（实际应用，可存在 OSS 或其他地方）

service 方法

打开 service/article_service 目录，新建 article_poster.go 文件，写入内容：

```
package article_service

import (
    "image"
```

```
    "image/draw"
    "image/jpeg"
    "os"

    "github.com/EDDYCJY/go-gin-example/pkg/file"
    "github.com/EDDYCJY/go-gin-example/pkg/qrcode"
)

type ArticlePoster struct {
    PosterName string
    *Article
    Qr *qrcode.QrCode
}

func NewArticlePoster(posterName string, article *Article, qr *qrcode.QrCode) *A
rticlePoster {
    return &ArticlePoster{
        PosterName: posterName,
        Article:    article,
        Qr:         qr,
    }
}

func GetPosterFlag() string {
    return "poster"
}

func (a *ArticlePoster) CheckMergedImage(path string) bool {
    if file.CheckNotExist(path+a.PosterName) == true {
        return false
    }

    return true
}

func (a *ArticlePoster) OpenMergedImage(path string) (*os.File, error) {
    f, err := file.MustOpen(a.PosterName, path)
    if err != nil {
        return nil, err
    }

    return f, nil
}

type ArticlePosterBg struct {
    Name string
```

```
    *ArticlePoster
    *Rect
    *Pt
}

type Rect struct {
    Name string
    X0   int
    Y0   int
    X1   int
    Y1   int
}

type Pt struct {
    X int
    Y int
}

func NewArticlePosterBg(name string, ap *ArticlePoster, rect *Rect, pt *Pt) *Art
iclePosterBg {
    return &ArticlePosterBg{
        Name:          name,
        ArticlePoster: ap,
        Rect:          rect,
        Pt:            pt,
    }
}

func (a *ArticlePosterBg) Generate() (string, string, error) {
    fullPath := qrcode.GetQrCodeFullPath()
    fileName, path, err := a.Qr.Encode(fullPath)
    if err != nil {
        return "", "", err
    }

    if !a.CheckMergedImage(path) {
        mergedF, err := a.OpenMergedImage(path)
        if err != nil {
            return "", "", err
        }
        defer mergedF.Close()

        bgF, err := file.MustOpen(a.Name, path)
        if err != nil {
            return "", "", err
        }
```

```
        defer bgF.Close()

        qrF, err := file.MustOpen(fileName, path)
        if err != nil {
            return "", "", err
        }
        defer qrF.Close()

        bgImage, err := jpeg.Decode(bgF)
        if err != nil {
            return "", "", err
        }
        qrImage, err := jpeg.Decode(qrF)
        if err != nil {
            return "", "", err
        }

        jpg := image.NewRGBA(image.Rect(a.Rect.X0, a.Rect.Y0, a.Rect.X1, a.Rect.
Y1))

        draw.Draw(jpg, jpg.Bounds(), bgImage, bgImage.Bounds().Min, draw.Over)
        draw.Draw(jpg, jpg.Bounds(), qrImage, qrImage.Bounds().Min.Sub(image.Pt
(a.Pt.X, a.Pt.Y)), draw.Over)

        jpeg.Encode(mergedF, jpg, nil)
    }

    return fileName, path, nil
}
```

这里重点留意 `func (a *ArticlePosterBg) Generate()` 方法，做了如下事情：

- 获取二维码存储路径
- 生成二维码图像
- 检查合并后图像（指的是存放合并后的海报）是否存在
- 若不存在，则生成待合并的图像 mergedF
- 打开事先存放的背景图 bgF
- 打开生成的二维码图像 qrF
- 解码 bgF 和 qrF 返回 image.Image
- 创建一个新的 RGBA 图像
- 在 RGBA 图像上绘制 背景图（bgF）
- 在已绘制背景图的 RGBA 图像上，在指定 Point 上绘制二维码图像（qrF）

- 将绘制好的 RGBA 图像以 JPEG 4：2：0 基线格式写入合并后的图像文件（mergedF）

错误码

新增 错误码，错误提示

路由方法

打开 routers/api/v1/article.go 文件，修改 GenerateArticlePoster 方法，编写最终的业务逻辑（含生成二维码及合并海报），如下：

```
const (
    QRCODE_URL = "https://github.com/EDDYCJY/blog#gin%E7%B3%BB%E5%88%97%E7%9B%AE%E5%BD%95"
)

func GenerateArticlePoster(c *gin.Context) {
    appG := app.Gin{c}
    article := &article_service.Article{}
    qr := qrcode.NewQrCode(QRCODE_URL, 300, 300, qr.M, qr.Auto) // 目前写死 gin 系列路径，可自行增加业务逻辑
    posterName := article_service.GetPosterFlag() + "-" + qrcode.GetQrCodeFileName(qr.URL) + qr.GetQrCodeExt()
    articlePoster := article_service.NewArticlePoster(posterName, article, qr)
    articlePosterBgService := article_service.NewArticlePosterBg(
        "bg.jpg",
        articlePoster,
        &article_service.Rect{
            X0: 0,
            Y0: 0,
            X1: 550,
            Y1: 700,
        },
        &article_service.Pt{
            X: 125,
            Y: 298,
        },
    )

    _, filePath, err := articlePosterBgService.Generate()
    if err != nil {
        appG.Response(http.StatusOK, e.ERROR_GEN_ARTICLE_POSTER_FAIL, nil)
        return
    }
```

```
    appG.Response(http.StatusOK, e.SUCCESS, map[string]string{
        "poster_url":      qrcode.GetQrCodeFullUrl(posterName),
        "poster_save_url": filePath + posterName,
    })
}
```

这块涉及到大量知识，强烈建议阅读下，如下：

- image.Rect
- image.Pt
- image.NewRGBA
- jpeg.Encode
- jpeg.Decode
- draw.Op
- draw.Draw
- go-imagedraw-package

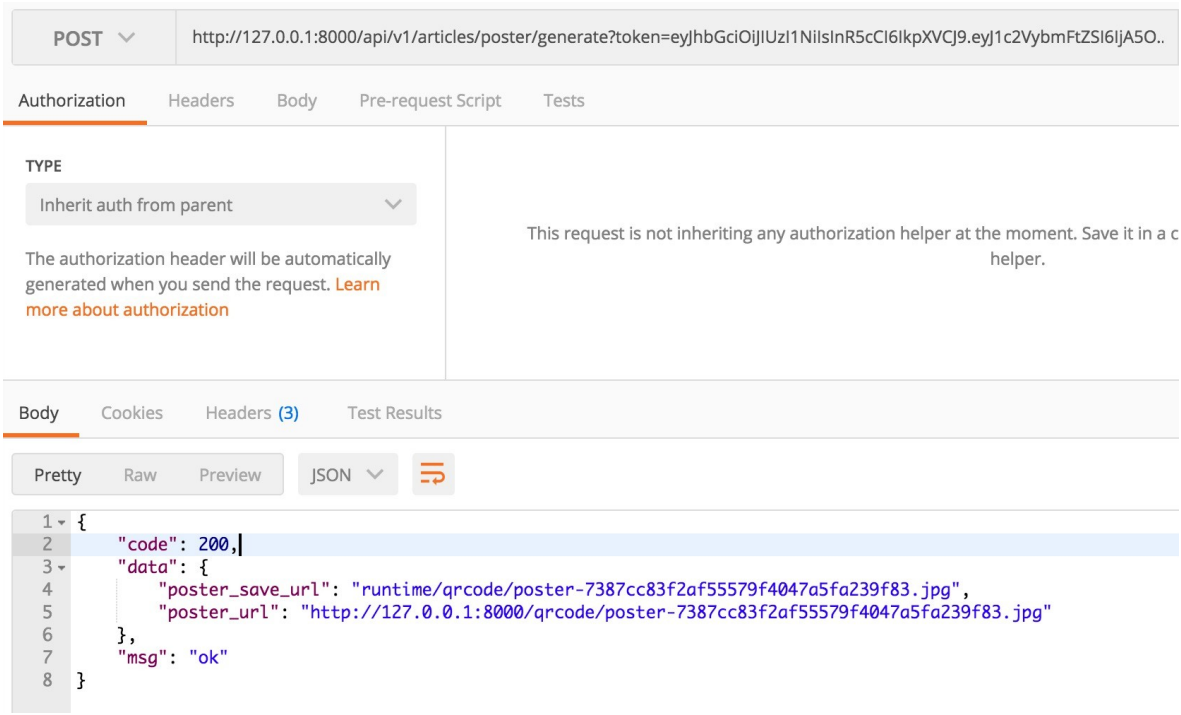
其所涉及、关联的库都建议研究一下

StaticFS

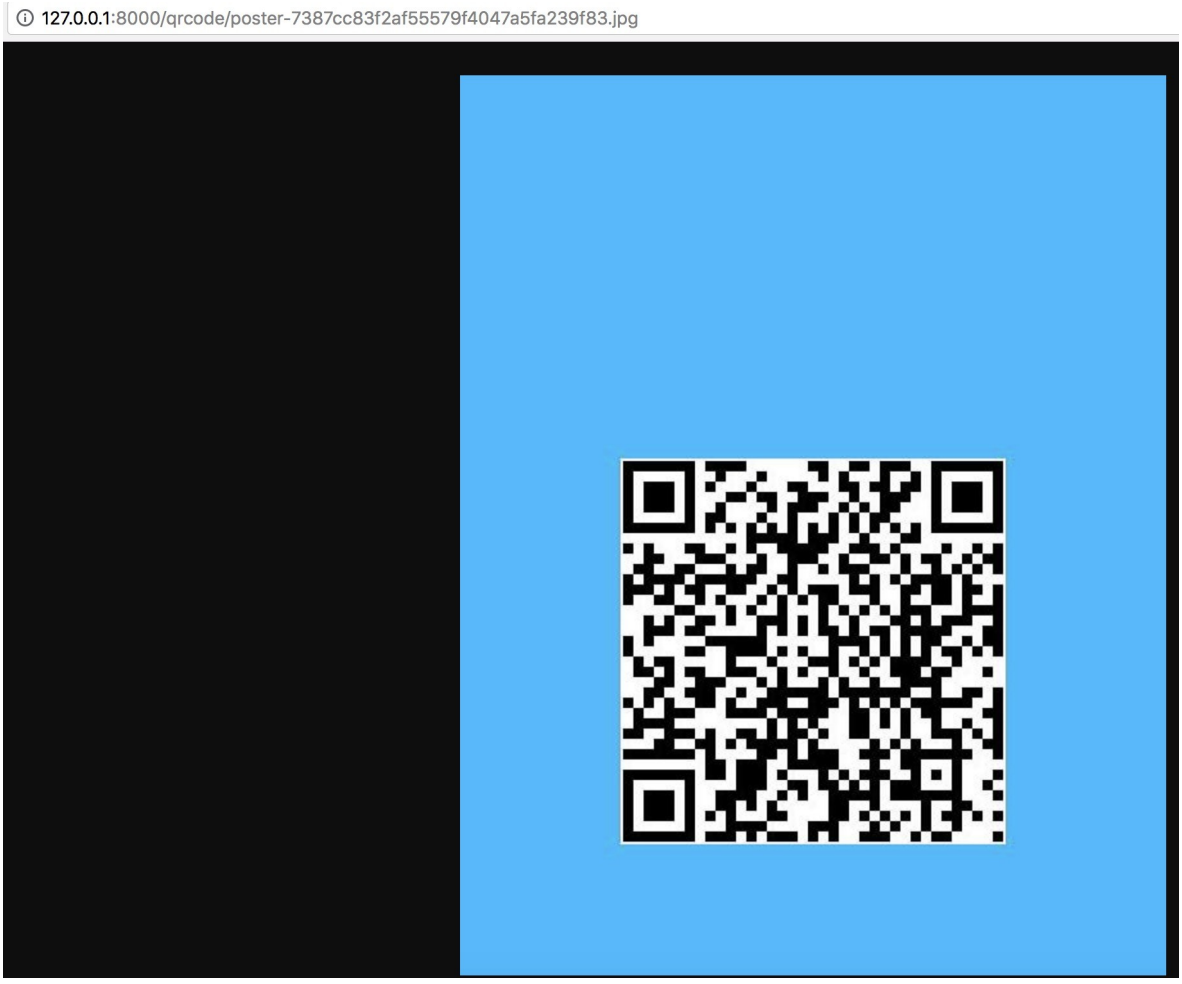
在 routers/router.go 文件，增加如下代码:

```
r.StaticFS("/qrcode", http.Dir(qrcode.GetQrCodeFullPath()))
```

验证



访问完整的 URL 路径，返回合成后的海报并扫除二维码成功则正确



总结

在本章节实现了两个很常见的业务功能，分别是生成二维码和合并海报。希望你能够仔细阅读我给出的链接，这块的知识量不少，想要用好图像处理的功能，必须理解对应的思路，举一反三

最后希望对你有所帮助

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 02 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

在图片上绘制文字

知识点

- 字体库使用
- 图片合成

本文目标

主要实现**合并后的海报上绘制文字**的功能（这个需求也是常见的很了），内容比较简单。

实现

这里使用的是 微软雅黑 的字体，请点击进行下载并**存放到 runtime/fonts 目录**下（字体文件占 16 MB 大小）

安装

```
$ go get -u github.com/golang/freetype
```

绘制文字

打开 service/article_service/article_poster.go 文件，增加绘制文字的业务逻辑，如下：

```
type DrawText struct {
    JPG    draw.Image
    Merged *os.File

    Title string
    X0    int
    Y0    int
    Size0 float64

    SubTitle string
    X1       int
    Y1       int
    Size1    float64
}
```

```
func (a *ArticlePosterBg) DrawPoster(d *DrawText, fontName string) error {
    fontSource := setting.AppSetting.RuntimeRootPath + setting.AppSetting.FontSa
vePath + fontName
    fontSourceBytes, err := ioutil.ReadFile(fontSource)
    if err != nil {
        return err
    }

    trueTypeFont, err := freetype.ParseFont(fontSourceBytes)
    if err != nil {
        return err
    }

    fc := freetype.NewContext()
    fc.SetDPI(72)
    fc.SetFont(trueTypeFont)
    fc.SetFontSize(d.Size0)
    fc.SetClip(d.JPG.Bounds())
    fc.SetDst(d.JPG)
    fc.SetSrc(image.Black)

    pt := freetype.Pt(d.X0, d.Y0)
    _, err = fc.DrawString(d.Title, pt)
    if err != nil {
        return err
    }

    fc.SetFontSize(d.Size1)
    _, err = fc.DrawString(d.SubTitle, freetype.Pt(d.X1, d.Y1))
    if err != nil {
        return err
    }

    err = jpeg.Encode(d.Merged, d.JPG, nil)
    if err != nil {
        return err
    }

    return nil
}
```

这里主要使用了 freetype 包，分别涉及如下细项：

1、freetype.NewContext：创建一个新的 Context，会对其设置一些默认值

```
func NewContext() *Context {
    return &Context{
        r:        raster.NewRasterizer(0, 0),
        fontSize: 12,
        dpi:      72,
        scale:    12 << 6,
    }
}
```

2、fc.SetDPI：设置屏幕每英寸的分辨率

3、fc.SetFont：设置用于绘制文本的字体

4、fc.SetFontSize：以磅为单位设置字体大小

5、fc.SetClip：设置剪裁矩形以进行绘制

6、fc.SetDst：设置目标图像

7、fc.SetSrc：设置绘制操作的源图像，通常为 image.Uniform

```
var (
        // Black is an opaque black uniform image.
        Black = NewUniform(color.Black)
        // White is an opaque white uniform image.
        White = NewUniform(color.White)
        // Transparent is a fully transparent uniform image.
        Transparent = NewUniform(color.Transparent)
        // Opaque is a fully opaque uniform image.
        Opaque = NewUniform(color.Opaque)
)
```

8、fc.DrawString：根据 Pt 的坐标值绘制给定的文本内容

业务逻辑

打开 service/article_service/article_poster.go 方法，在 Generate 方法增加绘制文字的代码逻辑，如下：

```
func (a *ArticlePosterBg) Generate() (string, string, error) {
    fullPath := qrcode.GetQrCodeFullPath()
    fileName, path, err := a.Qr.Encode(fullPath)
    if err != nil {
        return "", "", err
    }
```

```
    if !a.CheckMergedImage(path) {
        ...

        draw.Draw(jpg, jpg.Bounds(), bgImage, bgImage.Bounds().Min, draw.Over)
        draw.Draw(jpg, jpg.Bounds(), qrImage, qrImage.Bounds().Min.Sub(image.Pt
(a.Pt.X, a.Pt.Y)), draw.Over)

        err = a.DrawPoster(&DrawText{
            JPG:    jpg,
            Merged: mergedF,

            Title: "Golang Gin 系列文章",
            X0:    80,
            Y0:    160,
            Size0: 42,

            SubTitle: "---煎鱼",
            X1:       320,
            Y1:       220,
            Size1:    36,
        }, "msyhbd.ttc")

        if err != nil {
            return "", "", err
        }
    }

    return fileName, path, nil
}
```

验证

访问生成文章海报的接口 `$HOST/api/v1/articles/poster/generate?token=$token` ，检查其生成结果，如下图

总结

在本章节在 连载十五 的基础上增加了绘制文字，在实现上并不困难，而这两块需求一般会同时出现，大家可以多加练习，了解里面的逻辑和其他 API 😁

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 02 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

用Nginx部署Go应用

知识点

- Nginx。
- 反向代理。

本文目标

简单部署后端服务。

做什么

在本章节，我们将简单介绍 Nginx 以及使用 Nginx 来完成对 go-gin-example 的部署，会实现反向代理和简单负载均衡的功能。

Nginx

是什么

Nginx 是一个 Web Server，可以用作反向代理、负载均衡、邮件代理、TCP / UDP、HTTP 服务器等等，它拥有很多吸引人的特性，例如：

- 以较低的内存占用率处理 10,000 多个并发连接（每 10k 非活动 HTTP 保持活动连接约 2.5 MB ）
- 静态服务器（处理静态文件）
- 正向、反向代理
- 负载均衡
- 通过 OpenSSL 对 TLS / SSL 与 SNI 和 OCSP 支持
- FastCGI、SCGI、uWSGI 的支持
- WebSockets、HTTP/1.1 的支持
- Nginx + Lua

安装

请右拐谷歌或百度，安装好 Nginx 以备接下来的使用

简单讲解

常用命令

- nginx：启动 Nginx
- nginx -s stop：立刻停止 Nginx 服务
- nginx -s reload：重新加载配置文件
- nginx -s quit：平滑停止 Nginx 服务
- nginx -t：测试配置文件是否正确
- nginx -v：显示 Nginx 版本信息
- nginx -V：显示 Nginx 版本信息、编译器和配置参数的信息

涉及配置

1、 proxy_pass：配置**反向代理的路径**。需要注意的是如果 proxy_pass 的 url 最后为 /，则表示绝对路径。否则（不含变量下）表示相对路径，所有的路径都会被代理过去

2、 upstream：配置**负载均衡**，upstream 默认是以轮询的方式进行负载，另外还支持**四种模式**，分别是：

（1）weight：权重，指定轮询的概率，weight 与访问概率成正比

（2）ip_hash：按照访问 IP 的 hash 结果值分配

（3）fair：按后端服务器响应时间进行分配，响应时间越短优先级别越高

（4）url_hash：按照访问 URL 的 hash 结果值分配

部署

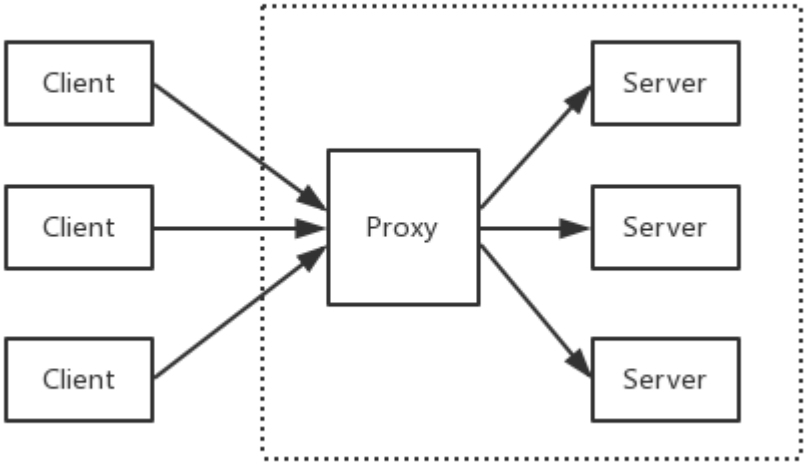
在这里需要对 nginx.conf 进行配置，如果你不知道对应的配置文件是哪个，可执行 `nginx -t` 看一下

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
```

显然，我的配置文件在 `/usr/local/etc/nginx/` 目录下，并且测试通过

反向代理

反向代理是指以代理服务器来接受网络上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。（来自百科）



配置 hosts

由于需要用本机作为演示，因此先把映射配上去，打开 `/etc/hosts` ，增加内容：

```
127.0.0.1       api.blog.com
```

配置 nginx.conf

打开 nginx 的配置文件 nginx.conf（我的是 /usr/local/etc/nginx/nginx.conf），我们做了如下事情：

增加 server 片段的内容，设置 server_name 为 api.blog.com 并且监听 8081 端口，将所有路径转发到 `http://127.0.0.1:8000/` 下

```
worker_processes  1;

events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;
```

```
    sendfile        on;
    keepalive_timeout  65;

    server {
        listen       8081;
        server_name  api.blog.com;

        location / {
            proxy_pass http://127.0.0.1:8000/;
        }
    }
}
```

验证

启动 go-gin-example

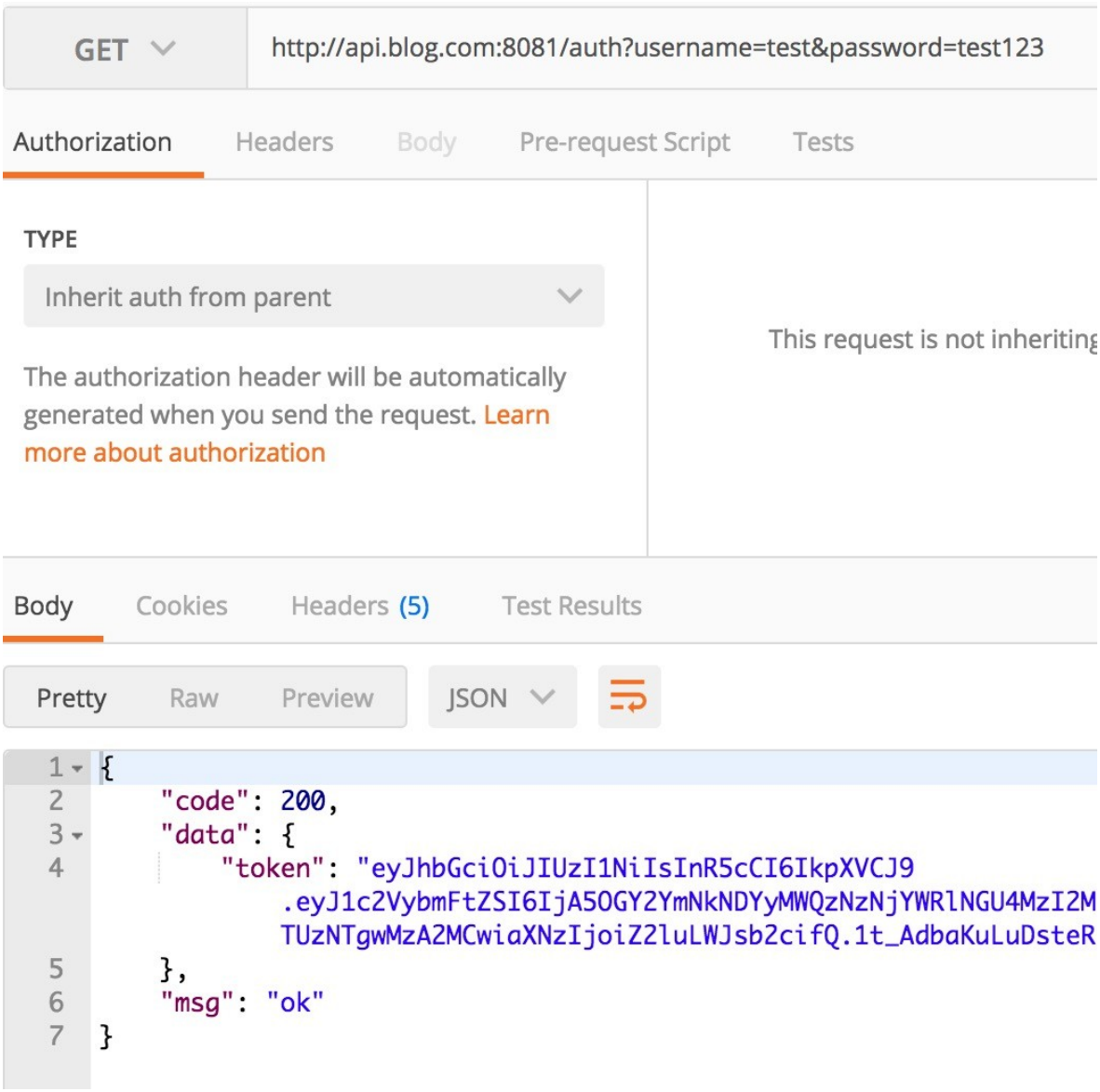
回到 go-gin-example 的项目下，执行 make，再运行 ./go-gin-exmaple

```
$ make
github.com/EDDYCJY/go-gin-example
$ ls
LICENSE        README.md      conf           go-gin-example middleware     pkg
runtime        vendor
Makefile       README_ZH.md   docs           main.go        models         route
rs        service
$ ./go-gin-example
...
[GIN-debug] DELETE /api/v1/articles/:id       --> github.com/EDDYCJY/go-gin-examp
le/routers/api/v1.DeleteArticle (4 handlers)
[GIN-debug] POST   /api/v1/articles/poster/generate --> github.com/EDDYCJY/go-gi
n-example/routers/api/v1.GenerateArticlePoster (4 handlers)
Actual pid is 14672
```

重启 nginx

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
$ nginx -s reload
```

访问接口



如此，就实现了一个简单的反向代理了，是不是很简单呢

负载均衡

负载均衡，英文名称为 Load Balance（常称 LB），其意思就是分摊到多个操作单元上进行执行（来自百科）

你能从运维口中经常听见，XXX 负载怎么突然那么高。 那么它到底是什么呢？

其背后一般有多台 server，系统会根据配置的策略（例如 Nginx 有提供四种选择）来进行动态调整，尽可能的达到各节点均衡，从而提高系统整体的吞吐量和快速响应

如何演示

前提条件为多个后端服务，那么势必需要多个 go-gin-example，为了演示我们可以启动多个端口，达到模拟的效果

为了便于演示，分别在启动前将 conf/app.ini 的应用端口修改为 8001 和 8002（也可以做成传入参数的模式），达到启动 2 个监听 8001 和 8002 的后端服务

配置 nginx.conf

回到 nginx.conf 的老地方，增加负载均衡所需的配置。新增 upstream 节点，设置其对应的 2 个后端服务，最后修改了 proxy_pass 指向（格式为 http:// + upstream 的节点名称）

```
worker_processes  1;

events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    upstream api.blog.com {
        server 127.0.0.1:8001;
        server 127.0.0.1:8002;
    }

    server {
        listen       8081;
        server_name  api.blog.com;

        location / {
            proxy_pass http://api.blog.com/;
        }
    }
}
```

重启 nginx

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
$ nginx -s reload
```

验证

再重复访问 `http://api.blog.com:8081/auth?username={USER_NAME}}&password={PASSWORD}` ，多访问几次便于查看效果

目前 Nginx 没有进行特殊配置，那么它是轮询策略，而 go-gin-example 默认开着 debug 模式，看看请求 log 就明白了

```
[GIN] 2018/09/01 - 18:45:19 | 200 |    2.001245ms |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:45:38 | 200 |     772.371µs |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:46:32 | 200 |    1.986631ms |       127.0.0.1 | GET      /auth?username=test&password=test123
```

```
[GIN] 2018/09/01 - 18:45:20 | 200 |     801.886µs |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:46:29 | 200 |     711.61µs |       127.0.0.1 | GET      /auth?username=test&password=test123
```

总结

在本章节，希望您能够简单习得日常使用的 Web Server 背后都是一些什么逻辑，Nginx 是什么？反向代理？负载均衡？

怎么简单部署，知道了吧。

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

Golang 交叉编译

知识点

- Nginx。
- 反向代理。

本文目标

简单部署后端服务。

做什么

在本章节，我们将简单介绍 Nginx 以及使用 Nginx 来完成对 go-gin-example 的部署，会实现反向代理和简单负载均衡的功能。

Nginx

是什么

Nginx 是一个 Web Server，可以用作反向代理、负载均衡、邮件代理、TCP / UDP、HTTP 服务器等等，它拥有很多吸引人的特性，例如：

- 以较低的内存占用率处理 10,000 多个并发连接（每 10k 非活动 HTTP 保持活动连接约 2.5 MB ）
- 静态服务器（处理静态文件）
- 正向、反向代理
- 负载均衡
- 通过 OpenSSL 对 TLS / SSL 与 SNI 和 OCSP 支持
- FastCGI、SCGI、uWSGI 的支持
- WebSockets、HTTP/1.1 的支持
- Nginx + Lua

安装

请右拐谷歌或百度，安装好 Nginx 以备接下来的使用

简单讲解

常用命令

- nginx：启动 Nginx
- nginx -s stop：立刻停止 Nginx 服务
- nginx -s reload：重新加载配置文件
- nginx -s quit：平滑停止 Nginx 服务
- nginx -t：测试配置文件是否正确
- nginx -v：显示 Nginx 版本信息
- nginx -V：显示 Nginx 版本信息、编译器和配置参数的信息

涉及配置

1、 proxy_pass：配置**反向代理的路径**。需要注意的是如果 proxy_pass 的 url 最后为 /，则表示绝对路径。否则（不含变量下）表示相对路径，所有的路径都会被代理过去

2、 upstream：配置**负载均衡**，upstream 默认是以轮询的方式进行负载，另外还支持**四种模式**，分别是：

（1）weight：权重，指定轮询的概率，weight 与访问概率成正比

（2）ip_hash：按照访问 IP 的 hash 结果值分配

（3）fair：按后端服务器响应时间进行分配，响应时间越短优先级别越高

（4）url_hash：按照访问 URL 的 hash 结果值分配

部署

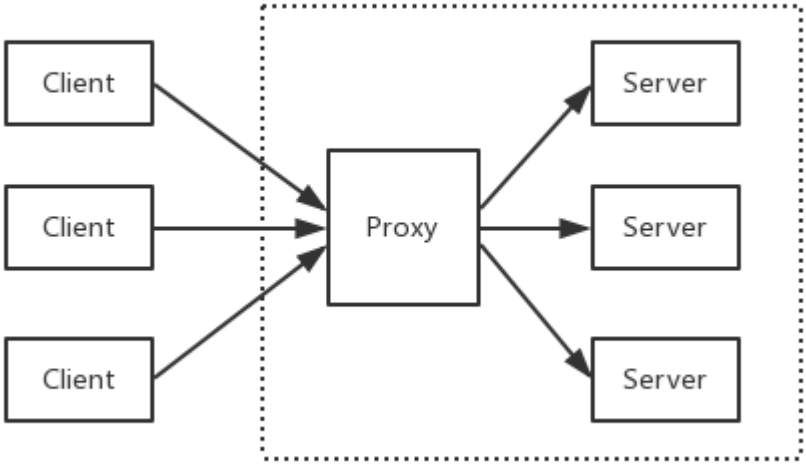
在这里需要对 nginx.conf 进行配置，如果你不知道对应的配置文件是哪个，可执行 `nginx -t` 看一下

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
```

显然，我的配置文件在 `/usr/local/etc/nginx/` 目录下，并且测试通过

反向代理

反向代理是指以代理服务器来接受网络上的连接请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回给请求连接的客户端，此时代理服务器对外就表现为一个反向代理服务器。（来自百科）



配置 hosts

由于需要用本机作为演示，因此先把映射配上去，打开 `/etc/hosts` ，增加内容：

```
127.0.0.1       api.blog.com
```

配置 nginx.conf

打开 nginx 的配置文件 nginx.conf（我的是 /usr/local/etc/nginx/nginx.conf），我们做了如下事情：

增加 server 片段的内容，设置 server_name 为 api.blog.com 并且监听 8081 端口，将所有路径转发到 `http://127.0.0.1:8000/` 下

```
worker_processes  1;

events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;
```

```
    sendfile        on;
    keepalive_timeout  65;

    server {
        listen       8081;
        server_name  api.blog.com;

        location / {
            proxy_pass http://127.0.0.1:8000/;
        }
    }
}
```

验证

启动 go-gin-example

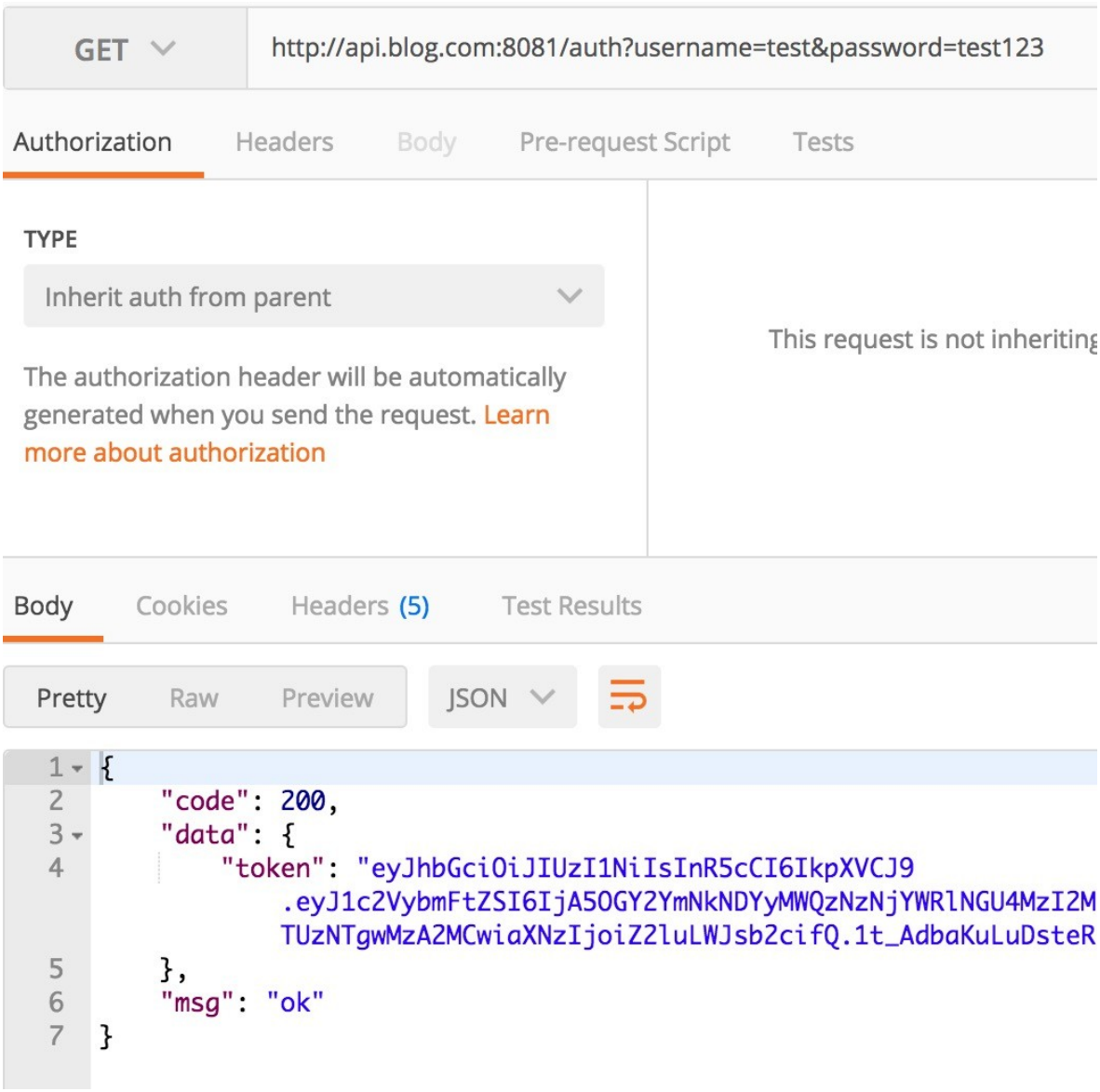
回到 go-gin-example 的项目下，执行 make，再运行 ./go-gin-exmaple

```
$ make
github.com/EDDYCJY/go-gin-example
$ ls
LICENSE        README.md      conf           go-gin-example middleware     pkg
runtime        vendor
Makefile       README_ZH.md   docs           main.go        models         route
rs        service
$ ./go-gin-example
...
[GIN-debug] DELETE /api/v1/articles/:id      --> github.com/EDDYCJY/go-gin-examp
le/routers/api/v1.DeleteArticle (4 handlers)
[GIN-debug] POST   /api/v1/articles/poster/generate --> github.com/EDDYCJY/go-gi
n-example/routers/api/v1.GenerateArticlePoster (4 handlers)
Actual pid is 14672
```

重启 nginx

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
$ nginx -s reload
```

访问接口



如此，就实现了一个简单的反向代理了，是不是很简单呢

负载均衡

负载均衡，英文名称为 Load Balance（常称 LB），其意思就是分摊到多个操作单元上进行执行（来自百科）

你能从运维口中经常听见，XXX 负载怎么突然那么高。 那么它到底是什么呢？

其背后一般有多台 server，系统会根据配置的策略（例如 Nginx 有提供四种选择）来进行动态调整，尽可能的达到各节点均衡，从而提高系统整体的吞吐量和快速响应

如何演示

前提条件为多个后端服务，那么势必需要多个 go-gin-example，为了演示我们可以启动多个端口，达到模拟的效果

为了便于演示，分别在启动前将 conf/app.ini 的应用端口修改为 8001 和 8002（也可以做成传入参数的模式），达到启动 2 个监听 8001 和 8002 的后端服务

配置 nginx.conf

回到 nginx.conf 的老地方，增加负载均衡所需的配置。新增 upstream 节点，设置其对应的 2 个后端服务，最后修改了 proxy_pass 指向（格式为 http:// + upstream 的节点名称）

```
worker_processes  1;

events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    upstream api.blog.com {
        server 127.0.0.1:8001;
        server 127.0.0.1:8002;
    }

    server {
        listen       8081;
        server_name  api.blog.com;

        location / {
            proxy_pass http://api.blog.com/;
        }
    }
}
```

重启 nginx

```
$ nginx -t
nginx: the configuration file /usr/local/etc/nginx/nginx.conf syntax is ok
nginx: configuration file /usr/local/etc/nginx/nginx.conf test is successful
$ nginx -s reload
```

验证

再重复访问 `http://api.blog.com:8081/auth?username={USER_NAME}}&password={PASSWORD}` ，多访问几次便于查看效果

目前 Nginx 没有进行特殊配置，那么它是轮询策略，而 go-gin-example 默认开着 debug 模式，看看请求 log 就明白了

```
[GIN] 2018/09/01 - 18:45:19 | 200 |    2.001245ms |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:45:38 | 200 |     772.371µs |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:46:32 | 200 |    1.986631ms |       127.0.0.1 | GET      /auth?username=test&password=test123
```

```
[GIN] 2018/09/01 - 18:45:20 | 200 |     801.886µs |       127.0.0.1 | GET      /auth?username=test&password=test123
[GIN] 2018/09/01 - 18:46:29 | 200 |     711.61µs |       127.0.0.1 | GET      /auth?username=test&password=test123
```

总结

在本章节，希望您能够简单习得日常使用的 Web Server 背后都是一些什么逻辑，Nginx 是什么？反向代理？负载均衡？

怎么简单部署，知道了吧。

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

？

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

请入门 Makefile

知识点

- 写一个 Makefile

本文目标

含一定复杂度的软件工程，基本上都是先编译 A，再依赖 B，再编译 C...，最后才执行构建。如果每次都人为编排，又或是每新来一个同事就问你项目 D 怎么构建、重新构建需要注意什么...等等情况，岂不是要崩溃？

我们常常会在开源项目中发现 Makefile，你是否有过疑问？

本章节会简单介绍 Makefile 的使用方式，最后建议深入学习。

怎么解决

对于构建编排，Docker 有 Dockerfile ，在 Unix 中有神器 Make ....

Make

是什么

Make 是一个构建自动化工具，会在当前目录下寻找 Makefile 或 makefile 文件。如果存在，会依据 Makefile 的**构建规则**去完成构建

当然了，实际上 Makefile 内都是你根据 make 语法规则，自己编写的特定 Shell 命令等

它是一个工具，规则也很简单。在支持的范围内，编译 A， 依赖 B，再编译 C，完全没问题

规则

Makefile 由多条规则组成，每条规则都以一个 target（目标）开头，后跟一个：冒号，冒号后是这一个目标的 prerequisites（前置条件）

紧接着新的一行，必须以一个 tab 作为开头，后面跟随 command（命令），也就是你希望这一个 target 所执行的构建命令

```
[target] ... : [prerequisites] ...
<tab>[command]
    ...
    ...
```

- target：一个目标代表一条规则，可以是一个或多个文件名。也可以是某个操作的名字（标签），称为**伪目标（phony）**
- prerequisites：前置条件，这一项是**可选参数**。通常是多个文件名、伪目标。它的作用是 target 是否需要重新构建的标准，如果前置条件不存在或有过更新（文件的最后一次修改时间）则认为 target 需要重新构建
- command：构建这一个 target 的具体命令集

简单的例子

本文将以 go-gin-example 去编写 Makefile 文件，请跨入 make 的大门

分析

在编写 Makefile 前，需要先分析构建先后顺序、依赖项，需要解决的问题等

编写

```
.PHONY: build clean tool lint help

all: build

build:
    go build -v .

tool:
    go tool vet . |& grep -v vendor; true
    gofmt -w .

lint:
    golint ./...

clean:
    rm -rf go-gin-example
    go clean -i .

help:
    @echo "make: compile packages and dependencies"
```

```
    @echo "make tool: run specified go tool"
    @echo "make lint: golint ./..."
    @echo "make clean: remove object files and cached files"
```

1、在上述文件中，使用了 `.PHONY` ，其作用是声明 build / clean / tool / lint / help 为**伪目标**，声明为伪目标会怎么样呢？

- 声明为伪目标后：在执行对应的命令时，make 就不会去检查是否存在 build / clean / tool / lint / help 其对应的文件，而是每次都会运行标签对应的命令
- 若不声明：恰好存在对应的文件，则 make 将会认为 xx 文件已存在，没有重新构建的必要了

2、这块比较简单，在命令行执行即可看见效果，实现了以下功能：

1. make: make 就是 make all
2. make build: 编译当前项目的包和依赖项
3. make tool: 运行指定的 Go 工具集
4. make lint: golint 一下
5. make clean: 删除对象文件和缓存文件
6. make help: help

为什么会打印执行的命令

如果你实际操作过，可能会有疑问。明明只是执行命令，为什么会打印到标准输出上了？

原因

make 默认会打印每条命令，再执行。这个行为被定义为**回声**

解决

可以在对应命令前加上 @，可指定该命令不被打印到标准输出上

```
build:
    @go build -v .
```

那么还有其他的特殊符号吗？有的，请课后去了解下 +、- 的用途

小结

这是一篇比较简洁的文章，希望可以让您对 Makefile 有一个基本了解。

参考

本系列示例代码

- go-gin-example

关于

修改记录

- 第一版：2018 年 02 月 16 日发布文章
- 第二版：2019 年 10 月 01 日修改文章

?

如果有任何疑问或错误，欢迎在 issues 进行提问或给予修正意见，如果喜欢或对你有所帮助，欢迎 Star，对作者是一种鼓励和推进。

我的公众号

我要煎鱼说

微信扫描二维码，关注一只普通的煎鱼

gRPC 应用

gRPC及相关介绍

项目地址：https://github.com/EDDYCJY/go-grpc-example

作为开篇章，将会介绍 gRPC 相关的一些知识。简单来讲 gRPC 是一个 基于 HTTP/2 协议设计的 RPC 框架，它采用了 Protobuf 作为 IDL

你是否有过疑惑，它们都是些什么？本文将会介绍一些常用的知识和概念，更详细的会给出手册地址去深入

一、RPC

什么是 RPC

RPC 代指远程过程调用（Remote Procedure Call），它的调用包含了传输协议和编码（对象序列号）协议等等。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，而开发人员无需额外地为这个交互作用编程

实际场景：

有两台服务器，分别是 A、B。在 A 上的应用 C 想要调用 B 服务器上的应用 D，它们可以直接本地调用吗？
答案是不能的，但走 RPC 的话，十分方便。因此常有人称使用 RPC，就跟本地调用一个函数一样简单

RPC 框架

我认为，一个完整的 RPC 框架，应包含负载均衡、服务注册和发现、服务治理等功能，并具有可拓展性便于流量监控系统等接入
那么它才算完整的，当然了。有些较单一的 RPC 框架，通过组合多组件也能达到这个标准

你认为呢？

常见 RPC 框架

- gRPC
- Thrift
- Rpcx
- Dubbo

比较一下

\	跨语言	多 IDL	服务治理	注册中心	服务管理
gRPC	√	×	×	×	×
Thrift	√	×	×	×	×
Rpcx	×	√	√	√	√
Dubbo	×	√	√	√	√

为什么要 RPC

简单、通用、安全、效率

RPC 可以基于 HTTP 吗

RPC 是代指远程过程调用，是可以基于 HTTP 协议的

肯定会有人说效率优势，我可以告诉你，那是基于 HTTP/1.1 来讲的，HTTP/2 优化了许多问题（当然也存在新的问题），所以你看到了本文的主题 gRPC

二、Protobuf

介绍

Protocol Buffers 是一种与语言、平台无关，可扩展的序列化结构化数据的方法，常用于通信协议，数据存储等等。相较于 JSON、XML，它更小、更快、更简单，因此也更受开发人员的青睐

语法

```
syntax = "proto3";

service SearchService {
    rpc Search (SearchRequest) returns (SearchResponse);
}

message SearchRequest {
  string query = 1;
  int32 page_number = 2;
  int32 result_per_page = 3;
}
```

```
message SearchResponse {
    ...
}
```

1、第一行（非空的非注释行）声明使用 `proto3` 语法。如果不声明，将默认使用 `proto2` 语法。同时我建议用 v2 还是 v3，都应当声明其使用的版本

2、定义 `SearchService` RPC 服务，其包含 RPC 方法 `Search` ，入参为 `SearchRequest` 消息，出参为 `SearchResponse` 消息

3、定义 `SearchRequest` 、 `SearchResponse` 消息，前者定义了三个字段，每一个字段包含三个属性：类型、字段名称、字段编号

4、Protobuf 编译器会根据选择的语言不同，生成相应语言的 Service Interface Code 和 Stubs

最后，这里只是简单的语法介绍，详细的请右拐
Language Guide (proto3)

数据类型

.proto Type	C++ Type	Java Type	Go Type	PHP Type
double	double	double	float64	float
float	float	float	float32	float
int32	int32	int	int32	integer
int64	int64	long	int64	integer/string
uint32	uint32	int	uint32	integer
uint64	uint64	long	uint64	integer/string
sint32	int32	int	int32	integer
sint64	int64	long	int64	integer/string
fixed32	uint32	int	uint32	integer
fixed64	uint64	long	uint64	integer/string
sfixed32	int32	int	int32	integer
sfixed64	int64	long	int64	integer/string

.proto Type	C++ Type	Java Type	Go Type	PHP Type
bool	bool	boolean	bool	boolean
string	string	String	string	string
bytes	string	ByteString	[]byte	string

v2 和 v3 主要区别

- 删除原始值字段的字段存在逻辑
- 删除 required 字段
- 删除 optional 字段，默认就是
- 删除 default 字段
- 删除扩展特性，新增 Any 类型来替代它
- 删除 unknown 字段的支持
- 新增 JSON Mapping
- 新增 Map 类型的支持
- 修复 enum 的 unknown 类型
- repeated 默认使用 packed 编码
- 引入了新的语言实现（C#，JavaScript，Ruby，Objective-C）

以上是日常涉及的常见功能，如果还想详细了解可阅读 Protobuf Version 3.0.0

相较 Protobuf，为什么不使用 XML？

- 更简单
- 数据描述文件只需原来的 1/10 至 1/3
- 解析速度是原来的 20 倍至 100 倍
- 减少了二义性
- 生成了更易使用的数据访问类

三、gRPC

介绍

gRPC 是一个高性能、开源和通用的 RPC 框架，面向移动和 HTTP/2 设计

多语言

- C++
- C#
- Dart
- Go
- Java
- Node.js
- Objective-C
- PHP
- Python
- Ruby

特点

1、HTTP/2

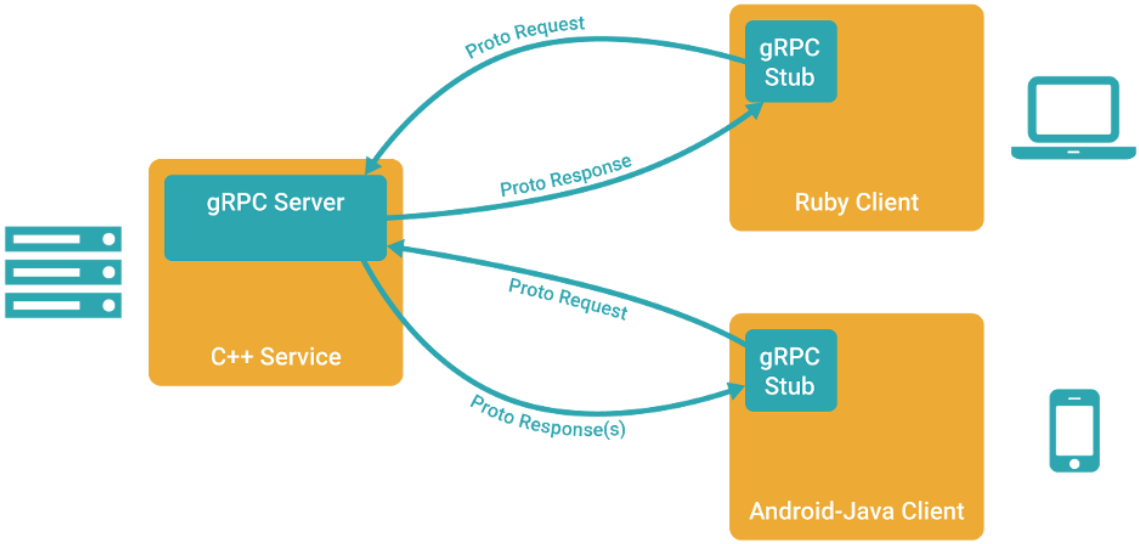
2、Protobuf

3、客户端、服务端基于同一份 IDL

4、移动网络的良好支持

5、支持多语言

概览



讲解

1、客户端（gRPC Sub）调用 A 方法，发起 RPC 调用

2、对请求信息使用 Protobuf 进行对象序列化压缩（IDL）

3、服务端（gRPC Server）接收到请求后，解码请求体，进行业务逻辑处理并返回

4、对响应结果使用 Protobuf 进行对象序列化压缩（IDL）

5、客户端接受到服务端响应，解码请求体。回调被调用的 A 方法，唤醒正在等待响应（阻塞）的客户端调用并返回响应结果

示例

在这一小节，将简单的给大家展示 gRPC 的客户端和服务端的示例代码，希望大家先有一个基础的印象，将会在下一章节详细介绍

构建和启动服务端

```
lis, err := net.Listen("tcp", fmt.Sprintf(":%d", *port))
if err != nil {
        log.Fatalf("failed to listen: %v", err)
}

grpcServer := grpc.NewServer()
```

```
...
pb.RegisterSearchServer(grpcServer, &SearchServer{})
grpcServer.Serve(lis)
```

1、监听指定 TCP 端口，用于接受客户端请求

2、创建 gRPC Server 的实例对象

3、gRPC Server 内部服务和路由的注册

4、Serve() 调用服务器以执行阻塞等待，直到进程被终止或被 Stop() 调用

创建客户端

```
var opts []grpc.DialOption
...
conn, err := grpc.Dial(*serverAddr, opts...)
if err != nil {
    log.Fatalf("fail to dial: %v", err)
}

defer conn.Close()
client := pb.NewSearchClient(conn)
...
```

1、创建 gRPC Channel 与 gRPC Server 进行通信（需服务器地址和端口作为参数）

2、设置 DialOptions 凭证（例如，TLS，GCE 凭据，JWT 凭证）

3、创建 Search Client Stub

4、调用对应的服务方法

思考题

1、什么场景下不适合使用 Protobuf，而适合使用 JSON、XML？

2、Protobuf 一节中提到的 packed 编码，是什么？

总结

在开篇内容中，我利用了尽量简短的描述给你介绍了接下来所必须、必要的知识点
希望你能够有所收获，建议能到我给的参考资料处进行深入学习，是最好的了

参考资料

- Protocol Buffers
- gRPC

gRPC Client and Server

前言

本章节将使用 Go 来编写 gRPC Server 和 Client，让其互相通讯。在此之上会使用到如下库：

- google.golang.org/grpc
- github.com/golang/protobuf/protoc-gen-go

安装

gRPC

```
go get -u google.golang.org/grpc
```

Protocol Buffers v3

```
wget https://github.com/google/protobuf/releases/download/v3.5.1/protobuf-all-3.5.1.zip
unzip protobuf-all-3.5.1.zip
cd protobuf-3.5.1/
./configure
make
make install
```

检查是否安装成功

```
protoc --version
```

若出现以下错误，执行 `ldconfig` 命名就能解决这问题

```
protoc: error while loading shared libraries: libprotobuf.so.15: cannot open shared object file: No such file or directory
```

Protoc Plugin

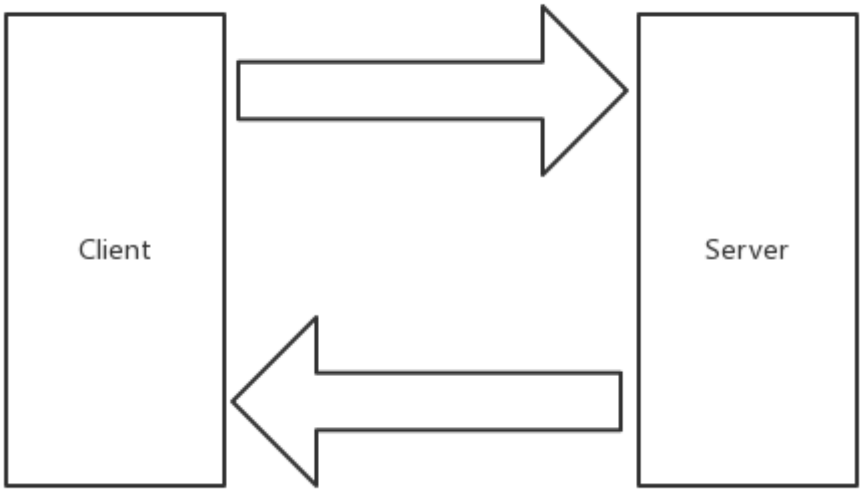
```
go get -u github.com/golang/protobuf/protoc-gen-go
```

安装环境若有问题，可参考我先前的文章 《介绍与环境安装》 内有详细介绍，不再赘述

gRPC

本小节开始正式编写 gRPC 相关的程序，一起上车吧 😄

图示



目录结构

```
$ tree go-grpc-example
go-grpc-example
├── client
├── proto
│   └── search.proto
└── server.go
```

IDL

编写

在 proto 文件夹下的 search.proto 文件中，写入如下内容：

```
syntax = "proto3";

package proto;

service SearchService {
    rpc Search(SearchRequest) returns (SearchResponse) {}
}

message SearchRequest {
    string request = 1;
}

message SearchResponse {
    string response = 1;
}
```

生成

在 proto 文件夹下执行如下命令：

```
$ protoc --go_out=plugins=grpc:. *.proto
```

- plugins=plugin1+plugin2：指定要加载的子插件列表

我们定义的 proto 文件是涉及了 RPC 服务的，而默认是不会生成 RPC 代码的，因此需要给出 `plugins` 参数传递给 `protoc-gen-go`，告诉它，请支持 RPC（这里指定了 gRPC）

- -go_out=.：设置 Go 代码输出的目录

该指令会加载 protoc-gen-go 插件达到生成 Go 代码的目的，生成的文件以 .pb.go 为文件后缀

- :（冒号）

冒号充当分隔符的作用，后跟所需要的参数集。如果这处不涉及 RPC，命令可简化为：

```
$ protoc --go_out=. *.proto
```

注：建议你看看两条命令生成的 .pb.go 文件，分别有什么区别

生成后

执行完毕命令后，将得到一个 .pb.go 文件，文件内容如下：

```
type SearchRequest struct {
    Request              string   `protobuf:"bytes,1,opt,name=request" json:"req
uest,omitempty"`
    XXX_NoUnkeyedLiteral struct{} `json:"-"`
    XXX_unrecognized     []byte   `json:"-"`
    XXX_sizecache        int32    `json:"-"`
}

func (m *SearchRequest) Reset()         { *m = SearchRequest{} }
func (m *SearchRequest) String() string { return proto.CompactTextString(m) }
func (*SearchRequest) ProtoMessage()    {}
func (*SearchRequest) Descriptor() ([]byte, []int) {
    return fileDescriptor_search_8b45f79ee13ff6a3, []int{0}
}

func (m *SearchRequest) GetRequest() string {
    if m != nil {
        return m.Request
    }
    return ""
}
```

通过阅读这一部分代码，可以知道主要涉及如下方面：

- 字段名称从小写下划线转换为大写驼峰模式（字段导出）
- 生成一组 Getters 方法，能便于处理一些空指针取值的情况
- ProtoMessage 方法实现 proto.Message 的接口
- 生成 Rest 方法，便于将 Protobuf 结构体恢复为零值
- Repeated 转换为切片

```
type SearchRequest struct {
    Request              string   `protobuf:"bytes,1,opt,name=request" json:"req
uest,omitempty"`
}

func (*SearchRequest) Descriptor() ([]byte, []int) {
    return fileDescriptor_search_8b45f79ee13ff6a3, []int{0}
}

type SearchResponse struct {
    Response             string   `protobuf:"bytes,1,opt,name=response" json:"re
sponse,omitempty"`
}
```

```
func (*SearchResponse) Descriptor() ([]byte, []int) {
    return fileDescriptor_search_8b45f79ee13ff6a3, []int{1}
}

...

func init() { proto.RegisterFile("search.proto", fileDescriptor_search_8b45f79ee
13ff6a3) }

var fileDescriptor_search_8b45f79ee13ff6a3 = []byte{
    // 131 bytes of a gzipped FileDescriptorProto
    0x1f, 0x8b, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, 0x02, 0xff, 0xe2, 0xe2, 0x29
, 0x4e, 0x4d, 0x2c,
    0x4a, 0xce, 0xd0, 0x2b, 0x28, 0xca, 0x2f, 0xc9, 0x17, 0x62, 0x05, 0x53, 0x4a
, 0x9a, 0x5c, 0xbc,
    0xc1, 0x60, 0xe1, 0xa0, 0xd4, 0xc2, 0xd2, 0xd4, 0xe2, 0x12, 0x21, 0x09, 0x2e
, 0xf6, 0x22, 0x08,
    0x53, 0x82, 0x51, 0x81, 0x51, 0x83, 0x33, 0x08, 0xc6, 0x55, 0xd2, 0xe1, 0xe2
, 0x83, 0x29, 0x2d,
    0x2e, 0xc8, 0xcf, 0x2b, 0x4e, 0x15, 0x92, 0xe2, 0xe2, 0x28, 0x82, 0xb2, 0xa1
, 0x8a, 0xe1, 0x7c,
    0x23, 0x0f, 0x98, 0xc1, 0xc1, 0xa9, 0x45, 0x65, 0x99, 0xc9, 0xa9, 0x42, 0xe6
, 0x5c, 0x6c, 0x10,
    0x01, 0x21, 0x11, 0x88, 0x13, 0xf4, 0x50, 0x2c, 0x96, 0x12, 0x45, 0x13, 0x85
, 0x98, 0xa3, 0xc4,
    0x90, 0xc4, 0x06, 0x16, 0x37, 0x06, 0x04, 0x00, 0x00, 0xff, 0xff, 0xf3, 0xba
, 0x74, 0x95, 0xc0,
    0x00, 0x00, 0x00,
}
```

而这一部分代码主要是围绕 `fileDescriptor` 进行，在这里 `fileDescriptor_search_8b45f79ee13ff6a3` 表示一个编译后的 proto 文件，而每一个方法都包含 Descriptor 方法，代表着这一个方法在 `fileDescriptor` 中具体的 Message Field

Server

这一小节将编写 gRPC Server 的基础模板，完成一个方法的调用。对 server.go 写入如下内容：

```
package main

import (
    "context"
```

```
    "log"
    "net"

    "google.golang.org/grpc"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

type SearchService struct{}

func (s *SearchService) Search(ctx context.Context, r *pb.SearchRequest) (*pb.Se
archResponse, error) {
    return &pb.SearchResponse{Response: r.GetRequest() + " Server"}, nil
}

const PORT = "9001"

func main() {
    server := grpc.NewServer()
    pb.RegisterSearchServiceServer(server, &SearchService{})

    lis, err := net.Listen("tcp", ":"+PORT)
    if err != nil {
        log.Fatalf("net.Listen err: %v", err)
    }

    server.Serve(lis)
}
```

- 创建 gRPC Server 对象，你可以理解为它是 Server 端的抽象对象
- 将 SearchService（其包含需要被调用的服务端接口）注册到 gRPC Server 的内部注册中心。这样可以在接受到请求时，通过内部的服务发现，发现该服务端接口并转接进行逻辑处理
- 创建 Listen，监听 TCP 端口
- gRPC Server 开始 lis.Accept，直到 Stop 或 GracefulStop

Client

接下来编写 gRPC Go Client 的基础模板，打开 client/client.go 文件，写入以下内容：

```
package main

import (
```

```
    "context"
    "log"

    "google.golang.org/grpc"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const PORT = "9001"

func main() {
    conn, err := grpc.Dial(":"+PORT, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("grpc.Dial err: %v", err)
    }
    defer conn.Close()

    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(context.Background(), &pb.SearchRequest{
        Request: "gRPC",
    })
    if err != nil {
        log.Fatalf("client.Search err: %v", err)
    }

    log.Printf("resp: %s", resp.GetResponse())
}
```

- 创建与给定目标（服务端）的连接交互
- 创建 SearchService 的客户端对象
- 发送 RPC 请求，等待同步响应，得到回调后返回响应结果
- 输出响应结果

验证

启动 Server

```
$ pwd
$GOPATH/github.com/EDDYCJY/go-grpc-example
$ go run server.go
```

启动 Client

```
$ pwd
$GOPATH/github.com/EDDYCJY/go-grpc-example/client
$ go run client.go
2018/09/23 11:06:23 resp: gRPC Server
```

总结

在本章节，我们对 Protobuf、gRPC Client/Server 分别都进行了介绍。希望你结合文中讲述内容再写一个 Demo 进行深入了解，肯定会更棒

参考

本系列示例代码

- go-grpc-example

gRPC Streaming, Client and Server

前言

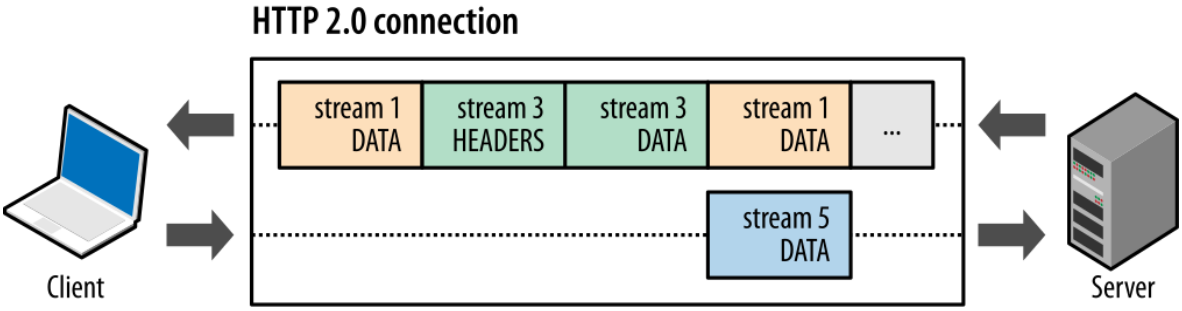
本章节将介绍 gRPC 的流式，分为三种类型：

- Server-side streaming RPC：服务器端流式 RPC
- Client-side streaming RPC：客户端流式 RPC
- Bidirectional streaming RPC：双向流式 RPC

流

任何技术，因为有痛点，所以才有了存在的必要性。如果您想要了解 gRPC 的流式调用，请继续

图



gRPC Streaming 是基于 HTTP/2 的，后续章节再进行详细讲解

为什么不用 Simple RPC

流式为什么要存在呢，是 Simple RPC 有什么问题吗？通过模拟业务场景，可得知在使用 Simple RPC 时，有如下问题：

- 数据包过大造成的瞬时压力
- 接收数据包时，需要所有数据包都接受成功且正确后，才能够回调响应，进行业务处理（无法客户端边发送，服务端边处理）

为什么用 Streaming RPC

- 大规模数据包
- 实时场景

模拟场景

每天早上 6 点，都有一批百万级别的数据集要同从 A 同步到 B，在同步的时候，会做一系列操作（归档、数据分析、画像、日志等）。这一次性涉及的数据量确实大

在同步完成后，也有人马上会去查阅数据，为了新的一天筹备。也符合实时性。

两者相较下，这个场景下更适合使用 Streaming RPC

gRPC

在讲解具体的 gRPC 流式代码时，会**着重在第一节讲解**，因为三种模式其实是不同的组合。希望你能够注重理解，举一反三，其实都是一样的知识点

目录结构

```
$ tree go-grpc-example
go-grpc-example
├── client
│   ├── simple_client
│   │   └── client.go
│   └── stream_client
│       └── client.go
├── proto
│   ├── search.proto
│   └── stream.proto
└── server
    ├── simple_server
    │   └── server.go
    └── stream_server
        └── server.go
```

增加 stream_server、stream_client 存放服务端和客户端文件，proto/stream.proto 用于编写 IDL

IDL

在 proto 文件夹下的 stream.proto 文件中，写入如下内容：

```
syntax = "proto3";

package proto;

service StreamService {
    rpc List(StreamRequest) returns (stream StreamResponse) {};

    rpc Record(stream StreamRequest) returns (StreamResponse) {};

    rpc Route(stream StreamRequest) returns (stream StreamResponse) {};
}


message StreamPoint {
  string name = 1;
  int32 value = 2;
}

message StreamRequest {
  StreamPoint pt = 1;
}

message StreamResponse {
  StreamPoint pt = 1;
}
```

注意关键字 stream，声明其为一个流方法。这里共涉及三个方法，对应关系为

- List：服务器端流式 RPC
- Record：客户端流式 RPC
- Route：双向流式 RPC

基础模板 + 空定义

Server

```
package main

import (
    "log"
    "net"

    "google.golang.org/grpc"
```

```
    pb "github.com/EDDYCJY/go-grpc-example/proto"

)

type StreamService struct{}

const (
    PORT = "9002"
)

func main() {
    server := grpc.NewServer()
    pb.RegisterStreamServiceServer(server, &StreamService{})

    lis, err := net.Listen("tcp", ":"+PORT)
    if err != nil {
        log.Fatalf("net.Listen err: %v", err)
    }

    server.Serve(lis)
}

func (s *StreamService) List(r *pb.StreamRequest, stream pb.StreamService_ListSe
rver) error {
    return nil
}

func (s *StreamService) Record(stream pb.StreamService_RecordServer) error {
    return nil
}

func (s *StreamService) Route(stream pb.StreamService_RouteServer) error {
    return nil
}
```

写代码前，建议先将 gRPC Server 的基础模板和接口给空定义出来。若有不清楚可参见上一章节的知识点

Client

```
package main

import (
    "log"
```

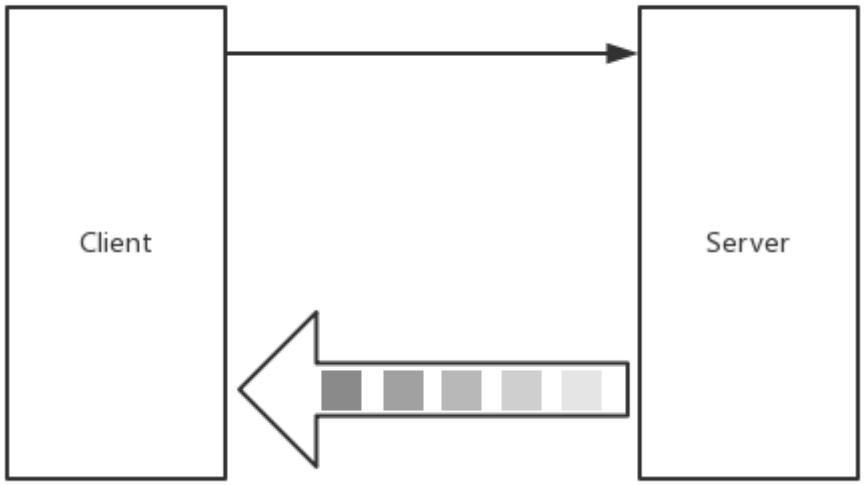
```
    "google.golang.org/grpc"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const (
    PORT = "9002"
)

func main() {
    conn, err := grpc.Dial(":"+PORT, grpc.WithInsecure())
    if err != nil {
        log.Fatalf("grpc.Dial err: %v", err)
    }

    defer conn.Close()

    client := pb.NewStreamServiceClient(conn)

    err = printLists(client, &pb.StreamRequest{Pt: &pb.StreamPoint{Name: "gRPC S
tream Client: List", Value: 2018}})
    if err != nil {
        log.Fatalf("printLists.err: %v", err)
    }

    err = printRecord(client, &pb.StreamRequest{Pt: &pb.StreamPoint{Name: "gRPC
 Stream Client: Record", Value: 2018}})
    if err != nil {
        log.Fatalf("printRecord.err: %v", err)
    }

    err = printRoute(client, &pb.StreamRequest{Pt: &pb.StreamPoint{Name: "gRPC S
tream Client: Route", Value: 2018}})
    if err != nil {
        log.Fatalf("printRoute.err: %v", err)
    }
}

func printLists(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    return nil
}

func printRecord(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    return nil
}
```

```
func printRoute(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    return nil
}
```

一、Server-side streaming RPC：服务器端流式 RPC

服务器端流式 RPC，显然是单向流，并代指 Server 为 Stream 而 Client 为普通 RPC 请求

简单来讲就是客户端发起一次普通的 RPC 请求，服务端通过流式响应多次发送数据集，客户端 Recv 接收数据集。大致如图：



Server

```
func (s *StreamService) List(r *pb.StreamRequest, stream pb.StreamService_ListSe
rver) error {
    for n := 0; n <= 6; n++ {
        err := stream.Send(&pb.StreamResponse{
            Pt: &pb.StreamPoint{
                Name:  r.Pt.Name,
                Value: r.Pt.Value + int32(n),
            },
        })
        if err != nil {
            return err
        }
    }
```

```
    return nil
}
```

在 Server，主要留意 `stream.Send` 方法。它看上去能发送 N 次？有没有大小限制？

```
type StreamService_ListServer interface {
    Send(*StreamResponse) error
    grpc.ServerStream
}

func (x *streamServiceListServer) Send(m *StreamResponse) error {
    return x.ServerStream.SendMsg(m)
}
```

通过阅读源码，可得知是 protoc 在生成时，根据定义生成了各式各样符合标准的接口方法。最终再统一调度内部的 `SendMsg` 方法，该方法涉及以下过程:

- 消息体（对象）序列化
- 压缩序列化后的消息体
- 对正在传输的消息体增加 5 个字节的 header
- 判断压缩+序列化后的消息体总字节长度是否大于预设的 maxSendMessageSize（预设值为 `math.MaxInt32`），若超出则提示错误
- 写入给流的数据集

Client

```
func printLists(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    stream, err := client.List(context.Background(), r)
    if err != nil {
        return err
    }

    for {
        resp, err := stream.Recv()
        if err == io.EOF {
            break
        }
        if err != nil {
            return err
        }

        log.Printf("resp: pj.name: %s, pt.value: %d", resp.Pt.Name, resp.Pt.Valu
```

```
e)
    }

    return nil
}
```

在 Client，主要留意 `stream.Recv()` 方法。什么情况下 `io.EOF` ？什么情况下存在错误信息呢?

```
type StreamService_ListClient interface {
    Recv() (*StreamResponse, error)
    grpc.ClientStream
}

func (x *streamServiceListClient) Recv() (*StreamResponse, error) {
    m := new(StreamResponse)
    if err := x.ClientStream.RecvMsg(m); err != nil {
        return nil, err
    }
    return m, nil
}
```

RecvMsg 会从流中读取完整的 gRPC 消息体，另外通过阅读源码可得知：

（1）RecvMsg 是阻塞等待的

（2）RecvMsg 当流成功/结束（调用了 Close）时，会返回 `io.EOF`

（3）RecvMsg 当流出现任何错误时，流会被中止，错误信息会包含 RPC 错误码。而在 RecvMsg 中可能出现如下错误：

- io.EOF
- io.ErrUnexpectedEOF
- transport.ConnectionError
- google.golang.org/grpc/codes

同时需要注意，默认的 MaxReceiveMessageSize 值为 1024 _ 1024 _ 4，建议不要超出

验证

运行 stream_server/server.go：

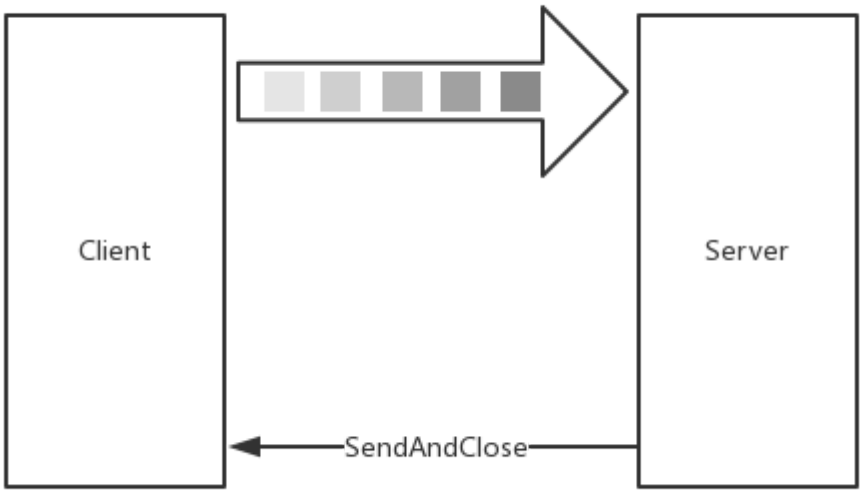
```
$ go run server.go
```

运行 stream_client/client.go：

```
$ go run client.go
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2018
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2019
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2020
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2021
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2022
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2023
2018/09/24 16:18:25 resp: pj.name: gRPC Stream Client: List, pt.value: 2024
```

二、Client-side streaming RPC：客户端流式 RPC

客户端流式 RPC，单向流，客户端通过流式发起**多次** RPC 请求给服务端，服务端发起**一次**响应给客户端，大致如图：



Server

```
func (s *StreamService) Record(stream pb.StreamService_RecordServer) error {
    for {
        r, err := stream.Recv()
        if err == io.EOF {
            return stream.SendAndClose(&pb.StreamResponse{Pt: &pb.StreamPoint{Name: "gRPC Stream Server: Record", Value: 1}})
        }
        if err != nil {
            return err
        }
```

```
        log.Printf("stream.Recv pt.name: %s, pt.value: %d", r.Pt.Name, r.Pt.Valu
e)
    }

    return nil
}
```

多了一个从未见过的方法 `stream.SendAndClose` ，它是做什么用的呢？

在这段程序中，我们对每一个 Recv 都进行了处理，当发现 `io.EOF` (流关闭) 后，需要将最终的响应结果发送给客户端，同时关闭正在另外一侧等待的 Recv

Client

```
func printRecord(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    stream, err := client.Record(context.Background())
    if err != nil {
        return err
    }

    for n := 0; n < 6; n++ {
        err := stream.Send(r)
        if err != nil {
            return err
        }
    }

    resp, err := stream.CloseAndRecv()
    if err != nil {
        return err
    }

    log.Printf("resp: pj.name: %s, pt.value: %d", resp.Pt.Name, resp.Pt.Value)

    return nil
}
```

`stream.CloseAndRecv` 和 `stream.SendAndClose` 是配套使用的流方法，相信聪明的你已经秒懂它的作用了

验证

重启 stream_server/server.go，再次运行 stream_client/client.go：

stream_client：

```
$ go run client.go
2018/09/24 16:23:03 resp: pj.name: gRPC Stream Server: Record, pt.value: 1
```

stream_server：

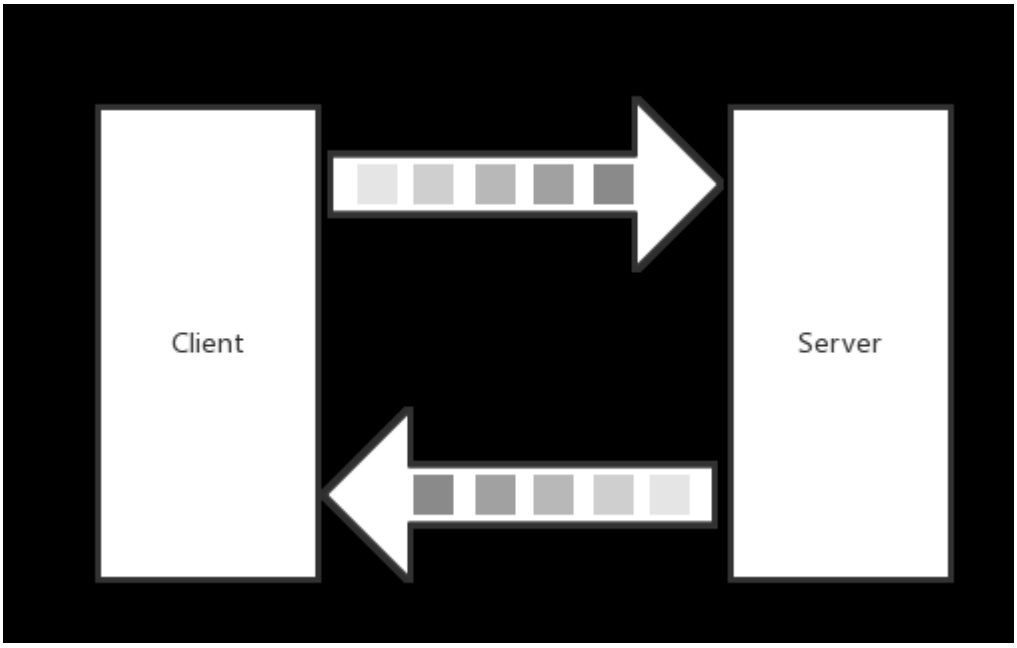
```
$ go run server.go
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
2018/09/24 16:23:03 stream.Recv pt.name: gRPC Stream Client: Record, pt.value: 2
018
```

三、Bidirectional streaming RPC：双向流式 RPC

双向流式 RPC，顾名思义是双向流。由客户端以流式的方式发起请求，服务端同样以流式的方式响应请求

首个请求一定是 Client 发起，但具体交互方式（谁先谁后、一次发多少、响应多少、什么时候关闭）根据程序编写的方式来确定（可以结合协程）

假设该双向流是**按顺序发送**的话，大致如图：



还是要强调，双向流变化很大，因程序编写的不同而不同。**双向流图示无法适用不同的场景**

Server

```
func (s *StreamService) Route(stream pb.StreamService_RouteServer) error {
    n := 0
    for {
        err := stream.Send(&pb.StreamResponse{
            Pt: &pb.StreamPoint{
                Name:  "gPRC Stream Client: Route",
                Value: int32(n),
            },
        })
        if err != nil {
            return err
        }

        r, err := stream.Recv()
        if err == io.EOF {
            return nil
        }
        if err != nil {
            return err
        }

        n++

        log.Printf("stream.Recv pt.name: %s, pt.value: %d", r.Pt.Name, r.Pt.Valu
e)
```

```
    }

    return nil
}
```

Client

```
func printRoute(client pb.StreamServiceClient, r *pb.StreamRequest) error {
    stream, err := client.Route(context.Background())
    if err != nil {
        return err
    }

    for n := 0; n <= 6; n++ {
        err = stream.Send(r)
        if err != nil {
            return err
        }

        resp, err := stream.Recv()
        if err == io.EOF {
            break
        }
        if err != nil {
            return err
        }

        log.Printf("resp: pj.name: %s, pt.value: %d", resp.Pt.Name, resp.Pt.Valu
e)
    }

    stream.CloseSend()

    return nil
}
```

验证

重启 stream_server/server.go，再次运行 stream_client/client.go：

stream_server

```
$ go run server.go
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
```

```
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
2018/09/24 16:29:43 stream.Recv pt.name: gRPC Stream Client: Route, pt.value: 20
18
```

stream_client

```
$ go run client.go
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 0
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 1
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 2
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 3
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 4
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 5
2018/09/24 16:29:43 resp: pj.name: gPRC Stream Client: Route, pt.value: 6
```

总结

在本文共介绍了三类流的交互方式，可以根据实际的业务场景去选择合适的方式。会事半功倍哦

参考

本系列示例代码

- go-grpc-example

TLS 证书认证

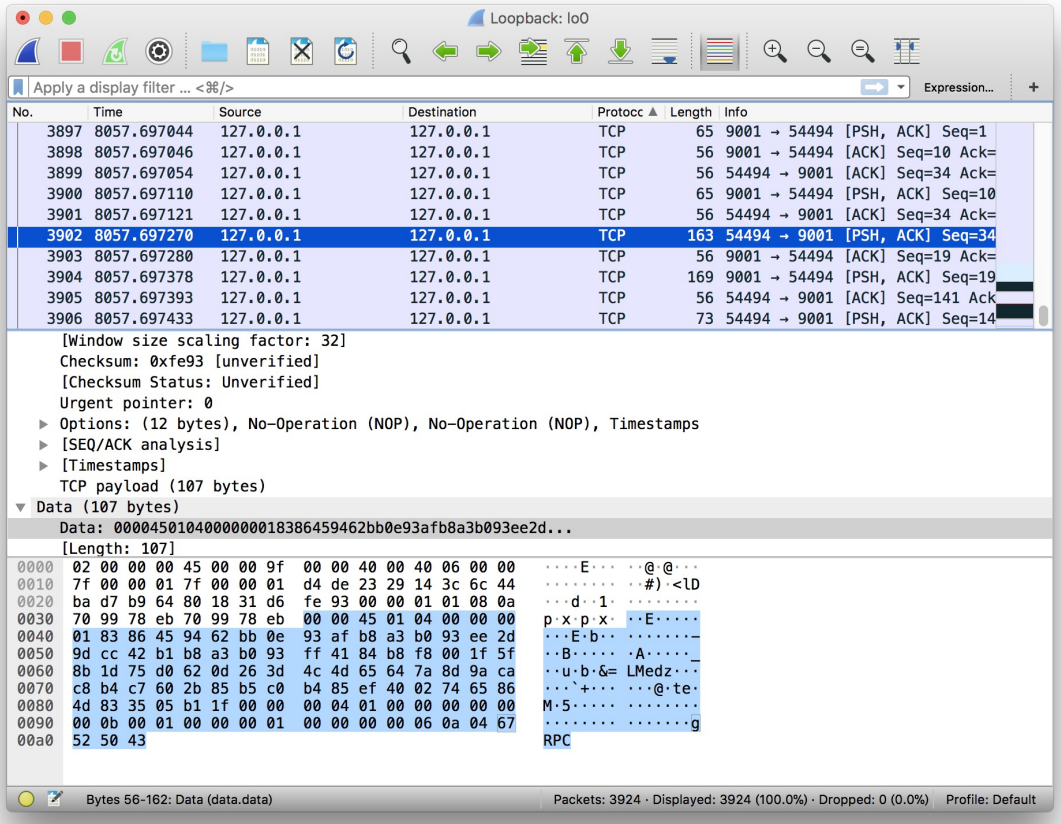
前言

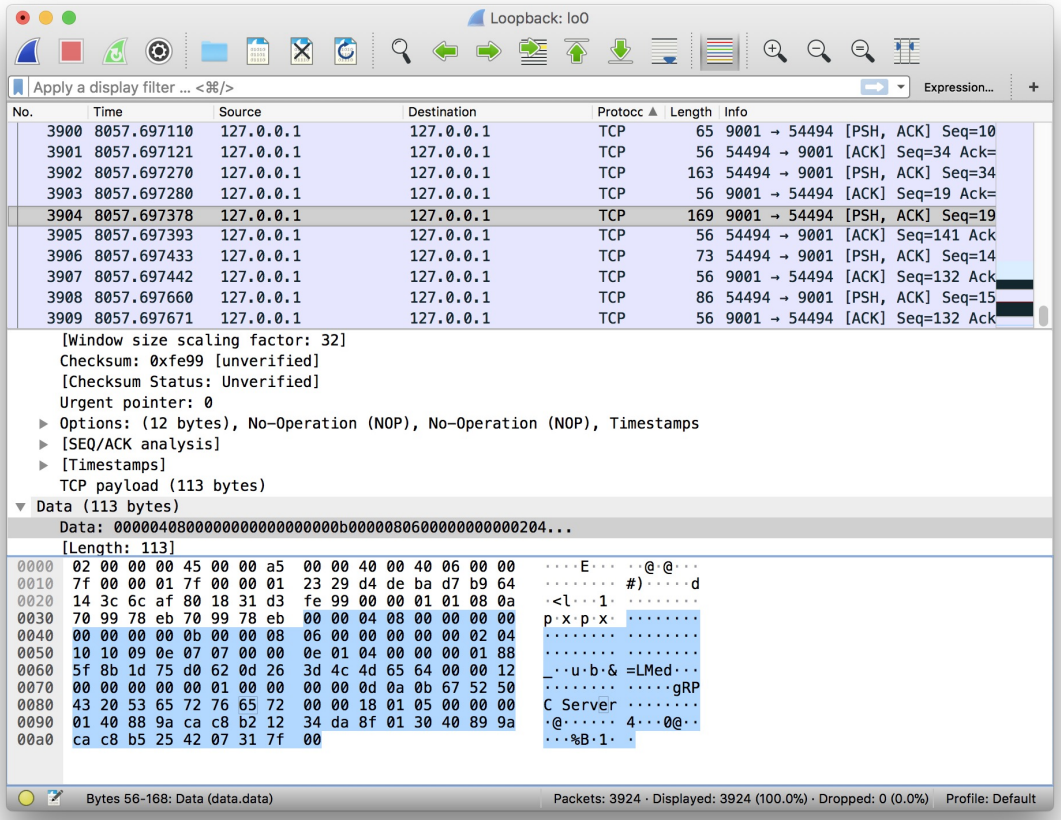
在前面的章节里，我们介绍了 gRPC 的四种 API 使用方式。是不是很简单呢

此时存在一个安全问题，先前的例子中 gRPC Client/Server 都是明文传输的，会不会有被窃听的风险呢？

从结论上来讲，是有的。在明文通讯的情况下，你的请求就是裸奔的，有可能被第三方恶意篡改或者伪造为“非法”的数据

抓个包





嗯，明文传输无误。这是有问题的，接下将改造我们的 gRPC，以便于解决这个问题

证书生成

私钥

```
openssl ecparam -genkey -name secp384r1 -out server.key
```

自签公钥

```
openssl req -new -x509 -sha256 -key server.key -out server.pem -days 3650
```

填写信息

```
Country Name (2 letter code) []:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
```

```
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:go-grpc-example
Email Address []:
```

生成完毕

生成证书结束后，将证书相关文件放到 conf/ 下，目录结构：

```
$ tree go-grpc-example
go-grpc-example
├── client
├── conf
│   ├── server.key
│   └── server.pem
├── proto
└── server
    ├── simple_server
    └── stream_server
```

由于本文偏向 gRPC，详解可参见 《制作证书》。后续番外可能会展开细节描述

为什么之前不需要证书

在 simple_server 中，为什么“啥事都没干”就能在不需要证书的情况下运行呢？

Server

```
grpc.NewServer()
```

在服务端显然没有传入任何 DialOptions

Client

```
conn, err := grpc.Dial(":"+PORT, grpc.WithInsecure())
```

在客户端留意到 `grpc.WithInsecure()` 方法

```
func WithInsecure() DialOption {
    return newFuncDialOption(func(o *dialOptions) {
        o.insecure = true
```

```
    })
}
```

在方法内可以看到 `WithInsecure` 返回一个 `DialOption` ，并且它最终会通过读取设置的值来禁用安全传输

那么它“最终”又是在哪里处理的呢，我们把视线移到 `grpc.Dial()` 方法内

```
func DialContext(ctx context.Context, target string, opts ...DialOption) (conn *ClientConn, err error) {
    ...

    for _, opt := range opts {
        opt.apply(&cc.dopts)
    }
    ...

    if !cc.dopts.insecure {
        if cc.dopts.copts.TransportCredentials == nil {
            return nil, errNoTransportSecurity
        }
    } else {
        if cc.dopts.copts.TransportCredentials != nil {
            return nil, errCredentialsConflict
        }
        for _, cd := range cc.dopts.copts.PerRPCCredentials {
            if cd.RequireTransportSecurity() {
                return nil, errTransportCredentialsMissing
            }
        }
    }
    ...

    creds := cc.dopts.copts.TransportCredentials
    if creds != nil && creds.Info().ServerName != "" {
        cc.authority = creds.Info().ServerName
    } else if cc.dopts.insecure && cc.dopts.authority != "" {
        cc.authority = cc.dopts.authority
    } else {
        // Use endpoint from "scheme://authority/endpoint" as the default
        // authority for ClientConn.
        cc.authority = cc.parsedTarget.Endpoint
    }
    ...
}
```

gRPC

接下来我们将正式开始编码，在 gRPC Client/Server 上实现 TLS 证书认证的支持

TLS Server

```
package main

import (
    "context"
    "log"
    "net"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

...

const PORT = "9001"

func main() {
    c, err := credentials.NewServerTLSFromFile("../../conf/server.pem", "../../conf/server.key")
    if err != nil {
        log.Fatalf("credentials.NewServerTLSFromFile err: %v", err)
    }

    server := grpc.NewServer(grpc.Creds(c))
    pb.RegisterSearchServiceServer(server, &SearchService{})

    lis, err := net.Listen("tcp", ":"+PORT)
    if err != nil {
        log.Fatalf("net.Listen err: %v", err)
    }

    server.Serve(lis)
}
```

- credentials.NewServerTLSFromFile：根据服务端输入的证书文件和密钥构造 TLS 凭证

```
func NewServerTLSFromFile(certFile, keyFile string) (TransportCredentials, erro
r) {
    cert, err := tls.LoadX509KeyPair(certFile, keyFile)
    if err != nil {
        return nil, err
    }
    return NewTLS(&tls.Config{Certificates: []tls.Certificate{cert}}), nil
}
```

- grpc.Creds()：返回一个 ServerOption，用于设置服务器连接的凭据。用于 `grpc.NewServer(opt ...ServerOption)` 为 gRPC Server 设置连接选项

```
func Creds(c credentials.TransportCredentials) ServerOption {
    return func(o *options) {
        o.creds = c
    }
}
```

经过以上两个简单步骤，gRPC Server 就建立起需证书认证的服务啦

TLS Client

```
package main

import (
    "context"
    "log"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const PORT = "9001"

func main() {
    c, err := credentials.NewClientTLSFromFile("../../conf/server.pem", "go-grpc
-example")
    if err != nil {
        log.Fatalf("credentials.NewClientTLSFromFile err: %v", err)
    }
```

```
    conn, err := grpc.Dial(":"+PORT, grpc.WithTransportCredentials(c))
    if err != nil {
        log.Fatalf("grpc.Dial err: %v", err)
    }
    defer conn.Close()

    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(context.Background(), &pb.SearchRequest{
        Request: "gRPC",
    })
    if err != nil {
        log.Fatalf("client.Search err: %v", err)
    }

    log.Printf("resp: %s", resp.GetResponse())
}
```

- credentials.NewClientTLSFromFile()：根据客户端输入的证书文件和密钥构造 TLS 凭证。serverNameOverride 为服务名称

```
func NewClientTLSFromFile(certFile, serverNameOverride string) (TransportCredentials, error) {
    b, err := ioutil.ReadFile(certFile)
    if err != nil {
        return nil, err
    }
    cp := x509.NewCertPool()
    if !cp.AppendCertsFromPEM(b) {
        return nil, fmt.Errorf("credentials: failed to append certificates")
    }
    return NewTLS(&tls.Config{ServerName: serverNameOverride, RootCAs: cp}), nil
}
```

- grpc.WithTransportCredentials()：返回一个配置连接的 DialOption 选项。用于 `grpc.Dial(target string, opts ...DialOption)` 设置连接选项

```
func WithTransportCredentials(creds credentials.TransportCredentials) DialOption {
    return newFuncDialOption(func(o *dialOptions) {
        o.copts.TransportCredentials = creds
    })
}
```

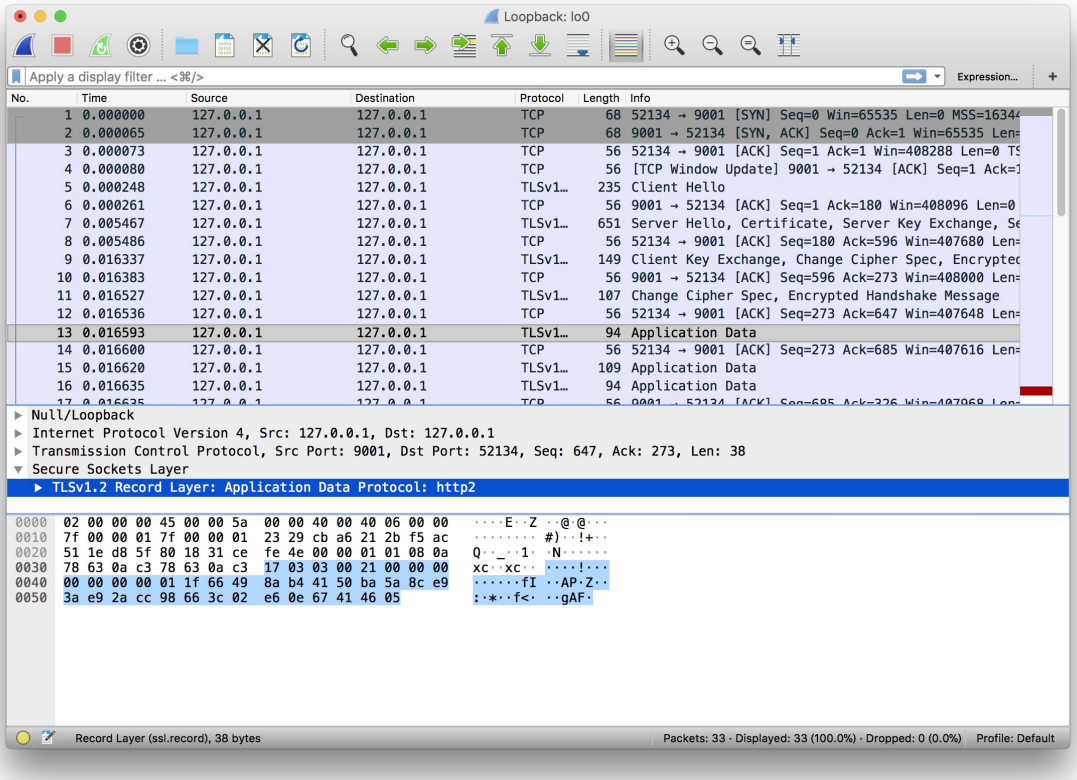
验证

请求

重新启动 server.go 和执行 client.go，得到响应结果

```
$ go run client.go
2018/09/30 20:00:21 resp: gRPC Server
```

抓个包



成功。

总结

在本章节我们实现了 gRPC TLS Client/Servert，你以为大功告成了吗？我不

问题

你仔细再看看，Client 是基于 Server 端的证书和服务名称来建立请求的。这样的话，你就需要将 Server 的证书通过各种手段给到 Client 端，否则是无法完成这项任务的

问题也就来了，你无法保证你的“各种手段”是安全的，毕竟现在的网络环境是很危险的，万一被...

我们将在下一章节解决这个问题，保证其可靠性

参考

本系列示例代码

- go-grpc-example

基于 CA 的 TLS 证书认证

前言

在上一章节中，我们提出了一个问题。就是如何保证证书的可靠性和有效性？你如何确定你 Server、Client 的证书是对的呢？

CA

为了保证证书的可靠性和有效性，在这里可引入 CA 颁发的根证书的概念。其遵守 X.509 标准

根证书

根证书（root certificate）是属于根证书颁发机构（CA）的公钥证书。我们可以通过验证 CA 的签名从而信任 CA，任何人都可以得到 CA 的证书（含公钥），用以验证它所签发的证书（客户端、服务端）

它包含的文件如下：

- 公钥
- 密钥

生成 Key

```
openssl genrsa -out ca.key 2048
```

生成密钥

```
openssl req -new -x509 -days 7200 -key ca.key -out ca.pem
```

填写信息

```
Country Name (2 letter code) []:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
```

```
Common Name (eg, fully qualified host name) []:go-grpc-example
Email Address []:
```

Server

生成 CSR

```
openssl req -new -key server.key -out server.csr
```

填写信息

```
Country Name (2 letter code) []:
State or Province Name (full name) []:
Locality Name (eg, city) []:
Organization Name (eg, company) []:
Organizational Unit Name (eg, section) []:
Common Name (eg, fully qualified host name) []:go-grpc-example
Email Address []:

Please enter the following 'extra' attributes
to be sent with your certificate request
A challenge password []:
```

CSR 是 Cerificate Signing Request 的英文缩写，为证书请求文件。主要作用是 CA 会利用 CSR 文件进行签名使得攻击者无法伪装或篡改原有证书

基于 CA 签发

```
openssl x509 -req -sha256 -CA ca.pem -CAkey ca.key -CAcreateserial -days 3650 -in server.csr -out server.pem
```

Client

生成 Key

```
openssl ecparam -genkey -name secp384r1 -out client.key
```

生成 CSR

```
openssl req -new -key client.key -out client.csr
```

基于 CA 签发

```
openssl x509 -req -sha256 -CA ca.pem -CAkey ca.key -CAcreateserial -days 3650 -in client.csr -out client.pem
```

整理目录

至此我们生成了一堆文件，请按照以下目录结构存放：

```
$ tree conf
conf
├── ca.key
├── ca.pem
├── ca.srl
├── client
│   ├── client.csr
│   ├── client.key
│   └── client.pem
└── server
    ├── server.csr
    ├── server.key
    └── server.pem
```

另外有一些文件是不应该出现在仓库内，应当保密或删除的。但为了真实演示所以保留着（敲黑板）

gRPC

接下来将正式开始针对 gRPC 进行编码，改造上一章节的代码。目标是基于 CA 进行 TLS 认证

Server

```
package main

import (
    "context"
    "log"
    "net"
    "crypto/tls"
    "crypto/x509"
    "io/ioutil"
```

```
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

...

const PORT = "9001"

func main() {
    cert, err := tls.LoadX509KeyPair("../../conf/server/server.pem", "../../con
f/server/server.key")
    if err != nil {
        log.Fatalf("tls.LoadX509KeyPair err: %v", err)
    }

    certPool := x509.NewCertPool()
    ca, err := ioutil.ReadFile("../../conf/ca.pem")
    if err != nil {
        log.Fatalf("ioutil.ReadFile err: %v", err)
    }

    if ok := certPool.AppendCertsFromPEM(ca); !ok {
        log.Fatalf("certPool.AppendCertsFromPEM err")
    }

    c := credentials.NewTLS(&tls.Config{
        Certificates: []tls.Certificate{cert},
        ClientAuth:   tls.RequireAndVerifyClientCert,
        ClientCAs:    certPool,
    })

    server := grpc.NewServer(grpc.Creds(c))
    pb.RegisterSearchServiceServer(server, &SearchService{})

    lis, err := net.Listen("tcp", ":"+PORT)
    if err != nil {
        log.Fatalf("net.Listen err: %v", err)
    }

    server.Serve(lis)
}
```

- tls.LoadX509KeyPair()：从证书相关文件中**读取**和**解析**信息，得到证书公钥、密钥对

```
func LoadX509KeyPair(certFile, keyFile string) (Certificate, error) {
    certPEMBlock, err := ioutil.ReadFile(certFile)
    if err != nil {
        return Certificate{}, err
    }
    keyPEMBlock, err := ioutil.ReadFile(keyFile)
    if err != nil {
        return Certificate{}, err
    }
    return X509KeyPair(certPEMBlock, keyPEMBlock)
}
```

- x509.NewCertPool()：创建一个新的、空的 CertPool
- certPool.AppendCertsFromPEM()：尝试解析所传入的 PEM 编码的证书。如果解析成功会将其加到 CertPool 中，便于后面的使用
- credentials.NewTLS：构建基于 TLS 的 TransportCredentials 选项
- tls.Config：Config 结构用于配置 TLS 客户端或服务器

在 Server，共使用了三个 Config 配置项：

（1）Certificates：设置证书链，允许包含一个或多个

（2）ClientAuth：要求必须校验客户端的证书。可以根据实际情况选用以下参数：

```
const (
    NoClientCert ClientAuthType = iota
    RequestClientCert
    RequireAnyClientCert
    VerifyClientCertIfGiven
    RequireAndVerifyClientCert
)
```

（3）ClientCAs：设置根证书的集合，校验方式使用 ClientAuth 中设定的模式

Client

```
package main

import (
    "context"
```

```
    "crypto/tls"
    "crypto/x509"
    "io/ioutil"
    "log"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const PORT = "9001"

func main() {
    cert, err := tls.LoadX509KeyPair("../../conf/client/client.pem", "../../con
f/client/client.key")
    if err != nil {
        log.Fatalf("tls.LoadX509KeyPair err: %v", err)
    }

    certPool := x509.NewCertPool()
    ca, err := ioutil.ReadFile("../../conf/ca.pem")
    if err != nil {
        log.Fatalf("ioutil.ReadFile err: %v", err)
    }

    if ok := certPool.AppendCertsFromPEM(ca); !ok {
        log.Fatalf("certPool.AppendCertsFromPEM err")
    }

    c := credentials.NewTLS(&tls.Config{
        Certificates: []tls.Certificate{cert},
        ServerName:   "go-grpc-example",
        RootCAs:      certPool,
    })

    conn, err := grpc.Dial(":"+PORT, grpc.WithTransportCredentials(c))
    if err != nil {
        log.Fatalf("grpc.Dial err: %v", err)
    }
    defer conn.Close()

    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(context.Background(), &pb.SearchRequest{
        Request: "gRPC",
    })
```

```
    if err != nil {
        log.Fatalf("client.Search err: %v", err)
    }

    log.Printf("resp: %s", resp.GetResponse())
}
```

在 Client 中绝大部分与 Server 一致，不同点的地方是，在 Client 请求 Server 端时，Client 端会使用根证书和 ServerName 去对 Server 端进行校验

简单流程大致如下：

1. Client 通过请求得到 Server 端的证书
2. 使用 CA 认证的根证书对 Server 端的证书进行可靠性、有效性等校验
3. 校验 ServerName 是否可用、有效

当然了，在设置了 `tls.RequireAndVerifyClientCert` 模式的情况下，Server 也会使用 CA 认证的根证书对 Client 端的证书进行可靠性、有效性等校验。也就是两边都会进行校验，极大的保证了安全性

验证

重新启动 server.go 和执行 client.go，查看响应结果是否正常

总结

在本章节，我们使用 CA 颁发的根证书对客户端、服务端的证书进行了签发。进一步的提高了两者的通讯安全

这回是真的大功告成了！

参考

本系列示例代码

- go-grpc-example

Unary and Stream interceptor

前言

我想在每个 RPC 方法的前或后做某些事情，怎么做？

本章节将要介绍的拦截器（interceptor），就能帮你在合适的地方实现这些功能。

有几种方法

在 gRPC 中，大类可分为两种 RPC 方法，与拦截器的对应关系是：

- 普通方法：一元拦截器（grpc.UnaryInterceptor）
- 流方法：流拦截器（grpc.StreamInterceptor）

看一看

grpc.UnaryInterceptor

```
func UnaryInterceptor(i UnaryServerInterceptor) ServerOption {
    return func(o *options) {
        if o.unaryInt != nil {
            panic("The unary server interceptor was already set and may not be r
eset.")
        }
        o.unaryInt = i
    }
}
```

函数原型：

```
type UnaryServerInterceptor func(ctx context.Context, req interface{}, info *Una
ryServerInfo, handler UnaryHandler) (resp interface{}, err error)
```

通过查看源码可得知，要完成一个拦截器需要实现 `UnaryServerInterceptor` 方法。形参如下：

- ctx context.Context：请求上下文

- req interface{}：RPC 方法的请求参数
- info *UnaryServerInfo：RPC 方法的所有信息
- handler UnaryHandler：RPC 方法本身

grpc.StreamInterceptor

```
func StreamInterceptor(i StreamServerInterceptor) ServerOption
```

函数原型：

```
type StreamServerInterceptor func(srv interface{}, ss ServerStream, info *Stream
ServerInfo, handler StreamHandler) error
```

StreamServerInterceptor 与 UnaryServerInterceptor 形参的意义是一样，不再赘述

如何实现多个拦截器

另外，可以发现 gRPC 本身居然只能设置一个拦截器，难道所有的逻辑都只能写在一起？

关于这一点，你可以放心。采用开源项目 go-grpc-middleware 就可以解决这个问题，本章也会使用它。

```
import "github.com/grpc-ecosystem/go-grpc-middleware"

myServer := grpc.NewServer(
    grpc.StreamInterceptor(grpc_middleware.ChainStreamServer(
        ...
    )),
    grpc.UnaryInterceptor(grpc_middleware.ChainUnaryServer(
        ...
    )),
)
```

gRPC

从本节开始编写 gRPC interceptor 的代码，我们会将实现以下拦截器：

- logging：RPC 方法的入参出参的日志输出
- recover：RPC 方法的异常保护和日志输出

实现 interceptor

logging

```
func LoggingInterceptor(ctx context.Context, req interface{}, info *grpc.UnarySe
rverInfo, handler grpc.UnaryHandler) (interface{}, error) {
    log.Printf("gRPC method: %s, %v", info.FullMethod, req)
    resp, err := handler(ctx, req)
    log.Printf("gRPC method: %s, %v", info.FullMethod, resp)
    return resp, err
}
```

recover

```
func RecoveryInterceptor(ctx context.Context, req interface{}, info *grpc.UnaryS
erverInfo, handler grpc.UnaryHandler) (resp interface{}, err error) {
    defer func() {
        if e := recover(); e != nil {
            debug.PrintStack()
            err = status.Errorf(codes.Internal, "Panic err: %v", e)
        }
    }()

    return handler(ctx, req)
}
```

Server

```
import (
    "context"
    "crypto/tls"
    "crypto/x509"
    "errors"
    "io/ioutil"
    "log"
    "net"
    "runtime/debug"

    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "google.golang.org/grpc/status"
    "google.golang.org/grpc/codes"
    "github.com/grpc-ecosystem/go-grpc-middleware"
```

```
    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

...

func main() {
    c, err := GetTLSCredentialsByCA()
    if err != nil {
        log.Fatalf("GetTLSCredentialsByCA err: %v", err)
    }

    opts := []grpc.ServerOption{
        grpc.Creds(c),
        grpc_middleware.WithUnaryServerChain(
            RecoveryInterceptor,
            LoggingInterceptor,
        ),
    }

    server := grpc.NewServer(opts...)
    pb.RegisterSearchServiceServer(server, &SearchService{})

    lis, err := net.Listen("tcp", ":"+PORT)
    if err != nil {
        log.Fatalf("net.Listen err: %v", err)
    }

    server.Serve(lis)
}
```

验证

logging

启动 simple_server/server.go，执行 simple_client/client.go 发起请求，得到结果：

```
$ go run server.go
2018/10/02 13:46:35 gRPC method: /proto.SearchService/Search, request:"gRPC"
2018/10/02 13:46:35 gRPC method: /proto.SearchService/Search, response:"gRPC Ser
ver"
```

recover

在 RPC 方法中人为地制造运行时错误，再重复启动 server/client.go，得到结果：

client

```
$ go run client.go
2018/10/02 13:19:03 client.Search err: rpc error: code = Internal desc = Panic e
rr: assignment to entry in nil map
exit status 1
```

server

```
$ go run server.go
goroutine 23 [running]:
runtime/debug.Stack(0xc420223588, 0x1033da9, 0xc420001980)
    /usr/local/Cellar/go/1.10.1/libexec/src/runtime/debug/stack.go:24 +0xa7
runtime/debug.PrintStack()
    /usr/local/Cellar/go/1.10.1/libexec/src/runtime/debug/stack.go:16 +0x22
main.RecoveryInterceptor.func1(0xc420223a10)
...
```

检查服务是否仍然运行，即可知道 Recovery 是否成功生效

总结

通过本章节，你可以学会最常见的拦截器使用方法。接下来其它“新”需求只要举一反三即可。

参考

本系列示例代码

- go-grpc-example

让你的服务同时提供 HTTP 接口

前言

- 接口需要提供给其他业务组访问，但是 RPC 协议不同无法内调，对方问能否走 HTTP 接口，怎么办？
- 微信（公众号、小程序）等第三方回调接口只支持 HTTP 接口，怎么办

我相信你在实际工作中都会遇到如上问题，在 gRPC 中都是有解决方案的，本章节将会进行介绍

为什么可以同时提供 HTTP 接口

关键一点，gRPC 的协议是基于 HTTP/2 的，因此应用程序能够在单个 TCP 端口上提供 HTTP/1.1 和 gRPC 接口服务（两种不同的流量）

怎么同时提供 HTTP 接口

检测协议

```
if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"), "application/grpc") {
    server.ServeHTTP(w, r)
} else {
    mux.ServeHTTP(w, r)
}
```

流程

1. 检测请求协议是否为 HTTP/2
2. 判断 Content-Type 是否为 application/grpc（gRPC 的默认标识位）
3. 根据协议的不同转发到不同的服务处理

gRPC

TLS

在前面的章节，为了便于展示因此没有简单封装

在本节需复用代码，重新封装了，可详见：go-grpc-example

目录结构

新建 simple_http_client、simple_http_server 目录，目录结构如下：

```
go-grpc-example
├─── client
|    ├─── simple_client
|    ├─── simple_http_client
|    └─── stream_client
├─── conf
├─── pkg
|    └─── gtls
├─── proto
├─── server
|    ├─── simple_http_server
|    ├─── simple_server
|    └─── stream_server
```

Server

在 simple_http_server 目录下新建 server.go，写入文件内容：

```
package main

import (
    "context"
    "log"
    "net/http"
    "strings"

    "github.com/EDDYCJY/go-grpc-example/pkg/gtls"
    pb "github.com/EDDYCJY/go-grpc-example/proto"

    "google.golang.org/grpc"
)

type SearchService struct{}

func (s *SearchService) Search(ctx context.Context, r *pb.SearchRequest) (*pb.Se
archResponse, error) {
```

```
    return &pb.SearchResponse{Response: r.GetRequest() + " HTTP Server"}, nil
}

const PORT = "9003"

func main() {
    certFile := "../../conf/server/server.pem"
    keyFile := "../../conf/server/server.key"
    tlsServer := gtls.Server{
        CertFile: certFile,
        KeyFile:  keyFile,
    }

    c, err := tlsServer.GetTLSCredentials()
    if err != nil {
        log.Fatalf("tlsServer.GetTLSCredentials err: %v", err)
    }

    mux := GetHTTPServeMux()

    server := grpc.NewServer(grpc.Creds(c))
    pb.RegisterSearchServiceServer(server, &SearchService{})

    http.ListenAndServeTLS(":"+PORT,
        certFile,
        keyFile,
        http.HandlerFunc(func(w http.ResponseWriter, r *http.Request) {
            if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"
), "application/grpc") {
                server.ServeHTTP(w, r)
            } else {
                mux.ServeHTTP(w, r)
            }

            return
        }),
    )
}

func GetHTTPServeMux() *http.ServeMux {
    mux := http.NewServeMux()
    mux.HandleFunc("/", func(w http.ResponseWriter, r *http.Request) {
        w.Write([]byte("eddycjy: go-grpc-example"))
    })
```

```
    return mux
}
```

- http.NewServeMux：创建一个新的 ServeMux，ServeMux 本质上是一个路由表。它默认实现了 ServeHTTP，因此返回 Handler 后可直接通过 HandleFunc 注册 pattern 和处理逻辑的方法
- http.ListenAndServeTLS：可简单的理解为提供监听 HTTPS 服务的方法，重点的协议判断转发，也在这里面

其实，你理解后就会觉得很简单，核心步骤：判断 -> 转发 -> 响应。我们改变了前两步的默认逻辑，仅此而已

Client

在 simple_http_server 目录下新建 client.go，写入文件内容：

```
package main

import (
    "context"
    "log"

    "google.golang.org/grpc"

    "github.com/EDDYCJY/go-grpc-example/pkg/gtls"
    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const PORT = "9003"

func main() {
    tlsClient := gtls.Client{
        ServerName: "go-grpc-example",
        CertFile:   "../../conf/server/server.pem",
    }
    c, err := tlsClient.GetTLSCredentials()
    if err != nil {
        log.Fatalf("tlsClient.GetTLSCredentials err: %v", err)
    }

    conn, err := grpc.Dial(":"+PORT, grpc.WithTransportCredentials(c))
    if err != nil {
        log.Fatalf("grpc.Dial err: %v", err)
```

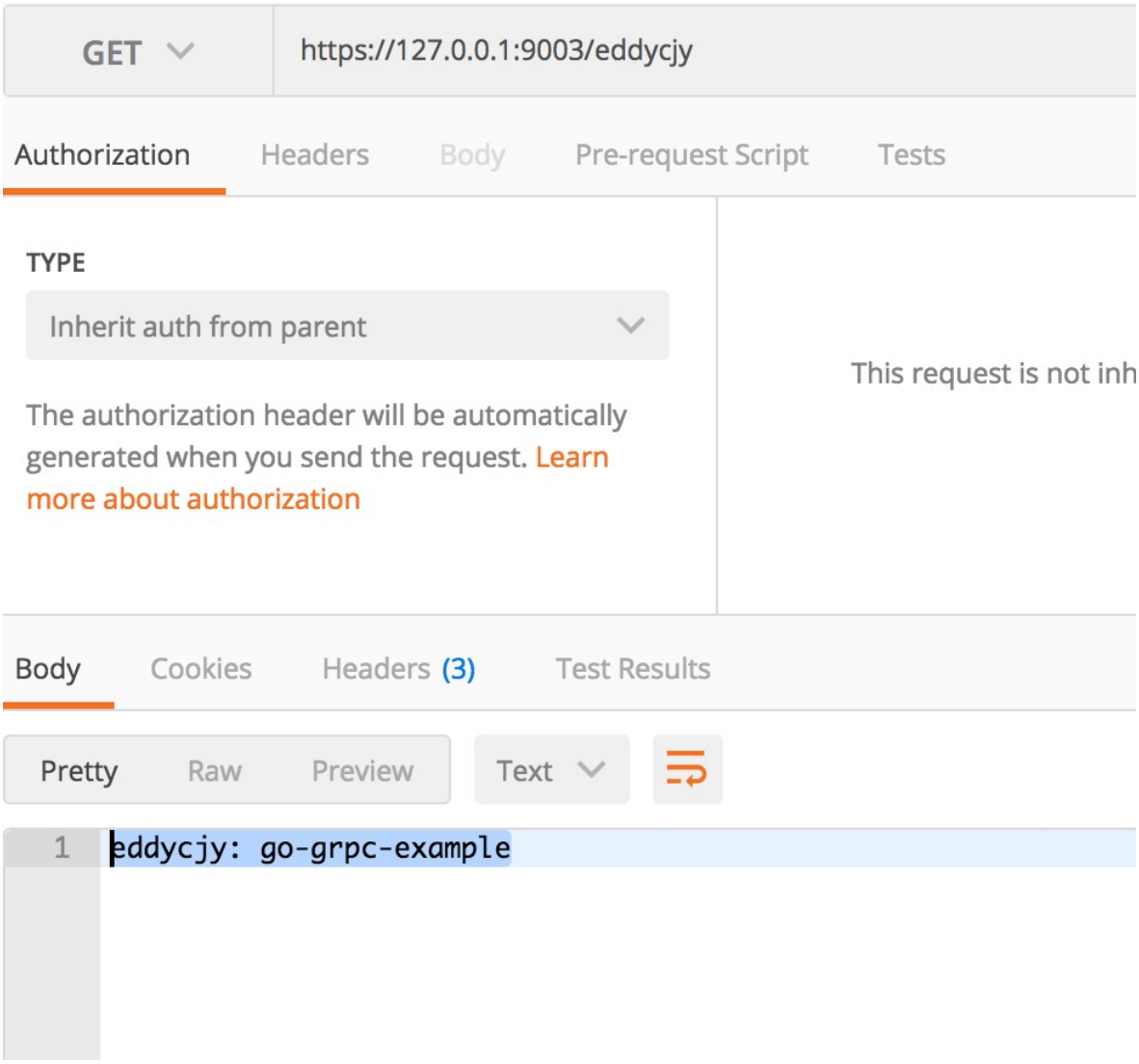
```
    }
    defer conn.Close()

    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(context.Background(), &pb.SearchRequest{
        Request: "gRPC",
    })
    if err != nil {
        log.Fatalf("client.Search err: %v", err)
    }

    log.Printf("resp: %s", resp.GetResponse())
}
```

验证

gRPC Client

```
$ go run client.go
2018/10/04 14:56:56 resp: gRPC HTTP Server
```

HTTP/1.1 访问



总结

通过本章节，表面上完成了同端口提供双服务的功能，但实际上，应该是加深了 HTTP/2 的理解和使用，这才是本质

拓展

如果你有一个需求，是要**同时提供** RPC 和 RESTful JSON API 两种接口的，不要犹豫，点进去：gRPC + gRPC Gateway 实践

问题

你以为这个方案就万能了吗，不。Envoy Proxy 的支持就不完美，无法同时监听一个端口的两种流量

参考

本系列示例代码

- go-grpc-example

对 RPC 方法做自定义认证

前言

在前面的章节中，我们介绍了两种（证书算一种）可全局认证的方法：

1. TLS 证书认证
2. 基于 CA 的 TLS 证书认证
3. Unary and Stream interceptor

而在实际需求中，常常会对某些模块的 RPC 方法做特殊认证或校验。今天将会讲解、实现这块的功能点

课前知识

```
type PerRPCCredentials interface {
    GetRequestMetadata(ctx context.Context, uri ...string) (map[string]string, error)
    RequireTransportSecurity() bool
}
```

在 gRPC 中默认定义了 PerRPCCredentials，它就是本章节的主角，是 gRPC 默认提供用于自定义认证的接口，它的作用是将所需的安全认证信息添加到每个 RPC 方法的上下文中。其包含 2 个方法：

- GetRequestMetadata：获取当前请求认证所需的元数据（metadata）
- RequireTransportSecurity：是否需要基于 TLS 认证进行安全传输

目录结构

新建 simple_token_server/server.go 和 simple_token_client/client.go，目录结构如下：

```
go-grpc-example
├─── client
|    ├─── simple_client
|    ├─── simple_http_client
|    ├─── simple_token_client
|    └─── stream_client
├─── conf
├─── pkg
```

```
├──── proto
├──── server
|     ├──── simple_http_server
|     ├──── simple_server
|     ├──── simple_token_server
|     └──── stream_server
└──── vendor
```

gRPC

Client

```
package main

import (
    "context"
    "log"

    "google.golang.org/grpc"

    "github.com/EDDYCJY/go-grpc-example/pkg/gtls"
    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

const PORT = "9004"

type Auth struct {
    AppKey    string
    AppSecret string
}

func (a *Auth) GetRequestMetadata(ctx context.Context, uri ...string) (map[string]string, error) {
    return map[string]string{"app_key": a.AppKey, "app_secret": a.AppSecret}, nil
}

func (a *Auth) RequireTransportSecurity() bool {
    return true
}

func main() {
    tlsClient := gtls.Client{
        ServerName: "go-grpc-example",
```

```
        CertFile:   "../../conf/server/server.pem",
    }
    c, err := tlsClient.GetTLSCredentials()
    if err != nil {
        log.Fatalf("tlsClient.GetTLSCredentials err: %v", err)
    }

    auth := Auth{
        AppKey:    "eddycjy",
        AppSecret: "20181005",
    }
    conn, err := grpc.Dial(":"+PORT, grpc.WithTransportCredentials(c), grpc.With
PerRPCCredentials(&auth))
    ...
}
```

在 Client 端，重点实现 `type PerRPCCredentials interface` 所需的方法，关注两点即可：

- struct Auth：GetRequestMetadata、RequireTransportSecurity
- grpc.WithPerRPCCredentials

Server

```
package main

import (
    "context"
    "log"
    "net"

    "google.golang.org/grpc"
    "google.golang.org/grpc/codes"
    "google.golang.org/grpc/metadata"
    "google.golang.org/grpc/status"

    "github.com/EDDYCJY/go-grpc-example/pkg/gtls"
    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

type SearchService struct {
    auth *Auth
}
```

```
func (s *SearchService) Search(ctx context.Context, r *pb.SearchRequest) (*pb.Se
archResponse, error) {
    if err := s.auth.Check(ctx); err != nil {
        return nil, err
    }
    return &pb.SearchResponse{Response: r.GetRequest() + " Token Server"}, nil
}

const PORT = "9004"

func main() {
    ...
}

type Auth struct {
    appKey    string
    appSecret string
}

func (a *Auth) Check(ctx context.Context) error {
    md, ok := metadata.FromIncomingContext(ctx)
    if !ok {
        return status.Errorf(codes.Unauthenticated, "自定义认证 Token 失败")
    }

    var (
        appKey    string
        appSecret string
    )
    if value, ok := md["app_key"]; ok {
        appKey = value[0]
    }
    if value, ok := md["app_secret"]; ok {
        appSecret = value[0]
    }

    if appKey != a.GetAppKey() || appSecret != a.GetAppSecret() {
        return status.Errorf(codes.Unauthenticated, "自定义认证 Token 无效")
    }

    return nil
}

func (a *Auth) GetAppKey() string {
    return "eddycjy"
}
```

```
func (a *Auth) GetAppSecret() string {
    return "20181005"
}
```

在 Server 端就更简单了，实际就是调用 `metadata.FromIncomingContext` 从上下文中获取 metadata，再在不同的 RPC 方法中进行认证检查

验证

重新启动 server.go 和 client.go，得到以下结果：

```
$ go run client.go
2018/10/05 20:59:58 resp: gRPC Token Server
```

修改 client.go 的值，制造两者不一致，得到无效结果：

```
$ go run client.go
2018/10/05 21:00:05 client.Search err: rpc error: code = Unauthenticated desc =
 invalid token
exit status 1
```

一个个加太麻烦

我相信你肯定会问一个个加，也太麻烦了吧？有这个想法的你，应当把 `type PerRPCCredentials interface` 做成一个拦截器（interceptor）

总结

本章节比较简单，主要是针对 RPC 方法的自定义认证进行了介绍，如果是想做全局的，建议是举一反三从拦截器下手哦。

参考

本系列示例代码

- go-grpc-example

gRPC Deadlines

前言

在前面的章节中，已经介绍了 gRPC 的基本用法。那你想想，让它这么裸跑真的没问题吗？

那么，肯定是有问题了。今天将介绍 gRPC Deadlines 的用法，这一个必备技巧。内容也比较简单

Deadlines

Deadlines 意指截止时间，在 gRPC 中强调 TL;DR（Too long, Don’t read）并建议**始终设定截止日期**，为什么呢？

为什么要设置

当未设置 Deadlines 时，将采用默认的 DEADLINE_EXCEEDED（这个时间非常大）

如果产生了阻塞等待，就会造成大量正在进行的请求都会被保留，并且所有请求都有可能达到最大超时

这会使服务面临资源耗尽的风险，例如内存，这会增加服务的延迟，或者在最坏的情况下可能导致整个进程崩溃

gRPC

Client

```
func main() {
    ...
    ctx, cancel := context.WithDeadline(context.Background(), time.Now().Add(tim
e.Duration(5 * time.Second)))
    defer cancel()

    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(ctx, &pb.SearchRequest{
        Request: "gRPC",
    })
    if err != nil {
        statusErr, ok := status.FromError(err)
```

```
        if ok {
            if statusErr.Code() == codes.DeadlineExceeded {
                log.Fatalln("client.Search err: deadline")
            }
        }

        log.Fatalf("client.Search err: %v", err)
    }

    log.Printf("resp: %s", resp.GetResponse())
}
```

- context.WithDeadline：会返回最终上下文截止时间。第一个形参为父上下文，第二个形参为调整的截止时间。若父级时间早于子级时间，则以父级时间为准，否则以子级时间为最终截止时间

```
func WithDeadline(parent Context, d time.Time) (Context, CancelFunc) {
    if cur, ok := parent.Deadline(); ok && cur.Before(d) {
        // The current deadline is already sooner than the new one.
        return WithCancel(parent)
    }
    c := &timerCtx{
        cancelCtx: newCancelCtx(parent),
        deadline:  d,
    }
    propagateCancel(parent, c)
    dur := time.Until(d)
    if dur <= 0 {
        c.cancel(true, DeadlineExceeded) // deadline has already passed
        return c, func() { c.cancel(true, Canceled) }
    }
    c.mu.Lock()
    defer c.mu.Unlock()
    if c.err == nil {
        c.timer = time.AfterFunc(dur, func() {
            c.cancel(true, DeadlineExceeded)
        })
    }
    return c, func() { c.cancel(true, Canceled) }
}
```

- context.WithTimeout：很常见的另外一个方法，是便捷操作。实际上是对于 WithDeadline 的封装

```
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc) {
    return WithDeadline(parent, time.Now().Add(timeout))
}
```

- status.FromError：返回 GRPCStatus 的具体错误码，若为非法，则直接返回 `codes.Unknown`

Server

```
type SearchService struct{}

func (s *SearchService) Search(ctx context.Context, r *pb.SearchRequest) (*pb.SearchResponse, error) {
    for i := 0; i < 5; i++  {
        if ctx.Err() == context.Canceled {
            return nil, status.Errorf(codes.Canceled, "SearchService.Search canceled")
        }

        time.Sleep(1 * time.Second)
    }

    return &pb.SearchResponse{Response: r.GetRequest() + " Server"}, nil
}

func main() {
    ...
}
```

而在 Server 端，由于 Client 已经设置了截止时间。Server 势必要去检测它

否则如果 Client 已经结束掉了，Server 还傻傻的在那执行，这对资源是一种极大的浪费

因此在这里需要用 `ctx.Err() == context.Canceled` 进行判断，为了模拟场景我们加了循环和睡眠

验证

重新启动 server.go 和 client.go，得到结果：

```
$ go run client.go
2018/10/06 17:45:55 client.Search err: deadline
exit status 1
```

总结

本章节比较简单，你需要知道以下知识点：

- 怎么设置 Deadlines
- 为什么要设置 Deadlines

你要清楚地明白到，gRPC Deadlines 是很重要的，否则这小小的功能点就会要了你生产的命
□

参考

本系列示例代码

- go-grpc-example

资料

- gRPC and Deadlines

分布式链路追踪 gRPC + Opentracing + Zipkin

在实际应用中，你做了那么多 Server 端，写了 N 个 RPC 方法。想看看方法的指标，却无处下手？

本文将通过 gRPC + Opentracing + Zipkin 搭建一个**分布式链路追踪系统**来实现查看整个系统的链路、性能等指标。

Opentracing

是什么

OpenTracing 通过提供平台无关、厂商无关的API，使得开发人员能够方便的添加（或更换）追踪系统的实现

不过 OpenTracing 并不是标准。因为 CNCF 不是官方标准机构，但是它的目标是致力为分布式追踪创建更标准的 API 和工具

名词解释

Trace

一个 trace 代表了一个事务或者流程在（分布式）系统中的执行过程

Span

一个 span 代表在分布式系统中完成的单个工作单元。也包含其他 span 的 “引用”，这允许将多个 spans 组合成一个完整的 Trace

每个 span 根据 OpenTracing 规范封装以下内容：

- 操作名称
- 开始时间和结束时间
- key:value span Tags
- key:value span Logs
- SpanContext

Tags

Span tags（跨度标签）可以理解为用户自定义的 Span 注释。便于查询、过滤和理解跟踪数据

Logs

Span logs（跨度日志）可以记录 Span 内特定时间或事件的日志信息。主要用于捕获特定 Span 的日志信息以及应用程序本身的其他调试或信息输出

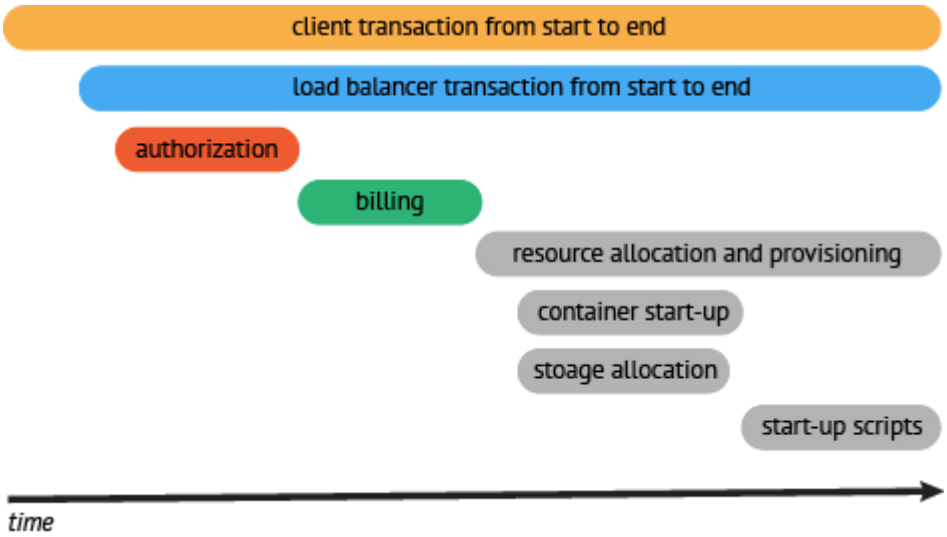
SpanContext

SpanContext 代表跨越进程边界，传递到子级 Span 的状态。常在追踪示意图中创建上下文时使用

Baggage Items

Baggage Items 可以理解为 trace 全局运行中额外传输的数据集合

一个案例

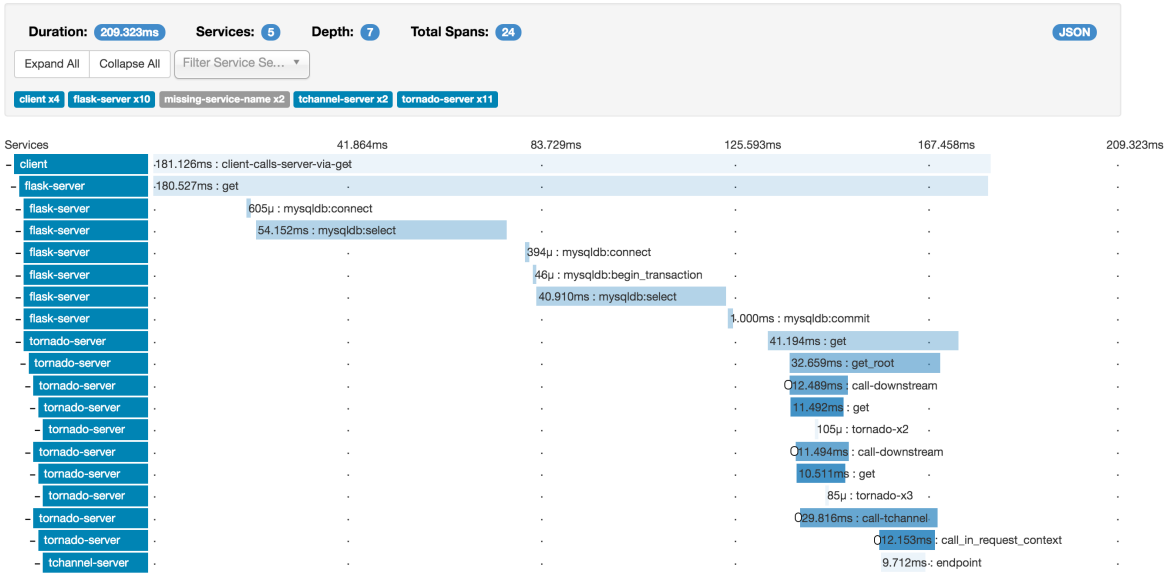


图中可以看到以下内容：

- 执行时间的上下文
- 服务间的层次关系
- 服务间串行或并行调用链

结合以上信息，在实际场景中我们可以通过整个系统的调用链的上下文、性能等指标信息，一下子就能够发现系统的痛点在哪儿

Zipkin



是什么

Zipkin 是分布式追踪系统。它的作用是收集解决微服务架构中的延迟问题所需的时序数据。它管理这些数据的收集和查找

Zipkin 的设计基于 Google Dapper 论文。

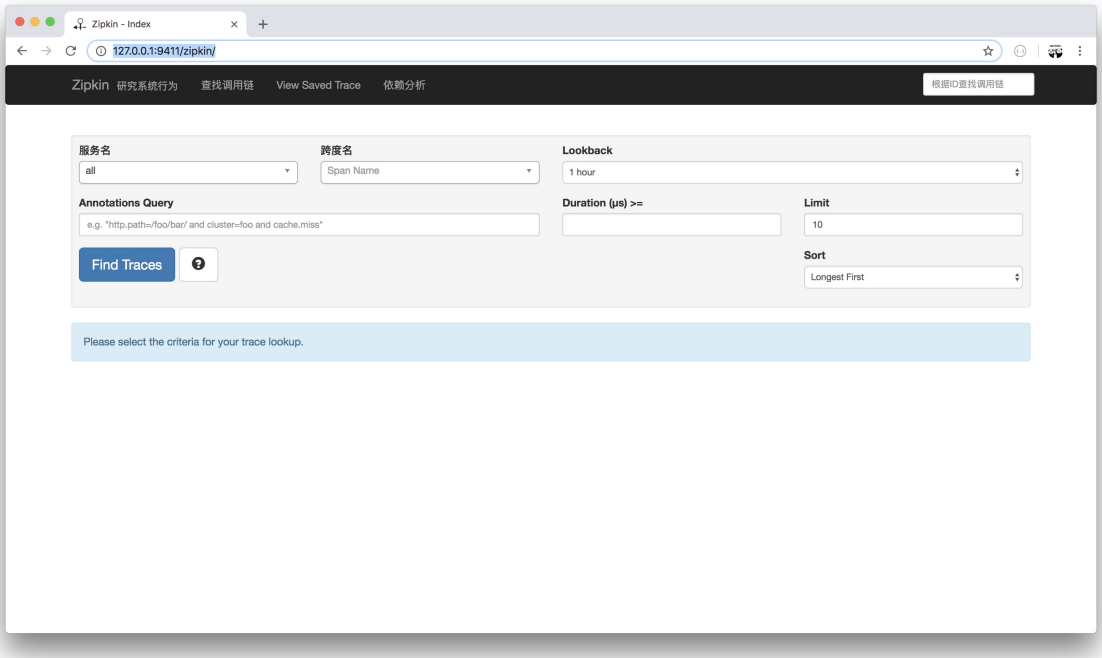
运行

```
docker run -d -p 9411:9411 openzipkin/zipkin
```

其他方法安装参见：https://github.com/openzipkin/zipkin

验证

访问 http://127.0.0.1:9411/zipkin/ 检查 Zipkin 是否运行正常



gRPC + Opentracing + Zipkin

在前面的小节中，我们做了以下准备工作：

- 了解 Opentracing 是什么
- 搭建 Zipkin 提供分布式追踪系统的功能

接下来实现 gRPC 通过 Opentracing 标准 API 对接 Zipkin，再通过 Zipkin 去查看数据

目录结构

新建 simple_zipkin_client、simple_zipkin_server 目录，目录结构如下：

```
go-grpc-example
├──── LICENSE
├──── README.md
├──── client
|     ├──── ...
|     ├──── simple_zipkin_client
├──── conf
├──── pkg
├──── proto
├──── server
|     ├──── ...
```

```
|    ├──── simple_zipkin_server
└──── vendor
```

安装

```
$ go get -u github.com/openzipkin/zipkin-go-opentracing
$ go get -u github.com/grpc-ecosystem/grpc-opentracing/go/otgrpc
```

gRPC

Server

```
package main

import (
    "context"
    "log"
    "net"

    "github.com/grpc-ecosystem/go-grpc-middleware"
    "github.com/grpc-ecosystem/grpc-opentracing/go/otgrpc"
    zipkin "github.com/openzipkin/zipkin-go-opentracing"
    "google.golang.org/grpc"

    "github.com/EDDYCJY/go-grpc-example/pkg/gtls"
    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

type SearchService struct{}

func (s *SearchService) Search(ctx context.Context, r *pb.SearchRequest) (*pb.SearchResponse, error) {
    return &pb.SearchResponse{Response: r.GetRequest() + " Server"}, nil
}

const (
    PORT = "9005"

    SERVICE_NAME               = "simple_zipkin_server"
    ZIPKIN_HTTP_ENDPOINT       = "http://127.0.0.1:9411/api/v1/spans"
    ZIPKIN_RECORDER_HOST_PORT = "127.0.0.1:9000"
)
```

```
func main() {
    collector, err := zipkin.NewHTTPCollector(ZIPKIN_HTTP_ENDPOINT)
    if err != nil {
        log.Fatalf("zipkin.NewHTTPCollector err: %v", err)
    }

    recorder := zipkin.NewRecorder(collector, true, ZIPKIN_RECORDER_HOST_PORT, S
ERVICE_NAME)

    tracer, err := zipkin.NewTracer(
        recorder, zipkin.ClientServerSameSpan(false),
    )
    if err != nil {
        log.Fatalf("zipkin.NewTracer err: %v", err)
    }

    tlsServer := gtls.Server{
        CaFile:   "../../conf/ca.pem",
        CertFile: "../../conf/server/server.pem",
        KeyFile:  "../../conf/server/server.key",
    }
    c, err := tlsServer.GetCredentialsByCA()
    if err != nil {
        log.Fatalf("GetTLSCredentialsByCA err: %v", err)
    }

    opts := []grpc.ServerOption{
        grpc.Creds(c),
        grpc_middleware.WithUnaryServerChain(
            otgrpc.OpenTracingServerInterceptor(tracer, otgrpc.LogPayloads()),
        ),
    }
    ...
}
```

- zipkin.NewHTTPCollector：创建一个 Zipkin HTTP 后端收集器
- zipkin.NewRecorder：创建一个基于 Zipkin 收集器的记录器
- zipkin.NewTracer：创建一个 OpenTracing 跟踪器（兼容 Zipkin Tracer）
- otgrpc.OpenTracingClientInterceptor：返回 grpc.UnaryServerInterceptor，不同点在于该拦截器会在 gRPC Metadata 中查找 OpenTracing SpanContext。如果找到则为该服务的 Span Context 的子节点

- otgrpc.LogPayloads：设置并返回 Option。作用是让 OpenTracing 在双向方向上记录应用程序的有效载荷（payload）

总的来讲，就是初始化 Zipkin，其又包含收集器、记录器、跟踪器。再利用拦截器在 Server 端实现 SpanContext、Payload 的双向读取和管理

Client

```
func main() {
    // the same as zipkin server
    // ...
    conn, err := grpc.Dial(":"+PORT, grpc.WithTransportCredentials(c),
        grpc.WithUnaryInterceptor(
            otgrpc.OpenTracingClientInterceptor(tracer, otgrpc.LogPayloads()),
        ))
    ...
}
```

- otgrpc.OpenTracingClientInterceptor：返回 grpc.UnaryClientInterceptor。该拦截器的核心功能在于：

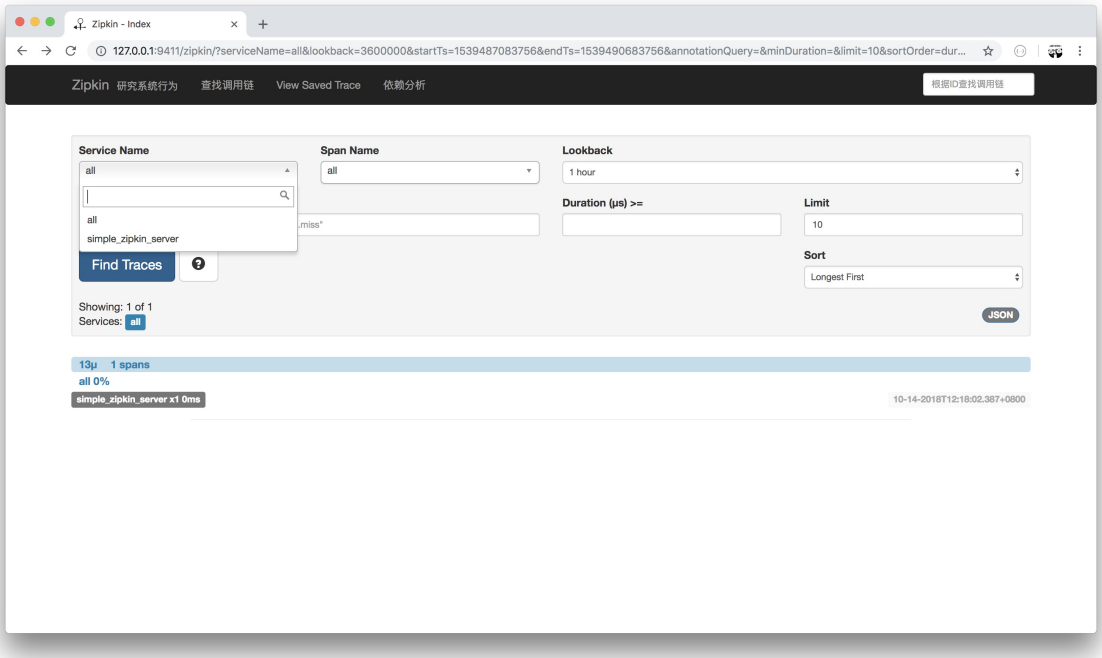
（1）OpenTracing SpanContext 注入 gRPC Metadata

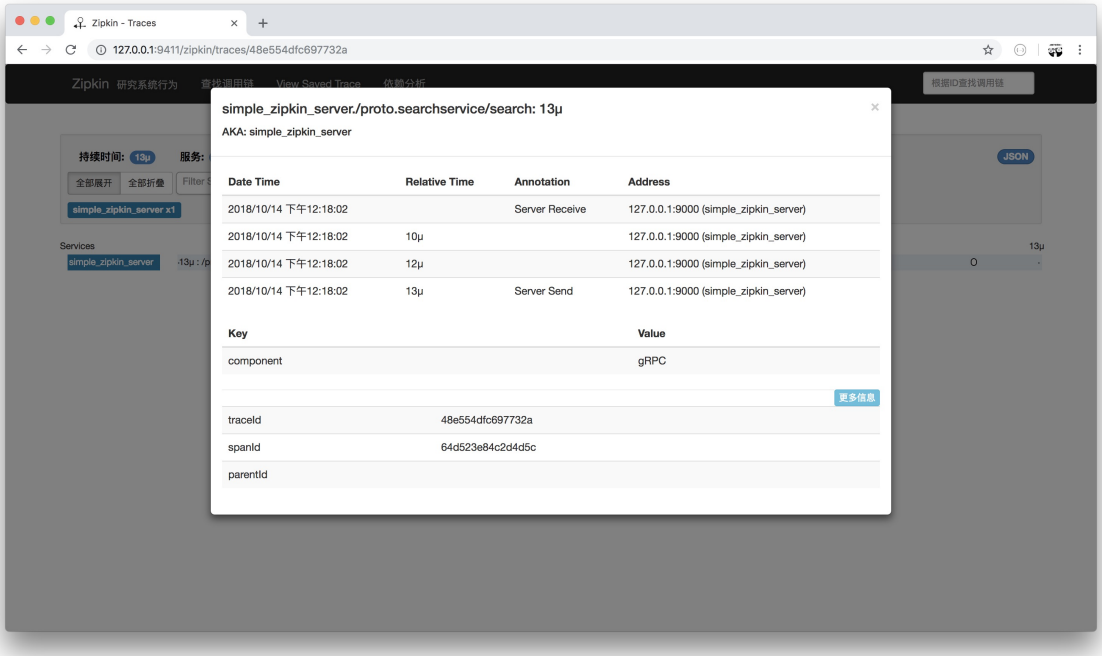
（2）查看 context.Context 中的上下文关系，若存在父级 Span 则创建一个 ChildOf 引用，得到一个子 Span

其他方面，与 Server 端是一致的，先初始化 Zipkin，再增加 Client 端特需的拦截器。就可以完成基础工作啦

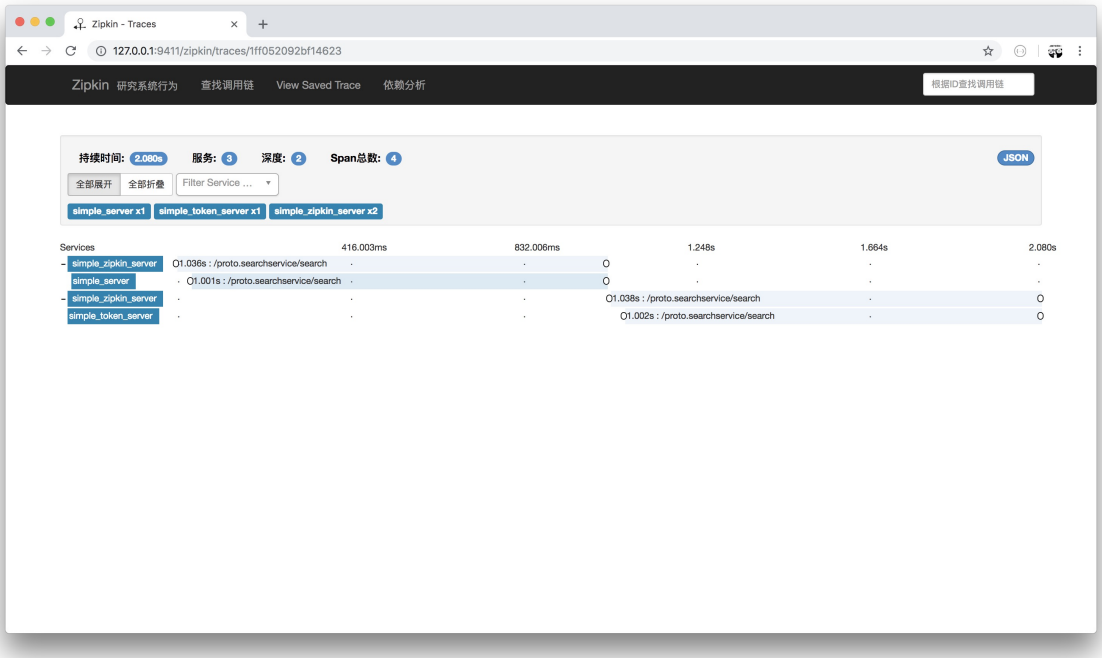
验证

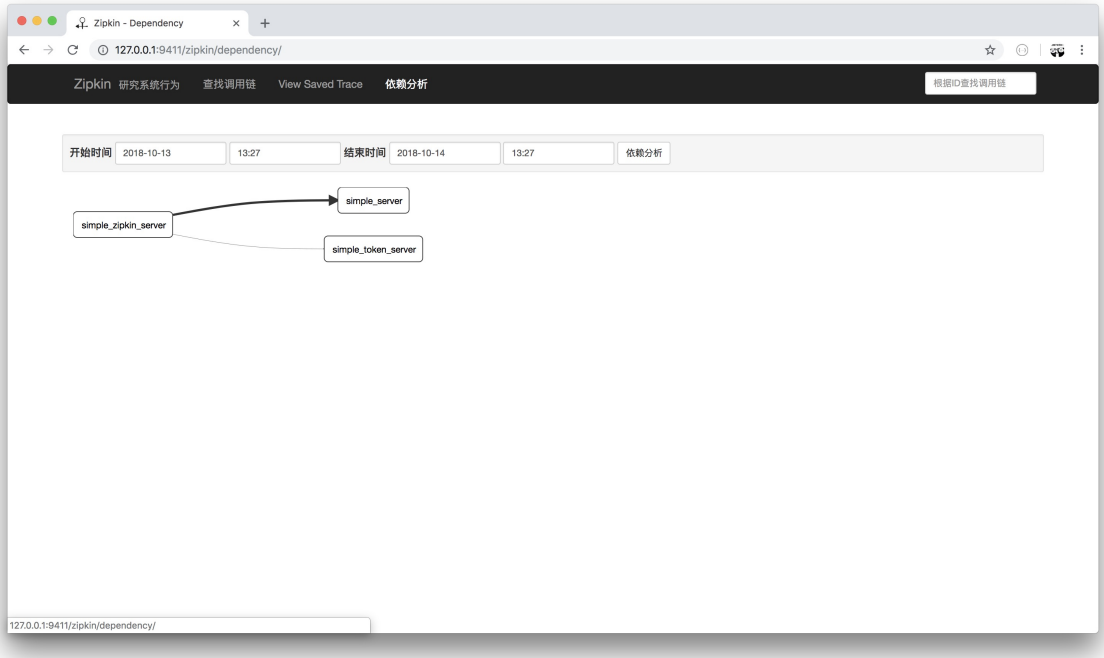
启动 Server.go，执行 Client.go。查看 http://127.0.0.1:9411/zipkin/ 的示意图：


Zipkin - Index
127.0.0.1:9411/zipkin/?serviceName=all&lookback=3600000&startTs=1539487083756&endTs=1539490683756&annotationQuery=&minDuration=&limit=10&sortOrder=dur...
Zipkin 研究系统行为
查找调用链
View Saved Trace
依赖分析
根据ID查找调用链
Service Name
all
simple_zipkin_server
Span Name
all
Lookback
1 hour
Duration (μs) >=
Limit
10
Sort
Longest First
Find Traces
Showing: 1 of 1
Services: all
JSON
13μ 1 spans
all 0%
simple_zipkin_server x1 0ms
10-14-2018T12:18:02.387+0800


Zipkin - Traces
127.0.0.1:9411/zipkin/traces/48e554dfc697732a
simple_zipkin_server./proto.searchservice/search: 13μ
AKA: simple_zipkin_server
Date Time
Relative Time
Annotation
Address
2018/10/14 下午12:18:02
Server Receive
127.0.0.1:9000 (simple_zipkin_server)
2018/10/14 下午12:18:02
10μ
2018/10/14 下午12:18:02
12μ
2018/10/14 下午12:18:02
13μ
Server Send
Key
Value
component
gRPC
更多信息
traceId
48e554dfc697732a
spanId
64d523e84c2d4d5c
parentId

复杂点





来，自己实践一下

总结

在多服务下的架构下，串行、并行、服务套服务是一个非常常见的情况，用常规的方案往往很难发现问题在哪里（成本太大）。而这种情况就是**分布式追踪系统**大展拳脚的机会了

希望你通过本章节的介绍和学习，能够了解其概念和搭建且应用一个追踪系统。

参考

本系列示例代码

- go-grpc-example

资料

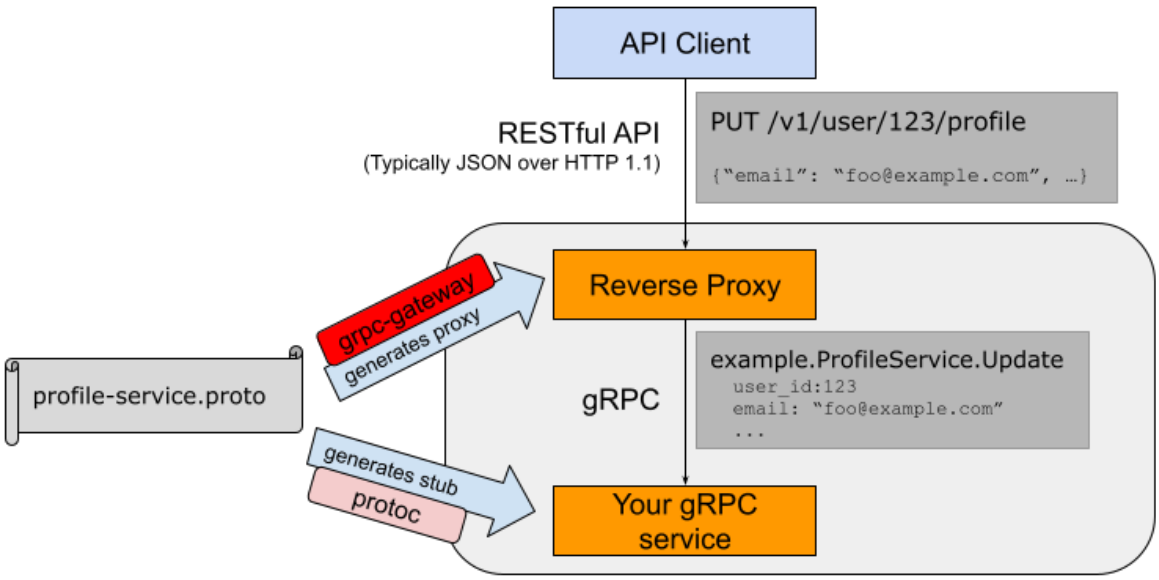
- opentracing
- zipkin

grpc+grpc-gateway 应用

gRPC介绍与环境安装

假定我们有一个项目需求，希望用 `Rpc` 作为内部 `API` 的通讯，同时也想对外提供 `Restful Api` ，写两套又太繁琐不符合

于是我们想到了 `Grpc` 以及 `Grpc Gateway` ，这就是我们所需要的



准备环节

在正式开始我们的 `Grpc` + `Grpc Gateway` 实践前，我们需要先配置好我们的开发环境

- Grpc
- Protoc Plugin
- Protocol Buffers
- Grpc-gateway

Grpc

是什么

Google对 `Grpc` 的定义：

> A high performance, open-source universal RPC framework

也就是 `Grpc` 是一个高性能、开源的通用RPC框架，具有以下特性：

- 强大的`IDL`，使用`Protocol Buffers`作为数据交换的格式，支持`v2`、`v3`（推荐`v3`）
- 跨语言、跨平台，也就是`Grpc`支持多种平台和语言
- **支持HTTP2**，双向传输、多路复用、认证等

安装

1、官方推荐（需科学上网）

```
go get -u google.golang.org/grpc
```

2、通过`github.com`

进入到第一个$GOPATH目录（因为`go get`会默认安装在第一个下）下，新建`google.golang.org`目录，拉取`golang`在`github`上的镜像库：

```
cd /usr/local/go/path/src

mkdir google.golang.org

cd google.golang.org/

git clone https://github.com/grpc/grpc-go

mv grpc-go/ grpc/
```

目录结构：

```
google.golang.org/
└─── grpc
    ...
```

而在`grpc`下有许多常用的包，例如：

- metadata：定义了`grpc`所支持的元数据结构，包中方法可以对`MD`进行获取和处理
- credentials：实现了`grpc`所支持的各种认证凭据，封装了客户端对服务端进行身份验证所需要的所有状态，并做出各种断言
- codes：定义了`grpc`使用的标准错误码，可通用

Protoc Plugin

是什么

编译器插件

安装

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

将 `Protoc Plugin` 的可执行文件从$GOPATH中移动到$GOBIN下

```
mv /usr/local/go/path/bin/protoc-gen-go /usr/local/go/bin/
```

Protocol Buffers v3

是什么

> Protocol buffers are a flexible, efficient, automated mechanism for serializing structured data – think XML, but smaller, faster, and simpler. You define how you want your data to be structured once, then you can use special generated source code to easily write and read your structured data to and from a variety of data streams and using a variety of languages. You can even update your data structure without breaking deployed programs that are compiled against the “old” format.

`Protocol Buffers` 是 `Google` 推出的一种数据描述语言，支持多语言、多平台，它是一种二进制的格式，总得来说就是更小、更快、更简单、更灵活，目前分别有 `v2` 、 `v3` 的版本，我们推荐使用 `v3`

- proto2 文档地址
- proto3 文档地址

建议可以阅读下官方文档的介绍，本系列会在使用时简单介绍所涉及的内容

安装

```
wget https://github.com/google/protobuf/releases/download/v3.5.1/protobuf-all-3.5.1.zip
unzip protobuf-all-3.5.1.zip
```

```
cd protobuf-3.5.1/
./configure
make
make install
```

检查是否安装成功

```
protoc --version
```

如果出现报错

```
protoc: error while loading shared libraries: libprotobuf.so.15: cannot open sha
red object file: No such file or directory
```

则执行 `ldconfig` 后，再次运行即可成功

为什么要执行 `ldconfig`

我们通过控制台输出的信息可以知道， `Protocol Buffers Libraries` 的默认安装路径在 `/usr/local/lib`

```
Libraries have been installed in:
   /usr/local/lib

If you ever happen to want to link against installed libraries
in a given directory, LIBDIR, you must either use libtool, and
specify the full pathname of the library, or use the `-LLIBDIR'
flag during linking and do at least one of the following:
   - add LIBDIR to the `LD_LIBRARY_PATH' environment variable
     during execution
   - add LIBDIR to the `LD_RUN_PATH' environment variable
     during linking
   - use the `-Wl,-rpath -Wl,LIBDIR' linker flag
   - have your system administrator add LIBDIR to `/etc/ld.so.conf'

See any operating system documentation about shared libraries for
more information, such as the ld(1) and ld.so(8) manual pages.
```

而我们安装了一个新的动态链接库， `ldconfig` 一般在系统启动时运行，所以现在会找不到这个 `lib` ，因此我们要手动执行 `ldconfig` ，**让动态链接库为系统所共享，它是一个动态链接库管理命令**，这就是 `ldconfig` 命令的作用

protoc使用

我们按照惯例执行 `protoc --help` （查看帮助文档），我们抽出几个常用的命令进行讲解

1、 `-IPATH, --proto_path=PATH` ：指定 `import` 搜索的目录，可指定多个，如果不指定则默认当前工作目录

2、 `--go_out` ：生成 `golang` 源文件

参数

若要将额外的参数传递给插件，可使用从输出目录中分离出来的逗号分隔的参数列表:

```
protoc --go_out=plugins=grpc,import_path=mypackage:. *.proto
```

- `import_prefix=xxx` ：将指定前缀添加到所有 `import` 路径的开头
- `import_path=foo/bar` ：如果文件没有声明 `go_package` ，则用作包。如果它包含斜杠，那么最右边的斜杠将被忽略。
- `plugins=plugin1+plugin2` ：指定要加载的子插件列表（我们所下载的repo中唯一的插件是grpc）
- `Mfoo/bar.proto=quux/shme` ： `M` 参数，指定 `.proto` 文件编译后的包名（ `foo/bar.proto` 编译后为包名为 `quux/shme` ）

Grpc支持

如果 `proto` 文件指定了 `RPC` 服务， `protoc-gen-go` 可以生成与 `grpc` 相兼容的代码，我们仅需要将 `plugins=grpc` 参数传递给 `--go_out` ，就可以达到这个目的

```
protoc --go_out=plugins=grpc:. *.proto
```

Grpc-gateway

是什么

> grpc-gateway is a plugin of protoc. It reads gRPC service definition, and generates a reverse-proxy server which translates a RESTful JSON API into gRPC. This server is generated according to custom options in your gRPC definition.

grpc-gateway是protoc的一个插件。它读取gRPC服务定义，并生成一个反向代理服务器，将RESTful JSON API转换为gRPC。此服务器是根据gRPC定义中的自定义选项生成的。

安装

```
go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-grpc-gateway
```

如果出现以下报错，我们分析错误提示可得知是连接超时（大概是被墙了）

```
package google.golang.org/genproto/googleapis/api/annotations: unrecognized import path "google.golang.org/genproto/googleapis/api/annotations" (https fetch: Get https://google.golang.org/genproto/googleapis/api/annotations?go-get=1: dial tcp 216.239.37.1:443: getsockopt: connection timed out)
```

有两种解决方法，

1、科学上网

2、通过 `github.com`

进入到第一个$GOTPATH目录的 `google.golang.org` 目录下，拉取 `genproto` 在 `github` 上的 `go-genproto` 镜像库：

```
cd /usr/local/go/path/src/google.golang.org

git clone https://github.com/google/go-genproto.git

mv go-genproto/ genproto/
```

在安装完毕后，我们将 `grpc-gateway` 的可执行文件从$GOPATH中移动到$GOBIN

```
mv /usr/local/go/path/bin/protoc-gen-grpc-gateway /usr/local/go/bin/
```

到这里我们这节就基本完成了，建议多反复看几遍加深对各个组件的理解！

参考

示例代码

- grpc-hello-world

Hello World

这节将开始编写一个复杂的Hello World，涉及到许多的知识，建议大家认真思考其中的概念

需求

由于本实践偏向 `Grpc` + `Grpc Gateway` 的方面，我们的需求是**同一个服务端支持 `Rpc` 和 `Restful Api`** ，那么就意味着 `http2` 、 `TLS` 等等的应用，功能方面就是一个服务端能够接受来自 `grpc` 和 `Restful Api` 的请求并响应

一、初始化目录

我们先在$GOPATH中新建 `grpc-hello-world` 文件夹，我们项目的初始目录目录如下：

```
grpc-hello-world/
├─── certs
├─── client
├─── cmd
├─── pkg
├─── proto
|    ├─── google
|    |    └─── api
└─── server
```

- `certs` ：证书凭证
- `client` ：客户端
- `cmd` ：命令行
- `pkg` ：第三方公共模块
- `proto` ： `protobuf` 的一些相关文件（含 `.proto` 、 `pb.go` 、 `.pb.gw.go` ）， `google/api` 中用于存放 `annotations.proto` 、 `http.proto`
- `server` ：服务端

二、制作证书

在服务端支持 `Rpc` 和 `Restful Api` ，需要用到 `TLS` ，因此我们要先制作证书

进入 `certs` 目录，生成 `TLS` 所需的公钥密钥文件

私钥

```
openssl genrsa -out server.key 2048

openssl ecparam -genkey -name secp384r1 -out server.key
```

- `openssl genrsa`：生成`RSA`私钥，命令的最后一个参数，将指定生成密钥的位数，如果没有指定，默认512
- `openssl ecparam`：生成`ECC`私钥，命令为椭圆曲线密钥参数生成及操作，本文中`ECC`曲线选择的是`secp384r1`

自签名公钥

```
openssl req -new -x509 -sha256 -key server.key -out server.pem -days 3650
```

- `openssl req`：生成自签名证书，`-new`指生成证书请求、`-sha256`指使用`sha256`加密、`-key`指定私钥文件、`-x509`指输出证书、`-days 3650`为有效期，此后则输入证书拥有者信息

填写信息

```
Country Name (2 letter code) [XX]:
State or Province Name (full name) []:
Locality Name (eg, city) [Default City]:
Organization Name (eg, company) [Default Company Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (eg, your name or your server's hostname) []:grpc server name
Email Address []:
```

三、`proto`

编写

1、`google.api`

我们看到`proto`目录中有`google/api`目录，它用到了`google`官方提供的两个`api`描述文件，主要是针对`grpc-gateway`的`http`转换提供支持，定义了`Protocol Buffer`所扩展的`HTTP Option`

`annotations.proto` 文件：

```
// Copyright (c) 2015, Google Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";

package google.api;

import "google/api/http.proto";
import "google/protobuf/descriptor.proto";

option java_multiple_files = true;
option java_outer_classname = "AnnotationsProto";
option java_package = "com.google.api";

extend google.protobuf.MethodOptions {
  // See `HttpRule`.
  HttpRule http = 72295728;
}
```

`http.proto` 文件：

```
// Copyright 2016 Google Inc.
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//     http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
```

```
// See the License for the specific language governing permissions and
// limitations under the License.

syntax = "proto3";

package google.api;

option cc_enable_arenas = true;
option java_multiple_files = true;
option java_outer_classname = "HttpProto";
option java_package = "com.google.api";


// Defines the HTTP configuration for a service.  It contains a list of
// [HttpRule][google.api.HttpRule], each specifying the mapping of an RPC method
// to one or more HTTP REST API methods.
message Http {
  // A list of HTTP rules for configuring the HTTP REST API methods.
  repeated HttpRule rules = 1;
}

// Use CustomHttpPattern to specify any HTTP method that is not included in the
// `pattern` field, such as HEAD, or "*" to leave the HTTP method unspecified fo
r
// a given URL path rule. The wild-card rule is useful for services that provide
// content to Web (HTML) clients.
message HttpRule {
  // Selects methods to which this rule applies.
  //
  // Refer to [selector][google.api.DocumentationRule.selector] for syntax detai
ls.
  string selector = 1;

  // Determines the URL pattern is matched by this rules. This pattern can be
  // used with any of the {get|put|post|delete|patch} methods. A custom method
  // can be defined using the 'custom' field.
  oneof pattern {
    // Used for listing and getting information about resources.
    string get = 2;

    // Used for updating a resource.
    string put = 3;

    // Used for creating a resource.
    string post = 4;
```

```
    // Used for deleting a resource.
    string delete = 5;

    // Used for updating a resource.
    string patch = 6;

    // Custom pattern is used for defining custom verbs.
    CustomHttpPattern custom = 8;
  }

  // The name of the request field whose value is mapped to the HTTP body, or
  // `*` for mapping all fields not captured by the path pattern to the HTTP
  // body. NOTE: the referred field must not be a repeated field.
  string body = 7;

  // Additional HTTP bindings for the selector. Nested bindings must
  // not contain an `additional_bindings` field themselves (that is,
  // the nesting may only be one level deep).
  repeated HttpRule additional_bindings = 11;
}

// A custom pattern is used for defining custom HTTP verb.
message CustomHttpPattern {
  // The name of this custom HTTP verb.
  string kind = 1;

  // The path matched by this custom verb.
  string path = 2;
}
```

2. `hello.proto`

这一小节将编写 `Demo` 的 `.proto` 文件，我们在 `proto` 目录下新建 `hello.proto` 文件，写入文件内容：

```
syntax = "proto3";

package proto;

import "google/api/annotations.proto";

service HelloWorld {
    rpc SayHelloWorld(HelloWorldRequest) returns (HelloWorldResponse) {
        option (google.api.http) = {
            post: "/hello_world"
```

```
            body: "*"
        };
    }
}

message HelloWorldRequest {
    string referer = 1;
}

message HelloWorldResponse {
    string message = 1;
}
```

在 `hello.proto` 文件中，引用了 `google/api/annotations.proto` ，达到支持 `HTTP Option` 的效果

- 定义了一个 `service` RPC服务 `HelloWorld` ，在其内部定义了一个 `HTTP Option` 的 `POST` 方法， `HTTP` 响应路径为 `/hello_world`
- 定义 `message` 类型 `HelloWorldRequest` 、 `HelloWorldResponse` ，用于响应请求和返回结果

编译

在编写完 `.proto` 文件后，我们需要对其进行编译，就能够在 `server` 中使用

进入 `proto` 目录，执行以下命令

```
# 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为hello_http.pb.proto
protoc -I . --go_out=plugins=grpc,Mgoogle/api/annotations.proto=grpc-hello-world/proto/google/api:. ./hello.proto

#编译hello_http.proto为hello_http.pb.gw.proto
protoc --grpc-gateway_out=logtostderr=true:. ./hello.proto
```

执行完毕后将生成 `hello.pb.go` 和 `hello.gw.pb.go` ，分别针对 `grpc` 和 `grpc-gateway` 的功能支持

四、命令行模块 `cmd`

介绍

这一小节我们编写命令行模块，为什么要独立出来呢，是为了将 `cmd` 和 `server` 两者解耦，避免混淆在一起。

我们采用 Cobra 来完成这项功能， `Cobra` 既是创建强大的现代CLI应用程序的库，也是生成应用程序和命令文件的程序。提供了以下功能：

- 简易的子命令行模式
- 完全兼容posix的命令行模式(包括短和长版本)
- 嵌套的子命令
- 全局、本地和级联 `flags`
- 使用 `Cobra` 很容易的生成应用程序和命令，使用 `cobra create appname` 和 `cobra add cmdname`
- 智能提示
- 自动生成commands和flags的帮助信息
- 自动生成详细的help信息 `-h` ， `--help` 等等
- 自动生成的bash自动完成功能
- 为应用程序自动生成手册
- 命令别名
- 定义您自己的帮助、用法等的灵活性。
- 可选与viper紧密集成的apps

编写 `server`

在编写 `cmd` 时需要先用 `server` 进行测试关联，因此这一步我们先写 `server.go` 用于测试

在 `server` 模块下 新建 `server.go` 文件，写入测试内容：

```
package server

import (
    "log"
)

var (
    ServerPort string
    CertName string
    CertPemPath string
```

```
    CertKeyPath string
)

func Serve() (err error){
    log.Println(ServerPort)

    log.Println(CertName)

    log.Println(CertPemPath)

    log.Println(CertKeyPath)

    return nil
}
```

编写 `cmd`

在 `cmd` 模块下 新建 `root.go` 文件，写入内容：

```
package cmd

import (
    "fmt"
    "os"

    "github.com/spf13/cobra"
)

var rootCmd = &cobra.Command{
    Use:   "grpc",
    Short: "Run the gRPC hello-world server",
}

func Execute() {
    if err := rootCmd.Execute(); err != nil {
        fmt.Println(err)
        os.Exit(-1)
    }
}
```

新建 `server.go` 文件，写入内容：

```
package cmd
```

```
import (
    "log"

    "github.com/spf13/cobra"

    "grpc-hello-world/server"
)

var serverCmd = &cobra.Command{
    Use:   "server",
    Short: "Run the gRPC hello-world server",
    Run: func(cmd *cobra.Command, args []string) {
        defer func() {
            if err := recover(); err != nil {
                log.Println("Recover error : %v", err)
            }
        }()

        server.Serve()
    },
}

func init() {
    serverCmd.Flags().StringVarP(&server.ServerPort, "port", "p", "50052", "serv
er port")
    serverCmd.Flags().StringVarP(&server.CertPemPath, "cert-pem", "", "./certs/s
erver.pem", "cert pem path")
    serverCmd.Flags().StringVarP(&server.CertKeyPath, "cert-key", "", "./certs/s
erver.key", "cert key path")
    serverCmd.Flags().StringVarP(&server.CertName, "cert-name", "", "grpc server
name", "server's hostname")
    rootCmd.AddCommand(serverCmd)
}
```

我们在 `grpc-hello-world/` 目录下，新建文件 `main.go` ，写入内容：

```
package main

import (
    "grpc-hello-world/cmd"
)

func main() {
    cmd.Execute()
}
```

讲解

要使用 `Cobra` ，按照 `Cobra` 标准要创建 `main.go` 和一个 `rootCmd` 文件，另外我们有子命令 `server`

1、 `rootCmd` ：
`rootCmd` 表示在没有任何子命令的情况下的基本命令

2、 `&cobra.Command` ：

- `Use` ： `Command` 的用法， `Use` 是一个行用法消息
- `Short` ： `Short` 是 `help` 命令输出中显示的简短描述
- `Run` ：运行:典型的实际工作功能。大多数命令只会实现这一点；另外还有 `PreRun` 、 `PreRunE` 、 `PostRun` 、 `PostRunE` 等等不同时期的运行命令，但比较少用，具体使用时再查看亦可

3、 `rootCmd.AddCommand` ： `AddCommand` 向这父命令（ `rootCmd` ）添加一个或多个命令

4、 `serverCmd.Flags().StringVarP()` ：

一般来说，我们需要在 `init()` 函数中定义 `flags` 和处理配置，以 `serverCmd.Flags().StringVarP(&server.ServerPort, "port", "p", "50052", "server port")` 为例，我们定义了一个 `flag` ，值存储在 `&server.ServerPort` 中，长命令为 `--port` ，短命令为 `-p` ，，默认值为 `50052` ，命令的描述为 `server port` 。这一种调用方式成为 `Local Flags`

我们延伸一下，如果觉得每一个子命令都要设一遍觉得很麻烦，我们可以采用 `Persistent Flags` ：

`rootCmd.PersistentFlags().BoolVarP(&Verbose, "verbose", "v", false, "verbose output")`

作用：

`flag` 是可以持久的，这意味着该 `flag` 将被分配给它所分配的命令以及该命令下的每个命令。对于全局标记，将标记作为根上的持久标志。

另外还有 `Local Flag on Parent Commands` 、 `Bind Flags with Config` 、 `Required flags` 等等，使用到再 传送 了解即可

测试

回到 `grpc-hello-world/` 目录下执行 `go run main.go server` ，查看输出是否为（此时应为默认值）：

```
2018/02/25 23:23:21 50052
2018/02/25 23:23:21 dev
2018/02/25 23:23:21 ./certs/server.pem
2018/02/25 23:23:21 ./certs/server.key
```

执行 `go run main.go server --port=8000 --cert-pem=test-pem --cert-key=test-key --cert-name=test-name` ，检验命令行参数是否正确：

```
2018/02/25 23:24:56 8000
2018/02/25 23:24:56 test-name
2018/02/25 23:24:56 test-pem
2018/02/25 23:24:56 test-key
```

若都无误，那么恭喜你 `cmd` 模块的编写正确了，下一部分开始我们的重点章节！

五、服务端模块 `server`

编写 `hello.go`

在 `server` 目录下新建文件 `hello.go` ，写入文件内容：

```
package server

import (
    "golang.org/x/net/context"

    pb "grpc-hello-world/proto"
)

type helloService struct{}

func NewHelloService() *helloService {
    return &helloService{}
}

func (h helloService) SayHelloWorld(ctx context.Context, r *pb.HelloWorldRequest) (*pb.HelloWorldResponse, error) {
    return &pb.HelloWorldResponse{
        Message : "test",
    }, nil
}
```

我们创建了 `helloService` 及其方法 `SayHelloWorld` ，对应 `.proto` 的 `rpc SayHelloWorld` ，这个方法需要有2个参数： `ctx context.Context` 用于接受上下文参数、 `r *pb.HelloWorldRequest` 用于接受 `protobuf` 的 `Request` 参数（对应 `.proto` 的 `message HelloWorldRequest` ）

*编写 `server.go`

这一小章节，我们编写最为重要的服务端程序部分，涉及到大量的 `grpc` 、 `grpc-gateway` 及一些网络知识的应用

1、在 `pkg` 下新建 `util` 目录，新建 `grpc.go` 文件，写入内容：

```
package util

import (
    "net/http"
    "strings"

    "google.golang.org/grpc"
)

func GrpcHandlerFunc(grpcServer *grpc.Server, otherHandler http.Handler) http.Handler {
    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)
        }
    })
}
```

`GrpcHandlerFunc` 函数是用于判断请求是来源于 `Rpc` 客户端还是 `Restful Api` 的请求，根据不同的请求注册不同的 `ServeHTTP` 服务； `r.ProtoMajor == 2` 也代表着请求必须基于 `HTTP/2`

2、在 `pkg` 下的 `util` 目录下，新建 `tls.go` 文件，写入内容：

```
package util

import (
    "crypto/tls"
    "io/ioutil"
    "log"

    "golang.org/x/net/http2"
)

func GetTLSConfig(certPemPath, certKeyPath string) *tls.Config {
    var certKeyPair *tls.Certificate
    cert, _ := ioutil.ReadFile(certPemPath)
    key, _ := ioutil.ReadFile(certKeyPath)

    pair, err := tls.X509KeyPair(cert, key)
    if err != nil {
        log.Println("TLS KeyPair err: %v\n", err)
    }

    certKeyPair = &pair

    return &tls.Config{
        Certificates: []tls.Certificate{*certKeyPair},
        NextProtos:   []string{http2.NextProtoTLS},
    }
}
```

`GetTLSConfig` 函数是用于获取 `TLS` 配置，在内部，我们读取了 `server.key` 和 `server.pem` 这类证书凭证文件

- `tls.X509KeyPair`：从一对 `PEM` 编码的数据中解析公钥/私钥对。成功则返回公钥/私钥对
- `http2.NextProtoTLS`：`NextProtoTLS` 是谈判期间的 `NPN/ALPN` 协议，用于**HTTP/2的TLS设置**
- `tls.Certificate`：返回一个或多个证书，实质我们解析 `PEM` 调用的 `X509KeyPair` 的函数声明就是 `func X509KeyPair(certPEMBlock, keyPEMBlock []byte) (Certificate, error)`，返回值就是 `Certificate`

总的来说该函数是用于处理从证书凭证文件（PEM），最终获取 `tls.Config` 作为 `HTTP2` 的使用参数

3、修改 `server` 目录下的 `server.go` 文件，该文件是我们服务里的核心文件，写入内容：

```
package server

import (
    "crypto/tls"
    "net"
    "net/http"
    "log"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "github.com/grpc-ecosystem/grpc-gateway/runtime"

    pb "grpc-hello-world/proto"
    "grpc-hello-world/pkg/util"
)

var (
    ServerPort string
    CertName string
    CertPemPath string
    CertKeyPath string
    EndPoint string
)

func Serve() (err error){
    EndPoint = ":" + ServerPort
    conn, err := net.Listen("tcp", EndPoint)
    if err != nil {
        log.Printf("TCP Listen err:%v\n", err)
    }

    tlsConfig := util.GetTLSConfig(CertPemPath, CertKeyPath)
    srv := createInternalServer(conn, tlsConfig)

    log.Printf("gRPC and https listen on: %s\n", ServerPort)

    if err = srv.Serve(tls.NewListener(conn, tlsConfig)); err != nil {
        log.Printf("ListenAndServe: %v\n", err)
    }

    return err
}
```

```
func createInternalServer(conn net.Listener, tlsConfig *tls.Config) (*http.Server) {
    var opts []grpc.ServerOption

    // grpc server
    creds, err := credentials.NewServerTLSFromFile(CertPemPath, CertKeyPath)
    if err != nil {
        log.Printf("Failed to create server TLS credentials %v", err)
    }

    opts = append(opts, grpc.Creds(creds))
    grpcServer := grpc.NewServer(opts...)

    // register grpc pb
    pb.RegisterHelloWorldServer(grpcServer, NewHelloService())

    // gw server
    ctx := context.Background()
    dcreds, err := credentials.NewClientTLSFromFile(CertPemPath, CertName)
    if err != nil {
        log.Printf("Failed to create client TLS credentials %v", err)
    }
    dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}
    gwmux := runtime.NewServeMux()

    // register grpc-gateway pb
    if err := pb.RegisterHelloWorldHandlerFromEndpoint(ctx, gwmux, EndPoint, dopts); err != nil {
        log.Printf("Failed to register gw server: %v\n", err)
    }

    // http服务
    mux := http.NewServeMux()
    mux.Handle("/", gwmux)

    return &http.Server{
        Addr:      EndPoint,
        Handler:   util.GrpcHandlerFunc(grpcServer, mux),
        TLSConfig: tlsConfig,
    }
}
```

`server` 流程剖析

我们将这一大块代码，分成以下几个部分来理解

一、启动监听

`net.Listen("tcp", EndPoint)` 用于监听本地的网络地址通知，它的函数原型 `func Listen(network, address string) (Listener, error)`

参数： `network` 必须传入 `tcp` 、 `tcp4` 、 `tcp6` 、 `unix` 、 `unixpacket` ，若 `address` 为空或为0则会自动选择一个端口号
返回值：通过查看源码我们可以得知其返回值为 `Listener` ，结构体原型：

```
type Listener interface {
    Accept() (Conn, error)
    Close() error
    Addr() Addr
}
```

通过分析得知，**最后 `net.Listen` 会返回一个监听器的结构体，返回给接下来的动作，让其执行下一步的操作**，它可以执行三类操作

- `Accept` ：接受等待并将下一个连接返回给 `Listener`
- `Close` ：关闭 `Listener`
- `Addr` ：返回 `Listener` 的网络地址

二、获取 `TLS`

通过 `util.GetTLSConfig` 解析得到 `tls.Config` ，传达给 `http.Server` 服务的 `TLSConfig` 配置项使用

三、创建内部服务

`createInternalServer` 函数，是整个服务端的核心流转部分

程序采用的是 `HTT2` 、 `HTTPS` 也就是需要支持 `TLS` ，因此在启动 `grpc.NewServer` 前，我们要将认证的中间件注册进去

而前面所获取的 `tlsConfig` 仅能给 `HTTP` 使用，因此**第一步**我们要创建 `grpc` 的 `TLS` 认证凭证

1、创建 `grpc` 的 `TLS` 认证凭证

新增引用 `google.golang.org/grpc/credentials` 的第三方包，它实现了 `grpc` 库支持的各种凭证，该凭证封装了客户机需要的所有状态，以便与服务器进行身份验证并进行各种断言，例如关于客户机的身份，角色或是否授权进行特定的呼叫

我们调用 `NewServerTLSFromFile` 来达到我们的目的，它能够从输入证书文件和服务器的密钥文件**构造TLS证书凭证**

```
func NewServerTLSFromFile(certFile, keyFile string) (TransportCredentials, error) {
    //LoadX509KeyPair读取并解析来自一对文件的公钥/私钥对
    cert, err := tls.LoadX509KeyPair(certFile, keyFile)
    if err != nil {
        return nil, err
    }
    //NewTLS使用tls.Config来构建基于TLS的TransportCredentials
    return NewTLS(&tls.Config{Certificates: []tls.Certificate{cert}}), nil
}
```

2、设置 `grpc ServerOption`

以 `grpc.Creds(creds)` 为例，其原型为 `func Creds(c credentials.TransportCredentials) ServerOption` ，该函数返回 `ServerOption` ，它为服务器连接设置凭据

3、创建 `grpc` 服务端

函数原型：

```
func NewServer(opt ...ServerOption) *Server
```

我们在此处创建了一个没有注册服务的 `grpc` 服务端，还没有开始接受请求

```
grpcServer := grpc.NewServer(opts...)
```

4、注册 `grpc` 服务

```
pb.RegisterHelloWorldServer(grpcServer, NewHelloService())
```

5、创建 `grpc-gateway` 关联组件

```
ctx := context.Background()
dcreds, err := credentials.NewClientTLSFromFile(CertPemPath, CertName)
if err != nil {
    log.Println("Failed to create client TLS credentials %v", err)
}
dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}
```

- `context.Background`：返回一个非空的空上下文。它没有被注销，没有值，没有过期时间。它通常由主函数、初始化和测试使用，并作为传入请求的**顶级上下文**
- `credentials.NewClientTLSFromFile`：从客户机的输入证书文件构造TLS凭证
- `grpc.WithTransportCredentials`：配置一个连接级别的安全凭据(例：`TLS`、`SSL`)，返回值为`type DialOption`
- `grpc.DialOption`：`DialOption`选项配置我们如何设置连接（其内部具体由多个的`DialOption`组成，决定其设置连接的内容）

6、创建`HTTP NewServeMux`及注册`grpc-gateway`逻辑

```
gwmux := runtime.NewServeMux()

// register grpc-gateway pb
if err := pb.RegisterHelloWorldHandlerFromEndpoint(ctx, gwmux, EndPoint, dopts);
err != nil {
    log.Println("Failed to register gw server: %v\n", err)
}

// http服务
mux := http.NewServeMux()
mux.Handle("/", gwmux)
```

- `runtime.NewServeMux`：返回一个新的`ServeMux`，它的内部映射是空的；`ServeMux`是`grpc-gateway`的一个请求多路复用器。它将`http`请求与模式匹配，并调用相应的处理程序
- `RegisterHelloWorldHandlerFromEndpoint`：如函数名，注册`HelloWorld`服务的`HTTP Handle`到`grpc`端点
- `http.NewServeMux`：`分配并返回一个新的ServeMux`
- `mux.Handle`：为给定模式注册处理程序

（带着疑问去看程序）为什么`gwmux`可以放入`mux.Handle`中？

首先我们看看它们的原型是怎么样的

（1）`http.NewServeMux()`

```
func NewServeMux() *ServeMux {
        return new(ServeMux)
}
```

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}
```

（2） `runtime.NewServeMux` ?

```
func NewServeMux(opts ...ServeMuxOption) *ServeMux {
    serveMux := &ServeMux{
        handlers:               make(map[string][]handler),
        forwardResponseOptions: make([]func(context.Context, http.ResponseWrite
r, proto.Message) error, 0),
        marshalers:             makeMarshalerMIMERegistry(),
    }
    ...
    return serveMux
}
```

（3） `http.NewServeMux()` 的 `Handle` 方法

```
func (mux *ServeMux) Handle(pattern string, handler Handler)
```

通过分析可得知，两者 `NewServeMux` 都是最终返回 `serveMux` ， `Handler` 中导出的方法仅有 `ServeHTTP` ，功能是用于响应HTTP请求

我们回到 `Handle interface` 中，可以得出结论就是任何结构体，只要实现了 `ServeHTTP` 方法，这个结构就可以称为 `Handle` ， `ServeMux` 会使用该 `Handler` 调用 `ServeHTTP` 方法处理请求，这也就是**自定义** **`Handler`**

而我们这里正是将 `grpc-gateway` 中注册好的 `HTTP Handler` 无缝的植入到 `net/http` 的 `Handle` 方法中

补充：在 `go` 中任何结构体只要实现了与接口相同的方法，就等同于实现了接口

7、注册具体服务

```
if err := pb.RegisterHelloWorldHandlerFromEndpoint(ctx, gwmux, EndPoint, dopts);
err != nil {
    log.Println("Failed to register gw server: %v\n", err)
}
```

在这段代码中，我们利用了前几小节的

- 上下文

- `gateway-grpc`的请求多路复用器
- 服务网络地址
- 配置好的安全凭据

注册了 `HelloWorld` 这一个服务

四、创建`tls.NewListener`

```
func NewListener(inner net.Listener, config *Config) net.Listener {
    l := new(listener)
    l.Listener = inner
    l.config = config
    return l
}
```

`NewListener` 将会创建一个 `Listener` ，它接受两个参数，第一个是来自内部 `Listener` 的监听器，第二个参数是 `tls.Config` （必须包含至少一个证书）

五、服务开始接受请求

在最后我们调用 `srv.Serve(tls.NewListener(conn, tlsConfig))` ，可以得知它是 `http.Server` 的方法，并且需要一个 `Listener` 作为参数，那么 `Serve` 内部做了些什么事呢？

```
func (srv *Server) Serve(l net.Listener) error {
    defer l.Close()
    ...

    baseCtx := context.Background() // base is always background, per Issue 1622
0
    ctx := context.WithValue(baseCtx, ServerContextKey, srv)
    for {
        rw, e := l.Accept()
        ...
        c := srv.newConn(rw)
        c.setState(c.rwc, StateNew) // before Serve can return
        go c.serve(ctx)
    }
}
```

粗略的看，它创建了一个 `context.Background()` 上下文对象，并调用 `Listener` 的 `Accept` 方法开始接受外部请求，在获取到连接数据后使用 `newConn` 创建连接对象，在最后使用 `goroutine` 的方式处理连接请求，达到其目的

补充：对于 `HTTP/2` 支持，在调用 `Serve` 之前，应将 `srv.TLSConfig` 初始化为提供的 `Listener` 的**TLS**配置。如果 `srv.TLSConfig` 非零，并且在 `Config.NextProtos` 中不包含字符串 `h2` ，则不启用 `HTTP/2` 支持

六、验证功能

编写测试客户端

在 `grpc-hello-world/` 下新建目录 `client` ，新建 `client.go` 文件，新增内容：

```
package main

import (
    "log"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"

    pb "grpc-hello-world/proto"
)

func main() {
    creds, err := credentials.NewClientTLSFromFile("../certs/server.pem", "dev")
    if err != nil {
        log.Println("Failed to create TLS credentials %v", err)
    }
    conn, err := grpc.Dial(":50052", grpc.WithTransportCredentials(creds))
    defer conn.Close()

    if err != nil {
        log.Println(err)
    }

    c := pb.NewHelloWorldClient(conn)
    context := context.Background()
    body := &pb.HelloWorldRequest{
        Referer : "Grpc",
    }

    r, err := c.SayHelloWorld(context, body)
    if err != nil {
        log.Println(err)
    }
```

```
    log.Println(r.Message)
}
```

由于客户端只是展示测试用，就简单的来了，原本它理应归类到 `cobra` 的管控下，配置管理等等都应可控化

在看这篇文章的你，可以试试将测试客户端归类好

启动服务端

回到 `grpc-hello-world/` 目录下，启动服务端 `go run main.go server` ，成功则仅返回

```
2018/02/26 17:19:36 gRPC and https listen on: 50052
```

执行测试客户端

回到 `client` 目录下，启动客户端 `go run client.go` ，成功则返回

```
2018/02/26 17:22:57 Grpc
```

执行测试Restful Api

```
curl -X POST -k https://localhost:50052/hello_world -d '{"referer": "restful_api"}'
```

成功则返回 `{"message":"restful_api"}`

最终目录结构

```
grpc-hello-world
├── certs
│   ├── server.key
│   └── server.pem
├── client
│   └── client.go
├── cmd
│   ├── root.go
│   └── server.go
```

```
├── main.go
├── pkg
│   └── util
│       ├── grpc.go
│       └── tls.go
├── proto
│   ├── google
│   │   └── api
│   │       ├── annotations.pb.go
│   │       ├── annotations.proto
│   │       ├── http.pb.go
│   │       └── http.proto
│   ├── hello.pb.go
│   ├── hello.pb.gw.go
│   └── hello.proto
└── server
    ├── hello.go
    └── server.go
```

至此本节就结束了，推荐一下 `jergoo` 的文章，大家有时间可以看看

另外本节涉及了许多组件间的知识，值得大家细细的回味，非常有意义！

参考

示例代码

- grpc-hello-world

Swagger了解一下

在上一节，我们完成了一个服务端同时支持 `Rpc` 和 `RESTful Api` 后，你以为自己大功告成了，结果突然发现要写 `Api` 文档和前端同事对接= = 。。。

你寻思有没有什么组件能够自动化生成 `Api` 文档来解决这个问题，就在这时你发现了 `Swagger` ，一起了解一下吧！

介绍

Swagger

`Swagger` 是全球最大的 `OpenAPI` 规范（OAS）API开发工具框架，支持从设计和文档到测试和部署的整个API生命周期的开发

`Swagger` 是目前最受欢迎的 `RESTful Api` 文档生成工具之一，主要的原因如下

- 跨平台、跨语言的支持
- 强大的社区
- 生态圈 Swagger Tools（Swagger Editor、Swagger Codegen、Swagger UI ...）
- 强大的控制台

同时 `grpc-gateway` 也支持 `Swagger`

[image]

`OpenAPI` 规范

`OpenAPI` 规范是 `Linux` 基金会的一个项目，试图通过定义一种用来描述API格式或API定义的语言，来规范 `RESTful` 服务开发过程。 `OpenAPI` 规范帮助我们描述一个API的基本信息，比如：

- 有关该API的一般性描述
- 可用路径（/资源）
- 在每个路径上的可用操作（获取/提交...）
- 每个操作的输入/输出格式

目前V2.0版本的OpenAPI规范（也就是SwaggerV2.0规范）已经发布并开源在github上。该文档写的非常好，结构清晰，方便随时查阅。

注： `OpenAPI` 规范的介绍引用自原文

使用

生成`Swagger`的说明文件

第一，我们需要检查$GOBIN下是否包含`protoc-gen-swagger`可执行文件

若不存在则需要执行：

```
go get -u github.com/grpc-ecosystem/grpc-gateway/protoc-gen-swagger
```

等待执行完毕后，可在`$GOPATH/bin`下发现该执行文件，将其移动到`$GOBIN`下即可

第二，回到`$GOPATH/src/grpc-hello-world/proto`下，执行命令

```
protoc -I/usr/local/include -I. -I$GOPATH/src/grpc-hello-world/proto/google/api
 --swagger_out=logtostderr=true:. ./hello.proto
```

成功后执行`ls`即可看到`hello.swagger.json`文件

下载`Swagger UI`文件

`Swagger`提供可视化的`API`管理平台，就是Swagger UI

我们将其源码下载下来，并将其`dist`目录下的所有文件拷贝到我们项目中的`$GOPATH/src/grpc-hello-world/third_party/swagger-ui`去

将`Swagger UI`转换为`Go`源代码

在这里我们使用的转换工具是go-bindata

它支持将任何文件转换为可管理的`Go`源代码。用于将二进制数据嵌入到`Go`程序中。并且在将文件数据转换为原始字节片之前，可以选择压缩文件数据

安装

```
go get -u github.com/jteeuwen/go-bindata/...
```

完成后，将`$GOPATH/bin`下的`go-bindata`移动到`$GOBIN`下

转换

在项目下新建`pkg/ui/data/swagger`目录，回到`$GOPATH/src/grpc-hello-world/third_party/swagger-ui`下，执行命令

```
go-bindata --nocompress -pkg swagger -o pkg/ui/data/swagger/datafile.go third_party/swagger-ui/...
```

检查

回到 `pkg/ui/data/swagger` 目录，检查是否存在 `datafile.go` 文件

`Swagger UI` 文件服务器（对外提供服务）

在这一步，我们需要使用与其配套的go-bindata-assetfs

它能够使用 `go-bindata` 所生成 `Swagger UI` 的 `Go` 代码，结合 `net/http` 对外提供服务

安装

```
go get github.com/elazarl/go-bindata-assetfs/...
```

编写

通过分析，我们得知生成的文件提供了一个 `assetFS` 函数，该函数返回一个封装了嵌入文件的 `http.Filesystem` ，可以用其来提供一个 `HTTP` 服务

那么我们来编写 `Swagger UI` 的代码吧，主要是两个部分，一个是 `swagger.json` ，另外一个是 `swagger-ui` 的响应

serveSwaggerFile

引用包 `strings` 、 `path`

```
func serveSwaggerFile(w http.ResponseWriter, r *http.Request) {
	if ! strings.HasSuffix(r.URL.Path, "swagger.json") {
		log.Printf("Not Found: %s", r.URL.Path)
		http.NotFound(w, r)
		return
	}

	p := strings.TrimPrefix(r.URL.Path, "/swagger/")
	p = path.Join("proto", p)

	log.Printf("Serving swagger-file: %s", p)

	http.ServeFile(w, r, p)
}
```

在函数中，我们利用 `r.URL.Path` 进行路径后缀判断

主要做了对 `swagger.json` 的文件访问支持（提供 `https://127.0.0.1:50052/swagger/hello.swagger.json` 的访问）

serveSwaggerUI

引用包 `github.com/elazarl/go-bindata-assetfs` 、 `grpc-hello-world/pkg/ui/data/swagger`

```
func serveSwaggerUI(mux *http.ServeMux) {
    fileServer := http.FileServer(&assetfs.AssetFS{
        Asset:    swagger.Asset,
        AssetDir: swagger.AssetDir,
        Prefix:   "third_party/swagger-ui",
    })
    prefix := "/swagger-ui/"
    mux.Handle(prefix, http.StripPrefix(prefix, fileServer))
}
```

在函数中，我们使用了go-bindata-assetfs来调度先前生成的 `datafile.go` ，结合 `net/http` 来对外提供 `swagger-ui` 的服务

结合

在完成功能后，我们发现 `path.Join("proto", p)` 是写死参数的，这样显然不对，我们应该将其导出成外部参数，那么我们来最终改造一番

首先我们在 `server.go` 新增包全局变量 `SwaggerDir` ，修改 `cmd/server.go` 文件：

```
package cmd

import (
    "log"

    "github.com/spf13/cobra"

    "grpc-hello-world/server"
)

var serverCmd = &cobra.Command{
    Use:   "server",
    Short: "Run the gRPC hello-world server",
    Run: func(cmd *cobra.Command, args []string) {
        defer func() {
```

```
            if err := recover(); err != nil {
                log.Println("Recover error : %v", err)
            }
        }()

        server.Run()
    },
}

func init() {
    serverCmd.Flags().StringVarP(&server.ServerPort, "port", "p", "50052", "serv
er port")
    serverCmd.Flags().StringVarP(&server.CertPemPath, "cert-pem", "", "./conf/ce
rts/server.pem", "cert-pem path")
    serverCmd.Flags().StringVarP(&server.CertKeyPath, "cert-key", "", "./conf/ce
rts/server.key", "cert-key path")
    serverCmd.Flags().StringVarP(&server.CertServerName, "cert-server-name", ""
"grpc server name", "server's hostname")
    serverCmd.Flags().StringVarP(&server.SwaggerDir, "swagger-dir", "", "proto",
"path to the directory which contains swagger definitions")

    rootCmd.AddCommand(serverCmd)
}
```

修改 `path.Join("proto", p)` 为 `path.Join(SwaggerDir, p)` ，这样的话我们 `swagger.json` 的文件路径就可以根据外部情况去修改它

最终 `server.go` 文件内容：

```
package server

import (
    "crypto/tls"
    "net"
    "net/http"
    "log"
    "strings"
    "path"

    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "google.golang.org/grpc/credentials"
    "github.com/grpc-ecosystem/grpc-gateway/runtime"
    "github.com/elazarl/go-bindata-assetfs"
```

```
    pb "grpc-hello-world/proto"
    "grpc-hello-world/pkg/util"
    "grpc-hello-world/pkg/ui/data/swagger"
)

var (
    ServerPort string
    CertServerName string
    CertPemPath string
    CertKeyPath string
    SwaggerDir string
    EndPoint string

    tlsConfig *tls.Config
)

func Run() (err error) {
    EndPoint = ":" + ServerPort
    tlsConfig = util.GetTLSConfig(CertPemPath, CertKeyPath)

    conn, err := net.Listen("tcp", EndPoint)
    if err != nil {
        log.Printf("TCP Listen err:%v\n", err)
    }

    srv := newServer(conn)

    log.Printf("gRPC and https listen on: %s\n", ServerPort)

    if err = srv.Serve(util.NewTLSListener(conn, tlsConfig)); err != nil {
        log.Printf("ListenAndServe: %v\n", err)
    }

    return err
}

func newServer(conn net.Listener) (*http.Server) {
    grpcServer := newGrpc()
    gwmux, err := newGateway()
    if err != nil {
        panic(err)
    }

    mux := http.NewServeMux()
    mux.Handle("/", gwmux)
    mux.HandleFunc("/swagger/", serveSwaggerFile)
```

```
    serveSwaggerUI(mux)

    return &http.Server{
        Addr:      EndPoint,
        Handler:   util.GrpcHandlerFunc(grpcServer, mux),
        TLSConfig: tlsConfig,
    }
}

func newGrpc() *grpc.Server {
    creds, err := credentials.NewServerTLSFromFile(CertPemPath, CertKeyPath)
    if err != nil {
        panic(err)
    }

    opts := []grpc.ServerOption{
        grpc.Creds(creds),
    }
    server := grpc.NewServer(opts...)

    pb.RegisterHelloWorldServer(server, NewHelloService())

    return server
}

func newGateway() (http.Handler, error) {
    ctx := context.Background()
    dcreds, err := credentials.NewClientTLSFromFile(CertPemPath, CertServerName)
    if err != nil {
        return nil, err
    }
    dopts := []grpc.DialOption{grpc.WithTransportCredentials(dcreds)}

    gwmux := runtime.NewServeMux()
    if err := pb.RegisterHelloWorldHandlerFromEndpoint(ctx, gwmux, EndPoint, dop
ts); err != nil {
        return nil, err
    }

    return gwmux, nil
}

func serveSwaggerFile(w http.ResponseWriter, r *http.Request) {
      if ! strings.HasSuffix(r.URL.Path, "swagger.json") {
        log.Printf("Not Found: %s", r.URL.Path)
        http.NotFound(w, r)
```

```
        return
    }

    p := strings.TrimPrefix(r.URL.Path, "/swagger/")
    p = path.Join(SwaggerDir, p)

    log.Printf("Serving swagger-file: %s", p)

    http.ServeFile(w, r, p)
}

func serveSwaggerUI(mux *http.ServeMux) {
    fileServer := http.FileServer(&assetfs.AssetFS{
        Asset:    swagger.Asset,
        AssetDir: swagger.AssetDir,
        Prefix:   "third_party/swagger-ui",
    })
    prefix := "/swagger-ui/"
    mux.Handle(prefix, http.StripPrefix(prefix, fileServer))
}
```

测试

访问路径 `https://127.0.0.1:50052/swagger/hello.swagger.json` ，查看输出内容是否为 `hello.swagger.json` 的内容，例如：
[image]

访问路径 `https://127.0.0.1:50052/swagger-ui/` ，查看内容
[image]

小结

至此我们这一章节就完毕了， `Swagger` 和其生态圈十分的丰富，有兴趣研究的小伙伴可以到其官网认真研究

而目前完成的程度也满足了日常工作的需求了，可较自动化的生成 `RESTful Api` 文档，完成与接口对接

参考

示例代码

- grpc-hello-world

gRPC+gRPC Gateway 能不能不用证书？

如果你以前有涉猎过 gRPC+gRPC Gateway 这两个组件，你肯定会遇到这个问题，就是**“为什么非得开 TLS，才能够实现同端口双流量，能不能不开？”**又或是**“我不想用证书就实现这些功能，行不行？”**。我被无数的人问过无数次这些问题，也说服过很多人，但说服归说服，不代表放弃。前年不行，不代表今年不行，在今天我希望分享来龙去脉和具体的实现方式给你。

过去

为什么 h2 不行

因为 `net/http2` 仅支持“h2”标识，而“h2”标识 HTTP/2 必须使用传输层安全性（TLS）的协议，此标识符用于 TLS 应用层协议协商字段以及识别 HTTP/2 over TLS。

简单来讲，也就 `net/http2` 必须使用 TLS 来交互。通俗来讲就要用证书，那么理所当然，也就无法支持非 TLS 的情况了。

寻找 h2c

那这条路不行，我们再想想别的路？那就是 HTTP/2 规范中的“h2c”标识了，”h2c”标识允许通过明文 TCP 运行 HTTP/2 的协议，此标识符用于 HTTP/1.1 升级标头字段以及标识 HTTP/2 over TCP。

但是这条路，早在 2015 年就已经有在 issue 中进行讨论，当时 @bradfitz 明确表示“不打算支持 h2c，对仅支持 TLS 的情况非常满意，一年后再问我一次”，原文回复如下：

> We do not plan to support h2c. I don’t want to receive bug reports from users who get bitten by transparent proxies messing with h2c. Also, until there’s widespread browser support, it’s not interesting. I am also not interested in being the chicken or the egg to get browser support going. I’m very happy with the TLS-only situation, and things like https://LetsEncrypt.org/ will make TLS much easier (and automatic) soon.

> Ask me again in one year.

琢磨其他方式

使用 cmux

基于多路复用器 soheilhy/cmux 的另类实现 Stoakes/grpc-gateway-example。若对 `cmux` 的实现方式感兴趣，还可以看看 《Golang: Run multiple services on one

port》。

使用第三方 h2

- veqryn/h2c

这种属于自己实现了 h2c 的逻辑，以此达到效果。

现在

经过社区的不断讨论，最后在 2018 年 6 月，代表 “h2c” 标志的 `golang.org/x/net/http2/h2c` 标准库正式合并进来，自此我们就可以使用官方标准库（h2c），这个标准库实现了 HTTP/2 的未加密模式，因此我们就可以利用该标准库在同个端口上既提供 HTTP/1.1 又提供 HTTP/2 的功能了。

使用标准库 h2c

```
import (
    ...

    "golang.org/x/net/http2"
    "golang.org/x/net/http2/h2c"
    "google.golang.org/grpc"

    "github.com/grpc-ecosystem/grpc-gateway/runtime"

    pb "github.com/EDDYCJY/go-grpc-example/proto"
)

...

func grpcHandlerFunc(grpcServer *grpc.Server, otherHandler http.Handler) http.Ha
ndler {
    return h2c.NewHandler(http.HandlerFunc(func(w http.ResponseWriter, r *http.R
equest) {
        if r.ProtoMajor == 2 && strings.Contains(r.Header.Get("Content-Type"),
"application/grpc") {
            grpcServer.ServeHTTP(w, r)
        } else {
            otherHandler.ServeHTTP(w, r)
        }
    }), &http2.Server{})
}
```

```
func main() {
    server := grpc.NewServer()

    pb.RegisterSearchServiceServer(server, &SearchService{})

    mux := http.NewServeMux()
    gwmux := runtime.NewServeMux()
    dopts := []grpc.DialOption{grpc.WithInsecure()}

    err := pb.RegisterSearchServiceHandlerFromEndpoint(context.Background(), gwm
ux, "localhost:"+PORT, dopts)
    ...
    mux.Handle("/", gwmux)
    http.ListenAndServe(":"+PORT, grpcHandlerFunc(server, mux))
}
```

我们可以看到关键之处在于调用了 `h2c.NewHandler` 方法进行了特殊处理，`h2c.NewHandler` 会返回一个 `http.handler`，主要的内部逻辑是拦截了所有 `h2c` 流量，然后根据不同的请求流量类型将其劫持并重定向到相应的 `Hander` 中去处理。

验证

HTTP/1.1

```
$ curl -X GET 'http://127.0.0.1:9005/search?request=EDDYCJY'
{"response":"EDDYCJY"}
```

HTTP/2(gRPC)

```
...
func main() {
    conn, err := grpc.Dial(":"+PORT, grpc.WithInsecure())
    ...
    client := pb.NewSearchServiceClient(conn)
    resp, err := client.Search(context.Background(), &pb.SearchRequest{
        Request: "gRPC",
    })
}
```

输出结果：

```
$ go run main.go
2019/06/21 20:04:09 resp: gRPC h2c Server
```

总结

在本文中我介绍了大致的前因后果，且介绍了几种解决方法，我建议你选择官方的 `h2c` 标准库去实现这个功能，也简单。在最后，不管你是否曾经为这个问题烦恼过许久，又或者正在纠结，都希望这篇文章能够帮到你。

参考

- https://github.com/golang/go/issues/13128
- https://github.com/golang/go/issues/14141
- https://github.com/golang/net/commit/c4299a1a0d8524c11563db160fbf9bddbceadb21
- https://go-review.googlesource.com/c/net/+/112997/

map

深入理解 Go map：初始化和访问元素

从本文开始咱们一起探索 Go map 里面的奥妙吧，看看它的内在是怎么构成的，又分别有什么值得留意的地方？

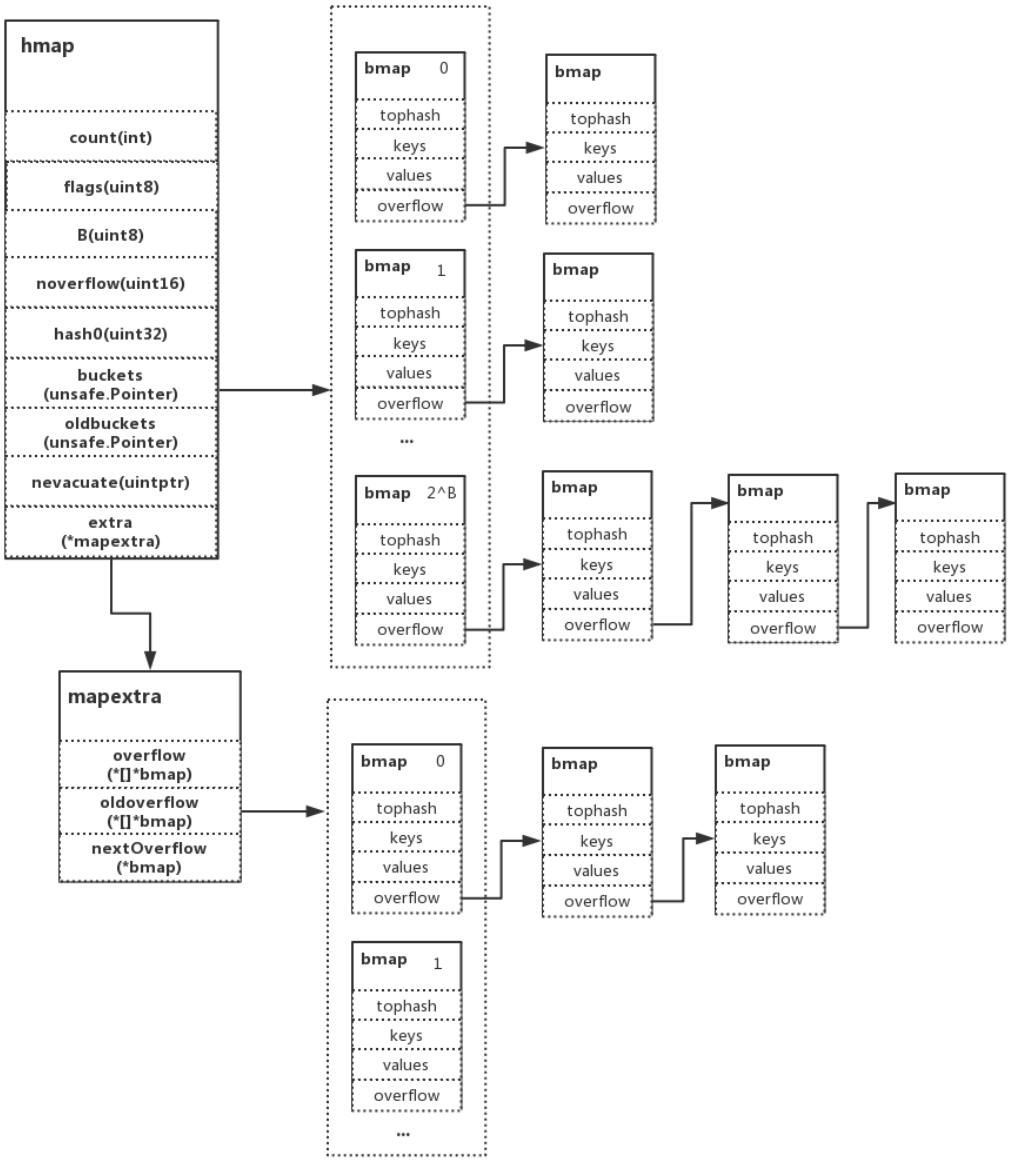
第一篇将探讨**初始化和访问元素**相关板块，咱们带着疑问去学习，例如：

- 初始化的时候会马上分配内存吗？
- 底层数据是如何存储的？
- 底层是如何使用 key 去寻找数据的？
- 底层是用什么方式解决哈希冲突的？
- 数据类型那么多，底层又是怎么处理的呢？

...

数据结构

首先我们一起看看 Go map 的基础数据结构，先有一个大致的印象



hmap

```
type hmap struct {
    count     int
    flags     uint8
    B         uint8
    noverflow uint16
    hash0     uint32
    buckets    unsafe.Pointer
    oldbuckets unsafe.Pointer
    nevacuate  uintptr
    extra *mapextra
```

```
}

type mapextra struct {
    overflow    *[]*bmap
    oldoverflow *[]*bmap
    nextOverflow *bmap
}
```

- count：map 的大小，也就是 len() 的值。代指 map 中的键值对个数
- flags：状态标识，主要是 goroutine 写入和扩容机制的相关状态控制。并发读写的判断条件之一就是该值
- B：桶，最大可容纳的元素数量，值为 **负载因子（默认 6.5）* 2 ^ B**，是 2 的指数
- noverflow：溢出桶的数量
- hash0：哈希因子
- buckets：保存当前桶数据的指针地址（指向一段连续的内存地址，主要存储键值对数据）
- oldbuckets，保存旧桶的指针地址
- nevacuate：迁移进度
- extra：原有 buckets 满载后，会发生扩容动作，在 Go 的机制中使用了增量扩容，如下为细项：
  - `overflow` 为 `hmap.buckets` （当前）溢出桶的指针地址
  - `oldoverflow` 为 `hmap.oldbuckets` （旧）溢出桶的指针地址
  - `nextOverflow` 为空闲溢出桶的指针地址

在这里我们要注意几点，如下：

1. 如果 keys 和 values 都不包含指针并且允许内联的情况下。会将 bucket 标识为不包含指针，使用 extra 存储溢出桶就可以避免 GC 扫描整个 map，节省不必要的开销
2. 在前面有提到，Go 用了增量扩容。而 `buckets` 和 `oldbuckets` 也是与扩容相关的载体，一般情况下只使用 `buckets`，`oldbuckets` 是为空的。但如果正在扩容的话，`oldbuckets` 便不为空，`buckets` 的大小也会改变
3. 当 `hint` 大于 8 时，就会使用 `*mapextra` 做溢出桶。若小于 8，则存储在 buckets 桶中

bmap


```
bucketCntBits = 3
bucketCnt     = 1 << bucketCntBits
...
type bmap struct {
    tophash [bucketCnt]uint8
}
```

- tophash：key 的 hash 值高 8 位
- keys：8 个 key
- values：8 个 value
- overflow：下一个溢出桶的指针地址（当 hash 冲突发生时）

实际 bmap 就是 buckets 中的 bucket，一个 bucket 最多存储 8 个键值对

tophash

tophash 是个长度为 8 的数组，代指桶最大可容纳的键值对为 8。

存储每个元素 hash 值的高 8 位，如果 `tophash [0] <minTopHash` ，则 `tophash [0]` 表示为迁移进度

keys 和 values

在这里我们留意到，存储 k 和 v 的载体并不是用 `k/v/k/v/k/v/k/v` 的模式，而是 `k/k/k/k/v/v/v/v` 的形式去存储。这是为什么呢？

```
map[int64]int8
```

在这个例子中，如果按照 `k/v/k/v/k/v/k/v` 的形式存放的话，虽然每个键值对的值都只占用 1 个字节。但是却需要 7 个填充字节来补齐内存空间。最终就会造成大量的内存“浪费”

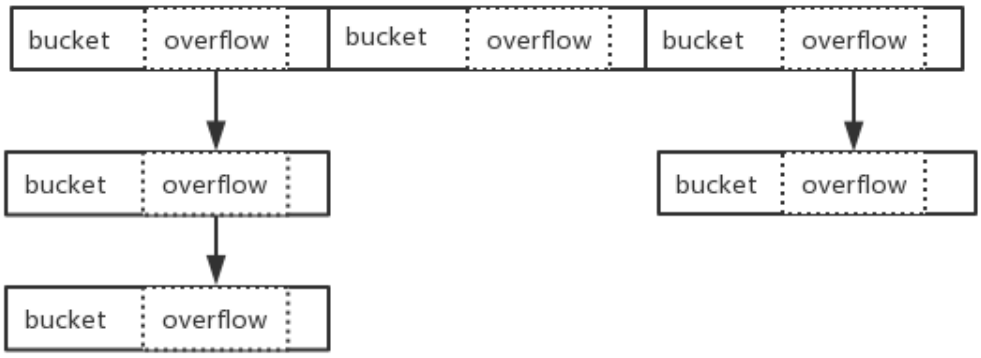
但是如果以 `k/k/k/k/v/v/v/v` 的形式存放的话，就能够解决因对齐所“浪费”的内存空间

因此这部分的拆分主要是考虑到内存对齐的问题，虽然相对会复杂一点，但依然值得如此设计

overflow

可能会有同学疑惑为什么会有溢出桶这个东西？实际上在不存在哈希冲突的情况下，去掉溢出桶，也就是只需要桶、哈希因子、哈希算法。也能实现一个简单的 hash table。但是哈希冲突（碰撞）是不可避免的...

而在 Go map 中当 `hmap.buckets` 满了后，就会使用溢出桶接着存储。我们结合分析可确定 Go 采用的是数组 + 链地址法解决哈希冲突



初始化

用法

```
m := make(map[int32]int32)
```

函数原型

通过阅读源码可得知，初始化方法有好几种。函数原型如下：

```
func makemap_small() *hmap
func makemap64(t *maptype, hint int64, h *hmap) *hmap
func makemap(t *maptype, hint int, h *hmap) *hmap
```

- makemap_small：当 `hint` 小于 8 时，会调用 `makemap_small` 来初始化 hmap。主要差异在于是否会马上初始化 hash table
- makemap64：当 `hint` 类型为 int64 时的特殊转换及校验处理，后续实质调用 `makemap`
- makemap：实现了标准的 map 初始化动作

源码

```
func makemap(t *maptype, hint int, h *hmap) *hmap {
    if hint < 0 || hint > int(maxSliceCap(t.bucket.size)) {
        hint = 0
    }

    if h == nil {
        h = new(hmap)
    }
    h.hash0 = fastrand()

    B := uint8(0)
    for overLoadFactor(hint, B) {
        B++
    }
    h.B = B

    if h.B != 0 {
        var nextOverflow *bmap
```

```
        h.buckets, nextOverflow = makeBucketArray(t, h.B, nil)
        if nextOverflow != nil {
            h.extra = new(mapextra)
            h.extra.nextOverflow = nextOverflow
        }
    }

    return h
}
```

- 根据传入的 `bucket` 类型，获取其类型能够申请的最大容量大小。并对其长度 `make(map[k]v, hint)` 进行边界值检验
- 初始化 hmap
- 初始化哈希因子
- 根据传入的 `hint` ，计算一个可以放下 `hint` 个元素的桶 `B` 的最小值
- 分配并初始化 hash table。如果 `B` 为 0 将在后续懒惰分配桶，大于 0 则会马上进行分配
- 返回初始化完毕的 hmap

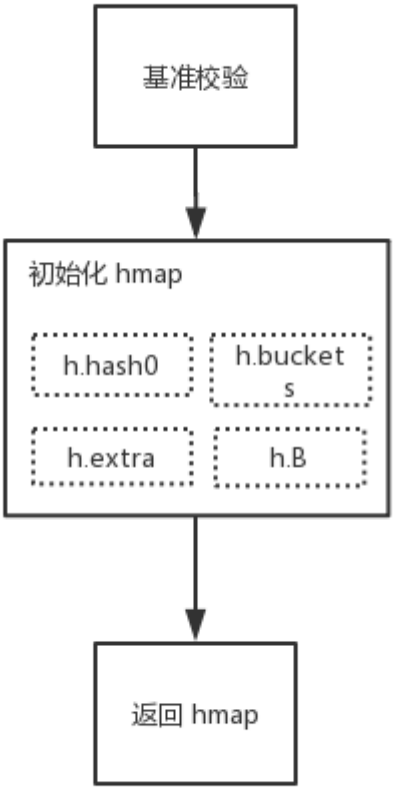
在这里可以注意到，（当 `hint` 大于等于 8 ）第一次初始化 map 时，就会通过调用 `makeBucketArray` 对 buckets 进行分配。因此我们常常会说，在初始化时指定一个适当大小的容量。能够提升性能。

若该容量过少，而新增的键值对又很多。就会导致频繁的分配 buckets，进行扩容迁移等 rehash 动作。最终结果就是性能直接的下降（敲黑板）

而当 `hint` 小于 8 时，这种问题**相对**就不会凸显的太明显，如下：

```
func makemap_small() *hmap {
    h := new(hmap)
    h.hash0 = fastrand()
    return h
}
```

图示



访问

用法

```
v := m[i]
v, ok := m[i]
```

函数原型

在实现 map 元素访问上有好几种方法，主要是包含针对 32/64 位、string 类型的特殊处理，总的函数原型如下：

```
mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
mapaccess2(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, bool)

mapaccessK(t *maptype, h *hmap, key unsafe.Pointer) (unsafe.Pointer, unsafe.Pointer)
```

```
mapaccess1_fat(t *maptype, h *hmap, key, zero unsafe.Pointer) unsafe.Pointer
mapaccess2_fat(t *maptype, h *hmap, key, zero unsafe.Pointer) (unsafe.Pointer, b
ool)

mapaccess1_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer
mapaccess2_fast32(t *maptype, h *hmap, key uint32) (unsafe.Pointer, bool)
mapassign_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer
mapassign_fast32ptr(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer

mapaccess1_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer
...

mapaccess1_faststr(t *maptype, h *hmap, ky string) unsafe.Pointer
...
```

- mapaccess1：返回 `h[key]` 的指针地址，如果键不在 `map` 中，将返回对应类型的零值
- mapaccess2：返回 `h[key]` 的指针地址，如果键不在 `map` 中，将返回零值和布尔值用于判断

源码

```
func mapaccess1(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    ...
    if h == nil || h.count == 0 {
        return unsafe.Pointer(&zeroVal[0])
    }
    if h.flags&hashWriting != 0 {
        throw("concurrent map read and map write")
    }
    alg := t.key.alg
    hash := alg.hash(key, uintptr(h.hash0))
    m := bucketMask(h.B)
    b := (*bmap)(add(h.buckets, (hash&m)*uintptr(t.bucketsize)))
    if c := h.oldbuckets; c != nil {
        if !h.sameSizeGrow() {
            // There used to be half as many buckets; mask down one more power o
f two.
            m >>= 1
        }
        oldb := (*bmap)(add(c, (hash&m)*uintptr(t.bucketsize)))
        if !evacuated(oldb) {
```

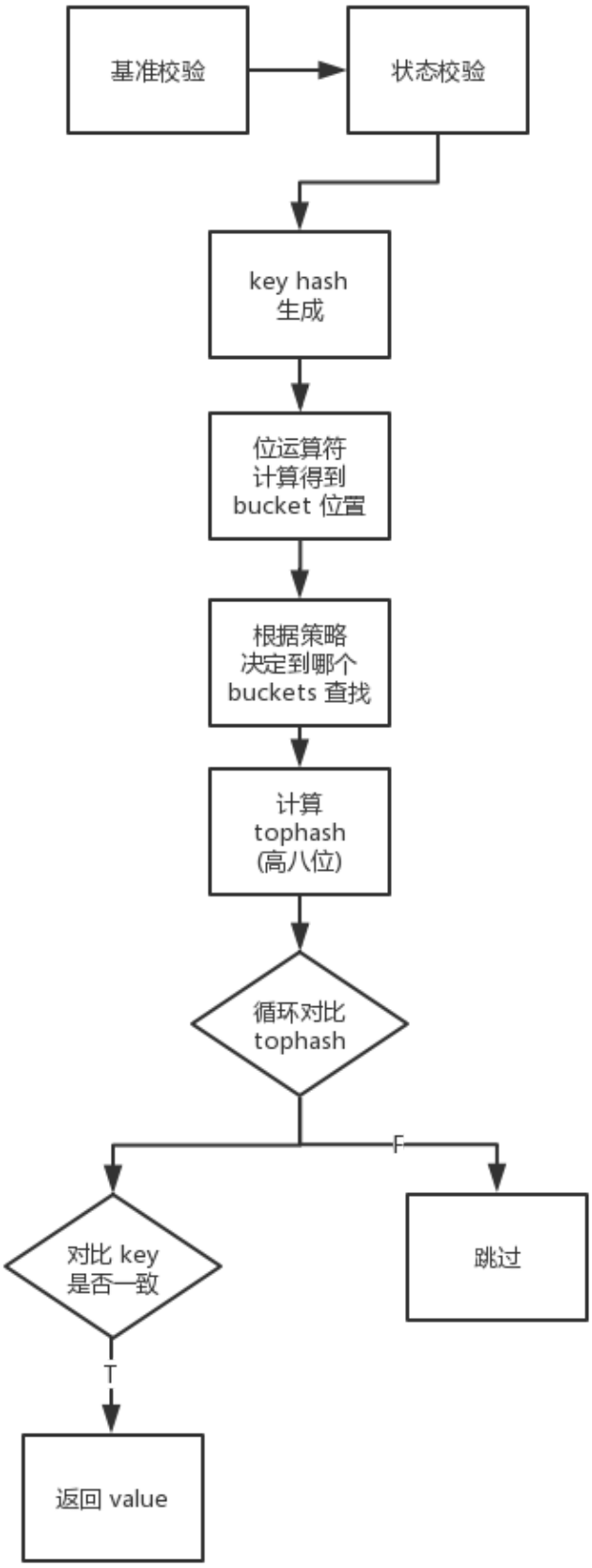
```
            b = oldb
        }
    }
    top := tophash(hash)
    for ; b != nil; b = b.overflow(t) {
        for i := uintptr(0); i < bucketCnt; i++ {
            if b.tophash[i] != top {
                continue
            }
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            if t.indirectkey {
                k = *((*unsafe.Pointer)(k))
            }
            if alg.equal(key, k) {
                v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysi
ze)+i*uintptr(t.valuesize))
                if t.indirectvalue {
                    v = *((*unsafe.Pointer)(v))
                }
                return v
            }
        }
    }
    return unsafe.Pointer(&zeroVal[0])
}
```

- 判断 map 是否为 nil，长度是否为 0。若是则返回零值
- 判断当前是否并发读写 map，若是则抛出异常
- 根据 key 的不同类型调用不同的 hash 方法计算得出 hash 值
- 确定 key 在哪一个 bucket 中，并得到其位置
- 判断是否正在发生扩容（h.oldbuckets 是否为 nil），若正在扩容，则到老的 buckets 中查找（因为 buckets 中可能还没有值，搬迁未完成），若该 bucket 已经搬迁完毕。则到 buckets 中继续查找
- 计算 hash 的 tophash 值（高八位）
- 根据计算出来的 tophash，依次循环对比 buckets 的 tophash 值（快速试错）
- 如果 tophash 匹配成功，则计算 key 的所在位置，正式完整的对比两个 key 是否一致
- 若查找成功并返回，若不存在，则返回零值

在上述步骤三中，提到了根据不同的类型计算出 hash 值，另外会计算出 hash 值的高八位和低八位。低八位会作为 bucket index，作用是用于找到 key 所在的 bucket。而高八位会存储在 bmap tophash 中

其主要作用是在上述步骤七中进行迭代快速定位。这样子可以提高性能，而不是一开始就直接用 key 进行一致性对比

图示


基准校验
状态校验
key hash
生成
位运算符
计算得到
bucket 位置
根据策略
决定到哪个
buckets 查找
计算
tophash
(高八位)
循环对比
tophash
F
对比 key
是否一致
跳过
T
返回 value

总结

在本章节，我们介绍了 map 类型的以下知识点：

- map 的基础数据结构
- 初始化 map
- 访问 map

从阅读源码中，得知 Go 本身**对于一些不同大小、不同类型的属性，包括哈希方法都有编写特定方法**去运行。总的来说，这块的设计隐含较多的思路，有不少点值得细细品尝 :)

注：本文基于 Go 1.11.5

深入理解 Go map：赋值和扩容迁移

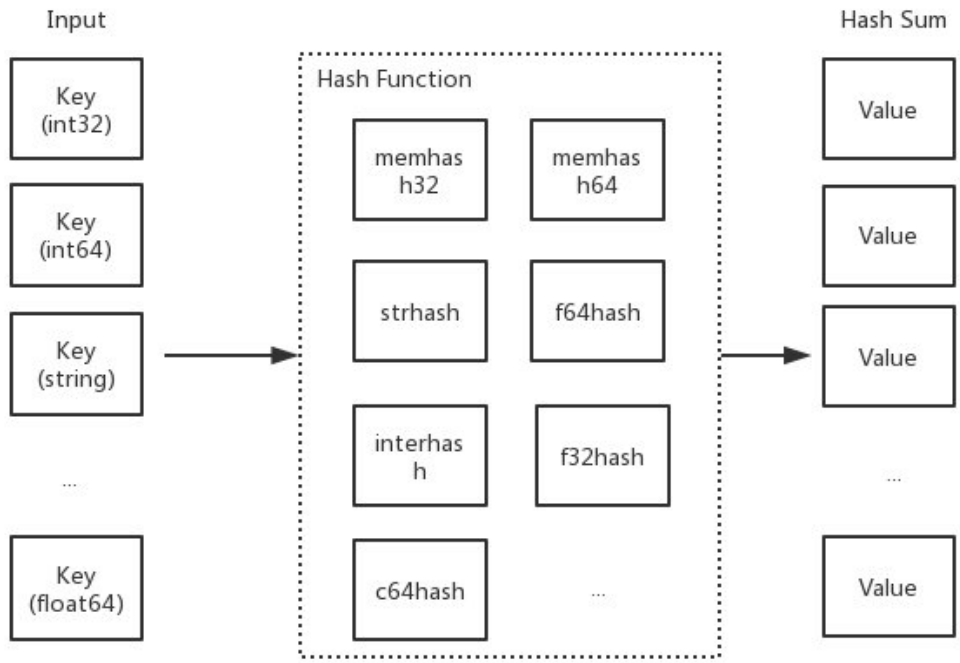
概要

在 上一章节中，数据结构小节里讲解了大量基础字段，可能你会疑惑需要 #&（！……#（！￥！ 来干嘛？接下来我们一起简单了解一下基础概念。再开始研讨今天文章的重点内容。我相信这样你能更好的读懂这篇文章

哈希函数

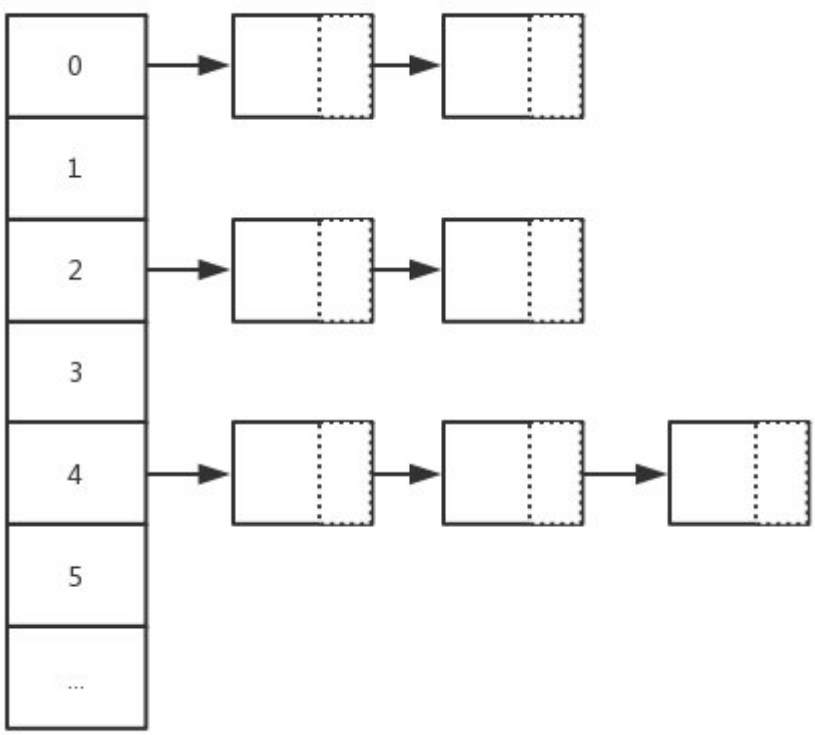
哈希函数，又称散列算法、散列函数。主要作用是通过特定算法将数据根据一定规则组合重新生成得到一个**散列值**

而在哈希表中，其生成的散列值常用于寻找其键映射到哈希桶上。而一个好的哈希函数，应当尽量少的出现哈希冲突，以此保证操作哈希表的时间复杂度（但是哈希冲突在目前来讲，是无法避免的。我们需要“解决”它）



链地址法

在哈希操作中，相当核心的一个处理动作就是“哈希冲突”的解决。而在 Go map 中采用的就是“链地址法“ 去解决哈希冲突，又称“拉链法”。其主要做法是数组 + 链表的数据结构，其溢出节点的存储内存都是动态申请的，因此相对更灵活。而每一个元素都是一个链表。如下图：



桶/溢出桶

```
type hmap struct {
    ...
    buckets    unsafe.Pointer
    ...
    extra *mapextra
}

type mapextra struct {
    overflow    *[]*bmap
    oldoverflow *[]*bmap
    nextOverflow *bmap
}
```

在上章节中，我们介绍了 Go map 中的桶和溢出桶的概念，在其桶中只能存储 8 个键值对元素。当超过 8 个时，将会使用溢出桶进行存储或进行扩容

你可能会有疑问，hint 大于 8 又会怎么样？答案很明显，性能问题，其时间复杂度改变（也就是执行效率出现问题）

前言

概要复习的差不多后，接下来我们将一同研讨 Go map 的另外三个核心行为：赋值、扩容、迁移。正式开始我们的研讨之旅吧 ： ）

赋值

```
m := make(map[int32]string)
m[0] = "EDDYCJY"
```

函数原型

在 map 的赋值动作中，依旧是针对 32/64 位、string、pointer 类型有不同的转换处理，总的函数原型如下：

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
func mapaccess1_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer
func mapaccess2_fast32(t *maptype, h *hmap, key uint32) (unsafe.Pointer, bool)
func mapassign_fast32(t *maptype, h *hmap, key uint32) unsafe.Pointer
func mapassign_fast32ptr(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer

func mapaccess1_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer
func mapaccess2_fast64(t *maptype, h *hmap, key uint64) (unsafe.Pointer, bool)
func mapassign_fast64(t *maptype, h *hmap, key uint64) unsafe.Pointer
func mapassign_fast64ptr(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer
func mapaccess1_faststr(t *maptype, h *hmap, ky string) unsafe.Pointer
func mapaccess2_faststr(t *maptype, h *hmap, ky string) (unsafe.Pointer, bool)
func mapassign_faststr(t *maptype, h *hmap, s string) unsafe.Pointer
...
```

接下来我们将分成几个部分去看看底层在赋值的时候，都做了些什么处理？

源码

第一阶段：校验和初始化

```
func mapassign(t *maptype, h *hmap, key unsafe.Pointer) unsafe.Pointer {
    if h == nil {
```

```
        panic(plainError("assignment to entry in nil map"))
    }
    ...
    if h.flags&hashWriting != 0 {
        throw("concurrent map writes")
    }
    alg := t.key.alg
    hash := alg.hash(key, uintptr(h.hash0))

    h.flags |= hashWriting

    if h.buckets == nil {
        h.buckets = newobject(t.bucket) // newarray(t.bucket, 1)
    }
    ...
}
```

- 判断 hmap 是否已经初始化（是否为 nil）
- 判断是否并发读写 map，若是则抛出异常
- 根据 key 的不同类型调用不同的 hash 方法计算得出 hash 值
- 设置 flags 标志位，表示有一个 goroutine 正在写入数据。因为 `alg.hash` 有可能出现 `panic` 导致异常
- 判断 buckets 是否为 nil，若是则调用 `newobject` 根据当前 bucket 大小进行分配（例如：上章节提到的 `makemap_small` 方法，就在初始化时没有初始 buckets，那么它在第一次赋值时就会对 buckets 分配）

第二阶段：寻找可插入位和更新既有值

```
...
again:
    bucket := hash & bucketMask(h.B)
    if h.growing() {
        growWork(t, h, bucket)
    }
    b := (*bmap)(unsafe.Pointer(uintptr(h.buckets) + bucket*uintptr(t.bucketsiz
e)))
    top := tophash(hash)

    var inserti *uint8
    var insertk unsafe.Pointer
    var val unsafe.Pointer
    for {
```

```
        for i := uintptr(0); i < bucketCnt; i++ {
            if b.tophash[i] != top {
                if b.tophash[i] == empty && 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.valuesize))
                }
                continue
            }
            k := add(unsafe.Pointer(b), dataOffset+i*uintptr(t.keysize))
            if t.indirectkey {
                k = *((*unsafe.Pointer)(k))
            }
            if !alg.equal(key, k) {
                continue
            }
            // already have a mapping for key. Update it.
            if t.needkeyupdate {
                typedmemmove(t.key, k, key)
            }
            val = add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)
+i*uintptr(t.valuesize))
            goto done
        }
        ovf := b.overflow(t)
        if ovf == nil {
            break
        }
        b = ovf
    }

    if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets
(h.noverflow, h.B)) {
        hashGrow(t, h)
        goto again // Growing the table invalidates everything, so try again
    }
    ...
```

- 根据低八位计算得到 bucket 的内存地址，并判断是否正在扩容，若正在扩容中则先迁移再接着处理
- 计算并得到 bucket 的 bmap 指针地址，计算 key hash 高八位用于查找 Key

- 迭代 buckets 中的每一个 bucket（共 8 个），对比 `bucket.tophash` 与 top（高八位）是否一致
- 若不一致，判断是否为空槽。若是空槽（有两种情况，第一种是**没有插入过**。第二种是**插入后被删除**），则把该位置标识为可插入 tophash 位置。注意，这里就是第一个可以插入数据的地方
- 若 key 与当前 k 不匹配则跳过。但若是匹配（也就是原本已经存在），则进行更新。最后跳出并返回 value 的内存地址
- 判断是否迭代完毕，若是则结束迭代 buckets 并更新当前桶位置
- 若满足三个条件：触发最大 `LoadFactor` 、存在过多溢出桶 `overflow buckets` 、没有正在进行扩容。就会进行扩容动作（以确保后续的动作）

总的来讲，这一块逻辑做了两件大事，第一是**寻找空位，将位置其记录在案，用于后续的插入动作**。第二是**判断 Key 是否已经存在哈希表中，存在则进行更新**。而若是第二种场景，更新完毕后就会进行收尾动作，第一种将继续执行下述的代码

第三阶段：申请新的插入位和插入新值

```
    ...
    if inserti == nil {
        newb := h.newoverflow(t, b)
        inserti = &newb.tophash[0]
        insertk = add(unsafe.Pointer(newb), dataOffset)
        val = add(insertk, bucketCnt*uintptr(t.keysize))
    }

    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)
    *inserti = top
    h.count++

done:
    ...
    return val
```

经过前面迭代寻找动作，若没有找到可插入的位置，意味着当前的所有桶都满了，将重新分配一个新溢出桶用于插入动作。最后再在上一步申请的新插入位置，存储键值对，返回该值的内存地址

第四阶段：写入

但是这里又疑惑了？最后为什么是返回内存地址。这是因为隐藏的最后一步写入动作（将值拷贝到指定内存区域）是通过底层汇编配合来完成的，在 runtime 中只完成了绝大部分的动作

```
func main() {
    m := make(map[int32]int32)
    m[0] = 6666666
}
```

对应的汇编部分：

```
...
0x0099 00153 (test.go:6)    CALL    runtime.mapassign_fast32(SB)
0x009e 00158 (test.go:6)    PCDATA    $2, $2
0x009e 00158 (test.go:6)    MOVQ    24(SP), AX
0x00a3 00163 (test.go:6)    PCDATA    $2, $0
0x00a3 00163 (test.go:6)    MOVL    $6666666, (AX)
```

这里分为了几个部位，主要是调用 `mapassign` 函数和拿到值存放的内存地址，再将 6666666 这个值存放进该内存地址中。另外我们看到 `PCDATA` 指令，主要是包含一些垃圾回收的信息，由编译器产生

小结

通过前面几个阶段的分析，我们可梳理出一些要点。例如：

- 不同类型对应哈希函数不一样
- 高八位用于定位 bucket
- 低八位用于定位 key，快速试错后再进行完整对比
- buckets/overflow buckets 遍历
- 可插入位的处理
- 最终写入动作与底层汇编的交互

扩容

在所有动作中，扩容规则是大家较关注的点，也是赋值里非常重要的一环。因此咱们将这节拉出来，对这块细节进行研讨

什么时候扩容

```
if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets(h.noverflow, h.B)) {
    hashGrow(t, h)
    goto again
}
```

在特定条件的情况下且当前没有正在进行扩容动作（以判断 `hmap.oldbuckets != nil` 为基准）。哈希表在赋值、删除的动作下会触发扩容行为，条件如下：

- 触发 `load factor` 的最大值，负载因子已达到当前界限
- 溢出桶 `overflow buckets` 过多

什么时候受影响

那么什么情况下会对这两个“值”有影响呢？如下：

1. 负载因子 `load factor`，用途是评估哈希表当前的时间复杂度，其与哈希表当前包含的键值对数、桶数量等相关。如果负载因子越大，则说明空间使用率越高，但产生哈希冲突的可能性更高。而负载因子越小，说明空间使用率低，产生哈希冲突的可能性更低
2. 溢出桶 `overflow buckets` 的判定与 buckets 总数和 overflow buckets 总数相关联

因子关系

loadFactor	%overflow	bytes/entry	hitprobe	missprobe
4.00	2.13	20.77	3.00	4.00
4.50	4.05	17.30	3.25	4.50
5.00	6.85	14.77	3.50	5.00
5.50	10.55	12.94	3.75	5.50
6.00	15.27	11.67	4.00	6.00
6.50	20.90	10.79	4.25	6.50
7.00	27.14	10.15	4.50	7.00

- loadFactor：负载因子
- %overflow：溢出率，具有溢出桶 `overflow buckets` 的桶的百分比
- bytes/entry：每个键值对所的字节数开销
- hitprobe：查找存在的 key 时，平均需要检索的条目数量
- missprobe：查找不存在的 key 时，平均需要检索的条目数量

这一组数据能够体现出不同的负载因子会给哈希表的动作带来怎么样的影响。而在上一章节我们有提到默认的负载因子是 6.5 (loadFactorNum/loadFactorDen)，可以看出来是经过测试后取出的一个比较合理的因子。能够较好的影响哈希表的扩容动作的时机

源码剖析

```
func hashGrow(t *maptype, h *hmap) {
    bigger := uint8(1)
    if !overLoadFactor(h.count+1, h.B) {
        bigger = 0
        h.flags |= sameSizeGrow
    }
    oldbuckets := h.buckets
    newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)
    ...
    h.oldbuckets = oldbuckets
    h.buckets = newbuckets
    h.nevacuate = 0
    h.noverflow = 0

    if h.extra != nil && h.extra.overflow != nil {
        if h.extra.oldoverflow != nil {
            throw("oldoverflow is not nil")
        }
        h.extra.oldoverflow = h.extra.overflow
        h.extra.overflow = nil
    }
    if nextOverflow != nil {
        if h.extra == nil {
            h.extra = new(mapextra)
        }
        h.extra.nextOverflow = nextOverflow
    }

    // the actual copying of the hash table data is done incrementally
    // by growWork() and evacuate().
}
```

第一阶段：确定扩容容量规则

在上小节有讲到扩容的依据有两种，在 `hashGrow` 开头就进行了划分。如下：

```
if !overLoadFactor(h.count+1, h.B) {
    bigger = 0
    h.flags |= sameSizeGrow
}
```

若不是负载因子 `load factor` 超过当前界限，也就是属于溢出桶 `overflow buckets` 过多的情况。因此本次扩容规则将是 `sameSizeGrow`，即是**不改变大小的扩容动作**。那要是前者的情况呢？

```
bigger := uint8(1)
...
newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)
```

结合代码分析可得出，若是负载因子 `load factor` 达到当前界限，将会动态扩容**当前大小的两倍**作为其新容量大小

第二阶段：初始化、交换新旧 桶/溢出桶

主要是针对扩容的相关数据**前置处理**，涉及 buckets/oldbuckets、overflow/oldoverflow 之类与存储相关的字段

```
...
oldbuckets := h.buckets
newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)

flags := h.flags &^ (iterator | oldIterator)
if h.flags&iterator != 0 {
    flags |= oldIterator
}

h.B += bigger
...
h.noverflow = 0

if h.extra != nil && h.extra.overflow != nil {
    ...
    h.extra.oldoverflow = h.extra.overflow
    h.extra.overflow = nil
}
if nextOverflow != nil {
```

```
    ...
    h.extra.nextOverflow = nextOverflow
}
```

这里注意到这段代码：`newbuckets, nextOverflow := makeBucketArray(t, h.B+bigger, nil)`。第一反应是扩容的时候就马上申请并初始化内存了吗？假设涉及大量的内存分配，那挺耗费性能的...

然而并不，内部只会先进行预分配，当使用的时候才会真正的去初始化

第三阶段：扩容

在源码中，发现第三阶段的流转并没有显式展示。这是因为流转由底层去做控制了。但通过分析代码和注释，可得知由第三阶段涉及 `growWork` 和 `evacuate` 方法。如下：

```
func growWork(t *maptype, h *hmap, bucket uintptr) {
    evacuate(t, h, bucket&h.oldbucketmask())

    if h.growing() {
        evacuate(t, h, h.nevacuate)
    }
}
```

在该方法中，主要是两个 `evacuate` 函数的调用。他们在调用上又分别有什么区别呢？如下：

- evacuate(t, h, bucket&h.oldbucketmask()): 将 oldbucket 中的元素迁移 rehash 到扩容后的新 bucket
- evacuate(t, h, h.nevacuate): 如果当前正在进行扩容，则再进行多一次迁移

另外，在执行扩容动作的时候，可以发现都是以 bucket/oldbucket 为单位的，而不是传统的 buckets/oldbuckets。再结合代码分析，可得知在 Go map 中**扩容是采取增量扩容的方式，并非一步到位**

为什么是增量扩容？

如果是全量扩容的话，那问题就来了。假设当前 hmap 的容量比较大，直接全量扩容的话，就会导致扩容要花费大量的时间和内存，导致系统卡顿，最直观的表现就是慢。显然，不能这么做

而增量扩容，就可以解决这个问题。它通过每一次的 map 操作行为去分摊总的一次性动作。因此有了 buckets/oldbuckets 的设计，它是逐步完成的，并且会在扩容完毕后才进行清空

小结

通过前面三个阶段的分析，可以得知扩容的大致过程。我们阶段性总结一下。主要如下：

- 根据需扩容的原因不同（overLoadFactor/tooManyOverflowBuckets），分为两类容量规则方向，为等量扩容（不改变原有大小）或双倍扩容
- 新申请的扩容空间（newbuckets/newoverflow）都是预分配，等真正使用的时候才会初始化
- 扩容完毕后（预分配），不会马上就进行迁移。而是采取**增量扩容**的方式，当有访问到具体 bukcet 时，才会逐渐的进行迁移（将 oldbucket 迁移到 bucket）

这时候又想到，既然迁移是逐步进行的。那如果在途中又要扩容了，怎么办？

```
again:
    bucket := hash & bucketMask(h.B)
    ...
    if !h.growing() && (overLoadFactor(h.count+1, h.B) || tooManyOverflowBuckets
(h.noverflow, h.B)) {
        hashGrow(t, h)
        goto again
    }
```

在这里注意到 `goto again` 语句，结合上下文可得若正在进行扩容，就会不断地进行迁移。待迁移完毕后才会开始进行下一次的扩容动作

迁移

在扩容的完整闭环中，包含着迁移的动作，又称“搬迁”。因此我们继续深入研究 `evacuate` 函数。接下来一起打开迁移世界的大门。如下：

```
type evacDst struct {
    b *bmap
    i int
    k unsafe.Pointer
    v unsafe.Pointer
}
```

`evacDst` 是迁移中的基础数据结构，其包含如下字段：

- b: 当前目标桶
- i: 当前目标桶存储的键值对数量
- k: 指向当前 key 的内存地址

- v: 指向当前 value 的内存地址

```
func evacuate(t *maptype, h *hmap, oldbucket uintptr) {
    b := (*bmap)(add(h.oldbuckets, oldbucket*uintptr(t.bucketsize)))
    newbit := h.noldbuckets()
    if !evacuated(b) {
        var xy [2]evacDst
        x := &xy[0]
        x.b = (*bmap)(add(h.buckets, oldbucket*uintptr(t.bucketsize)))
        x.k = add(unsafe.Pointer(x.b), dataOffset)
        x.v = add(x.k, bucketCnt*uintptr(t.keysize))

        if !h.sameSizeGrow() {
            y := &xy[1]
            y.b = (*bmap)(add(h.buckets, (oldbucket+newbit)*uintptr(t.bucketsiz
e)))
            y.k = add(unsafe.Pointer(y.b), dataOffset)
            y.v = add(y.k, bucketCnt*uintptr(t.keysize))
        }

        for ; b != nil; b = b.overflow(t) {
            ...
        }

        if h.flags&oldIterator == 0 && t.bucket.kind&kindNoPointers == 0 {
            b := add(h.oldbuckets, oldbucket*uintptr(t.bucketsize))
            ptr := add(b, dataOffset)
            n := uintptr(t.bucketsize) - dataOffset
            memclrHasPointers(ptr, n)
        }
    }

    if oldbucket == h.nevacuate {
        advanceEvacuationMark(h, t, newbit)
    }
}
```

- 计算并得到 oldbucket 的 bmap 指针地址
- 计算 hmap 在增长之前的桶数量
- 判断当前的迁移（搬迁）状态，以便流转后续的操作。若没有正在进行迁移 `!evacuated(b)` ，则根据扩容的规则的不同，当规则为等量扩容 `sameSizeGrow` 时，只使用一个 `evacDst` 桶用于分流。而为双倍扩容时，就会使用两个 `evacDst` 进行分流操作

- 当分流完毕后，需要迁移的数据都会通过 `typedmemmove` 函数迁移到指定的目标桶上
- 若当前不存在 flags 使用标志、使用 oldbucket 迭代器、bucket 不为指针类型。则取消链接溢出桶、清除键值
- 在最后 `advanceEvacuationMark` 函数中会对迁移进度 `hmap.nevacuate` 进行累积计数，并调用 `bucketEvacuated` 对旧桶 oldbuckets 进行不断的迁移。直至全部迁移完毕。那么也就表示扩容完毕了，会对 `hmap.oldbuckets` 和 `h.extra.oldoverflow` 进行清空

总的来讲，就是计算得到所需数据的位置。再根据当前的迁移状态、扩容规则进行数据分流迁移。结束后进行清理，促进 GC 的回收

总结

在本章节我们主要研讨了 Go map 的几个核心动作，分别是：“赋值、扩容、迁移”。而通过本次的阅读，我们能够更进一步的认识到一些要点，例如：

- 赋值的时候会触发扩容吗？
- 负载因子是什么？过高会带来什么问题？它的变动会对哈希表操作带来什么影响吗？
- 溢出桶越多会带来什么问题？
- 是否要扩容的基准条件是什么？
- 扩容的容量规则是怎么样的？
- 扩容的步骤是怎么样的？涉及到了哪些数据结构？
- 扩容是一次性扩容还是增量扩容？
- 正在扩容的时候又要扩容怎么办？
- 扩容时的迁移分流动作是怎么样的？
- 在扩容动作中，底层汇编承担了什么角色？做了什么事？
- 在 buckets/overflow buckets 中寻找时，是如何“快速”定位值的？低八位、高八位的用途？
- 空槽有可能出现在任意位置吗？假设已经没有空槽了，但是又有新值要插入，底层会怎么处理

最后希望你通过本文的阅读，能更清楚地了解到 Go map 是怎么样运作的 ：）

为什么遍历 Go map 是无序的

有的小伙伴没留意过 Go map 输出顺序，以为它是稳定的有序的；有的小伙伴知道是无序的，但却不知道为什么？有的却理解错误？今天我们将通过本文，揭开 `for range map` 的“神秘”面纱，看看它内部实现到底是怎么样的，输出顺序到底是怎么样？

前言

```
func main() {
    m := make(map[int32]string)
    m[0] = "EDDYCJY1"
    m[1] = "EDDYCJY2"
    m[2] = "EDDYCJY3"
    m[3] = "EDDYCJY4"
    m[4] = "EDDYCJY5"

    for k, v := range m {
        log.Printf("k: %v, v: %v", k, v)
    }
}
```

假设运行这段代码，输出结果是按顺序？还是无序输出呢？

```
2019/04/03 23:27:29 k: 3, v: EDDYCJY4
2019/04/03 23:27:29 k: 4, v: EDDYCJY5
2019/04/03 23:27:29 k: 0, v: EDDYCJY1
2019/04/03 23:27:29 k: 1, v: EDDYCJY2
2019/04/03 23:27:29 k: 2, v: EDDYCJY3
```

从输出结果上来讲，是非固定顺序输出的，也就是每次都不一样（标题也讲了）。但这是为什么呢？

首先**建议你先自己想想原因**。其次我在面试时听过一些说法。有人说因为是哈希的所以就是无（乱）序等等说法。当时我是有点 ？？？

这也是这篇文章出现的原因，希望大家可以一起研讨一下，理清这个问题 ：）

看一下汇编

```
    ...
    0x009b 00155 (main.go:11)    LEAQ    type.map[int32]string(SB), AX
```

```
    0x00a2 00162 (main.go:11)    PCDATA    $2, $0
    0x00a2 00162 (main.go:11)    MOVQ    AX, (SP)
    0x00a6 00166 (main.go:11)    PCDATA    $2, $2
    0x00a6 00166 (main.go:11)    LEAQ    ""..autotmp_3+24(SP), AX
    0x00ab 00171 (main.go:11)    PCDATA    $2, $0
    0x00ab 00171 (main.go:11)    MOVQ    AX, 8(SP)
    0x00b0 00176 (main.go:11)    PCDATA    $2, $2
    0x00b0 00176 (main.go:11)    LEAQ    ""..autotmp_2+72(SP), AX
    0x00b5 00181 (main.go:11)    PCDATA    $2, $0
    0x00b5 00181 (main.go:11)    MOVQ    AX, 16(SP)
    0x00ba 00186 (main.go:11)    CALL    runtime.mapiterinit(SB)
    0x00bf 00191 (main.go:11)    JMP    207
    0x00c1 00193 (main.go:11)    PCDATA    $2, $2
    0x00c1 00193 (main.go:11)    LEAQ    ""..autotmp_2+72(SP), AX
    0x00c6 00198 (main.go:11)    PCDATA    $2, $0
    0x00c6 00198 (main.go:11)    MOVQ    AX, (SP)
    0x00ca 00202 (main.go:11)    CALL    runtime.mapiternext(SB)
    0x00cf 00207 (main.go:11)    CMPQ    ""..autotmp_2+72(SP), $0
    0x00d5 00213 (main.go:11)    JNE    193
    ...
```

我们大致看一下整体过程，重点处理 Go map 循环迭代的是两个 runtime 方法，如下：

- runtime.mapiterinit
- runtime.mapiternext

但你可能会想，明明用的是 `for range` 进行循环迭代，怎么出现了这两个函数，怎么回事？

看一下转换后

```
var hiter map_iteration_struct
for mapiterinit(type, range, &hiter); hiter.key != nil; mapiternext(&hiter) {
    index_temp = *hiter.key
    value_temp = *hiter.val
    index = index_temp
    value = value_temp
    original body
}
```

实际上编译器对于 slice 和 map 的循环迭代有不同的实现方式，并不是 `for` 一扔就完事了，还做了一些附加动作进行处理。而上述代码就是 `for range map` 在编译器展开后的伪实现

看一下源码

runtime.mapiterinit

```
func mapiterinit(t *maptype, h *hmap, it *hiter) {
    ...
    it.t = t
    it.h = h
    it.B = h.B
    it.buckets = h.buckets
    if t.bucket.kind&kindNoPointers != 0 {
        h.createOverflow()
        it.overflow = h.extra.overflow
        it.oldoverflow = h.extra.oldoverflow
    }

    r := uintptr(fastrand())
    if h.B > 31-bucketCntBits {
        r += uintptr(fastrand()) << 31
    }
    it.startBucket = r & bucketMask(h.B)
    it.offset = uint8(r >> h.B & (bucketCnt - 1))
    it.bucket = it.startBucket
    ...

    mapiternext(it)
}
```

通过对 `mapiterinit` 方法阅读，可得知其主要用途是在 map 进行遍历迭代时**进行初始化动作**。共有三个形参，用于读取当前哈希表的类型信息、当前哈希表的存储信息和当前遍历迭代的数据

为什么

咱们关注到源码中 `fastrand` 的部分，这个方法名，是不是迷之眼熟。没错，它是一个生成随机数的方法。再看看上下文：

```
...
// decide where to start
r := uintptr(fastrand())
if h.B > 31-bucketCntBits {
    r += uintptr(fastrand()) << 31
}
it.startBucket = r & bucketMask(h.B)
```

```
it.offset = uint8(r >> h.B & (bucketCnt - 1))

// iterator state
it.bucket = it.startBucket
```

在这段代码中，它生成了随机数。用于决定从哪里开始循环迭代。更具体的话就是根据随机数，选择一个桶位置作为起始点进行遍历迭代

因此每次重新 `for range map` ，你见到的结果都是不一样的。那是因为它的起始位置根本就不固定！

runtime.mapiternext

```
func mapiternext(it *hiter) {
    ...
    for ; i < bucketCnt; i++ {
        ...
        k := add(unsafe.Pointer(b), dataOffset+uintptr(offi)*uintptr(t.keysize))
        v := add(unsafe.Pointer(b), dataOffset+bucketCnt*uintptr(t.keysize)+uint
ptr(offi)*uintptr(t.valuesize))
        ...
        if (b.tophash[offi] != evacuatedX && b.tophash[offi] != evacuatedY) ||
            !(t.reflexivekey || alg.equal(k, k)) {
            ...
            it.key = k
            it.value = v
        } else {
            rk, rv := mapaccessK(t, h, k)
            if rk == nil {
                continue // key has been deleted
            }
            it.key = rk
            it.value = rv
        }
        it.bucket = bucket
        if it.bptr != b {
            it.bptr = b
        }
        it.i = i + 1
        it.checkBucket = checkBucket
        return
    }
    b = b.overflow(t)
    i = 0
```

```
    goto next
}
```

在上小节中，咱们已经选定了起始桶的位置。接下来就是通过 `mapiternext` 进行**具体的循环遍历动作**。该方法主要涉及如下：

- 从已选定的桶中开始进行遍历，寻找桶中的下一个元素进行处理
- 如果桶已经遍历完，则对溢出桶 `overflow buckets` 进行遍历处理

通过对本方法的阅读，可得知其对 buckets 的**遍历规则**以及对于扩容的一些处理（这不是本文重点。因此没有具体展开）

总结

在本文开始，咱们先提出核心讨论点：“为什么 Go map 遍历输出是不固定顺序？”。而通过这一番分析，原因也很简单明了。就是 `for range map` 在开始处理循环逻辑的时候，就做了随机播种...

你想问为什么要这么做？当然是官方有意为之，因为 Go 在早期（1.0）的时候，虽是稳定迭代的，但从结果来讲，其实是无法保证每个 Go 版本迭代遍历规则都是一样的。而这将会导致可移植性问题。因此，改之。也请不要依赖...

参考

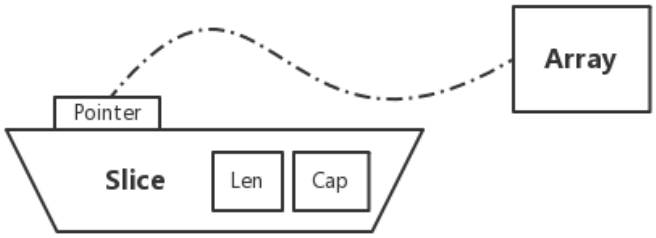
- [Go maps in action](https://blog.golang.org/g

slice

深入理解 Go Slice

Go Slice 最大容量大小是怎么来的

深入理解 Go Slice



是什么

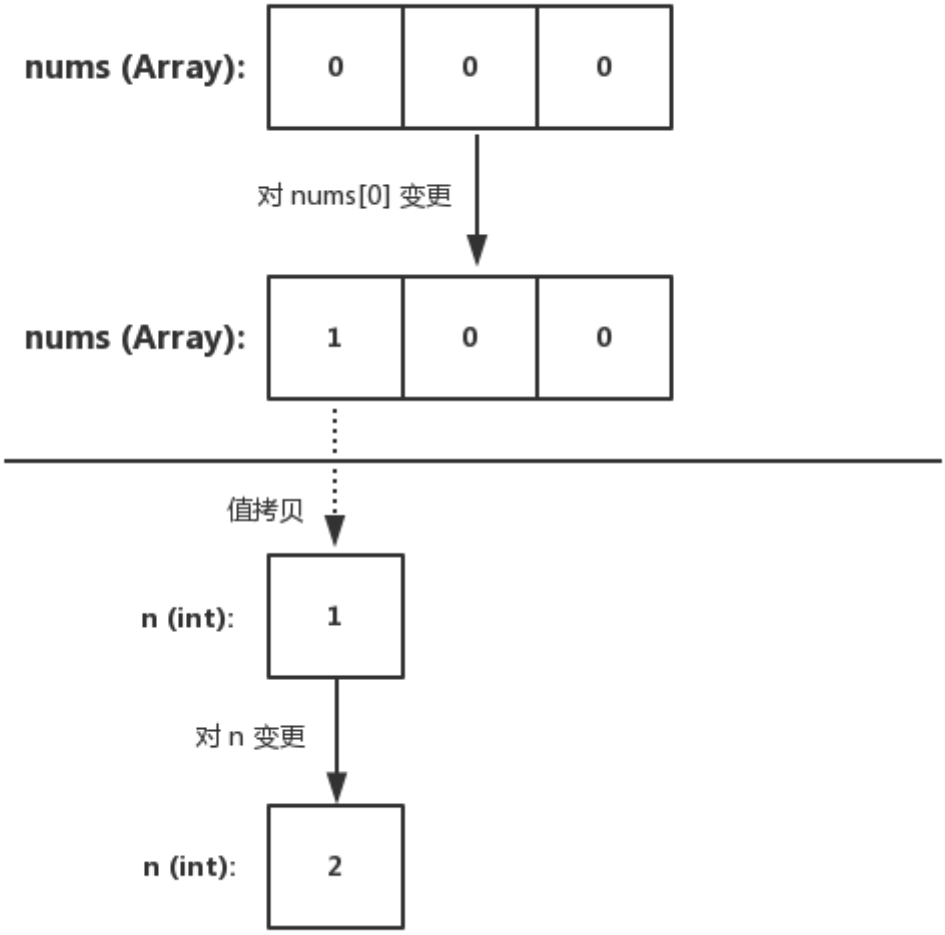
在 Go 中，Slice（切片）是抽象在 Array（数组）之上的特殊类型。为了更好地了解 Slice，第一步需要先对 Array 进行理解。深刻了解 Slice 与 Array 之间的区别后，就能更好的对其底层一番摸索 😄

用法

Array

```
func main() {
    nums := [3]int{}
    nums[0] = 1

    n := nums[0]
    n = 2

    fmt.Printf("nums: %v\n", nums)
    fmt.Printf("n: %d\n", n)
}
```

我们可得知在 Go 中，数组类型需要指定长度和元素类型。在上述代码中，可得知 `[3]int{}` 表示 3 个整数的数组，并进行了初始化。底层数据存储为一段连续的内存空间，通过固定的索引值（下标）进行检索



数组在声明后，其元素的初始值（也就是零值）为 0。并且该变量可以直接使用，不需要特殊操作

同时数组的长度是固定的，它的长度是类型的一部分，因此 `[3]int` 和 `[4]int` 在类型上是不同的，不能称为“一个东西”

输出结果

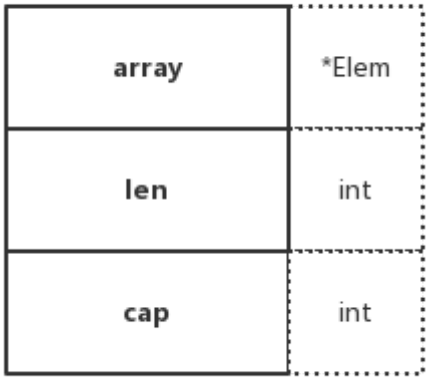
```
nums: [1 0 0]
n: 2
```

Slice

```
func main() {
    nums := [3]int{}
    nums[0] = 1

    dnums := nums[:]

    fmt.Printf("dnums: %v", dnums)
}
```

Slice 是对 Array 的抽象，类型为 `[]T` 。在上述代码中， `dnums` 变量通过 `nums[:]` 进行赋值。需要注意的是，Slice 和 Array 不一样，它不需要指定长度。也更加的灵活，能够自动扩容

数据结构




```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

Slice 的底层数据结构共分为三部分，如下：

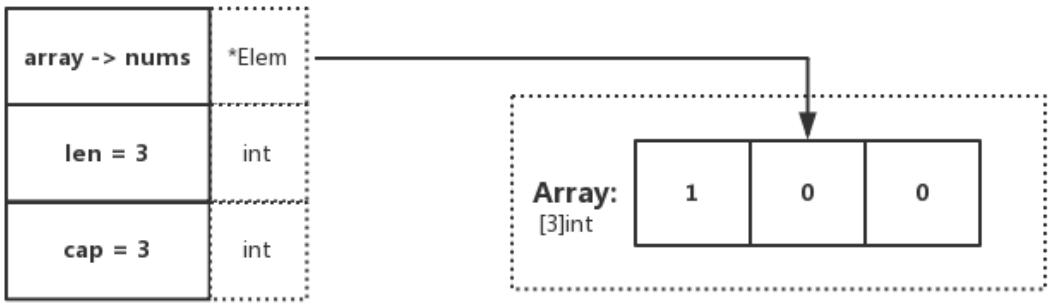
- array：指向所引用的数组指针（ `unsafe.Pointer` 可以表示任何可寻址的值的指针）
- len：长度，当前引用切片的元素个数
- cap：容量，当前引用切片的容量（底层数组的元素总数）

在实际使用中，cap 一定是大于或等于 len 的。否则会导致 panic

示例

为了更好的理解，我们回顾上小节的代码便于演示，如下：

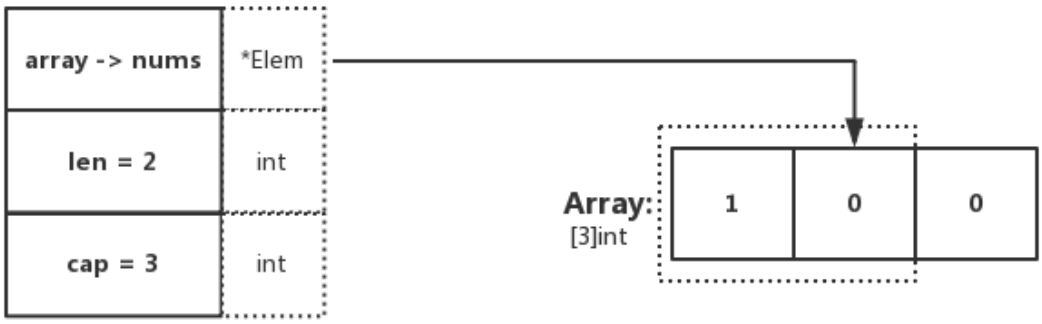
```
func main() {
    nums := [3]int{}
    nums[0] = 1

    dnums := nums[:]

    fmt.Printf("dnums: %v", dnums)
}
```



在代码中，可观察到 `dnums := nums[:]` ，这段代码确定了 Slice 的 Pointer 指向数组，且 len 和 cap 都为数组的基础属性。与图示表达一致

len、cap 不同

```
func main() {
    nums := [3]int{}
    nums[0] = 1

    dnums := nums[0:2]

    fmt.Printf("dnums: %v, len: %d, cap: %d", dnums, len(dnums), cap(dnums))
}
```



输出结果

```
dnums: [1 0], len: 2, cap: 3
```

显然，在这里指定了 `Slice[0:2]` ，因此 len 为所引用元素的个数，cap 为所引用的数组元素总个数。与期待一致 😄

创建

Slice 的创建有两种方式，如下：

- `var []T` 或 `[]T{}`
- `func make（[] T，len，cap）[] T`

可以留意 make 函数，我们都知道 Slice 需要指向一个 Array。那 make 是怎么做的呢？

它会在调用 make 的时候，分配一个数组并返回引用该数组的 Slice

```
func makeslice(et *_type, len, cap int) slice {
    maxElements := maxSliceCap(et.size)
    if len < 0 || uintptr(len) > maxElements {
        panic(errorString("makeslice: len out of range"))
    }

    if cap < len || uintptr(cap) > maxElements {
        panic(errorString("makeslice: cap out of range"))
    }

    p := mallocgc(et.size*uintptr(cap), et, true)
    return slice{p, len, cap}
}
```

- 根据传入的 Slice 类型，获取其类型能够申请的最大容量大小
- 判断 len 是否合规，检查是否在 0 < x < maxElements 范围内
- 判断 cap 是否合规，检查是否在 len < x < maxElements 范围内
- 申请 Slice 所需的内存空间对象。若为大型对象（大于 32 KB）则直接从堆中分配
- 返回申请成功的 Slice 内存地址和相关属性（默认返回申请到的内存起始地址）

扩容

当使用 Slice 时，若存储的元素不断增长（例如通过 append）。当条件满足扩容的策略时，将会触发自动扩容

那么分别是什么规则呢？让我们一起看看源码是怎么说的 😄

zerobase

```
func growslice(et *_type, old slice, cap int) slice {
    ...
    if et.size == 0 {
        if cap < old.cap {
            panic(errorString("growslice: cap out of range"))
        }

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

当 Slice size 为 0 时，若将要扩容的容量比原本的容量小，则抛出异常（也就是不支持缩容操作）。否则，将重新生成一个新的 Slice 返回，其 Pointer 指向一个 0 byte 地址（不会保留老的 Array 指向）

扩容 - 计算策略

```
func growslice(et *_type, old slice, cap int) slice {
    ...
    newcap := old.cap
    doublecap := newcap + newcap
    if cap > doublecap {
        newcap = cap
    } else {
        if old.len < 1024 {
```

```
            newcap = doublecap
        } else {
            for 0 < newcap && newcap < cap {
                newcap += newcap / 4
            }
            ...
        }
    }
    ...
}
```

- 若 Slice cap 大于 doublecap，则扩容后容量大小为 新 Slice 的容量（超了基准值，我就只给你需要的容量大小）
- 若 Slice len 小于 1024 个，在扩容时，增长因子为 1（也就是 3 个变 6 个）
- 若 Slice len 大于 1024 个，在扩容时，增长因子为 0.25（原本容量的四分之一）

注：也就是小于 1024 个时，增长 2 倍。大于 1024 个时，增长 1.25 倍

扩容 - 内存策略

```
func growslice(et *_type, old slice, cap int) slice {
    ...
    var overflow bool
    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))
        overflow = uintptr(newcap) > _MaxMem
        newcap = int(capmem)
        ...
    }

    if cap < old.cap || overflow || capmem > _MaxMem {
        panic(errorString("growslice: cap out of range"))
    }

    var p unsafe.Pointer
    if et.kind&kindNoPointers != 0 {
        p = mallocgc(capmem, nil, false)
        memmove(p, old.array, lenmem)
        memclrNoHeapPointers(add(p, newlenmem), capmem-newlenmem)
```

```
    } else {
        p = mallocgc(capmem, et, true)
        if !writeBarrier.enabled {
            memmove(p, old.array, lenmem)
        } else {
            for i := uintptr(0); i < lenmem; i += et.size {
                typedmemmove(et, add(p, i), add(old.array, i))
            }
        }
    }
    ...
}
```

1、获取老 Slice 长度和计算假定扩容后的新 Slice 元素长度、容量大小以及指针地址（用于后续操作内存的一系列操作）

2、确定新 Slice 容量大于老 Sice，并且新容量内存小于指定的最大内存、没有溢出。否则抛出异常

3、若元素类型为 `kindNoPointers` ，也就是**非指针**类型。则在老 Slice 后继续扩容

- 第一步：根据先前计算的 `capmem` ，在老 Slice cap 后继续申请内存空间，其后用于扩容
- 第二步：将 old.array 上的 n 个 bytes（根据 lenmem）拷贝到新的内存空间上
- 第三步：新内存空间（p）加上新 Slice cap 的容量地址。最终得到完整的新 Slice cap 内存地址 `add(p, newlenmem)` （ptr）
- 第四步：从 ptr 开始重新初始化 n 个 bytes（capmem-newlenmem）

注：那么问题来了，为什么要重新初始化这块内存呢？这是因为 ptr 是未初始化的内存（例如：可重用的内存，一般用于新的内存分配），其可能包含“垃圾”。因此在这里应当进行“清理”。便于后面实际使用（扩容）

4、不满足 3 的情况下，重新申请并初始化一块内存给新 Slice 用于存储 Array

5、检测当前是否正在执行 GC，也就是当前是否启用 Write Barrier（写屏障），若**启用**则通过 `typedmemmove` 方法，利用指针运算**循环拷贝**。否则通过 `memmove` 方法采取**整体拷贝**的方式将 lenmem 个字节从 old.array 拷贝到 ptr，以此达到更高的效率

注：一般会在 GC 标记阶段启用 Write Barrier，并且 Write Barrier 只针对指针启用。那么在第 5 点中，你就不难理解为什么会有两种截然不同的处理方式了

小结

这里需要注意的是，扩容时的内存管理的选择项，如下：

- 翻新扩展：当前元素为 `kindNoPointers` ，将在老 Slice cap 的地址后继续申请空间用于扩容
- 举家搬迁：重新申请一块内存地址，整体迁移并扩容

两个小“陷阱”

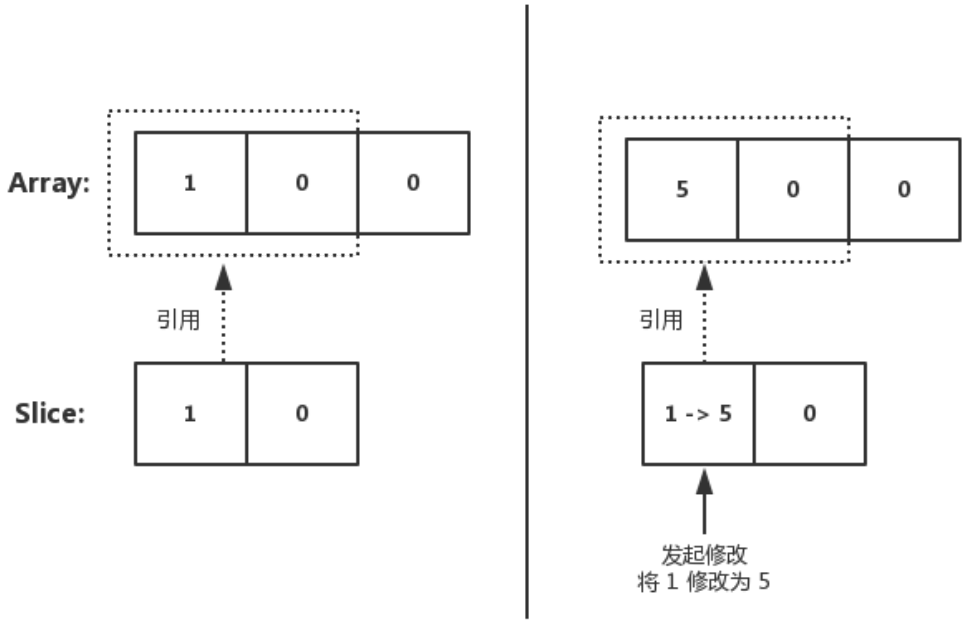
一、同根

```
func main() {
    nums := [3]int{}
    nums[0] = 1

    fmt.Printf("nums: %v , len: %d, cap: %d\n", nums, len(nums), cap(nums))

    dnums := nums[0:2]
    dnums[0] = 5

    fmt.Printf("nums: %v ,len: %d, cap: %d\n", nums, len(nums), cap(nums))
    fmt.Printf("dnums: %v, len: %d, cap: %d\n", dnums, len(dnums), cap(dnums))
}
```

输出结果：

```
nums: [1 0 0] , len: 3, cap: 3
nums: [5 0 0] ,len: 3, cap: 3
dnums: [5 0], len: 2, cap: 3
```

在**未扩容前**，Slice array 指向所引用的 Array。因此在 Slice 上的变更。会直接修改到原始 Array 上（两者所引用的是同一个）



二、时过境迁

随着 Slice 不断 append，内在的元素越来越多，终于触发了扩容。如下代码：

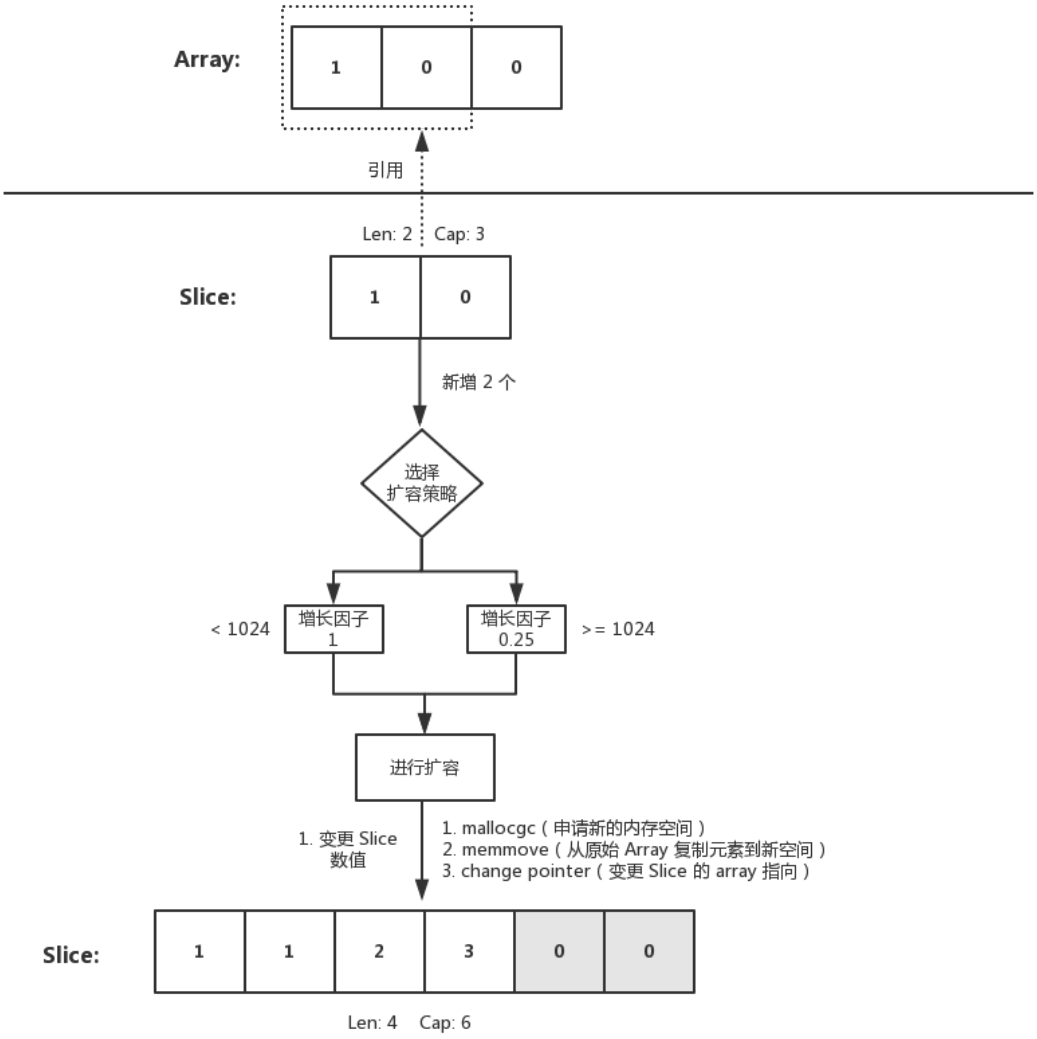
```
func main() {
    nums := [3]int{}
    nums[0] = 1

    fmt.Printf("nums: %v , len: %d, cap: %d\n", nums, len(nums), cap(nums))

    dnums := nums[0:2]
    dnums = append(dnums, []int{2, 3}...)
    dnums[1] = 1

    fmt.Printf("nums: %v ,len: %d, cap: %d\n", nums, len(nums), cap(nums))
    fmt.Printf("dnums: %v, len: %d, cap: %d\n", dnums, len(dnums), cap(dnums))
}
```

输出结果：

```
nums: [1 0 0] , len: 3, cap: 3
nums: [1 0 0] ,len: 3, cap: 3
dnums: [1 1 2 3], len: 4, cap: 6
```

往 Slice append 元素时，若满足扩容策略，也就是假设插入后，原本数组的容量就超过最大值了

这时候内部就会重新申请一块内存空间，将原本的元素**拷贝**一份到新的内存空间上。此时其与原本的数组就没有任何关联关系了，**再进行修改值也不会变动到原始数组**。这是需要注意的



复制

原型

```
func copy（dst, src []T）int
```

copy 函数将数据从**源 Slice**复制到**目标 Slice**。它返回复制的元素数。

示例

```
func main() {
    dst := []int{1, 2, 3}
    src := []int{4, 5, 6, 7, 8}
    n := copy(dst, src)

    fmt.Printf("dst: %v, n: %d", dst, n)
}
```

copy 函数支持在不同长度的 Slice 之间进行复制，若出现长度不一致，在复制时会按照最少的 Slice 元素个数进行复制

那么在源码中是如何完成复制这一个行为的呢？我们来一起看看源码的实现，如下：

```
func slicecopy(to, fm slice, width uintptr) int {
    if fm.len == 0 || to.len == 0 {
        return 0
    }

    n := fm.len
    if to.len < n {
        n = to.len
    }

    if width == 0 {
        return n
    }

    ...

    size := uintptr(n) * width
    if size == 1 {
        *(*byte)(to.array) = *(*byte)(fm.array) // known to be a byte pointer
    } else {
        memmove(to.array, fm.array, size)
    }
    return n
}
```

- 若源 Slice 或目标 Slice 存在长度为 0 的情况，则直接返回 0（因为压根不需要执行复制行为）
- 通过对比两个 Slice，获取最小的 Slice 长度。便于后续操作

- 若 Slice 只有一个元素，则直接利用指针的特性进行转换
- 若 Slice 大于一个元素，则从 `fm.array` 复制 `size` 个字节到 `to.array` 的地址处（会覆盖原有的值）

“奇特”的初始化

在 Slice 中流传着两个传说，分别是 Empty 和 Nil Slice，接下来让我们看看它们的小区别

Empty

```
func main() {
    nums := []int{}
    renums := make([]int, 0)

    fmt.Printf("nums: %v, len: %d, cap: %d\n", nums, len(nums), cap(nums))
    fmt.Printf("renums: %v, len: %d, cap: %d\n", renums, len(renums), cap(renum
s))
}
```

输出结果：

```
nums: [], len: 0, cap: 0
renums: [], len: 0, cap: 0
```

Nil

```
func main() {
    var nums []int
}
```

输出结果：

```
nums: [], len: 0, cap: 0
```

想一想

乍一看，Empty Slice 和 Nil Slice 好像一模一样？不管是 len，还是 cap 都为 0。好像没区别？我们再看看如下代码：

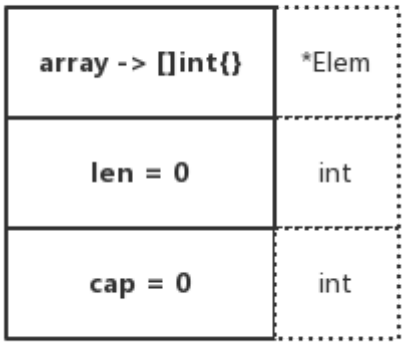
```
func main() {
    var nums []int
    renums := make([]int, 0)
    if nums == nil {
        fmt.Println("nums is nil.")
    }
    if renums == nil {
        fmt.Println("renums is nil.")
    }
}
```

你觉得输出结果是什么呢？你可能已经想到了，最终的输出结果：

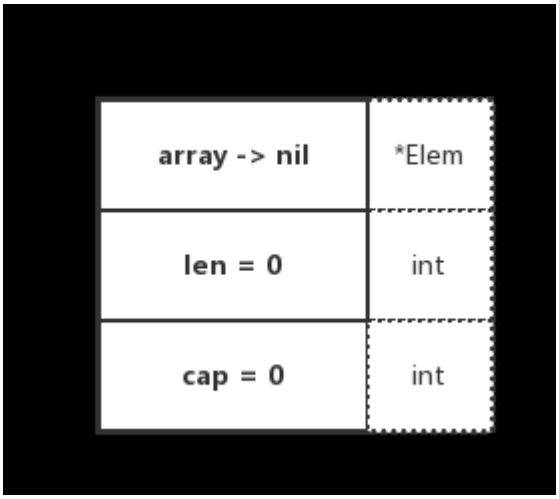
```
nums is nil.
```

为什么

Empty



Nil



从图示中可以看出来，两者有本质上的区别。其底层数组的指向指针是不一样的，Nil Slice 指向的是 nil，Empty Slice 指向的是实际存在的空数组地址

你可以认为，Nil Slice 代指不存在的 Slice，Empty Slice 代指空集合。两者所代表的意义是完全不同的

总结

通过本文，可得知 Go Slice 相当灵活。不需要你手动扩容，也不需要你关注加多少减多少。对 Array 是动态引用，是 Go 类型的一个极大的补充，也因此在应用中使用的更多、更便捷

虽然有个别要注意的 “坑”，但其实是合理的。你觉得呢？😄

Go Slice 最大容量大小是怎么来的

前言

在《深入理解 Go Slice》中，我们提到了“根据其类型大小去获取能够申请的最大容量大小”的处理逻辑。今天我们将更深入地去探究一下，底层到底做了什么东西，涉及什么知识点？

Go Slice 对应代码如下：

```
func makeslice(et *_type, len, cap int) slice {
    maxElements := maxSliceCap(et.size)
    if len < 0 || uintptr(len) > maxElements {
        ...
    }

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

    p := mallocgc(et.size*uintptr(cap), et, true)
    return slice{p, len, cap}
}
```

根据想要追寻的逻辑，定位到了 `maxSliceCap` 方法，它会根据**当前类型的大小获取到了所允许的最大容量大小**来进行阈值判断，也就是安全检查。这是浅层的了解，我们继续追下去看看还做了些什么？

maxSliceCap

```
func maxSliceCap(elemsize uintptr) uintptr {
    if elemsize < uintptr(len(maxElems)) {
        return maxElems[elemsize]
    }
    return maxAlloc / elemsize
}
```

maxElems

```
var maxElems = [...]uintptr{
    ^uintptr(0),
    maxAlloc / 1, maxAlloc / 2, maxAlloc / 3, maxAlloc / 4,
    maxAlloc / 5, maxAlloc / 6, maxAlloc / 7, maxAlloc / 8,
    maxAlloc / 9, maxAlloc / 10, maxAlloc / 11, maxAlloc / 12,
    maxAlloc / 13, maxAlloc / 14, maxAlloc / 15, maxAlloc / 16,
    maxAlloc / 17, maxAlloc / 18, maxAlloc / 19, maxAlloc / 20,
    maxAlloc / 21, maxAlloc / 22, maxAlloc / 23, maxAlloc / 24,
    maxAlloc / 25, maxAlloc / 26, maxAlloc / 27, maxAlloc / 28,
    maxAlloc / 29, maxAlloc / 30, maxAlloc / 31, maxAlloc / 32,
}
```

`maxElems` 是包含一些预定义的切片最大容量值的查找表，索引是切片元素的类型大小。而值看起来“奇奇怪怪”不大眼熟，都是些什么呢。主要是以下三个核心点：

- ^uintptr(0)
- maxAlloc
- maxAlloc / typeSize

^uintptr(0)

```
func main() {
    log.Printf("uintptr: %v\n", uintptr(0))
    log.Printf("^uintptr: %v\n", ^uintptr(0))
}
```

输出结果：

```
2019/01/05 17:51:52 uintptr: 0
2019/01/05 17:51:52 ^uintptr: 18446744073709551615
```

我们留意一下输出结果，比较神奇。取反之后为什么是 18446744073709551615 呢？

uintptr 是什么

在分析之前，我们要知道 uintptr 的本质（真面目），也就是它的类型是什么，如下：

```
type uintptr uintptr
```

uintptr 的类型是自定义类型，接着找它的真面目，如下：

```
#ifdef _64BIT
typedef    uint64        uintptr;
#else
typedef    uint32        uintptr;
#endif
```

通过对以上代码的分析，可得出以下结论：

- 在 32 位系统下，uintptr 为 uint32 类型，占用大小为 4 个字节
- 在 64 位系统下，uintptr 为 uint64 类型，占用大小为 8 个字节

^uintptr 做了什么事

^ 位运算符的作用是**按位异或**，如下：

```
func main() {
    log.Println(^1)
    log.Println(^uint64(0))
}
```

输出结果：

```
2019/01/05 20:44:49 -2
2019/01/05 20:44:49 18446744073709551615
```

接下来我们分析一下，这两段代码都做了什么事情呢

^1

二进制：0001

按位取反：1110

该数为有符号整数，最高位为符号位。低三位为表示数值。按位取反后为 1110，根据先前的说明，最高位为 1，因此表示为 -。取反后 110 对应十进制 -2

^uint64(0)

二进制：0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000

按位取反：1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

该数为无符号整数，该位取反后得到十进制值为：18446744073709551615

这个值是不是看起来很眼熟呢？没错，就是 `^uintptr(0)` 的值。也印证了其底层数据类型为 uint64 的事实 （本机为 64 位）。同时它又代表如下：

- math.MaxUint64
- 2 的 64 次方减 1

maxAlloc

```
const GoarchMips = 0
const GoarchMipsle = 0
const GoarchWasm = 0

...

_64bit = 1 << (^uintptr(0) >> 63) / 2

heapAddrBits = (_64bit*(1-sys.GoarchWasm))*48 + (1-_64bit+sys.GoarchWasm)*(32-(sys.GoarchMips+sys.GoarchMipsle))

maxAlloc = (1 << heapAddrBits) - (1-_64bit)*1
```

`maxAlloc` 是**允许用户分配的最大虚拟内存空间**。在 64 位，理论上可分配最大 `1 << heapAddrBits` 字节。在 32 位，最大可分配小于 `1 << 32` 字节

在本文，仅需了解它承载的是什么就好了。具体的在以后内存管理的文章再讲述

注：该变量在 go 10.1 为 `_MaxMem` ，go 11.4 已改为 `maxAlloc` 。相关的 `heapAddrBits` 计算方式也有所改变

maxAlloc / typeSize

我们再次回顾 `maxSliceCap` 的逻辑代码，这次重点放在控制逻辑，如下：

```
// func makeslice
maxElements := maxSliceCap(et.size)

...

// func maxSliceCap
if elemsize < uintptr(len(maxElems)) {
    return maxElems[elemsize]
```

```
}
return maxAlloc / elemsize
```

通过这段代码和 Slice 上下文逻辑，可得知在想得到该类型的最大容量大小时。会根据对应的类型大小去查找表查找索引（索引为类型大小，摆放顺序是有考虑原因的）。“迫不得已的情况下”才会手动的计算它的值，最终计算得到的内存字节大小都为该类型大小的整数倍

查找表的设置，更像是一个优化逻辑。减少常用的计算开销 :)

总结

通过本文的分析，可得出 Slice 所允许申请的最大容量大小，与当前**值类型**和当前**平台位数**有直接关系

最后

本文与《有点不安全却又一亮的 Go unsafe.Pointer》一同属于《深入理解 Go Slice》的关联章节。如果你在阅读源码时，对这些片段有疑惑。记得想尽办法深究下去，搞懂它

短短的一句话其实蕴含着不少知识点，希望这篇文章恰恰好可以帮你解惑

注：本文 Go 代码基于版本 11.4

panic

作为一个 gophper，我相信你对于 `panic` 和 `recover` 肯定不陌生，但是你有没有想过。当我们执行了这两条语句之后。底层到底发生了什么事呢？前几天和同事刚好聊到相关的话题，发现其实大家对这块理解还是比较模糊的。希望这篇文章能够从更深入的角度告诉你为什么，它到底做了什么事？

思考

一、为什么会中止运行

```
func main() {
    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
panic: EDDYCJY.

goroutine 1 [running]:
main.main()
    /Users/eddycjy/go/src/github.com/EDDYCJY/awesomeProject/main.go:4 +0x39
exit status 2
```

请思考一下，为什么执行 `panic` 后会导致应用程序运行中止？（而不是单单说执行了 `panic` 所以就结束了这么含糊）

二、为什么不会中止运行

```
func main() {
    defer func() {
        if err := recover(); err != nil {
            log.Printf("recover: %v", err)
        }
    }()

    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
2019/05/11 23:39:47 recover: EDDYCJY.
```

请思考一下，为什么加上 `defer` + `recover` 组合就可以保护应用程序？

三、不设置 defer 行不

上面问题二是 `defer` + `recover` 组合，那我去掉 `defer` 是不是也可以呢？如下：

```
func main() {
    if err := recover(); err != nil {
        log.Printf("recover: %v", err)
    }

    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
panic: EDDYCJY.

goroutine 1 [running]:
main.main()
    /Users/eddycjy/go/src/github.com/EDDYCJY/awesomeProject/main.go:10 +0xa1
exit status 2
```

竟然不行，啊呀毕竟入门教程都写的 `defer` + `recover` 组合“万能”捕获。但是为什么呢。去掉 `defer` 后为什么就无法捕获了？

请思考一下，为什么需要设置 `defer` 后 `recover` 才能起作用？

同时你还需要仔细想想，我们设置 `defer` + `recover` 组合后就能无忧无虑了吗，各种“乱”写了吗？

四、为什么起个 goroutine 就不行

```
func main() {
    go func() {
        defer func() {
            if err := recover(); err != nil {
                log.Printf("recover: %v", err)
```

```
            }
        }()
    }()

    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
panic: EDDYCJY.

goroutine 1 [running]:
main.main()
    /Users/eddycjy/go/src/github.com/EDDYCJY/awesomeProject/main.go:14 +0x51
exit status 2
```

请思考一下，为什么新起了一个 `Goroutine` 就无法捕获到异常了？到底发生了什么事...

源码

接下来我们将带着上述 4+1 个小思考题，开始对源码的剖析和分析，尝试从阅读源码中找到思考题的答案和更多为什么

数据结构

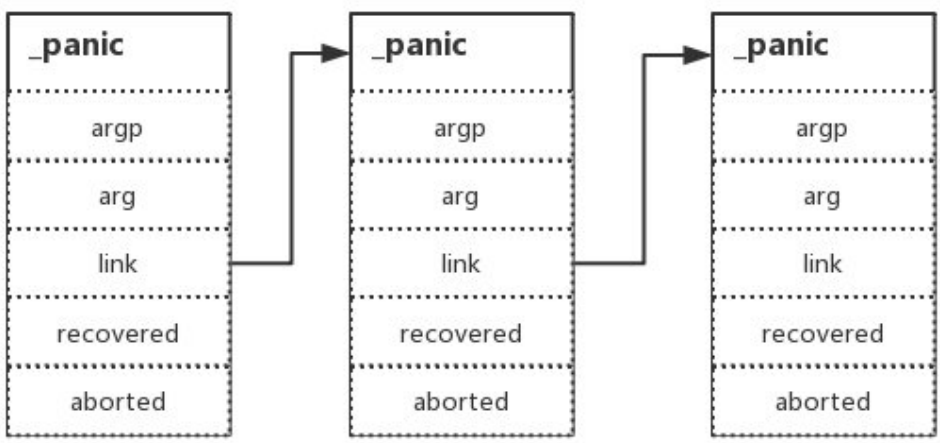
```
type _panic struct {
    argp      unsafe.Pointer
    arg       interface{}
    link      *_panic
    recovered bool
    aborted   bool
}
```

在 `panic` 中是使用 `_panic` 作为其基础单元的，每执行一次 `panic` 语句，都会创建一个 `_panic` 。它包含了一些基础的字段用于存储当前的 `panic` 调用情况，涉及的字段如下：

- argp：指向 `defer` 延迟调用的参数的指针
- arg：`panic` 的原因，也就是调用 `panic` 时传入的参数
- link：指向上一个调用的 `_panic`
- recovered：`panic` 是否已经被处理，也就是是否被 `recover`

- aborted：`panic` 是否被中止

另外通过查看 `link` 字段，可得知其是一个链表的数据结构，如下图：



恐慌 panic

```
func main() {
    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
panic: EDDYCJY.

goroutine 1 [running]:
main.main()
    /Users/eddycjy/go/src/github.com/EDDYCJY/awesomeProject/main.go:4 +0x39
exit status 2
```

我们去反查一下 `panic` 处理具体逻辑的地方在哪，如下：

```
$ go tool compile -S main.go
"".main STEXT size=66 args=0x0 locals=0x18
    0x0000 00000 (main.go:23)    TEXT    "".main(SB), ABIInternal, $24-0
    0x0000 00000 (main.go:23)    MOVQ    (TLS), CX
    0x0009 00009 (main.go:23)    CMPQ    SP, 16(CX)
```

```
    ...
    0x002f 00047 (main.go:24)    PCDATA    $2, $0
    0x002f 00047 (main.go:24)    MOVQ    AX, 8(SP)
    0x0034 00052 (main.go:24)    CALL    runtime.gopanic(SB)
```

显然汇编代码直指内部实现是 `runtime.gopanic` ，我们一起来看看这个方法做了什么事，如下（省略了部分）：

```
func gopanic(e interface{}) {
    gp := getg()
    ...
    var p _panic
    p.arg = e
    p.link = gp._panic
    gp._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

    for {
        d := gp._defer
        if d == nil {
            break
        }

        // defer...
        ...
        d._panic = (*_panic)(noescape(unsafe.Pointer(&p)))

        p.argp = unsafe.Pointer(getargp(0))
        reflectcall(nil, unsafe.Pointer(d.fn), deferArgs(d), uint32(d.siz), uint
32(d.siz))
        p.argp = nil

        // recover...
        if p.recovered {
            ...
            mcall(recovery)
            throw("recovery failed") // mcall should not return
        }
    }

    preprintpanics(gp._panic)

    fatalpanic(gp._panic) // should not return
    *(*int)(nil) = 0      // not reached
}
```

- 获取指向当前 `Goroutine` 的指针
- 初始化一个 `panic` 的基本单位 `_panic` 用作后续的操作
- 获取当前 `Goroutine` 上挂载的 `_defer`（数据结构也是链表）
- 若当前存在 `defer` 调用，则调用 `reflectcall` 方法去执行先前 `defer` 中延迟执行的代码，若在执行过程中需要运行 `recover` 将会调用 `gorecover` 方法
- 结束前，使用 `preprintpanics` 方法打印出所涉及的 `panic` 消息
- 最后调用 `fatalpanic` 中止应用程序，实际是执行 `exit(2)` 进行最终退出行为的

通过对上述代码的执行分析，可得知 `panic` 方法实际上就是处理当前 `Goroutine(g)` 上所挂载的 `._panic` 链表（所以无法对其他 `Goroutine` 的异常事件响应），然后对其所属的 `defer` 链表和 `recover` 进行检测并处理，最后调用退出命令中止应用程序

无法恢复的恐慌 fatalpanic

```
func fatalpanic(msgs *_panic) {
    pc := getcallerpc()
    sp := getcallersp()
    gp := getg()
    var docrash bool

    systemstack(func() {
        if startpanic_m() && msgs != nil {
            ...
            printpanics(msgs)
        }

        docrash = dopanic_m(gp, pc, sp)
    })

    systemstack(func() {
        exit(2)
    })

    *(*int)(nil) = 0
}
```

我们看到在异常处理的最后会执行该方法，似乎它承担了所有收尾工作。实际呢，它是在最后对程序执行 `exit` 指令来达到中止运行的作用，但在结束前它会通过 `printpanics` 递归输出所有的异常消息及参数。代码如下：

```
func printpanics(p *_panic) {
    if p.link != nil {
        printpanics(p.link)
        print("\t")
    }
    print("panic: ")
    printany(p.arg)
    if p.recovered {
        print(" [recovered]")
    }
    print("\n")
}
```

所以不要以为所有的异常都能够被 `recover` 到，实际上像 `fatal error` 和 `runtime.throw` 都是无法被 `recover` 到的，甚至是 oom 也是直接中止程序的，也有反手就给你来个 `exit(2)` 教做人。因此在写代码时你应该要相对注意些，“恐慌”是存在无法恢复的场景的

恢复 recover

```
func main() {
    defer func() {
        if err := recover(); err != nil {
            log.Printf("recover: %v", err)
        }
    }()

    panic("EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
2019/05/11 23:39:47 recover: EDDYCJY.
```

和预期一致，成功捕获到了异常。但是 `recover` 是怎么恢复 `panic` 的呢？再看看汇编代码，如下：

```
$ go tool compile -S main.go
"".main STEXT size=110 args=0x0 locals=0x18
    0x0000 00000 (main.go:5)    TEXT    "".main(SB), ABIInternal, $24-0
    ...
    0x0024 00036 (main.go:6)    LEAQ    "".main.func1·f(SB), AX
```

```
    0x002b 00043 (main.go:6)    PCDATA    $2, $0
    0x002b 00043 (main.go:6)    MOVQ    AX, 8(SP)
    0x0030 00048 (main.go:6)    CALL    runtime.deferproc(SB)
    ...
    0x0050 00080 (main.go:12)    CALL    runtime.gopanic(SB)
    0x0055 00085 (main.go:12)    UNDEF
    0x0057 00087 (main.go:6)    XCHGL    AX, AX
    0x0058 00088 (main.go:6)    CALL    runtime.deferreturn(SB)
    ...
    0x0022 00034 (main.go:7)    MOVQ    AX, (SP)
    0x0026 00038 (main.go:7)    CALL    runtime.gorecover(SB)
    0x002b 00043 (main.go:7)    PCDATA    $2, $1
    0x002b 00043 (main.go:7)    MOVQ    16(SP), AX
    0x0030 00048 (main.go:7)    MOVQ    8(SP), CX
    ...
    0x0056 00086 (main.go:8)    LEAQ    go.string."recover: %v"(SB), AX
    ...
    0x0086 00134 (main.go:8)    CALL    log.Printf(SB)
    ...
```

通过分析底层调用，可得知主要是如下几个方法：

- runtime.deferproc
- runtime.gopanic
- runtime.deferreturn
- runtime.gorecover

在上小节中，我们讲述了简单的流程，`gopanic` 方法会调用当前 `Goroutine` 下的 `defer` 链表，若 `reflectcall` 执行中遇到 `recover` 就会调用 `gorecover` 进行处理，该方法代码如下：

```
func gorecover(argp uintptr) interface{} {
    gp := getg()
    p := gp._panic
    if p != nil && !p.recovered && argp == uintptr(p.argp) {
        p.recovered = true
        return p.arg
    }
    return nil
}
```

这代码，看上去挺简单的，核心就是修改 `recovered` 字段。该字段是用于标识当前 `panic` 是否已经被 `recover` 处理。但是这和我们想象的并不一样啊，程序是怎么从

`panic` 流转回去的呢？是不是在核心方法里处理了呢？我们再看看 `gopanic` 的代码，如下：

```
func gopanic(e interface{}) {
    ...
    for {
        // defer...
        ...
        pc := d.pc
        sp := unsafe.Pointer(d.sp) // must be pointer so it gets adjusted during
stack copy
        freedefer(d)

        // recover...
        if p.recovered {
            atomic.Xadd(&runningPanicDefers, -1)

            gp._panic = p.link
            for gp._panic != nil && gp._panic.aborted {
                gp._panic = gp._panic.link
            }
            if gp._panic == nil {
                gp.sig = 0
            }

            gp.sigcode0 = uintptr(sp)
            gp.sigcode1 = pc
            mcall(recovery)
            throw("recovery failed")
        }
    }
    ...
}
```

我们回到 `gopanic` 方法中再仔细看看，发现实际上是包含对 `recover` 流转的处理代码的。恢复流程如下：

- 判断当前 `_panic` 中的 `recover` 是否已标注为处理
- 从 `_panic` 链表中删除已标注中止的 `panic` 事件，也就是删除已经被恢复的 `panic` 事件
- 将相关需要恢复的栈帧信息传递给 `recovery` 方法的 `gp` 参数（每个栈帧对应着一个未运行完的函数。栈帧中保存了该函数的返回地址和局部变量）
- 执行 `recovery` 进行恢复动作

从流程来看，最核心的是 `recovery` 方法。它承担了异常流转控制的职责。代码如下：

```
func recovery(gp *g) {
    sp := gp.sigcode0
    pc := gp.sigcode1

    if sp != 0 && (sp < gp.stack.lo || gp.stack.hi < sp) {
        print("recover: ", hex(sp), " not in [", hex(gp.stack.lo), ", ", hex(gp.stack.hi), "]\n")
        throw("bad recovery")
    }

    gp.sched.sp = sp
    gp.sched.pc = pc
    gp.sched.lr = 0
    gp.sched.ret = 1
    gogo(&gp.sched)
}
```

粗略一看，似乎就是很简单的设置了一些值？但实际上设置的是编译器中伪寄存器的值，常常被用于维护上下文等。在这里我们需要结合 `gopanic` 方法一同观察 `recovery` 方法。它所使用的栈指针 `sp` 和程序计数器 `pc` 是由当前 `defer` 在调用流程中的 `deferproc` 传递下来的，因此实际上最后是通过 `gogo` 方法跳回了 `deferproc` 方法。另外我们注意到：

```
gp.sched.ret = 1
```

在底层中程序将 `gp.sched.ret` 设置为了 `1`，也就是**没有实际调用** `deferproc` 方法，直接修改了其返回值。意味着默认它已经处理完成。直接转移到 `deferproc` 方法的下一条指令去。至此为止，异常状态的流转控制就已经结束了。接下来就是继续走 `defer` 的流程了

为了验证这个想法，我们可以看一下核心的跳转方法 `gogo` ，代码如下：

```
// void gogo(Gobuf*)
// restore state from Gobuf; longjmp
TEXT runtime·gogo(SB),NOSPLIT,$8-4
    MOVW    buf+0(FP), R1
    MOVW    gobuf_g(R1), R0
    BL    setg<>(SB)

    MOVW    gobuf_sp(R1), R13    // restore SP==R13
    MOVW    gobuf_lr(R1), LR
    MOVW    gobuf_ret(R1), R0
```

```
    MOVW    gobuf_ctxt(R1), R7
    MOVW    $0, R11
    MOVW    R11, gobuf_sp(R1)    // clear to help garbage collector
    MOVW    R11, gobuf_ret(R1)
    MOVW    R11, gobuf_lr(R1)
    MOVW    R11, gobuf_ctxt(R1)
    MOVW    gobuf_pc(R1), R11
    CMP    R11, R11 // set condition codes for == test, needed by stack split
    B    (R11)
```

通过查看代码可得知其主要作用是从 `Gobuf` 恢复状态。简单来讲就是将寄存器的值修改为对应 `Goroutine(g)` 的值，而在文中讲了很多次的 `Gobuf` ，如下：

```
type gobuf struct {
    sp   uintptr
    pc   uintptr
    g    guintptr
    ctxt unsafe.Pointer
    ret  sys.Uintreg
    lr   uintptr
    bp   uintptr
}
```

讲道理，其实它存储的就是 `Goroutine` 切换上下文时所需要的一些东西

拓展

```
const(
    OPANIC       // panic(Left)
    ORECOVER     // recover()
    ...
)
...
func walkexpr(n *Node, init *Nodes) *Node {
    ...
    switch n.Op {
    default:
        Dump("walk", n)
        Fatalf("walkexpr: switch 1 unknown op %+S", n)

    case ONONAME, OINDREGSP, OEMPTY, OGETG:
    case OTYPE, ONAME, OLITERAL:
        ...
    case OPANIC:
```

```
        n = mkcall("gopanic", nil, init, n.Left)

    case ORECOVER:
        n = mkcall("gorecover", n.Type, init, nod(OADDR, nodfp, nil))
    ...
}
```

实际上在调用 `panic` 和 `recover` 关键字时，是在编译阶段先转换为相应的 OPCODE 后，再由编译器转换为对应的运行时方法。并不是你所想像那样一步到位，有兴趣的小伙伴可以研究一下

总结

本文主要针对 `panic` 和 `recover` 关键字进行了深入源码的剖析，而开头的 4+1 个思考题，就是希望您能够带着疑问去学习，达到事半功倍的功效

另外本文和 `defer` 有一定的关联性，因此需要有一定的基础知识。若刚刚看的时候这部分不理解，学习后可以再读一遍加深印象

在最后，现在的你可以回答这几个思考题了吗？说出来了才是真的懂：）

标准库

log 标准库

fmt 标准库

有点不安全却又一亮的 Go unsafe.Pointer

log 标准库

日志

输出

```
2018/09/28 20:03:08 EDDYCJY Blog...
```

构成

[日期]<空格>[时分秒]<空格>[内容]<\n>

源码剖析

Logger

```
type Logger struct {
    mu     sync.Mutex
    prefix string
    flag   int
    out    io.Writer
    buf    []byte
}
```

1. mu：互斥锁，用于确保原子的写入
2. prefix：每行需写入的日志前缀内容
3. flag：设置日志辅助信息（时间、文件名、行号）的写入。可选如下标识位：

```
const (
    Ldate         = 1 << iota       // value: 1
    Ltime                           // value: 2
    Lmicroseconds                   // value: 4
    Llongfile                       // value: 8
    Lshortfile                      // value: 16
    LUTC                            // value: 32
    LstdFlags     = Ldate | Ltime   // value: 3
)
```

- Ldate：当地时区的格式化日期：2009/01/23
- Ltime：当地时区的格式化时间：01:23:23
- Lmicroseconds：在 Ltime 的基础上，增加微秒的时间数值显示
- Llongfile：完整的文件名和行号：/a/b/c/d.go:23
- Lshortfile：当前文件名和行号：d.go：23，会覆盖 Llongfile 标识
- LUTC：如果设置 Ldate 或 Ltime，且设置 LUTC，则优先使用 UTC 时区而不是本地时区
- LstdFlags：Logger 的默认初始值（Ldate 和 Ltime）

4. out：io.Writer
5. buf：用于存储将要写入的日志内容

New

```
func New(out io.Writer, prefix string, flag int) *Logger {
    return &Logger{out: out, prefix: prefix, flag: flag}
}

var std = New(os.Stderr, "", LstdFlags)
```

New 方法用于初始化 Logger，接受三个初始参数，可以定制化而在 log 包内默认会初始一个 std，它指向标准输入流。而默认的标准输出、标准错误就是显示器（输出到屏幕上），标准输入就是键盘。辅助的时间信息默认为 `Ldate | Ltime` ，也就是 `2009/01/23 01:23:23`

```
// os
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

- Stdin：标准输入
- Stdout：标准输出
- Stderr：标准错误

Getter

- Flags
- Prefix

Setter

- SetFlags
- SetPrefix
- SetOutput

Print, *Fatal*, Panic*

```
func Print(v ...interface{}) {
    std.Output(2, fmt.Sprint(v...))
}

func Printf(format string, v ...interface{}) {
    std.Output(2, fmt.Sprintf(format, v...))
}

func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...))
}

func Fatal(v ...interface{}) {
    std.Output(2, fmt.Sprint(v...))
    os.Exit(1)
}

func Panic(v ...interface{}) {
    s := fmt.Sprint(v...)
    std.Output(2, s)
    panic(s)
}

...
```

这一部分介绍最常用的日志写入方法，从源码可得知 `Xrintln` 、 `Xrintf` 函数 **换行、可变参数**都是通过 `fmt` 标准库的方法去实现的

`Fatal` 和 `Panic` 是通过 `os.Exit(1)` 、 `panic(s)` 集成实现的。而具体的组装逻辑是通过 `Output` 方法实现的

Logger.Output

```
func (l *Logger) Output(calldepth int, s string) error {
    now := time.Now() // get this early.
```

```
    var file string
    var line int
    l.mu.Lock()
    defer l.mu.Unlock()
    if l.flag&(Lshortfile|Llongfile) != 0 {
        // Release lock while getting caller info - it's expensive.
        l.mu.Unlock()
        var ok bool
        _, file, line, ok = runtime.Caller(calldepth)
        if !ok {
            file = "???"
            line = 0
        }
        l.mu.Lock()
    }
    l.buf = l.buf[:0]
    l.formatHeader(&l.buf, now, file, line)
    l.buf = append(l.buf, s...)
    if len(s) == 0 || s[len(s)-1] != '\n' {
        l.buf = append(l.buf, '\n')
    }
    _, err := l.out.Write(l.buf)
    return err
}
```

Output 方法，简单来讲就是将写入的日志事件信息组装并输出，它会根据 flag 标识位的不同来使用 `runtime.Caller` 去获取当前 goroutine 所执行的函数文件、行号等调用信息（log 标准库中默认深度为 2）。另外如果结尾不是换行符 `\n`，将自动补全一个换行

Logger.formatHeader

```
func (l *Logger) formatHeader(buf *[]byte, t time.Time, file string, line int) {
    *buf = append(*buf, l.prefix...)
    if l.flag&(Ldate|Ltime|Lmicroseconds) != 0 {
        if l.flag&LUTC != 0 {
            t = t.UTC()
        }
        if l.flag&Ldate != 0 {
            year, month, day := t.Date()
            itoa(buf, year, 4)
            *buf = append(*buf, '/')
            itoa(buf, int(month), 2)
            *buf = append(*buf, '/')
            itoa(buf, day, 2)
            *buf = append(*buf, ' ')
```

```
        }
        if l.flag&(Ltime|Lmicroseconds) != 0 {
            hour, min, sec := t.Clock()
            itoa(buf, hour, 2)
            *buf = append(*buf, ':')
            itoa(buf, min, 2)
            *buf = append(*buf, ':')
            itoa(buf, sec, 2)
            if l.flag&Lmicroseconds != 0 {
                *buf = append(*buf, '.')
                itoa(buf, t.Nanosecond()/1e3, 6)
            }
            *buf = append(*buf, ' ')
        }
    }
    if l.flag&(Lshortfile|Llongfile) != 0 {
        if l.flag&Lshortfile != 0 {
            short := file
            for i := len(file) - 1; i > 0; i-- {
                if file[i] == '/' {
                    short = file[i+1:]
                    break
                }
            }
            file = short
        }
        *buf = append(*buf, file...)
        *buf = append(*buf, ':')
        itoa(buf, line, -1)
        *buf = append(*buf, ": "...)
    }
}
```

该方法主要是用于格式化日志头（前缀），根据入参不同的标识位，添加分隔符和对应的值到日志信息中。执行流程如下：

（1）如果不是空值，则将 prefix 写入 buf

（2）如果设置 `Ldate` 、 `Ltime` 、 `Lmicroseconds` ，则对应将日期和时间写入 buf

（3）如果设置 `Lshortfile` 、 `Llongfile` ，则对应将文件和行号信息写入 buf

Logger.itoa

```
func itoa(buf *[]byte, i int, wid int) {
    // Assemble decimal in reverse order.
    var b [20]byte
    bp := len(b) - 1
    for i >= 10 || wid > 1 {
        wid--
        q := i / 10
        b[bp] = byte('0' + i - q*10)
        bp--
        i = q
    }
    // i < 10
    b[bp] = byte('0' + i)
    *buf = append(*buf, b[bp:]...)
}
```

该方法主要用于将整数转换为定长的十进制 ASCII，同时给出负数宽度避免左侧补 0。另外会以相反的顺序组合十进制

如何定制化 Logger

在标准库内，可通过其开放的 New 方法来实现各种各样的自定义 Logger 组件，但是为什么也可以直接 `log.Print*` 等方法呢？

```
func New(out io.Writer, prefix string, flag int) *Logger
```

其实是在标准库内，如果你刚刚细心的看了前面的小节，不难发现其默认实现了一个 Logger 组件

```
var std = New(os.Stderr, "", LstdFlags)
```

这也是一个小小的精妙之处

总结

通过查阅 log 标准库的源码，可得知最简单的一个日志包应该如何编写。另外 log 包是在所有涉及到 Logger 的地方都对 `sync.Mutex` 进行操作（以此解决原子问题），其余逻辑均为组装日志信息和转换数值格式，该包较为经典，可以多读几遍 😄

问题

为什么在调用 `runtime.Caller` 前要先解锁，后再加锁呢?

fmt 标准库

前言

```
package main

import (
    "fmt"
)

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

标准开场见多了，那内部标准库又是怎么输出这段英文的呢？今天一起来围观下源码吧

原型

```
func Print(a ...interface{}) (n int, err error) {
    return Fprint(os.Stdout, a...)
}

func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}

func Printf(format string, a ...interface{}) (n int, err error) {
    return Fprintf(os.Stdout, format, a...)
}
```

- Print：使用默认格式说明符打印格式并写入标准输出。当两者都不是字符串时，在操作数之间添加空格
- Println：同上，不同的地方是始终在操作数之间添加空格，并附加换行符
- Printf：根据格式说明符进行格式化并写入标准输出

以上三类就是最常见的格式化 I/O 的方法，我们将基于此去进行拆解描述

执行流程

案例一：Print

在这里我们使用 `Print` 方法做一个分析，便于后面的加深理解 😄

```
func Print(a ...interface{}) (n int, err error) {
    return Fprint(os.Stdout, a...)
}
```

`Print` 使用默认格式说明符打印格式并写入标准输出。另外当两者都为非空字符串时将插入一个空格

原型

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrint(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

该函数一共有两个形参：

- w：输出流，只要实现 io.Writer 就可以（抽象）为流的写入
- a：任意类型的多个值

分析主干流程

1、 p := newPrinter(): 申请一个临时对象池（sync.Pool）

```
var ppFree = sync.Pool{
    New: func() interface{} { return new(pp) },
}

func newPrinter() *pp {
    p := ppFree.Get().(*pp)
    p.panicking = false
    p.erroring = false
    p.fmt.init(&p.buf)
    return p
}
```

- ppFree.Get()：基于 sync.Pool 实现 *pp 的临时对象池，每次获取一定会返回一个新的 pp 对象用于接下来的处理
- *pp.panicking：用于解决无限递归的 panic、recover 问题，会根据该参数在 catchPanic 及时掐断
- *pp.erroring：用于表示正在处理错误无效的 verb 标识符，主要作用是防止调用 handleMethods 方法
- *pp.fmt.init(&p.buf)：初始化 fmt 配置，会设置 buf 并且清空 fmtFlags 标志位

2、 p.doPrint(a): 执行约定的格式化动作（参数间增加一个空格、最后一个参数增加换行符）

```
func (p *pp) doPrint(a []interface{}) {
    prevString := false
    for argNum, arg := range a {
        true && false
        isString := arg != nil && reflect.TypeOf(arg).Kind() == reflect.String
        // Add a space between two non-string arguments.
        if argNum > 0 && !isString && !prevString {
            p.buf.WriteByte(' ')
        }
        p.printArg(arg, 'v')
        prevString = isString
    }
}
```

可以看到底层通过判断该入参，**同时**满足以下条件就会添加分隔符（空格）：

- 当前入参为多个参数（例如：Slice）
- 当前入参不为 nil 且不为字符串（通过反射确定）
- 当前入参不为首项或上一个入参不为字符串

而在 `Print` 方法中，不需要指定格式符。实际上在该方法内直接指定为 `v` 。也就是默认格式的值

```
p.printArg(arg, 'v')
```

3. w.Write(p.buf): 写入标准输出（io.Writer）

4. *pp.free(): 释放已缓存的内容。在使用完临时对象后，会将 buf、arg、value 清空再重新存放到 ppFree 中。以便于后面再取出重用（利用 sync.Pool 的临时对象特性）

案例二：Printf

标识符

Verbs

```
%v    the value in a default format
    when printing structs, the plus flag (%+v) adds field names
%#v    a Go-syntax representation of the value
%T    a Go-syntax representation of the type of the value
%%    a literal percent sign; consumes no value
%t    the word true or false
```

Flags

```
+    always print a sign for numeric values;
    guarantee ASCII-only output for %q (%+q)
-    pad with spaces on the right rather than the left (left-justify the field)
#    alternate format: add leading 0 for octal (%#o), 0x for hex (%#x);
    0X for hex (%#X); suppress 0x for %p (%#p);
    for %q, print a raw (backquoted) string if strconv.CanBackquote
    returns true;
    always print a decimal point for %e, %E, %f, %F, %g and %G;
    do not remove trailing zeros for %g and %G;
    write e.g. U+0078 'x' if the character is printable for %U (%#U).
' '    (space) leave a space for elided sign in numbers (% d);
    put spaces between bytes printing strings or slices in hex (% x, % X)
0    pad with leading zeros rather than spaces;
    for numbers, this moves the padding after the sign
```

详细建议参见 Godoc

原型

```
func Fprintf(w io.Writer, format string, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintf(format, a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

与 Print 相比，最大的不同就是 doPrintf 方法了。在这里我们来详细看看其代码，如下：

```
func (p *pp) doPrintf(format string, a []interface{}) {
    end := len(format)
```

```
    argNum := 0         // we process one argument per non-trivial format
    afterIndex := false // previous item in format was an index like [3].
    p.reordered = false
formatLoop:
    for i := 0; i < end; {
        p.goodArgNum = true
        lasti := i
        for i < end && format[i] != '%' {
            i++
        }
        if i > lasti {
            p.buf.WriteString(format[lasti:i])
        }
        if i >= end {
            // done processing format string
            break
        }

        // Process one verb
        i++

        // Do we have flags?
        p.fmt.clearflags()
    simpleFormat:
        for ; i < end; i++ {
            c := format[i]
            switch c {
            case '#':   //'#'、'0'、'+'、'-'、' '
                ...
            default:
                if 'a' <= c && c <= 'z' && argNum < len(a) {
                    ...
                    p.printArg(a[argNum], rune(c))
                    argNum++
                    i++
                    continue formatLoop
                }

                break simpleFormat
            }
        }

        // Do we have an explicit argument index?
        argNum, i, afterIndex = p.argNumber(argNum, format, i, len(a))

        // Do we have width?
```

```
        if i < end && format[i] == '*' {
            ...
        }

        // Do we have precision?
        if i+1 < end && format[i] == '.' {
            ...
        }

        if !afterIndex {
            argNum, i, afterIndex = p.argNumber(argNum, format, i, len(a))
        }

        if i >= end {
            p.buf.WriteString(noVerbString)
            break
        }

        ...

        switch {
        case verb == '%': // Percent does not absorb operands and ignores f.wid
 and f.prec.
            p.buf.WriteByte('%')
        case !p.goodArgNum:
            p.badArgNum(verb)
        case argNum >= len(a): // No argument left over to print for the current
verb.
            p.missingArg(verb)
        case verb == 'v':
            ...
            fallthrough
        default:
            p.printArg(a[argNum], verb)
            argNum++
        }
    }

    if !p.reordered && argNum < len(a) {
        ...
    }
}
```

分析主干流程

1. 写入 % 之前的字符内容
2. 如果所有标志位处理完毕（到达字符尾部），则跳出处理逻辑
3. （往后移）跳过 % ，开始处理其他 verb 标志位
4. 清空（重新初始化） fmt 配置
5. 处理一些基础的 verb 标识符（simpleFormat）。如：'#'、'0'、'+'、'-'、' ' 以及**简单的 verbs 标识符（不包含精度、宽度和参数索引）。需要注意的是，若当前字符为简单 verb 标识符。则直接进行处理。完成后会直接后移到下一个字符**。其余标志位则变更 fmt 配置项，便于后续处理
6. 处理参数索引（argument index）
7. 处理参数宽度（width）
8. 处理参数精度（precision）
9. % 之后若不存在 verbs 标识符则返回 `noVerbString` 。值为 %!(NOVERB)
10. 处理特殊 verbs 标识符（如：'%%'、'%#v'、'%+v'）、错误情况（如：参数索引指定错误、参数集个数与 verbs 标识符数量不匹配）或进行格式化参数集
11. 常规流程处理完毕

在特殊情况下，若提供的参数集比 verb 标识符多。fmt 将会贪婪检查下去，将多出的参数集以特定的格式输出，如下：

```
fmt.Printf("%d", 1, 2, 3)
// 1%!(EXTRA int=2, int=3)
```

- 约定前缀额外标志：%!(EXTRA
- 当前参数的类型
- 约定格式符：=
- 当前参数的值（默认以 %v 格式化）
- 约定格式符：)

值得注意的是，当指定了参数索引或实际处理的参数小于入参的参数集时，就不会进行贪婪匹配来展示

案例三：Println

原型

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintln(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

在这个方法中，最大的区别就是 doPrintln，我们一起来看看，如下：

```
func (p *pp) doPrintln(a []interface{}) {
    for argNum, arg := range a {
        if argNum > 0 {
            p.buf.WriteByte(' ')
        }
        p.printArg(arg, 'v')
    }
    p.buf.WriteByte('\n')
}
```

分析主干流程

- 循环入参的参数集，并以空格分隔
- 格式化当前参数，默认以 `%v` 对参数进行格式化
- 在结尾添加 `\n` 字符

如何格式化参数

在上例的执行流程分析中，可以看到格式化参数这一步是在 `p.printArg(arg, verb)` 执行的，我们一起来看看它都做了些什么？

```
func (p *pp) printArg(arg interface{}, verb rune) {
    p.arg = arg
    p.value = reflect.Value{}

    if arg == nil {
        switch verb {
        case 'T', 'v':
            p.fmt.padString(nilAngleString)
        default:
            p.badVerb(verb)
        }
        return
    }

    switch verb {
    case 'T':
        p.fmt.fmt_s(reflect.TypeOf(arg).String())
        return
    case 'p':
```

```
        p.fmtPointer(reflect.ValueOf(arg), 'p')
        return
    }

    // Some types can be done without reflection.
    switch f := arg.(type) {
    case bool:
        p.fmtBool(f, verb)
    case float32:
        p.fmtFloat(float64(f), 32, verb)
    ...
    case reflect.Value:
        if f.IsValid() && f.CanInterface() {
            p.arg = f.Interface()
            if p.handleMethods(verb) {
                return
            }
        }
        p.printValue(f, verb, 0)
    default:
        if !p.handleMethods(verb) {
            p.printValue(reflect.ValueOf(f), verb, 0)
        }
    }
}
```

在小节代码中可以看见，fmt 本身对不同的类型做了不同的处理。这样子就避免了通过反射确定。相对的提高了性能

其中有两个特殊的方法，分别是 `handleMethods` 和 `badVerb` ，接下来分别来看看他们的作用是什么

1、badVerb

它主要用于格式化并处理错误的行为。我们可以一起来看看，代码如下：

```
func (p *pp) badVerb(verb rune) {
    p.erroring = true
    p.buf.WriteString(percentBangString)
    p.buf.WriteRune(verb)
    p.buf.WriteByte('(')
    switch {
    case p.arg != nil:
        p.buf.WriteString(reflect.TypeOf(p.arg).String())
        p.buf.WriteByte('=')
        p.printArg(p.arg, 'v')
```

```
    ...
    default:
        p.buf.WriteString(nilAngleString)
    }
    p.buf.WriteByte(')')
    p.erroring = false
}
```

在处理错误格式化时，我们可以对比以下例子：

```
fmt.Printf("%s", []int64{1, 2, 3})
// [%!s(int64=1) %!s(int64=2) %!s(int64=3)]%
```

在 badVerb 中可以看到错误字符串的处理主要分为以下部分：

- 约定前缀错误标志：%!
- 当前的格式化操作符
- 约定格式符：(
- 当前参数的类型
- 约定格式符：=
- 当前参数的值（默认以 %v 格式化）
- 约定格式符：)

2、handleMethods

```
func (p *pp) handleMethods(verb rune) (handled bool) {
    if p.erroring {
        return
    }
    // Is it a Formatter?
    if formatter, ok := p.arg.(Formatter); ok {
        handled = true
        defer p.catchPanic(p.arg, verb)
        formatter.Format(p, verb)
        return
    }

    // If we're doing Go syntax and the argument knows how to supply it, take ca
re of it now.
    ...
```

```
    return false
}
```

这个方法比较特殊，一般在自定义结构体和未知情况下进行调用。主要流程是：

- 若当前参数为错误 verb 标识符，则直接返回
- 判断是否实现了 Formatter
- 实现，则利用自定义 Formatter 格式化参数
- 未实现，则最大程度的利用 Go syntax 默认规则去格式化参数

拓展

在 fmt 标准库中可以通过自定义结构体来实现方法的自定义，大致如下几种

fmt.State

```
type State interface {
    Write(b []byte) (n int, err error)

    Width() (wid int, ok bool)

    Precision() (prec int, ok bool)

    Flag(c int) bool
}
```

State 用于获取标志位的状态值，涉及如下：

- Write：将格式化完毕的字符写入缓冲区中，等待下一步处理
- Width：返回宽度信息和是否被设置
- Precision：返回精度信息和是否被设置
- Flag：返回特殊标志符（'#'、'0'、'+'、'-'、' '）是否被设置

fmt.Formatter

```
type Formatter interface {
    Format(f State, c rune)
}
```

Formatter 用于实现**自定义格式化方法**。可通过在自定义结构体中实现 Format 方法来实现这个目的

另外，可以通过 f 获取到当前标识符的宽度、精度等状态值。c 为 verb 标识符，可以得到其动作是什么

fmt.Stringer

```
type Stringer interface {
    String() string
}
```

当该对象为 String、Array、Slice 等类型时，将会调用 `String()` 方法对类字符串进行格式化

fmt.GoStringer

```
type GoStringer interface {
    GoString() string
}
```

当格式化特定 verb 标识符（%v）时，将调用 `GoString()` 方法对其进行格式化

总结

通过本文对 fmt 标准库的分析，可以发现它有以下特点：

- 在拓展性方面，可以自定义格式化方法等
- 在完整度方面，尽可能的贪婪匹配，输出参数集
- 在性能方面，每种不同的参数类型，都实现了不同的格式化处理操作
- 在性能方面，尽可能的最短匹配，格式化参数集

总的来说，fmt 标准库有许多值得推敲的细节，希望你能够在本文学到 😄

有点不安全却又一亮的 Go unsafe.Pointer

在上一篇文章《深入理解 Go Slice》中，大家会发现其底层数据结构使用了 `unsafe.Pointer` 。因此想着再介绍一下其关联知识

前言

在大家学习 Go 的时候，肯定都学过“Go 的指针是不支持指针运算和转换”这个知识点。为什么呢？

首先，Go 是一门静态语言，所有的变量都必须为标量类型。不同的类型不能够进行赋值、计算等跨类型的操作。那么指针也对应着相对的类型，也在 Compile 的静态类型检查的范围内。同时静态语言，也称为强类型。也就是一旦定义了，就不能再改变它

错误示例

```
func main(){
    num := 5
    numPointer := &num

    flnum := (*float32)(numPointer)
    fmt.Println(flnum)
}
```

输出结果：

```
# command-line-arguments
...: cannot convert numPointer (type *int) to type *float32
```

在示例中，我们创建了一个 `num` 变量，值为 5，类型为 `int` 。取了其对于的指针地址后，试图强制转换为 `*float32` ，结果失败...

unsafe

针对刚刚的“错误示例”，我们可以采用今天的男主角 `unsafe` 标准库来解决。它是一个神奇的包，在官方的诠释中，有如下概述：

- 围绕 Go 程序内存安全及类型的操作
- 很可能会是不可移植的
- 不受 Go 1 兼容性指南的保护

简单来讲就是，不怎么推荐你使用。因为它是 unsafe（不安全的），但是在特殊的场景下，使用了它。可以打破 Go 的类型和内存安全机制，让你获得眼前一亮的惊喜效果 😄

Pointer

为了解决这个问题，需要用到 `unsafe.Pointer` 。它表示任意类型且可寻址的指针值，可以在不同的指针类型之间进行转换（类似 C 语言的 void * 的用途）

其包含四种核心操作：

- 任何类型的指针值都可以转换为 Pointer
- Pointer 可以转换为任何类型的指针值
- uintptr 可以转换为 Pointer
- Pointer 可以转换为 uintptr

在这一部分，重点看第一点、第二点。你再想想怎么修改“错误示例”让它运行起来？

```
func main(){
    num := 5
    numPointer := &num

    flnum := (*float32)(unsafe.Pointer(numPointer))
    fmt.Println(flnum)
}
```

输出结果：

```
0xc4200140b0
```

在上述代码中，我们小加改动。通过 `unsafe.Pointer` 的特性对该指针变量进行了修改，就可以完成任意类型（*T）的指针转换

需要注意的是，这时还无法对变量进行操作或访问。因为不知道该指针地址指向的东西具体是什么类型。不知道是什么类型，又如何进行解析呢。无法解析也就自然无法对其变更了

Offsetof

在上小节中，我们对普通的指针变量进行了修改。那么它是否能做更复杂一点的事呢？

```
type Num struct{
    i string
    j int64
}
```

```
func main(){
    n := Num{i: "EDDYCJY", j: 1}
    nPointer := unsafe.Pointer(&n)

    niPointer := (*string)(unsafe.Pointer(nPointer))
    *niPointer = "煎鱼"

    njPointer := (*int64)(unsafe.Pointer(uintptr(nPointer) + unsafe.Offsetof(n.
j)))
    *njPointer = 2

    fmt.Printf("n.i: %s, n.j: %d", n.i, n.j)
}
```

输出结果：

```
n.i: 煎鱼, n.j: 2
```

在剖析这段代码做了什么事之前，我们需要了解结构体的一些基本概念：

- 结构体的成员变量在内存存储上是一段连续的内存
- 结构体的初始地址就是第一个成员变量的内存地址
- 基于结构体的成员地址去计算偏移量。就能够得出其他成员变量的内存地址

再回来看看上述代码，得出执行流程：

- 修改 `n.i` 值：`i` 为第一个成员变量。因此不需要进行偏移量计算，直接取出指针后转换为 `Pointer`，再强制转换为字符串类型的指针值即可
- 修改 `n.j` 值：`j` 为第二个成员变量。需要进行偏移量计算，才可以对其内存地址进行修改。在进行了偏移运算后，当前地址已经指向第二个成员变量。接着重复转换赋值即可

需要注意的是，这里使用了如下方法（来完成偏移计算的目标）：

1、uintptr：`uintptr` 是 Go 的内置类型。返回无符号整数，可存储一个完整的地址。后续常用于指针运算

```
type uintptr uintptr
```

2、unsafe.Offsetof：返回成员变量 x 在结构体当中的偏移量。更具体的讲，就是返回结构体初始位置到 x 之间的字节数。需要注意的是入参 `ArbitraryType` 表示任意类型，并非定义的 `int` 。它实际作用是一个占位符

```
func Offsetof(x ArbitraryType) uintptr
```

在这一部分，其实就是巧用了 `Pointer` 的第三、第四点特性。这时候就已经可以对变量进行操作了 😄

错误示例

```
func main(){
    n := Num{i: "EDDYCJY", j: 1}
    nPointer := unsafe.Pointer(&n)
    ...

    ptr := uintptr(nPointer)
    njPointer := (*int64)(unsafe.Pointer(ptr + unsafe.Offsetof(n.j)))
    ...
}
```

这里存在一个问题， `uintptr` 类型是不能存储在临时变量中的。因为从 GC 的角度来看， `uintptr` 类型的临时变量只是一个无符号整数，并不知道它是一个指针地址

因此当满足一定条件后， `ptr` 这个临时变量是可能被垃圾回收掉的，那么接下来的内存操作，岂不成迷？

总结

简洁回顾两个知识点。第一是 `unsafe.Pointer` 可以让你的变量在不同的指针类型转来转去，也就是表示为任意可寻址的指针类型。第二是 `uintptr` 常用于与 `unsafe.Pointer` 打配合，用于做指针运算，巧妙地很

最后还是那句，没有特殊必要的话。是不建议使用 `unsafe` 标准库，它并不安全。虽然它常常能让你眼前一亮

defer

在上一章节 《深入理解 Go panic and recover》中，我们发现了 `defer` 与其关联性极大，还是觉得非常有必要深入一下。希望通过本章节大家可以对 `defer` 关键字有一个深刻的理解，那么我们开始吧。你先等等，请排好队，我们这儿采取后进先出 LIFO 的出站方式...

特性

我们简单的过一下 `defer` 关键字的基础使用，让大家先有一个基础的认知

一、延迟调用

```
func main() {
    defer log.Println("EDDYCJY.")

    log.Println("end.")
}
```

输出结果：

```
$ go run main.go
2019/05/19 21:15:02 end.
2019/05/19 21:15:02 EDDYCJY.
```

二、后进先出

```
func main() {
    for i := 0; i < 6; i++ {
        defer log.Println("EDDYCJY" + strconv.Itoa(i) + ".")
    }


    log.Println("end.")
}
```

输出结果：

```
$ go run main.go
2019/05/19 21:19:17 end.
2019/05/19 21:19:17 EDDYCJY5.
```

```
2019/05/19 21:19:17 EDDYCJY4.
2019/05/19 21:19:17 EDDYCJY3.
2019/05/19 21:19:17 EDDYCJY2.
2019/05/19 21:19:17 EDDYCJY1.
2019/05/19 21:19:17 EDDYCJY0.
```

三、运行时间点

```
func main() {
    func() {
        defer log.Println("defer.EDDYCJY.")
    }()

    log.Println("main.EDDYCJY.")
}
```

输出结果：

```
$ go run main.go
2019/05/22 23:30:27 defer.EDDYCJY.
2019/05/22 23:30:27 main.EDDYCJY.
```

四、异常处理

```
func main() {
    defer func() {
        if e := recover(); e != nil {
            log.Println("EDDYCJY.")
        }
    }()

    panic("end.")
}
```

输出结果：

```
$ go run main.go
2019/05/20 22:22:57 EDDYCJY.
```

源码剖析

```
$ go tool compile -S main.go
"".main STEXT size=163 args=0x0 locals=0x40
    ...
    0x0059 00089 (main.go:6)    MOVQ    AX, 16(SP)
    0x005e 00094 (main.go:6)    MOVQ    $1, 24(SP)
    0x0067 00103 (main.go:6)    MOVQ    $1, 32(SP)
    0x0070 00112 (main.go:6)    CALL    runtime.deferproc(SB)
    0x0075 00117 (main.go:6)    TESTL   AX, AX
    0x0077 00119 (main.go:6)    JNE     137
    0x0079 00121 (main.go:7)    XCHGL   AX, AX
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
    0x0084 00132 (main.go:7)    ADDQ    $64, SP
    0x0088 00136 (main.go:7)    RET
    0x0089 00137 (main.go:6)    XCHGL   AX, AX
    0x008a 00138 (main.go:6)    CALL    runtime.deferreturn(SB)
    0x008f 00143 (main.go:6)    MOVQ    56(SP), BP
    0x0094 00148 (main.go:6)    ADDQ    $64, SP
    0x0098 00152 (main.go:6)    RET
    ...
```

首先我们需要找到它，找到它实际对应什么执行代码。通过汇编代码，可得知涉及如下方法：

- runtime.deferproc
- runtime.deferreturn

很显然是运行时的方法，是对的人。我们继续往下走看看都分别承担了什么行为

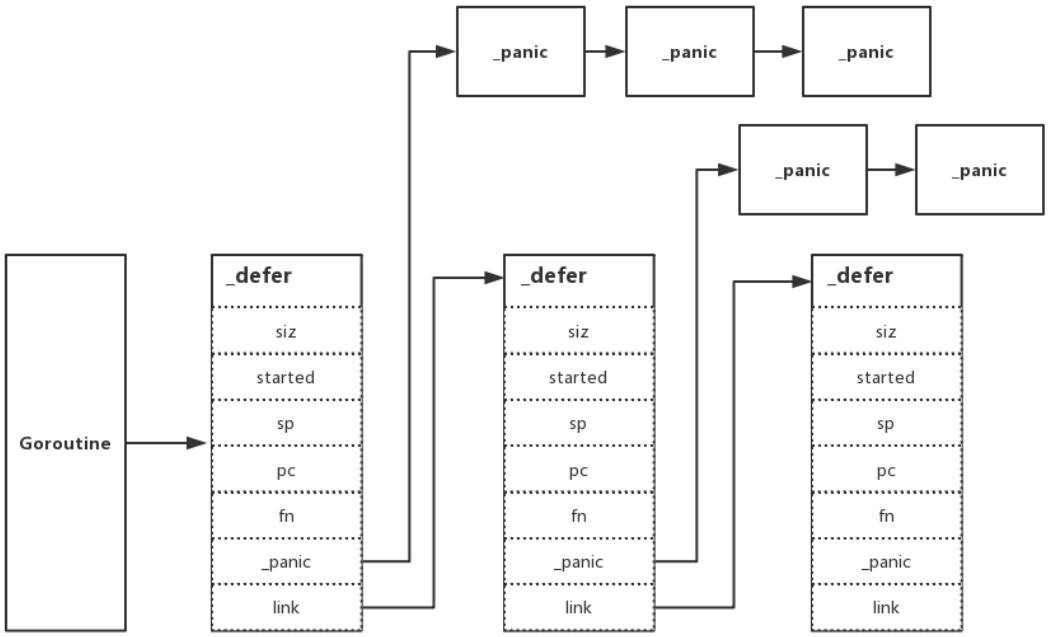
数据结构

在开始前我们需要先介绍一下 `defer` 的基础单元 `_defer` 结构体，如下：

```
type _defer struct {
    siz     int32
    started bool
    sp      uintptr // sp at time of defer
    pc      uintptr
    fn      *funcval
    _panic  *_panic // panic that is running defer
    link    *_defer
}

...
type funcval struct {
```

```
    fn uintptr
    // variable-size, fn-specific data here
}
```

- siz：所有传入参数的总大小
- started：该 `defer` 是否已经执行过
- sp：函数栈指针寄存器，一般指向当前函数栈的栈顶
- pc：程序计数器，有时称为指令指针(IP)，线程利用它来跟踪下一个要执行的指令。在大多数处理器中，PC 指向的是下一条指令，而不是当前指令
- fn：指向传入的函数地址和参数
- _panic：指向 `_panic` 链表
- link：指向 `_defer` 链表



deferproc

```
func deferproc(siz int32, fn *funcval) {
    ...
    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
    callerpc := getcallerpc()

    d := newdefer(siz)
```

```
    ...
    d.fn = fn
    d.pc = callerpc
    d.sp = sp
    switch siz {
    case 0:
        // Do nothing.
    case sys.PtrSize:
        *(*uintptr)(deferArgs(d)) = *(*uintptr)(unsafe.Pointer(argp))
    default:
        memmove(deferArgs(d), unsafe.Pointer(argp), uintptr(siz))
    }

    return0()
}
```

- 获取调用 `defer` 函数的函数栈指针、传入函数的参数具体地址以及 PC （程序计数器），也就是下一个要执行的指令。这些相当于是预备参数，便于后续的流转控制
- 创建一个新的 `defer` 最小单元 `_defer`，填入先前准备的参数
- 调用 `memmove` 将传入的参数存储到新 `_defer` （当前使用）中去，便于后续的使用
- 最后调用 `return0` 进行返回，这个函数非常重要。能够避免在 `deferproc` 中又因为返回 `return`，而诱发 `deferreturn` 方法的调用。其根本原因是一个停止 `panic` 的延迟方法会使 `deferproc` 返回 1，但在机制中如果 `deferproc` 返回不等于 0，将会总是检查返回值并跳转到函数的末尾。而 `return0` 返回的就是 0，因此可以防止重复调用

小结

在**这个函数中会为新的 `_defer` 设置一些基础属性，并将调用函数的参数集传入。最后通过特殊的返回方法结束函数调用**。另外这一块与先前 《深入理解 Go panic and recover》 的处理逻辑有一定关联性，其实就是 `gp.sched.ret` 返回 0 还是 1 会分流至不同处理方式

newdefer

```
func newdefer(siz int32) *_defer {
    var d *_defer
    sc := deferclass(uintptr(siz))
    gp := getg()
    if sc < uintptr(len(p{}.deferpool)) {
        pp := gp.m.p.ptr()
```

```
        if len(pp.deferpool[sc]) == 0 && sched.deferpool[sc] != nil {
            ...
            lock(&sched.deferlock)
            d := sched.deferpool[sc]
            unlock(&sched.deferlock)
        }
        ...
    }
    if d == nil {
        systemstack(func() {
            total := roundupsize(totaldefersize(uintptr(siz)))
            d = (*_defer)(mallocgc(total, deferType, true))
        })
        ...
    }
    d.siz = siz
    d.link = gp._defer
    gp._defer = d
    return d
}
```

- 从池中获取可以使用的 `_defer`，则复用作为新的基础单元
- 若在池中没有获取到可用的，则调用 `mallocgc` 重新申请一个新的
- 设置 `defer` 的基础属性，最后修改当前 `Goroutine` 的 `_defer` 指向

通过这个方法我们可以注意到两点，如下：

- `defer` 与 `Goroutine(g)` 有直接关系，所以讨论 `defer` 时基本离不开 `g` 的关联
- 新的 `defer` 总是会在现有的链表中的最前面，也就是 `defer` 的特性后进先出

小结

这个函数主要承担了获取新的 `_defer` 的作用，它有可能是从 `deferpool` 中获取的，也有可能是重新申请的

deferreturn

```
func deferreturn(arg0 uintptr) {
    gp := getg()
    d := gp._defer
    if d == nil {
        return
```

```
    }
    sp := getcallersp()
    if d.sp != sp {
        return
    }

    switch d.siz {
    case 0:
        // Do nothing.
    case sys.PtrSize:
        *(*uintptr)(unsafe.Pointer(&arg0)) = *(*uintptr)(deferArgs(d))
    default:
        memmove(unsafe.Pointer(&arg0), deferArgs(d), uintptr(d.siz))
    }
    fn := d.fn
    d.fn = nil
    gp._defer = d.link
    freedefer(d)
    jmpdefer(fn, uintptr(unsafe.Pointer(&arg0)))
}
```

如果在一个方法中调用过 `defer` 关键字，那么编译器将会在结尾处插入 `deferreturn` 方法的调用。而该方法中主要做了如下事项：

- 清空当前节点 `_defer` 被调用的函数调用信息
- 释放当前节点的 `_defer` 的存储信息并放回池中（便于复用）
- 跳转到调用 `defer` 关键字的调用函数处

在这段代码中，跳转方法 `jmpdefer` 格外重要。因为它显式的控制了流转，代码如下：

```
// asm_amd64.s
TEXT runtime·jmpdefer(SB), NOSPLIT, $0-16
    MOVQ    fv+0(FP), DX    // fn
    MOVQ    argp+8(FP), BX    // caller sp
    LEAQ    -8(BX), SP    // caller sp after CALL
    MOVQ    -8(SP), BP    // restore BP as if deferreturn returned (harmless if
 framepointers not in use)
    SUBQ    $5, (SP)    // return to CALL again
    MOVQ    0(DX), BX
    JMP    BX    // but first run the deferred function
```

通过源码的分析，我们发现它做了两个很“奇怪”又很重要的事，如下：

- MOVQ -8(SP), BP：`-8(BX)` 这个位置保存的是 `deferreturn` 执行完毕后的地址

- SUBQ $5, (SP)：`SP` 的地址减 5 ，其减掉的长度就恰好是 `runtime.deferreturn` 的长度

你可能会问，为什么是 5？好吧。翻了半天最后看了一下汇编代码...嗯，相减的确是 5 没毛病，如下：

```
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
```

我们整理一下思绪，照上述逻辑的话，那 `deferreturn` 就是一个“递归”了哦。每次都会重新回到 `deferreturn` 函数，那它在什么时候才会结束呢，如下：

```
func deferreturn(arg0 uintptr) {
    gp := getg()
    d := gp._defer
    if d == nil {
        return
    }
    ...
}
```

也就是会不断地进入 `deferreturn` 函数，判断链表中是否还存着 `_defer` 。若已经不存在了，则返回，结束掉它。简单来讲，就是处理完全部 `defer` 才允许你真的离开它。果真如此吗？我们再看看上面的汇编代码，如下：

```
    。..
    0x0070 00112 (main.go:6)    CALL    runtime.deferproc(SB)
    0x0075 00117 (main.go:6)    TESTL   AX, AX
    0x0077 00119 (main.go:6)    JNE     137
    0x0079 00121 (main.go:7)    XCHGL   AX, AX
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
    0x0084 00132 (main.go:7)    ADDQ    $64, SP
    0x0088 00136 (main.go:7)    RET
    0x0089 00137 (main.go:6)    XCHGL   AX, AX
    0x008a 00138 (main.go:6)    CALL    runtime.deferreturn(SB)
    ...
```

的确如上述流程所分析一致，验证完毕

小结

这个函数主要承担了清空已使用的 `defer` 和跳转到调用 `defer` 关键字的函数处，非常重要

总结

我们有提到 `defer` 关键字涉及两个核心的函数，分别是 `deferproc` 和 `deferreturn` 函数。而 `deferreturn` 函数比较特殊，是当应用函数调用 `defer` 关键字时，编译器会在其结尾处插入 `deferreturn` 的调用，它们俩一般都是成对出现的

但是当一个 `Goroutine` 上存在着多次 `defer` 行为（也就是多个 `_defer` ）时，编译器会进行利用一些小技巧， 重新回到 `deferreturn` 函数去消耗 `_defer` 链表，直到一个不剩才允许真正的结束

而新增的基础单元 `_defer` ，有可能是被复用的，也有可能是全新申请的。它最后都会被追加到 `_defer` 链表的表头，从而设定了后进先出的调用特性

关联

- 深入理解 Go panic and recover

参考

- Scheduling In Go
- Dive into stack and defer/panic/recover in go
- golang-notes

crawler

爬取豆瓣电影 Top250

爬取汽车之家 二手车产品库

了解一下Golang的市场行情

爬取豆瓣电影 Top250

爬虫是标配了，看数据那一刻很有趣。第一个就从最最最简单最基础的爬虫开始写起吧！

项目地址：https://github.com/go-crawler/douban-movie

目标

我们的目标站点是 豆瓣电影 Top250，估计大家都很眼熟了

本次爬取 8 个字段，用于简单的概括分析。具体的字段如下：

简单的分析一下目标源

- 一页共 25 条
- 含分页（共 10 页）且分页规则是正常的
- 每一项的数据字段排序都是规则且不变

开始

由于量不大，我们的爬取步骤如下

- 分析页面，获取所有的分页
- 分析页面，循环爬取所有页面的电影信息
- 爬取的电影信息入库

安装

```
$ go get -u github.com/PuerkitoBio/goquery
```

运行

```
$ go run main.go
```

代码片段

1、获取所有分页

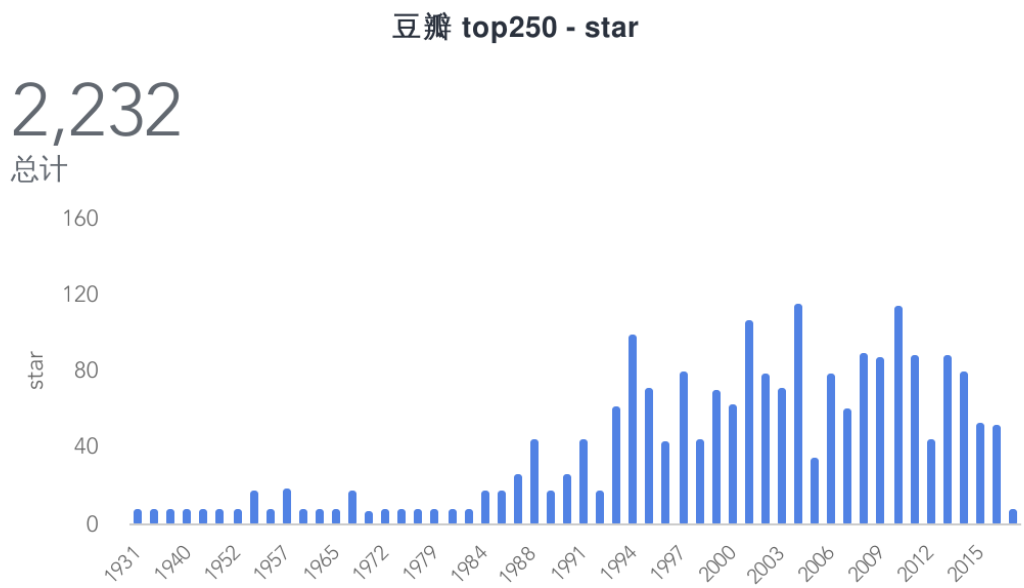
```
func ParsePages(doc *goquery.Document) (pages []Page) {
    pages = append(pages, Page{Page: 1, Url: ""})
    doc.Find("#content > div > div.article > div.paginator > a").Each(func(i int
, s *goquery.Selection) {
        page, _ := strconv.Atoi(s.Text())
        url, _ := s.Attr("href")

        pages = append(pages, Page{
            Page: page,
            Url:  url,
        })
    })

    return pages
}
```

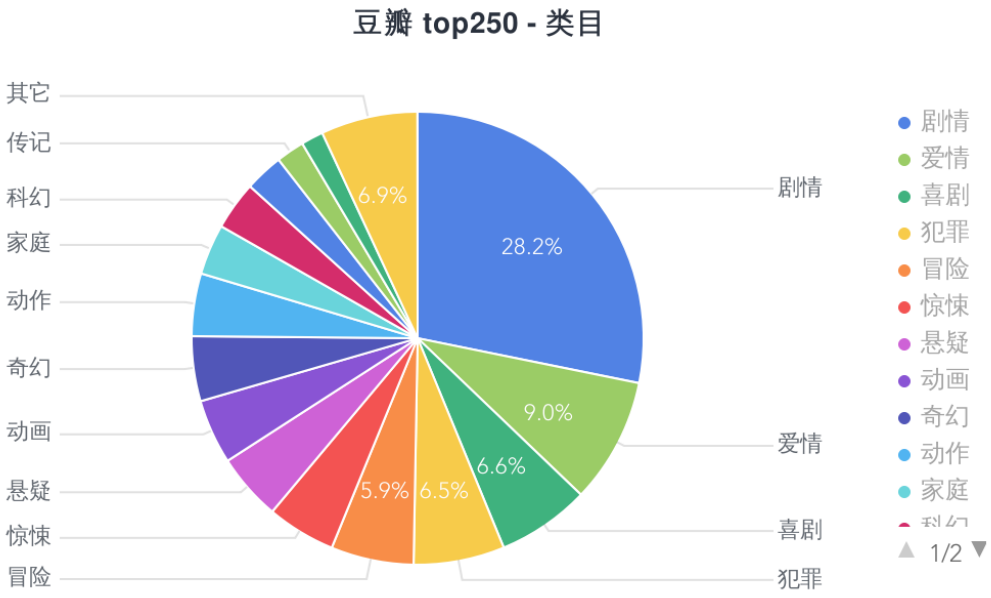
2、分析豆瓣电影信息

```
func ParseMovies(doc *goquery.Document) (movies []Movie) {
    doc.Find("#content > div > div.article > ol > li").Each(func(i int, s *goque
ry.Selection) {
        title := s.Find(".hd a span").Eq(0).Text()

        ...

        movieDesc := strings.Split(DescInfo[1], "/")
        year := strings.TrimSpace(movieDesc[0])
        area := strings.TrimSpace(movieDesc[1])
        tag := strings.TrimSpace(movieDesc[2])
```

```
        star := s.Find(".bd .star .rating_num").Text()

        comment := strings.TrimSpace(s.Find(".bd .star span").Eq(3).Text())
        compile := regexp.MustCompile("[0-9]")
        comment = strings.Join(compile.FindAllString(comment, -1), "")

        quote := s.Find(".quote .inq").Text()

        ...

        log.Printf("i: %d, movie: %v", i, movie)

        movies = append(movies, movie)
    })

    return movies
}
```

数据





看到这些数据，你有什么想法呢，真是好奇 :=)

爬取汽车之家 二手车产品库

项目地址：https://github.com/go-crawler/car-prices

目标

最近经常有人在耳边提起汽车之家，也好奇二手车在国内的价格是怎么样的，因此本次的目标站点是 汽车之家 的二手车产品库

分析目标源：

- 一页共 24 条
- 含分页，但这个老产品库，在 100 页后会存在问题，因此我们爬取 99 页
- 可以获取全部城市
- 共可爬取 19w+ 数据

开始

爬取步骤

- 获取全部的城市
- 拼装全部城市 URL 入队列
- 解析二手车页面结构
- 下一页 URL 入队列

- 循环拉取所有分页的二手车数据
- 循环拉取队列中城市的二手车数据
- 等待，确定队列中无新的 URL
- 爬取的二手车数据入库

获取城市

通过页面查看，可发现在城市筛选区可得到全部的二手车城市列表，但是你仔细查阅代码。会发现它是 JS 加载进来的，城市也统一放在了一个变量中

```
                    expireHours: 36000 * 24,
                    domain: ".autohome.com.cn"
                };
            this.set = function (newCityId, oldCityId) {

                autoHeaderObj.cookies.del("cityId");
                autoHeaderObj.cookies.del("cookieCityId");
                autoHeaderObj.cookies.set("cityId", oldCityId, expire);
                autoHeaderObj.cookies.set("cookieCityId", newCityId, expire);

            };
        }
    };

    var autoHeaderObj = new AutoHeader();
    var areaJson = [{ "Id": 340000, "Name": "安徽", "FirstCharacter": "A", "Pinyin": "anhui", "City": [{ "Id": 340100, "Name": "合肥", "FirstChar
"Name": "芜湖", "FirstCharacter": "W", "Pinyin": "wuhu", "OldCityId": 9 }, { "Id": 340800, "Name": "安庆", "FirstCharacter": "A", "Pinyin": "anqi
"FirstCharacter": "F", "Pinyin": "fu_yang", "OldCityId": 4 }] }, { "Id": 110000, "Name": "北京", "FirstCharacter": "B", "Pinyin": "beijing", "Cit
"Pinyin": "beijing", "OldCityId": 646 }] }, { "Id": 500000, "Name": "重庆", "FirstCharacter": "C", "Pinyin": "chongqing", "City": [{ "Id": 500100
"OldCityId": 648 }] }, { "Id": 350000, "Name": "福建", "FirstCharacter": "F", "Pinyin": "fujian", "City": [{ "Id": 350100, "Name": "福州", "First
350200, "Name": "厦门", "FirstCharacter": "X", "Pinyin": "xiamen", "OldCityId": 23 }, { "Id": 350500, "Name": "泉州", "FirstCharacter": "Q", "Pir
"甘肃", "FirstCharacter": "G", "Pinyin": "gansu", "City": [{ "Id": 620100, "Name": "兰州", "FirstCharacter": "L", "Pinyin": "lanzhou", "OldCityId
"Pinyin": "guangdong", "City": [{ "Id": 440100, "Name": "广州", "FirstCharacter": "G", "Pinyin": "guangzhou", "OldCityId": 62 }, { "Id": 440200,
"OldCityId": 65 }, { "Id": 440300, "Name": "深圳", "FirstCharacter": "S", "Pinyin": "shenzhen", "OldCityId": 670 }, { "Id": 440400, "Name": "珠海
"Id": 440500, "Name": "汕头", "FirstCharacter": "S", "Pinyin": "shantou", "OldCityId": 69 }, { "Id": 440600, "Name": "佛山", "FirstCharacter": "F
"江门", "FirstCharacter": "J", "Pinyin": "jiangmen", "OldCityId": 76 }, { "Id": 441300, "Name": "惠州", "FirstCharacter": "H", "Pinyin": "huizhou
"FirstCharacter": "D", "Pinyin": "dongguan", "OldCityId": 73 }, { "Id": 442000, "Name": "中山", "FirstCharacter": "Z", "Pinyin": "zhongshan", "Ol
"FirstCharacter": "G", "Pinyin": "guangxi", "City": [{ "Id": 450100, "Name": "南宁", "FirstCharacter": "N", "Pinyin": "nanning", "OldCityId": 574
"Pinyin": "liuzhou", "OldCityId": 576 }, { "Id": 450300, "Name": "桂林", "FirstCharacter": "G", "Pinyin": "guilin", "OldCityId": 575 }] }, { "Id"
"guizhou", "City": [{ "Id": 520100, "Name": "贵阳", "FirstCharacter": "G", "Pinyin": "guiyang", "OldCityId": 106 }, { "Id": 520300, "Name": "遵义
```

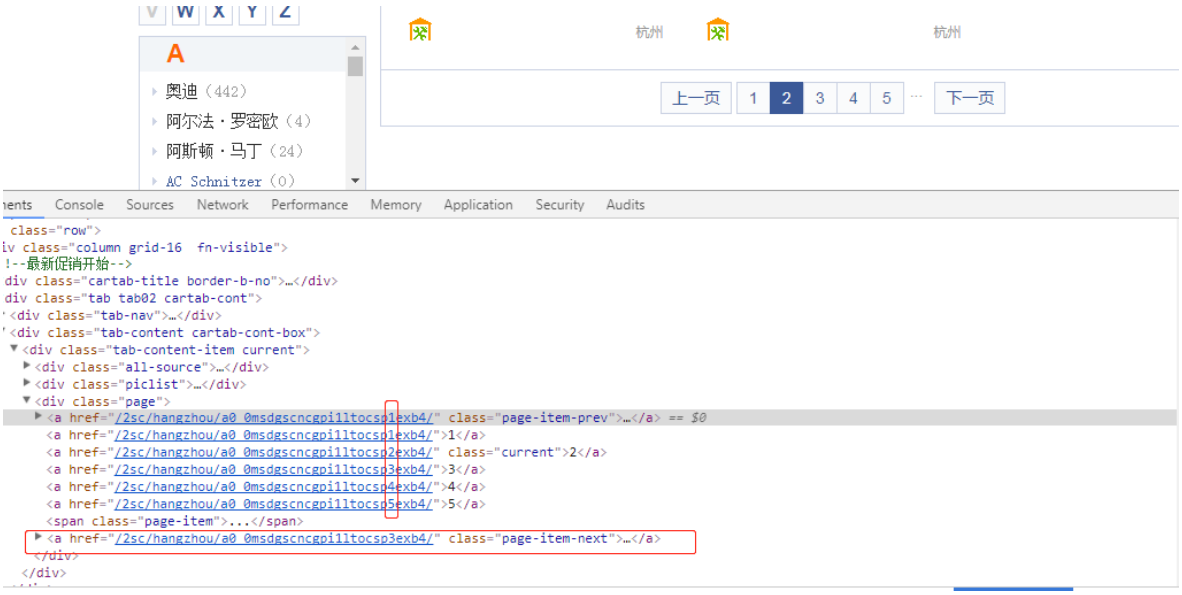
有两种提取方法

- 分析 JS 变量，提取出来

- 直接将 `areaJson` 复制出来作为变量解析

在这里我们直接将其复制粘贴出来即可，因为这是比较少变动的值

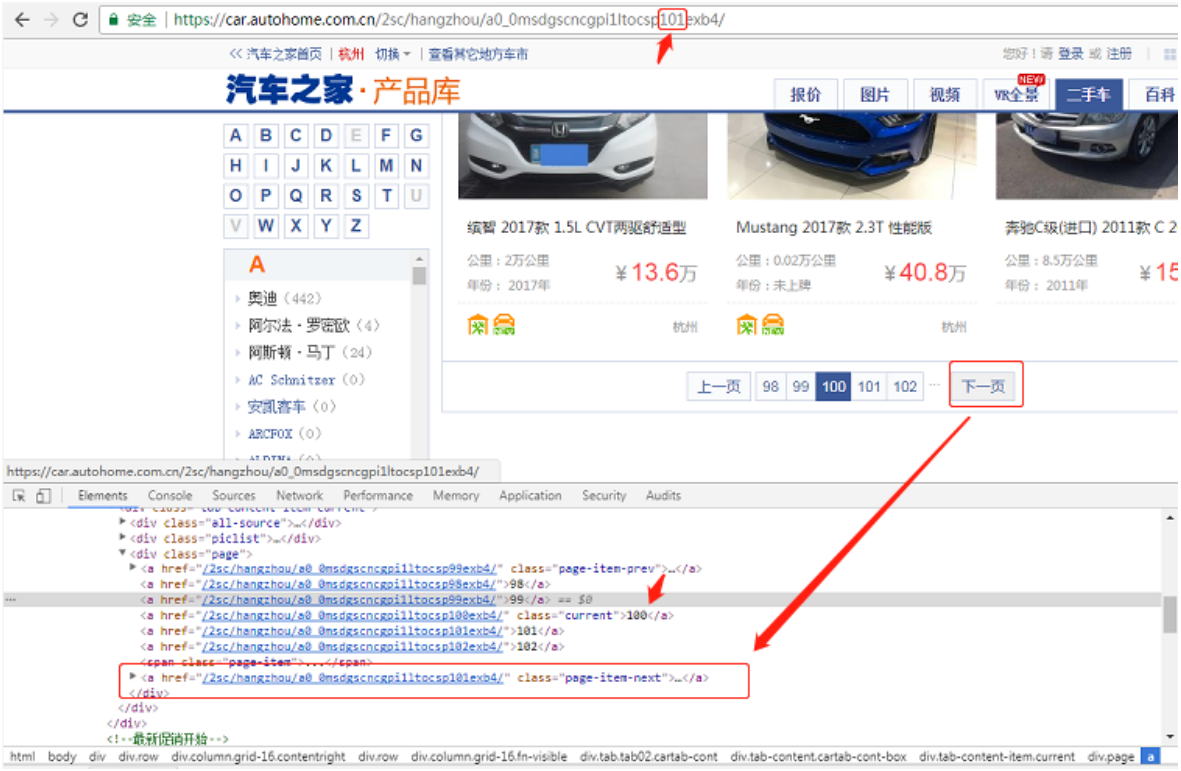
获取分页



通过分析页面可以得知分页链接是有一定规律的，例如：`/2sc/hangzhou/a0_0msdgscncgpi1ltocsp2exb4/`，可以发现 `sp%d`，`sp` 后面为页码

按照常理，可以通过预测所有分页链接，推入队列后 `go routine` 一波 即可快速拉取

但是在这老产品库存在一个问题，在超过 100 页后，下一页永远是 101 页



因此我们采取比较传统的做法，通过拉取下一页的链接去访问，以便适应可能的分页链接改变； 100 页以后的分页展示也很奇怪，先忽视

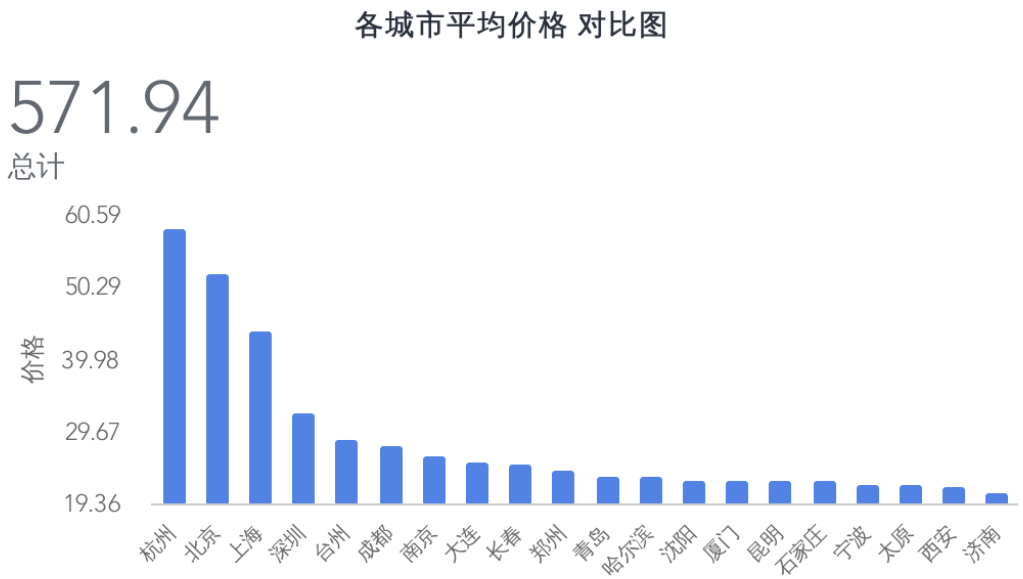
获取二手车数据

页面结构较为固定，常规的清洗 HTML 即可

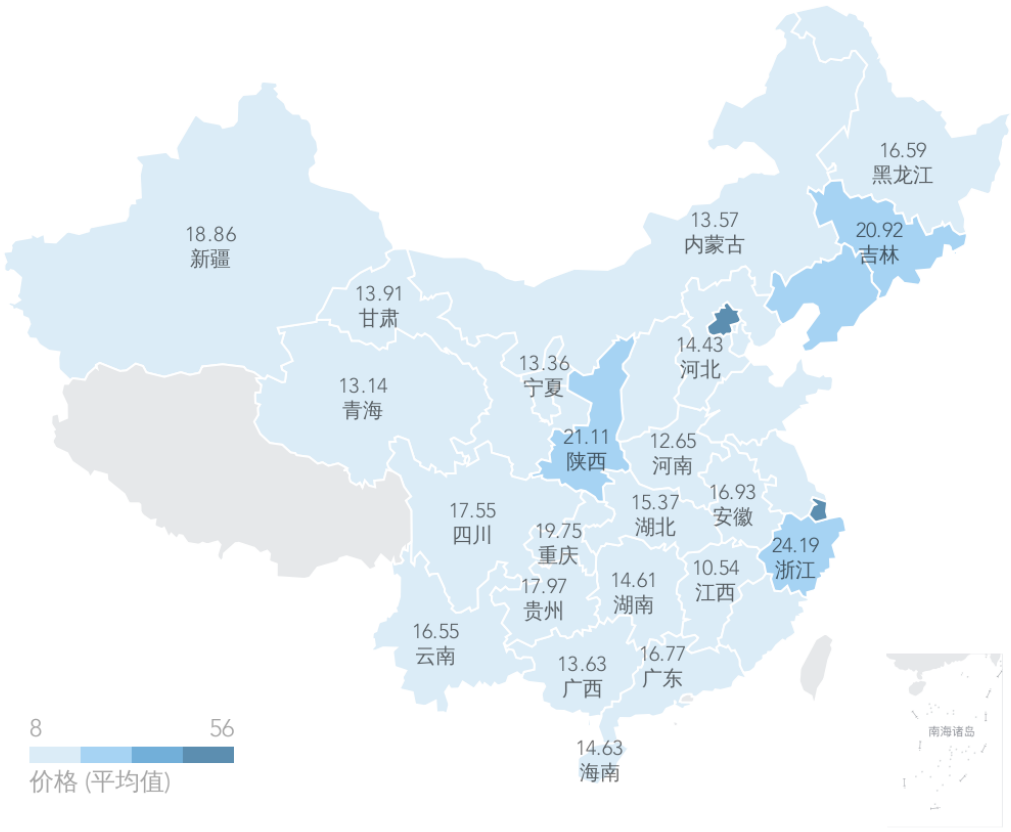
```
func GetCars(doc *goquery.Document) (cars []QcCar) {
    cityName := GetCityName(doc)
    doc.Find(".piclist ul li:not(.line)").Each(func(i int, selection *goquery.Selection) {
        title := selection.Find(".title a").Text()
        price := selection.Find(".detail .detail-r").Find(".colf8").Text()
        kilometer := selection.Find(".detail .detail-l").Find("p").Eq(0).Text()
        year := selection.Find(".detail .detail-l").Find("p").Eq(1).Text()

        kilometer = strings.Join(compileNumber.FindAllString(kilometer, -1), "")
        year = strings.Join(compileNumber.FindAllString(strings.TrimSpace(year), -1), "")
        priceS, _ := strconv.ParseFloat(price, 64)
        kilometerS, _ := strconv.ParseFloat(kilometer, 64)
        yearS, _ := strconv.Atoi(year)

        cars = append(cars, QcCar{
            CityName: cityName,
```

```
            Title: title,
            Price: priceS,
            Kilometer: kilometerS,
            Year: yearS,
        })
    })

    return cars
}
```

数据

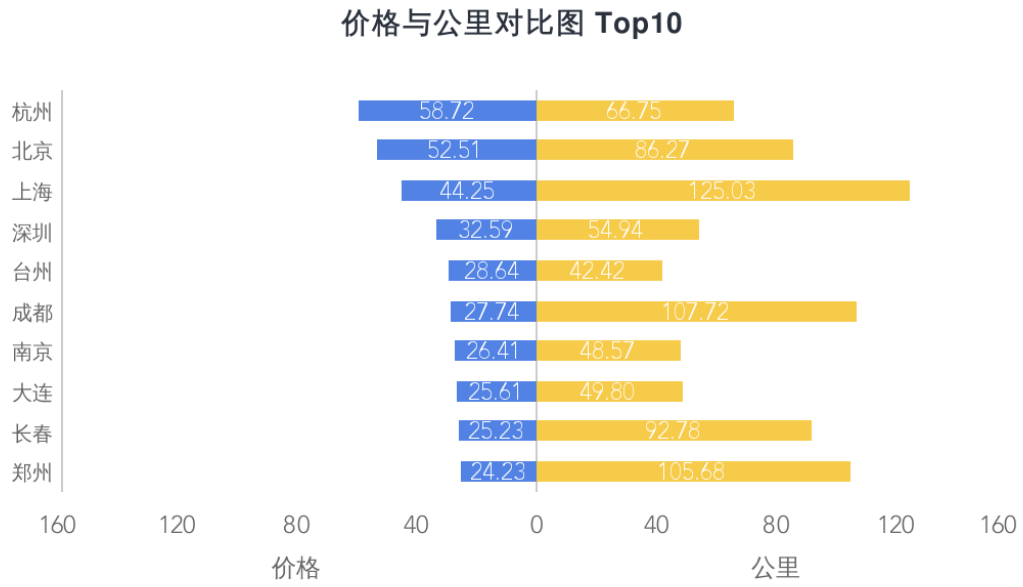


全国各省价格平均值对比

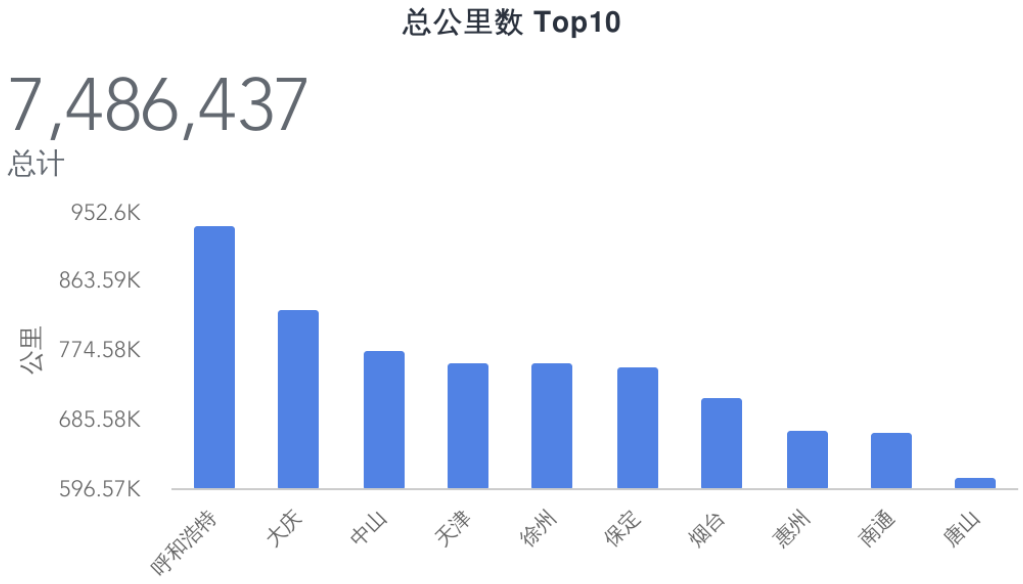


在各城市的平均价格对比中，我们可以发现北上广深里的北京、上海、深圳都在榜单上，而近年势头较猛的杭州直接占领了榜首，且后几名都有一些距离

而其他城市大致都是梯级下降的趋势，看来一线城市的二手车也是不便宜了，当然这只是均价



我们可以看到价格和公里数的对比，上海、成都、郑州的等比差异是有点大，感觉有需求的话可以在价格和公里数上做一个衡量



这图有点儿有趣，粗略的统计了一下总公里数。在前几张图里，平均价格排名较高的统统没有出现在这里，反倒是呼和浩特、大庆、中山等出现在了榜首

是否侧面反应了一线城市的车辆更新换代较快，而较后的城市的车辆倒是换代较慢，公里数基本都杠杠的

词云

通过对标题的分析，可以得知车辆产品库的命名基本都是品牌名称+自动/手动+XXXX 款+属性，看标题就能知道个概况了

参考

爬虫项目地址

- https://github.com/go-crawler/car-prices

了解一下Golang的市场行情

项目地址：https://github.com/go-crawler/lagou_jobs

如果对你有所帮助，欢迎 Star，给文章来波赞，这样可以让更多的人看见 :)

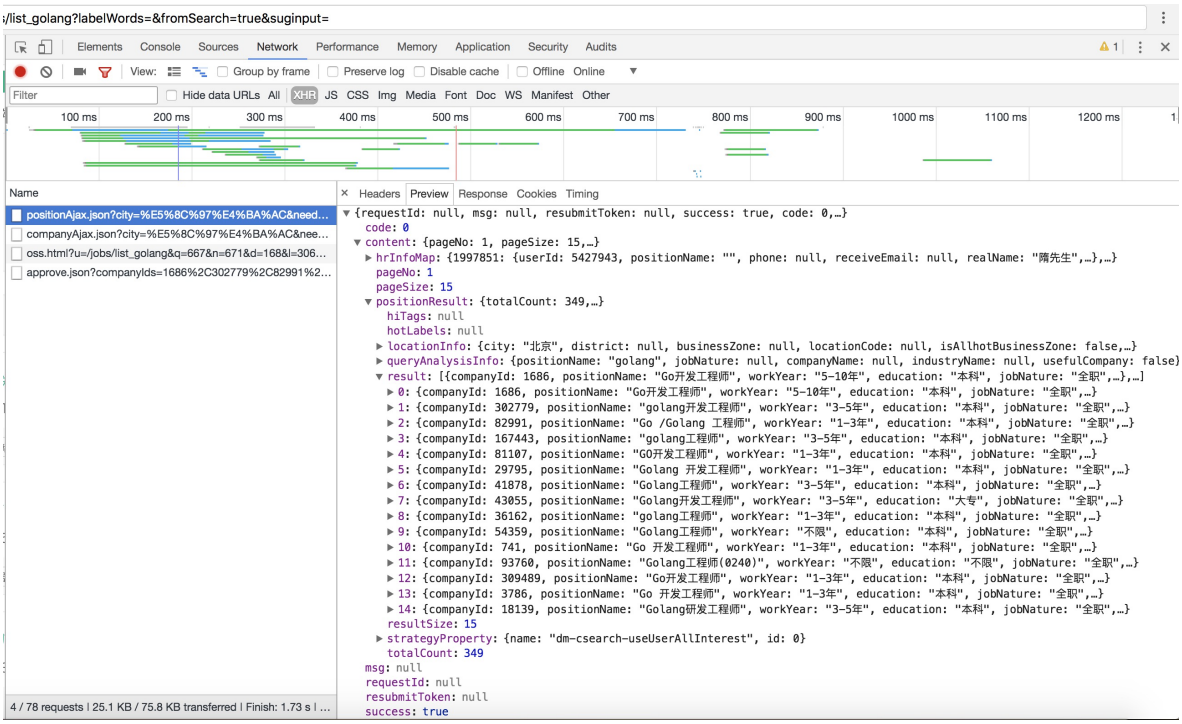
目标

在工作中 Golang 已是一份子，想让大家了解一下 Golang 的市场行情，也想让更多的人熟悉它。因此主要是展示数据分析的结果

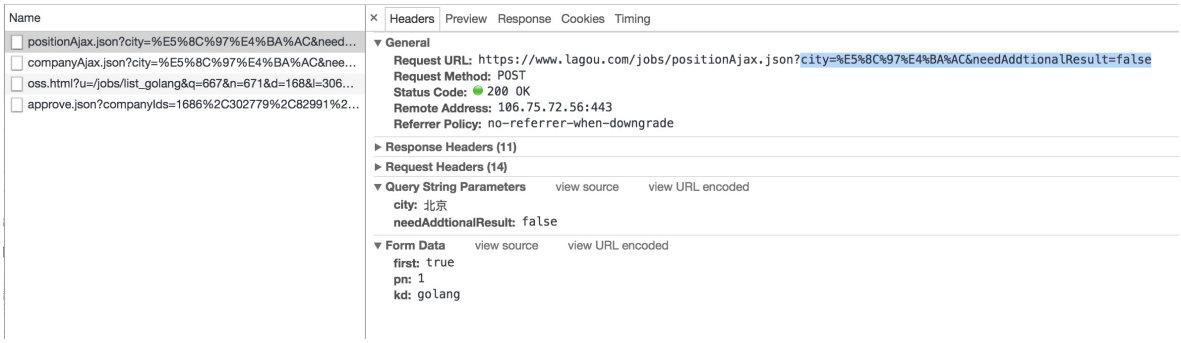
目标站点是 某招聘网站 的职位数据抓取和分析，爬取城市分别为 北京、上海、广州、深圳、杭州、成都，再得出一个结论

分析

首先需要进行页面分析，找到我们的抓取方向



搜索 golang 关键字，打开页面 F12 就能看到它发送了四个请求，留意 positionAjax.json 这个请求



我们仔细研判这个接口的入参和出参

入参

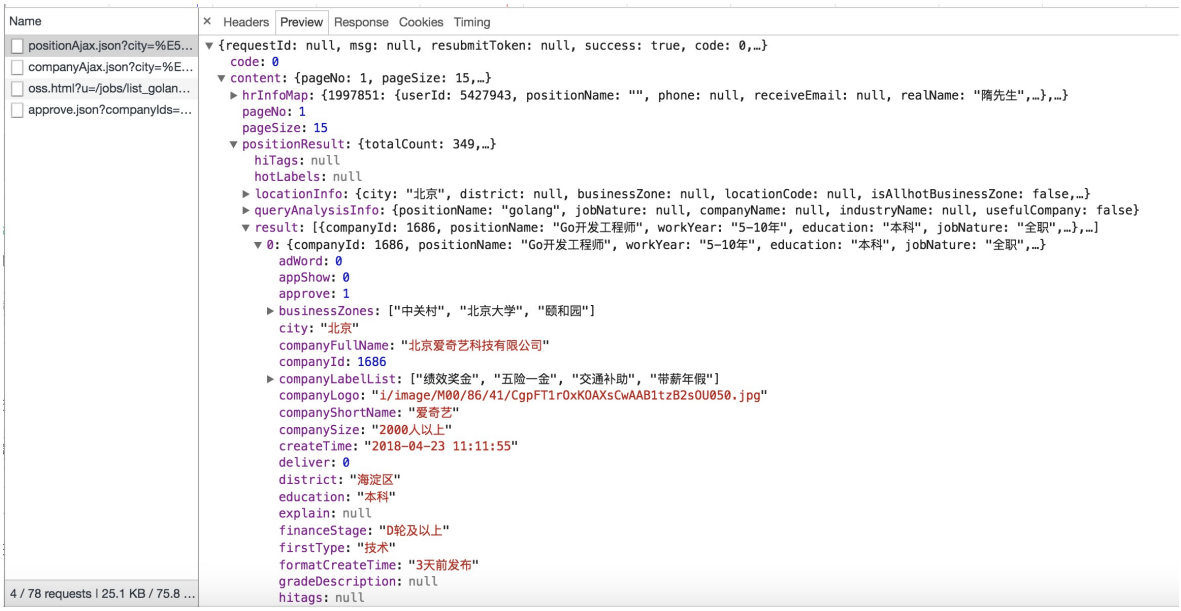
1、Query String Param

- city：请求的城市
- needAddtionalResult：是否需要补充额外的参数，这里默认 false

2、Form Data

- first：是否首页
- pn：页码
- kd：关键字

出参



就是它了，从返回结果可得出许多有用的信息

- companyFullName：公司全称
- companyLabelList：公司标签
- companyShortName：公司简称
- companySize：公司规模
- education：学历要求
- financeStage：融资阶段

等等~

分页

在上面两张图中，可以发现在 content 节点中包含 pageNo、pageSize 字段，content.positionResult 节点有 totalCount 字段，可以得知当前是第几页，每页显示多少条，当前的职位总条数

需要注意一下，分页的计算是要向上取整的

模拟浏览器头

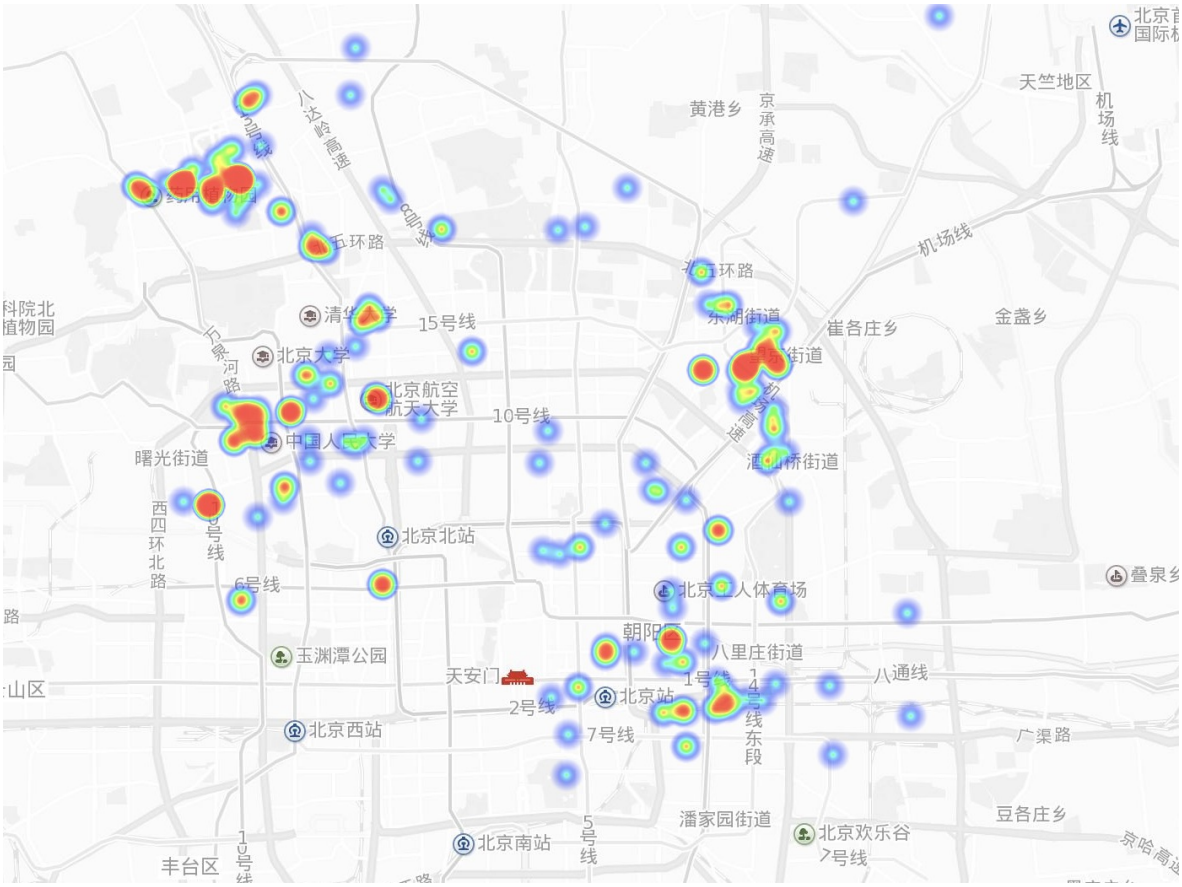
User-Agent 可以用 fake-useragent 这个项目来随机生成 UA 头 😄

数据

一、分布图

不同工作、工种，自然也会遍布在不同的工作区域，我们先了解一下各个城市的 Golang 工程师都主要在哪个区上班，心里留个底

北京


北京首都国际机场
天竺地区
机场线
黄港乡
京承高速
五环路
机场线
北京大学
清华大学
15号线
东湖街道
崔各庄乡
金盏乡
北京航空航天大学
10号线
中国人民大学
曙光街道
酒仙桥街道
西四环北路
9号线
北京北站
6号线
叠泉乡
北京工人体育场
朝阳门
八里庄街道
玉渊潭公园
天安门
1号线
八通线
2号线
北京站
北京西站
7号线
广渠路
潘家园街道
豆各庄乡
北京南站
5号线
北京欢乐谷
丰台区
京哈高速

上海


罗店镇
宝山区
潘石镇
长兴乡
小农场
嘉定镇
马陆镇
外冈镇
刘行镇
方泰镇
安亭镇
黄渡镇
华新镇
普陀区
江桥镇
虹口区
曹路镇
龚路镇
陆行镇
白龙港镇
长宁区
蔡路镇
华漕镇
唐镇
川沙新镇
徐汇区
机场镇
六团社区
九亭镇
凤凰山镇
闵行区
六灶镇
佘山镇
曹行镇
陈行镇
浦江镇
东海
洞泾镇
坦直镇
三灶镇
广富林镇
新桥镇
塘湾镇
惠南镇
松江区
航头镇

广州


蚌湖镇
太和镇
镇龙镇
鹤龙街道
里水镇
云城街道
龙洞街道
永和镇
萝岗镇
白云区
新塘街道
广州市
天河区
大沥镇
多宝街道
黄埔区
荔联街道
南岗镇
新窖镇
新造镇
佛山市
南村镇
石壁街道
莲花山镇
乐从镇
陈村镇
祺丰农场
番禺区

深圳


龙华区
凤岗
龙岗区
坪山区
坪山
庆丰农场
观澜街道
深圳宝安
国际机场
大塘农作社
布吉街道
梅沙街道
山城好面道
西丽
东湖街道
盐田街道
盐田区
深圳市
罗湖区
福田区
元朗区
大埔区

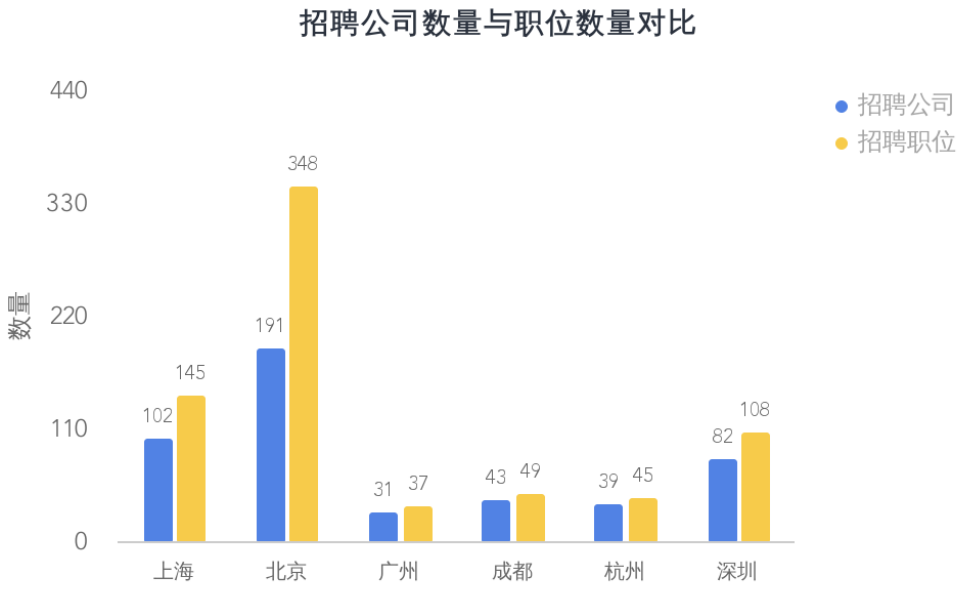
杭州


长安镇
安溪镇
余杭区
许村镇
许巷乡
星桥镇
崇贤镇
良渚镇
瓶窑镇
康桥镇
丁兰街道
乔司镇
拱墅区
彭埠镇
下城区
西湖区
杭州市
南阳镇
滨江区
新街镇
坎山镇
瓜沥镇
长河镇
闻堰镇
双浦镇
所前镇
义桥镇
夏履镇
柯桥区
富阳区
里山镇
临浦镇

成都


犀浦东站
凤凰山公园
木兰镇
四川大学
锦城学院
3号线
青龙街道
龙潭街道
成都站
成都西站
青羊区
成都理工大学
4号线
宽窄巷子
成华区
成都市
永康森林公园
西河镇
四川大学
3号线
四川大学
成都东站
成都南站
1号线
2号线
成都信息
工程大学
双流区
学生东园
学生西园7舍围合
双流西站
蓉遵高速
新兴镇
维也纳森林公园
四川现代
职业学院
南湖梦幻岛
体验公园
黄甲镇
万安镇
公兴站

二、招聘与职位数量对比



通过分析图中的数据，我们可以得知各城市的招聘职位数量

- 北京：348
- 上海：145
- 广州：37
- 成都：49
- 杭州：45
- 深圳：108

总共招聘的职位数量为 732 个，数量顺序分别为 北京 > 上海 > 深圳 > 成都 > 杭州 > 广州

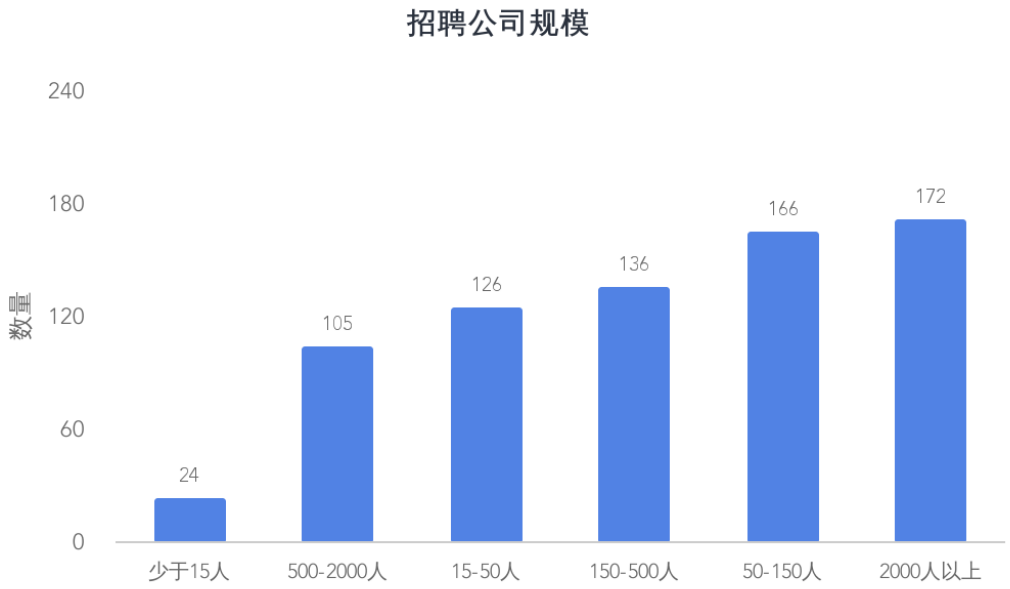
还有另外一个关注点，就是招聘公司数量与职位的数量对比，可以看到 北京 招聘的职位数量为 348 个，而招聘的公司数量为 191 个，约为 1.82 的比例，也就是一家公司能提供两个 Golang 职位，它可能类别不同、（中级、中高级、高级）级别不同，具有一定可能性。而在广州，为 31 对比 37，虽然差额不大，但仍然存在这种现象

可以得出结果，Golang 在市场上具有一定的伸缩空间，也就是具有上升空间，一家公司会将 Golang 应用在多个不同的应用场景，也就是方向不同，需要的级别人才也就不同了

但是需要注意的是，Golang 的市场招聘人数目前份额还是较低，六个城市总数仅为 732 个，与其他大热语言相差有一定距离，需要谨慎

同时，面试 Golang 的人与其他大热语言相比会少些，职位的争夺是否小点呢？

三、招聘公司规模



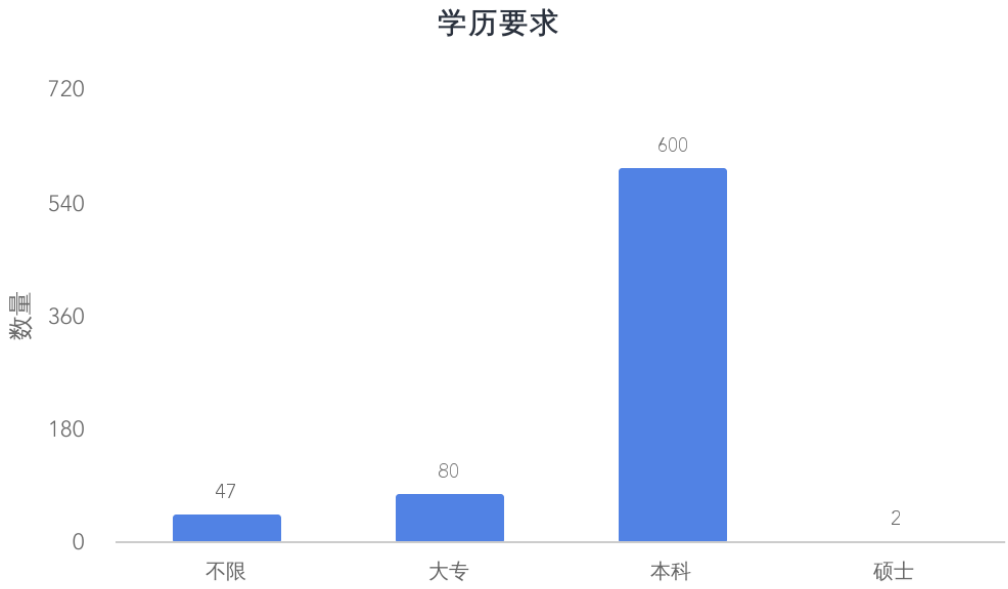
通过查看招聘 Golang 工程师的公司规模，可以很直观的发现，微型公司使用 Golang 较少，其他类别的规模都有一定程度的应用，且差距不大。在 2000 人以上、50 - 150 人的公司规模中最受青睐

为什么呢，我认为有以下可能

- 大型公司结合场景，想通过 Golang 的特性来解决一些痛点问题
- 在小型公司 Golang 这颗新星实施起来更便捷，有一定的应用场景

你觉得呢，是不是应该有更多的选择它的原因？

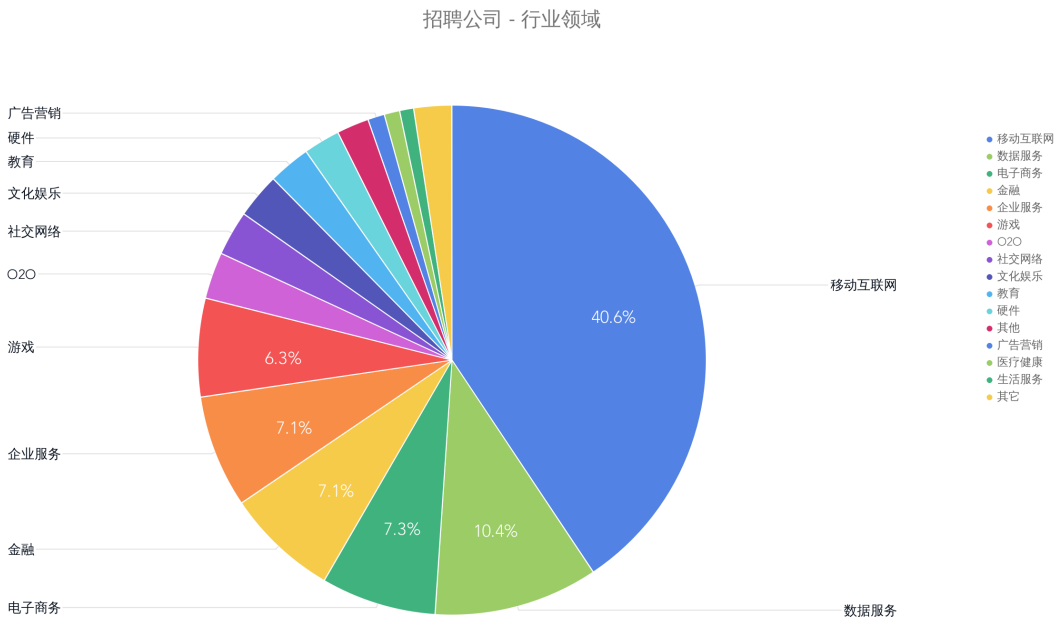
四、学历要求



在招聘市场上，Golang 的招聘者更希望你是本科学历，大专和不限也有一定的份额，但市场份额相差较大

硕士学历要求的为两个，可以得出，在市场上 Golang 招聘者们对高学历的需求并不高，或者并不强制高学历

五、行业领域



在这里，重点关注 Golang 工程师的招聘公司都分别在什么行业领域，大头移动互联网是不容置疑的了，还可以惊喜的发现

- 数据服务
- 电子商务
- 金融
- 企业服务
- 游戏

Golang 在这几个方面都有所应用，说明了在市场上，Golang 的路子是比较广阔的，前景不错

同时，如果可以涉及多个领域的内容，想必身为工程师的你，肯定很激动

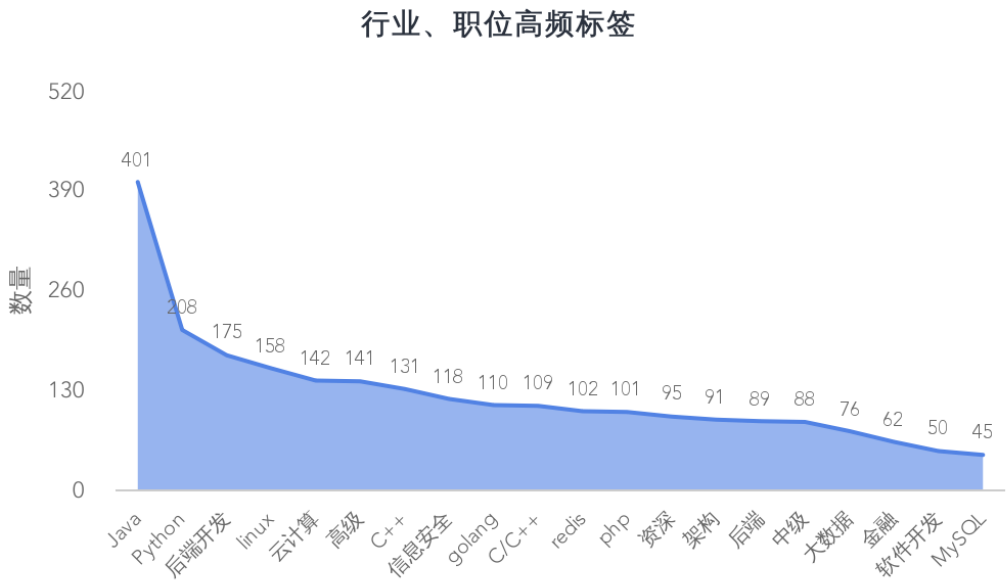
六、职位诱惑

职位诱惑

职位诱惑是投简历时必看的一点了，可以看到高频词条基本都是 IT 从业者关心的话题了，这里你懂的...

重点，我看到了一个“免费三餐”的词条命中 7 次，分别来自北京的海淀区、东城区、朝阳区，上海的黄浦区的七家不同的公司，辛苦了

七、行业、职位标签



在招聘 JD 中，描述和标签常用于给求职者了解这一职业的具体工作内容和其关联性

在图中你可以看到 Golang 常常和什么内容搭上边，这点很有意义哦

1、语言

- Java
- Python
- C/C++
- PHP

在图中可以看出，Golang 与以上四种语言有一定关联性，而 Java 和 Python 分别第一、第二名，可以说明市场上对复合型人才的渴望度更高，也许你不懂也行，但你懂了就最好（加分项）。需要你自身有多语言的经验，也便于和其他人对接

同时 Golang 目前存在许多内部转语言写的情况，所以这一点可以参考

2、职称

- 高级
- 资深

- 中级

特意将职称放在第二位，可以发现在市场上 Golang 标签的需求是 高级 > 资深 > 中级，关联第一项“语言关联”不难得出这个结论，因为语言只是解决问题的工具，到了中级及以上的工程师都是懂多门语言的居多，再采取不同的方案去解决应用场景上的问题

可得出结论，市场目前对 Golang 更期望是中高、高级、资深的人才，而中级的反而少一点点

大家可以努力再往上冲击冲击

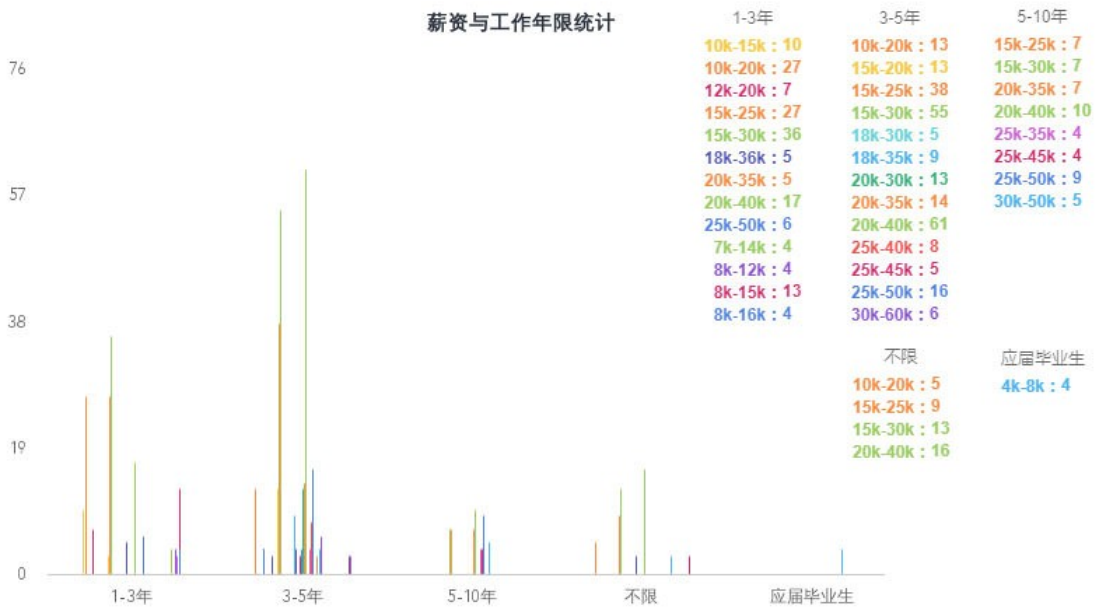
3、组件

- Linux
- Redis
- Mysql

4、行业

- 云计算
- 信息安全
- 大数据
- 金融
- 软件开发

八、薪资与工作年限



1、1-3 年

一个（成长）特殊的阶段，有个位数也有双位数的，大头可以到 15-30k，20-40k，而初级的也有 8-16k

2、3-5 年

厚积待发的阶段，薪酬范畴的跨度是较大，10-60k 的薪酬都有，这充分说明能力决定你的上下

3、5-10 年

核心，招聘网站上的招聘数量反而少，都会走内推或猎头，不需要特别介绍了

小结

这一部分，相信是很多人关注的地方

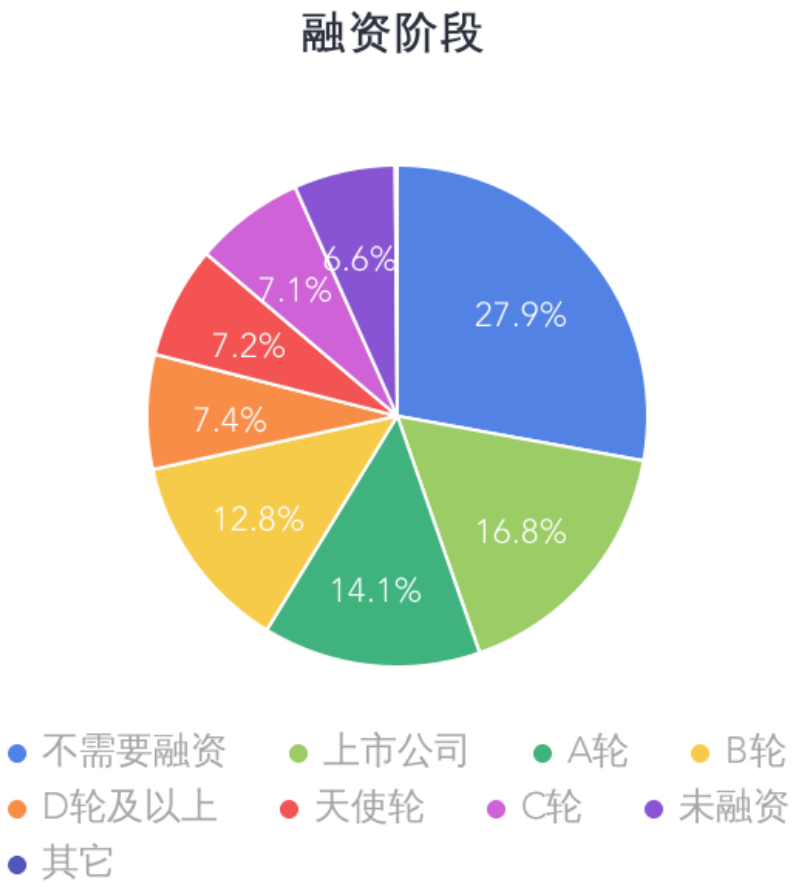
在有的文章中会看到，他们的薪资部分是以平均值来展示的。我就很纳闷，因为对平均值并不是很关心，**重点是无法体现薪资幅度**。因此这里我会尽可能的把数据展现给你们看

（正文）从图表来看，Golang 当前的薪酬水平还是很不错的，市场能根据不同阶段（水平）的人给出一个好的价位

（题外话）看完之后希望你能知道以下内容

- 你当前工作年限的最高、最低薪资范畴
- 你的下一阶段的薪资范畴
- 为什么有的人高，有的人低
- 在大头部队还是小头，为什么
- 不要满足于平均值

九、融资阶段

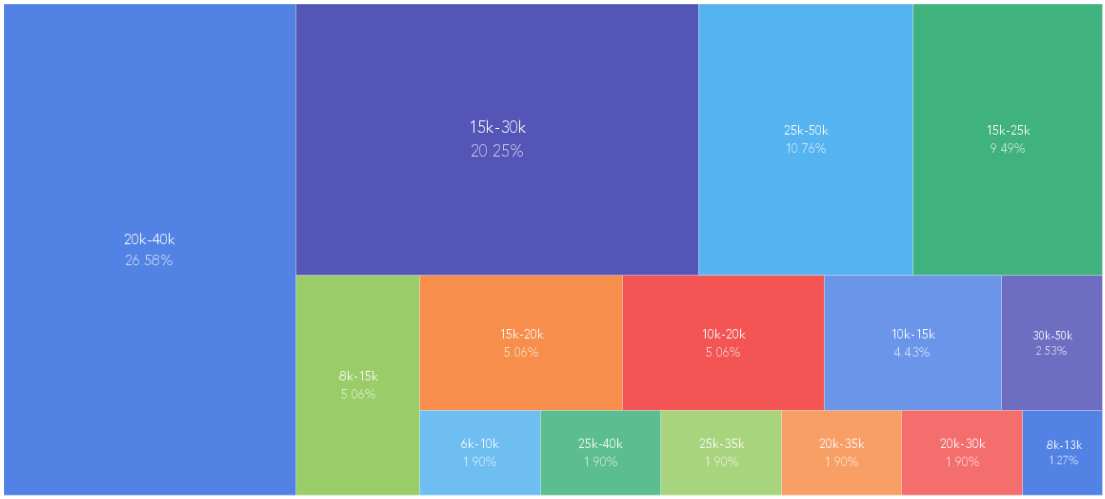


选用 Golang 的公司大多数都较为稳定，有一部分比较刺激 :)

融资阶段与薪资范畴对比

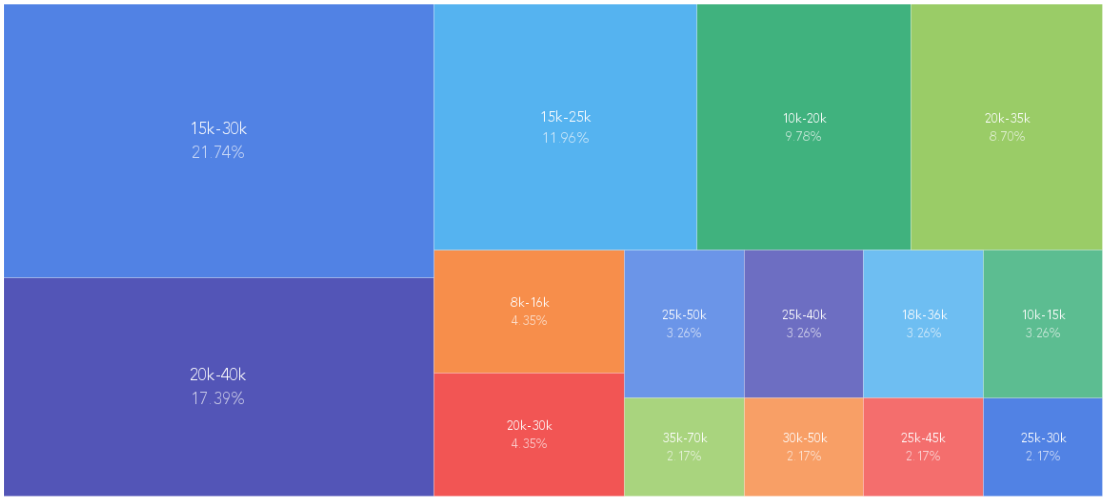
不需要融资

各融资阶段的职位工资比例



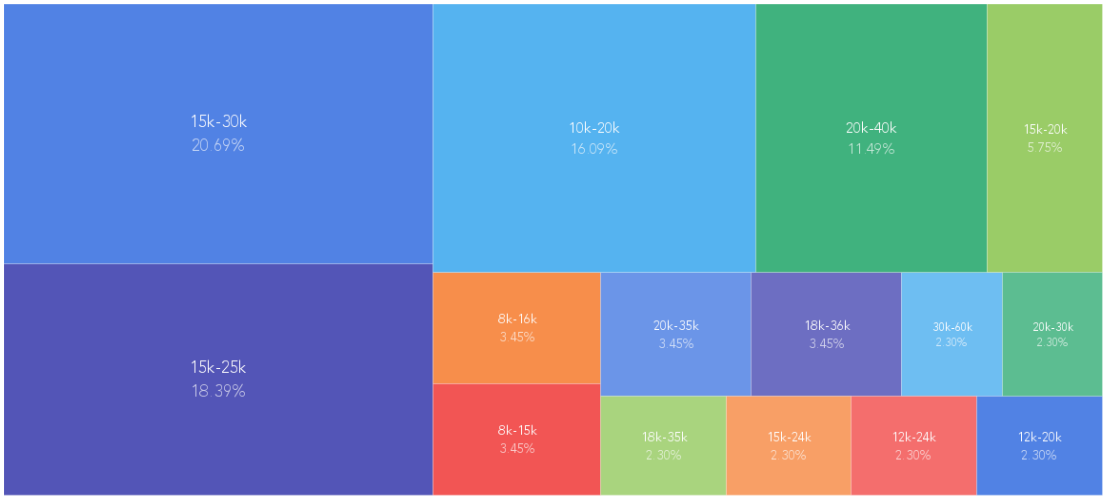
上市公司

各融资阶段的职位工资比例



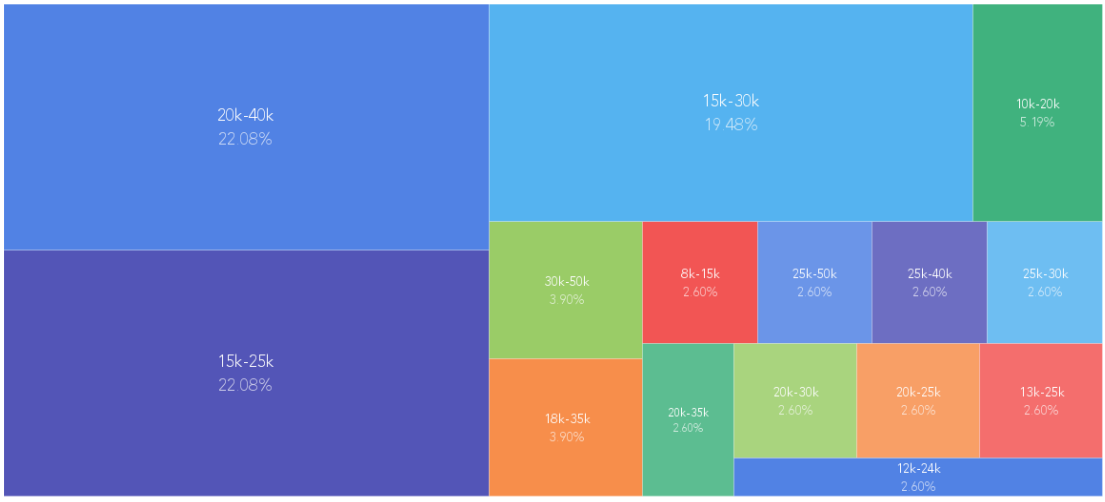
A 轮

各融资阶段的职位工资比例

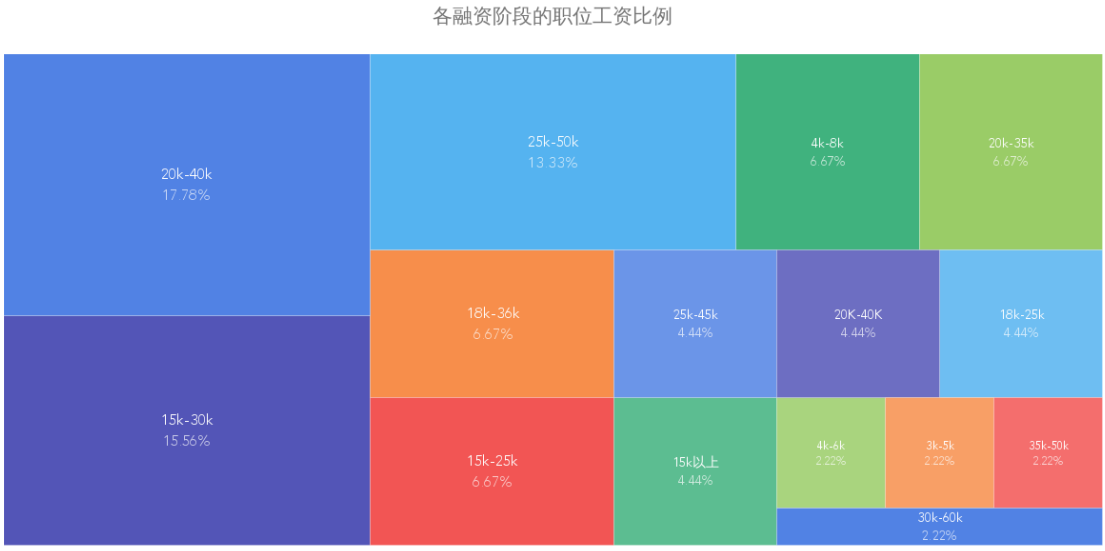


B 轮

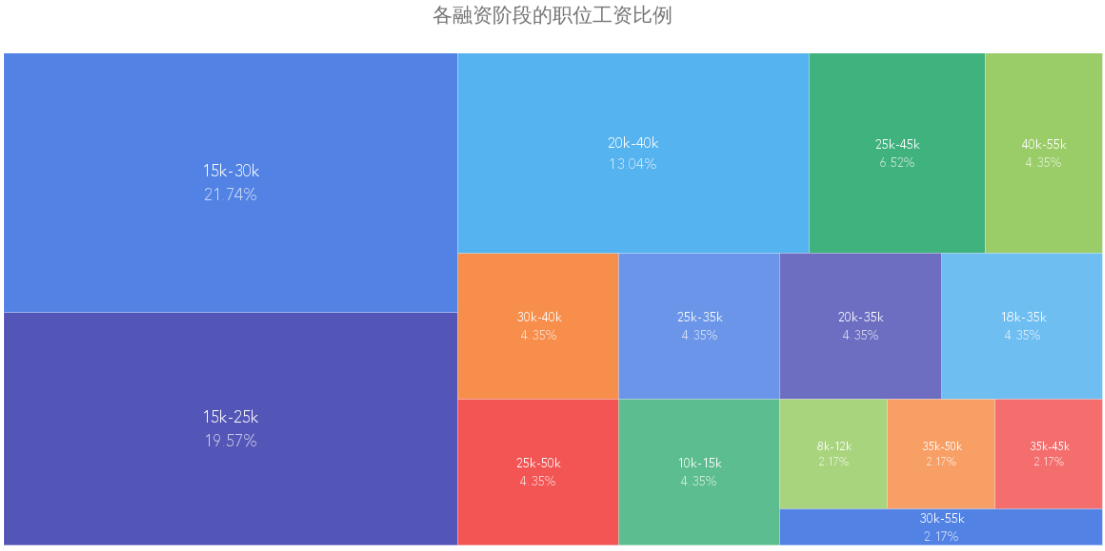
各融资阶段的职位工资比例



C 轮



D 轮以上

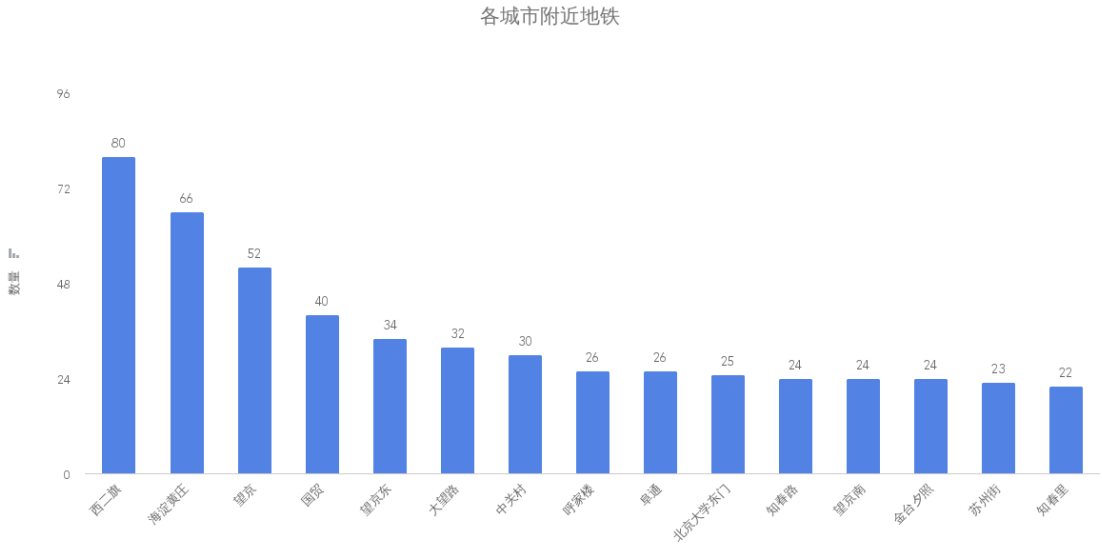


十、附近的地铁

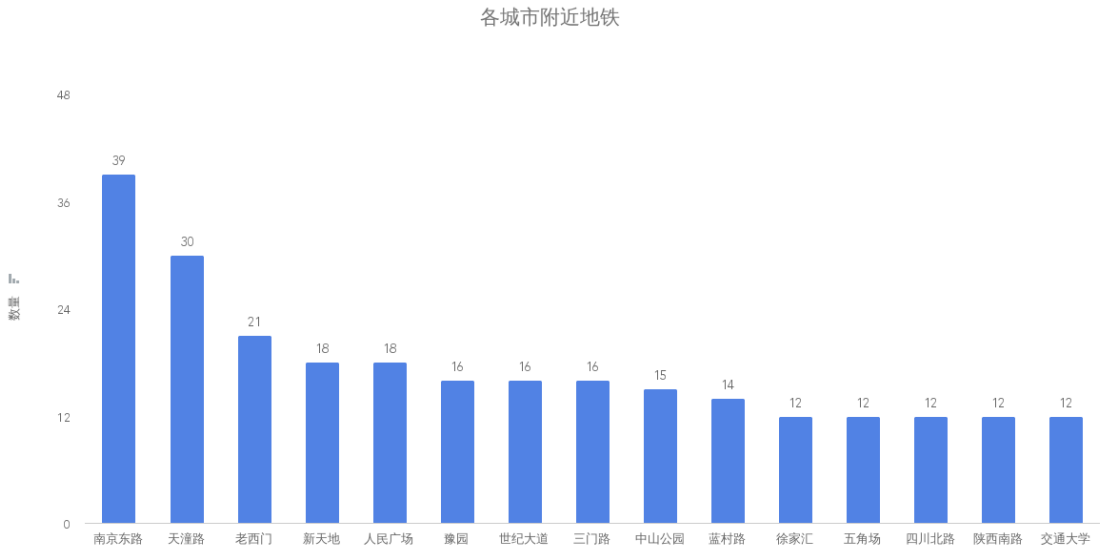
Golang 工程师都驻扎在什么地铁站附近呢

经常在地铁上看到同行在看代码，来了解一下都分布在哪 :)

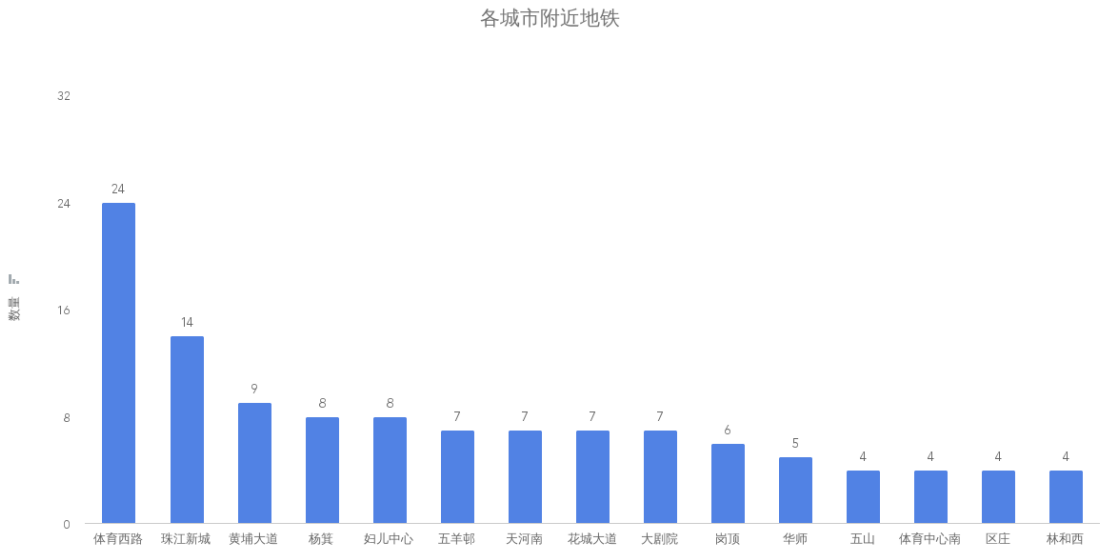
北京


各城市附近地铁
数量
96
72
48
24
0
80
66
52
40
34
32
30
26
26
25
24
24
24
23
22
西二旗
海淀黄庄
望京
国贸
望京东
大望路
中关村
呼家楼
阜通
北京大学东门
知春路
望京南
金台夕照
苏州街
知春里

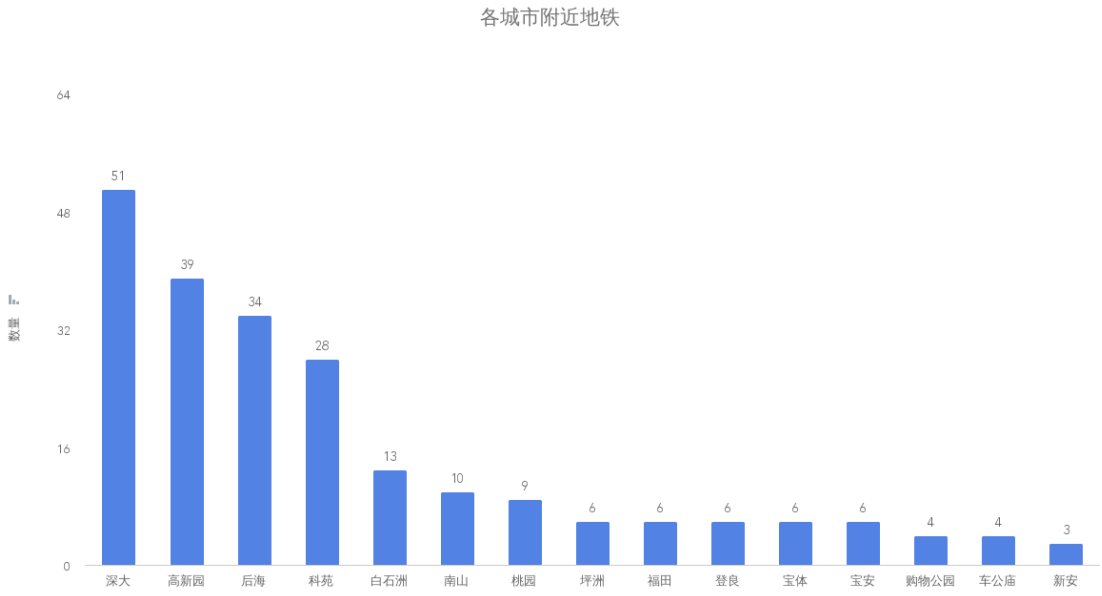
上海


各城市附近地铁
数量
48
36
24
12
0
39
30
21
18
18
16
16
16
15
14
12
12
12
12
12
南京东路
天潼路
老西门
新天地
人民广场
豫园
世纪大道
三门路
中山公园
蓝村路
徐家汇
五角场
四川北路
陕西南路
交通大学

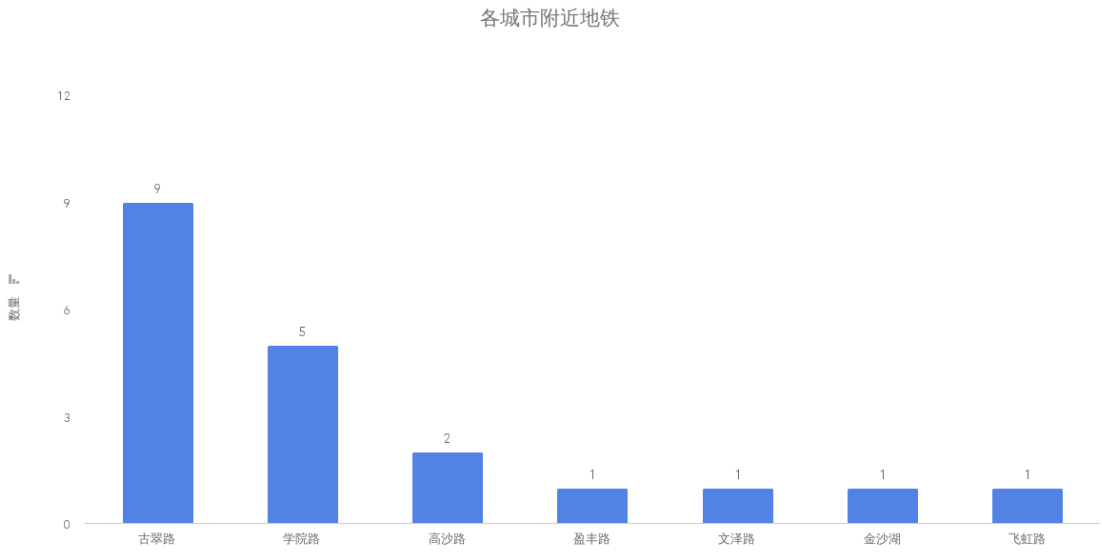
广州


各城市附近地铁
数量
32
24
16
8
0
24
14
9
8
8
7
7
7
7
6
5
4
4
4
4
体育西路
珠江新城
黄埔大道
杨箕
妇儿中心
五羊邨
天河南
花城大道
大剧院
岗顶
华师
五山
体育中心南
区庄
林和西

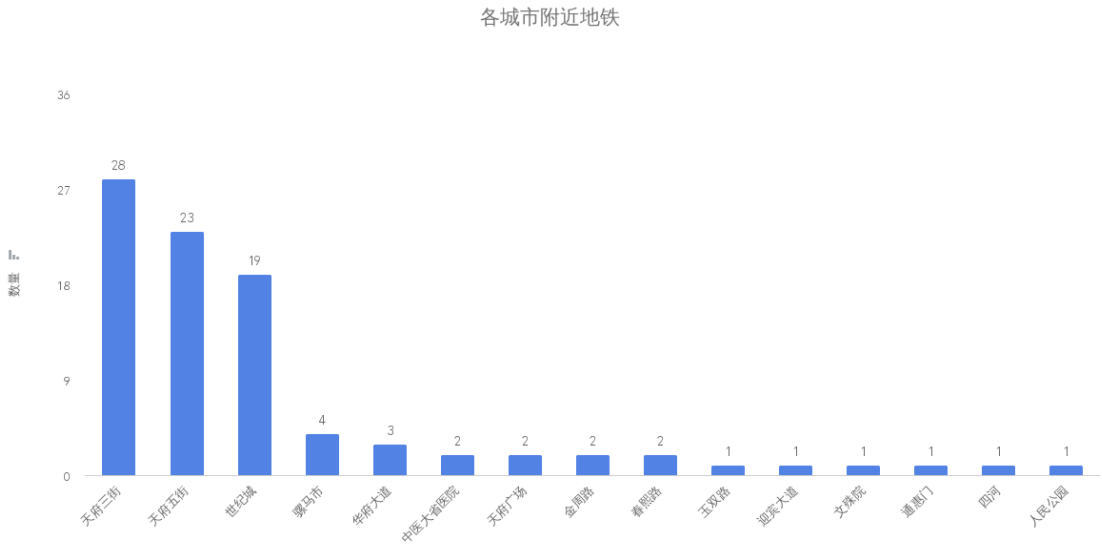
深圳


各城市附近地铁
数量
64
48
32
16
0
51
39
34
28
13
10
9
6
6
6
6
6
4
4
3
深大
高新园
后海
科苑
白石洲
南山
桃园
坪洲
福田
登良
宝体
宝安
购物公园
车公庙
新安

杭州



成都



结论

如同官方所说 “Go has been on an amazing journey over the last 8+ years”，作为一门新生语言，一直在不断地发展，**缺点肯定是有的，你要去识别它**

从数量来看

单从这个招聘网站上来看，数量方面，与大热语言的招聘职位数量仍然有一定的差距，但 Golang 存在许多内部转语言开发的情况，当前展现出来的数据，**招聘数量不多，但质量不错**

从分布图来看

一线城市基本都有 Golang 的职位，虽然其他城市较少，但对于新语言来说是需要持续关注的过程，不能一刀切

从职称级别来看

Golang 中高、高级、资深仍然是占大头，给的薪资也基本符合市场行情

从方向来看

Golang 涉及的行业领域广泛，移动互联网、数据服务、电子商务、金融、企业服务、云计算等都是它的战场之一

从开源项目来看

docker、k8s、etcd、consul 都挺稳

总的来说，Golang 处于一个发展的阶段，市场行情也还行、应用场景较广，不过招聘数量不多，你又怎么看呢？

最后放上今天新发布的 Logo :)

如果对你有所帮助，欢迎 Star，给文章点个赞，这样可以让更多的人看见这篇文章

参考

- 项目地址：https://github.com/go-crawler/lagou_jobs

go-moduels

干货满满的 Go Modules 和 goproxy.cn

Go Modules 终极入门

干货满满的 Go Modules 和 goproxy.cn

大家好，我是一只普通的煎鱼，周四晚上很有幸邀请到 goproxy.cn 的作者 @盛傲飞（@aofei） 到 Go 夜读给我们进行第 61 期 《Go Modules、Go Module Proxy 和 goproxy.cn》的技术分享。

本次 @盛傲飞 的夜读分享，是对 Go Modules 的一次很好的解读，比较贴近工程实践，我必然希望把这块的知识更多的分享给大家，因此有了今天本篇文章，同时大家也可以多关注 Go 夜读，每周会通过 zoom 在线直播的方式分享 Go 相关的技术话题，希望对大家有所帮助。

前言

Go 1.11 推出的模块（Modules）为 Go 语言开发者打开了一扇新的大门，理想化的依赖管理解决方案使得 Go 语言朝着计算机编程史上的第一个依赖乌托邦（Deptopia）迈进。随着模块一起推出的还有模块代理协议（Module proxy protocol），通过这个协议我们可以实现 Go 模块代理（Go module proxy），也就是依赖镜像。

Go 1.13 的发布为模块带来了大量的改进，所以模块的扶正就是这次 Go 1.13 发布中开发者能直接感觉到的最大变化。而问题在于，Go 1.13 中的 GOPROXY 环境变量拥有了一个在中国大陆无法访问到的默认值 `proxy.golang.org` ，经过大家在 golang/go#31755 中激烈的讨论（有些人甚至将话提上升到了“自由世界”的层次），最终 Go 核心团队仍然无法为中国开发者提供一个可在中国大陆访问的官方模块代理。

为了今后中国的 Go 语言开发者能更好地进行开发，七牛云推出了非营利性项目 `goproxy.cn` ，其目标是为中国和世界上其他地方的 Gopher 们提供一个免费的、可靠的、持续在线的且经过 CDN 加速的模块代理。可以预见未来是属于模块化的，所以 Go 语言开发者能越早切入模块就能越早进入未来。

如果说 Go 1.11 和 Go 1.12 时由于模块的不完善你不愿意切入，那么 Go 1.13 你则可以大胆地开始放心使用。本次分享将讨论如何使用模块和模块代理，以及在它们的使用中会常遇见的坑，还会讲解如何快速搭建自己的私有模块代理，并简单地介绍一下七牛云推出的 `goproxy.cn` 以及它的出现对于中国 Go 语言开发者来说重要在何处。

目录

    - 坑 3:从 dep、glide 等迁移至 Go Modules
    - 坑 4:拉取私有模块
    - 坑 5:更新现有的模块
    - 坑 6:主版本号
- Go Module Proxy 简介
- Goproxy 中国(goproxy.cn)

Go Modules 简介

Go modules (前身 vgo) 是 Go team (Russ Cox) **强推**的一个**理想化**的**类语言级**依赖管理解决方案，它是和 Go1.11 一同发布的，在 Go1.13 做了大量的优化和调整，目前已经变得比较不错，如果你想用 Go modules，但还停留在 1.11/1.12 版本的话，强烈建议升级。

三个关键字

强推

首先这并不是乱说的，因为 Go modules 确实是被强推出来的，如下：

- 之前：大家都知道在 Go modules 之前还有一个叫 dep 的项目，它也是 Go 的一个官方的实验性项目，目的同样也是为了解决 Go 在依赖管理方面的短板。在 Russ Cox 还没有提出 Go modules 的时候，社区里面几乎所有的人都认为 dep 肯定就是未来 Go 官方的依赖管理解决方案了。

- 后来：谁都没想到半路杀出个程咬金，Russ Cox 义无反顾地推出了 Go modules，这瞬间导致一石激起千层浪，让社区炸了锅。大家一致认为 Go team 实在是太霸道、太独裁了，连个招呼都不打一声。我记得当时有很多人在网上跟 Russ Cox 口水战，各种依赖管理解决方案的专家都冒出来发表意见，讨论范围甚至一度超出了 Go 语言的圈子触及到了其他语言的领域。

理想化

从他强制要求使用语义化版本控制这一点来说就很理想化了，如下：

- Go modules 狠到如果你的 Tag 没有遵循语义化版本控制那么它就会忽略你的 Tag，然后根据你的 Commit 时间和哈希值再为你生成一个假定的符合语义化版本控制的版本号。
- Go modules 还默认认为，只要你的主版本号不变，那这个模块版本肯定就不包含 Breaking changes，因为语义化版本控制就是这么规定的啊。是不是很理想化。

类语言级：

这个关键词其实是我自己瞎编的，我只是单纯地个人认为 Go modules 在设计上就像个语言级特性一样，比如如果你的主版本号发生变更，那么你的代码里的 import path 也得跟着变，它认为主版本号不同的两个模块版本是完全不同的两个模块。此外，Go moduels 在设计上跟 go 整个命令都结合得相当紧密，无处不在，所以我才说它是一个有点儿像语言级的特性，虽然不是太严谨。

推 Go Modules 的人是谁

那么在上文中提到的 Russ Cox 何许人也呢，很多人应该都知道他，他是 Go 这个项目目前代码提交量最多的人，甚至是第二名的两倍还要多。

Russ Cox 还是 Go 现在的掌舵人（大家应该知道之前 Go 的掌舵人是 Rob Pike，但是听说由于他本人不喜欢特朗普执政所以离开了美国，然后他岁数也挺大的了，所以也正在逐渐交权，不过现在还是在参与 Go 的发展）。

Russ Cox 的个人能力相当强，看问题的角度也很独特，这也就是为什么他刚一提出 Go modules 的概念就能引起那么大范围的响应。虽然是被强推的，但事实也证明当下的 Go modules 表现得确实很优秀，所以这表明一定程度上的“独裁”还是可以接受的，至少可以保证一个项目能更加专一地朝着一个方向发展。

总之，无论如何 Go modules 现在都成了 Go 语言的一个密不可分的组件。

GOPATH

Go modules 出现的目的之一就是为了解决 GOPATH 的问题，也就相当于是抛弃 GOPATH 了。

Opt-in

Go modules 还处于 Opt-in 阶段，就是你想用就用，不用就不用，不强制你。但是未来很有可能 Go2 就强制使用了。

“module” != “package”

有一点需要纠正，就是“模块”和“包”，也就是 “module” 和 “package” 这两个术语并不是等价的，是 “集合” 跟 “元素” 的关系，“模块” 包含 “包”，“包” 属于 “模块”，一个 “模块” 是零个、一个或多个 “包” 的集合。

Go Modules 相关属性

Go Modules 简介

- 1 个开关环境变量：GO111MODULE
- 5 个辅助环境变量：GOPROXY、GONOPROXY、GOSUMDB、GONOSUMDB 和 GOPRIVATE
- 2 个辅助概念：Go module proxy 和 Go checksum database
- 2 个主要文件：go.mod 和 go.sum
- 1 个主要管理子命令：go mod
- 内置在几乎所有其他子命令中：go get、go install、go list、go test、go run、go build……
- Global Caching：不同项目的相同模块版本只会在你的电脑上缓存一份儿（让 npm 嫉妒死 😉）

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26

5

go.mod

```
module example.com/foobar

go 1.13

require (
    example.com/apple v0.1.2
    example.com/banana v1.2.3
    example.com/banana/v2 v2.3.4
```

```
    example.com/pineapple v0.0.0-20190924185754-1b0db40df49a
)

exclude example.com/banana v1.2.4
replace example.com/apple v0.1.2 => example.com/rda v0.1.0
replace example.com/banana => example.com/hugebanana
```

go.mod 是启用了 Go moduels 的项目所必须的最重要的文件，它描述了当前项目（也就是当前模块）的元信息，每一行都以一个动词开头，目前有以下 5 个动词:

- module：用于定义当前项目的模块路径。
- go：用于设置预期的 Go 版本。
- require：用于设置一个特定的模块版本。
- exclude：用于从使用中排除一个特定的模块版本。
- replace：用于将一个模块版本替换为另外一个模块版本。

这里的填写格式基本为包引用路径+版本号，另外比较特殊的是 `go $version` ，目前从 Go1.13 的代码里来看，还只是个标识作用，暂时未知未来是否有更大的作用。

go.sum

go.sum 是类似于比如 dep 的 Gopkg.lock 的一类文件，它详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

```
example.com/apple v0.1.2 h1:WXkYYl6Yr3qBf1K79EBnL4mak0OimBfB0XUf9Vl28OQ=
example.com/apple v0.1.2/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBksp03qcyfWMU=
 example.com/banana v1.2.3 h1:qHgHjyoNFV7jgucU8QZUuU4gcdhfs8QW1kw68OD2Lag=
example.com/banana v1.2.3/go.mod h1:HSdplMjZKSmBqAxg5vPj2TmRDmfkzw+cTzAElWljhcU=
example.com/banana/v2 v2.3.4 h1:zl/OfRA6nftbBK9qTohYBJ5xvw6C/oNKizR7cZGl3cI= exa
mple.com/banana/v2 v2.3.4/go.mod h1:eZbhyaAYD41SGSSsnmcpxVoRiQ/MPUTjUdIIOT9Um7Q=
...
```

我们可以看到一个模块路径可能有如下两种：

```
example.com/apple v0.1.2 h1:WXkYYl6Yr3qBf1K79EBnL4mak0OimBfB0XUf9Vl28OQ=
example.com/apple v0.1.2/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBksp03qcyfWMU=
```

前者为 Go modules 打包整个模块包文件 zip 后再进行 hash 值，而后者为针对 go.mod 的 hash 值。他们两者，要不就是同时存在，要不就是只存在 go.mod hash。

那什么情况下会不存在 zip hash 呢，就是当 Go 认为肯定用不到某个模块版本的时候就会省略它的 zip hash，就会出现不存在 zip hash，只存在 go.mod hash 的情况。

GO111MODULE

这个环境变量主要是 Go modules 的开关，主要有以下参数：

- auto：只在项目包含了 go.mod 文件时启用 Go modules，在 Go 1.13 中仍然是默认值，详见
  ：golang.org/issue/31857。
- on：无脑启用 Go modules，推荐设置，未来版本中的默认值，让 GOPATH 从此成为历史。
- off：禁用 Go modules。

GOPROXY

这个环境变量主要是用于设置 Go 模块代理，主要如下：

- 它的值是一个以英文逗号 “,” 分割的 Go module proxy 列表（稍后讲解）
  - 作用：用于使 Go 在后续拉取模块版本时能够脱离传统的 VCS 方式从镜像站点快速拉取。它拥有一个默认：`https://proxy.golang.org,direct`，但很可惜 `proxy.golang.org` 在中国无法访问，故而建议使用 `goproxy.cn` 作为替代，可以执行语句：`go env -w GOPROXY=https://goproxy.cn,direct`。
  - 设置为 “off” ：禁止 Go 在后续操作中使用任 何 Go module proxy。

刚刚在上面，我们可以发现值列表中有 “direct” ，它又有什么作用呢。其实值列表中的 “direct” 为特殊指示符，用于指示 Go 回源到模块版本的源地址去抓取(比如 GitHub 等)，当值列表中上一个 Go module proxy 返回 404 或 410 错误时，Go 自动尝试列表中的下一个，遇见 “direct” 时回源，遇见 EOF 时终止并抛出类似 “invalid version: unknown revision...” 的错误。

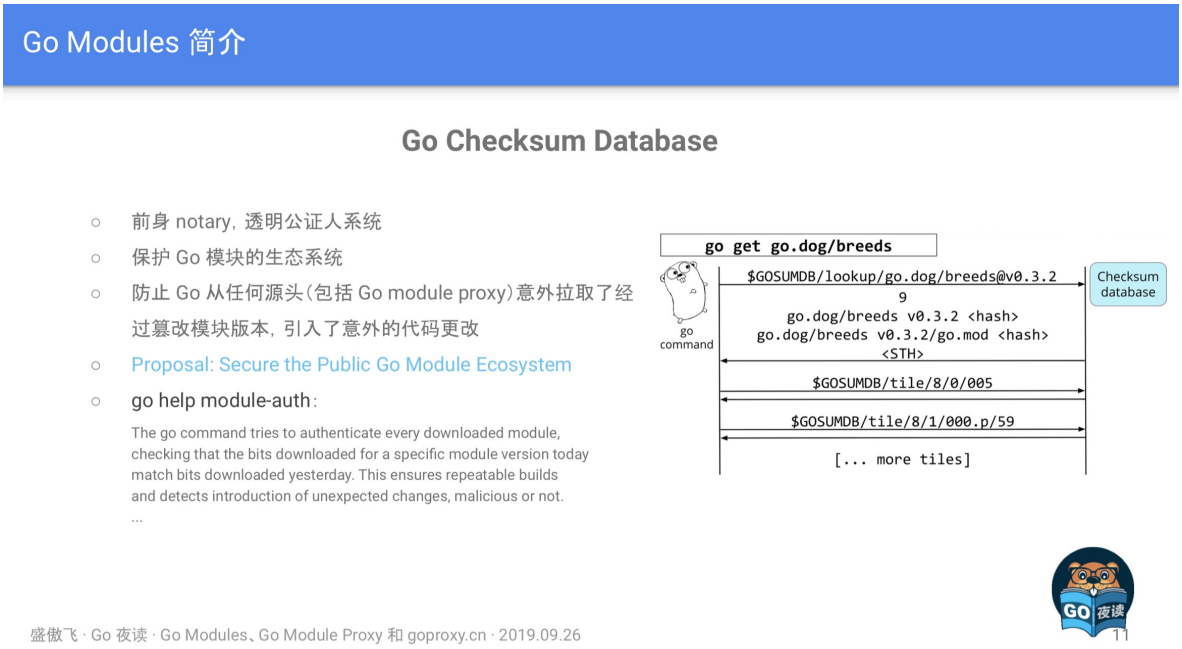
GOSUMDB

它的值是一个 Go checksum database，用于使 Go 在拉取模块版本时(无论是从源站拉取还是通过 Go module proxy 拉取)保证拉取到的模块版本数据未经篡改，也可以是“off”即禁止 Go 在后续操作中校验模块版本

- 格式 1：`<SUMDB_NAME>+<PUBLIC_KEY>`。
- 格式 2：`<SUMDB_NAME>+<PUBLIC_KEY> <SUMDB_URL>`。
- 拥有默认值：`sum.golang.org`(之所以没有按照上面的格式是因为 Go 对默认值做了特殊处理)。
- 可被 Go module proxy 代理 (详见：Proxying a Checksum Database)。
- `sum.golang.org` 在中国无法访问，故而更加建议将 GOPROXY 设置为 `goproxy.cn`，因为 `goproxy.cn` 支持代理 `sum.golang.org`。

Go Checksum Database

Go checksum database 主要用于保护 Go 不会从任何源头拉到被篡改过的非法 Go 模块版本，其作用（左）和工作机制（右）如下图：



如果有兴趣的小伙伴可以看看 Proposal: Secure the Public Go Module Ecosystem，有详细介绍其算法机制，如果想简单一点，查看 `go help module-auth` 也是一个不错的选择。

GONOPROXY/GONOSUMDB/GOPRIVATE

这三个环境变量都是用在当前项目依赖了私有模块，也就是依赖了由 GOPROXY 指定的 Go module proxy 或由 GOSUMDB 指定 Go checksum database 无法访问到的模块时的场景

- 它们三个的值都是一个以英文逗号 “,” 分割的模块路径前缀，匹配规则同 path.Match。

- 其中 GOPRIVATE 较为特殊，它的值将作为 GONOPROXY 和 GONOSUMDB 的默认值，所以建议的最佳姿势是只是用 GOPRIVATE。

在使用上来讲，比如 `GOPRIVATE=*.corp.example.com` 表示所有模块路径以 `corp.example.com` 的下一级域名 (如 `team1.corp.example.com`) 为前缀的模块版本都将不经过 Go module proxy 和 Go checksum database，需要注意的是不包括 `corp.example.com` 本身。

Global Caching

这个主要是针对 Go modules 的全局缓存数据说明，如下：

- 同一个模块版本的数据只缓存一份，所有其他模块共享使用。
- 目前所有模块版本数据均缓存在 `$GOPATH/pkg/mod` 和 `$GOPATH/pkg/sum` 下，未来或将移至 `$GOCACHE/mod` 和 `$GOCACHE/sum` 下(可能会在当 `$GOPATH` 被淘汰后)。
- 可以使用 `go clean -modcache` 清理所有已缓存的模块版本数据。

另外在 Go1.11 之后 GOCACHE 已经不允许设置为 off 了，我想着这也是为了模块数据缓存移动位置做准备，因此大家应该尽快做好适配。

快速迁移项目至 Go Modules

- 第一步: 升级到 Go 1.13。
- 第二步: 让 GOPATH 从你的脑海中完全消失，早一步踏入未来。
  - 修改 GOBIN 路径（可选）：`go env -w GOBIN=$HOME/bin`。
  - 打开 Go modules：`go env -w GO111MODULE=on`。
  - 设置 GOPROXY：`go env -w GOPROXY=https://goproxy.cn,direct` # 在中国是必须的，因为它的默认值被墙了。
- 第三步(可选): 按照你喜欢的目录结构重新组织你的所有项目。
- 第四步: 在你项目的根目录下执行 `go mod init <OPTIONAL_MODULE_PATH>` 以生成 go.mod 文件。
- 第五步: 想办法说服你身边所有的人都去走一下前四步。

迁移后 go get 行为的改变

- 用 `go help module-get` 和 `go help gopath-get` 分别去了解 Go modules 启用和未启用两种状态下的 go get 的行为
- 用 `go get` 拉取新的依赖
  - 拉取最新的版本(优先择取 tag)：`go get golang.org/x/text@latest`
  - 拉取 `master` 分支的最新 commit：`go get golang.org/x/text@master`
  - 拉取 tag 为 v0.3.2 的 commit：`go get golang.org/x/text@v0.3.2`
  - 拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2：`go get golang.org/x/text@342b2e`
  - 用 `go get -u` 更新现有的依赖
  - 用 `go mod download` 下载 go.mod 文件中指明的所有依赖
  - 用 `go mod tidy` 整理现有的依赖
  - 用 `go mod graph` 查看现有的依赖结构
  - 用 `go mod init` 生成 go.mod 文件 (Go 1.13 中唯一一个可以生成 go.mod 文件的子命令)
- 用 `go mod edit` 编辑 go.mod 文件
- 用 `go mod vendor` 导出现有的所有依赖 (事实上 Go modules 正在淡化 Vendor 的概念)
- 用 `go mod verify` 校验一个模块是否被篡改过

这里我们注意到有两点比较特别，分别是：

- 第一点：为什么“拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2”呢。这是因为虽然我们设置了拉取 @342b2e commit，但是因为 Go modules 会与 tag 进行对比，若发现对应的 commit 与 tag 有关联，则进行转换。
- 第二点：为什么不建议使用 `go mod vendor`，因为 Go modules 正在淡化 Vendor 的概念，很有可能 Go2 就去掉了。

使用 Go Modules 时常遇见的坑

坑 1: 判断项目是否启用了 Go Modules

使用 Go Modules 时常遇见的坑(坑 1:判断项目是否启用了 Go Modules)

- 切记，在 Go 1.13 中，一个项目只要包含了 go.mod 文件，且它所处环境中的 GO111MODULE 不为 off，那么 Go 就会为这个项目启用 Go modules
- 为了一劳永逸地解决这个判断烦恼，建议大家将 GO111MODULE 设置为 on
  - go env -w GO111MODULE=on
- 另外，若当前项目没有包含 go.mod 文件，且 GO111MODULE 为 on，那么每一次构建代码时 Go 都会从头推算并拉取所需要的模块版本，**但是并不会自动生成 go.mod 文件**(以前会自动生成，Go 1.13 做了调整)

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 18

坑 2: 管理 Go 的环境变量

使用 Go Modules 时常遇见的坑(坑 2:管理 Go 的环境变量)

- 在 Go 1.13 中管理环境变量会变得稍显混乱，因为 Go 1.13 建议将所有跟 Go 相关的环境变量都交由新出的 go env -w 来管理，比如执行 go env -w GO111MODULE=on 就会在 $HOME/.config/go/env 文件中追加一行 “GO111MODULE=on”
- **但 go env -w 不会覆盖你的系统环境变量**
- 所以建议大家在升级到 Go 1.13 后做一次环境变量的管理方式的转变，比如删除你系统中所有跟 Go 相关的环境变量，并使用 go env -w 重写会
- os.UserConfigDir:

> UserConfigDir returns the default root directory to use for user-specific configuration data. Users should create their own application-specific subdirectory within this one and use that.
>
> On Unix systems, it returns $XDG_CONFIG_HOME as specified by https://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html if non-empty, else $HOME/.config. On Darwin, it returns $HOME/Library/Application Support. On Windows, it returns %AppData%. On Plan 9, it returns $home/lib.
>
> If the location cannot be determined (for example, $HOME is not defined), then it will return an error.

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 19

这里主要是提到 Go1.13 新增了 `go env -w` 用于写入环境变量，而写入的地方是 `os.UserConfigDir` 所返回的路径，需要注意的是 `go env -w` 不会覆写。

坑 3: 从 dep、glide 等迁移至 Go Modules

使用 Go Modules 时常遇见的坑(坑 3:从 dep、glide 等迁移至 Go Modules)

- 在执行 go mod init <OPTIONAL_MODULE_PATH> 时，如果当前目录包含了 Gopkg.lock、glide.lock 等老的依赖管理文件时，Go 会用它们所指定的依赖版本信息来推算并生成 go.mod 文件，此时所有的推算行为都是直接回源的，并不会经过 Go module proxy，这是个 Bug，详见:golang.org/issue/33767
- 不出意外的话这个 Bug 将在 Go 1.14 中被修复，我们正在努力
- 临时的解决办法时先按照 go help go.mod 里指定的规则手动创建一个 go.mod 文件(可以只包含第一行的模块路径)，然后再执行 go mod tidy 来推算并补充 go.mod 文件，这样儿就可以走 Go module proxy 了

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 20

这里主要是指从旧有的依赖包管理工具（dep/glide 等）进行迁移时，因为 BUG 的原因会导致不经过 GOPROXY 的代理，解决方法有如下两个：

- 手动创建一个 go.mod 文件，再执行 go mod tidy 进行补充。
- 上代理，相当于不使用 GOPROXY 了。

坑 4:拉取私有模块

使用 Go Modules 时常遇见的坑(坑 4:拉取私有模块)

- 常见的公共 Go module proxy，比如 proxy.golang.org 和 goproxy.cn 都是无权访问任何人的私有模块的
- 即使不使用任何 Go module proxy，也就是将 GOPROXY 设置为了 direct，同样默认情况下 Go 也是无法抓取你的私有模块的，详见:golang.org/doc/faq#git_https
- 解决办法是想办法修改你的 VCS 拉取行为，比如使用 Git 时就在 $HOME/.gitconfig 文件中追加:

```
[url "ssh://git@github.com/"]
        insteadOf = https://github.com/
```

- 并为 GOPROXY 设置一个 fallback 选项(当列表中的前一个 404 或 410 时就自动尝试下一个，遇见 direct 时回源也就是直接去模块所在地拉取)
  - go env -w GOPROXY=https://goproxy.cn,direct
- 还可以通过设置 GONOPROXY 或 GOPRIVATE 来指使 Go 在拉取哪些模块时忽略 Go module proxy

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 21

这里主要想涉及两块知识点，如下：

- GOPROXY 是无权访问到任何人的私有模块的，所以你放心，安全性没问题。
- GOPROXY 除了设置模块代理的地址以外，还需要增加 “direct” 特殊标识才可以成功拉取私有库。

坑 5:更新现有的模块

使用 Go Modules 时常遇见的坑(坑 5:更新现有的模块)

- go get -u
  - 只会更新主要模块，忽略了单元测试
- go get -u ./...
  - 递归更新所有子目录的所有模块，忽略了单元测试
- go get -u -t
  - 同样是只会更新主要模块，但考虑了单元测试
- go get -u -t ./...
  - 同样是递归更新所有子目录的所有模块，但考虑了单元测试
- go get -u all
  - 更新所有模块，推荐使用

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 22

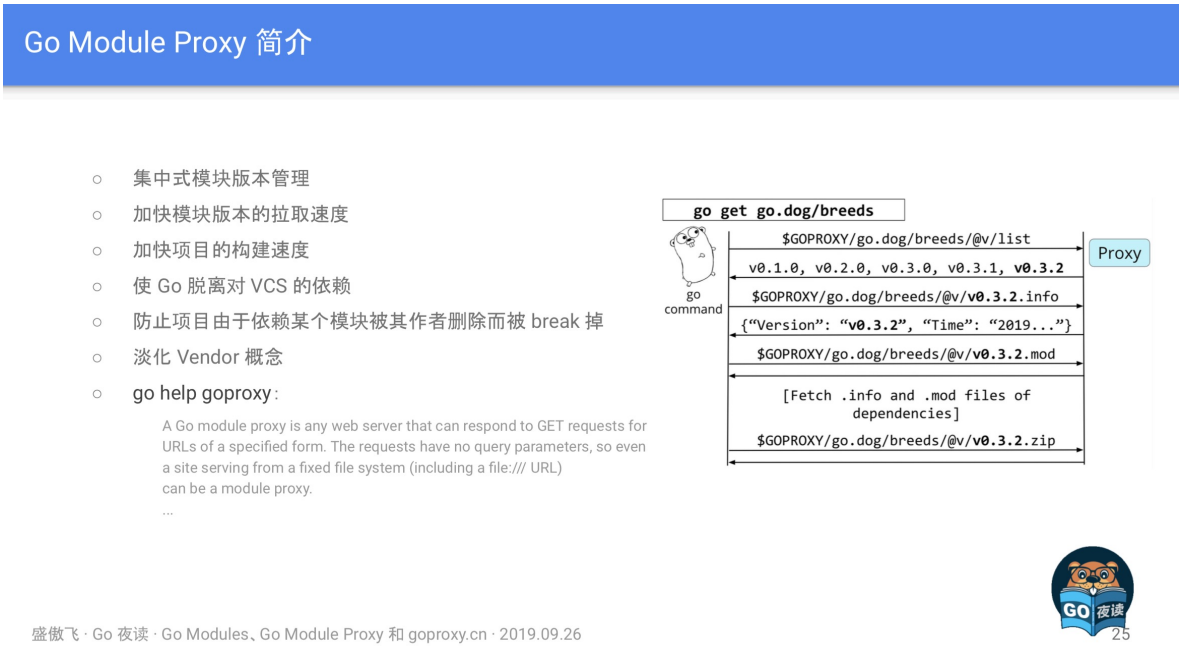
坑 6:主版本号

使用 Go Modules 时常遇见的坑(坑 6:主版本号)

- 除了 v0 和 v1 外主版本号必须显示地出现在模块路径的尾部
- Go 会将 example.com/foo/bar 和 example.com/foo/bar/v2 视为完全不同的两个模块来对待
- 一个项目中可能同时存在多个拥有不同的主版本号的相同模块，但互不影响
- go get -u 不会更新主版本号，如需更新，需要手动修改代码中对应的导入路径

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 23

Go Module Proxy 简介



在这里再次强调了 Go Module Proxy 的作用（图左），以及其对应的协议交互流程（图右），有兴趣的小伙伴可以认真看一下。

Goproxy 中国(goproxy.cn)

在这块主要介绍了 Goproxy 的实践操作以及 goproxy.cn 的一些 Q&A 和 近况，如下：

Q&A

Q：如果中国 Go 语言社区没有咱们自己家的 Go Module Proxy 会怎么样？

A：在 Go 1.13 中 GOPROXY 和 GOSUMDB 这两个环境变量都有了在中国无法 访问的默认值，尽管我在 golang.org/issue/31755 里努力尝 试过，但最终仍然无法为咱们中国的 Go 语言开发者谋得一个完美的解决方案。所以从今以后咱 们中国的所有 Go 语言开发者，只要是 使用了 Go modules 的，那么都必须先修改 GOPROXY 和 GOSUMDB 才能正常使用 Go 做开发，否则可能连一个最简单的程序都跑不起 来(只要它有依 赖第三方模 块)。

Q： 我创建 Goproxy 中国(goproxy.cn)的主要原因？

A：其实更早的时候，也就是今年年初我也曾 试图在 golang.org/issue/31020 中请求 Go team 能想办法避免那时的 GOPROXY 即将拥有的默认值可以在中国正常访问，但 Go team 似乎也无能为力，为此我才坚定了创建 goproxy.cn 的信念。既然别人没法儿帮忙，那咱们就得自己动手，不为别的，就为了让大家以后能够更愉快地使用 Go 语言配合 Go modules 做开发。

最初我先是和七牛云的 许叔(七牛云的 创始人兼 CEO 许式伟)提出了我打算 创建 goproxy.cn 的想法，本是抱着 试试看的目的，但没想 到 许叔几乎是没有超过一分钟的考虑便认可了我的想

法并表示愿意一起推 动。那一阵子刚好赶上我在写毕业论文，所以项目开发完后就 一直没和七牛云做交接，一直跑在我的个人服 务器上。直到有一次 goproxy.cn 被攻击了，一下午的功夫烧了我一百多美元，然后我才 意识到这种项目真不能个人来做。个人来做不靠 谱，万一依赖这个项目的人多了，项目再出什么事儿，那就会给大家□成不必要的损 失。所以我赶紧和七牛云做了交接，把 goproxy.cn 完全交给了七牛云，甚至连域名都过户了去。

近况

区域	最快/最慢	平均
全网	北京(鹏博士) 0.2ms/柬埔寨金边(cambo.host) 360.8ms	25.6ms
移动	安徽合肥(hccn2) 3.3ms/广东广州(移动) 35.8ms	15.1ms
联通	天津(联通) 6.7ms/重庆(联通) 27.7ms	17.2ms
电信	四川成都(天翼云) 1.5ms/新疆乌鲁木齐(天翼云) 66.3ms	27.4ms
华南	广东广州(腾讯云) 9.0ms / 海南海口(天翼云) 49.9ms	25.3ms
华北	北京(鹏博士) 0.2ms / 内蒙古呼和浩特(蓝视云) 45.4ms	24.0ms
华东	上海(阿里云) 1.0ms / 山东青岛(天翼云) 48.2ms	20.1ms
华中	湖南长沙(天翼云) 5.4ms / 河南郑州(景安网络) 58.9ms	20.3ms
东北	辽宁鞍山(电信) 19.6ms / 吉林长春(天翼云) 57.7ms	33.3ms
西北	陕西西安(天翼云一区) 13.8ms / 新疆乌鲁木齐(天翼云) 66.3ms	41.7ms
西南	四川成都(天翼云) 1.5ms / 贵州贵阳(华为云) 58.5ms	26.6ms
港澳台	台湾台北(远传电信) 0.6ms / 香港(hkc) 155.6ms	27.9ms
亚洲	新加坡(腾讯云) 0.4ms/柬埔寨金边(cambo.host) 360.8ms	15.3ms
欧洲	意大利米兰 0.4ms/法国巴黎(VULTR) 146.5ms	22.1ms
非洲	南非约翰内斯堡 0.2ms/尼日利亚拉各斯 265.2ms	48.9ms
北美洲	美国西雅图(VULTR) 0.3ms/美国波特兰(AMAZON-LIGHTSAIL) 97.2ms	34.9ms
南美洲	巴西圣保罗(GCP) 3.7ms/阿根廷布宜诺斯艾利斯(latincloud.com) 145.0ms	50.9ms
大洋洲	澳大利亚悉尼(xtom.com) 0.7ms/澳大利亚阿德莱德 52.6ms	15.1ms

- Goproxy 中国 (goproxy.cn) 是目前中国最可靠的 Go module proxy (真不是在自卖自夸)。
- 为中国 Go 语言开发者量身打□，支持代理 GOSUMDB 的默认值，经过全球 CDN 加速，高可用，可 应用进公司复杂的开发环境中，亦可用作上游代理。
- 由中国倍受信赖的云服务提供商七牛云无偿提供基础设施支持的开源的非营利性项目。
- 目标是为中国乃至全世界的 Go 语言开发者提供一个免 费的、可靠的、持 续在线的且经过 CDN 加□的 Go module proxy。
- 域名已由七牛云进行了备案 (沪ICP备11037377号-56)。

情况

Goproxy 中国（goproxy.cn）

Goproxy 中国（goproxy.cn）截止到 2019-09-25 的状态

- 存储空间总大小已超过 **568.88 GB**
- 已缓存 **278,737** 个模块版本
- **0-10 MB** 的模块版本的 ZIP 文件共有 **262,615** 个
- **10-50 MB** 的模块版本的 ZIP 文件共有 **15,580** 个
- **50-100 MB** 的模块版本的 ZIP 文件共有 **451** 个
- **大于 100 MB** 的模块版本的 ZIP 文件共有 **91** 个（哥几个……你们是往代码里面加铁块儿了吗？🤨）

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 36

此处呈现的是存储大小，主要是针对模块包代码，而一般来讲代码并不会有多大，0-10MB，10-50MB 占最大头，也是能够理解，但是大于 100MB 的模块包代码就比较夸张了。

Goproxy 中国（goproxy.cn）

Goproxy 中国（goproxy.cn）在 21 天内的数据

2019-09-04 至 2019-09-25（15 个工作日）

- 处理了 **12,369,839** 条请求
- 出站了 **444.18 GB** 的流量
- 峰值带宽 **157.37 Mbit/s**
- 平均峰值带宽 **95.20 Mbit/s**

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 37

此时主要是展示了一下近期 goproxy.cn 的网络数据情况，我相信未来是会越来越高的，值得期待。

Q&A

Q：如何解决 Go 1.13 在从 GitLab 拉取模块版本时遇到的，Go 错误地按照非期望值的路径寻找目标模块版本结果致使最终目标模块拉取失败的问题？

A：GitLab 中配合 goget 而设置的 `<meta>` 存在些许问题，导致 Go 1.13 错误地识别了模块的具体路径，这是个 Bug，据说在 GitLab 的新版本中已经被修复了，详细内容可以看 https://github.com/golang/go/issues/34094 这个 Issue。然后目前的解决办法的话除了升级 GitLab 的版本外，还可以参考 https://github.com/developer-learning/night-reading-go/issues/468#issuecomment-535850154 这条回复。

Q：使用 Go modules 时可以同时依赖同一个模块的不同的两个或者多个小版本（修订版本号不同）吗？

A：不可以的，Go modules 只可以同时依赖一个模块的不同的两个或者多个大版本（主版本号不同）。比如可以同时依赖 foobar@v1.2.3"">example.com/foobar@v1.2.3 和 v2@v2.3.4"">example.com/foobar/v2@v2.3.4，因为他们的模块路径（module path）不同，Go modules 规定主版本号不是 v0 或者 v1 时，那么主版本号必须显式地出现在模块路径的尾部。但是，同时依赖两个或者多个小版本是不支持的。比如如果模块 A 同时直接依赖了模块 B 和模块 C，且模块 A 直接依赖的是模块 C 的 v1.0.0 版本，然后模块 B 直接依赖的是模块 C 的 v1.0.1 版本，那么最终 Go modules 会为模块 A 选用模块 C 的 v1.0.1 版本而不是模块 A 的 go.mod 文件中指明的 v1.0.0 版本。

这是因为 Go modules 认为只要主版本号不变，那么剩下的都可以直接升级采用最新的。但是如果采用了最新的结果导致项目 Break 掉了，那么 Go modules 就会 Fallback 到上一个老的版本，比如在前面的例子中就会 Fallback 到 v1.0.0 版本。

Q：在 go.sum 文件中的一个模块版本的 Hash 校验数据什么情况下会成对出现，什么情况下只会存在一行？

A：通常情况下，在 go.sum 文件中的一个模块版本的 Hash 校验数据会有两行，前一行是该模块的 ZIP 文件的 Hash 校验数据，后一行是该模块的 go.mod 文件的 Hash 校验数据。但是也有些情况下只会出现一行该模块的 go.mod 文件的 Hash 校验数据，而不包含该模块的 ZIP 文件本身的 Hash 校验数据，这个情况发生在 Go modules 判定为你当前这个项目完全用不到该模块，根本也不会下载该模块的 ZIP 文件，所以就没必要对其作出 Hash 校验保证，只需要对该模块的 go.mod 文件作出 Hash 校验保证即可，因为 go.mod 文件是用得着的，在深入挖取项目依赖的时候要用。

Q：能不能更详细地讲解一下 go.mod 文件中的 replace 动词的行为以及用法？

A：这个 replace 动词的作用是把一个“模块版本”替换为另外一个“模块版本”，这是“模块版本”和“模块版本（module path）”之间的替换，“=>”标识符前面的内容是待替换的“模块版本”的“模块路径”，后面的内容是要替换的目标“模块版本”的所在地，即路径，这个路径可以是一个本地磁盘的相对路径，也可以是一个本地磁盘的绝对路径，还可以是一个网络路径，但是这个目标路径并不会在今后你的项目代码中作为你“导入路径（import path）”出现，代码里的“导入路径”还是得以你替换成的这个目标“模块版本”的“模块路径”作为前缀。

另外需要注意，Go modules 是不支持在 “导入路径” 里写相对路径的。举个例子，如果项目 A 依赖了模块 B，比如模块 B 的“模块路径”是 example.com/b，然后它在的磁盘路径是 ~~/b，在项目 A 里的 go.mod 文件中你有一行 replace example.com/b=>~~/b，然后在项目 A 里的代

码中的“导入路基”就是 import”example.com/b”，而不是 import”~/b”，剩下的工作是 Go modules 帮你自动完成了的。

然后就是我在分享中也提到了， exclude 和 replace 这两个动词只作用于当前主模块，也就是当前项目，它所依赖的那些其他模块版本中如果出现了你待替换的那个模块版本的话，Go modules 还是会为你依赖的那个模块版本去拉取你的这个待替换的模块版本。

举个例子，比如项目 A 直接依赖了模块 B 和模块 C，然后模块 B 也直接依赖了模块 C，那么你在项目 A 中的 go.mod 文件里的 replace c=>~/some/path/c 是只会影响项目 A 里写的代码中，而模块 B 所用到的还是你 replace 之前的那个 c，并不是你替换成的 ~/some/path/c 这个。

总结

在 Go1.13 发布后，接触 Go modules 和 Go module proxy 的人越来越多，经常在各种群看到各种小伙伴在咨询，包括我自己也贡献了好几枚 “坑”，因此我觉得傲飞的这一次 《Go Modules、Go Module Proxy 和 goproxy.cn》的技术分享，非常的有实践意义。如果后续大家还有什么建议或问题，欢迎随时来讨论。

最后，感谢 goproxy.cn 背后的人们（@七牛云 和 @盛傲飞）对中国 Go 语言社区的无私贡献和奉献。

进一步阅读

- night-reading-go/issues/468
- B站：【Go 夜读】第 61 期 Go Modules、Go Module Proxy 和 goproxy.cn
- youtube：【Go 夜读】第 61 期 Go Modules、Go Module Proxy 和 goproxy.cn

Go Modules 终极入门

大家好，我是一只普通的煎鱼，周四晚上很有幸邀请到 goproxy.cn 的作者 @盛傲飞（@aofei） 到 Go 夜读给我们进行第 61 期 《Go Modules、Go Module Proxy 和 goproxy.cn》的技术分享。

本次 @盛傲飞 的夜读分享，是对 Go Modules 的一次很好的解读，比较贴近工程实践，我必然希望把这块的知识更多的分享给大家，因此有了今天本篇文章，同时大家也可以多关注 Go 夜读，每周会通过 zoom 在线直播的方式分享 Go 相关的技术话题，希望对大家有所帮助。

前言

Go 1.11 推出的模块（Modules）为 Go 语言开发者打开了一扇新的大门，理想化的依赖管理解决方案使得 Go 语言朝着计算机编程史上的第一个依赖乌托邦（Deptopia）迈进。随着模块一起推出的还有模块代理协议（Module proxy protocol），通过这个协议我们可以实现 Go 模块代理（Go module proxy），也就是依赖镜像。

Go 1.13 的发布为模块带来了大量的改进，所以模块的扶正就是这次 Go 1.13 发布中开发者能直接感觉到的最大变化。而问题在于，Go 1.13 中的 GOPROXY 环境变量拥有了一个在中国大陆无法访问到的默认值 `proxy.golang.org` ，经过大家在 golang/go#31755 中激烈的讨论（有些人甚至将话提上升到了“自由世界”的层次），最终 Go 核心团队仍然无法为中国开发者提供一个可在中国大陆访问的官方模块代理。

为了今后中国的 Go 语言开发者能更好地进行开发，七牛云推出了非营利性项目 `goproxy.cn` ，其目标是为中国和世界上其他地方的 Gopher 们提供一个免费的、可靠的、持续在线的且经过 CDN 加速的模块代理。可以预见未来是属于模块化的，所以 Go 语言开发者能越早切入模块就能越早进入未来。

如果说 Go 1.11 和 Go 1.12 时由于模块的不完善你不愿意切入，那么 Go 1.13 你则可以大胆地开始放心使用。本次分享将讨论如何使用模块和模块代理，以及在它们的使用中会常遇见的坑，还会讲解如何快速搭建自己的私有模块代理，并简单地介绍一下七牛云推出的 `goproxy.cn` 以及它的出现对于中国 Go 语言开发者来说重要在何处。

目录

    - 坑 3:从 dep、glide 等迁移至 Go Modules
    - 坑 4:拉取私有模块
    - 坑 5:更新现有的模块
    - 坑 6:主版本号
- Go Module Proxy 简介
- Goproxy 中国(goproxy.cn)

Go Modules 简介

Go modules (前身 vgo) 是 Go team (Russ Cox) **强推**的一个**理想化**的**类语言级**依赖管理解决方案，它是和 Go1.11 一同发布的，在 Go1.13 做了大量的优化和调整，目前已经变得比较不错，如果你想用 Go modules，但还停留在 1.11/1.12 版本的话，强烈建议升级。

三个关键字

强推

首先这并不是乱说的，因为 Go modules 确实是被强推出来的，如下：

- 之前：大家都知道在 Go modules 之前还有一个叫 dep 的项目，它也是 Go 的一个官方的实验性项目，目的同样也是为了解决 Go 在依赖管理方面的短板。在 Russ Cox 还没有提出 Go modules 的时候，社区里面几乎所有的人都认为 dep 肯定就是未来 Go 官方的依赖管理解决方案了。

- 后来：谁都没想到半路杀出个程咬金，Russ Cox 义无反顾地推出了 Go modules，这瞬间导致一石激起千层浪，让社区炸了锅。大家一致认为 Go team 实在是太霸道、太独裁了，连个招呼都不打一声。我记得当时有很多人在网上跟 Russ Cox 口水战，各种依赖管理解决方案的专家都冒出来发表意见，讨论范围甚至一度超出了 Go 语言的圈子触及到了其他语言的领域。

理想化

从他强制要求使用语义化版本控制这一点来说就很理想化了，如下：

- Go modules 狠到如果你的 Tag 没有遵循语义化版本控制那么它就会忽略你的 Tag，然后根据你的 Commit 时间和哈希值再为你生成一个假定的符合语义化版本控制的版本号。
- Go modules 还默认认为，只要你的主版本号不变，那这个模块版本肯定就不包含 Breaking changes，因为语义化版本控制就是这么规定的啊。是不是很理想化。

类语言级：

这个关键词其实是我自己瞎编的，我只是单纯地个人认为 Go modules 在设计上就像个语言级特性一样，比如如果你的主版本号发生变更，那么你的代码里的 import path 也得跟着变，它认为主版本号不同的两个模块版本是完全不同的两个模块。此外，Go moduels 在设计上跟 go 整个命令都结合得相当紧密，无处不在，所以我才说它是一个有点儿像语言级的特性，虽然不是太严谨。

推 Go Modules 的人是谁

那么在上文中提到的 Russ Cox 何许人也呢，很多人应该都知道他，他是 Go 这个项目目前代码提交量最多的人，甚至是第二名的两倍还要多。

Russ Cox 还是 Go 现在的掌舵人（大家应该知道之前 Go 的掌舵人是 Rob Pike，但是听说由于他本人不喜欢特朗普执政所以离开了美国，然后他岁数也挺大的了，所以也正在逐渐交权，不过现在还是在参与 Go 的发展）。

Russ Cox 的个人能力相当强，看问题的角度也很独特，这也就是为什么他刚一提出 Go modules 的概念就能引起那么大范围的响应。虽然是被强推的，但事实也证明当下的 Go modules 表现得确实很优秀，所以这表明一定程度上的“独裁”还是可以接受的，至少可以保证一个项目能更加专一地朝着一个方向发展。

总之，无论如何 Go modules 现在都成了 Go 语言的一个密不可分的组件。

GOPATH

Go modules 出现的目的之一就是为了解决 GOPATH 的问题，也就相当于是抛弃 GOPATH 了。

Opt-in

Go modules 还处于 Opt-in 阶段，就是你想用就用，不用就不用，不强制你。但是未来很有可能 Go2 就强制使用了。

“module” != “package”

有一点需要纠正，就是“模块”和“包”，也就是 “module” 和 “package” 这两个术语并不是等价的，是 “集合” 跟 “元素” 的关系，“模块” 包含 “包”，“包” 属于 “模块”，一个 “模块” 是零个、一个或多个 “包” 的集合。

Go Modules 相关属性

Go Modules 简介

- 1 个开关环境变量：GO111MODULE
- 5 个辅助环境变量：GOPROXY、GONOPROXY、GOSUMDB、GONOSUMDB 和 GOPRIVATE
- 2 个辅助概念：Go module proxy 和 Go checksum database
- 2 个主要文件：go.mod 和 go.sum
- 1 个主要管理子命令：go mod
- 内置在几乎所有其他子命令中：go get、go install、go list、go test、go run、go build……
- Global Caching：不同项目的相同模块版本只会在你的电脑上缓存一份儿（让 npm 嫉妒死 😏）

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 5

go.mod

```
module example.com/foobar

go 1.13

require (
    example.com/apple v0.1.2
    example.com/banana v1.2.3
    example.com/banana/v2 v2.3.4
```

```
    example.com/pineapple v0.0.0-20190924185754-1b0db40df49a
)

exclude example.com/banana v1.2.4
replace example.com/apple v0.1.2 => example.com/rda v0.1.0
replace example.com/banana => example.com/hugebanana
```

go.mod 是启用了 Go moduels 的项目所必须的最重要的文件，它描述了当前项目（也就是当前模块）的元信息，每一行都以一个动词开头，目前有以下 5 个动词:

- module：用于定义当前项目的模块路径。
- go：用于设置预期的 Go 版本。
- require：用于设置一个特定的模块版本。
- exclude：用于从使用中排除一个特定的模块版本。
- replace：用于将一个模块版本替换为另外一个模块版本。

这里的填写格式基本为包引用路径+版本号，另外比较特殊的是 `go $version` ，目前从 Go1.13 的代码里来看，还只是个标识作用，暂时未知未来是否有更大的作用。

go.sum

go.sum 是类似于比如 dep 的 Gopkg.lock 的一类文件，它详细罗列了当前项目直接或间接依赖的所有模块版本，并写明了那些模块版本的 SHA-256 哈希值以备 Go 在今后的操作中保证项目所依赖的那些模块版本不会被篡改。

```
example.com/apple v0.1.2 h1:WXkYYl6Yr3qBf1K79EBnL4mak0OimBfB0XUf9Vl28OQ=
example.com/apple v0.1.2/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBksp03qcyfWMU=
 example.com/banana v1.2.3 h1:qHgHjyoNFV7jgucU8QZUuU4gcdhfs8QW1kw68OD2Lag=
example.com/banana v1.2.3/go.mod h1:HSdplMjZKSmBqAxg5vPj2TmRDmfkzw+cTzAElWljhcU=
example.com/banana/v2 v2.3.4 h1:zl/OfRA6nftbBK9qTohYBJ5xvw6C/oNKizR7cZGl3cI= exa
mple.com/banana/v2 v2.3.4/go.mod h1:eZbhyaAYD41SGSSsnmcpxVoRiQ/MPUTjUdIIOT9Um7Q=
...
```

我们可以看到一个模块路径可能有如下两种：

```
example.com/apple v0.1.2 h1:WXkYYl6Yr3qBf1K79EBnL4mak0OimBfB0XUf9Vl28OQ=
example.com/apple v0.1.2/go.mod h1:xHWCNGjB5oqiDr8zfno3MHue2Ht5sIBksp03qcyfWMU=
```

前者为 Go modules 打包整个模块包文件 zip 后再进行 hash 值，而后者为针对 go.mod 的 hash 值。他们两者，要不就是同时存在，要不就是只存在 go.mod hash。

那什么情况下会不存在 zip hash 呢，就是当 Go 认为肯定用不到某个模块版本的时候就会省略它的 zip hash，就会出现不存在 zip hash，只存在 go.mod hash 的情况。

GO111MODULE

这个环境变量主要是 Go modules 的开关，主要有以下参数：

- auto：只在项目包含了 go.mod 文件时启用 Go modules，在 Go 1.13 中仍然是默认值，详见
  ：golang.org/issue/31857。
- on：无脑启用 Go modules，推荐设置，未来版本中的默认值，让 GOPATH 从此成为历史。
- off：禁用 Go modules。

GOPROXY

这个环境变量主要是用于设置 Go 模块代理，主要如下：

- 它的值是一个以英文逗号 “,” 分割的 Go module proxy 列表（稍后讲解）
  - 作用：用于使 Go 在后续拉取模块版本时能够脱离传统的 VCS 方式从镜像站点快速拉取。它拥有一个默认：`https://proxy.golang.org,direct`，但很可惜 `proxy.golang.org` 在中国无法访问，故而建议使用 `goproxy.cn` 作为替代，可以执行语句：`go env -w GOPROXY=https://goproxy.cn,direct`。
  - 设置为 “off” ：禁止 Go 在后续操作中使用任 何 Go module proxy。

刚刚在上面，我们可以发现值列表中有 “direct” ，它又有什么作用呢。其实值列表中的 “direct” 为特殊指示符，用于指示 Go 回源到模块版本的源地址去抓取(比如 GitHub 等)，当值列表中上一个 Go module proxy 返回 404 或 410 错误时，Go 自动尝试列表中的下一个，遇见 “direct” 时回源，遇见 EOF 时终止并抛出类似 “invalid version: unknown revision...” 的错误。

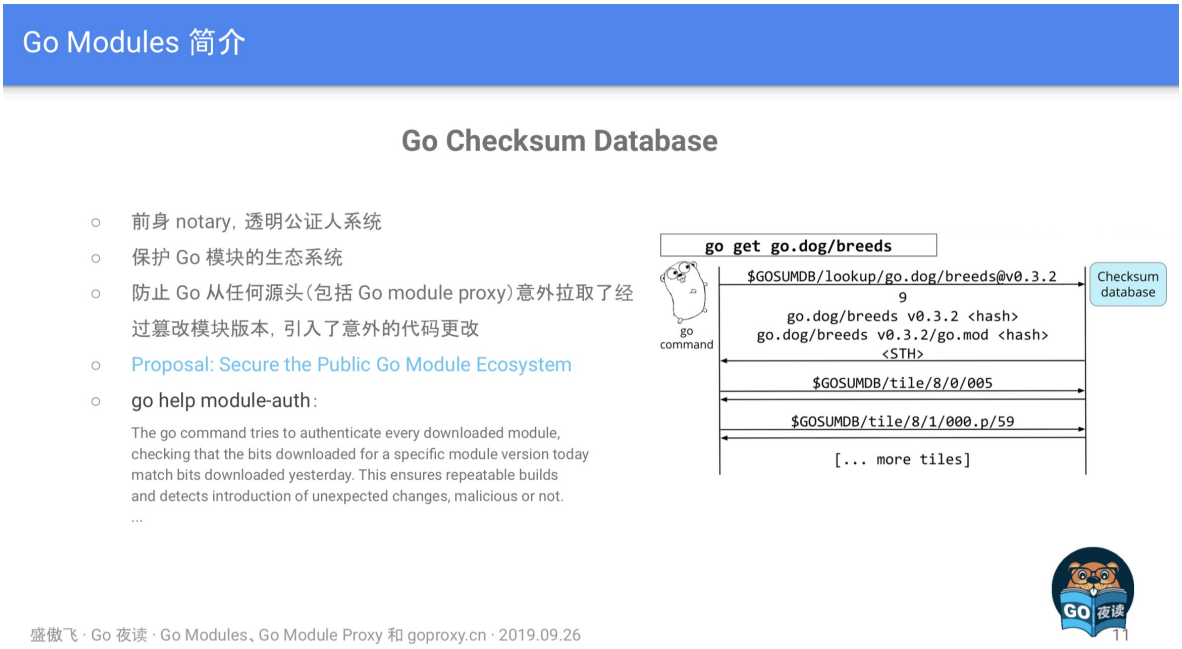
GOSUMDB

它的值是一个 Go checksum database，用于使 Go 在拉取模块版本时(无论是从源站拉取还是通过 Go module proxy 拉取)保证拉取到的模块版本数据未经篡改，也可以是“off”即禁止 Go 在后续操作中校验模块版本

- 格式 1：`<SUMDB_NAME>+<PUBLIC_KEY>`。
- 格式 2：`<SUMDB_NAME>+<PUBLIC_KEY> <SUMDB_URL>`。
- 拥有默认值：`sum.golang.org`(之所以没有按照上面的格式是因为 Go 对默认值做了特殊处理)。
- 可被 Go module proxy 代理 (详见：Proxying a Checksum Database)。
- `sum.golang.org` 在中国无法访问，故而更加建议将 GOPROXY 设置为 `goproxy.cn`，因为 `goproxy.cn` 支持代理 `sum.golang.org`。

Go Checksum Database

Go checksum database 主要用于保护 Go 不会从任何源头拉到被篡改过的非法 Go 模块版本，其作用（左）和工作机制（右）如下图：



如果有兴趣的小伙伴可以看看 Proposal: Secure the Public Go Module Ecosystem，有详细介绍其算法机制，如果想简单一点，查看 `go help module-auth` 也是一个不错的选择。

GONOPROXY/GONOSUMDB/GOPRIVATE

这三个环境变量都是用在当前项目依赖了私有模块，也就是依赖了由 GOPROXY 指定的 Go module proxy 或由 GOSUMDB 指定 Go checksum database 无法访问到的模块时的场景

- 它们三个的值都是一个以英文逗号 “,” 分割的模块路径前缀，匹配规则同 path.Match。

- 其中 GOPRIVATE 较为特殊，它的值将作为 GONOPROXY 和 GONOSUMDB 的默认值，所以建议的最佳姿势是只是用 GOPRIVATE。

在使用上来讲，比如 `GOPRIVATE=*.corp.example.com` 表示所有模块路径以 `corp.example.com` 的下一级域名 (如 `team1.corp.example.com`) 为前缀的模块版本都将不经过 Go module proxy 和 Go checksum database，需要注意的是不包括 `corp.example.com` 本身。

Global Caching

这个主要是针对 Go modules 的全局缓存数据说明，如下：

- 同一个模块版本的数据只缓存一份，所有其他模块共享使用。
- 目前所有模块版本数据均缓存在 `$GOPATH/pkg/mod` 和 `$GOPATH/pkg/sum` 下，未来或将移至 `$GOCACHE/mod` 和 `$GOCACHE/sum` 下(可能会在当 `$GOPATH` 被淘汰后)。
- 可以使用 `go clean -modcache` 清理所有已缓存的模块版本数据。

另外在 Go1.11 之后 GOCACHE 已经不允许设置为 off 了，我想着这也是为了模块数据缓存移动位置做准备，因此大家应该尽快做好适配。

快速迁移项目至 Go Modules

- 第一步: 升级到 Go 1.13。
- 第二步: 让 GOPATH 从你的脑海中完全消失，早一步踏入未来。
  - 修改 GOBIN 路径（可选）：`go env -w GOBIN=$HOME/bin`。
  - 打开 Go modules：`go env -w GO111MODULE=on`。
  - 设置 GOPROXY：`go env -w GOPROXY=https://goproxy.cn,direct` # 在中国是必须的，因为它的默认值被墙了。
- 第三步(可选): 按照你喜欢的目录结构重新组织你的所有项目。
- 第四步: 在你项目的根目录下执行 `go mod init <OPTIONAL_MODULE_PATH>` 以生成 go.mod 文件。
- 第五步: 想办法说服你身边所有的人都去走一下前四步。

迁移后 go get 行为的改变

- 用 `go help module-get` 和 `go help gopath-get` 分别去了解 Go modules 启用和未启用两种状态下的 go get 的行为
- 用 `go get` 拉取新的依赖
  - 拉取最新的版本(优先择取 tag)：`go get golang.org/x/text@latest`
  - 拉取 `master` 分支的最新 commit：`go get golang.org/x/text@master`
  - 拉取 tag 为 v0.3.2 的 commit：`go get golang.org/x/text@v0.3.2`
  - 拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2：`go get golang.org/x/text@342b2e`
  - 用 `go get -u` 更新现有的依赖
  - 用 `go mod download` 下载 go.mod 文件中指明的所有依赖
  - 用 `go mod tidy` 整理现有的依赖
  - 用 `go mod graph` 查看现有的依赖结构
  - 用 `go mod init` 生成 go.mod 文件 (Go 1.13 中唯一一个可以生成 go.mod 文件的子命令)
- 用 `go mod edit` 编辑 go.mod 文件
- 用 `go mod vendor` 导出现有的所有依赖 (事实上 Go modules 正在淡化 Vendor 的概念)
- 用 `go mod verify` 校验一个模块是否被篡改过

这里我们注意到有两点比较特别，分别是：

- 第一点：为什么“拉取 hash 为 342b231 的 commit，最终会被转换为 v0.3.2”呢。这是因为虽然我们设置了拉取 @342b2e commit，但是因为 Go modules 会与 tag 进行对比，若发现对应的 commit 与 tag 有关联，则进行转换。
- 第二点：为什么不建议使用 `go mod vendor`，因为 Go modules 正在淡化 Vendor 的概念，很有可能 Go2 就去掉了。

使用 Go Modules 时常遇见的坑

坑 1: 判断项目是否启用了 Go Modules

使用 Go Modules 时常遇见的坑(坑 1:判断项目是否启用了 Go Modules)

- 切记，在 Go 1.13 中，一个项目只要包含了 go.mod 文件，且它所处环境中的 GO111MODULE 不为 off，那么 Go 就会为这个项目启用 Go modules
- 为了一劳永逸地解决这个判断烦恼，建议大家将 GO111MODULE 设置为 on
  - go env -w GO111MODULE=on
- 另外，若当前项目没有包含 go.mod 文件，且 GO111MODULE 为 on，那么每一次构建代码时 Go 都会从头推算并拉取所需要的模块版本，**但是并不会自动生成 go.mod 文件**(以前会自动生成，Go 1.13 做了调整)

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 18

坑 2: 管理 Go 的环境变量

使用 Go Modules 时常遇见的坑(坑 2:管理 Go 的环境变量)

- 在 Go 1.13 中管理环境变量会变得稍显混乱，因为 Go 1.13 建议将所有跟 Go 相关的环境变量都交由新出的 go env -w 来管理，比如执行 go env -w GO111MODULE=on 就会在 $HOME/.config/go/env 文件中追加一行 “GO111MODULE=on”
- **但 go env -w 不会覆盖你的系统环境变量**
- 所以建议大家在升级到 Go 1.13 后做一次环境变量的管理方式的转变，比如删除你系统中所有跟 Go 相关的环境变量，并使用 go env -w 重写会
- os.UserConfigDir:

> UserConfigDir returns the default root directory to use for user-specific configuration data. Users should create their own application-specific subdirectory within this one and use that.
>
> On Unix systems, it returns $XDG_CONFIG_HOME as specified by https://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html if non-empty, else $HOME/.config. On Darwin, it returns $HOME/Library/Application Support. On Windows, it returns %AppData%. On Plan 9, it returns $home/lib.
>
> If the location cannot be determined (for example, $HOME is not defined), then it will return an error.

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 19

这里主要是提到 Go1.13 新增了 `go env -w` 用于写入环境变量，而写入的地方是 `os.UserConfigDir` 所返回的路径，需要注意的是 `go env -w` 不会覆写。

坑 3: 从 dep、glide 等迁移至 Go Modules

使用 Go Modules 时常遇见的坑(坑 3:从 dep、glide 等迁移至 Go Modules)

- 在执行 go mod init <OPTIONAL_MODULE_PATH> 时，如果当前目录包含了 Gopkg.lock、glide.lock 等老的依赖管理文件时，Go 会用它们所指定的依赖版本信息来推算并生成 go.mod 文件，此时所有的推算行为都是直接回源的，并不会经过 Go module proxy，这是个 Bug，详见:golang.org/issue/33767
- 不出意外的话这个 Bug 将在 Go 1.14 中被修复，我们正在努力
- 临时的解决办法时先按照 go help go.mod 里指定的规则手动创建一个 go.mod 文件(可以只包含第一行的模块路径)，然后再执行 go mod tidy 来推算并补充 go.mod 文件，这样儿就可以走 Go module proxy 了

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 20

这里主要是指从旧有的依赖包管理工具（dep/glide 等）进行迁移时，因为 BUG 的原因会导致不经过 GOPROXY 的代理，解决方法有如下两个：

- 手动创建一个 go.mod 文件，再执行 go mod tidy 进行补充。
- 上代理，相当于不使用 GOPROXY 了。

坑 4:拉取私有模块

使用 Go Modules 时常遇见的坑(坑 4:拉取私有模块)

- 常见的公共 Go module proxy，比如 proxy.golang.org 和 goproxy.cn 都是无权访问任何人的私有模块的
- 即使不使用任何 Go module proxy，也就是将 GOPROXY 设置为了 direct，同样默认情况下 Go 也是无法抓取你的私有模块的，详见:golang.org/doc/faq#git_https
- 解决办法是想办法修改你的 VCS 拉取行为，比如使用 Git 时就在 $HOME/.gitconfig 文件中追加:

```
[url "ssh://git@github.com/"]
        insteadOf = https://github.com/
```

- 并为 GOPROXY 设置一个 fallback 选项(当列表中的前一个 404 或 410 时就自动尝试下一个，遇见 direct 时回源也就是直接去模块所在地拉取)
  - go env -w GOPROXY=https://goproxy.cn,direct
- 还可以通过设置 GONOPROXY 或 GOPRIVATE 来指使 Go 在拉取哪些模块时忽略 Go module proxy

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 21

这里主要想涉及两块知识点，如下：

- GOPROXY 是无权访问到任何人的私有模块的，所以你放心，安全性没问题。
- GOPROXY 除了设置模块代理的地址以外，还需要增加 “direct” 特殊标识才可以成功拉取私有库。

坑 5:更新现有的模块

使用 Go Modules 时常遇见的坑(坑 5:更新现有的模块)

- go get -u
  - 只会更新主要模块，忽略了单元测试
- go get -u ./...
  - 递归更新所有子目录的所有模块，忽略了单元测试
- go get -u -t
  - 同样是只会更新主要模块，但考虑了单元测试
- go get -u -t ./...
  - 同样是递归更新所有子目录的所有模块，但考虑了单元测试
- go get -u all
  - 更新所有模块，推荐使用

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 22

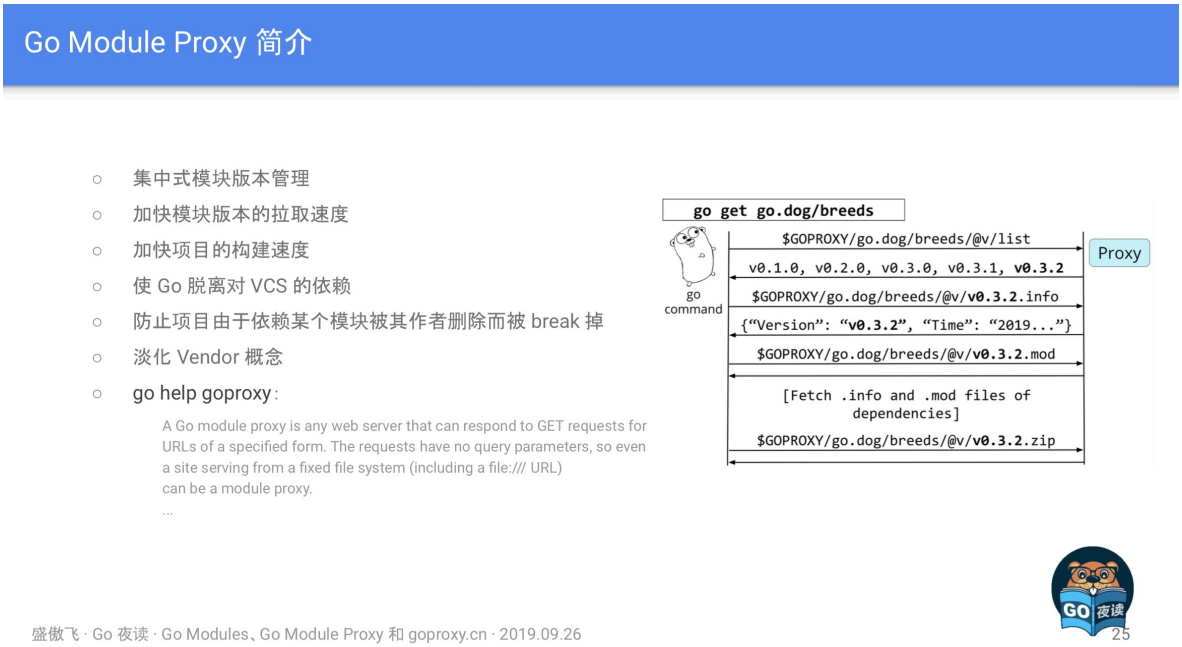
坑 6:主版本号

使用 Go Modules 时常遇见的坑(坑 6:主版本号)

- 除了 v0 和 v1 外主版本号必须显示地出现在模块路径的尾部
- Go 会将 example.com/foo/bar 和 example.com/foo/bar/v2 视为完全不同的两个模块来对待
- 一个项目中可能同时存在多个拥有不同的主版本号的相同模块，但互不影响
- go get -u 不会更新主版本号，如需更新，需要手动修改代码中对应的导入路径

盛傲飞 · Go 夜读 · Go Modules、Go Module Proxy 和 goproxy.cn · 2019.09.26 23

Go Module Proxy 简介



在这里再次强调了 Go Module Proxy 的作用（图左），以及其对应的协议交互流程（图右），有兴趣的小伙伴可以认真看一下。

Goproxy 中国(goproxy.cn)

在这块主要介绍了 Goproxy 的实践操作以及 goproxy.cn 的一些 Q&A 和 近况，如下：

Q&A

Q：如果中国 Go 语言社区没有咱们自己家的 Go Module Proxy 会怎么样？

A：在 Go 1.13 中 GOPROXY 和 GOSUMDB 这两个环境变量都有了在中国无法 访问的默认值，尽管我在 golang.org/issue/31755 里努力尝 试过，但最终仍然无法为咱们中国的 Go 语言开发者谋得一个完美的解决方案。所以从今以后咱 们中国的所有 Go 语言开发者，只要是 使用了 Go modules 的，那么都必须先修改 GOPROXY 和 GOSUMDB 才能正常使用 Go 做开发，否则可能连一个最简单的程序都跑不起 来(只要它有依 赖第三方模 块)。

Q： 我创建 Goproxy 中国(goproxy.cn)的主要原因？

A：其实更早的时候，也就是今年年初我也曾 试图在 golang.org/issue/31020 中请求 Go team 能想办法避免那时的 GOPROXY 即将拥有的默认值可以在中国正常访问，但 Go team 似乎也无能为力，为此我才坚定了创建 goproxy.cn 的信念。既然别人没法儿帮忙，那咱们就得自己动手，不为别的，就为了让大家以后能够更愉快地使用 Go 语言配合 Go modules 做开发。

最初我先是和七牛云的 许叔(七牛云的 创始人兼 CEO 许式伟)提出了我打算 创建 goproxy.cn 的想法，本是抱着 试试看的目的，但没想 到 许叔几乎是没有超过一分钟的考虑便认可了我的想

法并表示愿意一起推 动。那一阵子刚好赶上我在写毕业论文，所以项目开发完后就 一直没和七牛云做交接，一直跑在我的个人服 务器上。直到有一次 goproxy.cn 被攻击了，一下午的功夫烧了我一百多美元，然后我才 意识到这种项目真不能个人来做。个人来做不靠 谱，万一依赖这个项目的人多了，项目再出什么事儿，那就会给大家□成不必要的损 失。所以我赶紧和七牛云做了交接，把 goproxy.cn 完全交给了七牛云，甚至连域名都过户了去。

近况

区域	最快/最慢	平均
全网	北京(鹏博士) 0.2ms/柬埔寨金边(cambo.host) 360.8ms	25.6ms
移动	安徽合肥(hccn2) 3.3ms/广东广州(移动) 35.8ms	15.1ms
联通	天津(联通) 6.7ms/重庆(联通) 27.7ms	17.2ms
电信	四川成都(天翼云) 1.5ms/新疆乌鲁木齐(天翼云) 66.3ms	27.4ms
华南	广东广州(腾讯云) 9.0ms / 海南海口(天翼云) 49.9ms	25.3ms
华北	北京(鹏博士) 0.2ms / 内蒙古呼和浩特(蓝视云) 45.4ms	24.0ms
华东	上海(阿里云) 1.0ms / 山东青岛(天翼云) 48.2ms	20.1ms
华中	湖南长沙(天翼云) 5.4ms / 河南郑州(景安网络) 58.9ms	20.3ms
东北	辽宁鞍山(电信) 19.6ms / 吉林长春(天翼云) 57.7ms	33.3ms
西北	陕西西安(天翼云一区) 13.8ms / 新疆乌鲁木齐(天翼云) 66.3ms	41.7ms
西南	四川成都(天翼云) 1.5ms / 贵州贵阳(华为云) 58.5ms	26.6ms
港澳台	台湾台北(远传电信) 0.6ms / 香港(hkc) 155.6ms	27.9ms
亚洲	新加坡(腾讯云) 0.4ms/柬埔寨金边(cambo.host) 360.8ms	15.3ms
欧洲	意大利米兰 0.4ms/法国巴黎(VULTR) 146.5ms	22.1ms
非洲	南非约翰内斯堡 0.2ms/尼日利亚拉各斯 265.2ms	48.9ms
北美洲	美国西雅图(VULTR) 0.3ms/美国波特兰(AMAZON-LIGHTSAIL) 97.2ms	34.9ms
南美洲	巴西圣保罗(GCP) 3.7ms/阿根廷布宜诺斯艾利斯(latincloud.com) 145.0ms	50.9ms
大洋洲	澳大利亚悉尼(xtom.com) 0.7ms/澳大利亚阿德莱德 52.6ms	15.1ms

- Goproxy 中国 (goproxy.cn) 是目前中国最可靠的 Go module proxy (真不是在自卖自夸)。
- 为中国 Go 语言开发者量身打□，支持代理 GOSUMDB 的默认值，经过全球 CDN 加速，高可用，可 应用进公司复杂的开发环境中，亦可用作上游代理。
- 由中国倍受信赖的云服务提供商七牛云无偿提供基础设施支持的开源的非营利性项目。
- 目标是为中国乃至全世界的 Go 语言开发者提供一个免 费的、可靠的、持 续在线的且经过 CDN 加□的 Go module proxy。
- 域名已由七牛云进行了备案 (沪ICP备11037377号-56)。

情况

此处呈现的是存储大小，主要是针对模块包代码，而一般来讲代码并不会有多大，0-10MB，10-50MB 占最大头，也是能够理解，但是大于 100MB 的模块包代码就比较夸张了。

此时主要是展示了一下近期 goproxy.cn 的网络数据情况，我相信未来是会越来越高的，值得期待。

Q&A

Q：如何解决 Go 1.13 在从 GitLab 拉取模块版本时遇到的，Go 错误地按照非期望值的路径寻找目标模块版本结果致使最终目标模块拉取失败的问题？

A：GitLab 中配合 goget 而设置的 `<meta>` 存在些许问题，导致 Go 1.13 错误地识别了模块的具体路径，这是个 Bug，据说在 GitLab 的新版本中已经被修复了，详细内容可以看 https://github.com/golang/go/issues/34094 这个 Issue。然后目前的解决办法的话除了升级 GitLab 的版本外，还可以参考 https://github.com/developer-learning/night-reading-go/issues/468#issuecomment-535850154 这条回复。

Q：使用 Go modules 时可以同时依赖同一个模块的不同的两个或者多个小版本（修订版本号不同）吗？

A：不可以的，Go modules 只可以同时依赖一个模块的不同的两个或者多个大版本（主版本号不同）。比如可以同时依赖 foobar@v1.2.3"">example.com/foobar@v1.2.3 和 v2@v2.3.4"">example.com/foobar/v2@v2.3.4，因为他们的模块路径（module path）不同，Go modules 规定主版本号不是 v0 或者 v1 时，那么主版本号必须显式地出现在模块路径的尾部。但是，同时依赖两个或者多个小版本是不支持的。比如如果模块 A 同时直接依赖了模块 B 和模块 C，且模块 A 直接依赖的是模块 C 的 v1.0.0 版本，然后模块 B 直接依赖的是模块 C 的 v1.0.1 版本，那么最终 Go modules 会为模块 A 选用模块 C 的 v1.0.1 版本而不是模块 A 的 go.mod 文件中指明的 v1.0.0 版本。

这是因为 Go modules 认为只要主版本号不变，那么剩下的都可以直接升级采用最新的。但是如果采用了最新的结果导致项目 Break 掉了，那么 Go modules 就会 Fallback 到上一个老的版本，比如在前面的例子中就会 Fallback 到 v1.0.0 版本。

Q：在 go.sum 文件中的一个模块版本的 Hash 校验数据什么情况下会成对出现，什么情况下只会存在一行？

A：通常情况下，在 go.sum 文件中的一个模块版本的 Hash 校验数据会有两行，前一行是该模块的 ZIP 文件的 Hash 校验数据，后一行是该模块的 go.mod 文件的 Hash 校验数据。但是也有些情况下只会出现一行该模块的 go.mod 文件的 Hash 校验数据，而不包含该模块的 ZIP 文件本身的 Hash 校验数据，这个情况发生在 Go modules 判定为你当前这个项目完全用不到该模块，根本也不会下载该模块的 ZIP 文件，所以就没必要对其作出 Hash 校验保证，只需要对该模块的 go.mod 文件作出 Hash 校验保证即可，因为 go.mod 文件是用得着的，在深入挖取项目依赖的时候要用。

Q：能不能更详细地讲解一下 go.mod 文件中的 replace 动词的行为以及用法？

A：这个 replace 动词的作用是把一个“模块版本”替换为另外一个“模块版本”，这是“模块版本”和“模块版本（module path）”之间的替换，“=>”标识符前面的内容是待替换的“模块版本”的“模块路径”，后面的内容是要替换的目标“模块版本”的所在地，即路径，这个路径可以是一个本地磁盘的相对路径，也可以是一个本地磁盘的绝对路径，还可以是一个网络路径，但是这个目标路径并不会在今后你的项目代码中作为你“导入路径（import path）”出现，代码里的“导入路径”还是得以你替换成的这个目标“模块版本”的“模块路径”作为前缀。

另外需要注意，Go modules 是不支持在 “导入路径” 里写相对路径的。举个例子，如果项目 A 依赖了模块 B，比如模块 B 的“模块路径”是 example.com/b，然后它在的磁盘路径是 ~~/b，在项目 A 里的 go.mod 文件中你有一行 replace example.com/b=>~~/b，然后在项目 A 里的代

码中的“导入路基”就是 import”example.com/b”，而不是 import”~/b”，剩下的工作是 Go modules 帮你自动完成了的。

然后就是我在分享中也提到了， exclude 和 replace 这两个动词只作用于当前主模块，也就是当前项目，它所依赖的那些其他模块版本中如果出现了你待替换的那个模块版本的话，Go modules 还是会为你依赖的那个模块版本去拉取你的这个待替换的模块版本。

举个例子，比如项目 A 直接依赖了模块 B 和模块 C，然后模块 B 也直接依赖了模块 C，那么你在项目 A 中的 go.mod 文件里的 replace c=>~/some/path/c 是只会影响项目 A 里写的代码中，而模块 B 所用到的还是你 replace 之前的那个 c，并不是你替换成的 ~/some/path/c 这个。

总结

在 Go1.13 发布后，接触 Go modules 和 Go module proxy 的人越来越多，经常在各种群看到各种小伙伴在咨询，包括我自己也贡献了好几枚 “坑”，因此我觉得傲飞的这一次 《Go Modules、Go Module Proxy 和 goproxy.cn》的技术分享，非常的有实践意义。如果后续大家还有什么建议或问题，欢迎随时来讨论。

最后，感谢 goproxy.cn 背后的人们（@七牛云 和 @盛傲飞）对中国 Go 语言社区的无私贡献和奉献。

进一步阅读

- night-reading-go/issues/468
- B站：【Go 夜读】第 61 期 Go Modules、Go Module Proxy 和 goproxy.cn
- youtube：【Go 夜读】第 61 期 Go Modules、Go Module Proxy 和 goproxy.cn

tools

Go 大杀器之性能剖析 PProf

前言

写了几吨代码，实现了几百个接口。功能测试也通过了，终于成功的部署上线了

结果，性能不佳，什么鬼？😨

想做性能分析

PProf

想要进行性能优化，首先瞩目在 Go 自身提供的工具链来作为分析依据，本文将带你学习、使用 Go 后花园，涉及如下：

- runtime/pprof：采集程序（非 Server）的运行数据进行分析
- net/http/pprof：采集 HTTP Server 的运行时数据进行分析

是什么

pprof 是用于可视化和分析性能分析数据的工具

pprof 以 profile.proto 读取分析样本的集合，并生成报告以可视化并帮助分析数据（支持文本和图形报告）

profile.proto 是一个 Protocol Buffer v3 的描述文件，它描述了一组 callstack 和 symbolization 信息， 作用是表示统计分析的一组采样的调用栈，是很常见的 stacktrace 配置文件格式

支持什么使用模式

- Report generation：报告生成
- Interactive terminal use：交互式终端使用
- Web interface：Web 界面

可以做什么

- CPU Profiling：CPU 分析，按照一定的频率采集所监听的应用程序 CPU（含寄存器）的使用情况，可确定应用程序在主动消耗 CPU 周期时花费时间的位置

- Memory Profiling：内存分析，在应用程序进行堆分配时记录堆栈跟踪，用于监视当前和历史内存使用情况，以及检查内存泄漏
- Block Profiling：阻塞分析，记录 goroutine 阻塞等待同步（包括定时器通道）的位置
- Mutex Profiling：互斥锁分析，报告互斥锁的竞争情况

一个简单的例子

我们将编写一个简单且有点问题的例子，用于基本的程序初步分析

编写 demo 文件

（1）demo.go，文件内容：

```
package main

import (
    "log"
    "net/http"
    _ "net/http/pprof"
    "github.com/EDDYCJY/go-pprof-example/data"
)

func main() {
    go func() {
        for {
            log.Println(data.Add("https://github.com/EDDYCJY"))
        }
    }()

    http.ListenAndServe("0.0.0.0:6060", nil)
}
```

（2）data/d.go，文件内容：

```
package data

var datas []string

func Add(str string) string {
    data := []byte(str)
    sData := string(data)
    datas = append(datas, sData)
```

```
    return sData
}
```

运行这个文件，你的 HTTP 服务会多出 /debug/pprof 的 endpoint 可用于观察应用程序的情况

分析

一、通过 Web 界面

查看当前总览：访问 `http://127.0.0.1:6060/debug/pprof/`

```
/debug/pprof/

profiles:
0    block
5    goroutine
3    heap
0    mutex
9    threadcreate

full goroutine stack dump
```

这个页面中有许多子页面，咱们继续深究下去，看看可以得到什么？

- cpu（CPU Profiling）：`$HOST/debug/pprof/profile`，默认进行 30s 的 CPU Profiling，得到一个分析用的 profile 文件
- block（Block Profiling）：`$HOST/debug/pprof/block`，查看导致阻塞同步的堆栈跟踪
- goroutine：`$HOST/debug/pprof/goroutine`，查看当前所有运行的 goroutines 堆栈跟踪
- heap（Memory Profiling）：`$HOST/debug/pprof/heap`，查看活动对象的内存分配情况
- mutex（Mutex Profiling）：`$HOST/debug/pprof/mutex`，查看导致互斥锁的竞争持有者的堆栈跟踪
- threadcreate：`$HOST/debug/pprof/threadcreate`，查看创建新 OS 线程的堆栈跟踪

二、通过交互式终端使用

（1）go tool pprof http://localhost:6060/debug/pprof/profile?seconds=60

```
$ go tool pprof http://localhost:6060/debug/pprof/profile\?seconds\=60

Fetching profile over HTTP from http://localhost:6060/debug/pprof/profile?second
s=60
Saved profile in /Users/eddycjy/pprof/pprof.samples.cpu.007.pb.gz
Type: cpu
Duration: 1mins, Total samples = 26.55s (44.15%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

执行该命令后，需等待 60 秒（可调整 seconds 的值），pprof 会进行 CPU Profiling。结束后将默认进入 pprof 的交互式命令模式，可以对分析的结果进行查看或导出。具体可执行 `pprof help` 查看命令说明

```
(pprof) top10
Showing nodes accounting for 25.92s, 97.63% of 26.55s total
Dropped 85 nodes (cum <= 0.13s)
Showing top 10 nodes out of 21
      flat  flat%   sum%        cum   cum%
    23.28s 87.68% 87.68%     23.29s 87.72%  syscall.Syscall
     0.77s  2.90% 90.58%      0.77s  2.90%  runtime.memmove
     0.58s  2.18% 92.77%      0.58s  2.18%  runtime.freedefer
     0.53s  2.00% 94.76%      1.42s  5.35%  runtime.scanobject
     0.36s  1.36% 96.12%      0.39s  1.47%  runtime.heapBitsForObject
     0.35s  1.32% 97.44%      0.45s  1.69%  runtime.greyobject
     0.02s 0.075% 97.51%     24.96s 94.01%  main.main.func1
     0.01s 0.038% 97.55%     23.91s 90.06%  os.(*File).Write
     0.01s 0.038% 97.59%      0.19s  0.72%  runtime.mallocgc
     0.01s 0.038% 97.63%     23.30s 87.76%  syscall.Write
```

- flat：给定函数上运行耗时
- flat%：同上的 CPU 运行耗时总比例
- sum%：给定函数累积使用 CPU 总比例
- cum：当前函数加上它之上的调用运行总耗时
- cum%：同上的 CPU 运行耗时总比例

最后一列为函数名称，在大多数的情况下，我们可以通过这五列得出一个应用程序的运行情况，加以优化

（2）go tool pprof http://localhost:6060/debug/pprof/heap

```
$ go tool pprof http://localhost:6060/debug/pprof/heap
Fetching profile over HTTP from http://localhost:6060/debug/pprof/heap
Saved profile in /Users/eddycjy/pprof/pprof.alloc_objects.alloc_space.inuse_objects.inuse_space.008.pb.gz
Type: inuse_space
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 837.48MB, 100% of 837.48MB total
      flat  flat%   sum%        cum   cum%
  837.48MB   100%   100%   837.48MB   100%  main.main.func1
```

- -inuse_space：分析应用程序的常驻内存占用情况
- -alloc_objects：分析应用程序的内存临时分配情况

（3） go tool pprof http://localhost:6060/debug/pprof/block

（4） go tool pprof http://localhost:6060/debug/pprof/mutex

三、PProf 可视化界面

这是令人期待的一小节。在这之前，我们需要简单的编写好测试用例来跑一下

编写测试用例

（1）新建 data/d_test.go，文件内容：

```
package data

import "testing"

const url = "https://github.com/EDDYCJY"

func TestAdd(t *testing.T) {
    s := Add(url)
    if s == "" {
        t.Errorf("Test.Add error!")
    }
}

func BenchmarkAdd(b *testing.B) {
    for i := 0; i < b.N; i++ {
        Add(url)
    }
}
```

（2）执行测试用例

```
$ go test -bench=. -cpuprofile=cpu.prof
pkg: github.com/EDDYCJY/go-pprof-example/data
BenchmarkAdd-4       10000000               187 ns/op
PASS
ok      github.com/EDDYCJY/go-pprof-example/data    2.300s
```

-memprofile 也可以了解一下

启动 PProf 可视化界面

方法一：

```
$ go tool pprof -http=:8080 cpu.prof
```

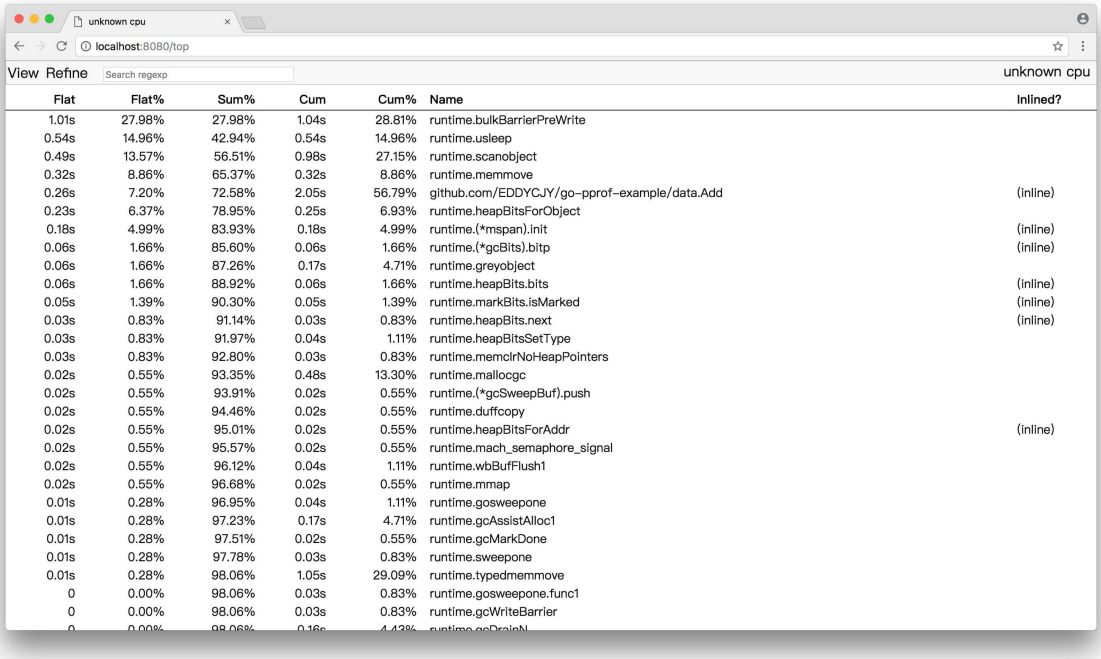
方法二：

```
$ go tool pprof cpu.prof
$ (pprof) web
```

如果出现 `Could not execute dot; may need to install graphviz.` ，就是提示你要安装 `graphviz` 了（请右拐谷歌）

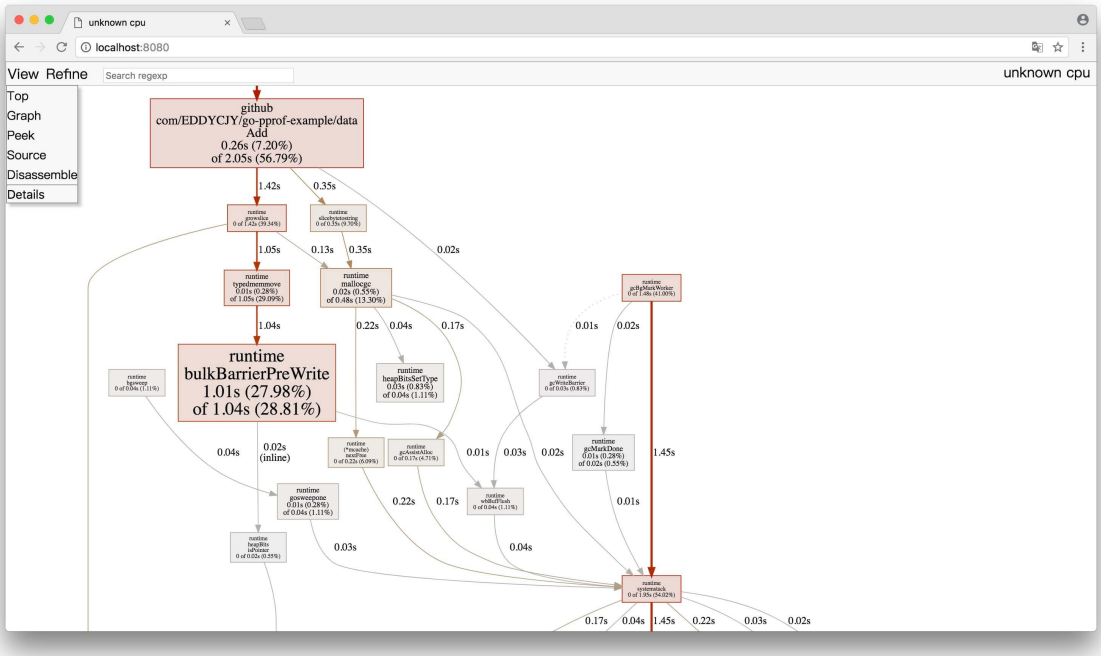
查看 PProf 可视化界面

（1）Top



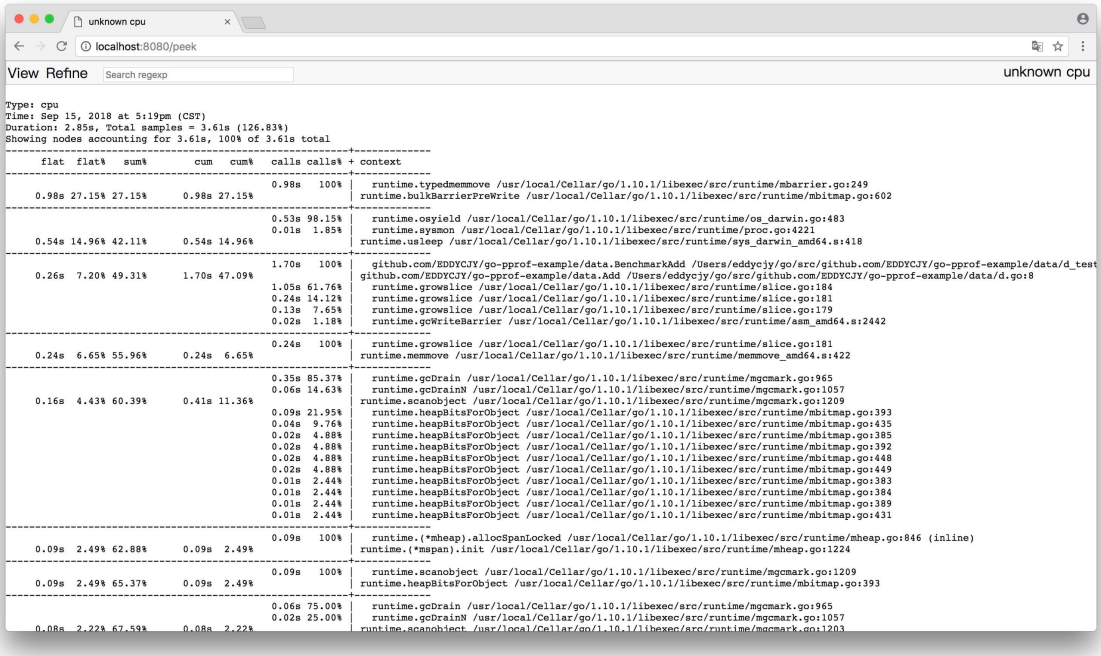
Flat	Flat%	Sum%	Cum	Cum%	Name	Inlined?
1.01s	27.98%	27.98%	1.04s	28.81%	runtime.bulkBarrierPreWrite	
0.54s	14.96%	42.94%	0.54s	14.96%	runtime.usleep	
0.49s	13.57%	56.51%	0.98s	27.15%	runtime.scanobject	
0.32s	8.86%	65.37%	0.32s	8.86%	runtime.memmove	
0.26s	7.20%	72.58%	2.05s	56.79%	github.com/EDDYCJY/go-pprof-example/data.Add	(inline)
0.23s	6.37%	78.95%	0.25s	6.93%	runtime.heapBitsForObject	
0.18s	4.99%	83.93%	0.18s	4.99%	runtime.(*mspan).init	(inline)
0.06s	1.66%	85.60%	0.06s	1.66%	runtime.(*gcBits).bitp	(inline)
0.06s	1.66%	87.26%	0.17s	4.71%	runtime.greyobject	
0.06s	1.66%	88.92%	0.06s	1.66%	runtime.heapBits.bits	(inline)
0.05s	1.39%	90.30%	0.05s	1.39%	runtime.markBits.isMarked	(inline)
0.03s	0.83%	91.14%	0.03s	0.83%	runtime.heapBits.next	(inline)
0.03s	0.83%	91.97%	0.04s	1.11%	runtime.heapBitsSetType	
0.03s	0.83%	92.80%	0.03s	0.83%	runtime.memclrNoHeapPointers	
0.02s	0.55%	93.35%	0.48s	13.30%	runtime.mallocgc	
0.02s	0.55%	93.91%	0.02s	0.55%	runtime.(*gcSweepBuf).push	
0.02s	0.55%	94.46%	0.02s	0.55%	runtime.duffcopy	
0.02s	0.55%	95.01%	0.02s	0.55%	runtime.heapBitsForAddr	(inline)
0.02s	0.55%	95.57%	0.02s	0.55%	runtime.mach_semaphore_signal	
0.02s	0.55%	96.12%	0.04s	1.11%	runtime.wbBufFlush1	
0.02s	0.55%	96.68%	0.02s	0.55%	runtime.mmap	
0.01s	0.28%	96.95%	0.04s	1.11%	runtime.gosweepone	
0.01s	0.28%	97.23%	0.17s	4.71%	runtime.gcAssistAlloc1	
0.01s	0.28%	97.51%	0.02s	0.55%	runtime.gcMarkDone	
0.01s	0.28%	97.78%	0.03s	0.83%	runtime.sweepone	
0.01s	0.28%	98.06%	1.05s	29.09%	runtime.typedmemmove	
0	0.00%	98.06%	0.03s	0.83%	runtime.gosweepone.func1	
0	0.00%	98.06%	0.03s	0.83%	runtime.gcWriteBarrier	

（2）Graph



框越大，线越粗代表它占用的时间越大哦

（3）Peek



（4）Source

通过 PProf 的可视化界面，我们能够更方便、更直观的看到 Go 应用程序的调用链、使用情况等，并且在 View 菜单栏中，还支持如上多种方式的切换

你想想，在烦恼不知道什么问题的时候，能用这些辅助工具来检测问题，是不是瞬间效率翻倍了呢

四、PProf 火焰图

另一种可视化数据的方法是火焰图，需手动安装原生 PProf 工具：

（1） 安装 PProf

```
$ go get -u github.com/google/pprof
```

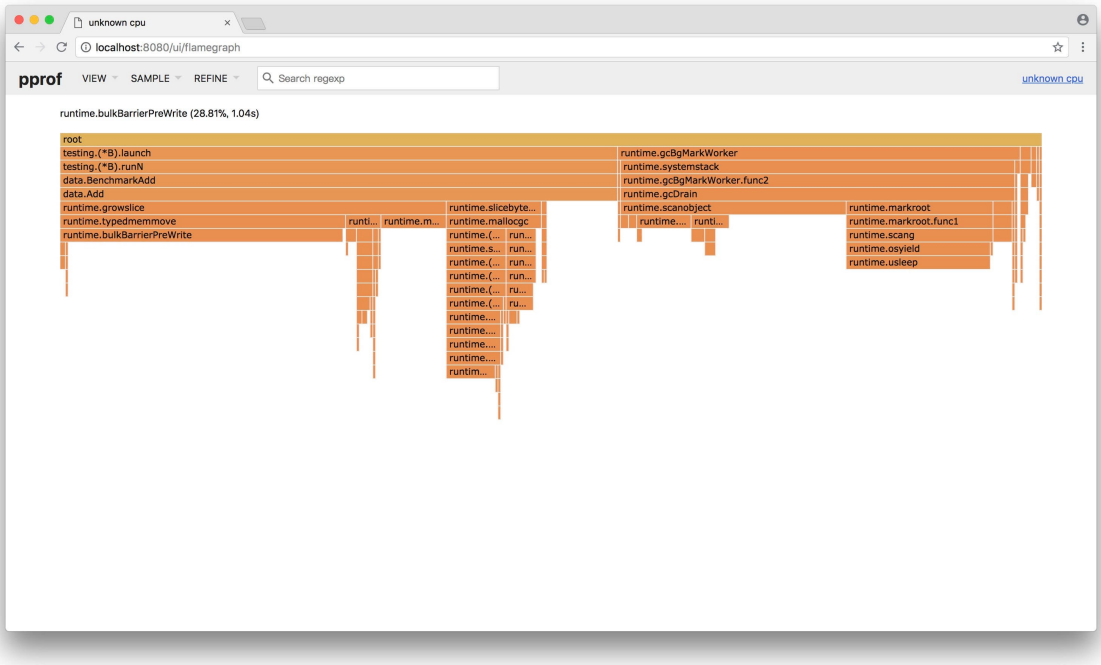
（2） 启动 PProf 可视化界面:

```
$ pprof -http=:8080 cpu.prof
```

（3） 查看 PProf 可视化界面

打开 PProf 的可视化界面时，你会明显发现比官方工具链的 PProf 精致一些，并且多了 Flame Graph（火焰图）

它就是本次的目标之一，它的最大优点是动态的。调用顺序由上到下（A -> B -> C -> D），每一块代表一个函数，越大代表占用 CPU 的时间更长。同时它也支持点击块深入进行分析！



总结

在本章节，粗略地介绍了 Go 的性能利器 PProf。在特定的场景中，PProf 给定位、剖析问题带了极大的帮助

希望本文对你有所帮助，另外建议能够自己实际操作一遍，最好是可以深入琢磨一下，内含大量的用法、知识点

思考题

你很优秀的看到了最后，那么有两道简单的思考题，希望拓展你的思路

（1）flat 一定大于 cum 吗，为什么？什么场景下 cum 会比 flat 大？

（2）本章节的 demo 代码，有什么性能问题？怎么解决它？

Go 大杀器之跟踪剖析 trace

在 Go 中有许许多多的分析工具，在之前我有写过一篇 《Golang 大杀器之性能剖析 PProf》来介绍 PProf，如果有小伙伴感兴趣可以去我博客看看。

但单单使用 PProf 有时候不一定足够完整，因为在真实的程序中还包含许多的隐藏动作，例如 Goroutine 在执行时会做哪些操作？执行/阻塞了多长时间？在什么时候阻止？在哪里被阻止的？谁又锁/解锁了它们？GC 是怎么影响到 Goroutine 的执行的？这些东西用 PProf 是很难分析出来的，但如果你又想知道上述的答案的话，你可以用本文的主角 `go tool trace` 来打开新世界的大门。目录如下：

初步了解

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

func main() {
    trace.Start(os.Stderr)
    defer trace.Stop()

    ch := make(chan string)
    go func() {
```

```
        ch <- "EDDYCJY"
    }()

    <-ch
}
```

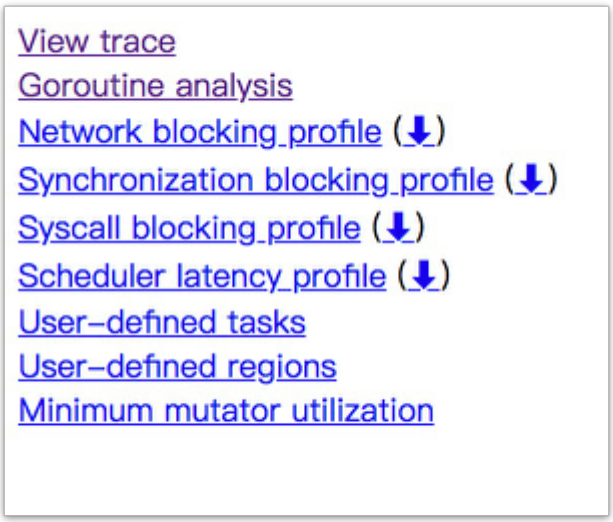
生成跟踪文件：

```
$ go run main.go 2> trace.out
```

启动可视化界面：

```
$ go tool trace trace.out
2019/06/22 16:14:52 Parsing trace...
2019/06/22 16:14:52 Splitting trace...
2019/06/22 16:14:52 Opening browser. Trace viewer is listening on http://127.0.0.1:57321
```

查看可视化界面：

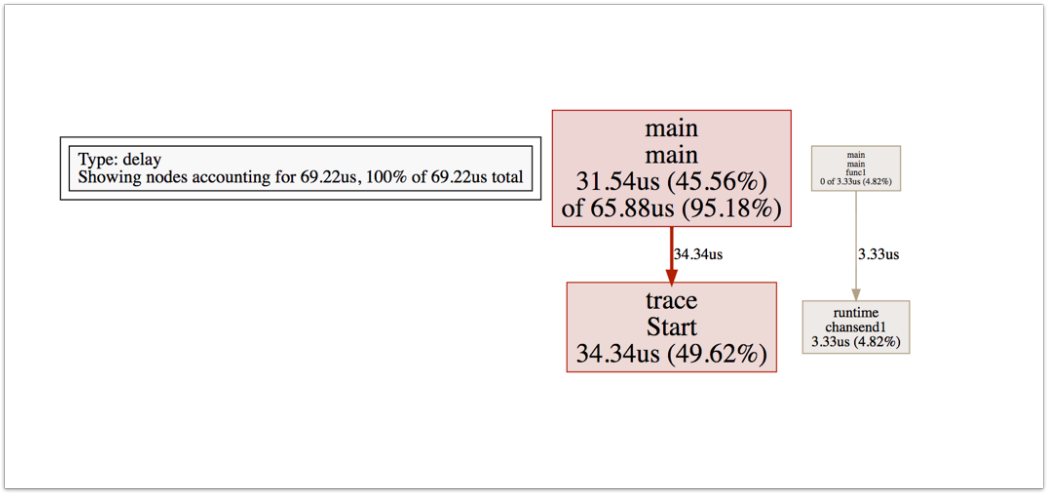


- View trace：查看跟踪
- Goroutine analysis：Goroutine 分析
- Network blocking profile：网络阻塞概况
- Synchronization blocking profile：同步阻塞概况
- Syscall blocking profile：系统调用阻塞概况

- Scheduler latency profile：调度延迟概况
- User defined tasks：用户自定义任务
- User defined regions：用户自定义区域
- Minimum mutator utilization：最低 Mutator 利用率

Scheduler latency profile

在刚开始查看问题时，除非是很明显的现象，否则不应该一开始就陷入细节，因此我们一般先查看 “Scheduler latency profile”，我们能通过 Graph 看到整体的调用开销情况，如下：

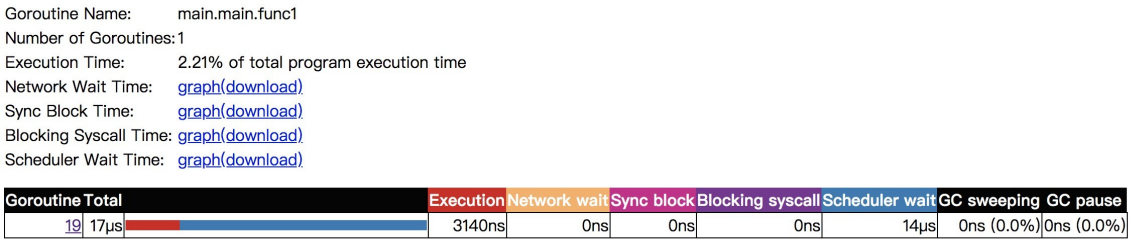


演示程序比较简单，因此这里就两块，一个是 `trace` 本身，另外一个是 `channel` 的收发。

Goroutine analysis

第二步看 “Goroutine analysis”，我们能通过这个功能看到整个运行过程中，每个函数块有多少个有 Goroutine 在跑，并且观察每个的 Goroutine 的运行开销都花费在哪个阶段。如下：

通过上图我们可以看到共有 3 个 goroutine，分别是 `runtime.main` 、 `runtime/trace.Start.func1` 、 `main.main.func1` ，那么它都做了些什么事呢，接下来我们可以通过点击具体细项去观察。如下：



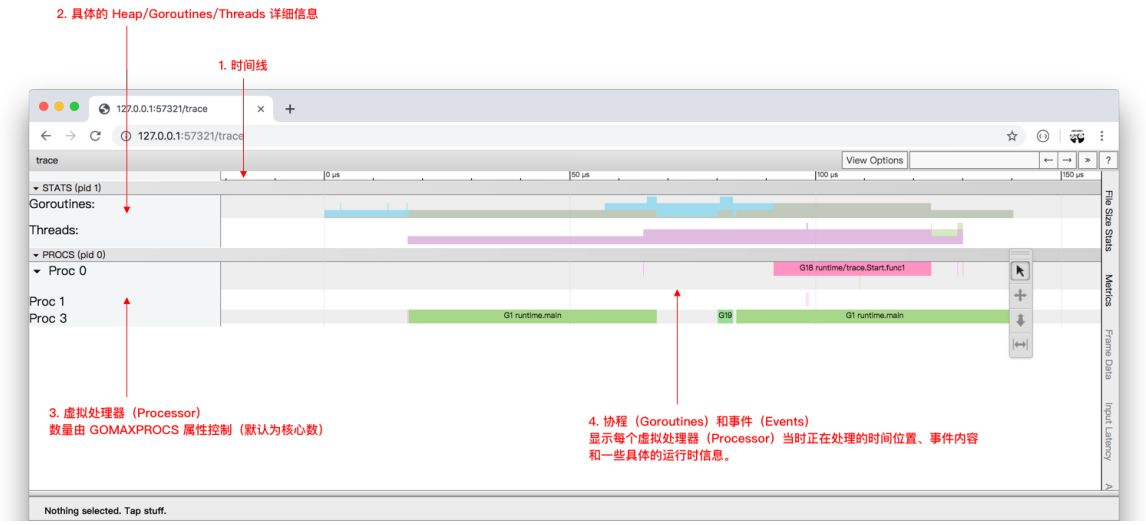
同时也可以看到当前 Goroutine 在整个调用耗时中的占比，以及 GC 清扫和 GC 暂停等待的一些开销。如果你觉得还不够，可以把图表下载下来分析，相当于把整个 Goroutine 运行时掰开来看了，这块能够很好的帮助我们**对 Goroutine 运行阶段做一个的剖析，可以得知到底慢哪，然后再决定下一步的排查方向**。如下：

名称	含义	耗时
Execution Time	执行时间	3140ns
Network Wait Time	网络等待时间	0ns
Sync Block Time	同步阻塞时间	0ns
Blocking Syscall Time	调用阻塞时间	0ns
Scheduler Wait Time	调度等待时间	14ns

名称	含义	耗时
GC Sweeping	GC 清扫	0ns
GC Pause	GC 暂停	0ns

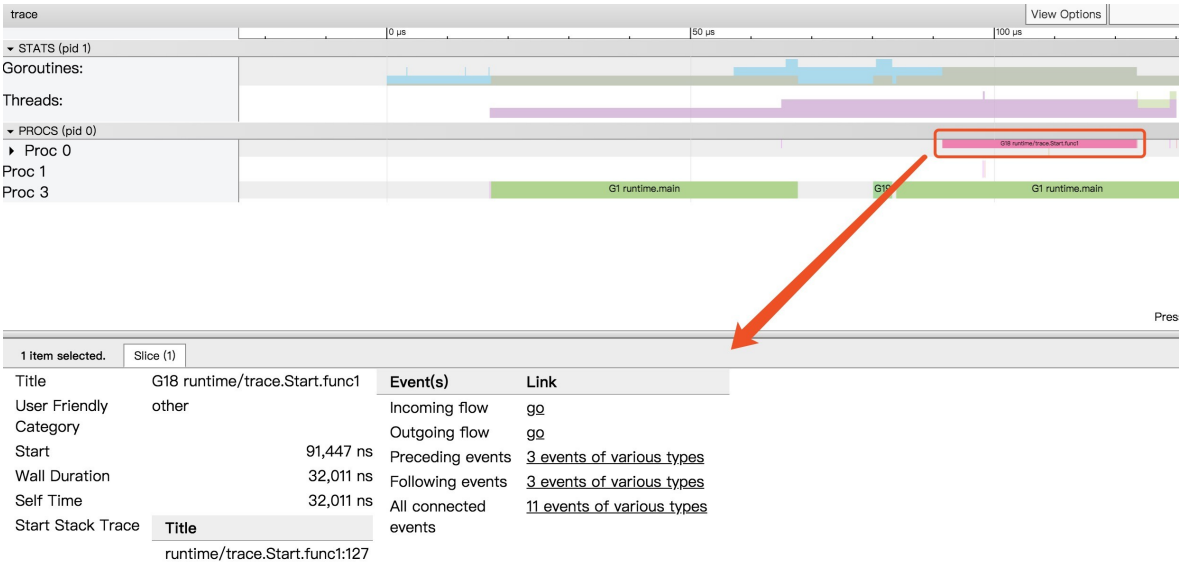
View trace

在对当前程序的 Goroutine 运行分布有了初步了解后，我们再通过“查看跟踪”看看之间的关联性，如下：



这个跟踪图粗略一看，相信有的小伙伴会比较懵逼，我们可以依据注解一块块查看，如下：

1. 时间线：显示执行的时间单元，根据时间维度的不同可以调整区间，具体可执行 `shift` + `?` 查看帮助手册。
2. 堆：显示执行期间的内存分配和释放情况。
3. 协程：显示在执行期间的每个 Goroutine 运行阶段有多少个协程在运行，其包含 GC 等待（GCWaiting）、可运行（Runnable）、运行中（Running）这三种状态。
4. OS 线程：显示在执行期间有多少个线程在运行，其包含正在调用 Syscall（InSyscall）、运行中（Running）这两种状态。
5. 虚拟处理器：每个虚拟处理器显示一行，虚拟处理器的数量一般默认为系统内核数。
6. 协程和事件：显示在每个虚拟处理器上有什么 Goroutine 正在运行，而连线行为代表事件关联。

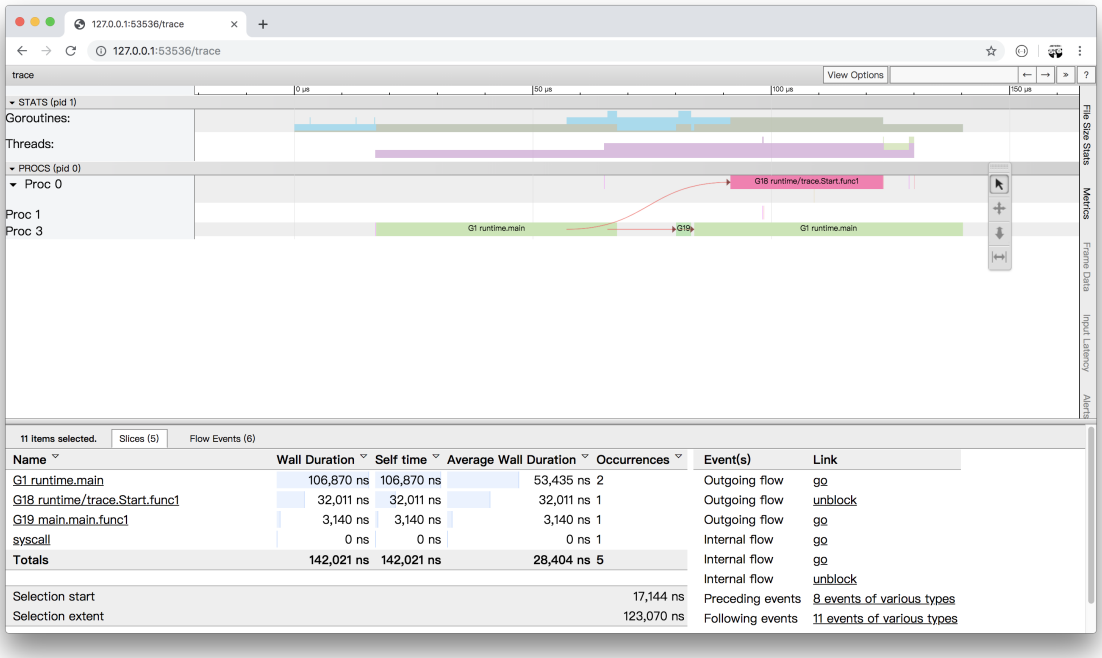


点击具体的 Goroutine 行为后可以看到其相关联的详细信息，这块很简单，大家实际操作一下就懂了。文字解释如下：

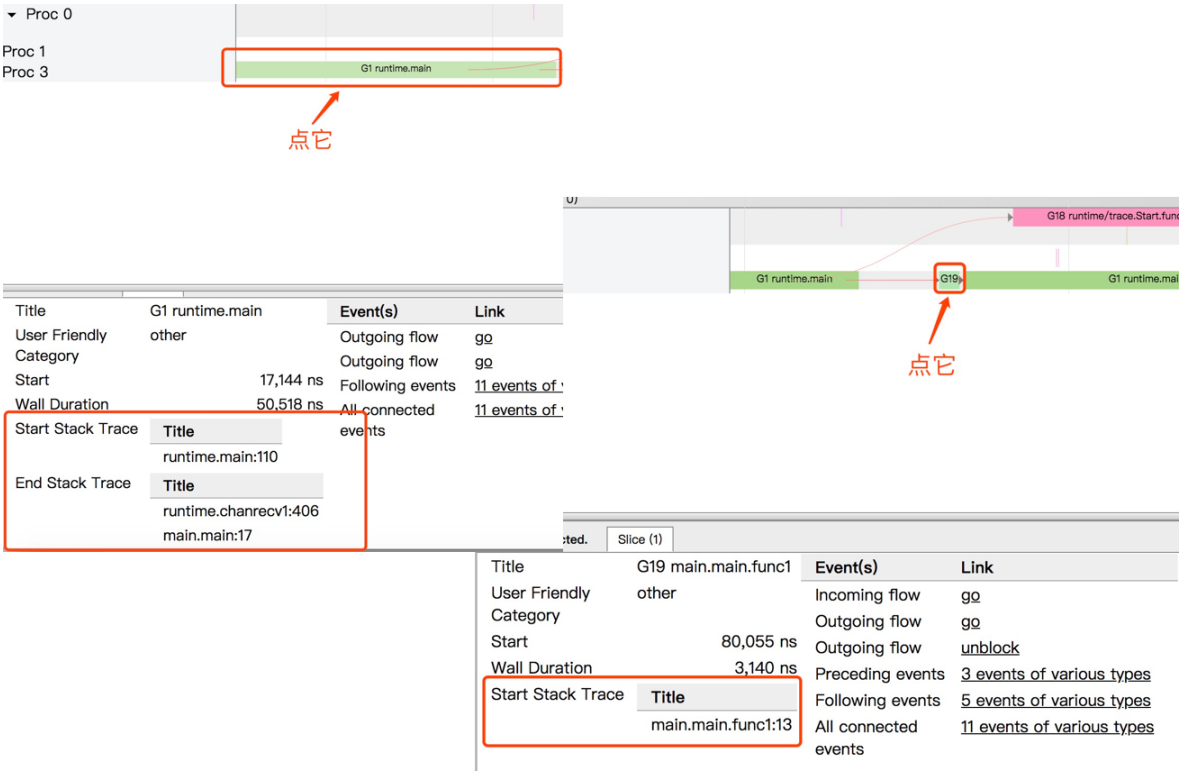
- Start：开始时间
- Wall Duration：持续时间
- Self Time：执行时间
- Start Stack Trace：开始时的堆栈信息
- End Stack Trace：结束时的堆栈信息
- Incoming flow：输入流
- Outgoing flow：输出流
- Preceding events：之前的事件
- Following events：之后的事件
- All connected：所有连接的事件

View Events

我们可以通过点击 View Options-Flow events、Following events 等方式，查看我们应用运行中的事件流情况。如下：



通过分析图上的事件流，我们可得知这程序从 `G1 runtime.main` 开始运行，在运行时创建了 2 个 Goroutine，先是创建 `G18 runtime/trace.Start.func1`，然后再是 `G19 main.main.func1`。而同时我们可以通过其 Goroutine Name 去了解它的调用类型，如：`runtime/trace.Start.func1` 就是程序中在 `main.main` 调用了 `runtime/trace.Start` 方法，然后该方法又利用协程创建了一个闭包 `func1` 去进行调用。



在这里我们结合开头的代码去看的话，很明显就是 `ch` 的输入输出的过程了。

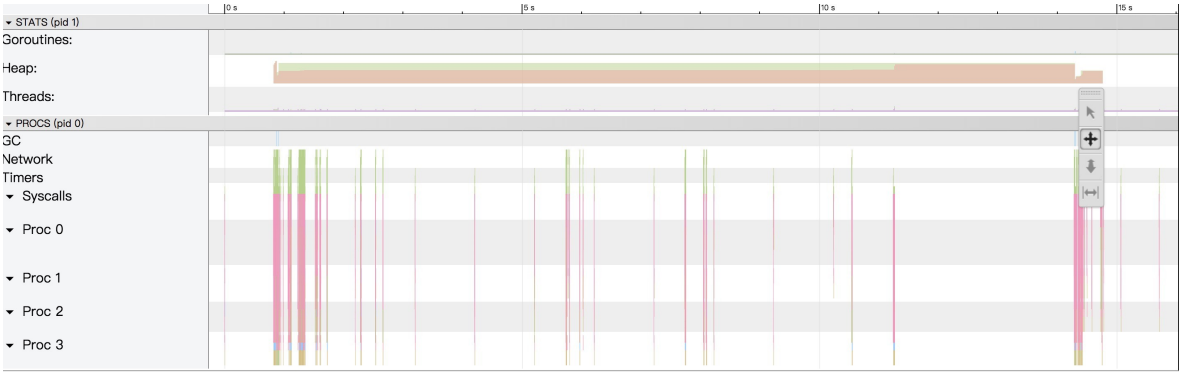
结合实战

今天生产环境突然出现了问题，机智的你早已埋好 `_ "net/http/pprof"` 这个神奇的工具，你麻利的执行了如下命令：

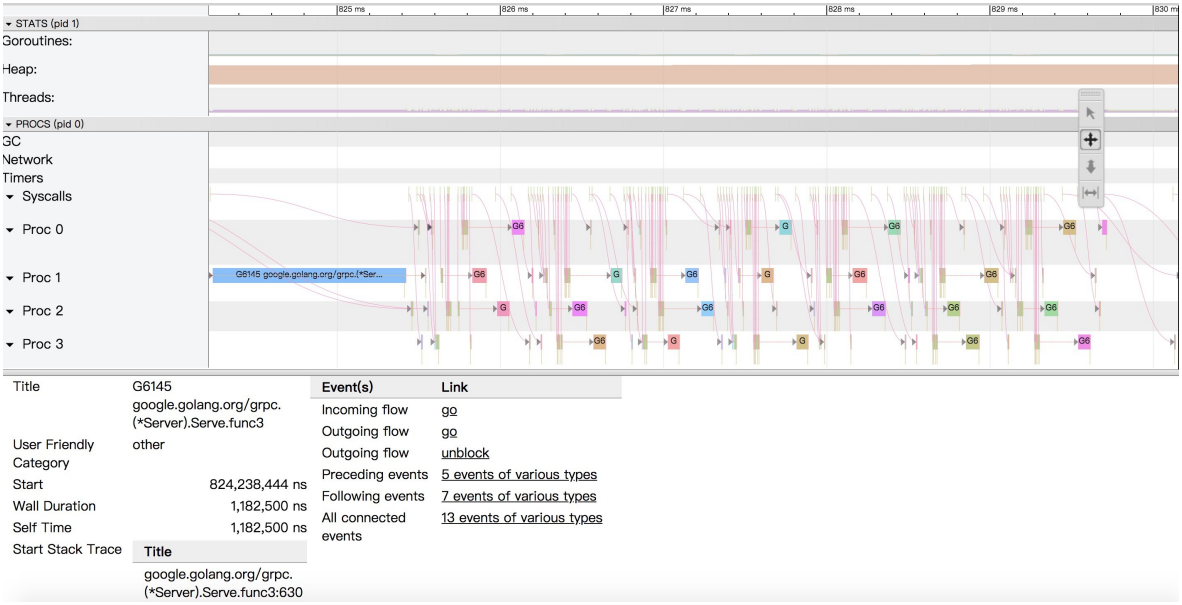
- curl http://127.0.0.1:6060/debug/pprof/trace\?seconds\=20 > trace.out
- go tool trace trace.out

View trace

你很快的看到了熟悉的 List 界面，然后不信邪点开了 View trace 界面，如下：



完全看懵的你，稳住，对着合适的区域执行快捷键 `W` 不断地放大时间线，如下：



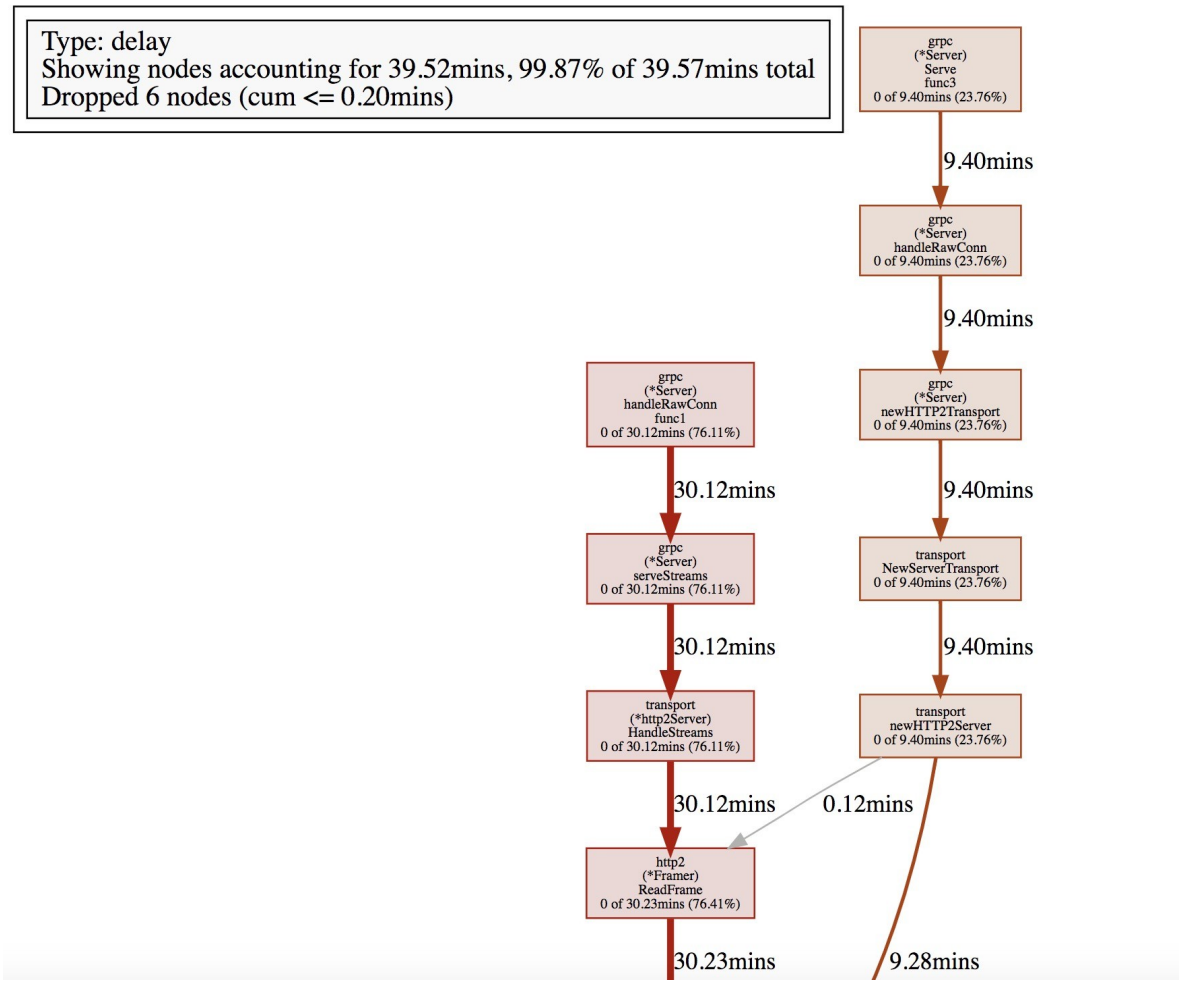
经过初步排查，你发现上述绝大部分的 G 竟然都和 `google.golang.org/grpc.(*Server).Serve.func` 有关，关联的一大串也是 `Serve` 所触发的相关动作。

30 items selected. Slices (12) Flow Events (18)

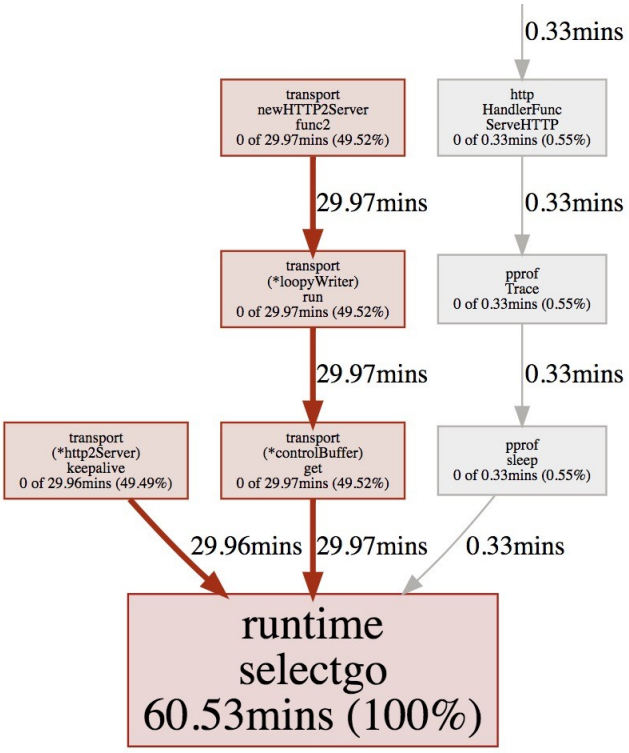
Name	Wall Duration	Self time	Average Wall Duration	Occurrences	Event(s)	Link
G6970 google.golang.org/grpc.(*Server).serveStreams.func1.1	64,501 ns	64,501 ns	32,250 ns	2	Outgoing flow	sysexit
G6979 net/http.(*Transport).getConn.func4	35,695 ns	35,695 ns	17,848 ns	2	Outgoing flow	go
G6980 internal/singleflight.(*Group).doCall	12,249 ns	12,249 ns	6,125 ns	2	Outgoing flow	go
G6981 net.cgolPLookup	18,389 ns	18,389 ns	18,389 ns	1	Outgoing flow	go
syscall	0 ns	0 ns	0 ns	2	Outgoing flow	go
G6682 google.golang.org/grpc.(*Server).handleRawConn.func1	93,361 ns	93,361 ns	93,361 ns	1	Outgoing flow	unblock
unblock	0 ns	0 ns	0 ns	1	Outgoing flow	unblock
G6680 google.golang.org/grpc/internal/transport.newHTTP2Server.func2	23,334 ns	23,334 ns	23,334 ns	1	Outgoing flow	unblock
Totals	247,529 ns	247,529 ns	20,627 ns	12	Outgoing flow	unblock

这时候有经验的你心里已经有了初步结论，你可以继续追踪 View trace 深入进去，不过我建议先鸟瞰全貌，因此我们再往下看 “Network blocking profile” 和 “Syscall blocking profile” 所提供的信息，如下：

Network blocking profile



Syscall blocking profile



通过对以上三项的跟踪分析，加上这个泄露，这个阻塞的耗时，这个涉及的内部方法名，很明显就是哪位又忘记关闭客户端连接了，赶紧改改改。

总结

通过本文我们习得了 `go tool trace` 的武林秘籍，它能够跟踪捕获各种执行中的事件，例如 Goroutine 的创建/阻塞/解除阻塞，Syscall 的进入/退出/阻止，GC 事件，Heap 的大小改变，Processor 启动/停止等等。

希望你能够用好 Go 的两大杀器 pprof + trace 组合，此乃排查好搭档，谁用谁清楚，即使他并不万能。

参考

- https://about.sourcegraph.com/go/an-introduction-to-go-tool-trace-rhys-hiltner
- https://www.itcodemonkey.com/article/5419.html
- https://making.pusher.com/go-tool-trace/
- https://golang.org/cmd/trace/
- https://docs.google.com/document/d/1FP5apqzBgr7ahCCgFO-yoVhk4YZrNIDNf9RybngBc14/pub
- https://godoc.org/runtime/trace

用 GODEBUG 看调度跟踪

让 Go 更强大的原因之一莫过于它的 GODEBUG 工具，GODEBUG 的设置可以让 Go 程序在运行时输出调试信息，可以根据你的要求很直观的看到你想要的调度器或垃圾回收等详细信息，并且还不需要加装其它的插件，非常方便，今天我们将先讲解 GODEBUG 的调度器相关内容，希望对你有所帮助。

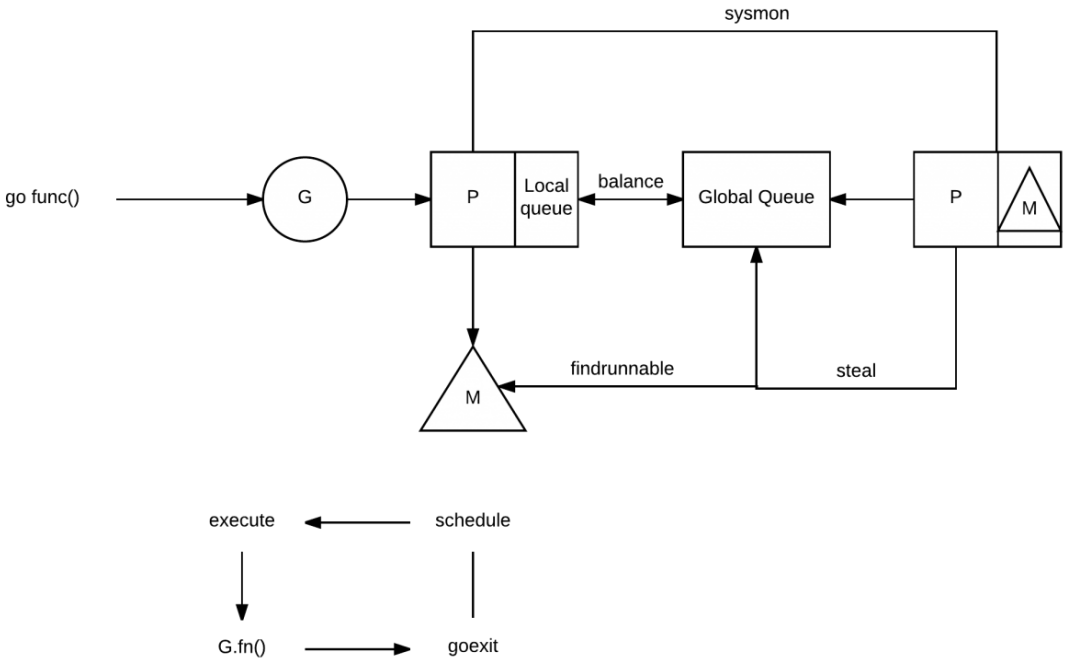
不过在开始前，没接触过的小伙伴得先补补如下前置知识，便于更好的了解调试器输出的信息内容。

前置知识

Go scheduler 的主要功能是针对在处理器上运行的 OS 线程分发可运行的 Goroutine，而我们一提到调度器，就离不开三个经常被提到的缩写，分别是：

- G：Goroutine，实际上我们每次调用 `go func` 就是生成了一个 G。
- P：处理器，一般为处理器的核数，可以通过 `GOMAXPROCS` 进行修改。
- M：OS 线程

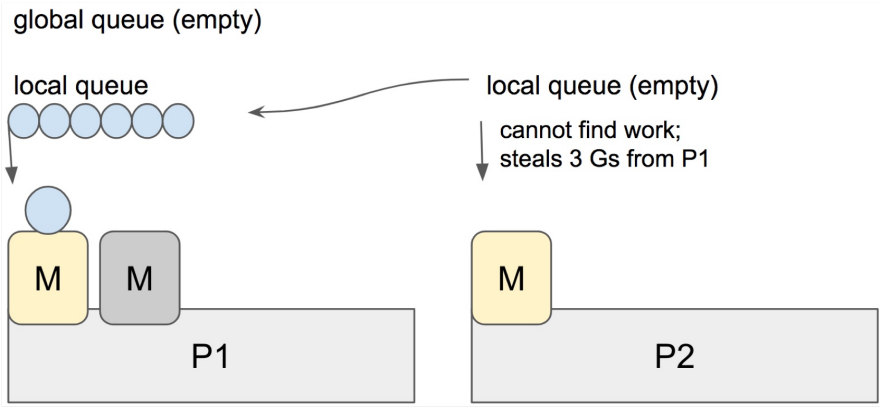
这三者交互实际来源于 Go 的 M: N 调度模型，也就是 M 必须与 P 进行绑定，然后不断地在 M 上循环寻找可运行的 G 来执行相应的任务，如果想具体了解可以详细阅读 《Go Runtime Scheduler》，我们抽其中的工作流程图进行简单分析，如下：



1. 当我们执行 `go func()` 时，实际上就是创建一个全新的 Goroutine，我们称它为 G。
2. 新创建的 G 会被放入 P 的本地队列（Local Queue）或全局队列（Global Queue）中，准备下一步的动作。
3. 唤醒或创建 M 以便执行 G。
4. 不断地进行事件循环
5. 寻找在可用状态下的 G 进行执行任务
6. 清除后，重新进入事件循环

而在描述中有提到全局和本地这两类队列，其实在功能上来讲都是用于存放正在等待运行的 G，但是不同点在于，本地队列有数量限制，不允许超过 256 个。并且在新建 G 时，会优先选择 P 的本地队列，如果本地队列满了，则将 P 的本地队列的一半的 G 移动到全局队列，这其实可以理解为调度资源的共享和再平衡。

另外我们可以看到图上有 steal 行为，这是用来做什么的呢，我们都知道当你创建新的 G 或者 G 变成可运行状态时，它会被推送加入到当前 P 的本地队列中。但其实当 P 执行 G 完毕后，它也会“干活”，它会将其从本地队列中弹出 G，同时会检查当前本地队列是否为空，如果为空会随机的从其他 P 的本地队列中尝试窃取一半可运行的 G 到自己的名下。例子如下：



在这个例子中，P2 在本地队列中找不到可以运行的 G，它会执行 `work-stealing` 调度算法，随机选择其它的处理器 P1，并从 P1 的本地队列中窃取了三个 G 到它自己的本地队列中去。至此，P1、P2 都拥有了可运行的 G，P1 多余的 G 也不会被浪费，调度资源将会更加平均的在多个处理器中流转。

GODEBUG

GODEBUG 变量可以控制运行时内的调试变量，参数以逗号分隔，格式为：`name=val`。本文着重点在调度器观察上，将会使用如下两个参数：

- schedtrace：设置 `schedtrace=X` 参数可以使运行时在每 X 毫秒发出一行调度器的摘要信息到标准 err 输出中。
- scheddetail：设置 `schedtrace=X` 和 `scheddetail=1` 可以使运行时在每 X 毫秒发出一次详细的多行信息，信息内容主要包括调度程序、处理器、OS 线程 和 Goroutine 的状态。

演示代码

```
func main() {
    wg := sync.WaitGroup{}
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func(wg *sync.WaitGroup) {
            var counter int
            for i := 0; i < 1e10; i++ {
                counter++
            }
            wg.Done()
        }(&wg)
    }

    wg.Wait()
}
```

schedtrace

```
$ GODEBUG=schedtrace=1000 ./awesomeProject
SCHED 0ms: gomaxprocs=4 idleprocs=1 threads=5 spinningthreads=1 idlethreads=0 ru
nqueue=0 [0 0 0 0]
SCHED 1000ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 2000ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 3001ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 4010ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 5011ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 6012ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 [1 2 2 1]
SCHED 7021ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
```

```
runqueue=4 [0 1 1 0]
SCHED 8023ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=4 [0 1 1 0]
SCHED 9031ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=4 [0 1 1 0]
SCHED 10033ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=
0 runqueue=4 [0 1 1 0]
SCHED 11038ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=
0 runqueue=4 [0 1 1 0]
SCHED 12044ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=
0 runqueue=4 [0 1 1 0]
SCHED 13051ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=
0 runqueue=4 [0 1 1 0]
SCHED 14052ms: gomaxprocs=4 idleprocs=2 threads=5
...
```

- sched：每一行都代表调度器的调试信息，后面提示的毫秒数表示启动到现在的运行时间，输出的时间间隔受 `schedtrace` 的值影响。
- gomaxprocs：当前的 CPU 核心数（GOMAXPROCS 的当前值）。
- idleprocs：空闲的处理器数量，后面的数字表示当前的空闲数量。
- threads：OS 线程数量，后面的数字表示当前正在运行的线程数量。
- spinningthreads：自旋状态的 OS 线程数量。
- idlethreads：空闲的线程数量。
- runqueue：全局队列中中的 Goroutine 数量，而后面的 [0 0 1 1] 则分别代表这 4 个 P 的本地队列正在运行的 Goroutine 数量。

在上面我们有提到“自旋线程”这个概念，如果你之前没有了解过相关概念，一听“自旋”肯定会比较懵，我们引用 《Head First of Golang Scheduler》 的内容来说明：

> 自旋线程的这个说法，是因为 Go Scheduler 的设计者在考虑了“OS 的资源利用率”以及“频繁的线程抢占给 OS 带来的负载”之后，提出了“Spinning Thread”的概念。也就是当“自旋线程”没有找到可供其调度执行的 Goroutine 时，并不会销毁该线程，而是采取“自旋”的操作保存了下来。虽然看起来这是浪费了一些资源，但是考虑一下 syscall 的情景就可以知道，比起“自旋”，线程间频繁的抢占以及频繁的创建和销毁操作可能带来的危害会更大。

scheddetail

如果我们想要更详细的看到调度器的完整信息时，我们可以增加 `scheddetail` 参数，就能够更进一步的查看调度的细节逻辑，如下：

```
$ GODEBUG=scheddetail=1,schedtrace=1000 ./awesomeProject
SCHED 1000ms: gomaxprocs=4 idleprocs=0 threads=5 spinningthreads=0 idlethreads=0
runqueue=0 gcwaiting=0 nmidlelocked=0 stopwait=0 sysmonwait=0
  P0: status=1 schedtick=2 syscalltick=0 m=3 runqsize=3 gfreecnt=0
  P1: status=1 schedtick=2 syscalltick=0 m=4 runqsize=1 gfreecnt=0
  P2: status=1 schedtick=2 syscalltick=0 m=0 runqsize=1 gfreecnt=0
  P3: status=1 schedtick=1 syscalltick=0 m=2 runqsize=1 gfreecnt=0
  M4: p=1 curg=18 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 spinning=fa
lse blocked=false lockedg=-1
  M3: p=0 curg=22 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 spinning=fa
lse blocked=false lockedg=-1
  M2: p=3 curg=24 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 spinning=fa
lse blocked=false lockedg=-1
  M1: p=-1 curg=-1 mallocing=0 throwing=0 preemptoff= locks=1 dying=0 spinning=f
alse blocked=false lockedg=-1
  M0: p=2 curg=26 mallocing=0 throwing=0 preemptoff= locks=0 dying=0 spinning=fa
lse blocked=false lockedg=-1
  G1: status=4(semacquire) m=-1 lockedm=-1
  G2: status=4(force gc (idle)) m=-1 lockedm=-1
  G3: status=4(GC sweep wait) m=-1 lockedm=-1
  G17: status=1() m=-1 lockedm=-1
  G18: status=2() m=4 lockedm=-1
  G19: status=1() m=-1 lockedm=-1
  G20: status=1() m=-1 lockedm=-1
  G21: status=1() m=-1 lockedm=-1
  G22: status=2() m=3 lockedm=-1
  G23: status=1() m=-1 lockedm=-1
  G24: status=2() m=2 lockedm=-1
  G25: status=1() m=-1 lockedm=-1
  G26: status=2() m=0 lockedm=-1
```

在这里我们抽取了 1000ms 时的调试信息来查看，信息量比较大，我们先从每一个字段开始了解。如下：

G

- status：G 的运行状态。
- m：隶属哪一个 M。
- lockedm：是否有锁定 M。

在第一点中我们有提到 G 的运行状态，这对于分析内部流转非常的有用，共涉及如下 9 种状态：

状态	值	含义
_Gidle	0	刚刚被分配，还没有进行初始化。
_Grunnable	1	已经在运行队列中，还没有执行用户代码。
_Grunning	2	不在运行队列里中，已经可以执行用户代码， 此时已经分配了 M 和 P。
_Gsyscall	3	正在执行系统调用，此时分配了 M。
_Gwaiting	4	在运行时被阻止，没有执行用户代码， 也不在运行队列中，此时它正在某处阻塞等待中。
*Gmoribund*unused	5	尚未使用，但是在 gdb 中进行了硬编码。
_Gdead	6	尚未使用，这个状态可能是刚退出或是刚被初始化， 此时它并没有执行用户代码， 有可能有也有可能没有分配堆栈。
*Genqueue*unused	7	尚未使用。
_Gcopystack	8	正在复制堆栈，并没有执行用户代码， 也不在运行队列中。

在理解了各类的状态的意思后，我们结合上述案例看看，如下：

```
G1: status=4(semacquire) m=-1 lockedm=-1
G2: status=4(force gc (idle)) m=-1 lockedm=-1
G3: status=4(GC sweep wait) m=-1 lockedm=-1
G17: status=1() m=-1 lockedm=-1
G18: status=2() m=4 lockedm=-1
```

在这个片段中，G1 的运行状态为 `_Gwaiting`，并没有分配 M 和锁定。这时候你可能好奇在片段中括号里的是什么东西呢，其实是因为该 `status=4` 是表示 `Goroutine` 在**运行时时被阻止**，而阻止它的事件就是 `semacquire` 事件，是因为 `semacquire` 会检查信号量的情况，在合适的时机就调用 `goparkunlock` 函数，把当前 `Goroutine` 放进等待队列，并把它设为 `_Gwaiting` 状态。

那么在实际运行中还有什么原因会导致这种现象呢，我们一起看看，如下：

```
    waitReasonZero                                  // ""
    waitReasonGCAssistMarking                       // "GC assist marking"
    waitReasonIOWait                                // "IO wait"
```

```
    waitReasonChanReceiveNilChan                // "chan receive (nil cha
n)"
    waitReasonChanSendNilChan                   // "chan send (nil chan)"
    waitReasonDumpingHeap                       // "dumping heap"
    waitReasonGarbageCollection                 // "garbage collection"
    waitReasonGarbageCollectionScan             // "garbage collection sca
n"
    waitReasonPanicWait                         // "panicwait"
    waitReasonSelect                            // "select"
    waitReasonSelectNoCases                     // "select (no cases)"
    waitReasonGCAssistWait                      // "GC assist wait"
    waitReasonGCSweepWait                       // "GC sweep wait"
    waitReasonChanReceive                       // "chan receive"
    waitReasonChanSend                          // "chan send"
    waitReasonFinalizerWait                     // "finalizer wait"
    waitReasonForceGGIdle                       // "force gc (idle)"
    waitReasonSemacquire                        // "semacquire"
    waitReasonSleep                             // "sleep"
    waitReasonSyncCondWait                      // "sync.Cond.Wait"
    waitReasonTimerGoroutineIdle                // "timer goroutine (idl
e)"
    waitReasonTraceReaderBlocked                // "trace reader (blocke
d)"
    waitReasonWaitForGCCycle                    // "wait for GC cycle"
    waitReasonGCWorkerIdle                      // "GC worker (idle)"
```

我们通过以上 `waitReason` 可以了解到 `Goroutine` 会被暂停运行的原因要素，也就是会出现在括号中的事件。

M

- p：隶属哪一个 P。
- curg：当前正在使用哪个 G。
- runqsize：运行队列中的 G 数量。
- gfreecnt：可用的G（状态为 Gdead）。
- mallocing：是否正在分配内存。
- throwing：是否抛出异常。
- preemptoff：不等于空字符串的话，保持 curg 在这个 m 上运行。

P

- status：P 的运行状态。

- schedtick：P 的调度次数。
- syscalltick：P 的系统调用次数。
- m：隶属哪一个 M。
- runqsize：运行队列中的 G 数量。
- gfreecnt：可用的G（状态为 Gdead）。

状态	值	含义
_Pidle	0	刚刚被分配，还没有进行进行初始化。
_Prunning	1	当 M 与 P 绑定调用 acquirep 时，P 的状态会改变为 _Prunning。
_Psyscall	2	正在执行系统调用。
_Pgcstop	3	暂停运行，此时系统正在进行 GC，直至 GC 结束后才会转变到下一个状态阶段。
_Pdead	4	废弃，不再使用。

总结

通过本文我们学习到了调度的一些基础知识，再通过神奇的 GODEBUG 掌握了观察调度器的方式方法，你想想，是不是可以和我上一篇文章的 `go tool trace` 来结合使用呢，在实际的使用中，类似的办法有很多，组合巧用是重点。

参考

- Debugging performance issues in Go programs
- A whirlwind tour of Go’s runtime environment variables
- Go调度器系列（2）宏观看调度器
- Go’s work-stealing scheduler
- Scheduler Tracing In Go
- Head First of Golang Scheduler
- goroutine 的状态切换
- Environment_Variables

用 GODEBUG 看 GC

什么是 GC

在计算机科学中，垃圾回收（GC）是一种自动管理内存的机制，垃圾回收器会去尝试回收程序不再使用的对象及其占用的内存。而最早 John McCarthy 在 1959 年左右发明了垃圾回收，以简化 Lisp 中的手动内存管理的机制（来自 wikipedia）。

为什么要 GC

手动管理内存挺麻烦，管错或者管漏内存也很糟糕，将会直接导致程序不稳定（持续泄露）甚至直接崩溃。

GC 带来的问题

硬要说会带来什么问题的话，也就数大家最关注的 Stop The World（STW），STW 代指在执行某个垃圾回收算法的某个阶段时，需要将整个应用程序暂停去处理 GC 相关的工作事项。例如：

行为	会不会 STW	为什么
标记开始	会	在开始标记时，准备根对象的扫描，会打开写屏障（Write Barrier） 和 辅助GC（mutator assist）， 而回收器和应用程序是并发运行的， 因此会暂停当前正在运行的所有 Goroutine。
并发标记中	不会	标记阶段，主要目的是标记堆内存中仍在使用的值。
标记结束	会	在完成标记任务后，将重新扫描部分根对象， 这时候会禁用写屏障（Write Barrier）和辅助GC（mutator assist），而标记阶段和应用程序是并发运行的， 所以在标记阶段可能会有新的对象产生， 因此在重新扫描时需要进行 STW。

如何调整 GC 频率

可以通过 GOGC 变量设置初始垃圾收集器的目标百分比值，对比的规则为当新分配的数值与上一次收集后剩余的实时数值的比例达到设置的目标百分比时，就会触发 GC，默认值为

GOGC=100。如果将其设置为 GOGC=off 可以完全禁用垃圾回收器，要不试试？

简单来讲就是，GOGC 的值设置的越大，GC 的频率越低，但每次最终所触发到 GC 的堆内存也会更大。

各版本 GC 情况

版本	GC 算法	STW 时间
Go 1.0	STW（强依赖 tcmalloc）	百ms到秒级别
Go 1.3	Mark STW, Sweep 并行	百ms级别
Go 1.5	三色标记法, 并发标记清除。同时运行时从 C 和少量汇编，改为 Go 和少量汇编实现	10-50ms级别
Go 1.6	1.5 中一些与并发 GC 不协调的地方更改，集中式的 GC 协调协程，改为状态机实现	5ms级别
Go 1.7	GC 时由 mark 栈收缩改为并发，span 对象分配机制由 freelist 改为 bitmap 模式，SSA引入	ms级别
Go 1.8	混合写屏障（hybrid write barrier）, 消除 re-scanning stack	sub ms
Go 1.12	Mark Termination 流程优化	sub ms, 但几乎减少一半

注：资料来源于 @boya 在深圳 Gopher Meetup 的分享。

GODEBUG

GODEBUG 变量可以控制运行时内的调试变量，参数以逗号分隔，格式为：`name=val`。本文着重点在 GC 的观察上，主要涉及 gctrace 参数，我们通过设置 `gctrace=1` 后就可以使得垃圾收集器向标准错误流发出 GC 运行信息。

涉及术语

- mark：标记阶段。
- markTermination：标记结束阶段。

- mutator assist：辅助 GC，是指在 GC 过程中 mutator 线程会并发运行，而 mutator assist 机制会协助 GC 做一部分的工作。
- heap_live：在 Go 的内存管理中，span 是内存页的基本单元，每页大小为 8kb，同时 Go 会根据对象的大小不同而分配不同页数的 span，而 heap_live 就代表着所有 span 的总大小。
- dedicated / fractional / idle：在标记阶段会分为三种不同的 mark worker 模式，分别是 dedicated、fractional 和 idle，它们代表着不同的专注程度，其中 dedicated 模式最专注，是完整的 GC 回收行为，fractional 只会干部分的 GC 行为，idle 最轻松。这里你只需要了解它是不同专注程度的 mark worker 就好了，详细介绍我们可以等后续的文章。

演示代码

```
func main() {
    wg := sync.WaitGroup{}
    wg.Add(10)
    for i := 0; i < 10; i++ {
        go func(wg *sync.WaitGroup) {
            var counter int
            for i := 0; i < 1e10; i++ {
                counter++
            }
            wg.Done()
        }(&wg)
    }

    wg.Wait()
}
```

gctrace

```
$ GODEBUG=gctrace=1 go run main.go
gc 1 @0.032s 0%: 0.019+0.45+0.003 ms clock, 0.076+0.22/0.40/0.80+0.012 ms cpu, 4
->4->0 MB, 5 MB goal, 4 P
gc 2 @0.046s 0%: 0.004+0.40+0.008 ms clock, 0.017+0.32/0.25/0.81+0.034 ms cpu, 4
->4->0 MB, 5 MB goal, 4 P
gc 3 @0.063s 0%: 0.004+0.40+0.008 ms clock, 0.018+0.056/0.32/0.64+0.033 ms cpu,
4->4->0 MB, 5 MB goal, 4 P
gc 4 @0.080s 0%: 0.004+0.45+0.016 ms clock, 0.018+0.15/0.34/0.77+0.065 ms cpu, 4
```

```
->4->1 MB, 5 MB goal, 4 P
gc 5 @0.095s 0%: 0.015+0.87+0.005 ms clock, 0.061+0.27/0.74/1.8+0.023 ms cpu, 4->4->1 MB, 5 MB goal, 4 P
gc 6 @0.113s 0%: 0.014+0.69+0.002 ms clock, 0.056+0.23/0.48/1.4+0.011 ms cpu, 4->4->1 MB, 5 MB goal, 4 P
gc 7 @0.140s 1%: 0.031+2.0+0.042 ms clock, 0.12+0.43/1.8/0.049+0.17 ms cpu, 4->4->1 MB, 5 MB goal, 4 P
...
```

格式

```
gc # @#s #%: #+#+# ms clock, #+#/#/#+# ms cpu, #->#->#  MB, # MB goal, # P
```

含义

- `gc#`：GC 执行次数的编号，每次叠加。
- `@#s`：自程序启动后到当前的具体秒数。
- `#%`：自程序启动以来在GC中花费的时间百分比。
- `#+...+#`：GC 的标记工作共使用的 CPU 时间占总 CPU 时间的百分比。
- `#->#->#  MB`：分别表示 GC 启动时, GC 结束时, GC 活动时的堆大小.
- `#MB goal`：下一次触发 GC 的内存占用阈值。
- `#P`：当前使用的处理器 P 的数量。

案例

```
gc 7 @0.140s 1%: 0.031+2.0+0.042 ms clock, 0.12+0.43/1.8/0.049+0.17 ms cpu, 4->4->1 MB, 5 MB goal, 4 P
```

- gc 7：第 7 次 GC。
- @0.140s：当前是程序启动后的 0.140s。
- 1%：程序启动后到现在共花费 1% 的时间在 GC 上。
- 0.031+2.0+0.042 ms clock：
  - 0.031：表示单个 P 在 mark 阶段的 STW 时间。
  - 2.0：表示所有 P 的 mark concurrent（并发标记）所使用的时间。
  - 0.042：表示单个 P 的 markTermination 阶段的 STW 时间。
- 0.12+0.43/1.8/0.049+0.17 ms cpu：
  - 0.12：表示整个进程在 mark 阶段 STW 停顿的时间。

    - 0.43/1.8/0.049：0.43 表示 mutator assist 占用的时间，1.8 表示 dedicated + fractional 占用的时间，0.049 表示 idle 占用的时间。
    - 0.17ms：0.17 表示整个进程在 markTermination 阶段 STW 时间。
- 4->4->1 MB：
    - 4：表示开始 mark 阶段前的 heap_live 大小。
    - 4：表示开始 markTermination 阶段前的 heap_live 大小。
    - 1：表示被标记对象的大小。
- 5 MB goal：表示下一次触发 GC 回收的阈值是 5 MB。
- 4 P：本次 GC 一共涉及多少个 P。

总结

通过本章节我们掌握了使用 GODEBUG 查看应用程序 GC 运行情况的方法，只要用这种方法我们就可以观测不同情况下 GC 的情况了，甚至可以做出非常直观的对比图，大家不妨尝试一下。

关联文章

- 用 GODEBUG 看调度跟踪

参考

- Go GC打印出来的这些信息都是什么含义？
- GODEBUG之gctrace解析
- 关于Golang GC的一些误解–真的比Java GC更领先吗？
- @boya 深入浅出Golang Runtime PPT

talk

聊一聊，Go 的相对路径问题

Go 的 fake-useragent 了解一下

用 Go 来了解一下 Redis 通讯协议

使用 Gomock 进行单元测试

在 Go 中恰到好处的内存对齐

来，控制一下 goroutine 的并发数量

for-loop 与 json.Unmarshal 性能分析概要

简单围观一下有趣的 //go: 指令

我要在栈上。不，你应该在堆上

Go1.12 defer 会有性能损耗，尽量不要用？

从实践到原理，带你参透 gRPC

Go1.13 defer 的性能是如何提高的

Go 应用内存占用太多，让排查？（VSZ篇）

聊一聊，Go 的相对路径问题

前言

`Golang` 中存在各种运行方式，如何**正确的引用文件路径**成为一个值得商议的问题

以 gin-blog 为例，当我们在项目根目录下，执行 `go run main.go` 时能够正常运行（ `go build` 也是正常的）

```
[$ gin-blog]# go run main.go
[GIN-debug] [WARNING] Running in "debug" mode. Switch to "release" mode in produ
ction.
 - using env:    export GIN_MODE=release
 - using code:    gin.SetMode(gin.ReleaseMode)

[GIN-debug] GET    /api/v1/tags              --> gin-blog/routers/api/v1.GetTags
(3 handlers)
...
```

那么在不同的目录层级下，不同的方式运行，又是怎么样的呢，带着我们的疑问去学习

问题

1、 go run
我们上移目录层级，到 `$GOPATH/src` 下，执行 `go run gin-blog/main.go`

```
[$ src]# go run gin-blog/main.go
2018/03/12 16:06:13 Fail to parse 'conf/app.ini': open conf/app.ini: no such fil
e or directory
exit status 1
```

2、 go build，执行 `./gin-blog/main`

```
[$ src]# ./gin-blog/main
2018/03/12 16:49:35 Fail to parse 'conf/app.ini': open conf/app.ini: no such fil
e or directory
```

这时候你要打一个大大的问号，就是我的程序读取到什么地方去了

我们通过分析得知，`Golang` **的相对路径是相对于执行命令时的目录**；自然也就读取不到了

思考

既然已经知道问题的所在点，我们就可以寻思做点什么：)

我们想到相对路径是相对执行命令的目录，那么我们获取可执行文件的地址，拼接起来不就好了吗？

实践

我们编写**获取当前可执行文件路径的方法**

```
import (
    "path/filepath"
    "os"
    "os/exec"
    "string"
)

func GetAppPath() string {
    file, _ := exec.LookPath(os.Args[0])
    path, _ := filepath.Abs(file)
    index := strings.LastIndex(path, string(os.PathSeparator))

    return path[:index]
}
```

将其放到启动代码处查看路径

```
log.Println(GetAppPath())
```

我们分别执行以下两个命令，查看输出结果

1、 go run

```
$ go run main.go
2018/03/12 18:45:40 /tmp/go-build962610262/b001/exe
```

2、 go build

```
$ ./main
2018/03/12 18:49:44 $GOPATH/src/gin-blog
```

剖析

我们聚焦在 `go run` 的输出结果上，发现它是一个临时文件的地址，这是为什么呢？

在 `go help run` 中，我们可以看到

```
Run compiles and runs the main package comprising the named Go source files.
A Go source file is defined to be a file ending in a literal ".go" suffix.
```

也就是 `go run` 执行时会将文件放到 `/tmp/go-build...` 目录下，编译并运行

因此 `go run main.go` 出现 `/tmp/go-build962610262/b001/exe` 结果也不奇怪了，因为它已经跑到临时目录下去执行可执行文件了

这就已经很清楚了，那么我们想想，会出现哪些问题呢

- 依赖相对路径的文件，出现路径出错的问题
- `go run` 和 `go build` 不一样，一个到临时目录下执行，一个可手动在编译后的目录下执行，路径的处理方式会不同
- 不断`go run`，不断产生新的临时文件

这其实就是**根本原因**了，因为 `go run` 和 `go build` 的编译文件执行路径并不同，执行的层级也有可能不一样，自然而然就出现各种读取不到的奇怪问题了

解决方案

一、获取编译后的可执行文件路径

1、将配置文件的相对路径与 `GetAppPath()` 的结果相拼接，可解决 `go build main.go` 的可执行文件跨目录执行的问题（如：`./src/gin-blog/main`）

```
import (
    "path/filepath"
    "os"
    "os/exec"
    "string"
)

func GetAppPath() string {
    file, _ := exec.LookPath(os.Args[0])
    path, _ := filepath.Abs(file)
    index := strings.LastIndex(path, string(os.PathSeparator))
```

```
    return path[:index]
}
```

但是这种方式，对于 `go run` 依旧无效，这时候就需要2来补救

2、 通过传递参数指定路径，可解决 `go run` 的问题

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    var appPath string
    flag.StringVar(&appPath, "app-path", "app-path")
    flag.Parse()
    fmt.Printf("App path: %s", appPath)
}
```

运行

```
go run main.go --app-path "Your project address"
```

二、增加 `os.Getwd()` 进行多层判断

参见 beego 读取 `app.conf` 的代码

该写法可兼容 `go build` 和在项目根目录执行 `go run` ，但是若跨目录执行 `go run` 就不行

三、配置全局系统变量

我们可以通过 `os.Getenv` 来获取系统全局变量，然后与相对路径进行拼接

1、 设置项目工作区

简单来说，就是设置项目（应用）的工作路径，然后与配置文件、日志文件等相对路径进行拼接，达到相对的绝对路径来保证路径一致

参见 gogs 读取 `GOGS_WORK_DIR` 进行拼接的代码

2、 利用系统自带变量

简单来说就是通过系统自带的全局变量，例如 `$HOME` 等，将配置文件存放在 `$HOME/conf` 或 `/etc/conf` 下

这样子就能更加固定的存放配置文件，**不需要额外去设置一个环境变量**

（这点今早与一位SFer讨论了一波，感谢）

拓展

`go test` 在一些场景下也会遇到路径问题，因为 `go test` 只能够在当前目录执行，所以在执行测试用例的时候，你的执行目录已经是测试目录了

需要注意的是，如果采用获取外部参数的办法，用 `os.args` 时， `go test -args` 和 `go run` 、 `go build` 会有命令行参数位置的不一致问题

小结

这三种解决方案，在目前可见的开源项目或介绍中都能找到这些的身影

优缺点也是显而易见的，我认为应在**不同项目选定合适的解决方案**即可

建议大家不要强依赖读取配置文件的模块，应当将其“堆积木”化，**需要什么配置才去注册什么配置变量**，可以解决一部分的问题

大家又有什么想法呢，一起讨论一波？

Go 的 fake-useragent 了解一下

有的网站会根据 User-Agent 的不同，跳转到不同（PC、M）的站点，也有根据版本的不同给出不一样的提示等等，而 User-Agent 的变化更是爬虫里的基础姿势

使用 Go 编写网络爬虫或需要模拟浏览器头（User-Agent）的时候，你是否会觉得很麻烦，获取请求头（Request Headers）的 User-Agent 还得找来找去，挺繁琐。先前我也遇到了这个问题，因此有了这个项目 fake-useragent，用来解决你我的痛点

项目地址：https://github.com/EDDYCJY/fake-useragent

支持

- All User-Agent Random
- Chrome
- InternetExplorer (IE)
- Firefox
- Safari
- Android
- MacOSX
- IOS
- Linux
- IPhone
- IPad
- Computer
- Mobile

安装

```
$ go get github.com/EDDYCJY/fake-useragent
```

用法

```
package main

import (
```

```
    "log"

    "github.com/EDDYCJY/fake-useragent"
)

func main() {
    // 推荐使用
    random := browser.Random()
    log.Printf("Random: %s", random)

    chrome := browser.Chrome()
    log.Printf("Chrome: %s", chrome)

    internetExplorer := browser.InternetExplorer()
    log.Printf("IE: %s", internetExplorer)

    firefox := browser.Firefox()
    log.Printf("Firefox: %s", firefox)

    safari := browser.Safari()
    log.Printf("Safari: %s", safari)

    android := browser.Android()
    log.Printf("Android: %s", android)

    macOSX := browser.MacOSX()
    log.Printf("MacOSX: %s", macOSX)

    ios := browser.IOS()
    log.Printf("IOS: %s", ios)

    linux := browser.Linux()
    log.Printf("Linux: %s", linux)

    iphone := browser.IPhone()
    log.Printf("IPhone: %s", iphone)

    ipad := browser.IPad()
    log.Printf("IPad: %s", ipad)

    computer := browser.Computer()
    log.Printf("Computer: %s", computer)

    mobile := browser.Mobile()
    log.Printf("Mobile: %s", mobile)
}
```

定制

你可以调整抓取数据源的最大页数、时间间隔以及最大超时时间。 如果不填写，则为默认值。

```
client := browser.Client{
    MaxPage: 3,
    Delay: 200 * time.Millisecond,
    Timeout: 10 * time.Second,
}
cache := browser.Cache{}
b := browser.NewBrowser(client, cache)

random := b.Random()
```

更新浏览器头的临时文件缓存

```
client := browser.Client{}
cache := browser.Cache{
    UpdateFile: true,
}
b := browser.NewBrowser(client, cache)
```

最后，建议常规用法就好，默认参数能够满足日常需求

输出

```
Random: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_13_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.132 Safari/537.36

Chrome: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/60.0.3112.113 Safari/537.36

IE: Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; WOW64; Trident/5.0)

Firefox: Mozilla/5.0 (Windows NT 6.3; WOW64; rv:41.0) Gecko/20100101 Firefox/41.0

Safari: Mozilla/5.0 (iPhone; CPU iPhone OS 11_2_5 like Mac OS X) AppleWebKit/604.5.6 (KHTML, like Gecko) Version/11.0 Mobile/15D60 Safari/604.1

Android: Mozilla/5.0 (Linux; Android 6.0; MYA-L22 Build/HUAWEIMYA-L22) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/62.0.3202.84 Mobile Safari/537.36
```

```
MacOSX: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/602.2.14 (KHTML, like Gecko) Version/10.0.1 Safari/602.2.14

IOS: Mozilla/5.0 (iPhone; CPU iPhone OS 10_1 like Mac OS X) AppleWebKit/602.2.14 (KHTML, like Gecko) Version/10.0 Mobile/14B72 Safari/602.1

Linux: Mozilla/5.0 (X11; Linux x86_64; rv:42.0) Gecko/20100101 Firefox/42.0

IPhone: Mozilla/5.0 (iPhone; CPU iPhone OS 10_2 like Mac OS X) AppleWebKit/602.3.12 (KHTML, like Gecko) Version/10.0 Mobile/14C92 Safari/602.1

IPad: Mozilla/5.0 (iPad; CPU OS 5_0_1 like Mac OS X) AppleWebKit/534.46 (KHTML, like Gecko) Version/5.1 Mobile/9A405 Safari/7534.48.3

Computer: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:54.0) Gecko/20100101 Firefox/54.0

Mobile: Mozilla/5.0 (Linux; Android 7.0; Redmi Note 4 Build/NRD90M) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.111 Mobile Safari/537.36
```

注意

如果第一次使用，fake-useragent 将收集数据并在临时目录中创建一个文件作为文件缓存，请耐心等待几秒钟

最后

如果在项目中发现了什么问题，欢迎提交 PR 或者 issue。希望你能够喜欢这个项目，根本目的还是为了解决痛点，欢迎 Star！😁

项目地址：https://github.com/EDDYCJY/fake-useragent

用 Go 来了解一下 Redis 通讯协议

Go、PHP、Java... 都有那么多包来支撑你使用 Redis，那你是否有想过

有了服务端，有了客户端，他们俩是怎样通讯，又是基于什么通讯协议做出交互的呢？

介绍

基于我们的目的，本文主要讲解和实践 Redis 的通讯协议

Redis 的客户端和服务端是通过 TCP 连接来进行数据交互， 服务器默认的端口号为 6379

客户端和服务器发送的命令或数据一律以 \r\n （CRLF）结尾（这是一条约定）

协议

在 Redis 中分为**请求**和**回复**，而请求协议又分为新版和旧版，新版统一请求协议在 Redis 1.2 版本中引入，最终在 Redis 2.0 版本成为 Redis 服务器通信的标准方式

本文是基于新版协议来实现功能，不建议使用旧版（1.2 挺老旧了）。如下是新协议的各种范例：

请求协议

1、 格式示例

```
*<参数数量> CR LF
$<参数 1 的字节数量> CR LF
<参数 1 的数据> CR LF
...
$<参数 N 的字节数量> CR LF
<参数 N 的数据> CR LF
```

在该协议下所有发送至 Redis 服务器的参数都是二进制安全（binary safe）的

2、打印示例

```
*3
$3
SET
$5
mykey
```

```
$7
myvalue
```

3、实际协议值

```
"*3\r\n$3\r\nSET\r\n$5\r\nmykey\r\n$7\r\nmyvalue\r\n"
```

这就是 Redis 的请求协议规范，按照范例1编写客户端逻辑，最终发送的是范例3，相信你已经有大致的概念了，Redis 的协议非常的简洁易懂，这也是好上手的原因之一，你可以想想协议这么定义的好处在哪？

回复

Redis 会根据你请求协议的不同（执行的命令结果也不同），返回多种不同类型的回复。在这个回复“协议”中，可以通过检查第一个字节，确定这个回复是什么类型，如下：

- 状态回复（status reply）的第一个字节是 “+”
- 错误回复（error reply）的第一个字节是 “-“
- 整数回复（integer reply）的第一个字节是 “:”
- 批量回复（bulk reply）的第一个字节是 “$”
- 多条批量回复（multi bulk reply）的第一个字节是 “*”

有了回复的头部标识，结尾的 CRLF，你可以大致猜想出回复“协议”是怎么样的，但是实践才能得出真理，斋知道怕是你很快就忘记了

实践

与 Redis 服务器交互

```
package main

import (
    "log"
    "net"
    "os"

    "github.com/EDDYCJY/redis-protocol-example/protocol"
)

const (
    Address = "127.0.0.1:6379"
```

```
    Network = "tcp"
)

func Conn(network, address string) (net.Conn, error) {
    conn, err := net.Dial(network, address)
    if err != nil {
        return nil, err
    }

    return conn, nil
}

func main() {
        // 读取入参
    args := os.Args[1:]
    if len(args) <= 0 {
        log.Fatalf("Os.Args <= 0")
    }

        // 获取请求协议
    reqCommand := protocol.GetRequest(args)

    // 连接 Redis 服务器
    redisConn, err := Conn(Network, Address)
    if err != nil {
        log.Fatalf("Conn err: %v", err)
    }
    defer redisConn.Close()

        // 写入请求内容
    _, err = redisConn.Write(reqCommand)
    if err != nil {
        log.Fatalf("Conn Write err: %v", err)
    }

        // 读取回复
    command := make([]byte, 1024)
    n, err := redisConn.Read(command)
    if err != nil {
        log.Fatalf("Conn Read err: %v", err)
    }

        // 处理回复
    reply, err := protocol.GetReply(command[:n])
    if err != nil {
        log.Fatalf("protocol.GetReply err: %v", err)
```

```
    }

        // 处理后的回复内容
    log.Printf("Reply: %v", reply)
    // 原始的回复内容
    log.Printf("Command: %v", string(command[:n]))
}
```

在这里我们完成了整个 Redis 客户端和服务端交互的流程，分别如下：

1、读取命令行参数：获取执行的 Redis 命令

2、获取请求协议参数

3、连接 Redis 服务器，获取连接句柄

4、将请求协议参数写入连接：发送请求的命令行参数

5、从连接中读取返回的数据：读取先前请求的回复数据

6、根据回复“协议”内容，处理回复的数据集

7、输出处理后的回复内容及原始回复内容

请求

```
func GetRequest(args []string) []byte {
    req := []string{
        "*" + strconv.Itoa(len(args)),
    }

    for _, arg := range args {
        req = append(req, "$"+strconv.Itoa(len(arg)))
        req = append(req, arg)
    }

    str := strings.Join(req, "\r\n")
    return []byte(str + "\r\n")
}
```

通过对 Redis 的请求协议的分析，可得出它的规律，先加上标志位，计算参数总数量，再循环合并各个参数的字节数量、值就可以了

回复

```
func GetReply(reply []byte) (interface{}, error) {
    replyType := reply[0]
    switch replyType {
    case StatusReply:
        return doStatusReply(reply[1:])
    case ErrorReply:
        return doErrorReply(reply[1:])
    case IntegerReply:
        return doIntegerReply(reply[1:])
    case BulkReply:
        return doBulkReply(reply[1:])
    case MultiBulkReply:
        return doMultiBulkReply(reply[1:])
    default:
        return nil, nil
    }
}

func doStatusReply(reply []byte) (string, error) {
    if len(reply) == 3 && reply[1] == 'O' && reply[2] == 'K' {
        return OkReply, nil
    }

    if len(reply) == 5 && reply[1] == 'P' && reply[2] == 'O' && reply[3] == 'N'
 && reply[4] == 'G' {
        return PongReply, nil
    }

    return string(reply), nil
}

func doErrorReply(reply []byte) (string, error) {
    return string(reply), nil
}

func doIntegerReply(reply []byte) (int, error) {
    pos := getFlagPos('\r', reply)
    result, err := strconv.Atoi(string(reply[:pos]))
    if err != nil {
        return 0, err
    }

    return result, nil
}
```

```
...
```

在这里我们对所有回复类型进行了分发，不同的回复标志位对应不同的处理方式，在这里需求注意几项问题，如下：

1、当请求的值不存在，会将特殊值 -1 用作回复

2、服务器发送的所有字符串都由 CRLF 结尾

3、多条批量回复是可基于批量回复的，要注意理解

4、无内容的多条批量回复是存在的

最重要的是，对不同回复的规则的把控，能够让你更好的理解 Redis 的请求、回复的交互过程

小结

写这篇文章的起因，是因为常常在使用 Redis 时，只是用，你不知道它是基于什么样的通讯协议来通讯，这样的感觉是十分难受的

通过本文的讲解，我相信你已经大致了解 Redis 客户端是怎么样和服务端交互，也清楚了其所用的通讯原理，希望能够对你有所帮助！

最后，如果想详细查看代码，右拐项目地址：https://github.com/EDDYCJY/redis-protocol-example

如果对你有所帮助，欢迎点个 Star

参考

- 通信协议

使用 Gomock 进行单元测试

在实际项目中，需要进行单元测试的时候。却往往发现有一大堆依赖项。这时候就是 Gomock 大显身手的时候了

Gomock 是 Go 语言的一个 mock 框架，官方的那种

安装

```
$ go get -u github.com/golang/mock/gomock
$ go install github.com/golang/mock/mockgen
```

1. 第一步：我们将安装 gomock 第三方库和 mock 代码的生成工具 mockgen。而后者可以大大的节省我们的工作量。只需要了解其使用方式就可以
2. 第二步：输入 `mockgen` 验证代码生成工具是否安装正确。若无法正常响应，请检查 `bin` 目录下是否包含该二进制文件

用法

在 `mockgen` 命令中，支持两种生成模式：

1. source：从源文件生成 mock 接口（通过 -source 启用）

```
mockgen -source=foo.go [other options]
```

2. reflect：通过使用反射程序来生成 mock 接口。它通过传递两个非标志参数来启用：导入路径和逗号分隔的接口列表

```
mockgen database/sql/driver Conn,Driver
```

从本质上来讲，两种方式生成的 mock 代码并没有什么区别。因此选择合适的就可以了

写测试用例

在本文将模拟一个简单 Demo 来编写测试用例，熟悉整体的测试流程

步骤

1. 想清楚整体逻辑

2. 定义想要（模拟）依赖项的 interface（接口）
3. 使用 `mockgen` 命令对所需 mock 的 interface 生成 mock 文件
4. 编写单元测试的逻辑，在测试中使用 mock
5. 进行单元测试的验证

目录

```
├─── mock
├─── person
|    └─── male.go
└─── user
     ├─── user.go
     └─── user_test.go
```

编写

interface 方法

打开 person/male.go 文件，写入以下内容：

```
package person

type Male interface {
    Get(id int64) error
}
```

调用方法

打开 user/user.go 文件，写入以下内容：

```
package user

import "github.com/EDDYCJY/mockd/person"

type User struct {
    Person person.Male
}

func NewUser(p person.Male) *User {
    return &User{Person: p}
}

func (u *User) GetUserInfo(id int64) error {
```

```
    return u.Person.Get(id)
}
```

生成 mock 文件

回到 `mockd/` 的根目录下，执行以下命令

```
$ mockgen -source=./person/male.go -destination=./mock/male_mock.go -package=moc
k
```

在执行完毕后，可以发现 `mock/` 目录下多出了 male_mock.go 文件，这就是 mock 文件。那么命令中的指令又分别有什么用呢？如下：

- -source：设置需要模拟（mock）的接口文件
- -destination：设置 mock 文件输出的地方，若不设置则打印到标准输出中
- -package：设置 mock 文件的包名，若不设置则为 `mock_` 前缀加上文件名（如本文的包名会为 mock_person）

想了解更多的指令符，可参见 官方文档

输出的 mock 文件

```
// Code generated by MockGen. DO NOT EDIT.
// Source: ./person/male.go

// Package mock is a generated GoMock package.
package mock

import (
    gomock "github.com/golang/mock/gomock"
    reflect "reflect"
)

// MockMale is a mock of Male interface
type MockMale struct {
    ctrl     *gomock.Controller
    recorder *MockMaleMockRecorder
}

// MockMaleMockRecorder is the mock recorder for MockMale
type MockMaleMockRecorder struct {
    mock *MockMale
}
```

```
// NewMockMale creates a new mock instance
func NewMockMale(ctrl *gomock.Controller) *MockMale {
    mock := &MockMale{ctrl: ctrl}
    mock.recorder = &MockMaleMockRecorder{mock}
    return mock
}

// EXPECT returns an object that allows the caller to indicate expected use
func (m *MockMale) EXPECT() *MockMaleMockRecorder {
    return m.recorder
}

// Get mocks base method
func (m *MockMale) Get(id int64) error {
    ret := m.ctrl.Call(m, "Get", id)
    ret0, _ := ret[0].(error)
    return ret0
}

// Get indicates an expected call of Get
func (mr *MockMaleMockRecorder) Get(id interface{}) *gomock.Call {
    return mr.mock.ctrl.RecordCallWithMethodType(mr.mock, "Get", reflect.TypeOf
((*MockMale)(nil).Get), id)
}
```

测试用例

打开 user/user_test.go 文件，写入以下内容：

```
package user

import (
    "testing"

    "github.com/EDDYCJY/mockd/mock"

    "github.com/golang/mock/gomock"
)

func TestUser_GetUserInfo(t *testing.T) {
    ctl := gomock.NewController(t)
    defer ctl.Finish()

    var id int64 = 1
    mockMale := mock.NewMockMale(ctl)
```

```
    gomock.InOrder(
        mockMale.EXPECT().Get(id).Return(nil),
    )

    user := NewUser(mockMale)
    err := user.GetUserInfo(id)
    if err != nil {
        t.Errorf("user.GetUserInfo err: %v", err)
    }
}
```

1. gomock.NewController：返回 `gomock.Controller` ，它代表 mock 生态系统中的顶级控件。定义了 mock 对象的范围、生命周期和期待值。另外它在多个 goroutine 中是安全的

2. mock.NewMockMale：创建一个新的 mock 实例

3. gomock.InOrder：声明给定的调用应按顺序进行（是对 gomock.After 的二次封装）

4. mockMale.EXPECT().Get(id).Return(nil)：这里有三个步骤， `EXPECT()` 返回一个允许调用者设置**期望**和**返回值**的对象。 `Get(id)` 是设置入参并调用 mock 实例中的方法。 `Return(nil)` 是设置先前调用的方法出参。简单来说，就是设置入参并调用，最后设置返回值

5. NewUser(mockMale)：创建 User 实例，值得注意的是，在这里**注入了 mock 对象**，因此实际在随后的 `user.GetUserInfo(id)` 调用（入参：id 为 1）中。它调用的是我们事先模拟好的 mock 方法

6. ctl.Finish()：进行 mock 用例的期望值断言，一般会使用 `defer` 延迟执行，以防止我们忘记这一操作

测试

回到 `mockd/` 的根目录下，执行以下命令

```
$ go test ./user
ok      github.com/EDDYCJY/mockd/user
```

看到这样的结果，就大功告成啦！你可以自己调整一下 `Return()` 的返回值，以此得到不一样的测试结果哦 😄

查看测试情况

测试覆盖率

```
$ go test -cover ./user
ok      github.com/EDDYCJY/mockd/user    (cached)    coverage: 100.0% of stateme
nts
```

可通过设置 `-cover` 标志符来开启覆盖率的统计，展示内容为 `coverage: 100.0%` 。

可视化界面

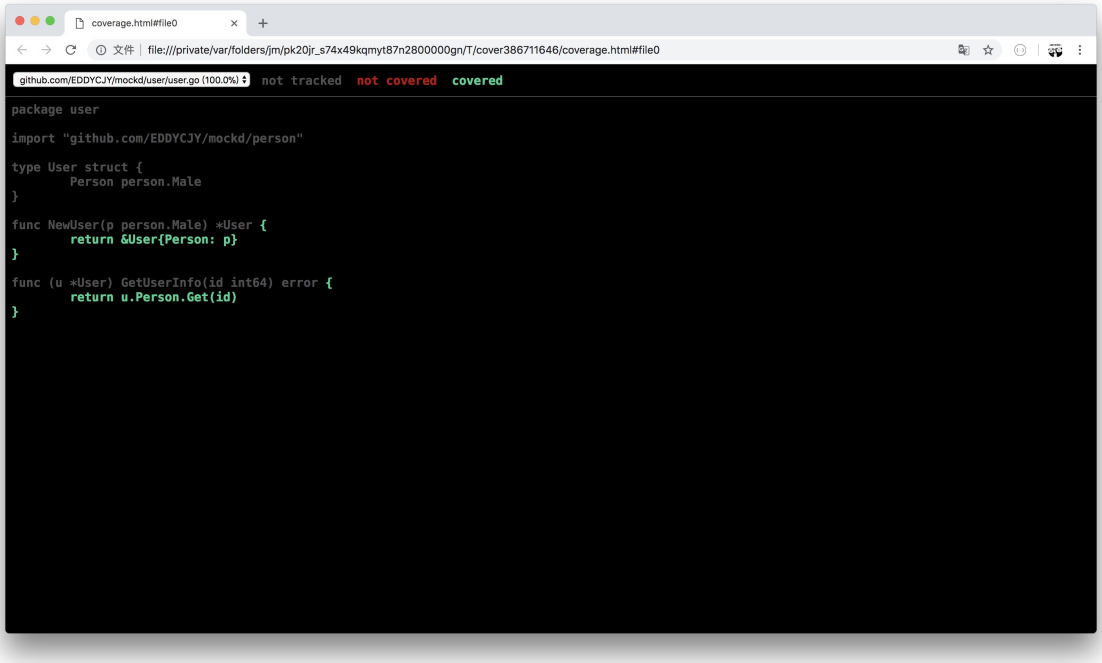
1. 生成测试覆盖率的 profile 文件

```
$ go test ./... -coverprofile=cover.out
```

2. 利用 profile 文件生成可视化界面

```
$ go tool cover -html=cover.out
```

3. 查看可视化界面，分析覆盖情况



更多

一、常用 mock 方法

调用方法

- Call.Do()：声明在匹配时要运行的操作
- Call.DoAndReturn()：声明在匹配调用时要运行的操作，并且模拟返回该函数的返回值
- Call.MaxTimes()：设置最大的调用次数为 n 次
- Call.MinTimes()：设置最小的调用次数为 n 次
- Call.AnyTimes()：允许调用次数为 0 次或更多次
- Call.Times()：设置调用次数为 n 次

参数匹配

- gomock.Any()：匹配任意值
- gomock.Eq()：通过反射匹配到指定的类型值，而不需要手动设置
- gomock.Nil()：返回 nil

建议更多的方法可参见 官方文档

二、生成多个 mock 文件

你可能会想一条条命令生成 mock 文件，岂不得崩溃？

当然，官方提供了更方便的方式，我们可以利用 `go:generate` 来完成批量处理的功能

```
go generate [-run regexp] [-n] [-v] [-x] [build flags] [file.go... | packages]
```

修改 interface 方法

打开 person/male.go 文件，修改为以下内容：

```
package person

//go:generate mockgen -destination=../mock/male_mock.go -package=mock github.com/EDDYCJY/mockd/person Male

type Male interface {
    Get(id int64) error
}
```

我们关注到 `go:generate` 这条语句，可分为以下部分：

1. 声明 `//go:generate` （注意不要留空格）
2. 使用 `mockgen` 命令
3. 定义 `-destination`

4. 定义 `-package`
5. 定义 `source`，此处为 person 的包路径
6. 定义 `interfaces`，此处为 `Male`

重新生成 mock 文件

回到 `mockd/` 的根目录下，执行以下命令

```
$ go generate ./...
```

再检查 `mock/` 发现也已经正确生成了，在多个文件时是不是很方便呢

总结

在单元测试这一环，gomock 给我们提供了极大的便利。能够 mock 掉许许多多的依赖项

其中还有很多的使用方式和功能。你可以 mark 住后详细阅读下官方文档，记忆会更深刻

在 Go 中恰到好处的内存对齐

在实际项目中，需要进行单元测试的时候。却往往发现有一大堆依赖项。这时候就是 Gomock 大显身手的时候了

Gomock 是 Go 语言的一个 mock 框架，官方的那种

安装

```
$ go get -u github.com/golang/mock/gomock
$ go install github.com/golang/mock/mockgen
```

1. 第一步：我们将安装 gomock 第三方库和 mock 代码的生成工具 mockgen。而后者可以大大的节省我们的工作量。只需要了解其使用方式就可以
2. 第二步：输入 `mockgen` 验证代码生成工具是否安装正确。若无法正常响应，请检查 `bin` 目录下是否包含该二进制文件

用法

在 `mockgen` 命令中，支持两种生成模式：

1. source：从源文件生成 mock 接口（通过 -source 启用）

```
mockgen -source=foo.go [other options]
```

2. reflect：通过使用反射程序来生成 mock 接口。它通过传递两个非标志参数来启用：导入路径和逗号分隔的接口列表

```
mockgen database/sql/driver Conn,Driver
```

从本质上来讲，两种方式生成的 mock 代码并没有什么区别。因此选择合适的就可以了

写测试用例

在本文将模拟一个简单 Demo 来编写测试用例，熟悉整体的测试流程

步骤

1. 想清楚整体逻辑

2. 定义想要（模拟）依赖项的 interface（接口）
3. 使用 `mockgen` 命令对所需 mock 的 interface 生成 mock 文件
4. 编写单元测试的逻辑，在测试中使用 mock
5. 进行单元测试的验证

目录

```
├─── mock
├─── person
|    └─── male.go
└─── user
     ├─── user.go
     └─── user_test.go
```

编写

interface 方法

打开 person/male.go 文件，写入以下内容：

```
package person

type Male interface {
    Get(id int64) error
}
```

调用方法

打开 user/user.go 文件，写入以下内容：

```
package user

import "github.com/EDDYCJY/mockd/person"

type User struct {
    Person person.Male
}

func NewUser(p person.Male) *User {
    return &User{Person: p}
}

func (u *User) GetUserInfo(id int64) error {
```

```
    return u.Person.Get(id)
}
```

生成 mock 文件

回到 `mockd/` 的根目录下，执行以下命令

```
$ mockgen -source=./person/male.go -destination=./mock/male_mock.go -package=moc
k
```

在执行完毕后，可以发现 `mock/` 目录下多出了 male_mock.go 文件，这就是 mock 文件。那么命令中的指令又分别有什么用呢？如下：

- -source：设置需要模拟（mock）的接口文件
- -destination：设置 mock 文件输出的地方，若不设置则打印到标准输出中
- -package：设置 mock 文件的包名，若不设置则为 `mock_` 前缀加上文件名（如本文的包名会为 mock_person）

想了解更多的指令符，可参见 官方文档

输出的 mock 文件

```
// Code generated by MockGen. DO NOT EDIT.
// Source: ./person/male.go

// Package mock is a generated GoMock package.
package mock

import (
    gomock "github.com/golang/mock/gomock"
    reflect "reflect"
)

// MockMale is a mock of Male interface
type MockMale struct {
    ctrl     *gomock.Controller
    recorder *MockMaleMockRecorder
}

// MockMaleMockRecorder is the mock recorder for MockMale
type MockMaleMockRecorder struct {
    mock *MockMale
}
```

```
// NewMockMale creates a new mock instance
func NewMockMale(ctrl *gomock.Controller) *MockMale {
    mock := &MockMale{ctrl: ctrl}
    mock.recorder = &MockMaleMockRecorder{mock}
    return mock
}

// EXPECT returns an object that allows the caller to indicate expected use
func (m *MockMale) EXPECT() *MockMaleMockRecorder {
    return m.recorder
}

// Get mocks base method
func (m *MockMale) Get(id int64) error {
    ret := m.ctrl.Call(m, "Get", id)
    ret0, _ := ret[0].(error)
    return ret0
}

// Get indicates an expected call of Get
func (mr *MockMaleMockRecorder) Get(id interface{}) *gomock.Call {
    return mr.mock.ctrl.RecordCallWithMethodType(mr.mock, "Get", reflect.TypeOf
((*MockMale)(nil).Get), id)
}
```

测试用例

打开 user/user_test.go 文件，写入以下内容：

```
package user

import (
    "testing"

    "github.com/EDDYCJY/mockd/mock"

    "github.com/golang/mock/gomock"
)

func TestUser_GetUserInfo(t *testing.T) {
    ctl := gomock.NewController(t)
    defer ctl.Finish()

    var id int64 = 1
    mockMale := mock.NewMockMale(ctl)
```

```
    gomock.InOrder(
        mockMale.EXPECT().Get(id).Return(nil),
    )

    user := NewUser(mockMale)
    err := user.GetUserInfo(id)
    if err != nil {
        t.Errorf("user.GetUserInfo err: %v", err)
    }
}
```

1. gomock.NewController：返回 `gomock.Controller`，它代表 mock 生态系统中的顶级控件。定义了 mock 对象的范围、生命周期和期待值。另外它在多个 goroutine 中是安全的

2. mock.NewMockMale：创建一个新的 mock 实例

3. gomock.InOrder：声明给定的调用应按顺序进行（是对 gomock.After 的二次封装）

4. mockMale.EXPECT().Get(id).Return(nil)：这里有三个步骤，`EXPECT()` 返回一个允许调用者设置**期望**和**返回值**的对象。`Get(id)` 是设置入参并调用 mock 实例中的方法。`Return(nil)` 是设置先前调用的方法出参。简单来说，就是设置入参并调用，最后设置返回值

5. NewUser(mockMale)：创建 User 实例，值得注意的是，在这里**注入了 mock 对象**，因此实际在随后的 `user.GetUserInfo(id)` 调用（入参：id 为 1）中。它调用的是我们事先模拟好的 mock 方法

6. ctl.Finish()：进行 mock 用例的期望值断言，一般会使用 `defer` 延迟执行，以防止我们忘记这一操作

测试

回到 `mockd/` 的根目录下，执行以下命令

```
$ go test ./user
ok      github.com/EDDYCJY/mockd/user
```

看到这样的结果，就大功告成啦！你可以自己调整一下 `Return()` 的返回值，以此得到不一样的测试结果哦 😄

查看测试情况

测试覆盖率

```
$ go test -cover ./user
ok      github.com/EDDYCJY/mockd/user    (cached)    coverage: 100.0% of stateme
nts
```

可通过设置 `-cover` 标志符来开启覆盖率的统计，展示内容为 `coverage: 100.0%` 。

可视化界面

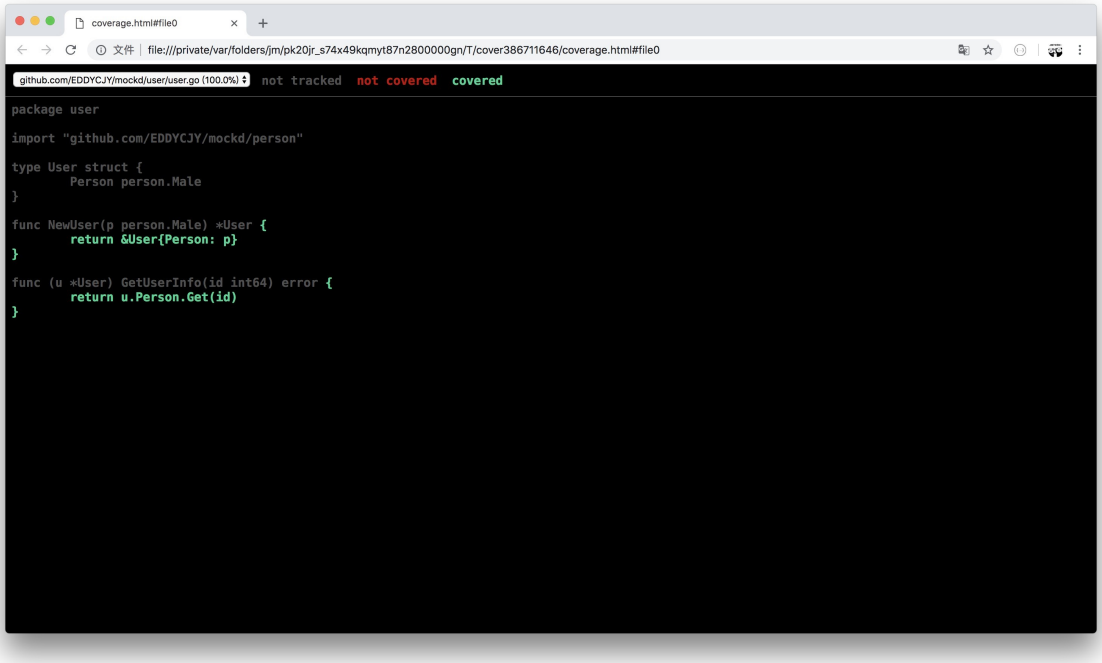
1. 生成测试覆盖率的 profile 文件

```
$ go test ./... -coverprofile=cover.out
```

2. 利用 profile 文件生成可视化界面

```
$ go tool cover -html=cover.out
```

3. 查看可视化界面，分析覆盖情况



更多

一、常用 mock 方法

调用方法

- Call.Do()：声明在匹配时要运行的操作
- Call.DoAndReturn()：声明在匹配调用时要运行的操作，并且模拟返回该函数的返回值
- Call.MaxTimes()：设置最大的调用次数为 n 次
- Call.MinTimes()：设置最小的调用次数为 n 次
- Call.AnyTimes()：允许调用次数为 0 次或更多次
- Call.Times()：设置调用次数为 n 次

参数匹配

- gomock.Any()：匹配任意值
- gomock.Eq()：通过反射匹配到指定的类型值，而不需要手动设置
- gomock.Nil()：返回 nil

建议更多的方法可参见 官方文档

二、生成多个 mock 文件

你可能会想一条条命令生成 mock 文件，岂不得崩溃？

当然，官方提供了更方便的方式，我们可以利用 `go:generate` 来完成批量处理的功能

```
go generate [-run regexp] [-n] [-v] [-x] [build flags] [file.go... | packages]
```

修改 interface 方法

打开 person/male.go 文件，修改为以下内容：

```
package person

//go:generate mockgen -destination=../mock/male_mock.go -package=mock github.com/EDDYCJY/mockd/person Male

type Male interface {
    Get(id int64) error
}
```

我们关注到 `go:generate` 这条语句，可分为以下部分：

1. 声明 `//go:generate` （注意不要留空格）
2. 使用 `mockgen` 命令
3. 定义 `-destination`

4. 定义 `-package`
5. 定义 `source`，此处为 person 的包路径
6. 定义 `interfaces`，此处为 `Male`

重新生成 mock 文件

回到 `mockd/` 的根目录下，执行以下命令

```
$ go generate ./...
```

再检查 `mock/` 发现也已经正确生成了，在多个文件时是不是很方便呢

总结

在单元测试这一环，gomock 给我们提供了极大的便利。能够 mock 掉许许多多的依赖项

其中还有很多的使用方式和功能。你可以 mark 住后详细阅读下官方文档，记忆会更深刻

来，控制一下 goroutine 的并发数量

问题

```
func main() {
    userCount := math.MaxInt64
    for i := 0; i < userCount; i++ {
        go func(i int) {
            // 做一些各种各样的业务逻辑处理
            fmt.Printf("go func: %d\n", i)
            time.Sleep(time.Second)
        }(i)
    }
}
```

在这里，假设 `userCount` 是一个外部传入的参数（不可预测，有可能值非常大），有人会全部丢进去循环。想着全部都并发 goroutine 去同时做某一件事。觉得这样子会效率会更高，对不对！

那么，你觉得这里有没有什么问题？

噩梦般的开始

当然，在**特定场景下**，问题可大了。因为在本文被丢进去同时并发的可是一个极端值。我们可以一起观察下图的指标分析，看看情况有多“崩溃”。下图是上述代码的表现：

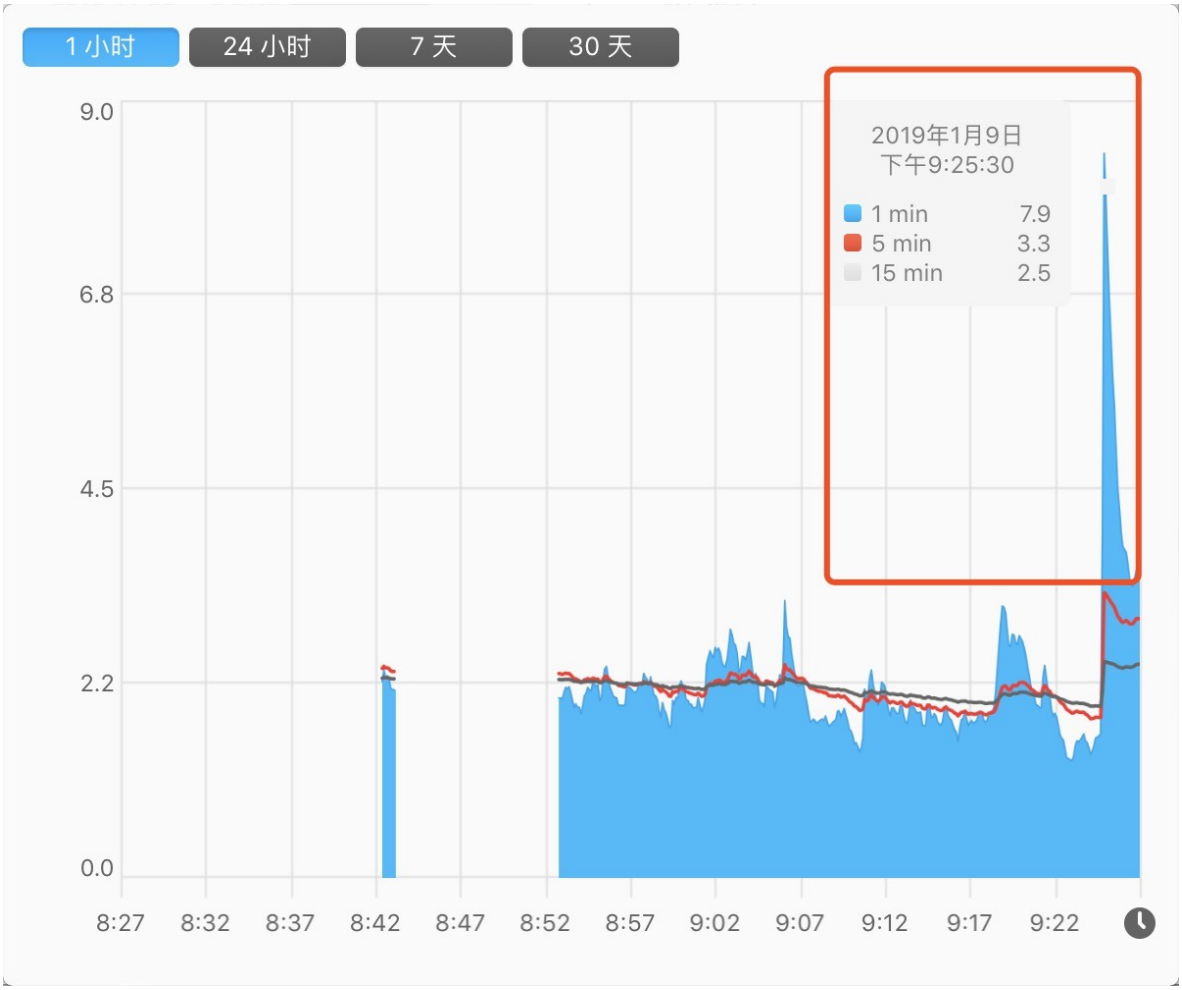
输出结果

```
...
go func: 5839
go func: 5840
go func: 5841
go func: 5842
go func: 5915
go func: 5524
go func: 5916
go func: 8209
go func: 8264
signal: killed
```

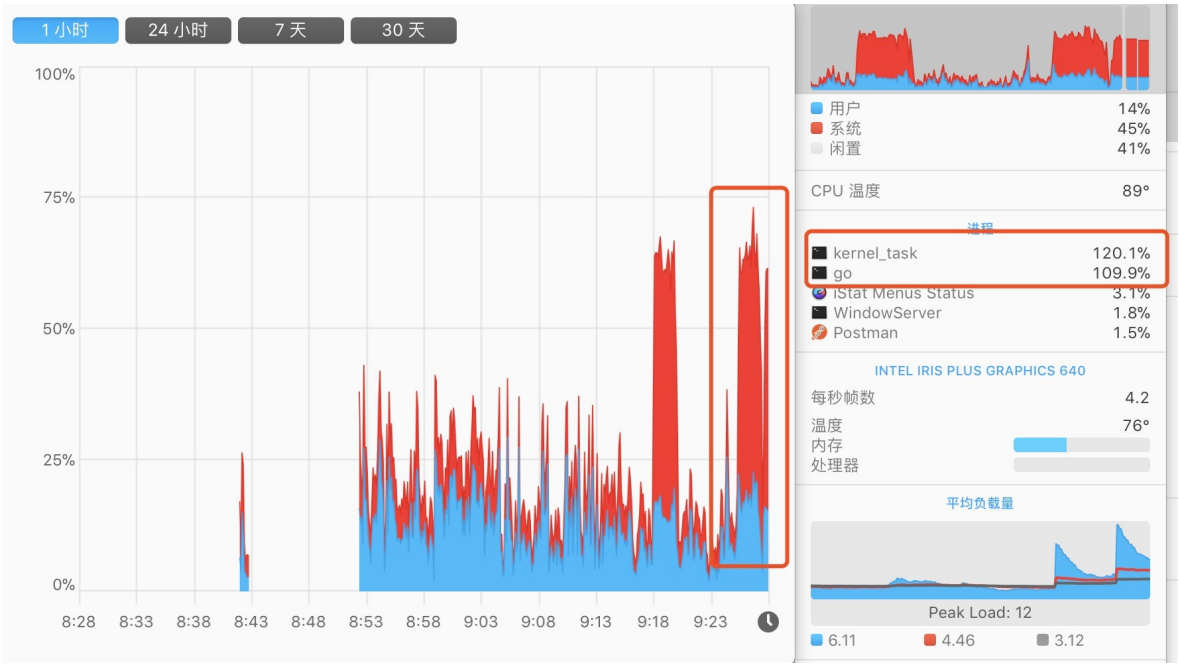
如果你自己执行过代码，在“输出结果”上你会遇到如下问题：

- 系统资源占用率不断上涨
- 输出一定数量后：控制台就不再刷新输出最新的值了
- 信号量：signal: killed

系统负载

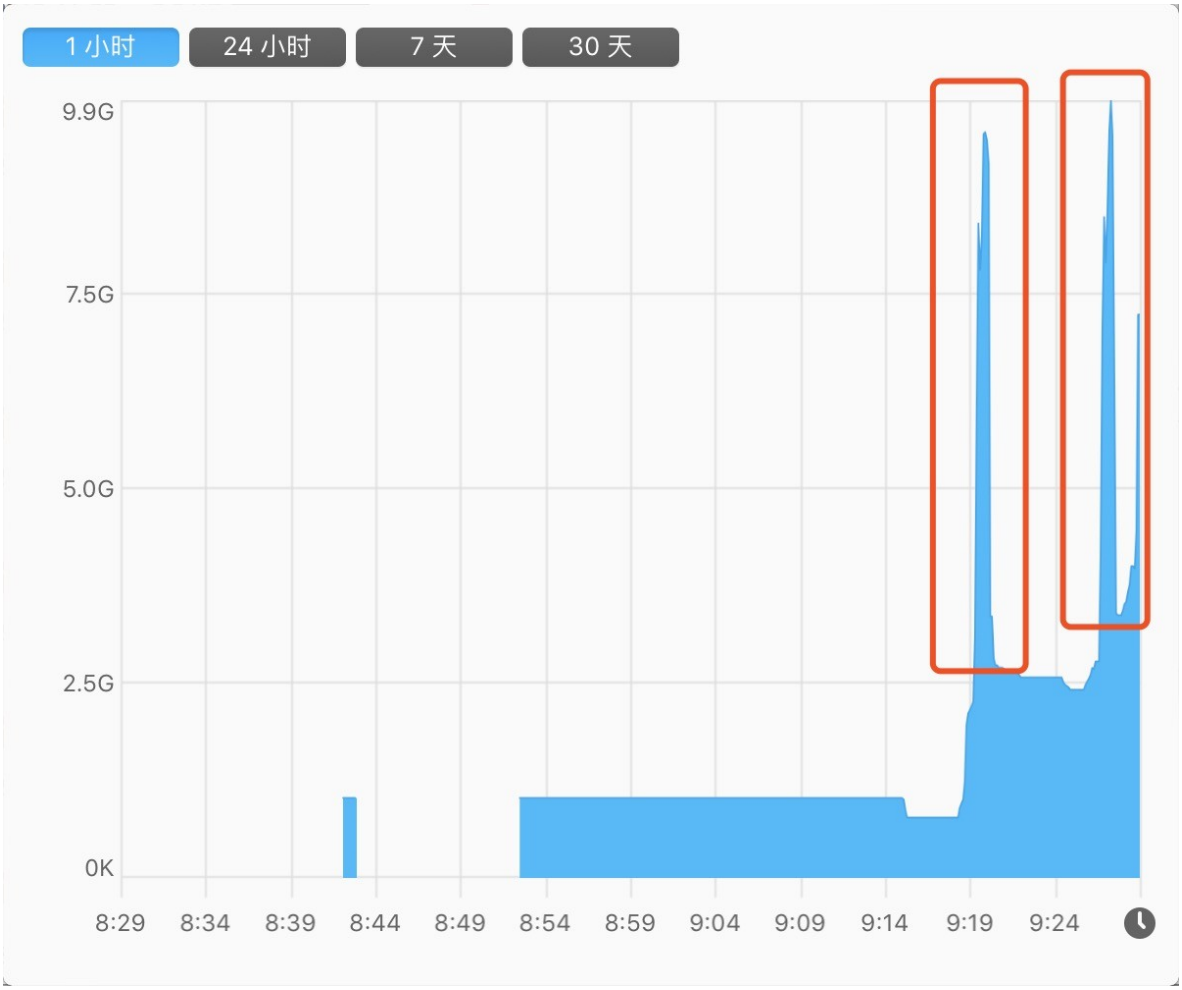


CPU



短时间内系统负载暴增

虚拟内存



短时间内占用的虚拟内存暴增

top

```
PID    COMMAND      %CPU  TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  P
PID   STATE    BOOSTS
...
73414  test         100.2 01:59.50 9/1   0    18    6801M+ 0B     114G+  73403 7
3403 running  *0[1]
```

小结

如果仔细看过监控工具的示意图，就可以知道其实我间隔的执行了两次，能看到系统间的使用率幅度非常大。当进程被杀掉后，整体又恢复为正常值

在这里，我们回到主题，就是在**不控制并发的 goroutine 数量** 会发生什么问题？大致如下：

- CPU 使用率浮动上涨
- Memory 占用不断上涨。也可以看看 CMPRS，它表示进程的压缩数据的字节数。已经到达 114G+ 了
- 主进程崩溃（被杀掉了）

简单来说，“崩溃”的原因就是对系统资源的占用过大。常见的比如：打开文件数（too many files open）、内存占用等等

危害

对该台服务器产生非常大的影响，影响自身及相关联的应用。很有可能导致不可用或响应缓慢，另外启动了复数“失控”的 goroutine，导致程序流转混乱

解决方案

在前面花了大量篇幅，渲染了在存在大量并发 goroutine 数量时，不控制的话会出现“严重”的问题，接下来一起思考下解决方案。如下：

1. 控制/限制 goroutine 同时并发运行的数量
2. 改变应用程序的逻辑写法（避免大规模的使用系统资源和等待）
3. ~~调整服务的硬件配置、最大打开数、内存等阈值~~

控制 goroutine 并发数量

接下来正式的开始解决这个问题，希望你认真阅读的同时加以思考，因为这个问题在实际项目中真的是太常见了！

问题已经抛出来了，你需要做的是**想想有什么办法**解决这个问题。建议你自行思考一下技术方案。再接着往下看 :-)

尝试 chan

```
func main() {
    userCount := 10
    ch := make(chan bool, 2)
    for i := 0; i < userCount; i++ {
        ch <- true
        go Read(ch, i)
    }

    //time.Sleep(time.Second)
}

func Read(ch chan bool, i int) {
    fmt.Printf("go func: %d\n", i)
    <- ch
}
```

输出结果：

```
go func: 1
go func: 2
go func: 3
go func: 4
go func: 5
go func: 6
go func: 7
go func: 8
go func: 0
```

嗯，我们似乎很好的控制了 2 个 2 个的“顺序”执行多个 goroutine。但是，问题出现了。你仔细数一下输出结果，才 9 个值？

这明显就不对。原因出在当主协程结束时，子协程也是会被终止掉的。因此剩余的 goroutine 没来及把值输出，就被送上路了（不信你把 `time.Sleep` 打开看看，看看输出数量）

尝试 sync

```
...
var wg = sync.WaitGroup{}

func main() {
    userCount := 10
    for i := 0; i < userCount; i++ {
        wg.Add(1)
        go Read(i)
    }

    wg.Wait()
}

func Read(i int) {
    defer wg.Done()
    fmt.Printf("go func: %d\n", i)
}
```

嗯，单纯的使用 `sync.WaitGroup` 也不行。没有控制到同时并发的 goroutine 数量（代指达不到本文所要求的目标）

小结

单纯**简单**使用 channel 或 sync 都有明显缺陷，不行。我们再看看组件配合能不能实现

尝试 chan + sync

```
...
var wg = sync.WaitGroup{}

func main() {
    userCount := 10
    ch := make(chan bool, 2)
    for i := 0; i < userCount; i++ {
        wg.Add(1)
        go Read(ch, i)
    }

    wg.Wait()
}

func Read(ch chan bool, i int) {
    defer wg.Done()
```

```
    ch <- true
    fmt.Printf("go func: %d, time: %d\n", i, time.Now().Unix())
    time.Sleep(time.Second)
    <-ch
}
```

输出结果：

```
go func: 9, time: 1547911938
go func: 1, time: 1547911938
go func: 6, time: 1547911939
go func: 7, time: 1547911939
go func: 8, time: 1547911940
go func: 0, time: 1547911940
go func: 3, time: 1547911941
go func: 2, time: 1547911941
go func: 4, time: 1547911942
go func: 5, time: 1547911942
```

从输出结果来看，确实实现了控制 goroutine 以 2 个 2 个的数量去执行我们的“业务逻辑”，当然结果集也理所应当的是乱序输出

方案一：简单 Semaphore

在确立了简单使用 chan + sync 的方案是可行后，我们重新将流转逻辑封装为 gsema，主程序变成如下：

```
import (
    "fmt"
    "time"

    "github.com/EDDYCJY/gsema"
)

var sema = gsema.NewSemaphore(3)

func main() {
    userCount := 10
    for i := 0; i < userCount; i++ {
        go Read(i)
    }

    sema.Wait()
}
```

```
func Read(i int) {
    defer sema.Done()
    sema.Add(1)

    fmt.Printf("go func: %d, time: %d\n", i, time.Now().Unix())
    time.Sleep(time.Second)
}
```

分析方案

在上述代码中，程序执行流程如下：

- 设置允许的并发数目为 3 个
- 循环 10 次，每次启动一个 goroutine 来执行任务
- 每一个 goroutine 在内部利用 `sema` 进行调控是否阻塞
- 按允许并发数逐渐释出 goroutine，最后结束任务

看上去人模人样，没什么严重问题。但却有一个“大”坑，认真看到第二点“每次启动一个 goroutine”这句话。这里**有点问题**，提前产生那么多的 goroutine 会不会有什么问题，接下来一起分析下利弊，如下：

利

- 适合**量不大、复杂度低**的使用场景
  - 几百几千个、几十万个也是可以接受的（看具体业务场景）
  - 实际业务逻辑在运行前就已经被阻塞等待了（因为并发数受限），基本实际业务逻辑损耗的性能比 goroutine 本身大
  - goroutine 本身很轻便，仅损耗极少许的内存空间和调度。这种等待响应的情况都是躺好了，等待任务唤醒
- Semaphore 操作复杂度低且流转简单，容易控制

弊

- 不适合**量很大、复杂度高**的使用场景
  - 有几百万、几千万个 goroutine 的话，就浪费了大量调度 goroutine 和内存空间。恰好你的服务器也接受不了的话
- Semaphore 操作复杂度提高，要管理更多的状态

小结

- 基于什么业务场景，就用什么方案去做事
- 有足够的时间，允许你去追求更优秀、极致的方案（用第三方库也行）

用哪种方案，我认为主要基于以上两点去思考，都是 OK 的。没有对错，只有当前业务场景能不能接受，这个预先启动的 goroutine 数量你的系统是否能够接受

当然了，常见/简单的 Go 应用采用这类技术方案，基本就能解决问题了。因为像本文第一节“问题”如此超巨大数量的情况，情况很少。其并不存在那些“特殊性”。因此用这个方案基本 OK

灵活控制 goroutine 并发数量

小手一紧。隔壁老王发现了新的问题。“方案一”中，在**输入输出一体**的情况下，在常见的业务场景中确实可以

但，这次新的业务场景比较特殊，要控制输入的数量，以此达到**改变允许并发运行 goroutine 的数量**。我们仔细想想，要做出如下改变：

- 输入/输出要抽离，才可以分别控制
- 输入/输出要可变，理所应当在 for-loop 中（可设置数值的地方）
- 允许改变 goroutine 并发数量，但它也必须有一个**最大值**（因为允许改变是相对）

方案二：灵活 chan + sync

```
package main

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

var wg sync.WaitGroup

func main() {
    userCount := 10
    ch := make(chan int, 5)
    for i := 0; i < userCount; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
```

```
            for d := range ch {
                fmt.Printf("go func: %d, time: %d\n", d, time.Now().Unix())
                time.Sleep(time.Second * time.Duration(d))
            }
        }()
    }

    for i := 0; i < 10; i++ {
        ch <- 1
        ch <- 2
        //time.Sleep(time.Second)
    }

    close(ch)
    wg.Wait()
}
```

输出结果：

```
...
go func: 1, time: 1547950567
go func: 3, time: 1547950567
go func: 1, time: 1547950567
go func: 2, time: 1547950567
go func: 2, time: 1547950567
go func: 3, time: 1547950567
go func: 1, time: 1547950568
go func: 2, time: 1547950568
go func: 3, time: 1547950568
go func: 1, time: 1547950568
go func: 3, time: 1547950569
go func: 2, time: 1547950569
```

在“方案二”中，我们可以随时随地的根据新的业务需求，做如下事情：

- 变更 channel 的输入数量
- 能够根据特殊情况，变更 channel 的循环值
- 变更最大允许并发的 goroutine 数量

总的来说，就是可控空间都尽量放开了，是不是更加灵活了呢 :-)

方案三：第三方库

- go-playground/pool
- nozzle/throttler
- Jeffail/tunny
- panjf2000/ants

比较成熟的第三方库也不少，基本都是以生成和管理 goroutine 为目标的池工具。我简单列了几个，具体建议大家阅读下源码或者多找找，原理相似

总结

在本文的开头，我花了大力气（极端数量），告诉你**同时并发过多的 goroutine 数量会导致系统占用资源不断上涨。最终该服务崩盘的极端情况**。为的是希望你今后避免这种问题，给你留下深刻的印象

接下来我们以 “控制 goroutine 并发数量” 为主题，展开了一番分析。分别给出了三种方案。在我看来，各具优缺点，我建议你**挑选合适自身场景的技术方案**就可以了

因为，有不同类型的技术方案也能解决这个问题，千人千面。本文推荐的是较常见的解决方案，也欢迎大家在评论区继续补充 :-)

for-loop 与 json.Unmarshal 性能分析概要

在项目中，常常会遇到循环交换赋值的数据处理场景，尤其是 RPC，数据交互格式要转为 Protobuf，赋值是无法避免的。一般会有如下几种做法：

- for
- for range
- json.Marshal/Unmarshal

这时候又面临“选择困难症”，用哪个好？又想代码量少，又担心性能有没有影响啊...

为了弄清楚这个疑惑，接下来将分别编写三种使用场景。来简单看看它们的性能情况，看看谁更“好”

功能代码

```
...
type Person struct {
    Name   string `json:"name"`
    Age    int    `json:"age"`
    Avatar string `json:"avatar"`
    Type   string `json:"type"`
}

type AgainPerson struct {
    Name   string `json:"name"`
    Age    int    `json:"age"`
    Avatar string `json:"avatar"`
    Type   string `json:"type"`
}

const MAX = 10000

func InitPerson() []Person {
    var persons []Person
    for i := 0; i < MAX; i++ {
        persons = append(persons, Person{
            Name:   "EDDYCJY",
            Age:    i,
            Avatar: "https://github.com/EDDYCJY",
            Type:   "Person",
        })
    }
```

```
    return persons
}

func ForStruct(p []Person, count int) {
    for i := 0; i < count; i++ {
        _, _ = i, p[i]
    }
}

func ForRangeStruct(p []Person) {
    for i, v := range p {
        _, _ = i, v
    }
}

func JsonToStruct(data []byte, againPerson []AgainPerson) ([]AgainPerson, error)
{
    err := json.Unmarshal(data, &againPerson)
    return againPerson, err
}

func JsonIteratorToStruct(data []byte, againPerson []AgainPerson) ([]AgainPerso
n, error) {
    var jsonIter = jsoniter.ConfigCompatibleWithStandardLibrary
    err := jsonIter.Unmarshal(data, &againPerson)
    return againPerson, err
}
```

测试代码

```
...
func BenchmarkForStruct(b *testing.B) {
    person := InitPerson()
    count := len(person)
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        ForStruct(person, count)
    }
}

func BenchmarkForRangeStruct(b *testing.B) {
    person := InitPerson()
```

```
    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        ForRangeStruct(person)
    }
}

func BenchmarkJsonToStruct(b *testing.B) {
    var (
        person = InitPerson()
        againPersons []AgainPerson
    )
    data, err := json.Marshal(person)
    if err != nil {
        b.Fatalf("json.Marshal err: %v", err)
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        JsonToStruct(data, againPersons)
    }
}

func BenchmarkJsonIteratorToStruct(b *testing.B) {
    var (
        person = InitPerson()
        againPersons []AgainPerson
    )
    data, err := json.Marshal(person)
    if err != nil {
        b.Fatalf("json.Marshal err: %v", err)
    }

    b.ResetTimer()
    for i := 0; i < b.N; i++ {
        JsonIteratorToStruct(data, againPersons)
    }
}
```

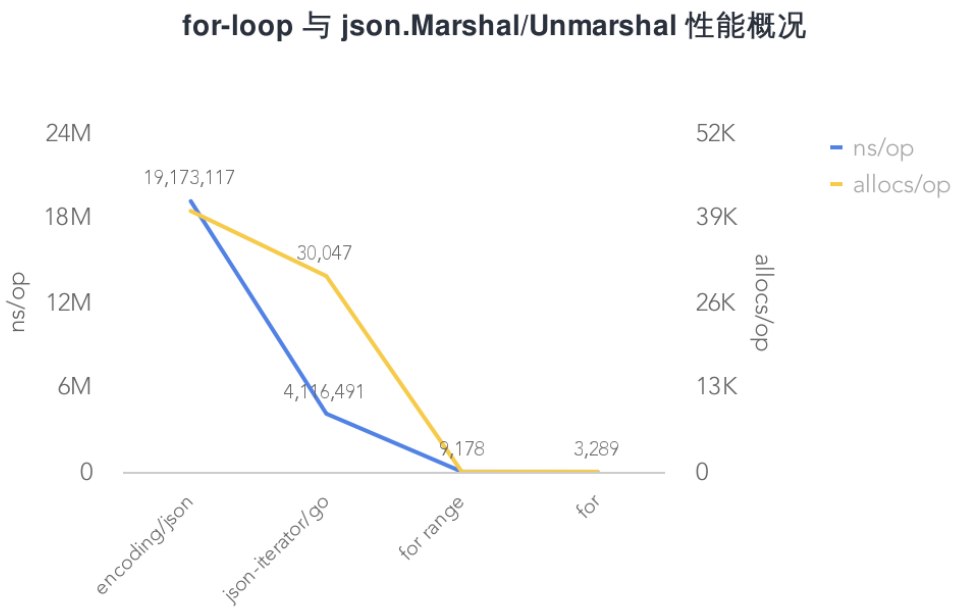
测试结果

```
BenchmarkForStruct-4                    500000          3289 ns/op            0
 B/op          0 allocs/op
BenchmarkForRangeStruct-4               200000          9178 ns/op            0
 B/op          0 allocs/op
```

```
BenchmarkJsonToStruct-4                   100         19173117 ns/op         2618509
 B/op         40036 allocs/op
BenchmarkJsonIteratorToStruct-4           300          4116491 ns/op         3694017
 B/op         30047 allocs/op
```

从测试结果来看，性能排名为：for < for range < json-iterator < encoding/json。接下来我们看看是什么原因导致了这样子的排名？

性能对比



for-loop

在测试结果中，`for range` 在性能上相较 `for` 差。这是为什么呢？在这里我们可以参见 `for range` 的 实现，伪实现如下：

```
for_temp := range
len_temp := len(for_temp)
for index_temp = 0; index_temp < len_temp; index_temp++ {
    value_temp = for_temp[index_temp]
    index = index_temp
    value = value_temp
    original body
}
```

通过分析伪实现，可得知 `for range` 相较 `for` 多做了如下事项

Expression

```
RangeClause = [ ExpressionList "=" | IdentifierList ":=" ] "range" Expression .
```

在循环开始之前会对范围表达式进行求值，多做了“解”表达式的动作，得到了最终的范围值

Copy

```
...
value_temp = for_temp[index_temp]
index = index_temp
value = value_temp
...
```

从伪实现上可以得出， `for range` 始终使用**值拷贝**的方式来生成循环变量。通俗来讲，就是在每次循环时，都会对循环变量重新分配

小结

通过上述的分析，可得知其比 `for` 慢的原因是 `for range` 有额外的性能开销，主要为**值拷贝的动作**导致的性能下降。这是它慢的原因

那么其实在 `for range` 中，我们可以使用 `_` 和 `T[i]` 也能达到和 `for` 差不多的性能。但这可能不是 `for range` 的设计本意了

json.Marshal/Unmarshal

encoding/json

json 互转是在三种方案中最慢的，这是为什么呢？

众所皆知，官方的 `encoding/json` 标准库，是通过大量反射来实现的。那么“慢”，也是必然的。可参见下述代码：

```
...
func newTypeEncoder(t reflect.Type, allowAddr bool) encoderFunc {
    ...
    switch t.Kind() {
    case reflect.Bool:
        return boolEncoder
    case reflect.Int, reflect.Int8, reflect.Int16, reflect.Int32, reflect.Int64:
        return intEncoder
    case reflect.Uint, reflect.Uint8, reflect.Uint16, reflect.Uint32, reflect.Ui
nt64, reflect.Uintptr:
```

```
        return uintEncoder
    case reflect.Float32:
        return float32Encoder
    case reflect.Float64:
        return float64Encoder
    case reflect.String:
        return stringEncoder
    case reflect.Interface:
        return interfaceEncoder
    case reflect.Struct:
        return newStructEncoder(t)
    case reflect.Map:
        return newMapEncoder(t)
    case reflect.Slice:
        return newSliceEncoder(t)
    case reflect.Array:
        return newArrayEncoder(t)
    case reflect.Ptr:
        return newPtrEncoder(t)
    default:
        return unsupportedTypeEncoder
    }
}
```

既然官方的标准库存在一定的“问题”，那么有没有其他解决方法呢？目前在社区里，大多为两类方案。如下：

- 预编译生成代码（提前确定类型），可以解决运行时的反射带来的性能开销。缺点是增加了预生成的步骤
- 优化序列化的逻辑，性能达到最大化

接下来的实验，我们用第二种方案的库来测试，看看有没有改变。另外也推荐大家了解如下项目：

- json-iterator/go
- mailru/easyjson
- pquerna/ffjson

json-iterator/go

目前社区较常用的是 json-iterator/go，我们在测试代码中用到了它

它的用法与标准库 100% 兼容，并且性能有较大提升。我们一起粗略的看下是怎么做到的，如下：

reflect2

利用 modern-go/reflect2 减少运行时调度开销

```
...
type StructDescriptor struct {
    Type   reflect2.Type
    Fields []*Binding
}

...
type Binding struct {
    levels    []int
    Field     reflect2.StructField
    FromNames []string
    ToNames   []string
    Encoder   ValEncoder
    Decoder   ValDecoder
}

type Extension interface {
    UpdateStructDescriptor(structDescriptor *StructDescriptor)
    CreateMapKeyDecoder(typ reflect2.Type) ValDecoder
    CreateMapKeyEncoder(typ reflect2.Type) ValEncoder
    CreateDecoder(typ reflect2.Type) ValDecoder
    CreateEncoder(typ reflect2.Type) ValEncoder
    DecorateDecoder(typ reflect2.Type, decoder ValDecoder) ValDecoder
    DecorateEncoder(typ reflect2.Type, encoder ValEncoder) ValEncoder
}
```

struct Encoder/Decoder Cache

类型为 struct 时，只需要反射一次 Name 和 Type，会缓存 struct Encoder 和 Decoder

```
var typeDecoders = map[string]ValDecoder{}
var fieldDecoders = map[string]ValDecoder{}
var typeEncoders = map[string]ValEncoder{}
var fieldEncoders = map[string]ValEncoder{}
var extensions = []Extension{}

....

fieldNames := calcFieldNames(field.Name(), tagParts[0], tag)
fieldCacheKey := fmt.Sprintf("%s/%s", typ.String(), field.Name())
decoder := fieldDecoders[fieldCacheKey]
if decoder == nil {
```

```
    decoder = decoderOfType(ctx.append(field.Name()), field.Type())
}
encoder := fieldEncoders[fieldCacheKey]
if encoder == nil {
    encoder = encoderOfType(ctx.append(field.Name()), field.Type())
}
```

文本解析优化

小结

相较于官方标准库，第三方库 `json-iterator/go` 在运行时上做的更好。这是它快的原因

有个需要注意的点，在 Go1.10 后 `map` 类型与标准库的已经没有太大的性能差异。但是，例如 `struct` 类型等仍然有较大的性能提高

总结

在本文中，我们首先进行了性能测试，再分析了不同方案，得知为什么了快慢的原因。那么最终在选择方案时，可以根据不同的应用场景去抉择：

- 对性能开销有较高要求：选用 `for`，开销最小
- 中规中矩：选用 `for range`，大对象慎用
- 量小、占用小、数量可控：选用 `json.Marshal/Unmarshal` 的方案也可以。其**重复代码少**，但开销最大

在绝大多数场景中，使用哪种并没有太大的影响。但作为工程师你应当清楚其利弊。以上就是不同的方案**分析概要**，希望对你有所帮助 :)

简单围观一下有趣的 //go: 指令

前言

如果你平时有翻看源码的习惯，你肯定会发现。咦，怎么有的方法上面总是写着 `//go:` 这类指令呢。他们到底是干嘛用的？

今天我们一同揭开他们的面纱，我将简单给你介绍一下，它们都负责些什么

go:linkname

```
//go:linkname localname importpath.name
```

该指令指示编译器使用 `importpath.name` 作为源代码中声明为 `localname` 的变量或函数的目标文件符号名称。但是由于这个伪指令，可以破坏类型系统和包模块化。因此只有引用了 unsafe 包才可以使用

简单来讲，就是 `importpath.name` 是 `localname` 的符号别名，编译器实际上会调用 `localname` 。但前提是使用了 `unsafe` 包才能使用

案例

time/time.go

```
...
func now() (sec int64, nsec int32, mono int64)
```

runtime/timestub.go

```
import _ "unsafe" // for go:linkname

//go:linkname time_now time.now
func time_now() (sec int64, nsec int32, mono int64) {
    sec, nsec = walltime()
    return sec, nsec, nanotime() - startNano
}
```

在这个案例中可以看到 `time.now` ，它并没有具体的实现。如果你初看可能会懵逼。这时候建议你全局搜索一下源码，你就会发现其实现在 `runtime.time_now` 中

配合先前的用法解释，可得知在 runtime 包中，我们声明了 `time_now` 方法是 `time.now` 的符号别名。并且在文件头引入了 `unsafe` 达成前提条件

go:noescape

```
//go:noescape
```

该指令指定下一个有声明但没有主体（意味着实现有可能不是 Go）的函数，不允许编译器对其做逃逸分析

一般情况下，该指令用于内存分配优化。因为编译器默认会进行逃逸分析，会通过规则判定一个变量是分配到堆上还是栈上。但凡事有意外，一些函数虽然逃逸分析其是存放到堆上。但是对于我们来说，它是特别的。我们就可以使用 `go:noescape` 指令强制要求编译器将其分配到函数栈上

案例

```
// memmove copies n bytes from "from" to "to".
// in memmove_*.s
//go:noescape
func memmove(to, from unsafe.Pointer, n uintptr)
```

我们观察一下这个案例，它满足了该指令的常见特性。如下：

- memmove_*.s：只有声明，没有主体。其主体是由底层汇编实现的
- memmove：函数功能，在栈上处理性能会更好

go:nosplit

```
//go:nosplit
```

该指令指定文件中声明的下一个函数不得包含堆栈溢出检查。简单来讲，就是这个函数跳过堆栈溢出的检查

案例

```
//go:nosplit
func key32(p *uintptr) *uint32 {
    return (*uint32)(unsafe.Pointer(p))
}
```

go:nowritebarrierrec

```
//go:nowritebarrierrec
```

该指令表示编译器遇到写屏障时就会产生一个错误，并且允许递归。也就是这个函数调用的其他函数如果有写屏障也会报错。简单来讲，就是针对写屏障的处理，防止其死循环

案例

```
//go:nowritebarrierrec
func gcFlushBgCredit(scanWork int64) {
    ...
}
```

go:yeswritebarrierrec

```
//go:yeswritebarrierrec
```

该指令与 `go:nowritebarrierrec` 相对，在标注 `go:nowritebarrierrec` 指令的函数上，遇到写屏障会产生错误。而当编译器遇到 `go:yeswritebarrierrec` 指令时将会停止

案例

```
//go:yeswritebarrierrec
func gchelper() {
    ...
}
```

go:noinline

该指令表示该函数禁止进行内联

案例

```
//go:noinline
func unexportedPanicForTesting(b []byte, i int) byte {
    return b[i]
}
```

我们观察一下这个案例，是直接通过索引取值，逻辑比较简单。如果不加上 `go:noinline` 的话，就会出现编译器对其进行内联优化

显然，内联有好有坏。该指令就是提供这一特殊处理

go:norace

```
//go:norace
```

该指令表示禁止进行竞态检测。而另外一种常见的形式就是在启动时执行 `go run -race` ，能够检测应用程序中是否存在双向的数据竞争。非常有用

案例

```
//go:norace
func forkAndExecInChild(argv0 *byte, argv, envv []*byte, chroot, dir *byte, attr *ProcAttr, sys *SysProcAttr, pipe int) (pid int, err Errno) {
    ...
}
```

go:notinheap

```
//go:notinheap
```

该指令常用于类型声明，它表示这个类型不允许从 GC 堆上进行申请内存。在运行时中常用其来做较低层次的内部结构，避免调度器和内存分配中的写屏障。能够提高性能

案例

```
// notInHeap is off-heap memory allocated by a lower-level allocator
// like sysAlloc or persistentAlloc.
//
// In general, it's better to use real types marked as go:notinheap,
// but this serves as a generic type for situations where that isn't
// possible (like in the allocators).
//
//go:notinheap
type notInHeap struct{}
```

总结

在本文我们简单介绍了一些常见的指令集，我建议仅供了解。一般我们是用不到的，因为你的瓶颈可能更多的在自身应用上

但是了解这一些，对你了解底层源码和运行机制会更有帮助。如果想再深入些，可阅读我给出的参考链接 ： ）

参考

- HACKING
- Command compile

我要在栈上。不，你应该在堆上

我们在写代码的时候，有时候会想这个变量到底分配到哪里了？这时候可能会有人说，在栈上，在堆上。信我准没错...

但从结果上来讲你还是一知半解，这可不行，万一被人懵了呢。今天我们一起来深挖下 Go 在这块的奥妙，自己动手丰衣足食

问题

```
type User struct {
    ID     int64
    Name   string
    Avatar string
}

func GetUserInfo() *User {
    return &User{ID: 13746731, Name: "EDDYCJY", Avatar: "https://avatars0.github
usercontent.com/u/13746731"}
}

func main() {
    _ = GetUserInfo()
}
```

开局就是一把问号，带着问题进行学习。请问 main 调用 `GetUserInfo` 后返回的 `&User{...}` 。这个变量是分配到栈上了呢，还是分配到堆上了？

什么是堆/栈

在这里并不打算详细介绍堆栈，仅简单介绍本文所需的基础知识。如下：

- 堆（Heap）：一般来讲是人为手动进行管理，手动申请、分配、释放。一般所涉及的内存大小并不定，一般会存放较大的对象。另外其分配相对慢，涉及到的指令动作也相对多
- 栈（Stack）：由编译器进行管理，自动申请、分配、释放。一般不会太大，我们常见的函数参数（不同平台允许存放的数量不同），局部变量等等都会存放在栈上

今天我们介绍的 Go 语言，它的堆栈分配是通过 Compiler 进行分析，GC 去管理的，而对其的分析选择动作就是今天探讨的重点

什么是逃逸分析

在编译程序优化理论中，逃逸分析是一种确定指针动态范围的方法，简单来说就是分析在程序的哪些地方可以访问到该指针

通俗地讲，逃逸分析就是确定一个变量要放堆上还是栈上，规则如下：

1. 是否有在其他地方（非局部）被引用。只要**有可能**被引用了，那么它**一定**分配到堆上。否则分配到栈上
2. 即使没有被外部引用，但对象过大，无法存放在栈区上。依然有可能分配到堆上

对此你可以理解为，逃逸分析是编译器用于决定变量分配到堆上还是栈上的一种行为

在什么阶段确立逃逸

在编译阶段确立逃逸，注意并不是在运行时

为什么需要逃逸

这个问题我们可以反过来想，如果变量都分配到堆上了会出现什么事情？例如：

- 垃圾回收（GC）的压力不断增大
- 申请、分配、回收内存的系统开销增大（相对于栈）
- 动态分配产生一定量的内存碎片

其实总的来说，就是频繁申请、分配堆内存是有一定“代价”的。会影响应用程序运行的效率，间接影响到整体系统。因此“按需分配”最大限度的灵活利用资源，才是正确的治理之道。这就是为什么需要逃逸分析的原因，你觉得呢？

怎么确定是否逃逸

第一，通过编译器命令，就可以看到详细的逃逸分析过程。而指令集 `-gcflags` 用于将标识参数传递给 Go 编译器，涉及如下：

- `-m` 会打印出逃逸分析的优化策略，实际上最多总共可以用 4 个 `-m` ，但是信息量较大，一般用 1 个就可以了
- `-l` 会禁用函数内联，在这里禁用掉 inline 能更好的观察逃逸情况，减少干扰

```
$ go build -gcflags '-m -l' main.go
```

第二，通过反编译命令查看

```
$ go tool compile -S main.go
```

注：可以通过 `go tool compile -help` 查看所有允许传递给编译器的标识参数

逃逸案例

案例一：指针

第一个案例是一开始抛出的问题，现在你再看看，想想，如下：

```
type User struct {
    ID     int64
    Name   string
    Avatar string
}

func GetUserInfo() *User {
    return &User{ID: 13746731, Name: "EDDYCJY", Avatar: "https://avatars0.github
usercontent.com/u/13746731"}
}

func main() {
    _ = GetUserInfo()
}
```

执行命令观察一下，如下：

```
$ go build -gcflags '-m -l' main.go
# command-line-arguments
./main.go:10:54: &User literal escapes to heap
```

通过查看分析结果，可得知 `&User` 逃到了堆里，也就是分配到堆上了。这是不是有问题啊...再看看汇编代码确定一下，如下：

```
$ go tool compile -S main.go
"".GetUserInfo STEXT size=190 args=0x8 locals=0x18
    0x0000 00000 (main.go:9)    TEXT    "".GetUserInfo(SB), $24-8
    ...
    0x0028 00040 (main.go:10)    MOVQ    AX, (SP)
    0x002c 00044 (main.go:10)    CALL    runtime.newobject(SB)
```

```
    0x0031 00049 (main.go:10)    PCDATA    $2, $1
    0x0031 00049 (main.go:10)    MOVQ    8(SP), AX
    0x0036 00054 (main.go:10)    MOVQ    $13746731, (AX)
    0x003d 00061 (main.go:10)    MOVQ    $7, 16(AX)
    0x0045 00069 (main.go:10)    PCDATA    $2, $-2
    0x0045 00069 (main.go:10)    PCDATA    $0, $-2
    0x0045 00069 (main.go:10)    CMPL    runtime.writeBarrier(SB), $0
    0x004c 00076 (main.go:10)    JNE    156
    0x004e 00078 (main.go:10)    LEAQ    go.string."EDDYCJY"(SB), CX
    ...
```

我们将目光集中到 CALL 指令，发现其执行了 `runtime.newobject` 方法，也就是确实是分配到了堆上。这是为什么呢？

分析结果

这是因为 `GetUserInfo()` 返回的是指针对象，引用被返回到了方法之外了。因此编译器会把该对象分配到堆上，而不是栈上。否则方法结束之后，局部变量就被回收了，岂不是翻车。所以最终分配到堆上是理所当然的

再想想

那你可能会想，那就是所有指针对象，都应该在堆上？并不。如下：

```
func main() {
    str := new(string)
    *str = "EDDYCJY"
}
```

你想想这个对象会分配到哪里？如下：

```
$ go build -gcflags '-m -l' main.go
# command-line-arguments
./main.go:4:12: main new(string) does not escape
```

显然，该对象分配到栈上了。很核心的一点就是它有没有被作用域之外所引用，而这里作用域仍然保留在 `main` 中，因此它没有发生逃逸

案例二：未确定类型

```
func main() {
    str := new(string)
    *str = "EDDYCJY"
```

```
    fmt.Println(str)
}
```

执行命令观察一下，如下：

```
$ go build -gcflags '-m -l' main.go
# command-line-arguments
./main.go:9:13: str escapes to heap
./main.go:6:12: new(string) escapes to heap
./main.go:9:13: main ... argument does not escape
```

通过查看分析结果，可得知 `str` 变量逃到了堆上，也就是该对象在堆上分配。但上个案例时它还在栈上，我们也就 `fmt` 输出了它而已。这...到底发生了什么事？

分析结果

相对案例一，案例二只加了一行代码 `fmt.Println(str)` ，问题肯定出在它身上。其原型：

```
func Println(a ...interface{}) (n int, err error)
```

通过对其分析，可得知当形参为 `interface` 类型时，在编译阶段编译器无法确定其具体的类型。因此会产生逃逸，最终分配到堆上

如果你有兴趣追源码的话，可以看下内部的 `reflect.TypeOf(arg).Kind()` 语句，其会造成堆逃逸，而表象就是 `interface` 类型会导致该对象分配到堆上

案例三、泄露参数

```
type User struct {
    ID     int64
    Name   string
    Avatar string
}

func GetUserInfo(u *User) *User {
    return u
}

func main() {
    _ = GetUserInfo(&User{ID: 13746731, Name: "EDDYCJY", Avatar: "https://avatar
```

```
s0.githubusercontent.com/u/13746731"})
}
```

执行命令观察一下，如下：

```
$ go build -gcflags '-m -l' main.go
# command-line-arguments
./main.go:9:18: leaking param: u to result ~r1 level=0
./main.go:14:63: main &User literal does not escape
```

我们注意到 `leaking param` 的表述，它说明了变量 `u` 是一个泄露参数。结合代码可得知其传给 `GetUserInfo` 方法后，没有做任何引用之类的涉及变量的动作，直接就把这个变量返回出去了。因此这个变量实际上并没有逃逸，它的作用域还在 `main()` 之中，所以分配在栈上

再想想

那你再想想怎么样才能让它分配到堆上？结合案例一，举一反三。修改如下：

```
type User struct {
    ID     int64
    Name   string
    Avatar string
}

func GetUserInfo(u User) *User {
    return &u
}

func main() {
    _ = GetUserInfo(User{ID: 13746731, Name: "EDDYCJY", Avatar: "https://avatars
0.githubusercontent.com/u/13746731"})
}
```

执行命令观察一下，如下：

```
$ go build -gcflags '-m -l' main.go
# command-line-arguments
./main.go:10:9: &u escapes to heap
./main.go:9:18: moved to heap: u
```

只要一小改，它就考虑会被外部所引用，因此妥妥的分配到堆上了

总结

在本文我给你介绍了逃逸分析的概念和规则，并列举了一些例子加深理解。但实际肯定远远不止这些案例，你需要做到的是掌握方法，遇到再看就好了。除此之外你还需要注意：

- 静态分配到栈上，性能一定比动态分配到堆上好
- 底层分配到堆，还是栈。实际上对你来说是透明的，不需要过度关心
- 每个 Go 版本的逃逸分析都会有所不同（会改变，会优化）
- 直接通过 `go build -gcflags '-m -l'` 就可以看到逃逸分析的过程和结果
- 到处都用指针传递并不一定是最好的，要用对

之前就有想过要不要写“逃逸分析”相关的文章，直到最近看到在夜读里有人问，还是有写的必要。对于这块的知识点。我的建议是适当了解，但没必要硬记。靠基础知识点加命令调试观察就好了。像是曹大之前讲的“你琢磨半天逃逸分析，一压测，瓶颈在锁上”，完全没必要过度在意...

参考

- Golang escape analysis
- FAQ
- 逃逸分析

Go1.12 defer 会有性能损耗，尽量不要用？

上个月在 @polaris @轩脉刃 的全栈技术群里看到一个小伙伴问**“说 defer 在栈退出时执行，会有性能损耗，尽量不要用，这个怎么解？”**。

恰好前段时间写了一篇 《深入理解 Go defer》 去详细剖析 `defer` 关键字。那么这一次简单结合前文对这个问题进行探讨一波，希望对你有所帮助，但在此之前希望你花几分钟，自己思考一下答案，再继续往下看。

测试

```
func DoDefer(key, value string) {
    defer func(key, value string) {
        _ = key + value
    }(key, value)
}

func DoNotDefer(key, value string) {
    _ = key + value
}
```

基准测试：

```
func BenchmarkDoDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        DoDefer("煎鱼", "https://github.com/EDDYCJY/blog")
    }
}

func BenchmarkDoNotDefer(b *testing.B) {
    for i := 0; i < b.N; i++ {
        DoNotDefer("煎鱼", "https://github.com/EDDYCJY/blog")
    }
}
```

输出结果：

```
$ go test -bench=. -benchmem -run=none
goos: darwin
goarch: amd64
pkg: github.com/EDDYCJY/awesomeDefer
BenchmarkDoDefer-4          20000000               91.4 ns/op            48 B/op
```

```
1 allocs/op
BenchmarkDoNotDefer-4        30000000                41.6 ns/op            48 B/op
1 allocs/op
PASS
ok      github.com/EDDYCJY/awesomeDefer 3.234s
```

从结果上来，使用 `defer` 后的函数开销确实比没使用高了不少，这损耗用到哪里去了呢？

想一下

```
$ go tool compile -S main.go
"".main STEXT size=163 args=0x0 locals=0x40
    ...
    0x0059 00089 (main.go:6)    MOVQ    AX, 16(SP)
    0x005e 00094 (main.go:6)    MOVQ    $1, 24(SP)
    0x0067 00103 (main.go:6)    MOVQ    $1, 32(SP)
    0x0070 00112 (main.go:6)    CALL    runtime.deferproc(SB)
    0x0075 00117 (main.go:6)    TESTL   AX, AX
    0x0077 00119 (main.go:6)    JNE     137
    0x0079 00121 (main.go:7)    XCHGL   AX, AX
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
    0x0084 00132 (main.go:7)    ADDQ    $64, SP
    0x0088 00136 (main.go:7)    RET
    0x0089 00137 (main.go:6)    XCHGL   AX, AX
    0x008a 00138 (main.go:6)    CALL    runtime.deferreturn(SB)
    0x008f 00143 (main.go:6)    MOVQ    56(SP), BP
    0x0094 00148 (main.go:6)    ADDQ    $64, SP
    0x0098 00152 (main.go:6)    RET
    ...
```

我们在前文提到 `defer` 关键字其实涉及了一系列的连锁调用，内部 `runtime` 函数的调用就至少多了三步，分别是 `runtime.deferproc` 一次和 `runtime.deferreturn` 两次。

而这还只是在运行时的显式动作，另外编译器做的事也不少，例如：

- 在 `deferproc` 阶段（注册延迟调用），还得获取/传入目标函数地址、函数参数等等。
- 在 `deferreturn` 阶段，需要在函数调用结尾处插入该方法的调用，同时若有被 `defer` 的函数，还需要使用 `runtime•jmpdefer` 进行跳转以便于后续调用。

这一些动作途中还要涉及最小单元 `_defer` 的获取/生成， `defer` 和 `recover` 链表的逻辑处理和消耗等动作。

Q&A

最后讨论的时候有提到**“问题指的是本来就是用来执行 close() 一些操作的，然后说尽量不能用，例子就把 defer db.close() 前面的 defer 删去了”**这个疑问。

这是一个比较类似“教科书”式的说法，在一些入门教程中会潜移默化的告诉你在资源控制后加个 `defer` 延迟关闭一下。例如：

```
resp, err := http.Get(...)
if err != nil {
    return err
}
defer resp.Body.Close()
```

但是一定得这么写吗？其实并不，很多人给出的理由都是“怕你忘记”这种说辞，这没有毛病。但需要认清场景，假设我的应用场景如下：

```
resp, err := http.Get(...)
if err != nil {
    return err
}
defer resp.Body.Close()
// do something
time.Sleep(time.Second * 60)
```

嗯，一个请求当然没问题，流量、并发一下子大了呢，那可能就是个灾难了。你想想为什么？从常见的 `defer` + `close` 的使用组合来讲，用之前建议先看清楚应用场景，在保证无异常的情况下确保尽早关闭才是首选。如果只是小范围调用很快就返回的话，偷个懒直接一套组合拳出去也未尝不可。

结论

一个 `defer` 关键字实际上包含了不少的动作和处理，和你单纯调用一个函数一条指令是没法比的。而与对照物相比，它确确实实是有性能损耗，目前延迟调用的全部开销大约在 50ns，但 `defer` 所提供的作用远远大于此，你从全局来看，它的损耗非常小，并且官方还不断地在优化中。

因此，对于“Go defer 会有性能损耗，尽量不能用？”这个问题，我认为**该用就用，应该及时关闭就不要延迟，在 hot paths 用时一定要想清楚场景**。

补充

最后补充上柴大的回复：**“不是性能问题，defer 最大的功能是 Panic 后依然有效。如果没有 defer，Panic 后就会导致 unlock 丢失，从而导致死锁了”**，非常经典。

从实践到原理，带你参透 gRPC

最近 Go1.13 终于发布了，其中一个值得关注的特性就是 **defer 在大部分的场景下性能提升了 30%**，但是官方并没有具体写是怎么提升的，这让大家非常的疑惑。而我因为之前写过《深入理解 Go defer》 和 《Go defer 会有性能损耗，尽量不要用？》 这类文章，因此我挺感兴趣它是做了什么改变才能得到这样子的结果，所以今天和大家一起探索其中奥妙。

一、测试

Go1.12

```
$ go test -bench=. -benchmem -run=none
goos: darwin
goarch: amd64
pkg: github.com/EDDYCJY/awesomeDefer
BenchmarkDoDefer-4          20000000            91.4 ns/op          48 B/op
1 allocs/op
BenchmarkDoNotDefer-4       30000000            41.6 ns/op          48 B/op
1 allocs/op
PASS
ok      github.com/EDDYCJY/awesomeDefer    3.234s
```

Go1.13

```
$ go test -bench=. -benchmem -run=none
goos: darwin
goarch: amd64
pkg: github.com/EDDYCJY/awesomeDefer
BenchmarkDoDefer-4          15986062            74.7 ns/op          48 B/op
1 allocs/op
BenchmarkDoNotDefer-4       29231842            40.3 ns/op          48 B/op
1 allocs/op
PASS
ok      github.com/EDDYCJY/awesomeDefer    3.444s
```

在开场，我先以不标准的测试基准验证了先前的测试用例，确确实实在这两个版本中，`defer` 的性能得到了提高，但是看上去似乎不是百分百提高 30 %。

二、看一下

之前（Go1.12）

```
    0x0070 00112 (main.go:6)    CALL    runtime.deferproc(SB)
    0x0075 00117 (main.go:6)    TESTL   AX, AX
    0x0077 00119 (main.go:6)    JNE     137
    0x0079 00121 (main.go:7)    XCHGL   AX, AX
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
```

现在（Go1.13）

```
    0x006e 00110 (main.go:4)    MOVQ    AX, (SP)
    0x0072 00114 (main.go:4)    CALL    runtime.deferprocStack(SB)
    0x0077 00119 (main.go:4)    TESTL   AX, AX
    0x0079 00121 (main.go:4)    JNE     139
    0x007b 00123 (main.go:7)    XCHGL   AX, AX
    0x007c 00124 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x0081 00129 (main.go:7)    MOVQ    112(SP), BP
```

从汇编的角度来看，像是 `runtime.deferproc` 改成了 `runtime.deferprocStack` 调用，难道是做了什么优化，我们**抱着疑问**继续看下去。

三、观察源码

_defer

```
type _defer struct {
    siz     int32
    siz     int32 // includes both arguments and results
    started bool
    heap    bool
    sp      uintptr // sp at time of defer
    pc      uintptr
    fn      *funcval
    ...
```

相较于以前的版本，最小单元的 `_defer` 结构体主要是新增了 `heap` 字段，用于标识这个 `_defer` 是在堆上，还是在栈上进行分配，其余字段并没有明确变更，那我们可以把聚焦点放在 `defer` 的堆栈分配上了，看看是做了什么事。

deferprocStack

```
func deferprocStack(d *_defer) {
    gp := getg()
    if gp.m.curg != gp {
        throw("defer on system stack")
    }

    d.started = false
    d.heap = false
    d.sp = getcallersp()
    d.pc = getcallerpc()

    *(*uintptr)(unsafe.Pointer(&d._panic)) = 0
    *(*uintptr)(unsafe.Pointer(&d.link)) = uintptr(unsafe.Pointer(gp._defer))
    *(*uintptr)(unsafe.Pointer(&gp._defer)) = uintptr(unsafe.Pointer(d))

    return0()
}
```

这一块代码挺常规的，主要是获取调用 `defer` 函数的函数栈指针、传入函数的参数具体地址以及PC（程序计数器），这块在前文 《深入理解 Go defer》 有详细介绍过，这里就不再赘述了。

那这个 `deferprocStack` 特殊在哪呢，我们可以看到它把 `d.heap` 设置为了 `false`，也就是代表 `deferprocStack` 方法是针对将 `_defer` 分配在栈上的应用场景的。

deferproc

那么问题来了，它又在哪里处理分配到堆上的应用场景呢？

```
func newdefer(siz int32) *_defer {
    ...
    d.heap = true
    d.link = gp._defer
    gp._defer = d
    return d
}
```

那么 `newdefer` 是在哪里调用的呢，如下：

```
func deferproc(siz int32, fn *funcval) { // arguments of fn follow fn
    ...
    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
```

```
    callerpc := getcallerpc()

    d := newdefer(siz)
    ...
}
```

非常明确，先前的版本中调用的 `deferproc` 方法，现在被用于对应分配到堆上的场景了。

小结

- 第一点：可以确定的是 `deferproc` 并没有被去掉，而是流程被优化了。
- 第二点：编译器会根据应用场景去选择使用 `deferproc` 还是 `deferprocStack` 方法，他们分别是针对分配在堆上和栈上的使用场景。

四、编译器如何选择

esc

```
// src/cmd/compile/internal/gc/esc.go
case ODEFER:
    if e.loopdepth == 1 { // top level
        n.Esc = EscNever // force stack allocation of defer record (see ssa.go)
        break
    }
```

ssa

```
// src/cmd/compile/internal/gc/ssa.go
case ODEFER:
    d := callDefer
    if n.Esc == EscNever {
        d = callDeferStack
    }
    s.call(n.Left, d)
```

小结

这块结合来看，核心就是当 `e.loopdepth == 1` 时，会将逃逸分析结果 `n.Esc` 设置为 `EscNever` ，也就是将 `_defer` 分配到栈上，那这个 `e.loopdepth` 到底又是何方神圣呢，我们再详细看看代码，如下：

```
// src/cmd/compile/internal/gc/esc.go
type NodeEscState struct {
    Curfn             *Node
    Flowsrc           []EscStep
    Retval            Nodes
    Loopdepth         int32
    Level             Level
    Walkgen           uint32
    Maxextraloopdepth int32
}
```

这里重点查看 `Loopdepth` 字段，目前它共有三个值标识，分别是:

- -1：全局。
- 0：返回变量。
- 1：顶级函数，又或是内部函数的不断增长值。

这个读起来有点绕，结合我们上述 `e.loopdepth == 1` 的表述来看，也就是当 `defer func` 是顶级函数时，将会分配到栈上。但是若在 `defer func` 外层出现显式的迭代循环，又或是出现隐式迭代，将会分配到堆上。其实深层表示的还是迭代深度的意思，我们可以来证实一下刚刚说的方向，显式迭代的代码如下：

```
func main() {
    for p := 0; p < 10; p++ {
        defer func() {
            for i := 0; i < 20; i++ {
                log.Println("EDDYCJY")
            }
        }()
    }
}
```

查看汇编情况：

```
$ go tool compile -S main.go
"".main STEXT size=122 args=0x0 locals=0x20
    0x0000 00000 (main.go:15)    TEXT    "".main(SB), ABIInternal, $32-0
    ...
    0x0048 00072 (main.go:17)    CALL    runtime.deferproc(SB)
    0x004d 00077 (main.go:17)    TESTL    AX, AX
    0x004f 00079 (main.go:17)    JNE    83
    0x0051 00081 (main.go:17)    JMP    33
    0x0053 00083 (main.go:17)    XCHGL    AX, AX
```

```
    0x0054 00084 (main.go:17)    CALL    runtime.deferreturn(SB)
    ...
```

显然，最终 `defer` 调用的是 `runtime.deferproc` 方法，也就是分配到堆上了，没毛病。而隐式迭代的话，你可以借助 `goto` 语句去实现这个功能，再自己验证一遍，这里就不再赘述了。

总结

从分析的结果上来看，官方说明的 Go1.13 defer 性能提高 30%，主要来源于其延迟对象的堆栈分配规则的改变，措施是由编译器通过对 `defer` 的 `for-loop` 迭代深度进行分析，如果 `loopdepth` 为 1，则设置逃逸分析的结果，将分配到栈上，否则分配到堆上。

的确，我个人觉得对大部分的使用场景来讲，是优化了不少，也解决了一些人吐槽 `defer` 性能“差”的问题。另外，我想从 Go1.13 起，你也需要稍微了解一下它这块的机制，别随随便便就来个狂野版嵌套迭代 `defer`，可能没法效能最大化。

如果你还想了解更多细节，可以看看 `defer` 这块的的提交内容，官方的测试用例也包含在里面。

Go1.13 defer 的性能是如何提高的

最近 Go1.13 终于发布了，其中一个值得关注的特性就是 **defer 在大部分的场景下性能提升了 30%**，但是官方并没有具体写是怎么提升的，这让大家非常的疑惑。而我因为之前写过《深入理解 Go defer》 和 《Go defer 会有性能损耗，尽量不要用？》 这类文章，因此我挺感兴趣它是做了什么改变才能得到这样子的结果，所以今天和大家一起探索其中奥妙。

一、测试

Go1.12

```
$ go test -bench=. -benchmem -run=none
goos: darwin
goarch: amd64
pkg: github.com/EDDYCJY/awesomeDefer
BenchmarkDoDefer-4          20000000            91.4 ns/op          48 B/op
1 allocs/op
BenchmarkDoNotDefer-4       30000000            41.6 ns/op          48 B/op
1 allocs/op
PASS
ok      github.com/EDDYCJY/awesomeDefer    3.234s
```

Go1.13

```
$ go test -bench=. -benchmem -run=none
goos: darwin
goarch: amd64
pkg: github.com/EDDYCJY/awesomeDefer
BenchmarkDoDefer-4          15986062            74.7 ns/op          48 B/op
1 allocs/op
BenchmarkDoNotDefer-4       29231842            40.3 ns/op          48 B/op
1 allocs/op
PASS
ok      github.com/EDDYCJY/awesomeDefer    3.444s
```

在开场，我先以不标准的测试基准验证了先前的测试用例，确确实实在这两个版本中，`defer` 的性能得到了提高，但是看上去似乎不是百分百提高 30 %。

二、看一下

之前（Go1.12）

```
    0x0070 00112 (main.go:6)    CALL    runtime.deferproc(SB)
    0x0075 00117 (main.go:6)    TESTL   AX, AX
    0x0077 00119 (main.go:6)    JNE     137
    0x0079 00121 (main.go:7)    XCHGL   AX, AX
    0x007a 00122 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x007f 00127 (main.go:7)    MOVQ    56(SP), BP
```

现在（Go1.13）

```
    0x006e 00110 (main.go:4)    MOVQ    AX, (SP)
    0x0072 00114 (main.go:4)    CALL    runtime.deferprocStack(SB)
    0x0077 00119 (main.go:4)    TESTL   AX, AX
    0x0079 00121 (main.go:4)    JNE     139
    0x007b 00123 (main.go:7)    XCHGL   AX, AX
    0x007c 00124 (main.go:7)    CALL    runtime.deferreturn(SB)
    0x0081 00129 (main.go:7)    MOVQ    112(SP), BP
```

从汇编的角度来看，像是 `runtime.deferproc` 改成了 `runtime.deferprocStack` 调用，难道是做了什么优化，我们**抱着疑问**继续看下去。

三、观察源码

_defer

```
type _defer struct {
    siz     int32
    siz     int32 // includes both arguments and results
    started bool
    heap    bool
    sp      uintptr // sp at time of defer
    pc      uintptr
    fn      *funcval
    ...
```

相较于以前的版本，最小单元的 `_defer` 结构体主要是新增了 `heap` 字段，用于标识这个 `_defer` 是在堆上，还是在栈上进行分配，其余字段并没有明确变更，那我们可以把聚焦点放在 `defer` 的堆栈分配上了，看看是做了什么事。

deferprocStack

```
func deferprocStack(d *_defer) {
    gp := getg()
    if gp.m.curg != gp {
        throw("defer on system stack")
    }

    d.started = false
    d.heap = false
    d.sp = getcallersp()
    d.pc = getcallerpc()

    *(*uintptr)(unsafe.Pointer(&d._panic)) = 0
    *(*uintptr)(unsafe.Pointer(&d.link)) = uintptr(unsafe.Pointer(gp._defer))
    *(*uintptr)(unsafe.Pointer(&gp._defer)) = uintptr(unsafe.Pointer(d))

    return0()
}
```

这一块代码挺常规的，主要是获取调用 `defer` 函数的函数栈指针、传入函数的参数具体地址以及PC（程序计数器），这块在前文 《深入理解 Go defer》 有详细介绍过，这里就不再赘述了。

那这个 `deferprocStack` 特殊在哪呢，我们可以看到它把 `d.heap` 设置为了 `false`，也就是代表 `deferprocStack` 方法是针对将 `_defer` 分配在栈上的应用场景的。

deferproc

那么问题来了，它又在哪里处理分配到堆上的应用场景呢？

```
func newdefer(siz int32) *_defer {
    ...
    d.heap = true
    d.link = gp._defer
    gp._defer = d
    return d
}
```

那么 `newdefer` 是在哪里调用的呢，如下：

```
func deferproc(siz int32, fn *funcval) { // arguments of fn follow fn
    ...
    sp := getcallersp()
    argp := uintptr(unsafe.Pointer(&fn)) + unsafe.Sizeof(fn)
```

```
    callerpc := getcallerpc()

    d := newdefer(siz)
    ...
}
```

非常明确，先前的版本中调用的 `deferproc` 方法，现在被用于对应分配到堆上的场景了。

小结

- 第一点：可以确定的是 `deferproc` 并没有被去掉，而是流程被优化了。
- 第二点：编译器会根据应用场景去选择使用 `deferproc` 还是 `deferprocStack` 方法，他们分别是针对分配在堆上和栈上的使用场景。

四、编译器如何选择

esc

```
// src/cmd/compile/internal/gc/esc.go
case ODEFER:
    if e.loopdepth == 1 { // top level
        n.Esc = EscNever // force stack allocation of defer record (see ssa.go)
        break
    }
```

ssa

```
// src/cmd/compile/internal/gc/ssa.go
case ODEFER:
    d := callDefer
    if n.Esc == EscNever {
        d = callDeferStack
    }
    s.call(n.Left, d)
```

小结

这块结合来看，核心就是当 `e.loopdepth == 1` 时，会将逃逸分析结果 `n.Esc` 设置为 `EscNever` ，也就是将 `_defer` 分配到栈上，那这个 `e.loopdepth` 到底又是何方神圣呢，我们再详细看看代码，如下：

```
// src/cmd/compile/internal/gc/esc.go
type NodeEscState struct {
    Curfn             *Node
    Flowsrc           []EscStep
    Retval            Nodes
    Loopdepth         int32
    Level             Level
    Walkgen           uint32
    Maxextraloopdepth int32
}
```

这里重点查看 `Loopdepth` 字段，目前它共有三个值标识，分别是:

- -1：全局。
- 0：返回变量。
- 1：顶级函数，又或是内部函数的不断增长值。

这个读起来有点绕，结合我们上述 `e.loopdepth == 1` 的表述来看，也就是当 `defer func` 是顶级函数时，将会分配到栈上。但是若在 `defer func` 外层出现显式的迭代循环，又或是出现隐式迭代，将会分配到堆上。其实深层表示的还是迭代深度的意思，我们可以来证实一下刚刚说的方向，显式迭代的代码如下：

```
func main() {
    for p := 0; p < 10; p++ {
        defer func() {
            for i := 0; i < 20; i++ {
                log.Println("EDDYCJY")
            }
        }()
    }
}
```

查看汇编情况：

```
$ go tool compile -S main.go
"".main STEXT size=122 args=0x0 locals=0x20
    0x0000 00000 (main.go:15)    TEXT    "".main(SB), ABIInternal, $32-0
    ...
    0x0048 00072 (main.go:17)    CALL    runtime.deferproc(SB)
    0x004d 00077 (main.go:17)    TESTL    AX, AX
    0x004f 00079 (main.go:17)    JNE    83
    0x0051 00081 (main.go:17)    JMP    33
    0x0053 00083 (main.go:17)    XCHGL    AX, AX
```

```
    0x0054 00084 (main.go:17)    CALL    runtime.deferreturn(SB)
    ...
```

显然，最终 `defer` 调用的是 `runtime.deferproc` 方法，也就是分配到堆上了，没毛病。而隐式迭代的话，你可以借助 `goto` 语句去实现这个功能，再自己验证一遍，这里就不再赘述了。

总结

从分析的结果上来看，官方说明的 Go1.13 defer 性能提高 30%，主要来源于其延迟对象的堆栈分配规则的改变，措施是由编译器通过对 `defer` 的 `for-loop` 迭代深度进行分析，如果 `loopdepth` 为 1，则设置逃逸分析的结果，将分配到栈上，否则分配到堆上。

的确，我个人觉得对大部分的使用场景来讲，是优化了不少，也解决了一些人吐槽 `defer` 性能“差”的问题。另外，我想从 Go1.13 起，你也需要稍微了解一下它这块的机制，别随随便便就来个狂野版嵌套迭代 `defer`，可能没法效能最大化。

如果你还想了解更多细节，可以看看 `defer` 这块的的提交内容，官方的测试用例也包含在里面。

Go 应用内存占用太多，让排查？（VSZ篇）

前段时间，某同学说某服务的容器因为超出内存限制，不断地重启，问我们是不是有内存泄露，赶紧排查，然后解决掉，省的出问题。我们大为震惊，赶紧查看监控+报警系统和性能分析，发现应用指标压根就不高，不像有泄露的样子。

那么问题是出在哪里了呢，我们进入某个容器里查看了 `top` 的系统指标，结果如下：

```
PID       VSZ    RSS   ... COMMAND
67459     2007m  136m  ... ./eddycjy-server
```

从结果上来看，也没什么大开销的东西，主要就一个 Go 进程，一看，某同学就说 VSZ 那么高，而某云上的容器内存指标居然恰好和 VSZ 的值相接近，因此某同学就怀疑是不是 VSZ 所导致的，觉得存在一定的关联关系。

而从最终的结论上来讲，上述的表述是不全对的，那么在今天，本篇文章将**主要围绕 Go 进程的 VSZ 来进行剖析**，看看到底它为什么那么“高”，而在正式开始分析前，第一节为前置的补充知识，大家可按顺序阅读。

基础知识

什么是 VSZ

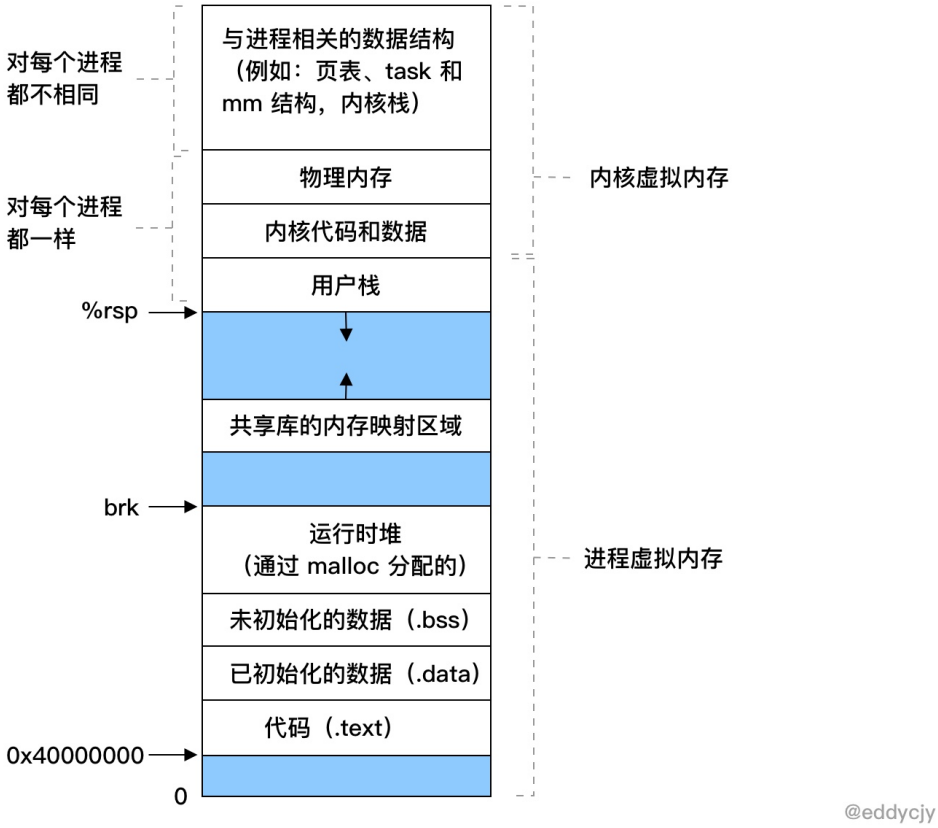
VSZ 是该进程所能使用的虚拟内存总大小，它包括进程可以访问的所有内存，其中包括了被换出的内存（Swap）、已分配但未使用的内存以及来自共享库的内存。

为什么要虚拟内存

在前面我们有了解到 VSZ 其实就是该进程的虚拟内存总大小，那**如果我们想了解 VSZ 的话，那我们得先了解“为什么要虚拟内存？”**。

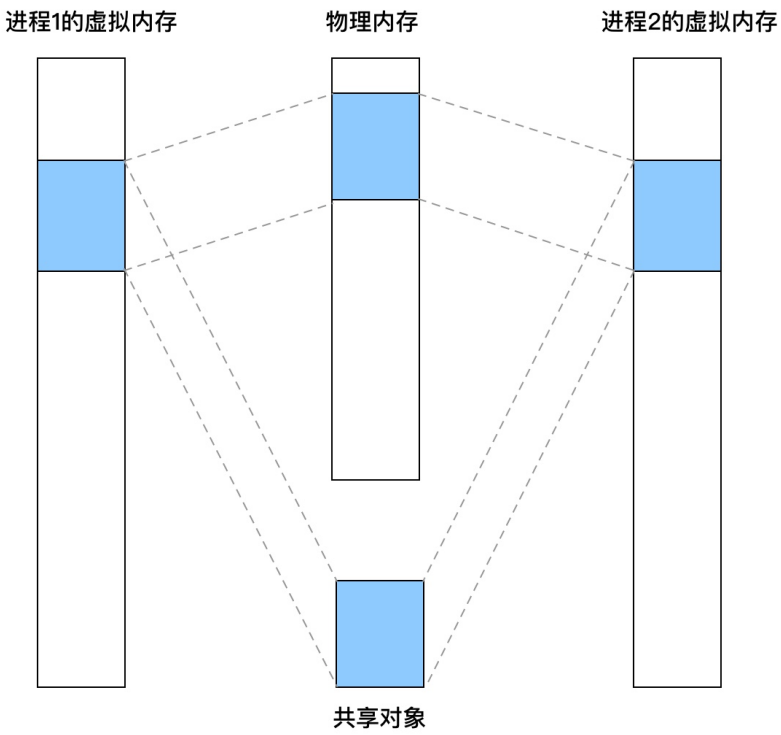
本质上来讲，在一个系统中的进程是与其他进程共享 CPU 和主存资源的，而在现代的操作系统中，多进程的使用非常的常见，那么如果太多的进程需要太多的内存，那么在没有虚拟内存的情况下，物理内存很可能会不够用，就会导致其中有些任务无法运行，更甚至会出现一些很奇怪的现象，例如“某一个进程不小心写了另一个进程使用的内存”，就会造成内存破坏，因此虚拟内存是非常重要的一个媒介。

虚拟内存包含了什么



而虚拟内存，又分为内核虚拟内存和进程虚拟内存，每一个进程的虚拟内存都是独立的， 呈现如上图所示。

这里也补充说明一下，在内核虚拟内存中，是包含了内核中的代码和数据结构，而内核虚拟内存中的某些区域会被映射到所有进程共享的物理页面中去，因此你会看到 ”内核虚拟内存“ 中实际上是包含了 ”物理内存“ 的，它们两者存在映射关系。而在应用场景上来讲，每个进程也会去共享内核的代码和全局数据结构，因此就会被映射到所有进程的物理页面中去。



虚拟内存的重要能力

为了更有效地管理内存并且减少出错，现代系统提供了一种对主存的抽象概念，也就是今天的主角，叫做虚拟内存（VM），虚拟内存是硬件异常、硬件地址翻译、主存、磁盘文件和内核软件交互的地方，它为每个进程提供了一个大的、一致的和私有的地址空间，虚拟内存提供了三个重要的能力：

1. 它将主存看成是一个存储在磁盘上的地址空间的高速缓存，在主存中只保存活动区域，并根据需要在磁盘和主存之间来回传送数据，通过这种方式，它高效地使用了主存。
2. 它为每个进程提供了一致的地址空间，从而简化了内存管理。
3. 它保护了每个进程的地址空间不被其他进程破坏。

小结

上面发散的可能比较多，简单来讲，对于本文我们重点关注这些知识点，如下：

- 虚拟内存它是有各式各样内存交互的地方，它包含的不仅仅是“自己”，**而在本文中，我们只需要关注 VSZ，也就是进程虚拟内存，它包含了你的代码、数据、堆、栈段和共享库**。
- 虚拟内存作为内存保护的工具，能够保证进程之间的内存空间独立，不受其他进程的影响，因此每一个进程的 VSZ 大小都不一样，互不影响。

- 虚拟内存的存在，系统给各进程分配的内存之和是可以大于实际可用的物理内存的，因此你也会发现你进程的物理内存总是比虚拟内存低的多的多。

排查问题

在了解了基础知识后，我们正式开始排查问题，第一步我们先编写一个测试程序，看看没有什么业务逻辑的 Go 程序，它初始的 VSZ 是怎么样的。

测试

应用代码：

```
func main() {
    r := gin.Default()
    r.GET("/ping", func(c *gin.Context) {
        c.JSON(200, gin.H{
            "message": "pong",
        })
    })
    r.Run(":8001")
}
```

查看进程情况：

```
$ ps aux 67459
USER       PID  %CPU %MEM      VSZ    RSS   ...
eddycjy 67459   0.0  0.0  4297048    960   ...
```

从结果上来看，VSZ 为 4297048K，也就是 4G 左右，咋一眼看过去还是挺吓人的，明明没有什么业务逻辑，但是为什么那么高呢，真是令人感到好奇。

确认有没有泄露

在未知的情况下，我们可以首先看下 `runtime.MemStats` 和 `pprof`，确定应用到底有没有泄露。不过我们这块是演示程序，什么业务逻辑都没有，因此可以确定和应用没有直接关系。

```
# runtime.MemStats
# Alloc = 1298568
# TotalAlloc = 1298568
# Sys = 71893240
# Lookups = 0
```

```
# Mallocs = 10013
# Frees = 834
# HeapAlloc = 1298568
# HeapSys = 66551808
# HeapIdle = 64012288
# HeapInuse = 2539520
# HeapReleased = 64012288
# HeapObjects = 9179
...
```

Go FAQ

接着我第一反应是去翻了 Go FAQ（因为看到过，有印象），其问题为 “Why does my Go process use so much virtual memory?”，回答如下：

> The Go memory allocator reserves a large region of virtual memory as an arena for allocations. This virtual memory is local to the specific Go process; the reservation does not deprive other processes of memory.
>
> To find the amount of actual memory allocated to a Go process, use the Unix top command and consult the RES (Linux) or RSIZE (macOS) columns.

这个 FAQ 是在 2012 年 10 月 提交 的，这么多年了也没有更进一步的说明，再翻了 issues 和 forum，一些关闭掉的 issue 都指向了 FAQ，这显然无法满足我的求知欲，因此我继续往下探索，看看里面到底都摆了些什么。

查看内存映射

在上图中，我们有提到进程虚拟内存，主要包含了你的代码、数据、堆、栈段和共享库，那初步怀疑是不是进程做了什么内存映射，导致了大量的内存空间被保留呢，为了确定这一点，我们通过如下命令去排查：

```
$ vmmap --wide 67459
...
==== Non-writable regions for process 67459
REGION TYPE                        START - END             [ VSIZE  RSDNT  DIRTY
SWAP] PRT/MAX SHRMOD PURGE    REGION DETAIL
__TEXT                 00000001065ff000-000000010667b000 [  496K   492K     0K
0K] r-x/rwx SM=COW          /bin/zsh
__LINKEDIT             0000000106687000-0000000106699000 [   72K    44K     0K
0K] r--/rwx SM=COW          /bin/zsh
MALLOC metadata        000000010669b000-000000010669c000 [    4K     4K     4K
0K] r--/rwx SM=COW          DefaultMallocZone_0x10669b000 zone structure
...
```

```
__TEXT                 00007fff76c31000-00007fff76c5f000 [  184K   168K     0K
0K] r-x/r-x SM=COW          /usr/lib/system/libxpc.dylib
__LINKEDIT             00007fffe7232000-00007ffff32cb000 [192.6M  17.4M     0K
0K] r--/r-- SM=COW          dyld shared cache combined __LINKEDIT
...

==== Writable regions for process 67459
REGION TYPE                      START - END             [ VSIZE  RSDNT  DIRTY
SWAP] PRT/MAX SHRMOD PURGE    REGION DETAIL
__DATA                 000000010667b000-0000000106682000 [   28K    28K    28K
0K] rw-/rwx SM=COW          /bin/zsh
...
__DATA                 0000000106716000-000000010671e000 [   32K    28K    28K
4K] rw-/rwx SM=COW          /usr/lib/zsh/5.3/zsh/zle.so
__DATA                 000000010671e000-000000010671f000 [    4K     4K     4K
0K] rw-/rwx SM=COW          /usr/lib/zsh/5.3/zsh/zle.so
__DATA                 0000000106745000-0000000106747000 [    8K     8K     8K
0K] rw-/rwx SM=COW          /usr/lib/zsh/5.3/zsh/complete.so
__DATA                 000000010675a000-000000010675b000 [    4K     4K     4K
0K] rw-
...
```

这块主要是利用 macOS 的 `vmmap` 命令去查看内存映射情况，这样就可以知道这个进程的内存映射情况，从输出分析来看，**这些关联共享库占用的空间并不大，导致 VSZ 过高的根本原因不在共享库和二进制文件上，但是并没有发现大量保留内存空间的行为，这是一个问题点**。

注：若是 Linux 系统，可使用 `cat /proc/PID/maps` 或 `cat /proc/PID/smaps` 查看。

查看系统调用

既然在内存映射中，我们没有明确的看到保留内存空间的行为，那我们接下来看看该进程的系统调用，确定一下它是否存在内存操作的行为，如下：

```
$ sudo dtruss -a ./awesomeProject
...
 4374/0x206a2:      15620        6        3 mprotect(0x1BC4000, 0x1000, 0x0)
= 0 0
...
 4374/0x206a2:      15781        9        4 sysctl([CTL_HW, 3, 0, 0, 0, 0] (2), 0x7F
FEEFBFFA64, 0x7FFEEFBFFA68, 0x0, 0x0)          = 0 0
 4374/0x206a2:      15783        3        1 sysctl([CTL_HW, 7, 0, 0, 0, 0] (2), 0x7F
FEEFBFFA64, 0x7FFEEFBFFA68, 0x0, 0x0)          = 0 0
 4374/0x206a2:      15899        7        2 mmap(0x0, 0x40000, 0x3, 0x1002, 0xFFFFFF
```

```
FFFFFFFFFF, 0x0)          = 0x4000000 0
 4374/0x206a2:      15930        3        1 mmap(0xC000000000, 0x4000000, 0x0, 0x100
2, 0xFFFFFFFFFFFFFFFF, 0x0)          = 0xC000000000 0
 4374/0x206a2:      15934        4        2 mmap(0xC000000000, 0x4000000, 0x3, 0x101
2, 0xFFFFFFFFFFFFFFFF, 0x0)          = 0xC000000000 0
 4374/0x206a2:      15936        2        0 mmap(0x0, 0x2000000, 0x3, 0x1002, 0xFFFF
FFFFFFFFFFFF, 0x0)          = 0x59B7000 0
 4374/0x206a2:      15942        2        0 mmap(0x0, 0x210800, 0x3, 0x1002, 0xFFFFF
FFFFFFFFFFF, 0x0)          = 0x4040000 0
 4374/0x206a2:      15947        2        0 mmap(0x0, 0x10000, 0x3, 0x1002, 0xFFFFFF
FFFFFFFFFF, 0x0)          = 0x1BD0000 0
 4374/0x206a2:      15993        3        0 madvise(0xC000000000, 0x2000, 0x8)
= 0 0
 4374/0x206a2:      16004        2        0 mmap(0x0, 0x10000, 0x3, 0x1002, 0xFFFFFF
FFFFFFFFFF, 0x0)          = 0x1BE0000 0
...
```

在这小节中，我们通过 macOS 的 `dtruss` 命令监听并查看了运行这个程序所进行的所有系统调用，发现了与内存管理有一定关系的方法如下：

- mmap：创建一个新的虚拟内存区域，但这里需要注意，**就是当系统调用 mmap 时，它只是从虚拟内存中申请了一段空间出来，并不会去分配和映射真实的物理内存，而当你访问这段空间的时候，才会在当前时间真正的去分配物理内存**。那么对应到我们实际应用的进程中，那就是 VSZ 的增长后，而该内存空间又未正式使用的话，物理内存是不会有增长的。
- madvise：提供有关使用内存的建议，例如：MADV_NORMAL、MADV_RANDOM、MADV_SEQUENTIAL、MADV_WILLNEED、MADV_DONTNEED 等等。
- mprotect：设置内存区域的保护情况，例如：PROT_NONE、PROT_READ、PROT_WRITE、PROT_EXEC、PROT_SEM、PROT_SAO、PROT_GROWSUP、PROT_GROWSDOWN 等等。
- sysctl：在内核运行时动态地修改内核的运行参数。

在此比较可疑的是 `mmap` 方法，它在 `dtruss` 的最终统计中一共调用了 10 余次，我们可以相信它在 Go Runtime 的时候进行了大量的虚拟内存申请，我们再接着往下看，看看到底是在什么阶段进行了虚拟内存空间的申请。

注：若是 Linux 系统，可使用 `strace` 命令。

查看 Go Runtime

启动流程

通过上述的分析，我们可以知道在 Go 程序启动的时候 VSZ 就已经不低了，并且确定不是共享库等的原因，且程序在启动时系统调用确实存在 `mmap` 等方法的调用，那么我们可以充分怀疑 Go 在初始化阶段就保留了该内存空间。那我们第一步要做的就是查看一下 Go 的引导启动流程，看看是在哪里申请的，引导过程如下：

```
graph TD
A(rt0_darwin_amd64.s:8<br/>_rt0_amd64_darwin) -->|JMP| B(asm_amd64.s:15<br/>_rt0_amd64)
B --> |JMP|C(asm_amd64.s:87<br/>runtime-rt0_go)
C --> D(runtime1.go:60<br/>runtime-args)
D --> E(os_darwin.go:50<br/>runtime-osinit)
E --> F(proc.go:472<br/>runtime-schedinit)
F --> G(proc.go:3236<br/>runtime-newproc)
G --> H(proc.go:1170<br/>runtime-mstart)
H --> I(在新创建的 p 和 m 上运行 runtime-main)
```

- runtime-osinit：获取 CPU 核心数。
- runtime-schedinit：初始化程序运行环境（包括栈、内存分配器、垃圾回收、P等）。
- runtime-newproc：创建一个新的 G 和 绑定 runtime.main。
- runtime-mstart：启动线程 M。

注：来自@曹大的 《Go 程序的启动流程》和@全成的 《Go 程序是怎样跑起来的》，推荐大家阅读。

初始化运行环境

显然，我们要研究的是 runtime 里的 `schedinit` 方法，如下：

```
func schedinit() {
    ...
    stackinit()
    mallocinit()
    mcommoninit(_g_.m)
    cpuinit()       // must run before alginit
    alginit()       // maps must not be used before this call
    modulesinit()   // provides activeModules
    typelinksinit() // uses maps, activeModules
    itabsinit()     // uses activeModules

    msigsave(_g_.m)
    initSigmask = _g_.m.sigmask

    goargs()
```

```
    goenvs()
    parsedebugvars()
    gcinit()
  ...
}
```

从用途来看，非常明显， `mallocinit` 方法会进行内存分配器的初始化，我们继续往下看。

初始化内存分配器

mallocinit

接下来我们正式的分析一下 `mallocinit` 方法，在引导流程中， `mallocinit` 主要承担 Go 程序的内存分配器的初始化动作，而今天主要是针对虚拟内存地址这块进行拆解，如下：

```
func mallocinit() {
    ...
    if sys.PtrSize == 8 {
        for i := 0x7f; i >= 0; i-- {
            var p uintptr
            switch {
            case GOARCH == "arm64" && GOOS == "darwin":
                p = uintptr(i)<<40 | uintptrMask&(0x0013<<28)
            case GOARCH == "arm64":
                p = uintptr(i)<<40 | uintptrMask&(0x0040<<32)
            case GOOS == "aix":
                if i == 0 {
                    continue
                }
                p = uintptr(i)<<40 | uintptrMask&(0xa0<<52)
            case raceenabled:
                ...
            default:
                p = uintptr(i)<<40 | uintptrMask&(0x00c0<<32)
            }
            hint := (*arenaHint)(mheap_.arenaHintAlloc.alloc())
            hint.addr = p
            hint.next, mheap_.arenaHints = mheap_.arenaHints, hint
        }
    } else {
        ...
    }
}
```

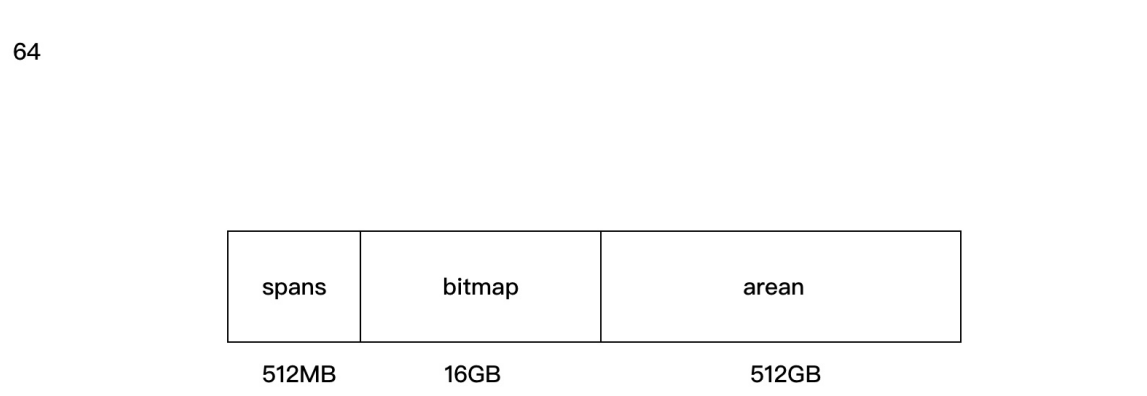
- 判断当前是 64 位还是 32 位的系统。
- 从 0x7fc000000000~0x1c000000000 开始设置保留地址。
- 判断当前 `GOARCH` 、`GOOS` 或是否开启了竞态检查，根据不同的情况申请不同大小的连续内存地址，而这里的 `p` 是即将要要申请的连续内存地址的开始地址。
- 保存刚刚计算的 arena 的信息到 `arenaHint` 中。

可能会有小伙伴问，为什么要判断是 32 位还是 64 位的系统，这是因为不同位数的虚拟内存的寻址范围是不同的，因此要进行区分，否则会出现高位的虚拟内存映射问题。而在申请保留空间时，我们会经常提到 `arenaHint` 结构体，它是 `arenaHints` 链表里的一个节点，结构如下：

```
type arenaHint struct {
    addr uintptr
    down bool
    next *arenaHint
}
```

- addr：`arena` 的起始地址
- down：是否最后一个 `arena`
- next：下一个 `arenaHint` 的指针地址

那么这里疯狂提到的 `arena` 又是什么东西呢，这其实是 Go 的内存管理中的概念，Go Runtime 会把申请的虚拟内存分为三个大块，如下：



- spans：记录 arena 区域页号和 mspan 的映射关系。

- bitmap：标识 arena 的使用情况，在功能上来讲，会用于标识 arena 的哪些空间地址已经保存了对象。
- arean：arean 其实就是 Go 的堆区，是由 mheap 进行管理的，它的 MaxMem 是 512GB-1。而在功能上来讲，Go 会在初始化的时候申请一段连续的虚拟内存空间地址到 arean 保留下来，在真正需要申请堆上的空间时再从 arean 中取出来处理，这时候就会转变为物理内存了。

在这里的话，你需要理解 arean 区域在 Go 内存里的作用就可以了。

mmap

我们刚刚通过上述的分析，已经知道 `mallocinit` 的用途了，但是你可能还是会有疑惑，就是我们之前所看到的 `mmap` 系统调用，和它又有什么关系呢，怎么就关联到一起了，接下来我们先一起来看看更下层的代码，如下：

```
func sysAlloc(n uintptr, sysStat *uint64) unsafe.Pointer {
    p, err := mmap(nil, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_PRIVATE, -1, 0
)
    ...
    mSysStatInc(sysStat, n)
    return p
}

func sysReserve(v unsafe.Pointer, n uintptr) unsafe.Pointer {
    p, err := mmap(v, n, _PROT_NONE, _MAP_ANON|_MAP_PRIVATE, -1, 0)
    ...
}

func sysMap(v unsafe.Pointer, n uintptr, sysStat *uint64) {
    ...
    munmap(v, n)
    p, err := mmap(v, n, _PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_FIXED|_MAP_PRIVA
TE, -1, 0)
  ...
}
```

在 Go Runtime 中存在着一系列的系统级内存调用方法，本文涉及的主要如下：

- sysAlloc：从 OS 系统上申请清零后的内存空间，调用参数是 `_PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_PRIVATE`，得到的结果需进行内存对齐。
- sysReserve：从 OS 系统中保留内存的地址空间，这时候还没有分配物理内存，调用参数是 `_PROT_NONE, _MAP_ANON|_MAP_PRIVATE`，得到的结果需进行内存对齐。

- sysMap：通知 OS 系统我们要使用已经保留了的内存空间，调用参数是 `_PROT_READ|_PROT_WRITE, _MAP_ANON|_MAP_FIXED|_MAP_PRIVATE`。

看上去好像很有道理的样子，但是 `mallocinit` 方法在初始化时，到底是在哪里涉及了 `mmap` 方法呢，表面看不出来，如下：

```
for i := 0x7f; i >= 0; i-- {
    ...
    hint := (*arenaHint)(mheap_.arenaHintAlloc.alloc())
    hint.addr = p
    hint.next, mheap_.arenaHints = mheap_.arenaHints, hint
}
```

实际上在调用 `mheap_.arenaHintAlloc.alloc()` 时，调用的是 `mheap` 下的 `sysAlloc` 方法，而 `sysAlloc` 又会与 `mmap` 方法产生调用关系，并且这个方法与常规的 `sysAlloc` 还不大一样，如下：

```
var mheap_ mheap
...
func (h *mheap) sysAlloc(n uintptr) (v unsafe.Pointer, size uintptr) {
    ...
    for h.arenaHints != nil {
        hint := h.arenaHints
        p := hint.addr
        if hint.down {
            p -= n
        }
        if p+n < p {
            v = nil
        } else if arenaIndex(p+n-1) >= 1<<arenaBits {
            v = nil
        } else {
            v = sysReserve(unsafe.Pointer(p), n)
        }
        ...
}
```

你可以惊喜的发现 `mheap.sysAlloc` 里其实有调用 `sysReserve` 方法，而 `sysReserve` 方法又正正是从 OS 系统中保留内存的地址空间的特定方法，是不是很惊喜，一切似乎都串起来了。

小结

在本节中，我们先写了一个测试程序，然后根据非常规的排查思路进行了一步步的跟踪怀疑，整体流程如下：

- 通过 `top` 或 `ps` 等命令，查看进程运行情况，分析基础指标。
- 通过 `pprof` 或 `runtime.MemStats` 等工具链查看应用运行情况，分析应用层面是否有泄露或者哪儿高。
- 通过 `vmmap` 命令，查看进程的内存映射情况，分析是不是进程虚拟空间内的某个区域比较高，例如：共享库等。
- 通过 `dtruss` 命令，查看程序的系统调用情况，分析可能出现的一些特殊行为，例如：在分析中我们发现 `mmap` 方法调用的比例是比较高的，那我们有充分的理由怀疑 Go 在启动时就进行了大量的内存空间保留。
- 通过上述的分析，确定可能是在哪个环节申请了那么多的内存空间后，再到 Go Runtime 中去做进一步的源码分析，因为源码面前，了无秘密，没必要靠猜。

从结论上而言，VSZ（进程虚拟内存大小）与共享库等没有太大的关系，主要与 Go Runtime 存在直接关联，也就是在前图中表示的运行时堆（malloc）。转换到 Go Runtime 里，就是在 `mallocinit` 这个内存分配器的初始化阶段里进行了一定量的虚拟空间的保留。

而保留虚拟内存空间时，受什么影响，又是一个哲学问题。从源码上来看，主要如下：

- 受不同的 OS 系统架构（GOARCH/GOOS）和位数（32/64 位）的影响。
- 受内存对齐的影响，计算回来的内存空间大小是需要经过对齐才会进行保留。

总结

我们通过一步步地分析，讲解了 Go 会在哪里，又会受什么因素，去调用了什么方法保留了那么多的虚拟内存空间，但是我们肯定会忧心进程虚拟内存（VSZ）高，会不会存在问题呢，我分析如下：

- VSZ 并不意味着你真正使用了那些物理内存，因此是不需要担心的。
- VSZ 并不会给 GC 带来压力，GC 管理的是进程实际使用的物理内存，而 VSZ 在你实际使用它之前，它并没有过多的代价。
- VSZ 基本都是不可访问的内存映射，也就是它并没有内存的访问权限（不允许读、写和执行）。

看到这里舒一口气，因为 Go VSZ 的高，并不会对我们产生什么非常实质性的问题，但是又仔细一想，为什么 Go 要申请那么多的虚拟内存呢，到底有啥用呢，考虑如下：Go 的设计是考虑到 `arena` 和 `bitmap` 的后续使用，先提早保留了整个内存地址空间。 然后随着 Go Runtime 和应用的逐步使用，肯定也会开始实际的申请和使用内存，这时候 `arena` 和

`bitmap` 的内存分配器就只需要将事先申请好的内存地址空间保留更改为实际可用的物理内存就好了，这样子可以极大的提高效能。

参考

- 曹大的 Go 程序的启动流程
- 全成的 Go 程序是怎样跑起来的
- 推荐阅读 欧神的 go-under-the-hood
- High virtual memory allocation by golang
- GO MEMORY MANAGEMENT
- GoBigVirtualSize
- GoProgramMemoryUse