作业一：分布式系统的历史演变

(自然辩证法课程报告)

姓　　名：肖文韬

学　　号：2020214245

专 业 方 向：电子信息（计算机技术）

邮　　箱：xwt20@mails.tsinghua.edu.cn

二〇二一年三月二十七日

第 1 章　引言

随着世界科技的不断发展壮大，分布式系统的应用越来越广泛。它们是计算机科学中一个庞大而复杂的研究领域。本文旨在介绍分布式系统的基本情况，展示这种系统的不同类别，同时不深入探讨细节。

1.1　分布式系统的定义

分布式系统最简单的定义是一组计算机共同工作，在终端用户看来就是一台计算机。这些机器有一个共享的状态，同时运行，并且可以独立地发生故障而不影响整个系统的正常运行时间。我们循序渐进地通过一个分布式系统的例子，让大家对它有一个更好的认识。

我们以数据库和网络应用组成的系统为例。传统的数据库存储在一台机器的文件系统中，无论何时你想在其中获取/插入信息–你都要直接与这台机器对话。我们要想分发这个数据库系统，就需要让这个数据库同时在多台机器上运行。用户必须能够和他选择的任何一台机器对话，而且不应该让人看出他不是在和一台机器对话–如果他在节点 #1 中插入一条记录，节点 #3 必须能够返回这条记录。

1.2　为什么要分布式系统?

系统总是分布式的，这是必然的。事实是–管理分布式系统是一个复杂的话题，充满了陷阱和地雷。部署、维护和调试分布式系统是一件令人头疼的事情，那为什么还要去做呢?

分布式系统能够让你做到的是横向扩展。回到我们之前的单一数据库服务器的例子，处理更多流量的唯一方法就是升级数据库运行的硬件。这就是所谓的垂直扩展。垂直扩展在你可以做到的时候都是很好的，但是到了一定程度之后你会发现，即使是最好的硬件也不足以满足足够的流量。横向扩展简单来说就是增加更多的电脑，而不是升级单台电脑的硬件。

在一定的门槛后，它的价格明显比垂直扩展便宜，但这并不是它作为主要优选案例的理由。垂直扩展只能将你的性能提升到最新硬件的能力。事实证明，这些能力对于拥有中等到大型工作负载的技术公司是不够的。横向扩展最好的一点是，你可以扩展多少没有上限–每当性能下降时，你只需再增加一台机器，就有可能达到无限大。

易扩展并不是你从分布式系统中获得的唯一好处。容错和低延迟也同样重要。

1. **容错**。一个由 10 台机器组成的集群横跨两个数据中心，本质上比单台机器更容错。即使一个数据中心着火，你的应用程序仍然可以工作。
2. **低延迟**。一个网络数据包走遍世界的时间在物理上是受光速限制的。例如，在纽约到悉尼的光缆中，一个请求的往返时间（也就是来回走）最短的时间是 160ms。分布式系统允许你在两个城市都有一个节点，让流量打到离它最近的节点。

不过，为了使分布式系统发挥作用，你需要在这些机器上运行的软件被专门设计成可以同时在多台计算机上运行，并处理随之而来的问题。事实证明，这并非易事。

1.3　扩展我们的数据库

想象一下，我们的网络应用变得非常流行。想象一下，我们的数据库开始得到每秒两倍于它能处理的查询量。你的应用程序将立即开始性能下降，这将被你的用户注意到。我们需要努力让我们的数据库规模满足我们的高要求。

在一个典型的网络应用中，你阅读信息的频率通常比你插入新信息或修改旧信息的频率高得多。有一种方法可以提高读取性能，那就是通过所谓的主-副复制策略。在这里，你创建两个新的数据库服务器，与主服务器同步。问题是，你只能从这些新实例中读取。每当你插入或修改信息时，你都会与主数据库对话。反过来，它也会异步地通知副本的变化，并且它们也会保存它。恭喜你，你现在可以执行 3 倍的读取查询了! 这不是很好吗?

1.4　陷阱

然而，上面的例子可能会出现问题，我们失去了关系型数据库的 ACID 保证中的 C，它代表了一致性。现在存在一种可能，就是我们在数据库中插入一条新的记录，之后马上对它发出读取查询，却什么也没有得到，就像它不存在一样! 将新信息从主节点传播到副本并不是瞬间发生的。实际上存在一个时间窗口，在这个时间窗口中，你可以获取陈旧的信息。如果不是这样，你的写性能就会受到影响，因为它必须同步等待数据的传播。

分布式系统会有少量的权衡。如果你想充分扩展，这个特殊的问题是你必须承受的。

1.5　继续扩大规模

使用复制数据库的方法，我们可以在一定程度上水平扩展我们的读取流量。这是很好的，但我们在写流量方面遇到了障碍–它仍然全部在一台服务器上！这也是我们的一个问题。我们在这里没有太多的选择。我们只需要将我们的写入流量分成多个服务器，因为一个服务器无法处理。

一种方法是采用多主节点复制策略。在那里，你有多个支持读写的主节点，而不是只能读取的复制体。不幸的是，这变得非常复杂，因为你现在有能力创建冲突（例如插入两个具有相同 ID 的记录）。

我们再来看看另一种技术，叫碎片（也叫分区）。使用碎片机制，您可以将您的服务器分割成多个较小的服务器，称为碎片。这些碎片都保存着不同的记录–你要创建一个规则，规定什么样的记录进入哪个碎片。创建规则，使数据以统一的方式传播是非常重要的。

一个可能的方法是根据记录的一些信息来定义范围（例如名字为 A-D 的用户）。这个碎片机制的键值应该非常谨慎地选择，因为基于任意列的负载并不总是相等的。(例如更多的人的名字是以 C 而不是 Z 开头)。一个单一的碎片如果收到的请求比其他碎片多，就被称为热点，必须避免。一旦被拆分，重新共享数据就会变得非常昂贵，并可能导致大量的停机，就像 FourSquare 臭名昭著的 11 小时停机一样。

为了使我们的例子简单化，假设我们的客户端（Rails 应用）知道对每条记录使用哪个数据库。值得注意的是，有很多策略可以用于碎片机制，这是一个简单的例子来说明这个概念。我们现在已经赢得了不少好处–我们可以将我们的写入流量增加 N 倍，其中 N 是碎片的数量。这实际上让我们几乎没有限制–想象一下，通过这种分区，我们可以获得多么精细的粒度。

软件工程中的一切都或多或少地存在着权衡，这也不例外。碎片机制不是一件简单的事，除非真的需要，否则最好避免。我们现在已经让除分区键以外的键的查询变得非常低效（它们需要通过所有的碎片）。SQL JOIN 查询更糟糕，复杂的查询实际上变得无法使用。

以下是按照分布式系统出现时间，对分布式系统的历史演变进行简要介绍。

第 2 章　分布式数据存储

分布式数据存储是应用最广泛、公认的分布式数据库。大多数分布式数据库是 NoSQL 非关系型数据库，仅限于键值语义。它们以一致性或可用性为代价，提供了令人难以置信的性能和可扩展性。

已知规模–据悉，苹果早在 2015 年就使用了 75000 个 Apache Cassandra 节点，存储了超过 10PB 的数据。

如果不首先介绍 CAP 定理，我们就无法进入对分布式数据存储的讨论。

2.1　CAP 理论

早在 2002 年就被证明，CAP 定理指出，一个分布式数据存储不能同时具有一致性、可用性和分区容忍性。

1. **一致性**。读写得到的结果必须按照她们执行的结果的顺序保持一致。
2. **可用性**。整个系统不会死亡–每个非故障节点总是返回一个响应。
3. **分区容错**。尽管有网络分区，但系统继续运行，并坚持其一致性/可用性保证。

实际上，对于任何分布式数据存储来说，分区容错必须是一个充分条件。没有分区容错，你就无法拥有一致性和可用性。想一想：如果你有两个接受信息的节点，而它们的连接却失效了–它们怎么能同时可用，同时为你提供一致性呢？它们没有办法知道另一个节点在做什么，因此要么离线（不可用），要么用陈旧的信息工作（不一致）。

最后，你要选择你的系统在网络分区下是要强一致还是高可用。实践表明，大多数应用更重视可用性。你不一定总是需要强一致性。即使如此，这种权衡也不一定是因为你需要 100% 的可用性保证，而是因为在必须同步机器以实现强一致性时，网络延迟可能是一个问题。这些和更多的因素使得应用程序通常会选择提供高可用性的解决方案。这种数据库用最弱的一致性模型–最终一致性来解决。这种模型保证了如果没有对某个项目进行新的更新，最终所有对该项目的访问都会返回最新的更新值。

这些系统提供 BASE 属性（相对于传统数据库的 ACID）：

1. **基本可用（Basically Available）**。系统总是返回一个响应。
2. **软状态（Soft state）**。系统可能会随着时间的推移而改变，即使在没有投入的时期也是如此（由于最终的一致性）。

3. **最终一致性（Eventual consistency）**。在没有输入的情况下，数据迟早会传播到每一个节点–从而变得一致。

这种可用的分布式数据库的例子–Cassandra、Riak、Voldemort。当然，还有其他一些数据存储喜欢更强的一致性–HBase、Couchbase、Redis、Zookeeper。CAP 定理本身就值得写多篇文章–有些是关于如何根据客户端的行为调整系统的 CAP 属性，有些则是关于如何不正确理解它。

第 3 章　分布式计算

分布式计算是我们近年来看到的大数据处理大量涌入的关键。它是将一个巨大的任务（如汇总 1000 亿条记录）分割成许多较小的任务的技术，而这些任务没有一台计算机能够单独实际执行，每一个任务都可以放入一台商品机。你把你的巨大任务分割成许多较小的任务，让它们在许多机器上并行执行，适当地聚合数据，你就解决了最初的问题。这种方法又能让你横向扩展–当你有更大的任务时，只需在计算中加入更多的节点。

已知规模–2012 年，Folding@Home 有 16 万台活跃的机器。

这个领域的早期创新者是谷歌，由于其大量数据的需要，谷歌不得不发明一种新的分布式计算模式–MapReduce。他们在 2004 年发表了一篇论文，后来开源社区在此基础上创建了 Apache Hadoop。

3.1　MapReduce

MapReduce 可以简单地定义为两个步骤–映射数据和将其还原为有意义的东西。假设我们是 Medium 博客公司，出于仓储的目的，我们将庞大的信息存储在二级分布式数据库中。我们想获取代表 2017 年 4 月（一年前）全年每天发出的评论数量的数据。

这个例子尽可能的简短明了，但是想象一下，我们正在处理大量的数据（例如分析数十亿次的评论），显然我们不会把所有这些信息都存储在一台机器上，也不会只用一台机器来分析这些信息。我们也不会查询生产环境数据库，而是查询一些专门为低优先级离线工作建立的” 仓库” 数据库。

每个 Map 作业都是一个独立的节点，尽可能多的转换数据。每个作业都会遍历给定存储节点中的所有数据，并将其映射为日期和数字一的简单元组。然后，完成三个中间步骤–打乱顺序、排序和分区。它们基本上是将数据进一步排列并删除到相应的 reduce 作业中。由于我们要处理的是大数据，所以我们将每个 Reduce 作业分开，只对一个日期进行工作。

这是一个很好的范式，令人惊讶的是，你可以用它做很多事情–例如，你可以链接多个 MapReduce 作业。

3.2　更好的技术

MapReduce 现在可以说是经典方法，并且带来了一些问题。因为它是以批处理（作业）的方式工作的，所以出现了一个问题，如果你的作业失败了–你需要重新启动整个作业。一个 2 小时的作业失败真的会拖慢你的整个数据处理管道，你最不希望出现这种情况，尤其是在高峰期。另一个问题是你等待到收到结果的时间。在实时分析系统中（这些系统都有大数据，因此使用分布式计算），重要的是让你最新的破碎数据尽可能新鲜，当然不是几个小时前的数据。

因此，出现了其他架构来解决这些问题。即 Lambda 架构（批处理和流处理的混合）和 Kappa 架构（只有流处理）。这些领域的进步带来了新的工具使其得以实现–Kafka Streams、Apache Spark、Apache Storm、Apache Samza。

第 4 章　分布式文件系统

分布式文件系统可以被认为是分布式数据存储。它们的概念是一样的–在一个机器集群上存储和访问大量的数据，所有的机器都显得像一个整体。它们通常与分布式计算相辅相成。

已知的规模–雅虎早在 2011 年就以在超过 42,000 个节点上运行 HDFS 来存储 600PB 的数据而闻名。

维基百科定义的区别是，分布式文件系统允许使用与本地文件相同的接口和语义来访问文件，而不是通过像 Cassandra 查询语言（CQL）这样的自定义 API。

4.1　HDFS

Hadoop 分布式文件系统（HDFS）是通过 Hadoop 框架用于分布式计算的分布式文件系统。它被广泛采用，用于在许多机器上存储和复制大文件（GB 或 TB 大小）。

其架构主要由 NameNodes 和 DataNodes 组成。NameNodes 负责保存集群的元数据，比如哪个节点包含哪些文件块。它们作为网络的协调者，找出存储和复制文件的最佳位置，跟踪系统的健康状况。DataNodes 只是存储文件，并执行复制文件、写入新文件等命令。毫不奇怪，HDFS 最好与 Hadoop 一起用于计算，因为它为计算作业提供了数据意识。然后，所述作业会在存储数据的节点上运行。这利用了数据的位置性–优化计算并减少网络上的流量。

4.2　IPFS

星际文件系统（IPFS）是一个令人兴奋的新的分布式文件系统的点对点协议/网络。利用区块链技术，它拥有一个完全去中心化的架构，没有单一的所有者，也没有故障点。IPFS 提供了一个名为 IPNS 的命名系统（类似于 DNS），并让用户可以轻松地访问信息。它通过历史版本存储文件，类似于 Git 的做法。这允许访问一个文件以前的所有状态。

第 5 章　分布式应用

如果你在一个负载均衡器后面卷起 5 台 Rails 服务器，全部连接到一个数据库，你能称之为分布式应用吗？回顾一下我在上面的定义：

分布式系统是指一组计算机共同工作，在终端用户看来是一台计算机。这些机器具有共享状态，同时运行，可以独立地发生故障而不影响整个系统的正常运行时间。

如果你把数据库算作共享状态，你可以说这可以归为分布式系统–但你错了，因为你忽略了定义中的” 共同工作” 部分。一个系统只有当节点之间相互沟通协调行动时，才是分布式的。

因此，像在点对点网络上运行其后端代码的应用程序这样的东西，最好归为分布式应用程序。无论如何，这都是无谓的分类，没有任何作用，只能说明我们对事物的归类是多么的挑剔。

已知规模–2014 年 4 月，《权力的游戏》一集有 193000 个节点的 BitTorrent 群。

5.1　Erlang 虚拟机

Erlang 是一种函数式语言，它在并发、分发和容错方面有很好的语义。Erlang 虚拟机本身就负责处理 Erlang 应用的分发。其模型的工作原理是拥有许多孤立的轻量级进程，这些进程都能够通过内置的消息传递系统相互对话。这就是所谓的” 角色模型”，而 Erlang OTP 库可以看作是一个分布式角色框架（类似于 JVM 的 Akka）。

这种模式是帮助它实现极大并发的原因，相当简单–进程分布在运行它们的系统的可用内核上。由于这与网络环境没有区别（除了可以丢弃消息），Erlang 的虚拟机可以连接到运行在同一数据中心甚至另一个大陆的其他 Erlang 虚拟机。这个虚拟机群运行一个单一的应用程序，并通过接管（另一个节点被安排运行）来处理机器故障。事实上，该语言的分布式层是为了提供容错功能而添加的。运行在单台机器上的软件总是面临着该单台机器死机并使你的应用程序离线的风险。在许多节点上运行的软件可以更容易地处理硬件故障，但前提是应用程序在构建时要考虑到这一点。

5.2　BitTorrent

BitTorrent 是最广泛使用的协议之一，通过 torrent 在网络上传输大文件。其主要思想是促进网络中不同对等体之间的文件传输，而不必通过主服务器。使用 BitTorrent 客户端，您可以连接到世界各地的多台计算机来下载文件。当您打开一个.torrent 文件时，您会连接到一个所谓的跟踪器，这是一个作为协调器的机器。它有助于同行发现，向您展示网络中拥有您想要的文件的节点。

你有两种用户的概念，一个是水蛭，一个是播种者。水蛭是下载文件的用户，播种者是上传文件的用户。点对点网络的有趣之处在于，你作为一个普通用户，有能力加入并为网络做出贡献。

BitTorrent 及其前身（Gnutella、Napster）允许你自愿托管文件并上传给其他想要文件的用户。BitTorrent 如此受欢迎的原因是，它是第一个为向网络做出贡献的人提供激励措施。自由行，即用户只下载文件，是以前的文件共享协议的一个问题。

BitTorrent 在一定程度上解决了自由行的问题，让播种者向那些提供最佳下载率的人上传更多的内容。它的工作原理是激励你在下载文件的同时进行上传。不幸的是，在你完成下载后，没有任何东西让你在网络中保持活跃。这就导致网络中缺乏拥有完整文件的播种者，由于协议严重依赖这样的用户，私人跟踪器这样的解决方案就应运而生了。私有跟踪器要求你成为一个社区的成员 (通常只有邀请函)，才能参与到分布式网络中。

在该领域取得进步后，发明了无跟踪山洪。这是对 BitTorrent 协议的升级，它不依靠中心化的跟踪器来收集元数据和寻找同行，而是使用新的算法。其中一个例子是 Kademlia（主线 DHT），这是一个分布式哈希表（DHT），它允许你通过其他同行找到同行。实际上，每个用户都执行了一个追踪者的职责。

5.3　区块链

区块链是目前用于分布式账本的底层技术，事实上标志着它们的开始。这一分布式领域最新最伟大的创新，使得有史以来第一个真正的分布式支付协议–比特币得以诞生。

区块链是一种分布式账本，携带着其网络中曾经发生的所有交易的有序清单。交易被分组并存储在区块中。整个区块链本质上是一个链接的区块列表（因此而得名）。所述区块的创建在计算上是昂贵的，并通过加密技术彼此紧密相连。

简单地说，每个区块都包含了当前区块内容（以 Merkle 树的形式）加上前一个区块的哈希值的特殊哈希值（以 X 量的零开始）。这个哈希值需要大量的 CPU

功率来产生，因为只有通过蛮力才能得出。

矿工是试图计算哈希值的节点（通过蛮力）。矿工们互相竞争，谁能提出一个随机的字符串（称为 nonce），当与内容相结合时，就会产生上述的散列。一旦有人找到了正确的 nonce–他就会把它广播到整个网络。然后，每个节点都会对这个字符串进行验证，并将其纳入自己的链中。

5.4　以太坊

Ethereum 可以被认为是一个基于区块链的可编程软件平台。它有自己的加密货币（以太坊），为其区块链上的智能合约部署提供了动力。

智能合约是一段代码，作为单笔交易存储在 Ethereum 区块链中。要运行代码，你所要做的就是发出一个以智能合约为目的的交易。这反过来使矿工节点执行代码和它所发生的任何变化。代码是在 Ethereum 虚拟机内部执行的。

Solidity 是 Ethereum 的原生编程语言，是用来编写智能合约的。它是一种图灵完备的编程语言，直接与 Ethereum 区块链对接，可以查询余额或其他智能合约结果等状态。为了防止无限循环，运行代码需要一定量的以太币。由于区块链可以被解释为一系列的状态变化，很多分布式应用 (DApps) 已经建立在 Ethereum 和类似平台之上。