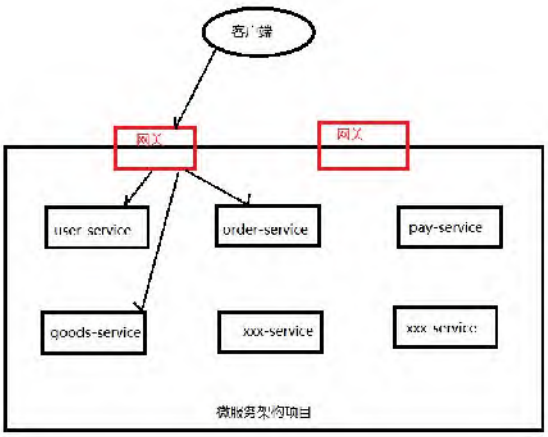
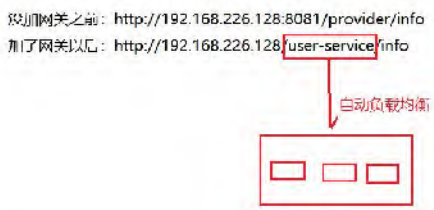
【Spring Cloud Gateway】

1.什么是网关

网关是**微服务最边缘的服务，直接暴露给用户，用来做用户和微服务的桥梁**





1.没有网关：客户端直接访问我们的微服务，会需要在客户端配置很多的 ip：port，如果 user-service 并发比较大，则无法完成负载均衡

2.有网关：客户端访问网关，网关来访问微服务，（网关可以和注册中心整合，通过服务名称找到目标的 ip：prot）这样只需要使用**服务名称即可访问微服务**，可以实现负载均衡，可以实现 token 拦截，权限验证，限流等操作

2.Spring Cloud Gateway 简介

你们项目里面 用的什么网关？ gateway zuul

它是 Spring Cloud **官方**提供的用来取代 **zuul（netflix）**的新一代网关组件

（zuul：1.0，2.0 ，zuul 的本质，**一组过滤器，根据自定义的过滤器顺序来执行，本质就是 web 组件 web 三大组件（监听器 过滤器 servlet） 拦截 springmvc**）

Zuul1.0 使用的是 BIO（Blocking IO） tomcat7.0 以前都是 BIO 性能一般

Zuul2.0 性能好 NIO

AIO 异步非阻塞 io a+nio = aio = async + no blocking io

它基于 spring5.x，springboot2.x 和 ProjectReactor 等技术。

它的目地是让**路由更加简单，灵活，还提供了一些强大的过滤器功能**，例如：熔断、限流、重试,自义定过滤器等 token 校验 ip 黑名单等

SpringCloud Gateway作为Spring Cloud生态的网关,目标是替代Zuul,在SpringCloud2.0以上的版本中，没有对新版本的 zuul2.0 以上的最新高性能版本进行集成，仍然还是使用的 zuul1.x[可以看项目依赖找到]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 模式通信框架的 Netty

NIO(非阻塞式 io) BIO 你只需要了解网关能做什么？ 网关里面写什么代码 就可以了

Spring Cloud Gateway 2.2.5

总览 学习 样品

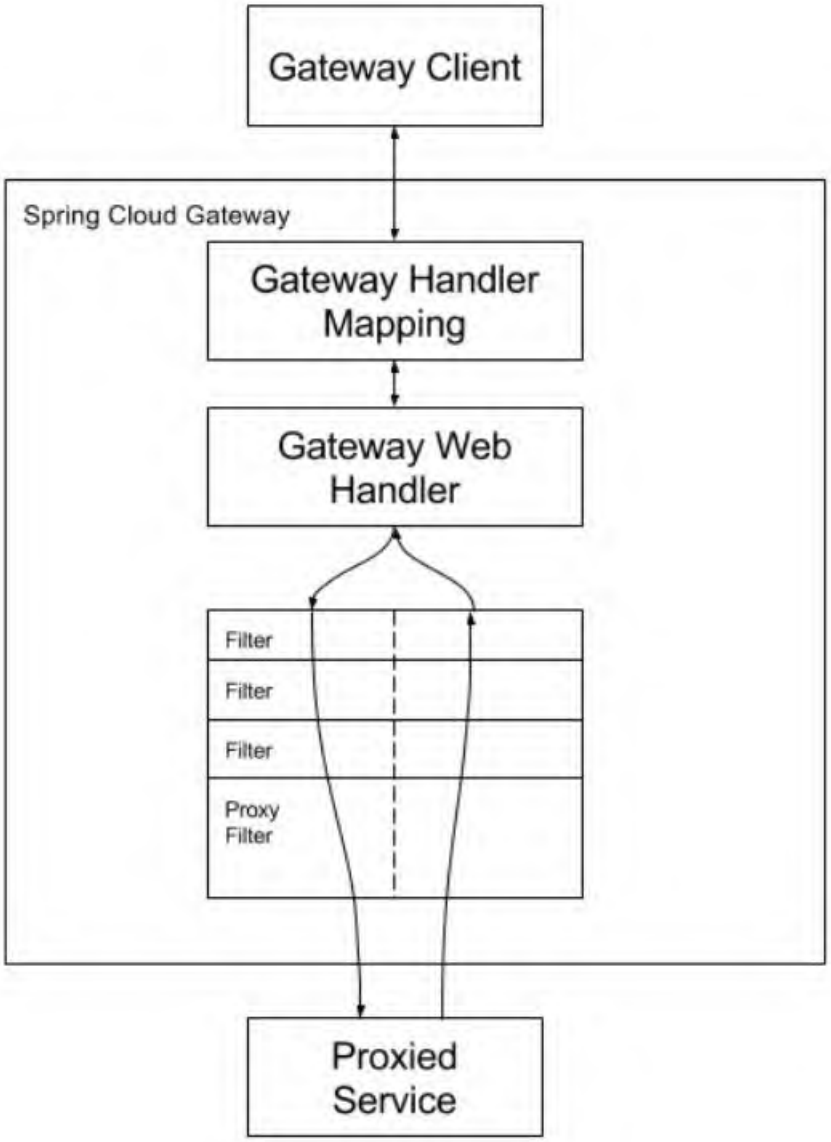
该项目提供了一个用于在Spring MVC之上构建API网关的库。Spring Cloud Gateway旨在提供一种简单而有效的方法来路由到API，并为它们提供跨领域的关注点，例如：安全性，监视/指标和弹性。

特征

Spring Cloud Gateway功能：

- 建立在Spring Framework 5，Project Reactor和Spring Boot 2.0之上
- 能够匹配任何请求属性上的路由。
- 谓词和过滤器特定于路由。
- Hystrix断路器集成。
- Spring Cloud DiscoveryClient集成
- 易于编写的谓词和过滤器
- 请求速率限制
- 路径改写

3.Spring Cloud Gateway 工作流程



客户端向 springcloud 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来将请求发送到我们实际的服务的业务逻辑，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爱丽请求之前【pre】或之后【post】执行业务逻辑，对其进行加强或处理。

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。

总结：Gateway 的核心逻辑也就是 路由转发 + 执行过滤器链

4.Spring Cloud Gateway 三大核心概念

4.1 Route(路由)（重点 和 eureka 结合做动态路由)

路由信息的组成：

由一个 ID、一个目的 URL、一组断言工厂、一组 Filter 组成。

如果路由断言为真，说明请求 URL 和配置路由匹配。

4.2 Predicate(断言）(就是一个返回 bool 的表达式)

Java 8 中的断言函数。 lambda 四大接口 供给形，消费性，函数型，断言型

Spring Cloud Gateway 中的断言函数输入类型是 Spring 5.0 框架中的 ServerWebExchange。Spring Cloud Gateway 的断言函数允许开发者去定义匹配来自于 Http Request 中的任何信息比如请求头和参数。

4.3 Filter(过滤)（重点）

一个标准的 Spring WebFilter。 Web 三大组件(servlet listener filter) mvc interceptor

Spring Cloud Gateway 中的 Filter 分为两种类型的 Filter，分别是 Gateway Filter 和 Global Filter。过滤器 Filter 将会对请求和响应进行修改处理。

一个是针对某一个路由(路径)的 filter 对某一个接口做限流

一个是针对全局的 filter token ip 黑名单

5.Nginx 和 Gateway 的区别

Nginx 在做路由，负载均衡，限流之前，都有修改 nginx.conf 的配置文件，把需要负载均衡，路由，限流的规则加在里面。Eg:使用 nginx 做 tomcat 的负载均衡

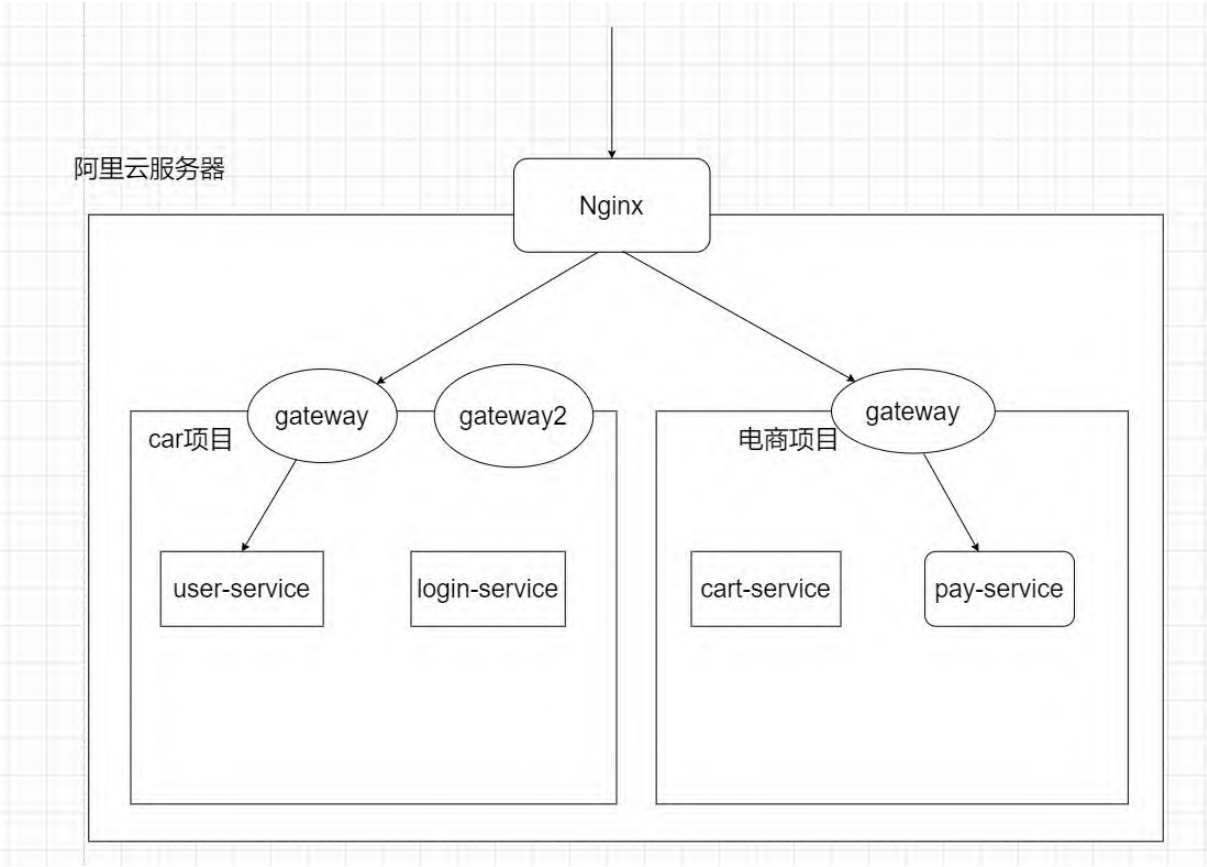
但是 gateway 不同，gateway **自动的负载均衡和路由**，gateway 和 eureka 高度集成，实现自动的路由，和 Ribbon 结合，实现了负载均衡（lb），gateway 也能轻易的实现限流和权限验证。

Nginx（c）比 gateway（java）的性能高一点。

本质的区别呢?

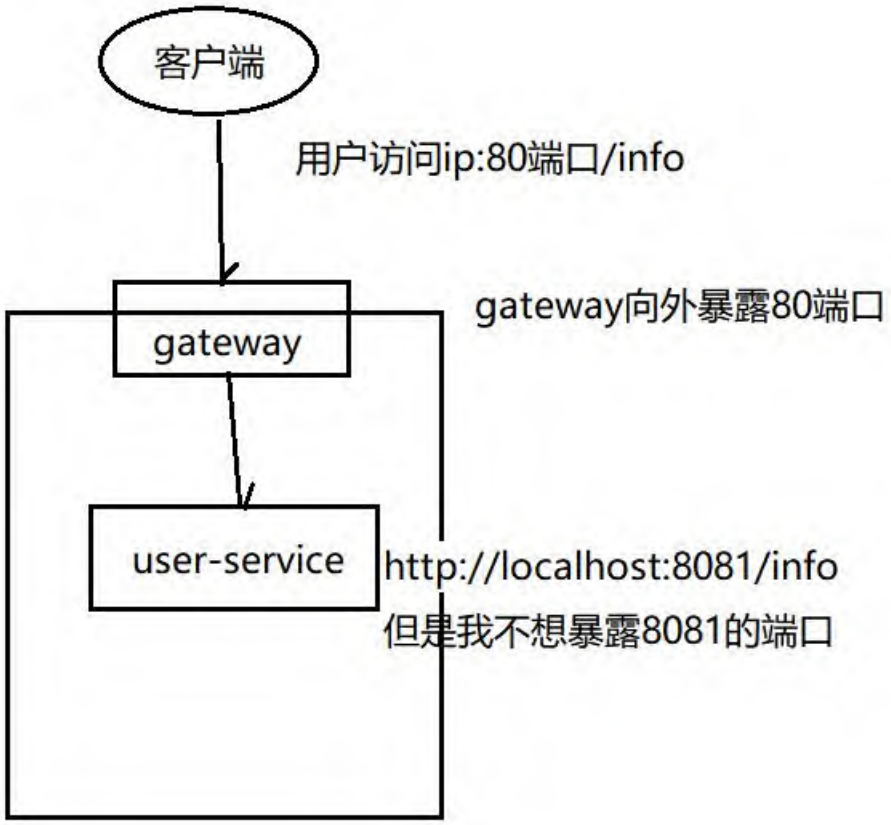
Nginx （更大 服务器级别的）

Gateway （项目级别的）

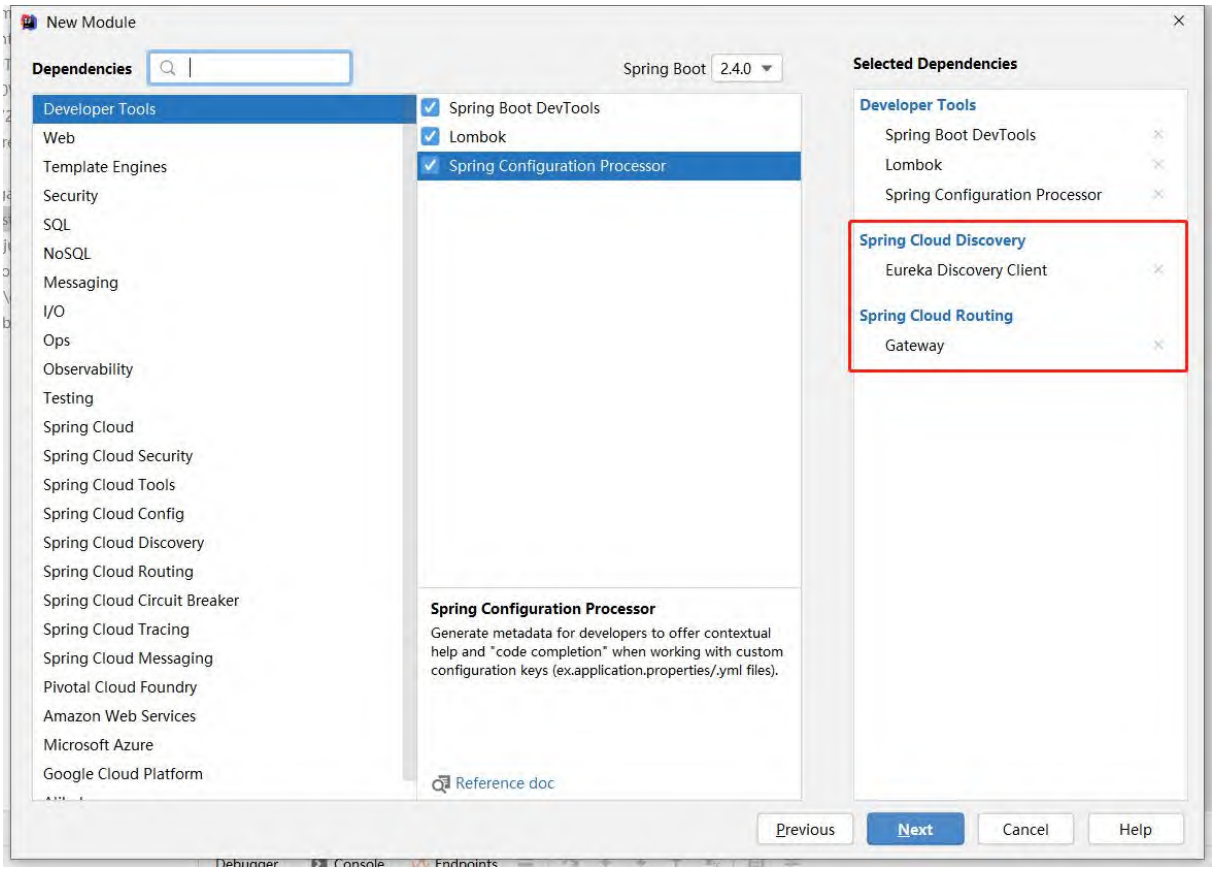

阿里云服务器
Nginx
gateway
gateway2
car项目
user-service
login-service
gateway
电商项目
cart-service
pay-service

6.Gateway 快速入门

6.1 本次访问流程



6.2 新建项目选择依赖(不要选 web)



6.3 修改启动类

```
@SpringBootApplication
@EnableEurekaClient //网关也是 eureka 的客户端
public class Gateway80Application {

    public static void main(String[] args) {
        SpringApplication.run(Gateway80Application.class, args);
    }

}
```

6.4 修改配置文件

```
server:
    port: 80
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
            enabled: true   #开启网关，默认是开启的
            routes: #设置路由，注意是数组，可以设置多个，按照 id 做隔离
                -   id: user-service-router    #路由 id，没有要求，保持唯一即可
                    uri: http://localhost:8081  #设置真正的服务 ip:port
                    predicates: #断言匹配
                        - Path=/info/** #和服务中的路径匹配,是正则匹配的模式
                -   id: provider-service-router
                    uri: http://localhost:8082
                    predicates:
                        - Path=/info/** #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡
#eureka 的配置
eureka:
    instance:
        instance-id: ${spring.application.name}:${server.port}
        prefer-ip-address: true
    client:
        service-url:
            defaultZone: http://localhost:8761/eureka/
```

6.5 启动测试

启动 eureka-server

启动 consumer-user-service

启动 gateway

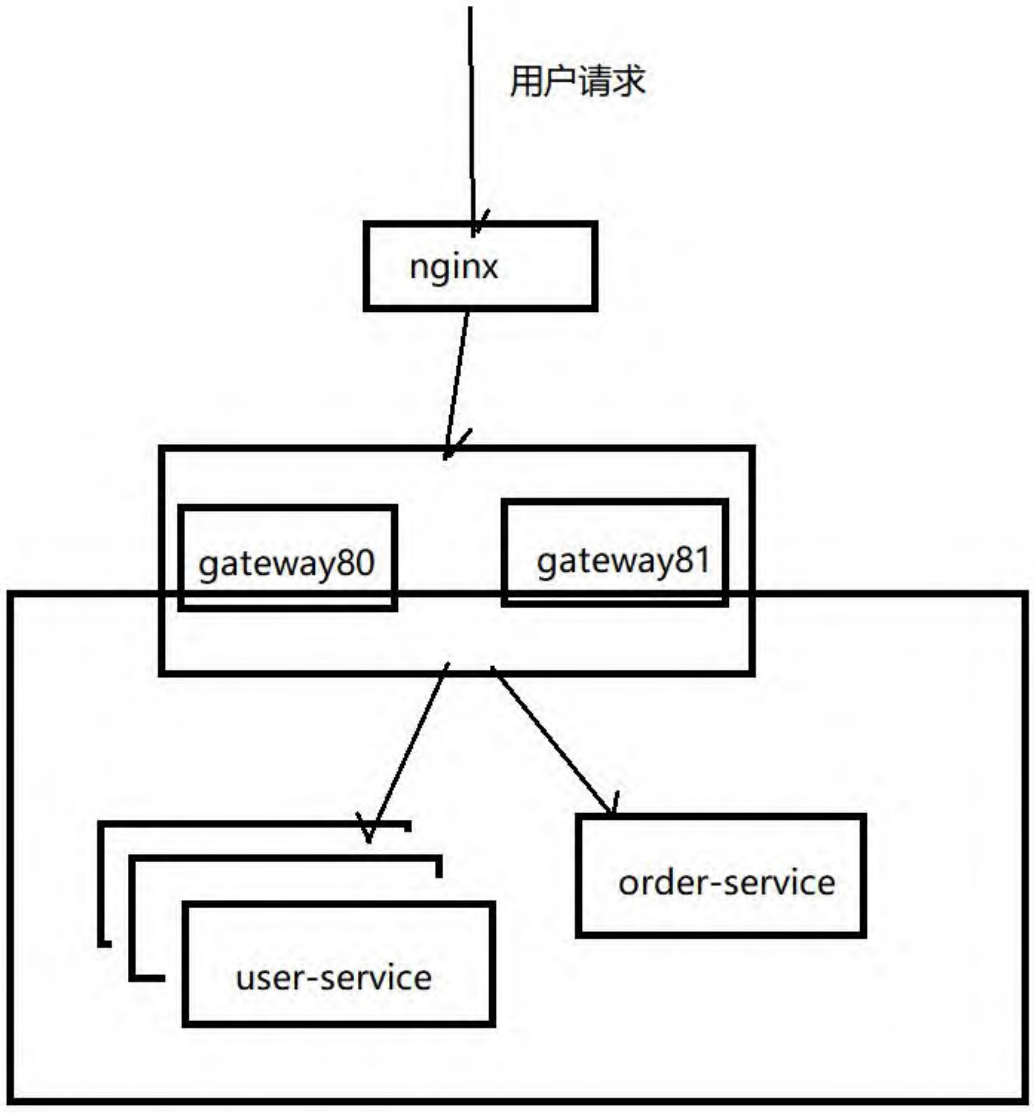
访问：http://192.168.137.1/info

user-service-info

7.Gateway 集群

基础服务设施 （eureka, gateway configserver auth-server）

这里使用虚拟机实现



7.1 创建两个 gateway，端口分别为 80 和 81

```
server:
    port: 80
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
            enabled: true   #开启网关，默认是开启的
```

```
        discovery:
            locator:
                enabled: true
                lower-case-service-id: true
        routes: #设置路由，注意是数组，可以设置多个，按照 id 做隔离
            - id: user-service-router    #路由 id，没有要求，保持唯一即可
              uri: http://192.168.226.1:8081  #设置真正的服务 ip:port
              predicates: #断言匹配
                - Path=/info/** #和服务中的路径匹配,是正则匹配的模式
            - id: provider-service-router
              uri: http://192.168.226.1:8082
              predicates:
                - Path=/info/** #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡
#eureka 的配置
eureka:
    instance:
        instance-id: ${spring.application.name}:${server.port}
        prefer-ip-address: true
    client:
        service-url:
            defaultZone: http://localhost:8761/eureka/
```

7.2 Nginx 的配置文件修改

```
upstream www.gateway.com{
        server 192.168.226.1:80;
        server 192.168.226.1:81;

}

server {
    listen       8080;
    server_name  localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
                proxy_pass http://www.gateway.com;
    }
```

7.3 访问测试

user-service-info

8.Gateway 的两种路由配置方式

8.1 代码路由方式（掌握）

官网给出的配置类，我们照葫芦画瓢

https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#modifying-the-way-remote-addresses-are-resolved

Example 12. GatewayConfig.java

```
RemoteAddressResolver resolver = XForwardedRemoteAddressResolver
    .maxTrustedIndex(1);

...

.route("direct-route",
    r -> r.remoteAddr("10.1.1.1", "10.10.1.1/24")
        .uri("https://downstream1")
.route("proxied-route",
    r -> r.remoteAddr(resolver, "10.10.1.1", "10.10.1.1/24")
        .uri("https://downstream2")
)
```

8.1.1 创建配置类 GatewayConfig

```
@Configuration
public class GatewayConfig {


    @Bean
    public RouteLocator routeLocator(RouteLocatorBuilder routeLocatorBuilder) {
        RouteLocatorBuilder.Builder routes = routeLocatorBuilder.routes();
        routes
                .route("path_rote_guonei", r -> r.path("/guonei").uri("http://news.baidu.com/guonei"))
                .route("path_rote_guoji", r -> r.path("/guoji").uri("http://news.baidu.com/guoji"))
                .route("path_rote_tech", r -> r.path("/tech").uri("http://news.baidu.com/tech"))
                .route("path_rote_lady", r -> r.path("/lady").uri("http://news.baidu.com/lady"))
                .build();
        return routes.build();
    }
```

8.1.2 启动测试

8.2 使用 yml 方式（重点）

和上面的快速入门一样，使用 yml 的方式，在开发中是常用的

```yaml
server:
    port: 80
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
            enabled: true    #开启网关，默认是开启的
            routes: #设置路由，注意是数组，可以设置多个，按照id做隔离
                -   id: user-service-router     #路由id，没有要求，保持唯一即可
                    uri: http://localhost:8081  #设置真正的服务ip:port
                    predicates: #断言匹配
                        - Path=/info/** #和服务中的路径匹配,是正则匹配的模式
                -   id: provider-service-router
                    uri: http://localhost:8082
                    predicates:
                        - Path=/info/** #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡
```

9.Gateway 微服务名动态路由，负载均衡

9.1 概述

从之前的配置里面我们可以看到我们的 URL 都是写死的，这不符合我们微服务的要求，我们微服务是只要知道服务的名字，根据名字去找，而直接写死就没有负载均衡的效果了

默认情况下 **Gateway 会根据注册中心的服务列表，以注册中心上微服务名为路径创建动态路由进行转发，从而实现动态路由的功能**

需要注意的是 uri 的协议为 lb（load Balance），表示启用 Gateway 的负载均衡功能。

lb://serviceName 是 spring cloud gateway 在微服务中自动为我们创建的负载均衡 uri

协议：就是双方约定的一个接头暗号

http：//

9.2 最佳实践

9.2.1 修改 gateway 配置

```
server:
    port: 80
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
            discovery:
                locator:
                    enabled: true   #开启动态路由
                    lower-case-service-id: true #动态路由小驼峰规则
            routes: #设置路由，注意是数组，可以设置多个，按照 id 做隔离
                -   id: user-service-router     #路由 id，没有要求，保持唯一即可
```

```
                uri: lb://provider  #使用 lb 协议 微服务名称做负均衡
                predicates: #断言匹配
                    - Path=/info/**  #和服务中的路径匹配,是正则匹配的模式
            -   id: provider-service-router
                uri: http://localhost:8082
                predicates:
                    - Path=/info/** #如果匹配到第一个路由，则第二个就不会走了，注意这不是负载均衡
#eureka 的配置
eureka:
    instance:
        instance-id: ${spring.application.name}:${server.port}
        prefer-ip-address: true
    client:
        service-url:
            defaultZone: http://localhost:8761/eureka/
```

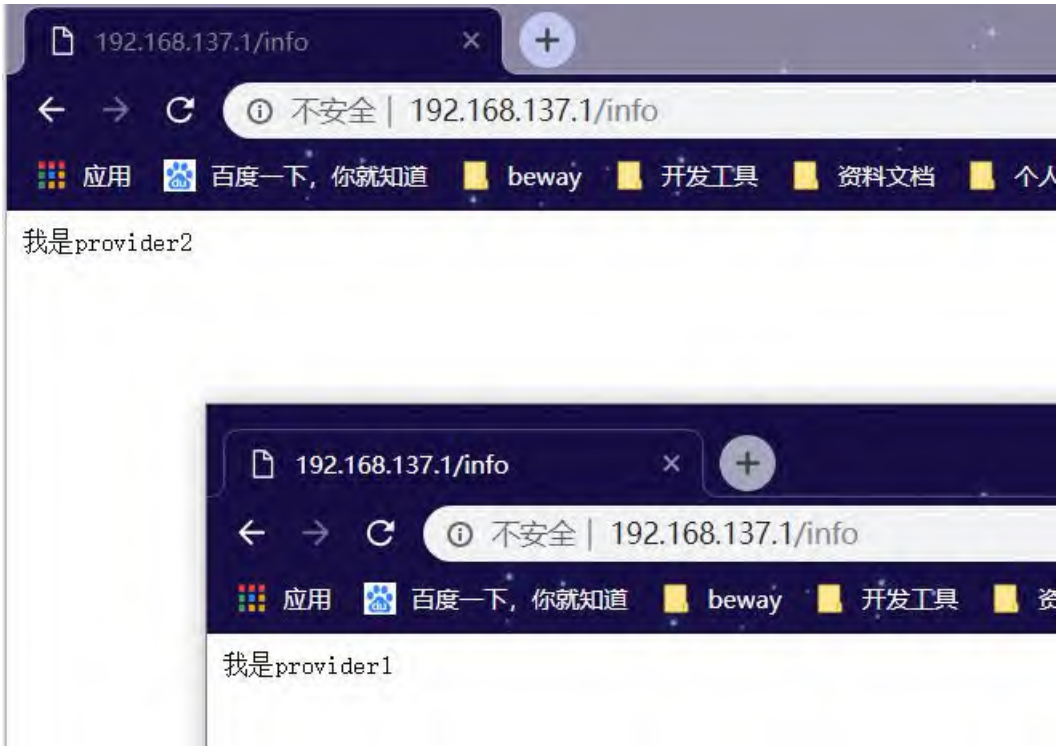
9.2.2 启动测试

启动 eureka-server

启动两个服务名为 provider 的服务，和 uri 里面 lb://服务名一致

在 provider 里面提供两个接口/info

访问测试：http://192.168.137.1/info 正常访问

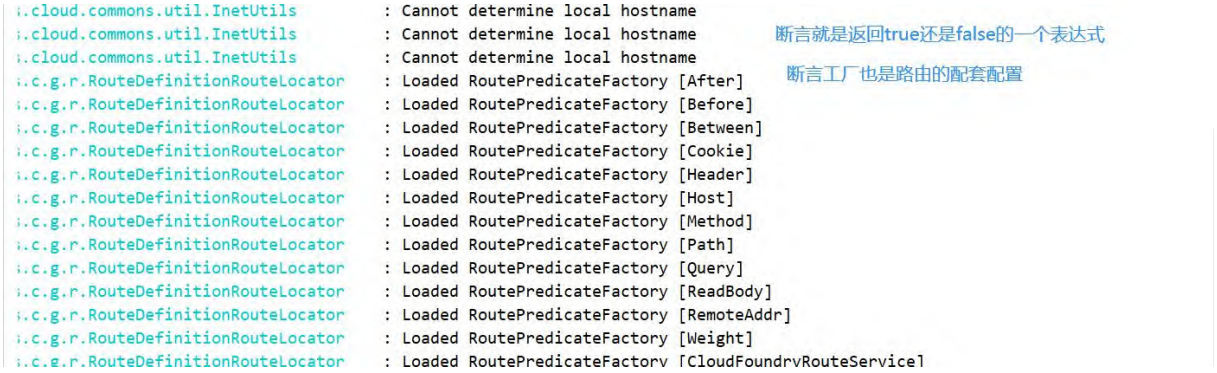


当我们新起一个服务，那么 gateway 可以实现服务发现功能，我们并没有再 routers 里面配置路由规则，然而我们访问 新起的 provider-order-service，测试访问 http://localhost/provider-order-service/info 可以成功，这就是动态路由和服务发现

10. Predicate 断言工厂的使用【了解】

在 gateway 启动时会去加载一些路由断言工厂(判断一句话是否正确 一个 boolean 表达式)

https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#gateway-request-predicates-factories



10.1 什么是断言，Gateway 里面有哪些断言

断言就是路由添加一些条件(丰富路由功能的)

通俗的说，**断言就是一些布尔表达式**，满足条件的返回 `true`，不满足的返回 `false`。

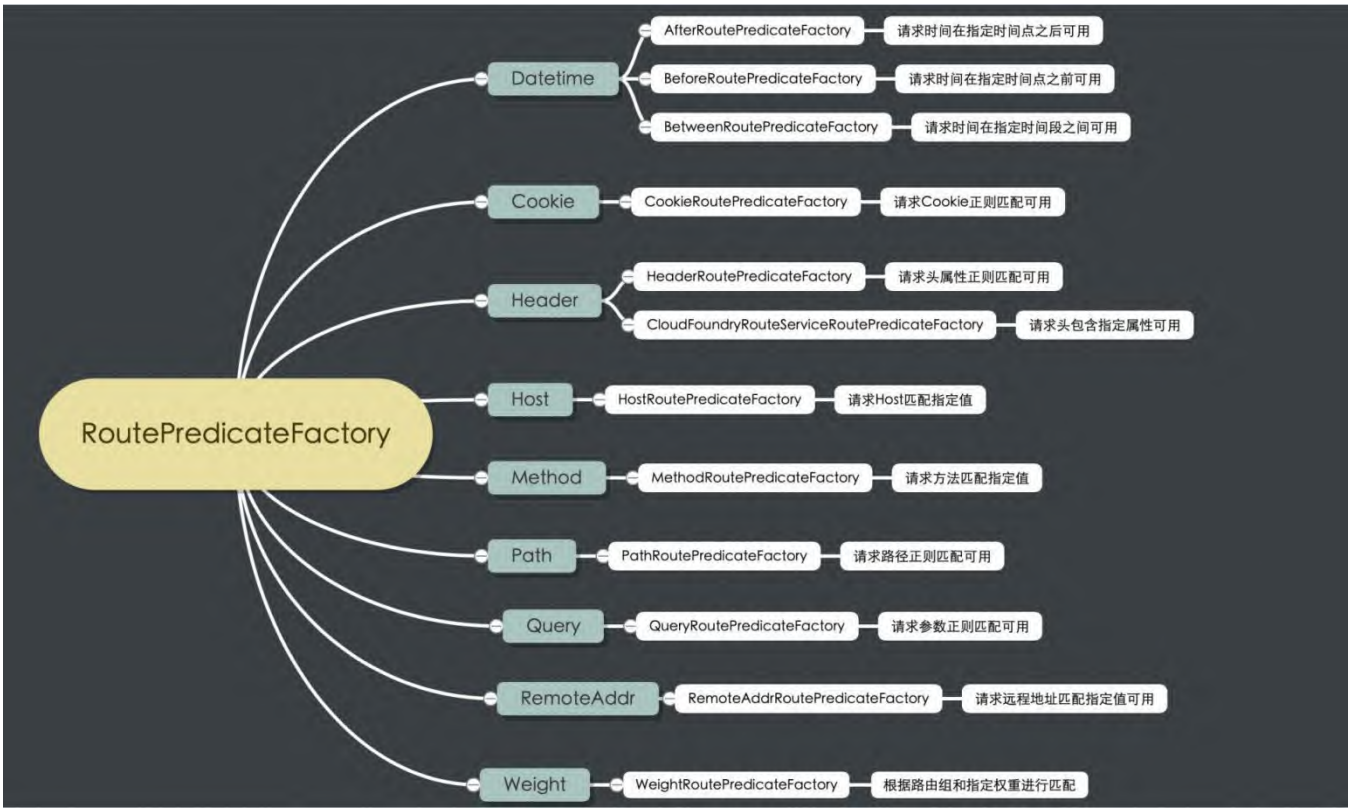
Spring Cloud Gateway 将路由作为 Spring WebFlux HandlerMapping 基础架构的一部分进行匹配。Spring Cloud Gateway 包括许多内置的路由断言工厂。所有这些断言都与 HTTP 请求的不同属性匹配。您可以将**多个路由断言可以组合**使用

Spring Cloud Gateway 创建对象时，使用 RoutePredicateFactory 创建 Predicate 对象，Predicate 对象可以赋值给 Route。

```
@FunctionalInterface
public interface RoutePredicateFactory<C> extends ShortcutConfigurable, Configurable<C> {
```

Choose Implementation of **RoutePredicateFactory** (14 found)

- AbstractRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- AfterRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- BeforeRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- BetweenRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- CloudFoundryRouteServiceRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- CookieRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- HeaderRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- HostRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- MethodRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- PathRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- QueryRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- ReadBodyPredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- RemoteAddrRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)
- WeightRoutePredicateFactory (org.springframework.cloud.gateway.handler.predicate)



10.2 如何使用这些断言

使用断言判断时，我们常用 yml 配置文件的方式进行配置

```
server:
    port: 80
```

```
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
            enabled: true    #开启网关，默认是开启的
            routes: #设置路由，注意是数组，可以设置多个，按照 id 做隔离
                -   id: user-service    #路由 id，没有要求，保持唯一即可
                    uri: lb://provider  #使用 lb 协议 微服务名称做负均衡
                    predicates: #断言匹配
                        - Path=/info/**  #和服务中的路径匹配,是正则匹配的模式
                        - After=2020-01-20T17:42:47.789-07:00[Asia/Shanghai]    #此断言匹配发生在指定日期时间之后的请求，ZonedDateTime dateTime=ZonedDateTime.now()获得
                        - Before=2020-06-18T21:26:26.711+08:00[Asia/Shanghai]    #此断言匹配发生在指定日期时间之前的请求
                        -
Between=2020-06-18T21:26:26.711+08:00[Asia/Shanghai],2020-06-18T21:32:26.711+08:00[Asia/Shanghai]
#此断言匹配发生在指定日期时间之间的请求
                        - Cookie=name,xiaobai    #Cookie 路由断言工厂接受两个参数，Cookie 名称和 regexp(一个 Java 正则表达式)。此断言匹配具有给定名称且其值与正则表达式匹配的 cookie
                        - Header=token,123456    #头路由断言工厂接受两个参数，头名称和 regexp(一个 Java 正则表达式)。此断言与具有给定名称的头匹配，该头的值与正则表达式匹配。
                        - Host=**.bai*.com:*     #主机路由断言工厂接受一个参数:主机名模式列表。该模式是一个 ant 样式的模式。作为分隔符。此断言匹配与模式匹配的主机头
                        - Method=GET,POST        #方法路由断言工厂接受一个方法参数，该参数是一个或多个参数:要匹配的 HTTP 方法
                        - Query=username,cxs         #查询路由断言工厂接受两个参数:一个必需的 param 和一个可选的 regexp(一个 Java 正则表达式)。
                        - RemoteAddr=192.168.1.1/24 #RemoteAddr 路由断言工厂接受一个源列表(最小大小 1)，这些源是 cidr 符号(IPv4 或 IPv6)字符串，比如 192.168.1.1/24(其中 192.168.1.1 是 IP 地址，24 是子网掩码)。
```

还有一个访问权重的设置，意思是说：

80%的请求，由 https://weighthigh.org 这个 url 去处理

20%的请求由 https://weightlow.org 去处理

```
spring:
```

```
    cloud:
        gateway:
            routes:
                -   id: weight_high
                    uri: https://weighthigh.org
                    predicates:
                        - Weight=group1, 8
                -   id: weight_low
                    uri: https://weightlow.org
                    predicates:
                        - Weight=group1, 2
```

10.3 断言总结

Predicate 就是为了实现一组匹配规则，让请求过来找到对应的 Route 进行处理

~~10.4 自定义断言工厂（了解）~~

其实 gateway 默认为我提供的断言工厂已经够用了，但是我们想自己定义呢

自定义路由断言工厂需要继承 AbstractRoutePredicateFactory 类，重写 apply 方法的逻辑。

在 apply 方法中可以通过 exchange.getRequest() 拿到 ServerHttpRequest 对象，从而可以获取到请求的参数、请求方式、请求头等信息。

apply 方法的参数是自定义的配置类，在使用的时候配置参数，在 apply 方法中直接获取使用。

命名需要以 RoutePredicateFactory 结尾，比如 CheckAuthRoutePredicateFactory，那么在使用的时候 CheckAuth 就是这个路由断言工厂的名称。

10.4.1 最佳实践

10.4.1.1 创建配置类

```
@Component
public class CheckAuthRoutePredicateFactory
        extends AbstractRoutePredicateFactory<CheckAuthRoutePredicateFactory.Config>
{

    public CheckAuthRoutePredicateFactory() {
        super(Config.class);
    }

    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        return exchange -> {
            System.err.println("进入了 CheckAuthRoutePredicateFactory\t" +
config.getName());
            if (config.getName().equals("xiaobai")) {
                return true;
            }
            return false;
        };
    }

    @Data
    static class Config {
        private String name;
    }
}
```

10.4.1.2 修改配置文件

```
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
```

```
        enabled: true   #开启网关，默认是开启的
        routes: #设置路由，注意是数组，可以设置多个，按照 id 做隔离
          - id: user-service    #路由 id，没有要求，保持唯一即可
            uri: lb://provider  #使用 lb 协议 微服务名称做负均衡
            predicates: #断言匹配
              - name: CheckAuth
                args:
                  name: xiaobai
```

10.4.1.3 访问测试是否进入自定义断言器

随便访问，测试一下而已

11. Filter 过滤器工厂（重点）

11.1 概述

gateway 里面的过滤器和 Servlet 里面的过滤器，功能差不多，路由过滤器可以用于修改进入 Http 请求和返回 Http 响应

11.2 分类

11.2.1 按生命周期分两种

pre 在业务逻辑之前

post 在业务逻辑之后

11.2.2 按种类分也是两种

GatewayFilter 需要配置某个路由，才能过滤。如果需要使用全局路由，需要配置 Default

Filters。

GlobalFilter　全局过滤器，不需要配置路由，系统初始化作用到所有路由上

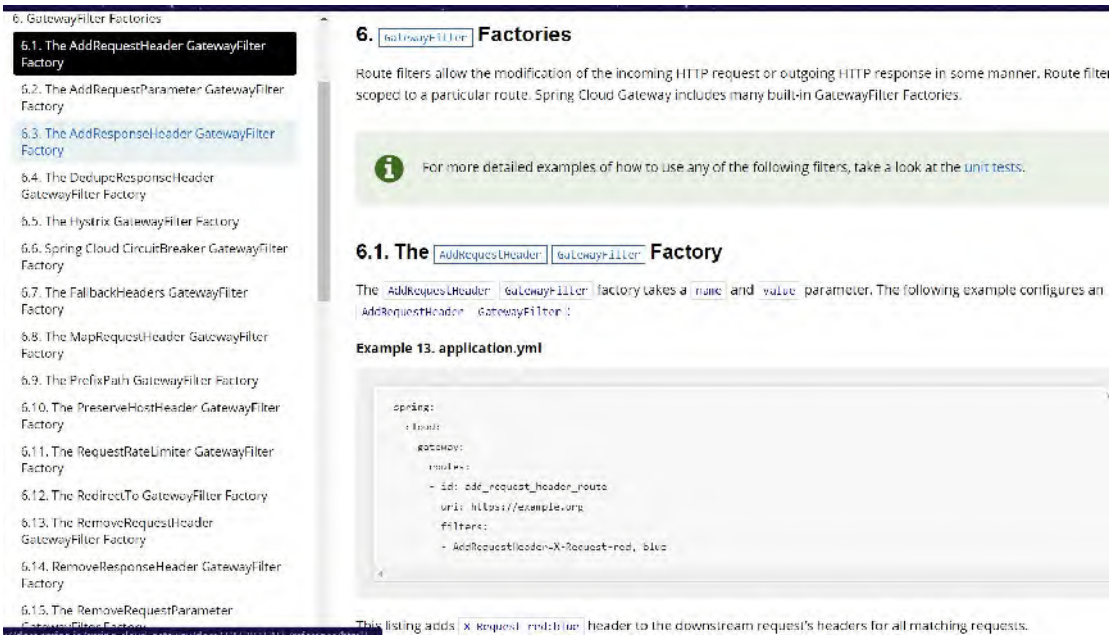
全局过滤器 统计请求次数　限流　token 的校验　ip 黑名单拦截　跨域本质(filter)

144 开头的电话 限制一些 ip 的访问

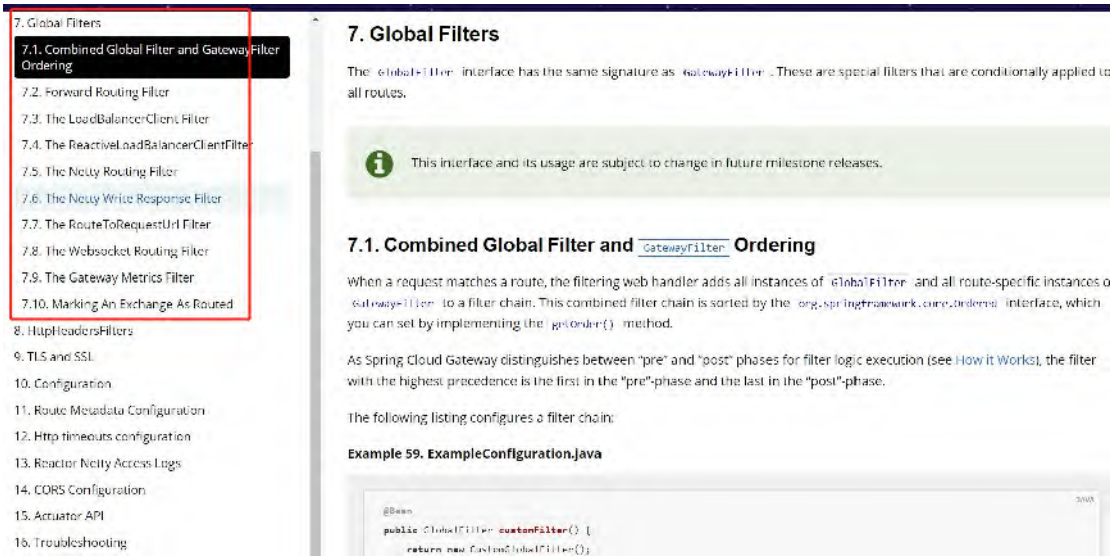
11.3 官方文档查看过滤器

11.3.1 单一过滤器（31 个）

https://docs.spring.io/spring-cloud-gateway/docs/2.2.5.RELEASE/reference/html/#gatewayfilter-factories



11.3.2 全局过滤器（9 个）



11.4 自定义网关过滤器(重点)

11.4.1 自定义全局过滤器

全局过滤器的优点的初始化时默认挂到所有路由上，我们可以使用它来完成 IP 过滤，限流等功能

11.4.2 创建配置类 GlobalFilterConfig

```
@Component
@Slf4j
public class GlobalFilterConfig implements GlobalFilter, Ordered {

    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

        log.info("进入了我自己的全局过滤器");
        String token = exchange.getRequest().getQueryParams().getFirst("token");
        if (token == null) {
            log.error("token 为空，说明没有认证");
            exchange.getResponse().setStatusCode(HttpStatus.NOT_ACCEPTABLE);
            return exchange.getResponse().setComplete();
```

```
        }
        log.info("验证通过");
        return chain.filter(exchange);
    }

    /**
     * order 越小 越先执行
     *
     * @return
     */
    @Override
    public int getOrder() {
        return 0;
    }
}
```

11.4.3 访问测试

http://localhost/info?token=asdad

12. IP 认证拦截实战

12.1 创建 IPGlobalFilter

```
@Component
@Slf4j
public class IPCheckFilter implements GlobalFilter, Ordered {
    @SneakyThrows
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        String ip = exchange.getRequest().getHeaders().getHost().getHostName();
        //这里写死了，只做演示
        if (ip.equals("localhost")) {
            //说明是黑名单里面的 ip
            ServerHttpResponse response = exchange.getResponse();
            response.setStatusCode(HttpStatus.UNAUTHORIZED);
            Map<String, Object> map = new HashMap<>();
            map.put("code", HttpStatus.UNAUTHORIZED);
```

```
            map.put("msg", "非法访问");
            response.getHeaders().add("content-Type",
"application/json;charset=UTF-8");
            ObjectMapper objectMapper = new ObjectMapper();
            byte[] bytes = objectMapper.writeValueAsBytes(map);
            DataBuffer buffer = response.bufferFactory().wrap(bytes);
            return response.writeWith(Mono.just(buffer));
        }
        return chain.filter(exchange);
    }

    /**
     * 设置此过滤器的执行顺序
     *
     * @return
     */
    @Override
    public int getOrder() {
        return 1;
    }
}
```

12.2 测试访问

13. 限流实战（会问）

13.1 什么是限流

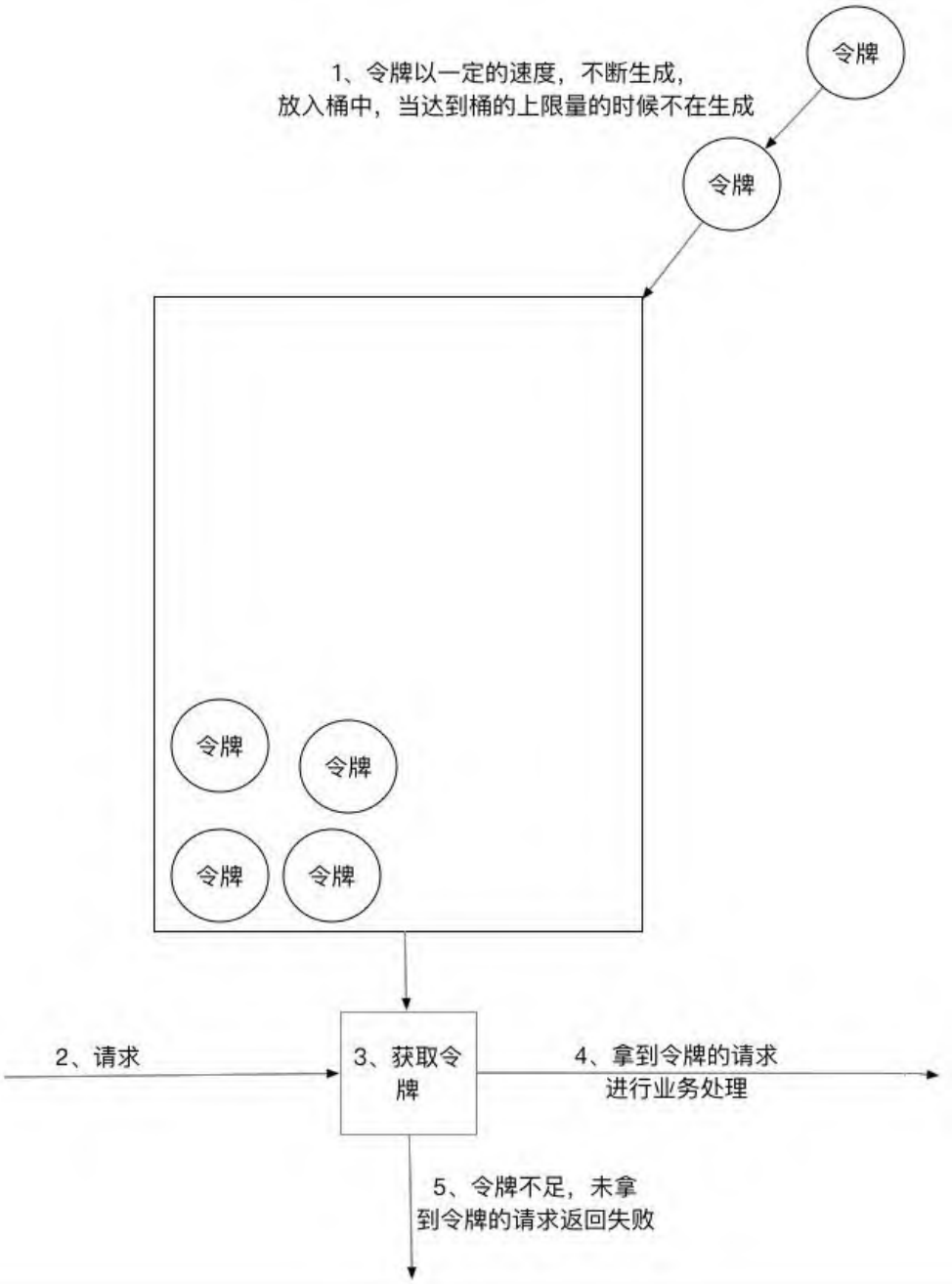
通俗的说，限流就是**限制一段时间内，用户访问资源的次数**，减轻服务器压力，限流大致分为两种：

1. IP 限流（5s 内同一个 ip 访问超过 3 次，则限制不让访问，过一段时间才可继续访问）

2. 请求量限流 (只要在一段时间内(窗口期)，请求次数达到阀值，就直接拒绝后面来的访问了，过一段时间才可以继续访问）（粒度可以细化到一个 api（url），一个服务）

13.2 本次限流模型

限流模型：漏斗算法 ，令牌桶算法，窗口滑动算法 计数器算法



入不敷出

1）、所有的请求在处理之前都需要拿到一个可用的令牌才会被处理；

2）、根据限流大小，设置按照一定的**速率**往桶里添加令牌；

3）、**桶设置最大的放置令牌限制**，当桶满时、新添加的令牌就被丢弃或者拒绝；

4）、请求达到后首先要获取令牌桶中的令牌，拿着令牌才可以进行其他的业务逻辑，处理完业务逻辑之后，将令牌直接删除；

5）、令牌桶有最低限额，当桶中的令牌达到最低限额的时候，请求处理完之后将不会删除令牌，以此保证足够的限流；

13.3 Gateway 结合 redis 实现请求量限流

Spring Cloud Gateway 已经内置了一个 RequestRateLimiterGatewayFilterFactory，我们可以直接使用。

目前 RequestRateLimiterGatewayFilterFactory 的实现依赖于 Redis，所以我们还要引入 spring-boot-starter-data-redis-reactive。

13.3.1 添加依赖

```
<!--限流要引入 Redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis-reactive</artifactId>
</dependency>
```

13.3.2 修改配置文件

```
server:
    port: 80
spring:
    application:
        name: gateway-80
    cloud:
        gateway:
```

```
            enabled: true
            routes:
                -   id: user-service
                    uri: lb://consumer-user-service
                    predicates:
                        - Path=/info/**
                    filters:
                        - name: RequestRateLimiter
                          args:
                              key-resolver: '#{@hostAddrKeyResolver}'
                              redis-rate-limiter.replenishRate: 1
                              redis-rate-limiter.burstCapacity: 3


    redis:  #redis 的配置

        host: 192.168.226.128
        port: 6379
        database: 0
eureka:
    instance:
        instance-id: ${spring.application.name}:${server.port}
        prefer-ip-address: true
    client:
        service-url:
            defaultZone: http://localhost:8761/eureka/
```

13.3.3 配置文件说明

在上面的配置文件，配置了 redis 的信息，并配置了 RequestRateLimiter 的限流过滤器，

该过滤器需要配置三个参数：

burstCapacity：令牌桶总容量。

replenishRate：令牌桶每秒填充平均速率。

key-resolver：用于限流的键的解析器的 Bean 对象的名字。它使用 SpEL 表达式根据

#{@beanName}从 Spring 容器中获取 Bean 对象。

13.3.4 创建配置类 RequestRateLimiterConfig

```
@Configuration
public class RequestRateLimiterConfig {


    /**
     * IP 限流
     * 把用户的 IP 作为限流的 Key
     *
     * @return
     */
    @Bean
    @Primary
    public KeyResolver hostAddrKeyResolver() {
        return (exchange) -> Mono.just(exchange.getRequest().getRemoteAddress().getHostName());
    }



    /**
     * 用户 id 限流
     * 把用户 ID 作为限流的 key
     *
     * @return
     */
    @Bean
    public KeyResolver userKeyResolver() {
        return exchange -> Mono.just(exchange.getRequest().getQueryParams().getFirst("userId"));
    }



    /**
     * 请求接口限流
     * 把请求的路径作为限流 key
     *
     * @return
     */
```

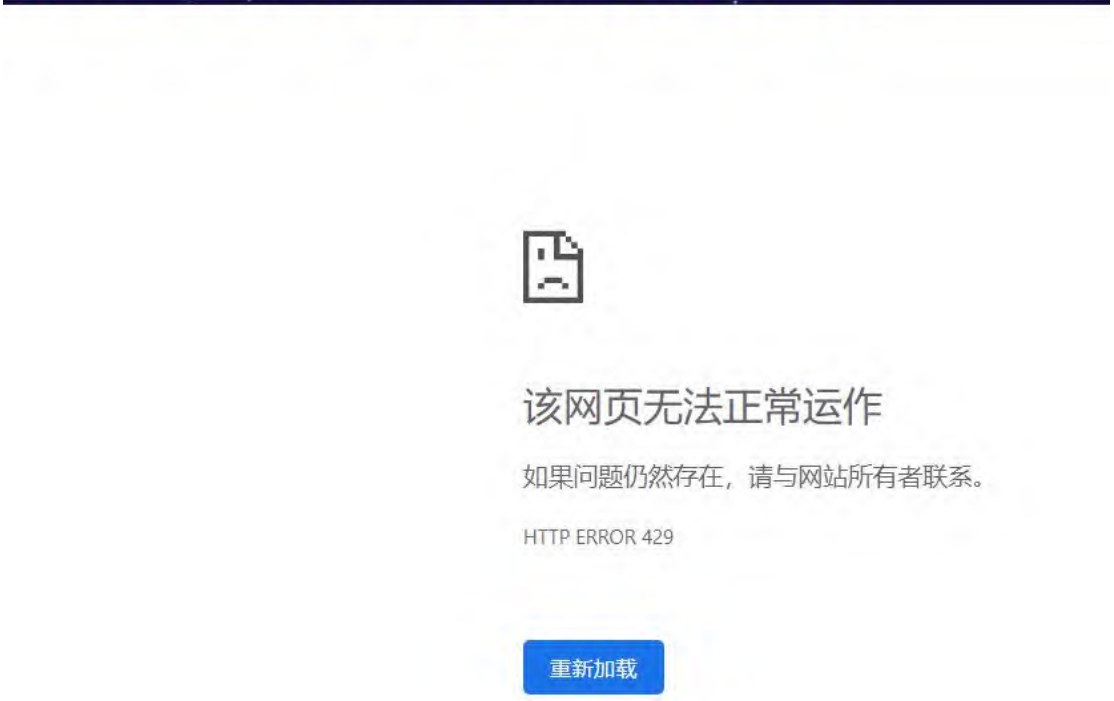
```
    @Bean
    public KeyResolver apiKeyResolver() {
        return exchange -> Mono.just(exchange.getRequest().getPath().value());
    }

}
```

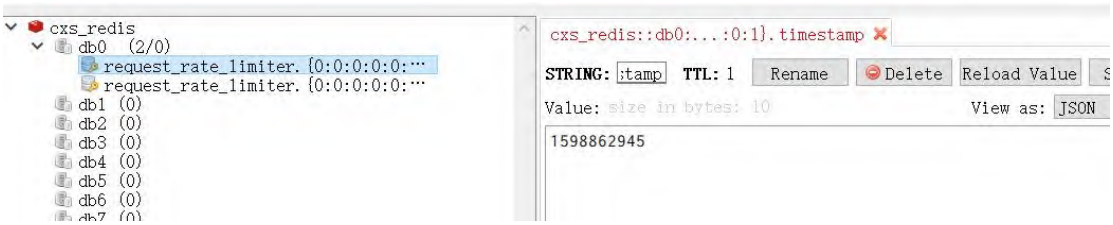
13.3.5 启动快速访问测试

http://localhost/info?token=asdad 快速访问后报 429

429 就是限流



查看 redis



14. 跨域配置

跨域? ajax 同源策略 8080 8081

因为网关是微服务的边缘 所有的请求都要走网关 跨域的配置只需要写在网关即可

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsWebFilter corsFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedMethod("*");
        config.addAllowedOrigin("*");
        config.addAllowedHeader("*");

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource(new PathPatternParser());
        source.registerCorsConfiguration("/**", config);

        return new CorsWebFilter(source);
    }
}
```

```
spring:
  cloud:
    gateway:
      globalcors:
        corsConfigurations:
          '[/**]':  // 针对哪些路径

            allowCredentials: true // 这个是可以携带 cookie
```

```
            allowedHeaders: '*'
            allowedMethods: '*'
            allowedOrigins: '*'
```

15. 总结和面试

1.你们网关用的什么 ? Gateway zuul

2.你们网关里面写什么代码?

跨域，路由（动态路由，负载均衡）ip 黑名单拦截，Token 的校验，对请求进行过滤（请求参数校验） 对响应做处理（状态码，响应头） 熔断 限流

微服务的网关，可以很好地将具体的服务和浏览器隔离开，只暴露网关的地址给到浏览器

在微服务网关中，可以很好的实现校验认证，负载均衡（lb），黑名单拦截，限流等。

15.1 Gateway 和 zuul 的区别 ZuulFilter

Zuul 也是 web 网关，本质上就是一组过滤器，按照定义的顺序，来执行过滤操作

二者的区别：

1. 两者均是 web 网关，处理的是 http 请求

2. Gateway 是 springcloud 官方的组件，zuul 则是 netflix 的产品

springcloud，netflix ，alibaba（nacos，sentinel，dubbo zk，seata，rocketmq）

3. gateway 在 spring 的支持下，内部实现了限流、负载均衡等，扩展性也更强，但同时也限制了仅适合于 Spring Cloud 套件。而 zuul 则可以扩展至其他微服务框架中，其内部没有实现限流、负载均衡等。

4. Gateway(Netty NIO)很好的支持异步(spring5.x ,webFlux 响应式编程默认是异步的),而 zuul1.0 仅支持同步 BIO zuul2.0 以后也支持异步了

15.2 Nginx 在微服务中的地位

最后简单聊一下 nginx，在过去几年微服务架构还没有流行的日子里，nginx 已经得到了广大开发者的认可，其性能高、扩展性强、可以灵活利用 lua 脚本构建插件的特点让人没有抵抗力。（nginx 的请求转发 最大并发是多个次，每秒 5w-10w 左右） 3w 左右

有一个能满足我所有需求还很方便我扩展的东西，还免费，凭啥不用？？

但是，如今很多微服务架构的项目中不会选择 nginx，我认为原因有以下几点：

微服务框架一般来说是配套的，集成起来更容易

如今微服务架构中，仅有很少的公司会面对无法解决的性能瓶颈，而他们也不会因此使用 nginx，而是选择开发一套适合自己的微服务框架（很多公司会对现有框架进行修改）

spring boot 对于一些模板引擎如 FreeMarker、themleaf 的支持是非常好的，很多应用还没有达到动、静态文件分离的地步，对 nginx 的需求程度并不大。

动静分离： css js 可以放在 nginx

单体项目需要部署 对 nginx 的使用的需求还是比较大的

斗鱼 不是使用后端技术 如何实现大规模缓存

使用 Nginx 做大规模的静态资源缓存

不是为了用技术而用技术　按照实际业务来　目的是盈利

无论如何，nginx 作为一个好用的组件，最终使不使用它都是由业务来驱动的，只要它能为我们方便的解决问题，那用它又有何不可呢?

找工作思想：　不要为了用技术而去用技术

ssm 吃一辈子　稳定 挣钱　　你想技术提升 跳大厂　　先入行 一年就跳槽

15.3 关于限流，面试不会直接问，而是间接来问 问 不卖超

比如：如果在抢购过程中，用户量请求非常大，怎么确保商品不会卖超

Redis 单线程 （IO 为什么快，因为我们现在的处理器是多核心数的，redis 底层使用的是 IO 的**多路复用**）

一般人只会在意商品卖超，而忘记了限流的重要性

Mq（限流 削峰，异步，解耦合）

15.4 健康状态检查等

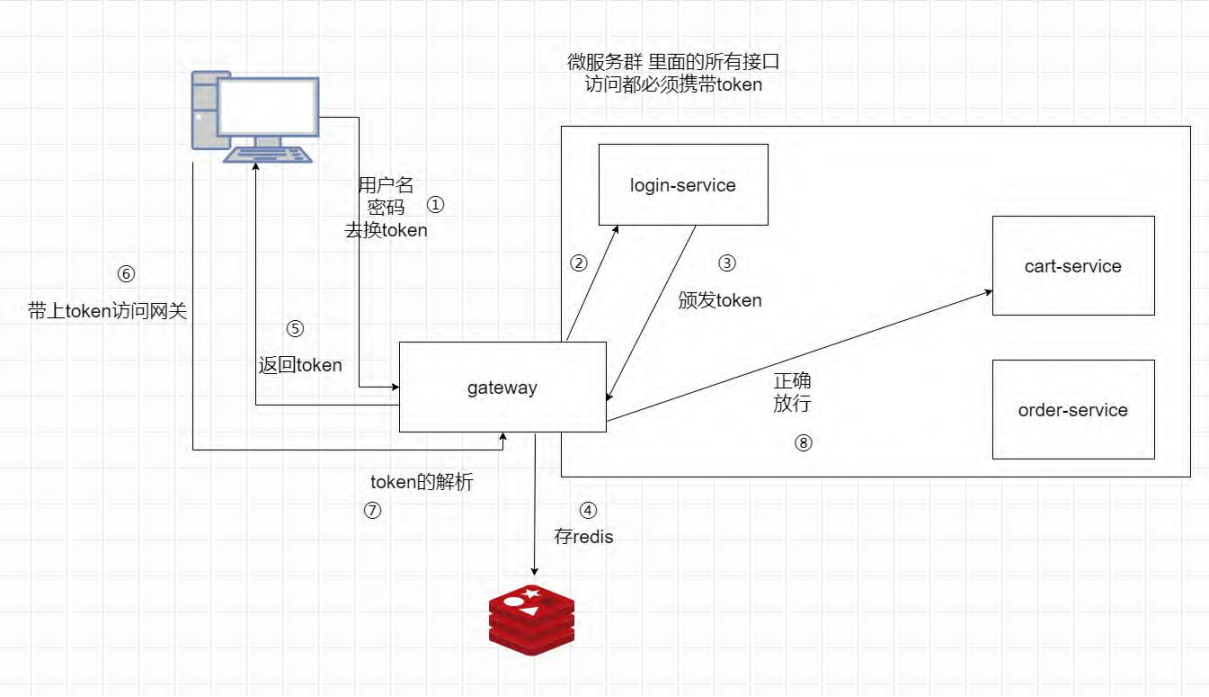
健康检查的依赖

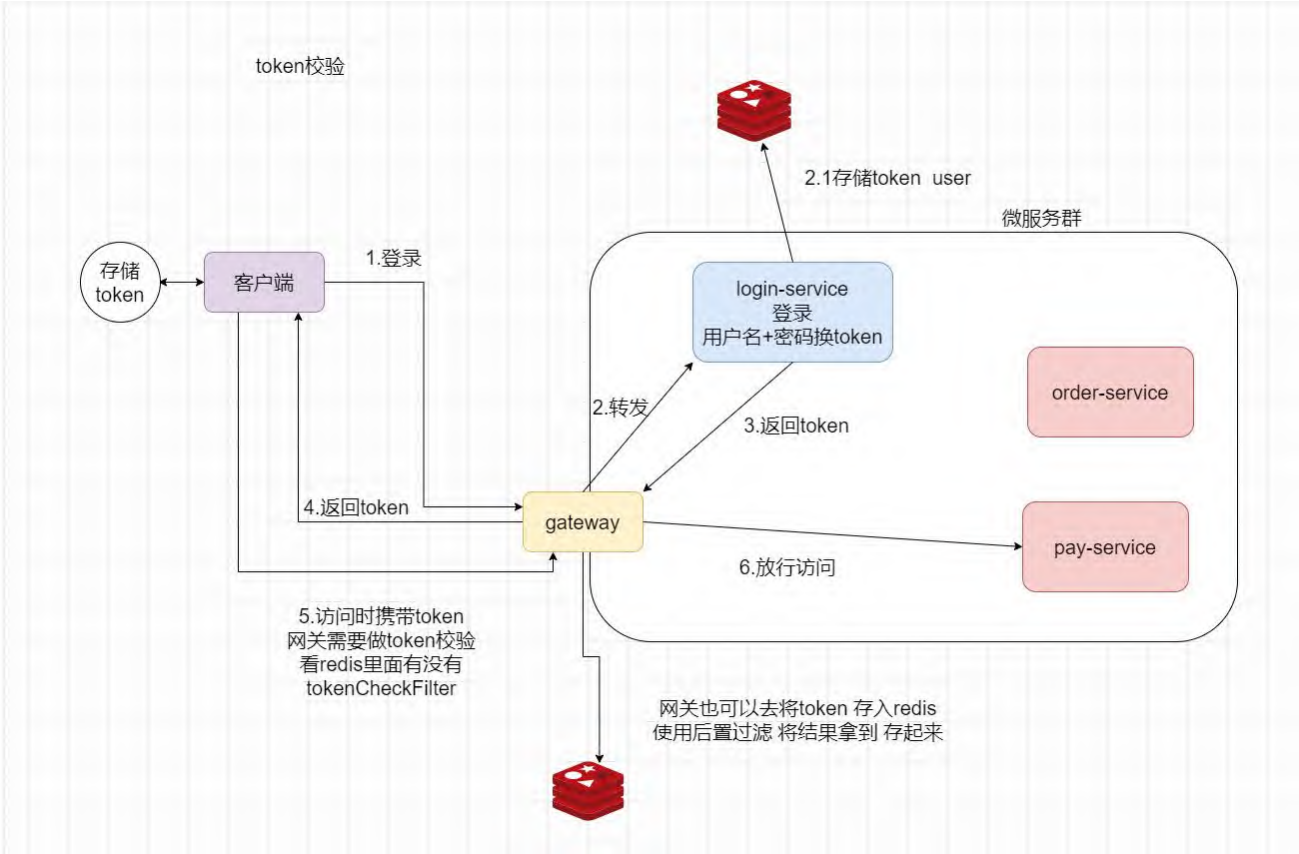
```
<!--        健康检查的依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

添加配置文件

```
management:
    endpoints:
        web:
            exposure:
```

```
include: '*'    #暴露检查的端点
```




token校验
2.1存储token user
微服务群
存储
token
客户端
1.登录
login-service
登录
用户名+密码换token
order-service
2.转发
3.返回token
4.返回token
gateway
pay-service
6.放行访问
5.访问时携带token
网关需要做token校验
看redis里面有没有
tokenCheckFilter
网关也可以去将token 存入redis
使用后置过滤 将结果拿到 存起来