> 主讲老师：Fox
> 有道云笔记地址：https://note.youdao.com/s/LSRtfYCP

1. 什么是微服务架构

2012年，Fred George分享了题为Micro Services Architecture-small，short lived services rather than SOA的演讲。在这次演讲中，他描述了2005—2009年，他和团队成员如何将100万行的传统J2EE程序，通过解耦、自动化验证等实践，逐渐分解成20多个5000行代码的小服务。这是对微服务架构进行定义的最早版本。

从2014年起，微服务架构由Martin Fowler、Adrain Cockcroft、Neal Ford等人接力进行介绍、完善、演进、实践，一直维持着较高的热度，直到现在。关于微服务架构的定义，可以参考Martin Fowler在2014年所写的micro-services文章。在这篇文章里，Martin Fowler对微服务架构进行了定义，内容如下：

微服务架构是一种架构模式，它提倡将原本独立的单体应用，拆分成多个小型服务。这些小型服务各自独立运行，服务与服务间的通信采用轻量级通信机制（一般基于HTTP协议的RESTful API），达到互相协调、互相配合的目的。被拆分后的服务都围绕着具体的业务进行构建，每个服务都能独立地进行开发、部署、扩展。由于相互独立且采用轻量级通信机制，因此各个小型服务能够使用不同的语言开发，也可以使用不同的数据存储技术。

> 英文:https://martinfowler.com/articles/microservices.html
> 中文:http://blog.cuicc.com/blog/2015/07/22/microservices

Martin Fowler主要对微服务架构与单体应用进行比较，并畅想了微服务架构的未来。当时，微服务架构基本上还处于理论阶段。随后的几年间，越来越多的微服务架构解决方案逐渐出现和开源。Java语言下，主流的就是Spring Cloud及其衍生出的一些框架，比如Spring Cloud Alibaba。

图灵电商项目就是基于微服务架构实现的，项目架构图如下图所示：

2. 为什么要使用微服务架构

“为什么要使用微服务？”这个问题其实包含了好几个问题，比如“哪些原因导致系统架构往微服务架构方向上演进？”“微服务架构解决了哪些痛点？”“微服务架构有哪些优点？”

2.1 服务架构的演进

好的架构不是设计出来的，而是演进出来的。

系统立项之初，就想着设计一个大而全的架构，期待着它能够解决各个阶段的各种问题，这是不可能的。因为在初期很难预估后期业务的变化，如果在初期就落地一个大而全的项目，那么人力成本和时间成本都会很高。同时，架构并不是千篇一律的，千万不能在不同的业务和系统中生搬硬套同一个架构。先快速落地，并关注业务的变化和系统的健壮程度，在不同阶段对当前架构所面临的问题进行复盘和处理，选择一个更适合自身的方向进行优化和改进，这才是常规的做法。在每个阶段，找到对应该阶段网站架构所面临的问题，在不断解决这些问题的过程中，系统的架构在不断地朝着正确的方向演进。

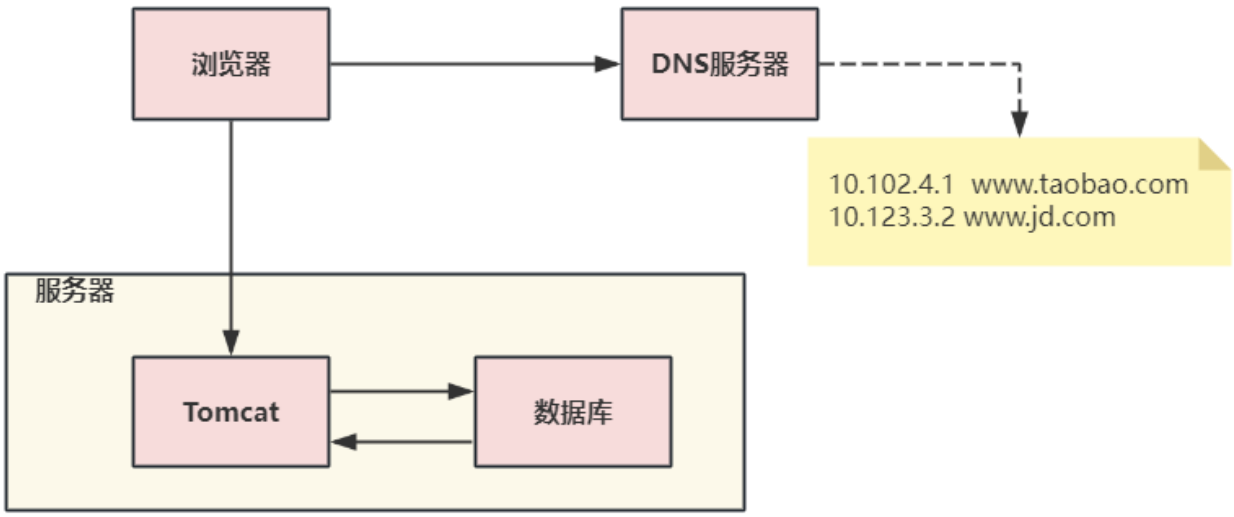
这里我们以淘宝为例，分析淘宝网站从一百个并发到亿级并发情况下服务架构的演进过程。淘宝作为电商平台，其架构经历了多次演进，以支持其快速增长的业务和高并发的用户请求。早期的淘宝采用了单体架构，随着业务的发展，逐步演进到使用分布式缓存、负载均衡、服务化等技术，最终采用了微服务架构。

初始架构：单机架构

在淘宝网站最初时，应用数量与用户数都较少，可以把Tomcat和数据库部署在同一台服务器上。

浏览器往www.taobao.com发起请求时，首先经过DNS服务器（域名系统）把域名转换为实际IP地址10.102.4.1，浏览器转而访问该IP对应的Tomcat。

如下图所示：



新的技术挑战: 随着用户数的增长，Tomcat和数据库之间竞争资源，单机性能不足以支撑业务，架构演进势在必行。

第一次演进：Tomcat与数据库分开部署

第一次演进没有什么特别的，将 Tomcat 和数据库分别独占服务器资源，显著提高两者各自性能。

如下图所示：

新的技术挑战: 随着用户数的增长，并发读写数据库成为瓶颈。

第二次演进：引入本地缓存和分布式缓存

第二次架构演进引入了缓存，在Tomcat服务器上增加本地缓存，并在外部增加分布式缓存，缓存热门商品信息或热门商品的html页面等。

通过缓存能把绝大多数请求在读写数据库前拦截掉，大大降低数据库压力。其中涉及的技术包括：使用memcached作为本地缓存，使用Redis作为分布式缓存，还会涉及缓存一致性、缓存穿透/击穿、缓存雪崩、热点数据集中失效等问题。

演进之后，如下图所示：

新的技术挑战： 缓存抗住了大部分的访问请求，随着用户数的增长，并发压力主要落在单机的Tomcat上，响应逐渐变慢

第三次演进：引入反向代理实现负载均衡

在多台服务器上分别部署Tomcat，使用反向代理软件（Nginx）把请求均匀分发到每个Tomcat中。此处假设Tomcat最多支持100个并发，Nginx最多支持50000个并发，那么理论上Nginx把请求分发到500个Tomcat上，就能抗住50000个并发。

其中涉及的技术包括：Nginx、HAProxy，两者都是工作在网络第七层的反向代理软件，主要支持http协议，还会涉及session共享、文件上传下载的问题。

一起来看看使用反向代理之后的架构图：

新的技术挑战: 反向代理使应用服务器可支持的并发量大大增加，但并发量的增长也意味着更多请求穿透到数据库，单机的数据库最终成为瓶颈

第四次演进：数据库读写分离

把数据库划分为读库和写库，读库可以有多个，通过同步机制把写库的数据同步到读库。对于需要查询最新写入数据场景，可通过在缓存中多写一份，通过缓存获得最新数据。

其中涉及的技术包括：Mycat，它是数据库中间件，可通过它来组织数据库的分离读写和分库分表，客户端通过它来访问下层数据库，还会涉及数据同步，数据一致性的问题。

读写分离之后的架构图：

新的技术挑战: 业务逐渐变多，不同业务之间的访问量差距较大，不同业务直接竞争数据库,相互影响性能

第五次演进：数据库按业务分库

数据库按业务分库，把不同业务的数据保存到不同的数据库中，使业务之间的资源竞争降低，对于访问量大的业务，可以部署更多的服务器来支撑。这样同时会导致跨业务的表无法直接做关联分析，需要通过其他途径来解决。

分库之后的架构图如下所示：

新的技术挑战: 随着用户数的增长，单机的写库会逐渐达到性能瓶颈

第六次演进：把大表拆分为小表

比如针对评论数据，可按照商品ID进行hash，路由到对应的表中存储。

针对支付记录，可按照小时创建表，每个小时表继续拆分为小表，使用用户ID或记录编号来路由数据。

只要实时操作的表数据量足够小，请求能够足够均匀的分发到多台服务器上的小表，那数据库就能通过水平扩展的方式来提高性能。其中前面提到的Mycat也支持在大表拆分为小表情况下的访问控制。

这种做法显著的增加了数据库运维的难度，对DBA的要求较高。数据库设计到这种结构时，已经可以称为分布式数据库。

我们来看拆分小表之后的架构图：

新的技术挑战: 数据库和Tomcat都能够水平扩展，可支撑的并发大幅提高。然而随着用户数的增长，最终单机的Nginx会成为瓶颈

第七次演进：使用LVS或F5来使多个Nginx负载均衡

由于瓶颈在Nginx，因此无法通过两层的Nginx来实现多个Nginx的负载均衡。LVS和F5是工作在网络第四层的负载均衡解决方案，其中LVS是软件，运行在操作系统内核态，可对TCP请求或更高层级的网络协议进行转发，因此支持的协议更丰富，并且性能也远高于Nginx，可假设单机的LVS可支持几十万个并发的请求转发。

F5是一种负载均衡硬件，与LVS提供的能力类似，性能比LVS更高，但价格昂贵。

由于LVS是单机版的软件，若LVS所在服务器宕机则会导致整个后端系统都无法访问，因此需要有备用节点。

架构图如下：

新的技术挑战：由于LVS也是单机的，随着并发数增长到几十万时，LVS服务器最终会达到瓶颈。此时用户数达到千万甚至上亿级别，用户分布在不同的地区，与服务器机房距离不同，导致了访问的延迟会明显不同

第八次演进：通过DNS轮询实现机房间的负载均衡

在DNS服务器中可配置一个域名对应多个IP地址，每个IP地址对应到不同的机房里的虚拟IP。

当用户访问www.taobao.com时，DNS服务器会使用轮询策略或其他策略，来选择某个IP供用户访问。此方式能实现机房间的负载均衡

至此，系统可做到机房级别的水平扩展，千万级到亿级的并发量都可通过增加机房来解决，系统入口处的请求并发量不再是问题。

演进之后的架构图如下：

新的技术挑战: 随着数据的丰富程度和业务的发展，检索、分析等需求越来越丰富，单单依靠数据库无法解决如此丰富的需求

第九次演进：引入NoSQL数据库和搜索引擎等技术

当数据库中的数据多到一定规模时，数据库就不适用于复杂的查询了，往往只能满足普通查询的场景。

对于统计报表场景，在数据量大时不一定能跑出结果，而且在跑复杂查询时会导致其他查询变慢。

对于全文检索、可变数据结构等场景，数据库天生不适用。因此需要针对特定的场景，引入合适的解决方案。

如对于海量文件存储，可通过分布式文件系统HDFS解决，对于key value类型的数据，可通过Redis解决，对于全文检索场景，可通过搜索引擎如ElasticSearch解决，对于多维分析场景，可通过Kylin或Druid等方案解决。

当然，引入更多组件同时会提高系统的复杂度，不同的组件保存的数据需要同步，需要考虑一致性的问题，需要有更多的运维手段来管理这些组件等。

引入NoSQL和搜索引擎的架构图：

新的技术挑战: 引入更多组件解决了丰富的需求，业务维度能够极大扩充，随之而来的是一个应用中包含了太多的业务代码，业务的升级迭代变得困难。

第十次演进：大应用拆分为小应用

为了应对日益复杂的业务场景，通过使用分而治之的手段将整个网站业务拆分成不同的产品线，通过分布式服务来协同工作。
按照业务板块来划分应用代码，使单个应用的职责更清晰，相互之间可以做到独立升级迭代。这时候应用之间可能会涉及到一些公共配置，可以通过分布式配置中心Zookeeper来解决。
架构图如下：

新的技术挑战: 不同应用之间存在共用的模块，由应用单独管理会导致相同代码存在多份，导致公共功能升级时全部应用代码都要跟着升级。

第十一次演进：复用的功能抽离成微服务

如用户管理、订单、支付、鉴权等功能在多个应用中都存在，那么可以把这些功能的代码单独抽取出来形成一个单独的服务来管理，这样的服务就是所谓的微服务。应用通过HTTP、TCP或RPC请求等多种方式来访问服务，每个单独的服务都可以由单独的团队来管理。
此外，可以通过Dubbo、SpringCloud等框架实现服务治理、限流、熔断、降级等功能，提高服务的稳定性和可用性。

微服务架构并不是神话故事中的孙悟空，某一天忽然从石头缝里蹦出来了。微服务架构并不神秘，在“微服务架构”这个概念“火”起来之前，微服务架构叫什么？或者换一个说法：“微服务架构的雏形是什么？”其实，前文中网站架构演进的过程已经给出了答案。**在微服务架构这个概念变得流行之前，技术架构也在不断优化和演进。**
在微服务架构这个概念“火”起来之前，人们会用“分布式服务”或“服务化”来概括这种将大系统拆分为小系统的架构模式，与微服务架构的方式很像，也是对巨无霸的单体应用进行拆分，并结合RPC协议进行服务通信和调用。常见技术有Dubbo、DubboX、CXF、gRPC、HSF、Motan等。随着微服务概念的流行、微服务生态的完善和微服务架构落地规则的细化，现在业内人士都默认将这种架构方式称为微服务架构了。

有人肯定会有疑问，难道只能往微服务架构的方向上演进吗？答案肯定不是，在前面的架构演进图中，演进方向是一条笔直的线。而现实情况中肯定是有不同分支的，系统架构的演进并不是一条笔直的线，根据业务大小和业务侧重点的不同，系统架构在演进时也会朝着不同的方向发展，微服务架构只是众多技术架构中的一个，适合自身业务系统和技术团队的才是最好的架构。而且近些年又出现了Service Mesh、DDD领域驱动、云原生等比较流行的技术方案，今后还会有更加优秀的技术架构和落地方案出现。

2.2 哪些原因导致系统架构往微服务架构的方向演进

前文已经对网站架构演进做了分析。这个演进过程比较常见，不过，在不同的业务和技术团队中，优化和演进肯定不会完全相同，中间可能会有微小的差异。不过总结下来，网站架构演进主要包括三个原因：

1．初始架构

2．系统优化

3．服务化拆分

随着业务的增长和技术团队的完善，系统在逐渐优化。在优化方案应用后，业务量还在不断增长，此时为了应对日益复杂的业务场景，就要使用分而治之的手段进行服务化拆分。将整个网站业务拆分成不同的产品线，通过分布式服务来协同工作。

导致系统架构向微服务架构方向演进的原因：

1．**业务规模的增长**

从业务规模来说，初始的系统架构肯定无法支撑越来越复杂的业务场景，此时就需要进行系统优化，优化手段有缓存、集群、前后端分离、动静分离、读写分离、分布式数据库和分布式文件系统等。若这些系统优化手段都已经用上，还是不能满足业务的成长速度，就要进行业务梳理和系统拆解。

2．**敏捷开发与快速迭代**

从沟通成本和敏捷开发的角度来说，当技术团队中的成员已经有成千上万个，技术小组也有成百上千个的时候，如果还在一个巨无霸的单体项目上开发，都在一个工程里提交代码、修改Bug、切换不同的分支，这就是灾难了。系统的分工不明确、责任不清晰，导致沟通成本高、研发效率低，也无法做到快速迭代。毕竟不是在项目刚开始的阶段，当时可能只有一个技术团队和少量的开发人员。

3．**技术储备完善**

从团队的技术储备来说，项目刚开始的阶段，技术团队人数比较少，团队人员主要是以开发人员为主。但是随着业务规模和企业规模的扩大，在项目架构的不断优化过程中，各种人才的储备已经充足。技术团队也日趋完善，前端技术团队、后端技术团队、测试团队、公共服务团队、DBA团队、运维团队、架构团队等都已经存在，此时再进行业务拆分和架构的完善就有了足够的底层支撑。

4．**微服务架构生态的完善**

从“微服务架构”的发展来说，微服务架构的实践并不是空中楼阁，可以实际落地了。微服务架构已经由最初的理论派逐渐落地和完善，与微服务相关的生态已经建立起来，开源框架和企业内部自研的框架都已投入生产，技术实践也不再是遥不可及的了。比如Spring Cloud框架及与之相关配套的微服务组件为行业提供了一站式的解决方案，解决了很多企业和技术团队关于架构选型和维护方面的困难。

3. 微服务架构的优缺点

3.1 微服务架构的优点

1．更易于开发和维护

因为一个服务只关注一个特定的业务功能，所以它的业务清晰、代码量少。开发的独立和部署的独立都使得开发和维护单个微服务变得简单。

2．快速迭代+灵活

未拆分时，在巨无霸单体项目中开发、提交代码、测试都非常复杂。数不尽的代码分支和代码冲突，还有耗时耗力的测试，都会让人心力交瘁。独立开发与独立部署的微服务，可以更加快速地进行功能迭代。

服务之间的耦合低，甚至可以随时加入一个新的服务或剔除过时的服务，灵活度提升了很多。如果某个功能出现问题，针对性地修改和发版即可，不会像未拆分之前“牵一发而动全身”。

3．系统的伸缩性增强

对高频访问和资源需求高的服务投入更多的资源，比如增加服务器、数据库、带宽等资源的配置。对于低频访问和资源需求相对低的服务不需要投入过多的资源。实现最优的资源利用，提升资源的利用率，并提升系统的伸缩性。

4．技术选型灵活

这一点在微服务架构的定义中已经讲明了，单个服务可以结合具体的业务和团队的特点，选择合适的编程语言和技术栈进行实现。

5．错误隔离

A服务出现了问题或宕机了，这个错误只会影响小范围的相关功能，不会影响整个系统的运行。在微服务架构中，可以使用流量控制、服务熔断、服务降级等手段来对系统进行保护，让局部的错误只影响系统的局部而不是影响系统的全部。

3.2 微服务架构的缺点

凡事都有两面性，微服务架构也不例外。讲完它的优点之后，再来列举一些它的不足之处。

1．落地一个微服务架构项目比较复杂

实施和上线一个微服务架构项目的复杂度很高，工作量很大，要考虑和解决的问题很多。微服务架构实施前的技术选型、微服务组件的搭建和底层支撑、项目拆分时的边界和具体落地的细则、微服务项目的开发和上线、后期的维护等具体的工作都摆在面前，需要一个一个地处理。在落地微服务架构项目时不仅要编码，还要考虑微服务架构的搭建和底层支撑，这件事就像“大兵团作战”，不是一个五人突击队就能够完成任务的。

2．服务依赖和调用链路更复杂

微服务架构中的单个微服务，不可避免地会出现依赖性及由此导致的问题。比如，H服务依赖S服务，S服务依赖A服务，如果A服务在线上出现问题或A服务需要修改部分逻辑，那么S服务和H服务也可能受到牵连，或者级联修改。虽然已经做了服务拆分，影响范围不大，但是这些问题还是存在的。
另外一个问题就是微服务中的调用链路复杂，调用时间相对于单体应用的调用时间肯定是要延长的。微服务在服务调用时难免要建立服务连接，不管是基于HTTP协议还是基于其他的RPC协议，都会难以避免地发生网络损耗，相对于单体应用中的服务调用是同一个项目中的方法调用，更加复杂。

3．数据一致性问题

用前文中的H服务、S服务和A服务举例来说，在调用过程中，如果遇到网络延迟或A服务出现了异常导致数据回滚，但是上游H服务和S服务的数据都已经入库了，就会导致数据不一致的问题。此时就需要做好数据一致性的解决方案，相对于单体应用中的本地事务处理，复杂度又提升了。

4．问题排查的链路加长

前文已经提到了微服务架构项目中的调用链路更复杂，链路复杂和链路的拉长会导致定位线上问题时要排查的地方增加，出现了处理一个问题要查看和定位多个服务的情况。

5．学习成本高

对于开发人员来说，微服务架构的学习和上手比较难。不像学习某一个技术栈，如想要学习和上手Spring Boot技术栈，看教程后动手做几个功能和项目也就学会了。在学习微服务架构时，需要学习很多内容，包括理念、组成部分、各个组件的功能与使用等，都需要理解，还要动手搭建和整合各个微服务组件，否则很难完完整整地掌握。到了具体编码和实战的过程中，又有很多的难点要克服。

4. 什么情况下可以考虑使用微服务架构

微服务架构最大的难点就是复杂度提升了很多个量级，并不是一个开发人员或一个小型的开发团队能够解决和应对的。就像在现实世界中，人们都听过的一句话：“贵一点的东西除价格高外，其他的都好。”拿这句话来做一个类比：微服务架构、服务网格等架构模式，除复杂度高外，其他的也都还好。
微服务架构是一个很优秀的架构模式，能够解决项目开发中的一些痛点，但是想要落地和用好它，需要克服很多的难点。因为它入门难、实践难、部署难、优化难、招人难，总结下来就是技术门槛高。
如果技术人员的水平高且人员充足，那么上述的这些“难”就不复存在了，此时不仅微服务架构不是问题，其他的技术架构在落地和实践时也不是问题。
如果公司的业务量不大，也没有强烈的扩容需求，并且开发团队的项目组就一个，后端开发人员也就几个人，那么此时就不适合去尝试微服务架构了，针对公司系统的问题做针对性的优化即可，切不可做“大炮轰蚊子”这种傻事。
如果公司的业务量增长到一定程度，并且技术团队的人员也充足，那么此时可以考虑尝试微服务架构。如果做完技术评估觉得微服务架构非常契合当前的业务和开发人员，那么进行微服务架构的落地就再适合不过了。

什么情况下可以考虑使用微服务架构?

1. 已经对当前的系统使用了很多优化手段，微服务架构是架构模式，并不是一种具体的系统优化手段。
2. 做完技术评估后，得出的结论是微服务架构能够给技术团队和业务扩展带来正向的影响。
3. 最重要的一点是技术团队的人员齐整、技术支撑足够。

如果以上三点都能够满足，再考虑使用微服务架构，进行实际的架构升级和功能开发。不要为了微服务而使用微服务，要根据自身业务和技术团队来考量是否适合使用这种架构、是否有足够的技术支撑来解决服务化过程中出现的问题。如果不适合，那么最终结果可能就是“大炮轰蚊子”。没有足够的技术沉淀和技术人员来做支撑，反而会对系统的开发和维护造成适得其反的效果。