分布式系统调研 2

出现背景：

单个节点的处理能力无法满足日益增长的计算、存储任务，硬件的提升（加内存、加磁盘、使用更好的 CPU）高昂到得不偿失，应用程序也不能进一步优化。

官方定义：

分布式系统是一组电脑，透过网络相互连接传递消息与通信后并协调它们的行为而形成的系统。组件之间彼此进行交互以实现一个共同的目标。

通俗理解：

一个分布式系统是一组计算机系统一起工作，在终端用户看来，就像一台计算机在工作一样。这组一起工作的计算机，拥有共享的状态，他们同时运行，独立机器的故障不会影响整个系统的正常运行。

举个例子，传统的数据库是存储在一台机器的文件系统上的。每当我们取出或者插入信息的时候，我们直接和那台机器进行交互。那么现在我们把这个传统的数据库设计成**分布式数据库**。假设我们使用了三台机器来构建这台分布式数据库，我们追求的结果是，在机器 1 上插入一条记录，需要在机器 3 上可以返回那条记录，当然了，机器 1 和 2 也要能够返回这条记录。

从进程角度看，两个程序分别运行在两个台主机的进程上，它们相互协作最终完成同一个服务（或者功能），那么理论上这两个程序所组成的系统，也可以称作是“分布式系统”。当然，这个两个程序可以是不同的程序，也可以是相同的程序。如果是相同的程序，我们又可以称之为“集群”。所谓集群，就是将相同的程序，通过不断横向扩展，以提高服务能力的方式。

例子：

有所不同的面向服务的架构，大型多人在线游戏，对等网络应用。

集中式系统：

集中式系统就是把所有的程序、功能都集中到一台主机上，从而往外提供服务的方式。

分布式系统与集群的关联：

这两者并不是对立的。因为分布式系统是通过多个节点的集群来完成一个任务，让外界看起来是跟一套系统作为一个整体打交道。一套分布式系统可以有多个集群，这些集群可以业务进行划分，也可以物理区域进行划分。每一个集群可以作为这个分布式系统的一个节点。这些集群节点组成的分布式系统，又可以作为单个的节点与其他的节点组成一个集群。

分布式系统有哪些优势：

1.可扩展性。分布式系统最大的好处就是能够让你横向的扩展系统。以单一数据库为例，能够处理更多流量的唯一方式就是升级数据库运行的硬件，这就是纵向扩展。而纵向扩展的是有局限性的。当到了一定程度以后，我们会发现即使最好的硬件，也不能够满足当前流量的需求。横向扩展是指通过增加更多的机器来提升整个系统的性能，而不是靠升级单台计算机的硬件。从价格上来说，横向扩展相比纵向扩展更容易控制。最根本的问题是纵向扩展有很强的局限性，达到最新硬件的能力以后，还是无法满足中等或者大型工作负载的技术要求。

横向扩展则没有这个限制，它没有上限，每当性能下降的时候，你就需要增加一台机器，这样理论上讲可以达到无限大的工作负载支持。
2.容错性。容错性是指分布式系统的某个节点出现错误以后，并不会导致整个系统的瘫痪。而单机系统出错以后，可能会导致整个系统的崩溃。
3.低延迟性。低延迟是通过在不同的物理位置部署不同的机器，通过就近获取的原则降低访问的延迟时间。
4.硬件资源可以与多个节点共享，而不是只限于一个节点。
5.可用性与可靠性：一般来说，分布式系统是需要长时间甚至 7*24 小时提供服务的。可用性是指系统在各种情况对外提供服务的能力，简单来说，可以通过不可用时间与正常服务时间的必知来衡量；而可靠性而是指计算结果正确、存储的数据不丢失。
6.高性能：不管是单机还是分布式系统，大家都非常关注性能。不同的系统对性能的衡量指标是不同的，最常见的：高并发，单位时间内处理的任务越多越好；低延迟：每个任务的平均时间越少越好。这个其实跟操作系统 CPU 的调度策略很像。
7.一致性：分布式系统为了提高可用性可靠性，一般会引入冗余（复制集）。那么如何保证这些节点上的状态一致，这就是分布式系统不得不面对的一致性问题。一致性有很多等级，一致性越强，对用户越友好，但会制约系统的可用性；一致性等级越低，用户就需要兼容数据不一致的情况，但系统的可用性、并发性很高很多。

分布式系统弊端：

1.实现上会更加复杂，分布式系统将会是更难理解、设计、构建和管理的（如连接到分布式系统的数据库），同时意味着应用程序的根源问题更难发现。
2.部署维护和调试分布式系统是非常头疼的一件事情。
3.在分布式系统中**很难提供足够的安全**，因为节点以及连接都需要安全。
4.**一些消息和数据在从一个节点转移到另一个节点时，可能会在网络中丢失**。
5.如果**分布式系统的所有节点都试图同时发送数据，网络中可能会出现过载现象**。

分布式系统会面临哪些挑战：

1.异构性：分布式系统由于基于不同的网络、操作系统、计算机硬件和编程语言来构造，必须要考虑一种通用的网络通信协议来屏蔽异构系统之间的差异。一般交由中间件来处理这些差异。
2.缺乏全球时钟：在程序需要协作时，它们通过交换消息来协调它们的动作。紧密的协调经常依赖于对程序动作发生时间的共识，但是，实际上网络上计算机同步时钟的准确性受到极大的限制，即没有一个正确时间的全局概念。这是通过网络发送消息作为唯一的通信方式这一事实带来的直接结果。
3.一致性：数据被分散或者复制到不同的机器上，如何保证各台主机之间的数据的一致性将成为一个难点。
4.故障的独立性：任何计算机都有可能故障，且各种故障不尽相同。他们之间出现故障的时机也是相互独立的。一般分布式系统要设计成被允许出现部分故障而不影响整个系统的正常使用。
5.并发：分布式系统的目的，是为了更好的共享资源。那么系统中的每个资源都必须被设计成在并发环境中是安全的。
6.透明性：分布式系统中任何组件的故障、或者主机的升级、迁移对于用户来说都是透明的，不可见的。

7.开放性：分布式系统由不同的程序员来编写不同的组件，组件最终要集成成为一个系统，那么组件所发布的接口必须遵守一定的规范且能够被互相理解。
8.安全性：加密用于给共享资源提供适当的保护，在网络上所有传递的敏感信息，都需要进行加密。拒绝服务攻击仍然是一个有待解决的问题。
9.可扩展性：系统要设计成随着业务量的增加，相应的系统也必须要能扩展来提供对应的服务。
10.不可靠的网络：节点间通过网络通信，而网络是不可靠的。可能的网络问题包括：网络分割、延时、丢包、乱序。相比单机过程调用，网络通信最让人头疼的是超时：节点 A 向节点 B 发出请求，在约定的时间内没有收到节点 B 的响应，那么 B 是否处理了请求，这个是不确定的，这个不确定会带来诸多问题，最简单的，是否要重试请求，节点 B 会不会多次处理同一个请求。

分布式系统工作原理：

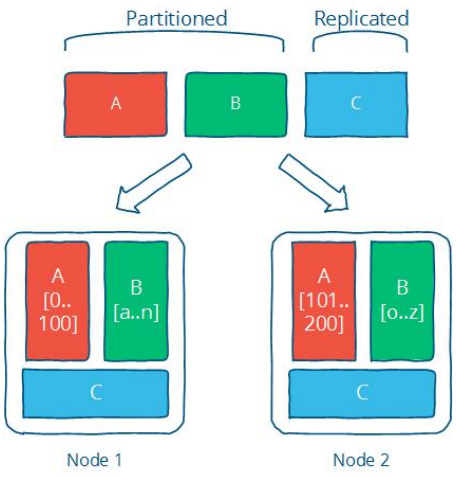
在操作系统中，对计算与存储有非常详尽的讨论，分布式系统只不过将这些理论推广到多个节点罢了。

那么分布式系统怎么将任务分发到这些计算机节点呢，很简单的思想，分而治之，即分片（partition）。对于计算，那么就是对计算任务进行切换，每个节点算一些，最终汇总就行了，这就是 MapReduce 的思想；对于存储，更好理解一下，每个节点存一部分数据就行了。当数据规模变大的时候，Partition 是唯一的选择，同时也会带来一些好处：

（1）提升性能和并发，操作被分发到不同的分片，相互独立

（2）提升系统的可用性，即使部分分片不能用，其他分片不会受到影响

理想的情况下，有分片就行了，但事实的情况却不大理想。原因在于，分布式系统中有大量的节点，且通过网络通信。单个节点的故障（进程 crash、断电、磁盘损坏）是个小概率事件，但整个系统的故障率会随节点的增加而指数级增加，网络通信也可能出现断网、高延迟的情况。在这种一定会出现的“异常”情况下，分布式系统还是需要继续稳定的对外提供服务，即需要较强的容错性。最简单的办法，就是冗余或者复制集（Replication），即多个节点负责同一个任务，最为常见的就是分布式存储中，多个节点复杂存储同一份数据，以此增强可用性与可靠性。同时，Replication 也会带来性能的提升，比如数据的 locality 可以减少用户的等待时间。



Partition 与 Replication 是如何协作的

CAP 定理：

CAP 定理指出，任何分布式数据存储只能提供以下三个保证中的两个：

一致性 Consistency: 每次读取都会收到最近的写入或错误。

可用性 Availability: 整个系统不会崩溃，每个非故障节点总会有一个相应，但不能保证它包含最近的写入。

分区容忍 Partition tolerant: 尽管节点之间的网络丢弃（或延迟）任意数量的消息，系统仍继续运行。

对于任何分布式系统来说，分区容忍是一个给定的条件，如果没有这一点，就不可能做到一致性和可用性。试想如果两个节点链接断掉了，他们如何能够做到既可用又一致?
最后你只能选择在网络分区情况下，你的系统要么强一致，要么高可用。

实践表明大多数应用程序更看重可用性。这个考量的主要原因是在不得不同步机器里实现强于一致性是时，网络延迟会成为一个问题。

这类因素使得应用程序通常会选择提供高可用性的解决方案。

此时采用的是最弱的一致性模型来解决的，这种模型保证了如果没有对某个项目进行新的更新，最终对该项目的所有访问都会返回最新的值。

这些系统提供了 BASE 属性，这是相对于传统数据库的 ACID 来讲的。 也就是(Basically available)基本上是可用的，系统总会返回一个响应。

(Soft state)软状态，系统可以随着时间的推移而变化，甚至在没有输入的情况下也可以变化，如保持最终的一致性的同步。

(Eventual consistency)最终的一致性，在没有输入的情况下，数据迟早会传播到每一个节点上，从而变得一致。

追求高可用的分布式数据库例子有 Cassandra；看重强一致性的数据库，有 HBase， Redis，Zookeeper。

附：解释

没有分布式系统可以避免网络故障，因此通常必须容忍网络分区。如果存在分区，则剩下两个选项：一致性或可用性。选择可用性的一致性时，系统将返回错误或在由于网络分区而无法保证特定信息，如果无法保证最新信息。在选择过度的可用性时，系统将始终处理查询并尝试返回最新的信息版本，即使它不能保证由于网络分区而最新。

CAP 经常被误解为在任何时候都可以选择放弃三个保证中的哪一个。事实上，只有在发生网络分区或故障时，才会在一致性和可用性之间进行选择。在没有网络故障的情况下，可以同时满足可用性和一致性。

如何来设计分布式：

设计分布式系统的本质就是“如何合理将一个系统拆分成多个子系统部署到不同机器上”。所以首要考虑的问题是如何合理的将系统进行拆分。由于拆分后的各个子系统不可能孤立的存在，必然是通过网络进行连接交互，所以它们之间如何通信变得尤为重要。当然在通信过程要识别“敌我”，防止信息在传递过程中被拦截和窜改，这就涉及到安全问题了。分布式系统要适应不断增长的业务需求，那么就需要考虑其扩展性。分布式系统还必须要保证可靠性和数据的一致性。

概况起来，在设计分布式系统时，应考虑以下几个问题：

系统如何拆分为子系统？

如何规划子系统间的通信？

通信过程中的安全如何考虑？

如何让子系统可以扩展？

子系统的可靠性如何保证？

数据的一致性是如何实现的？

我们在设计通信时，可以采用面向消息的中间件，比如 Apache ActiveMQ、RabbitMQ、Apache RocketMQ、Apache Kafka 等，也有类似与 Google Protocol Buffer、Thrift 等 RPC 框架。在设计分布式计算时，我们分布式计算可以采用 MapReduce、Apache Hadoop、Apache Spark 等。在大数据和分布式存储方面，我们可以选择 Apache HBase、Apache Cassandra、Memcached、Redis、MongoDB 等。在分布式监控方面，常用的技术包括 Nagios、Zabbix、Consul、ZooKeeper 等。

分布式系统设计思路：

1.分布式系统设计推演

我们先讲一个场景，我们现有的网络应用变得越来越流行，服务的人数也越来越多，导致我们的应用程序每秒收到的请求，远远超过能够正常处理的数量。这会导致应用程序性能下降明显，用户也会注意到这一点。

那我们下面就来扩展一下我们的应用程序来满足更高的要求。**一般来说我们读取信息的频率要远远超过插入或者修改的频率。**

下面我们使用**主从复制策略**来实现扩展系统。我们可以创建两个新的数据库服务器，他们与主服务器同步。用户业务对这两个新的数据库只能读取。每次当向主数据库插入和修改信息时，都会异步的通知副本数据库进行更新变化。

在这一步上我们已经有了三倍于原来系统读取数据的性能支持。但是这里有一个问题，在数据库事务的设计当中，我们遵循 ACID 原则。但是在我们同时对其他两个数据库进行数据更新的时候，我们有一个时间窗口失去了一致性原则。如果在这个时间窗口内对两个新的数据库进行查询，可能查不到数据。这个时候如果同步这三个数据库的数据，就会影响写操作的性能。这是我们在设计分布式系统的时候，不得不承受的一些代价。

上面的主从复制策略解决了用户读取性能方面的需求，但是当数据量达到一定程度，一台机子上无法存放的时候，我们需要扩展写操作性能。要解决这样的问题，我们可以使用分区技术。分区技术是指根据特定的算法，比如用户名 a 到 z 作为不同的分区，分别指向不同的数据库写入，每个写入数据库会有若干读取的从数据库进行同步提升读取性能。

当然，这样使得整套系统变得更加复杂。最重要的难点是分区算法。你们试想一下，如果 c 开头的用户名比其他开头的用户名要多很多，这会导致 c 区的数据量非常庞大，相应地，对于 c 区的请求也会远远大于其他区。此时 c 区成为热点。要避免热点，需要对 c 区进行拆分。此时要进行共享数据就会变得非常昂贵，甚至可能导致停机。

如果一切都很理想，那我们就拥有了 n 倍的写入流量，n 是分区的数目。

当然这里也存在一个陷阱，我们进行数据分区以后，导致除了分区键以外的查询都变得非常低效，尤其是对于 sql 语句如 join 查询就变得非常之糟糕，导致一些复杂的查询根本无法使用。

2.分布系统进行分布的常用方式：哈希

以哈希方式把不同的值进行哈希运算，映射到不同的机器或者节点上。这种方式在扩展的时候比较困难，因为数据分散在多个机器上很容易出现分布不均的情况，常见的哈希对象有 ip，url，id 等。

数据范围

按数据范围分布，比如 ID 在 1~100 的在机器 a 上，ID 在 100~200 的在机器 b 上，诸如此类。这种分布方法数据比较均匀。如果某个节点处理能力有限，可以直接分裂这个节点。维护数据分布的这些原数据，如果量非常大的话，可能会出现单点瓶颈。因此一定要严格控制元数

据量。

数据量
按数据量来分布数据，是以较为固定的大小将数据划分为若干的数据块，再把不同的数据块分布到不同的服务器上。以数据量来进行分布的这些数据，也需要被记录下来作为元数据来管理。当集群规模很大时，元数据的量也会变大。

副本与数据分布
这种方式是指把数据给分散到多个服务器上。如果其中一台出现问题，请求就会被转到其他服务器上。其原理是多个机器互为副本，这是比较理想的实现负载分压的方式。

一致性哈希
一致性哈希。通过哈希域构造哈希环，在增加机器时，变动的是其附近的节点，分摊的是附近节点的压力，其元数据的维护和按数量分布的维护方式一致。

使用以上方式进行分布的例子:
GFS, HDFS: 按数据量分布。
Map reduce: 按 GFS 数据分布做本地化。
BigTable, HBase 按数据范围分布。
Pnuts: 按哈希方式或者数据范围分布。
Dynamo, Cassndra: 按一致性哈希分布。
Mola, Armor, BigPipe: 按哈希方式分布。
Doris: 按哈希方式和按数据量分布进行组合。

3.分布式应用通常使用的架构类型哪些?
（1）客户端服务器
在这个类型中，分布式系统架构有一个服务器作为共享资源。比如打印机数据库或者网络服务器。它有多个客户机，这些客户机决定何时使用共享资源，如何使用和显示改变数据，并将其送回服务器，像 git 这样的代码仓，这是一个很好的例子。
（2）三层架构
这种架构把系统分为表现层，逻辑层和数据层，这简化了应用程序的部署，大部分早期的网络应用都是三层的。
（3）多层架构
上面的三层架构是多层架构的一种特殊形式。一般会把上面的三层进行更详细的划分，比如说以业务的形式进行分层。
（4）点对点架构
在这种架构中，没有专门的机器提供服务或管理网络资源。而是将责任统一分配给所有的机器，成为对等机，对等机既可以作为客户机，也可以作为服务器。这种架构的例子，包括 bittorrent 和区块链。
（5）以数据库为中心
这种架构是指用一个共享的数据库，使分布式的各个节点在不需要任何形式直接通信的情况下，进行协同工作的架构。

对于上面的各种技术与理论，业界有哪些实现：

负载均衡：

Nginx：高性能、高并发的 web 服务器；功能包括负载均衡、反向代理、静态内容缓存、访问控制；工作在应用层

LVS： Linux virtual server，基于集群技术和 Linux 操作系统实现一个高性能、高可用的服务器；工作在网络层

webserver：

Java：Tomcat，Apache，Jboss

Python：gunicorn、uwsgi、twisted、webpy、tornado

service：

SOA、微服务、spring boot，django

容器：

docker，kubernetes

cache：

memcache、redis 等

协调中心：

zookeeper、etcd 等

zookeeper 使用了 Paxos 协议 Paxos 是强一致性，高可用的去中心化分布式。zookeeper 的使用场景非常广泛，之后细讲。

rpc 框架：

grpc、dubbo、brpc

dubbo 是阿里开源的 Java 语言开发的高性能 RPC 框架，在阿里系的诸多架构中，都使用了 dubbo + spring boot

消息队列：

kafka、rabbitMQ、rocketMQ、QSP

消息队列的应用场景：异步处理、应用解耦、流量削锋和消息通讯

实时数据平台：

storm、akka

离线数据平台：

hadoop、spark

PS: apark、akka、kafka 都是 scala 语言写的，看到这个语言还是很牛逼的

dbproxy：

cobar 也是阿里开源的，在阿里系中使用也非常广泛，是关系型数据库的 sharding + replica 代理

db：

mysql、oracle、MongoDB、HBase

搜索：

elasticsearch、solr

日志：

rsyslog、elk、flume