**微服务(spring cloud)及docker技术分享**

**date ：2018.07.16**

**author：christian**

目录

[2 前言 2](#_Toc15981055)

[2.1 读本文需要的一些基础知识； 2](#_Toc15981056)

[2.2 关于springcloud升级 2](#_Toc15981057)

[3 文献资料 3](#_Toc15981058)

[4 直奔主题 3](#_Toc15981059)

[5 软件版本信息 3](#_Toc15981060)

[6 Springcloud 4](#_Toc15981061)

[7 Springcloud架构介绍 4](#_Toc15981062)

[7.1 简介 4](#_Toc15981063)

[7.2 整体架构图 4](#_Toc15981064)

[8 Springcloud核心组件介绍 5](#_Toc15981065)

[8.1 springboot升级到2.1.6 5](#_Toc15981066)

[8.2 服务注册发现 6](#_Toc15981067)

[8.2.1 介绍 6](#_Toc15981068)

[8.2.2 关系图 8](#_Toc15981069)

[8.2.3 使用 8](#_Toc15981070)

[8.3 统一配置中心 10](#_Toc15981071)

[8.3.1 介绍 10](#_Toc15981072)

[8.3.2 架构图 11](#_Toc15981073)

[8.3.3 使用 11](#_Toc15981074)

[8.3.4 其它项目使用springcloudconfig 12](#_Toc15981075)

[8.4 服务消费者生产者 12](#_Toc15981076)

[8.4.1 介绍 12](#_Toc15981077)

[8.4.2 使用 13](#_Toc15981078)

[8.5 断路器（hystrix） 14](#_Toc15981079)

[8.5.1 介绍 14](#_Toc15981080)

[8.5.2 使用 15](#_Toc15981081)

[8.6 网关 springcloudgateway（推荐） 16](#_Toc15981082)

[8.6.1 springcloudgateway的filter 16](#_Toc15981083)

[8.6.2 网关应该具备的功能 17](#_Toc15981084)

[8.6.3 搭建程序框架 18](#_Toc15981085)

[8.7 api网关zuul（不建议使用） 20](#_Toc15981086)

[8.7.1 介绍 20](#_Toc15981087)

[8.7.2 使用 20](#_Toc15981088)

[8.8 分布式锁 21](#_Toc15981089)

[8.9 其它开发过程注意事项 21](#_Toc15981090)

[8.9.1 微服务下数据库事务 21](#_Toc15981091)

[8.9.2 服务的幂等性（接口幂等性） 21](#_Toc15981092)

[8.9.3 提供的服务需要具备独立性及完整性 22](#_Toc15981093)

[8.9.4 生产者消费者接口bean 22](#_Toc15981094)

[8.10 断路器监控 22](#_Toc15981095)

[8.11 服务链路追踪 22](#_Toc15981096)

[8.12 消息总线 22](#_Toc15981097)

[9 部署实施 22](#_Toc15981098)

[9.1 操作系统 22](#_Toc15981099)

[9.2 安装docker 22](#_Toc15981100)

[9.2.1 配置docker加速器 24](#_Toc15981101)

[9.3 制作docker镜像 25](#_Toc15981102)

[9.3.1 创建dockerfile 25](#_Toc15981103)

[9.3.2 创建镜像 25](#_Toc15981104)

[9.3.3 查看及删除 25](#_Toc15981105)

[9.4 新建docker容器 25](#_Toc15981106)

[9.4.1 配置docker网络 25](#_Toc15981107)

[9.4.2 创建容器 26](#_Toc15981108)

[9.5 常用docker命令 26](#_Toc15981109)

[9.6 制作jar包 27](#_Toc15981110)

[9.6.1 安装maven并配置maven环境变量 27](#_Toc15981111)

[9.6.2 编译并打包 27](#_Toc15981112)

[10 Springclouddemo演示 27](#_Toc15981113)

[11 其它 28](#_Toc15981114)

[11.1 gitlab安装及使用 28](#_Toc15981115)

[11.1.1 安装 28](#_Toc15981116)

[11.1.2 使用 28](#_Toc15981117)

# 前言

## 读本文需要的一些基础知识；

了解spring boot

了解maven

## 关于springcloud升级

升级过程主要升级pom和修改引用类。升级pom后maven update project时需要选中eclipse {force update 。。。。}

# 文献资料

Springcloud中文官网

<https://springcloud.cc/spring-cloud-dalston.html#_spring_cloud_config>

https://springcloud.cc/spring-cloud-dalston.html

# 直奔主题

微服务框架主要有 dubbo和springcloud。不选择dubbo是不想重复造轮子。我们主要阐述springcloud的实际应用

|  |  |  |
| --- | --- | --- |
|  | Dubbo | Springcloud |
| 服务注册中心 | Zookeeper | Spring cloud Netflix eureka |
| 服务调用方式 | Rpc | Rest api |
| 服务网关 | 无 | Spring cloud Netflix zuul |
| 断路器 | 不完善 | Spring cloud Netflix hystrix |
| 分布式配置 | 无 | Spring cloud config |
| 服务跟踪 | 无 | Spring cloud sleuth --zipkin |
| 消息总线 | 无 | Spring cloud bus |
| 数据流 | 无 | Spring cloud stream |
| 批量任务 | 无 | Spring cloud task |
| …… | …… | …… |
|  |  |  |

# 软件版本信息

|  |  |  |  |
| --- | --- | --- | --- |
| 软件名称 | 版本 | 社区活跃度 | 备注 |
| Springboot | 1.5.9 部分有1.5.2 |  |  |
| Sprng cloud | **Dalston.SR4** |  |  |
| Maven | 3.5.3 |  |  |
| Docker | 17.06.2-ce |  |  |
| Mysql | 5.7 |  |  |
| Java | 1.8 |  |  |
| Centos操作系统 | 3.10.0-x86\_64 |  |  |
|  |  |  |  |
|  |  |  |  |

# Springcloud

项目演示版本基于 springboot 1.5.9 和1.5.2 。

基于 springcloud dalston.SR4

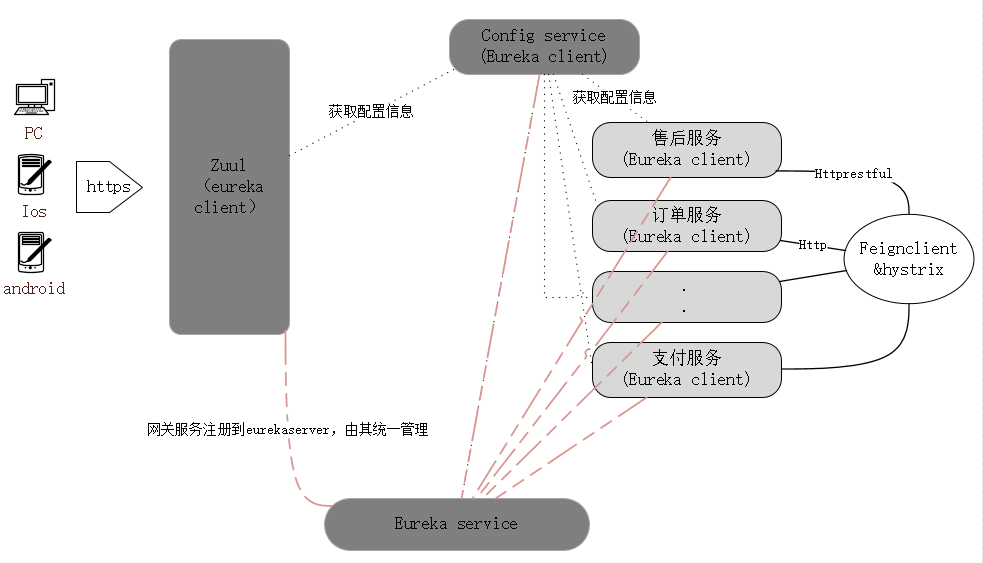
注意Springboot和springcloud之间的版本对应关系很重要。

# Springcloud架构介绍

## 简介

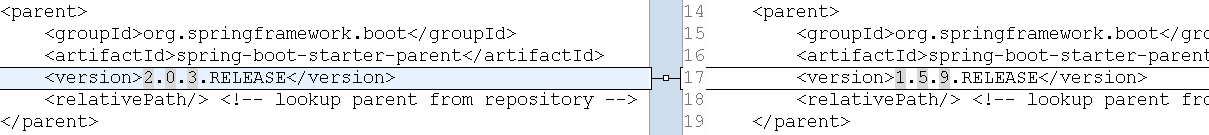
Springcloud 是一系列框架的集合，它基于cpringboo的开发便利性实现了大部分基础设施的开发。如 服务发现注册等服务治理，配置中心，网关，断路器等等。

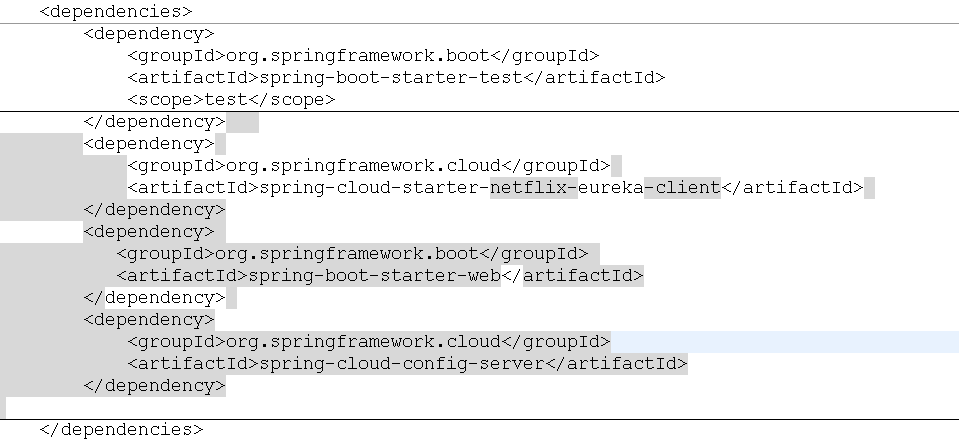
## 整体架构图

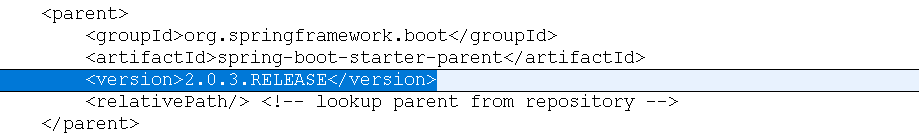


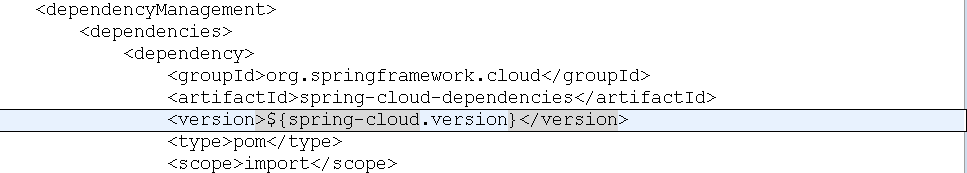
# Springcloud核心组件介绍

## springboot升级到2.1.6









## 服务注册发现

### 介绍

eureka是一个高可用的组件，它没有后端缓存，每一个实例注册之后需要向注册中心发送心跳（因此可以在内存中完成），在默认情况下erureka server也是一个eureka client ,须要指定一个 eureka server。

Eureka分为Eureka Server和Eureka client，采用的客户端发现模式

#### Eureka Server

Eureka Server是一个服务注册发现中心，为服务实例注册管理和查询可用实例提供了REST API。

Eureka Server自保模式：eurekaserver在一定时间内没有收到某个微服务实例心跳后会从注册表注销此实例，当由于网络等非服务本身故障发生时，eurekaserver是不应该注销这个实例的，所以eurekaserver在短时间内丢失过多客户端就进入了自保模式，一旦进入自保模式，eurekaserver就会保护注册表中的信息。

#### Eureka Server彼此之间

采用的是对等通信(P2P),无中心化的架构，无master/slave区分.每个节点需要添加一个或多个有效的serviceUrl指向另一个节点

#### Eureka Client

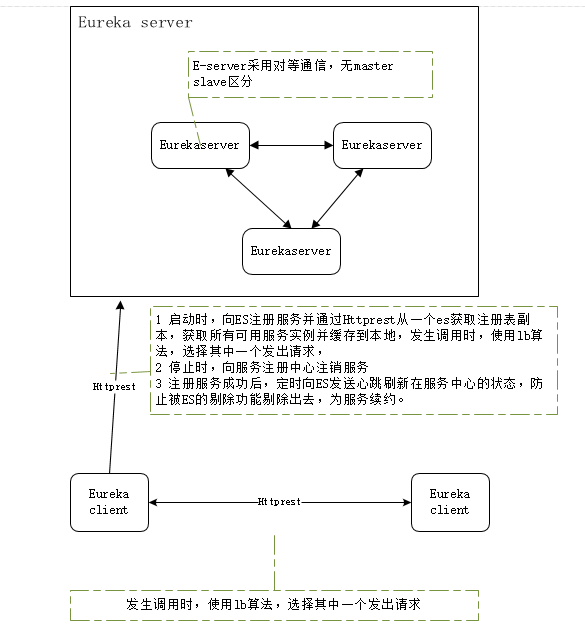
Eureka Client是一个普通的springboot服务，

Eureka Client服务启动后，Eureka Client向服务注册中心注册服务，同时会拉取注册中心注册表副本，并缓存到本地。服务调用时，客户端使用负载均衡（feign）算法从多个后端服务实例中选择出一个，然后发出请求。这种客户端发现模式的优点是服务调用、负载均衡不需要和Eureka Server通信，直接使用本地注册表副本，因此Eureka Server不可用时是不会影响正常的服务调用，性能也不会因为网络延迟和eurekaserver的故障或延迟受到影响。但其缺点也很明显，当某个服务不可用时，各个Eureka Client不能及时的知道，需要1~3个心跳周期才能感知，但是，由于基于Netflix的服务调用端都会使用Hystrix来容错和降级，当服务调用不可用时Hystrix也能及时感知到，通过熔断机制来降级服务调用，因此弥补了基于客户端服务发现的时效性的缺点。

Eureka Client服务停止的时候，Eureka Client向服务注册中心注销服务；

Eureka Client服务注册后，Eureka Client会定时的发送心跳来刷新服务的最新状态，防止eurekaserver的剔除功能将其剔除出去，继而维持服务的续约。

### 关系图

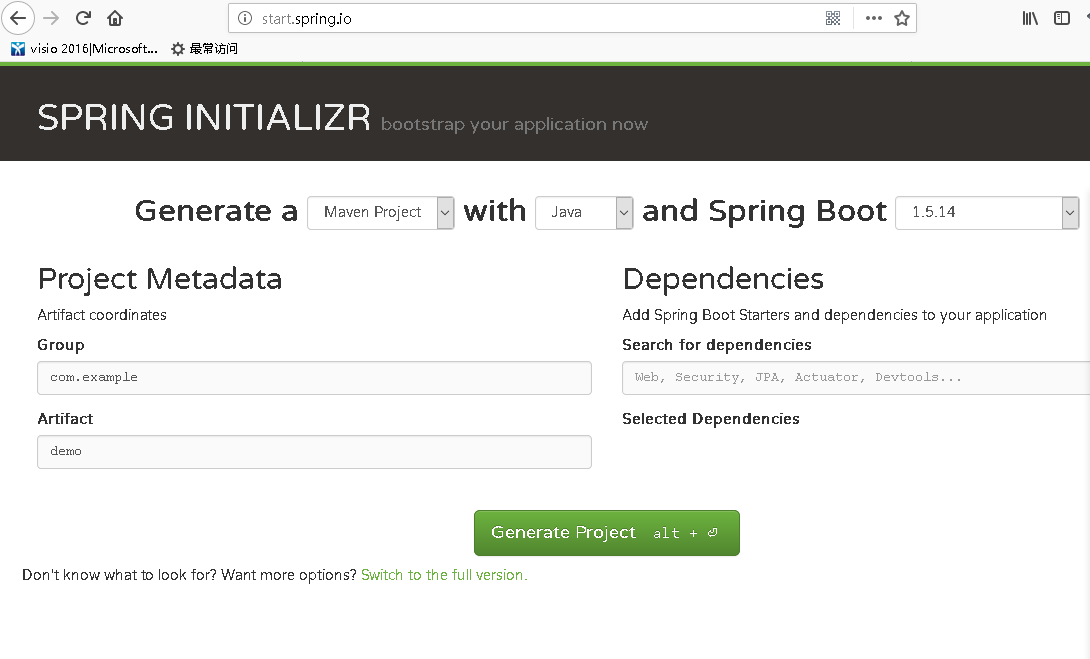


### 使用

#### 建立项目

首先进入 <http://start.spring.io/> 生成项目

选择 语言，版本，然后下载到本地，并导入到各自的开发工具中（因为生成的是maven项目，所以不局限用什么IDE）参考下图：



#### 配置pom.xml 文件

主要加入如下jar包：

<!--eureka server -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka-server</artifactId>

</dependency>

#### 启动类application类上追加注解

这样在服务启动的时候，服务注册发现中心就会随着启动起来。

@EnableEurekaServer

@SpringBootApplication

public class EurekaserverApplication {

public static void main(String[] args) {

SpringApplication.run(EurekaserverApplication.class, args);

}

//

// @Bean

// public Filter webRequestLoggingFilter() {

// return new CommonsRequestLoggingFilter();

// }

}

#### 配置文件

eureka server的配置文件appication.yml ，看项目源码

enableSelfPreservation: false

evictionIntervalTimerInMs: 4000

所谓自我保护模式是指，出现网络等故障、eureka在短时间内丢失过

多客户端时，会进入自我保护模式，即一个服务长时间没有发送心跳，eureka也不会将其删除。默认为true

registerWithEureka 是否将自己注册到eureka

#### 启动工程

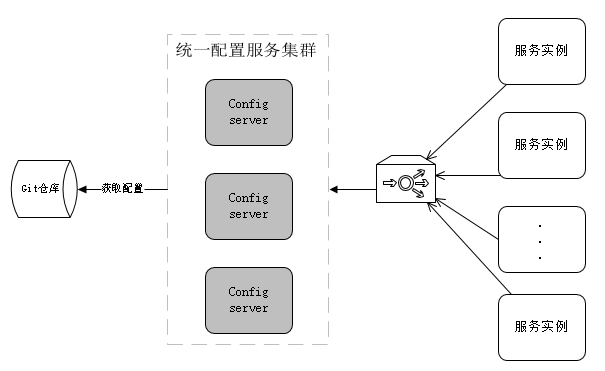
<http://localhost:8761>

## 统一配置中心

### 介绍

在分布式系统中，由于服务数量巨多，为了方便配置文件的统一管理，实时更新，需要分布式配置中心组件，springcloud分布式配置中心组件是spring cloud config，它支持将配置数据放在内存中，也可以放在git或svn中，接下来的分享和实例都是基于配置文件保存在git中

### 架构图



### 使用

#### 建立项目

新建一个springboot项目

#### 配置pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

#### 配置启动类

追加注解 @EnableConfigServer

#### 配置文件

#git仓库地址，如果是gitlab，必须以.git结尾，github不必。

spring.cloud.config.server.git.uri=https://github.com/likemytea/SpringcloudConfig.git

#配置仓库路径

spring.cloud.config.server.git.searchPaths=configcenter

#配置仓库的分支

spring.cloud.config.label=master

[spring.cloud.config.server.git.username=liuxing\_lhfe@yeah.net](mailto:spring.cloud.config.server.git.username=liuxing_lhfe@yeah.net)

spring.cloud.config.server.git.password=Liuxing1980

#### 注意

使用配置中心的eurekaclient要在pom.xml中配置

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

这样，所有的eurekaclient都可以将全部配置数据放到git上，而properties文件中只需配置是哪个配置文件即可，如下：

spring.cloud.config.label=master

spring.cloud.config.profile=dev

spring.cloud.config.uri=http://172.27.158.3:8888/

### 其它项目使用springcloudconfig

spring.application.name=wsbq-api

spring.cloud.config.label=master

spring.cloud.config.profile=dev

spring.cloud.config.uri=http://172.16.176.51:8888/

## 服务消费者生产者

### 介绍

Eurekaclient按照调用角度可分为 服务消费者和服务生产者，本节主要阐述消费者

被调用者为生产者,调用者为消费者

在微服务架构中，通常每个业务会被拆分成一个个独立的服务，服务与服务的通讯是基于http restful的。Spring cloud有两种服务调用方式，一种是ribbon+restTemplate，另一种是feign，Feign默认集成了ribbon，所以我们接下来的实例都是使用的feign

#### Feign介绍

Feign是一个声明式的伪Http客户端，它使得写Http客户端变得更简单。使用Feign，只需要创建一个接口并注解。它具有可插拔的注解特性，可使用Feign 注解和JAX-RS注解。Feign支持可插拔的编码器和解码器。Feign默认集成了Ribbon，并和Eureka结合，默认实现了负载均衡的效果。

通常：feign是netflix提供的服务间基于http的rpc调用框架，在spring cloud得到广泛应用。默认情况下，一个feign client是在hystrix断路器中执行，并利用ribbon进行软负载选择远程target service，所以可以想象出一个feign client的层次架构是包裹的层次，hystrix控制整个rpc从调用到方法返回，而ribbon控制从选址到socket返回。

### 使用

建立工程同上，这里主要阐述针对feign的增量代码。

#### 配置pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

#### 配置 application.properties

#注册服务地址

eureka.client.service-url.defaultZone=http://172.27.158.4:8761/eureka/

#eurekaclient向eurekaserver发送心跳的频率的间隔时间，单位为秒。

eureka.instance.lease-renewal-interval-in-seconds=12

#eurekaserver在最近一次收到eurekaclient A 后，到当前的间隔时间，单位为秒。

eureka.instance.lease-expiration-duration-in-seconds=24

spring.application.name=eurekafeign98

server.port=8765

#开启熔断器

feign.hystrix.enabled=true

#### 程序代码

##### 注解

在程序启动类上追加注解。

@EnableDiscoveryClient 用于服务注册发现

@EnableFeignClients 启动feign clients

##### 程序-消费者代码

定义一个feign接口，通过@ FeignClient（“被调用服务的properties文件中的spring.application.name”），这个属性值最好全部大写，这样消费这个值的时候直接可以在eurekaserver页面看就可以了。来指定调用哪个服务。比如在代码中调用了service-user服务的“/getUser”接口，代码如下：

/\*\*

\* @author liuxing

\*

\*/

@FeignClient(value = "eurekaclient111", fallback = SchedualServiceHiHystric.class)

public interface ServiceUser {

@RequestMapping(value = "/getUser", method = RequestMethod.GET)

String getUserByName(@RequestParam(value = "name") String name);

}

##### 程序-生产者代码

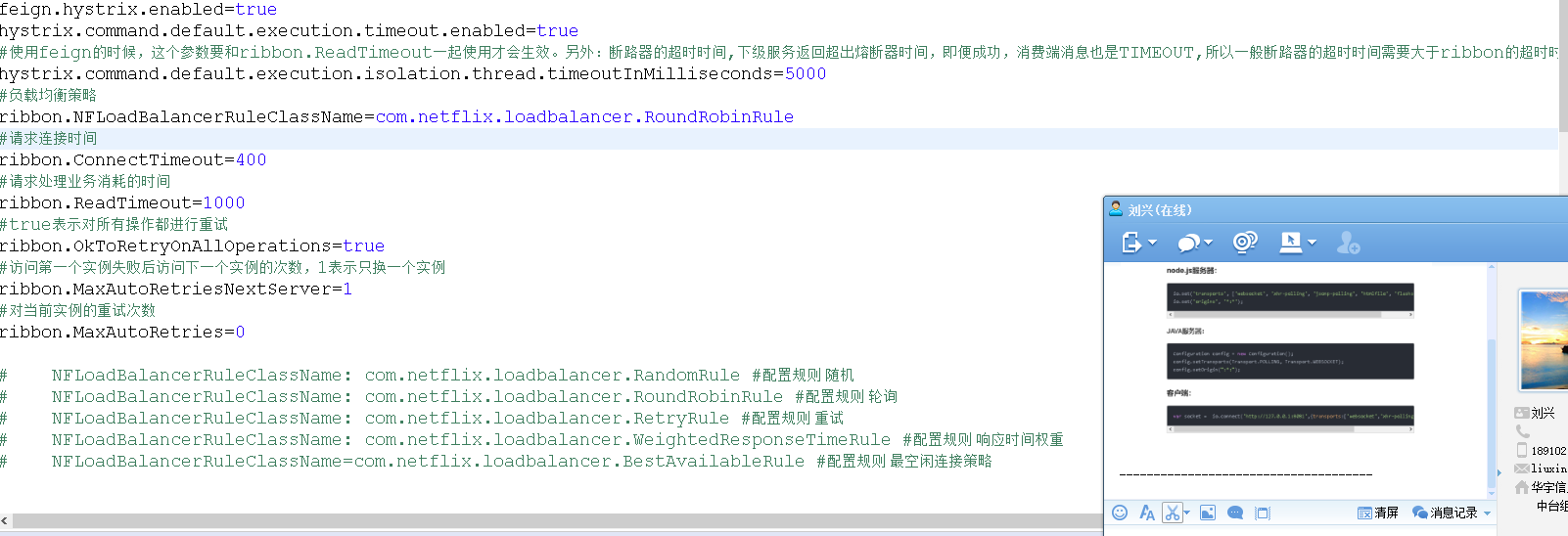
暴漏一个接口即可：在生产者client中的Web层的controller层，对外暴露一个”/ getUser”的接口。

##### feign使用ribbon负载均衡策略

使用如下的配置后，当访问到故障请求后，因为配置的MaxAutoRetries=0，所以不会再尝试访问当前故障实例，然后会根据MaxAutoRetriesNextServer=1访问另外一个实例，如果这个实例也失败，则返回失败信息，如果配置的是2，则再访问其它一个实例。

故而，根据下面的配置，应该有两个实例同时不可访问，才可能会出现访问失败的情况。所以我个人觉得，hystrix的用处并不是特别大，当然对于hystrix的服务降级业务除外。

只需要加如下配置即可。



## 断路器（hystrix）

### 介绍

为了解决繁杂多变的业务需求，在微服务架构中，我们根据业务来拆分成一个个的服务，服务与服务之间相互调用（RPC），在Spring Cloud可以用RestTemplate+Ribbon和Feign来调用。为了保证其高可用，单个服务通常会集群部署。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务的实例就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会交替传播，进而发生连锁故障，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。上述内容是本节要解决的问题。

### 使用

#### 配置文件

默认情况下feign是自带断路器的，所以只需在配置文件中追加如下代码：

feign.hystrix.enabled=true

#### 程序代码

新建一个普通的java类，本例中的类名SchedualServiceHiHystric，并实现我们之前做的demo中写的那个 ServiceUser这个接口。

在ServiceUser这个接口中，注解后面追加如下内容fallback = SchedualServiceHiHystric.class

然后SchedualServiceHiHystric需要实现SchedualServiceHi 接口，并注入到Ioc容器中

@Component

public class SchedualServiceHiHystric implements Servicehi {

@Override

public String sayHiFromClient(String name) {

return "服务配置错误或网络异常： " + name;

}

}

#### 测试

关闭被调用的服务，会提示配置文件中预先设置的信息提示

调用

<http://39.107.82.1:8766/user/sayHi01?id=111&name=adnihao&currentpage=1&pagesize=1>

## 网关 springcloudgateway（推荐）

参考资料:https://www.cnblogs.com/davidwang456/p/10411451.html

API 网关出现的原因是微服务架构的出现，不同的微服务一般会有不同的网络地址，而外部客户端可能需要调用多个服务的接口才能完成一个业务需求，如果让客户端直接与各个微服务通信，会有以下的问题：

1. 客户端会多次请求不同的微服务，增加了客户端的复杂性。
2. 存在跨域请求，在一定场景下处理相对复杂。
3. 认证复杂，每个服务都需要独立认证。
4. 难以重构，随着项目的迭代，可能需要重新划分微服务。例如，可能将多个服务合并成一个或者将一个服务拆分成多个。如果客户端直接与微服务通信，那么重构将会很难实施。
5. 某些微服务可能使用了防火墙 / 浏览器不友好的协议，直接访问会有一定的困难。

使用 API 网关后的优点如下：

1. 易于监控。可以在网关收集监控数据并将其推送到外部系统进行分析。
2. 易于认证。可以在网关上进行认证，然后再将请求转发到后端的微服务，而无须在每个微服务中进行认证。
3. 减少了客户端与各个微服务之间的交互次数。

我们可以使用springcloudgateway通过filter工作流程点，在“pre”类型的过滤器做参数校验、权限校验、流量监控、日志输出、协议转换等，在“post”类型的过滤器中可以做响应内容、响应头的修改，日志的输出，流量监控等。首先需要弄清一点为什么需要网关这一层，这就不得不说下filter的作用了。

### springcloudgateway的filter



#### 编写自己的filter：TODO;

<https://blog.csdn.net/forezp/article/details/85057268>

#### loadbalancer

本文档主要介绍负载均衡客户端过滤器lb。

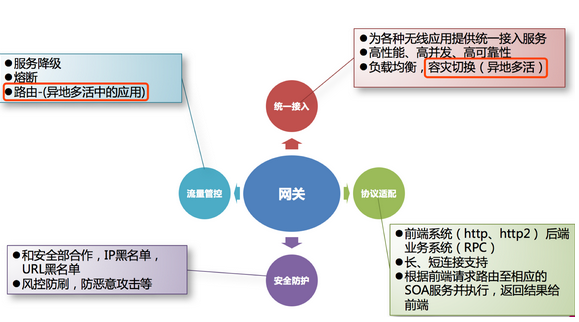
ribbon主要实现负载均衡算法，用的是自定义的一致性hash负载均衡算法。

hystrix主要实现服务可用性的实现。当然，是结合的eureka服务治理共同实现的。

当某挂载在后端的服务不可用时，hystrix先断路，不至于引发整个平台服务不可用及出现雪崩，这时eureka还未必发现此服务是不可用的（因为eureka是定时心跳，需要下次心跳才能发现最新的服务状态）。

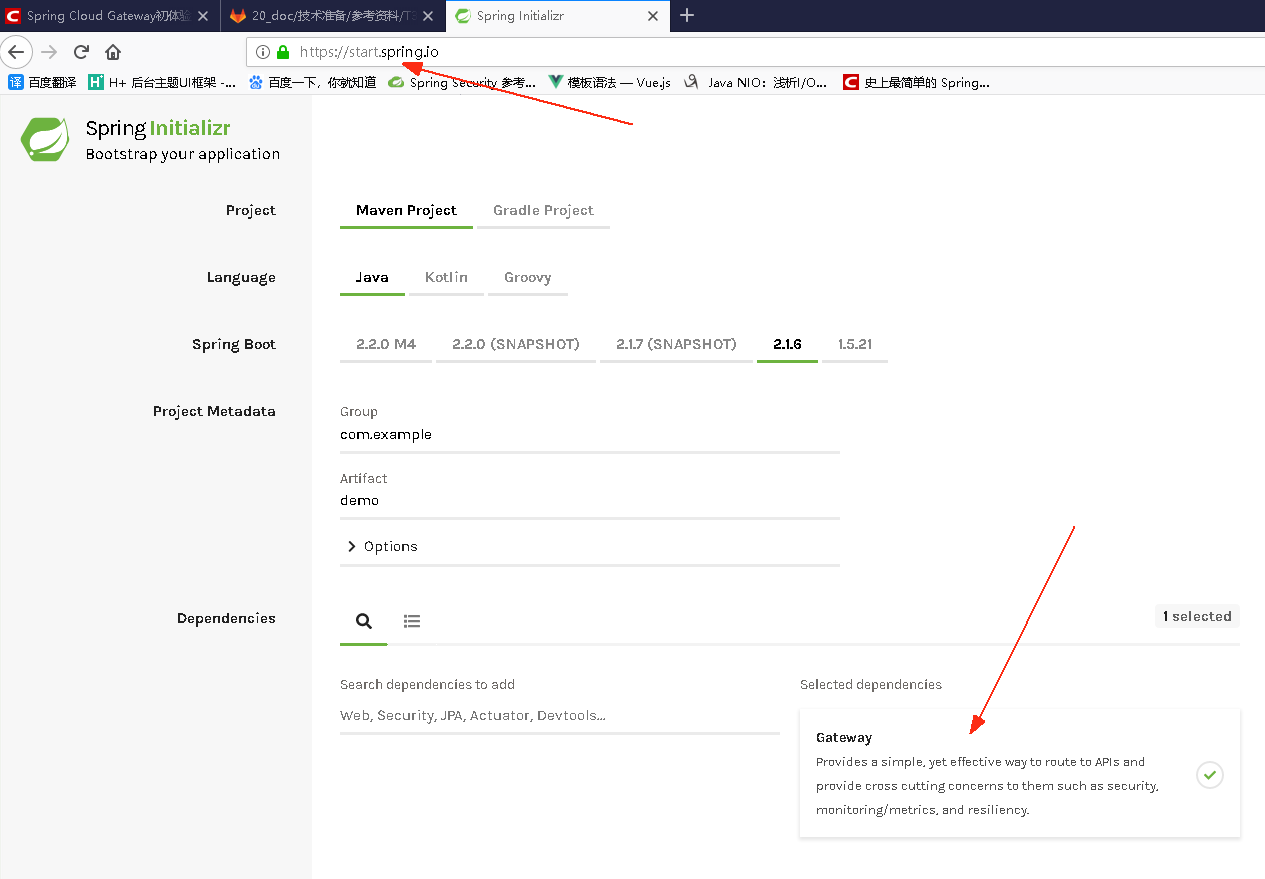
hystrix有定期检测机制，检测后面挂载的服务是否处于可用状态。

### 网关应该具备的功能

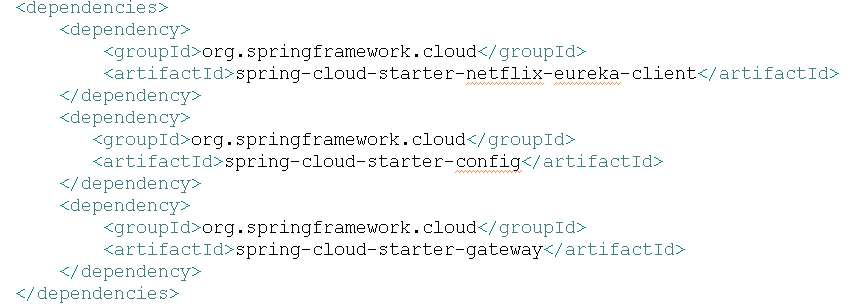


### 搭建程序框架

#### 建立工程



#### pom



#### properties



#### 测试

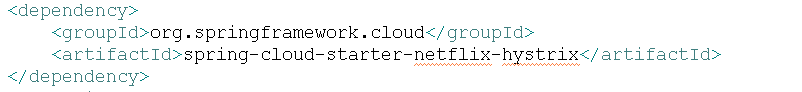
<http://172.16.14.190:8772/wsbq-api/user/demo?id=111&name=xxx&currentpage=1&pagesize=1>

#### springcloudgateway集成 hystrix全局断路器

##### properties

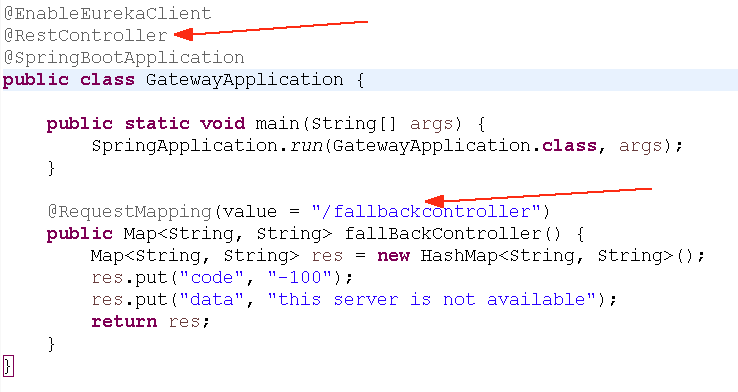
参考实例

##### pom



##### 程序

注意，要加restcontroller注解



#### 自定义gateway的负载均衡策略

##### 重写LoadBalancerClientFilter

##### 重写AbstractLoadBalancerRule

##### 依赖注入上面两个bean即可。

## api网关zuul（不建议使用）

### 介绍

Springcloud实现了网关，组件名称zuul

#### 网关作用：

为了保护内部系统的安全性，内网和外网是隔离的，内部系统服务只接收网关转发过来的请求，另外，我们在日常开发中，除了要实现业务逻辑功能外，还面临非业务逻辑功能，比如安全类的，鉴权，限流，监控等。这些任务的开发，本不该和业务逻辑代码混淆在一起。

### 使用

#### 配置pom.xml

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

#### 配置启动类注解

@EnableZuulProxy

#### 配置properties文件

zuul.routes.test-myservice.path=/api-test1/\*\*

zuul.routes.test-myservice.serviceId=eurekafeign98

zuul.routes.test-myservice2.path=/api-test2/\*\*

zuul.routes.test-myservice2.serviceId=userservice

#### 演示

详见demo！

## 分布式锁

实现分布式锁的方式有很多种，比如 最简单的就是使用数据库的 select …for update

使用zookeeper的临实节点特性来实现分布式锁

咱们这里使用的时redis的setNX 和setEx特性

代码在common3的工程里

## 其它开发过程注意事项

### 微服务下数据库事务

数据库事务的代码和调用服务的代码分开处理，避免数据库响应能力被网络或业务拖累的情况

### 服务的幂等性（接口幂等性）

微服务环境下，绝大部分服务都被对外提供服务，这时要注意涉及到（insert和update）操作的接口要保证幂等性：即同一个请求，调用一次和若干次得到的结果是一样的。

解决的方式具体业务具体分析：

例如：支付接口可以用 订单号字段+状态字段来保证业务的幂等性

如果不涉及到数据库的或者数据库没有类似字段的，可以使用redis保证其某个时间范围内的唯一性。

### 提供的服务需要具备独立性及完整性

### 生产者消费者接口bean

demo :com.thunisoft.wsbq.api. TestController

各个service的接口的bean对象使用共同的，还是各自使用各自的，具体业务具体使用吧，各有利弊。

如果用共同的，则服务之间存在耦合，违背微服务化思想。

如果各自用各自的，会有管理混乱的情况。

## 断路器监控

TODO;

## 服务链路追踪

TODO;

## 消息总线

TODO;

# 部署实施

项目可以部署在 centos7环境下，可以使用docker容器部署，下面介绍具体实施步骤

## 操作系统

安装centos7操作系统（docker的新版在17.2，这类版本最好使用centos内核3.0以上的，所以要求是centos7）

操作系统安装过程这里就略过了 - -！

## 安装docker

详见官网：

<https://docs.docker.com/engine/installation/linux/docker-ce/centos/#set-up-the-repository>

如果登陆不上去，按照下面的流程走也可以！写文档时docker官网这个页面又登陆不上去了，我另找时间补充上，大概四五个命令行…

1 yum remove docker \

docker-client \

docker-client-latest \

docker-common \

docker-latest \

docker-latest-logrotate \

docker-logrotate \

docker-selinux \

docker-engine-selinux \

docker-engine

2 yum install -y yum-utils \

device-mapper-persistent-data \

lvm2

3 yum-config-manager \

--add-repo \

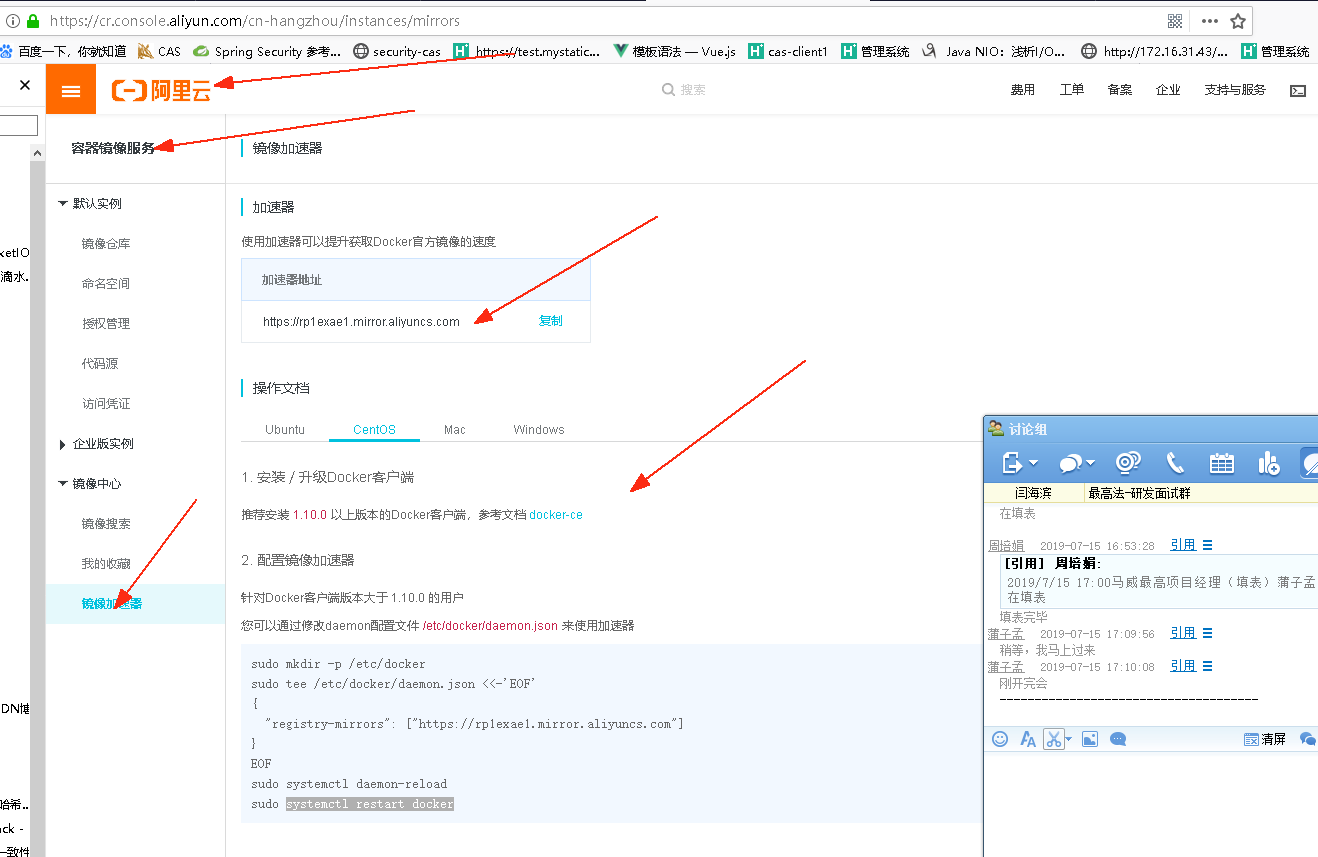
https://download.docker.com/linux/centos/docker-ce.repo

如果不成则改称阿里云的地址试试

yum-config-manager --add-repo http://mirrors.aliyun.com/docker-ce/linux/centos/docker-ce.repo

4 最后 yum install docker-ce

### 配置docker加速器



推荐安装1.10.0以上版本的Docker客户端，参考文档 [docker-ce](https://yq.aliyun.com/articles/110806)

配置镜像加速器

#### mkdir -p /etc/docker

tee /etc/docker/daemon.json <<-'EOF'

{

"registry-mirrors": ["https://rp1exae1.mirror.aliyuncs.com"]

}

#### systemctl daemon-reload

#### systemctl restart docker

## 制作docker镜像

### 创建dockerfile

首先在当前目录下创建Dockerfile，内容如下：

FROM lwieske/java-8:latest --来自于哪个基础镜像，这个可以从官网下载用docker pull

ENV JAVA\_OPTS="-server -Xms64m -Xmx64m -XX:+UseConcMarkSweepGC -XX:+UseFastAccessorMethods -XX:+UseParNewGC" –配置javajvm参数

ENTRYPOINT [ "sh", "-c", "java $JAVA\_OPTS -Djava.security.egd=file:/dev/./urandom -jar /usr/liuxing/app.jar" ] -指定启动的jar包

### 创建镜像

执行创建镜像的命令

docker build -t liuxingjavabase .

### 查看及删除

Docker images

Docker rmi 镜像名称 –删除镜像

## 新建docker容器

### 配置docker网络

使用Docker的macvlan为容器提供桥接网络及跨主机通讯

macvlan的原理是在宿主机物理网卡上虚拟出多个子网卡，通过不同的MAC地址在数据链路层（Data Link Layer）进行网络数据转发的，它是比较新的网络虚拟化技术，需要较新的内核支持，我们使用的是centos7，支持此种网络模式。

#### 1创建网络

docker network create -d macvlan --subnet=172.16.14.0/17 --gateway=172.16.1.1 -o parent=eth0 mcv

# 解释：

# 1.创建macvlan网络，使用macvlan网络驱动

# 2.指定要桥接的网络地址（问运维）

# 3.指定网关（问运维）

# 4.设置要在宿主机上的哪一块网卡上建立虚拟子网卡

#### 2设置网络模式 ：

开启为混杂模式

ip link set eno16777984 promisc on

#### 3 查看

Docker network list

### 创建容器

docker run --name wsbq-api -dit --net=mcv --privileged=true --ip=172.16.14.244 -v /opt/wsbqapi:/usr/liuxing:ro -v /var/log/applog:/var/log/applog:rw baoquanjavabase

#1 –d 指在后台运行

#2 –net 为指定网络模式

#3 设定true时，root拥有真正的root权限，否则root只是一个普通的外部用户

#4 指定ip

#5 挂载目录，前面的为宿主机，后面的为容器， ro含义为容器设定只读

#6--oom-score-adj=170 数值越大，越容易被kill

#7--oom-kill-disable 保护该容器进程不被杀死，此参数在实际环境中作用不大，因为进程通常默认都是此值，所以设置前面的参数更为有效

#8--memory=256m 限制容器支取宿主机内存的最大值

#9--cpu="1.5" 如果时两核的，则此设置含义为可以用1.5个，生产环境没用过这个配置，意义不大

查看容器启动日志

docker logs –f –tail 100 容器名

查看容器内存使用状况

docker stats

## 常用docker命令

docker inspect registry1 |grep IP --根据容器名查找容器信息。

docker stop registry1 && docker rm registry1 停止并删除容器

docker save -o lwieske-java-8.tar lwieske/java-8 保存docker镜像到本地

docker load -i lwieske-java-8.tar 从本地创建镜像

docker network list 查看网络

docker ps -a 查看所有容器

docker ps -s 查看运行中的容器

## 制作jar包

### 安装maven并配置maven环境变量

这个略了 --！

### 编译并打包

Dos进入项目目录下：

执行 mvn clean package --清除原有的文件并重新打包

或者执行 mvn clean package -DskipTests --清除原有的文件并跳过编译test重新打包

把打包的文件放到

docker run --name client1 -dit --net=mcv --privileged=true --ip=192.168.1.124 -v /opt/client1:/usr/liuxing:ro javabase3 中的 opt/client1下面，并把jar包名字改成 app.jar即可。

使用 Jenkins时也可以按照这个思路来处理。

# Springclouddemo演示

服务注册页面

<http://39.107.82.1:8761/>

登录页<http://39.107.82.1:8766/homepage>

服务调用demo （feign hystrix）

http://39.107.82.1:8766/user/sayHi01?id=111&name=adnihao&currentpage=1&pagesize=1

api网关

<http://39.107.82.1:8770/api-test1/hi?id=111&name=adnihao&currentpage=1&pagesize=1&token=11123>

# spring cloud 深挖掘

## 服务上下线实时刷新

https://blog.csdn.net/u010457081/article/details/88627926

解决办法： 1.服务下线升级时，通过DiscoveryClient的shutdown（）方法通知注册中心。

2.然后使用消息广播通知客户端即时刷新同步注册中心。

3.通知ribbon即时跟eureka client同步。

# 其它

## gitlab安装及使用

### 安装

#### 下载dockers镜像

|  |  |
| --- | --- |
|  | docker pull beginor/gitlab-ce:11.0.1-ce.0 |

#### 建立目录

通常会将 GitLab 的配置 (etc) 、 日志 (log) 、数据 (data) 放到容器之外， 便于日后升级， 因此请先准备这三个目录

mkdir -p /mnt/gitlab/etc

mkdir -p /mnt/gitlab/log

mkdir -p /mnt/gitlab/data

#### 建立容器

docker run --name gitlab -dit --net=mcv --privileged=true --ip=172.16.14.101 -v /mnt/gitlab/etc:/etc/gitlab -v /mnt/gitlab/log:/var/log/gitlab -v /mnt/gitlab/data:/var/opt/gitlab beginor/gitlab-ce:11.0.1-ce.0

#### 访问

<http://172.16.14.101/>

### 使用

注册，登陆后，新建项目，获取到项目路径后，就能在eclipse的git，share中使用这个项目了。