中山大學

分布式计算理论课作业

习题一

姓　　名　张瑞程

学　　号　22354189

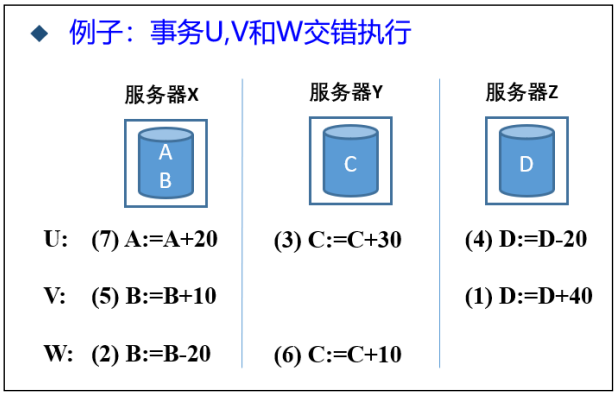
院　　系　智能工程学院

专　　业　智能科学与技术

指导教师　余晨韵

2024 年 11 月

1. knapp 将分布式死锁检测分为 4 类，请简要介绍。请结合下图中的例子，说明边追逐法、扩散计算如何检测分布式系统中的死锁。



（1）分布式死锁检测的四种方法：

① 路径推动算法的基本思想是在分布式系统的每个节点（或称为站点）上构建一个简化形式的全局等待图（Wait-for Graph，WFG）。当一个节点执行死锁计算时，它会将其局部 WFG 发送给所有相邻的节点。这些局部 WFG 在相邻节点被接收并更新后，会再次被传递到更多的节点。这个过程会不断重复，直到某个节点获得了足够的信息来构建一个完整的全局 WFG，从而可以宣布死锁的存在或者确定系统中不存在死锁。

执行过程

1. 初始化：每个节点开始时都只有关于自己本地的等待关系的信息。
2. 发送局部 WFG：节点将自己的局部 WFG 发送给邻居节点。
3. 更新和传递：邻居节点接收到 WFG 后，会将其与自己的 WFG 合并，并更新本地 WFG。然后，这个更新后的 WFG 会被继续传递给更多的节点。
4. 重复过程：上述发送和更新的过程不断重复，直到有足够的信息来确定全局状态。
5. 死锁检测：当一个节点获得了足够的信息后，它将能够确定是否能够构建出一个完整的 WFG，从而检测出是否存在死锁。

②全局状态检测算法在不需要暂停系统的正常运行的情况下通过局部快照确定一致的全局状态，无需暂停当前的计算来确定一个全局等待图。

该算法把分布式系统简化成一个有向图，进程是有向图的节点，通信信道看成是图的有向边（包括输入通道和输出通道），要求通道中的消息保持有序。而局部快照是指每个进程的当前状态和它的输入通道中有序的消息。

执行过程

1. 由系统的任意进程**发起一个快照过程**，记录该进程记录当前状态，并将该信道标记为空（视为已处理）；生成一个特殊的标记消息（marker），通过所有输出信道将 marker 消息发送给其他进程，并开始监听所有流入信道的消息。
2. 其他进程在接收到 marker 消息时：若没有记录过当前状态，则记录，并重复发起快照的过程，并开始记录其他输入通道中的消息；若已经记录过当前的状态，则记录其他输入通道在收到 marker 消息之前收到的消息。

3. 进程的快照在收到所有输入通道的消息后才保存。所有进程完成快照时， 快照结束，**全局状态**确定。然后结合每个进程保存的消息，确定等待图。

③边追逐算法基于这样一个理念：在分布式系统中，每个进程可以看作是一个节点，进程间的资源请求和释放可以看作是有向边。如果存在一个循环，即表示系统中存在死锁。

执行过程

1. 算法使用特殊**探针**（token）来追踪路径，当一个节点需要资源时，它发送一个探针沿着请求的路径向前传播。
2. 探针沿着有向边从一个节点**传递**到另一个节点。如果一个节点接收到自己发出的探针，它就知道自己在一个**循环**上，从而检测到死锁。如果探针到达一个可以释放资源的节点，该节点会分配资源，并让探针沿着原路返回。
3. 当所有探针都返回到起始节点，或者所有资源都被成功分配，算法结束。

④ 扩散计算通过在 Wait-for Graph（WFG）上进行查询的扩散和收缩来检测死锁。

执行过程

1.扩散阶段：当一个 WFG 中的某个节点怀疑自己处于死锁状态时，它会启动扩散计算。事务管理器负责发起扩散进程，向依赖于它的进程（即等待该事务释放资源的进程）发送查询。该节点从原来的 neutral（中立）状态变为 active（活跃）状态，并向其子节点发送 query（查询），这些查询会沿着 WFG 进行扩散。每个节点收到 first query（也称为 engaging query）后激发为 active 状态，激发后的节点将继续 沿着其子节点发送 query。

2.收缩阶段：每个被 engaging query 激活的节点并不会直接响应这个查询，但会响应除了 engaging query 以外的 query。任何一个节点，需要收到其所有子节点的响应，才能响应最开始激活它的 engaging query，若无子节点，则永远不响应；当节点响应了 engaging query 后，恢复原来的 neutral 状态，视作扩散进程收缩。

3.死锁检测：如果在扩散和收缩过程中，根节点（最开始发起扩散的节点）收到了所有子节点的响应并且恢复为 neutral 状态，那么可以确认系统中存在死锁。

4.死锁解除：一旦检测出死锁，就需要解除。通常的做法是放弃环路中的某个事务，以打破死锁循环。

（2）边追逐算法的举例说明

1. 事务和资源分布

事务 U 在服务器 X 上执行，需要访问资源 A 和 B。

事务 V 在服务器 Y 上执行，需要访问资源 B 和 C。

事务 W 在服务器 Z 上执行，需要访问资源 C 和 A。

2. 死锁检测过程

死锁检测过程从服务器 X 开始，因为事务 U 需要资源 B，而资源 B 位于服务器 Y。

步骤 1：服务器 X 发起死锁检测：服务器 X 发起死锁检测过程，向对象 B 的服

务器 Y 发送探寻消息<W→U>。这意味着事务 U（在服务器 X 上）正在等待资源 B（在服务器 Y 上），所以探寻消息包含了事务 U 和它等待的资源 B。

步骤 2：服务器 Y 处理探寻消息：服务器 Y 收到探寻消息<W→U>后，发现对象 B 当前被事务 V 拥有。因此，服务器 Y 将事务 V 附加到探寻消息上，形成新的探寻消息<W→U→V>。这表示事务 U 等待事务 V 释放资源 B。

步骤 3：探寻消息转发到服务器 Z：由于事务 V 在服务器 Z 上等待对象 C，探寻消息<W→U→V>被转发到服务器 Z。

步骤 4：服务器 Z 检测到死锁：服务器 Z 收到探寻消息<W→U→V>，并且发现资源 C 被事务 W 拥有。因此，服务器 Z 将事务 W 附加到探寻消息后，形成<W→U→V→W>。这个路径包含了一个环路：W 等待 U 释放资源 A，U 等待 V 释放资源 B，V 等待 W 释放资源 C。这个环路表明存在死锁。

3. 死锁解除

一旦检测到死锁，系统需要采取措施来解除死锁。通常的做法是选择一个或多个事务进行回滚（放弃），以打破等待循环。根据事务的优先级或其他策略来决定哪个事务被放弃。

（3）扩散计算算法的举例说明

1. 事务和资源分布

事务 U 在服务器 X 上执行，需要访问资源 A 和 B。

事务 V 在服务器 Y 上执行，需要访问资源 B 和 C。

事务 W 在服务器 Z 上执行，需要访问资源 C 和 A。

2. **扩散阶段**

步骤 1：怀疑死锁，激活扩散计算：

事务 U 怀疑自己处于死锁状态，因此它激活扩散计算，从服务器 X 向依赖于它的进程（即等待它释放资源的进程）发送查询请求。此时，事务 U 的状态变为 active。

步骤 2：U 向 V 发送查询请求，V 被激活：

事务 U 向事务 V 发送查询请求（engaging query），因为事务 U 等待事务 V 释放资源 B。事务 V 收到查询请求后，被激活（变为 active 状态）。

步骤 3：V 向 W 发送查询请求，W 被激活：

事务 V 接着向事务 W 发送查询请求，因为它等待事务 W 释放资源 C。事务 W 收到查询请求后，也被激活（变为 active 状态）。

3. **收缩阶段**

步骤 4：W 向 U 发送查询请求，U 回复请求：

事务 W 向事务 U 发送查询请求，因为事务 W 等待事务 U 释放资源 A。由于事务 U 已经被激活，它直接回复这个查询请求。

步骤 5：W 收到所有子节点（U）的回复，激活状态消失（neutral），向父节点（V）回复请求：

事务 W 收到事务 U 的回复后，它已经收集了所有子节点的回复（在这

个例子中，W 只有一个子节点 U）。因此，事务 W 的激活状态消失，变为 neutral，并回复其父节点事务 V。

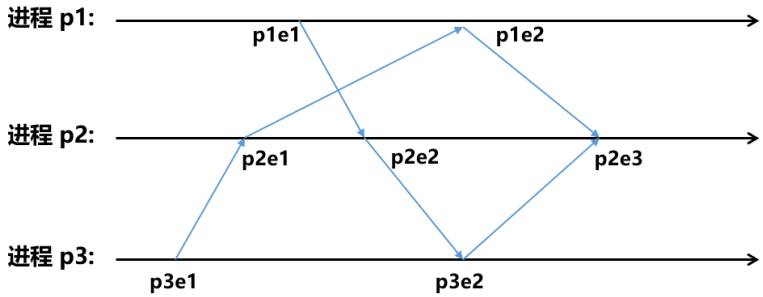
步骤 6：V 同理，激活状态消失，向 U 回复请求：

事务 V 收到事务 W 的回复后，也收集了所有子节点的回复（V 也只有一个子节点 W）。事务 V 的激活状态消失，变为 neutral，并回复其父节点事务 U。

步骤 7：U 收到所有子节点回复，激活状态消失，扩散进程终止，宣布出现死锁：

事务 U 收到事务 V 的回复后，收集了所有子节点的回复（U 也只有一个子节点 V）。事务 U 的激活状态消失，变为 neutral。此时，扩散计算进程终止，宣布系统中出现死锁。

2. 什么是向量时钟算法，有何优缺点？相比于向量时钟算法，矩阵时钟算法有何优势？采用矩阵时钟算法，给出下图各个节点的状态矩阵。（参考 ppt 第三章 page15）



答：

（1）什么是向量时钟算法

向量时钟算法是一种用于分布式系统中的逻辑时钟机制，用于确定事件的因果关系和部分顺序。每个进程维护一个向量时钟，这是一个向量，其中的每个分量对应于系统中的一个进程，记录了该进程生成的最大逻辑时钟值。

工作原理：

1. 所有时钟分量**初始化为 0**。
2. 进程在处理本地事件时，将自己的**逻辑时钟加 1**。
3. 发送消息时，进程将自己的向量时钟作为消息的一部分发送给接收者。
4. 接收进程收到消息后，将**自己的逻辑时钟加 1**，并更新自己的向量时钟，将每个分量设置为本地分量和接收到的向量时钟对应分量的**最大值**。

（2）向量时钟算法的优缺点

优点：

1. **不需要全局时钟**和维护节点上的版本数，适用于分布式系统。
2. **可以直接判断两个事件的因果关系**，如果一个事件的向量时钟小于另一个事件的向量时钟，则前者在因果关系上先于后者。
3. 提高了系统的**容错性**，因为每个进程**独立维护自己的局部时钟**。

缺点：

1. 每个进程需要存储和传输一个向量，随着系统规模的扩大，**开销增加迅速**。

2. 对于**没有直接因果关系**的事件，向量时钟可能无法准确判断它们的顺序，存在歧义。

（3）矩阵时钟算法及其优势

矩阵时钟算法是在向量时钟的基础上发展起来的，它为每个进程增加了一个矩阵来记录更多的时间信息。

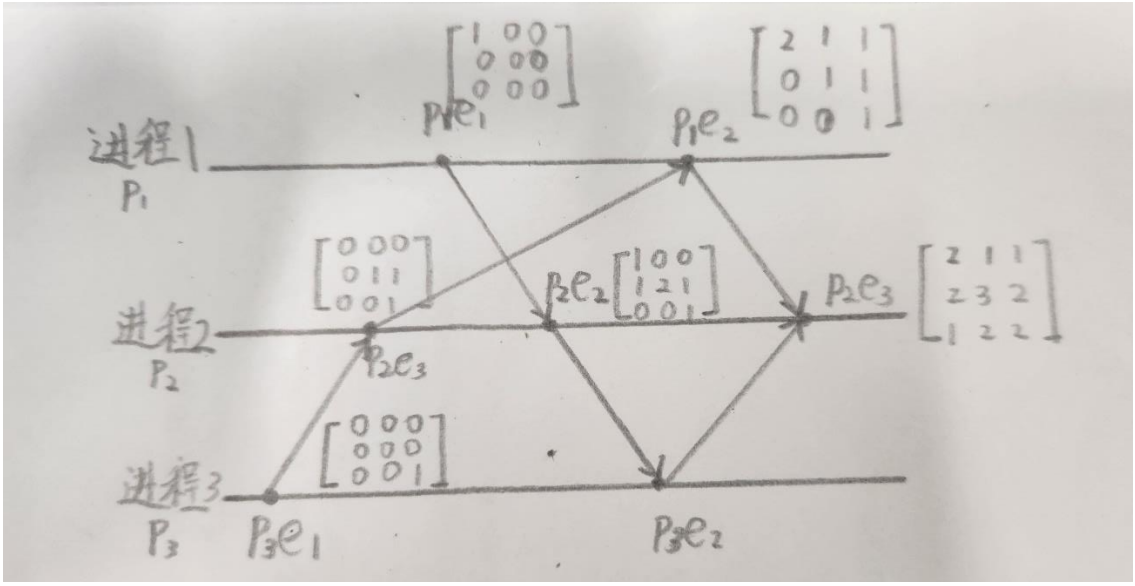
工作原理：

1. 每个进程维护一个 n × n 的**矩阵**，其中 n 是系统中进程的数量。矩阵的每行代表一个进程的视图，每列代表一个进程。
2. 当新事件发生时，进程将自己的**对应行**的**对角线元素增加**。
3. 发送消息时，进程将矩阵时间戳**附带**在消息中。
4. 接收进程收到消息后，**更新**自己的矩阵时间戳，对于每个元素，取本地矩阵和接收到的矩阵对应元素的**最大值**。

优势：

1. **更详细的时间信息**：矩阵时钟提供了比向量时钟更详细的时间信息，因为它记录了每个进程对其他进程时间的了解。
2. **过时消息丢弃**：矩阵时钟可以识别并丢弃过时的消息，减少不必要的处理和存储开销。
3. **更好的因果关系判断**：由于包含了更多信息，矩阵时钟在判断事件的因果关系时更为准确。

（4）各个节点的状态矩阵



3. 一个服务器管理对象 a1, a2,…, an.该服务器为客户提供两种操作：

- Read (i) ：返回对象 ai 的值。
- Write (i, value)：将对象 ai 的值设置为 value。

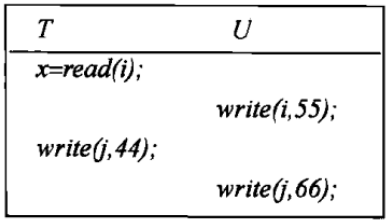
事务 T 和 U 定义如下：

T：y=read (k); x=read (i); write (j, 44);

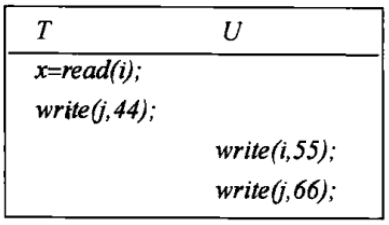
U: write (i, 55); write (j, 66);

对象 ai 和 aj 的初始值分别为 10 和 20。

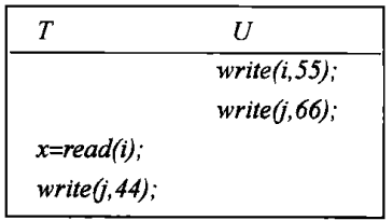
(1) 下面有 4 种执行情况，哪些是串行等价的？写出最终的 ai 和 aj 结果。



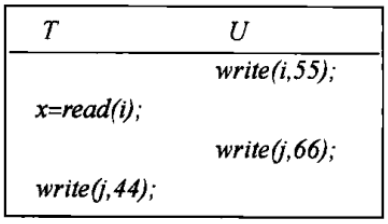
a）



b）



c）



d）

答：

串行等价是指一组事务顺序执行的结果与他们实际并发执行的结果相同。

经计算可得出：

（a）x=10，ai=55，aj=66

（b）x=10，ai=55，aj=66

（c）x=55，ai=55，aj=44

（d）x=55，ai=55，aj=44

(a),(b),(c),(d)均是串行等价的：

(a)并发执行 = 先执行 T 后执行 U;

(b)并发执行 = 先执行 T 后执行 U;

(c)并发执行 = 先执行 U 后执行 T;

(d)并发执行 = 先执行 U 后执行 T。

(2) 考虑事务 U 和 T 交错执行（同时处于活动状态），在使用具有向后验证的并发控制时，描述事务 T 和 U 的操作顺序和执行效果（根据读规则或者写规则，操作是否允许执行，验证是否通过，以及 ai 和 aj 的最终结果）。

T	*U*
OpenTransaction	*OpenTransaction*
y = read(k);	
	write(i, 55);
	write(j, 66);
	commit
x = read(i);	
write(j, 44);	

答：

（1）操作顺序：**先验证 U，再验证 T**。因为 U 比 T 先提交，因此先进入验证。

（2）执行结果：事**务 U 的验证可以通过，事务 T 验证不通过**（事务 T 不能读取事务 U 写的对象，即事务 T 对对象 i 的读操作与事务 U 的对 i 的写操作发生了冲突。这意味着事务 T 在向后验证阶段检测到了与事务 U 的冲突，违反了所选的并发控制规则。因此，事务 T 的向后验证失败）。最终 **ai = 55，aj = 66**。

4. 分布式互斥与分布式事务有何联系？实现分布式事务有哪些方法？如何设计容错的三阶段提交：考虑节点在各个状态发生故障时，如何进行状态恢复？

答：

（1）分布式互斥与分布式事务的联系

分布式互斥和分布式事务都是为了解决分布式系统中数据的一致性问题，但他们的侧重解决的问题不同：

1. **分布式互斥：**

   目的：确保在分布式系统中对共享资源的访问是互斥的，即在同一时间只有一个进程可以访问特定的资源。

   侧重点：侧重对资源的使用，解决多个进程对同一共享资源的并发访问操作的一致性问题，防止资源竞争和冲突。

2. **分布式事务：**

   目的：确保事务的原子性、一致性、隔离性和持久性（ACID 属性），即使事务的操作分布在不同的节点上。

   侧重点：侧重的是操作步骤之间的协作，确保事务中的所有操作要么全部成功，要么全部失败，保持事务的整体性和一致性。

（2）实现分布式事务的方法

1. **二阶段提交协议（2PC）：**

   这是一种同步处理方法，用于确保所有参与事务的节点要么都提交事务，要么都不提交。

   过程：

   准备阶段：协调者询问所有参与者是否准备好提交事务。

   提交阶段：如果所有参与者都准备好了，协调者指示所有参与者提交事务；否则，它指示它们回滚事务。

   优点：简单，易于理解。

   缺点：在准备阶段，所有节点必须等待，可能导致性能瓶颈。

2. **三阶段提交协议（3PC）：**

   这是二阶段提交协议的改进版，增加了一个额外的阶段来减少阻塞。

   过程：

   询问阶段：协调者询问参与者是否准备好提交事务。

   准备阶段：参与者准备提交事务，但不真正提交。（比二段提交协议多

的阶段）

提交阶段：协调者根据参与者的准备情况决定是提交还是回滚事务。

优点：减少了参与者在提交阶段的阻塞时间。

缺点：实现复杂，可能遇到网络分区时的不确定性。

3. **基于消息的最终一致性方法**：

这种方法通过引入消息中间件（如 Kafka、RabbitMQ 等）来实现事务的异步处理。

过程：

消息发送：事务的每个步骤被封装成消息发送到消息队列。

消息处理：消息消费者根据消息内容执行相应的操作。

最终一致性：系统保证所有消息最终都会被处理，但可能不是立即。

优点：提高了系统的可扩展性和响应性，因为操作是异步执行的。

缺点：需要额外的消息中间件，增加了系统的复杂性，且不能保证即时的一致性。

（3）如何设计容错的三阶段提交

1.在协调者和参与者中引入**超时机制**，为每个阶段设置超时时间，如果参与者或协调者在超时时间内未收到消息，则根据当前状态和协议规则采取行动。

2.在第一、二阶段之间加入**准备阶段**，并在在每个阶段结束时，发送状态确认消息，以确保所有节点都同意下一步的行动。

具体方案如下：

Can-Commit 阶段

协调者故障：如果协调者在发送 Vote-request 后故障，参与者将等待超时后决定是否继续等待或中止事务。

参与者故障：如果参与者在接收 Vote-request 后故障，恢复后应向协调者查询当前事务状态，并根据协调者的指示行动。

Pre-Commit 阶段

协调者故障：协调者在发送 Prepare-commit 后故障，参与者应等待超时。如果收到 Global-abort，则中止事务；如果超时未收到任何消息，参与者应根据持久化日志和其他参与者的反馈决定是提交还是中止事务。

参与者故障：参与者在接收 Prepare-commit 后故障，恢复后应检查持久化日志，并向其他参与者查询以确定是否已经准备好提交。

Do-Commit 阶段

协调者故障：协调者在发送 Global-commit 或 Global-abort 后故障，参与者应等待超时。如果收到 Global-abort，则中止事务；如果超时未收到任何消息，参与者应根据持久化日志和其他参与者的反馈决定是否已经提交。

参与者故障：参与者在接收 Global-commit 或 Global-abort 后故障，恢复后应检查持久化日志，并根据协调者的最终决定执行提交或中止操作。

3. 状态恢复策略

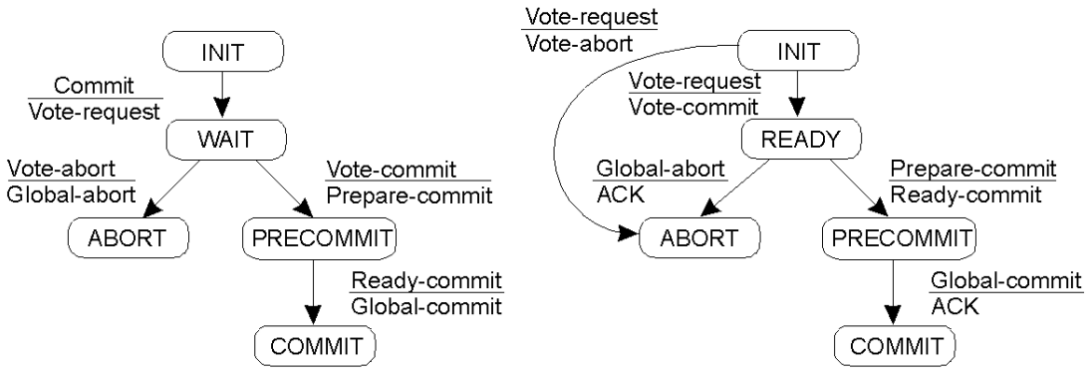
状态查询：参与者在恢复后，应向协调者和其他参与者查询当前事务状态。

日志分析：参与者应分析持久化日志，以确定在故障前的状态。

多数决定：在不确定的情况下，参与者可以根据多数参与者的意见来决定是提交还是中止事务。

安全默认：如果无法确定状态，应采取安全的默认操作，通常是中止事务，以避免数据不一致。

下图：有容错的三阶段提交有限状态机



协调者（左） **参与者（右）**

5. CAP 和 ACID 中的 C 和 A 是一样的吗？CAP 理论与 BASE 理论的区别是什么？发布订阅和消息队列模式都支持系统解耦，两者是否一致呢，为什么？

答:

（1）CAP 和 ACID 中的 C 和 A 是一样的吗？

都不一样!

CAP 原则中的 C（一致性）：强调在分布式系统中，不同节点间的数据副本能够保持同步，即在任何时刻，所有节点上的数据有一致性。这通常通过节点间的数据复制技术实现。**比如**，假设有一个在线购物平台，该平台在多个数据中心运行，每个数据中心都存储有商品库存信息的副本。用户想要购买一件商品，该商品在所有数据中心的库存信息中都显示为有货。且用户下单后，订单处理系统会更新所有数据中心的库存信息，减少相应商品的库存数量。

ACID 属性中的 C（一致性）：关注的是事务执行前后数据的完整性和一致性。事务必须确保数据库从一个一致性状态转换到另一个一致性状态，同时满足所有预定义的完整性约束和规则。**比如**，一个支付系统，在转账前后双方的余额之和应该相等，且买家被扣的钱要等于卖家收到的钱。

CAP 原则中的 A（可用性）：关注的是系统在面对网络分区或其他故障时，是否能够继续提供服务而不出现整体性宕机。换句话说，即使系统的一部分出现问题，系统仍能对用户的请求做出响应。**比如**，一个新闻网站，在高流量事件（如重大新闻爆发）期间，网站的访问量激增。为了保证可用性，网站可能会采取一些措施，比如提供简化版的网页内容，减少图片和视频的加载，或者实施缓存策略，以确保用户

仍然可以访问新闻内容，尽管可能不是完整的用户体验。

ACID 属性中的 A（原子性）：它强调事务的处理是“全有或全无”的，即事务中的所有操作要么全部成功执行，要么全部不执行。这意味着如果事务中的某个操作失败，整个事务将回滚到操作之前的状态，确保数据状态的一致性。**比如**，淘宝的购物车结算过程，系统会执行多个操作，包括扣减库存、生成订单、扣款等。要么全部成功，要么系统必须能够回滚之前的所有操作，包括恢复库存到原始状态。

（2）CAP 理论与 BASE 理论的区别是什么？

CAP 理论（**一致性、可用性、分区容错性**）和 BASE 理论（**基本可用、柔性状态、最终一致性**）代表了在分布式系统中处理一致性、可用性和分区容错性的不同方法和权衡。CAP 理论和 BASE 理论反映了分布式系统设计中的两种不同思路：一种是理论上的*不可能性结果*，另一种是*实践中的可用性和一致性权衡*。

CAP 理论

CAP 理论指出，在分布式数据存储系统中，不可能同时满足以下三个特性（最多满足其中 2 个）：

1. **一致性（C：Consistency）**：确保在分布式系统中的所有数据副本始终保持同步；在任何时刻，对系统的任何节点进行读操作，都能得到最新的数据状态。
2. **可用性（A：Availability）**：系统在任何时候都能响应用户的请求，即使在出现故障的情况下也能保证基本的服务。
3. **分区容错性（P：Partition tolerance）**：系统能够容忍网络分区，即当网络发生故障导致系统被分割成多个分区时，系统仍然能够继续运行。

满足 CA，适用于单机系统，不涉及网络分区的问题；

满足 CP，适用于要求数据强一致性的场景；

满足 AP，适用于要求及时响应用户，但对数据一致性要求较低的场景。

BASE 理论

BASE 理论是对 CAP 理论中一致性和可用性权衡的一种实践，它包含以下三个核心原则：

1. **基本可用（BA：Basically Available）**：

系统出现故障时，保证核心功能可用，允许部分不可用。

例如，在高流量或高负载情况下，系统可能会降级服务，提供简化的响应。

2. **柔性状态（S：Soft state）**：

系统的状态可以有一定的灵活性，不需要时刻保持强一致性。

允许系统中的数据副本在一段时间内是不一致的，但最终会达到一致状态。

3. **最终一致性（E：Eventual consistency）**：

系统不保证立即的一致性，但保证经过一段时间后，所有副本最终会达到一致状态。

这种模型适用于对实时性要求不高，但需要最终数据一致性的场景。

（3）发布订阅和消息队列模式都支持系统解耦，两者是否一致呢，为什么？

发布订阅（Pub-Sub）模式和消息队列（Message Queue）模式都是消息传递范式，它们都支持系统解耦，但它们在数据结构、解耦方式和使用场景上存在差异：

发布订阅模式

数据结构：发布订阅模式采用了消息中心来存储生产发布的数据，有主题、broker 等概念。采用了 map 或者数组等方式存储

解耦方式：在发布订阅模式中，消费者需要提前向消息中心订阅自己感兴趣的数据，待生产者发布数据到消息中心后，消息中心需要根据订阅信息推送数据给消费者。此模式下，生产者和消费者之间没有直接的通信关系，由消息代理负责消息的路由。

适用场景：发布订阅模式适合于消息广播的场景，其中单个生产者向多个消费者发布消息，且消费者数量可能随时变化。它也适用于解耦服务，尤其是当消费者需要实时响应发布的消息时。

消息队列模式

数据结构：消息队列模式使用消息队列中心来存储和管理消息。采用队列储存，并遵循先进先出（FIFO）的原则，有主题、broker 等概念。

解耦方式：在消息队列模式中，生产者将消息发送到队列后不需要关心消息何时被处理，同样消费者从队列中拉取消息进行处理而不需要知道消息的来源。这种模式下，生产者和消费者之间的通信是通过队列进行的，它们之间没有直接的联系。

适用场景：消息队列模式适合于需要可靠消息传递的场景，比如任务异步处理、命令的执行等。它也适用于处理负载均衡和流量削峰，因为消息队列可以缓冲突然的大量消息。

两者的区别

1. 消息存储方式不同：消息队列模式中的消息通常存**储在队列中**，直到被消费者处理；而发布订阅模式中的消息可能不存储，**直接传递**给当前的订阅者。
2. 消息传递方式不同：发布订阅模式依赖于**订阅关系**，消费者需要显式订阅主题；消息队列模式中，消费者通过**监听队列**来接收消息，不需要提前订阅。
3. 消费者处理方式不同：在发布订阅模式中，**所有订阅者都会接收到消息**；而在消息队列模式中，消息通常**由一个消费者处理后从队列中移除**。
4. 应用场景不同：消息队列模式适合消费者为**临时用户**的场景，因为它不需要提前知道消费者信息；发布订阅模式适合消费者为**常驻进程**的场景，因为需要提前建立订阅关系。