## 前言

本书起始于 2018 年 9 月 25 日，只关注的是分布式领域的一致性问题，因为少即是多。

本书预计完成 15 小节，书中内容绝大部分来源于原创，部分内容取自互联网并进行了修改。

好的架构不是设计出来的，而是修改出来的。同样道理，好的内容不是写完之后束之高阁，而是需要不断地修订。所以，本书的内容会不断地修订，追求观点新颖，力图简单但不肤浅。诚然，由于个人的能力总有不全之处，我也欢迎更多的人提出建议并给以指正。

**本书内容仅限内部使用，不得外传。**

## 第一节：理解数据副本

数据副本是个很常见的概念和术语，我们经常遇到，但是也最容易让人产生迷惑。

副有以下几种含义：
（1）辅助的，区别于正和主，例如：副职，副手，副官。
（2）附带的，次要的，例如：副业，副品，副食，副刊。
（3）相配、相称之意，例如：名实相副，其实难副。
（4）量词，表示一组或一套，例如：一副手套，全副武装；也可指态度，例如，一副笑脸。

在分布式系统和大数据领域，所有的节点，所有的数据块，地位都是平等，所以副本并不是指正副的“副“，只是用作量词，没有主副之分。可以这么说，所有的数据都可以叫做“数据副本”。

## 第二节：一致性问题的起源

一致性问题立足于数据副本，有数据副本的地方必定会有一致性问题。对于数据副本，人们往往印象不深，但是对于缓存，我想人人皆知。缓存就是数据副本。

我们知道，数据存在于内存中，然后被加载到寄存器中，此时出现了两个数据副本，如何保证两者的一致性呢？这就是一致性问题的起源。

一致性问题并不是分布式领域才有的问题，只不过在分布式领域更常见罢了。因为存在多个数据副本，如果对第一个数据副本进行了更新，而对其他的数据副本没有更新操作，那么这会导致数据不一致性。

人们发现了一致性问题之后，随后提出了 CAP 理论，而一致性位列其中的 C 位。一致性是

个非常重要的概念，它前承数据副本 ，后接 CAP 理论。

副本、一致性这些基本的概念很重要。能够不假思索的说出来，很考验了一个的功底。搞分布式的，搞大数据的，如果这些都不甚明白，那真的还算没有入门。

**第三节：缓存与数据库的一致性问题：先更新缓存还是先更新数据库**

在当今软件开发界，缓存基本上成了项目的标配。本地缓存也好、内存数据库缓存也好，只要用了缓存就会涉及数据一致性的问题，这个一致性指的缓存中的数据和数据库中数据的一致性，如何保证缓存与数据库的数据一致呢，很多人会说在对缓存数据修改时要同步修改两处地方(缓存、数据库)不就可以了吗？但这样做真能达到同步效果吗？不可能，不管如何实现，它都是有先后顺序的，到底是先更新缓存呢还是先更新数据库呢，这个问题困扰了很多的人。

先更新缓存，这样可以保证缓存数据都是最新的，但这样做有一定风险，就是缓存更新完成，再更新数据库时发现异常，导致回滚，数据库没更新，这就出现了数据不一致的情况。有人就说，回滚时把缓存清空不就可以吗？是可以，但在这个时间间隔内就产生了数据不一致的情况，如果恰巧有用户访问了缓存，就会读取到脏数据。

那么，我们先更新数据库后更新缓存不就行了吗？这种方式是不是就解决了不一致的问题呢？答案是没有。因为你在提交数据库到成功更新缓存的这个时间段还是存在短暂的不一致性。如果你的业务对数据一致性要求不高，那没问题，可以使用这种方案。如果业务对一致性要求极高，那就需要进一步优化为：数据库事务提交前清空缓存中的对应数据，事务提交成功后再更新缓存数据库为新数据。这一时刻如果请求读取缓存发现没有数据，于是进行数据库查询，而数据库此时有事务更新被上锁（可以使用 select...for update），所以新请求会阻塞等到事务执行成功，然后就可以读取到新数据了，读取新数据后放入缓存中，这样缓存和数据库也就一致了，这样也就保证了数据一致性，但是牺牲了可用性来换取的。

**第三节：微观一致性问题：内存可见性和寄存器可见性**

数据在哪里存在呢？只在内存吗？不是的。数据在内存中存在，但是当用的时候会加载到 CPU 的寄存器里面。内存和寄存器是两个地方，从而出现了新的名词：内存可见性和寄存器可见性。

为什么叫内存可见性呢？感觉很奇怪的名字。其实，明白以下道理就不奇怪了。数据的流动过程是：内存->寄存器->计算器

很多时候，数据从内存地址读取到寄存器里面，后面的计算过程中 CPU 就一直使用寄存器里面的值，即是内存地址上的值发生变化，CPU 也不知道，CPU 此时就是井底之蛙，而变量此时可以称为：寄存器可见性。

但是当变量被 volatile 修饰之后，CPU 就不再偷懒，只要用到数据，它都会越过寄存器，直接从内存中读取，然后再在计算器中计算，最后返回给内存，这个时候变量就不在寄存器里面停留，就当寄存器不存在一样，这就称为内存可见性。

**第四节：volatile 与一致性**

在 Java 中，synchronized 修饰的是代码块，代码块里面的变量都实现了内存可见性。内存可见性的底层是 CPU 的指令实现的。volatile 修饰的是变量，它的作用也是实现内存可见性，底层用的同一个 CPU 指令。可以这样理解：synchronized 里面的变量都是 volatile 修饰的。

我们都用过 synchronized，但是用 volatile 机会很少，其实换个角度想想，synchronized 里面的变量完全可以看做被 volatile 修饰，这样一想，是不是感觉 volatile 离我们很近很亲切，不陌生了。

这个提问来源于群里的成员：volatile 除了保证内存可见性，还有个作用是防止指令重排。

关于这一点，我想做一个说明和补充。

这个成员提出：JVM 会对 new 对象的过程进行指令重排，先分配空间，再把空间地址返回给变量，最后才进行对象实例化，加了 volatile 后会强制先进行实例化，最后才把对象地址返回被变量。

我觉得，指令重排是不是最终的思想来源还是内存可见性呢？如果两个互不相关的思想，用到一个事物上，感觉怪怪的。

我后来想了想，寄存器和主存的隔离造成了数据的不一致，volatile 的初衷是保证数据的强一致性，当赋值基本简单类型的时候，这种一致性很容易实现。但是赋值对象类型的时候，这种一致性分为强一致性和弱一致性，重排是弱一致性，而有序则是强一致性，volatile 的目的是强一致性，所以最终它要求指令不得重排。现在我感觉可以把可见性和有序性都统一到一致性上面了。

**第五节：CAP 定理**

## 1、什么是 CAP 定理

2000 年的时候，Eric Brewer 教授提出了 CAP 猜想，2 年后，被 Seth Gilbert 和 Nancy Lynch 从理论上证明了猜想的可能性，从此，CAP 理论正式在学术上成为了分布式计算领域的公认定理，并深深的影响了分布式计算的发展。

CAP 理论告诉我们，一个分布式系统不可能同时满足一致性（Consistency），可用性（Availability）和分区容错性（Partition tolerance）这三个基本需求，最多只能同时满足其中的 2 个。

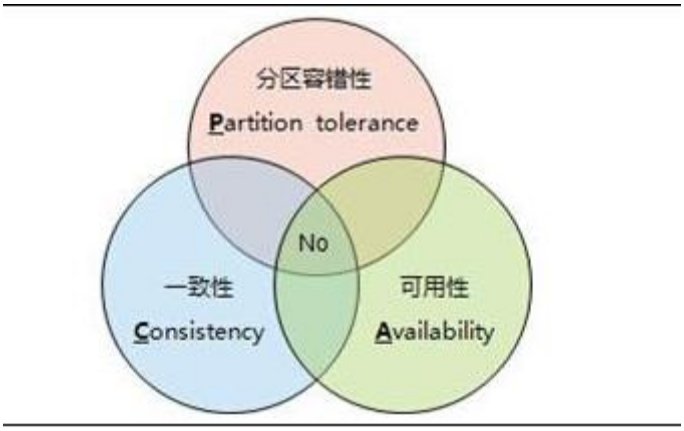
| 选项 | 描述 |
| --- | --- |
| C（Consistence） | **一致性**，指数据在多个副本之间能够保持一致的特性（严格的一致性）。 |
| A（Availability） | **可用性**，指系统提供的服务必须一直处于可用的状态，每次请求都能获取到正确的响应，但是不保证获取的数据为最新数据。 |
| P（Partition tolerance） | **分区容错性**，分布式系统在遇到任何网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务，除非整个网络环境都发生了故障。 |

## 2、什么是分区？什么是分区容错性？

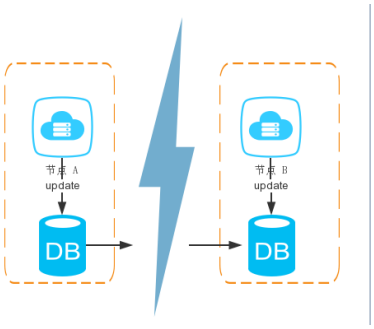
在分布式系统中，不同的节点分布在不同的子网络中，由于一些特殊的原因，这些子节点之间出现了网络不通的状态，但他们的内部子网络是正常的。从而导致了整个系统的环境被切分成了若干个孤立的区域，这就是分区。

分区容错性，说白了，就是是否允许出现分区。一旦出现分区，则各个分区之间的数据副本无法达到一致性。

### 3、为什么只能 3 选 2




假设有一个系统如下：




整个系统由两个节点配合组成，节点之间通过网络通信，当节点 A 进行更新数据库操作的时候，需要同时更新节点 B 的数据库（满足一致性）。这个系统怎么满足 CAP 呢?

C：当节点 A 更新的时候，节点 B 也要更新；
A：必须保证两个节点都是可用的；
P：当节点 A,B 出现了网络分区，必须保证对外可用。

可见，三者根本无法同时满足，只要出现了网络分区，一致性就无法满足，因为节点 A 根本连接不上节点 B。如果强行满足一致性，就必须停止服务运行，从而放弃可用性 。所以，最多满足两个条件，并有下面几种情况：

| 组合 | 分析结果 |
| --- | --- |
| CA | 满足一致性和可用，放弃分区容错。此时已经变成了单机系统，谈不上是分布式系统了。对于分布式系统而言，分区容错性是一个最基本的要求，因为分布式系统中的组件必然需要部署到不通的节点，必然会出现子网络，在分布式系统中，网络问题是必定会出现的异常。因此分布式系统只能在 C（一致性）和 A（可用 |

| 组合 | 分析结果 |
| --- | --- |
| | 性）之间进行权衡。<br><br>放弃分区容错性，比较简单的方式就是把所有的数据都放在一个分布式节点上。那不就又成为了单机应用了吗? |
| CP | 满足一致性和分区容错，也就是说，要放弃可用。当系统被分区，为了保证一致性，必须放弃可用性，让服务停用。<br><br>放弃可用性，一旦出现网络故障，受到影响的服务需要再等待一定时间，因为系统处于不可用的状态。 |
| AP | 满足可用性和分区容错，当出现分区，同时为了保证可用性，必须让节点继续对外服务，这样必然导致失去一致性。<br><br>放弃一致性，这里所指的一致性是强一致性，但是确保最终一致性。是很多分布式系统的选择。 |

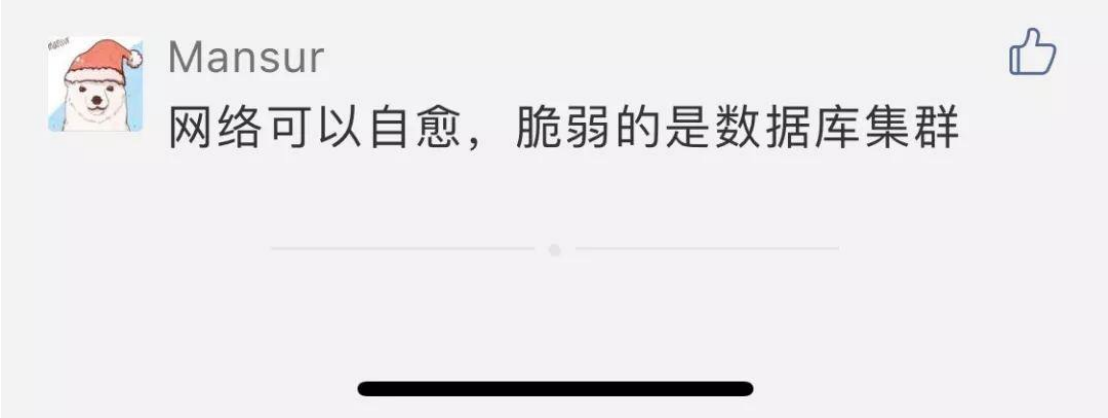
**4、能不能解决 3 选 2 的问题**

难道真的没有办法解决这个问题吗？CAP 理论已经提出了 13 年，也许可以做些改变。仔细想想，分区是百分之百出现的吗？如果不出现分区，那么就能够同时满足 CAP。如果出现了分区，可以根据策略进行调整。比如 C 不必使用那么强的一致性，可以先将数据存起来，稍后再更新，实现所谓的 “最终一致性”。这个思路又是一个庞大的问题，同时也引出了第二个理论 Base 理论，我们将在后面的文章中详细介绍。

**第六节：真实的 CP 场景：GitHub 事故：断网 43 秒，瘫痪 24 小时：**

2018 年 10 月 31 日，GitHub 技术负责人 Jason Warner 的一篇技术深度解析稿成为 IT 圈爆款。文中，Jason 坦诚地对外讲述了 10 月 21 日 100G 光缆设备故障后，Github 服务降级的应急过程以及反思总结。

从 Jason Warner 的文章中不难看出，造成断网 43 秒瘫痪 24 小时的罪魁祸首是数据库。由于部署在两个数据中心的数据库集群没有实时同步。意外发生时，Github 的工程师担心数据丢失，不敢快速将主数据库安全切换到东海岸的备份数据中心。




程序员们在 GitHub 这篇“忏悔录”下面留言，表达对数据库集群的“哀悼”。但更多 IT 从业者关心的问题是，如何避免这样的灾难事件降临到自己的公司，自己维护的系统。

分布式数据库专家认为，此次 Github 事件是典型的城市级故障。如果系统采用的是高可用的三地五中心解决方案，就可以自如应对。

原来，Github 类似银行采用的传统数据库两地三中心模式，即“主库（主机房）+同城热备库（同城热备机房）+异地灾备库（异地灾备机房）”。这种方式下通常只有主机房的服务器能提供写服务。如果主城市出现城市级故障，灾备城市的数据库虽然可以工作，但由于没有同步的最新数据，因此灾备库的数据是有损的。Github 表示，为了保证数据完整性，他们不得不牺牲恢复时间。

其实，这个问题采用三地五中心方案可以更好的应对。城市故障时，只要活着的两个城市的三个机房两两之间能够通信，就可以正常服务，也不会有任何的数据损失。

**第七节：BASE 理论**

**1、CAP 理论**

CAP 理论，指的是在一个分布式系统中，不可能同时满足 Consistency（一致性）、Availability（可用性）、Partition tolerance（分区容错性）这三个基本需求，最多只能满足其中的两项。

（1）Consistency（一致性）：所有节点在同一时间具有相同的数据。
（2）Availability（可用性）：保证每个请求不管成功或者失败都有响应。
（3）Partition Tolerance（分区容错性）：系统中任意信息的丢失或失败不会影响系统的继续运作。

对于分布式系统而言，分区容错性是一个最基本的要求，因为分布式系统中的组件必然需要部署到不通的节点，必然会出现子网络，在分布式系统中，网络问题是必定会出现的异常。因此分布式系统只能在 C（一致性）和 A（可用性）之间进行权衡。

### 2、BASE 理论

BASE 理论面向的是大型高可用可扩展的分布式系统，通过牺牲强一致性来获得可用性。

BASE 是 Basically Available（基本可用）、Soft state（软状态）和 Eventually consistent（最终一致性）三个短语的简写。

base 是对 cap 中一致性和可用性的权衡的结果。是根据 cao 理论演变而来，核心思想是即使无法做到强一致性，但是每个应用根据自身的业务特点，采用适当的方式来使系统达到最终与执行。

（1）基本可用
基本可用指的是分布式系统出现了不可预知故障的时候，允许损失部分可用性。响应时间合理延长，功能上适当做服务降级。
（2）弱状态
弱状态指的是允许系统中的数据存在中间状态，并认为该中间状态不会永祥系统的整体可用性，即允许在各个节点数据同步时存在延时。
（3）最终一致性
最终一致性强调的是系统中所有的数据副本，在经过一点时间 的同步之后，最终能够达到一个一致的状态。因此，最终一致性的本质是需要系统保证数据最终能够达到一致。而不需要实时保证系统数据的一致性。

## 第八节：2PC

为了使系统尽量能够达到 CAP，于是有了 BASE 协议，而 BASE 协议是在可用性和一致性之间做的取舍和妥协。人们往往需要在系统的可用性和数据一致性之间反复的权衡。为了解决分布式问题，涌现了很多经典的算法和协议，最著名的就是二阶段提交协议，简称 2PC，三阶段提交协议，Paxos 算法等。

### 1、什么是 2PC

在分布式系统中，会有多个机器节点，因此需要一个“协调者”，而各个节点就是“参与者”，协调者统一调度所有分布式节点的执行逻辑，这些被调度的分布式节点就是“参与者”。协调者最终决定这些参与者是否要把事务真正进行提交。正式基于这个思想，有了二

阶段提交和三阶段提交。

2PC 是 Two-Phase Commit，可以认为是一种算法，也可以认为是一种协议，主要目的就是为了保证分布式系统数据的一致性。

顾名思义，二阶段提交就是讲事务的提交过程分成了两个阶段来进行处理。流程如下：

**2、2PC 阶段一**

（1）事务询问
协调者向所有的参与者询问，是否准备好了执行事务，并开始等待各参与者的响应。

（2）执行事务
各参与者节点执行事务操作，并将 Undo 和 Redo 信息记入事务日志中。

（3）各参与者向协调者反馈事务询问的响应
如果参与者成功执行了事务操作，那么就反馈给协调者 Yes 响应，表示事务可以执行；如果参与者没有成功执行事务，就返回 No 给协调者，表示事务不可以执行。

从上面的过程可以看出，这个阶段是“投票阶段”。所有的节点都投票决定是否执行事务操作。

**3、2PC 阶段二**

在阶段二中，会根据阶段一的投票结果执行 2 种操作：执行事务提交或者中断事务。

**执行事务提交步骤如下：**

（1）发送提交请求：协调者向所有参与者发出 commit 请求。
（2）事务提交：参与者收到 commit 请求后，会正式执行事务提交操作，并在完成提交之后释放整个事务执行期间占用的事务资源。
（3）反馈事务提交结果：参与者在完成事务提交之后，向协调者发送 Ack 信息。
（4）协调者接收到所有参与者反馈的 Ack 信息后，完成事务。

**中断事务步骤如下：**

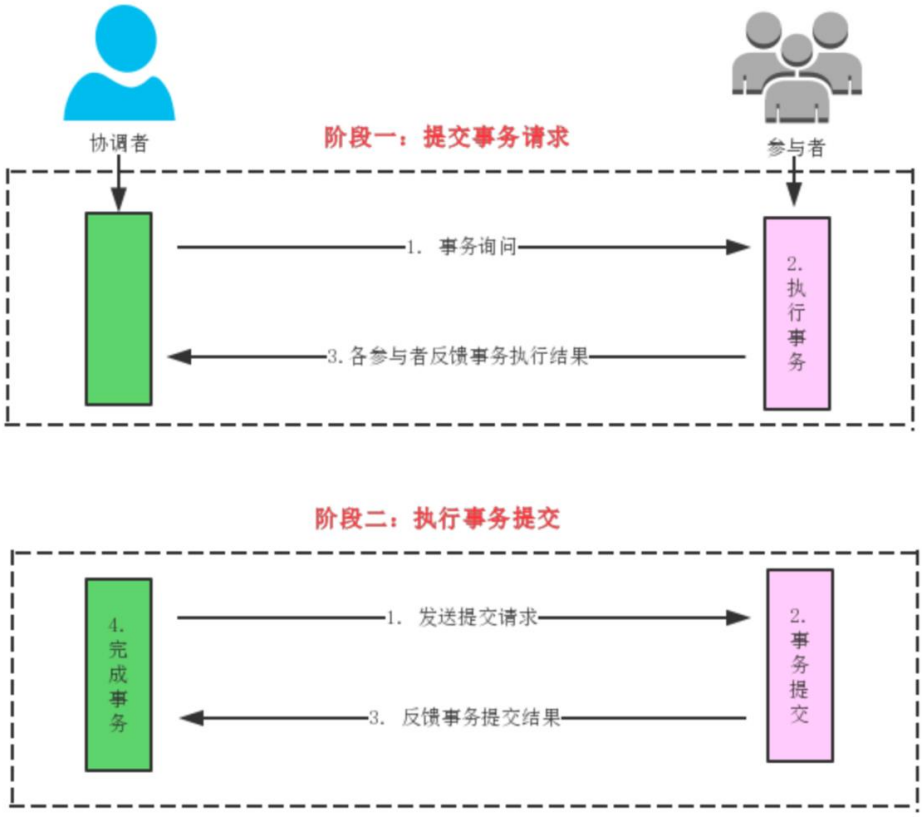
（1）发送回滚请求：协调者向所有参与者发出 Rollback 请求。
（2）事务回滚：参与者接收到 Rollback 请求后，会利用其在阶段一种记录的 Undo 信息

来执行事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源。

（3）反馈事务回滚结果：参与者在完成事务回滚之后，想协调者发送 Ack 信息。

（4）中断事务：协调者接收到所有参与者反馈的 Ack 信息后，完成事务中断。

总之，从上面的逻辑可以看出，二阶段提交就做了两个事情：投票，执行。2PC 核心是对每个事务都采用先尝试后提交的处理方式，因此也可以将二阶段提交看成一个强一致性的算法。整个事务的执行过程，如下图所示：




**4、优点缺点**

优点：原理简单，实现方便。

缺点：同步阻塞，单点问题，数据不一致，过于保守

（1）同步阻塞：

在二阶段提交的过程中，所有的节点都在等待其他节点的响应，无法进行其他操作。这种同步阻塞极大的限制了分布式系统的性能。

（2）单点问题：

协调者在整个二阶段提交过程中很重要，如果协调者在提交阶段出现问题，那么整个流程将无法运转，更重要的是：其他参与者将会处于一直锁定事务资源的状态中，而无法继续完成事务操作。

（3）数据不一致：
假设当协调者向所有的参与者发送 commit 请求之后，发生了局部网络异常或者是协调者在尚未发送完所有 commit 请求之前自身发生了崩溃，导致最终只有部分参与者收到了 commit 请求。这将导致严重的数据不一致问题。

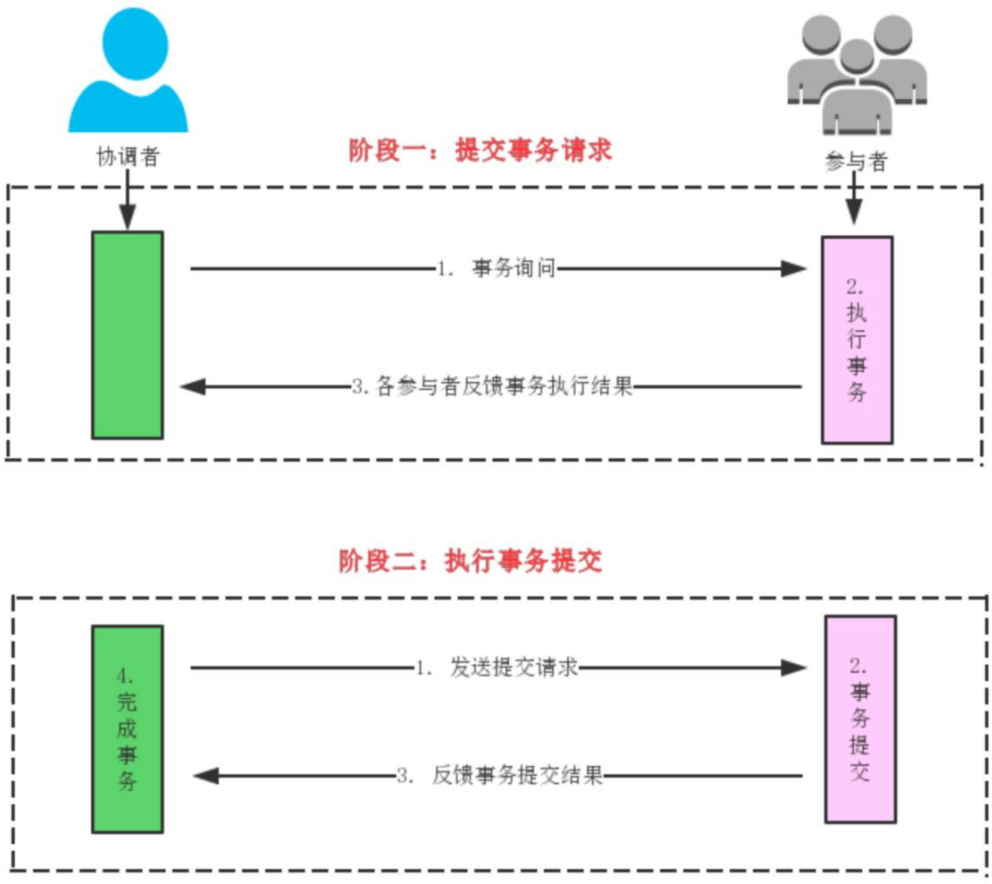
（4）过于保守：
如果在二阶段提交的提交询问阶段中，参与者出现故障而导致协调者始终无法获取到所有参与者的响应信息的化，这时协调者只能依靠其自身的超时机制来判断是否需要中断事务，显然，这种策略过于保守。换句话说，**二阶段提交协议没有设计较为完善的容错机制，任意一个节点是失败都会导致整个事务的失败。**

## 第九节：一致性协议之 3PC

为了实现 BASE 理论，需要在可用性和一致性之间找到一个合适的一致性理论，于是提出了 2PC 理论，也就是两阶段提交，二阶段提交原理简单，实现方便，但是缺点则是同步阻塞，单点问题，数据不一致，过于保守。而为了弥补二阶段提交的缺点，研究者们在它的基础上，提出了三阶段提交。

### 1、什么是三阶段提交

3PC，全称“Three Phase Commit”，是 2PC 的改进版，其将 2PC 的“提交事务请求”过程一分为二。回忆一下 2PC 的过程：



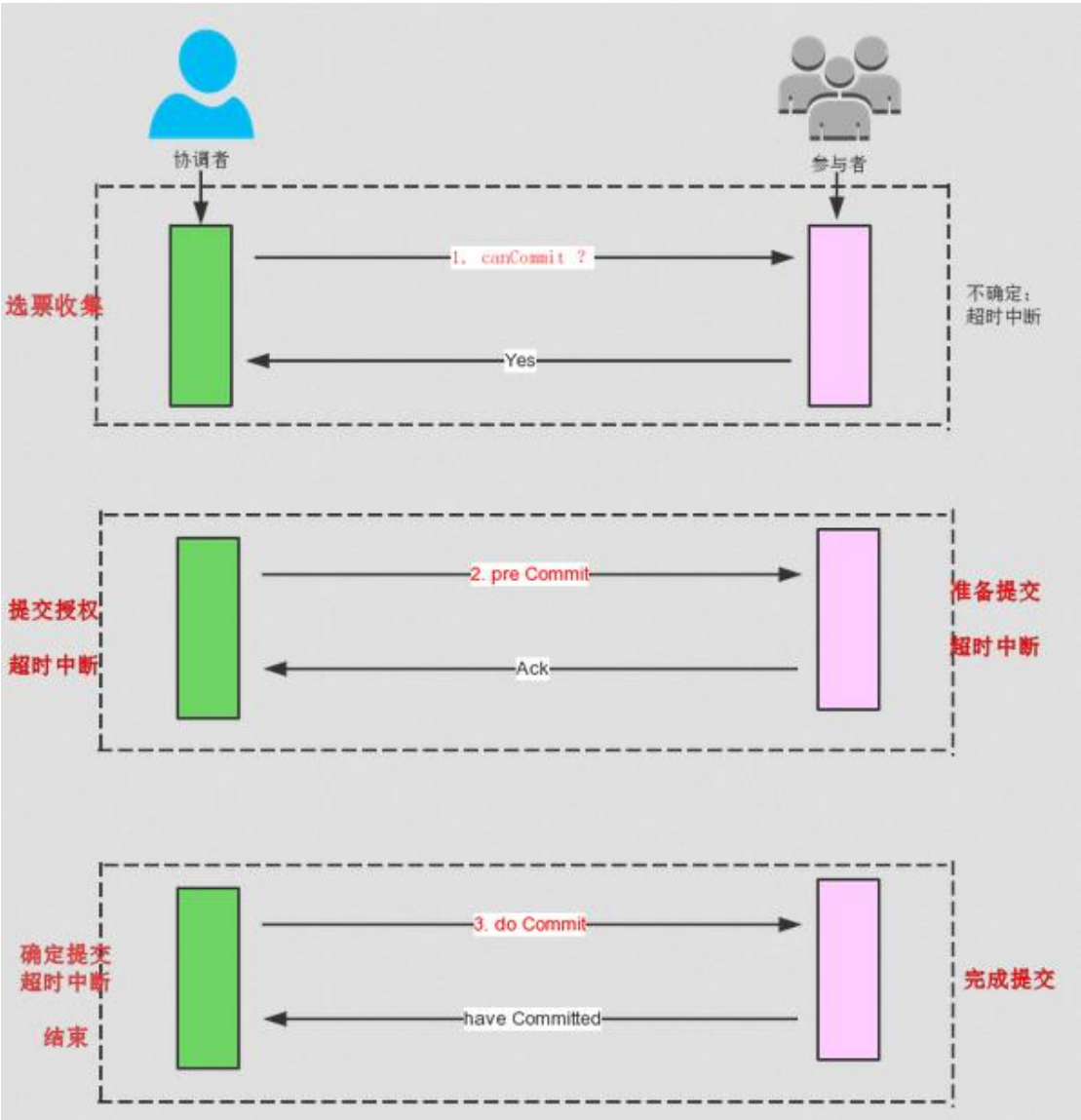

也就是说，3PC 将阶段一 "提交事务请求" 分成了 2 部分，总共形成了 3 个部分：

（1）CanCommit

（2）PreCommit

（3）doCommit

如下图所示：




## 2、阶段一：CanCommit

第一个阶段： CanCommit

事务询问：协调者向所有的参与者发送一个包含事务内容的 canCommit 请求，询问是否可以执行事务提交操作，并开始等待各参与者的响应。

各参与者向协调者反馈事务询问的响应：参与者接收来自协调者的 canCommit 请求，如果参与者认为自己可以顺利执行事务，就返回 Yes，否则反馈 No 响应。

## 3、阶段 二：PreCommit

协调者在得到所有参与者的响应之后，会根据结果执行两种操作：执行事务预提交或者中断事务。

**（1）执行事务预提交分为 3 个步骤：**

第一步：发送预提交请求：协调者向所有参与者节点发出 preCommit 的请求，并进入 prepared 状态。

第二步：事务预提交：参与者受到 preCommit 请求后，会执行事务操作，对应 2PC 中的“执行事务”，也会 Undo 和 Redo 信息记录到事务日志中。

第三步：各参与者向协调者反馈事务执行的结果：如果参与者成功执行了事务，就反馈 Ack 响应，同时等待指令：提交（commit） 或终止（abor）。

**（2）中断事务也分为 2 个步骤：**

第一步：发送中断请求：协调者向所有参与者节点发出 abort 请求 。

第二步：中断事务：参与者如果收到 abort 请求或者超时了，都会中断事务。

**4、阶段三：do Commit**

该阶段做真正的提交，同样也会出现两种情况：

**（1）执行提交**

发送提交请求：进入这一阶段，如果协调者正常工作，并且接收到了所有协调者的 Ack 响应，那么协调者将从 “预提交” 状态变为 “提交” 状态，并向所有的参与者发送 doCommit 请求 。

事务提交：参与者收到 doCommit 请求后，会正式执行事务提交操作，并在完成之后释放在整个事务执行期间占用的事务资源。

反馈事务提交结果：参与者完成事务提交后，向协调者发送 Ack 消息。

完成事务：协调者接收到所有参与者反馈的 Ack 消息后，完成事务。

**（2）中断事务**

假设有任何参与者反馈了 no 响应，或者超时了，就中断事务。

发送中断请求：协调者向所有的参与者节点发送 abort 请求。

事务回滚：参与者接收到 abort 请求后，会利用其在二阶段记录的 undo 信息来执行事务回滚操作，并在完成回滚之后释放整个事务执行期间占用的资源。

反馈事务回滚结果：参与者在完成事务回滚之后，想协调者发送 Ack 消息。

中断事务：协调者接收到所有的 Ack 消息后，中断事务。

注意：一旦进入阶段三，可能会出现 2 种故障：

（1）协调者出现问题
（2）协调者和参与者之间的网络故障

一旦出现了任一一种情况，最终都会导致参与者无法收到 doCommit 请求或者 abort 请求，针对这种情况，参与者都会在等待超时之后，继续进行事务提交。

**5、总结：**

优点：相比较 2PC，最大的优点是减少了参与者的阻塞范围（第一个阶段是不阻塞的），并且能够在单点故障后继续达成一致（2PC 在提交阶段会出现此问题，而 3PC 会根据协调者的状态进行回滚或者提交）。

缺点：如果参与者收到了 preCommit 消息后，出现了网络分区，那么参与者等待超时后，都会进行事务的提交，这必然会出现事务不一致的问题。

**第十节：Paxos 算法介绍**

Google Chubby 的作者 Mike Burrows 说过这个世界上只有一种一致性算法，那就是 Paxos，其它的算法都是残次品。

Paxos 算法问世已经有将近 30 年的历史了，是目前公认最有效的解决分布式场景下一致性问题的算法之一，但是缺点是比较难懂，工程化比较难。

为了实现集群的高可用性，用户的数据往往要多重备份，多个副本虽然避免了单点故障，但同时也引入了新的挑战。

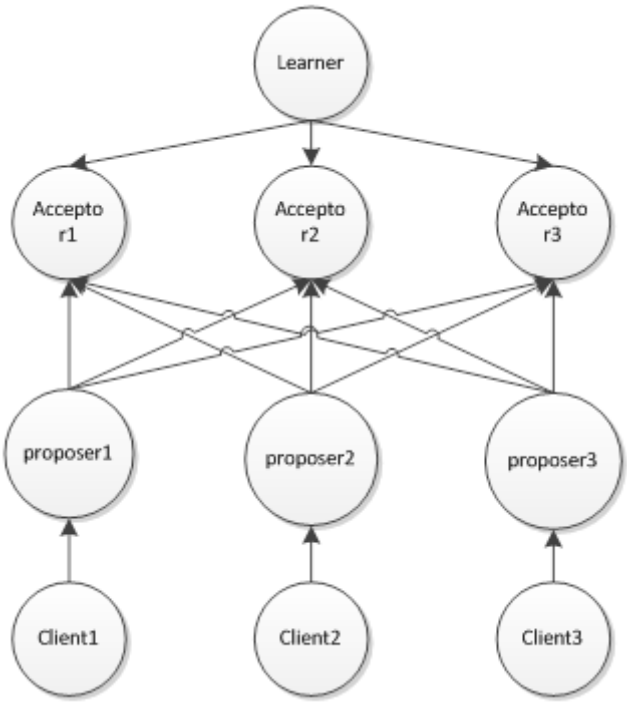
假设有一组服务器保存了用户的余额，初始是 100 块，现在用户提交了两个订单，一个订单是消费 10 元，一个订单是充值 50 元。由于网络错误和延迟等原因，导致一部分服务器只收到了第一个订单（余额更新为 90 元），一部分服务器只收到了第二个订单（余额更新为 150 元），还有一部分服务器两个订单都接收到了（余额更新为 140 元），这三者无法就最终余额达成一致。这就是一致性问题。

一致性算法并不保证所有提出的值都是正确的（这可能是安全管理员的职责）。我们假设所有提交的值都是正确的，算法需要对到底该选哪个做出决策，并使决策的结果被所有参与者获悉。

一致性算法并不保证所有节点中的数据是完全一致的，但它能保证即使一小部分节点出现故障，仍然能对外提供一致的服务（客户端无感知）。

在 Paxos 算法中，分为 4 种角色：
Proposer：提议者
Acceptor：决策者
Client：产生议题者
Learner：最终决策学习者

上面 4 种角色中，提议者和决策者是很重要的，其他的 2 个角色在整个算法中应该算做打酱油的，Proposer 就像 Client 的使者，由 Proposer 使者拿着 Client 的议题去向 Acceptor 提议，让 Acceptor 来决策。这里上面出现了个新名词：最终决策。现在来系统的介绍一下 Paxos 算法中所有的行为：
（1）Proposer 提出议题
（2）Acceptor 初步接受 或者 Acceptor 初步不接受
（3）如果上一步 Acceptor 初步接受则 Proposer 再次向 Acceptor 确认是否最终接受
（4）Acceptor 最终接受 或者 Acceptor 最终不接受
上面 Learner 最终学习的目标是 Acceptor 们最终接受了什么议题。注意，这里是向所有 Acceptor 学习，如果有多数派个 Acceptor 最终接受了某提议，那就得到了最终的结果，算法的目的就达到了。画一幅图来更加直观：




为什么需要 3 个 Acceptor？因为 Acceptor 必须是最少大于等于 3 个，并且必须是奇数个，因为要形成多数派嘛，如果是偶数个，比如 4 个，2 个接受 2 个不接受，各执己见，没法搞下去了。

为什么是 3 个 Proposer？ 其实无所谓是多少个了，1~n 都可以的。如果是 1 个 Proposer，毫无竞争压力，很顺利的完成 2 阶段提交，Acceptor 们最终批准了事。如果是多个 Proposer 就比较复杂了，请继续看。

上面的图中是画了很多节点的，每个节点需要一台机器么？答案是不需要的，上面的图是逻辑图，物理中，可以将 Acceptor 和 Proposer 以及 Client 放到一台机器上，只是使用了不同的端口号罢了，Acceptor 们启动不同端口的 TCP 监听，Proposer 来主动连接即可；完全可以将 Client、Proposer、Acceptor、Learner 合并到一个程序里面。

现在通过一则故事来学习 Paxos 的算法的流程(2 阶段提交)，有 2 个 Client(老板，老板之间是竞争关系)和 3 个 Acceptor(政府官员)：

（1）现在需要对一项议题来进行 Paxos 过程，议题是“A 项目我要中标！”，这里的“我”指每个带着他的秘书 Proposer 的 Client 老板。

（2）Proposer 当然听老板的话了，赶紧带着议题和现金去找 Acceptor 政府官员。

（3）作为政府官员，当然想谁给的钱多就把项目给谁。

（4）Proposer-1 小姐带着现金同时找到了 Acceptor-1~Acceptor-3 官员，1 与 2 号官员分别收取了 10 比特币，找到第 3 号官员时，没想到遭到了 3 号官员的鄙视，3 号官员告诉她，Proposer-2 给了 11 比特币。不过没关系，Proposer-1 已经得到了 1，2 两个官员的认可，形成了多数派(如果没有形成多数派，Proposer-1 会去银行提款在来找官员们给每人 20 比特币，这个过程一直重复每次+10 比特币，直到多数派的形成)，满意的找老板复命去了，但是此时 Proposer-2 保镖找到了 1，2 号官员，分别给了他们 11 比特币，1，2 号官员的态度立刻转变，都说 Proposer-2 的老板懂事，这下子 Proposer-2 放心了，搞定了 3 个官员，找老板复命去了，当然这个过程是第一阶段提交，只是官员们初步接受贿赂而已。故事中的比特币是编号，议题是 value。这个过程保证了在某一时刻，某一个 Proposer 的议题会形成一个多数派进行初步支持；

==========华丽的分割线，第一阶段结束==========

（5）现在进入第二阶段提交，现在 Proposer-1 小姐使用分身术(多线程并发)分了 3 个自己分别去找 3 位官员，最先找到了 1 号官员签合同，遭到了 1 号官员的鄙视，1 号官员告诉他 Proposer-2 先生给了他 11 比特币，因为上一条规则的性质 Proposer-1 小姐知道 Proposer-2 第一阶段在她之后又形成了多数派(至少有 2 位官员的赃款被更新了),此时她赶紧去提款准备重新贿赂这 3 个官员(重新进入第一阶段)，每人 20 比特币。刚给 1 号官员 20 比特币，1 号官员很高兴初步接受了议题，还没来得及见到 2，3 号官员的时候，
这时 Proposer-2 先生也使用分身术分别找 3 位官员(注意这里是 Proposer-2 的第二阶段)，被第 1 号官员拒绝了告诉他收到了 20 比特币，第 2,3 号官员顺利签了合同，这时 2，3 号官员记录 Client-2 老板用了 11 比特币中标，因为形成了多数派，所以最终接受了 Client2 老板中标这个议题，对于 Proposer-2 先生已经出色的完成了工作；

这时 Proposer-1 小姐找到了 2 号官员，官员告诉她合同已经签了，将合同给她看，Proposer-1 小姐是一个没有什么职业操守的聪明人，觉得跟 Client1 老板混没什么前途，所以将自己的议题修改为“Client2 老板中标”，并且给了 2 号官员 20 比特币，这样形成了一个多数派。顺利的再次进入第二阶段。由于此时没有人竞争了，顺利的找 3 位官员签合同，3 位官员看到议题与上次一次的合同是一致的，所以最终接受了，形成了多数派，Proposer-1 小姐跳槽到 Client2 老板的公司去了。

==========华丽的分割线，第二阶段结束==========

Paxos 过程结束了，这样，一致性得到了保证，算法运行到最后所有的 Proposer 都投“Client2 中标”所有的 Acceptor 都接受这个议题，也就是说在最初的第二阶段，议题是先入为主的，谁先占了先机，后面的 Proposer 在第一阶段就会学习到这个议题而修改自己本身的议题，因为这样没职业操守，才能让一致性得到保证，这就是 Paxos 算法的一个过程。原来 Paxos 算法里的角色都是这样的不靠谱，不过没关系，结果靠谱就可以了。该算法

就是为了追求结果的一致性。

**第十一节：数据一致性问题无关乎数据的对错**

为了实现集群的高可用性，用户的数据往往要多重备份，多个副本虽然避免了单点故障，但同时也引入了新的挑战。
假设有一组服务器保存了用户的余额，初始是 100 块，现在用户提交了两个订单，一个订单是消费 10 元，一个订单是充值 50 元。由于网络错误和延迟等原因，导致一部分服务器只收到了第一个订单（余额更新为 90 元），一部分服务器只收到了第二个订单（余额更新为 150 元），还有一部分服务器两个订单都接收到了（余额更新为 140 元），这三者无法就最终余额达成一致。这就是一致性问题。
一致性算法并不保证所有提出的值都是正确的（这可能是安全管理员的职责）。我们假设所有提交的值都是正确的，算法需要对到底该选哪个做出决策，并使决策的结果被所有参与者获悉。
一致性算法并不保证所有节点中的数据是完全一致的，但它能保证即使一小部分节点出现故障，仍然能对外提供一致的服务（客户端无感知）

**第十二节：主从复制架构的高可用方案**

我们知道，传统数据库产品最初都是单点架构，并不具备高可用设计，更多的是基于高端硬件产品满足“硬件可靠”的假设。

随着时间的推移，传统数据库产品先后推出了采用“主从复制”架构的高可用方案，其主要思路是：在原有的单数据库节点（主节点）之外再增加一个对等的数据库节点（从节点），通过数据库层面的复制技术（通常是日志复制）将主节点产生的数据实时复制到从节点。

正常情况下从节点不提供对外服务，当主节点发生故障时，在从节点上执行“切主”动作将从节点变成主节点，继续提供服务。在主从节点的部署方式上，除了本地单机房部署外，往往也支持同城灾备部署和异地灾备部署，因此也就具备了机房级容灾和城市级容灾的能力。很多新兴的数据库产品 (如 MySQL) 也是采用“主从复制”模式来实现高可用及容灾特性。

虽然数据库的主从复制技术，使得意外发生时数据库可以在一定时间内恢复服务，并且大部分数据不会丢失，具备了一定的高可用及容灾能力，但还是有一些无法解决的问题：

（1）通常情况下无法做到 RPO=0，即主节点发生故障或者灾难时有交易数据的损失。可能有的读者会说：你说错了！主从复制技术也能实现 RPO=0！是的，从理论上讲主从复制技术是可以利用强同步模式做到 RPO=0，但实际应用中，像银行核心系统这样的关键业务里却不会采用。为什么呢？因为这种模式将主节点和从节点以及主从节点之间的网络环境紧紧地绑在了一起，主节点的稳定性将不再由它自己决定，而要同时看从节点和网络环境的脸色：一旦从节点或者网络环境稍微抖动一下，主节点的性能就会受到直接影响。如果主节点和从节点之间是跨机房甚至跨城市部署，发生这种问题的概率会更大，影响也会变得更加显著。从某种程度上讲，和单节点模式相比，这种模式下主节点的稳定性不但没有增加，反而是降低了。如果有银行的朋友在关键业务中应用过主从模式，对强同步模式所带来的问题应该是深有体会的。

（2）RTO 相对较大，即系统恢复时间通常以十分钟甚至小时为计算单位，会给业务带来比较大的损失。造成这一情况的根本原因，是“主从复制”模式下从节点不具备自动切主的能力。由于“主从复制“模式中缺少第三方仲裁者的角色，当主从节点之间的心跳信号异常时，从节点无法靠自己判断到底是主点故障了，还是主从之间的网络故障了。此时，如果从节点认为是主节点故障而将自己自动切换成主节点，就极容易导致“双主”、“脑裂”（brain-split）的局面，对用户来说这是绝对无法接受的结果。所以数据库“主从复制”技术从来不会提供“从节点自动切换为主节点”的功能，一定要由“人”来确认主节点确实故障了，并手工发起从节点的切主动作，这就大大增加了系统恢复的时间（RTO）。

**第十三节：纵观原子操作发展史**

我们都知道，原子由原子核和绕核运动的电子组成。这是从物理状态而言的，原子可以分割的，但是在化学反应中原子是不可分割的。

原子是化学反应不可再分的最小微粒，原子是构成一般物质的最小单位，也称之为元素。

物质是由原子构成，这是客观存在的现象，在计算机领域，人的思想变得更抽象，更升华了一步：一个字节就是一个“原子”。因为当一个处理器读取一个字节时，其他处理器不能访问这个字节的内存地址，所以每次访问总是能获得一个完整的字节，不会出现半字节。

字节即是原子，操作字节的读写操作则为原子操作。后来，人们把不可被中断的一个操作或多个操作称之为原子操作。

除了读字节和写字节是原子操作之外，还有一个更复杂的读写操作也是原子操作。这就是大名鼎鼎的 CAS。CAS 是英文单词 CompareAndSwap 的缩写，中文意思是“比较并替换”。CAS 需要有 3 个操作数：内存地址 V，旧的预期值 A，即将要更新的目标值 B。

CAS 指令执行时，当且仅当内存地址 V 的值与预期值 A 相等时，将内存地址 V 的值修改为 B，否则就什么都不做。整个比较并替换的操作是一个原子操作。

很多人看到 CAS 就感觉这个是高深的玩意，其实不然。可以看做是读字节和写字节的进阶而已。