**1 spring-boot**

**1.1 srping-boot 简介**

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

### 1.1 Spring Boot特点

创建独立的Spring应用程序

嵌入的Tomcat，无需部署WAR文件

简化Maven配置

自动配置Spring，没有冗余代码生成和XML配置的要求

### 1.1.2 系统要求：

Java 7及以上

Spring Framework 4.1.5及以上

### 1.1.3 使用Maven构建项目

(1)通过eclipse工具创建Maven Project项目

### (2)引入spring-boot jar包依赖（注：为了简化依赖图，Boot的功能是模块化的，通过导入Boot所谓的“starter”模块，可以将许多的依赖添加到工程之中。为了更容易地管理依赖版本和使用默认配置，框架提供了一个parent POM，工程可以继承它。）

Spring Boot工程的样例POM文件定义如程序如下所示：

<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>com..nf.helloword</groupId>

<artifactId>helloword</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>helloword</name>

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

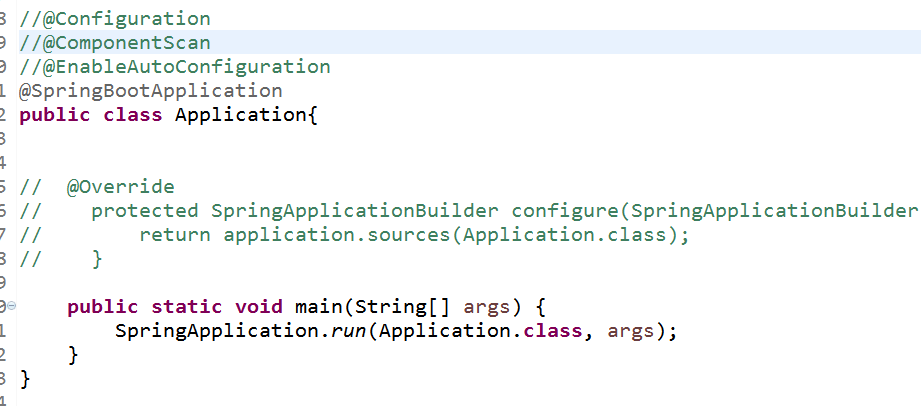
</dependencies>

</project>

### (3)编写HelloWorld服务

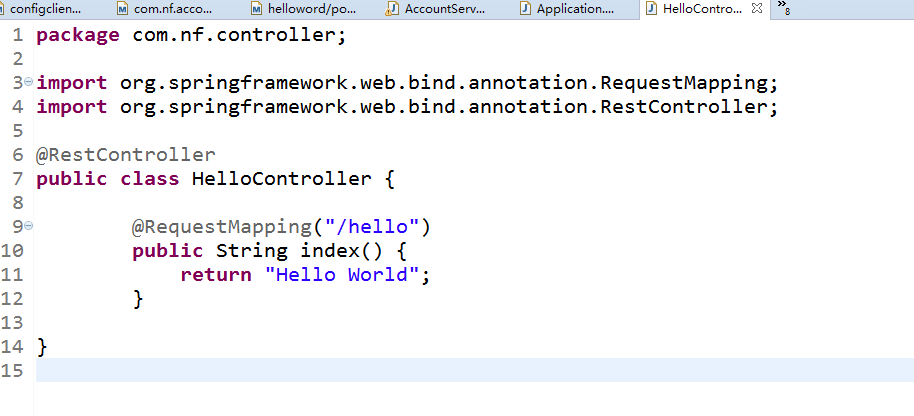
创建package命名为com.nf（根据实际情况修改）

创建Application.java主程序启动类，内容如下



创建package命名为com.nf.controller（根据实际情况修改）

创建HelloController类，内容如下



启动主程序，打开浏览器访问http://localhost:8080/hello，可以看到页面输出Hello World

@Controller：修饰class，用来创建处理http请求的对象

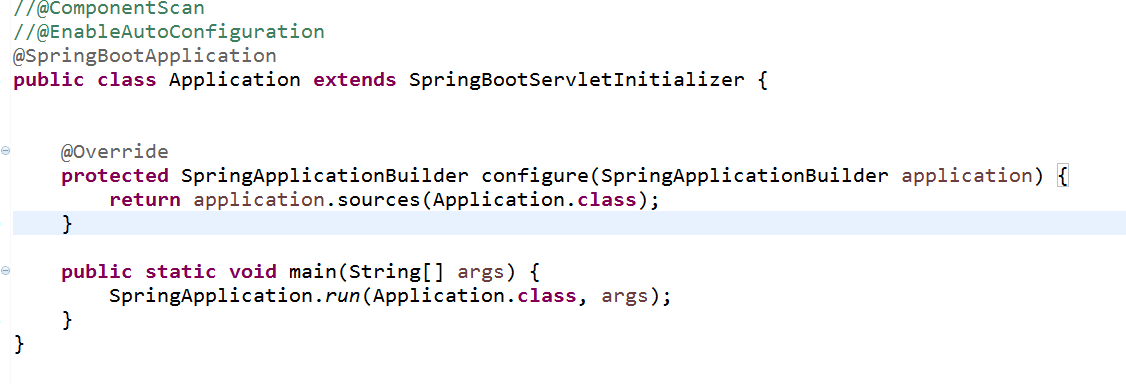
@RestController：Spring4之后加入的注解，原来在@Controller中返回json需要@ResponseBody来配合，如果直接用@RestController替代@Controller就不需要再配置@ResponseBody，默认返回json格式。

@RequestMapping：配置url映射

## 1.2 [spring-boot项目在外部tomcat环境下部署](http://blog.csdn.net/james_wade63/article/details/51009423)

[spring](http://lib.csdn.net/base/javaee)-boot默认提供内嵌的tomcat，所以打包直接生成jar包，用[Java](http://lib.csdn.net/base/java) -jar命令就可以启动。但是，有时候我们更希望一个tomcat来管理多个项目，这种情况下就需要项目是war格式的包而不是jar格式的包。spring-boot同样提供了解决方案，只需要简单的几步更改就可以了，这里提供maven项目的解决方法：

### 1.2.1将项目的启动类Application.java继承SpringBootServletInitializer,并重写configure方法，如下截图：



### 1.2.2在pom.xml文件中，project下面增加package标签

<packaging>war</packaging>

### 1.2.3.还是在pom.xml文件中，dependencies下面添加

<dependency>

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

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

<!-- 打包部署到tomcat上面时，不需要打包tmocat相关的jar包，否则会引起jar包冲突 -->

<exclusions>

<exclusion>

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

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

</exclusion>

</exclusions>

</dependency>

<dependency>

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

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

<scope>provided</scope>

</dependency>

### 1.2.4还是在pom.xml文件中，build下面添加

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<plugin>

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

<artifactId>maven-war-plugin</artifactId>

<configuration>

<warName>springboot</warName>

</configuration>

</plugin>

</plugins>

</build>

这样，只需要以上3步就可以打包成war包，并且部署到tomcat中了。需要注意的是这样部署的request url需要在端口后加上项目的名字才能正常访问,项目的名字即<warName>springboot</warName>。spring-boot更加强大的一点，就是：即便项目是以上配置，依然可以用内嵌的tomcat来调试。

按上面的例子，启动主程序，打开浏览器访问http://localhost:8080/springboot/hello，可以看到页面输出Hello World

## 1.3 Spring Boot中使用Swagger2构建强大的RESTful API文档

Swagger2可以轻松的整合到Spring Boot中，并与Spring MVC程序配合组织出强大RESTful API文档。它既可以减少我们创建文档的工作量，同时说明内容又整合入实现代码中，让维护文档和修改代码整合为一体，可以让我们在修改代码逻辑的同时方便的修改文档说明。另外Swagger2也提供了强大的页面测试功能来调试每个RESTful API。下面来具体介绍，如果在Spring Boot中使用Swagger2

### 1.3.1添加Swagger2依赖

<dependency>

<groupId>io.springfox</groupId>

<artifactId>springfox-swagger2</artifactId>

<version>2.2.2</version>

</dependency>

<dependency>

<groupId>io.springfox</groupId>

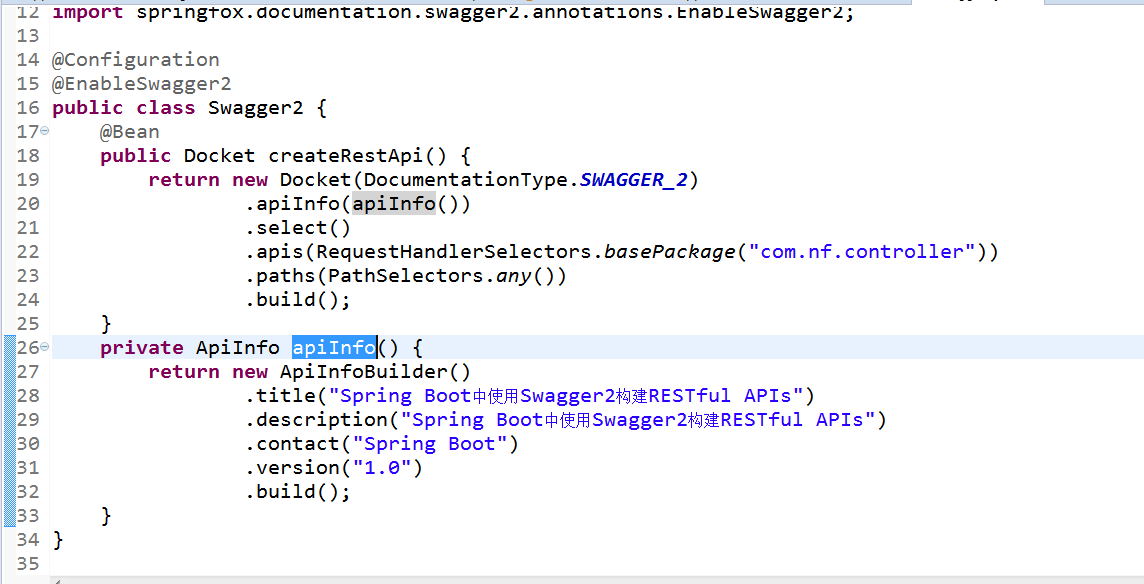
<artifactId>springfox-swagger-ui</artifactId>

<version>2.2.2</version>

</dependency>

### 1.3.2创建Swagger2配置类

在Application.java同级创建Swagger2的配置类Swagger2。代码如下图所示：



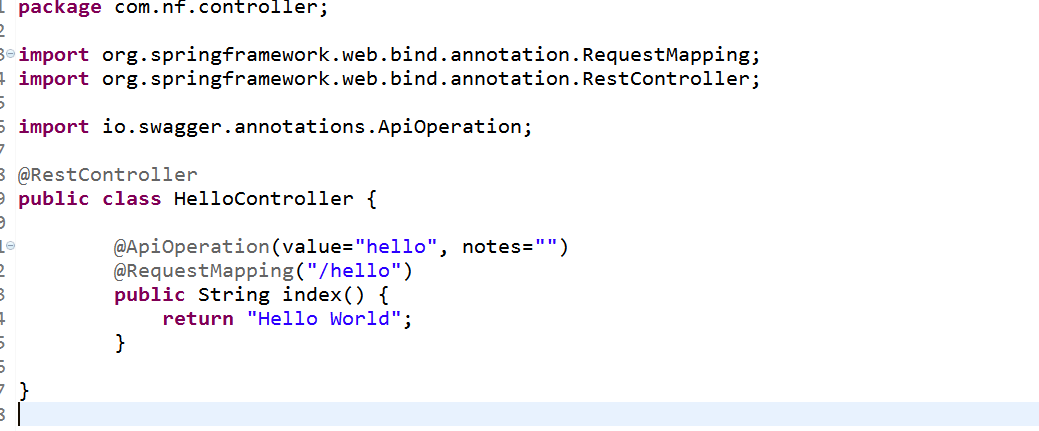
如上代码所示，通过@Configuration注解，让Spring来加载该类配置。再通过@EnableSwagger2注解来启用Swagger2。

再通过createRestApi函数创建Docket的Bean之后，apiInfo()用来创建该Api的基本信息（这些基本信息会展现在文档页面中）。select()函数返回一个ApiSelectorBuilder实例用来控制哪些接口暴露给Swagger来展现，本例采用指定扫描的包路径来定义，Swagger会扫描该包下所有Controller定义的API，并产生文档内容（除了被@ApiIgnore指定的请求）。

### 1.3.3 添加文档内容

在完成了上述配置后，其实已经可以生产文档内容，但是这样的文档主要针对请求本身，而描述主要来源于函数等命名产生，对用户并不友好，我们通常需要自己增加一些说明来丰富文档内容。如下所示，我们通过@ApiOperation注解来给API增加说明、通过@ApiImplicitParams、@ApiImplicitParam注解来给参数增加说明。

完成上述代码添加上，启动Spring Boot程序，访问：<http://localhost:8080/swagger-ui.html>  
。就能看到前文所展示的RESTful API的页面。



## 1.4  [Spring Boot 日志记录 SLF4J](http://blog.csdn.net/catoop/article/details/50501714)

这里使用了SLF4J的logback来输出日志，其比log4j 要好，因为他效率更高。

### 1.4.1 [spring](http://lib.csdn.net/base/javaee) Boot 提供了一套日志系统，logback是最优先的选择。

配置了logback.xml可以利用Spring Boot提供的默认日志配置：

<?xml version="1.0" encoding="UTF-8"?>

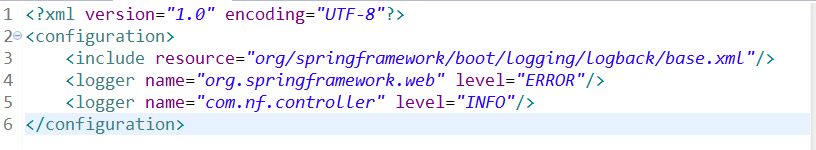
<configuration>

<include resource="org/springframework/boot/logging/logback/base.xml"/>

<logger name="org.springframework.web" level="DEBUG"/>

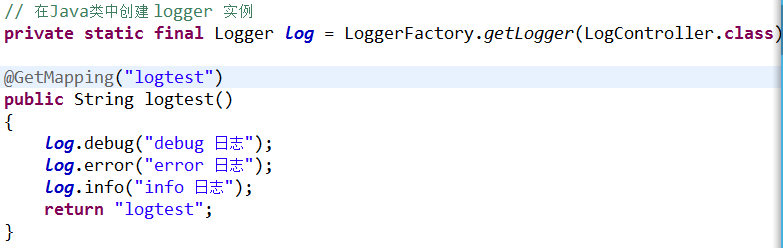
</configuration>

Spring Boot提供的默认日志配置，所以我们只需要在src/main/resources目录下建立一个logback.xml，并将我们需要记录日志的包路径和日志级别配置好，我们就可以进行日志的输出了。如下图所示：



在logback.xml中配置了包路径为com.nf.controller日志级别为INFO.

### 1.4.2 在Java代码中创建实例，并在需要输出日志的地方使用。



### 1.4.3 将日志记录到日志文件中

一般我们记录日志都是将日志记录到日志文件中，要记录到日志文件中，这需要我们在application.properties文件中加上logging.file= log/logtest.log的配置。这样我们就可以将控制台输出的日志记录到log文件夹里的logtest.log中了。

## 1.5 Spring Boot中使用JdbcTemplate访问数据库

通常我们做App也好，做Web应用也好，都需要内容，而内容通常存储于各种类型的数据库，服务端在接收到访问请求之后需要访问数据库获取并处理成展现给用户使用的数据形式。接下来我们介绍一下，spring boot是如何访问数据库的。

### 1.5.1 数据源的配置

（1）首先，为了连接数据库需要引入jdbc支持，这里我使用的是mysql数据库，在pom.xml中引入如下配置：

<dependency>

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

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

</dependency>

<dependency>

<groupId>mysql</groupId>

<artifactId>mysql-connector-java</artifactId>

<version>5.1.21</version>

</dependency>

（2）在src/main/resources/application.properties中配置数据库连接信息，配置如下：

spring.datasource.url=jdbc:mysql://localhost:3306/test

spring.datasource.username=root

spring.datasource.password=123456

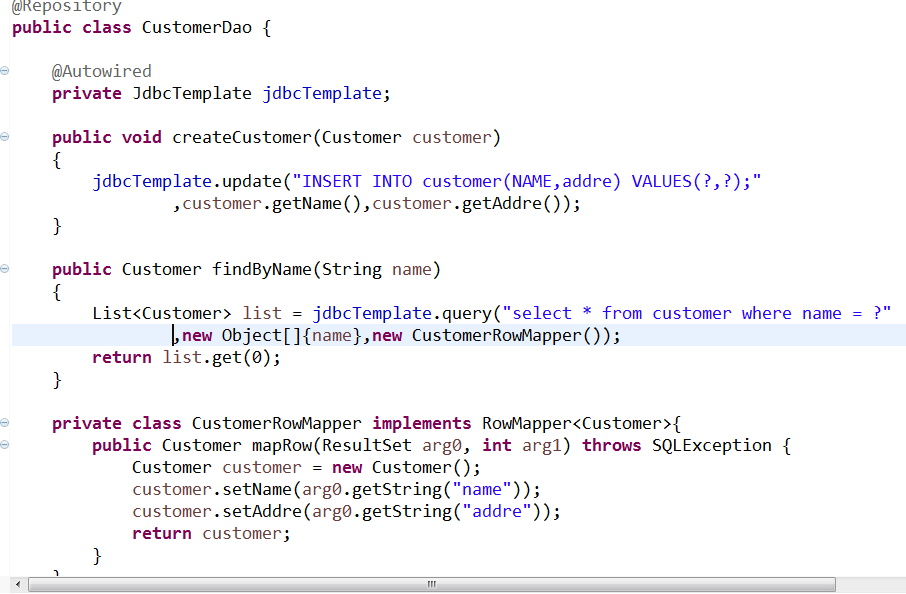
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

### 1.5.2 使用JdbcTemplate操作数据库

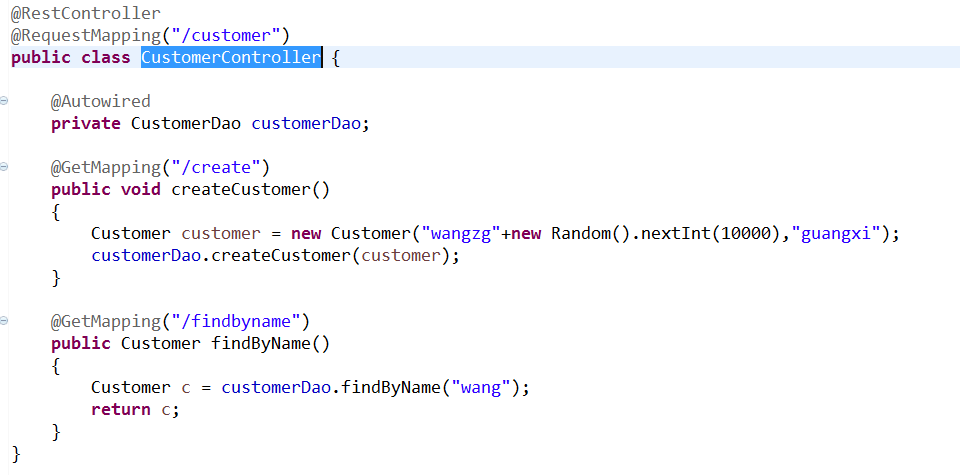
Spring的JdbcTemplate是自动配置的，你可以直接使用@Autowired来注入到你自己的bean中来使用。

举例：我们在创建customer表，包含属性name、addre，下面来编写数据访问对象和接口进行测试。

* 通过JdbcTemplate实现customerdao 插入和查询的数据访问操作



### 1.5.3创建CustomerController进行接口测试，通过调用创建和查询的接口来验证数据库操作的正确性。



# 2.Spring Cloud

## Spring Cloud简介

Spring Cloud是一个基于Spring Boot实现的云应用开发工具，它为基于JVM的云应用开发中的配置管理、服务发现、断路器、智能路由、微代理、控制总线、全局锁、决策竞选、分布式会话和集群状态管理等操作提供了一种简单的开发方式。

## 2.1分布式配置中心

Spring Cloud Config为服务端和客户端提供了分布式系统的外部化配置支持。配置服务器为各应用的所有环境提供了一个中心化的外部配置。配置服务器默认采用git来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过git客户端工具来方便的管理和访问配置内容，当然他也提供本地化文件系统的存储方式。

### 2.1.1构建Config Server

（1）pom.xml中引入spring-cloud-config-server依赖，完整依赖配置如下：

<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>com.nf.config</groupId>

<artifactId>configservice</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>configservice</name>

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<plugin>

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

<artifactId>maven-war-plugin</artifactId>

<configuration>

<warName>springboot</warName>

</configuration>

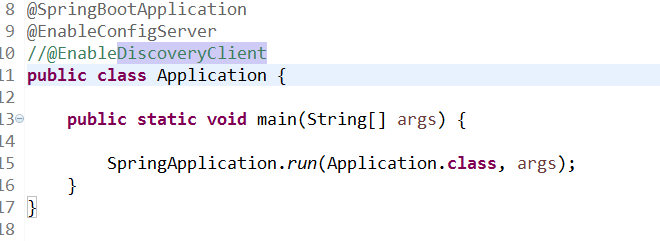
</plugin>

</plugins>

</build>

</project>

* （2）创建Spring Boot的程序主类，并添加@EnableConfigServer注解，开启Config Server



（3）application.properties中配置服务信息以及git信息，例如：

server.port=8889

spring.application.name=config-server

spring.cloud.config.server.git.uri=https://github.com/wangzigui/hello-wolrd

spring.cloud.config.server.git.username=username

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

* spring.cloud.config.server.git.uri：配置git仓库位置
* spring.cloud.config.server.git.username：访问git仓库的用户名
* spring.cloud.config.server.git.password：访问git仓库的用户密码

到这里，使用一个通过Spring Cloud Config实现，并使用git管理内容的配置中心已经完成了，启动该应用，成功后开始下面的内容。

Spring Cloud Config也提供本地存储配置的方式。我们只需要设置属性spring.profiles.active=native，Config Server会默认从应用的src/main/resource目录下检索配置文件。也可以通过spring.cloud.config.server.native.searchLocations=file:F:/properties/属性来指定配置文件的位置。虽然Spring Cloud Config提供了这样的功能，但是为了支持更好的管理内容和版本控制的功能，还是推荐使用git的方式

### 2.1.2微服务端映射配置

在完成并验证了配置服务中心之后，下面看看我们如何在微服务应用中获取配置信息。

（1）创建一个Spring Boot应用，在pom.xml中引入spring-cloud-starter-config依赖，完整依赖关系如下：

<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>com.nf.configclient</groupId>

<artifactId>configclient</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>configclient</name>

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

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

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<plugin>

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

<artifactId>maven-war-plugin</artifactId>

<configuration>

<warName>springboot</warName>

</configuration>

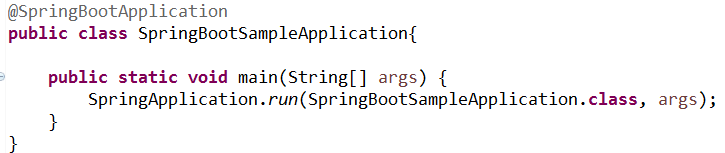
</plugin>

</plugins>

</build>

</project>

（2）创建最基本的Spring Boot启动主类



（3）创建bootstrap.properties配置，来指定config server，例如：

spring.application.name=client1

server.port=8006

spring.cloud.config.uri=http://localhost:8889

spring.cloud.config.name=sample-cloud-client

spring.cloud.config.profile=${config.profile:dev}

这里需要格外注意：上面这些属性必须配置在bootstrap.properties中，config部分内容才能被正确加载。因为config的相关配置会先于application.properties，而bootstrap.properties的加载也是先于application.properties

（4）创建application.properties并配置：

app.message=local message

user.username=my name is shanhy.(local)

user.userage=28

（5）在git仓库<https://github.com/wangzigui/hello-wolrd>创建一个sample-cloud-client-dev.properties的文件，文件里设置

app.message=Hi~wangzigui. message version 1.4 (dev)

（6）创建一个Rest Api来返回配置中心的from属性，具体如下：



启动该应用，并访问：http://localhost:8006/test，我们就可以根据配置内容输出对应环境的app.message内容了

## 2.2 服务注册与发现

### 2.2.1创建“服务注册中心”

（1）创建一个基础的Spring Boot工程，并在pom.xml中引入需要的依赖内容：

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

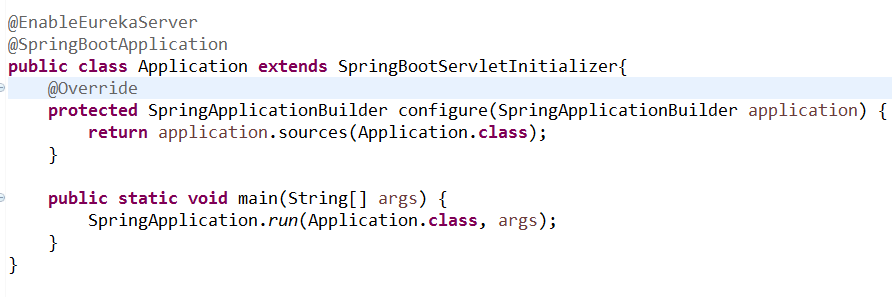
<groupId>org.springframework.cloud</groupId>

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

</dependency>

</dependencies>

（2）通过@EnableEurekaServer注解启动一个服务注册中心提供给其他应用进行对话。这一步非常的简单，只需要在一个普通的Spring Boot应用中添加这个注解就能开启此功能，比如下面的例子：



在默认设置下，该服务注册中心也会将自己作为客户端来尝试注册它自己，所以我们需要禁用它的客户端注册行为，只需要在application.properties中问增加如下配置：

server.port=8001

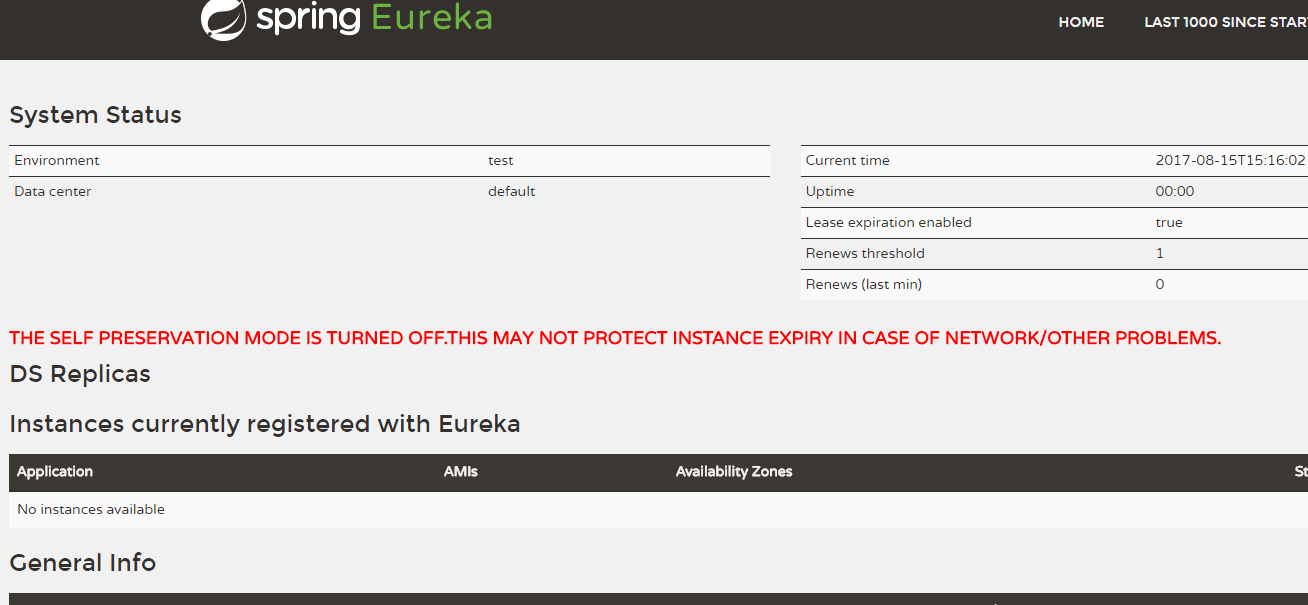
eureka.client.register-with-eureka=false

eureka.client.fetch-registry=false

eureka.client.serviceUrl.defaultZone=http://localhost:${server.port}/eureka

启动工程后，访问：<http://localhost:8001/>

可以看到下面的页面，其中还没有发现任何服务



### 2.2.2创建“服务提供方”

下面我们创建提供服务的客户端，并向服务注册中心注册自己。

1. 创建一个基本的Spring Boot应用，在pom.xml中，加入如下配置：

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

</dependencies>

1. 创建application.properties做一些配置工作，具体如下：

spring.application.name=client1

server.port=8006

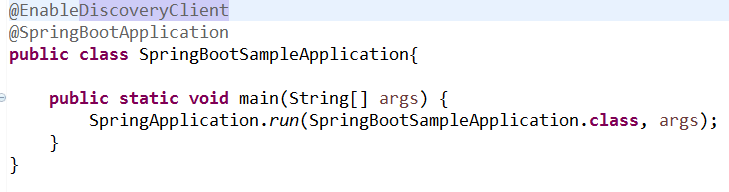
#配置服务注册中心

eureka.client.service-url.defaultZone=http://localhost:8001/eureka

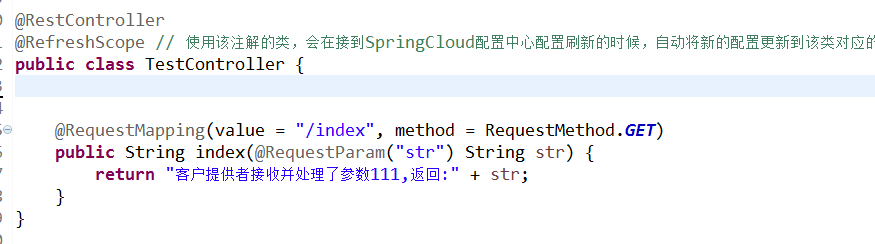
通过spring.application.name属性，我们可以指定微服务的名称后续在调用的时候只需要使用该名称就可以进行服务的访问。

eureka.client.serviceUrl.defaultZone属性对应服务注册中心的配置内容，指定服务注册中心的位置。

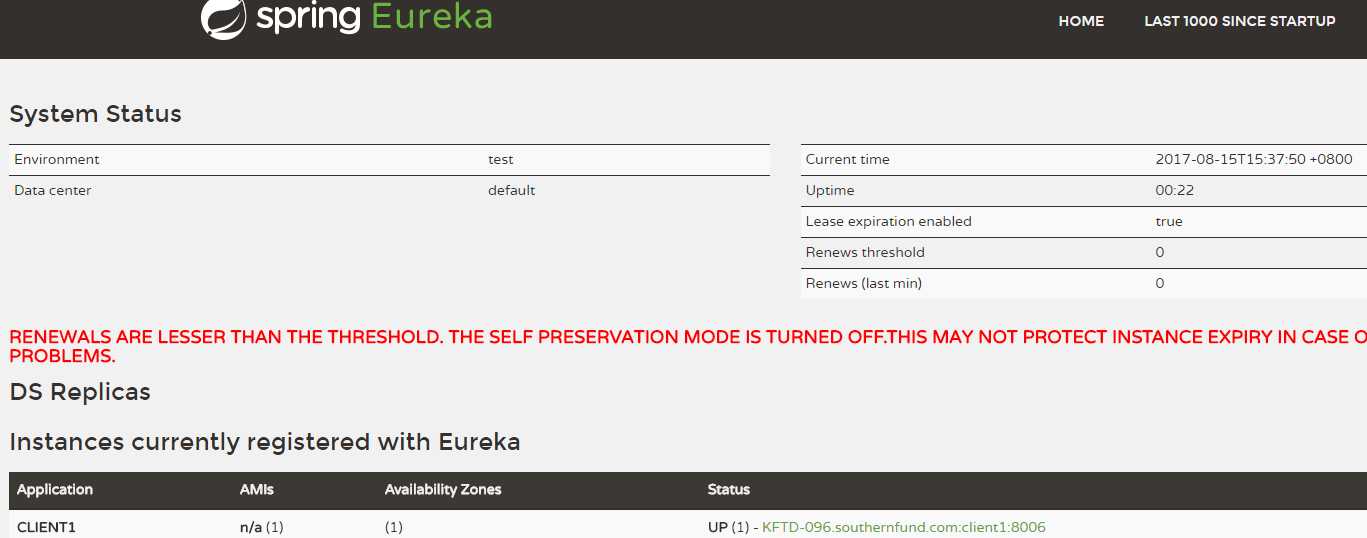
1. 在主类中通过加上@EnableDiscoveryClient注解，该注解能激活Eureka中的DiscoveryClient实现



1. 最后，实现/index请求处理接口，通过DiscoveryClient对象，在日志中打印出相关内容。



启动该工程后，再次访问：<http://localhost:8001/>，可以看到我们刚刚启动的微服务已经注册到注册中心



### 2.2.3 构建服务消费者

上面我们已经成功创建了“服务注册中心”，实现并注册了一个“服务提供者：client1。那么我们要如何去消费服务提供者的接口内容呢？

Feign是一个声明式的Web Service客户端，它使得编写Web Serivce客户端变得更加简单。我们只需要使用Feign来创建一个接口并用注解来配置它既可完成。它具备可插拔的注解支持，包括Feign注解和JAX-RS注解。Feign也支持可插拔的编码器和解码器。Spring Cloud为Feign增加了对Spring MVC注解的支持，还整合了Ribbon和Eureka来提供均衡负载的HTTP客户端实现

下面我们通过一个例子来展现Feign如何方便的声明对上述client1服务的定义和调用。

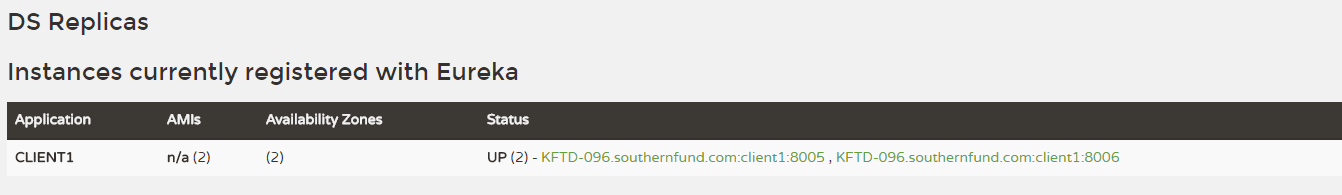
准备工作：

启动上面构建中的服务注册中心：eureka-server

启动上面构建中的服务提供方：client1

修改client1中的server-port为8005，再启动一个服务提供方：client1

此时访问：<http://localhost:8001/>



1. 再创建一个Spring Boot工程，配置pom.xml，具体如下：

<parent>

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

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

<version>1.5.2.RELEASE</version>

<relativePath/>

</parent>

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Dalston.RELEASE</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

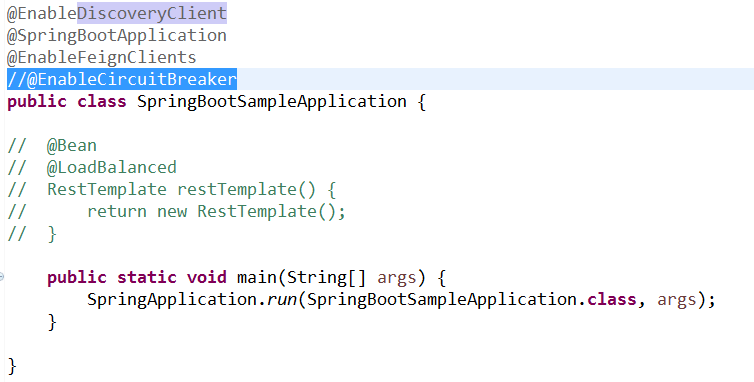
<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

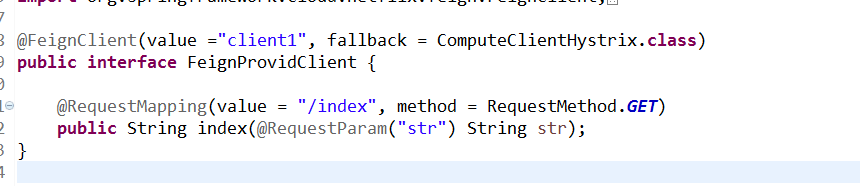
</dependency>

</dependencies>

（2）在应用主类中通过@EnableFeignClients注解开启Feign功能，具体如下：



（3）定义client1服务的接口，具体如下：



使用@FeignClient("client1")注解来绑定该接口对应compute-service服务

通过Spring MVC的注解来配置client1服务下的具体实现。

（4）在web层中调用上面定义的FeignProvidClient，具体如下



1. application.properties指定eureka服务注册中心即可，如：

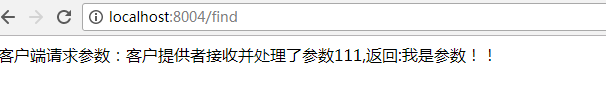
spring.application.name=client

server.port=8004

#配置服务注册中心

eureka.client.service-url.defaultZone=http://localhost:8001/eureka/

启动该应用，访问几次：<http://localhost:8004/find>



可以看到client调用了client1的接口成功，并返回了参数。

## 2.3[Hystrix断路器(熔断器)](http://blog.csdn.net/zhuchuangang/article/details/51289593)

Netflix创建了一个名为Hystrix的库,实现了断路器的模式。

在microservice[架构](http://lib.csdn.net/base/architecture)通常有多个层的服务调用。 低水平的服务的服务失败会导致级联故障一直给到用户。当调用一个特定的服务达到一定阈值(默认5秒失败20次),打开断路器。在错误的情况下和一个开启的断路回滚应可以由开发人员提供。 有一个断路器阻止级联失败并且允许关闭服务一段时间进行愈合。回滚会被其他hystrix保护调用，静态数据或健全的空值。

下面是使用Spring Cloud Hystrix实现断路器的过程：

### 2.3.1 pom.xml的dependencies节点中引入spring-cloud-starter-hystrix依赖：

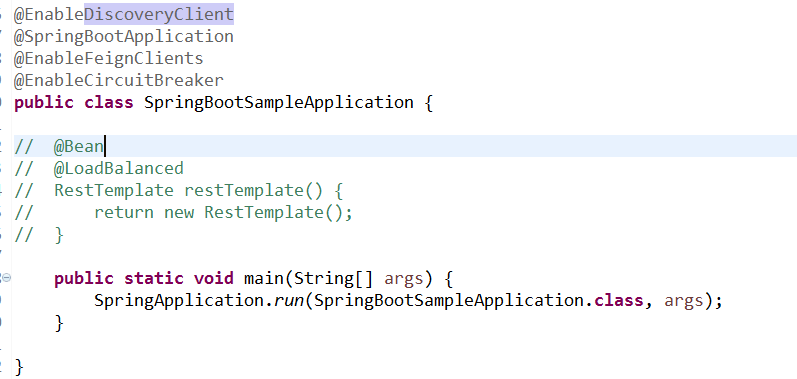
<dependency>

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

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

</dependency>

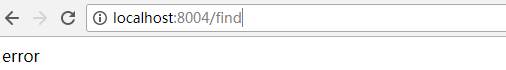
### 2.3.2 在应用主类中使用@EnableCircuitBreaker或@EnableHystrix注解开启Hystrix的使用：



### 2.3.3为具体执行逻辑的函数上增加@HystrixCommand注解，并实现fallbackMethod对应的方法



启动服务，不启动FeignProvidClient类对应的外部服务 然后访问<http://localhost:8004/find>，可以看到如下返回。



## 2.4Spring Cloud服务网关

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。

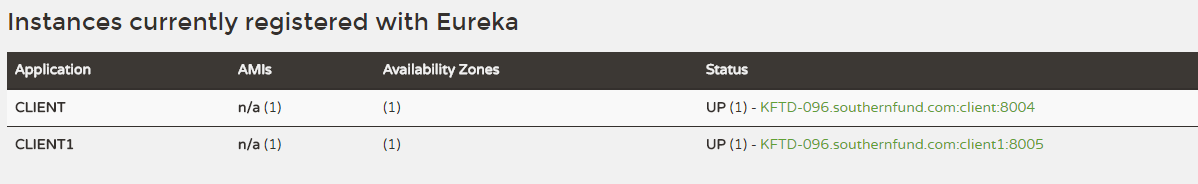
下面将要介绍一个Spring Cloud很重要的一个组件：服务网关

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。

下面我们通过实例例子来使用一下Zuul来作为服务的路有功能。

### 准备工作

在使用Zuul之前，我们先构建一个服务注册中心、以及两个简单的服务，比如：我构建了一个client，一个client1。然后启动eureka-server和这两个服务。通过访问eureka-server，我们可以看到client和client1已经注册到了服务中心。



### 2.4.1开始使用Zuul

（1）新建一个spring-boot工程

（2）引入依赖spring-cloud-starter-zuul、spring-cloud-starter-eureka，如果不是通过指定serviceId的方式，eureka依赖不需要，但是用serviceId来避免可以直接引用url的方式。

<dependency>

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

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

</dependency>

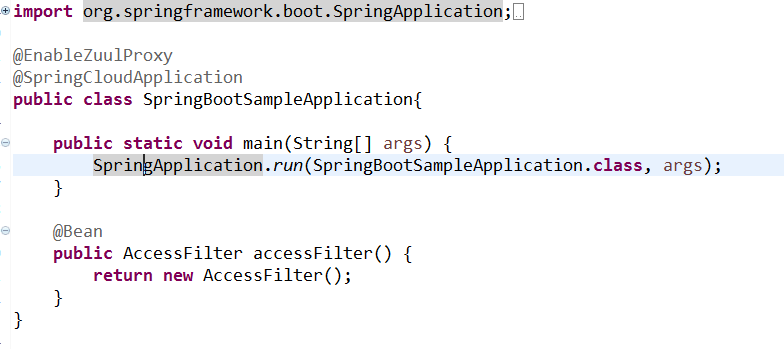
<dependency>

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

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

</dependency>

* 应用主类使用@EnableZuulProxy注解开启Zuul



*这里用了*@SpringCloudApplication*注解，之前没有提过，通过源码我们看到，它整合了*@SpringBootApplication*、*@EnableDiscoveryClient*、*@EnableCircuitBreaker*，主要目的还是简化配置。这几个注解的具体作用这里就不做详细介绍了，之前的文章已经都介绍过。*

* application.properties中配置Zuul应用的基础信息，如：应用名、服务端口等。

spring.application.name=api-gateway

* server.port=5555

### 2.4.2服务路由

通过服务路由的功能，我们在对外提供服务的时候，只需要通过暴露Zuul中配置的调用地址就可以让调用方统一的来访问我们的服务，而不需要了解具体提供服务的主机信息了。

在Zuul中提供了两种映射方式：

（1）通过url直接映射，我们可以如下配置：

zuul.routes.api-a-url.path=/api-a-url/\*\*

zuul.routes.api-a-url.url=http://localhost:8004/

该配置，定义了，所有到Zuul的中规则为：/api-a-url/\*\*的访问都映射到http://localhost:8004/上，也就是说当我们访问http://localhost:5555/api-a-url/test的时候，Zuul会将该请求路由到：http:// localhost:8004/test上。

其中，配置属性zuul.routes.api-a-url.path中的api-a-url部分为路由的名字，可以任意定义，但是一组映射关系的path和url要相同，下面讲serviceId时候也是如此。

1. 通过url映射的方式对于Zuul来说，并不是特别友好，Zuul需要知道我们所有为服务的地址，才能完成所有的映射配置。而实际上，我们在实现微服务架构时，服务名与服务实例地址的关系在eureka server中已经存在了，所以只需要将Zuul注册到eureka server上去发现其他服务，我们就可以实现对serviceId的映射。例如，我们可以如下配置：

zuul.routes.api-a.path=/api-a/\*\*

zuul.routes.api-a.serviceId=client

zuul.routes.api-b.path=/api-b/\*\*

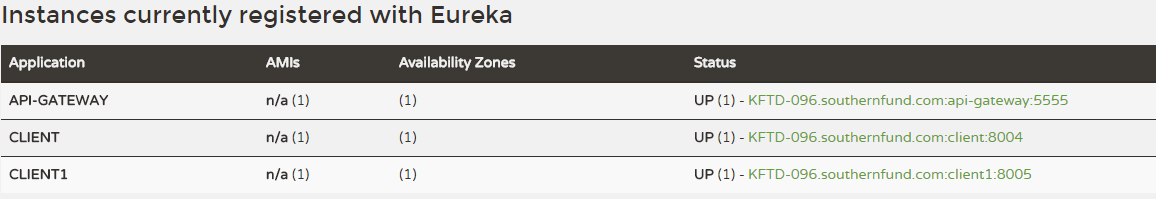
zuul.routes.api-b.serviceId=client1

# 注册中心地址

eureka.client.service-url.defaultZone=http://localhost:8001/eureka

针对我们在准备工作中实现的两个微服务client和client1，定义了两个路由api-a和api-b来分别映射。另外为了让Zuul能发现client和client1，也加入了eureka的配置。

接下来，我们将eureka-server、client、client1以及这里用Zuul实现的服务网关启动起来，在eureka-server的控制页面中，我们可以看到分别注册了client、client1以及api-gateway



尝试通过服务网关来访问client和client1，根据配置的映射关系，分别访问下面的url

* http://localhost:5555/api-a/test：通过serviceId映射访问service-A中的add服务
* http://localhost:5555/api-b/test：通过serviceId映射访问service-B中的add服务

*推荐使用serviceId的映射方式，除了对Zuul维护上更加友好之外，serviceId映射方式还支持了断路器，对于服务故障的情况下，可以有效的防止故障蔓延到服务网关上而影响整个系统的对外服务*

**2.4.3服务过滤**

在完成了服务路由之后，我们对外开放服务还需要一些安全措施来保护客户端只能访问它应该访问到的资源。所以我们需要利用Zuul的过滤器来实现我们对外服务的安全控制。

在服务网关中定义过滤器只需要继承ZuulFilter抽象类实现其定义的四个抽象函数就可对请求进行拦截与过滤。

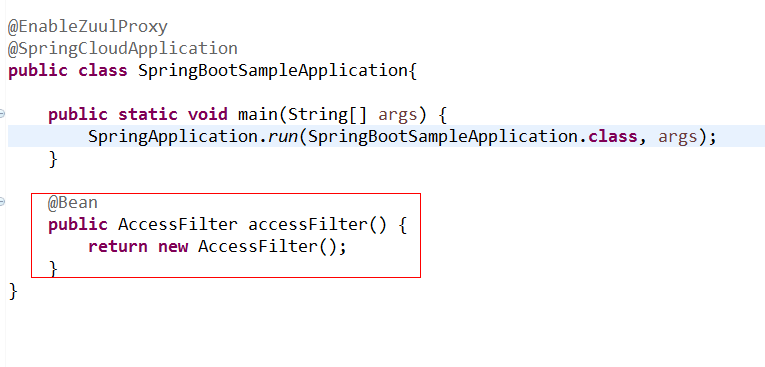
比如下面的例子，定义了一个Zuul过滤器，实现了在请求被路由之前检查请求中是否有accessToken参数，若有就进行路由，若没有就拒绝访问，返回401 Unauthorized错误。



自定义过滤器的实现，需要继承ZuulFilter，需要重写实现下面四个方法：

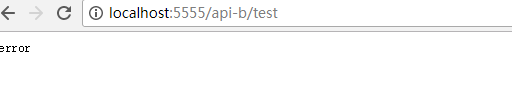
* filterType：返回一个字符串代表过滤器的类型，在zuul中定义了四种不同生命周期的过滤器类型，具体如下：
  + pre：可以在请求被路由之前调用
  + routing：在路由请求时候被调用
  + post：在routing和error过滤器之后被调用
  + error：处理请求时发生错误时被调用
* filterOrder：通过int值来定义过滤器的执行顺序
* shouldFilter：返回一个boolean类型来判断该过滤器是否要执行，所以通过此函数可实现过滤器的开关。在上例中，我们直接返回true，所以该过滤器总是生效。
* run：过滤器的具体逻辑。需要注意，这里我们通过ctx.setSendZuulResponse(false)令zuul过滤该请求，不对其进行路由，然后通过ctx.setResponseStatusCode(401)设置了其返回的错误码，当然我们也可以进一步优化我们的返回，比如，通过ctx.setResponseBody(body)对返回body内容进行编辑等。

在实现了自定义过滤器之后，还需要实例化该过滤器才能生效，我们只需要在应用主类中增加如下内容：



启动该服务网关后，访问：

未加上accessToken=token参数



加上accessToken=token参数

