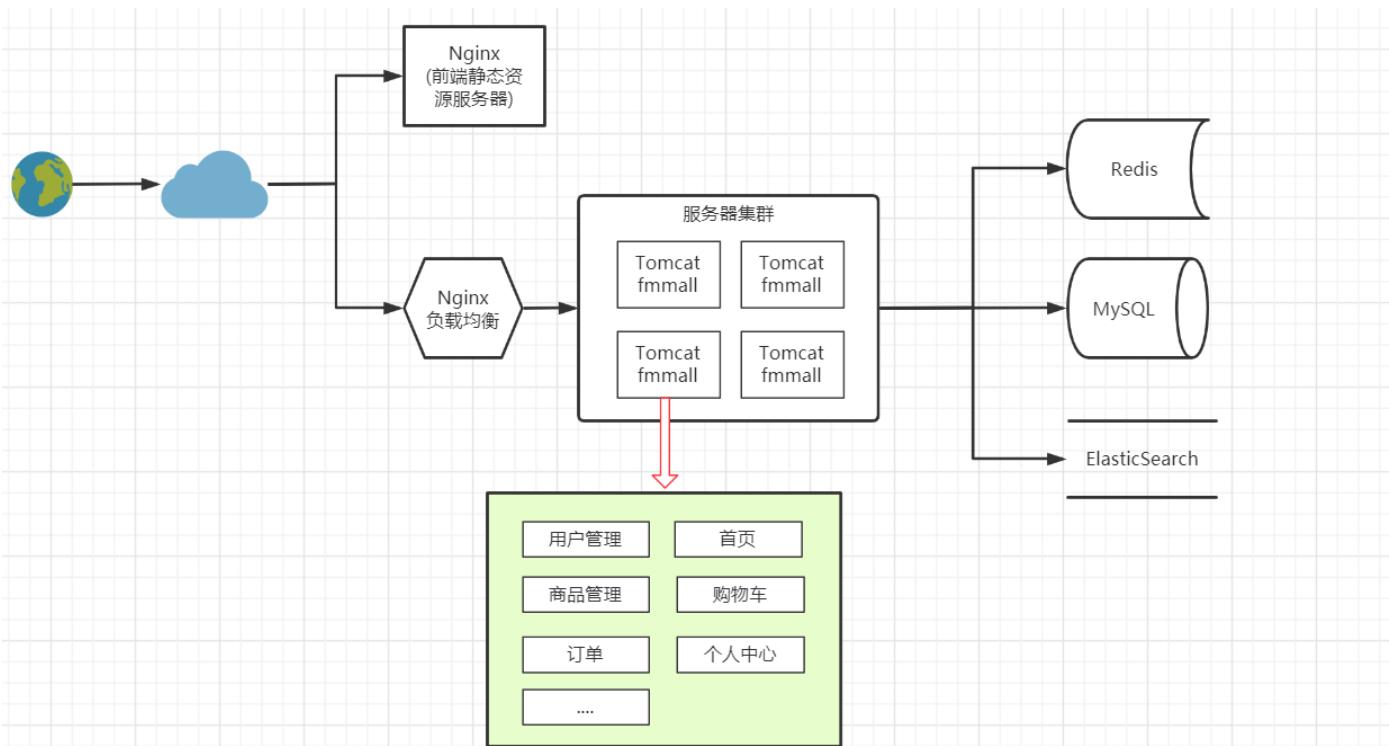


# 一、单体应用存在的问题

## 1.1 锋迷商城项目架构

为了解决高并发问题，我们的Tomcat采用了集群部署，但是每个Tomcat节点上依然是单体项目（虽然是前后端分离，但是后端采用的是单体开发——所有的接口都在同一个项目中）



## 1.2 单体项目存在的问题

一个成功的应用必然有一个趋势：用户量会不断增加、项目的业务也会不断的扩展

用户量的增加会带来高并发的问题，高并发问题解决方案：

- 应用服务器 --> 单体优化 --> 集群（负载均衡、分布式并发）
- 数据库服务器 --> 数据库优化 --> 缓存redis --> 分布式数据库

项目业务的扩展，也会带来一些问题：

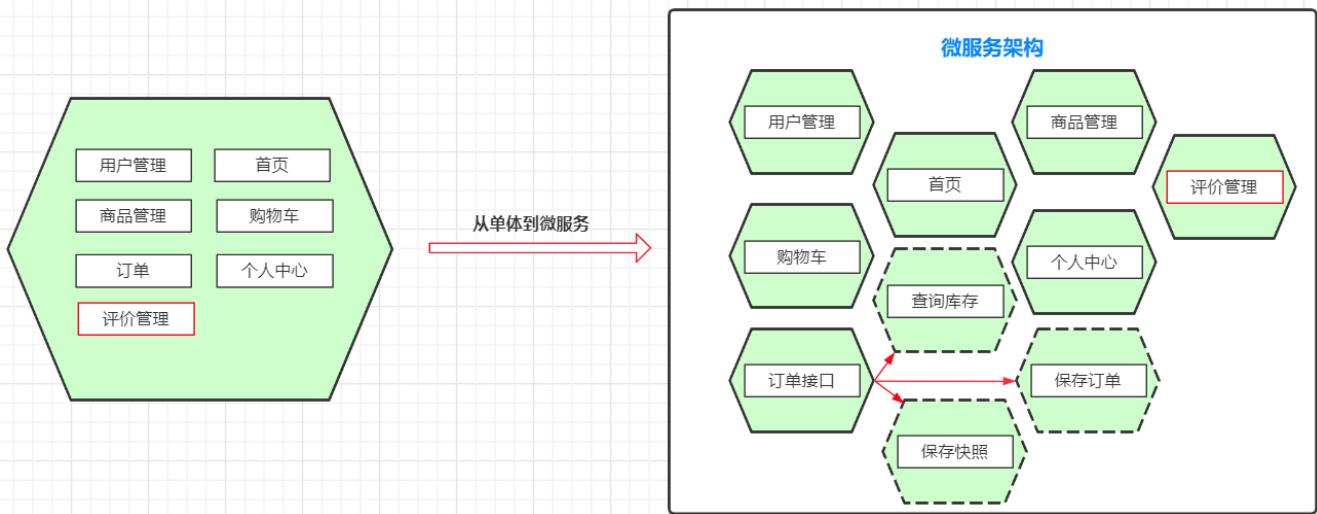
- 项目结构越来越臃肿（项目结构和代码复杂、项目体积逐渐变得庞大）
- 项目结构和代码复杂导致项目不易维护和二次开发、扩展和更新就会变得困难
- 项目体积逐渐变得庞大导致启动时间越来越长、生产力大受限制
- 单体应用中任何一个模块的任何一个bug都会导致整个系统不可用（单点故障）
- 复杂的单体项目也会带来持续部署的障碍
- 单体项目使得采用新的技术和框架会变得困难

## 二、微服务架构

### 2.1 微服务架构概念

微服务架构是一种架构概念，就是将一个单体应用中的每个功能分解到各个离散的服务中以实现对单体应用的解耦，并提供更加灵活的服务支持

#### 从单体到微服务



### 2.2 微服务架构优点

- 解决了单体项目的复杂性问题
- 每个服务都可以由单独的团队进行开发
- 每个服务都可以使用单独的技术栈进行开发
- 每个服务都是独立的进行部署和维护
- 每个服务都可以独立进行扩展

### 2.3 微服务架构缺点

- 微服务架构本身就是一个缺点：如何把握“微”的粒度；
- 微服务架构是一个分布式系统，虽然单个服务变得简单了，但是服务之间存在相互的调用，整个服务架构的系统变得复杂了；
- 微服务架构需要依赖分布式数据库架构；
- 微服务的单元测试及调用变得比单体更为复杂；
- 部署基于微服务架构的应用程序变得非常复杂；
- 进行微服务架构的应用程序开发的技术成本变得更高。

## 三、微服务架构开发需要解决的问题

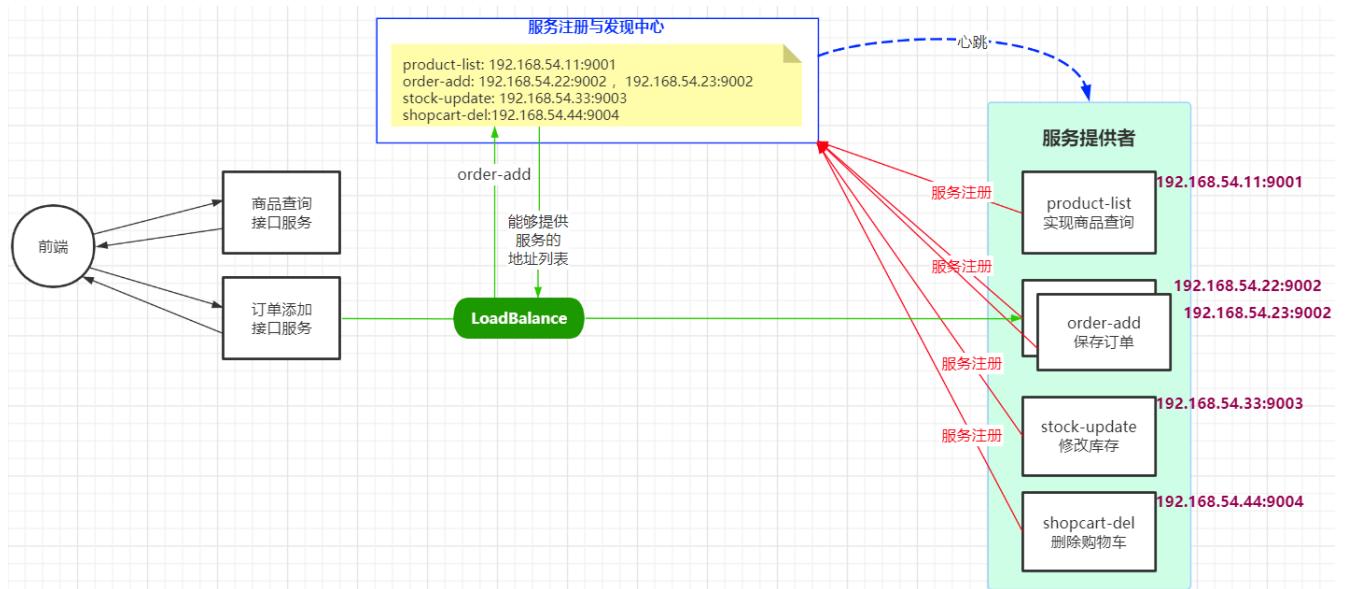
在微服务架构开发的系统中必然存在很多个服务，服务之间需要相互感知对方的存在，需要进行服务间的调用，该如何实现呢？——进行微服务架构开发需要解决的问题：

- 如此多的服务，服务间如何相互发现？
- 服务与服务之间该如何通信？
- 如果某个服务挂了，该如何处理？
- 前端访问多个不同的服务时该如何统一访问路径呢？

### 3.1 服务之间如何相互发现？

微服务架构——每个服务只处理一件事情/一个步骤，在一个复杂的业务中必然存在服务间的相互调用，服务想要相互调用就需要先发现对方。

#### 通过服务注册与发现中心实现服务间的相互发现



- 服务注册与发现中心也是一台独立服务器
- 1.服务提供者在服务注册与发现中心进行注册
- 2.服务注册与发现中心进行服务记录，并与服务提供者保持心跳
- 3.服务消费者通过服务注册与发现中心进行服务查询（服务发现）
- 4.服务注册与发现中心返回可用的服务的服务器地址列表
- 5.服务消费者通过负载均衡访问服务提供者

### 3.2 服务之间如何进行通信？

服务消费者在调用服务提供者时，首先需要通过 **服务注册与发现中心** 进行服务服务查询，返回服务列表给服务消费者，**服务消费者** 通过LoadBalance调用 **服务提供者**，那么他们之间是如何通信呢？—— 数据传输规则

服务与服务间的通信方式有2种：同步调用 和 异步调用

### 3.2.1 同步调用

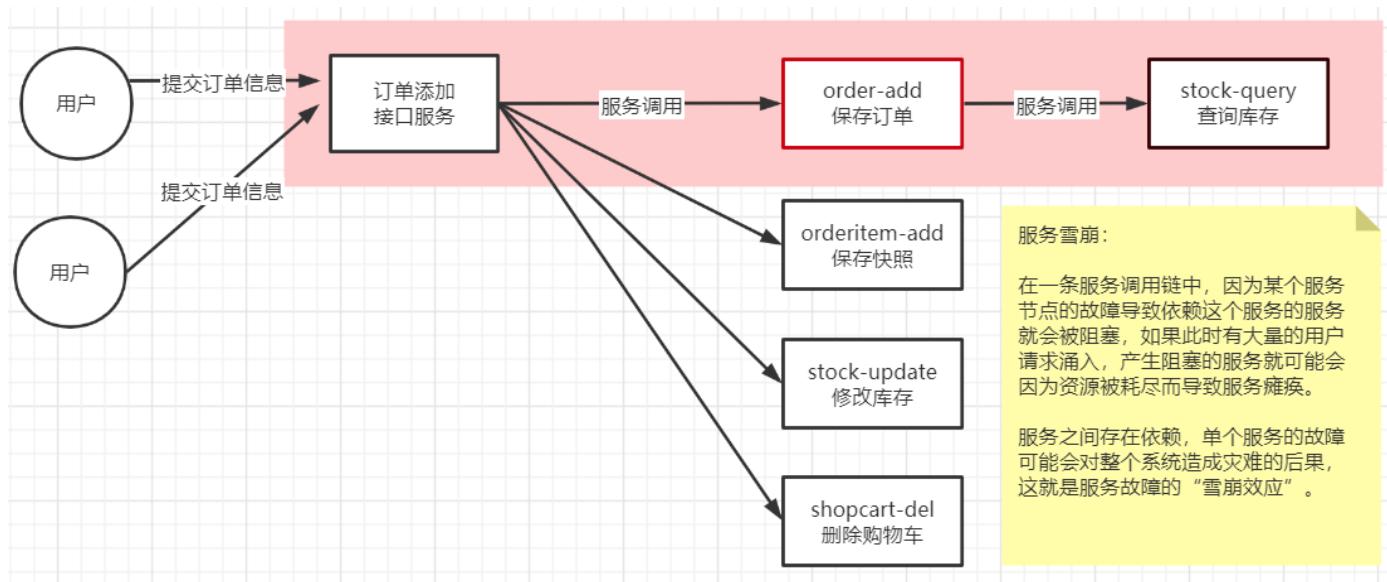
- REST (SpringCloud Netflix, SpringCloud Alibaba)
  - 基于HTTP协议的请求和响应
  - 更容易实现、技术更灵活
  - 支持多语言、同时可以实现跨客户端
  - 适用面很广
- RPC (Dubbo)
  - 基于网络层协议通信
  - 传输效率高
  - 安全性更高
  - 如果有统一的开发规划或者框架，开发效率是比较高的

### 3.2.2 异步调用

服务间的异步通信通常是通过消息队列实现的

## 3.3 服务挂了该如何解决？

### 3.3.1 服务故障雪崩

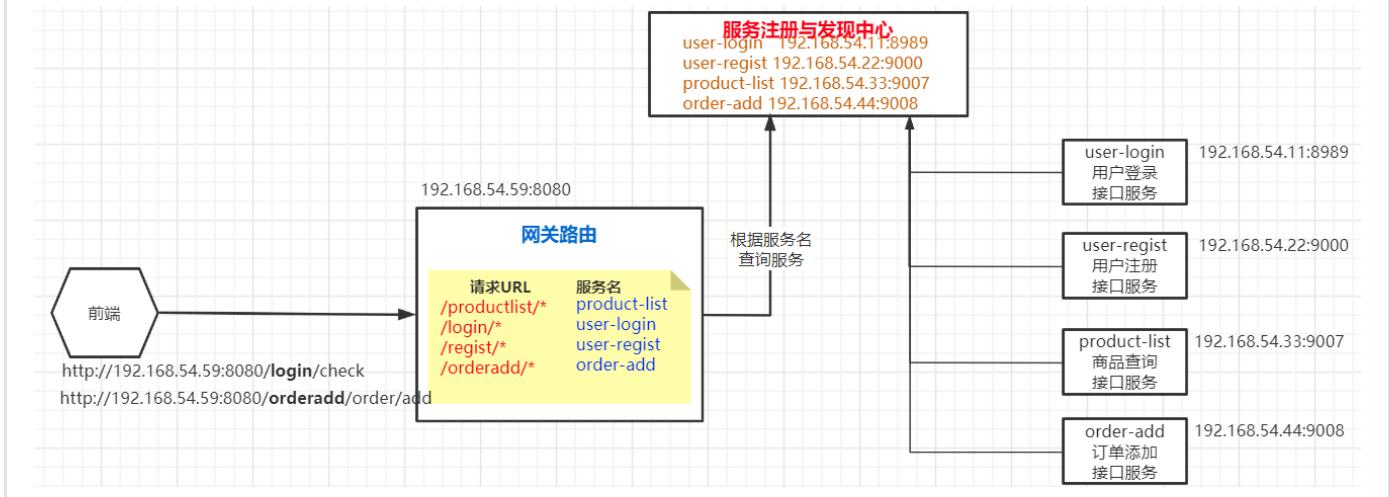


### 3.3.2 如何解决服务故障雪崩

- 服务集群——尽量保证每个服务可用
- 服务降级与熔断——避免请求阻塞造成正常的服务出现故障

## 3.4 客户端如何统一访问多个接口服务？

## 通过路由网关实现接口的统一访问



## 四、微服务架构框架

### 4.1 主流的微服务架构框架

- Dubbo (阿里、开源apache) : 2012年推出、2014年停更、2015年又继续更新
- Dubbox (当当网基于Dubbo的更新)
- jd-hydra (京东基于Dubbo的更新)
- SpringCloud Netflix (2016年) / SpringCloud Alibaba
- ServiceComb (CSE) 华为 2017年

### 4.2 SpringCloud简介

Spring Cloud 是一个基于SpringBoot实现的微服务架构应用开发框架，它为我们进行微服务架构应用开发提供了服务注册与发现、熔断器、网关路由、配置管理、负载均衡、消息总线、数据监控等一系列工具。

Spring Cloud比较成熟的两个体系：

- Spring Cloud Netflix
- Spring Cloud Alibaba

### 4.3 Spring Cloud核心组件

- Spring Cloud Netflix
  - Eureka 服务注册与发现中心，用于服务治理
  - Ribbon 服务访问组件、进行服务调用，实现了负载均衡
  - Hystrix 熔断器，服务容错管理
  - Feign 服务访问组件（对Ribbon和Hystrix的封装）
  - zuul 网关组件
- Spring Cloud Config 配置管理的组件—分布式配置中心
- Spring Cloud Bus 消息总线
- Spring Cloud Consul 服务注册与发现中心（功能类似eureka）

## 4.4 SpringCloud版本介绍

- SpringCloud版本 : A-H,2020.0.2
- SpringCloud的版本对SpringBoot版本时有依赖的
  - A ---- 1.2
  - B ---- 1.3
  - C ---- 1.4
  - D-E ---- 1.5
  - F-G-H ---- 2.x

# 五、搭建服务注册与发现中心

使用Spring Cloud Netflix 中的 Eureka 搭建服务注册与发现中心

## 5.1 创建SpringBoot应用，添加依赖

- spring web
- eureka server

## 5.2 配置服务注册与发现中心

```
## 设置服务注册与发现中心的端口
server:
  port: 8761

## 在微服务架构中，服务注册中心是通过服务应用的名称来区分每个服务的
## 我们在创建每个服务之后，指定当前服务的 应用名/项目名
spring:
  application:
    name: service-eureka

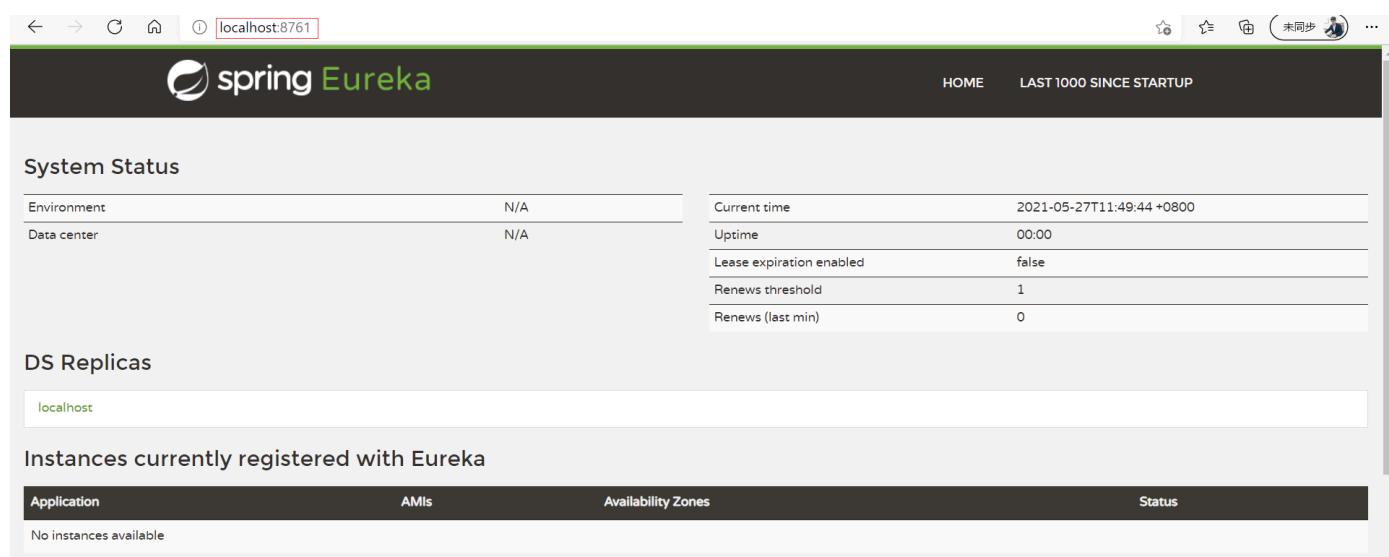
eureka:
  client:
    ## ip 就是服务注册中心服务器的ip
    ## port 就是服务注册与发现中心设置的port
    service-url:
      defaultZone: http://192.168.54.59:8761/eureka
  ## 设置服务注册与发现中心是否为为集群搭建（如果为集群模式，多个eureka节点之间需要相互注册）
  register-with-eureka: false
  ## 设置服务注册与发现中是否作为服务进行注册
  fetch-registry: false
```



## 5.3 在启动类添加 @EnableEurekaServer 注解

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

## 5.4 运行及访问



The screenshot shows a web browser window with the URL `localhost:8761` in the address bar. The page title is "spring Eureka". The top navigation bar includes links for "HOME" and "LAST 1000 SINCE STARTUP". Below the header, there's a section titled "System Status" with tables for Environment, Data center, Current time, Uptime, Lease expiration enabled, Renews threshold, and Renews (last min). Under "DS Replicas", there's a table for "Instances currently registered with Eureka" with columns for Application, AMIs, Availability Zones, and Status. A note says "No instances available".

# 六、服务注册

创建保存订单的服务（order-add）注册到服务注册与发现中心

## 6.1 创建SpringBoot应用

创建spring boot应用，完成功能开发

## 6.2 注册服务

将能够完成特定业务的SpringBoot应用作为服务提供者，注册到服务注册与发现中心

## 6.2.1 添加依赖

- eureka-server [ 注意版本! ]

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



## 6.2.2 配置application.yml

```
## 当前服务的port
server:
  port: 9001

## 当前应用名会作为服务唯一标识注册到eureka
spring:
  application:
    name: order-add
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db_2010_sc?characterEncoding=utf-8
    username: root
    password: admin123

  mybatis:
    mapper-locations: classpath:mappers/*
    type-aliases-package: com.qfedu.order.beans

## 配置Eureka服务注册与发现中心的地址
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka
```



## 6.2.3 在当前服务应用的启动类添加 **@EnableEurekaClient** 注解

```
@SpringBootApplication
@MapperScan("com.qfedu.order.dao")
@EnableEurekaClient
public class OrderAddApplication {

    public static void main(String[] args) {
        SpringApplication.run(OrderAddApplication.class, args);
    }
}
```



# 七、服务发现-Ribbon

服务消费者（api-order-add）通过eureka查找服务提供者（order-add），通过服务调用组件调用提供者

- eureka server
- ribbon

## 7.1 基础配置

Ribbon客户端已经停更进维啦

### 7.1.1 创建SpringBoot应用，添加依赖

- eureka server
- ribbon

### 7.1.2 配置application.yml

```
server:  
  port: 8001  
spring:  
  application:  
    name: api-order-add  
eureka:  
  client:  
    service-url:  
      defaultZone: http://localhost:8761/eureka
```



### 7.1.3 在启动类添加 @EnableDiscoveryClient 注解

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



## 7.2 服务调用

### 7.2.1 配置RestTemplate

```
@Configuration
public class AppConfig {

    @LoadBalanced //启用Ribbon (负载均衡)
    @Bean
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }

}
```

## 7.2.2 在Service中注入RestTemplate对象调用服务

```
@Service
public class OrderAddServiceImpl implements OrderAddService {

    @Autowired
    private RestTemplate restTemplate;

    @Override
    public ResultVO saveOrder(Order order) {
        //1. 调用 order-add服务进行保存
        ResultVO vo = restTemplate.postForObject("http://order-add/order/add", order,
ResultVO.class);

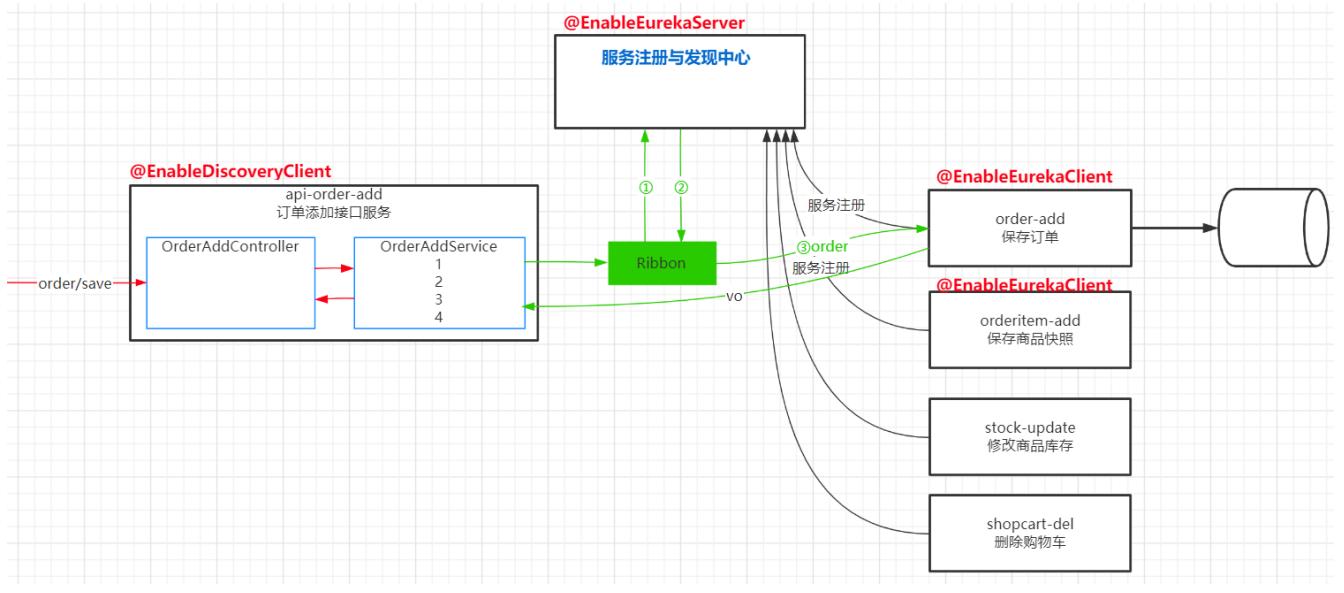
        //2. 调用 orderitem-add 保存订单快照

        //3. 调用 stock-update 修改商品库存

        //4. 调用 shopcart-del 删除购物车记录
        return null;
    }
}
```

## 7.3 案例流程图

## 服务注册与发现-案例流程图



## 7.4 Ribbon服务调用说明

@LoadBalanced注解是Ribbon的入口，在RestTemplate对象上添加此注解之后，再使用RestTemplate发送REST请求的时候，就可以通过Ribbon根据服务名称从Eureka中查找服务对应的访问地址列表，再根据负载均衡策略（默认轮询）选择其中的一个，然后完成服务的调用

- 获取服务列表
- 根据负载均衡策略选择服务
- 完成服务调用

# 八、基于Ribbon进行服务调用的参数传递

## 8.1 RestTemplate发送调用请求的方法

SpringCloud的服务调用是基于REST的，因此当服务提供者规定了请求的方式，服务消费者必须发送对应方式的请求才能完成服务的调用，RestTemplate提供了多个方法用于发送不同形式的请求

```
//post方式请求  
restTemplate.postForObject();  
//get方式请求  
restTemplate.getForObject();  
//delete方式请求  
restTemplate.delete();  
//put方式请求  
restTemplate.put();
```

## 8.2 put/post请求传参

- 服务消费者请求传参

```
//参数1：访问服务的url  
//参数2：传递的对象参数  
//参数3：指定服务提供者返回的数据类型  
ResultVO vo = restTemplate.postForObject("http://order-add/order/add", order, ResultVO.class);
```



- 服务提供者接收参数

```
@PostMapping("/add")  
public ResultVO addOrder(@RequestBody Order order){  
    return orderService.saveOrder(order);  
}
```



## 8.3 get请求传参

- 服务消费者请求传参

```
String userId = order.getUserId();  
ResultVO vo = restTemplate.getForObject("http://order-add/order/add?userId=" + userId,  
ResultVO.class);
```



- 服务提供者接收参数

```
@GetMapping("/add")  
public ResultVO addOrder(Order order){  
    return orderService.saveOrder(order);  
}  
  
@GetMapping("/add")  
public ResultVO addOrder(String userId){  
    //return orderService.saveOrder(order);  
}
```



# 九、服务发现-Feign

## 9.1 基础配置

### 9.1.1 创建SpringBoot应用，添加依赖

- spring web
- eureka server
- OpenFeign

### 9.1.2 配置application.yml

```
server:  
  port: 8002  
spring:  
  application:  
    name: api-order-add-feign  
eureka:  
  client:  
    service-url:  
      defaultZone: http://localhost:8761/eureka
```



### 9.1.3 在启动类添加注解

```
@SpringBootApplication  
@EnableDiscoveryClient //声明为服务消费者  
@EnableFeignClients //声明启用feign客户端  
public class ApiOrderAddFeignApplication {  
  
  public static void main(String[] args) {  
    SpringApplication.run(ApiOrderAddFeignApplication.class, args);  
  }  
  
}
```



## 9.2 服务调用

使用Feign进行服务调用的时候，需要手动创建一个服务访问客户端（接口）

### 9.2.1 创建Feign客户端

```
@FeignClient("order-add")  
public interface OrderAddClient {  
  
  @PostMapping("order/add")  
  public ResultVO addOrder(Order order);  
  
}
```



### 9.2.2 使用Feign客户端调用服务

```
@Service  
public class OrderAddServiceImpl implements OrderAddService {  
  
  @Autowired  
  private OrderAddClient orderAddClient;  
  
  @Override  
  public ResultVO saveOrder(Order order) {  
    //1. 调用 order-add服务进行保存  
    ResultVO vo = orderAddClient.addOrder(order);  
  }
```



```
//2. 调用 orderitem-add 保存订单快照  
  
//3. 调用 stock-update 修改商品库存  
  
//4. 调用 shopcart-del 删除购物车记录  
return vo;  
}  
}
```

## 9.3 Feign传参

### 9.3.1 POST请求

- 通过请求体传递对象

- 服务提供者

```
@PostMapping("/add")  
public ResultVO addOrder(@RequestBody Order order){  
    System.out.println("-----order-add");  
    System.out.println(order);  
    return orderService.saveOrder(order);  
}
```



- 服务消费者(Feign客户端)

```
@FeignClient("order-add")  
public interface OrderAddClient {  
  
    @PostMapping("order/add")  
    public ResultVO addOrder(Order order);  
  
}
```



- 通过请求行传参

- 服务提供者

```
@PostMapping("/add")  
public ResultVO addOrder(@RequestBody Order order, String str){  
    System.out.println("-----order-add");  
    System.out.println(order);  
    System.out.println(str);  
    return orderService.saveOrder(order);  
}
```



- 服务消费者 (Feign客户端)

```
//1.对用POST请求调用服务，Feign客户端的方法参数默认为body传值（body只能有一个值）
//2.如果有多个参数，则需要通过@RequestParam声明参数为请求行传值
@PostMapping("order/add")
public ResultVO addOrder(Order order, @RequestParam("str") String str);
```

### 9.3.2 Get请求

Get请求调用服务，只能通过url传参

在Feign客户端的方法中，如果不指定参数的传值方式，则默认为body传参，Get请求也不例外；因此对于get请求传递参数，必须通过@RequestParam注解声明

- 服务提供者

```
@GetMapping("/get")
public Order addOrder(String orderId){
    return new Order();
}
```

- 服务消费者（Feign客户端）

```
@GetMapping("order/get")
public Order getOrder(@RequestParam("orderId") String orderId);
```

## 十、服务注册与发现中心的可靠性和安全性

### 10.1 可靠性

在微服务架构系统中，服务消费者是通过服务注册与发现中心发现服务、调用服务的，服务注册与发现中心服务器一旦挂掉，将会导致整个微服务架构系统的崩溃，如何保证Eureka的可靠性呢？

- 使用eureka集群

Eureka集群搭建

相互注册、相互发现

```
## 设置服务注册与发现中心的端口
server:
  port: 8761

## 在微服务架构中，服务注册中心是通过服务应用的名称来区分每个服务的
## 我们在创建每个服务之后，指定当前服务的 应用名/项目名
spring:
  application:
    name: service-eureka

eureka:
```

```
client:  
    ## 设置服务注册与发现中心是否为集群搭建  
    register-with-eureka: true  
    ## 设置服务注册与发现中是否作为服务进行注册  
    fetch-registry: true  
    ## ip 就是服务注册中心服务器的ip  
    ## port 就是服务注册与发现中心设置的port  
    service-url:  
        defaultZone: http://192.168.54.10:8761/eureka
```

## 10.2 安全性

当完成Eureka的搭建之后，只要知道ip和port就可以随意的注册服务、调用服务，这是不安全的，我们可以通过设置帐号和密码来限制服务的注册及发现。

- 在eureka中整合Spring Security安全框架实现帐号和密码验证

### 10.2.1 添加SpringSecurity的依赖

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



### 10.2.2 设置访问eureka的帐号和密码

```
spring:  
    security:  
        user:  
            name: zhangsan  
            password: 123456
```



### 10.2.3 配置Spring Security

```
@Configuration  
@EnableWebSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
  
    @Override  
    protected void configure(HttpSecurity http) throws Exception {  
        http.csrf().disable();  
        //设置当前服务器的所有请求都要使用spring security的认证  
        http.authorizeRequests().anyRequest().authenticated().and().httpBasic();  
    }  
}
```



#### 10.2.4 服务提供者和服务消费者连接到注册中心都要帐号和密码

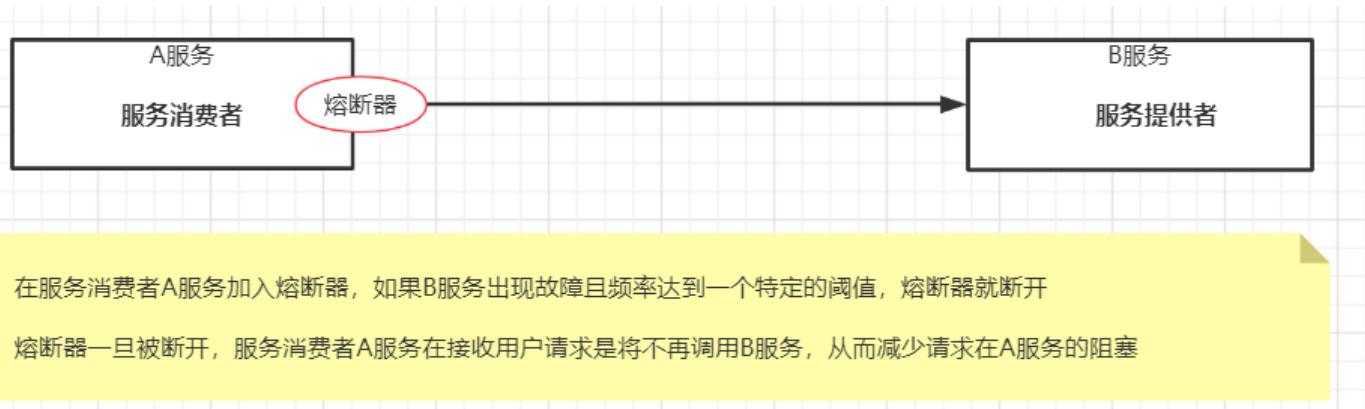
```
eureka:  
  client:  
    service-url:  
      defaultZone: http://zhangsan:123456@localhost:8761/eureka
```

## 十一、熔断器-Hystrix

服务故障的雪崩效应：当A服务调用B服务时，由于B服务的故障导致A服务处于阻塞状态，大量的请求可能会导致A服务因资源耗尽而出现故障。

为了解决服务故障的雪崩效应，出现了熔断器模型。

### 11.1 熔断器介绍



熔断器作用：

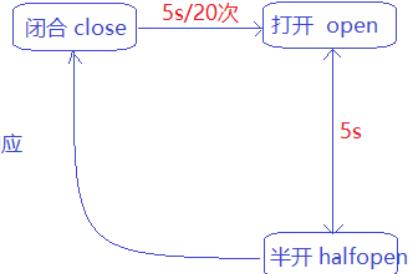
- **服务降级**：用户请求A服务，A服务调用B服务，当B服务出现故障或者在特定的时间段内不能给A服务响应，为了避免A服务因等待B服务而产生阻塞，A服务就不等B服务的结果了，直接给用户一个降级响应
- **服务熔断**：用户请求A服务，A服务调用B服务，当B服务出现故障的频率过高达到特定阈值（5s 20次）时，当用户再请求A服务时，A服务将不再调用B服务，直接给用户一个降级响应

### 11.2 熔断器的原理



#### 熔断器的状态

1. 熔断器默认为闭合 (close) 状态, 当用户请求A服务时, A服务调用B服务, 如果B服务再设定的时间不能给A服务响应, A服务则使用降级方案响应, 同时记录B服务的故障
2. 当B服务的故障率达到阈值 (Hystrix默认 5s/20次), 熔断器就会被断开进入`打开`(open)状态
3. 当熔断器为`打开`(open)状态时, 用户请求A服务, A服务不再调用B服务, 而是之间进行降级响应
4. 状态`打开`(open)状态的熔断器经过一个时间周期后会进入`半开`(half open)状态
5. 当容器为`半开`(half open)状态时, 当用户请求A服务时, A服务会对B服务进行一次调用  
如果B服务成功响应A服务, 熔断器则进入闭合 (close) 状态;  
如果B服务响应失败, 熔断器则回到`打开`(open)状态, 再进入一个周期的熔断。



## 11.3 基于Ribbon服务调用的熔断器使用

### 11.3.1 服务消费者的 服务降级

- 添加熔断器依赖 hystrix

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



- 在启动类添加 @EnableHystrix 注解

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableHystrix
public class ApiOrderAddApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiOrderAddApplication.class, args);
    }
}
```



- 在调用服务提供者的业务处理方法中, 进行降级配置

```
@Service
public class OrderAddServiceImpl implements OrderAddService {

    @Autowired
    private RestTemplate restTemplate;

    @HystrixCommand(fallbackMethod = "fallbackSaveOrder", commandProperties = {
        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="3000")
    })
    public ResultVO saveOrder(Order order) {
        //1. 调用 order-add服务进行保存
        //参数1: 访问服务的url
    }
}
```



```

    //参数2：传递的对象参数
    //参数3：指定服务提供者返回的数据类型
    ResultVO vo =
        restTemplate.postForObject("http://order-add/order/add", order, ResultVO.class);
    System.out.println(vo);
    //2. 调用 orderitem-add 保存订单快照
    //3. 调用 stock-update 修改商品库存
    //4. 调用 shopcart-del 删除购物车记录
    return vo;
}

/**
 * 降级方法：与业务方法拥有相同的参数和返回值
 * @return
 */
public ResultVO fallbackSaveOrder(Order order){
    return ResultVO.fail("网络异常, 请重试!", null);
}

}

```

### 11.3.2 服务提供者的 服务降级

- 配置步骤一致
- 服务提供者接口降级



```

@RestController
@RequestMapping("/order")
public class OrderController {

    @Autowired
    private OrderService orderService;

    @HystrixCommand(fallbackMethod = "fallbackAddOrder", commandProperties = {
        @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="3000")
    })
    @PostMapping("/add")
    public ResultVO addOrder(@RequestBody Order order){
        System.out.println("-----order-add");
        System.out.println(order);
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        return orderService.saveOrder(order);
    }

    public ResultVO fallbackAddOrder(@RequestBody Order order){
        System.out.println("-----order-add--fallback");
        return ResultVO.fail("订单保存失败!", null);
    }
}

```

```
}
```

```
}
```

### 11.3.3 服务熔断配置

- 熔断器状态：闭合、打开、半开
- 服务熔断配置

```
@HystrixCommand(fallbackMethod = "fallbackSaveOrder", commandProperties = {  
    @HystrixProperty(name="execution.isolation.thread.timeoutInMilliseconds", value="3000"),  
    @HystrixProperty(name="circuitBreaker.enabled", value="true"), //启用服务熔断  
    @HystrixProperty(name="circuitBreaker.sleepWindowInMilliseconds", value="10000"), //时间  
    @HystrixProperty(name="circuitBreaker.requestVolumeThreshold", value="10"), //请求次数  
    @HystrixProperty(name="circuitBreaker.errorThresholdPercentage", value="50") //服务错误率  
})  
public ResultVO saveOrder(Order order) {  
    //1. 调用 order-add服务进行保存  
    ResultVO vo = restTemplate.postForObject("http://order-add/order/add", order,  
    ResultVO.class);  
    System.out.println(vo);  
    //2. 调用 orderitem-add 保存订单快照  
    //3. 调用 stock-update 修改商品库存  
    //4. 调用 shopcart-del 删除购物车记录  
    return vo;  
}  
  
/**  
 * 降级方法：与业务方法拥有相同的参数和返回值  
 * @return  
 */  
public ResultVO fallbackSaveOrder(Order order){  
    return ResultVO.fail("网络异常, 请重试!", null);  
}
```



服务熔断：当用户请求服务A，服务A调用服务B时，如果服务B的故障率达到特定的阈值时，熔断器就会被打开一个时间周期（默认5s，可自定义），在这个时间周期内如果用户请求服务A，服务A将不再调用服务B，而是直接响应降级服务。

## 11.4 基于Feign服务调用的熔断器使用

Feign是基于Ribbon和Hystrix的封装

### 11.4.1 Feign中的熔断器使用

- 添加依赖

SpringBoot 2.3.11 Spring Cloud H

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.11.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <java.version>1.8</java.version>
    <spring-cloud.version>Hoxton.SR11</spring-cloud.version>
</properties>
```

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

- 在application.yml启用熔断器机制

```
feign:
  hystrix:
    enabled: true
```

- 在启动类添加 @EnableHystrix

```
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
@EnableHystrix
public class ApiOrderAddFeignApplication {

    public static void main(String[] args) {
        SpringApplication.run(ApiOrderAddFeignApplication.class, args);
    }
}
```

- 创建服务降级处理类

FeignClient的服务降级类：1.必须实现Feign客户端接口，2.必须交给Spring容器管理

```
@Component
public class OrderAddClientFallback implements OrderAddClient {
    public ResultVO addOrder(Order order, String str) {
        System.out.println("-----addOrder的降级服务");
        return ResultVO.fail("fail", null);
    }

    public Order getOrder(String orderId) {
        System.out.println("-----getOrder的降级服务");
        return new Order();
    }
}
```

- 在Feign客户端指定降级处理类

```
@FeignClient(value = "order-add", fallback = OrderAddClientFallback.class)
public interface OrderAddClient {

    //1. 对用POST请求调用服务，Feign客户端的方法参数默认为body传值（body只能有一个值）
    //2. 如果有多个参数，则需要通过@RequestParam声明参数为请求行传值
    @PostMapping("order/add")
    public ResultVO addOrder(Order order, @RequestParam("str") String str);

    @GetMapping("order/get")
    public Order getOrder(@RequestParam("orderId") String orderId);

}
```

- Service类通过Feign客户端调用服务

```
@Service
public class OrderAddServiceImpl implements OrderAddService {

    @Autowired
    private OrderAddClient orderAddClient;
    //当我们创建Feign客户端的降级类并交给Spring管理后 在Spring容器中就会出现两个OrderAddClient对象

    @Override
    public ResultVO saveOrder(Order order) {
        //1. 调用 order-add服务进行保存
        ResultVO vo = orderAddClient.addOrder(order, "测试字符串");
        Order order1 = orderAddClient.getOrder("订单编号");
        System.out.println(order1);
        //2. 调用 orderitem-add 保存订单快照
        //3. 调用 stock-update 修改商品库存
        //4. 调用 shopcart-del 删除购物车记录
        return vo;
    }
}
```

## 11.4.2 Ribbon 参数配置



## 11.5 熔断器仪表盘监控

如果服务器的并发压力过大、服务器无法正常响应，则熔断器状态变为open属于正常的情况；但是如果一个熔断器一直处于open状态、或者说服务器提供者没有访问压力的情况下熔断器依然为open状态，说明熔断器的状态就不属于正常情况了。如何了解熔断器的工作状态呢？

- 熔断器仪表盘

### 11.5.1 搭建熔断器仪表盘

- 创建SpringBoot项目，添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-hystrix-dashboard</artifactId>
</dependency>
```

- 配置仪表盘的port和appName

```
server:
  port: 9999
spring:
  application:
    name: hystrix-dashboard

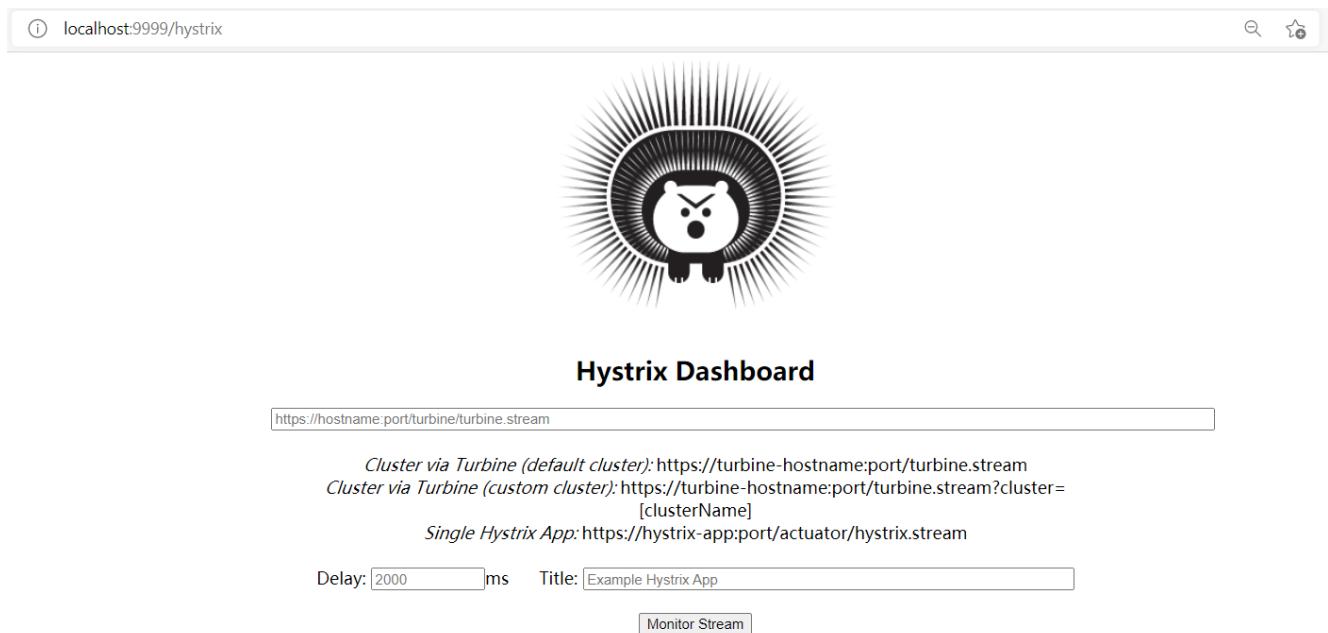
hystrix:
  dashboard:
    proxy-stream-allow-list: "localhost"
```

- 启动类添加 `@EnableHystrixDashboard` 注解

```
@SpringBootApplication
@EnableHystrixDashboard
public class HystrixDashboardApplication {

    public static void main(String[] args) {
        SpringApplication.run(HystrixDashboardApplication.class, args);
    }
}
```

- 访问 <http://localhost:9999/hystrix>



### 11.5.2 配置使用了熔断器的服务可被监控

- 添加依赖

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



- 配置

```
@Configuration
public class DashBoardConfig {

    @Bean
    public ServletRegistrationBean getServletRegistrationBean(){
        HystrixMetricsStreamServlet streamServlet = new HystrixMetricsStreamServlet();
        ServletRegistrationBean registrationBean =
            new ServletRegistrationBean(streamServlet);
        registrationBean.setName("HystrixMetricsStreamServlet");
        registrationBean.setLoadOnStartup(1);
        registrationBean.addUrlMappings("/hystrix.stream");
        return registrationBean;
    }

}
```



- 查看指定服务的熔断器工作参数

① localhost:9999/hystrix

The Hystrix Dashboard interface includes a central logo and a title "Hystrix Dashboard". Below the title is a search bar containing "http://localhost:8002/hystrix.stream". A red box highlights this URL. Below the search bar are three configuration options: "Cluster via Turbine (default cluster): https://turbine-hostname:port/turbine.stream", "Cluster via Turbine (custom cluster): https://turbine-hostname:port/turbine.stream?cluster=[clusterName]", and "Single Hystrix App: https://hystrix-app:port/actuator/hystrix.stream". A "Delay" input field set to "2000 ms" and a "Title" input field set to "Example Hystrix App" are also present. A "Monitor Stream" button is at the bottom right.

**Hybris Monitor**

Hybris Stream: http://localhost:8002/hystrix.stream

**Circuit** Sort: Error then Volume | Alphabetical | Volume | Error | Mean | Median | 90 | 99 | 99.5 Success | Short-Circuited | Bad Request | Timeout | Rejected | Failure | Error %

Order	Method	Call Type	Success	Failure	Volume	Mean	Median	90th	99th	99.5th	99.9th	99.99th
1	Order#dOrder(Order, String)	Host	1.0/s	0.0 %	12	0	0	0	0	0	0	0
2	OrderAddClient#addOrder(String)	Host	1.1/s	0.0 %	12	0	0	0	0	0	0	0

Hosts: 1  
Median: 1006ms  
Mean: 627ms

Hosts: 1  
Median: 1ms  
Mean: 1ms

**Thread Pools** Sort: Alphabetical | Volume |

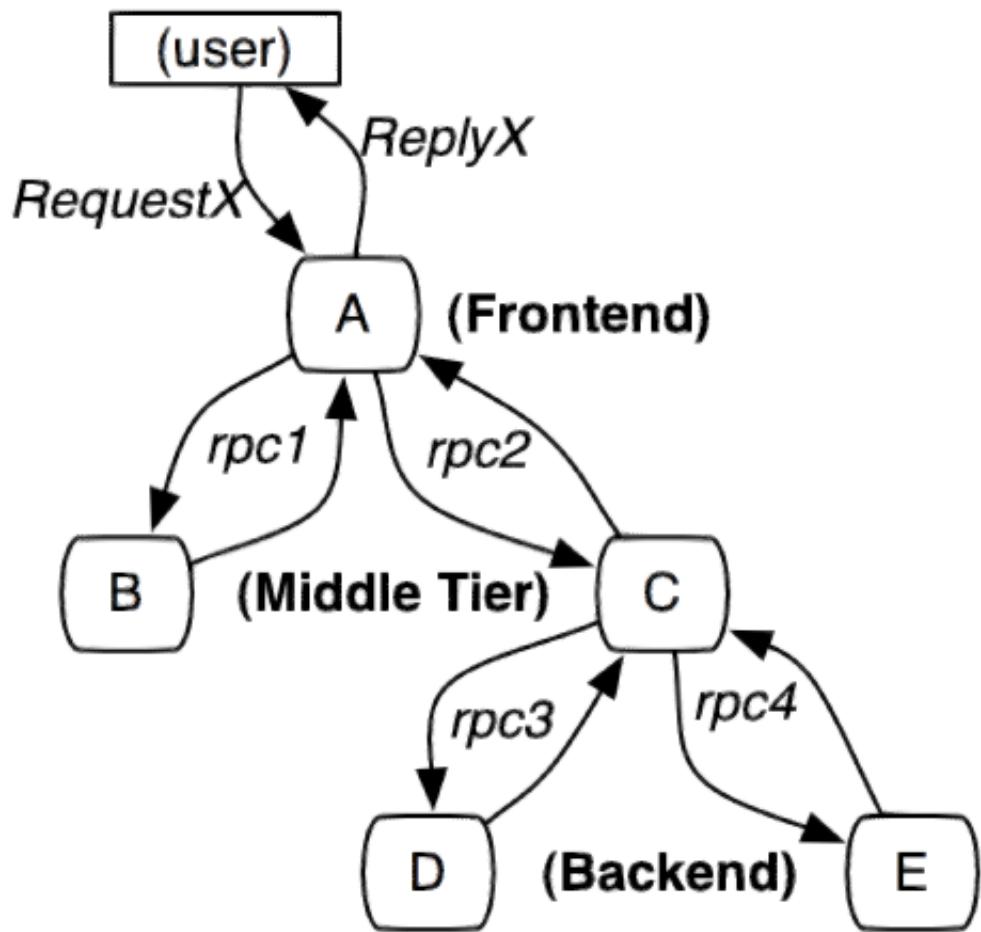
Pool	Active	Queued	Max Active	Queue Size
order-add	0	0	1	5
order-add	2.4/s	2.4/s	2.4/s	2.4/s

## 十二、《锋迷商城》微服务拆分

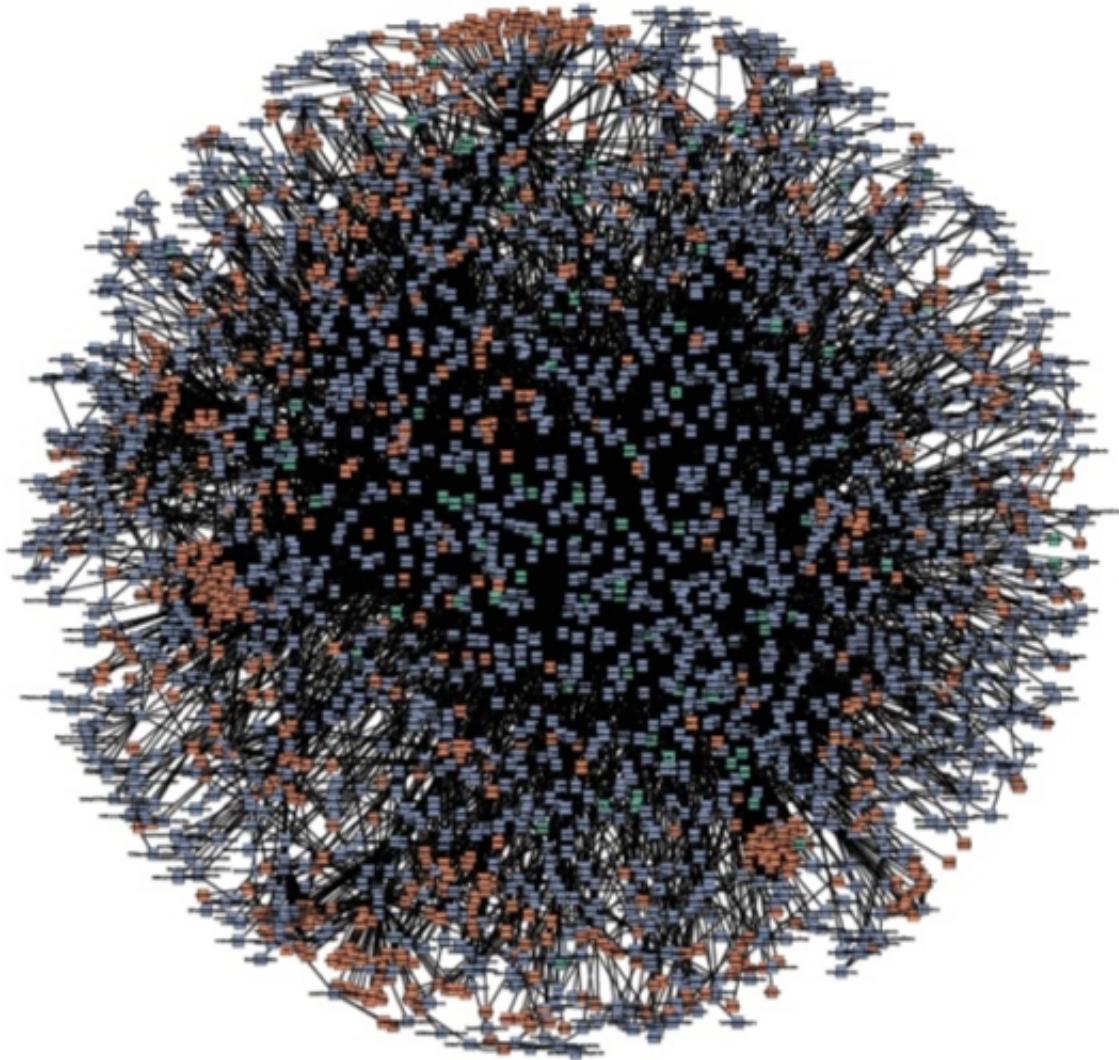
## 十三、服务链路追踪

### 13.1 服务追踪说明

- 微服务架构是通过业务来划分服务的，使用REST调用。对外暴露的一个接口，可能需要很多个服务协同才能完成这个接口功能，如果链路上任何一个服务出现问题或者网络超时，都会形成导致接口调用失败。



- 随着业务的不断扩张，服务之间互相调用会越来越复杂，它们之间的调用关系也许如下：



- 随着服务的越来越多，对调用链的分析会越来越复杂。

## 13.2 Zipkin

- ZipKin是一个开放源代码的分布式跟踪系统，由Twitter公司开源，它致力于收集服务的定时数据，以解决微服务架构中的延迟问题，包括数据的收集、存储、查找和展现。它的理论模型来自于Google Dapper论文。
- 每个服务向 ZipKin 报告计时数据，ZipKin 会根据调用关系通过 ZipKin UI 生成依赖关系图，显示了多少跟踪请求通过每个服务，该系统让开发者可通过一个 Web 前端轻松的收集和分析数据，例如用户每次请求服务的处理时间等，可方便的监测系统中存在的瓶颈。

## 13.3 搭建zipkin服务器

- 创建SpringBoot项目（版本2.1.x）
- 添加依赖

```
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-server</artifactId>
    <version>2.11.10</version>
</dependency>
<!--zipkin界面-->
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</artifactId>
    <version>2.11.10</version>
</dependency>
```

- 在启动类添加 `@EnableZipkinServer` 注解

```
@SpringBootApplication
@EnableZipkinServer
public class ZipkinApplication {

    public static void main(String[] args) {
        SpringApplication.run(ZipkinApplication.class, args);
    }

}
```

- 配置yml

```
spring:
  application:
    name: zipkin

  server:
    port: 9411
  management:
    endpoints.web.exposure.include: '*'
    metrics.web.server.auto-time-requests: false
```

## 13.4 服务中Sleuth配置

- 在服务应用中添加Sleuth依赖

```
<!-- spring-cloud-sleuth-zipkin -->
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-zipkin</artifactId>
    <version>2.0.2.RELEASE</version>
</dependency>
```

- 在服务应用中配置yml

```
spring:
  application:
    name: goods-provider
  zipkin:
    enabled: true
    base-url: http://localhost:9411
  sleuth:
    sampler:
      probability: 0.1
```

## 13.5 zipkin服务数据存储

- 创建数据库数据表

```
CREATE TABLE IF NOT EXISTS zipkin_spans (
  `trace_id` BIGINT NOT NULL,
  `id` BIGINT NOT NULL,
  `name` VARCHAR(255) NOT NULL,
  `parent_id` BIGINT,
  `debug` BIT(1),
  `start_ts` BIGINT COMMENT "Span.timestamp(): epoch micros used for endTs query and to
implement TTL",
  `duration` BIGINT COMMENT "Span.duration(): micros used for minDuration and maxDuration
query"
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED;

ALTER TABLE zipkin_spans ADD UNIQUE KEY(`trace_id`, `id`) COMMENT "ignore insert on
duplicate";
ALTER TABLE zipkin_spans ADD INDEX(`trace_id`, `id`) COMMENT "for joining with
zipkin_annotations";
ALTER TABLE zipkin_spans ADD INDEX(`trace_id`) COMMENT "for getTracesByIds";
ALTER TABLE zipkin_spans ADD INDEX(`name`) COMMENT "for getTraces and getSpanNames";
ALTER TABLE zipkin_spans ADD INDEX(`start_ts`) COMMENT "for getTraces ordering and range";

CREATE TABLE IF NOT EXISTS zipkin_annotations (
  `trace_id` BIGINT NOT NULL COMMENT "coincides with zipkin_spans.trace_id",
  `span_id` BIGINT NOT NULL COMMENT "coincides with zipkin_spans.id",
  `a_key` VARCHAR(255) NOT NULL COMMENT "BinaryAnnotation.key or Annotation.value if type
=-1",
  `a_value` BLOB COMMENT "BinaryAnnotation.value(), which must be smaller than 64KB",
  `a_type` INT NOT NULL COMMENT "BinaryAnnotation.type() or -1 if Annotation",
  `a_timestamp` BIGINT COMMENT "Used to implement TTL; Annotation.timestamp or
zipkin_spans.timestamp",
  `endpoint_ipv4` INT COMMENT "Null when Binary/Annotation.endpoint is null",
  `endpoint_ipv6` BINARY(16) COMMENT "Null when Binary/Annotation.endpoint is null, or no
IPv6 address",
  `endpoint_port` SMALLINT COMMENT "Null when Binary/Annotation.endpoint is null",
  `endpoint_service_name` VARCHAR(255) COMMENT "Null when Binary/Annotation.endpoint is
null"
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED;
```

```

ALTER TABLE zipkin_annotations ADD UNIQUE KEY(`trace_id`, `span_id`, `a_key`,
`a_timestamp`) COMMENT "Ignore insert on duplicate";
ALTER TABLE zipkin_annotations ADD INDEX(`trace_id`, `span_id`) COMMENT "for joining with
zipkin_spans";
ALTER TABLE zipkin_annotations ADD INDEX(`trace_id`) COMMENT "for getTraces/ByIds";
ALTER TABLE zipkin_annotations ADD INDEX(`endpoint_service_name`) COMMENT "for getTraces
and getServiceNames";
ALTER TABLE zipkin_annotations ADD INDEX(`a_type`) COMMENT "for getTraces";
ALTER TABLE zipkin_annotations ADD INDEX(`a_key`) COMMENT "for getTraces";

CREATE TABLE IF NOT EXISTS zipkin_dependencies (
    `day` DATE NOT NULL,
    `parent` VARCHAR(255) NOT NULL,
    `child` VARCHAR(255) NOT NULL,
    `call_count` BIGINT
) ENGINE=InnoDB ROW_FORMAT=COMPRESSED;

ALTER TABLE zipkin_dependencies ADD UNIQUE KEY(`day`, `parent`, `child`);

```

- pom依赖

```

<!-- zipkin-storage-mysql-v1 -->
<dependency>
    <groupId>io.zipkin.zipkin2</groupId>
    <artifactId>zipkin-storage-mysql-v1</artifactId>
    <version>2.11.12</version>
</dependency>

<!--mysql驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
</dependency>

```



- 配置yml

```

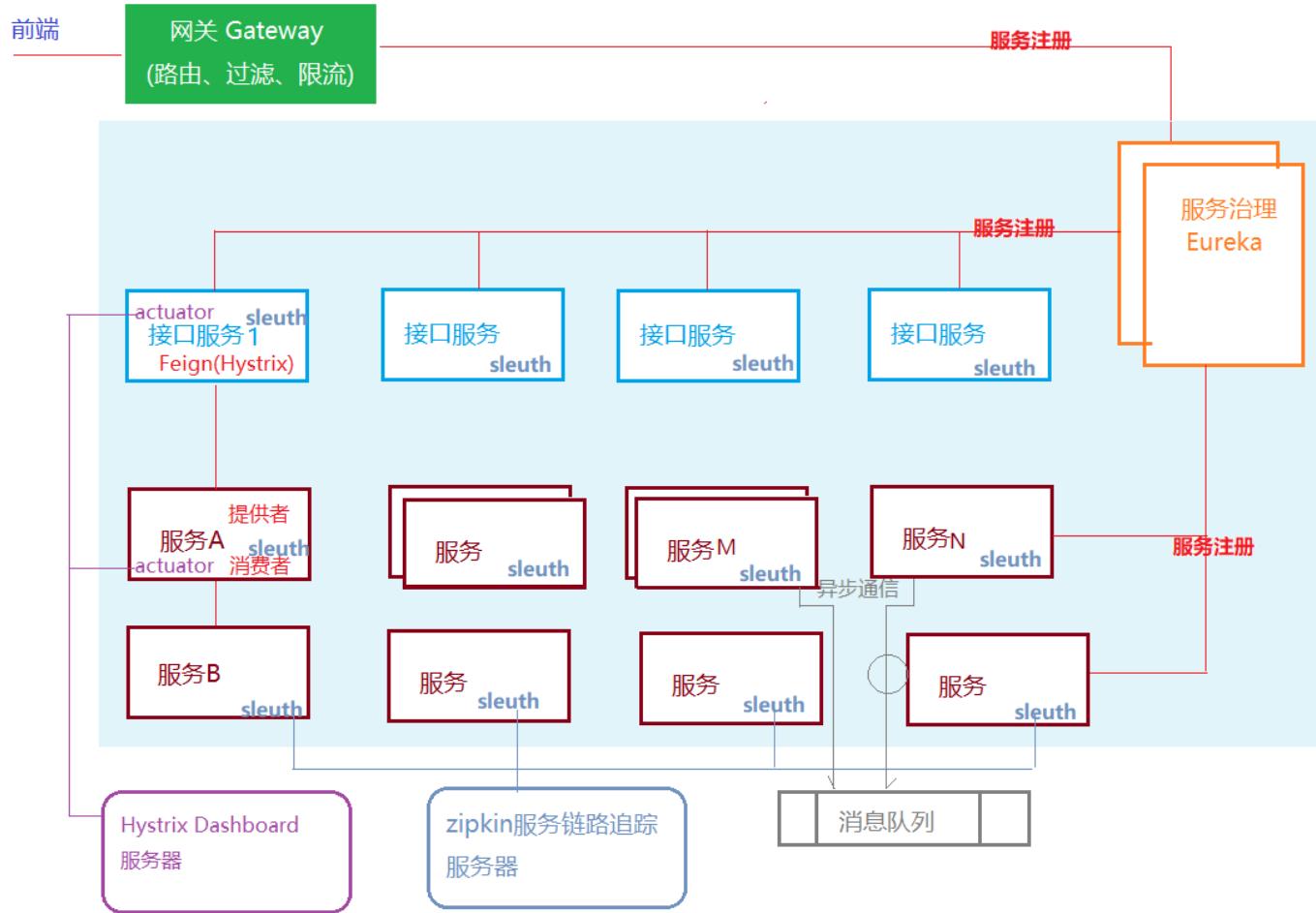
spring:
  application:
    name: zipkin
  datasource:
    username: root
    password: admin123
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/zipkin

zipkin:
  storage:
    type: mysql

```



## 十四、微服务架构总结

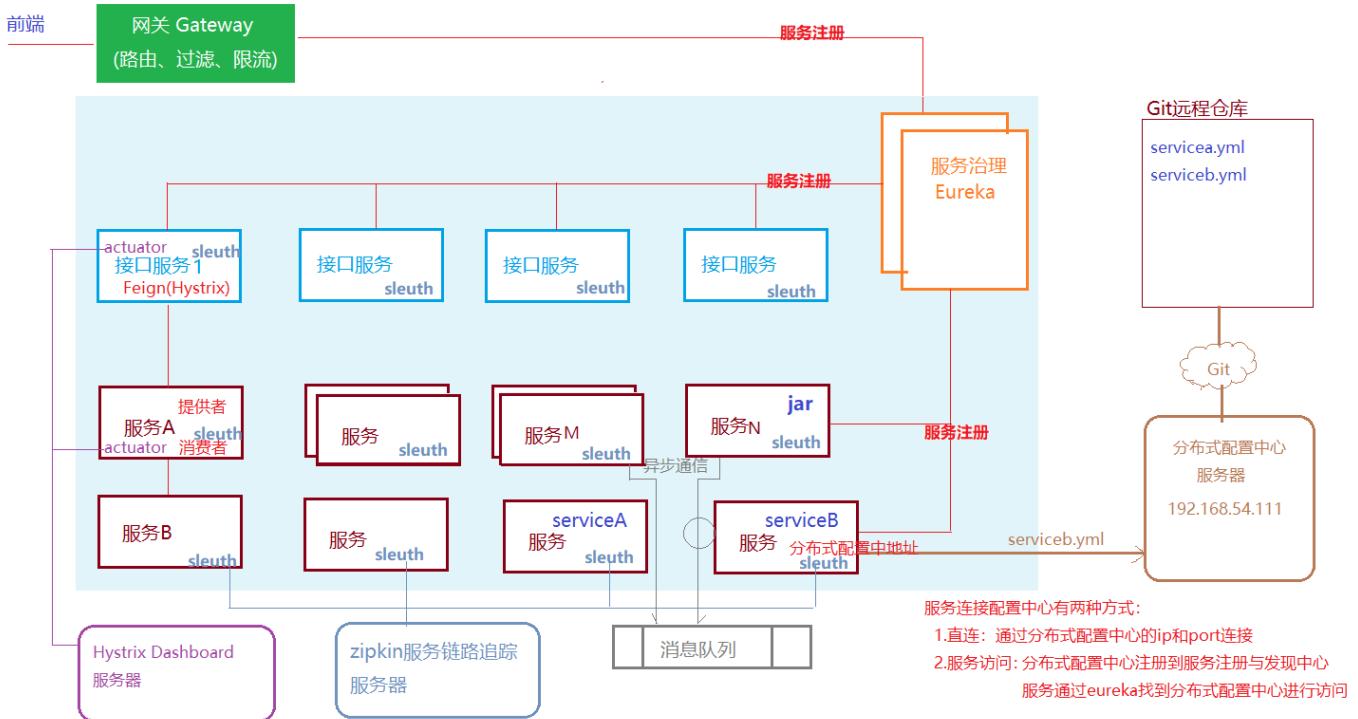


## 十五、分布式配置中心

在使用微服务架构开发的项目中，每个服务都有自己的配置文件（application.yml），如果将每个服务的配置文件直接写在对应的服务中，存在以下问题：

1. 服务开发完成之后，需要打包部署，配置文件也会打包在jar文件中，不便于项目部署之后的配置修改（在源码中修改——重新打包——重新上传——重新运行）
2. 微服务架构中服务很多，配置文件也很多，分散在不同服务中不便于配置的管理
3. 如果想要对服务进行集群部署，需要打包多个jar文件，上传，运行

### 15.1 分布式配置中心介绍



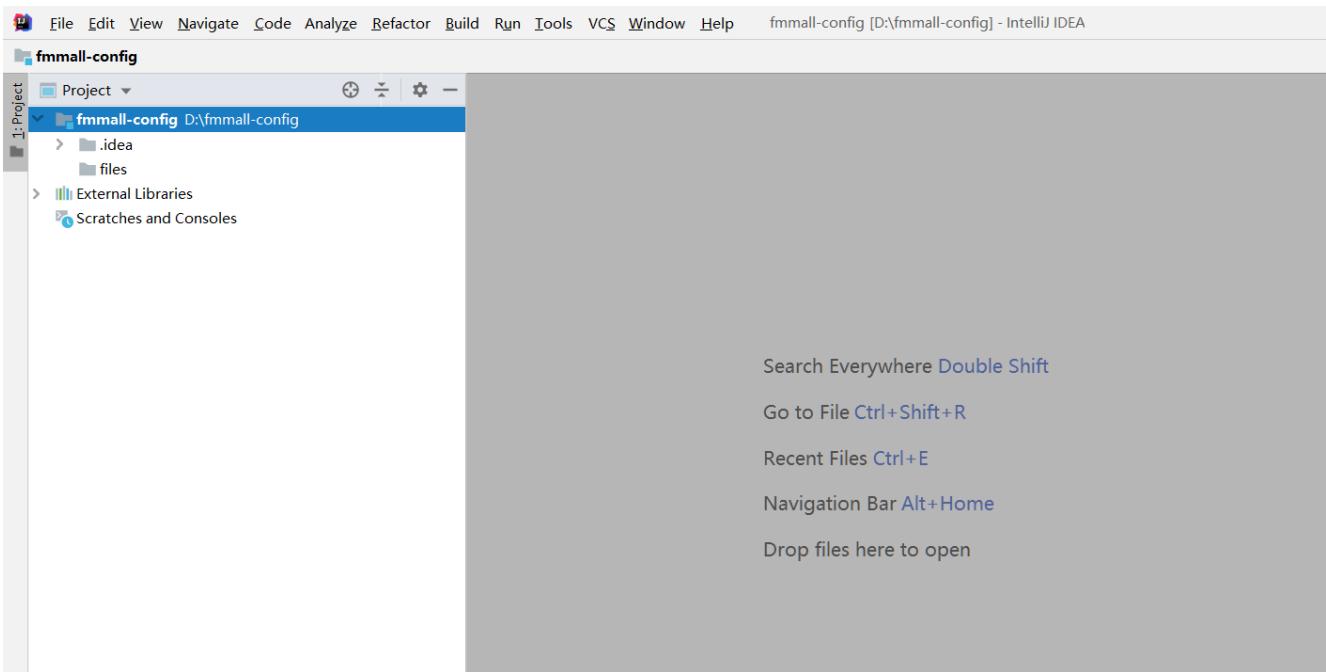
## 15.2 《锋迷商城》分布式配置中心搭建

步骤：

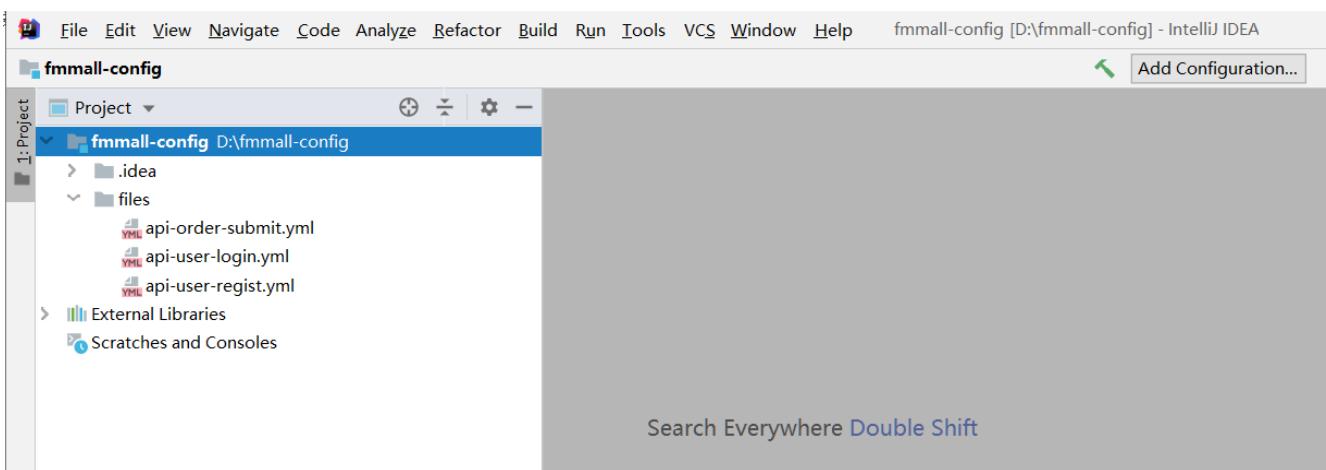
- 创建一个Git远程仓库，用来存放配置文件
- 搭建分布式配置中心服务器（Spring Cloud Config）Config server
  - 连接到配置文件的Git仓库
  - 注册到eureka
- 修改每个服务，删除application.yml中的所有配置，连接到分布式配置中心

### 15.2.1 创建Git远程仓库

- 创建远程仓库：<https://gitee.com/qfytao/fmmall-config.git>
  - 366274379 / admin123
- 在本地D盘创建 fmmall-config目录，作为本地存放配置文件的目录，在目录中创建files目录
- 使用idea打开 fmmall-config 目录



- 将《锋迷商城》项目中服务的配置文件拷贝粘贴到files目录，以服务的名称给配置文件命名

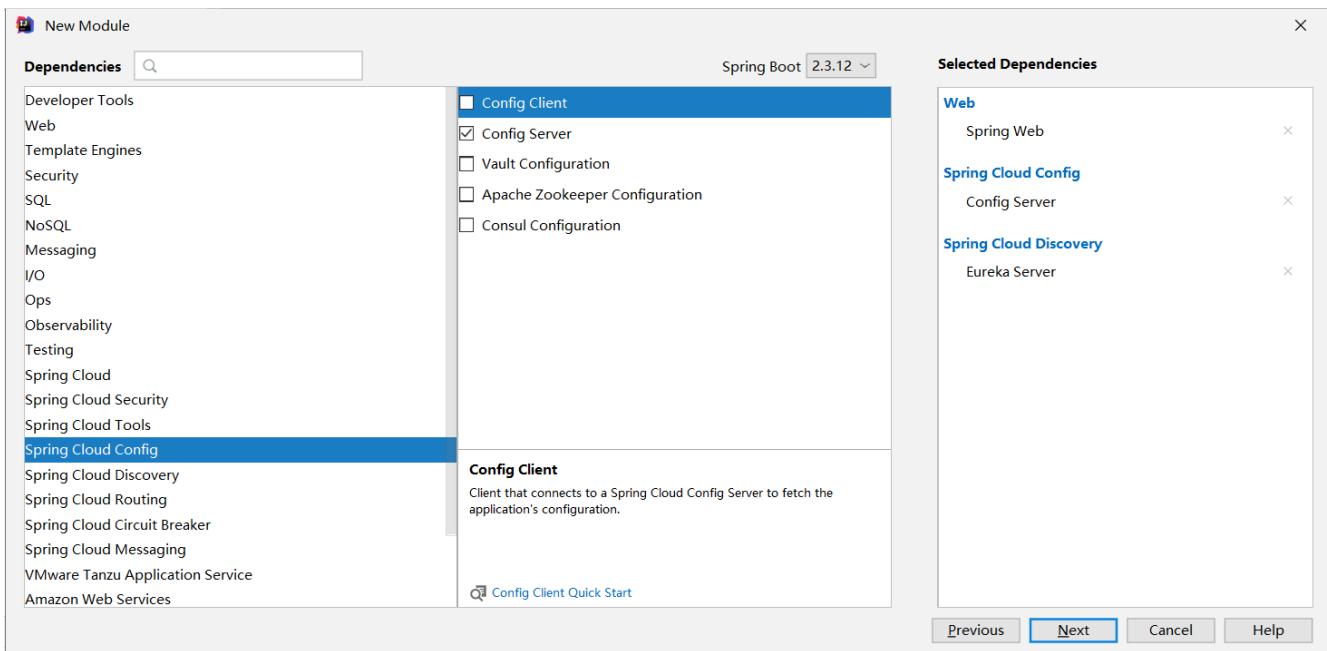


- 将 `fmmall-config` 创建成git仓库（本地仓库）
- 将本地仓库push到创建的git远程仓库

文件名	操作	时间
api-order-submit.yml	初始化	1分钟前
api-user-login.yml	初始化	1分钟前
api-user-regist.yml	初始化	1分钟前

## 15.2.2 搭建分布式配置中心服务器

- 创建SpringBoot应用，添加依赖



- 配置application.yml

```
server:  
  port: 8888  
spring:  
  application:  
    name: config-server  
  cloud:  
    config:  
      server:  
        git:  
          uri: https://gitee.com/qfytao/fmall-config.git  
          search-paths: files  
          username: 366274379@qq.com  
          password: admin123  
  
    eureka:  
      client:  
        service-url:  
          defaultZone: http://zhangsan:123456@localhost:8761/eureka
```



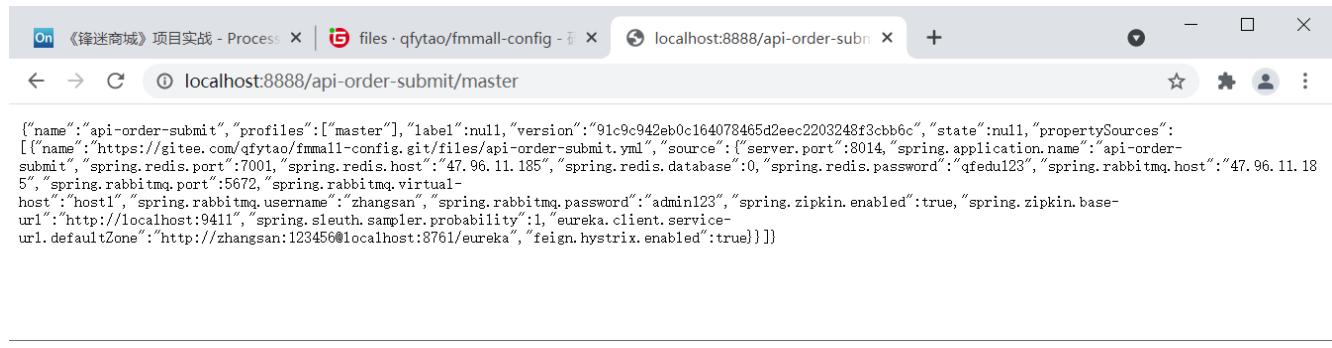
- 在启动类添加注解

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



- 运行测试

访问 <http://localhost:8888/api-order-submit/master>



```
{"name": "api-order-submit", "profiles": ["master"], "label": null, "version": "91c9c942eb0c164078465d2eec2203248f3cbb6c", "state": null, "propertySources": [{"name": "https://gitlab.com/qfytao/fmmall-config.git/files/api-order-submit.yaml", "source": {"server.port": "8014", "spring.application.name": "api-order-submit", "spring.redis.port": "7001", "spring.redis.host": "47.96.11.185", "spring.redis.database": "0", "spring.redis.password": "qfedul23", "spring.rabbitmq.host": "47.96.11.185", "spring.rabbitmq.port": "5672", "spring.rabbitmq.virtual-host": "host1", "spring.rabbitmq.username": "zhangsan", "spring.rabbitmq.password": "admin123", "spring.zipkin.enabled": true, "spring.zipkin.base-url": "http://localhost:9411", "spring.sleuth.sampler.probability": "1", "eureka.client.service-url.defaultZone": "http://zhangsan:123456@localhost:8761/eureka", "feign.hystrix.enabled": true}}]}
```

### 15.2.3 配置服务，通过分布式配置中心加载配置文件

- 添加依赖

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



- 配置服务的application.yml

```
spring:
  cloud:
    config:
      uri: http://localhost:8888
      name: api-order-submit
      label: master
```

