事务

事务的基本特性

ACID 属性:

- Atomic: 事务在外界看来是不可分割的。
- Consistent: 事务不违反系统不变性。
- Isolated: 并发事务之间不相互干扰。
- Durable: 一旦事务提交，变化是永久的。

事务API: `openTransaction`, `closeTransaction`, `abortTransaction`

事务实现

串行等价/并发控制：为了确保分布式系统中多个事务同时执行时的数据一致性，事务的实现依赖于多种并发控制方法，其中包括两阶段锁、时间戳排序和乐观并发控制。

1. **两阶段锁 (Two-Phase Locking, 2PL)**:
    - **原理**：事务在执行过程中分为两个阶段：增长阶段和收缩阶段。在增长阶段，事务可以获取任何必要的锁；在收缩阶段，事务只能释放已持有的锁，不能再获取新锁。
    - **优势**：确保所有冲突操作以相同的顺序执行，从而实现串行等价。
    - **严格两阶段锁**：所有在事务执行过程中获取的锁必须在事务提交或放弃后才能释放，防止事务放弃时的脏数据读取和过早写入问题。
2. **时间戳排序 (Timestamp Ordering)**:
    - **原理**：每个事务在开始时被赋予一个全局唯一的时间戳，所有操作按照时间戳排序进行。时间戳较小的事务的操作会先于时间戳较大的事务。
    - **优势**：保证所有事务按照其时间戳顺序执行，避免了死锁问题，但可能会导致较高的事务中止率。
3. **乐观并发控制 (Optimistic Concurrency Control, OCC)**:
    - **原理**：事务在执行期间不进行冲突检测，而是在提交阶段进行验证，确保没有其他事务对同一数据进行了冲突操作。
    - **优势**：适用于冲突较少的环境，减少了锁的开销，增加了并发度。
    - **缺点**：提交阶段可能会因检测到冲突而中止事务，导致性能下降。

事务放弃时的恢复：当事务由于某种原因（如系统故障或冲突）被放弃时，系统必须能够恢复到一致的状态。通过严格两阶段锁实现：

- 在这种方法中，所有事务在提交或放弃之前都不能释放已持有的锁。这确保了在事务放弃时，其他事务无法看到未提交的中间状态，避免了脏读和过早写入的问题。
- **优势**：通过确保所有事务提交后才释放锁，保证了数据一致性。
- **恢复**：若事务放弃，系统会回滚所有未提交的修改，确保系统状态恢复到事务开始前的状态。

事务的故障模型/Lampson

故障类型：

1. 磁盘故障：持久存储的写操作可能失败，如文件写错误或写入错误的块。
2. 服务器故障：服务器可以在任何时候崩溃，包括重启恢复时。
3. 通信故障：消息传递可能有任意长的延迟，消息可能丢失、重复或损坏。

解决方法：

1. 读数据时校验和判断数据块是否损坏：使用校验和（Checksum）机制来检测数据块是否在存储或传输过程中损坏。
2. 原子写解决磁盘故障：确保写操作是原子的，即写操作要么完全成功，要么完全失败，不会出现部分写入的情况。
3. 新进程替换崩溃进程：通过启动一个新进程替换崩溃的进程，保证系统能继续运行。
4. 可靠的RPC机制屏蔽通信错误：使用可靠的远程过程调用（RPC）机制，确保消息的传递是可靠的，即使在通信错误时也能处理重传和校验。
5. 接收方通过校验和检测受损消息：接收方使用校验和来检测消息是否在传输过程中损坏。

故障模型结论: 算法可在预见故障下正常工作，但对灾难性故障则无法保证

分布式事务

分布式事务是指在多个服务器上管理和操作对象的一种事务。其关键特性包括：

- **访问由多个服务器管理的对象**：分布式事务涉及对多个服务器上数据对象的操作。这些对象可能分布在不同的地理位置或不同的系统中。
- **结束时所有参与服务器全部提交或放弃**：分布式事务的原子性要求，当事务完成时，所有参与的服务器要么全部提交事务的操作，要么全部放弃操作，以保证数据一致性。
- **协调者负责保证所有服务器结果一致**：协调者在分布式事务中扮演关键角色，负责管理事务的提交和回滚操作，确保所有参与的服务器对事务的处理结果一致。

两阶段提交协议（2PC）是确保分布式事务在多个服务器之间一致性的一种常见协议。其工作机制分为两个阶段：

1. **准备阶段（Voting Phase）**：
    - 协调者向所有参与者发送准备提交（Prepare to Commit）请求。
    - 每个参与者接收到请求后，执行事务操作但不提交，并记录所有修改。如果所有操作成功，参与者返回准备就绪（Prepared）状态；否则，返回拒绝（Abort）状态。
2. **提交阶段（Commit Phase）**：
    - 协调者收集所有参与者的回复。如果所有参与者都返回准备就绪状态，协调者向所有参与者发送提交（Commit）请求，所有参与者提交事务。
    - 如果有任何参与者返回拒绝状态，协调者向所有参与者发送回滚（Abort）请求，所有参与者回滚事务。

分布式事务的类型

平面事务（Flat Transaction）：

- **定义**：平面事务是最基本的分布式事务类型，事务中的操作顺序执行，每个操作依次访问不同服务器上的对象。
- **特点**：平面事务中的每个请求都是顺序执行的，事务需要等待前一个请求完成后才能继续下一个请求。
- **优点**：实现相对简单，适用于操作顺序严格依赖的场景。
- **缺点**：并发性低，无法充分利用系统资源。

嵌套事务（Nested Transaction）：

- **定义**：嵌套事务允许在顶层事务中创建子事务，每个子事务可以进一步嵌套子事务。
- 特点：
  - **并发执行**：同层次的子事务可以并发执行，互不影响。
  - **灵活性**：子事务可以根据需要创建和销毁，提供了更高的灵活性。
  - **局部提交**：子事务可以单独提交或回滚，事务的父级状态取决于所有子事务的状态。
- **优点**：提高了并发性和灵活性，适用于复杂的业务逻辑和依赖关系。
- **缺点**：实现复杂度较高，涉及到更多的并发控制和状态管理。

分布式事务的并发控制

目标：分布式事务的并发控制的主要目标是确保所有事务以串行等价的方式执行。串行等价性意味着并发执行的事务的结果与某个顺序执行这些事务的结果相同。这保证了数据的一致性和正确性，即使在多个事务同时操作相同的数据时。

方法：

1. **时间戳排序 (Timestamp Ordering)**
   - **原理**：每个事务在开始时被分配一个唯一的时间戳。所有操作按照时间戳的顺序进行排序。
   - 过程：
     - 读操作：若当前事务的时间戳小于最近一次写操作的时间戳，则该读操作被拒绝，以保持一致性。
     - 写操作：若当前事务的时间戳小于最近一次读操作或写操作的时间戳，则该写操作被拒绝。
   - **优点**：避免了死锁问题，因为事务不需要等待其他事务的完成。
   - **缺点**：可能导致较高的事务中止率，因为较旧的事务可能被较新的事务频繁中止。
2. **乐观并发控制 (Optimistic Concurrency Control, OCC)**
   - **原理**：假设大多数事务不会冲突，因此在事务的执行阶段不进行冲突检测，而在提交阶段进行验证。
   - 过程：
     - 执行阶段：事务在本地执行操作，不对共享数据进行锁定。
     - 验证阶段：在提交前，系统检查是否存在冲突（即其他事务是否修改了当前事务读或写的数据）。
     - 提交阶段：如果没有冲突，事务提交；如果存在冲突，事务回滚并重试。

- **优点**：减少了锁的开销，增加了并发度，适用于低冲突环境。
- **缺点**：在高冲突环境中，事务提交阶段的冲突检测可能导致大量事务回滚，影响系统性能。

3. **两阶段锁 (Two-Phase Locking, 2PL)**
   - **原理**：事务的锁定过程分为两个阶段：获取锁（增长阶段）和释放锁（收缩阶段）。在增长阶段，事务可以获取任何必要的锁；在收缩阶段，事务只能释放锁，不能再获取新锁。
   - 过程：
     - 在执行事务操作时，事务获取所需的数据锁。
     - 在提交事务之前，事务释放所有已获取的锁。
   - **严格两阶段锁**：所有锁在事务提交或回滚后才能释放。这防止了脏数据读取和过早写入的问题。
   - **优点**：确保事务的串行等价性，防止脏读、不可重复读和幻读等问题。
   - **缺点**：可能导致死锁，需要额外的死锁检测和解决机制。

并发控制机制：更新丢失和不一致检索问题：

1. **更新丢失（Lost Update Problem）**
   - **描述**：两个事务同时读取相同的数据，并基于这些旧值进行更新，导致其中一个事务的更新丢失。
   - 示例：事务T1读取值X并计算新值Y，事务T2同时也读取值X并计算新值Z。T1写入Y后，T2覆盖了Y，写入Z，导致T1的更新丢失。
   - **解决方案**：通过锁机制或时间戳排序，确保事务的更新操作按顺序执行，避免覆盖前一个事务的更新。
2. **不一致检索（Inconsistent Retrieval Problem）**
   - **描述**：一个事务在另一个事务进行更新操作时，读取到部分更新的数据，导致数据不一致。
   - 示例：事务T1正在更新账户A和账户B的余额，事务T2在T1更新了账户A但未更新账户B时，读取了两个账户的余额，导致读取到不一致的数据。
   - **解决方案**：使用锁机制或乐观并发控制，确保读操作与更新操作互斥，防止事务读取到未完成更新的数据。

分布式死锁

死锁是指在计算机系统中，两个或多个事务互相等待对方释放资源，导致这些事务都无法继续执行的状态。死锁不仅在单服务器环境中可能发生，在分布式事务中也常见，由于分布式系统中资源分散在多个服务器上，死锁检测和解除变得更加复杂。

检测和解除方法：

1. **锁超时 (Lock Timeout)**
   - **原理**：每个锁都有一个时间期限，超过期限后锁将自动释放。这种方法通过强制事务等待时间不超过预设值来避免死锁。
   - 过程：
     - 每个事务获取锁时，设置一个超时时间。
     - 如果事务在超时时间内无法获取所需的所有锁，则放弃当前锁并回滚操作。
   - **优点**：实现简单，容易理解。
   - **缺点**：可能导致不必要的事务回滚，即系统中本没有死锁，但因锁超时机制导致事务回滚；选择合适的超时时间比较困难，过短可能导致频繁回滚，过长则难以及时发现死锁。

2. **等待图检测 (Wait-For Graph)**
    - **原理**：构建一个图，其中节点代表事务，边表示事务等待的资源。通过检测图中的环来判断是否存在死锁。
    - 过程：
        - 服务器记录每个事务的等待情况，构建等待图。
        - 定期检查等待图中的环，若存在环则表示发生死锁。
        - 通过回滚环中某个事务来解除死锁。
    - **优点**：能准确检测死锁。
    - **缺点**：构建和维护等待图开销较大，适用于事务较少的环境。
3. **集中式死锁检测 (Centralized Deadlock Detection)**
    - **原理**：通过一个中央控制器（全局死锁检测器）收集和分析各个服务器的局部等待图，合并为全局等待图，检测死锁。
    - 过程：
        - 各服务器定期向全局死锁检测器发送局部等待图。
        - 全局死锁检测器合并局部等待图，构建全局等待图，检测环路。
        - 若发现死锁，选择回滚某个事务，解除死锁，并通知相关服务器。
    - **优点**：统一管理，能有效检测全局死锁。
    - **缺点**：中央控制器可能成为单点故障，通信开销大，缺乏容错性和可扩展性。
4. **边追逐方法（非集中式）(Edge Chasing)**
    - **原理**：不需要构建全局等待图，而是通过在系统中传递探测消息（probe message）来检测环路。各服务器在发现事务等待时发送探测消息，通过消息的流转发现死锁。
    - 过程：
        - 当服务器A发现事务T开始等待事务U时，向阻塞U的服务器B发送探测消息<T,U>。
        - 服务器B收到消息后，若U也在等待其他事务，则转发探测消息；否则认为没有死锁。
        - 若探测消息返回到初始发送服务器，表示存在环路，即发生死锁。
        - 检测到死锁后，选择环路中的一个事务回滚，解除死锁。
    - **优点**：分布式检测，无需集中控制，具备较好的容错性和可扩展性。
    - **缺点**：消息传递和处理的开销较大，可能导致检测延迟。

事务恢复

事务放弃时的恢复

在分布式事务处理中，确保系统在事务放弃（即abort）后能够正确恢复至一致状态是至关重要的。这涉及处理未提交的数据和确保未提交事务的影响不会传播到其他事务中。

1. **脏数据读取（Dirty Read）**
    - **定义**：脏数据读取是指一个事务读取了另一个未提交事务所写入的数据。如果未提交的事务最终被回滚，那么读取该数据的事务就会基于无效的数据进行操作，导致数据不一致。
    - 示例：事务A修改了数据X并未提交，事务B读取了X的修改值。如果事务A随后回滚，事务B已经基于这个无效数据进行了操作，导致数据不一致。
2. **过早写入（Premature Writes）**

- **定义**：过早写入是指一个事务在另一个未提交事务所修改的数据基础上进行了修改。如果未提交的事务被回滚，那么后续的事务将基于无效的数据进行写入，导致数据不一致。
- 示例：事务A修改了数据X并未提交，事务B基于事务A修改后的值进一步修改X。如果事务A回滚，那么事务B的修改基于无效的数据，导致数据不一致。

在分布式事务处理中，恢复机制的设计和实现直接影响系统的数据一致性和可靠性。通过严格的并发控制策略，如严格两阶段锁、使用前映像以及乐观并发控制，系统能够有效地避免脏数据读取和过早写入问题。

- **严格两阶段锁**提供了强有力的锁机制，确保事务提交或回滚前不会释放锁，防止其他事务读取未提交的数据。
- **使用前映像**的策略通过保存数据修改前的状态，确保在事务回滚时能够恢复数据的一致性。
- **乐观并发控制**则通过提交阶段的冲突检测，确保事务在提交前不会受到未提交事务的影响，提高了系统的并发性能。

服务器崩溃后的恢复

在分布式系统中，服务器崩溃是不可避免的，为了保证数据的一致性和完整性，系统必须具备有效的恢复机制。恢复管理器（Recovery Manager）在事务恢复过程中起着至关重要的作用。

1. **保存已提交事务对象**
   - **任务**：确保所有已提交的事务对象被持久化存储。
   - **作用**：即使服务器崩溃，也可以通过恢复已提交的事务对象来保证数据的一致性。
   - **实现**：通常通过日志记录每个事务的提交状态和相关数据，确保在崩溃后能够重建这些对象。
2. **恢复服务器对象**
   - **任务**：在服务器崩溃后，恢复管理器负责恢复所有对象的状态，使其反映所有已提交事务的结果。
   - **作用**：保证系统在重启后能够继续正常运行，并保持数据一致性。
   - **实现**：恢复管理器从恢复文件中读取对象的最新状态，并将其应用到服务器的运行状态中。
3. **重组恢复文件**
   - **任务**：定期重组恢复文件，以提高恢复效率和节省存储空间。
   - **作用**：通过重组恢复文件，可以加快崩溃后的恢复过程，并回收未使用的存储空间。
   - **实现**：创建检查点，将所有已提交的对象和事务状态记录到新的恢复文件中，替换旧的恢复文件。
4. **处理介质故障**
   - **任务**：应对硬盘等存储介质的故障，确保数据不丢失。
   - **作用**：通过冗余和备份机制，在介质故障发生时能够恢复数据。
   - **实现**：常用方法包括数据镜像、RAID（独立磁盘冗余阵列）、定期备份等。

恢复文件的形式：

1. **日志**
   - **定义**：日志文件记录了服务器上执行的所有操作的历史，包括每个事务的开始、提交、回滚和数据修改。
   - 结构：
     - **操作历史**：记录所有对象值、事务状态和意图列表。

        - **顺序写入**：日志中的记录按事务执行的顺序写入，确保数据一致性。
    - 优点：
        - 提供详细的事务执行历史，有助于在崩溃后完整恢复系统状态。
        - 顺序写入提高了写入效率，减少了随机写的开销。
    - 恢复过程：读取日志文件，从最后一次有效的检查点开始，应用所有已提交的事务操作，忽略未完成或已回滚的事务。
2. **阴影版本 (Shadow Paging)**
    - **定义**：阴影版本通过维护数据的影子副本，确保在事务提交前的所有修改都应用到影子页，而不是直接修改原始数据页。
    - 结构：
        - **影子页表**：事务开始时创建的页表，记录影子页的地址。
        - **提交过程**：事务提交时，将影子页表替换为新的页表，原始页表变为旧的影子页表。
    - 优点：
        - 避免了对原始数据的直接修改，提高了数据的一致性和可靠性。
        - 在回滚时，只需丢弃影子页表即可恢复原始数据，无需逐条撤销操作。
    - 恢复过程：通过影子页表恢复所有已提交的事务数据，忽略影子页表中未提交的事务修改。

恢复管理器在分布式系统中承担了确保数据一致性和完整性的重任。在服务器崩溃后，通过保存已提交事务对象、恢复服务器对象、重组恢复文件和处理介质故障，恢复管理器能够有效地恢复系统状态，确保系统的高可用性和可靠性。

两阶段提交协议的恢复

在分布式系统中，两阶段提交协议（2PC）是一种常用的分布式事务处理协议，旨在确保所有参与节点在提交或回滚时达到一致性。然而，服务器崩溃是不可避免的，因此2PC的恢复机制至关重要。

2PC的恢复需要处理两种状态：完成和不确定：

1. **完成状态（Completed）**
    - **定义**：事务的所有操作已经完成，所有参与者已经提交或回滚事务，协调者已经确认所有参与者的操作结果。
    - **恢复动作**：如果恢复文件中的状态为完成，则不需要任何进一步的操作，因为所有事务已经处理完毕并一致。
2. **不确定状态（Uncertain）**
    - **定义**：事务已经进入准备阶段（Prepared），但尚未收到提交或回滚的最终决议。参与者已准备好提交，但在收到协调者的最终决议前发生了崩溃。
    - **恢复动作**：如果恢复文件中的状态为不确定，恢复管理器需要与协调者或其他参与者通信，以获取最终的决议（提交或回滚）。

恢复文件的信息包括事务状态、意图列表、协调者和参与者：

1. **事务状态（Transaction Status）**
    - **内容**：记录事务的当前状态（如准备好、提交、回滚）。
    - **作用**：在服务器崩溃后，恢复管理器需要根据事务状态来决定接下来的恢复操作。例如，如果事务状态为准备好，则需要等待或请求协调者的最终决议。
2. **意图列表（Intention List）**
    - **内容**：记录事务对各个数据对象的操作意图，包括数据对象的标识和修改内容。

- **作用**：在事务提交或回滚时，恢复管理器根据意图列表执行相应的操作，以确保数据的一致性。例如，提交时将意图列表中的修改应用到数据对象，回滚时则撤销这些修改。

3. **协调者（Coordinator）**
   - **内容**：记录协调者的标识和通信信息。
   - **作用**：在参与者崩溃后，恢复管理器需要联系协调者以获取事务的最终决议。如果协调者也发生崩溃，则需要选举新的协调者或等待协调者恢复。
4. **参与者（Participants）**
   - **内容**：记录参与者的标识和通信信息。
   - **作用**：协调者在崩溃后，需要联系所有参与者以确认事务的状态，并根据参与者的回复决定提交或回滚事务。

恢复过程：

1. **协调者的恢复**
   - **准备好状态**：协调者在崩溃恢复后，首先检查恢复文件。如果发现事务状态为准备好（但未完成），则协调者需要重新发送提交或回滚请求给所有参与者。
   - **已提交状态**：如果协调者发现事务状态为已提交，则需要确认所有参与者是否完成提交操作。如果参与者未回复，则继续发送提交请求。
   - **已放弃状态**：如果事务状态为已放弃，协调者需要确认所有参与者是否已完成回滚操作，并发送回滚请求。
2. **参与者的恢复**
   - **不确定状态**：参与者在崩溃恢复后，如果发现事务状态为不确定，则需要向协调者发送请求，获取事务的最终决议（提交或回滚）。
   - **已提交状态**：如果事务状态为已提交，则参与者执行提交操作，并向协调者确认。
   - **已放弃状态**：如果事务状态为已放弃，则参与者执行回滚操作，并向协调者确认。

恢复示例：

- **协调者的恢复管理器**
  - **准备好状态**：协调者崩溃后恢复，发现事务状态为准备好，但未完成决议。协调者需重新与参与者通信，发送提交或回滚请求，等待所有参与者的确认。
  - **已提交状态**：协调者恢复后，发现事务状态为已提交。协调者需确认所有参与者是否完成提交操作，未回复的参与者需重新发送提交请求。
  - **已放弃状态**：协调者恢复后，发现事务状态为已放弃。协调者需确认所有参与者是否完成回滚操作，未回复的参与者需重新发送回滚请求。
- **参与者的恢复管理器**
  - **不确定状态**：参与者崩溃后恢复，发现事务状态为不确定。参与者需向协调者发送请求，获取事务的最终决议（提交或回滚）。
  - **已提交状态**：参与者恢复后，发现事务状态为已提交。参与者需执行提交操作，并向协调者确认。
  - **已放弃状态**：参与者恢复后，发现事务状态为已放弃。参与者需执行回滚操作，并向协调者确认。

恢复文件的重组

在分布式系统中，为了提高恢复过程的效率和减少存储开销，需要定期对恢复文件进行重组。恢复文件重组的核心任务是通过检查点机制，确保恢复文件的内容始终反映当前系统的最新状态，同时优化存储空间的使用。

检查点是恢复文件重组的关键机制，它涉及将当前系统的状态记录到一个新的恢复文件中，以便在系统崩溃后能够快速恢复。检查点主要包含以下步骤：

1. **写入当前服务器所有已提交对象版本**
    - **任务**：将所有已提交事务所涉及的数据对象的最新状态写入新的恢复文件。
    - **作用**：确保在恢复过程中，能够快速重建系统的当前状态，而无需逐条回放所有事务日志。
    - 实现：
        - 创建一个新的恢复文件，记录当前所有已提交对象的版本。
        - 这些对象版本应包括所有最新的、已提交的事务修改，确保数据的一致性和完整性。
2. **更新恢复文件内容**
    - **任务**：将现有恢复文件中的有效内容转移到新的恢复文件，并将旧的恢复文件标记为过期或删除。
    - **作用**：通过将无效或冗余的数据移除，优化恢复文件的存储空间，并提高系统恢复效率。
    - 实现：
        - 在新的恢复文件中，记录事务状态、意图列表等必要的元数据，以确保在事务恢复过程中能够正确识别和处理所有事务。
        - 逐步将旧恢复文件中的未提交事务和相关信息转移到新文件中，确保事务的完整性。

恢复文件重组的步骤：

1. **创建新的恢复文件**
    - 启动检查点操作时，首先创建一个新的恢复文件，用于存储所有已提交对象的最新状态和事务元数据。
2. **写入当前已提交对象版本**
    - 将系统中所有已提交事务所修改的对象的最新版本写入新的恢复文件。
    - 确保这些对象版本反映了所有已提交事务的最新状态，提供完整的数据快照。
3. **转移未提交事务信息**
    - 将旧恢复文件中尚未提交的事务及其相关信息（如事务状态、意图列表）转移到新的恢复文件中。
    - 记录这些事务的当前状态，以便在系统恢复时能够正确处理这些事务。
4. **标记旧恢复文件**
    - 在新恢复文件创建完成并验证无误后，将旧恢复文件标记为过期或删除。
    - 确保新恢复文件成为系统的主要恢复数据源，提高数据恢复的效率和准确性。

检查点的优点：

1. **提高恢复效率**：通过定期记录系统的快照，减少了崩溃后需要回放的事务日志条目数量，从而加快系统恢复速度。
2. **优化存储空间**：通过清理旧的、无效的恢复文件内容，减少存储空间的浪费，提高系统存储资源的利用效率。

3. **确保数据一致性**：检查点操作确保系统在任意时刻都有一个最新的、一致的状态快照，即使在崩溃后也能迅速恢复到这一状态。

实践中的注意事项：

1. **检查点频率**：检查点的频率需要根据系统的事务处理量和恢复时间要求来设定。频率过低可能导致恢复时间过长，频率过高则可能带来额外的系统开销。
2. **事务状态的处理**：在创建检查点时，必须确保所有事务状态正确记录，特别是那些处于准备提交或等待提交状态的事务，以避免恢复过程中出现数据不一致。
3. **系统性能**：检查点操作可能会对系统性能产生一定影响，因此需要在系统负载较低时执行，或者采用增量检查点技术，减少对系统运行的影响。