

Spring Cloud Netflix Zuul 服务网关



历史修订

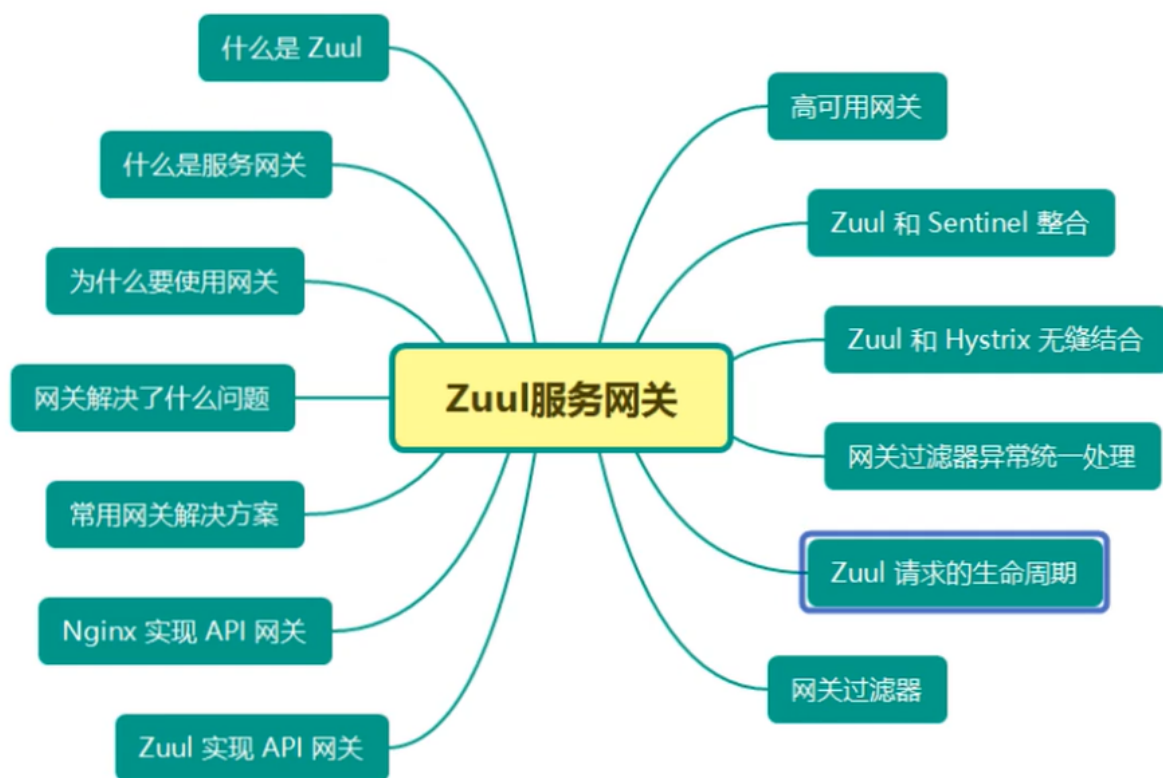
本次修订日期: 2023-03-03	下次修订日期:
--------------------	---------

修订编号	修订日期	变更描述	说明
V0.1	2023-03-03	起草	李宗在
V0.2	2023-03-11	测试验证	李宗在
V0.3			

1. 技术介绍

- Spring Boot
- Spring Cloud
- Netflix Zuul
- Feign
- Mybatis/Mybatis-Plus
- MySQL
- Docker
- Ubuntu
- Redis
- Postman
- swagger

2. 学习目标



3. 什么是Zuul

Zuul 是从设备和网站到应用程序后端的所有请求的前门。作为边缘服务应用程序，Zuul 旨在实现动态路由，监视，弹性和安全性。Zuul 包含了对请求的**路由**和**过滤**两个最主要的功能。

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

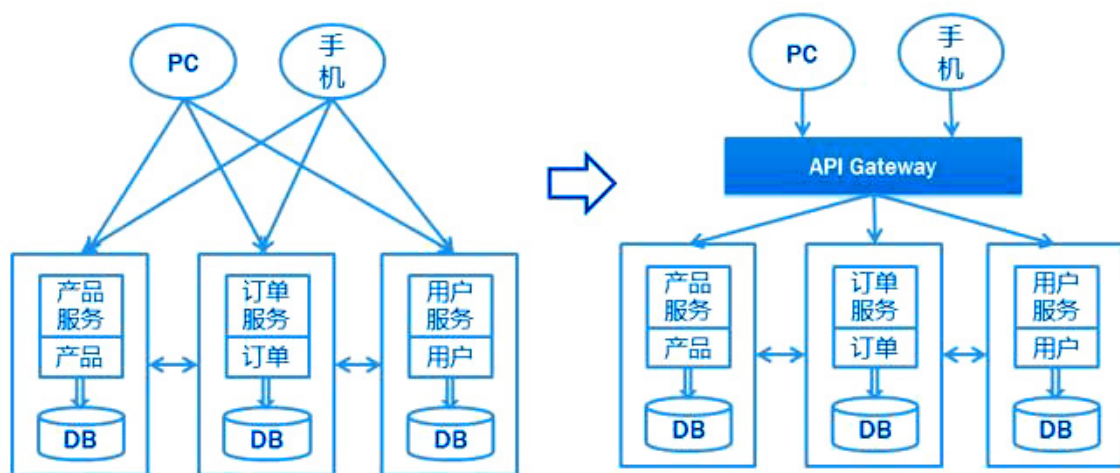
- **身份认证与安全**：识别每个资源的验证要求，并拒绝那些与要求不符的请求
- **审查与监控**：在边缘位置追踪有意义的数据和统计结果，从而带来精确的生产视图
- **动态路由**：动态地将请求路由到不同的后端集群
- **压力测试**：逐渐增加只想集群的流量，以了解性能
- **负载分配**：为每一种负载类型分配对应容量，并弃用超出限定值的请求
- **静态响应处理**：在边缘位置直接建立部份响应，从而避免其转发到内部集群
- **多区域弹性**：跨越AWS Region进行请求路由，旨在实现ELB（Elastic Load Balancing）使用的多样化，以及让系统的边缘更贴近系统的使用者

4. 什么是服务网关

API Gateway（APIGW / API 网关），顾名思义，是出现在系统边界上的一个面向 API 的、串行集中式的强管控服务，这里的边界是企业 IT 系统的边界，可以理解为 **企业级应用防火墙**，主要起到 **隔离外部访问与内部系统的作用**。在微服务概念的流行之前，API 网关就已经诞生了，例如银行、证券等领域常见的前置机系统，它也是解决访问认证、报文转换、访问统计等问题的。

API 网关的流行，源于近几年来移动应用与企业间互联需求的兴起。移动应用、企业互联，使得后台服务支持的对象，从以前单一的 Web 应用，扩展到多种使用场景，且每种使用场景对后台服务的要求都不尽相同。这不仅增加了后台服务的响应量，还增加了后台服务的复杂性。随着微服务架构概念的提出，API 网关成为了微服务架构的一个标配组件。

API 网关是一个服务器，是系统对外的唯一入口。API 网关封装了系统内部架构，为每个客户端提供定制的 API。所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有非业务功能。API 网关并不是微服务场景中必须的组件，如下图，不管有没有 API 网关，后端微服务都可以通过 API 很好地支持客户端的访问。



但对于服务数量众多、复杂度比较高、规模比较大的业务来说，引入 API 网关也有一系列的好处：

- 聚合接口使得服务对调用者透明，客户端与后端的耦合度降低
- 聚合后台服务，节省流量，提高性能，提升用户体验
- 提供安全、流控、过滤、缓存、计费、监控等 API 管理功能

5. 为什么使用网关

- 单体应用：浏览器发起请求到单体应用所在的机器，应用从数据库查询数据原路返回给浏览器，对于单体应用来说是不需要网关的。
- 微服务：微服务的应用可能部署在不同机房，不同地区，不同域名下。此时客户端（浏览器/手机/软件工具）想要请求对应的服务，都需要知道机器的具体 IP 或者域名 URL，当微服务实例众多时，这是非常难以记忆的，对于客户端来说也太复杂难以维护。此时就有了网关，客户端相关的请求直接发送到网关，由网关根据请求标识解析判断出具体的微服务地址，再把请求转发到微服务实例。这其中的记忆功能就全部交由网关来操作了。



总结

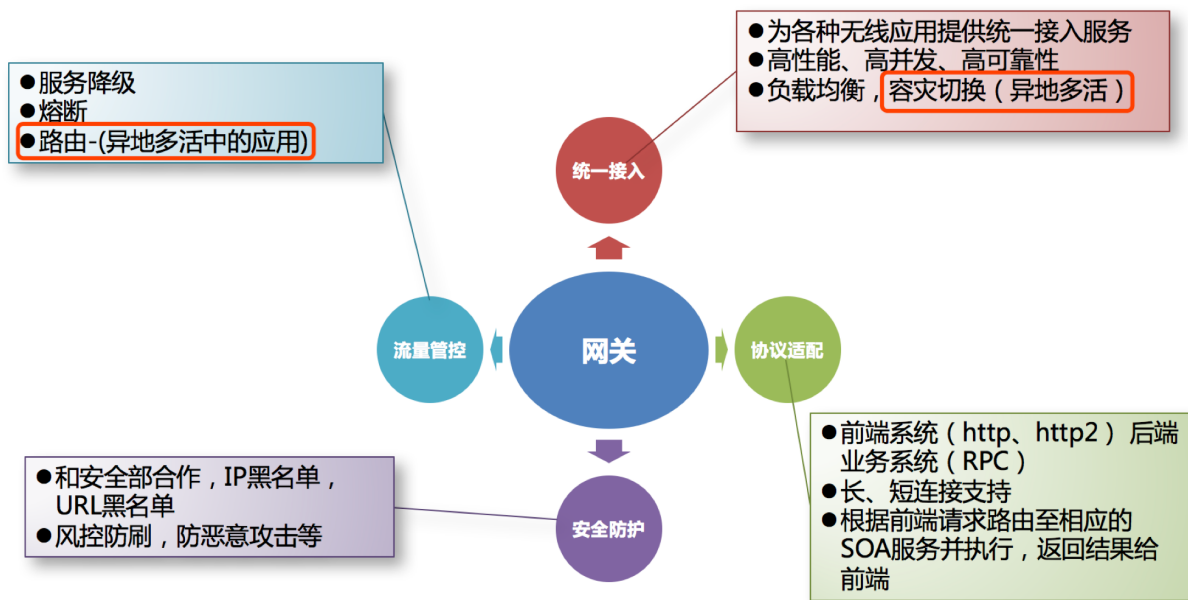
如果让客户端直接与各个微服务交互：

- 客户端会多次请求不同的微服务，增加了客户端的复杂性
- 存在跨域请求，在一定场景下处理相对复杂
- 身份认证问题，每个微服务需要独立身份认证
- 难以重构，随着项目的迭代，可能需要重新划分微服务
- 某些微服务可能使用了防火墙/浏览器不友好的协议，直接访问会有一些困难

因此，我们需要网关介于客户端与服务器之间的中间层，所有外部请求率先经过微服务网关，客户端只需要与网关交互，只需要知道网关地址即可。这样便简化了开发且有以下优点：

- 易于监控，可在微服务网关收集监控数据并将其推送到外部系统进行分析
- 易于认证，可在微服务网关上进行认证，然后再将请求转发到后端的微服务，从而无需在每个微服务中进行认证
- 减少了客户端与各个微服务之间的交互次数

6. 网关解决了什么问题



网关具有身份认证与安全、审查与监控、动态路由、负载均衡、缓存、请求分片与管理、静态响应处理等功能。当然最主要的职责还是与“外界联系”。

总结一下，网关应当具备以下功能：

- 性能：API 高可用，负载均衡，容错机制。
- 安全：权限身份认证、脱敏，流量清洗，后端签名（保证全链路可信调用），黑名单（非法调用的限制）。
- 日志：日志记录，一旦涉及分布式，全链路跟踪必不可少。
- 缓存：数据缓存。
- 监控：记录请求响应数据，API 耗时分析，性能监控。
- 限流：流量控制，错峰流控，可以定义多种限流规则。
- 灰度：线上灰度部署，可以减小风险。
- 路由：动态路由规则。

7. 常用网关解决方案

Nginx 是由 Igor Sysoev 为俄罗斯访问量第二的 Rambler.ru 站点开发的，一个高性能的 HTTP 和反向代理服务器。Nginx 一方面可以做反向代理，另外一方面做可以做静态资源服务器。

- Nginx 是 C 语言开发，而 Zuul 是 Java 语言开发
- Nginx 负载均衡实现，采用服务器实现负载均衡，而 Zuul 负载均衡的实现是采用 Ribbon + Eureka 来实现本地负载均衡
- Nginx 适合于服务器端负载均衡，Zuul 适合微服务中实现网关
- Nginx 相比 Zuul 功能会更加强大，因为 Nginx 可以整合一些脚本语言（Nginx + Lua）
- Nginx 是一个高性能的 HTTP 和反向代理服务器，也是一个 IMAP / POP3 / SMTP 服务器。Zuul 是 Spring Cloud Netflix 中的开源的一个 API Gateway 服务器，本质上是一个 Servlet 应用，提供动态路由，监控，弹性，安全等边缘服务的框架。Zuul 相当于是从设备和网站到应用程序后端的所有请求的前门。

- Nginx 适合做门户网关，是作为整个全局的网关，对外的处于最外层的那种；而 Zuul 属于业务网关，主要用来对应不同的客户端提供服务，用于聚合业务。各个微服务独立部署，职责单一，对外提供服务的时候需要有一个东西把业务聚合起来。

- Zuul 可以实现熔断、重试等功能，这是 Nginx 不具备的。

7.1 Kong

Kong 是 Mashape 提供的一款 API 管理软件，它本身是基于 Nginx + Lua 的，但比 Nginx 提供了更简单的配置方式，数据采用了 ApacheCassandra/PostgreSQL 存储，并且提供了一些优秀的插件，比如验证，日志，调用频次限制等。Kong 非常诱人的地方就是提供了大量的插件来扩展应用，通过设置不同的插件可以为服务提供各种增强的功能。

优点：基于 Nginx 所以在性能和稳定性上都没有问题。Kong 作为一款商业软件，在 Nginx 上做了很扩展工作，而且还有很多付费的商业插件。Kong 本身也有付费的企业版，其中包括技术支持、使用培训服务以及 API 分析插件。

缺点：如果你使用 Spring Cloud，Kong 如何结合目前已有的服务治理体系？

7.2 Traefik

Traefik 是一个开源的 GO 语言开发的为了让部署微服务更加便捷而诞生的现代 HTTP 反向代理、负载均衡工具。它支持多种后台 (Docker, Swarm, Kubernetes, Marathon, Mesos, Consul, Etcd, Zookeeper, BoltDB, Rest API, file...) 来自动化、动态的应用它的配置文件设置。Traefik 拥有一个基于 AngularJS 编写的简单网站界面，支持 Rest API，配置文件热更新，无需重启进程。高可用集群模式等。

相对 Spring Cloud 和 Kubernetes 而言，目前比较适合 Kubernetes

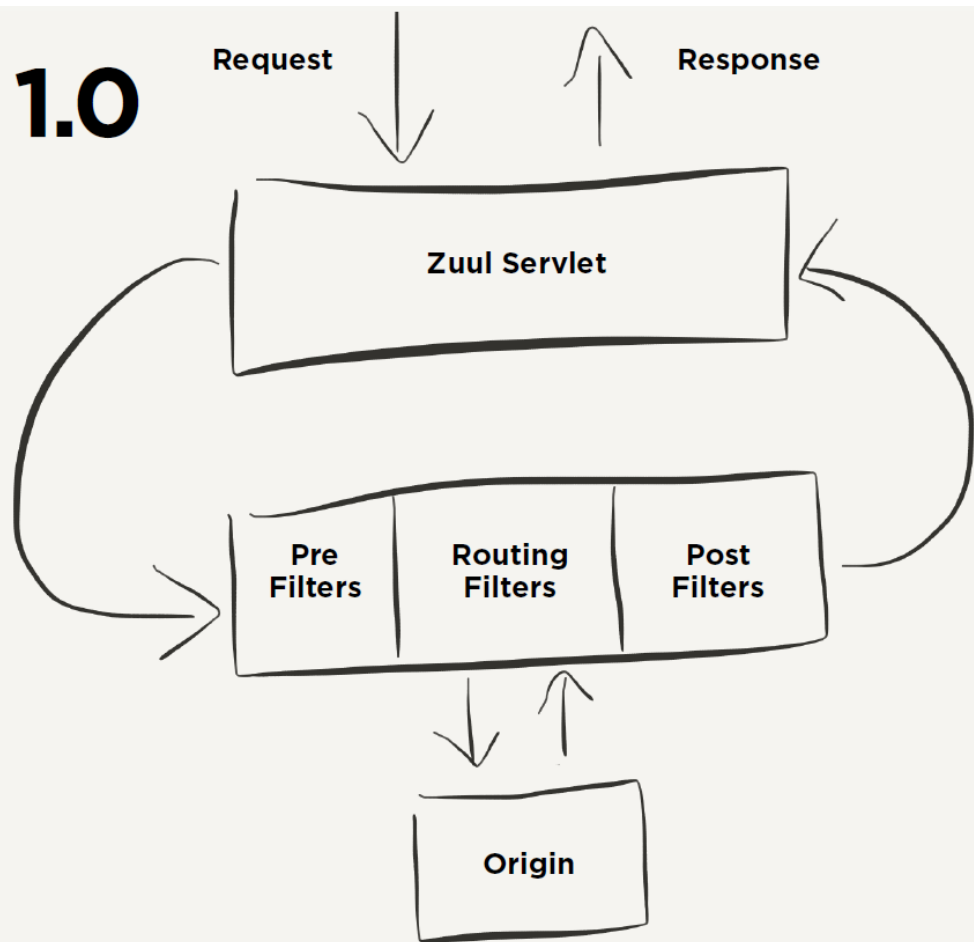
7.3 Spring Cloud Netflix Zuul

Zuul 是 Netflix 公司开源的一个 API 网关组件，Spring Cloud 对其进行二次基于 Spring Boot 的注解式封装做到开箱即用。目前来说，结合 Spring Cloud 提供的服务治理体系，可以做到请求转发，根据配置或者默认的路由规则进行路由和 Load Balance，无缝集成 Hystrix。

虽然可以通过自定义 Filter 实现我们想要的功能，但是由于 Zuul 本身的设计是基于 单线程的接收请求和转发处理，是阻塞 IO，不支持长连接。目前来看 Zuul 就显得很鸡肋，随着 Zuul 2.x 一直跳票（2019 年 5 月发布了 Zuul 2.0 版本），Spring Cloud 推出自己的 Spring Cloud Gateway。

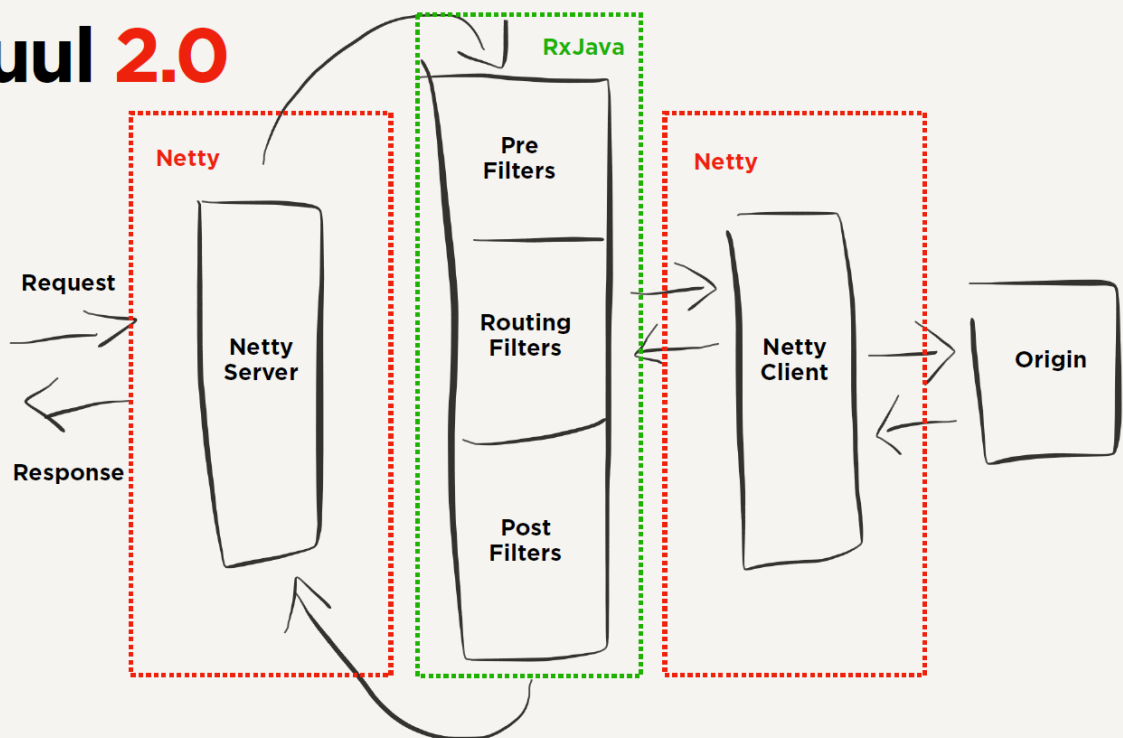
大意就是：Zuul 已死，Spring Cloud Gateway 永生（手动狗头）。

Zuul 1.0



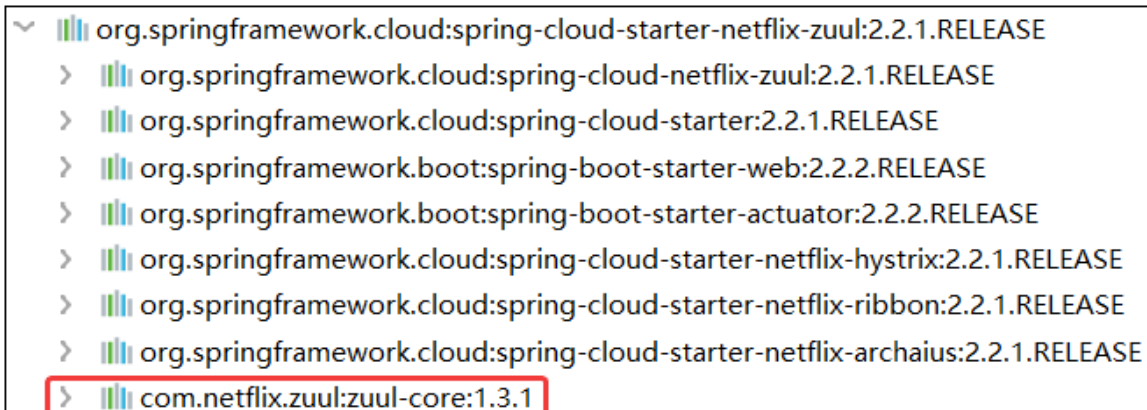
V2.0

Zuul 2.0



7.4 Spring Cloud Gateway

Spring Cloud Gateway 作为 Spring Cloud 生态系统中的网关，目标是替代 Netflix Zuul，其不仅提供统一的路由方式，并且还基于 Filter 链的方式提供了网关基本的功能。目前最新版 Spring Cloud 中引用的还是 Zuul 1.x 版本，而这个版本是基于过滤器的，是阻塞 IO，不支持长连接。



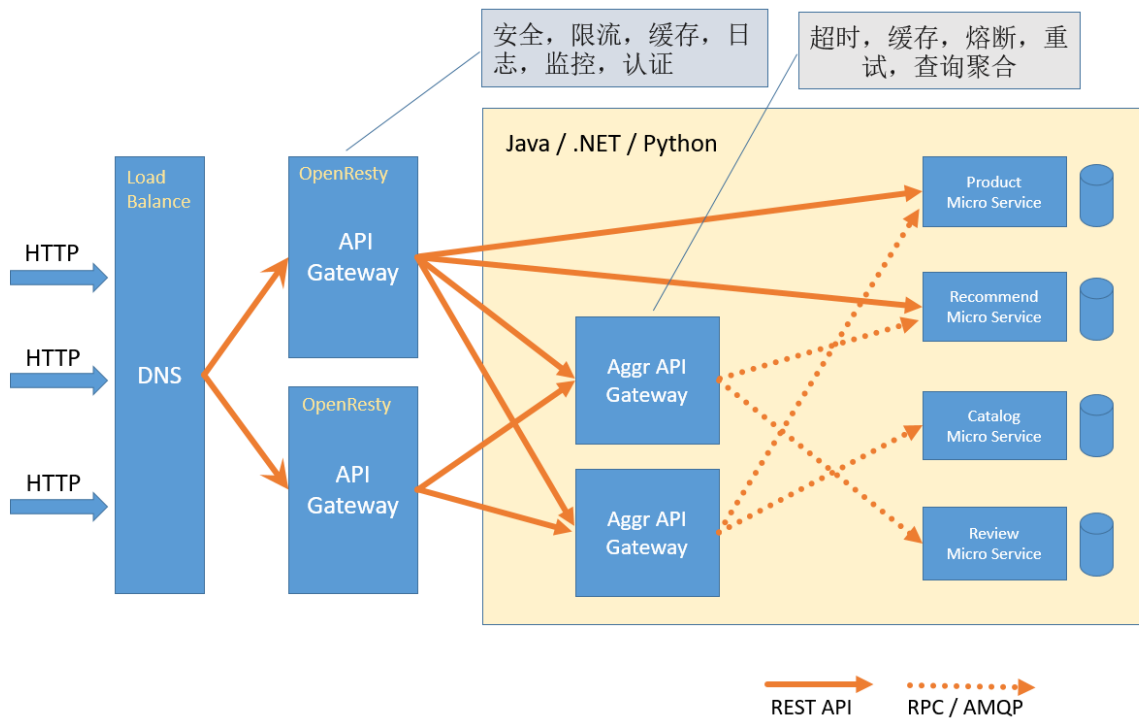
```
~ org.springframework.cloud:spring-cloud-starter-netflix-zuul:2.2.1.RELEASE
  > org.springframework.cloud:spring-cloud-netflix-zuul:2.2.1.RELEASE
  > org.springframework.cloud:spring-cloud-starter:2.2.1.RELEASE
  > org.springframework.boot:spring-boot-starter-web:2.2.2.RELEASE
  > org.springframework.boot:spring-boot-starter-actuator:2.2.2.RELEASE
  > org.springframework.cloud:spring-cloud-starter-netflix-hystrix:2.2.1.RELEASE
  > org.springframework.cloud:spring-cloud-starter-netflix-ribbon:2.2.1.RELEASE
  > org.springframework.cloud:spring-cloud-starter-netflix-archaius:2.2.1.RELEASE
  > com.netflix.zuul:zuul-core:1.3.1
```

7.5 总结

Zuul 2.x 版本一直跳票，2019 年 5 月，Netflix 终于开源了支持异步调用模式的 Zuul 2.0 版本，真可谓千呼万唤始出来。但是 Spring Cloud 已经不再集成 Zuul 2.x 了。

Spring Cloud Gateway 是基于 Spring 生态系统之上构建的 API 网关，包括：Spring 5，Spring Boot 2 和 Project Reactor。Spring Cloud Gateway 旨在提供一种简单而有效的方法来路由到 API，并为它们提供跨领域的关注点，例如：安全性，监视/指标，限流等。由于 Spring 5.0 支持 Netty，Http2，而 Spring Boot 2.0 支持 Spring 5.0，因此 Spring Cloud Gateway 支持 Netty 和 Http2 顺理成章。

API 网关在微服务架构中的作用大概是这样的：



8. 环境准备

zuul聚合工程。

- `eureka-server 01`: 注册中心
- `eureka-server02`: 注册中心
- `product-service`: 商品服务, 提供了根据主键查询商品接口
`http://localhost:9090/product/{id}`
- `order-service`: 订单服务, 提供了根据主键查询订单接口
`http://localhost:9091/order/{id}` 且订单服务调用商品服务。

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EUREKA-SERVER	n/a (2)	(2)	UP (2) - 192.168.126.1:8762 , 192.168.126.1:8761
ORDER-SERVICE	n/a (1)	(1)	UP (1) - 192.168.126.1:9091
PRODUCT-SERVICE	n/a (1)	(1)	UP (1) - 192.168.126.1:9090

9. Nginx服务器

NGINX

之前的课程中我们已经详细的讲解过 Nginx 关于反向代理、负载均衡等功能的使用，这里不再赘述。这里主要通过 Nginx 来实现 API 网关方便大家更好的学习和理解 Zuul 的使用。

9.1 下载

官网：<http://nginx.org/en/download.html> 下载稳定版。为了方便学习，请下载 Windows 版本。

9.2 安装

解压文件后直接运行根路径下的 `nginx.exe` 文件即可。

Nginx 默认端口为 80，访问：<http://localhost:80/> 看到下图说明安装成功。

Welcome to nginx!

If you see this page, the nginx web server is successfully installed and working. Further configuration is required.

For online documentation and support please refer to nginx.org.
Commercial support is available at nginx.com.

Thank you for using nginx.

9.3 配置路由规则

进入 Nginx 的 `conf` 目录，打开 `nginx.conf` 文件，配置路由规则：

```
http {  
  
    ...  
  
    server {  
        listen      80;  
        server_name localhost;  
  
        ...  
  
        # 路由到商品服务  
        location /api-product {  
            proxy_pass http://localhost:7070/;  
        }  
  
        # 路由到订单服务  
        location /api-order {
```

```
        proxy_pass http://localhost:9090/;  
    }  
    ...  
}  
...  
}
```

9.4 访问

之前我们如果要访问服务，必须由客户端指定具体服务地址访问，现在统一访问 Nginx，由 Nginx 实现网关功能将请求路由至具体的服务。

访问：<http://localhost/api-product/product/1> 结果如下：



访问：<http://localhost/api-order/order/1> 结果如下：

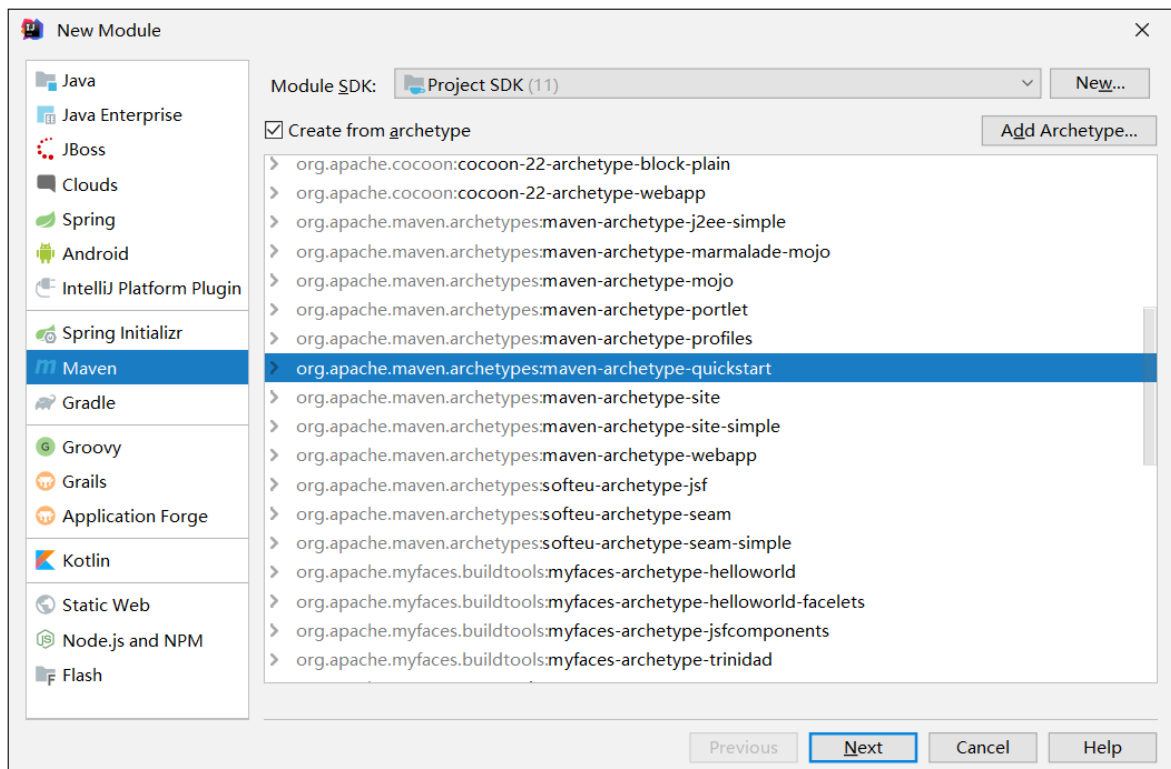
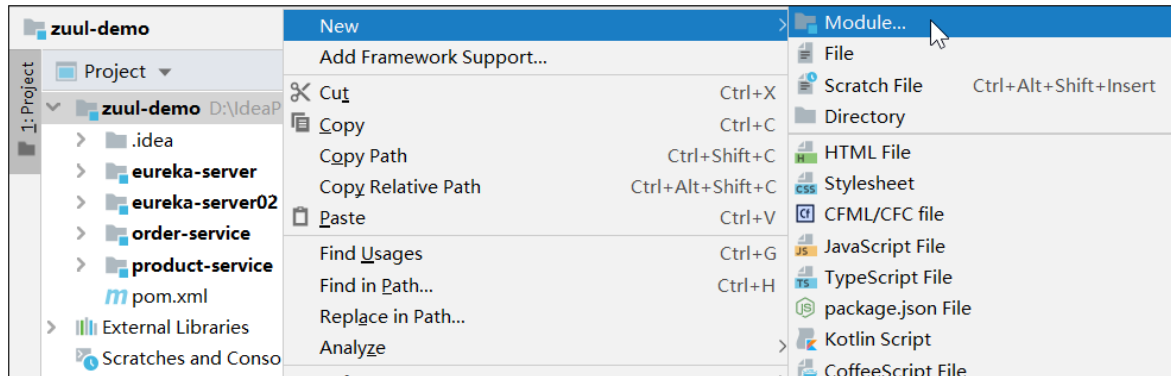


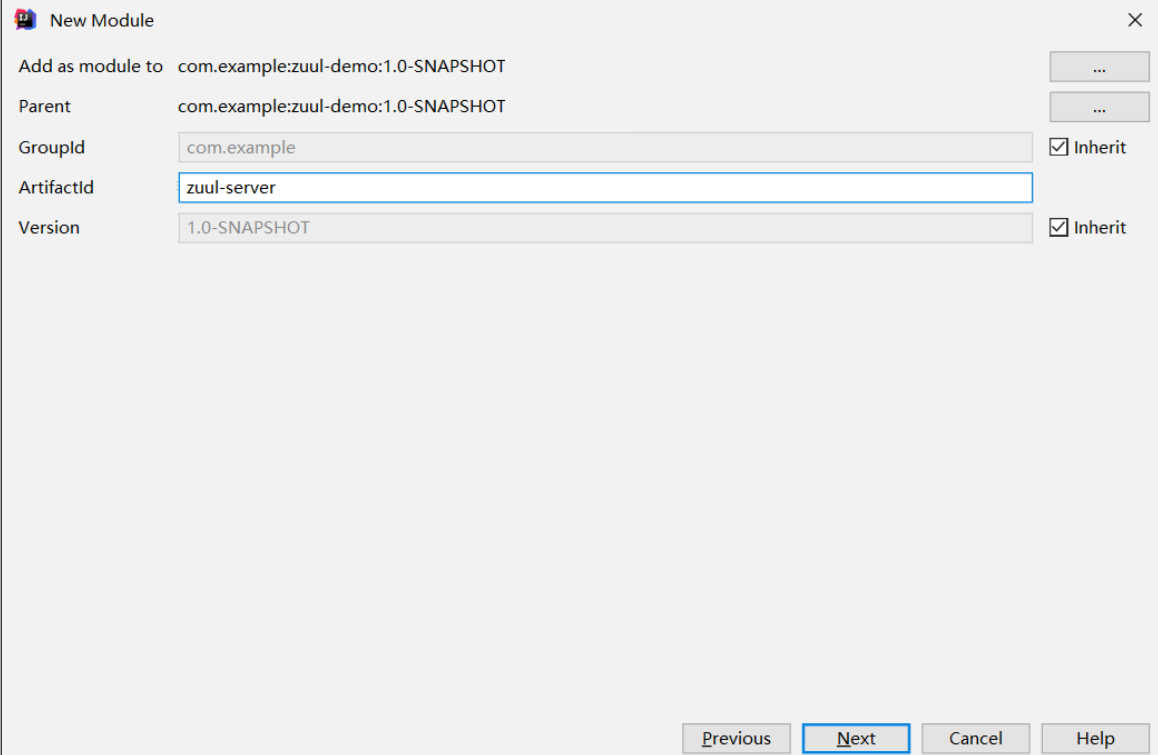
10. Zuul实现API网关

官网文档: <https://cloud.spring.io/spring-cloud-static/spring-cloud-netflix/2.2.1.RELEASE/reference/html/#router-and-filter-zuul>

10.1 搭建网关服务

创建 `zuul-server` 项目。



A screenshot of the 'New Module' dialog box in an IDE. The dialog has a title bar with a close button. It contains several fields: 'Add as module to' with the value 'com.example:zuul-demo:1.0-SNAPSHOT' and a browse button; 'Parent' with the same value and a browse button; 'GroupId' with the value 'com.example'; 'ArtifactId' with the value 'zuul-server'; and 'Version' with the value '1.0-SNAPSHOT'. There are two 'Inherit' checkboxes, both of which are checked. At the bottom, there are four buttons: 'Previous', 'Next' (which is highlighted with a blue border), 'Cancel', and 'Help'.

New Module

Add as module to: com.example:zuul-demo:1.0-SNAPSHOT

Parent: com.example:zuul-demo:1.0-SNAPSHOT

GroupId: com.example

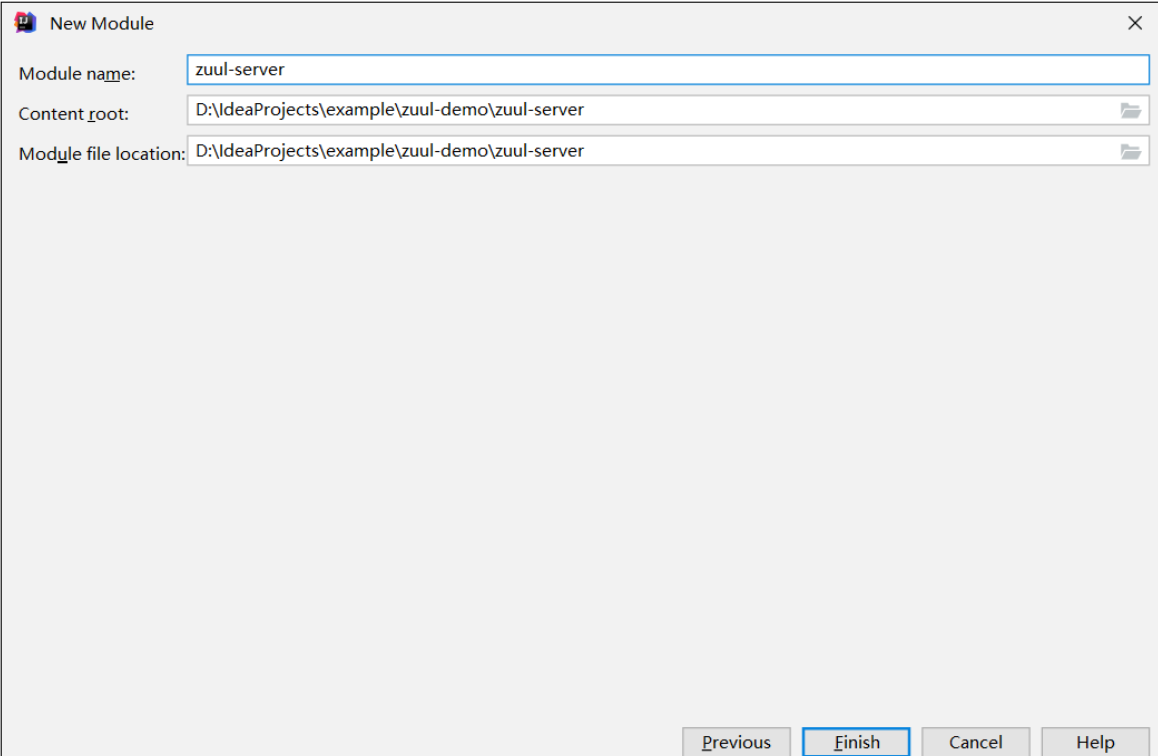
ArtifactId: zuul-server

Version: 1.0-SNAPSHOT

☒ Inherit

☒ Inherit

Previous Next Cancel Help

A screenshot of the 'New Module' dialog box in an IDE, showing the final configuration. The fields are: 'Module name' with 'zuul-server', 'Content root' with 'D:\IdeaProjects\example\zuul-demo\zuul-server', and 'Module file location' with 'D:\IdeaProjects\example\zuul-demo\zuul-server'. At the bottom, there are four buttons: 'Previous', 'Finish' (which is highlighted with a blue border), 'Cancel', and 'Help'.

New Module

Module name: zuul-server

Content root: D:\IdeaProjects\example\zuul-demo\zuul-server

Module file location: D:\IdeaProjects\example\zuul-demo\zuul-server

Previous Finish Cancel Help

10.2 添加依赖

添加 spring cloud netflix zuul 依赖。

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <parent>
```

```

    <artifactId>zuul</artifactId>
    <groupId>com.example</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>

  <artifactId>zuul-server</artifactId>
  <packaging>jar</packaging>

  <name>zuul-server</name>
  <url>http://maven.apache.org</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  </properties>

  <dependencies>
    <!--netflix eureka client 依赖-->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
      <version>3.0.2</version>
    </dependency>

    <!-- spring cloud netflix zuul 依赖 -->
    <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-netflix-zuul</artifactId>
    </dependency>
  </dependencies>
</project>

```

10.3 配置文件

application.yml

```

server:
  port: 9000 # 端口

spring:
  application:
    name: zuul-server # 应用名称

```

10.4 启动类

```

package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

```

```
import org.springframework.cloud.netflix.zuul.EnableZuulServer;

@SpringBootApplication
@EnableZuulProxy
public class ZuulServerApplication {

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

}
```

10.5 配置路由规则

10.5.1 URL地址路由

```
# 路由规则
zuul:
  routes:
    product-service:      # 路由 id 自定义
      path: /product-service/** # 配置请求 url 的映射路径
      url: http://localhost:9090/ # 映射路径对应的微服务地址
```

通配符含义：

通配符	含义	举例	解释
?	匹配任意单个字符	/product-service/?	/product-service/a, /product-service/b, ...
*	匹配任意数量字符不包括子路径	/product-service/*	/product-service/aa, /product-service/bbb, ...
**	匹配任意数量字符包括所有下级路径	/product-service/**	/product-service/aa, /product-service/aaa/b/cc

访问：<http://localhost:9000/product-service/product/1> 结果如下：



10.5.2 服务名称路由

微服务一般是由几十、上百个服务组成，对于 URL 地址路由的方式，如果对每个服务实例手动指定一个唯一访问地址，这样做显然是不合理的。

Zuul 支持与 Eureka 整合开发，根据 serviceId 自动从注册中心获取服务地址并转发请求，这样做的好处不仅可以通过单个端点来访问应用的所有服务，而且在添加或移除服务实例时不用修改 Zuul 的路由配置。

添加 Eureka Client 依赖

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

配置路由规则和注册中心

```
# 路由规则
zuul:
  routes:
    product-service:          # 路由 id 自定义
      path: /product-service/** # 配置请求 url 的映射路径
      serviceId: product-service # 根据 serviceId 自动从注册中心获取服务地址并转发请求

# 配置Eureka Server注册中心
eureka:
  instance:
    # 主机名，不配置的时候将根据操作系统的主机名称来获取
    hostname: localhost
    # 是否开启IP地址注册
    prefer-ip-address: true
    # 主机地址+端口号
    instance-id: ${spring.cloud.client.ip-address}:${server.port}
  client:
    serviceUrl:
      # 注册中心对外暴露的注册地址
      defaultZone:
http://root:123456@localhost:8761/eureka/,http://root:123456@localhost:8762/eureka/
    register-with-eureka: true
    fetch-registry: true
```

启动类

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.zuul.EnableZuulProxy;

@SpringBootApplication
// 开启 Zuul 注解
@EnableZuulProxy
// 开启 EurekaClient 注解, 目前版本如果配置了 Eureka 注册中心, 默认会开启该注解
//@EnableEurekaClient
public class ZuulServerApplication {

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

}
```

访问

访问: <http://localhost:9000/product-service/product/1> 结果如下:



10.5.3 简化路由配置

Zuul 为了方便大家使用, 提供了默认路由配置: 路由 id 和 微服务名称 一致, path 默认对应 /微服务名称/**, 所以以下配置就没必要再写了。

```
# 路由规则
zuul:
  routes:
    product-service:          # 路由 id 自定义
      path: /product-service/** # 配置请求 url 的映射路径
      serviceId: product-service # 根据 serviceId 自动从注册中心获取服务地址并转发请求
```

访问

此时我们并没有配置任何订单服务的路由规则，访问：<http://localhost:9000/order-service/order/1/product> 结果如下：



10.6 路由排除

10.6.1 URL地址路由

我们可以通过路由排除设置不允许被访问的资源。允许被访问的资源可以通过路由规则进行设置。

```
# 路由规则
zuul:
  ignored-patterns: /**/order/** # URL 地址排除，排除所有包含 /order/ 的路径
  # 不受路由排除影响
  routes:
    product-service: # 路由 id 自定义
      path: /product-service/** # 配置请求 url 的映射路径
      serviceId: product-service # 根据 serviceId 自动从注册中心获取服务地址并转发请求
```

10.6.2 服务名称排除

```
# 路由规则
zuul:
  ignored-services: order-service # 服务名称排除，多个服务逗号分隔，'*' 排除所有
  # 不受路由排除影响
  routes:
    product-service: # 路由 id 自定义
      path: /product-service/** # 配置请求 url 的映射路径
      serviceId: product-service # 根据 serviceId 自动从注册中心获取服务地址并转发请求
```

10.6.3 路由前缀

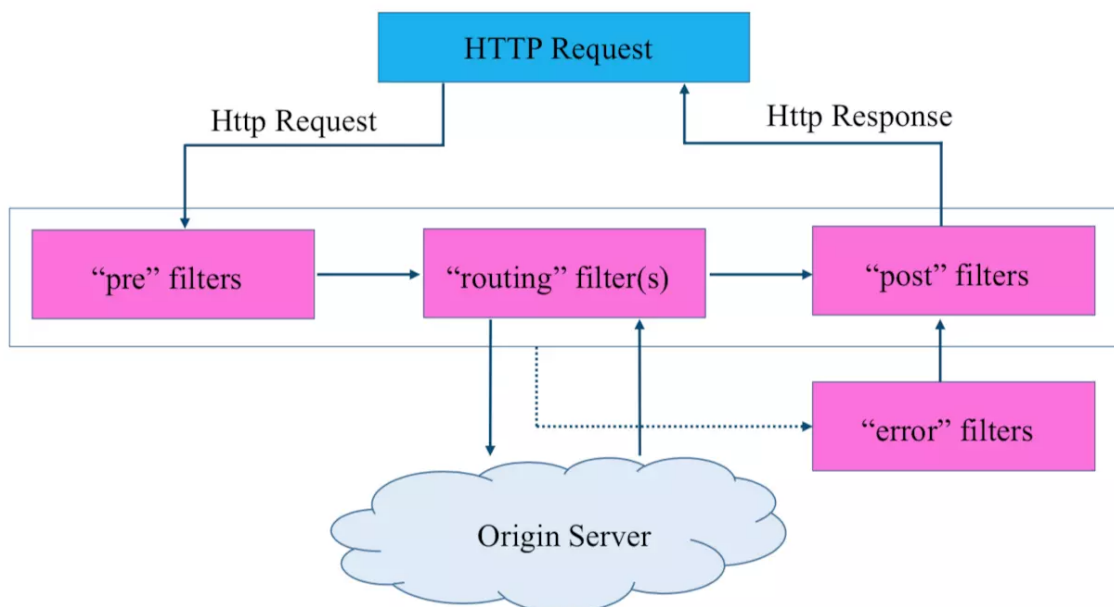
```
zuul:  
  prefix: /api
```

访问

访问: <http://localhost:9000/api/product-service/product/1> 结果如下:



11. 网关过滤器



Zuul 包含了对请求的路由和过滤两个核心功能，其中路由功能负责将外部请求转发到具体的微服务实例上，是实现外部访问统一入口的基础；而过滤器功能则负责对请求的处理过程进行干预，是实现请求校验，服务聚合等功能的基础。然而实际上，路由功能在真正运行时，它的路由映射和请求转发都是由几个不同的过滤器完成的。

路由映射主要通过 `pre` 类型的过滤器完成，它将请求路径与配置的路由规则进行匹配，以找到需要转发的目标地址；而请求转发的部分则是由 `routing` 类型的过滤器来完成，对 `pre` 类型过滤器获得的路由地址进行转发。所以说，过滤器可以说是 Zuul 实现 API 网关功能最核心的部件，每一个进入 Zuul 的 http 请求都会经过一系列的过滤器处理链得到请求响应并返回给客户端。

11.1 关键词

- **类型**：定义路由流程中应用过滤器的阶段。共 `pre`、`routing`、`post`、`error` 4 个类型。
- **执行顺序**：在**同类型**中，定义过滤器执行的顺序。比如多个 `pre` 类型的执行顺序。
- **条件**：执行过滤器所需的条件。`true` 开启，`false` 关闭。
- **动作**：如果符合条件，将执行的动作。具体操作。

11.2 过滤器类型

- `pre`：请求被路由到源服务器之前执行的过滤器
 - 身份认证
 - 选路由
 - 请求日志
- `routing`：处理将请求发送到源服务器的过滤器
- `post`：响应从源服务器返回时执行的过滤器
 - 对响应增加 HTTP 头
 - 收集统计和度量指标
 - 将响应以流的方式发送回客户端
- `error`：上述阶段中出现错误时执行的过滤器

11.3 入门案例

创建过滤器

Spring Cloud Netflix Zuul 中实现过滤器必须包含 4 个基本特征：过滤器类型，执行顺序，执行条件，动作（具体操作）。这些步骤都是 `ZuulFilter` 接口中定义的 4 个抽象方法：

```
package com.example.filter;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import javax.servlet.http.HttpServletRequest;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

/**
 * 功能描述：网关过滤器
 *
 * @author lizongzai
 * @date 2023/03/04 15:04
 */
```

```

    * @since 1.0.0
    */
@Component
public class CustomZuulFilter extends ZuulFilter {

    private static final Logger logger =
        LoggerFactory.getLogger(CustomZuulFilter.class);

    /**
     * 过滤器类型 pre routing post error
     *
     * @return
     */
    @Override
    public String filterType() {
        return "pre";
    }

    /**
     * 执行顺序 数值越小，优先级越高
     *
     * @return
     */
    @Override
    public int filterOrder() {
        return 0;
    }

    /**
     * 执行条件 true 开启 false 关闭
     *
     * @return
     */
    @Override
    public boolean shouldFilter() {
        return true;
    }

    /**
     * 动作（具体操作） 具体逻辑
     *
     * @return
     * @throws ZuulException
     */
    @Override
    public Object run() throws ZuulException {
        // 获取请求上下文
        RequestContext currentContext = RequestContext.getCurrentContext();
        HttpServletRequest request = currentContext.getRequest();
        logger.info("CustomZuulFilter--> method={}, url={}", request.getMethod(),
            request.getRequestURL().toString());
        return null;
    }
}

```

- `filterType`

: 该函数需要返回一个字符串代表过滤器的类型，而这个类型就是在 http 请求过程中定义各个阶段。在 Zuul 中默认定义了 4 个不同的生命周期过程类型，具体如下：

- `pre`: 请求被路由之前调用
- `routing`: 路由请求时被调用
- `post`: **routing 和 error 过滤器之后被调用**
- `error`: 处理请求时发生错误时被调用
- `filterOrder`: 通过 int 值来定义过滤器的执行顺序，数值越小优先级越高。
- `shouldFilter`: 返回一个 boolean 值来判断该过滤器是否要执行。
- `run`: 过滤器的具体逻辑。在该函数中，我们可以实现自定义的过滤逻辑，来确定是否要拦截当前的请求，不对其进行后续路由，或是在请求路由返回结果之后，对处理结果做一些加工等。

访问

访问: <http://localhost:9000/api/order-service/order/1/product/list> 控制台输出如下:

```
CustomZuulFilter--> method=GET, url=http://localhost:9000/api/order-service/order/1/product/list
```

11.4 统一鉴权

接下来我们在网关过滤器中通过 token 判断用户是否登录，完成一个统一鉴权案例。

创建过滤器

```
package com.example.filter;

import com.netflix.zuul.ZuulFilter;
import com.netflix.zuul.context.RequestContext;
import com.netflix.zuul.exception.ZuulException;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpStatus;
import org.springframework.stereotype.Component;

import javax.servlet.http.HttpServletRequest;
import java.io.IOException;
import java.io.PrintWriter;

/**
 * 权限验证过滤器
 */
@Component
public class AccessFilter extends ZuulFilter {

    private static final Logger logger = LoggerFactory.getLogger(AccessFilter.class);

    @Override
    public String filterType() {
        return "pre";
    }
}
```



```

@Override
public int filterOrder() {
    return 1;
}

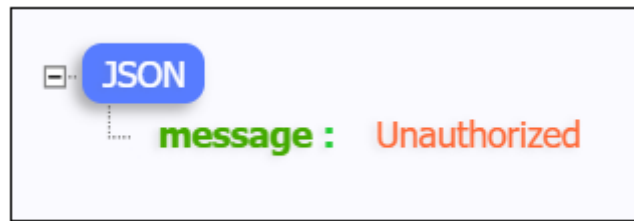
@Override
public boolean shouldFilter() {
    return true;
}

@Override
public Object run() throws ZuulException {
    // 获取请求上下文
    RequestContext rc = RequestContext.getCurrentContext();
    HttpServletRequest request = rc.getRequest();
    // 获取表单中的 token
    String token = request.getParameter("token");
    // 业务逻辑处理
    if (null == token) {
        logger.warn("token is null...");
        // 请求结束，不在继续向下请求。
        rc.setSendZuulResponse(false);
        // 响应状态码，HTTP 401 错误代表用户没有访问权限
        rc.setResponseStatusCode(HttpStatus.UNAUTHORIZED.value());
        // 响应类型
        rc.getResponse().setContentType("application/json; charset=utf-8");
        PrintWriter writer = null;
        try {
            writer = rc.getResponse().getWriter();
            // 响应内容
            writer.print("{\"message\":\"" +
HttpStatus.UNAUTHORIZED.getReasonPhrase() + "\"}");
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            if (null != writer)
                writer.close();
        }
    } else {
        // 使用 token 进行身份验证
        logger.info("token is OK!");
    }
    return null;
}
}

```

访问

访问: <http://localhost:9000/product-service/product/1> 结果如下:



访问: <http://localhost:9000/product-service/product/1?token=abc123> 结果如下:

