**Microservice Spring cloud**

## 一、Spring cloud简介

**1 Spring Cloud 为开发人员提供了快速构建分布式系统的一些工具，包括配置管理、服务发现、断路器、路由、微代理、事件总线、全局锁、决策竞选、分布式会话等等。**

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

**Spring Cloud和云计算没有关系，只是一个基于Spring Boot的快速构建分布式系统的工具集。**

**Spring Cloud Config：**配置管理工具包，让你可以把配置放到远程服务器，集中化管理集群配置，目前支持本地存储、Git以及Subversion。

**Spring Cloud Bus：**事件、消息总线，用于在集群（例如，配置变化事件）中传播状态变化，可与Spring Cloud Config联合实现热部署。

**Eureka：**云端服务发现，一个基于 REST 的服务，用于定位服务，以实现云端中间层服务发现和故障转移。

**Hystrix：**熔断器，容错管理工具，旨在通过熔断机制控制服务和第三方库的节点,从而对延迟和故障提供更强大的容错能力。

**Ribbon：**提供云端负载均衡，有多种负载均衡策略可供选择，可配合服务发现和断路器使用。

**Zuul：**Zuul 是在云平台上提供动态路由,监控,弹性,安全等边缘服务的框架。Zuul 相当于是设备和 Netflix 流应用的 Web 网站后端所有请求的前门。

**Archaius：**配置管理API，包含一系列配置管理API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

**Consul：**封装了Consul操作，consul是一个服务发现与配置工具，与Docker容器可以无缝集成。

**Spring Cloud for Cloud Foundry：**通过Oauth2协议绑定服务到CloudFoundry，CloudFoundry是VMware推出的开源PaaS云平台。

**Spring Cloud Sleuth：**日志收集工具包，封装了Dapper和log-based追踪以及Zipkin和HTrace操作，为SpringCloud应用实现了一种分布式追踪解决方案。



## 二、服务的注册与发现

**1 服务注册**

在服务治理框架中，通常都会构建一个注册中心，每个服务单元向注册中心登记自己提供的服务，包括服务的主机与端口号、服务版本号、通讯协议等一些附加信息。注册中心按照服务名分类组织服务清单，同时还需要以心跳检测的方式去监测清单中的服务是否可用，若不可用需要从服务清单中剔除，以达到排除故障服务的效果。

**2 服务发现**

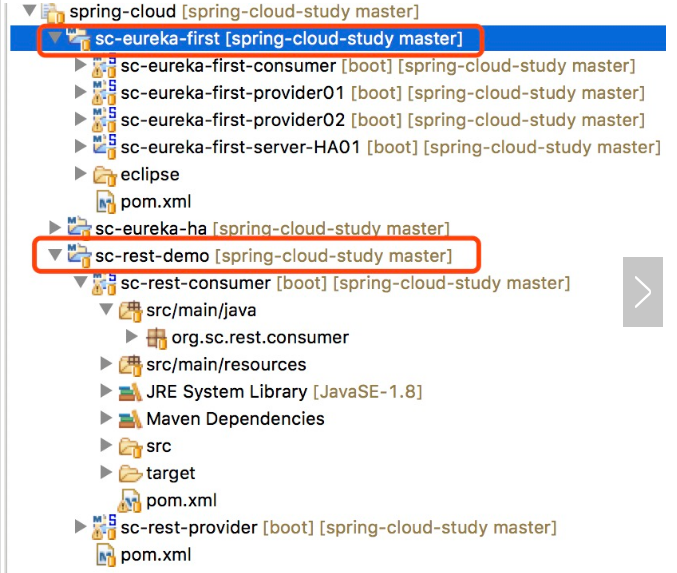
在服务治理框架下，服务间的调用不再通过指定具体的实例地址来实现，而是通过服务名发起请求调用实现。服务调用方通过服务名从服务注册中心的服务清单中获取服务实例的列表清单，通过指定的负载均衡策略取出一个服务实例位置来进行服务调用。

**3 Eureka服务端**

Eureka服务端，即服务注册中心。它同其他服务注册中心一样，支持高可用配置。依托于强一致性提供良好的服务实例可用性，可以应对多种不同的故障场景。Eureka服务端支持集群模式部署，当集群中有分片发生故障的时候，Eureka会自动转入自我保护模式。它允许在分片发生故障的时候继续提供服务的发现和注册，当故障分配恢复时，集群中的其他分片会把他们的状态再次同步回来。集群中的的不同服务注册中心通过异步模式互相复制各自的状态，这也意味着在给定的时间点每个实例关于所有服务的状态可能存在不一致的现象。

**4 Eureka客户端**

Eureka客户端，主要处理服务的注册和发现。客户端服务通过注册和参数配置的方式，嵌入在客户端应用程序的代码中。在应用程序启动时，Eureka客户端向服务注册中心注册自身提供的服务，并周期性的发送心跳来更新它的服务租约。同时，他也能从服务端查询当前注册的服务信息并把它们缓存到本地并周期行的刷新服务状态。



**5 搭建注册中心-Eureka Server**

**1.引入pom依赖**

<?xml version="1.0"?>

<project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd" xmlns="http://maven.apache.org/POM/4.0.0"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<modelVersion>4.0.0</modelVersion>

<!-- 引入spring boot的依赖 -->

<parent>

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

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

<version>1.4.3.RELEASE</version>

</parent>

<artifactId>sc-eureka-first-server-HA01</artifactId>

<name>sc-eureka-first-server-HA01</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<!-- 引入Spring Cloud Eureka依赖 -->

<dependency>

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

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

</dependency>

</dependencies>

<!-- 引入spring cloud的依赖 -->

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Camden.SR5</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- 添加spring-boot的maven插件-->

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

**2.在Resources目录下创建application.yml**

server:

port: 8761 # 指定该Eureka实例的端口

eureka:

client:

#表示是否将自己注册到Eureka Server上，默认为true，当前应用为Eureka Server所以无需注册

registerWithEureka: false

#表示是否从Eureka Server获取注册信息，默认为true。因为这是一个单点的Eureka Server，不需要同步其他的Eureka Server节点的数据，故而设为false。

fetchRegistry: false

#Eureka Server的访问地址，服务注册和client获取服务注册信息均通过该URL，多个服务注册地址用,隔开

serviceUrl:

defaultZone: http://localhost:8761/eureka/

**5 [expected <block end>, but found BlockMappingStart解决方法](https://www.cnblogs.com/chenkeyu/p/6858342.html)**

**在配置文件中vim /etc/elasticsearch/elasticsearch.yml中。**

**每个配置行前需要有空格**

**每个‘：’两边需要有空格**

**数组中间加空格**

**还有注释掉的参数不能在#后边加空格不然报错**

**6 创建Spring Boot主应用程序启动代码**

package org.xujin.sc.eureka.server;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@SpringBootApplication

@EnableEurekaServer

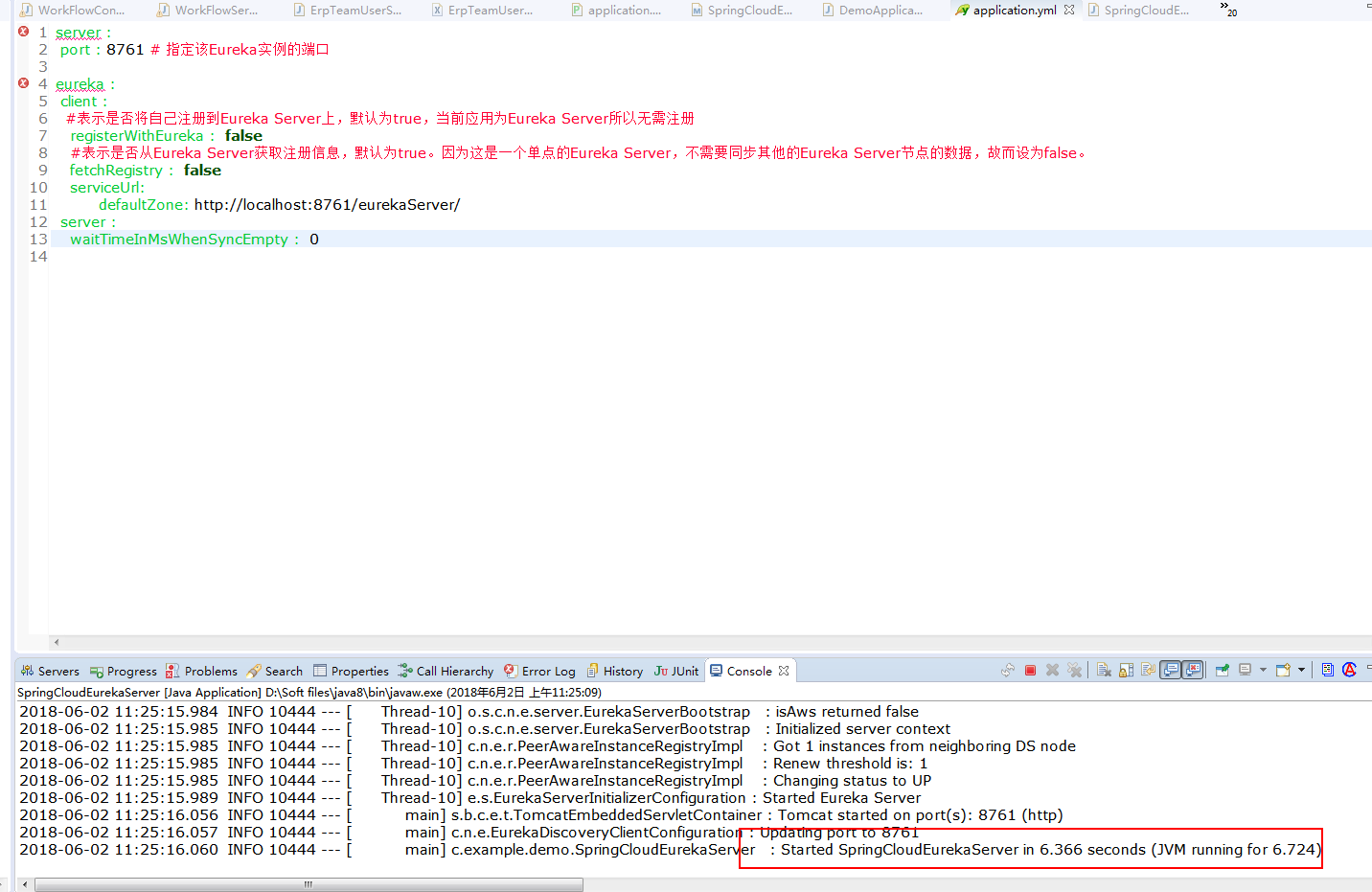
public class SpringCloudEurekaServer {

public static void main(String[] args) {

SpringApplication.run(SpringCloudEurekaServer.class, args);

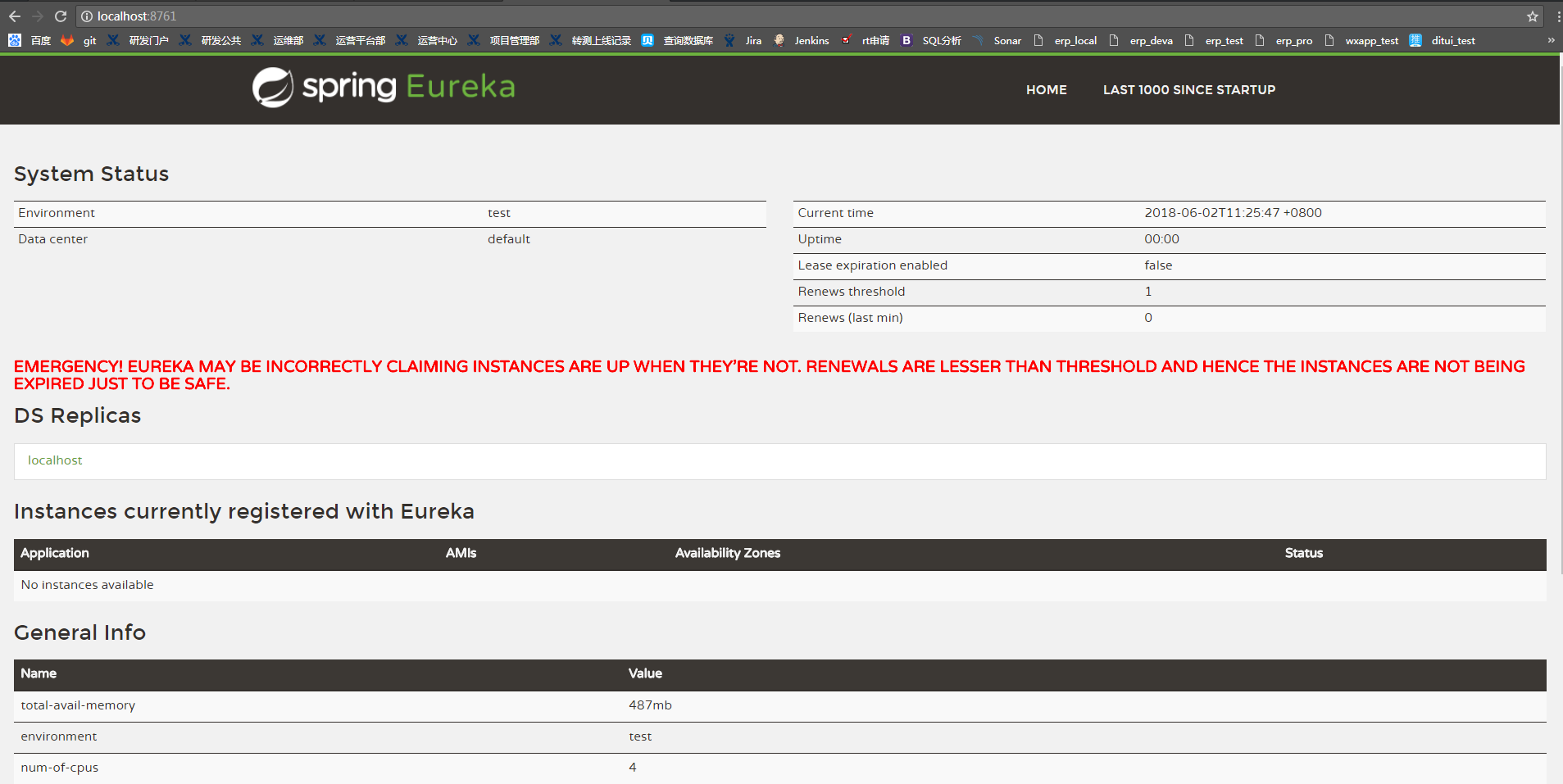
}

}



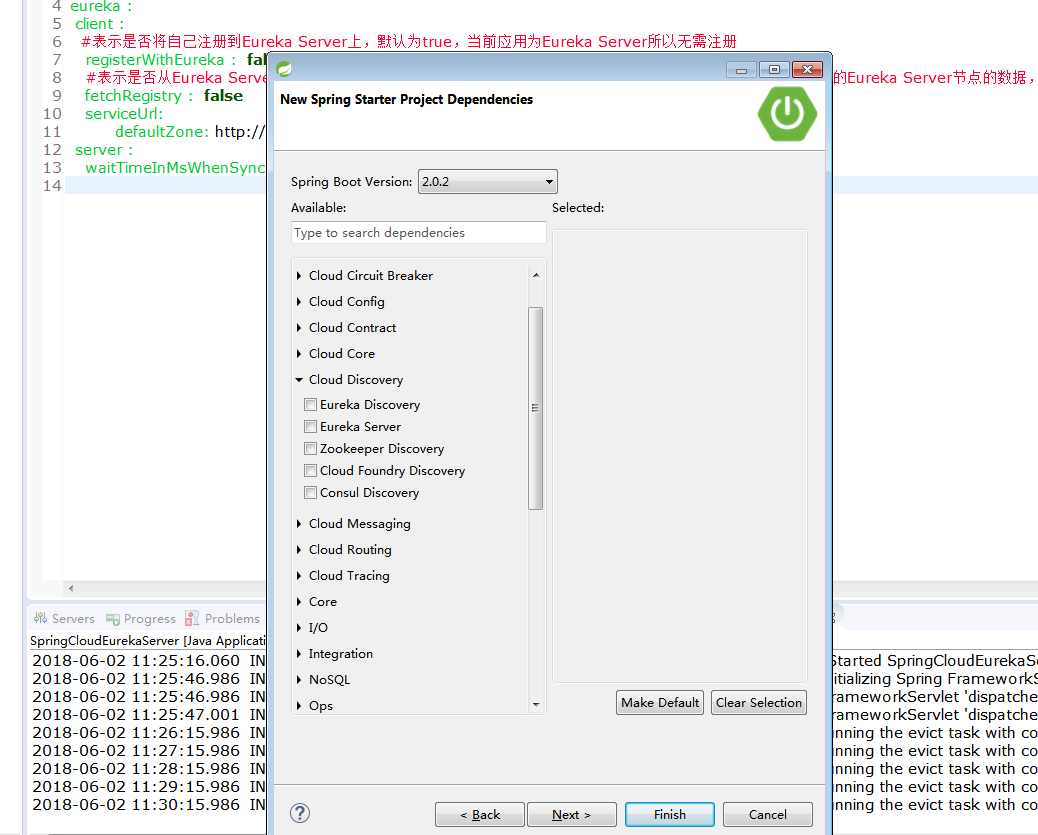
**7 启动Eureka server测试：**

启动sc-eureka-first-server-HA01，访问http://localhost:8761/ ,如下图所示:



**8 创建服务提供者**

1.服务提供者，为了演示在这里提供一个简单的订单查询服务，如工程sc-eureka-first-provider01和sc-eureka-first-provider02所示。



2.主程序入口代码，如下所示：

package org.xujin.sc.eureka.first.order;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

@SpringBootApplication

@EnableDiscoveryClient

// @EnableEurekaClient

public class OrderProviderSpringBootAppliaction {

public static void main(String[] args) {

SpringApplication.run(OrderProviderSpringBootAppliaction.class, args);

}

}

**Tips:如果使用Eureka, 可以使用@EnableEurekaClient注解，但是推荐使用@EnableDiscoveryClient代替@EnableEurekaClient注解，因为@EnableDiscoveryClient是一个高度的抽象， 来自于spring-cloud-commons， 由于Spring Cloud选型是中立的因此抽象出该接口当服务注册中心选型改变为Eureka，ZK，Consul时，不需要修改原有代码中的注解。**

**9 服务提供者暴露的服务-OrderController.java**

@RestController

public class OrderController {

@Autowired

private OrderService orderService;

@GetMapping("/sc/order/{id}")

public OrderModel findOrderById(@PathVariable Long id) {

OrderModel orderModel = orderService.findOrderByOrderId(id);

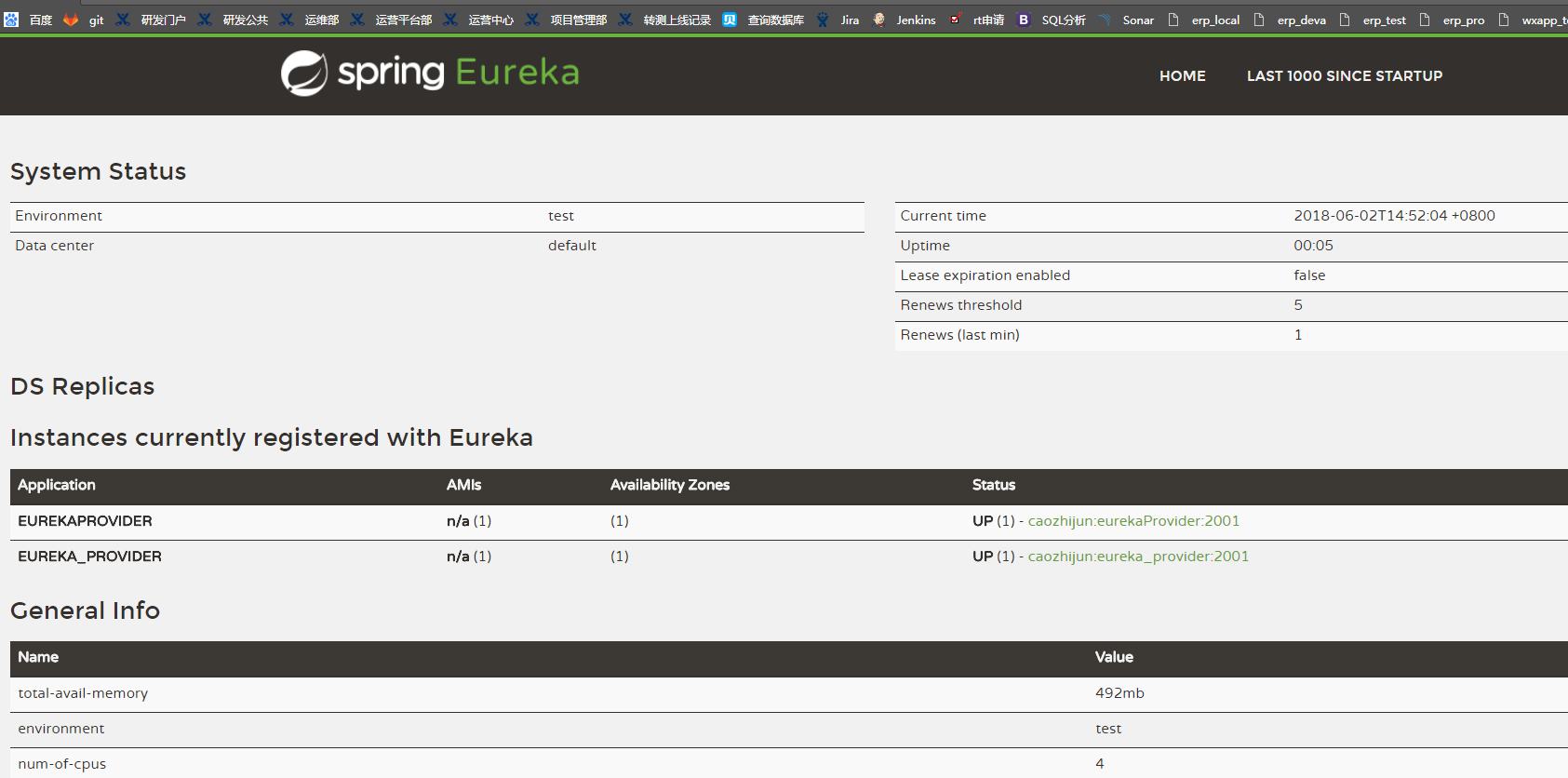
return orderModel;

}

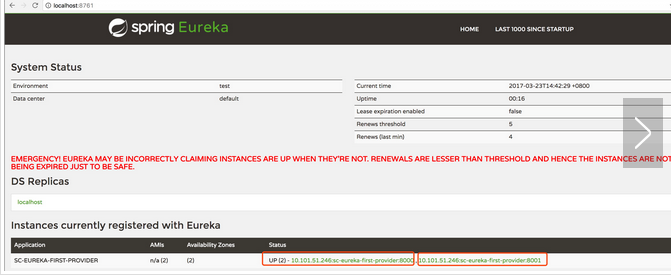
}

启动服务提供者，把服务注册信息，注册到Eureka Server注册中心

启动sc-eureka-first-provider01,当启动其中一个服务后刷新Eureka Server会出现安全模式,如下图所示:



启动sc-eureka-first-provider02，刷新Eureka Server如下图所示。



**10 服务消费者主要是一个简单的用户服务，用户服务查询订单服务的订单信息。**

**1）.引入相应的依赖**

<?xml version="1.0"?>

<project

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

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

<modelVersion>4.0.0</modelVersion>

<groupId>org.xujin.sc</groupId>

<artifactId>sc-eureka-first-consumer</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>sc-eureka-first-consumer</name>

<url>http://maven.apache.org</url>

<!-- 引入spring boot的依赖 -->

<parent>

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

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

<version>1.4.3.RELEASE</version>

</parent>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

</properties>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.projectlombok</groupId>

<artifactId>lombok</artifactId>

<version>1.16.6</version>

<scope>provided</scope>

</dependency>

</dependencies>

<!-- 引入spring cloud的依赖 -->

<dependencyManagement>

<dependencies>

<dependency>

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

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

<version>Camden.SR4</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<!-- 添加spring-boot的maven插件 -->

<build>

<plugins>

<plugin>

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

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

</plugin>

</plugins>

</build>

</project>

**2）.主程序入口代码**

package org.xujin.sc.eureka.user;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.cloud.client.discovery.EnableDiscoveryClient;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

//消费者端加入服务发现注解

@EnableDiscoveryClient

@SpringBootApplication

public class UserConsumerApplication {

@Bean

public RestTemplate restTemplate() {

return new RestTemplate();

}

public static void main(String[] args) {

SpringApplication.run(UserConsumerApplication.class, args);

}

}

**3）.消费者调用Controller。**

@RestController

public class UserController {

private static final Logger logger = LoggerFactory.getLogger(UserController.class);

@Autowired

private RestTemplate restTemplate;

@Autowired

private DiscoveryClient discoveryClient;

// discoveryClient获取服务列表中，应用名为sc-eureka-first-provider一个服务注册信息

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient

.getInstances("sc-eureka-first-provider");

if (list != null && list.size() > 0) {

return String.valueOf(list.get(0).getUri());

}

return null;

}

@GetMapping("/sc/user/{id}")

public Order findByIdByEurekaServer(@PathVariable Long id) {

String providerServiceUrl = serviceUrl();

return this.restTemplate.getForObject(providerServiceUrl + "sc/order/" + id,Order.class);

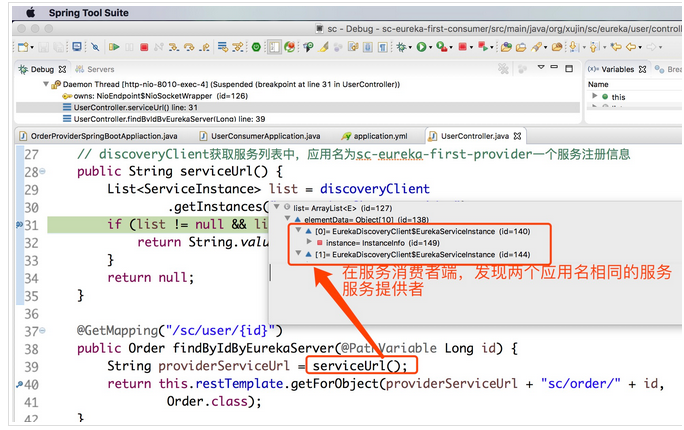
}

}

**如上述代码，所示使用discoveryClient.getInstances("sc-eureka-first-provider")获取服务名为sc-eureka-first-provider的服务注册列表信息**。

**4）.测试**

先后启动sc-eureka-first-consumer,如没有异常，打开浏览器访问:http://localhost:8010/sc/user/2 ,debug如下所示可以看到



**11 获取消费者获取服务端消费列表**

**使用EurekaClient获取服务注册信息**

@Autowired

private EurekaClient discoveryClient;

public String serviceUrl() {

InstanceInfo instance = discoveryClient.getNextServerFromEureka("STORES", false);

return instance.getHomePageUrl();

}

**使用DiscoveryClient获取服务注册信息**

@Autowired

private DiscoveryClient discoveryClient;

public String serviceUrl() {

List<ServiceInstance> list = discoveryClient.getInstances("STORES");

if (list != null && list.size() > 0 ) {

return list.get(0).getUri();

}

return null;

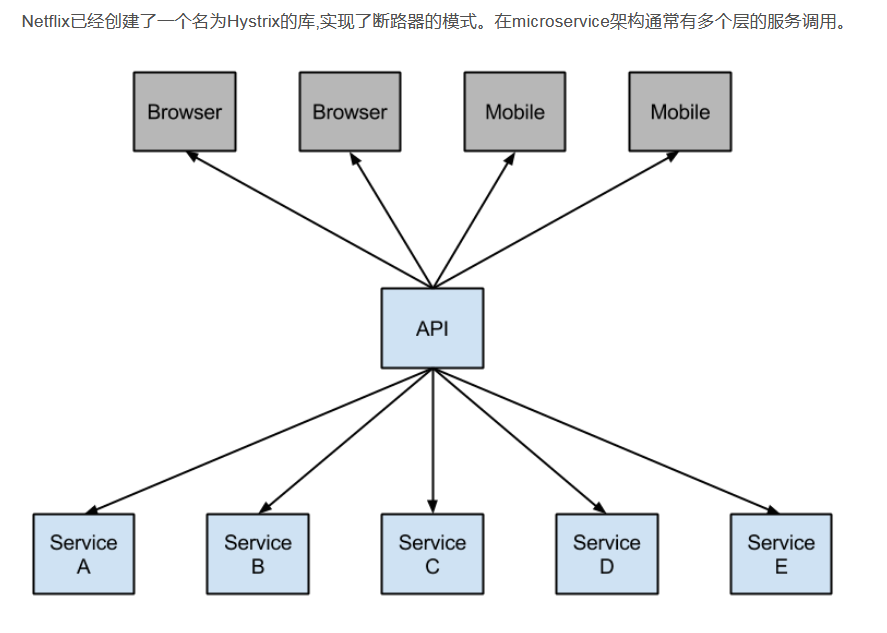
}



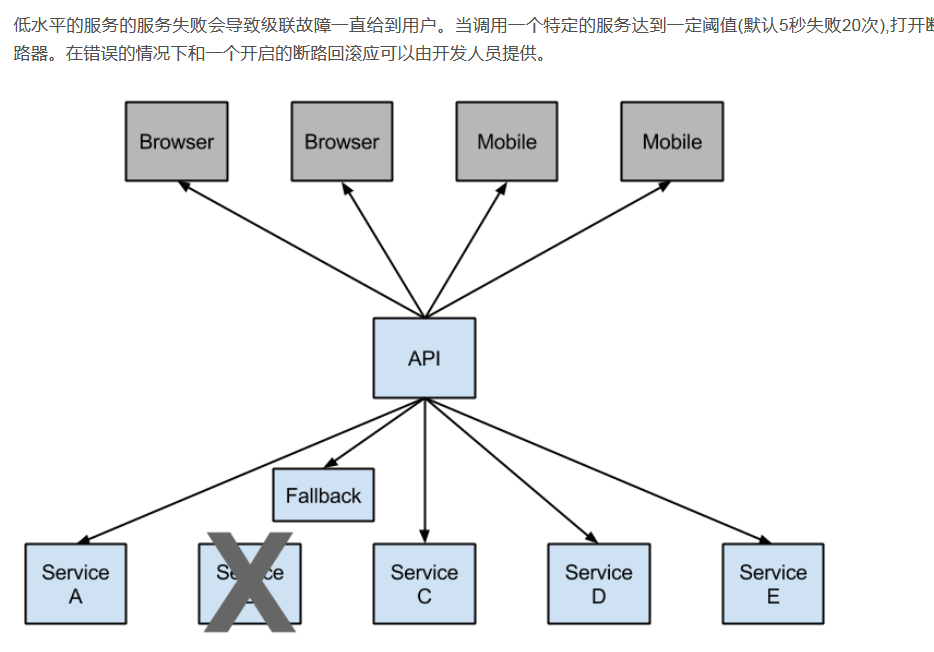


## 三、断路器

**1 SpringCloud Netflix实现了断路器库的名字叫Hystrix.** 在微服务架构下，通常会有多个层次的服务调用. 下面是微服架构下, 浏览器端通过API访问后台微服务的一个示意图：



2 一个微服务的超时失败可能导致瀑布式连锁反映，下图中，Hystrix通过自主反馈实现的断路器， 防止了这种情况发生。



图中的服务B因为某些原因失败，变得不可用，所有对服务B的调用都会超时。当对B的调用失败达到一个特定的阀值**(5秒之内发生20次失败是Hystrix定义的缺省值**), 链路就会被处于open状态， **之后所有所有对服务B的调用都不会被执行， 取而代之的是由断路器提供的一个表示链路open的Fallback消息**. Hystrix提供了相应机制，可以让开发者定义这个Fallbak消息.

open的链路阻断了瀑布式错误， 可以让被淹没或者错误的服务有时间进行修复。这个fallback可以是另外一个Hystrix保护的调用, 静态数据，或者合法的空值. Fallbacks可以组成链式结构，所以，最底层调用其它业务服务的第一个Fallback返回静态数据.

**3 在之前的两HELLO WORLD服务集群中加入断路器， 防止其中一个Hello world挂掉后， 导致系统发生连锁超时失败。**

**4 在maven工程**（前面章节中介绍的Ribbon或者Feign工程）的pom.xml中添加hystrix库支持断路器

<dependency>

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

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

</dependency>

**5 在Ribbon应用中使用断路器**

1). 在Spring Boot启动类上添加@EnableCircuitBreaker注解

@SpringBootApplication

@EnableDiscoveryClient

@EnableCircuitBreaker

public class ServiceRibbonApplication {

public static void main(String[] args) {

SpringApplication.run(ServiceRibbonApplication.class, args);

}

**2). 用@HystrixCommand注解标注访问服务的方法**

@Service

public class HelloService {

@Autowired RestTemplate restTemplate;

@HystrixCommand(fallbackMethod = "serviceFailure")

public String getHelloContent() {

return restTemplate.getForObject("http://SERVICE-HELLOWORLD/",String.class);

}

public String serviceFailure() {

return "hello world service is not available !";

}

}

@HystrixCommand注解定义了一个断路器，它封装了getHelloContant()方法， 当它访问的SERVICE-HELLOWORLD失败达到阀值后，将不会再调用SERVICE-HELLOWORLD， 取而代之的是返回由fallbackMethod定义的方法serviceFailure（）。@HystrixCommand注解定义的fallbackMethod方法，需要特别注意的有两点：

第一，  **fallbackMethod的返回值和参数类型需要和被@HystrixCommand注解的方法完全一致。**否则会在运行时抛出异常。比如本例中，serviceFailure（）的返回值和getHelloContant()方法的返回值都是String。

第二，  **当底层服务失败后，fallbackMethod替换的不是整个被@HystrixCommand注解的方法（本例中的getHelloContant), 替换的只是通过restTemplate去访问的具体服务。**可以从中的system输出看到， 即使失败，控制台输出里面依然会有“call SERVICE-HELLOWORLD”。

启动eureka服务，只启动两个Helloworld服务，然后中断其中一个（模拟其中一个微服务挂起），访问http://localhost:8901/然后刷新, 由于有负载均衡可以看到以下两个页面交替出现。可以看到第二个被挂起的服务，被定义在Ribbon应该里面的错误处理方法替换了。

**6 关于 hystrix 的异常 fallback method wasn't found**

这样就会出现如上所述的异常，这是因为指定的 备用方法 和 原方法 的参数个数，类型不同造成的,这样就可以解决上述的异常了。

@HystrixCommand(fallbackMethod = "fallbackHi")

public String getHi(String x) {

String msg = restTemplate.getForObject("http://jack/hi", String.class);

return msg;

}

public String fallbackHi(String x){

return "can't say hi, and get: " + x;

}

**7 在Feign应用中使用断路器**

1）. Feign内部已经支持了断路器，所以不需要想Ribbon方式一样，在Spring Boot启动类上加额外注解

2）. 用@FeignClient注解添加fallback类， 该类必须实现@FeignClient修饰的接口。

1 @FeignClient(name = "SERVICE-HELLOWORLD", fallback = HelloWorldServiceFailure.class)

2 public interface HelloWorldService {

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

4 public String sayHello();

5 }

3). 创建HelloWorldServiceFailure类， 必须实现被@FeignClient修饰的HelloWorldService接口。注意添加@Component或者@Service注解，在Spring容器中生成一个Bean

1 @Component

2 public class HelloWorldServiceFailure implements HelloWorldService {

3 @Override

4 public String sayHello() {

5 System.out.println("hello world service is not available !");

6 return "hello world service is not available !";

7 }

8 }

## 四、路由网关

1 在微服务架构中，需要几个关键的组件，服务注册与发现、服务消费、负载均衡、断路器、智能路由、配置管理等，由这几个组件可以组建一个简单的微服务架构。客户端的请求首先经过负载均衡（zuul、Ngnix），**再到达服务网关（zuul集群）**，然后再到具体的服务，服务统一注册到高可用的服务注册中心集群，服务的所有的配置文件由配置服务管理（下一篇文章讲述），配置服务的配置文件放在Git仓库，方便开发人员随时改配置。

2 基于Spring的微服务结点在能力上没有高低贵贱之分，但是在角色上会分为边缘服务和内部服务两部分。内部服务顾名思义是为对内暴露服务的结点，供架构内部来调用；边缘服务是对外部网络暴露的服务结点，也就是对外API接口。

**开发人员头疼的地方：**为了防止我的程序在网络上被人攻击，我们需要写各种权限机制，这些机制在每个微服务结点都要实现一次。一旦鉴权上有什么bug，又要全部节点上推倒重来，噩梦。

**运维人员头疼的地方：**边缘服务前段都会架一个F5或者Nginx等负载均衡的代理，需要手动维护一份服务列表和服务地址的路由信息，随着结点的扩展或地址调整这份列表要变来变去。

3 为了解决鉴权重复的问题，使业务结点本身只关心实现自己的业务，**将对权限的处理抽离到上层**。外部客户先请求到Zuul上，在Zuul服务上对权限进行统一实现和过滤，以实现微服务结点的过滤和验证。为了解决请求路由和安全过滤，Spring Cloud推出了一个API gateway组件：Spring Cloud Zuul。在路由方面，Zuul将自己作为一个微服务结点注册到Eureka上，就获取了所有微服务的实例信息，同时又以服务名为ContextPath的方式创建路由映射。

**4 Pom依赖**

<dependencies>

<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>

</dependencies>

**5 Application主程序**

@EnableZuulProxy

@SpringCloudApplication

public class DemoFeignApplication {

public static void main(String[] args) {

SpringApplication.run(DemoFeignApplication.class, args);

}

@Bean

public AccessFilter accessFilter() {

return new AccessFilter();

}

}

**新注解：@EnableZuulProxy，标识启动zuul网管控制；**

**创建过滤器的实例，启动过滤器AccessFilter。**

**6 application.yml参数配置：**

server:

port: 9053

spring:

application:

name: demo-zuul

eureka:

client:

serviceUrl:

defaultZone: http://peer1:1111/eureka/,http://peer2:1112/eureka/

zuul:

routes:

demo-feign-freeservice:

path: /api-service/\*\*

user-service:

path: /api-consumer/\*\*

serviceId: demo-feign-freeconsumer

163:

path: /163

url: http://www.163.com/

这里对路由的配置用了3种格式，体现了API Gateway的路由分发功能：

**第一种格式（path-url）：**如果请求/163这个地址，将会转发到http://www.163.com上去

**第二种(path-serviceId)：**如果请求/api-consumer开头的地址，将会转发到eureka上serviceId为“demo-feign-freeconsumer”这个服务上去。

**第三种(给微服务名指定path)：**给demo-feign-freeservice这个微服务指定了它的请求地址是/api-service/\*\*

**7 路由匹配规则：**

/user-service/? 很少用，支持/user-service/后拼接任意一个字符，例如/user-service/a、/user-service/1，不支持/user-service/123

/user-service/\* 较少用，支持/user-service/后拼接任意一个字符例如/user-service/abc、/user-service/1，不支持多级目录/user-service/abc/b

/user-service/\*\* 通用，只要以/user-service/开头就可以，后面路径没要求。

**8 过滤器代码：**

public class AccessFilter extends ZuulFilter{

@Override

public Object run() {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest request = ctx.getRequest();

Object accessToken = request.getParameter("accessToken");

if(accessToken==null) {

ctx.setSendZuulResponse(false);

ctx.setResponseStatusCode(401);

//为了被error过滤器捕获

ctx.set("error.status\_code", HttpServletResponse.SC\_INTERNAL\_SERVER\_ERROR);

ctx.set("error.exception",new RuntimeException("AccessToken is null"));

return null;

}

return null;

}

@Override

public boolean shouldFilter() {

RequestContext ctx = RequestContext.getCurrentContext();

HttpServletRequest request = ctx.getRequest();

if(request.getRequestURI().equals("/163")) {

return false;

}else {

return true;

}

}

@Override

public int filterOrder() {

return 0;

}

@Override

public String filterType() {

return "pre";

}

}

**9 继承抽象类zuulFilter，有4个方法需要实现：**

**filterType:过滤器类型**，决定了过滤器在哪个周期生效。类型有pre、route、post、error，对应Spring AOP里的前加强、前后加强、后加强、异常处理。

**filterOrder:过滤器的执行顺序**，多个过滤器同时存在时根据这个order来决定先后顺序，越小优先级越高

**shouldFilter:过滤器是否被执行**，只有true时才会执行run()里的代码。我们这里除开访问163会放行其他情况都需要进行过滤判断，在生产环境一般是要根据函数条件来判断的。

**run:过滤逻辑**，整个代码的zuul的代码逻辑很简单，对外统一的访问路径是zuul服务的地址，如果直接访问163过滤器不生效直接放行，访问其它内部服务结点需要判断是否有accessToken。

由于我内部服务设置了随机sleep时间，测试时发现请求consumer的时候页面会TIMEOUT，检查日志发现service和consumer都正常，再回来看zuul的日志：

Caused by: com.netflix.hystrix.exception.HystrixRuntimeException:demo-feign-freeservice timed-out and no fallback available.

再回头细想Zuul的原理，它包含了Hystrix的部分，怀疑是我被请求的服务处理超时了引起了客户端Zuul的服务降级，但是我又没给Zuul开发fallback代码，所以请求失败报了这个错。

为了验证我的观点给zuul的application.yml添加如下内容：

hystrix:

command:

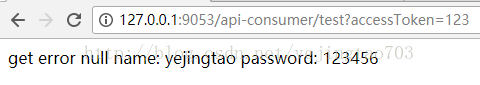
default:

execution:

timeout:

enabled: false

重启Zuul后再试下：搞定



再验证下zuul默认的路由规则：

http://127.0.0.1:9053/demo-feign-freeservice/feign-service/serviceGet?name=yuanyuan&accessToken=123

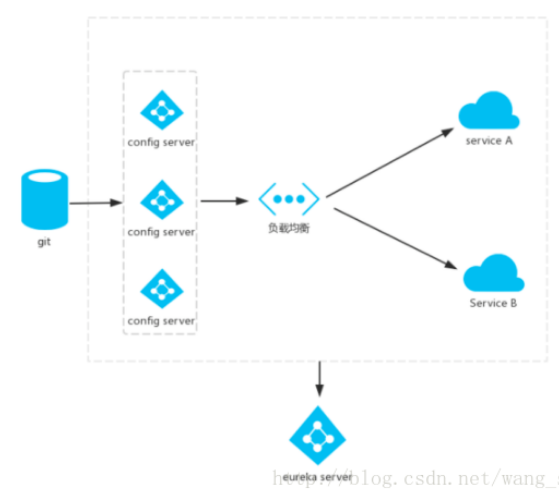
http://127.0.0.1:9053/api-service/feign-service/serviceGet?name=yuanyuan&accessToken=123

两个是等效的。

## 五、高可用的分布式配置中心

**1 下面我们通过整合Eureka来实现配置中心的高可用，因为作为架构内的配置管理，本身其实也是可以看作架构中的一个微服务，我们可以把config server也注册为服务，这样所有客户端就能以服务的方式进行访问。通过这种方法，只需要启动多个指向同一Gitlab仓库位置的config server端就能实现高可用了。**

2 配置中心如何从远程git读取配置文件，当服务实例很多时，都从配置中心读取文件，这时可以考虑将配置中心做成一个微服务，将其集群化，从而达到高可用，架构图如下：



**3 新建一个eureka server工程，命名为config-eureka-server,作为配置服务的注册中心。pom文件如下：**

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

<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.springcloud</groupId>

<artifactId>config-eureka-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>config-eureka-server</name>

<description>Demo project for Spring Boot</description>

<parent>

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

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

<version>1.5.9.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>Edgware.RELEASE</spring-cloud.version>

</properties>

<dependencies>

<dependency>

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

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

<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>

</project>

**4 入口类加入相应注解：**

@EnableEurekaServer

@SpringBootApplication

public class ConfigEurekaServerApplication {

public static void main(String[] args) {

SpringApplication.run(ConfigEurekaServerApplication.class, args);

}

}

**5 修改配置文件如下：**

server.port=8889

eureka.instance.hostname=localhost

#配置成服务

eureka.client.registerWithEureka= false

eureka.client.fetchRegistry=false

eureka.client.service-url.defaultZone= http://${eureka.instance.hostname}:${server.port}/eureka/

**6 修改之前的config-server工程在pom中加入需要的依赖；**

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

**7 修改配置文件：**

spring.application.name=config-server

server.port=8888

#配置文件在本地

#spring.profiles.active=native

#配置文件的目录

#spring.cloud.config.server.native.search-locations=D:/workspace/GitHub/spring-config/config-file

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

spring.cloud.config.server.git.uri=https://github.com/iot-wangshuyu/springcloud

#配置仓库路径

spring.cloud.config.server.git.searchPaths=SpringcloudConfig/respo/

#配置仓库的分支

spring.cloud.config.label=master

#访问git仓库的用户名

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

#访问git仓库的用户密码

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

#注册服务地址

eureka.client.service-url.defaultZone=http://localhost:8889/eureka/

**8 修改启动类：**

@SpringBootApplication

@EnableConfigServer

@EnableEurekaClient

public class ConfigServerApplication {

public static void main(String[] args) {

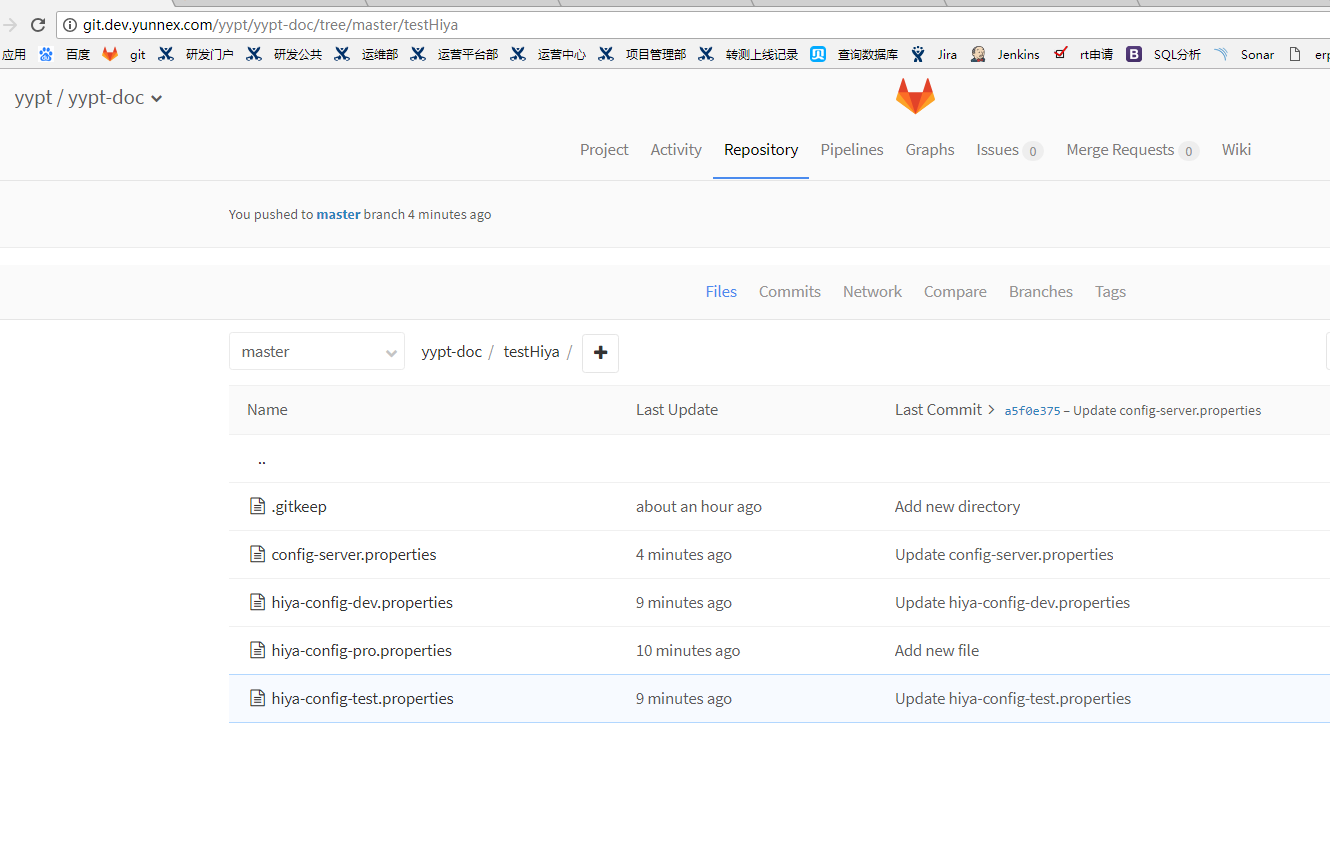
SpringApplication.run(ConfigServerApplication.class, args);

}

}

**9 http请求地址和资源文件映射如下:**

* /{application}/{profile}[/{label}]
* /{application}-{profile}.yml
* /{label}/{application}-{profile}.yml
* /{application}-{profile}.properties
* /{label}/{application}-{profile}.properties







**10 修改config-client工程pom中添加相应的依赖：**

<dependency>

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

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

<dependency>

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

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

</dependency>

**11 修改配置文件名称为：bootstrap.properties**

#服务名称

spring.application.name=config-client

#仓库分支

spring.cloud.config.label=master

#dev开发环境配置文件

#test测试环境

#pro正式环境

spring.cloud.config.profile=dev

#使用eureka server的时候需要注释掉

#spring.cloud.config.uri= http://localhost:8888/

#服务注册地址

eureka.client.service-url.defaultZone=http://localhost:8889/eureka/

#从远程获取配置文件

spring.cloud.config.discovery.enabled=true

#订阅的服务名称

spring.cloud.config.discovery.service-id=config-server

server.port=8881

如果配置服务部署多份，通过负载均衡，从而高可用。

依次启动程序，config-eureka-server,config-server,config-client,

浏览器访问：http://localhost:8889/可以查看服务注册情况；

浏览器访问：http://localhost:8881/hi；返回配置文件中的foo属性；

说明高可用分布式配置中心配置成功



**12 Spring boot 嵌入的tomcat不能启动： Unregistering JMX-exposed beans on shutdown**

在官网(http://start.spring.io/)上generate project，将生成的项目导入idea中，运行之后控制台输出“Unregistering JMX-exposed beans on shutdown”，tomcat也没有运行。寻找原因，看了下pom.xml文件中tomcat依赖关系如下：

<dependency>

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

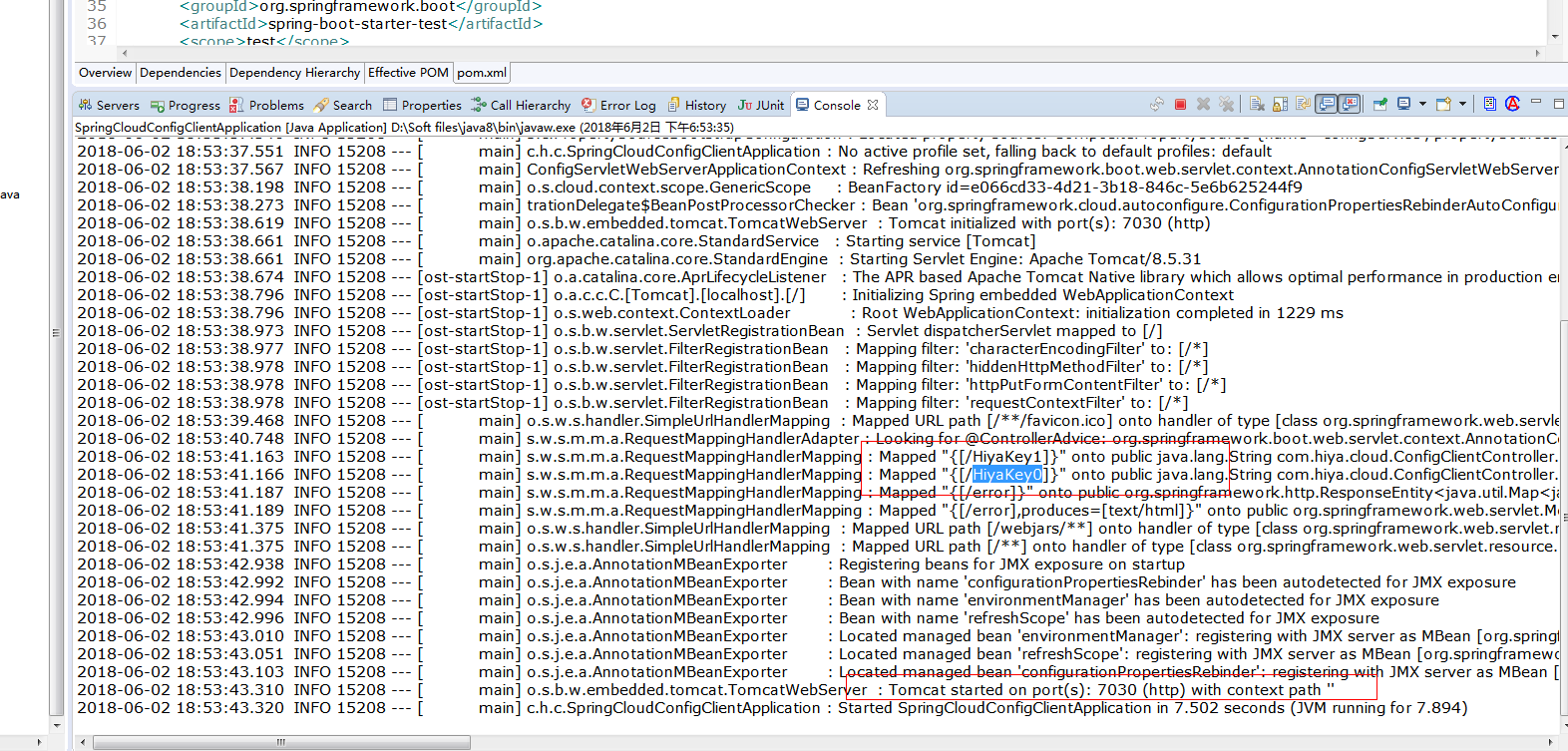
<artifactId>spring-boot-starter-tomcat</artifactId>

<scope>provided</scope>

</dependency>

重新运行项目便可启动嵌入的tomcat服务器：Tomcat started on port(s): 8080 (http)

**13 Spring boot启动日志：**



**六、消息总线**

**1 简介**

在微服务架构的系统中，我们通常会**使用轻量级的消息代理来构建一个共用的消息主题让系统中所有微服务实例都连接上来**，由于该主题中产生的消息会被所有实例监听和消费，所以我们称它为消息总线。

**spring CloudBus 将分布式的节点和轻量的消息代理连接起来。这可以用于广播配置文件的更改或者其他的管理工作。一个关键的思想就是，消息总线可以为微服务做监控，也可以作为应用程序之间相互通讯。本文要讲述的是用AMQP实现通知微服务架构的配置文件的更改。**

**2 消息代理**

消息代理(Message Broker)是一种消息验证、传输、路由的架构模式。它在应用程序之间起到通信调度并最小化应用之间的依赖的作用，使得应用程序可以高效地解耦通信过程。消息代理是一个中间件产品，它的核心是一个消息的路由程序，用来实现接收和分发消息， 并根据设定好的消息处理流来转发给正确的应用。 它包括独立的通信和消息传递协议，能够实现组织内部和组织间的网络通信。设计代理的目的就是为了能够从应用程序中传入消息，并执行一些特别的操作，下面这些是在企业应用中，我们经常需要使用消息代理的场景：

1.将消息路由到一个或多个目的地。

2.消息转化为其他的表现方式。

3.执行消息的聚集、消息的分解，并将结果发送到它们的目的地，然后重新组合响应返回给消息用户。

4.调用Web服务来检索数据。

5.响应事件或错误。

6.使用发布－订阅模式来提供内容或基千主题的消息路由。

目前已经有非常多的开源产品可以供大家使用， 比如：

**ActiveMQ**

**Kafka**

**RabbitMQ**

**RocketMQ**

**3 SpringCloud+RabbitMQ**

（1）RabbitMQ简介、安装不赘述。

（2）pom.xml

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

（3）application.yml

spring:

application:

name: rabbitmq-hello

rabbitmq:

host: \*\*\*.\*\*\*.\*\*\*.\*\*\*

port: 5672

username: guest

password: guest

（4）发送者Sender

@Component

public class Sender {

private static final Logger log = LoggerFactory.getLogger(Sender.class);

@Autowired

private AmqpTemplate amqpTemplate;

public void send() {

String context = "hello " + new Date();

log.info("Sender : " + context);

this.amqpTemplate.convertAndSend("hello", context);

}

}

（5）接受者Receiver

@Component

@RabbitListener(queues = "hello")

public class Receiver {

private static final Logger log = LoggerFactory.getLogger(Receiver.class);

@RabbitHandler

public void process(String hello) {

log.info("Receiver : " + hello);

}

}

（6）创建RabbitMQ的配置类 RabbitConfig

@Configuration

public class RabbitConfig {

@Bean

public Queue helloQueue(){

return new Queue("hello");

}

}

（7）创建单元测试类， 用来调用消息生产

@RunWith(SpringJUnit4ClassRunner.class)

@SpringBootTest(classes = SpringcloudbusrabbitmqApplication.class)

public class HelloApplicationTests {

@Autowired

private Sender sender;

@Test

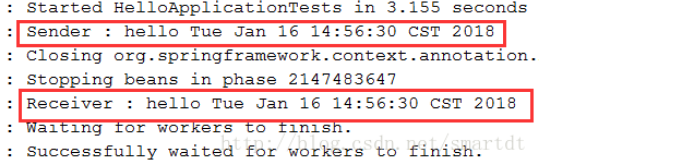
public void hello() throws Exception {

sender.send();

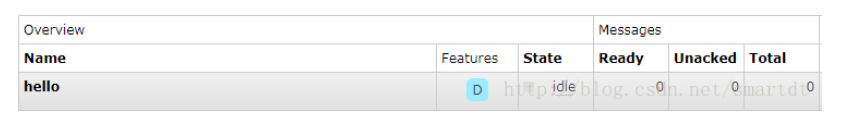
}

}

（8）测试，执行HelloApplicationTests



（9）访问host:15672



**4 改造Config-Client（整合springcloud bus）**

（1）pom.xml

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

</dependency>

<dependency>

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

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

<scope>test</scope>

</dependency>

</dependencies>

（2）bootstrap.properties

spring.application.name=configspace

spring.cloud.config.label=master

spring.cloud.config.profile=dev

spring.cloud.config.uri= http://localhost:5588/

eureka.client.serviceUrl.defaultZone=http://localhost:5555/eureka/

server.port=5589

spring.rabbitmq.host=118.89.237.88

spring.rabbitmq.port= 5672

spring.rabbitmq.username=guest

spring.rabbitmq.password=guest

management.security.enabled=false

（3）其他不用改变

**5 测试**

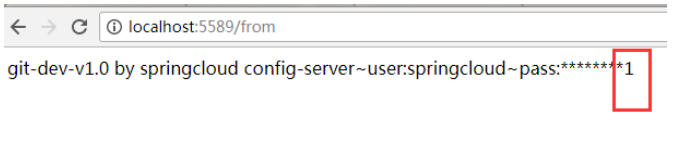
（1）测试准备

一个服务注册中心，EUREKASERVER，端口为5555；

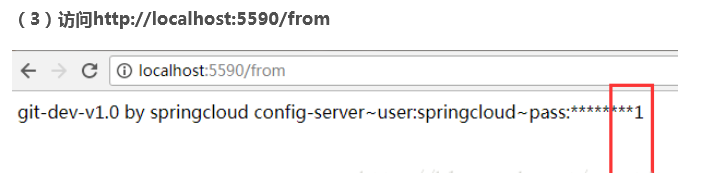
一个分布式配置中心，ConfigServer，端口为5588；

二个分布式配置，ConfigClient，端口为5589、5590；

（2）访问http://localhost:5589/from



（3）访问http://localhost:5590/from



## 七、docker部署spring cloud

**1 docker简介**

Docker是一个开源的引擎，**可以轻松的为任何应用创建一个轻量级的、可移植的、自给自足的容器。**开发者在笔记本上编译测试通过的容器可以批量地在生产环境中部署，包括VMs（虚拟机）、bare metal、OpenStack 集群和其他的基础应用平台。

**2 Docker通常用于如下场景**：web应用的自动化打包和发布；自动化测试和持续集成、发布；

在服务型环境中部署和调整数据库或其他的后台应用；从头编译或者扩展现有的OpenShift或Cloud Foundry平台来搭建自己的PaaS环境。

**3 Docker 的优点**

**1、简化程序**：   
Docker 让开发者可以打包他们的应用以及依赖包到一个可移植的容器中，然后发布到任何流行的 Linux 机器上，便可以实现虚拟化。Docker改变了虚拟化的方式，使开发者可以直接将自己的成果放入Docker中进行管理。方便快捷已经是 Docker的最大优势，过去需要用数天乃至数周的 任务，在Docker容器的处理下，只需要数秒就能完成。

**2、避免选择恐惧症：**   
如果你有选择恐惧症，还是资深患者。Docker 帮你 打包你的纠结！比如 Docker 镜像；Docker 镜像中包含了运行环境和配置，所以 Docker 可以简化部署多种应用实例工作。比如 Web 应用、后台应用、数据库应用、大数据应用比如 Hadoop 集群、消息队列等等都可以打包成一个镜像部署。

**3、节省开支：**   
一方面，云计算时代到来，使开发者不必为了追求效果而配置高额的硬件，Docker 改变了高性能必然高价格的思维定势。Docker 与云的结合，让云空间得到更充分的利用。不仅解决了硬件管理的问题，也改变了虚拟化的方式。

**4 准备工作**

环境条件：

linux系统，不建议windows

docker最新版本

jdk 1.8

maven3.0

**5 改造工程、构建镜像**

改造eureka-server工程在pom文件加上插件：

<build>

<plugins>

<plugin>

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

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

</plugin>

<!-- tag::plugin[] -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.3</version>

<configuration>

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

<dockerDirectory>src/main/docker</dockerDirectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

</configuration>

</plugin>

<!-- end::plugin[] -->

</plugins>

</build>

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

imageName指定了镜像的名字，本例为 forep/eureka-server

dockerDirectory指定 Dockerfile 的位置

resources是指那些需要和 Dockerfile 放在一起，在构建镜像时使用的文件，一般应用 jar 包需要纳入。

**修改下配置文件：**

server:

port: 8761

eureka:

instance:

prefer-ip-address: true

client:

registerWithEureka: false

fetchRegistry: false

**编写dockerfile文件：**

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD eureka-server-0.0.1-SNAPSHOT.jar app.jar#RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

EXPOSE 8761

**docker file编写指令：**

FROM

FROM <image>

FROM <image>:<tag>

FROM <image> <digest>

FROM指令必须指定且需要在Dockerfile其他指令的前面，指定的基础image可以是官方远程仓库中的，也可以位于本地仓库。后续的指令都依赖于该指令指定的image。当在同一个Dockerfile中建立多个镜像时，可以使用多个FROM指令。

VOLUME

格式为：

**VOLUME ["/data"]**

使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也可以共享给其他容器。当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令。

ADD

从src目录复制文件到容器的dest。其中src可以是Dockerfile所在目录的相对路径，也可以是一个URL，还可以是一个压缩包

ENTRYPOINT

指定Docker容器启动时执行的命令，可以多次设置，但是只有最后一个有效。

EXPOSE

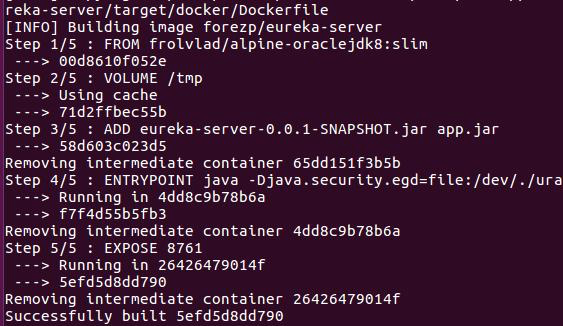
为Docker容器设置对外的端口号。在启动时，可以使用-p选项或者-P选项。

构建镜像

执行构建docker镜像maven命令：

mvn clean

mvn package docker:build



构建eureka-server镜像成功。

同理构建service-hi镜像

pom文件导入同eurek-server

修改下配置文件：

eureka:

client:

serviceUrl:

defaultZone: http://eureka-server:8761/eureka/ # 这个需要改为eureka-serverserver:

port: 8763spring:

application:

name: service-hi

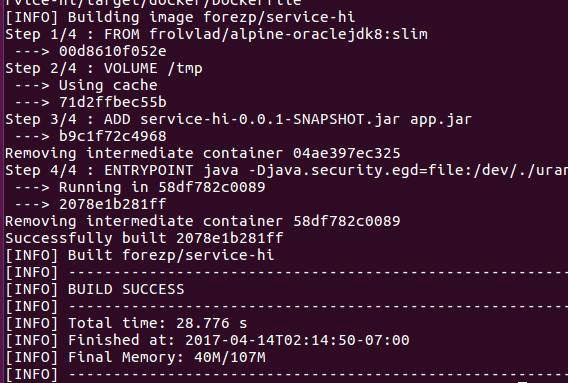
在这里说下：defaultZone发现服务的host改为镜像名。

dockefile 编写同eureka-server

**6 构建镜像：**

mvn clean

mvn package docker:build

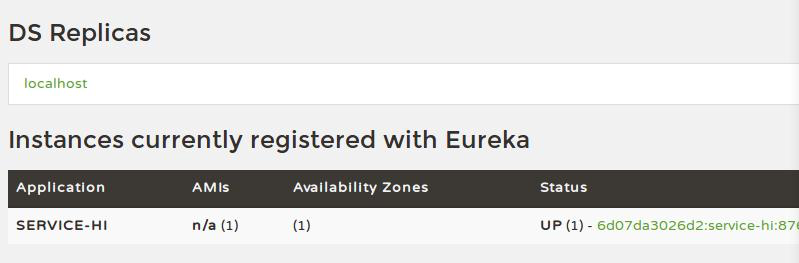


这时我们运行docke的eureka-server 和service-hi镜像：

docker run -p 8761: 8761 -t forezp/eureka-server

docker run -p 8763: 8763 -t forezp/service-hi

**7 访问localhost:8761**



**8 采用docker-compose启动镜像**

Compose 是一个用于定义和运行多容器的Docker应用的工具。使用Compose，你可以在一个配置文件（yaml格式）中配置你应用的服务，然后使用一个命令，即可创建并启动配置中引用的所有服务。下面我们进入Compose的实战吧。

采用docker-compose的方式编排镜像，启动镜像：

version: '3'

services:

eureka-server:

image: forezp/eureka-server

restart: always

ports:

- 8761:8761

service-hi:

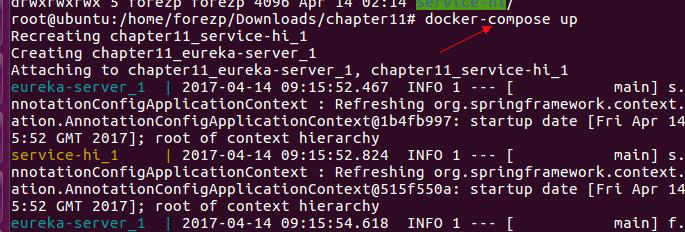
image: forezp/service-hi

restart: always

ports:

- 8763:8763

输入命令： docker-compose up



发现2个镜像按照指定的顺序启动了。

源码下载：[https://github.com/forezp/SpringCloudLearning/tree/master/chapter11](https://github.com/forezp/SpringCloudLearning/tree/master/chapter11" \t "https://blog.csdn.net/forezp/article/details/_blank)

**9 采用docker-compose编排并启动镜像**

docker-compose也可以构建镜像，现在我们采用docker-compose的方式构建镜像。

现在以eureka-server为例：   
将Dockerfile移到eureka-server的主目录，改写ADD的相对路径：

FROM frolvlad/alpine-oraclejdk8:slim

VOLUME /tmp

ADD ./target/eureka-server-0.0.1-SNAPSHOT.jar app.jar#RUN bash -c 'touch /app.jar'

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-jar","/app.jar"]

EXPOSE 8761

同理修改service-hi目录；

编写构建镜像docker-compose-dev文件：

version: '3'

services:

eureka-server:

build: eureka-server

ports:

- 8761:8761

service-hi:

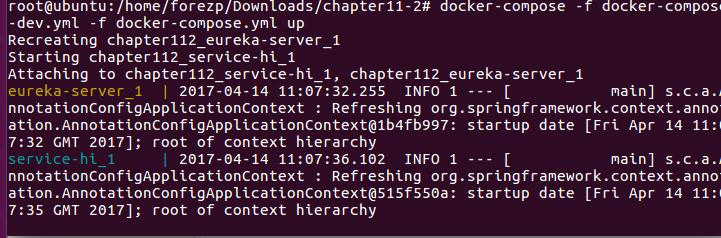
build: service-hi

ports:

- 8763:8763

**10 命令构建镜像并启动：**

docker-compose -f docker-compose.yml -f docker-compose-dev.yml up



## 八、Ribbon负载均衡

Netfilx发布的负载均衡器，是一个基于**http、tcp**的客户端负载均衡工具，具有控制http、tcp客户端的行为，为ribbon配置服务提供者的地址后，ribbon就可以经过springCloud的封装实心客户端负载均衡的服务调用。**负载均衡主要是实现对系统的高可用、网络压力的缓解、处理能力的伸缩。对于数据流量过大，往往单一设备是无法承担的，需要多台的设计进行分流。**

**1.软负载均衡**

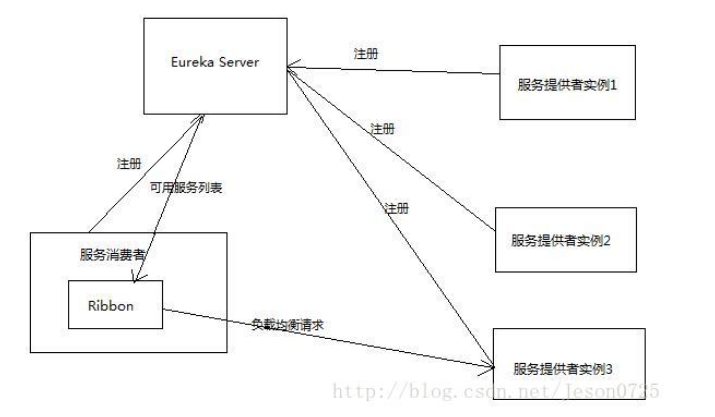
在一台机器上安装附加的某种软件，如nginx负载均衡，配置简单、成本低。根据部署的应用于系统的状态来分配资源进行负载、负载的能力不过受限于机器本身，性能越好，负载能力越大。

**2.硬负载均衡**

通过服务器和外部网络间安装负载均衡的设备，称为"负载均衡器"，硬件的负载均衡在功能想、性能上往往高于软负载均衡，不过价格昂贵，例：F5负载均衡器。能够通过智能交换机来实现负载，负载的能力与系统、应用无关，主要是通过网络层来判断，比如某时候系统处理能力已经不行了，但是可以通过网络来进行分配，成本高，除设备价格高昂，而且配置冗余．很难想象后面服务器做一个集群，但最关键的负载均衡设备却是单点配置；无法有效掌握服务器及应用状态。

**3 服务器端负载均衡：**例如Nginx，通过Nginx进行负载均衡，先发送请求，然后通过负载均衡算法，在多个服务器之间选择一个进行访问；即在服务器端再进行负载均衡算法分配。

**4 客户端负载均衡：**例如spring cloud中的ribbon，客户端会有一个服务器地址列表，在发送请求前通过负载均衡算法选择一个服务器，然后进行访问，这是客户端负载均衡；即在客户端就进行负载均衡算法分配。



**5 Ribbon自带负载均衡策略比较**

|  |  |  |  |
| --- | --- | --- | --- |
| 策略名 | 策略声明 | 策略描述 | 实现说明 |
| **BestAvailableRule** | public class BestAvailableRule extends ClientConfigEnabledRoundRobinRule | 选择一个最小的并发请求的server | 逐个考察Server，如果Server被tripped了，则忽略，在选择其中ActiveRequestsCount最小的server |
| **AvailabilityFilteringRule** | public class AvailabilityFilteringRule extends PredicateBasedRule | 过滤掉那些因为一直连接失败的被标记为circuit tripped的后端server，并过滤掉那些高并发的的后端server（active connections 超过配置的阈值） | 使用一个AvailabilityPredicate来包含过滤server的逻辑，其实就就是检查status里记录的各个server的运行状态 |
| **WeightedResponseTimeRule** | public class WeightedResponseTimeRule extends RoundRobinRule | 根据相应时间分配一个weight，相应时间越长，weight越小，被选中的可能性越低。 | 一个后台线程定期的从status里面读取评价响应时间，为每个server计算一个weight。Weight的计算也比较简单responsetime 减去每个server自己平均的responsetime是server的权重。当刚开始运行，没有形成statas时，使用roubine策略选择server。 |
| **RetryRule** | public class RetryRule extends AbstractLoadBalancerRule | 对选定的负载均衡策略机上重试机制。 | 在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server |
| **RoundRobinRule** | public class RoundRobinRule extends AbstractLoadBalancerRule | roundRobin方式轮询选择server | 轮询index，选择index对应位置的server |
| **RandomRule** | public class RandomRule extends AbstractLoadBalancerRule | 随机选择一个server | 在index上随机，选择index对应位置的server |
| **ZoneAvoidanceRule** | public class ZoneAvoidanceRule extends PredicateBasedRule | 复合判断server所在区域的性能和server的可用性选择server | 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），AvailabilityPredicate用于过滤掉连接数过多的Server。 |

**6 启动Eureka Server**

请参考该例：

[spring cloud中启动Eureka Server](http://blog.csdn.net/liuchuanhong1/article/details/54666715" \t "https://blog.csdn.net/liuchuanhong1/article/details/_blank)

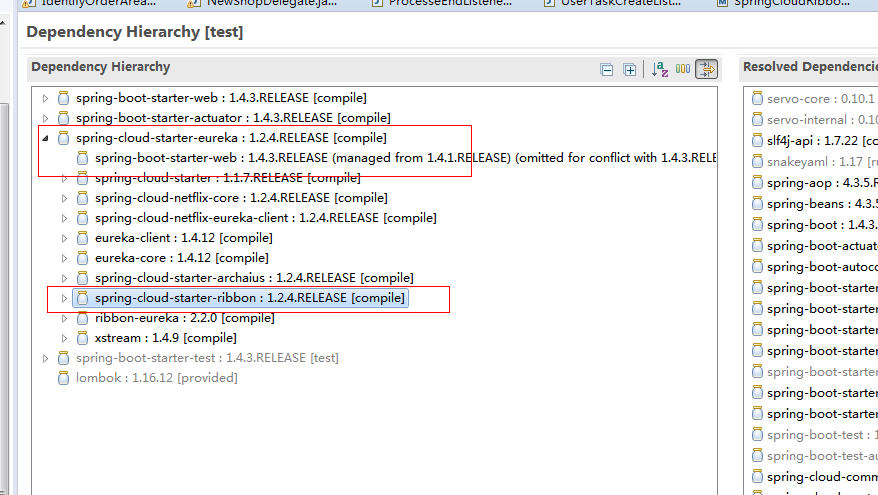
**7 启动微服务，并注册到Eureka Server上**

[spring cloud-将spring boot服务注册到Eureka Server上](http://blog.csdn.net/liuchuanhong1/article/details/54666777" \t "https://blog.csdn.net/liuchuanhong1/article/details/_blank)

为了演示负载均衡的效果，再启动一个为服务，注意需要将端口号改成不一致

**8 添加Ribbon支持**

1、添加Ribbon的依赖



2、添加负载均衡支持

package com.chhliu.springboot.restful;

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

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.boot.web.client.RestTemplateBuilder;

import org.springframework.cloud.client.loadbalancer.LoadBalanced;

import org.springframework.cloud.netflix.eureka.EnableEurekaClient;

import org.springframework.context.annotation.Bean;

import org.springframework.web.client.RestTemplate;

@SpringBootApplication

@EnableEurekaClient

public class SpringbootRestTemplateApplication {

    @Autowired

    private RestTemplateBuilder builder;

     @Bean

    @LoadBalanced  // 添加负载均衡支持，很简单，只需要在RestTemplate上添加@LoadBalanced注解，那么RestTemplate即具有负载均衡的功能,如果不加@LoadBalanced注解的话，会报java.net.UnknownHostException:springboot-h2<span style="font-size:12px;"><span style="font-family:'微软雅黑';">异常，此时无法通过注册到Eureka Server上的服务名来调用服务，因为RestTemplate是无法从服务名映射到ip:port的，映射的功能是由LoadBalancerClient来实现的。</span></span><span style="font-family:'微软雅黑';font-size:14px;">

</span>

    public RestTemplate restTemplate() {

        return builder.build();

    }

    public static void main(String[] args) {

        SpringApplication.run(SpringbootRestTemplateApplication.class, args);

    }

}

3、修改调用微服务的URL

package com.chhliu.springboot.restful.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 org.springframework.web.client.RestTemplate;

import com.chhliu.springboot.restful.vo.User;

@RestController

public class RestTemplateController {

    @Autowired

    private RestTemplate restTemplate;

    @GetMapping("/template/{id}")

    public User findById(@PathVariable Long id) {// 将原来的ip:port的形式，改成注册到Eureka Server上的应用名即可

        User u = this.restTemplate.getForObject("http://springboot-h2/user/" + id, User.class);

        System.out.println(u);

        return u;

    }

}

**9 查看Eureka Server状态**



**10 在浏览器中，多次刷新http://localhost:7904/template/2地址**

**11 测试结果**

7900端口服务：

[java] [view plain](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "view plain) [copy](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "copy)

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

7901端口服务：

[java] [view plain](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "view plain) [copy](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "copy)

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

Hibernate: select user0\_.id as id1\_0\_0\_, user0\_.age as age2\_0\_0\_, user0\_.balance as balance3\_0\_0\_, user0\_.name as name4\_0\_0\_, user0\_.username as username5\_0\_0\_ from user user0\_ where user0\_.id=?

7904端口服务：

[java] [view plain](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "view plain) [copy](https://blog.csdn.net/liuchuanhong1/article/details/54691566" \o "copy)

User [id=2, username=user2, name=李四, age=20, balance=100.00]

2017-01-23 09:58:05.682  INFO 7436 --- [erListUpdater-0] c.netflix.config.ChainedDynamicProperty  : Flipping property: springboot-h2.ribbon.ActiveConnectionsLimit to use NEXT property: niws.loadbalancer.availabilityFilteringRule.activeConnectionsLimit = 2147483647

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

User [id=2, username=user2, name=李四, age=20, balance=100.00]

从上面的测试结果可以看出，总共调了7904端口服务9次，其中7904端口服务调7900端口服务4次，调7901端口5次，刚好是9次  
经过上面的几个步骤，就基本使用Ribbon实现了客户端负载均衡的功能