分布式系统原理介绍

刘 杰

目录

前言

半年前，同学提出希望学习分布式系统的相关知识，然而笔者发觉缺少一份既有一定理论内容、又贴近工程实践，既深入浅出又较为全面的学习资料。为此，笔者尝试对自己在学习、开发分布式系统过程中获得的一些理论与实践进行总结，进而催生了写作本文的想法。

分布式系统理论体系非常庞大，涉及知识面也非常广博，由于笔者的肤浅，本文精心选择了部分在工程实践中应用广泛、简单有效的分布式理论、算法、协议加以介绍。全文分为两大部分，第一部分介绍了分布式系统的一些基本概念并框定了本文的问题模型和问题域，作为后续章节的基础。第二部分介绍了一些分布式系统的理论，在介绍这些理论时，注重引入实例并加以应用，同时将这些理论投影到真实的系统中。

在原本的写作计划中，本文还有第三部分的内容。第三部将分析若干有代表性的分布式系统，着重分析第二部分中的理论是如何被综合运用在这些真实的分布式系统中的。在具体写作时，笔者将这部分的内容拆解到第二章各节的“工程投影”中，简要分析了第二章的理论是如何体现在各个典型分布式系统中的。即便如此，笔者觉得后续可以再作一篇《典型分布式系统分析》，从各个系统的角度横向分析这些系统的特点。

开发分布式系统需要多方面的知识、技能与经验。受限于作者的能力，本文将讨论的重点集中在分布式层面的协议和算法设计，开发分布式系统所需要的其他方面的知识与技术则不在本文的讨论范围。

最后，感谢李海磊先生、吴学林先生对笔者学习、实践分布式系统给予的大力指导；感谢梁建平先生、徐鹏先生对本文提出的诸多宝贵意见和建议。

2012 年 5 月

1 概念

1.1 模型

一些经典的分布式系统的资料对分布式系统的全貌做了比较详细的介绍[1][2]。为了控制规模，在开始讨论分布式系统的协议、原理与设计之前，首先给出在本文中研究的分布式系统在分布式层面的基本问题模型。后续所有的讨论都限定在这个模型的范围内，超过模型范围的内容则不在本文中讨论。本文尽量精简分布式系统模型是为了控制问题的规模。

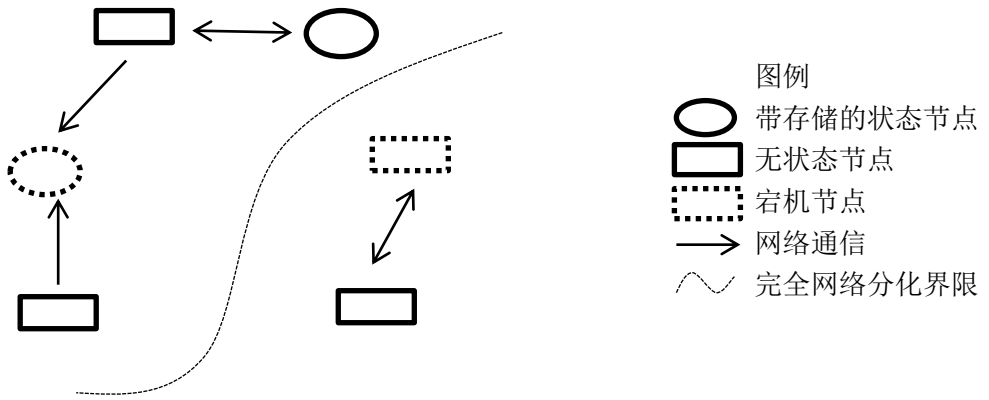


图 1-1 本文的分布式系统模型

1.1.1 节点

节点是指一个可以独立按照分布式协议完成一组逻辑的程序个体。在具体的工程项目中，一个节点往往是一个操作系统上的进程。在本文的模型中，认为节点是一个完整的、不可分的整体，如果某个程序进程实际上由若干相对独立部分构成，则在模型中可以将一个进程划分为多个节点。本文不考虑拜占庭问题，即认为节点始终按照约定的分布式协议工作。图 1-1 中以矩形和椭圆形表示了节点。

1.1.2 通信

节点与节点之间是完全独立、相互隔离的，节点之间传递信息的唯一方式是通过不可靠的网络进行通信。即一个节点可以向其他节点通过网络发送消息，但发送消息的节点无法确认消息是否被接收节点完整正确收到。在 1.1.4.2 节，将重点讨论这种网络通信异常的问题。图 1-1 中以带箭头的直线表示了消息通信，其中某些节点间使用双箭头表示网络双向可达，而某些节点间只有单箭头表示网络单向可达，而某些节点间没有箭头表示网络完全不可达。

1.1.3 存储

节点可以通过将数据写入与节点在同一台机器的本地存储设备保存数据。通常的存储设备有磁

盘、SSD 等。存储、读取数据的节点称为有状态的节点，反之称为无状态的节点。如果某个节点 A 存储数据的方式是将数据通过网络发送到另一个节点 B，由节点 B 负责将数据存储到节点 B 的本地存储设备，那么不能认为节点 A 是有状态的节点，而只有节点 B 是有状态的节点。图 1-1 中以矩形节点表示无状态节点，以椭圆形节点表示有状态节点。

1.1.4 异常

分布式系统核心问题之一就是处理各种异常(failure)情况。本节着重讨论在本文范围内系统会遇到的各种异常。

1.1.4.1 机器宕机

机器宕机是最常见的异常之一。在大型集群中每日宕机发生的概率为千分之一左右。在实践中，一台宕机的机器恢复的时间通常认为是 24 小时，一般需要人工介入重启机器。宕机造成的后果通常为该机器上节点不能正常工作。假设节点的工作流程是三个独立原子的步骤，宕机造成的后果是节点可能在处理完某个步骤后不再继续处理后续步骤。由于本文不考虑拜占庭问题，认为不会出现执行完第一个步骤后跳过第二个步骤而执行第三个步骤的情况。宕机是一个完全随机的事件，在本文中，认为在任何时刻都可能发生宕机，从而造成某些协议流程无法完成。

当发生宕机时，节点无法进入正常工作的状态，称之为“不可用”(unavailable)状态。机器重启后，机器上的节点可以重新启动，但节点将失去所有的内存信息。在某些分布式系统中，节点可以通过读取本地存储设备中的信息或通过读取其他节点数据的方式恢复内存信息，从而恢复到某一宕机前的状态，进而重新进入正常工作状态、即“可用”(available)状态。而另一些分布式系统中的某些无状态节点则无需读取读取任何信息就可以立刻重新“可用”。使得节点在宕机后从“不可用”到“可用”的过程称为宕机恢复。

图 1-1 中虚线节点表示宕机的节点。

1.1.4.2 网络异常

网络异常是另一类常见的异常形式。在 1.1.2 中已经定义，节点间通过不可靠的网络进行通信。本节定义了本文范畴内的各种网络异常。

1.1.4.2.1 消息丢失

消息丢失是最常见的网络异常。对于常见的 IP 网络来说，网络层不保证数据报文(IP fragment)的可靠传递，在发生网络拥塞、路由变动、设备异常等情况时，都可能发生发送的数据丢失。由于网络数据丢失的异常存在，直接决定了分布式系统的协议必须能处理网络数据丢失的情况。

依据网络质量的不同，网络消息丢失的概率也不同，甚至可能出现在一段时间内某些节点之间

的网络消息完全丢失的情况。如果某些节点的直接的网络通信正常或丢包率在合理范围内，而某些节点之间始终无法正常通信，则称这种特殊的网络异常为“网络分化”(network partition)。网络分化是一类常见的网络异常，尤其当分布式系统部署在多个机房之间时。图 1-1 中，用虚线分割了两片节点，这两片节点之间彼此完全无法通信，即出现了“网络分化”。

例 1.1：某分布式系统部署于两个机房，机房间使用内部独立光纤链路。由于机房间的光纤链路交割调整，两个机房间通信中断，期间，各机房内的节点相互通信正常。更为严重的是，所有的英特网用户都可以正常访问两个机房内对外服务节点。本文后续将讨论出现这种严重的网络分化时，对分布式系统的设计带来的巨大挑战。

1.1.4.2.2 消息乱序

消息乱序是指节点发送的网络消息有一定的概率不是按照发送时的顺序依次到达目的节点。通常由于 IP 网络的存储转发机制、路由不确定性等问题，网络报文乱序也是一种常见的网络异常。这就要求设计分布式协议时，考虑使用序列号等机制处理网络消息的乱序问题，使得无效的、过期的网络消息不影响系统的正确性。

1.1.4.2.3 数据错误

网络上传输的数据有可能发生比特错误，从而造成数据错误。通常使用一定的校验码机制可以较为简单的检查出网络数据的错误，从而丢弃错误的数据。

1.1.4.2.4 不可靠的 TCP

TCP 协议为应用层提供了可靠的、面向连接的传输服务。TCP 协议是最优秀的传输层协议之一，其设计初衷就是在不可靠的网络之上建立可靠的传输服务。TCP 协议通过为传输的每一个字节设置顺序递增的序列号，由接收方在收到数据后按序列号重组数据并发送确认信息，当发现数据包丢失时，TCP 协议重传丢失的数据包，从而 TCP 协议解决了网络数据包丢失的问题和数据包乱序问题。TCP 协议为每个 TCP 数据段（以太网上通常最大为 1460 字节）使用 32 位的校验和从而检查数据错误问题。TCP 协议通过设置接收和发送窗口的机制极大的提高了传输性能，解决了网络传输的时延与吞吐问题。TCP 协议最为复杂而巧妙的是其几十年来不断改进的拥塞控制算法，使得 TCP 可以动态感知底层链路的带宽加以合理使用并与其他 TCP 链接分享带宽（TCP friendly）。

上述种种使得 TCP 协议成为一个在通常情况下非常可靠的协议，然而在分布式系统的协议设计中不能认为所有网络通信都基于 TCP 协议则通信就是可靠的。一方面，TCP 协议保证了 TCP 协议栈之间的可靠的传输，但无法保证两个上层应用之间的可靠通信。通常的，当某个应用层程序通过 TCP 的系统调用发送一个网络消息时，即使 TCP 系统调用返回成功，也仅仅只能意味着该消息被本机的 TCP 协议栈接受，一般这个消息是被放入了 TCP 协议栈的缓冲区中。再退一步讲，即使目的机器的 TCP 协议栈后续也正常收到了该消息，并发送了确认数据包，也仅仅意味着消息达到了对方

机器的协议栈，而不能认为消息被目标应用程序进程接收到并正确处理了。当发送过程中出现宕机等异常时，TCP 协议栈缓冲区中的消息有可能被丢失从而无法被目标节点正确处理。更有甚者，在网络中断前，某数据包已经被目标进程正确处理，之后网络立刻中断，由于接收方的 TCP 协议栈发送的确认数据包始终被丢失，发送方的 TCP 协议栈也有可能告知发送进程发送失败。另一方面，TCP 协议只能保证同一个 TCP 链接内的网络消息不乱序，TCP 链接之间的网络消息顺序则无法保证。但在分布式系统中，一个节点向另一个节点发送数据，有可能是先后使用多个 TCP 链接发送，也有可能是同时并发多个 TCP 链接发送，那么发送进程不能认为先调用 TCP 系统调用发送的消息就一定会先于后发送的消息到达对方节点并被处理。

由上述分析，在设计分布系统的网络协议时即使使用 TCP 协议，也依旧要考虑网络异常，不能简单的认为使用 TCP 协议后通信就是可靠的。另一方面，如果完全放弃使用 TCP 协议，使用 UDP 协议加自定义的传输控制机制，则会使得系统设计复杂。尤其是要设计、实现一个像 TCP 那样优秀的拥塞控制机制是非常困难的。

1.1.4.3 分布式系统的三态

由于网络异常的存在，分布式系统中请求结果存在“三态”的概念。在单机系统中，我们调用一个函数实现一个功能，则这个函数要么成功、要么失败，只要不发生宕机其执行的结果是确定的。然而在分布式系统中，如果某个节点向另一个节点发起 RPC(Remote procedure call)调用，即某个节点 A 向另一个节点 B 发送一个消息，节点 B 根据收到的消息内容完成某些操作，并将操作的结果通过另一个消息返回给节点 A，那么这个 RPC 执行的结果有三种状态：“成功”、“失败”、“超时（未知）”，称之为分布式系统的三态。

如果请求 RPC 的节点 A 收到了执行 RPC 的节点 B 返回的消息，并且消息中说明执行成功，则该 RPC 的结果为“成功”。如果请求 RPC 的节点 A 收到了执行 RPC 的节点 B 返回的消息，并且消息中说明执行失败，则该 RPC 的结果为“失败”。但是，如果请求 RPC 的节点 A 在给定的时间内没有收到执行 RPC 的节点 B 返回的消息，则认为该操作“超时”。对于超时的请求，我们无法获知该请求是否被节点 B 成功执行了。这是因为，如果超时是由于节点 A 发向节点 B 的请求消息丢失造成的，则该操作肯定没有被节点 B 成功执行；但如果节点 A 成功的向节点 B 发送了请求消息，且节点 B 也成功的执行了该请求，但节点 B 发向节点 A 的结果消息被网络丢失了或者节点 B 在执行完该操作后立刻宕机没有能够发出结果消息，从而造成从节点 A 看来请求超时。所以一旦发生超时，请求方是无法获知 RPC 的执行结果的。图 1-2 给出了操作成功但超时的例子。

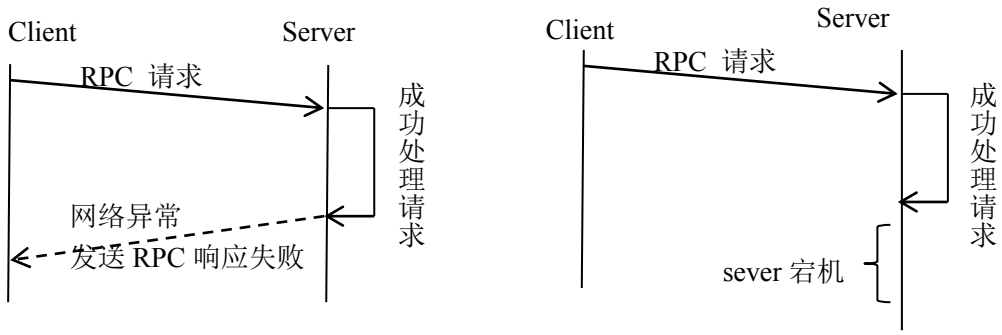


图 1-2RPC 执行成功但超时的例子

一个非常易于理解的例子是在网上银行进行转账操作，当系统超时，页面提示：“如果系统长时间未返回，请检查账户余额以确认交易是否成功”。

分布式系统一般需要区别对待 RPC 的“成功”、“失败”、“超时”三种状态。当出现“超时”时可以通过发起读取数据的操作以验证 RPC 是否成功（例如银行系统的做法）。另一种简单的做法是，设计分布

式协议时将执行步骤设计为可重试的，即具有所谓的“幂等性”。例如覆盖写就是一种常见的幂等性操作，因为重复的覆盖写最终的结果都相等。如果使用可重试的设计，当出现“失败”和“超时”时，一律重试操作直到“成功”。这样，即使超时的操作实际上已经成功了，重试操作也不会对正确性造成影响，从而简化了设计。

后续本文中，如果说明“不成功”即指“失败”或“超时”两种状态之一。如果说明“失败”则表示收到了明确的“失败”消息。

1.1.4.4 存储数据丢失

数据丢失指节点存储的数据不可被读取或读取出的数据错误。数据丢失是另一类常见的异常。尤其是使用机械硬盘做存储介质时，硬盘损坏的概率较大。对于有状态节点来说，数据丢失意味着状态丢失，通常只能从其他节点读取、恢复存储的状态。

1.1.4.5 无法归类的异常

在工程实践中，大量异常情况是无法预先可知的，更可恶的是这些异常往往是“半死不活”的状态。例如，磁盘故障会导致 IO 操作缓慢，从而有可能使得进程运行速度非常慢，“缓慢”是异常吗？如果慢的超过某种程度，则对系统会造成影响。更有甚者，几十秒进程非常慢甚至完全阻塞住，又几十秒恢复了，如此交替。又例如网络不稳定时也会引起“半死不活”异常，例如网络发生严重拥塞时约等于网络不通，过一会儿又恢复，恢复后又拥塞，如此交替。对于这些极端的无法预先归

类的异常，需要在具体的项目中，通过长期的工程实践调整应对。

1.1.4.6 异常处理的原则

在设计、推导、验证分布式系统的协议、流程时，最重要的工作之一就是思考在执行流程的每个步骤时一旦发生各种异常的情况下系统的处理方式及造成的影响。

例 1.2：某分布式协议实现一个 echo 功能，即由节点 A 向节点 B 发送一个消息，内容是一个整数，节点 B 收到后返回相同的消息。节点 A 发送的消息每次递增加 1。

节点 A 的处理流程为：

1. 向节点 B 发送一个消息，消息内容为当前需要发送的整数；
2. 等待接收从节点 B 发回的响应消息；
3. 若 B 发回的消息等于当前需要发送的整数，
    a) 将当前需要发送的整数加 1；
    b) 否则返回 1；

上述简单的流程可能遇到各种异常且不能正确处理：第一、当前需要发送的整数没有持久化，在上述流程中，一旦节点 A 宕机，节点 A 无法继续上述流程。第二、节点 B 一旦宕机，节点 A 不会收到响应消息，流程将卡在第二步无法进行下去。第三、若 A 发给 B 或 B 发回 A 的消息有一个丢失，节点 A 也不会收到响应消息。在本节中，不讨论如何修改这个流程以处理上述异常，举这个例子是为了说明异常分析的基本方法。

被大量工程实践所检验过的异常处理黄金原则是：任何在设计阶段考虑到的异常情况一定会在系统实际运行中发生，但在系统实际运行遇到的异常却很有可能在设计时未能考虑，所以，除非需求指标允许，在系统设计时不能放过任何异常情况。

工程中常常容易出问题的一种思路是认为某种异常出现的概率非常小以至于可以忽略不计。这种思路的错误在于只注意到了单次异常事件发生的概率，而忽略了样本的大小。即使单次异常概率非常小，由于系统规模和运行时间的作用，事件样本将是一个非常大的值，从而使得异常事件实际发生的概率变大。例如，某系统中某种异常每天发生的概率为 10^{-9}，但系统每天处理的请求数为 10^8 次，每次请求都要执行有异常风险的流程，那么系统每天发生这种异常的概率就为$1-(1-10^{-9})^{10^8}=10\%$。

1.2 副本

1.2.1 副本的概念

副本（replica/copy）指在分布式系统中为数据或服务提供的冗余。对于数据副本指在不同的节点上持久化同一份数据，当出现某一个节点的存储的数据丢失时，可以从副本上读到数据。数据副本是分布式系统解决数据丢失异常的唯一手段。另一类副本是服务副本，指数个节点提供某种相同的服务，这种服务一般并不依赖于节点的本地存储，其所需数据一般来自其他节点。

例如，GFS 系统的同一个 chunk 的数个副本就是典型的数据副本，而 Map Reduce 系统的 Job Worker 则是典型的服务副本。

副本协议是贯穿整个分布式系统的理论核心，在后续章节中，本文将讨论在工程中广泛使用的各种副本协议。

1.2.2 副本一致性

分布式系统通过副本控制协议，使得从系统外部读取系统内部各个副本的数据在一定的约束条件下相同，称之为副本一致性(consistency)。副本一致性是针对分布式系统而言的，不是针对某一个副本而言。

例 1.3：某系统有 3 副本，某次更新数据完成了其中 2 个副本的更新，第 3 个副本由于异常而更新失败，此时仅有 2 个副本的数据是一致的，但该系统通过副本协议使得外部用户始终只读更新成功的第 1、2 个副本，不读第 3 个副本，从而对于外部用户而言，其独到的数据始终是一致的。

依据一致性的强弱即约束条件的不同苛刻程度，副本一致性分为若干变种或者级别，本文挑选几种常见的一致性级别介绍：

强一致性(strong consistency)：任何时刻任何用户或节点都可以读到最近一次成功更新的副本数据。强一致性是程度最高的一致性要求，也是实践中最难以实现的一致性。

单调一致性(monotonic consistency)：任何时刻，任何用户一旦读到某个数据在某次更新后的值，这个用户不会再读到比这个值更旧的值。单调一致性是弱于强一致性却非常实用的一种一致性级别。因为通常来说，用户只关心从己方视角观察到的一致性，而不会关注其他用户的一致性情况。

会话一致性(session consistency)：任何用户在某一次会话内一旦读到某个数据在某次更新后的值，这个用户在这次会话过程中不会再读到比这个值更旧的值。会话一致性通过引入会话的概念，在单调一致性的基础上进一步放松约束，会话一致性只保证单个用户单次会话内数据的单调修改，对于不同用户间的一致性和同一用户不同会话间的一致性没有保障。实践中有许多机制正好对应会话的概念，例如 php 中的 session 概念。可以将数据版本号等信息保存在 session 中，读取数据时验证副

本的版本号，只读取版本号大于等于 session 中版本号的副本，从而实现会话一致性。

最终一致性(eventual consistency)：最终一致性要求一旦更新成功，各个副本上的数据最终将达到完全一致的状态，但达到完全一致状态所需要的时间不能保障。对于最终一致性系统而言，一个用户只要始终读取某一个副本的数据，则可以实现类似单调一致性的效果，但一旦用户更换读取的副本，则无法保障任何一致性。

弱一致性(week consistency)：一旦某个更新成功，用户无法在一个确定时间内读到这次更新的值，且即使在某个副本上读到了新的值，也不能保证在其他副本上可以读到新的值。弱一致性系统一般很难在实际中使用，使用弱一致性系统需要应用方做更多的工作从而使得系统可用。

1.3 衡量分布式系统的指标

评价分布式系统有一些常用的指标。依据设计需求的不同，分布式系统对于这些指标也有不同的要求。

1.3.1 性能

无论是分布式系统还是单机系统，都会对性能(performance)有所要求。对于不同的系统，不同的服务，关注的性能不尽相同、甚至相互矛盾。常见的性能指标有：系统的吞吐能力，指系统在某一时间可以处理的数据总量，通常可以用系统每秒处理的总的数据量来衡量；系统的响应延迟，指系统完成某一功能需要使用的时间；系统的并发能力，指系统可以同时完成某一功能的能力，通常也用 QPS(query per second)来衡量。上述三个性能指标往往会相互制约，追求高吞吐的系统，往往很难做到低延迟；系统平均响应时间较长时，也很难提高 QPS。

1.3.2 可用性

系统的可用性(availability)指系统在面对各种异常时可以正确提供服务的能力。系统的可用性可以用系统停服务的时间与正常服务的时间的比例来衡量，也可以用某功能的失败次数与成功次数的比例来衡量。可用性是分布式的重要指标，衡量了系统的鲁棒性，是系统容错能力的体现。

1.3.3 可扩展性

系统的可扩展性(scalability)指分布式系统通过扩展集群机器规模提高系统性能（吞吐、延迟、并发）、存储容量、计算能力的特性。可扩展性是分布式系统的特有性质。分布式系统的设计初衷就是利用集群多机的能力处理单机无法解决的问题。然而，完成某一具体任务的所需要的机器数目即集群规模取决于系统的性能和任务的要求。当任务的需求随着具体业务不断提高时，除了升级系统的性能，另一个做法就是通过增加机器的方式扩展系统的规模。好的分布式系统总在追求“线性扩展性”，也就是使得系统的某一指标可以随着集群中的机器数量线性增长。

1.3.4 一致性

分布式系统为了提高可用性，总是不可避免的使用副本的机制，从而引发副本一致性的问题。根据具体的业务需求的不同，分布式系统总是提供某种一致性模型，并基于此模型提供具体的服务。正如 1.2.2 提到的，越是强的一致的性模型，对于用户使用来说使用起来越简单。例如通常我们总是希望某次更新后可以立刻读到最新的修改，如果成功更新后的数据依旧有可能不一致读到旧数据，那么用户就需要在写入数据时加入序列号等信息，并在读取数据时首先自行实现过滤去重后再使用数据。

2 分布式系统原理

2.1 数据分布方式

所谓分布式系统顾名思义就是利用多台计算机协同解决单台计算机所不能解决的计算、存储等问题。单机系统与分布式系统的最大的区别在于问题的规模，即计算、存储的数据量的区别。将一个单机问题使用分布式解决，首先要解决的就是如何将问题拆解为可以使用多机分布式解决，使得分布式系统中的每台机器负责原问题的一个子集。由于无论是计算还是存储，其问题输入对象都是数据，所以如何拆解分布式系统的输入数据成为分布式系统的基本问题，本文称这样的数据拆解为数据分布方式，在本节中介绍几种常见的数据分布方式，最后对这几种方式加以对比讨论。

2.1.1 哈希方式

哈希方式是最常见的数据分布方式，其方法是按照数据的某一特征计算哈希值，并将哈希值与机器中的机器建立映射关系，从而将不同哈希值的数据分布到不同的机器上。所谓数据特征可以是 key-value 系统中的 key，也可以是其他与应用业务逻辑相关的值。例如，一种常见的哈希方式是按数据属于的用户 id 计算哈希值，集群中的服务器按 0 到机器数减 1 编号，哈希值除以服务器的个数，结果的余数作为处理该数据的服务器编号。工程中，往往需要考虑服务器的副本冗余，将每数台（例如 3）服务器组成一组，用哈希值除以总的组数，其余数为服务器组的编号。图 2-1 给出了哈希方式分数据的一个例子，将数据按哈希值分配到 4 个节点上。

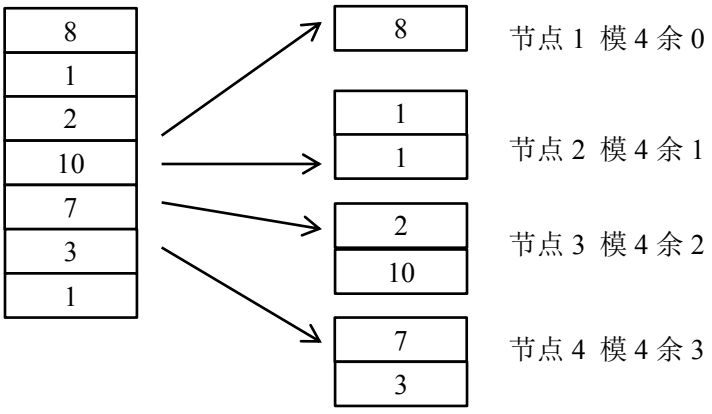


图 2-1 哈希方式分数据

可以将哈希方式想象为一个大的哈希表，每台（组）机器就是一个哈希表中的桶，数据根据哈希值被分布到各个桶上面。

只要哈希函数的散列特性较好，哈希方式可以较为均匀的将数据分布到集群中去。哈希方式需要记录的元信息也非常简单，任何时候，任何节点只需要知道哈希函数的计算方式及模的服务器总

数就可以计算出处理具体数据的机器是哪台。

哈希分布数据的缺点同样明显，突出表现为可扩展性不高，一旦集群规模需要扩展，则几乎所有的数据需要被迁移并重新分布。工程中，扩展哈希分布数据的系统时，往往使得集群规模成倍扩展，按照数据重新计算哈希，这样原本一台机器上的数据只需迁移一半到另一台对应的机器上即可完成扩展。

针对哈希方式扩展性差的问题，一种思路是不再简单的将哈希值与机器做除法取模映射，而是将对应关系作为元数据由专门的元数据服务器管理。访问数据时，首先计算哈希值并查询元数据服务器，获得该哈希值对应的机器。同时，哈希值取模个数往往大于机器个数，这样同一台机器上需要负责多个哈希取模的余数。在集群扩容时，将部分余数分配到新加入的机器并迁移对应的数据到新机器上，从而使得扩容不再依赖于机器数量的成本增长。这种做法和 2.1.2 中按数据范围分布数据、2.1.3 按数据量分布数据的有一个共同点，都需要以较复杂的机制维护大量的元数据。

哈希分布数据的另一个缺点是，一旦某数据特征值的数据严重不均，容易出现“数据倾斜”（data skew）问题。例如，某系统中以用户 id 做哈希分数据，当某个用户 id 的数据量异常庞大时，该用户的数据始终由某一台服务器处理，假如该用户的数据量超过了单台服务器处理能力的上限，则该用户的数据不能被处理。更为严重的是，无论如何扩展集群规模，该用户的数据始终只能由某一台服务器处理，都无法解决这个问题。图 2-2 给出了一个数据倾斜的例子，当使用用户 id 分数据，且用户 1 的数据特别多时，该用户的数据全部堆积到节点 2 上。

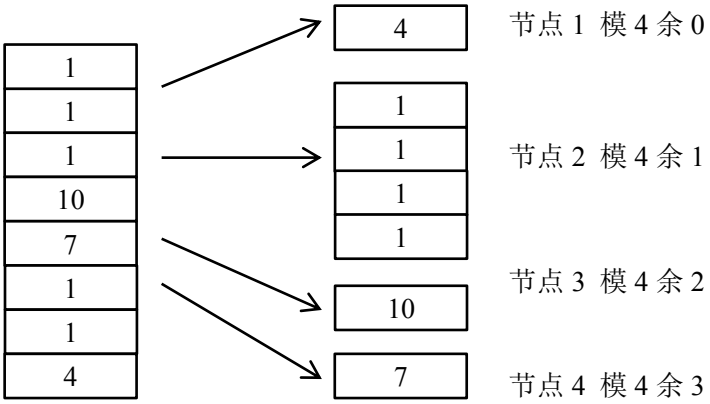


图 2-2 哈希方式的数据倾斜

在这种情况下只能重新选择需要哈希的数据特征，例如选择用户 id 与另一个数据维度的组合作为哈希函数的输入，如这样做，则需要完全重新分布数据，在工程实践中可操作性不高。另一种极端的思路是，使用数据的全部而不是某些维度的特征计算哈希，这样数据将被完全打散在集群中。然而实践中有时并不这样做，这是因为这样做使得每个数据之间的关联性完全消失，例如上述例子

中一旦需要处理某种指定用户 id 的数据，则需要所有的机器参与计算，因为一个用户 id 的数据可能分布到任何一台机器上。如果系统处理的每条数据之间没有任何逻辑上的联系，例如一个给定关键词的查询系统，每个关键词之间并没有逻辑上的联系，则可以使用全部数据做哈希的方式解决数据倾斜问题。

2.1.2 按数据范围分布

按数据范围分布是另一个常见的数据分布式，将数据按特征值的值域范围划分为不同的区间，使得集群中每台（组）服务器处理不同区间的数据。

例 2.1.1：已知某系统中用户 id 的值域范围是[1,100)，集群有 3 台服务器，使用按数据范围划分数据的数据分布方式。将用户 id 的值域分为三个区间[1, 33),，[33, 90)，[90, 100)分别由 3 台服务器负责处理。图 2-3 给出这个例子的示意图。

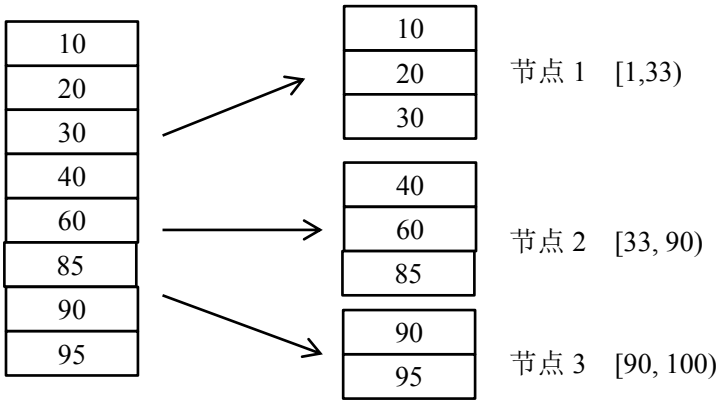


图 2-3 按数据范围分布

值得注意的是，每个数据区间的数据大小和区间大小是没有关系的。例如，上例中按用户 id 划分的三个区间，虽然从用户 id 的值域看来，不是等大小的，但三个区间中的数据量却有可能是差不多的。这是因为可能有的用户 id 的数据量大，有些用户 id 数据量小。也有可能某些区间中实际存在的用户 id 多，有些区间中实际存在的用户 id 少。工程中，为了数据迁移等负载均衡操作的方便，往往利用动态划分区间的技术，使得每个区间中服务的数据量尽量的一样多。当某个区间的数据量较大时，通过将区间“分裂”的方式拆分为两个区间，使得每个数据区间中的数据量都尽量维持在一个较为固定的阈值之下。

与哈希分布数据的方式只需要记录哈希函数及分桶个数（机器数）不同，按数据范围分布数据需要记录所有的数据分布情况。一般的，往往需要使用专门的服务器在内存中维护数据分布信息，称这种数据的分布信息为一种元信息。甚至对于大规模的集群，由于元信息的规模非常庞大，单台计算机无法独立维护，需要使用多台机器作为元信息服务器。

例 2.1.2：某分布式系统使用数据范围的方式分布数据，每个数据分区中保存 256MB 的数据，每个数据分区有 3 个副本。每台服务器有 10TB 的存储容量，集群规模为 1000 台服务器。每个数据分区需要 1KB 的元信息记录数据分区情况及副本所在的服务器。1000 台服务器的总存储量为 10000TB，总分区数为 10000TB/256MB = 40M，由于使用 3 副本，则独立分区数为 40M / 3 = 13M，需要的元信息量 13M * 1KB = 13GB，假设考虑到读写压力单个元数据服务器可以维护的元数据量为 2GB，则需要 7 台元数据服务器。

实际工程中，一般也不按照某一维度划分数据范围，而是使用全部数据划分范围，从而避免数据倾斜的问题。

哈希分布数据的方式使得系统中的数据类似一个哈希表。按范围分数据的方式则使得从全局看数据类似一个 B 树。每个具体的服务器都是 B 树的叶子节点，元数据服务器是 B 树的中间节点。

使用范围分布数据的方式的最大优点就是可以灵活的根据数据量的具体情况拆分原有数据区间，拆分后的数据区间可以迁移到其他机器，一旦需要集群完成负载均衡时，与哈希方式相比非常灵活。另外，当集群需要扩容时，可以随意添加机器，而不限为倍增的方式，只需将原机器上的部分数据分区迁移到新加入的机器上就可以完成集群扩容。

按范围分布数据方式的缺点是需要维护较为复杂的元信息。随着集群规模的增长，元数据服务器较为容易成为瓶颈，从而需要较为负责的多元数据服务器机制解决这个问题。

2.1.3 按数据量分布

另一类常用的数据分布方式则是按照数据量分布数据。与哈希方式和按数据范围方式不同，数据量分布数据与具体的数据特征无关，而是将数据视为一个顺序增长的文件，并将这个文件按照某一较为固定的大小划分为若干数据块（chunk），不同的数据块分布到不同的服务器上。与按数据范围分布数据的方式类似的是，按数据量分布数据也需要记录数据块的具体分布情况，并将该分布信息作为元数据使用元数据服务器管理。

由于与具体的数据内容无关，按数据量分布数据的方式一般没有数据倾斜的问题，数据总是被均匀切分并分布到集群中。当集群需要重新负载均衡时，只需通过迁移数据块即可完成。集群扩容也没有太大的限制，只需将部分数据库迁移到新加入的机器上即可以完成扩容。按数据量划分数据的缺点是需要管理较为复杂的元信息，与按范围分布数据的方式类似，当集群规模较大时，元信息的数据量也变得很大，高效的管理元信息成为新的课题。

2.1.4 一致性哈希

一致性哈希（consistent hashing）是另一个种在工程中使用较为广泛的数据分布方式。一致性哈希最初在 P2P 网络中作为分布式哈希表（DHT）的常用数据分布算法。一致性哈希的基本方式是使

用一个哈希函数计算数据或数据特征的哈希值，令该哈希函数的输出值域为一个封闭的环，即哈希函数输出的最大值是最小值的前序。将节点随机分布到这个环上，每个节点负责处理从自己开始顺时针至下一个节点的全部哈希值域上的数据。

例 2.1.3：某一致性哈希函数的值域为[0, 10)，系统有三个节点 A、B、C，这三个节点处于的一致性哈希的位置分别为 1，4，9，则节点 A 负责的值域范围为[1，4)，节点 B 负责的范围为[4, 9)，节点 C 负责的范围为[9, 10)和[0, 1)。若某数据的哈希值为 3，则该数据应由节点 A 负责处理。图 2-4 给出了这个例子的示意图。

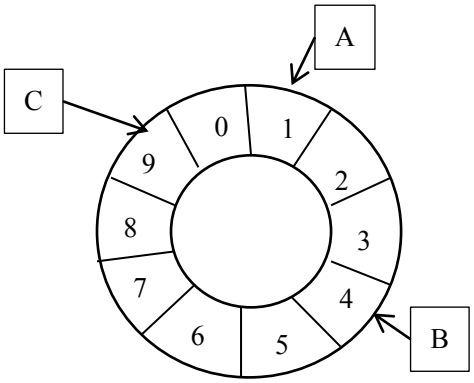


图 2-4 一致性哈希

哈希分布数据的方式在集群扩容时非常复杂，往往需要倍增节点个数，与此相比，一致性哈希的优点在于可以任意动态添加、删除节点，每次添加、删除一个节点仅影响一致性哈希环上相邻的节点。

例 2.1.4：假设需要在例 2.1.3 中增加一个新节点 D，为 D 分配的哈希位置为 3，则首先将节点 A 中[3, 4)的数据从节点 A 中拷贝到节点 D，然后加入节点 D 即可。

使用一致性哈希的方式需要将节点在一致性哈希环上的位置作为元信息加以管理，这点比直接使用哈希分布数据的方式要复杂。然而，节点的位置信息只于集群中的机器规模相关，其元信息的量通常比按数据范围分布数据和按数据量分布数据的元信息量要小很多。

上述最基本的一致性哈希算法有很明显的缺点，随机分布节点的方式使得很难均匀的分布哈希值域，尤其在动态增加节点后，即使原先的分布均匀也很难保证继续均匀，由此带来的另一个较为严重的缺点是，当一个节点异常时，该节点的压力全部转移到相邻的一个节点，当加入一个新节点时只能为一个相邻节点分摊压力。

为此一种常见的改进算法是引入虚节点（virtual node）的概念，系统初始时就创建许多虚节点，虚节点的个数一般远大于未来集群中机器的个数，将虚节点均匀分布到一致性哈希值域环上，其功

能与基本一致性哈希算法中的节点相同。为每个节点分配若干虚节点。操作数据时，首先通过数据的哈希值在环上找到对应的虚节点，进而查找元数据找到对应的真实节点。使用虚节点改进有多个优点。首先，一旦某个节点不可用，该节点将使得多个虚节点不可用，从而使得多个相邻的真实节点负载失效节点的压里。同理，一旦加入一个新节点，可以分配多个虚节点，从而使得新节点可以负载多个原有节点的压力，从全局看，较容易实现扩容时的负载均衡。

2.1.5 副本与数据分布

上述几节讨论了几种常见的数据分布方式，这些讨论中没有考虑数据副本的问题。分布式系统容错、提高可用性的基本手段就是使用副本。对于数据副本的分布方式主要影响系统的可扩展性。

一种基本的数据副本策略是以机器为单位，若干机器互为副本，副本机器之间的数据完全相同。这种策略适用于上述各种数据分布方式。其优点是非常简单，其缺点是恢复数据的效率不高、可扩展性也不高。图 2-5 给出了以机器为单位的副本例子，机器 1、2、3 互为副本，机器 4、5、6 互为副本，机器 7、8、9 互为副本。

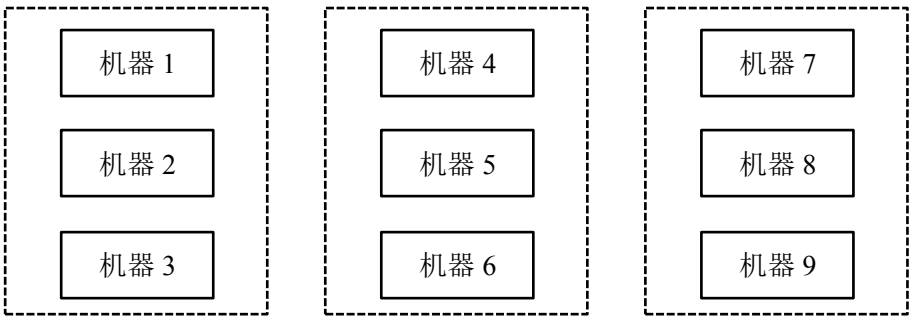


图 2-5 以机器为单位的副本

首先、使用该方式时，一旦出现副本数据丢失，需要恢复副本数据时效率不高。假设有 3 个副本机器，某时刻其中某台机器磁盘损坏，丢失了全部数据，此时使用新的机器替代故障机器，为了是的新机器也可以提供服务，需要从正常的两台机器上拷贝数据。此种全盘拷贝数据一般都较为消耗资源，为了不影响服务质量，实践中往往采用两种方式：一、将一台可用的副本机器下线，专门作为数据源拷贝数据，这样做的缺点是造成实际正常副本数只有 1 个，对数据安全性造成巨大隐患，且如果服务由于分布式协议设计或压力的要求必须 2 个副本才能正常工作，则该做法完全不可行。二、以较低的资源使用限速的方法从两个正常副本上拷贝数据，此方法不停服务，但可以选择服务压力较小的时段进行。该方法的缺点是速度较慢，如果需要恢复的数据量巨大（例如数 T），，限速较小（例如 10MB/s），往往需要数天才能够完成恢复。

再者、该种方式不利于提高系统扩展性。一个极端的例子是，假设系统原有 3 台机器，互为副本，现在如果只增加 2 台机器，由于 2 台机器无法组成新的副本组，则无法扩容。这里，为集群增

加了 66%的机器，却扩容失败。

最后、这种方式不利于系统容错。当出现一台机器宕机时，该机器上的原有压力只能被剩下的副本机器承担，假设有 3 个副本，宕机一台后，剩下两台的压力将增加 50%，有可能直接超过单台的处理能力。理想的情况是，若集群有 N 台机器，宕机一台后，该台机器的压力可以均匀分散到剩下的 N-1 台机器上，每台机器的压力仅仅增加 1/N-1。

更合适的做法不是以机器作为副本单位，而是将数据拆为较合理的数据段，以数据段为单位作为副本。实践中，常常使得每个数据段的大小尽量相等且控制在一定的大小以内。数据段有很多不同的称谓，segment，fragment，chunk，partition 等等。数据段的选择与数据分布方式直接相关。对于哈希分数据的方式，每个哈希分桶后的余数可以作为一个数据段，为了控制数据段的大小，常常使得分桶个数大于集群规模。例如，有 3 台服务器，10G 数据，为了使得每个数据段都是 100M 左右大小，哈希后按 100 取模，得 1000 个数据段，每台服务器可以负责 333 个数据段。对于按数据范围分布数据的方式，可以将每个数据区间作为一个数据段，并控制数据区间中数据的大小。对于按数据量分数据的方式，可以自然的按照每个数据块作为数据段。对于一致性哈希分布数据的方式，通常的做法是讲一致性哈希环分为若干等长分区，分区个数一般远大于节点个数，假设哈希函数均匀，则每个分区中的数据可以作为一个数据段。

一旦将数据分为数据段，则可以以数据段为单位管理副本，从而副本与机器不再硬相关，每台机器都可以负责一定数据段的副本。

例 2.1.5：某系统中的数据有 3 个数据段 o、p、q，每个数据段都有三个副本，系统中有 4 台机器，第一台机器上有数据段 o、p、q，第二台机器上有数据段 o、p，第三台机器上有数据段 p、q，第四台机器上有数据段 q、o。

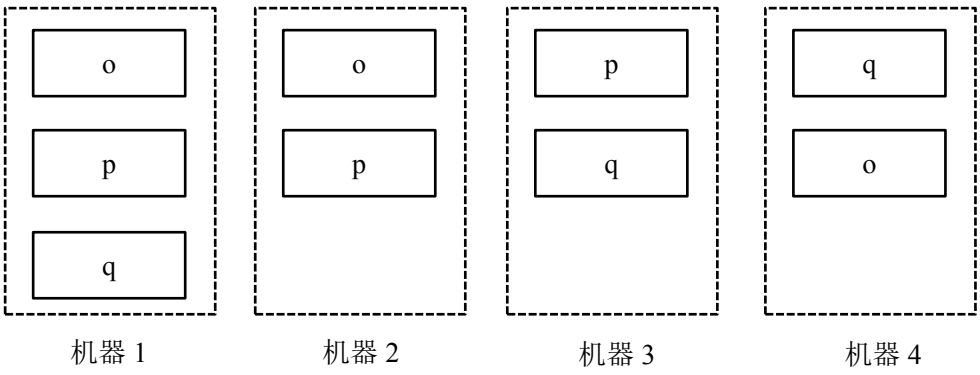


图 2-6 以数据段为单位的副本

一旦副本分布与机器无关，数据丢失后的恢复效率将非常高。这是因为，一旦某台机器的数据丢失，其上数据段的副本将分布在整个集群的所有机器中，而不是仅在几个副本机器中，从而可以从整个集群同时拷贝恢复数据，而集群中每台数据源机器都可以以非常低的资源做拷贝。作为恢复

数据源的机器即使都限速 1MB/s，若有 100 台机器参与恢复，恢复速度也能达到 100MB/s。再者，副本分布与机器无关也利于集群容错。如果出现机器宕机，由于宕机机器上的副本分散于整个集群，其压力也自然分散到整个集群。最后，副本分布与机器无关也利于集群扩展。理论上，设集群规模为 N 台机器，当加入一台新的机器时，只需从各台机器上迁移 1/N – 1/N+1 比例的数据段到新机器即实现了新的负载均衡。由于是从集群中各机器迁移数据，与数据恢复同理，效率也较高。

工程中，完全按照数据段建立副本会引起需要管理的元数据的开销增大，副本维护的难度也相应增大。一种折中的做法是将某些数据段组成一个数据段分组，按数据段分组为粒度进行副本管理。这样做可以将副本粒度控制在一个较为合适的范围内。

2.1.6 本地化计算

本节到此为止都是讨论的数据分布的方式，容易被理解为仅仅是对数据存储分布方式的讨论。对于分布式系统而言，除了解决大规模存储问题更需要解决大规模的计算问题。然而计算离不开数据，计算的规模往往与输入的数据量或者计算产生的中间结果的数据量正相关。在分布式系统中，数据的分布方式也深深影响着计算的分布方式。

在分布式系统中计算节点和保存计算数据的存储节点可以在同一台物理机器上，也可以位于不同的物理机器。如果计算节点和存储节点位于不同的物理机器则计算的数据需要通过网络传输，此种方式的开销很大，甚至网络带宽会成为系统的总体瓶颈。另一种思路是，将计算尽量调度到与存储节点在同一台物理机器上的计算节点上进行，这称之为本地化计算。本地化计算是计算调度的一种重要优化，其体现了一种重要的分布式调度思想：“移动数据不如移动计算”。

2.1.7 数据分布方式的选择

本节分析了各种常见的数据分布方式及这些方式的优缺点。在实际工程实践中，可以根据需求及实施复杂度合理选择数据分布方式。另外，上述数据分布方式是如果可以灵活组合使用，往往可以兼备各种方式的优点，收到较好的综合效果。

例 2.1.6：继续讨论 2.1.1 中提到的数据倾斜问题，在按哈希分数据的基础上引入按数据量分布数据的方式，解决该数据倾斜问题。按用户 id 的哈希值分数据，当某个用户 id 的数据量特别大时，该用户的数据始终落在某一台机器上。此时，引入按数据量分布数据的方式，统计用户的数据量，并按某一阈值将用户的数据切为多个均匀的数据段，将这些数据段分布到集群中去。由于大部分用户的数据量不会超过阈值，所以元数据中仅仅保存超过阈值的用户的数据段分布信息，从而可以控制元数据的规模。这种哈希分布数据方式与按数据量分布数据方式组合使用的方案，在某真实系统中使用，取得了较好的效果。

2.1.8 工程投影

几乎所有的分布式系统都会涉及到数据分布问题。这里列举了几个常见的分布式系统的数据分布方式如下。

GFS[9] & HDFS	按数据量分布
Map reduce[10]	按 GFS 的数据分布做本地化
Big Table[11] & HBase	按数据范围分布
PNUTS[14]	哈希方式/按数据范围分布（可选）
Dynamo[16] & Cassandra[17]	一致性哈希
Mola & Armor *[18]	哈希方式
Big Pipe *[18]	哈希方式
Doris *[18]	哈希方式与按数据量分布组合

2.2 基本副本协议

本节讨论基本的副本控制协议，着重分析两大类典型的副本控制协议。

副本控制协议指按特定的协议流程控制副本数据的读写行为，使得副本满足一定的可用性和一致性要求的分布式协议。副本控制协议要具有一定的对抗异常状态的容错能力，从而使得系统具有一定的可用性，同时副本控制协议要能提供一定一致性级别。由 CAP 原理（在 2.9 节详细分析）可知，要设计一种满足强一致性，且在出现任何网络异常时都可用的副本协议是不可能的。为此，实际中的副本控制协议总是在可用性、一致性与性能等各要素之间按照具体需求折中。

本文将副本控制协议分为两大类:“中心化(centralized)副本控制协议”和“去中心化(decentralized)副本控制协议”。

2.2.1 中心化副本控制协议

中心化副本控制协议的基本思路是由一个中心节点协调副本数据的更新、维护副本之间的一致性。图 2-7 给出了中心化副本协议的通用架构。中心化副本控制协议的优点是协议相对较为简单，所有的副本相关的控制交由中心节点完成。并发控制将由中心节点完成，从而使得一个分布式并发控制问题，简化为一个单机并发控制问题。所谓并发控制，即多个节点同时需要修改副本数据时，需要解决“写写”、“读写”等并发冲突。单机系统上常用加锁等方式进行并发控制。对于分布式并发控制，加锁也是一个常用的方法，但如果没有中心节点统一进行锁管理，就需要完全分布式化的锁系统，会使得协议非常复杂。中心化副本控制协议的缺点是系统的可用性依赖于中心化节点，当中心节点异常或与中心节点通信中断时，系统将失去某些服务（通常至少失去更新服务），所以中心化副本控制协议的缺点正是存在一定的停服务时间。

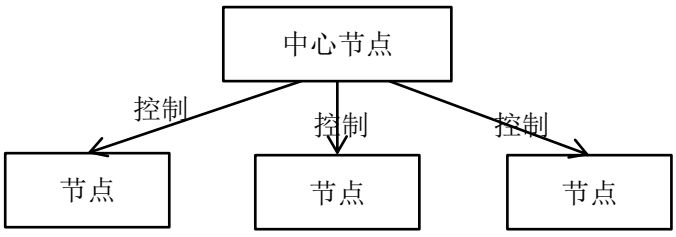


图 2-7 中心化副本控制

2.2.2 primary-secondary 协议

本文着重介绍一种非常常用的 primary-secondary（也称 primary-backup）的中心化副本控制协议。在 primary-secondary 类型的协议中，副本被分为两大类，其中有且仅有一个副本作为 primary 副本，除 primary 以外的副本都作为 secondary 副本。维护 primary 副本的节点作为中心节点，中心节点负责维护数据的更新、并发控制、协调副本的一致性。

Primary-secondary 类型的协议一般要解决四大类问题：数据更新流程、数据读取方式、Primary 副本的确定和切换、数据同步（reconcile）。以下依次分析。

2.2.2.1 数据更新基本流程

流程 2.2.1： Primary-secondary 协议的数据更新流程

1. 数据更新都由 primary 节点协调完成。
2. 外部节点将更新操作发给 primary 节点
3. primary 节点进行并发控制即确定并发更新操作的先后顺序
4. primary 节点将更新操作发送给 secondary 节点
5. primary 根据 secondary 节点的完成情况决定更新是否成功并将结果返回外部节点

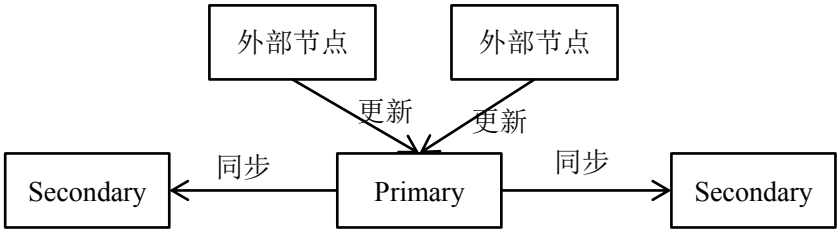


图 2-8 primary-secondary 基本更新流程

上述基本更新流程体现了 primary-secondary 协议的更新流程的基本思路。图 2-8 也给出了这个流程的基本示意图。其中第 4 步 primary 节点将更新操作发送到 secondary 节点时，往往发送的也是更新的数据。在工程实践中，如果由 primary 直接同时发送给其他 N 个副本发送数据，则每个 secondary 的更新吞吐受限于 primary 总的出口网络带宽，最大为 primary 网络出口带宽的 1/N。为了解决这个问题，有些系统（例如，GFS），使用接力的方式同步数据，即 primary 将更新发送给第一个 secondary 副本，第一个 secondary 副本发送给第二 secondary 副本，依次类推。由于异常，第 4 步可能在有些副本上成功，有些副本上失败，在有些副本上超时。不同的副本控制协议对于第 4 步异常的处理都不一样。例如，在提供最终一致性服务的系统中，secondary 节点可以与 primary 不一致，只要后续 secondary 节点可以慢慢同步到与 primary 一致的状态即可满足最终一致性的要求。对于第 4 步的具体处理，本节先不展开讨论，在 2.4 中介绍一种基于 Quorum 的副本控制机制。第 5 步中对于最终更新结果的处理也依赖于第 4 步的具体处理，这里同样先不展开讨论。

2.2.2.2 数据读取方式

数据读取方式是 primary-secondary 类协议需要解决的第二个问题。与数据更新流程类似，读取

方式也与一致性高度相关。如果只需要最终一致性，则读取任何副本都可以满足需求。如果需要会话一致性，则可以为副本设置版本号，每次更新后递增版本号，用户读取副本时验证版本号，从而保证用户读到的数据在会话范围内单调递增。使用 primary-secondary 比较困难的是实现强一致性。

这里简单讨论 primary-secondary 实现强一致性的几种思路。

第一、由于数据的更新流程都是由 primary 控制的，primary 副本上的数据一定是最新的，所以如果始终只读 primary 副本的数据，可以实现强一致性。如果只读 primary 副本，则 secondary 副本将不提供读服务。实践中，如果副本不与机器绑定，而是按照数据段为单位维护副本，仅有 primary 副本提供读服务在很多场景下并不会造出机器资源浪费。回忆 2.1.5 节，将数据分为数据段，以数据段为副本的基本单位，将副本分散到集群中个，假设 primary 也是随机的确定的，那么每台机器上都有一些数据的 primary 副本，也有另一些数据段的 secondary 副本。从而某台服务器实际都提供读写服务。

例 2.2.1，继续以例 2.1.5 说明，某系统中的数据有 3 个数据段 o、p、q，每个数据段都有三个副本，其中有一个 primary 副本，系统中有 4 台机器，第一台机器上有数据段 o(Primary)、p、q，第二台机器上有数据段 o、p(primary)，第三台机器上有数据段 p、o，第四台机器上有数据段 q(primary)、o。从这个例子可以看出，只要 primary 副本分散到集群中，即使只有 primary 副本提供读写服务，也可以充分利用集群机器资源。

第二、由 primary 控制节点 secondary 节点的可用性。当 primary 更新某个 secondary 副本不成功时，primary 将该 secondary 副本标记为不可用，从而用户不再读取该不可用的副本。不可用的 secondary 副本可以继续尝试与 primary 同步数据，当与 primary 完成数据同步后，primary 可以副本标记为可用。这种方式使得所有的可用的副本，无论是 primary 还是 secondary 都是可读的，且在一个确定的时间内，某 secondary 副本要么更新到与 primary 一致的最新状态，要么被标记为不可用，从而符合较高的一致性要求。这种方式依赖于一个中心元数据管理系统，用于记录哪些副本可用，哪些副本不可用。某种意义上，该方式通过降低系统的可用性来提高系统的一致性。

第三、基于 Quorum 机制，本文在 2.4 节中详细分析 Quorum 机制。这里不展开讨论。

2.2.2.3 primary 副本的确定与切换

在 primary-secondary 类型的协议中，另一个核心的问题是如何确定 primary 副本，尤其是在原 primary 副本所在机器出现宕机等异常时，需要有某种机制切换 primary 副本，使得某个 secondary 副本成为新的 primary 副本。

通常的，在 primary-secondary 类型的分布式系统中，哪个副本是 primary 这一信息都属于元信息，由专门的元数据服务器维护。执行更新操作时，首先查询元数据服务器获取副本的 primary 信息，从而进一步执行数据更新流程。

切换副本的难点在于两个方面，首先，如何确定节点的状态以发现原 primary 节点异常是一个较为复杂的问题。在 2.3 中，详细介绍一种基于 Lease 机制确定节点状态的方法。再者，切换 primary 后，不能影响副本的一致性。尤其是提供较强一致性服务的系统，切换 primary 的影响更是需要控制。要达到这个目的，一种直观的方式是切换的新 primary 的副本数据必须与原 primary 的副本一致。然而在原 primary 已经发送宕机等异常时，如何确定一个 secondary 副本使得该副本上的数据与原 primary 一致又成为新的问题。该问题和上节中选择一个 secondary 副本上读取最新的数据是一个等价问题。上节中本文在 2.4.5 介绍一种基于 Quorum 机制确定新 primary 的方法。

由于分布式系统中可靠的发现节点异常是需要一定的探测时间的，这样的探测时间通常是 10 秒级别（见 2.3.3 使用 lease 确定节点状态），这也意味着一旦 primary 异常，最多需要 10 秒级别的发现时间，系统才能开始 primary 的切换，在这 10 秒时间内，由于没有 primary，系统不能提供更新服务，如果系统只能读 primary 副本，则这段时间内甚至不能提供读服务。从这里可以看到，primary-backup 类副本协议的最大缺点就是由于 primary 切换带来的一定的停服务时间。

2.2.2.4 数据同步

Primary-secondary 型协议一般都会遇到 secondary 副本与 primary 不一致的问题。此时，不一致的 secondary 副本需要与 primary 进行同步（reconcile）。

通常不一致的形式有三种：一、由于网络分化等异常，secondary 上的数据落后于 primary 上的数据。二、在某些协议下，secondary 上的数据有可能是脏数据，需要被丢弃。所谓脏数据是由于 primary 副本没有进行某一更新操作，而 secondary 副本上反而进行的多余的修改操作，从而造成 secondary 副本数据错误。三、secondary 是一个新增加的副本，完全没有数据，需要从其他副本上拷贝数据。

对于第一种 secondary 数据落后的情况，常见的同步方式是回放 primary 上的操作日志（通常是 redo 日志），从而追上 primary 的更新进度。本文将在 2.5 节详细讨论日志技术。对于脏数据的情况，较好的做法是设计的分布式协议不产生脏数据。如果协议一定有产生脏数据的可能，则也应该使得产生脏数据的概率降到非常低得情况，从而一旦发生脏数据的情况可以简单的直接丢弃有脏数据的副本，这样相当于副本没有数据。另外，也可以设计一些基于 undo 日志的方式从而可以删除脏数据。如果 secondary 副本完全没有数据，则常见的做法是直接拷贝 primary 副本的数据，这种方法往往比回放日志追更新进度的方法快很多。但拷贝数据时 primary 副本需要能够继续提供更新服务，这就要求 primary 副本支持快照(snapshot)功能。即对某一刻的副本数据形成快照，然后拷贝快照，拷贝完成后使用回放日志的方式追快照形成后的更新操作。

2.2.3 去中心化副本控制协议

去中心化副本控制是另一类较为复杂的副本控制协议。与中心化副本系统协议最大的不同是，

去中心化副本控制协议没有中心节点，协议中所有的节点都是完全对等的，节点之间通过平等协商达到一致。从而去中心化协议没有因为中心化节点异常而带来的停服务等问题。图 2-9 给出了去中心化副本控制协议的示意图。

然而，没有什么事情是完美的，去中心化协议的最大的缺点是协议过程通常比较复杂。尤其当去中心化协议需要实现强一致性时，协议流程变得复杂且不容易理解。由于流程的复杂，去中心化协议的效率或者性能一般也较中心化协议低。一个不恰当的比方就是，中心化副本控制协议类似专制制度，系统效率高但高度依赖于中心节点，一旦中心节点异常，系统受到的影响较大；去中心化副本控制协议类似民主制度，节点集体协商，效率低下，但个别节点的异常不会对系统总体造成太大影响。

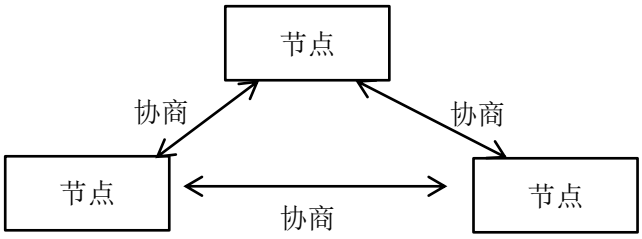


图 2-9 去中心化副本控制协议

与中心化副本控制协议具有某些共性不同，各类去中心化副本控制协议则各有各的巧妙。本节不再就去中心化副本控制协议做进一步详细分析。Paxos 是唯一在工程中得到应用的强一致性去中心化副本控制协议。本文在 2.8 节详细分析 paxos 协议。

2.2.4 工程投影

这里简要分析几种典型的分布式系统在副本控制协议方面的特点。工程中大量的副本控制协议都是 primary-secondary 型协议。从下面这些具体的分布式系统中不难看出，Primary-secondary 型副本控制虽然简单，但使用却极其广泛。

2.2.4.1 GFS 中的 Primary-Secondary 协议

GFS 系统的副本控制协议是典型的 Primary-Secondary 型协议，Primary 副本由 Master 指定，Primary 副本决定并发更新操作的顺序。虽然在 GFS 中，更新操作的数据由客户端提交，并在各个副本之间流式的传输，及由上一个副本传递到下一个副本，每个副本都即接受其他副本的更新，也向下更新另一个副本，但是数据的更新过程完全是由 primary 控制的，所以也可以认为数据是由 primary 副本同步到 secondary 副本的。

2.2.4.2 PNUTS 中的 Primary-Secondary 协议

PNUTS 的副本控制协议也是典型的 primary-secondary 型协议。Primary 副本负责将更新操作向 YMB 中提交，当更新记录写入 YMB 则认为更新成功。YMB 本身是一个分布式的消息发布、订阅系统，其具有多副本、高可用、跨地域等特性。Secondary 副本向 YMB 订阅 primary 发布的更新操作，当收到更新操作后，secondary 副本更新本地数据。从而，数据的更新过程也完全是由 primary 副本进行控制的。

2.2.4.3 Niobe 中的 Primary-Secondary 协议

Niobe 协议又是一个典型的 primary-secondary 型协议。Niobe 协议中，primary 信息由 GSM 模块维护，更新操作由 primary 副本同步到 secondary 副本。

2.2.4.4 Dynamo/Cassandra 的去中心化副本控制协议

Dynamo / Cassandra 使用基于一致性哈希的去中心化协议。虽然 Dynamo 尝试通过引入 Quorum 机制和 vector clock 机制解决读取数据的一致性问题，但其一致性模型依旧是一个较大的问题。由于缺乏较好的一致性，应用在编程时的难度被大大增加了。本文在 2.4.6 中较为详细的分析了 Dynamo 的一致性模型及其优缺点。

2.2.4.5 Chubby/Zookeeper 的副本控制协议

Chubby[13]和 Zookeeper 使用了基于 Paxos 的去中心化协议选出 primary 节点，但完成 primary 节点的选举后，这两个系统都转为中心化的副本控制协议，即由 primary 节点负责同步更新操作到 secondary 节点。本文在 2.8.6 中进一步分析这三个系统的工作原理。

2.2.4.6 Megastore 的副本控制协议

虽然都使用了 Paxos 协议，但与 Chubby 和 Zookeeper 不同的是，Megastore 中每次数据更新操作都基于一个改进的 Paxos 协议的实例，而不是利用 paxos 协议先选出 primary 后，再转为中心化的 primary-secondary 方式[12]。另一方面，Megastore 又结合了 Primary-secondary 本文在 2.8.6 中进一步分析 Megastore 使用 paxos 的工作原理。

2.2.4.7 其他系统的副本控制协议

Mola*/Armor*和 Big Pipe*也无一例外的使用了 Primary-secondary 协议控制副本。

2.3 Lease 机制

Lease 机制是最重要的分布式协议，广泛应用于各种实际的分布式系统中。即使在某些系统中相似的设计不被称为 lease，但我们可以分析发现其本质就是一种 lease 的实现。本节从一个分布式 cache 系统出发介绍最初的 lease 机制，接着加以引申，探讨 lease 机制的本质。最后介绍了 lease 机制最重要的应用：判定节点状态。

2.3.1 基于 lease 的分布式 cache 系统

本节先通过讨论一种分布式 cache 系统的实例来介绍 lease 机制。Lease 机制最初也是被运用于这种系统[4]。

基本的问题背景如下：在一个分布式系统中，有一个中心服务器节点，中心服务器存储、维护着一些数据，这些数据是系统的元数据。系统中其他的节点通过访问中心服务器节点读取、修改其上的元数据。由于系统中各种操作都依赖于元数据，如果每次读取元数据的操作都访问中心服务器节点，那么中心服务器节点的性能成为系统的瓶颈。为此，设计一种元数据 cache，在各个节点上 cache 元数据信息，从而减少对中心服务器节点的访问，提高性能。另一方面，系统的正确运行严格依赖于元数据的正确，这就要求各个节点上 cache 的数据始终与中心服务器上的数据一致，cache 中的数据不能是旧的脏数据。最后，设计的 cache 系统要能最大可能的处理节点宕机、网络中断等异常，最大程度的提高系统的可用性。

为此，利用 lease 机制设计一套 cache 系统，其基本原理为如下。中心服务器在向各节点发送数据时同时向节点颁发一个 lease。每个 lease 具有一个有效期，和信用卡上的有效期类似，lease 上的有效期通常是一个明确的时间点，例如 12:00:10，一旦真实时间超过这个时间点，则 lease 过期失效。这样 lease 的有效期与节点收到 lease 的时间无关，节点可能收到 lease 时该 lease 就已经过期失效。这里首先假设中心服务器与各节点的时钟是同步的，下节中讨论时钟不同步对 lease 的影响。中心服务器发出的 lease 的含义为：在 lease 的有效期内，中心服务器保证不会修改对应数据的值。因此，节点收到数据和 lease 后，将数据加入本地 Cache，一旦对应的 lease 超时，节点将对应的本地 cache 数据删除。中心服务器在修改数据时，首先阻塞所有新的读请求，并等待之前为该数据发出的所有 lease 超时过期，然后修改数据的值。

具体的服务器与客户端节点一个基本流程如下：

流程 2.3.1：基于 lease 的 cache，客户端节点读取元数据

1. 判断元数据是否已经处于本地 cache 且 lease 处于有效期内

   1.1 是：直接返回 cache 中的元数据

   1.2 否：向中心服务器节点请求读取元数据信息

1.2.1 服务器收到读取请求后，返回元数据及一个对应的 lease

1.2.2 客户端是否成功收到服务器返回的数据

1.2.2.1 失败或超时：退出流程，读取失败，可重试

1.2.2.2 成功：将元数据与该元数据的 lease 记录到内存中，返回元数据

流程 2.3.2：基于 lease 的 cache，客户端节点修改元数据流程

1. 节点向服务器发起修改元数据请求。
2. 服务器收到修改请求后，阻塞所有新的读数据请求，即接收读请求，但不返回数据。
3. 服务器等待所有与该元数据相关的 lease 超时。
4. 服务器修改元数据并向客户端节点返回修改成功。

上述机制可以保证各个节点上的 cache 与中心服务器上的中心始终一致。这是因为中心服务器节点在发送数据的同时授予了节点对应的 lease，在 lease 有效期内，服务器不会修改数据，从而客户端节点可以放心的在 lease 有效期内 cache 数据。上述 lease 机制可以容错的关键是：服务器一旦发出数据及 lease，无论客户端是否收到，也无论后续客户端是否宕机，也无论后续网络是否正常，服务器只要等待 lease 超时，就可以保证对应的客户端节点不会再继续 cache 数据，从而可以放心的修改数据而不会破坏 cache 的一致性。

上述基础流程有一些性能和可用性上的问题，但可以很容易就优化改性。优化点一：服务器在修改元数据时首先要阻塞所有新的读请求，造成没有读服务。这是为了防止发出新的 lease 从而引起不断有新客户端节点持有 lease 并缓存着数据，形成“活锁”。优化的方法很简单，服务器在进入修改数据流程后，一旦收到读请求则只返回数据但不颁发 lease。从而造成在修改流程执行的过程中，客户端可以读到元数据，只是不能缓存元数据。进一步的优化是，当进入修改流程，服务器颁发的 lease 有效期限选择为已发出的 lease 的最大有效期限。这样做，客户端可以继续在服务器进入修改流程后继续缓存元数据，但服务器的等待所有 lease 过期的时间也不会因为颁发新的 lease 而不断延长。实际使用中，第一层优化就足够了，因为等待 lease 超时的时间会被“优化点二”中的优化方法大大减少。优化点二：服务器在修改元数据时需要等待所有的 lease 过期超时，从而造成修改元数据的操作时延大大增大。优化的方法是，在等待所有的 lease 过期的过程中，服务器主动通知各个持有 lease 的节点放弃 lease 并清除 cache 中的数据，如果服务器收到客户端返回的确认放弃 lease 的消息，则服务器不需要在等待该 lease 超时。该过程中，如果因为异常造成服务器通知失败或者客户端节点发送应答消息失败，服务器只需依照原本的流程等待 lease 超时即可，而不会影响协议的正确性。

最后，我们分析一下 cache 机制与多副本机制的区别。Cache 机制与多副本机制的相似之处都

是将一份数据保存在多个节点上。但 Cache 机制却要简单许多，对于 cache 的数据，可以随时删除丢弃，并命中 cache 的后果仅仅是需要访问数据源读取数据；然而副本机制却不一样，副本是不能随意丢弃的，每失去一个副本，服务质量都在下降，一旦副本数下降到一定程度，则往往服务将不再可用。

2.3.2 lease 机制的分析

上节中介绍了一个 lease 的实例，本节进一步分析 lease 机制的本质。

首先给出本文对 lease 的定义：Lease 是由颁发者授予的在某一有效期内的承诺。颁发者一旦发出 lease，则无论接受方是否收到，也无论后续接收方处于何种状态，只要 lease 不过期，颁发者一定严守承诺；另一方面，接收方在 lease 的有效期内可以使用颁发者的承诺，但一旦 lease 过期，接收方一定不能继续使用颁发者的承诺。

由于 lease 是一种承诺，具体的承诺内容可以非常宽泛，可以是上节的例子中数据的正确性；也可以是某种权限，例如当需要做并发控制时，同一时刻只给某一个节点颁发 lease，只有持有 lease 的节点才可以修改数据；也可以是某种身份，例如在 primary-secondary(2.2.2)架构中，给节点颁发 lease，只有持有 lease 的节点才具有 primary 身份。Lease 的承诺的内涵还可以非常宽泛，这里不再一一列举。

Lease 机制具有很高的容错能力。首先，通过引入有效期，Lease 机制能否非常好的容错网络异常。Lease 颁发过程只依赖于网络可以单向通信，即使接收方无法向颁发者发送消息，也不影响 lease 的颁发。由于 lease 的有效期是一个确定的时间点，lease 的语义与发送 lease 的具体时间无关，所以同一个 lease 可以被颁发者不断重复向接受方发送。即使颁发者偶尔发送 lease 失败，颁发者也可以简单的通过重发的办法解决。一旦 lease 被接收方成功接受，后续 lease 机制不再依赖于网络通信，即使网络完全中断 lease 机制也不受影响。再者，Lease 机制能较好的容错节点宕机。如果颁发者宕机，则宕机的颁发者通常无法改变之前的承诺，不会影响 lease 的正确性。在颁发者机恢复后，如果颁发者恢复出了之前的 lease 信息，颁发者可以继续遵守 lease 的承诺。如果颁发者无法恢复 lease 信息，则只需等待一个最大的 lease 超时时间就可以使得所有的 lease 都失效，从而不破坏 lease 机制。例如上节中的 cache 系统的例子中，一旦服务器宕机，肯定不会修改元数据，重新恢复后，只需等待一个最大的 lease 超时时间，所有节点上的缓存信息都将被清空。对于接受方宕机的情况，颁发者不需要做更多的容错处理，只需等待 lease 过期失效，就可以收回承诺，实践中也就是收回之前赋予的权限、身份等。最后，lease 机制不依赖于存储。颁发者可以持久化颁发过的 lease 信息，从而在宕机恢复后可以使得在有效期的 lease 继续有效。但这对于 lease 机制只是一个优化，如之前的分析，即使颁发者没有持久化 lease 信息，也可以通过等待一个最大的 lease 时间的方式使得之前所有颁发的 lease 失效，从而保证机制继续有效。

Lease 机制依赖于有效期，这就要求颁发者和接收者的时钟是同步的。一方面，如果颁发者的时钟比接收者的时钟慢，则当接收者认为 lease 已经过期的时候，颁发者依旧认为 lease 有效。接收者可以用在 lease 到期前申请新的 lease 的方式解决这个问题。另一方面，如果颁发者的时钟比接收者的时钟快，则当颁发者认为 lease 已经过期的时候，接收者依旧认为 lease 有效，颁发者可能将 lease 颁发给其他节点，造成承诺失效，影响系统的正确性。对于这种时钟不同步，实践中的通常做法是将颁发者的有效期设置得比接收者的略大，只需大过时钟误差就可以避免对 lease 的有效性的影响。

2.3.3 基于 lease 机制确定节点状态

在分布式系统中确定一个节点是否处于正常工作状态是一个困难的问题。由于可能存在网络分化，节点的状态是无法通过网络通信来确定的。下面举一个较为具体的例子来讨论这个问题。

例 2.3.1：在一个 primary-secondary 架构的系统中，有三个节点 A、B、C 互为副本，其中有一个节点为 primary，且同一时刻只能有一个 primary 节点。另有一个节点 Q 负责判断节点 A、B、C 的状态，一旦 Q 发现 primary 异常，节点 Q 将选择另一个节点作为 primary。假设最开始时节点 A 为 primary，B、C 为 secondary。节点 Q 需要判断节点 A、B、C 的状态是否正常。

首先需要说明的是基于“心跳”(Heartbeat)的方法无法很好的解决这个问题。节点 A、B、C 可以周期性的向 Q 发送心跳信息，如果节点 Q 超过一段时间收不到某个节点的心跳则认为这个节点异常。这种方法的问题是假如节点 Q 收不到节点 A 的心跳，除了节点 A 本身的异常外，也有可能是因为节点 Q 与节点 A 之间的网络中断导致的。在工程实践中，更大的可能性不是网络中断，而是节点 Q 与节点 A 之间的网络拥塞造成的所谓“瞬断”，“瞬断”往往很快可以恢复。另一种原因甚至是节点 Q 的机器异常，以至于处理节点 A 的心跳被延迟了，以至于节点 Q 认为节点 A 没有发送心跳。假设节点 A 本身工作正常，但 Q 与节点 A 之间的网络暂时中断，节点 A 与节点 B、C 之间的网络正常。此时节点 Q 认为节点 A 异常，重新选择节点 B 作为新的 primary，并通知节点 A、B、C 新的 primary 是节点 B。由于节点 Q 的通知消息到达节点 A、B、C 的顺序无法确定，假如先到达 B，则在这一时刻，系统中同时存在两个工作中的 primary，一个是 A、另一个是 B。假如此时 A、B 都接收外部请求并与 C 同步数据，会产生严重的数据错误。上述即所谓“双主”问题，虽然看似这种问题出现的概率非常低，但在工程实践中，笔者不止一次见到过这样的情况发生。

上述问题的出现的原因在于虽然节点 Q 认为节点 A 异常，但节点 A 自己不认为自己异常，依旧作为 primary 工作。其问题的本质是由于网络分化造成的系统对于“节点状态”认知的不一致。

上面的例子中的分布式协议依赖于对节点状态认知的全局一致性，即一旦节点 Q 认为某个节点 A 异常，则节点 A 也必须认为自己异常，从而节点 A 停止作为 primary，避免“双主”问题的出现。解决这种问题有两种思路，第一、设计的分布式协议可以容忍“双主”错误，即不依赖于对节点状态的全局一致性认识，或者全局一致性状态是全体协商后的结果；第二、利用 lease 机制。对于第一

种思路即放弃使用中心化的设计，而改用去中心化设计，超过本节的讨论范畴。下面着重讨论利用 lease 机制确定节点状态。

由中心节点向其他节点发送 lease，若某个节点持有有效的 lease，则认为该节点正常可以提供服务。用于例 2.3.1 中，节点 A、B、C 依然周期性的发送 heart beat 报告自身状态，节点 Q 收到 heart beat 后发送一个 lease，表示节点 Q 确认了节点 A、B、C 的状态，并允许节点在 lease 有效期内正常工作。节点 Q 可以给 primary 节点一个特殊的 lease，表示节点可以作为 primary 工作。一旦节点 Q 希望切换新的 primary，则只需等前一个 primary 的 lease 过期，则就可以安全的颁发新的 lease 给新的 primary 节点，而不会出现“双主”问题。

在实际系统中，若用一个中心节点发送 lease 也有很大的风险，一旦该中心节点宕机或网络异常，则所有的节点没有 lease，从而造成系统高度不可用。为此，实际系统总是使用多个中心节点互为副本，成为一个小的集群，该小集群具有高可用性，对外提供颁发 lease 的功能。chubby 和 zookeeper 都是基于这样的设计。

2.3.4 lease 的有效期时间选择

Lease 的有效期虽然是一个确定的时间点，当颁发者在发布 lease 时通常都是将当前时间加上一个固定的时长从而计算出 lease 的有效期。如何选择 Lease 的时长在工程实践中是一个值得讨论的问题。如果 lease 的时长太短，例如 1s，一旦出现网络抖动 lease 很容易丢失，从而造成节点失去 lease，使得依赖 lease 的服务停止；如果 lease 的时长太大，例如 1 分钟，则一旦接受者异常，颁发者需要过长的时间收回 lease 承诺。例如，使用 lease 确定节点状态时，若 lease 时间过短，有可能造成网络瞬断时节点收不到 lease 从而引起服务不稳定，若 lease 时间过长，则一旦某节点宕机异常，需要较大的时间等待 lease 过期才能发现节点异常。工程中，常选择的 lease 时长是 10 秒级别，这是一个经过验证的经验值，实践中可以作为参考并综合选择合适的时长。

2.3.5 工程投影

本节介绍几个典型分布式系统中的运用 Lease 机制的情况。

2.3.5.1 GFS 中的 Lease

GFS 中使用 Lease 确定 Chuck 的 Primary 副本。Lease 由 Master 节点颁发给 primary 副本，持有 Lease 的副本成为 primary 副本。Primary 副本控制该 chuck 的数据更新流量，确定并发更新操作在 chuck 上的执行顺序。GFS 中的 Lease 信息由 Master 在响应各个节点的 HeartBeat 时附带传递（piggyback）。对于每一个 chuck，其上的并发更新操作的顺序在各个副本上是一致的，首先 master 选择 primary 的顺序，即颁发 Lease 的顺序，在每一任的 primary 任期内，每个 primary 决定并发更新的顺序，从而更新操作的顺序最终全局一致。当 GFS 的 master 失去某个节点的 HeartBeat 时，只

需待该节点上的primary chuck的Lease超时，就可以为这些chuck重新选择primary副本并颁发lease。

2.3.5.2 Niobe中的Lease

Niobe中虽然没有明确说明使用了Lease机制，但是通过分析可以发现，这是一个Lease机制。Niobe协议中的Lease与常见的由中间节点Master颁发给primary不太相同。Niobe协议中，也是通过Lease机制维持Primary副本的选择，不同的是Niobe中的Lease是由Secondary节点向Primary节点发送。

在Niobe协议中，有一个高可用的全局元数据服务节点称为GSM（global state manager）。GSM上的元信息有唯一地址的版本号（称为epoch），每次更新该元信息都必须附带提交之前读取到的版本号，并进行condition-write，即GSM会检验客户节点提交的版本号是否与当前的版本号相同，如果相同则允许提交更新操作，并递增版本号，否则更新失败，从而实现了元信息更新的全局顺序一致。

在Niobe协议中，每个Secondary副本都会给Primary副本发送Lease，这个Lease的含义是：在Lease时间内，本副本承认你是Primary节点。在Niobe协议中，一旦出现primary失去了某个secondary的lease，此时primary和secondary都会尝试去kill对方，secondary在kill对方的同时还会尝试成为新的primary。当primary失去某个secondary的lease后，primary会立刻尝试修改GSM中的元信息，将secondary在元信息中标记为“不可用”，从而kill掉该secondary，被kill掉的secondary只有重新与primary同步后才会被重新标记为“可用”并提供服务。而当某个secondary因不能与primary通信，造成无法给primary发送lease后，在lease超时后，也会尝试修改GSM中，将primary标记为“不可用”且将primary设置为自己。由于在GSM上更新操作靠epoch实现全局一致。Primary与secondary相互kill对方的操作有且仅有一个会成功，失败的那个副本需要重新读取GSM上的元数据后才能发起新的更新元数据的操作，而如果被对方 kill，那么重新读取元数据时会发现自己已经被设置为“不可用”从而无法再kill对方。

从这里我们可以看到，niobe中的lease含义也可以理解为：“我承诺在接下来lease时间内，我不会kill你”。这确实是一种另类的lease含义。

2.3.5.3 Chubby与Zookeeper中的Lease

Chubby中有两处使用到Lease机制。

我们知道chubby通过paxos协议实现去中心化的选择primary节点（见2.8.6 ）。一旦某个节点获得了超过半数的节点认可，该节点成为primary节点，其余节点成为secondary节点。Secondary节点向primary节点发送lease，该lease的含义是：“承诺在lease时间内，不选举其他节点成为primary节点”。只要primary节点持有超过半数节点的lease，那么剩余的节点就不能选举出新的primary。

一旦 primary 宕机，剩余的 secondary 节点由于不能向 primary 节点发送 lease，将发起新的一轮 paxos 选举，选举出新的 primary 节点。这种由 secondary 向 primary 发送 lease 的形式与 niobe 的 lease 形式有些类似。

除了 secondary 与 primary 之间的 lease，在 chubby 中，primary 节点也会向每个 client 节点颁发 lease。该 lease 的含义是用来判断 client 的死活状态，一个 client 节点只有只有合法的 lease，才能与 chubby 中的 primary 进行读写操作。一个 client 如果占有 chubby 中的一个节点锁后 lease 超时，那么这个 client 占有的 chubby 锁会被自动释放，从而实现了利用 chubby 对节点状态进行监控的功能。另外，chubby 中 client 中保存有数据的 cache，故此 chubby 的 primary 为 cache 的数据颁发 cache lease，该过程与 2.3.1 中介绍的基于 lease 的 cahce 机制完全类似。虽然相关文献上没有直接说明，但笔者认为，chubby 的 cache lease 与 primary 用于判断 client 死活状态的 lease 是可以合并为同一个 lease 的，从而可以简化系统的逻辑。

与 Chubby 不同，Zookeeper 中的 secondary 节点（在 zookeeper 中称之为 follower）并不向 primary 节点（在 zookeeper 中称之为 leader）发送 lease，zookeeper 中的 secondary 节点如果发现没有 primary 节点则发起新的 paxos 选举，只要 primary 与 secondary 工作正常，新发起的选举由于缺乏多数 secondary 的参与而不会成功。与 Chubby 类似的是，Zookeeper 的 primary 节点也会向 client 颁发 lease，lease 的时间正是 zookeeper 中的 session 时间。在 Zookeeper 中，临时节点是与 session 的生命期绑定的，当一个 client 的 session 超时，那么这个 client 创建的临时节点会被 zookeeper 自动删除。通过监控临时节点的状态，也可以很容易的实现对节点状态的监控。在这一点上，zookeeper 和 chubby 完全是异曲同工。

2.3.5.4 间接使用 Lease

笔者很难想象，如何在工程上既不使用 Lease 而又实现一个一致性较高的系统。直接实现 lease 机制的确会对增加系统设计的复杂度。然而，由于有类似 Zookeeper 这样的开源的高可用系统，在工程中完全可以间接使用 Lease。借助 zookeeper，我们可以简单的实现高效的、无单点选主、状态监控、分布式锁、分布式消息队列等功能，而实际上，这些功能的实现都是依赖于背后 zookeeper 与 client 之间的 Lease 的。

2.4 Quorum 机制

Quorum 机制是一种简单有效的副本管理机制。本节首先讨论一种最简单的副本控制规则 write-all-read-one，在此基础上，放松约束，讨论 quorum 机制。

2.4.1 约定

为了简化讨论，本节先做这样的约定：更新操作（write）是一系列顺序的过程，通过其他机制确定更新操作的顺序（例如 primary-secondary 架构中由 primary 决定顺序），每个更新操作记为 w_i，i 为更新操作单调递增的序号，每个 w_i 执行成功后副本数据都发生变化，称为不同的数据版本，记作 v_i。假设每个副本都保存了历史上所有版本的数据。

2.4.2 Write-all-read-one

Write-all-read-one（简称 WARO）是一种最简单的副本控制规则，顾名思义即在更新时写所有的副本，只有在所有的副本上更新成功，才认为更新成功，从而保证所有的副本一致，这样在读取数据时可以读任一副本上的数据。

假设有一种 magic 的机制，当某次更新操作 w_i 一旦在所有 N 个副本上都成功，此时全局都能知道这个信息，此后读取操作将指定读取数据版本为 v_i 的数据，称在所有 N 个副本上都成功的更新操作为“成功提交的更新操作”，称对应的数据为“成功提交的数据”。在 WARO 中，如果某次更新操作 w_i 在某个副本上失败，此时该副本的最新的数据只有 v_{i-1}，由于不满足在所有 N 个副本上都成功，则 w_i 不是一个“成功提交的更新操作”，此时，虽然其他 N-1 个副本上最新的数据是 v_i，但 v_i 不是一个“成功提交的数据”，最新的成功提交的数据只是 v_{i-1} 。

这里需要特别强调的是，在工程实践中，这种 magic 的机制往往较难实现或效率较低。通常实现这种 magic 机制的方式就是将版本号信息存放到某个或某组元数据服务器上。假如更新操作非常频繁，那么记录更新成功的版本号 v_i 的操作将成为一个关键操作，容易成为瓶颈。另外，为了实现强一致性，在读取数据的前必须首先读取元数据中的版本号，在大压力下也容易因为元数据服务器的性能造成瓶颈。

分析一下 WARO 的可用性。由于更新操作需要在所有的 N 个副本上都成功，更新操作才能成功，所以一旦有一个副本异常，更新操作失败，更新服务不可用。对于更新服务，虽然有 N 个副本，但系统无法容忍任何一个副本异常。另一方面，N 个副本中只要有一个副本正常，系统就可以提供读服务。对于读服务而言，当有 N 个副本时，系统可以容忍 N-1 个副本异常。

从上述分析可以发现 WARO 读服务的可用性较高，但更新服务的可用性不高，甚至虽然使用了副本，但更新服务的可用性等效于没有副本。

2.4.3 Quorum 定义

WARO 牺牲了更新服务的可用性，最大程度的增强读服务的可用性。下面将 WARO 的条件进行松弛，从而使得可以在读写服务可用性之间做折中，得出 Quorum 机制。

在 Quorum 机制下，当某次更新操作 w_i 一旦在所有 N 个副本中的 W 个副本上都成功，则就称该更新操作为“成功提交的更新操作”，称对应的数据为“成功提交的数据”。令 R>N-W，由于更新操作 w_i 仅在 W 个副本上成功，所以在读取数据时，最多需要读取 R 个副本则一定能读到 w_i 更新后的数据 v_i 。如果某次更新 w_i 在 W 个副本上成功，由于 W+R>N，任意 R 个副本组成的集合一定与成功的 W 个副本组成的集合有交集，所以读取 R 个副本一定能读到 w_i 更新后的数据 v_i。如图 2-10，Quorum 机制的原理可以文森图表示。

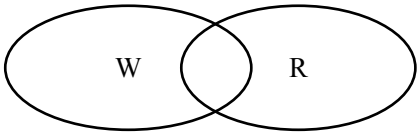


图 2-10 Quorum 机制

例 2.4.1，某系统有 5 个副本，W=3，R=3，最初 5 个副本的数据一致，都是 v_1，某次更新操作 w_2 在前 3 副本上成功，副本情况变成（v_2 v_2 v_2 v_1 v_1）。此时，任意 3 个副本组成的集合中一定包括 v_2。

在上述定义中，令 W=N，R=1，就得到 WARO，即 WARO 是 Quorum 机制的一种特例。

与分析 WARO 相似，分析 Quorum 机制的可用性。限制 Quorum 参数为 W+R=N+1。由于更新操作需要在 W 个副本上都成功，更新操作才能成功，所以一旦 N-W+1 个副本异常，更新操作始终无法在 W 个副本上成功，更新服务不可用。另一方面，一旦 N-R+1 个副本异常，则无法保证一定可以读到与 W 个副本有交集的副本集合，则读服务的一致性下降。

例 2.4.2：N=5，W=2，R=3 时，若 4 个副本异常，更新操作始终无法完成。若 3 个副本异常时，剩下的两个副本虽然可以提供更新服务，但对于读取者而言，在缺乏某些 magic 机制的，即如果读取者不知道当前最新已成功提交的版本是什么的时候，仅仅读取 2 个副本并不能保证一定可以读到最新的已提交的数据。

这里再次强调：仅仅依赖 quorum 机制是无法保证强一致性的。因为仅有 quorum 机制时无法确定最新已成功提交的版本号，除非将最新已提交的版本号作为元数据由特定的元数据服务器或元数据集群管理，否则很难确定最新成功提交的版本号。在下一节中，将讨论在哪些情况下，可以仅仅通过 quorum 机制来确定最新成功提交的版本号。

Quorum 机制的三个系统参数 N、W、R 控制了系统的可用性，也是系统对用户的服务承诺：数据最多有 N 个副本，但数据更新成功 W 个副本即返回用户成功。对于一致性要求较高的 Quorum 系统，系统还应该承诺任何时候不读取未成功提交的数据，即读取到的数据都是曾经在 W 个副本上成功的数据。

2.4.4 读取最新成功提交的数据

上节中，假设有某种 magic 的机制使得读取者知道当前已提交的数据版本号。本节取消这种假设，分析在 Quorum 机制下，如何始终读取成功提交的数据，以及如何确定最新的已提交的数据。

Quorum 机制只需成功更新 N 个副本中的 W 个，在读取 R 个副本时，一定可以读到最新的成功提交的数据。但由于有不成功的更新情况存在，仅仅读取 R 个副本却不一定能确定哪个版本的数据是最新的已提交的数据。对于一个强一致性 Quorum 系统，

例 2.4.3，在 N=5，W=3，R=3 的系统中，某时刻副本最大版本号为（v_2 v_2 v_2 v_1 v_1）。注意，这里继续假设有 v_2 的副本也有 v_1，上述列出的只是最大版本号。此时，最新的成功提交的副本应该是 v_2，因为从全局看 v_2 已经成功更新了 3 个副本。读取任何 3 个副本，一定能读到 v_2。但仅读 3 个副本时，有可能读到（v_2 v_1 v_1），如图 2-11（a）。此时，由于 v_2 蕴含 v_1，可知 v_1 是一个成功提交的版本，但却不能判定 v_2 一定是一个成功提交的版本。这是因为，图 2-11（b）,假设副本最大版本号为（v_2 v_1 v_1 v_1 v_1），当读取 3 个副本时也可能读到（v_2 v_1 v_1）,此时 v_2 是一个未成功提交的版本。所以在本例中，仅仅读到（v_2 v_1 v_1）时，可以肯定的是最新的成功提交的数据要么是 v_1 要么是 v_2，却没办法确定究竟是哪一个。

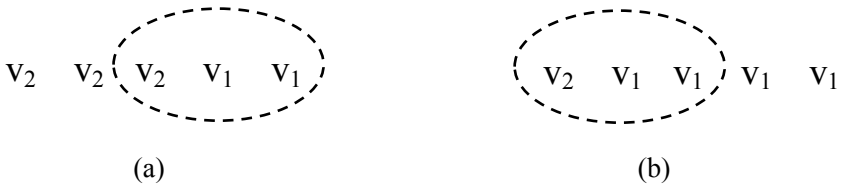


图 2-11 仅读 R 个副本无法判断最新已成功提交的数据

对于一个强一致性系统，应该始终读取返回最新的成功提交的数据，在 quorum 机制下，要达到这一目的需要对读取条件做进一步加强。

1. 限制提交的更新操作必须严格递增，即只有在前一个更新操作成功提交后才可以提交后一个更新操作，从而成功提交的数据版本号必须是连续增加的。

2. 读取 R 个副本，对于 R 个副本中版本号最高的数据，

2.1 若已存在 W 个，则该数据为最新的成功提交的数据

2.2 若存在个数据少于 W 个，假设为 X 个，则继续读取其他副本，直若成功读取到 W 个该版本的副本，则该数据为最新的成功提交的数据；如果在所有副本中该数据的个数肯定不满足 W 个，则 R 中版本号第二大的为最新的成功提交的副本。

例 2.4.4：依旧接例 2.4.3，在读取到（v_2 v_1 v_1）时，继续读取剩余的副本，若读到剩余两个副本为（v_2 v_2）则 v_2 是最新的已提交的副本；若读到剩余的两个副本为（v_2 v_1）或（v_1 v_1）则 v_1 是最新成功提交的版本；若读取后续两个副本有任一超时或失败，则无法判断哪个版本是最新的成功提交的版本。

可以看出，在单纯使用 Quorum 机制时，若要确定最新的成功提交的版本，最多需要读取 R+（W-R-1）=N 个副本，当出现任一副本异常时，读最新的成功提交的版本这一功能都有可能不可用。实际工程中，应该尽量通过其他技术手段，回避通过 Quorum 机制读取最新的成功提交的版本。例如，当 quorum 机制与 primary-secondary 控制协议结合使用时，可以通过读取 primary 的方式读取到最新的已提交的数据。

2.4.5 基于 Quorum 机制选择 primary

本节介绍一种介于 quorum 机制选择 primary 的技术。回忆 2.2.2 节，基本 primary-secondary 协议中，primary 负责进行更新操作的同步工作。现在基本 primary-secondary 协议中引入 quorum 机制，即 primary 成功更新 W 个副本(含 primary 本身)后向用户返回成功。读取数据时依照一致性要求的不同可以有不同的做法：如果需要强一致性的立刻读取到最新的成功提交的数据，则可以简单的只读取 primary 副本上的数据即可，也可以通过上节的方式读取；如果需要会话一致性，则可以根据之前已经读到的数据版本号在各个副本上进行选择性读取；如果只需要弱一致性，则可以选择任意副本读取。

在 primary-secondary 协议中，当 primary 异常时，需要选择出一个新的 primary，之后 secondary 副本与 primary 同步数据。通常情况下，选择新的 primary 的工作是由某一中心节点完成的，在引入 quorum 机制后，常用的 primary 选择方式与读取数据的方式类似，即中心节点读取 R 个副本，选择 R 个副本中版本号最高的副本作为新的 primary。新 primary 与至少 W 个副本完成数据同步后作为新的 primary 提供读写服务。首先，R 个副本中版本号最高的副本一定蕴含了最新的成功提交的数据。再者，虽然不能确定最高版本号的数是一个成功提交的数据，但新的 primary 在随后与 secondary 同步数据，使得该版本的副本个数达到 W，从而使得该版本的数据成为成功提交的数据。

例 2.4.5：在 N=5，W=3，R=3 的系统中，某时刻副本最大版本号为（v_2 v_2 v_1 v_1 v_1），此时 v_1 是系统的最新的成功提交的数据，v_2 是一个处于中间状态的未成功提交的数据。假设此刻原 primary 副本异常，中心节点进行 primary 切换工作。这类“中间态”数据究竟作为“脏数据”被删除，还是作为新的数据被同步后成为生效的数据，完全取决于这个数据能否参与新 primary 的选举。下面

分别分析这两种情况。

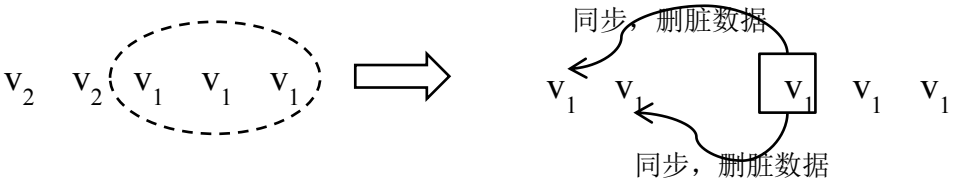
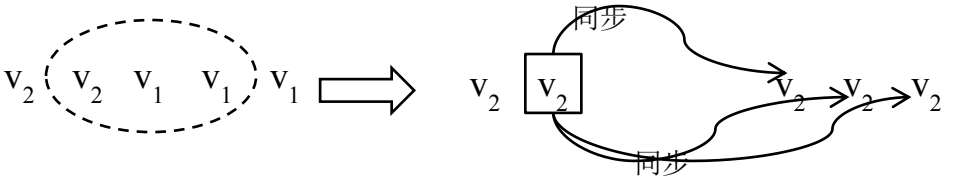


图 2-12 基于 quorum 选择 primary 情况 1

第一、如图 2-12，若中心节点与其中 3 个副本通信成功，读取到的版本号为（v_1 v_1 v_1），则任选一个副本作为primary，新 primary 以 v_1 作为最新的成功提交的版本并与其他副本同步，当与第 1、第 2 个副本同步数据时，由于第 1、第 2 个副本版本号大于 primary，属于脏数据，可以按照 2.2.2.4 节中介绍的处理脏数据的方式解决。实践中，新 primary 也有可能与后两个副本完成同步后就提供数据服务，随后自身版本号也更新到 v_2，如果系统不能保证之后的 v_2 与之前的 v_2 完全一样，则新 primary 在与第 1、2 个副本同步数据时不但要比较数据版本号还需要比较更新操作的具体内容是否一样。



第二、若中心节点与其他 3 个副本通信成功，读取到的版本号为（v_2 v_1 v_1），则选取版本号为 v2 的副本作为新的 primary，之后，一旦新 primary 与其他 2 个副本完成数据同步，则符合 v_2 的副本个数达到 W 个，成为最新的成功提交的副本，新 primary 可以提供正常的读写服务。

2.4.6 工程投影

2.4.6.1 GFS 中的 Quorum

GFS 使用 WARO 机制读写副本，即如果更新所有副本成功则认为更新成功，一旦更新成功，则可以任意选择一个副本读取数据；如果更新某个副本失败，则更显失败，副本之间处于不一致的状态。GFS 系统不保证异常状态时副本的一致性，GFS 系统需要上层应用通过 Checksum 等机制自行判断数据是否合法。值得注意的是 GFS 中的 append 操作，一旦 append 操作某个 chunck 的副本上失败，GFS 系统会自动新增一个 chunck 并尝试 append 操作，由于可以让新增的 chunck 在正常的机器上创建，从而解决了由于 WARO 造成的系统可用性下降问题。进而在 GFS 中，append 操作不保证一定在文件的结尾进行，由于在新增的 chunk 上重试 append，append 的数据可能会出现多份重复

的现象，但每个 append 操作会返回用户最终成功的 offset 位置，在这个位置上，任意读取某个副本一定可以读到写入的数据。这种在新增 chunk 上进行尝试的思路，大大增大了系统的容错能力，提高了系统可用性，是一种非常值得借鉴的设计思路。

2.4.6.2 Dynamo 中的 Quorum

Dynamo/Cassandra 是一种去中心化的分布式存储系统。Dynamo 使用 Quorum 机制来管理副本。用户可以配置 N、R、W 的参数，并保证满足 R+W>N 的 quorum 要求。与其他系统的 Quorum 机制类似，更新数据时，至少成功更新 W 个副本返回用户成功，读取数据时至少返回 R 个副本的数据。然而 Dynamo 是一个没有 primary 中的去中心化系统，由于缺乏中心控制，每次更新操作都可能由不同的副本主导，在出现并发更新、系统异常时，其副本的一致性完全无法得到保障。

下面着重分析 Dynamo 在异常时副本的一致性情况。首先，Dynamo 使用一致性哈希分布数据，理论上，即使出现一个节点异常，更新操作也可以顺着一致性哈希环的顺序找到 N 个节点完成。不过，这里我们简化其模型，认为始终只有初始的 N 个副本，在实际中可以等效为网络异常造成用户只能和初始的 N 个副本通信。更复杂的是，虽然可以沿哈希环找到下一个节点临时加入，但无法解决异常节点又重新加入的问题，所以这里的这种简化模型是完全合理的。我们通过一个例子来考察 Dynamo 的一致性。

例 2.4.6，在 Dynamo 系统中，N=3，R=2，W=2，初始时，数据 3 个副本（A、B、C）上的数据一致，这里假设数据值都为 1，即（1，1，1）。

某次更新操作需在原有数据的基础上增加新数据，这里假设为+1 操作，该操作由副本 A 主导，副本 A 成功更新自己及副本 C，由于异常，更新副本 B 失败，由于已经满足 W=2 的要求，返回用户更新成功。此时 3 个副本上的数据分别为（2，1，2）。

接着，进行新的更新操作，该操作需要在原有数据的基础上增加新数据，假设为+2 操作，假设用户端由于异常联系副本 A 失败，联系副本 B 成功，本次更新操作由副本 B 主导，副本 B 读取本地数据 1，完成加 2 操作后同步给其他副本，假设同步副本 C 成功，此时满足 W=2 的要求，返回用户更新成功。此时 3 个副本上的数据分别为(2, 3, 3)。这里需要说明的是在 Dynamo 中，副本 C 必须要接受副本 B 发过来的更新并覆盖自身数据，即使从全局角度说该更新与副本 C 上的已有数据是冲突的，但副本 C 自身无法判断自己的数据是否有效。假如第一次副本 A 主导的更新只在副本 C 上成功，那么此时副本 C 上的数据本身就是错误的脏数据，被副本 B 主导的这次更新覆盖也是完全应该的。

最后，用户读取数据，假设成功读取副本 A 及副本 B 上的数据，满足 R=2 的需求，用户将拿到两个完全不一致的数据 2 与 3，Dynamo 将解决这种不一致的情况留给了用户进行。

为了帮助用户解决这种不一致的情况，Dynamo 提出了一种 clock vector 的方法，该方法的思路

就是记录数据的版本变化，以类似 MVCC（2.7 ）的方式帮助用户解决数据冲突。所谓 clock vector 即记录了数据变化的路径的向量，为每个更新操作维护分配一个向量元素，记录数据的版本号及主导该次更新的副本名字。接着例 2.4.7 来介绍 clock vector 的过程。

例 2.4.8：在 Dynamo 系统中，N=3，R=2，W=2，初始时，数据 3 个副本（A、B、C）上的数据一致，这里假设数据值都为 1，即（1，1，1）,此时三个副本的 clock vector 都为空([], [], [])。

某次更新操作需在原有数据的基础上增加新数据，这里假设为+1 操作，该操作由副本 A 主导，副本 A 成功更新自己及副本 C，返回用户更新成功。此时 3 个副本上的数据分别为（2，1，2），而三个副本的 clock vector 为([(1, A)], [], [(1, A)])，A、C 的 clock vector 表示数据版本号为 1，更新是有副本 A 主导的。

接着，进行新的更新操作，该操作需要在原有数据的基础上增加新数据，假设为+2 操作，假设本次更新操作由副本 B 主导，副本 B 读取本地数据 1，完成加 2 操作后同步给其他副本，假设同步副本 C 成功，。此时 3 个副本上的数据分别为(2, 3, 3)，此时三个副本的 clock vector 为([(1, A)], [(1, B)], [(1, B)])。

为了说明 clock vector，这里再加入一次+3 操作，并由副本 A 主导，更新副本 A 及副本 C 成功，此时数据为（5, 3, 5），此时三个副本的 clock vector 为([(2, A), (1, A)], [(1, B)], [(2, A), (1, A)])。

最后，用户读取数据，假设成功读取副本 A 及副本 B 上的数据，得到两个完全不一致的数据 5 与 3，及这两个数据的版本信息[(2, A), (1, A)], [(1, B)]。用户可以根据自定义的策略进行合并，例如假设用户判断出，其实这些加法操作可以合并，那么最终的数据应该是 7，又例如用户可以选择保留一个数据例如 5 作为自己的数据。

由于提供了 clock vector 信息，不一致的数据其实成为了多版本数据，用户可以通过自定义策略选择合并这些多版本数据。Dynamo 建议可以简单的按照数据更新的时间戳进行合并，即用数据时间戳较新的数据替代较旧的数据。如果是简单的覆盖写操作，例如设置某个用户属性，这样的策略是有效且正确的。然而类似上例中这类并发的加法操作（例如“向购物车中增加商品”），简单的用新数据替代旧数据的方式就是不正确的，会造成数据丢失。

2.4.6.3 Zookeeper 中的 Quorum

Zookeeper 使用的 paxos 协议本身就是利用了 Quorum 机制，在 2.8 中有详细分析，这里不赘述。当利用 paxos 协议外选出 primary 后，Zookeeper 的更新流量由 primary 节点控制，每次更新操作，primary 节点只需更新超过半数（含自身）的节点后就返回用户成功。每次更新操作都会递增各个节点的版本号（xzid）。当 primary 节点异常，利用 paxos 协议选举新的 primary 时，每个节点都会以自己的版本号发起 paxos 提议，从而保证了选出的新 primary 是某个超过半数副本集合中版本号最大的副本。这个原则与 2.4.5 中描述的完全一致。值得一提的是，在 2.4.5 中分析到，新 primary 的版本

号未必是一个最新已提交的版本，可能是一个只更新了少于半数副本的中间态的更新版本，此时新 primary 完成与超过半数的副本同步后，这个版本的数据自动满足 quorum 的半数要求；另一方面，新 primary 的版本可能是一个最新已提交的版本，但可能会存在其他副本没有参与选举但持有一个大于新 primary 的版本号的数据（中间态版本），和 2.4.5 分析的一样，此时这样的中间态版本数据将被认为是脏数据，在与新 primary 进行数据同步时被 zookeeper 丢弃。

2.4.6.4 Mola*/Armor*中的 Quorum

Mola*和 Armor*系统中所有的副本管理都是基于 Quorum，即数据在多数副本上更新成功则认为成功。Mola 系统的读取通常不关注强一致性，而提供最终一致性。对于 Armor*，可以通过读取副本版本号的方式，按 Quorum 规则判断最新已提交的版本（参考 2.4.4 ）。由于每次读数据都需要读取版本号，降低了系统系统性能，Doris*系统在读取 Armor*数据时采用了一种优化思路：由于 Doris*系统的数据是批量更新，Doris*维护了 Armor*副本的版本号，并只在每批数据更新完成后再刷新 Armor*副本的版本号，从而大大减少了读取 Armor*副本版本号的开销。

2.4.6.5 Big Pipe*中的 Quorum

Big Pipe*中的副本管理也是采用了 WARO 机制。值得一提的是，Big Pipe 利用了 zookeeper 的高可用性解决了 WARO 在更新失败时副本的不一致的难题。当更新操作失败时，每个副本都会尝试将自己的最后一条更新操作写入 zookeeper。但最多只有一个副本能写入成功，如果副本发现之前已经有副本写入成功后则放弃写入，并以 zookeeper 中的记录为准与自身的数据进行同步。另一方面，与 GFS 更新失败后新建 chunck 类似，当 Big Pipe*更新失败后，会将更新切换到另一组副本，这组副本首先读取 zookeeper 上的最后一条记录，并从这条更新记录之后继续提供服务。

例如，一共有 3 个副本 A、B、C，某次更新操作在 A、B 上成功，各副本上的数据为（2，2，1），此时 3 个副本一旦探测出异常，都会尝试向 zookeeper 写入最后的记录。如果 A 或 B 写入成功，则意味着最后一次的更新操作成功，副本 C 会尝试同步到这条更新，新切换的副本组也会在数据 2 的基础上继续提供服务。如果 C 写入成功，则相当于最后一次更新失败，当副本 A、B 读到 zookeeper 上的信息后会将最后一个更新操作作为“脏数据”并回退掉最后一个更新操作，新切换的副本组也只会在数据 1 的基础上继续提供服务。

从这里不难看出，当出现更新失败时，最后一条更新操作成为类似 2.4.5 中的中间态数据。与 2.4.5 中的中间态数据的命运取决于是否能参与新 primary 的选举类似，Big Pipe*中这条中间态数据的命运完全取决于是哪个副本抢先完成写 zookeeper 的过程。

2.5 日志技术

日志技术是宕机恢复的主要技术之一[3]。日志技术最初使用在数据库系统中。严格来说日志技术不是一种分布式系统的技术，但在分布式系统的实践中，却广泛使用了日志技术做宕机恢复，甚至如 BigTable 等系统将日志保存到一个分布式系统中进一步增强了系统容错能力。

本章首先简单介绍数据库系统中的日志技术，进而抽象简化问题模型，在简化模型的基础上介绍两种实用的日志技术 Redo Log 与 No Redo/No undo Log。

2.5.1 数据库系统日志技术简述

在数据库系统中实现宕机恢复，其难点在于数据库操作需要满足 ACID，尤其在支持事务（transaction）的数据库系统中宕机往往发生在某些事务只执行了部分操作的时候。此时宕机恢复的主要目标就是数据库系统恢复到一个稳定可靠状态，消除未完成的事务对数据库状态的影响。

数据库的日志主要分为 Undo Log、Redo Log、Redo/Undo Log 与 No Redo/No Undo Log。这四类日志的区别在更新日志文件和数据文件的时间点要求不同，从而造成性能和效率也不相同。本文不就数据库中的这四类日志技术做深入讨论，相关信息可以参考有关数据库系统方面的资料。

2.5.2 Redo Log 与 Check point

2.5.2.1 问题模型

首先简化原数据库系统中的问题模型为一个较为简单的模型：假设需要设计一个高速的单机查询系统，将数据全部存放在内存中以实现高速的数据查询，每次更新操作更新一小部分数据（例如 key-value 中的某一个 key）。现在问题为利用日志技术实现该内存查询系统的宕机恢复。与数据库的事务不同的是，这个问题模型中的每个成功的更新操作都会生效。这也等效为数据库的每个事务只有一个更新操作，且每次更新操作都可以也必须立即提交（Auto commit）。

2.5.2.2 Redo Log

Redo Log 是一种非常简单实用的日志技术。在上节的问题模型中，只需按如下流程更新既可以实用 Redo Log。

流程 2.5.1：Redo Log 更新流程

1. 将更新操作的结果（例如 Set K1=1，则记录 K1=1）以追加写（append）的方式写入磁盘的日志文件

2. 按更新操作修改内存中的数据

3. 返回更新成功

上述更新流程中第 2 步没有考虑修改内存数据需要多线程互斥等问题，但对于说明 Redo Log 的原理没有影响。

从 Redo Log 的流程可以看出，Redo 写入日志的是更新操作完成后的结果（虽然本文不讨论 Undo Log，这点是与 Undo Log 的区别之一），且由于是顺序追加写日志文件，在磁盘等对顺序写有力的存储设备上效率较高。

用 Redo Log 进行宕机恢复非常简单，只需要“回放”日志即可。

流程 2.5.2：Redo Log 的宕机恢复

1. 从头读取日志文件中的每次更新操作的结果，用这些结果修改内存中的数据。

从 Redo Log 的宕机恢复流程也可以看出，只有写入日志文件的更新结果才能在宕机后恢复。这也是为什么在 Redo Log 流程中需要先更新日志文件再更新内存中的数据的原因。假如先更新内存中的数据，那么用户立刻就能读到更新后的数据，一旦在完成内存修改与写入日志之间发生宕机，那么最后一次更新操作无法恢复，但之前用户可能已经读取到了更新后的数据，从而引起不一致的问题。

2.5.2.3 Check point

宕机恢复流量的缺点是需要回放所有 redo 日志，效率较低，假如需要恢复的操作非常多，那么这个宕机恢复过程将非常漫长。解决这一问题的方法即引入 check point 技术。在简化的模型下，check point 技术的过程即将内存中的数据以某种易于重新加载的数据组织方式完整的 dump 到磁盘，从而减少宕机恢复时需要回放的日志数据。

流程 2.5.3：check point

1. 向日志文件中记录“Begin Check Point”

2. 将内存中的数据以某种易于重新加载的数据组织方式 dump 到磁盘上

3. 向日志文件中记录“End Check Point”

在 check point 流程中，数据可以继续按照流程 2.5.1 被更新，这段过程中新更新的数据可以 dump 到磁盘也可以不 dump 到磁盘，具体取决于实现。例如，check point 开始时 k1=v1，check point 过程中某次更新为 k1 = v2，那么 dump 到磁盘上的 k1 的值可以是 v1 也可以是 v2。

流程 2.5.4：基于 check point 的宕机恢复流程

1. 将 dump 到磁盘的数据加载到内存。

2. 从后向前扫描日志文件，寻找最后一个“End Check Point”日志。

3. 从最后一个“End Check Point”日志向前找到最近的一个“Begin Check Point”日志，并回放该日志之后的所有更新操作日志。

上述 check point 的方式依赖 redo 日志中记录的都是更新后的数据结果这一特征，所以即使 dump 的数据已经包含了某些操作的结果，重新回放这些操作的日志也不会造成数据错误。同一条日志可以重复回放的操作即所谓具有“幂等性”的操作。工程中，有些时候 Redo 日志无法具有幂等性，例如加法操作、append 操作等。此时，dump 的内存数据一定不能包括“begin check point”日志之后的操作。为此，有两种方法，其一是 check point 的过程中停更新服务，不再进行新的操作，另一种方法是，设计一种支持快照（snapshot）的内存数据结构，可以快速的将内存生成快照，然后写入 check point 日志再 dump 快照数据。至于如何设计支持快照的内存数据结构，方式也很多，例如假设内存数据结构维护 key-value 值，那么可以使用哈希表的数据结构，当做快照时，新建一个哈希表接收新的更新，原哈希表用于 dump 数据，此时内存中存在两个哈希表，查询数据时查询两个哈希表并合并结果。

2.5.3 No Undo/No Redo log

本节介绍另一种特殊的日志技术“No Undo/No Redo log”，这种技术也称之为“0/1 目录”(0/1 directory)。

本节介绍这种技术并不再使用上节的问题场景，而假设另一种问题场景：若数据维护在磁盘中，某批更新由若干个更新操作组成，这些更新操作需要原子生效，即要么同时生效，要么都不生效。

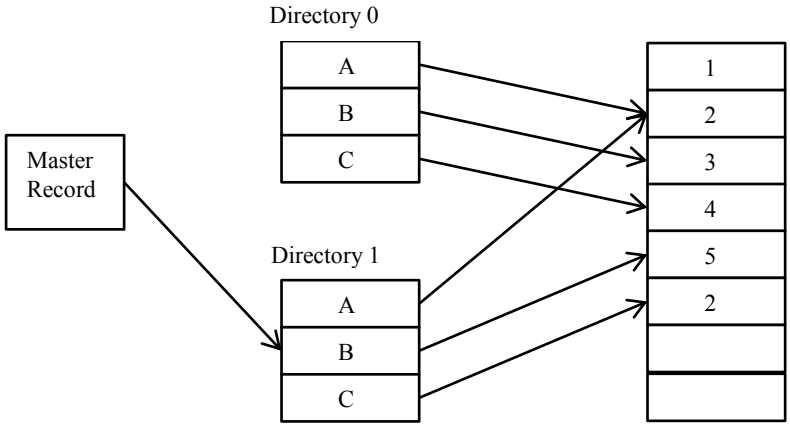


图 2-13 0/1 目录示例

0/1 目录技术中有两个目录结构，称为目录 0(Directory 0)和目录 1(Directory 1)。另有一个结构称为主记录（Master record）记录当前正在使用的目录称为活动目录。主记录中要么记录使用目录 0，要么记录使用目录 1。目录 0 或目录 1 中记录了各个数据的在日志文件中的位置。

图 1-1 给出了一个 0/1 目录的例子。活动目录为目录 1，数据有 A、B、C 三项。查目录 1 可得 A、B、C 三项的值分别为 2、5、2。

0/1 目录的数据更新过程始终在非活动目录上进行，只是在数据生效前，将主记录中的 0、1 值反转，从而切换主记录。

流程 2.5.5：0/1 目录数据更新流程

1. 将活动目录完整拷贝到非活动目录。

2. 对于每个更新操作，新建一个日志项纪录操作后的值，并在非活动目录中将相应数据的位置修改为新建的日志项的位置。

3. 原子性修改主记录：反转主记录中的值，使得非活动目录生效。

0/1 目录的更新流程非常简单，通过 0、1 目录的主记录切换使得一批修改的生效是原子的。

0/1 目录将批量事务操作的原子性通过目录手段归结到主记录的原子切换。由于多条记录的原子修改一般较难实现而单条记录的原子修改往往可以实现，从而降低了问题实现的难度。在工程中 0/1 目录的思想运用非常广泛，其形式也不局限在上述流程中，可以是内存中的两个数据结构来回切换，也可以是磁盘上的两个文件目录来回生效切换。

2.5.4 工程投影

日志技术的使用非常广泛，在 zookeeper 系统中，为了实现高效的数据访问，数据完全保存在内存中，但更新操作的日志不断持久化到磁盘，另一方面，为了实现较快速度的宕机恢复，zookeeper 周期性的将内存数据以 checkpoint 的方式 dump 到磁盘。

MySQL 的主从库设计也是基于日志。从库只需通过回放主库的日志，就可以实现与主库的同步。由于从库同步的速度与主库更新的速度没有强约束，这种方式只能实现最终一致性。

Mola*与 Armor*系统支持多种不同的存储引擎，对于接受到的更新操作，这两个系统将操作日志（redo log）保存到磁盘，引擎可以通过回放日志实现副本数据的同步。在 mola*中，由于不需要强一致性，日志与数据分离，且日志也保存多个副本，当日志副本更新满足 quorum 要求后就返回用户更新成功。引擎通过回放日志的方式实现数据更新，由于回放速度不一致，mola 提供最终一致性保证。同时，由于返回用户更新成功时只保证日志更新成功，此时读取引擎数据未必可以读到最新更新的数据。Armor*中更新了这一设计，日志与数据不分离，更新日志的同时也更新引擎数据，从而可以立刻读取到成功更新的数据。

2.6 两阶段提交协议

两阶段提交协议是一种经典的强一致性中心化副本控制协议[2][3]。虽然在工程中该协议有较多的问题，但研究该协议能很好的理解分布式系统的几个典型问题。

2.6.1 问题背景

两阶段提交（two phase commit）协议是一种历史悠久的分布式控制协议。最早用于在分布式数据库中，实现分布式事务。这里有必要首先简单介绍一下两阶段提交的最初问题背景，从而能更好的理解该协议。

在经典的分布式数据库模型中，同一个数据库的各个副本运行在不同的节点上，每个副本的数据要求完全一致。数据库中的操作都是事务(transaction)，一个事务是一系列读、写操作，事务满足ACID。每个事务的最终状态要么是提交(commit)，要么是失败(abort)。一旦一个事务成功提交，那么这个事务中所有的写操作中成功，否则所有的写操作都失败。在单机上，事务靠日志技术或 MVCC 等技术实现。在分布式数据库中，需要有一种控制协议，使得事务要么在所有的副本上都提交，要么在所有的副本上都失败。对同一个事务而言，虽然在所有副本上执行的事务操作都完全一样，但可能在某些副本上可以提交，在某些副本上不能提交。这是因为，在某些副本上，其他的事务可能与本事务有冲突（例如死锁），从而造成在有些副本上事务可以提交，而有些副本上事务无法提交。本文不再深入讨论事务冲突的问题，只是将问题背景介绍情况，该类问题可以通过阅读经典的数据库系统相关资料了解。

2.6.2 流程描述

按本文的分类，两阶段提交协议是一种典型的“中心化副本控制”协议。在该协议中，参与的节点分为两类：一个中心化协调者节点（coordinator）和 N 个参与者节点（participant）。每个参与者节点即上文背景介绍中的管理数据库副本的节点。

两阶段提交的思路比较简单，在第一阶段，协调者询问所有的参与者是否可以提交事务（请参与者投票），所有参与者向协调者投票。在第二阶段，协调者根据所有参与者的投票结果做出是否事务可以全局提交的决定，并通知所有的参与者执行该决定。在一个两阶段提交流程中，参与者不能改变自己的投票结果。两阶段提交协议的可以全局提交的前提是所有的参与者都同意提交事务，只要有一个参与者投票选择放弃(abort)事务，则事务必须被放弃。

协议流程如下：

流程 2.6.1：两阶段提交协调者流程

1. 写本地日志“begin_commit”， 并进入 WAIT 状态；

2. 向所有参与者发送“prepare 消息”；

3. 等待并接收参与者发送的对“prepare 消息”的响应；

3.1 若收到任何一个参与者发送的“vote-abort 消息”；

3.1.1 写本地“global-abort”日志，进入 ABORT；

3.1.2 向所有的参与者发送“global-abort 消息”；

3.1.3 进入 ABORT 状态；

3.2 若收到所有参与者发送的“vote-commit”消息；

3.2.1 写本地“global-commit”日志，进入 COMMIT 状态；

3.1.2 向所有的参与者发送“global-commit 消息”；

4. 等待并接收参与者发送的对“global-abort 消息”或“global-commit 消息”的确认响应消息，一旦收到所有参与者的确认消息，写本地“end_transaction”日志流程结束。

流程 2.6.2：两阶段提交协调者流程

1.写本地日志“init”记录，进入 INIT 状态

2.等待并接受协调者发送的“prepare 消息”，收到后

2.1 若参与者可以提交本次事务

2.1.1 写本地日志“ready”，进入 READY 状态

2.1.2 向协调者发送“vote-commit”消息

2.1.4 等待协调者的消息

2.1.4.1 若收到协调者的“global-abort”消息

2.1.4.1.1 写本地日志“abort”，进入 ABORT 状态

2.1.4.1.2 向协调者发送对“global-abort”的确认消息

2.1.4.2 若收到协调者的“global-commit”消息

2.1.4.1.1 写本地日志“commit”，进入 COMMIT 状态

2.1.4.1.2 向协调者发送对“global-commit”的确认消息

2.2 若参与者无法提交本次事务

2.2.1 写本地日志“abort”，进入 ABORT 状态

2.2.2 向协调者发送“vote-abort”消息

2.2.3 流程对该参与者结束

2.2.4 若后续收到协调者的“global-abort”消息可以响应

3. 即使流程结束，但任何时候收到协调者发送的“global-abort”消息或“global-commit”消息也都要发送一个对应的确认消息。

2.6.3 异常处理

2.6.3.1 宕机恢复

两阶段提交协议中，使用了日志技术从而在宕机后可以恢复流程状态。这里简单分析一下两阶段提交试用日志做宕机恢复的过程。

2.6.3.1.1 协调者宕机恢复

协调者宕机恢复后，首先通过日志查找到宕机前的状态。

如果日志中最后是“begin_commit”记录，说明宕机前协调者处于 WAIT 状态，协调者可能已经发送过“prepare 消息”也可能还没发送，但协调者一定还没有发送过“global-commit 消息”或“global-abort 消息”，即事务的全局状态还没有确定。此时，协调者可以重新发送“prepare 消息”继续两阶段提交流程，即使参与者已经发送过对“prepare 消息”的响应，也不过是再次重传之前的响应而不会影响协议的一致性。

如果日志中最后是“global-commit”或“global-abort”记录，说明宕机前协调者处于 COMMIT 或 ABORT 状态。此时协调者只需重新向所有的参与者发送“global-commit 消息”或“global-abort 消息”就可以继续两阶段提交流程。

2.6.3.1.2 参与者宕机恢复

参与者宕机恢复后，首先通过日志查找宕机前的状态。

如果日志中最后是“init”记录，说明参与者处于 INIT 状态，还没有对本次事务做出投票选择，参与者可以继续流程等待协调者发送的“prepare 消息”。

如果日志中最后是“ready”记录，说明参与者处于 REDAY 状态，此时说明参与者已经就本次事务做出了投票选择，但宕机前参与者是否已经向协调者发送“vote-commit”消息并不可知。所以此时参与者可以向协调者重发“vote-commit”，并继续协议流程。

如果日志中最后是“commit”或“abort”记录，说明参与者已经收到过协调者的“global-commit消息”（处于 COMMIT 状态）或者“global-abort 消息”（处于 ABORT 状态）。至于是否向协调者发送过对“global-commit”或“global-abort”的确认消息则未知。但即使没有发送过确认消息，由于协调者会不断重发“global-commit”或“global-abort”，只需在收到这些消息时发送确认消息既可，不影响协议的全局一致性。

2.6.3.2 响应超时

协议主要的异常最终会体现在流程中“等待消息”超时上，即等待了一个足量长的时间后，不能接收到需要的消息，使得流程无法进行下去。下面逐一分析这些超时的原因和对协议的影响。

2.6.3.2.1 协调者在 WAIT 状态超时

协调者在 WAIT 状态状态超时，即协调者等待参与者对“prepare 消息”的响应超时，在超时时间内始终不能收到所有的参与者的投票结果而收到的响应都是“vote-commit”消息，从而协调者无法确定该事务是否可以提交。这种超时可能的原因有：

1. 协调者与某个参与者网络中断，协调者的“prepare”消息无法发送到参与者，或者参与者的响应消息无法发送到协调者。

2. 参与者宕机，如果某个参与者宕机，则无法响应协调者的“prepare 消息”，只有等该参与者恢复后才能响应消息。

对于这种超时，协调者可以选择直接放弃整个事务，向所有参与者发送“global-abort”消息，进入 ABORT 状态。由于协调者在超时前并没有发送任何“global-abort”或者“global-commit”消息，所以协调者此时放弃事务不影响协议的一致性。

2.6.3.2.2 协调者在 COMMIT 或 ABORT 状态超时

协调者在 COMMIT 或 ABORT 状态超时，即协调者等待参与者对“global-commit”或“global-abort”消息的响应时超时，从而协调者无法确认两阶段提交是否完成。这种超时可能的原因有：

1. 协调者与某个参与者网络中断，协调者的“global-commit” 或“global-abort”消息无法发送到参与者，或者参与者的响应消息无法发送到协调者。

2. 参与者宕机，如果某个参与者宕机，则无法响应协调者的“global-commit”或“global-abort”，只有等该参与者恢复后才能响应消息。

对于这种超时，协调者只能不断重发“global-commit” 或“global-abort”消息给尚未响应的参与者，直到所有的参与者都发送响应。可以这么认为，两阶段提交协议对于这种超时的相关异常没有很好的容错机制，整个流程只能阻塞在这里，且流程状态处于未知。也许所有的参与者都完成了

各自的流程，只是由于协调者无法收到响应，整个两阶段提交协议就无法完成。

2.6.3.2.3 参与者在 INIT 状态超时

参与者等待协调者的“prepare”消息时超时，此种异常的原因可能是协调者宕机或者协调者与参与者网络中断。对于这种超时，参与者可以进入 ABORT 状态，这样即使后续收到了“prepare”消息，也不影响协议的一致性也不会阻塞其他流程，唯一的缺点是，该事务可能原本可以提交，现在却被放弃。

2.6.3.2.4 参与者在 READY 状态超时

参与者在 READY 状态等待协调者发送的“global-commit”或“global-abort”消息超时。出现这种超时的原因可能是协调者宕机也可能是网络中断。

因为参与者处于 READY 状态，说明参与者之前一定已经发送了“vote-commit”消息，从而参与者已经不能改变自己的投票选择。此时，参与者只能不断重发“vote-commit”消息，直到收到协调者的“global-commit”或“global-abort”消息后流程才可继续。可以这么认为，两阶段提交协议对于这种超时的相关异常也没有很好的容错机制，整个流程只能阻塞在这里，且对于参与者而言流程状态处于未知，参与者即不能提交本地节点上的事务，也不能放弃本地节点事务。

2.6.4 协议分析

两阶段提交协议在工程实践中真正使用的较少，主要原因有以下几点：

第一、两阶段提交协议的容错能力较差。从上文的分析可以看出，两阶段提交协议在某些情况下存在流程无法执行下去的情况，且也无法判断流程状态。在工程中好的分布式协议往往总是可以在即使发生异常的情况下也能执行下去。例如，回忆 Lease 机制（2.3 ），一旦 lease 发出，无论出现任何异常，Lease 服务器节点总是可以通过时间判定出 Lease 是否有效，也可以用等待 Lease 超时的方法收回 Lease 权限，整个 Lease 协议的流程不存在任何流程被阻塞而无法执行下去的情况。与 Lease 机制的简单有效相比，两阶段提交的协议显得较为复杂且容错能力差。

第二、两阶段提交协议的性能较差。一次成功的两阶段提交协议流程中，协调者与每个参与者之间至少需要两轮交互 4 个消息“prepare”、“vote-commit”、“global-commit”、“确认 global-commit”。过多的交互次数会降低性能。另一方面，协调者需要等待所有的参与者的投票结果，一旦存在较慢的参与者，会影响全局流程执行速度。

虽然存在一些改进的两阶段提交协议可以提高容错能力和性能，然而这类协议依旧是在工程中使用较少的一类协议，其理论价值大于实践意义。

2.7 基于 MVCC 的分布式事务

实现分布式事务除了使用类似“两阶段提交”协议等方式外，另一种简单高效的方式就是使用 MVCC(Multi-version Cocurrent Control，多版本并发控制)技术[3][5]。MVCC 技术最初也是在数据库系统中被提出，但这种思想并不局限于单机的分布式系统，在分布式系统中同样有效。

2.7.1 MVCC 简介

顾名思义，MVCC 即多个不同版本的数据实现并发控制的技术，其基本思想是为每次事务生成一个新版本的数据，在读数据时选择不同版本的数据即可以实现对事务结果的完整性读取。在使用 MVCC 时，每个事务都是基于一个已生效的基础版本进行更新，事务可以并行进行，从而可以产生一种图状结构。

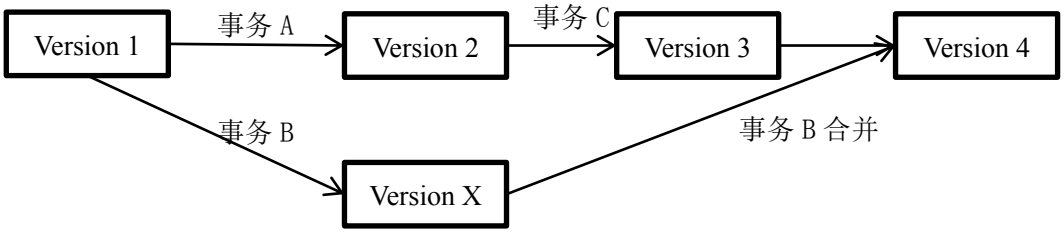


图 2-14 MVCC 示例

如图 2-14 所示，基础数据的版本为 1，同时产生了两个事务：事务 A 与事务 B。这两个事务都各自对数据进行了一些本地修改（这些修改只有事务自己可见，不影响真正的数据），之后事务 A 首先提交，生成数据版本 2；基于数据版本 2，又发起了事务 C，事务 C 继续提交，生成了数据版本 3；最后事务 B 提交，此时事务 B 的结果需要与事务 C 的结果合并，如果数据没有冲突，即事务 B 没有修改事务 A 与事务 C 修改过的变量，那么事务 B 可以提交，否则事务 B 提交失败。

MVCC 的流程过程非常类似于 SVN 等版本控制系统的流程，或者说 SVN 等版本控制系统就是使用的 MVCC 思想。

事务在基于基础数据版本做本地修改时，为了不影响真正的数据，通常有两种做法，一是将基础数据版本中的数据完全拷贝出来再修改，SVN 即使用了这种方法，SVN check out 即是拷贝的过程；二是每个事务中只记录更新操作，而不记录完整的数据，读取数据时再将更新操作应用到用基础版本的数据从而计算出结果，这个过程也类似 SVN 的增量提交。

2.7.2 分布式 MVCC

分布式 MVCC 的重点不在于并发控制，而在于实现分布式事务。这里首先给出一个简化的分布式事务的问题模型，之后对 MVCC 的讨论基于该问题展开。假设在一个分布式系统中，更新操作以事务进行，每个事务包括若干个对不同节点的不同更新操作。更新事务必须具有原子性，即事务中的所有更新操作要么同时在各个节点生效，要么都不生效。假设不存在并发的事务，即上一个事务成功提交后才进行下一个事务。

例如，用（site, k, op, oprd）表示在 site 节点上对变量 k 进行 op 操作，操作数为 oprd。那么一个典型的事务可能是{(site_A, var1, add, 10), (site_B, var2, sub, 1), (site_A, var3, set, 2)}，这个事务在 site_A 上将变量 var1 加 10，将变量 var3 设置为 2，在 site_B 上将变量 var2 减 1。

基于 MVCC 的分布式事务的方法为：为每个事务分配一个递增的事务编号，这个编号也代表了数据的版本号。当事务在各个节点上执行时，各个节点只需记录更新操作及事务编号，当事务在各个节点都完成后，在全局元信息中记录本次事务的编号。在读取数据时，先读取元信息中已成功的最大事务编号，再于各个节点上读取数据，只读取更新操作编号小于等于最后最大已成功提交事务编号的操作，并将这些操作应用到基础数据形成读取结果。

例 2.7.1：假设系统中有两个节点 A、B。节点 A、节点 B 状态如下表

节点	操作	事务序号
A	set var1 = 1	1
A	set var2 = 2	1
A	set var1 = var1 + 2	2
B	set var3 = 2	1
B	set var4 = 1	2

1. 若此时全局元信息中的最大的生效事务序号为 1，则在节点 A 上：var1=1，var2=2，在节点 B 上：var3 =2；

2. 若此时全局元信息中的最大的生效事务序号为 2，则在节点 A 上：var1= 1+2=3，var2=2，在节点 B 上：var3=2，var4=1；

从这个例子可以看出，每个节点上保存了对数据的更新操作，也就是数据的增量（delta），从而可以在读取数据时将应用不同的更新操作得出不同的数据版本。上例中，计算编号小于等于 1 的事务操作得出的数据即为版本号为 1 的数据，计算编号小于等于 2 的事务操作得出的数据即为版本号为 2 的数据。在新事务执行过程中，虽然更新操作已经逐步记录到各个节点，但只要全局元信息不修改，始终不会读到没有生效的事务数据，从而实现了全局一致性。另外，由于数据具有多个版本，可以自然实现对历史版本数据的读取。

上述方法的一个重要问题是，随着执行的事务越来越多，各个站点保存的更新操作会越来越多，

读取数据时需要应用的更新操作也越来越多。工程中可以对此周期性的启动合并操作，将历史上不再需要的版本合并为一个更新操作。例如，对例 2.7.1 中事务序号小于等于 2 的操作进行合并，合并后的节点状态如下表：

节点	操作	事务序号
A	set var1 = 3	2
A	set var2 = 2	2
B	set var3 = 2	2
B	set var4 = 1	2

这里合并后事务序号设置为合并使用的事务序号。如果节点中存在序号大于 2 的操作，则需要保留这些操作不参与合并。

2.7.3 工程投影

2.7.3.1 Megastore 中的 MVCC

Megastore 利用了 Big Table 中数据的多版本特性实现分布式的更新事务。每个事务更新的都是不同版本（timestamp）的 Big Table 数据，在读取数据时利用 timestamp 过滤，从而不会读到正在进行的尚未生效的事务数据。其原理与本节中介绍完全一致，不再赘述。

2.7.3.2 Doris*中的 MVCC

在 Doris*系统中，数据按批量进行更新，每个批量的数据都可以认为是一个事务，必须同时原子性的生效。为此，Doris*将每条数据附带了一个导入的版本号，在读取数据时根据元数据中已生效的版本号与数据上的导入版本号做过滤，从而不读取正在更新的尚未生效的数据，实现了分布式事务更新。其详细流量与本节中介绍的一致。

2.8 Paxos 协议

2.8.1 简介

Paxos 协议是少数在工程实践中证实的强一致性、高可用的去中心化分布式协议[6][7]。

Paxos 协议的流程较为复杂，但其基本思想却不难理解，类似于人类社会的投票过程。Paxos 协议中，有一组完全对等的参与节点（称为 accpetor），这组节点各自就某一事件做出决议，如果某个决议获得了超过半数节点的同意则生效。Paxos 协议中只要有超过一半的节点正常，就可以工作，能很好对抗宕机、网络分化等异常情况。

介绍 Paxos 协议的资料很多，Lamport 的论文也写得简明有趣。与大多数材料不同的是，本文不首先介绍协议的推理和证明过程，而是从工程上的算法流程描述起，感性的介绍协议过程。进而用一些复杂的例子演示协议的过程。最后，本文再介绍协议是如何推导设计出来的。

2.8.2 协议描述

2.8.2.1 节点角色

Paxos 协议中，有三类节点：

Proposer：提案者。Proposer 可以有多个，Proposer 提出议案（value）。所谓 value，在工程中可以是任何操作，例如“修改某个变量的值为某个值”、“设置当前 primary 为某个节点”等等。Paxos 协议中统一将这些操作抽象为 value。不同的 Proposer 可以提出不同的甚至矛盾的 value，例如某个 Proposer 提议“将变量 X 设置为 1”，另一个 Proposer 提议“将变量 X 设置为 2”，但对同一轮 Paxos 过程，最多只有一个 value 被批准。

Acceptor：批准者。Acceptor 有 N 个，Proposer 提出的 value 必须获得超过半数(N/2+1)的 Acceptor 批准后才能通过。Acceptor 之间完全对等独立。

Learner：学习者。Learner 学习被批准的 value。所谓学习就是通过读取各个 Proposer 对 value 的选择结果，如果某个 value 被超过半数 Proposer 通过，则 Learner 学习到了这个 value。回忆（2.4 ）不难理解，这里类似 Quorum 机制，某个 value 需要获得 W=N/2 + 1 的 Acceptor 批准，从而学习者需要至少读取 N/2+1 个 Accpetor，至多读取 N 个 Acceptor 的结果后，能学习到一个通过的 value。

上述三类角色只是逻辑上的划分，实践中一个节点可以同时充当这三类角色。

2.8.2.2 流程描述

Paxos 协议一轮一轮的进行，每轮都有一个编号。每轮 Paxos 协议可能会批准一个 value，也可能无法批准一个 value。如果某一轮 Paxos 协议批准了某个 value，则以后各轮 Paxos 只能批准这个

value。上述各轮协议流程组成了一个 Paxos 协议实例，即一次 Paxos 协议实例只能批准一个 value，这也是 Paxos 协议强一致性的重要体现。

每轮 Paxos 协议分为阶段，准备阶段和批准阶段，在这两个阶段 Proposer 和 Acceptor 有各自的处理流程。

流程 2.8.1：Proposer 的流程

（准备阶段）

1. 向所有的 Acceptor 发送消息“Prepare(b)”；　　这里 b 是 Paxos 的轮数，每轮递增

2. 如果收到任何一个 Acceptor 发送的消息“Reject(B)”，则对于这个 Proposer 而言本轮 Paxos 失败，将轮数 b 设置为 B+1 后重新步骤 1；

（批准阶段，根据收到的 Acceptor 的消息作出不同选择）

3. 如果接收到的 Acceptor 的“Promise(b, v_i)”消息达到 N/2+1 个（N 为 Acceptor 总数，除法取整，下同）；v_i 表示 Acceptor 最近一次在 i 轮批准过 value v。

3.1 如果收到的“Promise(b, v)”消息中，v 都为空，Proposer 选择一个 value v，向所有 Acceptor 广播 Accept(b, v)；

3.2 否则，在所有收到的“Promise(b, v_i)”消息中，选择 i 最大的 value v，向所有 Acceptor 广播消息 Accept(b，v)；

4. 如果收到 Nack(B)，将轮数 b 设置为 B+1 后重新步骤 1；

流程 2.8.2：Accpetor 流程

（准备阶段）

1. 接受某个 Propeser 的消息 Prepare(b)。

参数 B 是该 Acceptor 收到的最大 Paxos 轮数编号；V 是 Acceptor 批准的 value，可以为空

1.1 如果 b>B，回复 Promise(b, V_B)，设置 B=b; 表示保证不再接受编号小于 b 的提案。

1.2 否则，回复 Reject(B)

（批准阶段）

2. 接收 Accept(b, v)，

2.1 如果 b < B, 回复 Nack(B)，暗示 proposer 有一个更大编号的提案被这个 Acceptor 接收了

2.2 否则设置 V=v。表示这个 Acceptor 批准的 Value 是 v。广播 Accepted 消息。

2.8.3 实例

理解 Paxos 协议的最直观的方法就是考察几个协议运行的实例。本文给出几个典型的场景下协议运行的例子。

2.8.3.1 基本例子

基本例子里有 5 个 Acceptor，1 个 Proposer，不存在任何网络、宕机异常。我们着重考察各个 Accpetor 上变量 B 和变量 V 的变化，及 Proposer 上变量 b 的变化。

1. 初始状态

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	0	0	0	0	0
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

2. Proposer 向所有 Accpetor 发送“Prepare(1)”，所有 Acceptor 正确处理，并回复 Promise(1, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	NULL	NULL	NULL	NULL	NULL

	Proposer 1
b	1

3. Proposer 收到 5 个 Promise(1, NULL)，满足多余半数的 Promise 的 value 为空，此时发送 Accept(1, v_1)，其中 v_1 是 Proposer 选择的 Value。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	v_1	v_1	v_1	v_1	v_1

4. 此时，v_1 被超过半数的 Acceptor 批准，v_1 即是本次 Paxos 协议实例批准的 Value。如果 Learner 学习 value，学到的只能是 v_1

2.8.3.2 批准的 Value 无法改变

在同一个 Paxos 实例中，批准的 Value 是无法改变的，即使后续 Proposer 以更高的序号发起 Paxos 协议也无法改变 value。

1. 例如，某次 Paxos 协议运行后，Acceptor 的状态是：

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	2	2
V	v_1	v_1	v_1	NULL	NULL

5 个 Acceptor 中，有 3 个已经在第三轮 Paxos 协议批准了 v_1 作为 value。其他两个 Acceptor 的 V 为空，这可能是因为 Proposer 与这两个 Acceptor 的网络中断或者这两个 Acceptor 宕机造成的。

2. 此时，即使有新的 Proposer 发起协议，也无法改变结果。假设 Proposer 发送“prepare(4)消息”，由于 4 大于所有的 Accpetor 的 B 值，所有收到 prepare 消息的 Acceptor 回复 promise 消息。但前三个 Acceptor 只能回复 promise(4, v_1_3)，后两个 Acceptor 回复 promise(4, NULL)。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_1	v_1	v_1	NULL	NULL

3. 此时，Proposer 可能收到若干个 Acceptor 发送的 promise 消息，没有收到的 promise 消息可能是网络异常造成的。无论如何，Proposer 要收到至少 3 个 Acceptor 的 promise 消息后才满足协议中大于半数的约束，才能发送 accpet 消息。这 3 个 promise 消息中，至少有 1 个消息是 promise(4, v_1_3)，至多 3 个消息都是 promise(4，v_1_3)。另一方面，Proposer 始终不可能收到 3 个 promise(4, NULL)消息，最多收到 2 个。综上，按协议流程，Proposer 发送的 accept 消息只能是“accept(4, v_1)”而不能自由选择 value。

无论这个 accept 消息是否被各个 Acceptor 接收到，都无法改变 v_1 是被批准的 value 这一事实。即从全局看，有且只有 v_1 是满足超过多数 Acceptor 批准的 value。例如，假设 accept(4, v_1)消息被 Acceptor 1、Acceptor2、Acceptor4 收到，那么状态变为：

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_1	v_1	v_1	v_1	NULL

从这个例子我们可以看到一旦一个 value 被批准，此后永远只能批准这个 value。

2.8.3.3 一种不可能出现的状态

Paxos 协议的核心就在与“批准的 value 无法改变”，这也是整个协议正确性的基础，为了更好的理解后续对 Paxos 协议的证明。这里再看一种看似可能，实际违反协议的状态，这种状态也是后续反证法证明协议时的一种错误状态。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	2	2
V	v_1	v_1	v_1	v_2	v_2

上述状态中，3 个轮次为 1 的 Acceptor 的 value 为 v_1，2 个轮次更高的 Acceptor 的 value 为 v_1。此时被批准的 value 是 v_1。

假设此时发生新的一轮 b=3 的 Paxos 过程，Proposer 有可能收到 Acceptor 3、4、5 发出的 3 个 promise 消息分别为“promise(1, v_1_1)”, “promise(2, v_2_2)” “promise(2, v_2_2)”。按协议，proposer 选择 value 编号最大的 promise 消息，即 v_2_2 的 promise 消息，发送消息“Accept(3, v_2)”，从而使得最终的批准的 value 成为 v_2。就使得批准的 value 从 v_1 变成了 v_2。

上述假设看似正确，其实不可能发生。这是因为本节中给出的初始状态就是不可能出现的。这是因为，要到达成上述状态，发起 prepare(2)消息的 proposer 一定成功的向 Acceptor 4、Acceptor 5 发送了 accept(2, v_2)。但发送 accept(2, v_2)的前提只能是 proposer 收到了 3 个“promise(2, NULL)”消息。 然而，从状态我们知道，在 b=1 的那轮 Paxos 协议里，已经有 3 个 Acceptor 批准了 v_1,这 3 个 Acceptor 在 b=2 时发出的消息只能是 promise(2，v_1_1)，从而造成 proposer 不可能收到 3 个“promise(2, NULL)”，至多只能收到 2 个“promise(2, NULL)”。另外，只要 proposer 收到一个“promise(2，v_1_1)”，其发送的 accept 消息只能是 accept(2, v_1)。

从这个例子我们看到 Prepare 流程中的第 3 步是协议中最为关键的一步，它的存在严格约束了“批准的 value 无法改变”这一事实。在后续协议推导中我们将看到这一步是如何被设计出来的。

2.8.3.4 节点异常

这里给一个较为复杂的异常状态下 Paxos 运行实例。本例子中有 5 个 Acceptor 和 2 个 Proposer。

1. Proposer 1 发起第一轮 Paxos 协议，然而由于异常，只有 2 个 Acceptor 收到了 prepare(1)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	0	0	0
V	NULL	NULL	NULL	NULL	NULL

2. Proposer 1 只收到 2 个 promise 消息，无法发起 accept 消息；此时，Proposer 2 发起第二轮 Paxos 协议，由于异常，只有 Acceptor 1、3、4 处理了 prepare 消息，并发送 promise(2, NULL)消息

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	1	2	2	0
V	NULL	NULL	NULL	NULL	NULL

3. Proposer 2 收到了 Acceptor 1、3、4 的 promise(2, NULL) 消息，满足协议超过半数的要求，选择了 value 为 v_1,广播了 accept(2, v_1)的消息。由于异常，只有 Accptor 3、4 处理了这个消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	1	2	2	0
V	NULL	NULL	v_1	v_1	NULL

4. Proposer 1 以 b=3 发起新一轮的 Paxos 协议，由于异常，只有 Acceptor 1、2、3、5 处理了 prepare(3)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5

B	3	3	3	2	3
V	NULL	NULL	v_1	v_1	NULL

5. 由于异常，Proposer 1 只收到 Acceptor1、2、5 的 promise(3, NULL)的消息，符合协议要求，Proposer 1 选择 value 为 v_2，广播 accept(3, v_2)消息。由于异常，这个消息只被 Acceptor 1、2 处理。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	2	3
V	v_2	v_2	v_1	v_1	NULL

当目前为止，没有任何 value 被超过半数的 Acceptor 批准，所以 Paxos 协议尚没有批准任何 value。然而由于没有 3 个 NULL 的 Acceptor，此时能被批准的 value 只能是 v_1 或者 v_2 其中之一。

6. 此时 Proposer 1 以 b=4 发起新的一轮 Paxos 协议，所有的 Acceptor 都处理了 prepare(4)消息。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_2	v_2	v_1	v_1	NULL

7. 由于异常，Proposer 1 只收到了 Acceptor3 的 promise(4, v_1_3)消息、Acceptor4 的 promise(4, v_1_2)、Acceptor5 的 promise(4, NULL)消息，按协议要求，只能广播 accept(4, v_1)消息。假设 Acceptor2、3、4 收到了 accept(4, v_1)消息。由于批准 v_1 的 Acceptor 超过半数，最终批准的 value 为 v_1。

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	4	4	4	4	4
V	v_2	v_1	v_1	v_1	NULL

2.8.4 竞争及活锁

从前面的例子不难看出，Paxos 协议的过程类似于“占坑”，哪个 value 把超过半数的“坑”（Acceptor）占住了，哪个 value 就得到批准了。

这个过程也类似于单机系统并行系统的加锁过程。假如有这么单机系统：系统内有 5 个锁，有多个线程执行，每个线程需要获得 5 个锁中的任意 3 个才能执行后续操作，操作完成后释放占用的锁。我们知道，上述单机系统中一定会发生“死锁”。例如，3 个线程并发，第一个线程获得 2 个锁，第二个线程获得 2 个锁，第三个线程获得 1 个锁。此时任何一个线程都无法获得 3 个锁，也不会主动释放自己占用的锁，从而造成系统死锁。

但在 Paxos 协议过程中，虽然也存在着并发竞争，不会出现上述死锁。这是因为，Paxos 协议引入了轮数的概念，高轮数的 paxos 提案可以抢占低轮数的 paxos 提案。从而避免了死锁的发生。然而这种设计却引入了“活锁”的可能，即 Proposer 相互不断以更高的轮数提出议案，使得每轮 Paxos 过程都无法最终完成，从而无法批准任何一个 value。

1. Proposer 1 以 b=1 提起议案，发送 prepare(1)消息，各 Acceptor 都正确处理,回应 promise(1, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	1	1	1	1	1
V	NULL	NULL	NULL	NULL	NULL

2. Proposer 2 以 b=2 提起议案，发送 prepare(2)消息，各 Acceptor 都正确处理,回应 promise(2, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	2	2	2	2	2
V	NULL	NULL	NULL	NULL	NULL

3. Proposer 1 收到 5 个 promise(1, NULL)消息，选择 value 为 v_1 发送 accept(1, v_1)消息，然而这个消息被所有的 Acceptor 拒绝，收到 5 个 Nack(2)消息。

4. Proposer 1 以 b=3 提起议案，发送 prepare(3)消息，各 Acceptor 都正确处理,回应 promise(3, NULL)

	Acceptor 1	Acceptor 2	Acceptor 3	Acceptor 4	Acceptor 5
B	3	3	3	3	3
V	NULL	NULL	NULL	NULL	NULL

5. Proposer 2 收到 5 个 promise(2, NULL)消息，选择 value 为 v_2 发送 accept(2, v_2)消息，然而这个消息被所有的 Acceptor 拒绝，收到 5 个 Nack(3)消息。

上述过程交替进行，则永远无法批准一个 value，从而形成 Paxos 协议活锁。Paxos 协议活锁问题也是这个协议的主要问题。

2.8.5 协议推导

Paxos 协议是被人为设计出来，其设计过程也是协议的推导过程。Paxos 协议利用了 Quorom 机制，选择的 W=R=N/2+1。简单而言，协议就是 Proposer 更新 Acceptor 的过程，一旦某个 Acceptor 成功更新了超过半数的 Acceptor，则更新成功。Learner 按 Quorum 去读取 Acceptor，一旦某个 value 在超过半数的 Proposer 上被成功读取，则说明这是一个被批准的 value。协议通过引入轮次，使得高轮次的提议抢占低轮次的提议来避免死锁。

协议设计关键点是如何满足“在一次 Paxos 算法实例过程中只批准一个 Value”这一约束条件。称这个约束条件为“约束条件 1”。由于直接在工程中实现“约束条件 1”很困难，所以协议的设计过程就是不断推导出“约束条件 1”的必要不充分条件，直到某个必要不充分条件在工程上易于实现。从而满足这个条件也就能满足“约束条件 P1”。

一、提出“约束条件 2：一旦一个 value 获得超过半数的 Acceptor 批准，之后 Paxos 协议只能批准这个 value”，易证明“约束条件 2”=>“约束条件 1”。

二、加强该约束“约束条件 2”，寻找其必要不充分条件，提出“约束条件 3：一旦一个 value

获得超过半数的 Acceptor 批准，之后任何 Acceptor 只能批准这个 value”。容易证明“约束条件 3”=>“约束条件 2”

三、既然“约束条件 3”中要使得“之后任何 Acceptor 只能批准这个 value”那么等价于“之后 Proposer 发送的 accept 消息也只能是这个 value”。所以“约束条件 3”等价于“约束条件 4：一旦一个 value v 获得超过半数的 Acceptor 批准，之后 Proposer 提议的 value 只能是 v”。

四、加强“约束条件 4”，得到 Paxos 协议中的 Proposer 流程的“步骤 3”，即“约束条件 5：Proposer 提议一个 value v 前，要么之前没有任何一个 value 被批准，要么存在一个大小为 N/2+1 的 Acceptor 集合，这个集合内的各个 Acceptor 批准过的轮数最大的 value 是 v。”

“约束条件 5”的前半部分“提议 value v 前，没有任何一个 value 被批准”所以选择任意 value 提案一定不违背“约束条件 4”。确定“之前没有任何一个 value 被批准”的方法就是读取 Acceptor，如果有超过半数的 Acceptor 批准的 value 为空，那么肯定没有一个 value 被批准过。这也就是 Proposer 流程“步骤 3.1”。

“约束条件 5”的后半部分对应 Proposer 流程“步骤 3.2”，即读取了 N/2+1 的 Acceptor 的状态，这些 Acceptor 批准了某些 value，由于没有读取所有的 Acceptor，故可能无法确定是否一定有 value 已经被批准了。例如，5 个 Acceptor 时读取了 3 个 Acceptor 状态情况是（B=1, V=v_1）(B=2, V=v_2) (B=3, V=NULL)，v_2 可能是一个已经被批准的 value，是否已经被批准取决于另外两个 Acceptor 上的状态，如果另外两个 Acceptor 都批准了 v_2,则 v_2 是一个已经被批准的 value，如果另外两个 Acceptor 上的 value 为 NULL，那么 Paxos 协议还没有批准任何 value。“约束条件 5”的做法是在这种情况下选择一个轮数号大的 value 即 v_2，从而可以保证：要么此时 Paxos 协议还没有批准任何一个 value，要么之前 Paxos 协议批准的也只能是 v_2。

下面证明“约束条件 5”的后半部分=> “约束条件 4”：

1. 如果之前 Paxos 尚没有批准任何 value，那么选择轮次编号最大的 value 提案显然是“安全的”不违反“约束条件 4”。

2. 假设之前 Paxos 已经批准过一个 value，记为 v_0。下面证明在任意第 n 轮 paxos 过程中，v_0 一定是此时任意一个 N/2+1 的 Acceptor 集合中轮数最大的 value。

令 v_0 是在第 m 轮被批准的，则在 m 轮时至少有 N/2+1 个 Acceptor 的状态是（B=m，V=v_0）

在第 m+1 轮，由于 Quorum 限制，任意一个 N/2+1 个 Acceptor 组成的集合中，至少有 1 个 Acceptor 的状态是（B=m+1，V=v_0）,即 Proposer 至少收到一个 promise(m+1, v_0_m)消息，由于此刻 m 必然是所有 value 中最大的编号，所以 Proposer 按“约束条件 5”发出只能提案 value 只能是 v_0。

在第 m+1 轮、m+2 轮… n-1 轮，按递推规则，这些轮提案的 value 也只能是 v_0。

进一步反证法证明第 n 轮：假设在第 n 轮中，提案的 value 不是 v_0 而是 v_x。根据“约束条件 5”，因为 v_x 不是 v_0,那么 v_x 只能是一个没有被超过半数 Acceptor 批准过的 value，说明在 n-1 轮存在一个 N/2+1 的集合，该集合内所有的 value 都没有批准过 v_0。这个与从 m 轮到 n-1 轮提案的 value 是 v_0 相矛盾。至此已经归纳证明了“约束条件 5”的后半部分。

2.8.6 工程投影

2.8.6.1 Chubby 中的 Paxos

Chubby 是最早基于 Paxos 的分布式系统之一。Chubby 的设计人员没有直接提供一种 Paxos 的开发库，而是利用 Paxos 实现一个高可用的分布式系统，再利用这个分布式系统对外提供高可用存储、分布式锁等服务，从而间接的提供了 Paxos 功能。

Chubby 中的节点完全是对等的，通过 Paxos 协议，这些节点选举出一个 Master 节点（Primary），公开的资料中没有解释 Chubby 使用 Paxos 的细节，例如如何选择 Paxos 的轮次号，如何避免 Paxos 活锁等。当选举出 Primary 节点后，所有读写操作都由 Primary 节点控制，Chubby 系统从一个完全对等的去中心化状态变为一个 Primary-Secondary 的中心化状态。当 Primary 异常时，Chubby 节点将重新利用 paxos 协议发起新一轮的选举以确定新的 primary 节点。新 primary 节点与原 primary 节点具有完全一样的持久化信息，新 primary 将代替原 primary 节点对外提供读写服务。

基于 Chubby 的服务，其他的分布式系统可以很容易的实现选择 primary、保存最核心元数据等功能。利用 Chubby 可以大大简化分布式系统的设计：可以认为整个 Chubby 集群逻辑上是一个 magic 的高可用（几乎不会停服务）的中心节点，其他分布式系统可以基于这个大中心节点可以实现中心化的副本控制协议。由于 Chubby 集群本身是由多个节点组成的分布式系统，基于 Chubby 的分布式系统无需直接实现 Paxos 协议，就可以利用 Paxos 协议实现全局完全无单点。

2.8.6.2 Zookeeper 中的 Paxos

Zookeeper 使用了一种修改后的 Paxos 协议。

首先，Zookeeper 的协议运行依赖 TCP 协议实现 FIFO，Zookeeper 通过 TCP 协议获得两点保障：1、数据总是严格按照 FIFO（first in first out）规则从一个节点传递到另一个节点的；2、当某个 TCP 链接关闭后，这个链接上不再有数据传递。由于 TCP 协议为传输的每一个字节设置了序列号(sequence number)及确认(acknowledgment)，上述两点在 TCP 协议上是完全可以保证的。需要注意的是 Zookeeper 并不要求 TCP 协议可以可靠的将数据传输到对端节点，正如本文在 1.1.4.2.4 分析过的，基于 TCP 协议实现真正意义上的可靠传输也是做不到的。Zookeeper 基于 TCP 的上述两点保障，可以较大的简化问题模型，忽略诸如网络消息乱序、网络消息重复等的异常，从而较大的简化协议设计。

再者，在 Zookeeper 中，始终分为两种场景：一、Leader activation，在这个场景里，系统中缺乏 Leader（primary），通过一个类似 paxos 协议的过程完成 Leader 选举。二、Active messaging，在这个场景里，Leader 接收客户端发送的更新操作，以一种类似两阶段提交的过程在各个 follower（secondary）节点上进行更新操作。在 Leader activation 场景中完成 leader 选举及数据同步后，系统转入 Active messaging 场景，在 active messaging 中 leader 异常后，系统转入 Leader activation 场景。

无论在那种场景，Zookeeper 依赖于一个全局版本号：zxid。zxid 由(epoch, count)两部分组成，高位的 epoch 部分是选举编号，每次提议进行新的 leader 选举时 epoch 都会增加，低位的 count 部分是 leader 为每个更新操作决定的序号。可以认为，一个 leader 对应一个唯一的 epoch，每个 leader 任期内产生的更新操作对应一个唯一的有序的 count，从而从全局的视野，一个 zxid 代表了一个更新操作的全局序号（版本号）。

每个 zookeeper 节点都有各自最后 commit 的 zxid，表示这个 zookeeper 节点上最近成功执行的更新操作，也代表了这个节点的数据版本。在 Leader activation 阶段，每个 zookeeper 节点都以自己的 zxid 作为 Paxos 中的 b 参数发起 paxos 实例，设置自己作为 leader（此为 value）。每个 zookeeper 节点既是 proposer 又是 acceptor，所以，每个 zookeeper 节点只会 accpet 提案编号 b 大于自身 zxid 的提案。不难理解，通过 paxos 协议过程，某个超过 quorum 半数的节点中持有最大的 zxid 的节点会成为新的 leader。值得注意的是，假如参与选举的每个 zookeeper 节点的 zxid 都一样，即所有的节点都以相同的 b=zxid 发提案，那么就有可能发送类似 2.8.4 中无法选举出 leader 的情况。zookeeper 解决这个问题的办法很简单，zookeeper 要求为每个节点配置一个不同的的节点编号，记为 nodeid，paxos 过程中以 b=(zxid, nodeid)发起提议，从而当 zxid 相同时会优先选择节点编号较大的节点成为 leader。成为新 leader 的节点首先与 follower 完成数据同步后，再次说明，数据同步过程可能会涉及删除 follower 上的最后一条脏数据，详细分析见 2.4.6.3 。当与至少半数节点完成数据同步后，leader 更新 epoch，在各个 follower 上以(epoch + 1, 0) 为 zxid 写一条没有数据的更新操作。这个更新操作称为 NEW_LEADER 消息，是为了在各个节点上更新 leader 信息，当收到超过半数的 follower 对 NEW_LEADER 的确认后，leader 发起对 NEW_LEADER 的 COMMIT 操作，并进入 active messaging 状态提供服务。

进入 active messaging 状态的 leader 会接收从客户端发来的更新操作，为每个更新操作生成递增的 count，组成递增的 zxid。Leader 将更新操作以 zxid 的顺序发送给各个 follower（包括 leader 本身，一个 leader 同时也是 follower），当收到超过半数的 follower 的确认后，Leader 发送针对该更新操作的 COMMIT 消息给各个 follower。这个更新操作的过程很类似两阶段提交，只是 leader 永远不会对更新操作做 abort 操作。

如果 leader 不能更新超过半数的 follower，也说明 leader 失去了 quorum，此时可以发起新的 leader 选举，最后一条更新操作处于“中间状态”，其是否生效取决于选举出的新 leader 是否有该条更新操作。从另一个角度，当 leader 失去 quorum 的 follower，也说明可能有一个超过半数的节点集合正在

选举新的 leader。

Zookeeper 通过 zxid 将两个场景阶段较好的结合起来，且能保证全局的强一致性。由于同一时刻只有一个 zookeeper 节点能获得超过半数的 follower，所以同一时刻最多只存在唯一的 leader；每个 leader 利用 FIFO 以 zxid 顺序更新各个 follower，只有成功完成前一个更新操作的才会进行下一个更新操作，在同一个 leader 任期内，数据在全局满足 quorum 约束的强一致，即读超过半数的节点一定可以读到最新已提交的数据；每个成功的更新操作都至少被超过半数的节点确认，使得新选举的 leader 一定可以包括最新的已成功提交的数据。

2.8.6.3 Megastore 中的 Paxos

Megastore 中的副本数据更新基于一个改良的 Paxos 协议进行。与 Chubby 和 Zookeeper 仅仅利用 Paxos 选出 primary 不同的是，Megastore 的每次数据更新都是基于一个 Paxos 协议的实例。从而使得 Megastore 具有一个去中心化的副本控制机制。另一方面，为了获得较大的数据更新性能，Megastore 又引入了类似 Primary 的 leader 角色以在绝大部分的正常流程时优化原有 paxos 协议。

基本的 Paxos 协议两个特点使得其性能不会太高：1. 每个 Paxos 运行实例，至少需要经历三轮网络交互：Proposer 发送 prepare 消息、Acceptor 发送 promise 消息、Proposer 再发送 accept 消息；2. 读取 Paxos 上的数据时，需要读取超过半数的 Acceptor 上的结果才能获得数据。对于一个高吞吐、高并发的在线存储系统，上述特性会制约系统的性能。为此，Megastore 使用了一些方式对 Paxos 协议进行了改良。Megastore 中的副本一般都是跨机房、跨地域部署，在通常状态下，某个用户只会访问特点机房中的副本。为此，Megastore 在每个机房为每个副本部署一个特殊的称为协调器（coordinator）的服务。Coordinator 服务相对 Megastore 的底层 Big table 系统而言显得非常简单，其主要功能就是维护副本直接一致性的信息，外部节点（主要是 Megastore 的 client）可以通过访问 Coordinator 获知当前本地副本是否与其他副本一致，即当前副本是否具有最新的已提交的数据。利用 Coordinator，如果判断出当前本地副本已经是最新的数据，则只需读取本地副本，而不需要读取超过半数的节点就可以读取到最新的数据。

下面介绍 Megastore 中的数据读取流程，Megastore 中的数据读取流程除了读取数据外还有两个重要功能：1. 尝试更新本地副本的 coordinator。2. 解决中间态数据问题。这里需要说明的是，Megastore 的更新日志与数据是分离的，每个 Megastore 副本收到更新操作后，都会立刻更新自己的日志，但不会立刻把更新操作应用到对应的 Big Table 中。这是因为，在类似 Paxos 的 Prepare 阶段就发送到各个副本了，此时更新操作会写入日志，但只有收到 Accept 消息后，副本才能确定这是一个已经成功提交的 Paxos 的数据，才可以将更新操作真正写入 Big Table。

流程 2.8.3：Megastore 数据读取流程

1. 查询本地副本对应 Coodinator 以获知本地副本是否已经是最新的已提交的数据。

2. 选择一个要读取的副本，使得该副本肯定包含最新的已提交的更新操作。

   a) 如果从 1 中发现本地副本已经包含最新的已提交的数据，则选择本地副本。

   b) 如果 1 中检查失败，则读取半数以上的副本，从中挑选版本号最大的副本。

3. 追赶数据。对于选择的副本，如果不能确定副本上某次更新操作是已经提交的，则通过查询其他副本确定。如果读取所有副本都无法确定，则以 paxos 协议发起一次空的更新操作。则要么空操作成为本次 paxos 的 value，要么之前不能确定的更新操作成为 paxos 的 value。

4. 修正 Coordinator。如果在 2 中选择了本地副本，且在 1 中 Coordinator 认为本地副本不包含最新已提交的数据，则向 Coordinator 发送一个 Validation 消息，告诉 Coorinator 本地副本以及与其他副本一致。

5. 向 2 中选择的副本查询数据，如果查询失败，重新发起本流程并选择其他副本读数据。

这里解释追赶数据这一过程。假设有 3 个副本，正如在 2.4.4 中详细分析的，如果仅仅读取两个副本，虽然已经满足 Quorum，但在这两个副本中选择版本号最大的一个副本，却不能知道该副本上最后一个版本的数据是不是最新的已提交的数据。例如，如果 3 副本的版本是（3, 2, 3），那么读取前两个副本，3 已经是最新的已提交的版本，但如果 3 副本的版本是（3, 2, 2），那么读取前两个副本，3 不是一个最新的已提交的版本。在 Megastore 中，系统利用 Paxos 协议更新，如果某个副本收到对于某个版本的 Accept 消息，则说明该版本数据已经提交提交，对于没有收到 Accept 消息的数据，副本本身无法判断该数据是否已经提交。为此 Megastore 在读取数据增加了追赶数据的过程，就是为了在各个副本上确定每个 paxos 实例最终产生的 value。

另一方面，如果某次更新对应的 paxos 实例不完整，那么也无法确定该次更新产生的 value。例如，accept 消息只在某一个副本上产生效果并生成对于的更新日志，此时读取所有的副本可以发现该日志并非一个已经成功提交的更新，且对应的那个 paxos 实例也还产生 value，有可能那次更新操作已经失败，也有可能那次更新操作正在进行。为此 Megastore 发起一次 Paxos 空操作，要么空操作成为最后 paxos 的 value，要么正在进行的更新操作成为 paxos 的 value。

从上述流程不难发现，在没有异常，各副本一致的情况下，查询只会发生在本地副本，而无需读取多个副本。

再继续讨论 Megastore 的更新流程。Megastore 每次成功的更新操作都会附带指定下一次更新操作的 Leader 副本。通常，客户端指定本地机房的副本作为 leader 副本。所谓 Leader 副本非常类似 Primary-Secondary 中的 Primary，但 leader 副本不是必须的，只是一种性能优化，利用 leader 副本尝试跳过 Paxos 的准备阶段，简化了 Paxos 流程。但当 leader 副本失败（类似于代码优化中的 fast path 失败），系统退化到普通的 paxos 过程。

流程 2.8.4：Megastore 中的数据更新流程

1. 尝试使用 Leader 直接提交数据。访问 Leader 节点，请求 Leader 节点以 paxos 编号 0 直接向各副本发送 Accept 消息。如果成功，转 3.
2. 准备阶段：通过更新操作在日志中的位置，获得当前 paxos 实例的编号。在本次 paxos 实例中，选择一个最大的轮次号 b 发起正常的 Paxos 准备流程，如果收到超过半数的 promise 消息，则转 3.
3. 批准阶段：向所有的副本发送 Accept 消息，如果失败，转 2.
4. 修正 Coordinator。如果没有收到某个副本的 Accepted 消息，向该副本对应的 Coordinator 发送一个 Invalid 消息，告知该 Coordinator 对应副本已经不与其他副本同步。
5. 各个节点根据本次操作日志更新对应的 Big Table 中的数据。

上述流程中，步骤 1 尝试使用 Leader 副本进行快速更新。如果该 leader 副本收到的是本次 paxos 实例（对应于全局更新操作的次序）第一个更新请求，则该流程可以生效。当 leader 节点失效，或者有并发的多个更新请求时，该优化失败，转为正常的 Paxos 过程。

流程中的步骤 4 是不能失败的，如果某个副本处于不一致的状态，而又不能通知对应的 Coordinator，则用户就有可能在读取流程中读到该副本上的数据，从而打破系统的强一致性。Megastore 对此的办法是：1. 相比于底层的 Big table 系统，Coordinator 是一个非常简单的无状态的轻量级服务，其稳定性本身较高。2. 每个 Coordinator 的状态都会计入 Chubby，一旦 Coordinator 失去 Chubby 中锁，即失去 Chubby lease，Coordinator 会将对应副本的状态标记为不一致。其他节点可以通过监控 Coordinator 再 Chubby 中对应的锁而获知 Coordinator 的状态。从而，一旦一个 Coordinator 异常失效，更新流程可能会阻塞在第 4 步直到这个 Coordinator 失去在 Chubby 中的锁。在极端网络分化等异常下，可能有这样的情况：更新流程的执行节点无法给 Coordinator 发送 Invalid 消息，而 Coordinator 却能始终占有 Chubby 中对应的锁。Megastore 将这种极端情况通过 OP 手动杀 Coordinator 解决。

Megastore 通过改良的 Paxos 协议给出了一种跨机房、跨地域实现高可用系统的方案。与 PNUTS 的跨机房方案相比，Megastore 的方案具有强一致性，且可以随时在多个副本上读取最新的已提交的数据。而 PNUTS 的虽然也具有读最新已提交的数据的功能，但由于副本之间采用异步同步的方式，通常只能在 Primary 副本上才能读到最新的已提交的数据。

2.9 CAP 理论

CAP 理论是由 Eric Brewer 提出的分布式系统中最为重要的理论之一[8]。本文将 CAP 理论安排在原理部分的最后介绍是为了利用前面已经介绍过的几种分布式协议来帮助理解 CAP 理论。

2.9.1 定义

CAP 理论的定义很简单，CAP 三个字母分别代表了分布式系统中三个相互矛盾的属性：

Consistency (一致性)：CAP 理论中的副本一致性特指强一致性（1.3.4 ）；

Availiablity(可用性)：指系统在出现异常时已经可以提供服务；

Tolerance to the partition of network (分区容忍)：指系统可以对网络分区（1.1.4.2 ）这种异常情况进行容错处理；

CAP 理论指出：无法设计一种分布式协议，使得同时完全具备 CAP 三个属性，即 1)该种协议下的副本始终是强一致性，2)服务始终是可用的，3)协议可以容忍任何网络分区异常；分布式系统协议只能在 CAP 这三者间所有折中。

CAP 理论的详细证明可以参考相关论文。这里可以简单用一个反例证明不存在 CAP 兼具的系统。假设系统只有两个副本 A 和 B，Client 更新这两个副本，假设在网络分化时，Client 与副本 A 可以正常通信，但副本 B 与 Client、副本 B 与副本 A 无法通信，此时，Client 对副本 A 更新的信息永远无法同步到副本 B 上。如果希望系统依旧具有强一致的属性，则此时需要停止更新服务，即不再修改数据，从而让副本 A 与副本 B 保持一致；如果希望系统依旧可以提供更新服务，则只能更新副本 A 而无法更新副本 B，此时无法保证副本 A 与副本 B 一致。

2.9.2 CAP 理论的意义

热力学第二定律说明了永动机是不可能存在的，不要去妄图设计永动机。与之类似，CAP 理论的意义就在于明确提出了不要去妄图设计一种对 CAP 三大属性都完全拥有的完美系统，因为这种系统在理论上就已经被证明不存在。

2.9.3 协议分析

本节分析在第二章中介绍的几种分布式协议是如何在 CAP 三大属性中做折中与取舍的。在三章介绍典型分布式系统时，也会用 CAP 理论对这些系统的分布式协议进行分析。

2.9.3.1 Lease 机制

Lease 机制牺牲了部分异常情况下的 A，从而获得了完全的 C 与很好的 P。

上面这句话有点抽象，下面一一解释。首先，Lease 机制不是在任何情况下都具有可用性的，使用 Lease 机制的协议，在发生异常时，需要等待 Lease 超时才能收回 Lease 权限。然而，Lease 的持有者可能在 Lease 超时前就已经出现异常而不能提供服务了，直到 Lease 超时这段时间内，系统服务的可用性都有问题。例如，如果用 lease 决定 Primary 副本的，Primary 副本节点宕机后，只有待 Lease 超时才能选出新的 primary 副本，这段时间由于缺乏 primary 副本是没有更新服务的。再者，Lease 协议本身保证了对于 Lease 约定的承诺在 Lease 颁发者和持有者之间是始终一致的。即使 Lease 持有者由于网络分化没有真正收到 Lease，Lease 颁发者也会在 Lease 时间内执行自己的承诺；而一旦 Lease 持有者收到 Lease，则即使再出现网络分化，也无法影响双方对 Lease 承诺理解的一致性。最后，Lease 协议引入了“时间”这一概念，使得在对抗网络分化上有其特别的优势，另外，Lease 只需由颁发者向持有者通信，即使网络是单向的也不影响 Lease 协议的正常工作。

2.9.3.2 Quorum 机制

这里仅讨论一般的 Quorum 机制，即总共有 N 个副本，成功更新 W 个副本则算成功提交，读取时读 R 个副本。这种一般的 Quorum 机制，在 CAP 三大因素中都各做了折中，有一定的 C，有较好的 A，也有较好的 P，是一种较为平衡的分布式协议。

首先，读取 R 个副本时，可以保证读取到成功提交的版本，但无法保证读取到最新的成功提交的版本（2.4.4 ）。也就是说，系统具有一定的一致性，却无法真正做到强一致性。再者，无论是更新 W 个副本，还是读取 R 个副本，协议可以允许部分副本异常而不影响更新或者读取服务。最后，只要能与 W 个副本通信就可以提供更新服务，能与 R 个副本通信就可以提供读服务，协议具有一定的容忍网络分化的能力。工程中，当使用 3 个副本时，可以讲三个副本部署在三个不同的机房，只有同时出现两个机房的网络都异常时才会影响服务，这种情况的概率本身已经非常低了。

2.9.3.3 两阶段提交协议

两阶段提交系统具有完全的 C，很糟糕的 A，很糟糕的 P。

首先，两阶段提交协议保证了副本间是完全一致的，这也是协议的设计目的。再者，协议在一个节点出现异常时，就无法更新数据，其服务可用性较低。最后，一旦协调者与参与者之间网络分化，无法提供服务。

2.9.3.4 Paxos 协议

同样是强一致性协议，Paxos 在 CAP 三方面较之两阶段提交协议要优秀得多。Paxos 协议具有完全的 C，较好的 A，较好的 P。Paxos 的 A 与 P 的属性与 Quorum 机制类似，因为 Paxos 的协议本身就具有 Quorum 机制的因素。

首先，无需赘述，Paxos 协议是一种强一致性协议。再者，Paxos 协议只有两种情况下服务不可

用：一是超过半数的 Proposer 异常，二是出现活锁（2.8.4 ）。前者可以通过增加 Proposer 的个数来降低由于 Proposer 异常影响服务的概率，后者本身发生的概率就极低。最后，只要能与超过半数的 Proposer 通信就可以完成协议流程，协议本身具有较好的容忍网络分区的能力。

3 参考资料

[1] Andrew S. Distributed Sytems: Principles and Paradigms.

[2] M.Tamer, Patrick. Principles of Distributed Database.

[3] Philip A. Bernstein, Vassos Hadzilacos, Nathan Goodman. Concurrency Control and Recovery in Database Systems.

[4] Cary G. Gray, David R. Cheritioin. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency.

[5] Philip A. Bernstein, Nathan Goodman. Multiversion Concurrency Control – Theory and Alorithms.

[6] Lamport, Leslie. The Part-Time Parliament.

[7] Leslie Lamport. Paxos Made Simple.

[8] Eric Brewer. Towards Robust Distributed System.

[9] Sanjay Ghemawat, Howard Gobiooff, Shun-Tak Leung. The Google File System.

[10] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.

[11] Fay Chang, Jeffrey Dean. Bigtable: A Distrubted Storage System for Structured Data.

[12] Jason Baker, Chris Bond. Megastore: Providing Scalable, Highly Available Storage for Interactive Services.

[13] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems.

[14] Brian F. Cooper, Raghu Ramakrishnan. PNUTS: Yahoo!’s Hosted Data Serving Platform.

[15] John Maccormick, Chandu Thekkath. Niobe: A Practical Replication Protocol.

[16] Giuseppe DeCandia, Deniz Hastorun. Dynamo: Amazon’s Highly Available Key-value Store.

[17] Avinash lakshman, Prashant Malk. Cassandra – A Decentralized Structed Storage System.

[18] Mola*, Armor*, Big Pipe*, Doris*.