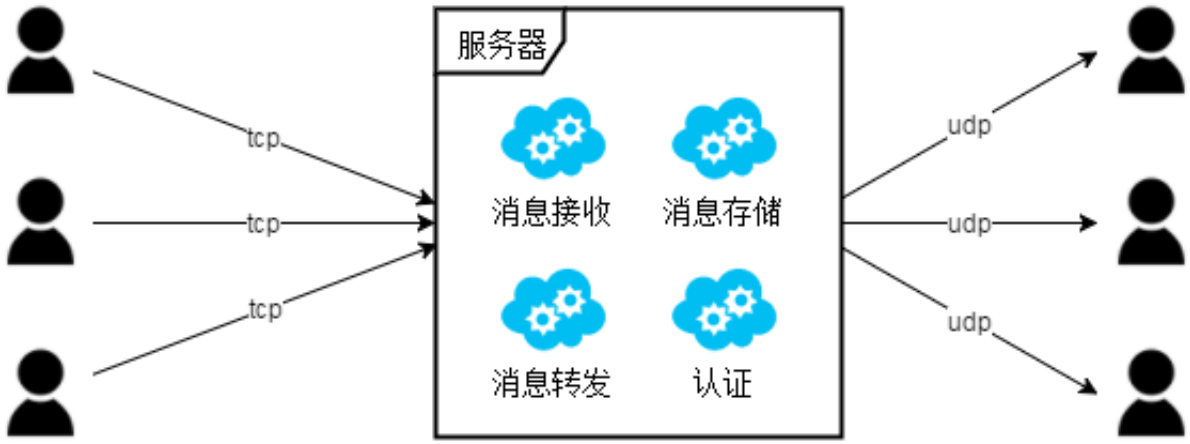
背景：虚拟化技术

虚拟化技术从重量级到轻量级可以分为以下几种：
1. 虚拟机，目标是建立一个可以用来执行整套操作系统的沙盒独立执行环境。典型代表如VirtualBox、VMWare等。
2. 容器技术，将一个应用程序所需的相关程序代码、相关库、环境配置文件打包起来建立沙盒执行环境。典型代表如Docker、LXC。
3. 内核级别的隔离，这种隔离不需要下载额外的软件，直接使用内核提供的隔离来执行程序，程序运行的环境与宿主机完全一致但是不会对宿主机造成任何影响。典型代表如Windows上的Sandboxie以及Linux上的Namespace。

微服务架构

动机

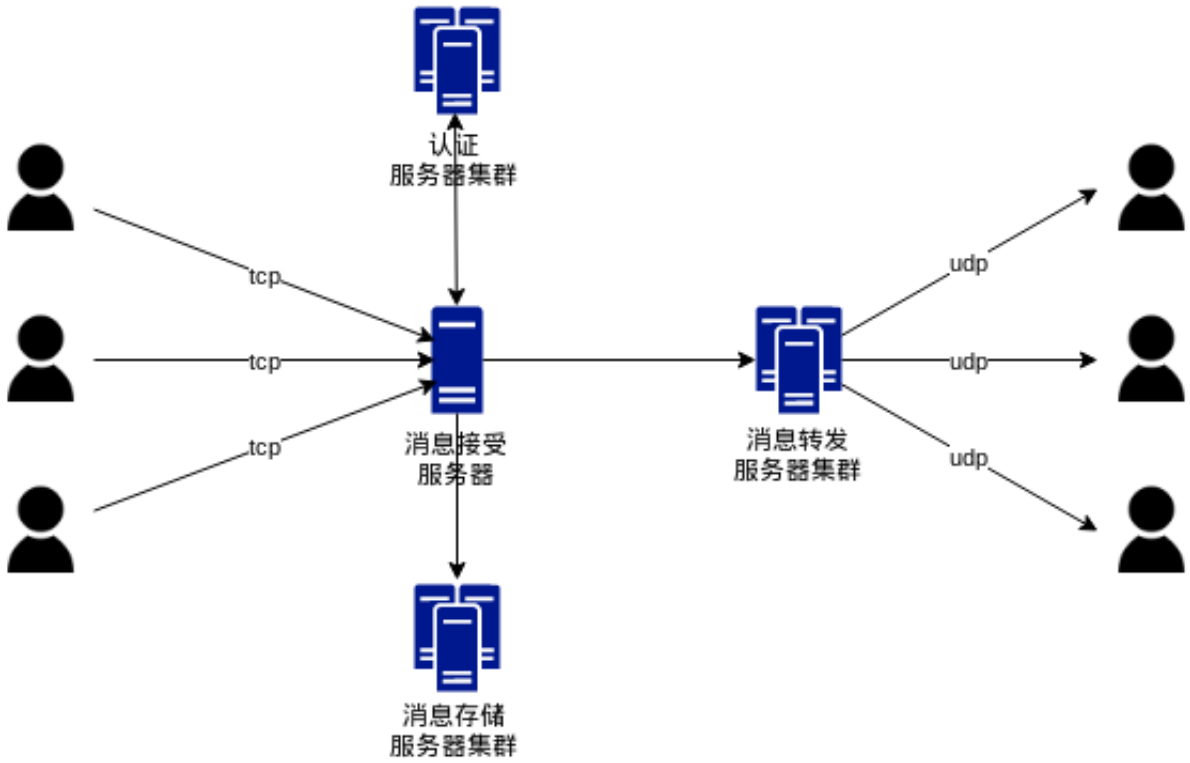


如上图所示，这是我们最开始的架构图，可以看到，所有的功能或者服务都写在了一个应用里。这样的架构有很多好处：IDE都是为开发单个应用设计的、容易测试——在本地就可以启动完整的系统、容易部署——直接打包为一个完整的Jar包，拷贝到任意一个地方执行即可。

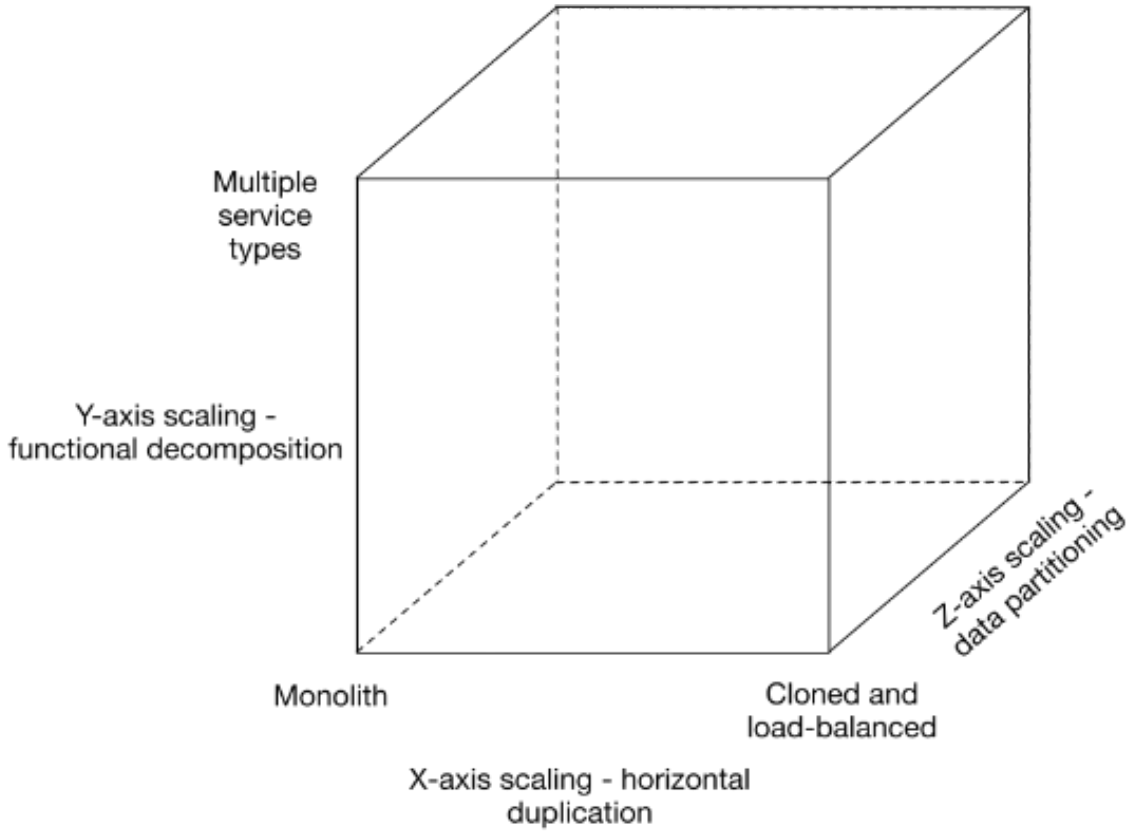
但是，上述的好处是有条件的：应用不那么复杂。对于大规模的复杂应用，这样的架构会显得特别笨重：要修改一个地方就要将整个应用全部部署；编译时间过长；回归测试周期过长；开发效率降低等。另外，这样的架构也不利于更新技术框架，除非你愿意将系统全部重写。

除此之外，还不利于扩展，可以看到，所有的功能都打包到了一个服务器里，当客户很多的时候，我们也不能很好的自由扩展，如只增加消息转发的服务器的数量。

解决方案



上图是我们现在的架构图。当业务规模大规模爬升的时候，我们可以很好的应付各种情况，只需要增加相应的服务器即可。老外的抽象能力比我们强，他们给出了下面这样一张图：



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，例如用户认证，消息转发，消息存储等等。

服务划分的时候有两个原则要遵循：（1）每个服务应该尽可能符合单一职责原则——Single Responsible Principle，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。

最后还要强调的是：系统分解的目标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型应用在业务急剧增长时遇到的问题。

优点和缺点

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。

关键问题

微服务架构的通信机制

1. 客户端与服务器之间的通信
   参考QQ架构，客户端与服务端通过TCP保持连接，认证服务器负责维护所有的在线用户列表。
   消息转发通过UDP连接，客户端接收到消息后需要通过UDP发回一条ACK信息，否则该消息会被服务器重发
2. 内部服务之间的通信。
   通过Java RMI通信，有一个注册中心，所有的服务之间都是相互独立的。

分布式数据管理

通过分布式事务处理。

Docker在微服务系统中所扮演的角色

在Docker出现之前，虽然我们谈论微服务架构，但是其实是很难实现的。微服务要运

行，首先需要一套执行的环境。这套环境不能对外部有依赖性。同时，执行环境的粒度又必须足够的小，这样才能称之为“微”，否则必然是对资源的巨大浪费。一个微服务可以跑在一台虚拟机上面，但是虚拟机粒度太大，即使最小的虚拟机，也至少也有一个核。同时，虚拟机有没有一套方便的管理机制，能够快速的让这些服务之间能够组合和重构。Docker出现以后，我们看到了微服务的一个非常完美的运行环境。

- 独立性：一个容器就是一个完整的执行环境，不依赖外部任何的东西。
- 细粒度：一台物理机器可以同时运行成百上千个容器。其计算粒度足够的小。
- 快速创建和销毁：容器可以在秒级进行创建和销毁，非常适合废物的快速构建和重组。
- 完善的管理工具：数量众多的容器编排管理工具，能够快速的实现服务的组合和调度。

集群管理

目前比较流行的集群管理工具有：

- Apache的Mesos
- Google的Kubernetes

那么他们有什么不同呢？根据StackOverFlow上的解答，概括如下：

Kubernetes主要针对容器集群，而 Mesos适用于任何的框架和应用，所以Kubernetes可以运行于Mesos上。

- 如果你是一个集群世界的新手，那Kubernetes是一个很棒的开始。它可以用最快的、最简单的、最轻量级的方式来解决你的问题，并帮 助你进行面向集群的开发。它提供了一个高水平的可移植方案，因为很多厂商已经开始支持Kubernetes，例如微软、IBM、Red Hat、CoreOS、MesoSphere、VMWare等。
- 如果你拥有已经存在的工作任务（Hadoop、Spark、Kafka等），那Mesos可以给你提供了一个将不同工作任务相互交错的框架，然后还可以加入一些新的东西，比如Kubernetes应用。
- 如果你想使用的功能Kuberntes还没实现，那Mesos是一个不错的替代品，毕竟它已经成熟。

对于我们来说，我们需要一个轻量级的，能够快速部署的Docker管理工具，故Kubernetes是一个不错的选项。

参考资料

[1]: 云计算的三种服务模式：IaaS，PaaS和SaaS
[2]: 基于微服务的软件架构模式
[3]: 基于逻辑回归模型的中文垃圾信息过滤
[4]: 微服务革命：容器化的应用、数据及其它一切
[5]: 微服务架构模式简介
[6]: DOCKER 容器化技术演进
[7]: PaaS与容器化
[8]: 再谈Docker-微服务的场景化应用
[9]: 初次微服务体验：从Docker容器农场说起