介绍

随着互联网的快速崛起和网络购物的广泛普及,消费者在网购时越来越注重平台的购物体验，购物平台则面临着越来越高的并发访问量，这无疑对网购系统提出了更高的要求。单体架构下，扩展业务通常采取分布式加集群的方式构建和部署系统以应对高并发访问，但是随着线上消费的爆炸式增长，传统的系统扩展方式出现了瓶颈，无法满足庞大的消费需求。不同于单体架构，微服务架构将单一的应用划分为多个微服务，服务之间各司其职又互相配合，为用户提供最终价值;每个微服务按照业务逻辑划分,独立开发和部署,更好地满足单一职责原则和高内聚低耦合的设计原则。具有更高的灵活性和扩展性。为了缓解系统高并发访问的压力，本文基于微服务架构实现了一个商城购物系统。

首先，本文将系统核心设计为5个微服务:商品微服务、用户微服务、购物车微服务、订单微服务和支付微服务。其次，引入Nacos作为配置中心和注册中心，并使用JWT进行登录认证。然后通过请求限流和服务熔断等方式进行微服务的保护。

1.软件架构

1.1单体架构

单体架构（monolithic structure）：顾名思义，整个项目中所有功能模块都在一个工程中开发；项目部署时需要对所有模块一起编译、打包；项目的架构设计、开发模式都非常简单。

当项目规模较小时，这种模式上手快，部署、运维也都很方便，因此早期很多小型项目都采用这种模式。

但随着项目的业务规模越来越大，团队开发人员也不断增加，单体架构就呈现出越来越多的问题：

**A)团队协作成本高**：试想一下，你们团队数十个人同时协作开发同一个项目，由于所有模块都在一个项目中，不同模块的代码之间物理边界越来越模糊。最终要把功能合并到一个分支，你绝对会陷入到解决冲突的泥潭之中。

**B)系统发布效率低**：任何模块变更都需要发布整个系统，而系统发布过程中需要多个模块之间制约较多，需要对比各种文件，任何一处出现问题都会导致发布失败，往往一次发布需要数十分钟甚至数小时。

**C)系统可用性差**：单体架构各个功能模块是作为一个服务部署，相互之间会互相影响，一些热点功能会耗尽系统资源，导致其它服务低可用。

1.2微服务架构

微服务架构，首先是服务化，就是将单体架构中的功能模块从单体应用中拆分出来，独立部署为多个服务。同时要满足下面的一些特点：

**A)单一职责**：一个微服务负责一部分业务功能，并且其核心数据不依赖于其它模块。

**B)团队自治**：每个微服务都有自己独立的开发、测试、发布、运维人员，团队人员规模不超过10人

**C)服务自治**：每个微服务都独立打包部署，访问自己独立的数据库。并且要做好服务隔离，避免对其它服务产生影响

例如，本项目购物商城项目，我们就可以把商品、用户、购物车、交易等模块拆分，交给不同的团队去开发，并独立部署，如下图1.1所示

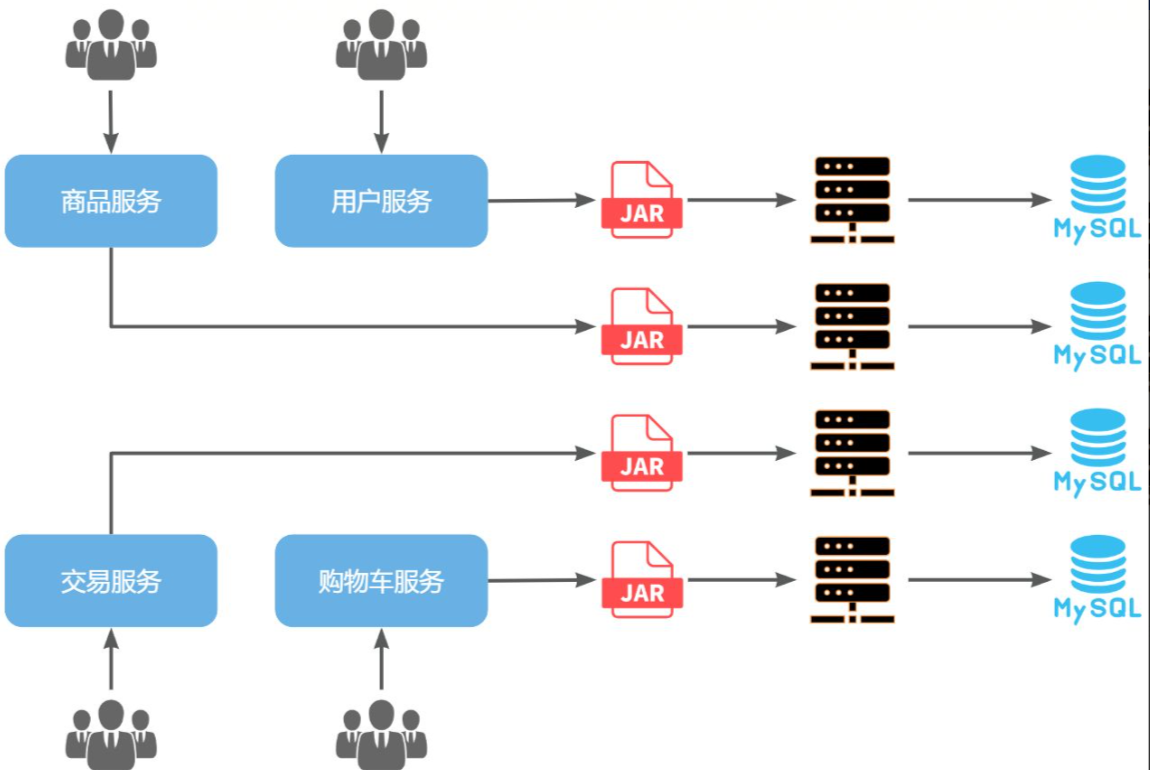


图1.1 购物商城微服务

1.3SpringCloud

微服务拆分以后碰到的各种问题都有对应的解决方案和微服务组件，而SpringCloud框架可以说是目前Java领域最全面的微服务组件的集合了。

而且SpringCloud依托于SpringBoot的自动装配能力，大大降低了其项目搭建、组件使用的成本。对于没有自研微服务组件能力的中小型企业，使用SpringCloud全家桶来实现微服务开发可以说是最合适的选择了。

本项目中我们使用的版本是：Spring Cloud 2021.0.3以及Spring Boot 2.7.12版本。如下图1.2所示。

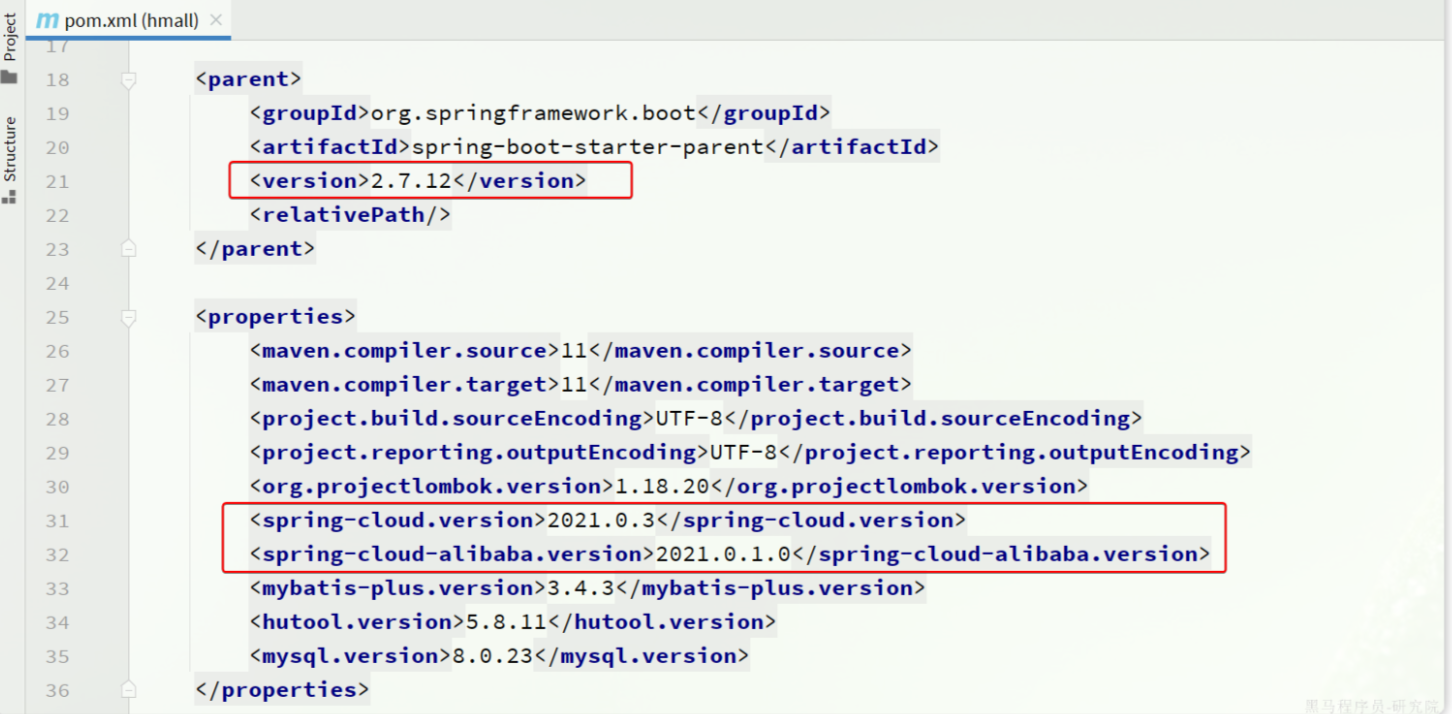


图1.2 Spring Cloud版本配置图

2.服务远程调用

当将一个项目部署为多个微服务，我们发现一个问题：就是购物车业务中需要查询商品信息，但商品信息查询的逻辑全部迁移到了item-service服务，导致我们无法查询。最终结果就是查询到的购物车数据不完整，因此要想解决这个问题，我们就必须改造其中的代码，把原本本地方法调用，改造成跨微服务的远程调用（RPC，即**R**emote **P**roduce **C**all）。因此，现在查询购物车列表的流程变成了如下图2.1所示

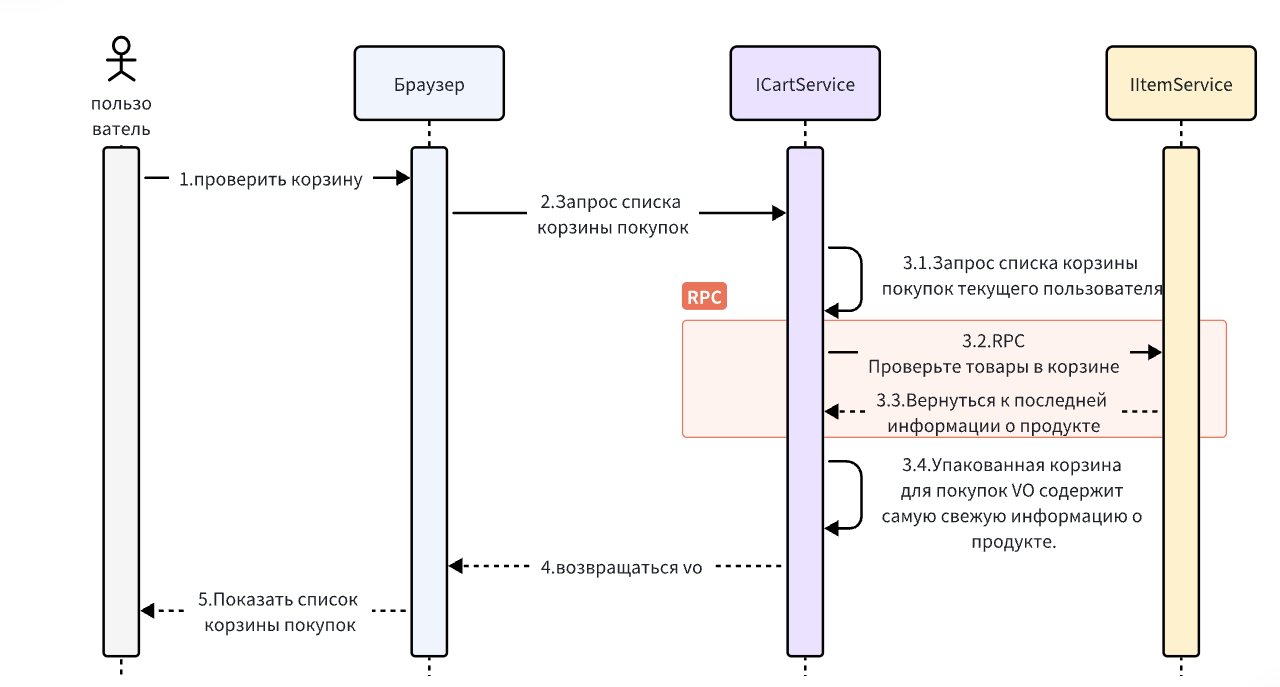


图2.1查询购物车时序图

RPC的实现方式有很多，比如：基于Http协议，基于Dubbo协议。本项目中使用的是Http方式，这种方式不关心服务提供者的具体技术实现，只要对外暴露Http接口即可，更符合微服务的需要。

Spring给我们提供了一个RestTemplate的API，可以方便的实现Http请求的发送。其中提供了大量的方法，方便我们发送Http请求，可以看到常见的Get、Post、Put、Delete请求都支持，如果请求参数比较复杂，还可以使用exchange方法来构造请求。

我们在cart-service服务中定义一个配置类，如下图2.2所示

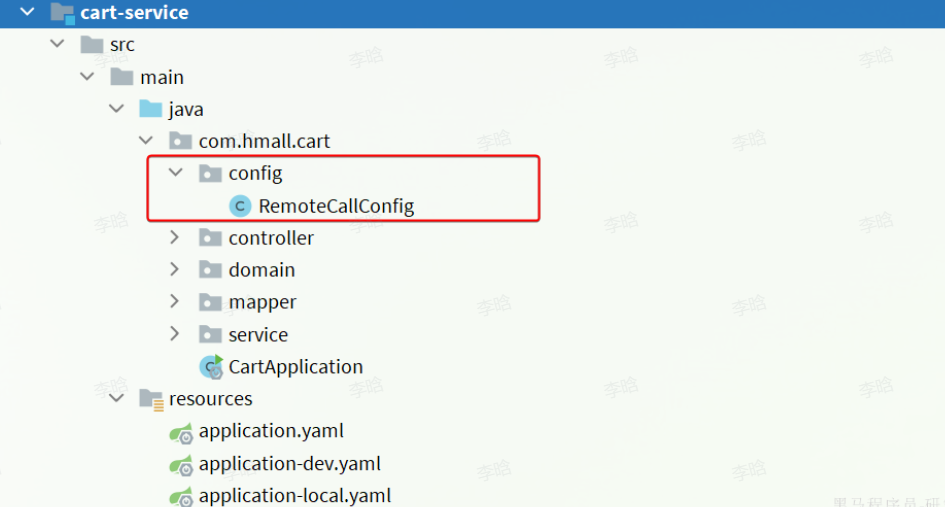


图2.2 远程调用配置类

接下来，我们修改cart-service中的CartServiceImpl的handleCartItems方法，发送http请求到item-service，如下图2.3所示。



图2.3 修改handleCartItems方法

最后重启cart-service，测试查询我的购物车列表接口，可以发现，所有商品相关数据都已经查询到了。在这个过程中，item-service提供了查询接口，cart-service利用Http请求调用该接口。因此item-service可以称为服务的提供者，而cart-service则称为服务的消费者或服务调用者。

3.服务注册和发现

3.1注册中心原理

我们通过Http请求实现了跨微服务的远程调用。不过这种手动发送Http请求的方式存在一些问题。试想一下，假如商品微服务被调用较多，为了应对更高的并发，我们进行了多实例部署，如图3.1所示。



图3.1 cart-service 多实例部署

此时，每个item-service的实例其IP或端口不同，产生的问题如下：

A)item-service这么多实例，cart-service如何知道每一个实例的地址

B)http请求要写url地址，cart-service服务到底该调用哪个实例呢

C)如果在运行过程中，某一个item-service实例宕机，cart-service依然在调用该怎么办

D)如果并发太高，item-service临时多部署了N台实例，cart-service如何知道新实例的地址

为了解决上述问题，就必须引入注册中心了，接下来我们就一起来分析下注册中心的原理。

在微服务远程调用的过程中，包括两个角色：服务提供者：提供接口供其它微服务访问，比如item-service；服务消费者：调用其它微服务提供的接口，比如cart-service

在大型微服务项目中，服务提供者的数量会非常多，为了管理这些服务就引入了**注册中心**的概念。注册中心、服务提供者、服务消费者三者间关系如下图3.2：

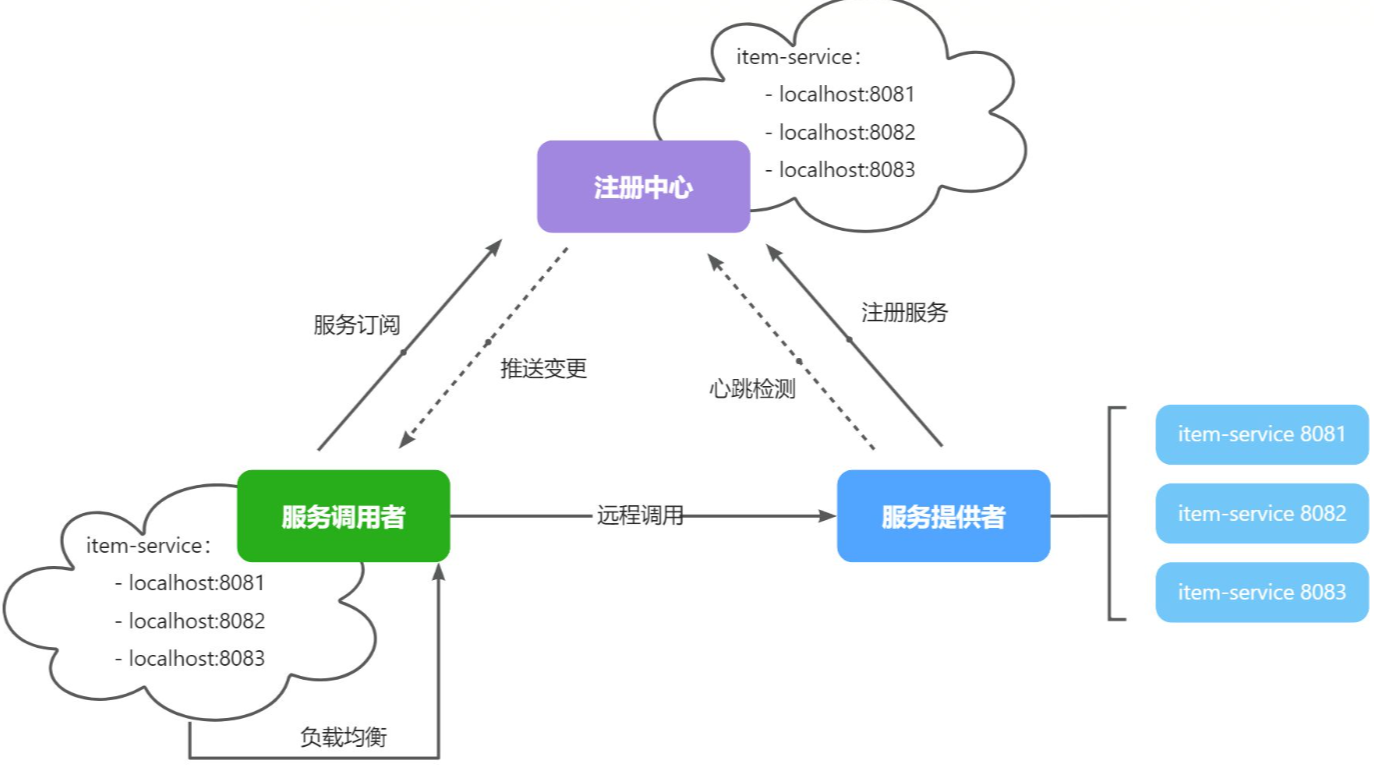


图3.2 注册中心概念图

流程如下：

服务动时就会注册自己的服务信息（服务名、IP、端口）到注册中心

调用者可以从注册中心订阅想要的服务，获取服务对应的实例列表（1个服务可能多实例部署）

调用者自己对实例列表负载均衡，挑选一个实例

调用者向该实例发起远程调用

3.2Nacos注册中心

目前开源的注册中心框架有很多，国内比较常见的有：

Eureka：Netflix公司出品，目前被集成在SpringCloud当中，一般用于Java应用；Nacos：Alibaba公司出品，目前被集成在SpringCloudAlibaba中，一般用于Java应用；Consul：HashiCorp公司出品，目前集成在SpringCloud中，不限制微服务语言

以上几种注册中心都遵循SpringCloud中的API规范，因此在业务开发使用上没有太大差异。本项目中我们采用Nacos作为注册中心。

3.3服务注册

接下来，我们以item-service为例，将其注册到Nacos，步骤如下：引入依赖；配置Nacos地址；重启。

在item-service的pom.xml中添加依赖，如下图3.3

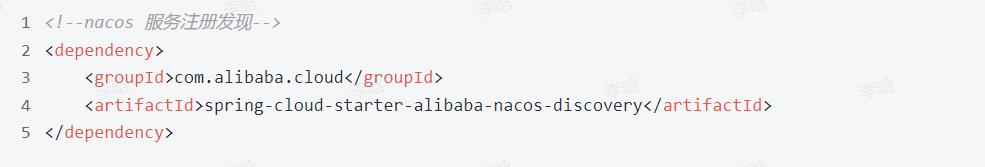


图3.3

在item-service的application.yml中添加nacos地址配置，如下图3.4



图3.4

为了测试一个服务多个实例的情况，我们再配置一个item-service的部署实例，重启item-service的两个实例，访问nacos控制台，可以发现服务注册成功，如下图3.5



图3.5

3.4服务发现

服务的消费者要去nacos订阅服务，这个过程就是服务发现，步骤如下：引入依赖；配置Nacos地址；发现并调用服务。

服务发现除了要引入nacos依赖以外，由于还需要负载均衡，因此要引入SpringCloud提供的LoadBalancer依赖。我们在cart-service中的pom.xml中添加如下图3.6的依赖：

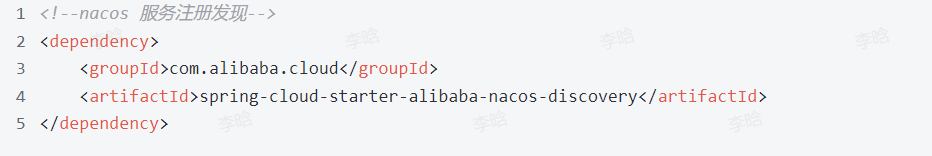


图3.6

可以发现，这里Nacos的依赖于服务注册时一致，这个依赖中同时包含了服务注册和发现的功能。因为任何一个微服务都可以调用别人，也可以被别人调用，即可以是调用者，也可以是提供者。

在cart-service的application.yml中添加nacos地址配置，如下图3.7所示

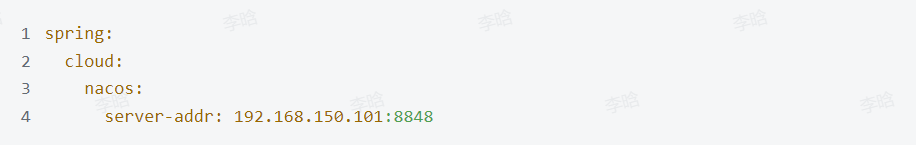


图3.7

接下来，服务调用者cart-service就可以去订阅item-service服务了。不过item-service有多个实例，而真正发起调用时只需要知道一个实例的地址。

因此，服务调用者必须利用负载均衡的算法，从多个实例中挑选一个去访问。常见的负载均衡算法有：随机、轮询、IP的hash、最近最少访问等，本项目中我们选择随机负载均衡。

4.网关路由及登录校验

4.1网关路由

顾名思义，网关就是**网**络的**关**口。数据在网络间传输，从一个网络传输到另一网络时就需要经过网关来做数据的**路由和转发以及数据安全的校验**。在微服务项目中，前端请求不能直接访问微服务，而是要请求网关：网关可以做安全控制，也就是登录身份校验，校验通过才放行；通过认证后，网关再根据请求判断应该访问哪个微服务，将请求转发过去。

在SpringCloud当中，提供了两种网关实现方案：Netflix Zuul和SpringCloudGateway。其中Netflix Zuul早期实现，目前已经淘汰，而SpringCloudGateway是基于Spring的WebFlux技术，完全支持响应式编程，吞吐能力更强，本项目中我们使用SpringCloudGateway。

由于网关本身也是一个独立的微服务，因此也需要创建一个模块开发功能。大概步骤如下：创建网关微服务；引入SpringCloudGateway、NacosDiscovery依赖；编写启动类；配置网关路由。

在gateway模块的resources目录新建一个application.yaml文件，如下图4.1为配置路由。



图4.1

4.2登录校验

单体架构时我们只需要完成一次用户登录、身份校验，就可以在所有业务中获取到用户信息。而拆分为微服务后，每个微服务都独立部署，不再共享数据。也就意味着每个微服务都需要做登录校验，这显然不可取。

我们的登录是基于JWT来实现的，校验JWT的算法复杂，而且需要用到秘钥。如果每个微服务都去做登录校验，这就存在着两大问题：每个微服务都需要知道JWT的秘钥，不安全；每个微服务重复编写登录校验代码、权限校验代码，麻烦。

既然网关是所有微服务的入口，一切请求都需要先经过网关。我们完全可以把登录校验的工作放到网关去做，这样之前说的问题就解决了：只需要在网关和用户服务保存秘钥；只需要在网关开发登录校验功能。登录校验的流程如图4.2所示。

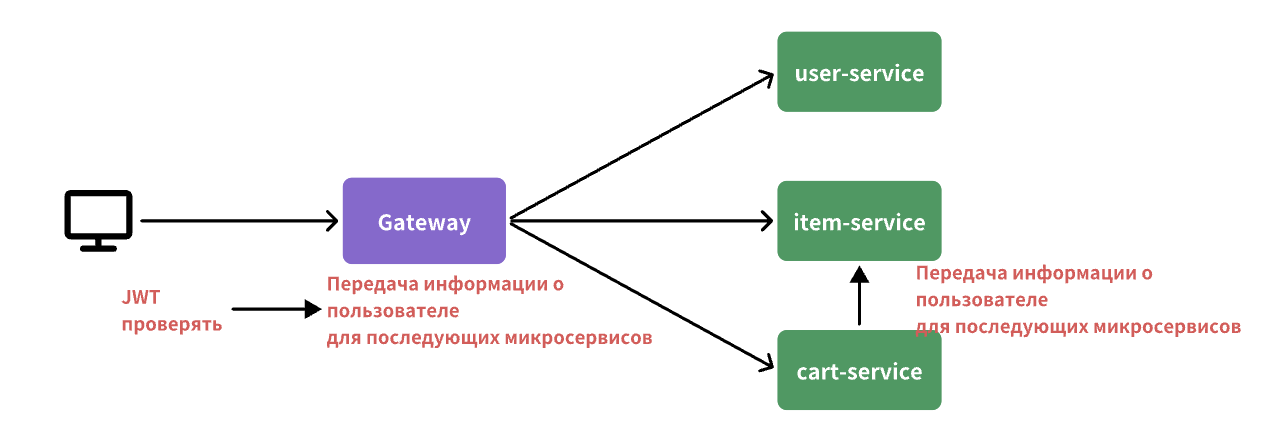


图4.2 登录校验流程图

登录校验必须在请求转发到微服务之前做，否则就失去了意义。而网关的请求转发是Gateway内部代码实现的，要想在请求转发之前做登录校验，就必须了解Gateway内部工作的基本原理。

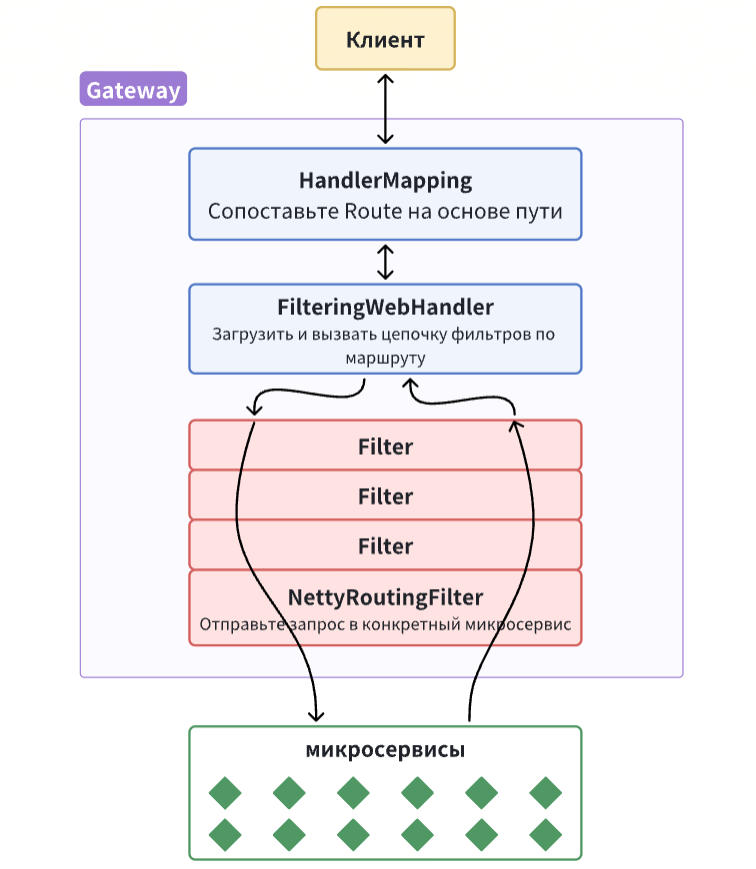


图4.3

如图4.3所示：

A)客户端请求进入网关后由HandlerMapping对请求做判断，找到与当前请求匹配的路由规则（Route），然后将请求交给WebHandler去处理。

B)WebHandler则会加载当前路由下需要执行的过滤器链（Filter chain），然后按照顺序逐一执行过滤器（后面称为Filter）。

C)图中Filter被虚线分为左右两部分，是因为Filter内部的逻辑分为pre和post两部分，分别会在请求路由到微服务**之前**和**之后**被执行。

D)只有所有Filter的pre逻辑都依次顺序执行通过后，请求才会被路由到微服务。

E)微服务返回结果后，再倒序执行Filter的post逻辑。

F)最终把响应结果返回。

最终请求转发是有一个名为NettyRoutingFilter的过滤器来执行的，而且这个过滤器是整个过滤器链中顺序最靠后的一个。**如果我们能够定义一个过滤器，在其中实现登录校验逻辑，并且将过滤器执行顺序定义到**NettyRoutingFilter**之前**，这就符合我们项目的需求。

网关过滤器链中的过滤器有两种：GatewayFilter：路由过滤器，作用范围比较灵活，可以是任意指定的路由Route；GlobalFilter：全局过滤器，作用范围是所有路由，不可配置。

我们项目中利用自定义GlobalFilter来完成登录校验。其中GlobalFilter核心代码如下图4.4所示。



图4.4 GlobalFilter核心代码图

接下来进行测试，访问我们设置的拦截路径，未登录状态下请求会被拦截，并且返回401状态码，如下图4.5所示。

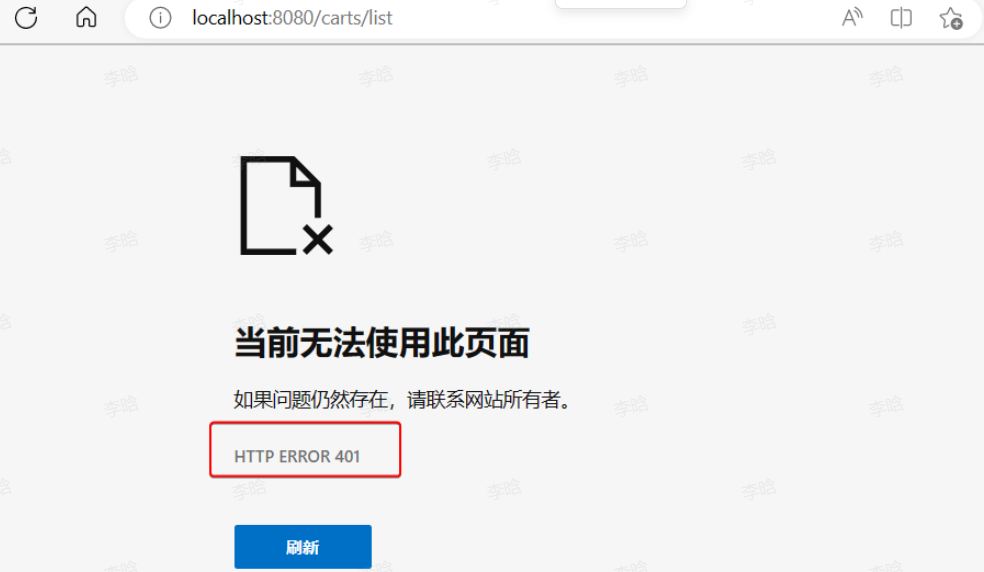


图4.5

5.服务保护

在微服务远程调用的过程中，还存在几个问题需要解决。

首先是**业务健壮性**问题：

例如在之前的查询购物车列表业务中，购物车服务需要查询最新的商品信息，与购物车数据做对比，提醒用户。从业务角度来说，为了提升用户体验，即便是商品查询失败，购物车列表也应该正确展示出来，哪怕是不包含最新的商品信息。

还有**级联失败**问题：

还是查询购物车的业务，假如商品服务业务并发较高，占用过多Tomcat连接。可能会导致商品服务的所有接口响应时间增加，延迟变高，甚至是长时间阻塞直至查询失败。

此时查询购物车业务需要查询并等待商品查询结果，从而导致查询购物车列表业务的响应时间也变长，甚至也阻塞直至无法访问。而此时如果查询购物车的请求较多，可能导致购物车服务的Tomcat连接占用较多，所有接口的响应时间都会增加，整个服务性能很差， 甚至不可用。依次类推，整个微服务群中与购物车服务、商品服务等有调用关系的服务可能都会出现问题，最终导致整个集群不可用。

5.1服务保护方案

5.1.1请求限流

服务故障最重要原因，就是并发太高。解决了这个问题，就能避免大部分故障。当然，接口的并发不是一直很高，而是突发的。因此请求限流，就是**限制或控制**接口访问的并发流量，避免服务因流量激增而出现故障。

请求限流往往会有一个限流器，数量高低起伏的并发请求曲线，经过限流器就变的非常平稳。这就像是水电站的大坝，起到蓄水的作用，可以通过开关控制水流出的大小，让下游水流始终维持在一个平稳的量。

5.1.2服务熔断

商品服务可能会拖慢购物车服务（服务调用方）的接口响应速度。而且商品查询的故障依然会导致查询购物车功能出现故障，购物车业务也变的不可用了。

所以，我们要做两件事情：1.**编写服务降级逻辑**：就是服务调用失败后的处理逻辑，根据业务场景，可以抛出异常，也可以返回友好提示或默认数据。2.**异常统计和熔断**：统计服务提供方的异常比例，当比例过高表明该接口会影响到其它服务，应该拒绝调用该接口，而是直接走降级逻辑。

微服务保护的技术有很多，本项目使用Sentinel实现微服务的保护，Sentinel是阿里巴巴开源的一款服务保护框架，目前已经加入SpringCloudAlibaba中。

5.2请求限流

在Sentinel中的簇点链路后面点击流控按钮，即可对其做限流配置，如图5.1。



图5.1

可以设置把查询购物车列表这个簇点资源的流量限制在了每秒6个，也就是最大QPS为6，如图5.2

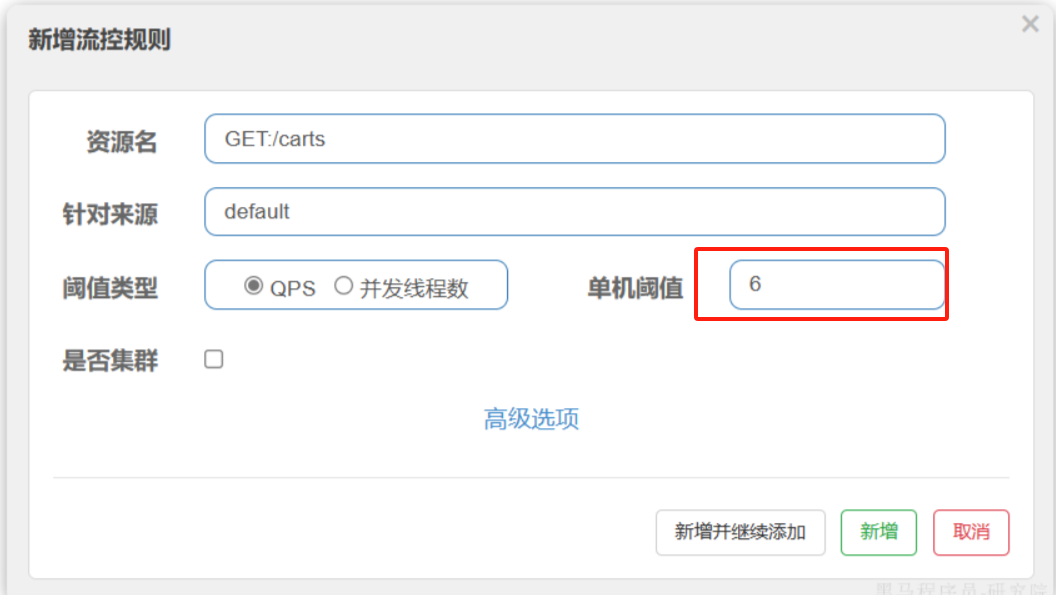


图5.2

5.3服务熔断

查询商品的RT较高，从而导致查询购物车的RT也变的很长。这样不仅拖慢了购物车服务，消耗了购物车服务的更多资源，而且用户体验也很差。

对于商品服务这种不太健康的接口，我们应该停止调用，直接走降级逻辑，避免影响到当前服务。也就是将商品查询接口**熔断**。当商品服务接口恢复正常后，再允许调用。这其实就是**断路器**的工作模式了。

Sentinel中的断路器不仅可以统计某个接口的**慢请求比例**，还可以统计**异常请求比例**。当这些比例超出阈值时，就会**熔断**该接口，即拦截访问该接口的一切请求，降级处理；当该接口恢复正常时，再放行对于该接口的请求。

断路器的工作状态切换有一个状态机来控制(如下图5.3)：

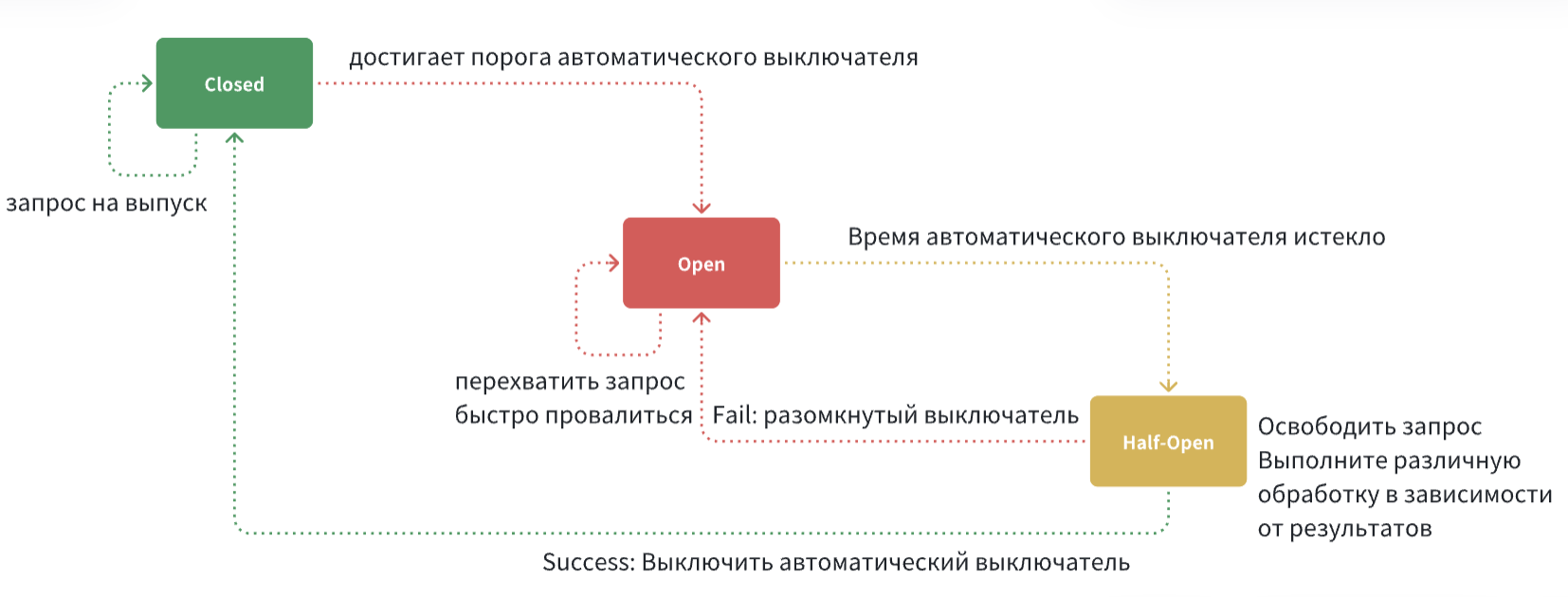


图5.3

状态机包括三个状态：

**close**：关闭状态，断路器放行所有请求，并开始统计异常比例、慢请求比例。超过阈值则切换到open状态

**open**：打开状态，服务调用被**熔断**，访问被熔断服务的请求会被拒绝，快速失败，直接走降级逻辑。Open状态持续一段时间后会进入half-open状态

**half-open**：半开状态，放行一次请求，根据执行结果来判断接下来的操作。

请求成功：则切换到closed状态，请求失败：则切换到open状态。

我们可以在控制台通过点击簇点后的熔断按钮来配置熔断策略，如下图5.4，图5.5。



图5.4



图5.5

结论

本文对网络购物和微服务架构进行研究，设计并实现了基于微服务架构的商城购物系统。分析了网络购物快速发展的原因以及利用单体架构搭建网购系统的不足，针对当前比较流行的微服务架构，深入研究了微服务的核心组件及微服务的发展趋势引入Nacos实现服务的注册、发现与管理，引入OpenFeign 解决了服务之间的互相调用问题，引入Gateway作为微服务的网关入口过滤与转发请求至相应微服务，引入Sentinel实现请求限流和服务熔断来保护微服务。

参考文献

1.张素娟，曾强、领跑实华开商务之路[J].中国电子商务,2004(11):25-29.

2.蔡勇劲.中国互联网三代“三巨头”的浮沉与迭代[J].大数据时代,2020(07):60-76

3.祝慧敏，社交电商的发展策略研究[D].山东师范大学,2020

4.胡航，陈晨，向海洋等.微商时代人际关系的转变及对微商未来的影响[J].现代商贸工业2021,42(03):56-58.

5.柴乔杉.92天上市,快手快人一步[J].中国品牌,2021(03):76-79.

6.SharmaA, Kumar M, AgarwalS. A complete survey on software architectural styles and pattems[9][J].Procedia Computer Science,2015,70:1628.

7.Dragoni N, Giallorenzo S, Lafuente A L, et al. Microservices: Yesterday, today, and tomorrow [G]// Present and Ulterior Software Engineering. Berlin: Springer, 2017: 195216.

8.Namiot D, Sneps-Sneppe M. On Micro-services Architecture[J]. international joural of openinformation technologies, 2014,2(9).

9.Jamshidi P, Pahl C, Mendonca N C, et al. Microservices: The Joumey So Far and ChallengesAhead[Jl.lEEE Software,2018,35(3):24-35.

10.Gray J.A conversation with Wemer Vogels{J]. Queue, 2006, 4(4):14-22

11.N.M. Josuttis, SOA in Practice: The Art of Distributed System Design[M]. O'Reily Media, Inc2007.