

Spring Cloud 微服务实践

主讲：军哥

蛙课网（动力节点旗下品牌）

<http://www.wkcto.com>

学 Java 全栈，上蛙课网

第一章：微服务架构概述

1-1. 系统进化理论概述

在系统架构与设计的实践中，经历了两个阶段，一个阶段是早些年常见的集中式系统，一个阶段是近年来流行的分布式系统；

集中式系统：

集中式系统也叫单体应用，就是把所有的程序、功能、模块都集中到一个项目中，部署在一台服务器上，从而对外提供服务；

分布式系统

分布式系统就是把所有的程序、功能拆分成不同的子系统，部署在多台不同的服务器上，这些子系统相互协作共同对外提供服务，而对用户而言他并不知道后台是多个子系统和多台服务器在提供服务，在使用上和集中式系统一样；

集中式系统跟分布式系统是相反的两个概念，他们的区别体现在“合”与“分”。

1-2. 系统进化理论背景

系统进化的背景与中国互联网用户规模庞大有巨大关系，中国互联网用户规模有7.7亿，庞大的用户访问量对系统的架构设计是巨大的挑战；

产品或者网站初期，通常功能较少，用户量也不多，所以一般按照单体应用进行设计和开发，按照经典的MVC三层架构设计；

随着业务的发展，应用功能的增加，访问用户的增多，传统的采用集中式系统进行开发的方式就不再适用了，因为在这种情况下，集中式系统就会逐步变得非常

庞大，很多人维护这么一个系统，开发、测试、上线都会造成很大问题，比如代码冲突，代码重复，逻辑错综混乱，代码逻辑复杂度增加，响应新需求的速度降低，隐藏的风险增大；

所以需要按照业务维度进行应用拆分，采用分布式开发，每个应用专于做某些方面的事情，比如将一个集中式系统拆分为用户服务、订单服务、产品服务、交易服务等，各个应用服务之间通过相互调用完成某一项业务功能。

1-3. 什么是微服务架构

分布式强调系统的拆分，微服务也是强调系统的拆分，微服务架构属于分布式架构的范畴；

并且到目前为止，微服务并没有一个统一的标准的定义，那么微服务究竟是什么？

微服务一词源于 Martin Fowler (马丁·福勒) 的名为 Microservices 的博文，可以在他的官方博客上找到这篇文章：

<http://martinfowler.com/articles/microservices.html>

中文翻译版本：

<https://www.martinfowler.cn/articles/microservices.html>

简单地说，微服务是系统架构上的一种设计风格，它的主旨是将一个原本独立的系统拆分成多个小型服务，这些小型服务都在各自独立的进程中运行，服务之间通过基于 HTTP 的 RESTful API 进行通信协作；

被拆分后的每一个小型服务都围绕着系统中的某一项业务功能进行构建，并且每个服务都是一个独立的项目，可以进行独立的测试、开发和部署等；

由于各个独立的服务之间使用的是基于 HTTP 的 JSON 作为数据通信协作的基

础，所以这些微服务可以使用不同的语言来开发；

1-4. 微服务架构的优缺点

- 1、我们知道微服务架构是将系统中的不同功能模块拆分成多个不同的服务，这些服务进行独立地开发和部署，每个服务都运行在自己的进程中，这样每个服务的更新都不会影响其他服务的运行；
- 2、由于每个服务是独立部署的，所以我们可以更准确地监控每个服务的资源消耗情况，进行性能容量的评估，通过压力测试，也很容易发现各个服务间的性能瓶颈所在；
- 3、由于每个服务都是独立开发，项目的开发也比较方便，减少代码的冲突、代码的重复，逻辑处理流程也更加清晰，让后续的维护与扩展更加容易；
- 4、微服务可以使用不同的编程语言进行开发；

但是在系统架构领域关于微服务架构也有一些争论，有人倾向于在系统设计与开发中采用微服务架构实现软件系统的低耦合，被认为是系统架构的未来方向，Martin Fowler（马丁·福勒）也给微服务架构很高的评价；

同时，对微服务架构也有人持反对观点，他们表示：

- 1、微服务架构增加了系统维护、部署的难度，导致一些功能模块或代码无法复用；
- 2、随着系统规模的日渐增长，微服务在一定程度上也会导致系统变得越来越复杂，增加了集成测试的复杂度；
- 3、随着微服务的增多，数据的一致性问题，服务之间的通信成本等都凸显了出来；

所以在系统架构时也要提醒自己：不要为了微服务而微服务。

1-5. 为什么选择 Spring Cloud 构建微服务

微服务一词是 Martin Fowler (马丁·福勒) 于 2014 年提出来的，近几年微服务架构的讨论非常火热，无数的架构师和开发者在实际项目中实践着微服务架构的设计理念，他们在微服务架构中针对不同应用场景出现的各种问题，也推出了很多解决方案和开源框架，其中我们国内的互联网企业也有一些著名的框架和方案；

整个微服务架构是由大量的技术框架和方案构成，比如：

服务基础开发	Spring MVC、Spring、SpringBoot
服务注册与发现	Netflix 的 Eureka、Apache 的 ZooKeeper 等
服务调用	RPC 调用有阿里巴巴的 Dubbo，Rest 方式调用有当当网 Dubbo 基础上扩展的 Dubbox、还有其他方式实现的 Rest，比如 Ribbon、Feign
分布式配置管理	百度的 Disconf、360 的 QConf、淘宝的 Diamond、Netflix 的 Archaius 等
负载均衡	Ribbon
服务熔断	Hystrix
API 网关	Zuul
批量任务	当当网的 Elastic-Job、LinkedIn 的 Azkaban
服务跟踪	京东的 Hydra、Twitter 的 Zipkin 等

但是在微服务架构上，几乎大部分的开源组件都只能解决某一个场景下的问题，所以这些实施微服务架构的公司也是整合来自不同公司或组织的诸多开源框架，并加入针对自身业务的一些改进，没有一个统一的架构方案；所以当我们准备实施微服务架构时，我们要整合各个公司或组织的开源软件，而且某些开源软件又有多种选择，这导致在做技术选型的初期，需要花费大量的时

间进行预备研、分析和实验，这些方案的整合没有得到充分的测试，可能在实践中会遇到各种各样的问题；

Spring Cloud 的出现，可以说是为微服务架构迎来一缕曙光，有 SpringCloud 社区的巨大支持和技术保障，让我们实施微服务架构变得异常简单了起来，它不像我们之前所列举的框架那样，只是解决微服务中的某一个问题，而是一个解决微服务架构实施的综合性解决框架，它整合了诸多被广泛实践和证明有效的框架作为实施的基础组件，又在该体系基础上创建了一些非常优秀的边缘组件将它们很好地整合起来。

加之 Spring Cloud 有其 Spring 的强大技术背景，极高的社区活跃度，也许未来 Spring Cloud 会成为微服务的标准技术解决方案；

第二章：认识 Spring Cloud

2-1. Spring Cloud 是什么

- 1、Spring Cloud 是一个一站式的开发分布式系统的框架，为开发者提供了一系列的构建分布式系统的工具集；
- 2、Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具（比如：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）；
- 3、开发分布式系统都需要解决一系列共同关心的问题，而使用 Spring Cloud 可以快速地实现这些分布式开发共同关心的问题，并能方便地在任何分布式环境中部署与运行。
- 4、Spring Cloud 这个一站式地分布式开发框架，被近年来流行的“微服务”架构所大力推崇，成为目前进行微服务架构的优先选择工具；
- 5、Spring Cloud 基于 Spring Boot 框架构建微服务架构，学习 Spring Cloud 需要先学习 Spring Boot；

6、SpringCloud 官网：<http://spring.io>

2-2. Spring Cloud 的版本

Spring Cloud 最早是从 2014 年推出的，在推出的前期更新迭代速度非常快，频繁发布新版本，目前更趋于稳定，变化稍慢一些；
Spring Cloud 的版本并不是传统的使用数字的方式标识，而是使用诸如：Angel、Brixton、Camden.....等伦敦的地名来命名版本，版本的先后顺序使用字母表 A-Z 的先后来标识，现在已经进入 F 版本；

Spring Cloud 与 Spring Boot 版本匹配关系

Finchley	兼容 Spring Boot 2.0.x, 不兼容 Spring Boot 1.5.x
Edgware	兼容 Spring Boot 1.5.x, 不兼容 Spring Boot 2.0.x
Dalston	兼容 Spring Boot 1.5.x, 不兼容 Spring Boot 2.0.x
Camden	兼容 Spring Boot 1.4.x, 也兼容 Spring Boot 1.5.x
Brixton	兼容 Spring Boot 1.3.x, 也兼容 Spring Boot 1.4.x
Angel	兼容 Spring Boot 1.2.x

Spring Cloud 并不是从 0 开始开发一整套微服务解决方案，而是集成各个开源软件，构成一整套的微服务解决方案，这其中也有非常著名的 Netflix 公司的开源产品；

Netflix 公司成立于 1997 年，是目前美国最大的版权视频交易网站；
Netflix 公司在不断发展的过程中，也成为了一家云计算公司，并积极参与开源项目，Netflix OSS (Open Source) 就是由 Netflix 公司主持开发的一套代码

框架和库，github 地址：<https://github.com/Netflix>；

Spring Cloud 所包含的众多组件中，Spring Cloud Netflix 就是其中一组不可忽视的组件，由 netflix 公司开发后又并入 Spring Cloud 大家庭；

目前 Netflix 公司贡献的活跃项目包括：

spring-cloud-netflix-eureka

spring-cloud-netflix-hystrix

spring-cloud-netflix-stream

spring-cloud-netflix-archaius

spring-cloud-netflix-ribbon

spring-cloud-netflix-zuul

2-3. Spring Cloud 开发环境

SpringBoot 2.0.x

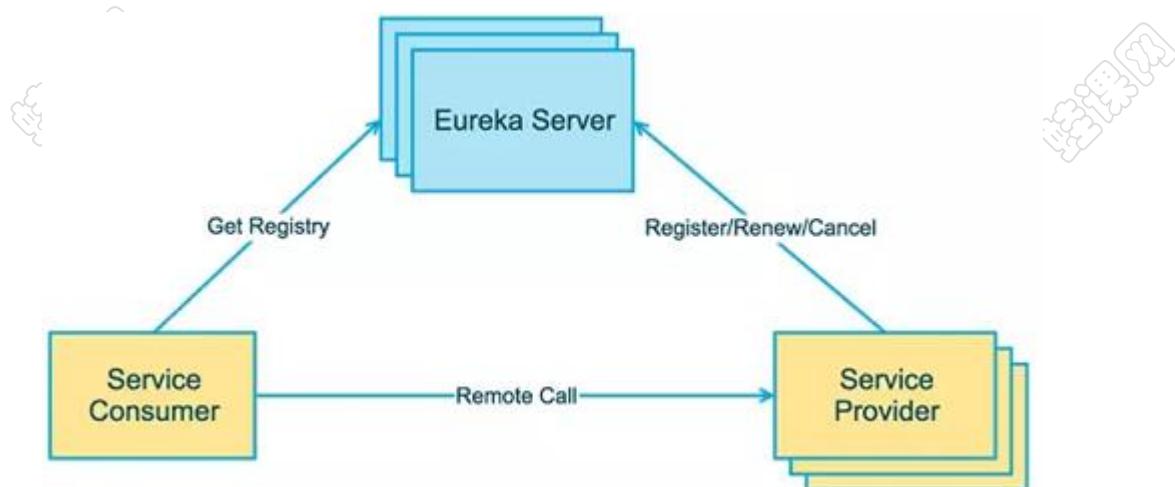
Spring Cloud Finchley RC2

Maven 3.5.3

JDK 1.8.152

IntelliJ IDEA

2-4. Spring Cloud 的整体架构



Service Provider: 暴露服务的服务提供方。

Service Consumer: 调用远程服务的服务消费方。

Eureka Server: 服务注册中心和服务发现中心。

第三章：SpringCloud 快速开发入门

3-1. 搭建和配置一个服务提供者

我们知道，SpringCloud 构建微服务是基于 SpringBoot 开发的。

1、创建一个 SpringBoot 工程，并且添加 SpringBoot 的相关依赖；

2、创建服务提供者的访问方法，也就是后续消费者如何访问提供者；

Spring Cloud 是基于 rest 的访问，所以我们添加一个 Controller，在该 Controller 中提供一个访问入口：

```
@RestController
public class HelloController {
    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    public String hello() {
        return "Hello Spring Cloud";
    }
}
```

3、启动运行该 SpringBoot 程序，访问该 controller；

3-2. 搭建和配置一个服务消费者

服务消费者也是一个 SpringBoot 项目，服务消费者主要用来消费服务提供者提供的服务；

1、创建一个 SpringBoot 工程，并且添加 SpringBoot 的相关依赖；

2、开发一个消费者方法，去消费服务提供者提供的服务，这个消费者方法也是一个 Controller：

```
@RestController
public class ConsumerController {
```

```
@Autowired  
  
private RestTemplate restTemplate;  
  
@RequestMapping(value="/cloud/hello")  
public String helloController() {  
    return restTemplate.getEntity("http://localhost:9100/hello",  
        String.class).getBody();  
}  
}
```

3、启动该 SpringBoot 程序，测试服务消费者调用服务提供者；

3-3. 走进服务注册中心 Eureka

在微服务架构中，服务注册与发现是核心组件之一，手动指定每个服务是很低效的，Spring Cloud 提供了多种服务注册与发现的实现方式，例如：Eureka、Consul、Zookeeper。

Spring Cloud 支持得最好的是 Eureka，其次是 Consul，再次是 Zookeeper。

什么是服务注册？

服务注册：将服务所在主机、端口、版本号、通信协议等信息登记到注册中心上；

什么是服务发现？

服务发现：服务消费者向注册中心请求已经登记的服务列表，然后得到某个服务的主机、端口、版本号、通信协议等信息，从而实现对具体服务的调用；

Eureka 是什么？

Eureka 是一个服务治理组件，它主要包括服务注册和服务发现，主要用来搭建服务注册中心。

Eureka 是一个基于 REST 的服务，用来定位服务，进行中间层服务器的负载均衡和故障转移；

Eureka 是 Netflix 公司开发的，Spring Cloud 封装了 Netflix 公司开发的 Eureka 模块来实现服务注册和发现，也就是说 Spring Cloud 对 Netflix Eureka 做了二次封装；

Eureka 采用了 C-S (客户端/服务端) 的设计架构，也就是 Eureka 由两个组件组成：Eureka 服务端和 Eureka 客户端。Eureka Server 作为服务注册的服务端，它是服务注册中心，而系统中的其他微服务，使用 Eureka 的客户端连接到 Eureka Server 服务端，并维持心跳连接，Eureka 客户端是一个 Java 客户端，用来简化与服务器的交互、负载均衡、服务的故障切换等；有了 Eureka 注册中心，系统的维护人员就可以通过 Eureka Server 来监控系统中各个微服务是否正常运行。

3-4. Eureka 与 Zookeeper 的比较

著名的 CAP 理论指出，一个分布式系统不可能同时满足 C(一致性)、A(可用性) 和 P(分区容错性)。

由于分区容错性在是分布式系统中必须要保证的，因此我们只能在 A 和 C 之间进行权衡，在此 Zookeeper 保证的是 CP，而 Eureka 则是 AP。

Zookeeper 保证 CP

在 ZooKeeper 中，当 master 节点因为网络故障与其他节点失去联系时，剩余节点会重新进行 leader 选举，但是问题在于，选举 leader 需要一定时间，且选举期间整个 ZooKeeper 集群都是不可用的，这就导致在选举期间注册服务瘫痪。在云部署的环境下，因网络问题使得 ZooKeeper 集群失去 master 节点是大概率事件，虽然服务最终能够恢复，但是在选举时间内导致服务注册长期不可用是难以容忍的。

Eureka 保证 AP

Eureka 优先保证可用性，Eureka 各个节点是平等的，某几个节点挂掉不会影响正常节点的工作，剩余的节点依然可以提供注册和查询服务。而 Eureka 的客户端在向某个 Eureka 注册或时如果发现连接失败，则会自动切换至其它节点，只要有一台 Eureka 还在，就能保证注册服务可用(保证可用性)，只不过查到的信息可能不是最新的(不保证强一致性)。



所以 Eureka 在网络故障导致部分节点失去联系的情况下，只要有一个节点可用，那么注册和查询服务就可以正常使用，而不会像 zookeeper 那样使整个注册服务瘫痪，Eureka 优先保证了可用性。

3-5. 搭建与配置 Eureka 服务注册中心

Spring Cloud 要使用 Eureka 注册中心非常简单和方便，Spring Cloud 中的 Eureka 服务注册中心实际上也是一个 Spring Boot 工程，我们只需通过引入相关依赖和注解配置就能让 Spring Boot 构建的微服务应用轻松地与 Eureka 进行整合。

具体步骤如下：

1、创建一个 SpringBoot 项目，并且添加 SpringBoot 的相关依赖；

03-springcloud-eureka-server

2、添加 eureka 的依赖：

```
<!--Spring Cloud 的eureka-server 起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

3、在 Spring Boot 的入口类上添加一个@EnableEurekaServer 注解，用于开启 Eureka 注册中心服务端

4、在 application.properties 文件中配置 Eureka 服务注册中心信息：

```
#内嵌定时tomcat的端口
server.port=8761
#设置该服务注册中心的hostname
eureka.instance.hostname=localhost
#由于我们目前创建的应用是一个服务注册中心，而不是普通的应用，默认情况下，这个应用会向注册中心（也是它自己）注册它自己，设置为false 表示禁止这种自己向自己注册的默认行为
eureka.client.register-with-eureka=false
#表示不去检索其他的服务，因为服务注册中心本身的职责就是维护服务实例，它不需要去检索其他服务
eureka.client.fetch-registry=false
#指定服务注册中心的位置
eureka.client.service-url.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka
```

3-6. 启动与测试 Eureka 服务注册中心

- 1、完成上面的项目搭建后，我们就可以启动 SpringBoot 程序，main 方法运行；
- 2、启动成功之后，通过在浏览器地址栏访问我们的注册中心；

3-7. 向 Eureka 服务注册中心注册服务

我们前面搭建了服务提供者项目，接下来我们就可以将该服务提供者注册到 Eureka 注册中心，步骤如下：

- 1、在该服务提供者中添加 eureka 的依赖，因为服务提供者向注册中心注册服务，需要连接 eureka，所以需要 eureka 客户端的支持；

```
<!--SpringCloud 集成eureka 客户端的起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

- 2、激活 Eureka 中的 EnableEurekaClient 功能：

在 Spring Boot 的入口函数处，通过添加@EnableEurekaClient 注解来表明自己是一个 eureka 客户端，让我的服务提供者可以连接 eureka 注册中心；

- 3、配置服务名称和注册中心地址

```
spring.application.name=02-springcloud-service-provider
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

- 4、启动服务提供者 SpringBoot 程序的 main 方法运行；

5、启动运行之后，通过在浏览器地址栏访问我们之前搭建好的 eureka 注册中心，就可以看到有一个服务已经注册成功了；

3-8. 从 Eureka 服务注册中心发现与消费服务

我们已经搭建一个服务注册中心，同时也向这个服务注册中心注册了服务，接下来我们就可以发现和消费服务了，这其中服务的发现由 eureka 客户端实现，而服务的消费由 Ribbon 实现，也就是说服务的调用需要 eureka 客户端和 Ribbon

两者配合起来才能实现；

Eureka 客户端是什么？

Eureka 客户端是一个 Java 客户端，用来连接 Eureka 服务端，与服务端进行交互、负载均衡，服务的故障切换等；

Ribbon 是什么？

Ribbon 是一个基于 HTTP 和 TCP 的客户端负载均衡器，当使用 Ribbon 对服务进行访问的时候，它会扩展 Eureka 客户端的服务发现功能，实现从 Eureka 注册中心中获取服务端列表，并通过 Eureka 客户端来确定服务端是否已经启动。Ribbon 在 Eureka 客户端服务发现的基础上，实现了对服务实例的选择策略，从而实现对服务的负载均衡消费。

接下来我们来让服务消费者去消费服务：

我们前面搭建了服务消费者项目，接下来我们就可以使用该服务消费者通过注册中心去调用服务提供者，步骤如下：

1、在该消费者项目中添加 eureka 的依赖，因为服务消费者从注册中心获取服务，需要连接 eureka，所以需要 eureka 客户端的支持；

```
<!--SpringCloud 集成eureka 客户端的起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2、激活 Eureka 中的 EnableEurekaClient 功能：

在 Spring Boot 的入口函数处，通过添加@EnableEurekaClient 注解来表明自己是一个 eureka 客户端，让我的服务消费者可以使用 eureka 注册中心；

3、配置服务的名称和注册中心的地址：

```
spring.application.name=03-springcloud-web-consumer
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

4、前面我介绍了服务的发现由 eureka 客户端实现，而服务的真正调用由 ribbon 实现，所以我们需要在调用服务提供者时使用 ribbon 来调用：

```
@LoadBalanced
@Bean
```

```
public RestTemplate restTemplate () {  
    return new RestTemplate();  
}
```

加入了 ribbon 的支持，那么在调用时，即可改为使用服务名称来访问：

```
restTemplate.getForEntity("http://01-SPRINGCLOUD-SERVICE-PROVIDER/cloud/hello",  
String.class).getBody();
```

5、完成上面的步骤后，我们就可以启动消费者的 SpringBoot 程序，main 方法运行；

6、启动成功之后，通过在浏览器地址栏访问我们的消费者，看是否可以正常调用远程服务提供者提供的服务；

第四章：服务注册中心 Eureka

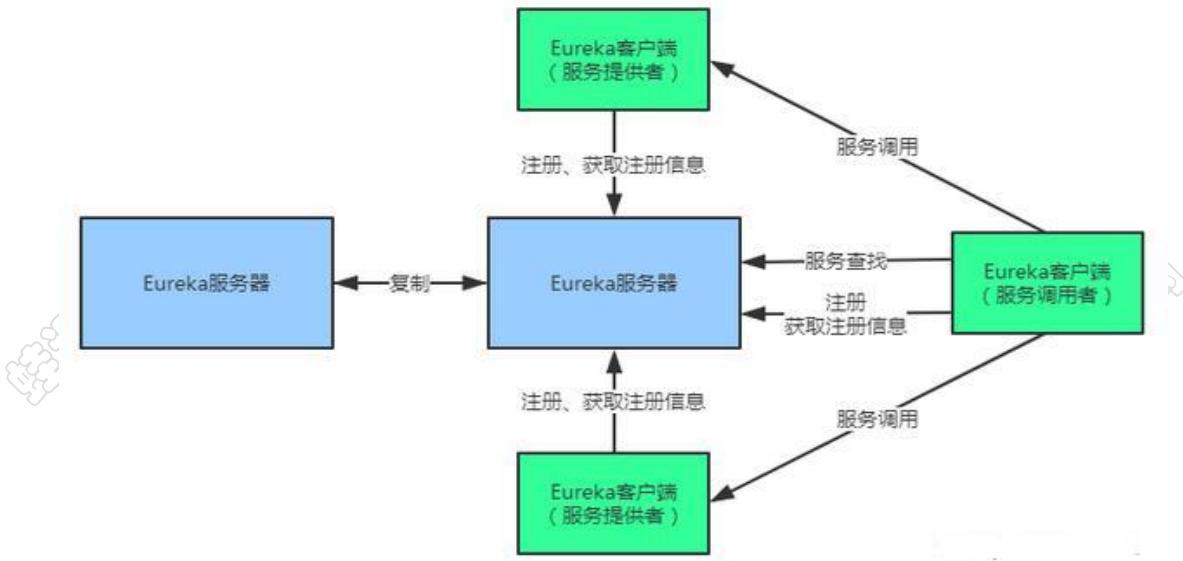
4-1. Eureka 注册中心高可用集群概述

在微服务架构的这种分布式系统中，我们要充分考虑各个微服务组件的高可用性问题，不能有单点故障，由于注册中心 eureka 本身也是一个服务，如果它只有一个节点，那么它有可能发生故障，这样我们就不能注册与查询服务了，所以我们需要一个高可用的服务注册中心，这就需要通过注册中心集群来解决。

eureka 服务注册中心它本身也是一个服务，它也可以看做是一个提供者，又可以看做是一个消费者，我们之前通过配置：

eureka.client.register-with-eureka=false 让注册中心不注册自己，但是我们可以向其他注册中心注册自己；

Eureka Server 的高可用实际上就是将自己作为服务向其他服务注册中心注册自己，这样就会形成一组互相注册的服务注册中心，进而实现服务清单的互相同步，往注册中心 A 上注册的服务，可以被复制同步到注册中心 B 上，所以从任何一台注册中心上都能查询到已经注册的服务，从而达到高可用的效果。



4-2. Eureka 注册中心高可用集群搭建

我们知道，Eureka 注册中心高可用集群就是各个注册中心相互注册，所以：

1.在 8761 的配置文件中，让它的 service-url 指向 8762，在 8762 的配置文件中让它的 service-url 指向 8761

2.由于 8761 和 8762 互相指向对方，实际上我们构建了一个双节点的服务注册中心集群

```
eureka.client.service-url.defaultZone=http://eureka8762:8762/eureka/
eureka.client.service-url.defaultZone=http://eureka8761:8761/eureka/
```

然后在本地 hosts 文件配置：C:\Windows\System32\drivers\etc\hosts

```
127.0.0.1 eureka8761
127.0.0.1 eureka8762
```

运行时，在运行配置项目 Program Arguments 中配置：

```
--spring.profiles.active=eureka8761
--spring.profiles.active=eureka8762
```

分别启动两个注册中心，访问两个注册中心页面，观察注册中心页面是否正常；

4-3. Eureka 注册中心高可用集群测试

在要进行注册的服务中配置：

```
eureka.client.service-url.defaultZone=http://eureka8761:8761/eureka/,http://eureka8762:8762/eureka/
```

启动服务提供者服务，然后观察注册中心页面，可以看到服务会在两个注册中心上都注册成功；

4-4. Eureka 服务注册中心自我保护机制

自我保护机制是 Eureka 注册中心的重要特性，当 Eureka 注册中心进入自我保护模式时，在 Eureka Server 首页会输出如下警告信息：

EMERGENCY! EUREKA MAY BE INCORRECTLY CLAIMING INSTANCES ARE UP WHEN THEY'RE NOT. RENEWALS ARE LESSER THAN THRESHOLD AND HENCE THE INSTANCES ARE NOT BEING EXPIRED JUST TO BE SAFE.

emergency! eureka may be incorrectly claiming instances are up when they're not. renewals are lesser than threshold and hence the instances are not being expired just to be safe.

在没有 Eureka 自我保护的情况下，如果 Eureka Server 在一定时间内没有接收到某个微服务实例的心跳，Eureka Server 将会注销该实例，但是当发生网络分区故障时，那么微服务与 Eureka Server 之间将无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是正常的，此时不应该注销这个微服务，如果没有自我保护机制，那么 Eureka Server 就会将此服务注销掉。

Eureka 通过“自我保护模式”来解决这个问题——当 Eureka Server 节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么就会把这个微服务节点进行保护。一旦进入自我保护模式，Eureka Server 就会保护服务注册表中的信息，不删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该 Eureka Server 节点会再自动退出自我保护模式。

所以，自我保护模式是一种应对网络异常的安全保护措施，它的架构哲学是宁可

同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务，使用自我保护模式，可以让 Eureka 集群更加的健壮、稳定。

当然也可以使用配置项：`eureka.server.enable-self-preservation = false` 禁用自我保护模式。

但是 Eureka Server 自我保护模式也会给我们带来一些困扰，如果在保护期内某个服务提供者刚好非正常下线了，此时服务消费者就会拿到一个无效的服务实例，此时会调用失败，对于这个问题需要服务消费者端具有一些容错机制，如重试，断路器等。

Eureka 的自我保护模式是有意义的，该模式被激活后，它不会从注册列表中剔除因长时间没收到心跳导致注册过期的服务，而是等待修复，直到心跳恢复正常之后，它自动退出自我保护模式。这种模式旨在避免因网络分区故障导致服务不可用的问题。

例如，两个微服务客户端实例 A 和 B 之间有调用的关系，A 是消费者，B 是提供者，但是由于网络故障，B 未能及时向 Eureka 发送心跳续约，这时候 Eureka 不能简单的将 B 从注册表中剔除，因为如果剔除了，A 就无法从 Eureka 服务器中获取 B 注册的服务，但是这时候 B 服务是可用的；

所以，Eureka 的自我保护模式最好还是开启它。

关于自我保护常用几个配置如下：

服务器端配置：

```
#测试时关闭自我保护机制，保证不可用服务及时踢出  
eureka.server.enable-self-preservation=false
```

客户配置：

```
#每间隔 2s, 向服务端发送一次心跳, 证明自己依然“存活”
eureka.instance.lease-renewal-interval-in-seconds=2
#告诉服务端, 如果我 10s 之内没有给你发心跳, 就代表我故障了, 将我踢出掉
eureka.instance.lease-expiration-duration-in-seconds=10
```

第五章：客户端负载均衡 Ribbon

5-1. Spring Cloud 中的 Ribbon 是什么？

我们通常说的负载均衡是指将一个请求均匀地分摊到不同的节点单元上执行，负载均分为硬件负载均衡和软件负载均衡：

硬件负载均衡：比如 F5、深信服、Array 等；

软件负载均衡：比如 Nginx、LVS、HAProxy 等；

硬件负载均衡或是软件负载均衡，他们都会维护一个可用的服务端清单，通过心跳检测来剔除故障的服务端节点以保证清单中都是可以正常访问的服务端节点。当客户端发送请求到负载均衡设备的时候，该设备按某种算法（比如轮询、权重、最小连接数等）从维护的可用服务端清单中取出一台服务端的地址，然后进行转发。

Ribbon 是 Netflix 发布的开源项目，主要功能是提供客户端的软件负载均衡算法，是一个基于 HTTP 和 TCP 的客户端负载均衡工具。

Spring Cloud 对 Ribbon 做了二次封装，可以让我们使用 RestTemplate 的服务请求，自动转换成客户端负载均衡的服务调用。

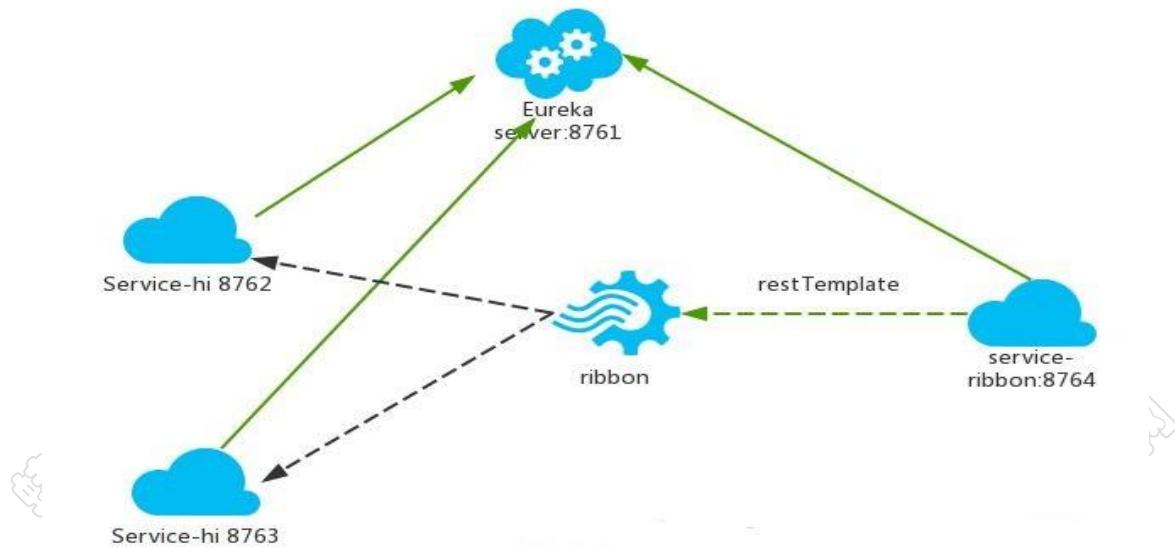
Ribbon 支持多种负载均衡算法，还支持自定义的负载均衡算法。

Ribbon 只是一个工具类框架，比较小巧，Spring Cloud 对它封装后使用也非常方便，它不像服务注册中心、配置中心、API 网关那样需要独立部署，Ribbon 只需要在代码直接使用即可；

Ribbon 与 Nginx 的区别

Ribbon 是客户端的负载均衡工具，而客户端负载均衡和服务端负载均衡最大的区别在于服务清单所存储的位置不同，在客户端负载均衡中，所有客户端节点下的服务端清单，需要自己从服务注册中心上获取，比如 Eureka 服务注册中心。同服务端负载均衡的架构类似，在客户端负载均衡中也需要心跳去维护服务端清单的健康性，只是这个步骤需要与服务注册中心配合完成。在 Spring Cloud 中，由于 Spring Cloud 对 Ribbon 做了二次封装，所以默认会创建针对 Ribbon 的自动化整合配置；

在 Spring Cloud 中，Ribbon 主要与 RestTemplate 对象配合起来使用，Ribbon 会自动化配置 RestTemplate 对象，通过 @LoadBalanced 开启 RestTemplate 对象调用时的负载均衡。



5-2. Ribbon 实现客户端负载均衡

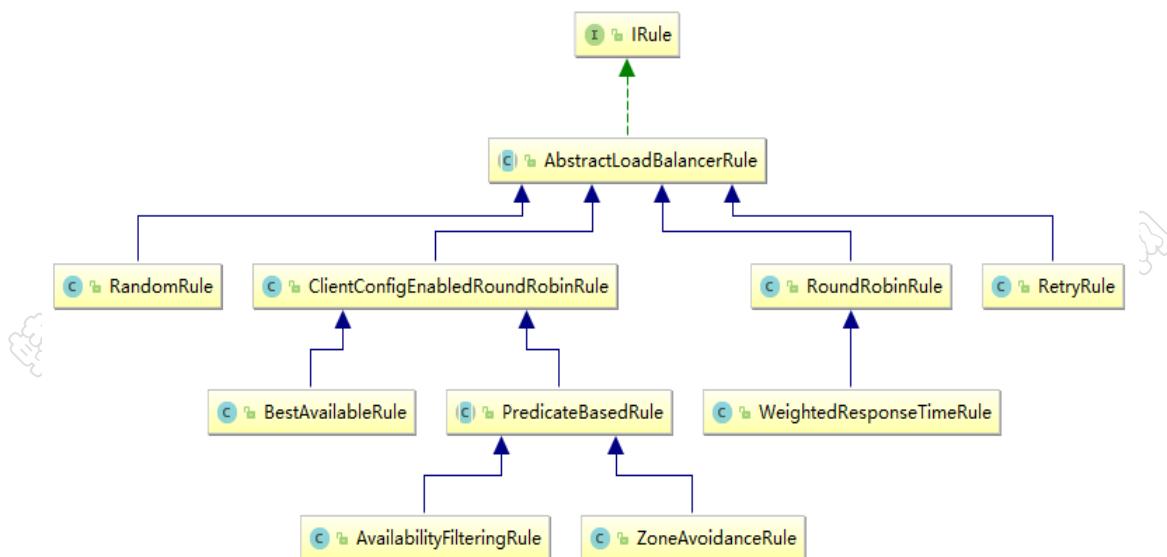
由于 Spring Cloud Ribbon 的封装，我们在微服务架构中使用客户端负载均衡调用非常简单，只需要如下两步：

- 1、启动多个服务提供者实例并注册到一个服务注册中心或是服务注册中心集群。
- 2、服务消费者通过被@LoadBalanced 注解修饰过的 RestTemplate 来调用服务提供者。

这样，我们就可以实现服务提供者的高可用以及服务消费者的负载均衡调用。

5-3. Ribbon 负载均衡策略

Ribbon 的负载均衡策略是由 IRule 接口定义，该接口由如下实现：



RandomRule	随机
RoundRobinRule	轮询
AvailabilityFilteringRule	先过滤掉由于多次访问故障的服务，以及并发连接数超过阈值的服务，然后对剩下的服务按照轮询策略进行访问；
WeightedResponseTimeRule	根据平均响应时间计算所有服务的权重，响应时间越快服务权重就越大被选中的概率即越高，如果服务刚启动时统计信息不足，则使用 RoundRobinRule 策略，待统计信息足够会切换到该 WeightedResponseTimeRule 策略；
RetryRule	先按照 RoundRobinRule 策略分发，如果分发到的服务不能访问，则在指定时间内进行重试，分发其他可用的服务；

BestAvailableRule	先过滤掉由于多次访问故障的服务，然后选择一个并发量最小的服务；
ZoneAvoidanceRule	综合判断服务节点所在区域的性能和服务节点的可用性，来决定选择哪个服务；

5-4. Rest 请求模板类解读

当我们从服务消费端去调用服务提供者的服务的时候，使用了一个极其方便的对象叫 `RestTemplate`，当时我们只使用了 `RestTemplate` 中最简单的一个功能 `getForEntity` 发起了一个 `get` 请求去调用服务端的数据，同时，我们还通过配置`@LoadBalanced` 注解开启客户端负载均衡，`RestTemplate` 的功能非常强大，那么接下来我们就来详细的看一下 `RestTemplate` 中几种常见请求方法的使用。

在日常操作中，基于 Rest 的方式通常是四种情况，它们分表是：

GET 请求 --查询数据

POST 请求 -添加数据

PUT 请求 - 修改数据

DELETE 请求 -删除数据

下面我们逐一解读。

5-5. RestTemplate 的 GET 请求

Get 请求可以有两种方式：

第一种：`getForEntity`

该方法返回一个 `ResponseEntity<T>` 对象，`ResponseEntity<T>` 是 Spring 对 HTTP 请求响应的封装，包括了几个重要的元素，比如响应码、`contentType`、`contentLength`、响应消息体等；

```
ResponseEntity<String> responseEntity =  
restTemplate.getForEntity("http://01-SPRINGCLOUD-SERVICE-PROVIDER/service/hello",  
String.class);
```



```
String body = responseEntity.getBody();
HttpStatus statusCode = responseEntity.getStatusCode();
int statusCodeValue = responseEntity.getStatusCodeValue();
HttpHeaders headers = responseEntity.getHeaders();

System.out.println(body);
System.out.println(statusCode);
System.out.println(statusCodeValue);
System.out.println(headers);
```

以上代码：

getForEntity 方法第一个参数为要调用的服务的地址，即服务提供者提供的 `http://01-SPRINGCLOUD-SERVICE-PROVIDER/service/hello` 接口地址，注意这里是通过服务名调用而不是服务地址，如果改为服务地址就无法使用 Ribbon 实现客户端负载均衡了。

getForEntity 方法第二个参数 String.class 表示希望返回的 body 类型是 String 类型，如果希望返回一个对象，也是可以的，比如 User 对象；

另外两个重载方法：

```
public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object...
uriVariables) throws RestClientException
```

比如：

```
restTemplate.getForEntity("http://01-SPRINGCLOUD-SERVICE-PROVIDER/service/hello?id={1}&name={2}", String.class, "{1, '张无忌'}").getBody();
```

```
public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
Map<String, ?> uriVariables) throws RestClientException
```

比如：

```
Map<String, Object> paramMap = new ConcurrentHashMap<>();
paramMap.put("id", 1);
paramMap.put("name", "张无忌");
restTemplate.getForEntity("http://01-SPRINGCLOUD-SERVICE-PROVIDER/service/hello?id={id}&name={name}", String.class, paramMap).getBody();
```

第二种：getForObject()

与 `getForEntity` 使用类似，只不过 `getForObject` 是在 `getForEntity` 基础上进行了再次封装，可以将 `http` 的响应体 `body` 信息转化成指定的对象，方便我们的代码开发；

当你不需要返回响应中的其他信息，只需要 `body` 体信息的时候，可以使用这个更方便；

它也有两个重载的方法，和 `getForEntity` 相似；

```
<T> T getForObject(URI url, Class<T> responseType) throws RestClientException;  
  
<T> T getForObject(String url, Class<T> responseType, Object... uriVariables) throws  
RestClientException;  
  
<T> T getForObject(String url, Class<T> responseType, Map<String, ?> uriVariables)  
throws RestClientException;
```

5-6. RestTemplate 的 POST 请求：

Post 与 Get 请求非常类似：

```
restTemplate.postForObject()  
restTemplate.postForEntity()  
restTemplate.postForLocation()
```

5-7. RestTemplate 的 PUT 请求：

```
restTemplate.put();
```

5-8. RestTemplate 的 DELETE 请求：

```
restTemplate.delete();
```

第六章：服务熔断 Hystrix

6-1. Hystrix 是什么

在微服务架构中，我们是将一个单体应用拆分成多个服务单元，各个服务单元之间通过注册中心彼此发现和消费对方提供的服务，每个服务单元都是单独部署，在各自的服务进程中运行，服务之间通过远程调用实现信息交互，那么当某个服

务的响应太慢或者故障，又或者因为网络波动或故障，则会造成调用者延迟或调用失败，当大量请求到达，则会造成请求的堆积，导致调用者的线程挂起，从而引发调用者也无法响应，调用者也发生故障。

比如电商中的用户下订单，我们有两个服务，一个下订单服务，一个减库存服务，当用户下订单时调用下订单服务，然后下订单服务又调用减库存服务，如果减库存服务响应延迟或者没有响应，则会造成下订单服务的线程挂起等待，如果大量的用户请求下订单，或导致大量的请求堆积，引起下订单服务也不可用，如果有另外一个服务依赖于订单服务，比如用户服务，它需要查询用户订单，那么用户服务查询订单也会引起大量的延迟和请求堆积，导致用户服务也不可用。

所以在微服务架构中，很容易造成服务故障的蔓延，引发整个微服务系统瘫痪不可用。

为了解决此问题，微服务架构中引入了一种叫熔断器的服务保护机制。

熔断器也有叫断路器，他们表示同一个意思，最早来源于微服务之父 Martin Fowler 的论文 CircuitBreaker 一文。“熔断器”本身是一种开关装置，用于在电路上保护线路过载，当线路中有电器发生短路时，能够及时切断故障电路，防止发生过载、发热甚至起火等严重后果。

微服务架构中的熔断器，就是当被调用方没有响应，调用方直接返回一个错误响应即可，而不是长时间的等待，这样避免调用时因为等待而线程一直得不到释放，避免故障在分布式系统间蔓延；

Spring Cloud Hystrix 实现了熔断器、线程隔离等一系列服务保护功能。该功能也是基于 Netflix 的开源框架 Hystrix 实现的，该框架的目标在于通过控制那些访问远程系统、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。Hystrix 具备服务降级、服务熔断、线程和信号隔离、请求缓存、请求合并以及服务监控等强大功能。

6-2. Hystrix 快速入门

在 SpringCloud 中使用熔断器 Hystrix 是非常简单和方便的，只需要简单两步即可：

1、添加依赖

```
<!--Spring Cloud 熔断器起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.4.RELEASE</version>
</dependency>
```

2、在入口类中使用@EnableCircuitBreaker 注解开启断路器功能，也可以使用一个名为@SpringCloudApplication 的注解代替主类上的三个注解；

3、在调用远程服务的方法上添加注解：

```
@HystrixCommand(fallbackMethod="error")
```

6-3. 服务消费者 Hystrix 测试

hystrix 默认超时时间是 1000 毫秒，如果你后端的响应超过此时间，就会触发断路器；

修改 hystrix 的默认超时时间：

```
@HystrixCommand(fallbackMethod="error", commandProperties={
    @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds",
value="1500")}) //熔断器，调用不通，回调error()方法
public String webHello () {
```

6-4. Hystrix 的服务降级

有了服务的熔断，随之就会有服务的降级，所谓服务降级，就是当某个服务熔断之后，服务端提供的服务将不再被调用，此时由客户端自己准备一个本地的 fallback 回调，返回一个默认值来代表服务端的返回；

这种做法，虽然不能得到正确的返回结果，但至少保证了服务的可用，比直接抛出错误或服务不可用要好很多，当然这需要根据具体的业务场景来选择；



6-5. Hystrix 的异常处理

我们在调用服务提供者时，我们自己也有可能会抛异常，默认情况下方法抛了异常会自动进行服务降级，交给服务降级中的方法去处理；

当我们自己发生异常后，只需要在服务降级方法中添加一个 Throwable 类型的参数就能够获取到抛出的异常的类型，如下：

```
public String error(Throwable throwable) {  
    System.out.println(throwable.getMessage());  
    return "error";  
}
```

此时我们可以在控制台看到异常的类型；

如果远程服务有一个异常抛出后我们不希望进入到服务降级方法中去处理，而是直接将异常抛给用户，那么我们可以在@HystrixCommand 注解中添加忽略异常，如下：

```
@HystrixCommand(fallbackMethod="error", ignoreExceptions = Exception.class)
```

自定义 Hystrix 请求的服务异常熔断处理

我们也可以自定义类继承自 HystrixCommand 来实现自定义的 Hystrix 请求，在 getFallback 方法中调用 getExecutionException 方法来获取服务抛出的异常；

```
com.netflix.hystrix.HystrixCommand.Setter.withGroupKey(HystrixCommandGroupKey.Factory.asKey(""))
```

6-6. Hystrix 仪表盘监控

Hystrix 仪表盘（Hystrix Dashboard），就像汽车的仪表盘实时显示汽车的各项数据一样，Hystrix 仪表盘主要用来监控 Hystrix 的实时运行状态，通过它我

们可以看到 Hystrix 的各项指标信息，从而快速发现系统中存在的问题进而解决它。

要使用 Hystrix 仪表盘功能，我们首先需要有一个 Hystrix Dashboard，这个功能我们可以在原来的消费者应用上添加，让原来的消费者应用具备 Hystrix 仪表盘功能，但一般地，微服务架构思想是推崇服务的拆分，Hystrix Dashboard 也是一个服务，所以通常会单独创建一个新的工程专门用做 Hystrix Dashboard 服务；

搭建一个 Hystrix Dashboard 服务的步骤：

第一步：创建一个普通的 Spring Boot 工程

比如创建一个名为 springcloud-hystrix-dashboard 的 Spring Boot 工程，建立好基本的结构和配置；

第二步：添加相关依赖

在创建好的 Spring Boot 项目的 pom.xml 文件中添加相关依赖，如下：

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

第三步：入口类上添加注解

添加好依赖之后，在入口类上添加@EnableHystrixDashboard 注解开启仪表盘功能，如下：

```
@SpringBootApplication
@EnableHystrixDashboard
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
```

```

    }
}

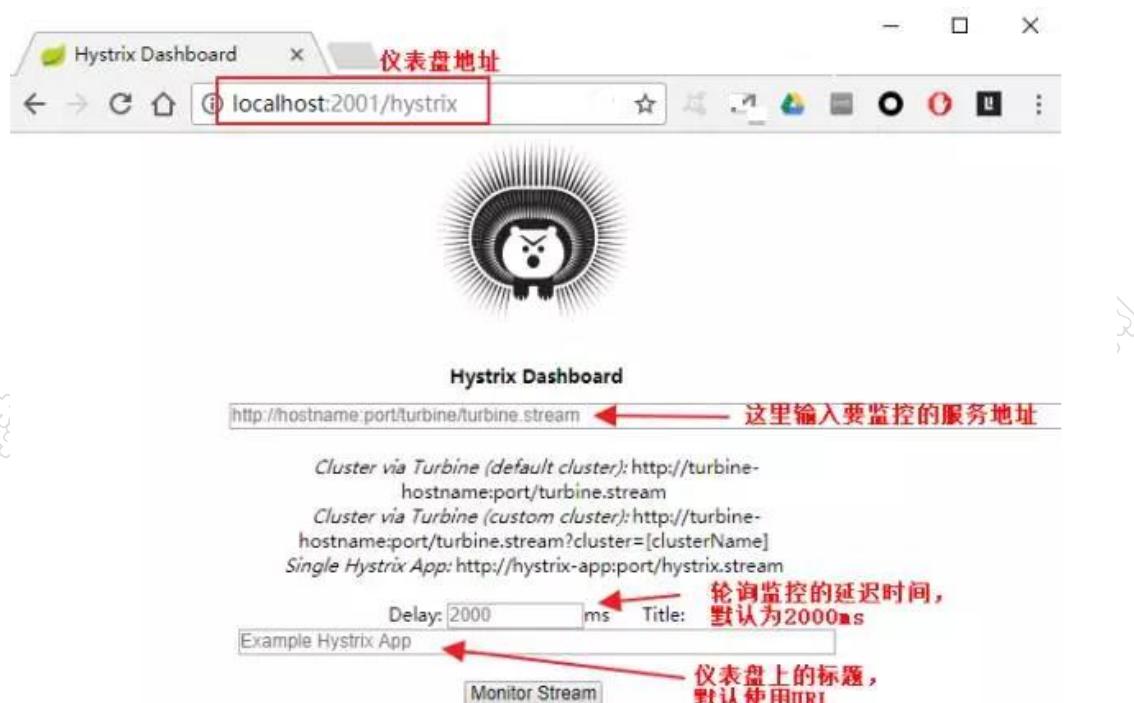
```

第四步：属性配置

最后，我们可以根据个人习惯配置一下 application.properties 文件，如下：

```
server.port=3721
```

至此，我们的 Hystrix 监控环境就搭建好了；



Hystrix 仪表盘工程已经创建好了，现在我们需要有一个服务，让这个服务提供一个路径为 /actuator/hystrix.stream 接口，然后就可以使用 Hystrix 仪表盘来对该服务进行监控了；

我们改造消费者服务，让其能提供 /actuator/hystrix.stream 接口，步骤如下：

1、消费者项目需要有 hystrix 的依赖：

```

<!--Spring Cloud 熔断器起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>

```

```
<version>1.4.5.RELEASE</version>
</dependency>
```

2、需要有一个 spring boot 的服务监控依赖：

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

3、配置文件需要配置 spring boot 监控端点的访问权限：

```
management.endpoints.web.exposure.include=*
```

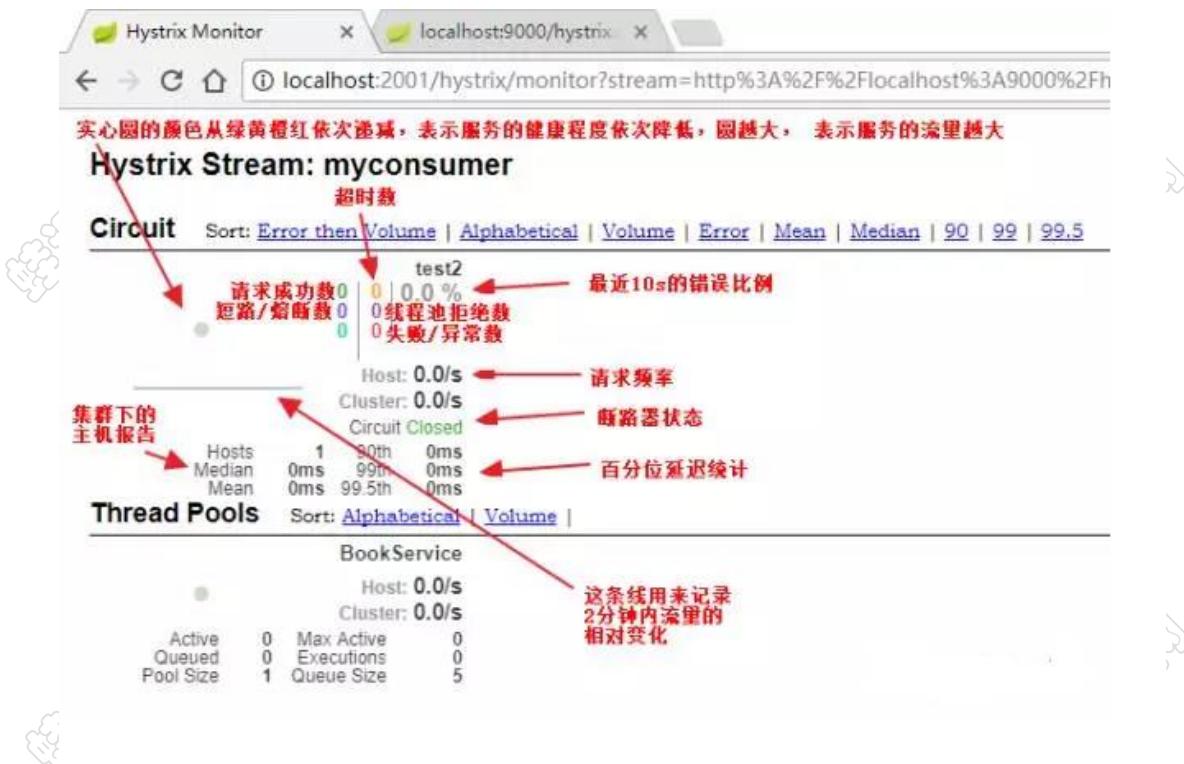
这个是用来暴露 endpoints 的，由于 endpoints 中会包含很多敏感信息，除了 health 和 info 两个支持直接访问外，其他的默认不能直接访问，所以我们让它都能访问，或者指定：

```
management.endpoints.web.exposure.include=hystrix.stream
```

4、访问入口 <http://localhost:8081/actuator/hystrix.stream>

 注意：这里有一个细节需要注意，要访问/hystrix.stream 接口，首先得访问 consumer 工程中的任意一个其他接口，否则直接访问/hystrix.stream 接口时会输出出一连串的 ping： ping: ..., 先访问 consumer 中的任意一个其他接口，然后再访问/hystrix.stream 接口即可；

6-7. Hystrix 仪表盘监控数据解读



第七章：声明式服务消费 Feign

7-1. Feign 是什么

Feign 是 Netflix 公司开发的一个声明式的 REST 调用客户端；
 Ribbon 负载均衡、Hystrix 服务熔断是我们 Spring Cloud 中进行微服务开发非常基础的组件，在使用的过程中我们也发现它们一般都是同时出现的，而且配置也都非常相似，每次开发都有很多相同的代码，因此 Spring Cloud 基于 Netflix Feign 整合了 Ribbon 和 Hystrix 两个组件，让我们的开发工作变得更加简单，就像 Spring Boot 是对 Spring+SpringMVC 的简化一样，Spring Cloud Feign 对 Ribbon 负载均衡、Hystrix 服务熔断进行简化，在其基础上进行了进一步的封装，不仅在配置上大大简化了开发工作，同时还提供了一种声明式的 Web 服务客户端定义方式；

7-2. 使用 Feign 实现消费者

使用 Feign 实现消费者，我们通过下面步骤进行：

第一步：创建普通 Spring Boot 工程

首先我们来创建一个普通的 Spring Boot 工程，取名为：

05-springcloud-service-feign；

第二步：添加依赖

要添加的依赖主要是 spring-cloud-starter-netflix-eureka-client 和 spring-cloud-starter-feign，如下：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-feign</artifactId>
    <version>1.4.5.RELEASE</version>
</dependency>
<!--Spring Cloud 熔断器起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-hystrix</artifactId>
    <version>1.4.5.RELEASE</version>
</dependency>
```

第三步：添加注解

在项目入口类上添加@EnableFeignClients 注解表示开启 Spring Cloud Feign 的支持功能；

第四步：声明服务

定义一个 HelloService 接口，通过@FeignClient 注解来指定服务名称，进而绑

定服务，然后再通过 SpringMVC 中提供的注解来绑定服务提供者提供的接口，如下：

```
@FeignClient("01-springcloud-service-provider")
public interface HelloService {
    @RequestMapping("/service/hello")
    public String hello();
}
```

这相当于绑定了一个名叫 01-springcloud-service-provider (这里

01-springcloud-service-provider 大小写 01-SPRINGCLOUD-SERVICE-PROVIDER 都可以) 的服务提供者提供的/service/hello 接口；

我们服务提供者提供的接口如下：

```
@GetMapping("/service/hello")
public String hello() {
    System.out.println("服务提供者 1. . . . .");
    return "Hello, Spring Cloud, Provider 1";
}
```

第五步：使用 Controller 中调用服务

接着来创建一个 Controller 来调用上面的服务，如下：

```
@RestController
public class FeignController {
    @Autowired
    HelloService helloService;
    @RequestMapping("/web/hello")
    public String hello() {
        return helloService.hello();
    }
}
```

第六步：属性配置

在 application.properties 中指定服务注册中心、端口号等信息，如下：

```
server.port=8082
#配置服务的名称
spring.application.name=05-springcloud-service-feign
#配置eureka 注册中心地址
```

```
eureka.client.service-url.defaultZone=http://eureka8761:8761/eureka/,http://eureka8762:8762/eureka/
```

第七步：测试

依次启动注册中心、服务提供者和 feign 实现服务消费者，然后访问如下地址：

<http://localhost:8082/web/hello>

7-3. 使用 Feign 实现消费者的测试

负载均衡：

我们知道，Spring Cloud 提供了 Ribbon 来实现负载均衡，使用 Ribbo 直接注入一个 RestTemplate 对象即可，RestTemplate 已经做好了负载均衡的配置；在 Spring Cloud 下，使用 Feign 也是直接可以实现负载均衡的，定义一个注解有@FeignClient 注解的接口，然后使用@RequestMapping 注解到方法上映射远程的 REST 服务，此方法也是做好负责均衡配置的。

服务熔断：

1、在 application.properties 文件开启 hystrix 功能

```
feign.hystrix.enabled=true
```

2、指定熔断回调逻辑

```
@FeignClient(name="01-springcloud-service-provider", fallback = MyFallback.class)
```

```
@Component
public class MyFallback implements HelloService {
    @Override
    public String hello() {
        return "发生异常了";
    }
}
```

3、服务熔断获取异常信息：

为 @FeignClient 修饰的接口加上 fallback 方法可以实现远程服务发生异常后进行服务的熔断，但是不能获取到远程服务的异常信息，如果要获取远程服务的异常信息，怎么办？此时可以使用 fallbackFactory：

样例代码：

```
@FeignClient(name="01-springcloud-service-provider", fallbackFactory=MyFallbackFactory.class)
```

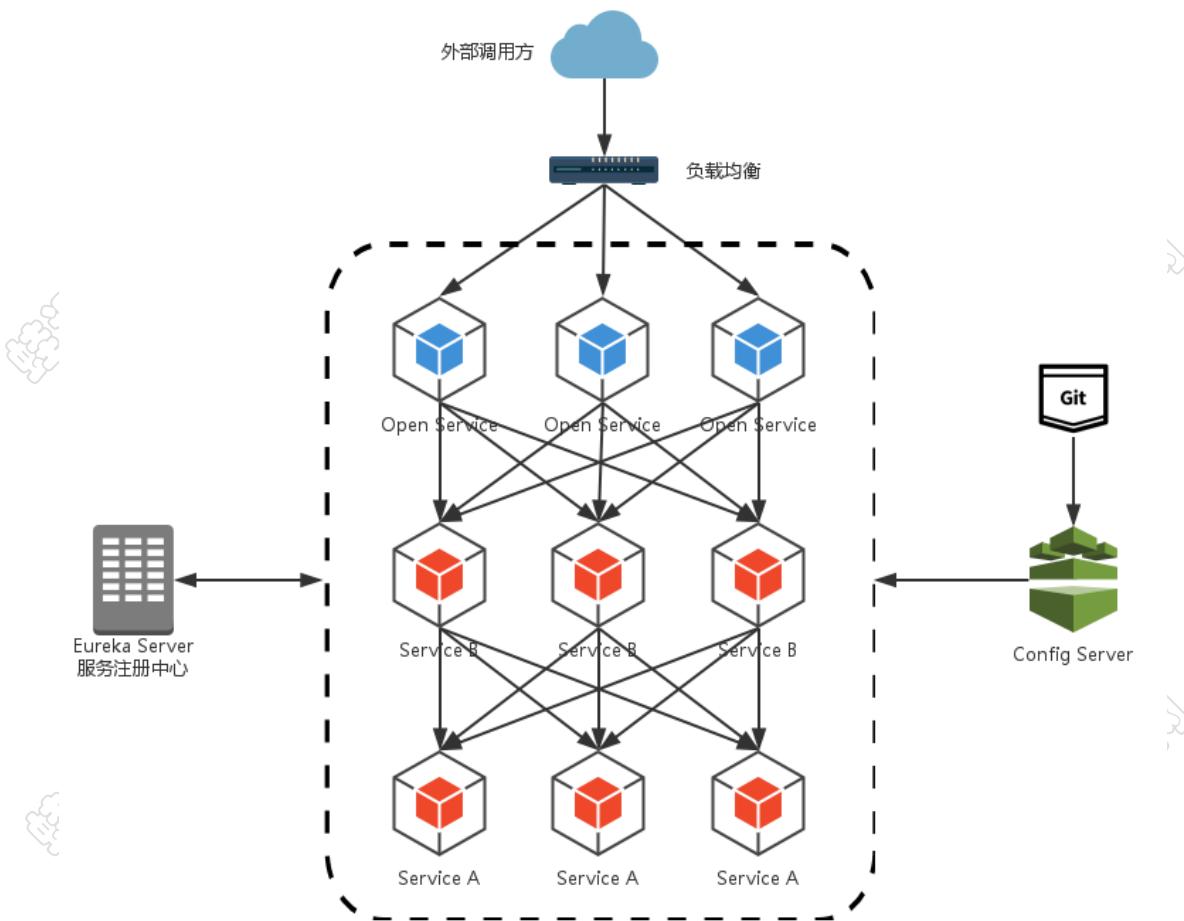
```
@Component
public class MyFallbackFactory implements FallbackFactory<HelloService> {
    @Override
    public HelloService create(Throwable throwable) {
        return new HelloService() {
            @Override
            public String hello() {
                return throwable.getMessage();
            }
        };
    }
}
```

第八章：API 网关 Zuul

8-1. Spring Cloud 的 Zuul 是什么

通过前面内容的学习，我们已经可以基本搭建出一套简略版的微服务架构了，我们有注册中心 Eureka，可以将服务注册到该注册中心中，我们有 Ribbon 或 Feign 可以实现对服务负载均衡地调用，我们有 Hystrix 可以实现服务的熔断，但是我们还缺少什么呢？

我们首先来看一个微服务架构图：



在上面的架构图中，我们的服务包括：内部服务 Service A 和内部服务 Service B，这两个服务都是集群部署，每个服务部署了 3 个实例，他们都会通过 Eureka Server 注册中心注册与订阅服务，而 Open Service 是一个对外的服务，也是集群部署，外部调用方通过负载均衡设备调用 Open Service 服务，比如负载均衡使用 Nginx，这样的实现是否合理，或者是否有更好的实现方式呢？接下来我们主要围绕该问题展开讨论。

- 1、如果我们的微服务中有很多个独立服务都要对外提供服务，那么我们要如何去管理这些接口？特别是当项目非常庞大的情况下要如何管理？
- 2、在微服务中，一个独立的系统被拆分成了很多个独立的服务，为了确保安全，

权限管理也是一个不可回避的问题，如果在每一个服务上都添加上相同的权限验证代码来确保系统不被非法访问，那么工作量也就太大了，而且维护也非常不方便。

为了解决上述问题，微服务架构中提出了 API 网关的概念，它就像一个安检站一样，所有外部的请求都需要经过它的调度与过滤，然后 API 网关来实现请求路由、负载均衡、权限验证等功能；

那么 Spring Cloud 这个一站式的微服务开发框架基于 Netflix Zuul 实现了 Spring Cloud Zuul，采用 Spring Cloud Zuul 即可实现一套 API 网关服务。

8-2. 使用 Zuul 构建 API 网关

1、创建一个普通的 Spring Boot 工程名为 06-springcloud-api-gateway，然后添加相关依赖，这里我们主要添加两个依赖 zuul 和 eureka 依赖：

```
<!-- 添加 spring cloud 的 zuul 的起步依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
</dependency>
<!-- 添加 spring cloud 的 eureka 的客户端依赖-->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

2、在入口类上添加@EnableZuulProxy 注解，开启 Zuul 的 API 网关服务功能：

```
@EnableZuulProxy //开启 Zuul 的 API 网关服务功能
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

3、在 application.properties 文件中配置路由规则：

```
#配置服务内嵌的 Tomcat 端口
server.port=8080

#配置服务的名称
spring.application.name=06-springcloud-api-gateway

#配置路由规则
zuul.routes.api-wkcto.path=/api-wkcto/**
zuul.routes.api-wkcto.serviceId=05-springcloud-service-feign

#配置API 网关到注册中心上, API 网关也将作为一个服务注册到eureka-server 上
eureka.client.service-url.defaultZone=http://eureka8761:8761/eureka/,http://
/eureka8762:8762/eureka/
```

以上配置，我们的路由规则就是匹配所有符合/api-wkcto/**的请求，只要路径中带有/api-wkcto/都将被转发到 05-springcloud-service-feign 服务上，至于 05-springcloud-service-feign 服务的地址到底是什么则由 eureka-server 注册中心去分析，我们只需要写上服务名即可。

以我们目前搭建的项目为例，请求 `http://localhost:8080/api-wkcto/web/hello` 接口则相当于请求 `http://localhost:8082/web/hello`（05-springcloud-service-feign 服务的地址为 `http://localhost:8082/web/hello`），路由规则中配置的 api-wkcto 是路由的名字，可以任意定义，但是一组 path 和 serviceId 映射关系的路由名要相同。

如果以上测试成功，则表示们的 API 网关服务已经构建成功了，我们发送的符合路由规则的请求将自动被转发到相应的服务上去处理。

8-3. 使用 Zuul 进行请求过滤

我们知道 Spring cloud Zuul 就像一个安检站，所有请求都会经过这个安检站，所以我们可以在这个安检站内实现对请求的过滤，下面我们以一个权限验证案例说这一点：

1、我们定义一个过滤器类并继承自 ZuulFilter，并将该 Filter 作为一个 Bean：

```
@Component
public class AuthFilter extends ZuulFilter {
    @Override
    public String filterType() {
        return "pre";
    }
    @Override
    public int filterOrder() {
        return 0;
    }
    @Override
    public boolean shouldFilter() {
        return true;
    }
    @Override
    public Object run() throws ZuulException {
        RequestContext ctx = RequestContext.getCurrentContext();
        HttpServletRequest request = ctx.getRequest();
        String token = request.getParameter("token");
        if (token == null) {
            ctx.setSendZuulResponse(false);
            ctx.setStatusCode(401);
        }
        ctx.addZuulResponseHeader("content-type", "text/html;charset=utf-8");
        ctx.setResponseBody("非法访问");
    }
}
```

(1) filterType 方法的返回值为过滤器的类型，过滤器的类型决定了过滤器在

哪个生命周期执行，pre 表示在路由之前执行过滤器，其他值还有 post、error、route 和 static，当然也可以自定义。

(2) filterOrder 方法表示过滤器的执行顺序，当过滤器很多时，我们可以通过该方法的返回值来指定过滤器的执行顺序。

(3) shouldFilter 方法用来判断过滤器是否执行，true 表示执行，false 表示不执行。

(4) run 方法则表示过滤的具体逻辑，如果请求地址中携带了 token 参数的话，则认为是合法请求，否则为非法请求，如果是非法请求的话，首先设置 ctx.setSendZuulResponse(false); 表示不对该请求进行路由，然后设置响应码和响应值。这个 run 方法的返回值目前暂时没有任何意义，可以返回任意值。

2、通过 <http://localhost:8080/api-wkcto/web/hello> 地址访问，就会被过滤器过滤。

8-4. Zuul 的路由规则

(1) 在前面的例子中：

```
#配置路由规则
zuul.routes.api-wkcto.path=/api-wkcto/**
zuul.routes.api-wkcto.serviceId=05-springcloud-service-feign
```

当访问地址符合/api-wkcto/**规则的时候，会被自动定位到 05-springcloud-service-feign 服务上，不过两行代码有点麻烦，还可以简化为：

```
zuul.routes.05-springcloud-service-feign=/api-wkcto/**
```

zuul.routes 后面跟着的是服务名，服务名后面跟着的是路径规则，这种配置方式更简单。



(2) 如果映射规则我们什么都不写，系统也给我们提供了一套默认的配置规则

默认的配置规则如下：

```
#默认的规则
zuul.routes.05-springcloud-service-feign.path=/05-springcloud-service-feign/**
zuul.routes.05-springcloud-service-feign.serviceId=05-springcloud-service-feign
```

(3) 默认情况下，Eureka 上所有注册的服务都会被 Zuul 创建映射关系来进行路由。

但是对于我这里的例子来说，我希望：

05-springcloud-service-feign 提供服务；

而 01-springcloud-service-provider 作为服务提供者只对服务消费者提供服务，不对外提供服务。

如果使用默认的路由规则，则 Zuul 也会自动为

01-springcloud-service-provider 创建映射规则，这个时候我们可以采用如下方式来让 Zuul 跳过 01-springcloud-service-provider 服务，不为其创建路由规则：

```
#忽略掉服务提供者的默认规则
zuul.ignored-services=01-springcloud-service-provider
```

不给某个服务设置映射规则，这个配置我们可以进一步细化，比如说我不想给 /hello 接口路由，那我们可以按如下方式配置：

```
#忽略掉某一些接口路径
zuul.ignored-patterns=/**/hello/**
```

此外，我们也可以统一的为路由规则增加前缀，设置方式如下：

```
#配置网关路由的前缀
zuul.prefix=/myapi
```

此时我们的访问路径就变成了 `http://localhost:8080/myapi/web/hello`

(4) 路由规则通配符的含义：

通配符	含义	举例	说明
<code>?</code>	匹配任意单个字符	<code>/05-springcloud-service-feign/?</code>	匹配 <code>/05-springcloud-service-feign/a,</code> <code>/05-springcloud-service-feign/b,</code> <code>/05-springcloud-service-feign/c</code> 等
<code>*</code>	匹配任意数量的字符	<code>/05-springcloud-service-feign/*</code>	匹配 <code>/05-springcloud-service-feign/aaa,</code> <code>/05-springcloud-service-feign/bbb,</code> <code>/05-springcloud-service-feign/ccc</code> 等, 无法匹配 <code>/05-springcloud-service-feign/a/b/c</code>
<code>**</code>	匹配任意数量的字符	<code>/05-springcloud-service-feign/**</code>	匹配 <code>/05-springcloud-service-feign/aaa,</code> <code>/05-springcloud-service-feign/bbb,</code> <code>/05-springcloud-service-feign/ccc</code> 等, 也可以匹配 <code>/05-springcloud-service-feign/a/b/c</code>

(5) 一般情况下 API 网关只是作为各个微服务的统一入口，但是有时候我们可能也需要在 API 网关服务上做一些特殊的业务逻辑处理，那么我们可以让请求到达 API 网关后，再转发给自己本身，由 API 网关自己来处理，那么我们可以进行如下的操作：

在 06-springcloud-api-gateway 项目中新建如下 Controller：

```

@RestController
public class GateWayController {
    @RequestMapping("/api/local")
    public String hello() {
        return "exec the api gateway.";
    }
}

```

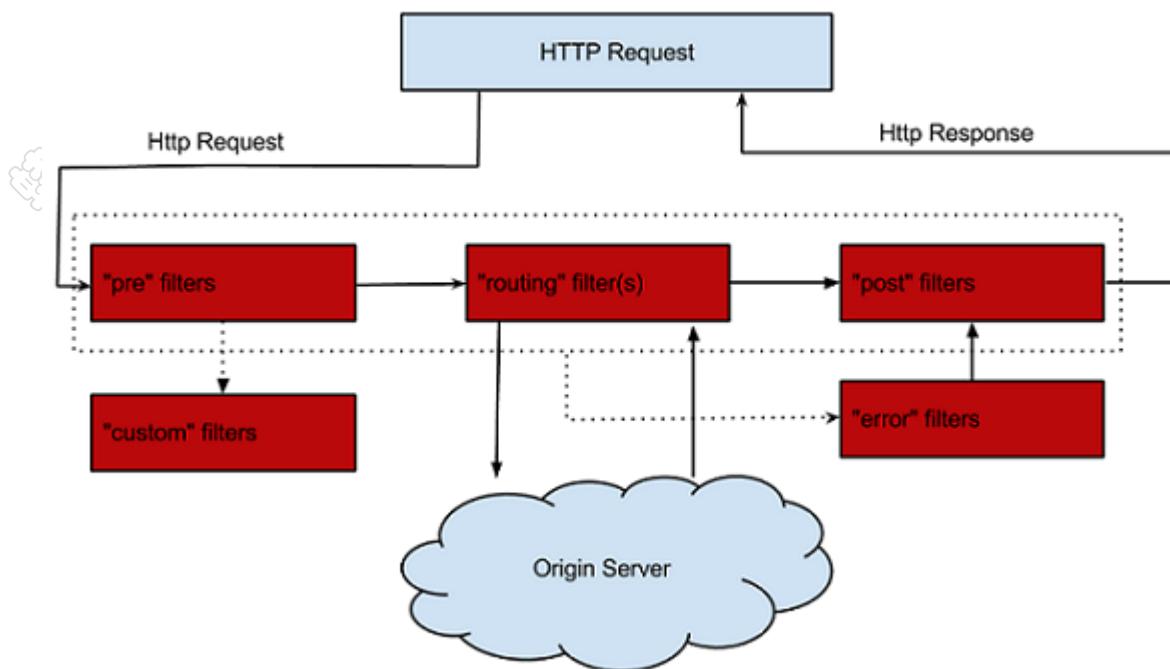
然后在 `application.properties` 文件中配置：

```
zuul.routes.gateway.path=/gateway/**  
zuul.routes.gateway.url=forward:/api/local
```

8-5. Zuul 的异常处理

Spring Cloud Zuul 对异常的处理是非常方便的，但是由于 Spring Cloud 处于迅速发展中，各个版本之间有所差异，本案例是以 Finchley.RELEASE 版本为例，来说明 Spring Cloud Zuul 中的异常处理问题。

首先我们来看一张官方给出的 Zuul 请求的生命周期图：



1. 正常情况下所有的请求都是按照 pre、route、post 的顺序来执行, 然后由 post 返回 response

2. 在 pre 阶段, 如果有自定义的过滤器则执行自定义的过滤器

3. pre、routing、post 的任意一个阶段如果抛异常了, 则执行 error 过滤器

我们可以有两种方式统一处理异常:

1、禁用 zuul 默认的异常处理 SendErrorFilter 过滤器, 然后自定义我们自己的

Errorfilter 过滤器

```
zuul.SendErrorFilter.error.disable=true
```

```
@Component
public class ErrorFilter extends ZuulFilter {
    private static final Logger logger =
LoggerFactory.getLogger(ErrorFilter.class);
    @Override
    public String filterType() {
        return "error";
    }
    @Override
    public int filterOrder() {
        return 1;
    }
    @Override
    public boolean shouldFilter() {
        return true;
    }
    @Override
    public Object run() throws ZuulException {
        try {
            RequestContext context = RequestContext.getCurrentContext();
            ZuulException exception = (ZuulException)context.getThrowable();
            logger.error("进入系统异常拦截", exception);
            HttpServletResponse response = context.getResponse();
            response.setContentType("application/json; charset=utf8");
            response.setStatus(exception.getStatusCode());
            PrintWriter writer = null;
            try {
                writer = response.getWriter();
                writer.print("{code:"+ exception.getStatusCode +",message:\\""+ exception.getMessage() +"\\"}");
            } catch (IOException e) {
                e.printStackTrace();
            } finally {
                if(writer!=null){
                    writer.close();
                }
            }
        } catch (Exception var5) {
            ReflectionUtils.rethrowRuntimeException(var5);
        }
    }
}
```

```
        }
        return null;
    }
}
```

2、自定义全局 error 错误页面

```
@RestController
public class ErrorHandlerController implements ErrorController {
    /**
     * 出异常后进入该方法，交由下面的方法处理
     */
    @Override
    public String getErrorPath() {
        return "/error";
    }
    @RequestMapping("/error")
    public Object error(){
        RequestContext ctx = RequestContext.getCurrentContext();
        ZuulException exception = (ZuulException)ctx.getThrowable();
        return exception.getStatusCode() + " -- " + exception.getMessage();
    }
}
```

第九章：Spring Cloud Config

9-1. Spring Cloud Config 是什么

在分布式系统中，尤其是当我们的分布式项目越来越多，每个项目都有自己的配置文件，对配置文件的统一管理就成了一种需要，而 Spring Cloud Config 就提供了对各个分布式项目配置文件的统一管理支持。Spring Cloud Config 也叫分布式配置中心，市面上开源的分布式配置中心有很多，比如国内的，360 的 QConf、淘宝的 diamond、百度的 disconf 都是解决分布式系统配置管理问题，国外也有很多开源的配置中心 Apache 的 Apache Commons Configuration、owner、cfg4j 等等；

Spring Cloud Config 是一个解决分布式系统的配置管理方案。它包含 Client

和 Server 两个部分，Server 提供配置文件的存储、以接口的形式将配置文件的内容提供出去，Client 通过接口获取数据、并依据此数据初始化自己的应用。Spring cloud 使用 git 或 svn 存放配置文件，默认情况下使用 git。

9-2. 构建 Springcloud config 配置中心

构建一个 spring cloud config 配置中心按照如下方式进行：

1、创建一个普通的 Spring Boot 项目

2、在 pom.xml 文件中添加如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
</dependency>
```

3、在入口类，也就是 main 方法的类上添加注解 `@EnableConfigServer`

4、在 application.properties 中配置一下 git 仓库信息，此处我们使用 GitHub

（也可以使用码云 gitee），首先在我的 GitHub 上创建一个名为 spring-cloud-config 的项目，创建之后，再做如下配置：

`server.port=3721`

`spring.application.name=07-springcloud-config-server`

```
spring.cloud.config.server.git.uri=https://github.com/myspring/spring-cloud-config.git
spring.cloud.config.server.git.search-paths=config-center
spring.cloud.config.server.git.username=xxxx@163.com
spring.cloud.config.server.git.password=xxxx123456
```

其中：

1. uri 表示配置中心所在仓库的位置
2. search-paths 表示仓库下的子目录



3. username 表示你的 GitHub 用户名

4. password 表示你的 GitHub 密码

至此我们的配置中心服务端就创建好了。

9-3. 构建 Springcloud config 配置中心仓库

接下来我们需要在 github 上设置好配置中心，首先在本地创建一个文件夹叫 wkcto，然后在里面创建一个文件夹叫 config-center，然后在 config-center 中创建四个配置文件，如下：

`application.properties`
`application-dev.properties`
`application-test.properties`
`application-online.properties`

在四个文件里面分别写上要测试的内容：

`url=http://www.wkcto.com`
`url=http://dev.wkcto.com`
`url=http://test.wkcto.com`
`url=http://online.wkcto.com`

然后回到 wkcto 目录下，依次执行如下命令将本地文件同步到 Github 仓库中：

说明：在使用命令之前先从 <https://git-scm.com/> 网站下载安装 git 的 window 客户端

1、添加提交人的账号信息，git 需要知道提交人的信息作为标识；

`git config --global user.name 'junge'`
`git config --global user.email 'junge@163.com'`

2、将该目录变为 git 可以管理的目录；

`git init`

2、将文件添加到暂存区；

`git add config-center/`



3、把文件提交到本地仓库；

```
git commit -m 'add config-center'
```

4、添加远程主机；

```
git remote add origin https://github.com/hnlylj/spring-cloud-config.git
```

5、将本地的 master 分支推送到 origin 主机；

```
git push -u origin master
```

至此，我们的配置文件就上传到 GitHub 上了。

此时启动我们的配置中心，通过/{application}/{profile}/{label}就能访问到我们的配置文件了；

其中：

{application} 表示配置文件的名字，对应的配置文件即 application，
{profile} 表示环境，有 dev、test、online 及默认，
{label} 表示分支，默认我们放在 master 分支上，

通过浏览器上访问 <http://localhost:3721/application/dev/master>

返回的 JSON 格式的数据：

name 表示配置文件名 application 部分，

profiles 表示环境部分，

label 表示分支，

version 表示 GitHub 上提交时产生的版本号，

同时当我们访问成功后，在控制台会打印了相关的日志信息；

当访问成功后配置中心会通过 git clone 命令将远程配置文件在本地也保存一份，以确保在 git 仓库故障时我们的应用还可以继续正常使用。

9-4. 构建 Springcloud config 配置中心客户端

前面已经搭建好了配置中心的服务端，接下来我们来看看如何在客户端应用中使用。

1、创建一个普通的 Spring Boot 工程 08-springcloud-config-client，并添加如下依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-config</artifactId>
</dependency>
```

2、创建 bootstrap.properties 文件，用于获取配置信息，文件内容如下：

(注意这些信息一定要放在 bootstrap.properties 文件中才有效)

```
server.port=3722

spring.application.name=application
spring.cloud.config.profile=dev
spring.cloud.config.label=master
spring.cloud.config.uri=http://localhost:3721/
```

其中：

name 对应配置文件中的 application 部分，

profile 对应了 profile 部分，

label 对应了 label 部分，

uri 表示配置中心的地址。

3、创建一个 Controller 进行测试：

```
@RestController
@RefreshScope
public class ConfigController {
    @Value("${url}")
```



```
private String url;  
@Autowired  
private Environment env;  
  
@RequestMapping("/cloud/url")  
public String url () {  
    return this.url;  
}  
@RequestMapping("/cloud/url2")  
public String url2 () {  
    return env.getProperty("url");  
}  
}
```

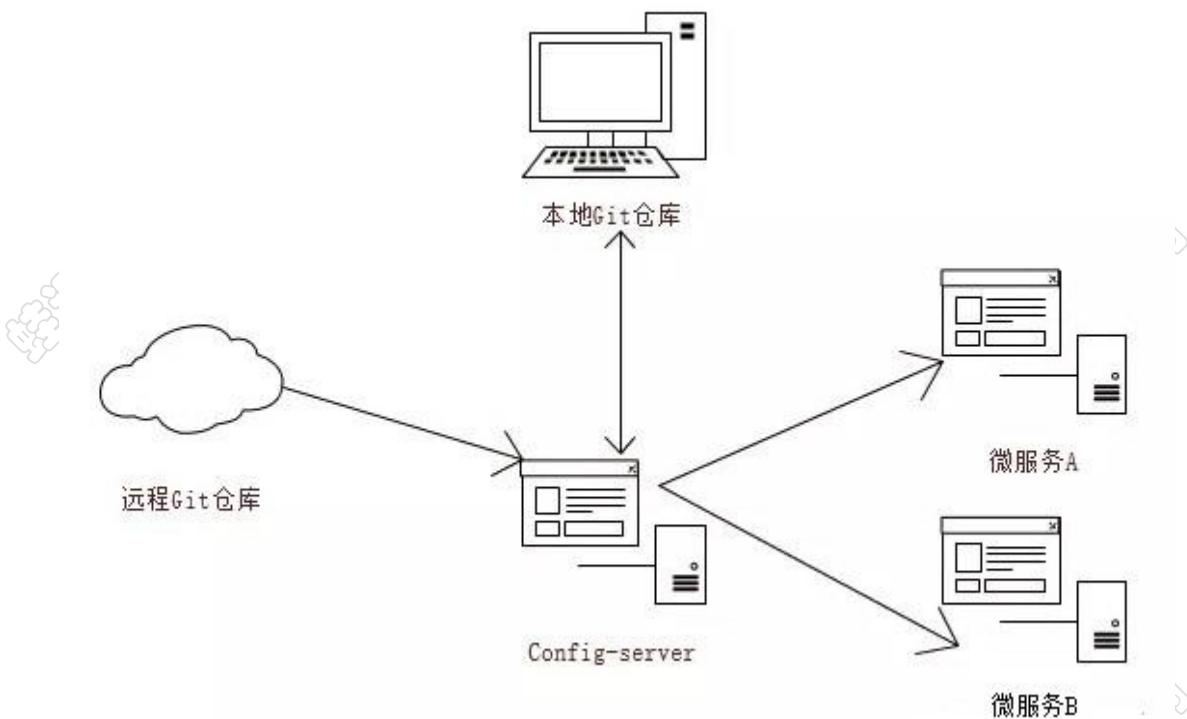
我们可以直接使用@Value 注解注入配置的属性值，也可以通过 Environment 对象来获取配置的属性值。

9-5. Springcloud config 配置中心客户端测试

通过客户端应用测试是否能够获取到配置中心配置的数据；

9-6. Springcloud config 的工作原理

Spring cloud Config Server 的工作过程如下图所示：



- 1、首先需要一个远程 Git 仓库，平时测试可以使用 GitHub，在实际生产环境中，需要自己搭建一个 Git 服务器，远程 Git 仓库的主要作用是用来保存我们的配置文件；
- 2、除了远程 Git 仓库之外，我们还需要一个本地 Git 仓库，每当 Config Server 访问远程 Git 仓库时，都会克隆一份到本地，这样当远程仓库无法连接时，就直接使用本地存储的配置信息；
- 3、微服务 A、微服务 B 则是我们的具体应用，这些应用在启动的时候会从 Config Server 中获取相应的配置信息；
4. 当微服务 A、微服务 B 尝试从 Config Server 中加载配置信息的时候，Config Server 会先通过 git clone 命令克隆一份配置文件保存到本地；
- 5、由于配置文件是存储在 Git 仓库中，所以配置文件天然具有版本管理功能；

9-7. Springcloud config 的安全保护

生产环境中我们的配置中心肯定是不能随随便便被人访问的，我们可以加上适当的保护机制，由于微服务是构建在 Spring Boot 之上，所以整合 Spring Security 是最方便的方式。

1、在 springcloud config server 项目添加依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

2、在 springcloud config server 项目的 application.properties 中配置用户名密码：

```
spring.security.user.name=wkcto
spring.security.user.password=123456
```

3、在 springcloud config client 上 bootstrap.properties 配置用户名和密码：

```
spring.cloud.config.username=wkcto
spring.cloud.config.password=123456
```

4、最后测试验证；