# CAP

Consistency、Availability和Partition-tolerance。

**强一致性**：等同于所有节点访问同一份最新的数据副本。系统在执行过某项操作后仍然处于一致的状态。在分布式系统中，更新操作执行成功后所有的用户都应该读取到最新的值，这样的系统被认为具有强一致性。

**可用性**：对数据更新具备高可用性。每一个操作总是能够在一定的时间内返回结果，这里需要注意的是“一定时间内”和“返回结果”。

**分区容错性**：能容忍网络分区。分区容错性可以理解为系统在存在网络分区的情况下仍然可以接受请求(满足一致性和可用性)。这里网络分区是指由于某种原因网络被分成若干个孤立的区域，而区域之间互不相通。还有一些人将分区容错性理解为系统对节点动态加入和离开的处理能力，因为节点的加入和离开可以认为是集群内部的网络分区。

如果Partition-tolerance的需求为0，只需要满足CA，可以延伸出ACID协议，比如RDBMS的事务一致性。然而对于互联网的分布式系统，鉴于高压力、大数据，那么必须对分区容忍性要求高，且高可用性，则采用BASE 弱一致性或者最终一致性。而对于很多互联网应用来说完全牺牲一致性是不可取的，否则数据是混乱的，那么系统可用性再高分布式再好也没有了价值。牺牲一致性，只是不再要求关系型数据库中的强一致性。从客户体验出发，最终一致性的关键是时间窗口，尽量让用户“用户感知到的一致性”。

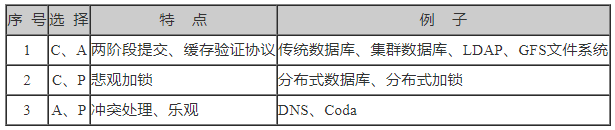
CAP理论断言任何基于网络的数据共享系统，最多只能满足数据一致性、可用性、分区容忍性三要素中的两个要素。但是通过显式处理分区情形，系统设计师可以做到优化数据一致性和可用性，进而取得三者之间的平衡。

CAP理论的表述很好地服务了它的目的，即**开阔设计师的思路，在多样化的取舍方案下设计出多样化的系统**。数据一致性和可用性的相对关系上有很多的争论，“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。**

虽然设计师仍然需要在分区的前提下对数据一致性和可用性做取舍，但具体如何处理分区和恢复一致性，这里面有不计其数的变通方案和灵活度。**当代CAP实践应将目标定为针对具体的应用，在合理范围内最大化数据一致性和可用性的“合力”。**这样的思路延伸为如何规划分区期间的操作和分区之后的恢复，从而启发设计师加深对CAP的认识，突破过去由于CAP理论的表述而产生的思维局限。

## CAP的组合

根据CAP理论，这三个条件最多只能实现两点，不能三者兼顾，因此在架构设计的时候必须进行取舍。然而三个要素的取舍不是绝对的0%，100%的关系。 而是某个度的衡量，譬如C(30%), P(80%), A(99%)等。而且大系统内套小系统，大小子系统对CAP三个参数的要求也不一样。



放弃P：如果想避免分区容错性问题的发生，一种做法是将所有的数据(与事务相关的)都放到一台机器上。虽然无法100%地保证系统不会出错，但不会碰到由分区带来的负面效果。当然，这个选择会严重影响系统的扩展性。

放弃A：相对于放弃“分区容错性”来说，其反面就是放弃可用性。一旦遇到分区容错故障，那么受到影响的服务需要等待数据一致，因此在等待期间系统就无法对外提供服务。

放弃C：这里所说的放弃一致性，并不是完全放弃数据的一致性，而是**放弃数据的强一致性**，而保留数据的最终一致性。以网络购物为例，对只剩最后一件库存的商品，如果同时接收到了两份订单，那么较晚的订单将被告知商品售罄。

除了这些组合，还有比如BASE，BASE支持最终一致性，其实是放弃C的一个特例。

**一致性与可用性**：在网络隔离的情况下，数据库将数据集中，就要接受数据丢失的风险。

**一致性与扩展性**：看得出即使读写一致性保证降低了副本集的扩展性，只有在原子写模型中才可以以一种相对可扩展的方式处理写冲突。原子读改写模型通过给数据加上临时性的全局锁来避免冲突。这表明，数据或操作之间的依赖，即使是很小范围内或很短时间的，也会损害扩展性。所以精心设计数据模型，将数据分片分开存放对于扩展性非常重要。

**一致性与延迟**：当数据库需要提供强一致性或者持久性的时候应该偏向于读写所有副本技术。但是很明显一致性与请求延迟成反比，所以使用若干副本技术会是比较中允的办法。

**故障转移与一致性、扩展性、延迟**：有趣的是容错性与一致性、扩展性、延迟的取舍冲突并不剧烈。通过合理的放弃一些性能与一致性，集群可以容忍多达 up to 的节点失效。这种折中在两阶段提交与 PAXOS 协议的区别里体现得很明显。这种折中的另一个例子是增加特定的一致性保障，比如使用严格会话进程的“读己所写”，但这又增加了故障转移的复杂性。

## 为什么“三选二”有误导性

理解CAP理论的最简单方式是想象两个节点分处分区两侧。允许至少一个节点更新状态会导致数据不一致，即丧失了C性质。如果为了保证数据一致性，将分区一侧的节点设置为不可用，那么又丧失了A性质。除非两个节点可以互相通信，才能既保证C又保证A，这又会导致丧失P性质。一般来说跨区域的系统，设计师无法舍弃P性质，那么就只能在数据一致性和可用性上做一个艰难选择。

不确切地说，NoSQL运动的主题其实是创造**各种可用性优先、数据一致性其次的方案**；而传统数据库坚守ACID特性，做的是相反的事情。

关于数据一致性和可用性的争论，首先表现为ACID和BASE之争。当时BASE还不怎么被接受，主要是大家看重ACID的优点而不愿意放弃。提出CAP理论目的是证明有必要开拓更广阔的设计空间，因此才有了“三选二”公式。

“三选二”的观点在几个方面起了误导作用，**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。

要探索这些细微的差别，就要突破传统的分区处理方式，而这是一项根本性的挑战。因为分区很少出现，CAP在大多数时候允许完美的C和A。但当分区存在或可感知其影响的情况下，就要预备一种策略去探知分区并显式处理其影响。这样的策略应分为三个步骤：**探知分区发生，进入显式的分区模式以限制某些操作，启动恢复过程以恢复数据一致性并补偿分区期间发生的错误。**

## CAP与延迟的联系

CAP理论的经典解释是忽略网络延迟的，但在实际中延迟和分区紧密相关。CAP从理论变为现实的场景发生在操作的间歇，系统需要在这段时间内做出关于分区的一个重要决定。

* 取消操作因而降低系统的可用性，还是
* 继续操作，以冒险损失系统一致性为代价

依靠多次尝试通信的方法来达到一致性，比如Paxos算法或者两阶段事务提交，仅仅是推迟了决策的时间。系统终究要做一个决定；无限期地尝试下去，本身就是选择一致性牺牲可用性的表现。

因此以实际效果而言，分区相当于对通信的时限要求。系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在C和A之间做出选择。这就从延迟的角度抓住了设计的核心问题：**分区两侧是否在无通信的情况下继续其操作？**

从这个实用的观察角度出发可以导出若干重要的推论。第一，分区并不是全体节点的一致见解，因为有些节点检测到了分区，有些可能没有。第二，检测到分区的节点即进入分区模式——这是优化C和A的核心环节。

最后，这个观察角度还意味着设计师可以根据期望中的响应时间，有意识地设置时限；时限设得越短，系统进入分区模式越频繁，其中有些时候并不一定真的发生了分区的情况，可能只是网络变慢而已。

有时候在跨区域的系统，放弃强一致性来避免保持数据一致所带来的高延迟是非常有意义的。Yahoo的PNUTS系统因为以异步的方式维护远程副本而带来数据一致性的问题。但好处是主副本就放在本地，减小操作的等待时间。这个策略在实际中很实用，因为一般来讲，用户数据大都会根据用户的（日常）地理位置做分区。最理想的状况是每一位用户都在他的数据主副本附近。

Facebook使用了相反的策略：主副本被固定在一个地方，因此远程用户一般访问到的是离他较近，但可能已经过时的数据副本。不过当用户更新其页面的时候是直接对主副本进行更新，而且该用户的所有读操作也被短暂转向从主副本读取，尽管这样延迟会比较高。20秒后，该用户的流量被重新切换回离他较近的副本，此时副本应该已经同步好了刚才的更新。

## CAP之惑

CAP理论经常在不同方面被人误解，对于可用性和一致性的作用范围的误解尤为严重，可能造成不希望看到的结果。如果用户根本获取不到服务，那么其实谈不上C和A之间做取舍，除非把一部分服务放在客户端上运行，即所谓的无连接操作或称离线模式。离线模式正变得越来越重要。HTML5的一些特性，特别是客户端持久化存储特性，将会促进离线操作的发展。支持离线模式的系统通常会在C和A中选择A，那么就不得不在长时间处于分区状态后进行恢复。

“一致性的作用范围”其实反映了这样一种观念，即在一定的边界内状态是一致的，但超出了边界就无从谈起。比如在一个主分区内可以保证完备的一致性和可用性，而在分区外服务是不可用的。Paxos算法和原子性多播（atomic multicast）系统一般符合这样的场景。像Google的一般做法是将主分区归属在单一个数据中心里面，然后交给Paxos算法去解决跨区域的问题，一方面保证全局协商一致（global consensus）如Chubby，一方面实现高可用的持久性存储如Megastore。

分区期间，独立且能自我保证一致性的节点子集合可以继续执行操作，只是无法保证全局范围的不变性约束不受破坏。数据分片（sharding）就是这样的例子，设计师预先将数据划分到不同的分区节点，分区期间单个数据分片多半可以继续操作。相反，如果被分区的是内在关系密切的状态，或者有某些全局性的不变性约束非保持不可，那么最好的情况是只有分区一侧可以进行操作，最坏情况是操作完全不能进行。

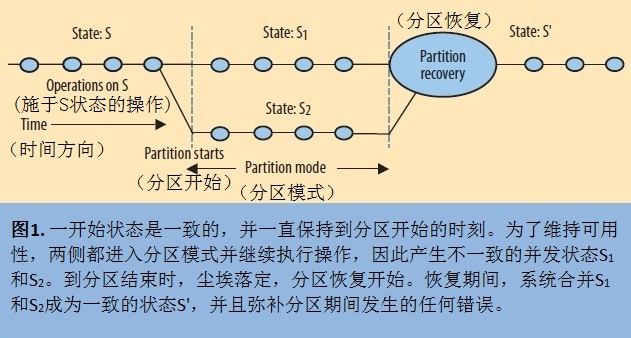
“三选二”的时候取CA而舍P是否合理？已经有研究者指出了其中的要害——怎样才算“舍P”含义并不明确。设计师可以选择不要分区吗？哪怕原来选了CA，当分区出现的时候，你也只能回头重新在C和A之间再选一次。我们最好从概率的角度去理解：选择CA意味着我们假定，分区出现的可能性要比其他的系统性错误（如自然灾难、并发故障）低很多。

这种观点在实际中很有意义，因为某些故障组合可能导致同时丢掉C和A，所以说CAP三个性质都是一个度的问题。实践中，大部分团体认为（位于单一地点的）数据中心内部是没有分区的，因此在单一数据中心之内可以选择CA；CAP理论出现之前，系统都默认这样的设计思路，包括传统数据库在内。然而就算可能性不高，单一数据中心完全有可能出现分区的情况，一旦出现就会动摇以CA为取向的设计基础。最后，考虑到跨区域时出现的高延迟，在数据一致性上让步来换取更好性能的做法相对比较常见。

CAP还有一个方面很多人认识不清，那就是放弃一致性其实有隐藏负担，即需要明确了解系统中存在的不变性约束。满足一致性的系统有一种保持其不变性约束的自然倾向，即便设计师不清楚系统中所有的不变性约束，相当一部分合理的不变性约束会自动地维持下去。相反，当设计师选择可用性的时候，因为需要在分区结束后恢复被破坏的不变性约束，显然必须将各种不变性约束一一列举出来，可想而知这件工作很有挑战又很容易犯错。放弃一致性为什么难，其核心还是“并发更新问题”，跟多线程编程比顺序编程难的原因是一样的。

## 管理分区

怎样缓和分区对一致性和可用性的影响是对设计师的挑战。其关键是以非常明确、公开的方式去管理分区，不仅需要主动察觉分区的发生，还需要为分区期间所有可能受侵害的不变性约束预备专门的恢复过程和计划。管理分区有三个步骤：



1. 检测到分区开始；
2. 明确进入分区模式，限制某些操作；
3. 当通信恢复后启动分区恢复过程。

图中可见分区的演变过程。普通的操作都是顺序的原子操作，因此分区总是在两笔操作之间开始。一旦系统在操作间歇检测到分区发生，检测方一侧即进入分区模式。如果确实发生了分区的情况，那么一般分区两侧都会进入到分区模式，不过单方面完成分区也是可能的。单方面分区要求在对方按需要通信的时候，本方要么能正确响应，要么不需要通信；总之操作不得破坏一致性。但不管怎么样，由于检测方可能有不一致的操作，它必须进入分区模式。采取了quorum决定机制的系统即为单方面分区的例子。其中一方拥有“法定通过节点数”，因此可以执行操作，而另一方不可以执行操作。支持离线操作的系统明显地含有“分区模式”的概念，一些支持原子多播（atomic multicast）的系统也含有这个概念，如Java平台的JGroups。

当系统进入到分区模式，它有两种可行的策略。其一是限制部分操作，因此会削弱可用性。其二是额外记录一些有利于后面分区恢复的操作信息。系统可通过持续尝试恢复通信来察觉分区何时结束。

## 哪些操作可以执行？

决定限制哪些操作，主要取决于系统需要维持哪几项不变性约束。在给定了不变性约束条件之后，设计师需要决定在分区模式下，是否坚持不触动某项不变性约束，抑或以事后恢复为前提去冒险触犯它。例如，对于“表中键的惟一性”这项不变性约束，设计师一般都选择在分区期间放宽要求，容许重复的键。重复的键很容易在恢复阶段检查出来，假如重复键可以合并，那么设计师不难恢复这项不变性约束。

对于分区期间必须维持的不变性约束，设计师应当禁止或改动可能触犯该不变性约束的操作。（一般而言，我们没办法知道操作是否真的会破坏不变性约束，因为无法知道分区另一侧的状态。）信用卡扣费等具有外部化特征的事件常以这种方式工作。适合这种情况的策略，是记录下操作意图，然后在分区恢复后再执行操作。这类事务往往从属于一些更大的工作流，在工作流明确含有类似“订单处理中”状态的情况下，将操作推迟到分区结束并无明显的坏处。设计师以用户不易察觉的方式牺牲了可用性。用户只知道自己下了指令，系统稍后会执行。

说得更概括一点，分区模式给用户界面提出了一种根本性的挑战，即如何传达“任务正在进行尚未完成”的信息。研究者已经从离线操作的角度对此问题进行了一些深入的探索，离线操作可以看成时间很长的一次分区。例如Bayou的日历程序用颜色来区分显示可能（暂时）不一致的条目13。工作流应用和带离线模式的云服务中也常见类似的提醒，前者的例子如交易中的电子邮件通知，后者的例子如Google Docs。

在分区模式的讨论中，我们将关注点放在有明确意义的原子操作而非单纯的读写，其中一个原因是操作的抽象级别越高，对不变性约束的影响通常就越容易分析清楚。大体来说，设计师要建立一张所有操作与所有不变性约束的叉乘表格（cross product），观察并确定其中每一处操作可能与不变性约束相冲突的地方。对于这些冲突情况，设计师必须决定是否禁止、推迟或修改相应的操作。在实践中，这类决定还受到分区前状态和/或环境参数的影响。例如有的系统为特定的数据设立了主节点，那么一般允许主节点执行操作，不允许其他节点操作。

对分区两侧跟踪操作历史的最佳方式是使用版本向量，版本向量可以反映操作间的因果依赖关系。向量的元素是（节点, 逻辑时间）数值对，分别对应一个更新了对象的节点和它最后更新的时间。对于同一对象的两个给定的版本A和B，当所有结点的版本向量一致有A的时间大于或等于B的时间，且至少有一个节点的版本向量有A的时间较大，则A新于B。

如果不可能对版本向量排序，那么更新操作是并发的，而且有可能出现不一致的情况。只要知道分区两侧版本向量的沿革。系统不难判断哪些操作的执行顺序是确定的，哪些操作是并发的。最近的研究成果证明14，当设计师选择可用性优先，一般最多只能将一致性收紧到这样的程度。

## 分区恢复

到了某个时刻，通信恢复，分区结束。由于每一侧在分区期间都是可用的，其状态仍继续向前进展，但是分区会推迟某些操作并侵犯一些不变性约束。分区结束的时刻，系统知道分区两侧的当前状态和历史记录，因为它在分区模式下记录了详尽的日志。当前状态不如历史记录有价值，因为通过历史记录，系统可以判断哪些操作违反了不变性约束，产生了何种外在的后果（如发送了响应给用户）。在分区恢复过程中，设计师必须解决两个问题：

1. 分区两侧的状态最终必须保持一致；
2. 并且必须补偿分区期间产生的错误。

通常情况，矫正当前状态最简单的解决方法是回退到分区开始时的状态，以特定方式推进分区两侧的一系列操作，并在过程中一直保持一致的状态。Bayou就是这个实现机制，它会回滚数据库到正确的时刻并按无歧义的、确定性的顺序重新执行所有的操作，最终使所有的节点达到相同的状态15。同样地，并发版本控制系统CVS在合并分支的时候，也是从从一个共享的状态一致点开始，逐步将更新合并上去。

大部分系统都存在不能自动合并的冲突。比如，CVS时不时有些冲突需要手动介入，带离线模式的wiki系统总是把冲突留在产生的文档里给用户处理。

相反，有些系统用了限制操作的办法来保证冲突总能合并。一个例子就是Google Docs将其文本编辑操作精简为应用样式、添加文本和删除文本。因此，虽然总的来说冲突问题不可解，但现实中设计师可以选择在分区期间限制使用部分操作，以便系统在恢复的时候能够自动合并状态。如果要实施这种策略，推迟有风险的操作是相对简单的实现方式。

还有一种办法是让操作可以交换顺序，这种办法最接近于形成一种解决自动状态合并问题的通用框架。此类系统将线性合并各日志并重排操作的顺序，然后执行。操作满足交换率，意味着操作有可能重新排列成一种全局一致的最佳顺序。不幸的是，只允许满足交换率的操作这个想法实现起来没那么容易。比如加法操作可以交换顺序，但是加入了越界检查的加法就不行了。

Marc Shapiro及其INRIA同事最近的工作对于可交换顺序的操作在状态合并方面的应用起了很大的促进作用。该团队提出一种从理论上证明可以保证分区后合并的数据类型，称为可交换多副本数据类型（commutative replicated data types，CRDTs）。他们介绍了如何使用此类数据结构来：

* 保证分区期间进行的所有操作都是可交换顺序的，或者
* 用“格（lattice）”的数学概念来表示数据，并保证相对于“格”来说，分区期间的所有操作都是单调递增的。

用后一种方法合并状态会汇总分区两边的最大集合。这种方法是对亚马逊购物车合并算法20的形式化总结和改良，合并后的数据是两边购物车的并集，而并运算是一种单调的集合运算。这种策略的坏处是删掉的购物车商品有可能再次出现。

其实CRDTs完全可以实现同时支持增、删操作的分区耐受集合。此方法的本质是维护两个集合：一个放增加的项目，一个放删除的项目，两集合之差即为真正的集合成员。增集合、删集合分别合并起来都不困难，因而增删集合之差合并起来也不困难。在某个时间点上，系统可以从两个集合中清理掉删除的数据项。假如按照一般的设计，像这种清理操作仅在系统没分区的时候才可行，属于设计师必须在分区期间禁止或推迟的特定操作，但是CRDTs的清理操作并不会对可用性产生外在的影响。因此通过CRDTs来实现状态，设计师既保证了可用性，又保证了分区后系统自动合并状态。

## 补偿错误

比计算分区后状态更难解决的问题是如何弥补分区期间造成的错误。跟踪和限制分区模式下的操作，这两种措施足以使设计师确知哪些不变性约束可能被违反，然后分别为它们制定恢复策略。一般系统在分区恢复期间检查违反情况，修复工作也必须在这段时间内完成。

恢复不变性约束的方法有很多，粗陋一点的办法如“最后写入者胜”（因此会忽略部分更新），聪明一点的办法如合并操作和人为跟进事态（human escalation）。人为跟进事态的例子如飞机航班“超售”的情形：可以把乘客登机看作是对之前售票情况的分区恢复，必须恢复“座位数不少于乘客数”这项不变性约束。那么当乘客太多的时候，有些乘客将失去座位，客服最好能设法补偿他们。

航班的例子揭示了一个外在错误（externalized mistake）：假如航空公司没说过乘客一定有座位，这个问题会好解决得多。因此我们看到推迟有风险的操作的又一个理由——到了分区恢复的时候，我们才知道真实的情况。矫正此类错误的核心概念是“补偿（compensation）”；设计师必须设立补偿操作，除了恢复不变性约束，还要纠正外在错误。

技术上CRDTs只允许局部可验证的不变性约束，所以没有补偿的必要，虽然这种限制降低了CRDTs方法本身的能力。用了CRDTs来处理状态合并的设计方案可以允许暂时违反全局性的不变量约束，分区结束后才合并状态，以及履行必要的补偿。

恢复外在错误通常要求知道一些有关外在输出的历史信息。以“喝醉酒打电话”为例，一位老兄不记得自己昨晚喝高了的时候打过几个电话，虽然他第二天白天恢复了正常状态，但通话日志上的记录都还在，其中有些通话很可能是错误的。拨出的电话就是这位老兄的状态（喝高了）的外在影响。而由于这位老兄不记得打过什么电话，也就很难补偿其中可能造成的麻烦。

又以机器为例，电脑可能在分区期间把一份订单执行了两次。如果系统能区分两份一样的订单是有意的还是重复了，它就能取消掉一份重复的订单。如果这次错误产生了外在影响，补偿策略可以是自动生成一封电子邮件，向顾客解释系统意外将订单执行了两次，现在错误已经被纠正，附上一张优惠券下次可以用。假如没有完善的历史记录，就只好靠顾客亲自去发现错误了。

曾经有人正式研究过将补偿性事务作为处理长寿命事务（long-lived transactions）的一种手段21,22。长时间运行的事务会面临另一种形态的分区决策：是长时间持有锁来保证一致性比较好呢？还是及早释放锁向其他事务暴露未提交的数据，提高并发能力比较好呢？比如在单笔事务中更新所有的员工记录就是一个典型例子。按照一般的方式串行化这笔事务，将导致所有的记录都被锁定，阻止并发。而补偿性事务采取另一种方式，它将大事务拆成多个分别提交的子事务。如果要中止大事务，系统必须发起一笔新的、起纠正作用的事务，逐一撤销所有已经提交的子事务，这笔新事务就是所谓的补偿性事务。

总的来说，补偿性事务的目的是避免中止其他用了未正确提交数据的事务（即不允许级联取消）。这种方案不依赖串行化或隔离的手段来保障正确性，其正确性取决于事务序列对状态和输出所产生的净影响。那么，经过补偿，数据库的状态究竟是不是相当于那些子事务根本没执行过一样呢？考虑等价必须连外在行为也包括在内；举个例子，把重复扣取的交易款退还给顾客，很难说成等于一开始就没多收顾客的钱，但从结果上看勉强算扯平了。分区恢复也延续同样的思路。虽然服务不一定总能直接撤销其错误，但起码承认错误并做出新的补偿行为。怎样在分区恢复中运用这种思路效果最好，这个问题没有固定的答案。“自动柜员机上的补偿问题”小节以一个很小的应用领域为例点出了一些思考方向。

当系统中存在分区，系统设计师不应该盲目地牺牲一致性或可用性。运用以上讨论的方法，设计师通过细致地管理分区期间的不变性约束，两方面的性质都可以取得最佳的表现。随着版本向量和CRDTs等比较新的技术逐渐被纳入一些简化其用法的框架，这方面的优化手段会得到比较普遍的应用。但引入CAP实践毕竟不像引入ACID事务那么简单，实施的时候需要对过去的策略进行全面的考虑，最佳的实施方案极大地依赖于具体服务的不变性约束和操作细节。

## 自动柜员机上的补偿问题

以自动柜员机（ATM）的设计来说，强一致性看似符合逻辑的选择，但现实情况是可用性远比一致性重要。理由很简单：高可用性意味着高收入。不管怎么样，讨论如何补偿分区期间被破坏的不变性约束，ATM的设计很适合作为例子。

ATM的基本操作是存款、取款、查看余额。关键的不变性约束是余额应大于或等于零。因为只有取款操作会触犯这项不变性约束，也就只有取款操作将受到特别对待，其他两种操作随时都可以执行。

ATM系统设计师可以选择在分区期间禁止取款操作，因为在那段时间里没办法知道真实的余额，当然这样会损害可用性。现代ATM的做法正相反，在stand-in模式下（即分区模式），ATM限制净取款额不得高于k，比如k为$200。低于限额的时候，取款完全正常；当超过限额的时候，系统拒绝取款操作。这样，ATM成功将可用性限制在一个合理的水平上，既允许取款操作，又限制了风险。

分区结束的时候，必须有一些措施来恢复一致性和补偿分区期间系统所造成的错误。状态的恢复比较简单，因为操作都是符合交换率的，补偿就要分几种情况去考虑。最后的余额低于零违反了不变性约束。由于ATM已经把钱吐出去了，错误成了外部实在。银行的补偿办法是收取透支费并指望顾客偿还。因为风险已经受到限制，问题并不严重。还有一种情况是分区期间的某一刻余额已经小于零（但ATM不知道），此时一笔存款重新将余额变为正的。银行可以追溯产生透支费，也可以因为顾客已经缴付而忽略该违反情况。

总而言之，因为通信延迟的存在，银行系统不依靠一致性来保证正确性，而更多地依靠审计和补偿。“空头支票诈骗”也是类似的例子，顾客赶在多家分行对账之前分别取出钱来然后逃跑。透支的错误过后才会被发现，对错误的补偿也许体现为法律行动的形式。

# ACID

ACID是强一致性协议，Atomicity、Consistency、Isolation和Durability

**原子性**：事务中的操作要么都做，要么都不做。所有的系统都受惠于原子性操作。当我们考虑可用性的时候，没有理由去改变分区两侧操作的原子性。而且满足ACID定义的、高抽象层次的原子操作，实际上会简化分区恢复。

**一致性**：ACID的C指的是事务不能破坏任何数据库规则，如键的唯一性。与之相比，CAP的C仅指单一 副本这个意义上的一致性，因此只是ACID一致性约束的一个严格的子集。ACID一致性不可能在分区过程中保持，因此分区恢复时需要重建ACID一致性。推而广之，分区期间也许不可能维持某些不变性约束，所以有必要仔细考虑哪些操作应该禁止，分区后又如何恢复这些不变性约束。

**隔离性**：一个事务的执行不能被其他事务所干扰。隔离是CAP理论的核心：如果系统要求ACID隔离性，那么它在分区期间最多可以在分区一侧维持操作。事务的可串行性（serializability）要求全局的通信，因此在分区的情况下不能成立。只要在分区恢复时进行补偿，在分区前后保持一个较弱的正确性定义是可行的。

**持续性**：一个已提交的事务对数据库中数据的改变是永久性的。牺牲持久性没有意义，理由和原子性一样，虽然开发者有理由（持久性成本太高）选择BASE风格的软状态来避免实现持久性。这里有一个细节，分区恢复可能因为回退持久性操作，而无意中破坏某项不变性约束。但只要恢复时给定分区两侧的持久性操作历史记录，破坏不变性约束的操作还是可以被检测出来并修正的。通常来讲，让分区两侧的事务都满足ACID特性会使得后续的分区恢复变得更容易，并且为分区恢复时事务的补偿工作奠定了基本的条件。

事务是用户定义的一个数据库操作序列，这些操作要么全不做，要么全做，是一个不可分割的工作单位。调当程序员定义的事务完成时，数据库处于一致的状态，比如对于转帐来说，事务完成时必须是A少了多少钱B就多了多少钱。ACID是事务所具有的特性。

# BASE

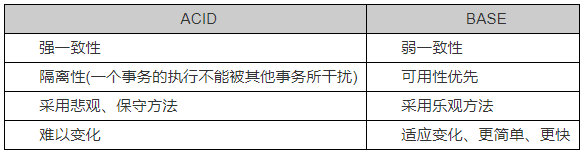
BASE是弱一致性协议，Basically Available、Soft-state、Eventually consistency。

基本可用：系统能够基本运行，一直提供服务；

软状态：系统不要求一直保持强一致状态；

最终一致性：系统需要在某一时刻后达到一致性要求。

# ACID和BASE的比较



ACID和BASE代表了两种截然相反的设计哲学，分别处于一致性和可用性图谱的两端。ACID注重一致性，是数据库的传统设计思路。而BASE是根据一些针对高可用性的设计思路发展而来。现代大规模跨区域分布的系统，同时运用了这两种思路。这两个术语都好记有余而精确不足，出现较晚的BASE硬凑的感觉更明显。

CAP与ACID的关系更复杂一些，也因此引起更多误解。其中一个原因是ACID的C(Consistency)和A(Atomicity)字母所代表的概念不同于CAP的C(Consistency)和A(Avalability)。还有一个原因是选择可用性只部分地影响ACID约束。

# 一致性理解

在分布式系统中，强一致性从client和server两个角度分析。

从client角度理解：更新操作执行成功后，所有的用户都能够读取到最新的值；

从server角度理解：在任意时刻，所有节点中的数据是一样的，要求更新过的数据能被后续的访问都能看到。要求无论更新操作是在哪个数据副本上执行，之后所有的读操作都要能获得最新的数据。

对于单副本数据来说，读写操作是在同一数据上执行的，容易保证一致性，但是对于多副本的数据来说，则需要使用分布式事务协议（2PC或者Paxos等）。

如果数据更新后，能容忍后续部分访问不到，但是不是全部都访问不到，则是弱一致性。在这种一致性下，用户读到某一操作对系统特定数据的更新需要一段时间，我们将这段时间称为"不一致性窗口"。**弱一致性有弱到什么程度的问题，因此包含很多种不同的实现，目前分布式系统中广泛实现的是最终一致性。**

正如CAP理论所指出的，一致性、可用性和分区容错性不能同时满足。对于数据不断增长的系统（如社会计算、网络服务的系统），它们**对可用性及分区容错性的要求高于强一致性**，一些分布式系统通过复制数据来提高系统的可靠性和容错性，并且将数据的不同的副本存放在不同的机器上，由于维护数据副本的一致性代价很高，因此许多系统采用弱一致性来提高性能，一些不同的一致性模型也相继被提出。

**最终一致性是弱一致性的一种**，不保证在任意时刻任意节点上的同一份数据都是相同的，但是随着时间的迁移，不同节点上的同一份数据总是在向趋同的方向变化，在这种一致性下系统保证用户最终能够读取到某操作对系统特定数据的更新。也可以简单理解为在一段时间后，节点间的数据会最终达到一致性状态，这其实就是BASE。此种情况下，如果没有发生失败，"不一致性窗口"的大小依赖于交互延迟、系统的负载，以及复制技术中副本的个数（这个可以理解为master/slave模式中，slave的个数）。DNS系统可以说是在最终一致性方面最出名的系统，当更新一个域名的IP以后，根据配置策略以及缓存控制策略的不同，最终所有的客户都会看到最新的值。

最终一致性模型也根据其提供的不同保证可以划分为更多的模型：

因果一致性：假如有相互独立的A、B、C三个进程对数据进行操作。进程A对某数据进行更新后并将该操作通知给B，那么B接下来的读操作能够读取到A更新的数据值。但是由于A没有将该操作通知给C，那么系统将不保证C一定能够读取到A更新的数据值。

读自写一致性：这个一致性是指用户更新某个数据后读取该数据时能够获取其更新后的值，而其他的用户读取该数据时则不能保证读取到最新值。

会话一致性：是指读取自写更新一致性被限制在一个会话的范围内，也就是说提交更新操作的用户在同一个会话里读取该数据时能够保证数据是最新的。

单调一致性：是指用户读取某个数据值，后续操作不会读取到该数据更早版本的值。

时间轴一致性：要求数据的所有副本以相同的顺序执行所有的更新操作。

# 参考

<http://blog.csdn.net/duxingxia356/article/details/43992015>

<http://www.infoq.com/cn/articles/cap-twelve-years-later-how-the-rules-have-changed/>

<http://www.jdon.com/bigdata/how-to-understand-cap.html>

NoSQL的比较

<http://highscalability.com/blog/2015/12/30/how-to-choose-an-in-memory-nosql-solution-performance-measur.html>