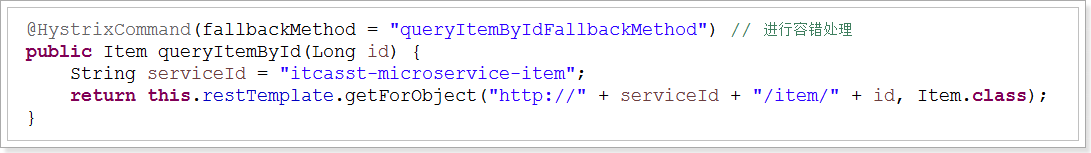
# 今日大纲

1. 使用Feign实现声明式的REST调用
2. 服务网关 Spring Cloud Zuul
3. 使用Spring Cloud Config统一使用管理微服务的配置
4. 使用Spring Cloud Bus（消息总线）实现自动更新

# 使用Feign实现声明式的REST调用

## 分析

之前我们通过RestTemplate调用REST服务，代码是这样的：

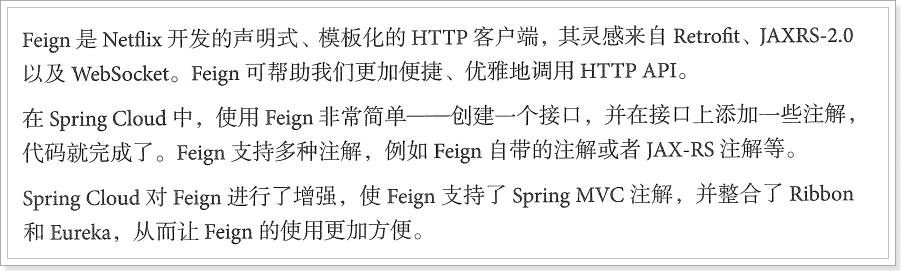


虽然使用了Ribbon和Hystrix可以实现负载均衡和容错处理，但是这个编码在实现大量业务时会显得太过于冗余（如，多参数的URL拼接）。

有没有更加优雅的实现呢？

## Feign的简介

项目主页：<https://github.com/OpenFeign/feign>



## 快速入门

在订单微服务中增加Feign的支持。

### 导入依赖

<dependency>

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

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

</dependency>

### 创建一个ItemFeignClient接口

**package** cn.itcast.microservice.order.feign;

**import** org.springframework.cloud.netflix.feign.FeignClient;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RequestMapping;

**import** org.springframework.web.bind.annotation.RequestMethod;

**import** cn.itcast.microservice.order.pojo.Item;

@FeignClient(value = "itcast-microservice-item") // 申明这是一个Feign客户端，并且指明服务id

**public** **interface** ItemFeignClient {

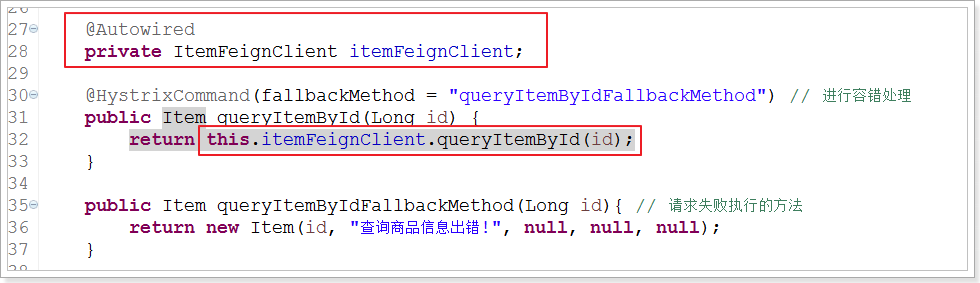
// 这里定义了类似于SpringMVC用法的方法，就可以进行RESTful的调用了

@RequestMapping(value = "/item/{id}", method = RequestMethod.***GET***)

**public** Item queryItemById(@PathVariable("id") Long id);

}

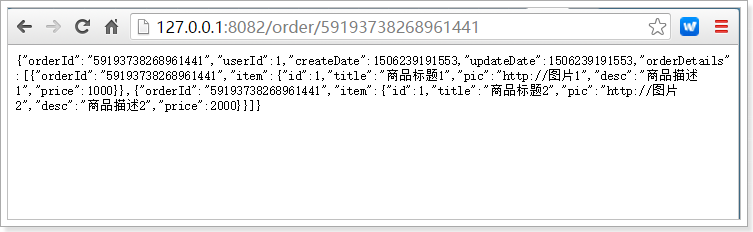
### 改造ItemService



### 在启动类中添加@EnableFeignClients注解

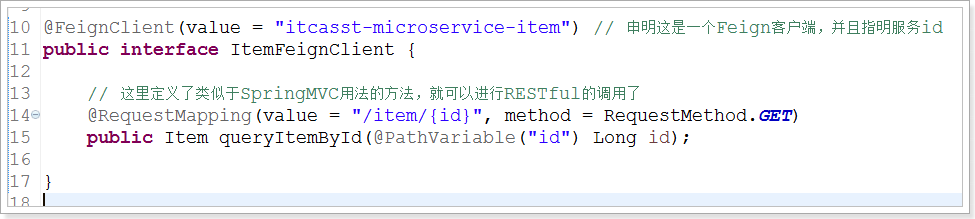


### 重启测试



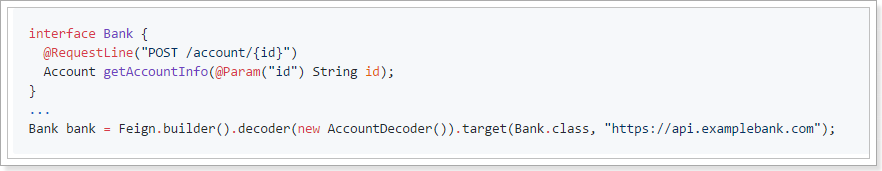
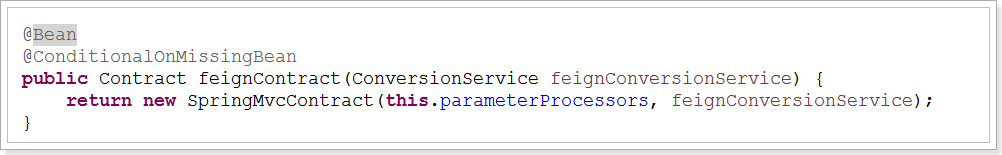
测试结果，一切正常。

## 到底发生了什么？

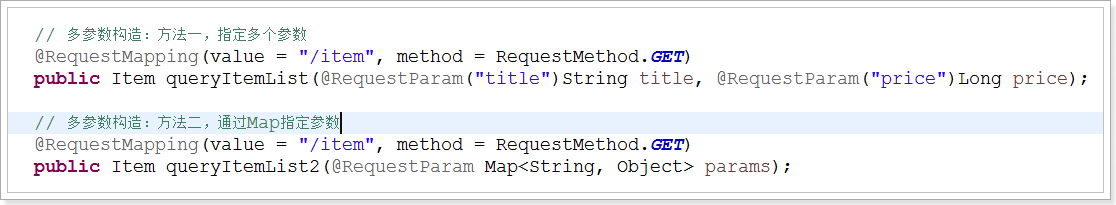


写这样的代码，就可以访问RESTful服务啦？

流程分析：

1. 由于我们在入口@EnableFeignClients注解，Spring启动后会扫描标注了@FeignClient注解的接口，然后生成代理类
2. 我们在@FeignClient接口中指定了value，其实就是指定了在Eureka中的服务名称
3. 在FeignClient中的定义方法以及使用了SpringMVC的注解，Feign就会根据注解中的内容生成对应的URL，然后基于Ribbon的负载均衡去调用REST服务
   1. 为什么使用的是SpringMVC的注解？
      1. 其实，Feign是有自己的注解的，是因为Spring Cloud对Feign做了增强，兼容了SpringMVC的注解，使我们的学习成本更低  
         
      2. 专业的解释是这样的：  
         org.springframework.cloud.netflix.feign.FeignClientsConfiguration  
           
         设置的默认的契约是SpringMVC契约。

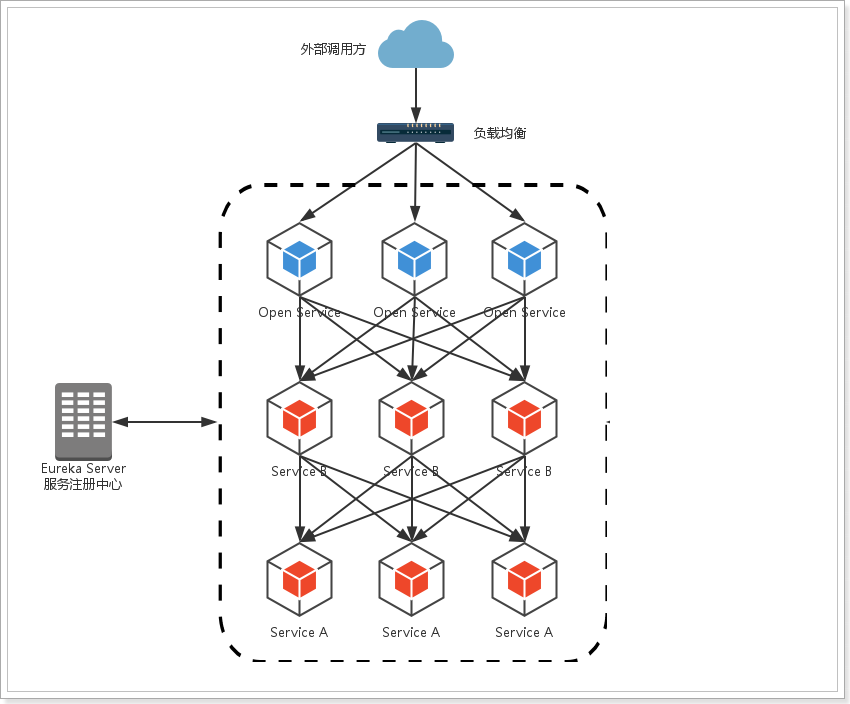
## Feign的多参数构造



# 服务网关 Spring Cloud Zuul

## 分析

通过前面的学习，使用Spring Cloud实现微服务的架构基本成型，大致是这样的：



我们使用Spring Cloud Netflix中的Eureka实现了服务注册中心以及服务注册与发现；而服务间通过Ribbon或Feign实现服务的消费以及均衡负载；通过Spring Cloud Config实现了应用多环境的外部化配置以及版本管理。为了使得服务集群更为健壮，使用Hystrix的融断机制来避免在微服务架构中个别服务出现异常时引起的故障蔓延。

在该架构中，我们的服务集群包含：内部服务Service A和Service B，他们都会注册与订阅服务至Eureka Server，而Open Service是一个对外的服务，通过均衡负载公开至服务调用方。我们把焦点聚集在对外服务这块，这样的实现是否合理，或者是否有更好的实现方式呢？

先来说说这样架构需要做的一些事儿以及存在的不足：

* 首先，破坏了服务无状态特点。
  + 为了保证对外服务的安全性，我们需要实现对服务访问的权限控制，而开放服务的权限控制机制将会贯穿并污染整个开放服务的业务逻辑，这会带来的最直接问题是，破坏了服务集群中REST API无状态的特点。
  + 从具体开发和测试的角度来说，在工作中除了要考虑实际的业务逻辑之外，还需要额外可续对接口访问的控制处理。
* 其次，无法直接复用既有接口。
  + 当我们需要对一个即有的集群内访问接口，实现外部服务访问时，我们不得不通过在原有接口上增加校验逻辑，或增加一个代理调用来实现权限控制，无法直接复用原有的接口。

面对类似上面的问题，我们要如何解决呢？ 答案是：服务网关！

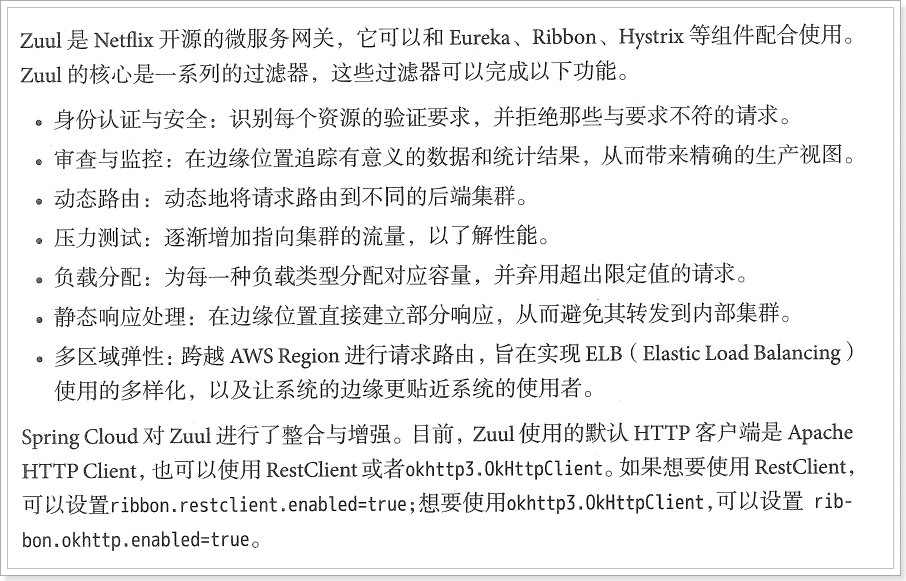
为了解决上面这些问题，我们需要将权限控制这样的东西从我们的服务单元中抽离出去，而最适合这些逻辑的地方就是处于对外访问最前端的地方，我们需要一个更强大一些的均衡负载器 🡺 服务网关。

服务网关是微服务架构中一个不可或缺的部分。通过服务网关统一向外系统提供REST API的过程中，除了具备服务路由、均衡负载功能之外，它还具备了权限控制等功能。Spring Cloud Netflix中的Zuul就担任了这样的一个角色，为微服务架构提供了前门保护的作用，同时将权限控制这些较重的非业务逻辑内容迁移到服务路由层面，使得服务集群主体能够具备更高的可复用性和可测试性。

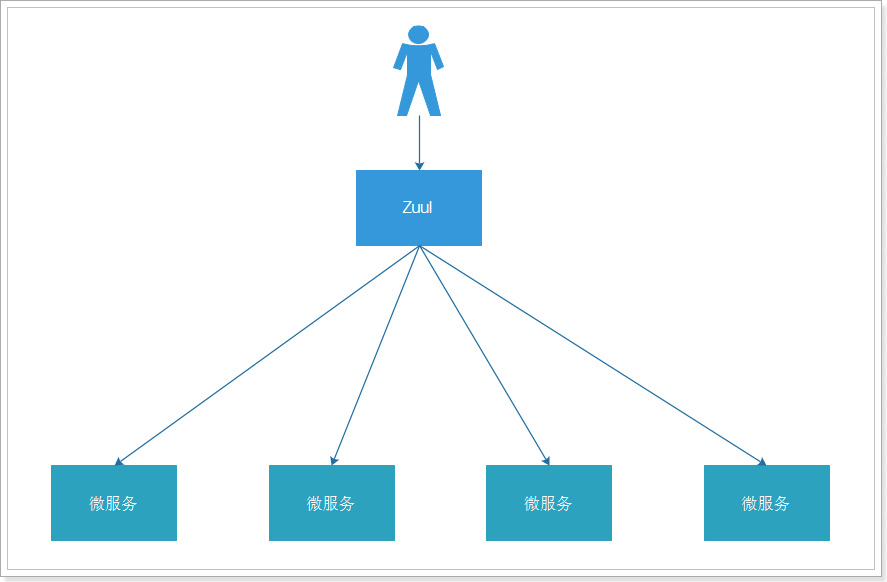
## Zuul的简介

官网：<https://github.com/Netflix/zuul>





## 使用Zuul之后的架构

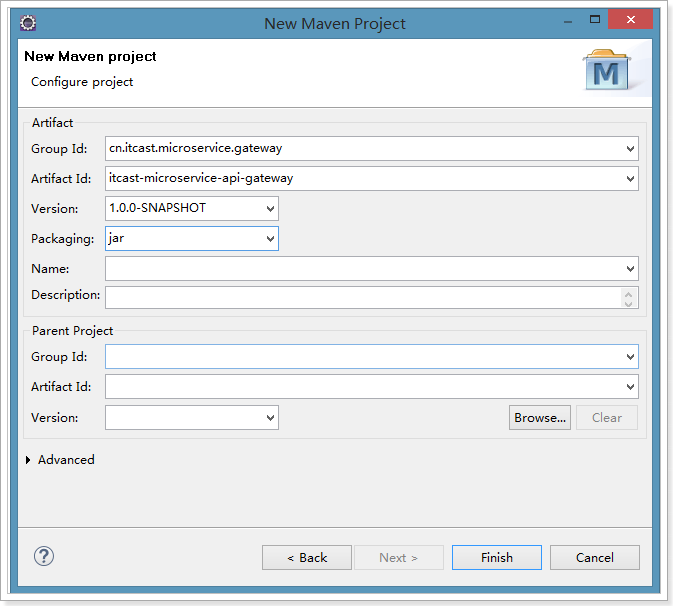


从架构图中可以看出，客户端请求微服务时，先经过Zuul之后再请求，这样就可以将一些类似于校验的业务逻辑放到zuul中完成。

而微服务自身只需要关注自己的业务逻辑即可。

## 快速入门

### 创建工程itcast-microsrvice-api-gateway



### 导入依赖

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice.gateway</groupId>

<artifactId>itcast-microservice-api-gateway</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 导入Spring Cloud的依赖管理 -->

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR3</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

### 编写启动类ApiGatewayApplication

**package** cn.itcast.microservice;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@EnableZuulProxy

@SpringBootApplication

**public** **class** ApiGatewayApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ApiGatewayApplication.**class**, args);

}

}

### 编写application.yml文件

server:

port: 6677 #服务端口

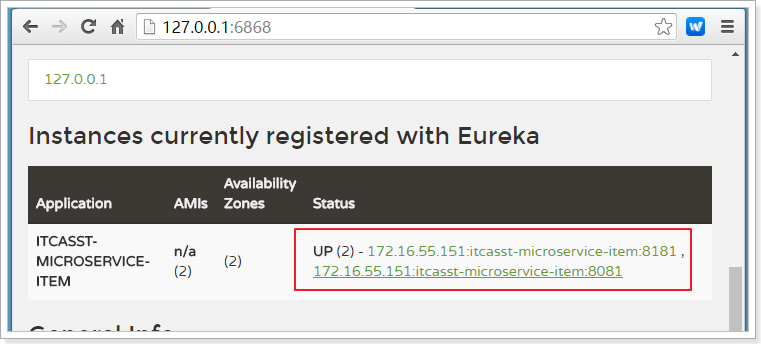
spring:

application:

name: itcasst-microservice-api-gateway #指定服务名

### 编写路由规则

首先，查看Eureka中的服务：



可以看到，当前Eureka中有2个商品的微服务。

接下来，我们编写路由规则：

server:

port: 6677 #服务端口

spring:

application:

name: itcasst-microservice-api-gateway #指定服务名

zuul:

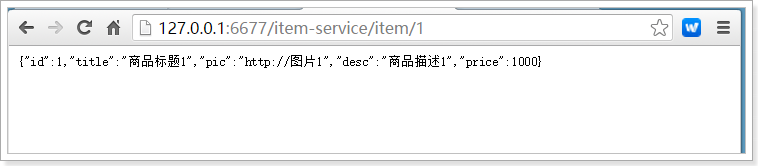
routes:

item-service: #item-service这个名字是任意写的

path: /item-service/\*\* #配置请求URL的请求规则

url: http://127.0.0.1:8081 #真正的微服务地址

### 启动测试



可以看到，已经通过zuul访问到了商品微服务。

## 面向服务的路由

在快速入门中我们已经通过了URL映射，访问到了商品微服务。这样做会存在一个问题，就是，如果商品微服务的地址发生了改变怎么办？

很自然的能够想到，不应该配置具体的url而是走Eureka注册中心获取地址。

### 添加Eureka服务的依赖

<dependency>

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

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

<exclusions>

<exclusion>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</exclusion>

</exclusions>

</dependency>

### 修改application.yml配置文件

server:

port: 6677 #服务端口

spring:

application:

name: itcasst-microservice-api-gateway #指定服务名

zuul:

routes:

item-service: #item-service这个名字是任意写的

path: /item-service/\*\* #配置请求URL的请求规则

#url: http://127.0.0.1:8081 #真正的微服务地址

serviceId: itcast-microservice-item #指定Eureka注册中心中的服务id

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

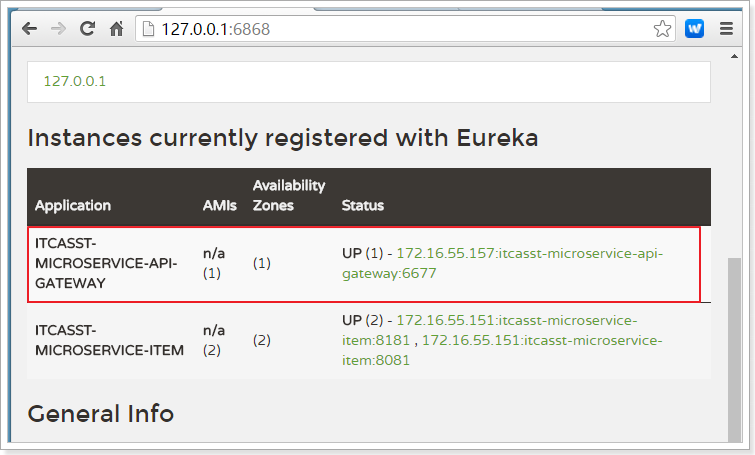
defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

### 启动测试

查看Eureka注册中：



发现已经有itcasst-microservice-api-gateway在注册中心了。

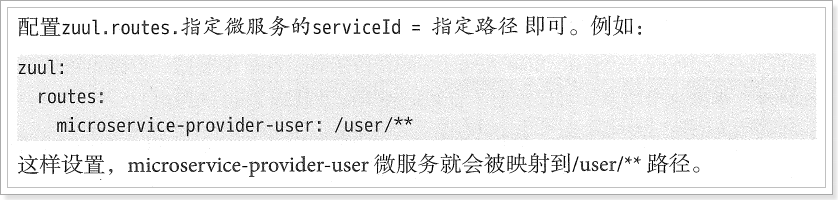
接下来测试，功能是否正常：



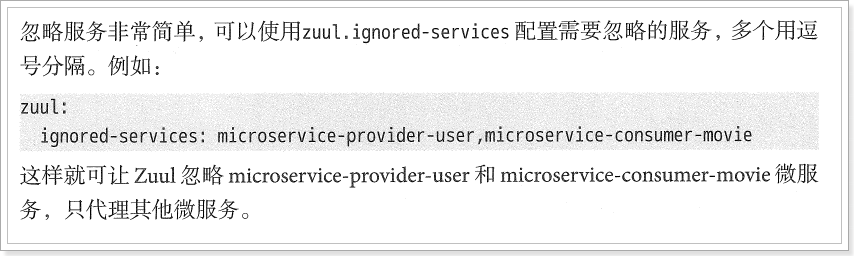
发现一切正常。

## zuul配置详解

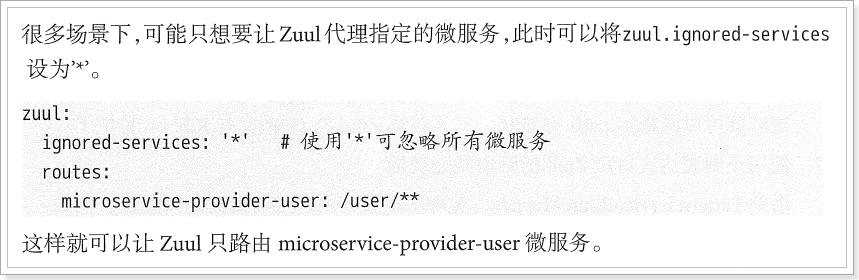
### 指定服务id



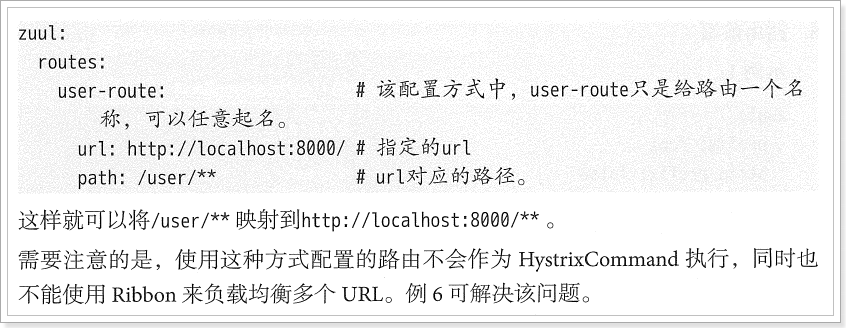
### 忽略指定服务



### 忽略所有服务，只是有路由指定



### 同时配置path和url



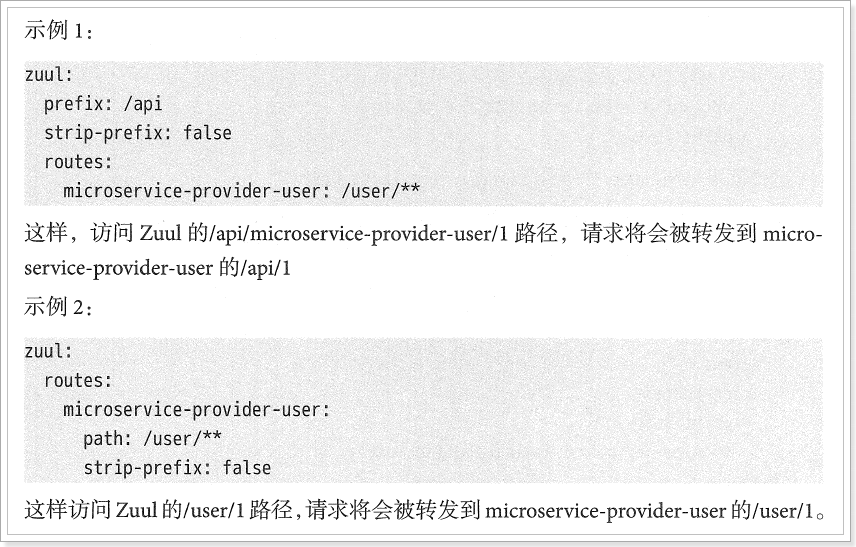
### 面向服务配置，不破坏Hystrix、Ribbon特性



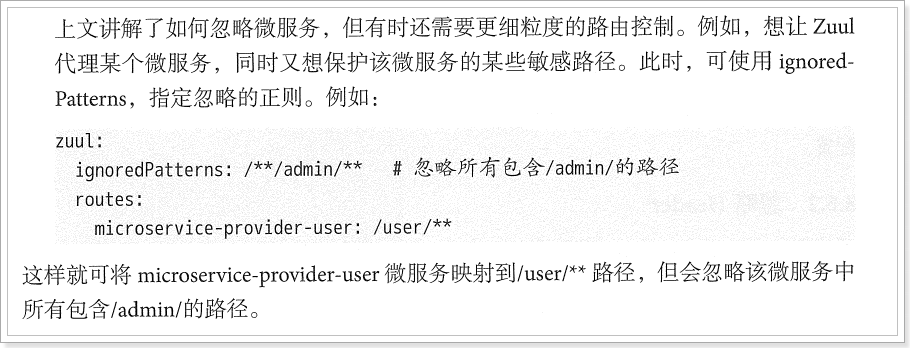
### 使用正则表达式指定路由规则



### 路由前缀



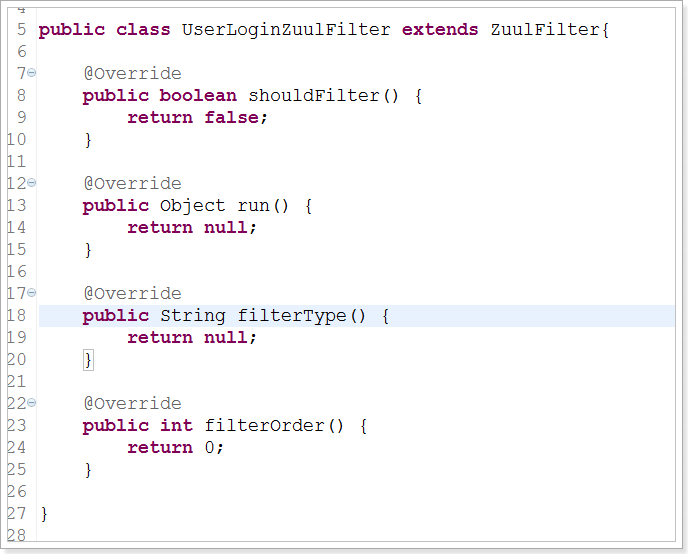
### 忽略某些路径



## 过滤器

过滤器是Zuul的重要组件。

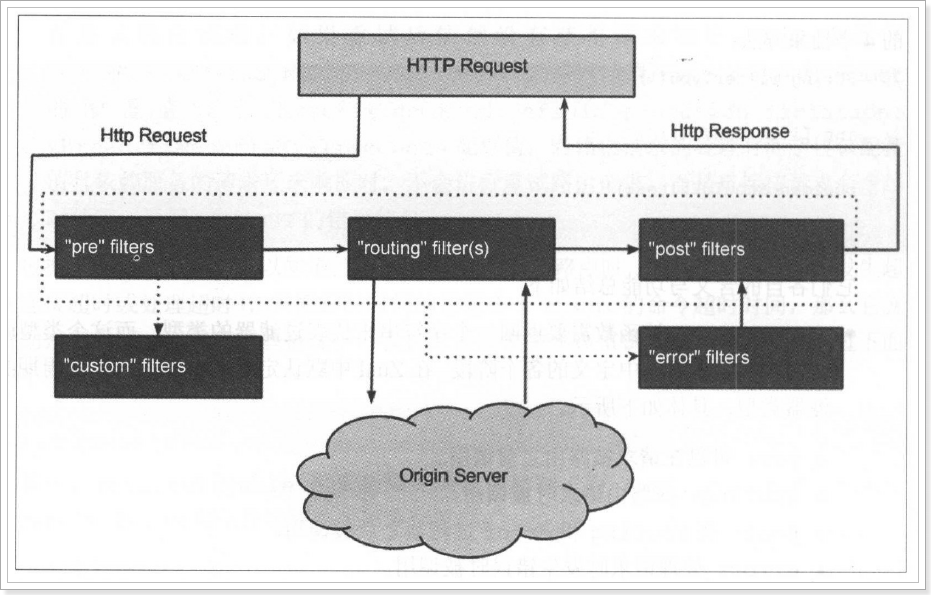
### 过滤器ZuulFilter

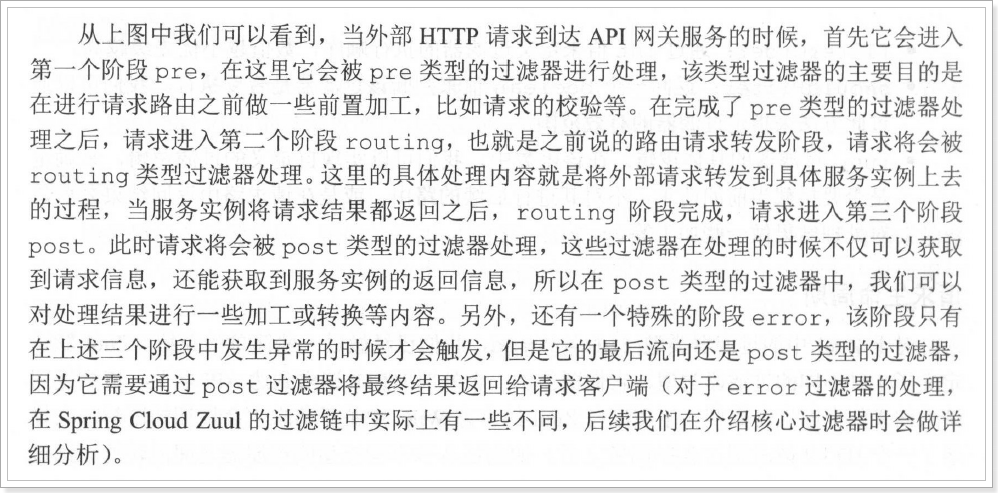


ZuulFilter是一个抽象类，其实现类需要实现4个方法：

1. shouldFilter：返回一个Boolean值，判断该过滤器是否需要执行。返回true执行，返回false不执行。
2. run：过滤器的具体业务逻辑。
3. filterType：返回字符串代表过滤器的类型
   1. pre：请求在被路由之前执行
   2. routing：在路由请求时调用
   3. post：在routing和errror过滤器之后调用
   4. error：处理请求时发生错误调用
4. filterOrder：通过返回的int值来定义过滤器的执行顺序，数字越小优先级越高。

### 执行流程





## 过滤器实战

需求：通过编写过滤器实现用户是否登录的检查。

实现：通过判断请求中是否有token，如果有认为就是已经登录的，如果没有就认为是非法请求，响应401.

### 编写UserLoginZuulFilter

**package** cn.itcast.microservice.filters;

**import** javax.servlet.http.HttpServletRequest;

**import** org.apache.commons.lang3.StringUtils;

**import** org.springframework.stereotype.Component;

**import** com.netflix.zuul.ZuulFilter;

**import** com.netflix.zuul.context.RequestContext;

@Component //加入到Spring容器

**public** **class** UserLoginZuulFilter **extends** ZuulFilter{

@Override

**public** **boolean** shouldFilter() {

**return** **true**; // 该过滤器需要执行

}

@Override

**public** Object run() { //编写业务逻辑

RequestContext requestContext = RequestContext.*getCurrentContext*();

HttpServletRequest request = requestContext.getRequest();

String token = request.getParameter("token");

**if**(StringUtils.*isEmpty*(token)){

requestContext.setSendZuulResponse(**false**); // 过滤该请求，不对其进行路由

requestContext.setResponseStatusCode(401); // 设置响应状态码

**return** **null**;

}

**return** **null**;

}

@Override

**public** String filterType() {

**return** "pre"; // 设置过滤器类型为：pre

}

@Override

**public** **int** filterOrder() {

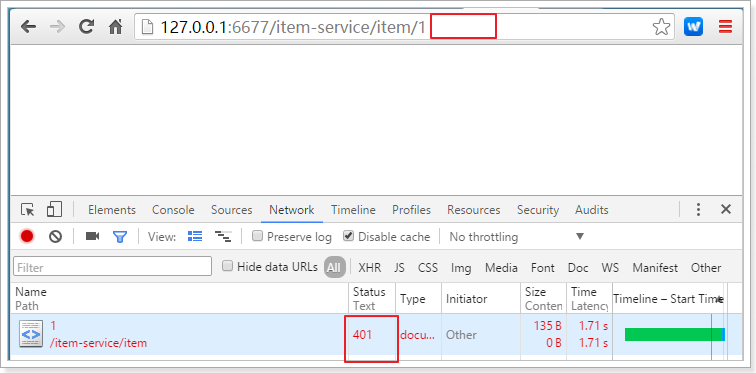
**return** 0;// 设置执行顺序

}

}

### 启动测试





可以看到过滤器已经生效。

# 使用Spring Cloud Config统一管理微服务的配置

## 之前的配置文件用法存在什么问题？

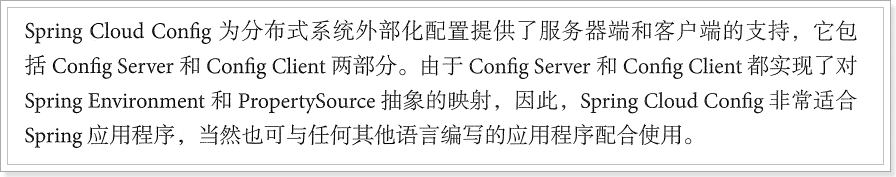
在我们开发项目时，需要有很多的配置项需要写在配置文件中，如：数据库的连接信息等。

这样看似没有问题，但是我们想一想，如果我们的项目已经启动运行，那么数据库服务器的ip地址发生了改变，我们该怎么办？

如果真是这样，我们的应用需要重新修改配置文件，然后重新启动，如果应用数量庞大，那么这个维护成本就太大了！

有没有好的办法解决呢？当然是有的，Spring Cloud Config提供了这样的功能，可以让我们统一管理配置文件，以及实时同步更新，并不需要重新启动应用程序。

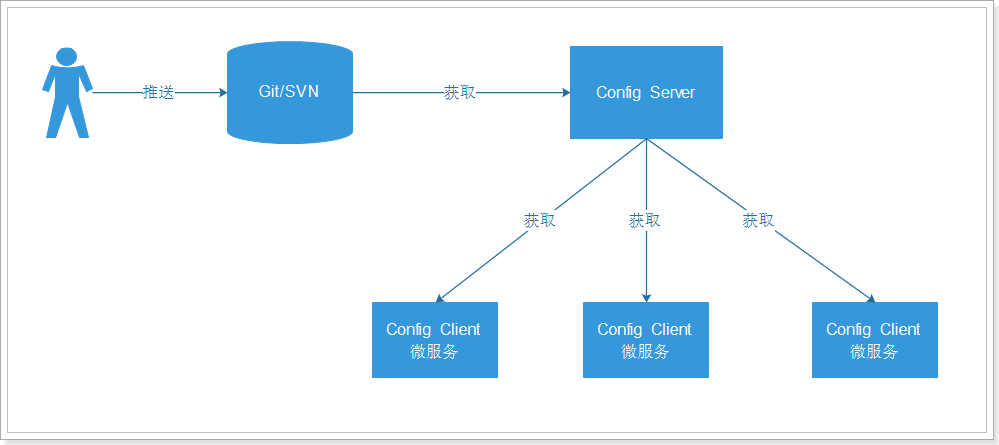
## Spring Cloud Config 简介



Config Server是一个可横向扩展、集中式的配置服务器，它用于集中管理应用程序各个环境下的配置，默认使用Git存储配置文件内容，也可以使用SVN存储，或者是本地文件存储。

Config Client是Config Server的客户端，用于操作存储在Config Server中的配置内容。微服务在启动时会请求Config Server获取配置文件的内容，请求到后再启动容器。

使用Spring Cloud Config的架构：



## 快速入门 – 搭建Config Server

### 准备3个配置文件，推送到Git服务器

准备3个文件：

microservice-dev.properties

microservice-production.properties

microservice-test.properties

该文件的命名规则是：{application}-{profile}.properties

其内容是（另外2个文件内容稍有不同即可）：

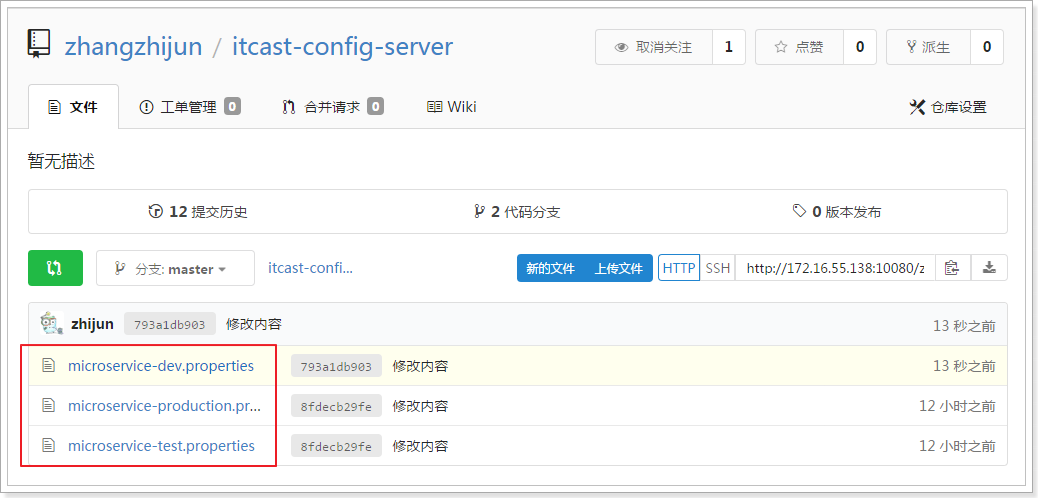
jdbc.driverClassName=com.mysql.jdbc.Driver

jdbc.url=jdbc:mysql://127.0.0.1:3306/taotao?useUnicode=true&characterEncoding=utf8&autoReconnect=true&allowMultiQueries=true

jdbc.username=root

jdbc.password=root

推送文件到git服务器，这里使用的是我们内网的git服务器（Gogs），当然也可以使用github或者使用svn。



### 创建工程itcast-microservice-config-server

pom依赖：

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>cn.itcast.microservice.config</groupId>

<artifactId>itcast-microservice-config-server</artifactId>

<version>1.0.0-SNAPSHOT</version>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

</parent>

<!-- 导入Spring Cloud的依赖管理 -->

<dependencyManagement>

<dependencies>

<dependency>

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

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR3</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<!-- 资源文件拷贝插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-resources-plugin</artifactId>

<configuration>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

<!-- java编译插件 -->

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<configuration>

<source>1.8</source>

<target>1.8</target>

<encoding>UTF-8</encoding>

</configuration>

</plugin>

</plugins>

</build>

</project>

### 编写入口ConfigApplication

**package** cn.itcast.microservice;

**import** org.springframework.boot.SpringApplication;

**import** org.springframework.boot.autoconfigure.SpringBootApplication;

**import** org.springframework.cloud.config.server.EnableConfigServer;

@EnableConfigServer //开启配置服务

@SpringBootApplication

**public** **class** ConfigApplication {

**public** **static** **void** main(String[] args) {

SpringApplication.*run*(ConfigApplication.**class**, args);

}

}

### 编写application.yml

server:

port: 6688 #服务端口

spring:

application:

name: itcasst-microservice-config-server #指定服务名

cloud:

config:

server:

git: #配置git仓库地址

uri: http://172.16.55.138:10080/zhangzhijun/itcast-config-server.git

#username: zhangzhijun

#password: 123456

### 启动测试



测试已经看到了配置文件的内容。

请求配置文件的规则如下：

/{application}/{profile}/[label]

/{application}-{profile}.yml

/{label}/{application}-{profile}.yml

/{application}-{profile}.properties

/{label}/{application}-{profile}.properties

其中{label}是指分支，默认是master。

## 快速入门 – 搭建Config Client

我们在itcast-microservice-item项目中添加Config Client的支持。来读取JDBC的配置文件的内容。

### 导入依赖

<dependency>

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

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

</dependency>

### 创建配置文件bootstrap.yml

spring:

cloud:

config:

name: microservice #对应的配置服务中的应用名称

uri: http://127.0.0.1:6869/

profile: dev #对应配置服务中的{profile}

label: master #对应的分支

### 编写JdbcConfigBean

编写对象通过@Value注解读取Config Server中的值。

**package** cn.itcast.microservice.config;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.stereotype.Component;

@Component//加入到Spring容器

**public** **class** JdbcConfigBean {

@Value("${jdbc.url}")

**private** String url;

@Value("${jdbc.username}")

**private** String username;

@Value("${jdbc.password}")

**private** String password;

@Value("${jdbc.driverClassName}")

**private** String driverClassName;

**public** String getUrl() {

**return** url;

}

**public** **void** setUrl(String url) {

**this**.url = url;

}

**public** String getUsername() {

**return** username;

}

**public** **void** setUsername(String username) {

**this**.username = username;

}

**public** String getPassword() {

**return** password;

}

**public** **void** setPassword(String password) {

**this**.password = password;

}

**public** String getDriverClassName() {

**return** driverClassName;

}

**public** **void** setDriverClassName(String driverClassName) {

**this**.driverClassName = driverClassName;

}

@Override

**public** String toString() {

**return** "JdbcConfigBean [url=" + url + ", username=" + username

+ ", password=" + password + ", driverClassName="

+ driverClassName + "]";

}

}

### 编写测试方法进行测试

在ItemController中编写test方法：

**package** cn.itcast.microservice.controller;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.web.bind.annotation.GetMapping;

**import** org.springframework.web.bind.annotation.PathVariable;

**import** org.springframework.web.bind.annotation.RestController;

**import** cn.itcast.microservice.config.JdbcConfigBean;

**import** cn.itcast.microservice.pojo.Item;

**import** cn.itcast.microservice.service.ItemService;

@RestController

**public** **class** ItemController {

@Autowired

**private** ItemService itemService;

@Autowired

**private** JdbcConfigBean jdbcConfigBean;

/\*\*

\* 对外提供接口服务，查询商品信息

\*

\* **@param** id

\* **@return**

\*/

@GetMapping(value = "item/{id}")

**public** Item queryItemById(@PathVariable("id") Long id) {

**return** **this**.itemService.queryItemById(id);

}

@GetMapping(value = "test")

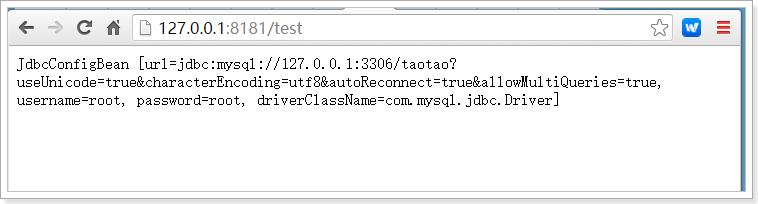
**public** String test(){

**return** **this**.jdbcConfigBean.toString();

}

}

### 启动测试



测试结果显示，已经从Config Server中获取到配置文件的内容。

## 手动更新运行中的配置文件

如果git服务器中的配置文件更新了怎么办？正在运行的应用中的配置内容如何更新？

### 测试

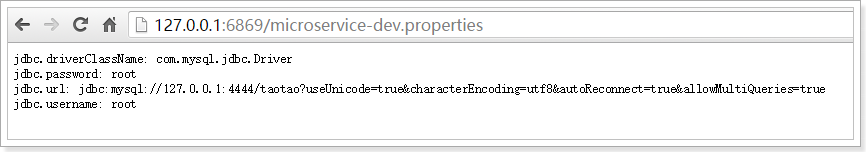
现在我们更改git服务器中的配置文件内容：



修改成4444：

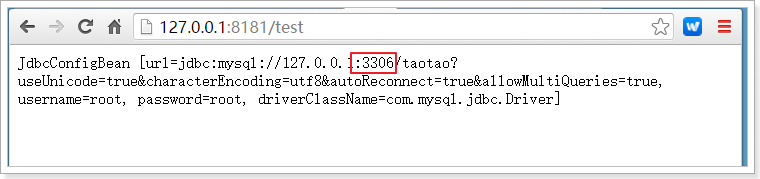


然后刷新Config Server地址观察：



可以看到这里查询到的是新的数据。

在Config Client中测试：



看到依然是旧的数据。

如何才能在重启应用的情况下，获取到最新的配置文件内容呢？ -- 为Config Client添加refresh支持。

### 加入依赖

<dependency>

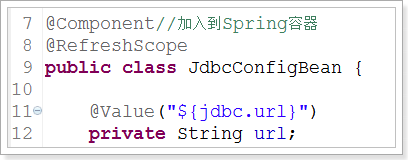
<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

### 为JdbcConfigBean添加@RefreshScope注解

需要为动态更新配置内容的bean添加@RefreshScope注解。



### 修改application.yml文件

server:

port: 8181 #服务端口

spring:

application:

name: itcast-microservice-item #指定服务名

logging:

level:

org.springframework: INFO

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ipAddress: 127.0.0.1

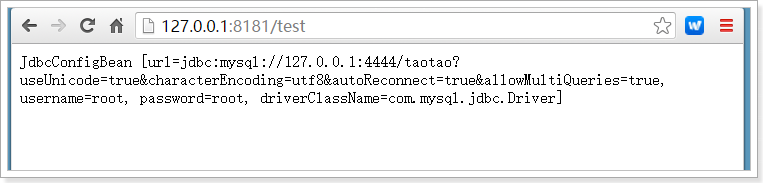
management:

security:

enabled: **false** #是否开启actuator安全认证

### 重启做测试

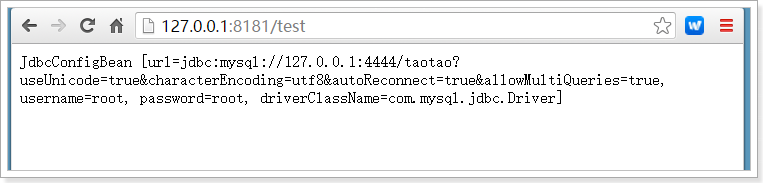
刷新Config Client的地址：



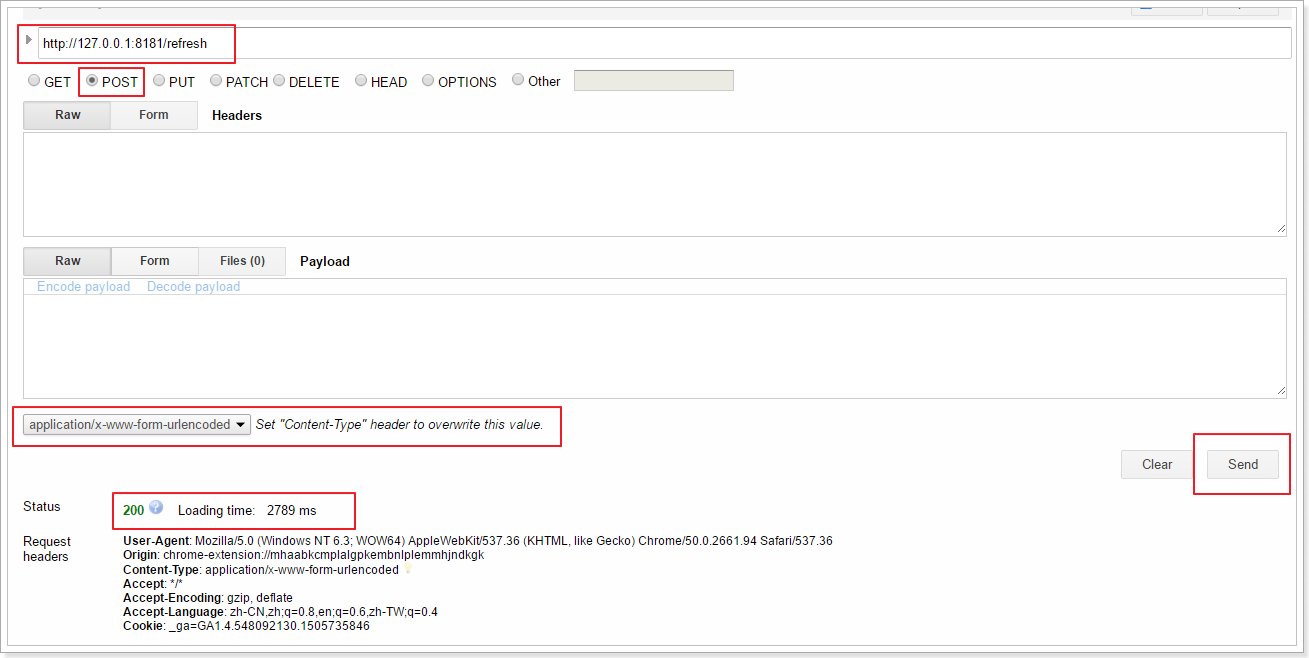
为了测试，需要再次修改配置文件的内容，将原来的端口4444改为5555。



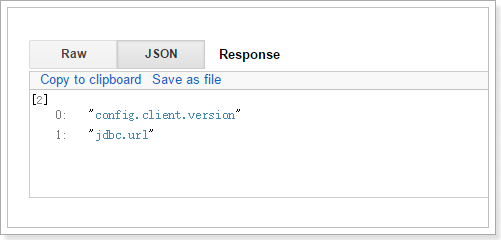
可以看到Config Client中依然是4444：



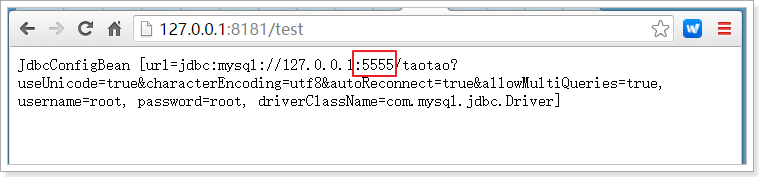
然后，post请求/refresh来更新配置内容：



响应：



可以看到有配置文件内容更新了。



可以看到应该更新了最新的配置文件内容，我们也就实现了在未重启项目的情况下实现了动态修改配置文件内容。

但是，这并不实用，原因是项目已经发布上线了，不可能人为的守在服务器前，发现有更新了就手动请求/refresh.

是否自动完成呢？

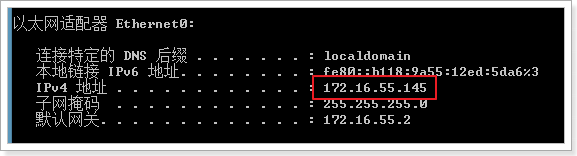
## 借助与git的webhook（web钩子）实现自动更新

gogs、github等git服务器提供了web hook功能，意思是，在仓库中的资源发生更新时会通知给谁，这里的谁是一个url地址。





查看本机ip地址：





点击“添加Web钩子”。



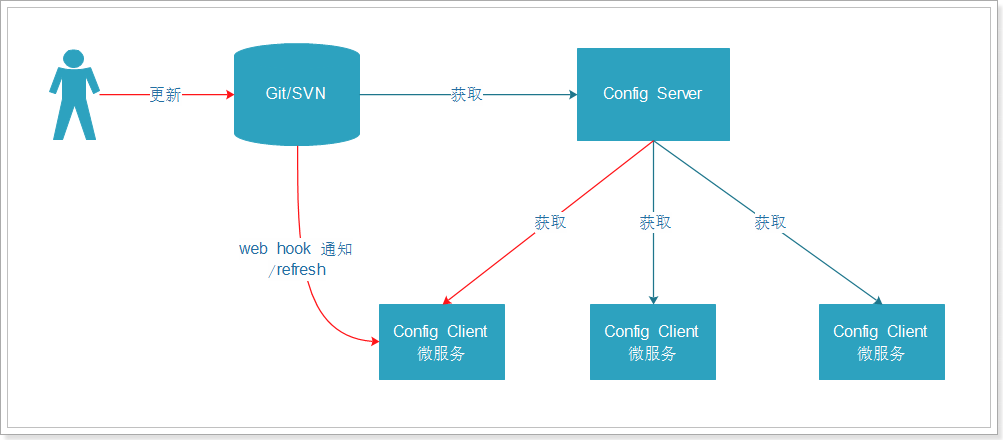
添加完成。

接下来进行测试，更新配置文件的内容。

测试的结果会发现，配置文件内容会动态更到Bean中。



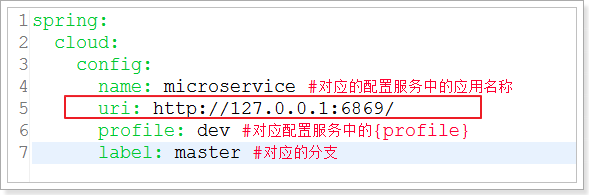
总结下流程：



## Config Client配置的优化

### 分析

在itcast-microservice-item中作为Config Client，在配置文件中了配置了Config Server的地址：



这样的硬编码是不好的，如果配置中心的ip地址发生了改变，那么久需要重新修改并且重启应用了。

想想，有什么好的办法解决呢？ 如果将Config Server作为一个微服务，并且将其注册的Eureka中，是不是就可以不用硬编码了？

### 在itcast-microservice-config-server中添加Eureka的依赖

<!-- 导入Eureka服务的依赖 -->

<dependency>

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

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

<exclusions>

<exclusion>

<groupId>com.fasterxml.jackson.dataformat</groupId>

<artifactId>jackson-dataformat-xml</artifactId>

</exclusion>

</exclusions>

</dependency>

### 修改application.yml文件

server:

port: 6869 #服务端口

spring:

application:

name: itcasst-microservice-config-server #指定服务名

cloud:

config:

server:

git: #配置git仓库地址

uri: http://172.16.55.138:10080/zhangzhijun/itcast-config-server.git

#username: zhangzhijun

#password: 123456

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

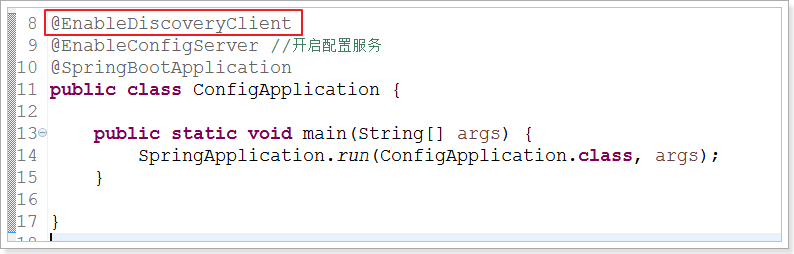
defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

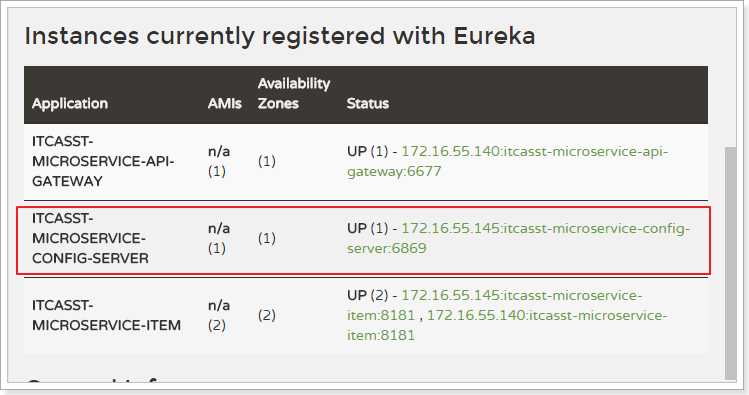
prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

ipAddress: 127.0.0.1

### ConfigApplication添加@EnableDiscoveryClient注解



### 重新启动



可以看到在Eureka中已经有配置中心的服务。

### 在itcast-microservice-item中修改bootstrap.yml配置

eureka:

client:

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

spring:

cloud:

config:

name: microservice #对应的配置服务中的应用名称

#uri: http://127.0.0.1:6869/

profile: dev #对应配置服务中的{profile}

label: master #对应的分支

discovery:

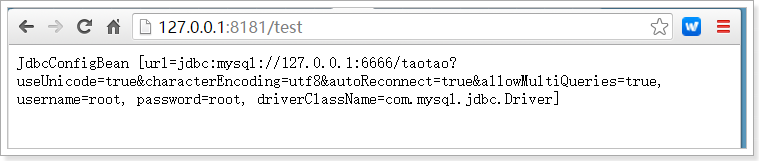
enabled: **true** #启用发现服务功能

service-id: itcasst-microservice-config-server #指定服务名称

疑问：在application.yml中以及配置Eureka的信息，为什么在bootstrap.yml还需要配置？

因为在Spring Boot中bootstrap.yml在application.yml之前加载，所以即使在application.yml中以及配置Eureka的信息，是使用不了的，所以需要在bootstrap.yml中配置Eureka的信息。

### 测试



测试结果，一切正常。这就完美解决了硬编码的问题。

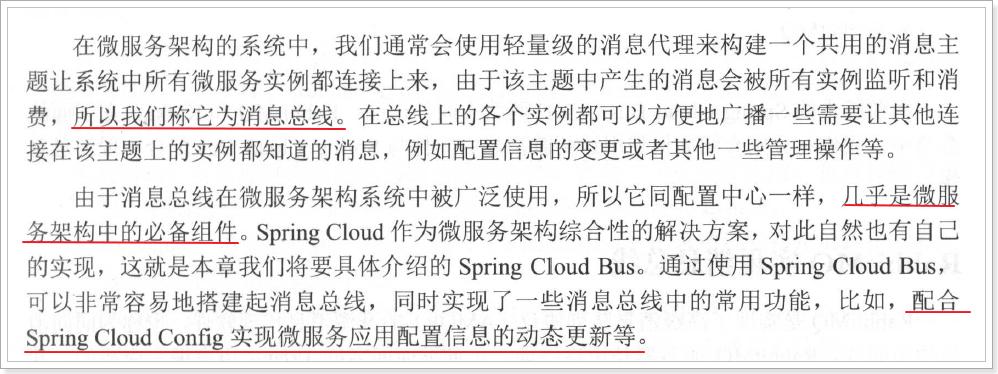
# 使用Spring Cloud Bus（消息总线）实现自动更新

## 分析

虽然通过Gogs Git的web hook可以实现自动更新，但是，如果Config Client有很多的话，那么需要在web hook中维护很多地址，这显然是不现实的做法。

有没有更好的方案呢？ 通过消息实现通知。

## Spring Cloud Bus消息总线的简介

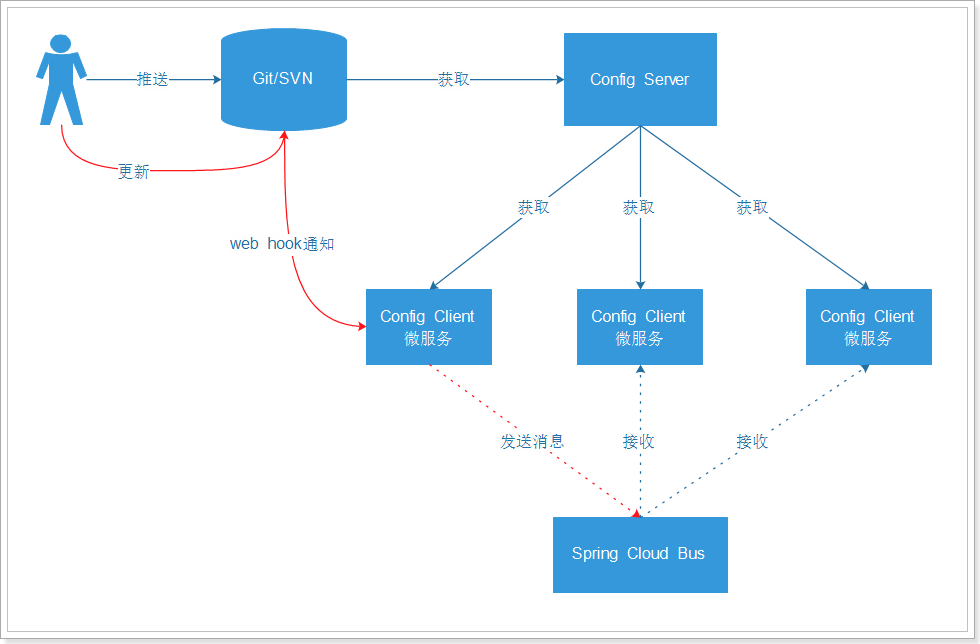


目前Spring Cloud Bus消息总线只是实现了对RabbitMQ以及Kafka的支持。

## RabbitMQ的安装

参考《RabbitMQ-3.4.1安装手册.docx》

## 使用Spring Cloud Bus的架构



## 实现

### 在itcast-microservice-item添加依赖

<dependency>

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

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

### 在bootstrap.yml添加rabbitmq的配置

eureka:

client:

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

spring:

cloud:

config:

name: microservice #对应的配置服务中的应用名称

#uri: http://127.0.0.1:6869/

profile: dev #对应配置服务中的{profile}

label: master #对应的分支

discovery:

enabled: **true** #启用发现服务功能

service-id: itcasst-microservice-config-server #指定服务名称

rabbitmq: #RabbitMQ相关的配置

host: 127.0.0.1

port: 5672

username: guest

password: guest

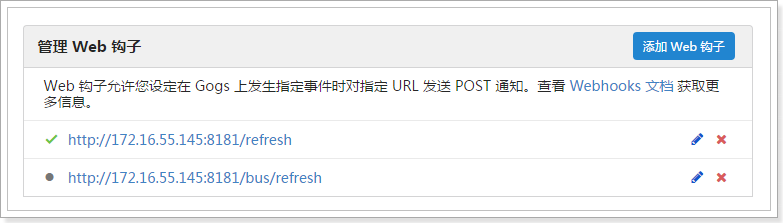
### 修改web hook的地址

在启动后会看到这样的日志：

o.s.b.a.e.mvc.EndpointHandlerMapping : Mapped "{[/bus/refresh],methods=[POST]}" onto public void org.springframework.cloud.bus.endpoint.RefreshBusEndpoint.refresh(java.lang.String)

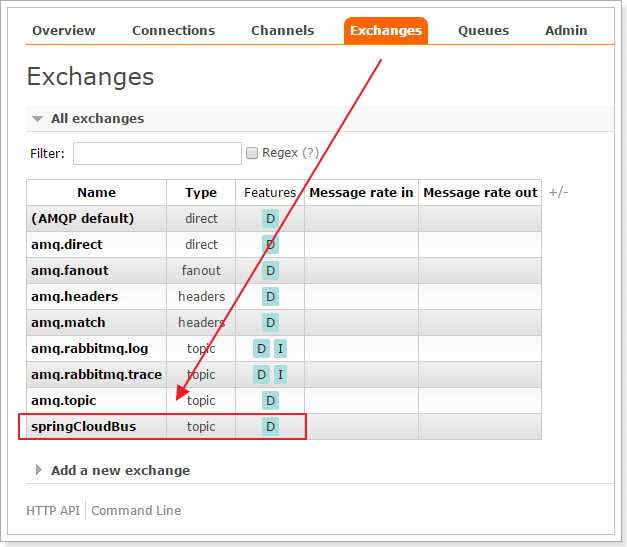
说明刷新的地址/bus/refresh是有Spring Cloud Bus来处理，之前的/refresh依然是由以前的逻辑处理。

所以要修改Gogs中的web hook的地址：（修改或者添加都可以）

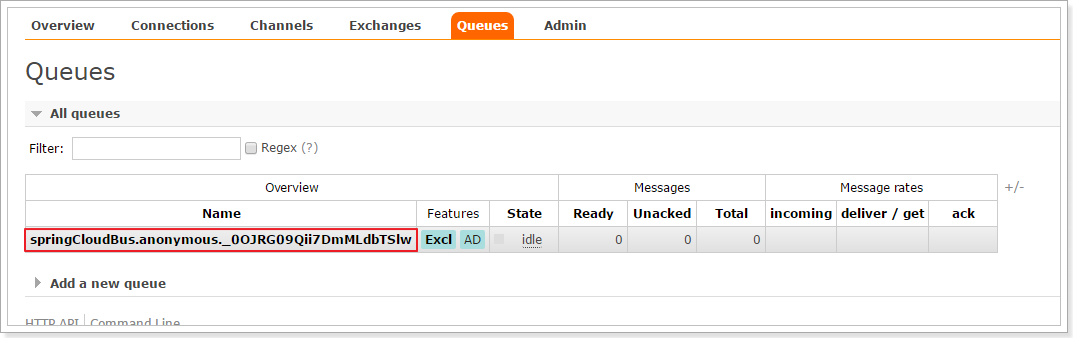


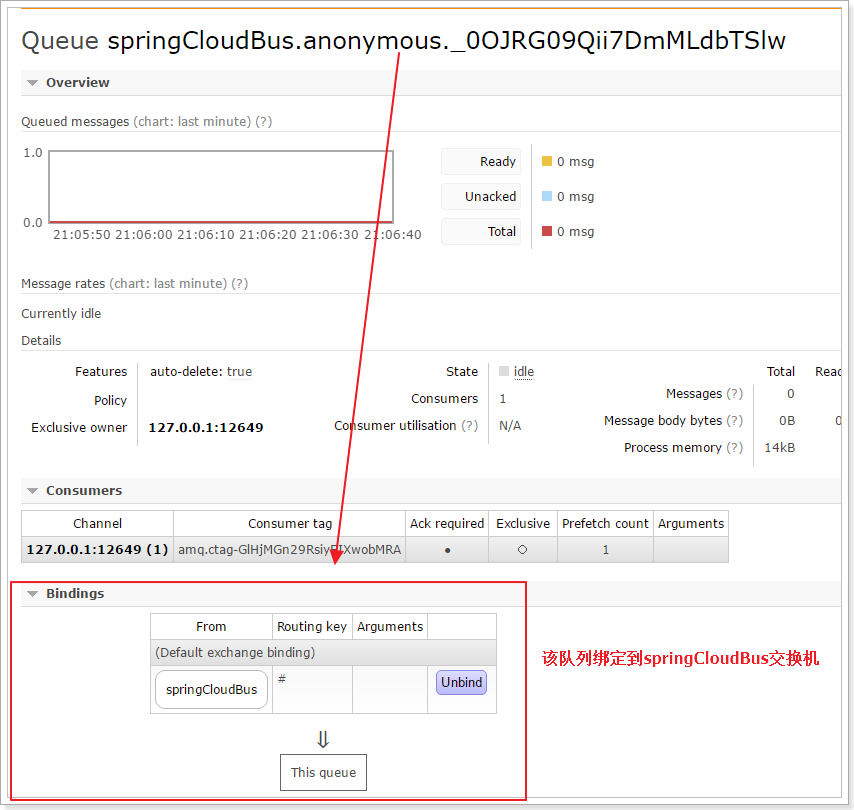
### 启动测试

查看RabbitMQ中的交换机：

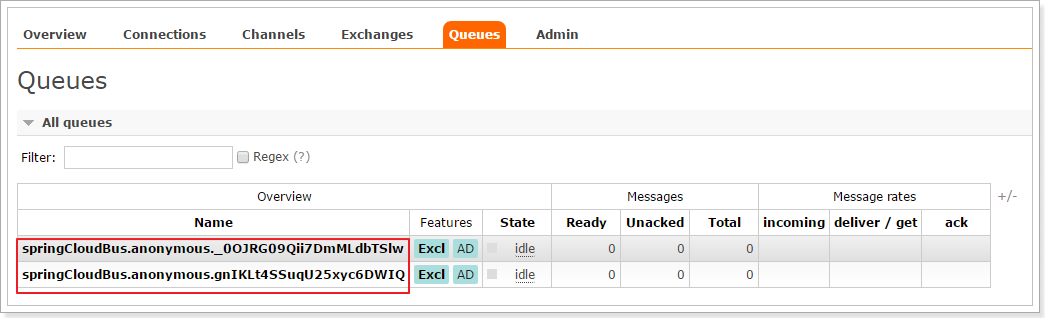


再看队列：



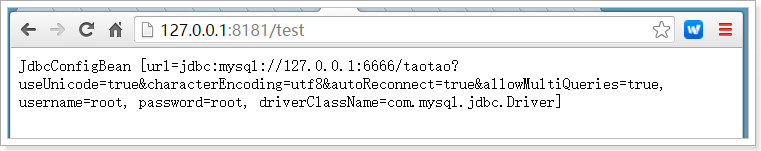


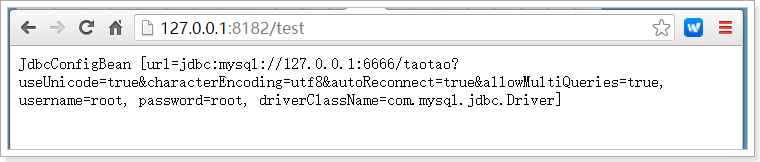
接着，将itcast-microservice-item的端口改成8182，再启动一个itcast-microservice-item实例，进行测试。



发现，有2个队列，分别都绑定到springCloudBus的交换机。

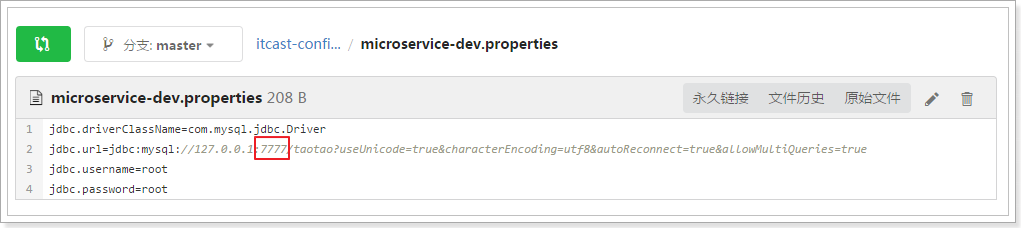
接下里，修改配置文件的内容进行测试。





可以看到8181和8182这2个实例查询到的信息都是一样的。

接下来，修改配置文件内容将6666改成7777：







结果显示，都是获取到最新的数据。

在测试时，会发现，由于Gogs的web钩子推送到8181，所以8181的更新快一些，而8182更新就相对慢一些。

### 流程总结

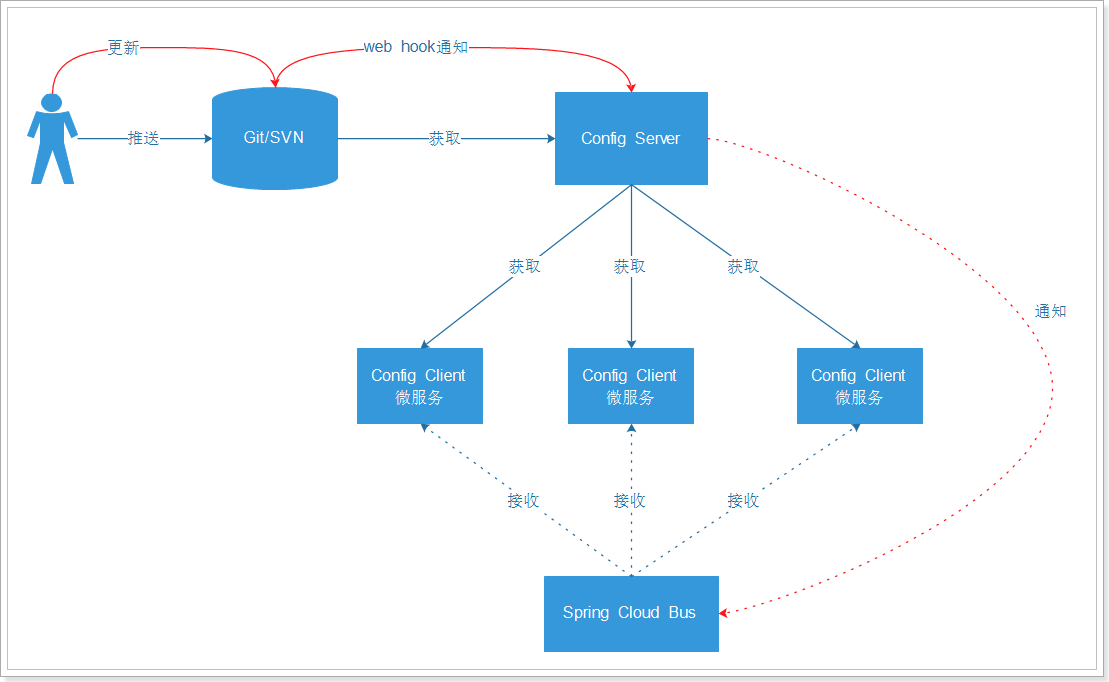
更新文件到Gogs，Gogs通过web钩子通知到8181的/bus/refresh，8181的实例将消息发送到springCloudBus的交换机，由于8181的队列页绑定到交换机，所以8081也获取到了更新的通知，然后去Config Server获取最新的数据。

## 架构优化

在前面实现的架构中，发现8181这个实例不仅仅是提供了商品查询的服务，还负责发送更新的消息到RabbitMQ。

这其实是违反了微服务架构中的职责单一的原则。

其实这个架构是可以改进的，就是将原有的Config Server不仅仅是提供配置查询的服务，而且还要负责更新消息的发送。



### 在itcast-microservice-config-server中导入依赖

<dependency>

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

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

### 修改application.yml配置文件

server:

port: 6869 #服务端口

spring:

application:

name: itcasst-microservice-config-server #指定服务名

cloud:

config:

server:

git: #配置git仓库地址

uri: http://172.16.55.138:10080/zhangzhijun/itcast-config-server.git

#username: zhangzhijun

#password: 123456

rabbitmq: #RabbitMQ相关的配置

host: 127.0.0.1

port: 5672

username: guest

password: guest

eureka:

client:

registerWithEureka: **true** #是否将自己注册到Eureka服务中，默认为true

fetchRegistry: **true** #是否从Eureka中获取注册信息，默认为true

serviceUrl: #Eureka客户端与Eureka服务端进行交互的地址

defaultZone: http://itcast:itcast123@127.0.0.1:6868/eureka/

instance:

prefer-ip-address: **true** #将自己的ip地址注册到Eureka服务中

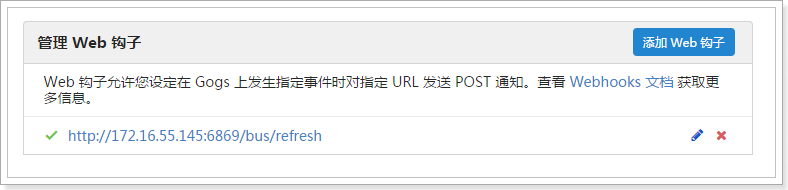
ipAddress: 127.0.0.1

management:

security:

enabled: **false** #是否开启actuator安全认证

### 修改Gogs中的web钩子



### 重启测试

测试结果，和之前一样，可以同步更新到8181和8081。