微服务

微服务背景

2014 年，Martin Fowler 撰写的《Microservices》[1]使得许多国内的先行者接触到微服务这个概念并将其引入国内，2015 年越来越多的人通过各种渠道了解到微服务的概念并有人开始在生产环境中落地，2016-2017 年，微服务的概念被越来越多的人认可，带动了一大批公司以微服务和容器为核心开始技术架构的全面革新，于是微服务架构应运而生。

在微服务架构中，随着微服务承担的职责越来越多，服务间的治理开始变得必要，于是又衍生了一批微服务治理框架，该框架与微服务架构本身息息相关，可以说微服务治理框架解决了微服务架构下遇到的种种难题。

至今微服务已经历了两代发展，第一代以 Dubbo、Spring Cloud 为代表的微服务治理框架，该框架在微服务发展的前几年一度独领风骚，甚至在部分人群中成为微服务的代名词，但事实上该框架并不能友好的解决微服务自身带来的一些问题，例如微服务的调用依赖、版本迭代、安全性、可观测性等；第二代微服务治理框架为服务网格（Service Mesh），它的出现解决了大部分开发人员在使用 Spring Cloud 中遇到的不足和痛点。

微服务概念

Matin Fowler 对微服务概念解释的原文如下：

In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

1 https://martinfowler.com/articles/microservices.html

微服务，就是将一个完整应用中所有的模块拆分成多个不同的服务，其中每个服务都可以进行独立部署、维护和扩展，服务之间通常通过 RESTful API 进行通信，这些服务围绕业务能力构建，且每个服务均可使用不同的编程语言和不同的数据存储技术。

微服务设计的本质在于使用功能较明确、业务较精炼的服务去解决更大、更实际的问题。

微服务架构与单体架构的区别

单体架构顾名思义，即将所有的功能放在单个应用程序中，为传统构建应用程序的方式。通常，单体架构包括一个客户端界面，一个服务端应用和一个数据库，图 0.1 简易描述了单体架构：

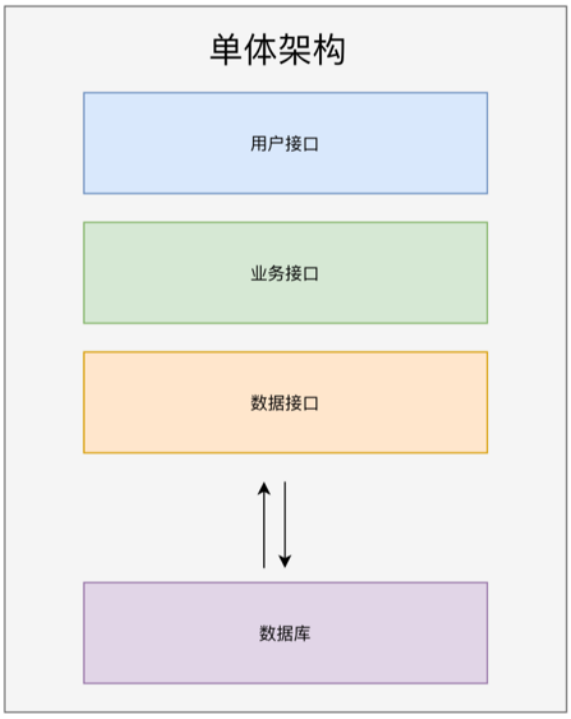

图 0.1 单体架构

这种一体式应用的设计模式拥有许多优势，也同时带有一些缺陷。

单体架构的优势

易于调试

由于单体应用是一个不可分割的单元，因此对于开发者而言容易进行端到端调试和测试；

易于部署

由于单体应用设计模式较为简单，因此应用的部署相对容易；

易于管理

统一的应用程序使得开发、运维、安全这一闭环流程更易于管理；

单体架构的挑战

由于业务规模的不断上升，单体应用变得愈发复杂，因此一系列潜在问题逐渐暴露，例如：

1. 业务难以理解

模块边界随着业务不断增加会变得模糊，使得开发人员难以理解；

2. 复杂度高

应用必会经历性能瓶颈期，由于系统的模块间耦合度低，在修改相应代码时有牵一发而动全身的风险；

3. 扩展性低

由于单体应用只能作为一个整体进行扩展，无法根据业务需求进行弹性伸缩，因此扩展性受到限制；

4. 部署风险高

随着业务规模的增加，应用部署时间也会相应增加，从而导致每次功能变更均需重新部署，而全量部署通常非常耗时且影响范围大，增大了部署风险。

相比微服务架构，其将单体应用分解为一个个独立的单元，其中，每一个独立单元都可由一个团队来进行开发和维护，不同的单元甚至可以使用不同的编程语言和架构去实现，并且每个单元可以独立更新、部署和扩展。

图 0.2 简易描述了微服务架构：

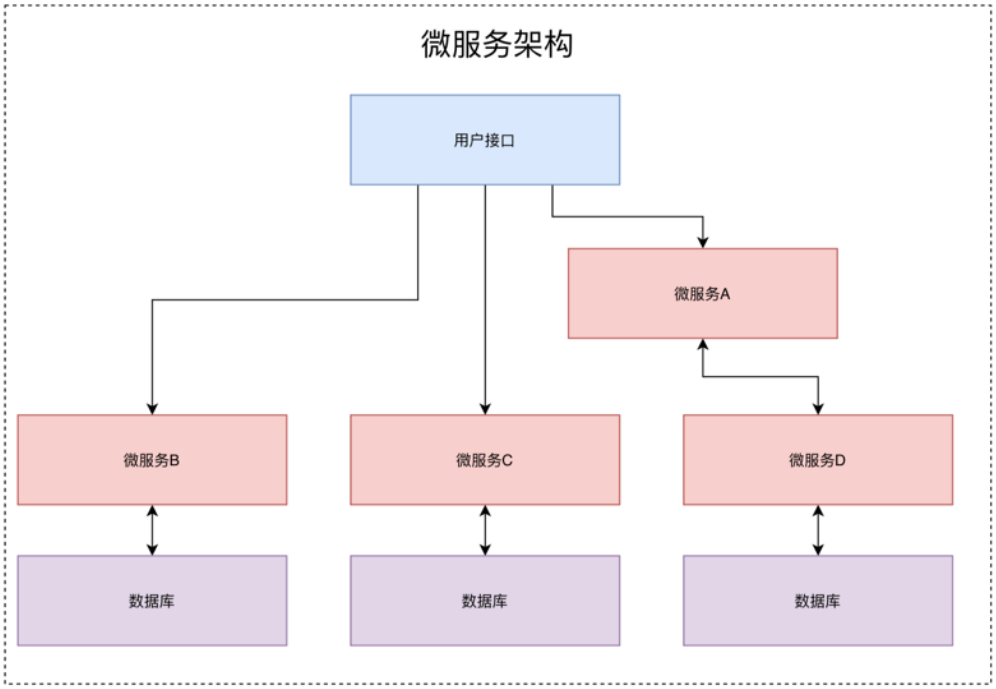

图 0.2 微服务架构

微服务架构的优势

1. 易于开发和维护

由于微服务架构的每个独立单元只关注某一个特定的业务功能，因此有着代码量少、逻辑清晰、易于理解、容错率高、独立部署、扩展性强等优点，进而开发和维护变得更为灵活。

2. 技术栈不受限制

微服务架构下，技术选型是去中心化的，每个团队可根据自身服务的需求选择适合的技术栈。

3. 复杂性可控

由于微服务架构下应用可被拆解为独立单元的特性，因此在一定程度上规避了业务复杂度的问题，每个独立单元专注于单一功能，并通过自定义的 API 清晰描述边界，从而使得应用复杂度可控。

微服务架构的挑战

1. 分布式固有复杂性

由于微服务架构是分布式的，因此需要设置各模块间、模块与数据库间的连接，并且需要保证连接的可靠性。此外，系统容错、网络延迟、分布式事务等问题在微服务体量较大时也将是一项繁重的工作；

2. 运维要求高

更多的微服务意味着需要更多运维投入，当微服务体量较大时，将给运维人员带来挑战；

通过以上介绍，我们可以看出微服务架构与单体架构的区别主要为：

（1）单体架构在设计上具备模块耦合度高，复杂性高的特点，而微服务架构设计原则为高内聚、低耦合；

（2）单体架构下应用部署和运维容易，但随着业务量的增大，可能会伴随一定风险，微服务架构下每个单元独立部署，整体来看部署较为复杂但基本不受业务规模影响；

（3） 单体架构因受其设计理念影响，扩展性差，而微服务架构扩展性强；

主流微服务治理框架介绍

通过上面小节我们对什么是微服务治理框架有了一定了解，本节将针对这些治理框架进行一一介绍。

第一代微服务治理框架

第一代微服务治理框架主要以 Dubbo 和 SpringCloud 为主，这一代运用的技术相对于微服务还有较大的侵入性，但也在早期被大规模使用。

Dubbo

Dubbo 为阿里在 2011 年 11 月开源的分布式服务治理框架，致力于提供高性能和透明化的远程服务调用(Remote Protocedure Call，RPC)方案。目前为 Apache 的开源项目2。

2 https://github.com/apache/dubbo

随着单体架构到微服务架构的转变，Dubbo 架构经历了多次衍变，从其官方提供的架构路线图就可以看得出来，如图 0.3 所示：

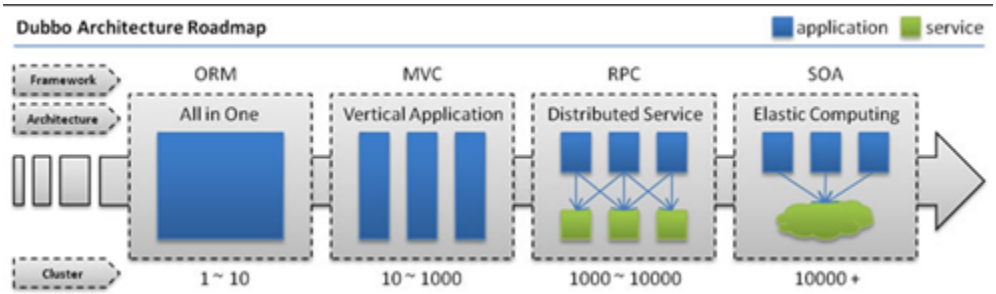

图 0.3 Dubbo 架构技术路线图

Dubbo 在架构设计之初采用传统的 ORM（Object-Relational Mapping ）框架，其集群管理节点数量通常为 1 至 10 个，随着业务的不断增多，单体架构及节点数已无法满足需求，于是 Dubbo 通过 MVC(Model-view-controller)框架实现了将单体应用分为若干个垂直应用，进而提升了开发效率，节点数也支持扩展至 1000 个。当垂直应用数量不断增多，应用之间的交互在所难免，Dubbo 将每个垂直应用的核心业务提取出来并作为独立的服务（图中绿色框），采用 RPC 方式实现分布式服务。随着微服务理念的不断冲击，Dubbo 最终采用了面向服务的架构（Service-oriented architecture SOA），通过调度中心快速实现了资源的弹性化扩展部署，其支持的节点数也扩展至 10000+。

Dubbo 作为国内最早的微服务框架，已然包含微服务连通性、健壮性、可伸缩性、可升级性等优点，但同时也包含一些不足，例如仅支持 JAVA 语言、服务调用强依赖于 RPC 等。Dubbo 为微服务治理提供了一个基础平台，为最早接触微服务的先行者带来了红利。

SpringCloud

Spring Cloud 作为第一代微服务框架代表，由 Pivotal 公司在 2016 年推出 1.0 release 版本，关于 SpringCloud 概念，官方的介绍是：

Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state). Coordination of distributed systems leads to boiler plate patterns, and using Spring Cloud developers can quickly stand up services and applications that implement those patterns.

They will work well in any distributed environment, including the developer’s own laptop, bare metal data centres, and managed platforms such as Cloud Foundry.3

我们可以理解为 Spring Cloud 为开发人员提供了快速构建分布式系统中一些常见模式的工具，例如配置管理、服务发现、熔断、智能路由、微代理、控制总线、一次性令牌、全局锁、领导选举、分布式会话、集群状态。Spring Cloud 为每一个微服务节点提供开发模板，以便业务人员可以按照模板实现应用的快速开发。Spring Cloud 可在任何分布式环境中工作，包括开发者自己的设备、裸金属机器数据中心、托管云平台（例如 Cloud Foundry）等。

Spring Cloud 是在 Spring Boot 的基础上构建微服务的，两者是紧密相连的，Spring Boot 为 Spring 的一套快速配置脚手架。通过 Spring Boot，开发者可以专注于快速且方便地开发单个个体微服务，通过 Spring Cloud，开发者可以全局地对微服务进行协调治理。

图 0.4 为 Spring Cloud 官方提供的微服务架构图：

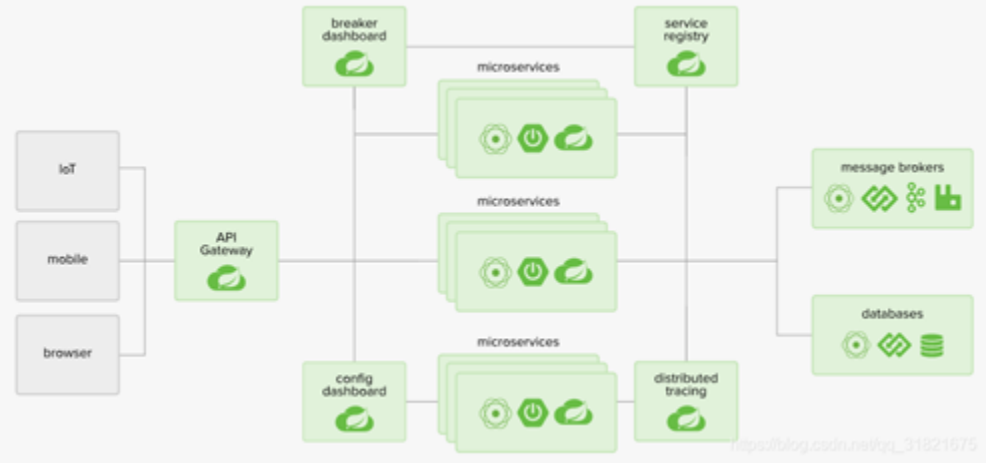

图 0.4 Spring Cloud 微服务架构图

上图可以看出 Spring Cloud 为各个微服务提供了许多重要功能，其中包括 API 网关、配置中心、熔断服务、服务注册、分布式追踪、数据库等。Spring Cloud 由许多子项目组成，各个子项目有其不同的发展，关于 Spring Cloud 组件详细信息可参考官方文档4。

3 https://spring.io/projects/spring-cloud
4 https://spring.io/projects/spring-cloud

第二代微服务治理框架：服务网格

第二代微服务治理框架为服务网格，其主要采用代理模式（Sidecar[5]），即通过在服务旁部署 Sidecar 实现对微服务的治理，且对服务透明。这种代理模式实现了对业务的零侵入并且避免了编程语言的限制，完美解决了第一代微服务治理框架中遗留的诸多问题。由此可见服务网格将成为未来微服务发展的一大趋势，鉴于此，我们将在附录 203 服务网格进行更为详细的介绍。

目前较为流行的服务网格有 Istio、Linkerd、Envoy、NginMesh、Conduit 等，其中 Istio 凭借着社区热度以及大厂联合开发投入成为了服务网格的业界标准。

小结

随着技术的不断发展，单体应用架构已然不能满足企业日益增长和不断变化的需求，微服务的出现成为必然，而基于云的特性，未来的微服务一定是云原生化的，本章我们为各位读者介绍了微服务相关的背景、概念及主流的微服务治理框架，希望可为各位读者带来帮助。

5 一种单节点、多容器的应用设计形式。Sidecar 主张以额外的容器来扩展或增强主容器，而这个额外的容器被称为 Sidecar 容器