主讲老师： Fox

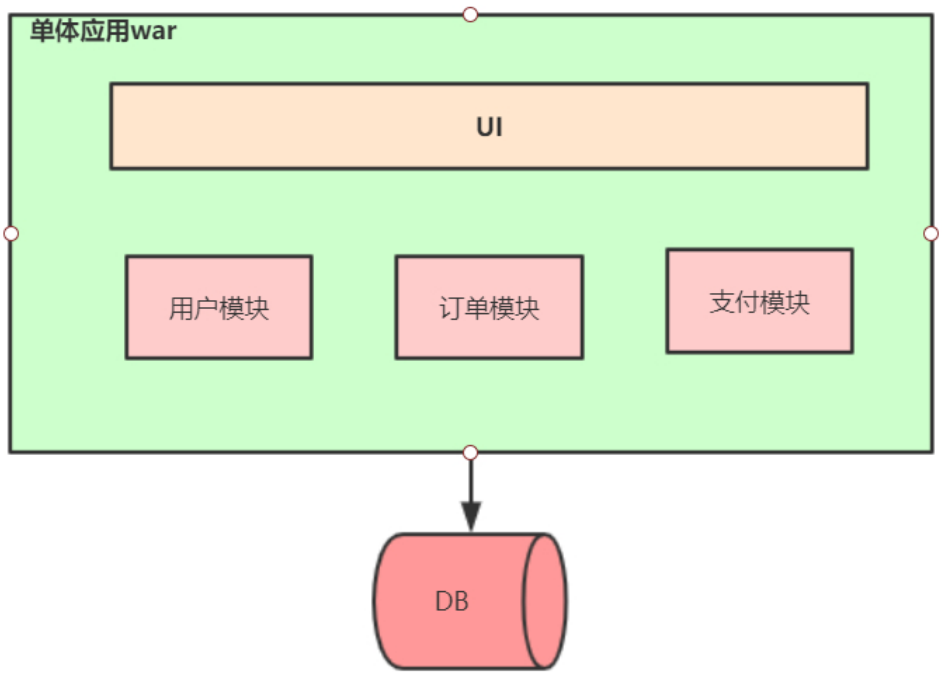
> 有道笔记：https://note.youdao.com/s/6HX48AYR

1. 系统架构的演变

俗话说，没有最好的架构，只有最合适的架构。微服务架构也是随着信息产业的发展而出现的最有普遍适用性的一套架构模式。通常来说，我们认为架构发展历史经历了这样一个过程：单体架构——> 垂直架构 ——> SOA 面向服务架构 ——> 微服务架构。

1.1 单体架构

互联网早期，一般的网站应用流量较小，只需一个应用，将所有功能代码都部署在一起就可以，这样可以减少开发、部署和维护的成本。比如说一个电商系统，里面会包含很多用户管理，商品管理，订单管理，物流管理等等很多模块，我们会把它们做成一个web项目，然后部署到一台tomcat服务器上。



很多传统互联网公司或者创业型公司早期基本都会采用这样的架构，因为这样的架构足够简单，能够快速开发和上线。而且对于项目初期用户量不大的情况，这样的架构足以支撑业务的正常运行。

优点：

- 项目架构简单，小型项目的话， 开发成本低
- 项目部署在一个节点上， 维护方便

缺点：

- 全部功能集成在一个工程中，对于大型项目来讲不易开发和维护
- 项目模块之间紧密耦合，单点容错率低
- 无法针对不同模块进行针对性优化和水平扩展

1.2 垂直架构

随着用户量越来越大，网站的访问量不断增大，导致后端服务器的负载越来越高。 用户量大了，产品需要满足不同用户的需求来留住用户，使得业务场景越来越多并且越来越复杂。
我们可以从两个方面进行优化：

- 通过横向增加服务器，把单台机器变成多台机器的集群。
- 按照业务的垂直领域进行拆分，减少业务的耦合度，以及降低单个war包带来的伸缩性困难问题。

优点：

- 系统拆分实现了流量分担，可以针对不同模块进行优化
- 方便水平扩展，负载均衡，容错率提高
- 系统间相互独立，互不影响，新的业务迭代时更加高效

缺点：

- 服务之间相互调用，如果某个服务的端口或者IP地址发生改变。调用的系统得手动变化
- 服务之间调用方式不统一，基于httpclient,webservice，接口协议不统一
- 搭建集群之后，实现负载均衡比较复杂。比如：内网负载，在迁移得时候会影响调用方的路由，导致线上故障
- 服务监控不到位

1.3 SOA架构

为了让大家更好地理解SOA，我们来看两个场景：
• 假设一个用户执行下单操作，系统的处理逻辑是先去库存子系统检查商品的库存，只有在库存足够的情况下才会提交订单，那么这个检查库存的逻辑是放在订单子系统中还是库存子系统中呢？在整个系统中，一定会存在非常多类似的共享业务的场景，这些业务场景的逻辑肯定会被重复创建，从而产生非常多冗余的业务代码，这些冗余代码的维护成本会随着时间的推移越来越高，能不能够把这些共享业务逻辑抽离出来形成可重用的服务呢？
• 在一个集团公司下有很多子公司，每个子公司都有自己的业务模式和信息沉淀，各个子公司之间不进行交互和共享。这个时候每个子公司虽然能够创造一定的价值，但是由于各个子公司之间信息不是互联互通的，彼此之间形成了信息孤岛，使得价值无法最大化。
基于这些问题，就引入了SOA（Service-Oriented Architecture），也就是面向服务的架构。在SOA中，会采用ESB（企业服务总线）来作为系统和服务之间的通信桥梁，ESB本身还提供服务地址的管理、不同系统之间的协议转化和数据格式转化等。调用端不需要关心目标服务的位置，从而使得服务之间的交互是动态的，这样做的好处是实现了服务的调用者和服务的提供者之间的高度解耦。
总的来说，SOA主要解决的问题是：

- 信息孤岛
- 共享业务的重用

优点:

- 使用治理中心（ESB）解决了服务间调用关系的自动调节

缺点:

- 服务间会有依赖关系，一旦某个环节出错会影响较大(服务雪崩)
- 服务关系复杂，运维、测试部署困难

1.4 微服务架构

微服务架构在某种程度上是面向服务的架构SOA继续发展的下一步，它更加强调服务的"彻底拆分"。面向服务（SOA）和微服务本质上都是服务化思想的一种体现。如果SOA是面向服务开发思想的雏形，那么微服务就是针对可重用业务服务的更进一步优化，我们可以把SOA看成微服务的超集，也就是多个微服务可以组成一个SOA服务。伴随着服务粒度的细化，会导致原本10个服务可能拆分成了100个微服务，一旦服务规模扩大就意味着服务的构建、发布、运维的复杂度也会成倍增加，所以实施微服务的前提是软件交付链路及基础设施的成熟化。

由于SOA和微服务两者的关注点不一样，造成了这两者有非常大的区别：

- SOA关注的是服务的重用性及解决信息孤岛问题。
- 微服务关注的是解耦，虽然解耦和可重用性从特定的角度来看是一样的，但本质上是有区别的，解耦是降低业务之间的耦合度，而重用性关注的是服务的复用。
- 微服务会更多地关注在DevOps的持续交付上，因为服务粒度细化之后使得开发运维变得更加重要，因此微服务与容器化技术的结合更加紧密。

微服务架构就是将每个具体的业务服务构成可独立运行的微服务，每个微服务只关注某个特定的功能，服务之间采用轻量级通信机制REST API进行通信。

> 英文:https://martinfowler.com/articles/microservices.html
> https://microservices.io/patterns/microservices.html
> 中文:http://blog.cuicc.com/blog/2015/07/22/microservices

优点：

- 复杂度可控：通过对共享业务服务更细粒度的拆分，一个服务只需要关注一个特定的业务领域，并通过定义良好的接口清晰表述服务边界。由于体积小、复杂度低，开发、维护会更加简单。
- 技术选型更灵活：每个微服务都由不同的团队来维护，所以可以结合业务特性自由选择技术栈。
- 可扩展性更强：可以根据每个微服务的性能要求和业务特点来对服务进行灵活扩展，比如通过增加单个服务的集群规模，提升部署了该服务的节点的硬件配置。
- 独立部署：由于每个微服务都是一个独立运行的进程，所以可以实现独立部署。当某个微服务发生变更时不需要重新编译部署整个应用，并且单个微服务的代码量比较小，使得发布更加高效。
- 容错性：在微服务架构中，如果某一个服务发生故障，我们可以使故障隔离在单个服务中。其他服务可以通过重试、降级等机制来实现应用层面的容错。

缺点：

微服务架构不是银弹，它并不能解决所有的架构问题。在拥抱微服务架构的过程中，我们经常会遇到数据库的拆分、API交互、大量的微服务开发和维护、运维等问题。即便成功实现了微服务的主体，也还是会面临下面这样一些挑战。

- 故障排查：一次请求可能会经历多个不同的微服务的多次交互，交互的链路可能会比较长，每个微服务会产生自己的日志，在这种情况下如果出现一个故障，开发人员定位问题的根源会比较困难。
- 服务监控：在一个单体架构中很容易实现服务的监控，因为所有的功能都在一个服务中。在微服务架构中，服务监控开销会非常大，可以想象一下，在几百个微服务组成的架构中，我们不仅要对整个链路进行监控，还需要对每一个微服务都实现一套类似单体架构的监控。
- 分布式架构的复杂性：微服务本身构建的是一个分布式系统，分布式系统涉及服务之间的远程通信，而网络通信中网络的延迟和网络故障是无法避免的，从而增加了应用程序的复杂度。
- 服务依赖：微服务数量增加之后，各个服务之间会存在更多的依赖关系，使得系统整体更为复杂。假设你在完成一个案例，需要修改服务A、B、C，而A依赖B，B依赖C。在单体式应用中，你只需要改变相关模块，整合变化，再部署就好了。对比之下，微服务架构模式就需要考虑相关改变对不同服务的影响。比如，你需要更新服务C，然后是B，最后才是A，幸运的是，许多改变一般只影响一个服务，需要协调多服务的改变很少。
- 运维成本：在微服务中，需要保证几百个微服务的正常运行，对于运维的挑战是巨大的。比如单个服务流量激增时如何快速扩容、服务拆分之后导致故障点增多如何处理、如何快速部署和统一管理众多的服务等。

2. 如何实现微服务架构

2.1 微服务架构下的技术挑战

微服务架构主要的目的是实现业务服务的解耦。随着公司业务的高速发展，微服务组件会越来越多，导致服务与服务之间的调用关系越来越复杂。同时，服务与服务之间的远程通信也会因为网络通信问题的存在变得更加复杂，比如需要考虑重试、容错、降级等情况。那么这个时候就需要进行服务治理，将服务之间的依赖转化为服务对服务中心的依赖。除此之外，还需要考虑：

- 服务的注册与发现
- 分布式配置中心
- 服务路由
- 负载均衡
- 熔断限流
- 分布式链路监控

这些都需要对应的技术来实现，我们是自己研发还是选择市场上比较成熟的技术拿来就用呢？如果市场上有多种相同的解决方案，应该如何做好技术选型？

2.2 微服务技术栈选型

业内比较主流的微服务解决方案进行分析，主要包括：

- Spring Cloud Netflix
- Spring Cloud Alibaba

什么是Spring Cloud全家桶

Spring Cloud提供了一些可以让开发者快速构建微服务应用的工具，比如配置管理、服务发现、熔断、智能路由等，这些服务可以在任何分布式环境下很好地工作。Spring Cloud主要致力于解决如下问题：

- Distributed configuration，分布式配置
- Service registration and discovery，服务注册与发现
- Routing，服务路由
- Service-to-service calls，服务调用
- Load balancing，负载均衡
- Circuit Breakers，断路器
- Distributed messaging，分布式消息

需要注意的是，Spring Cloud并不是Spring团队全新研发的框架，它只是把一些比较优秀的解决微服务架构中常见问题的开源框架基于Spring Cloud规范进行了整合，通过Spring Boot这个框架进行再次封装后屏蔽掉了复杂的配置，给开发者提供良好的开箱即用的微服务开发体验。不难看出，Spring Cloud其实就是一套规范，而Spring Cloud Netflix、Spring Cloud Alibaba才是Spring Cloud规范的实现。

	Spring Cloud Netflix	Spring Cloud 官方	Spring Cloud Zookeeper	Spring Cloud Consul	Spring Cloud Kubernetes	Spring Cloud Alibaba
分布式配置	Archaius	Spring Cloud Config	Zookeeper	Consul	ConfigMap	Nacos
服务注册/发现	Eureka	–	Zookeeper	Consul	Api Server	Nacos
服务熔断	Hystrix	–	–	–	–	Sentinel
服务调用	Feign	OpenFeign RestTemplate	–	–	–	Dubbo RPC
服务路由	Zuul	Spring Cloud Gateway	–	–	–	Dubbo PROXY
分布式消息	–	SCS RabbitMQ	–	–	–	SCS RocketMQ
负载均衡	Ribbon	–	–	–	–	Dubbo LB
分布式事务	–	–	–	–	–	Seata

Alibaba的开源组件在服务治理上和处理高并发的能力上有天然的优势，毕竟这些组件都经历过数次双11的考验，也在各大互联网公司大规模应用过。所以，相比Spring Cloud Netflix来说，Spring Cloud Alibaba在服务治理这块的能力更适合于国内的技术场景，同时，Spring Cloud Alibaba在功能上不仅完全覆盖了Spring Cloud Netflix原生特性，而且还提供了更加稳定和成熟的实现

Spring Cloud Alibaba版本选择

> 版本说明：https://github.com/alibaba/spring-cloud-alibaba/wiki/%E7%89%88%E6%9C%AC%E8%AF%B4%E6%98%8E

本期我们选择版本：Spring Cloud Alibaba 2022.0.0.0

Spring Cloud Alibaba Version	Spring Cloud Version	Spring Boot Version
2022.0.0.0*	Spring Cloud 2022.0.0	3.0.2
2022.0.0.0-RC2	Spring Cloud 2022.0.0	3.0.2
2022.0.0.0-RC1	Spring Cloud 2022.0.0	3.0.0

组件版本关系

每个 Spring Cloud Alibaba 版本及其自身所适配的各组件对应版本如下表所示（注意，Spring Cloud Dubbo 从 2021.0.1.0 起已被移除出主干，不再随主干演进）：

Spring Cloud Alibaba Version	Sentinel Version	Nacos Version	RocketMQ Version	Dubbo Version	Seata Version
2022.0.0.0	1.8.6	2.2.1	4.9.4	~	1.7.0
2022.0.0.0-RC2	1.8.6	2.2.1	4.9.4	~	1.7.0-native-rc2
2021.0.5.0	1.8.6	2.2.0	4.9.4	~	1.6.1
2.2.10-RC1	1.8.6	2.2.0	4.9.4	~	1.6.1
2022.0.0.0-RC1	1.8.6	2.2.1-RC	4.9.4	~	1.6.1
2.2.9.RELEASE	1.8.5	2.1.0	4.9.4	~	1.5.2

构建Maven项目的父pom

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.0.2</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <java.version>17</java.version>
    <spring-cloud-alibaba.version>2022.0.0.0</spring-cloud-alibaba.version>
    <spring-cloud.version>2022.0.0</spring-cloud.version>
</properties>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
```

```
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>${spring-cloud.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <dependency>
            <groupId>com.alibaba.cloud</groupId>
            <artifactId>spring-cloud-alibaba-dependencies</artifactId>
            <version>${spring-cloud-alibaba.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

3. Alibaba 服务注册与发现组件Nacos实战

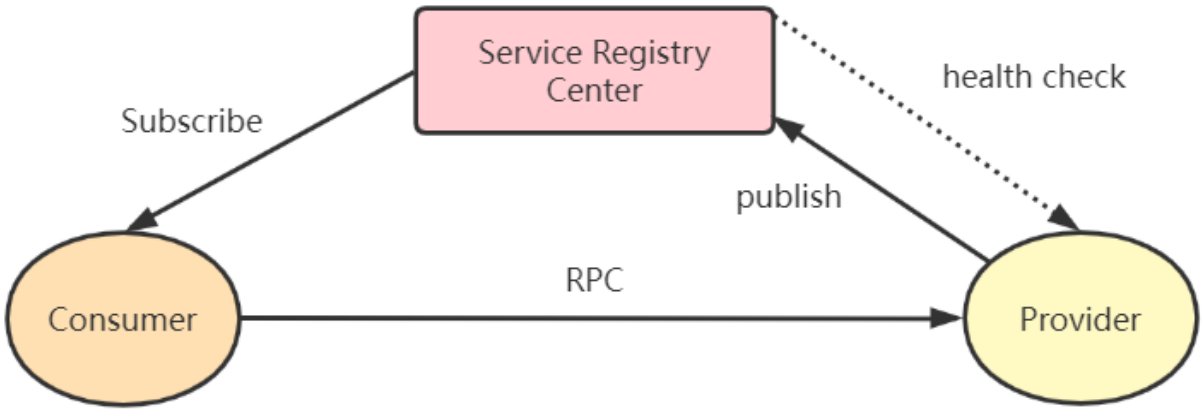
3.1 为什么需要注册中心

思考：如果服务提供者发生变动，服务调用者如何感知服务提供者的ip和端口变化?

```
//微服务之间通过RestTemplate调用，ip:port写死,如果ip或者port变化呢？
String url = "http://localhost:8020/order/findOrderByUserId/"+id;
R result = restTemplate.getForObject(url,R.class);
```

服务注册中心的作用就是服务注册与发现

- 服务注册，就是将提供某个服务的模块信息(通常是这个服务的ip和端口)注册到1个公共的组件上去。
- 服务发现，就是新注册的这个服务模块能够及时的被其他调用者发现。不管是服务新增和服务删减都能实现自动发现。



3.2 注册中心选型

	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

3.3 Nacos是什么

> 官方文档：https://nacos.io/zh-cn/docs/v2/what-is-nacos.html

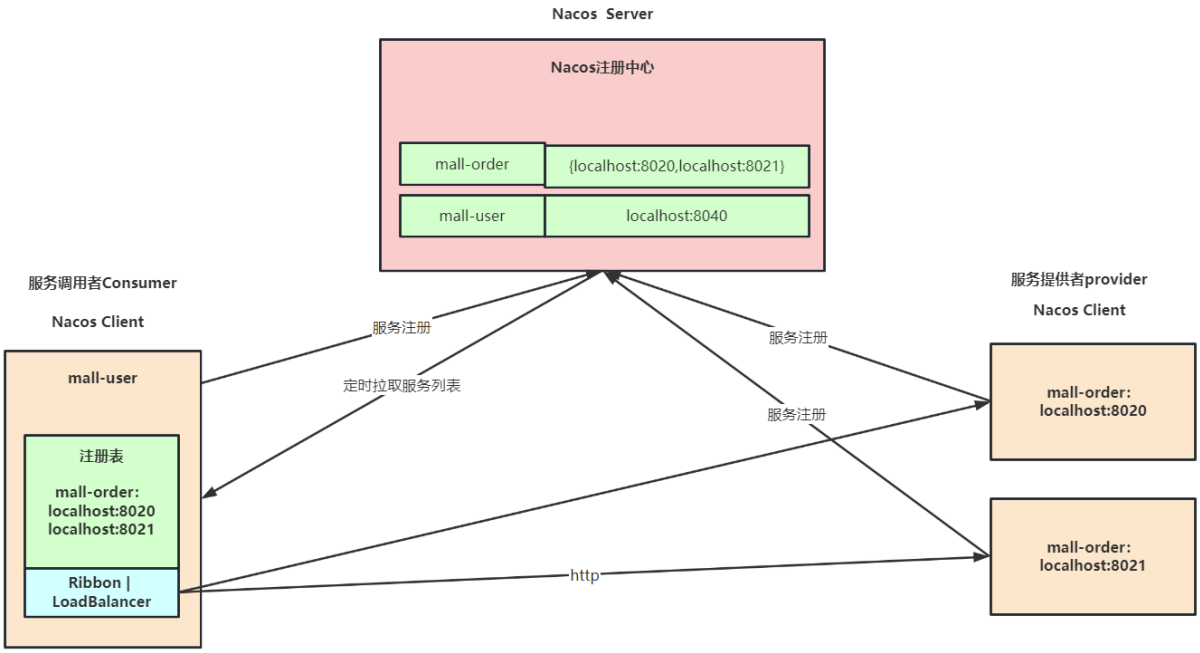
Nacos /nɑ:kəʊs/ 是 Dynamic Naming and Configuration Service的首字母简称，一个更易于构建云原生应用的动态服务发现、配置管理和服务管理平台。

Nacos 致力于帮助您发现、配置和管理微服务。Nacos 提供了一组简单易用的特性集，帮助您快速实现动态服务发现、服务配置、服务元数据及流量管理。

Nacos 优势

- **易用**：简单的数据模型，标准的 restfulAPI，易用的控制台，丰富的使用文档。
- **稳定**：99.9% 高可用，脱胎于历经阿里巴巴 10 年生产验证的内部产品，支持具有数百万服务的大规模场景，具备企业级 SLA 的开源产品。
- **实时**：数据变更毫秒级推送生效；1w 级，SLA 承诺 1w 实例上下线 1s，99.9% 推送完成；10w 级，SLA 承诺 1w 实例上下线 3s，99.9% 推送完成；100w 级别，SLA 承诺 1w 实例上下线 9s 99.9% 推送完成。
- **规模**：十万级服务/配置，百万级连接，具备强大扩展性。

3.4 Nacos 注册中心架构



相关核心概念

服务 (Service)

服务是指一个或一组软件功能（例如特定信息的检索或一组操作的执行），其目的是不同的客户端可以为不同的目的重用（例如通过跨进程的网络调用）。Nacos 支持主流的服务生态，如 Kubernetes Service、gRPC|Dubbo RPC Service 或者 Spring Cloud RESTful Service。

服务注册中心 (Service Registry)

服务注册中心，它是服务及其实例和元数据的数据库。服务实例在启动时注册到服务注册表，并在关闭时注销。服务和路由器的客户端查询服务注册表以查找服务的可用实例。服务注册中心可能会调用服务实例的健康检查 API 来验证它是否能够处理请求。

服务元数据 (Service Metadata)
服务元数据是指包括服务端点(endpoints)、服务标签、服务版本号、服务实例权重、路由规则、安全策略等描述服务的数据。

服务提供方 (Service Provider)
是指提供可复用和可调用服务的应用方。

服务消费方 (Service Consumer)
是指会发起对某个服务调用的应用方。

核心功能

服务注册：Nacos Client会通过发送REST请求的方式向Nacos Server注册自己的服务，提供自身的元数据，比如ip地址、端口等信息。Nacos Server接收到注册请求后，就会把这些元数据信息存储在一个双层的内存Map中。
服务心跳：在服务注册后，Nacos Client会维护一个定时心跳来持续通知Nacos Server，说明服务一直处于可用状态，防止被剔除。默认5s发送一次心跳。
服务同步：Nacos Server集群之间会互相同步服务实例，用来保证服务信息的一致性。
服务发现：服务消费者（Nacos Client）在调用服务提供者的服务时，会发送一个REST请求给Nacos Server，获取上面注册的服务清单，并且缓存在Nacos Client本地，同时会在Nacos Client本地开启一个定时任务定时拉取服务端最新的注册表信息更新到本地缓存
服务健康检查：Nacos Server会开启一个定时任务用来检查注册服务实例的健康情况，对于超过15s没有收到客户端心跳的实例会将它的healthy属性置为false(客户端服务发现时不会发现)，如果某个实例超过30秒没有收到心跳，直接剔除该实例(被剔除的实例如果恢复发送心跳则会重新注册)

3.5 微服务整合Nacos注册中心实战

Nacos Server环境搭建

> 官方文档：https://nacos.io/zh-cn/docs/v2/guide/admin/deployment.html

1) 下载nacos server安装包
选择安装nacos server版本： v2.2.1

```
wget https://github.com/alibaba/nacos/releases/download/2.2.1/nacos-server-2.2.1.tar.gz
```

2) 进入conf/application.properties，配置nacos.core.auth.plugin.nacos.token.secret.key密钥

```
# 默认鉴权插件用于生成用户登陆临时accessToken所使用的密钥，使用默认值有安全风险 (2.2.0.1后无默认值)
nacos.core.auth.plugin.nacos.token.secret.key=aiDLyHlCgaXB08FL5zS3W6YQZssTVNScY
```

注意：在2.2.0.1版本后，社区发布版本需要自行填充nacos.core.auth.plugin.nacos.token.secret.key的值，否则无法启动节点。

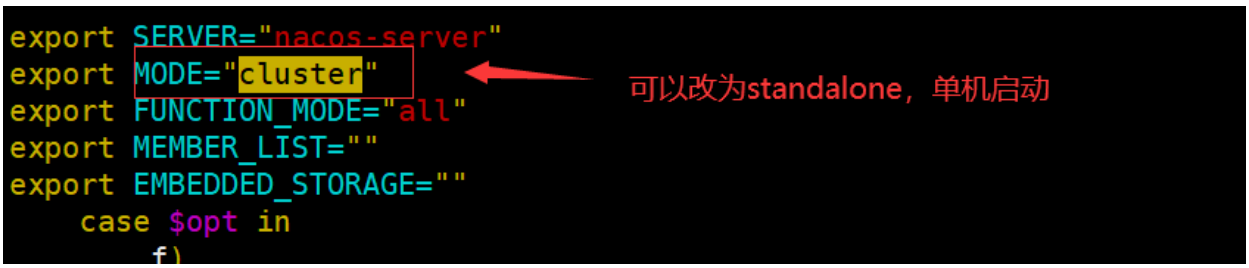
自定义密钥时，推荐将配置项设置为Base64编码的字符串，且原始密钥长度不得低于32字符。

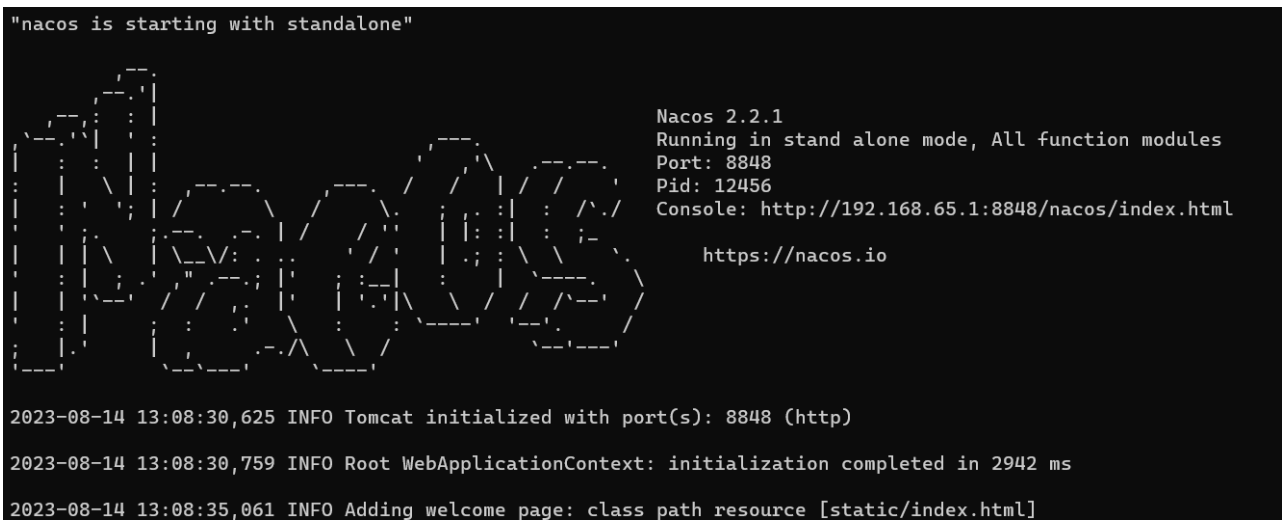
> 权限认证：https://nacos.io/zh-cn/docs/v2/guide/user/auth.html
>
> 随机字符串生成工具: http://tool.pfan.cn/random?chknumber=1&chklower=1&chkupper=1

3) 解压，进入nacos目录，单机模式启动nacos

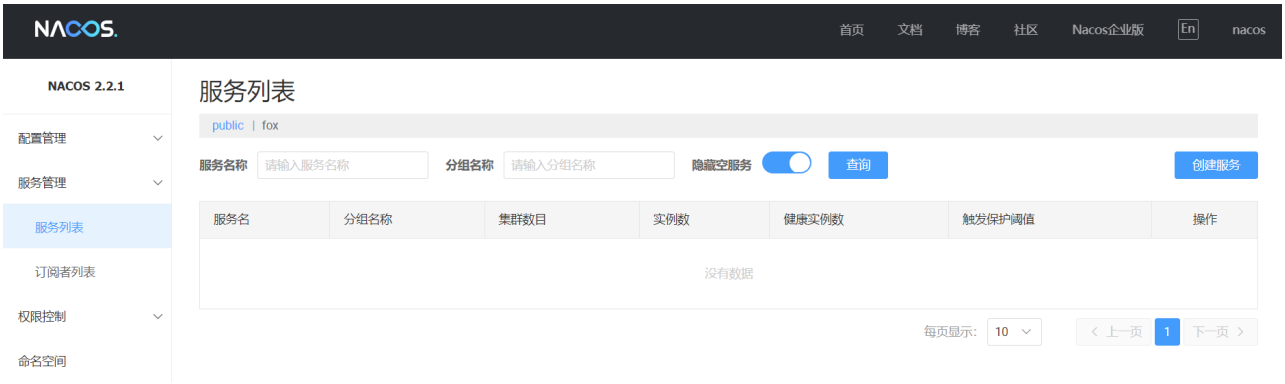
```
bin/startup.sh -m standalone
```

也可以修改默认启动方式





4） 访问nacos的管理端：http://192.168.65.1:8848/nacos ，默认的用户名密码是 nacos/nacos



微服务提供者整合Nacos

使用 Spring Cloud Alibaba Nacos Discovery，可基于 Spring Cloud 的编程模型快速接入 Nacos 服务注册功能。服务提供者可以通过 Nacos 的服务注册发现功能将其服务注册到 Nacos server 上。

以mall-order整合nacos为例

1）引入依赖

mall-order模块pom中引入nacos-client依赖

```
<!-- nacos服务注册与发现 -->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2）启动类上添加@EnableDiscoveryClient注解，此注解可以省略

3）yml配置文件中配置nacos注册中心地址

```
server:
  port: 8020

spring:
  application:
    name: mall-order  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
```

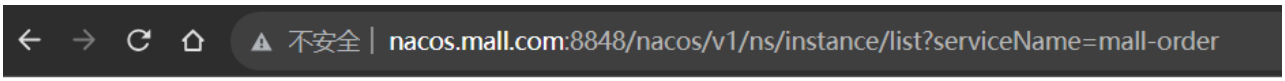
```
    discovery:
      server-addr: nacos.mall.com:8848    #注册中心地址，建议用域名替换ip
```

> 更多配置：https://github.com/alibaba/spring-cloud-alibaba/wiki/Nacos-discovery

4）启动mall-order服务，nacos管理端界面查看mall-order是否注册成功

5) 测试，通过Open API查询实例列表

http://nacos.mall.com:8848/nacos/v1/ns/instance/list?serviceName=mall-order




```
{
    name: "DEFAULT_GROUP@@mall-order",
    groupName: "DEFAULT_GROUP",
    clusters: "",
    cacheMillis: 10000,
  - hosts: [
      - {
            ip: "192.168.65.1",
            port: 8020,
            weight: 1,
            healthy: true,
            enabled: true,
            ephemeral: true,
            clusterName: "DEFAULT",
            serviceName: "DEFAULT_GROUP@@mall-order",
          - metadata: {
                preserved.register.source: "SPRING_CLOUD"
            },
            ipDeleteTimeout: 30000,
            instanceHeartBeatInterval: 5000,
            instanceHeartBeatTimeOut: 15000,
        }
    ],
    lastRefTime: 1691991370725,
    checksum: "",
    allIPs: false,
    reachProtectionThreshold: false,
    valid: true,
}
```


微服务调用者整合Nacos

服务调用者可以通过 Nacos 的服务注册发现功能从 Nacos server 上获取到它要调用的服务。

以mall-user整合nacos为例

mall-user模块pom中引入nacos-client依赖

```xml
<!-- nacos服务注册与发现 -->
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

2）启动类上添加@EnableDiscoveryClient注解，此注解可以省略

3）yml配置文件中配置nacos注册中心地址

```yaml
server:
  port: 8040

spring:
  application:
    name: mall-user  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: nacos.mall.com:8848
```

4）启动mall-user服务，nacos管理端界面查看mall-user是否注册成功

整合RestTemplate+Spring Cloud LoadBalancer实现微服务调用

Spring Cloud LoadBalancer是Spring Cloud官方自己提供的客户端负载均衡器实现，用来替代Ribbon。对于负载均衡机制，增加了ReactiveLoadBalancer接口，并提供了基于round-robin轮询和Random随机的实现。

> 官方文档：https://docs.spring.io/spring-cloud-commons/docs/4.0.4/reference/html/#spring-cloud-loadbalancer

loadbalancer常用的配置：

```yaml
spring:
```

```yaml
cloud:
  # 负载均衡配置
  loadbalancer:
    ribbon:
      #禁用ribbon
      enabled: false
    cache:
      #启用本地缓存，根据实际情况权衡
      enabled: true
      #缓存空间大小
      capacity: 1000
      #缓存的存活时间，单位s
      ttl: 2
      #caffeine缓存的配置，需引入caffeine依赖
      caffeine:
        #initialCapacity初始的缓存空间大小,expireAfterWrite最后一次写入后经过固定时间过期
        spec: initialCapacity=500,expireAfterWrite=5s
    health-check:
      #重新运行运行状况检查计划程序的时间间隔。
      interval: 25s
      #运行状况检查计划程序的初始延迟值
      initial-delay: 30
    retry: #需要引入Spring Retry依赖
      #该参数用来开启重试机制，默认是关闭
      enabled: true
      #切换实例的重试次数
      max-retries-on-next-service-instance: 2
      #对当前实例重试的次数
      max-retries-on-same-service-instance: 0
      #对所有的操作请求都进行重试
      retry-on-all-operations: true
      #Http响应码进行重试
      retryable-status-codes: 500,404,502
```

mall-user调用mall-order获取用户订单信息为例

1）引入依赖

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-loadbalancer</artifactId>
</dependency>
```

2）使用RestTemplate进行服务调用

给 RestTemplate 实例添加 @LoadBalanced 注解，开启 @LoadBalanced 与 loadbalancer 的集成

```java
@Configuration
public class RestConfig {

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

}
```

3) mall-user中编写调用逻辑，调用mall-order服务

```java
@RequestMapping(value = "/findOrderByUserId/{id}")
public R  findOrderByUserId(@PathVariable("id") Integer id) {
    //利用@LoadBalanced，restTemplate需要添加@LoadBalanced注解
    String url = "http://mall-order/order/findOrderByUserId/"+id;
    R result = restTemplate.getForObject(url,R.class);
    return result;
}
```

测试：http://localhost:8040/user/findOrderByUserId/1，返回数据：

3.6 Nacos注册中心常用配置

服务分级存储模型

注册中心的核心数据是服务的名字和它对应的网络地址，当服务注册了多个实例时，我们需要对不 健康的实例进行过滤或者针对实例的一些特征进行流量的分配，那么就需要在实例上存储一些例如 健康状态、权重等属性。随着服务规模的扩大，渐渐的又需要在整个服务级别设定一些权限规则、 以及对所有实例都生效的一些开关，于是在服务级别又会设立一些属性。再往后，我们又发现单个 服务的实例又会有划分为多个子集的需求，例如一个服务是多机房部署的，那么可能需要对每个机 房的实例做不同的配置，这样又需要在服务和实例之间再设定一个数据级别。

Nacos 在经过内部多年生 产经验后提炼出的数据模型，则是一种服务-集群-实例的三层模型。这样基本可以满足 服务在所有场景下的数据存储和管理。

集群配置

在原有配置加入以下配置

```yaml
spring:
  application:
    name: mall-order  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: nacos.mall.com:8848
        cluster-name: SH
```

案例：跨集群调用优先本地集群的场景实现

利用cluster-name可以实现跨集群调用时，优先选择本地集群的实例，本地集群不可访问时，再去访问其他集群。

下面是Ribbon的NacosRule实现的负载均衡算法，就是利用了cluster-name实现了优先调用本地集群实例。

```
public Server choose(Object key) {
    try {
        String clusterName = this.nacosDiscoveryProperties.getClusterName();
        String group = this.nacosDiscoveryProperties.getGroup();
        DynamicServerListLoadBalancer loadBalancer = (DynamicServerListLoadBalancer) getLoadBalancer();
        String name = loadBalancer.getName();

        NamingService namingService = nacosServiceManager.getNamingService();
        List<Instance> instances = namingService.selectInstances(name, group, b: true);
        if (CollectionUtils.isEmpty(instances)) {
            LOGGER.warn("no instance in service {}", name);
            return null;
        }
        instances = filterInstanceByIpType(instances);

        List<Instance> instancesToChoose = instances;
        if (StringUtils.isNotBlank(clusterName)) {
            List<Instance> sameClusterInstances = instances.stream()
                    .filter(instance -> Objects.equals(clusterName,
                            instance.getClusterName()))
                    .collect(Collectors.toList());
            if (!CollectionUtils.isEmpty(sameClusterInstances)) {
                instancesToChoose = sameClusterInstances;
            }
            else {
                LOGGER.warn(
                        "A cross-cluster call occurs, name = {}, clusterName = {}, instance = {}",
                        name, clusterName, instances);
            }
        }

        Instance instance = ExtendBalancer.getHostByRandomWeight2(instancesToChoose);
        convertIPv4ToIPv6(instance);

        return new NacosServer(instance);
    }
    catch (Exception e) {
```

LoadBalancer默认情况下使用的ReactiveLoadBalancer实现是RoundRobinLoadBalancer。要切换到不同的实现，无论是针对所选服务还是所有服务，您都可以使用自定义LoadBalancer配置机制。

```
# 注意：  不要用@Configuration修饰
public class CustomLoadBalancerConfiguration {

    @Bean
    ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(Environment environment,
            LoadBalancerClientFactory loadBalancerClientFactory,
NacosDiscoveryProperties nacosDiscoveryProperties){
        String name = environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new NacosLoadBalancer(loadBalancerClientFactory
                .getLazyProvider(name,
ServiceInstanceListSupplier.class),name,nacosDiscoveryProperties);
```

```
    }
}
```

在启动类上添加@LoadBalancerClient注解

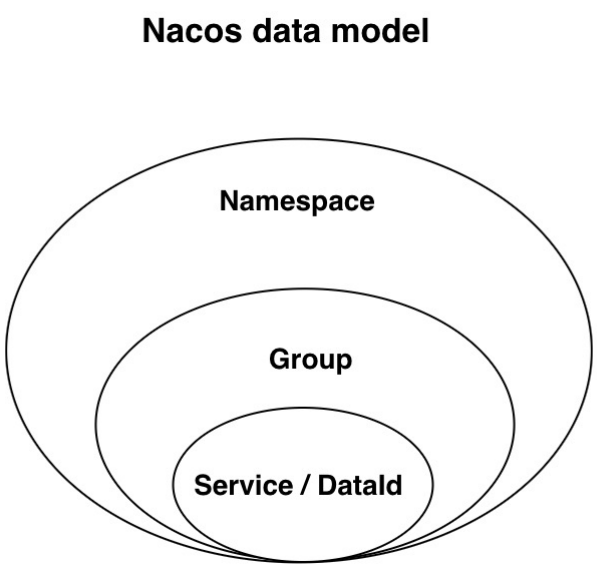
```
@SpringBootApplication
@LoadBalancerClient(value = "mall-order", configuration =
CustomLoadBalancerConfiguration.class)
public class MallUserApplication {

    public static void main(String[] args) {
        SpringApplication.run(MallUserApplication.class, args);
    }

}
```

服务逻辑隔离

Nacos 数据模型 Key 由三元组唯一确定, Namespace默认是空串，公共命名空间（public），分组默认是DEFAULT_GROUP。



Namespace 隔离设计

命名空间(Namespace)用于进行租户（用户）粒度的隔离，Namespace 的常用场景之一是不同环境的隔离，例如开发测试环境和生产环境的资源（如配置、服务）隔离等。

修改yml配置

```yaml
spring:
  application:
    name: mall-user  #微服务名称

  cloud:
    nacos:
      discovery:
        server-addr: 127.0.0.1:8848   #配置nacos注册中心地址
        namespace: bc50d386-8870-4a26-8803-0187486c57be  # dev 开发环境
```

启动mall-user，进入nacos控制台可以看到mall-user注册成功，所属namespace是dev

测试：http://localhost:8040/user/findOrderByUserId/1，报错

原因：mall-order和mall-user使用了不同的namespace，导致服务隔离，mall-user无法发现可用的mall-order服务。

group服务分组

不同的服务可以归类到同一分组，group也可以起到服务隔离的作用。yml中可以通过spring.cloud.nacos.discovery.group参数配置。group更多应用场景是配置分组。

临时实例和持久化实例

在定义上区分临时实例和持久化 实例的关键是健康检查的方式。临时实例使用客户端上报模式，而持久化实例使用服务端反向探测模式。临时实例需要能够自动摘除不健康实例，而且无需持久化存储实例。持久化实例使用服务端探测的健康检查方式，因为客户端不会上报心跳， 所以不能自动摘除下线的实例。

在大中型的公司里，这两种类型的服务往往都有。一些基础的组件例如数据库、缓存等，这些往往不能上报心跳，这种类型的服务在注册时，就需要作为持久化实例注册。而上层的业务服务，例如 微服务，服务的 Provider 端支持添加汇报心跳的逻辑，此时就可以使用动态服务的注册方式。

Nacos 1.x 中持久化及非 持久化的属性是作为实例的一个元数据进行存储和识别。Nacos 2.x 中继续沿用了持久化及非持久化的设定，但是有了一些调整。在 Nacos2.0 中将是否持久化的数据抽象至服务级别， 且不再允许一个服务同时存在持久化实例和非持久化实例，实例的持久化属性继承自服务的持久化属性。

```
# 持久化实例
spring.cloud.nacos.discovery.ephemeral: false
```

nacos开启权限认证

> https://nacos.io/zh-cn/docs/v2/guide/user/auth.html

nacos server端 conf/application.properties添加如下配置

```
# 开启认证
nacos.core.auth.enabled=true
# 配置自定义身份识别的key（不可为空）和value（不可为空）
#这两个属性是auth的白名单，用于标识来自其他服务器的请求。具体实现见com.alibaba.nacos.core.auth.AuthFilter
nacos.core.auth.server.identity.key=authKey
nacos.core.auth.server.identity.value=nacosSecurty
```

微服务端 application.yml中添加如下配置

```
spring:
  application:
    name: mall-order  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: nacos.mall.com:8848
        username: nacos
        password: nacos
```

如果没有配置username，password，微服务端启动会抛出如下错误：

```
Caused by: com.alibaba.nacos.api.exception.NacosException Create breakpoint : user not found!
    at com.alibaba.nacos.client.naming.remote.gprc.NamingGrpcClientProxy.requestToServer(NamingGrpcClientProxy
    at com.alibaba.nacos.client.naming.remote.gprc.NamingGrpcClientProxy.doRegisterService(NamingGrpcClientPro
    at com.alibaba.nacos.client.naming.remote.gprc.NamingGrpcClientProxy.registerService(NamingGrpcClientProxy
    at com.alibaba.nacos.client.naming.remote.NamingClientProxyDelegate.registerService(NamingClientProxyDeleg
    at com.alibaba.nacos.client.naming.NacosNamingService.registerInstance(NacosNamingService.java:143) ~[naco
```

3.7 Nacos集群搭建

> 官网文档： https://nacos.io/zh-cn/docs/v2/guide/admin/cluster-mode-quick-start.html

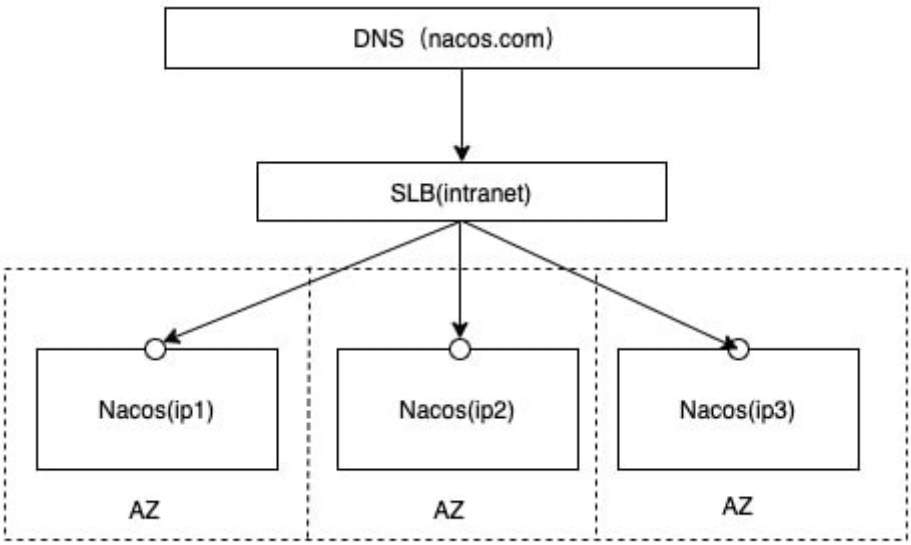
集群部署架构图

因此开源的时候推荐用户把所有服务列表放到一个vip下面，然后挂到一个域名下面

http://ip1:port/openAPI 直连ip模式，机器挂则需要修改ip才可以使用。

http://SLB:port/openAPI 挂载SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，直连SLB即可，下面挂server真实ip，可读性不好。

http://nacos.com:port/openAPI 域名 + SLB模式(内网SLB，不可暴露到公网，以免带来安全风险)，可读性好，而且换ip方便，推荐模式



端口	与主端口的偏移量	描述
8848	0	主端口，客户端、控制台及OpenAPI所使用的HTTP端口
9848	1000	客户端gRPC请求服务端端口，用于客户端向服务端发起连接和请求
9849	1001	服务端gRPC请求服务端端口，用于服务间同步等
7848	-1000	Jraft请求服务端端口，用于处理服务端间的Raft相关请求

使用VIP/nginx请求时，需要配置成TCP转发，不能配置http2转发，否则连接会被nginx断开。 9849和7848端口为服务端之间的通信端口，请勿暴露到外部网络环境和客户端测。

三节点集群搭建

1）环境准备

- 安装好 JDK，需要 1.8 及其以上版本
- 建议: 2核 CPU / 4G 内存 及其以上
- 建议: 生产环境 3 个节点 及其以上

```
# 准备三台centos7服务器
192.168.65.174
192.168.65.192
192.168.65.204
```

- 准备好nacos安装包

2）配置集群配置文件

在nacos的解压目录nacos/的conf目录下，有配置文件cluster.conf，请每行配置成ip:port。

```
mv conf/cluster.conf.example conf/cluster.conf
vim  conf/cluster.conf

# ip:port
192.168.65.174:8848
192.168.65.192:8848
192.168.65.204:8848
```

注意：不要使用localhost或127.0.0.1，针对多网卡环境，nacos可以指定网卡或ip

```
#多网卡选择
#ip-address参数可以直接设置nacos的ip
#该参数设置后，将会使用这个IP去cluster.conf里进行匹配，请确保这个IP的值在cluster.conf里是存在的
nacos.inetutils.ip-address=10.11.105.155

#use-only-site-local-interfaces参数可以让nacos使用局域网ip，这个在nacos部署的机器有多网卡时很有用，可以让nacos选择局域网网卡
nacos.inetutils.use-only-site-local-interfaces=true

#ignored-interfaces支持网卡数组，可以让nacos忽略多个网卡
```

```
nacos.inetutils.ignored-interfaces[0]=eth0
nacos.inetutils.ignored-interfaces[1]=eth1

#preferred-networks参数可以让nacos优先选择匹配的ip，支持正则匹配和前缀匹配
nacos.inetutils.preferred-networks[0]=30.5.124.
nacos.inetutils.preferred-networks[0]=30.5.124.(25[0-5]|2[0-4]\\d|((1d{2})|([1-9]?\\d))),30.5.124.(25[0-5]|2[0-4]\\d|((1d{2})|([1-9]?\\d)))
```

3）开启默认鉴权插件

修改conf目录下的application.properties文件

```
nacos.core.auth.enabled=true
nacos.core.auth.system.type=nacos
nacos.core.auth.plugin.nacos.token.secret.key=${自定义，保证所有节点一致}
nacos.core.auth.server.identity.key=${自定义，保证所有节点一致}
nacos.core.auth.server.identity.value=${自定义，保证所有节点一致}
```

4）配置数据源

使用外置mysql数据源，生产使用建议至少主备模式

4.1）初始化 MySQL 数据库

sql脚本：https://github.com/alibaba/nacos/blob/2.2.1/distribution/conf/mysql-schema.sql

4.2）修改application.properties配置

```
spring.datasource.platform=mysql
db.num=1
db.url.0=jdbc:mysql://192.168.65.174:3306/nacos_devtest?characterEncoding=utf8&connectTimeout=1000&socketTimeout=3000&autoReconnect=true&useUnicode=true&useSSL=false&serverTimezone=UTC
db.user.0=root
db.password.0=root
```

5) 分别启动三个nacos节点

以192.168.65.204为例，进入nacos目录，启动nacos

```
bin/startup.sh
```

6) 访问nacos管理界面

登录http://192.168.65.204:8848/nacos，用户名和密码都是nacos

微服务yml中配置

```yaml
spring:
  application:
    name: mall-user  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: 192.168.65.174:8848,192.168.65.192:8848,192.168.65.204:8848
        username: nacos
        password: nacos
```

Nginx配置负载均衡

使用VIP/nginx请求时，需要配置成TCP转发，不能配置http2转发，否则连接会被nginx断开。 9849和7848端口为服务端之间的通信端口，请勿暴露到外部网络环境和客户端测。

1）准备nginx环境

1.1 ）如果安装了nginx，先检查nginx是否有stream模块，输出中包含：--with-stream

```
nginx -V
```

1.2) 安装nginx

```
#安装依赖包
yum -y install gcc gcc-c++ autoconf automake
yum -y install zlib zlib-devel openssl openssl-devel pcre-devel

# 下载nginx
```

```
wget https://nginx.org/download/nginx-1.18.0.tar.gz
tar -zxvf nginx-1.18.0.tar.gz
cd nginx-1.18.0

#编译nginx  如果使用 nginx 的 stream 功能，在编译时一定要加上“--with-stream”
./configure --with-stream
make && make install
#安装后nginx默认路径/usr/local/nginx
```

2）配置http模块

在nginx的http下面配置http协议相关的地址和端口：

```
http {
    # nacos服务器http相关地址和端口
    upstream nacos-server {
        server 192.168.65.174:8848;
        server 192.168.65.192:8848;
        server 192.168.65.204:8848;
    }
    server {
        listen 8848;
        location / {
            proxy_pass http://nacos-server/;
        }
    }
}
```

3）配置grpc

需要nginx有stream模块支持

```
# nacos服务器grpc相关地址和端口，需要nginx已经有stream模块
# stream块用于做TCP转发
stream {
    upstream nacos-server-grpc {
        server 192.168.65.174:9848;
        server 192.168.65.192:9848;
```

```
        server 192.168.65.204:9848;
    }
    server {
        listen 9848;
        proxy_pass nacos-server-grpc;
    }
}
```

4) 启动nginx，然后就可以正常使用了。

```
sbin/nginx -c conf/nginx.conf
```

微服务yml中配置

```
spring:
  application:
    name: mall-user  #微服务名称

  #配置nacos注册中心地址
  cloud:
    nacos:
      discovery:
        server-addr: nacos.mall.com:8848  #nacos.mall.com 需建立和nginx ip的域名映射
        username: nacos
        password: nacos
```