

Spring Cloud微服务开发笔记

一.分布式架构演变

1.传统企业级开发架构

- 单体应用
- SSH, SSM
- 分为三层架构 (Web层, 业务逻辑层, 数据访问层)

2.分布式架构

- 项目进行垂直划分 (模块划分)
- 以电商系统为例：订单系统，会员系统，支付，购物车...
- 优点：解耦，适合大团队
- 缺点：访问需要自行管理访问列表，成本非常高，实现方式是基于WebService，传统的RPC这样性能较差

3.SOA架构

- 面向服务架构，
- SOA传统的方式还是使用传输XML，后来用JSON取代XML
- 传统的SOA架构底层实现通过WebService和ESB（企业服务总线）

4.微服务架构

- SOA架构演变而来
- 解决了上述WebService,ESB总线架构的性能差的问题
- 引入一个服务的注册中心（中心化服务注册与发现）
- 不再使用XML传输数据，而是使用JSON来传输数据
- 将服务再进行一次精细化（细粒度服务组件），使用HTTP协议，使用RESTful风格开发模式

二.微服务如何拆分的问题

2.1 原则

- 如果某一个模块不能再细分，职责很单一，那么就可以成为微服务
- 每一个服务运行在一个独立进程
- 每一个服务有自己的一个数据库存储，缓存系统，消息队列...

2.2 微服务的原始文档

- <https://www.cnblogs.com/liuning8023/p/4493156.html>

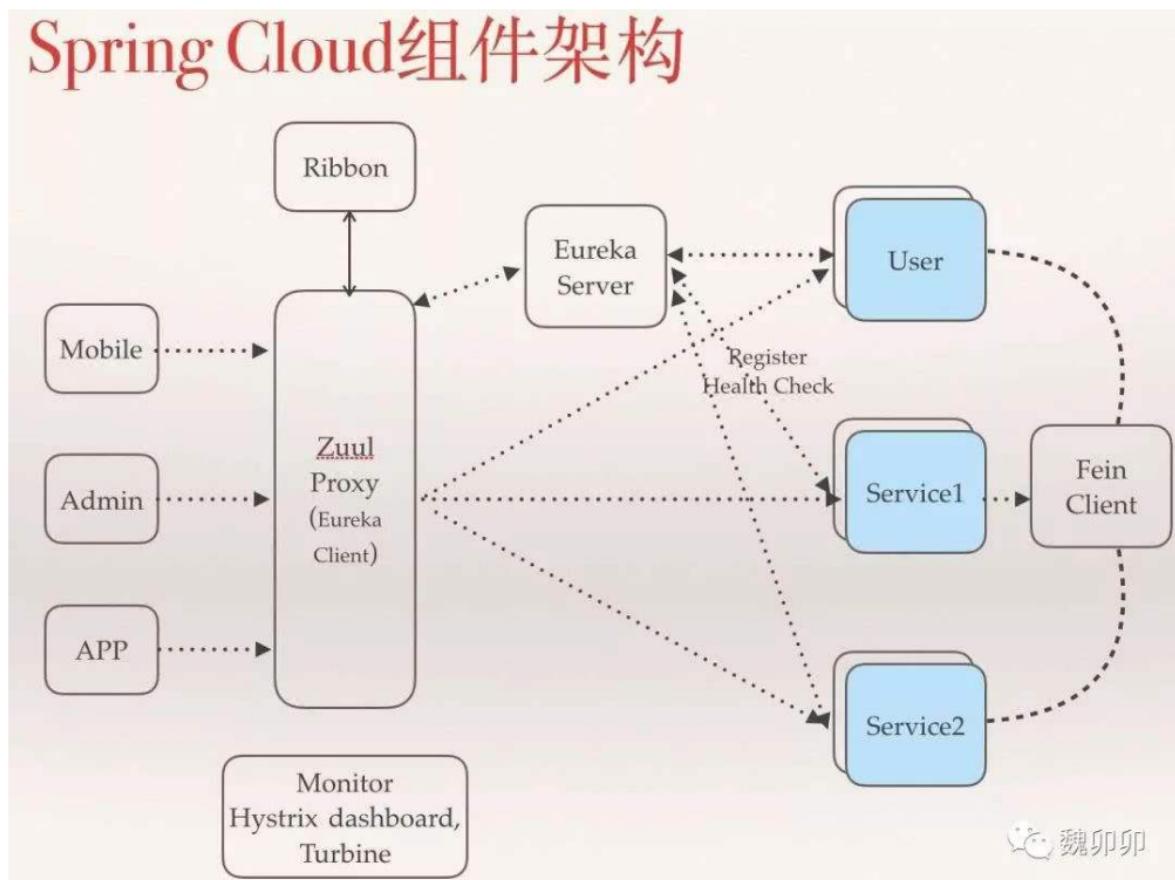
三.Spring Cloud微服务框架

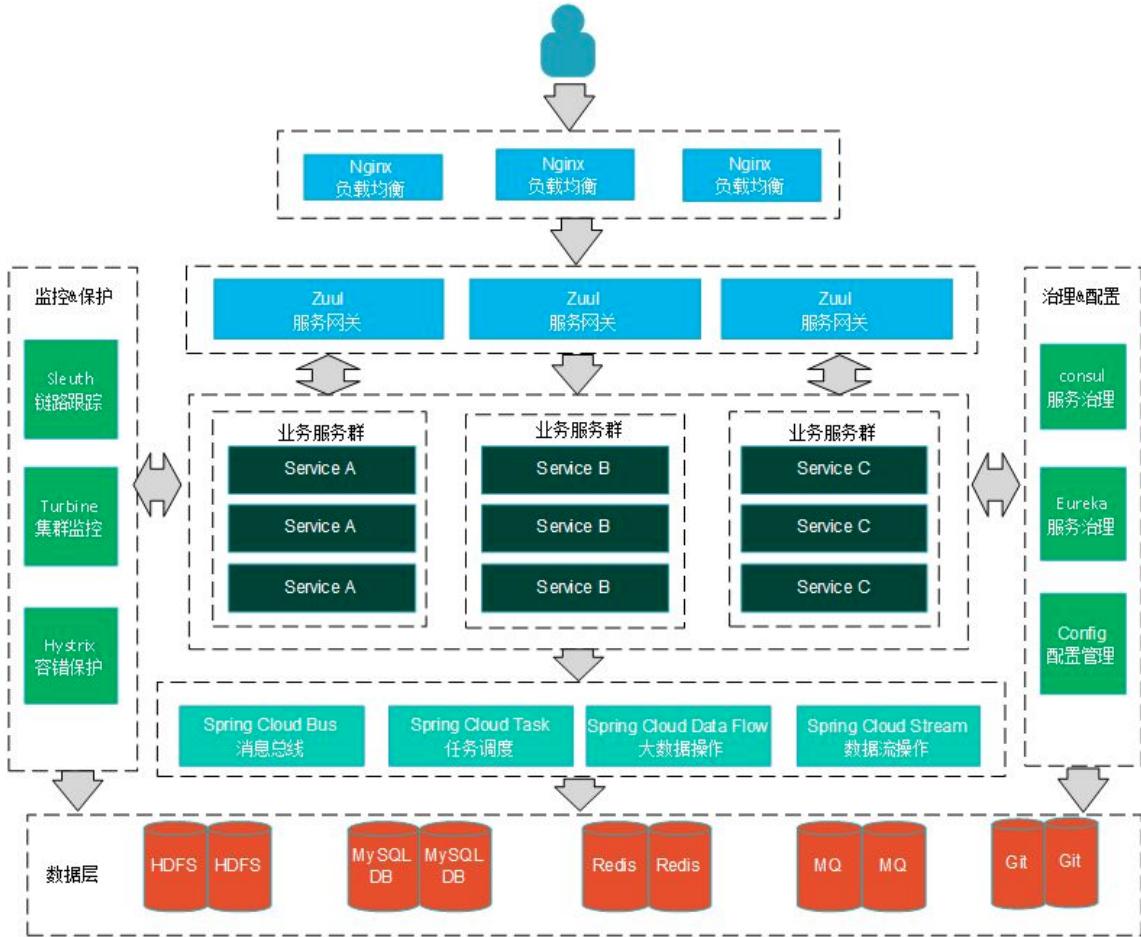
3.1 Spring Cloud vs 其他微服务架构

- Spring团队对整个JavaEE开发领导力。Spring Cloud提供了一套完整的微服务解决方案（服务的发现与注册，容错，网关，负载均衡，配置中心，本地调用机制...）
- 阿里的Dubbo，功能比较单一。Dubbo主要是做服务的发现与注册(Eureka,Consul)
- Spring Cloud有分布式相关解决方案
 - 分布式配置中心：携程阿波罗，Netflix Archaius...
 - 分布式调度中心：xxl-job,elastic-job, Spring Cloud的task
 - 服务跟踪：服务链路追踪(Spring Cloud Sleuth)

3.2 Spring Cloud概述

- Spring Boot基础之上开发的一套微服务框架。微服务完整解决方案。
- 服务治理、注册与发现、配置管理、跟踪管理、断路器，路由、负载均衡，控制总线，微代理...
- <https://spring.io/projects/spring-cloud>
- <https://springcloud.cc/>
- Spring Cloud几大组件（五大神兽）
 - Spring Cloud Config：分布式配置中心
 - Spring Cloud netflix
 - Eureka：服务的发现与注册
 - Hystrix：服务熔断，服务的保护
 - Ribbon：负载均衡
 - Feign：声明式服务调用组件
 - Zuul：网关组件，提供智能路由





四.服务治理Eureka

4.1 服务的治理

- 服务之间的依赖关系，管理成本很高，需要有一个中间件（注册中心）来管理依赖关系
- 实现服务的发现与注册，服务调用，负载均衡，容错

4.2 服务的发现与注册

- 服务的提供方注册服务（注册中心）。当服务器启动时服务注册到注册中心（Eureka Server）。
- Eureka Server将服务地址进行维护。以别名的方式注册到注册中心
- 服务的消费方在注册中心上获取到服务列表
- 以RPC（HttpClient）调用服务

4.3 搭建注册中心

- 编写pom.xml

```

1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>2.0.2.RELEASE</version>

```

```
5      <relativePath/> <!-- lookup parent from repository -->
6  </parent>
7
8
9  <dependencies>
10     <dependency>
11         <groupId>org.springframework.cloud</groupId>
12         <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
13     </dependency>
14
15     <dependency>
16         <groupId>org.springframework.boot</groupId>
17         <artifactId>spring-boot-starter-test</artifactId>
18         <scope>test</scope>
19     </dependency>
20 </dependencies>
21
22 <dependencyManagement>
23     <dependencies>
24         <dependency>
25             <groupId>org.springframework.cloud</groupId>
26             <artifactId>spring-cloud-dependencies</artifactId>
27             <version>Finchley.SR2</version>
28             <type>pom</type>
29             <scope>import</scope>
30         </dependency>
31     </dependencies>
32 </dependencyManagement>
```

- 编写application.properties/application.yml

```
1
2 server.port=8080
3 eureka.instance.hostname=127.0.0.1
4 eureka.client.service-
5   url.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka
6 #不要将本身进行注册
7 eureka.client.register-with-eureka=false
8 #不要在该应用中检测服务
9 eureka.client.fetch-registry=false
```

- 运行启动类

```
1 @SpringBootApplication
2 // 启动了注册中心（启动了Eureka服务）
3 @EnableEurekaServer
4 public class SpringcloudEurekaServerApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(SpringcloudEurekaServerApplication.class,
8         args);
9     }
10 }
```

4.4 搭建服务的提供者

- 编写pom.xml

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5   </dependency>
6
7   <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-web</artifactId>
10  </dependency>
11 </dependencies>
```

- 编写application.properties/application.yml

```
1 server.port=8081
2 spring.application.name=springcloud-eureka-producer
3 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
4 eureka.client.register-with-eureka=true
5 eureka.client.fetch-registry=true
```

- 运行启动类

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class SpringcloudEurekaProducerApplication {
4
5   public static void main(String[] args) {
6     SpringApplication.run(SpringcloudEurekaProducerApplication.class,
7     args);
8   }
9 }
```

4.5 搭建服务的消费方

- 编写pom文件

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
5   </dependency>
6
7   <dependency>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-web</artifactId>
10  </dependency>
11 </dependencies>
```

- 编写application.properties/application.yml

```
1 server.port=8082
2 spring.application.name=springcloud-eureka-consumer
3 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
4 eureka.client.register-with-eureka=true
5 eureka.client.fetch-registry=true
```

- 运行启动类

```
1  @SpringBootApplication
2  @EnableEurekaClient
3  public class SpringcloudEurekaProducerApplication {
4
5      public static void main(String[] args) {
6          SpringApplication.run(SpringcloudEurekaProducerApplication.class,
7              args);
8      }
9  }
```

- 编写配置类

```
1  @Configuration
2  public class EurekaConfig {
3
4      /**
5       * 依赖Ribbon负载均衡器 使用注解@LoadBalanced
6       * @LoadBalanced注解不能省略
7       * @return
8       */
9  @Bean
10 @LoadBalanced
11 public RestTemplate restTemplate() {
12     return new RestTemplate();
13 }
14 }
```

- 编写控制器类

```
1  @RestController
2  public class OrderController {
3
4      /**
5       * RestTemplate可以进行服务的调用。对象是SpringBoot提供
6       * 底层
7       */
8  @Autowired
9  private RestTemplate restTemplate;
10
11 @GetMapping("/findOrder")
12 public String findOrder() {
13     String url = "http://springcloud-eureka-producer/sayHello";
14     String result = this.restTemplate.getForObject(url, String.class);
15     System.out.println(result);
16     return result;
17 }
18 }
```

五.高可用的注册中心

5.1 概念

- 注册中心实现服务治理，作用非常重要。
- 如果注册中心宕机了，会导致整个应用（微服务）不可用
- 使得注册中心达到高可用，就是使用集群

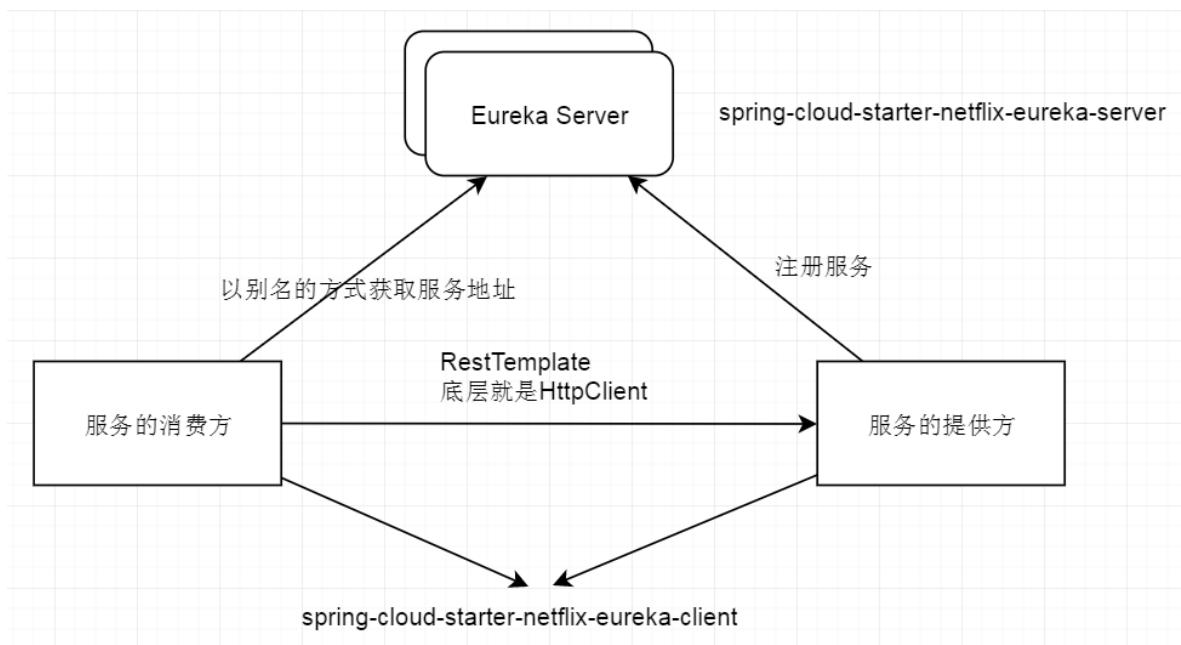
5.2 Eureka高可用实现原理

- 默认情况下，Eureka让服务器不注册自身（不要注册自己）

```
1 eureka.client.register-with-eureka=false  
2 eureka.client.fetch-registry=false
```

- 如何实现高可用

```
1 eureka.client.register-with-eureka=true  
2 eureka.client.fetch-registry=true
```



5.3 模拟两台注册中心

- 第一台机器 (8080)

```
1 server.port=8080  
2 spring.application.name=springcloud-eureka-server  
3 eureka.instance.hostname=127.0.0.1  
4 # http://127.0.0.1:8080/eureka/  
5 eureka.client.service-  
  url.defaultZone=http://${eureka.instance.hostname}:9090/eureka/  
6 #\u6CE8\u518C\u4E2D\u5FC3\uFF0C\u4E0D\u9700\u8981\u5C06\u81EA\u8EAB\u6CE8\u518C  
7 eureka.client.register-with-eureka=true  
8 #\u6CE8\u518C\u4E2D\u5FC3\uFF0C\u4E0D\u9700\u8981\u68C0\u7D22\u670D\u52A1  
9 eureka.client.fetch-registry=true
```

- 第二台机器 (9090)

```
1 server.port=9090
2 spring.application.name=springcloud-eureka-server
3 eureka.instance.hostname=127.0.0.1
4 # http://127.0.0.1:8080/eureka/
5 eureka.client.service-
url.defaultZone=http://${eureka.instance.hostname}:8080/eureka/
6 #\u6CE8\u518C\u4E2D\u5FC3\uFF0C\u4E0D\u9700\u8981\u5C06\u81EA\u8EAB\u6CE8\u518C
7 eureka.client.register-with-eureka=true
8 #\u6CE8\u518C\u4E2D\u5FC3\uFF0C\u4E0D\u9700\u8981\u68C0\u7D22\u670D\u52A1
9 eureka.client.fetch-registry=true
```

六.Eureka细节

6.1 服务消费者模式（获取服务）

- 本地缓存服务列表

```
1 #每隔30秒更新服务（从注册中心更新服务地址）
2 eureka.client.registry-fetch-interval-seconds=30
```

6.2 服务的注册模式（注册服务）

- 服务下线（失效剔除）
 - 服务的提供者会向注册中心一个应答：告知注册中心我下线了。这种下线是一种正常下线（开发和测试阶段重启服务）
- 失效剔除
 - 有些服务在注册中心中已经没有能力给客户端提供服务了，服务的提供方没有告知注册中心我下线了（内存溢出，网络故障...）此时就必须有一种机制将这个服务去除
 - Eureka Server有一种机制：指定一个定时任务，每隔60秒将服务列表中这种服务（不可用）去除。
- 自我保护(重点)**
 - 默认情况下，Eureka Client会定时向Eureka Server发送心跳。如果Eureka Server在一定的时间内没有收到心跳，此时Eureka Server会从服务列表中剔除（90秒）
 - 如果在短时间内丢失了大量心跳，此时Eureka Server自动开启自我保护机制（不要轻易删除服务）。开发测试阶段其实会经常出现没有心跳（建议开发测试阶段不要开启自我保护，产品阶段建议开启）。

Renews threshold	3
Renews (last min)	0

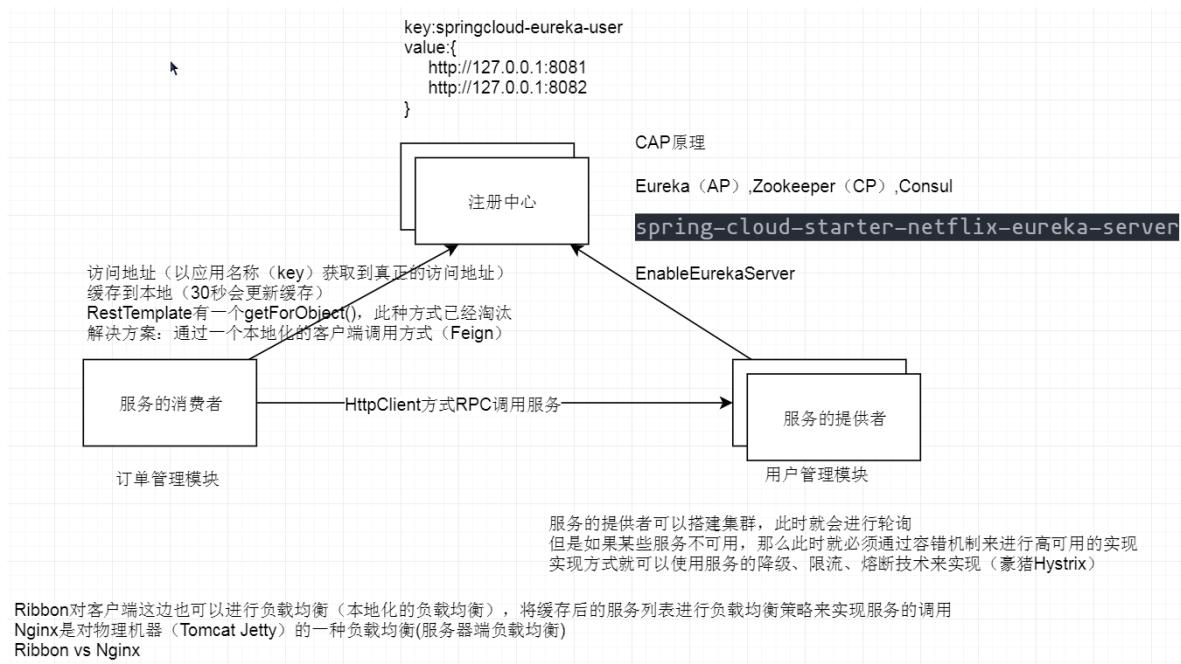
- renews：最后一分钟收到的心跳数，如果这个值<renews threshold（心跳阈值），会触发自我保护
- 矛盾体：
 - 如果网络延迟，网络故障，本身服务健康，那么应该开启自我保护
- 开启服务保护**

```
#客户端向服务器发送心跳间隔时间  
#eureka.instance.lease-renewal-interval-in-seconds=1  
#eureka.instance.lease-expiration-duration-in-seconds=2
```

```
#开启服务的自我保护，false表示关闭自我保护  
#eureka.server.enable-self-preservation=true  
#eureka.server.eviction-interval-timer-in-ms=
```

- 结论
 - 开发测试阶段，不建议开启自我保护
 - 产品阶段建议开启自我保护

七.Spring Cloud复习



八. 使用其他的注册中心

8.1 背景

- Eureka已经闭源
- 使用另外的注册中心：Consul, Zookeeper

8.2 Consul

- <https://www.consul.io/>
- 分布式服务发现和配置的管理系统
- <https://github.com/HashiCorp/consul>
- 下载

- 搭建环境
 - 解压
 - 配置环境变量
 - 启动consul

```
1 consul agent -dev -ui -node=cy
```

- ## ○ 测试安装

1 http://localhost:8500



- 搭建客户端

- ## ○ 服务的提供方

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-consul-discovery</artifactId>
4 </dependency>
```

org.springframework.boot spring-boot-starter-actuator

```
1  ``properties  
2  server.port=8501  
3  spring.application.name=springcloud-consul-user-producer
```

```
spring.cloud.consul.host=127.0.0.1 spring.cloud.consul.port=8500 #服务地址（ip地址）  
spring.cloud.consul.discovery.hostname=192.168.43.100 ````
```

```
1  ```java
2  @RestController
3  public class UserController {
4
5      @GetMapping("/findUser")
6  public String findUser() {
7      return "用户服务, 订单服务调用用户服务";
8  }
9 }
10 ```
11
12 ```java
13 @SpringBootApplication
14 @EnableDiscoveryClient
15 public class SpringcloudConsulUserProducerApplication {
16
17     public static void main(String[] args) {
18         SpringApplication.run(SpringcloudConsulUserProducerApplication.class,
19         args);
20     }
21 }
22 ````
```

- 服务的消费方

```

1  @RestController
2  public class OrderController {
3
4      @Autowired
5      private RestTemplate restTemplate;
6
7      @Autowired
8      private DiscoveryClient discoveryClient;
9
10     @GetMapping("/findOrder")
11     public String findOrder() {
12         List<String> serviceList = new ArrayList<>();
13         List<ServiceInstance> serviceInstanceList =
14             this.discoveryClient.getInstances("springcloud-consul-user-producer");
15         if (!CollectionUtils.isEmpty(serviceInstanceList)) {
16             for (ServiceInstance serviceInstance : serviceInstanceList) {
17                 serviceList.add(serviceInstance.getUri().toString());
18             }
19         }
20         String service = serviceList.get(0);
21         String url = service + "/findUser";
22
23         String result = restTemplate.getForObject(url, String.class);
24         return result;
25     }
26 }
27 }
```

8.3 Zookeeper

- 服务的协调工具
- 实现服务的注册与发现，实现分布式锁，实现负载均衡
- 搭建环境

```

1  拷贝zoo_sample.cfg文件，重命名为zoo.cfg
2  修改数据存放路径（temdir）
3
4  sh zkServer.sh start
```

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4               xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5               http://maven.apache.org/xsd/maven-4.0.0.xsd">
6     <modelVersion>4.0.0</modelVersion>
7     <parent>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-parent</artifactId>
10        <version>2.0.1.RELEASE</version>
11        <relativePath/> <!-- lookup parent from repository -->
12    </parent>
13    <groupId>com.prosay</groupId>
14    <artifactId>springcloud-zookeeper-order-consumer</artifactId>
15    <version>0.0.1-SNAPSHOT</version>
16    <name>springcloud-zookeeper-order-consumer</name>
```

```
15
16     <properties>
17         <java.version>1.8</java.version>
18     </properties>
19
20     <dependencies>
21         <dependency>
22             <groupId>org.springframework.boot</groupId>
23             <artifactId>spring-boot-starter-web</artifactId>
24         </dependency>
25         <dependency>
26             <groupId>org.springframework.cloud</groupId>
27             <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>
28         </dependency>
29
30         <dependency>
31             <groupId>org.springframework.boot</groupId>
32             <artifactId>spring-boot-devtools</artifactId>
33             <scope>runtime</scope>
34             <optional>true</optional>
35         </dependency>
36         <dependency>
37             <groupId>org.springframework.boot</groupId>
38             <artifactId>spring-boot-starter-test</artifactId>
39             <scope>test</scope>
40         </dependency>
41         <dependency>
42             <groupId>org.springframework.boot</groupId>
43             <artifactId>spring-boot-starter-actuator</artifactId>
44         </dependency>
45     </dependencies>
46
47     <dependencyManagement>
48         <dependencies>
49             <dependency>
50                 <groupId>org.springframework.cloud</groupId>
51                 <artifactId>spring-cloud-dependencies</artifactId>
52                 <version>Finchley.M7</version>
53                 <type>pom</type>
54                 <scope>import</scope>
55             </dependency>
56         </dependencies>
57     </dependencyManagement>
58
59     <repositories>
60         <repository>
61             <id>spring-milestones</id>
62             <name>Spring Milestones</name>
63             <url>https://repo.spring.io/libs-milestone</url>
64             <snapshots>
65                 <enabled>false</enabled>
66             </snapshots>
67         </repository>
68     </repositories>
69
70     <build>
71         <plugins>
72             <plugin>
73                 <groupId>org.springframework.boot</groupId>
74                 <artifactId>spring-boot-maven-plugin</artifactId>
75             </plugin>
76         </plugins>
77     </build>
78
79 </project>
```

```
1 server.port=8081
2 spring.application.name=springcloud-zookeeper-user-producer
3 spring.cloud.zookeeper.connect-string=192.168.43.135:2181
```

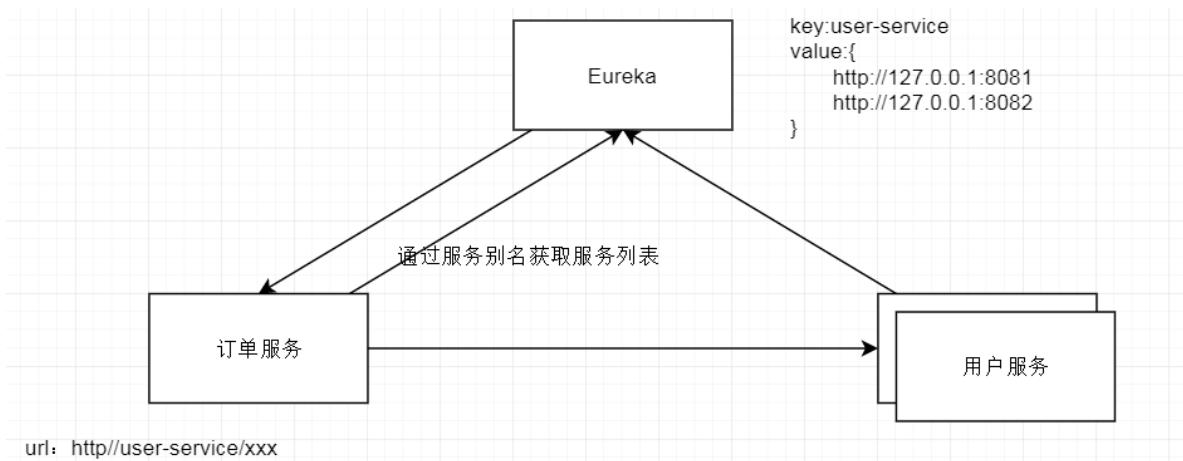
```
1 http://springcloud-zookeeper-user-producer/findUser
```

九. 客户端负载均衡

9.1 概念

- 在Spring Cloud中使用一个第三方组件Ribbon来实现负载均衡（本地负载均衡）。
- 服务的消费方会从注册中心获取服务列表，将服务列表缓存到本地
- **Ribbon vs Nginx**
 - Nginx是服务器端负载均衡。
 - Nginx是客户端所有请求统一交给它，然后由它进行负载均衡，属于一种服务器端负载均衡。Nginx主要用来对Tomcat，Jetty来实现负载均衡
 - Ribbon将服务地址缓存到本地，然后在本地进行负载均衡。主要用于微服务架构中这种RPC调用的负载均衡

9.2 使用DiscoveryClient实现负载均衡



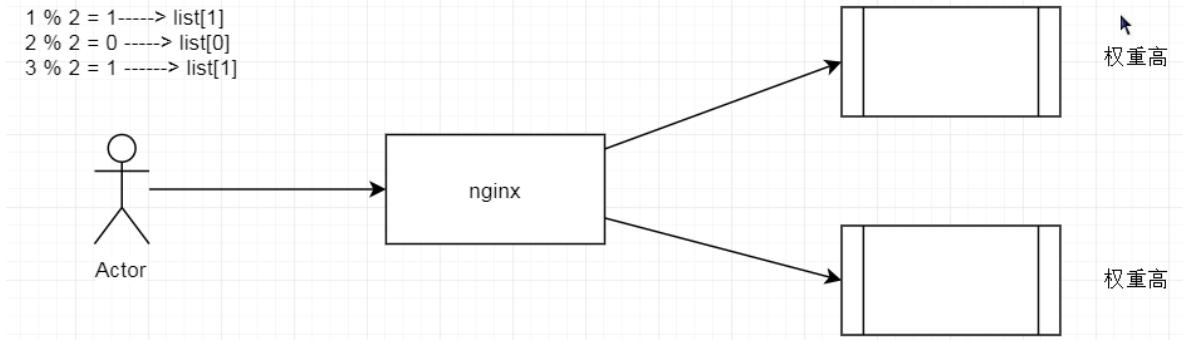
url: http://user-service/xxx

- 1.本地会将服务地址列表缓存到本地
- 2.此时就必须有一种负载均衡算法来真正地获取地址（本地实现默认轮询进行负载）

服务的提供方
可以搭建集群

请求数 % 总服务数得到一个余数，通过这个余数在服务地址列表中以下标索引去取值

```
requestCount = 1
serviceCount = 2
1 % 2 = 1 ----> list[1]
2 % 2 = 0 -----> list[0]
3 % 2 = 1 -----> list[1]
```



```

1  @RestController
2  public class OrderController {
3
4      /**
5       * RestTemplate可以进行服务的调用。对象是SpringBoot提供
6       * 底层
7       */
8      @Autowired
9      private RestTemplate restTemplate;
10
11     @Autowired
12     private DiscoveryClient discoveryClient;
13
14     private Integer requestCount = 1;
15
16
17     @GetMapping("/findOrder")
18     public String findOrder() {
19         String url = "http://springcloud-eureka-producer/sayHello";
20         String result = this.restTemplate.getForObject(url, String.class);
21         System.out.println(result);
22         return result;
23     }
24
25     @GetMapping("/discoveryClient")
26     public String discoveryClient() {
27         String serviceId = "springcloud-eureka-producer";
28         List<ServiceInstance> serviceInstanceList =
29             this.discoveryClient.getInstances(serviceId);
30         if (CollectionUtils.isEmpty(serviceInstanceList)) {
31             return "No available instances found";
32         }
33         return "Available instances: " + serviceInstanceList.size();
34     }
35 }
```

```
30         return null;
31     }
32
33     // 服务集群数（几台机器提供服务）
34     int serviceCount = serviceInstanceList.size();
35     int index = this.requestCount % serviceCount;
36     this.requestCount++;
37
38     String url = serviceInstanceList.get(index).getUri().toString();
39
40     url = url + "/findUser";
41     return this restTemplate.getForObject(url, String.class);
42 }
43 }
44 }
```

十.Spring Cloud服务的保护机制框架-Hystrix(重点)

10.1 微服务高可用技术

- 高可用架构是贯穿这整个的分布式系统。属于三高的其中的一个特点
- 高可用的解决什么问题：我们必须提出一些解决方案，避免整个分布式系统不要被某个微服务的故障而连累
 - 服务间的调用的超时
 - 服务调用失败
 - 物理机器的宕机
 - 网络的延迟
 - 网络的故障
 - ...
- 解决方案
 - 服务的降级
 - 服务的熔断
 - 服务的限流
 - 集群容错
 - 超时控制
 - 重试机制
 - 资源的隔离
 -

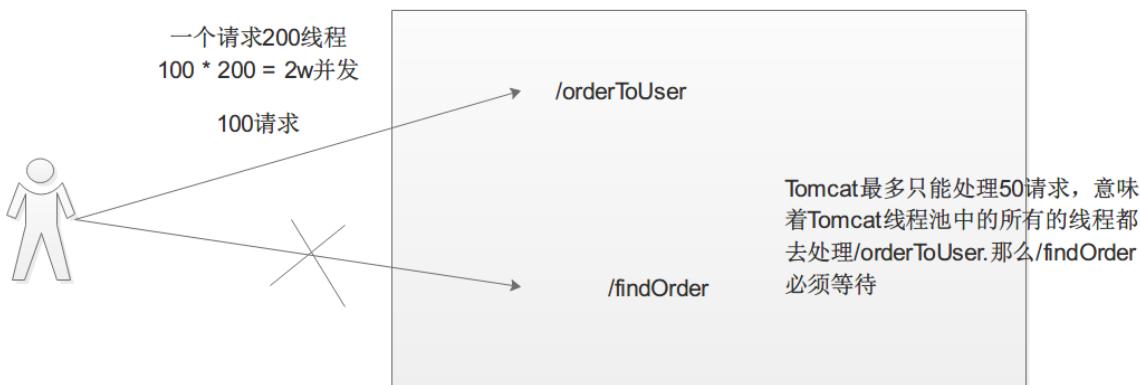
10.2 模拟一个场景（服务的雪崩问题）

- 默认情况下，项目运行在tomcat容器中。tomcat默认情况下只有一个线程池来处理客户端发送的请求
- 在高并发的场景下，如果客户端所有的请求堆积在一个服务接口上，此时会产生Tomcat线程池中的所有的线程都会去处理这个请求，那么其他的接口中的方法就不会被处理，此时导致服务的延迟和等待

/orderToUser 服务的调用延迟1.5s



假设Tomcat最大请求数为50，客户端发送了100个请求。此时会产生50个请求延迟等待
Tomcat是单个线程池，所有的线程都会去处理50请求（/orderToUser），那么等待的那50个
请求都会延迟。言下之意，/findOrder会等待



10.3 解决方案

10.3.1 服务的降级

- 在高并发的场景下，尽量将主要业务被用户访问到，而那些非主要业务而又会产生异常的访问要尽量的避免经常被访问。
- 防止用户一直等待服务的访问，而是直接给用户一个友好的提示

10.3.2 服务的熔断

- 当请求达到一定的极限（设置阈值），如果达到了或超出了阈值，后续的请求直接拒绝
- 我们可以结合服务降低进行处理

10.3.3 服务的限流

- 对接口访问进行限制，常用一些算法进行限流
 - 令牌桶
 - 漏桶
 - 计数器

10.3.4 服务隔离

- 默认情况下，只有一个线程池处理请求，可能会产生服务雪崩
- 使用线程池隔离和信号量隔离
 - 线程池隔离原理：每一个服务接口都有自己独立的线程池

- 信号量隔离原理：使用计数器（信号量）记录当前有多少个线程在运行，当请求处理首先要判断计数器的值，若这个值超过了最大线程数则拒绝该请求，如果没有超过，此时计数器加1。

10.4 Hystrix概述

- 服务熔断框架，服务的保护管理框架。主要用来实现服务的降级、限流和熔断
- Hystrix是Netflix开源项目。中文名叫做“豪猪”。具有自我保护能力。
- Hystrix主要用来解决服务的雪崩问题



10.5 搭建环境

```
1 springcloud-xxx-parent
2 ----springcloud-xxx-common
3 ----springcloud-xxx-api(接口)
4 -----springcloud-xxx-api-user (用户接口)
5 -----springcloud-xxx-api-order (订单接口)
6 ----springcloud-xxx-api-user-impl (用户接口的实现)
7 ----springcloud-xxx-api-order-impl (订单接口的实现)
```

```
1 springcloud-xxx-parent
2 ----springcloud-xxx-common
3 ----springcloud-xxx-api(接口)
4 ----springcloud-xxx-manage-user(分层次 web, service, dao)
5 ----springcloud-xxx-manage-order(分层次)
```

10.5.1 编写pom文件

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

- 服务的提供方

```
1 server:
2   port: 8010
3
4 spring:
5   application:
6     name: springcloud-hystrix-user
7
8 eureka:
9   client:
10    service-url:
11      defaultZone: http://127.0.0.1:9090/eureka
12      register-with-eureka: true
13      fetch-registry: true
```

- 服务的消费方

```
1 server:
2   port: 8020
3
4 spring:
5   application:
6     name: springcloud-hystrix-order
7
8 eureka:
9   client:
10    service-url:
11      defaultZone: http://127.0.0.1:9090/eureka
12      register-with-eureka: true
13      fetch-registry: true
14
15 ribbon:
16   ###指的是建立连接所用的时间，适用于网络状况正常的情况下，两端连接所用的时间。
17   ReadTimeout: 5000
18   ###指的是建立连接后从服务器读取到可用资源所用的时间。
19   ConnectTimeout: 5000
20
21 feign:
22   hystrix:
23     enabled: true
```

```
1 @RestController
2 public class OrderServiceImpl implements OrderService {
3
4   @Autowired
5   private UserServiceFeign userServiceFeign;
6
7   @Override
8   @GetMapping("/orderToUserHystrix")
9   @HystrixCommand(fallbackMethod = "orderToUserFallback")
10  public String orderToUserHystrix() {
11    User user = this.userServiceFeign.findUser();
12    if (user != null) {
13      return user.toString();
14    }
15    return "没有此人";
16  }
17
18  /**
19   * 回调方法，友好提示
20   * @return
21   */
22  public String orderToUserFallback() {
```

```

23     return "您出错了，请重试";
24 }
25
26 @Override
27 @GetMapping("/orderToUser")
28 public String orderToUser(String name) {
29     User user = this.userServiceFeign.findUserByName(name);
30     if (user != null) {
31         return user.toString();
32     }
33     return "没有此人";
34 }
35
36 @Override
37 @GetMapping("/findOrder")
38 public String findOrder() {
39     Order order= new Order();
40     order.setId(1);
41     order.setName("测试订单");
42     if (order != null) {
43         return order.toString();
44     }
45
46     return "没有此订单";
47 }
48 }
49

```

```

1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableFeignClients
4 @EnableHystrix
5 public class SpringCloudHystrixOrderApp<EnableDiscoveryClient> {
6
7     public static void main(String[] args) {
8         SpringApplication.run(SpringCloudHystrixOrderApp.class,args);
9     }
10 }

```

十一.本地化的声明式的服务调用框架Feign

- Feign是一个声明式Http远程调用工具
- 提供了**接口和注解**方式结合来实现本地客户端调用

```

1 @FeignClient("springcloud-hystrix-user")
2 public interface UserServiceFeign extends UserService {
3 }

```

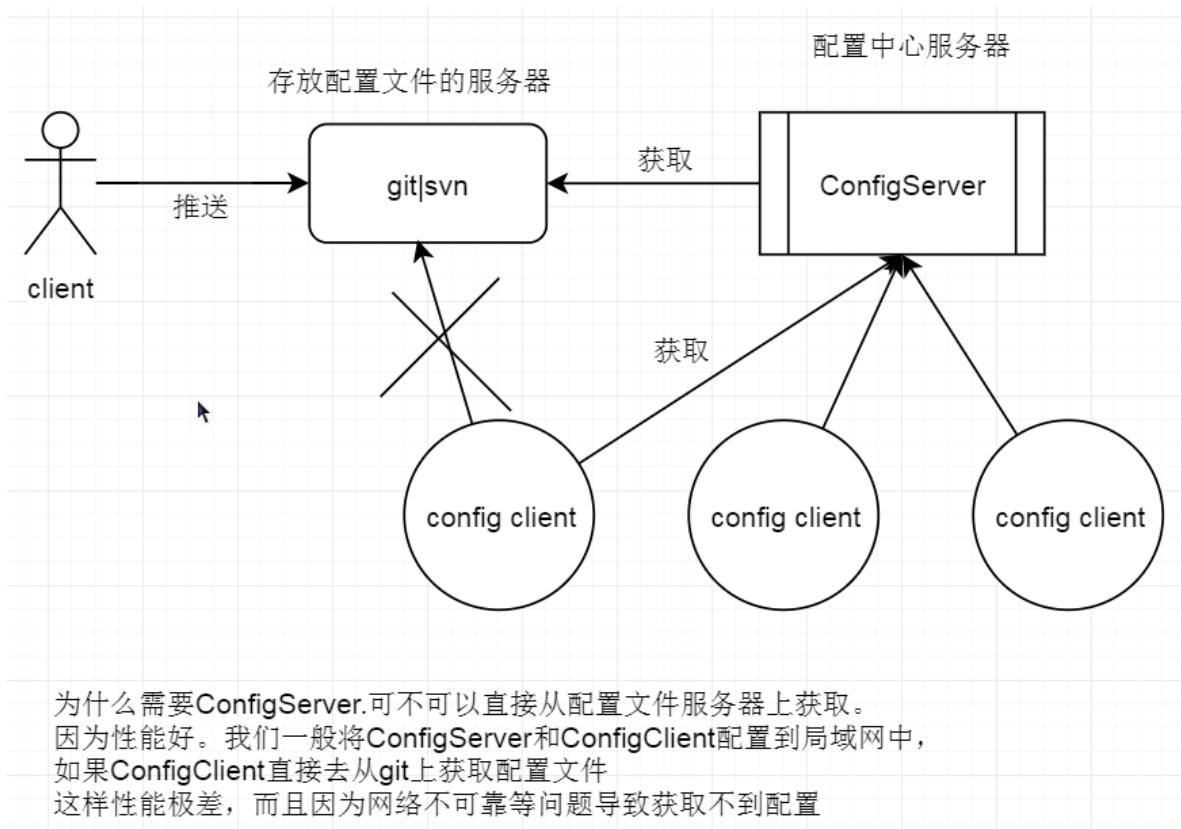
- Feign继承特性
 - 为了避免代码冗余
- 总结
 - Feign简化了客户端调用。因为RestTemplate调用方式很麻烦（url获取）

十二.分布式配置中心（Spring Cloud Config）

12.1 概述

- 由于微服务数量会很多，为了方便服务的配置的统一管理，**实时更新**，所以我们需要引入分布式配置管理
- Spring Cloud中有分布式配置中心组件（Spring Cloud Config）。但是我们也可以使用另外的配置中心组件（携程Apollo）
- Spring Cloud Config远程访问Git仓库，然后需要配置ConfigServer和ConfigClient。都是Eureka的客户端
- ConfigServer是一个集中式的配置服务器。这个服务器需要从Git仓库中获取配置文件，然后缓存到本地
- ConfigClient是ConfigServer的客户端，这些客户端操作存储在ConfigServer中的配置文件。
- 当微服务启动时会请求ConfigServer获取配置文件，再启动容器

12.2 架构



12.3 搭建环境

12.3.1 搭建git环境

- 远程仓库github, gitlab, gitee。我们选用码云（gitee）
- 注册用户
- 建立仓库
- 建立文件夹（项目中的模块（微服务）为单位建立文件夹）
 - user_config
 - order_config
- 建立配置文件

- 命名规范。公司项目开发的场景来区分(dev, test, pro)
 - {application}-{profile}
 - test-config-dev.properties
 - test-config-pro.properties

12.3.2 搭建Eureka Server

12.3.3 搭建ConfigServer

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>org.springframework.cloud</groupId>
8   <artifactId>spring-cloud-config-server</artifactId>
9 </dependency>
```

```

1 server.port=8010
2 spring.application.name=springcloud-config-server
3 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
4 spring.cloud.config.server.git.uri=https://gitee.com/zhouzhengjeff/prosay_config.git
5 spring.cloud.config.server.git.search-paths=user_config
6 spring.cloud.config.label=master
```

```

1 @SpringBootApplication
2 @EnableConfigServer
3 public class SpringcloudConfigServerApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringcloudConfigServerApplication.class, args);
7     }
8
9 }
```

12.3.4 搭建ConfigClient

```

1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-config-client</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>org.springframework.cloud</groupId>
8   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
9 </dependency>
```

```
1 server.port=8020
2 #####值与git远程仓库中的配置文件名(applicationName-profile-dev.properties)的
3 applicationName(应用程序名)一样
4 spring.application.name=test-config
5 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
6 spring.cloud.config.profile=dev
7 spring.cloud.config.discovery.service-id=springcloud-config-server
8 spring.cloud.config.discovery.enabled=true
```

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class SpringcloudConfigClientApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringcloudConfigClientApplication.class, args);
7     }
8 }
```

12.3.5 动态刷新配置文件配置

- 手动刷新：采用actuator
 - 编写pom.xml

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-actuator</artifactId>
4 </dependency>
```

- 编写application.properties

```
1 #####开启监控断点
2 management.endpoints.web.exposure.include=*
```

- 加上@RefreshScope注解

```
1 /actuator/refresh
```

- 利用Postman来发送请求一次。而且这次请求是POST请求

```
1 http://localhost:8020/actuator/refresh
```

- 实时刷新：利用Spring Task，Spring Cloud Bus

十二.Zuul（服务网关）

12.1 概述

- 在我们的微服务架构中，由于服务数量猛增，而且服务关系特别复杂，例如我们有很多的服务都可能需要进行权限认证，进行日志的记录，统一设置...
- 基于上面的场景，如果每个微服务都去实现认证，日志，统一设置，那么项目会非常冗余
- 上面的问题已经暴露：直接将服务暴露给客户端，公共的任务冗余

- 解决方案：微服务网关。

12.2 网关分类

- 开发API(Open API)。企业需要将数据作为一种开发平台向外部开放。以rest方法来开放。微信，qq，淘宝。必须知道平台接入接口，还需要一些权限....
- 微服务网关

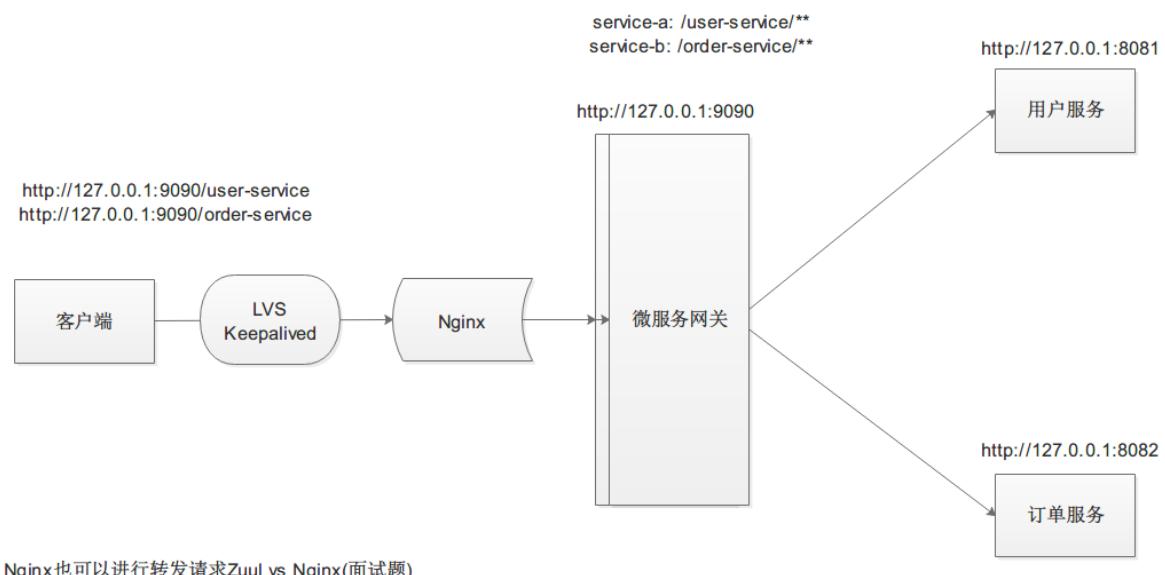
12.3 网关的框架

- Kong网关
- **Netflix Zuul**: <https://github.com/Netflix/zuul>
- orange: 国人开发: <https://github.com/orlabs/orange>

12.4 网关应用场景

- 路由转发
- 日志
- 权限控制
- 监控
- 负载均衡

12.5 网关架构



Nginx也可以进行转发请求Zuul vs Nginx(面试题)

- 1.Nginx主要是做服务器（硬件）的负载均衡，反向代理
- 2.zuul网关一种基于软件层面路由转发(Ribbon和eureka)，权限，日志处理，监控...

微服务网关也会做集群，此时就要用Nginx来代理zuul

12.6 Zuul的概述

- <https://baike.baidu.com/item/zuul/7625508?fr=aladdin>
- Zuul在微服务架构中是守门的BOSS

```
1 Zuul is the front door for all requests from devices and web sites to the
2 backend of the Netflix streaming application. As an edge service application,
3 Zuul is built to enable dynamic routing, monitoring, resiliency and security.
4 It also has the ability to route requests to multiple Amazon Auto Scaling
5 Groups as appropriate.
```

12.7 搭建Zuul环境

12.7.1 编写pom文件

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>org.springframework.cloud</groupId>
8   <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
9 </dependency>
10
11 <dependency>
12   <groupId>org.springframework.boot</groupId>
13   <artifactId>spring-boot-starter-actuator</artifactId>
14 </dependency>
```

12.7.2 编写application.yml

```
1 server.port=80
2 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
3 spring.application.name=springcloud-zuul-gateway
4 #spring.cloud.config.profile=dev
5 #spring.cloud.config.discovery.enabled=true
6 #spring.cloud.config.discovery.service-id=springcloud-config-server
7
8
9
10 ####配置了路由网关
11 zuul.routes.service-a.path=/user-service/**
12 zuul.routes.service-a.serviceId=springcloud-user-service
13 zuul.routes.service-b.path=/order-service/**
14 zuul.routes.service-b.serviceId=springcloud-order-service
```

12.7.3 编写启动类

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableZuulProxy
4 public class SpringcloudZuulGatewayApplication {
5
6   public static void main(String[] args) {
7     SpringApplication.run(SpringcloudZuulGatewayApplication.class, args);
8   }
9
10 }
```

```
1 http://localhost/order-service/index---->订单服务的index请求
2 http://localhost/user-service/index---->用户服务的index请求
```

12.7.4 网关可以进行过滤操作

```
1 @Component
2 public class AuthenticateFilter extends ZuulFilter {
3
4     @Override
5     public String filterType() {
6         return "pre";
7     }
8
9     @Override
10    public int filterOrder() {
11        return 0;
12    }
13
14    @Override
15    public boolean shouldFilter() {
16        return true;
17    }
18
19    @Override
20    public Object run() throws ZuulException {
21        System.out.println("过滤器进来了");
22        RequestContext requestContext = RequestContext.getCurrentContext();
23        HttpServletRequest request = requestContext.getRequest();
24
25
26        /* HttpSession session = request.getSession();
27        User user = (User) session.getAttribute("user");
28        if (user == null) {
29            // 不要继续向下执行了，不要再调用微服务
30            requestContext.setSendZuulResponse(false);
31            requestContext.setResponseStatusCode(401);
32            requestContext.setResponseBody("没有认证，请重新认证");
33            return null;
34        }
35    }*/
36
37        return null;
38    }
39 }
```

12.8 动态网关

- 将路由配置在配置文件中，如果路由规则发生了变化，需要重启服务器。这样做肯定不合适。
- 解决方案：使用Spring cloud config（分布式配置中心），这样就可以实现动态网关
- 如何实现
 - 在git上建立配置文件
 - 配置分布式配置中心服务器端
 - 将网关工程进行重构
 - 将路由配置放在git配置文件中

master ▾

prosay_config / user_config / zuul-service-dev.properties

zuul-service-dev.properties 206 Bytes

幸福和完整的家 提交于 1分钟前 . 更新 zuul-service-dev.properties

```
1 zuul.routes.service-a.path=/user-service/**  
2 zuul.routes.service-a.serviceId=springcloud-user-service  
3 zuul.routes.service-b.path=/order-service/**  
4 zuul.routes.service-b.serviceId=springcloud-order-service
```

■ 修改pom文件(加上下面配置)

```
1 <dependency>  
2   <groupId>org.springframework.cloud</groupId>  
3   <artifactId>spring-cloud-config-client</artifactId>  
4 </dependency>
```

```
1 server.port=80  
2 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka  
3  
4 spring.application.name=zuul-service  
5 spring.cloud.config.profile=dev  
6 spring.cloud.config.discovery.enabled=true  
7 spring.cloud.config.discovery.service-id=springcloud-config-server  
8  
9 #####开启监控端点  
10 management.endpoints.web.exposure.include=*
```

11
12
13 #####配置了路由网关
14 #zuul.routes.service-a.path=/user-service/**
15 #zuul.routes.service-a.serviceId=springcloud-user-service
16 #zuul.routes.service-b.path=/order-service/**
17 #zuul.routes.service-b.serviceId=springcloud-order-service

```
1 @SpringBootApplication  
2 @EnableEurekaClient  
3 @EnableZuulProxy  
4 public class SpringcloudZuulGatewayApplication {  
5  
6     public static void main(String[] args) {  
7         SpringApplication.run(SpringcloudZuulGatewayApplication.class, args);  
8     }  
9  
10    /**  
11     * Zuul配置需要实时更新  
12     * @return  
13     */  
14    @RefreshScope  
15    public ZuulProperties zuulProperties() {  
16        return new ZuulProperties();  
17    }  
18  
19 }
```

十三.Swagger API文档的生成工具

13.1 背景

- 微服务架构项目开发，为了让服务之间调用通畅，为了让前后台分离方便。我们必须给出详细的API接口文档
- 文档是必须规范，而且一般都要集中管理
- 有了API文档，便于测试

13.2 Swagger概述

- 官网：<https://swagger.io/>
- 支持注解，自动生成文档，而且界面友好
- 与SpringBoot整合是非常容易。
- 实时更新

13.3 搭建环境

```
1 <!--swagger集成-->
2 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger2 -->
3 <dependency>
4   <groupId>io.springfox</groupId>
5   <artifactId>springfox-swagger2</artifactId>
6   <version>2.9.2</version>
7 </dependency>
8
9
10 <!-- https://mvnrepository.com/artifact/io.springfox/springfox-swagger-ui -->
11 <dependency>
12   <groupId>io.springfox</groupId>
13   <artifactId>springfox-swagger-ui</artifactId>
14   <version>2.9.2</version>
15 </dependency>
```

```
1 @Configuration
2 @EnableSwagger2
3 public class Swagger2Config {
4
5     @Bean
6     public Docket docket() {
7         return new Docket(DocumentationType.SWAGGER_2)
8             .apiInfo(apiInfo())
9             .select()
10            .apis(RequestHandlerSelectors.basePackage("com.prosay.springboot"))
11            .paths(PathSelectors.any())
12            .build();
13     }
14
15     private ApiInfo apiInfo() {
16         return new ApiInfoBuilder()
17             .title("动脑学院")
18             .description("动脑学院分布式微服务专业培训")
19             .termsOfServiceUrl("http://www.prosay.com")
20             .version("1.0.0")
21             .build();
22     }
23 }
```

```
24  
25  }
```

```
1  http://localhost:8080/swagger-ui.html
```

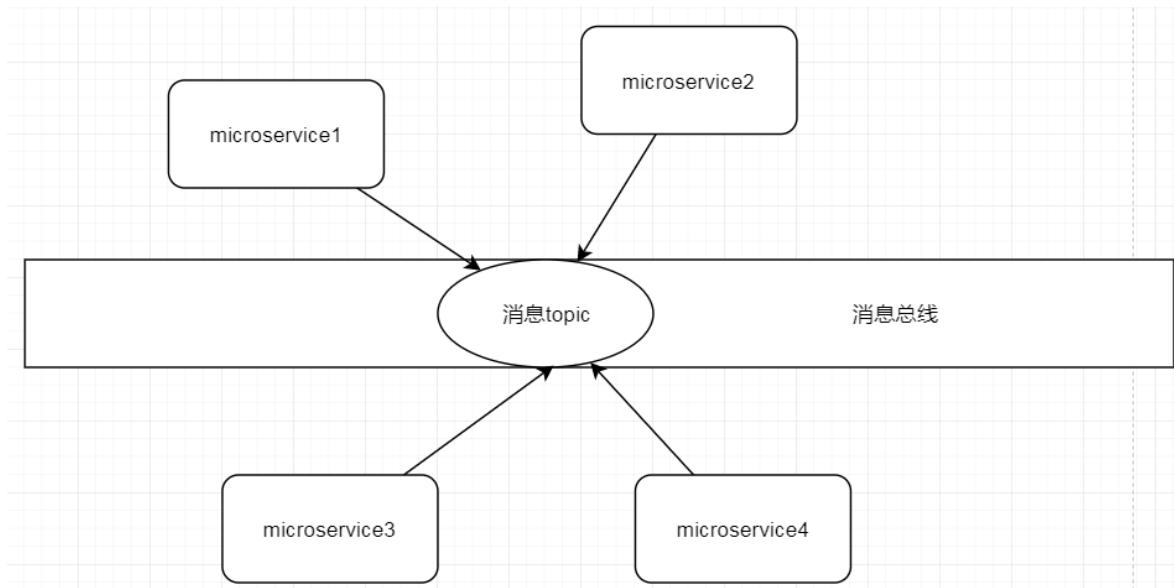
13.4 zuul和swagger整合

- 概述
 - 通过微服务网关与swagger2整合来进行微服务的API文档管理
- 搭建环境

十四.Spring Cloud Bus

14.1 概念

- 在微服务架构中，使用消息中间件来构建一个公用的消息主题，这个主题用来连接各个微服务。
- 这个主题可以进行广播消息让每一个微服务得到消息



- 在Spring Cloud中也有对应的解决方案----Spring Cloud Bus。将分布式的节点用消息中间件连接起来，这样便于搭建消息总线，然后与Spring Cloud Config实现微服务应用配置动态更新功能。
- Spring Cloud Bus实际上整合了RabbitMQ和Kafka。

14.2 搭建环境

14.2.1 搭建微服务的分布式配置中心

- gitee
- config server

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
4 </dependency>
5 <dependency>
6   <groupId>org.springframework.cloud</groupId>
7   <artifactId>spring-cloud-config-server</artifactId>
8 </dependency>
9 <dependency>
10  <groupId>org.springframework.cloud</groupId>
11  <artifactId>spring-cloud-starter-bus-amqp</artifactId>
12 </dependency>
```

```
1 server.port=8010
2 spring.application.name=springcloud-config-server
3 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
4 spring.cloud.config.server.git.uri=https://gitee.com/zhouzhengjeff/prosay_config.git
5 spring.cloud.config.server.git.search-paths=config
6 spring.cloud.config.label=master
```

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableConfigServer
4 public class SpringcloudConfigServerApplication {
5
6     public static void main(String[] args) {
7         SpringApplication.run(SpringcloudConfigServerApplication.class,
8             args);
9     }
10 }
```

- config client

```
1 <dependency>
2   <groupId>org.springframework.cloud</groupId>
3   <artifactId>spring-cloud-config-client</artifactId>
4 </dependency>
5
6 <dependency>
7   <groupId>org.springframework.cloud</groupId>
8   <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
9 </dependency>
10
11 <dependency>
12   <groupId>org.springframework.boot</groupId>
13   <artifactId>spring-boot-starter-actuator</artifactId>
14 </dependency>
15
16 <dependency>
17   <groupId>org.springframework.cloud</groupId>
18   <artifactId>spring-cloud-starter-bus-amqp</artifactId>
19 </dependency>
```

```
1 server.port=8020
2 #####\u503C\u4E0Egit\u8FDC\u7A0B\u4ED3\u5E93\u4E2D\u7684\u914D\u7F6E\u6587\u4EF
6\u540D\uFF08applicationName-profile-
dev.properties\uFF09\u7684applicationName(\u5E94\u7528\u7A0B\u5E8F\u540D)\u4E
00\u6837
3 spring.application.name=prosayconfig
4 eureka.client.service-url.defaultZone=http://127.0.0.1:8080/eureka
5 spring.cloud.config.profile=dev
6 spring.cloud.config.discovery.service-id=springcloud-config-server
7 spring.cloud.config.discovery.enabled=true
8
9 ####\u5F00\u542F\u76D1\u63A7\u65AD\u70B9
10 management.endpoints.web.exposure.include=bus-refresh
```

```
1 @RestController
2 @RefreshScope
3 public class ConfigClientController {
4
5     @Value("${prosayInfo}")
6     private String prosayInfo;
7
8     @GetMapping("/index")
9     public String index() {
10         return this.prosayInfo;
11     }
12
13 }
```

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class SpringcloudConfigClientApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringcloudConfigClientApplication.class, args);
7     }
8
9 }
```

- 将spring cloud bus整合在项目中
 - 在前面的三个项目中都加上相关依赖

```
1 <dependency>
2     <groupId>org.springframework.cloud</groupId>
3     <artifactId>spring-cloud-starter-bus-amqp</artifactId>
4 </dependency>
5
6 <dependency>
7     <groupId>org.springframework.boot</groupId>
8     <artifactId>spring-boot-starter-actuator</artifactId>
9 </dependency>
```

```
1 // post请求
2 http://127.0.0.1:8020/actuator/bus-refresh
```

十五.Spring Cloud Stream

15.1 概述

- 消息驱动式的一种微服务架构编程模式
- 简化消息中间件使用
- 自动配置化的功能可以简化开发，而且可以跟rabbitmq和kafka进行非常好的整合。屏蔽掉消息中间件的底层实现。我们只需关注业务即可。
- Stream组件对RabbitMQ和Kafka进行了一次统一的封装

```

1 Spring Cloud Stream is a framework for building highly scalable event-driven
microservices connected with shared messaging systems.
2
3 The framework provides a flexible programming model built on already
established and familiar Spring idioms and best practices, including support
for persistent pub/sub semantics, consumer groups, and stateful partitions.

```

- 官网
 - <https://github.com/spring-cloud/spring-cloud-stream>
 - <https://spring.io/projects/spring-cloud-stream>

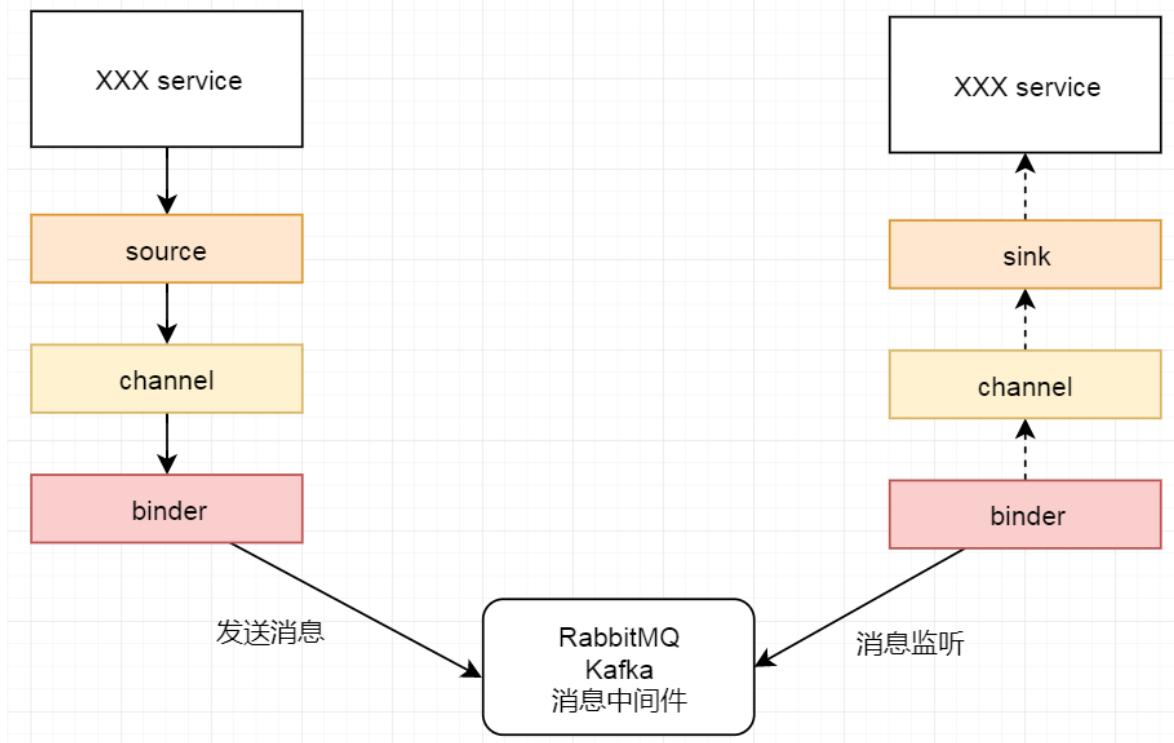
15.2 核心组件

15.2.1 Binder (绑定器)

- 实现应用程序与消息中间件进行的一个连接

消息驱动的原理：

1. 通过Binder将应用程序与消息中间件进行隔离
2. 通过向应用程序暴露统一的Channel,应用程序不需要关注使用哪一种消息中间件
3. 如果消息中间件发生了改变或版本升级，我们只需更新Binder即可，应用程序不需要改变



- source：将消息进行序列化操作。如果发送的消息是自定义对象，那么source组件就会将其序列化成json对象后然后再发送到channel。
- sink：从通道中获取消息将其反序列化成自定义对象，然后交给消息监听器处理业务
- channel：是Stream的抽象概念。向消息中间件发送消息或者监听消息时需要指定主题、队列名称。
- binder：是Stream的抽象概念。Binder可以绑定不同的消息中间件。通过Binder提供统一的消息接收和发送的接口，这样做就可以根据实际需要部署不同的消息中间件

15.3 搭建环境

15.3.1 搭建生产者

```

1 <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-
   starter-stream-rabbit -->
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
5 </dependency>
```

```

1 server.port=8888
2 spring.application.name=springcloud-stream-provider
3 #spring.rabbitmq.host=127.0.0.1
4 #spring.rabbitmq.port=5672
5 #spring.rabbitmq.username=admin
6 #spring.rabbitmq.password=admin
7 #spring.rabbitmq.virtual-host=prosay_virtualhost
8
9 spring.cloud.stream.bindings.prosayMsg.producer.partition-key-expression=payLoad
10 spring.cloud.stream.bindings.prosayMsg.producer.partition-count=2
```

```

1 public interface WriteMessageChannel {
2
3     @Output("prosayMsg")
4     SubscribableChannel sendMessage();
5 }
```

```

1 @RestController
2 public class MessageController {
3
4     @Autowired
5     private WriteMessageChannel writeMessageChannel;
6
7     @GetMapping("/sendMessage")
8     public String sendMessage() {
9         String message = "helloworld spring cloud stream";
10        Message<byte[]> msg =
11            MessageBuilder.withPayload(message.getBytes()).build();
12        writeMessageChannel.sendMessage().send(msg);
13        return "success";
14    }
15 }
```

```
1 @SpringBootApplication
2 @EnableBinding(WriteMessageChannel.class)
3 public class SpringcloudStreamProviderApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringcloudStreamProviderApplication.class, args);
7     }
8
9 }
```

15.3.2 消息的消费方

```
1 <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-
2 starter-stream-rabbit -->
3 <dependency>
4     <groupId>org.springframework.cloud</groupId>
5     <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
6 </dependency>
```

```
1
2 server.port=9999
3 spring.application.name=springcloud-stream-consumer
4
5 spring.cloud.stream.bindings.prosayMsg.group=stream
6 spring.cloud.stream.bindings.prosayMsg.consumer.partitioned=true
7 #总实例数
8 spring.cloud.stream.instance-count=2
9 #当前实例的索引号
10 spring.cloud.stream.instance-index=1
```

```
1 public interface ReadMessageChannel {
2
3     @Input("prosayMsg")
4     SubscribableChannel readMesage();
5 }
```

```
1 @Component
2 public class MessageConsumer {
3
4     @StreamListener(value = "prosayMsg")
5     public void readMessage(String message) {
6         System.out.println("消息者获取了消息: " + message);
7     }
8 }
```

```
1 @SpringBootApplication
2 @EnableBinding(value = ReadMessageChannel.class)
3 public class SpringcloudStreamConsumerApplication {
4
5     public static void main(String[] args) {
6         SpringApplication.run(SpringcloudStreamConsumerApplication.class, args);
7     }
8
9 }
```

十六. Spring Cloud Security

16.1 概述

- 微服务架构的**认证和授权**框架。还可以进行第三方认证OAuth2（QQ互联，微信认证...）,实现单点登录（SSO，JWT+CAS）
- Spring Security vs Shiro
- 认证：登录。根据用户的账号查询该用户，如果查询不到则直接返回null，抛出异常（未知账户异常）。如果查询得到这个用户再进行凭证匹配，如果不能匹配则抛出异常（错误的凭证异常）。如果匹配了则认证成功。
- 授权：资源的控制访问。认证通过后，能对什么资源进行操作的一个判断（控制）
- Spring Security是什么
 - 基于Spring框架企业应用的权限管理框架。类似于Apache Shiro
 - <https://spring.io/projects/spring-security>

```
1 Spring Security is a powerful and highly customizable authentication and
   access-control framework. It is the de-facto standard for securing Spring-
   based applications.
2
3 Spring Security is a framework that focuses on providing both
   authentication and authorization to Java applications. Like all Spring
   projects, the real power of Spring Security is found in how easily it can
   be extended to meet custom requirements
```

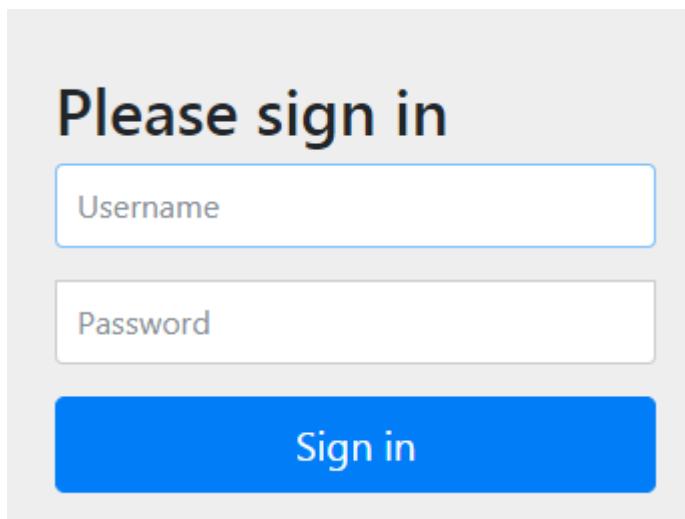
- 应用场景
 - 认证授权
 - OAuth2的授权认证
 - 安全防护（跨站点请求）

16.2 Spring Security认证模式

16.2.1 HttpBasic认证

- 浏览器(用户)与HTTP服务器之间的一种认证模式
- 密码使用了一种加密（BASE64）

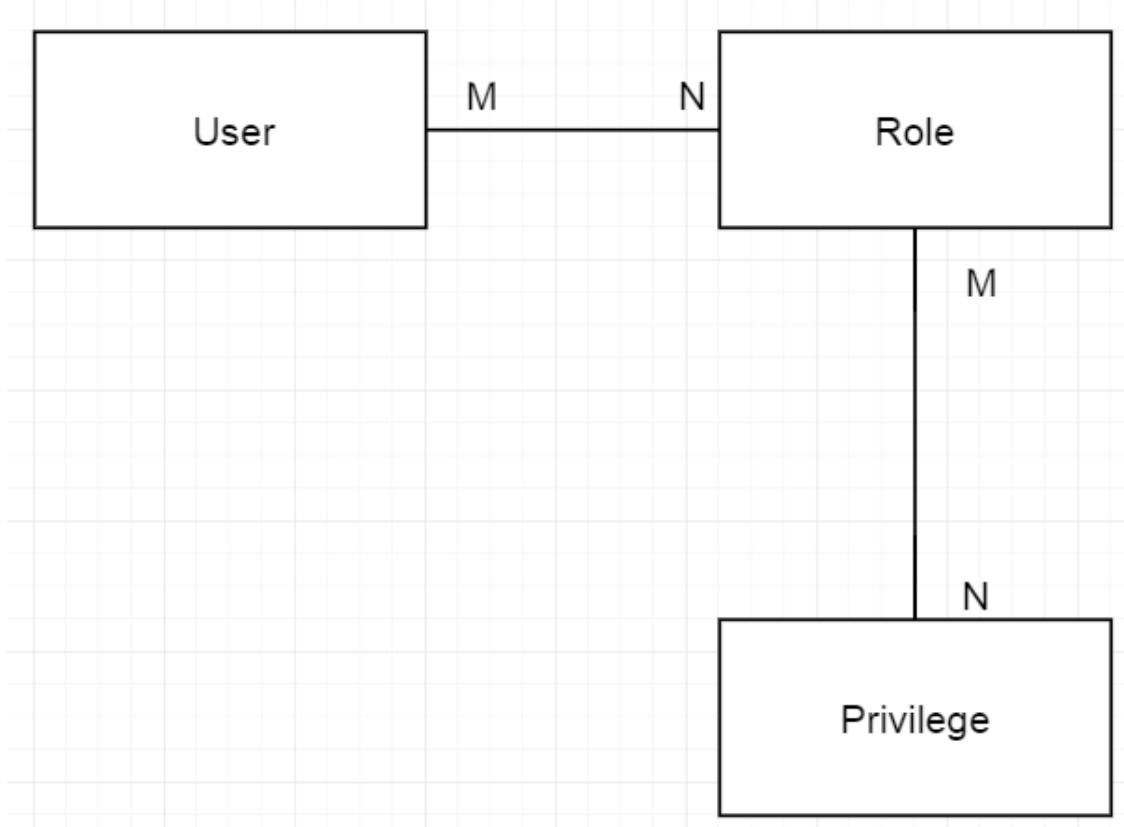
16.2.2 FormLogin认证



```
1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfig extends WebSecurityConfigurerAdapter {
4
5      /**
6       * 配置用户的信息和权限信息
7       *
8       * @param auth
9       * @throws Exception
10      */
11     @Override
12     protected void configure(AuthenticationManagerBuilder auth) throws Exception
13     {
14         // admin用户增删改查操作都能做
15
16         auth.inMemoryAuthentication().withUser("admin").password("123456").authorities(
17             "add", "delete", "update",
18             "findAll");
19
20
21         /**
22          * 配置拦截资源
23          *
24          * @param http
25          * @throws Exception
26          */
27         @Override
28         protected void configure(HttpSecurity http) throws Exception {
29             // 使用的是HttpBasic认证模式，这种方式不推荐
30
31             // http.authorizeRequests().antMatchers("/**").fullyAuthenticated().and().httpBasic();
32
33             http.authorizeRequests()
34                 .antMatchers("/add").hasAnyAuthority("add")
35                 .antMatchers("/delete").hasAnyAuthority("delete")
36                 .antMatchers("/update").hasAnyAuthority("update")
37                 .antMatchers("/findAll").hasAnyAuthority("findAll")
38                 .antMatchers("/login").permitAll()
39                 .antMatchers("/**").fullyAuthenticated()
40                 .and().formLogin().loginPage("/login")
41                 .and().csrf().disable();
42
43         }
44
45         @Bean
46         public static NoOpPasswordEncoder passwordEncoder() {
47             return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
48         }
49     }
50 }
```

16.2.3 SpringBoot + Spring Security + MyBatis

- 权限模型(RBAC 基于角色的访问控制)
 - User
 - Role
 - Privilege: 资源（模块，操作）。实际上是**无限极分类**（树状结构）
 - 类图如下



- ### ○ 关系模型：五张表

```
1  @Component
2  public class CustomUserDetailsService implements UserDetailsService {
3
4      @Autowired
5      private UserMapper userMapper;
6
7      @Override
8      public UserDetails loadUserByUsername(String username) throws
UsernameNotFoundException {
9          User user = null;
10
11         // 根据用户名查询该用户
12         UserExample userExample = new UserExample();
13         UserExample.Criteria criteria = userExample.createCriteria();
14         criteria.andUserNameEqualTo(username);
15
16         List<User> userList = this.userMapper.selectByExample(userExample);
17         if (!CollectionUtils.isEmpty(userList)) {
18             user = userList.get(0);
19         }
20
21         List<Privilege> privilegeList =
this.userMapper.findPrivilegesByUsername(username);
22         if (!CollectionUtils.isEmpty(privilegeList)) {
23             List<GrantedAuthority> grantedAuthorityList = new ArrayList<>();
24
25
26             for (Privilege privilege : privilegeList) {
```

```
27             grantedAuthorityList.add(new
28                 SimpleGrantedAuthority(privilege.getPrivilegeCode())));
29         }
30     user.setAuthorities(grantedAuthorityList);
31 }
32 }
33
34     return user;
35 }
36 }
```

```
1 @Configuration
2 @EnableWebSecurity
3 public class SecurityConfig extends WebSecurityConfigurerAdapter {
4
5     @Autowired
6     private CustomUserDetailsService customUserDetailsService;
7
8     @Autowired
9     private PrivilegeMapper privilegeMapper;
10
11
12     @Override
13     protected void configure(AuthenticationManagerBuilder auth) throws
Exception {
14
15         /*auth.userDetailsService(customUserDetailsService).passwordEncoder(new
PasswordEncoder()) {
16             @Override
17             public String encode(CharSequence rawPassword) {
18                 return null;
19             }
20
21             @Override
22             public boolean matches(CharSequence rawPassword, String
encodedPassword) {
23                 return false;
24             }
25         };*/
26
27         auth.userDetailsService(customUserDetailsService).passwordEncoder(this.passwo
rdEncoder());
28     }
29
30     @Override
31     protected void configure(HttpSecurity http) throws Exception {
32
33         ExpressionUrlAuthorizationConfigurer<HttpSecurity>.ExpressionInterceptUrlReg
istry authorizeRequests =
34             http.authorizeRequests();
35
36         // 查询出所有的权限
37         PrivilegeExample privilegeExample = new PrivilegeExample();
38         List<Privilege> privilegeList =
39             privilegeMapper.selectByExample(privilegeExample);
40         for (Privilege privilege : privilegeList) {
41
42             authorizeRequests.antMatchers(privilege.getPrivilegeUrl()).hasAnyAuthority(p
rivilege.getPrivilegeCode());
43         }
44     }
45 }
```

```
41     authorizeRequests.antMatchers("/login").permitAll().antMatchers("/**").fully
42             Authenticated()
43             .and().formLogin().loginPage("/login")
44             .and().csrf().disable();
45     }
46
47     @Bean
48     public static NoOpPasswordEncoder passwordEncoder() {
49         return (NoOpPasswordEncoder) NoOpPasswordEncoder.getInstance();
50     }
51
52 }
53
```