1. 系统架构的演变
2. 集中式架构
3. 概述

当网站的流量很小时，只需要一个应用，将所有的功能都部署在一起，以减少部署节点和成本。

1. 优点

开发速度快，维护成本低。

1. 缺点

代码耦合度高，维护困难，无法水平扩展，容错率低，并发能力差。

1. 垂直拆分
2. 概述

由于访问量逐渐增大，单一的应用无法满足需求，需要根据业务功能对系统进行拆分。

1. 优点

系统拆分实现了流量分担，解决了并发问题，可以针对不同的模块进行优化，方便水平扩展，负载均衡，提高容错率。

1. 缺点

系统间相互独立，有很多重复开发工作。

1. 分布式服务
2. 概述

当垂直应用越来越多时，应用之间的交互也增多了，我们可以将核心的业务抽取出来作为独立的服务，形成基础服务中心，这样其他应用调用这些常用的服务就很方便。

1. 优点

抽取了基础服务，提高代码的复用和开发效率。

1. 缺点

系统间耦合度增高，调用关系复杂，难以维护。

1. 面向服务结构（SOA）
2. 概述

包含多个服务，服务之间通过相互依赖提供一系列的功能。一个服务通常以独立的形式存在于操作系统进程中，各个服务之间通过网络调用。各个服务之间通过ESB（企业服务总线）连接，为了集成不同的服务，不同协议的服务，ESB做了消息的转化解释和路由工作，让不同的服务互联互通。

1. 缺点

服务粒度较大，ESB集成整合所有服务和协议，数据使得部署，维护困难，且所有的服务都通过一个通道通信，降低了通信的速度。

1. 微服务架构
2. 概述

微服务架构是使用一套小服务来开发单个应用的方式，每个服务基于单一业务能力构建，运行在自己的进程中，并使用轻量级机制通信，通常是HTTP API，并能够通过自动化部署来独立部署。这些服务可以使用不同语言，存储技术，并保持最低限度的集中式管理。

1. 流程

终端通过一个Gateway网关来请求服务，服务通过服务注册中心进行服务注册和管理。网关是一个服务器，是系统的唯一入口，它为每个客户端提供一个定制的API，所有的客户端和消费端都是通过统一的网关接入微服务的，在网关层处理所有的非业务功能。如身份验证，监控，负载均衡，缓存，等等。网关提供RESTful/HTTP的方式访问服务。

1. 特点
2. 每个服务对应单一的业务。
3. 服务拆分粒度很小。
4. 每个服务只需要提供标准接口API，不论该服务如何实现。
5. 服务间相互独立，互不干扰。
6. 前后端分离，提供统一的Rest接口，后端不需要再为PC，移动端开发不同的接口。
7. 数据库分离，每个服务使用自己的数据源。
8. 微服务与SOA的区别

微服务架构和SOA都是对系统进行拆分，SOA中有ESB作为中心总线，而微服务做了去中心化处理，将ESB去掉了，更加灵活，着重分散管理，而不是中央管理。

1. 服务调用方式
2. RPC和HTTP

服务之间的远程调用方式有两种：RPC和HTTP。

1. RPC

RPC全称为 Remote Produce Call远程过程调用，RPC是基于Socket，工作在会话层，自定义数据格式，速度快，效率高。代表产品为webservice,dubbo。

1. HTTP

http是一种网络传输协议，基于TCP,工作在应用层，规定了数据传输的格式。客户端和服务器的通信基本都是HTTP协议，缺点是消息封装臃肿，优点是对于服务的提供和调用没有技术限定，不会限制使用什么语言，技术，HTTP只负责消息的传递。Rest风格，Spring Cloud套件就是通过HTTP协议实现的。

1. RPC与HTTP的区别

RPC是根据语言的API来定义的，而不是根据网络的应用来定义的。它会限制使用语言，技术的灵活性。如果使用的JAVA语言，那么使用Dubbo比较合适。

如果不只是使用一种语言，技术，那么使用HTTP方式来实现服务间的调用更好。

1. Spring RestTemplate的使用

一般有三种http客户端工具类包可以方便地进行http服务调用：httpClient，okHttp，JDK原生URLConnection。Spring提供了RestTemplate的工具类对上述的三种http客户端工具进行了封装，可以直接在Spring项目中使用RestTempplate进行服务调用。

1. Spring Cloud
2. 概述

Spring Cloud是Spring旗下的项目，完全支持Spring Boot的开发，用很少的配置就能够完成微服务框架的搭建。Spring Cloud能够将其他非常流行的技术整合到一起，如配置管理，服务发现，智能路由，负载均衡，熔断器，控制总线，集群状态等功能；协调分布式环境中各个系统，为各类服务提供模板性配置，

1. 组件构成

Spring Cloud不是一个组件，而是许多组件的集合。主要的组件为：注册中心（Eureka），服务网关（Zuul, Gateway），负载均衡（Ribbon），服务调用（Feign），熔断器（Hystrix或Resilience4j）。当我们需要使用某些组件时，直接在项目中添加相关的启动器依赖即可。

1. 流程

当请求通过网关进入后，负载均衡从服务注册中心中获取请求的地址列表，并基于负载均衡算法来选择一个地址来访问；如果服务之间有调用的需求，那么通过服务调用Feign进行调用；如果服务需要修改配置，那么从配置服务器中实时地更新配置。

1. 版本

Spring Cloud 的版本不是普通的数字，而是使用伦敦地铁站的名字来命名的（脑残操作）。

四．服务提供方实例

1. 创建项目工程

创建微服务父工程SpringCloudParentDemo，用户服务提供工程SpringCloudProducer，服务消费工程SpringCloudConsumer。后面两个工程继承父工程，父工程添加Spring Boot父坐标和管理其他组件的依赖；用户服务工程用于整合Mybatis查询数据库中用户数据，提供查询用户数据服务；服务消费工程用于调用查询用户服务数据并输出到浏览器。

1. 配置父工程pom.xml

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.SpringCloud</groupId>  
 <artifactId>com.SpringCloud.ParentDemo</artifactId>  
 <packaging>pom</packaging>  
 <version>1.0-SNAPSHOT</version>  
 <modules>  
 <module>SpringCloudConsumer</module>  
 <module>SpringCloudProducer</module>  
 </modules>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.1.5.RELEASE</version>  
 <relativePath/>  
 </parent>  
  
 <properties>  
 <java.version>1.8</java.version>  
 <spring-cloud.version>Greenwich.SR1</spring-cloud.version>  
 <mapper.starter.version>2.1.5</mapper.starter.version>  
 <mysql.version>5.1.46</mysql.version>  
 </properties>  
  
 <dependencyManagement>  
 <dependencies>  
 <!-- springCloud -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-dependencies</artifactId>  
 <version>${spring-cloud.version}</version>  
 <type>pom</type>  
 <scope>import</scope>  
 </dependency>  
 <!-- 通用Mapper启动器 -->  
 <dependency>  
 <groupId>tk.mybatis</groupId>  
 <artifactId>mapper-spring-boot-starter</artifactId>  
 <version>${mapper.starter.version}</version>  
 </dependency>  
 <!-- mysql驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 <version>${mysql.version}</version>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-config</artifactId>  
 </dependency>  
 </dependencies>  
 </dependencyManagement>  
 <dependencies>  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.16.0</version>  
 </dependency>  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

modules标签内表示的是该父工程中子工程的名称；

parent标签内继承了SpringBoot启动器依赖；

properties标签内表示的是其他配置信息，如java版本，SpringCloud版本，mybatis版本，mysql驱动版本，后面可以直接引用这些版本信息。

dependencyManagement标签内表示的是依赖的坐标管理，只是限定我们需要依赖的坐标，但是并没有真正地导入这些坐标。真正地导入坐标需要在每个子工程中自行导入。

由于spring-cloud-dependencies依赖的scope是import，因此只要包括了该依赖的项目都能够继承该工程中的依赖，因此父工程项目不仅继承了spring-boot-parent，还继承了spring-cloud-dependencies，不需要自己再定义SpringCloud组件的版本了。

1. 配置服务提供子工程的pom.xml

由于父工程中已经配置了依赖的坐标管理，因此，我们直接在子工程导入已经管理的坐标即可，如Mapper启动器，mysql驱动，不需要定义版本。同时还有web启动器。

<dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!-- 通用Mapper启动器 -->  
 <dependency>  
 <groupId>tk.mybatis</groupId>  
 <artifactId>mapper-spring-boot-starter</artifactId>  
 </dependency>  
 <!-- mysql驱动 -->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
</dependencies>

1. 创建引导类和SpringBoot配置文件appliaction.yml

创建引导类入口:

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

配置application.yml，其中包括服务器的端口号，使用数据源的驱动，地址，用户名，密码，同时还有设置mybatis别名包的扫描，这个包和包中的实体类需要自行创建。

server:  
 port: 9091  
spring:  
 datasource:  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 url: jdbc:mysql://localhost:3306/springcloud  
 username:root  
 password:  
  
mybatis:  
 type-aliases-package: com.user.pojo

创建实体类User:

package com.user.pojo;  
  
import lombok.Data;  
import tk.mybatis.mapper.annotation.KeySql;  
  
import javax.persistence.Id;  
import javax.persistence.Table;  
import java.util.Date;  
  
@Data  
@Table(name = "tb\_user")  
public class User{  
 // id  
 @Id  
 //开启主键自动回填  
 @KeySql(useGeneratedKeys = true)  
 private Long id;  
  
 // 用户名  
 private String userName;  
  
 // 密码  
 private String password;  
  
 // 姓名  
 private String name;  
  
 // 年龄  
 private Integer age;  
  
 // 性别，1男性，2女性  
 private Integer sex;  
  
 // 出生日期  
 private Date birthday;  
  
 // 创建时间  
 private Date created;  
  
 // 更新时间  
 private Date updated;  
  
 // 备注  
 private String note;  
}

1. 创建数据表

根据需求创建数据表。

1. 创建Mapper接口

Mapper接口继承Mapper接口，并且要传入参数User对象。

package com.user.mapper;  
  
import com.user.pojo.User;  
import tk.mybatis.mapper.common.Mapper;  
  
public interface UserMapper extends Mapper<User> {  
}

同时需要将该Mapper接口的全包名设置到引导类的注解@MapperScan的参数上面，用来扫描到这个Mapper包。

1. 创建service层的类

创建service层的类来调用Mapper的方法，查询用户User的数据。本次使用的是Mapper中固有的方法，通过主键查询，而不是自定义的UserMapper类中的方法。

package com.user.service;  
  
import com.user.mapper.UserMapper;  
import com.user.pojo.User;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class UserService {  
  
 @Autowired  
 private UserMapper userMapper;  
  
 public User queryById(Long id){  
 return userMapper.selectByPrimaryKey(id);  
 }  
}

1. 创建controller层的类

创建UserController调用userService类的queryById方法，该方法的参数通过请求的地址中的id值来获取。

@RestController是@Controller和@Responsebody的结合，代表着直接返回一个json字符串到页面上，而不是view；

@GetMapping 表示着这是一个Get请求方法的@RequestMapping注解；

@PathVariable 表示着从地址中取出某个参数；

@RestController  
@RequestMapping("/user")  
public class UserController {  
 @Autowired  
 private UserService userService;  
  
 @GetMapping("/{id}")  
 public User queryById(@PathVariable Long id){  
 return userService.queryById(id);  
 }  
}

1. 测试代码运行情况

通过引导类运行Spring程序，访问localhost:8080/user/9将数据库根据id=9索引出来的数据显示在页面上。

五．服务消费方实例

1. 需求

微服务消费方的需求是访问localhost:8081/consumer/8 使用RestTemplate获取localhost:8080/user/8的数据。

1. 配置pom.xml

继承之前创建的父工程SpringCloudParentDemo，然后再添加一个web的启动依赖。

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <parent>  
 <artifactId>com.SpringCloud.ParentDemo</artifactId>  
 <groupId>com.SpringCloud</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
 <modelVersion>4.0.0</modelVersion>  
  
 <artifactId>SpringCloudConsumer</artifactId>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 </dependencies>  
  
</project>

3， 创建引导类和配置文件

创建一个引导类，同时需要在引导类中提前注册一个RestTemplate，以供调用。

@SpringBootApplication  
public class ConsumerApp {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ConsumerApp.class, args);  
 }  
  
 @Bean  
 public RestTemplate restTemplate(){  
 return new RestTemplate();  
 }  
}

4.创建Controller

添加一个Controller，其中有函数queryBuId（）通过RestTemplate将微服务提供的服务数据截取过来，返回一个User对象，这个User对象与服务提供方的User对象是一样的。注意，User对象一定要设置get和set方法，不然会出错。

@RestController  
@RequestMapping("/consumer")  
public class ConsumerController {  
 @Autowired  
 private RestTemplate restTemplate;  
  
 public User queryBuId(Long id){  
 String url = "http://localhost:8081/user/8";  
 return restTemplate.getForObject(url, User.class);  
 }  
}

1. 测试微服务消费方

启动引导类，访问localhost:8081/consumer/8，就会得到微服务提供方获取的数据库的数据，并显示在页面上。

六．总结项目的问题

（一）服务管理

1. 服务如何自动注册和发现，

2. 如何实现状态监管，服务提供方的状态和服务消费方的状态都是需要监测。

3. 如何实现动态路由。

（二）负载均衡

当有多个服务器运行时，如何实现服务的负载均衡。

（三）容灾问题

当有服务器宕机时，如何解决容灾问题。

（四）服务统一配置问题

若有多个服务器，如何对多个服务器进行统一的配置。

七． Eureka注册中心

（一）概述

上面的服务项目实例中，服务提供方对外提供服务，需要对外暴露自己的地址，而服务调用者需要记录服务提供者的地址，将来地址出现变动时，需要及时更新。当服务数量比较少时，这些不会造成困扰，一旦服务数量增多，管理上就非常麻烦。这就需要服务注册中心来统一管理。

（二）Eureka注册中心

Eureka负责管理 ，记录服务提供者的消息；服务调用者无需自己寻找服务，而是把自己的需求告诉Eureka，然后Eureka会把符合你需求的服务提供给你。同时，服务提供方与Eureka之间通过“心跳”机制来进行监控，当某个服务提供方出现问题时，Eureka会把它从服务列表中删除。通过服务注册中心实现了服务的自动注册，发现，状态监控。

1. 原理流程
2. 服务提供者实例化服务，将服务注册到Eureka服务注册中心中。
3. Eureka记录服务提供方的地址。
4. 服务消费方从Eureka中获取服务列表的地址。
5. 基于负载均衡算法从地址列表中选择一个服务地址调用服务。
6. 服务提供方会定期发送心跳给Eureka，告诉它这个服务时可用的。Eureka会定期检查那些没有定期发送心跳的服务，将它们在一定时间内剔除出服务地址列表、

八．搭建Eureka注册中心工程

（一）配置pom.xml

依然是继承之前设置的父工程SpringCloudParentDemo，然后添加eureka-server的启动依赖，这是netflix公司的，版本不需要设置，因为在父工程中已经添加了spring-cloud依赖了。

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <groupId>com.springboot</groupId>  
 <artifactId>eureka-server</artifactId>  
 <version>1.0-SNAPSHOT</version>  
  
 <parent>  
 <artifactId>com.SpringCloud.ParentDemo</artifactId>  
 <groupId>com.SpringCloud</groupId>  
 <version>1.0-SNAPSHOT</version>  
 </parent>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>  
 </dependencies>  
  
  
</project>

（二）创建引导类和配置文件

创建引导类，并且添加注解@EnableEurekaServer，表示这是一个Eureka服务器。

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

添加配置文件application.yml

配置文件application.yml配置Eureka服务器的参数：

server:  
 port: 8082  
spring:  
 application:  
 name: eureka-server  
eureka:  
 client:  
 service-url:   
 *#eureka服务地址，如果是集群的话，需要制定其他集群eureka的地址* defaultZone: http:127.0.0.1/8082/eureka  
 *#单个服务器不注册自己到Eureka中* register-with-eureka: false  
 *#不拉取服务* fetch-registry: false

spring.application.name是Eureka服务的名称。

eureka.client.service-url 是eureka服务的地址，当有多个服务器成集群时，需要指定其他集群eureka的地址

eureka.client.register-with-eureka：是否注册Eureka本身到Eureka中，在单个服务器中是无所谓的，当呈集群存在时，需要注册Eureka本身到Eureka中被其他Eureka发现。

eureka.client.fetch-registry:是否需要拉取服务。

（三）服务的注册与发现

我们需要将服务提供方的服务注册到eureka并且在服务消费方能够根据服务名称调用服务。

1. 服务注册
2. 在服务提供方添加Eureka客户端依赖，自动将服务注册到EurekaServer服务地址列表中。

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

1. 改造引导类，添加开启Eureka客户端发现的注解。

@EnableEurekaClient

该注解用于开启Eureka客户端的发现功能。

1. 修改配置文件，添加服务提供方的应用名称，设置Eureka服务地址。

spring.application.name=user-service

eureka.client.service-url.defaultZone=http://127.0.0.1:8082/eureka

1. 启动测试

在Eureka服务器已经开启的情况下，开启服务提供方，那么服务提供方就会出现在Eureka的管理界面上。

1. 服务发现
2. 在服务消费方添加Eureka客户端依赖，可以使用工具类根据服务名称获取对应的服务地址列表。

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

1. 改造引导类，添加开启Eureka客户端发现的注解。

@EnableEurekaClient

该注解用于开启Eureka客户端的发现功能。

1. 修改配置文件，设置Eureka服务地址，并且设置服务消费方的名称。

server:  
 port: 8081  
  
eureka:  
 client:  
 service-url:  
 defaultZone: http://127.0.0.1:8082/eureka  
spring:  
 application:  
 name: consumer-demo

1. 改造Controller，使用工具类DiscoveryClient根据服务名称获取的对应服务地址列表。

原来的Controller中服务提供方的地址是直接写死在代码中的，这样不利于扩展，我们可以注入一个工具类DiscoveryClient，该工具类可以获取Eureka服务器中注册的服务实例，通过该实例可以过去该实例的主机，以及端口号，以此可以拼接成该服务的地址。再通过RestTemplate的getForObject函数来访问该地址，得到服务提供的数据。

@RestController  
@RequestMapping("/consumer")  
public class ConsumerController {  
 @Autowired  
 private RestTemplate restTemplate;  
  
 @Autowired  
 private DiscoveryClient discoveryClient;  
  
 @GetMapping("/{id}")  
 public User queryBuId(@PathVariable Long id){  
 String url = "http://localhost:8080/user/8";  
  
 //获取eureka中注册的user-service的实例  
 List<ServiceInstance> serviceInstances = discoveryClient.getInstances("user-service");  
 ServiceInstance serviceInstance = serviceInstances.get(0);  
 url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/" + id;  
 return restTemplate.getForObject(url, User.class);  
 }  
}

1. 测试服务消费方

开启服务消费方，可以得到服务提供方提供的服务数据，并且服务提供方和服务消费方都已经在Eureka管理界面中了。

九．高可用的Eureka Server

（一）概述

在之前的案例中，我们搭建了一个EurekaServer，而EurekaServer也可以是一个集群，形成高可用的Eureka中心。所谓的高可用，就是保证当一个Eureka Server宕机后，另一个Eureka Server 能够立即补上去，这就需要多个Eureka Server之间需要数据同步。

（二）服务同步

多个Eureka Server之间能够互相注册为服务，集群中的每个Eureka Server 需要将自己注册到其他Eureka Server 服务之中，这样其他的Eureka Server就能够互相发现。当服务提供者注册到Eureka Server集群中的某个节点时，该节点会把服务的信息同步给集群中的每个节点，从而实现数据同步。因此，无论客户端访问到Eureka Server集群中的任意一个节点，都能够获取到完整的服务列表信息。

（三）配置高可用的Eureka Server

直接修改之前配置的Eureka Server，之前配置的Eureka 的服务地址使用的是自己的Eureka Server地址，现在我们要配置集群，需要修改为其他服务器的地址。

需要将自己注册到Eureka 中，设置register-with-eureka，默认是true。

server:  
 port: ${port:8082}  
spring:  
 application:  
 name: eureka-server  
eureka:  
 client:  
 service-url:  
 *#eureka服务地址，如果是集群的话，需要制定其他集群eureka的地址* defaultZone: ${defaultZone:http://127.0.0.1/8083/eureka}

这里，我们使用一种新的语法：

port: ${port:8082}

这种语法的意思是如果port中有值的话，就使用port中的值，否则使用后面的值。要使用这种语法，需要设置这些参数的默认值，在IDEA的右上角选择Eureka 项目的配置，在Configuration 的Environment中，设置VM options为使用的参数的默认值，使用-D表示这是默认值。如下所示：

-Dport=8082 -DdefaultZone=http://127.0.0.1:8083/eureka

可以看到，这里设置的port默认值为8082，这是本台Eureka Server的端口号，defaultZone的默认值为其他Eureka Server 的地址，即如果有集群，则注册其他的Eureka Server的地址到本台Eureka Server中。

（四）复制配置好的Eureka Server应用

同样是在Eureka Server的应用配置页面，直接复制该应用，得到另一个Eureka Server，修改名称，并且修改参数的默认值，将port的默认值改为新的端口号8083，将defaultZone的默认值改为<http://127.0.0.1:8082/eureka>。

1. 测试高可用集群

同时启动两个Eureka Server应用，可以在两个的Eureka 管理界面中看到，每个都已经注册了两个Eureka Server了，一个是8082端口，一个是8083端口。设置服务提供方和服务消费方中配置的都是8082端口的服务器，当我们停掉8082端口的服务器时，发现服务提供方和服务消费方依旧能够运行。

在配置服务提供者和服务消费方时，为了保证稳定，我们可以将多个Eureka Server的地址都配置到其中。

十．Eureka客户端与服务端配置

（一）Eureka服务提供方客户端配置

1. 服务注册

服务提供者在启动时，会检测配置属性中的 eureka.client.register-with-eruka-true参数是否正确，默认是true。如果值是true，那么服务提供者就会向Eureka Server发起一个Rest请求，并携带自己的元数据信息，Eureka Server会把这些信息保存到一个双层Map结构中。双层Map结构如下所示：

第一层Map的Key是服务id。根据我们配置的spring.application.name属性来获取。

第二层Map的Key是服务的实例id，一般是host+serviced+port.

值是服务的实例对象，一个服务可以启动多个不同的实例，形成集群。

且，在默认注册时是使用的是主机名或者localhost，如果想用ip进行注册，需要在服务提供者的配置中手动添加ip地址，并且设置倾向ip地址：

eureka：

instance:

ip-address: 127.0.0.1

prefer-ip-address：true

注意，这样设置之后，只是在代码中变成了ip地址，然而Eureka的控制台中依然是localhost。

2．服务续约

注册服务以后，服务提供者会持续地提供一个心跳，即定时向Eureka Server 发起Rest请求，表示该服务依旧存在，这种操作称为服务续约。服务提供方默认的定时续约的间隔时间为30秒，同时默认的服务失效的等待心跳时间为90秒，即若服务提供方90秒内没有心跳，则该服务就会失效，但是服务何时从Eureka Server服务列表中剔除出来，就要看Eureka Server定时扫描时间的设置了，只有Eureka Server定时扫描发现服务没有发送心跳时，他才会从服务列表中剔除。

设置服务续约的参数，如下所示：

eureka:

instance:

#服务失效

lease-expiration-duration-in-seconds: 90

#服务续约

leas-renewal-interval-in-seconds: 30

1. Eureka服务消费方客户端配置
2. 获取服务列表

当服务消费方启动时，会检测Eureka.client.fetch-registry参数的值，如果是true，那么会从Eureka Server的服务列表中拉取只读备份，缓存在本地中，每隔30秒重新拉取更新。在服务消费方的配置中修改如下参数：

eureka:

client:

registry-fetch-interval-seconds: 30

1. Eureka Server服务端配置
2. 服务下线

当服务正常关闭时，它会触发一个服务下线的Rest请求给Eureka Server，服务器接收到该请求后将服务设置为下线状态。

1. 失效剔除

当Eureka Server没有收到服务下线的请求，同时Eureka Server有一个定时扫描任务，默认是60秒扫描一次当前服务清单中超时未续约的服务，并剔除出服务列表。可以通过eureka.server.eviction-interval-timer-in-ms 参数默认的时间，单位是毫秒。

eureka:

server:

eviction-interval-timer-in-ms: 60000

1. 自我保护

当我们关停一个服务时，Eureka Server的控制台中会有一个警告，这是Eureka的自我为保护机制，当服务没有按时续约时，Eureka会统计该服务实例 最近15分钟续约的比例是否低于85%。由于网络的波动，服务未续约的情况时有发生，因此Eureka Server在扫描到有未续约的服务时，不会直接删除，而是等待网络恢复正常。这种保护机制保障了服务的稳定。在生产环境中比较实用，但是在开发过程中我们会频繁地打开关闭服务，因此，最好将自我保护机制关掉。

可以通过在Eureka Server中设置如下参数关闭自我保护：

eureka:

server:

enable-self-preservation: false

十一. 负载均衡Ribbon

1. 概述

在实际开发中，一个服务一定有多个服务地址，当我们在服务消费方通过DiscoveryClient获取服务实例信息时，获取到的是一个服务地址列表，这时就需要负载均衡算法来决定到底访问哪一个服务。

1. Ribbon

Eureka 中已经集成了负载均衡组件：Ribbon，这是Netflix发布的负载均衡器，它有助于控制HTTP和TCP客户端的行为，Ribbon配置服务提供者地址列表后，Ribbon就可以基于某种负载均衡算法，自动的为服务消费者去请求，包括轮询，随机等算法。Ribbon默认的负载均衡算法时轮询，通过在服务消费方的配置文件中修改如下参数：{服务名称}.ribbon.NFLoadBalancerRuleClassName，前面是服务的名称，后面是算法的名称。

如：修改服务user-service为随机：

user-service:

ribbon:

NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule

1. 实现步骤
2. 要实现负载均衡，首先最起码要有两个及以上的服务提供方，创建一个8079端口，一个8080端口。
3. 修改服务消费方的RestTemplate实例化方法，添加负载均衡注解@LoadBalnced

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

1. 修改服务消费方的controller，直接使用服务名称来替代原来的url中的host+port，这样Ribbon就能够自动负载均衡到一个服务地址中。

String url = "http://user-service/user/" + id;  
return restTemplate.getForObject(url, User.class);

1. 实现原理

为何我们只在url中提供了服务的名称就可以实现负载均衡呢？显然是有组件根据服务名称，获取到了服务实例的ip和端口号。Spring的负载均衡自动配置类LoadBalancerAutoConfiguration.LoadBalancerInterceptorConfig会自动配置负载均衡拦截器LoadBalncertInterceptor。这个类会对RestTemplate的请求进行拦截，从Eureka中根据服务的名称获取到服务列表，最后根据负载均衡算法得到真实的服务地址信息，替换url地址。访问服务地址，并且获取服务数据。

十二. 熔断器Hystrix

1. 概述

Hystrix是一款提供保护机制的组件，由netflix公司研发。它是一个延迟和容错库，用于隔离访问远程服务，第三方库，防止出现级联失败。

1. 雪崩问题

在微服务的调用过程中，一个请求可能需要调用多个微服务接口才能够实现，调用过程比较复杂。当它调用的某一个服务发生异常时，请求被阻塞了，且这个线程并不会被释放，后面请求会继续发送来一直阻塞在这里。服务器资源被耗尽，导致其他服务都不可用了，形成了雪崩效应。Hystrix解决雪崩问题的手段是服务降级，包括线程隔离和服务降级。

1. 线程隔离

Hystrix为每个服务调用分配了一个小的线程池，如果线程池已经满调用了就立即拒绝服务调用，如果线程池中还有空闲的线程可以访问，则通过它来访问服务。

1. 服务降级

当线程池已经满了，或者请求超时时，则会进行服务降级处理。所谓的服务降级指的是返回一个服务调用失败的结果，至少用户的请求不会被阻塞，更不会无休止地等待到系统崩溃。

1. Hystrix实践
2. 引入hystrix依赖

由于Hystrix是作用于服务消费方消费服务的时候，因此在服务消费方的pom.xml中添加依赖：

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

1. 开启降级

在服务消费者的引导类上添加注解：@EnableCircuitBreaker

注意，在微服务中，服务消费方引导类通常会引入三个注解：

@SpringBootApplication

@EnableDiscoveryClient

@EnableCircuitBreaker

这三个注解可以组合成一个注解：@SpringCloudApplication，但是本人觉得还是不要合起来写最好。

1. 服务降级逻辑编写

（1）概述

当服务访问失败时，就会通过服务降级返回失败的结果，同样是在服务消费方的Controller中定义服务访问失败结果，我们需要定义一个新的方法，一个服务对应一个访问失败的方法。

（2）添加注解

首先，要在该服务方法上面添加注解

@HystrixCommand(fallbackMethod=”queryByIdFallBack”)

其中的参数fallbackMethod代表着该服务对应的访问失败方法，值为方法的名称。

（3）定义访问失败的方法

定义这个访问失败的方法，注意，这个方法返回的一定是String，同样的，该方法对应的服务方法的返回值也要改为String，这是重点。

@GetMapping("/{id}")  
 @HystrixCommand(fallbackMethod = "queryByIdFallBack")  
 public String queryBuId(@PathVariable Long id){  
// String url = "http://localhost:8080/user/8";  
//  
// //获取eureka中注册的user-service的实例  
// List<ServiceInstance> serviceInstances = discoveryClient.getInstances("user-service");  
// ServiceInstance serviceInstance = serviceInstances.get(0);  
// url = "http://" + serviceInstance.getHost() + ":" + serviceInstance.getPort() + "/user/" + id;  
 String url = "http://user-service/user/" + id;  
 return restTemplate.getForObject(url, String.class);  
 }  
  
 public String queryByIdFallBack(Long id){  
 return "Sorry, the connection has failed";  
 }

注意，原来的服务方法已经将返回值从User改为了String！！！

此时，进行测试，当我们停止服务提供方的运行时，服务消费方访问会返回失败方法的返回值。

（4）为类设置默认的访问失败方法

为每个服务方法定义一个访问失败的方法是非常麻烦的，我们可以为整个类定义一个默认的访问失败的方法，作为该类中所有的服务访问失败的默认方法。

在类上面添加注解  
@DefaultProperties(defaultFallback=”defaultFallback”)

同样的，，defaultFallback参数表示这是默认的访问失败方法，值为方法的名称。继而定义这个方法。

在对应的服务方法上添加注解@HystrixCommand，不过这次不用添加参数

public String defaultFallback(){

return “default fail”;

}

我尝试着在对应的服务方法上依旧添加注解@HystrixCommand(fallbackMethod = "queryByIdFallBack")，这样可以为每个服务设定个性化的访问失败方法。

1. 设置Hystrix访问超时时间

在服务消费方中配置Hystrix访问超时时间，通过以下参数：

hystrix:  
 command:  
 default:  
 execution:  
 isolation:  
 thread:  
 timeoutInMilliseconds: 3000

这样，超时时间就设定为了3秒，当访问服务超过3秒没有响应时，那么就会启动服务降级，调用访问失败方法。我们可以通过浏览器自带的检查来查看访问该服务耗费的时间。

十三. 服务熔断

1. 熔断原理

在分布式系统中应用服务熔断后，服务调用方可以自己进行判断哪些服务反应慢或者存在大量超时，可以针对这些服务进行主动熔断，防止整个系统被拖垮。Hystrix的服务熔断机制，可以实现弹性容错，当服务请求情况好转以后，可以自动重连。通过断路的方式，将后续请求直接拒绝，一段时间（默认5秒）之后尝试地允许部分请求通过，如果这些请求调用成功则回到熔断器关闭的状态，否则继续打开熔断器，拒绝请求的服务。

1. Hystrix熔断状态机模型
2. closed

熔断器关闭状态，所有的需求都能够正常访问。

1. open

熔断器打开状态，所有的服务都会被降级。Hystrix会对请求情况进行继受，当一定时间内失败请求百分比达到阈值，则触发熔断，熔断器会完全打开。默认失败比例的阈值时50%，请求次数最少不低于20次。

1. Half open

半开状态，不是永久的。熔断器打开后会进入休眠时间（默认是5s），随后熔断器会自动进入半开状态，此时会尝试地是释放部分请求，如果这些请求都能够执行成功的话，则会关闭熔断器，否则熔断器将继续保持打开，再次进行休眠计时。

1. 修改服务熔断的参数

我们可以在服务消费方中修改服务熔断的参数。如下所示：

hystrix:  
 command:  
 default:  
 cirouitBreaker:  
 errorThresholdPercentage: 50 *#触发熔断错误比例阈值* sleepWindowInMilliseconds: 10000 *#熔断后休眠时间* requestVolumeThreshold: 10 *#熔断触发最小请求次数*

errorThresholdPercentage: 触发熔断错误比例阈值

sleepWindowInMilliseconds: 熔断后休眠的时间

requestVolumeThreshold: 熔断触发最小请求次数