# Spring Cloud & Spring Cloud Alibaba

## 什么是微服务：

微服务不是指spring cloud，格局大一点！微服务是分布式架构的一种，所谓分布式架构就是指对服务做拆分，而拆分的过程必然会产生各种各样的问题要去解决，而spring cloud仅仅是解决了服务拆分时的服务治理的问题，至于其他的分布式更加复杂的问题并没有解决！

所以一个完整的分布式技术栈，并不仅仅是spring cloud！

那么微服务保底包含哪些知识呢？

### 服务实例

微服务要做的第一件事就是对应用做拆分：传统的单体架构应用，所有的功能都写在一起，随着业务越来越复杂，代码耦合也就越来越多，将来升级维护就越来越来困难！所以大型应用一般都必须做拆分，微服务会根据业务功能把一个大型的单体项目拆分成许多个独立的项目，每个项目完成独立的功能，开发独立开发、部署独立部署！我们把这些独立的项目称为服务。

### 注册中心

一个大型的服务往往会包含成百上千的服务，最终形成一个服务集群。

而一个业务往往需要多个服务来完成，比如一个请求来了，可能先要调用服务a，a服务又调用服务b，b服务还要调用服务c。。。。，当业务越来越多，越来越复杂的时候，这些服务之间的调用关系也会越来越复杂，如此复杂的调用关系要靠人去维护、记住是不可能的，那么微服务中必定会有一个组件：注册中心，用来记录和维护微服务中每一个服务的信息（ip、端口、功能等），当一个服务需要调用另外一个服务的时候，只需要去找注册中心就行了！从注册中心去拉取对应的信息。

### 配置中心

同时，随着服务越来越多，每个服务都有自己的配置文件，将来如果要去更改配置，难道要去逐个服务修改吗？这就太麻烦了，所以微服务中还应该会有一个配置中心，可以去统一管理整个服务群里成千上万的配置，如果以后有配置需要变更，只需要找到配置中心就可以，配置中心回去通知相关的微服务，从而实现配置的热更新！

### 网关组件

微服务运行起来以后，用户就可以来访问服务了，这个时候还需要网关组件：因为一个大型应用微服务成百上千，用户怎么知道要访问那个服务呢？而且也不是随便什么人都可以访问服务，就像小区，小区门口往往会有门卫，看门大爷，不能随便什么人来了都让进，访客码掏出来，验证成功了才让你进！访客告诉大爷要访问谁，大爷告诉你几栋几楼几号！所以服务网关组件一方面是对用户身份进行校验；另一方面可以把用户请求路由到具体的服务，当然了，在路由的过程中还可以做一些负载均衡。

### 分布式缓存

服务接收到用户请求，该访问数据库就去访问数据库，之后再把查询到的数据返回给用户，数据库肯定无法抗住海量用户的高并发请求，因此，微服务架构的项目中往还需要加入缓存，缓存简单来说就是把数据库数据放到内存，内存查询效率肯定比数据库磁盘快很多，而且缓存也不是的单体缓存，为了应对高并发，还得要做成分布式缓存，即也是一种集群，用户请求先到缓存，缓存没有命中，再去查询数据库。

### 分布式搜索

以后业务中还会有一些复杂的搜索，简单查询可以走缓存，一些海量数据的复杂搜索、统计分析，缓存是做不了了，这个时候就需要用到分布式搜索功能，数据库在分布式应用中的功能就是主要做一些数据的写操作，以及事务类型的对数据安全较高的数据存储。

### 消息队列组件

在微服务中还需要异步通讯作用的消息队列组件，因为在分布式应用中，即微服务中，业务往往会跨越多个服务，一个请求来了，先要调用服务a，a服务又调用服务b，b服务还要调用服务c。。。。，整个业务的链路会很长，调用时长就会等于每个服务的执行时长的和，性能会有一定的下降，而异步通讯的意思就是：一个请求来了，先要调用服务a，而a服务不是去调用服务b和服务c，而是发消息通知服务b、服务c，告诉服务b、服务c去执行业务，服务a就直接结束了，此时业务链路就变短了，响应时间也缩短了，吞吐能力就变强了，所以异步通讯可以大大提高服务的并发，一些如秒杀等高并发场景往往可以用到！

### 分布式日志服务

当然，在如此庞大的微服务应用中，万一出现了什么问题，异常的排查是很困难的，所以在微服务运行中，还要引入两个组件来解决服务的异常定位：分布式日志服务组件：作用是可以统计整个微服务应用集群的成千上万服务的运行日志，统一地去做存储、统计、分析，将来服务运行出现问题就比较好定位。

### 系统监控、链路追踪

系统监控和链路追踪组件：可以实时监控整个微服务应用集群中每一个服务节点的运行状态、cpu的负载、内存占用等，一旦出现问题，直接可以定位到具体的方法、栈信息，那么就可以快速地定位到异常的所在。

### 持续集成

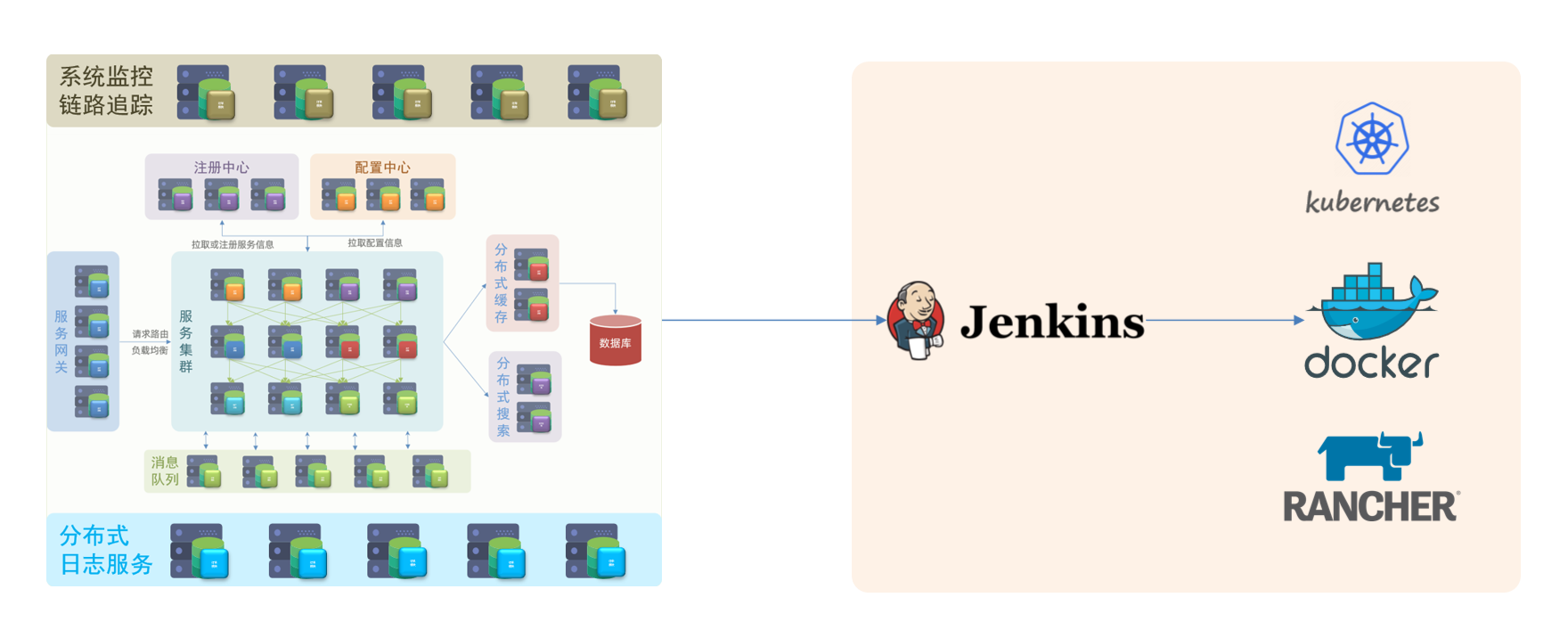
如此放大、复杂的微服务集群，部署再如何部署，还是和单体应用一样人工一个一个去部署吗？这显然是不现实的！所以微服务集群还需要做自动化部署，那么就可以用Jenkins，Jenkins可以对微服务项目做自动化的编译，并基于docker做打包形成镜像，在基于kubernets或者是rancher这样的技术去实现自动化部署，这就是所谓的持续集成。

## 微服务技术栈

以上就是完成的微服务技术栈：

微服务 = 网关服务 + 注册中心 + 配置中心 + 服务调用 + 缓存组件 + 搜索组件 + 异步通讯组件 + 分布式日志组件 + 系统监控和链路追踪组件 + 持续集成

显然微服务不是简单的spring cloud！



### 微服务技术栈五部分

其实微服务百分之二十的知识足以应付百分之八十的工作！微服务技术栈可以分为五部分：

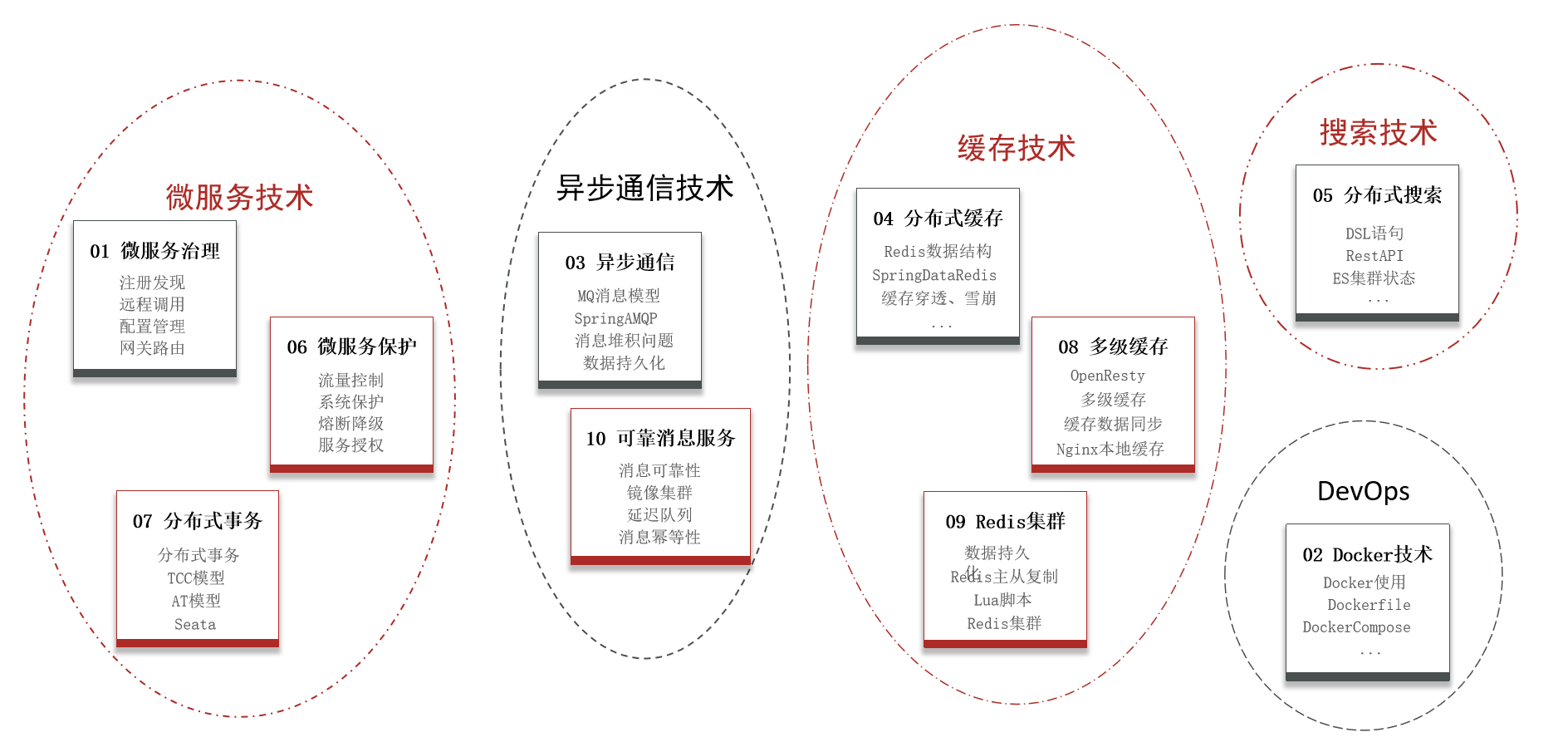
微服务治理；

缓存技术；

异步通信技术；

分布式搜索技术；

持续集成DevOps技术；



#### 微服务治理：

为服务治理其实就是我们所说的spring cloud干的事，就是spring cloud这个框架包含的技术。

包括：注册发现 负载均衡 远程调用 配置管理 网关路由 流量控制 服务授权 熔断降级 分布式事务 TCC模型 AT模型 Seata，。。。

#### 异步通信技术：

MQ消息模型 SpringAMQP 消息堆积问题 消息可靠性 仲裁队列 延迟队列 镜像集群 数据持久化，。。。

#### 缓存技术：

SpringDataRedis OpenResty 缓存穿透、雪崩 Redis主从赋值 缓存数据同步 nginx本地缓存 Redis持久化 Redis分片集群 多级缓存分层 Lua脚本 Redis数据结构，。。。。

#### 搜索技术：

DSL语句 ES集群 RestAPI 集群脑裂 竞价排名 聚合统计 自动补全 地理坐标 拼音分词，。。。。

#### DevOps：

Docker的使用 Dockerfile DockerCompose Kenkins GrayLog

SkyWalking Kubernetes，。。。。

## 微服务框架介绍：

### 微服务架构的演变：

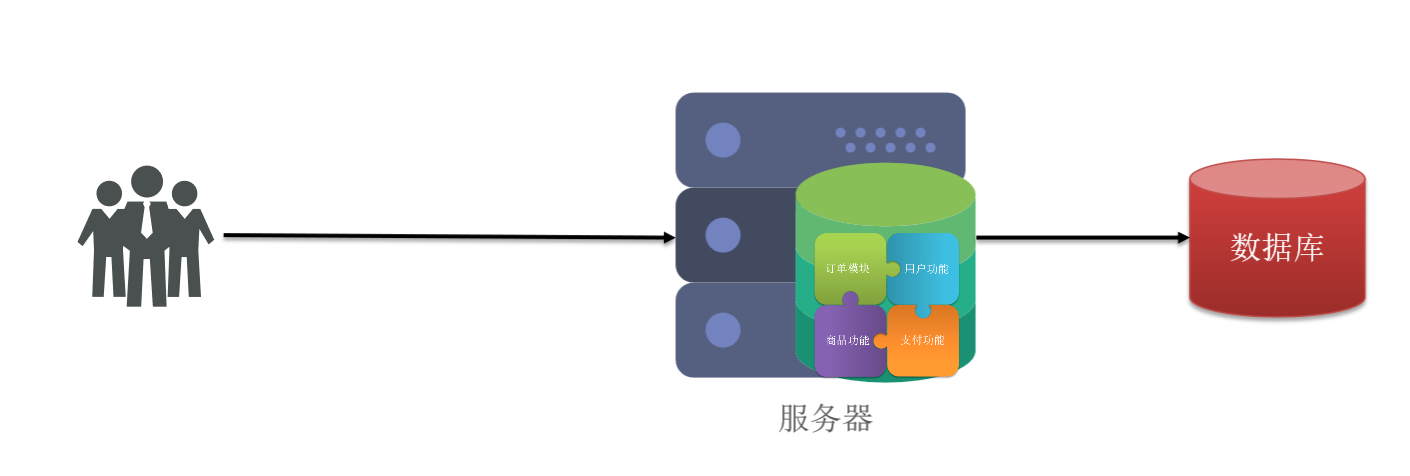
#### 单体架构：

将业务的所有功能集中在一个项目中开发，打包成一个包部署！

优点：

1：架构简单，不用去做复杂的架构设计理念；

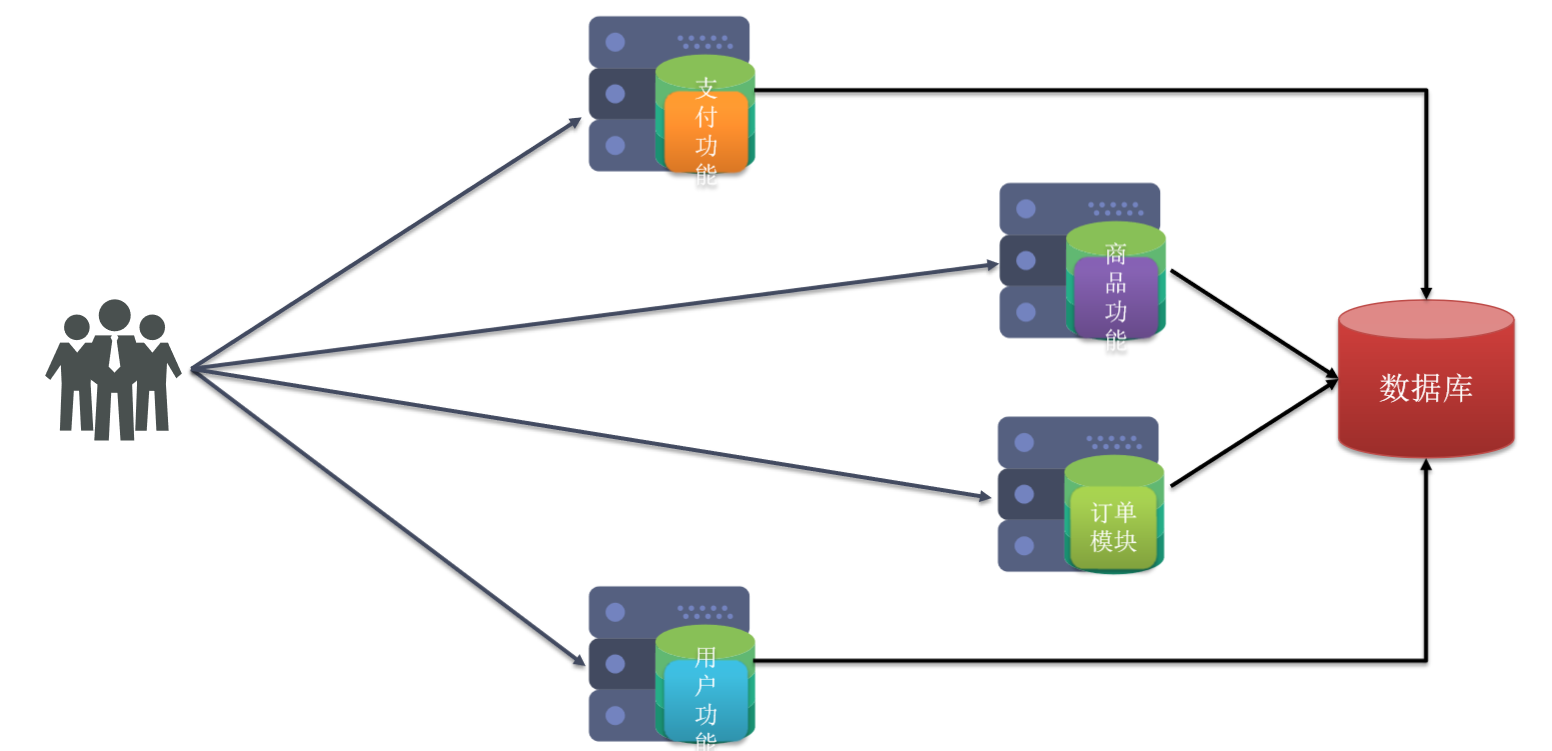
2：部署成本低，打成一个包到服务器一运行，用户就能访问，访问量多了，大不了加多两台机器，形成负载均衡的集群；



单体适合企业内部使用的项目，因为单体项目有个严重的缺陷，就是耦合度高，即所有代码写在一个项目里面，如果事大型的互联网项目：拼多多、淘宝，业务可不就是两三个模块了，往往是成百上千的业务模块，代码量以数十万行、数百万行计，这个时候项目光编译打包，可能就要用几个小时了，效率太低，而且模块太多，单体项目在开发过程中往往你中有我，我中有你，边界会越来越模糊，你改了一个地方的代码，很有可能导致其他模块的代码跟着改变，这个时候你可能就不敢乱动代码了，牵一发而动全身。所以耦合度高是不适合大型项目的。大型项目一定要做分布式架构！

#### 分布式架构：

分布式架构：根据业务功能对系统进行拆分，每个业务模块作为独立项目开发，称为一个服务。



优点：

降低服务耦合；

有利于服务升级拓展；

问题：

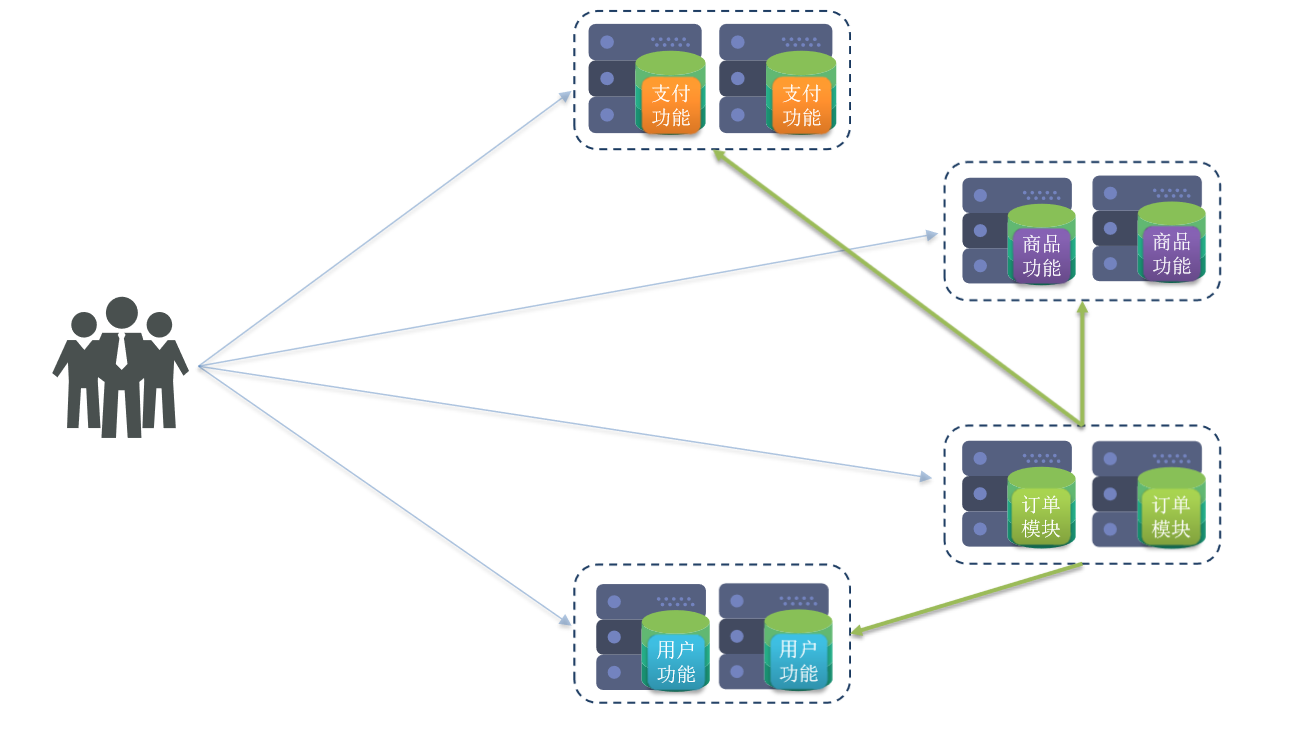
1：服务如何拆分；

2：服务集群地址如何维护；

3：服务之间如何远程调用；

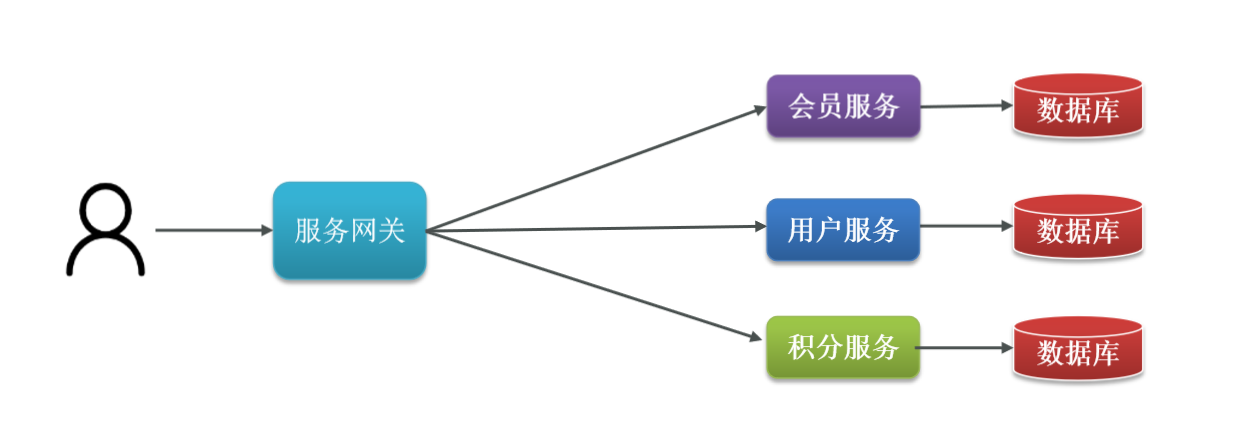
4：服务健康状态如何感知；

5：服务如何部署；



#### 微服务架构：

微服务是一种进过良好架构设计的分布式架构方案。



微服务架构特征：

1：单一职责：微服务拆分粒度要小，每一个服务都对应唯一一个业务能力，做到单一职责，比卖你重复业务开发；

2：面向服务：微服务对外暴露业务接口，以便其他服务调用功能；

3：自治：团队独立、技术独立、开发独立、数据独立、部署独立。

4：隔离性强：服务调用做好隔离、容错、降级、避免出现级联问题。

#### 总结：

##### 单体架构特点？

简单方便，高度耦合，扩展性差，适合小型项目：如学生管理系统；

##### 分布式架构特点：

松耦合，扩展性好，但架构复杂，难度大，适合大型互联网项目，例如京东、淘宝

##### 微服务：

一种良好的分布式架构方案，优点差分粒度更小，服务更独立，耦合度更低；缺点架构非常复杂，部署、运维、监控难度提高。

### SpringCloud：微服务的一种最佳落地实现

#### 微服务技术：

微服务这种方案需要技术框架来落地，我们最常用的就是springcloud和阿里巴巴的dubbo。不管是什么公司的解决方案，既然都是微服务的技术框架，所以所包含的组件、实现的功能基本是一致的：

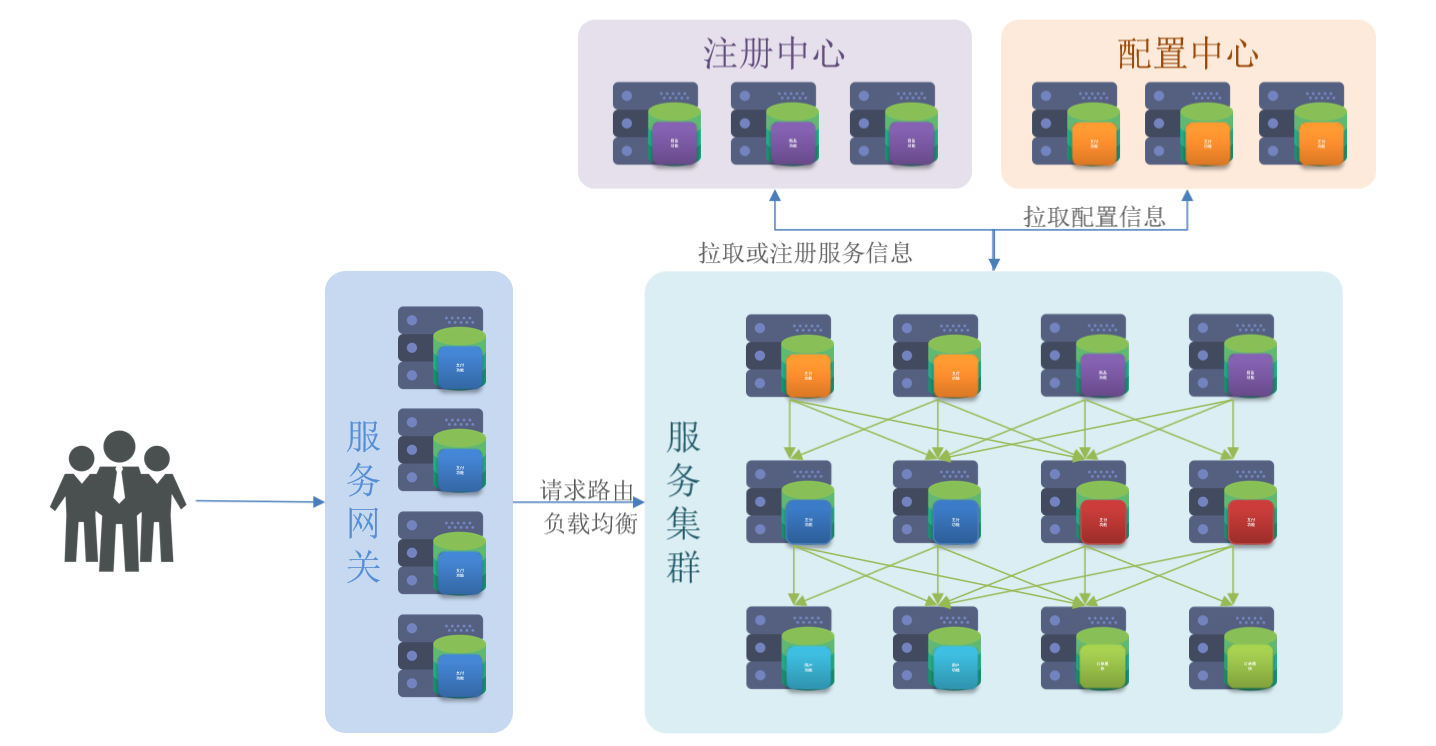
1：都需要去做微服务的拆分，形成微服务的集群，而集群中的每一个服务都要遵循单一职责的原则，并且要面向服务，对外暴露接口，这样服务之间就可以做相互的调用，只不过不同微服务技术框架实现这些接口可能会有差异。

2：不管是哪一种技术，服务之间的调用关系错综复杂，必定不能人为去维护，太复杂了，所以在微服务里往往会有一个注册中心，可以去维护微服务里面每个节点的信息，并且去监控这些节点的状态；

3：另外将来随着微服务越来越多，如果有配置需要进行修改我们手动去微服务里去改，显然是不现实的，因此在微服务里还会有配置中心，可以统一去管理整个微服务群的配置，如果将来有配置的变更，配置中心可以利用通知的方式让对应的服务监控到变化，从而实现配置的热更新。

4：将来微服务一旦部署上线，必然要用户访问，那么如此之多的微服务，用户怎么知道要访问哪一个？所以微服务群往往还会需要一个网关作为入口，用户访问网关，然后由网关把用户请求路由到微服务群驴踢的服务，在路由过程中，还可以去做负载均衡。

5：并且路由的时候，或者服务之间相互调用的过程中，还要做好服务的容错处理，避免因为服务故障带来级联失败，还要做好服务保护、隔离、降级等等这些措施。



以上聊到的服务注册中心、配置中心、服务网关、服务之间调用关系、服务容错、服务保护，这些功能都需要对应的技术去实现，springcloud和阿里巴巴的dubbo在实现时是有差异的。

#### 微服务技术对比：

##### Dobbo

Dubbo最早在2012年左右就已经开源出来了，2012年微服务技术在国内可能连听都没听过，所以Dubbo并不是严格意义上的微服务技术，在2012年，dubbo的核心功能就是服务的远程调用以及注册发现，所以Dobbo中技术体系并不完整。

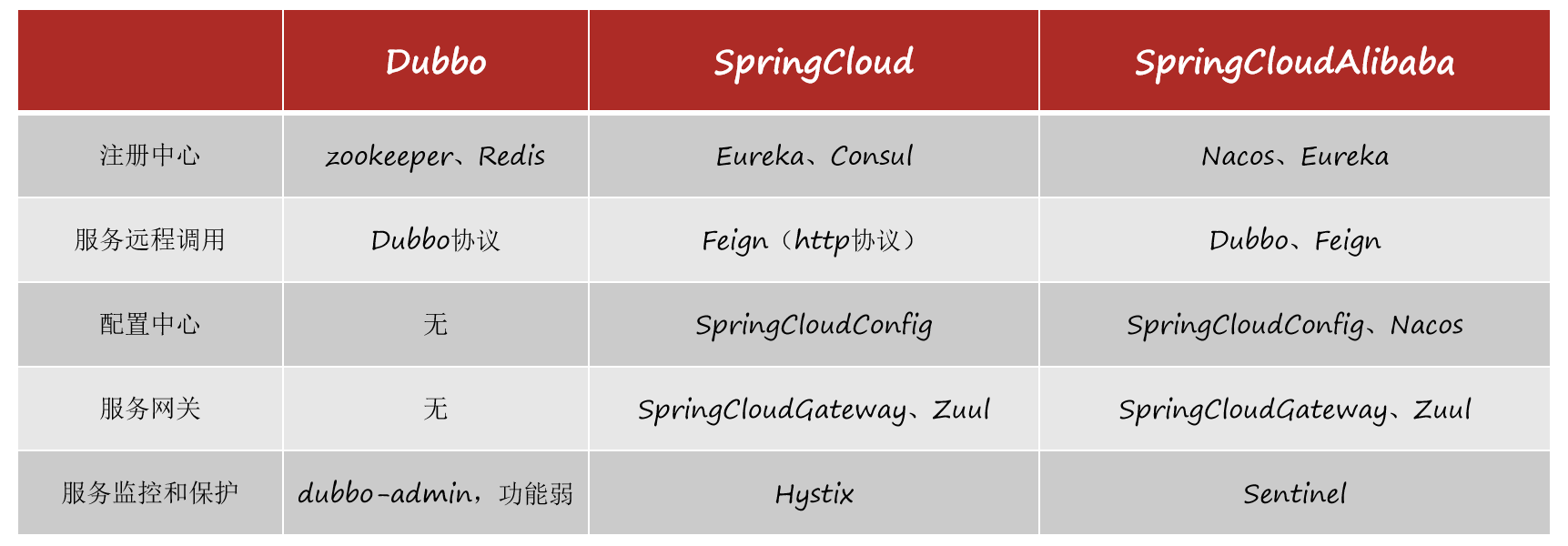
##### SpringCloud

2015到2017年这段时间，是微服务概念就、微服务技术井喷的时候，但是一直没有一个技术一统江湖，知道SpringCloud横空出世。SpringCloud并不是发明了什么东西，而是整合！SpringCloud把全世界各公司开源的微服务技术整合起来了，而后形成了一套完整的微服务技术体系：有注册中心、配置中心、服务网关、服务远程调用、服务监控和保护。

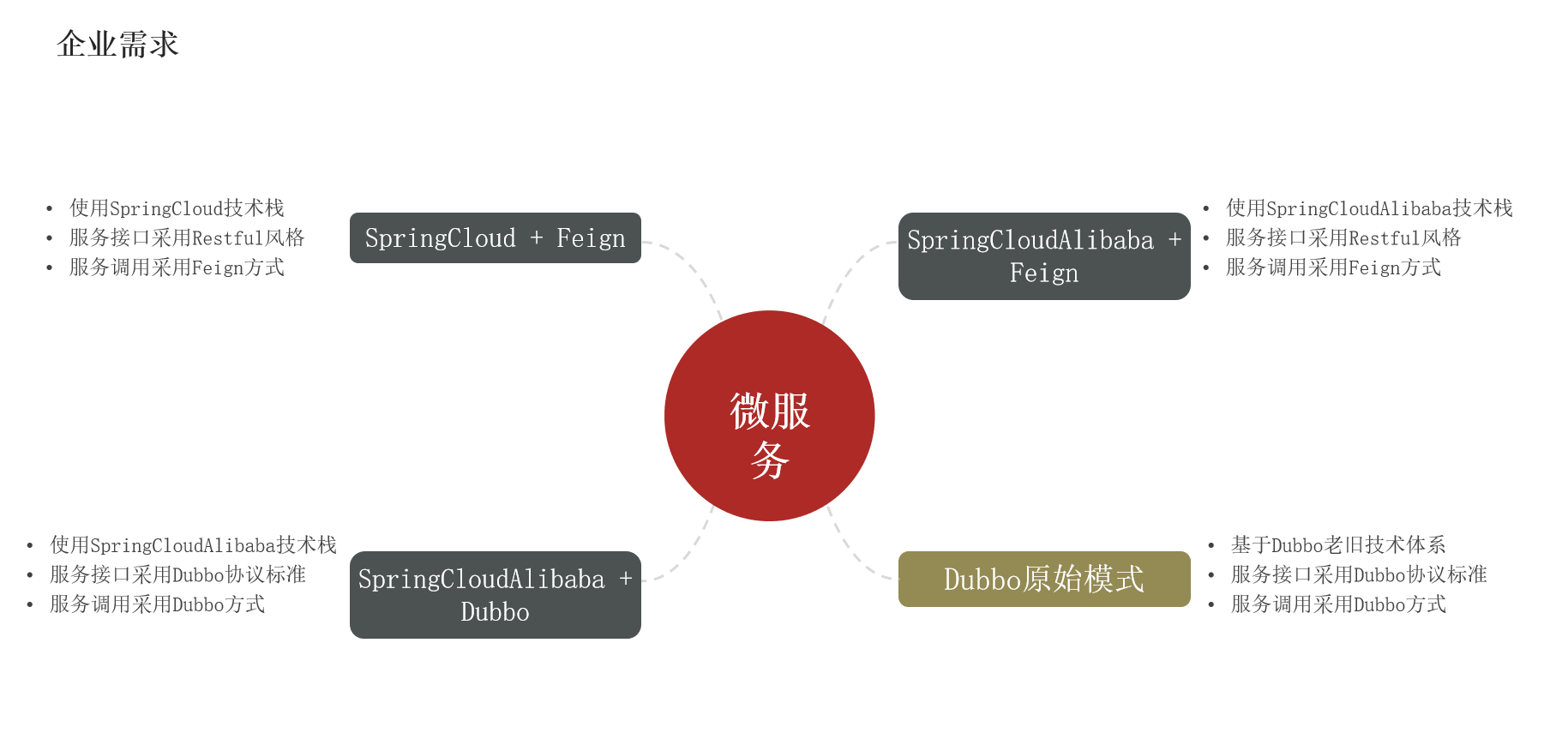
##### SpringCloudAlbaba

SpringCloudAlbaba以SpringCLoud开头，说明SpringCloudAlbaba是SpringCLoud的一部分。SpringCloudAlbaba是实现了SpringCLoud的标准接口，所以你会发现SpringCloudAlbaba中很多的技术组件用起来和SpringCloud当中的没什么区别。为什么没区别？因为实现了统一的接口规范啊！

从下图可以看到SpringCloudAlbaba甚至兼容SpringCLoud中的eureka、兼容feign、兼容gateway网关等等。SpringCloudAlbaba更适合中国程序员的体质！



企业中会碰到下图中四种情况，图中就是为了告诉你不用怕，：



## Spring Cloud

### 开箱即用的Spring Cloud

Spring Cloud是国内使用最广泛的微服务框架。Spring Cloud集成了各种微服务功能组件，并基于Spring Boot实现了这些组件的自动装配，从而提供了良好的开箱即用体验！



问：SpringCloud都是Spring从其他公司开源的组件整合进来的，为什么要用Spring整合的，而不是直接用官方原生的？

答：其实因为SpringCloud里有个非常重要的SpringBoot！SpringBoot最擅长的就是自动装配，而SpringCloud就是把官方原生开源的组件整合进来，并且基于SpringBoot做了自动装配，你只需要拿过来就能用，无需去做复杂的配置！用起来非常舒服，这就是为什么SpringCloud把微服务推广开来的原因！

### SpringCloud与SpringBoot的版本兼容关系如下：

SpringCloud与SpringBoot，SpringCloudAlibaba与SpringCloud都是有版本对应关系的！不对应代码可能会报错或者是用不了！

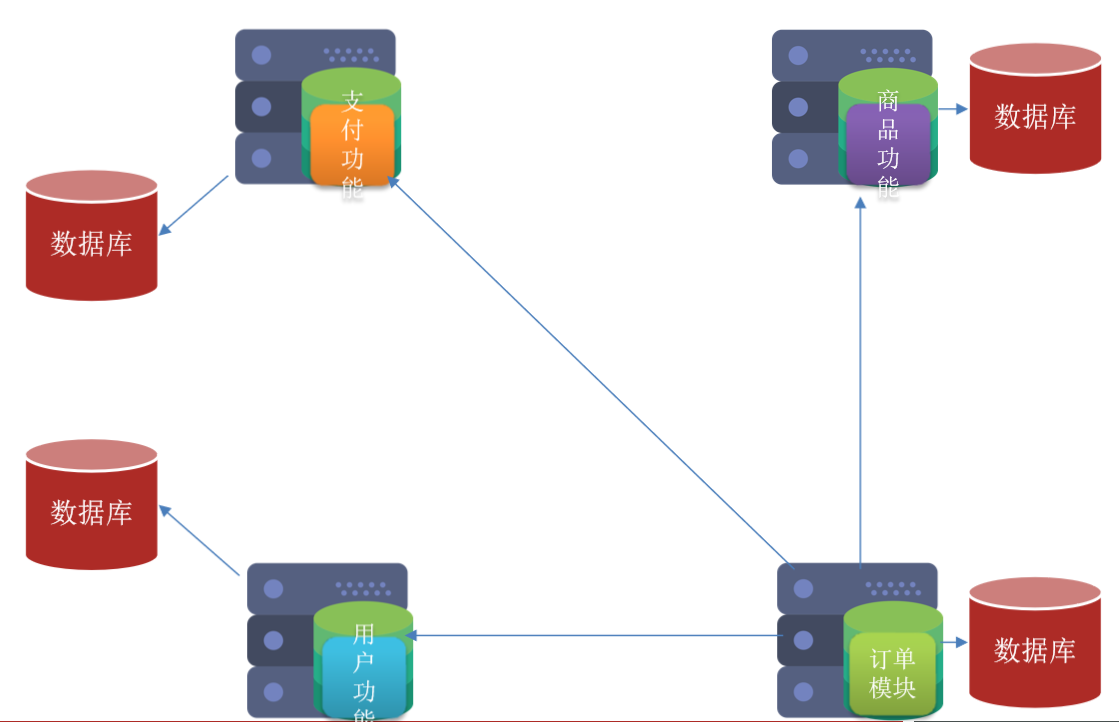
参考：<https://www.jianshu.com/p/96b49b3f5275>



## 服务的拆分以及远程调用

### 服务拆分

服务拆分说起来很简单，一个单体架构按照功能模块惊醒拆分，变成多个服务就行了！



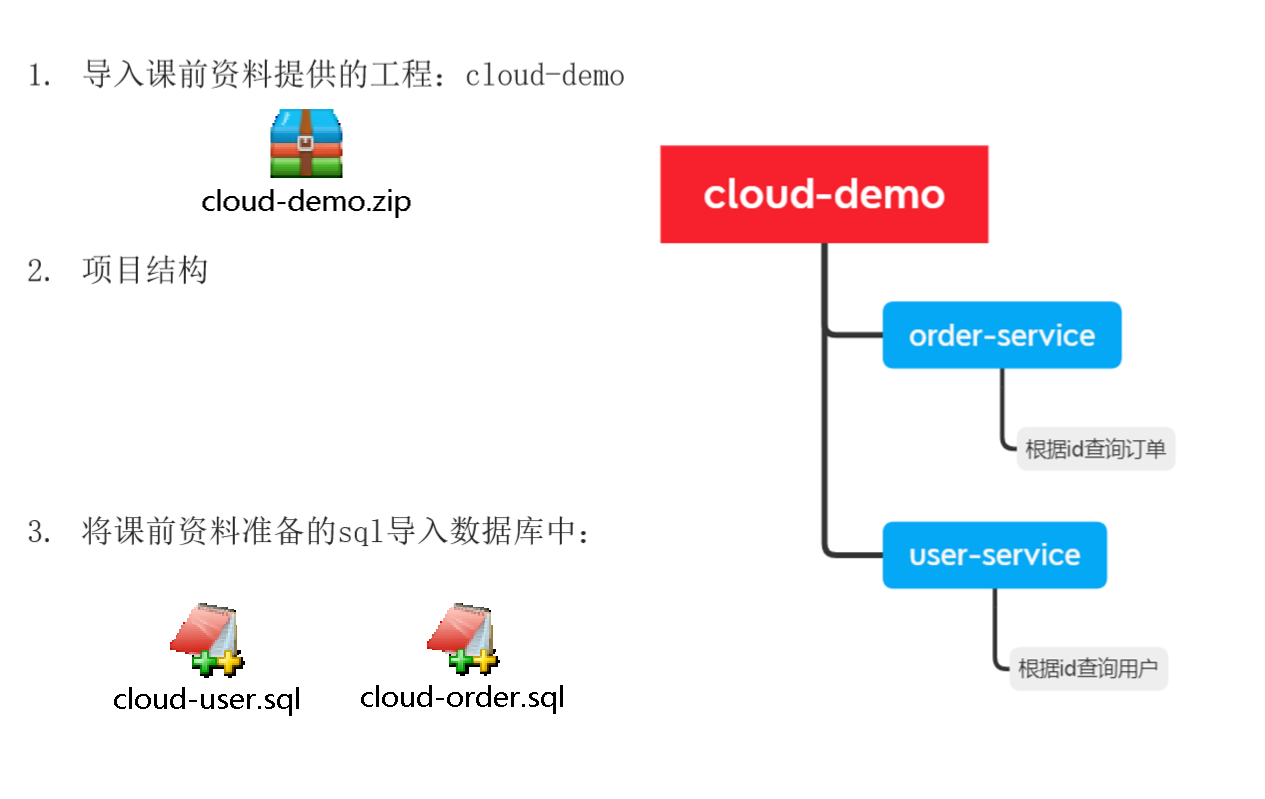
服务拆分注意事项：

1：不同微服务，不要重复开发相同业务，其实就是单一职责；

2：微服务数据独立，不要访问其他微服务的数据库，其实就是自治；

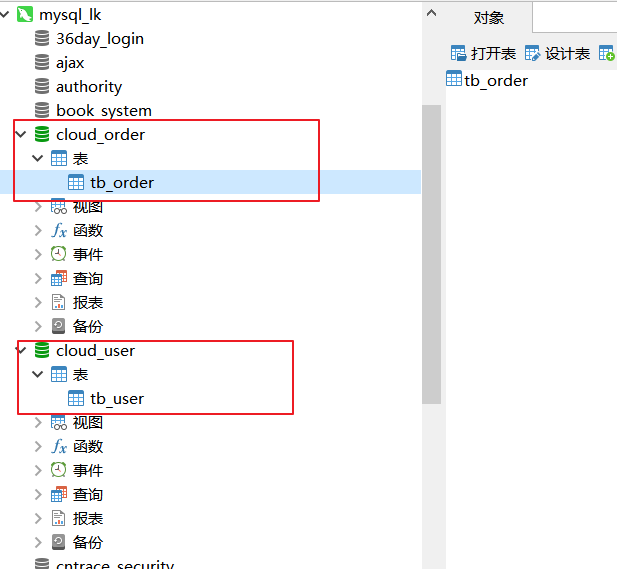
3：微服务可以将自己的业务暴露为接口，供其他微服务调用，其实就是面向服务。

### 服务拆分demo步骤：



1：看可以从项目结构看出，业务是解除了耦合的，order-service做订单相关，user-service做用户相关；

2：数据层面，也要遵循规定做好分离，订单业务查询的表是订单表，用户查询的表是用户表，将来在实际生产部署时，一定会把这两张表部署在不同的数据库里，不过现在是测试，条件有限，所以就把来个那张表放在一个MySQL数据库的不同的database中。



#### 服务拆分注意事项总结：

1：微服务需要根据业务模块拆分，做到单一职责，不要重复开发相同业务；

2：微服务可以将业务暴露为接口，供其他微服务调用；

3：不同微服务都应该有自己独立的数据库；

### RestTemplate实现服务之间的调用：

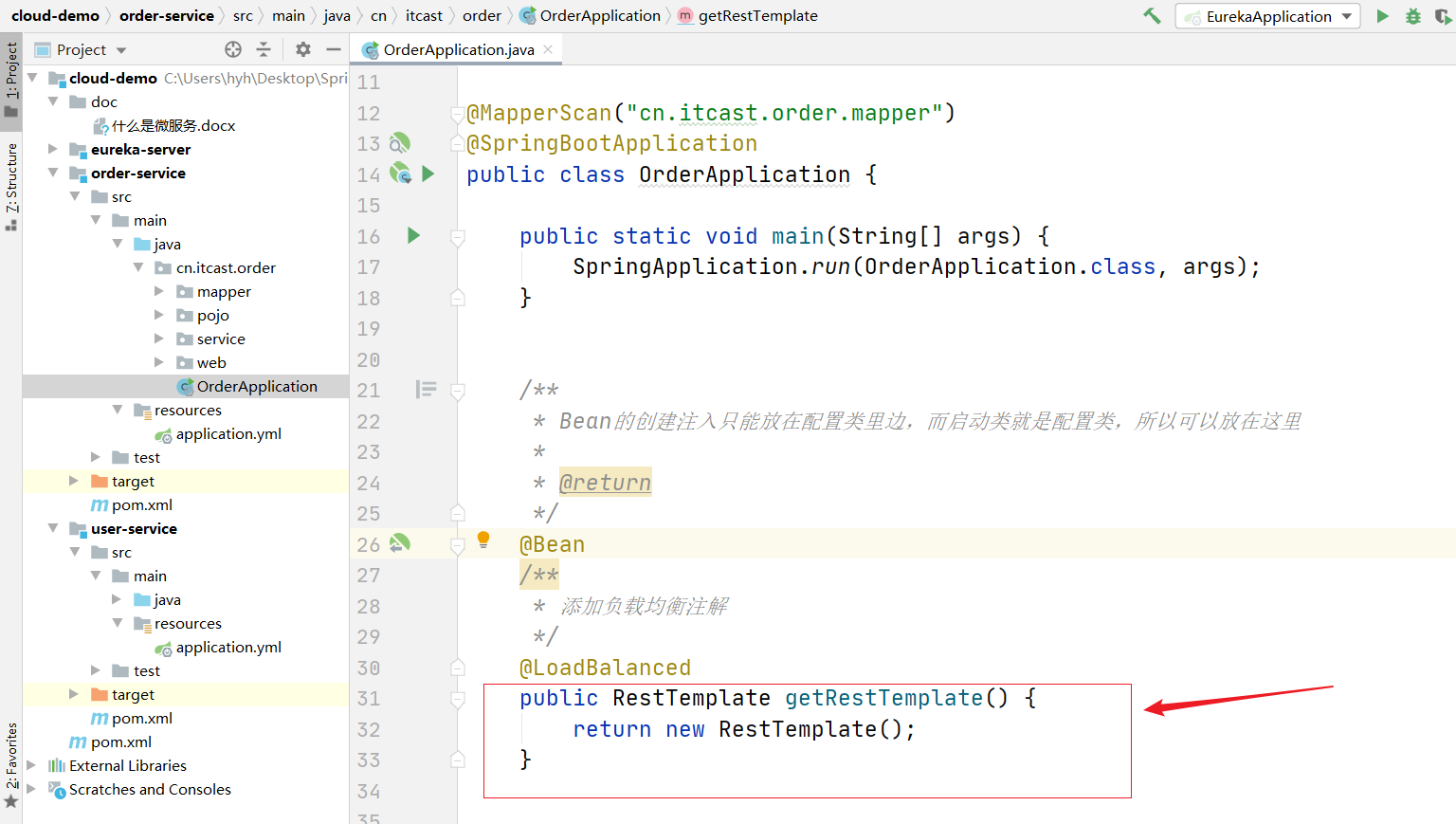
传统方式：使用httpclient

我们之前没有学过从一个服务到另一个服务的远程调用，但是我们以用从浏览器像服务发请求，并获得服务返回的响应数据。既然浏览器能发请求到服务查询数据，服务一样也可以！

即如何在JAVA代码中发送http请求！如果能发HTTP请求，就能做远程调用！

Spring提供了一个工具：RestTemplate，这个工具就是用来做HTTP请求的。

可以通过@Bean的方式把RestTemplate注册为Spring的对象，就来就能在项目中的任何地方注入RestTemplate这个对象！



基于RestTemplate发送Http请求实现远程调用，这种请求是与语言和技术无关的，只需要知道对方的ip、端口、接口路径、请求参数即可！



## 微服务的两个概念：服务提供者和服务消费者！

服务提供者：一次业务中，被其它微服务调用的服务，即提供接口给其他微服务；

服务消费者，一次业务中，调用其他微服务的服务。即调用其他微服务提供的接口。

## Eureka注册中心

### 问题引入：基于RestTemplate实现服务之间远程调用存在的问题

上面基于RestTemplate实现服务之间远程调用的问题：



如上图，Demo中，我们是从order服务发送HTTP请求到user服务，在代码中，我们是将user服务的ip和端口硬编码到代码当中的，这样的写法存在很大的问题，在实际开发中，我们会有开发环境、生产环境、测试环境等，每次环境的变更可能服务的地址因为会发生变化，那如果采用硬编码的方式，难道说每一环境变更你都要去修改代码编译打包吗？而且为了应对将来更多的并发，user服务可能部署多实例形成一个集群，每个user实例的ip、端口是不同的，这个时候order里的硬编码应该怎么写呢？所以一定不能采用硬编码！

所以：

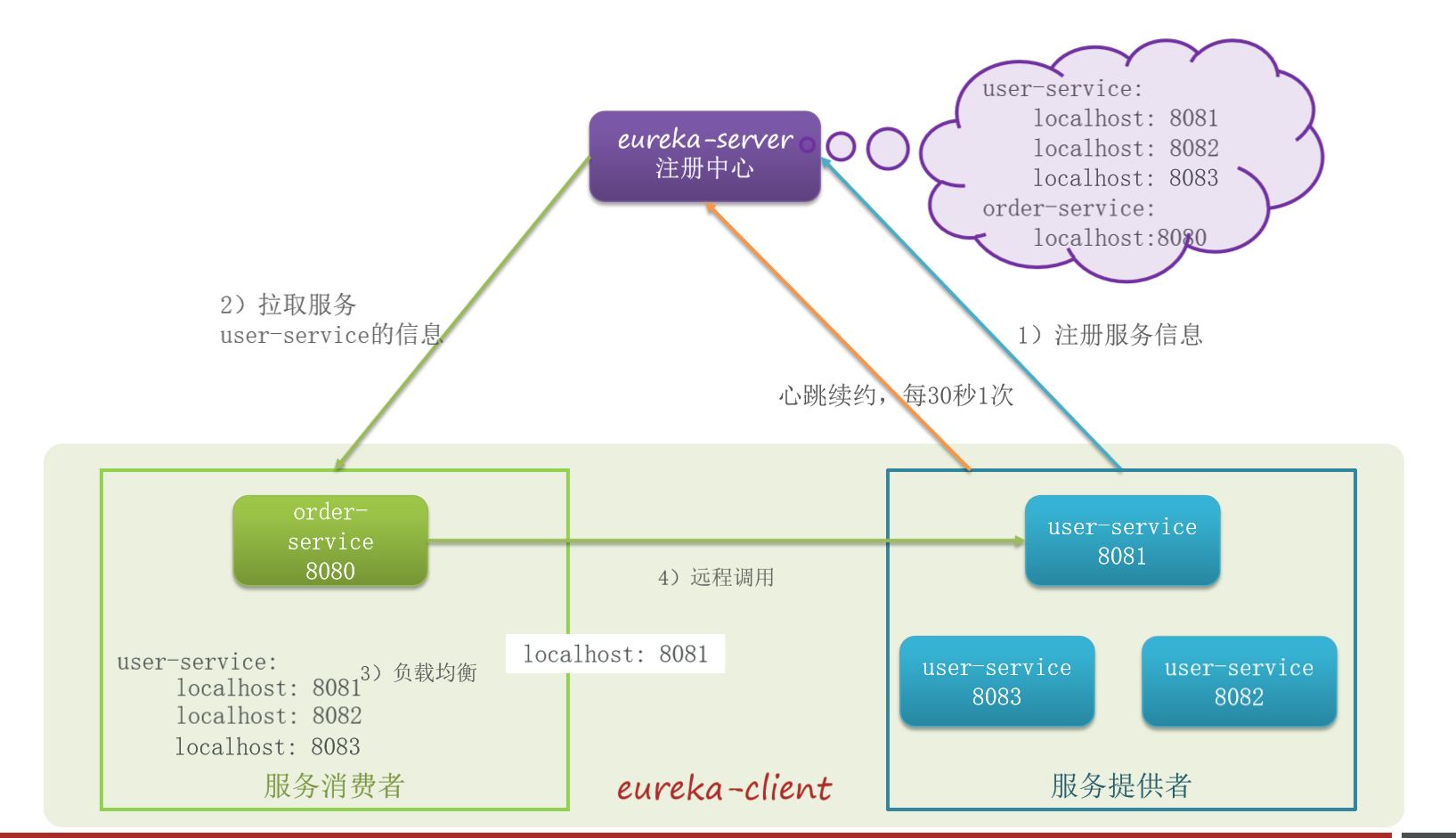
#### 问题一：服务消费者应该如何获取服务提供者的地址信息？

#### 问题二：如果由多个服务提供者，消费者应该如何选择？

#### 问题三：消费者如何得知每个服务提供者的健康状态？

以上的问题这么多，Eureka就可以解决！

### Eureka基本原理



如上图所示，Eureka的结构中，分此两个概念，eureka-server注册中心，也就是服务端，注册中心的作用是记录和管理微服务；而第二个角色就是eureka-client客户端，也就是被管理的微服务群。

Eureka-client在启动的时候，会把自己的信息注册给eureka-serve，注意了，是每一个服务提供时都会做这种事，只要时eureka-client，而eureka-server就会把eureka-client服务的信息记录下来（服务名称、IP、端口、、、、、），那么一个微服务项目中的所有的服务信息都在eureka-server中了，当有一个服务需要消费，比如order服务，那么order服务不需要自己记录服务提供者的信息，只需要去eureka注册中心找对应消费提供者user的信息，找到就把user服务列表信息给order服务，于是问题一这个问题就解决了！

现在，order服务从eureka注册中心拿到了user服务列表信息，要挑一个服务去发送请求，这个时候就要用到负载均衡的知识，这个时候，怎么保证挑到的服务会不会时挂了的？是不会的，因为我们的服务每隔30秒，都会向eureka发一个心跳，来向eureka确认自己的状态，如果有一天，eureka没有心跳发过来了，这时候eureka就会把该服务从服务列表中剔除。那么order再次去拉取user服务信息时，就拉取不到这个挂掉的服务，这个时候，就相当于服务消费者感知到了服务提供者的健康状态！那么问题二、问题三也就解决了！

#### 解决问题一：服务消费者应该如何获取服务提供者的地址信息？

1：服务提供者启动时向eureka注册自己的信息；

2：eureka保存这些信息；

3：消费者根据服务名称向eureka拉取提供者信息

#### 解决问题二：如果由多个服务提供者，消费者应该如何选择？

1：服务消费者利用负载均衡算法，从服务列表中挑选一个

#### 解决问题三：消费者如何得知每个服务提供者的健康状态？

1：服务提供者会每隔30秒向EurekaServer发送心跳请求，报告健康状态；

2：eureka会更新记录服务列表信息，心跳不正常就会被剔除；

3：消费者就可以拉区到最新的信息

#### 小总结：

在Eureka架构中，微服务角色有两类：

1：EurekaServer：服务端。注册中心，作用是记录服务信息，心跳监控；

2：EurekaClient：

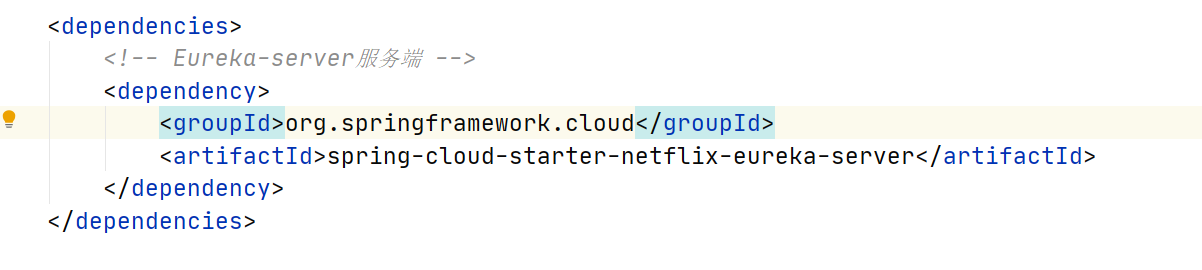
Provider：服务提供者，例如案例中的user-service，注册自己的信息到EurekaServer；每隔30秒向EurekaServer发送心跳。

Consumer：服务消费者。例如案例中的order-service，根据服务名称从eureka中拉取服务列表；基于服务列表做负载均衡，选中一个微服务后发起远程调用。

### Eureka入门使用

#### 搭建Eureka-Server

1：引依赖



2：加注解

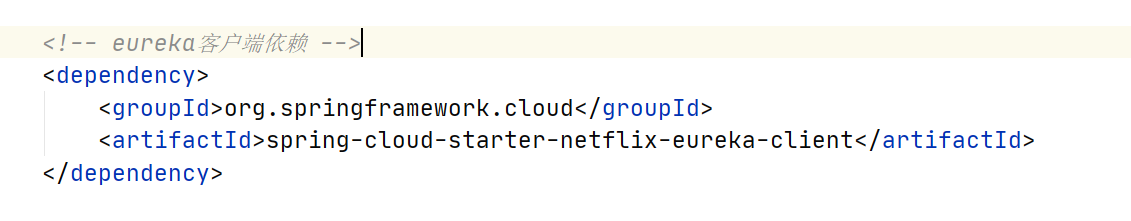


3：做配置



#### 服务注册：Eureka-Client客户端

1：引依赖



2：加配置



#### Eureka服务发现和拉取

服务拉取基于服务名称获取服务列表，然后再对服务列表做负载均衡！

1：修改OrderService中的代码，修改访问的url路径，用服务名代替IP、端口：



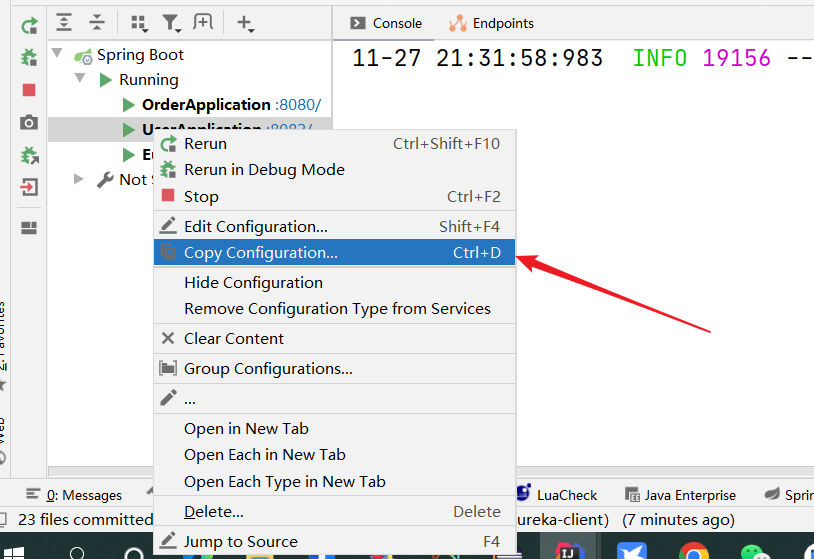
2：在order-service项目的ReseTemplate添加负载均衡注解！



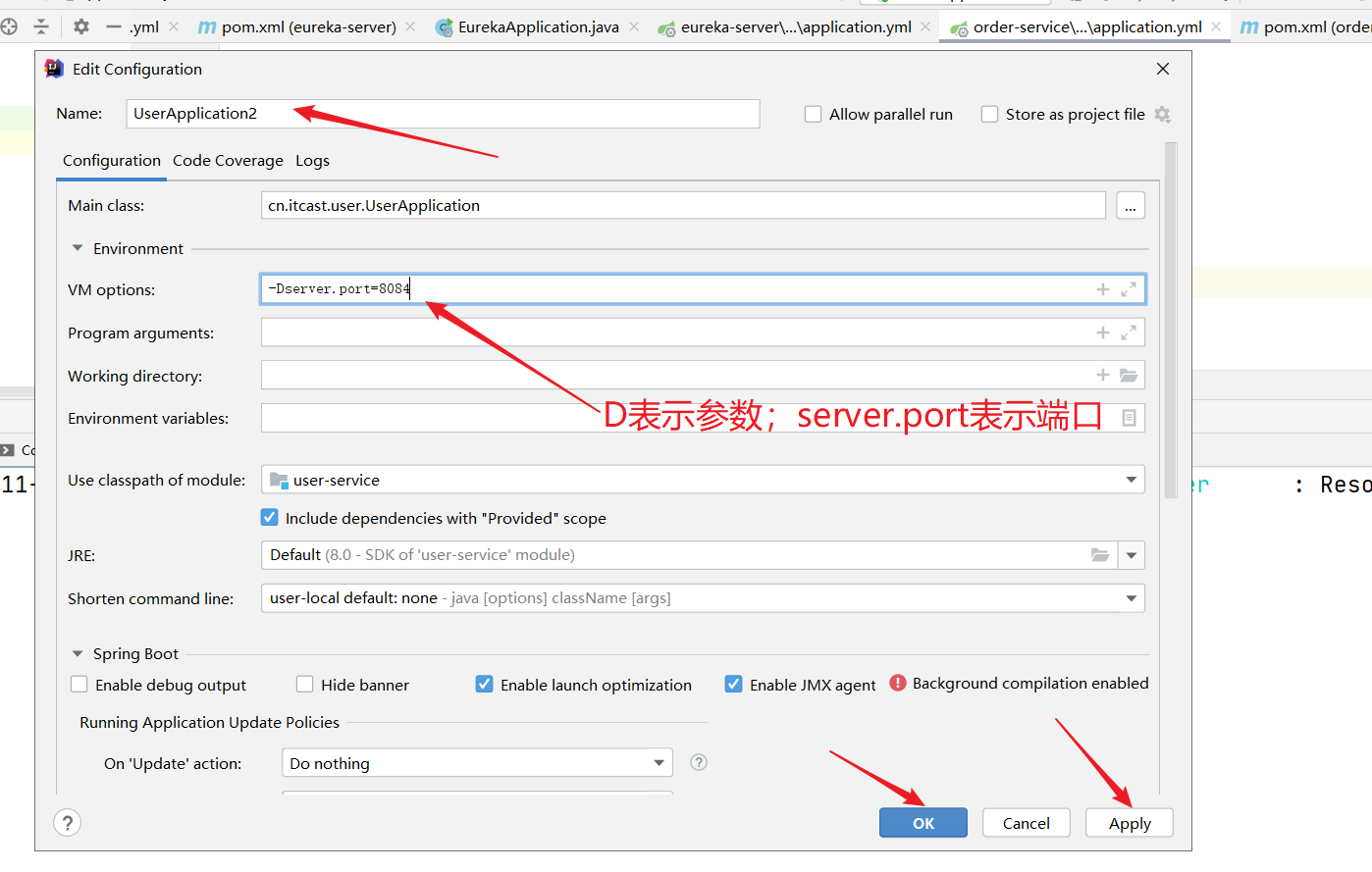
做完之后，可以发起请求，可以看到确实实现了负载均衡（轮询方式）

#### 小技巧：如果想给一个服务启动多个实例，可以按照以下步骤：

第一步：



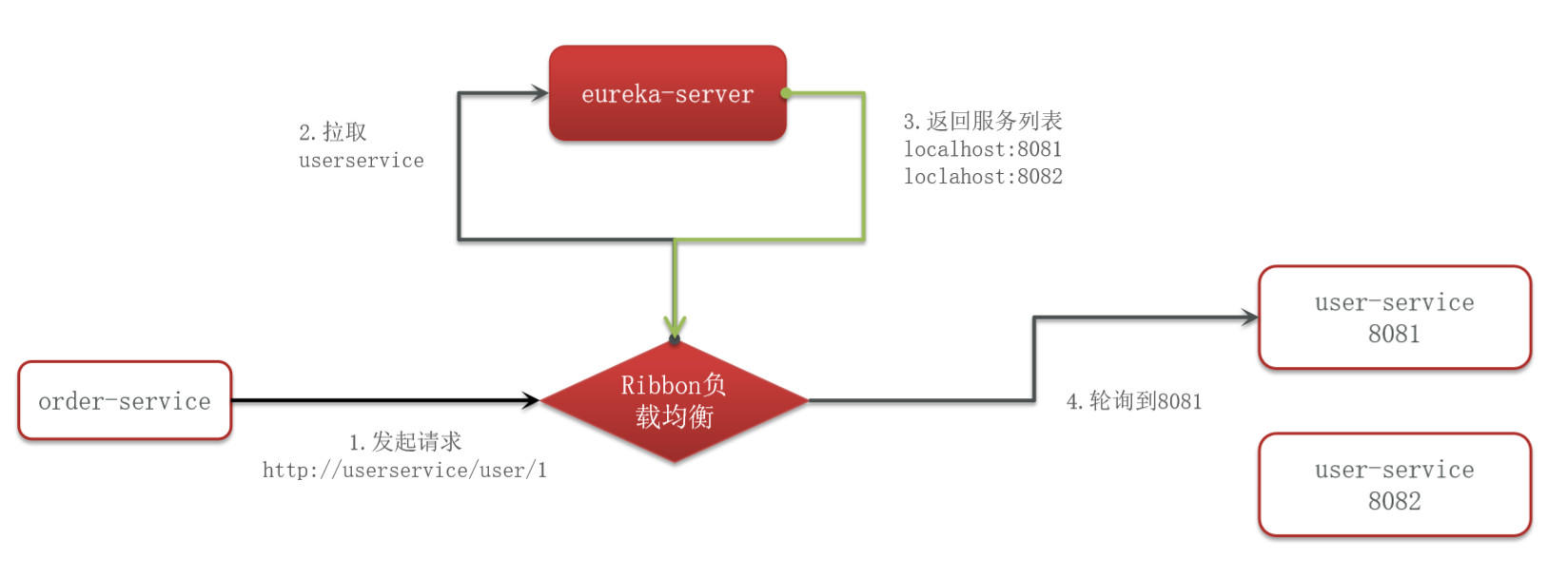
第二步：



### Eureka实现负载均衡原理：Ribbon

以上用Eureka实现了服务的拉取和请求负载均衡，但是为什么只要换成了服务名称和加了@LoadBalance注解，一切就自动发生了，那是什么时候完成了服务拉取，什么时候做到负载均衡，负载均衡的原理是什么，负载均衡的策略是什么，一切还都不知道，下面来探讨一下。

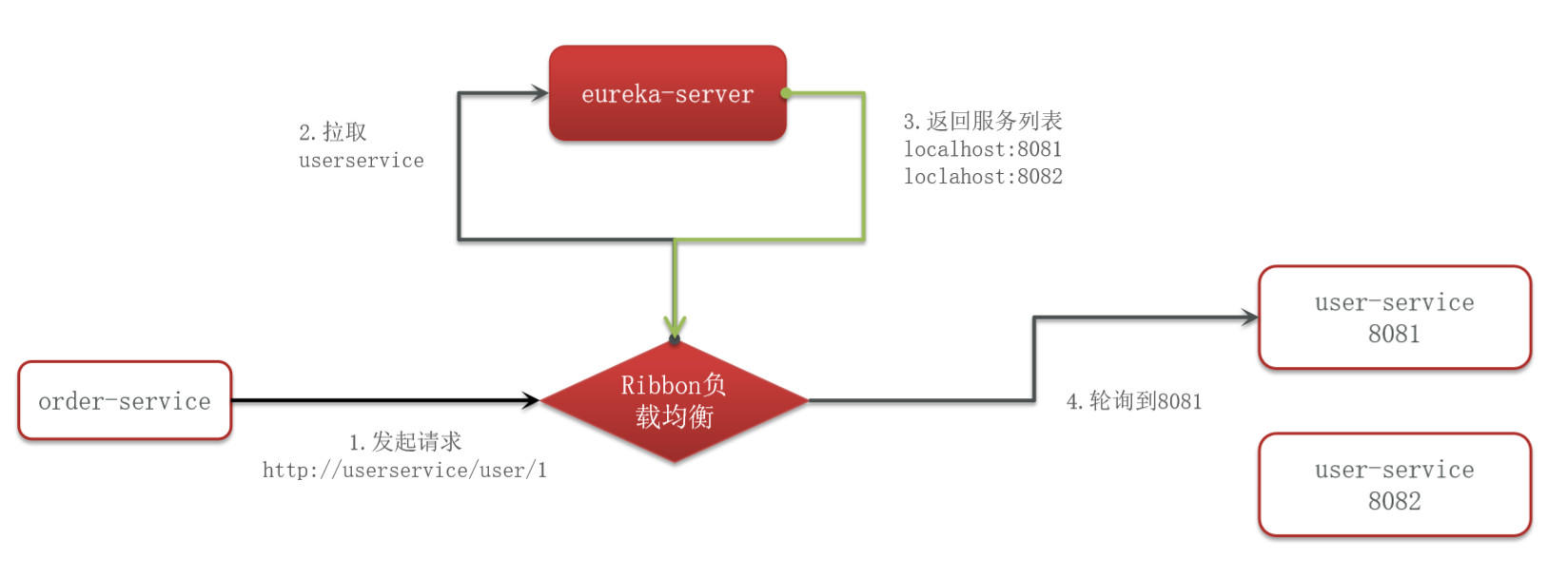
首先要明确，eureka的负载均衡是由Ribbon组件实现的！



#### Ribbon：负载均衡器

当服务消费者使用服务提供者的服务名去发起请求，调用服务提供者提供的接口时，并不是一个在浏览器真实可用的地址，不信你可以复制这个地址到浏览器试一下，因为这不是一个真正的域名和ip或者端口，因此，当服务消费者发起这个请求的时候，其实是无法到达服务提供者的，因此请求中间一定有人把这个请求拦下来做一些处理，即找到服务消费者的真实ip、端口，谁去做这些动作的呢，就是Ribbon！

Ribbon是eureka内置的ribbon。



#### Ribbon拦截处理的过程

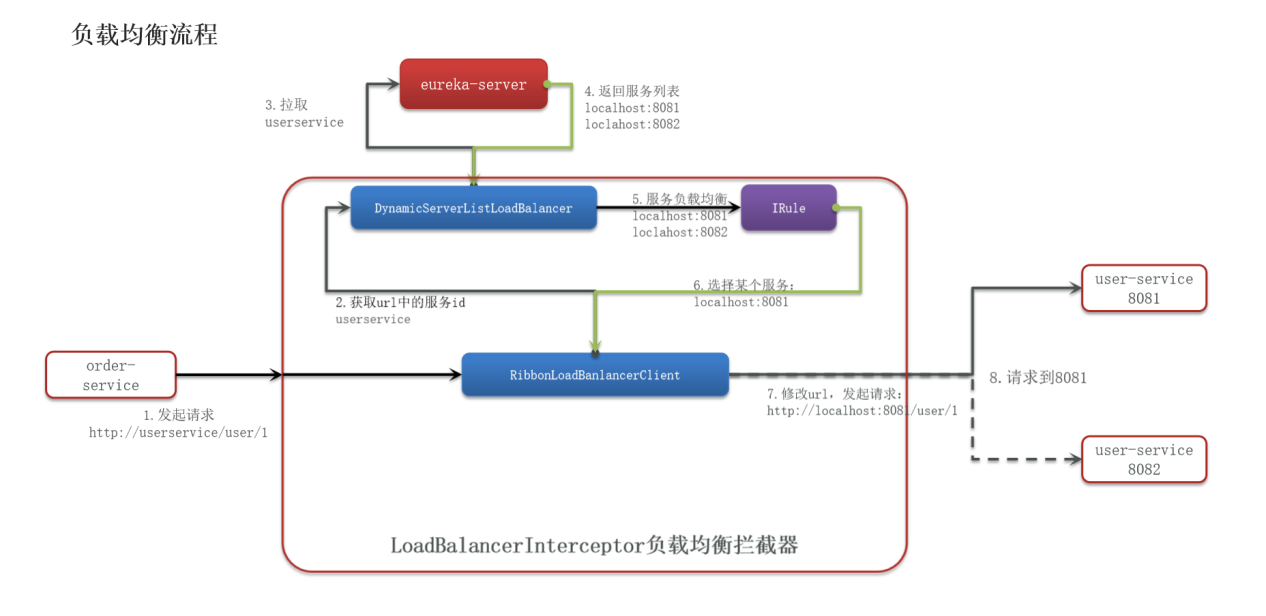
1：ribbon中的LoadBalancerInterceptor负载均衡拦截器会拦截请求；

2：拦截器把请求中交给RibbonLoadBalancerClient负载均衡客户端；

3：负载均衡客户端获取请求中的服务名称，并把服务名称交给DynamicServerListLoadBalancer；

4：DynamicServerListLoadBalancer则根据服务命去注册中心拉取对应的服务列表；

5：获取到的服务列表会由IRule按照负载均衡策略，选择出服务实例；6：最终向LoadBalancerInterceptor返回选择到的服务实例，得到真实的请求地址去发起请求，进行调用！



#### 配置Ribbon负载均衡策略：

Ribbon的负载均衡规则是一个叫IRule的接口来定义的，每一个子接口都是一种负载均衡规则!



要调整Ribbon的负载均衡，可以通过定义IRule实现，就可以修改负载均衡规则，有两种方式：

##### 1：代码方式：

定义一个IRule，此种方式是作用于全局的，也就是说，一旦用代码方式配置了这种负载均衡策略，在服务里不管调用了那个微服务，不管是调用user服务，还是商品服务，都是随机！



##### 2：配置方式：

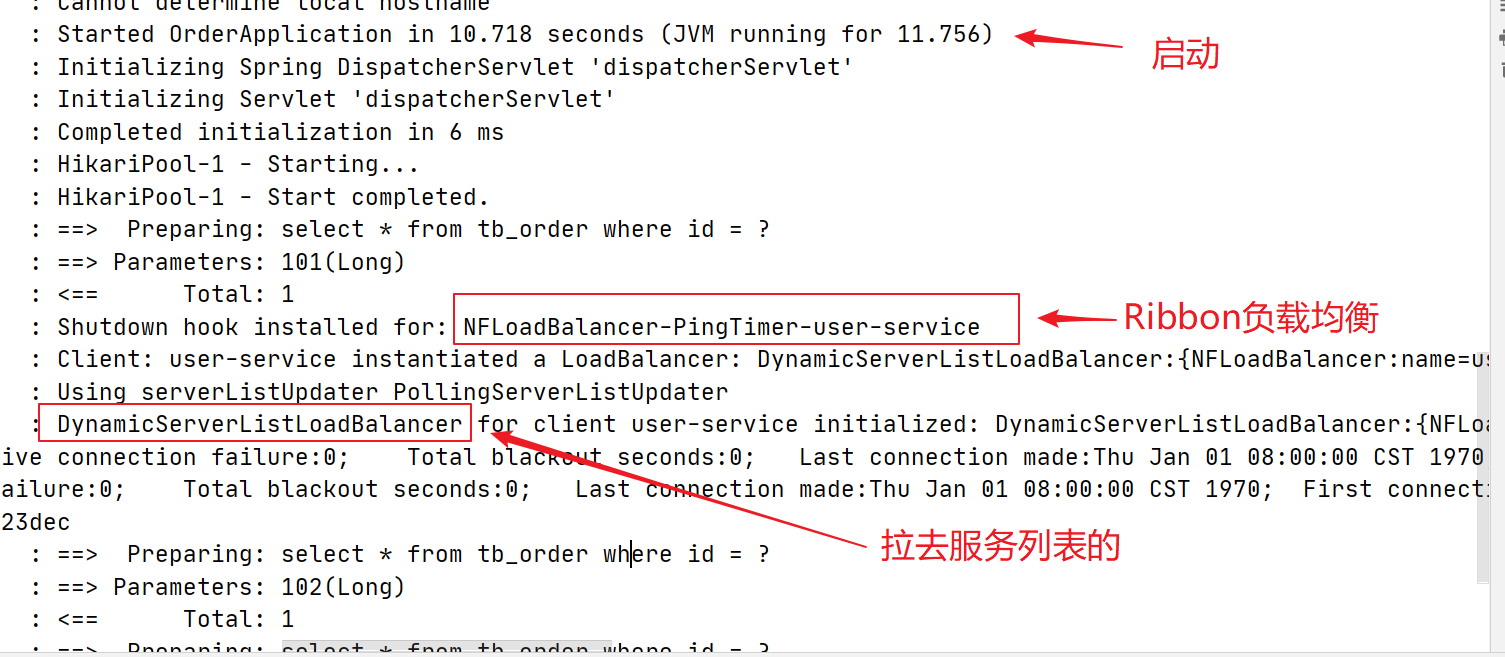
在application.yml中添加新的配置也可以修改规则，这种配置方案作用范围不是全局的，因为配置时是要先指定服务名称，再去指定负载均衡策略，所以这种方式是针对某个微服务而言的！



#### Ribbon的加载策略：

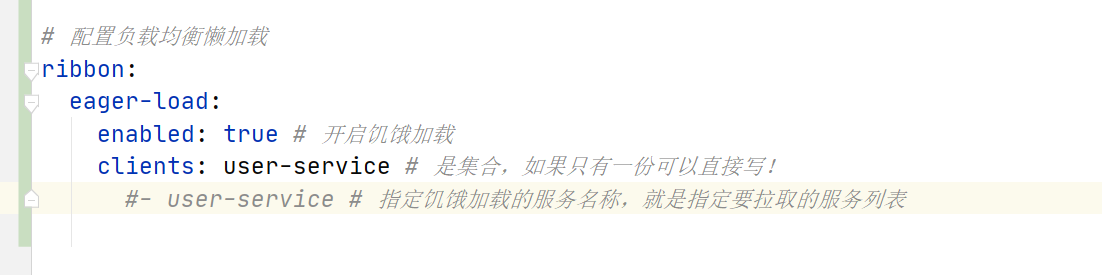
##### 懒加载：默认

Ribbon默认采用的加载方式是懒加载，即第一次访问的时候才会去创建负载均衡客户端LoadBalanceClient，请求时间会很长，因为创建需要时间，服务拉取也需要时间（当然以后就不需要这么长时间了，因为服务列表会缓存在内存）！如下如图：



##### 饥饿加载：可以通过配置文件来配置

而饥饿加载则会在项目启动时创建，降低第一次访问的耗时，可以通过下面配置开启饥饿加载：



效果如下：



#### Ribbon总结：

##### Ribbon负载均衡规则：

1：规则接口时IRule；

2：默认实现时ZoneAvoidanceRule，根据zone选择服务列表，然后轮询；

##### 负载均衡自定义方式：

1：代码方式：配置灵活，但修改时需要重新打包发布；

2：配置方式：直观、方便，无需重新打包发布，但是无法做全局配置

##### 饥饿加载步骤：

1：开启饥饿加载

2：指定姐家在的微服务名称