基于微服务的软件架构模式

杜琪 (/u/28d7875c78df) + 关注
2015.12.05 21:32* 字数 3775 阅读 32054 评论 19 喜欢 141 赞赏 3
(/u/28d7875c78df)

今天阅读了两篇关于微服务的文章，总结一些笔记，不敢贸然翻译：一是因为水平不够，翻译的过程会丢掉作者的原意；二是因为技术翻译是一个略微吃力不讨好的活。

微服务（micro services）这个概念不是新概念，很多公司已经在实践了，例如亚马逊、Google、FaceBook，Alibaba。微服务架构模式（Microservices Architecture Pattern）的目的是将大型的、复杂的、长期运行的应用程序构建为一组相互配合的服务，每个服务都可以很容易得局部改良。 Micro这个词意味着每个服务都应该足够小，但是，这里的小不能用代码量来比较，而应该是从业务逻辑上比较——符合SRP原则的才叫微服务。

暂且不讨论大小问题，读者朋友你首先要考虑的是如何解决目前技术团队遇到的开发问题、部署问题。正是在解决这些问题的过程中，才渐渐总结提炼出了微服务架构模式的概念。

微服务跟SOA有什么区别呢，可以把微服务当做去除了ESB的SOA。ESB是SOA架构中的中心总线，设计图形应该是星形的，而微服务是去中心化的分布式软件架构。

接下来会讨论以下几个话题：

1. 应用微服务的动机，跟传统巨石应用的比较
2. 微服务的优点与缺点
3. 应用微服务架构设计时可能遇到的关键问题（内部服务通信、分布式数据管理）

一、巨石（monolith）

web应用程序发展的早期，大部分web工程是将所有的功能模块（service side）打包到一起并放在一个web容器中运行，很多企业的Java应用程序打包为war包。其他语言（Ruby，Python或者C++）写的程序也有类似的问题。

假设你正在构建一个在线商店系统：客户下订单、核对清单和信用卡额度，并将货物运输给客户。很快，你们团队一定能构造出如下图所示的系统。

Fig1- the monolithic architecture

这种将所有功能都部署在一个web容器中运行的系统就叫做巨石型应用。巨石型应用有很多好处：IDE都是为开发单个应用设计的、容易测试——在本地就可以启动完整的系统、容易部署——直接打包为一个完整的包，拷贝到web容器的某个目录下即可运行。

但是，上述的好处是有条件的：应用不那么复杂。对于大规模的复杂应用，巨石型应用会显得特别笨重：要修改一个地方就要将整个应用全部部署（*PS：在不同的场景下优势也变成了劣势*）；编译时间过长；回归测试周期过长；开发效率降低等。另外，巨石应用不利于更新技术框架，除非你愿意将系统全部重写（代价太高你愿意老板也不愿意）。

二、应用拆分

详细一个网站在业务大规模爬升时会发生什么事情？并发度不够？OK，加web服务器。数据库压力过大？OK，买更大更贵的数据库。数据库太贵了？将一个表的数据分开存储，俗称“分库分表”。这些都没有问题，good job。不过，老外的抽象能力比我们强，看下图Fig2。

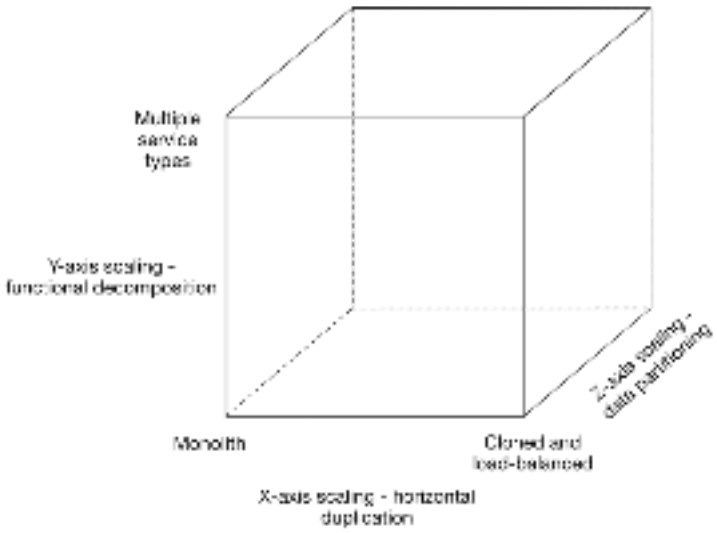

Fig2 - the scale cube

这张图从三个维度概括了一个系统的扩展过程：（1）x轴，水平复制，即在负载均衡服务器后增加多个web服务器；（2）z轴扩展，是对数据库的扩展，即分库分表（分库是将关系紧密的表放在一台数据库服务器上，分表是因为一张表的数据太多，需要将一张表的数据通过hash放在不同的数据库服务器上）；（3）y轴扩展，是功能分解，将不同职能的模块分成不同的服务。从y轴这个方向扩展，才能将巨型应用分解为一组不同的服务，例如订单管理中心、客户信息管理中心、商品管理中心等等。

将系统划分为不同的服务有很多方法：（1）按照用例划分，例如在线商店系统中会划分出一个checkout UI服务，这个服务实现了checkout这个用例；（2）按照资源划分，例如可以划分出一个catlog服务来存储产品目录。

服务划分有两个原则要遵循：（1）每个服务应该尽可能符合单一职责原则——Single Responsible Principle (https://link.jianshu.com?t=http://www.objectmentor.com/resources/articles/srp.pdf)，即每个服务只做一件事，并把这件事做好；（2）参考Unix命令行工具的设计，Unix提供了大量的简单易用的工具，例如grep、cat和find。每个工具都小而美。

最后还要强调：系统分解的目标并不仅仅是搞出一堆很小的服务，这不是目标；真正的目标是解决巨石型应用在业务急剧增长时遇到的问题。

对于上面的例子，按照功能和资源划分后，就形成下面图3的架构图。分解后的微服务架构包含多个前端服务和后端服务。前端服务包括Catalog UI（用于商品搜索和浏览）、Checkout UI（用于实现购物车和下单操作）；后端服务包括一些业务逻辑模块，我们将在巨石应用中的每个服务模块重构为一个单独的服务。这么做有什么问题呢?

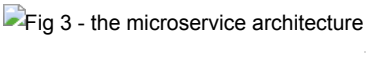

Fig 3 - the microservice architecture

三、微服务架构的优点与缺点

1. 优点

- 每个服务足够内聚，足够小，代码容易理解、开发效率提高
- 服务之间可以独立部署，微服务架构让持续部署成为可能；
- 每个服务可以各自进行x扩展和z扩展，而且，每个服务可以根据自己的需要部署到合适的硬件服务器上；
- 容易扩大开发团队，可以针对每个服务（service）组件开发团队；
- 提高容错性（fault isolation），一个服务的内存泄露并不会让整个系统瘫痪；
- 系统不会被长期限制在某个技术栈上。

2. 缺点

《人月神话》中讲到：没有银弹，意思是只靠一把锤子是盖不起摩天大楼的，要根据业务场景选择设计思路和实现工具。我们看下为了换回上面提到的好处，我们付出（trade）了什么？

- 开发人员要处理分布式系统的复杂性；开发人员要设计服务之间的通信机制，对于需要多个后端服务的user case，要在没有分布式事务的情况下实现代码非常困难；涉及多个服务直接的自动化测试也具备相当的挑战性；
- 服务管理的复杂性，在生产环境中要管理多个不同的服务的实例，这意味着开发团队需要全局统筹（*PS：现在docker的出现适合解决这个问题*）
- 应用微服务架构的时机如何把握？对于业务还没有理清楚、业务数据和处理能力还没有开始爆发式增长之前的创业公司，不需要考虑微服务架构模式，这时候最重要的是快速开发、快速部署、快速试错。

四、微服务架构的关键问题

1. 微服务架构的通信机制

（1）客户端与服务器之间的通信

在巨石型架构下，客户端应用程序（web或者app）通过向服务端发送HTTP请求；但是，在微服务架构下，原来的巨石型服务器被一组微服务替代，这种情况下客户端如何发起请求呢?

如图4中所示，客户端可以向micro service发起RESTful HTTP请求，但是会有这种情况发生：客户端为了完成一个业务逻辑，需要发起多个HTTP请求，从而造成系统的吞吐率下降，再加上无线网络的延迟高，会严重影响客户端的用户体验。

Fig4 - calling services directly

为了解决这个问题，一般会在服务器集群前面再加一个角色：API gateway，由它负责与客户度对接，并将客户端的请求转化成对内部服务的一系列调用。这样做还有个好处是，服务升级不会影响到客户端，只需要修改API gateway即可。加了API gateway之后的系统架构图如图5所示。

Fig5 - API gateway

（2）内部服务之间的通信

内部服务之间的通信方式有两种：基于HTTP协议的同步机制（REST、RPC）；基于消息队列的异步消息处理机制（AMQP-based message broker）。

Dubbo (https://link.jianshu.com?t=http://dubbo.io/)是阿里巴巴开源的分布式服务框架，属于同步调用，当一个系统的服务太多时，需要一个注册中心来处理**服务发现**问题，例如使用ZooKeeper这类配置服务器进行服务的地址管理：服务的发布者要向ZooKeeper发送请求，将自己的服务地址和函数名称等信息记录在案；服务的调用者要知道服务的相关信息，具体的机器地址在ZooKeeper查询得到。这种同步的调用机制足够直观简单，只是没有“订阅——推送”机制。

AMQP-based的代表系统是kafka (https://link.jianshu.com?t=http://kafka.apache.org/)、RabbitMQ (https://link.jianshu.com?t=http://rabbitmq-into-chinese.readthedocs.org/zh_CN/latest/)等。这类分布式消息处理系统将订阅者和消费者解耦合，消息的生产者不需要消费者一直在线；消息的生产者只需要把消息发送给消息代理，因此也不需要服务发现机制。

两种通信机制都有各自的优点和缺点，实际中的系统经常包含两种通信机制。例如，在分布式数据管理中，就需要同时用到同步HTTP机制和异步消息处理机制。

2. 分布式数据管理

（1）处理读请求

在线商店的客户账户有限额，当客户试图下单时，系统必须判断总的订单金额是否超过他的信用卡额度。信用卡额度由CustomerService管理、下订单的操作由OrderService负责，因此Order Service要通过RPC调用向Customer Service请求数据；这种方法能够保

证每次Order Service都获取到准确的额度，单缺点是多一次RPC调用、而且Customer Service必须保持在线。

还有一种处理方式是，在OrderService这边存放一份信用卡额度的副本，这样就不需要实时发起RPC请求，但是还需要一种机制保证——当Customer Service拥有的信用卡额度发生变化时，要及时更新存放在Order Service这边的副本。

（2）处理更新请求

当一份数据位于多个服务上时，必须保证数据的一致性。

- 分布式事务（Distributed transactions）
  使用分布式事务非常直观，即要更新Customer Service上的信用卡额度，就必须同时更新其他服务上的副本，这些操作要么全做要么全不做。使用分布式事务能够保证数据的强一致，但是会降低系统的可用性——所有相关的服务必须始终在线；而且，很多现代的技术栈并不支持事务，例如REST、NoSQL数据库等。
- 基于事件的异步更新（Event-driven asynchronous updates）
  Customer Service中的信用卡额度改变时，它对外发布一个事件到“message broker（消息代理人）”；其他订阅了这个事件的服务受到提示后就更新数据。事件流如图6所示。

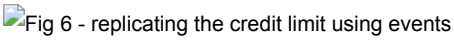

Fig 6 - replicating the credit limit using events

五、重构巨石型应用

在实际工作中，很少有机会参与一个全新的项目，需要处理的差不多都是存在这样那样问题的复杂、大型应用。这时候如何在维护老服务的同时，将系统渐渐重构为微服务架构呢?

1. 不要让事情更坏，有新的需求过来时，如果可以独立开发为一个服务，就单独开发，然后为老服务和新服务直接编写**胶水代码（Glue Code）**——这个过程不容易，但这是分解巨型服务的第一步，如图7所示；

Fig-7 - extracting a service

2. 识别巨石型应用中的可以分离出来当做单独服务的模块，一般适合分离的模块具有如下特点：两个模块对资源的需求是冲突的（一个是CPU密集型、一个是IO密集型）；授权鉴定层也适合单独分离出一个服务。每分离出一个服务，就需要编写对应的胶水代码来与剩下的服务通信，这样，在逐渐演进过程中，就完成了整个系统的架构更

新。

关于重构，有篇文章推荐大家阅读——推倒重来的讲究 (https://link.jianshu.com?t=http://www.luanxiang.org/blog/archives/2176.html)，关于重构有很多可以写的，希望我能快速进步，多写点总结与大家分享。

总结

微服务并不是治百病的良药，也不是什么新的技术，我从中学到的最大的一点就是**scale cube**，从这个坐标轴出发去考虑大规模系统的构建比较容易分析和实践。

参考资料

1. Microservices: Decomposing Applications for Deployability and Scalability (https://link.jianshu.com?t=http://www.infoq.com/articles/microservices-intro)
2. Microservices by Martin Fowler (https://link.jianshu.com?t=http://martinfowler.com/articles/microservices.html)
3. Pattern: Microservices Architecture (https://link.jianshu.com?t=http://microservices.io/patterns/microservices.html)
4. 数据库Sharding的基本思想 (https://link.jianshu.com?t=http://blog.csdn.net/bluishglc/article/details/6161475)