微服务网关背景及简介

不同的微服务一般有不同的网络地址，而外部的客户端可能需要调用多个服务的接口才能完成一个业务需求。比如一个电影购票的收集APP,可能回调用电影分类微服务，用户微服务，支付微服务等。如果客户端直接和微服务进行通信，会存在一下问题：

客户端会多次请求不同微服务，增加客户端的复杂性

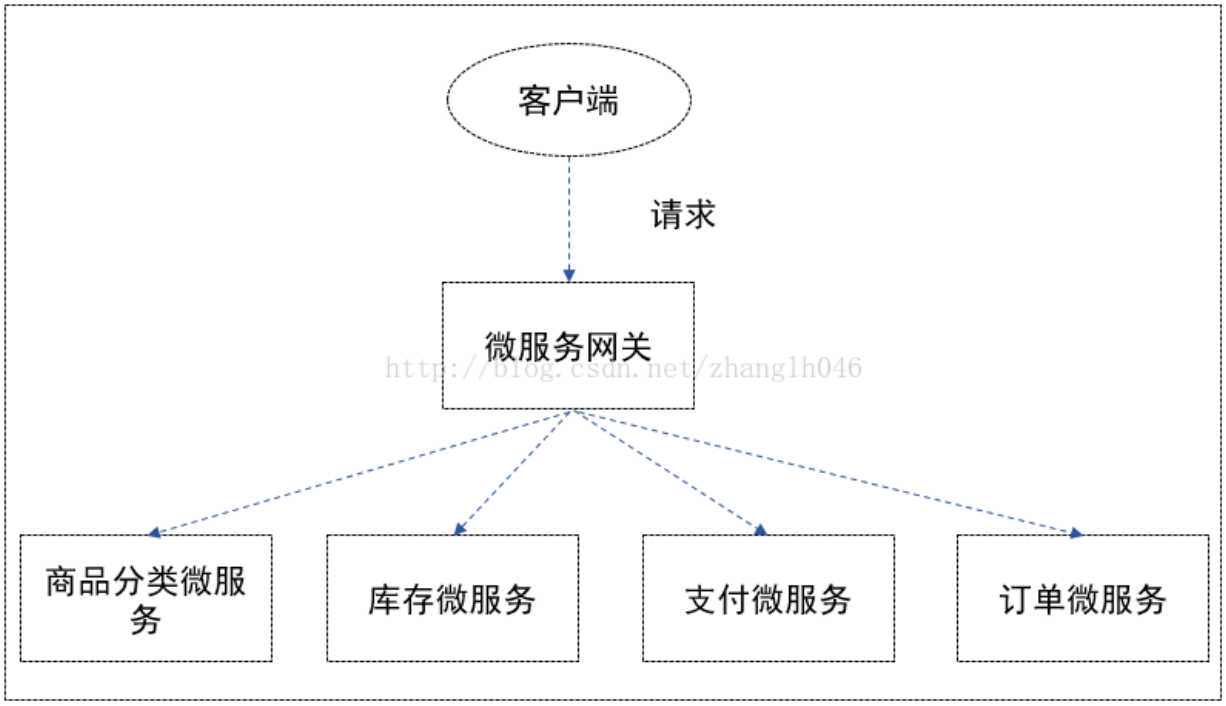
存在跨域请求，在一定场景下处理相对复杂

认证复杂，每一个服务都需要独立认证

难以重构，随着项目的迭代，可能需要重新划分微服务，如果客户端直接和微服务通信，那么重构会难以实施

某些微服务可能使用了其他协议，直接访问有一定困难

上述问题，都可以借助微服务网管解决。微服务网管是介于客户端和服务器端之间的中间层，所有的外部请求都会先经过微服务网关，架构演变成：



这样客户端只需要和网关交互，而无需直接调用特定微服务的接口，而且方便监控，易于认证，减少客户端和各个微服务之间的交互次数

zuul微服务网关

Zuul是Netflix开源的微服务网关，他可以和Eureka,Ribbon,Hystrix等组件配合使用。Zuul组件的核心是一系列的过滤器，这些过滤器可以完成以下功能：

动态路由：动态将请求路由到不同后端集群

压力测试：逐渐增加指向集群的流量，以了解性能

负载分配：为每一种负载类型分配对应容量，并弃用超出限定值的请求
静态响应处理：边缘位置进行响应，避免转发到内部集群
身份认证和安全: 识别每一个资源的验证要求，并拒绝那些不符的请求
Spring Cloud对Zuul进行了整合和增强。目前，Zuul使用的默认是Apache的HTTP Client，也可以使用Rest Client，可以设置ribbon.restclient.enabled=true.

编写一个简单微服务网关，见示例：08-ms-gateway-zuul
添加依赖并在启动类上增加注解@EnableZuulProxy，声明一个zuul代理。该代理使用Ribbon来定位注册在 Eureka server中的微服务；同时，该代理还整合了Hystrix，从而实现了容错，所有经过 zuul的请求都会在 Hystrix命令中执行。

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

配置文件application.yml如下：

```
server:
  port: 8040
spring:
  application:
    name: microservice-gateway-zuul
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8761/eureka/
  instance:
    prefer-ip-address: true
```

运行项目(**需启动两个用户微服务和一个订单微服务**，参看视频)，访问地址：
http://localhost:8040/microservice-provider-user/getIpAndPort
发现zuul会代理所有注册到eureka server的微服务，并使用ribbon做负载均衡，zuul的路由规则如下，可以访问地址：http://localhost:8040/routes
http://ZUUL_HOST:ZUUL_PORT/微服务在Eureka上的serviceId/**会被转发到serviceId对应的微服务
zuul还自动整合了hystrix，访问地址：http://localhost:8040/hystrix.stream

zuul聚合微服务

许多场景下，外部请求需要查询 zuul后端的多个微服务。举个例子，一个电影售票手机APP,在购票订单页上，既需要查询“电影微服务"获得电影相关信息，又需要查询“用户微

服务"获得当前用户的信息。如果让手机端直接请求各个微服务（即使使用 zuul进行转发），那么网络开销、流量耗费、耗费时长可能都无法令我们满意。那么对于这种场景，可使用 zuul聚合微服务请求————手机 APP只需发送一个请求给 zuul,由 zuul请求用户微服务以及电影微服务，并组织好数据给手机 APP，使用这种方式，手机端只须发送一次请求即可，简化了客户端侧的开发；不仅如此，由于 zuul、用户微服务、电影微服务一般都在同一个局域网中，因此速度会非常快，效率会非常高。

见示例：08-ms-gateway-zuul-aggregation

```
@RestController
public class AggregationController {
  public static final Logger LOGGER = LoggerFactory.getLogger(ZuulApplication.class);

  @Autowired
  private RestTemplate restTemplate;

  @GetMapping("/aggregate/{id}")
  public Map<String, User> findById(@PathVariable Long id) throws Exception {
      Map<String, User> dataMap = new HashMap<>();
      User user = restTemplate.getForObject("http://microservice-provider-user/" + id, User.class);
      User orderUser = restTemplate.getForObject("http://microservice-consumer-order/user/" + id, User.class);
      dataMap.put("user", user);
      dataMap.put("orderUser", orderUser);
      return dataMap;
  }
}
```

zuul的路由配置

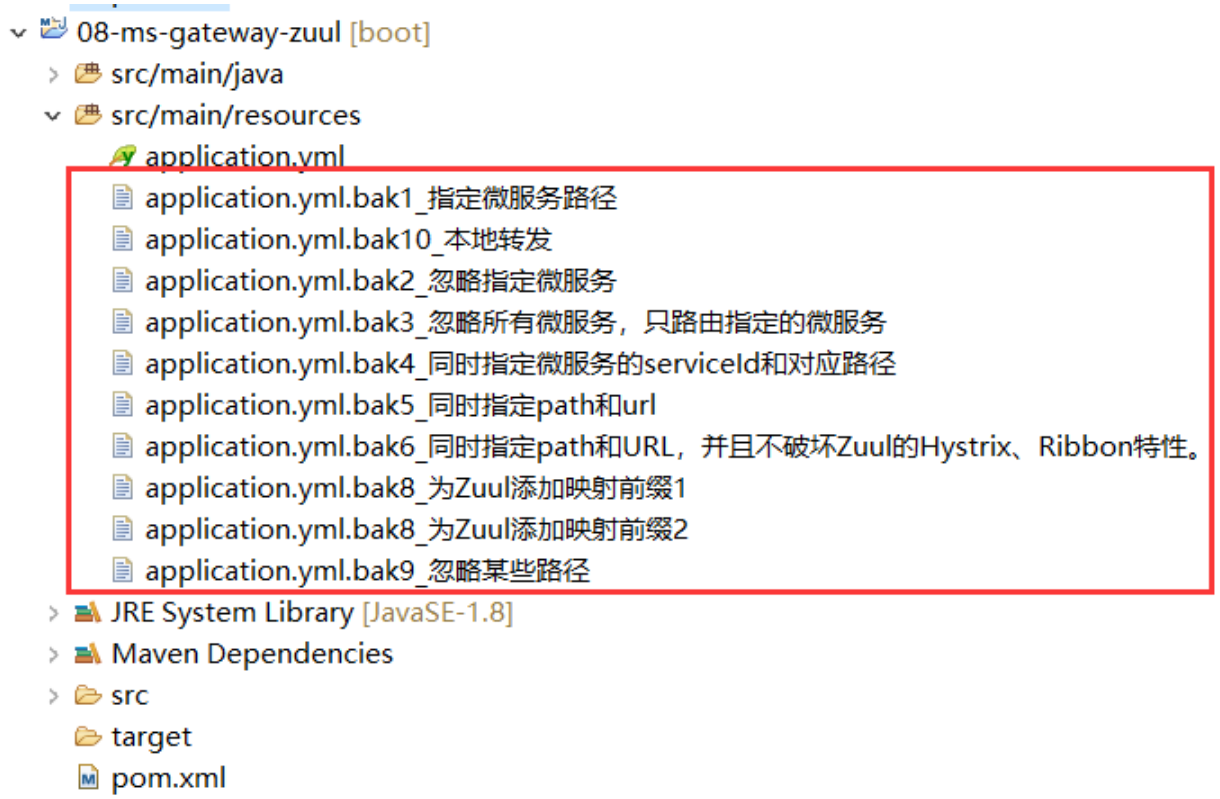
前文已经编写了一个简单的 zuul网关，并让该网关代理了所有注册到 Eureka server的微服务。但在现实中可能只想让 zuul代理部分微服务，又或者需要对 URL进行更加精确的控制。

见示例：08-ms-gateway-zuul

配置忽略指定微服务，只需在application.yml里加上如下配置

```
zuul:
  ignored-services: microservice-provider-user,microservice-consumer-movie
```

其他路由配置课查看项目示例的配置文件



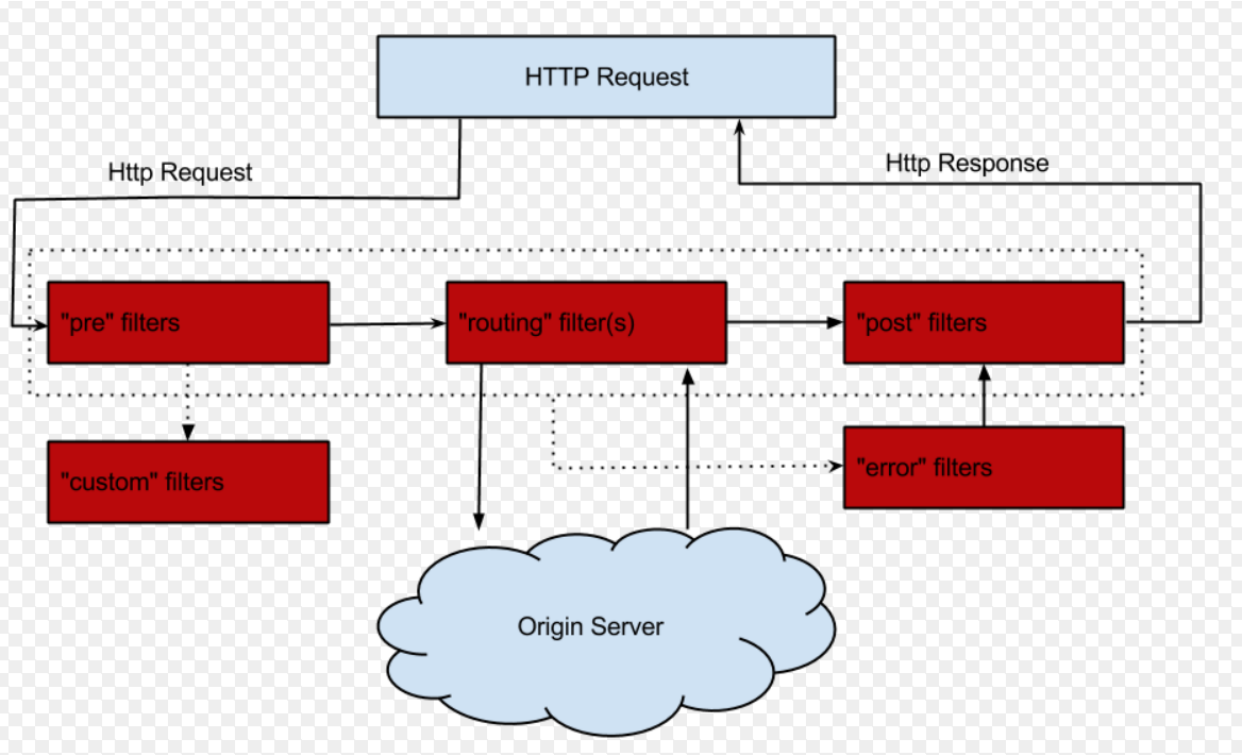
zuul过滤器使用及详解

过滤器(filter)是zuul的核心组件

zuul大部分功能都是通过过滤器来实现的。 zuul中定义了4种标准过滤器类型，这些过滤器类型对应于请求的典型生命周期。

- PRE：这种过滤器在请求被路由之前调用。可利用这种过滤器实现身份验证、在集群中选择请求的微服务、记录调试信息等。
- ROUTING：这种过滤器将请求路由到微服务。这种过滤器用于构建发送给微服务的请求，并使用 Apache HttpClient或 Netfilx Ribbon请求微服务
- POST:这种过滤器在路由到微服务以后执行。这种过滤器可用来为响应添加标准的 HTTP Header、收集统计信息和指标、将响应从微服务发送给客户端等。
- ERROR：在其他阶段发生错误时执行该过滤器。

zuul请求的生命周期如下图



编写一个过滤器，见示例：08-ms-gateway-zuul-filter

增加过滤器类<PreRequestLogFilter>，需继承抽象类<ZuulFilter>，然后实现几个抽象方法

```
public class PreRequestLogFilter extends ZuulFilter {
  private static final Logger LOGGER = LoggerFactory.getLogger(PreRequestLogFilter.class);

  @Override
  public String filterType() {
    return FilterConstants.PRE_TYPE;
  }

  @Override
  public int filterOrder() {
    return FilterConstants.PRE_DECORATION_FILTER_ORDER - 1;
  }

  @Override
  public boolean shouldFilter() {
    return true;
  }

  @Override
  public Object run() {
    RequestContext ctx = RequestContext.getCurrentContext();
    HttpServletRequest request = ctx.getRequest();
    PreRequestLogFilter.LOGGER.info(String.format("send %s request to %s", request.getMethod(),
    return null;
  }
}
```

由代码可知，自定义的 zuul Filter需实现以下几个方法。

- 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。

- filter0rder: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。本例中让它打印了请求的 HTTP方法以及请求的地址

运行项目访问地址：http://localhost:8040/microservice-provider-user/getIpAndPort
可以看到zuul服务的后台正常打印了run方法里的日志

禁用zuul过滤器

Spring Cloud默认为Zuul编写并启用了一些过滤器，例如DebugFilter、FormBodyWrapperFilter等，这些过滤器都存放在spring-cloud-netflix-core这个jar包里，一些场景下，想要禁用掉部分过滤器，该怎么办呢?
只需在application.yml里设置zuul.<SimpleClassName>.<filterType>.disable=true
例如，要禁用上面我们写的过滤器，这样配置就行了：
zuul.PreRequestLogFilter.pre.disable=true

zuul的容错与回退

大家可以想一下如果zuul代理的后端微服务挂了会出现什么情况?
zuul默认已经整合了hystrix，也就是zuul也是可以利用hystrix做降级容错处理的，但是zuul监控的粒度是微服务级别，而不是某个API
见示例：08-ms-gateway-zuul-fallback
编写zuul的降级回退类<MyFallbackProvider>如下：

```java
@Component
public class MyFallbackProvider implements FallbackProvider {
  @Override
  public String getRoute() {
    // 表明是为哪个微服务提供回退，*表示为所有微服务提供回退
    return "*";
  }

  @Override
  public ClientHttpResponse fallbackResponse(Throwable cause) {
    if (cause instanceof HystrixTimeoutException) {
      return response(HttpStatus.GATEWAY_TIMEOUT);
    } else {
      return this.fallbackResponse();
    }
  }

  @Override
  public ClientHttpResponse fallbackResponse() {
    return this.response(HttpStatus.INTERNAL_SERVER_ERROR);
  }

  private ClientHttpResponse response(final HttpStatus status) {
    return new ClientHttpResponse() {
      @Override
      public HttpStatus getStatusCode() throws IOException {
        //fallback时的状态码
        return status;
      }

      @Override
      public int getRawStatusCode() throws IOException {
        //数字类型的状态码，本例返回的其实就是200，详见HttpStatus
```

关闭zuul代理的用户微服务，再运行本项目，访问地址：http://localhost:8040/microservice-provider-user/getIpAndPort，将会返回如下内容：

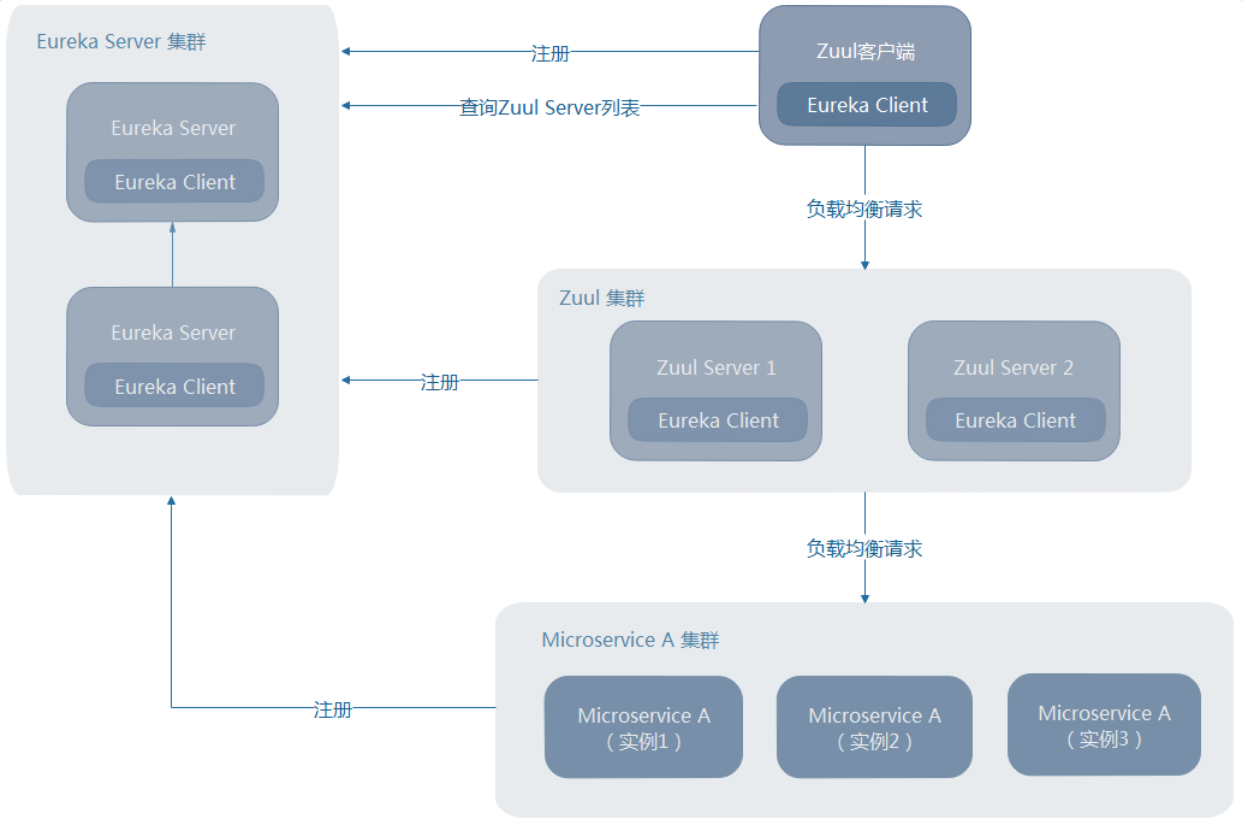
服务不可用，请稍后再试

zuul的高可用

分两种场景讨论Zuul的高可用

1、Zuul客户端也注册到了Eureka Server上

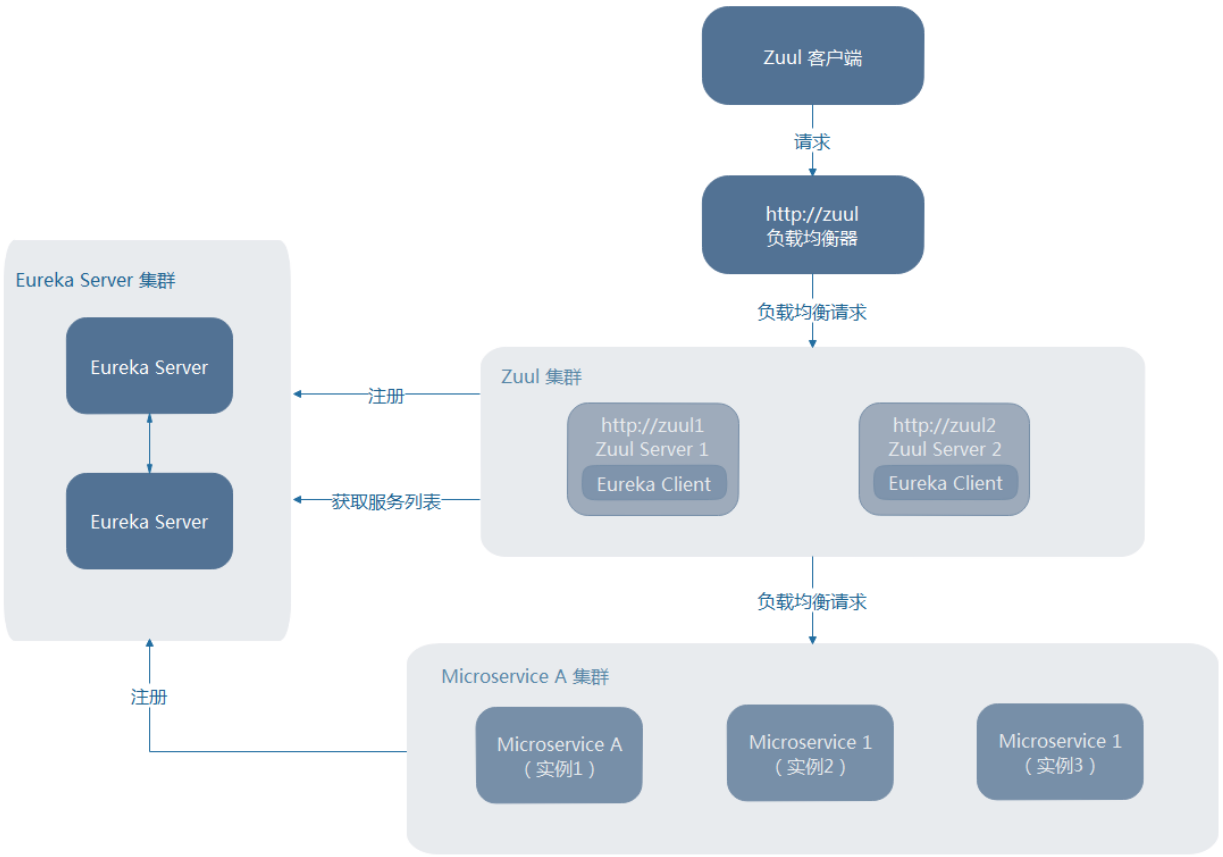
这种情况下，Zuul的高可用非常简单，只需将多个Zuul节点注册到Eureka Server上，就可实现Zuul的高可用。此时，Zuul的高可用与其他微服务的高可用没什么区别。见下图



当Zuul客户端也注册到Eureka Server上时，只需部署多个Zuul节点即可实现其高可用。Zuul客户端会自动从Eureka Server中查询Zuul Server的列表，并使用Ribbon负载均衡地请求Zuul集群。

2、Zuul客户端未注册到Eureka Server上

现实中，这种场景往往更常见，例如，Zuul客户端是一个手机APP——我们不可能让所有的手机终端都注册到Eureka Server上。这种情况下，我们可借助一个额外的负载均衡器来实现Zuul的高可用，例如Nginx、HAProxy、F5等。



Zuul客户端将请求发送到负载均衡器，负载均衡器将请求转发到其代理的其中一个Zuul节点。这样，就可以实现Zuul的高可用。

补充：

1、feign接口部分方法禁用hystrix

2、**springboot-acuator使用**

3、eureka集群

4、dashboard（实时监控）怎么根据历史报错数据做运维监控

springboot-actuator监控

springboot-actuator提供了很多系统的监控端点，如下所示：

ID	描述	敏感（Sensi
autoconfig	显示一个auto-configuration的报告，该报告展示所有auto-configuration候选者及它们被应用或未被应用的原因	true
beans	显示一个应用中所有Spring Beans的完整列表	true
configprops	显示一个所有@ConfigurationProperties的	true

	整理列表	
dump	执行一个线程转储	true
env	暴露来自Spring ConfigurableEnvironment的属性	true
health	展示应用的健康信息（当使用一个未认证连接访问时显示一个简单的’status’，使用认证连接访问则显示全部信息详情）	false
info	显示任意的应用信息	false
metrics	展示当前应用的’指标’信息	true
mappings	显示一个所有@RequestMapping路径的整理列表	true
shutdown	允许应用以优雅的方式关闭（默认情况下不启用）	true
trace	显示trace信息（默认为最新的一些HTTP请求）	true

见示例：08-ms-provider-user

在项目中添加依赖：

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

增加配置：

```
management:
  security:
    enabled: false  #关掉安全认证
  port: 8888  #管理端口调整成8888,独立的端口可以做安全控制
  context-path: /monitor  #actuator的访问路径
```

Spring Boot Admin监控

Spring Boot Admin 是一个管理和监控Spring Boot 应用程序的开源软件，它针对spring-boot的actuator接口进行UI美化封装

搭建spring boot admin的服务端，见示例：08-ms-spring-boot-admin

添加依赖并在启动类上增加注解@EnableAdminServer

```
  <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server</artifactId>
      <version>1.5.6</version>
  </dependency>
  <dependency>
      <groupId>de.codecentric</groupId>
      <artifactId>spring-boot-admin-server-ui</artifactId>
      <version>1.5.6</version>
  </dependency>
```

```
@Configuration
@EnableAutoConfiguration
@EnableAdminServer
public class SpringBootAdminApplication {

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

搭建spring boot admin的客户端，见示例：08-ms-provider-user

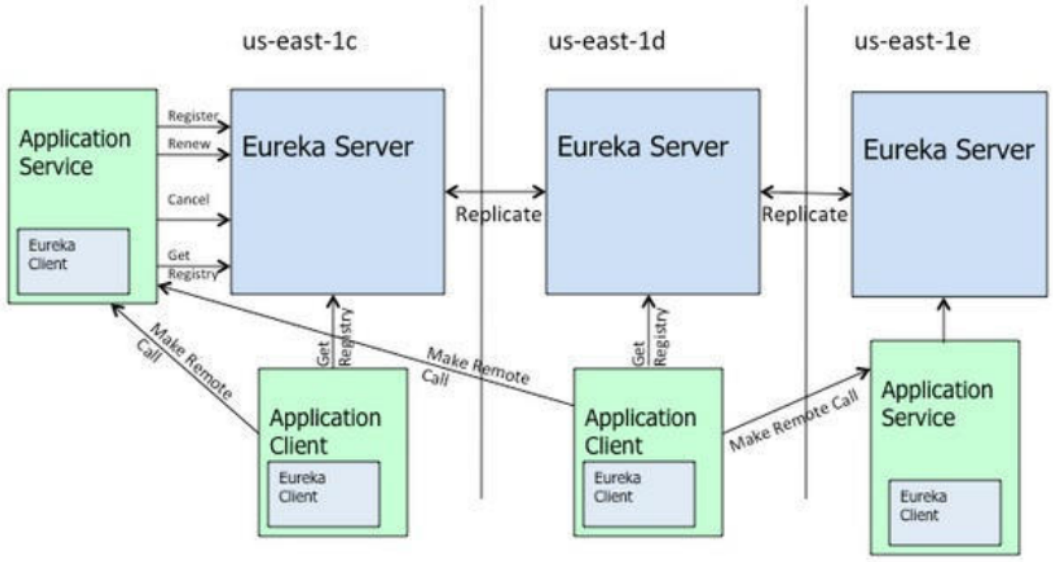
添加依赖

```
<!-- spring boot admin监控客户端依赖 -->
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
  <version>1.5.6</version>
</dependency>
```

增加application.yml配置：

spring.boot.admin.url=http://localhost:9999　# spring boot admin服务端地址，搜集客户端监控数据

eureka集群



上图是来自eureka的官方架构图，这是基于集群配置的eureka；

- 处于不同节点的eureka通过Replicate进行数据同步
- Application Service为服务提供者
- Application Client为服务消费者
- Make Remote Call完成一次服务调用

服务启动后向Eureka注册，Eureka Server会将注册信息向其他Eureka Server进行同步，当服务消费者要调用服务提供者，则向服务注册中心获取服务提供者地址，然后会将服务提供者地址缓存在本地，下次再调用时，则直接从本地缓存中取，完成一次调用。

见示例：08-ms-eureka-server-ha

引入依赖

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

application.yml配置如下(见配置文件application-ha.yml)

```
spring:
  application:
    name: microservice-eureka-server-ha
---
spring:
  profiles: peer1                                 # 指定profile=peer1
server:
  port: 8761
eureka:
  instance:
    hostname: peer1                               # 指定当profile=peer1时，主机名是peer1
  client:
    serviceUrl:
      defaultZone: http://peer2:8762/eureka/      # 将自己注册到peer2这个Eureka上面去

---
spring:
  profiles: peer2
server:
  port: 8762
eureka:
  instance:
    hostname: peer2
  client:
    serviceUrl:
      defaultZone: http://peer1:8761/eureka/
```

配置host：127.0.0.1 peer1 peer2

用jar包方式启动：

java -jar 项目的jar包 --spring.profiles.active=peer1