微服务架构与实践

lainding

Published
with GitBook

目錄

微服务架构与实践

王磊 著

- 基础篇
- 实践篇
- 进阶篇

第1章 单块架构及其面临的挑战

三层架构通常包括表示层、业务逻辑层以及数据访问层。

三层架构的出现，解决了系统间调用复杂、职责不清的问题，也有效降低了层与层之间的依赖关系，称为软件架构的经典模式之一。

虽然三层架构将系统在逻辑上分成了三层，但它并不是物理上的分层。也就是说，对不同层的代码而言，经历编译、打包、部署后，所有的代码最终是运行在同一个进程中。

1.1 三层应用架构

1.1.1 三层应用架构的发展

在软件架构模式领域，经过多年的发展，也有了层的概念：

- 层能够被单独构造。
- 每层具有区别于其他层的显著特点。
- 层与层之间能够互相连接，互相支撑，互相作用，互相协作，从而构成一个整体。
- 层的内部可以被替换成其他可工作的部分，但对整体的影响不大。

1.1.2 什么是三层架构

三层架构通常包括表示层、业务逻辑层以及数据访问层。

- 表示层 表示层部分通常指当用户使用应用程序时，看见的、听见的、输入的或者交互的部分。
- 业务逻辑层 业务逻辑部分是根据用户输入的信息，进行逻辑计算或者业务处理的部分。
- 数据访问层 在用户同应用程序交互的过程中，会产生数据。这类数据需要通过某种机制被有效地保存，并在将来能够被重复使用，或者提供给其他系统。

1.1.3 三层架构的优势

三层架构的出现，一方面是为解决应用程序中代码间调用复杂、代码职责不清的问题。其通过在各层间定义接口，并将接口与实现分离，可以很容易地用不同的实现来替换原有层次的实现，从而有效降低层与层之间的依赖。这种方式不仅有利于帮助团队理解整个应用架构，降低后期维护成本，同时也有利于指定整个应用程序架构的标准。

另一方面，三层架构的出现从某种程度上解决了企业内部如何有效根据技能调配人员，提高生产效率的问题。

三层架构的出现不仅标准化了复杂系统的逻辑划分，更帮助企业解决了如何有效行程技术人员组织架构的问题，因此在很长一段时间里，它一直是软件架构的经典模式之一。

1.2 单块架构

1.2.1 什么是单块架构

虽然软件的三层架构帮助我们将应用在逻辑上分成了三层，但并不是物理上的分层。对于这种功能几种、代码和数据中心化、一个发布包、部署后运行在同一进程的应用程序，我们通常称之为单块架构应用。

1.2.2 单块架构的优势

- 易于开发 对单块架构的应用程序而言，开发方式相对简单。
- 易于测试 单块架构应用程序也非常容易被测试，因为所有的功能都运行在一个进程中，启动集成开发环境或者将发布包部署到某一环境，一旦启动该进程，就可以立即开始系统测试或功能测试。
- 易于部署 对单块架构的应用程序而言，部署也比较容易。
- 易于水平伸缩 对单块架构的应用程序而言，水平伸缩也比较容易。

1.2.3 单块架构面临的挑战

- 维护成本增加
- 持续交付周期长
- 新人培养周期长
- 技术选型成本高

  > ThoughtWorks的技术雷达

- 可扩展性差
  - 垂直扩展（Vertical Scaling或Scale-up）

    > 在垂直扩展模型中，想要增加系统负荷就意味着要在系统现有的部件上下工夫，即通过提高系统部件的能力来实现。

  - 水平扩展（Horizongtal Scaling或Scale-out）

> 在水平扩展模型中，我们不是通过增加单个系统成员的负荷而是简单的通过增加更多的系统成员来实现。

水平扩展通常的做法是建立一个集群，通过在集群中不断添加新节点，然后借助前段的负载均衡器，将用户的请求按照某种算法，譬如轮转法、散列法或者最小连接法等合理地将请求分配到不同的节点上。

- 构建全功能团队难

第2章 微服务架构综述

微服务架构模式（Microservice Architect Pattern）是近两年在软件架构模式领域出现的一个新名词。

本章包括：

- 什么是为服务架构
- 微服务架构与SOA
- 微服务的本质
- 微服务不是银弹

2.1 什么是微服务架构

从业界的讨论来看，微服务本身并没有一个严格的定义。不过，ThoughtWorks的首席科学家（Martin Flowler）的描述更加通俗易懂

> 微服务架构是一种架构模式，它提倡将单一应用程序划分成一组小的服务，服务之间互相协调、互相配合，为用户提供最终价值。每个服务运行在其独立的进程中，服务于服务间采用轻量级的通信机制互相沟通（通常是基于HTTP的RESTful API）。每个服务都围绕着具体业务进行构建，并且能够被独立地部署到生产环境、类生产环境等。另外，应尽量避免统一的、集中式的服务管理机制，对具体的一个服务而言，应根据业务上下文，选择合适的语言、工具对其进行构建。
>
> ————摘自 马丁·福勒先生的博客
> (http://martinfowler.com/articles/microservices.html)

2.1.1 多微才够微

微服务架构通过对特定业务领域的分析与建模，将复杂的应用分解成小而专一、耦合度低并且高度自治的一组服务，每个服务都是很小的应用。

- 代码行数 对于实现统一的功能，选择不同的语言，代码的行数会千差万别。因此，代码行数这种量化的数字显然无法成为横向微服务是否够“微”的决定因素。
- 重写时间 多长时间能够重写该服务也不能作为衡量其是否“微”的重要因素。
- 团队觉得好才是真的好 微服务的“微”并不是一个真正可衡量、看得见、摸得着的微。这个“微”所表达的，是一种设计思想和指导方针，是需要团队或者组织共同努力找到一个平衡点。 所以，微服务到底有多微，是个仁者见仁，智者见智的问题，最重要的是团队觉得合适。但注意，如果达成“团队觉得合适”的结论，至少还应遵循以下两个基本前提。
  - 业务独立性 首先，应保证微服务是具有业务独立性的单元，并不能只是为了微而微。关于如何拍段业务的独立性，也有不同的考量。
  - 团队自主性 其次，考虑到团队的沟通及协作成本，一般不建议超过10人。当团队超过10人，在沟通，协作上锁耗费的成本会显著增加，这也是大部分敏捷实践里提倡的。

2.1.2 单一职责

`高内聚，低耦合` ，所谓高内聚，是一个模块内各个元素彼此结合的紧密程度高。低耦合，是指对于一个完整的系统，模块与模块之间，尽可能独立存在。

在面向对象的设计中，更是有放之四海而皆准的“ `SOLID原则` ”。SOLID原则中的 `S` 表示的是SRP（Single Responsibility Principle，单一职责原则）：即一个对象应该只有一个发生变化的原因，如果一个对象可被多个原因改变，那么久说明这个对象承担了多个职责。

对于每个服务而言，我们希望它处理的业务逻辑能够单一，在服务架构层面遵循单一职责原则。也就是说，微服务架构中的每个服务，都是具有业务逻辑的，符合高内聚、低耦合原则以及单一职责原则的单元，不同的服务通过“管道”的方式灵活组合，从而构建出庞大的系统。

2.1.3 轻量级通信

服务之间应通过轻量级的通信机制，实现彼此之间的互通互联，互相协作。所谓轻量级通信机制，通常指语言无关、平台无关的交互方式。

对于轻量级通信的格式而言，我们熟悉的 `XML` 或者 `JSON` ，它们的解析和使用基本与语言无关、平台无关。对于轻量级通信而言，通常基于HTTP，能让服务间的通信变得标准化并且无状态化。 `REST` （Representational State Transfer），是实现服务之间互相协作的轻量级通信机制之一。

对于微服务而言通过使用语言无关、平台无关的轻量级通信机制，使服务与服务之间的写作变得更加标准化，也就意味着在保持服务外部通信机制轻量级的情况下，团队可以选择更适合的语言、工具或者平台来开发服务本身。

2.1.4 独立性

独立性是指在应用的交付过程中，开发、测试以及部署的独立。

在传统的单块架构应用中，所有功能都存在于同一个代码库中。当修改了代码库中的某个功能，很容易出现功能之间相互影响的情况。尤其是随着代码量、功能的不断增加，风险也会逐渐增加。

除此之外，当多个特性被不同小组实现完毕，需要经过集成，经过回归测试，团队才有足够的信心，保证功能相互配合、正常工作并且互不影响。因此，测试过程不是一个独立的过程。

当所有测试验证完毕，单块架构应用将被构建成一个部署包，并标记相应的版本。在部署过程中，单块架构部署包将被部署到生产环境，如果其中某个特性存在缺陷，则有可能导致整个部署过程的失败或回滚。

在微服务架构中，每个服务都是一个独立的业务单元，当对某个服务进行改变时，对其他的服务不会产生影响。换句话说，服务和服务之间是独立的。对于每个服务，都有独立的代码库。当对当前服务的代码进行修改后，并不会影响其他服务。从代码库的层面而言，服务与服务是隔离的。

对于每个服务，都有独立的测试机制，并不担心破坏其他功能而需要建立大范围的回归测试。

由于构建包是独立的，部署流程也就能够独立，因此服务能够运行在不同的进程中。

2.1.5 进程隔离

所有功能运行在同一个进程中，也就意味着，当对应用进行部署时，必须停掉当前正在运行的应用，部署完成后，再重新启动进程，无法做到对部署。如果当前某应用中包含定时任务的功能，则要考虑在什么时间窗口适合部署，是否先停掉消息队列或者切断与数据源的联系，以防止数据被读入应用程序内存，但还未处理完，应用就被停止而导致的数据不一致性。

为了提高代码的重用以及可维护性，在应用开发中，我们有时也会将重复的代码提取出来，封装成组件。在传统的单块架构中，组件通常的形态叫共享库。当应用程序在运行期时，所有的组件最终也会被加载到同一个进程中运行。

在微服务架构中，应用程序由多个服务组成，每个服务都是一个具有高度自治的独立业务实体。通常情况下，每个服务都能运行在一个独立的操作系统进程中，这就意味着不同的服务能非常容易的被部署到不同的主机上。作为运行微服务的环境，我们希望它能够保持高度自治性和隔离型。如果多个服务运行在同一个服务器节点上，虽然省去了节点的开销，但是增加了部署和扩展的复杂度。

微服务架构其实是将单一的应用程序划分成一组小的服务，每个服务都是具有业务属性的独立单元，同时能够被独立开发、独立运行、独立测试以及独立部署。

2.2 微服务的诞生背景

2.2.1 互联网行业的快速发展

互联网时代的产品通常有两类特点：需求变化快和用户群体庞大。在这种情况下，如何从系统架构的角度出发，构建理货、易扩展的系统，快速应对需求的变化；同时，随着用户的增加，如何保证系统的可伸缩性、高可用性，成为系统架构面临的挑战。

2.2.2 敏捷、精益方法论的深入人心

精益创业（Lean Startup）通过迭代持续改进，帮助组织分析并建立最小可实行产品（Minimum Viable Product）；

敏捷方法帮助组织消除浪费，通过反馈不断找到正确的方向；

持续交付帮助组织构建更快、更可靠、可频繁发布的交付机制；

云、虚拟化和基础实施自动化（Infrastructure As Code）的使用则极大简化了基础设施的创建、配置以及系统的安装和部署；

DevOps文化的推行更是打破了传统开发与运维之间的壁垒，帮助组织行程全功能化的高效能团队；

2.2.3 单块架构系统面临的挑战

2.2.4 容器虚拟化技术

Docker 是一个开源的应用容器（Linux Container）引擎，允许开发者将他们的应用及依赖包打包到一个可移植的容器中，然后发布到任何装有Docker的Linux机器上。

同传统的虚拟化技术相比，基于容器技术的Docker，不需要额外的hypervisor机制的支持，因此具有更高的性能和效率。另外，Docker的优势也主要体现在以下几个方面：

- 更快速地交付和部署。
- 更轻松的迁移和扩展。
- 更简单地管理。

Docker的出现，有效的解决了微服务架构下，服务粒度细、服务数量多所导致的开发环境搭建、部署以及运维成本高的问题。同时，利用Docker的容器化技术，能够实现在一个节点上运行成百甚至上千的Docker容器，每个容器都能独立运行一个服务，因此极大降低了随着微服务数量增多导致的节点数量增多的成本。

如果说之前的敏捷、精益、持续交付以及DevOps是微服务诞生的催化剂，那Docker的出现，则有效解决了微服务的环境搭建、部署以及运维成本高的问题，为微服务朝大规模应用起到了推波助澜的关键作用。

2.3 微服务架构与SOA

2.3.1 SOA概述

早在1996年，Gartner就提出了面向服务架构（SOA）。SOA阐述了“对于复杂的企业IT系统，应按照不同的、可重用的粒度划分，将功能相关的一组功能提供者组织在一起为消费者提供服务”，其目的是为解决企业内部不同IT资源之间无法互联而导致的信息孤岛问题。

2.3.2 微服务与SOA

实际上，微服务架构并不是一个全新的概念。仔细分析SOA的概念，就会发现，其和我们今天所谈到的微服务思想几乎一致。 相比传统SOA的服务实现方式，微服务更具灵活性、可实施性以及可扩展性，其强调的是一种独立测试、独立部署、独立运行的软件架构模式。

2.4 微服务的本质

微服务的本质特征通常包括如下几个部分：

- 服务作为组件
- 围绕业务组织团队
- 关注产品而非项目
- 技术多样性
- 业务数据独立
- 基础设施自动化
- 演进式架构

2.4.1 服务作为组件