

【微服务】

<https://www.cnblogs.com/dreamroute/p/10980423.html>

1. 微服务简介

随着互联网技术的飞速发展，目前全球超过半的人口在使用互联网，人们的生活随着互联网的发展，发生了翻天覆地的变化。各行各业都在应用互联网 国家政策也在大力支持互联网的发展。随着越来越多的用户参与，业务场景越来越复杂，传统的单体架构已经很难满足互联网技术的发展要求。这主要体现在两方面， 是随着业务复杂度的提高，代码的可维护性、扩展性和可读性在降低； 是维护系统的成本、修改系统的成本在提高。所以，改变单体应用架构已经势在必行。另外，随着云计算、大数据、人工智能的飞速发展，对系统架构也提出了越来越高的要求。

微服务(不是一个框架 而是一种架构思想)，是著名的 oo（面向对象， Object Oriented）专家 Martin Fowler 提出来的，它是用来描述将软件应用程序设计为独立部署的服务的种特殊方式。最近两年，微服务在各大技术会议、文章、书籍上出现的频率已经让人意识到它对于软件领域所带来的影响力。**微服务架构的系统是个分布式系统，按业务领域划分为独立的服务单元，有自动化运维、容错、快速演进的特点，它能够解决传统单体架构系统的痛点，同时也能满足越来越复杂的业务需求。**

1.1 单体架构不足

在应用的初始阶段，单体架构无论是在开发速度、运维难度上，还是服务器的成本上都有着显著的优势。在一个产品的前景不明确的初始阶段，用单体架构是非常明智的选择。随着应用业务的发展和业务复杂度的提高，这种架构明显存在很多的不足，主要体现在以下 3 个方面：

1. 业务越来越复杂，单体应用的代码量越来越大，代码的可读性、可维护性和可扩展性下降，新人接手代码所需的时间成倍增加，业务扩展带来的代价越来越大。
2. 随着用户越来越多，程序承受的并发越来越高，单体应用的并发能力有限。
3. 测试的难度越来越大，单体应用的业务都在同个程序中，随着业务的扩张、复杂度的增加，单体应用修改业务或者增加业务或许会给其他业务带来定的影响，导致测试难度增加。

1.2 到底什么是微服务

什么是微服务呢？

就是将一个大的应用，拆分成多个小的模块，每个模块都有自己的功能和职责，每个模块可以进行交互，这就是微服务

对于微服务，业界没有严格统一的定义，但是作为“微服务”这名词的发明人，Martin Fowler 对微服务的定义似乎更具有权威性和指导意义，他的理解如下：

简而言之，微服务架构的风格，就是将单一程序开发成一个微服务，每个微服务运行在自己的进程中，并使用轻量级通信机制，通常是 HTTP RESTFUL API。这些服务围绕业务能力来划分构建的，并通过完全自动化部署机制来独立部署这些服务可以使用不同的编程语言，以及不同数据存储技术，以保证最低限度的集中式管理。

1.2.1 总结出微服务的特点

1. 按业务(功能)划分为一个独立运行的程序，即服务单元。
2. 服务之间通过 HTTP 协议相互通信。http 是一个万能的协议 (web 应用都支持的模式)
3. 自动化部署。
4. 可以用不同的编程语言。
5. 可以用不同的存储技术。
6. 服务集中化管理。
7. 微服务是一个分布式系统。

1.3 微服务特点的具体阐述

1.3.1 微服务单元按业务来划分（不是绝对的）

微服务的“微”到底需要定义到什么样的程度，这是个非常难以界定的概念，可以从以个方面来界定：是根据代码量来定义，根据代码的多少来判断程序的大小：是根据开发时间的长短来判断：是根据业务的大小来划分。根据 Martin Fowler 的定义，微服务的“微”是按照业务来划分的。一个大的业务可以拆分成若干小的业务，个小的业务又可以拆分成若干更小的业务，业务到底怎么拆分才算合适，这需要开发人员自己去决定。例如微博最常见的功能是微博内容、关注和粉丝，而其中微博内容又有点赞、评论等，如何将微博这个复杂的程序划分

为单个的服务，需要由开发团队去决定。按业务划分的微服务单元独立部署，运行在独立的进程中 这些微服务单元是高度组件化的模块，并提供了稳定的模块边界，服务与服务之间没有任何的耦合 有非常好的扩展性和复用性。传统的软件开发模式通常由 UI 团队、服务端团队、数据库和运维团队构成，相应地将软件按照职能划分为 UI、服务端、数据库和运维等模块。通常这些开发人员各司其职 很少有人跨职能去工作。如果按照业务来划分服务，每个服务都需要独立的 UI、服务端、数据库和运维。也就是说，一个小的业务的微服务需要动用整个团队的人去协作，这显然增加了团队与团队之间交流协作的成本。所以产生了跨职能团队，这个团队负责一个服务的所有工作，包括 UI、服务端和数据库。当这个团队只有一个人的时候，就对开发人员提出了更高的要求。

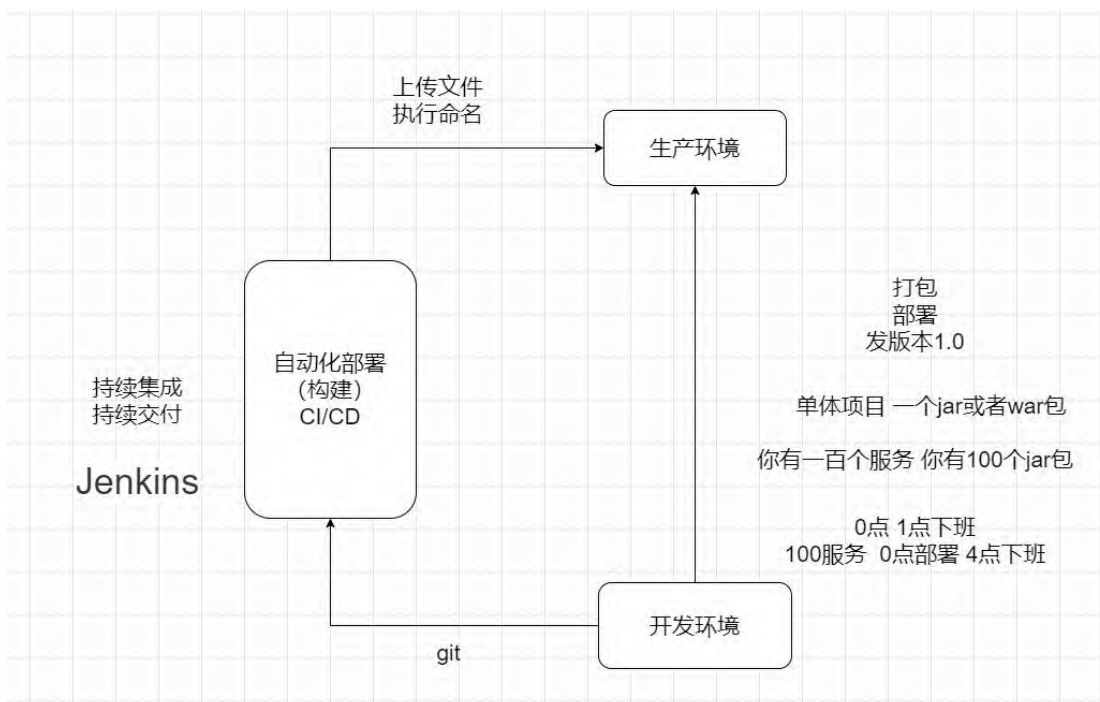
1.3.2 微服务通过 HTTP 来互相通信

按照业务划分的微服务单元独立部署 并运行在各自的进程中。微服务单元之间的通信方式般倾向于使用 HTTP 这种简单的通信机制，更多的时候是使用 RESTfulAPI。这种接受请求、处理业务逻辑、返回数据的 HTTP 模式非常高效，并且这种通信机制与平台和语言无关。例如用 Java 写的服务可以消费用 Go 语言写的服务

1.3.3 微服务的数据库独立

在单体架构中，所有的业务都共用一个数据库。随着业务量的增加，数据库的表的数量越来越多，难以管理和维护，并且数据量的增长会导致查询速度越来越慢。例如一个应用有这样几个业务：用户的信息、用户的账户、用户的购物、数据报表服务等。典型的单体架构如微服务的一个特点就是按业务划分服务，服务与服务之间无耦合，就连数据库也是独立的。一个典型的微服务的架构就是每个微服务都有自己独立的数据库，数据库之间没有任何联系这样做的好处在于，随着业务的不断扩张，服务与服务不需要提供数据库集成，而是提供 API 接口相互调用；还有一个好处是数据库独立，单业务的数据量少，易于维护，数据库性能有着明显的优势，数据库的迁移也很方便。另外，随着存储技术的发展，数据库的存储方式不再仅仅是关系型数据库，非关系型数据库的应用也非常广泛，例如 MongoDB，它们有着良好的读性能，因此越来越受欢迎。一个典型的微服务的系统，可能每个服务的数据库都不相同，每个服务所使用的数据存储技

1.3.4 微服务的自动化部署 (CI /CD) (持续集成 持续交付)



在微服务架构中，系统会被拆分为若干个微服务，每个微服务又是一个独立的应用程序。单体架构的应用程序只需要部署一次，而微服务架构有多少个服务就需要部署多少次。随着服务数量的增加，如果微服务按照单体架构的部署方式，部署的难度会呈指数增加。业务的粒度划分得越细，微服务的数量就越多，这时需要更稳定的部署机制。随着技术的发展，尤其是 Docker 容器技术的推进，以及自动化部署工具（例如开源组件 Jenkins）的出现，自动化部署变得越来越简单。

自动化部署可以提高部署的效率，减少人为的控制，部署过程中出现错误的概率降低，部署过程的每一步自动化，提高软件的质量。构建一个自动化部署的系统，虽然在前期需要开发人员或者运维人员的学习，但是对于整个软件系统来说是一个全新的概念。在软件系统的整个生命周期之中，每一步是由程序控制的，而不是人为控制，软件的质量提高到了一个新的高度。随着 DevOps 这种全新概念的推进，自动化部署必然会成为微服务部署的一种方式。

1.3.5 服务集中化管理

微服务系统是按业务单元来划分服务的，服务数量越多，管理起来就越复杂，因此微服务必须使用集中化管理。目前流行的微服务框架中，例如 Spring Cloud 采用 Eureka 来注册服务和发现服务，另外， Zookeeper、 Consul 等都是非常优秀的服务集中化管理框架。

1.3.6 分布式架构

分布式系统是集群部署的，由很多计算机相互协作共同构成，它能够处理海量的用户请求。当分布式系统对外提供服务时，用户是毫不知情的，还以为是一台服务器在提供服务。分布式系统的复杂任务通过计算机之间的相互协作来完成，当然简单的任务也可以在一台计算机上完成。分布式系统通过网络协议来通信，所以分布式系统在空间上没有任何限制，即分布式服务器可以部署不同的机房和不同的地区。微服务架构是分布式架构，分布式系统比单体系统更加复杂，主要体现在服务的独立性和服务相互调用的可靠性，以及分布式事务、全局锁、全局 Id 等，而单体系统不需要考虑这些复杂性。

另外，分布式系统的应用都是集群化部署，会给数据一致性带来困难。分布式系统中的服务通信依赖于网络，网络不好，必然会对分布式系统带来很大的影响。在分布式系统中，服务之间相互依赖，如果一个服务出现了故障或者是网络延迟，在高并发的情况下，会导致线程阻塞，在很短的时间内该服务的线程资源会消耗殆尽，最终使得该服务不可用。由于服务的相互依赖，可能会导致整个系统的不可用，这就是“雪崩效应”。为了防止此类事件的发生，分布式系统必然要采取相应的措施，例如“熔断机制”。

1.3.7 熔断机制 Hystrix

为了防止“雪崩效应”事件的发生，分布式系统采用了熔断机制。在用 SpringCloud 构建的微服务系统中，采用了熔断器（即 Hystrix 令组件的 Circuit Breaker）去做熔断。例如在微服务系统中，有 a、b、c、d、e、

如果此时服务 b 出现故障或者网络延迟，服务 b 会出现大量的线程阻塞，有可能在很短的时间内线程资源就被消耗完了，导致服务 b 的不可用。如果服务 b 为较底层的服务，会影响到其他服务，导致其他服务会一直等待服务 b 的处理。如果服务 b 迟迟不处理，大量的网络请求不仅仅堆积在服务 b，而且会堆积到依赖于服务 b 的其他服务。而因服务 b 出现故障影响的服务，也会影响到依赖于因服务 b 出现故障影响的服务的其他服务，从而由 b 开始，影响到整个系统，导致整个系统的不可用。这是一件非常可怕的事，因为服务器运营商的不可靠，必然会导致服务的不可靠，而网络服务商的不可靠性，也会导致服务的不可靠。在高并发的场景下，稍微有点不可靠，由于故障的传播性，会导致大量的服务不可用，甚至导致整个系统崩溃。

为了解决这一难题，微服务架构引入了熔断机制。当服务 b 出现故障，请求失败次数超过设定的阈值之后，服务 b 就会开启熔断器，之后服务 b 不进行任何的逻辑操作，执行快速

失败，直接返回请求失败的信息。其他依赖于 b 的服务就不会因为得不到响应而线程阻塞，这时除了服务 b 和依赖于服务 b 的部分功能不可用外，其他功能正常。

1.4 微服务的优势

相对于单体服务来说，微服务具有很多的优势，主要体现在以下方面。

1. 将一个复杂的业务分解成若干小的业务，每个业务拆分成一个服务，服务的边界明确，将复杂的问题简单化。服务按照业务拆分，编码也是按照业务来拆分，代码的可读性和可扩展性增加。新人加入团队，不需要了解所有的业务代码，只需要了解他所接管的服务的代码，新人学习时间成本减少。
2. 由于微服务系统是分布式系统，服务与服务之间没有任何的耦合。随着业务的增加，可以根据业务再拆分服务，具有极强的横向扩展能力。随着应用的用户量的增加，并发量增加，可以将微服务集群化部署，从而增加系统的负载能力。简而言之，微服务系统的微服务单元具有很强的横向扩展能力。
3. 服务与服务之间通过 HTTP 网络通信协议来通信，单个微服务内部高度耦合，服务与服务之间完全独立，无耦合。这使得微服务可以采用任何的开发语言和技术来实现。开发人员不再被强迫使用公司以前的技术或者已经过时的技术，而是可以自由选择最适合业务场景的或者最适合自己的开发语言和技术，提高开发效率、降低开发成本。
4. 如果是一个单体的应用，由于业务的复杂性、代码的耦合性，以及可能存在的历史问题。在重写一个单体应用时，要求重写的应用的人员了解所有的业务，所以重写单体应用是非常困难的，并且重写风险也较高。如果是微服务系统，由于微服务系统是按照业务的进行拆分的，并且有坚实的服务边界，所以重写某个服务就相当于重写某一个业务的代码，非常简单。
5. 微服务的每个服务单元都是独立部署的，即独立运行在某个进程里。微服务的修改和部署对其他服务没有影响。试想，假设一个应用只有一个简单的修改，如果是单体架构，需要测试和部署整个应用；而如果采用微服务架构，只需要测试并部署被修改的那个服务，这就大大减少了测试和部署的时间。
6. 微服务在 CAP 理论中采用的是 AP 架构，即具有高可用和分区容错的特点。高可用主要体现在系统 7 x 24 小时不间断的服务，它要求系统有大量的服务器集群，从而提高了系统的负载能力。另外，分区容错也使得系统更加健壮。

1.5 微服务的不足(正视它的不足)

凡事都有两面性，微服务也不例外，微服务相对于单体应用来说具有很多的优势，当然也有它的不足，主要体现在如下方面：

1. 微服务的复杂度
2. 分布式事务问题
3. 服务的划分（按照功能划分 还是按照组件来划分呢） 分工
4. 服务的部署（不用自动化部署 自动化部署）

1.6 微服务架构的设计原则（项目起步搭建）

开闭原则 单一原则 6 大设计原则 架构设计和代码设计思路一样的

软件设计就好比建筑设计。Architect 这个词在建筑学中是“建筑师”的意思，而在软件领域里则是“架构师”的意思，可见它们确实有相似之处。无论是建筑师还是架构师，他们都希望把作品设计出自己的特色，并且更愿意把创造出的东西被称为艺术品。然而现实却是，建筑设计和软件设计有非常大的区别。建筑师设计并建造出来的建筑往往很难有变化，除非拆了重建。而架构师设计出来的软件系统，为了满足产品的业务发展，在它的整个生命周期中，每一个版本都有很多的变化。

软件设计每一个版本都在变化，所以软件设计应该是渐进式发展。软件从一开始就不应该被设计成微服务架构，微服务架构固然有优势，但是它需要更多的资源，包括服务器资源、技术人员等。追求大公司所带来的技术解决方案，刻意地追求某个新技术，企图使用技术解决所有的问题，这些都是软件设计的误区。

技术应该是随着业务的发展而发展的，任何脱离业务的技术是不能产生价值的。在初创公司，业务很单一时，如果在 LAMP 单体构架够用的情况下，就应该用 LAMP，因为它开发速度快，性价比高。随着业务的发展，用户量的增加，可以考虑将数据库读写分离、加缓存、加复杂均衡服务器、将应用程序集群化部署等。如果业务还在不断发展，这时可以考虑使用分布式系统，例如微服务架构的系统。不管使用什么样的架构，驱动架构的发展一定是业务的发展，只有当前架构不再适合当前业务的发展，才考虑更换架构。

在微服务架构中，**有三大难题**，那就是****服务故障的传播性(熔断)、服务的划分和分布式事务**。在微服务设计时，一定要考虑清楚这三个难题，从而选择合适的框架。目前比较流行的微服务框架有 Spring 社区的 Spring Cloud、Google 公司的 Kubernetes 等。不管使用哪

一种框架或者工具，都需要考虑这三大难题。为了解决服务故障的传播性，一般的微服务框架都有熔断机制组件。另外，服务的划分没有具体的划分方法，一般来说根据业务来划分服务，领域驱动设计具有指导作用。最后，分布式事务一般的解决办法就是两阶段提交或者三阶段提交，不管使用哪一种都存在事务失败，导致数据不一致的情况，关键时刻还得人工去恢复数据。总之，微服务的设计一定是渐进式的，并且是随着业务的发展而发展的。

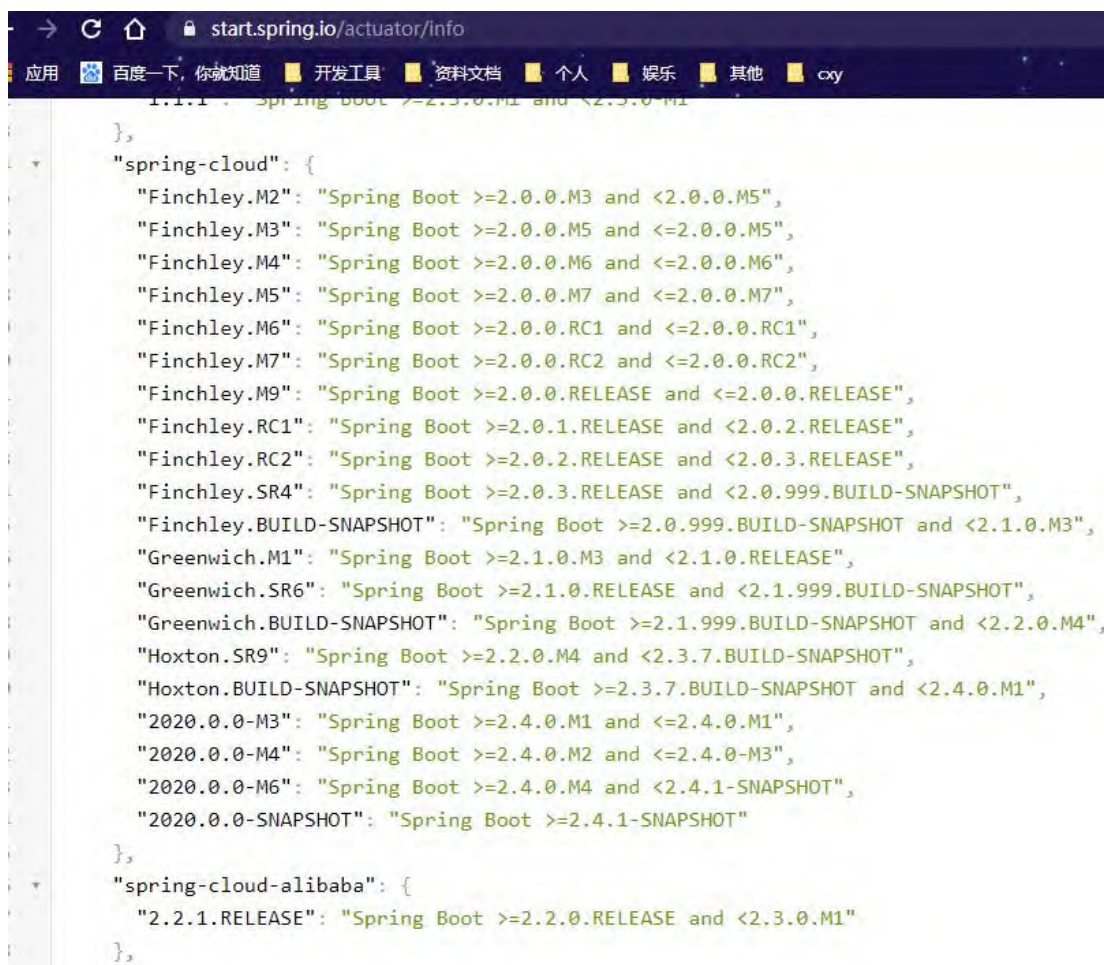
2. SpringCloud 简介(管家 注重服务的管理)

Spring Cloud 作为 Java 语言的微服务框架，它依赖于 Spring Boot，有快速开发、持续交付和容易部署等特点。Spring Cloud 的组件非常多，涉及微服务的方方面面，并在开源社区 Spring 和 Netflix、Pivotal 两大公司的推动下越来越完善，如今 alibaba 也加入到其中。spring 官方 netflix alibaba

Spring Cloud 在开发部署上继承了 Spring Boot 的一些优点，提高其在开发和部署上的效率。Spring Cloud 的首要目标就是通过提供一系列开发组件和框架，帮助开发者迅速搭建一个分布式的微服务系统。Spring Cloud 是通过包装其他技术框架来实现的，例如包装开源的 Netflix oss 组件，实现了一套通过基于注解、Java 配置和基于模版开发的微服务框架。Spring Cloud 提供了开发分布式微服务系统的一些常用组件，例如服务注册和发现、配置中心、熔断器、远程调用，智能路由、微代理、控制总线、全局锁、分布式会话等。

2.1 SpringCloud 版本对应关系【开发重点】

A B C D E **F G H I** (2020 版) Hoxton.SR12 2.3.12.RELEASE
<https://start.spring.io/actuator/info>



```
11.1.1 - Spring Boot >=2.3.0.M1 and <2.3.0.M1
},
"spring-cloud": {
  "Finchley.M2": "Spring Boot >=2.0.0.M3 and <2.0.0.M5",
  "Finchley.M3": "Spring Boot >=2.0.0.M5 and <=2.0.0.M5",
  "Finchley.M4": "Spring Boot >=2.0.0.M6 and <=2.0.0.M6",
  "Finchley.M5": "Spring Boot >=2.0.0.M7 and <=2.0.0.M7",
  "Finchley.M6": "Spring Boot >=2.0.0.RC1 and <=2.0.0.RC1",
  "Finchley.M7": "Spring Boot >=2.0.0.RC2 and <=2.0.0.RC2",
  "Finchley.M9": "Spring Boot >=2.0.0.RELEASE and <=2.0.0.RELEASE",
  "Finchley.RC1": "Spring Boot >=2.0.1.RELEASE and <2.0.2.RELEASE",
  "Finchley.RC2": "Spring Boot >=2.0.2.RELEASE and <2.0.3.RELEASE",
  "Finchley.SR4": "Spring Boot >=2.0.3.RELEASE and <2.0.999.BUILD-SNAPSHOT",
  "Finchley.BUILD-SNAPSHOT": "Spring Boot >=2.0.999.BUILD-SNAPSHOT and <2.1.0.M3",
  "Greenwich.M1": "Spring Boot >=2.1.0.M3 and <2.1.0.RELEASE",
  "Greenwich.SR6": "Spring Boot >=2.1.0.RELEASE and <2.1.999.BUILD-SNAPSHOT",
  "Greenwich.BUILD-SNAPSHOT": "Spring Boot >=2.1.999.BUILD-SNAPSHOT and <2.2.0.M4",
  "Hoxton.SR9": "Spring Boot >=2.2.0.M4 and <2.3.7.BUILD-SNAPSHOT",
  "Hoxton.BUILD-SNAPSHOT": "Spring Boot >=2.3.7.BUILD-SNAPSHOT and <2.4.0.M1",
  "2020.0.0-M3": "Spring Boot >=2.4.0.M1 and <=2.4.0.M1",
  "2020.0.0-M4": "Spring Boot >=2.4.0.M2 and <=2.4.0-M3",
  "2020.0.0-M6": "Spring Boot >=2.4.0.M4 and <2.4.1-SNAPSHOT",
  "2020.0.0-SNAPSHOT": "Spring Boot >=2.4.1-SNAPSHOT"
},
"spring-cloud-alibaba": {
  "2.2.1.RELEASE": "Spring Boot >=2.2.0.RELEASE and <2.3.0.M1"
},
}
```

2.2 SpringCloud 常用组件表（管家）

服务的注册和发现。（eureka,nacos,consul）

服务的负载均衡。（ribbon,dubbo）

服务的相互调用。（openFeign,dubbo）

服务的容错。（hystrix, sentinel）

服务网关。（gateway, zuul）

服务配置的统一管理。（config-server,nacos,apollo）

服务消息总线。（bus）

服务安全组件。（security,oauth2.0）

服务监控。（admin）（jvm）

链路追踪。（sleuth+zipkin）

2.3 总结

SpringCloud 就是微服务理念的一种**具体落地实现方式**，帮助微服务架构提供了必备的功能
目前开发中常用的落地实现有三种：

Dubbo+Zookeeper 半自动化的微服务实现架构 （别的管理没有）

SpringCloud Netflix 一站式微服务架构

SpringCloud Alibaba 新的一站式微服务架构

三大公司

Spring Netflix Alibaba

2.4 最终架构图详见 PDF

你怎么理解微服务的 说说你的理解
结构化回答问题

SpringCloud 30-40% 会有

回顾之前的知识点 ✓

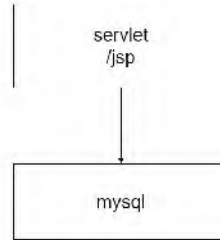
servlet + jsp + mysql

回顾之前的架构分析

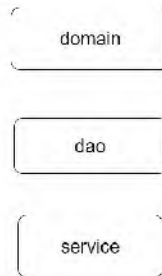
展望未来

ORM--->MVC--->RPC--->SOA
SpringCloud
划分职责清晰
精确部署 降低服务器压力

All in one



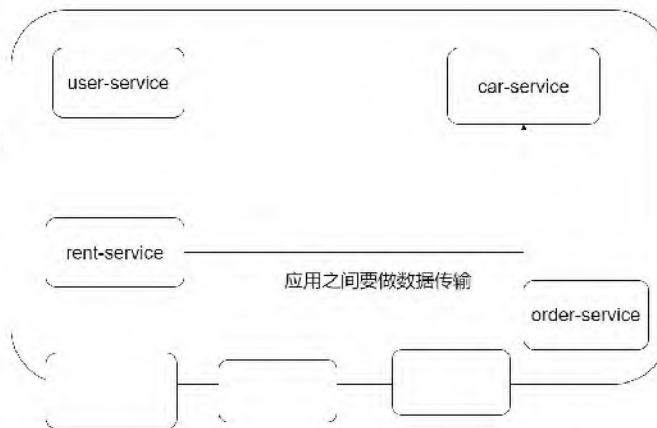
baseDao/DbUtil
自己手动写的dao层框架
ORM 对象关系映射



MVC 理念
model view controller
springMvc是mvc落地产品

boot不是一个架构模式
只是简化了mvc 封装了mvc而已

项目和项目之间
怎么通讯
又要可靠
又要快速
还要轻量化



两个进程之间通讯
mq
http
tcp

云计算

IPV6网络

由美团团队提供云计算服务

绝大部分常用的软件/项目
都是分布式项目

一个请求的结果 由多个服务器共同计算完成

