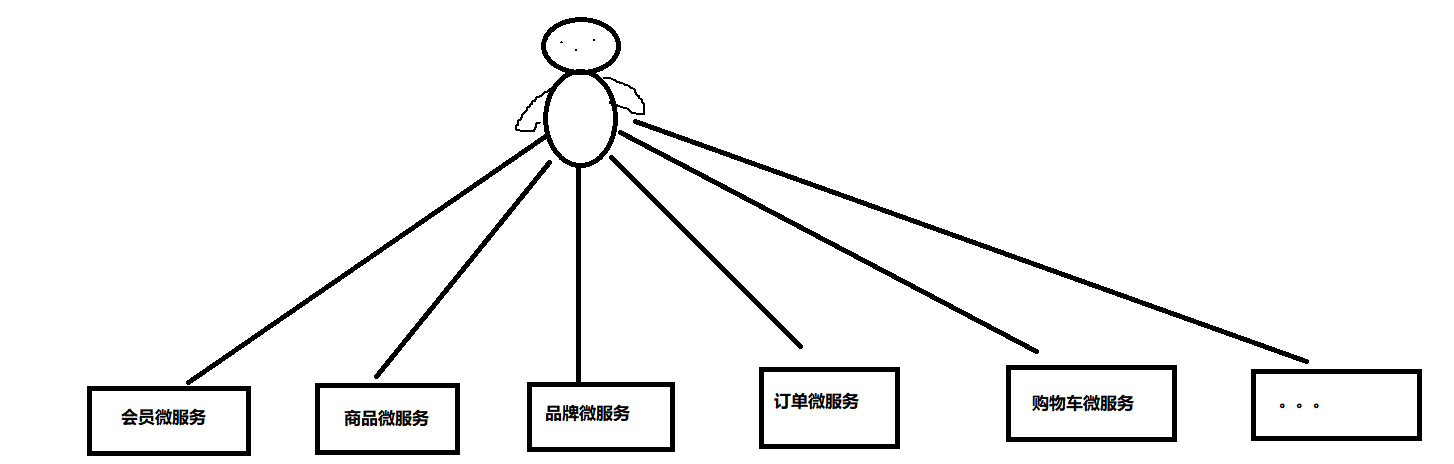
# 前言

**为什么微服务架构需要服务网关**

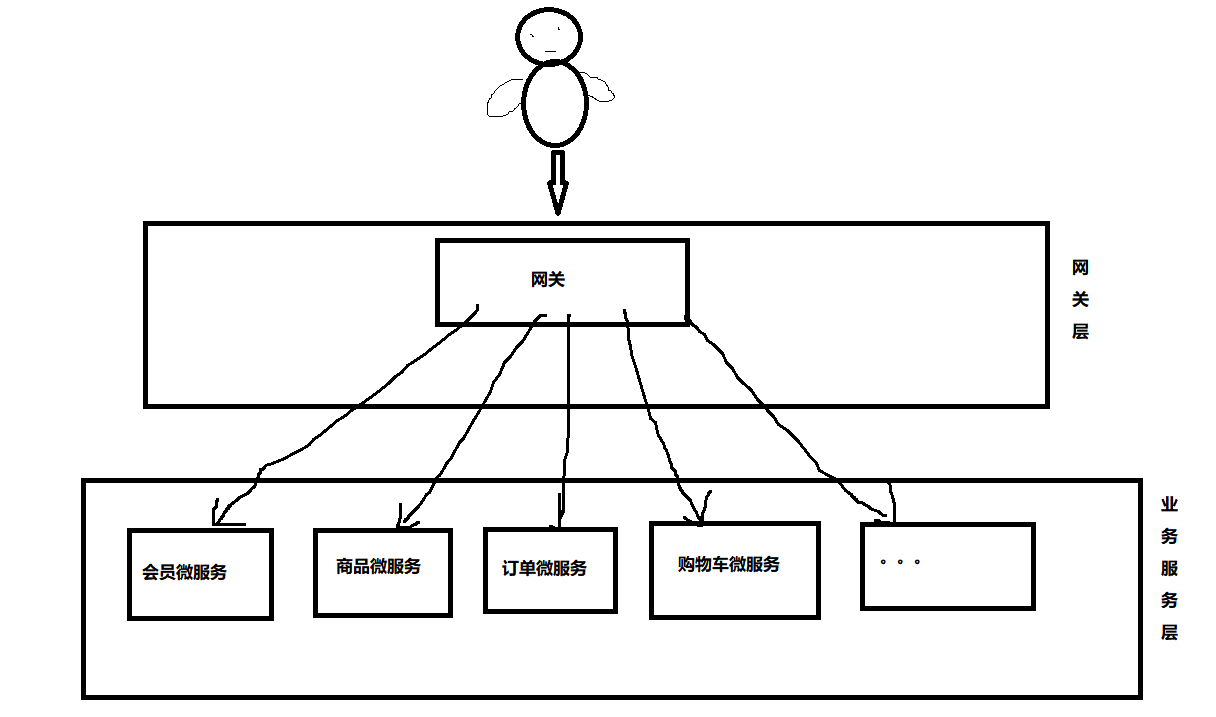
在学习完前面的知识后，微服务架构已经初具雏形。服务除了内部相互之间调用和通信之外，最终要以某种方式暴露出去，才能让外界系统（例如客户的浏览器、移动设备等等）访问到。但是这时候还有一些问题：不同的微服务一般会有不同的网络地址，那么这里就会遇到一个问题，APP/Browser怎么去访问这些后端的服务?客户端在访问这些微服务时必须记住几十甚至几百个地址，这对于客户端方来说太复杂也难以维护。如下图：



如果让客户端直接与各个微服务通讯，可能会有很多问题，比如如下问题

|  |
| --- |
| * 客户端会请求多个不同的服务，需要维护不同的请求地址，增加开发难度 * 在某些场景下存在跨域请求的问题 * 每个微服务都会需要鉴权、限流、权限校验等逻辑，如果每个业务都各自为战，自己造轮子实现一遍，会很头疼，完全可以抽出来，放到一个统一的地方去做。 * 后端每个微服务可能是由不同语言编写的、采用了不同的协议，比如HTTP、Dubbo、GRPC等，但是你不可能要求客户端去适配这么多种协议，这是一项非常有挑战的工作，项目会变的非常复杂且很难维护。 |

因此，**我们需要一个微服务网关**，**介于客户端与服务器之间的中间层**，所有的外部请求都会先经过微服务网关。



客户端只需要与网关交互，只知道一个网关地址即可，这样简化了客户端开发，当然还有更多优点，咱们下面来分析一下。

|  |
| --- |
| 可以看到，网关层是最外层，客户端与服务器交互时经过的第一个服务节点，它主要起屏蔽下游业务服务的作用，对于客户端而言，只需要跟网关交互就相当于在与下游多个业务服务节点交互，让客户端觉得他在和一台服务器交互。  这样的好处显而易见，不管是下游业务服务、支撑服务、基础服务，都对于客户端屏蔽，与服务器的交互变的非常简单，客户端无需关心各个服务的依赖关系、如何协同工作，客户端只会了解到本次请求是否成功；开发者可以灵活的增加业务服务模块；可以在网关层做一些最上层的公用的操作，如**过滤恶意请求**、设置**ip黑白名单**、做**身份认证**、**限流**、**负载均衡**等。  换个角度考虑一下，如果去掉网关层，客户端交互的最外层服务是业务服务层，由于需要解决单点登陆问题，必须在**每个业务服务节点上去写一套身份认证的逻辑**，从开发的角度上看，明显增加了复杂度，试问本可以只需要在一个网关服务上构建身份认证的逻辑，为何要选择在多个（并且可能还会增加）的业务服务上构建身份认证的逻辑呢？  另外，**从开发上讲可能还需要解决跨域请求的问题，特别是在前后端分离架构中**；对于同一服务多节点的负载均衡也不好实现，难道需要浏览器每次访问前都去访问一次注册中心？  通过分析发现，微服务架构中，对于再小的业务量的项目，**服务网关都是必不可少的**。 |

# 什么是服务网关

服务网关是介于我们客户端和各个服务中间的一个流量入口，因为高度集中所以他还充当服务调用的中央策略执行点，因为他的入口性，和集中性，那么我们**横切服务的关注点就很容易在一个地方实现**，而不需要在各个开发团队来实现这些关注点。

**服务网关 = 路由转发 + 过滤器**

* **路由转发**：接收一切外界(客户端)请求，转发到后端的微服务上去；
* **过滤器**：在服务网关中可以完成一系列的横切功能，例如权限校验、限流以及监控等，这些都可以通过过滤器完成（其实路由转发也是通过过滤器实现的）。

# Zuul(如欧)是什么

ZUUL是Netflix开源的微服务网关，Spring Cloud Zuul对Zuul进行了整合和增强，它可以和Eureka、Ribbon、Hystrix等组件配合使用，Zuul组件包含了对**请求的路由**和**过滤**两个最主要的功能：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。通过将Zuul和Eureka进行整合，将Zuul自身注册为Eureka服务治理下的应用，同时从Eureka中获得其他微服务的消息，也即**以后通过客户端(浏览器，APP)访问微服务都是通过Zuul跳转后获得**。

注意：Zuul服务最终还是会注册进Eureka Server

# 搭建Zuul网关服务器

1. **创建工程并导入Zuul相应依赖**

在IDEA中创建Zull网关服务器工程 shopcloud-zuul-server,并添加相应依赖，注意这里也要添加Eureka客户端的依赖，因为最终咱们的Zuul服务也要注册到Eureka Server中

|  |
| --- |
| <!--导入zuul相关依赖--> <!-- https://mvnrepository.com/artifact/org.springframework.cloud/spring-cloud-starter-netflix-zuul --> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  <version>2.2.1.RELEASE</version> </dependency>  <!--导入Eureka客户端的依赖--> <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  <version>1.4.7.RELEASE</version> </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId> </dependency> |

1. **编写启动类并添加Zuul相应注解@EnableZuulProxy**
2. **编写配置文件**

* 进行端口号配置
* 进行服务名配置
* 进行Eureka客户端配置
* 进行服务描述配置

|  |
| --- |
| **server**:  **port**: 8088  *#进行服务名配置* **spring**:  **application**:  **name**: shopcloud-gateway-zuul-server  *#进行Eureka客户端配置* **eureka**:  **client**:  **service-url**:  **defaultZone**: http://eureka-server22222:22222/eureka/  **instance**:  **instance-id**: shopcloud-gateway-zuul-server8088  **prefer-ip-address**: **true** *#进行服务描述* **info**:  **app.name**: shopcloud-gateway-zuul-server  **company.name**: feihujiaoyu  **build.artifactId**: $project.artifactId$  **build.version**: $project.version$ |

1. **进行测试**

* 不用路由调用**品牌微服务的查询热销品牌API接口**
* 启用录用调用品牌微服务的查询热销品牌API接口

# Zuul默认的路由规则

最直观的理解：“路由”是指根据请求URL，将请求分配到对应的处理程序。在微服务体系中，Zuul负责接收客户端所有的请求。根据不同的URL匹配规则，将不同的请求转发到不同的微服务处理。通过Zuul和Eureka进行整合，Zuul会根据服务名自动的从注册中心中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的时候不用修改Zuul的路由配置。**Zuul指定了默认的路由规则：默认情况下，一切服务的映射路径就是服务名本身。例如服务名为: shopcloud-provider-brand ，则默认的映射路径就是: /shopcloud-provider-brand/\*\***

# Zuul进行路由配置

如果有些情况下，不想让客户端知道我们某个微服务的真实的服务名，我们可以通过进行Zuul的路由配置来达到该目的：

|  |
| --- |
| zuul:  routes:  shopcloud-provider-brand: /abc/\*\* |

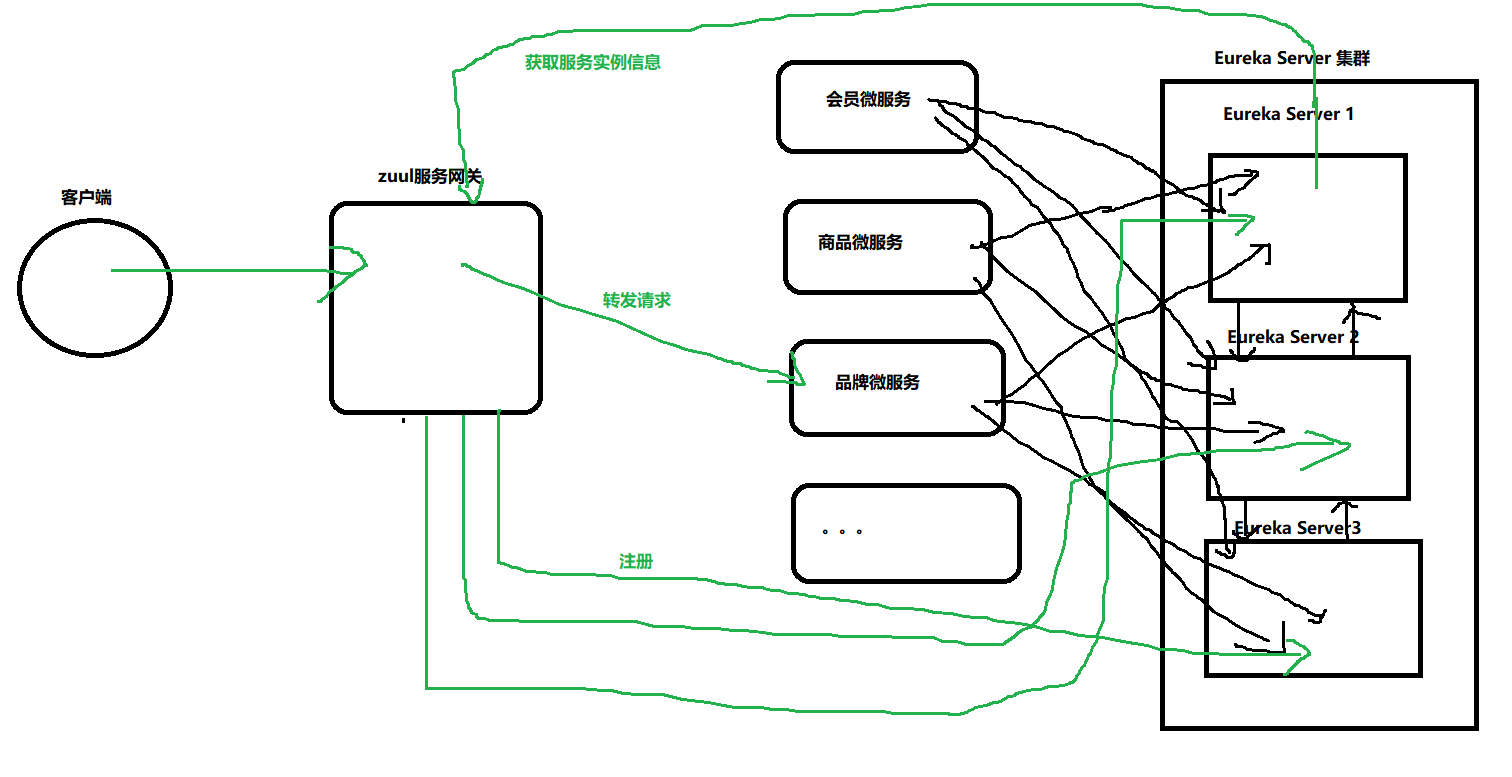
但是此时会存在一个问题，就是使用原服务名映射地址和现服务名映射地址均可成功访问到目标微服务的API接口。

# Zuul忽略真实的服务名

|  |
| --- |
| zuul:  ignored-services: shopcloud-provider-brand |

如果有多个真实的服务名需要被忽略请使用\*

# Zuul加入后的微服务系统架构



# Zuul中的过滤器

通过之前的学习，我们得知Zuul它包含了两个核心功能：对**请求的路由和过滤**。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。其实，路由功能在真正运行时，它的路由映射和请求转发同样也由几个不同的过滤器完成的。所以，**过滤器可以说是Zuul实现微服务网关功能最为核心的部件**，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。

# Zuul Filter简介

Zuul 中的过滤器跟我们之前使用的 javax.servlet.Filter 不一样，javax.servlet.Filter 只有一种类型，可以通过配置 urlPatterns 来拦截对应的请求。而 Zuul 中的过滤器总共有 4 种类型，且每种类型都有对应的使用场景。

|  |
| --- |
| * **PRE**：这种过滤器在请求被路由之前调用。我们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HTTPHeader、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提供了自定义过滤器的功能实现起来也十分简单，只需要编写一个类去实现zuul提供的接口。**

|  |
| --- |
| public abstract ZuulFilter implements IZuulFilter{  abstract public String filterType();  abstract public int filterOrder();  boolean shouldFilter();// 来自IZuulFilter  Object run() throws ZuulException;// IZuulFilter  } |

ZuulFilter是过滤器的顶级父类。在这里我们看一下其中定义的4个最重要的方法

* **shouldFilter**:返回一个 Boolean 值，判断该过滤器是否需要执行。返回true执行，返回false不执行。
* **run**:过滤器的具体业务逻辑。
* **filterType**:返回字符串，代表过滤器的类型。包含以下4种：

**pre:请求在被路由之前执行**

routing:在路由请求时调用微服务的时候执行

post:在routing和errror过滤器之后调用

error:处理请求时发生错误调用

* **filterOrder**:通过返回的int值来定义过滤器的执行顺序，数字越小优先级越高。

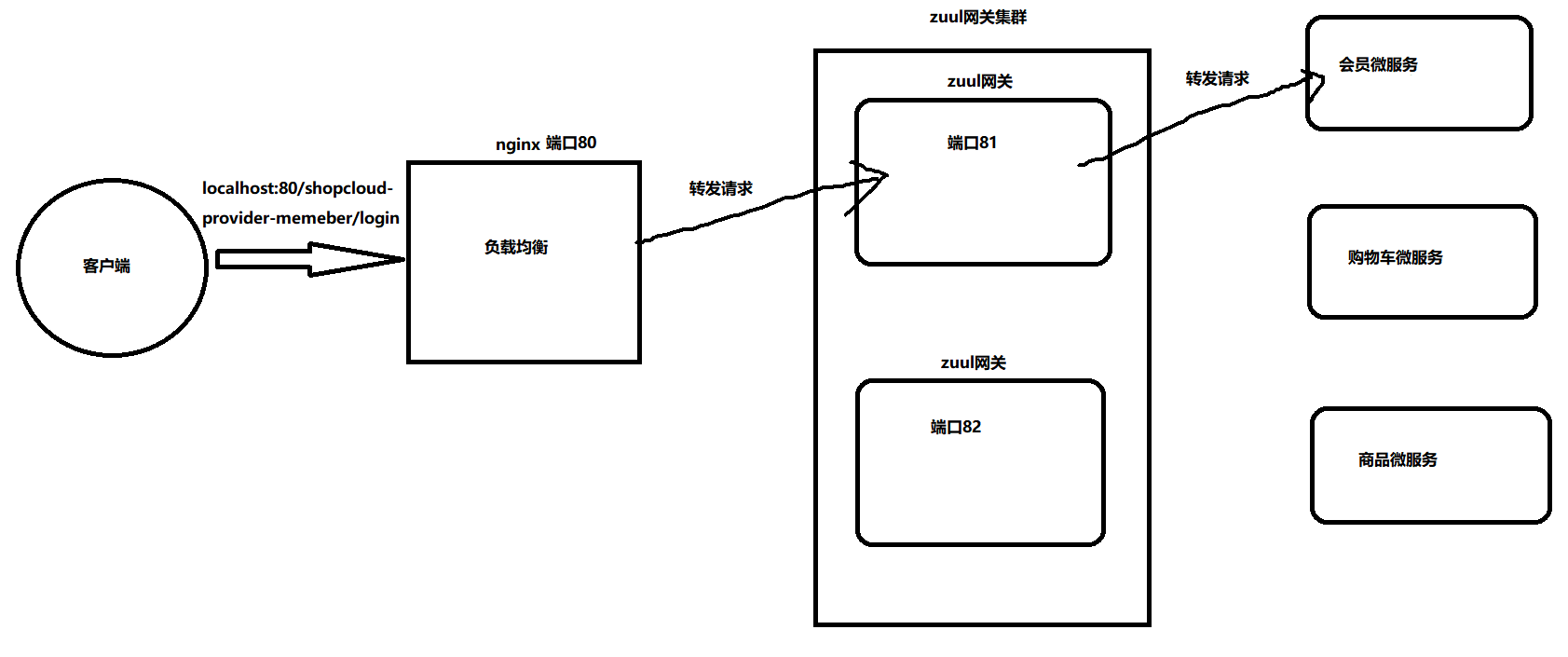
# 自定义CORS跨域过滤器

# 自定义基于TOKEN登录过滤器

# Zuul服务网关集群搭建

Zuul的高可用非常关键，因为外部请求到后端微服务的流量都会经过Zuul。故而在生产环境中，我们一般都需要部署高可用的Zuul以避免单点故障。这时候我们就需要进行 Zuul 的集群配置，这个时候可以借助额外的一些负载均衡器比如 Nginx 。

**Nginx+Zuul实现网关集群示例图**



客户端访问localhost：80时，nginx根据负载均衡算法(默认为轮询)，将请求转发到一台网关服务器(需要配置nginx的nginx.conf文件)

网关(Zuul)收到nginx转发的请求后，根据映射地址找到相对应的服务名，然后去Eureka注册中心中查询该服务名对应的所有实例信息，然后将请求转发到对应的微服务实例上(zuul已经继承了ribbon实现了负载均衡，默认为轮询)。

1. **修改Zuul网关过滤器的代码，在run方法中将当前网关的端口号打印出来**
2. **分别使用不同的端口号启动两个Zuul网关**
3. **下载安装Nginx，并在配置文件中做如下配置**



1. **双击nginx.exe启动nginx服务器**
2. **进行测试**

通过浏览器访问某个微服务的API接口，我们可以看到两个网关分别输出了日志，实现了负载均衡