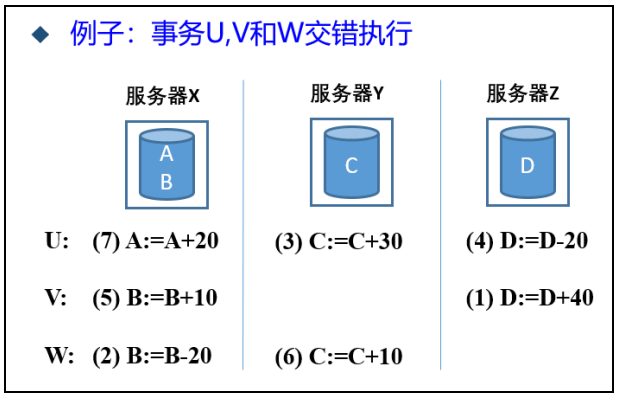
分布式计算-习题 1

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。



路径传递算法（集中式方案）：通过在每个节点间传递路径信息来检测死锁。当路径信息中检测到一个循环时，即表明存在死锁。

边追踪算法：通过在等待图的边上传递“探针”消息来检测死锁。当探针返回到起始点时，就检测到了一个死锁。

全局状态检测算法：这种算法通过获取系统全局状态的快照，并分析快照是否存在死锁。

扩散计算算法：在这种算法中，进程通过图传递计算，并根据响应来判断是否存在死锁。

结合例子说明

边追逐法：

服务器 Y 发起死锁检测过程，向对象 D 的服务器 Z 发送探寻消息<W→U>;

服务器 Z 收到探寻消息<W→U>后，发现对象 B 被事务 V 拥有，因此将 V 附加在探寻消息上，产生<W→U→V>.

由于 V 在服务器 X 上等待对象 B，因此该探寻消息被转发到服务器 X；

服务器 X 收到探寻消息<W→U→V>，并且发现 C 被事务 W 拥有，那么将 W 附加在探寻消息后形成<W→U→V→W>。该路径包含环路，检测出死锁。因此后续需要根据事务的优先级来选择放弃某个事务来解除死锁。

扩散计算：

W→U (对象 C) Z: U→V (对象 D) X: V→W (对象 B)

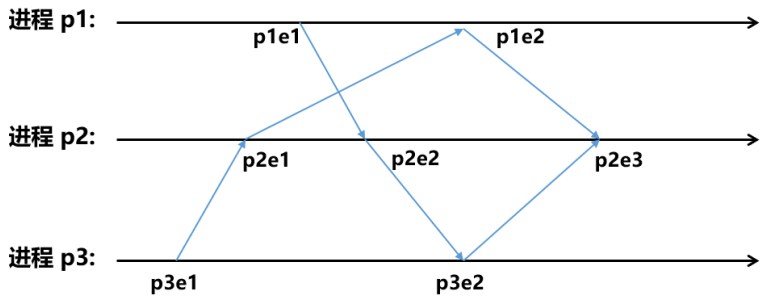
U 怀疑死锁，激活扩散计算；向 V 发送查询请求，V 被激活；

V 向 W 发送查询请求，W 被激活。

W 向 U 发送查询请求，因为 U 已经被激活，所以 U 回复请求。W 收到所有子节点（U）的回复，激活状态消失，向父节点（V）回复请求。V 同理，激活状态消失，向 U 回复请求。

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。

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）



什么是向量时钟算法，有何优缺点？

向量时钟是一种用于分布式系统中事件顺序跟踪的算法。每个节点维护一个向量，其中每个元素表示该节点对系统中所有节点事件的计数。向量时钟的更新规则如下：

1. 当节点 PiP_iPi 执行一个事件时，增加其向量的第 iii 个元素。
2. 当节点 PiP_iPi 接收到来自节点 PjP_jPj 的消息时，将自己向量时钟与消息中的向量逐元素取最大值更新。

优点：

- 向量时钟可以精确地表示事件的偏序关系，能够判断两个事件是否存在因果关系。能衡量同时发生的事件如何表示先后顺序。

缺点：

- 向量时钟在规模较大的分布式系统中，存储和传输的开销较大，因为每个进程都需要存储整个系统中所有进程的计数。
- 占用更多存储空间；若两个事件无因果联系，可能存在歧义，难以判断

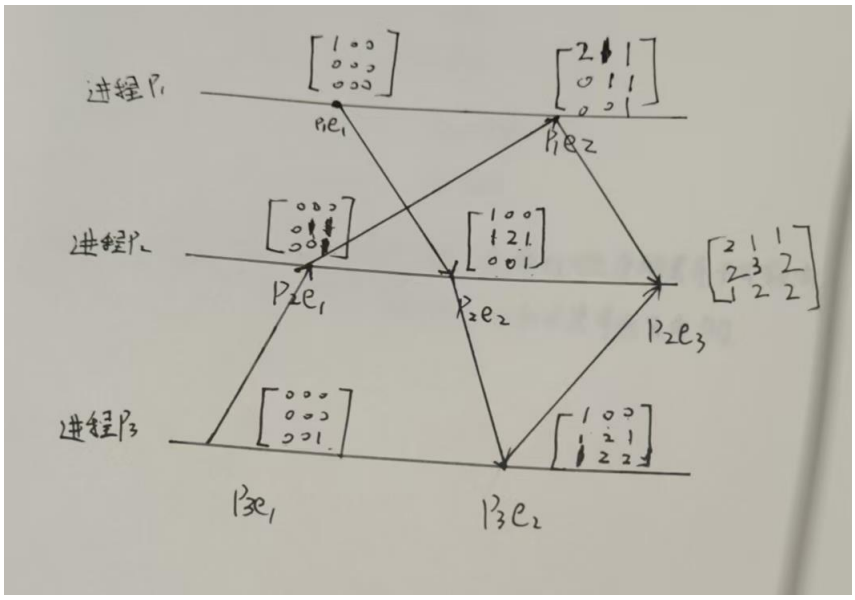
矩阵时钟算法有何优势？

（1）矩阵时钟在向量时间的基础上，添加了系统其他进程视图的信息。矩阵时钟除了包含向量时钟的所有特性之外，还有 $\min(mti[k,i]) \geq t$，也即，pi 知道进程 pk 对 pi 的时间了解已经到达 t 。其他进程不需要 pi 发送 t 之前的消息，以此丢弃过时的消息。

（2）矩阵时钟可以在较低的开销下提供分布式系统中的因果关系跟踪，因为它只需在消息传递时同步矩阵中的部分信息。

（3）矩阵时钟能够直接从矩阵中推断任意两个节点之间的时间戳关系。

（4）尽管空间开销更大，但在进程数量变化的情况下，矩阵时钟可以更好地适应系统动态。



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。

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

a）

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

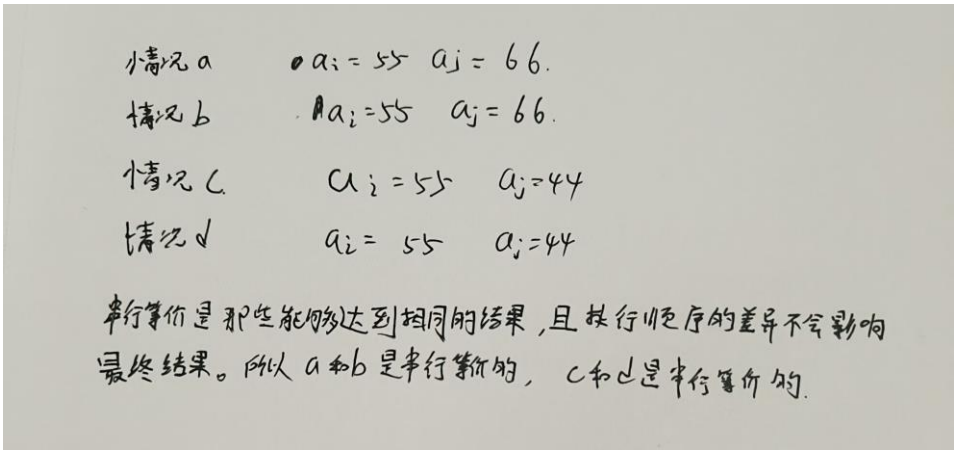
b）

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

c）

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

d）



(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);	

1. 事务 T:从 a_k 读取数据，赋值给 y；U 这个时候还没开始任何操作，允许执行
2. 事务 U:在 a_i 上执行写操作，将其值设为 55；T 未对 a_i 有任何操作，允许执行
3. 事务 U:在 a_j 上执行写操作，将其值设 66;T 未对 a_j 有任何操作，允许执行
4. 事务 U:U 提交，对 a_i 和 a_j 的修改写入数据库；
5. 事务 T:从 a_i 读取数据，赋值给 x；由于 U 已提交对 a_i 的修改，故读取的值为 55
6. 事务 T:在 a_j 上执行写操作，将其值设为 44；无法通过验证，因为它会破坏已经提交的 U 对 a_j 的更新。因此，系统回滚 T，确保 j 保持 U 提交的值 66，而 a_i 也保持为 55。

最终执行结果为：$a_i = 55, a_j = 66$

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？

一、分布式互斥与分布式事务有何联系？

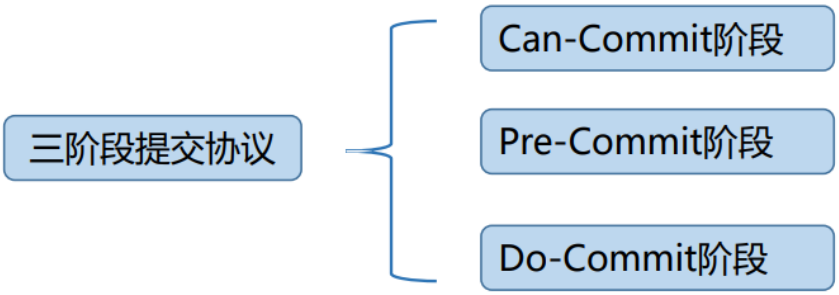
（1）在实现分布式事务时，经常需要对一些关键资源（如数据记录、文件）进行锁定，以防止并发访问导致数据不一致。这就涉及到分布式互斥。例如，一个事务可能需要锁定一条记录，直到事务完成或回滚，这就是分布式互斥的应用。

（2）分布式事务依赖分布式互斥来确保数据在并发环境中的一致性，而分布式互斥则为分布式事务提供了资源锁定的基础，以保证事务可以安全地执行。

二、实现分布式事务有哪些方法？

基于 XA 协议的两阶段提交、三阶段提交协议、基于日志和补偿的分布式事务、最终一致性和 BASE 模型

三、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？



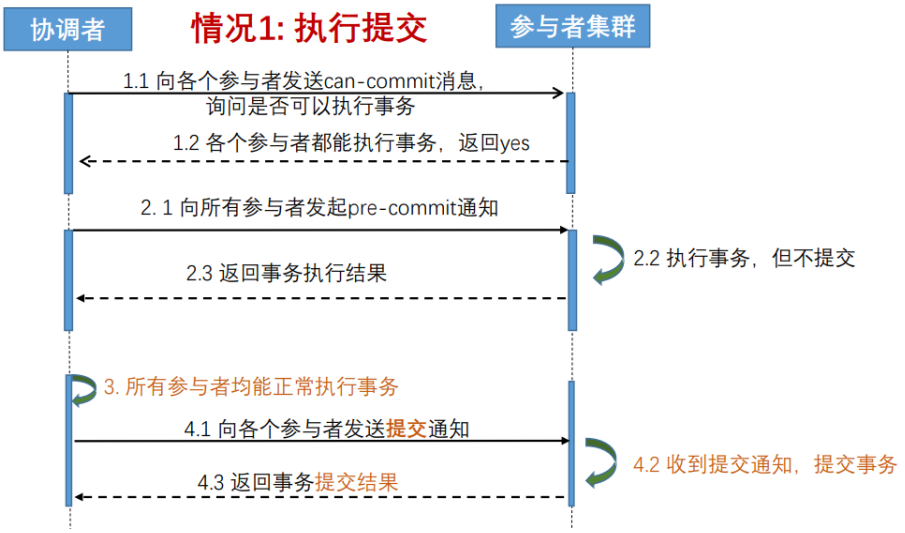
阶段一：Can-Commit ① 事务询问：协调者向参与者发送 Can-Commit 请求，询问是否 可以执行事务提交操作，然后等待响应； ② 响应反馈：参与者收到 Can-Commit 请求后，如果判断可以顺 利执行事务则回复 Yes 响应，否则回复“No” 。

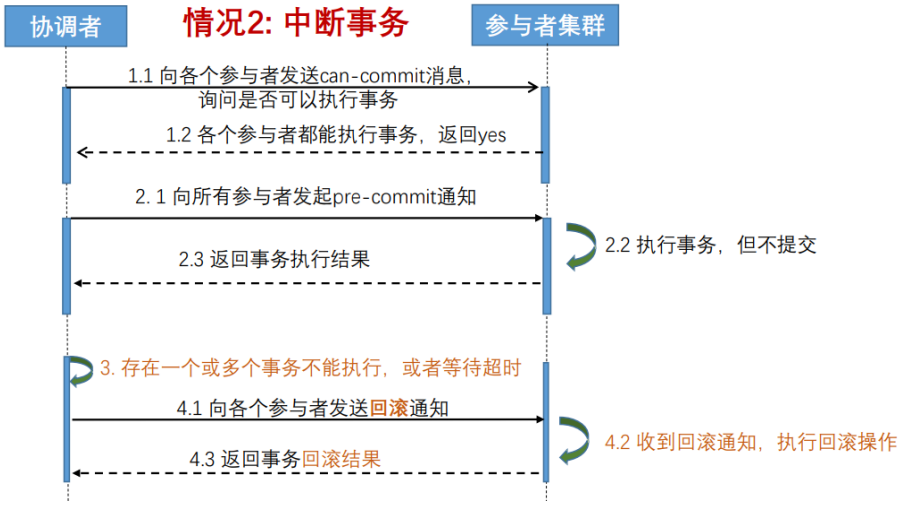
Pre-Commit 阶段（情况 1） 假如协调者从参与者处收到的响应皆为 Yes，则进行事务的预执行： ① 发送预提交请求：协调者向参与者发送 Pre-Commit 请求，进入 prepared 阶段； ② 事务预提交：参与者收到 Pre-Commit 请求后，执行事务操作， 并将 undo 和 redo 信息记录到事务日志； ③ 响应反馈：若参与者成功执行了事务操作，则返回 ACK 响应， 同时开始等到最终指令。

Pre-Commit 阶段（情况 2） 假如有任何一个参与者发送了 No 响应，或者等待超时后协调者未收 到参与者响应，则执行事务中断： ① 发送中断请求：协调者向所有参与者发送中断 abort 请求； ② 中断事务：参与者收到 abort 消息（或超时后仍未收到协调者消 息），执行事务的中断。

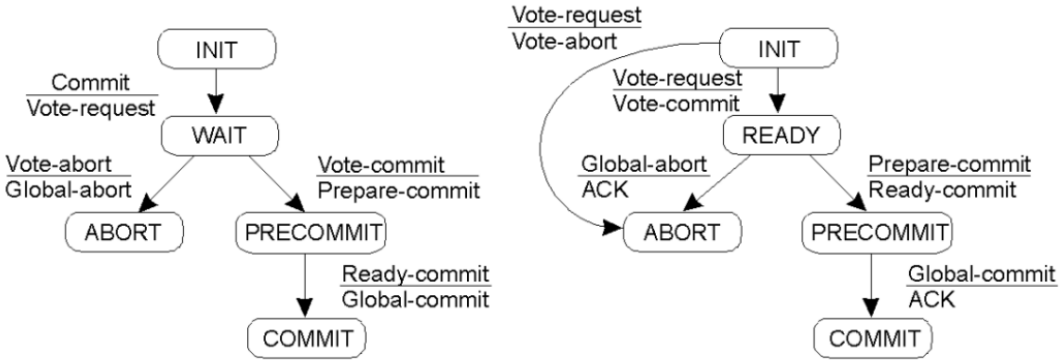
Do-Commit 阶段（情况 1） ① 协调者向各个参与者发送 Commit 通知； ② 所有参与者执行 commit 操作，释放资源后向协调者反馈结果。

Do-Commit 阶段（情况 2） ① 协调者向各个参与者发送 rollback 通知； ② 所有参与收到通知后执行 rollback 操作，并释放占有的资源； ③ 参与者向协调者反馈事务提交结果； ④ 协调者收到参与者反馈的 ACK 消息后，执行事务的中断。





3PC--有限的状态图



Coordinator　　Participant

- INIT：参与者此时崩溃不会产生任何影响，它可以直接中止事务并通知协调者。
- READY:此时正在等待pre-commit或abort消息，若此时崩溃并恢复后， 需要与其他参与者联系获取消息，再进行下一步动作。
  - 若其他参与者状态为pre-commit，则转为pre-commit；
  - 若有参与者状态为abort，则转为abort；
  - 若其他参与者均为ready，则继续等待协调者消息，超时则转为abort状态；
  - 若有参与者为init状态，则转为abort状态。
- ABORT:恢复状态后，中止事务，然后向协调者反馈。
- PRE-COMMIT: 恢复状态后，向协调者反馈，并等待Global-commit消 息，若超时还未收到消息，则联络其他进程再决定：
  - 若存在ready状态的进程，则说明此次分布式事务无法顺利完成；
  - 若所有进程都是pre-commit状态，则可以安全提交。
- COMMIT: 恢复到commit状态后，执行操作，然后向协调者反馈。

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？

一、CAP 和 ACID 中的 C 和 A 并不完全相同。

CAP 理论中的 C 和 A

- **C - 一致性**：在 CAP 中，一致性是指在分布式系统中，所有节点在数据更新后都能立即看到相同的数据状态。换句话说，如果一个更新操作成功完成，那么所有后续读取操作都应该返回该更新后的值。
- **A - 可用性**：可用性在 CAP 中意味着系统在任何时候都能够响应请求，即使部分节点发生故障。一个可用的系统应确保即便在发生分区情况下也能返回结果。

ACID 中的 C 和 A

- **C - 一致性**：在 ACID 中，一致性是指事务在开始和结束时，数据必须处于一致的状态。比如，一个银行转账事务在转账前后，账户的总余额应保持不变，即数据要符合系统的约束条件。ACID 中的一致性更关注事务对数据约束的遵守。
- **A - 原子性**：ACID 中的原子性表示一个事务中的所有操作要么全部成功，要么全部失败，不会发生部分执行的情况。这与 CAP 中的可用性并不相同。

二、CAP 与 BASE 的主要区别

（1）一致性要求：CAP 更关注强一致性（如 CP 系统），而 BASE 强调最终一致性，不要求所有节点实时一致。

（2）可用性权衡：CAP 理论基于严格的三选二原则，而 BASE 理论则允许系统在某些情况下部分可用，从而获得更高的扩展性和性能。

（3）应用场景：CAP 更适合事务性或要求数据高度一致的系统，如银行和金融系统；而 BASE 更适合一些对一致性要求不高、但需要高可用的场景，如电商网站、社交媒体等。

三、两者是否一致

发布订阅模式是一种消息传递模式，消息的发送者（发布者）不会直接将消息发送给特定的接收者，而是将消息发布到一个特定的主题（Topic），订阅者订阅主题后接收消息。消息队列模式是一种点对点的消息传递模式，消息的发送者将消息放入队列中，接收者从队列中取出消息进行处理。

区别：

- **消息传递模型**：发布订阅模式是一对多的广播模式，而消息队列是一对一的点

对点模式。

- **适用场景**：发布订阅适用于需要多个接收者接收同一消息的场景（如通知、事件），而消息队列适用于任务分发和负载均衡，确保每条消息只被一个消费者处理。
- **持久性**：消息队列更关注消息的可靠传递，支持持久化，而发布订阅模式中消息可能是瞬时的（视实现情况而定）。

两者在系统解耦方面的目标是一致的，但实现方式不同：

- 发布订阅模式更适合广播通知，接收者可以动态订阅取消订阅，适合实时性要求高的应用。
- 消息队列模式更适合任务处理和负载均衡，确保每条消息只被一个消费者处理，适合异步处理、任务分发等。