一致性和复制

分布式系统中的一个重要问题是数据的复制。通常复制数据是为了增强可靠性或提高性能。其中一个主要问题是保持副本的一致性。非正式地说，这意味着当一个副本被更新时，我们需要确保其他副本也被更新;否则，副本将不再相同。在本章中，我们将详细讨论复制数据的一致性实际上意味着什么，以及实现一致性的各种方法。

我们从一个一般性的介绍开始，讨论为什么复制是有用的，以及它与可伸缩性的关系。然后我们继续关注一致性到底意味着什么。一致性模型的一个重要假设是多个进程同时访问共享数据。当读取和更新共享数据时，知道其他人也在访问该数据时，可以根据进程的期望来制定这些情况的一致性。

在大规模分布式系统中，共享数据的一致性模型通常很难有效地实现。此外，在许多情况下可以使用更简单的模型，这些模型通常也更容易实现。一个特定的情况是由以客户端为中心的一致性模型构成的，该模型从单个(可能是移动的)客户端角度关注一致性。

一致性只是问题的一半。我们还需要考虑一致性实际上是如何实现的。我们基本上需要考虑两个或多或少独立的问题。首先，我们将重点放在管理副本上，它不仅考虑副本服务器的位置，还考虑内容如何分布到这些服务器。

第二个问题是副本如何保持一致。在大多数情况下，应用程序需要很强的一致性。非正式地说，这意味着更新将在副本之间或多或少地立即传播。有多种实现强一致性的方法，我们将在单独的部分中讨论。此外，还注意了缓存协议，它是一致性协议的一种特殊情况。

作为最大的分布式系统，我们分别关注基于web的系统中的缓存和复制，特别是内容交付网络和边缘服务器缓存技术。

7.1 简介

在本节中，我们首先讨论复制数据的重要原因。我们专注于将复制作为一种实现可伸缩性的技术，并解释了为什么一致性的推理如此重要。

**复制的原因**

复制数据有两个主要原因。首先，复制数据以提高系统的可靠性。如果一个文件系统已经被复制，那么在一个副本崩溃之后，只要切换到另一个副本，就可以继续工作。此外，通过维护多个副本，可以更好地保护数据免受损坏。例如，假设有一个文件的三个副本，每个副本都执行读写操作。通过将至少两个副本返回的值视为正确的值，我们可以防止一次失败的写操作。

复制数据的另一个原因是性能。当分布式系统需要按大小或所覆盖的地理区域进行伸缩时，性能复制非常重要。当越来越多的进程需要访问但一个服务器时，就会产生性能问题。在这种情况下，可以通过复制服务器并在访问数据的进程之间分配工作负载来改进性能。

相对于地理区域的扩展也可能需要复制。其基本思想是，通过将数据副本放在使用它们的进程附近，访问数据的时间就会减少。因此，该进程所感知到的性能会增加。这个例子还说明，复制对性能的好处可能很难评估。尽管客户机进程可能会感知到更好的性能，但也可能会消耗更多的网络带宽来保持所有副本都是最新的。

如果复制有助于提高可靠性和性能，那么谁会反对呢？不幸的是，在复制数据时要付出代价。复制的问题是，拥有多个副本可能会导致一致性问题。每当修改一个副本时，该副本就与其他副本不同。因此，必须对所有副本进行修改，以确保一致性。究竟什么时候以及如何进行这些修改决定了复制的价格。

要理解这个问题，可以考虑改进Web页面的访问时间。如果不采取特殊措施，从远程Web服务器获取页面有时甚至需要几秒钟。为了提高性能，Web浏览器通常在本地存储以前获取的Web页面的副本。如果用户再次需要该页面，浏览器将自动返回本地副本。用户感知的访问时间非常好。然而，如果用户总是想要一个页面的最新版本，这种方法可能不会生效。问题是，如果同时修改了页面，修改将不会传播到缓存副本，从而使这些副本过期。

将过期副本返回给用户的一个解决方案是首先禁止浏览器保存本地副本，从而有效地让服务器完全负责复制。但是，如果在用户附近没有放置副本，则此解决方案仍然可能导致较差的访问时间。另一个解决方案是让Web服务器使每个缓存副本失效或更新，但这要求服务器跟踪所有缓存并向它们发送消息。这进而可能降低服务器的整体性能。下面我们回到性能与可伸缩性的问题。

在接下来的文章中，我们将主要关注性能方面的复制。第8章讨论了可靠性的复制。

**Replication as scaling technique**

为提高性能而进行的复制和缓存作为扩展技术得到了广泛的应用。可伸缩性问题通常以性能问题的形式出现。将数据副本放在使用它们的进程附近可以通过减少访问时间提高性能，从而解决可伸缩性问题。

需要做出的一个可能的权衡是，使副本保持最新可能需要更多的网络带宽。考虑一个进程P每秒访问一个本地副本N次，而副本本身每秒更新M次。假设更新完全刷新本地副本的前一个版本。如果N << M，也就是说，访问更新比非常低，我们就会遇到这样的情况:P永远不会访问本地副本的许多更新版本，这使得这些版本的网络通信毫无用处。在这种情况下，最好不要安装接近P的本地副本，或者应用不同的策略来更新副本。

然而，一个更严重的问题是，保持多个副本的一致性本身可能存在严重的可伸缩性问题。直观地说，当副本总是相同时，副本集合是一致的。这意味着在任何副本上执行的读取操作总是返回相同的结果。因此，当对一个副本执行更新操作时，无论在哪个副本上启动或执行该操作，应该在后续操作发生之前将更新传播到所有副本。

这种类型的一致性有时称为非正式地(而且不精确地)一致性，由同步复制提供。(在第7.2节中，我们将提供一致性的精确定义，并介绍一系列一致性模型。) 关键思想是，更新作为单个原子操作或事务在所有副本上执行。不幸的是，当操作也需要快速完成时，实现涉及大量副本的原子性本质上是困难的，这些副本可能广泛分布在大型网络中。

困难在于我们需要同步所有副本。实际上，这意味着所有副本首先需要就何时在本地执行更新达成一致。例如，副本可能需要使用Lamport时间戳决定操作的全局顺序，或者让协调器分配这样的顺序。全局同步只需要大量的通信时间，特别是当副本分布在广域网中时。

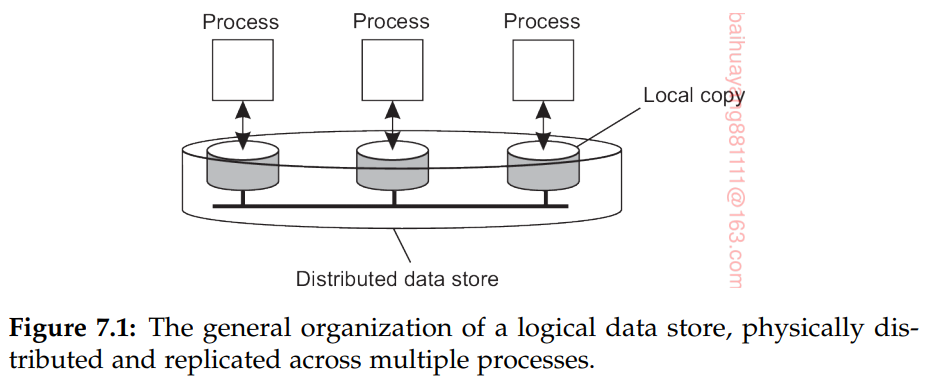
我们现在面临着进退两难的局面。一方面，通过应用复制和缓存可以缓解可伸缩性问题，从而提高性能。另一方面，为了保持所有副本的一致性，通常需要全局同步，这在性能方面有内在的代价。

在许多情况下，唯一真正的解决方案是放松一致性约束。换句话说，如果我们能够放松更新需要作为原子操作执行的要求，我们就能够避免(瞬时)全局同步，从而获得性能。付出的代价是，副本可能并不总是在所有地方都一样。事实证明，可以在多大程度上放松一致性，在很大程度上取决于对复制数据的访问和更新模式，以及使用这些数据的目的。

有一系列的一致性模型，以及通过所谓的分布和一致性协议实现模型的许多不同方法。

**7.2以数据为中心的一致性模型**

传统上，一致性是在共享数据的读写操作上下文中讨论的，这些操作通过(分布式)共享内存、(分布式)共享数据库或(分布式)文件系统来实现。这里，我们使用更广泛的术语**数据存储**。数据存储可以物理地分布在多台机器上。特别是，可以假定每个数据都存有本地副本，所以每个进程都可以进行访问。写操作被传播到其他副本，如图7.1所示。数据操作在更改数据时被分类为写操作，否则被分类为读操作。



**一致性模型**本质上是进程和数据存储之间的约定。它说，如果进程同意遵守某些规则，存储的数据承诺将正常工作。通常，对数据项执行读操作的进程期望该操作返回一个值，该值显示上次对该数据执行写操作的结果。

在没有全局时钟的情况下，很难精确地定义哪个写操作是最后一个写操作。作为一种替代方法，我们需要提供其他定义，从而产生一系列一致性模型。每个模型都有效地限制数据项上的读取操作可以返回的值。正如预期的那样，具有主要限制的应用程序很容易使用，例如在开发应用程序时，而具有次要限制的应用程序通常被认为很难在实践中使用。当然，代价是，易于使用的模型的性能不如难使用的模型。生活就是这样。Such is life~

**连续的一致性**

没有复制数据的最佳解决方案。复制数据会带来一致性问题，而这些问题通常无法有效地解决。只有我们放松一致性，才能有希望获得有效的解决方案。不幸的是，也没有放松一致性的一般规则:确切的说，只有高度依赖的应用程序才是可以容忍一致性的。

应用程序有不同的方法来指定它们可以容忍哪些不一致。Yu和Vahdat[2002]采用了一种通用的方法，通过区分三个独立的轴来定义不一致性:副本之间数值的偏差、副本之间陈旧程度的偏差以及更新操作顺序的偏差。他们将这些偏差称为形成**连续一致性**范围。

数据具有数值语义的应用程序可以使用数值偏差来测量不一致性。一个明显的例子是复制包含股票市场价格的记录。在这种情况下，应用程序可以指定两个副本的偏差不应超过0.02美元，这将是一个绝对的数值偏差。或者，可以指定一个相对的数值偏差，说明两个副本之间的差异不应超过0.5%。在这两种情况下，如果股票上涨(并且其中一个副本立即更新)而没有违反指定的数值偏差，那么副本仍然被认为是相互一致的。

数值偏差也可以理解为已应用于给定副本但尚未被其他人看到的更新的数量。例如，Web缓存可能没有看到由Web服务器执行的一批操作。在本例中，值的相关偏差也称为其权重。

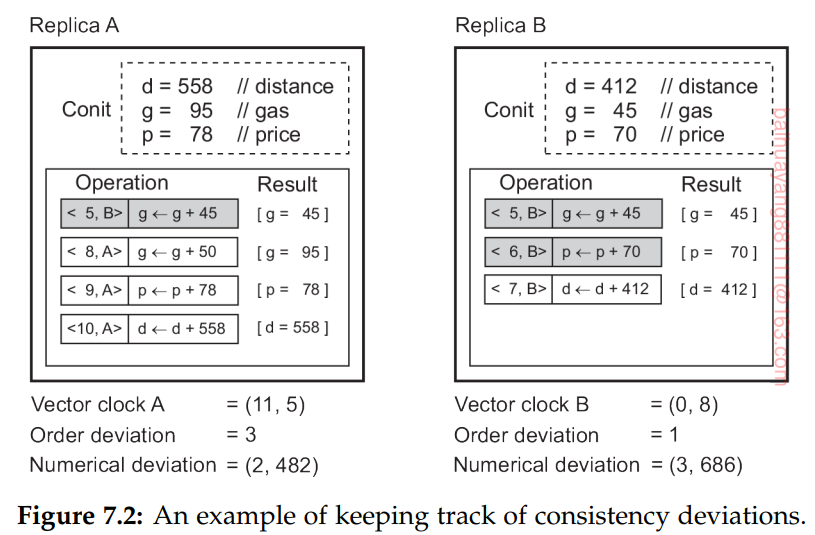
新旧偏差与副本更新的最后时间有关。对于某些应用程序，只要旧数据不太旧，就可以容忍副本提供旧数据。例如，天气报告通常在一段时间内保持相当准确，比如几个小时。在这种情况下，主服务器可能会收到及时的更新，但是可能决定只在一段时间内将更新传播到副本。

最后，在一些 应用程序类中，允许在不同的副本上执行不同的更新次序，只要这些差别是有限度的。查看这些更新的一种方法是，它们暂时应用于本地副本，等待所有副本的一起同步。因此，一些更新可能需要回滚并以不同的顺序应用，然后才能成为永久性的。直观地说，顺序偏差比其他两个一致性度量更难理解。

The notion of a conit

为了定义不一致性，Yu和Vahdat引入了一个**一致性单元**，缩写为**conit。**conit指定要测量一致性的单位。例如，在我们的股票交易所示例中，conit可以定义为表示单个股票的记录。另一个例子是单独的天气报告。

为了给出一个conit的例子，并同时说明数值和顺序偏差，请考虑跟踪车队的情况。特别是，车队老板想知道他平均为汽油付多少钱。为此，每当司机加满汽油时，他都会报告加满的汽油量(记为g)、支付的价格(记为p)以及自上次加满以来的总距离(记为变量d)。从技术上讲，g、p和d这三个变量构成了一个conit。这个conit被复制到两个服务器上，如图7.2所示，司机通过单独更新每个变量，定期向其中一个服务器报告他的汽油使用量(无需进一步考虑问题车辆)。



服务器的任务是保持conit的“一致性”被复制。为此，每个复制服务器维护一个二维矢量时钟。我们使用符号<T,R>， Ri来表示由副本R在(其)逻辑时间T执行的操作。在本例中，我们看到两个副本操作一个conit，其中包含本例中的数据项g、p和d。假设所有变量都初始化为0。副本A从副本B收到操作：



我们用灰色阴影表示该操作，表示A已将该操作提交到其本地存储。换句话说，它已经成为永久性的，不能回滚。副本A还列出了三个暂定的更新操作：<8,A>,<9,A>和<10,A>。就连续一致性而言，A有三个待提交的暂定操作，order deviation是3。类似地，在总共进行了三次操作(其中两次操作)的情况下，B的顺序为1。

从这个例子中，我们看到A的逻辑时钟值现在是11。我们假设向量的第一个分量用于A，第二个分量用于B。因为A接收到的B的最后一个操作的时间戳是5，所以A处的向量时钟将是(11,5)。同理，B处的逻辑时钟为(0,8)。在一个复制R处的数值偏差由两个部分组成:在所有其他尚未被R看到的复制上的操作数，以及相应的缺失值的总和(当然，也可能有更复杂的方案)。（就是别的机器的操作还没看到）。在我们的例子中，A看不到<6,B>和<7,B>操作，总共两个操作，值的和为70+412=486，所以值差为(2,482)。同样的，B也看不到A的三个操作，总差值是686，所以B的值差为(3,686)。

使用这些概念，可以指定特定的一致性方案。例如，我们可以通过指定一个可接受的最大值来限制这些偏差。同样，我们可能希望两个副本的数值偏差不超过1000。拥有这样的一致性方案确实需要一个副本知道它与其他副本的差异有多大，这意味着我们需要单独的通信来通知副本。基本的假设是，这种通信比保持副本同步的通信要便宜得多。诚然，如果这个假设也适用于我们的例子，那就值得怀疑了（怀疑啥？）。

虽然从概念的角度来看conits是捕获一致性需求的一种吸引人的方法，但是在将它们投入实际使用之前，有两个重要的问题需要处理。首先，为了加强一致性，我们需要协议。本章后面将讨论连续一致性协议。第二个问题是程序开发人员必须为他们的应用程序指定一致性需求。实践表明，获得这些需求可能极其困难。程序员通常不习惯处理复制，更不用说理解提供关于一致性的详细信息意味着什么。因此，必须有简单易懂的编程接口。

**操作顺序一致性**

在过去的几十年里，有大量以数据为中心的一致性模型的工作。一类重要的模型来自并行编程领域。在并行和分布式计算中，多个进程需要同时共享资源和访问这些资源，面对这一事实，研究人员试图在复制共享资源时表达并发访问的语义。我们在这里讨论的模型都处理对共享、复制数据的一致排序操作。原则上，当需要提交副本上的临时更新时，副本需要就全局(即更新的一致顺序)达成一致，因此模型增强了连续一致性。

**顺序一致性**

在下面，我们将使用一个特殊的符号，在这个符号中，我们沿着时间轴绘制进程的操作。时间轴总是水平绘制的，时间从左到右递增。我们使用符号Wi(x)来表示进程Pi将值a写入数据项x。类似地，Ri(x)b表示进程读取x并返回值b。我们假设每个数据项的初始值为NIL。当没有混淆哪个进程正在访问数据时，我们从符号W和R中省略索引。

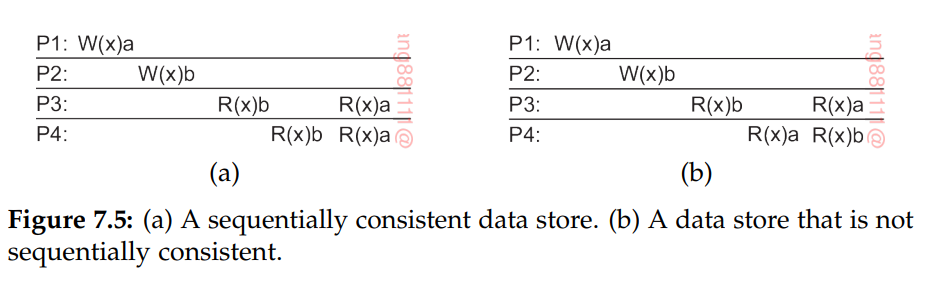
在图7.4中，p1向x写入数据a。注意，根据我们的系统模型，W1(x)a操作首先在P1本地的数据存储副本上执行，然后才传播到其他本地副本。在我们的示例中，P2稍后读取值NIL，在此之后的一段时间内读取值a(从存储的本地副本)。我们在这里看到的是，将x的更新传播到P2花费了一些时间，这完全可以接受。

**顺序一致性**是一种重要的以数据为中心的一致性模型，Lamport[1979]在多处理器系统共享内存的上下文中首次定义了该模型。当数据存储满足以下条件时，它被称为顺序一致的。

*The result of any execution is the same as if the (read and write) operations by all processes on the data store were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program.*

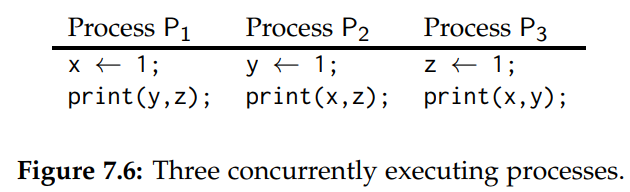
这个定义的意思是，当进程在(可能)不同的机器上并发运行时，任何有效的读写操作交错都是可接受的行为，但是所有进程都看到相同的操作交错（还是不懂，后面例子会讲解）。注意，这里没有提到时间;也就是说，数据项上没有对“最近”写操作的引用。此外，进程可以通过自己的读操作看到其他所有进程的写操作。

从图7.5中可以看出，时间没有发挥作用。考虑在相同数据项x上运行的四个进程。在图7.5（a），进程P1首先执行对x的写操作，写入a。之后（绝对时间上），进程P2同样执行x的写操作，写入b。然而，P3和P4都是先读到b值，然后读到a值。换句话说，写入b的操作先于写入a的操作。

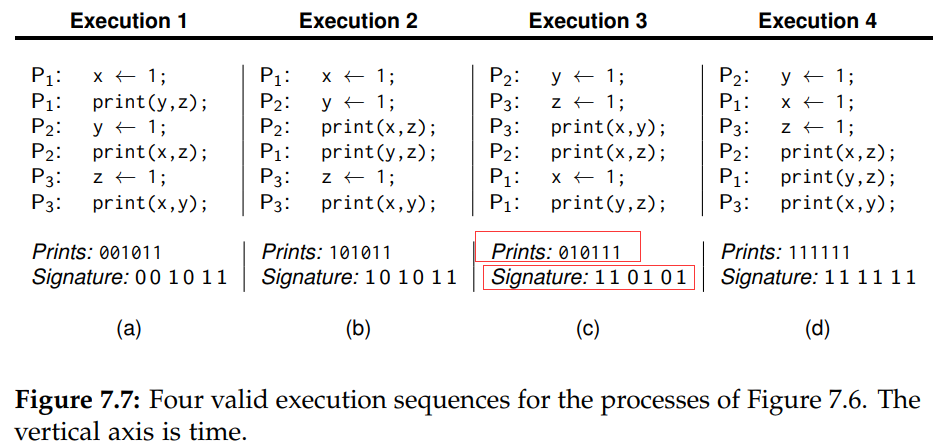


相反，图7.5(b)违反了顺序一致性，因为不是所有进程都看到相同的写操作交错。特别是在处理P3时，数据项似乎首先被更改为b，然后又被更改为a。另一方面，P4会得出最终的值是b。

为了使顺序一致性的概念更加具体，考虑三个并发执行的进程P1、P2和P3，如图7.6所示。本例中的数据项由三个整数变量x、y和z组成，它们存储在(可能是分布式的)共享的顺序一致的数据存储中。我们假设每个变量都初始化为0。在本例中，赋值对应于写操作，而print语句对应于它的两个参数的同时读取操作。所有的语句都被认为是不可分割的。



各种交错执行序列是可能的。使用6个独立语句，可能有720(6!)个可能的执行序列，尽管其中一些违反了程序顺序。考虑从x1开始的120(5!)个序列。其中一半在y =1之前已经打印(x,z)，因此违反了程序顺序。其中一半在z =1之前已经打印(x,y)，因此违反了程序顺序。因此只剩下四分之一，即30组有效的。另外30个有效序列可以从y =1开始，另外30个可以从z= 1开始，总共有90个有效的执行序列。其中的4种如图7.7所示。



在图7.7(a)中，这三个进程依次运行，首先是P1，然后是P2，然后是P3。另外三个例子展示了在时间上的交错不同但同样有效的语句。这三个进程中的每个进程打印两个变量。由于每个变量只能接受初值(0)或赋值(1)，所以每个进程生成一个2位字符串。打印后的数字是出现在输出设备上的实际输出。

如果我们按照这个顺序将P1、P2和P3的输出连接起来，就会得到一个6位的字符串，它表示语句之间的特定交错。这是作为图7.7中的签名列出的字符串。下面我们将通过每个订单的签名而不是打印输出来描述每个订单。

并不是所有64个签名模式都被允许。作为一个简单的例子，00 00 00是不允许的，因为这意味着print语句在赋值语句之前运行，违反了按程序顺序执行语句的要求。一个更微妙的例子是00 10 01。前两位00表示p打印时y和z都是0。只有当P1在P2或P3 \*之前执行两个语句时才会出现这种情况。接下来的两位，10，意味着P2必须在P1启动之后，P3启动之前运行。最后两位，01，意味着P3必须在P1开始之前完成，但我们已经看到P1必须先完成。因此，00 10 01是不允许的。

简而言之，90种不同的有效语句排序会产生各种不同的程序结果(不过小于64)，这些结果在序列一致性的假设下是允许的。进程与分布式共享系统之间的约定是进程必须接受这些结果作为有效的结果。换句话说，进程必须接受图7.7中所示的四个结果和所有其他有效结果作为有效的结果，并且必须在其中任何一个出现时正确工作。对其中一些结果有效而对其他结果无效的程序违反了与数据存储的约定，并且是不正确的。

**因果一致性**

因果一致性模型[Hutto and Ahamad, 1990]代表了弱化的顺序一致性，因为它区分了可能与因果相关的事件和不相关的事件。在前一章讨论向量时间戳时，我们已经遇到了因果关系。

如果事件b是由较早的事件a引起或影响的，因果关系要求其他人首先看到a，然后看到b。

考虑通过分布式共享数据库进行的简单交互。假设进程P1写了一个数据项x。进程p2读取x并写入y。这里读x和写y可能有因果关系，因为y的计算可能依赖于P2读x时的值。

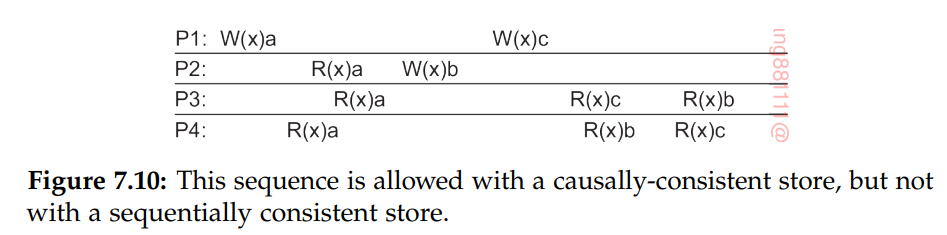
另一方面，如果两个进程同时自发地编写两个不同的数据项，则它们之间没有因果关系。没有因果关系的操作称为并发操作。要使数据存储被认为是因果一致的，必须使存储遵守以下条件：

*Writes that are potentially causally related must be seen by all processes*

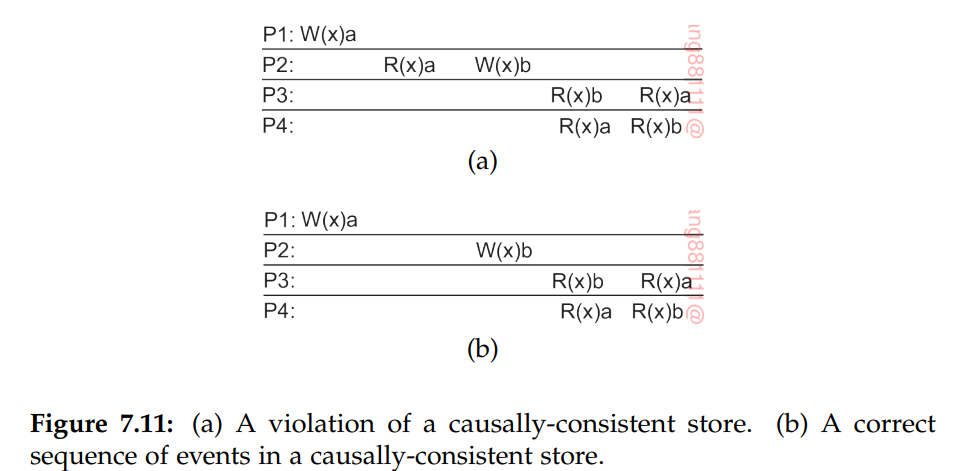
*in the same order. Concurrent writes may be seen in a different order on*

*different machines.*

作为因果一致性的一个例子，请考虑图7.10。在这里，我们有一个事件序列，它允许使用与原因一致的存储，但是禁止使用顺序一致的存储或严格一致的存储。需要注意的是，写入W2(x)b和W1(x)c是并发的，所以并不要求所有进程都以相同的顺序查看它们：



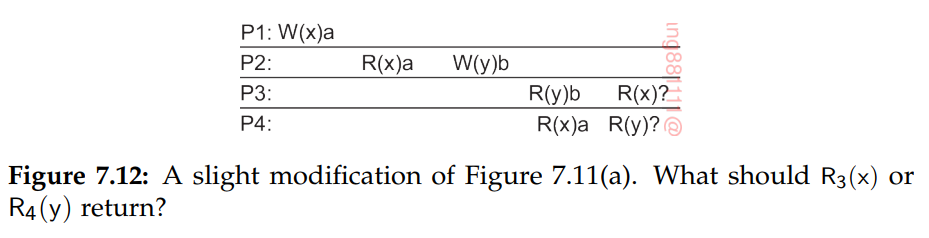
现在考虑第二个例子。在图7.11(a)中，我们有W2(x)b，它可能依赖于W1(x)a，因为将值b写到x中可能是R2(x)a先前读取值的计算结果。这两个写是因果相关的，所以所有进程必须以相同的顺序查看它们（即现在写入是p1->x(a) p2->x(b)，其他进程读取x 必须是 先a 后 b）。因此，图7.11(a)是不正确的。（3,4 必须以相同顺序读取a和b）。另一方面，在图7.11(b)中已经删除了读操作，所以W1(x)a和W2(x)b现在是并发写操作。原因一致的存储不需要对并发写进行全局排序，因此图7.11(b)是正确的。注意，图7.11(b)反映了顺序一致的存储不能接受的情况。



实现因果一致性需要跟踪哪些流程看到了哪些写入。有许多微妙的问题需要考虑。为了说明这一点，假设我们将图7.11(a)中的W2(x)b替换为W2(y)b，同样地，R3(x)b替换为R3(y)b。这种情况如图7.12所示。

首先来看R3(x)。进程P3在R3(y)b之后执行此操作。在这一点上我们确定W(x)a发生在W(y)b之前。特别是W(x)a->R (x)a->W(y)b，意思是如果我们要保持因果关系，从y中读取b后读取x只能返回a。如果系统将nil返回到P3，就违反了因果关系的保存。

R4 (y)呢?它能否返回y的初始值，即NIL? 答案是肯定的:虽然我们在没有从y中读取b的情况下 有W(x)a->W(y)b这层关系，进程P4仍然可以合理地观察到W(x)a的发生独立于y的初始化。



在实现方面，保留因果关系引入了一些有趣的问题。例如，考虑图7.12中的中间件基础进程P3。当这个中间件从读取y返回值b时，它必须知道关系W(x)a ->W (y) b。换句话说，当y的最新值传播到P3的中间件时，至少也应该传播关于y依赖关系的元数据（这里指W(x)a->W(y)b）。或者，传播也可以与更新P3节点上的x一起完成。总的来说，底线是我们需要一个依赖关系图，哪个操作依赖于哪个操作。当相关数据也在本地存储时，可以删除这样的图。

注：和上文非结果依赖顺序一致性区别：

非结果依赖，顺序一致性强调所有的进程必须执行出来的结果都保持相同的顺序，可以a b 也可以 b a，而结果依赖如7.11(a)，由于p2读取到了p1的写入值，则p2的写入b必须位于p1的写入a之后，则 p3 p4 读取的值 必须保持 先a后b。

**分组操作**

许多一致性模型是在基本读写操作级别定义的。这种粒度级别是出于历史原因：这些模型最初是为共享内存多处理器系统开发的，实际上是在硬件级实现的。

在许多情况下，这些一致性模型的细粒度与应用程序提供的粒度不匹配。我们看到共享数据的程序之间的并发性通常通过互斥和事务的同步机制得到控制。实际上，在程序级，读写操作由一对操作ENTER\_CS和LEAVE\_CS括起来。成功执行ENTER\_CS的进程将确保其本地存储中的所有数据都是最新的。此时，它可以安全地在该存储上执行一系列读和写操作，然后通过调用LEAVE\_CS结束这些操作。ENTER\_CS和LEAVE\_CS之间的数据和指令表示为一个临界区。实际上，在一个程序中，由一系列读和写操作操作的数据受到保护，不受并发访问的影响，并发访问将导致看到的不是作为一个整体执行该系列的结果。换句话说，括号将一系列读写操作转换为原子执行的单元，从而提高了粒度级别。

为了达到这一点，我们确实需要有关于ENTER\_CS和LEAVE\_CS操作的精确语义。这些语义可以根据共享**同步变量**或简单的**锁**来表示。锁具有与其关联的共享数据项，并且每个共享数据项最多与一个锁关联。在粗粒度同步的情况下，所有共享数据项都只关联到一个锁。当每个共享数据项都有自己独特的锁时，就可以实现细粒度同步。当然，这只是将共享数据与锁关联的两个极端。当一个进程进入临界区时，它应该获得相关的锁，同样地，当它离开临界区时，它释放这些锁。

每个锁都有一个当前所有者，即最后获得它的进程。当前不拥有锁但希望获得锁的进程必须向当前所有者发送一条消息，要求获得与锁关联的数据的所有权和当前值。虽然具有对锁的独占访问权，但允许进程执行读和写操作。几个进程也可能同时具有对锁的非排他性访问，这意味着它们可以读，但不能写关联的数据。当然，当且仅当没有其他进程具有独占访问权时，才可以授予非独占访问权。

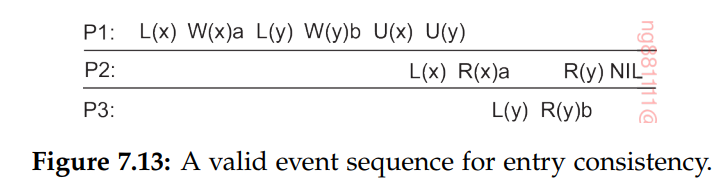
我们现在要求满足下列标准[Bershad et al.， 1993]:

只有当对其关联的共享数据的所有更新都完成时，获取锁才能成功。

只有当没有其他进程具有对锁的独占或非独占访问时，对锁的独占访问才能成功。

只有在完成了以前的任何独占访问(包括更新锁的相关数据)之后，才允许对锁进行非独占访问。

请注意，为了坚持顺序一致性，我们实际上要求将锁的使用线性化。图7.13显示了一个称为条目一致性的示例。我们将锁与每个数据项单独关联。我们使用符号L(x)作为获取x锁的缩写，即锁定x。同样，U(x)表示释放x上的锁，或解锁它。在这种情况下，P1锁定x，改变x一次，然后锁定y。进程P2还为x而不是y获取锁，因此它将为x读取值a，但可能为y读取NIL。但是，由于进程P3首先为y获取锁，所以当y被P1解锁时，它将读取值b。这里需要注意的是，每个进程都有一个变量的副本，但是这个副本不需要立即或自动更新。当锁定或解锁一个变量时，进程显式地告诉底层分布式系统，需要同步该变量的副本。因此，没有锁定的简单读取操作可能会导致读取实际上已过期的本地值。



条目一致性的编程问题之一是正确地将数据与锁关联起来。一种直接的方法是显式地告诉中间件将访问哪些数据，通常通过声明哪些数据库表将受到事务的影响来实现。在基于对象（object-based）的方法中，我们可以将一个惟一的锁与每个声明的对象关联起来，从而有效地序列化对此类对象的所有调用。

**一致性和连贯性**

在这一点上，澄清两个密切相关的概念之间的区别是有用的。到目前为止，我们所讨论的模型都处理这样一个事实:许多进程对一组数据项执行读写操作。一致性模型描述了当多个进程同时对该数据进行操作时，对该集合的期望。然后，如果集合符合模型所描述的规则，那么它就是一致的。

当数据一致性涉及到一组数据项时，**一致性模型**描述了只能为单个数据项保存的内容[Cantin et al.， 2005]。在本例中，我们假设复制了一个数据项;当不同的副本遵守由其关联一致性模式定义的规则时，它被称为是一致的。流行的模型是顺序一致性模型，但现在只应用于单个数据项。实际上，这意味着在并发写的情况下，所有进程最终将看到相同的更新顺序。

**最终一致性**

进程实际在多大程度上以并发方式运行，以及需要在多大程度上保证一致性，这些可能会有所不同。在许多示例中，并发性仅以受限制的形式出现。例如，在许多数据库系统中，大多数进程几乎从不执行更新操作;他们主要从数据库中读取数据。只有一个或很少的进程执行更新操作。只有一个或很少的进程执行更新操作。接下来的问题是，对于只读进程，应该以多快的速度提供更新。在全球运营内容交付网络的时代，开发人员常常选择缓慢地传播更新，隐式地假设大多数客户机总是重定向到相同的副本，因此永远不会出现不一致。

另一个例子是网络。实际上，在所有情况下，Web页面都由一个权威机构更新，例如Web管理员或页面的实际所有者。通常不需要解决写-写冲突。另一方面，为了提高效率，通常将浏览器和Web代理配置为将获取的页面保存在本地缓存中，并在下一次请求时返回该页面。这两种类型的Web缓存的一个重要问题是，它们可能返回过期的Web页面。换句话说，与实际Web服务器上可用的页面相比，返回给请求客户机的缓存页面是一个旧版本。事实证明，许多用户发现这种不一致是可以接受的(在一定程度上)，只要他们只能访问相同的缓存。实际上，他们仍然不知道已经发生了更新，就像以前的内容交付网络一样。

另一个例子是一个世界性的命名系统，比如DNS。DNS名称空间被划分为域，每个域被分配给一个命名机构，该机构充当该域的所有者。只有该权限的拥有者才能更新名称空间的一部分。因此，由于两个操作都希望对相同的数据执行更新而产生的写-写冲突，永远不会发生。惟一需要处理的情况是读写冲突，在这种情况下，一个进程希望更新数据项，而另一个进程同时尝试读取该数据项。事实证明，在这种情况下，以延迟的方式传播更新通常是可以接受的，这意味着读取过程只有在更新发生一段时间之后才会看到更新。

这些例子可以看作是(大规模)分布式和复制数据库的案例，这些数据库能够容忍相对较高程度的不一致性。它们的共同点是，如果长时间不进行更新，所有副本将逐渐变得一致，也就是说，存储的数据完全相同。这种形式的一致性称为最终一致性[Vogels, 2009]。

最终一致的数据存储具有这样的特性:在没有写-写冲突的情况下，所有副本都将收敛到彼此的相同副本。最终的一致性本质上只需要保证更新传播到所有副本。当假定只有一小组进程可以执行更新时，写-写冲突通常相对容易解决。在实践中，我们经常看到在冲突的情况下，一个特定的写操作(全局)被声明为“赢家”，覆盖了任何其他冲突写操作的效果。因此，最终一致性的实现成本通常很低。

**7.3以客户为中心的一致性模型**

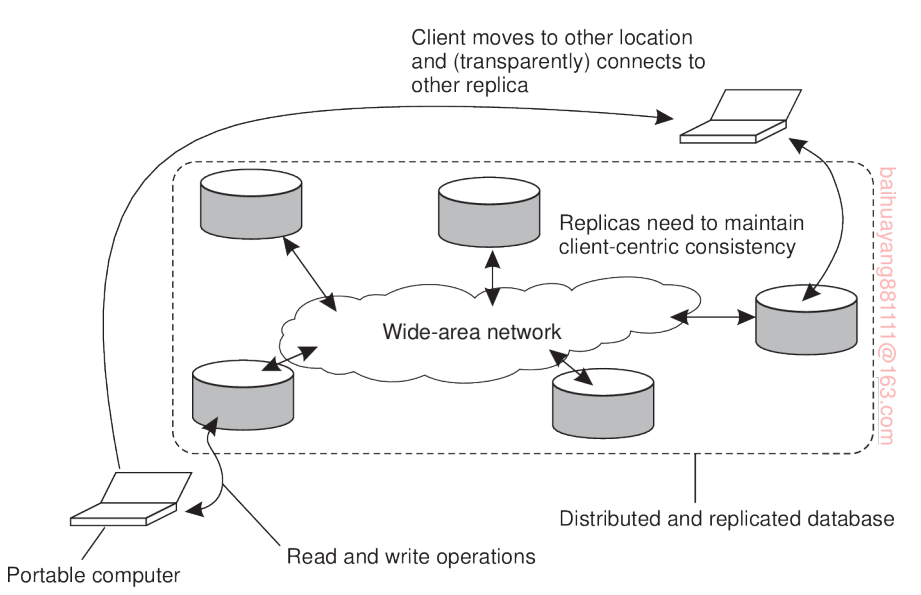
以数据为中心的一致性模型旨在为数据存储提供系统范围的一致视图。一个重要的假设是，并发进程可能同时更新数据存储，并且有必要在面对这种并发时提供一致性。例如，在基于对象的条目一致性的情况下，数据存储确保在调用对象时，向调用过程提供对象的副本，该副本反映到目前为止对对象所做的所有更改，可能是由其他进程所做的更改。在调用期间，还保证没有其他进程可以干扰，即为调用进程提供互斥访问。

能够在保持强一致性的同时处理共享数据上的并发操作是分布式系统的基础。出于性能原因，只有当流程使用诸如事务或同步变量之类的机制时，才可能保证强一致性。同样，也许不能保证强一致性，需要接受较弱的形式，例如因果一致性与最终一致性的结合。

**7.3以客户为中心的一致性模型**

以数据为中心的一致性模型旨在为数据存储提供系统范围的一致视图。一个重要的假设是，并发进程可能同时更新数据存储，并且有必要在面对这种并发时提供一致性。例如，在基于对象的条目一致性的情况下，数据存储确保在调用对象时，向调用过程提供对象的副本，该副本反映到目前为止对对象所做的所有更改，可能是由其他进程所做的更改。在调用期间，还保证没有其他进程可以干扰，即为调用进程提供互斥访问。能够在保持强一致性的同时处理共享数据上的并发操作是分布式系统的基础。出于性能原因，只有当流程使用诸如事务或同步变量之类的机制时，才可能保证强一致性。

在本节中，我们将研究一类特殊的分布式数据存储。我们所考虑的数据存储的特性是不考虑同步更新，或者说当更新发生时，可以假定这些更新能被容易的解决。大多数操作都需要读取数据。这些数据存储提供了一个弱一致性模型，比如最终一致性。通过引入特殊的以客户为中心的一致性模型，可以以一种相对节省成本的方式隐藏许多不一致性。



如果客户能总是访问同一个副本，最终一致性的存储将会工作的很好。然而，当在短时间内访问不同的副本时，就会出现问题。通过考虑移动用户访问分布式数据库，可以很好地说明这一点，如图7.15所示。

例如，移动用户Alice通过透明方式连接到其中一个副本来访问数据库。换句话说，在Alice的移动设备上运行的应用程序并不知道它实际在哪个副本上运行。假设Alice执行了几个更新操作，然后再次断开连接。稍后，她可能在移动到不同的位置或使用不同的访问设备之后再次访问数据库。此时，她可能连接到一个与以前不同的副本，如图7.15所示。但是，如果之前执行的更新尚未传播，Alice将注意到不一致的行为。以前做过的更改，但现在似乎什么都没有发生。

这个例子是最终一致的数据存储的典型例子，这是因为用户有时可能在更新还没有完全传播的情况下操作不同的副本。可以通过引入以**客户为中心的一致性**来缓解这个问题。本质上，以客户端为中心的一致性为单个客户端提供了关于该客户端访问数据存储的一致性的保证。对于不同客户机的并发访问，不提供任何保证。如果Bob修改与Alice共享但存储在不同位置的数据，我们可能很容易创建写-写冲突。而且，如果Alice和Bob都不能在一段时间内访问同一个位置，那么这种冲突可能需要很长时间才能被发现。

以客户端为中心的一致性模型源于Bayou上的工作，更普遍的是来自移动数据系统。Bayou是为移动计算而开发的数据库系统，它假定网络连接不可靠，并且会受到各种性能问题的影响。无线网络和跨大区域的网络，如因特网，属于这一类。

Bayou基本上区分了四种不同的一致性模型。为了解释这些模型，我们再次考虑物理上分布在多台机器上的数据存储。当一个进程访问数据存储时，它通常会连接到本地(或最近的)可用副本，尽管在原则上，任何副本都可以。所有读写操作都在该本地副本上执行。更新最终会传播到其他副本。

以客户端为中心的一致性模型使用以下符号进行描述。版本xi是从初始化到一系列写操作的集合，WS(xi)。通过将写操作附加到该系列中，我们获得了另一个版本xj，并说xj遵循xi。我们用符号WS(xi;xj)表示xj遵循xi。如果我们不清楚xj是否遵循xi，使用WS(xi|xj)。

**单调读取**

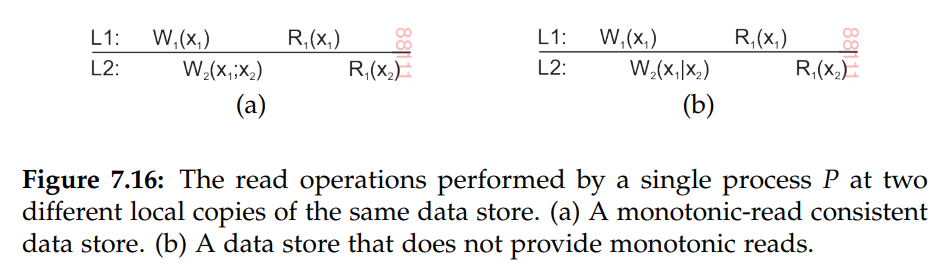
第一个以客户端为中心的一致性模型是单调读取模型。如果下列条件成立，则(分布式)数据存储提供**单调读一致性**：

*如果进程读取数据项x的值，则该进程对x执行的任何后续读取操作都将始终返回相同的值或更近期的值*

换句话说，单调读一致性保证了一个进程一旦看到x的值，它就永远不会看到旧版本的x。

作为单调读取非常有用的例子，考虑一个分布式电子邮件数据库。在这样的数据库中，每个用户的邮箱可以跨多台机器分布和复制。邮件可以进入任何位置的邮箱。然而，更新会发生的迟一些。只有当副本需要某些数据以保持一致性时，这些数据才传播到该副本。假设一个用户在旧金山阅读他的邮件。假设只读取邮件不会影响邮箱，即不会删除消息、将消息存储在子目录中，甚至标记为已经读取，等等。当用户稍后飞往纽约并再次打开邮箱时，单调读一致性保证了旧金山邮箱中的消息在纽约打开时也在邮箱中。

使用类似于以数据为中心的一致性模型的符号，单调读取一致性可以用图形表示，如图7.16所示。我们现在在示例L1和L2中显示本地数据存储，而不是沿着垂直轴显示进程。一个写或读操作由执行该操作的进程索引，即W1(x) A表示进程P1将值A写入x。由于我们对共享数据项的特定值不感兴趣，而是对它们的版本感兴趣，所以我们使用符号W1(x2)来表示进程P1在不了解其他版本的情况下生成了版本x2。W2(x1;x2)表示进程P2负责生成x1之后的版本x2。W2(x1,x2)表示在生产x1版本时，进程p2并发的生产版本x2（这可能会产生写写冲突）。R1(x2)表示P1读取x2版本。



在图7.16(a)中，进程P1首先在L1处对x执行写操作，生成版本x1，然后读取这个版本。在L2进程P2跟随版本x1，生成版本x2。当进程P1移动到L2并再次读取x时，它会发现一个更近期的值，至少考虑了之前的写操作。

图7.16(b)为违反单调读一致性的情况。当进程P1在L1处读取x1之后，它将在L2处执行操作R1(x2)。然而，已知进程P2在L2上执行的前一个写操作W2(x1|x2)生成的版本与x1不同。因此，已知P1在L2处的读操作不包括在L1处执行R1(x1)时的写操作的影响。

（个人理解：a是单调读表示 p1在L1的写记录会在L1处读取到，所以根据单调读规则，其结果应在其他存储地方读取到，而这里L2是可以读取到的，对x2的写入是依据x1的。对于b则有可能在L2读取不到x1的结果，所以不是单调读）。

**单调写**

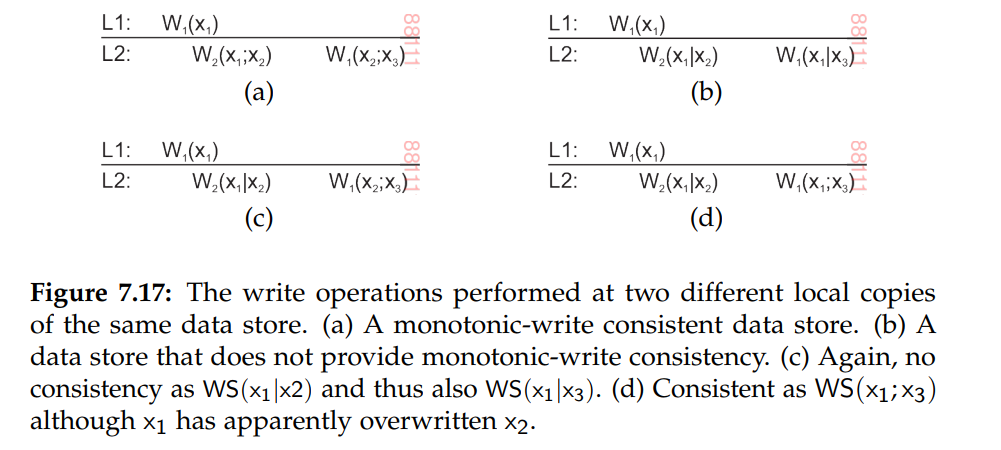
在许多情况下，按正确的顺序将写操作传播到数据存储的所有副本是非常重要的。此属性以单调写一致性表示。在单调写一致性存储中，有以下条件:

*数据项x上的进程的写操作先于同一进程对x上的任何后续写操作完成*

更正式的说，如果通过Pk进程有两个连续的操作Wk(xi)和Wk(xj)，那么无论Wk(xj)在哪里发生，我们都将有WS(xi;xj)。因此，完成一个写操作意味着执行后续操作的副本反映了由相同进程执行的前一个写操作的效果，无论该操作是在何处开始的。换句话说，对项目x的副本执行写操作的条件是，该副本必须通过该进程之前的任何写操作(可能发生在x的其他副本上)更新到最新。如果需要，新写的必须等待旧的写完。

注意，单调写入一致性类似于以数据为中心的FIFO一致性。FIFO一致性的本质是相同进程的写操作在任何地方都按照正确的顺序执行。这种顺序约束也适用于单调写，只是我们现在只考虑单个进程的一致性，而不是并发进程的集合。

当每个写操作完全覆盖x的现值时，不需要将x的副本更新到最新。然而，写操作通常只对数据项的部分状态执行。例如，考虑一个软件库。在许多情况下，通过替换一个或多个函数来更新这样的库，从而生成下一个版本。在单调写入一致性下，保证如果对库的副本执行更新，则将首先执行前面的所有更新。最终得到的库将成为最新的版本，并包含所有导致该库以前版本的更新。



单调写一致性如图7.17所示。在图7.17(a)中，进程P1在L1处对x执行写操作，表示为操作W1(x1)。之后，P1对x执行另一个写操作，但这次是在L2处，表示为W1(x2;x3)。P1在L2处生成的版本来自于流程P2的更新，后者基于版本x1，这个操作表示为W2（x1;x2）。为了保证单调写一致性，有必要将之前在L1处的写操作传播到L2，并进行可能的更新。

相比之下，图7.17(b)显示了一种不保证单调写入一致性的情况。与图7.17(a)相比，缺少的是在生成另一个版本的x之前x1到L2的传播，用W2操作表示(x1 | x2)。在本例中，进程P2生成了一个到x1的并发版本，在此之后，进程P1只生成版本x3，但仍然与x3产生并发关系。图7.17(c)所示的情况稍微微妙一些，但仍然违反单调写入一致性。进程P1现在生成版本x3，而这是依据x2开始的。而，因为x2没有包含导致x1的写操作，即WS(x1|x2)，所以我们也有WS(x1|x3)。

图7.17(d)显示了一个有趣的情况。操作W2(x1|x2)与x1并发生成x2版本。然而，后来的进程P1生成了版本x3，但显然是基于版本x1在L2中可用的事实。x1是如何以及何时被转移到L2的还没有确定，但是在任何情况下，都是与x2版本创建了write-write冲突，并以有利于x1的方式解决了这个冲突（因为x3是基于x1的）。其结果是，图7.17(d)所示的情况符合单调写一致性的规则。但是，请注意，进程P2在L2上的任何后续写入(没有读取版本x1)都将立即再次违反一致性。

注意，根据单调写一致性的定义，相同进程的写操作按初始顺序执行。单调写的一种较弱的形式是，只有在前面所有的写操作也都执行过之后才能看到写操作的效果，但可能不是按照它们最初开始的顺序执行的。这种一致性适用于写操作是可交换的情况，因此实际上没有必要进行排序。

**Read your writes**

如果符合以下条件，则数据存储将提供“read-your-writes”一致性:

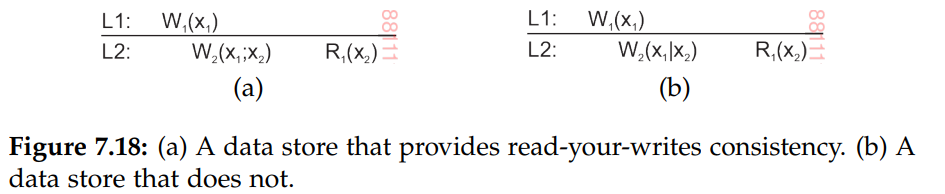
*同一个进程中，对数据项x的写操作，总是能够被后继读操作观察到*

换句话说，无论读操作发生在哪里，写操作总是在同一进程的后继读操作之前完成。

在更新Web文档并随后查看效果时，有时会体验到读写一致性的缺失。更新操作经常通过标准编辑器或文字处理器进行，这些操作可能嵌入到内容管理系统中，然后将新版本保存到Web服务器共享的文件系统中。用户的Web浏览器可能在从本地Web服务器请求该文件之后访问该文件。然而，一旦获取了文件，服务器或浏览器通常会缓存一个本地副本以供后续访问。因此，当Web页面更新时，如果浏览器或服务器返回缓存的副本而不是原始文件，用户将不会看到效果。Read-your-wirtes 一致性可以保证，如果编辑器和浏览器集成到一个程序中，那么当页面更新时缓存将失效，从而获取和显示更新后的文件。

更新密码时也会出现类似的效果。如果需要访问网页图书馆，则需要一个带密码的账户。然而，更改密码可能需要一些时间才能生效，其结果是用户可能在几分钟内无法访问库。延迟的原因是使用一个单独的服务器来管理密码，然后可能需要一些时间将(加密的)密码传播到构成库的各个服务器。

图7.18(a)显示了提供读写一致性的数据存储。注意，图7.18(a)与图7.16(a)非常相似，只是一致性现在由进程P1的最后一次写操作决定，而不是由进程P1的最后一次读操作决定。



在图7.18(a)中，进程P1执行一个写操作W1(x1)，然后在另一个本地副本上执行一个读操作。read -your-write一致性保证了写操作的效果可以被后续的读操作看到。它由W2（x1;x2）表示，它表示一个进程P2生成了一个新版本的x，但它是基于x1的。相反，在图7.18(b)中，进程P2与x1并发生成一个版本，表示为W2(x1|x2)。这意味着在生成x2时，进程P1之前的写操作的影响没有传播到L2。当P1读取x2时，它不会看到L1处自己写操作的效果。

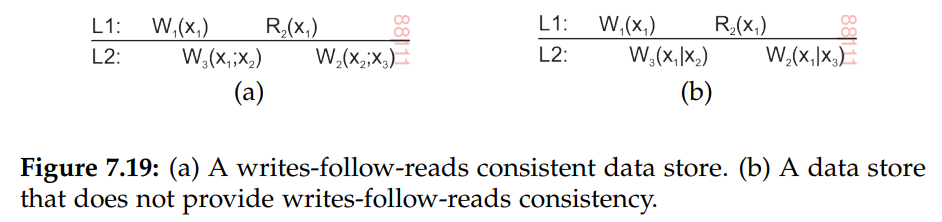
**Writes follow reads**

最后一个以客户端为中心的一致性模型是将更新作为之前读取操作的结果传播的模型。如果下列条件成立，则数据存储被称为提供 **writes-follow-reads** 一致性。

*一个进程对数据项x执行的写操作在该进程之前对x执行的读操作之后，将保证在被读的x的相同或更近的值上执行*

换句话说，一个进程对数据项x的任何后续写操作都将在一个x的副本上执行，该副本是最新的，且该进程最近读取的值是最新的。

write -follow-read一致性可用于确保网络新闻组的用户只有在看过原始文章之后才能看到对文章的反应。要理解这个问题，假设用户首先阅读文章a。然后，她发布了一个响应B。通过要求writes-follow-reads的一致性，只有在写入了A之后，才会将B写入新闻组的任何副本。注意，只阅读文章的用户不需要任何特定的以客户端为中心的一致性模型。write -follow-read的一致性保证了对文章的响应只存储在本地副本中，前提是原始副本也存储在本地副本中。



这个一致性模型如图7.19所示。在图7.19(a)中，进程P2读取本地副本L1上的版本x1。这个版本的x之前由进程P1通过操作W1(x1)在L1生成。该版本随后传播到L2，并由另一个进程P3使用，生成一个新的版本x2，表示为W3(x1;x2)。当进程P2在迁移到L2之后更新它的x版本时，我们知道它将操作从x1开始的版本，表示为W2(x2;x3)。因为我们还有W3(x1;x2)我们知道wS(x1;x3)。（R2在L1读取了x1，而W2在L2更新了x3，而x3依据x2，x2依据x1，所以满足）

图7.19(b)所示的情况有所不同。进程P3同时生成一个版本x2和版本x1。因此，当P2在读取x1之后更新x时，它将更新一个以前没有读取过的版本。这将违反Writes-follow-reads的读取一致性。（W2应该读取x2）