

## 23 | 知其然，知其所以然：数据的持久化和一致性

2019-11-01 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 22:18 大小 15.33M



你好，我是四火。

我想你很可能已经使用过许多存储层的技术了，例如缓存、文件、关系数据库，甚至一些云上 key-value 的存储服务，但就如同我之前提到的那样，某项具体技术总是相对好学，可对于全栈知识系统地学习，也包括持久层的学习，是一定要立足于技术的基础、原理和本质的。今天，我们要讲的就是其中之一 —— 一致性（Consistency）。

数据的可用性和一致性是很多工程师几乎每天都会挂在嘴边的概念，在存储系统的技术选型上面，一致性将会是我们一个重要的衡量因素，而在持久层架构设计上面，它也将是最重要的思考维度。

数据的一致性不但是数据持久化的一个核心内容，也是学习的一个难点，希望我们一起努力，从原理上去彻底理解它，并学习一些常见的应用模式，做到“知其然，知其所以然”，我们一起把这个难啃的骨头给啃下来。

## 概念和背景

数据持久化，本质上就是把内存中的数据给转换并写入指定的存储系统中，这个过程是保证数据不丢失的基本方式，而这个存储系统可以具备很多种形式，可以是网络、硬盘文件，也可以是数据库，还可以是前面两讲提到的某种形式的缓存。

你也许听说过对于一致性的不同解释，而我们在谈论数据持久化的时候讲到的一致性，我认为简单来说，指的就是在存储系统中，客户端对数据的读写行为都是可以预期、符合一定规则的。这里有两个值得注意的方面：

**可以预期和符合规则，而不是说读到的数据是“一致的”“准确的”或是“最新的”，**是因为存在不同的一致性模型，数据对一致性遵从的程度和规则都不同，下文我会讲到。

一致性判断的视角要从客户端来看，也就是说，**存储系统实际存储的数据可以在某些时候不遵从我们所要求的一致性，而只需要保证存储系统的客户端能读取到一致的数据就可以了。**举例来说，某一个数据更新的过程中，对于存储系统来说，新数据其实已经写入，但由于事务还未提交，这时客户端读到的还是老数据。

我想这个概念并没有什么特殊之处，但是，这里面隐含了一个事情，就是说，为什么要有数据备份呢？

为了可用性（Availability）。

**服务为了高可用，就要部署多个节点；数据为了高可用，就要存放多个备份。**这里的数据，既包括数据本身，又包括数据的读写服务，这是因为：

要让数据不丢失，冗余几乎是唯一的办法，因为再好的存储介质也架不住设备老化和各种原因的破坏；

同理，为了数据访问服务能保持可用，包括保证足够的性能，必须要提供多个节点的读写操作服务，于是，我们不得不创建多个数据副本。

那么一环扣一环，如果只有一份数据，是不存在一致性问题的，因为数据自己也只有一份，没法存在不一致，但有了数据副本，一致性就成为了课题。

## 一致性模型

你很可能已经听说过这三种一致性模型，下面我们来分别了解一下。

**强一致性 (Strong Consistency)：**强一致性要求任意时间下，读操作总是能取得最近一次写操作写入的数据。

注意，这里依然是从存储系统客户端的角度来描述的，**即便如强一致性的限制，也只要求在读取的时候能读到“最新”的数据就可以了，至于这个在上次写操作之后、这次读操作之前，对存储系统内部的数据是否是“最新”的并无要求。**我们经常使用的传统关系型数据库，比如 Oracle、MySQL，它们都是符合强一致性的。

**弱一致性 (Weak Consistency)：**弱一致性和强一致性相反，**读操作并不能保证**可以取得最新一次写操作写入的数据，也就是说，客户端可能读到最新的数据，也可能读不到最新的数据。

这个“并不能保证”就有点“搞笑”了——都不确定能不能读到最新值，那它有什么用？其实它的应用也挺广泛的，最常见的例子就是缓存，比如一个静态资源被浏览器缓存起来，那么这之后只要是从缓存内取得的数据，使用者其实根本不知道这个数据是不是最新的，因为即便它实际有了更新，服务端也不会通知你。

**最终一致性 (Eventual Consistency)：**最终一致性介于强一致性和弱一致性之间，写操作之后立即进行读操作，可能无法读到更新后的值，但是如果经过了一个指定的时间窗口，就能保证可以读到那个更新后的值。

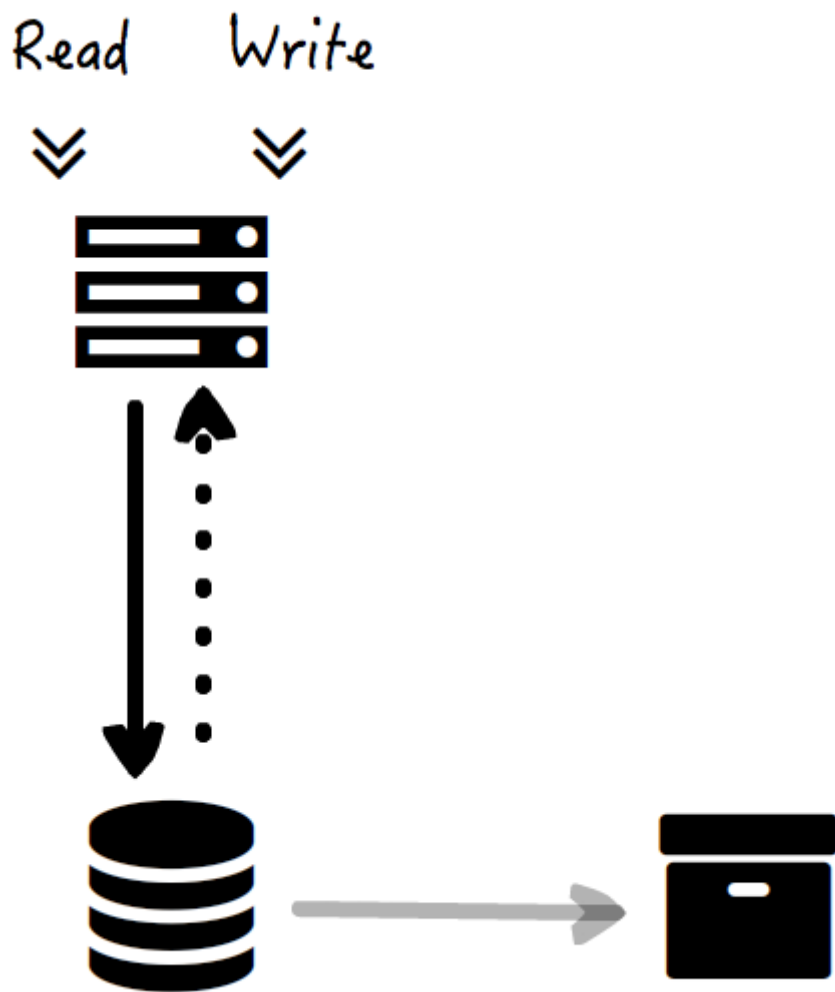
最终一致性可能是我们日常生活中最常见的一致性模型了。比如搜索引擎，搜索引擎的爬虫会定期爬取数据，并更新搜索数据，因此如果你的网站只是刚刚更新，可能还无法搜到这个更新内容，但只要是过了一定的时间窗口，它就会出现在搜索结果中了。

## 数据高可用的架构技术

接下来，我们探讨互联网应用中最常见的几种架构技术，它们都是用以解决数据可用性的问题，就如同我们在上文中所提到的那样，既包括数据本身的可用性，又包括数据读写服务的可用性。

## 1. 简单备份

简单备份（Backup）指的就是定期或按需对存储系统中的数据全量或增量进行复制，并保存为副本，从而降低数据丢失风险的一种方式。



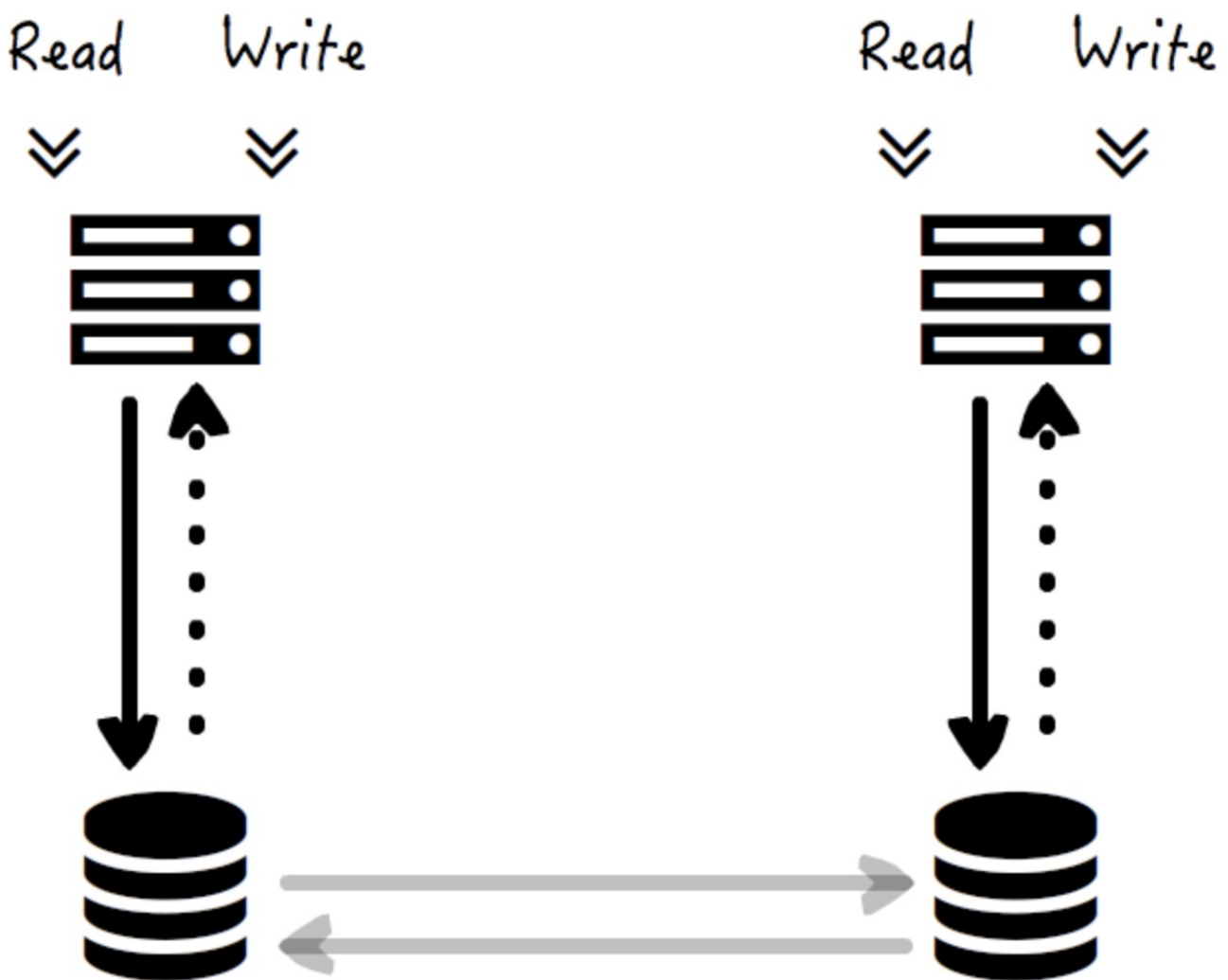
这是一种实现上最简单的技术，在个人电脑上极其常见，但即便在工业界，依然有大量的应用场景。比方说 Amazon RDS（将关系数据库搬到云上）的 Snapshot 技术，可以定期将所有数据导出为一份副本。在 CPU 和 I/O 等资源不成为瓶颈的情况下，因为是异步进行的，简单备份往往对存储系统读写操作的影响很小。

但是，这种方式存在存储系统的单点故障问题，一旦存储系统挂掉了，服务也就中断了，因此基本没法谈可用性。

你可能会说，可用性的话，可以给访问存储的 Web 服务器做双机备份啊。没错，但那解决的是 Web 服务器可用性的问题，并不是我们这里最关心的数据可用性的问题，数据存储依然是单点的。同时，如果什么时候存储系统挂掉了，那么只能恢复到最近一次的备份点，因此可能丢失大量的数据。

## 2. Multi-Master

Multi-Master 架构是指存在多个 Master（主）节点，各自都提供完整的读写服务，数据备份之间的互相拷贝为了不影响读写请求的性能，通常是异步进行的。



从图中，你可以看到，如果某一主节点对应的存储服务挂掉了，那么还有另一个主节点可以提供对应服务，因此，这种方式是可以提供高可用服务的。图中只放了两个主节点，但是其实是可以放置多个的。

关于一致性，通常情况下节点之间的数据互拷贝是异步进行的，因此是最终一致性。需要说明的是，这个数据互拷贝理论上也是可以做到同步进行的，即将数据拷贝到所有其它的主节点以后再将响应返回给用户，而且那种情况下就可以做到强一致性，不过实际却很少有这样做的，这是为什么呢？

第一个原因，显而易见，同步的数据拷贝会导致整体请求响应的时延增加。

第二个，也是更重要的原因，如果有节点异常，这个拷贝操作就可能会超时或失败，这种情况下，你觉得存储系统应该怎样对待这个错误？显然，存储系统会陷入两难的境地。

如果系统容许错误发生，不返回错误给用户，那么强一致性就无法保证，既然无法保证，那么这个拷贝过程就完全可以设计成异步的，因为既然无论如何也无法保证强一致性，这个同步除了增加时延以外，并未带来任何明显的好处。

如果系统不容许错误发生，即返回错误给用户，一致性就被严格保证了，但是这样的话，整个存储系统就不再是高可用了，因为任何一个主节点的不可用，就会导致其它任意主节点向其拷贝数据的失败，进而导致整个系统都变得不可用。我们使用多个主节点的目的就是要提高可用性，而现在这样的设计和高可用性的目的就自相矛盾了。

其实，对待这个节点间的数据拷贝错误，还有第三种方式，它结合了上述二者的优点。我先卖个关子，我们在下面 Master-Slave 的部分会谈到的。

再来说说 Multi-Master 的缺陷。**它最大的缺陷是关于事务处理的，本地事务（即单个存储节点）可以提交成功，但是全局事务（所有存储节点）却可能失败。**它包含这样两种典型的产生问题的场景：

由于是最终一致性，那么数据丢失也是可能发生的，即在写操作成功而节点间数据拷贝还没完成的时刻，如果主节点挂掉了，那么数据丢失也就发生了，只不过丢失的数据可能相对较少，但是全局事务的完整性就无从谈起了。

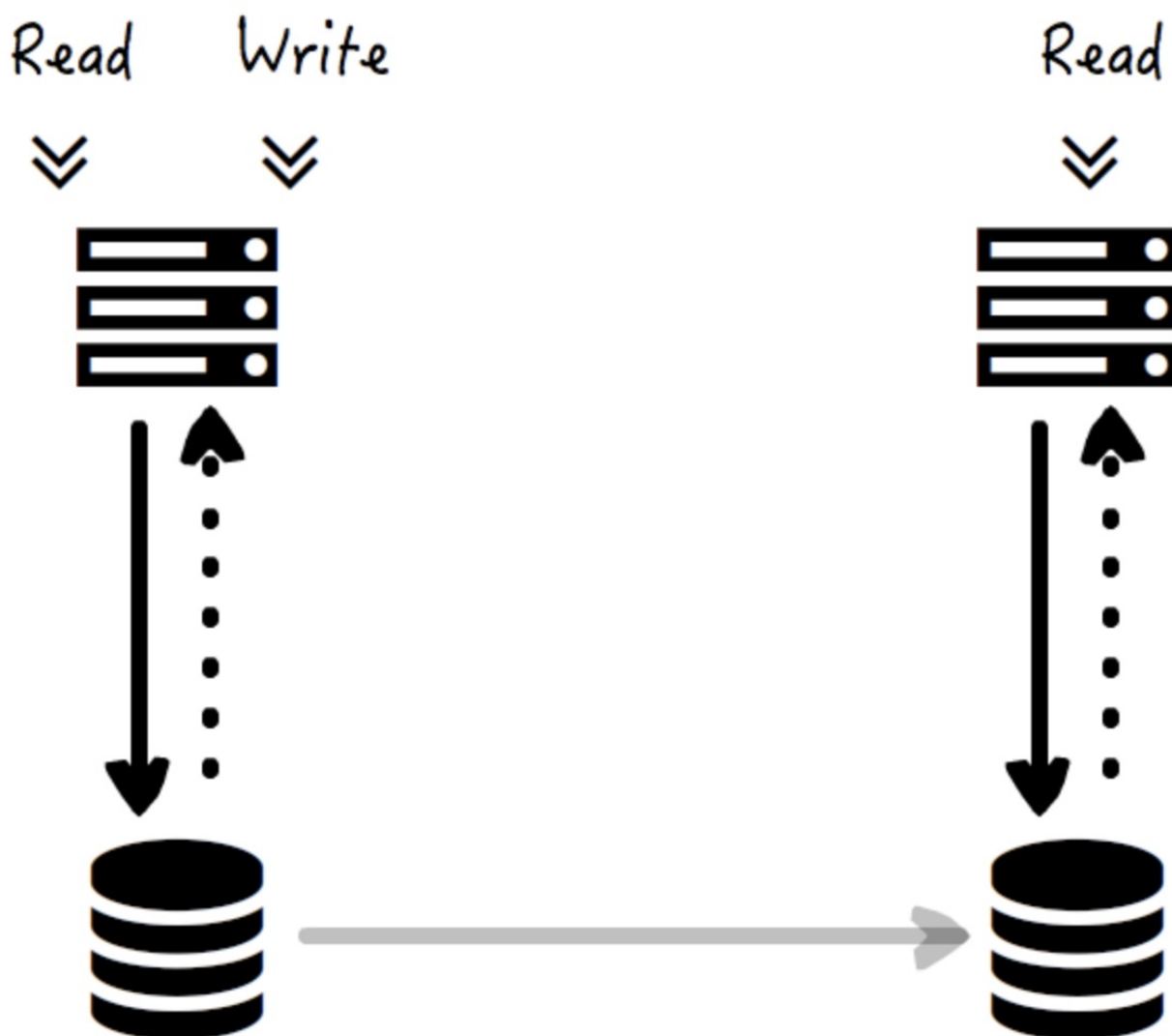
如果没有节点异常，主节点 A 的事务提交成功，主节点 B 的事务也提交成功，它们是做到了对本地数据库中事务操作的原子性。可是当进行节点间数据互拷贝时，一旦这两个提交的事务发生冲突（例如修改同一条记录），它们就傻眼了，到底应该以 A 还是以 B 的事务为准？这种冲突的解决会比较复杂，而且由于发生在异步的拷贝环节，这时候用户的请求都已经返回响应了，就没法告知用户事务冲突了。



因此，当我们要实现全局事务的时候，Multi-Master 往往不是一个好的选择。

### 3. Master-Slave

Master-Slave 架构是指存在一个可读可写（或者只写）的 Master 节点，而存在多个只读的 Slave 节点，每当有通过 Master 的更新出现，数据会以异步的方式单向拷贝到所有的 Slave 节点上去。



这种方式和 Multi-Master 比起来，将可写的节点数减少为了一个，而允许有多个只读的节点（图中只画了一个，但实际可以有多个），这种方式比较适用互联网较常见的业务，即读远大于写的场景，而且读的可扩展性（Scalability）较强（即增加一个 Slave 节点的代价较小），而且不存在 Multi-Master 的事务冲突问题。

当然了，**Master-Slave 的缺点也很明显**。既然**只有一个可写的节点**，那么写的可扩展性就很差了；而且和 Multi-Master 一样，数据从 Master 到 Slave 的拷贝是异步进行的，因此数据存在丢失的可能。

和 Multi-Master 一样，我们当然也可以让数据拷贝变成同步进行的，但是这又存在着上文讨论过的同样的缺陷。但是，有一种介于全同步和全异步之间的缓解这个问题的方法，即“最小副本数量”，比如可能存在 5 个 Slave 节点，但是从 Master 到 Slave 的数据拷贝一旦在 2 个节点成功了，就不用等另外 3 个返回，直接返回用户操作成功。即便那 3 个中存在失败，系统也可以标记失败节点，并按照既定策略自动处理掉，而不影响用户感知。

因此，我们可以说这种数据拷贝的方法是“部分同步”“部分异步”的，既降低了数据丢失的可能，又避免了因为某个 Slave 问题而导致 Master “死等”的情况发生。

最后，值得注意的是，写现在变成单点的了，为了避免单点故障引起的服务中断，一种方式是在 Master 挂掉的时候，Slave 可以挺身而出，变为 Master 顶上去提供写的服务。但是这件事情说说容易，实际要让它自动发生却有大量的工作要做，比如，谁顶上去，以及顶上去之后，原来以为挂掉的 Master 又活过来了怎么办，等等。

## 4. 其它

还有其它更为复杂的方法，一种是 2PC 或 3PC，即两阶段提交或三阶段提交，甚至采用高容错的分布式的共识算法 Paxos。这些方法能够保证强一致性，但是在实现上都要复杂许多，我在今天的扩展阅读中会介绍它们。

下面这张比较的表格来自 [☞ Transactions Across Datacenters](#) 这个著名的演讲，这张图在互联网上流传很广。



	Backups	M/S	MM	2PC	Paxos
Consistency	Weak	Eventual		Strong	
Transactions	No	Full	Local	Full	
Latency	Low			High	
Throughput	High			Low	Medium
Data loss	Lots	Some		None	
Failover	Down	Read only	Read/write		

简单说明一下，从上到下每行的含义依次为：一致性、事务支持、延迟、吞吐量、数据丢失和故障转移（指的是节点出现故障以后，其它节点可以自动顶替上来的能力）。

从中我们可以看到，没有一列能够做到全绿色，这正如我们所知道的那样，软件工程上的问题都“没有银弹”。特别是，Backups、M/S 和 MM 得益于异步的副本拷贝，能够做到低延迟，这就无法做到强一致性；而 2PC 和 Paxos 通过同步操作可以做到强一致性，却带来了高延迟。

## 总结思考

今天我们学习和理解了数据持久化中一致性的相关概念和实现技术，希望通过今天的学习，你能做到“知其然，知其所以然”。

现在，我来提一个问题，检验一下你的学习成果。请将下列存储系统按照“强一致性”“弱一致性”和“最终一致性”进行归类：

关系数据库

本地文件

浏览器缓存

网盘数据

CDN 节点上的静态资源

好，今天的正文内容就到这里。如果你对一致性哈希原理了解不够透彻的话，我强烈推荐你继续学习今天的选修课堂。

## 选修课堂：一致性哈希

在今天的选修课堂中，我们来学习一种特殊的哈希算法——一致性哈希（Consistent Hashing）。

首先，你可以在脑海里回忆一下，什么是哈希算法。哈希算法，又被称为散列算法，就是通过某种确定的键值函数，将源数据映射成为一个简短的新数据串，这个串叫做哈希值。**如果两个源数据的 hash 值不同，那么它们一定不相同；如果两个源数据的 hash 值相同，那么这两个源数据可能相同，也可能不相同。**

我们在 [🔗\[第 02 讲\]](#) 中谈到的数字签名，就是通过一个哈希算法，得到证书的哈希值，也就是它的“指纹”，是经过加密以后得到的。哈希是很常用的算法和技术，在后面的全栈内容中我们还会遇到。

我们有时候会使用一个特殊的哈希算法，来将每项数据都映射到某一个“位置”，从而将大量的数据分散存储到不同的位置中。哈希算法在数据量大，且单个节点（单台机器）无法处理的时候尤为有用。比如说，我们要将从 0 到 9999 这 10000 个连续自然数分散到 5 个数据存储的节点上，那我就可以设计一个基于取余数的哈希算法，做到均匀分布：

```
1 f(x) = x % 5
```


 复制代码

我们可以看到，这个函数的结果，也就是哈希值，只有 0、1、2、3、4 这 5 个，对应这 5 个节点，那么我可以根据其结果把这个 x 放到相应的节点上去。这样，每个节点就只需要存储 2000 个数。

好，这看起来是个挺好的解决办法，但是现在问题来了，由于业务的扩张，我们现在需要处理从 0 到 11999 这 12000 个数了，也就是说，多了 2000 个数。可是，节点承载的数据量已经基本到达了极限，没法再加入那么多数据了。

没问题，我们加机器吧，现在有 6 个节点了，我们就得修改这个算法：

```
1 f(x) = x % 6
```

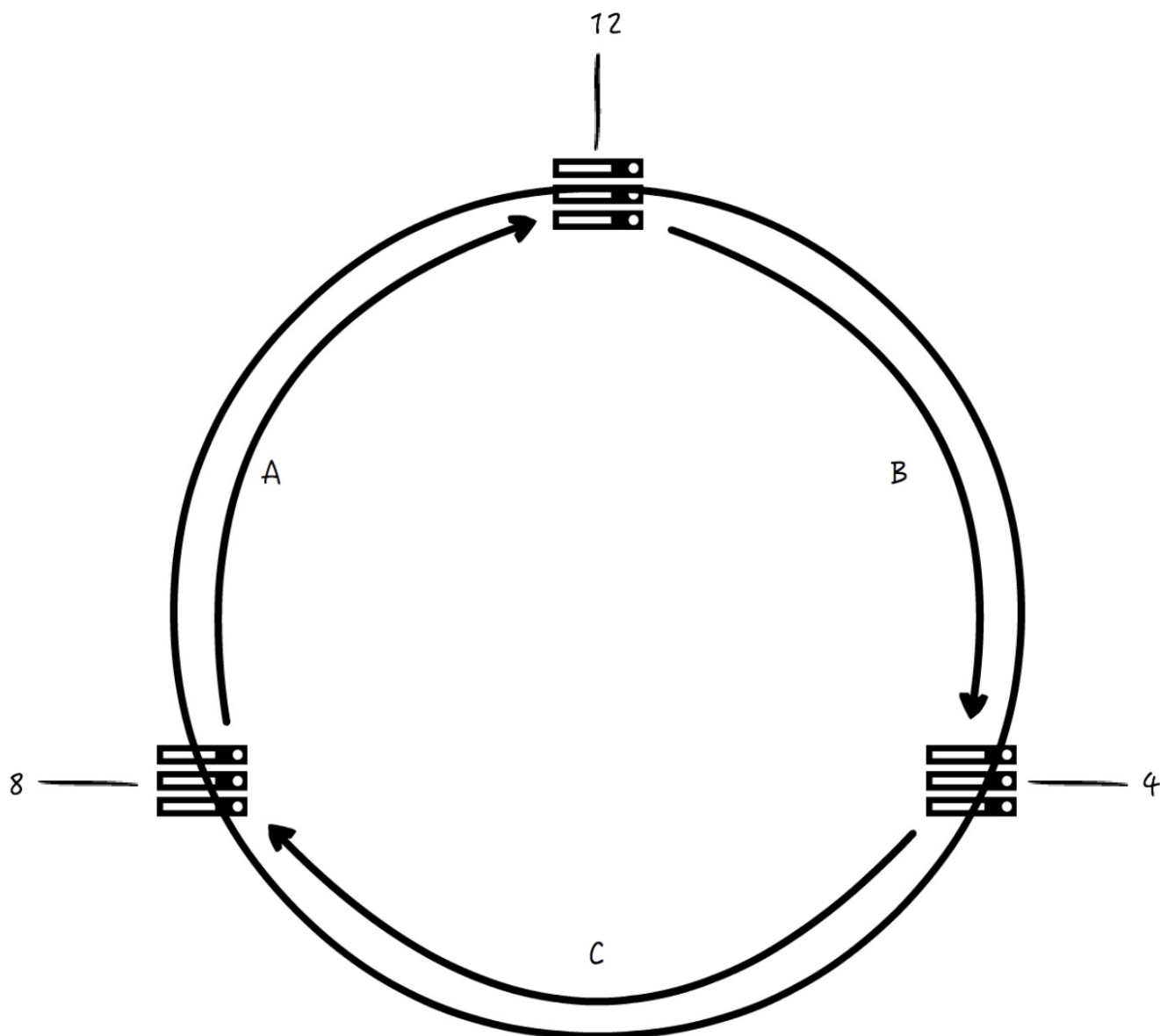
 复制代码

嗯，这样数据还是能均匀分布。

原理上没错，可是这又带来了一个问题，就是这些已经在节点上的数据，必须要调整位置了，毕竟算法变了嘛，因此这些数所在的节点可能要改变，这个过程叫做 **Rehashing**。这一调整，就傻眼了，只有同为 5 和 6 的倍数的数（即只有为 30 倍数的数），不用调整位置，其它全部都要调整。也就是说，就因为加了这一台机器， $29/30 = 96.7\%$  的数据全部都要调整位置！

这个代价显然是接受不了的，那有没有办法可以优化它呢？

当然！**一致性哈希**，就是一种尽可能减少 **Rehashing** 过程中进行数据迁移的算法。且看下面这张图：



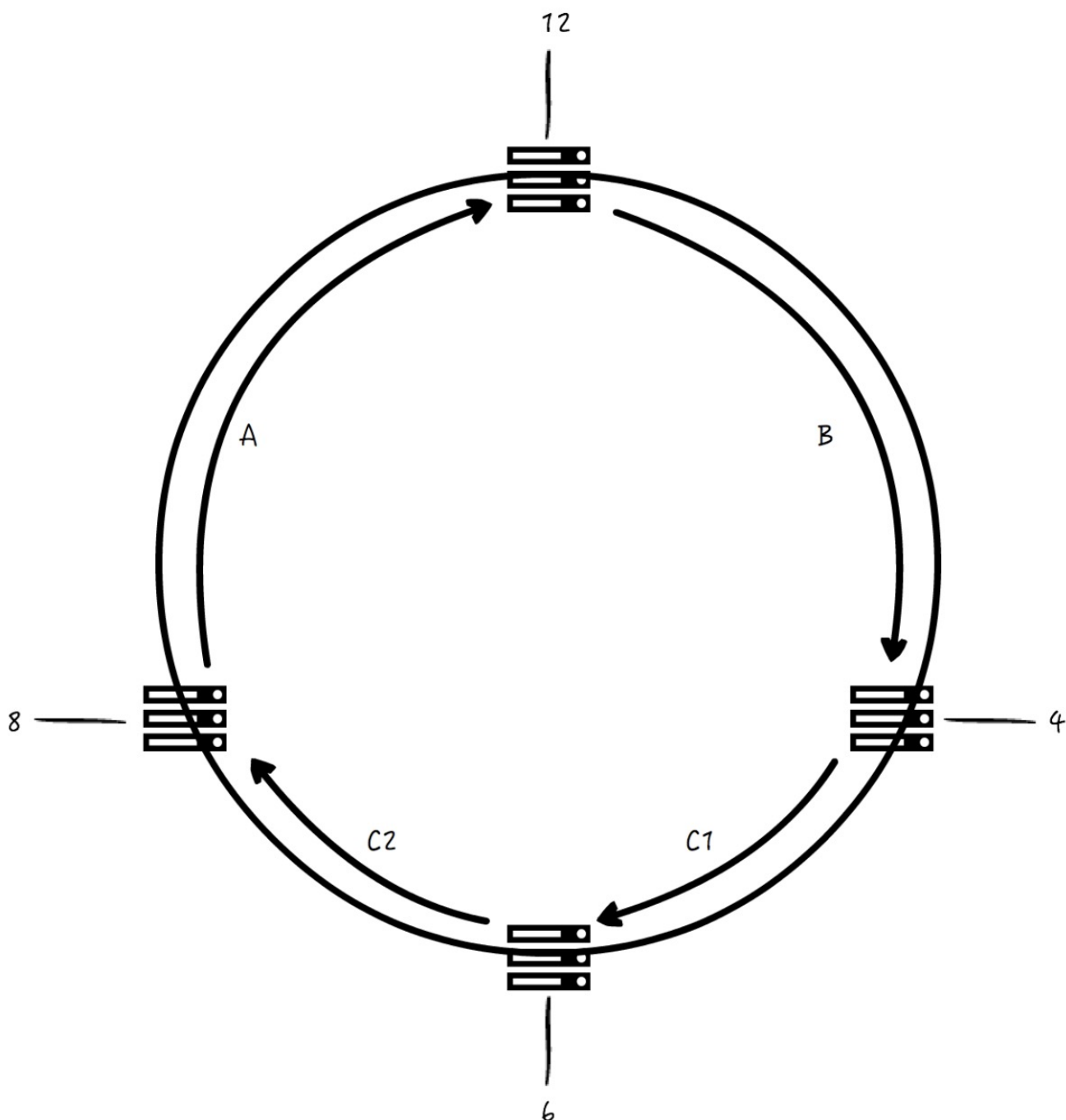
请你把上面的圆盘想象成一个时钟，总共有 12 格（0 点到 12 点），假如说我们通过上面类似的哈希算法，把数据映射到时钟的每个格子上。因为是时钟，我们这次取 12 的余数：

```
1 f(x) = x % 12
```

[复制代码](#)

同时，系统中总共有三台服务器，那么每台服务器就可以负责管理其中的“4 个小时”的数据。比如哈希值是 1~4 的数据（范围 B）存储在右下角的节点，5~8 的数据（范围 C）存储在左下角的节点，而 9~12 的数据（范围 A）存储到正上方的节点。

用这种方式来打散数据看起来似乎没有什么特别的对不对，别急，当我们添加新硬件，有一个新节点加入的时候，情况就不同了，请看下图：



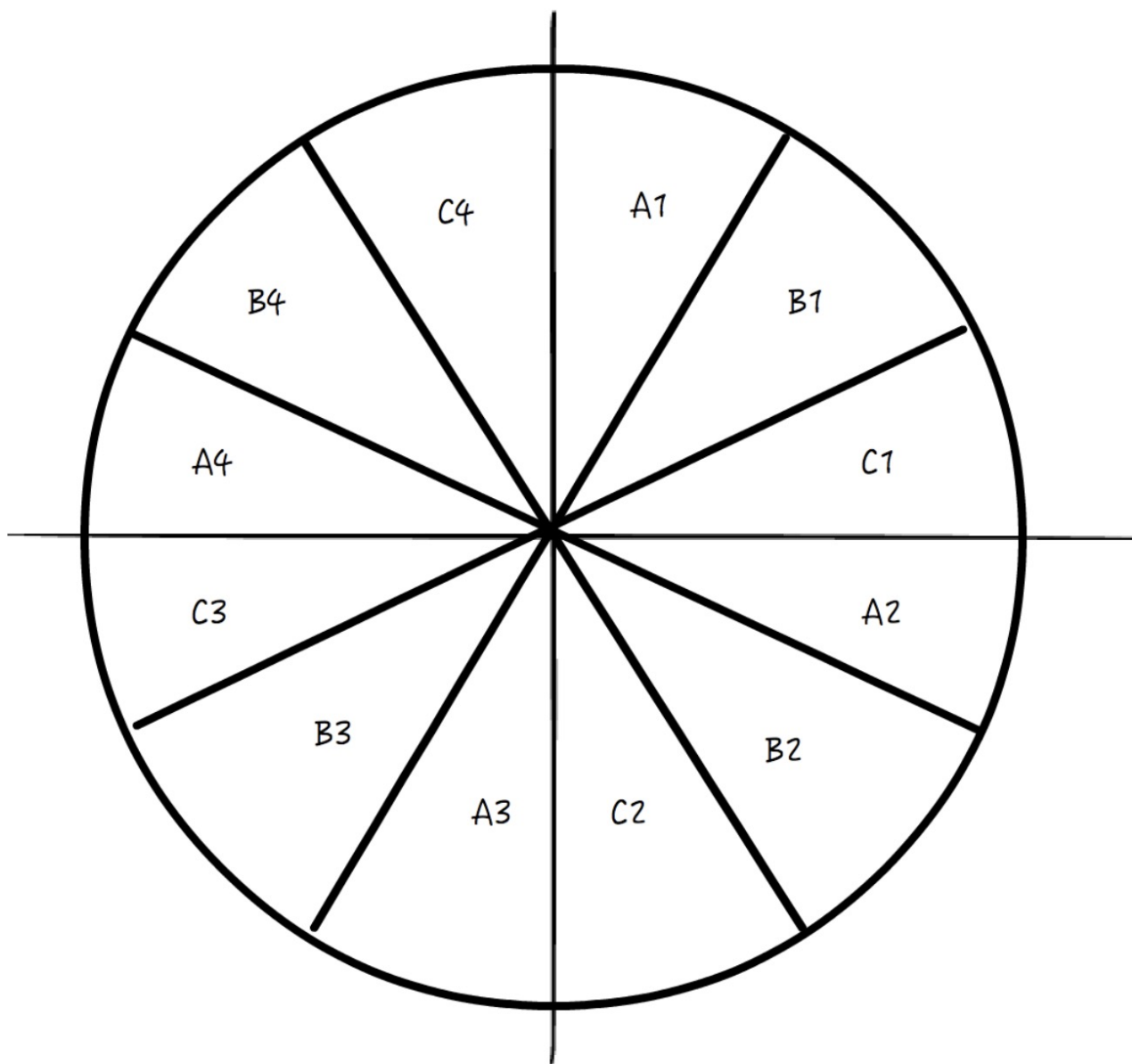
在这种情况下，正下方有一台机器被加入，原本 5~8 点的数据被分成两部分，7~8 点的数据（C2）依然存储在左下角的原节点，而 5~6 点的数据（C1）则需要迁移到新的，也就是正下方的节点上。

你看，这种情况下，添加一个节点，只需要移动其中的一部分数据，也就是  $2/12 = 1/6$  的数据就行，是不是对整个系统影响就小了很多？

等等！你可能会说，这样添加了一台服务器，如果原始数据哈希计算后的分布是均匀的，**经过了添加机器的操作，节点上承载数据分布却是不均匀的**——正上方、右下角的服务器分别承载了总共  $1/3$  的数据，而左下角、正下方的服务器却各自只需要承载  $1/6$  的数据。

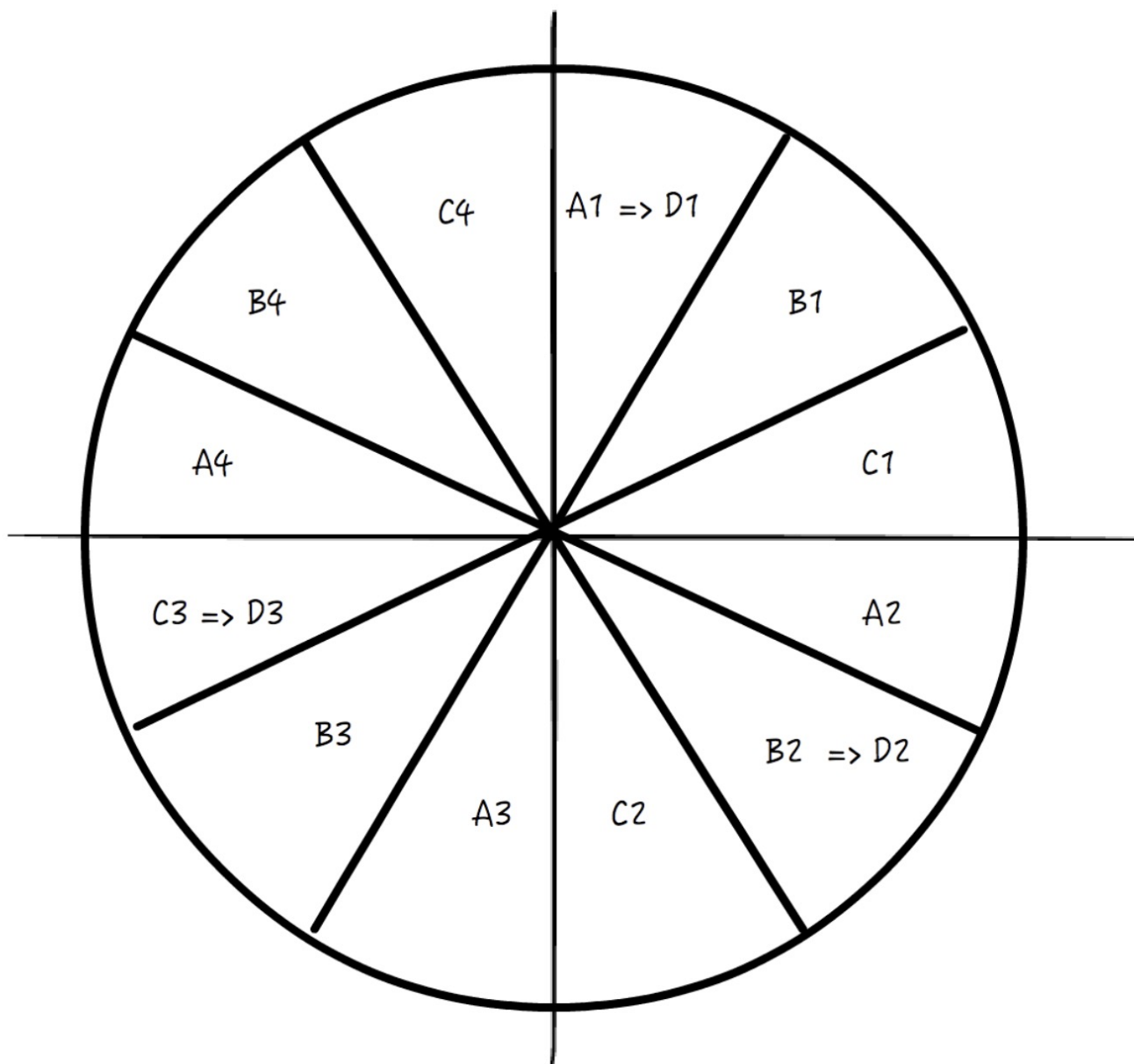
那么，这个问题，怎么解决？如果你能想到这个问题，那非常好。

解决方法就是引入“**虚拟节点**”，我们根据时钟的 12 个数字，把它均匀分成 12 个区域，分别由 12 个虚拟节点负责，并且顺时针按照 Ax-Bx-Cx 这样命名。这样，在添加机器以前，每台机器需要负责 4 块数据（例如某台机器 A 需要承载 A1、A2、A3 和 A4 的数据），并且它们均匀地散布在圆环上：



好，现在添加新机器，我们只需要把 A1、B2、C3 这三个虚拟节点的数据，搬迁到新机器 D 上：





你看，同样搬迁了最少量的数据，且元盘上的数据还是均匀分布的，只是从均匀分布在 3 台机器，变成了均匀分布到 4 台机器上。当然，作为示意，我这里是把圆环分成了 12 份，实际可以分成更多的  $2^n$  份。

## 扩展阅读

关于 2PC 和 3PC，如果你感兴趣的话，可以阅读维基百科的词条，[2PC](#) 和 [3PC](#)，或者是直接阅读 [The Two-Phase Commit Protocol](#) 和 [Three-Phase Commit Protocol](#)。

关于 Paxos，算法本身比较难，如果你很感兴趣，我找了一些中文材料，我觉得 [Paxos 算法详解](#) 这篇是相对讲得比较清楚的。

文中那个表格最早是来自于 Google I/O 2009 的 [Transactions Across Datacenters](#) 这个分享，后来有人 [上传到了 Bilibili 上](#)，我推荐你听一下这个分享。



# 全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 22 | 赫赫有名的双刃剑：缓存（下）

下一篇 24 | 尺有所短，寸有所长：CAP和数据存储技术选择

## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。