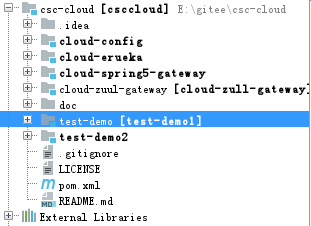
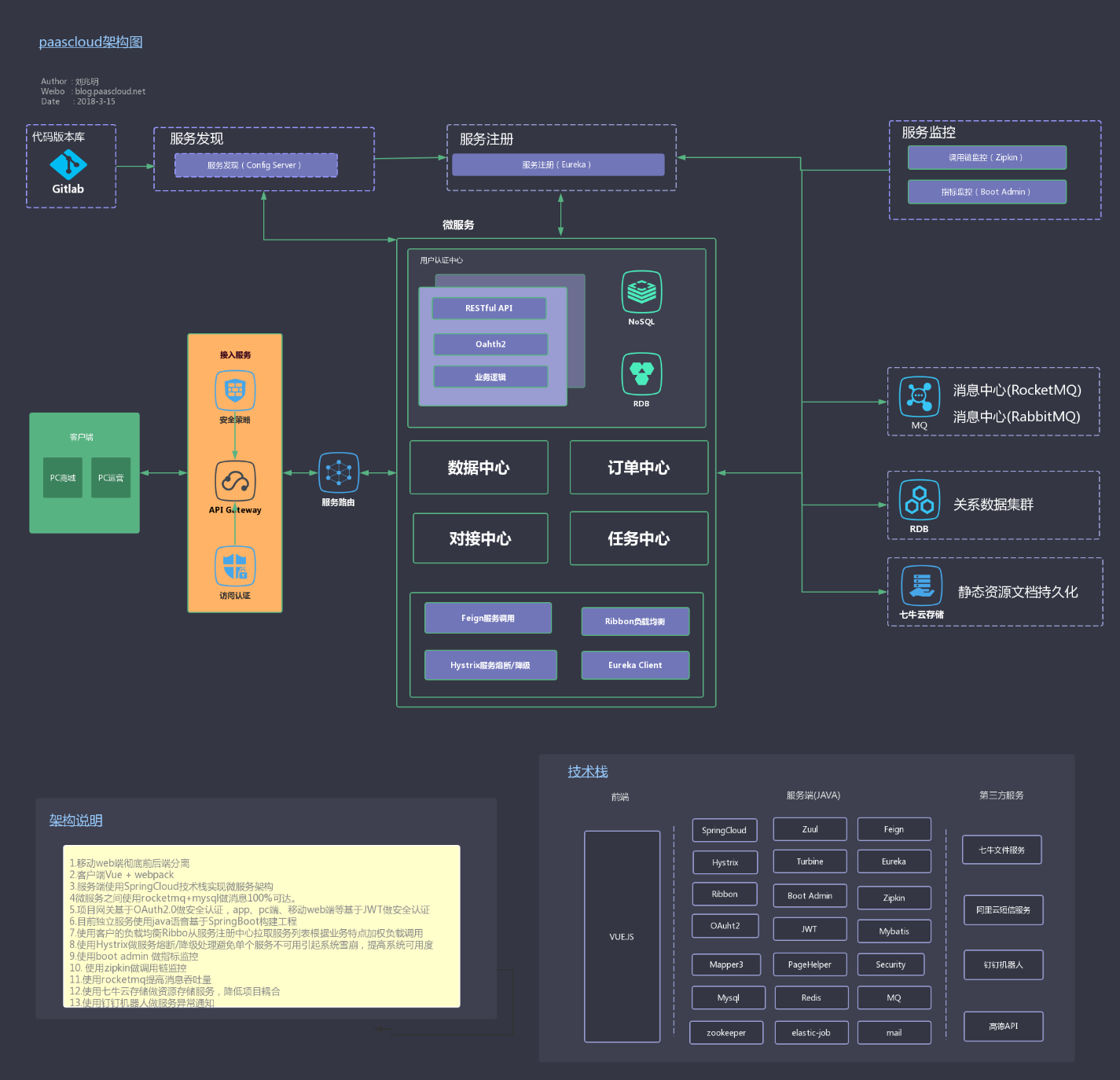
# Csc-cloud 基础架构简述

## 开发环境

* JDK版本:1.8
* 开发工具：idea/eclipse
* 数据库：MySQL 5.7(user:root,password:root)
* Tomcat版本：8.0 (tomcat开发环境可不安装 SpringBoot 内置tomcat项目打包jar包方式运行)
* Redis:3.0（没有设置用户密码）
* rabbitMQ:3.7.7（使用默认用户密码 user:guest,password:guest）

## 项目结构

* 
* Csc-cloud:父项目主要负责管理子项目pom依赖.
* Cloud-erueka:服务注册,发现与治理中心。
* Cloud-config:服务配置中心，主要负责管理配置文件，目前demo-test,demo-test2主配置文件放置在github 上管理。
* Cloud-zuul-gateway:服务网关，主要负责对客户端暴露访问端口，所有客户端请求都由该网关路由到微服务端口。（用户访问权限授权统一在该网关校验后面实现）后期计划升级到spring 5 的gateway 。
* Test-demo1,2:主要是功能实现，暴露接口提供测试。具体功能测试有：redis cache,ribbon组件调用微服务端口，ribbon组件的负载均衡,feign组件显示声明调用微服务，feign组件的熔断，后备，模式测试，SpringCloud Stream 事件驱动的使用。
* Doc:放置项目相关文件，sql脚本，开发文档等。
* 项目参考架构图：



# Csc-cloud 功能简介及实现过程

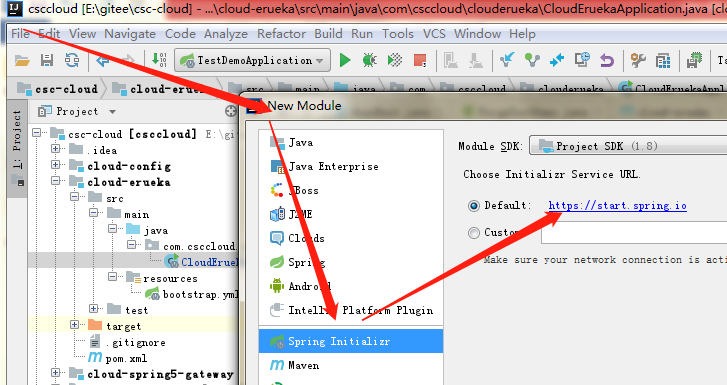
## Cloud-eureka 服务注册中心

### 功能简介

服务注册中心由Eureka server 组件提供实现。其可注册服务并发现治理服务。是分布式微服务架构地基础。

### 实现步骤

1. 新建项目添加pom依赖



使用Maven,SpringBoot构建项目。SpringBoot版本：2.0.0.RELEASE,SpringCloud版本:Finchley.RELEASE。

Pom依赖如下：

<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.0.0.RELEASE</**version**>  
 <**relativePath**/> *<!-- lookup parent from repository -->*</**parent**>  
  
<**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 <**project.reporting.outputEncoding**>UTF-8</**project.reporting.outputEncoding**>  
 <**java.version**>1.8</**java.version**>  
 <**spring-cloud.version**>Finchley.RELEASE</**spring-cloud.version**>  
 *<!--<spring-cloud.version>Finchley.SR1</spring-cloud.version>-->*</**properties**>  
  
<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-server</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 *<!--热部署-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
 <**optional**>true</**optional**>  
 </**dependency**>  
</**dependencies**>  
  
<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>${spring-cloud.version}</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>  
  
<**build**>  
 <**plugins**>  
 <**plugin**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-maven-plugin</**artifactId**>  
 </**plugin**>  
 </**plugins**>  
</**build**>

1. 在主配置类中加入注解声明该应用使用的组件。



其中：@SpringBootApplication为SpringBoot组合注解等价与@Configuration、@EnableAutoConfiguration、@ComponentScan 。其声明该类为配置类，并允许扫描组件，允许使用SpringBoot自动装配功能。

@EnableEurekaServer声明该应用为服务注册治理中心。

1. 编辑配置文件

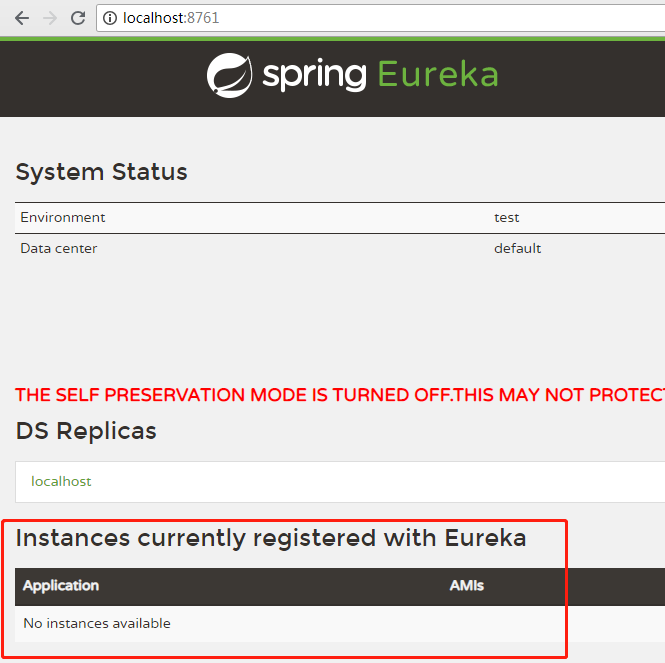
SpringBoot项目配置有YAML,properties两种风格配置文件格式。本项目统一采用YAML其配置就有层次结构方便阅读。



1. 启动项目

Run 主配置类的main()方法则该SpringBoot项目会打包成jar以jar包方式运行。

访问<http://localhost:8761/> 地址可以看到如下页面说明服务注册中心已经启动成功。



红色框中记录注册在注册中心的服务，现在还没有服务注册所以记录为空。

## Cloud-zuul-gateway

### 功能简介

### 实现步骤

1. 新建项目模块并引入pom 依赖。

Pom依赖如下：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-zuul</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-openfeign</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-actuator</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>log4j</**groupId**>  
 <**artifactId**>log4j</**artifactId**>  
 <**version**>1.2.17</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**version**>1.16.20</**version**>  
 <**scope**>provided</**scope**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 *<!--热部署-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
 <**optional**>true</**optional**>  
 </**dependency**>  
</**dependencies**>  
  
<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>${spring-cloud.version}</**version**>  
 *<!--<version>Finchley.M8</version>-->* <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

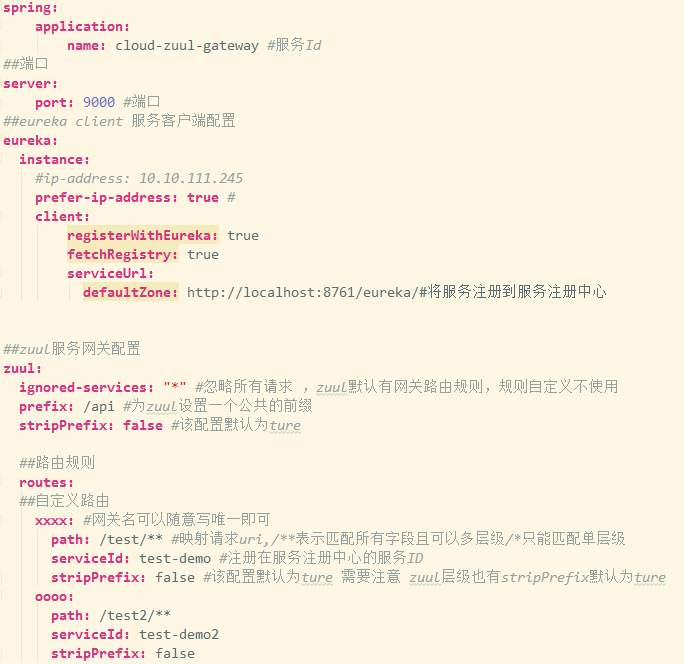
其中spring-cloud-starter-netflix-zuul依赖引入zuul服务网关组件。

spring-cloud-starter-netflix-eureka-client依赖引入eureka服务客户端组件。

1. 在主配置类中加入注解声明该应用使用的组件。



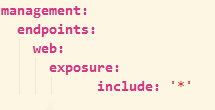
1. 编辑配置文件



1. 启动项目

访问<http://localhost:9000/actuator/routes> 可以看到返回json数据中映射了两个路由规则

即为配置中routes:自定义的两条路由规则。（ps: routes节点为spring-boot-starter-actuator组件所提供该组件默认只有info,health等节点。Routes节点需要配置。另外低版本不需要/actuator路径，高版本需要。否则访问404）

表明actuator组件所有节点都暴露提供使用。



自此zuul路由功能已经实现，后续将补充其过滤器功能主要负责校验数据，记录日志，授权等。后续考虑升级服务网关使用spring 5 gateway ,spring自己开发的路由组件。

actuator

## Cloud-config：服务配置管理中心

### 功能简介

微服务由于拆分了项目，并且微服务可能做了集群。这样带来问题之一就是每个服务都有自己的配置文件配置文件过多并不方便管理。SpringCloud-config即为解决该问题。配置中心将所有的主配置文件集中在Git远程仓库/本地仓库，各服务启动时根据配置对应的配置服务中心获取配置信息。

### 实现步骤

1. 新建项目模块及引入pom依赖

Pom文件如下：

<**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter</**artifactId**>  
 </**dependency**>  
  
 <**dependency**>  
 <**groupId**>org.projectlombok</**groupId**>  
 <**artifactId**>lombok</**artifactId**>  
 <**optional**>true</**optional**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-config</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-config-server</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-eureka-client</**artifactId**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-test</**artifactId**>  
 <**scope**>test</**scope**>  
 </**dependency**>  
 *<!--热部署-->* <**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-devtools</**artifactId**>  
 <**optional**>true</**optional**>  
 </**dependency**>  
</**dependencies**>  
  
<**dependencyManagement**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-dependencies</**artifactId**>  
 <**version**>${spring-cloud.version}</**version**>  
 <**type**>pom</**type**>  
 <**scope**>import</**scope**>  
 </**dependency**>  
 </**dependencies**>  
</**dependencyManagement**>

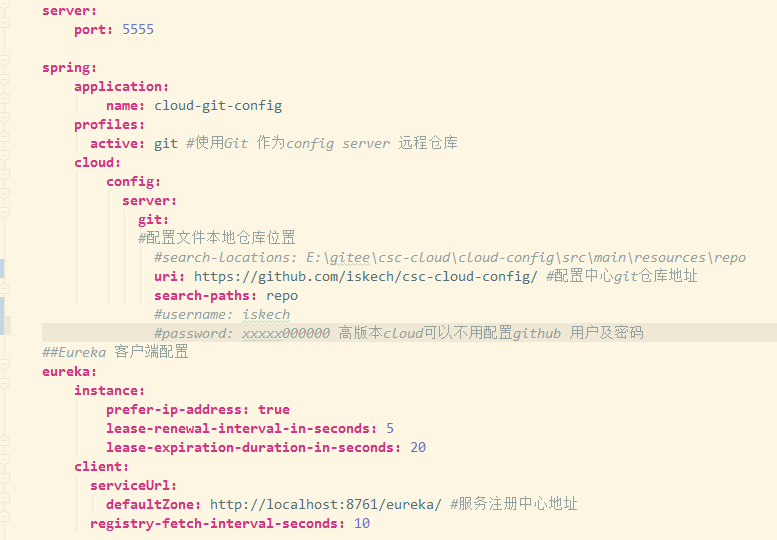
其中spring-cloud-config-server依赖引入服务配置中心服务端。

1. 在主配置类中加入注解声明该应用使用的组件。

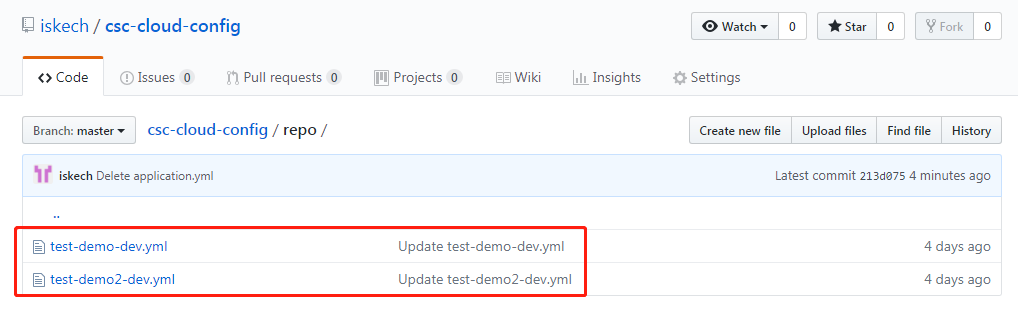


1. 编辑配置文件

配置中心YAML配置：

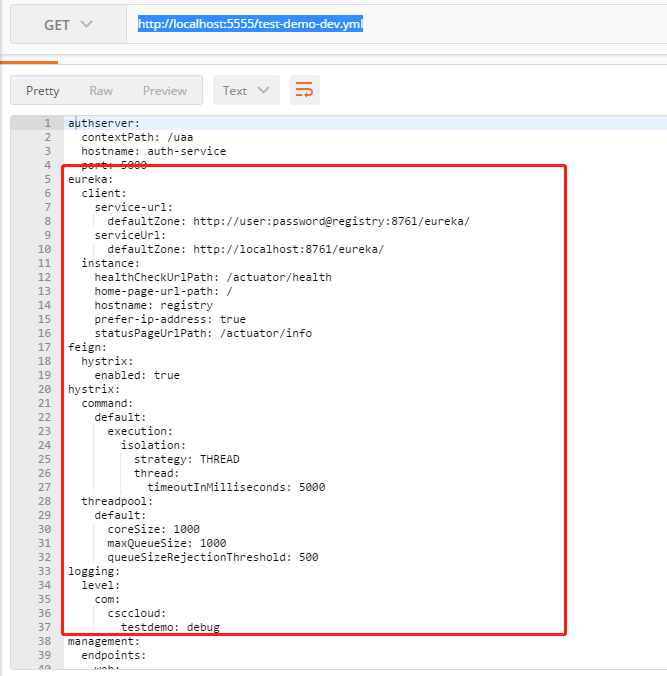


将项目配置文件托管到Git远程仓库：



1. 启动项目

启动项目并访问如下地址：<http://localhost:5555/test-demo-dev.yml> 可以看到配置中心将远程Github,仓库的test-demo-dev.yml配置文件拉取并返回如下信息，成功获取到远程仓库配置文件信息。



## Test-demo 各组件功能的实现

### SpringCLoud config client ，配置中心客户端的实现

##### 功能简介

配置中心客户端基于配置中心服务端实现，配置中心客户端启动项目时会去配置服务中心获取对应的配置信息，如此可以将微服务主要配置都放置在配置中心的仓库进行管理，从而不需要每一个微服务都管理一个主配置文件。

##### 实现步骤

1. 新建test-demo项目模块，引入pom依赖

Pom依赖如下：

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

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

其中spring-cloud-starter-netflix-eureka-client依赖为eureka客户端组件即使用该组件可以将服务注册到服务注册中心。

spring-cloud-config-client依赖为SpringCloud config client 客户端组件，使用该组件可以从配置服务中心获取配置文件。

1. 在主配置类中加入注解声明该应用使用的组件



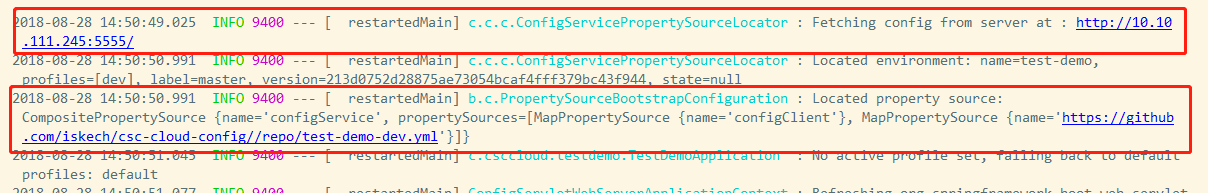
1. 编写配置文件



其中cloud.config.profile:dev 属性表示项目环境为dev-开发环境获取的配置文件也对应的为开发环境，cloud.config.label:master属性表示git项目分支为master主分支即获取到的配置文件都是master分支下的文件。需要注意的是该配置文件应该命名为bootstrap.yml,如此命名的配置文件优先被加载，application.yml随后被加载且其后加载的配置信息会覆盖前面加载的配置信息。后面所有在项目本地的配置文件都将采用bootstrap.yml的命名方式。

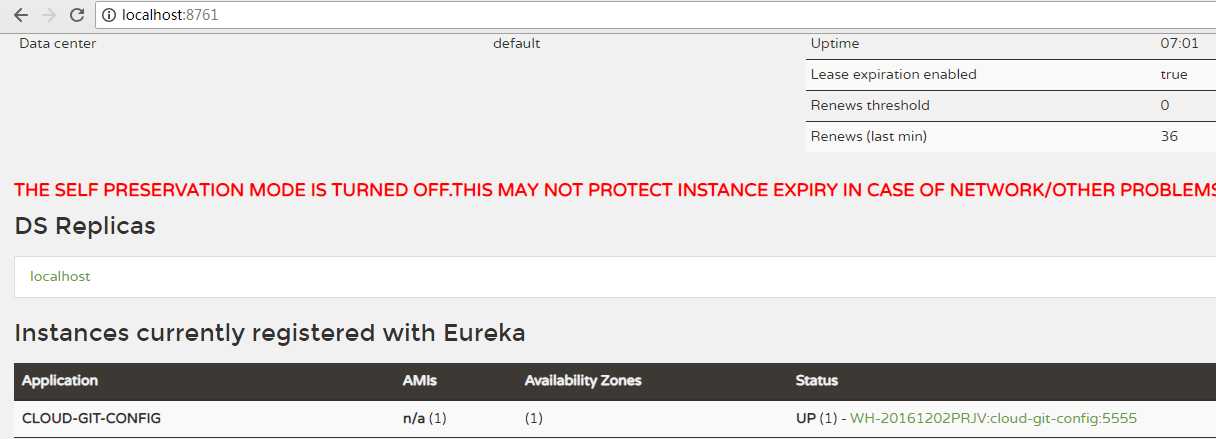
1. 启动项目

Run 主配置类的main方法启动项目，观察控制台输出信息：



可以看到项目尝试从配置中心地址获取配置信息，并从远程仓库github拉取到了test-demo-dev.yml YAML配置文件。

访问<http://localhost:8761/> 地址：



可以看到cloud-config服务已经注册到服务注册中心cloud-git-config 为其serviceId。

### 使用Ribbon及RestTemplate实现微服务之间的调用

#### 功能介绍

微服务概念是将一个项目拆分为多个子项目，带来的另外一个问题就是各子项目怎么互相调用彼此的接口获取数据。SpringCloud整合了Ribbon 结合RestTemplate可实现微服务之间的通讯。

#### 实现步骤

1. 前面我们已经创建了一个test-demo项目，现在我们需要创建另外一个test-demo2项目作为另一个微服务项目。升级两个项目为web项目使其具备暴露接口的能力。

test-demo ，test-demo2 Pom依赖文件如下：

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

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

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>

相对之前test-demo 项目添加了spring-boot-starter-web 依赖组件使之成为web项目。

1. 在主配置类中加入注解声明该应用使用的组件



1. 编辑配置文件



其中datesource即数据库连接配置需要改成本地环境，否则启动项目会报数据库连接失败的错误。demo-test2项目配置与demo-test配置类似就不再贴出。

1. 主要代码编写及测试

此处省略mybatis sql 语句，mapper,service,entity层代码编写。……

test-demo代码如下：



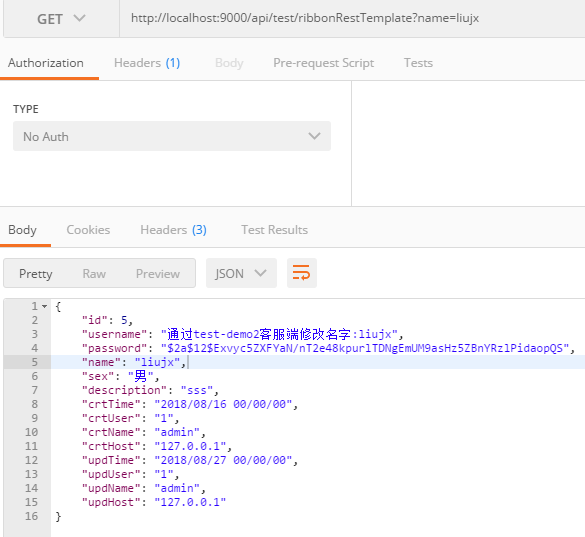
test-demo2代码如下：



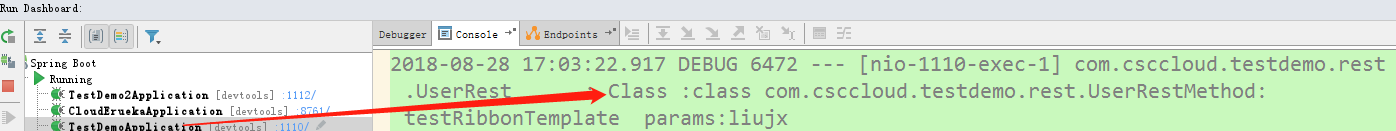
在UserRest controller中使用注入的restTemplate对象调用其getForEnty(“uri”,responseType.Class,params)方法。即可实现调用其它服务暴露的接口。

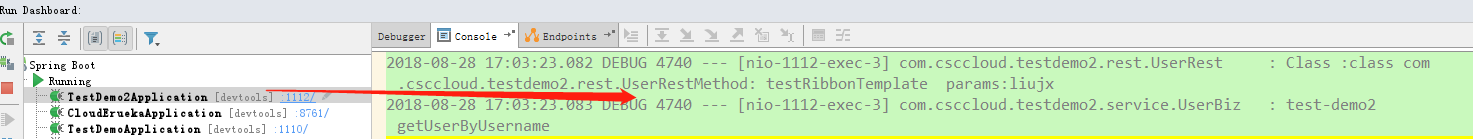
其中uri结构为：https:/serviceId/path/?{key}=”keyname”。SeviceId即为test-demo2服务注册在eureka服务中心的serviceId:test-demo2,path为test-demo2 中controller方法mapping映射路径，｛key｝={keyname}即为参数名，参数可以使用map进行封装其key对应keyname,value即为参数值。

在postman 中请求<http://localhost:9000/api/test/ribbonRestTemplate?name=liujx> 地址结果如下：



观察test-demo，test-demo2项目控制台信息如下：





可以看到访问test-demo项目的/api/test/ribbonRestTemplate接口成功，test-demo2的/api/test2/ribbonRestTemplate也成功被调用。相对直接使用RestTemplate调用远程接口，该方式的优势是url参数不需要写死服务主机ip及其服务端口，只需要使用注册在eureka服务中心的serviceId代替。这种方式可以做到服务集群而不需要更改代码。

### 使用Feign组件声明式微服务之间的调用

#### 功能简介

上面介绍了使用Ribbon组件结合RestTemplate实现微服务之间的调用。但该方式存在问题：如果调用远程接口参数较长则上面的方式实现代码就变得较为繁琐不易维护。SpringCloud

整合了netflix的Feign组件。Feign是声明式，模板化的Http客户端，使用SpringCloud整合的Feign组件可以非常简便的调用Http API。

#### 实现步骤

1. 在test-demo项目中新增Feign组件 pom依赖。

Pom文件如下：

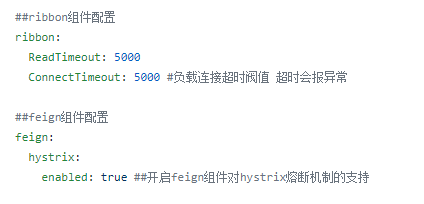
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-openfeign</**artifactId**>  
</**dependency**>

需要注意的是，在Finchley.RELEASE版本中依赖为spring-cloud-starter-openfeign而之前的低版本为spring-cloud-starter-feign，本项目使用SpringCloud Finchley.RELEASE版本。

1. 在主配置类中加入注解声明该应用使用的组件



1. 编辑配置文件



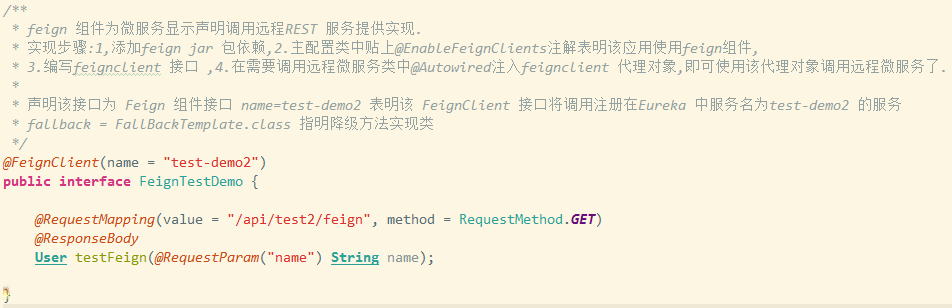
Feign组件若需要开启熔断机制支持则需要如上配置。

1. 代码编写及测试

Test-demo controller代码：



Feign组件接口代码：

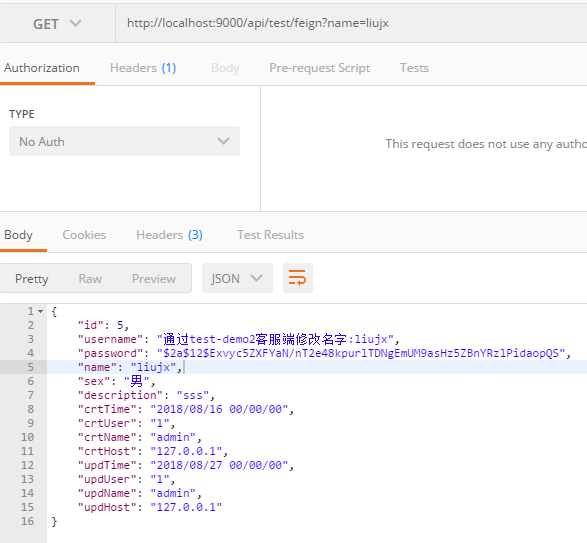


Test-demo2 controller 代码：

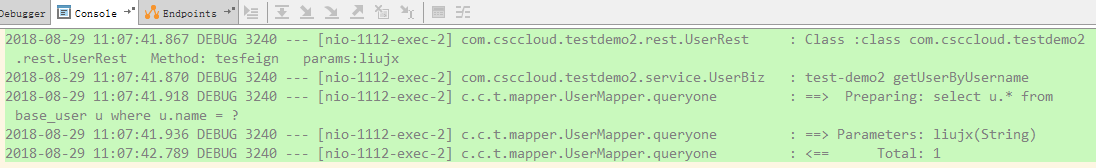


代码解释：Feign组件需要编写feignclient 接口，使用类级别注解@FeignClient(name=”test-demo2”)声明该接口为FeignClient接口，其属性name=”test-demo2” 用来创建Ribbon负载均衡器，会根据该属性值去找到Eureka服务注册中心注册的服务serviceId。通过该属性即可找到需要调用RESTful API 所在的微服务应用。同时SpringCloud Feign 组件默认契约是SpringMVCContract因此可以使用MVC的注解。上面代码中RequesMapping即为MVC注解映射访问路径。@RequestParam，@ResponseBody也是MVC注解，其功能就不一一解释了。现在就可以在test-demo中使用编写的FeignClient接口代理对象去调用test-demo2中的接口了。可以发现FeignClient接口的编写及使用和SpringMVC 的Controller非常类似，所以使用Feign组件调用RESTful API 是非常方便的。

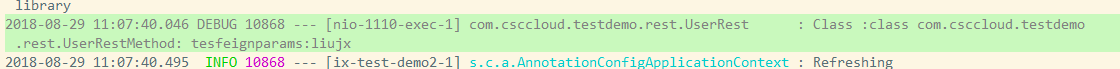
测试：<http://localhost:9000/api/test/feign?name=liujx>

查看test-demo1,2 项目输出台，打印日志显示 api/test/feign；api/test2/fiegn接口都被调用。即说明使用在服务demo-test中使用feign组件调用demo-test2 服务接口成功。

Test-demo2 控制台：



Test-demo 控制台：



### 使用Hystrix组件实现熔断机制

#### 功能简介

微服务将单一系统拆分为多服务带来另外一个问题就是容错。分布式服务较多，不可控因素也就更多。若出现某一服务响应不佳响应时间延迟。Hystrix组件熔断机制为当服务异常达到熔断阀值则进入熔断模式，在熔断模式下每隔一定时间尝试访问异常服务若成功则退出熔断模式，反之继续保持熔断模式。可以很灵活的应对微服务异常增强分布式环境的容错能力。

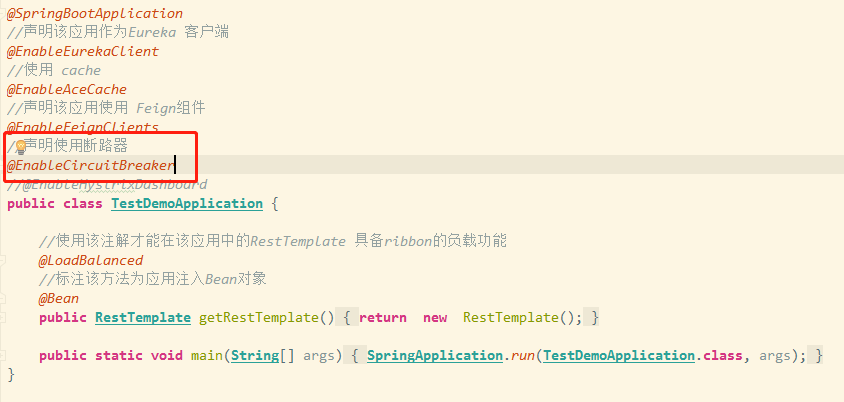
#### 实现步骤

1. 在demo-test,demo-test2项目基础上引入Hystrix组件pom文件依赖

Pom文件如下：

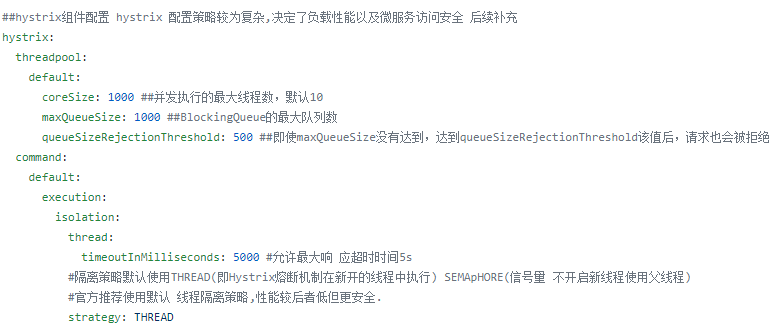
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-netflix-hystrix</**artifactId**>  
</**dependency**>

1. 在主配置类中加入注解声明该应用使用的组件



其中@EnableCircuitBreaker注解表明该应用使用Hystrix熔断器，使用@EnableHystrix注解效果一样。

1. 编辑配置



Hystrix组件配置策略较为复杂，目前使用默认隔离策略ThREAD即线程级别策略。

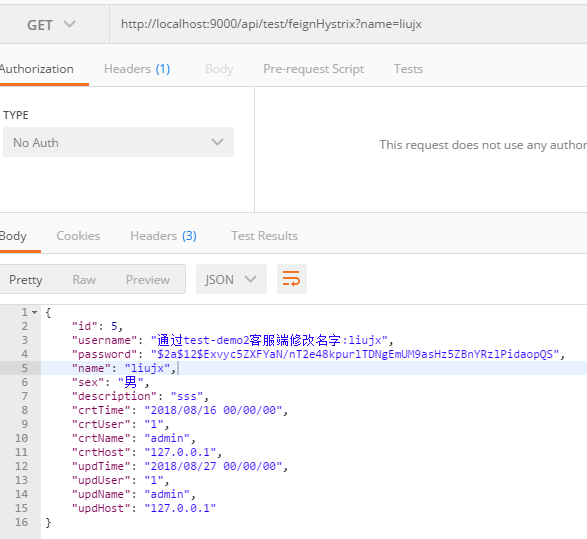
1. 代码编写及测试

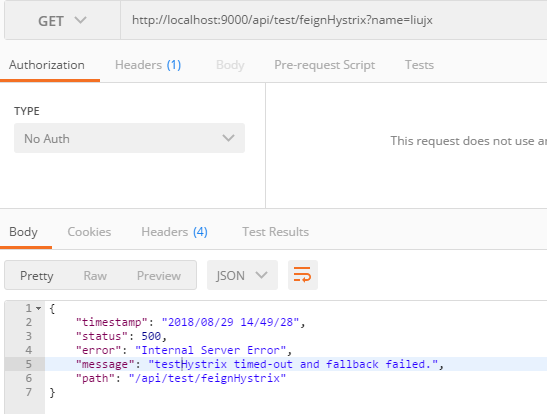


如上图所示方法级别注解@HystrixCommand表明被贴注解方法使用Hystrix组件熔断机制。@HystixProperty属性设置了方法响应超时时间阀值为1100毫秒（默认为1000毫秒），代码模拟了1/2的概率会线程睡眠1111毫秒即会触发熔断机制阀值进入Hystrix熔断模式。快速失败异常的接口调用。抑制因为分布式系统中单个服务响应不佳而”雪崩效应”耗尽整个系统资源。

测试：<http://localhost:9000/api/test/feignHystrix?name=liujx>

测试结果预期如下：





有一半的概率该接口有效的返回user数据，有一半的概率显示message: testHystrix timed-out and fallback failed ,被调用方法超时。如果没有熔断机制那就应该是访问该接口进入线程等待后再响应数据。若线程等待设置的时间为50s呢响应时间也得推迟50s如此若单个服务表现不佳不能及时响应数据，进入响应时间等待势必会影响用户体验甚至引起分布式环境“雪崩效应”。而有Hystrix熔断机制当响应时间超出设置的阀值时迅速进入Hystrix熔断机制达到“快速失败”的效果提高了分布式系统的容错能力。

### 使用Feign组件的fallback后备模式

#### 功能简介

上面介绍了Hystrix的熔断机制基于@HystrixCommand注解实现。但该方式只是当服务异常“快速失败”响应了错误信息给调用者，并不适用所有场景。若需要当服务异常时返回预定义的静态数据响应给调用者呢。这就需要Feign组件的fallback后备模式来实现。当feign组件调用服务异常时进入后备模式调用指定的业务代码返回我们预先定义的数据响应。

#### 实现步骤

1. 在test-demo项目中已经有hystrix,feign组件了，而feign组件的后备模式是基于这两组件实现的所以不需要新增pom依赖。
2. 在test-demo主配置类中已经声明使用Feign,Hystrix组件，不需要额外的组件。
3. 配置文件也无需更新
4. 代码编写及测试

FeignClient代码：



FeignClient接口实现类代码：



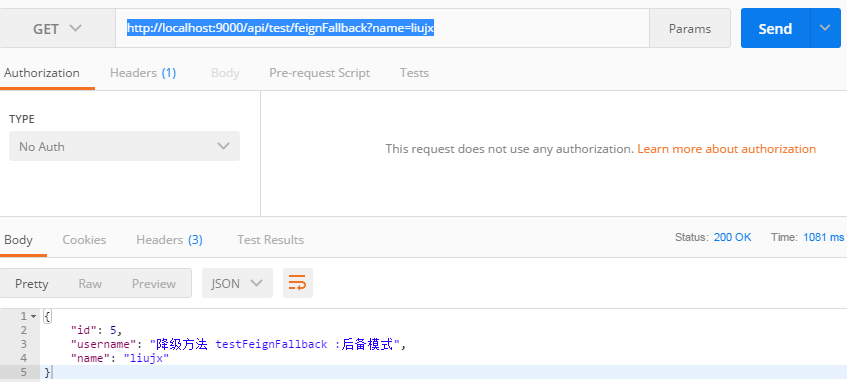
Test-demo2 模拟服务异常：



代码解释：FeignClient接口新增fallback=FallBackTempldate.class属性，其中FallBackTemplate是FeignClient接口实现类。FeignClient接口testFeignFallBack声明在调用服务test-demo2暴露的一个模拟异常的接口。FallBackTemplate实现了FeignClient接口的方法当远程服务调用异常就会调用该类实现的方法。注意FallBackTemplate需要贴Spring注解@service或者@component将其实例创建交由spring。

测试：<http://localhost:9000/api/test/feignFallback?name=liujx>

测试结果预期：



可以看到测试结果并没有报错，而是返回了我们在FallBackTemplate中实现方法的预先定义的数据。Feign组件的后备模式进一步增强细化分布式环境下系统的容错能力。

### 使用SpringCloud Stream实现事件驱动即消息机制的封装

#### 功能简介

SpringCloud Stream事件驱动是对消息机制的进一步封装。在此之前Spring已经集成了对主流消息组件的支持（activeMQ,RockeMQ,rabbitMQ,Kafka…）,简化了消息机制的开发。SpringCloud Stream事件驱动则进一步简化消息机制的开发及其耦合可以做到切换系统消息组件不修改代码，只需要修改配置对新消息组件的支持。目前支持rabbitMQ,kafka。

#### 实现步骤

1. Test-demo，test-demo2项目中新增springcloud stream 组件pom依赖

新增pom文件如下：

<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-stream</**artifactId**>  
</**dependency**>  
<**dependency**>  
 <**groupId**>org.springframework.cloud</**groupId**>  
 <**artifactId**>spring-cloud-starter-stream-rabbit</**artifactId**>  
</**dependency**>

其中spring-cloud-stream组件为事件驱动组件，spring-cloud-starter-stream-rabbit为支持的具体消息组件rabbitMQ。也可以换成kafka。

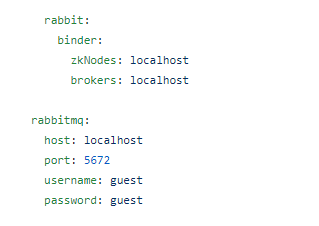
1. 在主配置类中加入注解声明该应用使用的组件



其中@EnableBinding注解表明绑定了三个接口的消息通道，DemoProcessor为自定义接口。Springcloud stream组件通过绑定器Binder和消息组件rabbitMQ/kafka的不同通道信息做交互。

1. 编辑配置





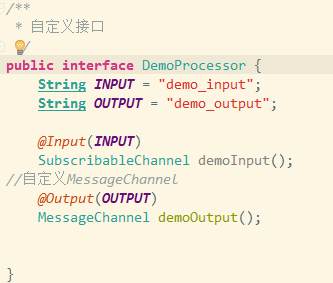
其中cloud.stream.bindings.demo\_output 属性为MQ绑定一个名为demo\_output的通道。

1. 代码编写及测试

UserRest controller中：

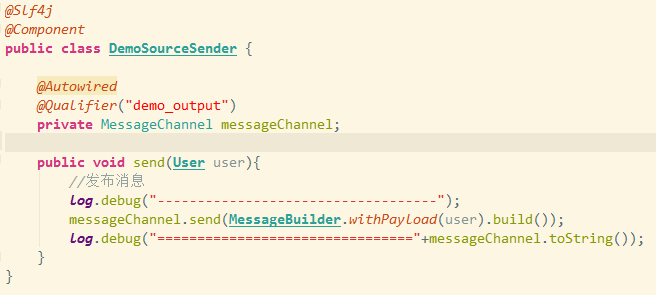


自定义通道接口：



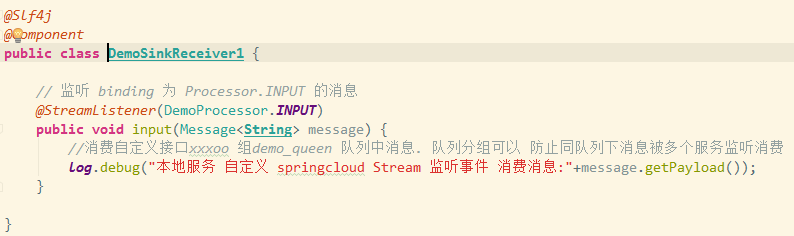
该接口声明了两个通道：demo\_input,demo\_output。

发布消息类：该接类负责通过自定义demo\_output通道发布消息到MQ队列中。



监听消息类：该类负责监听自定义通道demo\_input中获取消息队列中的消息。分为本地服务和异地服务两种环境。

Test-demo本地服务中：



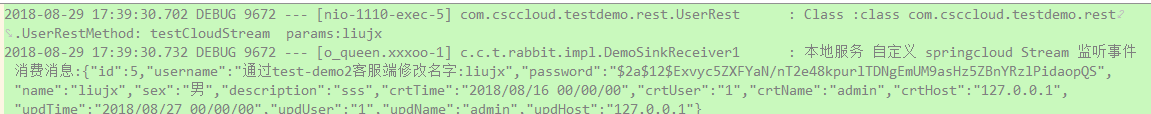
Test-demo2异地服务中：



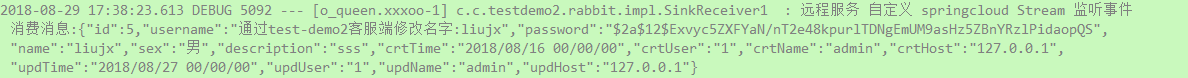
测试：<http://localhost:9000/api/test/cloudStream?name=liujx>

测试预期结果：

Test-demo 输出台：



Test-demo2 输出台：



可以看到test-demo,test-demo2输出台轮流打印日志显示已经监听到发布的消息。过程分析：

访问controller中调用自定义类发布消息到MQ消息队列中，自定义监听器方法因为配置中将自定义demo\_queen队列分组为xxxoo所以一个消息只能被同一服务监听器消费。即出现test-demo,test-demo2自定义消息监听器方法轮流消费消息。自此我们已经完成了自定义通道发布消息，不同服务中自定义监听器方法轮流消费消息的demo。可以看出代码量很少灵活度高，使用SpringCloud Stream实现消息机制非常便捷高效，代码耦合度低。

### 使用Redis实现缓存

#### 功能简介

Redis作为缓存提高服务查询效率，缓解数据库压力被广泛采用。但redis使用往往会在业务层入侵代码，项目使用github开源项目中封装的redis工具jar包实现使用注解的方式达到redis缓存的功能。减低代码耦合度的同时也减少开发的工作量。

#### 实现步骤

1. Test-demo项目中添加redis组件支持pom依赖

Pom文件如下：

<**dependency**>  
 <**groupId**>com.github.wxiaoqi</**groupId**>  
 <**artifactId**>ace-cache</**artifactId**>  
 <**version**>0.0.2</**version**>  
</**dependency**>

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

其中ace-cache依赖为第三方封装的redis工具jar包组件，spring-boot-strarter-data-redis依赖为SpringBoot对redis组件支持的依赖组件。

（第三方jar放置在test-demo/doc目录中，使用maven命令mvn install:install-file -DgroupId=com.github.wxiaoqi -DartifactId=ace-cache -Dversion=0.0.2 -Dpackaging=jar -Dfile=jar包所在目录\ace-cache-0.0.2.jar 即可安装第三方jar包到本地maven仓库。）

1. 在主配置类中加入注解声明该应用使用的组件



其中@EnableAceCache注解声明该应用使用为第三方redis工具组件。

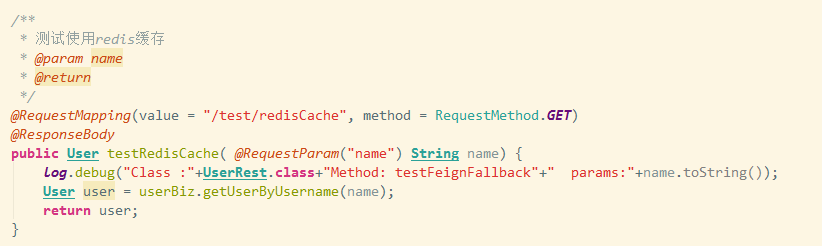
1. 编写配置



其中redis.sysName:liu属性声明该应用使用的redis库名为liu。库名可以自定义，根据不同服务需求考虑是否使用同名redis库以实现不同服务redis共享数据。推荐集群服务共享数据，不同业务服务不共享数据。

1. 代码编写及测试

UserRest controller：



UserServiceImpl:



其中@Cache(key=”user{1}”)方法级别注解，表示该方法使用redis缓存 ，注解属性key为该方法redis缓存的key值。其取值规则为-库名：左花括号前面的值：第n个方法参数的值 。key示例（liu:user:liujx）。

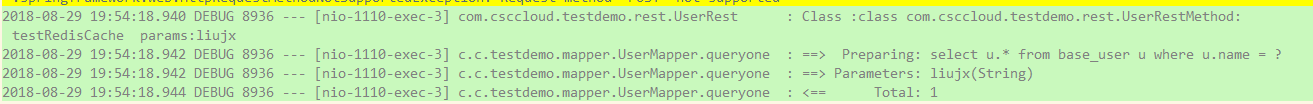
@CacheClear(pre=”user{1.name}”)表示清除redis key=liu:user:liujx对应缓存。即每次涉及到更新数据都清除redis相应key的缓存数据。

测试：url1:<http://localhost:9000/api/test/redisCache?name=liujx>

url2:

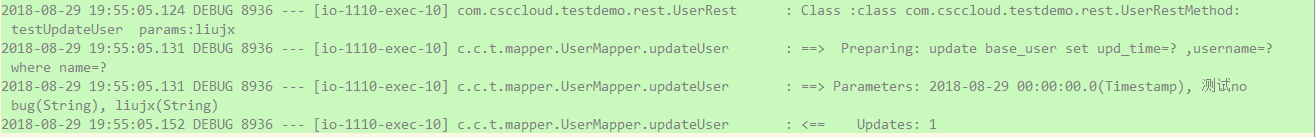
测试结果预期：

第一次访问url1:



可以看出后台有打印查询语句即发送了sql,而第二次就没有在打印sql日志了但仍然可以获取到User数据。说明缓存生效了。

访问url2:



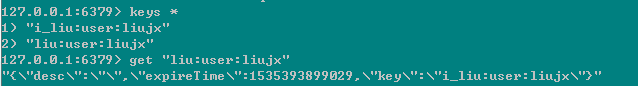
可以看出后台更新了user。

再次访问url1:



发现又打印了查询sql语句，即当数据更新时redis 对应key的数据被清空。保证了redis缓存的数据与持久化数据的一致性。

Redis 查看数据如下：



可以看到存在一条key=liu:user:liujx的数据。

自此第三方redis缓存工具使用演示完成。可以看到使用注解的方式还是比较方便简单的。

# Csc-cloud 使用

## 项目的获取与运行

* 使用Git 客户端 : git clone <https://gitee.com/iskech1/csc-cloud.git> 获取项目追踪master主分支。
* 使用svn checkout : <https://10.10.111.245/svn/csc-cloud/> ,user:root,password:root。
* 项目使用maven 构建，可使用主流IDE导入项目 。
* 项目运行顺序：cloud-eureka,cloud-config,cloud-zuul-gateway,test-demo1,2。运行主配置类main方法即可。（SpringBoot项目打包jar ,run main()即可将项目以jar包方式运行在SpringBoot内置tomcat中。）