# 概念

## 异常

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。

2、网络异常：消息丢失，两片节点之间彼此完全无法通信，即出现了“网络分化”；消息乱序，有一定的概率不是按照发送时的顺序依次到达目的节点，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；数据错误；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，但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。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。

3、分布式三态：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。

4、存储数据丢失：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。

5、异常处理原则：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。

## 副本

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。

副本协议是贯穿整个分布式系统的理论核心。

**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。

1、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。

2、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。

3、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。

4、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。

5、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。

## 衡量指标

1、性能：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。

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。

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。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。

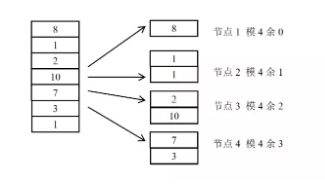
4、一致性：分布式系统为了提高可用性，总是不可避免的使用副本的机制，从而引发副本一致性的问题。越是强的一致的性模型，对于用户使用来说使用起来越简单。

# 分布式系统

## 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。

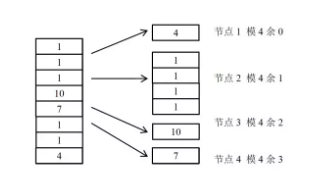
### 哈希方式



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。

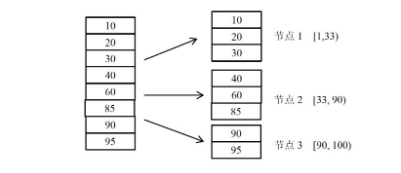
针对哈希方式扩展性差的问题，一种思路是不再简单的将哈希值与机器做除法取模映射，而是将对应关系作为元数据由专门的元数据服务器管理.同时，哈希值取模个数往往大于机器个数，这样同一台机器上需要负责多个哈希取模的余数。但需要以较复杂的机制维护大量的元数据。

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。



### 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。



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。

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。

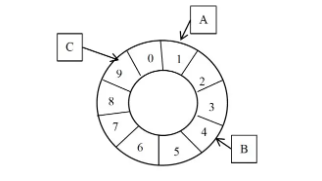
### 按数据量分布

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。

### 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。



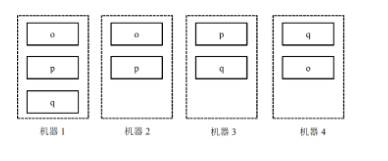
使用一致性哈希的方式需要将节点在一致性哈希环上的位置作为元信息加以管理，这点比直接使用哈希分布数据的方式要复杂。然而，节点的位置信息只于集群中的机器规模相关，其元信息的量通常比按数据范围分布数据和按数据量分布数据的元信息量要小很多。

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。

### 副本与数据分布

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。

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。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。



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。

### 本地化计算

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”。

### 数据分布方式的选择

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，数据分布方式是可以灵活组合使用的，往往可以兼备各种方式的优点，收到较好的综合效果。

例：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。

## 基本副本协议

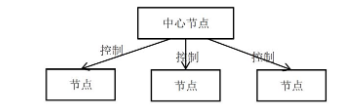
副本控制协议指按特定的协议流程控制副本数据的读写行为，使得副本满足一定的可用性和一致性要求的分布式协议。副本控制协议要具有一定的对抗异常状态的容错能力，从而使得系统具有一定的可用性，同时副本控制协议要能提供一定一致性级别。由CAP 原理（在2.9 节详细分析）可知，要设计一种满足强一致性，且在出现任何网络异常时都可用的副本协议是不可能的。为此，实际中的副本控制协议总是在可用性、一致性与性能等各要素之间按照具体需求折中。

副本控制协议可以分为两大类：“中心化(centralized)副本控制协议”和“去中心化(decentralized)副本控制协议”。

### 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。



### primary-secondary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）**。

#### 数据更新

**数据更新基本流程**

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

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。

#### 数据读取方式

数据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。

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。

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。

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。

#### 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。

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，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。

#### 数据同步

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：

1. 由于网络分化等异常，secondary上的数据落后于primary上的数据。
2. 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。
3. 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。

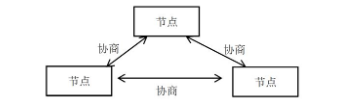
对于第一种secondary数据落后的情况，常见的同步方式是回放primary上的操作日志（通常是redo日志），从而追上primary的更新进度。对于脏数据的情况，较好的做法是设计的分布式协议不产生脏数据。如果协议一定有产生脏数据的可能，则也应该使得产生脏数据的概率降到非常低得情况，从而一旦发生脏数据的情况可以简单的直接丢弃有脏数据的副本，这样相当于副本没有数据。另外，也可以设计一些基于undo日志的方式从而可以删除脏数据。如果secondary副本完全没有数据，则常见的做法是直接拷贝primary副本的数据，这种方法往往比回放日志追更新进度的方法快很多。但拷贝数据时primary副本需要能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。

注：MySQL的主从复制是典型的primary-secondary协议的体现。

### 去中心化副本控制协议

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。

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**。



## Lease机制

Lease（租约）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。

### 基于lease的分布式cache系统

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，在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。

**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

1. 基于lease的cache，客户端节点修改元数据流程

2.1节点向服务器发起修改元数据请求。

2.2 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。 2.3 服务器等待所有与该元数据相关的lease超时。

2.4 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。

最后，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。

### lease机制的分析

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。

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。

### 基于lease机制确定节点状态

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。

### lease的有效期时间选择

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。

## Quorum机制

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wi，i为更新操作单调递增的序号，每个wi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vi。假设每个副本都保存了历史上所有版本的数据。

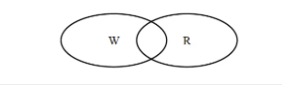
### write-all-read-one

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。

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。

### Quorum定义

在Quorum（法定人数）机制下，当某次更新操作wi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wi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wi更新后 的数据vi。如果某次更新wi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wi更新后的数据vi。如图，Quorum机制的原理可以文森图表示。



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v1，某次更新操作w2在前3副本上成功，副本情况变成（v2 v2 v2 v1 v1）。此时，任意3个副本组成的集合中一定包括 v2。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。

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。

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。

### 读取最新成功提交的数据

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，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：在读取到（v2 v1 v1）时，继续读取剩余的副本，若读到剩余两个副本为（v2 v2）则v2 是最新的已提交的副本；若读到剩余的两个副本为（v2 v1）或（v1 v1）则v1 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。

### 基于Quorum机制选择primary副本

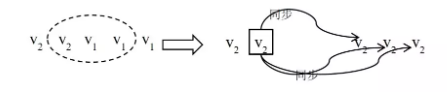
读取数据时依照一致性要求的不同可以有不同的做法：如果需要强一致性的立刻读取到最新的成功提交的数据，则可以简单的只读取primary副本上的数据即可，也可以通过上节的方式读取；如果需要会话一致性，则可以根据之前已经读到的数据版本号在各个副本上进行选择性读取；如果只需要弱一致性，则可以选择任意副本读取。

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：在N=5，W=3，R=3 的系统中，某时刻副本最大版本号为（v2 v2 v1 v1 v1），此时v1 是系统的最新的成功提交的数据，v2 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。



1. 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（v1 v1 v1），则任 选一个副本作为primary，新primary以v1 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v2，如果系统不能保证之后的v2 与之前的v2 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（v2 v1 v1），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v2 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。

## 日志技术

# 一致性

如果深究一致性的语义还是略有差别，这里简单归为3类介绍其区别：

Coherence更关注多核共享存储架构的cache数据一致性；

Consistence更关注分布式系统的一致性，该一致性往往以客户端为视角，比如数据库的事务一致性（ACID），分布式系统副本数据同步的CAP理论中的C都指的是这个一致性。尽管两者还有一定差异：ACID关注的是系统内部的数据一致性，CAP关注的是多副本数据对外界的一致性；

Consensus关注的是达成一致性的手段或者协议，是一种过程，比如通过Paxos达成共识，一致性是共识的目的。

几个C虽有差别，但也不是毫无关系，只是看待问题的角度和层次的差别，理论很多是相通的。

在分布式系统领域一直有两个重要的理论用于指导实践，分别是CAP和FLP。

## CAP

对于一个分布式系统，不能同时满足以下三点：

一致性（Consistency）

可用性（Availability）

分区容错性（Partition Tolerance）

CAP理论由加州大学伯克利分校的计算机教授Eric Brewer在2000年提出，其**核心思想是任何基于网络的数据共享系统最多只能满足数据一致性(Consistency)、可用性(Availability)和网络分区容忍(Partition Tolerance)三个特性中的两个**，三个特性的定义如下：

a. 一致性 (Consistency)：也就是线性一致性，一个写操作返回成功，那么之后的读请求都必须读到这个新数据；如果返回失败，那么所有读操作都不能读到这个数据。所有节点访问同一份最新的数据。

b. 可用性 (Availability)：对数据更新具备高可用性，请求能够及时处理，不会一直等待，即使出现节点失效。

c. 分区容错性 (Partition tolerance)：能容忍网络分区，在网络断开的情况下，被分隔的节点仍能正常对外提供服务。

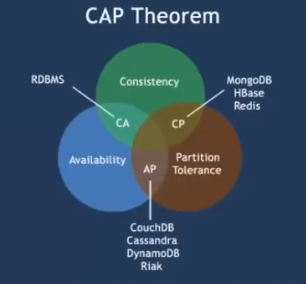
CAP理论的表述很好地服务了它的目的，开阔了分布式系统设计者的思路，在多样化的取舍方案下设计出多样化的系统。在过去的十几年里确实涌现了不计其数的新系统，也随之在一致性和可用性的相对关系上产生了相当多的争论。**一般来说使用网络通信的分布式系统，无法舍弃P性质，那么就只能在一致性和可用性上做选择。既然在分布式系统中一致性和可用性只能选一个**。

那Paxos、Raft等分布式一致性算法是如何做到在保证一定的可用性的同时，对外提供强一致性呢？在CAP理论提出十二年之后，其作者又出来辟谣。“三选二”的公式一直存在着误导性，它会过分简单化各性质之间的相互关系：

首先，由于分区很少发生，那么在系统不存在分区的情况下没什么理由牺牲C或A。

其次，C与A之间的取舍可以在同一系统内以非常细小的粒度反复发生，而每一次的决策可能因为具体的操作，乃至因为牵涉到特定的数据或用户而有所不同。

最后，这三种性质都可以在程度上衡量，并不是非黑即白的有或无。可用性显然是在0%到100%之间连续变化的，一致性分很多级别，连分区也可以细分为不同含义，如系统内的不同部分对于是否存在分区可以有不一样的认知。所以一致性和可用性并不是水火不容，非此即彼的。Paxos、Raft等分布式一致性算法就是在一致性和可用性之间做到了很好的平衡的见证。



满足AP：CouchDB、Cassandra、DynanoDB

满足CP：MongoDB、HBase、Redis

满足CA：RDBMS

## FLP

FLP定理是由Fischer，Lynch和Patterson三位科学家于1985年发表，是分布式理论中最重要的理论之一，它指出在最小化异步网络通信场景下，即使只有一个节点失败，也没有一种确定性的共识算法能够使得在其他节点之间保持数据一致。

a. 统模型假设

异步通信与同步通信的最大区别是没有时钟、不能时间同步、不能使用超时、不能探测失败、消息可任意延迟、消息可乱序

通信健壮只要进程非失败，消息虽会被无限延迟，但最终会被送达；并且消息仅会被送达一次（无重复）

fail-stop模型进程失败如同宕机，不再处理任何消息。相对Byzantine模型，不会产生错误消息

失败进程数量最多一个进程失败

b. 衡量标准

衡量一个分布式算法是否正确有三个标准：

Termination（终止性）非失败进程最终可以做出选择

Agreement（一致性）所有的进程必须做出相同的决议

Validity（合法性）进程的决议值，必须是其他进程提交的请求值 终止性，描述了算法必须在有限时间内结束，不能无限循环下去；一致性描述了我们期望的相同决议；合法性是为了排除进程初始值对自身的干扰。

**CAP与FLP关系**

FLP讨论的分布式共识（distributed consensus）的问题。分布式共识可实现的功能包括：

leader election

replicated state machine

distributed commit

而CAP关注的是复制存储（replicated storage）的问题，replicated storage可以看作是replicated state machine的一个特例。可以看出，复制存储是分布式共识的子问题。也即，FLP关注的问题更加通用，CAP问题是FLP问题的子集。

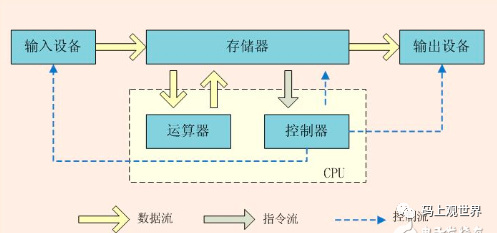
此外，CAP中的复制存储问题只讨论了这样一类问题：同一份数据在不同节点上进行存储（主从复制即是这样一类问题）；而FLP中的分布式共识问题除了CAP中的问题外，还讨论了这样一类问题：多个任务（数据）被调度到不同节点上并行执行（存储），不同节点上的任务和状态可能是不同的（2PC协议即包含了这样一类问题）。由此也可见，FLP中讨论的问题更加复杂。一些方案可能无法解决FLP中的问题，但可能能够解决CAP中的问题。

## 一致性模型

在计算机的物理世界中，每个系统的进程都是有距离的，比如一个没有在CPU本地缓存的值距离内存条的距离通常为30cm，不考虑CPU装载值等操作开销，仅仅按照信号传输来算，也需要1纳秒（10的负9次方秒）的时间（光信号速度为30万公里每秒）。

如果把本地取值扩展到跨数千公里范围外的计算机，这个时间达到数百毫秒。这就是说计算机物理世界的操作都是耗时的或者说操作是有延时的，因此可以说，客观世界的约束性才是导致上层应用复杂性的根源。而产生这种约束性的直接原因在于1945年美籍匈牙利人冯-诺依曼提出的经典冯-诺依曼计算机存储结构：该结构将指令数据和存储数据公用一个存储器，在控制器的作用下，运算器从存储器中加载指令和数据完成计算之后数据写回存储器，运算器和控制器组成中央处理器（CPU）。

如果输入和输出数据不在存储器中，还需要从外部输入设备读取数据和写出到外部输出设备，输入设备和输出设备组成I/O设备，这样一个微型的计算机结构就成型了，如下图所示：



然而这种结构有个很大的弊端：CPU的处理频率远高于存储器，另外运算器的容量也远小于主存储器，因此大量的数据存取和运算就导致了系统瓶颈。为了解决存储器和CPU运算速度不匹配的问题，其中一种解决办法是利用数据的局部性原理，引入CPU高速缓存（L1 cache），在此基础上，还有二级甚至三级缓存。于是导致了新的问题：当存储器中的数据更新，如果无法及时通知缓存失效，就会导致存储器的数据跟缓存数据不一致。

可以说正是因为这种存储和计算分离的结构问题导致了数据不一致问题。这个是一致性问题产生的微观原因，如果将这种结构逐步向上层推进，会发现计算机领域的很多系统设计也都存在这种问题，比如线程中的局部存储和主存中的数据，数据库或者其他外存设备的数据与内存中缓存的数据，分布式系统中的计算与存储分离架构如AWS的计算集群和外部存储系统S3都会存在数据不一致问题，而且随着层级的上升，不一致问题越来越明显，为解决一致性问题引入的方案越来越复杂。

比如在多线程并发写入和读取一个未加同步限制的共享变量，由于每个线程调度顺序的不确定性，导致对公共变量的读取并不能像在单线程中那么“确定“，对于调用方来说，这个读取是不符合预期，也可以说是不准确的。那如何让系统给出符合预期的准确结果呢？在介绍如何解决一致性问题之前，有必须先介绍下一致性模型：给定一些涉及操作与状态的规则，随着操作的演进，系统将一直遵循这些规则。

一致性模型是所有被允许的操作记录的集合。当我们运行一个程序，经过一系列集合中允许的操作，特定的执行结果总是一致的。如果程序意外地执行了非集合中的操作，我们就称执行记录是非一致的。如果任意可能的执行操作都在这个被允许的操作集合内，那么系统就满足一致性模型。因为对一致性要求的程度不同，实现一致性模型的复杂度和代价也不同。

按照一致性要求的严格程度不同，有以下几种类型：

顺序一致性（Sequential consistency）

线性一致性（Linearizability）

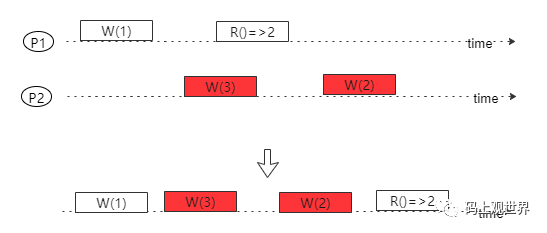
因果一致性（Casual consistency）

### 顺序一致性

顺序一致性概念是1979年Lamport 在论文《How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs 》中提出，该概念基于在多处理器环境下如何就存储器和处理器数据达成顺序一致：假设执行结果与这些处理器以某一串行顺序执行的结果相同，同时每个处理器内部操作的执行看起来又与程序描述的顺序一致。满足该条件的多处理器系统我们就认为是顺序一致的。实际上，处理器可以看做一个进程或者一个线程，甚至是一个分布式系统。顺序一致性概念里面有两个约束：

1、执行结果与这些处理器以某一串行顺序执行的结果相同多处理器在并发执行下会产生多种运行顺序的组合，如果能够找到一种合法的执行顺序，其结果跟把多处理器串行执行的结果一样，就可以认为符合顺序一致性模型，也就是可以做到顺序一致性。当然在设计这样的系统还需要其他的保证，比如事件通知等。这里的合法就是第二个约束的内容。

2、每个处理器内部操作的执行看起来又与程序描述的顺序一致 一个处理器内部的操作顺序按照程序定义的顺序执行。比如在一个线程内部，其操作都是串行的，该约束很容易保证。举一个顺序一致性模型的例子：两个线程P1和P2，两种类型的操作：W(x)对共享变量写操作，R()=>x对共享变量的读操作。因为操作是有时间延迟，所以用一个矩形表示：左边沿表示操作开始，右边沿表示操作结束，沿着时间轴方向排列，但P1的R操作和P2的W操作有时间上的重叠。



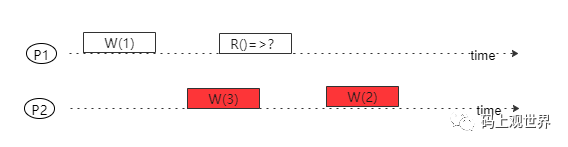
从图中可以看到两个线程如果按照W(1)=>W(3)=>W(2)=>R()，就可以保证顺序一致性，因为线程内部的两个操作都跟原来顺序保持相对一致，但是R操作时间上在W(2)后面执行。

### 线性一致性/原子一致性/严格一致性

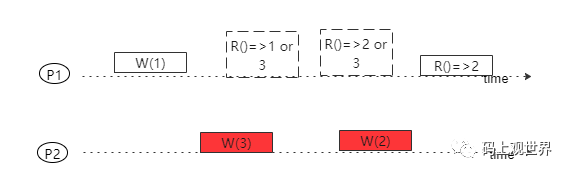
线性一致性(Linearizability)，或称原子一致性或严格一致性，指的是程序在执行顺序组合中存在可线性化点P的执行模型，这意味着一个操作将在程序的调用和返回之间的某个点P生效，之后被系统中并发运行的所有其他线程所感知。

线性一致性概念是1990年Maurice Herlihy · Jeannette M Wing在论文《Linearizability: A Correctness Condition for Concurrent Objects》中提出。

为理解该概念，先介绍一致性模型中同样存在的happen-before原则：任意的读写操作事件都分为调用发起和请求响应，比如读操作发起和读操作响应，如果某事件A的发起时刻在某事件B的响应时刻之后，那么就说事件B happen-before 事件A。在线程内部，所有事件都满足happen-before原则，满足此原则的操作，后续的读操作一定能够感知（获取）到写操作的结果。但是在多线程环境下，就不一定满足了，同样那上图来说明：



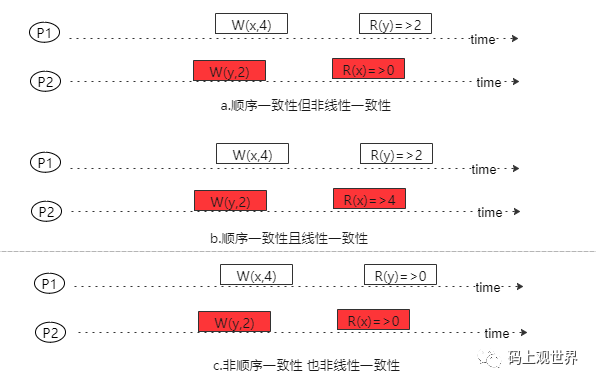
因为在进程P1内部R和W满足happen-before原则，因此W的写能够被R读取到，但是P1的R和P2的W【W(3)】不一定。我们主要关注进程P1和P2在时间上有重叠的读和写操作：在P1读取过程中，W写的响应还没返回，在写开始和返回的区间内，W写可能已经完成，也可能还在进行中，因此R读取的结果可能是1也可能是3，如果考虑到写操作的可中断性，这个值还可能是其他值，只有当P1的读和W满足上述原则，读取的值才是确定的，如下图：



但线性一致性要求如果在这个区间某个点，写操作一旦生效，那么后续的任何读一定能够获取到最新值，这意味着该写操作要么成功要么失败（所有后续读操作仍然读取到原来的值），也就是满足原子性，那么也就符合线性一致性。用直白话说就是，在并发场景下，一个线程对共享变量的操作能立即被其他线程感知到。

由于操作的延迟性，操作在发起和获得响应之间已经发生，因此线性一致性允许在操作被响应前是可以被其他线程访问到，而实现这种一致性要求操作必须是原子的：如果一个操作对于系统的其他部分是不可中断的，那这个操作就可以称之为原子的，线性的，不可分割的，那这个操作就可以称之为具有线性一致性的特性。

比较上述两种一致性模型可知：线性一致性是对顺序一致性的加强，两者都要保证在线程内部操作的相对顺序，但线性一致性暗含着一个全局时钟，所有线程按实际发生的时间顺序执行，而顺序一致性只需要保证相对顺序即可。满足线性一致性一定满足顺序一致性，反正不成立。为加深对两者模型的理解，再看下面的示例：



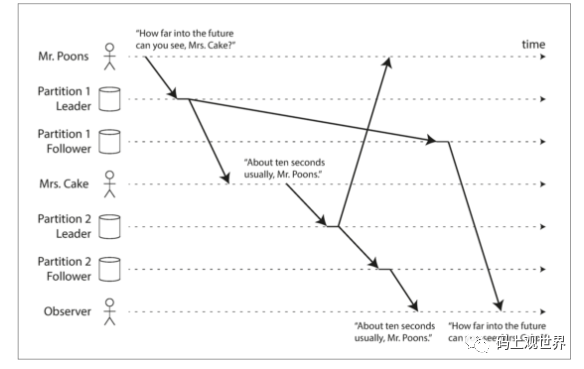
简单解释下：a中只要保证单进程的内部执行顺序基础上，R(x)和R(y)分别在W(x,4)和W(y,2)之后就得到期望结果（不论结果正确与否），符合顺序一致性，但是因为从全局时钟来看，R(x)在W(x)之后但是不能读取最新值，所以不符合线性一致性。在b中能读取到最新值，所以符合线性一致性。c因为x和y都不能读取最新值，所以不满足顺序一致性更不谈线性一致性。

### 因果一致性

线性一致性要求所有线程的操作按照一个绝对的时钟顺序执行，这意味着线性一致性是限制并发的，否则这种顺序性就无法保证。

由于在真实环境中很难保证绝对时钟同步，因此线性一致性是一种理论，实现线性一致性的代价也最高，但是实战中可以弱化部分线性一致性：只保证有因果关系的事件的顺序，没有因果关系的事件可以并发执行。

所谓因果也可以用前文中的happen-before原则来阐述：如果A事件 happen-before B事件，那么事件A，B之间存在因果关系。在分布式系统设计中，经常会因为副本数据同步的延迟导致因果关系颠倒的现象，以下引用一个问答系统中的案例：



  图中先有问然后才有答，但是因为在副本数据同步的时候，问的数据同步落后于答的数据同步，从而导致在观察者看来，先有答然后才有问。

## 分类

### 弱一致性

#### 最终一致性

最终一致性：往分布式数据库中写入数据，此时另一个节点读数据，系统是**无法保证能够读到最新的数据的**，但是可以保证将来某一时刻可以读到写入的数据。

注：中兴GoldenDB基于最终一致性，阿里OceanBase基于强一致性。

##### DNS

DNS（Domain Name System）

##### Gossip

Gossip（Cassandra的通信协议）

###### 背景

**可扩展性（Scalable）**

gossip协议是可扩展的，一般需要O(logN) 轮就可以将信息传播到所有的节点，其中N代表节点的个数。每个节点仅发送固定数量的消息。在数据传送的时候，节点并不会等待消息的ack，所以消息传送失败也没有关系，因为可以通过其他节点将消息传递给之前传送失败的节点。

**容错（Fault-tolerance）**

网络中任何节点的重启或者宕机都不会影响 gossip协议的运行。能够容忍较高的丢包率。

**健壮性（Robust）**

gossip协议是去中心化的协议，所以集群中的所有节点都是对等的，没有特殊的节点，所以任何节点出现问题都不会阻止其他节点继续发送消息。任何节点都可以随时加入或离开，而不会影响系统的整体服务质量（QOS）。

**最终一致性（Convergent consistency）**

gossip协议实现信息指数级的快速传播，因此在有新信息需要传播时，消息可以快速地发送到全局节点，在有限的时间内能够做到所有节点都拥有最新的数据。

###### 概述

从gossip单词就可以看到，其中文意思是八卦、流言等意思，我们可以想象下绯闻的传播（或者流行病的传播），gossip协议的工作原理就类似于这个。gossip协议通过一传十，十传百的方式将信息传播到整个网络中，并在一定时间内使得系统内的所有节点数据一致。大名鼎鼎的Bitcoin使用了Gossip协议来传播交易和区块信息。

###### 原理

###### 应用

 浪潮数据库ZNBase作为新一代分布式数据库，采用无中心对等网络架构。因此使用对等网络协议gossip作为处理集群各个节点间元数据同步的方案。

**什么是元数据**

元数据是指关于数据的组织及其关系的信息，简言之，元数据就是关于数据的数据。各个节点将自己的元数据传播到其他节点，并接收其他节点的元数据。在得到完整的集群元数据后可以对集群进行管理、调度、监控，以及制定sql分布式计划等。元数据主要包括以下几类：

数据地址：用于分布层寻找所需数据的位置，执行分布式计划；

节点心跳：节点的心跳信息，用于集群了解所有节点的状态；

集群配置：各节点共享集群配置信息；

节点信息：节点的详细信息，包括节点硬件信息、任务等数据；

存储信息：Store的详细信息，包括Store中的range信息，已使用，未使用的磁盘容量等；

共享信息：业务中需要共享的数据，例如表的一些配置，缓存刷新标记等；

**ZNBase中gossip是如何实现的**

**连接数**

每个节点发起的最大连接数由节点数决定，连接数b的3次方等于节点数n，并且最大允许b个节点连接自己。因此每个节点和2b个节点相连接。例如125个节点的集群，每个节点主动连接5个节点，并接受另外5个节点的连接。

**节点选择**

节点通常选择离自己最远的节点发起连接，并且定期结束离自己最近的节点，重新选择最远的节点连接。

**发送方式**

采用增量发送，通过version控制，每次只发送最新的数据。

**发送步骤**

1、节点A连接节点B后，向其发送自己的新信息（key,value,version），节点 B 在收到信息后对比自己的version，更新比自己新的数据。

2、节点B有新信息后，对比节点A的version，将自己的比节点A新的信息(key,value,version) 发送给A，A更新本地数据。

如果把两个节点数据同步一次定义为一个周期，理论上一个周期内可以使两个节点完全一致。

**如何使用元数据**

通过上述内容，已经可以了解ZNbase由于全对等网络架构，每个节点都通过gossip模块获取到完整的集群元数据，其他模块可以通过gossip模块直接获取元数据，也可以监控数据变化，在数据变化时及时获取到最新内容的推送。例如：

1. 集群初始化后，初始节点将集群信息和配置写入gossip，要加入集群的节点首先连接gossip获取集群信息和配置后加入集群，并写入缓存，加入集群后再启动数据库服务。集群信息和配置发送变化后，会推送给所有节点，刷新缓存。
2. 每个节点启动后也将自身的节点信息，存储信息，心跳信息按照不同的的频率周期性写入gossip，并同步到所有节点。同时，各个模块根据自身需要获取元数据。
3. 分发层需要获取数据地址并缓存，确定所需数据所在的节点后，才能执行计划。
4. 副本层在做range的分裂，合并，平衡等操作时需要存储信息和节点心跳，操作完成后会写入新的数据地址信息，并通知分发层刷新缓存。
5. raft模块需要监控心跳信息，判断其raft group是否正常，在心跳发送变化时及时作出处理。
6. sql层中，会使用大量缓存，如果数据发送变化需要刷新缓存时，会通过gossip通知其他节点同步刷新缓存。
7. 制定分布式计划需要确定节点心跳正常，节点停止命令通过设置节点状态为停止中，通过gossip通知整个集群，其他节点得知后停止向这个节点发送新的命令。
8. 集群管理功能通过元数据向客户展示整个集群的运行状态。

### 强一致性

强一致性主要是解决这一问题：数据不能存在单点上。

分布式系统对于FT（fault tolerance）的一般解决方案是state machine replication（可以理解为一个函数，具有一个初始状态和最终状态，对应的状态就是日志log），严格地讲是state machine replication的共识（consensus）算法。Paxos其实是一个共识算法，系统的最终一致性不仅仅需要达成共识，还会取决于client的行为。

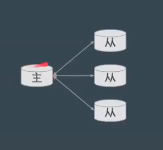
#### 主从同步

主从同步复制：

1、Master接受写请求；

2、Master复制日志到slave

3、Master等待，直到**所有**从库返回



**存在问题：**

一个节点失败，Master阻塞，导致整个集群不可用，保证了一致性，可用性却大大降低。

**基本想法：**多数派

每次写都保证写入大于N/2个节点，每次读保证从大于N/2个节点中读

但是，在并发环境下，多数派无法保证系统正确性，顺序非常重要！需要引入共识算法。



#### 多数派

#### 算法

##### Paxos

##### Chubby

##### Raft

##### ZAB

##### 对比分析

**paxos和raft**

Raft协议比paxos的优点是容易理解，容易实现。它强化了leader的地位，把整个协议可以清楚的分割成两个部分，并利用日志的连续性做了一些简化：

1. Leader在时。由Leader向Follower同步日志。
2. Leader挂掉了，选一个新Leader，Leader选举算法。

**zab和raft**

开始启动的情况下就必要选主，然后再提供正常服务都有一个异常场景的恢复过程

**共同点：都使用timeout来重新选择leader**

采用quorum来确定整个系统的一致性(也就是对某一个值的认可),这个quorum一般实现是集群中半数以上的服务器,zookeeper里还提供了带权重的quorum实现.

都由leader来发起写操作.都采用心跳检测存活性..

leader election都采用先到先得的投票方式

**区别：在于选主的方式**

zab是广播式互相计数方式，发现别人比自己牛逼的时候要帮助别人扩散消息，根据本机计数决定谁是主raft是个节点发起投票，大家根据规则选择投于不投，然后各自收到别人对自己的投票超过半数时宣布自己成为主

ZAB协议，只有当过半节点提交了事务，才会给客户端事务提交的回应，是一个类似二阶段提交的方式，重新选主后，特别有一个同步日志的阶段；而Raft协议，leader提交了事务，并且收到过半follower对准备完成事务的ack后，自身节点提交事务，至于过半数节点提交事务这件事，是在之后通信过程中渐渐完成的，重新选主后，没有单独日志同步的阶段。 这导致了一个问题，Raft中如果给客户端回应完，leader挂掉了，如何保证一致性。保证在集群中处理过的事务，不会被抹去？关于这点，Raft在选主阶段，提出了和ZAB类似的策略来解决：选择日志更多的服务器做leader，并给了更多选主的限制，以leader的日志为标准，同步日志。

下面主要从可理解性、效率、可用性和适用场景等几个角度进行对比分析。

###### 可理解性

众所周知，Paxos是出了名的晦涩难懂，不仅难以理解，更难以实现。

而Raft则以可理解性和易于实现为目标，Raft的提出大大降低了使用分布式一致性的门槛，将分布式一致性变的大众化、平民化，因此当Raft提出之后，迅速得到青睐，极大地推动了分布式一致性的工程应用。

###### 效率

我们主要从负载均衡、消息复杂度、Pipeline以及并发处理几个方面来对比Multi-Paxos、Raft。

负载均衡

Multi-Paxos和Raft的Leader负载更高，各副本之间负载不均衡，Leader容易成为瓶颈。

消息复杂度

Multi-Paxos和Raft选举出Leader之后，正常只需要一次网络来回就可以提交一条日志，但Multi-Paxos需要额外的异步Commit消息提交，Raft只需要推进本地的commit index，不使用额外的消息。因此消息复杂度，Raft最低，Paxos其次。

Pipeline

我们将Pipeline分为顺序Pipeline和乱序Pipeline。

**Multi-Paxos支持乱序Pipeline，Raft因为日志连续性假设，只支持顺序Pipeline。但Raft也可以实现乱序Pipeline，只需要在Leader上给每个Follower维护一个类似于TCP的滑动窗口，对应每个Follower上维护一个接收窗口，允许窗口里面的日志不连续，窗口外面是已经连续的日志，日志一旦连续则向前滑动窗口，窗口里面可乱序Pipeline**。

并发处理

Multi-Paxos沿用Paxos的策略，一旦发现并发冲突则回退重试，直到成功；Raft则使用强Leader来避免并发冲突，Follwer不与Leader竞争，避免了并发冲突。

Paxos是冲突回退，Raft是冲突避免。Paxos和Raft的日志都是线性的。

###### 可用性

Multi-Paxos和Raft均依赖Leader，Leader不可用了需要重新选举Leader，在新Leader未选举出来之前服务不可用。

Raft是强Leader，Follower必须等旧Leader的Lease到期后才能发起选举，Multi-Paxos是弱Leader，Follwer可以随时竞选Leader，虽然会对效率造成一定影响，但在Leader失效的时候能更快的恢复服务，因此Multi-Paxos比Raft可用性更好。

### 单调一致性

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zookeeper支持这个模式。

### 会话一致性

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

# 幽灵复现