Golang 实战

使用 gin+xorm 搭建 go 语言 web 框架详解

Author：winlion
Version: 0.0.0.1

目录

序言

2017 年底我们启动了新产品数织时尚的研发工作,当时我们面临这语言选型工作一个是 java 一个是 golang,我们一致认为 golang 是一款非常优秀的语言,但是因为生态不够丰富,缺少欣欣向荣的开发者气氛,因此我们选择了 java 阵容的 spring boot 作为开发语言,但是我个人心中总觉得需要为 golang 做一些什么,因此有了这本书。这本书适合有 golang 基础的读者,这本书的作用不在于提供一个 golang 框架，而在于提供封装一个框架的过程。

在写做本书前,我已经在互联网行业从事技术相关工作达到 10 年之久,期间也写过一些书如《你必须懂得 18 个移动互联网模型》,购买地址

https://yuedu.baidu.com/ebook/977275a5767f5acfa0c7cd55

笔者 winlion,个人微信号 jiepool-winlion,本书更新将在微信里做相应说明。

1、概述

1.1 go 语言的困境

2017 年我们公司需要快速迭代一款产品,当时,我们团队的后端框架是 spring mvc ，该框架结构清晰,上手快,但是由于我们的产品迭代速度快,底层数据库操作接口变动频繁,导致 service 层工作量巨大,不胜其烦。另外,随着项目的成长，代码量越来越大,项目启动越来越慢,严重影响了开发调试速度。

在这种情况下,我们希望寻找一种新的框架或编程语言,我们期望他具备调试简单，上手快，启动速度快，保密性高，以及适用于高并发及 web 编程,性能优越等优点。当时手上备选框架有 spring boot 和 golang,综合比较,在巨大的遗憾中,我们选择了 springboot，相关选型参考信息如下。

1、从功能满足度比较

在启动该产品研发时,我们梳理了自己的需求,主要在三个方面,一是支持高性能的 restful api 服务,二是支持 web 页面服务,三是支持快速跌代。Restful 接口及性能方面,spring boot 和 golanbg 都支持,且性能区别不大。页面服务方面,spring boot 封装了 spring mvc，相对成熟度更高，并且由于大量 javaer 的长期积累，该框架已经非常成熟；而 go lang 方面，也有成型的框架,如 beego，但是该框架使用度并不高．究其原因，是因为 beego 没有形成像 php 框架那样的生态，比如行业内 php 开源框架 thinkphp，目前基于该框架已经形成了大量的 cms，商城，erp,微信管理等软件,这也是我为什么要写作该书的原因，我希望通过写作该书,让更多的人认识到 go 语言,并参与到 golang 的生态搭建中来。回归正题,功能满足度,spring boot 更胜一筹。

2、学习成本

我们需要考虑到团队的学习成本,当时团队已经非常熟悉 spring mvc 框架,如果切换 spring boot，学习成本几乎为零,但是若切换到 golang,学习成本相对较高。另外,网络上关于 golang 框架,推荐并不多,比如 gin,decho，以及 beego，我们重点了解了 beego 框架,知乎上争论颇多。这些,都让我们对是否使用 go 语言,心里预期上打了一些折扣。

3、项目迁移成本

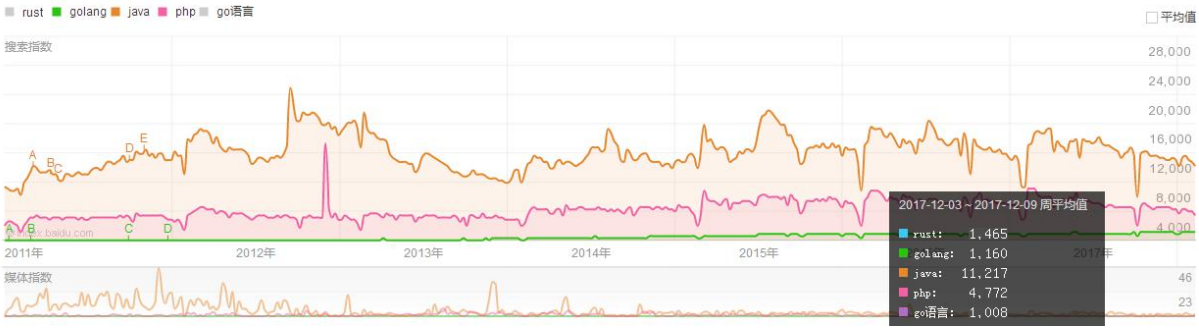
在选择框架的时候,我们还考虑了我们的集成成本。我们的客观情况是:公司的技术栈以及这些技术栈形成的项目积累,几乎都是 java,如果我们切换到 golang，那么产品运维成本将相对较高，另外由于目前行业类 golang 从业者较少,如果我们采用 go,将人为地为我们的项目团队组建带来巨大的难度.

4、社区活跃度

Go 语言社区活跃度并不高,具体可以看如下几个方面

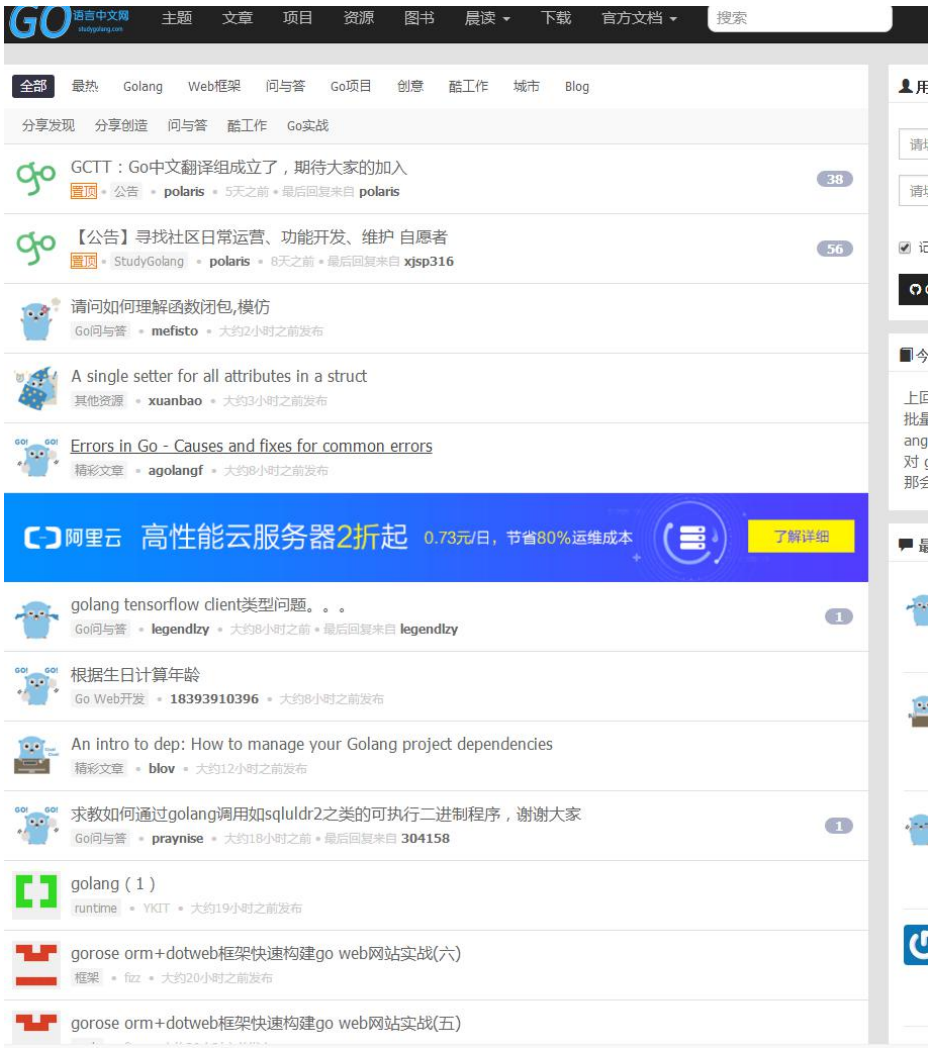
- 百度搜索指数

下图是 golang 和 java 以及 php 的搜索指数,近段时间以来,golang 关注度不断下降,已经到日均 1000pv，如果 golang 不孵化生态圈,将面临淘汰.事实上,在百度上搜索 golang,很多都是 2014 年的旧事了。



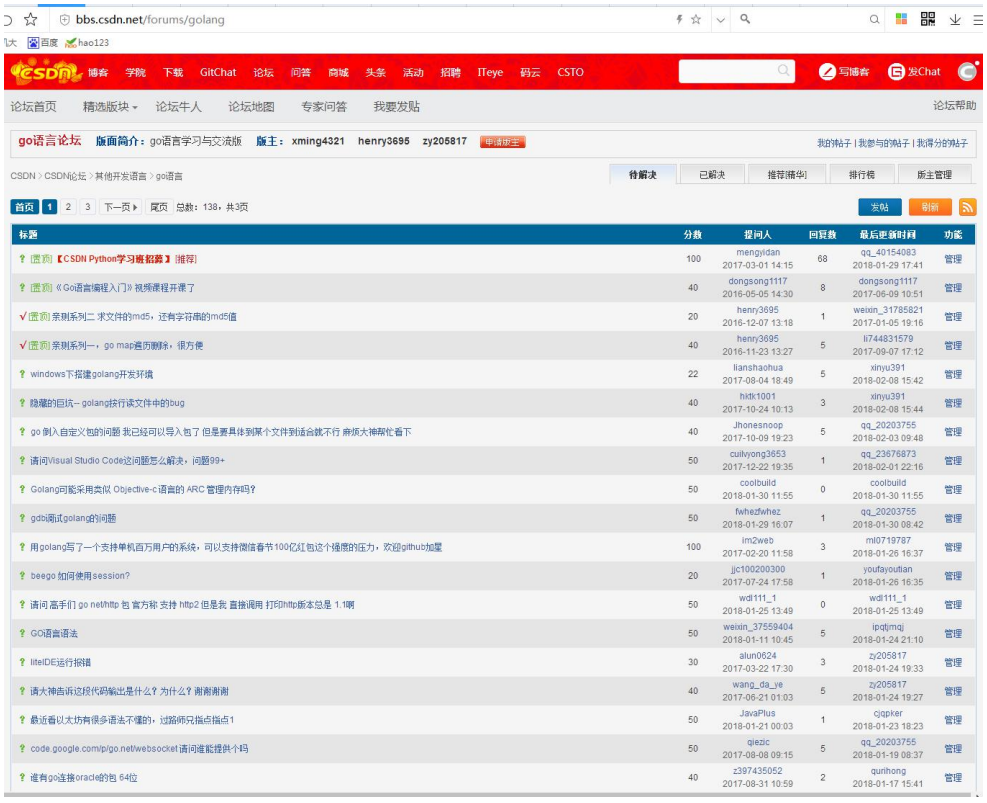
- Go 技术论坛

笔者 2018-02-10 18:11 访问了 https://studygolang.com/ 首页截图如下



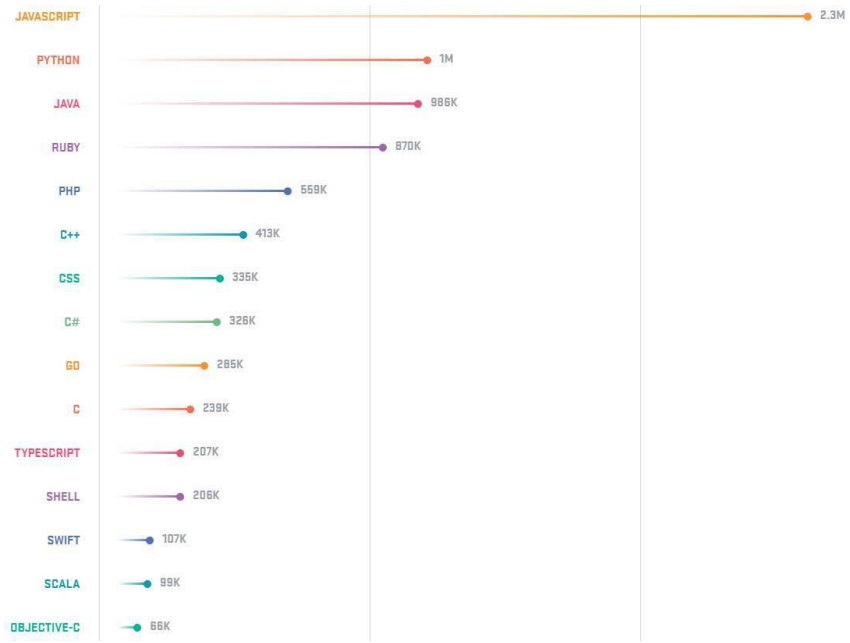
该社区在百度搜索 go 语言论坛 排名第一, 但是日更新不超过 20 篇文章, 一股悲凉浮在心头。

下图为 golang csdn 社区论坛, 一周以内不超过 10 篇帖子, 心中又凉了半截。



- Github go 活跃度

下图为 github 发布的编程语言 2017 年度排行 Go 语言排行第九.这意味着全球活跃度还是有,但相比 php,java 等而言 go 还有很长的任务要走。



1.2 我要做什么

个人觉得 golang 是一款非常优秀的语言,但是非常遗憾,目前还不是我们团队的最优选择,为

了弥补这份遗憾,我们需要做些什么。梳理一下,我要做的事情,主要有如下几点

1、演示如何集成 golang web 框架。本书的目的不是在于集成一个 golangweb 框架,而是在于演示如何集成一个 golang 框架,原因如下

- golang 框架已经很多。国外的如 matini,revel,gin 等,国内也有 beego.这些框架都很优秀,各有所长,但也正是因为这些,所以有毁有誉,当然这不是重点。
- 我们往往会碰到很多场景,现有的框架能解决我们的问题,但不是最好的解决方案,需要定制我们自己的框架。
- 我们对框架有要求,希望简单好用,同时又希望具备高扩展性和开发性。
- 我们看到上述各种框架的优势,我们希望将他们整合起来,扬长补短。

基于如上原因,我决定不重新制造轮子，而是给大家做一个示例,怎么制造轮子。同样地,正是因为这些因素，我们在搭建过程中会大量使用第三方优秀成果,这从根本上决定了我们产品的开放性。

2、丰富 golang 应用生态圈。golang 上手简单,和 php 有点相似,有人甚至会认为 golang 是一种解释性语言。Golang 效率高,天然支持并发,性能好，又能打包成可执行文件,无需容器，安全可靠,但是为什么这么一种语言,为什么没有获得和 php 一样高的使用率呢?我认为主要是 golang 应用生态圈没有丰富起来。

- 长期以来,人们认为 golang 只适合做高并发,高科技的事儿,这可能是因为 golang 出生高贵吧,golang 在又拍云得到大规模运用,和云计算沾上边儿,有点曲高和寡的味道。
- 类比 php，我们会发现博客软件有 wordpress，cms 有织梦帝国,商城有 ecshop、ecmall，开元框架有 thinkphp，laravel，yii,而 golang 语言,我们有什么?应用生态圈极度缺乏。
- 类比 java， java 有庞大的生态圈和中间件,java 已经是企业级应用的首选语言,这不单是因为 java 性能好,跨平台,还因为 java 已经有好几代使用者,是忠实粉丝，而 golang，使用者都是弄潮儿。

以上几点可以看出,丰富 golang 生态是 golang 推广的重中之重。

3、大力宣传和推广 golang。我计划大力宣传和推广 golang，具体做法如下

- 宣传和推广 golang 传统行业应用场景,主要包括 cms 应用,个人博客应用,商城应用、微信管理后端等五大基础领域。
- 宣传和推广 golang 移动互联网应用场景,golang 具备天然的高并发优势和快速迭代优势,我们可以 golang 推广到互联网场景应用，如小游戏上。
- 挖掘新的 golang 使用场景,使 golang 更接地气,如秒杀、拍卖、大数据采集和处理等场景都是 golang 用武之地。
- 其他渠道如搜索引擎 seo 推广、行业热点推广

2、框架基本结构

一个典型的项目框架,文件夹目录如下,下面我们来逐一说明这些文件夹或文件的作用.

```
|  app.properties
|  favicon.ico
|  log.xml
|  main.go
```

```
│   README.md
├──args
│       PageArg.go
├──assets
│   ├──css
│   ├──fonts
│   ├──images
│   ├──js
│   └──plugins
├──core
│       logger.go
├──ctrls
│       attach.go
│       ctrl.go
│       mod1.go
│       page.go
├──models
│       model1.go
├──service
├──tests
│       default_test.go
└──views
    │   index.html
    └──modx
```

2.1 控制器文件夹 controller

该目录主要存储控制器文件，特别地，我们做如下约定

1、 文件名称首字母一律大写,后面必须添加 Controller 结尾,用做标识这是控制器文件,如 UserCtrl，我们就能一目了然地看到这是一个控制器文件
2、 我们约定控制器文件名必须和业务强相关,比如用户相关的控制器,我们需要设计文件名为 UserCtrl.go，资源管理相关的控制器,我们需要设计名称为 AttachCtrl
3、 文件名称一律区分大小写

2.2 实体目录 entity

该目录主要存储数据库对应模型文件，和 java 类似,我们做如下约定

1、 文件名称首字母一律大写,并与数据库内对应的表名称保持一致。
2、 数据库表名称中,以下划线开头的字母,对应的实体类中相应的字段必须大写.如 user_info 类,对应的实体名称必须为 UserInfo.go
3、 文件名称一律区分大小写

2.3 视图目录 view

视图目录主要包括如下子目录

1、 公共模板目录 public,该目录下存放公共模板如用于统一设置 head 的 head.html,用于统一设置底部的 foot.html,用于统一错误页面的 error.html
2、 应用模块目录,我们强烈建议每一个模块用一个目录来独立存放,该方式结构接单清晰明了.
3、 假设用户模块有注册,登录,密码找回,个人资料等四个逻辑页面,那么我们可以建立 User 模块,下辖登录页面 login.html,注册页面 register.html,密码重置页面 resetpwd.html,个人资料页面 profile.html.

2.4 静态资源 asset

该目录主要用于存放静态资源,一般情况下,该目录下存放如下几个子目录

1、 image，用于存放静态图片文件
2、 css，用于存放 css 文件
3、 js，用于存放 js 文件.我们常页面的 js 逻辑文件也投放到该目录下,该目录可以建立子目录,和 view 下的子目录一一对应
4、 font,一般情况下,我们使用 bootstrap 框架时，会用到字体文件,那该文件夹用于存储字体文件
5、 plugin,该文件夹用于存放较大的插件,如 kindedit 插件,bootstarp,adminlte.等，在这里,我们将包含 css 和 js 或者 image 的文件包叫做插件

静态资源独立存放是有好处的,将来可以非常方别地实施动静态分离.

2.5 业务层目录 service

Service 文件夹用于存放业务层逻辑,所谓业务层是指具体某一业务实现的方式，对外提供接口，对内调用数据库操作。业务层命名我们做如下约定

1、 所有业务逻辑名称首字母必须大写,且只能为字母。
2、 所有业务逻辑文件必须以 Service 结尾

打个比方,对于用户业务逻辑,我们定义文件名称 UserService.go 即可

2.6 参数封装层 model

我们将每一个业务的请求参数封装成一个 struct,比如说,对于用户管理模块,我们可能用到根据关键字如姓名、电话等查询用户信息、根据注册时间查询用户信息的，以及分页支持、排

序支持等。因此我们可以将这些参数封装成一个 bean，比如本例子中我们可以作如下封装

```
type    PageArg struct{
    Kword  string  `form:”kword” json:”kword”`
    Datefrom  time.Time  `form:”datefrom”  json:”datefrom”`
    Dateto  time.Time  `form:”dateto”  json:”dateto”`
    Desc  string  `form:”desc”  json:”desc”`
    Asc  string  `form:”asc”  json:”asc”`
    Pagefrom  int  `form:”pagefron”  json:”pagefrom”`
    Pagesize  int  `form:”pagesize”  json:”pagesize”`
}
```

考虑到有些参数是常用的,我们可以将 UserArg 做如下定义

```
Type  UserArg struct{
    PageArg
     //…… other arg
}
```

2.7 核心包 restgo

核心包内置我们的应用管理框架以及我们我们需要用到的括常用的工具类软件，具体描述如下

1、 orm 封装工具 OrmEngin.go
2、 常用自定义函数数 FunMap.go
3、 参数响应结果封装 Result.go
4、 验证码管理 Captcha.go
5、 网络访问模块 Http.go
6、 加密方法封装 Crypto.go
7、 应用管理模块 Restgo.go

其他需要扩张的用户可以自行添加

2.8 配置目录 config

配置存放目录在 config 下,具体内容如下

1、 日志配置文件 log4g.xml
2、 应用配置文件 application. Properties

其他配置文件可以自行添加

2.9 其他

应用启动文件 main.go 位于跟目录下

3、配置参数

3.1 系统常用配置参数

对于一个 web 应用,我们需要关注的参数很多,如下代码展示了常用的参数配置

```
#应用运行模式,我们采用了 gin 框架,目前支持 debug/release/test 三种
restgo.app.mode=debug
#应用的名称,以后扩展,用做应用标识，便于分布式计算
restgo.app.name=restgo 演示
#应用部署的访问协议,支持 http/https 两种
restgo.app.protocal=http
#应用域名
restgo.app.domain=localhost
#静态资源所在的服务器地址,便于动静态分离
restgo.app.asset=localhost
#请求 contextpath
restgo.app.ctxpath=
#服务器绑定的地址
restgo.app.addr=
#端口
restgo.app.port=80
#sessionID 标识字符串,对标 PHP 的 SESSIONID,java 的 JSESSIONID
restgo.session.name=GSESSIONID
#session 过期时间以秒为单位,0 表示访问结束时过期
restgo.session.timelive=3600
#默认数据资源配置,数据库驱动类型为 mysql,详情可以参开 xorm
restgo.datasource.default.driveName=mysql
#连接串
restgo.datasource.default.dataSourceName=root:root@/restgo?charset=utf8
#连接池中 idle 态链接最大个个数
restgo.datasource.default.maxIdle=10
#连接池最大打开连接数
restgo.datasource.default.maxOpen=5
#是否显示 sql 语句
restgo.datasource.default.showSql=true

#支持配置多个数据库
```

```
restgo.datasource.ds1.driveName=mysql
#连接串
restgo.datasource.ds1.dataSourceName=root:root@/restgo2?charset=utf8
#连接池中 idle 态链接最大个个数
restgo.datasource.ds1.maxIdle=10
#连接池最大打开连接数
restgo.datasource.ds1.maxOpen=5
#是否显示 sql 语句
restgo.datasource.ds1.showSql=true

#log4g 日志配置文件路径地址
restgo.logger.filepath=config/log4g.xml

#静态资源及映射,如下配置则访问 localhost/assets/a.jpg 则系统将去 asset 目录
下寻找 a.jsp
restgo.static.assets=./asset
#图片资源存放路径访问 localhost/mnt/a.jpg 则系统将去 /data/restgo/mnt 目录下
寻找 a.jsp
restgo.static.mnt=/data/restgo/mnt
#favorite.ico 访问 localhost/favorite.ico 则渲染./favorite.ico
restgo.staticfile.favicon.ico=favicon.ico
#视图存放路径
restgo.view.path=view
#视图中模板标签开始标记
restgo.view.deliml={{
#视图中模板标签结束标记
restgo.view.delimr=}}
#自定义微信配置支持
restgo.weixin.appid=a1278287120128
restgo.weixin.appsecret=$^siwe2i23i^(12
restgo.weixin.token=helloword


#redis 服务器地址
restgo.redis.host=localhost
#redis 端口
restgo.redis.port=3306
#redis  密码
restgo.redis.passwd=$^siwe2i23i^(12
#使用的数据库
restgo.redis.db=0
```

3.2 解析配置文件

配置文件一般以.xml,.ini 等格式存储, java 中常见以.properties 文件和.yml 格式的文件,我们这里以.properties 格式来存储配置信息,这个在 java 中是非常常见的。配置对应结构体如下

```
type Config struct {
    App map[string]string
    Session map[string]string
    Datasource map[string](map[string]string)
    Static map[string]string
    StaticFile map[string]string
    Logger map[string]string
    Templete map[string]string
    All map[string]string
}
```

其中,配置文件解析的核心在于理解以 map 容器存储 key-value 类型数据.为了区分各种配置,我们针对不同的配置类型设计了不同的 map,在上图中,App 参数配置对应 restgo.app,Session 参数配置对应 restgo.session,其他依此类推。
需要重点说明的是 datasource,由于我们生产环境需要支持多数据源,因此我们需要配置多数据源支持,这样对应 map 设计成 `map[string](map[string]string)` 的形式。

3.3 添加自定义配置文件

考虑到系统灵活性,系统需要支持添加配置文件，本章节以微信 api 对接为例来说明如何定制自定义的配置。

1、 首先设计模块模块名称,比如设置为 weixin
2、 接着设计需要配置的属性,微信 api 对接需要关注的属性包括公众平台的 appid,appsecret, 以及双方加密因子 token。

最终设计输入配置如下

```
#自定义微信配置支持
restgo.weixin.appid=a1278287120128
restgo.weixin.appsecret=$^siwe2i23i^(12
restgo.weixin.token=helloword
```

3.4 获取配置信息

可以通过如下函数获得配置信息

```
//获取整数,
func (cfg *Config) LoadCfg(key string) string {
```

```
    return cfg.All[key]
}
//获取字符串配置
func (cfg *Config) LoadString(key string) string {
    return cfg.All[key]
}
//获取整数,
func (cfg *Config) LoadInt(key string) (int,error) {
    return strconv.Atoi (cfg.All[key])
}
//获取 32 位整数
func (cfg *Config) LoadInt64(key string) (int64,error) {
    return strconv.ParseInt(cfg.All[key],10,64)
}
//获取 64 位整数
func (cfg *Config) LoadInt32(key string) (int64,error) {
    return strconv.ParseInt(cfg.All[key],10,32)
}

//获取布尔配置
func (cfg *Config) LoadBool(key string) bool {
    return cfg.All[key]=="true" || "TRUE"==cfg.All[key]
}

//cfg.loadCfg("restgo.weixin.appid")
```

有时候我们可以对配置函数进行扩展，可以直接修改 Config.go 中

4、路由设置

4.1 路由统一管理

路由的本质是将用户请求的 requesturi 与后端实现的业务函数进行绑定,比如用户访问 http://localhot/user/register , requesturi 为/user/register ,代表用户需要调用注册业务,后端需要提供一个函数，或者展示注册页面,或者提供注册账号服务。另外，随着系统的日益复杂,路由配置参数越来越多,如果不统一管理将会导致前短接口混乱,不利于开发和维护。

那么,设计一个路由管理模块,需要考虑哪些因素呢？一般来说有如下几点:

1、 每一个控制器对应一个小模块,在控制器内部实现路由注册功能,这样有利于代码维护,并且思路清晰

2、 只需要对控制器进行注册,即可需要控制器中的路由注册,

3、 路由需要有容错功能,比如 http://localhot//user/register 和 http://localhot/user/register 应该

具备相似的操作结果

当前系统设计路由如下

```
//具体控制器
type PageController struct {
	restgo.Controller
}
//实现 Router 方法
func (ctrl *PageController)Router(router *gin.Engine){
	router.POST("page/create",ctrl.create)
	router.POST("page/update",ctrl.update)
	router.POST("page/query",ctrl.query)
	router.POST("page/delete",ctrl.delete)
	router.POST("page/findOne",ctrl.findOne)
	router.GET("/",ctrl.showIndex)
}
Main.go 主函数中调用如下代码

func registerRouter(router *gin.Engine){

	new(controller.PageController).Router(router)

}

func main(){
   router := gin.Default()
   //……
   registerRouter(router *gin.Engine)
}
```

如果我们需要添加一个控制器,在 registerRouter 中添加注册就可以了。

4.2 错误信息统一配置

错误信息统一处理是为了提高系统友好性，将不能识别的路由以及内部跳转的路由统一配置,别于前端统一定制。

1、路由路径错误

路由路径错误,比如大小写敏感,/user/register 被写作了/user/Register,再比如/user/register 变成了//user/register。

2、后端未提供服务

有一些统一处理的路由规则，如果规则和服务之间尚未提供明确的服务,则会调用错误信息。我们可以对这一特效加以利用,在下一个章节会重点提及。

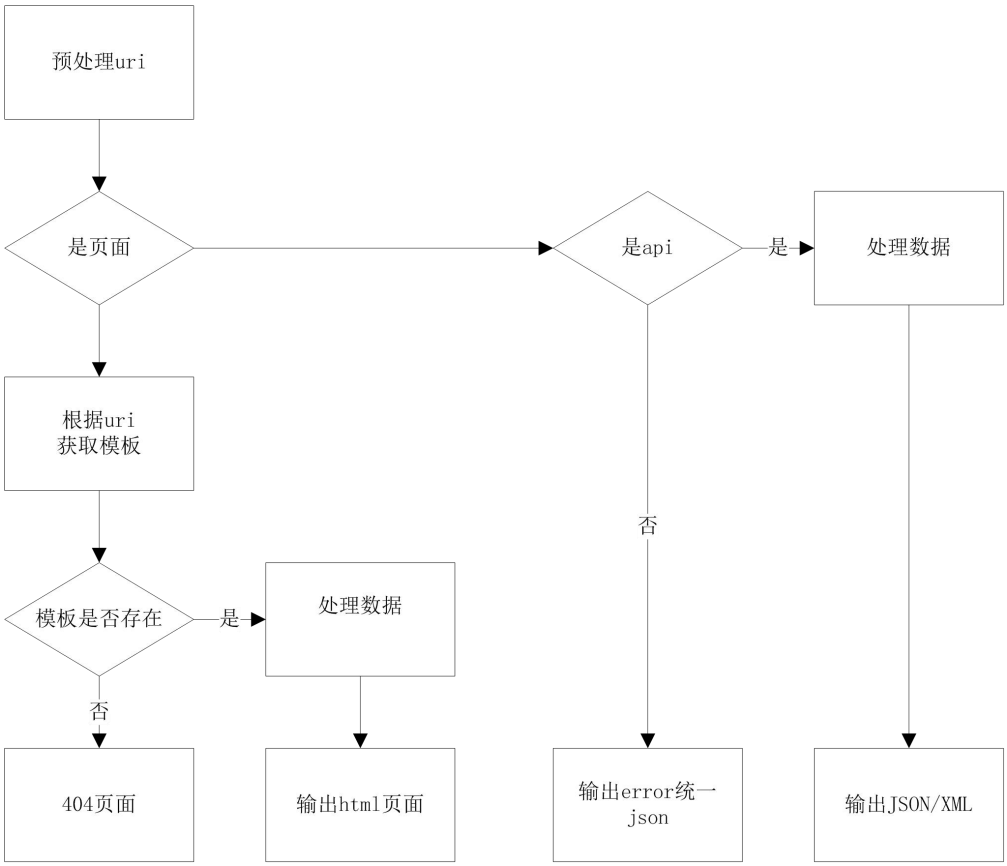
3、系统 runtime 类错误

比如外部传递俩个参数,a 和 b，内部计算 c=a/b 当 b 为 0 时则会出现 runtime 错误,此时会调

用错误配置方法.

4.3 定制自己的路由框架

如上所示展示了路由管理模块需要考虑的一些因素,下图展示一个路由管理模块的大致流程。



1、预处理 uri,主要是对 uri 进行预处理,包括格式判断,过滤斜杠等.

2、如何判断页面，我们约定使用.shtml 结尾的都是页面,不包含任务后缀的都是 api

3、如何判断模板是否存在：我们通过统一判断文件路径是否存在来判断模板是否存在,为了实现该功能,首先需要做出约定

所有模板都存放到 view 文件夹下;模板名和文件名存在一一对应关系,如下所示则对于模板文件位置为 view/user/list.html

```
{{define "user/list.html"}}
//somthing
{{end}}
```

接着在 Config 中对模板进行初始化

```
f, _ := filepath.Glob("view/**/*")
    for _, b := range f {
        cfg.TempFileMap[b] = 0
    }
```

最后在默认路由函数中判断 requesturi 是否存在在 cfg.TempFileMap 中,具体代码如下

```
var urimap map[string]int = make(map[string]int)
//Controller.go
//对未定义的路由规则进行处理
func NoRoute(ctx *gin.Context) {
   uri := ctx.Request.RequestURI
   isAjax := "XMLHttpRequest"==ctx.GetHeader("X-Requested-With")
   isPage := strings.Contains(ctx.Request.RequestURI,".shtml")

   uri = strings.TrimLeft(uri, "/")
   uri = strings.TrimSuffix(uri, ".shtml")
   //如果已经定义过了则是一定存在的
   //存在则=计算统计次数
   //不存在则为-1
   //0 代表初始化

   //如果定义了,

   stat, has := urimap[uri]
   //如果有
   if !has {
      //没有则先初始化一下
      urimap[uri] = 0

   }
   if 0 == stat {
      //寻找初始化的数据
      cfg := GetCfg()
      var flag int = -1
      for fpath, _ := range cfg.TempFileMap {
         fpath = strings.Replace(fpath,"\\","/",-1)
         if strings.Index(fpath, uri) > -1 {
            flag = 1
            break
         }
         fmt.Print(fpath)
      }
      urimap[uri] = stat + flag
   }

   //如果不存在则跳转出错页面
   if 0 > urimap[uri] {
      NoMethod(ctx)
   } else {
```

```
      //如果是 AJAX
      if isPage{
         //response html
         ctx.HTML(200, uri+".html", "Q")
      }else if isAjax{
         //response json
         ctx.JSON(200,nil)
      }

   }

}


func NoMethod(ctx *gin.Context) {
   uri := ctx.Request.RequestURI
   fmt.Printf("NoMethod" + uri)
   uri = strings.TrimLeft(uri, "/")
   uri = strings.TrimSuffix(uri, ".shtml")
   //ctx.HTML(http.StatusOK, model+"/"+action+".html", gin.H{"title":
"test"})
   ctx.HTML(200, uri+".html", "Q")
}
```

5 控制器 C

无论多么先进的框架，mvc 思想都是值得借鉴的, mvc 是模型(model)－视图(view)－控制器(controller)的缩写，他是一种软件设计典范，他用业务逻辑、数据、界面显示分离的方法组织代码，将业务逻辑聚集到一个部件里面，这样做有巨大的好处,他在改进和个性化定制界面及用户交互的同时，不需要重新编写业务逻辑。本章节重点阐述 controller 层.

5.1 控制器定义

控制器文件通常放在 controller 目录下，控制器 struct 名和文件名保持大小写一致，并采用驼峰命名（首字母大写）。
一个典型的控制器类定义如下：

```
//PageController.go
package controller

import (
   "restgo/restgo"
```

```
    "github.com/gin-gonic/gin"
    "net/http"
    "fmt"
)

type PageController struct {
    restgo.Controller
}
//前置操作
func (ctrl *PageController)before() gin.HandlerFunc {
    return func(ctx *gin.Context) {
        uri := ctx.Request.RequestURI
        fmt.Print(uri)
        if 1==1{
            ctx.Next()
        }
        return
    }
}

func (ctrl *PageController)Router(router *gin.Engine){
    router.GET("/",ctrl.showIndex)
    r := router.Group("page").Use(ctrl.before())
    r.POST("create",ctrl.create)
    r.POST("update",ctrl.update)
    r.POST("query",ctrl.query)
    r.POST("delete",ctrl.delete)
    r.POST("findOne",ctrl.findOne)

}

func (ctrl * PageController) create(ctx *gin.Context){
    ctrl.Data = []int{1,2,3}
    ctrl.AjaxData(ctx)
}
func (ctrl * PageController) showIndex(ctx *gin.Context){

    ctx.HTML(http.StatusOK,"panel/index.html","")
}
func (ctrl *PageController)delete(ctx *gin.Context){


}
func (ctrl *PageController)update(ctx *gin.Context){
```

```
}
func (ctrl *PageController)query(ctx *gin.Context){


}

func (ctrl *PageController)findOne(ctx *gin.Context){


}
```

控制器文件分为四部分

- struct 定义部分,一般我们让控制器和 restgo.Controller 拥有同样的方法即可。restgo.Controller 没有做很重的封装,只实现简单的数据响应接口。大家可以自行封装。
- 前置操作 before 部分,这一部分主要实现前置操作,当一个控制器里面的方法被访问时，需要进行特别的一些数据处理,比如鉴权，或者参数校验,可以在这个方法里面进行操作,前置操作逻辑不宜太复杂。
- 路由控制函数 router，这个函数是必须实现的方法,我们需要在该函数中实现路由配置和映射
- 业务函数实现,主要通过函数实现,一个控制器通常实现 fineOne/query/create/update 方法。

5.2 跳转和重定向

跳转重定向实现非常简单,我们调用 gin 框架内置的即可

```
func (ctrl *PageController)Redirect(ctx *gin.Context){

   ctx.Redirect(302,"/")

}
```

5.3 资源控制器

一般来说我们实现 api 服务器无需资源控制器,因为 api 一般只需要响应 json/xml，但有部分应用比如 cms,需要考虑到静态资源服务，这里我们需要使用资源控制器。资源控制器设计需要考虑几个方面

- 资源需要实现缓存,这样可以加快系统访问速度。
- 资源管理器需要支持灵活配置,因为很多资源服务是和 nginx 服务器反向代理相互整合的,需要考虑易整合性。
- 资源管理需要统一规划。资源有静态的如 css、js 文件也有动态产生的,如上传后得到的图片文件,因此需要动态规划保证资源的存储空间和访问规则等。

我们可以对 gin 框架做一些封装,就可以实现强大的资源管理功能

```
app.properties
#静态资源及映射,如下配置则访问 localhost/assets/a.jpg 则系统将去 asset 目录下寻找 a.jsp
restgo.static.assets=./asset
#图片资源存放路径访问 localhost/mnt/a.jpg 则系统将去 /data/restgo/mnt 目录下寻找 a.jsp
restgo.static.mnt=/data/restgo/mnt
#favorite.ico 访问 localhost/favorite.ico 则渲染./favorite.ico
restgo.staticfile.favicon.ico=./favicon.ico
```

这些实际上是通过 main 函数中的方法实现的

```
//main.go
for k,v :=range cfg.Static{
   router.Static(k, v)
}
for k,v :=range cfg.StaticFile{
   router.StaticFile(k, v)
}
```

其中 static 本质上是调用 golang 内置的文件服务，absolotepath 是文件夹绝对路径

```
fileServer := http.StripPrefix(absolutePath, http.FileServer(fs))
fileServer.ServeHTTP(c.Writer, c.Request)
```

而 staticfile 方法本质上是调用 gin 自身实现的 GET 方法

```
group.GET(relativePath, handler)
group.HEAD(relativePath, handler)
```

5.4 参数绑定

controller 绑定参数常用如下方法

1、 获取 path 中的参数

```
// this one will match /user/john/ and also /user/john/send
// If no other routers match /user/john, it will redirect to /user/john/
router.GET("/user/:name/*action", func(c *gin.Context) {
   name := c.Param("name")
   action := c.Param("action")
   message := name + " is " + action
   c.String(http.StatusOK, message)
})
```

2、 获取 query string 中的参数

```
// Query string parameters are parsed using the existing underlying request object.
// The request responds to a url matching:
/welcome?firstname=Jane&lastname=Doe
router.GET("/welcome", func(c *gin.Context) {
```

```
   firstname := c.DefaultQuery("firstname", "Guest")
   lastname := c.Query("lastname")
   // shortcut for   c.Request.URL.Query().Get("lastname")

   c.String(http.StatusOK, "Hello %s %s", firstname, lastname)
})
```

3、 处理表单数据

```
// Query string parameters are parsed using the existing underlying
router.POST("/form_post", func(c *gin.Context) {
   message := c.PostForm("message")
   nick := c.DefaultPostForm("nick", "anonymous")

   c.JSON(200, gin.H{
      "status":  "posted",
      "message": message,
      "nick":    nick,
   })
})
```

4、 处理单文件上传

```
//
curl -X POST http://localhost:8080/upload -F \
"file=@/Users/appleboy/test.zip" \
-H "Content-Type: multipart/form-data"

 router.POST("/upload", func(c *gin.Context) {
   // single file
   file, _ := c.FormFile("file")
   log.Println(file.Filename)

   // Upload the file to specific dst.
   // c.SaveUploadedFile(file, dst)

   c.String(http.StatusOK, fmt.Sprintf("'%s' uploaded!",
file.Filename))
})
```

5、 处理多文件上传

```
// curl -X POST http://localhost:8080/upload \
-F "upload[]=@/Users/appleboy/test1.zip" \
-F "upload[]=@/Users/appleboy/test2.zip" \
-H "Content-Type: multipart/form-data"
router := gin.Default()
// Set a lower memory limit for multipart forms (default is 32 MiB)
// router.MaxMultipartMemory = 8 << 20  // 8 MiB
```

```
router.POST("/upload", func(c *gin.Context) {
   // Multipart form
   form, _ := c.MultipartForm()
   files := form.File["upload[]"]

   for _, file := range files {
      log.Println(file.Filename)

      // Upload the file to specific dst.
      // c.SaveUploadedFile(file, dst)
   }
   c.String(http.StatusOK, fmt.Sprintf("%d files uploaded!", len(files)))
})
```

5.5 模型绑定

模型绑定常见方法分为俩类,一类是 MustBind 类,像 Bind,BindJSON,BindQuery 都是这一类,这一类绑定主要特征是一旦绑定失败则直接返回 400 错误;另一类绑定是 ShouldBind 类,像函数 ShouldBind，ShouldBindJSON，ShouldBindQuery 都是这一类,这一类绑定一旦失败,并不会立即响应 400 错误,而是将错误信息返回给上下文环境,开发者 需要自行处理。常见方法如下,实际上 gin 框架支持 JSON 、XML、Form、Query、FormPost、FormMultipart 、ProtoBuf 、MsgPack 等格式

```
ctx.ShouldBindJSON(&pageArg)
ctx.BindJSON(&pageArg)
ctx.ShouldBind(&pageArg)
ctx.BindWith(&pageArg,binding.JSON)
ctx.ShouldBindWith(&pageArg,binding.JSON)
```

下面我们演示数据绑定流程

1、 在 model 文件夹下定义 PageArg 结构体,注意其中如果需要绑定 json 数据则必须定义关于 json 的 tag，如果需要绑定 form 表单类型的数据则需要定义包含 form 的 tag

```
type PageArg struct {
   Kword string `form:"kword"`
   Datefrom time.Time `form:"datefrom" time_format:"2006-01-02
15:04:05"`
   Dateto time.Time   `form:"dateto" time_format:"2006-01-02
15:04:05"`
   Pagesize int       `form:"pagesize" json:"pagesize"`
   Pagefrom int       `form:"pagefrom" json:"pagefrom"`
   Desc string        `form:"desc" json:"desc"`
   Asc  string        `form:"asc" json:"asc"`
}
```

2、 客户端发送 json 格式数据,注意必须使用"区分,另外 JSON 格式必须符合规范

```
//#curl -v  -H "content-type:application/json" -d
"{\"pagefrom\":1,\"pagesize\":20}" \
 http://127.0.0.1/test/query
```

3、 服务器接收并进行数据处理返回数据

```
//定制路由映射绑定关系
func (ctrl *TestController)Router(router *gin.Engine){
   r := router.Group("test").Use(ctrl.before())
    r.Any("query",ctrl.query)
 }
//实现绑定
func (ctrl *TestController)query(ctx *gin.Context){
   var pageArg model.PageArg
   ctx.ShouldBindJSON(&pageArg)
   restgo.ResultOk(ctx,pageArg)
}
```

系统返回结果如下

```
* Connected to 127.0.0.1 (127.0.0.1) port 80 (#0)
> POST /test/query HTTP/1.1
> Host: 127.0.0.1
> User-Agent: curl/7.58.0
> Accept: */*
> content-type:application/json
> Content-Length: 28
>
* upload completely sent off: 28 out of 28 bytes
< HTTP/1.1 200 OK
< Content-Type: application/json; charset=utf-8
< Date: Sun, 18 Feb 2018 06:38:39 GMT
< Content-Length: 153
<
{"code":200,"data":{"Kword":"","Datefrom":"0001-01-01T00:00:00Z","Da
teto":"0001-
01-01T00:00:00Z","pagesize":20,"pagefrom":1,"desc":"","asc":""},"msg
":""}
```

5.6 参数校验

参数校验主要用于校验前端提交参数的合法性和合理性,当前有很多开源校验框架,事实上 gin 集成了 go-playground/validator.vx 作为校验插件，当前最新版本是 v9，详细文档见 https://godoc.org/gopkg.in/go-playground/validator.v9

go-playground/validator.vx 功能非常强大，不但内置了丰富的校验方法如 email/url/base64/required 等，用户也可以自定义校验。

```
//使用内置校验示例
func (ctrl *TestController)query(ctx *gin.Context){
   var pageArg model.PageArg

   ctx.ShouldBindJSON(&pageArg)
   err := validator.New().Struct(&pageArg)
   if err != nil {

      // this check is only needed when your code could produce
      // an invalid value for validation such as interface with nil
      // value most including myself do not usually have code like this.
      if _, ok := err.(*validator.InvalidValidationError); ok {
         fmt.Println(err)
         return
      }

      for _, err := range err.(validator.ValidationErrors) {

         fmt.Println(err.Namespace())
         fmt.Println(err.Field())
         fmt.Println(err.StructNamespace()) // can differ when a custom
TagNameFunc is registered or
         fmt.Println(err.StructField())    // by passing alt name to
ReportError like below
         fmt.Println(err.Tag())
         fmt.Println(err.ActualTag())
         fmt.Println(err.Kind())
         fmt.Println(err.Type())
         fmt.Println(err.Value())
         fmt.Println(err.Param())
         fmt.Println()
      }

      // from here you can create your own error messages in whatever language
you wish
      //return
   }
   restgo.ResultOk(ctx,err)

}
```

校验的其他示例可以参考

https://github.com/go-playground/validator/tree/v9/_examples

事实上,我们推荐使用自定义校验方案,因为自定义校验处理更灵活,交互体验更友好。

具体示例如下

```
type PageArg struct {
   Kword string `form:"kword"`
   Datefrom time.Time `form:"datefrom" time_format:"2006-01-02 15:04:05"`
   Dateto time.Time   `form:"dateto" time_format:"2006-01-02 15:04:05"`
   Pagesize int       `form:"pagesize" json:"pagesize"`
   Pagefrom int       `form:"pagefrom" json:"pagefrom"  validate:"gte=0"`
   Desc string        `form:"desc" json:"desc"`
   Asc  string        `form:"asc" json:"asc"`
}

func (p* PageArg)Validate() (bool,error){
   if p.Datefrom.IsZero() {
      return false,errors.New("请输入开始时间")
   }
   if p.Pagesize>100 {
      return false,errors.New("一次只能请求 100 条数据")
   }
   if p.Pagefrom<0 {
      return false,errors.New("分页参数错误")
   }
   return true,nil
}
//请求示例
ctx.ShouldBindJSON(&pageArg)
bindok,err := pageArg.Validate()
if !bindok{
        restgo.ResultFail(ctx,err)
}
```

5.7 数据响应

数据响应包含俩个方面的内容,一是数据格式的封装,一是数据响应类型.
先说数据响格式封装,为了别于运维,一般我们需要将响应结果封装到函数里面,这样前端获取的数据结构统一。对于 api 来说,响应到前端的数据包含如下几个参数

- code：用于指示数据请求状态,200 表示成功
- data:后端服务返回的基础数据,一般是对象
- msg:后端调用返回的操作提示,如恭喜你操作成功或者失败则提示失败原因
- rows:当后端返回是数组时,该参数代表获取的全部记录,该参数可以适配 jquerytable 系列前端框架
- total:当返回数组时,该参数用来标识全部记录数目

在 restgo/Result.go 中我们封装了 4 个函数如下

```
//对于基础响应函数,需要制定返回的 code
func Result(ctx * gin.Context,code int,data interface{},msg string){
   ctx.JSON(http.StatusOK, gin.H{"code": code, "data": data, "msg":msg})
}
//请求成功响应函数，携带数据
func ResultOk(ctx * gin.Context,data interface{}){
   ctx.JSON(http.StatusOK, gin.H{"code": http.StatusOK, "data": data,
"msg": ""})
}
//请求列表成功响应函数,携带数据列表和总页数
func ResultList(ctx * gin.Context,data interface{},total int64){
   ctx.JSON(http.StatusOK, gin.H{"code": http.StatusOK, "rows": data,
"msg": "","total":total})
}
//请求成功响应提示,携带提示和数据
func ResultOkMsg(ctx * gin.Context,data interface{},msg string){
   ctx.JSON(http.StatusOK, gin.H{"code": http.StatusOK, "data": data,
"msg": msg})
}
//请求失败响应函数,携带失败原因
func ResultFail(ctx * gin.Context,err interface{}){
   ctx.JSON(http.StatusOK, gin.H{"code": http.StatusBadRequest, "data":
nil, "msg":err})
}
//请求失败响应函数,携带失败原因和数据
func ResultFailData(ctx * gin.Context,data interface{},err interface{}){
   ctx.JSON(http.StatusOK, gin.H{"code": http.StatusBadRequest, "data":
data, "msg":err})
}
```

数据类型方面,gin 框架内置了各种数据类型响应格式,包括 xml\json\yaml\html 等

1、 xml 格式

```
func main() {
   r := gin.Default()
   r.GET("/someXML", func(c *gin.Context) {
      c.XML(http.StatusOK, gin.H{"message": "hey", "status":
http.StatusOK})
   })
   // Listen and serve on 0.0.0.0:8080
   r.Run(":8080")
}
```

2、 json 格式

```
func main() {
   r := gin.Default()
```

```
   r.GET("/moreJSON", func(c *gin.Context) {
   // You also can use a struct
   var msg struct {
      Name    string `json:"user"`
      Message string
      Number  int
   }
   msg.Name = "Lena"
   msg.Message = "hey"
   msg.Number = 123
   // Note that msg.Name becomes "user" in the JSON
   // Will output  :   {"user": "Lena", "Message": "hey", "Number": 123}
   c.JSON(http.StatusOK, msg)
})
   // Listen and serve on 0.0.0.0:8080
   r.Run(":8080")
}
```

3、 yaml 格式

```
func main() {
   r := gin.Default()
  r.GET("/someYAML", func(c *gin.Context) {
   c.YAML(http.StatusOK, gin.H{"message": "hey", "status":
http.StatusOK})
})
   // Listen and serve on 0.0.0.0:8080
   r.Run(":8080")
}
```

4、 静态资源文件

```
func main() {
   r := gin.Default()
  r.StaticFile("/favicon.ico", "./resources/favicon.ico")   // Listen
and serve on 0.0.0.0:8080
   r.Run(":8080")
}
```

5、 html 文件

```
func main() {
   router := gin.Default()
   router.LoadHTMLGlob("templates/*")
   //router.LoadHTMLFiles("templates/template1.html",
"templates/template2.html")
   router.GET("/index", func(c *gin.Context) {
      c.HTML(http.StatusOK, "index.tmpl", gin.H{
         "title": "Main website",
      })
```

```
    })
    router.Run(":8080")
}
view/index.html
{{templete  "index.html"}}
<html>
<h1>
{{ .title }}
</h1>
</html>
{{end}}
```

5.8 异常处理

异常处理在路由章节以及做了详细描述,这里不做详细阐述。

5.9 controller 示例

该章节我们以用户管理这一需求为例,阐述如何进行控制器编程。
首先我们分析用户管理需要的接口如下

用户管理模块 api 描述				
编号	请求格式	接口描述	请求数据	响应数据
1	/user/query	根据姓名、电话等注册时间等 最后登陆时间、角色等搜索和统计用户		
2	/user/findOne	根据用户编号获取基础信息		
3	/user/login	根据用户名、密码等进行登录操作		
4	/user/register	用户注册操作		
5	/user/resetpwd	用户重置密码		
6	/user/edit	编辑用户基础信息		
7	/user/profile	用户个人中心页面		

我们以接口 1 为例说明

1、新建控制器,在 controller 文件夹下新建控制器 UserController.go 内容如下

```
package controller
import (
    "restgo/restgo"
)
type UserController struct {
```

```
   restgo.Controller
}
```

2、添加路由映射规则，对 UserController 结构体扩展如下方法

```
func (ctrl *UserController)Router(router *gin.Engine){
    r := router.Group("user")
    r.Any("query",ctrl.query)
   }
```

3、实现 query 方法

```
func (ctrl *UserController)query(ctx *gin.Context){
   var userArg model.UserArg
   ctx.ShouldBindJSON(&userArg)
   //下面根据 userArg 拼接 sql 查询条件获取数据
   //var data = ...
   //最后响应数据列表到前端
   //restgo.ResultList(ctx,data,120)
}
```

4、重点说明:关于查询用的参数,我们不建议对参数名称采用特殊的处理,我们建议封装到一个结构体里面，如关于用户信息的查询参数,我们封装到 UserArg 结构体中。

```
type PageArg struct {
   Kword string `form:"kword"`
   Datefrom time.Time `form:"datefrom" time_format:"2006-01-02 15:04:05"`
   Dateto time.Time   `form:"dateto" time_format:"2006-01-02 15:04:05"`
   Pagesize int       `form:"pagesize" json:"pagesize"`
   Pagefrom int       `form:"pagefrom" json:"pagefrom"  validate:"gte=0"`
   Desc string        `form:"desc" json:"desc"`
   Asc  string        `form:"asc" json:"asc"`
}
type UserArg struct {
   PageArg
   ttype string `form:"ttype" json:"ttype"`
//用于区分是注册时间还是登陆时间
}
//使用时候
```

当使用时,我们使用如下方法判断

```
var userArg model.UserArg

ctx.ShouldBindJSON(&userArg)
if len (userArg.Kword)>0{

}
```

6 模型 M 和 ORM

熟悉 java 中 ssm 框架的应该清楚,和数据库操作相关的有 entity、dao、以及 service 层,entity 将数据库表结构和 java 对象关联起来,而 dao 专门用来处理对数据库的基本操作,service 层封装了具体的业务逻辑。他们结构清晰,作用明确。我们借鉴 java 框架,可以将 entity、dao、service 封装到一层,这一层,我们称之为模型层。
另一方面,对于数据库处理方面,我们希望用一个对象或者结构体完成对某一个表的操作,我们不愿意关注表的具体名字或者表内字段命名等细节;我们也不愿意去拼接很多 sql 语句,我们更不愿意将数据库操作返回的数组和字段进行 mapper 映射,我们希望这一切,用一个框架可以自动处理好。ORM 框架正是我们需要的，模型层和 ORM 切切相关。
目前我们 golang 里面的 ORM 框架有很多,xorm，gorm,以及 beegoorm,本文以 xorm 为例集成 ORM 框架。

6.1 使用 xorm

本文对 XORM 入门信息不做详细阐述, 详细知识请查阅 xorm 文档

https://www.kancloud.cn/kancloud/xorm-manual-zh-cn/56013

```
#go get github.com/go-xorm/xorm
```

典型的 xorm 使用方式如下

```
import (
_ "github.com/go-sql-driver/mysql"
"github.com/go-xorm/xorm"
)

var engine *xorm.Engine

func main() {
   var err error
   engine, err = xorm.NewEngine("mysql", "root:123@/test?charset=utf8")
}
```

为了支持多数据库引擎,我们需要对配置文件进行配置

```
#默认数据资源配置,数据库驱动类型为 mysql,详情可以参开 xorm
restgo.datasource.default.driveName=mysql
#连接串
restgo.datasource.default.dataSourceName=root:root@/restgo?charset=utf8
#连接池中 idle 态链接最大个个数
restgo.datasource.default.maxIdle=10
#连接池最大打开连接数
restgo.datasource.default.maxOpen=5
#是否显示 sql 语句
```

```
restgo.datasource.default.showSql=true

#支持配置多个数据库
restgo.datasource.ds1.driveName=mysql
#连接串
restgo.datasource.ds1.dataSourceName=root:root@/restgo2?charset=utf8
#连接池中 idle 态链接最大个个数
restgo.datasource.ds1.maxIdle=10
#连接池最大打开连接数
restgo.datasource.ds1.maxOpen=5
#是否显示 sql 语句
restgo.datasource.ds1.showSql=true
```

在使用数据源 ds1 时,我们通过如下方式调用

```
func (service *UserService)FindOne(userId int64)(entity.User){
   var user entity.User
   orm := restgo.OrmEngin("ds1")
   orm.Id(userId).Get(&user)
   return  user
}
```

当我们使用默认数据源时,我们通过如下方式调用

```
func (service *UserService)FindOne(userId int64)(entity.User){
   var user entity.User
   orm := restgo.OrmEngin()
   orm.Id(userId).Get(&user)
   return  user
}
```

6.2 实体 entity

在 entity 目录下新建 User.go 文件

```
package entity

import "time"

type User struct {
   ID int64 `xorm:"pk autoincr 'id'" json:"id"`
   CreateAt time.Time `xorm:"created" json:"create_at"
time_format:"2006-01-02 15:04:05"`
   Stat int `json:"stat"`
   UserName string `xorm:"varchar(40)" json:"user_name"`
   Passwd string `xorm:"varchar(40)" json:"-"`
   NickName string `xorm:"varchar(40)" json:"nick_name"`
```

```
    Avatar string `xorm:"varchar(180)" json:"avatar"`
}
```

具体细节请查阅 xorm 文档
https://www.kancloud.cn/kancloud/xorm-manual-zh-cn/56013

6.3 实现 service

在 service 目录中创建 UserService.go 文件其中内容如下所示

```
package service

import (
    "restgo/entity"
    "restgo/restgo"
)

type UserService struct {}
//根据 userId 获取用户编号
func (service *UserService)FindOne(userId int64)(entity.User){
    var user entity.User
    orm := restgo.OrmEngin("ds1")
    orm.Id(userId).Get(&user)
    return  user
}
```

上述定义了根据 ID 获取用户信息的服务

6.4 高级查询

对于部分比较特殊的服务,比如根据关键字、创建时间等查询获取用户信息,我们需要设计便于维护的数据结构,我们建议将查询条件封装到一个结构体中，具体操作如下
1、在 model 目录下定义用户信息查询条件结构体 UserArg.go

```
package model
type UserArg struct {
    PageArg
    ttype string `form:"ttype" json:"ttype"`
}
```

其中 PageArg 结构体是一个公用的结构体,它定义了常用的查询条件如时间范围 Datafrom,Dateto，关键字 Kname,以及分页起始页 pagefrom,分页大小 pagesize，asc 排序字段，desc 排序字段等。

```
type PageArg struct {
    Kword string `form:"kword"`
```

```
    Datefrom time.Time `form:"datefrom" time_format:"2006-01-02 15:04:05"`
    Dateto time.Time   `form:"dateto" time_format:"2006-01-02 15:04:05"`
    Pagesize int       `form:"pagesize" json:"pagesize"`
    Pagefrom int       `form:"pagefrom" json:"pagefrom"  validate:"gte=0"`
    Desc string        `form:"desc" json:"desc"`
    Asc  string        `form:"asc" json:"asc"`
}
```

2、在 UserService.go 中定义通用查询方法 Query

```
func (service *UserService)Query(arg model.UserArg)([]entity.User){
    var users []entity.User = make([]entity.User , 0)
    orm := restgo.OrmEngin("ds1")
    t := orm.Where("id>0")
    if (0<len(arg.Kword)){
        t = t.Where("name like ?","%"+arg.Kword+"%")
    }

    if (!arg.Datefrom.IsZero()){
        t = t.Where("create_at >= ?",arg.Datefrom)
    }
    if (!arg.Dateto.IsZero()){
        t = t.Where("create_at <= ?",arg.Dateto)
    }
    t.Limit(arg.GetPageFrom()).Find(&users)
    return  users
}
```

3、在 UserController.go 中调用 Query 方法

```
func (ctrl *UserController)query(ctx *gin.Context){
  var userArg model.UserArg
  ctx.ShouldBind(&userArg)
  ret := userService.Query(userArg)
  //最后响应数据列表到前端
  restgo.ResultList(ctx,ret,1024)
}
```

7 视图 V

当前 web3.0 时代,前后端分离已经成为主流。后端方面,restful 风格 api 大行其道;在前端上，各种 js 框架如 vue,reactjs，anglarjs 百花齐放。但是 golang 的视图层却是由后端渲染的,这和 java 类似,正因为这个原因,golang 适合做一些安全性要求较高的工作。本章节主要阐述 golang

框架视图相关的配置.

7.1 视图配置

视图主要有三个配置参数

```
#视图存放路径
restgo.view.path=view
#视图中模板标签开始标记
restgo.view.deliml={{
#视图中模板标签结束标记
restgo.view.delimr=}}
```

这些参数都将在启动时加载到服务器并进行解析

7.2 前后端分离

前后端分离主要工作在于俩个方面,一方面不用后端渲染,另一方面是劲量将系统参数模块化管理。具体起来有如下几个方面

1、将公用参数通过 js 的形式渲染，如下所示,js 文件有/public/userinfo.shtml 提供,这是一个后端提供的页面服务,我们可以将用户信息存放到该文件中

```
<!DOCTYPE html>
<html>
  <head>
     <script src="/public/userinfo.shtml"></script>
  </head>
  </head>
  <body class="hold-transition skin-blue sidebar-mini">
  //这里是内容
  </body>
</html>
```

2、采用 html 编写前端,并独立部署,二者通过 api 进行交互。这种方式要求非常高，需要在开始的时候将接口定下来。

3、前后端分离是必然趋势,因为随着大数据时代崛起,前端将呈现多样化,而后端要求高性能和高并发，必然要求前后端分离。

8 模板

8.1 模板基础语法

模板基本语法不是本文的重点，本章节只阐述常用基本语法,其他语法请自行网络查阅相关知识。

8.2 在模板中使用自定义函数

我们需要将自动以函数统一管理起来,这个管理模块在 restgo/Func.go 中,该模块已经内置了 ctxpath、version 等常用方法,那么如果需要定制一个新的方法，该怎么做呢？以 hello 方法为例

要使用自定义 hello 函数,首先需要在 restgo/Func.go 中添加 hello 函数,如下

```
func init(){
   restFuncMap["ctxpath"]=ctxpath
   restFuncMap["pageurl"]=pageurl
   restFuncMap["apiurl"]=apiurl
   restFuncMap["version"]=version
   restFuncMap["hello"]=hello
}

func GetFuncMap()(template.FuncMap){
   return restFuncMap
}

func hello(d string) string{
   return "hello "+d
}
```

在 view/user 目录下新建 hello.html，我们以如下方式调用该方法

```
{{define "user/hello.html"}}
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    {{hello "restgo"}}
  </body>
</html>
{{end}}
```

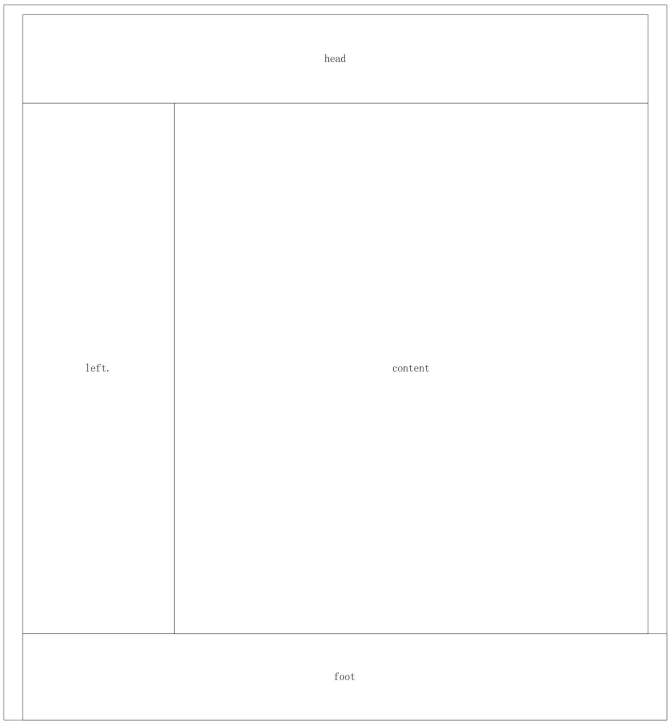
请求结果如下

```
C:\Users\Administrator>curl localhost/user/hello.shtml

<!DOCTYPE html>
<html>
<head>
</head>
<body>
hello restgo
</body>
</html>
```

8.3 包含文件

在 freemarker 或者 jsp 中我们非常熟悉 include 方法,该方法对菜单处理,公用头部文件和尾部文件等公用模板文件处理非常方便和快捷,在 golang 中,也有类似方法,他就是 template。以如下布局为例。



图中 head、left、foot 区域是公共的,不同的是 content 区域,那么我们抽象如下模板

```
//public/head.html
{{define "public/head.html"}}
这里是 head 区域内容
{{end}}

//public/foot.html
{{define "public/foot.html"}}
这里是 foot 区域内容
{{end}}
//public/left.html
{{define "public/left.html"}}
这里是 left 区域内容
{{end}}

//public/content.html
{{define "public/content.html"}}
        <html>
                <head>
```

```
            </head>
            <body>
{{template "public/head.html"}}
{{template "public/left.html"}}
<div class="content">
    这里是具体内容

</div>
{{template "public/foot.html"}}

            </body>
        </html>
{{end}}
```

8.4 静态文件处理

为了便于将来实现动静态分离,我们可以扩展一个 asset 方法,用于制定静态资源路径

```
func init(){
   restFuncMap["ctxpath"]=ctxpath
   restFuncMap["pageurl"]=pageurl
   restFuncMap["apiurl"]=apiurl
   restFuncMap["version"]=version
   restFuncMap["hello"]=hello
   restFuncMap["asset"]=asset
}

func asset(d string) string{
   cfg := GetCfg()
   return cfg.App["protocal"]+"://" + cfg.App["asset"]
}
```

摸板中以如下方式调用

```
<link rel="stylesheet"
href="{{asset}}/assets/plugins/font-awesome/css/font-awesome.min.css">
<!-- Ionicons -->
<link rel="stylesheet"
href="{{asset}}/assets/plugins/ionicons/css/ionicons.min.css">
```

当资源文件迁移时,我们可以直接修改如下配置

```
#静态资源所在的服务器地址,便于动静态分离
restgo.app.asset=localhost
```

9 杂项

9.1 session

Session 模块可选择很多,我们集成了 github.com/tommy351/gin-sessions，在 main 函数中，我们使用如下方法开启 session

```
store := sessions.NewCookieStore([]byte(cfg.Session["name"]))
router.Use(sessions.Middleware(cfg.Session["name"],store))
```

如果需要修改 session 配置,可以修改 app 配置文件

```
#sessionID 标识字符串,对标 PHP 的 SESSIONID,java 的 JSESSIONID
restgo.session.name=GSESSIONID
#session 过期时间以秒为单位,0 表示访问结束时过期
restgo.session.timelive=3600
```

9.2 验证码

我们集成了一个图片验证码,位置 restgo/Captcha.go，该文件提供验证码渲染方法 loadVerify 方法和验证码校验方法 ChecjVerify，该方案基于 session,用户可以自行集成图片验证码方案.

```
func LoadVerify(ctx *gin.Context) {
   d := make([]byte, 4)
   s :=NewLen(4)
   ss := ""
   d = []byte(s)
   for v := range d {
      d[v] %= 10
      ss += strconv.FormatInt(int64(d[v]), 32)
   }
   session := sessions.Get(ctx)
   session.Set("___verify",ss)
   session.Save()
   NewImage(d, 100, 40).WriteTo(ctx.Writer)

}
func CheckVerify(ctx *gin.Context,code string) bool{
   session := sessions.Get(ctx)
   v := session.Get("___verify")
   session.Delete("___verify")
   session.Save()
   return v== code
}
```

9.3 缓存

缓存分为内部缓存和第三方缓存机制,对于简单的业务,可以直接使用内部 map 进行缓存，对于比较复杂的系统,建议采用第三方缓存设备.第三方缓存方案很多,典型的如 redis,memcache 等。对于缓存选型需要考虑到的点在于分布式和连接池以及参数配置等因数

- 连接池，go-redis 已经支持连接池
- 参数配置,可以在 app 配置文件中进行配置

```
#redis 服务器地址
restgo.redis.host=localhost
#redis 端口
restgo.redis.port=3306
#redis  密码
restgo.redis.passwd=$^siwe2i23i^(12
#使用的数据库
restgo.redis.db=0
```

通过 LoadCfg 方法获取配置参数

9.4 鉴权

鉴权需要用户根据自己的业务进行扩展,我们已经提供了一个 Demo，位于 restgo/Auth.go

9.5 日志

系统集成了 log4go 日志系统,主要配置参数为 filepath，用来指定配置文件路径

```
#log4g 日志配置文件路径地址
restgo.logger.filepath=config/log4g.xml
```

配置文件初始化在 main 函数中

```
restgo.Configuration(cfg.Logger["filepath"])
```

日志管理模块位于 restgo/Logger.go 中，当前提供 Debug 方法和 Error 方法，用户可以根据需要进行扩充

```
func Debug(arg0 interface{}, args ...interface{}) {
   l4g.Debug(arg0, args)
}
func Error(arg0 interface{}, args ...interface{}) {
   l4g.Error(arg0, args)
}
```

用户也可以使用 gin 提供的日志记录方法

10 项目实战

10.1 restgo 后台管理框架

10.2 天天任务清单小程序

10.3 工业大数据采集

10.4 restgo cms

10.5 restgo 千人大群

11 联系我