

SpringCloud第二季

- 1、课程内容 (SpringCloud+SpringCloud alibaba)
- 2、技术要求: java8+maven+git(github)+Nginx+RabbitMQ+springboot2.0

什么是微服务

微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务与服务间采用轻量级的通讯机制互相协作(通常是基于HTTP机制的RESTful API)。每个服务都围绕着具体业务进行构建，并且能够独立的部署在生产环境、类生产环境

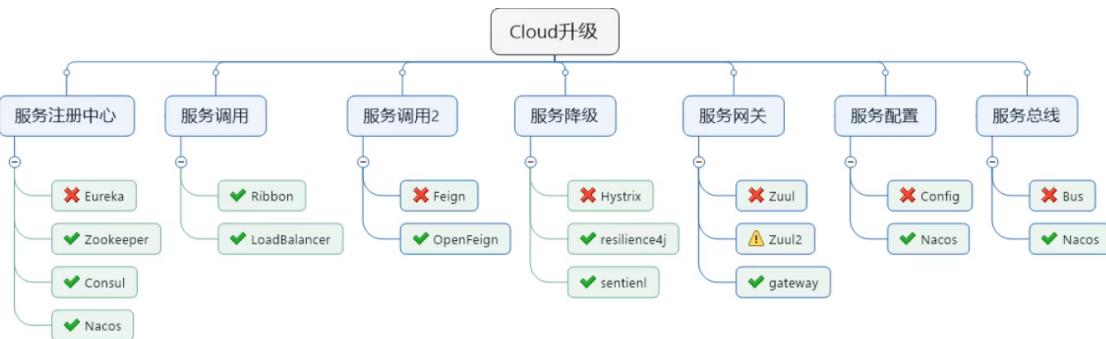
SpringCloud

分布式微服务架构的一站式解决方案，是多种微服务架构落地技术的集合体，俗称微服务全家桶。

版本选型：进入到spring官网<https://spring.io/>，Projects-->Spring Cloud-->LEARN下就可以看到最新的版本，点击[Reference Doc](#)就可以看到当前SpringCloud推荐的SpringBoot版本

SpringCloud与SpringBoot版本对应关系查看：

<https://start.spring.io/actuator/info>



参考资料：可以查看SpringCloud中文文档

父工程搭建

maven项目-->maven原型site-->.....搭建完成后；

- 设置：
- (1) 字符编码设置setting-->Editor-->File Encodings，设置3个UTF-8，复选框打勾；
 - (2) 注解生效激活：setting-->Build-->Compiler-->Annotation，勾上复选框Enable annotation
 - (3) java编译版本选8，setting-->Build-->Compiler-->Java Compiler，项目名后面选8
 - (4) File Type过滤器（可以不设置）：setting-->Editor-->File Type，最下面加上 .idea;.iml;

dependencyManagement

Maven 使用 `dependencyManagement` 元素来提供了一种管理依赖版本号的方式。

通常会在一个组织或者项目的最顶层的父POM 中看到 `dependencyManagement` 元素。

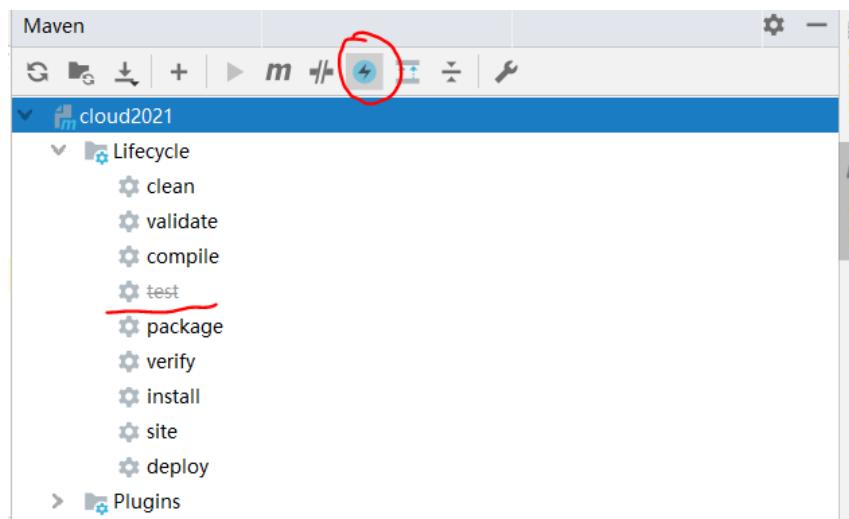
使用 `pom.xml` 中的 `dependencyManagement` 元素能让所有在子项目中引用一个依赖而不用显式的列出版本号。Maven 会沿着父子层次向上走，直到找到一个拥有 `dependencyManagement` 元素的项目，然后它就会使用这个 `dependencyManagement` 元素中指定的版本号。

这样做的好处就是：如果有多个子项目都引用同一样依赖，则可以避免在每个使用的子项目里都声明一个版本号，这样当想升级或切换到另一个版本时，只需要在顶层父容器里更新，而不需要一个一个子项目的修改；另外如果某个子项目需要另外的一个版本，只需要声明 `version` 就可。

- * `dependencyManagement` 里只是声明依赖，并不实现引入，因此子项目需要显示的声明需要用的依赖。
- * 如果不在子项目中声明依赖，是不会从父项目中继承下来的；只有在子项目中写了该依赖项，并且没有指定具体版本，才会从父项目中继承该项，并且 `version` 和 `scope` 都读取自父 `pom`；
- * 如果子项目中指定了版本号，那么会使用子项目中指定的jar版本。

https://blog.csdn.net/weixin_427460

Maven 中跳过单元测试：



创建微服务模块

1. 建model
2. 改pom
3. 写YML
4. 主启动
5. 业务类

热部署Devtools

开发时使用，生产环境关闭

1. Adding devtools to your project

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>
```

2.Adding plugin to your pom.xml

下段配置复制到聚合父类总工程的pom.xml

```
<build>
    <!--
        <finalName>你的工程名</finalName> (单一工程时添加)
    -->
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <fork>true</fork>
                <addResources>true</addResources>
            </configuration>
        </plugin>
    </plugins>
</build>
```

3.Enabling automatic build

File -> Settings(New Project Settings->Settings for New Projects) ->Complier

下面项勾选

Automatically show first error in editor
Display notification on build completion
Build project automatically
Compile independent modules in parallel

4.Update the value of

键入Ctrl + Shift + Alt + / , 打开Registry, 勾选:

compiler.automake.allow.when.app.running

actionSystem.requestFocusFromEdt

5.重启IDEA

RestTemplate

RestTemplate提供了多种便捷访问远程Http服务的方法，是一种简单便捷的访问restful服务模板类，是Spring提供的用于访问Rest服务的客户端模板工具集

使用:

使用restTemplate访问restful接口非常的简单粗暴无脑。
(url, requestMap, ResponseBean.class)这三个参数分别代表。
REST请求地址、请求参数、HTTP响应转换被转换成的对象类型。

Eureka

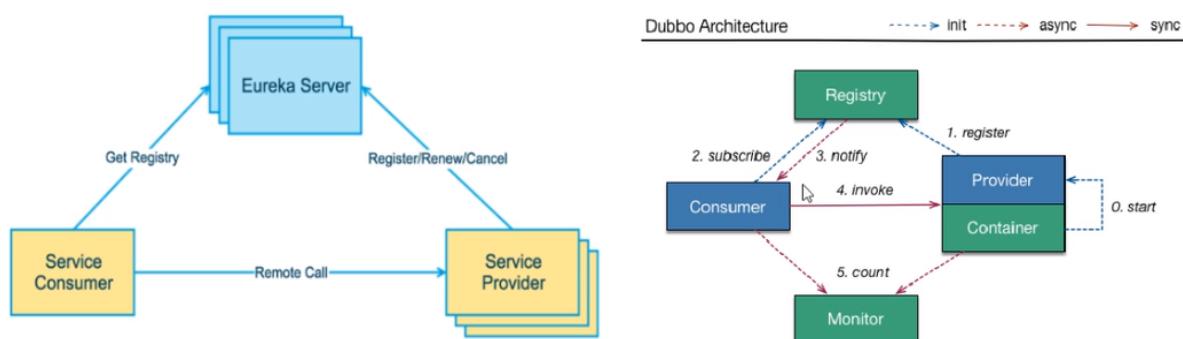
基础知识

什么是服务注册与发现

Eureka采用了CS的设计架构，Eureka Server作为服务注册功能的服务器，它是服务注册中心。而系统中的其他微服务，使用Eureka的客户端连接到 Eureka Server并维持心跳连接。这样系统的维护人员就可以通过Eureka Server来监控系统中各个微服务是否正常运行。

在服务注册与发现中，有一个注册中心。当服务器启动的时候，会把当前自己服务器的信息比如服务通讯地址等以别名方式注册到注册中心上。另一方(消费者服务提供者)，以该别名的方式去注册中心上获取到实际的服务通讯地址，然后再实现本地RPC调用。RPC远程调用框架核心设计思想：在于注册中心，因为使用注册中心管理每个服务与服务之间的一个依赖关系(服务治理概念)。在任何RPC远程框架中，都会有一个注册中心存放服务地址相关信息(接口地址)

下左图是Eureka系统架构，右图是Dubbo的架构，请对比



Eureka包含两个组件:Eureka Server和Eureka Client

Eureka Server提供服务注册服务

各个微服务节点通过配置启动后，会在EurekaServer中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观看到。

EurekaClient通过注册中心进行访问

它是一个Java客户端，用于简化Eureka Server的交互，客户端同时也具备一个内置的、使用轮询(round-robin)负载算法的负载均衡器。在应用启动后，将会向Eureka Server发送心跳(默认周期为30秒)。如果Eureka Server在多个心跳周期内没有接收到某个节点的心跳，EurekaServer将会从服务注册表中把这个服务节点移除 (默认90秒)

springboot整合Eureka

step1：搭建eureka服务端 Eureka Server

新建模块-->改pom (添加依赖) -->改配置 --> 启动类添加@EnableEurekaServer

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

```

server:
  port: 7001
eureka:
  instance:
    hostname: localhost #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: false #false表示自己端就是注册中心，我的职责就是维护服务实例，并不需要去检索服务
    service-url:
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/ #设置与eureka交互的地址，查询服务和注册服务都需要依赖这个地址。

```

step2: Eureka客户端

新建模块-->改pom (添加依赖) -->改配置 --> 启动类添加@EnableEurekaClient

```

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

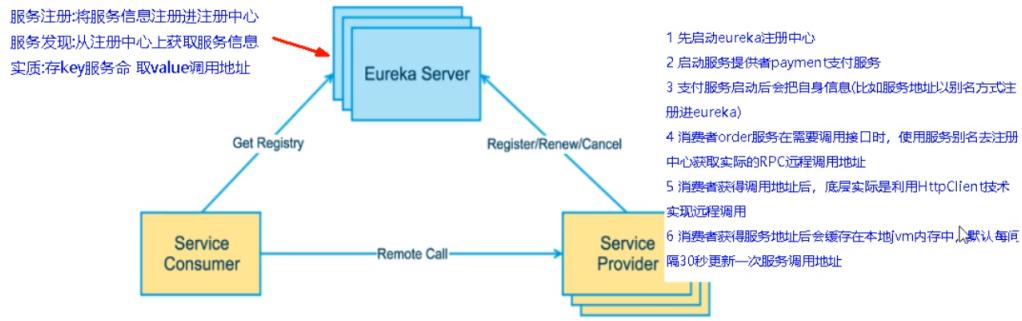
```

```

server:
  port: 8001
spring:
  application:
    name: cloud-payment-service
  datasource:
    type: com.alibaba.druid.pool.DruidDataSource      # 当前数据源操作类型
    driver-class-name: org.gjt.mm.mysql.Driver          # mysql驱动包
    url: jdbc:mysql://localhost:3306/cloud2020?
    useUnicode=true&characterEncoding=utf-8&useSSL=false
    username: root
    password: 123456
  mybatis:
    mapperLocations: classpath:mapper/*.xml
    type-aliases-package: com.atguigu.springcloud.entity   # 所有Entity别名类所在包
eureka:
  client:
    register-with-eureka: true # 表示是否将自己注册进EurekaServer， 默认为true
    fetch-registry: true #是否从EurekaServer抓取已有的注册信息， 默认为true。单节点无所谓，集群必须设置为true才能配合ribbon使用负载均衡
    service-url:
      defaultZone: http://localhost:7001/eureka

```

Eureka集群原理说明



问题: 微服务RPC远程服务调用最核心的是什么

高可用, 试想你的注册中心只有一个only one, 它出故障了那就呵呵(￣▽￣)"了, 会导致整个服务环境不可用, 所以

解决办法: 搭建Eureka注册中心集群, 实现负载均衡+故障容错

Eureka集群原理: 互相注册, 相互守望

Eureka Server集群搭建

在单机版的基础上主要修改配置文件

```
server:
  port: 7001
eureka:
  instance:
    hostname: eureka7001.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: false #false表示自己端就是注册中心, 我的职责就是维护服务实例, 并不需要去检索服务
    service-url:
      defaultZone: http://eureka7002.com:7002/eureka/ #设置与eureka交互的地址, 查询服务和注册服务都需要依赖这个地址。
```

```
server:
  port: 7002
eureka:
  instance:
    hostname: eureka7002.com #eureka服务端的实例名称
  client:
    register-with-eureka: false #false表示不向注册中心注册自己
    fetch-registry: false #false表示自己端就是注册中心, 我的职责就是维护服务实例, 并不需要去检索服务
    service-url:
      defaultZone: http://eureka7001.com:7001/eureka/ #设置与eureka交互的地址, 查询服务和注册服务都需要依赖这个地址。
```

服务提供者集群后, 服务消费者调用服务的方式

step1: 在RestTemplate配置上添加注解

```

@Configuration
public class ApplicationContextConfig {
    @Bean
    @LoadBalanced //赋予 RestTemplate 负载均衡的能力
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}

```

step2：修改调用的请求url，主机名改成服务提供者的应用名，例如 CLOUD-PAYMENT-SERVICE

```

public static final String PAYMENT_URL = "http://CLOUD-PAYMENT-SERVICE";
@GetMapping("/consumer/payment/get/{id}")
public CommonResult<Payment> getPayment(@PathVariable("id") Long id){
    return restTemplate.getForObject(PAYMENT_URL + "/payment/get/" + id,
CommonResult.class);
}

```

actuator微服务信息完善

```

eureka:
  instance:
    instance-id: payment8001
    prefer-ip-address: true #访问路径可以显示ip地址

```

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
CLOUD-ORDER-SERVICE	n/a (1)	(1)	UP (1) - LAPTOP-DHHL2HH:cloud-order-service:80
CLOUD-PAYMENT-SERVICE	n/a (2)	(2)	UP (2) - payment8002 , payment8001
General Info			Value
192.168.2.11:8002/actuator/info			

服务发现Discovery

step1：在controller中(提供者/消费者)，添加如下代码

```

@Resource
private DiscoveryClient discoveryClient;

@GetMapping("/payment/discovery")
public Object discovery(){
    List<String> services = discoveryClient.getServices(); //查询在当前注册中心
    //有哪些微服务
    for(String element : services){
        log.info("*****element: " + element);
    }
    //查询指定微服务的详细信息
    List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-
    PAYMENT-SERVICE");
    for(ServiceInstance instance : instances){
        log.info(instance.getInstanceId() + "\t" + instance.getHost() + "\t"
+ instance.getPort() + "\t" + instance.getUri() + "\t" +
        instance.getServiceId());
    }
}

```

```
    return discoveryClient;
}
```

step2: 在启动类上添加注解: @EnableDiscoveryClient

Eureka自我保护

导致原因

[为什么会产生Eureka自我保护机制?](#)

为了防止EurekaClient可以正常运行，但是与 EurekaServer网络不通情况下，EurekaServer[不会立刻](#)将EurekaClient服务剔除

[什么是自我保护模式?](#)

默认情况下，如果EurekaServer在一定时间内没有接收到某个微服务实例的心跳，EurekaServer将会注销该实例（默认90秒）。但是当网络分区故障发生(延时、卡顿、拥挤)时，微服务与EurekaServer之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，[此时本不应该注销这个微服务](#)。Eureka通过“自我保护模式”来解决这个问题——当EurekaServer节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。

https://blog.csdn.net/weixin_42746009

[在自我保护模式中，Eureka Server会保护服务注册表中的信息，不再注销任何服务实例。](#)

它的设计哲学就是宁可保留错误的服务注册信息，也不盲目注销任何可能健康的服务实例。[一句话讲解：好死不如赖活着](#)

综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留）也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。https://blog.csdn.net/weixin_42746009

关闭自我保护

eureka服务端配置文件添加

```
eureka:  
  server:  
    enable-self-preservation: false #关闭自我保护机制，保证不可用服务被及时剔除  
    eviction-interval-timer-in-ms: 2000 #时间间隔2秒
```

eureka客户端配置文件添加

```
eureka:  
  instance:  
    lease-renewal-interval-in-seconds: 1 #Eureka客户端向服务端发送心跳的时间间隔，单位  
    为秒（默认30s）  
    lease-expiration-duration-in-seconds: 2 #Eureka服务端在收到最后一次心跳后等待时间上  
    限，单位为秒（默认90s），超时将剔除服务
```

zookeeper作为注册中心

配置步骤

服务提供者和服务消费者步骤一样

step1: pom中添加依赖

```
<dependency>  
  <groupId>org.springframework.cloud</groupId>  
  <artifactId>spring-cloud-starter-zookeeper-discovery</artifactId>  
  <exclusions>  
    <exclusion>  
      <groupId>org.apache.zookeeper</groupId>  
      <artifactId>zookeeper</artifactId>  
    </exclusion>  
  </exclusions>  
</dependency>
```

```
</exclusions>
</dependency>
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.14</version>
</dependency>
```

step2: 配置文件

```
server:
  port: 8004
spring:
  application:
    name: cloud-provider-payment
  cloud:
    zookeeper:
      connect-string: 39.103.129.145:2181 #集群模式 后面加逗号，再写ip:端口，和Eureka一样
```

step3: 启动类上添加注解

```
@EnableDiscoveryClient //该注解用于向使用consul或者zookeeper作为注册中心时注册服务
public class PaymentMain8004 {}
```

注意：注册进zookeeper的服务节点是临时节点

服务调用

```
@Configuration
public class ApplicationContextConfig {
    @Bean
    @LoadBalanced //赋予 RestTemplate 负载均衡的能力
    public RestTemplate getRestTemplate(){
        return new RestTemplate();
    }
}
```

```
@RestController
@Slf4j
public class OrderZkController {
    private static final String INVOKE_URL = "http://cloud-provider-payment";
    @Resource
    private RestTemplate restTemplate;
    @GetMapping("/consumer/payment/zk")
    public String getPaymentZk(){
        return restTemplate.getForObject(INVOKE_URL + "/payment/zk",
String.class);
    }
}
```

Consul作为注册中心

简介

Consul是一套开源的分布式服务发现和配置管理系统，用Go语言开发。

提高了微服务系统中的服务治理、配置中心、控制总线等功能。这些功能中的每一个都可以根据需要单独使用，也可以一起使用以构建全方位的服务网络，总之Consul提供了一种完整的服务网格解决方案。

它具有很多优点。包括：基于raft协议，比较简洁；支持健康检查，同时支持HTTL和DNS协议，支持跨数据中心的WAN集群，提高图像界面，跨平台，支持Linux、Mac、Windows

安装并运行Consul

- 1、官网下载 <https://www.consul.io/downloads> consul_1.6.1_windows_amd64.zip 里面只有1个 consul.exe文件
- 2、在consul.exe文件的地址栏输入“cmd”，打开命令行窗口。并键入“consul”，若出现一连串英文则表示安装成功
- 3、启动consul：

```
consul agent -dev
```

- 4、验证：在浏览器地址栏输入：<http://localhost:8500/ui/dc1/services> 就可以看到页面了

配置步骤

step1：添加依赖

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

step2：配置文件

```
server:
  port: 8006
spring:
  application:
    name: consul-provider-payment
  cloud:
    consul:
      host: localhost
      port: 8500
      discovery:
        service-name: ${spring.application.name}
```

step3：主类上添加注解 @EnableDiscoveryClient

Ribbon负载均衡服务调用

Spring Cloud Ribbon是基于Netflix Ribbon实现的一套客户端 负载均衡的工具。

主要功能是提供客户端的软件负载均衡算法和服务调用。简单的说，就是在配置文件中列出Load Balance（简称LB）后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随机连接等）去连接这些机器。我们很容易使用Ribbon实现自定义的负载均衡算法。Ribbon目前也进入维护模式了。

一句话：负载均衡+RestTemplate调用

Ribbon & Nginx

Nginx是服务器负载均衡，客户端所有请求都会交给Nginx，然后由Nginx实现转发请求。即负载均衡是由服务端实现的。

Ribbon本地负载均衡，在调用微服务接口的时候，会在注册中心上获取注册信息服务列表之后缓存到JVM本地，从而在本地实现RPC远程服务调用技术。

集中式LB

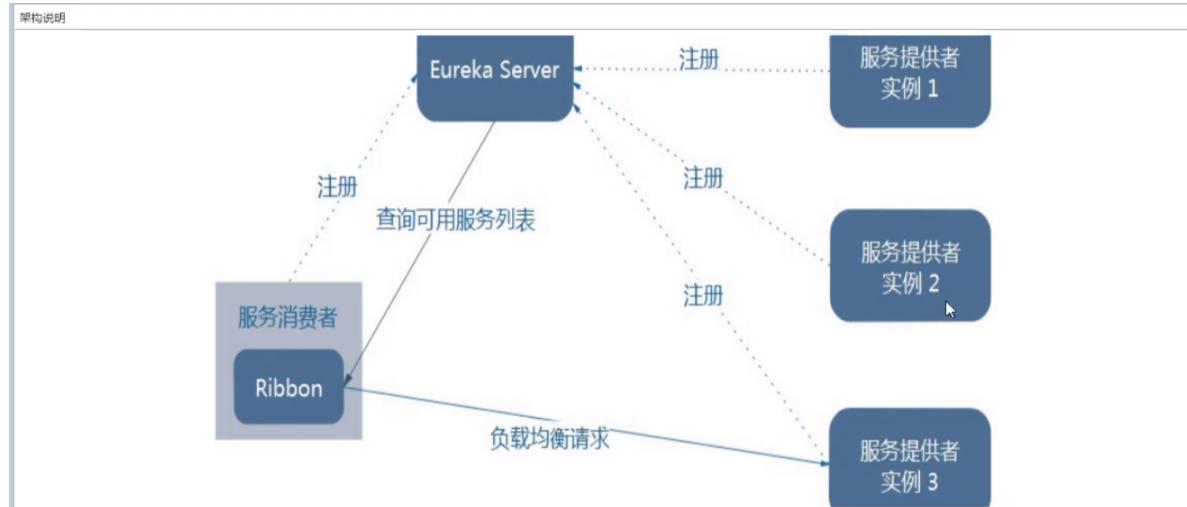
即在服务的消费方和提供方之间使用独立的LB设施(可以是硬件，如F5，也可以是软件，如nginx)，由该设施负责把访问请求通过某种策略转发至服务的提供方；

进程内LB

将LB逻辑集成到消费方，消费方从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

Ribbon就属于进程内LB，它只是一个类库，集成于消费方进程，消费方通过它来获取到服务提供方的地址。

Ribbon架构说明



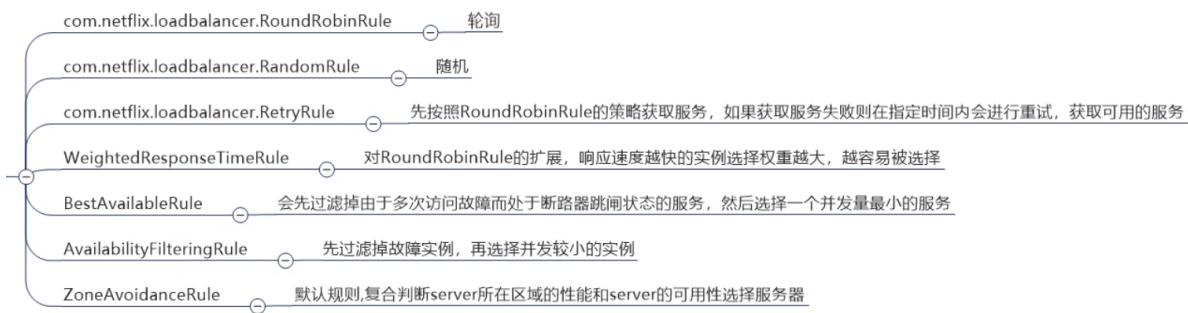
Ribbon在工作时分成两步：

step1：先选择EurekaServer，它优先选择在同一个区域内负责较少的server

step2：再根据用户指定的策略，在从server取到的服务注册列表中选择一个地址。

其中Ribbon提供了多种策略：比如轮询（默认）、随机和根据响应时间加权。

Ribbon负载均衡策略



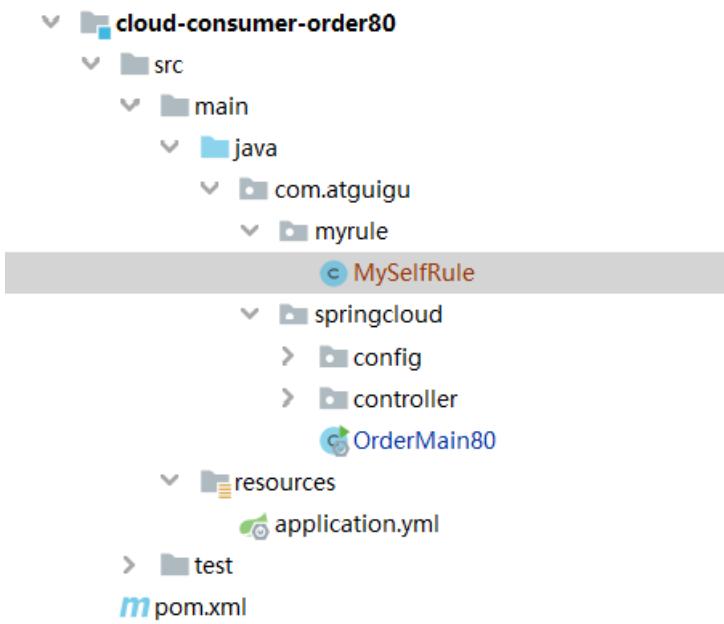
Ribbon负载规则替换

注意配置细节

官方文档明确给出警告：这个自定义配置类不能放在@ComponentScan所扫描的当前包及其子包下，否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享，达不到特殊化定制的目的了。

新建package

```
com.atguigu.myrule
```



新建MySelfRule规则类

```
@Configuration
public class MySelfRule {
    @Bean
    public IRule myRule(){
        return new RandomRule(); // 定义为随机
    }
}
```

主启动类添加@RibbonClient

```
@RibbonClient(name = "CLOUD-PAYMENT-SERVICE", configuration = MySelfRule.class)
```

轮询负载均衡算法

rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标

每次服务器重启后rest接口计数从1开始。

OpenFeign服务接口调用

Feign是一个声明式WebService客户端。使用Feign能让编写Web Service客户端更加简单。它的**使用方法是定义一个服务接口然后在上面添加注解**。Feign可以与Eureka和Ribbon组合使用以支持负载均衡。

OpenFeign超时控制

配置文件中添加配置：

```
ribbon:  
    ReadTimeout: 5000 # 指的是建立连接后从服务器读取到可用资源所用的时间  
    ConnectTimeout: 5000 # 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
```

OpenFeign日志增强

```
NONO: 默认的，不显示任何日志；  
BASIC: 仅记录请求方法、URL、响应状态码及执行时间；  
HEADERS: 除了BASIC中定义的信息之外，还有请求和响应的头信息；  
FULL: 除了HEADERS中定义的信息之外，还有请求和响应的正文及元数据
```

step1：配置日志bean

```
@Configuration  
public class FeignConfig {  
    @Bean  
    Logger.Level feignLoggerLevel(){  
        return Logger.Level.FULL;  
    }  
}
```

step2：yaml中添加配置

```
logging:  
  level:  
    # feign 日志以什么级别监控哪个接口  
    com.atguigu.springcloud.service.PaymentFeignService: debug
```

Hystrix断路器

概述

分布式系统面临的问题：复杂分布式体系结构中的应用程序有数十个依赖关系，每个依赖关系在某些时候将不可避免的失败。

服务雪崩

多个微服务之间调用的时候，假设微服务A调用微服务B和微服务C，微服务B和微服务C又调用其它的微服务，这就是所谓的“**扇出**”。如果扇出的链路上某个微服务的调用响应时间过长或者不可用，对微服务A的调用就会占用越来越多的系统资源，进而引起系统崩溃，所谓的“**雪崩效应**”。

对于高流量的应用来说，单一的后端依赖可能会导致所有服务器上的所有资源都在几秒钟内饱和。比失败更糟糕的是，这些应用程序还可能导致服务之间的延迟增加，备份队列，线程和其他系统资源紧张，导致整个系统发生更多的级联故障。这些都表示需要对故障和延迟进行隔离和管理，以便单个依赖关系的失败，不能取消整个应用程序或系统。

所以，

通常当你发现一个模块下的某个实例失败后，这时候这个模块依然还会接收流量，然后这个有问题的模块还调用了其他的模块，这样就会发生级联故障，或者叫雪崩。

Hystrix是一个用于处理分布式系统的**延迟**和**容错**的开源库，在分布式系统里，许多依赖不可避免的会调用失败，比如超时、异常等，Hystrix能够保证在一个依赖出问题的情况下，**不会导致整体服务失败，避免级联故障，以提高分布式系统的弹性**。

“断路器”本身是一种开关装置，当某个服务单元发生故障之后，通过断路器的故障监控（类似熔断保险丝），**向调用方返回一个符合预期的、可处理的备选响应（FallBack）**，而不是长时间的等待或者抛出调用方无法处理的异常，这样就保证了服务调用方的线程不会被长时间、不必要地占用，从而避免了故障在分布式系统中的蔓延，乃至雪崩。

Hystrix能干嘛？服务降级、服务熔断、接近实时的监控

服务降级

对方系统不可用了，你需要给我一个兜底的解决方法。服务器忙，请稍后再试，不让客户端等待并立刻返回一个友好提示，fallback。

哪些情况会触发降级：程序运行异常，超时，服务熔断触发降级，线程池/信号量打满也会导致服务降级

服务熔断

类比保险丝达到最大服务访问后，直接拒绝访问，拉闸限电，然后调用服务降级的方法并返回友好提示

服务限流

秒杀高并发等操作，严禁一窝蜂的过来拥挤，大家排队，一秒钟N个，有序进行

故障现象和导致原因

8001同一层次的其他接口服务被困死，因为Tomcat线程池里面的工作线程已经被挤占完毕

80此时调用8001，客户端访问响应缓慢，转圈圈。

如何解决

超时导致服务器变慢（转圈）--超时不再等待

出错（宕机或程序运行出错）--出错要有兜底

解决-服务降级

对方服务(8001)超时了，调用者(80)不能一直卡死等待，必须有服务降级

对方服务(8001)宕机了，调用者(80)不能一直卡死等待，必须有服务降级

对方服务(8001)OK，调用者(80)自己出故障或有自我要求(自己的等待时间小于服务提供者)，自己处理降级

```

@HystrixCommand(fallbackMethod = "paymentInfo_timeoutHandler", commandProperties
= {
    @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds",
value = "3000")
})
public String paymentInfo_timeout(Integer id){
    int timeNum = 5;
    try {
        // int num = 10/0;
        TimeUnit.SECONDS.sleep(timeNum);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    return "线程池: " + Thread.currentThread().getName() + " paymentInfo_timeout,
id: " + id + "\t" + "耗时(秒): " + timeNum;
}
public String paymentInfo_timeoutHandler(Integer id){
    return "线程池: " + Thread.currentThread().getName() + " 系统繁忙, 请稍后再试,
id: " + id + " /(ㄒoㄒ)/~~";
}

```

```

@EnableCircuitBreaker // 主类上添加这个注解(提供者)
@EnableHystrix // 主类上添加这个注解(消费者)

```

```

# 提供者配置文件添加配置
feign:
  hystrix:
    enabled: true

```

上面的服务降级，在服务提供者、消费者都可以用，一般大部分还是在消费者侧使用的多。

全局服务降级@DefaultProperties

- 1.@DefaultProperties
- 2.@HystrixCommand
- 3.payment_Global_FallbackMethod()

```

package com.atguigu.springcloud.controller;

@RestController
@Slf4j
@DefaultProperties(defaultFallback = "payment_Global_FallbackMethod")
public class OrderHystrixController {
    @Resource
    private PaymentHystrixService paymentHystrixService;

    @RequestMapping("/consumer/payment/hystrix/timeout/{id}")
    @HystrixCommand
    public String paymentInfo_timeout(@PathVariable("id") Integer id){
        int num = 10/0;
        String result = paymentHystrixService.paymentInfo_timeout(id);
        return result;
    }
}

```

```
// 全局 fallback 方法
public String payment_Global_FallbackMethod(){
    return "Global异常处理信息, 请稍后再试/(ㄒoㄒ)/~~";
}
```

通配服务降级FeignFallback

1、通过创建类PaymentFallbackService实现PaymentHytrixService

```
@Component
public class PaymentFallbackService implements PaymentHytrixService {
    @Override
    public String paymentInfo_ok(Integer id) {
        return "=====PaymentFallbackService====paymentInfo_ok====";
    }
    @Override
    public String paymentInfo_timeout(Integer id) {
        return "=====PaymentFallbackService====paymentInfo_timeout====";
    }
}
```

2、PaymentHytrixService类上注解中添加属性fallback指向它的实现类

```
@FeignClient(value = "CLOUD-PROVIDER-HYSTRIX-PAYMENT", fallback =
PaymentFallbackService.class)
public interface PaymentHytrixService {
    @RequestMapping("/payment/hystrix/ok/{id}")
    public String paymentInfo_ok(@PathVariable("id") Integer id);
}
```

服务熔断理论

熔断机制概述

熔断机制是应对雪崩效应的一种微服务链路保护机制。当扇出链路的某个微服务出错不可用或者响应时间太长时，会进行服务的降级，进而熔断该节点微服务的调用，快速返回错误的响应信息。

[当检测到该节点微服务调用响应正常后，恢复调用链路。](#)

在Spring Cloud框架里，熔断机制通过Hystrix实现。Hystrix会监控微服务间调用的状况，当失败的调用到一定阈值，缺省是5秒内20次调用失败，就会启动熔断机制。熔断机制的注解是@HystrixCommand。

Hytrix服务熔断案例

```
@RequestMapping("/payment/circuit/{id}")
public String paymentCircuitBreaker(@PathVariable("id") Integer id){
    String result = paymentService.paymentCircuitBreaker(id);
    log.info("==result = " + result);
    return result;
}
```

```
@HystrixCommand(fallbackMethod = "paymentCircuitBreaker_fallback",
commandProperties = {
    @HystrixProperty(name = "circuitBreaker.enabled", value = "true"), //是否开启
    断路器
```

```

    @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value =
"10"), //请求次数
    @HystrixProperty(name = "circuitBreaker.sleepWindowInMilliseconds", value =
"10000"), // 时间窗口期
    @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value =
"60") //失败率达到多少后跳闸
}
public String paymentCircuitBreaker(Integer id){
    if(id < 0){
        throw new RuntimeException("id 不能为负数");
    }
    String serialNumber = IdUtil.simpleUUID();
    return Thread.currentThread().getName() + "\t" + "调用成功, 流水号: " +
serialNumber;
}

public String paymentCircuitBreaker_fallback(Integer id){
    return "id 不能为负数, 请稍后再试, /(ㄒoㄒ)/~~ id = " + id;
}

```

更多@HystrixProperty配置可以查看类HystrixCommandProperties.java

现象：多次错误，然后慢慢正确，发现刚开始不满足条件，就算是正确的访问地址也不能进行

断路器在什么情况下开始起作用

涉及到断路器的三个重要参数：**快照时间窗、请求总数阀值、错误百分比阀值**。

1: 快照时间窗：断路器确定是否打开需要统计一些请求和错误数据，而统计的时间范围就是快照时间窗，默认为最近的10秒。

2: 请求总数阀值：在快照时间窗内，必须满足请求总数阀值才有资格熔断。默认为20，意味着在10秒内，如果该hystrix命令的调用次数不足20次，即使所有的请求都超时或其他原因失败，断路器都不会打开。

3: 错误百分比阀值：当请求总数在快照时间窗内超过了阀值，比如发生了30次调用，如果在这30次调用中，有15次发生了超时异常，也就是超过50%的错误百分比，在默认设定50%阀值情况下，这时候就会将断路器打开。

Hystrix图形化Dashboard监控

参照模块 cloud-consumer-hystrix-dashboard9001

引入依赖

```

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

```

主类添加注解：@EnableHystrixDashboard

被监控模块主类：

```

@SpringBootApplication
@EnableEurekaClient
@EnableCircuitBreaker
public class PaymentHystrixMain8001 {
    public static void main(String[] args) {

```

```

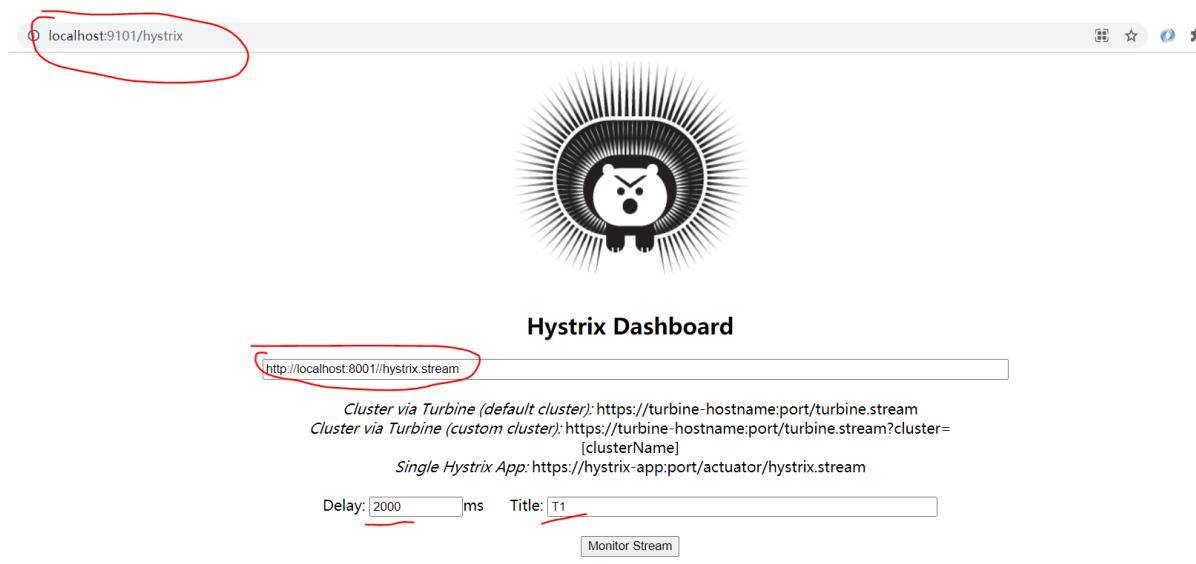
        SpringApplication.run(PaymentHystrixMain8001.class, args);
    }

    /**
     * 此配置是为了服务监控而配置，与服务本身容错无关，springcloud升级后的坑
     * ServletRegistrationBean因为springboot的默认路径不是"/hystrix.stream",
     * 只要在自己的项目里配置上下面的servlet就可以了
     * @return
     */
    @Bean
    public ServletRegistrationBean getServlet(){
        HystrixMetricsStreamServlet streamServlet = new
        HystrixMetricsStreamServlet();
        ServletRegistrationBean registrationBean = new
        ServletRegistrationBean(streamServlet);
        registrationBean.setLoadOnStartup(1);
        registrationBean.addUrlMappings("/hystrix.stream");
        registrationBean.setName("HystrixMetricsStreamServlet");
        return registrationBean;
    }
}

```

启动9001监控模块：

访问：<http://localhost:9001/hystrix>



Hystrix Stream: T1



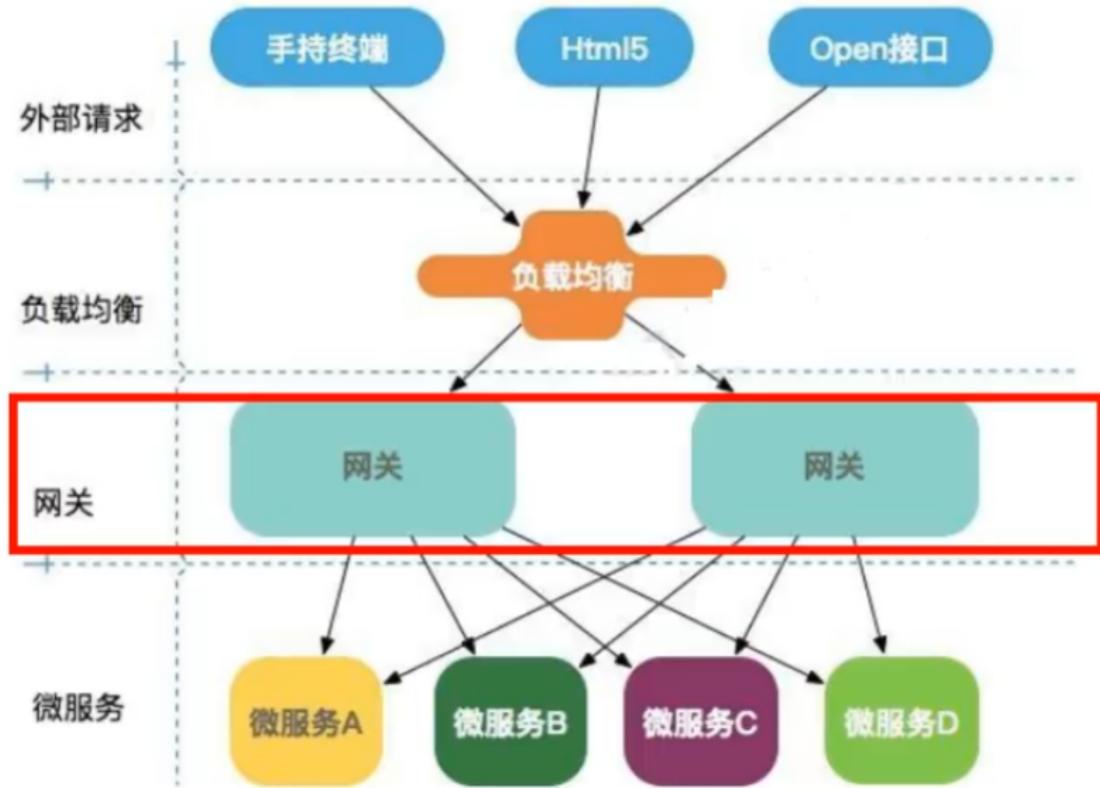
Gateway网关

Cloud全家桶中有个很重要的组件就是网关，在1.x版本中都是采用的Zuul网关；但在2.x版本中，zuul的升级一直跳票，SpringCloud最后自己研发了一个网关替代zuul，就是Spring Cloud Gateway；一句话，gateway是原zuul1.x的替代。

SpringCloud Gateway 是 Spring Cloud 的一个全新项目，基于 Spring 5.0+Spring Boot 2.0 和 Project Reactor 等技术开发的网关，它旨在为微服务架构提供一种简单有效的统一的 API 路由管理方式。

SpringCloud Gateway 作为 Spring Cloud 生态系统中的网关，目标是替代 Zuul，在Spring Cloud 2.0以上版本中，没有对新版本的Zuul 2.0以上最新高性能版本进行集成，仍然还是使用的Zuul 1.x非Reactor模式的老版本。而为了提升网关的性能，SpringCloud Gateway是基于WebFlux框架实现的，而WebFlux框架底层则使用了高性能的Reactor模式通信框架Netty。

Spring Cloud Gateway的目标提供统一的路由方式且基于 Filter 链的方式提供了网关基本的功能，例如：安全，监控/指标，和限流。



Spring Cloud Gateway 具有如下特性：

基于Spring Framework 5, Project Reactor 和 Spring Boot 2.0 进行构建;

动态路由：能够匹配任何请求属性；

可以对路由指定 Predicate (断言) 和 Filter (过滤器) ;

集成Hystrix的断路器功能；

集成 Spring Cloud 服务发现功能；

易于编写的 Predicate (断言) 和 Filter (过滤器) ；

请求限流功能；

支持路径重写。

Spring Cloud Gateway 与 Zuul的区别

在SpringCloud Finchley 正式版之前，Spring Cloud 推荐的网关是 Netflix 提供的Zuul：

1、Zuul 1.x, 是一个基于阻塞 I/ O 的 API Gateway

I

2、Zuul 1.x 基于Servlet 2.5使用阻塞架构它不支持任何长连接(如 WebSocket) Zuul 的设计模式和Nginx较像，每次 I/ O 操作都是从工作线程中选择一个执行，请求线程被阻塞到工作线程完成，但是差别是Nginx 用C++ 实现，Zuul 用 Java 实现，而 JVM 本身会有第一次加载较慢的情况，使得Zuul 的性能相对较差。

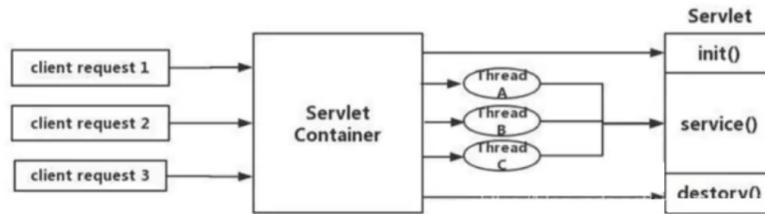
3、Zuul 2.x理念更先进，想基于Netty非阻塞和支持长连接，但SpringCloud目前还没有整合。Zuul 2.x的性能较 Zuul 1.x 有较大提升。在性能方面，根据官方提供的基准测试，Spring Cloud Gateway 的 RPS (每秒请求数) 是Zuul 的 1.6 倍。

4、Spring Cloud Gateway 建立在 Spring Framework 5、Project Reactor 和 Spring Boot 2 之上，使用非阻塞 API。

5、Spring Cloud Gateway 还 支持 WebSocket，并且与Spring紧密集成拥有更好的开发体验

Springcloud中所集成的Zuul版本，采用的是Tomcat容器，使用的是传统的Servlet IO处理模型。

I
学过尚硅谷web中期课程都知道一个题目，[Servlet的生命周期?](#) servlet由servlet container进行生命周期管理。
container启动时构造servlet对象并调用servlet init()进行初始化；
container运行时接受请求，并为每个请求分配一个线程（一般从线程池中获取空闲线程）然后调用service().
container关闭时调用servlet destroy()销毁servlet；



上述模式的缺点：

servlet是一个简单的网络IO模型，当请求进入servlet container时，servlet container就会为其绑定一个线程，在并发不高的场景下这种模型是适用的。但是一旦高并发(比如抽风用jmeter压)，线程数量就会上涨，而线程资源代价是昂贵的（上线文切换，内存消耗大）严重影响请求的处理时间。在一些简单业务场景下，不希望为每个request分配一个线程，只需要1个或几个线程就能应对极大并发的请求，这种业务场景下servlet模型没有优势

所以Zuul 1.X是[基于servlet之上的一个阻塞式处理模型](#)，即spring实现了处理所有request请求的一个servlet (DispatcherServlet) 并由该servlet阻塞式处理处理。所以Springcloud Zuul无法摆脱servlet模型的弊端

传统的Web框架，比如说：struts2，springmvc等都是基于Servlet API与Servlet容器基础之上运行的。

但是

[在Servlet3.1之后有了异步非阻塞的支持](#)。而WebFlux是一个典型非阻塞异步的框架，它的核心是基于Reactor的相关API实现的。相对于传统的web框架来说，它可以运行在诸如Netty，Undertow及支持Servlet3.1的容器上。非阻塞式+函数式编程（Spring5必须让你使用java8）

Spring WebFlux 是 Spring 5.0 引入的新的响应式框架，区别于 Spring MVC，它不需要依赖Servlet API，它是完全异步非阻塞的，并且基于 Reactor 来实现响应式流规范。

三大核心概念

Route(路由)：是构建网关的基本模块，它由ID，目标URI，一系列的断言和过滤器组成，如果断言为true则匹配该路由

Predicate(断言)：开发人员可以匹配HTTP请求中的所有内容(例如请求头或请求参数)，如果请求与断言相匹配则进行路由

Filter(过滤)：指的是Spring框架中GatewayFilter的实例，使用过滤器，可以在请求被路由前或后对请求进行修改

gateway工作流程

客户端向Spring Cloud Gateway发出请求，然后在Gateway Handler Mapping中找到与请求相匹配的路由，将其发送到Gateway Web Handler。

Handler再通过指定的过滤器链来将请求发送到我们实际的服务执行业务逻辑，然后返回。

过滤器之间用虚线分开是因为过滤器可能会在发送代理请求之前("pre")或之后("post")执行业务逻辑。

Filter在"pre"类型的过滤器可以做参数校验、权限校验、流量监控、日志输出、协议转换等，

在"post"类型的过滤器中可以做响应内容、响应头的修改，日志的输出、流量监控等有着非常重要的作用。

核心逻辑：路由转发+执行过滤器链

