# SpringCloud

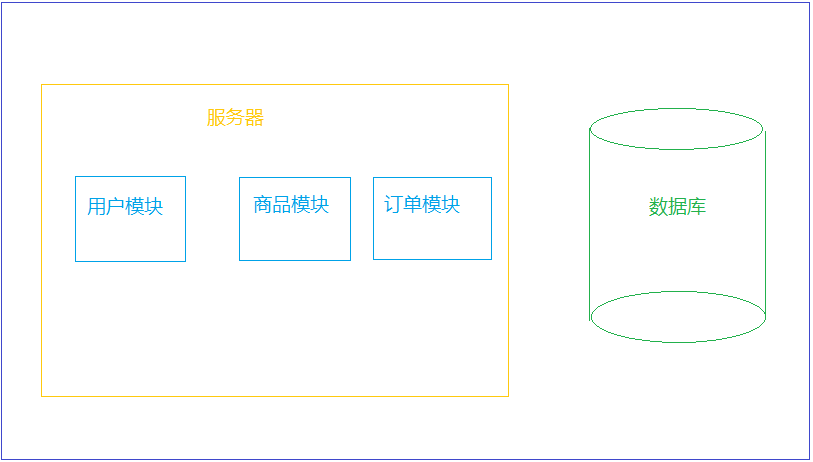
1. **WEB网站架构发展过程**

WEB网站架构是指为了应对一定数量级的请求量和响应速度，从技术选型到搭建整个网站结构的核心架构的过程。

WEB发展到现在由于系统规模越来越大，请求量并发量的增加，整个网站的架构已经经历了很多版本的变迁。了解WEB发展的历史，我们可以从宏观的角度去理解一门技术的必要性和重要性，从而做到对整个网站架构了然于胸，这是一个高级程序员必备的能力之一，以下我们来看一下WEB架构的发展历史。

**阶段一：单机架构**

我们以一个电商网站为例，一个电商系统中包含了用户模块、商品模块、订单模块。在WEB发展的初期或者说是在网站压力较小的情况下（服务器并发量低、数据库吞吐量要求不高），我们会将一个网站包含的所有模块都写在同一个项目中，然后Tomcat服务器和数据库都运行在一台电脑中，这些数据都保存在同一个数据库中。下图就是一个单体应用的结构。

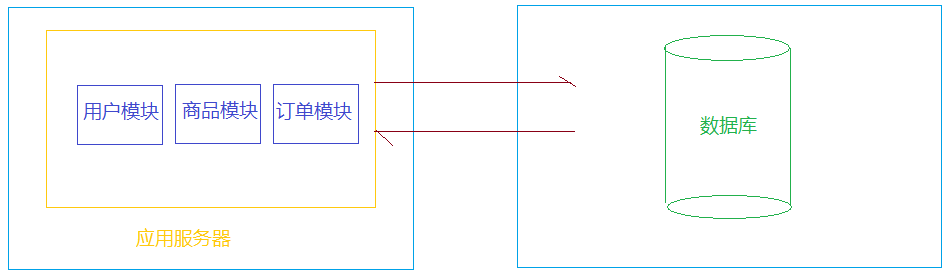


在这种单体结构中无论是用JSP+SERVLET，或是ssh、ssm都一样能够快速的开发出来，这种结构现在依然在很多项目的被广泛的使用。

**阶段二：应用服务器与数据库分离**

但是随着访问量逐步增大，服务器负载越来越大，单台计算机的CPU和内存压力都很大，同时代码层面又很难继续优化。这时我们可以将应用服务器和数据库服务器拆开，简单来说就是将TOMCAT装载一台服务器上，mysql装载另外一台服务器上，这就进入了第二个阶段。

如下图：



**阶段三：服务器集群**

假如现在采用二阶段的结构来设计网站，发现数据库的压力还不大，但是应用服务器已经处于高负载的情况下，我们可以从硬件的角度来解决这个问题，那就是加服务器，一台不够上两台，两台不够上三台。但是这种解决方案会带来一些新的问题：

1. 多个服务器上线的情况下，由谁来决定用户访问哪个服务器?

在多台服务器同时运行时，如何决定用户访问哪个服务器并同时保证多台服务器之间的负载是平衡的，不会出现一台服务器已经暴毙，另外一台服务器还在旁边嗑瓜子这种情况。这时就需要使用负载均衡来解决这个问题。Nginx就是这样一款反向代理服务器，所谓的反向代理是指用户将请求发送到反向代理服务器上，由反向代理服务器来进行请求分发，在这个过程中就可以实现负载的均衡。

Nginx负载均衡的3种方式：

轮询

权重:

服务器序列

权重数

当前权重数

每次服务器请求，Nginx会选出当前权重数最大的服务器来处理请求。

Ip\_hash

1. 服务器会保存用户的Session对象，用户在服务器A登录，在服务器B下订单，如何共享Session数据?

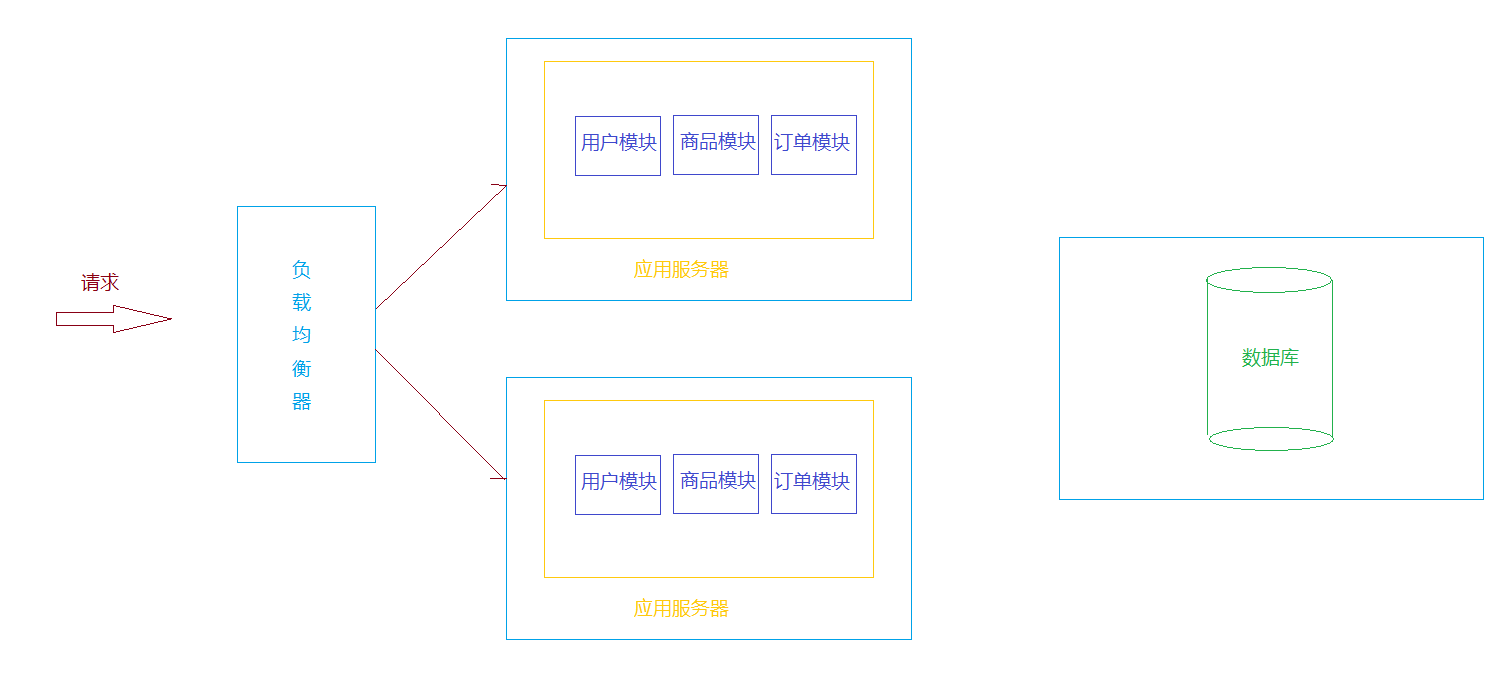
解决第二个问题的方式很多，例如：Session Sticky、Session Replication、数据库存储Session等。

Session Stickey是指在第一次会话创建以后就将用户的ip与服务器绑定，只要是这个ip发送的请求统一交给该服务器处理，这就不存在所谓Session共享的问题了。

Session Replication是指在服务器间进行Session的复制，保证每个应用服务器单独保存一份Session。

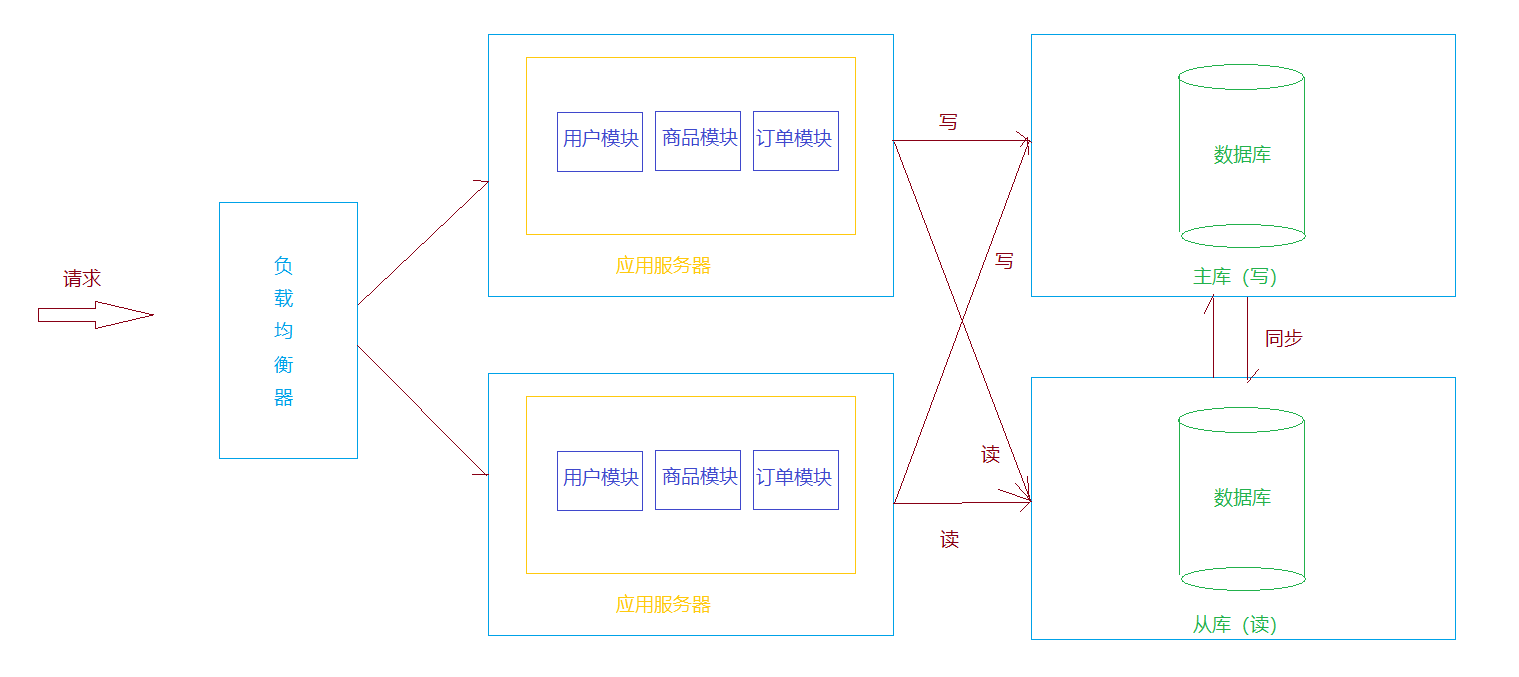
数据库存储Session是指将Session数据保存在数据库中，应用服务器需要从数据库中获取Session数据，例如将Session保存在Redis中，以便于在各个服务器间共享Session。

如下图：



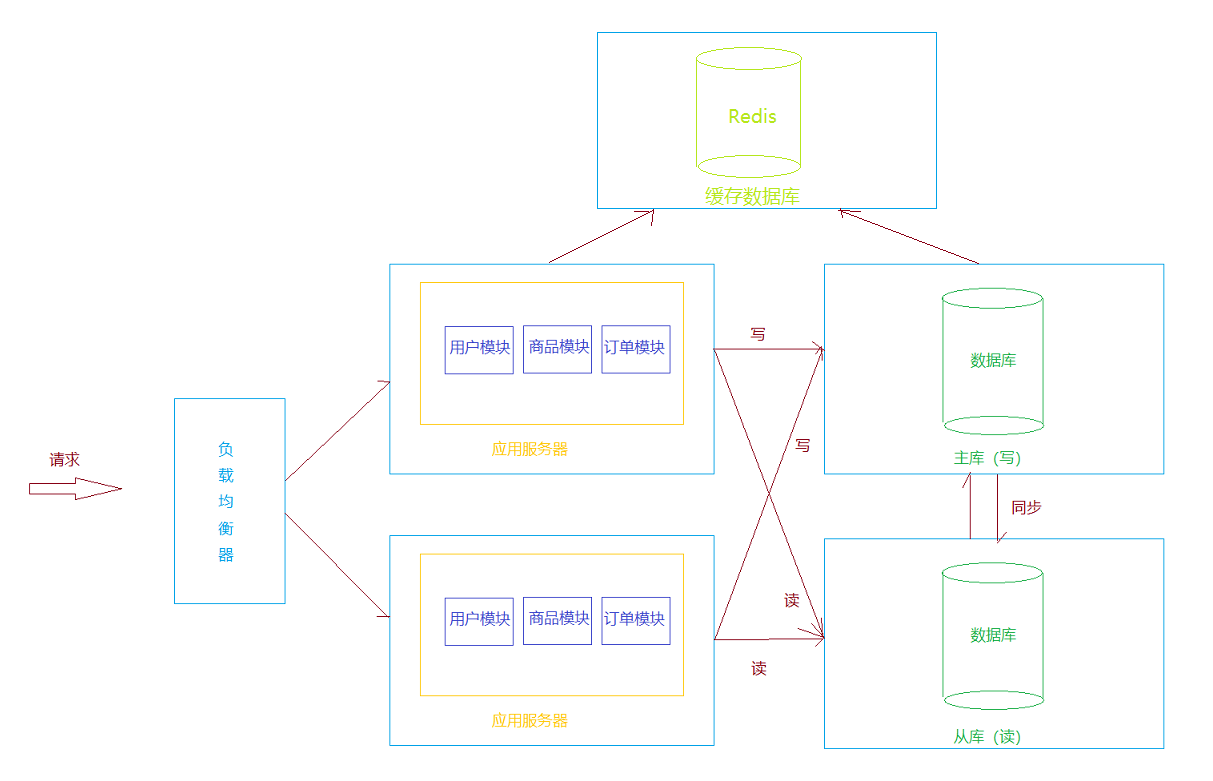
**阶段四：数据库读写分离**

通过部署多台服务器，服务器的压力变小了，但是随着请求继续，数据库的处理能力到达了一个瓶颈，有了上面的解决方案可能大家第一时间想到的是数据库服务器也部署多台，思路没错但是实现的细节有区别，因为我们如果简单直接部署两台数据库，这两台数据库的数据一致性问题如何来解决呢，在这个地方通常的解决方案是读写分离。所谓的读写分离是指当服务应用中读多写少的情况下（写的效率远远低于读），将数据库分为一台主服务器和多台从服务器，主服务器负责写数据和记录二进制日志，从数据库负责读操作和通过读取二进制日志进行数据同步。



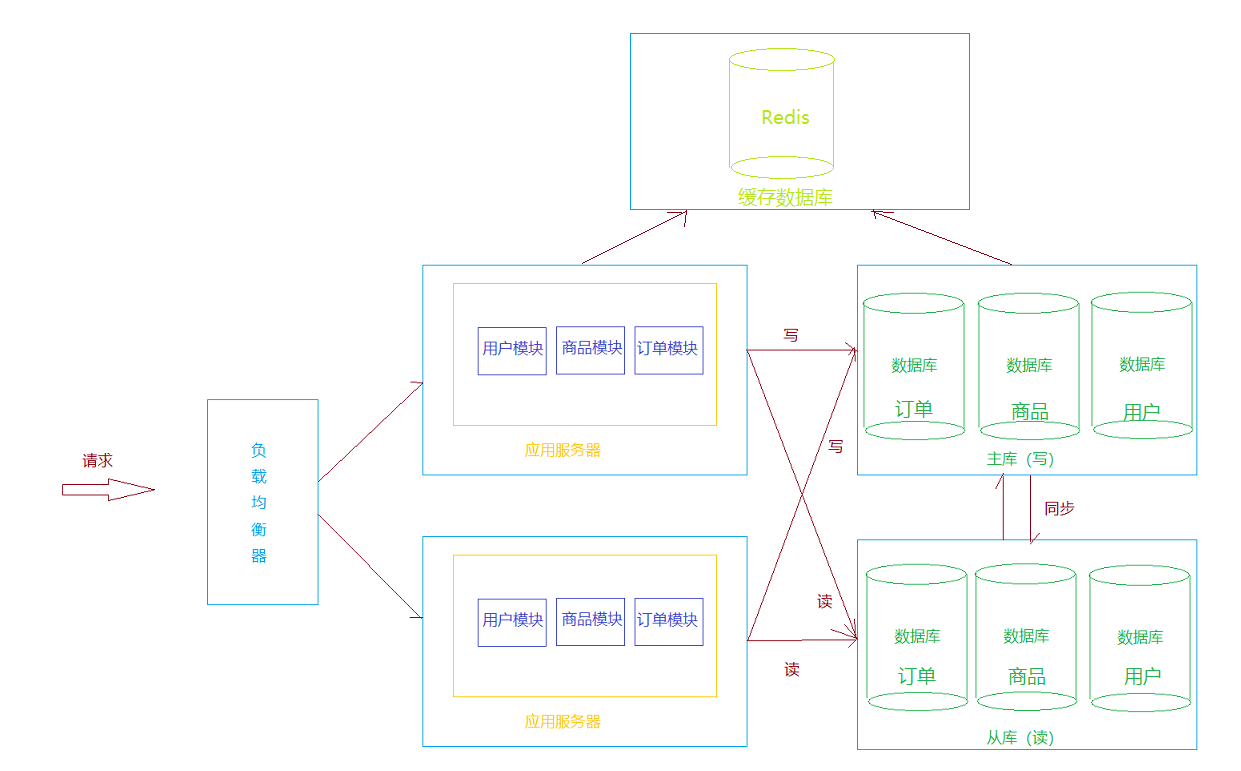
**阶段五：引入缓存降低数据库压力**

二八原则是很多地方都适用，在服务器中也类似，也就是80%的业务都在访问20%的数据，那么这20%的数据就是所谓的热点数据，对于这些热点数据，我们可以使用NOSQL数据库例如Redis、MongoDB等来保存这些数据，由于NOSQL数据库具备高效的响应速度，所以可以大大的降低数据库的压力。

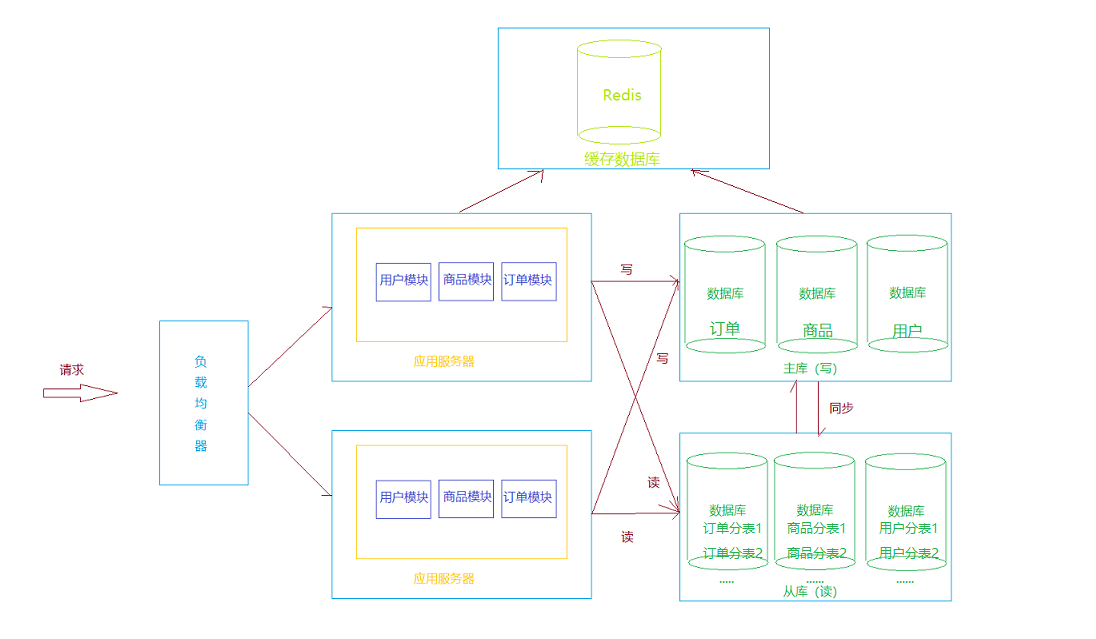


**阶段六：数据库分库和分表**

经过读写分离和数据缓存，数据库的压力还是没有解决的情况下，就要开始对数据库进行分库和分表。我们可以把不同的表保存在不同的数据库中，将压力分散到不同的数据库中。也就是说商品模块相关表放到一个数据库，订单模块数据放到另外一个数据库。



经过分库之后，数据库的压力降低了，但是对于单点业务的数据库压力并没有变小，也就是说如果商品模块数据库压力过大，单纯分库还是没办法解决这个问题，这时就需要分表了，所谓分表是指一张表的数据量过大，查询效率降低，就将一张表拆分称为多张表，不同的字段存储在不同的表中，这叫做垂直分表。又或者将一张表中的前1000W条数据保存到表A，后1000W条数据保存到表B，这是水平分表。分表分为库内分表和分库分表，库内分表指一个数据库中保存一张表拆分出的多张表，分库分表指一张表拆出来的多张表保存在多个数据库中。



一旦数据库采用了分表和分库的设计，这就会出现一个问题，那就是分布式事务，数据保存在不同的库中，一个业务需要同时对多个库进行写操作，甚至一个业务都不在同一个应用服务器上（SOA）。如何保证事务的一致性，这就是一个很复杂的问题，针对这个问题目前依然没有绝对完美的解决方案。目前一般有2PC、TCC、本地消息表等。

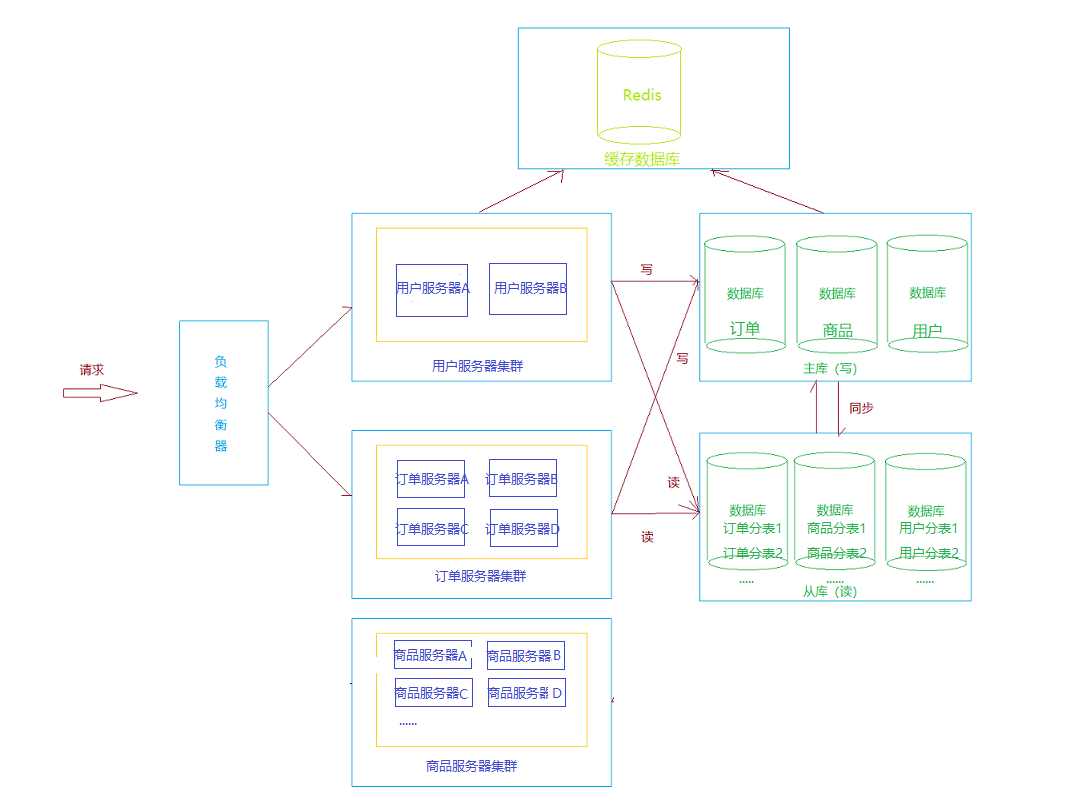
2PC：2PC意为二阶段提交，在程序中有一个事务协调者，事务协调者会首先向业务参与者（单个事务）发送确认消息，只有每个事务参与者都回复确认，然后再向业务参与者发送消息进行提交。这种方案在业界使用的较少，因为一旦事务链过长，将会极大的降低数据库效率。

TCC：TCC是指事务补偿机制，是指针对每个数据操作，都需要预先定义好失败和成功情况下的处理方式，当操作A成功则执行成功情况下的代码，如果操作A失败则执行失败情况下的代码。典型的案例是支付宝的异步回调。

**阶段七：SOA架构**

在通过各种手段处理了数据库的压力之后，业务服务器的压力可能又会成为系统中的一个吞吐瓶颈，同时每个服务器中都运行一套完整的项目，那么假如其中一个模块要进行维护，所有的服务器都必须一起停下来进行维护。SOA是面向服务的架构。SOA把系统分离成不同的服务，使用接口来进行数据交互，最终达到整合系统的目的。通过对业务进行拆分，首先提高了程序的可维护性，当订单系统需要维护，完全不需要停掉整个项目，只需要对订单服务器进行维护就可以。同时根据业务拆分模块以后，还可以根据不同业务访问的不同频率，搭建不同的服务器集群，对资源进行更好的利用，例如：

用户模块、订单模块、商品模块，就完全可以按照2 、4、 8这样的方式来搭建服务器集群。



最后现在流行的微服务架构，就是在SOA的基础上加上组件化思想，将SOA架构中的每一块，例如服务发现、服务治理、负载均衡等都分离成为独立的组件，在此思想的基础上所提出的一整套分布式系统的解决方案就是SpringCloud。

1. **RPC**

微服务的基础是SOA思想，SOA将业务拆分成不同的模块，模块间通过接口通讯来互相调用，那么接口间如何相互调用呢。这就涉及到了RPC的概念。

RPC全称Remote Process Call，翻译为远程过程调用。RPC的实现原理其实并不复杂，RPC包括了客户端和服务端，传统意义上的RPC中客户端是服务的调用方，而服务端是服务的提供方。服务端和客户端通过通讯协议通讯完成调用。

1. **SpringCloud简介**

Spring Cloud是一系列框架的有序集合。它利用Spring Boot的开发便利性巧妙地简化了分布式系统基础设施的开发，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控、网关等，都可以用Spring Boot的开发风格做到一键启动和部署。Spring Cloud并没有重复制造轮子，它只是将目前各家公司开发的比较成熟、经得起实际考验的服务框架组合起来，通过Spring Boot风格进行再封装屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包，是一套完整的分布式系统解决方案。

SpringCloud使用了SpringBoot来整合其他的框架，如果你企业中已经使用SpringBoot完成了业务，这时需要使用SpringCloud来进行重构，那就一定要注意SpringCloud和SpringBoot版本之间的兼容性，如下图：



1. **服务发现与服务调用**

在微服务的架构中，服务发现和服务调用是一个典型的消费生产者模式，服务生产者需要将服务注册到一个统一的服务注册中心，服务的消费者通过拉取服务注册中心的服务列表来调用这些服务。

Eureka是Netflix开发的服务发现框架，本身是一个基于REST的服务，主要用于定位运行在AWS域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的。SpringCloud将它集成在其子项目spring-cloud-netflix中，以实现SpringCloud的服务发现功能。

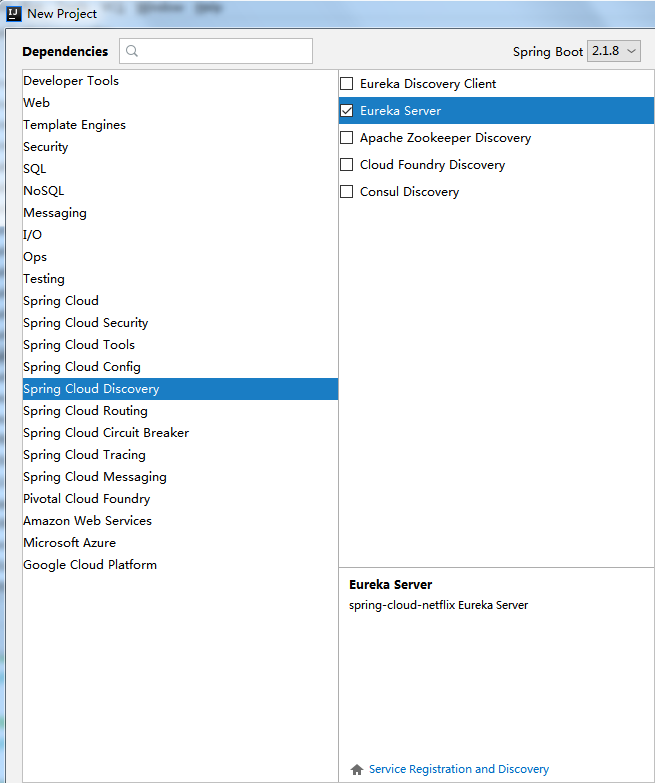
Eureka包含两个组件：Eureka Server和Eureka Client。

Eureka Server提供服务注册服务，各个节点启动后，会在Eureka Server中进行注册，这样EurekaServer中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。

Eureka Client是Eureka中的客户端，该客户端和以前学习过程中客户端的概念有所区别，这里的客户端包含了服务的消费者和服务器的生产者。

4.1 使用Eureka搭建服务注册中心

4.1.1 新建SpringBoot项目引入相关依赖



4.1.2 修改application.properties配置文件

#服务注册中心端口

server.port=8761

#服务注册中心地址

eureka.instance.hostname=localhost

#表示是否需要将自己注册到服务注册中心

eureka.client.register-with-eureka=false

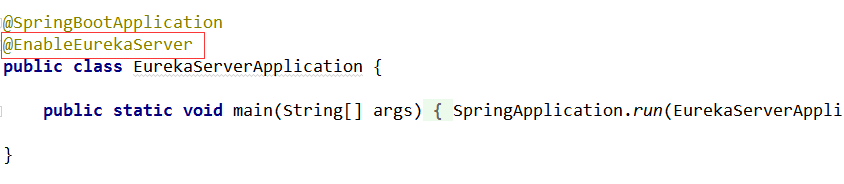
#表示是否需要拉取服务注册信息

eureka.client.fetch-registry=false

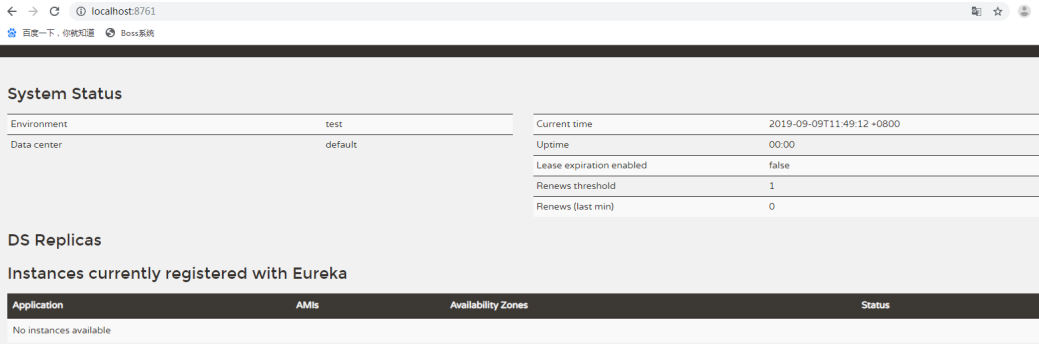
#服务注册中心地址，地址格式固定，必须以eureka结尾，只能修改前面的端口和地址

eureka.client.service-url.defaultZone=http://${eureka.instance.hostname}:${server.port}/eureka/

4.1.3 在启动类上添加注解

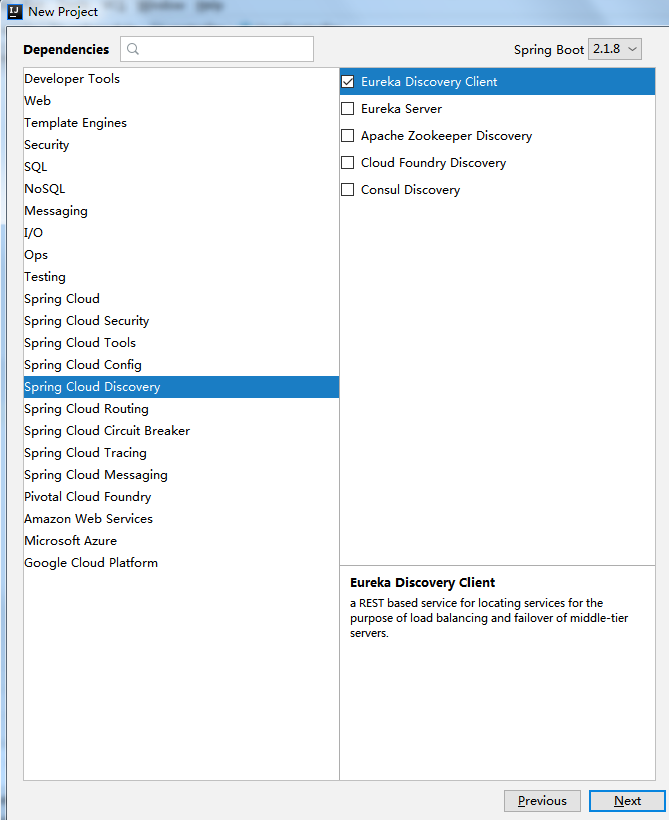


4.1.4 启动项目访问服务注册中心监控网页



4.2 使用Eureka进行服务注册

4.2.1 新建SpringBoot项目引入Eureka客户端依赖



4.2.2 修改application.properties配置文件

server.port=8081

#数据源配置

spring.datasource.url=jdbc:mysql://localhost:3306/ebuy?useUnicode=true&characterEncoding=utf-8&serverTimezone=GMT%2B8

spring.datasource.username=root

spring.datasource.password=

#mybatis配置

mybatis.mapperLocations=classpath:/mapper/\*Mapper.xml

#配置服务注册中心的域名

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

#客户端每隔五秒从服务器重新拉取服务器列表

eureka.client.registry-fetch-interval-seconds=5

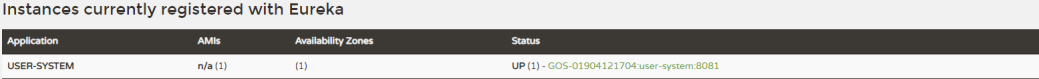
#修改应用名称 消费者可以使用应用名称来调用该服务中的接口

spring.application.name=user-system

4.2.4 在启动类上添加@EnableDiscoveryClient



先启动服务注册中心，然后启动客户端客户端项目就会作为服务注册到服务注册中心以便于其他服务调用。



这里要明确一点，一个客户端即可以是服务的生产者也可以是服务的消费者。换句话讲一个客户端中的服务可以发布到服务注册中心也可以从服务注册中心发现服务来进行调用。

客户端拉取服务列表间隔时间

eureka.client.registry-fetch-interval-seconds

上述属性表示eureka client间隔多久去拉取服务注册信息，默认为30秒，对于网关，如果要迅速获取服务注册状态，可以缩小该值，比如5秒

微服务心跳管理：

eureka.instance.lease-renewal-interval-in-seconds=30

定义服务注册中心向客户端发送心跳包的间隔时间

eureka.instance.lease-expiration-duration-in-seconds=90

上述属性表示eureka server距离上一次收到client的心跳之后，等待下一次心跳的超时时间，在这个时间内若没收到下一次心跳，则将移除该instance。默认为90秒。该值至少应该大于leaseRenewalIntervalInSeconds。

eureka.server.enable-self-preservation

上述属性用于设置是否开启自我保护模式，默认为true。默认情况下，如果Eureka Server在一定时间内没有接收到某个微服务实例的心跳，Eureka Server将会注销该实例（默认90秒）。但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，以上行为可能变得非常危险了——因为微服务本身其实是健康的，此时本不应该注销这个微服务。

Eureka通过“自我保护模式”来解决这个问题——当Eureka Server节点在短时间内丢失过多客户端时（可能发生了网络分区故障），那么这个节点就会进入自我保护模式。一旦进入该模式，Eureka Server就会保护服务注册表中的信息，不再删除服务注册表中的数据（也就是不会注销任何微服务）。当网络故障恢复后，该Eureka Server节点会自动退出自我保护模式。综上，自我保护模式是一种应对网络异常的安全保护措施。它的架构哲学是宁可同时保留所有微服务（健康的微服务和不健康的微服务都会保留），也不盲目注销任何健康的微服务。使用自我保护模式，可以让Eureka集群更加的健壮、稳定。

1. **服务路由Zuul**

在使用SpringCloud微服务架构设计完项目以后，我们的项目结构发生了巨大的变化，就拿上面的案例来说，服务组件都会有多个，每个服务组件都会单独运行在一个ip或者一个端口中。例如：

服务注册中心 ：localhost:8761

商品服务模块 ：localhost:8081

订单服务模块 ：localhost:8082

用户服务模块 ：localhost:8083

前端网页模块 ：localhost:8084

当我们想调用对应的服务模块时，需要记住大量的地址而且直接将地址和端口暴露出来，也是很不安全的，SpringCloud的zuul组件可以帮助我们解决这个问题。Zuul的意思是网关，对任何服务模块发起的请求都要请求到网关中，然后通过网关对不同的服务地址进行地址映射。例如上述的多个地址我们可以同时映射在网关地址之下。

商品服务模块 ：localhost:8088/product

订单服务模块 ：localhost:8088/order

用户服务模块 ：localhost:8088/user

前端网页模块 ：localhost:8088/web

这样做的可以隐藏服务器真实地址，同时更为重要的是所有请求都经过网关，可以在此处进行负载均衡和登陆校验等等。

5.1 使用zuul作为项目网关

5.1.1 新建springboot项目，引入eureka-client依赖、web依赖、zuul依赖。

5.1.2 修改配置文件

#当前服务的注册信息

#当前服务的端口

server.port=8083

#配置服务注册中心的域名

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/

#配置服务名称

spring.application.name=zuul

#zuul配置

#product-service 商品业务地址映射

zuul.routes.product-service.path=/product/\*\*

#商品业务服务名称

zuul.routes.product-service.service-Id=product-service

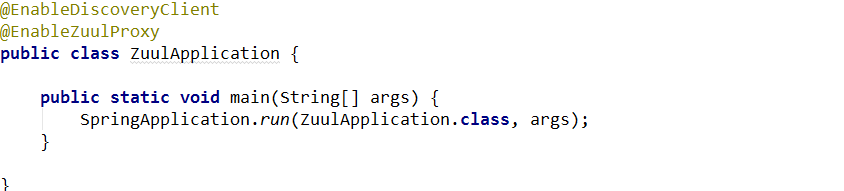
#web地址映射

zuul.routes.web.path=/web/\*\*

#web服务名称

zuul.routes.web.service-Id=web

5.1.3 启动类添加注解



5.1.4 注意事项

如果我们是通过网关在访问web模块，那么所有的请求地址都需要带上web模块在网关中的地址。

例如：原本web模块的地址是localhost:8084，在访问时前端的静态资源引入或者ajax地址都写作/js/xxxxx或者/product/get，但是一旦通过网关来访问web项目中的路径都必须修改为/web/js/xxxxx或者/web/product/get。

5.2 网关过滤器

网关作为所有请求的入口，在此处可以进行一些登陆校验或者权限验证等操作，这些操作都需要借助zuul的过滤器来实现。

spring cloud Zuul包含了对请求的路由和过滤2个功能。路由功能负责将请求转发到具体的微服务上，而过滤器负责对请求的处理过程进行干预，是实现权限校验、服务聚合等功能的基础。

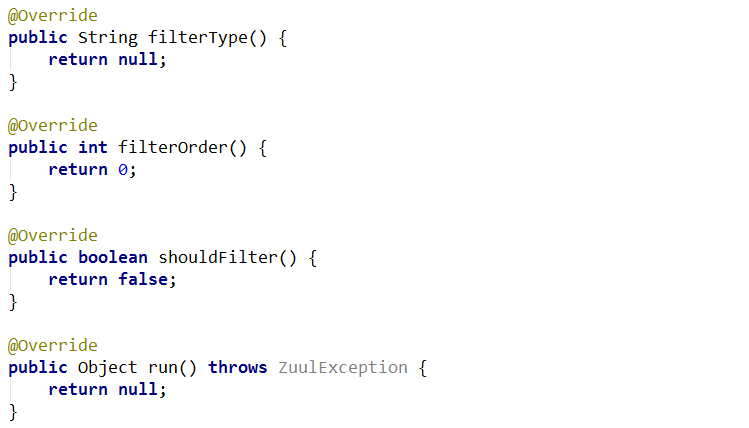
在实际运行时，路由映射和请求转发是由几个不同的过滤器完成的。每一个进入zuul的http请求都会经过一系列的过滤器处理链得到请求响应并返回给客户端。

5.2.1 zuul过滤器配置

5.2.1.1 新建类继承ZuulFilter



5.2.1.2 重写方法



重写的4个方法功能：

filterType：定义过滤器的执行时机，四种类型：pre，routing，error，post

pre：主要用在路由映射的阶段是寻找路由映射表的

routing:具体的路由转发过滤器是在routing路由器，具体的请求转发的时候会调用

error:一旦前面的过滤器出错了，会调用error过滤器。

post:当routing，error运行完后才会调用该过滤器

该方法的返回值为字符串，返回的那个字符串就定义了该过滤器的类型

filterOrder：在多个过滤器中的执行顺序，返回的数字越小越先执行，数字越大越靠后执行。

shouldFilter：定义是否启用该过滤器。

run：具体的过滤逻辑写在该方法中

该方法中可以使用请求容器对象来获取本次请求对象，也可以通过该对象阻止该请求向下去路由。

常用方法：

//获取容器

RequestContext ctx = RequestContext.getCurrentContext();

//停止向下路由

ctx.setSendZuulResponse(false);

//设置响应的错误编码

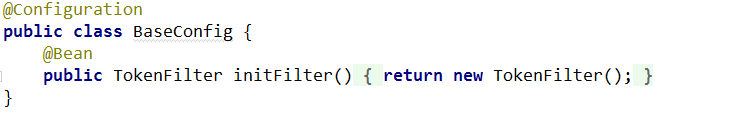
ctx.setResponseStatusCode(401);

//设置响应的错误信息

ctx.setResponseBody("miss token");

5.2.1.3

添加Config类将Filter类交给Spring来管理



5.3 使用zull+redis+spring-session实现session共享

在以前的单体项目中，用户信息可以保存在Session中，Session在一个服务器上保存，所有的业务请求都在这一个服务器上处理，但是一旦项目变成了微服务的项目，每个业务请求都有可能在不同的服务器上处理，那么如何保存用户的个人信息呢，这就是微服务架构中的Session共享问题。

Session共享问题有多种解决方案。目前最常见的就是将Session保存在Redis中，通过Spring-sesison来代理Session。

5.3.1 在网关中添加集成Redis和SpringSession。

添加redis依赖：

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-redis</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.session</groupId>

<artifactId>spring-session-data-redis</artifactId>

</dependency>

修改配置文件添加redis配置信息：

spring.redis.database=0

spring.redis.host=localhost

spring.redis.port=6379

spring.redis.password=123456

spring.redis.timeout=20000

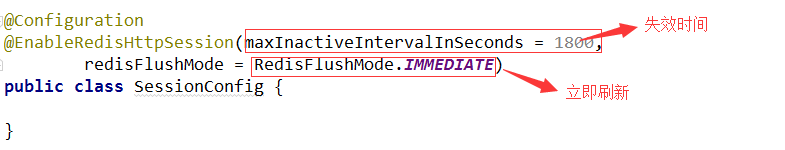
spring.redis.jedis.pool.max-active=8

spring.redis.jedis.pool.max-wait=-1

spring.redis.jedis.pool.max-idle=8

spring.redis.jedis.pool.min-idle=0

新建配置类开启注解



需要在网关中定义服务的头信息过滤规则，zuul默认会将头信息全部过滤掉，这会导致cookie丢失，就无法从redis中取出session数据了。

zuul.routes.服务名称.sensitiveHeaders="\*"

5.3.2 在web层中集成Redis和和SpringSession

按照5.3.1的步骤将Redis和SpringSession在集成到web模块中

5.3.3 然后在web层的控制层中就可以直接使用HttpSession，该Session中的数据时SpringSession从Redis中获取到的。

5.3.4 注意要解决第一次Session共享失败的问题，需要设置@EnableRedisHttpSession(flushMode = FlushMode.IMMEDIATE)数据存储模式为实时刷新，如果是springc-boot 2.2.0以下的版本，写作@EnableRedisHttpSession(redisFlushMode = RedisFlushMode.IMMEDIATE)。除此之外需要在网关处添加一个前置多滤器，在run方法中从网关的session中取出session-id通过Base64加密得到cookie的值，在添加到头信息中，转发到下层服务，这样就ok了。

//Base64加密代码

String cookie=Base64.getEncoder().encodeToString(session.getId().getBytes());

//将cookie加入请求头中

ctx.addZuulRequestHeader("Cookie", "SESSION="+cookie);

补充：网关中解决跨域问题

添加一个过滤器

@Bean

public CorsFilter corsFilter() {

final UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();

final CorsConfiguration config = new CorsConfiguration();

config.setAllowCredentials(true); // 允许cookies跨域

config.addAllowedOrigin("\*");// 允许向该服务器提交请求的URI，\*表示全部允许。。这里尽量限制来源域，比如http://xxxx:8080 ,以降低安全风险。。

config.addAllowedHeader("\*");// 允许访问的头信息,\*表示全部

config.setMaxAge(18000L);// 预检请求的缓存时间（秒），即在这个时间段里，对于相同的跨域请求不会再预检了

config.addAllowedMethod("\*");// 允许提交请求的方法，\*表示全部允许，也可以单独设置GET、PUT等

config.addAllowedMethod("HEAD");

config.addAllowedMethod("GET");// 允许Get的请求方法

config.addAllowedMethod("PUT");

config.addAllowedMethod("POST");

config.addAllowedMethod("DELETE");

config.addAllowedMethod("PATCH");

source.registerCorsConfiguration("/\*\*", config);

return new CorsFilter(source);

}

1. **负载均衡Ribbon**

负载均衡是分布式系统中用于平衡每一个服务器压力的组件。它通过负载均衡算法，使服务器集群中的服务按照一定的规则来分摊请求。

网关默认进行了负载均衡，采用了轮询的方式对经过网关的请求进行负载均衡。

6.1 使用Eureka+Ribbon完成对服务的调用

调用服务的地方本身也是另外一个客户端，所以前面的搭建过程参考4.2

引入Ribbon的依赖

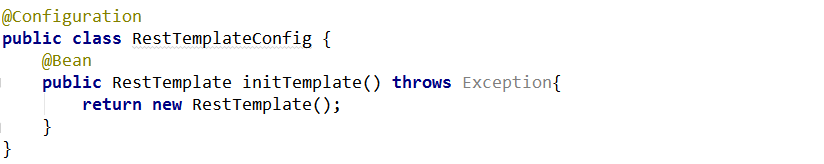
<dependency>

<groupId>org.springframework.cloud</groupId>

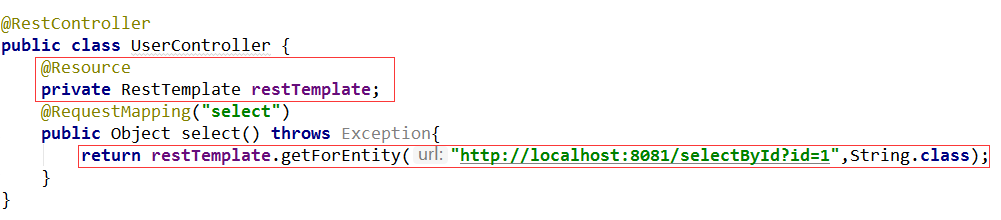
<artifactId>spring-cloud-starter-netflix-ribbon</artifactId>

</dependency>

使用SpringBoot的配置类来实例化RestTemplate对象，RestTemplate对象封装了HttpClient，可以对服务进行调用。



在需要调用服务的地方使用RestTemplate来调用远程服务

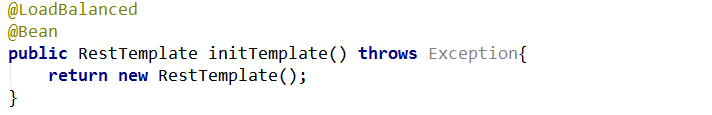


启动类添加客户端注解



如果使用服务的地址来调用服务，那么一旦服务发布到了其他的地址或者端口上，这边服务调用者也必须修改代码才可以，而且加入要做服务器集群，同一个服务会发布在不同的服务器上，这里使用硬编码来调用服务是非常不灵活的。所以可以在RestTemplate

配置类上添加@LoadBalanced注解，使用Ribbon来调用服务，Ribbon可以通过服务名称对服务发起请求，而且可以在服务器集群中采用负载均衡的方式来调用服务。



补充：RestTemplate的几种常见请求方式

GET请求

在RestTemplate中，发送一个GET请求，我们可以通过如下两种方式：

第一种：getForEntity

getForEntity方法的返回值是一个ResponseEntity，ResponseEntity是Spring对HTTP请求响应的封装。

包括了几个重要的元素，如响应码、contentType、contentLength、响应消息体等。

第二种：getForObject

getForObject函数实际上是对getForEntity函数的进一步封装，如果你只关注返回的消息体的内容，对其他信息都不关注，此时可以使用getForObject。

POST请求

在RestTemplate中，POST请求可以通过如下两个方法来发起：

第一种：postForEntity

该方法和get请求中的getForEntity方法类似。

第二种：postForObject

如果你只关注，返回的消息体，可以直接使用postForObject。用法和getForObject一致。

补充：getForEntity和postForEntity获取的body数据是按照ISO8859-1的编码格式来编码的，我们需要进行转码，先通过ISO8859-1解码为字节数组，然后在按照UTF-8进行编码成为字符串，

String json=new String(response.getBody().getBytes("ISO8859-1"),"utf-8");

重点补充：在上面的课程中我们讲到了Session共享。原理是通过cookie记录session键，然后通过redis来保存session数据并实现Session共享。这里就涉及到一个问题，如果是通过网关访问服务，那么cookie通过网关转发到服务，这样可以实现Session共享。但是如果是使用RestTemplate来实现服务间的内部调用，由于没有通过网关，并且在服务间内部调用时也没有携带cookie，那么就会出现服务间Session共享失效的问题，那么解决方案就是在通过RestTemplate发送请求时，将cookie包含在本次请求中，具体的代码如下：

//创建Http头信息对象

HttpHeaders headers = new HttpHeaders();

//创建集合保存cookie信息

List<String> cookies =new ArrayList<String>();

//定义变量保存session的键

String sessionkey=null;

//遍历request中的cookies取出session键

Cookie[] httpcookies=req.getCookies();

for (Cookie cookie:httpcookies){

if(cookie.getName().equals("SESSION")){

sessionkey=cookie.getValue();

}

}

//将session键保存到cookie中

cookies.add("SESSION="+sessionkey+";Path=/; HttpOnly");

//将cookie存入头部

headers.put(HttpHeaders.COOKIE,cookies);

//请求的参数

MultiValueMap<String, String> map = new LinkedMultiValueMap<String, String>();

map.add("testid", "1");

//封装参数和cookie

HttpEntity<MultiValueMap<String, String>> request = new HttpEntity<MultiValueMap<String, String>>(map, headers);

//发送请求

ResponseEntity<String> response = restTemplate.postForEntity("http://test-service/test", request, String.class);

6.2 负载均衡算法

默认情况下，Ribbon默认的负载均衡方式是轮询，也就是服务器集群中的每个服务器轮流处理一次请求。

我们可以通过简单的配置将默认的轮询方式修改为随机方式。

服务名称.ribbon.NFLoadBalancerRuleClassName=com.netflix.loadbalancer.RandomRule

如果是通过网关进行的负载均衡，那么需要在网关配置文件中进行配置。

如果是服务间调用时进行负载均衡，那么需要在每个单独的服务中进行配置。

1. **Hystrix提供服务器容错能力**

如果服务的提供者响应非常缓慢，消费者也必须等待服务处理完毕。在高负载的情况下，如果出现这样的问题，那么有可能就会导致整个服务器的崩溃。这是因为微服务架构的应用系统通常包含多个服务层，服务层之间通过网络来调用。任何的程序都不是100%的可用，网络也很脆弱，那么当一个处于下层的服务出现故障进而导致了上层服务故障，并且层层传递最终导致整个服务崩溃的情况被称为雪崩效应。

我们可以通过Hystrix断路器来解决这种问题，在生活中当我们家中的电力负载过大时，如果继续工作极有可能引发火灾，为了避免出现这种情况，在家中都会安装断路器，断路器在电力负载过大时主动切断电源，来保障安全。SpringCloud中的断路器组件也是出于这种思想来设计的，当断路器发现一个服务调用的时间过长，就会主动的停止调用该服务，并转而执行我们提前准备好的一个备用方法，当然备用方法并不能响应正确的执行结果，只能做一些默认数据的响应。

服务的熔断和降级：

服务的降级：当服务时间超出了我们的最大等待时间，本次请求会被判定为异常，服务将不在处于等待，会转而执行一个备用的服务方法，该方法返回的数据是默认的无效数据。这叫做服务的降级。

服务的熔断：在一批请求中的一部分请求都被判定为异常，进行了降级之后，断路器将会打开，此时的请求不会再尝试调用服务接口，而是直接访问备用服务。

服务的半熔断：当断路器开启一段时间以后（默认是5s），Hustrix会尝试将请求再次发送给正常服务来处理，如果发现已经已经正常，就会关闭断路器。

使用Hystrix步骤：

1. 在服务中引入Hystrix依赖

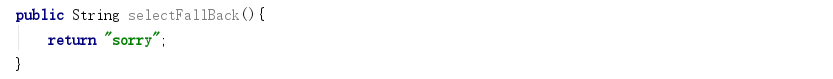
<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-netflix-hystrix</artifactId>

</dependency>

1. 在控制层的方法根据接口的功能定义相应的备用方法



1. 在控制层的服务方法上添加注解



在该注解中可以定义，该方法响应时间过长后的备用方法名以及最大的等待时间。

其他相关配置：

execution.isolation.thread.timeoutInMilliseconds

配置服务降级时间

circuitBreaker.requestVolumeThreshold

配置熔断判断开始的次数，默认20次

circuitBreaker.errorThresholdPercentage

配置熔断判断请求失败比例，默认50%

circuitBreaker.sleepWindowInMilliseconds

配置熔断以后，尝试恢复断路器的间隔时间，默认5000毫秒

1. 在启动类上添加@EnableHystrix注解启用断路器
2. 注意如果使用了网关，由于网关默认等待1秒没有响应就会短路，所以我们需要将网关的等待时长设置得更长一些。

在网关的配置文件中添加如下两段配置

ribbon.ConnectTimeout=10000

ribbon.ReadTimeout=10000

1. **高可用注册中心**

注册中心作为微服务体系中的最核心的组件，负责发现所有的服务。如果注册中心宕机将会导致整个微服务体系崩溃。为了保证微服务架构的高可用性，可以在微服务体系中搭建一个注册中心的集群，注册中心之间相互注册，同时每一个服务需要注册到所有的注册中心。这样即使其中一个注册中心出现故障，也依然不会出现整个微服务体系崩溃的情况。

以下给出注册中心的配置:

|  |
| --- |
| 注册中心1: |
| #服务注册中心端口 |
| server.port=8761 |
| #服务注册中心地址 |
| eureka.instance.hostname=localhost |
| #表示是否需要将自己注册到服务注册中心 |
| eureka.client.register-with-eureka=false |
| #表示表示是否需要拉取服务注册信息 |
| eureka.client.fetch-registry=false |
| #将该注册中心注册到注册中心2 |
| eureka.client.service-url.defaultZone=http://localhost:8762/eureka/ |
| #允许通过ip地址进行服务中心的注册 |
| eureka.instance.preferIpAddress=true |

|  |
| --- |
| 注册中心2 |
| #服务注册中心端口 |
| server.port=8762 |
| #服务注册中心地址 |
| eureka.instance.hostname=localhost |
| #表示是否需要将自己注册到服务注册中心 |
| eureka.client.register-with-eureka=false |
| #表示表示是否需要拉取服务注册信息 |
| eureka.client.fetch-registry=false |
| #将该注册中心注册到注册中心1 |
| eureka.client.service-url.defaultZone=http://localhost:8761/eureka/ |
| #允许通过id地址进行服务中心的注册 |
| eureka.instance.preferIpAddress=true |

同时除了注册中心以外的所有组件或者服务都需要同时注册到这两个注册中心，具体的配置方式就是在微服务向注册中心注册时写两个注册中心的地址，例如：

eureka.client.service-url.defaultZone=http://localhost:8761/eureka/,http://localhost:8762/eureka/

补充：也可以将注册中心的hostname进行自定义，以便于区分不同的注册中心，但是这种操作需要在对应的电脑系统中修改hosts文件。

例如：eureka.instance.hostname=server1

那么hosts中就需要在末尾添加一句127.0.0.1 server1进行地址和ip的映射

1. **配置中心**

对于传统的单体项目而言，常使用配置文件管理所有配置，然后在微服务架构中，微服务的配置管理一般有如下一些特殊的需求：

1. 集中管理配置，一个微服务体系中的服务可能成百上千，大多数服务所需的配置其实大同小异，因此集中进行管理是非常有必要的。
2. 不同环境的不同配置，一个项目从开发到测试到上线，每次都是一套不同的环境，频繁修改配置文件不但麻烦，而且很容易在这个过程中出现失误。
3. 运行期间动态调整，例如我们可以根据微服务的负载情况临时调整数据源或者修改熔断器的一些配置等等。

为了实现满足上面所谈到的需求，我们需要使用SpringCloud中的配置中心组件，SpringCloudConfig为分布式系统提供了外部化的配置方案。SpringCloudConfig包含ConfigServer和ConfigClient两个部分。

ConfigServer是一个配置服务器，它用于管理应用程序各个环境下的配置文件，默认使用git作为配置文件的存储位置。同时也支持svn、本地文件系统等等。

ConfigClient在微服务启动时从ConfigServer服务器拉取配置文件信息，并进行缓存以此提高性能。

在开发过程中为了便于管理配置文件，所以大多数情况下都会选择将配置文件统一配置在本地。

配置中心服务器配置：

新建项目引入相关依赖包括web、eureka-client、config-server。

修改配置文件：

|  |
| --- |
| #配置中心端口 |
| server.port=8888 |
| spring.application.name=config-server |
| #从本地读取配置文件 |
| spring.profiles.active=native |
| #申明本地配置文件的存放位置，可以是项目路径也可以是本地文件路径 |
| spring.cloud.config.server.native.searchLocations=classpath:/files/ |
| #注册中心 |
| eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/,http://localhost:8762/eureka/ |

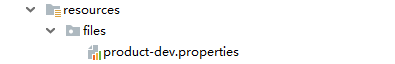
在resources下新建files文件夹用于保存配置文件，

配置文件的命名规则可以按照模块来分例如：

product-test，product-dev，product-online

也可以按照不同框架的配置来分，例如：

mysql-dev，mysql-test，mysql-online



启动类上添加注解：



配置中心客户端配置：

配置中心的客户端就是指需要从配置中心读取配置文件的各个微服务模块，我们只需要在以前的模块上加入config-client的配置即可。

引入依赖config-client

在resources下新建配置文件bootstrap.properties

配置信息如下：

|  |
| --- |
| #配置中心的地址 |
| spring.cloud.config.uri=http://localhost:8888/ |
| #对应的是配置文件规则中的前半部分 |
| spring.cloud.config.name=product |
| #对应的是配置文件规则中的后半部分，可以是多个，用逗号隔开。 |
| spring.cloud.config.profile=dev |

然后当我们启动配置中心以后，各个微服务模块就会按照我们的配置从配置中心统一的加载配置文件进行启动。