# 1 微服务知识

## 1 微服务组成

网关、服务注册与发现、配置中心、链路追踪、负载均衡、熔断

### 1.1 网关： 路由转发+过滤器

/api/v1/product/ 商品服务

/api/v1/order/ 订单服务

/api/v1/user/ 用户服务

### 1.2 服务注册发现： 调用和被调用方的信息维护

### 1.3 配置中心： 管理配置，动态更新 application.properties

### 1.4 链路追踪： 分析调用链路耗时

如：下单—》查询商品服务获取商品信息—》查询用户信息—》保存数据库

提交订单-查询用户信息-查询购物车信息-保存数据库

### 1.5 负载均衡： 分发负载

### 1.6 熔断： 保护自己和被调用方

### 1.7 常见微服务框架

consumer ：调用方

provider ：被调方

一个接口一般都会充当两个角色（但不会同时充当）

如：订单服务可以被用户服务调用（被掉方），然后回去调用数据看服务（调用方）

2.1 dubbo：zookeeper+dubbo+springmvc/springboot

官方地址：

配置套：

通信方式：RPC

注册中心：zookeeper/redis

配置中心：diamond

2.2springcloud：全家桶+轻松嵌入第三方组件（Netflix 奈飞）

通信方式：http restful

注册中心：eureka/consul

配置中心：config

断路器（熔断）： hystrix

网关：zuul

分布式追踪系统：sleuth+zipkin

## 3 微服务注册中心

### 3.1 理解注册中心：

管理服务，维护服务列表，心跳机制动态维护。

自我理解：有点像是一个大的配置文件（类似spring的配置文件，实现bean依赖的解耦），记录了调用方和被调用方的信息。然后，两者不直接获取对方获取信息。而是，都从该中心获取需要的信息，然后根据该信息去找目标服务。

从而，实现了调用方和被调方的解耦。便于维护和管理。

### 3.2 服务提供者provider：

启动时向注册中心上报自己的网络信息

### 3.3 服务调用者consumer：

启动时向注册中心上报自己的网络信息，拉取provider的相关网络信息

### 3.4 为什么要用注册中心：

微服务应用和机器越来越多，调用方需要知道接口的网络地址，如果靠配置文件的方式去控制网络地址。对其动态新增机器和维护带来很大问题。

### 3.5 主流的注册中心：

zookeeper Eureka consul etcd等

### 3.6 注册中的选择

zookeeper:CP设计，保证了一致性，集群搭建的时候，某个节点失效，则会进行选举新的leader（leader不提供服务），或者半数以上节点不可用，则无法提供服务，因此可用性没法满足。

ZK1

ZK1

ZK1

ZK1

ZK1

Eureka ：AP原则，不保证数据一致性。无主从节点，一个节点挂了，自动切换其他节点可以使用，去中心化。

结论：分布式系统中，P肯定要满足，所以只能在CA中二选一。

没有最好的选择，最好的选择就是根据业务场景来及进行框架设计。

如果要求一致性，则选择zookeeper，如金融行业。

如果要求可用性，则选择Eureka，如电商系统。

## 4 分布式应用知识CAP理论知识

### 4.1 CAP定理：

指的式在一个分布式系统中，一致性（Consistency）、可用性（Availability）、分区容错性（Partition tolerance），三者不可同时获得。只能获得其中两个。

一致性（C）： 在分布时系统中的所有数据备份，在同一时刻是否同样的值。（所有节点在同一时间的数据完全一致，越多节点，数据同步越耗时）

可用性（A）：负载过大后，集群整体是否还能响应客户端的读写请求。（服务一直可用，而且式正常响应时间）

分区容错性（P）：分区容错性，就是高可用性，一个节点崩溃了，并不影响其他的节点（100个节点，挂了几个，不影响服务，越多机器越好）

CAP理论就是说在分布式存储系统中，最多只能实现上面的两点。而由于当前的网络硬件肯定会出现延迟丢包等问题，所以分区容错性是我们必须需要实现的 。所以我们只能在一致性和可用性之间进行权衡。

### 4.2 为什么只能三选二？

C A 满足的情况下，P不能满足。

数据同步需要时间（C），同时又要在正常的时间内响应（A），就要求不能有太多的机器数量，所以P也就不能满足了。（一旦机器崩溃，服务可能无法进行）

C P 满足的情况下，A不能满足。

数据同步需要时间（C），机器数量又多（P），众多的机器同步数据会消耗大量时间，就不能在正常时间内响应（A）；

A P满足的情况下，C不能满足。

机器数量很多（P），又要正常时间内响应，那么数据就不能及时同步到其他节点，所以C不能满足。

# 2 Eureka ：server/client-服务注册与发现

## 5 SpringCloud 组件Eureka 和 闭源后影响

看本质：Eureka就是jar包而已，服务端jar包 和 客户端jar包

SpringCloud 体系：看官网 http://projects.spring.io/spring-cloud/

Eureka的基础知识—》 画图， 服务提供者< ---- >服务消费者；

Eureka-Server（Jar包）

Eureka-Client（Jar包）

订单服务

Eureka-Client（Jar包）

商品服务

注册与发现

注册与发现

从Server中获取需求服务的地址后，进行调用

Eureka 2.x闭源后选择：

相关文档学习：

https://www.jianshu.com/p/d32ae141f680

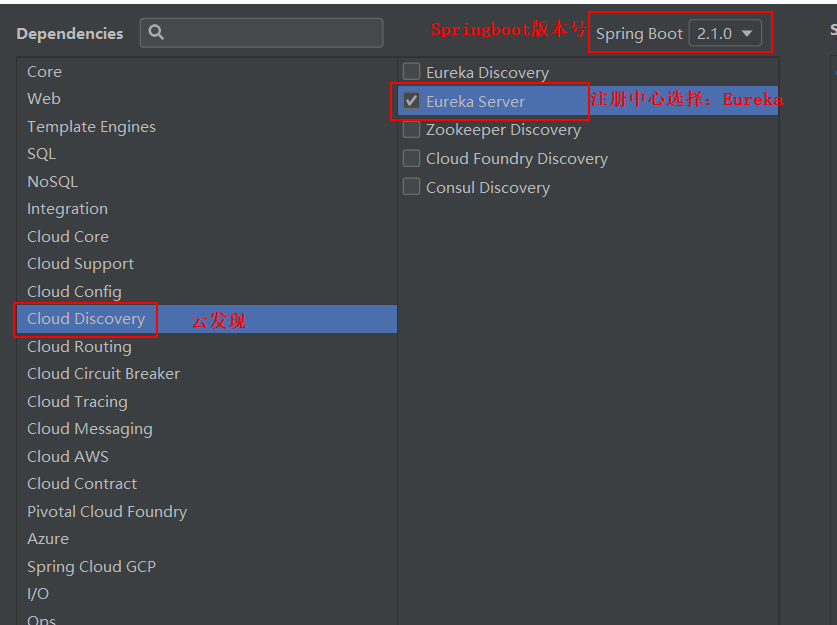
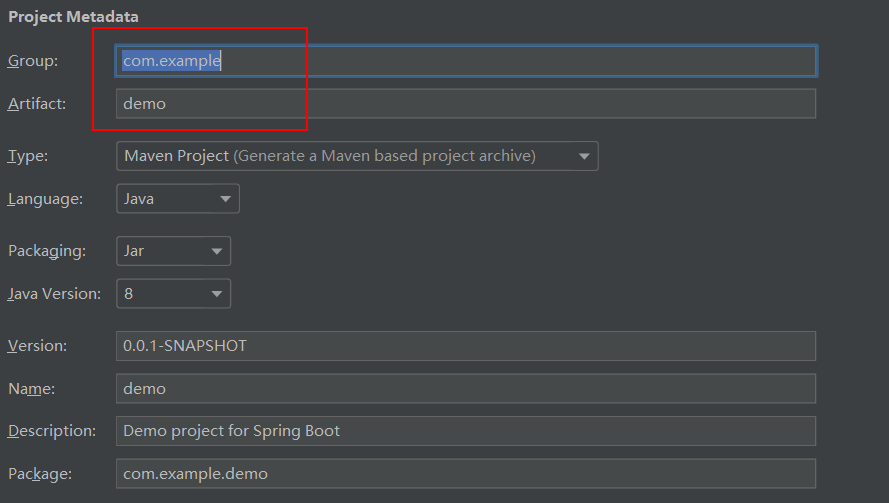
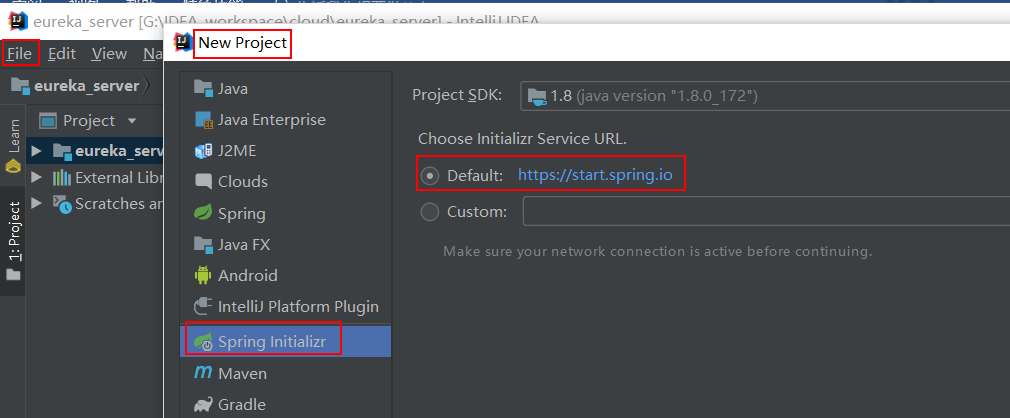
<https://blog.csdn.net/zjcjava/article/details/78608892>

## 6 服务注册与发现 Eureka -Server搭建

官方文档：

http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#spring-cloud-eureka-server

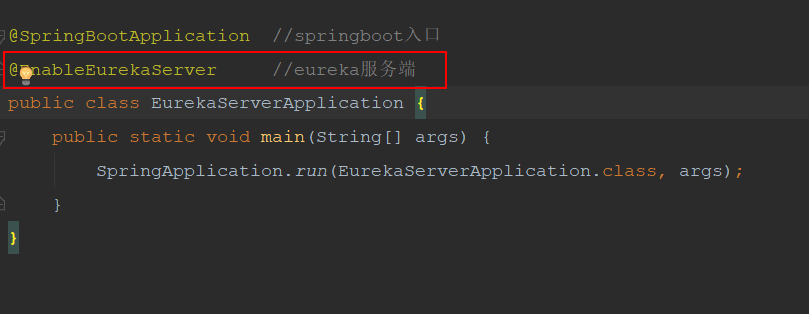
### 6.1 第一步：创建项目



会自动添加依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
</dependency>

### 6.2 第二步: 添加注解 @EnableEurekaServer



### 6.3第三步：增加配置application.yml

server:

port: 8761

eureka:

instance:

hostname: localhost

client:

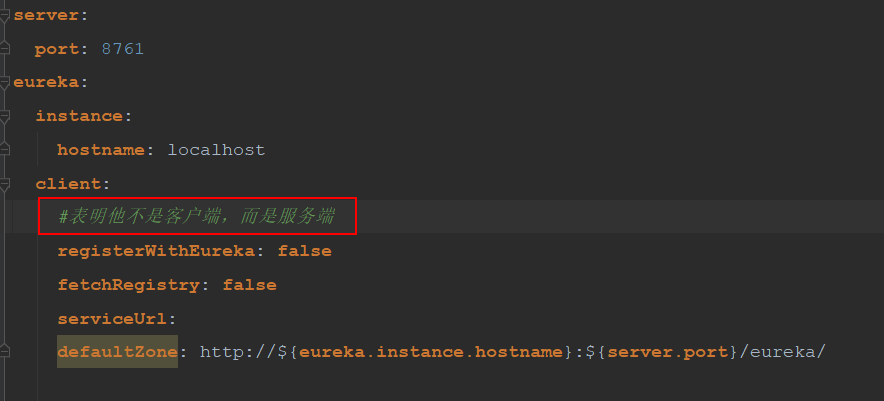
#声明自己是个服务端

registerWithEureka: false

fetchRegistry: false

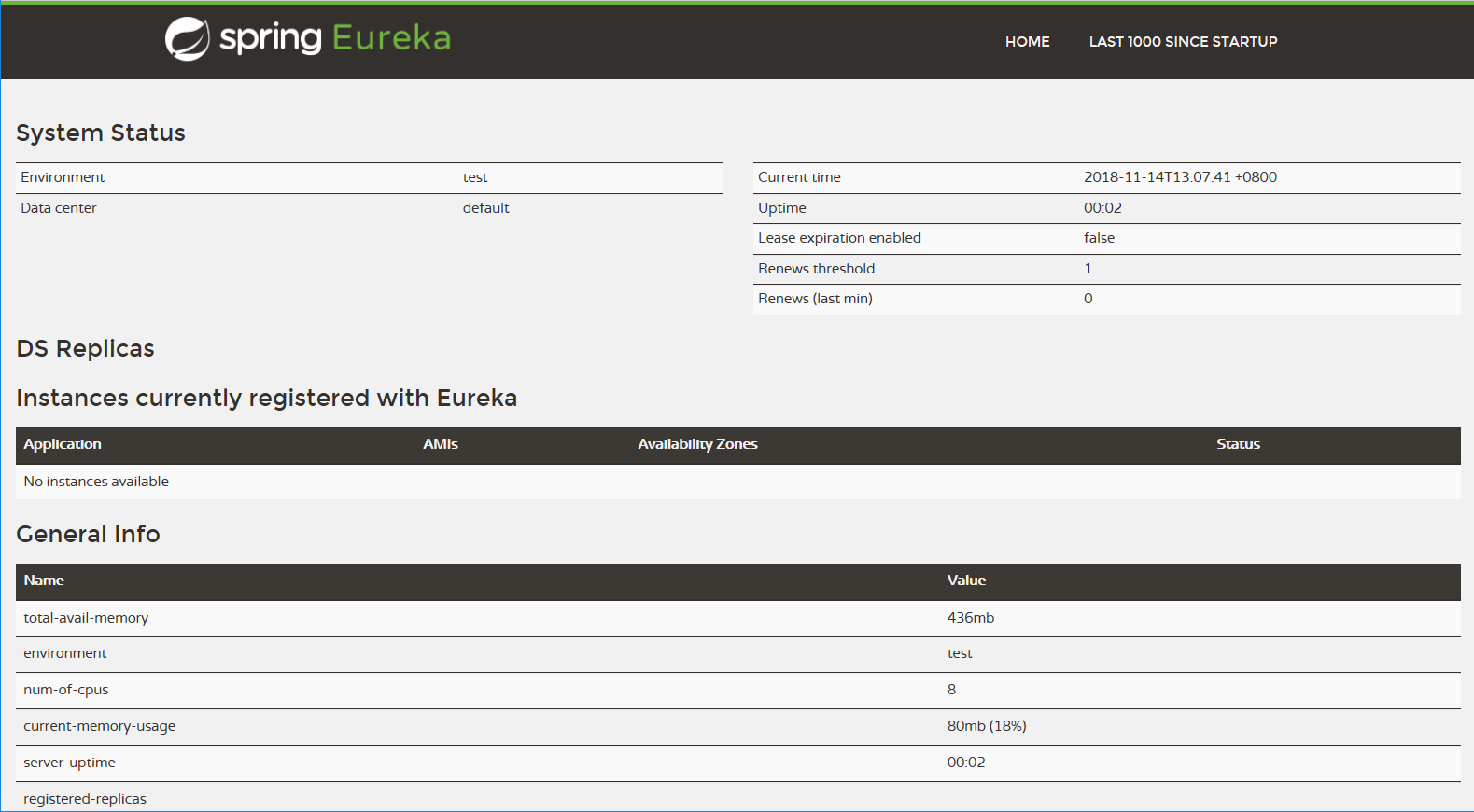
serviceUrl:

defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/



### 6.4 第四步：访问注册中心页面

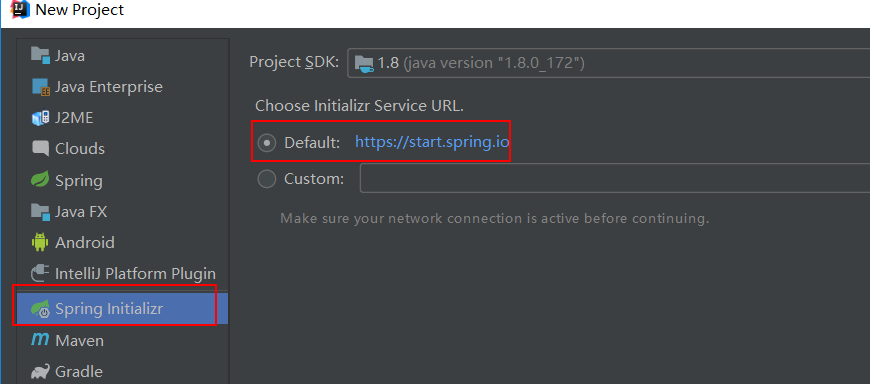
localhost:端口

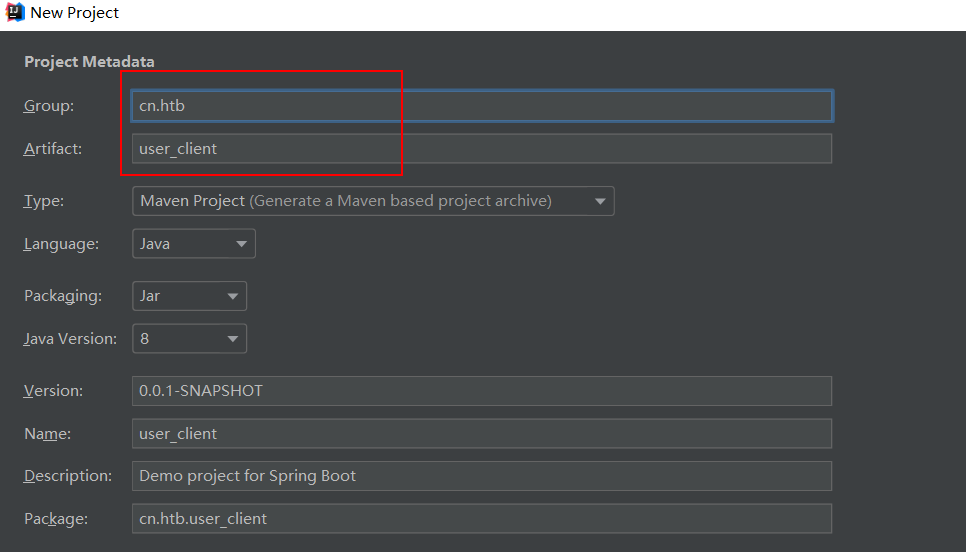


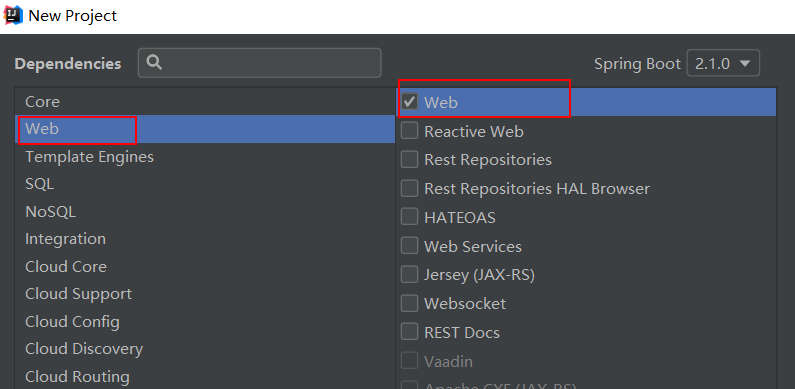
maven远程仓库地址修改: <https://www.cnblogs.com/sword-successful/p/6408281.html>

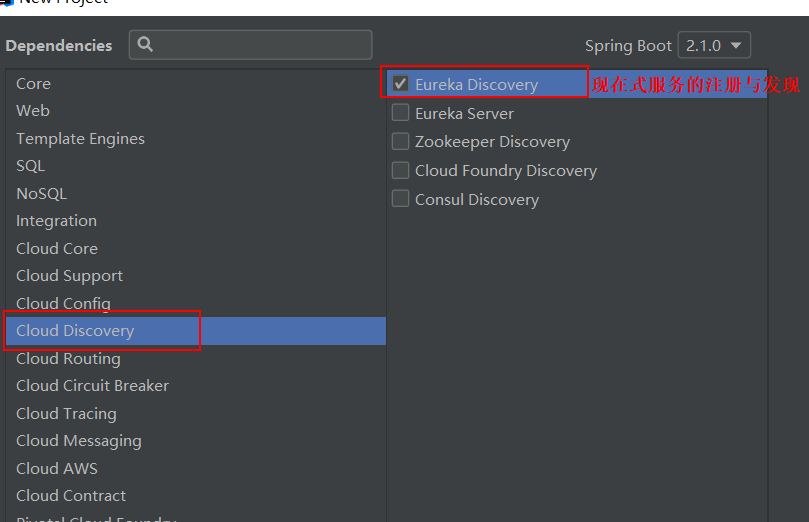
## 7 服务注册和发现之Eureka Client搭建

### 7.1 项目创建



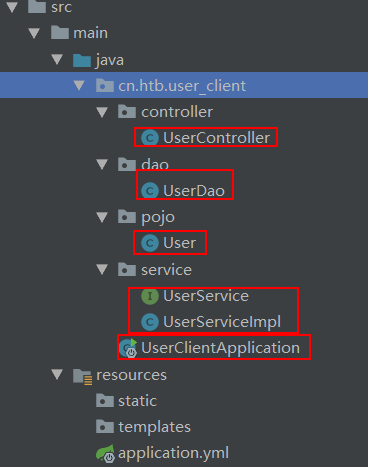




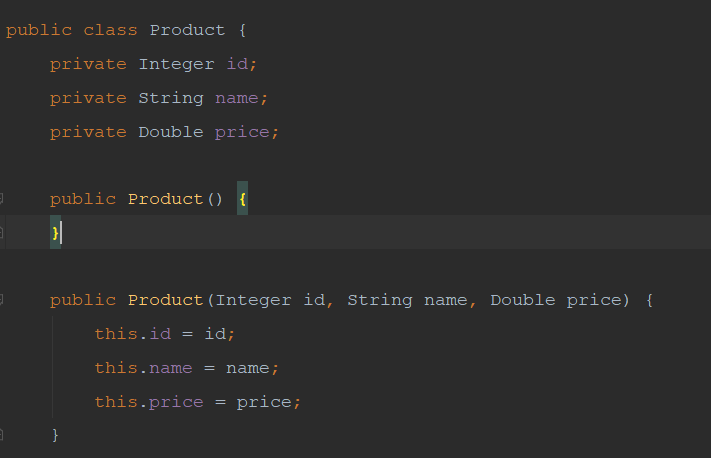


会自动添加依赖：  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
</dependency>

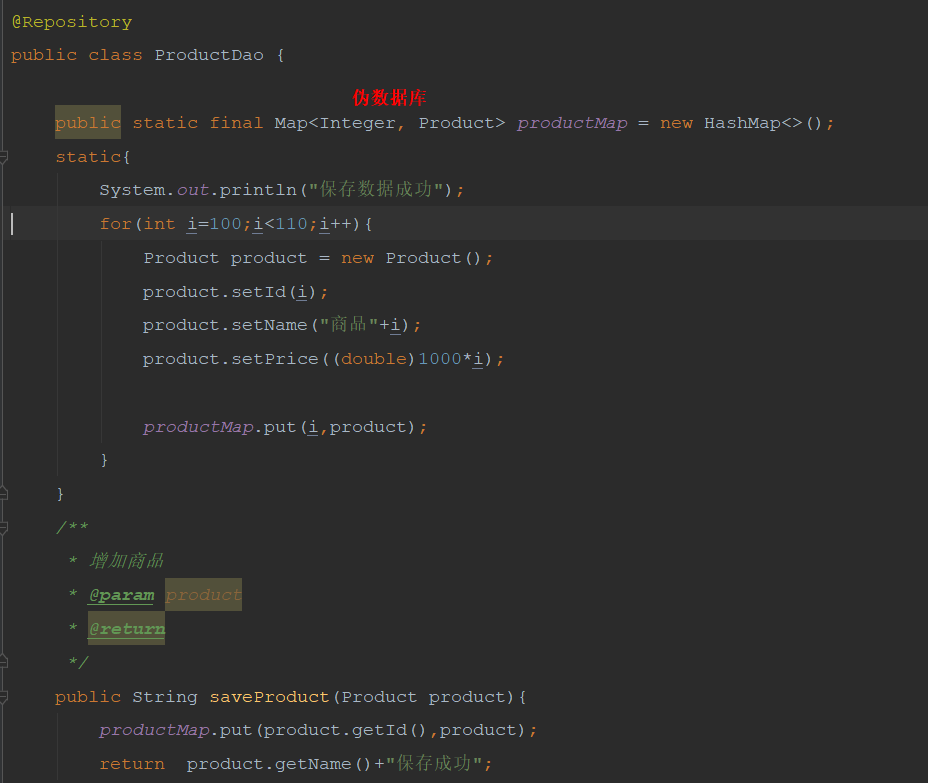
### 7.2 项目代码



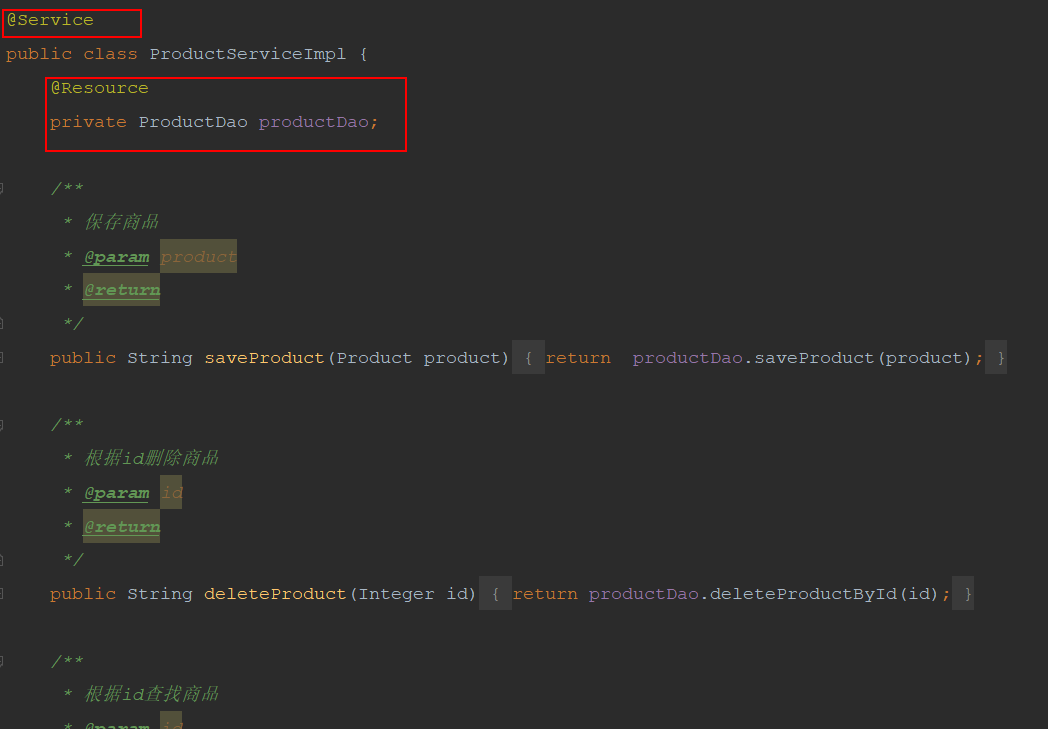
pojo



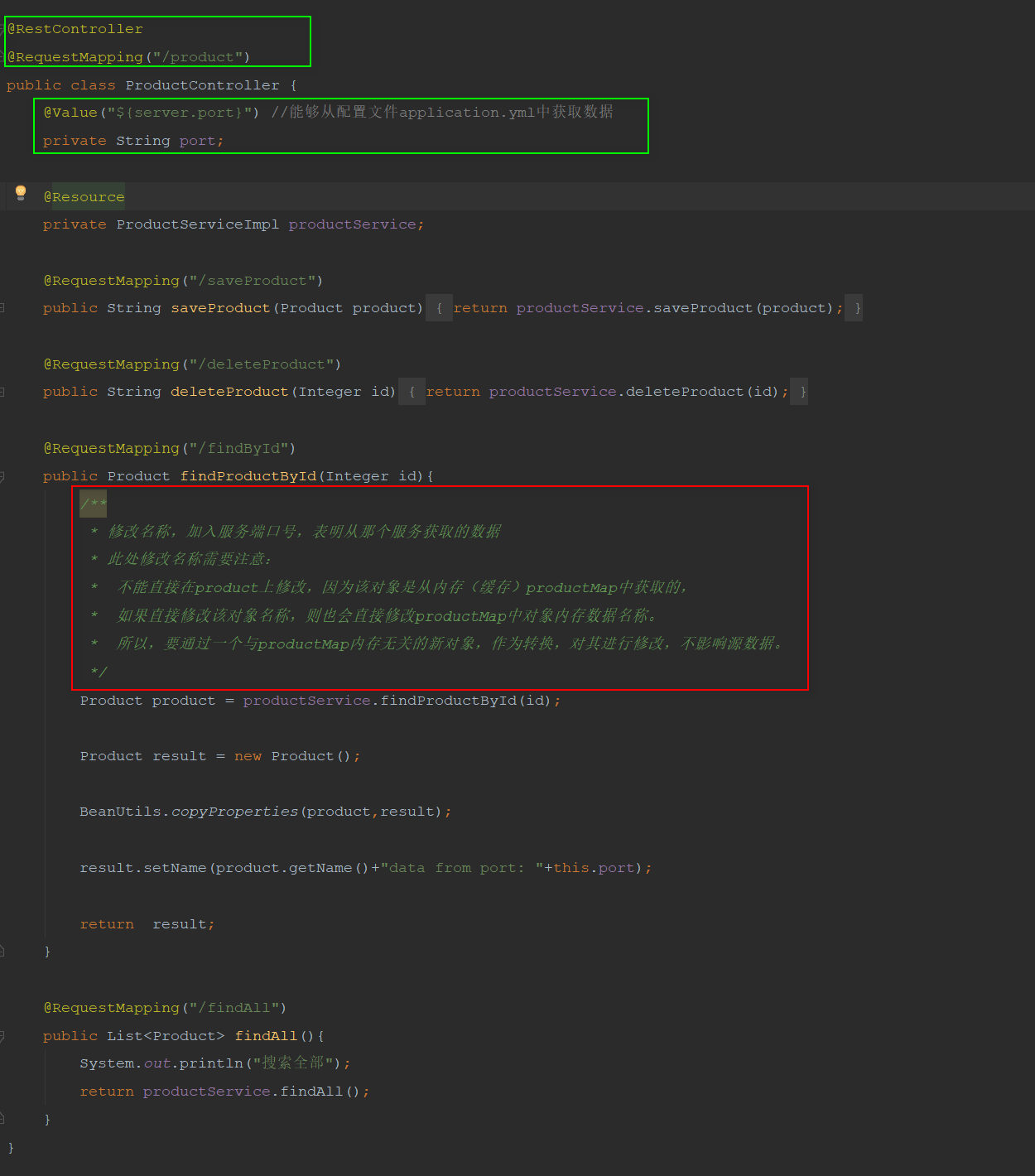
dao



service



controller

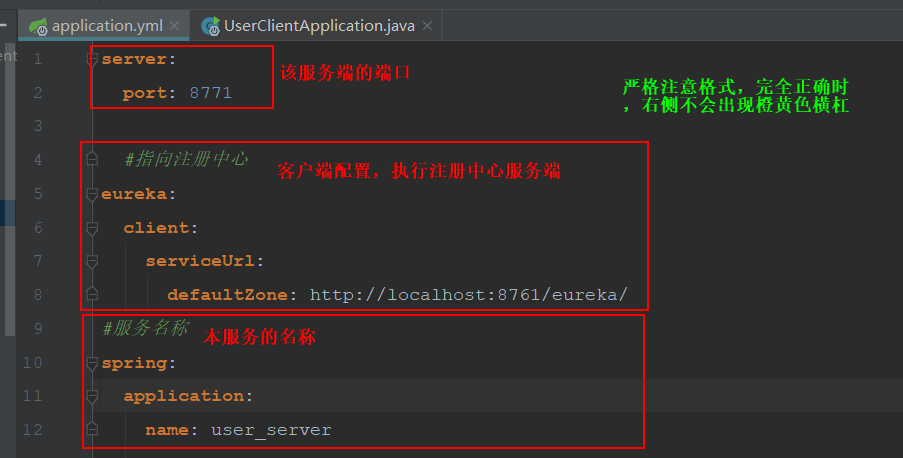


### 7.3配置文件加入注册中心地址

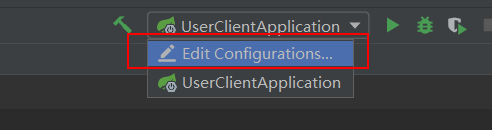
使用eureka客户端

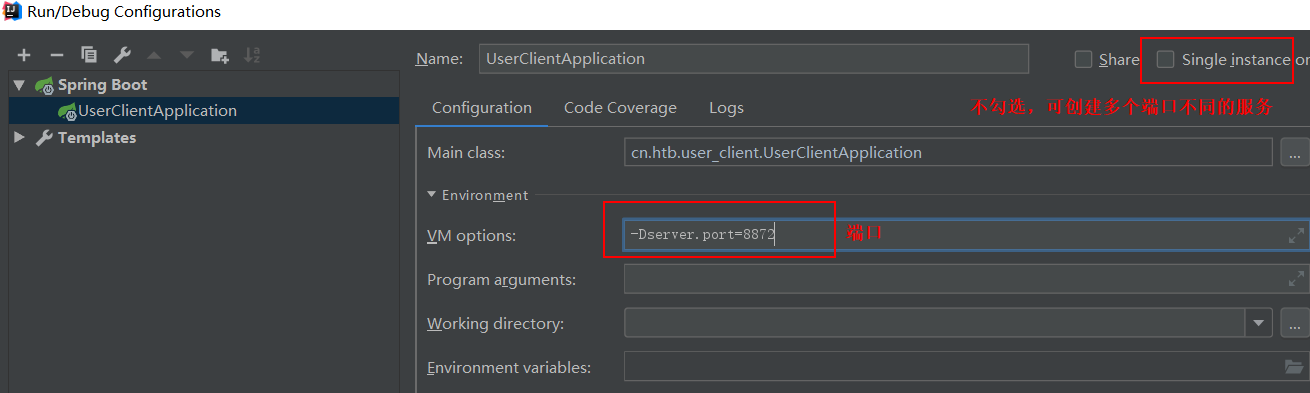
官方文档：

http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#netflix-eureka-client-starter

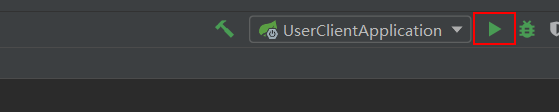


### 7.4 多服务启动





再次启动



## 8 Eureka服务注册中心配置控制台问题处理

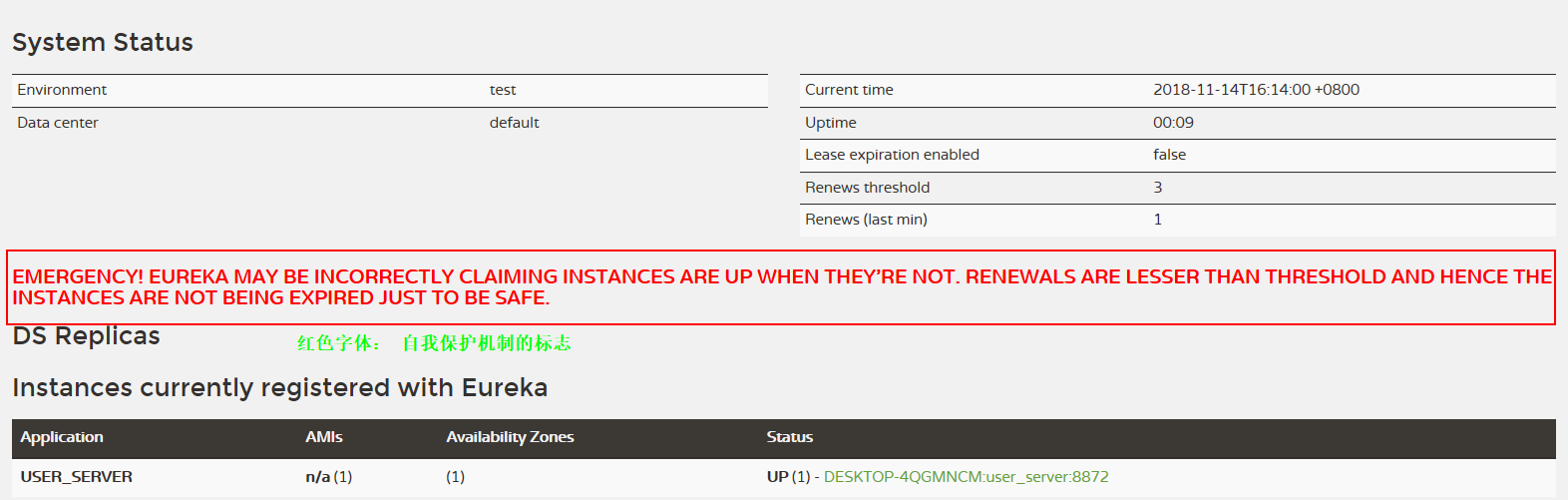
### 8.1 关闭Eureka自我保护机制（默认开启）

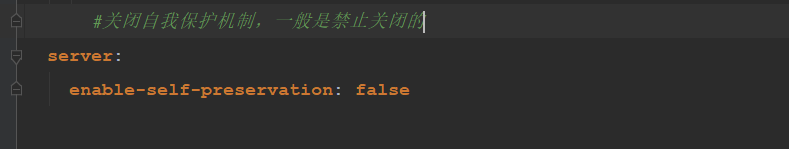
自我保护机制，就是当某种原因导致某个服务不能与服务注册中心通讯时，服务中心不会将其从服务列表中剔除，仍会正常的向需求方提供其网络地址。

（如因为网络波动，导致某个服务无法与注册中心通讯，但是该服务仍能正常工作，如果剔除了就会使该服务真的没用了；但不剔除的话，还能被正常调用；即使真的挂了，也能被捕获异常，进行处理）

如果，关闭保护机制，一旦服务不能与注册中心通讯，该服务将被从服务列表中剔除。

在Eureka的服务端的application.yml中添加：





关闭后，红色字体将改变，表明自我保护机制已关闭。一般禁止关闭。

### 8.2：为什么只加一个注册中心地址，就可以注册

By having spring-cloud-starter-netflix-eureka-client on the classpath, your application automatically registers with the Eureka Server. Configuration is required to locate the Eureka server, as shown in the following example:

# 3 Ribbon/Feign : 服务消费者调用方式

## Ribbon

### 9.1 常用的服务间调用方式

#### 9.1.1 RPC （Dubbo使用了这个）：

远程过程调用，像调用本地服务一样调用服务器的服务。

支持同步、异步调用

客户端和服务器之间建立TCP连接，可以一次建立一个，也可以多次调用服用一个连接。

#### RPC优点：

数据包小

protobuf

thrift

#### 缺点：编解码、序列化、连接问题、丢包问题、协议

#### 9.1.2 Rest（Http）：

优点：http请求，支持多种协议和功能

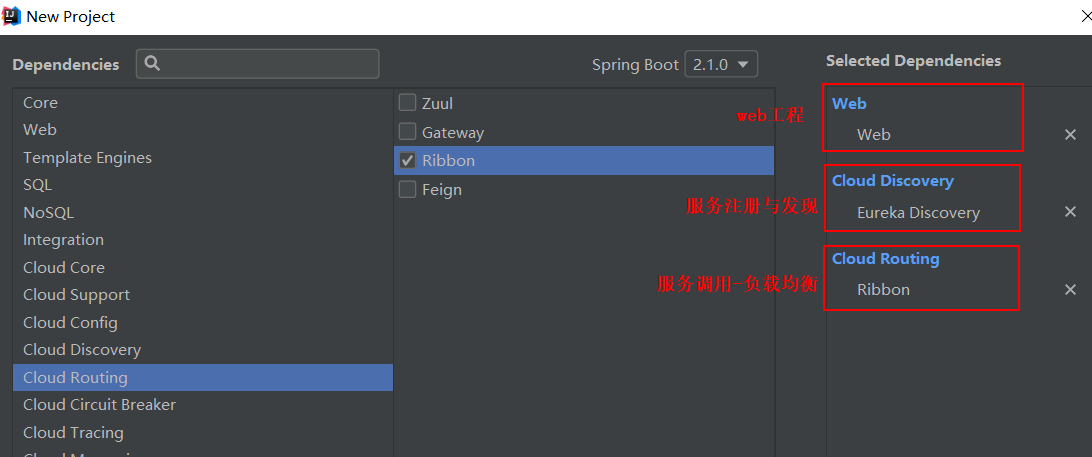
开发方便成本低

缺点：http数据包大

java开发：HttpClient，URLConnection

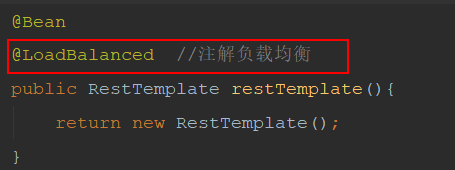
### 9.2 微服务调用方式之ribbon实战

#### 9.2.1 项目搭建

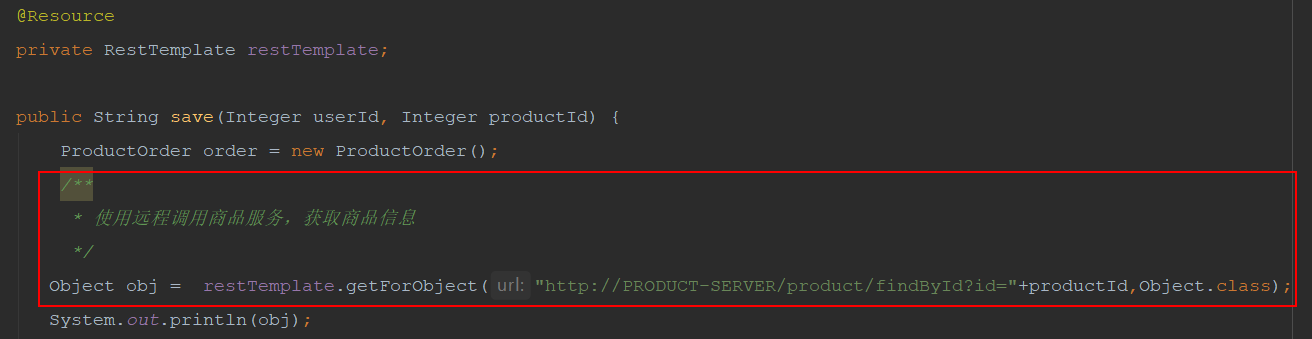


#### 9.2.2 项目代码 与 远程调用

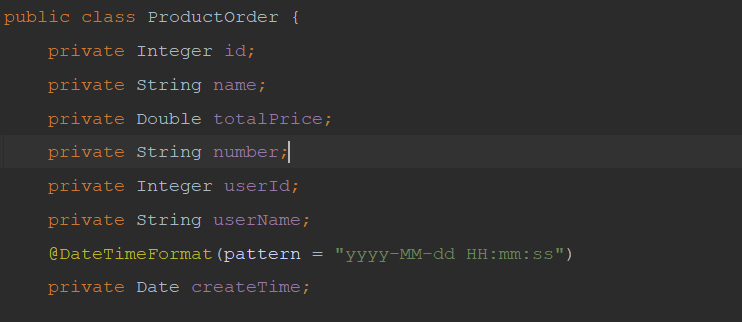
重点：启动文件中负载均衡配置



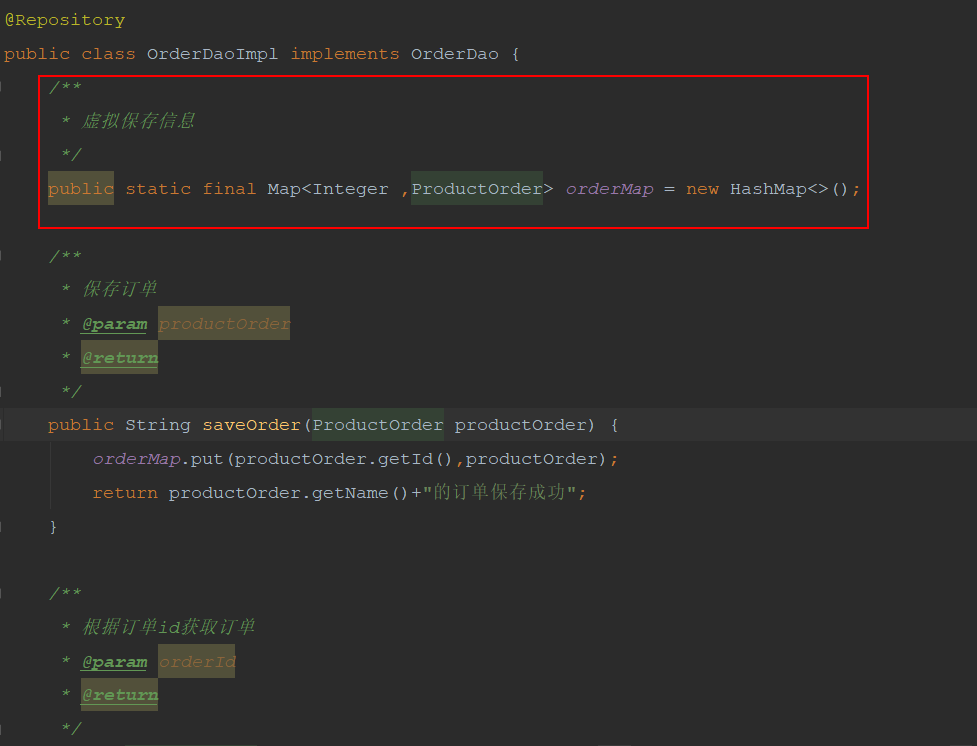
远程服务调用：RestTemplate



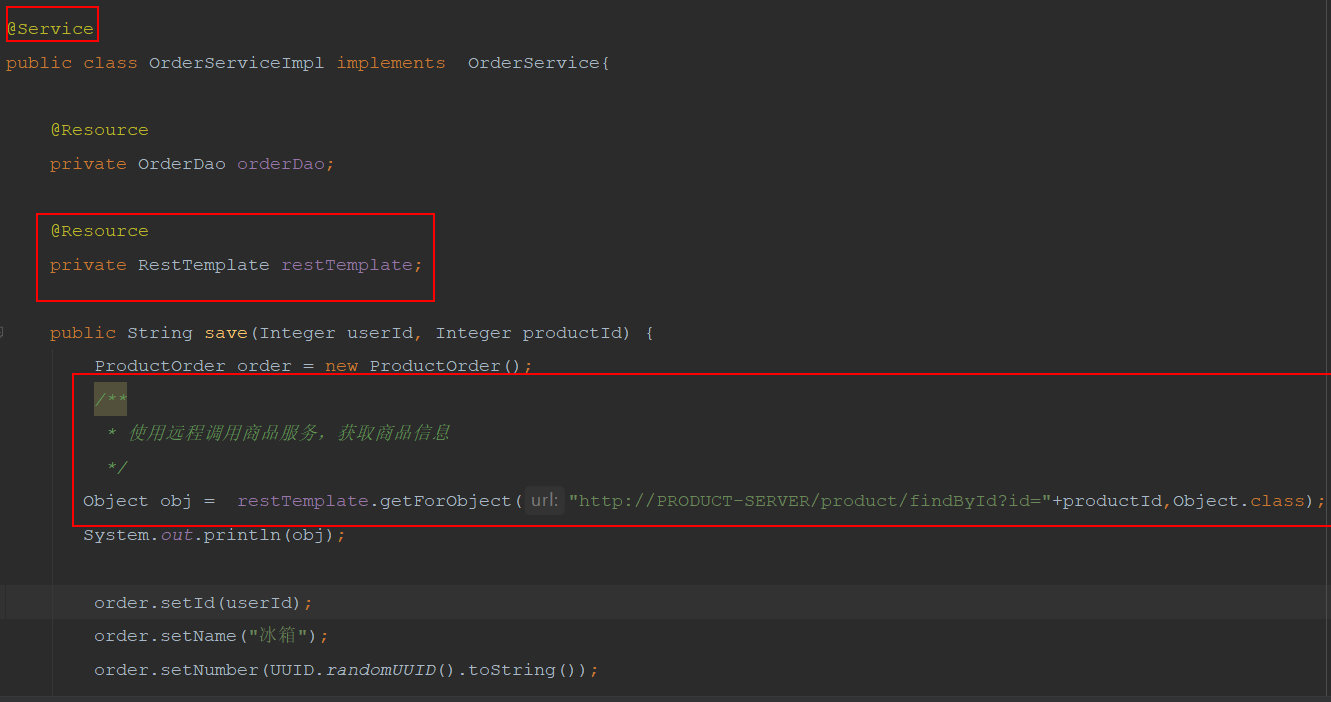
pojo

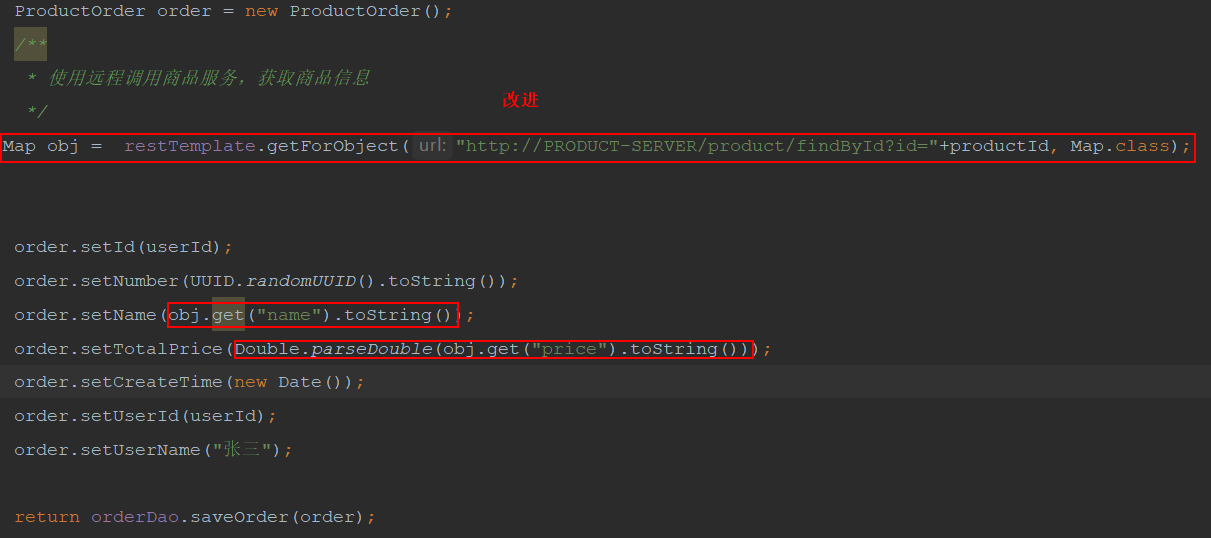


dao

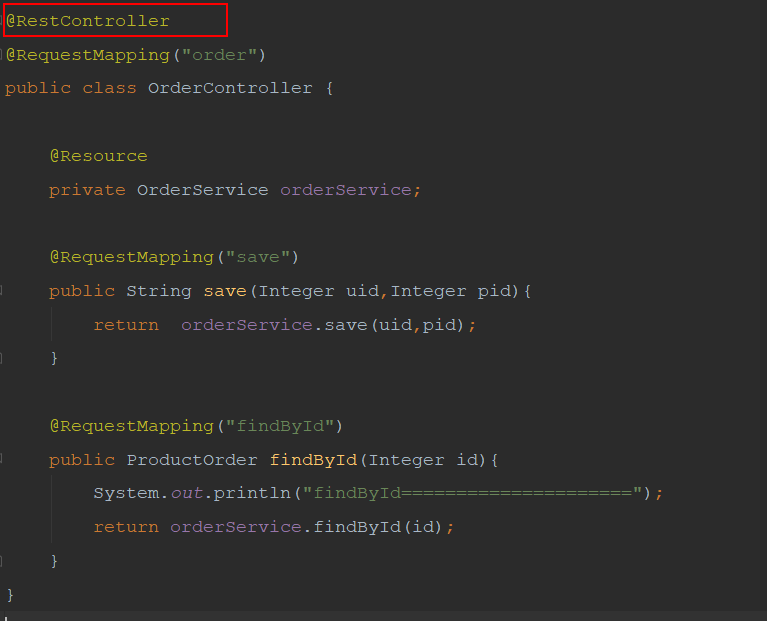


service





controller



启动文件：



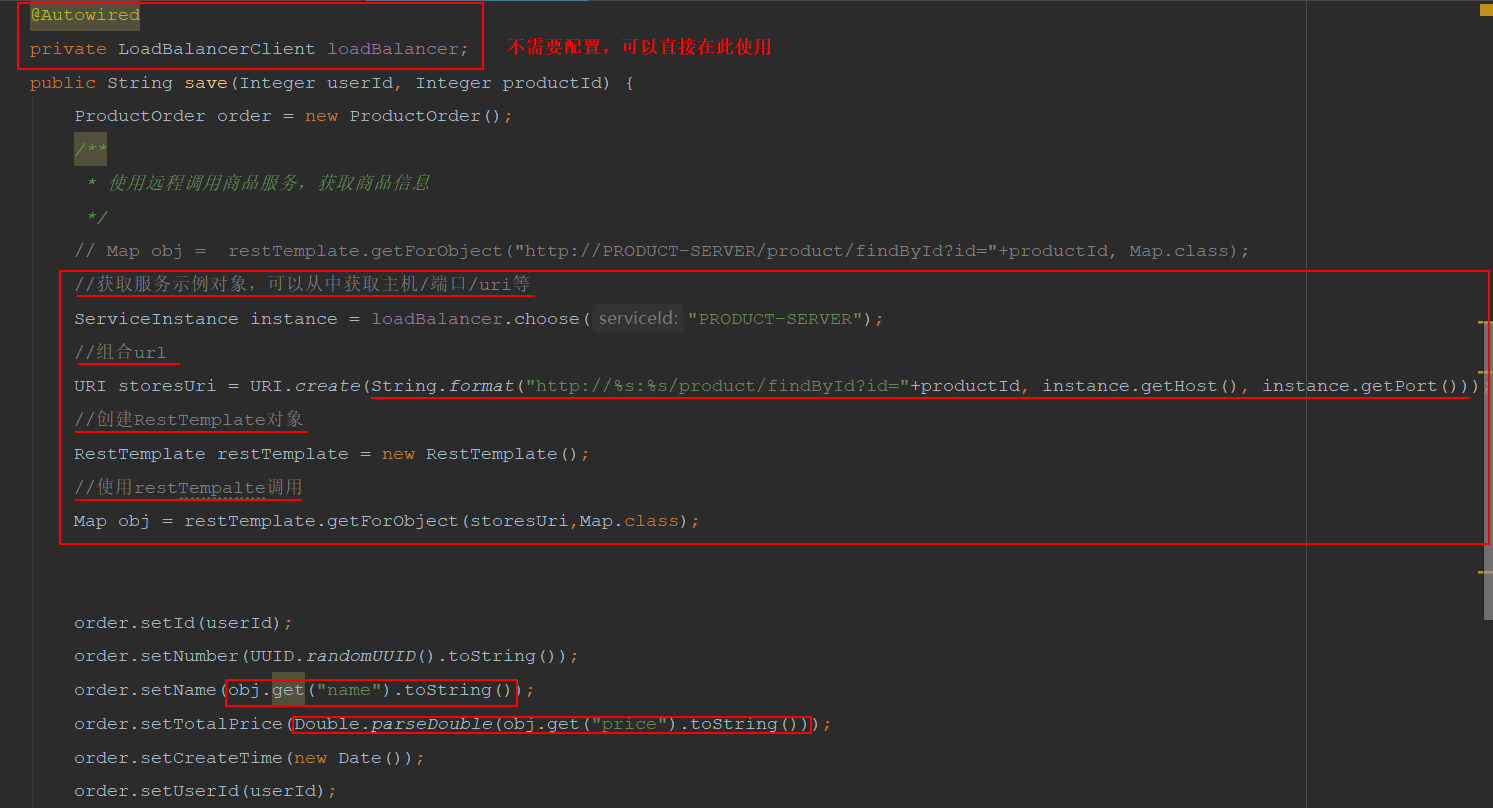
#### 9.2.3 注册信息配置



### 9.3 Ribbon实现负载均衡源码分析

#### 9.3.1 了解实现ribbon的另一种方法LoadBalancerClient

其中，%s是参数占位符。



#### 9.3.2 RestTemplate底层就是对LoadBalancerClient的封装

##### 9.3.2.1 设计思路：

1）getServer（）从Eureka中获取注册的服务列表

2）LoadBalancerClient通过一定策略获得指定服务地址并返给RestTemplate

ServiceInstance instance = loadBalancer.choose("PRODUCT-SERVER");

3）RestTemplate调用该地址

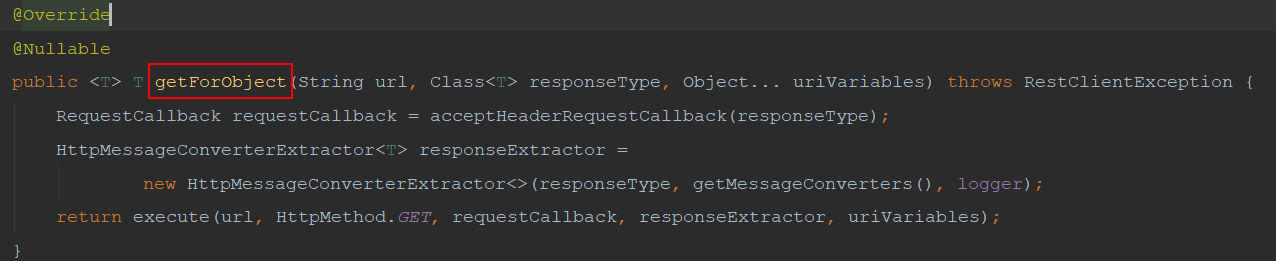
Map obj = restTemplate.getForObject(storesUri,Map.class);

##### 9.3.2.2过程分析

1 RestTemplate.getForObject()

->获取返回服务地址

->调用服务地址



UriComponentBuilder

->获得host,也就是serverId

2 RibbonLoadBalancerClient implements LoadBalancerClienet

->先获取到所有服务

3 BaseLoadBalancer

-> 调用this.rule.choose(key);

4 PredicateBasedRule

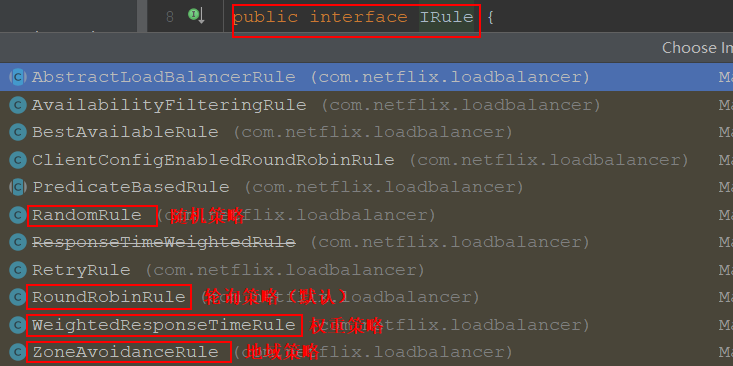
->Server choose(key)根据策略获取一个server

### 9.4 自定义均衡策略



*#自定义Ribbon的均衡策略*

**product-server**:  
 **ribbon**:  
 **NFLoadBalancerRuleClassName**: com.netflix.loadbalancer.RandomRule





策略选择：

1、如果每个机器配置一样，则建议不修改策略 (推荐)

2、如果部分机器配置强，则可以改为 WeightedResponseTimeRule

## Feign

### 9.1 依赖引入

Feign： 伪RPC客户端(本质还是用http)

官方文档: https://cloud.spring.io/spring-cloud-openfeign/

使用feign步骤讲解（新旧版本依赖名称不一样）

加入依赖

<dependency>

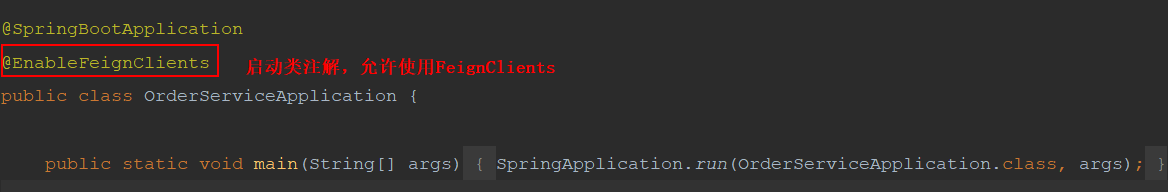
<groupId>org.springframework.cloud</groupId>

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

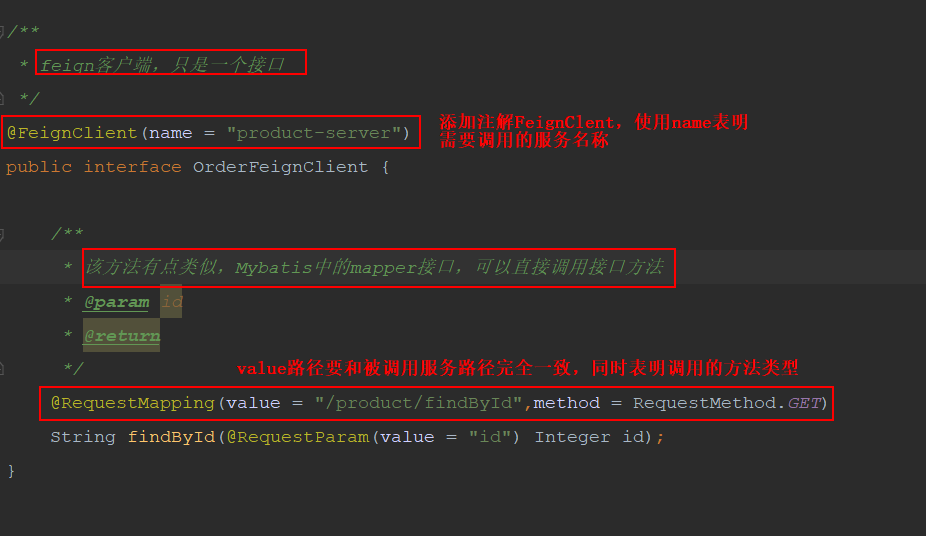
</dependency>

### 9.2 入口代码

启动类增加@EnableFeignClients



### 9.3 新增接口并@FeignClient(name="product-service")





### 9.4注意点：

1、路径要与被调用者路径完全一样。

2、Http方法必须对应

3、如果使用了RequestBody，应该使用@PostMapping

4、多个参数的时候，通过@RequestParam（"id") int id)方式调，

如果参数为对象时，要使用@ResqestBody注解

注意：消费端：feingclient要添加注解@RequestParm 和 @RequestBody

服务提供着同样要添加，不然就会报错：

404 是方法为找到

405 是请求方式不正确

### 9.5 feign 连接时间和超时时间配置

一般不要超过两秒

*#feignClient 连接时间和超时时间配置*

**feign**:  
 **client**:  
 **config**:  
 **default**:  
 **connectTimeout**: 2000  
 **readTimeout**: 2000

### 9.5 日志

官方文档有介绍



## Ribbon和Feign的选择

### 1、ribbon和feign两个的区别和选择： 选择feign

使用配置分析：

Ribbon：

1）配置bean：RestTemplate

2）在调用方服务类中注入restTemplate(这就耦合了：在调用方调用时，getForObject(uri,resultType,values),其中的url中包含了被调用方的信心，所以是耦合了)

Feign：

1) 两个注解：入口类@EnableFeignClients 服务接口类@FeignClient

2) 调用方中注入接口，直接调用接口方法（实现了解耦：在调用方内不包含被调用方的信息，其信息都在FeignClient接口中，所以可以实现解耦，而且代码增加清晰）

3) 默认集成了ribbon

综上： 选择Feign

默认集成了ribbon

写起来更加思路清晰和方便

采用注解方式进行配置，配置熔断等方式方便

### 2、超时配置

默认optons readtimeout是60，但是由于hystrix默认是1秒超时

*#feignClient 连接时间和超时时间配置*

**feign**:  
 **client**:  
 **config**:  
 **default**:  
 **connectTimeout**: 5000  
 **readTimeout**: 5000

### 3 模拟接口响应慢，线程睡眠新的方式

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

# 4 Hystrix : 降级、熔断 -互联网架构服务

## 4.1 熔断/降级概念理解

熔断和降级:当系统负载过高，突发流量或网络等异常情况时，常用解决方法。

### 4.1.1 熔断

类似保险丝，当某服务发生异常时熔断该异常服务，不再对其进行调用，防止整个系统故障，包含自己和下游服务。

如：订单服务 -> 调用商品服务

-> 调用用户服务（发生异常）->熔断该用户服务

虽然，没了用户的详细信息，但是仍然能完成订单。（只不过可能缺少一些东西，如：用户判断/会员打折等）

### 4.1.2 降级

断舍离，抛弃一些非核心的接口和数据。

如：天猫双十一，流量非常大。则在商品展示页，可以暂时不提供退货、评论功能（数据），只展示核心的商品数据。

### 4.1.3 熔断和降级对比

相同点：

1）都是为了防止系统崩溃，从可用性和可靠性出发。

2）最终结果，让用户体验到的是某些功能暂时不可用。

不同的：

1）熔断服务：一般是下游服务异常导致的，由服务被调方导致。

2）降级服务：一般是从整体系统负荷考虑，由服务调用方控制。

## 4.2 NetFlix开源组件断路器Hystrix介绍

文档地址：

<https://github.com/Netflix/Hystrix>

<https://github.com/Netflix/Hystrix/wiki>

### 4.2.1 Hystrix

Hystrix 对应中文是“豪猪”的意思，一种全身都是刺的动物，用来自我保护。

在分布式应用中，Hystrix的作用也是保护自己。

### 4.2.2 为什么使用Hystrix

在一个分布式系统里，一个服务依赖多个服务，可能存在某个服务调用失败。比如：超时、异常等。

如何能保证一个依赖服务出问题的情况下，不会导致整个系统崩溃，通过Hystrix就可以来实现。

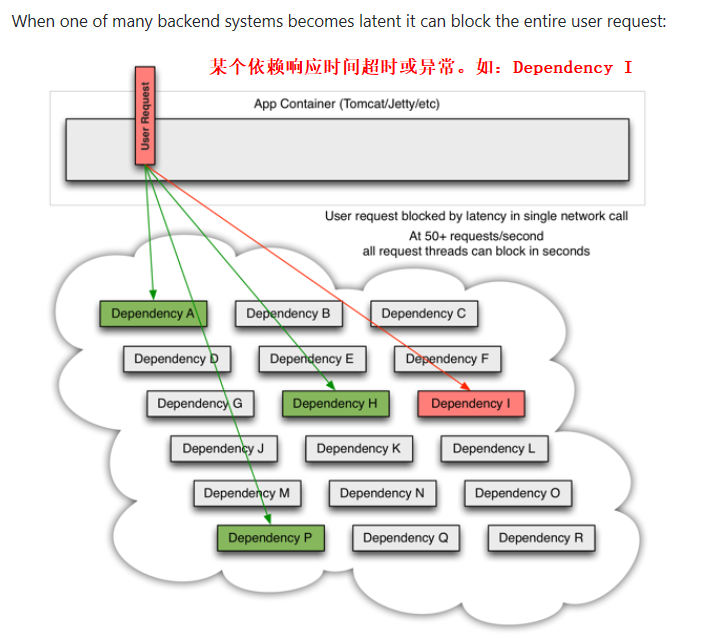
<http://cloud.spring.io/spring-cloud-netflix/single/spring-cloud-netflix.html#_circuit_breaker_hystrix_clients>

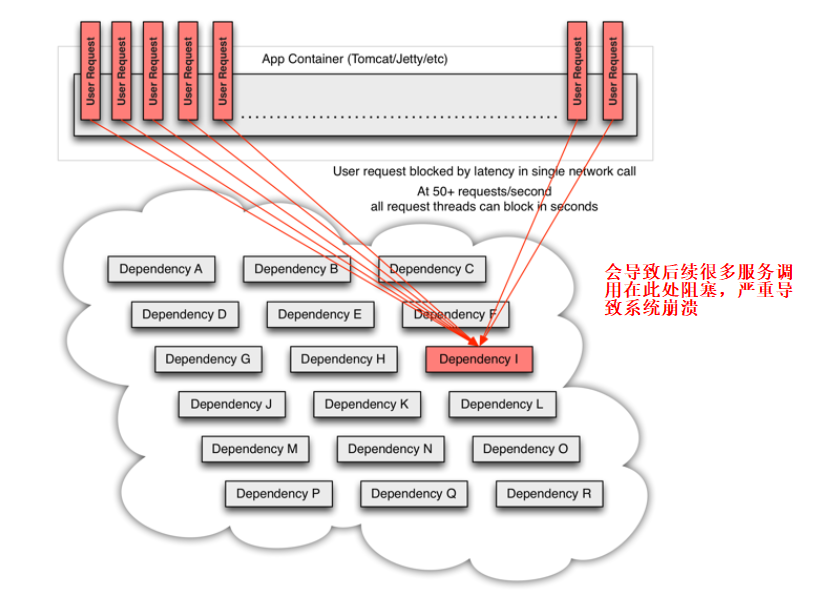
### 4.2.3 Hystrix 提供了熔断、隔离、Fallback、cache、监控等功能。

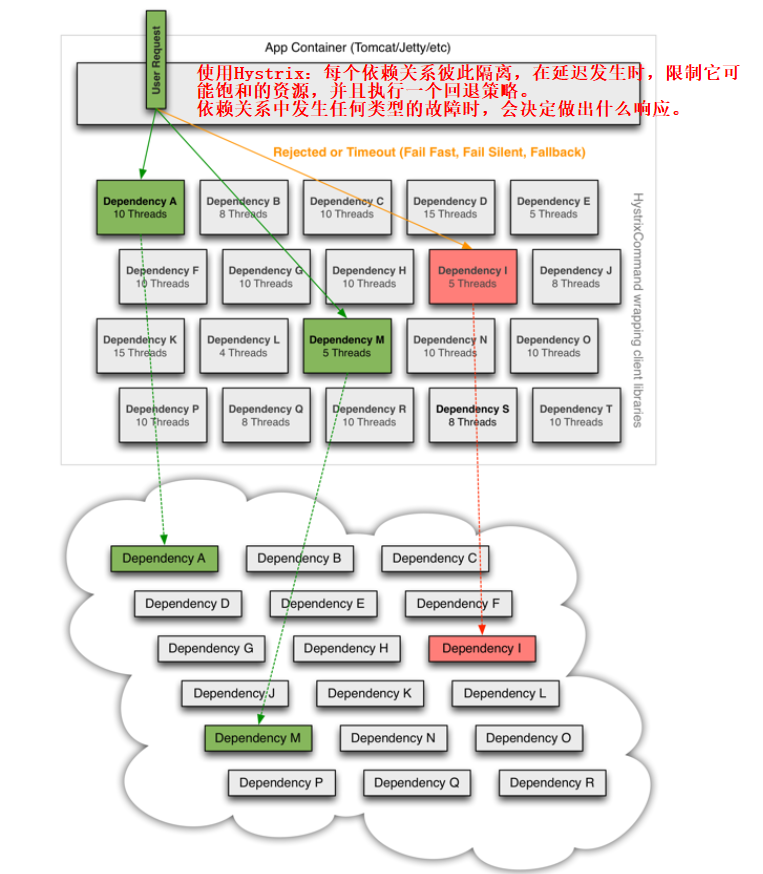
### 4.2.4 容断了怎么办？

出现错误之后可以 fallback处理错误信息 或 一些兜底数据









## 4.3 Hystrix基本使用方法

### 4.3.1 引入依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>  
</dependency>

### 补充： 修改maven仓库地址

pom.xml中修改

<repositories>

<repository>

<id>nexus-aliyun</id>

<name>Nexus aliyun</name>

<layout>default</layout> <url>http://maven.aliyun.com/nexus/content/groups/public</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<enabled>true</enabled>

</releases>

</repository>

</repositories>

### 4.3.2 启动类添加注解@EnableCircuitBreaker

@SpringBootApplication  
@EnableFeignClients  
@EnableCircuitBreaker   
public class OrderServiceApplication {

注解越来越多-》 SpringCloudApplication注解

包含了@SpringBootApplication和@EnableCircuitBreaker

### 4.3.3 Controller添加处理方法

熔断-》降级

方式一：直接在Controller中写对应方法的异常处理措施

1）最外层api使用，好比异常处理（网络异常，参数或者内部调用问题）

api方法上增加 @HystrixCommand(fallbackMethod = "saveFail")

注意：fallBack(参数。。)中的参数要和，被注解方法的参数一致

*\* 要注意:该方法的签名要和被处理方法的签名完全一样（返回值-参数）  
 \* 1 该方法的参数要和被处理方法的参数完全一样，即使不使用参数。  
 \* 2 该方法的返回值也要和被处理方法一样。*

@HystrixCommand(fallbackMethod = "fallBack")  
 @RequestMapping("/getTest2/{data}/{data2}")  
 public String getTest2(@PathVariable(name = "data") String data, @PathVariable(name = "data2") String data2){  
 System.*err*.println(data + data2);  
 return myFeignClient.getProvider(data,data2);  
 }  
  
 @HystrixCommand(fallbackMethod = "fallBack")  
 @RequestMapping("/getTest/{data}/{data2}")  
 public String getTest(@PathVariable(name = "data") String data, @PathVariable(name = "data2") String data2){  
 System.*err*.println(data + data2);  
  
 Map map = new HashMap<>();  
 map.put("data",data);  
 map.put("data2",data2);  
 ResponseEntity<String> entity = restTemplate.getForEntity("http://USER-PRVODER/getProvider/{data}/{data2}", String.class,map);  
  
 String body = entity.getBody();  
 System.*out*.println(body);  
 return body;  
 }  
 @RequestMapping("/fallBack") // 可以不用添加该注解  
 public String fallBack(String data,String data2){  
 return "服务器暂时走丢了。。。。";  
 }  
}

2）编写fallback方法实现，方法签名一定要和api方法签名一致（注意点！！！）

*/\*\*  
 \* 使用Feign  
 \** ***@param*** *uid  
 \** ***@param*** *pid  
 \** ***@return*** *\*/*@RequestMapping("save2")  
@HystrixCommand(fallbackMethod = "saveFail")  
public Map save2(Integer uid,Integer pid){  
 Map map = new HashMap();  
 map.put("code",0);  
 map.put("data",orderService.save2(uid,pid));  
 return map;  
}  
  
*/\*\*  
 \* 当保存方法异常时，调用该方法进行处理  
 \* 要注意:该方法的签名要和被处理方法的签名完全一样（返回值-参数）  
 \* 1 该方法的参数要和被处理方法的参数完全一样，即使不使用参数。  
 \* 2 该方法的返回值也要和被处理方法一样。  
 \** ***@param*** *uid  
 \** ***@param*** *pid  
 \** ***@return*** *\*/*public Map saveFail(Integer uid, Integer pid){  
 Map map = new HashMap();  
 map.put("code",-1);  
 map.put("msg","你好，请稍后再试");  
 return map;  
}

## 4.4 Hystrix 结Feign 使用(熔断降级方式二Hystrix+feignClient)

两种方法：选择方法二，解耦，是的结构更加清晰。这也是选择Feign作为服务调用方式的原因之一：方便配置熔断降级。

### 4.4.1 开启Feign整合Hystrix

*#开启整合hystrix*

**feign**:**hystrix**:  
 **enabled**: **true**

### 4.4.2 在FeignClient上添加属性fallback=xxx.class

@FeignClient(name = "USER-PRVODER",fallback = FallbackClent.class)

*官方注解文档*

*/\*\*  
 \*指定的Feign Client接口的回调类，必须实现那个被该注解注解的接口，而且这个类必须是个可用的spirng bean（也就是这个回调类要交个spring管理，可以添加@Component，实现控制反转）*

*Fallback class for the specified Feign client interface. The fallback class must implement the interface annotated by this annotation and be a valid spring bean.  
 \*/*Class<?> fallback() default void.class;

### 4.4.3 创建异常处理类xxx.class,继承FeignClient注解的类

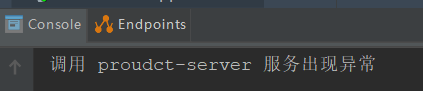
@Component

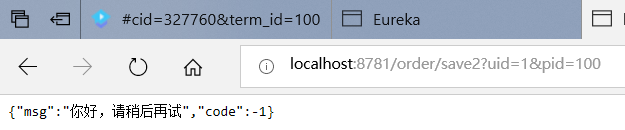
public class ProductClientFallback implements ProductFeignClient {  
  
 public String findById(Integer id) {  
 System.*out*.println("调用 proudct-server 服务出现异常");  
 return null;  
 }  
}

### 4.4.4 实现过程：

结合上述示例分析：当该服务异常或超时时，会调用该方法在控制台打印：调用 proudct-server 服务出现异常；然后，在controller中在调用另一个回调方法，显示信息给页面。

结果如下：





## 4.5熔断降级服务异常报警通知

### 4.5.1 加入redis依赖

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
</dependency>

### 4.5.2 配置redis链接信息

*#redis 数据库配置*

**spring**:  
 **redis**:  
 **database**: 0  
 **host**: 192.168.153.130  
 **port**: 6379  
 **timeout**: 2000

### 4.5.3 在异常处理信息方法使用

public Map saveFail(Integer uid, Integer pid, HttpServletRequest request){  
  
 String saveOrderKey = "save-order";  
 String sendValue = redisTemplate.opsForValue().get(saveOrderKey);  
  *//jdk 1.8 新特性Runnable*

new Thread( ()->{  
 if (StringUtils.*isBlank*(sendValue)) {  
 final String ip = request.getRemoteAddr();  
 System.*out*.println("紧急短信，用户下单失败，请离开查找原因,ip地址是="+ip);  
 //发送一个http请求，调用短信服务 *TODO*

*//第一个参数是key，第二个参数是value，第三个参数是失效时间，第四个参数是时间的单位：秒* redisTemplate.opsForValue().set(saveOrderKey, "save-order-fail", 20, TimeUnit.*SECONDS*);  
 }else{  
 System.*out*.println("已经发送过短信，20秒内不重复发送");  
 }  
 }).start();  
  
 Map map = new HashMap();  
 map.put("code",-1);  
 map.put("msg","你好，请稍后再试");  
 return map;  
}

## 4.6 Hystrix降级策略和调整

官方网址：<https://github.com/Netflix/Hystrix/wiki/Configuration#execution.isolation.strategy>

### 4.6.1 Hystrix配置类HystrixCommandProperties

该类中有很多的Hystrix的默认配置，配置其属性时可以查看该类。

### 4.6.2 禁用超时，一般不禁用

*#hystrix 超时时间关闭 默认开启；一般不应该禁用***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **timeout**:  
 **enabled**: false

### 4.6.3 修改超时时间

*#修改Hystrix超时时间为2000毫秒  
#execution.isolation.thread.timeoutInMilliseconds***hystrix**:  
 **command**:  
 **default**:  
 **execution**:  
 **isolation**:  
 **thread**:  
 **timeoutInMilliseconds**: 2000

### 4.6.4 隔离策略 execution.isolation.strategy

THREAD 线程池隔离 （默认）

SEMAPHORE 信号量

信号量适用于接口并发量高的情况，如每秒数千次调用的情况，导致的线程开销过高，通常只适用于非网络调用，执行速度快

execution.isolation.semaphore.maxConcurrentRequests

隔离策略为 信号量的时候，如果达到最大并发数时，后续请求会被拒绝，默认是10

## 4.7 Hystrix 整合Dashboard监控仪表盘

#### 4.7.5.1 加入依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

#### 4.7.5.2 启动类修改

1）添加注解@EnableHystrixDashboard

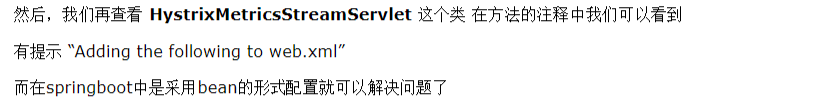
2）添加Bean -> ServletRegistrationBean（这个2.1.1不需要配置）

springboot 版本如果是2.0则需要添加 ServletRegistrationBean 因为springboot的默认路径不是 "/hystrix.stream"，只要在自己的项目里配置上下面的servlet就可以了

@SpringBootApplication  
@EnableFeignClients  
@EnableCircuitBreaker  
@EnableHystrixDashboard  
public class OrderServiceApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(OrderServiceApplication.class, args);  
 }  
  
 @Bean  
 @LoadBalanced //注解负载均衡  
 public RestTemplate restTemplate(){  
 return new RestTemplate();  
 }  
 //添加bean  
 @Bean  
 public ServletRegistrationBean getServlet() {  
 HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();  
 ServletRegistrationBean registrationBean = new ServletRegistrationBean(streamServlet);  
 registrationBean.setLoadOnStartup(1);  
 registrationBean.addUrlMappings("/hystrix.stream");  
 registrationBean.setName("HystrixMetricsStreamServlet");  
 return registrationBean;  
 }  
}

为什么要加这个bean？







#### 4.7.5.3 配置文件配置（这个必须配置）

**management**:  
 **endpoints**:  
 **web**:  
 **exposure**:  
 **include**: "\*"

访问入口

第一步： <http://localhost:8781/hystrix>

第二步： Hystrix Dashboard输入： <http://localhost:8781/actuator/hystrix.stream>

参考资料

默认开启监控配置

<https://docs.spring.io/spring-boot/docs/current-SNAPSHOT/reference/htmlsingle/#boot-features-security-actuator>

配置文件类：

spring-configuration-metadata.json

## 4.8 断路器监控仪表参数讲解和模拟熔断

简介：讲解 断路器监控仪表盘参数和模拟熔断

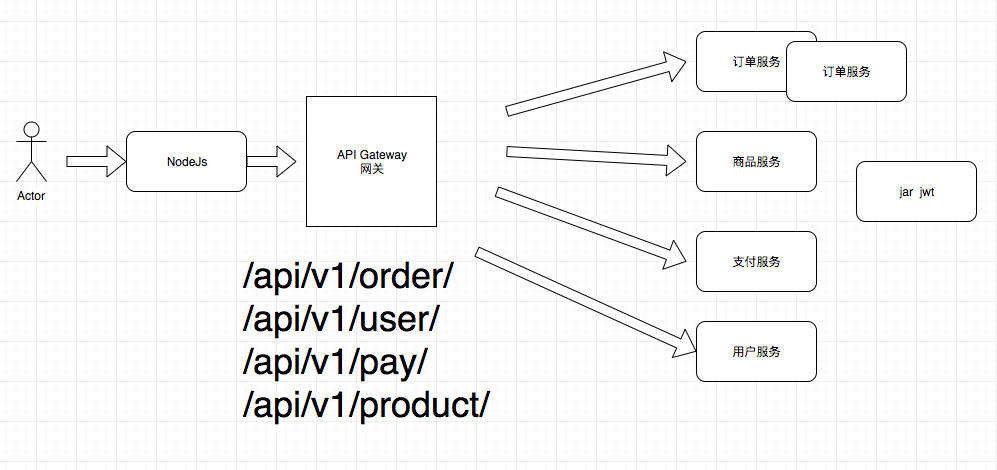
1、sse server-send-event推送到前端

资料：<https://github.com/Netflix/Hystrix/wiki/Dashboard>

# 5 Zuul : 为服务网关

## 5.1 Gateway网关介绍

### 5.1.1 什么是网关？



API Gateway，是服务系统对外的唯一入口，介于客户端和服务器之间的中间层，用来处理非业务功能，提供请求的路由、限流、鉴权、缓存等功能。

1）统一接入

智能路由

负载均衡、容灾处理

日志埋点（类似Nginx日志）

AB测试、灰度测试

2）流量监控

限流处理

服务降级

3）安全防护

鉴权处理

监控

机器网络隔离

### 5.1.2 主流网关？

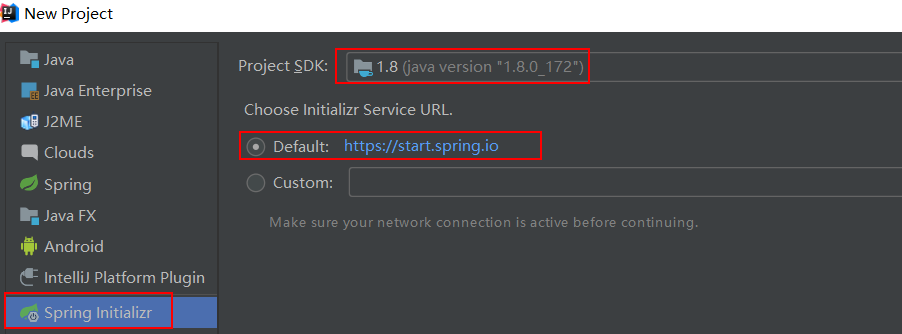
zuul:是Netflix开源的微服务网关，和Eureka、Ribbon、Hystrix等组件配置使用，Zuul 2.0 比1.0 性能提高很多。

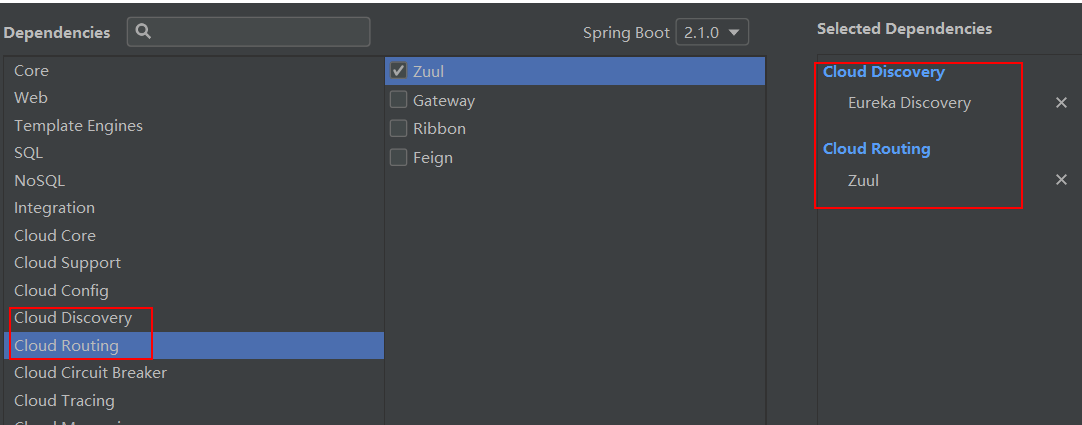
kong：由Mashape公司开源的，基于Nginx的API Gateway

Nginx+Lua：是一个高性能的HTTP和反向代理服务器，lua是脚本语言，让Nginx执行lua脚本，并且高并发、非阻塞的处理各种请求。

## 5.2 Zuul网关基本使用

### 5.2.1 项目创建：





会自动添加依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
</dependency>

### 5.2.2 配置文件

**#端口设置**

**server**:  
 **port**: 9000  
#网关服务名称  
**spring**:  
 **application**:  
 **name**: gateway-server  
#注册中心地址  
**eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://localhost:8761/eureka/

### 5.2.3 添加注解

@SpringBootApplication  
@EnableZuulProxy //开启网关策略  
public class GatewayServerApplication {

### 5.2.4 自定义路由-服务隔离

**zuul**:  
 **routes**:  
 **order-server**: /gateway/\*\* 自定义路由(其中不能包含zuul关键字)  
 *#ignored-services: product-server* 隔离指定服务  
 *#ignored-patterns: /\*-server/\*\**  隔离匹配路径的服务接口（请求）

### 5.2.5 访问规则

默认访问规则

http://ip:port/service-id/\*\*

http://ip:端口/服务id/路径

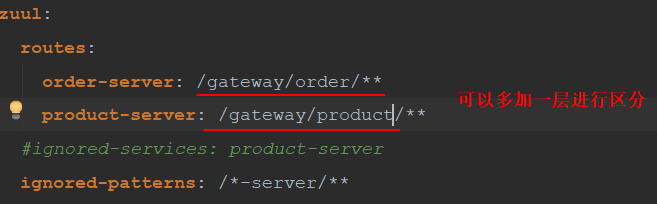
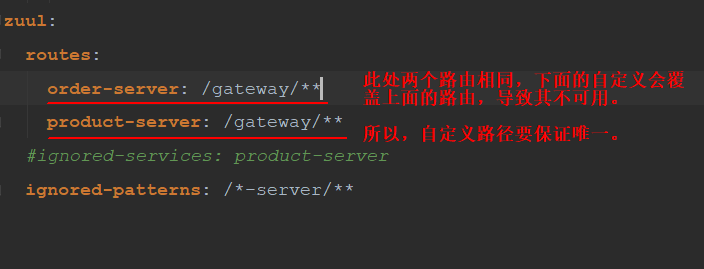
例子：默认 /order-service/order/save2?uid=2&pid=1

自定义 / gateway /order/save?ud=2&pid=1

## 5.3 Zuul常用问题分析和网关过滤器原理分析

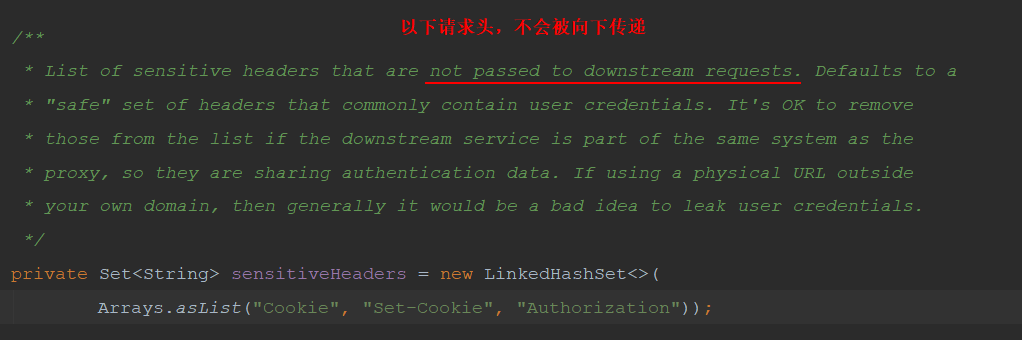
### 5.3.1 路由名称定义问题

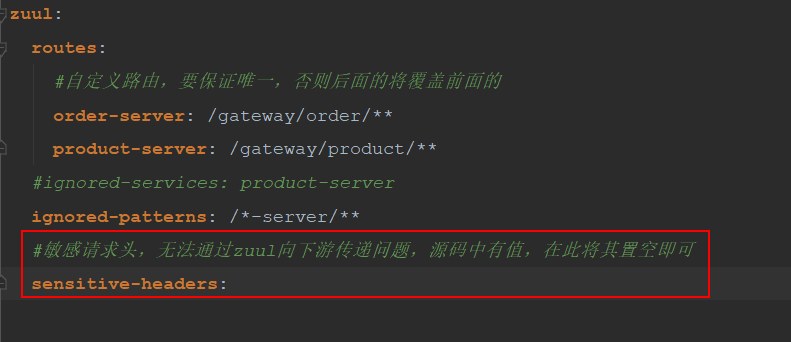
路由映射重复覆盖问题



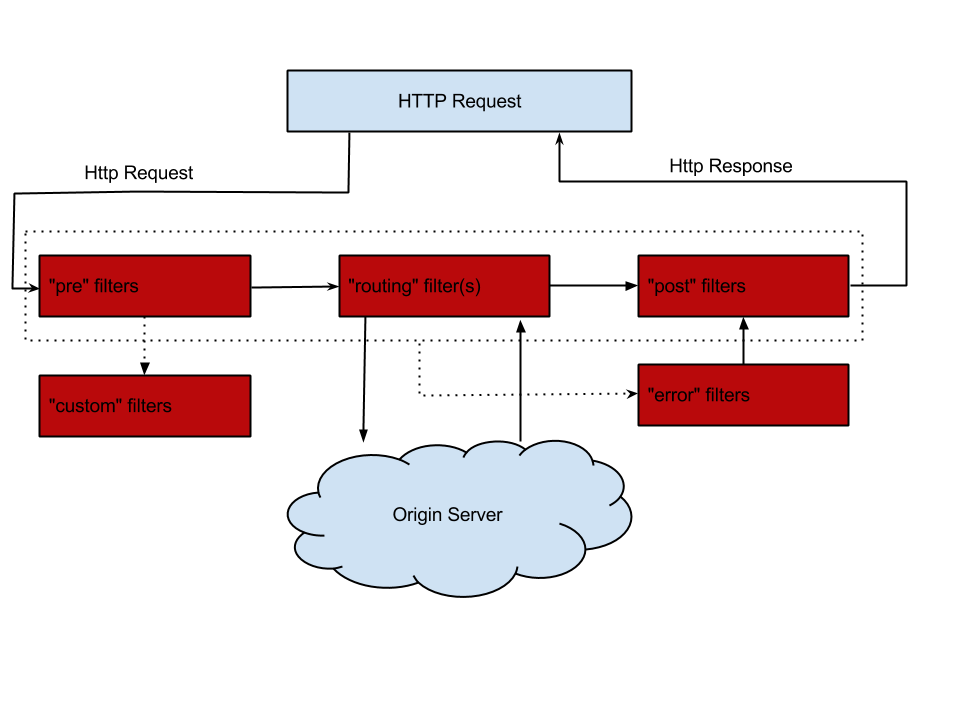
### 5.3.2 Http请求头过滤问题

Cookie Set-Cookie Authorization





### 5.3.3 过滤器执行顺序问题 ，过滤器的order值越小，越先执行



### 5.3.4 共享RequestContext，上下文对象

## 5.4 自定义Zuul过滤器

### 5.4.1 自定义Zuul过滤器要求

1）继承抽象类ZuulFilter

2）可用的bean，交给spring管理。@Component

### 5.4.2 示例代码 权限验证

@Component  
public class OrderZullFilter extends ZuulFilter {  
 */\*\*  
 \* 过滤器的类型，PRE\_TYPE POST\_TYPE  
 \* 本例是前置过滤器  
 \** ***@return*** *\*/* @Override  
 public String filterType() {  
 return FilterConstants.*PRE\_TYPE*;  
 }  
  
 */\*\*  
 \* filter的执行属性，order越小，越早执行  
 \* FilterContants中有-3 到10  
 \* 本例就定义为4  
 \** ***@return*** *\*/* @Override  
 public int filterOrder() {  
 return 4;  
 }  
  
 */\*\*  
 \* 判断是否拦截请求，false不拦截，true拦截  
 \* 本例对save请求进行拦截 /gateway/order/order/save2 如果与此路径匹配则拦截  
 \** ***@return*** *\*/* @Override  
 public boolean shouldFilter() {  
 //获取共享上下文RequestContext  
 RequestContext requestContext = RequestContext.*getCurrentContext*();  
 //获取HttpServletRequest  
 HttpServletRequest request = requestContext.getRequest();  
 //获取请求uri  
 String uri = request.getRequestURI();  
 // /gateway/order/order/save2 如果与此路径匹配则拦截  
 if ("/gateway/order/order/save2".equals(uri)) {  
 return true;  
 }  
 return false;  
 }  
  
 */\*\*  
 \* 拦截后的逻辑操作  
 \* 如果请求头或url参数中的token不为空，则放行，否则拦截  
 \** ***@return*** *\** ***@throws*** *ZuulException  
 \*/* @Override  
 public Object run() throws ZuulException {  
 //获取共享上下文RequestContext  
 RequestContext requestContext = RequestContext.*getCurrentContext*();  
 //获取HttpServletRequest  
 HttpServletRequest request = requestContext.getRequest();  
 //获取请求中的token  
 String token = request.getHeader("token");  
 //如果token为空，则先去url中查找是否有参数token  
 if(StringUtils.*isBlank*(token)){  
 token = request.getParameter("token");  
 }  
 /\*  
 最终判断token是否为空  
 1 为空则拦截  
 2 不空则放行  
 \*/  
 if(StringUtils.*isBlank*(token)){  
 //如果为空则拦截不放行  
 requestContext.setSendZuulResponse(false);  
 //返回状态码  
 requestContext.setResponseStatusCode(HttpStatus.*UNAUTHORIZED*.value());  
 System.*out*.println("拦截中================");  
 }  
 return null;  
 }  
}

### 5.4.3 JWT （json web token） 服务验证

#### 5.4.3.1 依赖引入

<dependency>  
 <groupId>com.auth0</groupId>  
 <artifactId>java-jwt</artifactId>  
 <version>3.4.1</version>  
</dependency>

#### 5.4.3.2 JWT基本使用（生成token 解码token）

1）验证登录生成token

@RequestMapping("login")  
public String login(@RequestParam("username") String username, @RequestParam("password") String password){  
 System.*out*.println("fangwenle ===============");  
 System.*out*.println(username);  
 String token = JWT.*create*().withAudience(username).sign(Algorithm.*HMAC256*(password));  
 return token;  
}  
  
 2）获取token进行解码验证

@RequestMapping("test")  
public String test(HttpServletRequest request){  
 String token = request.getHeader("token");  
 if(StringUtils.*isBlank*(token)){  
 return "没有权限，请登录";  
 }  
 String username = JWT.*decode*(token).getAudience().get(0);  
 return username;  
}

## 5.5高并发情况下接口限流特技

### 5.5.1基本网关限流使用

5.5.1.1 实现ZuulFilter

限流的本质也是过滤器的使用。只是在其中增加了一个“令牌操作”。

5.5.1.2 示例代码

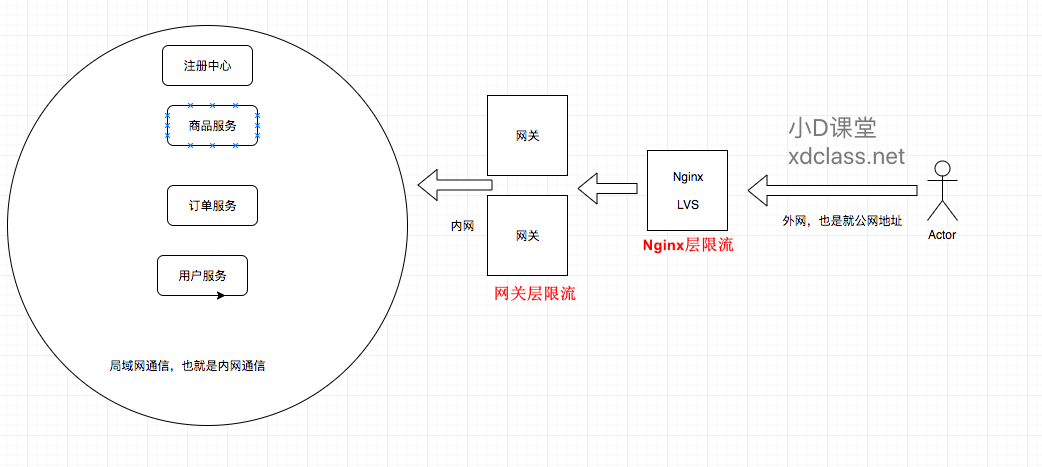
@Component  
public class OrderRateLimitFilter extends ZuulFilter {

//创建令牌桶（令牌集合）  
 private static final RateLimiter *RATE\_LIMIT* = RateLimiter.*create*(10);  
  
 @Override  
 public String filterType() {  
 return FilterConstants.*PRE\_TYPE*;  
 }  
  
 @Override  
 public int filterOrder() {  
 return -4;  
 }  
  
 @Override  
 public boolean shouldFilter() {  
  
 RequestContext requestContext = RequestContext.*getCurrentContext*();  
 HttpServletRequest request = requestContext.getRequest();  
 if ("/gateway/order/order/save2".equals(request.getRequestURI())) {  
 return true;  
 }  
 return false;  
 }  
  
 @Override  
 public Object run() throws ZuulException {  
 //如果为没有令牌，也就是返回false，则拦截  
 if (!*RATE\_LIMIT*.tryAcquire()){  
 //获取共享上下文RequestContext  
 RequestContext requestContext = RequestContext.*getCurrentContext*();  
 //拦截  
 requestContext.setSendZuulResponse(false);  
 //返回状态码  
 requestContext.setResponseStatusCode(HttpStatus.*TOO\_MANY\_REQUESTS*.value());  
 System.*out*.println("拦截中================");  
 }  
 return null;  
 }  
}

### 5.5.2 两层限流

1）nginx层限流

2）网关层限流



### 5.5.3 谷歌guava框架介绍，网关限流使用（待学习）

## 5.6 Zuul微服务网关集群搭建

简介：微服务网关Zull集群搭建

5.6.1 nginx+lvs+keepalive （待学习）

<https://www.cnblogs.com/liuyisai/p/5990645.html>

# 6 Sleuth+ZipKin : 分布式链路追踪系统

## 6.1 微服务下的链路追踪讲解和重要性

简介：讲解什么是分布式链路追踪系统，及使用好处

全链路性能监控 从整体维度到局部维度展示各项指标，将跨应用的所有调用链性能信息集中展现，可方便度量整体和局部性能，并且方便找到故障产生的源头，生产上可极大缩短故障排除时间。

我们除了性能指标之外，我们也需要链式追踪系统拥有以下功能：

1）请求链路追踪，故障快速定位：可以通过调用链结合业务日志快速定位错误信息。

2）可视化： 各个阶段耗时，进行性能分析。

3）依赖优化：各个调用环节的可用性、梳理服务依赖关系以及优化。

4）数据分析，优化链路：可以得到用户的行为路径，汇总分析应用在很多业务场景。

## 6.2 Sleuth 链路追踪组件实战

### 6.2.1 官方文档

http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#sleuth-adding-project

### 6.2.2什么是Sleuth

一个组件，专门用于记录链路数据的开源组件

[order-service,96f95a0dd81fe3ab,852ef4cfcdecabf3,false]

1）第一个值，spring.application.name的值

2）第二个值，96f95a0dd81fe3ab ，sleuth生成的一个ID，叫Trace ID，用来标识一条请求链路，一条请求链路中包含一个Trace ID，多个Span ID

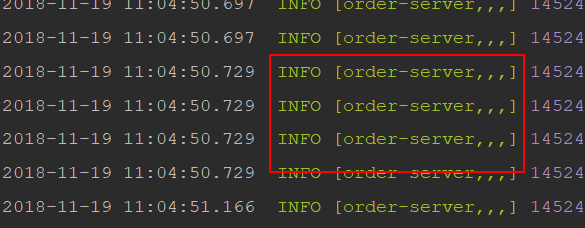
3）第三个值，852ef4cfcdecabf3、spanid 基本的工作单元，获取元数据，如发送一个http

4）第四个值：false，是否要将该信息输出到zipkin服务中来收集和展示。

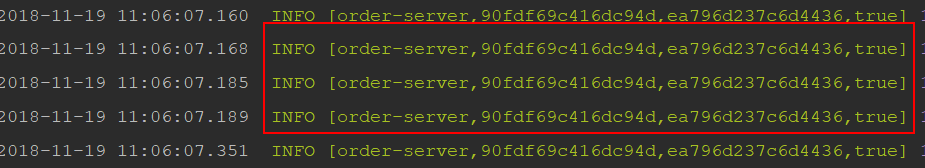
### 6.2.3 添加依赖

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-zipkin</artifactId>  
</dependency>

添加依赖后启动项目即可看到控制台日志效果：代表已生效

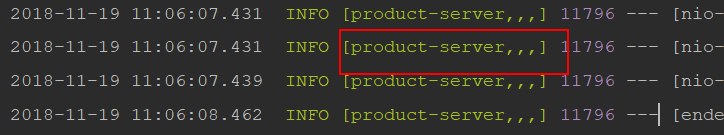


发出一次请求进行访问：这就是Sleuth的追踪记录



再进行一次访问：发现问题->

不再有追踪日志输出，同时发现该服务的下游服务日志输出如下：数据不完整



问题解决：6.2.4

### 6.2.4 Sleuth链路追踪组件常见问题说明

问题描述：6.2.3 ，主要是因为日志级别导致的。

解决方式：开启日志记录sflog4j，并使用级别INFO

1）在service中添加属性：Logger

@Service  
public class OrderServiceImpl implements OrderService{  
 private final Logger logger = LoggerFactory.*getLogger*(getClass());

2）在访问的方法中添加一句信息：

public String save2(Integer userId, Integer productId) {  
 logger.info("order - server - save2 ");

## 6.4 Zipkin 可视化链路追踪系统部署

### 6.4.1 什么是zipkin

官网：https://zipkin.io/

大规模分布式系统的APM工具（Application Performance Management）,基于Google Dapper的基础实现，和sleuth结合可以提供可视化web界面分析调用链路耗时情况

#### 6.4.1.1 zipkin组成：

Collector、Storage、Restful API（search）、Web UI组成

收集器 、存储、 搜索、 Web UI界面

There are 4 components that make up Zipkin:

* collector
* storage
* search
* web UI

##### Zipkin Collector

Once the trace data arrives at the Zipkin collector daemon, it is validated, stored, and indexed for lookups by the Zipkin collector.

##### Storage

Zipkin was initially built to store data on Cassandra since Cassandra is scalable, has a flexible schema, and is heavily used within Twitter. However, we made this component pluggable. In addition to Cassandra, we natively support ElasticSearch and MySQL. Other back-ends might be offered as third party extensions.

##### Zipkin Query Service

Once the data is stored and indexed, we need a way to extract it. The query daemon provides a simple JSON API for finding and retrieving traces. The primary consumer of this API is the Web UI.

##### Web UI

We created a GUI that presents a nice interface for viewing traces. The web UI provides a method for viewing traces based on service, time, and annotations. Note: there is no built-in authentication in the UI!

#### 6.4.1.2运行方式：

##### Docker

The [Docker Zipkin](https://github.com/openzipkin/docker-zipkin) project is able to build docker images, provide scripts and a [docker-compose.yml](https://github.com/openzipkin/docker-zipkin/blob/master/docker-compose.yml) for launching pre-built images. The quickest start is to run the latest image directly:

docker run -d -p 9411:9411 openzipkin/zipkin

##### Java

If you have Java 8 or higher installed, the quickest way to get started is to fetch the [latest release](https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec) as a self-contained executable jar:

curl -sSL https://zipkin.io/quickstart.sh | bash -s

java -jar zipkin.jar

##### Running from Source

Zipkin can be run from source if you are developing new features. To achieve this, you’ll need to get [Zipkin’s source](https://github.com/openzipkin/zipkin) and build it.

# get the latest source

git clone https://github.com/openzipkin/zipkin

cd zipkin

# Build the server and also make its dependencies

./mvnw -DskipTests --also-make -pl zipkin-server clean install

# Run the server

java -jar ./zipkin-server/target/zipkin-server-\*exec.jar

### 6.4.2同类产品

鹰眼（EagleEye）

CAT

twitter开源zipkin，结合sleuth

Pinpoint，运用JavaAgent字节码增强技术

StackDriver Trace (Google)

### 6.4.3 开始使用

https://github.com/openzipkin/zipkin

运行zipkin ：<https://zipkin.io/pages/quickstart.html>

#### 6.4.3.1 运行方式

##### Docker

The [Docker Zipkin](https://github.com/openzipkin/docker-zipkin) project is able to build docker images, provide scripts and a [docker-compose.yml](https://github.com/openzipkin/docker-zipkin/blob/master/docker-compose.yml) for launching pre-built images. The quickest start is to run the latest image directly:

docker run -d -p 9411:9411 openzipkin/zipkin

##### Java

If you have Java 8 or higher installed, the quickest way to get started is to fetch the [latest release](https://search.maven.org/remote_content?g=io.zipkin.java&a=zipkin-server&v=LATEST&c=exec) as a self-contained executable jar:

curl -sSL https://zipkin.io/quickstart.sh | bash -s

java -jar zipkin.jar

##### Running from Source

Zipkin can be run from source if you are developing new features. To achieve this, you’ll need to get [Zipkin’s source](https://github.com/openzipkin/zipkin) and build it.

# get the latest source

git clone https://github.com/openzipkin/zipkin

cd zipkin

# Build the server and also make its dependencies

./mvnw -DskipTests --also-make -pl zipkin-server clean install

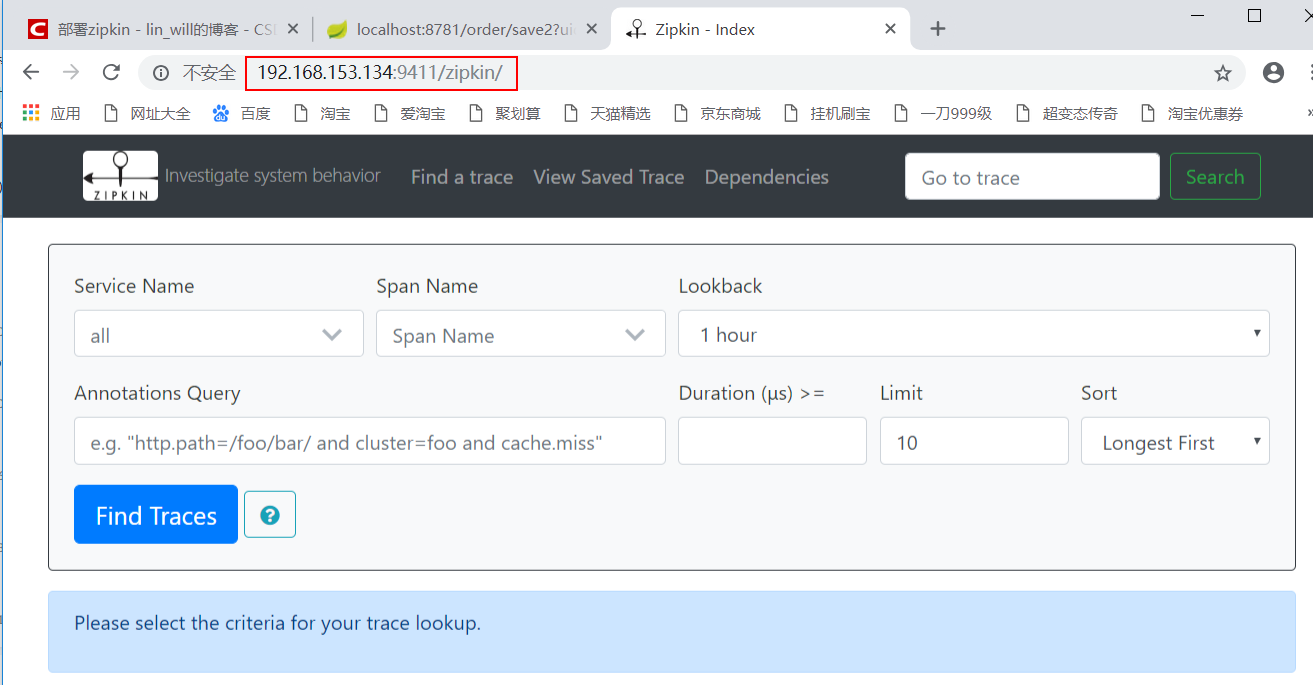
# Run the server

java -jar ./zipkin-server/target/zipkin-server-\*exec.jar

#### 6.4.3.2使用Docker运行Zipkin：

docker run -d -p 9411:9411 openzipkin/zipkin

启动成后访问：结果如下



### 6.4.4 知识拓展：OpenTracing

OpenTracing 已进入 CNCF，正在为全球的分布式追踪，提供统一的概念和数据标准。 通过提供平台无关、厂商无关的 API，使得开发人员能够方便的添加（或更换）追踪系统的实现。

推荐阅读：

http://blog.daocloud.io/cncf-3/

https://www.zhihu.com/question/27994350

https://yq.aliyun.com/articles/514488?utm\_content=m\_43347

## 6.5 Zipkin+Sleuth链路追踪组件实战

### 6.5.1文档

http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#\_sleuth\_with\_zipkin\_via\_http

sleuth收集跟踪信息通过http请求发送给zipkin server，zipkinserver进行跟踪信息的存储以及提供Rest API即可，Zipkin UI调用其API接口进行数据展示

默认存储是内存，可也用mysql、或者elasticsearch等存储

### 6.5.2 加入依赖

<dependency>

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

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

</dependency>

里面包含 spring-cloud-starter-sleuth、spring-cloud-sleuth-zipkin

### 6.5.3文档说明：

http://cloud.spring.io/spring-cloud-static/Finchley.SR1/single/spring-cloud.html#\_features\_2

### 6.5.4 配置zipkin.base-url

**spring**:

*#zipkin 服务地址***zipkin:  
 base-url:** http://192.168.153.134:9411/

#此处一定要加上发送类型：web /rabbit/kafaka 不然很容易出现问题：无法展示数据  
 **sender:  
 type:** *web  
#sleuth抓取概率***sleuth:  
 sampler:  
 probability:** 1

常见问题：

服务端发送数据正常，但是在web界面没有展示数据。

#此处一定要加上发送类型：web /rabbit/kafaka 不然很容易出现问题：无法展示数据  
 **sender:  
 type:** *web*

### 6.5.5 配置采样百分比spring.sleuth.sampler

**spring**:

*#配置采样百分比* **sleuth:  
 sampler:  
 probability: 1**

推荐资料：

<https://blog.csdn.net/jrn1012/article/details/77837710>

# 7 config-server ：分布式配置中心

传统的配置文件有什么缺点：

如果修改了配置文件，就需要重新打包发布，并重启项目；另外，每个环境变更配置文件比较复杂。

分布式配置中心优点：

将配置文件信息注册到配置中心平台上，可以直接使用分布式配置中心实时修改更新配置文件，既能统一管理，又不需要重新打包发布或重启项目。

什么是分布式配置中心？

项目中配置环境比较繁杂，而且不同环境的不同配置修改相对频繁，每次发布都需要对应的修改配置，如果配置出现错误，需要重新打包发布，时间成本较高，因此需要做一个统一的分布式注册中心，能够做到自动更新配置文件信息，解决以上问题。

分布式配置中心的职责？

解决分布式配置文件管理。

常用的分布式配置中心框架？

Disconf(依赖于zookeeper)

zookeeper(使用事件通知，保证配置文件信息实时更新)

diamond

Apollo

redis

## 7.1 Config-server

### 7.1．1 配置中心介绍

好处：统一管理配置, 快速切换各个环境的配置

相关产品：

百度的disconf

地址:https://github.com/knightliao/disconf

阿里的diamand

地址：https://github.com/takeseem/diamond

springcloud的configs-server:

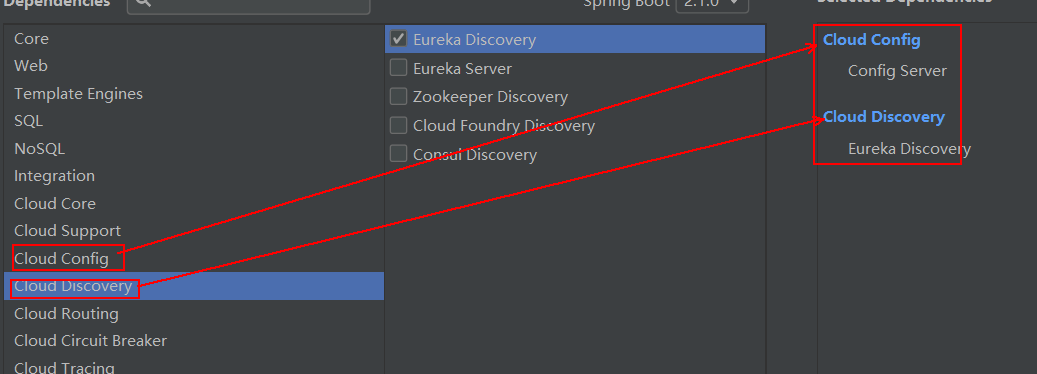
地址：<http://cloud.spring.io/spring-cloud-config/>

携程的阿波罗

推荐干货文章：<http://jm.taobao.org/2016/09/28/an-article-about-config-center/>

### 7.2 config-server 配置中心

#### 7.2.1 配置中心服务搭建



会自动添加依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-server</artifactId>  
</dependency>

#### 7.2.2 启动类注解添加@EnabledConfigServer

@SpringBootApplication

@EnableConfigServer  
public class ConfigServerApplication {

### 7.3 Git服务器结合Config搭建分布式配置中心

#### 7.3.1 config-server 默认使用git存储配置

使用git服务器，可以自己搭建gitlab服务器

或者使用github、开源中国git、阿里云git

#### 7.3.2 config-server配置文件配置

服务名称+git地址（uri、用户名、密码、超时时间、分支）

**uri**: <https://gitee.com/cnhtb/config_server>

此处不要带后缀：.git

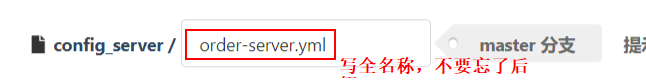
*#服务开放端口号***server**:  
 **port**: 9100  
*#注册中心地址***eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://localhost:8761/eureka/

*#服务名称***spring**:  
 **application**:  
 **name**: config-server  
 *#config-server git配置* **cloud**:  
 **config**:  
 **server**:  
 **git**:  
 *#地址+用户+密码* **uri**: https://gitee.com/cnhtb/config\_server  
 **username**: 1795683509@qq.com  
 **password**: cn.htb123  
 *#超时时间* **timeout**: 5  
 *#分支  
 #default-label: master*

#### 7.3.3 访问配置中心方式

##### 7.3.3.1 git中创建配置文件：





##### 7.3.3.2 git中配置文件访问方式：

多种访问路径，可以通过启动日志去查看

例子 http://localhost:9100/product-server.yml

/{name}-{profiles}.properties

/{name}-{profiles}.yml

/{name}-{profiles}.json

/{label}/{name}-{profiles}.yml

name 服务器名称 :product-server

profile 环境名称，开发、测试、生产:default\test\dev

lable 仓库分支、默认master分支 :master\test

### 7.4 config-server的项目使用

官方文档：http://cloud.spring.io/spring-cloud-config/single/spring-cloud-config.html#\_spring\_cloud\_config\_client

#### 7.4.1 依赖加入

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-config-client</artifactId>  
</dependency>

#### 7.4.2 修改对应服务的配置文件

##### 7.4.2.1 配置文件名称修改

application.yml -》bootstrap.yml

##### 7.4.2.2配置文件内容：

服务名称+注册中心地址+配置中心（开启配置中心\ID\label(分支)\profile（环境））

注释掉的部分，都包含在git仓库配置文件中：

*#server:  
# port: 8771***eureka**:  
 **client**:  
 **service-url**:  
 **defaultZone**: http://localhost:8761/eureka/  
**spring**:  
 **application**:  
 **name**: product-server  
*# zipkin:  
# base-url: http://192.168.153.134:9411/  
 #链路追踪配置  
# sleuth:  
# sampler:  
# probability: 1* **cloud**:  
 **config**:  
 **discovery**:  
 *#开启config-server服务* **enabled**: **true** *#设置config-server服务的id* **service-id**: config-server  
 *#加载配置文件的环境* **profile**: test  
 *#加载配置配件所在节点* **label**: master

## 7.2 Apollo使用

### 介绍：

介绍：<https://github.com/ctripcorp/apollo/wiki/Apollo%E9%85%8D%E7%BD%AE%E4%B8%AD%E5%BF%83%E4%BB%8B%E7%BB%8D>

简介：

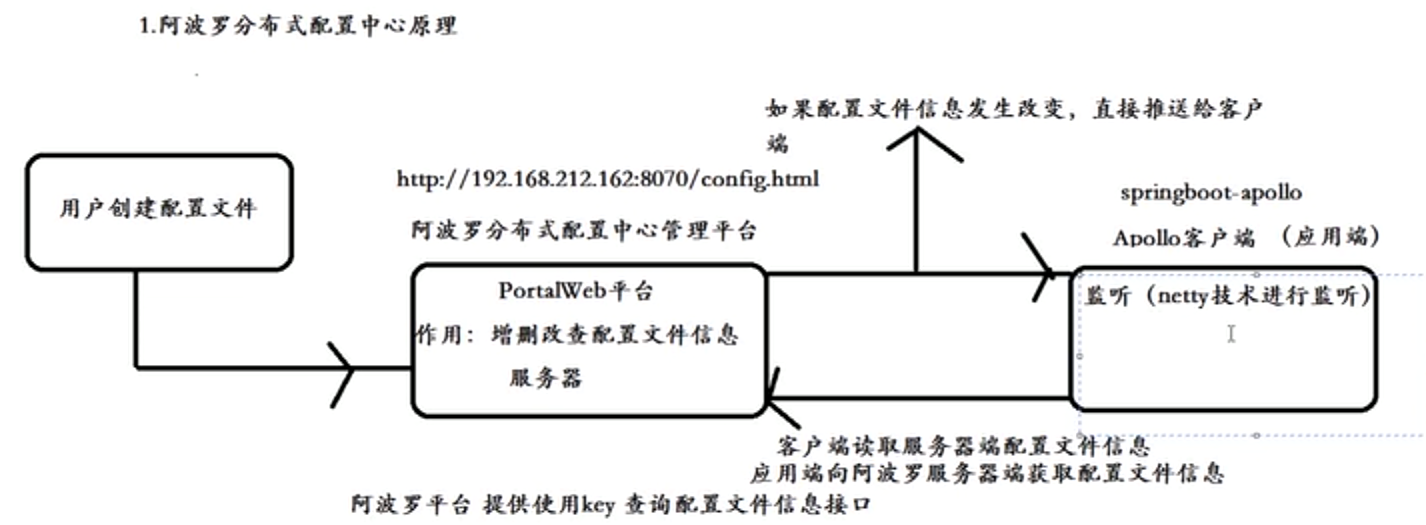
Apollo（阿波罗）是携程框架部门研发的开源配置管理中心，能够集中化管理应用不同环境、不同集群的配置，配置修改后能够实时推送到应用端，并且具备规范的权限、流程治理等特性。适用于微服务配置管理场景。

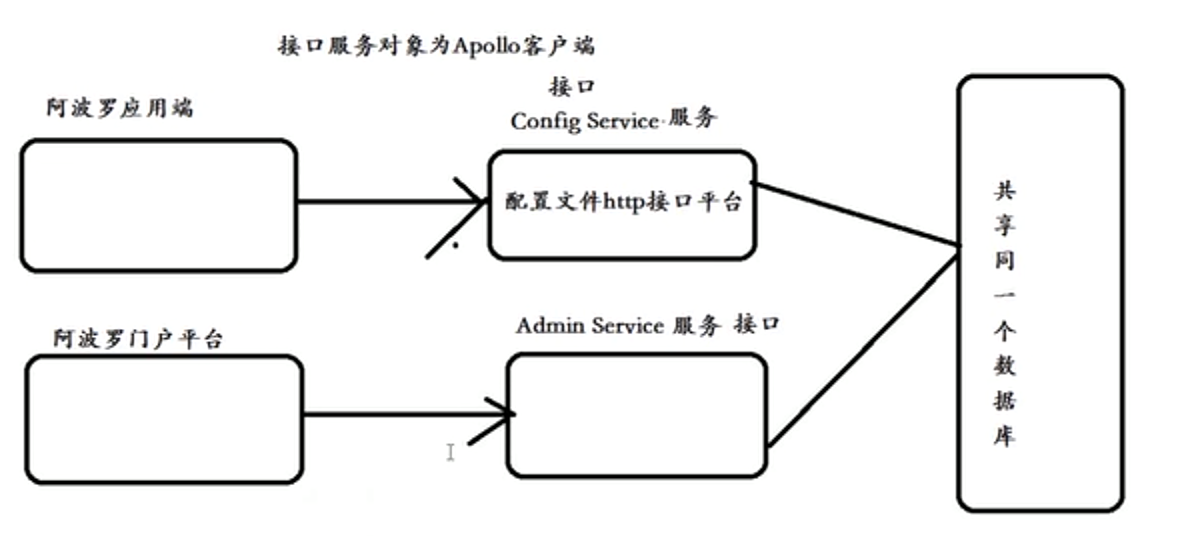
服务端基于SpringBoot和SpringCLoud开发，打包后可以直接运行，不需要额外安装Tomcat等应用容器。

Java客户端不依赖任务框架，能够运行于所有java运行环境，同时对Spirng/SpringBoot环境有较好的支持。

特点：

用户在Apollo修改完配置并发布后，客户端能实时（1秒）接收到最最新的配置，并通知到应用程序。





### 快速安装

快速安装不支持多环境配置。分布式配置的才可以支持多环境

#### 7.2.1 Apollo安装

安装包获取：<https://github.com/ctripcorp/apollo/wiki/Quick-Start>

Apollo-build-scripts-master 整个阿波罗环境包，使用一个命令启动整个阿波罗三个服务：./demo.sh start

1 环境要求：

虚拟机内容在2g以上

JDK1.8环境

2 搭建数据库

版本在5.7以上，因为阿波罗数据结构会有两个时间戳类型字段，低版本数据库不支持。

创建两个数据库：（Apollo-build-scripts-master包内有sql）

apolloconfigdb：存放配置文件信息

apolloportaldb：门户网站

3 创建文件夹：mkdir -p /usr/local/Apollo

解压压缩包：unzip Apollo-build-scripts-master.zip （不能使用unzip,就安装yum install unzip）

4 修改文件damo.sh 如下



5 启动项目

./damo.sh start

启动较慢，成功后会出现configserver 地址和portalservice地址

6 访问门户portal

<http://192.168.153.164:8070/config.html?#/appid=apollo_test>

用户名：apollo 密码：admin

7 创建项目：

记住自己的appid(测试使用)



8 测试是否可用：（使用apollo自带的客户端）

由于使用了新的项目，所以客户端需要修改appId信息。

编辑client/META-INF/app.properties，修改app.id为你新创建的app id。

app.id=你的appId (就是apollo\_test)

运行./demo.sh client启动Demo客户端即可。

输入timeout查看对应的值是否为变化

> timeout

> [SimpleApolloConfigDemo] Loading key : timeout with value: 200

#### 7.2.2 SpringBoot整合



添加依赖：

<dependency>  
 <groupId>com.ctrip.framework.apollo</groupId>  
 <artifactId>apollo-client</artifactId>  
 <version>1.1.1</version>  
</dependency>

配置文件：application.yml

**server:  
 port:** 9000  
**spring:  
 application:  
 name:** apollo\_test  
**app:  
 id:** apollo\_test  
*# set apollo meta server address, adjust to actual address if necessary***apollo:  
 meta:** http://192.168.153.164:8080

启动类添加注解：

@EnableApolloConfig

测试代码：配置中心修改该数据test，访问查看数据是否变化

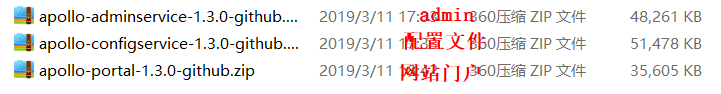
package cn.qianzui.config\_server2;  
  
import org.springframework.beans.factory.annotation.Value;  
import org.springframework.web.bind.annotation.RequestMapping;  
import org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class TEst {  
  
 @Value("${test:eeee}")  
 private String test;  
  
 @RequestMapping("/test")  
 public String getTest(){  
 return test;  
 }  
}

### 多环境安装

https://github.com/ctripcorp/apollo/wiki/%E5%88%86%E5%B8%83%E5%BC%8F%E9%83%A8%E7%BD%B2%E6%8C%87%E5%8D%97

#### 1 获取安装包：三个

<https://github.com/ctripcorp/apollo/releases>



#### 2 安装MySQL数据库 5.7以上版本

插入数据库：

apolloconfigdb：存放配置文件信息

apolloportaldb：门户网站

#### 3 各服务配置调整

**调整ApolloPortalDB配置**

配置项统一存储在ApolloPortalDB.ServerConfig表中，也可以通过管理员工具 - 系统参数页面进行配置，无特殊说明则修改完一分钟实时生效。

apollo.portal.envs - 可支持的环境列表

默认值是dev，如果portal需要管理多个环境的话，以逗号分隔即可（大小写不敏感），如：DEV,FAT,UAT,PRO 修改完需要重启生效。

**调整ApolloConfigDB配置**

配置项统一存储在ApolloConfigDB.ServerConfig表中，需要注意每个环境的ApolloConfigDB.ServerConfig都需要单独配置，修改完一分钟实时生效。

eureka.service.url - Eureka服务Url

不管是apollo-configservice还是apollo-adminservice都需要向eureka服务注册，所以需要配置eureka服务地址。 按照目前的实现，apollo-configservice本身就是一个eureka服务，所以只需要填入apollo-configservice的地址即可，如有多个，用逗号分隔（注意不要忘了/eureka/后缀）。

需要注意的是每个环境只填入自己环境的eureka服务地址，比如FAT的apollo-configservice是1.1.1.1:8080和2.2.2.2:8080，UAT的apollo-configservice是3.3.3.3:8080和4.4.4.4:8080，PRO的apollo-configservice是5.5.5.5:8080和6.6.6.6:8080，那么：

在FAT环境的

ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://1.1.1.1:8080/eureka/,http://2.2.2.2:8080/eureka/

在UAT环境的

ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://3.3.3.3:8080/eureka/,http://4.4.4.4:8080/eureka/

在PRO环境的

ApolloConfigDB.ServerConfig表中设置eureka.service.url为：

http://5.5.5.5:8080/eureka/,http://6.6.6.6:8080/eureka/

注1：这里需要填写本环境中全部的eureka服务地址，因为eureka需要互相复制注册信息

#### 4配置数据库

##### 配置apollo-configservice的数据库连接信息

1. 解压apollo-configservice-x.x.x-github.zip
2. 用程序员专用编辑器（如vim，notepad++，sublime等）打开config目录下的application-github.properties文件
3. 填写正确的ApolloConfigDB数据库连接串信息，注意用户名和密码后面不要有空格!
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloConfigDB?useSSL=false&characterEncoding=utf8

spring.datasource.username = someuser

spring.datasource.password = somepwd

注：由于ApolloConfigDB在每个环境都有部署，所以对不同的环境config-service需要配置对应环境的数据库参数

##### 配置apollo-adminservice的数据库连接信息

1. 解压apollo-adminservice-x.x.x-github.zip
2. 用程序员专用编辑器（如vim，notepad++，sublime等）打开config目录下的application-github.properties文件
3. 填写正确的ApolloConfigDB数据库连接串信息，注意用户名和密码后面不要有空格!
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloConfigDB?useSSL=false&characterEncoding=utf8

spring.datasource.username = someuser

spring.datasource.password = somepwd

注：由于ApolloConfigDB在每个环境都有部署，所以对不同的环境admin-service需要配置对应环境的数据库参数

##### 配置apollo-portal的数据库连接信息

1. 解压apollo-portal-x.x.x-github.zip
2. 用程序员专用编辑器（如vim，notepad++，sublime等）打开config目录下的application-github.properties文件
3. 填写正确的ApolloPortalDB数据库连接串信息，注意用户名和密码后面不要有空格!
4. 修改完的效果如下：

# DataSource

spring.datasource.url = jdbc:mysql://localhost:3306/ApolloPortalDB?useSSL=false&characterEncoding=utf8

spring.datasource.username = someuser

spring.datasource.password = somepwd

#### 5配置apollo-portal的meta service信息 ：多环境的关键

Apollo Portal需要在不同的环境访问不同的meta service(apollo-configservice)地址，所以我们需要在配置中提供这些信息。默认情况下，meta service和config service是部署在同一个JVM进程，所以meta service的地址就是config service的地址。

使用程序员专用编辑器（如vim，notepad++，sublime等）打开apollo-portal-x.x.x-github.zip中config目录下的apollo-env.properties文件。

假设DEV的apollo-configservice未绑定域名，地址是1.1.1.1:8080，FAT的apollo-configservice绑定了域名apollo.fat.xxx.com，UAT的apollo-configservice绑定了域名apollo.uat.xxx.com，PRO的apollo-configservice绑定了域名apollo.xxx.com，那么可以如下修改各环境meta service服务地址，格式为${env}.meta=http://${config-service-url:port}，如果某个环境不需要，也可以直接删除对应的配置项（如lpt.meta）：

dev.meta=http://1.1.1.1:8080

fat.meta=http://apollo.fat.xxx.com

uat.meta=http://apollo.uat.xxx.com

pro.meta=http://apollo.xxx.com

注1: 为了实现meta service的高可用，推荐通过SLB（Software Load Balancer）做动态负载均衡

#### 6部署服务

部署服务，就比较简单了，解压修改完数据后就算部署成功了。

生产环境部署三个服务

其他环境只需要部署 admin config即可

他们公用一个protal

#### 7 客户端调用

没大搞清楚，暂时参考快速安装里的。

### 问题：无法登录Protal门户

使用默认用户和密码无法登录：apollo / admin

可能是配置门户网站数据库（ApolloProtalDB）的时候，配置成了ApolloConfigDB

# 8 Bus+RabbitMQ : 消息总线+消息队列

## 8.1 消息总线Bus介绍和场景

1）什么是消息：

一个事件，需要广播或者单独传递给某个接口。

2）为什么使用：

如上述Git仓库配置更新后，但是其他系统不知道是否更新，可用用它来解决通知的问题。

## 8.2 RabbitMQ消息队列

### 8.2.1 消息队列：

参考：<https://www.cnblogs.com/linjiqin/p/5720865.html>

消息队列中间件是分布式系统中重要的组件，主要解决应用解耦，异步消息，流量削锋等问题，实现高性能，高可用，可伸缩和最终一致性架构。目前使用较多的消息队列有ActiveMQ，RabbitMQ，ZeroMQ，Kafka，MetaMQ，RocketMQ

### 8.2.2 SpringCloud默认推荐使用RabbitMQ

官方文档：

<http://www.rabbitmq.com/getstarted.html>

<http://www.rabbitmq.com/tutorials/tutorial-one-java.html>

中文文档：

<http://rabbitmq.mr-ping.com/>

RabbitMQ是一个消息代理：

它接受和转发消息。您可以将其视为邮局：当您将要发送的邮件放在邮箱中时，您可以确定邮递员最终会将邮件发送给您的收件人。在这个比喻中，RabbitMQ是邮箱、邮局和邮递员。

RabbitMQ和邮局之间的主要区别在于它不处理纸张，而是接受、存储和转发二进制数据 –-- 也就是消息

## 8.3 使用Docker搭建RabbitMQ 3.7

### 8.3.1 安装步骤

1)拉取镜像：docker pull rabbitmq:management

2)查看当前镜像列表：docker images

3)删除指定镜像：docker rmi IMAGE\_ID (如果需要强制删除加 -f) 4)创建容器（直接执行此步，存在执行，不存在自动拉取再执行）

docker run -d --name="myrabbitmq" -p 5672:5672 -p 15672:15672 rabbitmq:management

参数讲解:

run: 创建一个新的容器并运行一个命令

-d: 后台运行容器，并返回容器ID

-p: 端口映射，格式为：主机(宿主)端口:容器端口

--name="rabbitmq": 为容器指定一个名称

### 8.3.2 RabbitMQ访问

RabbitMQ默认创建了一个 guest 用户，密码也是 guest, 如果访问不了记得查看防火墙，端口或者云服务器的安全组管理http://127.0.0.1:15672

### 8.3.3 其他安装方式：

Linux安装：<https://blog.csdn.net/qq_34021712/article/details/72567786>

https://blog.csdn.net/liyuejin/article/details/78410586

windows安装：

http://www.rabbitmq.com/install-windows.html

## 8.4 消息总线Bus整合配置中心

### 8.4.1 启动RabbitMQ

docker run -d -p 5672:5672 -p 15672:15672 rabbitmq:management

### 8.4.2 config-client 加入依赖

<!--配置中心结合消息队列-->

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-bus-amqp</artifactId>  
</dependency>

### 8.4.3 修改配置文件名(使用配置中心时)

application.yml – 》bootstrap.yml

### 8.4.4 config-client 配置文件加RabbitMQ连接并暴露端点

(如果是本机，则可以直接启动，采用默认连接配置)

*#server:  
# port: 8771***eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8761/eureka/  
**spring:  
 application:  
 name:** product-server  
*# zipkin:  
# base-url: http://192.168.153.134:9411/  
 #链路追踪配置  
# sleuth:  
# sampler:  
# probability: 1* **cloud:  
 config:  
 discovery:** *#开启config-server服务* **enabled: true** *#设置config-server服务的id* **service-id:** config-server  
 *#加载配置文件的环境* **profile:** test  
 *#加载配置配件所在节点* **label:** master  
  
 *#rabbitmq地址* **rabbitmq:  
 host:** 192.168.153.134  
 **port:** 5672  
 **username:** guest  
 **password:** guest  
*#暴露全部的监控信息***management:  
 endpoints:  
 web:  
 exposure:  
 include:** "\*"

### 8.4.5 需要刷新配置的地方，增加注解

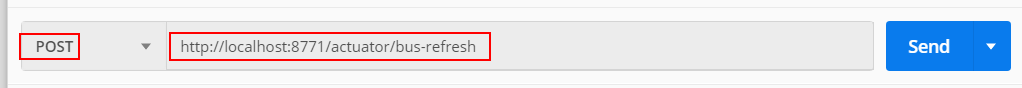
@RefreshScope

@RestController  
@RequestMapping("/product")

@RefreshScope  
public class ProductController {

### 8.4.6 通知刷新配置：访问验证 post方式：

<http://localhost:8773/actuator/bus-refresh>



### 8.4.7动态刷新配置:

在开发和测试环境使用，尽量少在生产环境使用

# 9 Docker容器基础

## 9.1 docker简介

一个开源的应用容器引擎，让开发者可以打包应用及依赖包到一个可移植容器中，然后发布到任何流行的 Linux 机器上，也可以实现虚拟化。

容器是完全使用沙箱机制，相互之间不会有任何接口；

使用go语言编写，在LCX（linux容器）基础上进行的封装

简单来说：

1）就是可以快速部署启动应用

2）实现虚拟化，完整资源隔离

3）一次编写，四处运行（有一定的限制，比如Docker是基于Linux 64bit的，无法在32bit的linux/Windows/unix环境下使用）

## 9.2为什么要用

一句话：开箱即用，快速部署，可移植性强，环境隔离

1、提供一次性的环境，假如需要安装Mysql，则需要安装很多依赖库、版本等，如果使用Docker则通过镜像就可以直接启动运行

2、快速动态扩容，应用可以制作成docker镜像，然后通过Dokcer快速启动

3、组建微服务架构，可以在一个机器上模拟出多个微服务，启动多个应用

4、更好的资源隔离和共享

## 9.3 Linux云服务器Centos7安装Docker

Linux Standard Base的缩写，lsb\_release命令用来显示LSB和特定版本的相关信息

命令： lsb\_release -a

阿里云安装手册：

https://help.aliyun.com/document\_detail/51853.html?spm=a2c4g.11186623.6.820.RaToNY

常见问题：

https://blog.csdn.net/daluguishou/article/details/52080250

## 9.4 Docker仓库、镜像、容器

### 9.4.1：Docker 仓库 - Docker registeries：

用来保存镜像，有公有和私有仓库，好比Maven的中央仓库和本地私服

### 9.4.2 Docker 镜像 - Docker images：

容器运行时的只读模板，操作系统+软件运行环境+用户程序

class User{

private String userName;

private int age;

}

### 9.4.3 Docker 容器 - Docker containers：

容器包含了某个应用运行所需要的全部环境

User user = new User()

镜像仓库：

（参考）配置国内镜像仓库：https://blog.csdn.net/zzy1078689276/article/details/77371782

### 9.4.4 对比面向对象的方式

Dokcer 里面的镜像 : Java里面的类 Class

Docker 里面的容器 : Java里面的对象 Object

通过类创建对象，通过镜像创建容器

## 9.5 Docker容器常见命令

启动docker

systemctl start docker

常用命令（安装部署好Dokcer，执行的命令是docker开头）,xxx是镜像名称

搜索镜像：docker search xxx

列出当前系统存在的镜像：docker images

拉取镜像：docker pull xxx

xxx是具体某个镜像名称(格式 REPOSITORY:TAG)

REPOSITORY：表示镜像的仓库源,TAG：镜像的标签

运行一个容器：docker run -d --name "xdclass\_mq" -p 5672:5672 -p 15672:15672 rabbitmq:management

docker run 运行一个容器

-d 后台运行

-p 端口映射

rabbitmq:management(格式 REPOSITORY:TAG)，如果不指定tag，默认使用最新的)

--name "xxx"

列举当前运行的容器：docker ps

可选参数 -a ：所有容器（运行的/未运行的）

检查容器内部信息：docker inspect 容器名称

删除镜像：docker rmi IMAGE\_NAME

强制移除镜像不管是否有容器使用该镜像 增加 -f 参数，

停止某个容器：docker stop 容器名称或ID

启动某个容器：docker start 容器名称或ID

移除某个容器：docker rm 容器名称或ID （容器必须是停止状态）

查看启动日志 docker logs -f containerid（启动返回的那个大长值）

文档：

https://blog.csdn.net/permike/article/details/51879578

## 9.6 Docker部署Nginx服务器

### 9.6.1获取镜像

docker run (首先会从本地找镜像，如果有则直接启动，没有的话，从镜像仓库拉起，再启动)

docker search nignx

### 9.6.2列举

docker images

### 9.6.3拉取

docker pull nignx

### 9.6.3启动

docker run -d --name "xdclass\_nginx" -p 8088:80 nginx

docker run -d --name "xdclass\_nginx2" -p 8089:80 nginx

docker run -d --name "xdclass\_nginx3" -p 8090:80 nginx

### 9.6.4访问

如果是阿里云服务，记得配置安全组，腾讯云也需要配置，这个就是一个防火墙

## 9.7 Docker镜像仓库使用讲解

### 9.7.1为啥要用镜像仓库

### 9.7.2官方公共镜像仓库和私有镜像仓库

1) 公共镜像仓库：

官方：https://hub.docker.com/，基于各个软件开发或者有软件提供商开发的

非官方：其他组织或者公司开发的镜像，供大家免费试用

2) 私有镜像仓库：

用于存放公司内部的镜像，不提供给外部试用；

SpringCloud 开发了一个支付系统 -》做成一个镜像 （操作系统+软件运行环境+用户程序）

## 9.8构建自己的镜像仓库

### 9.8.1阿里云镜像仓库：

https://dev.aliyun.com/search.html

点击管理控制台-》初次使用会提示开通，然后设置密码

xdclass.net123

### 9.8.2使用阿里云私有镜像仓库

#### 1)登录：

docker login --username=794666918@qq.com registry.cn-shenzhen.aliyuncs.com

#### 2) 推送本地镜像：

登录：

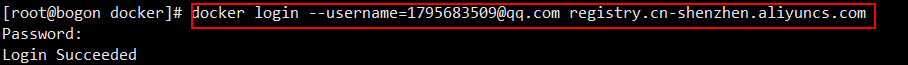
docker login --username=1795683509@qq.com registry.cn-shenzhen.aliyuncs.com

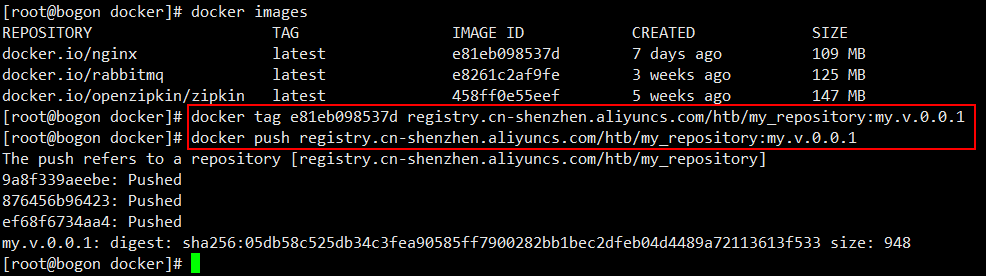
第二步：

docker tag e81eb098537d registry.cn-shenzhen.aliyuncs.com/htb/my\_repository:my.v.0.0.1

第三步：

docker push registry.cn-shenzhen.aliyuncs.com/htb/my\_repository:my.v.0.0.1





Jenkins 自动推送镜像

docker tag [ImageId] registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:[镜像版本号]

例子：

docker tag 2f415b0e9a6e registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:xd\_rabbitmq-v1.0.2

docker push registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:xd\_rabbitmq-v1.0.2

#### 3)线上服务器拉取镜像：

docker login --username=794666918@qq.com registry.cn-shenzhen.aliyuncs.com

docker pull registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:xd\_rabbitmq-v1.0.2

#### 4）启动容器：

docker run -d --name "xdclass\_mq" -p 5672:5672 -p 15672:15672 2f415b0e9a6e

## 9.9 阿里云镜像仓库：镜像推送-拉取 详细操作

### 1. 登录阿里云Docker Registry （root用户的话不用加sudo，不然可能会报错）

$ sudo docker login --username=1795683509@qq.com registry.cn-shenzhen.aliyuncs.com

用于登录的用户名为阿里云账号全名，密码为开通服务时设置的密码。

您可以在产品控制台首页修改登录密码。

### 2. 从Registry中拉取镜像

$ sudo docker pull registry.cn-shenzhen.aliyuncs.com/htb/my\_repository:[镜像版本号]

### 3. 将镜像推送到Registry

$ sudo docker login --username=1795683509@qq.com registry.cn-shenzhen.aliyuncs.com$ sudo docker tag [ImageId] registry.cn-shenzhen.aliyuncs.com/htb/my\_repository:[镜像版本号]

$ sudo docker push registry.cn-shenzhen.aliyuncs.com/htb/my\_repository:[镜像版本号]

请根据实际镜像信息替换示例中的[ImageId]和[镜像版本号]参数。

### 4. 选择合适的镜像仓库地址

从ECS推送镜像时，可以选择使用镜像仓库内网地址。推送速度将得到提升并且将不会损耗您的公网流量。

如果您使用的机器位于经典网络，请使用 registry-internal.cn-shenzhen.aliyuncs.com 作为Registry的域名登录，并作为镜像命名空间前缀。

如果您使用的机器位于VPC网络，请使用 registry-vpc.cn-shenzhen.aliyuncs.com 作为Registry的域名登录，并作为镜像命名空间前缀。

### 5. 示例

使用"docker tag"命令重命名镜像，并将它通过专有网络地址推送至Registry。

$ sudo docker images

REPOSITORY TAG IMAGE ID CREATED VIRTUAL SIZE

registry.aliyuncs.com/acs/agent 0.7-dfb6816 37bb9c63c8b2 7 days ago 37.89 MB

$ sudo docker tag 37bb9c63c8b2 registry-vpc.cn-shenzhen.aliyuncs.com/acs/agent:0.7-dfb6816

使用"docker images"命令找到镜像，将该镜像名称中的域名部分变更为Registry专有网络地址。

$ sudo docker push registry-vpc.cn-shenzhen.aliyuncs.com/acs/agent:0.7-dfb6816

# 10 SpringCloud和Docker整合部署

## 10.1 使用Docker的maven插件，构建springboot应用

官方文档：https://spring.io/guides/gs/spring-boot-docker/

### 10.1.1 步骤：maven里面添加配置pom.xml

<properties>

<docker.image.prefix>xdclass</docker.image.prefix>

</properties>

<build>

<finalName>docker-demo</finalName>

<plugins>

<plugin>

<groupId>com.spotify</groupId>

<artifactId>dockerfile-maven-plugin</artifactId>

<version>1.3.6</version>

<configuration>

<repository>${docker.image.prefix}/${project.artifactId}</repository>

<buildArgs>

<JAR\_FILE>target/${project.build.finalName}.jar</JAR\_FILE>

</buildArgs>

</configuration>

</plugin>

</plugins>

</build>

配置讲解

Spotify 的 docker-maven-plugin 插件是用maven插件方式构建docker镜像的。

${project.build.finalName} 产出物名称，缺省为${project.artifactId}-${project.version}

## 10.2 方式一： 本地安装有Docker：：打包SpringCloud镜像并上传私有仓库并部署

### 10.2.1创建Dockerfile,

默认是根目录，（可以修改为src/main/docker/Dockerfile,如果修则需要制定路径）

什么是Dockerfile : 由一系列命令和参数构成的脚本，这些命令应用于基础镜像, 最终创建一个新的镜像

FROM openjdk:8-jdk-alpine

VOLUME /tmp

ARG JAR\_FILE

COPY ${JAR\_FILE} app.jar

ENTRYPOINT ["java","-jar","/app.jar"]

参数讲解：

FROM <image>:<tag> 需要一个基础镜像，可以是公共的或者是私有的， 后续构建会基于此镜像，如果同一个Dockerfile中建立多个镜像时，可以使用多个FROM指令

VOLUME 配置一个具有持久化功能的目录，主机 /var/lib/docker 目录下创建了一个临时文件，并链接到容器的/tmp。改步骤是可选的，如果涉及到文件系统的应用就很有必要了。/tmp目录用来持久化到 Docker 数据文件夹，因为 Spring Boot 使用的内嵌 Tomcat 容器默认使用/tmp作为工作目录

ARG 设置编译镜像时加入的参数， ENV 是设置容器的环境变量

COPY : 只支持将本地文件复制到容器 ,还有个ADD更强大但复杂点

ENTRYPOINT 容器启动时执行的命令

EXPOSE 8080 暴露镜像端口

### 10.2.2构建镜像

博客：

https://www.cnblogs.com/modprobe/p/6033807.html

构建镜像

配置镜像推送主机：

环境变量

变量名：DOCKER\_HOST

变量值：tcp://120.79.223.147:2375

mvn install dockerfile:build

打标签

docker tag a1b9fc71720d registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:docker-demo-v201808

推送到镜像仓库

docker push registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:docker-demo-v201808

应用服务器拉取镜像

docker pull registry.cn-shenzhen.aliyuncs.com/xdclass/xdclass\_images:docker-demo-v201808

docker run -d --name xdclass\_docker\_demo1 -p 8099:8080 a1b9fc71720d

### 10.2.3查看启动日志

docker logs -f containerid

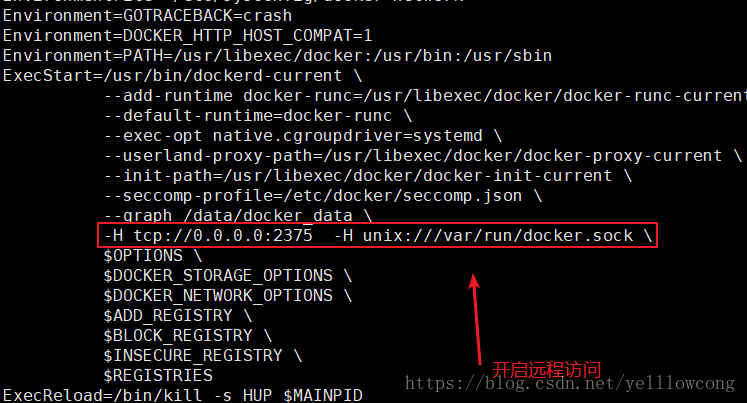
文档：https://yeasy.gitbooks.io/docker\_practice/image/dockerfile/

## 10.3 方式二： 通过调用远程的Docker制作镜像

### 10.3.1 远程Docker服务器端的配置（虚拟机端）

vim /usr/lib/systemd/system/docker.service

ExecStart=/usr/bin/dockerd -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock



#重新加载配置文件

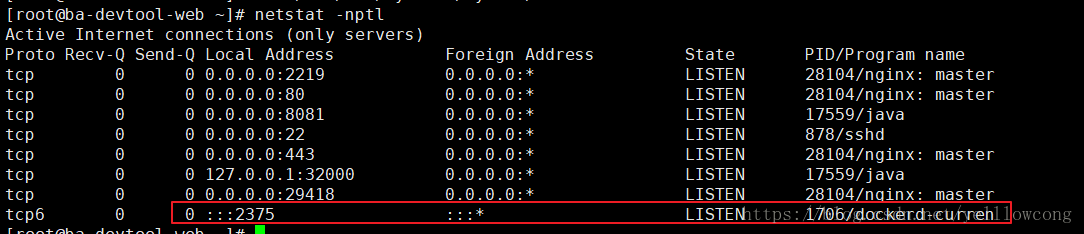
systemctl daemon-reload

#重启服务

systemctl restart docker.service

#查看端口是否开启

netstat -nptl



永久对外开发端口：

firewall-cmd --zone=public --add-port=2375/tcp --permanent  
查看端口列表：

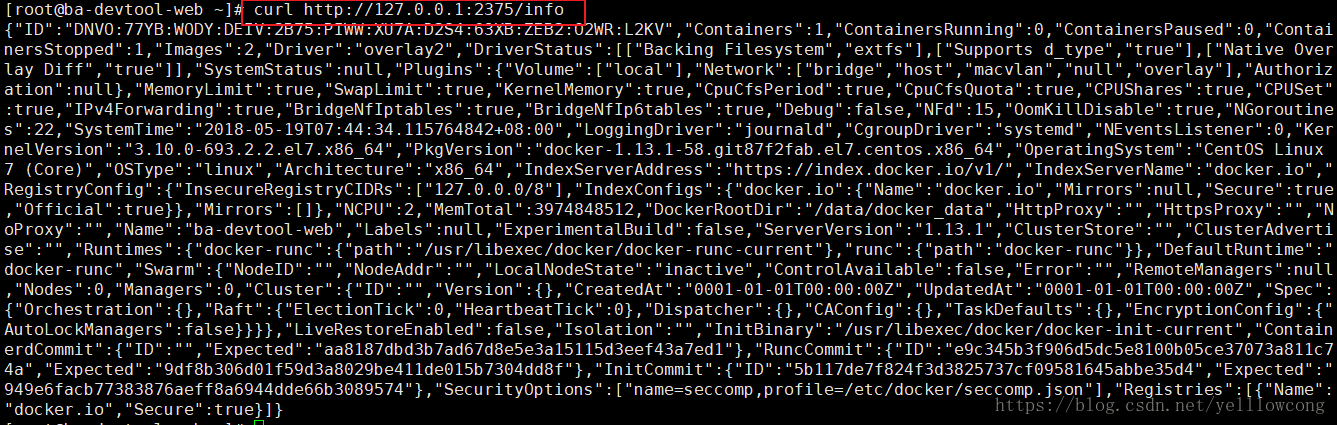
firewall-cmd --permanent --list-port

重启防火墙命令：

systemctl restart firewalld.service

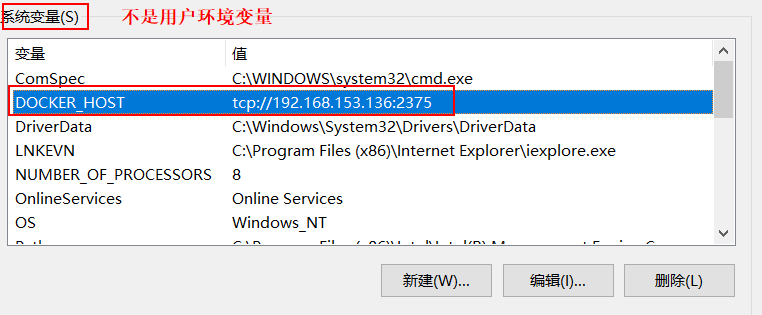
#直接curl看是否生效

curl http://120.79.223.147:2375/info



### 10.3.2 宿主机：源码所在计算机（虚拟机安装所在计算机）

1）添加“系统环境变量”



2）重启宿主机

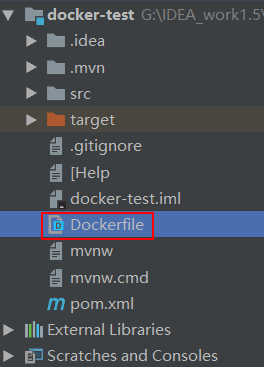
3）在Linux中启动docker

4）项目中添加依赖：

<properties>  
 <docker.image.prefix>cn.htb</docker.image.prefix>  
</properties>

<build>  
 <finalName>docker-image-test</finalName>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
  
 <plugin>  
 <groupId>com.spotify</groupId>  
 <artifactId>dockerfile-maven-plugin</artifactId>  
 <version>1.3.6</version>  
 <configuration>  
 <repository>${docker.image.prefix}/${project.artifactId}</repository>  
 <buildArgs>  
 <JAR\_FILE>target/${project.build.finalName}.jar</JAR\_FILE>  
 </buildArgs>  
 </configuration>  
 </plugin>  
 </plugins>  
</build>

5） 在项目根目录创建文件Dockerfile



内容：

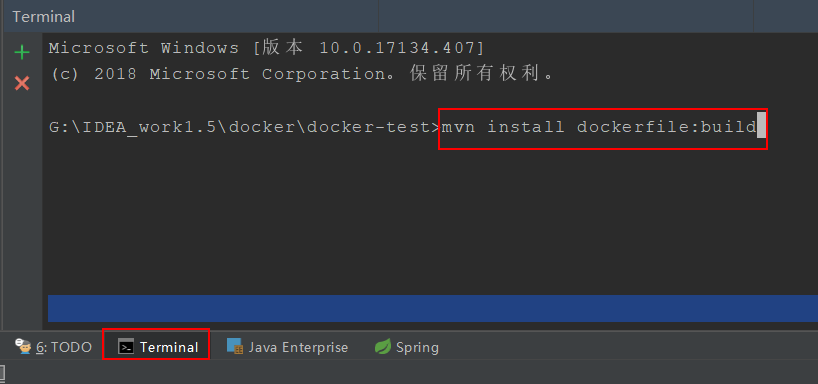
**FROM** openjdk:8-jdk-alpine  
**VOLUME /**tmp  
**ARG** *JAR\_FILE***COPY** ${*JAR\_FILE*} app.jar  
**ENTRYPOINT** ["java","-jar","/app.jar"]

5） 修改配置中的注册中心声明：

*#注册中心地址*

**eureka:  
 client:  
 service-url:  
 defaultZone:** http://192.168.153.136:8761/eureka/  
 **instance:  
 instance-id:** ${**spring.cloud.client.ip-address**}:${**server.port**}  
 **prefer-ip-address: true**

6） 开始构建：打开IDEA 的 Terminal



7）输入指令：

mvn install dockerfile:build

## 10.4 问题 ：在虚拟机中项目注册或通讯

### 10.4.1 服务无法注册：url地址保证正确



### 10.4.2 服务间无法通讯

在服务的注册中心地址声明出加上：

**instance:  
 instance-id:** ${**spring.cloud.client.ip-address**}:${**server.port**}  
 **prefer-ip-address: true**

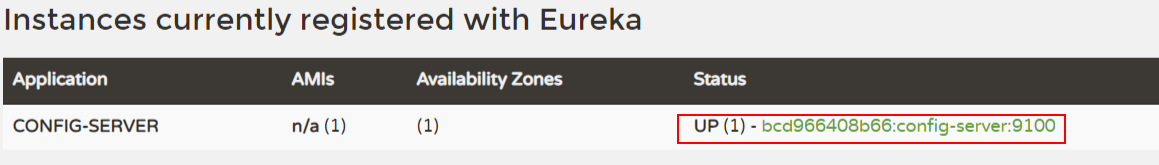
即：

*#注册中心地址*

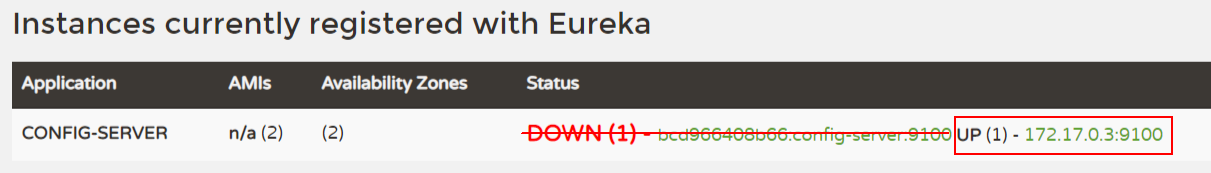
**eureka:  
 client:  
 service-url:  
 defaultZone:** http://192.168.153.136:8761/eureka/  
 **instance:  
 instance-id:** ${**spring.cloud.client.ip-address**}:${**server.port**}  
 **prefer-ip-address: true**

区别：

1） 不加时，在注册中心的样式：



2）加时，在注册中心的样式：



## 10.5 其他服务镜像打包：商品-订单

区别： 此类的服务主要是需要做俩件事：注册服务，从配置中心拉取配置

10.5.1 maven打包构建，会触发单元测试，部分情况要跳过，否则可能抛出异常

如：在源码中需要从配置文件中或取一个值,使用占位符${env},会抛出错误con`t resolve the placeholder ${env} 无法处理这个占位符

所以要使用：mvn install -Dmaven.test.skip=true dockerfile:build

10.5.2生产环境不能用localhost 或者 127.0.0.1, 一定要用内网通信ip (虚拟主机映射 hosts)

# 11 课程总结和常见问题处理

## 11.1 SpringCloud微服务常见问题和解决思路

1、技术选择：SpringCloud全家桶，每个组件又有多个替代，改怎么选择

1）选择的时候根据公司里面团队人员熟悉程度，降低学习成本

2）选择社区活跃的并且文档相对较多的，怎么判断活跃，可以看github上代码提交和start数

2、云服务选择： 腾讯云，阿里云 ，遇到问题可以提交工单，有专人跟进

## 11.2 部署了应用，但是访问不了

解决思路：

1)查看应用启动是否正常，如果有错误日志，复制错误日志去百度搜索！！！！！！！特别重要

2）启动正常，则先在本机使用 CURL "http://lcoalhost:8080/api/v1/user/find" 访问对应的接口，看是否有响应

3）启动正常，且curl有响应，则检查是否有关闭防火墙，或者开放对应的访问端口，开放端口才可以访问

4）腾讯云和阿里云都是有安全组，类似外层防火墙，一定要去web控制台检查是否有开启端口

## 11.3 内网和外网访问，鉴权问题安全问题

1）所有应用只能通过网关提供对外访问的入口

2）应用程序之间通讯，采用内网

## 11.4 、程序出错或者异常:

复制错误日志去百度搜索，网上有很多人会遇到类似的错误，多积累，特别强调

# 12课程总结和后续技术规划

1、springCloud全家桶，技术选择和知识点特别多，一定要学会记笔记，微服务更多关注的是里面架构和数据流转，而不是具体的业务。

2、SpringCloud架构这些配置，一般使用一次后就不会多次修改了，进入公司主要还是开发业务，业务开发一般都用springboot，比较少让新人搭建SpringCloud的架构

3、开发业务，无非就是CRUD，增删改查，只不过是初级和高级的区别，封装成通用和不通用的区别

4、微服务和容器盛行的情况下，容器编排和自动缩扩容越来越重要

可以关注：k8s / service mesh /server less 等技术

# 13 阿里云ECS服务器介绍和网络知识讲解

## 9.1 什么是云服务器及目前主要的几个厂商介绍

阿里云、腾讯云、亚马逊云（京东云、百度云）

阿里云：https://www.aliyun.com/

腾讯云：https://cloud.tencent.com/

亚马逊云：<https://aws.amazon.com/>

## 9.2 阿里云服务器远程登录和常用终端工具

1）windows工具 putty，xshell, security

参考资料：

https://jingyan.baidu.com/article/e75057f210c6dcebc91a89dd.html

https://www.jb51.net/softjc/88235.html

2)苹果系统MAC： 通过终端登录

ssh root@ip 回车后输入密码

ssh root@120.25.1.38

3)可以尝试自己通过百度进行找文档， 安装mysql jdk nginx maven git redis elk

9.3 互联网架构知识之网站部署上线基础准备

简介：讲解应用部署到可以公网访问需要步骤

1、一个http请求的详细过程

2、什么是cname和a记录

A记录和CNAME只可以同时生效一个，A记录优先

1)A记录    
又称IP指向，用户可以在此设置子域名并指向到自己的目标主机地址上，从而实现通过域名找到服务器。   
说明：指向的目标主机地址类型只能使用IP地址；      
  2) CNAME    
通常称别名指向。可以为一个主机设置别名。比如设置test.mydomain.com，用来指向一个主机www.rddns.com那么以后就可以用test.mydomain.com来代替访问www.rddns.com了。   
说明：CNAME的目标主机地址只能使用主机名，不能使用IP地址；

·主机名前不能有任何其他前缀，如：http://等是不被允许的；

·A记录优先于CNAME记录。即如果一个主机地址同时存在A记录和CNAME记录，则CNAME记录不生效。

3、域名和ip的关系，DNS作用

192.120.1.38

cn.htb.com

远端DNS服务

DNS解析

IP地址

域名

本地不存在则远端解析

解析域名

得到对应的IP

参考资料：

https://blog.csdn.net/benbenzhuhwp/article/details/44704319

https://www.baidu.com/s?ie=utf-8&f=8&rsv\_bp=0&rsv\_idx=1&ch=1&tn=98050039\_dg&wd=%E4%B8%80%E4%B8%AAhttp%E8%AF%B7%E6%B1%82%E7%9A%84%E8%AF%A6%E7%BB%86%E8%BF%87%E7%A8%8B&rsv\_pq=80a65c5f00005961&rsv\_t=a5fcWreuJzILdSwr4gI8pFqlO7HSu5BlhjwalyVzPiV9w2L%2BKEj78pPi1Qn6Vx4wXxI&rqlang=cn&rsv\_enter=1&rsv\_sug3=8&rsv\_sug1=8&rsv\_sug7=100&sug=%25E4%25B8%2580%25E4%25B8%25AAhttp%25E8%25AF%25B7%25E6%25B1%2582%25E7%259A%2584%25E8%25AF%25A6%25E7%25BB%2586%25E8%25BF%2587%25E7%25A8%258B&rsv\_n=1

9.4域名购买和配置解析实战

简介：域名购买和配置解析实战

1、购买域名，备案

阿里云 备案地址：https://beian.aliyun.com/

2、购买服务器，阿里云，腾讯云，亚马逊云aws

3、配置域名解析到服务器

# 扩展：

### 1 三种通讯应用层协议：

三种通用应用层协议protobuf、thrift、avro对比,完爆xml,json,http.

Google protobuf：  
优点  
    二进制消息，性能好/效率高（空间和时间效率都很不错）  
    proto文件生成目标代码，简单易用  
    序列化反序列化直接对应程序中的数据类，不需要解析后在进行映射(XML,JSON都是这种方式)  
    支持向前兼容（新加字段采用默认值）和向后兼容（忽略新加字段），简化升级  
    支持多种语言（可以把proto文件看做IDL文件）  
    Netty等一些框架集成  
  
缺点  
  
    官方只支持C++,JAVA和Python语言绑定  
    二进制可读性差（貌似提供了Text\_Fromat功能）  
    二进制不具有自描述特性  
    默认不具备动态特性（可以通过动态定义生成消息类型或者动态编译支持）  
    只涉及序列化和反序列化技术，不涉及RPC功能（类似XML或者JSON的解析器）  
  
Apache Thrift：  
应用  
  
    Facebook的开源的日志收集系统(scribe: <https://github.com/facebook/scribe>)  
    淘宝的实时数据传输平台(TimeTunnel<http://code.taobao.org/p/TimeTunnel/wiki/index>)  
    Evernote开放接口(<https://github.com/evernote/evernote-thrift>)  
    Quora(<http://www.quora.com/Apache-Thrift>)  
    HBase( <http://abloz.com/hbase/book.html#thrift> )  
    …  
  
优点  
  
    支持非常多的语言绑定  
    thrift文件生成目标代码，简单易用  
    消息定义文件支持注释  
    数据结构与传输表现的分离，支持多种消息格式  
    包含完整的客户端/服务端堆栈，可快速实现RPC  
    支持同步和异步通信  
  
缺点  
  
    和protobuf一样不支持动态特性  
  
Apache Avro：  
应用  
  
    Hadoop RPC （<http://hadoop.apache.org/#What+Is+Apache+Hadoop%3F>）  
  
优点  
  
    二进制消息，性能好/效率高  
    使用JSON描述模式  
    模式和数据统一存储，消息自描述，不需要生成stub代码（支持生成IDL）  
    RPC调用在握手阶段交换模式定义  
    包含完整的客户端/服务端堆栈，可快速实现RPC  
    支持同步和异步通信  
    支持动态消息  
    模式定义允许定义数据的排序（序列化时会遵循这个顺序）  
    提供了基于Jetty内核的服务基于Netty的服务  
  
缺点  
  
    只支持Avro自己的序列化格式  
    语言绑定不如Thrift丰富

### 2 TCP的三次握手四次挥手：

http://www.cocoachina.com/programmer/20180314/22588.html

第一次

第一次握手：建立连接时，[客户端](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E7%AB%AF)发送[syn](https://baike.baidu.com/item/syn)包（syn=j）到[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8" \t "_blank)，并进入[SYN\_SENT](https://baike.baidu.com/item/SYN_SENT)状态，等待服务器确认；SYN：同步序列编号（***Synchronize Sequence Numbers***）。

第二次

[第二次握手](https://baike.baidu.com/item/%E7%AC%AC%E4%BA%8C%E6%AC%A1%E6%8F%A1%E6%89%8B)：[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8)收到[syn](https://baike.baidu.com/item/syn)包，必须确认客户的SYN（[ack](https://baike.baidu.com/item/ack)=j+1），同时自己也发送一个SYN包（syn=k），即SYN+ACK包，此时服务器进入[SYN\_RECV](https://baike.baidu.com/item/SYN_RECV)状态；

第三次

第三次握手：[客户端](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E7%AB%AF" \t "_blank)收到[服务](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1)器的SYN+ACK包，向[服务器](https://baike.baidu.com/item/%E6%9C%8D%E5%8A%A1%E5%99%A8" \t "_blank)发送确认包ACK([ack](https://baike.baidu.com/item/ack)=k+1），此包发送完毕，客户端和服务器进入[ESTABLISHED](https://baike.baidu.com/item/ESTABLISHED" \t "_blank)（TCP连接成功）状态，完成三次握手。

完成三次握手，[客户端](https://baike.baidu.com/item/%E5%AE%A2%E6%88%B7%E7%AB%AF)与服务器开始传送[数据](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE)，在上述过程中，还有一些重要的概念：

### 3 HttpClient URLConnection

### 4 ObjectMapper BeanUtils StringUtils

### 5 线程睡眠新方法：

新方法，可读性好

//模拟线程超时新方法：睡眠一秒

try {  
 TimeUnit.*SECONDS*.sleep(1);  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}

旧方法，可读性差

//模拟线程超时旧方法：睡眠一秒

try {  
 Thread.*sleep*(1000);  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}

### 6 修改maven仓库地址

pom.xml中修改

<repositories>

<repository>

<id>nexus-aliyun</id>

<name>Nexus aliyun</name>

<layout>default</layout> <url>http://maven.aliyun.com/nexus/content/groups/public</url>

<snapshots>

<enabled>false</enabled>

</snapshots>

<releases>

<enabled>true</enabled>

</releases>

</repository>

</repositories>

### 7 http请求头 token cookie

# 问题：

1 pom.xml 依赖可能有红色波浪线，但是文件都确定已经下载完成

解决：删除pom.xml的相关依赖，保存报错消失，在撤回添加依赖。

2 远程服务无法调用



linuxDocker安装：<https://help.aliyun.com/document_detail/51853.html>

2 阿里云资料：

<https://help.aliyun.com/document_detail/57160.html?spm=a2c4g.11186623.3.2.58627c2blDdOBf>

# com.netflix.client.ClientException: Load balancer does not have available server for client

解决：

1 查看 被调用的服务名是否真的和Eureka中注册的匹配

2 添加：

**ribbon:  
 eureka:  
 enabled: true**

Eureka-server

Eureka-client

Eureka-client

注册与发现

注册与发现

根据从注册中心获取的地址调用

Ribbon/Feign

Hystrix断路器

Zuul网关：统一接口，监控，流量控制

config-server

注册

获取配置信息

Git仓库

cang

/d

Nignx+LVS+keepalive

Sleuth+Zipkin

链路追踪系统