Table of Contents

序

// TODO

1 微服务架构概述

微服务架构是当前软件开发领域的一个技术热点。笔者相信，大家对微服务这个名词不会感到陌生。那么究竟什么是微服务？微服务解决了哪些问题？微服务有哪些特点？诸多问题，本章将为您一一解答。

1.1 单体应用架构存在的问题

一个归档包包含了应用所有功能的应用程序，我们通常称之为单体应用。架构单体应用的方法，我们称之为单体架构。

我们以电影售票系统为例，如果采用单体架构，它的架构图如图所示。

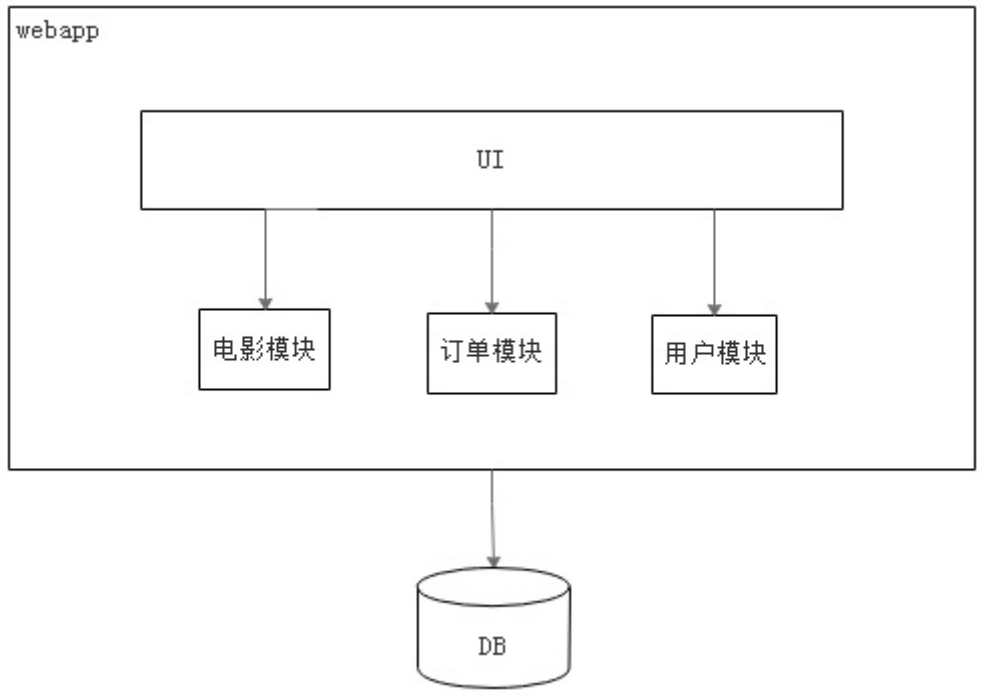

图1-1 单体架构示意图

我们可以看到，UI和若干业务模块都被打包在一个war包中，该war包包含了整个系统所有的功能。尽管应用本身也进行了模块化，但是它最终还是会打包并部署为一个单体应用。

单体应用比较容易部署、测试，在项目的初期，单体应用都可以很好地运行。然而随着项目业务的不断扩张，需求的不断增加，单体架构的优势已经逐渐无法适应互联网时代的需求。例如，随着需求的不断增加，越来越多的人加入开发团队，代码库也在飞速地膨胀。在这种情况下，单体应用会变得越来越臃肿，可维护性、灵活性降低，而维护的成本却不断地在增加。

- 技术债务

随着时间推移、需求变更和人员更迭，形成应用程序的技术债务，并且越积越多。“不坏不修（Not broken，don’t fix）”，这在软件开发中非常常见，在单体应用中这种思想更甚。已使用的系统设计或代码难以修改，因为应用程序的其他模块可能会以意料之外的方式使用它。

- 复杂性较高

以笔者经手的一个百万行级别的单体应用为例，整个项目非常复杂，包含的模块非常多，模块的边界模糊，依赖关系不清晰，多处存在循环依赖，代码质量还比较差，共有9000多个方法，其中只有2000个方法有注释。每次修改代码都心惊胆战，甚至改一个BUG还会造成隐含的缺陷。

- 交付周期长

随着应用功能的越来越多，构建和部署的时间也会增加。每次功能的变更或者缺陷的修复都会导致我们需要重新部署整个应用。这种部署方式耗时长、影响的范围大、风险高，因此，单体项目部署频率较低。而部署频率低又导致了两次发布之间有大量的功能或者缺陷需要变更，出错概率比较高。

- 扩展能力受限

单体应用作为一个整体部署，无法根据业务模块的特点进行伸缩，而只能作为一个整体进行扩展。例如：应用中有的模块是计算密集型的，它需要更多的CPU；有的模块是IO密集型的，需要更大的内存。由于这些模块部署在一起，我们不得不在硬件的选择上做出妥协。

- 妨碍持续交付

单体应用往往会比较大，构建和部署时间也相应的比较长，并且随着应用的扩展，启动时间会变得更长。这不利于频繁部署，阻碍持续交付。在移动应用开发中，这个问题会显得尤为严重；

- 阻碍技术创新

单体应用往往使用统一的技术平台或方案解决所有问题，每个团队成员都必须使用相同的开发语言及开发框架。然而，技术是不断在发展的，近几年也不断有开发效率更高、性能更强大的技术出现。如果想要尝试使用引入新的框架或技术会非常困难。例如：假设之前使用的是Struts2代码构建了一个200万行的单体应用，如果现在想要使用Spring MVC，切换成本、风险都是非常高的；

综上，随着业务需求的发展，功能的不断增加，单体架构很难满足业务快速变化的需要。一方面，代码的可维护性、灵活性、可扩展性在降低；另一方面运维、测试的成本在增加。如何解决单体架构存在的问题呢？

1.2.如何解决单体应用架构存在的问题

由上文我们看到，单体应用架构存在很多的问题。那么如何解决这些问题呢？如果有一种架构方式，能够做到以下几点：

- 能够降低应用的复杂性，让代码维护不是那么的困难；
- 同时对于局部的修改，部署相应的模块即可；
- 系统启动时间快；
- 能够根据业务的特点进行伸缩；
- 不绑定技术栈。

微服务就是这样的一种架构风格。下面我们来看下微服务架构究竟是什么。

1.3 什么是微服务

微服务是当前非常热门的话题，它在各种演讲、书籍的出镜率非常高。到底什么是微服务呢？

从业界的讨论看，微服务本身并没有一个严格的定义。大神Martin Fowler在他的博客中是这样描述微服务的，比较。

> 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.

翻译一下：简而言之，微服务架构风格这种开发方法，是以开发一组小型服务的方式来开发一个独立的应用系统的。其中每个小型服务都运行在自己的进程中，并经常采用HTTP资源API这样轻量的机制来相互通信。这些服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署。这些微服务可以使用不同的语言来编写，并且可以使用不同的数据存储技术。对这些微服务我们仅做最低限度的集中管理。

从中我们可以看到，微服务架构大致具备以下特点：

1. 每个微服务可独立运行在自己的进程里；
2. 一系列独立运行的微服务共同构建起了整个系统；
3. 每个服务为独立的业务开发，一个微服务一般完成某个特定的功能，比如：订单管理、用户管理等；
4. 微服务之间通过一些轻量的通信机制进行通信，例如通过REST API或者RPC的方式进行调用。

我们还以电影收票系统为例，如使用微服务架构，架构图如图1-2所示。

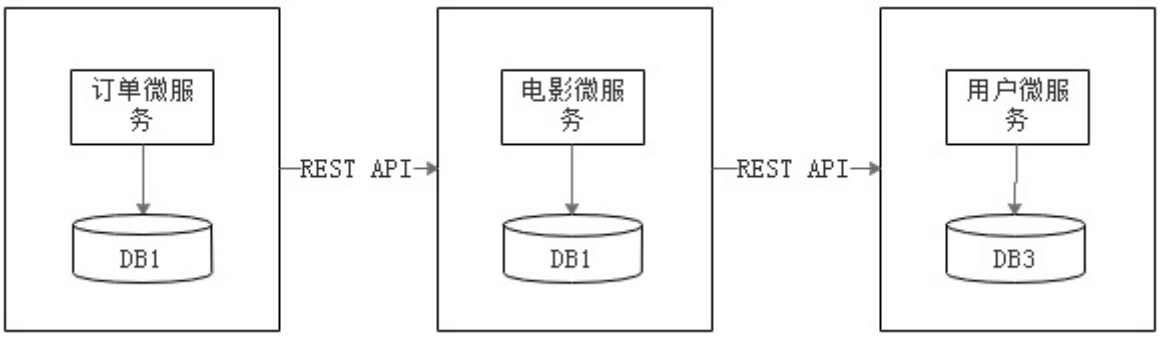

图1-2 微服务架构示意图

我们可以看到，我们将整个应用分解为3个微服务。每个微服务都运行在自己的进程中，都可以独立地运行；每个微服务分别有自己的数据库；微服务之间以REST API进行通信。

1.4 微服务架构的优点与挑战

微服务架构相对单体架构有着显著的优点。同时，微服务也存在着一定的缺点。我们先来分析一下使用微服务具有哪些优点。

微服务架构的优点

- 易于开发和维护

单个微服务只关注一个业务功能，所以业务清晰、代码量相对较小。开发和维护单个微服务相对来说是比较简单的；而整个应用是由若干个微服务构建而成，所以整个应用也会处于可控状态。

- 启动较快

对于单个微服务，启动是比较快的。

- 局部修改容易部署

单体应用只要有修改，就得重新部署整个应用，微服务则解决了这样的问题。多数情况下，对某个微服务进行了修改，只需要重新部署这个服务即可。

- 技术栈不受限

在微服务中，我们可以根据项目业务及团队的情况，选择想要的技术栈。例如某些服务使用了关系型数据库MySQL；而某些微服务有图形计算的需求，我们可以选择Neo4J；甚至可以根据需要，部分微服务使用Java开发，部分微服务使用NodeJS进行开发。

- 按需伸缩

由于微服务是细粒度的，我们可以按照需要，实现细粒度的扩展。例如：系统中的服务A发生了瓶颈，那我们可以根据需要扩展该服务，即可实现对整个应用的扩展。

通过以上的分析，我们可以发现，单体架构的缺点，恰恰是微服务的优点。这些优势使得微服务看起来简直完美。然而完美的东西是不存在的，银弹也不存在，微服务也存在不足之处。下面我们来讨论使用微服务为我们带来的挑战。

微服务架构的挑战

- 运维要求较高

更多的服务意味着更多的运维投入。在单体架构中，只需要保证一个应用的正常运行；而在微服务中，需要保证几十甚至几百个服务的正常运行与协作，这给项目的运维带来了很大的挑战。

- 分布式的复杂性

使用微服务构建的是分布式系统。对于一个分布式系统，系统容错、网络延迟、分布式事务、异步请求等问题都给我们带来了很大的挑战。

- 接口调整成本高

微服务之间通过接口进行通信。如果修改某一个服务的接口，可能使用了该接口的服务都需要做调整。

- 重复劳动

在很多服务中可能都会使用到同一个功能，而这一功能点并没有大到抽离一个微服务的程度，这个时候可能不同的服务团队都会开发这一功能，从而导致代码重复。尽管我们可以在服务之间使用共享库（例如可以将公共部分封装成一个公共组件，然后需要该功能的微服务引用该组件即可），但是在多语言环境下就不一定行得通了。

总而言之，尽管微服务有很多吸引人的地方，但是它对开发、运维都带来了一些新的挑战。下面我们来看看微服务架构的设计原则有哪些，应该怎么样去合理地架构微服务。

1.5 微服务设计原则

和数据库设计中的N范式一样，微服务也有一定的设计原则。这些原则指导大家更合理地架构微服务。

- 单一职责原则

单一职责原则相信大家并不陌生。它指的是一个单元（类、方法或者服务），应该只有一个职责。每个单元只负责整个系统功能的一个单独的、有界限的一部分。单一职责原则可以帮助我们更优雅地开发，并且让交付更加敏捷。

单一职责原则是SOLID原则之一。对SOLID原则感兴趣的读者可以前往 https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)) 扩展阅读。

- 服务自治原则

服务自治，是指每个服务应该具备独立的业务能力、依赖与运行环境，服务可以独立地配置、更新和管理。我们知道，在微服务架构中，服务是一个独立的业务单元，应该与其他服务高度解耦。每个服务从开发、测试、构建、部署，都应当可以独立运行，而不应该依赖其他的服务。

- 轻量级通信原则

服务之间应该通过轻量级的通信机制以实现交互。轻量级通信机制需要具备两点：首先是体量较轻，其次这些通信机制应该是跨语言、跨平台的。目前我们所熟悉的REST协议，就是一个典型的“轻量级通信机制”；而例如Java的RMI则协议不是很符合我们的要求，因为它绑定了Java的技术栈。

微服务中常用的协议有：REST、AMQP、STOMP、MQTT等。

- 接口明确原则

每个服务的对外接口应该明确定义，并尽量保持不变。这与前文我们讨论的使用微服务存在“接口调整成本高”的缺点是对应的。

1.6 如何实现微服务？

至此，我们已经知道了微服务是什么，以及其存在的优缺点，并总结出了一些指导性的原则，为我们如何架构微服务提供了支持。

下面我们来探讨一下，如何将微服务的理论付诸实践。

1.6.1 微服务技术选型有哪些

微服务应用的交付相对单体应用的交付要复杂很多。所以，我们不仅需要开发框架的支持，还需要一些自动化的部署工具，以及IaaS、PaaS（或CaaS）的支持。

下面从开发、持续集成、运行平台三个维度考虑挑选技术选型。

开发框架的选择

对于开发框架，我们可使用Spring Cloud作为微服务的开发框架。

首先，Spring Cloud提供了开箱即用的生产属性，可大大提升开发效率。再者，Spring Cloud的文档丰富，社区也比较热，遇到问题比较容易获得支持。更为可贵的是，Spring Cloud为微服务架构提供了完整的解决方案。

当然，实现微服务也可以使用其他的开发框架或者解决方案，例如：Dubbo、Dropwizard、armada等。

持续集成

持续集成是一个比较老生常谈的话题。持续集成工具有Jenkins（原Hudson）、TeamCity、Atlassian Bamboo等。这些工具基本都能满足我们的需要，基于免费、开源以及市场占有率等考虑，笔者不能免俗地使用了Jenkins作为持续集成的工具。

运行平台

出于轻量、灵活、应用支撑等方面的考虑，我们将应用部署在Docker上。当然，微服务并不绑定平台，将微服务部署在例如阿里云，AWS，或基于OpenStack、CloudStack构建的公有云、私有云上也是可以的。

1.6.2 微服务架构图及常用组件

在进入实战之前，我们首先展望一下，看一下一个微服务架构的最终架构图是什么样子的。

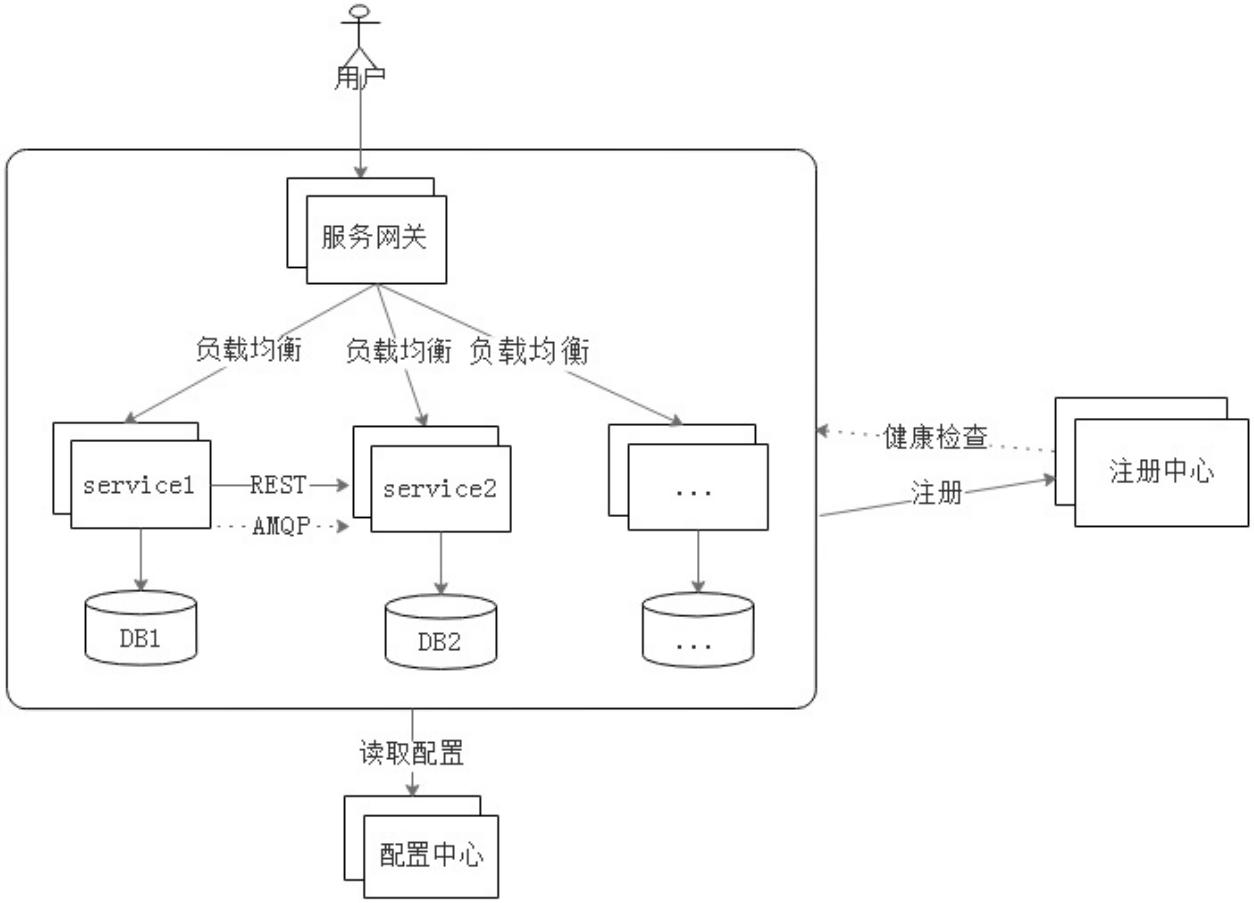

图1-3 微服务架构图

图1-3不严谨地表示了一个微服务应用的架构（之所以说不严谨，是因为配置中心可以注册到注册中心上；而注册中心也可以从配置中心读取配置信息）。

由图我们可以看到，除了业务所需要的service1、service2等业务相关的微服务以外，还有注册中心、服务网关、配置中心等组件。

配置中心大家很好理解，是一个用于管理配置的微服务。可注册中心、服务网关是什么？作用又是什么？我们将在实战的过程中进行具体的讲解。

2.1 Spring Cloud是什么

尽管Spring Cloud中带有Cloud的字样，但是它并不是云计算的解决方案，而是在Spring Boot的基础上构建的、用于快速构建分布式系统中的一些通用模式的工具集。