ch02-服务网关

公众号：锋哥聊编程

主讲：小锋

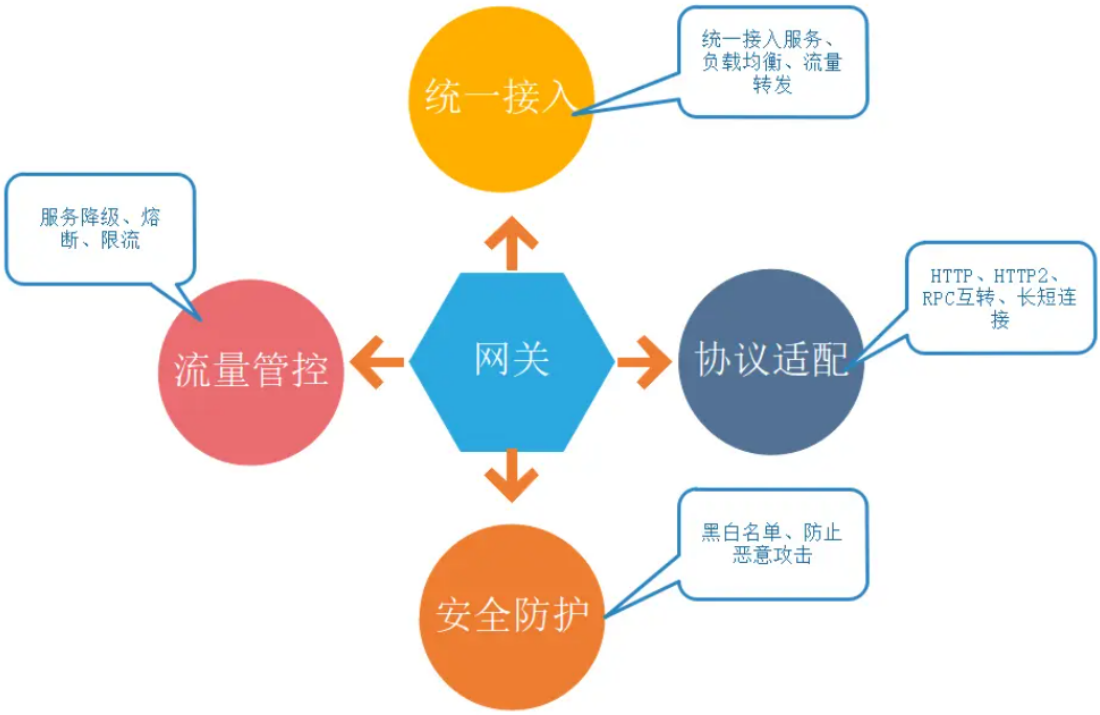
1、为什么需要网关（Gateway）？

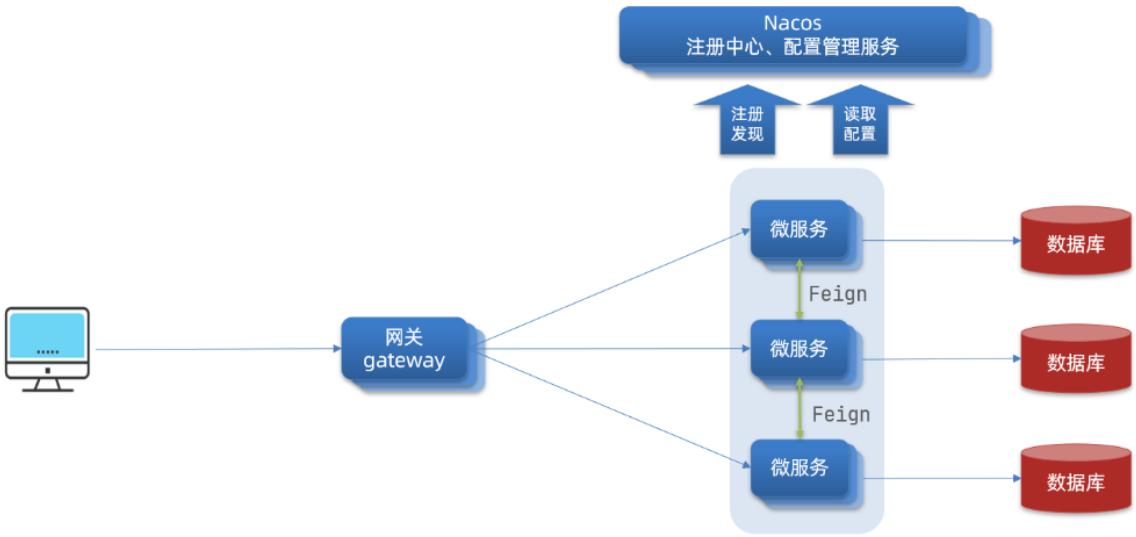
网关这个词，最早是出现在网络设备中，比如在彼此隔离的两个局域网中间的起到路由功能、隔离功能、安全验证功能的网络设备，通常被称为“网关”。

在软件开发方面，网关通常是用来隔离用户端和服务端的软件应用，通常被称为`API网关`。

微服务架构中，API网关的好处是：

- 面向前端开发人员更加友好，前端开发人员面向的入口减少，便于维护
- 服务访问的认证鉴权更加方便，可以放在API网关统一去做。避免分散造成的开发及维护成本。
- 访问日志、限流等公共服务也可以在网关上集中完成。避免分散造成的开发及维护成本。





2、还有必要学习Zuul吗?

Spring Cloud Gateway VS Zuul

当使用了API网关之后，意味着网关作为流量入口比微服务承担更多的流量负载。因此网关本身的架构性能及稳定性至关重要!

Zuul是Spring Cloud框架的第一代网关， 因为Zuul是是基于Servlet的阻塞IO模型开发的，性能成为Zuul最大的短板。

Zuul 1.0在Netflix官方已经进入了维护阶段， Netflix也很早就宣称了要对Zuul进行升级改造，也就是Zuul 2.0，但Zuul2.0与SpringCloud的整合一直处于难产阶段。后来， Spring 社区自己开发了`Spring Cloud Gateway`。采用了Spring 官方的响应式非阻塞框架`webflux`。官网测试结果性能是Zuul的1.6倍。

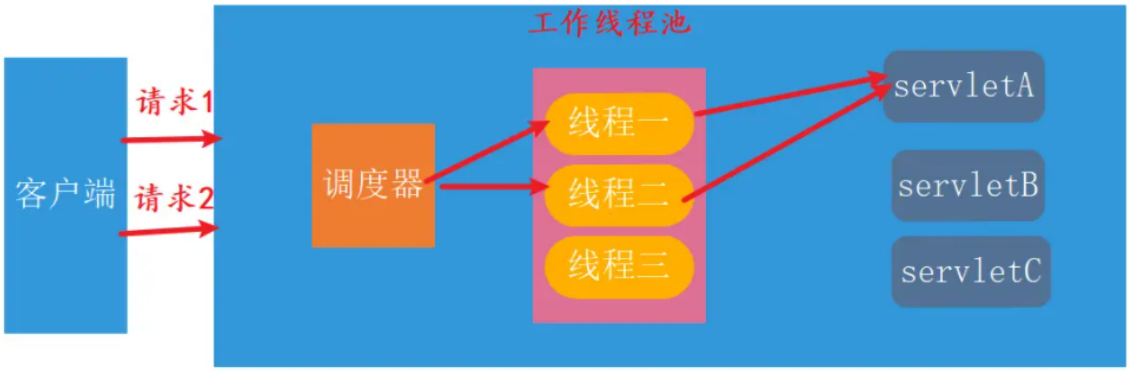
非阻塞异步IO模型

Spring Cloud GateWay 采用Spring Flux全新响应式的非阻塞IO框架实现， 性能有了长足的进步（宣称性能比Zuul提升1.6倍） 。而 WebFlux底层是基于高性能的非阻塞IO通信框架Netty实现的。

阻塞IO模型（Zuul）

Zuul是基于Servlet的阻塞IO模型，包含SpringMVC/Struts2等框架也是Servlet为基础的阻塞IO模型的代表。

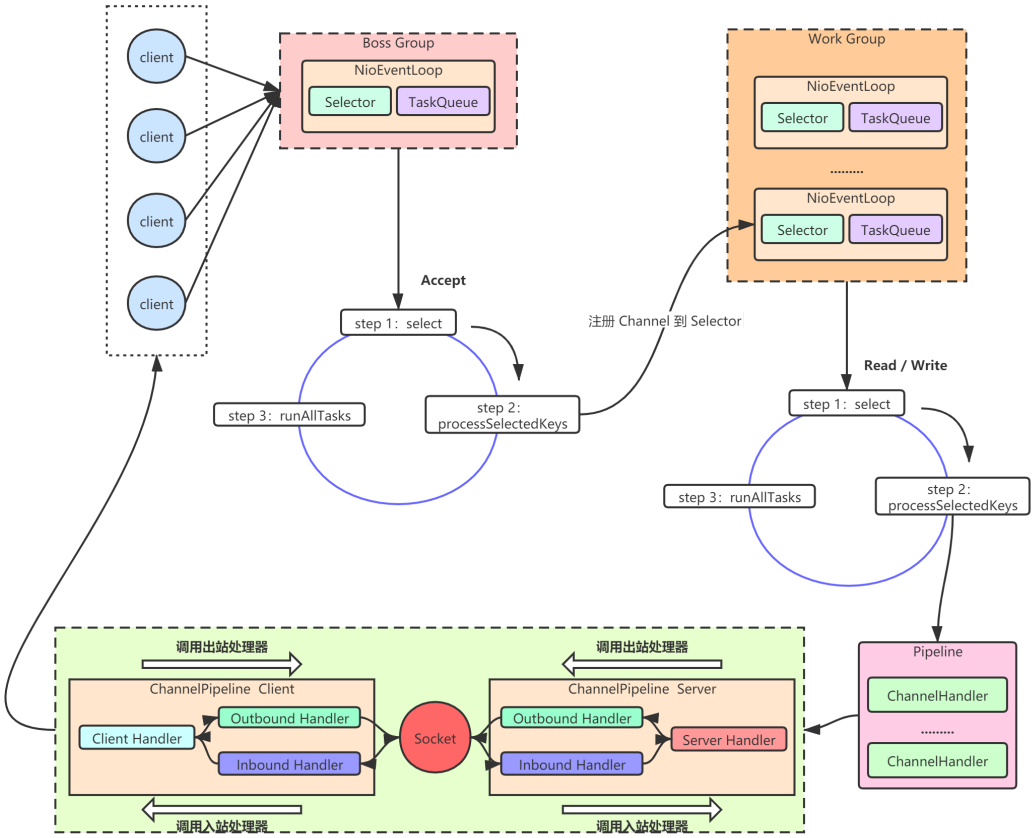
这种模型最大的问题是一个请求会占用一个线程，当线程池内线程都被占用后新来的请求就只能等待。



这种阻塞IO模型就好比公司的一名开发人，从需求分析、需求设计、代码开发、迭代测试、上线部署运维都是一个人完成。如果软件公司有10名开发人员，最多只能同时进行10个项目的开发。假设客户有更多的项目过来也没有办法，只能等着。

非阻塞IO模型（Spring Cloud Gateway）

WebFlux底层实现是久经考验的Netty非阻塞IO通信框架。Netty底层处理请求的流程大致如下：



分工明确： Netty 抽象出两组线程池 ： BoosGroup用于Accetpt连接建立事件并分发请求，WorkerGroup用于处理I/O读写事件和业务逻辑。

每个`Boss NioEventLoop`循环执行的任务包含3步：

1. 轮询accept事件
2. 处理accept I/O事件，与Client建立连接，生成NioSocketChannel，并将NioSocketChannel注册到某个Worker NioEventLoop的Selector上
3. 处理任务队列中的任务，runAllTasks。任务队列中的任务包括用户调用eventloop.execute或schedule执行的任务，或者其它线程提交到该eventloop的任务。

每个`Worker NioEventLoop`循环执行的任务包含3步：

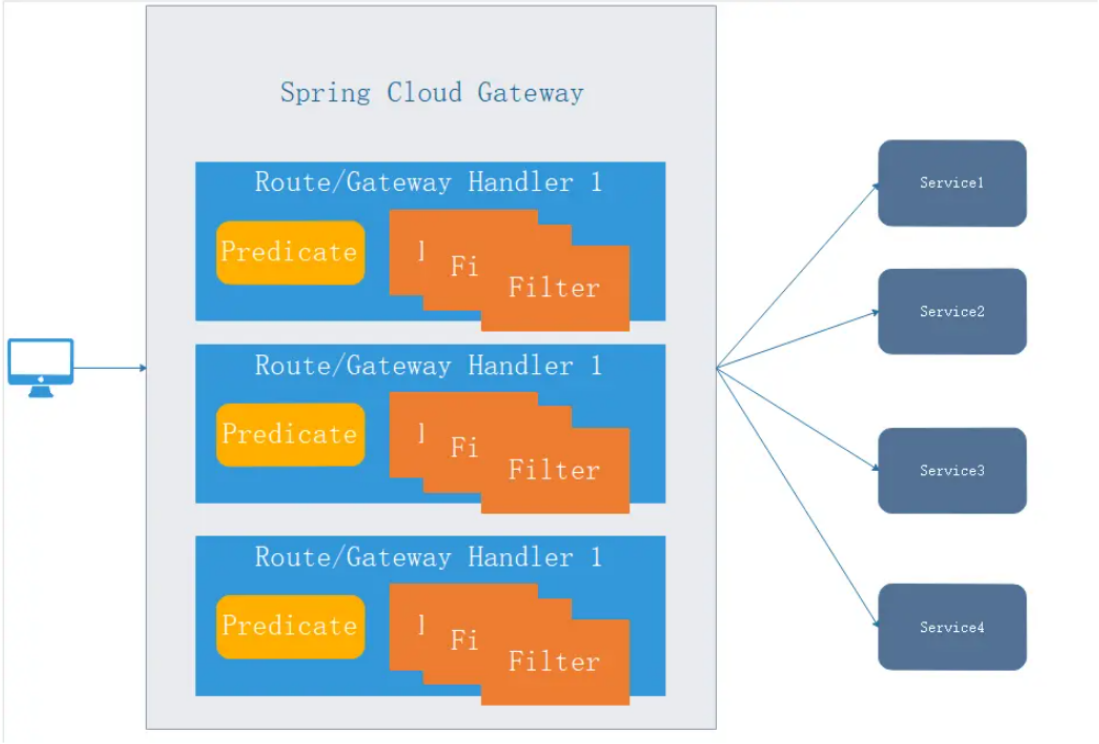
1. 轮询read、write事件；
2. 处I/O事件，即read、write事件，在NioSocketChannel可读、可写事件发生时进行处理
3. 处理任务队列中的任务，runAllTasks。

以上Netty这种非阻塞IO模型核心意义在于：**提高了有限资源下的服务请求并发处理能力，而部署缩短单个服务请求的响应时长。**

这就好比需求分析由需求分析师完成，需求设计由架构师完成，代码编写由开发人员完成，迭代测试由测试团队完成，上线部署运维由运维团队完成。公司内的每种角色分工明确，就能在有限的资源下，去承接更多的客户项目，也更合理地分配人力资源！

3、Spring Cloud Gateway核心概念与流程

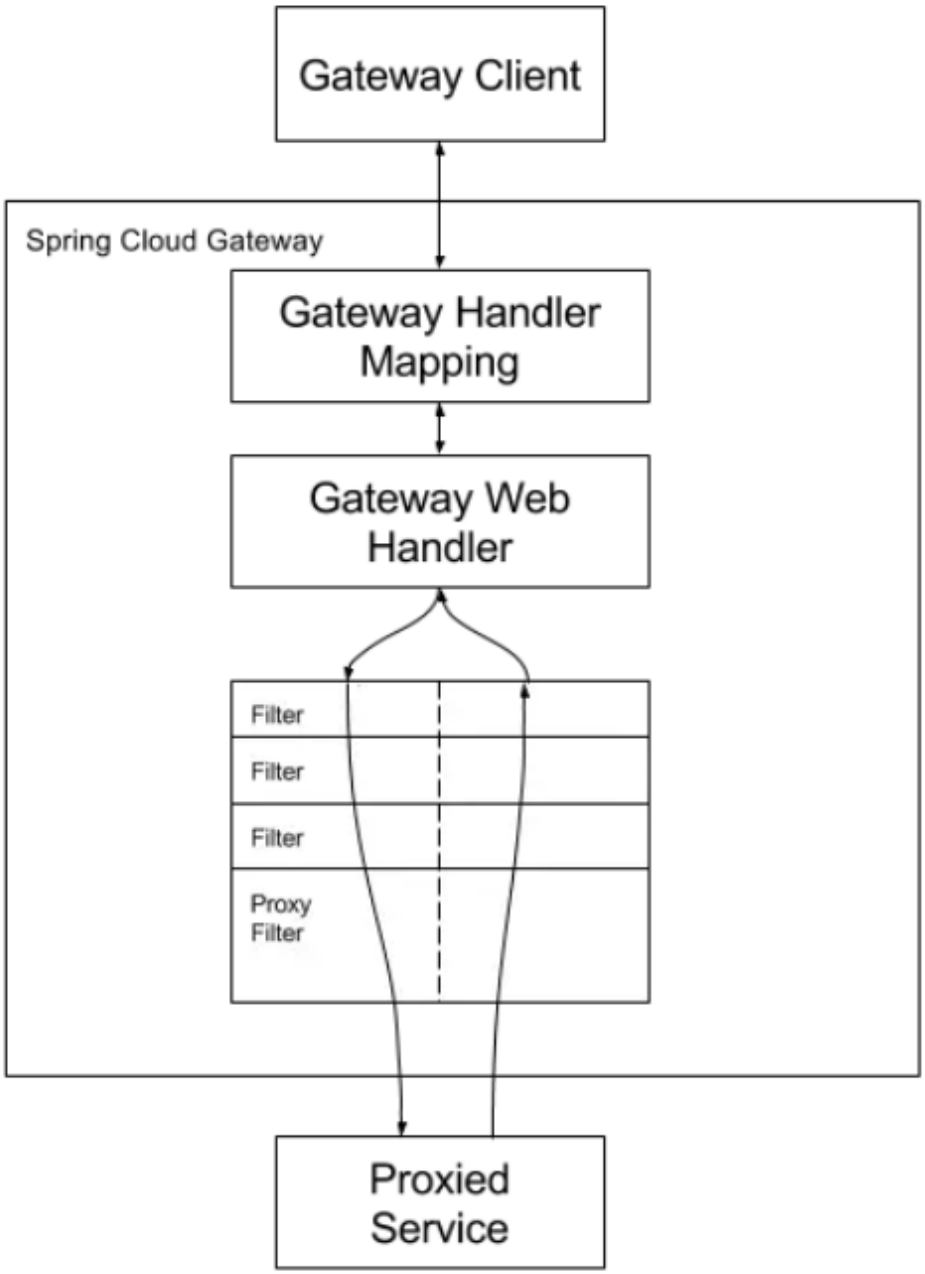
Spring Cloud Gateway核心概念



- Route（路由）：路由实现网关的基础元素，由id、url、predicate和filter组成。当请求通过网关的时候，由Gateway Handler Mapping通过predicate判断是否与路由匹配，当predicate=true的时候，匹配到对应的路由。
- Predicate（断言）：是Java8中提供的一个函数，允许开发人员根据其定义规则匹配请求。比如根据请求头、请求参数来匹配路由。可以认为它就是一个匹配条件的定义。
- Filter（过滤器）：对请求处理之前之后进行一些统一的业务处理、比如：认证、审计、日志、访问时长统计等。

Spring Cloud Gateway执行流程

Spring Cloud的工作原理图如下：



- 客户端向Spring Cloud Gateway发送请求，当请求的路径与网关定义的路由映射规则相匹配（断言）。该请求就会被发送到网关的Web Handler进行处理，执行特定的过滤器链。
- 大家注意图中的虚线，虚线左侧表示在请求处理之前的执行逻辑（Pre过滤器），虚线右侧表示请求处理之后的执行逻辑（post过滤器）

4、Spring Cloud Gateway快速入门

创建gateway工程，引入依赖

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

我们搭建的Spring Cloud Gateway(gateway-server)项目虽然是一个web项目，但是底层已经使用的是spring-boot-starter-webflux，而不是`spring-boot-starter-web`。所以`千万不要`同时导入`spring-boot-starter-web`，会报错的！

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

启动类

```
/**
 * 网关微服务
 */
@SpringBootApplication
public class GatewayServerApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayServerApplication.class,args);
    }
}
```

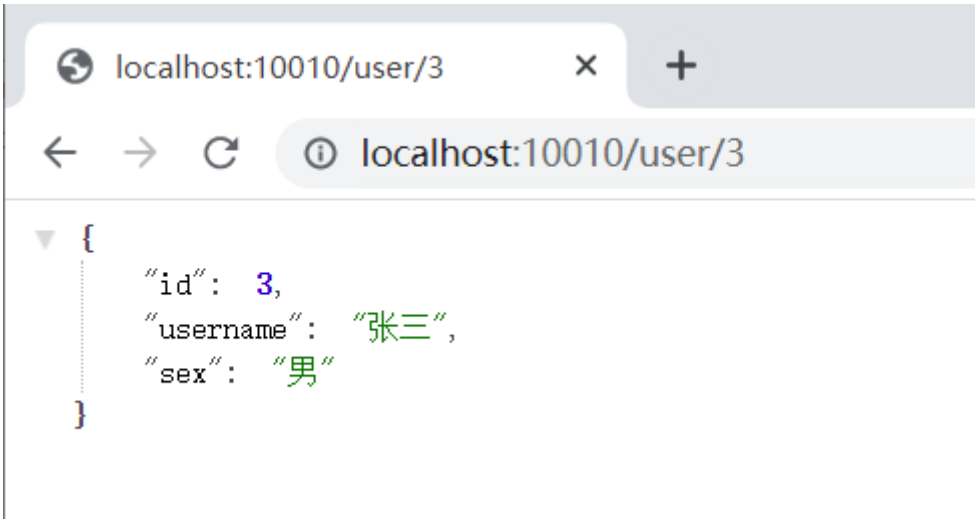
编写路由规则

application.yml

```
# 网关端口
server:
  port: 10010

spring:
  application:
    name: gateway-server # 服务名称

  cloud:
    gateway:
      routes:
        - id: user                      # 路由 ID，唯一
          uri: http://localhost:9101    # 目标 URI，路由到微服务的地址
          predicates:                   # 请求转发判断条件
            - Path=/user/**             # 匹配对应 URL 的请求，将匹配到的请求追加在目标
URI 之后
```

测试

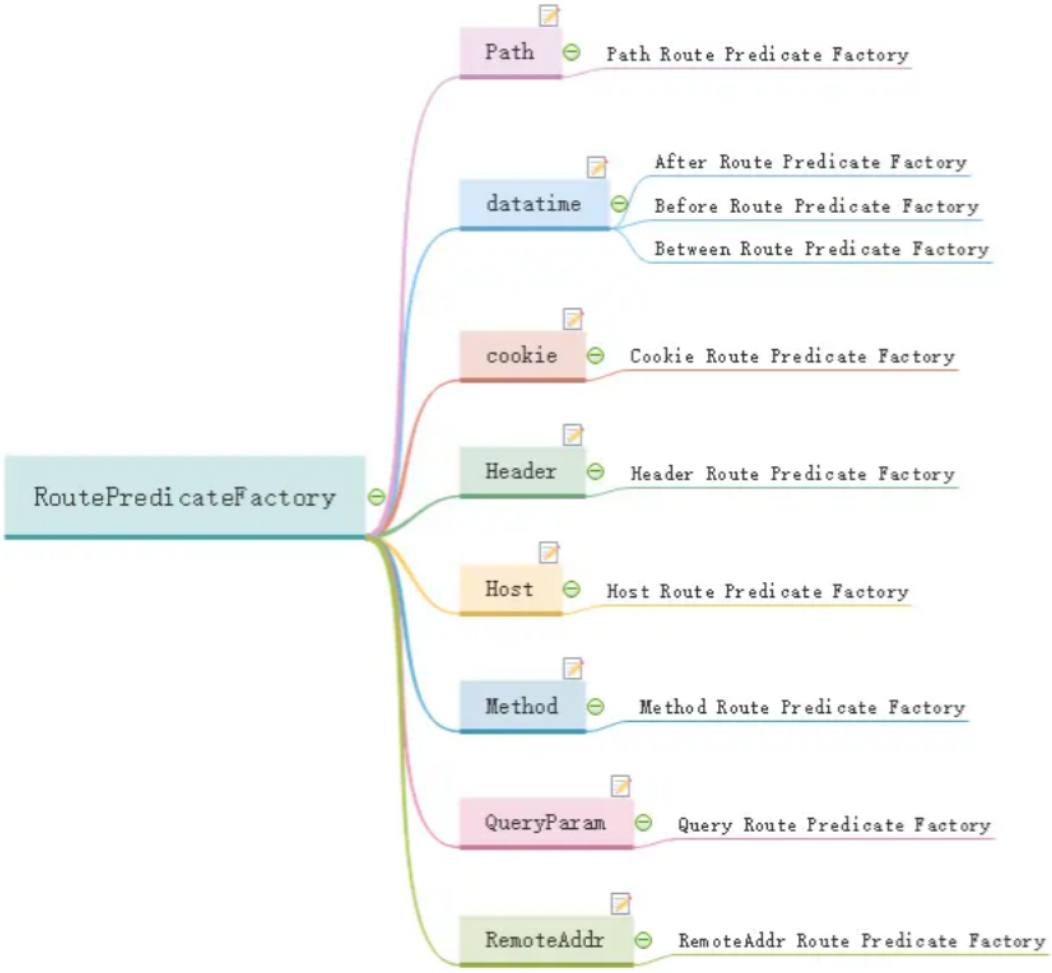
启动gateway-server，访问地址：http://localhost:10010/user/3



5、路由断言工厂的使用

路由断言工厂介绍

在 Spring Cloud Gateway 中 利用 Predicate 实现了各种路由匹配规则，有通过 Header、请求参数等不同的条件来进行作为条件匹配到对应的路由。 （说白了 Predicate 就是为了实现一组匹配规则，方便让请求过来找到对应的 Route 进行处理）



重点掌握Path这种Predicate 方法!

了解多种Predicate判断用法

通过日期时间条件匹配

After Route Predicate Factory使用的是时间作为匹配规则，只要当前时间大于设定时间，路由才会匹配请求。以下After规则配置：在东8区的2022-10-14T16:31:47之后，所有请求都转发到`http://localhost:9101`。

```
spring:
  cloud:
    gateway:
      routes:
      - id: after_route
        uri: http://localhost:9101
        predicates:
        - After=2022-10-14T16:31:47.789+08:00
```

Before Route Predicate Factory也是使用时间作为匹配规则，只要当前时间小于设定时间，路由才会匹配请求。以下Before规则配置：在东8区的2022-10-14T19:53:42之前，所有请求都转发到`http://localhost:9101`。

```
spring:
  cloud:
    gateway:
      routes:
      - id: before_route
        uri: http://localhost:9101
        predicates:
        - Before=2022-10-14T19:53:42.789+08:00
```

Between Route Predicate Factory也是使用两个时间作为匹配规则，只要当前时间大于第一个设定时间，并小于第二个设定时间，路由才会匹配请求。以下Between规则配置：在东8区的2022-10-14T16:31:47之后，2022-10-14T19:53:42之前的时间段内，所有请求都转发到`http://localhost:9101`。

```
spring:
  cloud:
    gateway:
      routes:
      - id: between_route
        uri: http://localhost:9101
        predicates:
        - Between=2022-10-14T16:31:47.789+08:00, 2022-10-14T19:53:42.789+08:00
```

通过 Cookie 匹配

Cookie Route Predicate 可以接收两个参数，一个是 Cookie name ,一个是正则表达式Cookie value，路由规则会通过获取对应的 Cookie name 值和正则表达式去匹配，如果匹配上就会执行路由，如果没有匹配上则不执行。

```
spring:
  cloud:
    gateway:
      routes:
      - id: cookie_route
        uri: http://localhost:9101
        predicates:
        - Cookie=cookiename, cookievalue
```

使用 curl 测试，命令行输入:`curl http://localhost:10010 --cookie "cookiename=cookievalue"`，则会返回`http://localhost:9101`页面代码。也就是说当我们的请求携带了指定的cookie键值对的时候，请求才向正确的uri地址转发。

通过 Header 属性匹配

Header Route Predicate 和 Cookie Route Predicate 一样，也是接收 2 个参数，一个 header 中属性名称和一个正则表达式value，这个属性值和正则表达式value匹配的时候才进行路由转发。

```
spring:
  cloud:
    gateway:
      routes:
      - id: header_route
        uri: http://localhost:9101
        predicates:
        - Header=X-Request-Id, \d+
```

上面的配置规则表示路由匹配存在名为`X-Request-Id`，内容为数字的header的请求，将请求转发到`http://localhost:9101`。
使用 curl 测试，命令行输入:`curl http://localhost:10010 -H "X-Request-Id:88"`,则返回页面代码证明匹配成功。将参数-H "X-Request-Id:88"改为-H "X-Request-Id:somestr"再次执行时返回404证明没有匹配。

通过 Host 匹配

Host Route Predicate 接收一组参数，一组匹配的域名列表，这个模板是一个 ant 风格的模板，用逗号作为分隔符。它通过参数中的主机地址作为匹配规则。

```
spring:
  cloud:
    gateway:
      routes:
      - id: host_route
        uri: http://localhost:9101
        predicates:
        - Host=**.somehost.org,**.anotherhost.org
```

路由会匹配Http的Host诸如：`www.somehost.org`或`beta.somehost.org`或`www.anotherhost.org`的请求。

通过请求Method匹配

通过HTTP的method是 POST、GET、PUT、DELETE 等不同的请求方式来进行路由。

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: method_route
        uri: http://localhost:9101
        predicates:
        - Method=GET
```

以上规则决定：路由会匹配到所有GET方法的请求，其他的HTTP方法不做匹配。
使用 curl 测试，命令行输入:

- curl 默认是以 GET 的方式去请求，`curl http://localhost:10010`,测试返回页面代码，证明匹配到路由predicate规则
- 指定curl使用POST方法发送请求，`curl -X POST http://localhost:10010`,返回 404 没有找到，证明没有匹配

通过请求参数匹配

Query Route Predicate 支持传入两个参数，一个是属性名一个为属性值，属性值可以是正则表达式。

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: query_route
        uri: http://localhost:9101
        predicates:
        - Query=foo, ba.
```

路由会匹配所有包含`foo`，并且`foo`的内容为诸如：`bar`或`baz`等符合`ba.`正则规则的请求。

使用 curl 测试，命令行输入:`curl localhost:10010?foo=bax`测试可以返回页面代码，将 foo的属性值改为 bazx再次访问就会报 404,证明路由需要匹配正则表达式才会进行路由。

通过请求 ip 地址进行匹配

Predicate 也支持通过设置某个 ip 区间号段的请求才会路由，RemoteAddr Route Predicate 接受 cidr 符号(IPv4 或 IPv6)字符串的列表(最小大小为1)，例如 192.168.0.1/16 (其中 192.168.0.1 是 IP 地址，16 是子网掩码)。

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: remoteaddr_route
        uri: http://localhost:9101
        predicates:
        - RemoteAddr=192.168.1.1/24
```

可以将此地址设置为本机的 ip (192.168.1.4)地址进行测试，`curl localhost:10010`，则此路由将匹配。

通过权重分流匹配

通过权重weight分流匹配的predicate有两个参数：`group`和`weight`（一个int）。权重是按组计算的。以下示例配置权重路由谓词：

```
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
```

这条路线会将约80%的流量转发至weighthigh.org，并将约20%的流量转发至weightlow.org。

组合使用

```
spring:
  cloud:
    gateway:
      routes:
        - id: multi-predicate
          uri: http://localhost:9101
          order: 0
          predicates:
            - Host=**.foo.org
            - Path=/headers
            - Method=GET
            - Header=X-Request-Id, \d+
            - Query=foo, ba.
            - Query=baz
            - Cookie=chocolate, ch.p
```

各种 Predicates 同时存在于同一个路由时，请求必须同时满足所有的条件才被这个路由匹配。

6、自定义断言工厂

虽然官方为我们提供了诸多的Predicate Factory，能够满足我们大部分的场景需求。但是不排除有些情况下，Predicate路由匹配条件比较复杂，这时就需要我们来自定义实现。

需求：所有Path为`/user`、`/role`、`/permission`、`/menu`开头的路径都转发到mafeng-user微服务。

```
package cn.mf5.gateway.predicate;

import
org.springframework.cloud.gateway.handler.predicate.AbstractRoutePredicateFactor
y;
import org.springframework.stereotype.Component;
```

```java
import org.springframework.web.server.ServerWebExchange;

import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

@Component
public class RbacAuthRoutePredicateFactory extends
AbstractRoutePredicateFactory<RbacAuthRoutePredicateFactory.Config> {

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

    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("flag");
    }

    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        return exchange -> {
            //获取访问路径
            String uri = exchange.getRequest().getURI().getPath();
            if(config.getFlag().equals("rbac")
                && ( uri.startsWith("/user")
                    ||uri.startsWith("/role")
                    ||uri.startsWith("/permission")
                    ||uri.startsWith("/menu")
                     )
            ) {
                return true;//匹配成功
            }
            return false; //匹配失败
        };
    }

    public static class Config{
        private String flag;

        public String getFlag() {
            return flag;
        }

        public void setFlag(String flag) {
            this.flag = flag;
        }
    }

}
```

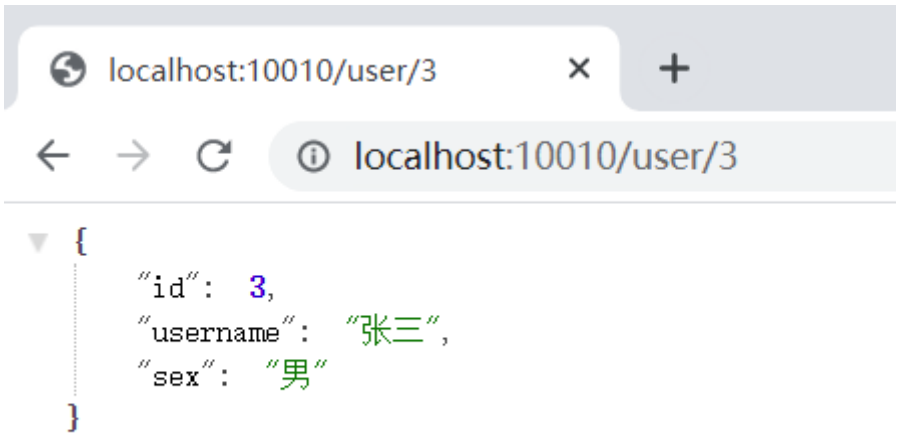
路由断言配置：

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user                        # 路由 ID，唯一
          uri: http://localhost:9101      # 目标 URI，路由到微服务的地址
          predicates:                     # 请求转发判断条件
            - RbacAuth=rbac
```

或者

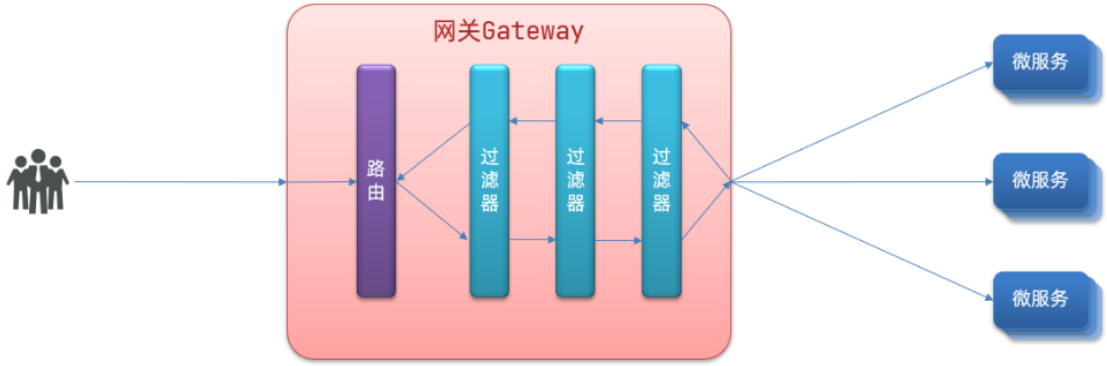
```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user                        # 路由 ID，唯一
          uri: http://localhost:9101      # 目标 URI，路由到微服务的地址
          predicates:                     # 请求转发判断条件
            - name: RbacAuth
              args:
                flag: rbac
```

访问：http://localhost:10010/user/3



7、过滤器介绍和使用

过滤器介绍



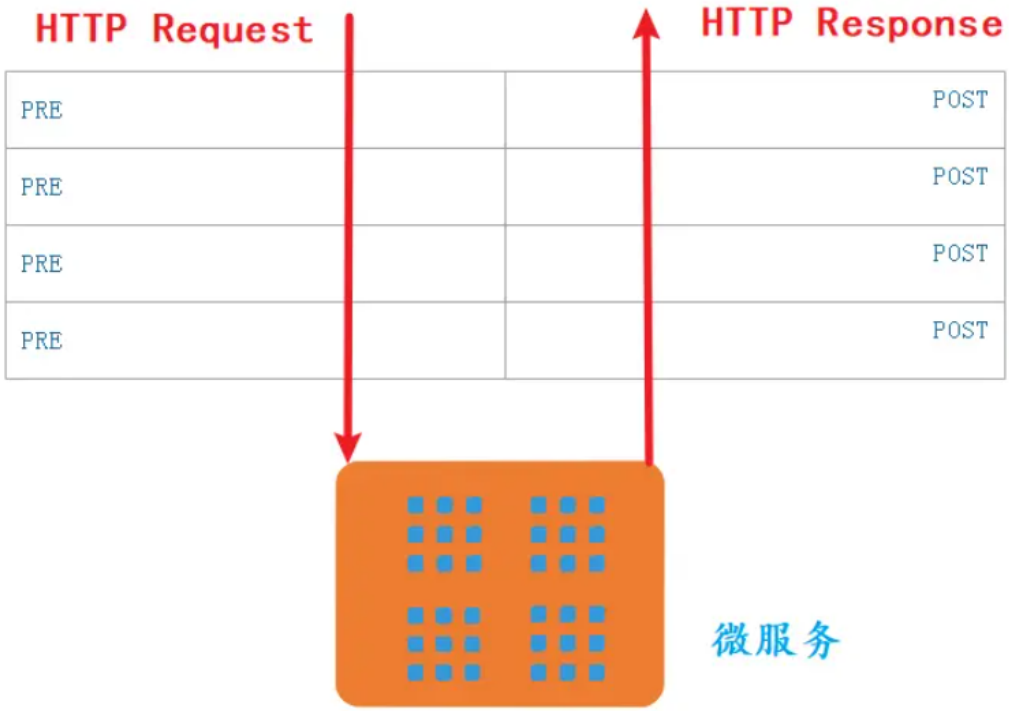
微服务网关经常需要对请求进行一些过滤操作，比如：鉴权之后添加Header携带令牌等。在过滤器中可以

- 为为请求增加请求头、增加请求参数 、增加响应头等等功能
- 鉴权、记录审计日志、统计请求响应时长等共性服务操作

微服务系统中有很多的服务，我们不希望在每个服务上都去开发鉴权、记录审计日志、统计请求响应时长等共性服务操作。所以对于这样的重复开发或继承类工作，放在Gateway上面统一去做是最好不过了。

过滤器生命周期

Spring Cloud Gateway 的 Filter 的生命周期很简单，只有两个：“PRE” 和 “POST”。



- 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。
- 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。

过滤器分类

Spring Cloud Gateway 的 Filter 从作用范围可分为：

- GatewayFilter（局部过滤器）：应用到单个路由上。

- GlobalFilter（全局过滤器）：应用到所有的路由上

Gateway filter（局部过滤器）

过滤器工厂	作用	参数
AddRequestHeader	为原始请求添加Header	Header的名称及值
AddRequestParameter	为原始请求添加请求参数	参数名称及值

过滤器工厂	作用	参数
AddResponseHeader	为原始响应添加Header	Header的名称及值
DedupeResponseHeader	剔除响应头中重复的值	需要去重的Header名称及去重策略
Hystrix	为路由引入Hystrix的断路器保护	`HystrixCommand`的名称
FallbackHeaders	为fallbackUri的请求头中添加具体的异常信息	Header的名称
PrefixPath	为原始请求路径添加前缀	前缀路径
PreserveHostHeader	为请求添加一个preserveHostHeader=true的属性，路由过滤器会检查该属性以决定是否要发送原始的Host	无
RequestRateLimiter	用于对请求限流，限流算法为令牌桶	keyResolver、rateLimiter、statusCode、denyEmptyKey、emptyKeyStatus
RedirectTo	将原始请求重定向到指定的URL	http状态码及重定向的url
RemoveHopByHopHeadersFilter	为原始请求删除IETF组织规定的一系列Header	默认就会启用，可以通过配置指定仅删除哪些Header
RemoveRequestHeader	为原始请求删除某个Header	Header名称
RemoveResponseHeader	为原始响应删除某个Header	Header名称
RewritePath	重写原始的请求路径	原始路径正则表达式以及重写后路径的正则表达式
RewriteResponseHeader	重写原始响应中的某个Header	Header名称，值的正则表达式，重写后的值
SaveSession	在转发请求之前，强制执行`webSession::save`操作	无
SecureHeaders	为原始响应添加一系列起安全作用的响应头	无，支持修改这些安全响应头的值
SetPath	修改原始的请求路径	修改后的路径
SetResponseHeader	修改原始响应中某个Header的值	Header名称，修改后的值

过滤器工厂	作用	参数
SetStatus	修改原始响应的状态码	HTTP 状态码，可以是数字，也可以是字符串
StripPrefix	用于截断原始请求的路径	使用数字表示要截断的路径的数量
Retry	针对不同的响应进行重试	retries、statuses、methods、series
RequestSize	设置允许接收最大请求包的大小。如果请求包大小超过设置的值，则返回 `413 Payload Too Large`	请求包大小，单位为字节，默认值为5M
ModifyRequestBody	在转发请求之前修改原始请求体内容	修改后的请求体内容
ModifyResponseBody	修改原始响应体的内容	修改后的响应体内容
Default	为所有路由添加过滤器	过滤器工厂名称及值

注意：每个过滤器工厂都对应一个实现类，并且这些类的名称必须以`GatewayFilterFactory`结尾，这是Spring Cloud Gateway的一个约定，例如`AddRequestHeader`对应的实现类为`AddRequestHeaderGatewayFilterFactory`。

AddRequestHeader例子

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user
          uri: http://localhost:9101
          predicates:
            - Path=/user/**
          filters:
            - AddRequestHeader=Authorization,1001  # 经过路由的请求都会携带Authorization请求头
```

StripPrefix例子

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user
          uri: http://localhost:9101
          predicates:
            - Path=/api/user/**
          filters:
            - StripPrefix=1  # 截断第一个路径/api
```

访问的路径为： http://localhost:10010/api/user/3

截断后的路径：http://localhost:10010/user/3

GlobalFilter（全局过滤器）

全局过滤器	作用
Forward Routing Filter	用于本地forward，也就是将请求在Gateway服务内进行转发，而不是转发到下游服务
LoadBalancerClient Filter	整合Ribbon实现负载均衡
Netty Routing Filter	使用Netty的 `HttpClient` 转发http、https请求
Netty Write Response Filter	将代理响应写回网关的客户端
RouteToRequestUrl Filter	将从request里获取的原始url转换成Gateway进行请求转发时所使用的url
Websocket Routing Filter	使用Spring Web Socket将转发 Websocket 请求
Gateway Metrics Filter	整合监控相关，提供监控指标

8、自定义全局过滤器

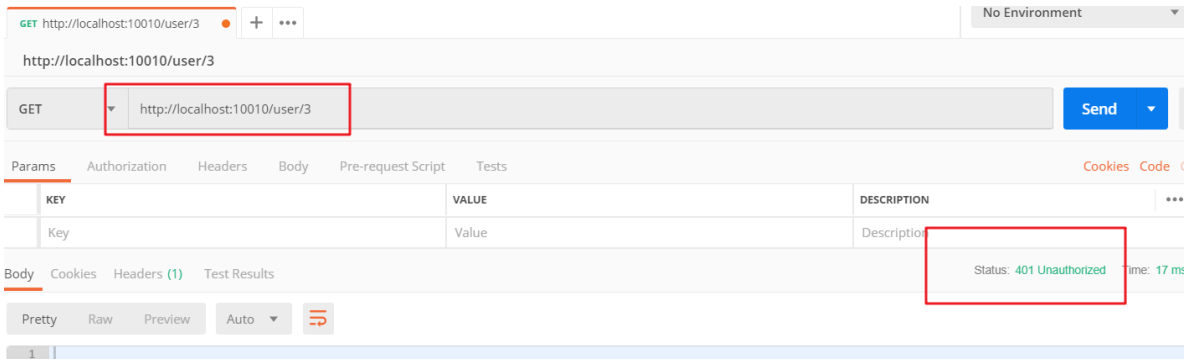
内置Global filter满足不了我们的需求，还可以自定义GlobalFilter

鉴权过滤器（PRE）

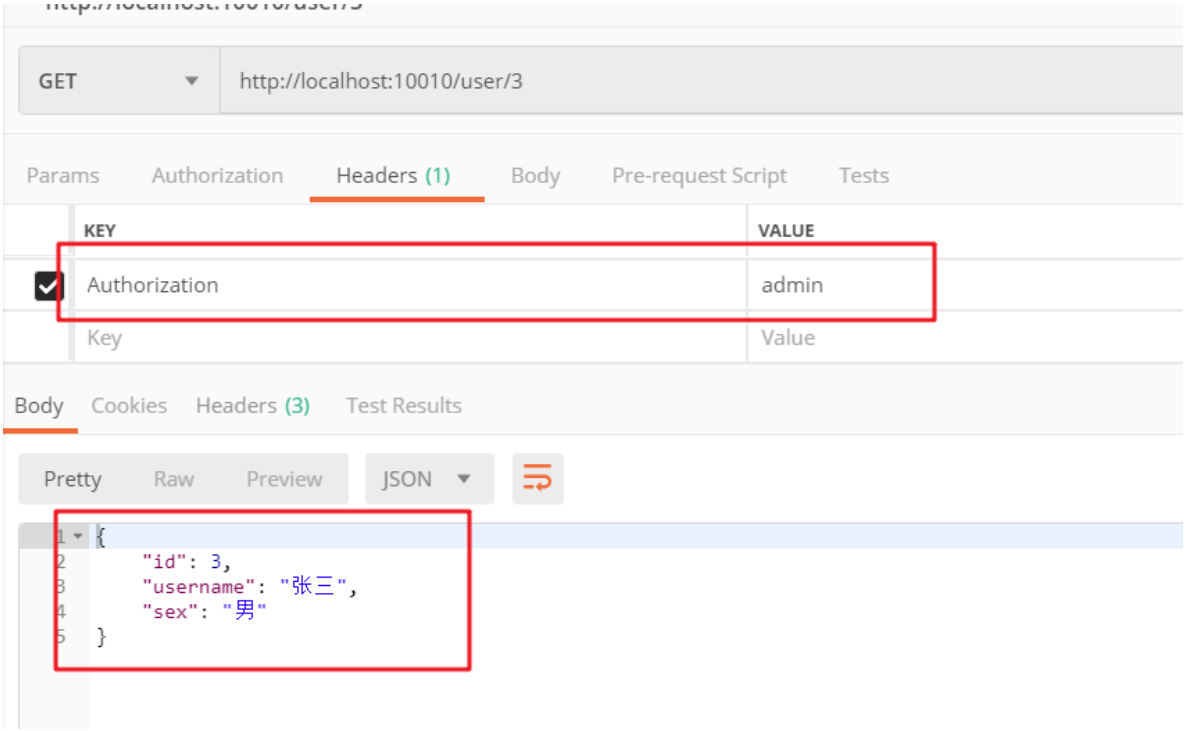
```java
/**
 * 统一鉴权过滤器
 */
@Component
@Order(1) // 数字越大，优先级越低
public class AuthorizationFilter implements GlobalFilter {
    @Override
```

```
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //获取请求与响应
        ServerHttpRequest request = exchange.getRequest();
        ServerHttpResponse response = exchange.getResponse();

        //获取请求头的Authorization请求头
        String authorization = request.getHeaders().getFirst("Authorization");

        if(StringUtils.isEmpty(authorization)){
            //返回401状态码
            response.setStatusCode(HttpStatus.UNAUTHORIZED);
            //结束请求
            return response.setComplete();
        }

        //校验合法性
        if(!"admin".equals(authorization)){
            //返回401状态码
            response.setStatusCode(HttpStatus.UNAUTHORIZED);
            //结束请求
            return response.setComplete();
        }

        //放行请求
        return chain.filter(exchange);
    }
}
```

请求头没有正确的Authorization



请求头携带正确的Authorization



统计接口IP执行时长（PRE+POST）

```
package cn.mf5.gateway.filter;

import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.annotation.Order;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

/**
 * 统计接口执行时长
 */
@Component
@Order(2)
public class ApiTimeStatisticsFilter implements GlobalFilter {
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        //请求执行之前
        Long startTime = System.currentTimeMillis();

        //放行请求，并在请求执行之后统计时间
        return chain.filter(exchange).then(Mono.fromRunnable(()->{
            //请求结束时间
            Long endTime = System.currentTimeMillis();
            //统计执行时长
            System.out.println(exchange.getRequest().getURI().getRawPath()+",
cost time:"+(endTime-startTime)+"ms");
        }));
    }
}
```

输出结果：

```
2022-10-16 21:53:03.836  INFO 17820 --- [
2022-10-16 21:53:03.871  INFO 17820 --- [
/user/3, cost time:54ms
```

允许指定IP访问（局部过滤器+带参数）

```
package cn.mf5.gateway.filter;

import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import
org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
import org.springframework.core.annotation.Order;
import org.springframework.http.HttpStatus;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.http.server.reactive.ServerHttpResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

import java.util.Arrays;
import java.util.List;

/**
 * IP禁用过滤器（只允许指定IP通过）
 */
@Component
@Order(3)
public class IPForbidGatewayFilterFactory extends
AbstractGatewayFilterFactory<IPForbidGatewayFilterFactory.Config> {
    public IPForbidGatewayFilterFactory() {
        super(Config.class);
    }

    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("permitIp"); //返回Config类中声明的属性集合
    }

    @Override
    public GatewayFilter apply(Config config) {
        return (exchange, chain) -> {
            //获取请求和响应
            ServerHttpRequest request = exchange.getRequest();
            ServerHttpResponse response = exchange.getResponse();
            //获取客户端IP地址
```

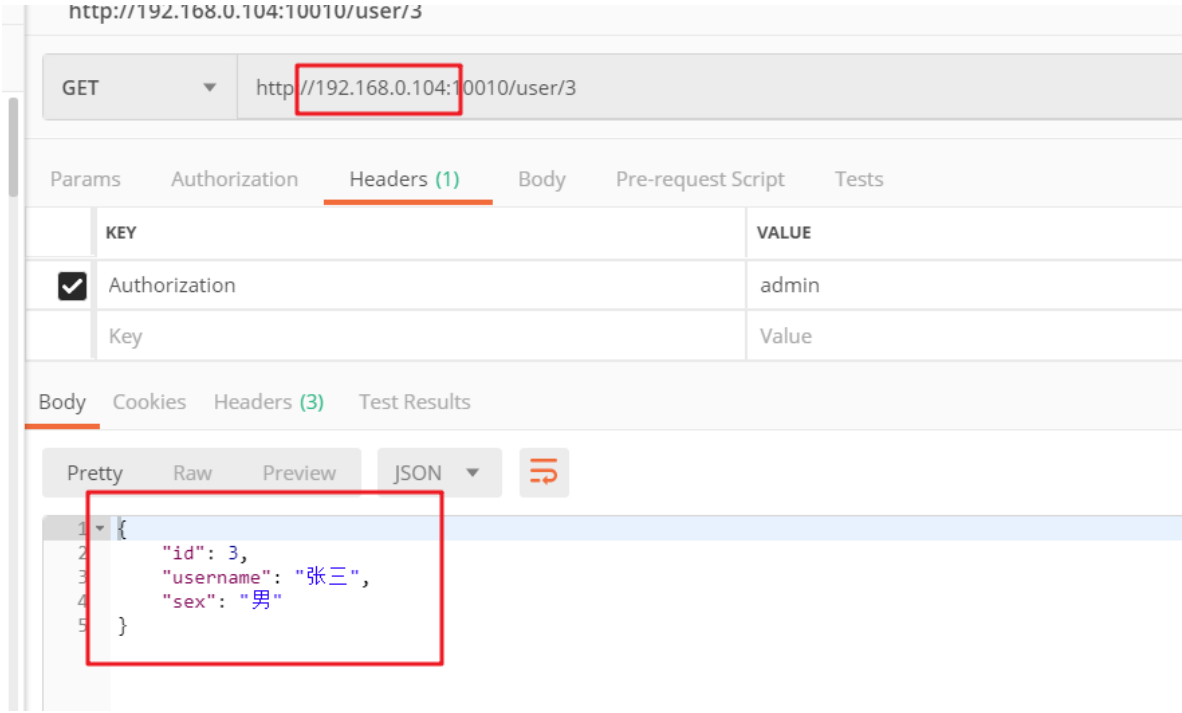
```
            String ip =
request.getRemoteAddress().getAddress().getHostAddress();
            System.out.println("Remote Address:"+ip);
            if(config.getPermitIp().equals(ip)){
                //请求放行
                return chain.filter(exchange);
            }else{
                //拒绝请求
                response.setStatusCode(HttpStatus.FORBIDDEN);
                //结束请求
                return response.setComplete();
            }
        };
    }

    static public class Config{  //该静态内部类用于接收外部配置传入的参数
        private String permitIp;

        public String getPermitIp() {
            return permitIp;
        }

        public void setPermitIp(String permitIp) {
            this.permitIp = permitIp;
        }
    }
}
```

配置:

```
spring:
  cloud:
    gateway:
      routes:
        - id: user                      # 路由 ID，唯一
          uri: http://localhost:9101    # 目标 URI，路由到微服务的地址
          predicates:                   # 请求转发判断条件
            - Path=/user/**             # 匹配对应 URL 的请求，将匹配到的请求追加在目标
URI 之后
          filters:
            - IPForbid=192.168.0.104
```

或者

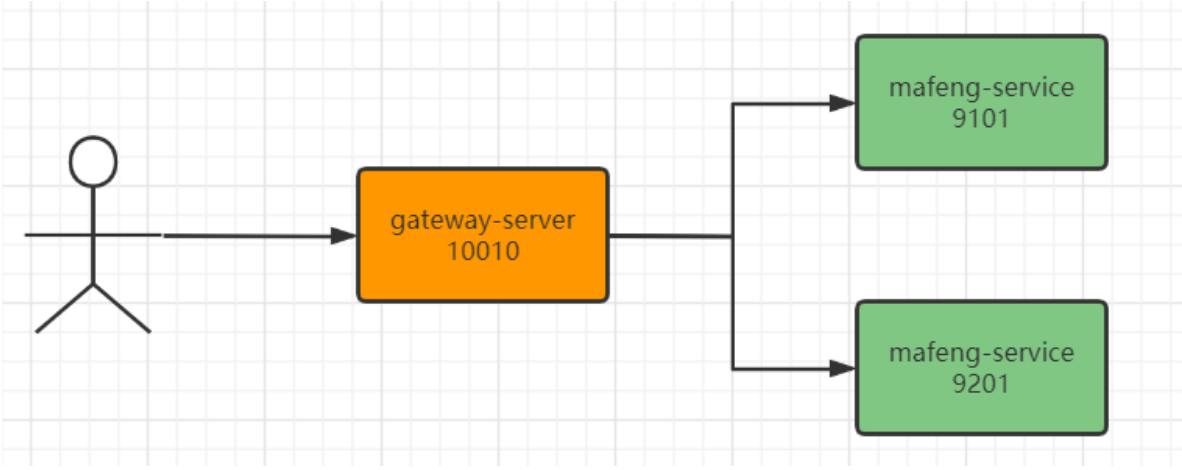
```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user                          # 路由 ID，唯一
          uri: http://localhost:9101        # 目标 URI，路由到微服务的地址
          predicates:                       # 请求转发判断条件
            - Path=/user/**                 # 匹配对应 URL 的请求，将匹配到的请求追加在目标
URI 之后
          filters:
            - name: IPForbid
              args:
                permitIp: 192.168.0.104
```



9、整合Nacos实现负载均衡

在之前的所有章节我们实现的例子中，路由规则的uri定义都是以http地址的形式写死的，如：`http://localhost:9101`,网关收到请求后根据路由规则将请求转发至对应的服务。

但是我们的微服务系统内通常都是一个服务启动多个实例，如下图所示：



为了达到网关接收到的请求能够负载均衡的转发给每个微服务的实例，我们将微服务网关注册到“服务注册中心”，比如：Nacos。这样：

- Gateway接收到请求后，首先从服务注册中心获取到各个微服务实例的访问地址
- 然后Gateway根据客户端负载均衡的规则，选择众多实例中的一个作为请求转发对象。

Gateway集成Nacos

导入nacos服务发现依赖

```
<dependency>
   <groupId>com.alibaba.cloud</groupId>
   <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

添加服务发现注解（可选）

```
@EnableDiscoveryClient
```

配置Nacos地址

```
spring:
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.66.133:7801
```

修改Gateway路由配置

```yaml
spring:
  cloud:
    gateway:
      routes:
        - id: user                      # 路由 ID，唯一
          uri: lb://mafeng-user  # 目标 URI，路由到微服务的地址
          predicates:                   # 请求转发判断条件
            - Path=/user/**             # 匹配对应 URL 的请求，将匹配到的请求追加在目标
URI 之后
```

`mafeng-user`是Nacos的serviceName， `lb`是LoadBalance的缩写

10、跨域访问配置

什么是跨域问题

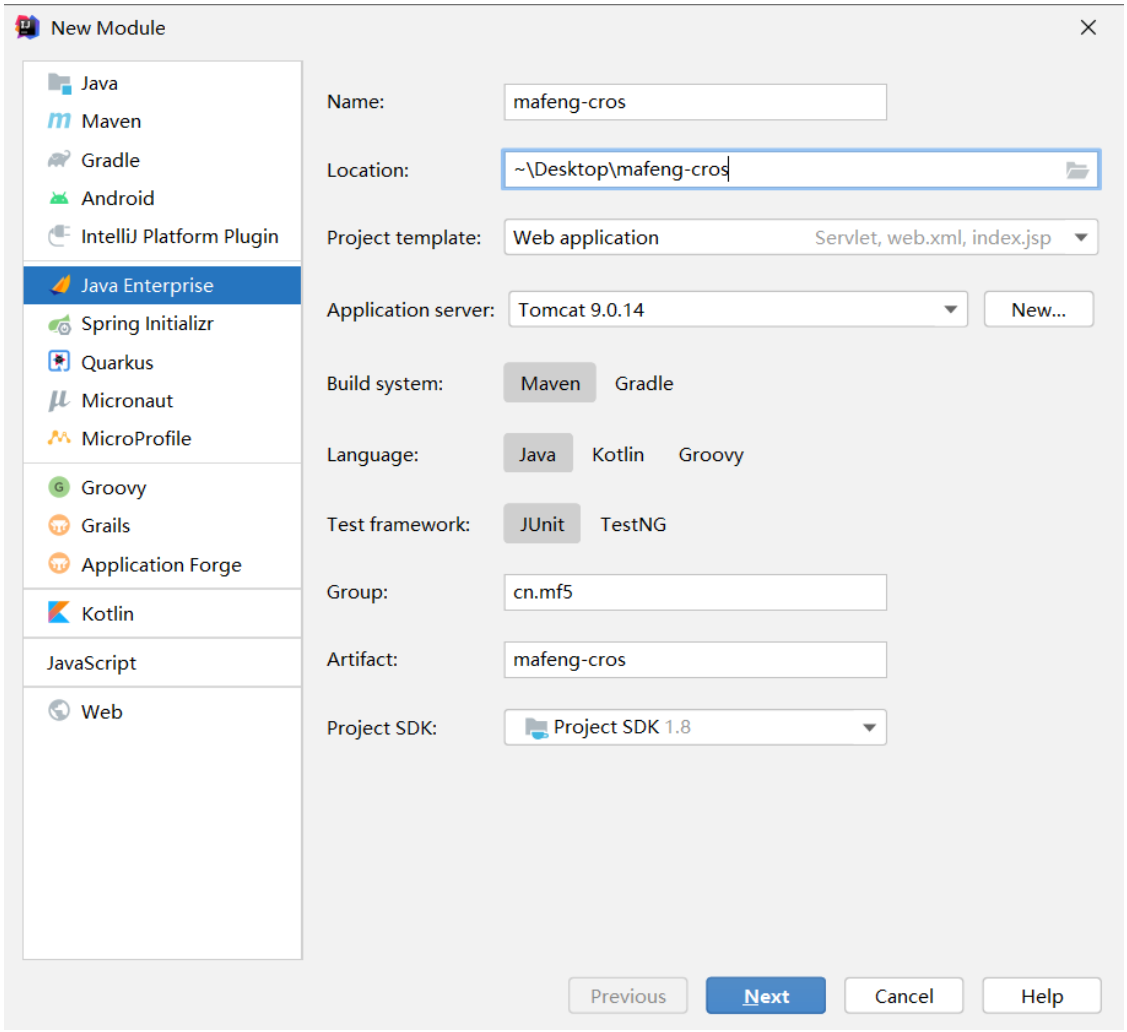
跨域：域名不一致就是跨域，主要包括：

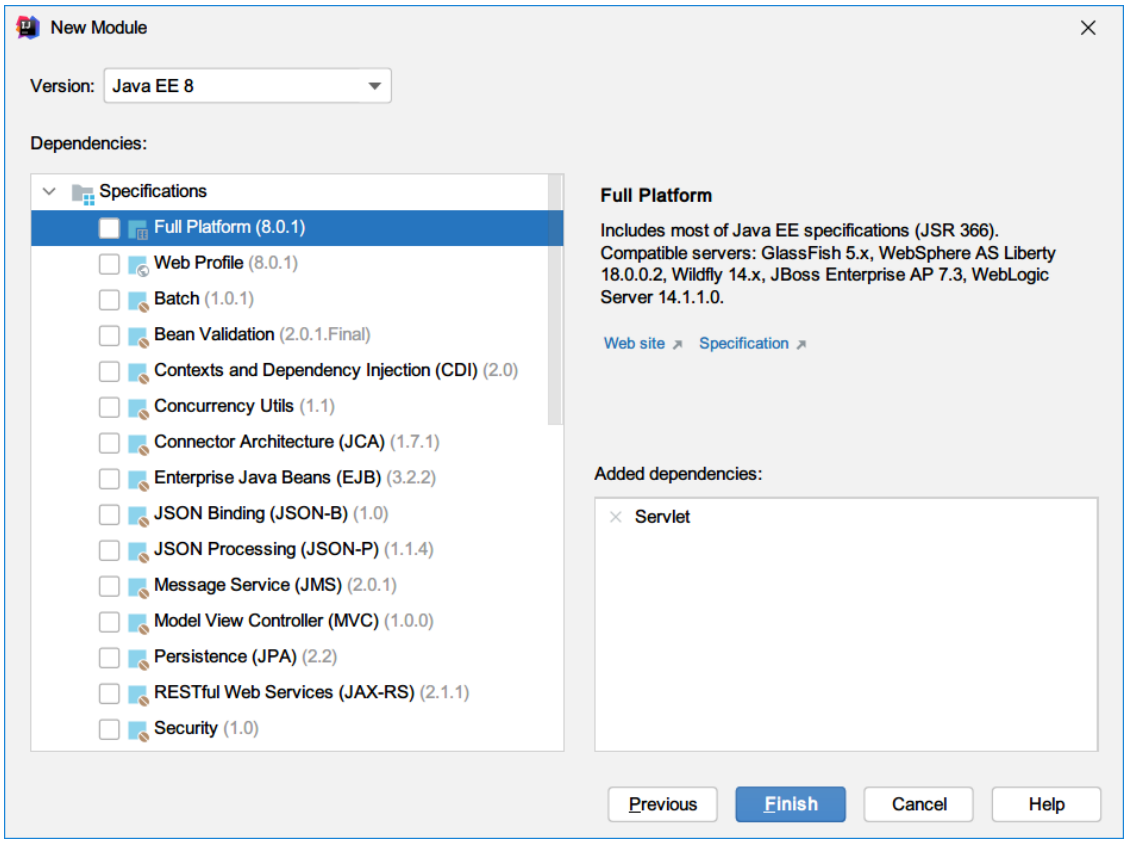
- 域名不同： www.taobao.com 和 www.jd.com
- 端口不同：localhost:8080和localhost:8081
- 协议不同：https://localhost:8080和http://localhost:8080

跨域问题：浏览器禁止请求的发起者与服务端发生跨域ajax请求，请求被浏览器拦截的问题

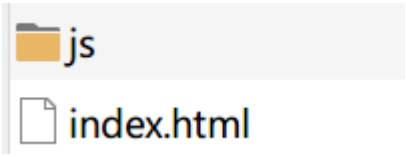
模拟跨域问题

1. 创建一个新的Java Enterprise工程

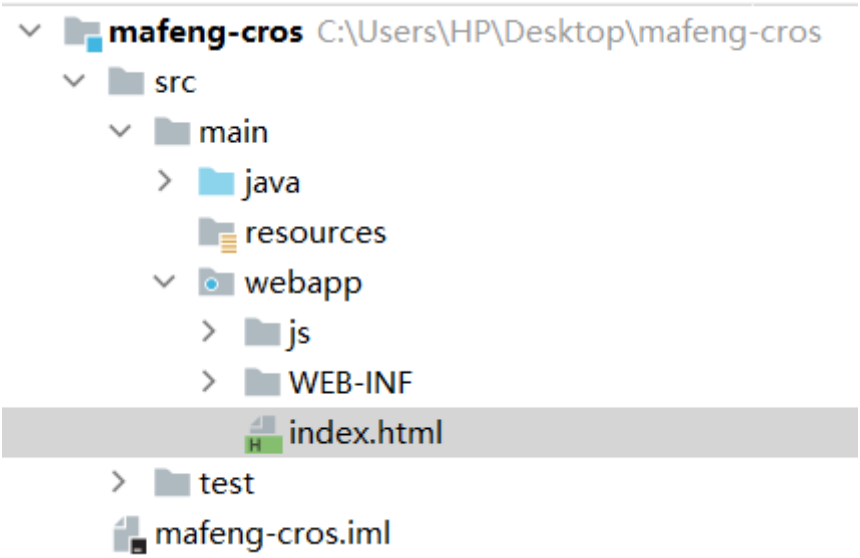




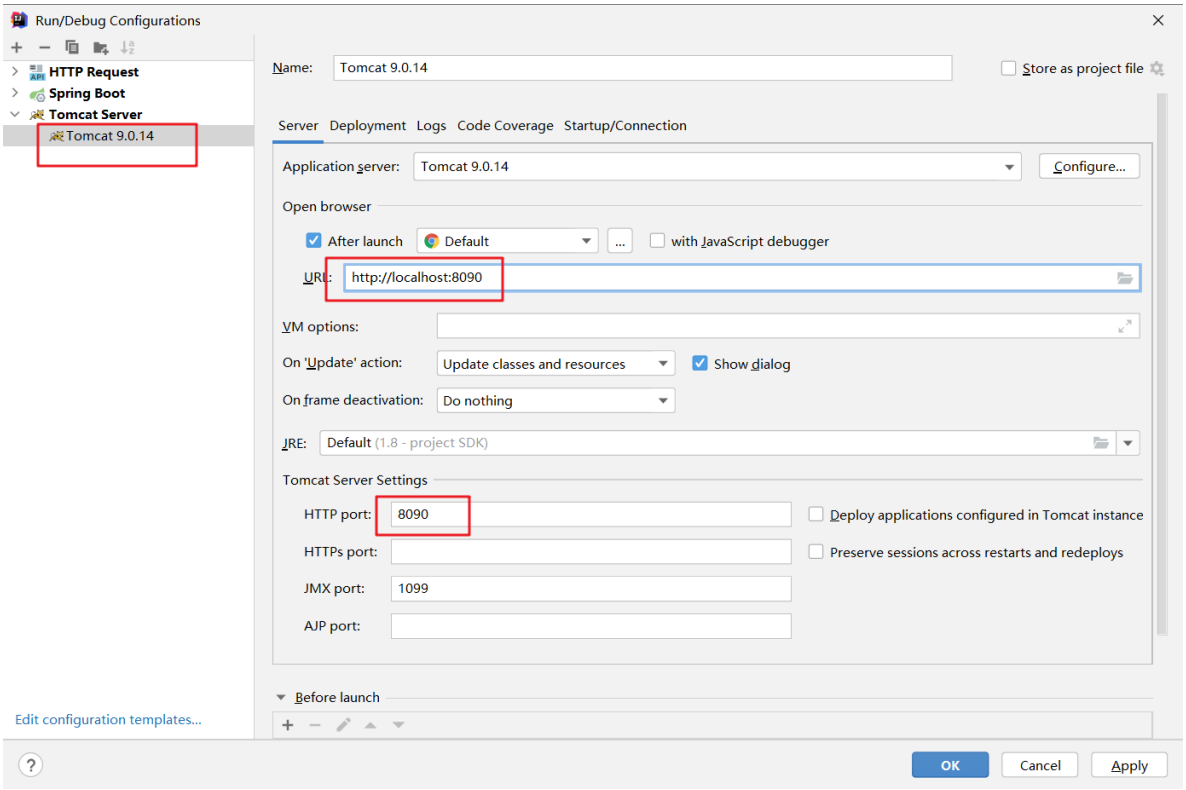
2. 找到素材页面文件，复制到webapp目录下



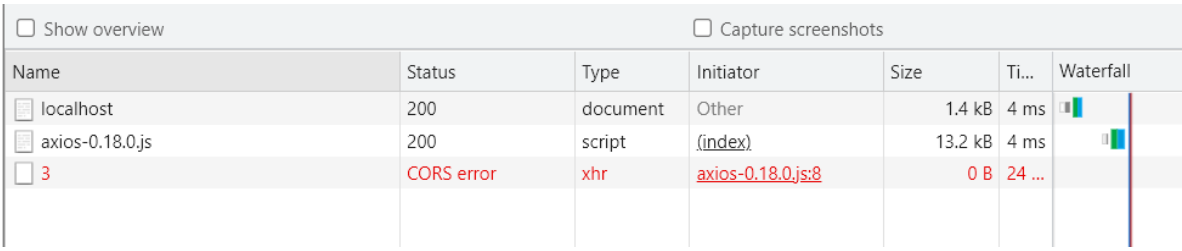
模块结构：



3. 部署到Tomcat服务器中，启动并访问，指定端口为8090



4. 可以在浏览器控制台看到下面的错误：



从localhost:8090访问localhost:10010，端口不同，显然是跨域的请求。

解决跨域问题

参数说明

因为所有的微服务都要经过网关，所以不需要每个微服务都去处理，在网关中处理就可以了。

CORS方案是浏览器向服务器询问是否允许本次请求跨域，这次询问是options请求，所以要让options请求通过。但如果每次跨域请求都询问，则会导致性能下降，于是设置一个有效期，在有效期内的请求不再每次询问，而是直接允许跨域请求。

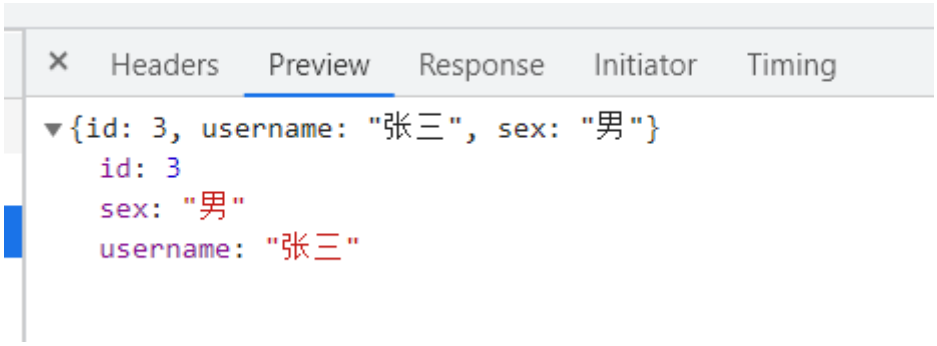
添加配置

1. 在gateway服务的application.yml文件中
2. globalcors元素在gateway的下一级
3. 注意端口号是8090，与上面的tomcat端口号要一致

```yaml
spring:
  cloud:
    gateway:
      # 全局的跨域处理
      globalcors:
        add-to-simple-url-handler-mapping: true # 解决options请求被拦截问题
        corsConfigurations:
          '[/**]':   # 哪些访问地址做跨域处理
            allowedOrigins: # 允许哪些网站的跨域请求
              - "http://localhost:8090"
            allowedMethods: # 允许的跨域ajax的请求方式
              - "GET"
              - "POST"
              - "DELETE"
              - "PUT"
              - "OPTIONS"
            allowedHeaders: "*" # 允许在请求中携带的头信息
            allowCredentials: true # 是否允许携带cookie
            maxAge: 360000  # 这次跨域检测的有效期
```

查看结果

配置完成以后查看浏览器的控制台



11、网关全局异常处理

之前的`AuthorizationFilter`在权限不足只是向前端返回401状态码而已，这样对前端并不友好！

接下来，我们利用网关的异常处理机制来进行统一的异常处理，统一返回json格式给前端。

自定义未授权异常类

```
@Data
public class UnAuthorizationException extends RuntimeException{
    private Integer code;
    public UnAuthorizationException(Integer code,String message){
        super(message);
        this.code= code;
    }
}
```

修改AuthorizationFilter类

```
    if(StringUtils.isEmpty(authorization)){
       /* //返回401状态码
        response.setStatusCode(HttpStatus.UNAUTHORIZED);
        //结束请求
        return response.setComplete();*/
        throw new UnAuthorizationException(401,"权限不足");
    }

    //校验合法性
    if(!"admin".equals(authorization)){
        //返回401状态码
      /*  response.setStatusCode(HttpStatus.UNAUTHORIZED);
        //结束请求
        return response.setComplete();*/
        throw new UnAuthorizationException(401,"权限不足");
    }
```

自定义全局网关异常处理

```
@Slf4j
@Order(-1)
@Component
@RequiredArgsConstructor
public class GatewayExceptionHandler implements ErrorWebExceptionHandler {
    private ObjectMapper objectMapper=new ObjectMapper();

    @Override
    public Mono<Void> handle(ServerWebExchange exchange, Throwable ex) {

        ServerHttpResponse response = exchange.getResponse();
        if (response.isCommitted()) {
            //对于已经committed(提交)的response，就不能再使用这个response向缓冲区写任何
东西
            return Mono.error(ex);
        }

        // header set 响应JSON类型数据，统一响应数据结构（适用于前后端分离JSON数据交换系
统）
        response.getHeaders().setContentType(MediaType.APPLICATION_JSON);

        // 处理异常
        String message;
        Integer code;
        if(ex instanceof UnAuthorizationException) {
            UnAuthorizationException e = (UnAuthorizationException)ex;
            code = e.getCode();
            message = e.getMessage();
        } else {
            code = 500;
            message = "服务器内部错误";
```

```
        }

        //全局通用响应数据结构，可以自定义。通常包含请求结果code、message、data
        AjaxResponse result = AjaxResponse.error(code,message);

        return response.writeWith(Mono.fromSupplier(() -> {
            DataBufferFactory bufferFactory = response.bufferFactory();
            byte[] bytes = null;
            try {
                bytes = objectMapper.writeValueAsBytes(result);
            } catch (JsonProcessingException e) {
                e.printStackTrace();
            }
            return bufferFactory.wrap(bytes);
        }));

    }
}
```

AjaxResponse类：

```
@Data
public class AjaxResponse<T> {
    private Integer code;
    private String message;
    private T data;

    public static AjaxResponse error(Integer code,String message){
        AjaxResponse ajaxResponse = new AjaxResponse();
        ajaxResponse.setCode(code);
        ajaxResponse.setMessage(message);
        return ajaxResponse;
    }

    public static AjaxResponse error(Integer code,String message,Object data){
        AjaxResponse ajaxResponse = new AjaxResponse();
        ajaxResponse.setCode(code);
        ajaxResponse.setMessage(message);
        ajaxResponse.setData(data);
        return ajaxResponse;
    }
}
```

效果图：


```
{
    "code":  401,
    "message":  "权限不足",
    "data":  null
}
```