

## 面试问题

微服务架构与传统的单体应用架构有什么优缺点？

Springcloud 与dubbo 有什么区别？

什么是Spring cloud ， Springcloud 与springboot的关系？

Springcloud有哪些微服务开发组件，说说这些组件的作用？

eureka服务注册与发现的原理？

微服务架构改造的过程中你有遇到哪些比较困难的点？

# 1 微服务介绍

## 1.1 单体应用架构

定义：一个归档包（例如war格式包）含所有功能的应用程序，通常称为单体应用，而架构单体应用的方法论就是单体应用架构。

不足：

- 1、不利于敏捷开发和部署
- 2、随着应用越来越大，启动慢、降低开发效率
- 3、可靠性差，某个模块出问题可能会影响整个应用的运行
- 4、阻碍技术创新：对于单体应用来说，技术是在开发之前经过慎重评估后选定的，每个团队成员都必须使用相同的开发语言、持久化存储及消息系统。
- 5、系统愈加复杂，模块多，模块的边界模糊，依赖关系不清晰、代码难以理解、不利于维护。

## 1.2 微服务应用架构

**定义：**微服务架构是一种将一个单一应用程序开发为一组小型服务的方法，每个服务运行在自己的进程中，服务间通讯采用轻量级通信机制（通常用HTTP RESTFUL API），这些服务围绕业务能力构建，并且可通过全自动部署机制，独立部署，这些服务共用一个最小型的集中式的管理服务，可用不同的语言开发，使用不同的数据存储技术。

优点：

- 1、每个服务足够内聚、足够小，代码易于理解，降低每个服务开发的复杂度。
- 2、服务独立测试、部署、升级、发布；服务之间相互影响较小。
- 3、技术栈不受限，在微服务架构中，可以结合项目业务及团队的特点，合理选择技术栈。新技术的应用，系统不会被长期限制在某个技术栈上；
- 4、按需收缩：可根据需求，实现细粒度的扩展。例如，系统中的某个微服务遇到了瓶颈，可以结合这个微服务的业务特点，增加内存、升级CPU或者增加节点。

缺点：

- 1、增大了系统运维部署的复杂度
- 2、分布式开发提高了技术门槛，例如：微服务有各自的数据库，很难在不采用分布式事务的情况下跨服务实现功能。
- 3、服务和 service 之间通过接口来“联系”，当某一个服务更改接口格式时，可能涉及到此接口的所有服务都需要做调整。

## 1.3 微服务具有以下特点

### 1.3.1 特点概述

按业务划分为一个独立运行的程序，即服务单元。

服务之间通过 HTTP 协议相互通信。

可以用不同的存储技术。

自动化部署。

可以用不同的编程语言。

服务集中化管理。

微服务是一个分布式系统。

## 1.3.1 特点详解

### 微服务单元按业务来划分

微服务的“微”是按照业务来划分的。一个大的业务可以拆分成若干小的业务，一个小的业务又可以拆分成若干更小的业务，业务到底怎么拆分才算合适，这需要开发人员自己去决定。

### 微服务通过 HTTP 来互相通信

按照业务划分的微服务单元独立部署，并运行在各自的进程中。微服务单元之间的通信方式一般倾向于使用 HTTP 这种简单的通信机制，更多的时候是使用 RESTful API 的。这种接受请求、处理业务逻辑、返回数据的 HTTP 模式非常高效，并且这种通信机制与平台和语言无关。例如用 Java 写的服务可以消费用 Go 语言写的服务，用 Go 写的服务又可以消费用 Ruby 写的服务。不同的服务可以采用不同的语言去实现，不同的平台去部署，它们之间使用 HTTP 进行通信

### 可以用不同的存储技术

微服务的一个特点就是按业务划分服务，服务与服务之间无耦合，就连数据库也是独立的。一个典型的微服务的架构就是每个微服务都有自己独立的数据库，数据库之间没有任何联系。数据库独立，每个服务的数据量少，易于维护，性能也有明显的优势。另外，随着存储技术的发展，数据库的存储方式不再仅仅是关系型数据库，非关系数据库的应用也非常广泛，例如 MongoDB、Redis，它们有着良好的读写性能，因此越来越受欢迎。一个典型的微服务的系统，可能每一个服务的数据库都不相同，每个服务所使用的数据存储技术需要根据业务需求来选择，

### 微服务的自动化部署

在微服务架构中，系统会被拆分为若干个微服务，每个微服务又是一个独立的应用程序。单体架构的应用程序只需要部署一次，而微服务架构有多少个服务就需要部署多少次。随着服务数量的增加，如果微服务按照单体架构的部署方式，部署的难度会呈指数增加。业务的粒度划分得越细，微服务的数量就越多，这时需要更稳定的部署机制。随着技术的发展，尤其是 Docker 容器技术的推进，以及自动化部署工具（例如开源组件 Jenkins）的出现，自动化部署变得越来越简单。

## 服务集中化管理

微服务系统是按业务单元来划分服务的，服务数量越多，管理起来就越复杂，因此微服务必须使用集中化管理。目前流行的微服务框架中，例如 Spring Cloud 采用 Eureka 来注册服务和发现服务，另外，Zookeeper、Consul 等都是非常优秀的服务集中化管理框架。

## 分布式架构

分布式系统是集群部署的，由很多计算机**相互协作共同**构成，它能够处理海量的用户请求。当分布式系统对外提供服务时，用户是毫不知情的，还以为是一台服务器在提供服务。

微服务架构是分布式架构，分布式系统比单体系统更加复杂，主要体现在服务的独立性和服务相互调用的可靠性，以及分布式事务、全局锁、全局 Id 等，而单体系统不需要考虑这些复杂性。

在分布式系统中，服务之间相互依赖，如果一个服务出现了故障或者是网络延迟，在高并发的情况下，会导致线程阻塞，在很短的时间内该服务的线程资源会消耗殆尽，最终使得该服务不可用。由于服务的相互依赖，可能会导致整个系统的不可用，这就是“雪崩效应”。

# 2 微服务开发框架

## 2.2 Spring Cloud 与Dubbo的比较

- 1、Dubbo 只是实现了服务治理，而 Spring Cloud 子项目分别覆盖了微服务架构下的众多部件，服务治理只是其中的一个方面。
- 2、Dubbo 支持各种通信协议，而且消费方和服务方使用长链接方式交互，**通信速度上略胜 Spring Cloud**，如果对于系统的响应时间有严格要求，长链接更合适。
- 3、Dubbo调用方应用对微服务提供的抽象接口存在强依赖关系，开发、测试、集成环境都需要严格的管理版本依赖。而Spring Cloud 通过 Json 交互，省略了版本管理的问题，但是具体字段含义需要统一管理，自身 Rest API 方式交互，为跨平台调用奠定了基础。

详细查看博文

<https://www.cnblogs.com/xingzc/p/9013804.html>

## 2.2 Spring Cloud简介

Spring Cloud是在SpringBoot基础上构建的，用于快速构建分布式系统的通用模式的工具集。

Spring Cloud 的首要目标就是通过提供一系列开发组件和框架，帮助开发者迅速搭建一个分布式的微服务系统。Spring Cloud 是通过包装其他技术框架来实现的，例如包装开源的 Netflix oss 组件，实现了一套通过基于注解、Java 配置和基于模版开发的微服务框架。Spring Cloud框架来自于Spring Resouces 社区，由 Pivotal 和 Netflix 两大公司和一些其他的开发者提供技术上的更新迭代。Spring Cloud 提供了开发分布式微服务系统的一些常用组件，例如服务注册和发现、配置中心、熔断器、智能路由、微代理、控制总线、全局锁、分布式会话等。

## 2.2 Spring Cloud常用组件

### (1) 服务注册和发现组件 Eureka

利用 Eureka 组件可以很轻松地实现服务的注册和发现的功能。Eureka 组件提供了服务的健康

监测，以及界面友好的 UI。通过 Eureka 组件提供的 UI, Eureka 组件可以让开发人员随时了

解服务单元的运行情况。另外 Spring Cloud 也支持 Consul 和 Zookeeper，用于注册和发现服务。

### (2) 熔断组件 Hystrix

Hystrix 是一个熔断组件，它除了有一些基本的熔断器功能外，还能够实现服务降级、服务限流的功能。另外 Hystrix 提供了熔断器的健康监测，以及熔断器健康数据的 API 接口。

Hystrix Dashboard 组件提供了单个服务熔断器的健康状态数据的界面展示功能，Hystrix

Turbine 组件提供了多个服务的熔断器的健康状态数据的界面展示功能。

### (3) 负载均衡组件 Ribbon

Ribbon 是一个负载均衡组件，它通常和 Eureka、Zuul、RestTemplate、Feign 配合使用。

Ribbon 和 Zuul 配合，很容易做到负载均衡，将请求根据负载均衡策略分配到不同的服务实例

中。Ribbon 和 RestTemplate、Feign 配合，在消费服务时能够做到负载均衡。

#### (4) 路由网关 Zuul

路由网关 Zuul 有智能路由和过滤的功能。内部服务的 API 接口通过 Zuul 网关统一对外暴露，

内部服务的 API 接口不直接暴露，防止了内部服务敏感信息对外暴露。在默认的情况下，Zuul 和 Ribbon 相结合，能够做到负载均衡、智能路由。Zuul 的过滤功能是通过拦截请求来实现的，可以

对一些用户的角色和权限进行判断，起到安全验证的作用，同时也可以用于输出实时的请求日志。

#### (5) Spring Cloud Config

Spring Cloud Config 组件提供了配置文件统一管理的功能。Spring Cloud Config 包括 Server

端和 Client 端，Server 端读取本地仓库或者远程仓库的配置文件，所有的 Client 向 Server 读

取配置信息，从而达到配置文件统一管理的目的。通常情况下，Spring Cloud Config 和 Spring

Cloud Bus 相互配合刷新指定 Client 或所有 Client 的配置文件。

#### (6) Spring Cloud Security

Spring Cloud Security 是对 Spring Security 组件的封装，Spring Cloud Security 向服务单元

提供了用户验证和权限认证。一般来说，单独在微服务系统中使用 Spring Cloud Security 是很

少见的，一般它会配合 Spring Security OAuth2 组件一起使用，通过搭建授权服务，验证 Token

或者 JWT 这种形式对整个微服务系统进行安全验证。

#### (7) Spring Cloud Sleuth

Spring Cloud Sleuth 是一个分布式链路追踪组件，它封装了 Dapper、Zipkin 和 Kibana 等组

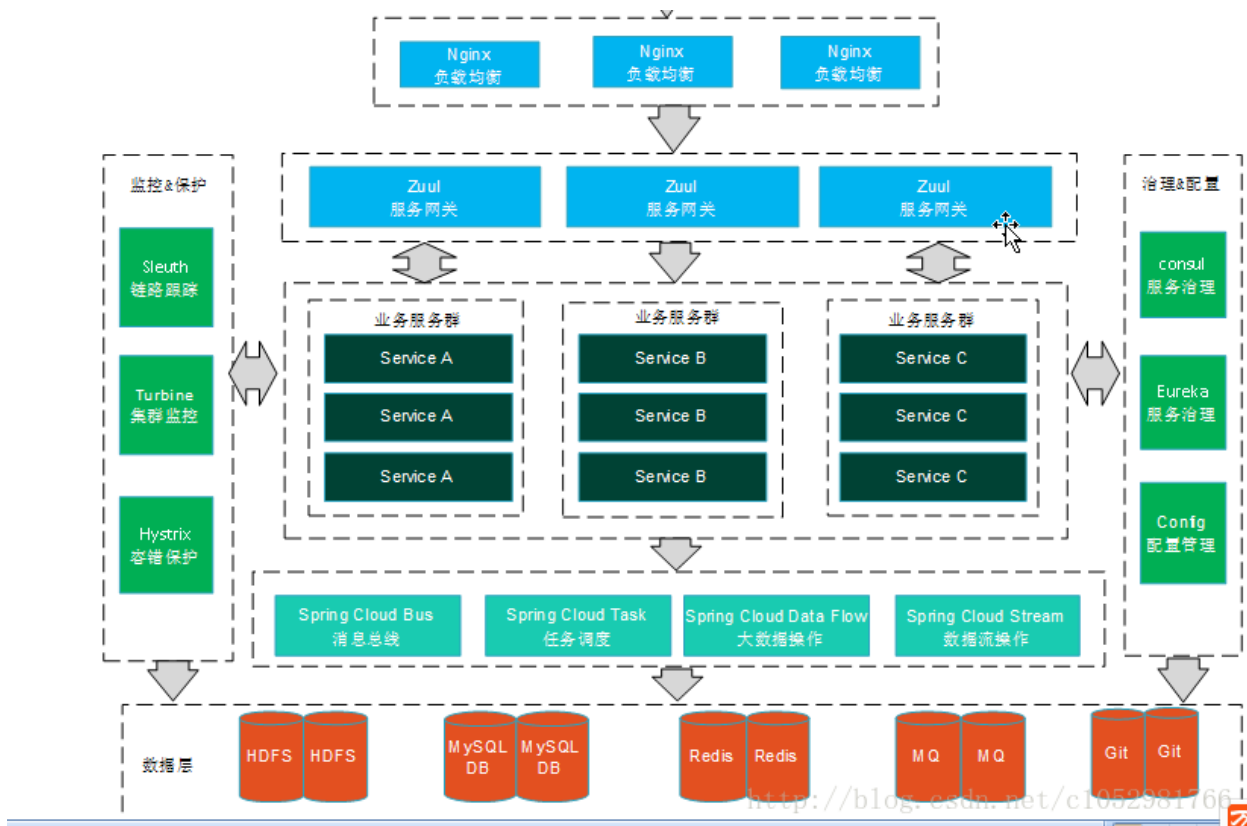
件，通过它可以知道服务之间的相互依赖关系，并实时观察链路的调用情况。

#### (8) Spring Cloud Stream

Spring Cloud Stream 是 Spring Cloud 框架的数据流操作包，可以封装 RabbitMq、ActiveMq、

Kafka、Redis 等消息组件，利用 Spring Cloud Stream 可以实现消息的接收和发送。

## Spring Cloud 微服务架构图



## 2.3 Spring Cloud 版本命名

Spring Cloud是一个由众多独立子项目组成的大型综合项目，每个子项目有不同的发行节奏，都维护着自己的发布版本号。Spring Cloud通过一个资源清单BOM（Bill of Materials）来管理每个版本的子项目清单。为避免与子项目的发布号混淆，所以没有采用版本号的方式，而是通过命名的方式。

这些版本名称的命名方式采用了伦敦地铁站的名称，同时根据字母表的顺序来对应版本时间顺序，比如：最早的Release版本：Angel，第二个Release版本：Brixton，然后是Camden、Dalston、Edgware，目前最新的是Finchley版本。

当一个版本的Spring Cloud项目的发布内容积累到临界点或者解决了一个严重bug后，就会发布一个“service releases”版本，简称SRX版本，其中X是一个递增数字。当前官网上最新的稳定版本是Edgware.SR3，里程碑版本是Finchley.M9。下表列出了这两个版本所包含的子项目及各子项目的版本号。

## 3 微服务工程搭建

### 3.1 工程结构

hshop-common-all （公共类工程）

---com.heima.hshop.common.all.exception

---com.heima.hshop.common.all.utils

hshop-service-api-goods （商品服务接口工程）

---com.heima.hshop.service.api.goods.api

---com.heima.hshop.service.api.goods.exception

---com.heima.hshop.service.api.goods.model.req

---com.heima.hshop.service.api.goods.model.res

hshop-service-impl-goods （商品服务实现工程）

---com.heima.hshop.service.impl.goods.biz

---com.heima.hshop.service.impl.goods.dao

---com.heima.hshop.service.impl.goods.po

---com.heima.hshop.service.impl.goods.ctl

hshop-service-api-stock （库存服务接口工程）

---com.heima.hshop.service.api.stock.api

---com.heima.hshop.service.api.stock.exception

---com.heima.hshop.service.api.stock.model.req

---com.heima.hshop.service.api.stock.model.res

hshop-service-impl-stock （库存服务实现工程）

---com.heima.hshop.service.impl.stock.biz

---com.heima.hshop.service.impl.stocks.dao

---com.heima.hshop.service.impl.stock.po

---com.heima.hshop.service.impl.stock.ctl

hshop-service-api-order （订单服务接口工程）



```
---com.heima.hshop.service.api.order.api
---com.heima.hshop.service.api.order.exception
---com.heima.hshop.service.api.order.model.req
---com.heima.hshop.service.api.order.model.res
```

hshop-service-impl-order (订单服务实现工程)

```
---com.heima.hshop.service.impl.order.biz
---com.heima.hshop.service.impl.order.dao
---com.heima.hshop.service.impl.order.po
---com.heima.hshop.service.impl.order.ctl
```

## 3.2 创建工程

### 3.2.1 创建父工程hshop-parent

hshop-parent pom.xml配置

```
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 http://maven.apach
5     e.org/xsd/maven-4.0.0.xsd">
6
7   <parent>
8     <groupId>org.springframework.boot</groupId>
9     <artifactId>spring-boot-starter-parent</artifactId>
10    <version>2.0.1.RELEASE</version>
11  </parent>
12  <groupId>com.heima</groupId>
13  <artifactId>hshop-parent</artifactId>
14  <version>1.0-SNAPSHOT</version>
15  <packaging>pom</packaging>
16
17
18
19  <dependencyManagement>
20    <dependencies>
21      <dependency>
22        <groupId>org.springframework.cloud</groupId>
```

```
23 <artifactId>spring-cloud-dependencies</artifactId>
24 <version>Finchley.SR1</version>
25 <type>pom</type>
26 <scope>import</scope>
27 </dependency>
28
29 </dependencies>
30 </dependencyManagement>
31
32
33 </project>
```

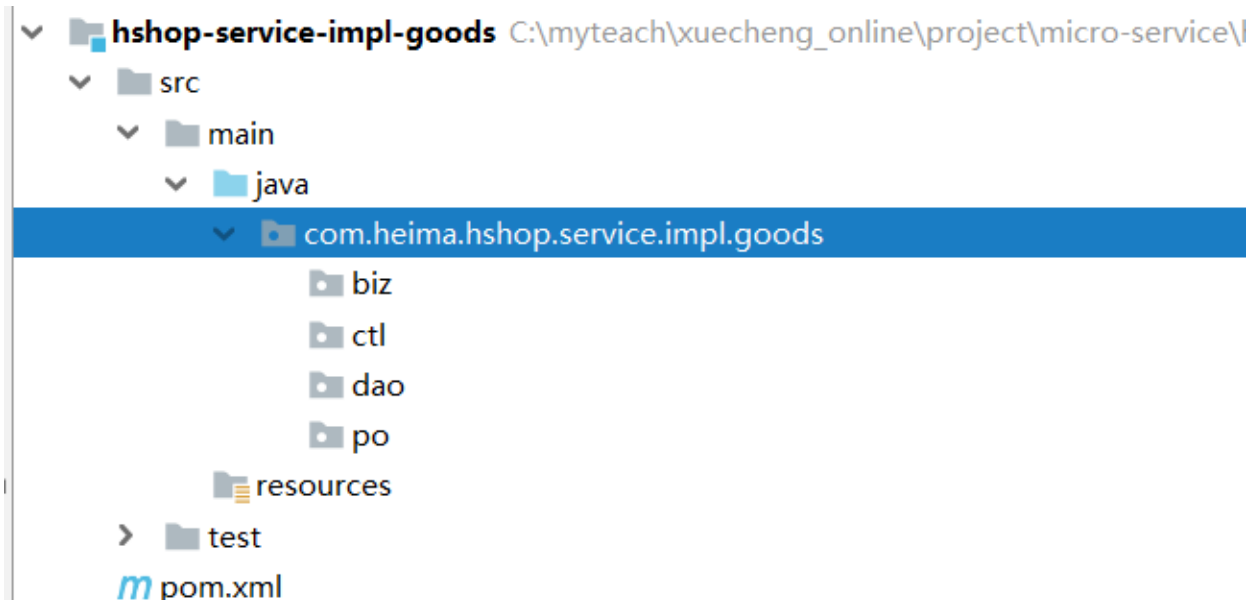
### 3.2.2 分别创建其它工程

各工程继承父工程hshop-parent

pom.xml示例 (hshop-common-public)

```
1
2 <?xml version="1.0" encoding="UTF-8"?>
3 <project xmlns="http://maven.apache.org/POM/4.0.0"
4   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <parent>
8     <groupId>com.heima</groupId>
9     <artifactId>hshop-parent</artifactId>
10    <version>1.0-SNAPSHOT</version>
11  </parent>
12  <groupId>com.heima</groupId>
13  <artifactId>hshop-common-all</artifactId>
14  <version>1.0-SNAPSHOT</version>
15
16
17 </project>
```

- ▼ **hshop-common-all** C:\myteach\xuecheng\_online\project\micro-service
  - ▼ **src**
    - ▼ **main**
      - ▼ **java**
        - ▼ **com.heima.hshop.common.all**
          - exception**
          - util**
          - web**
        - resources**
      - ▼ **test**
    - java**
    - pom.xml**
  - ▼ **hshop-parent** C:\myteach\xuecheng\_online\project\micro-service\hshop-parent
    - > **src**
    - pom.xml**
  - ▼ **hshop-service-api-goods** C:\myteach\xuecheng\_online\project\micro-service-api-goods
    - ▼ **src**
      - ▼ **main**
        - ▼ **java**
          - ▼ **com.heima.hshop.service.api.goods**
            - api**
            - ▼ **model**
              - req**
              - res**
          - resources**
        - > **test**
        - pom.xml**



详情结构请查看源码 [micro-service](#)

## 3.3创建表

**注意：**以下表只是用来演示，字段不全

商品表

```
1 CREATE TABLE `goods_info` (  
2   `product_id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '商品ID',  
3   `product_name` varchar(60) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '商品名称',  
4   `price` int(11) DEFAULT NULL COMMENT '商品销售价格',  
5   `description` varchar(255) CHARACTER SET utf8mb4 COLLATE utf8mb4_general_ci DEFAULT NULL COMMENT '商品描述',  
6   PRIMARY KEY (`product_id`)  
7 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci
```

订单主表

```
1 CREATE TABLE `order_primary` (  
2   `order_id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '订单ID',
```

```

3   `order_money` int(11) DEFAULT NULL COMMENT '订单金额',
4   `order_status` tinyint(4) DEFAULT NULL COMMENT '订单状态',
5   PRIMARY KEY (`order_id`)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci COMMENT='订单主表'

```

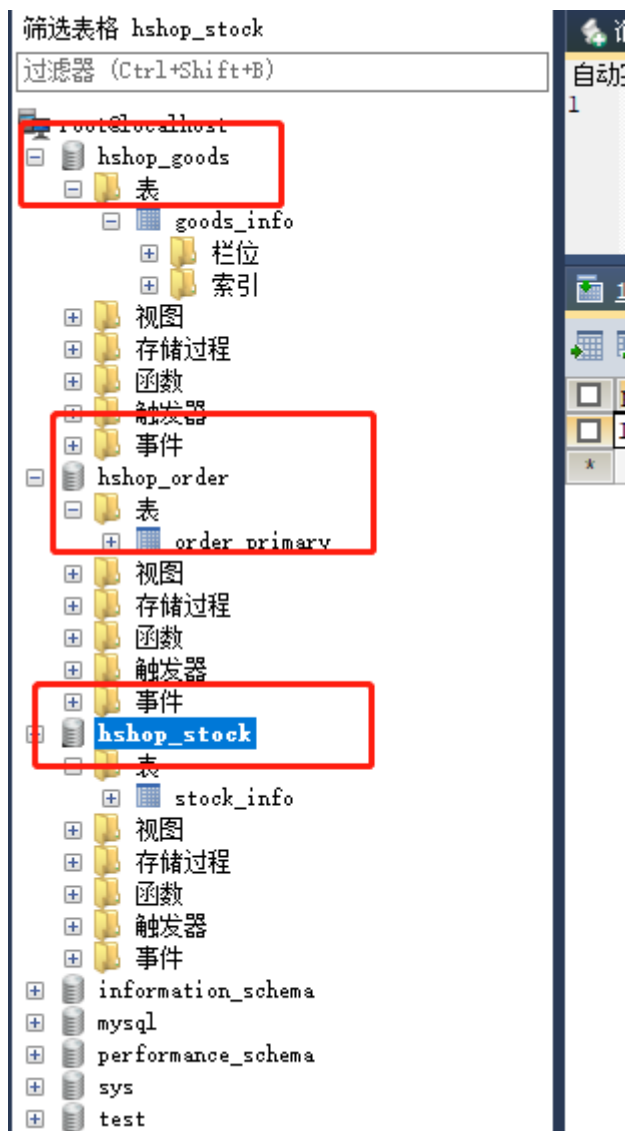
## 库存表

```

1 CREATE TABLE `stock_info` (
2   `wp_id` int(11) NOT NULL COMMENT '商品库存ID',
3   `product_id` int(11) NOT NULL COMMENT '商品ID',
4   `product_amount` int(11) DEFAULT NULL COMMENT '当前商品数量',
5   PRIMARY KEY (`wp_id`)
6 ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci COMMENT='商品库存表'

```

## 分别在各个库中创建对应的表



# 4 微服务接口开发

## 4.1概念：服务提供者与消费者

使用微服务构建的分布式系统，服务之间通过网络进行通讯，我们使用服务提供者与服务消费者来描述服务之间的调用关系。

**服务提供者**：服务的被调用方，即为其他服务提供服务的服务。

**服务消费者**：服务的调用方，既依赖其他服务的的服务。

## 4.2 编写服务提供者（开发获取商品信息接口）

### 1、定义接口响应模型

```
1
2 public class GetGoodsByIdRes {
3
4     /**商品id*/
5     private Integer productId;
6
7     /**商品名称*/
8     private String productName;
9
10    /**销售价格*/
11    private Integer price;
12
13    /**商品描述*/
14    private String description;
15
16
17 }
```

### 2、定义接口请求模型

```
1
2 public class GetGoodsByIdReq {
3
4     private Integer productId;
5
6 }
```

### 3、定义获取商品信息接口

```

1
2 public interface GoodsInfoCtlApi {
3
4     /**
5      * @description 根据商品id查询商品信息
6      * @author Xuejun Yang
7      * @date 2019/5/3 13:23
8      * @param getGoodsByIdReq
9      * @return
10     */
11     public Result<GetGoodsByIdRes> getGoodsById( Integer productId);
12 }
13
14

```

#### 4、编写实体类

```

1 public class GoodsInfo {
2
3     @Id
4     @Column(name = "product_id")
5     @GeneratedValue(strategy = GenerationType.IDENTITY)
6     private Integer productId;
7
8     @Column(name = "product_name")
9     private String productName;
10
11     @Column(name = "price")
12     private Integer price;
13
14     @Column(name = "description")
15     private String description;
16 }

```

#### 5、编写DAO

```

1
2 @Repository
3 public interface GoodsInfoRepository extends JpaRepository<GoodsInfo,Integer> {
4
5

```

```
6
7 }
```

## 5、编写服务类BIZ(Service)

```
1 @Service
2 public class GoodsInfoBiz {
3
4     @Autowired
5     private GoodsInfoRepository goodsInfoRepository;
6
7     public GetGoodsByIdRes getGoodsById(Integer productId){
8
9         Optional<GoodsInfo> goodsInfo = goodsInfoRepository.findById(productId);
10        if(goodsInfo.isPresent()){
11            GetGoodsByIdRes getGoodsByIdRes=new GetGoodsByIdRes();
12            BeanUtils.copyProperties(goodsInfo.get(),getGoodsByIdRes);
13            return getGoodsByIdRes;
14        }else{
15            return null;
16        }
17    }
18
19
20 }
```

## Controller

```
1 @RestController
2 @RequestMapping("/goods/goods_info")
3 public class GoodsInfoCtl implements GoodsInfoCtlApi {
4
5     @Autowired
6     GoodsInfoBiz goodsInfoBiz;
7
8     @RequestMapping("/{id}")
9     @Override
10    public GetGoodsByIdRes getGoodsById(@PathVariable("id") Integer product
11    Id) {
```



```
11     return goodsInfoBiz.getGoodsById(productId);
12 }
13 }
```

## 启动类

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

## 4.3编写服务消费者（下单接口开发）

### 1、编写接口响应模型

```
1 public class AddOrderRes {
2
3     private Integer orderId;
4
5     private Integer orderMoney;
6
7     private Integer orderStatus;
8
9     private Integer orderNum;
10
11
12 }
```

### 2、编写接口请求模型

```
1 public class AddOrderReq {
2
3     private Integer productId;
4
5     private Integer num;
6 }
```

```
7
8 }
```

### 3、定义下单接口

```
1
2 public interface OrderCtlApi {
3
4     /**
5      * @description 下单
6      * @author Xuejun Yang
7      * @date 2019/6/25 17:39
8      * @return
9      */
10    public Result<AddOrderRes> addOrder(AddOrderReq addOrderReq);
11
12 }
```

### 4、编写实体类

```
1 Entity
2 @Table(name = "order_primary")
3 public class OrderPrimary {
4
5     @Id
6     @GeneratedValue(strategy = GenerationType.IDENTITY)
7     @Column(name = "order_id")
8     private Integer orderId;
9     @Column(name = "order_money")
10    private Integer orderMoney;
11    @Column(name = "order_status")
12    private Integer orderStatus;
13    @Column(name = "order_num")
14    private String orderNum;
15 }
```

### 5、编写dao接口OrderPrimaryRepository

```
1 /**
2  * @author Xuejun Yang
3  * @version V1.0
4  * @description: TODO
```

```

5  * @date 2019/6/25 18:09
6  */
7  public interface OrderPrimaryRepository extends JpaRepository<OrderPrimary, Integer> {
8  }

```

## 6、编写服务类OrderBiz

```

1  @Service
2  public class OrderBiz {
3
4
5  @Autowired
6  private OrderPrimaryRepository orderPrimaryRepository;
7
8  @Autowired
9  RestTemplate restTemplate;
10
11  /**
12   * @description 下单
13   * @author Xuejun Yang
14   * @date 2019/6/25 18:00
15   * @param addOrderReq
16   * @return
17   */
18  public AddOrderRes addOrder(AddOrderReq addOrderReq) {
19
20  //1、查询商品信息
21  GetGoodsByIdRes getGoodsByIdRes = restTemplate.getForObject("http://localhost:8080/goods/goods_info/"+addOrderReq.getProductId(),
22  GetGoodsByIdRes.class);
23
24  if(getGoodsByIdRes==null){
25  throw new RuntimeException("商品不存在");
26  }
27
28  OrderPrimary orderPrimary=new OrderPrimary();
29  orderPrimary.setOrderNum(UUID.randomUUID().toString());
30  orderPrimary.setOrderStatus(1);
31  orderPrimary.setOrderMoney(getGoodsByIdRes.getPrice()*addOrderReq.getNum());

```

```

31  orderPrimaryRepository.save(orderPrimary);
32
33  AddOrderRes addOrderRes=new AddOrderRes();
34  BeanUtils.copyProperties(orderPrimary,addOrderRes);
35  return addOrderRes;
36  }
37
38  }

```

## 7、编写Controller

```

1  @RestController
2  @RequestMapping("/order/order")
3  public class OrderCtl implements OrderCtlApi {
4
5
6  @Autowired
7  private OrderBiz orderBiz;
8
9  /**
10   * @description 下单
11   * @author Xuejun Yang
12   * @date 2019/6/25 17:58
13   * @return
14   */
15  @PostMapping("")
16  @Override
17  public Result<AddOrderRes> addOrder(AddOrderReq addOrderReq) {
18
19  orderBiz.addOrder(addOrderReq);
20  return new Result<AddOrderRes>();
21  }
22  }

```

## 8、启动类

```

1  @SpringBootApplication
2  public class OrderServiceApp {
3
4  @Bean
5  public RestTemplate restTemplate(){
6  return new RestTemplate();

```

```
7   }  
8  
9   public static void main(String[] args) {  
10    SpringApplication.run(OrderServiceApp.class);  
11  }  
12  
13 }
```

## 4.4 思考

服务调用地址硬编码，如果服务提供者的网络地址发生了变化，将会影响服务消费者的调用。

# 5 Eureka 服务注册与发现

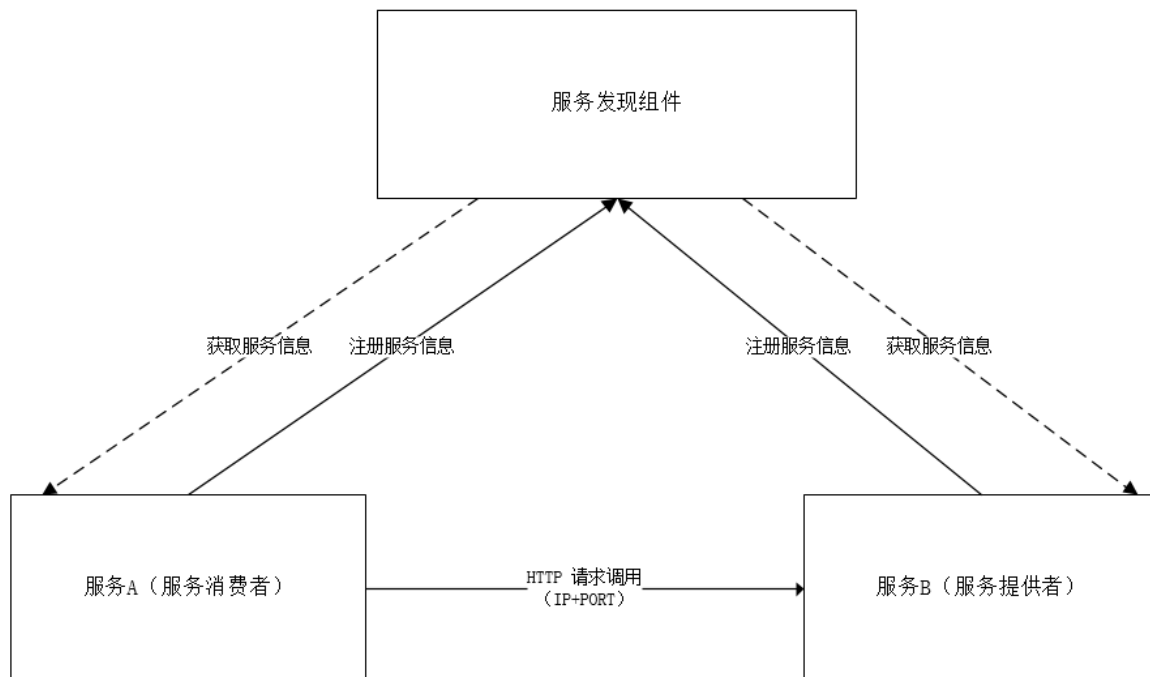
## 5.1 什么是服务注册与发现？

微服务将传统的"巨石"应用拆分成一个一个的组件应用，每个组件应用提供特定的服务，可以是一个，也可以是多个，并且组件所含服务应该是可以动态扩展的，随着时间推移、系统进化，可任意拆分、合并。

组件化应用和颗粒化的服务，遍布在系统的各个角落，由不同的项目成员进行维护，微服务的核心是化整为零、各司其职，这就要求开发人员不得操作其业务或服务范围以外的数据模型等资源，只能通过接口的访问，使用某一服务。

由于服务的跨度很大（公司很大的情况下）、数量很多（数以百计甚至更多），为保障系统的正常运行，必然需要有一个中心化的组件完成对各个服务的整合，即将分散于各处的服务进行汇总，汇总的信息可以是提供服务的组件名称、地址、数量等，每个组件拥有一个监听设备，当本组件内的某个服务的状态变化时报告至中心化的组件进行状态的更新。服务的调用方在请求某项服务时首先到中心化组件获取可提

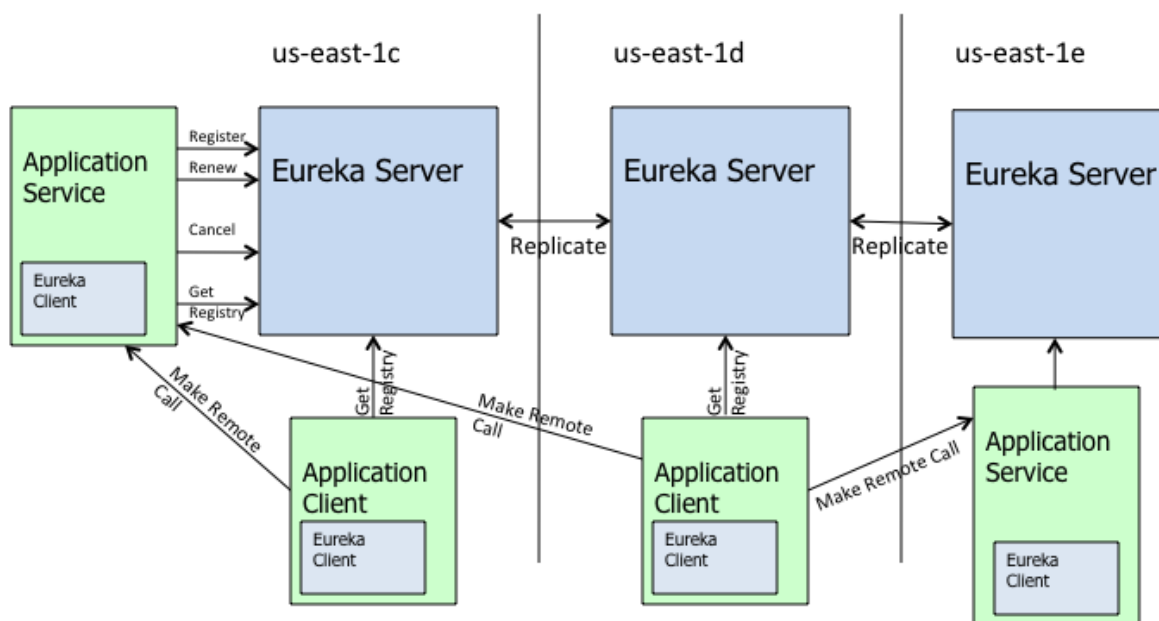
供该项服务的组件信息（IP、端口等），通过默认或自定义的策略选择该服务的某一提供者进行访问，实现服务的调用。



**Spring Cloud 中常见的 服务发现组件：** Zookeeper、ConSul、Eureka

Eureka包括两个端：

- Eureka Server：注册中心服务端，用于维护和管理注册服务列表。
- Eureka Client：注册中心客户端，向注册中心注册服务的应用都可以叫做Eureka Client（包括Eureka Server本身）。



<https://blog.csdn.net/waterson>

## 重点掌握eureka运行原理详细说明查看博文

<https://blog.csdn.net/waterson/article/details/81675259>

## 5.2 编写Eureka Server

### 1、创建工程demo-register-center

pom.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.
  w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apach
  e.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>com.heima</groupId>
7     <artifactId>hshop-parent</artifactId>

```

```
8 <version>1.0-SNAPSHOT</version>
9 </parent>
10 <artifactId>demo-register-center</artifactId>
11 <version>0.0.1-SNAPSHOT</version>
12
13 <properties>
14 <java.version>1.8</java.version>
15 </properties>
16
17
18
19
20 </project>
```

## 2、添加依赖包

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

## 3、编写启动类

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

## 4、配置文件

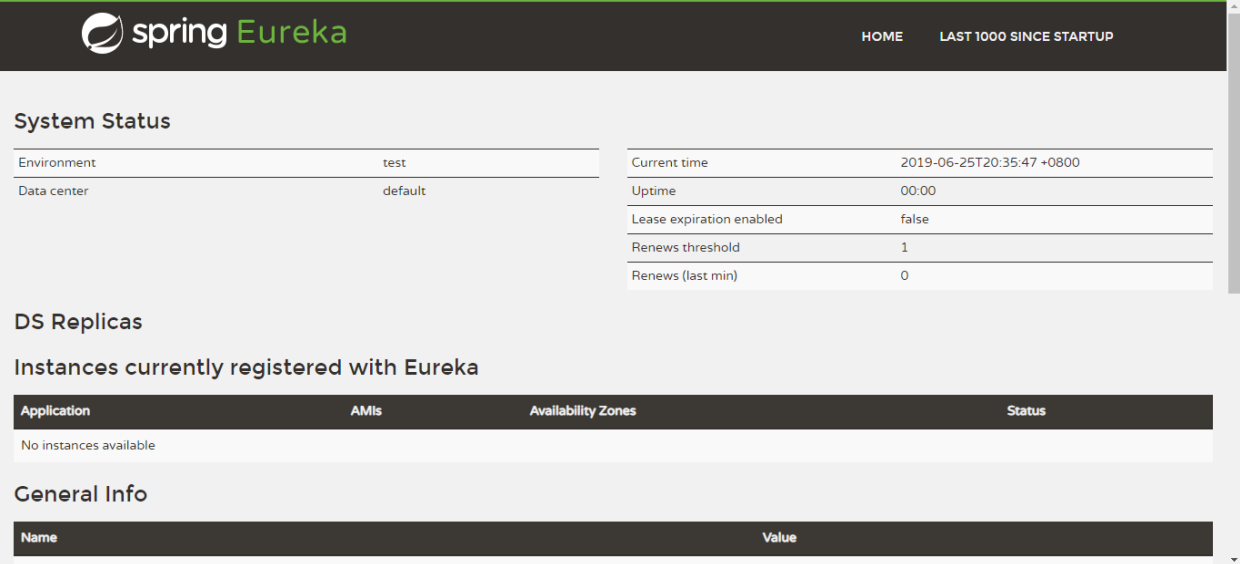
```
1
2 server:
3   port: 9000
4 eureka:
5   client:
6     fetch-registry: false
```



```
7 register-with-eureka: false
8 service-url:
9 defaultZone: http://localhost:9000/eureka/
10
```

## 5、访问

<http://localhost:9000/>



The screenshot shows the Spring Eureka web interface. The header includes the Spring Eureka logo and navigation links for HOME and LAST 1000 SINCE STARTUP. The main content area is divided into several sections:

- System Status:** A table showing environment details (Environment: test, Data center: default) and system metrics (Current time: 2019-06-25T20:35:47 +0800, Uptime: 00:00, Lease expiration enabled: false, Renews threshold: 1, Renews (last min): 0).
- DS Replicas:** A section for distributed system replicas.
- Instances currently registered with Eureka:** A table with columns for Application, AMIs, Availability Zones, and Status. It currently shows "No instances available".
- General Info:** A section for general information with a table for Name and Value.

## 5.3配置属性说明

register-with-eureka: 是否向服务注册中心注册自己

eureka.client.fetch-registry: 是否检索服务

eureka.client.serviceUrl.defaultZone: eureka sever 注册地址

更多请参考博文: <https://www.cnblogs.com/liukaifeng/p/10052594.html>

eureka zone和region的使用

<https://www.cnblogs.com/junjiang3/p/9061867.html>

## 5.4注册服务至Eureka Server

复制工程hshop-service-impl-order、hshop-service-impl-goods 为 demo-eureka-consumer-service、demo-eureka-provider-service

## 1、为两个工程服务引入jar包

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

## 2、为两个服务启动类添加@EnableEurekaClient

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

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 public class EurekaConsumerServiceApp {
4
5   @Bean
6   public RestTemplate restTemplate(){
7     return new RestTemplate();
8   }
9
10  public static void main(String[] args) {
11    SpringApplication.run(EurekaConsumerServiceApp.class);
12  }
13
14 }
```

## 5、为两个服务追加添加配置信息

```
1 spring:
2   application:
3     name: demo-eureka-provider-service
4   eureka:
5     client:
6       service-url:
7         defaultZone: http://localhost:9000/eureka
```

```
1 spring:
2   application:
3     name: demo-eureka-consumer-service
4   eureka:
5     client:
6       service-url:
7         defaultZone: http://localhost:9000/eureka
```

## 6、查看注册中心

出现两个服务实例

Renews threshold	5
Renews (last min)	0

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
DEMO-EUREKA-CONSUMER-SERVICE	n/a (1)	(1)	UP (1) - localhost:demo-eureka-consumer-service:8081
DEMO-EUREKA-PROVIDER-SERVICE	n/a (1)	(1)	UP (1) - localhost:demo-eureka-provider-service:8083

General Info

Name	Value
total-avail-memory	525mb
environment	test
num-of-cpus	8
current-memory-usage	362mb (68%)
server-uptime	00:01

## 5.5 通过服务名调用远程服务

修改工程demo-eureka-consumer-service

### 1、为OrderBiz类注入DiscoveryClient对象

```
1 @Autowired
2 DiscoveryClient discoveryClient;
3
```

### 2、修改OrderBiz类的addOrder方法

```
1 public AddOrderRes addOrder(AddOrderReq addOrderReq) {
```

```

2  List<ServiceInstance> instances = discoveryClient.getInstances("demo-eureka-provider-service");
3  String addr=null;
4  for (ServiceInstance instanceInfo: instances ) {
5      String ipAddr = instanceInfo.getHost();
6      int port = instanceInfo.getPort();
7      addr=ipAddr+": "+port;
8      System.out.println("addr:"+addr);
9  }
10
11  //1、查询商品信息
12  GetGoodsByIdRes getGoodsByIdRes = restTemplate.getForObject("http://"+addr+"/goods/goods_info/"+addOrderReq.getProductId(),
    GetGoodsByIdRes.class);
13
14  if(getGoodsByIdRes==null){
15      throw new RuntimeException("商品不存在");
16  }
17
18  OrderPrimary orderPrimary=new OrderPrimary();
19  orderPrimary.setOrderNum(UUID.randomUUID().toString());
20  orderPrimary.setOrderStatus(1);
21  orderPrimary.setOrderMoney(getGoodsByIdRes.getPrice()*addOrderReq.getNum());
22  orderPrimaryRepository.save(orderPrimary);
23
24  AddOrderRes addOrderRes=new AddOrderRes();
25  BeanUtils.copyProperties(orderPrimary,addOrderRes);
26  return addOrderRes;
27 }

```

3、启动运行demo-eureka-consumer-service和demo-eureka-provider-service, 请求<http://localhost:8081/order/order?productId=1&num=2>

## 6 负载均衡Ribbon

Spring Cloud Ribbon是一个基于HTTP和TCP的客户端负载均衡工具，它基于Netflix Ribbon实现。通过Spring Cloud的封装，可以让我们轻松地将面对服务的REST模块请求自动转换成客户端负载均衡的服务调用。Spring Cloud Ribbon几乎存在于每一个Spring Cloud构建的微服务和基础设施中

## 6.1改造服务消费者

复制工程demo-eureka-consumer-service 改为demo-ribbon-consumer-service

### 1、引入jar包

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

### 2、为RestTemplate 添加@LoadBalance注解

```
1 @Bean
2 @LoadBalanced
3 public RestTemplate restTemplate(){
4     return new RestTemplate();
5 }
```

### 3、修改OrderBiz类

```
1 public AddOrderRes addOrder(AddOrderReq addOrderReq) {
2
3     //1、查询商品信息
4     GetGoodsByIdRes getGoodsByIdRes = restTemplate.getForObject("http://demo-
5 -eureka-provider-service/goods/goods_info/"+addOrderReq.getProductId(), Get
6 GoodsByIdRes.class);
7
8     if(getGoodsByIdRes==null){
9         throw new RuntimeException("商品不存在");
10    }
11
12    OrderPrimary orderPrimary=new OrderPrimary();
13    orderPrimary.setOrderNum(UUID.randomUUID().toString());
14    orderPrimary.setOrderStatus(1);
15 }
```

```

13  orderPrimary.setOrderMoney(getGoodsByIdRes.getPrice()*addOrderReq.getNu
m());
14  orderPrimaryRepository.save(orderPrimary);
15
16  AddOrderRes addOrderRes=new AddOrderRes();
17  BeanUtils.copyProperties(orderPrimary,addOrderRes);
18  return addOrderRes;
19  }

```

## 6.2配置Ribbon

### 6.2.1java代码配置ribbon负载均衡规则

#### 1、全局指定ribbon客户端负载规则

```

1  @Configuration
2  public class RibbonConfiguration {
3
4      @Bean
5      public IRule ribbonRule(){
6          return new RandomRule();//随机负载
7      }
8
9  }

```

#### 2、限定某个客户端的负载规则，此时RibbonConfiguration不能有@Configuration注解

```

1  @Configuration
2  @RibbonClient(name = "demo-eureka-provider-service",configuration =Ribbon
Configuration.class )
3  public class ClientConfiguration {
4
5  }

```

### 6.2.2配置文件使用属性自定义Ribbon配置

从Spring Cloud Netflix1.2.0开始，Ribbon支持使用属性自定义Ribbon客户端。这种方式比使用Java代码配置的方式更加方便。

支持的属性如下，配置的前缀是<clientName>.ribbon.

- **NFLoadBalancerClassName**: 配置ILoadBalancer的实现类
- **NFLoadBalancerRuleClassName**: 配置IRule的实现类
- **NFLoadBalancerPingClassName**: 配置IPing的实现类
- **NIWSServerListClassName**: 配置ServerList的实现类
- **NIWSServerListFilterClassName**: 配置ServerListFilter的实现类

```
1 ribbon-provider-service:
2   ribbon:
3     NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RandomRule
```

详细配置查看博客

<https://www.cnblogs.com/jinjiyese153/p/8656567.html>

## Ribbon内置的规则

**AvailabilityFilteringRule**: 这条规则将跳过被认为是“电路跳闸”的服务器或具有高并发连接数的服务器。

**BestAvailableRule**: 这个规则会跳过“电路跳闸”的服务器，然后选取一个并发请求数最低的服务。

**PredicateBasedRule**: 将服务器过滤逻辑委托给{@link AbstractServerPredicate}实例的规则。过滤后，服务器会以循环的方式从过滤列表中返回。

**RandomRule**: 一种随机分配流量的负载平衡策略。

**RetryRule**: 在一个配置时间段内当选择server不成功，则一直尝试使用subRule的方式选择一个可用的server。

**RoundRobinRule**: 这条规则简单地通过轮询来选择服务器。它通常被用作更高级规则的默认规则或后退。

**WeightedResponseTimeRule**: 对于这个规则，每个服务器根据其平均响应时间给定一个权重。响应时间越长，得到的权重就越小。规则随机选择一个服务器，其中的可能性由服务器的权重决定。

**ZoneAvoidanceRule**: 使用ZoneAvoidancePredicate和AvailabilityPredicate来判断是否选择某个server，前一个判断判定一个zone的运行性能是否可用，剔除不可用的zone（的所有server），**AvailabilityPredicate**用于过滤掉连接数过多的Server。

# 7 Feign声明式调用REST接口

Feign是一个声明式的Web服务客户端，使用Feign可使得Web服务客户端的写入更加方便。

它具有可插拔注释支持，包括Feign注解和JAX-RS注解、Feign还支持可插拔编码器和解码器、Spring Cloud增加了对Spring MVC注释的支持，并HttpMessageConverters在Spring Web中使用了默认使用的相同方式。Spring Cloud集成了Ribbon和Eureka，在使用Feign时提供负载均衡的http客户端。

## 7.1修改消费端代码

### 1、添加依赖

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

### 2、创建feign接口，做相应的注解

```
1 @FeignClient(name = "demo-ribbon-provider-service")
2 @RequestMapping("/goods/goods_info")
3 public interface GoodsCtlApi {
4
5
6   @PostMapping("/{id}")
7   public GetGoodsByIdRes getGoodsById(@PathVariable("id") Integer productId);
8
9 }
```

### 3、修改OrderBiz的addOrder方法

```
1 public AddOrderRes addOrder(AddOrderReq addOrderReq) {
2
3   //1、参数校验
```



```

4
5  /*2、根据商品id查询商品信息*/
6  // GetGoodsByIdRes getGoodsByIdRes = restTemplate.getForObject("http://d
emo-ribbon-provider-service/goods/goods_info/" + addOrderReq.getProductId
()), GetGoodsByIdRes.class);
7  GetGoodsByIdRes getGoodsByIdRes = goodsCtlApi.getGoodsById(addOrderReq.g
etProductId());
8  if(getGoodsByIdRes==null){
9  throw new RuntimeException("不存在 的商品");
10 }
11
12 /*3、根据商品信息生成订单*/
13 OrderPrimary orderPrimary=new OrderPrimary();
14 orderPrimary.setOrderMoney(getGoodsByIdRes.getPrice()*addOrderReq.getNu
m());
15 orderPrimary.setOrderNum(UUID.randomUUID().toString());
16 orderPrimary.setOrderStatus(1);//已创建未支付状态
17 orderPrimaryRepository.save(orderPrimary);
18
19 AddOrderRes addOrderRes=new AddOrderRes();
20 BeanUtils.copyProperties(orderPrimary,addOrderRes);
21
22 return addOrderRes;
23 }
24
25 }

```

#### 4、为启动类添加注解@EnableFeignClients

```

1  @SpringBootApplication
2  @EnableEurekaClient
3  @EnableFeignClients
4  public class FeignConsumerServiceApp {
5
6  @LoadBalanced
7  @Bean
8  public RestTemplate restTemplate(){
9  return new RestTemplate();
10 }
11
12 public static void main(String[] args) {
13 SpringApplication.run(FeignConsumerServiceApp.class);

```

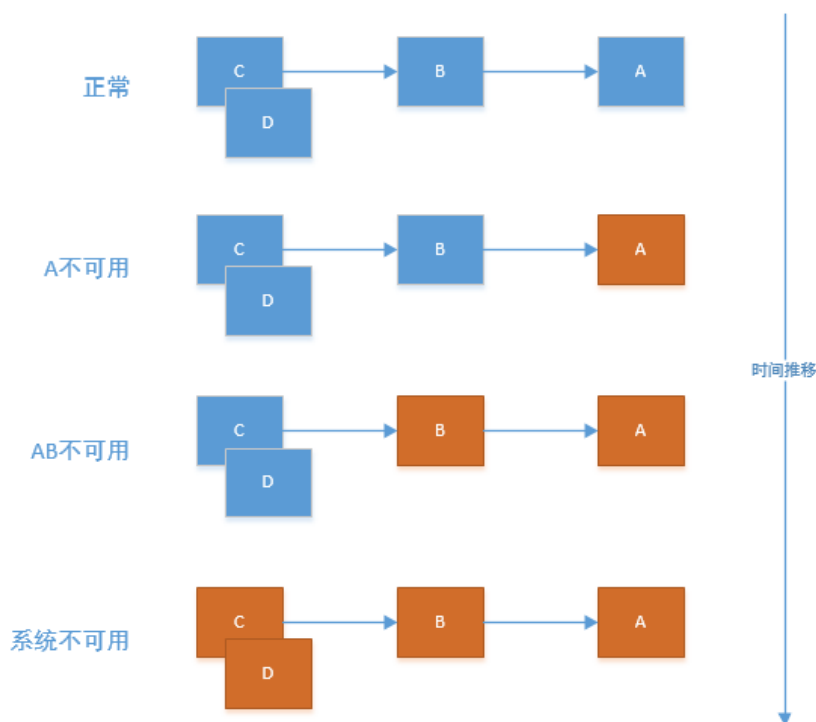
```
14    }  
15  
16 }
```

## 8 熔断器Hystrix

### 8.1 雪崩效应

在微服务架构中，根据业务来拆分成一个个的服务，服务与服务之间可以相互调用（RPC）。由于网络原因或者自身的原因，服务并不能保证100%可用，如果单个服务出现问题，调用这个服务就会出现线程阻塞，此时若有大量的请求涌入，Servlet容器的线程资源会被消耗完毕，导致服务瘫痪。服务与服务之间的依赖性，故障会传播，会对整个微服务系统造成灾难性的严重后果，这就是服务故障的“雪崩”效应。

如下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就形成了。



## 8.2 Hystrix介绍

分布式系统中服务与服务之间相互依赖,一种不可避免的情况就是当某些服务出现故障时,依赖于它们的其他服务出现远程调度的线程阻塞,在高并发的情况下,可能在几秒钟内就会使整个服务处于线程负载饱和状态,从而从一个服务不可用扩散到整个服务不可用,既雪崩效应.Hystrix是Netflix公司开源的一个项目,它提供了熔断器功能,能够阻止分布式系统中的联动故障.

### Hystrix提供的能力

hystrix的出现即为解决雪崩效应, 它通过四个方面的机制来解决这个问题

- **降级机制:** 当请求超时、资源不足(线程资源)、或服务器压力过大时, 我们直接返回简单或默认的数据, 不再继续等待处理该请求, 而是直接返回客户端。
- **资源隔离** (线程池隔离和信号量隔离): 限制调用分布式服务的资源使用, 某一个调用的服务出现问题不会影响其他服务调用。
- **熔断:** 当失败率达到阈值自动触发降级(如因网络故障/超时造成的失败率高), 熔断器触发的快速失败会进行快速恢复。
- **限流机制:** 限流机制主要是提前对各个类型的请求设置最高的QPS阈值, 若高于设置的阈值则对该请求直接返回, 不再调用后续资源。

## 8.4 Hystrix使用

1、创建工程demo-hystrix-consumer-service 和demo-hystrix-provider-service  
导入依赖包

```
1
2 <dependency>
3   <groupId>org.springframework.cloud</groupId>
4   <artifactId>spring-cloud-starter-netflix-hystrix</artifactId>
5 </dependency>
6
```

2、启动类添加注解@EnableCircuitBreaker

```
1 @SpringBootApplication
2 @EnableEurekaClient
3 @EnableFeignClients
4 @EnableCircuitBreaker
5 public class HystrixConsumerServiceApp {
6
7   @LoadBalanced
8   @Bean
9   public RestTemplate restTemplate(){
10     return new RestTemplate();
11   }
12
13   public static void main(String[] args) {
14     SpringApplication.run(HystrixConsumerServiceApp.class);
15   }
16
17
18 }
```

### 3、修改消费者OrderCtl

```
1 @RestController
2 @RequestMapping("/order/order")
3 public class OrderCtl implements OrderCtlApi {
4
5     @Autowired
6     private OrderBiz orderBiz;
7
8
9     @HystrixCommand(fallbackMethod = "addOrderFallback",
10     commandProperties = { @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000")},
11     threadPoolProperties = { @HystrixProperty(name = "coreSize", value = "5")},
12     @HystrixProperty(name = "maxQueueSize", value = "2"))
13 )
14 @RequestMapping("")
15 @Override
16 public Result<AddOrderRes> addOrder(AddOrderReq addOrderReq) {
17
18     AddOrderRes addOrderRes = orderBiz.addOrder(addOrderReq);
19     Result result=new Result(addOrderRes);
20     return result;
21 }
22
23
24 public Result<AddOrderRes> addOrderFallback(AddOrderReq addOrderReq){
25
26     return new Result<>(CommonCode.SERVER_ERROR);
27 }
28 }
29 }
```

### 4、观察结果

启动服务消费者，和注册中心，请求消费者服务 <http://localhost:8081/order/order?productId=1&num=2>

返回默认的值{"productId":-1,"productName":"出错了","price":null,"description":null}

由于服务提供者不可访问，此时会执行@HystrixCommand注解的fallbackMethod属性指定的方法，实现服务降级。

## 8.5 断路器状态监控

服务调用超时，会触发降级，执行回调方法`fallbackmethod`，但此时并未触发熔断，实际上服务调用方还是会调用服务消费方，只不过做了超时处理。`hystrix`会对服务调用失败率做统计，失败率到达一定阈值时，才会触发熔断。`hystrix`触发熔断后，服务调用方将不再调用服务消费方接口，直接执行`fallbackmethod`。

### 1、引入jar包

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

### 2、修改配置文件

```
1 spring:
2   cloud:
3     refresh:
4       refreshable: none
5
6 #显示健康具体信息 默认不会显示详细信息
7 management:
8   endpoint:
9     health:
10      show-details: always
```

### 3、观察结果

访问<http://localhost:8081/actuator/health> 查看健康信息

```
1
2 {"status":"UP","details":{"diskSpace":{"status":"UP","details":{"total":2
3 54792429568,"free":54496755712,"threshold":10485760}},"db":
4 {"status":"UP","details":{"database":"MySQL","hello":1}},"refreshScope":{"s
5 tatus":"UP"},"discoveryComposite":{"status":"UP","details":{"discoveryClien
6 t":{"status":"UP","details":{"services":["ribbon-consumer-service","ribbon-
7 provider-service","register-ha"]}},"eureka":{"description":"Eureka discover
```

```
y client has not yet successfully connected to a Eureka server", "status": "UP", "details": {"applications": {"RIBBON-CONSUMER-SERVICE": 1, "RIBBON-PROVIDER-SERVICE": 1, "REGISTER-HA": 1}}}}, "hystrix": {"status": "UP"}}}
```

"hystrix":{"status":"UP"} 说明此时断路器未打开

4、持续快速访问<http://localhost:8081/order/order> 多次后，再请求  
<http://localhost:8081/actuator/health>

健康信息会出现如下信息，表明断路器已打开

```
1 hystrix":{"status":"CIRCUIT_OPEN", "details":{"openCircuitBreakers":["OrderCtl::saveOrder"]}}
```

5、设置超时时间，默认是3秒

```
1 commandProperties = {  
2   @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "3000")  
3 }
```

## 8.5 Hystrix 隔离策略

Hystrix的资源隔离策略有两种，分别为：线程池和信号量。

Hystrix的隔离主要是为每个依赖组件提供一个隔离的线程环境，提供两种模式的隔离：

- a. 线程池隔离模式：使用一个线程池来处理当前的请求，线程池对请求作处理，设置任务返回处理超时时间，堆积的请求堆积进入线程池队列。这种方式需要为每个依赖的服务申请线程池，有一定的资源消耗，好处是可以应对突发流量（流量洪峰来临时，处理不完可将数据存储到线程池队里慢慢处理）
- a. 信号量隔离模式：使用一个原子计数器（或信号量）来记录当前有多少个线程在运行，请求来先判断计数器的数值，若超过设置的最大线程个数则丢弃改类型的新请求，若不超过则执行计数操作请求来计数器+1，请求返回计数器-1。这种方式是严格的控制线程且立即返回模式，无法应对突发流量（流量洪峰来临时，处理的线程超过数量，其他的请求会直接返回，不继续去请求依赖的服务）

比较

	线程池隔离	信号量隔离
--	-------	-------

	线程池	信号量
线程	与调用线程不相同线程	与调用线程相同
开销	排队、调度、上下文开销等	无线程切换，开销低
异步	支持	不支持
并发支持	支持（最大线程池大小）	支持（最大信号量上限）

### 如何在线程池和信号量之间做选择？

当请求的服务网络开销比较大的时候，或者是请求比较耗时的时候，我们最好是使用线程池隔离策略，这样的话，可以保证大量的容器(tomcat)线程可用，不会由于服务原因，一直处于阻塞或等待状态，快速失败返回。针对纯内存操作，比如比较复杂的耗时的逻辑，我们可以使用信号量隔离策略，因为这类服务的返回通常会非常的快，不会占用容器线程太长时间，而且也减少了线程切换的一些开销。

## 8.6 限流测试

1、对OrderCtl.addOrder 添加如下注解至对应方法设置线程池的大小及队列大小

```
1 threadPoolProperties = { @HystrixProperty(name = "coreSize", value = "5"),
2   @HystrixProperty(name = "maxQueueSize", value = "2")
```

2、JMeter并发请求

hystrix 参数 <https://www.cnblogs.com/520playboy/p/8074347.html>



# 9 服务网关zuul

## 9.1 为什么需要API网关?

- 屏蔽内部细节
- 反向路由
- 安全认证
- 限流熔断
- 日志监控

## 9.2 编写网关服务器

1、创建工程demo-gateway-zuul

2、引入jar包

```
1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.cloud</groupId>
4     <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
5   </dependency>
6 </dependencies>
```

3、编写启动类

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

4、添加配置文件

```
1 spring:
2   application:
3     name: gateway-zuul
4
5   eureka:
6     client:
```

```
7  service-url:
8  defaultZone: http://localhost:9000/eureka/
9  server:
10 port: 7077
```

## 5、观察运行

启动服务 <http://localhost:8081/order/order?productId=1&num=2>和zuul网关

<http://localhost:8081/order/order?productId=1&num=2>

<http://localhost:7077/demo-hystrix-consumer-service/order/order?productId=1&num=2>

<http://localhost:7077/demo/order/order?productId=1&num=2>

访问<http://localhost:7077/demo-ribbon-consumer-service/order/order?productId=1&num=2>

请求会被转发到<http://localhost:8081/goods/goods/1>

## 自定义路由规则

```
1  zuul:
2    routes:
3      demo:
4        path: /demo/**
5        serviceId: demo-hystrix-consumer-service
```

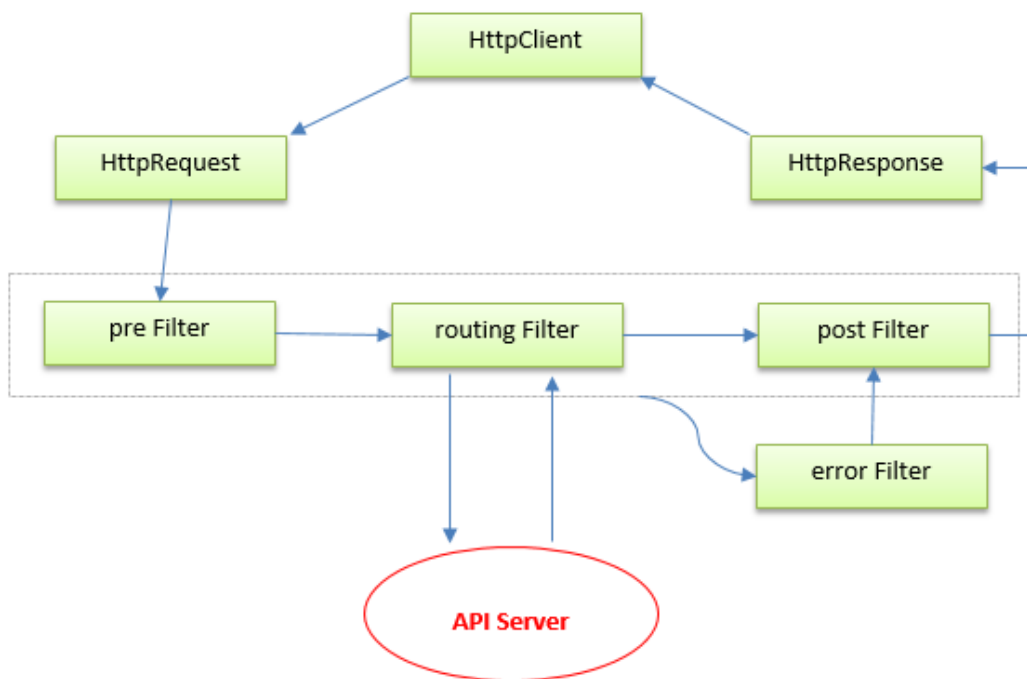
# 9.3查看路由信息

## 添加配置

```
1  management:
2    endpoints:
3      web:
4        exposure:
5          include: '*'
```

<http://localhost:7077/actuator/routes/detail>

## 9.4 Zuul过滤器



Zuul的生命周期

<https://blog.csdn.net/fanrenxiang>

### 9.4.1 spring cloud zuul包含4种类型的过滤器。

**pre过滤器**：在请求被路由之前调用。Zuul请求微服务之前。比如请求身份验证，选择微服务实例，记录调试信息等。

**route过滤器**：负责转发请求到微服务。原始请求在此构建，并使用Apache HttpClient或Netflix Ribbon发送原始请求。

**post过滤器**：在route和error过滤器之后被调用。可以在响应添加标准HTTP Header、收集统计信息和指标，以及将响应发送给客户端等。

**error过滤器**：在处理请求发生错误时被调用。

### 9.4.2 实现简单的鉴权过滤器

#### 1、编写过滤器类

```
1 public class TokenFilter extends ZuulFilter {
2     @Override
3     public String filterType() {
```

```

4   return "pre";
5   }
6   @Override
7   public int filterOrder() {
8       return 1;
9   }
10
11  @Override
12  public boolean shouldFilter() {
13      return true;
14  }
15
16  @Override
17  public Object run() throws ZuulException {
18      RequestContext currentContext = RequestContext.getCurrentContext();
19      HttpServletRequest request = currentContext.getRequest();
20      String token = request.getParameter("token");
21      HttpServletResponse response = currentContext.getResponse();
22      if(token==null || !token.equals("abc")){
23          try {
24              ServletOutputStream outputStream = response.getOutputStream();
25              outputStream.write("invalid token".getBytes("utf-8"));
26              outputStream.close();
27          } catch (IOException e) {
28              e.printStackTrace();
29          }
30      }
31
32
33      return null;
34  }
35  }

```

方法说明：

**filterType**: 返回过滤器的类型。有pre、route、post、error等几种取值，分别对应上文的几种过滤器。详细可以参考com.netflix.zuul.ZuulFilter.filterType() 中的注释。

**filterOrder**: 返回一个int值来指定过滤器的执行顺序，不同的过滤器允许返回相同的数字。

**shouldFilter**: 返回一个boolean值来判断该过滤器是否要执行，true表示执行，false表示不执行。

run：过滤器的具体逻辑。本例中，对非法的token进行拦截。

## 2、将过滤器加入spring容器中

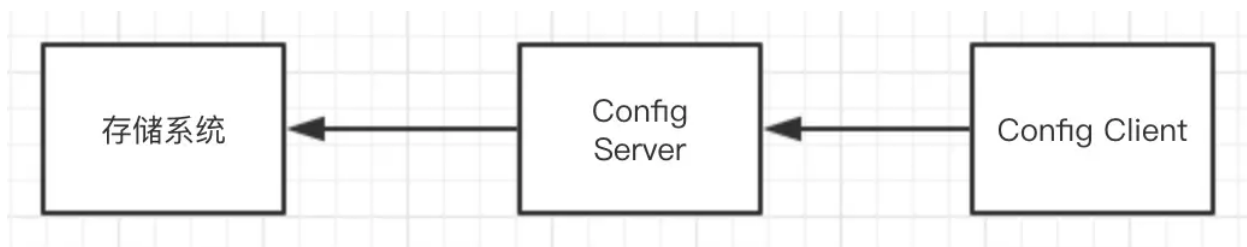
```
1 @Bean
2 public TokenFilter tokenFilter(){
3     return new TokenFilter();
4 }
```

# 10 分布式配置服务

## 10.1 作用

- 1、集中管理配置
- 2、运行期间可以动态调整配置属性

## 10.2 spring cloud config



### 角色说明：

**配置中心服务端 (Config Server)：** 为配置客户端提供对应的配置信息，配置信息的来源是配置仓库。应用启动时，会从配置仓库拉取配置信息缓存到本地仓库中。

**配置中心客户端(Config Client)：** 应用启动时从配置服务端拉取配置信息。

**配置仓库 (存储系统)：** 为配置中心服务端提供配置信息存储，Spring Cloud Config 默认是使用git作为仓库的。

## 10.3创建config-server

## 1、创建项目demo-config-server

## 2、引入jar包

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

## 3、配置文件

```
1 server:
2   port: 5050
3   spring:
4     application:
5       name: config-sever-app
6     cloud:
7       config:
8         server:
9           git:
10            uri: https://gitee.com/group1024/spring-cloud-config-repo.git #仓库地址
11            username: 960302601@qq.com #用户名
12            password: yxj123456 #密码
13            skip-ssl-validation: true
```

## 4、申请账号创建一个远程仓库: <https://gitee.com/group1024/spring-cloud-config-repo.git>

## 5、在仓库spring-cloud-config-rep中添加配置文件**config-sever-app-dev.yml**

```
1 myproperty:dev
```

## 6、启动服务器

端点与配置文件的映射规则

`/{{application}}/{{profile}}[/{label}]`

`/{{application}}-{{profile}}.yml`

`/ {label}/ {application}- {profile}. yml`

`/ {application}- {profile}. properties`

`/ {label}/ {application}- {profile}. properties`

以上端点都可以映射到`{application}- {profile}. properties`这个配置文件，`{application}`表示微服务的名称，`{label}`对应Git仓库的分支，默认是`master`。

例子：

访问端点：<http://localhost:5050/config-sever-app/dev>，即可获得`config-sever-app-dev. yml`配置文件内容

## 10.4 编写ConfigClient

1、创建项目demo-config-client

2、引入jar包

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

3、配置文件

```
1 server:
2   port: 5080
3 spring:
4   application:
5     name: config-client-app
6   cloud:
7     config:
8       uri: http://localhost:5050/ # config-server访问地址
```

```
9  profile: dev
10 label: master
```

#### 4、编写controller

```
1 @RestController
2 public class ConfigCtl {
3
4     @Value("${myproperty}")
5     private String myproperty;
6     @RequestMapping("/config")
7     public void getConfigi(){
8
9         System.out.println(myproperty);
10
11     }
12 }
```

#### 5、编写启动类

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