

微服务

什么是微服务？

微服务架构是一种将单一应用程序划分为一组小的、独立运行的服务的设计方法。每个服务都实现特定的业务功能，并且可以独立地部署和扩展。这些服务通过定义良好的API相互通信，它们通常是分布式的，部署在不同的服务器上或云环境中。

微服务的主要特点

- **模块化**: 每个微服务专注于单一的功能领域。
- **独立性**: 每个服务都可以独立开发、测试、部署和扩展。
- **轻量级通信**: 服务之间通常使用HTTP/REST、gRPC等轻量级协议进行通信。
- **去中心化治理**: 允许团队选择最适合他们需求的技术栈。
- **数据分散管理**: 每个微服务可以拥有自己的数据库，以便于数据管理和性能优化。

为什么使用微服务？

采用微服务架构有多种好处，以下是一些主要的原因：

1. **技术多样性**: 不同的服务可以使用不同的编程语言和技术栈来构建，这样可以根据具体的需求选择最合适的技术。
2. **加速上市时间**: 由于服务是独立的，因此可以单独对某个服务进行更新或改进，而不需要等待整个系统的完成，从而加快了新功能的发布速度。
3. **易于扩展**: 可以根据实际需要独立地扩展某些服务，而不是扩展整个应用，这有助于更有效地利用资源。
4. **增强的容错性和隔离性**: 如果一个服务出现问题，它不会影响其他服务的正常运作，提高了系统的稳定性和可靠性。
5. **简化维护**: 较小的代码库更容易理解和维护，而且问题定位更加简单，因为只需要关注出错的服务而不是整个系统。
6. **促进敏捷开发和DevOps实践**: 微服务架构鼓励持续集成和持续交付（CI/CD），使得团队能够更快地响应变化并实施自动化部署流程。

尽管微服务带来了许多优势，但同时也引入了复杂性，比如分布式系统的调试难度增加、服务间通信的成本以及数据一致性等问题。因此，在决定是否采用微服务架构时，需根据项目的具体情况权衡利弊。对于小型项目或者那些预计不会有太大增长的应用来说，单体架构可能是更合适的选择；而对于大型企业级应用或是预期会有快速发展的产品，则微服务架构可能提供更大的灵活性和可扩展性。

分布式基础

单体架构

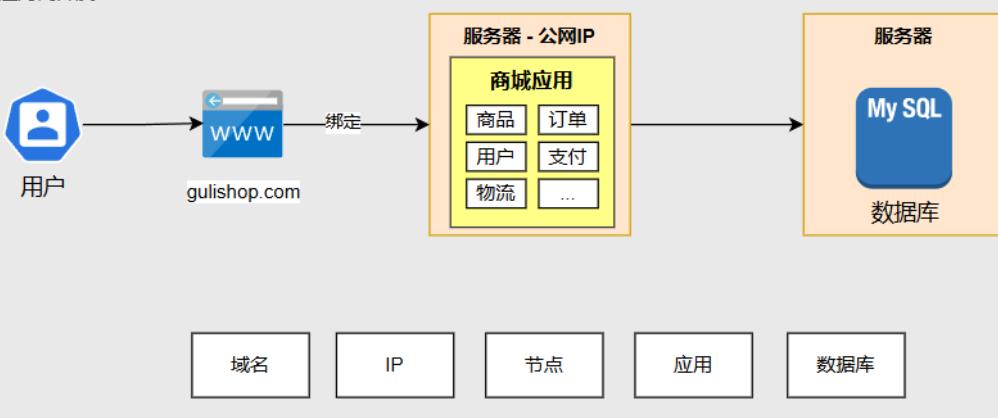
单体架构 (ALL IN ONE)

所有功能模块都在一个项目

1、项目打包

优点：开发部署

缺点：无法应对高并发



集群架构

集群架构

解决大并发

问题1：模块化升级

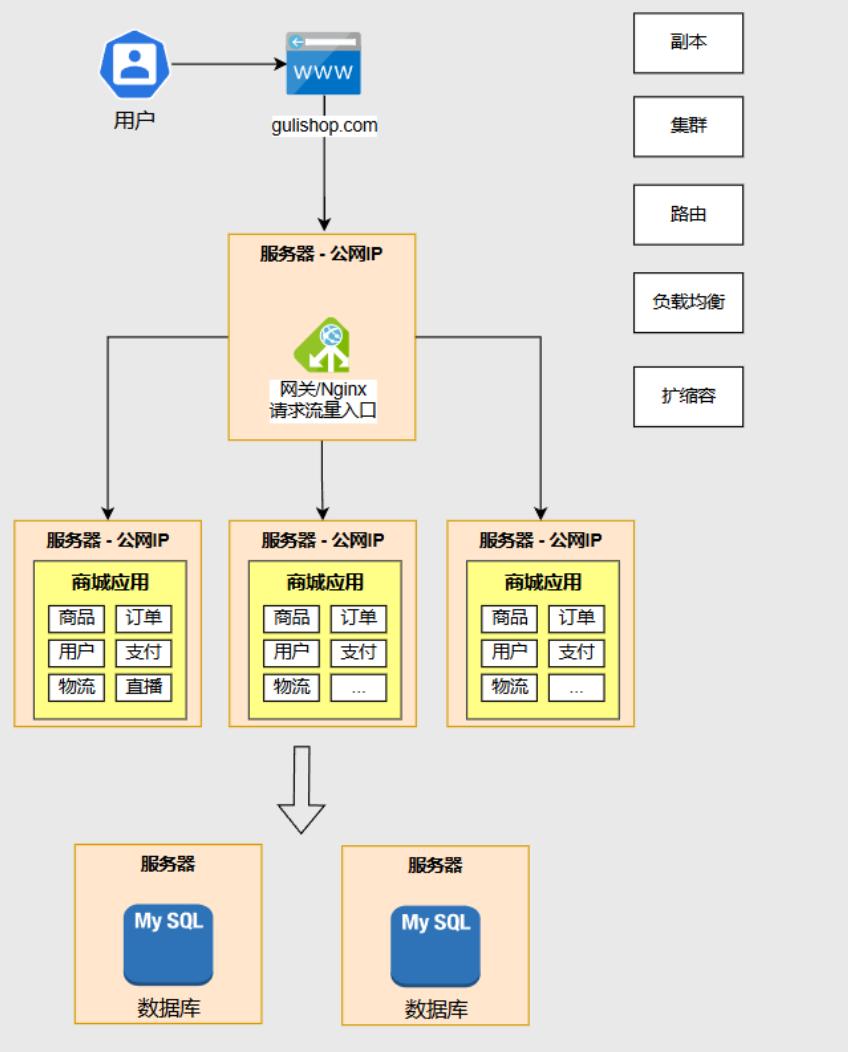
订单功能经常升级：v1.0, v2.0

问题2：多语言团队

C++直播模块

模块化升级

多语言团队



分布式架构

分布式架构

分布式：一个大型应用被拆分成很多小应用分布部署在各个机器；工作方式

集群：物理形态

MySQL

数据库



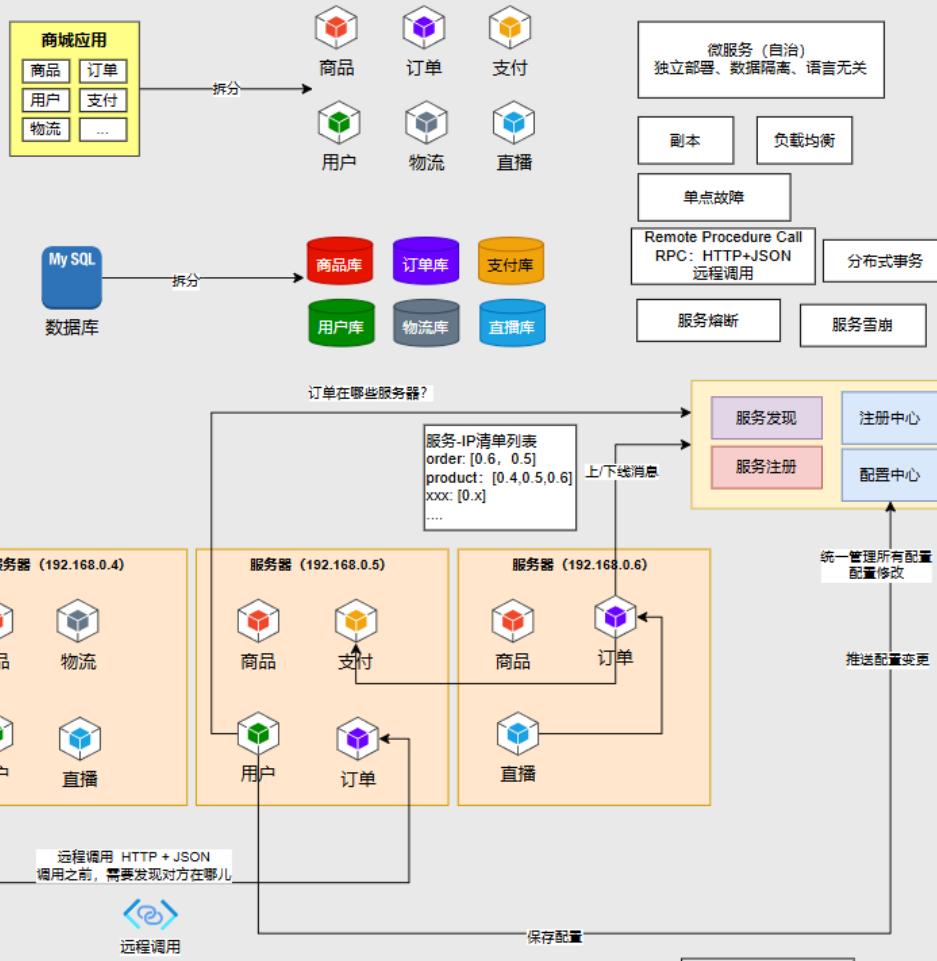
注册中心

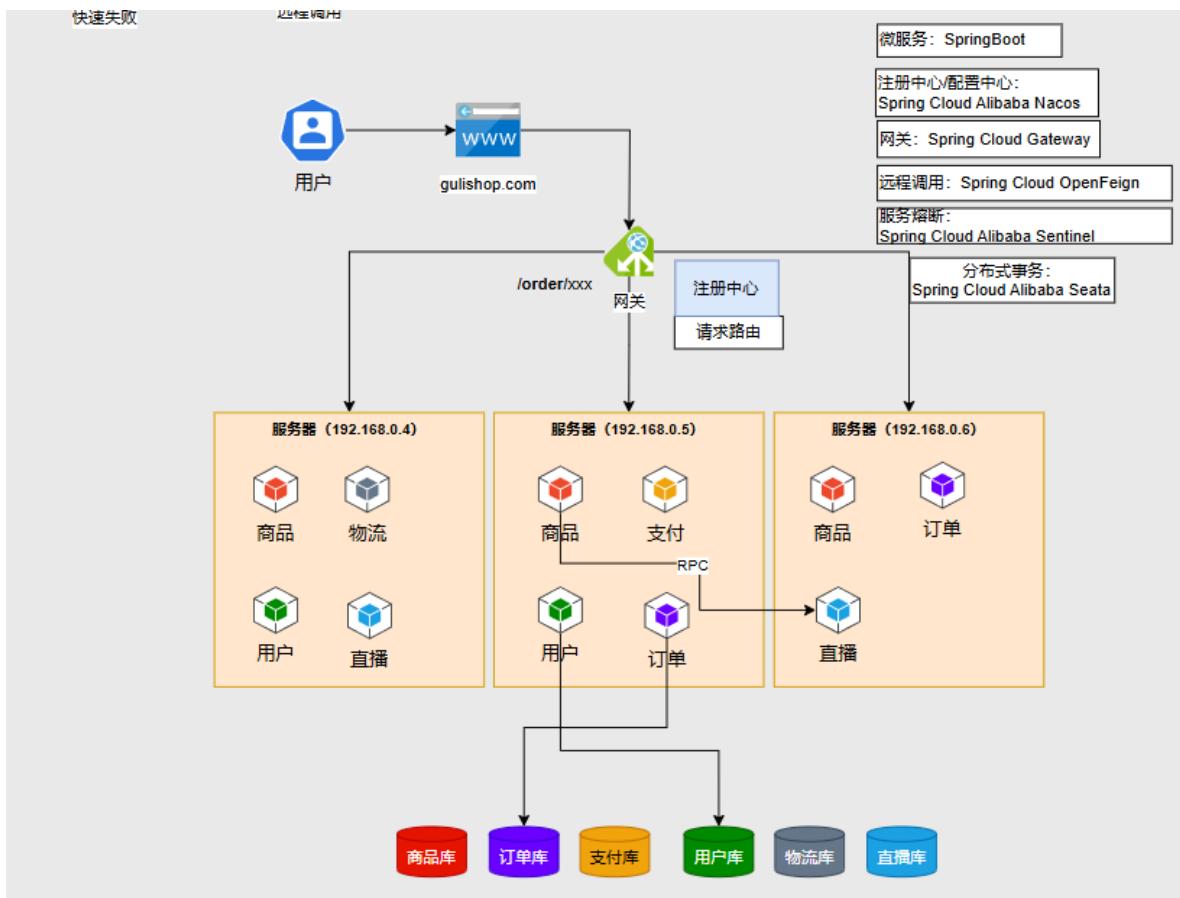


配置中心

模块化升级

多语言团队





Nacos

在nacos的bin目录下使用 `startup.cmd -m standalone` 命令启动nacos。通过8848端口访问nacos

注册中心

服务注册

| 流程 | 内容 | 核心 |
|-----|-----------|--|
| 步骤1 | 启动微服务 | SpringBoot 微服务web项目启动 |
| 步骤2 | 引入服务发现依赖 | <code>spring-cloud-starter-alibaba-nacos-discovery</code> |
| 步骤3 | 配置Nacos地址 | <code>spring.cloud.nacos.server-addr=127.0.0.1:8848</code> |
| 步骤4 | 查看注册中心效果 | 访问 <code>http://localhost:8848/nacos</code> |
| 步骤5 | 集群模式启动测试 | 单机情况下通过改变端口模拟微服务集群 |

在service-order模块的application.properties中加上

```
spring.application.name=service-order
server.port=8000

spring.cloud.nacos.server-addr=127.0.0.1:8848
```

再启动springboot应用

在nacos中会发现已经注册成功

服务发现

即获取注册中心所有的微服务及其信息

| 流程 | 内容 | 核心 |
|-----|-----------|------------------------|
| 步骤1 | 开启服务发现功能 | @EnableDiscoveryClient |
| 步骤2 | 测试服务发现API | DiscoveryClient |
| 步骤3 | 测试服务发现API | NacosServiceDiscovery |

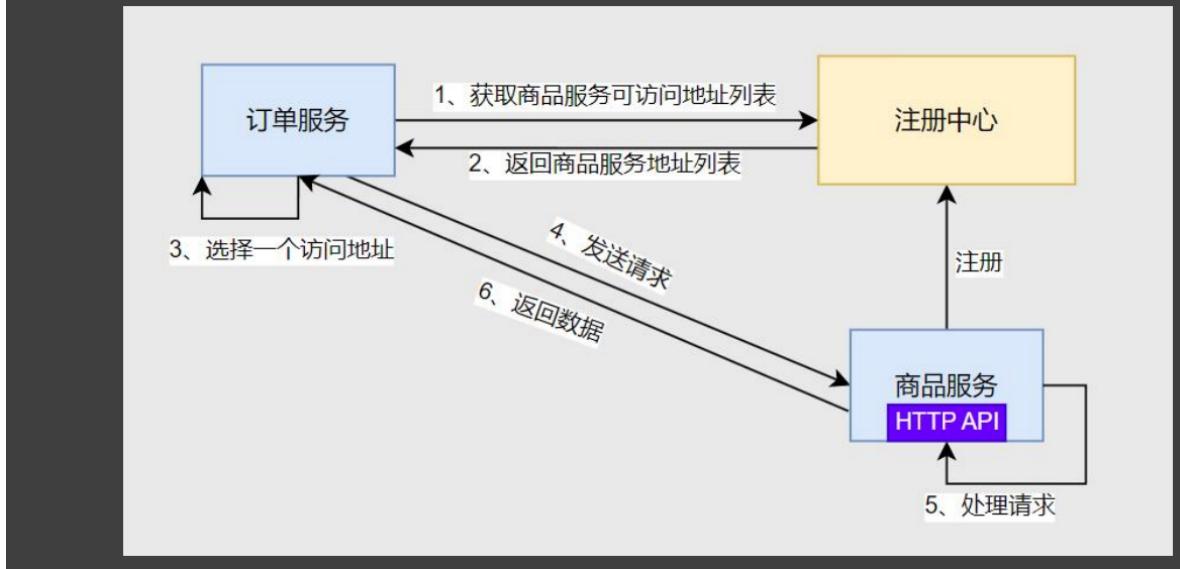
```
@SpringBootTest
public class DiscoveryTest {
    @Autowired
    DiscoveryClient discoveryClient;

    @Test
    void test01(){
        for (String service : discoveryClient.getServices()) {
            System.out.println("service:"+service);
            //获取ip和port
            List<ServiceInstance> instances =
discoveryClient.getInstances(service);
            for (ServiceInstance instance : instances) {
                System.out.println(instance.getHost()+":"+instance.getPort());
            }
        }
    }
}
```

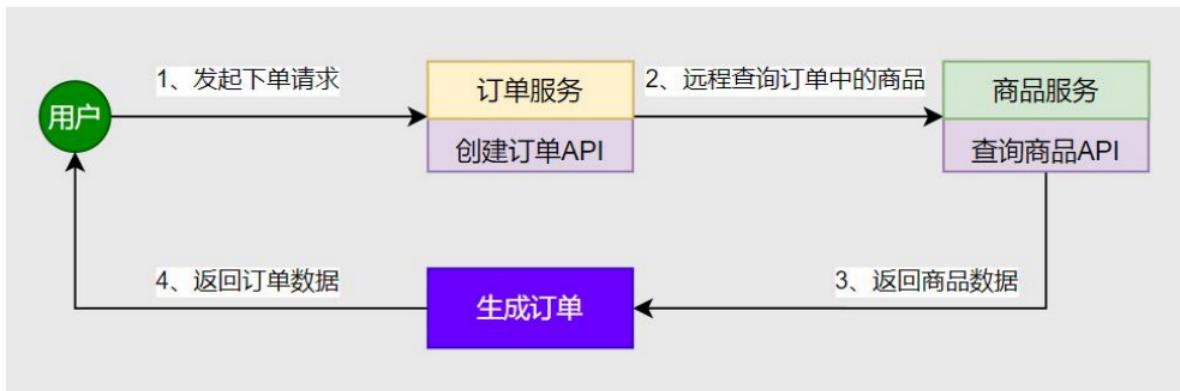
编写微服务API

远程调用基本流程：

远程调用 - 基本流程



下面用一个订单-商品业务简单实现API，基本流程如下：



实现：

- 把bean类单独做成一个模块model，并将其作为services的dependency
order:

```
public class Order {  
    private Long id;  
    private BigDecimal totalAmount;  
    private Long userId;  
    private String nickname;  
    private String address;  
    private List<Product> productList;  
}
```

product:

```
public class Product {  
    private Long id;  
    private BigDecimal price;  
    private String productName;  
    private int num;  
}
```

- service-product

- service:

```
@Service  
public class ProductServiceImpl implements ProductService {  
    @Override  
    public Product getProductId(Long id) {  
        Product product = new Product();  
        product.setId(id);  
        product.setPrice(new BigDecimal("99"));  
        product.setProductName("苹果"+id);  
        product.setNum(2);  
  
        return product;  
    }  
}
```

- controller:

```
@RestController  
public class ProductController {  
    @Autowired  
    ProductService productService;  
  
    //查询商品  
    @GetMapping("/product/{id}")  
    public Product getProductId(@PathVariable("id") Long id) {  
        Product product=productService.getProductId(id);  
        return product;  
    }  
}
```

模拟了通过id查询商品的业务流程

- service-order:

- config:

```

@Configuration
public class OrderServiceConfig {
    @Bean
    RestTemplate restTemplate(){
        return new RestTemplate();
    }
}

```

作用是注册一个RestTemplate组件，方便进行远程请求

- o service:

```

@Slf4j
@Service
public class OrderServiceImpl implements OrderService {
    @Autowired
    DiscoveryClient discoveryClient;
    @Autowired
    RestTemplate restTemplate;

    @Override
    public Order createOrder(Long userId, Long productId) {
        Product product = getProductFromRemote(productId);

        Order order = new Order();
        order.setId(1L);

        order.setTotalAmount((product.getPrice().multiply(new
BigDecimal(product.getNum()))));
        order.setUserId(userId);
        order.setNickname("zhangsan");
        order.setAddress("ss");
        order.setProductList(Arrays.asList(product));

        return order;
    }

    private Product getProductFromRemote(Long productId){
        //获取所有商品服务所在的所有机器IP+port
        List<ServiceInstance> instances =
discoveryClient.getInstances("service-product");

        ServiceInstance serviceInstance = instances.get(0);
        //远程url
        String url = "http://"+serviceInstance.getHost() + ":" +
serviceInstance.getPort()+"/product/"+productId;
        log.info("远程请求: {}"+url);
        //给远程发送请求
        Product product = restTemplate.getForObject(url, Product.class);
        return product;
    }
}

```

- o controller:

```

@RestController
public class OrderController {
    @Autowired
    OrderService orderService;

    @GetMapping("/create")
    public Order createOrder(@RequestParam("userId") Long userId,
                            @RequestParam("productId") Long productId)
    {

        Order order = orderService.createOrder(userId, productId);
        return order;
    }

}

```

模拟了不同服务器之间的模块之间的远程调用

负载均衡API

上面的API每次都给一个固定的位置发远程请求，但在真实环境中，如果这个服务器的这个组件出了故障，就会导致系统崩溃。所以我们考虑实现负载均衡API：能够以某种规则每次给不同服务器的相同业务模块发送远程请求。

步骤：

| 流程 | 内容 | 核心 |
|-----|-----------|-----------------------------------|
| 步骤1 | 引入负载均衡依赖 | spring-cloud-starter-loadbalancer |
| 步骤2 | 测试负载均衡API | LoadBalancerClient |
| 步骤3 | 测试远程调用 | RestTemplate |
| 步骤4 | 测试负载均衡调用 | @LoadBalanced |

- 在service-order中引入负载均衡依赖
- 修改service-order中的方法 `getProductFromRemote` 为
`getProductFromRemoteWithLoadBalancer`

```

private Product getProductFromRemoteWithLoadBalancer(Long productId){
    //获取所有商品服务所在的所有机器IP+port
    ServiceInstance choose=loadBalancerClient.choose("service-product");

    //远程url
    String url = "http://" + choose.getHost() + ":" +
choose.getPort() + "/product/" + productId;
    log.info("远程请求: {}" + url);
    //给远程发送请求
    Product product = restTemplate.getForObject(url, Product.class);
    return product;
}

```

用注解进行负载均衡:

- 给restTemplate加上注解:

```
@LoadBalanced  
@Bean  
RestTemplate restTemplate(){  
    return new RestTemplate();  
}
```

- 修改方法:

```
private Product getProductFromRemoteWithLoadBalancerAnnotation(Long productId){  
    //获取所有商品服务所在的所有机器IP+port  
    //ServiceInstance choose=loadBalancerClient.choose("service-product");  
    //远程url  
    //String url = "http://" + choose.getHost() + ":" +  
    choose.getPort() + "/product/" + productId;  
    String url = "http://service-product/product/" + productId;  
    log.info("远程请求: " + url);  
    //给远程发送请求 url中的service-product会被动态替换成负载均衡的地址  
    Product product = restTemplate.getForObject(url, Product.class);  
    return product;  
}
```

思考

如果注册中心宕机了（微服务正常运行），还能进行远程调用吗：

- 进行远程调用的流程：
 - 请求注册中心，获取微服务地址列表
 - 负载均衡地发送请求
- 使用过的微服务实例会有缓存，再次调用就不需要访问注册中心了
- 所以：
 - 调用过，可以
 - 没调用过（第一次发起远程调用），不行

配置中心

基本使用

配置中心 - 基本使用

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

```
spring.cloud.nacos.server-addr=127.0.0.1:8848
spring.config.import=nacos:service-order.properties
```

数据集: service-order.properties

内容:

```
1  order.timeout=10min
2  order.auto-confirm=7d
```

启动Nacos



引入依赖



application.properties 配置



创建 data-id (数据集)

- 引入依赖

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

- 指定配置

```
spring.config.import=nacos:service-order.properties
```

- 在nacos中创建配置
- 编写API进行测试：

```
@Value("${order.timeout}")
String ordertimeout;
@Value("${order.auto-confirm}")
String orderautoConfirm;

@GetMapping("/config")
public String config(){
    return ordertimeout+orderautoConfirm;
}
```

注意：在导入依赖后，如果不在application.properties中指定配置，会报错。

如果此时我们不需要配置，那么用 `spring.cloud.nacos.config.import-check.enabled=false` 改成不必须配置。

动态刷新

- 此时还不能做到在不重启项目的情况下自动刷新配置的更改，如果想要自动刷新，还要再 controller上加上注解：

```
@RefreshScope
```

- 自动刷新的方便做法：使用spring注解 `@ConfigurationProperties()` :

创建一个properties包, 创建orderProperties类, 进行注解

```
@Component  
@ConfigurationProperties(prefix = "order")  
@Data  
public class OrderProperties {  
  
    String timeout;  
  
    String autoConfirm;  
}
```

测试:

```
@Autowired  
OrderProperties orderProperties;  
  
@GetMapping("/config")  
public String config(){  
    return orderProperties.getTimeout()+orderProperties.getAutoConfirm();  
}
```

配置监听

模拟一个场景:

我们想要:

1. 项目启动就开始监听配置文件的变化
2. 发生变化后拿到变化值
3. 发送邮件

```
@Bean  
ApplicationRunner applicationRunner(NacosConfigManager nacosConfigManager) {  
    return args-> {  
        ConfigService configService = nacosConfigManager.getConfigService();  
        configService.addListener("service-order.properties",  
            "DEFAULT_GROUP",  
            new Listener() {  
                @Override  
                public Executor getExecutor() {  
                    return Executors.newFixedThreadPool(4); //线程池  
                }  
                //输出配置信息  
                @Override  
                public void receiveConfigInfo(String s) {  
                    System.out.println("变化的配置信息"+s);  
                }  
            });  
        System.out.println("=====");  
    };  
}
```

- ApplicationRunner是一个项目一启动就会执行的任务

思考

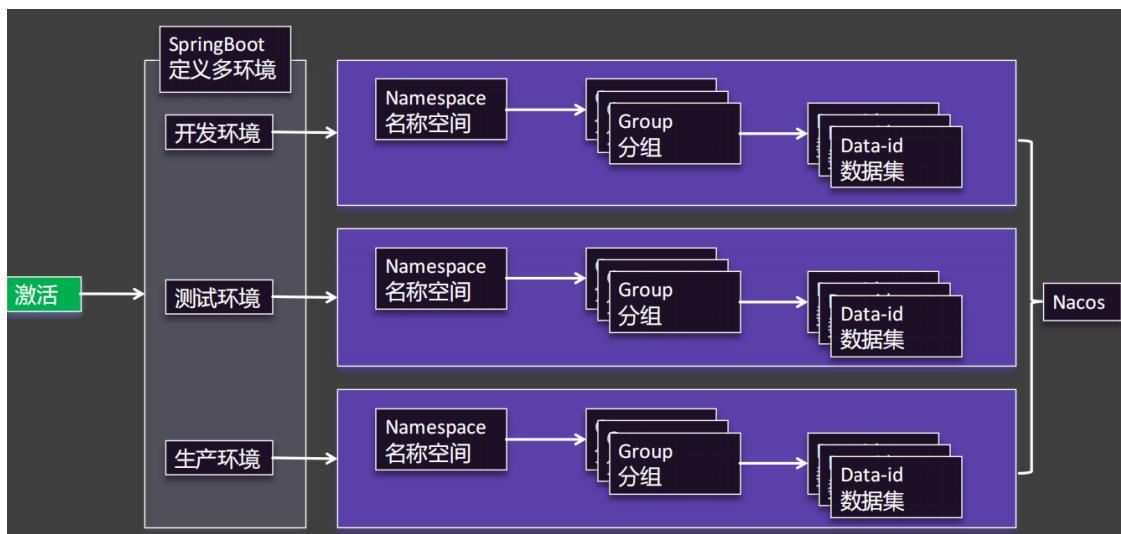
如果同一个配置在微服务和配置中心都进行了配置，以哪个为准？
还是那个规则：

先导入优先、外部优先

配置中心为准

数据隔离

- 需求描述
 - 项目有多套环境：dev, test, prod
 - 每个微服务，同一种配置，在每套环境的值都不一样。
 - 如：database.properties
 - 如：common.properties
 - 项目可以通过切换环境，加载本环境的配置
- 难点
 - 区分多套环境
 - 区分多种微服务
 - 区分多种配置
 - 按需加载配置



一个namespace对应一种开发环境、一个group对应一个微服务（order、product）。

在nacos中创建好配置后，需要实现按需加载

因为properties如果配置多了显得很乱，所以我们改用yml

```

server:
  port: 8080
spring:
  application:
    name: service-order
  profiles:
    active: dev #激活环境
  cloud:

```

```

nacos:
  server-addr: 127.0.0.1:8848
  config:
    namespace: ${spring.profiles.active:dev} #动态配置环境， 默认dev

# 不同环境进行不同的配置
---

spring:
  config:
    import:
      - nacos:common.properties?group=order
      - nacos:database.properties?group=order
  activate:
    on-profile: dev

---

spring:
  config:
    import:
      - nacos:common.properties?group=order
      - nacos:database.properties?group=order
  activate:
    on-profile: test

---

spring:
  config:
    import:
      - nacos:common.properties?group=order
      - nacos:database.properties?group=order
  activate:
    on-profile: prod

```

并且在OrderProperties类中加上相应的属性

总结

注册中心

- 引入 **spring-cloud-starter-alibaba-nacos-discovery** 依赖，配置Nacos地址
- @EnableDiscoveryClient** 开启服务发现功能

扩展：

- DiscoveryClient** 获取服务实例列表
- LoadBalancerClient** 负载均衡选择一个实例
(需要引入 **spring-cloud-starter-loadbalancer**)
- RestTemplate** 可以发起远程调用

配置中心

- 引入 **spring-cloud-starter-alibaba-nacos-config** 依赖，配置Nacos地址
- 添加 **数据集** (data-id) , 使用 **spring.config.import** 导入数据集
- @Value + @RefreshScope** 取值 + 自动刷新
- @ConfigurationProperties**批量绑定自动刷新
- NacosConfigManager** 监听配置变化 →

扩展：

配置优先级：namespace区分环境、group区分微服务、data-id区分配置 实现 数据隔离+环境切换

*OpenFeign

远程调用

在nacos学习中，我们已经使用了远程调用，但那是编程式 REST客户端。

而OpenFeign是springcloud用来做远程调用的组件，是一种声明式的REST客户端（Declared REST Client）。

rest客户端就是能发送HTTP请求进行远程调用的工具

- 注解驱动
 - 指定远程地址：@FeignClient
 - 指定请求方式：@GetMapping、@PostMapping、@DeleteMapping ...
 - 指定携带数据：@RequestHeader、@RequestParam、@RequestBody ...
 - 指定结果返回：响应模型

声明式实现

第一步：

依赖：

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

在主程序标注注解用来开启Feign远程调用功能。 @EnableFeignClients

第二步：

编写远程调用的客户端

```
package com.atguigu.order.feign;

import com.atguigu.product.bean.Product;
import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;

//说明这是一个远程调用的Feign客户端，并指定服务名
@EnableFeignClients(value = "service-product")
public interface ProductFeignClient {

    //MVC注解的两套使用逻辑
    //1.标注在controller上：接收这样的请求
    //2.标注在FeignClient上：发送这样的请求
}
```

```
@GetMapping("/product/{id}")
Product getProductById(@PathVariable("id") Long id);
//这里的参数作用也是和controller相反的，表示将id传递给uri
}
```

第三步：

在OrderServiceImpl中自动注入productFeignClient

并

修改OrderServiceImpl中方法：

```
//Product product = getProductFromRemoteWithLoadBalancerAnnotation(productId);
//使用Feign代替RESTTemplate完成远程调用
Product product=productFeignClient.getProductById(productId);
```

经过测试已经可以实现**负载均衡的远程调用了**

第三方API

上面是给我们自己注册的服务发送远程调用，除此之外，我们还可以对第三方API发送远程调用：

以墨迹天气为例：

WeatherFeignClient:

```
package com.atguigu.order.feign;

import org.springframework.cloud.openfeign.FeignClient;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestParam;

@FeignClient(value="weather-client",url="http://aliv18.data.moji.com")
public interface WeatherFeignClient {

    @PostMapping("/whapi/json/alicityweather/condition")
    String getWeather(@RequestHeader("Authorization") String auth,
                      @RequestParam("token") String token,
                      @RequestParam("cityId") String cityId);

}
```

test:

```
@SpringBootTest
```

```

public class WeatherTest {
    @Autowired
    WeatherFeignClient weatherFeignClient;

    @Test
    void test01() {
        String weather=weatherFeignClient.getWeather("APPCODE
93b7e19861a24c519a7548b17dc16d75",
        "50b53ff8dd7d9fa320d3d3ca32cf8ed1",
        "2182");
        System.out.println(weather);
    }
}

```

成功输出：

```
{"code":0,"data":{"city":{"cityId":2182,"counname":"中国","ianatimezone":"Asia/Shanghai","name":"西安市","pname":"陕西省","secondaryname":"陕西省","timezone":"8"},"condition":[{"condition":"阴","conditionId":85,"humidity":29,"icon":2,"pressure":981,"realFeel":5,"sunRise":"2025-03-29 06:36:00","sunSet":"2025-03-29 19:03:00","temp":7,"tips":"天冷了，该加衣服了！","updateTime":2025-03-29 23:05:08,"uvi":1,"vis":9700,"windDegrees":270,"windDir":西风,"windLevel":1,"windSpeed":0.89}], "msg": "success", "rc": {"c":0, "p": "success"}}}
```

小技巧：

当调用业务API（我们自己注册的服务）时，直接将controller对应的方法签名复制到feign接口中进行声明

比如：

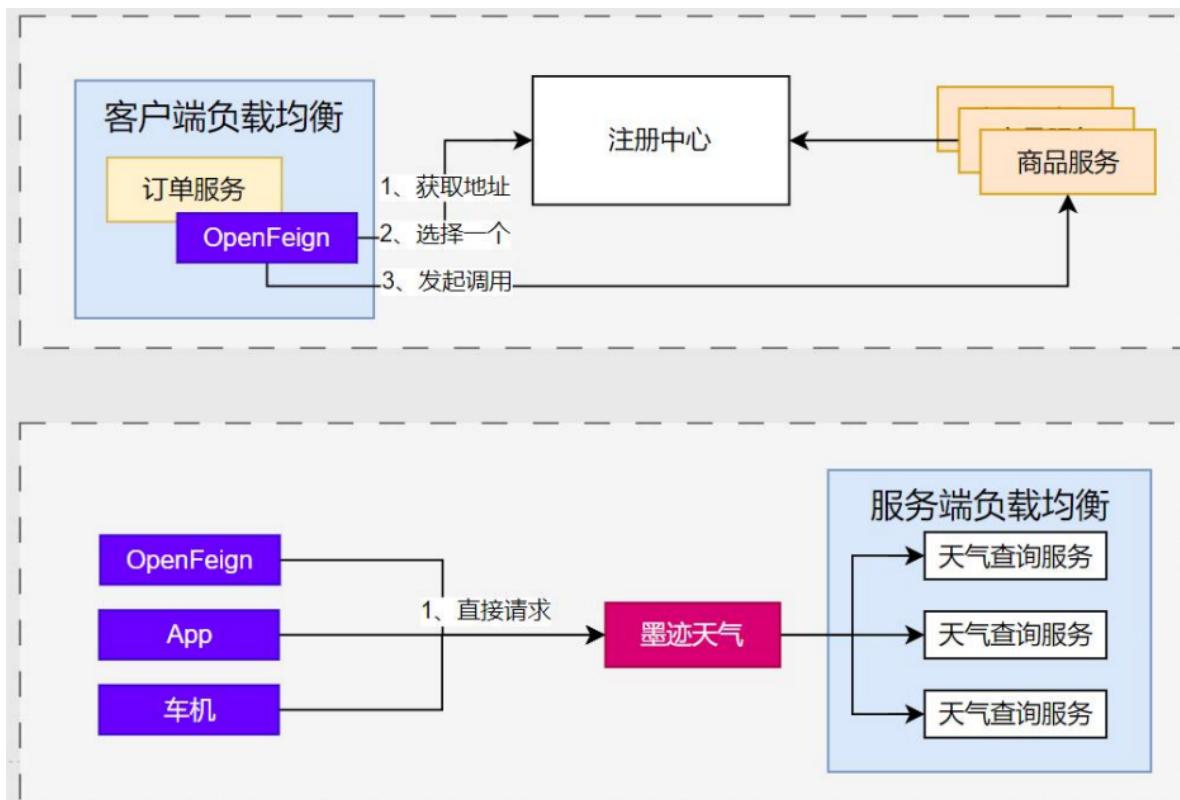
```

@GetMapping("/create")
public Order createOrder(@RequestParam("userId") Long userId,
    @RequestParam("productId") Long productId) {

```

第三方API：根据接口文档确定请求如何发

客户端负载均衡与服务端负载均衡区别



进阶配置

日志

配置信息:

```
logging:
  level:
    com.atguigu.order.feign: debug
```

在容器中加入组件:

在OrderServiceConfig中添加:

```
@Bean
Logger.Level feignLoggerLevel(){
    return Logger.Level.FULL;
}
```

接下来在发送远程调用请求时，就会多出日志。

超时控制

在远程调用时，如果OpenFeign远程调用的服务器宕机或速度太慢，容易造成雪崩。这时就需要进行超时控制。

超时又分两种:



接下来我们模拟超时的情况：

在 `productServiceImpl` 中睡眠 100 秒：

```

try {
    TimeUnit.SECONDS.sleep(100);
} catch (InterruptedException e) {
    throw new RuntimeException(e);
}
  
```

接下来启动 `order` 向 `product` 发送远程请求，在等待 60 秒后，返回了错误信息。

下面来修改配置，因为 `application.yml` 中配置过多，我们新创一个 `application-feign.yml`，其中的 `feign` 是环境标识，为了让此环境生效，在 `application.yml` 中进行如下配置

```

spring:
  profiles:
    include: feign
  
```

然后在 `application-feign.yml` 中进行如下配置：

```

spring:
  cloud:
    openfeign:
      client:
        config:
          default:
            logger-level: full
            connect-timeout: 1000
            read-timeout: 2000
          service-product:
            logger-level: full
            connect-timeout: 3000
            read-timeout: 5000
  
```

表示默认配置和指定配置，其中指定配置的名字（这里是service-product）取决于@FeignClient注解中connentId指定的值，如果没有connentId，那就是value指定的。

重试机制

远程调用超时失败后，还可以进行多次尝试，如果某次成功返回ok，如果多次依然失败则结束调用，返回错误

底层是默认不重试的，如果要设置重试，进行如下配置；

给容器中放入Retryer组件

```
@Bean
Retryer retryer(){
    return new Retryer.Default();
}
```

这里使用的是默认配置，如果想要修改，那么return中使用有参构造器，第一个参数是请求失败与第一次重试之间的间隔，之后每次请求失败与下一次重试之间的间隔都乘以1.5，第二个参数是请求失败与下一次重试之间的间隔的最大值，乘以1.5超过这个值就按这个值算。第三个参数是最大请求次数（包含第一次请求）

拦截器

在进行远程调用前，对OpenFeign发送的请求进行定制修改，使用了**请求拦截器**，在响应时，也可以使用**响应拦截器**进行响应预处理。

我们以请求拦截器为例，我们需要在请求头中放一个X-Token，如下：

```
package com.atguigu.order.interceptor;

import feign.RequestInterceptor;
import feign.RequestTemplate;
import java.util.UUID;

public class XTokenRequestInterceptor implements RequestInterceptor {
    //此次请求的详细信息封装到了requestTemplate中,
    //想要对请求进行修改，只需要对requestTemplate进行修改
    @Override
    public void apply(RequestTemplate requestTemplate) {
        System.out.println("XTokenRequestInterceptor.....");
        requestTemplate.header("X-Token", UUID.randomUUID().toString());
    }
}
```

要启用拦截器，可以在配置文件中进行设置，也可以在容器中加入组件

这里采用组件方式

```

@Bean
XTokenRequestInterceptor xTokenRequestInterceptor(){
    return new XTokenRequestInterceptor();
}

```

在productcontroller中加入测试逻辑

为了不影响编码，使用原生的HttpServletRequest

```

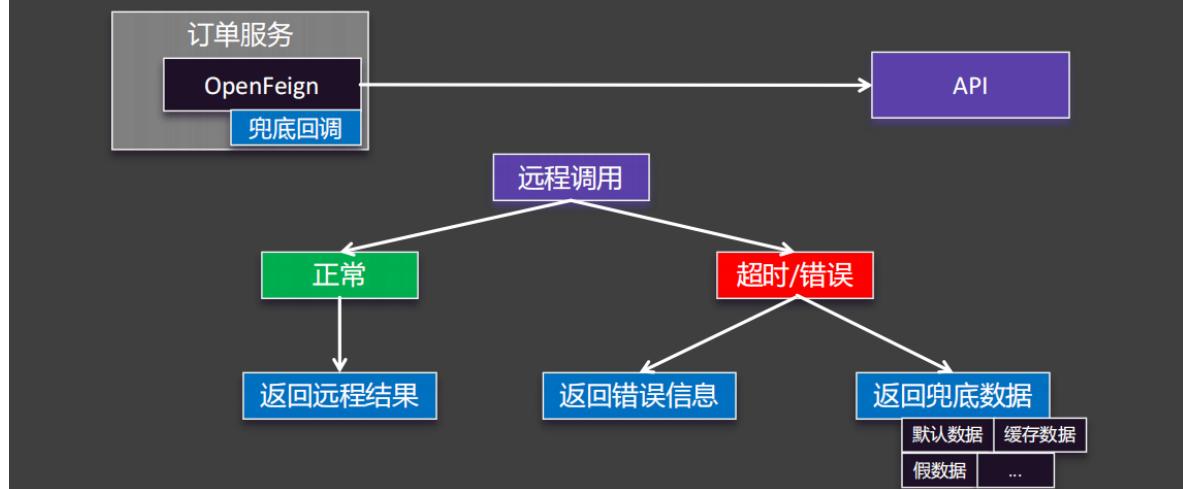
@GetMapping("/product/{id}")
public Product getProduct(@PathVariable("id") Long id,
                         HttpServletRequest request) {
    String head=request.getHeader("X-Token");
    System.out.println("token="+head);
    Product product=productService.getProductById(id);
    return product;
}

```

测试成功。

*Fallback(兜底返回)

- Fallback：兜底返回
 - 注意：此功能需要整合 Sentinel 才能实现



下面进行测试：

```

package com.atguigu.order.feign.fallback;
import java.math.BigDecimal;

import com.atguigu.order.feign.ProductFeignClient;
import com.atguigu.product.bean.Product;
import org.springframework.stereotype.Component;
//加入组件
@Component
public class ProductFeignClientFallback implements ProductFeignClient {
    @Override
    public Product getProductById(Long id) {
        System.out.println("兜底回调");
    }
}

```

```
        Product product = new Product();
        product.setId(id);
        product.setPrice(new BigDecimal("0"));
        product.setProductName("未知商品");
        product.setNum(0);
        return product;
    }
}
```

然后，在对应的FeignClient中的@FeignClient注解中加上

```
fallback=ProductFeignClientFallback.class
```

进行测试也非常简单，只需要把product服务关闭再发送请求。

但是，返回兜底数据需要配合sentinel，这边进行一个快速配置

导入sentinel依赖：

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

启用熔断：

```
feign:
  sentinel:
    enabled: true
```

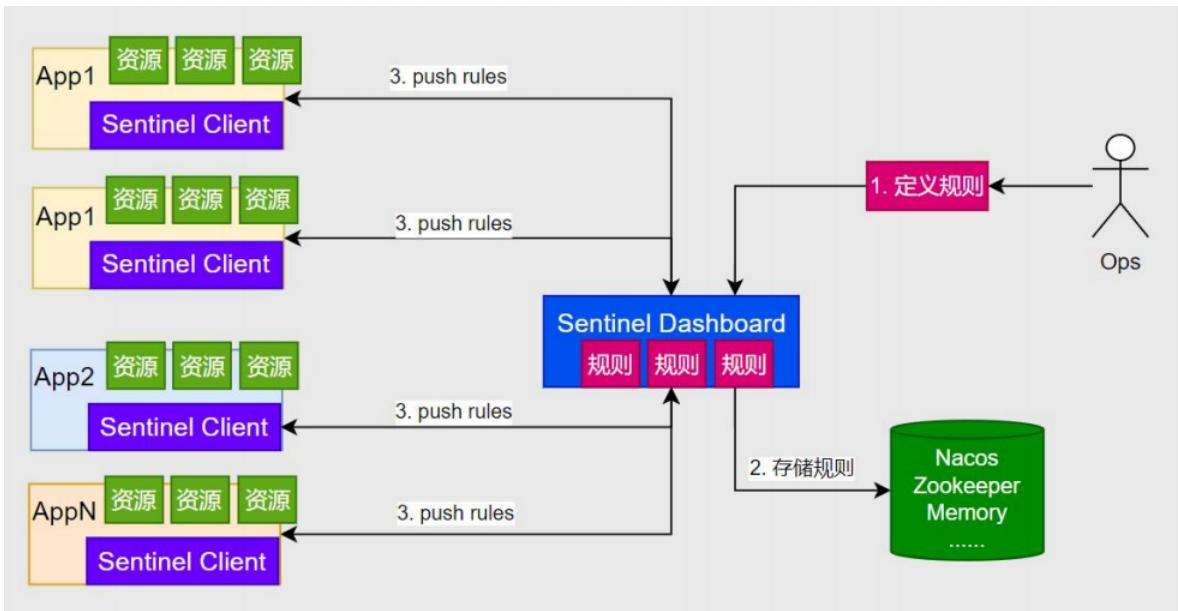
在进行远程调用发现兜底成功

Sentinel

引入

随着微服务的流行，服务和服务之间的稳定性变得越来越重要。Spring CloudAlibaba Sentinel 以流量为切入点，从流量控制、流量路由、熔断降级、系统自适应过载保护、热点流量防护等多个维度保护服务的稳定性。

架构原理：



资源&规则

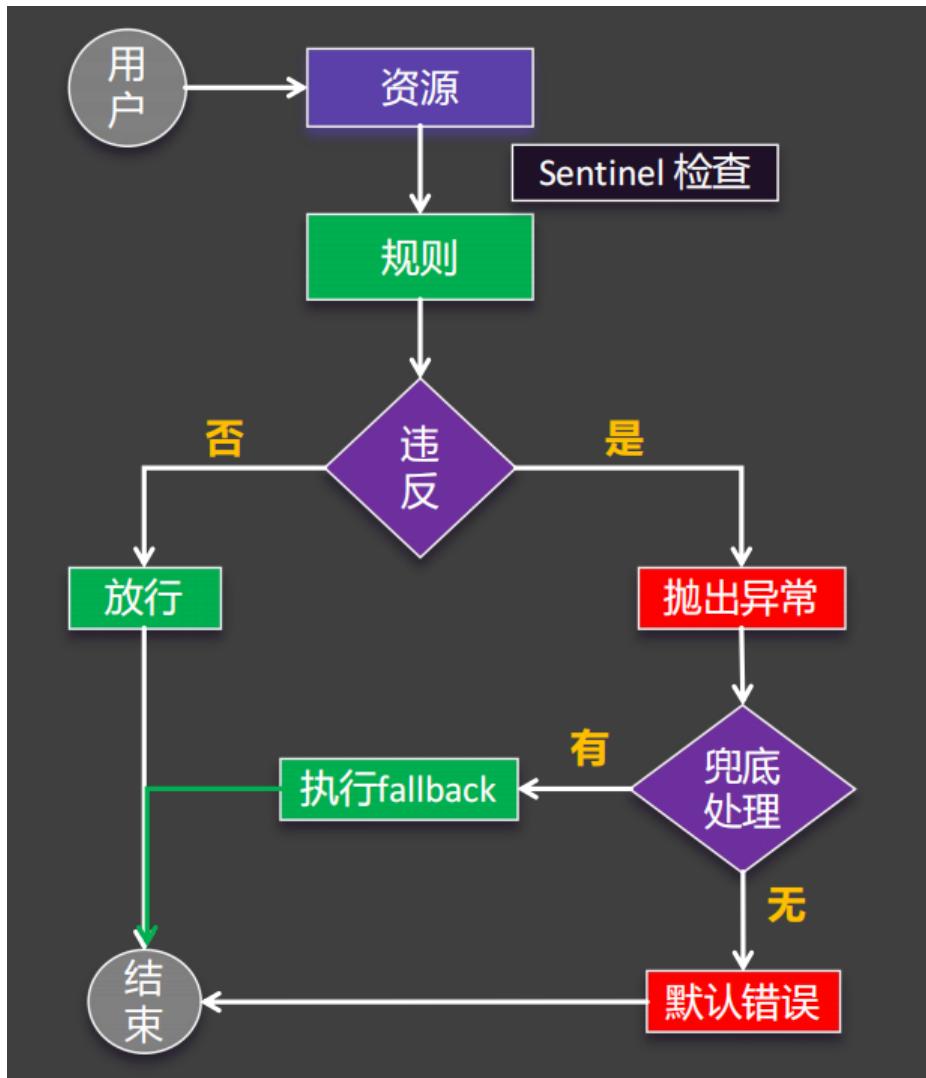
• 定义资源：

- 主流框架自动适配（Web Servlet、Dubbo、Spring Cloud、gRPC、Spring WebFlux、Reactor）；所有Web接口均为资源
- 编程式：SpfU API
- 声明式：@SentinelResource

• 定义规则：

- 流量控制（FlowRule）
- 熔断降级（DegradeRule）
- 系统保护（SystemRule）
- 来源访问控制（AuthorityRule）
- 热点参数（ParamFlowRule）

工作原理



整合/基础使用

cmd使用 `java -Dserver.port=9090 -jar sentinel-dashboard-1.8.8.jar` 在9090端口启动 sentinel dashboard。

在services模块中引入sentinel依赖

```

<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-sentinel</artifactId>
</dependency>

```

然后，每一个微服务都需要连接上sentinel控制台：

```

spring:
  cloud:
    sentinel:
      transport:
        dashboard: localhost:9090

```

而且sentinel默认的是懒加载机制，只有访问了请求，在控制台才会显示出应用的信息。所以为了方便起见

```
spring:  
  cloud:  
    sentinel:  
      eager:true
```

定义资源：

使用注解 `@SentinelResource(value=)` 给 `OrderServiceImpl` 中的方法定义为资源：

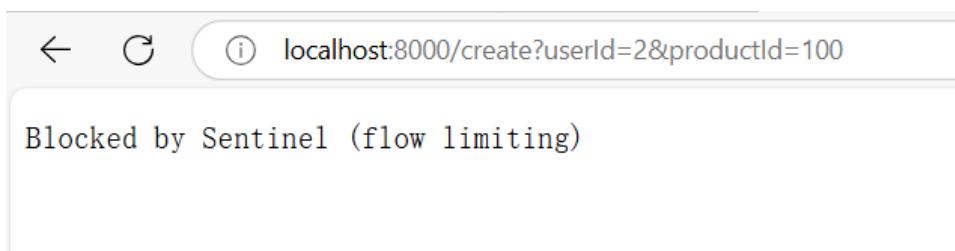
```
@SentinelResource(value = "createOrder")  
@Override  
public Order createorder(Long userId, Long productId) {
```

在启动项目并发送 (`http://localhost:8000/create?userId=2&productId=100`) 请求后，在 sentinel 控制台的 service-order 服务的簇点链路中可以发现 `create` 资源和 `createOrder` 资源，其中前者由于是 API，被 sentinel 自动识别为资源，而后者则是我们用注解手动添加的。

现在我们对 `create` 资源设置流控规则来进行演示



代表每秒最多一个请求，如果我们快速刷新以模拟快速请求：

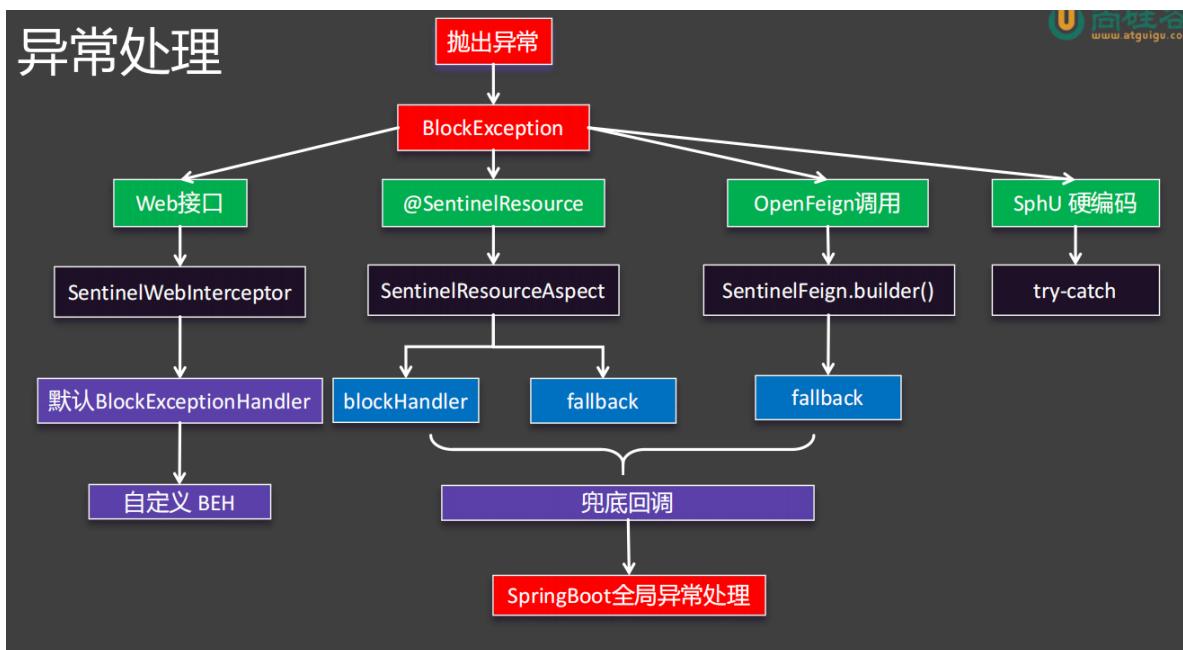


被阻塞的请求不会执行目标方法，那么就不会占用系统资源，即使高并发的场景，只有少量的目标方法执行并占用系统资源，那么就不会导致服务雪崩的问题。

*异常处理

但是现在请求被限制以后，返回的是默认 sentinel 的错误提示页，而我们需要给前端返回 json 数据，类似于：

```
{
    "code": "500",
    "msg": "流量超出限制",
    "data": ""
}
```



其中@SentinelResoruce标注在非controller层

web接口

DefaultBlockExceptionHandler是sentinel定义的专门用来处理springmvc的web接口异常的handler。它实现了BlockExceptionHandler接口。

所以想要自定义异常处理，就要自定义类来实现BlockExceptionHandler接口。

定义返回的对象R:

```

public class R {
    private Integer code;
    private String msg;
    private Object data;
    public static R ok() {
        R r = new R();
        r.setCode(200);
        return r;
    }
    public static R ok(String msg, Object data) {
        R r = new R();
        r.setCode(200);
        r.setMsg(msg);
        r.setData(data);
        return r;
    }
    public static R error() {
        R r = new R();
        r.setCode(500);
        return r;
    }
}
  
```

```

    }
    public static R error(Integer code, String msg) {
        R r = new R();
        r.setCode(code);
        r.setMsg(msg);
        return r;
    }
}

```

定义异常处理器：

```

@Component
public class MyBlockExceptionHandler implements BlockExceptionHandler {
    //springMVC底层整合了jackson提供的一个JSON工具
    private ObjectMapper objectMapper = new ObjectMapper();
    @Override
    public void handle(HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse,
            String s, BlockException e) throws Exception {
        httpServletResponse.setContentType("application/json; charset=utf-8");
        PrintWriter writer = httpServletResponse.getWriter();
        R error=R.error(500,s+"被Sentinel限制了，原因：" + e.getClass());
        String json = objectMapper.writeValueAsString(error);
        writer.write(json);
        writer.flush();
        writer.close();
    }
}

```

测试：

返回：

```
{
    "code": 500,
    "msg": "/create被Sentinel限制了，原因： class
com.alibaba.csp.sentinel.slots.block.flow.FlowException",
    "data": null
}
```

@SentinelResource

在@SentinelResource注解里，指定blockHandler、fallback、Defaultfallback中的一个，即可进行异常处理。如果都没有，将会向上抛出异常，由springboot定义异常处理机制，此时我们可以编写一个全局异常类对其进行处理。

我们就直接使用blockHandler指定

```

@SentinelResource(value = "createOrder",blockHandler = "createOrderFallback")
@Override
public Order createOrder(Long userId, Long productId) {
    //Product product =
    getProductFromRemoteWithLoadBalancerAnnotation(productId);
    //使用Feign代替RESTTemplate完成远程调用
}

```

```

        Product product=productFeignClient.getProductById(productId);
        Order order = new Order();
        order.setId(1L);
        order.setTotalAmount(( product.getPrice().multiply(new
BigDecimal(product.getNum()))));
        order.setUserId(userId);
        order.setNickname("zhangsan");
        order.setAddress("ss");
        order.setProductList(Arrays.asList(product));
        return order;
    }

//兜底回调
public Order createOrderFallback(Long userId, Long productId, Throwable
exception) {
    Order order = new Order();
    order.setId(0L);
    order.setTotalAmount(BigDecimal.ZERO);
    order.setUserId(userId);
    order.setNickname("未知用户");
    order.setAddress("异常信息: "+exception.getClass());
    return order;
}

```

在异常时返回：

```
{
    "id": 0,
    "totalAmount": 0,
    "userId": 2,
    "nickname": "未知用户",
    "address": "异常信息: class
com.alibaba.csp.sentinel.slots.block.flow.FlowException",
    "productList": null
}
```

注意：兜底回调方法里异常参数参数一定要定义为Throwable类型

```
public Order createOrderFallback(Long userId, Long productId, Throwable exception)
```

OpenFeign

详见OpenFeign的fallback

sphu.entry

`sphu` 包含了 try-catch 风格的 API。用这种方式，当资源发生了限流之后会抛出 `BlockException`。这个时候可以捕捉异常，进行限流之后的逻辑处理。示例代码如下：

```
// 1.5.0 版本开始可以利用 try-with-resources 特性（使用有限制）
// 资源名可使用任意有业务语义的字符串，比如方法名、接口名或其它可唯一标识的字符串。
try (Entry entry = Sphu.entry("resourceName")) {
    // 被保护的业务逻辑
    // do something here...
} catch (BlockException ex) {
    // 资源访问阻止，被限流或被降级
    // 在此处进行相应的处理操作
}
```

流控规则

新增流控规则

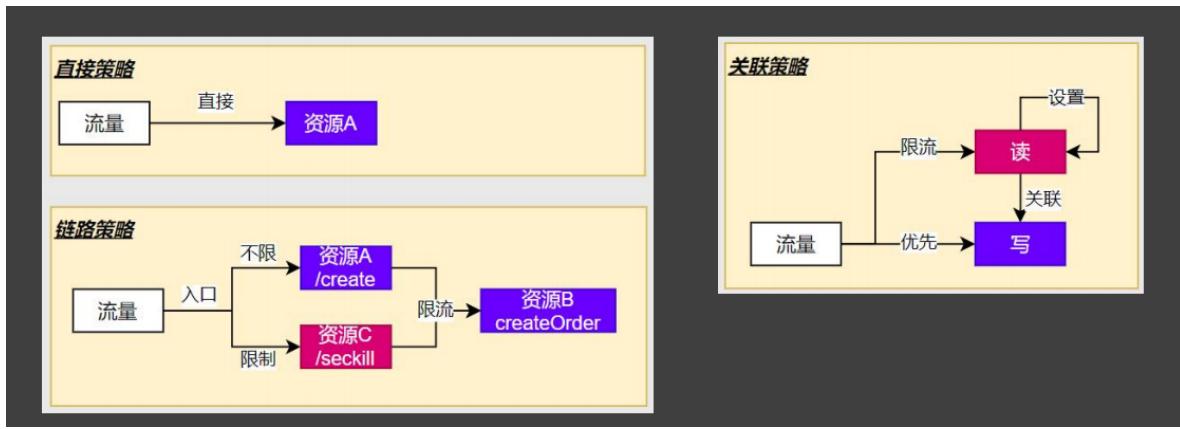
| | | | |
|------|--|------|---|
| 资源名 | /create | | |
| 针对来源 | default | | |
| 阈值类型 | <input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数 | 单机阈值 | 1 |
| 是否集群 | <input type="checkbox"/> | | |
| 流控模式 | <input checked="" type="radio"/> 直接 <input type="radio"/> 关联 <input type="radio"/> 链路 | | |
| 流控效果 | <input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待 | | |

[关闭高级选项](#)

- 针对来源：表示对何种来源的请求进行限制
- 阈值类型：QPS (Queries per second)：底层采用计数器的方式统计流量，性能高
并发线程数：线程池，性能低，适用于需要线程池的场景
- 是否集群：单机均摊：比如单机阈值设置为5，开了三个createservice，此模式下最多并发量为15
总体阈值：此模式下为5

流控模式：

调用关系包括调用方、被调用方；一个方法又可能会调用其它方法，形成一个调用链路的层次关系；有了调用链路的统计信息，我们可以衍生出多种流量控制手段。



直接策略: 只对此资源生效

链路策略: 不同的链路不同的策略

在配置里配置: `spring.cloud.sentinel.web-context-unify= false` 设置web上下文不统一，表示不统一链路。

增加一个链路:

```
@GetMapping("/seckill")
public Order seckill(@RequestParam("userId") Long userId,
                     @RequestParam("productId") Long productId) {

    Order order = orderservice.createOrder(userId, productId);
    order.setId(Long.MAX_VALUE);
    return order;
}
```

然后在sentinel中对createOrder进行流控，仅限制通过/seckill资源访问createOrder的链路:

编辑流控规则

| | | | |
|------------------------|--|------|---|
| 资源名 | createOrder | | |
| 针对来源 | default | | |
| 阈值类型 | <input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数 | 单机阈值 | 1 |
| 是否集群 | <input type="checkbox"/> | | |
| 流控模式 | <input type="radio"/> 直接 <input type="radio"/> 关联 <input checked="" type="radio"/> 链路 | | |
| 入口资源 | /seckill | | |
| 流控效果 | <input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待 | | |
| 关闭高级选项 | | | |

这样就做到了对不同链路实现不同的规则

关联策略：

举个例子：对数据库进行读写，我们需要写的优先级高，在写的请求量比较大时，才对读进行限流；如果写的请求量小的时候，不对读进行限流。

在controller中加入：

```
@GetMapping("/writeDb")
public String writeDb(){
    return "writeDb success"
}
@GetMapping("/readDb")
public String readDb(){
    return "readDb success"
}
```

| | |
|------------------------|--|
| 资源名 | /readDb |
| 针对来源 | default |
| 阈值类型 | <input checked="" type="radio"/> QPS <input type="radio"/> 并发线程数 |
| | 单机阈值 机阈值 |
| 是否集群 | <input type="checkbox"/> |
| 流控模式 | <input type="radio"/> 直接 <input checked="" type="radio"/> 关联 <input type="radio"/> 链路 |
| 关联资源 | /writeDb |
| 流控效果 | <input checked="" type="radio"/> 快速失败 <input type="radio"/> Warm Up <input type="radio"/> 排队等待 |
| 关闭高级选项 | |

那么，在write访问量超出阈值时，readDb无法正常访问，但writeDb本身没受影响。

流控效果

为了方便测试，引入apipost软件，并在MyBlockExceptionHandler中添加：

```
httpServletResponse.setStatus(429); //太多请求
```

- 直接失败：超过阈值的多余请求直接失败
- warmup：随时间逐渐增加可接受的访问量，经过预热时长后才达到设置的阈值
- 排队等待：设置的阈值为每秒放行的请求数量（**匀速**），超出数量的排队等待。还要设置最大等待时长。

Sentinel匀速排队等待策略是Leaky Bucket算法结合虚拟队列等待机制实现的。

注: `warmup` 和 `排队等待` 只支持【直接】的流控模式

| 属性 | 说明 | 默认值 |
|------------------------------|--|--------------------|
| <code>resource</code> | 资源名, 资源名是限流规则的作用对象 | |
| <code>count</code> | 限流阈值 | |
| <code>grade</code> | 限流阈值类型, QPS 模式 (1) 或 并发线程数模式 (0) | QPS 模式 |
| <code>limitApp</code> | 流控针对的调用来源 | default, 代表不区分调用来源 |
| <code>strategy</code> | 调用关系限流策略: 直接、链路、关联 | 直接 (根据资源本身) |
| <code>controlBehavior</code> | 流控效果 (直接拒绝/WarmUp/匀速+排队等待) , 不支持按调用关系限流 | 直接拒绝 |
| <code>clusterMode</code> | 是否集群限流 | 否 |

熔断规则

断路器控制原理

熔断降级 (Circuit Breaker Pattern) 是分布式系统中用于提高系统稳定性和容错能力的一种设计模式。它通过监控服务调用的成功率或响应时间，在检测到服务出现异常时，暂时阻止对该服务的进一步调用，从而避免故障扩散和雪崩效应。

- 熔断器的工作原理

熔断器的状态通常分为三种：

1. 关闭 (Closed) :

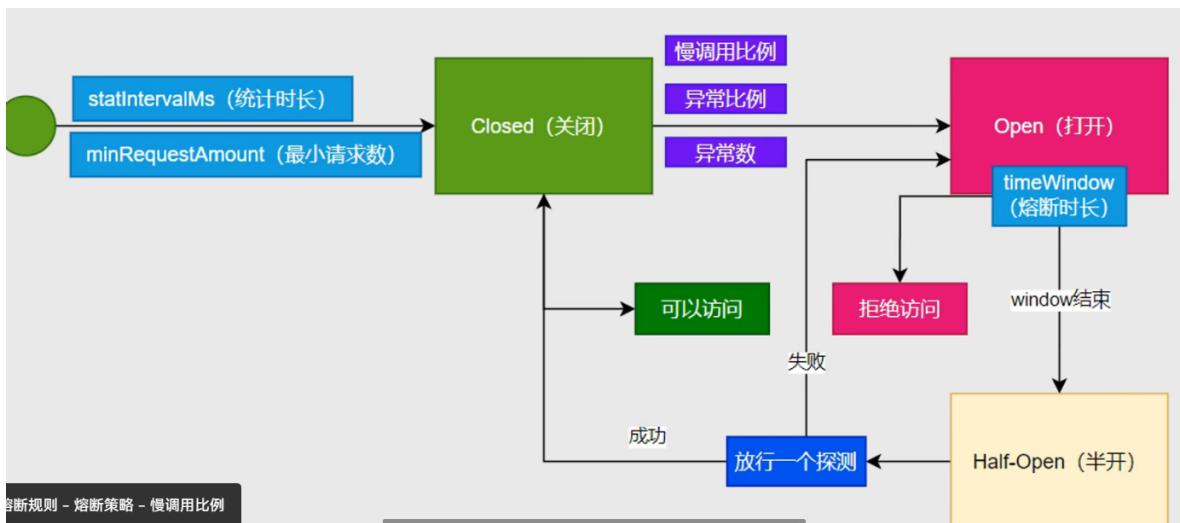
- 正常情况下，熔断器处于关闭状态，允许请求通过并转发给目标服务。
- 在此状态下，熔断器会统计失败的请求数量或平均响应时间。
- 如果在一定时间内失败率达到设定的阈值（例如超过50%的请求失败），则熔断器将切换到打开状态。

2. 打开 (Open) :

- 当熔断器处于打开状态时，所有对目标服务的请求都会立即返回错误，而不会实际发起请求。
- 这种快速失败的方式可以减少不必要的等待时间和资源消耗。
- 经过一段冷却时间（例如30秒）后，熔断器会进入半开状态。

3. 半开 (Half-Open) :

- 半开状态是一种试探性的恢复阶段。
- 在这个状态下，熔断器允许部分请求通过以测试目标服务是否恢复正常。
- 如果这些请求成功，则熔断器切换回关闭状态；如果仍然失败，则再次切换到打开状态，并重新计时冷却期。



熔断策略

远程调用时为了防止product异常对order产生影响，我们在order配置熔断策略

新增熔断规则

| | | | | |
|--|---|------|---------------|---|
| 资源名 | GET: http://service-product/product/{id} | | | |
| 熔断策略 | <input checked="" type="radio"/> 慢调用比例 <input type="radio"/> 异常比例 <input type="radio"/> 异落数 | | | |
| 最大 RT | RT (毫秒) | 比例阈值 | 取值 [0.0, 1.0] | |
| 熔断时长 | 熔断时长(s) | s | 最小请求数 | 5 |
| 统计时长 | 1000 | ms | | |
| <input type="button" value="新增并继续添加"/> <input type="button" value="新增"/> <input type="button" value="取消"/> | | | | |

最大RT: 最大响应时间 (response time)

熔断时长: 熔断器打开的时长

先说明有熔断和无熔断时远程调用发生异常/超时时的区别:

没有熔断规则: 每次都会把请求发送出去，超时或错误回调用兜底机制

有熔断规则: 远程出问题后，一段时间内直接进行兜底机制

- 慢比例调用
- 异常比例
- 异落数

| 属性 | 说明 | 默认值 |
|--------------------|---|---------|
| resource | 资源名, 必填 | |
| grade | 熔断策略, 支持慢调用比例/异常比例/异常数策略 | 慢调用比例 |
| count | 慢调用比例模式下为慢调用临界 RT (超出该值计为慢调用) ; 异常比例/异常数模式下为对应的阈值 | |
| timeWindow | 熔断时长, 单位为 s | |
| minRequestAmount | 熔断触发的最小请求数, 请求数小于该值时即使异常比率超出阈值也不会熔断 (1.7.0 引入) | 5 |
| statIntervalMs | 统计时长 (单位为 ms) , 如 60*1000 代表分钟级 (1.8.0 引入) | 1000 ms |
| slowRatioThreshold | 慢调用比例阈值, 仅慢调用比例模式有效 (1.8.0 引入) | |

热点规则

相当于是更细节的流量控制，除了能对资源进行控制以外，还能精确到参数满足什么规则后进行限制。

热点就可以理解为我们经常要访问的数据

下面进行一个示例进行演示：

需求1：每个用户秒杀 QPS 不得超过 1 (秒杀下单 userId 级别)

效果：携带此参数的参与流控，不携带不流控

需求2：6号用户是vvip，不限制QPS（例外情况）

需求3：666号是下架商品，不允许访问

注意：目前 Sentinel 自带的 adapter 仅 Dubbo 方法埋点带了热点参数，其它适配模块（如 Web）默认不支持热点规则，可通过自定义埋点方式指定新的资源名并传入希望的参数。注意自定义埋点的资源名不要和适配模块生成的资源名重复，否则会导致重复统计。

```

@GetMapping("/seckill")
@SentinelResource(value = "seckill-order", fallback = "seckillFallback")
public Order seckill(@RequestParam("userId") Long userId,
                      @RequestParam("productId") Long productId) {

    Order order = orderservice.createOrder(userId, productId);
    order.setId(Long.MAX_VALUE);
    return order;
}

public Order seckillFallback(@RequestParam("userId") Long userId,
                           @RequestParam("productId") Long productId,
                           BlockException exception) {
    System.out.println("seckillFallback....");
    Order order = new Order();
    order.setId(productId);
    order.setUserId(userId);
    order.setAddress("异常信息: "+exception.getMessage());
    return order;
}

```

给seckill起名为seckill-order，并启用兜底

编辑热点规则

资源名

seckill-order

限流模式 QPS 模式

参数索引

d

单机阈值

1

统计窗口时长

1

秒

是否集群



参数例外项

参数类型

▼

参数值

例外项参数值

限流阈值

限流阈值

+ 添加

参数值

参数类型

限流阈值

操作

6

long

10000000

删除

[关闭高级选项](#)

编辑热点规则

| | | | |
|------------------------|---|--------|-----------------------------------|
| 资源名 | seckill-order | | |
| 限流模式 | QPS 模式 | | |
| 参数索引 | 1 | | |
| 单机阈值 | 1000000 | 统计窗口时长 | 1 秒 |
| 是否集群 | <input type="checkbox"/> | | |
| 参数例外项 | | | |
| 参数类型 | <input type="button" value="例外项参数值"/> <input type="button" value="限流阈值"/> <input type="button" value="限流阈值"/> <input type="button" value="添加"/> | | |
| 参数值 | 666 | 参数类型 | long |
| | 0 | 限流阈值 | <input type="button" value="删除"/> |
| 关闭高级选项 | | | |

如上两个热点参数限流，可以实现要求的效果

| 属性 | 说明 | 默认值 |
|-------------------|---|--------|
| resource | 资源名，必填 | |
| count | 限流阈值，必填 | |
| grade | 限流模式 | QPS 模式 |
| durationInSec | 统计窗口时间长度（单位为秒），1.6.0 版本开始支持 | 1s |
| controlBehavior | 流控效果（支持快速失败和匀速排队模式），1.6.0 版本开始支持 | 快速失败 |
| maxQueueingTimeMs | 最大排队等待时长（仅在匀速排队模式生效），1.6.0 版本开始支持 | 0ms |
| paramIdx | 热点参数的索引，必填，对应 SphU.entry(xxx, args) 中的参数索引位置 | |
| paramFlowItemList | 参数例外项，可以针对指定的参数值单独设置限流阈值，不受前面 count 阈值的限制。仅支持基本类型和字符串类型 | |
| clusterMode | 是否是集群参数流控规则 | false |
| clusterConfig | 集群流控相关配置 | |

授权规则和系统规则

参考：<https://github.com/alibaba/Sentinel/wiki>

没什么用

持久化配置

目前定义的规则都是存储在内存中的，每一次启动项目都需要重新配置规则。

搭配nacos或数据库做规则的持久化

我们以service-order模块下的readDb为例，其在sentinel中的资源名为 /readdb

- 给本微服务引入依赖：

```
<dependency>
    <groupId>com.alibaba.csp</groupId>
    <artifactId>sentinel-datasource-nacos</artifactId>
</dependency>
```

- 在yml中配置：

```
spring:
  cloud:
    datasource:
      ds1: # 数据源名称
        nacos:
          server-addr: localhost:8848 # Nacos 地址
          data-id: service-order # 流控规则存储的 Data ID
          group-id: DEFAULT_GROUP           # Nacos 分组
          data-type: json
          rule-type: flow                  # 规则类型，这里是流控规则
```

1. `datasource:`

- 定义了 Sentinel 规则的数据源配置。通过配置不同的数据源，可以实现规则的持久化和动态更新。

2. `ds1:`

- 这是一个自定义的数据源名称，你可以根据需要命名。这个名称用于标识不同的数据源配置。如果有多个规则来源（如流控规则、降级规则等），可以通过不同的名字来区分它们。

3. `nacos:`

- 指定使用 Nacos 作为规则的持久化存储。这意味着 Sentinel 将从 Nacos 中读取并同步规则。

4. `server-addr: 127.0.0.1:8848`

- Nacos 服务器的地址。这是 Nacos 提供服务的 IP 和端口号。在这个例子中，Nacos 服务运行在本地机器上，监听 8848 端口。

5. `data-id: ${spring.application.name}-sentinel`

- `data-id` 是 Nacos 中的一个唯一标识符，用来指定某个具体的配置文件。
`${spring.application.name}` 是一个占位符，表示当前 Spring 应用的名称。因此，如果应用的名字是 `myapp`，那么完整的 `data-id` 就会是 `myapp-sentinel`。这样做的好处是可以为不同的应用提供独立的规则配置。

6. `group-id: DEFAULT_GROUP`

- `group-id` 在 Nacos 中用于组织配置。它可以将多个 `data-id` 归类在一起，便于管理和查找。这里设置为 `DEFAULT_GROUP` 表示使用默认分组。

7. `rule-type: flow`

- 指定了规则的类型。在这个例子中，`flow` 表示这是一个流控规则（即限流规则）。Sentinel 支持多种类型的规则，包括流控规则、降级规则、系统规则等。通过设置不同的 `rule-`

`type`，可以从 Nacos 加载相应类型的规则。

工作原理:

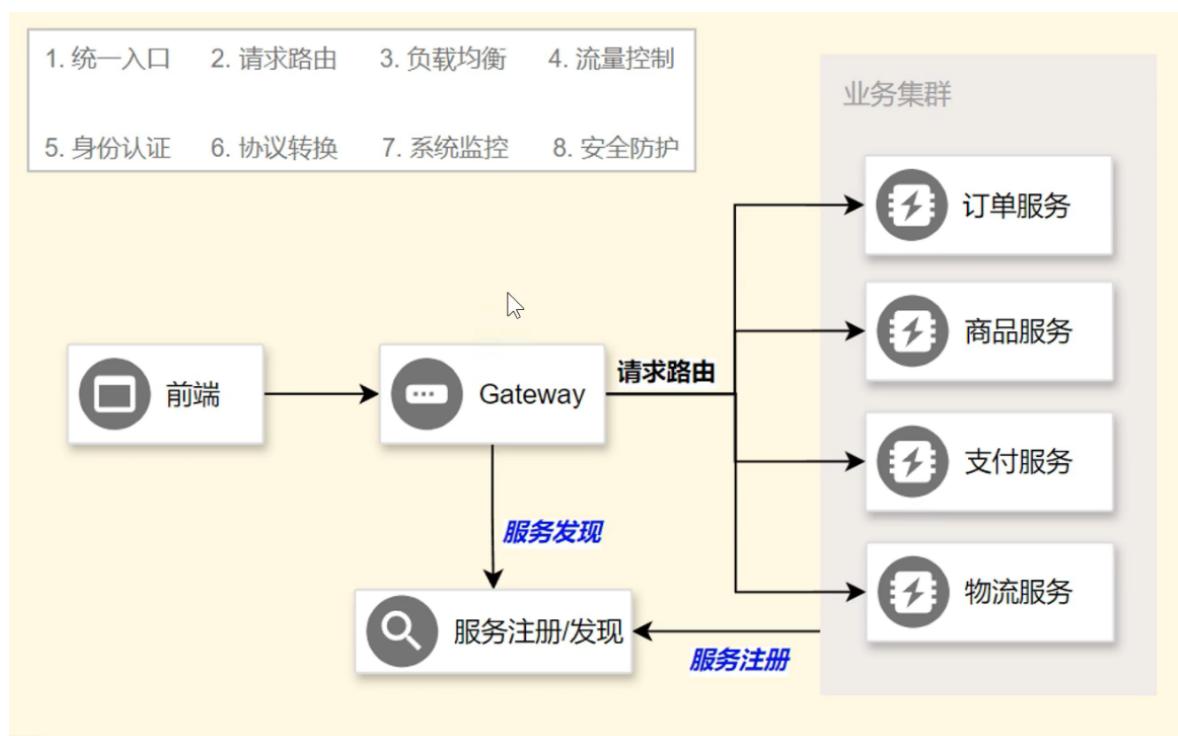
当你配置了上述内容后，Sentinel 会按照以下步骤工作：

1. **初始化**: 当你的应用启动时，Sentinel 会根据配置尝试连接到指定的 Nacos 服务器 (`server-addr`)。
2. **加载规则**: Sentinel 会从 Nacos 中读取与 `data-id` 和 `group-id` 匹配的配置文件，并将其转换为相应的规则（在这个例子中是流控规则）。
3. **应用规则**: 读取到的规则会被应用到 Sentinel 中，从而控制流量、保护资源。
4. **动态更新**: 如果 Nacos 中的相关配置发生变化，Sentinel 可以自动检测到这些变化，并实时更新其内部的规则，而无需重启应用。

- 在nacos对应namespace的对应group中创建配置（采用json格式）

```
[  
 {  
   "resource": "/readDb",  
   "limitApp": "default",  
   "grade": 1,  
   "count": 1,  
   "strategy": 0,  
   "controlBehavior": 0,  
   "clusterMode": false  
 }  
]
```

Gateway



作为系统的入口点，它负责接收所有的客户端请求，并将这些请求路由到后端的适当微服务。

创建网关

在cloud-demo下新建gateway模块(maven管理的java项目)

引入依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-loadbalancer</artifactId>
    </dependency>
</dependencies>
```

进行基础配置：

```
spring:
  application:
    name: gateway
  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848
  server:
    port: 80
```

编写主程序：

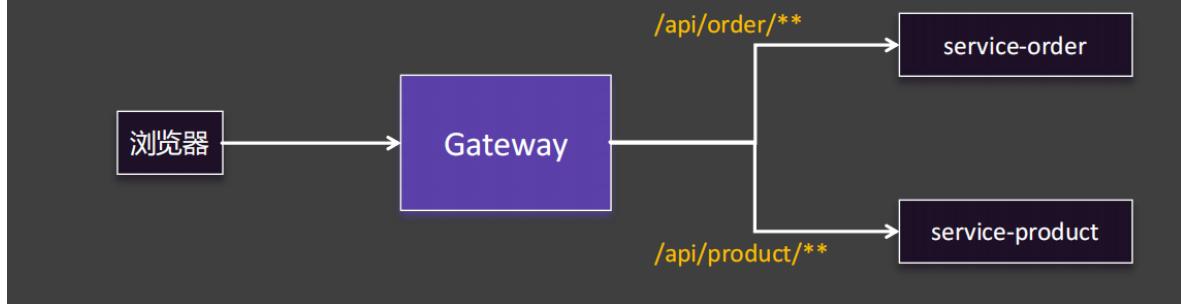
```
@EnableDiscoveryClient
@SpringBootApplication
public class GatewayMainApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayMainApplication.class, args);
    }
}
```

路由

规则配置

需求：

- 1. 客户端发送 `/api/order/**` 转到 `service-order`
- 2. 客户端发送 `/api/product/**` 转到 `service-product`
- 3. 以上转发有负载均衡效果



cloud提供了两种方式配置路由规则：

- 第一种：用配置文件配置路由规则

```

private String id;
private @NotEmpty @Valid List<PredicateDefinition> predicates = new
ArrayList();
private @Valid List<FilterDefinition> filters = new ArrayList();
private @NotNull URI uri;
private Map<String, Object> metadata = new HashMap();
private int order = 0;
  
```

以上是spring.cloud.gateway.routes的配置项

我们进行较为简单的配置：

```

spring:
  cloud:
    gateway:
      routes:
        - id: order-route #路由名字
          uri: lb://service-order #lb: 负载均衡。
          predicates: #断言，指定遵守哪种规则就转给订单
            - Path=/api/order/**
        - id: product-route
          uri: lb://service-product
          predicates:
            - Path=/api/product/**
  
```

此外，我们还需要给controller和feign添加**在gateway配置中设置的基准路径**（feign是加上远程调用的目标的基准路径）

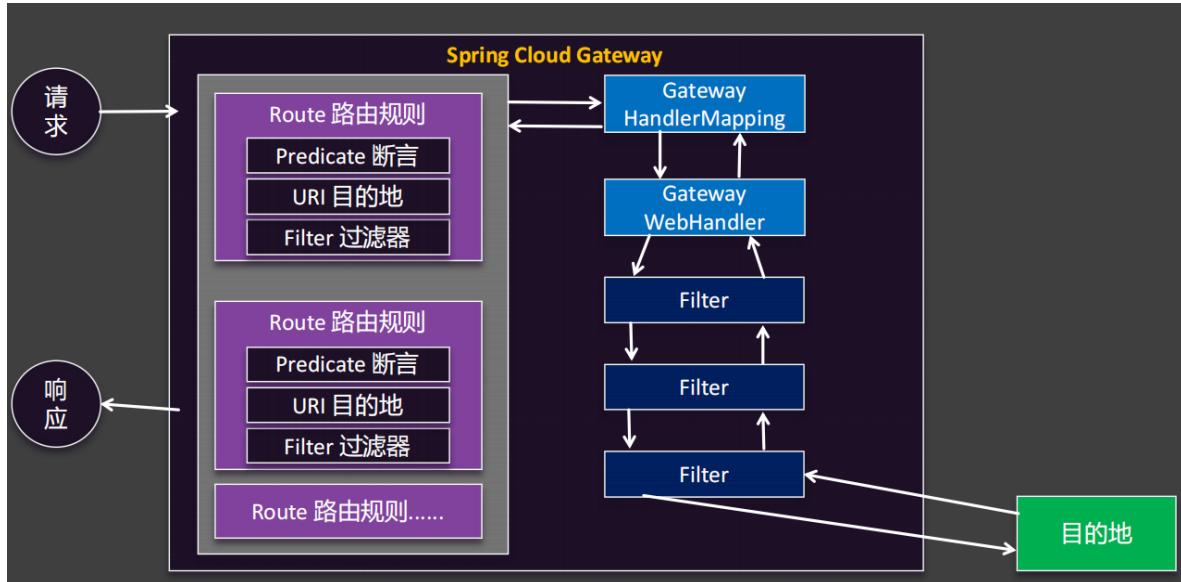
在controller上：

```
@RequestMapping("/api/order")
```

在feign上：（不允许标注@RequestMapping在feign上，所以我们给具体方法前面加上远程调用的目标的基准路径）

```
@GetMapping("/api/product/product/{id}")
```

工作原理



其中断言是从上往下进行匹配的，上面的匹配上了，就不会再继续向下匹配。但是还有一个order配置参数，用来指定断言匹配的顺序。

还有一个metadata配置参数，用来配置个人配置

在接下来，我们看一下如何使用filter配置参数，以及断言除了Path外的其他指定。

断言

断言的使用

所有断言：

参考spring官方文档[Route Predicate Factories :: Spring Cloud Gateway](#)

也可以查看RoutePredicateFactory接口下AbstractRoutePredicateFactory抽象类的实现类，断言的参数名就是这些实现类名去掉后面的RoutePredicateFactory后缀。而这些实现类的args可以参考他们的构造器中的config中定义的参数来写。

先介绍以下断言的短写法和长写法：以Path为例

短写法：

```
predicates:  
- Path=/api/product/**
```

长写法：

```

predicates:
- name: Path
  args:
    patterns: /api/product/**
  matchTrailingSlash: true #表示/api/和/api是一个路径

```

| 名 | 参数(个数/类型) | 作用 |
|-----------------------|---------------------|--|
| After | 1/datetime | 在指定时间之后 |
| Before | 1/datetime | 在指定时间之前 |
| Between | 2/datetime | 在指定时间区间内 |
| Cookie | 2/string,regexp | 包含cookie名且必须匹配指定值 |
| Header | 2/string,regexp | 包含请求头且必须匹配指定值 |
| Host | N/string | 请求host必须是指定枚举值 |
| Method | N/string | 请求方式必须是指定枚举值 |
| Path | 2/List<String>,bool | 请求路径满足规则,是否匹配最后的/ |
| Query | 2/string,regexp | 包含指定请求参数 |
| RemoteAddr | 1>List<String> | 请求来源于指定网络域(CIDR写法) |
| Weight | 2/string,int | 按指定权重负载均衡 |
| XForwarded RemoteAddr | 1>List<string> | 从X-Forwarded-For请求头中解析请求来源,并判断是否来源于指定网络域 |

下面以Query为例演示:

```

- id: bing-route
  uri: https://cn.bing.com/
  predicates:
    - name: Path
      args:
        patterns: /search
    - name: Query
      args:
        param: q
        regexp: haha

```

因为我们给网关指定了80端口,所以接下来我们在访问 `localhost/search?q=haha` 的时候相当于在 bing里搜索了haha。

自定义断言工厂

比如:

```
- id: bing-route
```

```

uri: https://cn.bing.com/
predicates:
- name: Path
  args:
    patterns: /search
- name: Query
  args:
    param: q
    regexp: haha
- vip=user,muite
(- name: Vip
  args:
    param: user
    value: muite)

```

需求是只有携带参数中的用户名为muite (VIP用户) 才转发到搜索，其他人不转发。

实现：

参考已实现的断言工厂类，这里参考QueryRoutePredicateFactory

在gateway模块下新建predicate包，创建VipRoutePredicateFactory继承AbstractRoutePredicateFactory：

```

//别忘了放进容器中
@Component
public class VipRoutePredicateFactory extends
AbstractRoutePredicateFactory<VipRoutePredicateFactory.Config> {

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

    //对请求是否满足断言要求进行判断
    //localhost:/search?q=haha&user=muite满足要求
    @Override
    public Predicate<ServerWebExchange> apply(Config config) {
        return new GatewayPredicate() {
            @Override
            public boolean test(ServerWebExchange serverWebExchange) {
                //获取当前请求
                ServerHttpRequest request = serverWebExchange.getRequest();
                //拿到参数
                String first = request.getQueryParams().getFirst(config.param);
                //使用StringUtil工具判断参数值是否符合规则
                return StringUtil.hasText(first)&&first.equals(config.value);
            }
        };
    }

    //指定短写法中参数值书写的前后顺序
    @Override
    public List<String> shortcutFieldOrder() {
        return Arrays.asList("param", "value");
    }
}

```

```

/*
* config代表可以配置的参数
*/
@validated
public static class Config {
    @NotEmpty
    private String param;

    @NotEmpty
    private String value;

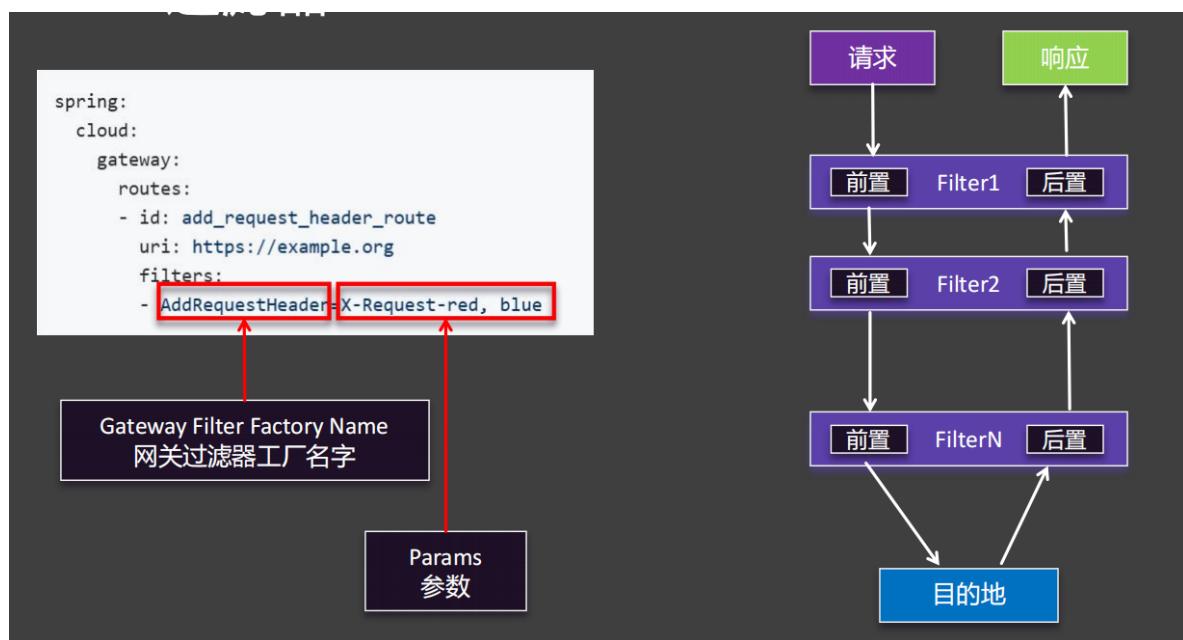
    public @NotEmpty String getParam() {
        return param;
    }
    public void setParam(@NotEmpty String param) {
        this.param = param;
    }
    public @NotEmpty String getValue() {
        return value;
    }
    public void setValue(@NotEmpty String value) {
        this.value = value;
    }
}

}

```

过滤器

参考[GatewayFilter Factories :: Spring Cloud Gateway](#)



基本实现

以RewritePathGateFilterFactory为例

在以前，我们想要通过网关发送请求，网关会照抄不误的把前端提交的地址交给目的地。详见此笔记的[Gateway--路由--规则配置](#) 中我们给controller和feign做的路径修改。

所以我们通过RewritePathGateFilterFactory实现如下效果：



想要实现这个功能我们就给网关添加filter

```
filters: #利用正则表达式将形如/api/order/...变为/...
- RewritePath=/api/order/?(<segment>.*), /${segment}
```

完毕

默认filter

上面的filter仅仅是给order进行配置的，如果product需要，还需要另行配置。

springcloud还有一个默认filter

配置如下：

```
spring:
  cloud:
    gateway:
      default-filters:
        - 具体filter
```

这是给所有路由都进行的filter配置

全局filter

我们来自定义一个可以应用于所有路由的RtGlobalFilter，用来记录请求耗时（响应时间）

全局filter不需要在配置文件中进行配置，而是直接将组件放入容器中

在网关模块下新建filter.RtGlobalFilter

```
@Slf4j
@Component
public class RtGlobalFilter implements GlobalFilter, Ordered{
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {
        ServerHttpRequest request = exchange.getRequest();
        ServerHttpResponse response = exchange.getResponse();
        String uri = request.getURI().toString();
        long start = System.currentTimeMillis();
        log.info("请求【{}】开始:时间: {}", uri, start);
        //=====以上是前者逻辑=====
        Mono<Void> filter = chain.filter(exchange)
            .doFinally((result) -> {
                //=====以下是后置逻辑=====
                long end = System.currentTimeMillis();
                log.info("请求【{}】结束: 时间: {},耗时: {}ms", uri, end, end - start);
            });
        //放行
        //因为响应式编程是全异步处理,
        //所以直接在这下面写的代码并不是等目标方法执行结束后执行的
        //所以并不能称之为后置逻辑, 后置逻辑可以用上面的链式编程实现
        return filter;
        //return filter 表示当前过滤器已经完成其职责,
        // 并将控制权交给网关框架, 让框架继续处理后续的逻辑(下一个过滤器或目标操作)。
    }
    //设置执行顺序
    @Override
    public int getOrder() {
        return 0;
    }
}
```

自定义过滤器工厂

举例:

需求:

每一个请求发出后，添加一个自定义的响应头，这个响应头是一个一次性令牌（uuid或JWT），但是也不是每一个路由都需要这个过滤器，这时候我们就可以自定义一个过滤器工厂。

参考AddResponseHeaderGatewayFilterFactory过滤器工厂我们来实现自定义工厂：

继承AbstractNameValueGatewayFilterFactory并进行重写

```
//每次响应之前，添加一个一次性令牌，支持uuid, jwt等各种格式
//我们想要响应头名为X-Response-Token，响应体为不同形式的令牌
```

```
@Component
public class OnceTokenGatewayFilterFactory extends
AbstractNameValueGatewayFilterFactory {
```

```

//这里的config是获取了我们在配置文件中配置的key和value
//apply 方法是核心方法，用于创建实际的过滤器实例。
@Override
public GatewayFilter apply(NameValueConfig config) {
    return new GatewayFilter() {
        //filter 方法是过滤器的核心逻辑，用于处理请求和响应。
        //参数 exchange 表示当前的 HTTP 请求和响应上下文。
        //参数 chain 表示过滤器链，用于将请求传递到下一个过滤器或目标服务。
        @Override
        public Mono<Void> filter(ServerWebExchange exchange,
GatewayFilterChain chain) {
            //chain.filter(exchange):将请求放行到下一个过滤器或目标服务进行处理。
            //返回的是一个 Mono<Void>, 表示异步操作流。
            //.then(Mono.fromRunnable(...)): .then 是响应式编程中的一个操作符，用
于在前一个操作完成后执行后续操作。
            //Mono.fromRunnable 用于封装一个没有返回值的操作（即副作用操作），在这里
用来修改响应头。
            return chain.filter(exchange).then(Mono.fromRunnable(()->{
                ServerHttpResponse response = exchange.getResponse();
                HttpHeaders headers = response.getHeaders();
                String value=config.getValue();
                if("uuid".equalsIgnoreCase(value)){
                    value= UUID.randomUUID().toString();
                }
                if("jwt".equalsIgnoreCase(value)){
                    //应该用代码生成jwt, 这里只作功能演示
                    value="eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI
6IkpvAg4gRG9lIiwiYWRTaw4iOnRydWUsImhlhdCI6MTUxNjIzOTAyMn0.SfIKxwRJSMeKKF2QT4fwpMe
Jf36POk6yJV_adQssw5c";
                }
                headers.add(config.getName(),value);
            }));
            //最终，chain.filter(exchange) 处理完成，并通过 .then 执行后置逻辑（添
加响应头）。
            //整个过滤器的执行结果以 Mono<Void> 的形式返回，表示异步操作完成。
        }
    };
}
}

```

跟断言工厂一样，filter名也是去掉后面的GatewayFilterFactory

全局跨域

[CORS Configuration :: Spring Cloud Gateway](#)

在前后分离的开发模式下，前端给后端发送请求，经常会出现跨域问题，之前我们是通过 @CrossOrigin解决的。

而内置的CorsFilter只能在本微服务下实现跨域

我们可以在网关设置统一跨域

```
spring:
  cloud:
    gateway:
      globalcors:
        cors-configurations:
          '//**': #对所有请求进行配置
            # 定义允许访问资源的来源（origin）。Origin 是指发起请求的客户端地址，例如
            http://localhost:3000。
            allowed-origin-patterns: '*'
            #允许的 HTTP 头信息
            allowed-headers: '*'
            #允许的 HTTP 请求方法
            allowed-methods: '*'
```

除了全局跨域，还支持只设置在某一个路由下允许跨域：

这是官方文档示例：

```
spring:
  cloud:
    gateway:
      routes:
        - id: cors_route
          uri: https://example.org
          predicates:
            - Path=/service/**
          metadata:
            cors:
              allowedOrigins: '*'
              allowedMethods:
                - GET
                - POST
              allowedHeaders: '*'
              maxAge: 30
```

思考

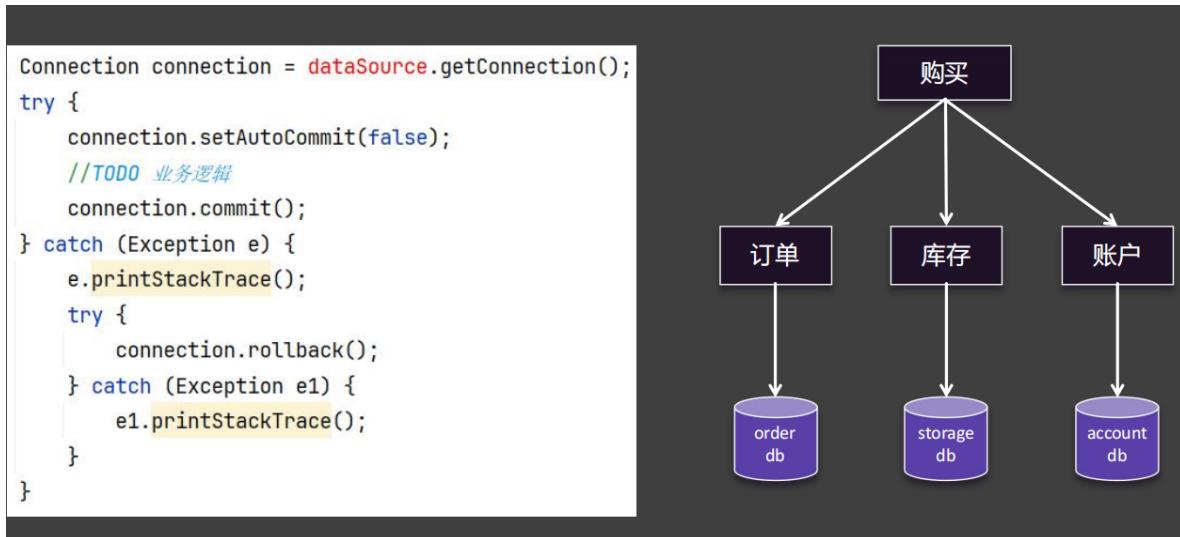
微服务之间的调用经过网关吗？

以上的示例都是不经过网关的，一个微服务去注册中心获取另一个微服务的地址，然后直接进行远程调用。

想要经过网关也可以，在上面的例子中，将ProductFignClient的@FeignClient注解中的value改为网关的服务名；并在方法的Mapping的路径中加上目标微服务的基准路径（比如/api/product）

但是，网关是用来和前端进行对接的，后端的微服务之间互相调用不需要经过网关。

Seata



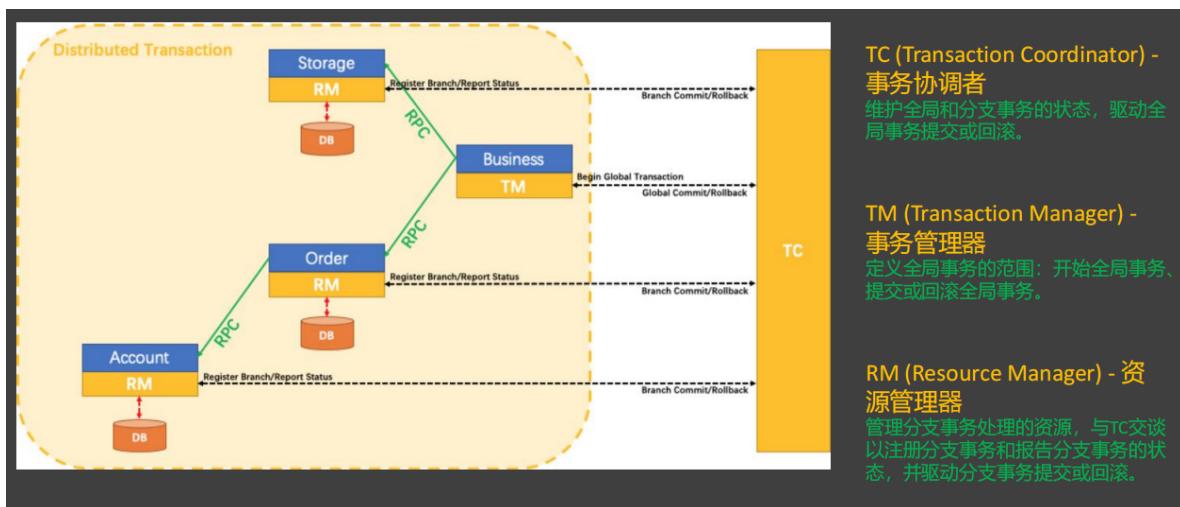
分布式事务的产生原因：

举个例子，在JDBC中，我们会对事务进行commit和rollback的控制，但这种控制仅限于一条连接中执行的逻辑。

当我们需要购买商品时，购买的操作可能涉及不同的微服务，每个微服务的数据源又可能是不一样的，这就会出现一个问题：比如当我的账户操作出现异常时，订单和库存由于和账户不在一条连接中，所以它们二者可能已经完成了提交，没办法和账户进行统一的commit或rollback操作。

Seata就提供了在分布式系统下保证多个数据库一起提交回滚，从而达到数据一致性状态的一站式解决方案。

原理



每一个微服务事务和全局事务启动时，都要向TC去注册自己的事务，而且每一个事务的状态也要汇报给TC。

如果某一个事务出现问题，TC会要求这个事务进行回滚，而且也会根据情况要求其他事务的回滚。

整合Seata

首先要启动seata服务器 (TC) , 然后给每个微服务引入seata的客户端

启动seata服务器：

进入seata的bin目录打开cmd, 执行 `seata-server.bat` 命令

其中7091端口是seata dashboard; 8091端口是默认用于 Seata Server 的 gRPC 通信端口, 是 TM (事务管理器) 和 RM (资源管理器) 与 TC (事务协调器) 之间通信的关键端口。

在每个微服务中引入seata客户端：

也就是引入seata依赖

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-seata</artifactId>
    <version>2023.0.3.2</version>
</dependency>
```

除此之外, 还要让每个微服务连接上seata服务器

所以在每个微服务的resources下创建file.conf , 在其中添加简单配置：

```
service {
    #transaction service group mapping
    vgroupMapping.default_tx_group = "default"
    #only support when registry.type=file, please don't set multiple addresses
    default.groupList = "127.0.0.1:8091"
    #degrade, current not support
    enableDegrade = false
    #disable seata
    disableGlobalTransaction = false
}
```

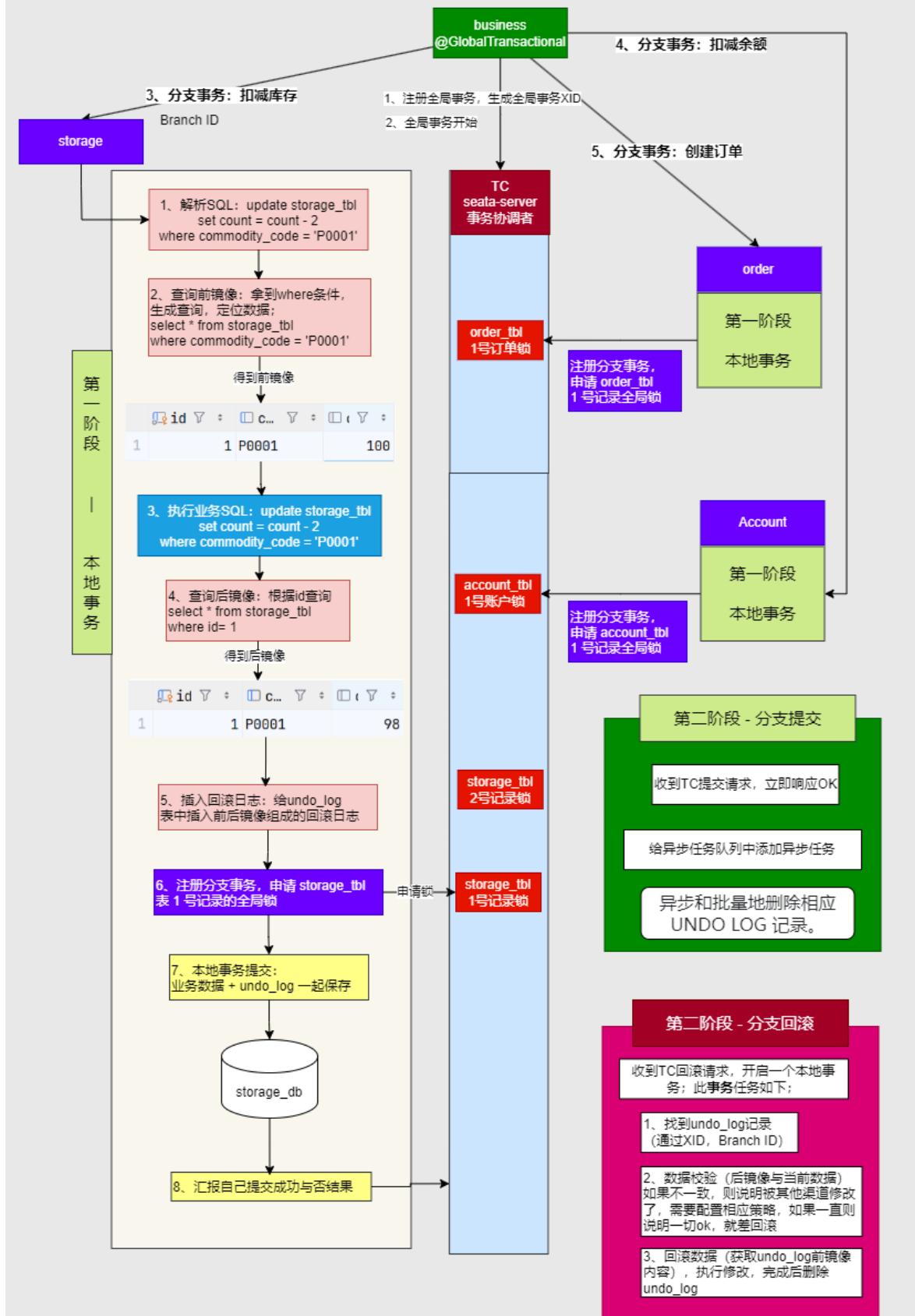
然而此时我们发现全局事务的回滚还是没有实现, 因为我们还需要在TM (上图的事务管理器) 的 `serviceimpl` 中的方法上标注 `@GlobalTransactional`

虽然 `@GlobalTransactional` 可以在技术层面上标注在任何 Spring 管理的 Bean 的方法上, 但最佳实践建议将其应用在服务层 (Service Layer), 即 Service 实现类的方法上, 而不是 Controller 层。这是因为: 按照典型的分层架构 (Controller -> Service -> Repository), Controller 层主要负责接收请求和返回响应, 而具体的业务逻辑应该封装在 Service 层。因此, 涉及事务管理的操作更适合放在 Service 层实现。

原理

二阶提交协议流程

Seata - 二阶提交协议



二阶提交协议



undo_log



四种事务模式

AT: 上面的模式 (推荐的模式), 性能高于XA

TCC: 也是二阶提交协议, 全部要手动编写代码, 第一阶段定义本地应该做什么; 第二阶段定义发生异常怎么做, 如何回滚。和成功了怎么做, 如何提交。所以一般适用于广义事务 (不局限于数据库事务) 的二阶提交处理

Saga: 整合消息队列来做事务的最终一致性。

XA: 也是二阶提交协议, 但是是数据库原生支持的, 效率较差, 一般不用。

- **AT 模式:** 适用于大多数情况, 特别是当希望最小化对现有业务代码的影响时。
- **TCC 模式:** 适用于需要精确控制事务流程且可以接受一定复杂度增加的场景。
- **Saga 模式:** 适用于长时间运行的服务调用或需要处理复杂业务流程的场景。
- **XA 模式:** 适用于对数据一致性要求极高且能够接受较低性能的场景。