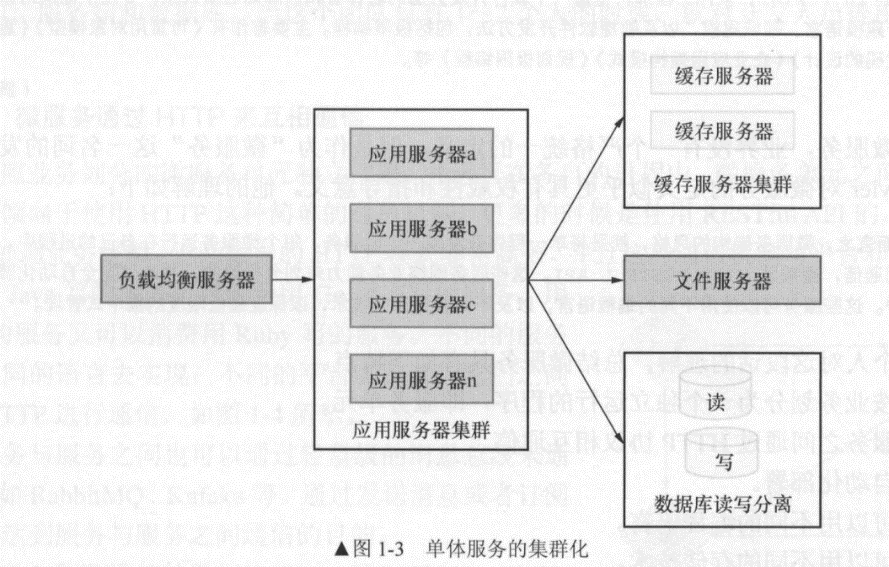
**SpringCloud 学习**

**聂冬佳 2019-03-23**

**1、单体应用的不足**

随着业务的发展，大多数公司会将单体应用进行集群部署，井增加负载均衡服务器（例如Nginx 等〉。另外，还需要增加集群部署的缓存服务器和文件服务器，并将数据库读写分离，以应对用户 量的增加而带来的高并发访问量。此时的系统架构如下图所示 。用负载均衡服务器分发高并发的网络请求，用户的不同访问被分派到的应用服务器，应用服务器的负载不再成为瓶颈，用户量增加时，添加应用服务器即可。通过添加缓存服务器来缓解数据库的数据以及数据库读取数据的压力。大多数的读取操作是由缓存完成的，但是仍然有少数读操作是从数据库读取的，例如缓存失效、实时数据等。当有大量的读写操作时，将数据库进行读写分离是一个不错的选择，例如 MySQL 的主从热备份，通过相关配置可 以将主数据库服务器的数据同步到从数据库服务器 ， 实现数据库的读写分离 ， 读写分离能够改善数据库的负载能力。这种架构有一定的处理高并发的能力， 也能应对一定复杂的业务需求，改善了系统的性能 ， 但是依然没有改变系统为单体架构的事实 ，此时存在的不足之处如下。



系统仍然为单体应用 ， 大量的业务必然会有大量的代码，代码的可读性和可维护性依

然很差。

面对海量的用户，数据库将会成为瓶颈，解决方案将使用分布式数据库，也就是将数据库进行分库分表 。

持续交付能力差 ，业务越复杂，代码越多 ，修改代码和添加代码所需 的时间越长。新人熟悉代码的时间长、成本高。

**在应用初期，单体应用从成本、开发时间和运维等方面都有明显的优势 。 但是随着业务盐和用户聋的增加，它所暴露出来的缺点也显而易见。单体架构己经不能满足复杂的业务和海茧的用户系统，改变单体架构势在必行。**

**2、微服务的优势**

相对于单体服务来说，微服务具有很多的优势，主要体现在以下方面 。

（1）将一个复杂的业务分解成若干小的业务，每个业务拆分成一个服务，服务的边界明

确，将复杂的问题简单化。服务按照业务拆分 ，编码也是按照业务来拆分，代码的可读性和可扩展性增加。新人加入团队，不需要了解所有的业务代码，只需要了解他所接管的服务的代码，新人学习时间成本减少。

(2）由于微服务系统是分布式系统 ，服务与服务之间没有任何的耦合。随着业务的增加，可以根据业务再拆分服务 ，具有极强的横向扩展能力。随着应用的用户量的增加，井发量增加，可以将微服务集群化部署，从而增加系统的负载能力 。 简而言之，微服务系统的微服务单元具有很强的横向扩展能力 。

(3 ）服务与服务之问通过 HTTP 网络通信协议来通信，单个微服务内部高度祸合，服务与

服务之间完全独立，无调合。这使得微服务可以采用任何的开发语言和技术来实现。开发人员不再被强迫使用公司以前的技术或者已经过时的技术，而是可 以 自由选择最适合业务场景的或者最适合自己的开发语言和技术，提高开发效率、降低开发成本。

(4 ）如果是一个单体的应用，由于业务的复杂性、代码的祸合性，以及可能存在的历史问

题。在重写一个单体应用时，要求重写的应用的人员了解所有的业务，所以重写单体应用是非常困难的，并且重写风险也较高。如果是微服务系统，由于微服务系统是按照业务的进行拆分的，并且有坚实的服务边界，所以重写某个服务就相当于重写某一个业务的代码，非常简单。

( 5 ）微服务的每个服务单元都是独立部署的 ，即独立运行在某个进程里。微服务的修改和部署对其他服务没有影响。试想，假设一个应用只有一个简单的修改，如果是单体架构，需要测试和部署整个应用；而如果采用微服务架构，只需要测试并部署被修改的那个服务，这就大大减少了测试和部署的时间。

( 6 ）微服务在 CAP 理论中采用的是 AP 架构，即 具有高可用和分区容错 的特点 。高可用主要体现在系统 7 x 24 小时不间断的服务，它要求系统有大量的服务器集群，从而提高了系统的负载能力。另外，分区容错也使得系统更加健壮。

3、微服务的不足

凡事都有两面性，微服务也不例外，微服务相对于单体应用来说具有很多的优势，当然也有它的不足 ， 主要体现在如下方面 。

微服务的复杂度 。

分布式事务。

服务的划分。

服务的部署。

1. 分布式事务

微服务架构所设计的系统是分布式系统。分布式系统有一个著名的 CAP 理论，即同时满足“ 一致性”“可用性”和“分区容错”是一件不可能的事。 CAP 理论是由 Eric Brewer在 2000 年 PODC 会议上提出的 ，该理论在两年后被证明成立。 CAP 理论告诉架构师不要妄想设计出同时满足三者的系统 ，应该有所取舍，设计出适合业务的系统。 CAP 理论如图 1-9 所示 。

Consistency：指数据的强一致性。如果写入某个数据成功，之后读取，读到的都是新

写入的数据：如果写入失败，之后读取的都不是写入失败的数据。

Availability：指服务的可用性 。

Partition-tolerance：指分区容错 。

两阶段提交

两阶段提交，将事务分成两部分能够大大提高分布式事务成功的概率。如果在第一阶段都成功了，而执行第二阶段的某一个节点失败，仍然导致数据的不准确，这时一般需要人工去处理，这就是当初在第一步记录日志的原因。另外，如果分布式事务涉及的节点很多，某一个节点的网络出现异常会导致整个事务处于阻塞状态，大大降低数据库的性能。所以一般情况下，尽量少用分布式事务。

4、微服务的设计原则

软件设计就好比建筑设计。 Architect 这个词在建筑学中是“建筑师”的意思，而在软件领域里则是“架构师”的意思，可见它们确实有相似之处 。无论是建筑师还是架构师，他们都希望把作品设计出自己的特色，并且更愿意把创造出的东西被称为艺术品。然而现实却是，建筑设计和软件设计有非常大的区别。建筑师设计并建造出来的建筑往往很难有变化，除非拆了重建。而架构师设计出来的软件系统，为了满足产品的业务发展，在它的整个生命周期中，每一个版本都有很多的变化 。

软件设计每一个版本都在变化，所以软件设计应该是渐进式发展 。 软件从一开始就不应该被设计成微服务架构，微服务架构固然有优势，但是它需要更多 的资源，包括服务器资源、技术人员等。追求大公司所带来的技术解决方案，刻意地追求某个新技术，企图使用技术解决所有的问题，这些都是软件设计的误区。

技术应该是随着业务的发展而发展的，任何脱离业务的技术是不能产生价值的 。在初创公司 ，业务很单一时，如果在 LAMP 单体构架够用的情况下，就应该用 LAMP，因为它开发速度快，性价比高。随着业务的发展，用户量的增加 ，可以考虑将数据库读写分离、加缓存、加复杂均衡服务器、将应用程序集群化部署等。如果业务还在不断发展 ，这时可以考虑使用分布式系统，例如微服务架构的系统。不管使用什么样的架构，驱动架构的发展一定是业务的发展，只有当前架构不再适合当前业务的发展，才考虑更换架构。