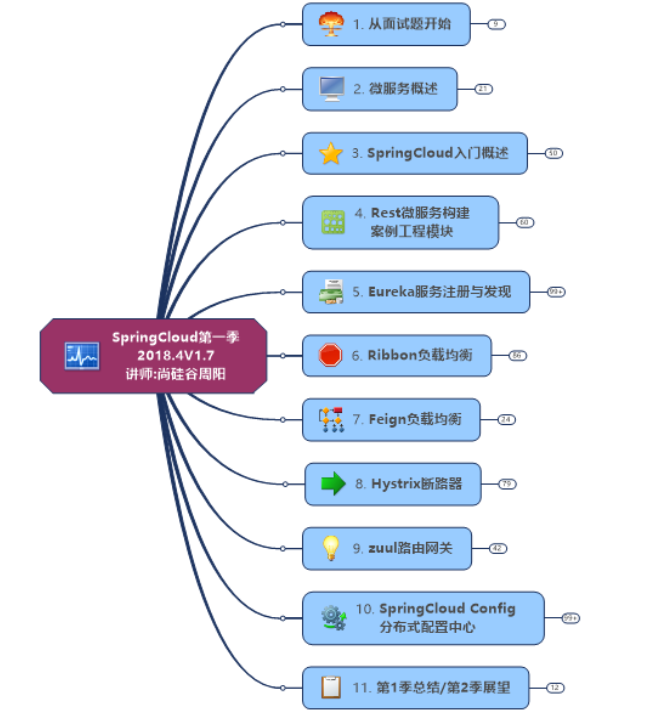
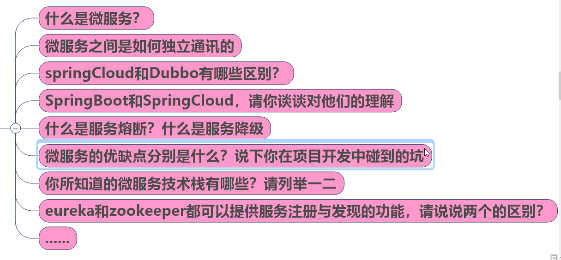
# 第一章：微服务架构概述

大纲：



常见面试题：



## 微服务的优点：

1. 易于开发和维护：一个微服务一个特定的业务功能，开发和维护单个微服务相对简单，整个应用由若干个微服务构建而成。单个微服务启动较快。局部修改容易部署。
2. 技术栈不受限，结合项目业务和团队特点合理选择技术栈。
3. 按需伸缩，根据需求，实现细粒度的扩展。

## 微服务架构面临的挑战

1. 运维要求提高，必须保证几百个服务的正常运行与协作
2. 分布式固有的复杂性：系统容错、网络延迟、分布式事务
3. 接口挑战成本高，如果修改某一个微服务的API，可能所有使用了该接口的微服务都需要调整
4. 重复劳动，某些服务可能使用到相同的功能，但这个功能还未达到分解为一个微服务的程度，则各个服务都需开发这个功能，代码重复。

## 微服务的设计原则

1. 单一职责原则：同一个单元应只关注整个系统功能中单独的一部分。
2. 服务自治原则：每个微服务应具备独立的业务能力、依赖与运行环境。
3. 轻量级通信机制：微服务之间应该通过轻量级的通信机制进行交互。可以跨语言跨平台，例如REST协议。而相反Java的RMI协议不符合要求因其绑定Java语言。常用协议有：REST、AMQP、STOMP、MQTT等。
4. 微服务粒度：如何根据实际情况使用合理的粒度划分微服务。

## 实现微服务架构

选择使用SpringCloud作为微服务开发框架，具备开箱即用的生产特性，可大大提升开发效率；为微服务架构提供了完整的解决方案。其他开发框架或方案有：Dubbo、Dropwizard、Armda等。

## 微服务技术栈

服务开发：SpringBoot、Spring、SpringMVC

服务治理：Arachaius、Diamond

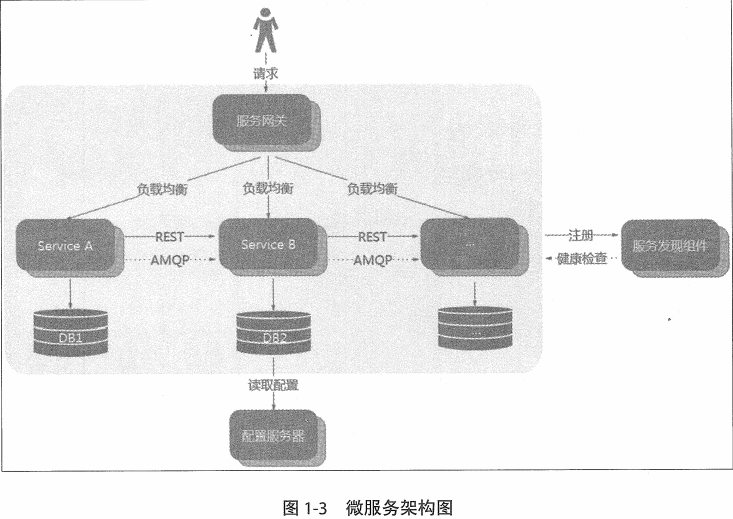
服务注册：Eureka、Consul、Zookeeper

服务调用：Rest、RPC、gRPC

服务负载均衡：Ribbon、Nginx

服务监控：Feign

架构图：



# 第二章：微服务开发框架SpringCloud

## SpringCloud简介

在SpringBoot基础上构建的，用于快速构建分布式系统的通用模式的工具集。

SpringCloud有一下特点：

1. 约定优于配置
2. 适用各种环境
3. 隐藏了组件的复杂性，提供声明式、无XML的配置方式
4. 轻量级组建：Eureka、Zuul等
5. 为微服务架构提供非常完整的支持：配置管理、服务发现、断路器、微服务网关等/
6. 选型丰富，可以按需灵活挑选。

## SpringCloud和Dubbo的区别

SpringCloud提供了一系列的解决方案，抛弃Dubbo的远程RPC调用，使用RestAPI，牺牲服务调用的性能，避免了原生RPC带来的区别，不存在代码级别的强依赖。

# 第三章：开始使用SpringCloud实战微服务

使用JDK1.8 + SpringBoot 1.4.3 RELESE + Maven 3.3.9构建项目。

使用微服务构建的是分布式系统，微服务之间通过网络进行通信。

## 构建简单的SpringBoot项目：

1. 开始搭建项目，Maven项目，pom依赖如下：

<?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.gjsyoung</groupId>  
 <artifactId>SpringCloudTest\_UserProvider</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 <packaging>war</packaging>  
  
 <!-- 引入SpringBoot -->  
 <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>  
 <!-- SpringMVC -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <!-- Jpa -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-jpa</artifactId>  
 <version>1.5.15.RELEASE</version>  
 </dependency>  
 <dependency>  
 <groupId>org.hibernate.javax.persistence</groupId>  
 <artifactId>hibernate-jpa-2.1-api</artifactId>  
 <version>1.0.0.Final</version>  
 </dependency>  
  
 <!-- Mysql数据库 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>

<version>5.1.40</version>  
 </dependency>  
  
 <!-- JSP依赖 -->  
 <dependency>  
 <groupId>javax.servlet</groupId>  
 <artifactId>jstl</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.tomcat.embed</groupId>  
 <artifactId>tomcat-embed-jasper</artifactId>  
 <!--<scope>provided</scope> 这里不能为provided-->  
 </dependency>  
 </dependencies>  
  
 <!-- 引入SpringCloud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SpringBoot的Maven插件 -->  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

1. 创建用户实体类、Repository、Controller

@Entity  
public class User {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private Long id;  
  
 @Column  
 private String username;  
  
 @Column  
 private String name;  
  
 @Column  
 private Integer age;  
  
 @Column  
 private BigDecimal balance;  
}

省略get/set方法

public interface UserRepository extends JpaRepository<User,Long> {}

@Controller  
public class UserController {  
  
 @Autowired  
 private UserRepository userRepository;  
  
 @GetMapping("/page/{id}")  
 public ModelAndView findByIdPage(@PathVariable Long id){  
 ModelAndView mav = new ModelAndView("userTest");  
 User one = userRepository.findOne(id);  
 mav.addObject("user",one);  
 return mav;  
 }  
  
 @GetMapping("/{id}")  
 @ResponseBody  
 public User findById(@PathVariable Long id){  
 User one = userRepository.findOne(id);  
 return one;  
 }  
  
}

@GetMapping是Spring4.3提供的新注解。相当于@RequestMapping（Method = GET），此时假设该项目提供一个自己的页面，也提供一个接口供外部查询User信息。

1. 编写SpringBoot的启动类

@SpringBootApplication  
public class ProviderUserApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ProviderUserApplication.class, args);  
 }  
}

@SpringBootApplication是组合注解，整合了@Configuration、@EnableAutoConfiguration和@ComponentScan注解，并开启了SpringBoot程序的组件扫描和自动配置功能。

1. 编写配置文件application.yml

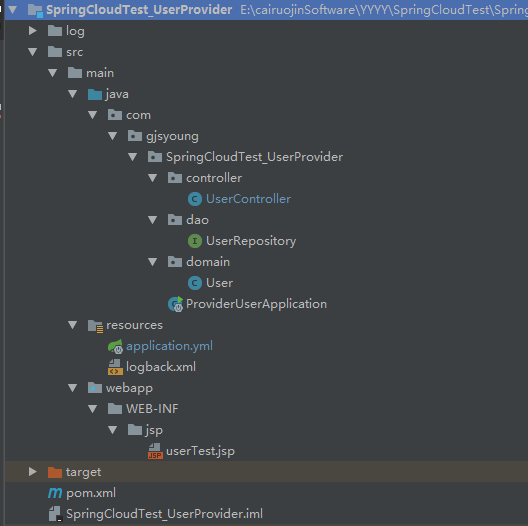
**server:  
 port:** 80  
  
**logging:  
 com.gjsyoung.SpringCloudTest:** DEBUG  
  
  
**spring:***# profiles:  
# active: dev* **jpa:  
 generate-ddl:** false  
 **show-sql:** true  
 **hibernate:  
 ddl-auto:** none  
  
 **datasource:** *# 数据库地址* **url:** jdbc:mysql://111.230.196.33:3306/SpringInAction?useSSL=false  
 **username:** cairuojin  
 **password:**   
 **driver-class-name:** com.mysql.jdbc.Driver  
  
 **mvc:  
 static:  
 static-path-pattern:** /\*\*  
 **view:  
 prefix:** /WEB-INF/jsp/  
 **suffix:** .jsp

yml的文件格式是YAML（Yet Another Markup Language）编写的文件格式，SpringBoot和SpringCloud支持使用prop或者yml作配置文件。

注意生成webapp和WEB-INF的方法：Project Structure 🡪 module 🡪 +添加一个Web 🡪

生成web.xml在\webapp\WEB-INF\web.xml，设置Web Resource在webapp。

此时即可测试SpringBoot项目搭建是否成功，当前项目文件如下：



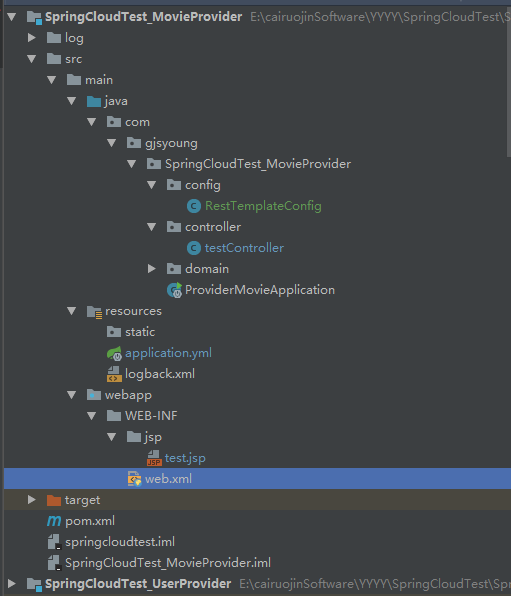
5、同理再编写一个服务的提供者，通过RestTemplate调用用户微服务的API。根据ID查询结果。

POM.xml依赖相同，application.yml中修改启动的端口，复制相同的User.class到该项目中，注册一个RestTemplate如下：

@Configuration  
public class RestTemplateConfig {  
  
 @Bean  
 public RestTemplate restTemplate(){  
 return new RestTemplate();  
 }  
}

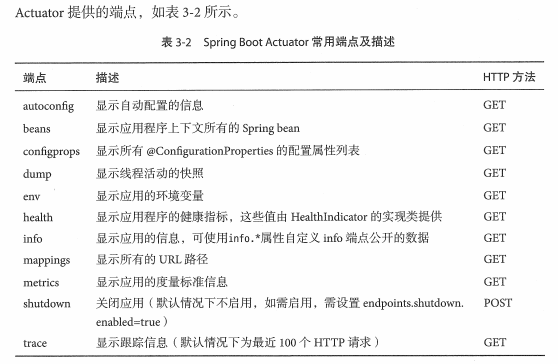
在Controller中进行RestTemplate远程调用并返回给前端：

@Controller  
public class MovieController {  
  
 @Autowired  
 RestTemplate restTemplate;  
  
 @RequestMapping("/movie")  
 public ModelAndView movieTest(Integer id){  
 ModelAndView mav = new ModelAndView("test");  
 //远程调用并获取对象  
 User user = restTemplate.getForObject("http://localhost/" + id, User.class);  
 mav.addObject("user",user);  
 return mav;  
 }  
}



## 整合Spring Boot Actuator

Spring Boot Actuator提供了很多监控的端点，使用/{endpoint}的形式访问这些端点。



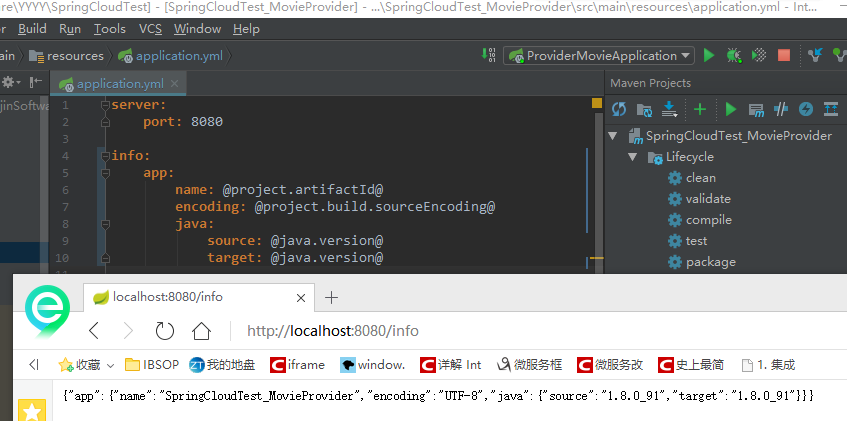
添加依赖：

<!-- actuator -->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-actuator</artifactId>  
</dependency>

此时即可访问这些端点:



Info端点为自定义展示信息的端点，在配置中进行配置想展示的内容：



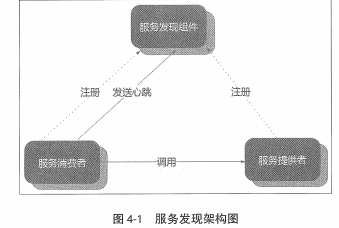
当前使用的是硬编码的RestTemplate对远程HTTP进行服务调用，此时写死了localhost:80，即使将该地址写在配置文件中亦在地址变换后需要修改，并且此时无法实现负载均衡。

# 第四章：微服务的注册与发现

为了避免硬编码问题，服务消费者需要一个强大的服务发现机制，动态获取服务提供者的网络信息。提供者的信息发生变化时，消费者也无须修改配置文件。

## 服务发现组件

在微服务启动时，将自己的网路地址等信息注册到服务发现组件中并保存起来，消费者从组件中查询到该地址并调用接口，各个微服务之间使用一定机制（例如心跳）通信。服务发现组件如果长时间无法与某微服务通信，就会注销该实例。



因此，服务发现组件应具备一下功能：

1服务注册表：是服务发现组件的核心，用他来记录各个微服务的信息。如名称+IP+端口。提供查询API和管理API。

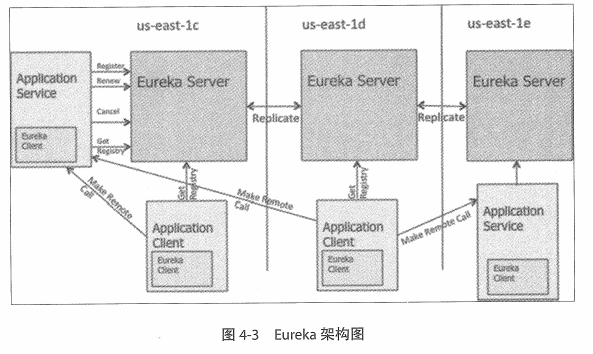
2、服务注册与服务发现：服务注册是指微服务在启动的时候，将自己的信息注册到服务发现组件上的过程。服务发现是指查询可用微服务列表及其网络地址的机制。

3、服务检查：使用一定机制定时检测已注册的服务，如果发现长时间无法访问就会从服务注册表中移除该实例。

常用的有Eureka、Zookeeper、Consul

## Eureka简介

是Netflix开源的服务发现组件，本身基于REST的服务。包含Server和Client两部分，SpringCloud将其集成在SpringCloudNetflix中，实现微服务的注册与发现。



Application Service：服务提供者

Application Client：服务消费者

Mark Remote Call：调用RestAPI行为

Eureka Server：提供服务发现能力，微服务启动时向其注册自己的信息。

Eureka Cline：一个java客户端，用于简化与Eureka Server的交互。

微服务启动后，会周期性（默认30秒）向Eureka Server发送心跳。如果一定时间没有接受到某个微服务的心跳（默认90秒），则会注销该实例。默认情况下，Eureka Server同时也是Eureka Client，互相之间通过复制的方式实现服务注册表中数据的同步。Eureka Client具有缓存功能，缓存注册表中的信息。

## 编写Eureka Server

新建一个Maven项目，添加EurekaServer的依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 <version>1.4.5.RELEASE</version>  
</dependency>

编写启动类：

@SpringBootApplication  
@EnableEurekaServer  
public class EurekaApplication {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(EurekaApplication.class,args);  
 }  
}

YML配置：

**server:  
 port:** 8761  
  
**eureka:  
 client:  
 register-with-eureka:** false  
 **fetch-registry:** false  
 **service-url:  
 defaultZone:** http://localhost:8761/eureka/

**register-with-eureka: false**表示不把自己注册到Eureka中心，默认为true，自己就是Eureka所以设置成false

**fetch-registry: false**表示是否从Eureka Server获取注册信息，默认为true，这里为单机版的Eureka Server，不需要同步其他Eureka Server节点的数据，所以设置为false

**service-url.** **defaultZone**表示与Eureka Server交互的地址，查询服务和注册服务都需要这个地址。这里也可以使用多个Url使用，分隔

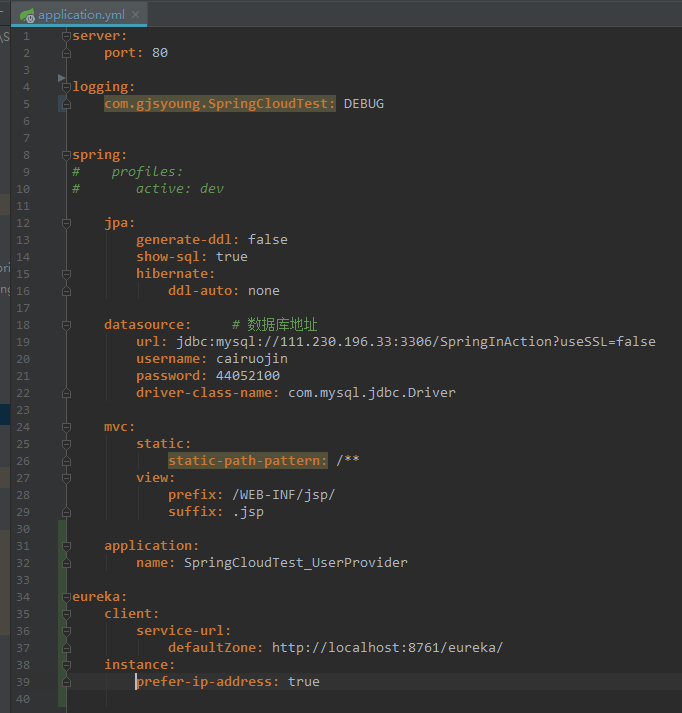
像启动SpringBoot一样启动启动类，即可启动EurekaServer，访问<http://localhost:8761/>可以进入到后台页面。

## 将微服务注册到Eureka上

将原来的UserProvider添加到Eureka上，在其pom中追加依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-eureka</artifactId>  
 <version>1.4.5.RELEASE</version>  
</dependency>

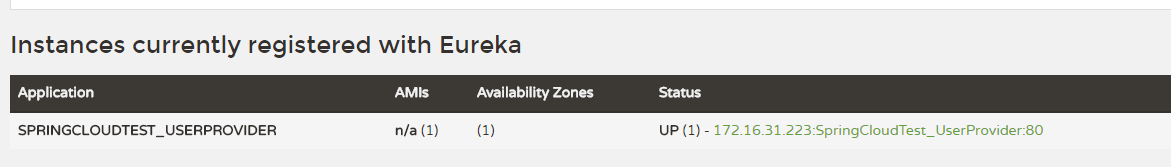
追加配置如下：



此时表示与Eureka Server交互的地址为Eureka的地址，**instance.prefer-ip-address:true**表示将自己的IP注册到Eureka Server（如果同时配置了IP，则会将这个IP注册到Eureka而不是本机IP），如果为false则表示将操作系统的hostname（本机IP）注册到Eureka Server。

在启动类上，加上**@EnableEurekaClient**表示该项目是一个Eureka Client

启动后效果如下：



也可以在启动类上注解**@EnableDiscoveryClient**，因为服务发现组件有多种选择，该 组件为多种选择都提供了支持，是spring-cloud-commons项目的注解。在Eureka项目中，两个注解没有区别。

## Eureka Server的高可用

单节点下的Eureka Server容易宕机，并不适合线上生产环境。当宕机时，Eureka Client中的缓存无法被更新，如果微服务也不可用，则会影响到调用和整个系统的稳定。

Eureka Server可以通过运行多个实例并相互注册的方式实现高可用部署，实例之间会彼此增量地同步信息，确保节点数据一致。

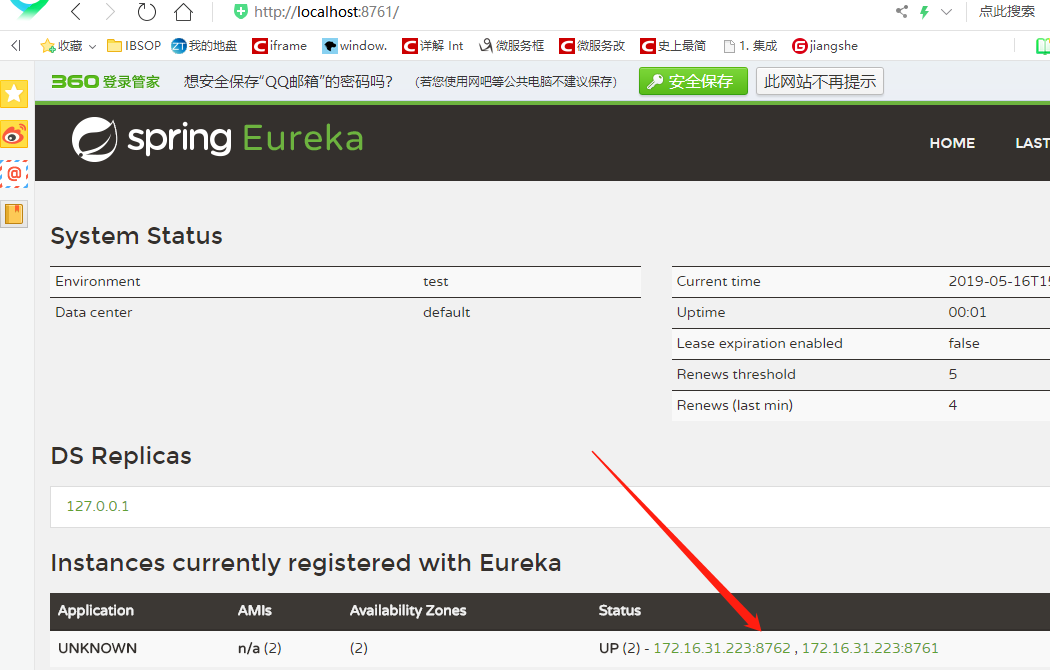
将Eureka工程直接复制一份，注意pom中修改artifactId，

两个Eureka的工程配置分别如下：

**server:  
 port:** 8762  
  
**eureka:  
 instance:  
 hostname:** peer2  
 **prefer-ip-address:** true  
  
 **client:  
 service-url:  
 defaultZone:** http://127.0.0.1:8761/eureka/  
  
**Spring:  
 application:  
 name:** SpringCloudTest\_EurekaServerTest

**server:  
 port:** 8761  
  
**eureka:  
 instance:  
 hostname:** peer1  
 **prefer-ip-address:** true  
  
 **client:  
 service-url:  
 defaultZone:** http://127.0.0.1:8762/eureka/  
  
**Spring:  
 application:  
 name:** SpringCloudTest\_EurekaServerTest\_ha

可见，分别占用两个端口，向另外一个Eureka注册自己。同时启动，效果如下：



此时即可将已有的微服务注册到Eureka中，可以不修改Eureka.client:.service-url:.defaultZone:的地址，因为Eureka Server会彼此同步数据，但还是建议把每一个地址都写上：

**eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8761/eureka/, http://localhost:8762/eureka/  
 **instance:  
 prefer-ip-address:** true

## Eureka用户认证

Eureka Server在不配置的情况下，是允许匿名访问的。也可以构建一个需要登录才能访问的Eureka Server。在pom中添加用户认证的能力依赖：

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 <version>1.5.6.RELEASE</version>  
</dependency>

使用SpringBoot集成SpringSecurity。

在配置中启动安全：

**security:  
 basic:  
 enabled:** true  
 **user:  
 name:** cairuojin  
 **password:** 123456

则此时登录Eureka Server或注册服务都需要登录验证。

将注册地址改为：<http://cairuojin:123456@localhost:8763/eureka/>

Name:password@/的形式即可认证。

## Eureka的元数据：

Eureka的元数据有两种，分别为**标准元数据**和**自定义元数据**。

标准元数据指主机名，IP地址，端口号，状态页和健康检查等。这些会发布在服务注册表中，用于服务之间的调用。自定义元数据可以用eureka.instance.metadata-map配置。如在其中一个微服务下添加自定义元数据：

**eureka:  
 client:  
 service-url:  
 defaultZone:** http://localhost:8761/eureka/,http://localhost:8762/eureka/,http://cairuojin:123456@localhost:8763/eureka/  
 **instance:  
 prefer-ip-address:** true  
 **metadata-map:  
 myname:** cairuojin  
 **myage:** 22

在另外一个微服务中可以获得此配置的元数据：

@Controller  
public class MovieController {  
  
 @Autowired  
 DiscoveryClient discoveryClient;  
  
 @ResponseBody  
 @RequestMapping("/showInfo")  
 public List<ServiceInstance> showInfo(){  
 return discoveryClient.getInstances("SpringCloudTest\_UserProvider");  
 }

}

使用discoveryClient.getInstances("SpringCloudTest\_UserProvider")获得其元数据列表。



## Eureka Server的REST端点

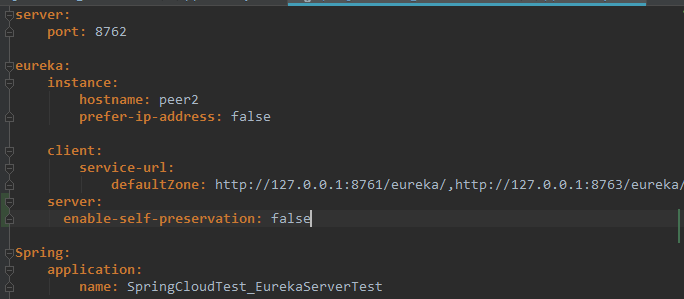
Eureka Server提供了一些REST端点，非JVM的微服务可使用这些REST端点操作Eureka，从而实现注册与发现。实际上Eureka Client是一个使用Java编写的操作这些REST端点的类库。使用这些端点即可对服务进行查看注册或注销等（传递XML或JSON）。



## Eureka的自我保护模式

进入Eureka的红字即是Eureka进入了自我保护模式。默认情况下（90秒）未手动某个微服务的心跳时，将会注销该实例。但是因此有可能因为网络不通的问题导致被注销。自我保护模式目的即解决此问题，应对网络异常的安全保护措施。当Eureka Server在短时间内丢失过多客户端时，这个节点就会进入自我保护模式，保护服务注册表中的信息，不再删除服务注册表中的数据。网络故障恢复时，自动退出自我保护模式。

在SpringCloud中可以使用eureka.server.enable-self-preservation = false禁用自我保护模式。



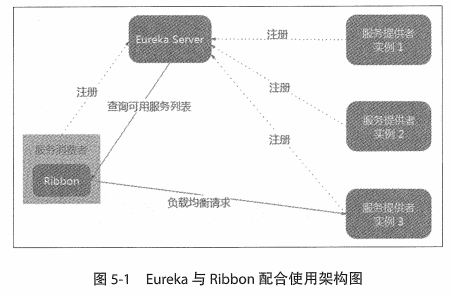
其他内容诸如Eureka的健康检查和IP的选择。

# 第五章：使用Ribbon实现客户端侧负载均衡

## Ribbon简介

Ribbon是Netflix发布的负载均衡器，有助于控制HTTP和TCP客户端的行为。为Ribbon配置服务提供者地址列表后，Ribbon就基于某种负载均衡方法帮助消费者去请求。默认算法有轮询、随机等，也可以自定义实现负载均衡算法。

Ribbon与Eureka配合使用时，Ribbon可自动从Eureka Server获取服务提供者地址列表，基于负载均衡算法请求其中一个服务提供者实例。



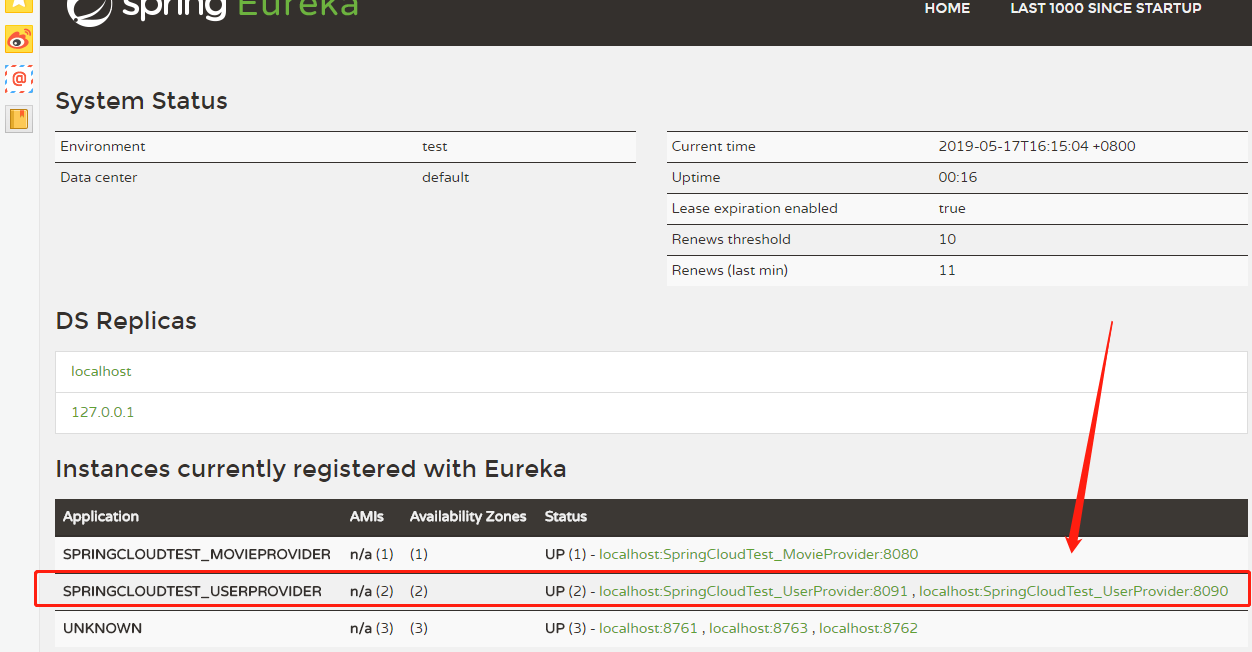
需要添加依赖：

<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-ribbon</artifactId>  
 <version>1.4.5.RELEASE</version>  
</dependency>

但如果项目已经依赖spring-cloud-starter-eureka，则已经自动依赖了ribbon。

复制User微服务为一个新的工程，注意其module的artifactId。两个的相同。同时启动，效果如下：

**Spring.application.name:** SpringCloudTest\_UserProvider



可见，当前这个微服务有两个实例。在Movie微服务中，添加负载均衡注解：

@Configuration  
public class RestTemplateConfig {  
  
 @Bean  
 @LoadBalanced  
 public RestTemplate restTemplate(){  
 return new RestTemplate();  
 }  
}

为RestTemplate整合Ribbon，添加负载均衡的能力。

在Controller中做验证：

@Controller  
public class MovieController {  
  
 @Autowired  
 RestTemplate restTemplate;  
  
 @Autowired  
 DiscoveryClient discoveryClient;  
  
 @Autowired  
 LoadBalancerClient loadBalancerClient;  
  
 @RequestMapping("/movie")  
 public ModelAndView movieTest(Integer id){  
 ModelAndView mav = new ModelAndView("test");  
 //远程调用并获取对象  
 User user = restTemplate.getForObject("http://SpringCloudTest\_UserProvider/" + id, User.class);  
 mav.addObject("user",user);  
 return mav;  
 }  
  
 @GetMapping("/log-instance")  
 @ResponseBody  
 public void logUserInstance(){  
 ServiceInstance userprovider = loadBalancerClient.choose("SpringCloudTest\_UserProvider");  
 System.*out*.println(userprovider.getServiceId() + " "  
 + userprovider.getHost() + ":"  
 + userprovider.getPort());  
 }  
  
 @ResponseBody  
 @RequestMapping("/showInfo")  
 public List<ServiceInstance> showInfo(){  
 return discoveryClient.getInstances("SpringCloudTest\_UserProvider");  
 }  
}

SpringCloudTest\_UserProvider是用户微服务的虚拟主机名。当Ribbon和Eureka配合使用时，会自动将虚拟主机名映射为微服务的网络地址。

注意：貌似虚拟主机名中不能包含\_，导致报空指针，将虚拟主机名全部修改为SpringCloudTest-UserProvider后正常。此时调用/movie?id=1可以发现两台user微服务都有机会提供服务，调/log-instance发现IP不一样。

默认情况下，虚拟主机名和服务名称（Spring.application.name）相同，可以使用eureka.instance.virtual-host-name手动指定虚拟主机名。

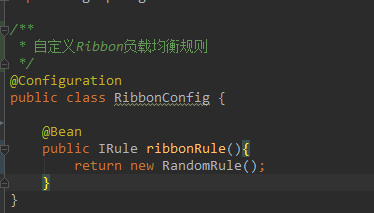
·不能将restTemplate.getForObject和loadBalancerClient.choose写在同一个方法中，两者会有冲突，因为restTemplate此时实际上是一个Ribbon客户端，本身已经包含choose行为。

## 自定义Ribbon配置

可以修改Ribbon的负载均衡规则等。



@Bean  
@ConditionalOnMissingBean  
public IRule ribbonRule(IClientConfig config){  
 ZoneAvoidanceRule rule = new ZoneAvoidanceRule();  
 rule.initWithNiwsConfig(config);  
 return rule;  
}

如上，BeanType为IRule，BeanName为ribbonRule，ClassName为ZoneAvoidanceRule，这是一种根据服务提供者所在Zone的性能以及服务提供者可用性综合计算，选择提供者节点的负载均衡规则。在SpringCloud中Ribbon默认的配置类是RibbonClientConfiguration。如上，即可实现Ribbon随机分配。

也可以使用属性来配置：



上面的操作都是在Ribbon与Eureka结合使用的情况下，Ribbon也可以脱离独自使用，请见书中5.5。

# 第六章：使用Feign实现声明式REST调用

## Feign简介

Feign是Netflix开发的声明式、模板化的HTTP客户端，可帮助更加快捷的调用HTTP API。

在SpringCloud对Feign进行了增强，支持SpringMVC注解，整合Ribbon和Eureka。

1. 添加依赖

<!-- Feign -->  
<dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-feign</artifactId>  
 <version>1.4.5.RELEASE</version>  
</dependency>

1. 创建一个Feign接口

@FeignClient(name = "SpringCloudTest-UserProvider")  
public interface UserFeignCline {  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
 public User getUser(@PathVariable Long id);  
  
}

使用@FeignClient 指定调用的微服务名称（由于使用了Eureka和Ribbon，所以会将名称解析成注册表中的服务），使用@RequestMapping指定调用的方法URL

1. 修改Controller中的调用

@Autowired  
UserFeignCline userFeignCline;  
  
*/\*\*  
 \* 调用用户微服务  
 \** ***@param*** *id  
 \** ***@return*** *\*/*@RequestMapping("/movie")  
public ModelAndView movieTest(Integer id){  
 ModelAndView mav = new ModelAndView("test");  
 //远程调用并获取对象  
 User user = restTemplate.getForObject("http://SpringCloudTest-UserProvider/" + 1 , User.class);  
 mav.addObject("user",user);  
 return mav;  
}  
  
@RequestMapping("/movie1")  
public ModelAndView movieTest1(Integer id){  
 ModelAndView mav = new ModelAndView("test");  
 User user = userFeignCline.getUser(1L);  
 mav.addObject("user",user);  
 return mav;  
}

Movie为原来的调用方式 movie1为使用Feign声明式调用方式。

1. 启动类中添加允许的注解：@EnableFeignClients

@SpringBootApplication  
@EnableDiscoveryClient  
@EnableFeignClients  
public class ProviderMovieApplication\_feign {  
  
 public static void main(String[] args) {  
 SpringApplication.*run*(ProviderMovieApplication\_feign.class, args);  
 }  
}

注意：如果报包冲突，尝试在pom.xml添加Ribbon依赖：



## 自定义Feign配置

在SpringCloud中，Feign的默认配置类是FeignClientsConfiguration，定义了Feign默认使用的编码器和解码器，所使用的契约等。

SpringCloud允许通过@FeignClient的configuration属性自定义Feign的配置，自定义属性配置的优先级比FeignClientsConfiguration高。

在SpringCloud中，Feign默认使用的契约是SpringMvcContract，因此他可以使用SpringMVC注解。

如果进行自定义Feign配置（例如修改契约为Feign默认，则此时不能够再使用SpringMVC注解）

@Configuration  
public class FeignConfig {  
  
 */\*\*  
 \* 将契约改成Feign默认契约，则可以使用Feign自身的注解  
 \** ***@return*** *\*/* @Bean  
 public Contract feignContract(){  
 return new feign.Contract.Default();  
 }  
  
}

@FeignClient(name = "SpringCloudTest-UserProvider", configuration = FeignConfig.class)  
public interface UserFeignCline {  
  
// @RequestMapping(value = "/{id}", method = RequestMethod.GET)  
// public User getUser(@PathVariable("id") Long id);  
  
 @RequestLine("GET /{id}")  
 public User findUser(@Param("id") Long id);  
  
}

同理：如果不想FeignConfig被所有Feign客户端共享，必须让SpringBoot不扫描到这个配置类并且在@FeignClient手动指定这个配置。

## 手动创建Feign

如果自定义Feign的方式无法满足需求，可使用Feign Builder API手动创建Feign。

假设有如下需求：

1. 假设用户微服务需要登录后才能调用，并且对于相同的API，不同角色的用户有不同行为。
2. 电影微服务同一个Feign接口，使用不同的账号登录，调用相应的用户微服务接口。
3. 首先，在用户微服务中整合Security。

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
 <version>1.4.5.RELEASE</version>  
</dependency>

1. 配置Security

@Configuration  
@EnableWebSecurity  
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {  
  
 @Autowired  
 CustomerUserDetailsService customerUserDetailsService;  
  
 */\*\*  
 \* 配置请求加密  
 \** ***@param*** *http  
 \** ***@throws*** *Exception  
 \*/* @Override  
 protected void configure(HttpSecurity http) throws Exception {  
 //所有的请求都通过HttpBasic  
 http.authorizeRequests().anyRequest().authenticated().and().httpBasic();  
 }  
  
 */\*\*  
 \* 配置用户登录  
 \** ***@param*** *auth  
 \** ***@throws*** *Exception  
 \*/* @Override  
 protected void configure(AuthenticationManagerBuilder auth) throws Exception {  
 auth.userDetailsService(customerUserDetailsService).passwordEncoder(this.passwordEncoder());  
 }  
  
 */\*\*  
 \* 不做密码加密  
 \** ***@return*** *\*/* @Bean  
 public PasswordEncoder passwordEncoder(){  
 return NoOpPasswordEncoder.*getInstance*();  
 }  
  
 */\*\*  
 \* 模拟取用户过程  
 \*/* @Component  
 class CustomerUserDetailsService implements UserDetailsService{  
 @Override  
 public UserDetails loadUserByUsername(String s) throws UsernameNotFoundException {  
 List<GrantedAuthority> userRole = new ArrayList<>();  
 userRole.add(new SimpleGrantedAuthority("user-role"));  
 List<GrantedAuthority> adminRole = new ArrayList<>();  
 adminRole.add(new SimpleGrantedAuthority("admin-role"));  
 if("user".equals(s)){  
 return new User("user","111111",userRole);  
 } else if("admin".equals(s)){  
 return new User("admin","123456",adminRole);  
 } else {  
 return null;  
 }  
 }  
 }  
}

1. 修改用户微服务Controller：

@Controller  
public class UserController {  
  
 @Autowired  
 private UserRepository userRepository;  
  
 @GetMapping("/{id}")  
 @ResponseBody  
 public User findById(@PathVariable Long id){  
 Object principal = SecurityContextHolder.*getContext*().getAuthentication().getPrincipal();  
 if(principal instanceof UserDetails){  
 UserDetails user = (UserDetails) principal;  
 Collection<? extends GrantedAuthority> authorities = user.getAuthorities();  
 for(GrantedAuthority grantedAuthority : authorities){  
 //打印当前用户  
 System.*out*.println(user.getUsername() + " " + user.getPassword() + " " + grantedAuthority.getAuthority());  
 }  
 }  
  
  
 User one = userRepository.findOne(id);  
 return one;  
 }  
}

此时，访问该接口即需要登录账号和密码才能调用。

1. 继续修改电影微服务

删除原本UserFeignClient上的@FeignClient注解

删除原本启动类上的@EnableFeignClients注解

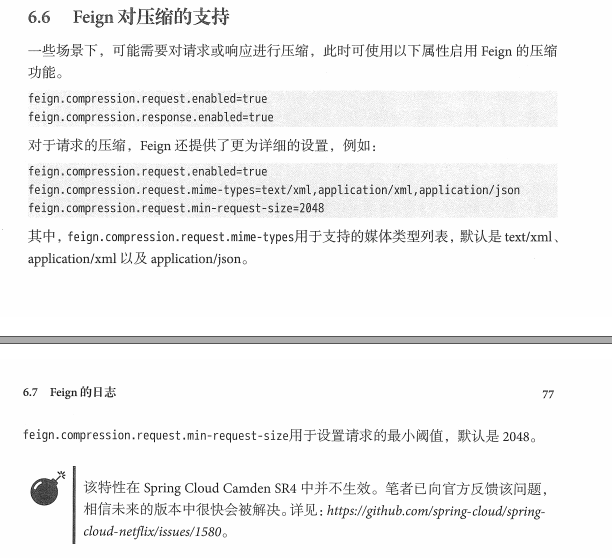
屏蔽原来的Controller，新建Controller如下：

@Import(FeignClientsConfiguration.class)  
@RestController  
public class MoviePersonalController {  
  
 private UserFeignCline userAdminFeignCline;  
  
 private UserFeignCline userUserFeignCline;  
  
  
 @Autowired  
 public MoviePersonalController(Decoder decoder, Encoder encoder, Client client, Contract contract){  
 this.userAdminFeignCline = Feign  
 .*builder*()  
 .client(client)  
 .encoder(encoder)  
 .decoder(decoder)  
 .contract(contract)  
 .requestInterceptor(  
 new BasicAuthRequestInterceptor("admin","123456")  
 )  
 .target(UserFeignCline.class, "http://SpringCloudTest-UserProvider/");  
  
 this.userUserFeignCline = Feign  
 .*builder*()  
 .client(client)  
 .encoder(encoder)  
 .decoder(decoder)  
 .contract(contract)  
 .requestInterceptor(  
 new BasicAuthRequestInterceptor("user","111111")  
 )  
 .target(UserFeignCline.class, "http://SpringCloudTest-UserProvider/");  
 }  
  
 */\*\*  
 \* 调用用户微服务  
 \** ***@param*** *id  
 \** ***@return*** *\*/* @RequestMapping("/userUser/{id}")  
 public User movieTest1(@PathVariable Long id){  
 return userUserFeignCline.findUser(id);  
 }  
 @RequestMapping("/adminUser/{id}")  
 public User movieTest2(@PathVariable Long id){  
 return userAdminFeignCline.findUser(id);  
 }  
}

@Import导入的是SpringCloud为Feign默认提供的配置类。有两个UserFeignClient的实现，一个登录账号为user，另一个为admin。通过不同的URL实现两个不同账号调用同一个接口。

## Feign对继承的支持

## Feign对压缩的支持



## Feign的日志

每个Feign客户端都会创建一个Logger，日志打印只会对DEBUG级别做出响应。使用配置修改其日志打印内容。

@Configuration  
public class FeignConfig {  
  
 */\*\*  
 \* 修改日志级别为FULL  
 \** ***@return*** *\*/* @Bean  
 public Logger.Level level(){  
 return Logger.Level.*FULL*;  
 }  
  
}

## 使用Feign构造多参数请求

如果使用Get请求目标接口，必须传递参数值个数相同，不能封装在对象中，但可以封装在Map中。

使用Post请求目标接口，封装在对象中。

# 第七章：使用Hystrix实现微服务的容错处理

防止雪崩效应，需要有容错机制：为网络请求设置超时，使用断路器模式（过载时返回，自我修复）。

## 使用Hystrix实现容错

Hystrix是由Netflix开源的一个延迟和容错库，用于隔离访问远程系统、服务或者第三方库，防止级联失败，从而提升系统的可用性与容错性。

1. 包裹请求：使用HystrixCommand包裹对依赖的调用逻辑，每个命令在独立的线程中执行。使用到了设计模式中的命令模式。
2. 跳闸机制：当某服务的错误率超过一定的阈值时，自动或者手动跳闸，停止请求该服务一段时间
3. 资源隔离：Hystrix为每个依赖都维护了一个小型的线程池。如果线程池已满，立即拒绝该依赖请求。
4. 监控：可以实时监控运行指标和配置的变化，例如成功、失败、超时以及被拒绝的请求等
5. 回退机制：当请求失败、超时被拒绝或断路器打开时，执行回退逻辑。可以由开发人员自行提供回退逻辑。
6. 自我修复：断路器打开一段时间后进入半开状态，允许一个请求访问依赖的服务，如果成功则关闭断路器，否则继续打开状态。

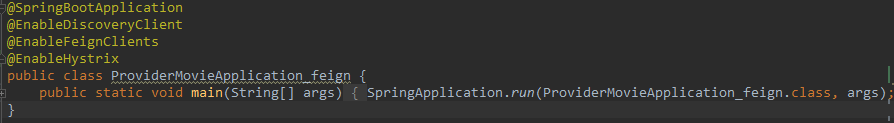
## 通用方式整合Hystrix

在Movie微服务中添加容错机制：

1. 添加依赖：

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

1. 启动类上注解@EnableCircuitBreaker或@EnableHystrix启动断路器支持



1. 在Controller上添加断路器容错方法

@RequestMapping("/findUserById")  
@HystrixCommand(fallbackMethod = "findByIdFallback")  
public User findUserById(Integer id){  
 User user = userFeignCline.getUser(1L);//调用用户微服务  
 return user;  
}  
  
*/\*\*  
 \* 容错方法  
 \** ***@return*** *\*/*public User findByIdFallback(Integer id){  
 User user = new User();  
 user.setId(0L);  
 user.setAge(0);  
 user.setName("默认用户");  
 user.setUsername("default");  
 return user;  
}

在普通方法上，注解@HystrixCommand，为其指定回退方法，该回退方法具有相同的参数与返回值，返回一个默认的User。

此时Movie微服务去调用User微服务时，如果发生超时等情况，则会返回一个默认用户：



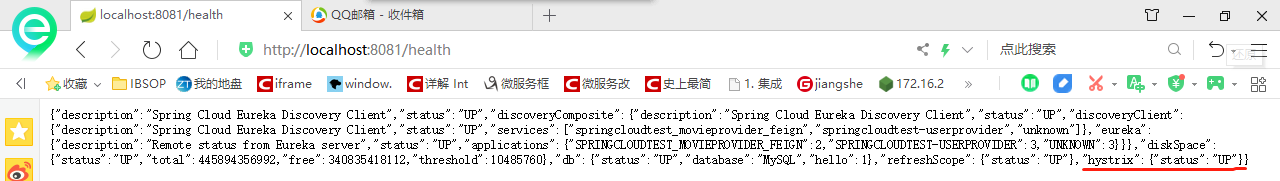
@HystrixCommand注解灵活，可以试用HystrixProperty的commandProperties来配置@HystrixCommand。

例如：

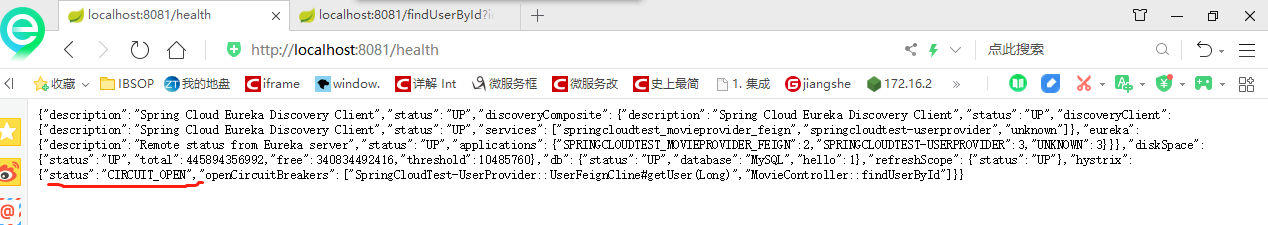
@RequestMapping("/findUserById123")  
@HystrixCommand(fallbackMethod = "findByIdFallback", commandProperties = {  
 @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "5000"),  
 @HystrixProperty(name = "metrics.rollingStats.timeInMillseconds", value = "10000")  
}, threadPoolProperties = {  
 @HystrixProperty(name = "coreSize", value = "1"),  
 @HystrixProperty(name = "maxQueueSize", value = "10")  
})  
public User findUserById123(Integer id){  
 User user = userFeignCline.getUser(1L);//调用用户微服务  
 return user;  
}

进入回退方法并不意味着断路器被打开，使用Actuator时，断路器的状态会暴露在Actuator提供的/health端点中。

当一切正常的时候，断路器的状态是关闭的（up）：



当回退方法被执行的时候，还是关闭状态，因为失败率还未达到阈值（默认5秒内失败2-次）。请求失败，超时，被拒绝以及断路器打开时都回执行回退逻辑，但不一定断路器打开了。快速访问失败可以让断路器打开：



## Hystrix的隔离策略

隔离策略：线程隔离和信号量隔离。

·THREAD（线程隔离）：使用该方式，HystrixCommand将会在单独的线程上执行，并发请求受线程池中的线程数量限制。

·SEMAPHORE（信号量隔离）：使用该方式，HystrixCommand将会在调用线程上执行，开销相对较小，并发请求受到信号量个数限制。

默认并推荐使用线程隔离，这种方式有一个除网络超时以外的额外保护层。一般只有当调用负债非常高时才需要使用信号量隔离。可使用execution.isolation.strategy属性指定隔离策略。

如果发生找不到上下文的运行时异常，可考虑将隔离策略设置为SEMAPHORE。

## Feign使用Hystrix

Feign是以接口的形式工作，前面通用方式的Hystrix不适用于Feign，SpringCloud默认为Feign整合了Hystrix，只要Hystrix在项目的Classpath中，Feign默认就会用断路器包裹所有方法。

为UserFeignClient接口添加回退：

1. 在Feign接口中添加fallback属性：

@FeignClient(  
 name = "SpringCloudTest-UserProvider",  
 configuration = FeignConfig.class,  
 fallback = UserFeignClientCallBack.class  
)  
public interface UserFeignCline {  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
 public User getUser(@PathVariable("id") Long id);  
  
}

1. 编写回退类UserFeignClientCallBack：

@Component  
public class UserFeignClientCallBack implements UserFeignCline{  
  
 */\*\*  
 \* 调用用户微服务容错方法  
 \** ***@param*** *id  
 \** ***@return*** *\*/* @Override  
 public User getUser(Long id) {  
 System.*out*.println("容错方法启动 ");  
 User user = new User();  
 user.setId(0L);  
 user.setAge(0);  
 user.setName("默认用户");  
 user.setUsername("default");  
 return user;  
 }  
}

实现该Fegin接口，因此这个接口有多少个方法，就必须有多少个容错方法，必须使用@Component使得Spring能加载

1. 正常调用Feign接口

@RequestMapping("/findUserById")  
@ResponseBody  
public User findUserById(Integer id){  
 User user = userFeignCline.getUser(1L);//调用用户微服务  
 return user;  
}

## FallbackFactory检查回退原因

使用@FeignClient的fallbackFactory属性：

@FeignClient(  
 name = "SpringCloudTest-UserProvider",  
 configuration = FeignConfig.class,  
 //fallback = UserFeignClientCallBack.class,  
 fallbackFactory = UserFeignFallBackFactory.class  
)  
public interface UserFeignCline {  
  
 @RequestMapping(value = "/{id}", method = RequestMethod.*GET*)  
 public User getUser(@PathVariable("id") Long id);  
  
}

@Component  
public class UserFeignFallBackFactory implements FallbackFactory<UserFeignCline> {  
 @Override  
 public UserFeignCline create(Throwable throwable) {  
 return new UserFeignCline() {  
 @Override  
 public User getUser(Long id) {  
 System.*out*.println("回退原因是：" + throwable);  
 User user = new User();  
 user.setId(0L);  
 user.setAge(0);  
 user.setName("默认用户");  
 user.setUsername("default");  
 return user;  
 }  
 };  
 }  
}

因此也可以在FallBackFactory中，根据回退的原因做出不同的返回。

## Feign禁用Hystrix

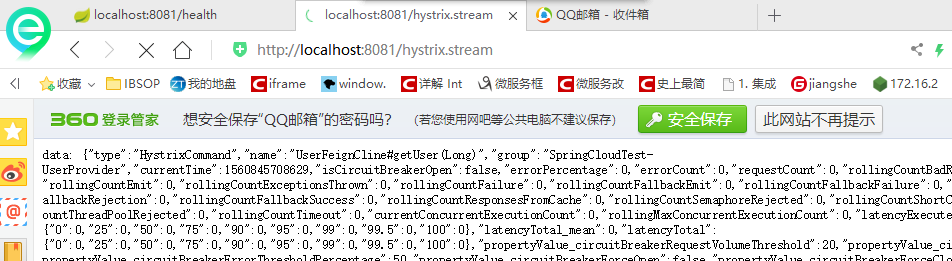
由于Hystrix在项目的classpath中Feign就会使用断路器包裹Feign客户端的所有方法， 但是某些时候不需要断路器，借助Feign的自定义配置可以手动禁用Hystrix。

@Configuration  
public class FeignConfig {  
  
 */\*\*  
 \* 修改日志级别为FULL  
 \** ***@return*** *\*/* @Bean  
 public Logger.Level level(){  
 return Logger.Level.*FULL*;  
 }  
  
 */\*\*  
 \* 禁用Hystrix  
 \** ***@return*** *\*/* @Bean  
 @Scope("prototype")  
 public Feign.Builder feignBuilder(){  
 return Feign.*builder*();  
 }  
  
}

全局禁用Hystrix，在application.yml中配置feign.hystrix.enabled = false

## Hystrix的监控

除了实现容错外，Hystrix还提供实时的监控。HystrixCommand和HystrixObservableCommand在执行时会生成执行结果和运行指标。使用Hystrix的模板hystrix-metrics-event-stream，就可将监控的指标信息以text/event/stream的格式暴露给外部系统。Spring-cloud-starter-hystrix已包含该模块，需要actuator即可使用/hystrix.stream端点获得Hystrix的监控信息。包含HystrixCommand的名称，group名称，断路器状态，错误率，错误数等。

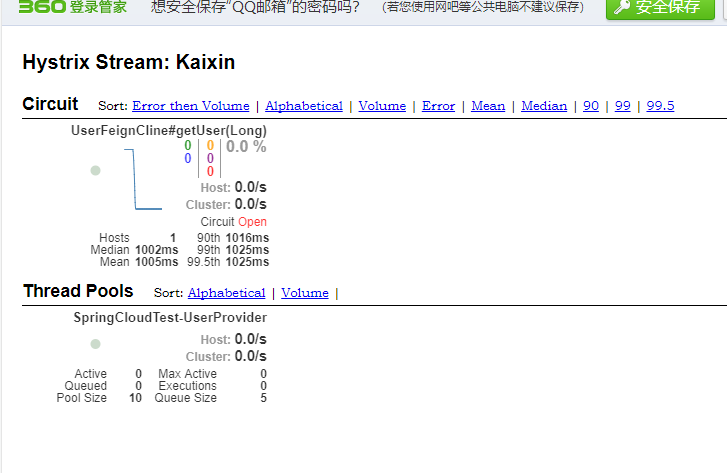


这种文字形式的数据非常难看，可使用HystrixDashboard可视化监控数据。

1. 添加依赖：

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

1. 在入口处添加@EnableHystrixDashboard
2. 继续访问[/hystrix](http://localhost:8081/hystrix.stream)即可，将<http://localhost:8081/hystrix.stream这个URL>下的监控输入到页面上并随意设置一个title即可图形化监控。



## 使用Turbine聚合监控数据

前面都是用/hystrix.stream监控单个微服务实例，可以使用Turbine聚合Hystrix监控数据的工具，将所有/hystrix.stream端点的数据聚合到一个组合的/turbine.stream中。