

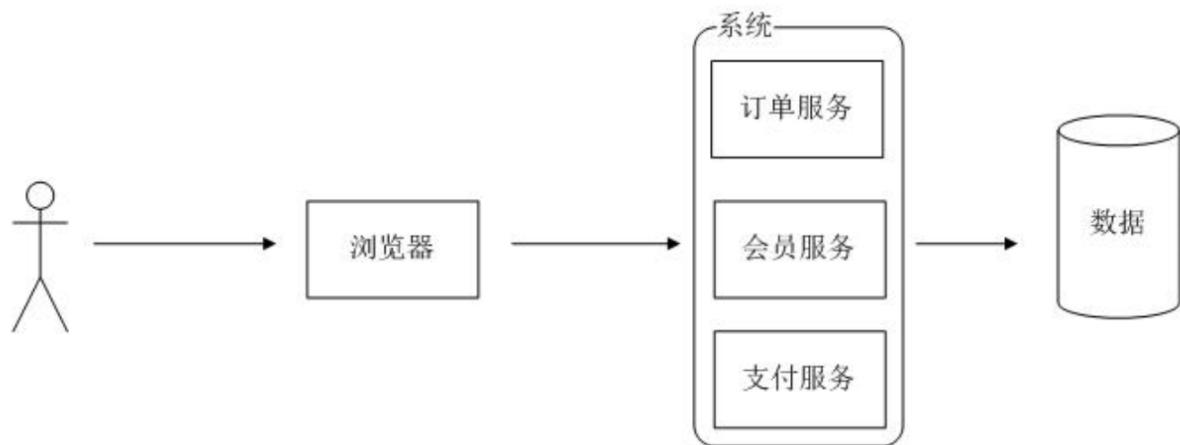
SpringCloud

1、微服务架构概述

1.1、系统架构发展

1.1.1、单体应用阶段

在互联网发展的初期，用户数量少，一般网站的流量也很少，但硬件成本较高。很多企业会将所有的功能都集成在一起开发一个单体应用，然后将单体应用部署到一台服务器上。一个简单的单本应用如图：



虽然应用是最初的架构，但是目前它并没有消失，还在不停的发展和演进，依然拥有巨大的市场。例如：采用MVC、MVP、MVVM开发的单体应用程序依然火爆，仍能够满足实际业务需求。

单体应用的优点

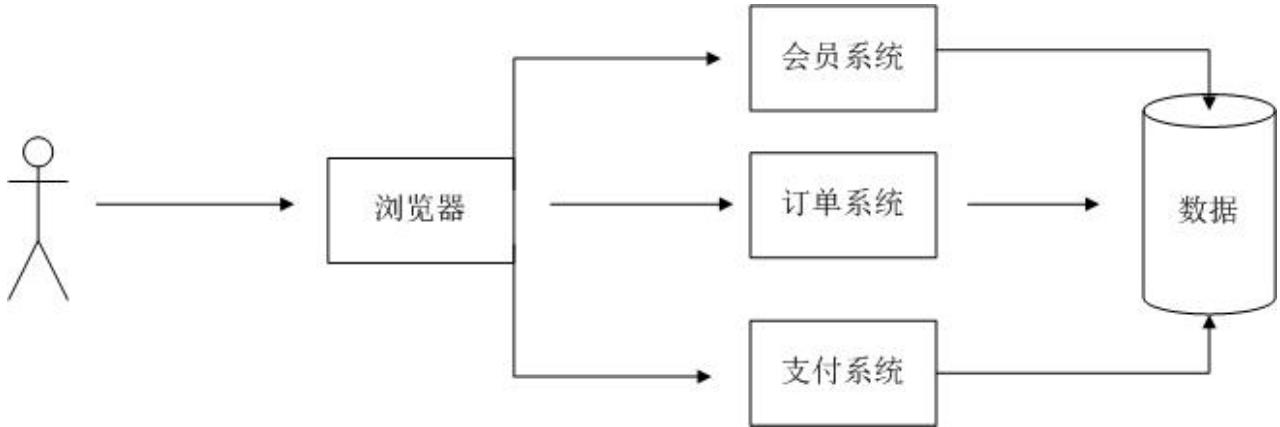
- 易于集中式开发、测试、管理和部署
- 无须考虑跨语言
- 能够避免功能重复开发

单体应用的缺点

- 团队合作困难
- 代码的维护、重构、部署都比较难
- 稳定性、可用性(停机维护)、扩展性不高
- 需要绑定某种特定的开发语言

1.1.2、垂直应用阶段

为了应对更高的并发、业务需求和解决单体应用的缺点，需要根据业务功能对单体应用的架构方法进行演进，如图：



- 拆分后的系统可以解决高并发、资源按需分配的问题
- 可以针对不同的模块进行优化和资源分配

拆分应有两种方式：水平拆分和垂直拆分

水平拆分

水平拆分是指根据业务来拆分应用。

例如：原应用中包含订单、会员两个部分，拆分后可以将其拆分成订单系统和会员系统。

其优点是：拆分后保证业务之间的相互影响较小，能合理地分配硬件资源(不同的业务对浏览和性能的要求是不一样的)。但是这样可能会导致整个系统存在"重复造轮子"的问题,而且难于维护

垂直拆分

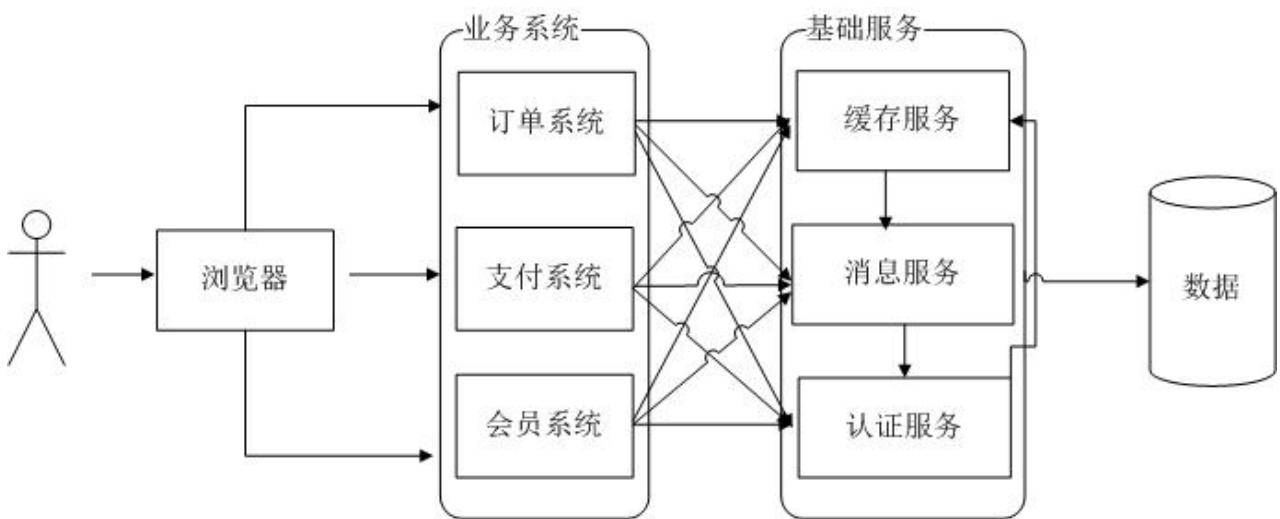
垂直拆分是根据调用方法对系统进行拆分。

例如：会员系统可以垂直拆分为普通用户和企业用户。

其优点是:能按需分配资源和流量,各个垂直调用之间互不影响;但是同样是"重复造轮子"

1.1.3、分布式系统阶段

在分布式系统中,各个小系统之间的交互是不可避免的,此时可将核心业务作为独立的服务抽取出来,逐渐形成稳定的基础服务,如图:

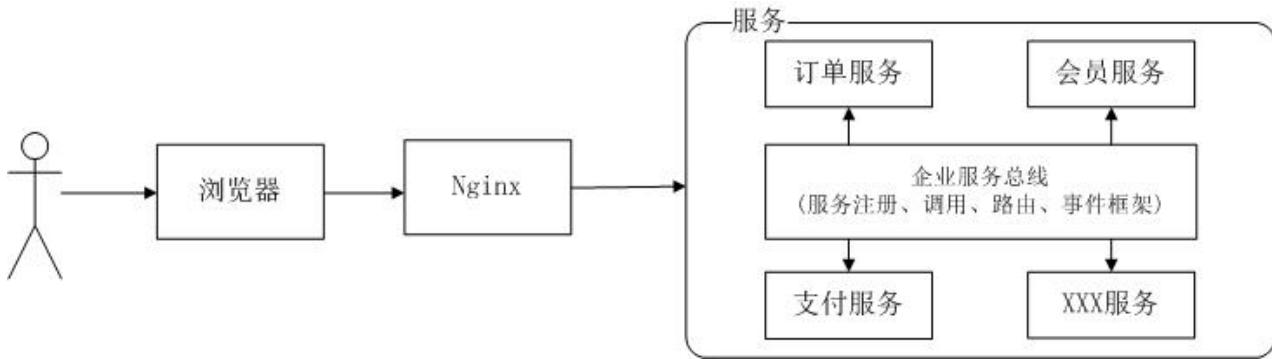


其优点是:对基础服务进行了抽取,服务之间可以相互调用,提高了代码的复用和开发效率

其缺点是:业务间耦合度变高;调用关系错综复杂;系统难以维护;在搭建集群后,很难实现负载均衡。

1.1.4、服务治理阶段

在服务治理 (SOA) 架构中,需要一个企业服务总线 (ESB) 将基于不同协议的服务节点连接起来,它的工作是转换、解释消息和路由。服务治理架构如图:



SOA解决了以下问题：

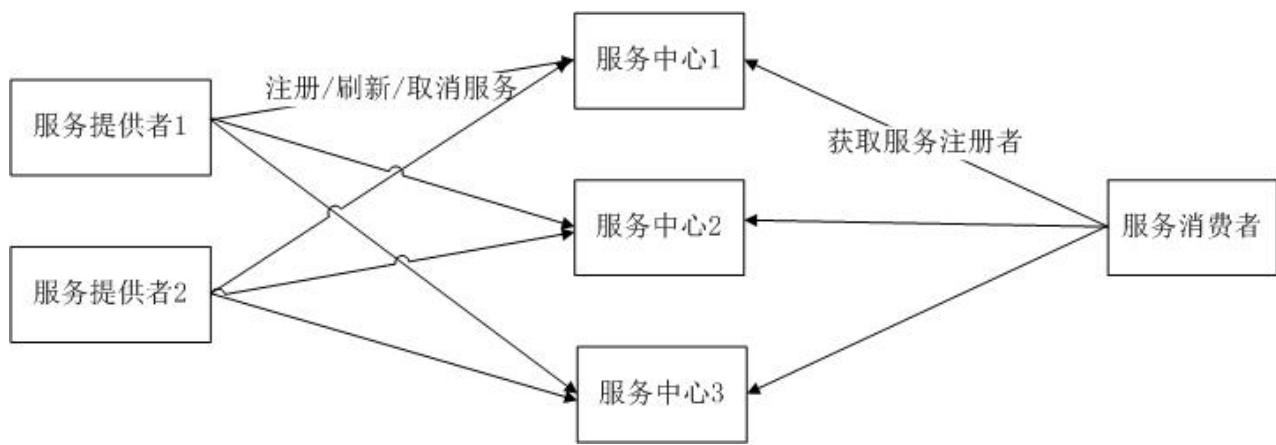
- 服务间的通信问题：引入了ESB、技术规范、服务管理规范，从而解决了不同服务间的通信问题
- 基础服务的梳理问题：以ESB为中心梳理和规整分布式服务
- 业务的服务化问题：以业务为驱动，将业务单元封装到服务中
- 服务的可复用性问题：将业务功能设计为通用的业务服务，以实现业务逻辑的复用

随着业务复杂性、需求多变性和用户规模的不断增长，敏捷开发和DevOps(一种过程、方法的统称，用于促进开发、运维和质量保证部门之间的沟通、合作和整合)变得特别重要。

1.1.5、微服务阶段

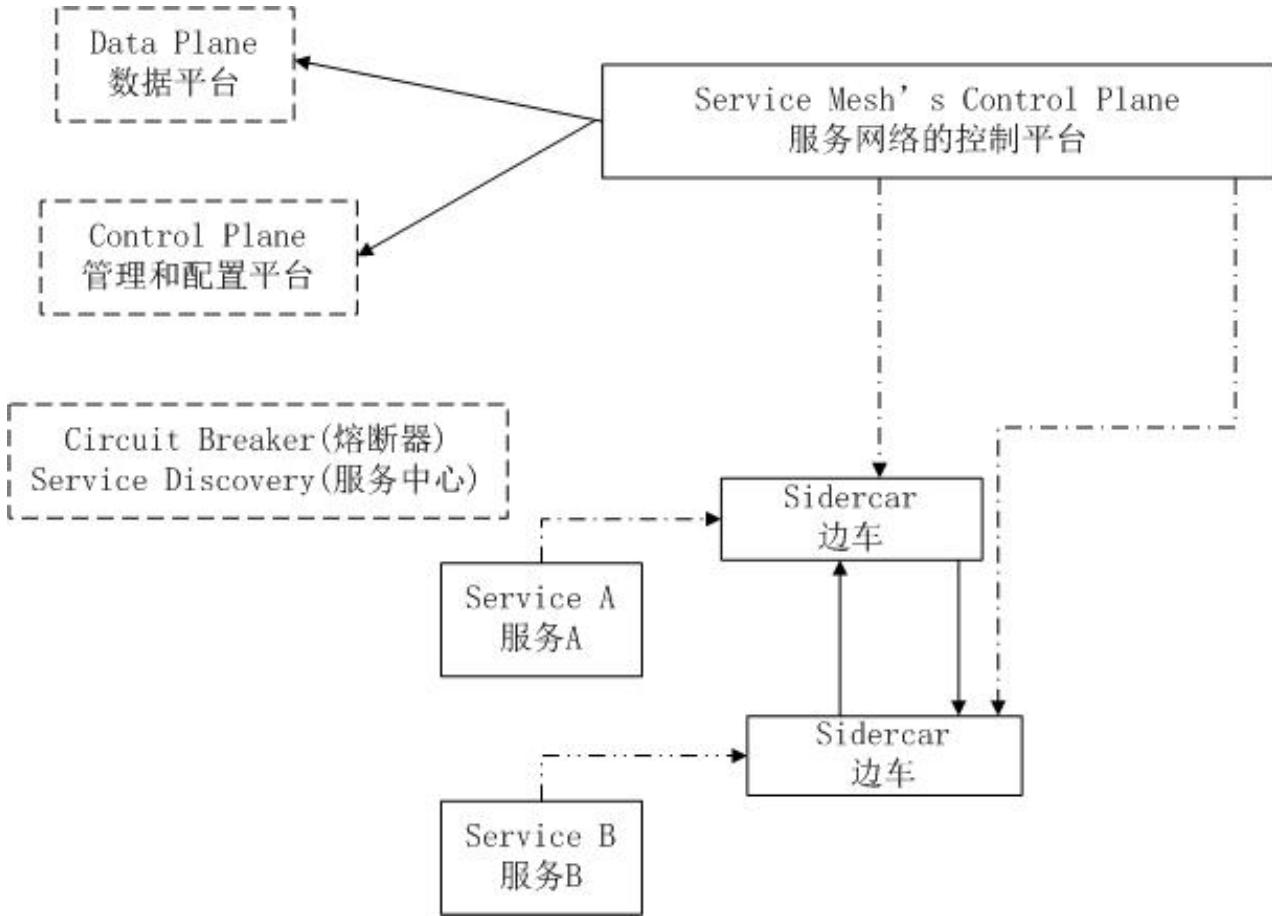
微服务(Microservices)架构是指：将系统的业务功能划分为极小的独立微服务,每个微服务只关注于完成某个小的任务。系统中的单个微服务可以被独立部署和扩展,且各个微服务之间高内聚、松耦合的。微服务之间采用轻量化通信机制暴露接口来实现通信。

微服务系统架构如图：



1.1.6、服务网络阶段

服务网格(Service Mesh)独立于服务之外运行,是服务间通信的基础设施层。服务网络类似于在每个服务上粘贴的功能模块。如图：



服务之间通过Sidercar(边车)进行通信,所有Sidercar和网络连接就形成了Service Mesh。

组件说明:

- SiderCar主要负责服务发现和容错处理
- 数据平台: 处理服务间通信, 并实现服务发现、负载均衡、流量管理、健康检查等
- 控制平台: 管理并配置Sidercar, 以执行策略和收集数据

1.2、系统架构发展背景

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，每个应用专用于做某一些方面的事情，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、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、微服务架构比较

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

Spring MVC、Spring、SpringBoot

Netflix的Eureka、Apache的ZooKeeper等

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

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；

2、微服务架构-SpringCloud

2.1、SpringCloud简介

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；

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）；

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。

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；

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；

SpringCloud官网：<http://spring.io>

2.2、SpringCloud架构发展

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H版本；Hoxton.RC2版本，对应SpringBoot 2.2.x

Spring Cloud与Spring Boot版本匹配关系

Release Train	Boot Version
2020.0.x aka Ilford	2.4.x
Hoxton	2.2.x, 2.3.x (Starting with SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.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、SpringCloud环境需求

SpringBoot 2.2.x

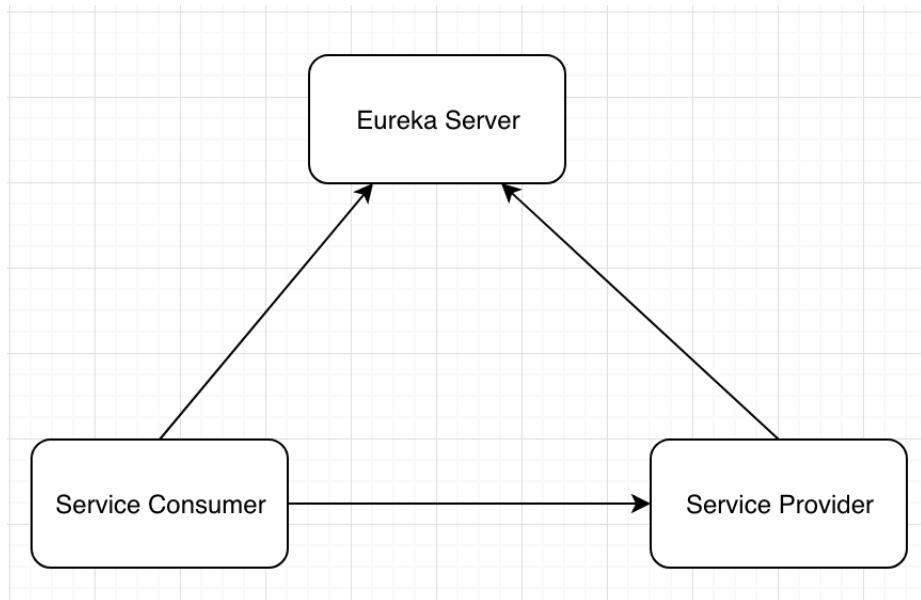
Spring Cloud Hoxton xxx

Maven 3.5.3

JDK 1.8.152

IntelliJ IDEA

2.4、SpringCloud架构体系



Eureka Server： 服务注册中心和服务发现中心

Service Provider： 服务提供者

Service Consumer: 服务消费者

3、SpringCloud快速开发

3.1、搭建和配置-SpringCloud架构

SpringCloud构建微服务是基于SpringBoot开发的。

创建**SpringBoot**工程

springcloud_eg1

配置**springboot**依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.3.7.RELEASE</version>
</parent>
```

3.2、搭建和配置-服务提供者

创建一个**Module**

01-service-provider

配置web程序开发依赖

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

编写启动类

ProviderApplication.java

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.Spr
ingBootApplication;

@SpringBootApplication
public class ProviderApplication {
    public static void main(String[ ] args)
{
    SpringApplication.run(ProviderApplication.
class);
}
}
```

编写控制器

HelloController.java

```
package com.ww.controller;

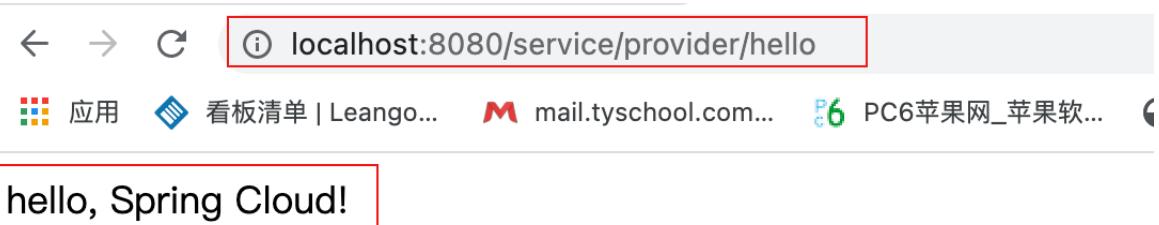
import
org.springframework.web.bind.annotation.RequestMapping;
import
org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @RequestMapping("/service/provider/hello")
    public String hello(){
        return "hello, Spring Cloud!";
    }
}
```

测试服务提供者接口

<http://localhost:8080/service/provider/hello>



3.3、搭建和配置-消费提供者

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；

创建一个Module

01-service-consumer

配置web程序开发依赖

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

修改端口号

8080被服务提供者占用，所有要调整服务消费者服务的端口号

application.yml

```
server:  
  port: 8081
```

编写启动类

ConsumerApplication.java

```
package com.ww;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class ConsumerApplication {  
    public static void main(String[] args)  
    {  
  
        SpringApplication.run(ConsumerApplication.  
class);  
    }  
}
```

编写控制器

WebController.java

```
package com.ww.controller;

import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class WebController {

    @RequestMapping("/service/consumer/hello")
    public String hello(){
        //如何才能从服务消费者调用服务提供者?
        RestTemplate类需要配置
        return null;
    }
}
```

配置RestTemplate类

BeanConfig.java

```
package com.ww.config;
```

```
import
org.springframework.context.annotation.Bean
;
import
org.springframework.context.annotation.Configuration;
import
org.springframework.web.client.RestTemplate
;

@Configuration //相当于spring中的
applicationContext.xml配置文件
public class BeanConfig {
    /*
     * 配置一个bean,相当于<bean id=""
class="" />
     */
    @Bean
    public RestTemplate restTemplate(){
        return new RestTemplate();
    }
}
```

完善WebController.java

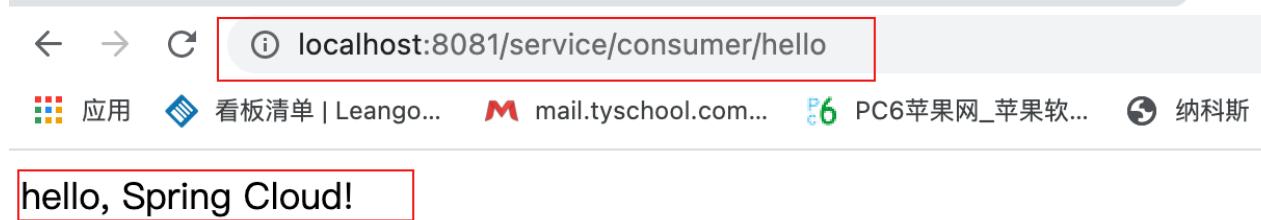
```
package com.ww.controller;
```

```
import  
org.springframework.beans.factory.annotation.Autowired;  
  
import  
org.springframework.context.annotation.Bean;  
  
import  
org.springframework.web.bind.annotation.RequestMapping;  
  
import  
org.springframework.web.bind.annotation.RestController;  
  
import  
org.springframework.web.client.RestTemplate;  
  
@RestController  
public class WebController {  
    @Autowired  
    private RestTemplate restTemplate;  
  
    @RequestMapping("/service/consumer/hello")  
    public String hello(){
```

```
//如何才能从服务消费者调用服务提供者中的地址? RestTemplate  
    return  
restTemplate.getForEntity("http://localhost  
:8080/service/provider/hello",String.class)  
.getBody();  
}  
}
```

测试服务消费者对服务提供者调用

<http://localhost:8081/service/consumer/hello>



4、注册中心-Eureka

4.1、注册中心概述

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。

4.1.1、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；

4.1.2、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；

4.1.3、Eureka

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。

4.1.4、Eureka与zookeeper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。

4.2、搭建注册中心-Eureka

创建一个Module

02-eureka-service

配置eureka

spring-cloud-starter-netflix-eureka-server

```
<dependency>

    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

编写Eureka服务端配置

application.yml

```
server:  
  port: 8000 #端口号  
eureka:  
  instance:  
    hostname: localhost #eureka服务器主机名  
  client:  
    service-url:  
      defaultZone:  
        http://${eureka.instance.hostname}:${server.port}/eureka/ #指定服务器中心注册位置  
        register-with-eureka: false #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  
        fetch-registry: false #服务器中心不需要去检查其他服务
```

编写启动类

EurekaApplication.java

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.netflix.eureka.se
rver.EnableEurekaServer;

@SpringBootApplication
@EnableEurekaServer //开启Eureka注册中心服务端
public class EurekaApplication {
    public static void main(String[ ] args)
{
    SpringApplication.run(EurekaApplication.cl
ass);
}
}
```

启动Eureka注册中心

<http://localhost:8000/>

HOME

The screenshot shows the Spring Eureka dashboard with the following sections:

- System Status**: Shows environment (test), data center (default), and various metrics like current time (2020-12-23T16:07:29 +0800), uptime (00:00), lease expiration enabled (false), renew threshold (1), and renew count (0).
- DS Replicas**: Instances currently registered with Eureka. A table shows Application (test), AMIs (No instances available), Availability Zones (Status).
- General Info**: A table of system properties:

Name	Value
total-avail-memory	308mb
environment	test
num-of-cpus	8
current-memory-usage	119mb (38%)
server-upptime	00:00
registered-replicas	
unavailable-replicas	
available-replicas	
- Instance Info**: A table showing instance details:

Name	Value
ipAddr	192.168.0.105
status	UP

LAST 1000 SINCE STARTUP

The screenshot shows the Spring Eureka dashboard with the following sections:

- System Status**: Same as the top dashboard.
- DS Replicas**: Shows last 1000 newly registered leases. A table includes columns for Timestamp and Lease. A note says "No results available".

4.3、注册服务

前面我们搭建了服务提供者项目(01-service-provider)，接下来我们就可以将该服务提供者注册到Eureke注册中心

4.3.1、添加依赖

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；

spring-cloud-starter-netflix-eureka-client

```
<dependency>

<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
<version>2.2.6.RELEASE</version>
</dependency>
```

4.3.2、配置服务

application.yml

```
server:
  port: 8080 #指定端口号
spring:
  application:
    name: service-provider #配置服务名字，这外
                           #名字将在服务消费者使用时被调用
  eureka:
    client:
      service-url:
        defaultZone:
          http://localhost:8000/eureka #指定eureka访问
                                      #地址
```

4.3.3、激活服务

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

```
//.....
@SpringBootApplication
@EnableEurekaClient
public class ProviderApplication {
//.....
}
```

4.3.4、控制器提供服务

自己编写的服务提供

```
//.....  
 @RestController  
 public class HelloController {  
     //.....  
  
     @RequestMapping("/service/provider/test")  
     public String test(){  
         return "使用了Eureka注册中心的服务提供者!";  
     }  
     //.....  
 }
```

4.3.5、查看注册

先启动Eureka服务器，再启动服务提供者

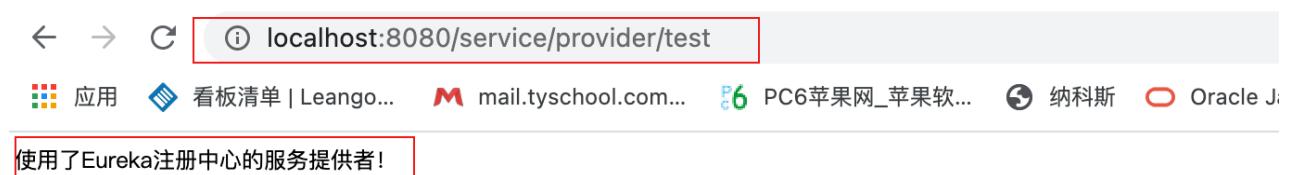
<http://localhost:8000/>



The screenshot shows the Eureka UI interface. At the top, there's a header with the text 'DS Replicas' and 'Instances currently registered with Eureka'. Below this is a table with the following data:

Application	AMIs	Availability Zones	Status
SERVICE-PROVIDER	n/a (1)	(1)	UP (1) - 192.168.0.105:service-provider:8080

<http://localhost:8080/service/provider/test>



The screenshot shows a browser window with the URL 'localhost:8080/service/provider/test' in the address bar. The page content is a single line of text: '使用了Eureka注册中心的服务提供者!' (Used the Eureka registration center service provider!).

4.4、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。

4.4.1、添加依赖

在该消费者项目（01-service-consumer）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；

```
<dependency>

    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

4.4.2、配置服务

application.yml

```
server:  
  port: 8081  
spring:  
  application:  
    name: service-consumer  
eureka:  
  client:  
    service-url:  
      defaultZone:  
        http://localhost:8000/eureka
```

4.4.3、激活服务

```
//.....  
@SpringBootApplication  
@EnableEurekaClient  
public class ConsumerApplication {  
//.....  
}
```

4.4.4、Ribbon对服务提供者调用

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。

配置RestTemplate

BeanConfig.java

```
//.....  
@Configuration //相当于spring中的  
applicationContext.xml配置文件  
public class BeanConfig {  
    //.....  
    @LoadBalanced //使用Ribbon的负载均衡从注册  
    中心中获取服务  
    @Bean  
    //.....  
}
```

调用服务提供者

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

```

//.....
@RestController
public class WebController {
    @Autowired
    private RestTemplate restTemplate;
    //.....

    @RequestMapping("/service/consumer/test")
    public String test(){
        return
restTemplate.getEntity("http://service-
provider/service/provider/test",String.cla
ss).getBody()+"服务消费者";
    }
    //.....
}

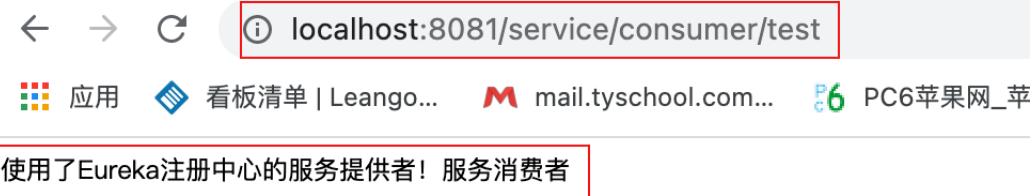
```

4.4.5、查看注册

<http://localhost:8000/>

DS Replicas			
Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
SERVICE-CONSUMER	n/a (1)	(1)	UP (1) - 192.168.0.105:service-consumer:8081
SERVICE-PROVIDER	n/a (1)	(1)	UP (1) - 192.168.0.105:service-provider:8080

<http://localhost:8081/service/consumer/test>



4.6、Eureka高可用集群

4.6.1、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。

4.6.2、高可用集群搭建

创建Eureka服务端项目

eureka8001、eureka8002

修改application.yml文件

application-eureka8001.yml

```
server:
  port: 8001
eureka:
  instance:
    hostname: eureka8001
  client:
    service-url:
      defaultZone:
        http://eureka8002:8002/eureka/
        register-with-eureka: false
        fetch-registry: false
```

application-eureka8002.yml

```
server:  
  port: 8002  
eureka:  
  instance:  
    hostname: eureka8002  
  client:  
    service-url:  
      defaultZone:  
        http://eureka8001:8001/eureka/  
        register-with-eureka: false  
        fetch-registry: false
```

修改hosts文件配置

windows

C:\Windows\System32\drivers\etc\hosts

mac

/etc/hosts ->在root用户下修改

```
127.0.0.1 eureka8001  
127.0.0.1 eureka8002
```

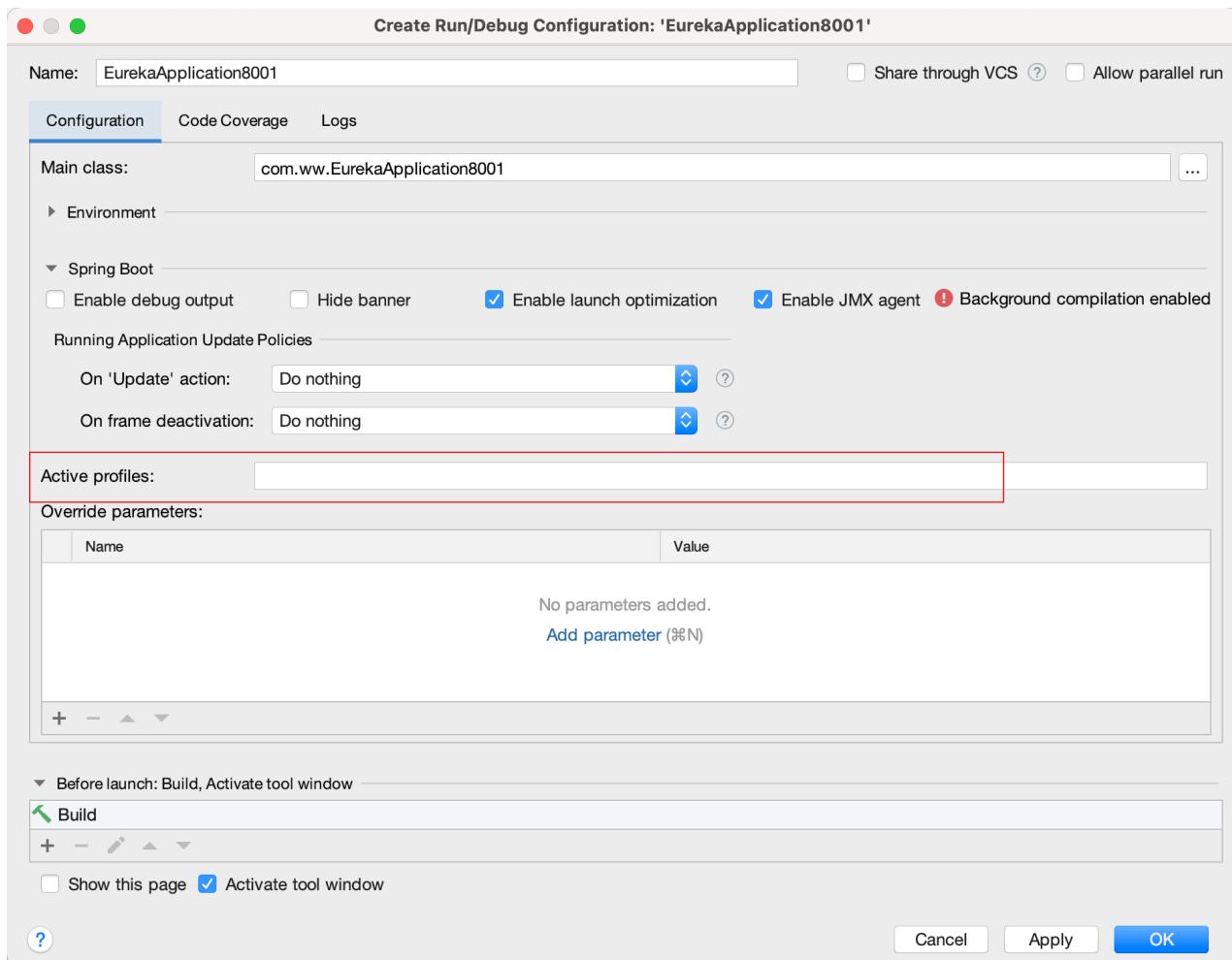
复制启动文件

EurekaApplication8001.java

EurekaApplication8002.java

创建运行项

右击启动文件---->"create EurekaApplication8001..."...



Active profiles:eureka8001 #设置配置文件,
appliation-eureka8001.yml, 省略application

启动服务

<http://eureka8001:8001/>

System Status

Environment	test
Data center	default

Current time	2020-12-23T21:36:45 +0800
Uptime	00:00
Lease expiration enabled	false
Renews threshold	1
Renews (last min)	0

DS Replicas

eureka8002

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
No instances available			

<http://eureka8002:8002/>**System Status**

Environment	test
Data center	default

Current time	2020-12-23T21:36:41 +0800
Uptime	00:03
Lease expiration enabled	false
Renews threshold	1

DS Replicas

eureka8001

4.6.3、高可用集群测试

配置服务提供者和服务消费者

服务提供者

```
server:
  port: 8080
spring:
  application:
    name: service-provider
eureka:
  client:
    service-url:
      #defaultZone:
      http://localhost:8000/eureka
      defaultZone:
      http://eureka8001:8001/eureka,http://eureka
      8002:8002/eureka
```

服务消费者

```

server:
  port: 8081
spring:
  application:
    name: service-consumer
eureka:
  client:
    service-url:
      defaultZone:
        http://eureka8001:8001/eureka,http://eureka
        8002:8002/eureka

```

启动服务

<http://eureka8001:8001/>

The screenshot shows the Spring Eureka dashboard with the following details:

- System Status:**

Environment	test	Current time	2020-12-23T21:59:25 +0800
Data center	default	Uptime	00:02
		Lease expiration enabled	false
		Renews threshold	5
		Renews (last min)	4
- DS Replicas:** eureka8002
- Instances currently registered with Eureka:**

Application	AMIs	Availability Zones	Status
SERVICE-CONSUMER	n/a (1)	(1)	UP (1) - 192.168.0.105:service-consumer:8081
SERVICE-PROVIDER	n/a (1)	(1)	UP (1) - 192.168.0.105:service-provider:8080

<http://eureka8002:8002/>

The screenshot shows the Spring Eureka dashboard. At the top, there's a header with the Spring logo and 'Eureka'. On the right, it says 'HOME' and 'LAST 1000 SINCE STARTUP'. Below the header, there are two main sections: 'System Status' and 'DS Replicas'. The 'System Status' section contains tables for Environment (Environment: test, Data center: default), Current time (2020-12-23T21:59:31 +0800), Uptime (00:01), Lease expiration enabled (false), Renews threshold (5), and Renews (last min) (1). The 'DS Replicas' section shows a single instance named 'eureka8001'. Under 'Instances currently registered with Eureka', there's a table with columns Application, AMIs, Availability Zones, and Status. It lists 'SERVICE-CONSUMER' with n/a (1) AMIs, (1) AZs, and UP (1) - 192.168.0.105:service-consumer:8081 status. It also lists 'SERVICE-PROVIDER' with n/a (1) AMIs, (1) AZs, and UP (1) - 192.168.0.105:service-provider:8080 status.

4.6.4、Eureka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.

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。

案例分析：

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。

自我保护配置

Eureka服务端

```
eureka.server.enable-self-preservation =  
false #关闭自我保护
```

作用是如果开启了自我保护模式以后，那么如果服务的提供者或消费者（Eureka的客户端）因为网络波动问题暂时失去了与服务器端的连接那么Eureka就会直接注销这个服务删除这个服务相关的数据，如果关闭了这个自我保护，Eureka只会先挂起这个服务，当网络恢复正常

以后这个服务将自动的恢复。

Eureka客户端

```
eureka.instance.lease-renewal-interval-in-seconds=2 #每间隔2s，向服务端发送一次心跳，证明自己依然"存活"，默认30秒
```

```
eureka.instance.lease-expiration-duration-in-seconds=10 #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，默认90秒
```

5、客户端负载均衡-Ribbon

5.1、SpringCloud-Ribbon概述

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。

5.1.1、负载均衡分类

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。

5.1.2、Ribbon概述

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；

5.1.3、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，只需要如下几步：

5.2.1、创建多个服务提供者

创建新moudle

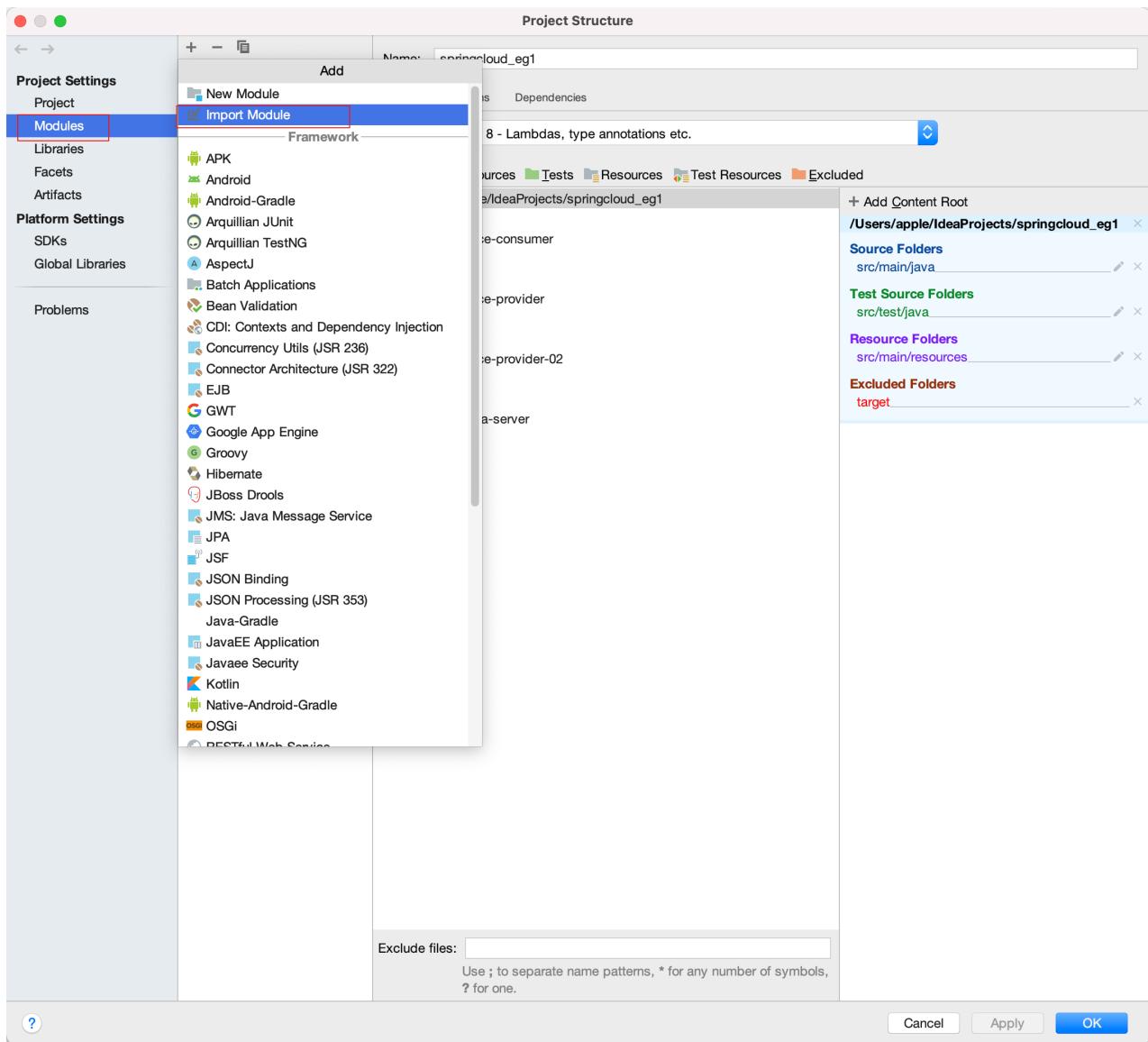
复制01-service-provider项目为01-service-provider-02

修改新moudle中的application.yml

```
server:
  port: 8082 #端口要改
spring:
  application:
    name: service-provider #服务提供者名字不用修改
  eureka:
    client:
      service-url:
        #defaultZone:
        http://localhost:8000/eureka
        defaultZone:
        http://eureka8001:8001/eureka,http://eureka
        8002:8002/eureka
    instance:
      lease-expiration-duration-in-seconds:
        60
      lease-renewal-interval-in-seconds: 2
```

导入moudle

右击项目，选择"Open moudle Setting"



修改服务提供者1和2的HelloController.java

```
public String test(){
    System.out.println("这是服务提供者
1.....");
    return "使用了Eureka注册中心的服务提供
者! provider 1";
}
```

```
public String test(){
    System.out.println("这是服务提供者
2.....");
    return "使用了Eureka注册中心的服务提供
者! provider 2";
}
```

启动服务

<http://eureka8001:8001/> 或 <http://eureka8002:8002/>

The screenshot shows the Eureka UI interface. At the top, there is a warning message: "THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS." Below this, there is a section titled "DS Replicas" with a single entry: "eureka8002". Underneath, there is a table titled "Instances currently registered with Eureka". The table has four columns: Application, AMIs, Availability Zones, and Status. There is one row for the application "SERVICE-PROVIDER", which has "n/a (2)" AMIs, "(2)" Availability Zones, and "UP (2)" status. The IP addresses listed are "192.168.0.105:service-provider:8082" and "192.168.0.105:service-provider:8080".

Application	AMIs	Availability Zones	Status
SERVICE-PROVIDER	n/a (2)	(2)	UP (2) 192.168.0.105:service-provider:8082 192.168.0.105:service-provider:8080

5.2.2、服务消费者

注意：

服务消费者通过被@LoadBalanced注解修饰过的RestTemplate来调用服务提供者。

启动服务

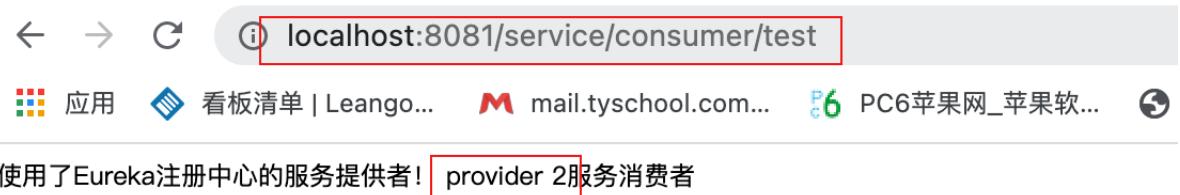
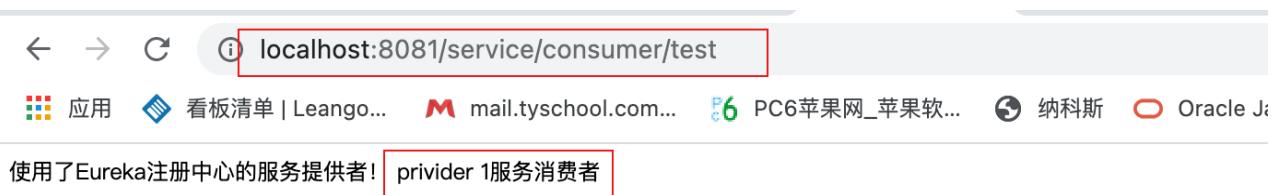
先测试服务提供者

<http://localhost:8080/service/provider/test>

<http://localhost:8082/service/provider/test>

在测试服务消费者

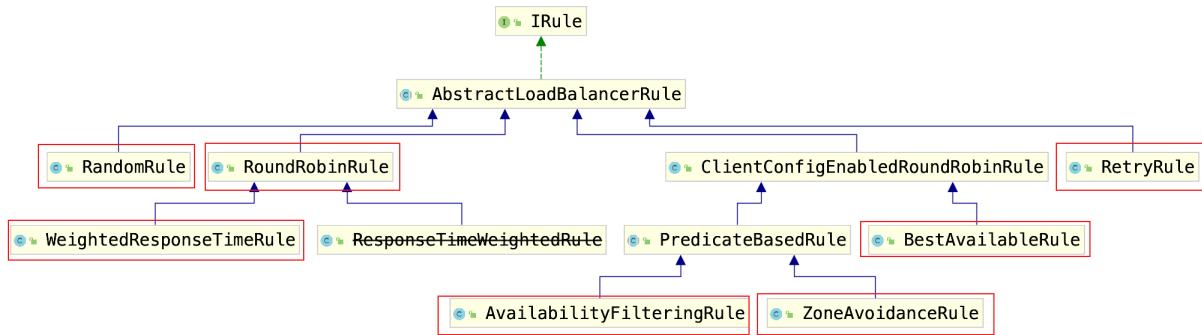
<http://localhost:8081/service/consumer/test>



5.3、负载均衡策略

5.3.1、策略概述

Ribbon的负载均衡策略是由IRule接口定义



类	负载策略
RandomRule	随机
RoundRobinRule	轮询（默认策略）
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；

5.3.2、使用随机策略

修改配置

服务消费者中的BeanConfig.java

```
//.....  
@Configuration //相当于spring中的  
applicationContext.xml配置文件  
public class BeanConfig {  
//.....  
/*  
 * 覆盖掉Ribbon原有的负载均衡策略  
 */  
@Bean  
public IRule iRule(){  
    return new RandomRule(); //使用随机策  
略  
}  
//.....  
}
```

启动服务

重启服务提供者与服务消费者服务

<http://localhost:8081/service/consumer/test>

5.3.3、使用重试策略

修改配置

服务消费者中的BeanConfig.java

```
//.....  
@Configuration //相当于spring中的  
applicationContext.xml配置文件  
public class BeanConfig {  
//.....  
/*  
 * 覆盖掉Ribbon原有的负载均衡策略  
 * */  
@Bean  
public IRule iRule(){  
    //return new RandomRule(); //使用随机  
策略  
    return new RetryRule(); //使用重试策略  
}  
//.....  
}
```

启动服务

重启服务提供者与服务消费者服务

<http://localhost:8081/service/consumer/test>

5.4、Rest请求模板类

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。

Rest常用请求：

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

5.4.1、GET

A、getForEntity()方法

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

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；

返回Http请求响应

(1) WebController.java添加方法

```
//.....
@RequestMapping("/service/consumer/test1")
public String test1(){
    ResponseEntity<String>
    responseEntity=restTemplate.getForEntity("h
    ttp://service-
    provider/service/provider/test",String.cla
    ss);
    int
    statusCodeValue=responseEntity.getStatusCode();
    /状态码
    HttpHeaders
    httpHeaders=responseEntity.getHeaders(); // 
    消息头： 请求头和响应头
```

```
String  
body=responseEntity.getBody(); //返回内容  
  
System.out.println(statusCodeValue);  
System.out.println(httpStatus);  
System.out.println(httpHeaders);  
System.out.println(body);  
  
return  
restTemplate.getForEntity("http://service-  
provider/service/provider/test",String.class)  
.getBody();  
}  
//.....
```

(2) 启动服务

<http://localhost:8081/service/consumer/test1>

#运行结果

200

200 OK

[Content-Type: "text/plain; charset=UTF-8",
Content-Length: "59", Date: "Thu, 24 Dec 2020
09:08:33 GMT", Keep-Alive: "timeout=60",
Connection: "keep-alive"]

使用了Eureka注册中心的服务提供者! provider 2

返回对象

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 Manager对象；

(1) 服务提供者，创建Manager.java

Manager.java

```
package com.ww.pojo;

public class Manager {
    private Integer id;
    private String username;
    private String password;
```

```
public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getUsername() {
    return username;
}

public void setUsername(String
username) {
    this.username = username;
}

public String getPassword() {
    return password;
}

public void setPassword(String
password) {
    this.password = password;
}
```

(2) 服务提供者，在HelloController.java添加manager方法

```
//.....  
 @RequestMapping( "/service/provider/manager"  
 )  
 public Manager manager(){  
     Manager manager = new Manager();  
     manager.setId(12);  
     manager.setUsername("zhangsan");  
     manager.setPassword("lisi");  
     return manager;  
 }  
//.....
```

(3) 服务提供者2，和前面操作 (1) (2) 一样

(4) 服务消费者，加入Manager.java

复制Manager.java到服务提供者中com.ww.pojo

(5) 服务消费者，在WebController.java中添加manager方法

```
//.....  
 @RequestMapping( "/service/consumer/manager"  
 )  
 public String manager(){
```

```
    ResponseEntity<Manager>
responseEntity=restTemplate.getForEntity( "h
ttp://service-
provider/service/provider/manager" , Manager.
class);

        int
statusCodeValue=responseEntity.getStatusCodeValue( );
        HttpStatus
httpStatus=responseEntity.getStatusCode();
        HttpHeaders
httpHeaders=responseEntity.getHeaders();
        Manager
manager=responseEntity.getBody();

System.out.println(statusCodeValue);
System.out.println(httpStatus);
System.out.println(httpHeaders);

System.out.println(manager.getId() + ", " + manager.getUsername() + ", " + manager.getPassword());
});
```

```
        return  
restTemplate.getForEntity("http://service-  
provider/service/provider/manager",String.c  
lass).getBody();  
}  
//.....
```

(6) 启动服务

<http://localhost:8081/service/consumer/manager>

```
#运行结果  
200  
200 OK  
[Content-Type: "application/json", Transfer-  
Encoding: "chunked", Date: "Thu, 24 Dec 2020  
09:58:51 GMT", Keep-Alive: "timeout=60",  
Connection: "keep-alive"]  
12,zhangsan,lisi
```

重载方法，带参-数组

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

(1) 服务提供者， HelloController.java中添加 setmanager方法

```
//.....  
 @RequestMapping( "/service/provider/setmanager")  
 public Manager  
 setManager(@RequestParam("id") Integer id,  
  
 @RequestParam("username") String username,  
  
 @RequestParam("password") String password)  
{  
     Manager manager=new Manager();  
     manager.setId(id);  
     manager.setUsername(username);  
     manager.setPassword(password);  
     return manager;  
}  
//.....
```

(2) 服务提供者2， 和前面操作 (1) 一样

(3) 服务消费者，在WebController.java中添加 setManager方法

```
//.....
```

```
@RequestMapping("/service/consumer/setmanager")
public Manager setManager(){
    String strArray[ ]={"12","李
四","1323lisi"};
    ResponseEntity<Manager>
responseEntity=restTemplate.getForEntity("h
ttp://service-
provider/service/provider/setmanager?id=
{0}&username={1}&password={2}",
    Manager.class,strArray);

    int
statusCodeValue=responseEntity.getStatusCode();
    HttpStatus
httpStatus=responseEntity.getStatusCode();
    HttpHeaders
httpHeaders=responseEntity.getHeaders();
    Manager
manager=responseEntity.getBody();

    System.out.println(statusCodeValue);
    System.out.println(httpStatus);
    System.out.println(httpHeaders);
```

```
        System.out.println(manager.getId() + ", " + manager.getUsername() + ", " + manager.getPassword());
    }

    return
restTemplate.getForEntity("http://service-
provider/service/provider/setmanager?id=
{0}&username={1}&password={2}" ,
Manager.class,strArray).getBody();
}

//.....
```

(4) 启动服务

<http://localhost:8081/service/consumer/setmanager>

```
#运行结果
200
200 OK
[Content-Type:"application/json", Transfer-
Encoding:"chunked", Date:"Thu, 24 Dec 2020
10:26:06 GMT", Keep-Alive:"timeout=60",
Connection:"keep-alive"]
12,李四,1323lisi
```

重载方法，带参-Map

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

(1) 服务消费者，将数组参数改为Map参数

修改setManager方法

```
@RequestMapping("/service/consumer/setmanager")
public Manager setManager(){
    //String strArray[ ]={"12","李
四","1323lisi"};
    Map<String, Object> map=new
HashMap();
    map.put("id", 123);
    map.put("username", "张三");
    map.put("password", "zhangsan");
```

```
// ResponseEntity<Manager>
responseEntity=restTemplate.getForEntity("h
ttp://service-
provider/service/provider/setmanager?id=
{0}&username={1}&password={2}" ,
// Manager.class,strArray);
ResponseEntity<Manager>
responseEntity=restTemplate.getForEntity("h
ttp://service-
provider/service/provider/setmanager?id=
{id}&username={username}&password=
{password}",Manager.class,map); //注意传参时没
有0, 1, 2, 只能用key
int
statusCodeValue=responseEntity.getStatusCode();
HttpStatus
httpStatus=responseEntity.getStatusCode();
HttpHeaders
httpHeaders=responseEntity.getHeaders();
Manager
manager=responseEntity.getBody();

System.out.println(statusCodeValue);
System.out.println(httpStatus);
```

```
        System.out.println(httpHeaders);

        System.out.println(manager.getId() + ", " + manager.getUsername() + ", " + manager.getPassword());
    });

    //return
    restTemplate.getForEntity("http://service-provider/service/provider/setmanager?id={0}&username={1}&password={2}" ,
        //
        Manager.class,strArray).getBody();
    return
    restTemplate.getForEntity("http://service-provider/service/provider/setmanager?id={id}&username={username}&password={password}" ,Manager.class, map).getBody();
}
```

(4) 启动服务

<http://localhost:8081/service/consumer/setmanager>

#运行结果

200

200 OK

```
[Content-Type: "application/json", Transfer-Encoding: "chunked", Date: "Thu, 24 Dec 2020 10:43:11 GMT", Keep-Alive: "timeout=60", Connection: "keep-alive"]
```

123,张三,zhangsan

B、**getForObject()**方法

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;
```

(1) 服务消费者，加入一个新的getManager方法

```
//.....  
 @RequestMapping( "/service/consumer/getmanager" )  
     public Manager getManger() {  
         Map<String, Object> map = new  
         HashMap();  
         map.put("id", 1234);  
         map.put("username", "zhangsan");  
         map.put("password", "zhangsan1234");  
  
         Manager manager = restTemplate.getForObject("http://s  
ervice-  
provider/service/provider/setmanager?id=  
{id}&username={username}&password=  
{password}",  
         Manager.class, map);  
         return manager;  
     }  
 //.....
```

自己可以去试一下数组

(2) 启动服务

<http://localhost:8081/service/consumer/getmanager>

5.4.2、POST

Post与Get请求非常类似：

postForEntity()方法

```
public <T> ResponseEntity<T>
postForEntity(String url, @Nullable Object
request, Class<T> responseType, Object...
uriVariables) throws RestClientException {}

public <T> ResponseEntity<T>
postForEntity(String url, @Nullable Object
request, Class<T> responseType, Map<String,
?> uriVariables) throws RestClientException
{}

public <T> ResponseEntity<T>
postForEntity(URI url, @Nullable Object
request, Class<T> responseType) throws
RestClientException {}
```

postForObject()方法

```
public <T> T postForObject(String url,
@Nullable Object request, Class<T>
responseType, Object... uriVariables)
throws RestClientException {}  
  
public <T> T postForObject(String url,
@Nullable Object request, Class<T>
responseType, Map<String, ?> uriVariables)
throws RestClientException {}  
  
public <T> T postForObject(URI url,
@Nullable Object request, Class<T>
responseType) throws RestClientException {}
```

postForLocation()方法

```
public URI postForLocation(String url,
@Nullable Object request, Object...
uriVariables) throws RestClientException {}

public URI postForLocation(String url,
@Nullable Object request, Map<String, ?>
uriVariables) throws RestClientException {}

public URI postForLocation(URI url,
@Nullable Object request) throws
RestClientException {}
```

(1) 在服务提供者，HelloController.java中加入addManager方法

```
//.....  
//@RequestMapping(value =  
"/service/provider/addmanager",method =  
RequestMethod.POST)  
@PostMapping("/service/provider/addmanager"  
)  
public Manager  
addManager(@RequestParam("id") Integer id,  
  
@RequestParam("username") String username,  
  
@RequestParam("password") String password)  
{  
    Manager manager=new Manager();  
    manager.setId(id);  
    manager.setUsername(username);  
    manager.setPassword(password);  
    return manager;  
}  
//.....
```

(2) 服务提供者2，HelloController.java方法中复制addManager方法

(3) 消费提供者，WebController.java中加入addManager方法

```
//.....  
 @RequestMapping( "/service/consumer/addmanager" )  
 public Manager addManager() {  
     String strArray[ ]={ "12" , "李  
四" , "1323lisi" };  
     //要传表单信息，不能使用Map对象，要用  
MultiValueMap  
     //Map<String, Object> map=new  
HashMap<String ,Object>();  
     MultiValueMap<String, Object> map =  
new LinkedMultiValueMap<String, Object>();  
     map.add( "id" , "1234" );  
     map.add( "username" , "zhangsan" );  
     map.add( "password" , "zhangsan1234" );  
  
     ResponseEntity  
responseEntity=restTemplate.postForEntity( "  
http://service-  
provider/service/provider/addmanager" ,map, M  
anager.class);  
     System.out.println( "-----  
postForEntity-----" );  
     int  
statusCodeValue=responseEntity.getStatusCode();
```

```
HttpStatus  
httpStatus=responseEntity.getStatusCode();  
  
HttpHeaders  
httpHeaders=responseEntity.getHeaders();  
  
Manager manager=(Manager)  
responseEntity.getBody();  
  
  
  
System.out.println(statusCodeValue);  
System.out.println(httpStatus);  
System.out.println(httpHeaders);  
  
  
  
System.out.println(manager.getId() + ", " + manager.getUsername() + ", " + manager.getPassword());  
  
  
  
Manager  
manager1=restTemplate.postForObject("http://  
/service-  
provider/service/provider/addmanager",map, M  
anager.class);  
  
System.out.println("-----  
postForObject-----");
```

```
System.out.println(manager1.getId() + ", " + manager1.getUsername() + ", " + manager1.getPassword());  
  
        return manager1;  
    }  
//.....
```

(4) 启动服务

<http://localhost:8081/service/consumer/addmanager>

5.4.3、PUT

前面我们提到了put是用于修改数据。

put()方法

```
public void put(String url, @Nullable  
Object request, Object... uriVariables)  
throws RestClientException {}  
  
public void put(String url, @Nullable  
Object request, Map<String, ?>  
uriVariables) throws RestClientException {}  
  
public void put(URI url, @Nullable Object  
request) throws RestClientException {}
```

(1) 在服务提供者，HelloController.java中加入updateManager方法

```
@PutMapping("/service/provider/updatemanager")
public Manager
updateManager(@RequestParam("id") Integer
id,
@RequestParam("username") String username,
@RequestParam("password") String password)
{
    Manager manager=new Manager();
    manager.setId(id);
    manager.setUsername(username);
    manager.setPassword(password);
    System.out.println(id+"----"
"+username+"----"+password);
    return manager;
}
```

(2) 服务提供者2， HelloController.java方法中复制
updateManager方法

(3) 消费提供者， WebController.java中加入
updateManager方法

```
//.....  
 @RequestMapping( "/service/consumer/updatemanager" )  
 public String updateManager(){  
     MultiValueMap<String, Object> map =  
 new LinkedMultiValueMap<String, Object>();  
     map.add( "id" , "1234" );  
     map.add( "username" , "zhangsan" );  
     map.add( "password" , "zhangsan1234" );  
     restTemplate.put( "http://service-provider/service/provider/updatemanager" ,ma  
p );  
     return "执行成功! " ;  
 }  
 //.....
```

(4) 启动服务

<http://localhost:8081/service/consumer/updatemanager>

5.4.4、DELETE

前面我们提到了delete是用于删除数据。

delete方法

```
public void delete(String url, Object...
uriVariables) throws RestClientException {}

public void delete(String url, Map<String,
?> uriVariables) throws RestClientException
{}

public void delete(URI url) throws
RestClientException {}
```

(1) 在服务提供者，HelloController.java中加入
deleteManager方法

```
@DeleteMapping("/service/provider/deletemanager")
public Manager
deleteManager(@RequestParam("id") Integer
id,
@RequestParam("username") String username,
@RequestParam("password") String password){

    Manager manager=new Manager();
    manager.setId(id);
    manager.setUsername(username);
    manager.setPassword(password);
    System.out.println(id+"----"
"+username+"----"+password);

    return manager;
}
```

(2) 服务提供者2，HelloController.java方法中复制 deleteManager方法

(3) 消费提供者，WebController.java中加入 deleteManager方法

```
//.....
@RequestMapping("/service/consumer/deletemanager")
```

```
public String deleteManager() {
    String isArray[] =
    {"1123", "lihao", "lihao12321"};
```



```
        Map<String, Object> map = new
HashMap<String, Object>();
        map.put("id", "1234");
        map.put("username", "zhangsan");
        map.put("password", "zhangsan1234");
```



```
        restTemplate.delete("http://service-
provider/service/provider/deletemanager?id=
{0}&username={1}&password={2}", isArray);
```



```
        restTemplate.delete("http://service-
provider/service/provider/deletemanager?id=
{id}&username={username}&password=
{password}", map);
```



```
        return "执行成功! ";
}
```

```
//.....
```

(4) 启动服务

<http://localhost:8081/service/consumer/deletemanager>

6、服务熔断-Hystrix

6.1、Hystrix概述

6.1.1、微服务架构故障

在微服务架构中，我们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。

6.1.2、案例说明

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，如果还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。

6.1.3、**Hystrix**

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快速开发

6.2.1、添加依赖

服务消费者项目01-service-consumer中加入依赖

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

6.2.2、开启熔断器

ConsumerApplication.java

```
//.....
@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker //开启熔断器功能
public class ConsumerApplication {
    public static void main(String[] args)
{
    SpringApplication.run(ConsumerApplication.
class);
}
}
```

注意：

主类上的三个注解可以通过@SpringCloudApplication 来替换

```
//.....  
//@SpringBootApplication  
//@EnableEurekaClient  
//@EnableCircuitBreaker //开启熔断器功能  
@SpringCloudApplication  
public class ConsumerApplication {  
    public static void main(String[] args)  
{  
  
    SpringApplication.run(ConsumerApplication.  
class);  
}  
}
```

6.2.3、编写熔断操作

服务消费者，在WebController.java里加入方法：
testHystri(), error()

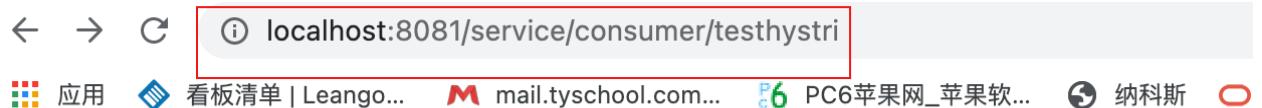
```
@RequestMapping("/service/consumer/testhystri")
@HystrixCommand(fallbackMethod = "error")
//当访问的服务提供者出现问题，跳转到error方法
public String testHystri(){
    return
restTemplate.getForEntity("http://service-
provider/service/provider/test",String.cla-
ss).getBody()+"服务消费者";
}
public String error(){
    //当访问远程服务失败，该如何处理？这些处理逻辑
就可以写在该方法中
    return "error";
}
```

6.2.4、测试

注意：

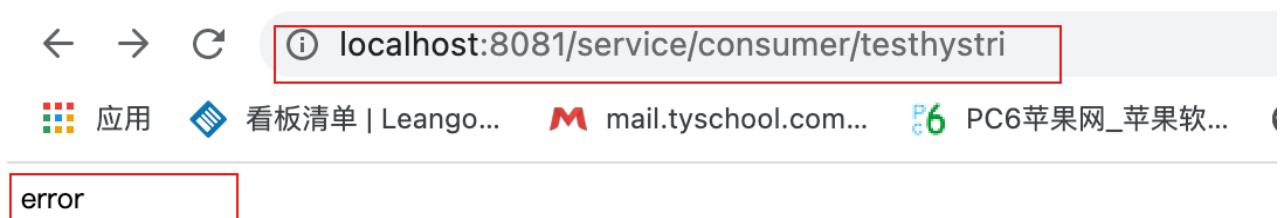
将负载均衡策略调整为轮询

<http://localhost:8081/service/consumer/testhystri>



使用了Eureka注册中心的服务提供者！ provider 1服务消费者

停止服务提供者2后



6.2.5、配置熔断时间

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；

设置时间

```
//设置熔断时间为3.5秒
@HystrixCommand(fallbackMethod = "error",
commandProperties = {

    @HystrixProperty(name="execution.isolation.
    thread.timeoutInMilliseconds",value="3500")

})
```

服务提供者2，test方法加入线程睡眠

```
@RequestMapping("/service/provider/test")
public String test(){
    try {
        Thread.sleep(6000);
    }catch (Exception e){}
    return "使用了Eureka注册中心的服务提供者! provider 2";
}
```

总结：

当请求test接口时，如果3.5秒服务器没有反应，就发生熔断。

6.3、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；

6.4、服务异常处理

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。

6.4.1、服务消费者发生异常

```
@RequestMapping("/service/consumer/testhystri")
@HystrixCommand(fallbackMethod = "error",
commandProperties = {

    @HystrixProperty(name="execution.isolation
.thread.timeoutInMilliseconds",value="3500"
)
}

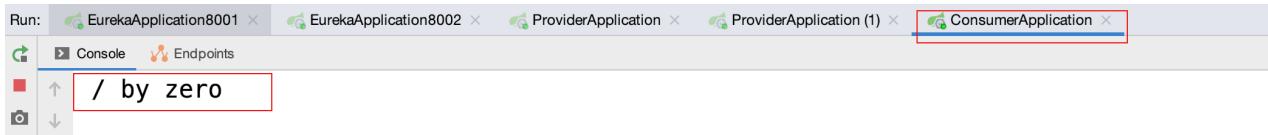
public String testHystri(){
    int a =10/0;//在调用服务提供者时，服务
消费者产生的异常
    return
restTemplate.getForEntity("http://service-
provider/service/provider/test",String.cla
ss).getBody()+"服务消费者";
}

public String error(Throwable throwable){

    System.out.println(throwable.getMessage())
//显示异常信息
    return "error";
}
```

测试

<http://localhost:8081/service/consumer/testhystri>



6.4.2、服务提供者发生异常

```
@RequestMapping("/service/provider/test")  
public String test(){  
    int a=10/0; //异常出现在服务提供者  
    return "使用了Eureka注册中心的服务提供者! provider 1";  
}
```

测试

<http://localhost:8081/service/consumer/testhystri>



6.4.3、异常抛给用户

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(ignoreExceptions = Exception.class)。

```

@HystrixCommand(fallbackMethod = "error",
ignoreExceptions = Exception.class,
commandProperties = {

    @HystrixProperty(name="execution.isolation
.thread.timeoutInMilliseconds",value="3500"
)
}
)

```

测试

<http://localhost:8081/service/consumer/testhystri>



6.5、自定义服务异常处理

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；

6.5.1、创建自定义异常熔断处理

在服务消费者中创建自定义熔断处理请求：

MyHystrixCommand.java

```
package com.ww.hystrix;

import com.netflix.hystrix.HystrixCommand;
import org.springframework.web.client.RestTemplate;
;
/*
 *自定义的hystrix请求
 */
public class MyHystrixCommand extends HystrixCommand<String> {
    private RestTemplate restTemplate;

    public MyHystrixCommand(Setter
setter,RestTemplate restTemplate) {
        super(setter);
        this.restTemplate=restTemplate;
    }

    @Override
```

```
protected String run() throws Exception
{
    //调用远程服务的方法
    return
restTemplate.getForEntity("http://service-
provider/service/provider/test",String.cla-
ss).getBody();
}
/**
 * 当远程服务超时，异常，不可用等情况时，会触
发该熔断方法
 */
@Override
protected String getFallback() {
    //实现服务熔断/降级逻辑方法
    return "error";
}
}
```

使用自定义异常熔断处理，WebController.java加入testHystri2方法

```
@RequestMapping("/service/consumer/testhystri2")
public String testHystri2(){
    MyHystrixCommand
    myHystrixCommand=new
    MyHystrixCommand(com.netflix.hystrix.HystrixCommand.Setter.withGroupKey(HystrixCommand
GroupKey.Factory.asKey("")),restTemplate);
    String s =
    myHystrixCommand.execute();
    //int a =10/0;//在调用服务提供者时，服
   务消费者产生的异常
    return s;
}
```

注意：

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

测试：

<http://localhost:8081/service/consumer/testhystri2>

6.5.2、同步与异步调用

将消费提供者的请求改为异步请求，修改testHystri2方法

```
@RequestMapping("/service/consumer/testhystri2")
public String testHystri2() throws
ExecutionException, InterruptedException {
    //int a =10/0;//在调用服务提供者时，服务消费者产生的异常
    MyHystrixCommand myHystrixCommand=new
MyHystrixCommand(com.netflix.hystrix.HystrixCommand.Setter.withGroupKey(HystrixCommand
GroupKey.Factory.asKey("")),restTemplate);
    //同步调用（该方法执行后，会等待远程的返回结果，拿到了远程的返回结果，该方法才返回，继承往下执行）
    //String s =
myHystrixCommand.execute();
    //异步调用（该方法执行后，不会马上有远程的返回结果，将来会有结果）
    Future<String>
future=myHystrixCommand.queue();
    //阻塞的方法，直到拿到返回结果
    String s=future.get();
    return s;
}
```

测试

<http://localhost:8081/service/consumer/testhystri2>

6.5.3、自定义异常熔断处理异常信息

修改MyHystrixCommand.java中的getFallback()方法

```
/**
 * 当远程服务超时，异常，不可用等情况时，会触发该熔断方法
 */
@Override
protected String getFallback() {
    Throwable throwable
    =super.getExecutionException();

    System.out.println(throwable.getMessage());
;

    System.out.println(throwable.getStackTrace());
;

    //实现服务熔断/降级逻辑方法
    return "error";
}
```

6.6、Hystrix仪表盘

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；

6.6.1、项目创建

创建maven项目：

02-hystrix-server

加入依赖：

spring-cloud-starter-netflix-hystrix-dashboard

```
<dependency>

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

<artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
<version>2.2.6.RELEASE</version>
</dependency>
```

创建启动类：

```
package com.ww;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.hystrix.dashboard.EnableHystrixDashboard;

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

编写配置文件

application.yml

```
server:  
  port: 9000  
#配置监控地址  
hystrix:  
  dashboard:  
    proxy-stream-allow-list: localhost
```

启动服务

<http://localhost:9000/hystrix>



Hystrix Dashboard

输入要监控的服务地址

Cluster via Turbine (default cluster): https://turbine-hostname:port/turbine.stream

Cluster via Turbine (custom cluster): https://turbine-hostname:port/turbine.stream?cluster=[clusterName]

Single Hystrix App: https://hystrix-app:port/actuator/hystrix.stream

Delay: ms Title:

↑
轮询监控的延迟时间， 默认为2000ms

仪表盘上的标题， 默认使用的是URL地址

6.6.2、仪表盘监控服务消费者

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；

添加依赖

```
<dependency>

<groupId>org.springframework.cloud</groupId>
<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
<version>2.2.6.RELEASE</version>
</dependency>

<!-- spring boot提供的一个服务健康检查监控的依赖-->
<dependency>

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

配置监控端点

```
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。

指定信息访问：

```
management.endpoints.web.exposure.include=h  
ystrix.stream
```

这样显示的信息除了默认的health和info外，我们还可能访问hystrix.stream

修改消费者配置：

application.yml

```
#springboot监控端点访问权限,*表示所有的端点都允许  
访问  
management:  
  endpoints:  
    web:  
      exposure:  
        include: "*"
```

启动服务

<http://localhost:8081/actuator/hystrix.stream>

注意：

直接访问/hystrix.stream接口时会输出出一连串的ping: ping: ...，那么在访问/hystrix.stream接口，首先得访问服务消费者工程中的任意一个其他接口（带熔断器接口），然后再访问/hystrix.stream接口即可；

6.6.3、仪表盘监控端口



Hystrix Dashboard

<http://localhost:8081/actuator/hystrix.stream>

Cluster via Turbine (default cluster): https://turbine-hostname:port/turbine.stream
Cluster via Turbine (custom cluster): https://turbine-hostname:port/turbine.stream?cluster=[clusterName]
Single Hystrix App: https://hystrix-app:port/actuator/hystrix.stream

Delay: ms Title:

[View Details](#) | [Edit](#) | [Delete](#)

Monitor Stream

10 of 10

进入后



7、声明式服务消费-Feign

7.1、Feign概述

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、Feign实现消费者

7.2.1、创建Feign项目

01-service-consumer-feign

7.2.2、添加依赖

spring-cloud-starter-netflix-eureka-client和spring-cloud-starter-openfeign

```
<dependency>

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

        <groupId>org.springframework.cloud</groupId>
        >
            <artifactId>spring-cloud-starter-
    netflix-eureka-client</artifactId>
            <version>2.2.6.RELEASE</version>
        </dependency>
        <dependency>
```

```
<groupId>org.springframework.cloud</groupId>
<dependency>
    <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
<dependency>

<groupId>org.springframework.cloud</groupId>
<dependency>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
    <version>2.2.6.RELEASE</version>
</dependency>
```

7.2.3、编写配置文件

application.yml

```
server:
  port: 8083
spring:
  application:
    name: service-consumer-feign
eureka:
  client:
    service-url:
      defaultZone:
        http://eureka8001:8001/eureka,http://eureka
        8002:8002/eureka
```

7.2.4、编写启动类

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.Spr
ingBootApplication;
import
org.springframework.cloud.netflix.eureka.En
ableEurekaClient;
```

```
import  
org.springframework.cloud.openfeign.EnableFeignClients;  
  
@SpringBootApplication  
@EnableEurekaClient  
@EnableFeignClients //表示开启Spring Cloud  
Feign的支持功能  
public class FeignApplication {  
    public static void main(String[] args)  
    {  
  
        SpringApplication.run(FeignApplication.class);  
    }  
}
```

注意：

@EnableFeignClients注解表示开启Spring Cloud Feign
的支持功能；

7.2.5、绑定服务提供者

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。

```
package com.ww.service;

import
org.springframework.cloud.openfeign.FeignClient;
import
org.springframework.web.bind.annotation.RequestMapping;

//FeignClient设置当前的接口时一个Feign的声明式接
口属性
// name 用于指定我们需要访问的服务提供者的服务名
@FeignClient(value = "service-provider")
public interface IHelloService {
    //RequestMapping 这里用于只当我们需要访问的
    //服务提供者的请求路径

    @RequestMapping("/service/provider/hello")
    public String hello();
}
```

7.2.6、调用服务提供者

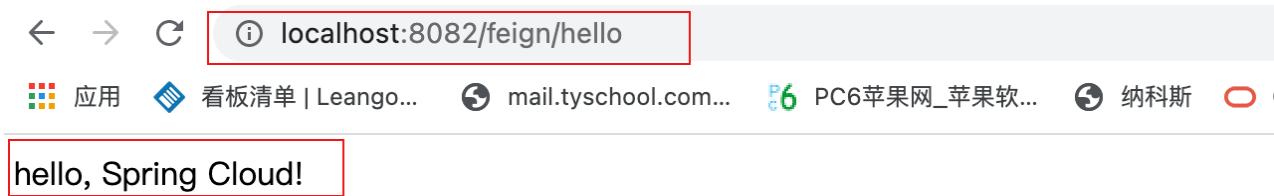
```
package com.ww.controller;

import com.ww.service.IHelloService;
```

```
import  
org.springframework.beans.factory.annotation.Autowired;  
import  
org.springframework.web.bind.annotation.RequestMapping;  
import  
org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class HelloFeignController {  
    @Autowired  
    IHelloService iHelloService;  
    @RequestMapping("/feign/hello")  
    public String hello(){  
        //调用声明式的接口方法，实现对远程服务的调用  
        return iHelloService.hello();  
    }  
}
```

7.2.7、启动服务

<http://localhost:8083/feign/hello>



7.3、Feign-负载均衡

启动服务提供者2，测试

<http://localhost:8083/feign/hello>

前面我们学习了Spring Cloud 提供了Ribbon来实现负载均衡，使用Ribbon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，此方法也是做好负载均衡配置的。

7.4、Feign-服务熔断

feign对hystrix功能支持（支持熔断），可以在@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。

7.4.1、开启熔断功能

appliation.yml加入配置

```
#开启熔断功能
feign:
    hystrix:
        enabled: true
```

7.4.2、指定回调内容

IHelloService.java

```
//.....
//fallback指定回调的类
@FeignClient(value = "service-
provider", fallback = MyFallBack.class)
//.....
```

创建MyFallBack.java

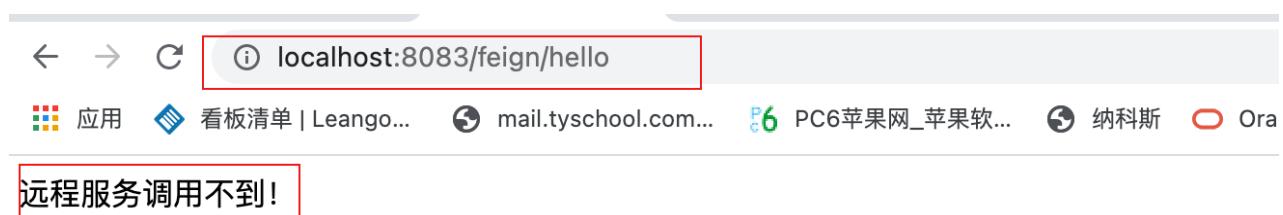
```
package com.ww.fallback;

import com.ww.service.IHelloService;
import org.springframework.stereotype.Component;

@Component
public class MyFallback implements
IHelloService {
    @Override
    public String hello() {
        return "远程服务调用不到!";
    }
}
```

7.4.3、启动服务

<http://localhost:8083/feign/hello>



7.4.4、服务熔断异常信息

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，此时可以使用fallbackFactory

指定回调内容，并可以返回异常信息

```
//.....
@FeignClient(value = "service-
provider", fallbackFactory =
MyFallBackFactory.class)
//.....
```

编写回调类

MyFallBackFactory.java

```
package com.ww.fallback;

import com.ww.service.IHelloService;
import feign.hystrix.FallbackFactory;
import
org.springframework.stereotype.Component;

@Component
public class MyFallBackFactory implements
FallbackFactory<IHelloService> {
```

```
@Override  
public IHelloService create(Throwable  
throwable) {  
    return new IHelloService() {  
        @Override  
        public String hello() {  
            return  
throwable.getMessage();  
        }  
    };  
}
```

启动服务

<http://localhost:8083/feign/hello>



8、配置中心-Config

8.1、Config概述

8.1.1、配置管理

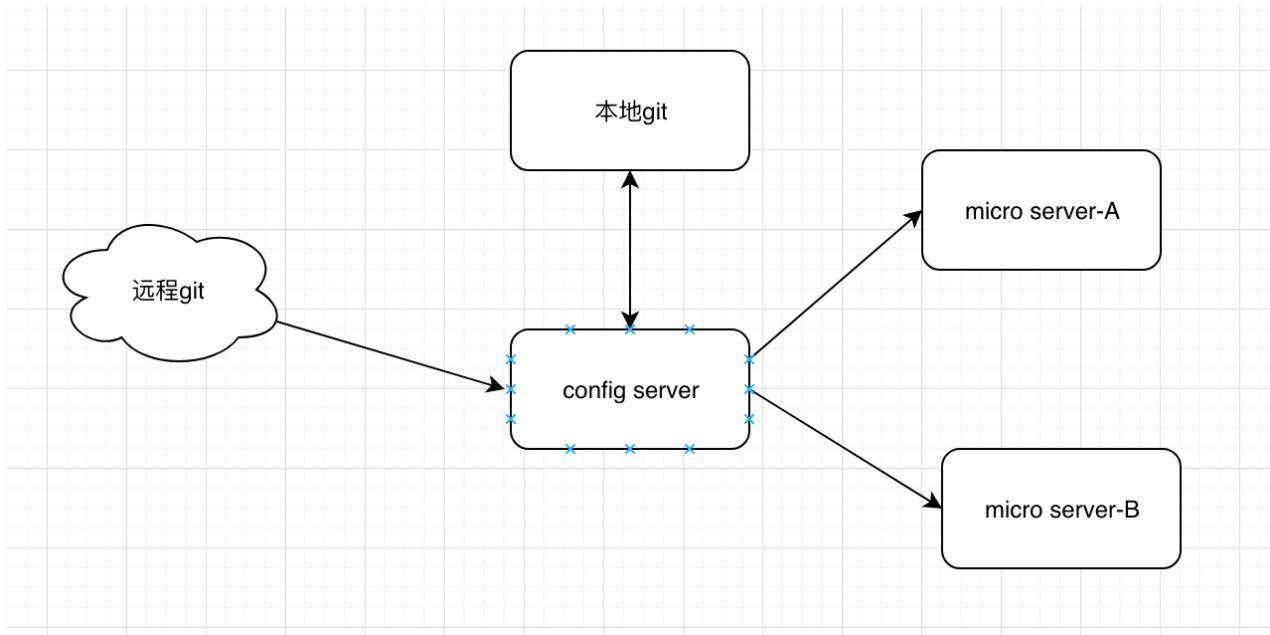
在分布式系统中，尤其是当我们的分布式项目越来越多，每个项目都有自己的配置文件，对配置文件的统一管理就成了一种需要，而 Spring Cloud Config 就提供了对各个分布式项目配置文件的统一管理支持。Spring Cloud Config 也叫分布式配置中心，市面上开源的分布式配置中心有很多，比如国内的，360 的 QConf、淘宝的 diamond、百度的 disconf 都是解决分布式系统配置管理问题，国外也有很多开源的配置中心 Apache 的 Apache Commons Configuration、owner、cfg4j 等等；

8.1.2、SpringCloud-Config

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。

8.1.3、Config工作原理



- 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 ConfigServer 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，ConfigServer 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；

8.2、构建Config配置中心服务端-git

8.2.1、创建项目

02-config-server

8.2.2、导入依赖

spring-cloud-config-server

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-config-server</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>
```

8.2.3、编写配置文件

application.yml

```
server:  
  port: 7000  
spring:  
  application:  
    name: config-server  
cloud:  
  config:  
    server:  
      git:  
        uri:  
          https://github.com/dataww/ww.git  
        username: dataww  
        password: 1985929www
```

- uri 表示配置中心所在仓库的位置
- search-paths 表示仓库下的子目录
- username 表示你的 GitHub 用户名
- password 表示你的 GitHub 密码

8.2.4、编写启动类

ConfigApplication.java

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.SpringBootApplication;
import
org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class ConfigApplication {
    public static void main(String[] args)
{
    SpringApplication.run(ConfigApplication.cl
ass,args);
}
}
```

8.2.5、启动服务

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；

{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（研发）、test（测试）、online（在线）及默认，
{label}

<http://localhost:7000/application/dev/master>



json

```
{
  "name": "application",
  "profiles": [ "dev" ],
  "label": "master",
  "version": "6161d8b282a85db6427b4644eb731322a1c88005",
  "state": null,
  "propertySources": [ {
    "name": "https://github.com/dataww/ww.git/application-dev.yml",
    "uri": "https://github.com/dataww/ww.git/application-dev.yml"
  } ]
}
```

```
"source": {  
    "uri": "http://www.zutuanxue.com/dev"  
}, {  
    "name":  
    "https://github.com/dataww/ww.git/application.yml",  
    "source": {  
        "uri": "http://www.zutuanxue.com"  
    }  
} ]  
}
```

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

8.3、构建Config配置中心客户端

8.3.1、创建项目

02-config-client

8.3.2、导入依赖

spring-cloud-starter-config

```
<dependency>

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

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

8.3.3、编写配置文件

创建 bootstrap.yml 文件，用于获取配置信息，文件内容如下：

bootstrap.yml

```
server:  
  port: 7001  
spring:  
  application:  
    name: config-client  
cloud:  
  config:  
    profile: dev  
    label: master  
    uri: http://localhost:7000/
```

注意：这些信息一定要放在 bootstrap.yml 文件中才有效

8.3.4、编写启动类

ConfigClientApplication.java

```
package com.ww;

import
org.springframework.boot.SpringApplication;
import
org.springframework.boot.autoconfigure.Spr
ingBootApplication;

@SpringBootApplication
public class ConfigClientApplication {
    public static void main(String[] args)
{
    SpringApplication.run(ConfigClientApplicat
ion.class,args);
}
}
```

8.3.5、编写服务调用

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。

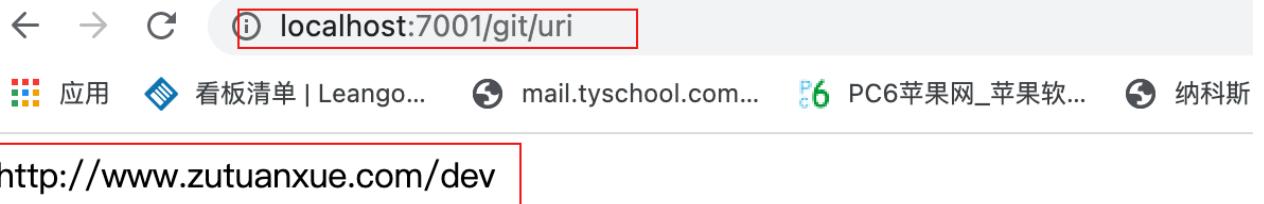
GitController.java

```
package com.ww.controller;
```

```
import  
org.springframework.beans.factory.annotation.Value;  
import  
org.springframework.web.bind.annotation.RequestMapping;  
import  
org.springframework.web.bind.annotation.RestController;  
  
@RestController  
public class GitController {  
    @Value("${uri}")  
    private String uri;  
  
    @RequestMapping("/git/uri")  
    public String git(){  
        return uri;  
    }  
}
```

8.3.6、启动服务

<http://localhost:7001/git/uri>



<http://localhost:7001/git/uri1>

