CAP theorem 的发展和总结

Turnura King

September 2023

1 Background

对于企业来说，强一致性数据是最容易满足的属性，很多具有 ACID 特性的关系型数据库都可以实现该属性，但将单个机器扩展到分布式/多机领域，就可能会节点故障等问题。2000 年，Eric Brewer 教授在 PODC 的研讨会上提出了一个猜想：一致性、可用性和分区容错性三者无法在分布式系统中被同时满足，并且最多只能满足其中两个！2002 年，Lynch 与其他人证明了 Brewer 猜想，从而把 CAP 上升为一个定理。但是，她只是证明了 CAP 三者不可能同时满足，并没有证明任意二者都可以满足，所以，该证明被认为是一个收窄的结果。在后续的二十年中，CAP 理论进行了充分的发展，人们在分布式系统构建领域，围绕着 CAP 理论展开了激烈的讨论。

2 Theorem

现在介绍下 CAP 理论：

Theorem 1 *在分布式服务中，无法同时保证三种属性同时成立：可用性（available），一致性（consistent）和分区容错性（Partitioned-Tolerant）*

在论文[2] 中，对这三个属性分别进行了抽象，正式定义一致性、可用性和分区容错这些术语的含义。

1. Consistency: 将 Consistency 更精细分为原子一致性。区别于数据库中的 ACID 特性，原子一致性仅指单个请求/响应操作序列的属性。它与 ACID 中的原子具有不同的含义，因为它包含原子性和一致性的数据库概念。即 Atomic 原子一致性 = linearizable 线性一致性 = strong 强一致性，此时任何在分布式系统上的操作都好像是在单个节点上执行的：即每次 read 都能读到最新的值（也有可能是旧的、但一致的值）。越好的一致性也导致系统读写的延迟大大增加，因为在节点之间的数据复制，数据传播所需的时间都被延长了。

2. Available: 对系统的读写请求是否最终被执行？当读写请求最终都被执行/返回一个正确响应，那么这个数据存储系统具有可用性。可用性只关心是否最终正确响应，而对响应时间没有要：是为了满足一致性这样的特性，所以节点之间需要通信。可用性要求系统即使在出错的情况下，仍然能够相应用户的请求，但这也有可能导致获取到过时或不一致的数据。

3. Partitioned- Tolerant: 即使系统发生了 partition 行为，系统仍需要正常工作：它确保通信故障期间的可用性，但可能会导致一致性权衡。即使系统内的某些节点无法运行或出现网络故障，系统也应保持其功能而不会出现任何中断。此处的正常工作即要求之前两个属性都能被满足。

2.1 Understanding the Trades-Offs

1. CA:
   (a) 在数据完整性和准确性至关重要的场景中，选择 CA 可能是正确的选择。
   (b) 然而，这可能会导致网络分区或故障期间系统响应能力下降
2. CP:
   (a) 在数据一致性至关重要且网络分区不可忽视的应用中，CP 是一个合理的选择
   (b) 这可能会导致分区事件期间暂时不可用
3. AP:
   (a) 当系统响应能力至关重要且预期出现网络分区时，AP 成为合适的选择
   (b) 这可能会导致节点之间的数据潜在不一致

此处单机模式能很好的满足 Consistency 和 Available，因此 CAP 理论讨论的是在分区过程中，能否同时满足 Available 和 Consistency。

3 Proof

3.1 Asynchronous Network

为了证明 CAP 理论，论文[2] 使用了异步网络，在异步模型中，没有时钟，节点必须仅根据收到的消息和本地计算做出决策。在论文中，使用反证法假设存在某个分布式系统（两节点）同时满足三个属性，然后最后证明该系统返回值与原子一致性矛盾（这里反证法没太看懂，很多概念都与现在的概念定义不同）。在论文中给出了三个属性两两组合的例子

1. Consistency(文中使用 Atomic）和 Partition Tolerant : 最简单满足该要求的是忽略所有请求的系统。更实用的一个例子是集中性协议，系统中存在一个维护所有变量的中心节点，所有接受到读写请求的非中心节点再将请求转发给中心节点，但如果中心节点出错，那么实用性将会减弱，但仍然可以满足 C 和 P 属性。
2. Consistency 和 Available： 单机模式即满足该属性，文中给出了 Intranet 和 LAN 上运行的系统的例子。
3. Available 和 Partition-Tolerant： 如果没有一致性要求，每次请求直接返回接收请求节点的当前值就可以。更实用的例子是 Web 缓存，可以提供弱一致性。

4 Partially Synchronous Networks

论文 [2] 认为规避到 CAP 理论是不现实的，因为在现实生活中，大多数网络并不是存粹异步的，即 Partially Synchronous Networks: 部分同步网络部分同步网络和异步网络区别：网络中每个节点都有一个时钟，每个时钟都以相同速率增加，它可以在同一实时显示不同的值进程可以通过观察本地时钟的值，来测量自己的运行时间。

Theorem 2 即使是部分同步网络，如果任意消息仍可能丢失，也不可能有一个总是可用、原子一致性的数据模型

该定理仍然使用反证法来证明，证明方法与定理一的类似。

4.1 Solution in the Partially Synchronous Model

在部分同步网络中，**Theorem 1** 不再成立，因为 **Theorem 1** 实际上依赖于节点不知道消息何时丢失。有一些部分同步算法，当执行中的所有消息都被传递时（即没有分区），它们将返回原子数据，并且仅在消息丢失时返回不一致（特别是陈旧）的数据。在上一节中描述的集中性协议就是该算法的例子，如果收到来自中央节点的响应，则该节点将传送所请求的数据（或确认）。如果在 $2 * t_{msg} + t_{local}$ 内没有收到响应，则节点断定消息已丢失。然后向客户端发送响应：发送本地节点的最佳已知值（对于读取操作），或确认（对于写操作）。在这种情况下，可能会违反原子一致性。但这种情况下，可能会影响原子一致性。

4.2 Weaker Consistency Conditions

在同时保证可用性和分区容错性之外，可以先保证所访问的数据具有最终一致性，即允许在存在分区时返回陈旧数据，但仍然对返回的陈旧数据的质量提出正式要求，对分区修复后恢复一致性所需时间提出了时间限制。

以上为 Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services 的主要内容，异步模型无法同时保证三属性同时成立，但在部分同步模型中，可以在一致性和可用性之间实现实际的折中。

5 CAP Twelve years later how the rules have changed

本节介绍 CAP 理论在提出后 20 年，工业界在解决 CAP 理论的方法[1]。

在 CAP 理论提出后，关于一致性和可用性的相对优点的很多争论，但也过度放大了两者之间的沟壑。CAP 理论其实只是禁止设计空间的一小部分：再存在分区时不能同时完美的实现可用性和一致性，现代 CAP 理论的目标应该是最大限度的组合一致性和可用性，接受更广泛的系统和权衡, 这和下一节讨论的问题不谋而合。

现如今 NoSQL 领域在 CAP 属性中，大多会将注意力放在可用性上，其次再关注一致性，而遵守 ACID 特性的数据库则相反，首先放在一致性上，其次再关注可用性

5.1 What is the difference between ACID, BASE and CAP

ACID 和 BASE 代表了一致性 - 可用性范围的两个极端，ACID 极其重视一致性，而 BASE 极其重视可用性 ACID 和 CAP 之间的关系比较复杂并且经常被误解，部分原因是 ACID 中的 C 和 A 代表与 CAP 中相同字母的概念不同，部分原因是选择可用性仅影响某些 ACID 保证。以下讲解 ACID 和 CAP 的交叉：

1. A : Atomicity

2. C : Consistency 。表示事务保留所有数据库规则，例如唯一键。相比之下，CAP 中的 C 仅指单副本一致性，它是 ACID 一致性的严格子集。ACID 一致性也无法跨分区维护——分区恢复需要恢复 ACID 一致性。更一般地说，在分区期间维护不变量可能是不可能的，因此需要仔细考虑禁止哪些操作以及在恢复期间如何恢复不变量。

3. I : Isolation。隔离是 CAP 定理的核心：如果系统需要 ACID 隔离，则在分区期间最多可以在一侧进行操作。可串行性通常需要通信，因此跨分区会失败。通过分区恢复期间的补偿，较弱的正确性定义可以跨分区实现。

4. D : Durability。与原子性一样，没有理由放弃持久性，尽管开发人员可能会因为其费用而选择通过软状态（以 BASE 风格）避免需要它。一个微妙的点是，在分区恢复期间，可以反转在操作期间不知不觉地违反不变量的持久操作。然而，在恢复时，鉴于双方都有持久的历史记录，此类操作可以被检测到并纠正。一般来说，在分区的每一侧运行 ACID 事务使恢复变得更容易，并启用一个用于补偿可用于从分区恢复的事务的框架。

BASE 中软状态和最终一致性是存在分区的情况下运行良好的技术，从而提高可用性。

1. Basically Available

2. Soft State

3. Eventually consistent

由于收入目标和合同规范，系统可用性非常重要，因此我们发现自己经常选择通过使用缓存或记录更新以供以后协调等策略来优化可用性。尽管这些策略确实提高了可用性，但其代价是一致性下降。

这种一致性与可用性争论的第一个版本是 ACID 与 BASE，这在当时并没有受到很好的欢迎，主要是因为人们喜欢 ACID 特性并且不愿意放弃它们。CAP 定理的目的是证明探索更广阔的设计空间的必要性——因此出现了“2 of 3”的表述。

具有迷惑性的点

1. 因为分区是少有的，当不分区时这并不能成为舍弃 C 或 A 性质的原因
2. 其次，C 和 A 之间的选择可以在同一系统中以非常细的粒度多次发生；子系统不仅可以做出不同的选择，而且选择可以根据操作甚至所涉及的特定数据或用户而改变。
3. 最后，所有三个属性都比二进制更加连续。可用性显然是从 0% 到 100% 连续的，但一致性也有很多级别，甚至分区也有细微差别，包括系统内部对于分区是否存在存在分歧

探索这些细微差别需要推动传统的分区处理方式，这是根本性的挑战。因为分区很少见，所以 CAP 在大多数情况下应该允许完美的 C 和 A，但是当存在或感知到分区时，检测分区并明确说明它们的策略是适当的。**该策略应该包含三个步骤：检测分区，进入可以限制某些操作的显式分区模式，以及启动恢复过程以恢复一致性并补偿分区期间发生的错误**。

5.2 CAP-Latency Connection

在其证明 CAP 理论的论文中 [2]，CAP 的证明忽略了延时，而在实践中，延迟和分区密切相关。从操作上来说，CAP 的本质发生在超时期间，在此期间程序必须做出基本决策-分区决策：

1. 取消操作，然后降低可用性
2. 处理程序，也因此会出现不一致性

重试通信以实现一致性（例如通过 Paxos 或两阶段提交）只会延迟决策。在某个时刻，程序必须做出决定；无限期地重试通信本质上是选择 C 而不是 A。因此，实际上，分区是对通信的时间限制。未能在时间限制内实现一致性意味着存在分区，因此该操作需要在 C 和 A 之间进行选择。**这些概念抓住了与延迟相关的核心设计问题：双方是否在没有通信的情况下继续前进？**

之前一直理解的 Partition-Tolerance 是分布式系统某个节点出现错误时，系统如何应对错误的节点，而真正意义上，分区容错是当两节点通信中断或有长延迟时，该系统进入分区模式，即此时系统各节点数据不一致。

这种务实的观点产生了几个重要的后果。首先，不存在分区的全局概念，因为某些节点可能会检测到分区，而其他节点可能不会。第二个结果是节点可以检测到分区并进

入分区模式——这是优化 C 和 A 的核心部分。最后，这种观点意味着设计者可以根据目标响应时间有意设置时间界限；边界更严格的系统可能会更频繁地进入分区模式，有时甚至是在网络速度缓慢且实际上未分区的情况下。有时放弃强大的 C 以避免在大范围内保持一致性的高延迟是有意义的。

雅虎的 PNUTS 系统通过异步维护远程副本而导致不一致。然而，它使主副本位于本地，从而减少了延迟。这种策略在实践中效果很好，因为单个用户数据是根据用户的（正常）位置自然分区的。理想情况下，每个用户的数据主机都在附近。

Facebook 使用相反的策略：主副本始终位于一个位置，因此远程用户通常拥有更接近但可能过时的副本。然而，当用户更新其页面时，尽管延迟较高，但更新会直接转到主副本，就像用户在短时间内进行的所有读取一样。20 秒后，用户的流量恢复到更接近的副本，此时应该反映更新。

5.3 CAP Confusion

如果用户根本无法访问服务，则也不需要在 C 和 A 之间做出选择。这种异常称为断开连接操作或离线模式，变得越来越重要。

一些 HTML5 功能（特别是客户端持久存储）使断开连接的操作变得更加容易。这些系统通常选择 A 而不是 C，因此必须从长分区中恢复。一致性范围反映了这样的想法：在某个边界内，状态是一致的，但在该边界之外，所有的赌注都将被取消。例如：在主分区内，可以保证完全的一致性和可用性，而在分区外，服务不可用。

Paxos 和原子多播系统通常符合这种情况。在 Google，主分区通常驻留在一个数据中心内；然而，Paxos 广泛用于确保全局共识，如 Chubby；例如高可用持久存储，如 MegaStore

独立的、自洽的子集可以在分区时取得进展，尽管不可能确保全局不变量。例如，通过分片，设计人员跨节点预分区数据，每个分片很可能在分区期间取得一些进展。相反，如果相关状态被分割成一个分区，或者全局不变量是必要的，那么最好的情况下只有一侧可以取得进展，而最坏的情况下则不可能取得进展。

选择一致性和可用性 (CA) 作为 “3 之 2” 有意义吗？正如一些研究人员正确指出的那样，放弃 P 究竟意味着什么尚不清楚。设计师可以选择不设置分区吗？如果选择 CA，然后出现分区，则选择必须恢复为 C 或 A。== 此时讲的是先保证 C 和 A 的时候，如果监测到分区，如何处理？最好从概率上考虑这个问题：选择 CA 应该意味着分区的概率远小于其他系统故障的概率，例如灾难或多个同时发生的故障。

这种观点是有道理的，因为真实系统在某些故障集下会同时失去 C 和 A，因此所有三个属性只是程度问题。在实践中，大多数团队假设数据中心（单站点）内部没有分区，因此在单站点内设计 CA；此类设计（包括传统数据库）是 CAP 之前的默认设计。然而，尽管数据中心内分区的可能性较小，但它们确实是可能的，这使得 CA 目标存在问题。最后，考虑到广域范围内的高延迟，为了获得更好的性能而放弃广域范围内的完

美一致性是相对常见的。

CAP 混乱的另一个方面是放弃一致性的隐性成本，即需要了解系统的不变量。一致系统的微妙之美在于，不变量往往保持不变，即使设计师不知道它们是什么。最后，一系列合理的不变量就可以很好地工作。相反，当设计者选择 A 时，需要在分区后恢复不变量，他们必须明确所有不变量，这既具有挑战性又容易出错。从本质上讲，这与使多线程比顺序编程更困难的并发更新问题相同。

5.4 Managing Partitions

对于设计人员来说，具有挑战性的情况是减轻分区对一致性和可用性的影响。关键思想是非常明确地管理分区，不仅包括检测，还包括特定的恢复过程以及针对分区期间可能违反的所有不变量的计划。这种管理方法分为三个步骤：

1. 检测分区的开始
2. 进入显示分区模式，这可能会限制某些操作
3. 当通信恢复时，启动分区恢复

最后一步的目的是恢复一致性并补偿系统分区时程序所犯的错误。图 1 显示了分区

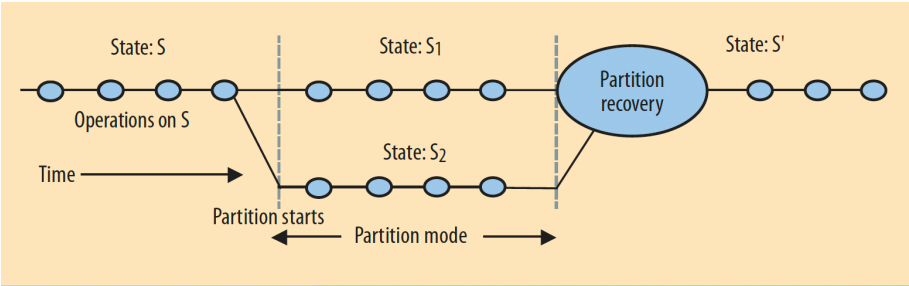

Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S′ and also compensates for any mistakes made during the partition.

图 1: 分区演变

的演变。正常操作是一系列原子操作，因此分区始终在操作之间启动。一旦系统超时，就会检测到分区，检测端进入分区模式。如果确实存在分区，双方都会进入此模式，但单侧分区也是可能的。在这种情况下，另一方根据需要进行通信，而本方要么正确响应，要么不需要通信；无论哪种方式，操作都保持一致。但由于检测端的操作可能不一致，因此必须进入分区模式。使用仲裁的系统就是这种单方面分区的一个例子。一方将达到法定人数并可以继续进行，但另一方则不能。支持断开连接操作的系统显然有分区模式的概念，一些原子多播系统也是如此，例如 Java 的 JGroups。

一旦系统进入分区模式，就有两种可能的策略。首先是限制某些操作，从而降低可用性。第二个是记录有关分区恢复期间有用的操作的额外信息。继续尝试通信将使系统能够辨别分区何时结束。

5.4.1 Which operations should proceed?

决定限制哪些操作主要取决于系统必须维持的不变量。

例如，对于表中的键是唯一的不变量，设计人员通常决定冒这个不变量的风险，并在分区期间允许重复的键。重复的键在恢复过程中很容易检测到，并且假设它们可以合并，设计者可以轻松地恢复不变量。然而，对于在分区期间必须维护的不变量，设计者必须禁止或修改可能违反它的操作。（一般来说，无法判断该操作是否实际上违反了不变量，因为另一方的状态是不可知的）。外部化事件（例如信用卡收费）通常以这种方式进行。

在这种情况下，策略是记录事件并在恢复后执行。此类事务通常是具有显式订单处理状态的较大工作流程的一部分，并且将操作延迟到分区结束几乎没有什么缺点。设计者以用户看不到的方式放弃了 A。用户只知道他们下了订单，系统稍后会执行它。

更一般地说，分区模式带来了一个基本的用户界面挑战，即传达任务正在进行但未完成的信息。研究人员已经针对断开操作（只是一个长分区）详细探讨了这个问题。例如，Bayou 的日历应用程序显示出潜在的不一致（暂定）不同颜色的条目。此类通知在工作流程应用程序（例如具有电子邮件通知的商务）和具有离线模式的云服务（例如 Google Docs）中经常可见。

关注显式原子操作而不仅仅是读取和写入的原因之一是，分析高级操作对不变量的影响要容易得多。本质上，设计者必须构建一个表来查看所有操作和所有不变量的叉积，并为每个条目确定该操作是否可能违反不变量。如果是这样，设计者必须决定是否禁止、延迟或修改操作。在实践中，这些决策还可以取决于已知状态、论点或两者。例如，在具有用于某些数据的主节点的系统中，操作通常可以在主节点上进行，但不能在其他节点上进行。

跟踪双方操作历史的最佳方法是使用版本向量，它捕获操作之间的因果依赖关系。向量的元素是一对（节点，逻辑时间），每个已更新对象的节点及其上次更新的时间都有一个条目。给定一个对象的两个版本 A 和 B，如果对于其向量中的每个公共节点，A 的时间大于或等于 B 的时间，并且 A 的至少一个时间大于 B，则 A 比 B 新。

如果不可能对向量进行排序，则更新是并发的并且可能不一致。因此，给定双方的版本向量历史记录，系统可以轻松判断哪些操作已经按已知顺序以及哪些操作同时执行。最近的工作证明，如果设计者选择关注可用性，这种因果一致性通常是最好的结果。

5.4.2 Partition Recovery

设计师必须解决两个在恢复过程中遇到的难题

1. 双方的状态必须一致

2. 必须对分区模式期间所犯的错误进行补偿

通常，通过从分区时的状态开始并以某种方式回滚两组操作，并在此过程中保持一致的状态，来修复当前状态通常更容易。Bayou 明确地通过将数据库回滚到正确的时间并以明确定义的确定性顺序重播全套操作来实现这一点，以便所有节点达到相同的状态。同样，源代码控制系统（例如并发版本控制）系统（CVS）从共享一致点开始，前滚更新以合并分支。

5.4.3 Compensating for mistakes

除了计算分区后状态之外，还有一个更困难的问题是修复分区期间发生的错误。分区模式操作的跟踪和限制确保了知道哪些不变量可能被违反，这反过来又使设计者能够为每个这样的不变量创建恢复策略。通常，系统在恢复期间发现违规，并且必须在那时实施任何修复。实现方法:

1. 最后写者胜

2. 更优质的汇总多种操作的方法

3. 人为升级

后者的例子是飞机超售：登机在某种意义上是分区恢复，其不变量是必须至少与乘客一样多的座位。如果乘客太多，有些人会失去座位，理想情况下，客户服务会以某种方式补偿这些乘客。

飞机的例子还展示了一个外部化的错误：如果航空公司没有说乘客有座位，那么解决问题就会容易得多。这是推迟危险行动的另一个原因：在恢复之时，真相就已知晓。补偿的想法确实是纠正此类错误的核心；设计者必须创建补偿操作，既恢复不变性，又更广泛地纠正外部错误。

CRDT (conflict-free replicated data type) 无冲突复制数据类型，是一种可以在网络中的多台计算机上复制的数据结构，副本可以独立和并行地更新，不需要在副本之间进行协调，并保证不会有冲突发生。

从技术上讲，CRDT 只允许本地可验证的不变量——这一限制使得补偿变得不必要，但这在一定程度上降低了该方法的威力。然而，使用 CRDT 进行状态收敛的解决方案可能允许暂时违反全局不变量，在分区后收敛状态，然后执行任何所需的补偿。

在机器环境中，计算机可以在分区期间执行命令两次。如果系统可以区分两个有意订单和两个重复订单，则可以取消其中一个重复订单。如果外部化，一种补偿策略是自

动生成一封电子邮件给客户，解释系统意外执行了两次订单，但错误已得到修复，并附上一张优惠券，以便在下一次订单时享受折扣。然而，如果没有正确的历史记录，发现错误的责任就落在了客户身上。

一些研究人员已经正式探索补偿交易作为处理长期交易的一种方式。长时间运行的事务面临分区决策的变化：是长时间持有锁以确保一致性更好，还是尽早释放锁并将未提交的数据暴露给其他事务但允许更高的并发性更好？一个典型的例子是尝试将所有员工记录作为单个事务进行更新。序列化此事务以正常方式锁定所有记录并防止并发。

终止较大交易，重新提交新交易来提交子交易。这种方法的正确性不取决于可串行性或隔离性，而是取决于事务序列对状态和输出的净影响。也就是说，在补偿之后，数据库本质上最终到达的位置是否相当于子事务从未执行时的位置？等价必须包括外化的动作；例如，对重复购买进行退款与一开始就不向该客户收取费用几乎没有什么不同，但可以说是等效的。同样的想法也适用于分区恢复服务或产品提供商不能总是直接纠正错误，但它的目标是承认错误并采取新的补偿行动。如何最好地将这些想法应用于分区恢复是一个悬而未决的问题。“自动柜员机中的补偿问题”侧栏仅描述了一个应用领域中的一些问题。

5.4.4 Compensation issues in an automated teller machine

在自动柜员机 (ATM) 的设计中，强一致性似乎是合理的选择，但在实践中，A 胜过 C。原因很简单：更高的可用性意味着更高的收入。无论如何，ATM 设计可以作为审查分区期间补偿不变违规所涉及的一些挑战的良好背景。

ATM 的基本操作是存款、取款和查询余额。关键的不变量是余额应该为零或更高。因为只有取款可以忽略不变量，所以需要特殊处理，但其他两个操作总是可以执行。

ATM 系统设计者可以选择在分区期间禁止取款，因为不可能知道当时的真实余额，但这会损害可用性。相反，现代 ATM 使用备用模式（分区模式）将净提款限制为最多 k，其中 k 可能是 200 美元。低于这个限度，提款完全有效；当余额达到限额时，系统拒绝提款。因此，ATM 选择了复杂的可用性限制，允许取款但限制风险。

当分区结束时，必须有某种方法来恢复一致性并补偿分区期间所犯的错误。系统已分区。恢复状态很容易，因为操作是可交换的，但补偿可以采取多种形式。最终余额低于零违反了不变量。正常情况下，ATM 机取款，导致错误变成外部错误。银行通过收取费用并期望还款来进行补偿。鉴于风险是有限的，问题并不严重。然而，假设在分区期间的某个时刻余额低于零（ATM 不知道），但后来的存款将其恢复。在这种情况下，银行可能仍会追溯收取透支费，或者可能会忽略违规行为，因为客户已经支付了必要的付款。

一般来说，由于通信延迟，银行系统不依赖于一致性来保证正确性，而是依赖于审计和补偿。另一个例子是“支票风筝”，即客户在能够沟通之前从多个分支机构提取资金，然后逃跑。透支行为将在稍后被发现，或许会导致以法律诉讼的形式获得赔偿。

6 Example

满足一致性和有效性的数据库: Google Cloud Firestore, SAP MaxDB, MemSQL, SAP IQ (Sybase IQ), VoltDB, Teradata, Google Cloud Datastore, Microsoft Azure Cosmos DB, Amazon RDS (Relational Database Service), IBM Cloudant, Oracle NoSQL Database, ArangoDB

满足一致性和分区容错性的数据库: MySQL, MongoDB, Oracle Database, Microsoft SQL Server, PostgreSQL, HBase, Aerospike, Google Cloud Spanner, RDBMS, Neo4j, SAP HANA, CockroachDB, IBM Db2, Hazelcast, TiDB, VoltDB

满足可用性和有效性的数据库: Cassandra, DynamoDB, Couchbase, ScyllaDB, RavenDB, RethinkDB, Riak, MongoDB, Redis。

现实使用场景:

1. Social Media Feed (社交媒体动态): 想象一下一个社交媒体平台，其中用户的帖子和评论需要在所有用户的设备上一致地显示。然而，即使出现一些网络分区，用户仍然可以与平台进行交互。在这里，同时实现一致性和可用性可能比分区容错性更重要。

2. E-commerce Inventory （电子商务应用）: 在电子商务应用中，保持不同地区一致的库存水平至关重要。如果客户下订单，则库存必须一致减少。然而，在网络分区期间，系统可能会专注于维护可用性，同时确保最终的一致性。

7 局限

在 CAP 理论中，C 指的线性一致性：系统表现的就像只有一个副本一样。传统对 CAP 理论的一致性介绍都颇为片面：1. 在分布式系统完成某写操作后的任何读操作，都应该获取到该写操作写入的那个最新的值，这只是线性一致性的特例 2. 保持所有节点在同一个时刻具有相同的、逻辑一致的数据，有点儿以偏概全

分布式事务处理的并不是同一个数据对象的多个副本的问题，而指的是将针对多个数据对象的各种操作组合起来，提供 ACID 的特性。将分布式事务看成是强一致性的保证，猜测可能实际上指的就是 ACID 的原子性。总之，「强一致性」这个词很容易产生误解，所以建议谨慎使用。

除此之外，当国内工程师对 CAP 痴迷的时候，国外的工程师和研究者对 CAP 提出了各种质疑，纷纷有用反例证明着 CAP 在各种场合不适用性，同时挑战着 Lynch 的证明结果。

纵观这些质疑，基本都是拿着一个非常具体的系统，用 CAP 的理论去套，最后发现要么 CAP 不能 Cover 所有的场景，要么是 CAP 的定义非常模糊，导致自相矛盾！一句话，把 CAP 接地气是非常困难的。

CAP 没有考虑不同的基础架构、不同的应用场景、不同的网络基础和用户需求，而

C、A、P 在这些不同场景中的含义可能完全不同，这种无视差异化的定义导致了非常大的概念模糊，同时也变成 CAP 被质疑的源头。

面对大量的质疑，Brewer 和 Lynch 终于坐不住了，因此两人纷纷出来澄清：

1. Brewer 于 2012 年重申："3 个中的 2 个"这个表述是不准确的，在某些分区极少发生的情况下，三者能顺畅地在一起配合。CAP 不仅仅是发生在整个系统中，可能是发生在某个子系统或系统的某个阶段。该声明并不否认像质疑 3 那种三个因素协同工作的情况，并把 CAP 应用在一些更细粒度的场景中。

2. Lynch 也在 10 年后的 2012 年重写了论文，该论文主要做了几件事：把 CAP 理论的证明局限在原子读写的场景，并申明不支持数据库事务之类的场景。一致性场景不会引入用户 agent，只是发生在后台集群之内。把分区容错归结为一个对网络环境的陈述，而非之前一个独立条件。这实际上就是更加明确了概念。引入了活性 (liveness) 和安全属性 (safety)，在一个更抽象的概念下研究分布式系统，并认为 CAP 是活性与安全熟悉之间权衡的一个特例。其中的一致性属于 liveness，可用性属于 safety。把 CAP 的研究推到一个更广阔的空间：网络存在同步、部分同步；一致性性的结果也从仅存在一个到存在 N 个（部分一致）；引入了通信周期 round，并引用了其他论文，给出了为了保证 N 个一致性结果，至少需要通信的 round 数。

其实 Lynch 的论文主要就是两件事：缩小 CAP 适用的定义，消除质疑的场景；展示了 CAP 在非单一一致性结果下的广阔的研究结果，并顺便暗示 CAP 定理依旧正确。

7.1 CAP 的不足

CAP 定理本身是没有考虑网络延迟的问题的，它认为一致性是立即生效的，但是，要保持一致性，是需要时间成本的，这就导致往往分布式系统多选择 AP 方式由于时代的演变，CAP 定理在针对所有分布式系统的时候，出现了一些力不从心的情况，导致很多时候它自己会把以前很严谨的数学定义改成了比较松弛的业务定义，类似于我们看到，CAP 定理把一致性、可用性、分区容错都变成了一个范围属性，而这和 CAP 定理本身这种数学定理般的称呼是有冲突的，出现了不符合数学严谨定义的问题。在实践中以及后来 CAP 定理的提出者也承认，一致性和可用性并不仅仅是二选一的问题，只是一些重要性的区别，当强调一致性的时候，并不表示可用性是完全不可用的状态。比如，Zookeeper 只是在 master 出现问题的时候，才可能出现几十秒的不可用状态，而别的时候，都会以各种方式保证系统的可用性。而强调可用性的时候，也往往会采用一些技术手段，去保证数据最终是一致的。CAP 定理并没有给出这些情况的具体描述。CAP 理论从工程角度来看只是一种状态的描述，它告诉大家当有错的时候，分布式系统可能处在什么状态。但是，状态是可能变化的。状态间如何转换，如何修补，如何恢复是没有提供方向的。

8 Conclusion

当分区存在时，系统设计者不应盲目牺牲一致性或可用性。使用所提出的方法，他们可以通过在分区期间仔细管理不变量来优化这两个属性。作为较新的技术，例如版本向量和 CRDT，进入简化其使用的框架，这种优化应该变得更加广泛。然而，与 ACID 事务不同，这种方法需要相对于过去的策略进行更周到的部署，并且最佳解决方案将在很大程度上取决于有关服务的不变量和操作的详细信息。

首先肯定的是，CAP 并不适合再作为一个适应任何场景的定理，它的正确性更加适合基于原子读写的 NoSQL 场景。质疑虽然很多，但很多质疑者只是偷换概念，并没有解决各个因素之间的取舍问题。而无论如何 C、A、P 这个三个概念始终存在任何分布式系统，只是不同的模型会对其有不同的呈现，可能某些场景对三者之间的关系敏感，而另一些不敏感。

就像 Lynch 所说，现在分布式系统有很多特性，比如扩展性、优雅降级等，随着时间的发展，或许这些也将被纳入研究范畴。而作为开发者，我们需要考虑的问题，不仅仅是 CAP 三者。

9 CAP 理论面试题

什么是 CAP 理论?
CAP 中的 P 是什么意思?
为什么说分布式系统，只能在 C、A 中二选一?
结合实际应用，CP、AP 该怎么选择？[3]

参考文献

[1] Eric Brewer. Cap twelve years later: How the ”rules” have changed. *Computer*, 45(2):23–29, 2012.

[2] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[3] 用户 7686797. Tp toolbox. `https://cloud.tencent.com/developer/article/1860632`.