**开发者指南**

|  |  |
| --- | --- |
| 版本 | V0.01 |
| 公司： | 杭州一共享科技有限公司 |
| 创建人 | 刘勇俊 |
| 联系电话 |  |
| 机密等级 |  |
|  |  |
|  |  |

版本历史

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 版本号 | 开始日期 | 结束日期 | 作者 | 审核 | 内容 |
| V0.01 | 。 |  | 刘勇俊 |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

# 入门

关于go的学习，入门知识和环境搭建，请参阅以下书籍 ：

Go 语言圣经 <https://books.studygolang.com/gopl-zh/index.html>

Go语言高级编程 <https://books.studygolang.com/advanced-go-programming-book>

# 开发环境

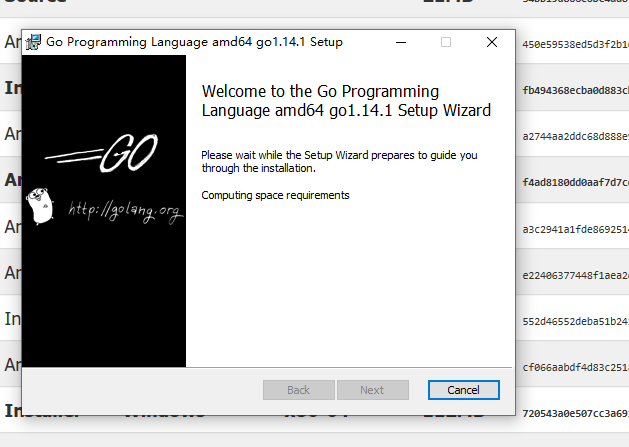
仅以个人本地开发环境为例。生产机器不在此列。

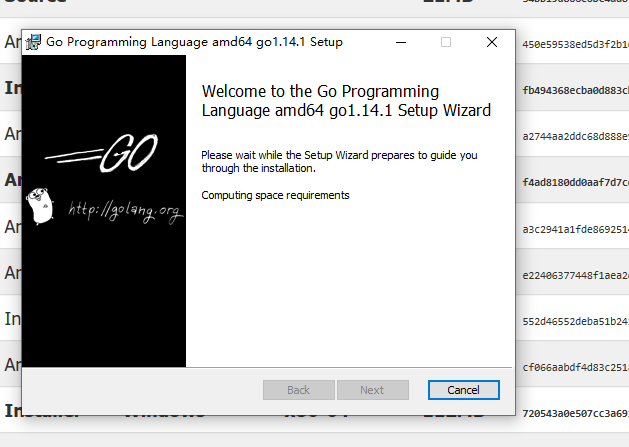
1. Go
2. Win10安装

请参照 <https://studygolang.com/articles/20120?fr=sidebar> 安装

下载地址：<https://studygolang.com/dl/golang/go1.14.1.windows-amd64.msi>

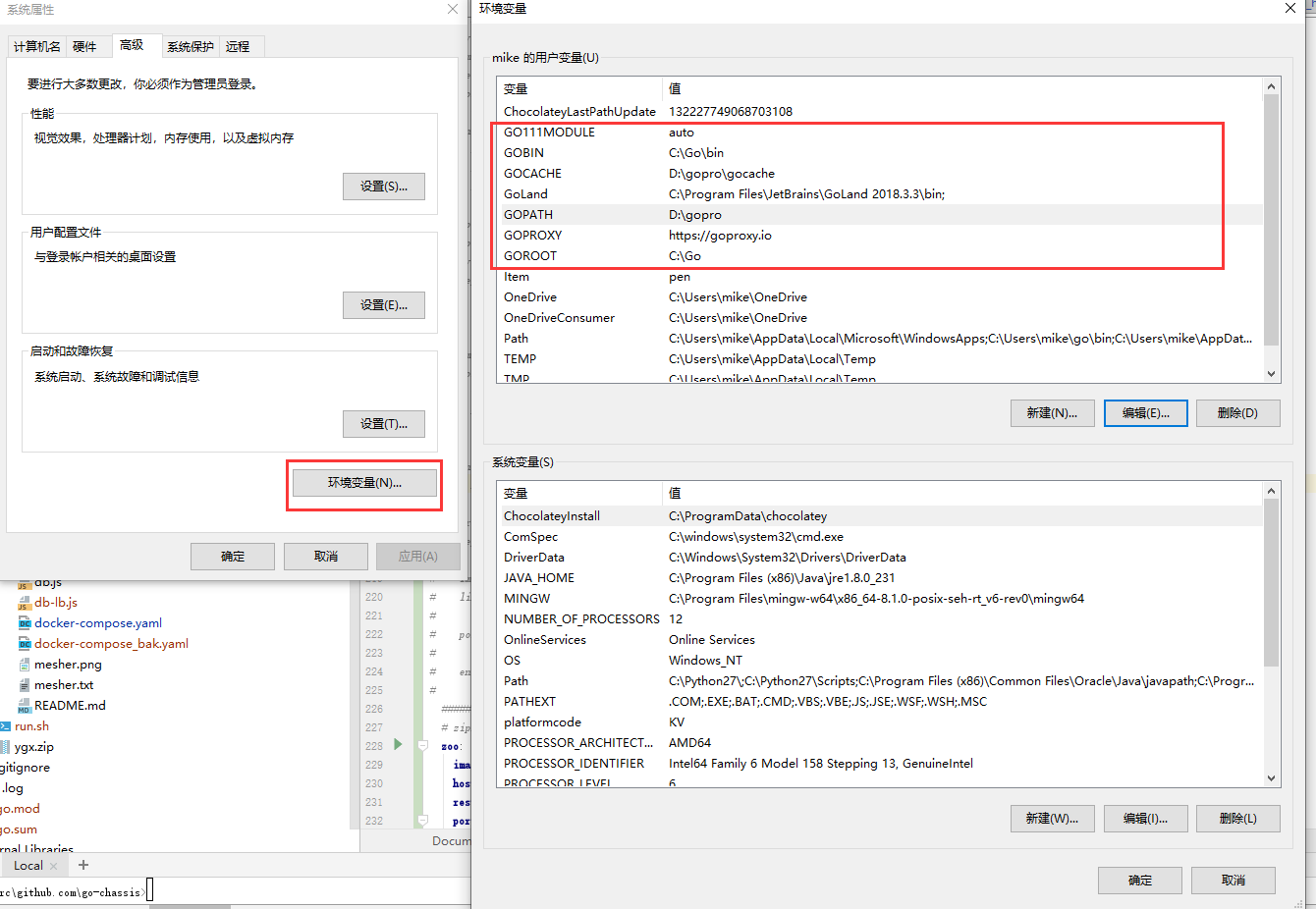
点击 下一步：



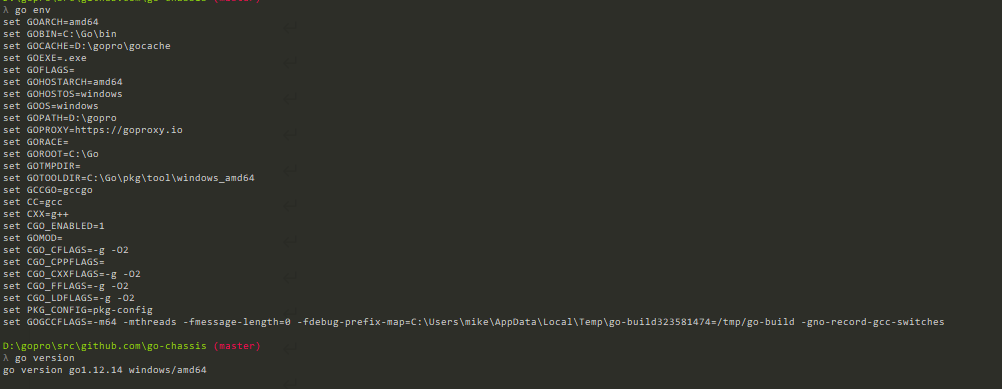


安装到预定的目录 c:/go 下面

环境变量：右击“我的电脑”=>“属性”=>”高级系统设置“ 在弹出的界面设置环境变量如下：



验证环境变量：



1. Centos7安装
2. 安装预置软件

Yun update

yum -y install gcc g++ automake wget

1. 安装go

wget <https://storage.googleapis.com/golang/go1.11.8.linux-amd64.tar.gz>

tar -C /usr/local -zxvf go1.11.8.linux-amd64.tar.gz

echo "export GOPROXY=https://goproxy.io/ " >> /etc/profile

echo "export GOROOT=/usr/local/go " >> /etc/profile

echo "export GOPATH=/gopath " >> /etc/profile

echo "export PATH=$PATH:$GOROOT/bin " >> /etc/profile

source /etc/profile

1. Docker
2. win10

在<https://hub.docker.com/editions/community/docker-ce-desktop-windows>

下载docker， 双击然后按界面“下一步”即可以安装好。

详细过程如下：

* 1. docker 下载

wget <https://storage.googleapis.com/golang/go1.11.8.linux-amd64.tar.gz>

tar -C /usr/local -zxvf go1.11.8.linux-amd64.tar.gz

echo "export GOPROXY=https://goproxy.io/ " >> /etc/profile

echo "export GOROOT=/usr/local/go " >> /etc/profile

echo "export GOPATH=/gopath " >> /etc/profile

echo "export PATH=$PATH:$GOROOT/bin " >> /etc/profile

source /etc/profile

首先先到Docker官网下载最新官方Docker for Windows：[Docker下载](https://docs.docker.com/docker-for-windows/install/" \l "download-docker-for-windows" \t "_blank)   
在官网内可以查看到Docker的开发文档。   
根据官网提示，Windows环境下下载Docker必须满足：Docker for Windows requires 64bit Windows 10 Pro and Microsoft Hyper-V（即需要64位Windows 10 Pro(专业版和企业版都可以)和Microsoft Hyper-V），Hyper-V是微软的虚拟机，在win10上是自带的，我们只需要启动就可以了。

* 1. 启动 hyper-v

wget <https://storage.googleapis.com/golang/go1.11.8.linux-amd64.tar.gz>

tar -C /usr/local -zxvf go1.11.8.linux-amd64.tar.gz

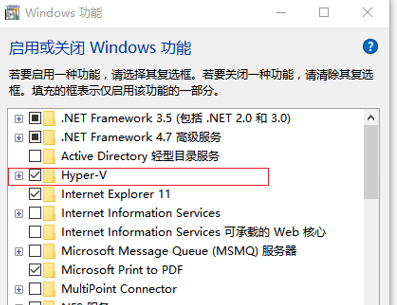
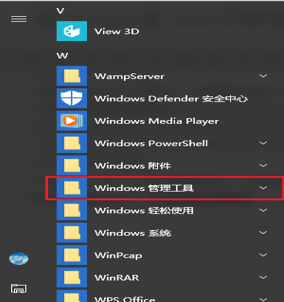
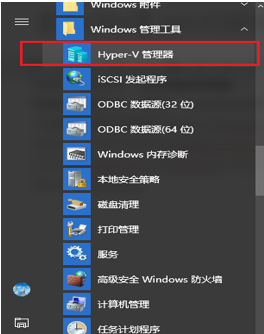
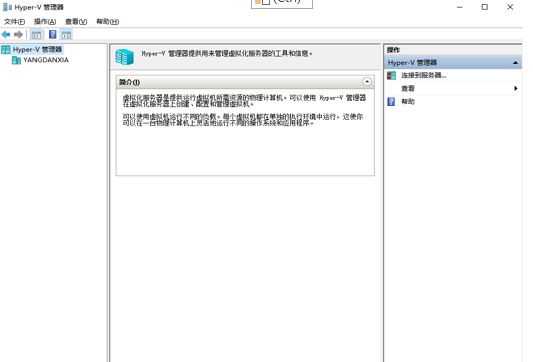
echo "export GOPROXY=https://goproxy.io/ " >> /etc/profile

echo "export GOROOT=/usr/local/go " >> /etc/profile

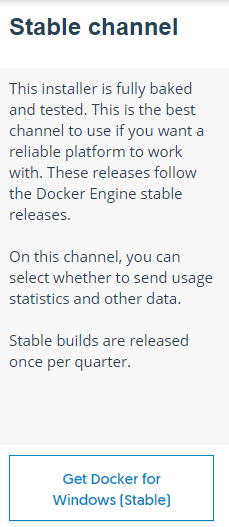
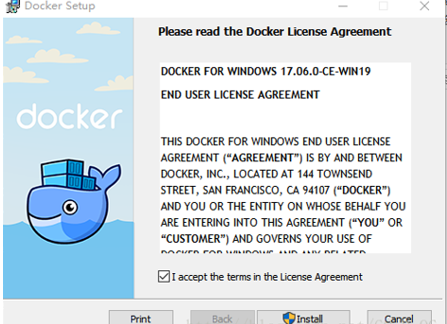
echo "export GOPATH=/gopath " >> /etc/profile

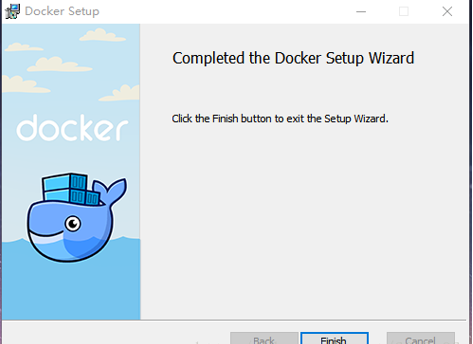
echo "export PATH=$PATH:$GOROOT/bin " >> /etc/profile

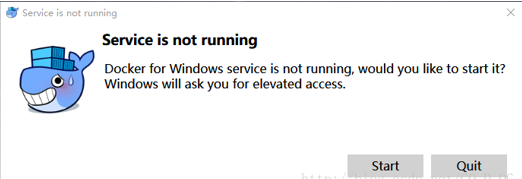
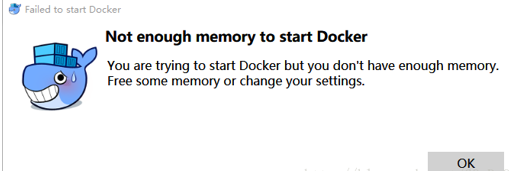
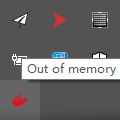
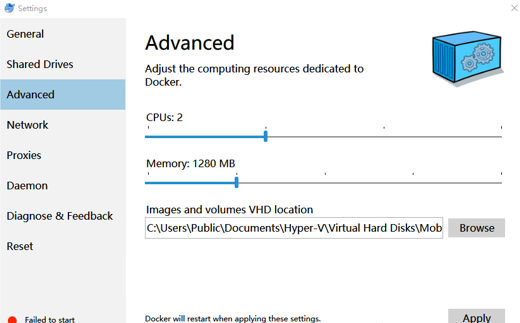
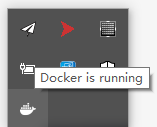
source /etc/profile

在电脑上打开“控制面板”->“程序”-> “启动或关闭Windows功能”。   
点击红框内容：   
  
如下，勾选Hyper-V选项：   
  
点击确定后，系统会自行更新，更新完成后提示是否重启电脑，要选择**立即重启电脑**。重启电脑后，左键点击win图标，查看电脑全部应用，找到“windows管理工具”：   
  
点开可以看到有一个Hyper-V管理器则表示已成功打开Hyper-V，点击可以查看管理器内容：   
  


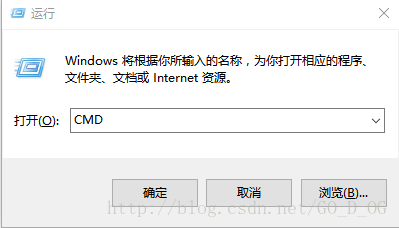
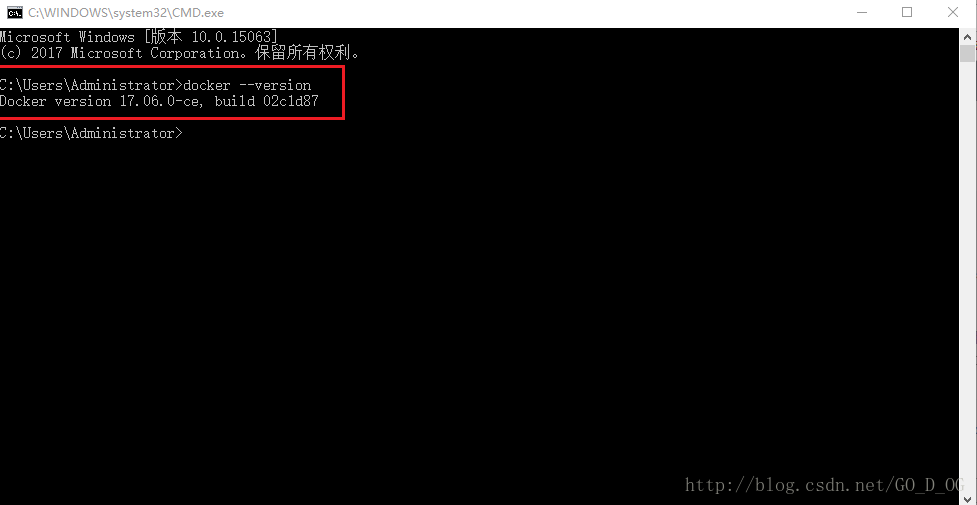
* 1. 安装docker

我下载的是稳定版：   
  
等待安装文件下载好后双击，会弹出：   
  
勾选接受后点击Install，会自动进行安装：   


安装成功后会显示：   
  
可以看到桌面上多了一个Docker for Windows图标。

在安装完成后，Docker会自行启动，但我的电脑半天都没有反应，我在任务管理器内看到Docker是有打开，但是就是运行不起来，所以我把电脑重新启动了，重新启动后弹出了下面的提示：   
  
在等待启动Docker时，弹出错误：Not enough memory to start Docker，无法启动Docker:   
  
意思是”您正在尝试启动Docker，但您没有足够的内存，释放一些内存或更改您的设置”。   
在任务栏找到Docker的图标：   
  
右键点击该图标，找到“settings….”，点击进去后如图修改Docker的内存，由2048MB改为1280MB，然后点击Apply，Docker会自动重启。   
  
现在可以看到Docker已经在运行了   


* 1. 运行docker

对于Docker的操作是用命令行进行操作的，所以我们需要先打开命令行工具（或者下载其他工具如PowerShell等）。键盘输入：win+R，打开运行工具，输入cmd,打开命令行工具：   
  
可以先尝试查看我们的Docker版本：   


1. win7

下载路径:

<https://github.com/docker/toolbox/releases>

<https://get.daocloud.io/toolbox/>

下载版本 DockerToolbox-18.09.0.exe

Win7如果使用最新的docker版本，可能报错，错误为: msg="Unable to use system certificate pool: crypto/x509: system root pool is not available on Windows"

启动的时候,会提示需要下载 最新版本的boot2docker.iso

Dockser 会一直卡在这里，

手动下载https://github.com/boot2docker/boot2docker/releases

boot2docker.iso v19.03.5

然后将文件放在：C:\Users\Administrator\.docker\machine\cache 再启动docker 就ok了.

如此本地环境即ok.

Win7 除注意上述情况外， 其他安装过程与win10 相同。此处不一一列出。

1. 注意

hyper-v 与 vmware虚拟机在docker 打开的时候是不能够共存的，否则会引起电脑蓝屏。

可以在hyper-v打开的时候将vmware关闭。

使用vmware 的时候将hyper-v关闭。

1. 服务器运行

上述环境安装完成，下载源码，解压之后进入源码文件中，即

进入 github.com\go-chassis\ygx\vm 文件下面

执行命令 docker-compose up ，服务器即开始运行。

服务注册中心端口为 30100，前端运维面板：<http://127.0.0.1:30103/#!/sc/dashboard>

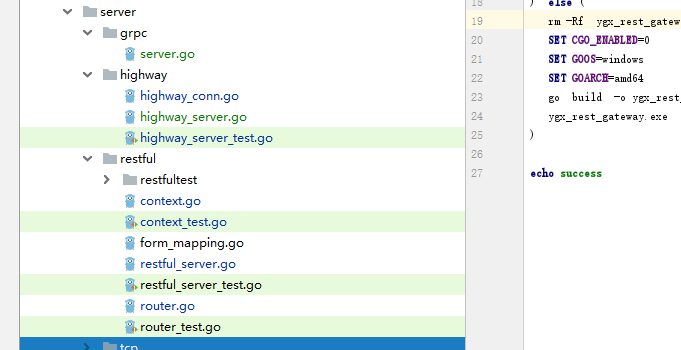
# 架构图



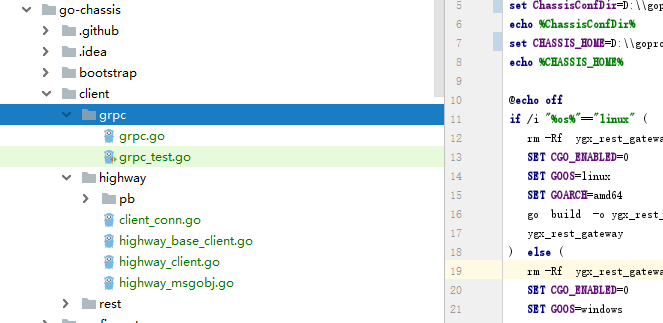
# 框架修改

1. 协议支持

本系统基于 **chassis v1.76** 开发。系统里面grpc 和 rpc 协议支持的代码不存在。下载相关版本后，需要在server添加分别添加下面的代码：



在client 下面添加，下面代码.



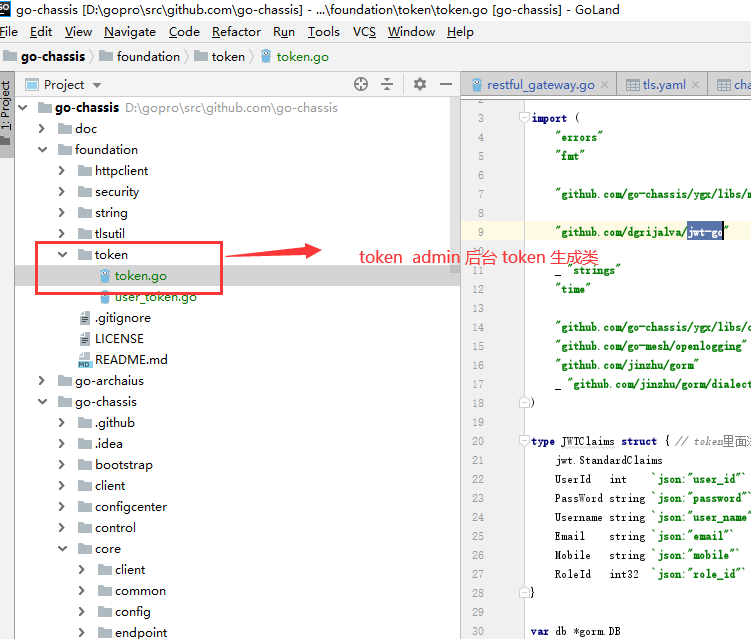
1. restful跨域

框架内restful 中https 请求http，如果端口不同，必须做跨域处理。修改service/restful/context.go代码，实现跨域



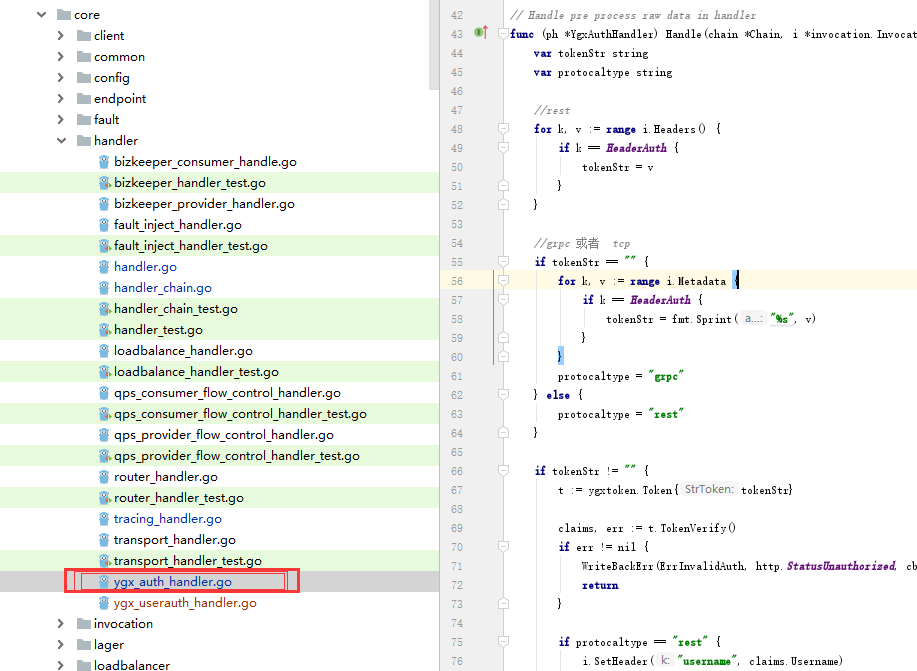
1. Token

新加token 类用来生成验证后端用户的token：



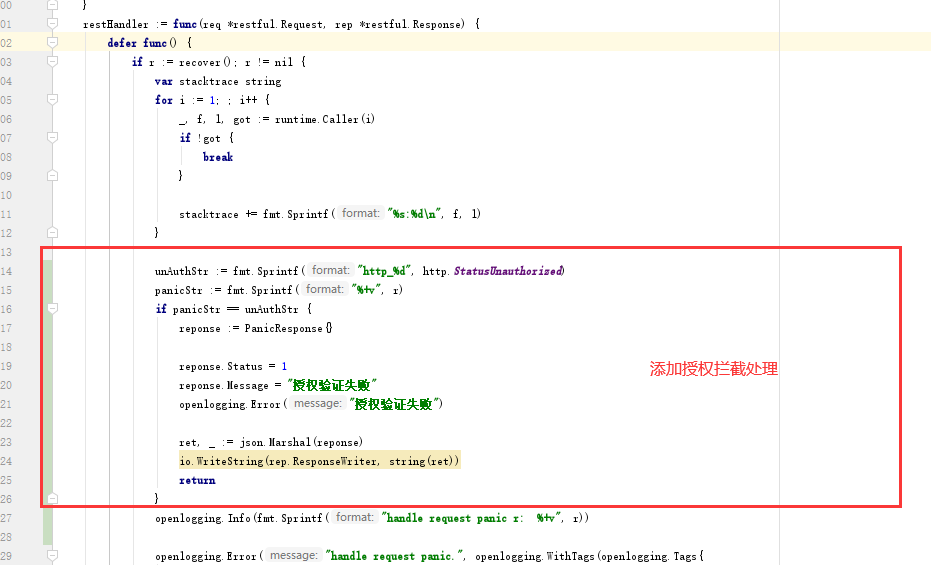
1. 授权handle

新加授权



ygx\_auth\_handler.go

在route 下面添加授权失败的拦截处理



# 配置

1. 服务定义
2. 概述

实例：一个流程是一个微服务实例，实例属于一个微服务

服务：服务是存储中的静态信息实体，它有实例

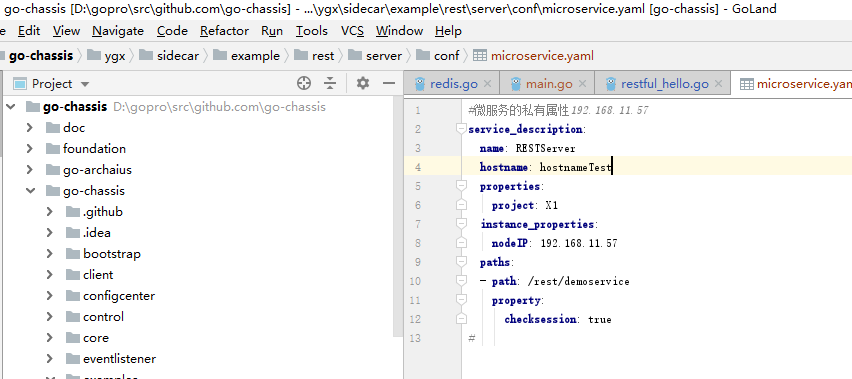
您可以将项目视为一个微服务，在编译、构建和运行之后，它就变成了一个微服务实例

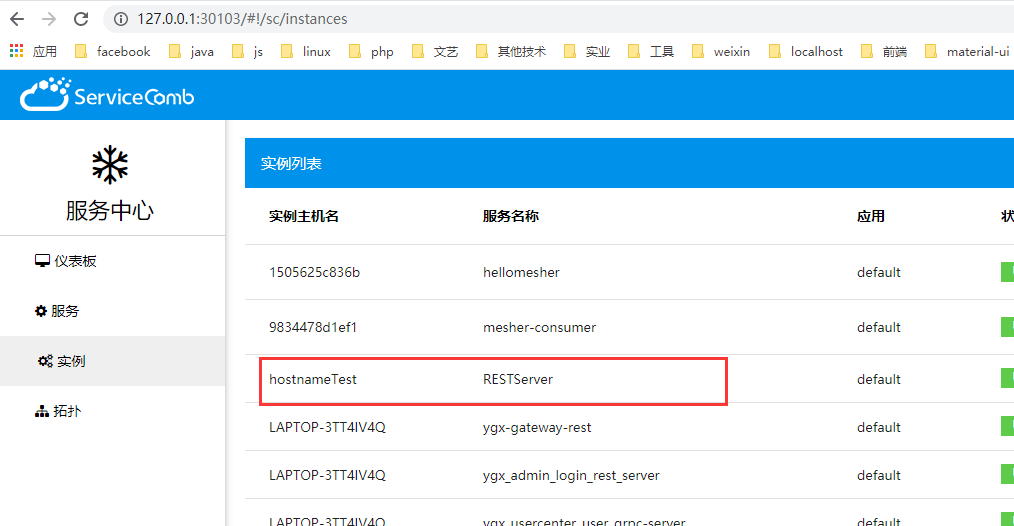
1. 配置

配置文件：microservice.yaml

参数：name : 微服务名字

hostname : 主机名 代码中os.hostname（）获取





Application\_id ： 对应上图应用名称

Version: 版本

Properties ： 传递的元数据，一般是不变的工程常量 代码中存在于 metadata

Instance\_Properties: 根据环境传递的元数据，可变的，代码中存在于 metadata

Paths ：作用未知 ？

对于我们自己的项目，只需要配置前面的3-4 参数

Example:

service\_description:

name: RESTServer

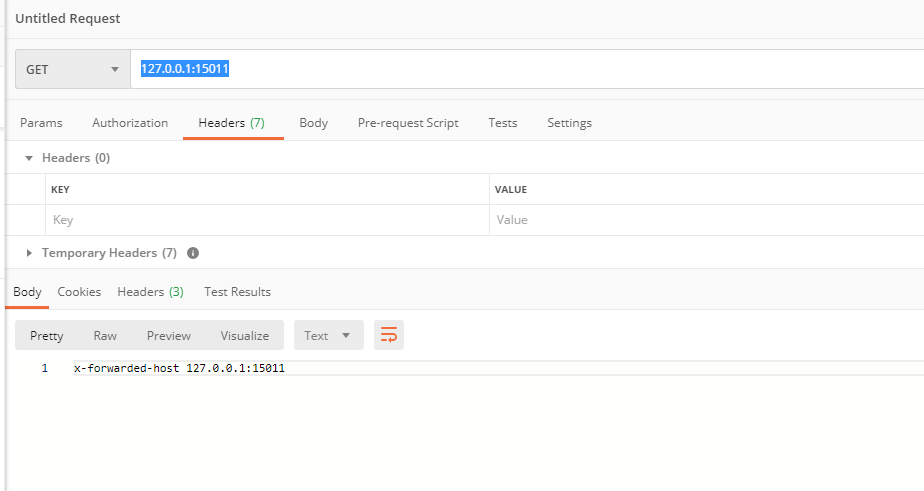
hostname: hostnameTest

1. 示例代码

位置： github.com\go-chassis\ygx\sidecar\example\rest

进入server文件下，在命令行输入:run.bat 按回车。既可以看到运行实例输出的日志。

然后在postman 工具中输入：



1. 服务注册
2. 概述

微服务的注册发现默认通过[服务中心](https://github.com/apache/servicecomb-service-center)完成。 用户可以配置与服务中心的通信方式，服务中心地址，以及自身注册到服务中心的信息。 微服务启动过程中，会自动向服务中心进行注册。 在微服务运行过程中，go-chassis会周期从服务中心查询其他服务的实例信息缓存到本地

1. 配置

配置文件：chassis.yaml

type: 对接的服务中心类型默认加载servicecenter和file两种，registry也支持用户定制registry并注册。

scope: 默认不允许跨应用间访问，只允许本应用间访问，当配置为full时则允许跨应用间访问，且能发现本租户全部微服务。

register: 配置项默认为自动注册，即框架启动时完成实例的自动注册。当配置manual时，框架只会注册配置文件中的微服务，不会注册实例，使用者可以通过服务中心对外的API完成实例注册。

api.version: 目前只支持v4版本。

Disabled (optional, bool) 是否开启服务注册发现模块，默认为false

Type (optional, string) 对接服务中心插件类型，默认为servicecenter

Scope (optional, bool) 默认为full，允许跨app发现，填入app以禁止跨应用发现

Address \*(optional, bool)\*服务中心地址 允许配置多个以逗号隔开，默认为空

Register (optional, bool) 是否自动自注册，默认为 auto，可选manual

refreshInterval

(optional, string) 更新实例缓存的时间间隔，格式为数字加单位（s/m/h），如1s/1m/1h，默认为30s

api.version (optional, string) 访问服务中心的api版本，默认为v4

watch (optional, bool) 是否watch实例变化事件，默认为false

Example:

**APPLICATION\_ID**: default *#optional*

**cse**:

**service**:

**registry**:

**disabled**: **false #optional**: 默认开启registry模块

**type**: **servicecenter #optional**: 默认类型为对接服务中心

**scope**: **full #optional**: scope为full注册时允许跨app

**address**: [http://10.0.0.1:30100, http://10.0.0.2:30100](http://10.0.0.1:30100,%20http://10.0.0.2:30100)

#支持多个servicecenter

**register**: auto *#optional：默认为自动 [auto manual]*

**refeshInterval** : 30s

**watch**: true

**api**:

**version**: v4

1. 示例代码

本项目： type 统一是servicecenter，在vm 文件夹下面配置。Disabled开启服务发现 Scope 使用默认跨app发现（即不同的链之间可以调用）， watch 配置为true,当配置文件更新时，系统内参数实时更新，体现为go-archaius 类配置内存更新 address 为服务中心地址， 查看服务中心配置。

同服务定义。

1. 服务发现
2. 概述

Service discovery 是关于如何发现服务的配置。 和Registry的区别是，他仅负责发现服务，而不负责注册服务 Service Discovery与Registry只能选择其一进行配置 启用此功能可以与Istio的Pilot集成.

1. 配置

配置文件chassis.yaml

和Registry的区别是，他仅负责发现服务，而不负责注册服务 Service Discovery与Registry只能选择其一进行配置 启用此功能可以与Istio的Pilot集成。

参数type:

默认为servicecenter，另外可选择pilotv2以及kube .

Address : \*(optional, bool)\*服务中心地址 允许配置多个以逗号隔开，默认为空

refreshInterval : 更新实例缓存的时间间隔，格式为数字加单位（s/m/h），如1s/1m/1h，默认为30s

disabled (optional, bool) 是否开启服务注册发现模块，此处应该为true

当registry type为pilotv2时需要指定pilot的地址address，当registry type为kube时需要指定与kube-apiserver交互所需的kubeconfig的配置文件位置，以下分别为

registry的最小示例。

**cse:**

**service:**

**Registry:**

**serviceDiscovery:**

**type: pilotv2**

**address: grpc://istio-pilot.istio-system:15010**

**refeshInterval : 30s**

**cse:**

**service:**

**Registry:**

**serviceDiscovery:**

**type: kube**

**configPath: /etc/.kube/config**

1. 示例代码

本项目： type 统一是servicecenter,使用服务注册，没有使用服务发现功能。

1. 协议服务器
2. 概述

可以在go-chassis中扩展自己的协议，目前支持rest（http）和gRPC, gRPC需要自己集成。

1. 配置

配置文件：chassis.yaml

Protocals : 可以配置协议 rest grpc

listenAddress : 监听example地址

example

**cse:**

**protocols:**

**rest:**

**listenAddress: 0.0.0.0:5000**

**rest-admin:**

**listenAddress: 0.0.0.0:5001**

**grpc:**

**listenAddress: 0.0.0.0:6000**

然后你可以使用后缀“admin”作为访问rest-admin服务器的端口。

req,\_:=rest.NewRequest("GET","http://RESTServer:admin/sayhello/world")

1. 示例代码

Grpc 协议demo：github.com\go-chassis\ygx\sidecar\example\grpc

进入server和client文件下，分别在命令行输入:run.bat 按回车。既可以看到运行实例输出的日志。

Restful 协议参考服务定义示例代码。

1. 处理链
2. 限流器
3. 概述：

用户可以通过配置限流策略限制Provider端或Consumer端的请求频率，使每秒请求数限制在最大请求量的大小。其中Provider端的配置可限制接收处理请求的频率，Consumer端的配置可限制发往指定微服务的请求的频率。

1. 配置：

限流配置在rate\_limiting.yaml中，同时需要在chassis.yaml的handler chain中添加handler。其中qps.limit.[service] 是指限制从service 发来的请求的处理频率，若该项未配置则global.limit生效。Consumer端不支持global全局配置，其他配置项与Provider端一致。配置项默认值配置说明

flowcontrol.qps.enabled true是否开启流控模块。

flowcontrol.qps.global.limit 2147483647（max int）流控全局配置。

flowcontrol.qps.limit.[service] 2147483647（max int）每秒允许的请求数。

Example

**handler:**

**chain:**

**Provider:**

**default: ratelimiter-provider**

**flowcontrol:**

**Provider:**

**qps:**

**enabled: true # enable rate limiting or not 是否开启限流，默认true**

**global:**

**limit: 1 # default limit of provider 每秒允许的请求数，默认2147483647max int）**

**limit:**

**someConsumerService: 1 # rate limit for request from a consumer 针对某微服务每秒允许的请求数 ，默认2147483647max int）**

**ygx-gateway-rest: 1**

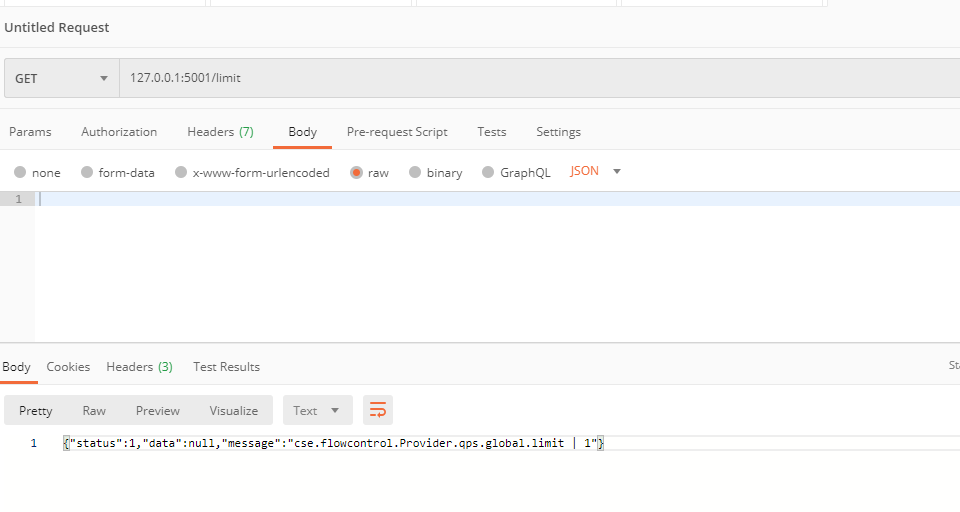
1. 实例代码

位置：github.com\go-chassis\ygx\sidecar\ratelimite

进入server文件下，在命令行输入:run.bat win按回车。既可以看到运行实例输出的日志

测试结果： 当一秒超过1次的时候，系统提

postman测试：



1. 熔断器
2. 概述

当系统出现问题的时候，熔断器有助于在运行时隔离上游服务，保证下游服务的正常运行，保证系统不因一个服务的原因出现雪崩。

1. 配置：

cse.circuitBreaker.scope: 服务、实例或api，默认为api，gochassis为每个api创建一个专用链路，调用是基api隔离。如果设置为service，则每个服务的所有api共享一个链路，它将隔离该服务。如果设置为实例，则每个实例将获得一个专用链路，它将只隔离一个实例。如果设置为api，每个api将得到一个专用链路，它将只隔离服务api。如果设置为instance api，则每个实例api将获得一个专用链路，它将只隔离一个实例api。

cse.isolation.timeoutInMilliseconds： 当请求延迟一段时间，此调用将被视为失败，默认值为30000， 从而触发熔断。

cse.isolation.maxConcurrentRequests：最大并发，缺省1000毫秒。

cse.circuitBreaker.sleepWindowInMilliseconds ：熔断发生时候，多久尝试再请求通过，如果尝试失败，将继续熔断，缺省30000毫秒.

cse.circuitBreaker.enabled : 是否可以熔断

cse.circuitBreaker.forceOpen：true 强制打开熔断

cse.circuitBreaker.requestVolumeThreshold:10秒内有多少请求失败，熔断器打开，缺省20.

cse.circuitBreaker.errorThresholdPercentage：失败百分率为多少， 熔断器打开，缺省50.

cse.fallback.enabled：失败回调

cse.fallbackpolicy.policy：回调策略，返回null,抛出异常

Example

**---**

**cse:**

**isolation:**

**Consumer:**

**timeoutInMilliseconds: 1000 # 请求延迟1000毫秒 词请求将视为失败**

**maxConcurrentRequests: 100 # 最大并发100**

**circuitBreaker:**

**Consumer:**

**enabled: false**

**forceOpen: false**

**forceClosed: false**

**sleepWindowInMilliseconds: 10000 # 熔断发生时候，10000毫秒尝试再请求通过，如果尝试失败，将继续熔断**

**requestVolumeThreshold: 20 # 20秒内有多少请求失败，熔断器打开，缺省20.**

**errorThresholdPercentage: 1 # 百分之一的请求失败，熔断器打开，**

**#容错处理函数，目前暂时按照开源的方式来不进行区分处理，统一调用fallback函数**

**fallback:**

**Consumer:**

**enabled: true**

**fallbackpolicy:**

**Consumer:**

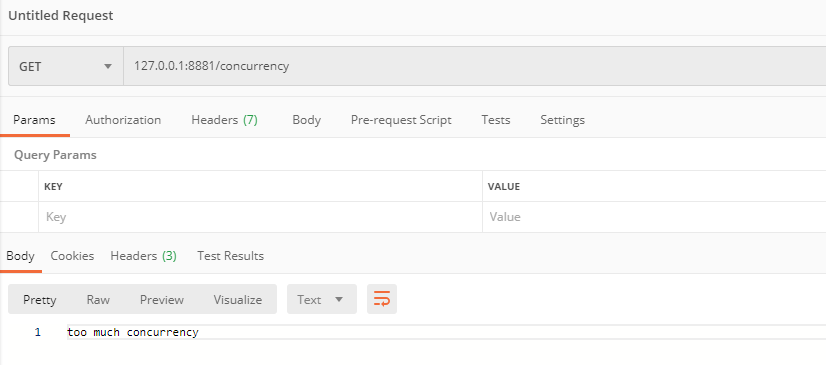
**policy: throwexception**

1. 实例代码

位置：github.com\go-chassis\ygx\sidecar\circuit

进入server文件下，在命令行输入:run.bat win 按回车。既可以看到运行实例输出的日志

postman测试：



1. 链路追踪
2. 概述

Go-chassis使用 zipkin 进行链路追踪。链路追踪就是采集各服务之间互相调用的信息：谁调用了谁，调用是否发生故障，调用耗时多少,方便快速定位服务故障点。

注意:

set GO111MODULE=on

set GOPROXY=direct (https://proxy.golang.org)

需要安装 go get github.com/openzipkin-contrib/zipkin-go-opentracing@v0.3.5

不能够大于 v0.3.5版本 之后版本有较大的出入

**go get gopkg.in/jcmturner/gokrb5.v7**

**go get github.com/pierrec/lz4**

**go get github.com/go-logfmt/logfmt**

**go get github.com/gogo/protobuf/proto**

**go get github.com/klauspost/compress/zstd**

**go get github.com/eapache/queue**

**go get github.com/hashicorp/go-uuid**

**go get github.com/jcmturner/gofork**

**go get golang.org/x/crypto/md4**

**go get golang.org/x/crypto/pbkdf2**

**go get gopkg.in/jcmturner/aescts.v1**

**go get gopkg.in/jcmturner/dnsutils.v1**

**go get gopkg.in/jcmturner/rpc.v1/mstypes**

**go get gopkg.in/jcmturner/rpc.v1/ndr**

原来链路追踪的代码：

<https://github.com/go-chassis/go-chassis-plugins/tree/master/tracing/zipkin> 但是里面的代码需要加载的包 github.com/openzipkin-contrib/zipkin-go-opentracing 只能够用 v0.3.5 否则会报错

现在移到系统中位置 **github.com/go-chassis/go-chassis/plugins/tracing/zipkin**

1. 配置

相关的配置文件Chassis.yaml 和 monitoring.yaml

Example

**handler:**

**chain:**

**Provider:**

**default: tracing-provider**

在monitoring.yaml

**cse:**

**metrics:**

**apiPath: /metrics # we can also give api path having prefix "/" ,like /adas/metrics**

**enable: true**

**enableGoRuntimeMetrics: true**

**flushInterval: 10s**

**tracing:**

**tracer: zipkin # tracing-provider,bizkeeper-provider zipkin**

**settings:**

**URI: http://127.0.0.1:9411/api/v1/spans**

**batchSize: 1**

**batchInterval: 1s**

1. 实例代码

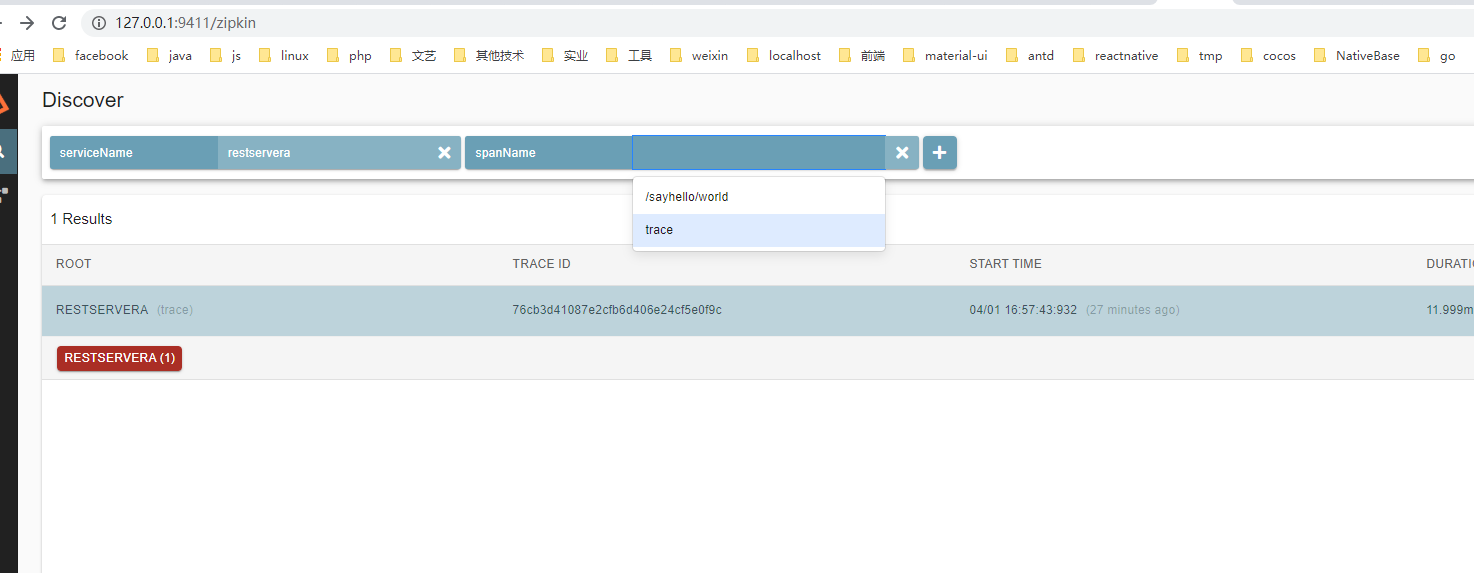
相关代码示例：github.com\go-chassis\ygx\sidecar\example\monitoring

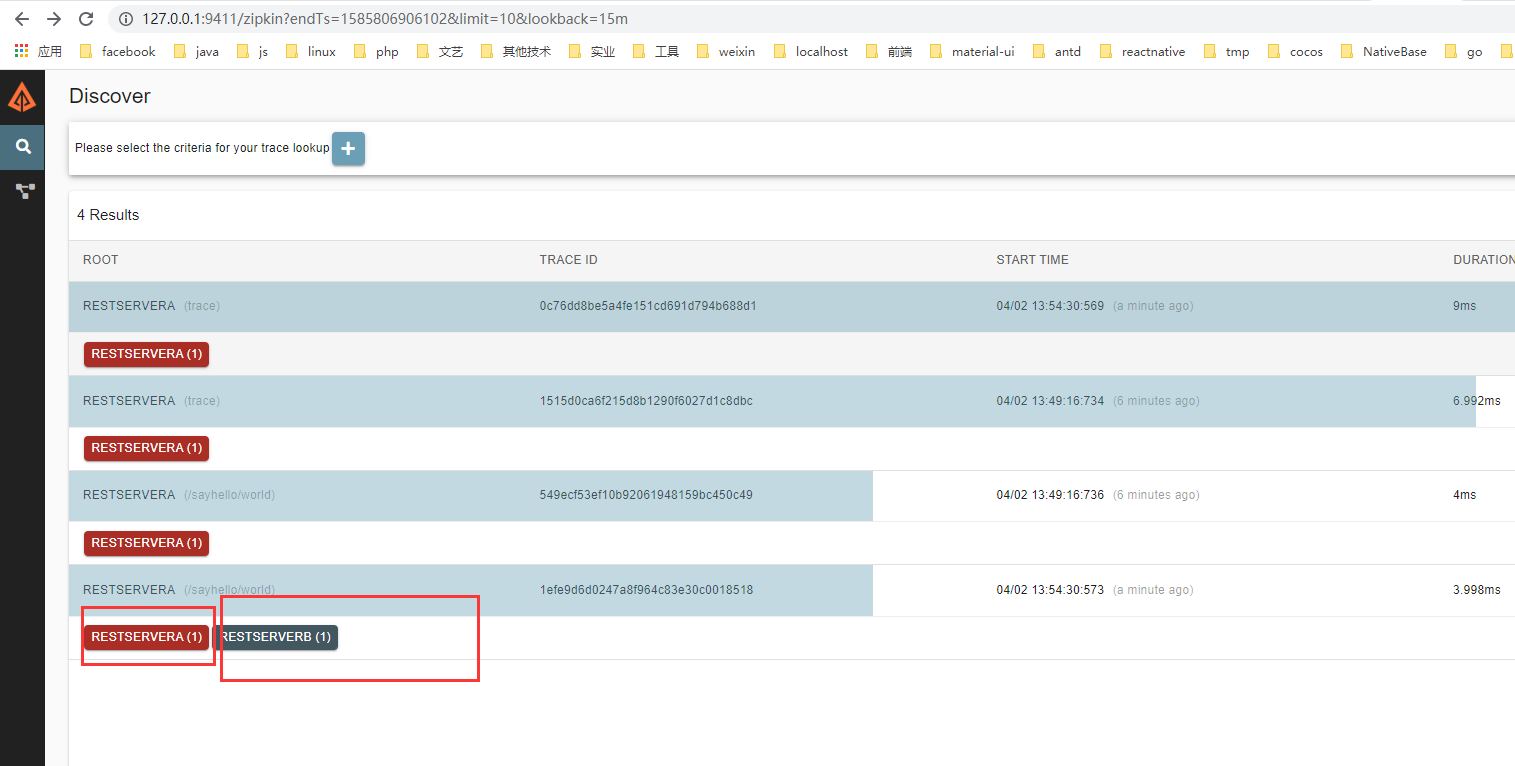
进入server文件下，在命令行输入:run.bat 按回车。既可以看到运行实例输出的日志

postman测试：

图像化查看性能指标：<http://127.0.0.1:9411/zipkin>

<http://127.0.0.1:15021/metrics>





1. Metric 指标统计
2. 概述

通常用于计算接口调用频率，如QPS(每秒的次数)，存在5种指标监控。

Gauges ：最简单的度量指标，只有一个简单的返回值，或者叫瞬时状态。

用于对瞬时值的测量，如我们可以过一段时间就对内存的使用量进行统计，并上报，那么所有的数据点集就是对应时间点的内存值.

Counters：Counter 就是计数器，即计数类统计，可以进行加或减，也可以进行归零操作，所有的操作都是在旧值的基础上进行的．这里可以通过每天归零，然后新增注册用户时加1来统计每天的注册用户。

Meters：Meter度量一系列事件发生的速率(rate)，如：RateMean 表示单位时间内发生的次数，如一分钟发送100次，则该值为100/60.

Histograms：Histogram统计数据的分布情况。比如最小值，最大值，中间值，还有中位数，75百分位, 90百分位, 95百分位, 98百分位, 99百分位, 和 99.9百分位的值(percentiles)。

Timer其实是 Histogram 和 Meter 的结合， histogram 某部分代码/调用的耗时， meter统计TPS。

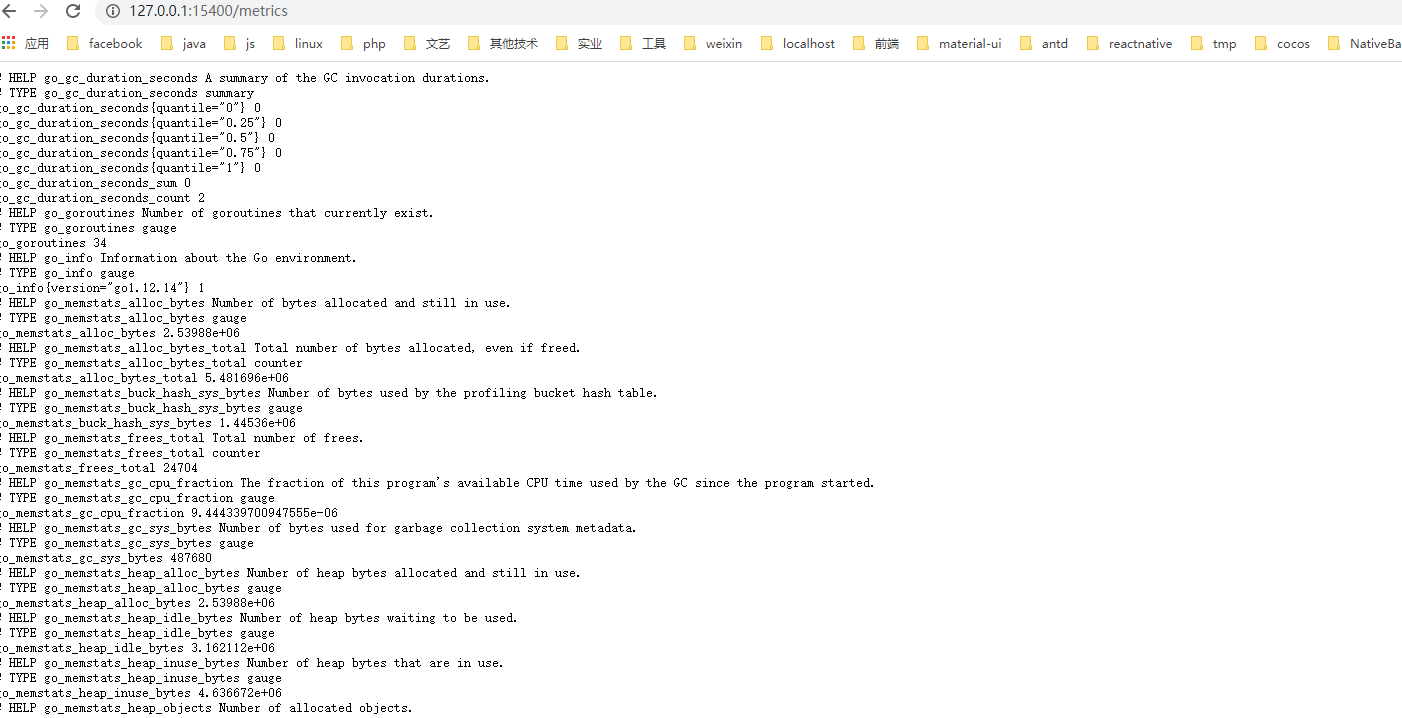
指标情况显示： <http://服务名称/metrics>

1. 实例代码

相关代码示例：github.com\go-chassis\ygx\sidecar\example\monitoring

进入server文件下，在命令行输入:run.bat 按回车。既可以看到运行实例输出的日志

postman测试：



1. Route （灰度部署）
2. 概述

路由策略可应用于AB测试场景和新版本的灰度升级，主要通过路由规则来根据请求的来源、目标服务、Http Header及权重将服务访问请求分发到不同版本的微服务实例中。比如通过路由管理你可以简单的实现新老版本服务切换。

1. 配置

配置文件

router.yaml

参数注释

servicecomb:

routeRule:

{targetServiceName}: |# 服务名

- precedence: {number} #优先级

match: #匹配策略

source: {sourceServiceName} #匹配某个服务名

headers: #header匹配

{key0}:

regex: {regex}

caseInsensitive: false # 是否区分大小写，默认为false，区分大小写

{key1}

exact: {=？}

route: #路由规则

- weight: {percent} #权重值

tags:

version: {version1}

app: {appId}

- precedence: {number1}

match:

refer: {sourceTemplateName} #参考某个source模板ID

route:

- weight: {percent}

tags:

version: {version2}

app: {appId}

1. 实例代码

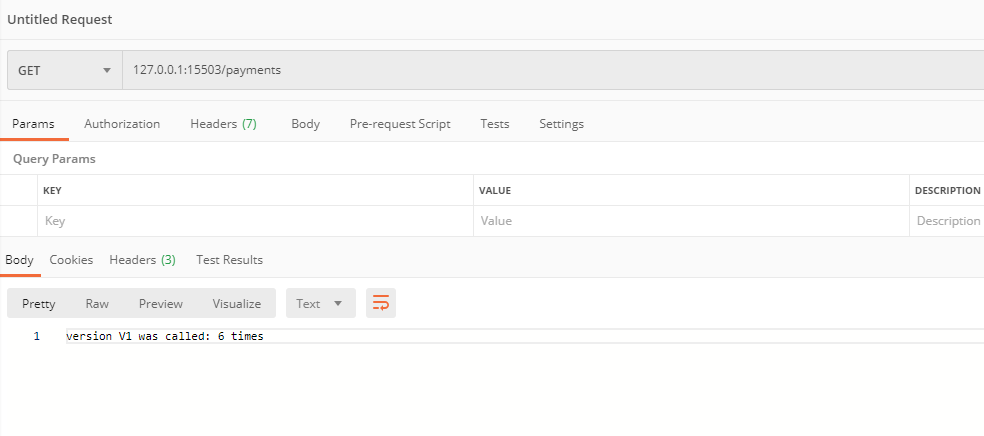
相关代码:

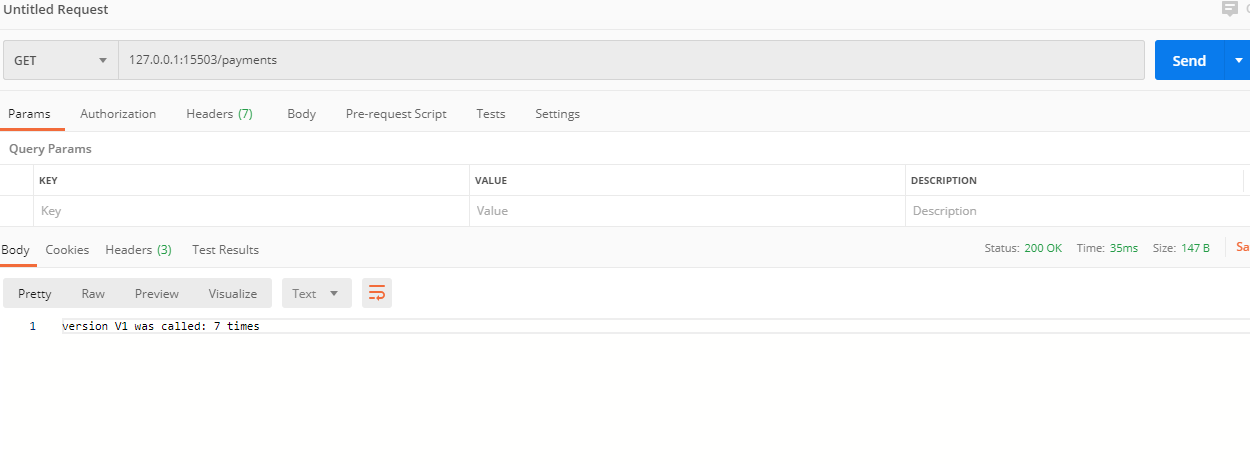
\github.com\go-chassis\ygx\sidecar\example\router\servicecomb\client

分别运行 serverV1，serverV2,client 三个文件夹下面的run.bat

命令 run.bat win

然后再postman 中输入如下路径请求，结果如下



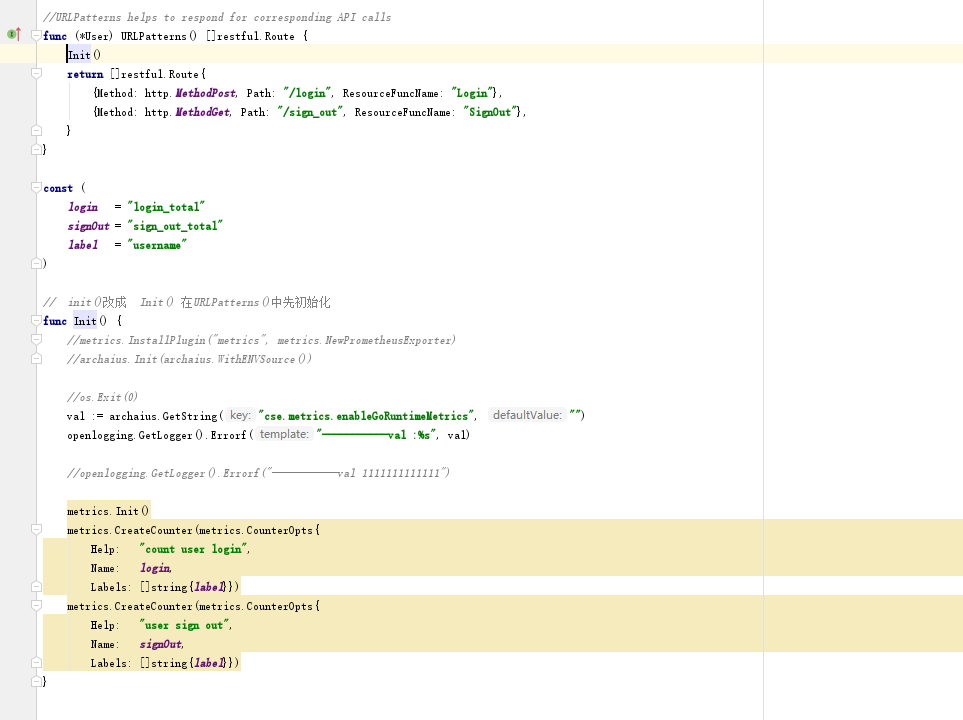


如果正式环境中，新版本的api接口开始调用只占5%， 如果有错就查看日志。

1. 注意事项

在代码中加 \_ **"github.com/go-chassis/go-chassis/plugins/tracing/zipkin"**

代码中原来是init () 这样会有错的。因为archaius 还没有初始化, 修改为下面这样.



1. Load balancing （负载均衡）
2. 概述

用户可以通过配置选择不同的负载均衡策略，当前支持轮询、随机、基于响应时间的权值、会话保持等多种负载均衡策略。

1. 配置

配置文件chassis.yaml

负载均衡的配置项为cse.loadbalance.[MicroServiceName].[PropertyName]，其中若省略MicroServiceName，

则为全局配置；若指定MicroServiceName，则为针对特定微服务的配置。优先级：针对特定微服务的配置 > 全局配置。

PropertyName字段

strategy.name (optional, bool) RoundRobin|策略，可选值：RoundRobin,Random, SessionStickiness, WeightedResponse。

策略的含义：使用SessionStickiness策略，配置即可以使用会话策略，可以在metadata将namespace传入以控制不同请求会话保持 如

\*options = core.InvokeOptions{

Metadata: map[string]interface{}{

common.SessionNameSpace: "go-chassis",

},

}

使用 WeightedResponse策略，启用后30s 策略会计算好数据并生效，80%左右的请求会被发送到延迟最低的实例里

配置chassis.yaml的负载均衡部分，以及添加处理链。

cse:

loadbalance: # 全局负载均衡配置

strategy:

name: RoundRobin

microserviceA: # 微服务级别的负载均衡配置

strategy:

name: SessionStickiness

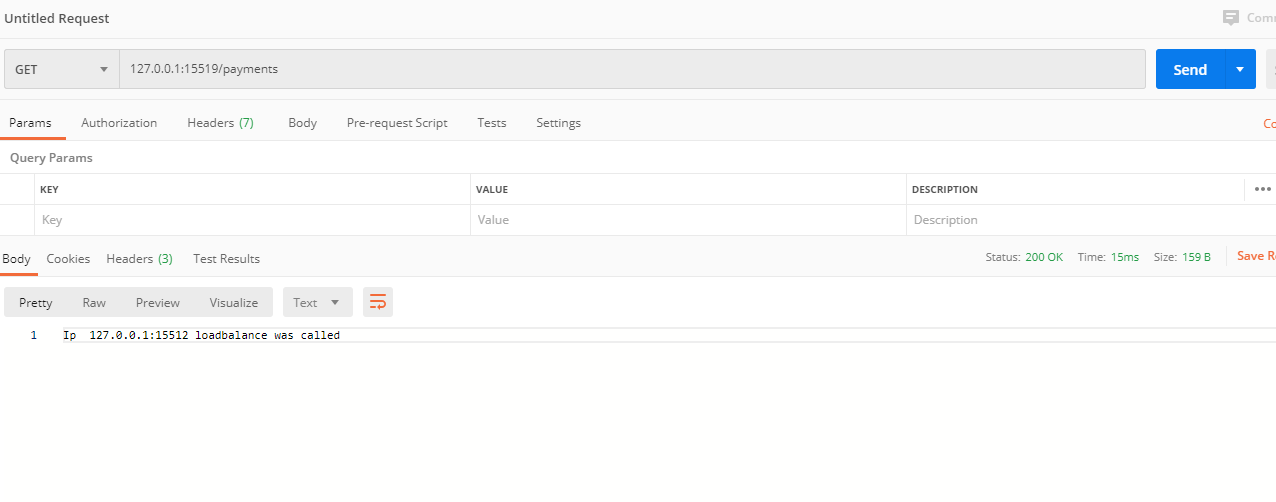
1. 实例代码

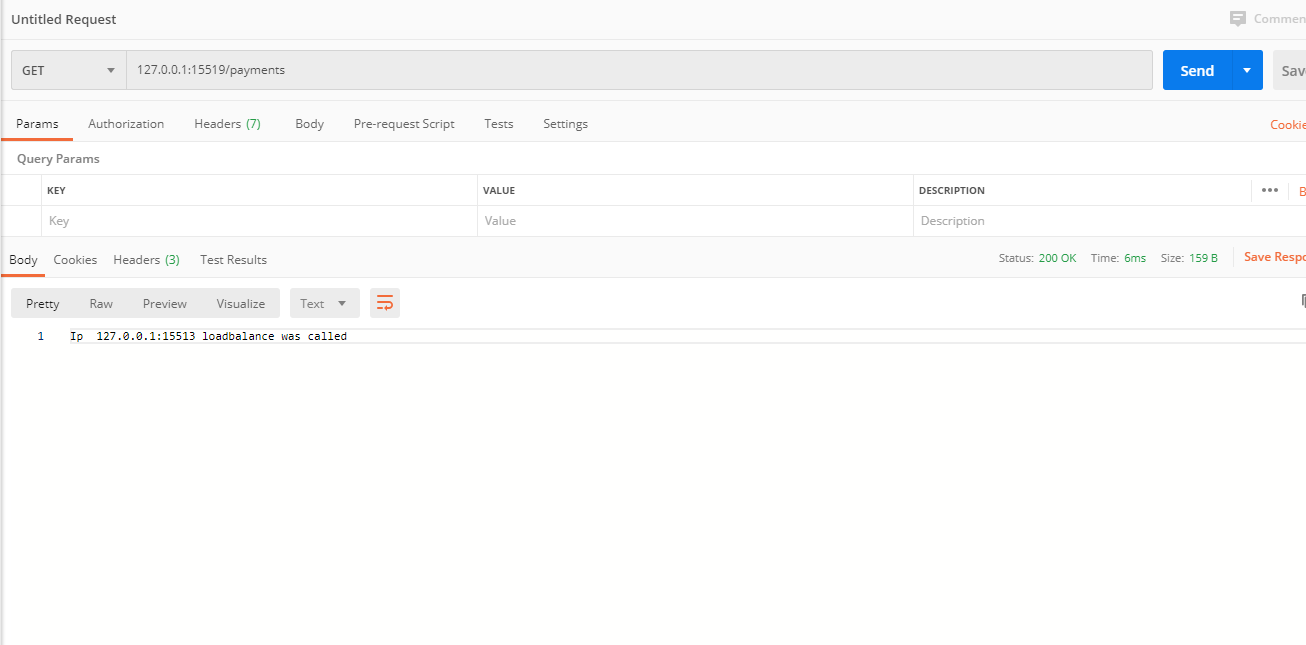
相关代码 \github.com\go-chassis\ygx\sidecar\example\loadbalancing

分别运行 serverA，serverB ,client 三个文件夹下面的run.bat win

命令 run.bat win

然后再postman 中输入如下路径请求，结果如下





**注意：如果负载均衡想使用更多的策略， 可以使用过滤器， 或者按自己的要求写过滤器代码，做有符合自己要求的负载均衡。**

<https://docs.go-chassis.com/user-guides/filter.html>

1. Health check(健康检查)
2. 概述

客户端健康检查（Health Check）是指客户端对服务端实例缓存进行健康性的判断。

在网络分区或延时较大的环境下，客户端可能会出现上报心跳到服务中心失败的情况，导致结果是，客户端会将收到的实例下线事件，并移除本地实例缓存，最终影响业务调用。

1. 配置

go-chassis默认不会主动注册服务端的健康检查接口，需要用户主动import到项目中。

// 注册健康检查接口

import \_ "github.com/go-chassis/go-chassis/healthz/provider" // 是在客户端的main.go 中加

健康检查接口：RESTful:

Method: GET

Path: /healthz

Response:

{

"appId": "string",

"serviceName": "string",

"version": "string"

}

配置文件 chassis.yaml

cse:

service:

Registry:

healthCheck: true

#serviceDiscovery:

# healthCheck: true # 同时支持单独开启服务发现能力时的客户端健康检查

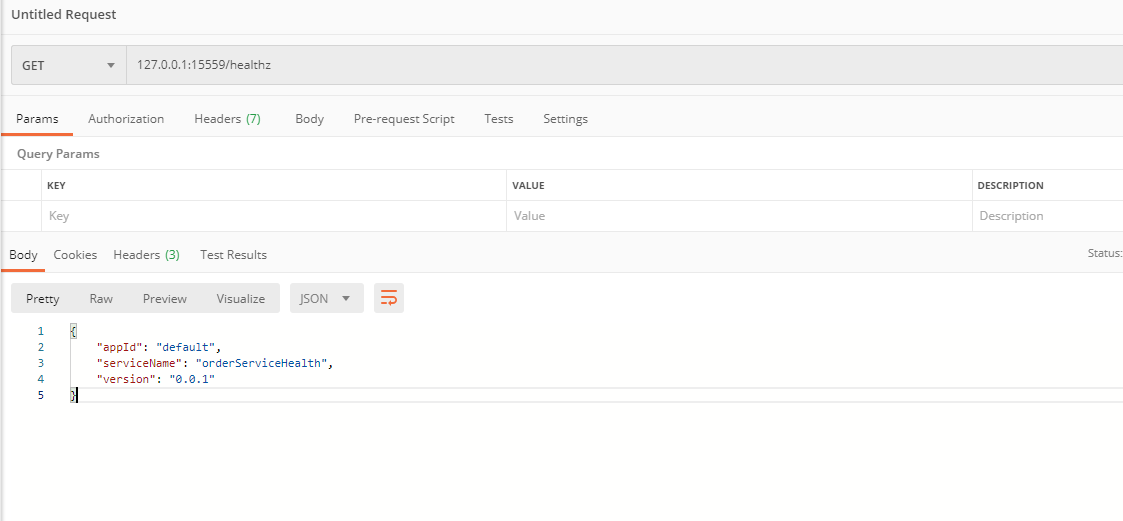
1. 实例代码

相关代码 github.com\go-chassis\ygx\sidecar\example\healthcheck

分别运行 server、client 三个文件夹下面的run.bat

命令 run.bat win

然后再postman 中输入如下路径请求，结果如下



1. 日志
2. 概述

用户可配置微服务的运行日志的相关属性，比如输出方式，日志级别，文件路 径以及日志转储相关属性。

1. 配置

日志配置文件为lager.yaml，配置模板如下：

logger\_level表示日志级别，由低到高分别为 DEBUG, INFO, WARN, ERROR, FATAL 共5个级别，这里设置的级别是日志输出的最低级别，只有不低于该级别的日 志 才会输出。

writers表示日志的输出方式，默认为文件和标准输出。

logger\_file表示日志输出文件。

log\_format\_text: 默认为false，即设定日志的输出格式为 json。若为true则输出格式为plaintext，类似log4j。建议使用json格式输出的日志。

rollingPolicy: 默认为size，即根据大小进行日志rotate操作；若配置为daily则基于事件做日志rotate。

log\_rotate\_date: 日志rotate时间配置，单位”day”，范围为(0, 10)。

log\_rotate\_size: 日志rotate文件大小配置，单位”MB”,范围为(0,50)。

log\_backup\_count: 日志最大存储数量，单位“个”,范围为[0,100)。

示例：

---

writers: file,stdout,file

# LoggerLevel: |DEBUG|INFO|WARN|ERROR|FATAL

logger\_level: DEBUG

logger\_file: log/chassis.log

log\_format\_text: false

#rollingPolicy daily/size

rollingPolicy: size

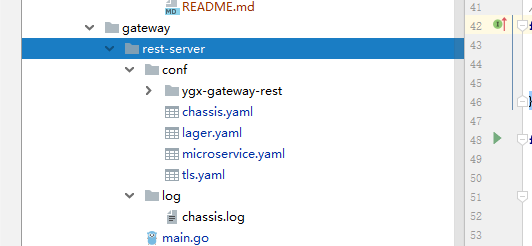
#log rotate and backup settings

log\_rotate\_date: 1

log\_rotate\_size: 10

log\_backup\_count: 7

开发当中使用debug 日志级别，生产环境使用error



1. 服务中心负载均衡

本服务软件是基于docker-compose,来说明和制作。

采用docker-compose的yaml是版本3.

1. etcd 分布式

cmd命令行创建虚拟网卡

*docker network create --subnet=172.20.0.0/16 etcdnetwork*

在docker-compose.yaml中添加网卡配置：

networks:

default:

external:

name: etcdnetwork

在docker-compose.yaml中添加etcd分布式配置：

## docker network create --subnet=172.20.0.0/16 etcdnetwork

######################## 172.20.0.4 etcd3 集群 start#################################

etcd-1:

container\_name: etcd1

image: quay.io/coreos/etcd:v3.3

entrypoint: /usr/local/bin/etcd

command:

- '--name=etcd-1'

- '--initial-advertise-peer-urls=http://etcd-1:2380'

- '--listen-peer-urls=http://0.0.0.0:2380'

- '--listen-client-urls=http://0.0.0.0:2379'

- '--advertise-client-urls=http://etcd-1:2379'

- '--initial-cluster-token=mys3cr3ttok3n'

- '--heartbeat-interval=250'

- '--election-timeout=1250'

- '--initial-cluster=etcd-1=http://etcd-1:2380,etcd-2=http://etcd-2:2380,etcd-3=http://etcd-3:2380'

- '--initial-cluster-state=new'

ports:

- 2379

volumes:

- etcd1:/etcd\_data

networks:

default:

ipv4\_address: 172.20.0.100

## 172.20.0.5

etcd-2:

container\_name: etcd2

image: quay.io/coreos/etcd:v3.3

entrypoint: /usr/local/bin/etcd

command:

- '--name=etcd-2'

- '--initial-advertise-peer-urls=http://etcd-2:2380'

- '--listen-peer-urls=http://0.0.0.0:2380'

- '--listen-client-urls=http://0.0.0.0:2379'

- '--advertise-client-urls=http://etcd-2:2379'

- '--initial-cluster-token=mys3cr3ttok3n'

- '--heartbeat-interval=250'

- '--election-timeout=1250'

- '--initial-cluster=etcd-1=http://etcd-1:2380,etcd-2=http://etcd-2:2380,etcd-3=http://etcd-3:2380'

- '--initial-cluster-state=new'

ports:

- 2379

volumes:

- etcd2:/etcd\_data

networks:

default:

ipv4\_address: 172.20.0.101

# 172.20.0.6

etcd-3:

container\_name: etcd3

image: quay.io/coreos/etcd:v3.3

entrypoint: /usr/local/bin/etcd

command:

- '--name=etcd-3'

- '--initial-advertise-peer-urls=http://etcd-3:2380'

- '--listen-peer-urls=http://0.0.0.0:2380'

- '--listen-client-urls=http://0.0.0.0:2379'

- '--advertise-client-urls=http://etcd-3:2379'

- '--initial-cluster-token=mys3cr3ttok3n'

- '--heartbeat-interval=250'

- '--election-timeout=1250'

- '--initial-cluster=etcd-1=http://etcd-1:2380,etcd-2=http://etcd-2:2380,etcd-3=http://etcd-3:2380'

- '--initial-cluster-state=new'

ports:

- 2379

volumes:

- etcd3:/etcd\_data

networks:

default:

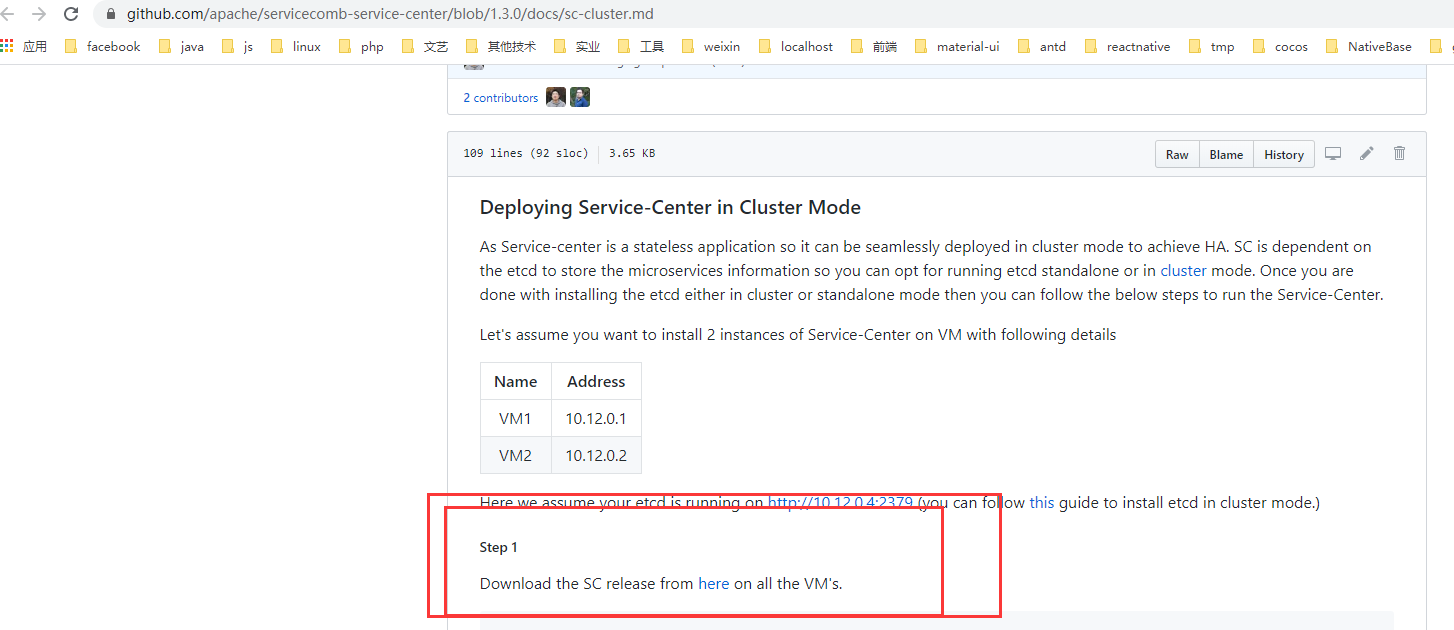
ipv4\_address: 172.20.0.102

######################## 172.20.0.4 etcd3 集群 end#################################

1. 负载均衡镜像

下载节点模式的service-center

<https://github.com/apache/servicecomb-service-center/blob/1.3.0/docs/sc-cluster.md>



制作分布式模式的service-center的dockefile:

#Pull base image. 此处使用centos 也可以直接使用alpine 基础的golang 免去安装golang

FROM centos:7

#下载地址：https://studygolang.com/dl/golang/go1.14.1.windows-amd64.msi

# https://github.com/apache/servicecomb-service-center/archive/1.3.0.tar.gz

# Install Go yum -y update yum -y install wget

RUN \

yum -y install gcc g++ automake && \

mkdir -p /gopath && \

mkdir -p /opt/service-center && \

mkdir -p /download

COPY dockerfile /download/

COPY startbak.sh /download/

COPY go1.11.8.linux-amd64.tar.gz /download/

RUN \

cd /download && \

tar -C /usr/local -zxvf go1.11.8.linux-amd64.tar.gz && \

echo "export GOPROXY=https://goproxy.io/ " >> /etc/profile && \

echo "export GOROOT=/usr/local/go " >> /etc/profile && \

echo "export GOPATH=/gopath " >> /etc/profile && \

echo "export PATH=$PATH:$GOROOT/bin " >> /etc/profile && \

source /etc/profile

ENV GO111MODULE on

ENV GOROOT /usr/local/go

ENV GOPATH /gopath

ENV PATH $GOROOT/bin:$GOPATH/bin:$PATH

ENV GOPROXY=https://goproxy.io/

WORKDIR /download

COPY servicecomb-service-center-1.3.0.tar.gz /download/

RUN \

cd /download && \

tar -C /opt -zxvf servicecomb-service-center-1.3.0.tar.gz && \

cd /opt/servicecomb-service-center-1.3.0 && \

go mod download && \

go mod vendor && \

go build -o service-center && \

mkdir /opt/service-center/conf && \

cp /opt/servicecomb-service-center-1.3.0/etc/conf/app.conf /opt/service-center/conf/app.conf && \

mv /opt/servicecomb-service-center-1.3.0/service-center /opt/service-center

RUN \

chmod -R 777 /opt/service-center && \

cd /opt/service-center

COPY start.sh /opt/service-center/

CMD ["/bin/sh","/opt/service-center/start.sh",">", "/opt/service-center/1.log","&"]

start.js:

#!/bin/sh

umask 027

cd /opt/service-center

sed -i "s|^registry\_plugin.\*=.\*$|registry\_plugin = etcd|g" conf/app.conf

sed -i "s|^manager\_cluster.\*=.\*$|manager\_cluster = ${BACKEND\_ADDRESS}|g" conf/app.conf

sed -i "s/^httpaddr.\*=.\*$/httpaddr = ${IP\_ADDRESS}/g" conf/app.conf

./service-center

cmd命令行建造生成镜像：

*docker build coollyj/servicecenter:0.001 .*

1. service-center配置

在docker-compose.yaml中添加service-center分布式配置：

######################## service-center 集群 start#################################

service-center:

depends\_on:

- etcd-1

image: 'coollyj/servicecenter:1.3.0.1'

hostname: servicecenter103

ports:

- "30100:30100"

environment:

BACKEND\_ADDRESS: etcd-1:2379,etcd-2:2379,etcd-3:2379

IP\_ADDRESS: 172.20.0.103

networks:

default:

ipv4\_address: 172.20.0.103

service-center1:

depends\_on:

- etcd-2

- etcd-3

image: 'coollyj/servicecenter:1.3.0.1'

hostname: servicecenter104

ports:

- "30101:30100"

networks:

default:

ipv4\_address: 172.20.0.104

environment:

BACKEND\_ADDRESS: etcd-1:2379,etcd-2:2379,etcd-3:2379

IP\_ADDRESS: 172.20.0.104

scfrontend1:

depends\_on:

- service-center1

image: 'servicecomb/scfrontend:latest'

links:

- service-center1

ports:

- "30104:30103"

environment:

SC\_ADDRESS: http://service-center1:30100

networks:

default:

ipv4\_address: 172.20.0.201

scfrontend:

depends\_on:

- service-center

image: 'servicecomb/scfrontend:latest'

ports:

- "30103:30103"

environment:

SC\_ADDRESS: http://service-center:30100

networks:

default:

ipv4\_address: 172.20.0.200

######################## service-center 集群 end#################################

1. 测试

启动 docker-compse up

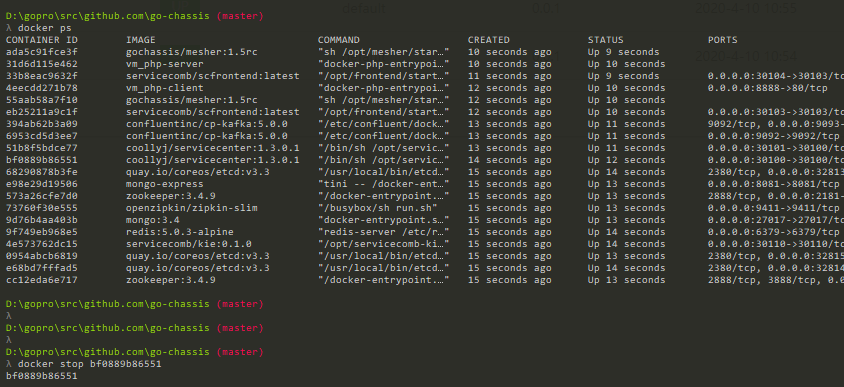
打开 <http://127.0.0.1:30103/#!/sc/services/>

<http://127.0.0.1:30104/#!/sc/services/>

进入\github.com\go-chassis\ygx\sidecar\example\rest\server 运行run.bat win

进入\github.com\go-chassis\ygx\sidecar\example\rest\client 运行run.bat win 运行成功。

然后关闭 service-center1，

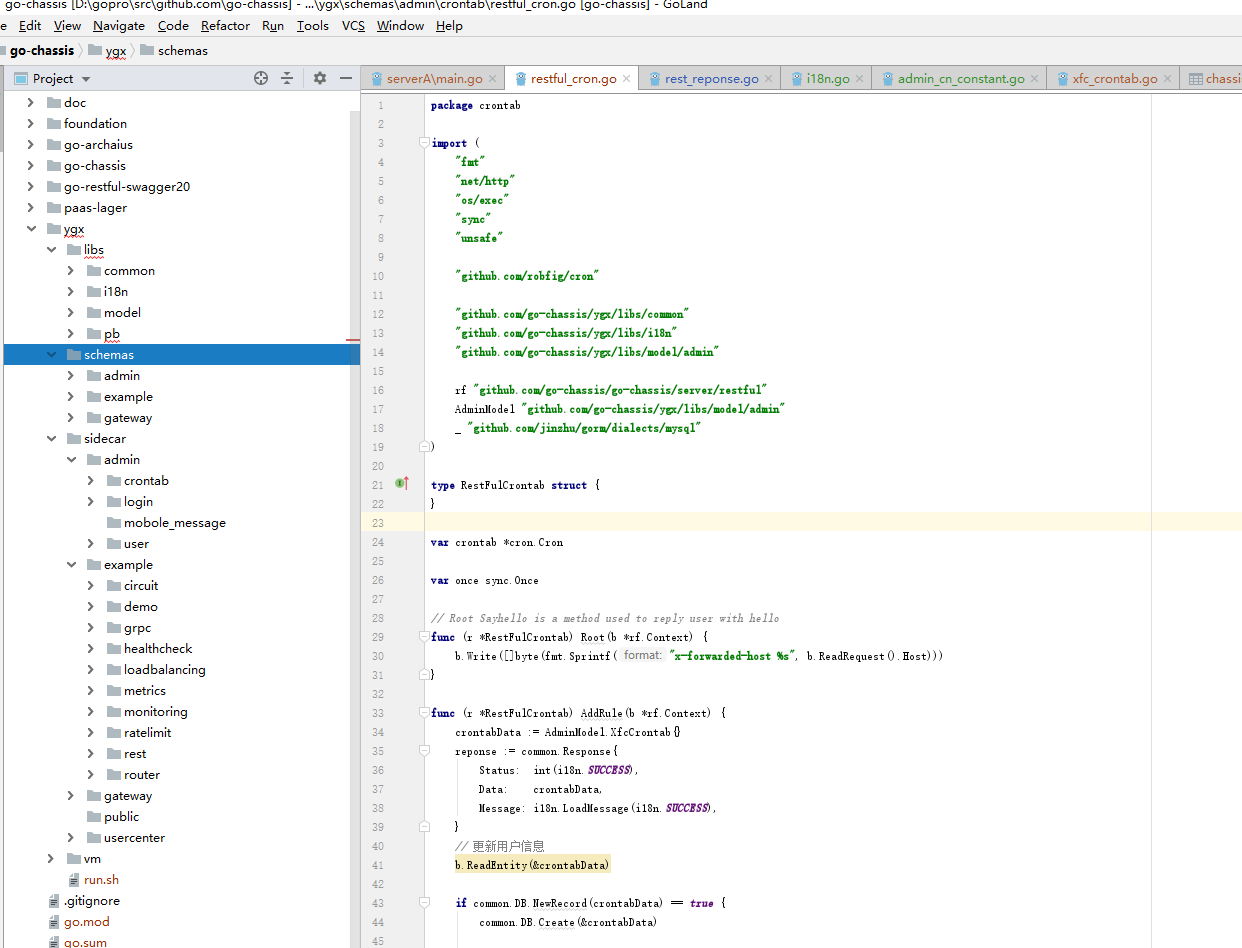


再在\github.com\go-chassis\ygx\sidecar\example\rest\client 运行run.bat win 仍然成功。

1. 转换器

# 目录结构

目录结构如下：



约定：

项目文件夹： ygx

公用库文件夹：ygx/libs

国际化消息定义：ygx/libs/i18n

数据库结构定义：ygx/libs/model

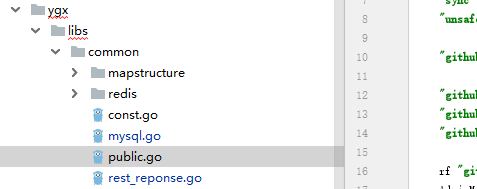
Probuff3 消息文件：ygx/libs/pb

其他公用库文件，公共函数：mysql.go 数据库连接

redis/\* redis操作

rest\_reponse.go 公共的返回结构 json类型

const.go 常量定义



微服务实现文件夹：ygx/schemas

后端微服务实现：ygx/chemas/admin

网关微服务实现：ygx/chemas/gateway

示例代码微服务实现：ygx/chemas/example

微服务sidecar文件夹：ygx/sidecar

后端sidecar： ygx/sidecar/admin

Example sidecar： ygx/sidecar/example

网关sidecar： ygx/sidecar/gateway

用户中心sidecar：ygx/sidecar/usercenter

其子目录命名: 功能/协议-server(或client)/

# 开发流程

1. Mysql 调用

数据库链接代码：github.com\go-chassis\ygx\libs\common\mysql.go

调用示例：

**……………………….**

**import** (

**"github.com/go-chassis/ygx/libs/common"**AdminModel **"github.com/go-chassis/ygx/libs/model/admin"**\_ **"github.com/jinzhu/gorm/dialects/mysql"**

**）**

**// 直接使用**common.DB 即调用

**//** 调用案例 <http://gorm.book.jasperxu.com/crud.html#u>

common.DB.Create(&crontabData)

Todo : 集群链接

1. Redis 调用

Redis链接代码 : github.com\go-chassis\ygx\libs\common\redis\redis.go

调用示例：

**……………………….**

**import** (

………

**"github.com/go-chassis/ygx/libs/common/redis"**

**）**

**// 直接使用**reids调用

hkey := **"XfcUserToken"**XfcUserTokenstr = redis.HashGet(hkey, userName)

Todo: 集群链接

1. 提示消息国际化

消息提示代码 : github.com\go-chassis\ygx\libs\i18n

调用示例：

**……………………….**

**import** (

………

**"github.com/go-chassis/ygx/libs/i18n"**

**）**

**Func test(){**

**// 加载消息号和消息提示**

reponse := common.Response{  
 Status: int(i18n.***SUCCESS***), //定义的消息号  
 Data: crontabData,  
 Message: i18n.LoadMessage(i18n.***SUCCESS***), //加载定义好的消息号对应的消息  
}

**}**

**func** init() {  
 *//国际化语言设置 表示 admin 后台，中文信息提示* i18n.SetPlatLang(**"admin"**, **"cn"**)  
}

1. 添加定时任务
2. 运行

先进入文件夹 ：github.com\go-chassis\ygx\sidecar\admin\crontab\rest-server

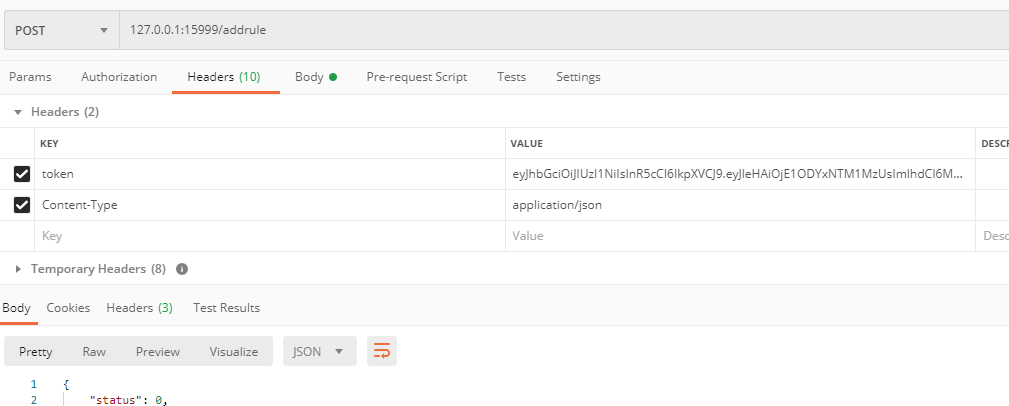
撰写 run.bat

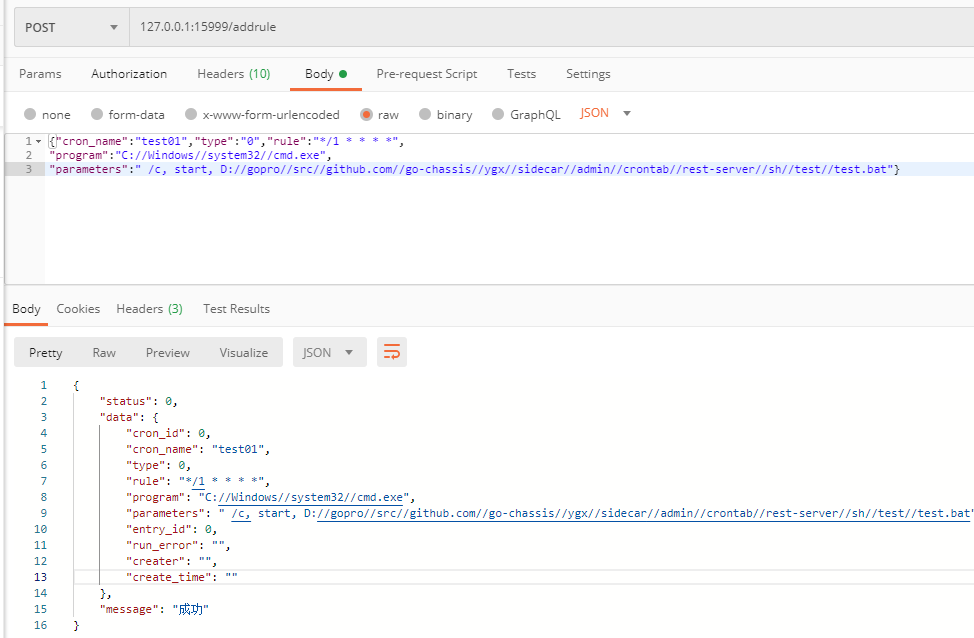
然后双击运行即可。

@**echo off  
echo %1  
  
set os**=%1  
**set ChassisConfDir**=..\\conf  
**echo %ChassisConfDir%  
set CHASSIS\_HOME**=.  
**echo %CHASSIS\_HOME%**@**echo off  
if** /i **"%os%"**==**"linux"** (  
 **SET CGO\_ENABLED**=0  
 **SET GOOS**=linux  
 **SET GOARCH**=amd64  
 go build -o ygx\_admin\_crontab\_rest\_server main.go  
 ygx\_admin\_crontab\_rest\_server  
) **else** (  
 **SET CGO\_ENABLED**=0  
 **SET GOOS**=windows  
 **SET GOARCH**=amd64  
 go build -o ygx\_admin\_crontab\_rest\_server.exe main.go  
 ygx\_admin\_crontab\_rest\_server.exe  
)  
  
**echo success**

1. 调用

打开postman ,在下面输入如下参数：





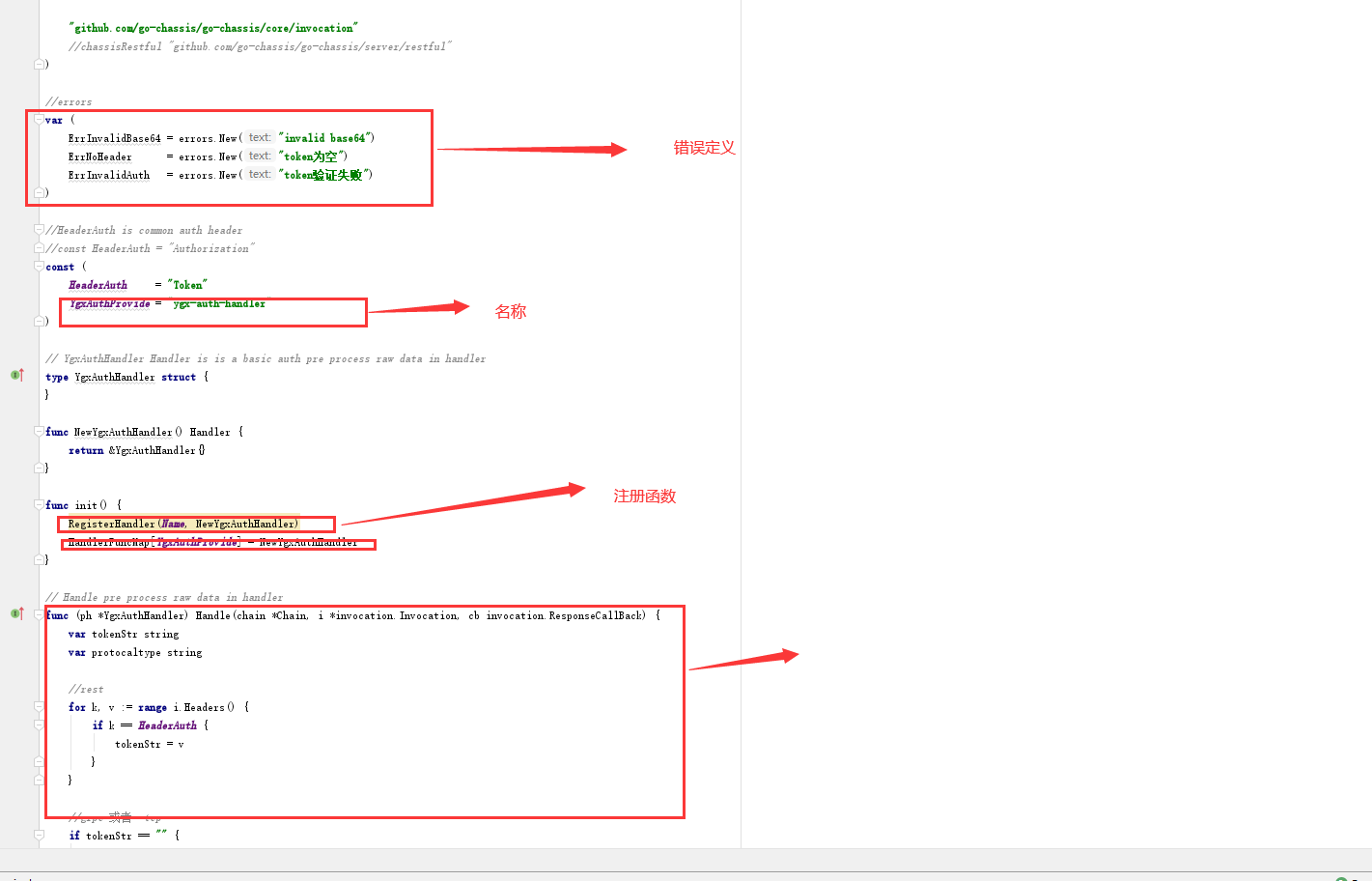
1. 授权认证配置

写成了中间件模式

token代码 : github.com\go-chassis\foundation\token

handle代码: github.com\go-chassis\ go-chassis\core\handle\ ygx\_auth\_handler.go

代码解析 ：



调用就是chassis.conf配置：

---  
**cse**:  
 **service**:  
 **registry**:  
 **address**: http://127.0.0.1:30100 *# If type is File then address will be the path of the file* **protocols**:  
 **rest**:  
 **listenAddress**: **"127.0.0.1:15999"  
 advertiseAddress**: **"127.0.0.1:15999"** *#internal\_ip* **handler**:  
 **chain**:  
 **Provider**:  
 **default**: ygx-auth-handler

1. 示例模块开发

创建sidecar

1. 创建package

现在github.com\go-chassis\ygx\sidecar\example\rest\ 创建package 根据要使用的通讯协议 如果是rest 就叫 rest-server，或者grpc 就叫 grpc-server, rest-server文件夹下面创建 conf，

1. 创建配置文件

conf文件夹下创建如下配置文件

microservice.yaml

*#微服务的私有属性192.168.11.57***service\_description**:  
 **name**: RESTServer

larger.yaml (日志配置)

---  
*#日志配置  
#Writers: "mike"***logger\_level**: **"DEBUG"  
logger\_file** : **"log\\chassis.log"  
log\_format\_text** : true  
**RollingPolicy**: **"size"  
log\_rotate\_date** : 1  
**log\_rotate\_size** : 4  
**log\_backup\_count** : 7

chassis.yaml

1. 微服务业务代码

在github.com\go-chassis\ygx\schemas\example\ 文件夹下创建

restful\_hello.go 如下

---  
**cse**:  
 **service**:  
 **registry**:  
 **disabled**: false  
 **refreshInterval**: 10s  
 **watch**: true  
 **address**: http://127.0.0.1:30100 *# If type is File then address will be the path of the file 127.0.0.1:5001* **protocols**:  
 **rest**:  
 **listenAddress**: **"127.0.0.1:15011"  
 advertiseAddress**: **"127.0.0.1:15011"** *# internal\_ip* **handler**:  
 **chain**:  
 **Provider**:  
 **default**: tracing-provider,bizkeeper-provider  
*# default: basicAuth, tracing-provider,bizkeeper-provider*

**package** schemas  
**import** (  
 **"errors"  
 "log"  
 "net/http"  
  
 "fmt"  
 "math/rand"** rf **"github.com/go-chassis/go-chassis/server/restful"**)  
**var** num = rand.Intn(100)  
  
*//RestFulHello is a struct used for implementation of restfull hello program***type** RestFulHello **struct** {  
}  
*//Sayhello is a method used to reply user with hello***func** (r \*RestFulHello) Sayhello(b \*rf.Context) {  
 id := b.ReadPathParameter(**"userid"**)  
 b.Write([]byte(fmt.Sprintf(**"user %s from %d"**, id, num)))  
}  
*// SayJSON is a method used to reply user hello in json format***func** (r \*RestFulHello) SayJSON(b \*rf.Context) {  
 reslut := **struct** {  
 Name string  
 }{}  
 err := b.ReadEntity(&reslut)  
 **if** err != nil {  
 b.WriteHeaderAndJSON(http.***StatusInternalServerError***, reslut, **"application/json"**)  
 **return** }  
 reslut.Name = **"hello "** + reslut.Name  
 b.WriteJSON(reslut, **"application/json"**, **""**)  
 **return**}  
*//URLPatterns helps to respond for corresponding API calls***func** (r \*RestFulHello) URLPatterns() []rf.Route {  
 **return** []rf.Route{  
 {Method: http.***MethodGet***, Path: **"/sayhello/{userid}"**, ResourceFunc: r.Sayhello,  
 Returns: []\*rf.Returns{{Code: 200}}},  
 {Method: http.***MethodPost***, Path: **"/sayjson"**,  
 ResourceFunc: r.SayJSON,  
 Metadata: **map**[string]**interface**{}{  
 **"tags"**: []string{**"users"**, **"test"**},  
 },  
 Returns: []\*rf.Returns{{Code: 200}}},  
 }  
}  
  
*//Saymessage is used to reply user with his name***func** (r \*RestFulMessage) Saymessage(b \*rf.Context) {  
 id := b.ReadPathParameter(**"name"**)  
  
 b.Write([]byte(**"get name: "** + id))  
}  
  
*//Sayhi is a method used to reply request user with hello world text***func** (r \*RestFulMessage) Sayhi(b \*rf.Context) {  
 reslut := **struct** {  
 Name string  
 }{}  
 err := b.ReadEntity(&reslut)  
 **if** err != nil {  
 b.Write([]byte(err.Error() + **":hello world"**))  
 **return** }  
 b.Write([]byte(reslut.Name + **":hello world"**))  
 **return**}  
  
*//Sayerror is a method used to reply request user with error***func** (r \*RestFulMessage) Sayerror(b \*rf.Context) {  
 b.WriteError(http.***StatusInternalServerError***, errors.New(**"test hystric"**))  
 **return**}  
  
*//URLPatterns helps to respond for corresponding API calls***func** (r \*RestFulMessage) URLPatterns() []rf.Route {  
 **return** []rf.Route{  
 {Method: http.***MethodGet***, Path: **"/saymessage/{name}"**, ResourceFunc: r.Saymessage},  
 {Method: http.***MethodPost***, Path: **"/sayhimessage"**, ResourceFunc: r.Sayhi},  
 {Method: http.***MethodGet***, Path: **"/sayerror"**, ResourceFunc: r.Sayhi},  
 }  
}

1. sidecar代码

在github.com\go-chassis\ygx\sidecar\example\rest\rest-server\main.go

**package** main。  
**import** (  
 **"github.com/go-chassis/go-chassis  
 "github.com/go-chassis/go-chassis/core/lager"  
 "github.com/go-chassis/go-chassis/core/server"  
 "github.com/go-chassis/go-chassis/examples/schemas"**)  
  
*//if you use go run main.go instead of binary run, plz export***func** main() {chassis.RegisterSchema(**"rest"**, &schemas.RestFulHello{}, server.WithSchemaID(**"RestHelloService"**))  
 **if** err := chassis.Init(); err != nil {  
 lager.Logger.Error(**"Init failed."** + err.Error())  
 **return** }  
 chassis.Run()  
}

1. 运行

创建运行脚本 run.bat

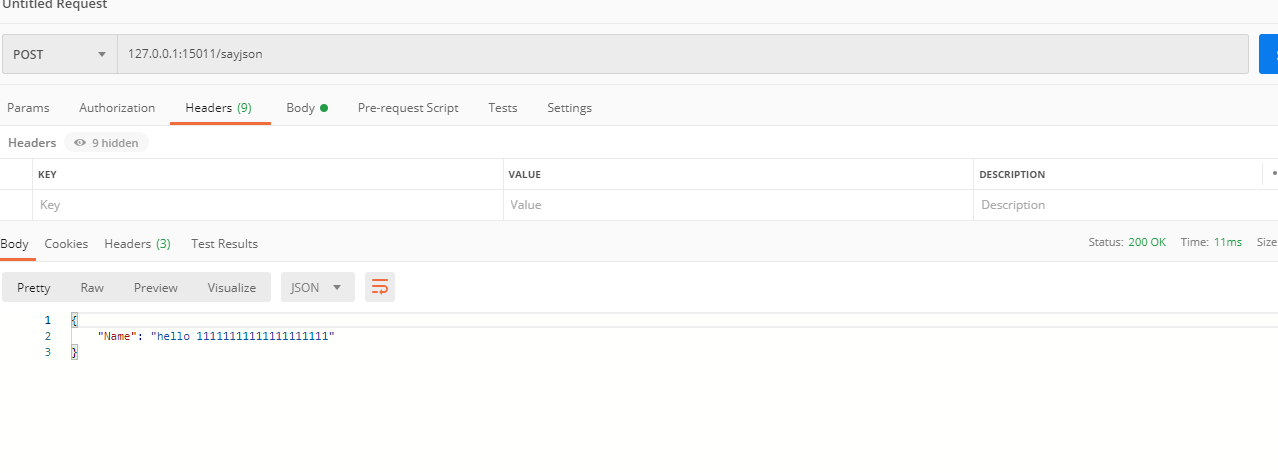
@**echo off  
echo %1  
set os**=%1  
**set ChassisConfDir**=..\\conf **set CHASSIS\_HOME**=.@**echo off  
if** /i **"%os%"**==**"linux"** (  
 **SET CGO\_ENABLED**=0  
 **SET GOOS**=linux  
 **SET GOARCH**=amd64  
 go build -o server main.go  
 server  
) **else** (  
 **SET CGO\_ENABLED**=0  
 **SET GOOS**=windows  
 **SET GOARCH**=amd64  
 go build -o server.exe main.go  
 server.exe  
)  
  
**echo success**

进入 github.com\go-chassis\ygx\vm 文件下面执行命令

docker-compose up 启动注册中心。

然后在当前文件夹下面运行 run.bat win

打开postman ,输入



1. 开发comsumer

开发和运行流程同上，具体代码参考

github.com\go-chassis\ygx\sidecar\example\rest\client

1. 网关程序运行
2. 先安装好 go和docker 开发环境，
3. 启动服务中心

进入 github.com\go-chassis\ygx\vm 文件下面

执行命令 docker-compose up

服务注册中心 端口为 30100 ，前端运维面板：[http://127.0.0.1:30103/#!/sc/dashboard](http://127.0.0.1:30103/" \l "!/sc/dashboard)

1. 启动网关微服务

运行微服务之前，需要产生签名的csr 和key 用于https

1. 产生带密码的私钥

openssl genrsa -des3 -out server.key 1024 提示输入密码 本

1. 产生csr 和 crt 的公钥

openssl req -new -key server.key -out server.csr

openssl x509 -req -days 365 -in server.csr -signkey server.key -out server.crt

1. 配置 tls.yaml

应用的链

**ssl**:  
 **rest.Provider.cipherPlugin**: default  
 **rest.Provider.cipherSuits**: TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256, TLS\_ECDHE\_RSA\_WITH\_AES\_256\_GCM\_SHA384  
 **rest.Provider.protocol**: TLSv1.2  
 **rest.Provider.keyFile**: server.key  
 **rest.Provider.certFile**: server.crt  
 **rest.Provider.certPwdFile**: pwd

运行之前先设置 环境变量 ChassisConfDIr ，其值是当前微服务的配置文件的目录地址。

编译 : go build -o ygx\_rest\_gateway.exe main.go

直接点击运行 ： ygx\_rest\_gateway.exe

对应swagger: 127.0.0.1:端口/apidocs.json

1. 测试 (代码未完成)
2. 单元测试
3. 压力测试
4. 队列kafka (代码未完成)

# 参考文档

Mysql grom <http://gorm.book.jasperxu.com/crud.html#u>

cron : <https://godoc.org/github.com/robfig/cron>