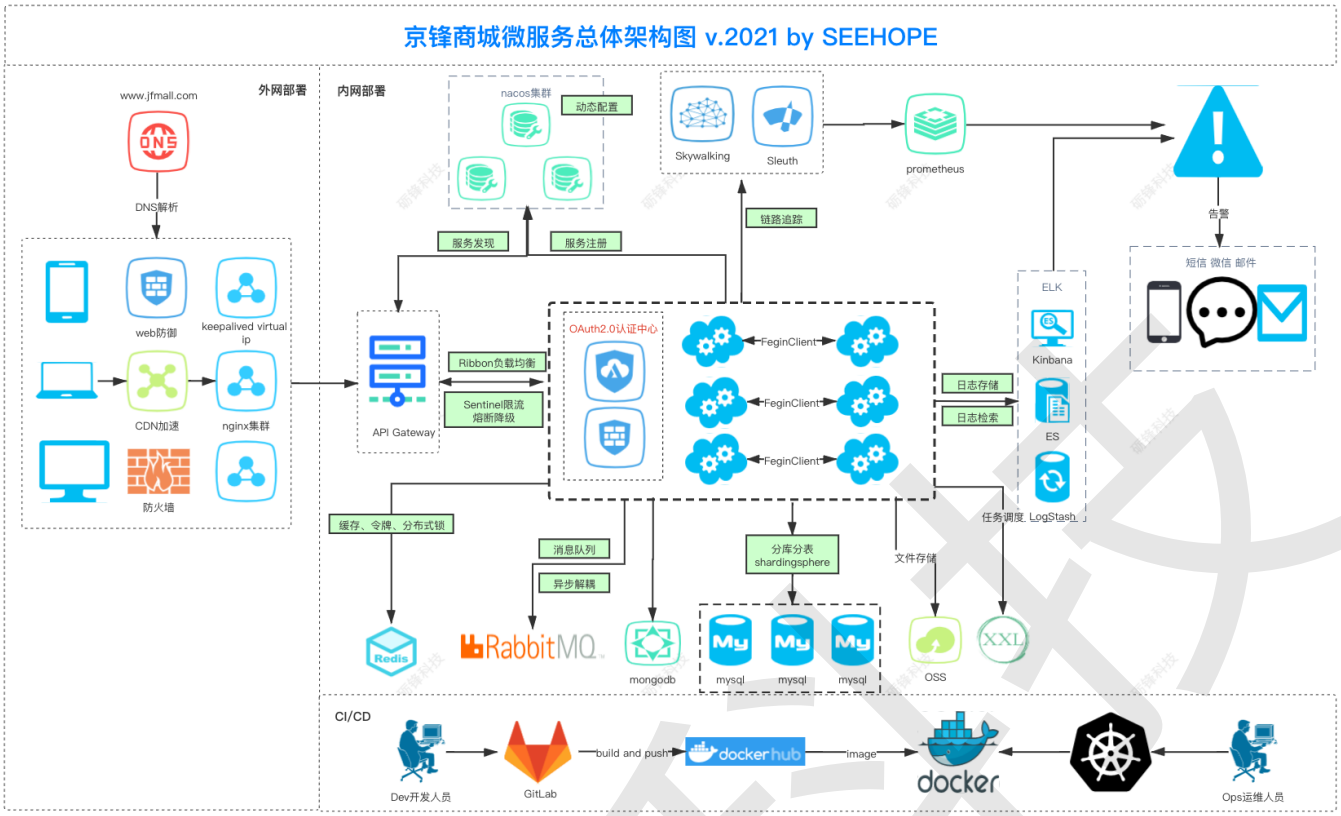
微服务项目-京锋购 06 - 微服务架构概述与后台基础搭建

京锋购

京东 X 砺锋 = 京锋购商城



回顾三阶段项目以及架构演进

我们在三阶段做 Dubbo 项目时已经知道什么是分布式以及什么是集群，这里不再赘述。

在我们的 Dubbo 项目中，我们的一个服务会负责几个功能，同时不同的服务可以分散部署在不同的机器上的独立运行，服务之间通过我们定义了好的接口和协议（RPC）进行交互调用。

当时我们使用的是面向 SOA（Service-Oriented Architecture） 的架构，也就是面向服务的架构。

> 注意：使用 Dubbo 不代表项目的架构就是 SOA，目前 Dubbo 可以很好地和 Spring Cloud 进行整合使用。

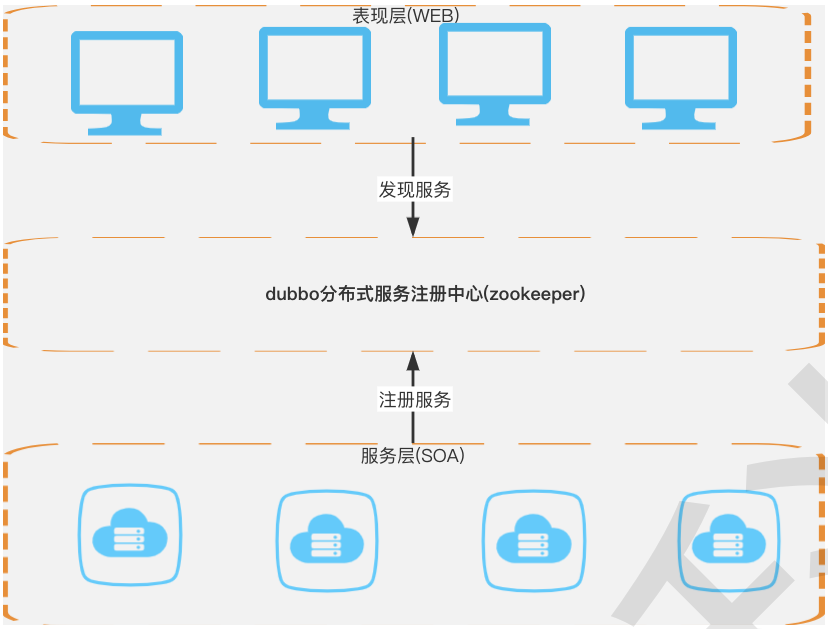
现在，我们要开始演进为微服务架构。

在此之前，我们需要回顾一下 SOA 架构的相关内容。

在 SOA 架构中，我们有 ESB（服务总线）的概念，ESB 负责服务之间的通信转发和接口适配。

要强调的是，在 SOA 实现中，ESB 处于核心地位，有很多专业的 ESB 厂商提供 ESB 中间件，例如 WebSphere 的 ESB、Oracle 的 ESB以及我们熟悉的阿里的 Dubbo 等。

下图很好地描述了我们 Dubbo 项目架构的核心部分：



如果对比我们即将使用的微服务架构，ESB 其实会显得有些“重”，因为微服务强调的是轻量级，迅捷，去中心化。

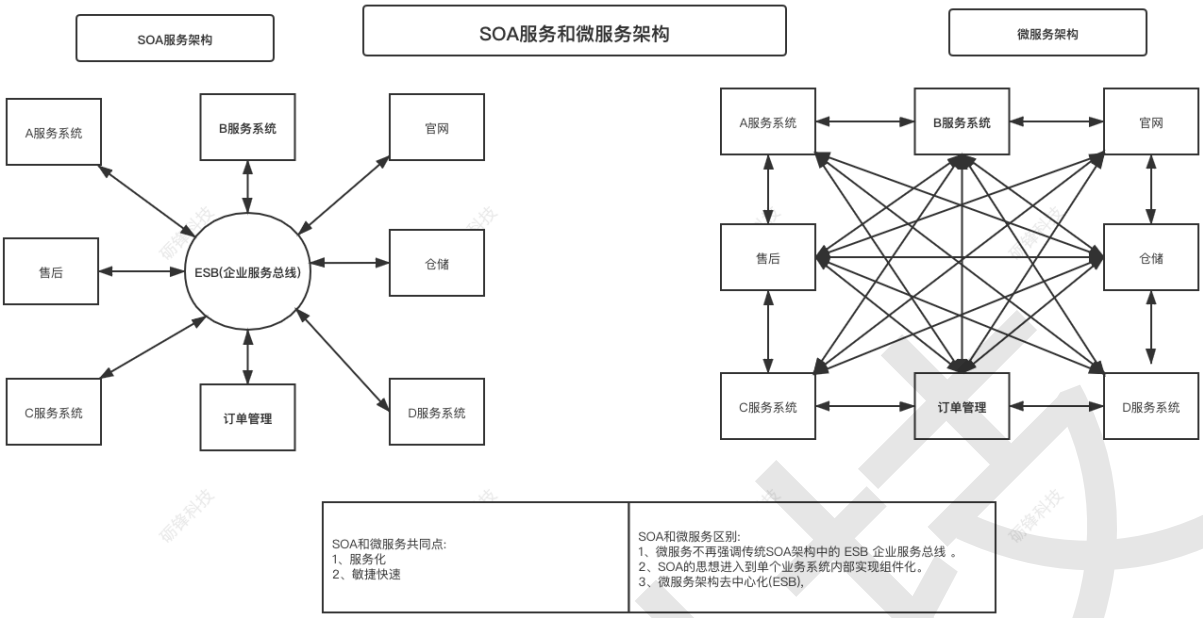
在微服务架构中，不需要 ESB，而通过 API 网关这样的技术来负责服务接口转发（回想我们学的 Spring Cloud Gateway）。

而 SOA 的设计思路是把一些组件和服务，通过服务总线组装，形成更大的应用系统（从小到大）；而微服务的设计思路是把应用拆分成独立自治的小的服务（从大到小）。

SOA 设计架构强调分层，通常会分为展现层、业务层、总线层和数据层。微服务架构中的服务更松散。

SOA 中的服务不强调业务领域的自治性，微服务架构强调基于领域的服务自治性。

微服务相比分布式服务来说，它的粒度更小，服务之间耦合度更低，由于每个微服务都由独立的小团队负责，因此它敏捷性更高。



总得来说，SOA 架构向微服务架构演化会是一种趋势，不过服务微服务化后带来的挑战也显而易见，比如服务粒度小，数量大，后期运维难度大增。

CAP 理论

目前来说，大型网站几乎都是分布式的。

> 注意：不管是 SOA 还是 微服务，指的是软件架构设计，而这里的分布式指的是系统部署方式，从这一点上看，SOA 和微服务都是分布式的。

分布式系统的最大难点，就是各个节点的状态如何同步。

CAP 理论是这方面的基本定理，也是理解分布式系统的起点。

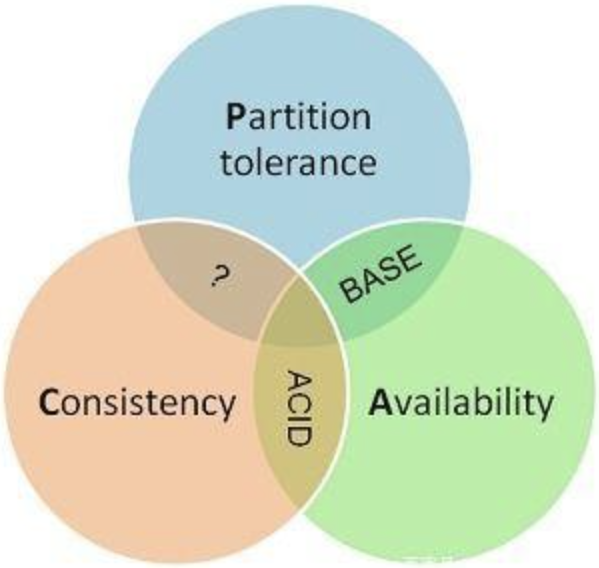
1998年，加州大学的计算机科学家 Eric Brewer 提出，分布式系统有三个指标。

- Consistency（满足一致性）
- Availability（可用性）
- Partition tolerance（分区容错性）

同时，Eric Brewer 认为这三个指标不可能同时做到，也就是三项中只能同时满足两项。

这就是 CAP 理论。

下面这张网络流行图很好的描述了这个理论：



Partition tolerance

大多数分布式系统都分布在多个子网络。每个子网络就叫做一个区（partition）。分区容错的意思是，区间通信可能失败。比如，一台服务器放在中国，另一台服务器放在美国，这就是两个区，它们之间可能无法通信。

一般来说，分区容错无法避免，因此可以认为 CAP 的 P 总是成立。CAP 定理告诉我们，剩下的 C 和 A 无法同时做到。

因为 P 的存在，所以：如果追求一致性，那么无法保证所有节点的可用性；
如果追求所有节点的可用性，那就没法做到一致性。

Consistency

一致性指的是写操作之后的读操作，必须返回一样的值。
举例来说，有两台服务器 A 和 B，某条记录是“年龄：18”，用户向服务器 A 发起一个写操作，将其改为“年龄：19”。
接下来，用户的向服务器 A 执行读操作得到“年龄：19”。

这就叫一致性。

但是如果此时用户的向服务器 B 执行读操作得到“年龄：18”。

这就不满足一致性了。

所以，为了让 B 也能得到“年龄：19”，就要在服务器 A 执行写操作的时候，让向服务器 B 发送一条消息，要求服务器B 也执行修改。

Availability

可用性是指只要收到用户的请求，服务器就必须给出回应。

用户可以选择向服务器 A 或 B 发起读操作。不管是哪台服务器，只要收到请求，就必须告诉用户，到底是“姓名：张三，年龄：18”还是“姓名：张三，年龄：19”，否则就不满足可用性。

CAP 面试题

一致性和可用性，为什么不可能同时成立?

答案很简单，因为可能通信失败（即出现分区容错）。

在什么场合，可用性高于一致性?

一般来说，网页的更新不是特别强调一致性。短时期内，一些用户拿到老版本，另一些用户拿到新版本，问题不会特别大。当然，所有人最终都会看到新版本。所以，这个场合就是可用性高于一致性。

热身面试题：为什么使用分布式、集群

快速回答：

- 分布式
  - 不同的模块独立部署，共同组成一套系统对外提供服务。
  - 用户不会感受到有多台服务器
- 集群
  - 提供一组相同服务的机器，可以通过负载均衡将请求转发到不同的服务器上处理

热身面试题：为什么使用微服务

快速回答几点：

避免单体应用演进为大型单体应用，基于业务边界进行服务微化拆分。

1. 每一个微服务都可以独立部署、运行、升级（不影响其他业务）
2. 技术(语言)、架构、业务都可以做到独立自治
3. 各模块系统功能松耦合
4. 可使用不同语言开发
5. 能够被拆分为各个小团队进行开发，可以更专注于业务逻辑

Spring Cloud 远程调用协议

微服务系统中，各个服务之间需要借助网络协议进行调用，当前使用的主流协议有两种：HTTP 和 RPC。

简单概述一下两者的相同点和不同点

- 相同点
  - 都是一种网络传输协议，规定了传输时的数据格式
  - 底层基于TCP
- 不同点
  - HTTP 协议
    - 只规定了网络传输时的数据格式，没有规定语言类型
    - Spring Cloud 用 HTTP，可跨语言
  - RPC 协议
    - RPC需要使用统一的语言互相调用（早期的WebService，现在流行的Dubbo）
    - RMI 是 java 端的 API

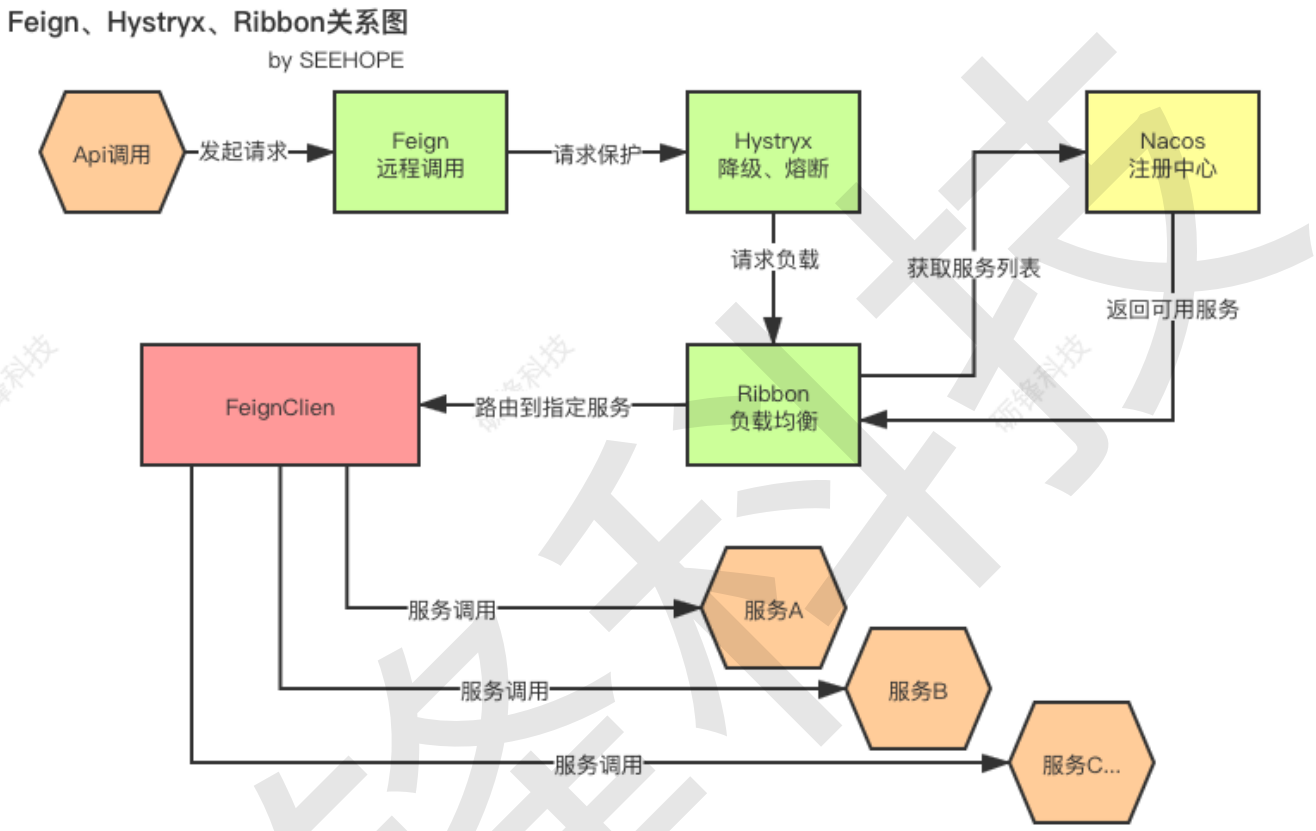
Spring Cloud 服务间调用的方式

RestTemplate 和 FeignClient 是当前主流的 Spring Cloud 服务间调用的两种方式：

- 共同点
  - 都是通过REST接口调用服务的http接口
  - 参数和结果默认都是通过jackson序列化和反序列化
  - Spring MVC 的 RestController 定义的接口，返回的数据都是通过 Jackson 序列化成JSON数据（Spring MVC 默认使用了 HttpMessageConverters）
- 不同点
  - RestTemplate
    - 使用起来较为麻烦，需要自己指定 Ribbon 的负载均衡，但参数较灵活，请求的路径可以使用程序灵活控制。
  - FeignClient

- 使用简单，默认集成了 Ribbon 负载均衡，无需自己配置，但参数不灵活，适用于写 API 的固定接口。

Feign 是一个声明式的 REST 客户端，它的目的就是让 REST 调用更加简单。



Spring Cloud 对 Feign 进行了封装，所以也支持了 SpringMVC 标准注解和 HttpMessageConverters。

反过来说，如果我们对 RestTemplate 进行封装，使用上也能做到和 Feign 一样简单。

目前来说，Feign 更加流行，但是注意：RestTemplate 有一个 Feign 没有的功能，那就是它可以调用微服务集群外的服务，因为它可以通过 IP 去调用。

关于负载均衡

在微服务架构中，微服务之间总是需要互相调用，以此来实现一些组合业务的需求。

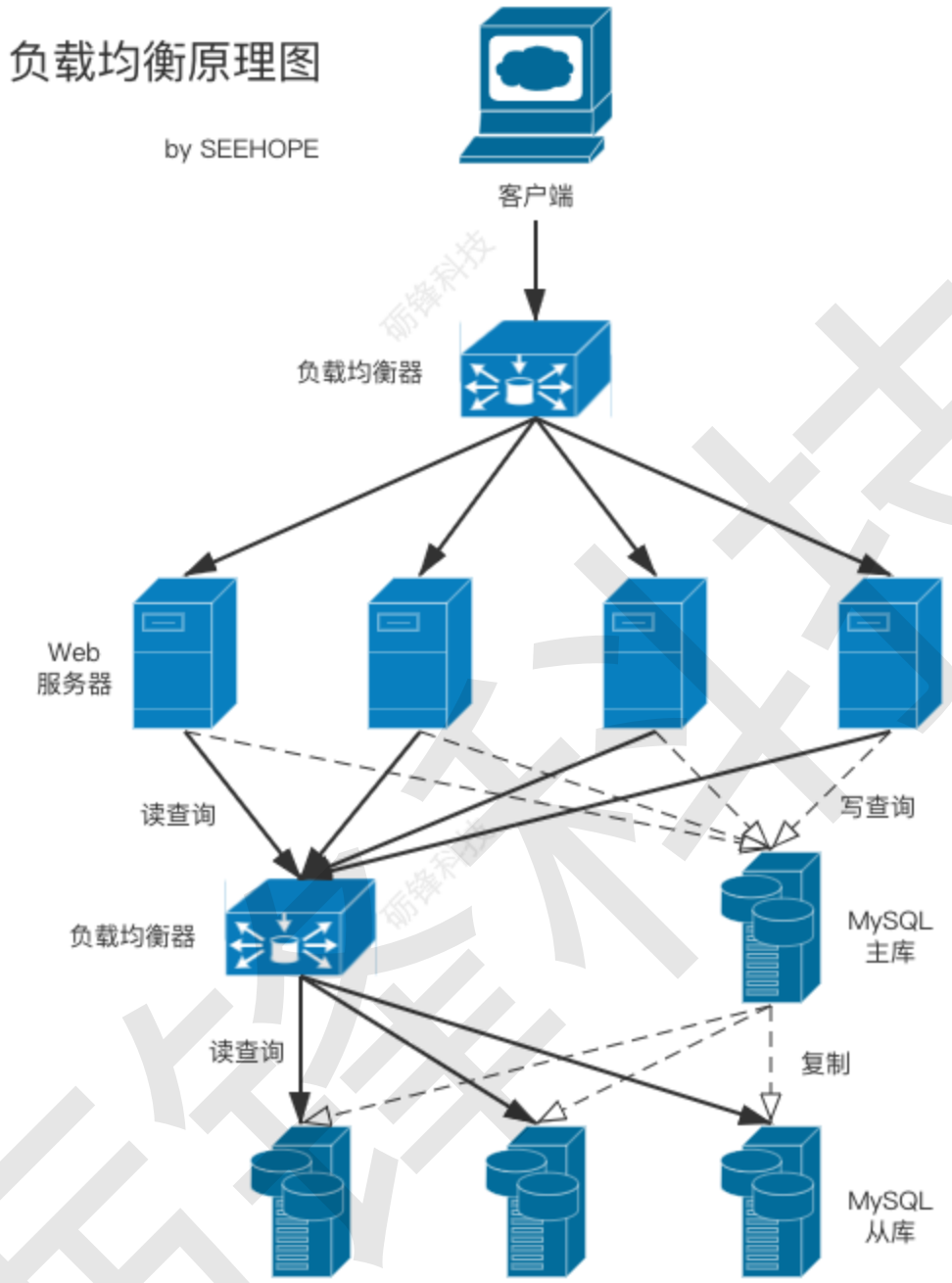
例如一个订单请求，由于订单里有用户信息，所以订单服务就得调用用户服务来获取用户信息。

负载均衡的目的是，相同服务部署到不同服务器，然后再通过 Nginx 技术手段将请求分发到不同服务器，由此来降低单一服务器的压力。

此时每个微服务都可能会存在有多个实例分布在不同的机器上，那么当一个微服务调用另一个微服务的时候就需要将请求均匀的分发到各个实例上，以此避免某些实例（服务器）负载过高，某些实例又太空闲。

在这种场景中，解决了上述问题的服务我们称为实现了负载均衡功能的服务，也称为负载均衡器。

负载均衡原理

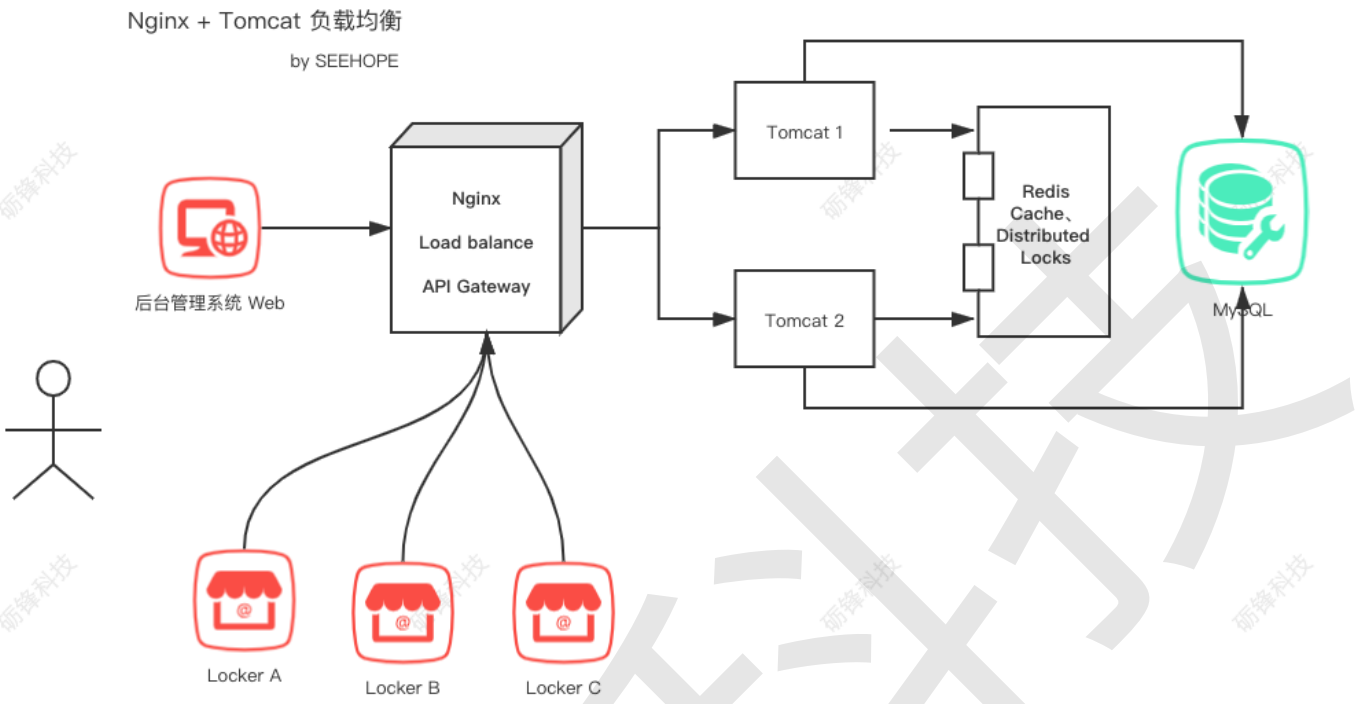


实现负载均衡主要的两种方式

服务端负载均衡

- 例如最经典的使用 Nginx 做负载均衡器。
- 用户的请求先发送到 Nginx，然后再由 Nginx 通过配置好的负载均衡算法将请求分发到各个实例上。
- 由于需要作为一个服务部署在服务端，所以该种方式称为服务端负载均衡。

如图：



客户端负载均衡

- 这种负载均衡方式是由发送请求的客户端来实现的，也是目前微服务架构中用于均衡服务之间调用请求的常用负载均衡方式。
- 因为采用这种方式的话服务之间可以直接进行调用，无需再通过一个专门的负载均衡器，这样能够提高一定的性能以及高可用性。
- 以微服务A调用微服务B举例，简单来说就是微服务A先通过服务发现组件获取微服务B所有实例的调用地址（回想一下我们当时做的 demo，我们把调用地址配置从 ip 改为域名），然后通过本地实现的负载均衡算法选取出其中一个调用地址进行请求。

常见的负载均衡算法

- 轮询
- 最小连接（优先选择最小连接数的服务器；要解决同IP请求分配到同一台服务器）
- 散列（同一个用户的请求会被发送到同一个服务器）

Spring Cloud 中的负载均衡

在 Spring Cloud 众多组件中也包含有负载均衡器，叫 Ribbon。

Ribbon 是一个基于 HTTP 和 TCP 客户端的负载均衡器。

回想我们之前引入 Feign 来进行服务调用的时候就配置了负载均衡，它内部其实使用了 Ribbon，只要使用@FeignClient时，Ribbon就会自动使用。

因为使用 Feign 是一个采用基于接口的注解的编程方式，所以，在 Spring Cloud 项目中，内部是使用 Ribbon 来进行客户端负载均衡的，但是是实际上会直接使用 Feign 组件，因为它更加简便。

Spring Cloud 服务注册中心

Spring Cloud 中使用的主流注册中心有：

- Eureka
- Zookeeper
- Consul
- Nacos

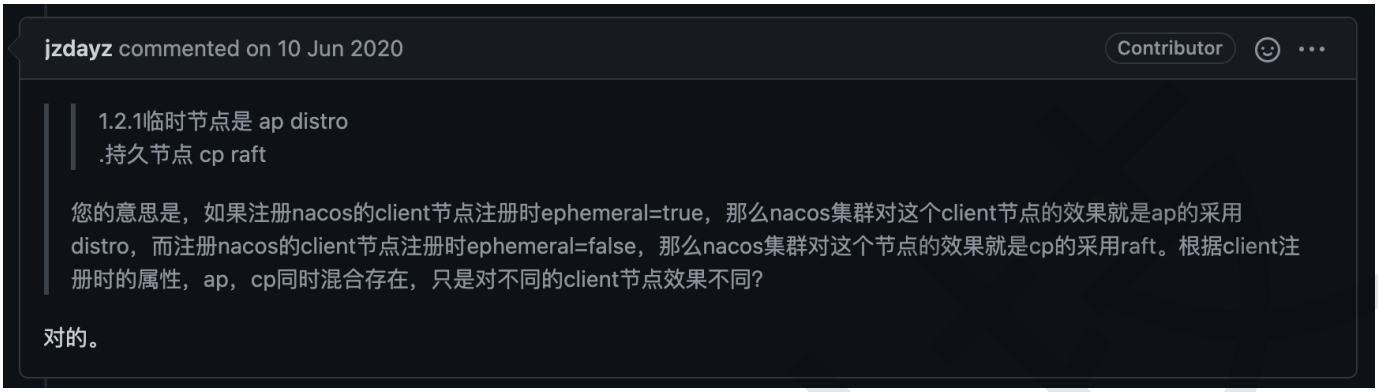
主流注册中心对比

	Nacos	Eureka	Consul	CoreDNS	Zookeeper
一致性协议	CP+AP	AP	CP	—	CP
健康检查	TCP/HTTP/MYSQL/Client Beat	Client Beat	TCP/HTTP/gRPC/Cmd	—	Keep Alive
负载均衡策略	权重/ metadata/Selector	Ribbon	Fabio	RoundRobin	—
雪崩保护	有	有	无	无	无
自动注销实例	支持	支持	支持	不支持	支持
访问协议	HTTP/DNS	HTTP	HTTP/DNS	DNS	TCP
监听支持	支持	支持	支持	不支持	支持
多数据中心	支持	支持	支持	不支持	不支持
跨注册中心同步	支持	不支持	支持	不支持	不支持
SpringCloud集成	支持	支持	支持	不支持	支持
Dubbo集成	支持	不支持	支持	不支持	支持
K8S集成	支持	不支持	支持	支持	不支持

注意：软件产品特性可能会改变。

nacos默认是 AP 模式吗?

偶尔看到的 github上的一个 issue：https://github.com/alibaba/nacos/issues/3000



Nacos 我们已经学完了，简单总结：
Nacos 支持基于 DNS 和基于 RPC 的服务发现。在 Spring Cloud 中使用 Nacos，只需要先下载 Nacos 并启动 Nacos server，然后简单的配置就可以完成服务的注册发现。

HashiCorp 事件

国外的商业软件公司 HashiCorp 旗下的知名的开源软件有 Consul、Vagrant、Terraform 等。

其官网曾经宣布：不允许中国境内使用、部署和安装该企业旗下的【企业版】产品和软件。

Terms of Evaluation for HashiCorp Software

Before you download and/or use our enterprise software for evaluation purposes, you will need to agree to a special set of terms ("Agreement"), which will be applicable for your use of the HashiCorp, Inc.'s ("HashiCorp", "we", or "us") enterprise software.

PLEASE NOTE THAT CHINESE EXPORT CONTROL REGULATIONS PROHIBIT HASHICORP FROM SELLING OR OTHERWISE MAKING THE ENTERPRISE VERSION OF VAULT AVAILABLE IN THE PEOPLE'S REPUBLIC OF CHINA. FOR THAT REASON, HASHICORP'S VAULT ENTERPRISE SOFTWARE MAY NOT BE USED, DEPLOYED OR INSTALLED IN THE PEOPLE'S REPUBLIC OF CHINA WITHOUT WRITTEN AGREEMENT BY HASHICORP.

PLEASE NOTE THAT THE SOFTWARE MAY NOT BE USED, DEPLOYED OR INSTALLED IN THE PEOPLE'S REPUBLIC OF CHINA.

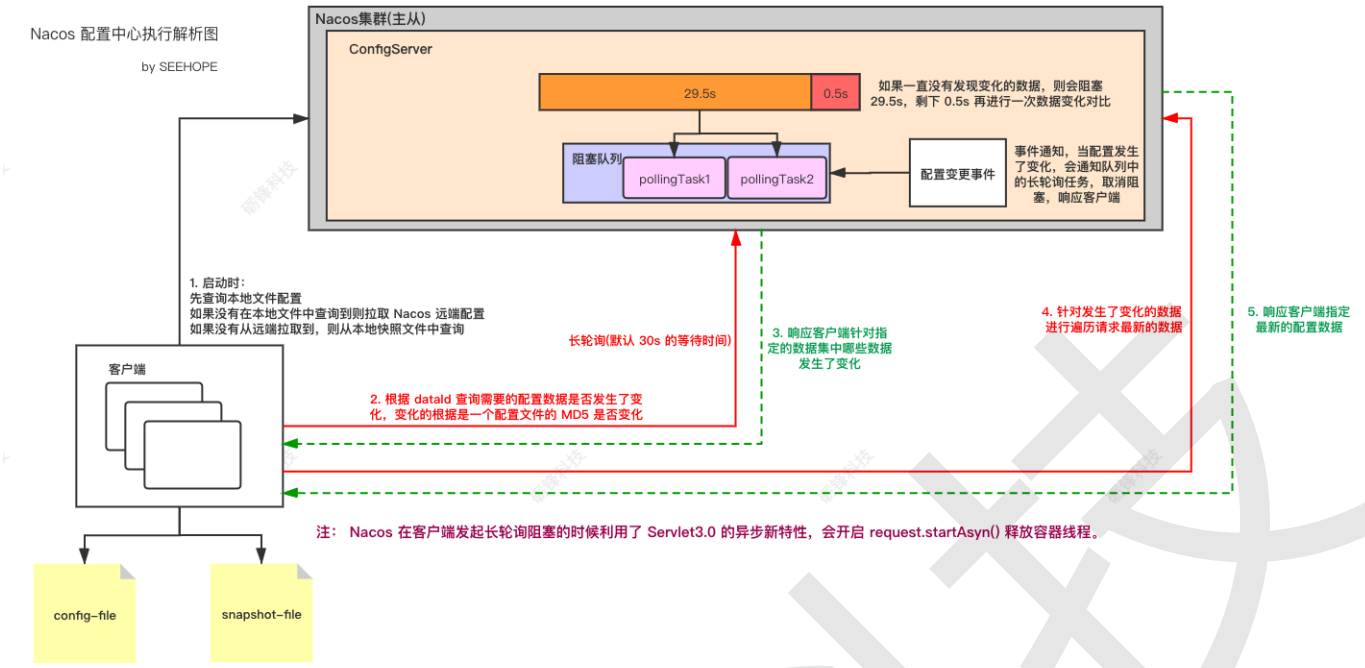
但后来官方立马对文章进行了修改和解释,指出只是因为国内管制,不允许使用其公司的 VAULT。

Spring Cloud 配置中心

Nacos 自身带有配置中心的功能，而携程开源的 Apollo 配置据说点麻烦，ps: 本人没用过。

目前来说，我们会使用 Spring Cloud Alibaba 的 Nacos 来做为配置中心。

Nacos 配置中心执行解析



服务熔断和服务降级

服务熔断

服务熔断的作用类似于我们家用的保险丝，当某服务出现不可用或响应超时的情况时，为了防止整个系统出现雪崩，暂时停止对该服务的调用。

通常做法是设置服务的超时，当被调用的服务经常失败到达某个阈值，我们可以开启断路保护机制，后来的请求不再去调用这个服务，而是本地直接返回默认的数据。

服务降级

服务降级是从整个系统的负荷情况出发和考虑的，应对某些负荷会比较高的情况。

为了预防系统某些功能（业务场景）出现负荷过载或者响应慢的情况，在其内部暂时舍弃对一些非核心的接口和数据的请求，而直接返回一个提前准备好的 fallback（退路）错误处理信息（回想我们的 ProviderFallback 类）。

也就是让非核心业务降级运行（降级:某些服务不处理，或者简单处理。）

虽然提供的是不全的服务，但却保证了整个系统的稳定性和可用性。

熔断和降级的对比

- 相同点
  - 目标一致 都是从可用性和可靠性出发，为了防止系统崩溃；
  - 用户体验类似 最终都让用户体验到的是某些功能暂时不可用；
- 不同点
  - 触发原因不同
    - 服务熔断一般是某个服务（下游服务）故障引起，而服务降级一般是从整体负荷考虑；

两者都可能是由于请求积压引起。

请求积压：底层服务响应慢，造成上层服务响应慢，造成整个服务响应慢

跨域

是什么

跨域指的是浏览器不能执行其他网站的脚本。它是由浏览器的同源策略造成的，是浏览器对 JavaScript 施加的安全限制。

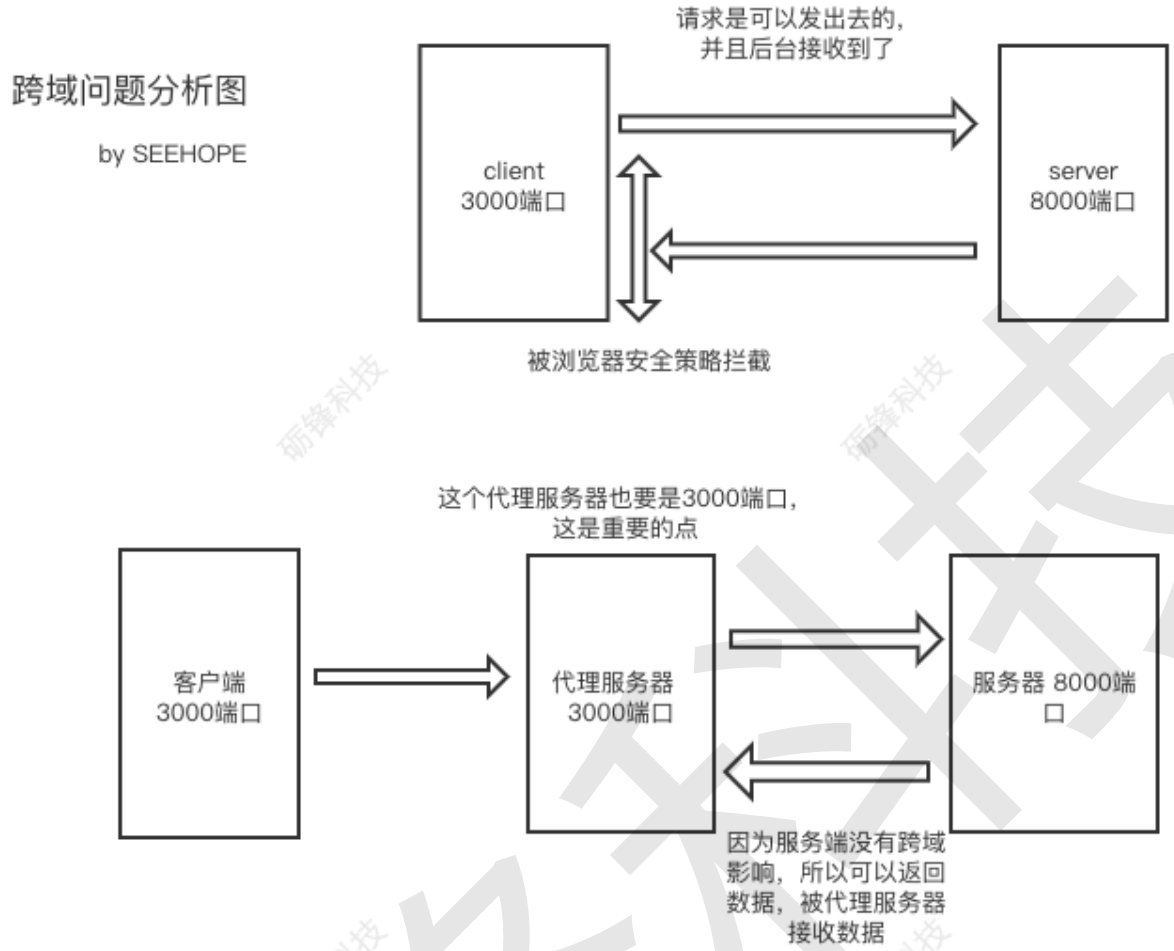
同源策略

同源策略是指协议，域名，端口都要相同，其中有一个不同都会产生跨域；

跨域问题分析与解决

使用 Nginx 代理

使用 nginx 部署为同一域：



配置当次请求允许跨域

添加响应头

- Access-Control-Allow-Origin：支持哪些来源的请求跨域
- Access-Control-Allow-Methods：支持哪些方法跨域
- Access-Control-Allow-Credentials：跨域请求默认不包含cookie，设置为true可以包含 cookie
- Access-Control-Expose-Headers：跨域请求暴露的字段
  - CORS请求时，XMLHttpRequest 对象的 getResponseHeader() 方法只能拿到6个基本字段：
    - Cache-Control
    - Content-Language
    - Content-Type
    - Expires
    - Last-Modified
    - Pragma
  - 如果想拿到其他字段，就必须在 Access-Control-Expose-Headers 里面进行指定

- Access-Control-Max-Age：表明该响应的有效时间为多少秒。在有效时间内，浏览器无 须为同一请求再次发起预检请求。请注意，浏览器自身维护了一个最大有效时间，如果 该首部字段的值超过了最大有效时间，将不会生效。

Java 复制代码

```java
/**
 * 配置跨域
 */
@Configuration
public class CorsConfiguration {

    @Bean
    public CorsResponseHeaderFilter corsResponseHeaderFilter() {
        return new CorsResponseHeaderFilter();
    }

    @Bean
    public CorsWebFilter corsWebFilter() {
        UrlBasedCorsConfigurationSource source = new
UrlBasedCorsConfigurationSource();
        CorsConfiguration corsConfiguration = new CorsConfiguration();

        corsConfiguration.addAllowedHeader("*");
        corsConfiguration.addAllowedMethod("*");
        corsConfiguration.addAllowedOrigin("*");
        corsConfiguration.setAllowCredentials(true);
        corsConfiguration.setMaxAge(600L);

        source.registerCorsConfiguration("/**", corsConfiguration);
        return new CorsWebFilter(source);
    }
}
```

常见电商模式

电商行业的一些常见模式：

- B2C：商家对个人，如：亚马逊、当当等
- C2C平台：个人对个人，如：闲鱼、拍拍网、ebay
- B2B平台：商家对商家，如：阿里巴巴、八方资源网等
- O2O：线上和线下结合，如：饿了么、电影票、团购等
- P2P：在线金融，贷款，如：网贷之家、人人聚财等。

- B2C平台：天猫、京东、一号店等

项目搭建

项目结构说明

microservice

上一个内容我们聊了微服务，现在我们的关注点放在 microservice 这里，整个商城的微服务分为两大类：

1. 后台微服务
2. 前台数据接口微服务

后台微服务

- jfmall_pms
  - 商品管理系统（product）
  - 端口：18081
- jfmall_ums
  - 用户管理系统（user）
  - 端口：18082
- jfmall_wms
  - 仓库管理系统（warehouse）
  - 端口：18083
- jfmall_oms
  - 订单管理系统（order）
  - 18074
- jfmall_sms
  - 营销管理系统（sale）
  - 端口：18085

暂定这么多。

前台数据接口微服务

- cart
  - 购物车微服务
- order
  - 订单微服务

- search
  - 搜索微服务
- item
  - 商品详情微服务
- member
  - 会员微服务

第三方微服务

- 物流
- 短信
- 支付
- 云存储
- 等等。

数据层

- MySQL（分库分表中间件：Mycat）
- Redis
- ElasticSearch
- Activemq/Rabbitmq
- 阿里OSS/七牛KODO
- 等等

后台管理前端

- jfmall-admin（1000）
- 前台门户：jfmall-shop （2000）
- 网关：jfmall-gateway（8888）

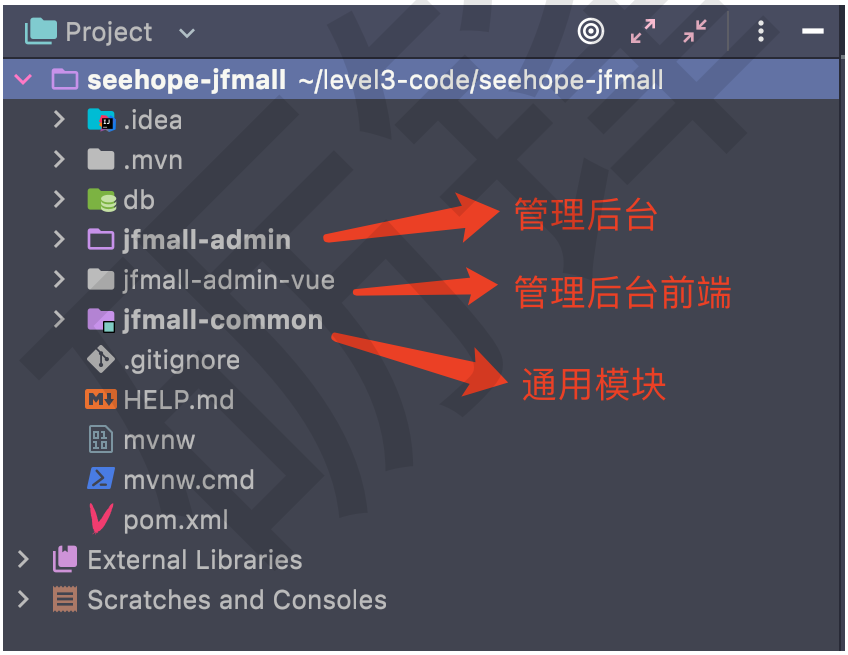
准备数据环境

把数据文件导入数据库：

导入数据库之后：

使用 IDEA 创建项目初始工程

创建 seehope-jfmall 作为父工程，然后把给大家准备好的基础工程 jfmall-admin 和 jf-common 拷贝到里面去。



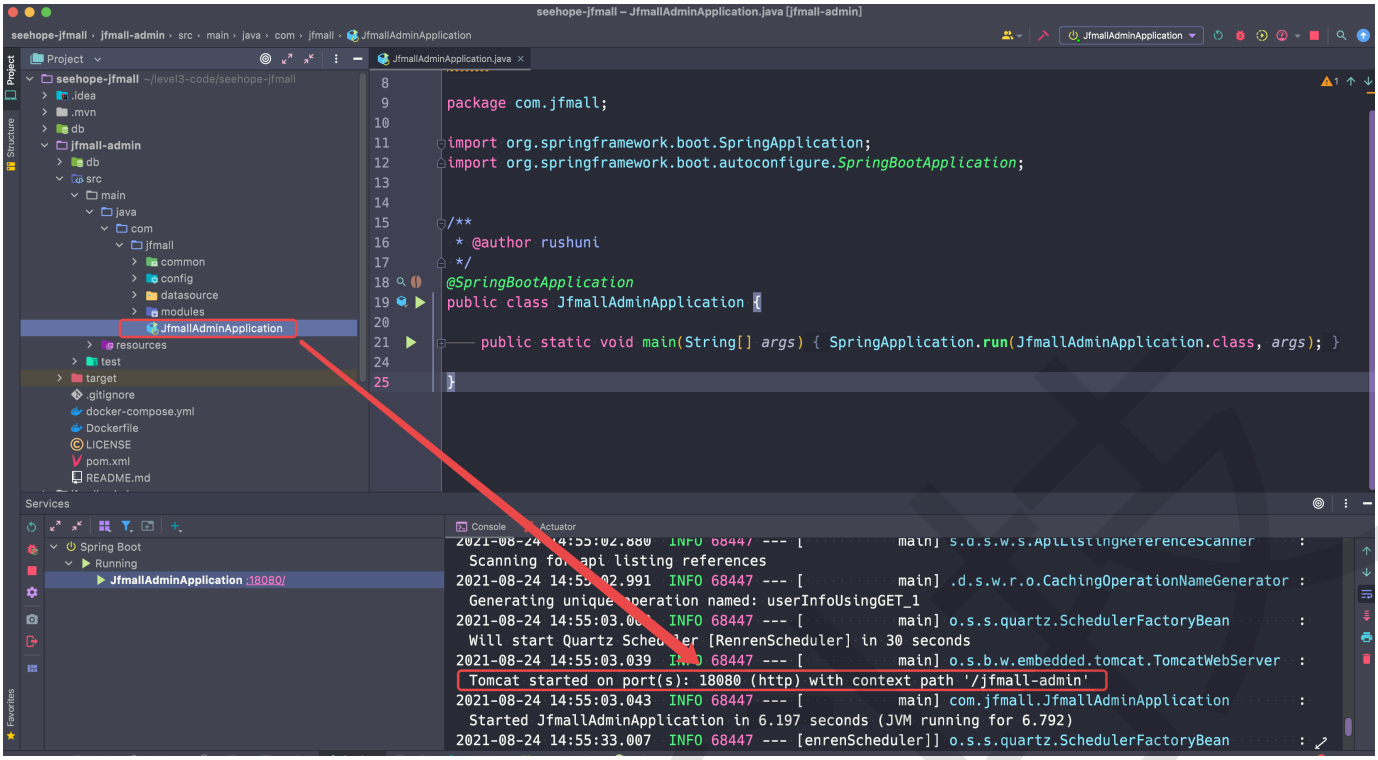
使用 Git 管理项目代码

项目依赖分析

- seehope-jfmall工程
  - 父工程，在 pom.xml 中统一管理了常用依赖的版本号
- jfmall-common
  - 包含了通用的实体类及工具类，所有工程都依赖于该 common 工程
- jfmall-admin
  - 后台管理控制台菜单管理系统。
- Spring Boot 的相关依赖：
  - data-redis
  - web
  - 。。。
- springCloud 的相关依赖有：
  - alibaba-nacos-discovery
  - alibaba-nacos-config
  - alibaba-sentinel
  - openfeign
  - 。。。

启动商城管理后台

启动后台



启动前端

注意，需要在后台前端文件夹下执行安装依赖的命令：

```shell
npm install
```

如果安装依赖过程中出现安装过慢、报错等情况，实在解决不了就改用 cnpm 命令吧。

cnpm 命令需要先安装：

```shell
npm install -g cnpm --registry=https://registry.npm.taobao.org
```

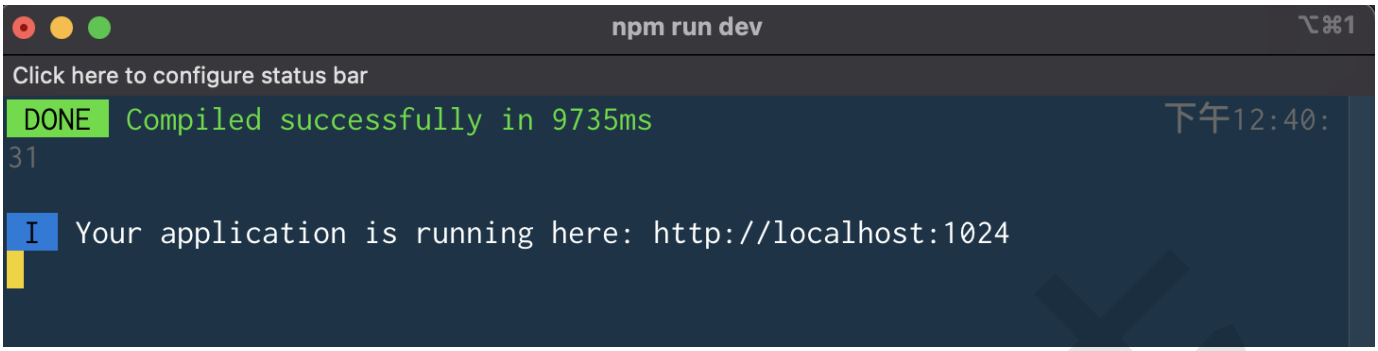
然后再使用 cnpm 来安装依赖：

```shell
cnpm install
```

然后尝试启动：

```shell
npm run dev
```

端口指定为 1000，但是如果 1000 不可用，那么会自动切换到一个可用端口，这里是 1024：



测试