[**CAP原理与最终一致性 强一致性 透析**](http://www.blogjava.net/hello-yun/archive/2012/04/27/376744.html)

参考：<http://www.blogjava.net/hello-yun/archive/2012/04/27/376744.html>

在足球比赛里，一个球员在一场比赛中进三个球，称之为帽子戏法(Hat-trick)。在分布式数据系统中，也有一个帽子原理(CAP Theorem)，不过此帽子非彼帽子。CAP原理中，有三个要素：

* 一致性(**C**onsistency)
* 可用性(**A**vailability)
* 分区容忍性(**P**artition tolerance)

CAP原理指的是，这三个要素最多只能同时实现两点，不可能三者兼顾。因此在进行分布式架构设计时，必须做出取舍。而**对于分布式数据系统，分区容忍性是基本要求**，否则就失去了价值。因此设计分布式数据系统，就是在一致性和可用性之间取一个平衡。对于大多数web应用，其实并不需要强一致性，因此牺牲一致性而换取高可用性，是目前多数分布式数据库产品的方向。

当然，牺牲一致性，并不是完全不管数据的一致性，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，而是只要系统能达到**最终一致性**即可，考虑到客户体验，这个最终一致的时间窗口，要尽可能的对用户透明，也就是需要保障“用户感知到的一致性”。通常是通过数据的多份异步复制来实现系统的高可用和数据的最终一致性的，“用户感知到的一致性”的时间窗口则取决于数据复制到一致状态的时间。

**最终一致性(eventually consistent)**

对于一致性，可以分为从客户端和服务端两个不同的视角。从客户端来看，一致性主要指的是多并发访问时更新过的数据如何获取的问题。从服务端来看，则是更新如何复制分布到整个系统，以保证数据最终一致。一致性是因为有并发读写才有的问题，因此在理解一致性的问题时，一定要注意结合考虑并发读写的场景。

从客户端角度，多进程并发访问时，更新过的数据在不同进程如何获取的不同策略，决定了不同的一致性。对于关系型数据库，要求更新过的数据能被后续的访问都能看到，这是**强一致性**。如果能容忍后续的部分或者全部访问不到，则是**弱一致性**。如果经过一段时间后要求能访问到更新后的数据，则是最终一致性。

最终一致性根据更新数据后各进程访问到数据的时间和方式的不同，又可以区分为：

* ***因果一致性****。如果进程A通知进程B它已更新了一个数据项，那么进程B的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程A无因果关系的进程C的访问遵守一般的最终一致性规则。*
* ***“读己之所写（read-your-writes）”一致性****。当进程A自己更新一个数据项之后，它总是访问到更新过的值，绝不会看到旧值。这是因果一致性模型的一个特例。*
* ***会话（Session）一致性****。这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统的保证不会延续到新的会话。*
* ***单调（Monotonic）读一致性****。如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。*
* ***单调写一致性****。系统保证来自同一个进程的写操作顺序执行。要是系统不能保证这种程度的一致性，就非常难以编程了。*

上述最终一致性的不同方式可以进行组合，例如单调读一致性和读己之所写一致性就可以组合实现。并且从实践的角度来看，这两者的组合，读取自己更新的数据，和一旦读取到最新的版本不会再读取旧版本，对于此架构上的程序开发来说，会少很多额外的烦恼。

从服务端角度，如何尽快将更新后的数据分布到整个系统，降低达到最终一致性的时间窗口，是提高系统的可用度和用户体验非常重要的方面。对于分布式数据系统：

* N — 数据复制的份数
* W — 更新数据是需要保证写完成的节点数
* R — 读取数据的时候需要读取的节点数

如果W+R>N，写的节点和读的节点重叠，则是强一致性。例如对于典型的一主一备同步复制的关系型数据库，N=2,W=2,R=1，则不管读的是主库还是备库的数据，都是一致的。

如果W+R<=N，则是弱一致性。例如对于一主一备异步复制的关系型数据库，N=2,W=1,R=1，则如果读的是备库，就可能无法读取主库已经更新过的数据，所以是弱一致性。

对于分布式系统，为了保证高可用性，一般设置N>=3。不同的N,W,R组合，是在可用性和一致性之间取一个平衡，以适应不同的应用场景。

* 如果N=W,R=1，任何一个写节点失效，都会导致写失败，因此可用性会降低，但是由于数据分布的N个节点是同步写入的，因此可以保证强一致性。
* 如果N=R,W=1，只需要一个节点写入成功即可，写性能和可用性都比较高。但是读取其他节点的进程可能不能获取更新后的数据，因此是弱一致性。这种情况下，如果W<(N+1)/2，并且写入的节点不重叠的话，则会存在写冲突