

第9章 微服务架构基础

广东财经大学信息学院

罗 东 俊 博士

ZSUJONE@126.COM

(内部资料，请勿外传)



目的和要求

- 了解微服务的基本概念和常见的解决方案。
- 了解**Spring Cloud**和**Spring Cloud Alibaba**的基本组件及其功能。
- 掌握微服务注册、配置和调用原理和实践方法。
- 掌握微服务容错原理和实践方法。
- 掌握微服务网关原理和实践方法。
- 掌握微服务分布式事务原理和实践方法。
- 掌握微服务链路追踪原理和实践方法。



主要内容

- 9.1 微服务概述
- 9.2 微服务注册、配置和调用
- 9.3 微服务容错
- 9.4 微服务网关
- 9.5 微服务分布式事务
- 9.6 微服务链路追踪



9.4 微服务网关

■ 9.4.1 概述

■ 9.4.2 使用Gateway中间件

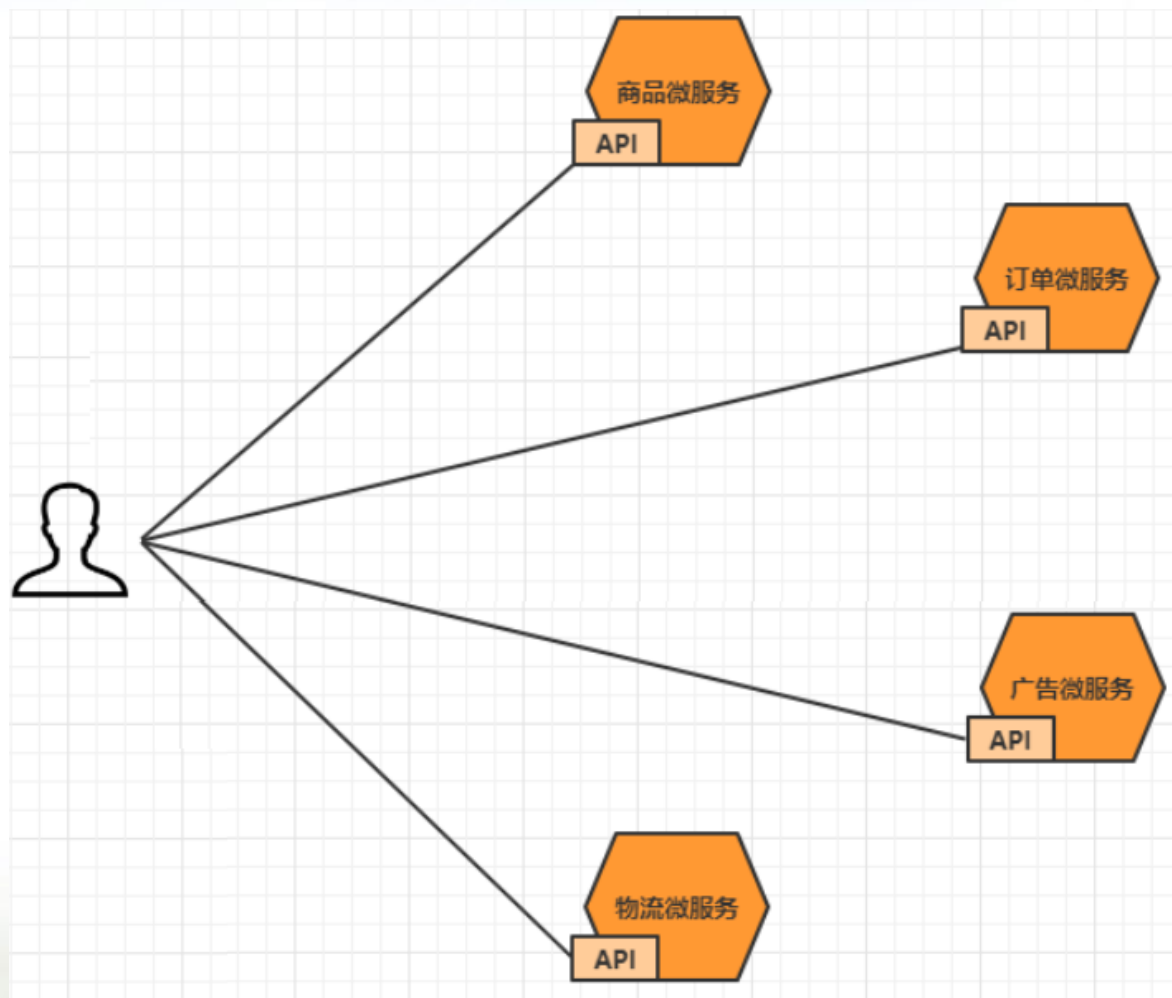


9.4.1 概述

- 1.网关简介
- 2.Gateway简介



引言



无网关的微服务架构存在的问题

■ 无网关微服务架构，会存在着诸多问题：

- ✧ 客户端多次请求不同的微服务，增加客户端代码或配置编写的复杂性。
- ✧ 认证复杂，每个服务都需要独立认证。
- ✧ 存在跨域请求，在一定场景下处理相对复杂。

■ 上面的这些问题可以借助**API**网关来解决。

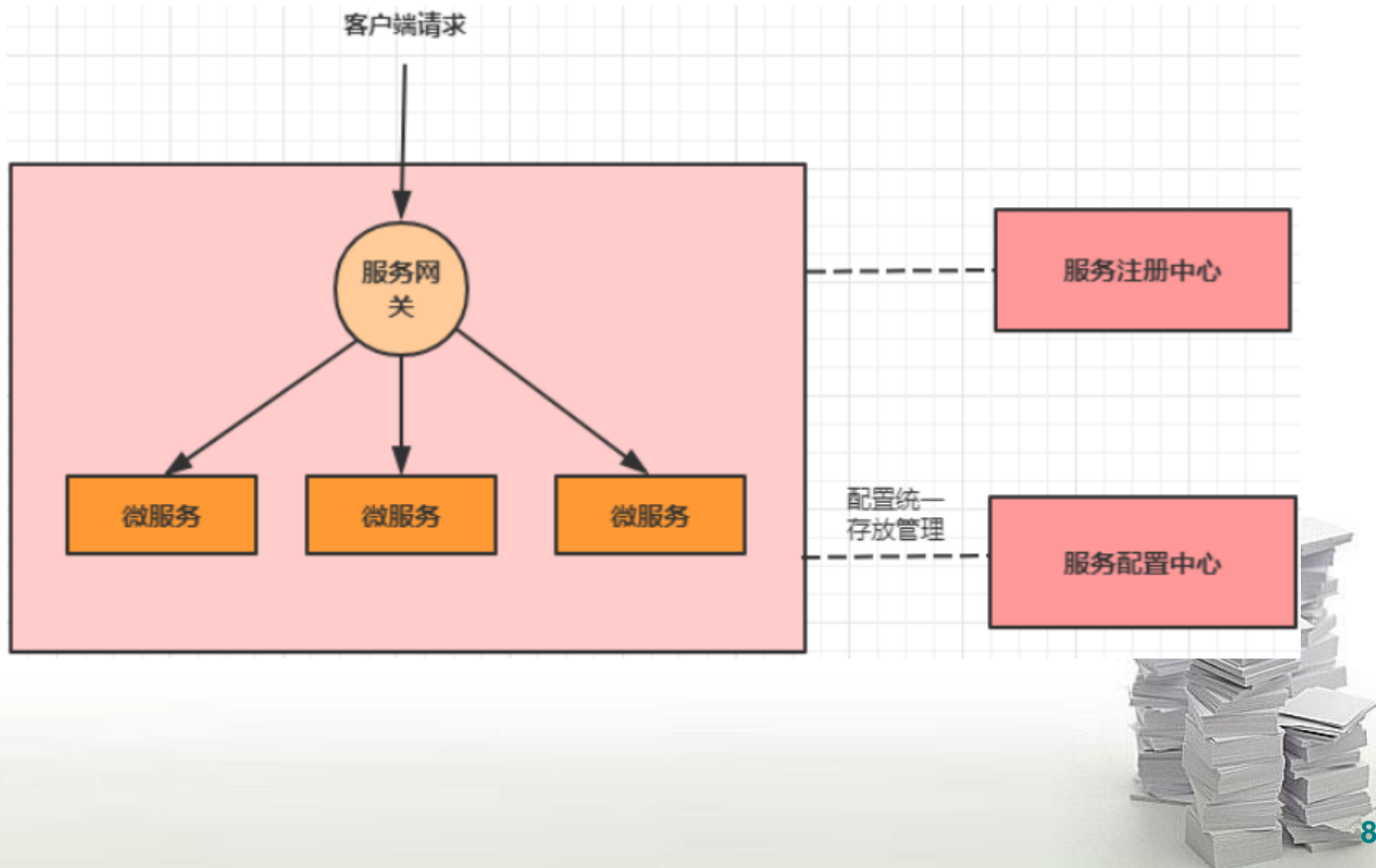


1.网关简介

- 所谓的**API**网关，就是指系统的统一入口，它封装了应用程序的内部结构，为客户端提供统一服务，一些与业务本身功能无关的公共逻辑可以在这里实现，诸如认证、鉴权、监控、路由转发、缓存、限流等等。



带网关的微服务架构



业界常见的网关

■ Ngnix+lua

- ✧使用**nginx**的反向代理和负载均衡可实现对**api**服务器的负载均衡及高可用
- ✧**lua**是一种脚本语言,可以用来编写一些简单的逻辑,**nginx**支持**lua**脚本

■ Kong

- ✧基于**Nginx+Lua**开发,性能高,稳定,有多个可用的插件(限流、鉴权等等)可以开箱即用。
- ✧问题: 只支持**Http**协议; 二次开发, 自由扩展困难; 提供管理**API**, 缺乏更易用的管控、配置方式。

业界常见的网关

■ Zuul

- ✧ **Netflix**开源的网关，功能丰富，使用**JAVA**开发，易于二次开发。
- ✧ 问题：缺乏管控，无法动态配置；依赖组件较多；处理**Http**请求依赖的是**Web**容器，性能不如**Nginx**

■ Spring Cloud Gateway

- ✧ **Spring**公司为了替换**Zuul**而开发的网关服务。

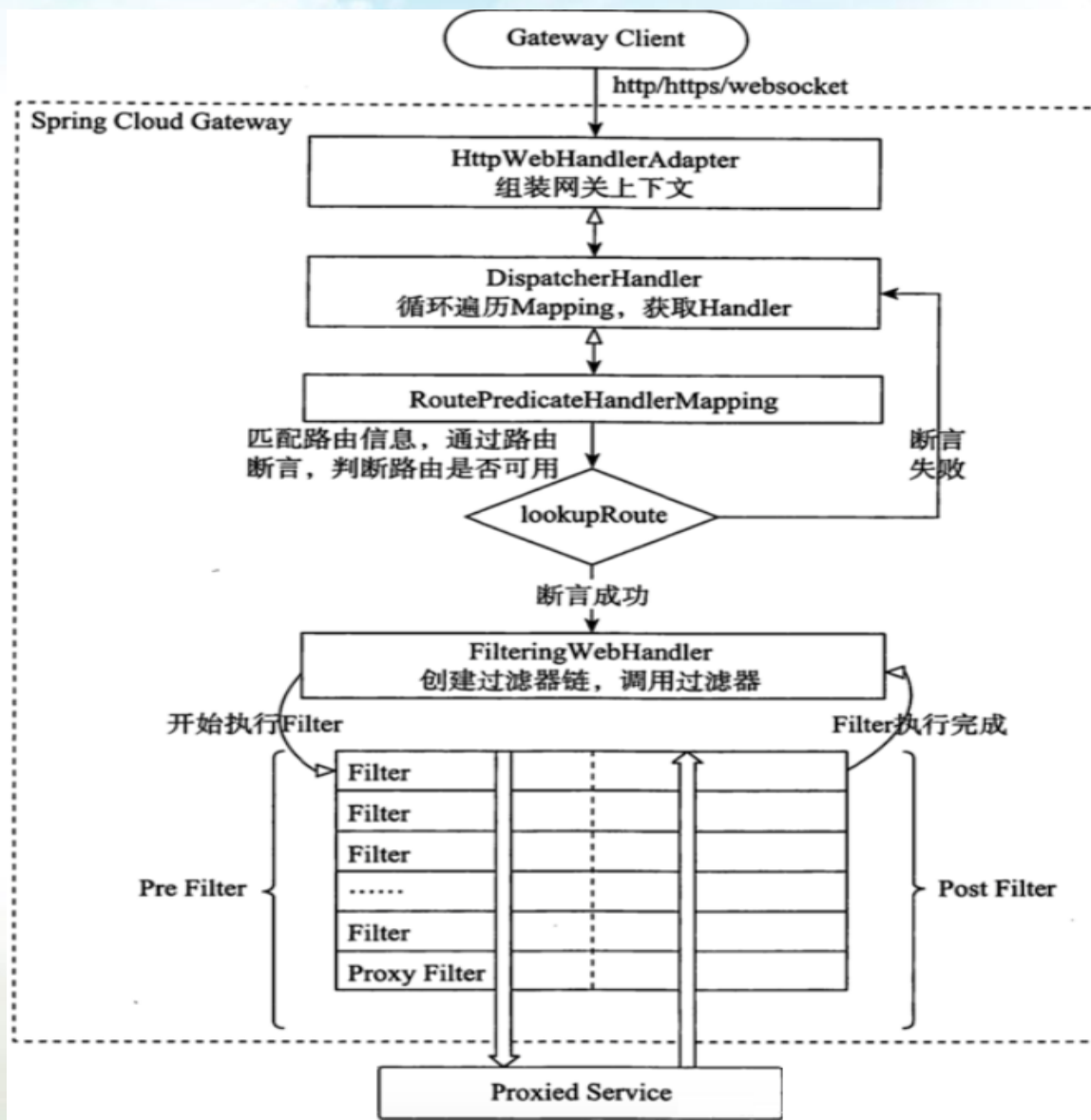


2. Gateway简介

- **Spring Cloud Gateway**旨在为微服务架构提供一种简单有效的统一的 **API路由**管理方式。
 - ◇ 其不仅提供统一的路由方式，并且基于 **Filter**链的方式提供了网关基本的功能，例如：安全，监控和限流。
- **优点：**
 - ◇ 性能强劲：是第一代网关**Zuul**的**1.6**倍。
 - ◇ 功能强大：内置了很多实用的功能，例如转发、监控、限流等
 - ◇ 设计优雅，容易扩展。
- **缺点：**
 - ◇ 其实现依赖**Netty**与**WebFlux**，不是传统的**Servlet**编程模型，学习成本高。
 - ◇ 不能将其部署在**Tomcat**、**Jetty**等**Servlet**容器里，只能打成**jar**包执行。
 - ◇ 需要**Spring Boot 2.0**及以上的版本，才支持。



1) Gateway执行流程



Gateway 执行流程

■ 执行流程大体如下：

- ✧ ① **Gateway Client**向**Gateway Server**发送请求；
- ✧ ② 请求首先会被**HttpWebHandlerAdapter**进行提取组装成网关上下文；
- ✧ ③ 然后网关的上下文会传递到**DispatcherHandler**，它负责将请求分发给**RoutePredicateHandlerMapping**；
- ✧ ④ **RoutePredicateHandlerMapping**负责路由查找，并根据路由断言判断路由是否可用；
- ✧ ⑤ 如果过断言成功，由**FilteringWebHandler**创建过滤器链并调用；
- ✧ ⑥ 请求会一次经过**PreFilter**—>微服务—>**PostFilter**的方法，最终返回响应。



2) Gateway中的核心概念

- ①路由
- ②断言
- ③过滤器



①路由

■ 路由（**Route**）是**Gateway**中最基本的概念之一，主要定义了以下信息：

- ✧ **id**：路由标识符，区别于其他 **Route**。
- ✧ **uri**：路由指向的目的地**uri**，即客户端请求最终被转发到的微服务。
- ✧ **order**：用于多个**Route**之间的优先级排序，数值越小，越优先匹配该路由。
- ✧ **predicates**：断言，作用是进行条件判断，只有断言都返回真，才会真正的执行路由。
- ✧ **filters**：过滤器，用于修改请求和响应信息。



例子

```
spring.cloud.gateway.routes[0].id=toOrderingService
spring.cloud.gateway.routes[0].uri=http://localhost:9010
spring.cloud.gateway.routes[0].order=0
spring.cloud.gateway.routes[0].predicates[0]=Path=/gateway/**
spring.cloud.gateway.routes[0].predicates[1]=Age=18,60
spring.cloud.gateway.routes[0].filters[0]=StripPrefix=1
spring.cloud.gateway.routes[0].filters[1]=AddRequestHeader=entry,gateway
spring.cloud.gateway.routes[0].filters[2]=Log=true,false
```



②断言

- 断言就是说：在什么条件下才能进行路由转发。
- **Spring Cloud Gateway**包括许多内置的断言工厂，所有这些断言都分别与**HTTP**请求的不同属性匹配。包括：
 - ✧ 基于**Datetime**类型的断言工厂
 - ✧ 基于远程地址的断言工厂
 - ✧ 基于**Cookie**的断言工厂
 - ✧ 基于**Header**的断言工厂
 - ✧ 基于**Host**的断言工厂
 - ✧ 基于**Method**请求方法的断言工厂
 - ✧ 基于**Path**请求路径的断言工厂
 - ✧ 基于**Query**请求参数的断言工厂
 - ✧ 基于路由权重的断言工厂



基于Datetime类型的断言工厂

■ 此类型的断言根据时间做判断，主要有三个：

✧ **AfterRoutePredicateFactory**: 接收一个日期参数，判断请求日期是否晚于指定日期

✧ **BeforeRoutePredicateFactory**: 接收一个日期参数，判断请求日期是否早于指定日期

✧ **BetweenRoutePredicateFactory**: 接收两个日期参数，判断请求日期是否在指定时间段内

■ 例如：

✧ **-After=2019-12-31T23:59:59.789+08:00[Asia/Shanghai]**



基于远程地址的断言工厂

■ **RemoteAddrRoutePredicateFactory**: 接收一个**IP**地址段，判断请求主机地址是否在地址段中。

■ 例如:

✧ **-RemoteAddr=192.168.1.1/24**



基于Cookie的断言工厂

- **CookieRoutePredicateFactory**: 接收两个参数: **cookie** 名字和一个正则表达式。判断请求**cookie**是否具有给定名称且值与正则表达式匹配。
- 例如:
 - ✧ **-Cookie=chocolate, ch.**



基于Header的断言工厂

■ **HeaderRoutePredicateFactory**: 接收两个参数: 标题名称和正则表达式。判断请求**Header**是否具有给定名称且值与正则表达式匹配。

■ 例如:

✧ **-Header=X-Request-Id, \d+**



基于Host的断言工厂

■ **HostRoutePredicateFactory**: 接收一个参数: 主机名模式。判断请求的**Host**是否满足匹配规则。

■ 例如:

✧ **-Host=**.testhost.org**



基于Method请求方法的断言工厂

■ **MethodRoutePredicateFactory**: 接收一个参数，判断请求类型是否跟指定的类型匹配。

■ 例如:

✧ **-Method=GET**



基于Path请求路径的断言工厂

- **PathRoutePredicateFactory**: 接收一个参数，判断请求的**URI**部分是否满足路径规则。
- 例如:
 - ✧ **-Path=/foo/{segment}**



基于Query请求参数的断言工厂

- **QueryRoutePredicateFactory** : 接收两个参数: 请求param和正则表达式, 判断请求参数是否具有给定名称且值与正则表达式匹配。
- 例如:
 - ✧ **-Query=baz, ba.**



基于路由权重的断言工厂

■ **WeightRoutePredicateFactory**: 接收两个参数: 组名和权重, 然后对于同一个组内的路由按照权重转发。

■ 例如:

- ✧ **-id: weight_route1**
- ✧ **-uri: host1**
- ✧ **-predicates:**
 - **-Path=/product/****
 - **-Weight=group3, 1**
- ✧ **-id: weight_route2**
- ✧ **-uri: host2**
- ✧ **-predicates:**
 - **-Path=/product/****
 - **-Weight= group3, 9**



③过滤器

■过滤器用于修改请求和响应信息，或进行统一验证。

■分类：

✧局部过滤器（**XxxGatewayFilter**）：作用在某一个路由上；

✧全局过滤器（**GlobalFilter**）：作用在全部路由上。

- 无需参数配置；
- 可以实现对权限的统一校验，安全性验证等功能。



内置局部过滤器

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
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	-
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

内置局部过滤器

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	-
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
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



内置全局过滤器



3) 网关限流

- 网关是所有请求的公共入口，所以可以在网关进行限流。
- **Sentinel**支持对**SpringCloud Gateway**、**Zuul**等主流网关进行限流。



Gateway限流依赖

```
<!--sentinel网关流控依赖-->
```

```
<dependency>
```

```
    <groupId>com.alibaba.cloud</groupId>
```

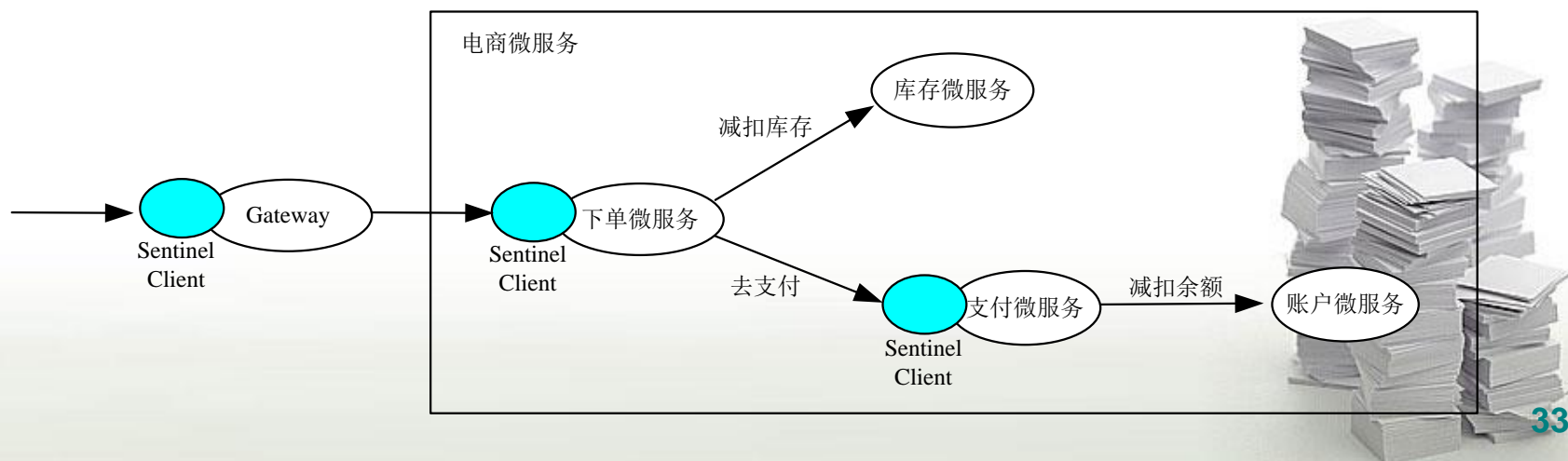
```
    <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
```

```
</dependency>
```



9.4.2 使用Gateway中间件

- 本小节以**Spring Cloud Gateway**为网关解决方案，为本章节前面搭建的电商项目微服务架构搭建高可用的网关环境，实现网关断言、过滤、限流、跨域等功能。
- 带网关的电商项目微服务架构：



实践内容

- 1.创建、注册并配置网关微服务
- 2.自定义断言
- 3.自定义过滤器
- 4.使用网关限流
- 5.网关跨域配置
- 6.搭建高可用网关环境



1.创建、注册并配置网关微服务

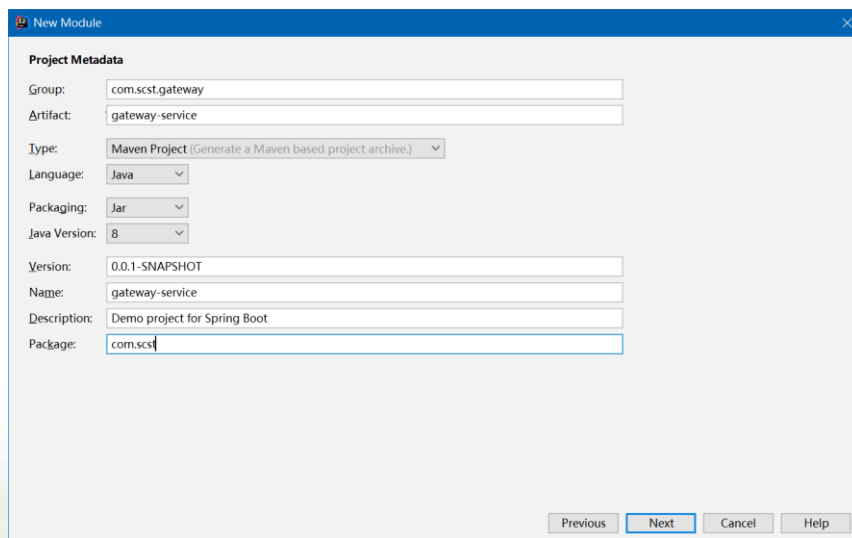
■搭建步骤:

- ✧1) 创建网关微服务模块
- ✧2) 引入gateway依赖
- ✧3) 添加微服务配置



1) 创建网关微服务模块

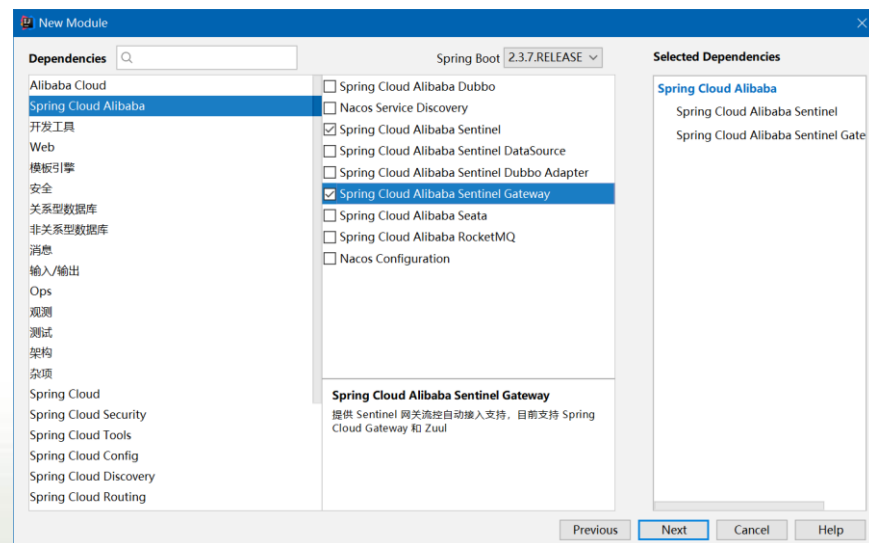
- 在chapter09项目中，使用Spring Initializr方式创建一个Spring Boot模块gateway-service，在Dependencies选择Sentinel依赖，但不能选择Spring Web依赖。



The 'New Module' dialog box, Project Metadata tab, is shown. The fields are filled as follows:

- Group: com.scst.gateway
- Artifact: gateway-service
- Type: Maven Project (generate a Maven based project archive.)
- Language: Java
- Packaging: Jar
- Java Version: 8
- Version: 0.0.1-SNAPSHOT
- Name: gateway-service
- Description: Demo project for Spring Boot
- Package: com.scst

Buttons at the bottom: Previous, Next, Cancel, Help.



The 'New Module' dialog box, Dependencies tab, is shown. The 'Spring Boot' version is 2.3.7.RELEASE. The 'Dependencies' list on the left includes:

- Alibaba Cloud
- Spring Cloud Alibaba
- 开发工具
- Web
- 模板引擎
- 安全
- 关系型数据库
- 非关系型数据库
- 消息
- 输入/输出
- Ops
- 观测
- 测试
- 架构
- 杂项
- Spring Cloud
- Spring Cloud Security
- Spring Cloud Tools
- Spring Cloud Config
- Spring Cloud Discovery
- Spring Cloud Routing

The 'Selected Dependencies' list on the right includes:

- Spring Cloud Alibaba
- Spring Cloud Alibaba Sentinel
- Spring Cloud Alibaba Sentinel Gateway

Buttons at the bottom: Previous, Next, Cancel, Help.

建立父项目依赖

- 在**gateway-service**模块的**pom.xml**文件中引入对父项目**chapter09**的依赖，示例代码如下。

```
<parent>
  <artifactId>chapter09</artifactId>
  <groupId>com.scst</groupId>
  <version>0.0.1-SNAPSHOT</version>
</parent>
```



2) 引入gateway依赖

- 在chapter09工程的gateway-service模块pom.xml中引入gateway依赖:

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



3) 添加微服务配置

- ①服务注册配置
- ②路由配置



①服务注册配置

■ 在gateway-service模块的 application.properties中添加以下配置:

应用名称

spring.application.name=gateway-service

应用服务 WEB 访问端口; 如果不存在JVM参数port, 则默认使用9000

server.port=\${port:9000}

#配置Nacos地址

spring.cloud.nacos.discovery.server-addr=localhost:8848

#让gateway可以发现nacos中的微服务

spring.cloud.gateway.discovery.locator.enabled=true

②路由配置

- 当gateway可以发现nacos中的微服务后，默认就可按照网关地址/微服务名称/接口的格式去访问微服务了。

✧如： <http://localhost:9000/ordering-service/ordering/purchase/p002/a001>

- 但也可在模块的application.properties中或Nacos中自定义路由。



自定义路由

■ 在gateway-service模块的 application.properties中添加以下配置:

#配置一条路由信息

```
spring.cloud.gateway.routes[0].id=toOrderingService
```

```
spring.cloud.gateway.routes[0].uri=lb://ordering-service
```

```
spring.cloud.gateway.routes[0].order=0
```

```
spring.cloud.gateway.routes[0].predicates[0]=Path=/gateway/**
```

```
spring.cloud.gateway.routes[0].filters[0]=StripPrefix=1
```

```
spring.cloud.gateway.routes[0].filters[1]=AddRequestHeader=entry,gateway
```



自定义路由

■ 配置一条路由信息：

- ✧ **id**: 本条路由标识，要求唯一，默认是**UUID**
- ✧ **uri**: 请求要转发到的**URL**地址，或者是请求要负载均衡地转发到的微服务名，如**lb://ordering-service**
- ✧ **order**: 本条路由的优先级,数字越小级别越高
- ✧ **predicates**: 断言，即路由转发要满足的每个条件
- ✧ **Path**: 定义**Path**规则，默认是**Path=/[请求要转发到的微服务name]/****；如果采用默认**Path**，则其他自定义配置均无效。
- ✧ **filters**: 过滤器，方便对请求在传递过程中做一定的修改
- ✧ **StripPrefix**: 定义转发之前去掉几层路径
 - 过滤器作用前: **http://localhost:9000/gateway/ordering/configs**
转向**http://localhost:9010/gateway/ordering/configs**
 - 过滤器作用后: **http://localhost:9000/gateway/ordering/configs**
转向**http://localhost:9010/ordering/configs**
- ✧ **AddRequestHeader**: 增加一个请求头



2.自定义断言

■ 设定一个场景：假设我们的应用仅仅让**age**在**(min,max)**之间的人来访问，则自定义断言步骤如下：

- ✧1) 添加**Age**断言配置
- ✧2) 自定义**Age**断言工厂
- ✧3) 启动测试



1) 添加Age断言配置

- 在gateway-service模块的application.properties中添加以下配置:

```
# 限制年龄只有在18到60岁之间的人能访问  
spring.cloud.gateway.routes[0].predicates[1]=Age=18,60
```



2) 自定义Age断言工厂

- 在gateway-service模块中新建一个断言工厂类AgeRoutePredicateFactory，实现从配置文件中接收配置信息和断言逻辑。
- 自定义路由断言工厂类，要求：
 - ✧ 类名必须是：配置名+RoutePredicateFactory
 - ✧ 必须继承AbstractRoutePredicateFactory<配置值接收类>



AgeRoutePredicateFactory类

```
package com.scst.predicates;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.apache.commons.lang3.StringUtils;
import org.springframework.cloud.gateway.handler.predicate.AbstractRoutePredicateFactory;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import java.util.Arrays;
import java.util.List;
import java.util.function.Predicate;

@Component

public class AgeRoutePredicateFactory extends AbstractRoutePredicateFactory<AgeRoutePredicateFactory.Config> {
```



AgeRoutePredicateFactory类

//配置值接收类，用于接收配置文件中对应参数值

@Data

@NoArgsConstructor

public static class Config {

private int minAge;

private int maxAge;

}

//构造函数

public AgeRoutePredicateFactory() {

super(Config.class);

}

//读取配置文件的中参数值，并将其赋值到配置值接收类的属性上

@Override

public List<String> shortcutFieldOrder() {

//这里参数名顺序必须跟配置文件中值的顺序对应

return Arrays.asList("minAge", "maxAge");

}

AgeRoutePredicateFactory类

//断言逻辑

@Override

```
public Predicate<ServerWebExchange> apply(Config config) {  
    return new Predicate<ServerWebExchange>() {  
        @Override  
        public boolean test(ServerWebExchange serverWebExchange) {  
            //1 接收访问路径中的age参数  
            String ageStr = serverWebExchange.getRequest().getQueryParams().getFirst("age");  
            if (StringUtils.isEmpty(ageStr)) {        //2 先判断是否为空  
                int age = Integer.parseInt(ageStr);    //3 如果不为空,再进行路由逻辑判断  
                if (age <= config.getMaxAge() && age >= config.getMinAge()) {  
                    return true;  
                } else { return false;}  
            }  
            return false;  
        }  
    };  
}
```


3) 启动测试

■ 测试发现当age在[18,60]可以访问，其它范围不能访问。

✧ <http://localhost:9000/gateway/ordering/purchase/p002/a005?age=18>

✧ <http://localhost:9000/gateway/ordering/purchase/p002/a005?age=60>



3.自定义过滤器

- 1) 自定义局部过滤器
- 2) 自定义全局过滤器



1) 自定义局部过滤器

■ 设定一个场景：针对某路由的访问，控制日志是否开启，则自定义局部过滤器步骤如下：

- ✧①添加**Log**过滤器配置
- ✧②自定义**Log**过滤器工厂
- ✧③启动测试



①添加Log过滤器配置

■ 在gateway-service模块的
application.properties中添加以下配置:

控制日志是否开启

```
spring.cloud.gateway.routes[0].filters[2]=Log=true,false
```



②自定义Log过滤器工厂

- 在gateway-service模块中新建一个过滤器工厂类LogGatewayFilterFactory，实现从配置文件中接收配置信息和过滤器逻辑。
- 自定义局部过滤器工厂类，要求：
 - ✧ 类名必须是：配置名+GatewayFilterFactory
 - ✧ 必须继承AbstractGatewayFilterFactory<配置值接收类>



LogGatewayFilterFactory类

```
package com.scst.filters;

import lombok.Data;
import lombok.NoArgsConstructor;
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.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;
import java.util.Arrays;
import java.util.List;

@Component
public class LogGatewayFilterFactory extends AbstractGatewayFilterFactory<LogGatewayFilterFactory.Config> {
```



LogGatewayFilterFactory类

//配置值接收类,用于接收配置文件中对应参数值

@Data

@NoArgsConstructor

public static class Config {

private boolean consoleLog;

private boolean cacheLog;

}

//构造函数

public LogGatewayFilterFactory() {

super(Config.class);

}

//读取配置文件的中参数值，并将其赋值到配置值接收类的属性上

@Override

public List<String> shortcutFieldOrder() {

//这里参数名顺序必须跟配置文件中值的顺序对应

return Arrays.asList("consoleLog", "cacheLog");

}

LogGatewayFilterFactory类

//过滤器逻辑

@Override

public GatewayFilter **apply**(Config config) {

 return new **GatewayFilter**() {

 @Override

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

 if (config.isCacheLog()) {

 System.out.println("cacheLog已经开启了....");

 }

 if (config.isConsoleLog()) {

 System.out.println("consoleLog已经开启了....");

 }

 return chain.filter(exchange); //调用chain.filter继续向下游执行

 }

 };

}

}

③启动测试

- 在浏览器中访问：

`http://localhost:9000/gateway/ordering/purchase/p002/a005?age=18`

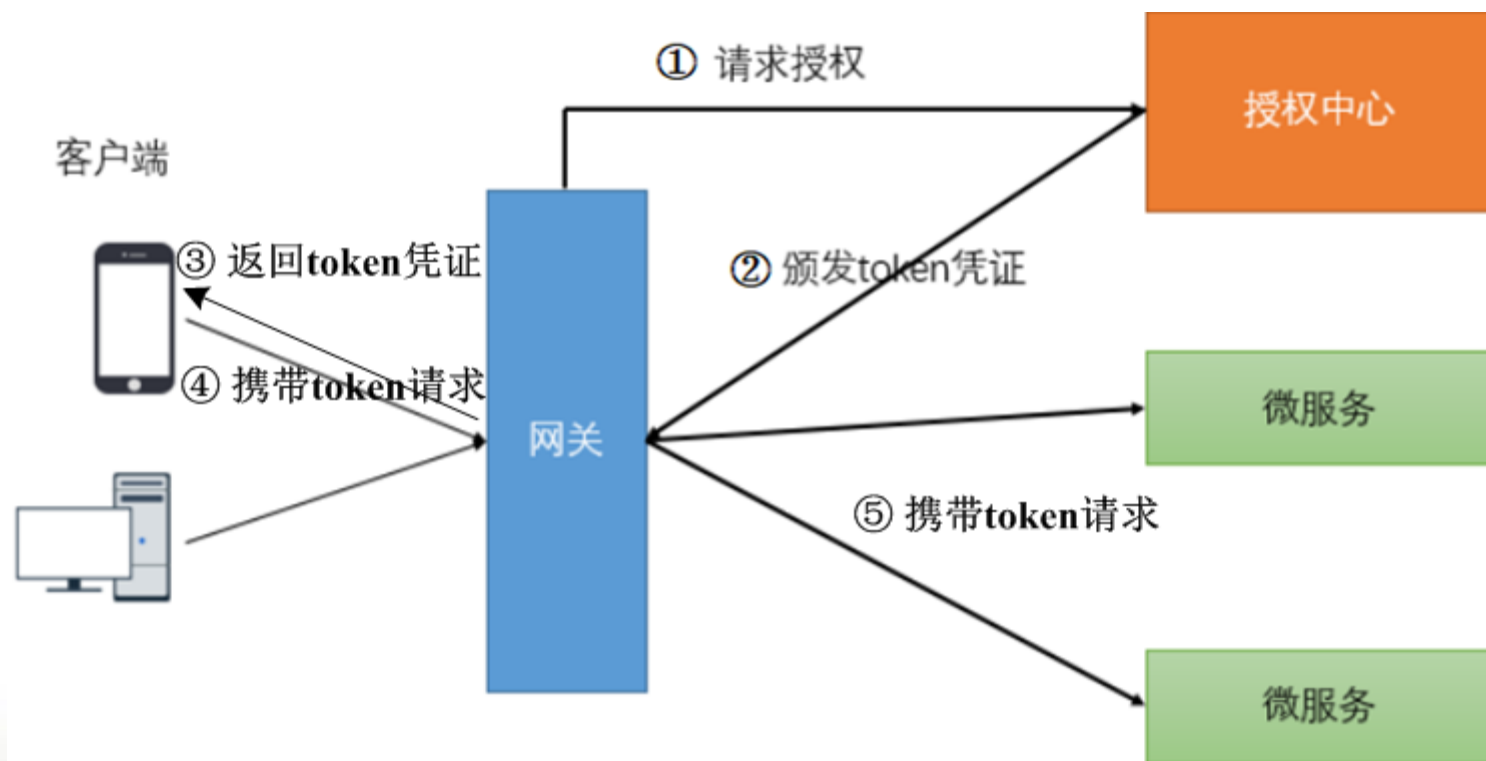
- 测试发现控制台输出：

✧ `consoleLog`已经开启了....



2) 自定义全局过滤器

■ 设定一个场景：在网关进行统一权限校验。



实现步骤

■ 实现步骤如下：

- ✧①自定义认证过滤器工厂
- ✧②实现**token**转发
- ✧③启动测试



①自定义认证过滤器工厂

- 在gateway-service模块中新建一个认证过滤器工厂类AuthGlobalFilter，要求实现GlobalFilter和Ordered接口。



AuthGlobalFilter类

```
package com.scst.filters;

import com.alibaba.fastjson.JSONArray;
import org.apache.commons.lang3.StringUtils;
import org.springframework.cloud.gateway.filter.GatewayFilterChain;
import org.springframework.cloud.gateway.filter.GlobalFilter;
import org.springframework.core.Ordered;
import org.springframework.http.HttpStatus;
import org.springframework.http.server.reactive.ServerHttpRequest;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;

//自定义全局过滤器， 需要实现GlobalFilter和Ordered接口
@Component
public class AuthGlobalFilter implements GlobalFilter, Ordered {
```



AuthGlobalFilter类

//完成判断逻辑

@Override

```
public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {  
    String token = exchange.getRequest().getQueryParams().getFirst("token");  
    if (!StringUtils.equals(token, "admin")) {  
        System.out.println("鉴权失败");  
        exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);  
        return exchange.getResponse().setComplete();  
    }  
    //添加token请求头  
    ServerHttpRequest request = exchange.getRequest().mutate()  
        .header("token", "admin", "123456").build();  
    //调用chain.filter继续向下游执行  
    return chain.filter(exchange.mutate().request(request).build());  
}
```

AuthGlobalFilter类

//顺序,数值越小,优先级越高

@Override

```
public int getOrder() {
```

```
    return 0;
```

```
}
```

```
}
```



②实现token转发

- 案例中，当用户访问：
`http://localhost:9000/gateway/ordering/purchase/p002/a005?age=18&&token=admin`时，**token**会由网关转发到**ordering-service**，然后又经**PaymentFeignClient**转发到**payment-service**，故在**ordering-service**模块中需创建一个请求头转发类**FeignRequestInterceptor**，并要求实现**RequestInterceptor**接口，然后配置**FeignClient**。



FeignRequestInterceptor类

```
package com.scst.feignclient;

import feign.RequestInterceptor;
import feign.RequestTemplate;
import org.springframework.stereotype.Component;
import org.springframework.web.context.request.RequestContextHolder;
import org.springframework.web.context.request.ServletRequestAttributes;
import javax.servlet.http.HttpServletRequest;
import java.util.*;

@Component
public class FeignRequestInterceptor implements RequestInterceptor {
```



FeignRequestInterceptor类

@Override

```
public void apply(RequestTemplate requestTemplate) {  
    ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHolder.getRequestAttributes();  
    HttpServletRequest request = attributes.getRequest();  
    Enumeration<String> headerNames = request.getHeaderNames();  
    if (headerNames != null) {  
        while (headerNames.hasMoreElements()) {  
            String name = headerNames.nextElement();  
            Enumeration<String> values = request.getHeaders(name);  
            List<String> list = new ArrayList<>();  
            while (values.hasMoreElements()) { list.add(values.nextElement()); }  
            requestTemplate.header(name, list); //转发请求头  
        }  
    }  
}
```

配置FeignClient

```
package com.scst.feignclient;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

//声明Feign客户端,其中value用于指定被调用的微服务, fallback或fallbackFactory用于指定容错类,
//configuration用于指定配置类
@FeignClient(value = "payment-service", fallbackFactory = PaymentFallbackFactory.class, configuration =
FeignRequestInterceptor.class)

public interface PaymentFeignClient {

    //调用pay方法:http://[payment-service address]/payment/pay/{id}
    @GetMapping("/payment/pay/{id}")
    public String pay(@PathVariable("id") String accountId);
}
```

③启动测试

- 为了验证token是否被转发到了payment-service微服务，在payment-service模块中创建一个新的来源分析类DefaultRequestOriginParser，来替代旧的来源分析类MyRequestOriginParser。
- 当用户访问：
`http://localhost:9000/gateway/ordering/purchase/p002/a005?age=18&&token=admin`时，测试发现payment-service模块控制台输出：

请求来源: gateway

Token: ["admin", "123456"]

DefaultRequestOriginParser类

```
package com.scst.parser;

import com.alibaba.csp.sentinel.adapter.spring.webmvc.callback.RequestOriginParser;
import com.alibaba.fastjson.JSONArray;
import org.springframework.stereotype.Component;
import javax.servlet.http.HttpServletRequest;
import java.util.Enumeraation;

//来源解析类
@Component
public class DefaultRequestOriginParser implements RequestOriginParser {

    @Override
    public String parseOrigin(HttpServletRequest request) {
        //来源origin必须在sentinel授权配置的白名单或黑名单里才可对其进行访问控制
        String origin = request.getHeader("entry");
        System.out.println("请求来源: "+origin);
        Enumeraation<String> secrets = request.getHeaders("token");
        System.out.println("Token: "+JSONArray.toJSONString(secrets));
        return origin;
    }
}
```

4.使用网关限流

■搭建步骤:

- ✧1) 引入**Sentinel**依赖
- ✧2) 添加**Sentinel**配置
- ✧3) 设置限流规则
- ✧4) 配置异常处理



1) 引入Sentinel依赖

■ 使用网关限流，需引入以下依赖：

```
<!--sentinel客户端依赖-->
```

```
<dependency>
```

```
  <groupId>com.alibaba.cloud</groupId>
```

```
  <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
```

```
</dependency>
```

```
<!--sentinel网关流控依赖-->
```

```
<dependency>
```

```
  <groupId>com.alibaba.cloud</groupId>
```

```
  <artifactId>spring-cloud-alibaba-sentinel-gateway</artifactId>
```

```
</dependency>
```


2) 添加Sentinel配置

- 在gateway-service模块的application.properties中添加以下配置:

#配置Sentinel DashBoard地址

spring.cloud.sentinel.transport.dashboard=localhost:8080

#取消Sentinel DashBoard懒加载(sentinel客户端项目一旦启动, sentinel客户端的信息就出现在Sentinel DashBoard上)

spring.cloud.sentinel.eager=true



3) 设置限流规则

- 启动gateway-service，访问自定义路由的微服务，则自定义路由会出现在Sentinel Dashboard的“簇点链路”列表中，进而可为其添加流控、熔断、热点、授权等规则。

gateway-service

树状视图 列表视图

簇点链路

192.168.31.103:8719

关键字

刷新

资源名	通过QPS	拒绝QPS	并发数	平均RT	分钟通过	分钟拒绝	操作
▼ sentinel_gateway_context\$route\$toOrderingService	0	0	0	0	0	0	<div>+ 流控</div> <div>+ 熔断</div> <div>+ 热点</div> <div>+ 授权</div>
<u>toOrderingService</u>	0	0	0	0	0	0	<div>+ 流控</div> <div>+ 熔断</div> <div>+ 热点</div> <div>+ 授权</div>
sentinel_default_context	0	0	0	0	0	0	<div>+ 流控</div> <div>+ 熔断</div> <div>+ 热点</div> <div>+ 授权</div>

共 3 条记录, 每页

16

条记录

4) 配置异常处理

- Sentinel整合Gateway后，提供了一个基于ServerWebExchange的BlockRequestHandler接口，让开发人员可以根据限流异常类型自定义异常返回信息。
- 为此，在gateway-service模块中增加一个BlockRequestHandler的实现类GatewayExceptionHandler，并新建一个配置类GatewayConfig在其中配置异常返回信息和异常处理策略。



GatewayExceptionHandler类

```
package com.scst.handler;

import com.alibaba.csp.sentinel.adapter.gateway.sc.callback.BlockRequestHandler;
import com.alibaba.csp.sentinel.slots.block.authority.AuthorityException;
import com.alibaba.csp.sentinel.slots.block.degrade.DegradeException;
import com.alibaba.csp.sentinel.slots.block.flow.FlowException;
import com.alibaba.csp.sentinel.slots.block.flow.param.ParamFlowException;
import com.alibaba.csp.sentinel.slots.system.SystemBlockException;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.BodyInserters;
import org.springframework.web.reactive.function.server.ServerResponse;
import org.springframework.web.server.ServerWebExchange;
import reactor.core.publisher.Mono;
```



GatewayExceptionHandler类

@Component

public class GatewayExceptionHandler implements BlockRequestHandler {

@Override

public Mono<ServerResponse> handleRequest(ServerWebExchange serverWebExchange, Throwable e) {

 ResponseData data = null;

 if (e instanceof FlowException) {

 data = new ResponseData(-1, "网关流控规则不通过");

 } else if (e instanceof DegradeException) {

 data = new ResponseData(-2, "网关熔断规则不通过");

 } else if (e instanceof ParamFlowException) {

 data = new ResponseData(-3, "网关热点规则不通过");

 } else if (e instanceof AuthorityException) {

 data = new ResponseData(-4, "网关授权规则不通过");

 } else if (e instanceof SystemBlockException) {

 data = new ResponseData(-5, "网关系统规则不通过");

 } else { data = new ResponseData(0, e.getMessage()); }

 return ServerResponse.status(HttpStatus.OK).contentType(MediaType.APPLICATION_JSON).body(BodyInserters.fromValue(data));

}

}

GatewayExceptionHandler类

```
@Data
@AllArgsConstructor//全参构造
@NoArgsConstructor
//无参构造
class ResponseData {
    private int code;
    private String message;
}
```



GatewayConfig类

```
package com.scst.config;

import com.alibaba.csp.sentinel.adapter.gateway.sc.callback.BlockRequestHandler;
import com.alibaba.csp.sentinel.adapter.gateway.sc.callback.GatewayCallbackManager;
import com.alibaba.csp.sentinel.adapter.gateway.sc.exception.SentinelGatewayBlockExceptionHandler;
import org.springframework.beans.factory.ObjectProvider;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.core.annotation.Order;
import org.springframework.http.codec.ServerCodecConfigurer;
import org.springframework.web.reactive.result.view.ViewResolver;
import javax.annotation.PostConstruct;
import java.util.Collections;
import java.util.List;
```



GatewayConfig类

```
@Configuration
public class GatewayConfig {
    @Autowired
    private BlockRequestHandler blockRequestHandler;
    private final List<ViewResolver> viewResolvers;
    private final ServerCodecConfigurer serverCodecConfigurer;

    public GatewayConfig(ObjectProvider<List<ViewResolver>> viewResolversProvider,
        ServerCodecConfigurer serverCodecConfigurer) {
        this.viewResolvers = viewResolversProvider.getIfAvailable(Collections::emptyList);
        this.serverCodecConfigurer = serverCodecConfigurer;
    }
}
```



GatewayConfig类

//配置限流异常返回信息，在依赖注入完成后被自动调用（执行顺序：构造函数GatewayConfig()-@Autowired-@PostConstruct）

@PostConstruct

```
public void initBlockHandlers() {
```

```
    GatewayCallbackManager.setBlockHandler(blockRequestHandler);
```

```
}
```

//配置限流异常处理器

@Bean

@Order(Ordered.HIGHEST_PRECEDENCE)

```
public SentinelGatewayBlockExceptionHandler sentinelGatewayBlockExceptionHandler() {
```

```
    return new SentinelGatewayBlockExceptionHandler(viewResolvers, serverCodecConfigurer);
```

```
}
```

```
}
```



5.网关跨域配置

- 1) 跨域访问
- 2) 跨域配置



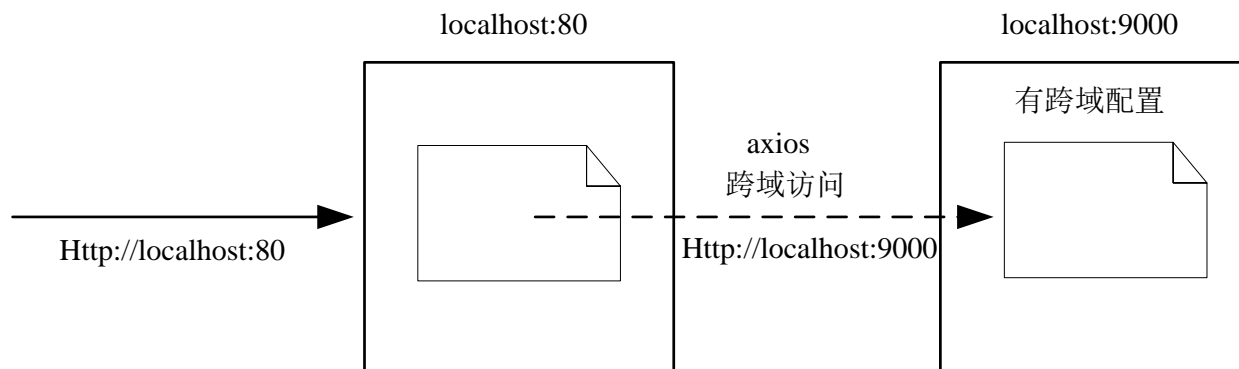
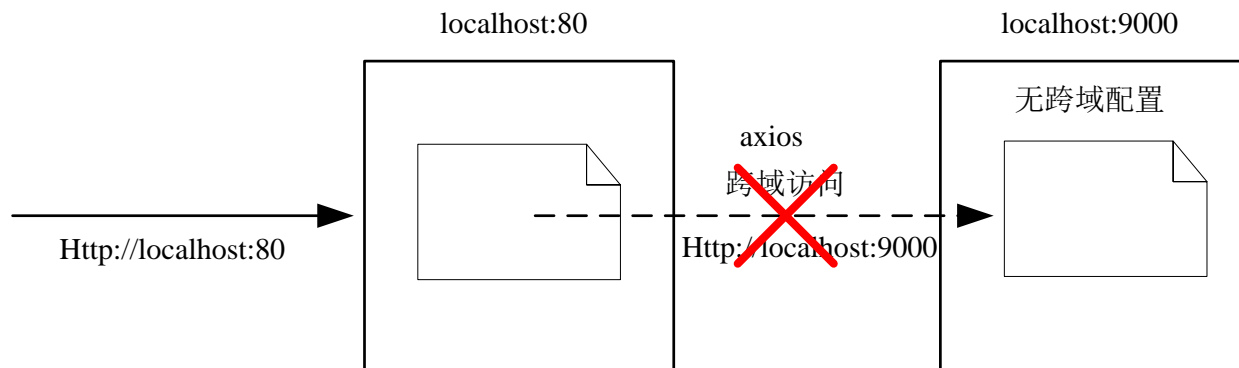
1) 跨域访问

■ 跨域访问即通过**HTTP**请求，在一个域去请求另一个域的资源。

◇ 只要协议、域名、端口有任何一个不相同，都会被当作是不同的域。



跨域访问



allowed-origins: "*" *"
allowed-headers: "*" *"
allowed-methods: "*" *"



2) 跨域配置

- 只需要在网关微服务中统一配置跨域，这样就不需在其他微服务中设置跨域。
- 有两种配置方法：
 - ✧①在配置文件中配置
 - ✧②在配置类中配置



①在配置文件中配置

■ 在gateway-service模块的application.properties配置文件中，添加如下代码：

#跨域配置：允许所有来源跨域访问所有资源

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-origins[0]="*"
```

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-headers[0]="*"
```

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-methods[0]="GET"
```

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allowed-methods[1]="POST"
```

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].allow-credentials=true
```

```
spring.cloud.gateway.globalcors.cors-configurations.[/**].max-age=3600
```



②在配置类中配置

- 在gateway-service模块中新建一个配置类CorsConfig，实现跨域访问网关设置。



CorsConfig类

```
package com.scst.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.reactive.CorsWebFilter;
import org.springframework.web.cors.reactive.UrlBasedCorsConfigurationSource;
import org.springframework.web.util.pattern.PathPatternParser;

@Configuration
public class CorsConfig {

    @Bean
    public CorsWebFilter corsWebFilter(){
        CorsConfiguration config=new CorsConfiguration();
        config.addAllowedHeader("*");
        config.addAllowedMethod("*");
        config.addAllowedOrigin("*");

        UrlBasedCorsConfigurationSource source =new UrlBasedCorsConfigurationSource(new PathPatternParser());
        source.registerCorsConfiguration("/*",config);
        return new CorsWebFilter(source);
    }
}
```

6.搭建高可用网关环境

■业内通常用多少9来衡量网站的可用性。

✧对于大多数网站，是2个9基本可用；

✧3个9是高可用；

✧4个9是拥有自动恢复能力的高可用。

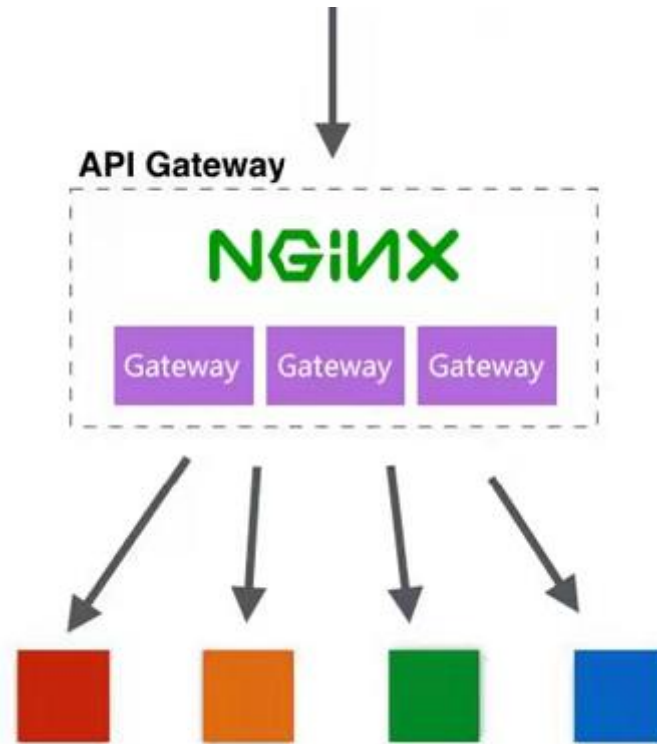
■实现高可用的主要手段是数据冗余备份和服务失效转移。主要有以下途径：

✧集群部署、负载均衡、健康检查、节点自动重启、熔断、服务降级、接口重试。

■这里采用Nginx+网关集群实现高可用网关。



高可用网关



Nginx配置

- 进入Nginx的conf目录，打开nginx.conf文件，配置网关集群：

```
1  http {
2
3      ...
4
5      # 网关集群
6      upstream gateway {
7          server 127.0.0.1:9000;
8          server 127.0.0.1:9001;
9      }
10
11     server {
12         listen      80;
13         server_name localhost;
14
15         ...
16
17         # 代理网关集群，负载均衡调用
18         location / {
19             proxy_pass http://gateway;
20         }
21
22         ...
23     }
24
25     ...
26
27 }
```



启动测试

■ Nginx配置完成后，按以下命令启动Nginx，或重载配置。

✧启动Nginx: `start nginx`

✧重载配置: `nginx -s reload`

■ 通过网关集群访问微服务效果:



