第5章

事务管理与并发控制

学习目标：

- ❖ 了解什么是事务。
- ❖ 掌握常见的事务操作。
- ❖ 了解事务的隔离级别。
- ❖ 掌握事务的异常处理。
- ❖ 掌握事务的并发控制。

事务管理与并发控制，是数据库管理系统中保证数据完整性和一致性的关键技术。事务被定义为一个或多个数据库操作的集合，这些操作作为一个单一的工作单元执行，要么全部成功，要么全部失败，从而保证了数据库的一致性。本章将针对事务管理与并发控制进行详细讲解。

5.1 事务机制

5.1.1 事务的概念

事务是数据库管理系统中的一个基本概念，它指的是作为单个逻辑工作单位执行的一系列操作，也可以说是一系列操作的集合，这些操作作为一个整体一起执行，要么全部成功，要么全部失败。

以现实生活中转账为例，转账可以分为转入和转出，只有这两部分都完成才认为转账成功，在数据库中，这个过程是使用两条 SQL 语句来完成的，如果其中任意一条语句出现异常没有执行，则会导致两个账户的金额不同步，造成错误。此时就可以使用事务来解决这个问题，将两条语句作为一个整体一起执行，要么全部执行成功，要么全部执行失败，以确保账户金额的同步。

事务的核心特点是原子性、一致性、隔离性、持久性，这些特性常被称为 ACID 属性。

1. 原子性

原子性(Atomicity)意味着事务中的所有操作要么全部成功执行，要么全部不执行。事

务作为一个整体被执行，不允许只执行事务中的一部分操作。如果事务中的某个操作失败，整个事务将被回滚（撤销）到事务开始之前的状态。

2. 一致性

一致性（Consistency）确保事务的执行将数据库从一个一致的状态转变到另一个一致的状态。一致状态的定义是基于数据库的完整性约束，如外键约束、唯一约束等。事务执行过程中不应违反这些约束。

3. 隔离性

隔离性（Isolation）是指并发执行的事务之间的操作是隔离的，一个事务的操作不应该被其他事务干扰。数据库系统通过并发控制机制实现事务的隔离性，以避免诸如脏读、不可重复读和幻读等问题。

4. 持久性

持久性（Durability）意味着一旦事务被提交，它对数据库所做的更改就是永久的，即使系统发生故障。提交后的事务结果被持久化存储在数据库中，不会因为系统故障而丢失。

事务的管理对于维护数据库的完整性和一致性至关重要。数据库管理系统提供了事务管理的机制，允许开发者控制事务的开始、执行、提交或回滚，以确保数据的准确性和可靠性。通过使用事务，开发者可以构建出强大且稳定的数据库应用程序，有效地处理并发数据访问，保护数据免受损坏。

5.1.2　事务的操作

openGauss 是一个开源的关系数据库管理系统，由华为主导开发。它支持 SQL 标准，并提供了事务管理功能，使得开发者可以在应用程序中利用事务来保证数据的一致性和隔离性。在 openGauss 中，事务的操作主要涉及以下几方面。

1. 开始事务

在 openGauss 中，可以使用 BEGIN 或 START TRANSACTION 命令来开始一个新的事务。这标志着事务的开始，之后的所有数据库操作都将作为这个事务的一部分。

```
1.    BEGIN;
2.    或
3.    START TRANSACTION;
```

2. 提交事务

当事务中的所有操作都成功完成，且想要将这些更改永久保存到数据库中时，可以使用 COMMIT 命令来提交事务。提交事务会将自事务开始以来进行的所有数据修改永久化到数据库中。

```
1.    COMMIT;
```

3. 回滚事务

如果在事务执行过程中遇到错误，或者出于某种原因决定放弃事务中所做的所有修改，可以使用 ROLLBACK 命令来回滚事务。回滚将撤销自事务开始以来所做的所有修改，将数据库恢复到事务开始时的状态。

```
1.    ROLLBACK;
```

4. 设置保存点

在事务执行的过程中,可以设置一个或多个保存点(SAVEPOINT)。保存点允许在事务内部标记一个特定的点,之后如果需要,可以仅将事务回滚到这个点,而不是完全回滚事务。使用 ROLLBACK TO SAVEPOINT 命令来回滚到特定的保存点。

```
ROLLBACK TO SAVEPOINT;
```

5. 事务隔离级别的设置

在 openGauss 中,可以通过 SET TRANSACTION 命令来设置事务的隔离级别,具体语句如下。

```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

上述语句设置了当前事务的隔离级别为"读已提交",这是 4 种标准 SQL 隔离级别之一。使用事务是保持数据库一致性和完整性的关键机制,尤其是在并发访问的环境中。在 openGauss 中有效地使用事务,可以帮助开发者确保应用的数据处理逻辑既准确又高效。

下面通过一个例子来演示事务的操作。假设有一个简单的银行账户表 accounts,其中包含三个字段 id(账户 ID)、account_number(卡号)和 balance(账户余额)。账户 1 初始余额 1000 元,账户 2 初始余额 2000 元,然后来模拟转账的过程。

首先,需要创建这个表并插入一些初始数据,具体语句如下。

```
CREATE TABLE accounts (
    account_id INT PRIMARY KEY,
    account_number VARCHAR(20),
    balance DECIMAL(10, 2)
);
INSERT INTO accounts (account_id, account_number, balance) VALUES (1, 6214850117715334, 1000.00);
INSERT INTO accounts (account_id, account_number, balance) VALUES (2, 6214850117715335, 2000.00);
```

然后将通过一个事务来模拟从一个账户向另一个账户转账的过程,具体语句如下。

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
COMMIT;
```

上述语句执行成功后,可以通过 SELECT 语句查询 accounts 表中两个账户的余额情况,查询结果如下。

```
schooldb = # SELECT * FROM accounts;
account_id | account_number | balance
------------+------------------+---------
         1 | 6214850117715334 |  900.00
         2 | 6214850117715335 | 2100.00
(2 rows)
```

在上述例子中,首先开始了一个事务,然后执行转账操作,如果语句操作均正常,则执行事务的提交操作,完成转账功能。

在实际应用中,事务可能会涉及更复杂的逻辑和错误处理机制,还需要根据具体的业务逻辑和数据一致性要求来设计和实现事务。

5.1.3　事务的异常处置

在数据库操作中，事务异常处置是处理事务执行过程中遇到的错误和异常的过程。这些错误可能是由于多种原因造成的，包括数据冲突、违反约束条件、系统错误等。正确处理这些异常对于保持数据库的完整性和一致性至关重要。以下是一些常见的事务异常处置策略。

1. 回滚事务

回滚(Rollback)是处理事务异常的最直接方式。当事务中的一个操作失败时，可以使用回滚操作撤销事务中所有已执行的操作，使数据库回到事务开始之前的状态。这样可以保证数据库的一致性不被破坏。

```
BEGIN;
-- 执行一系列数据库操作
DO SOMETHING;
-- 如果发生错误
ROLLBACK; -- 撤销所有更改
```

接下来仍以转账为例来演示事务回滚，设置账户 1 初始余额为 1000 元，账户 2 初始余额为 2000 元，假设账户 1 想要给账户 2 转账 100 元，此时可以开启一个事务，通过 UPDATE 语句将账户 1 的 100 元钱转给账户 2，具体语句如下。

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
```

上述语句执行完成后，使用 SELECT 语句查询两个账户的余额，查询结果如下。

```
schooldb = # SELECT * FROM accounts;
account_id |  account_number  | balance
--------+---------------+----------
         1 | 6214850117715334 |  900.00
         2 | 6214850117715335 | 2100.00
(2 rows)
```

从上述结果可以看出，账户 1 的余额为 900 元，账户 2 的余额为 2100 元，说明转账成功。然而此时账户 1 反悔了不想给账户 2 转账，由于事务还没有提交，就可以将事务回滚，取消本次转账操作，具体语句如下。

```
ROLLBACK;
```

ROLLBACK 语句执行成功后，再次使用 SELECT 语句查询两个账户的余额，查询结果如下。

```
schooldb = # SELECT * FROM accounts;
account_id |  account_number  | balance
--------+---------------+----------
         1 | 6214850117715334 | 1000.00
         2 | 6214850117715335 | 2000.00
(2 rows)
```

从查询结果可以看出，账户 1 的金额是 1000 元且账户 2 的金额是 2000 元，并没有完成转账的功能，因此说明当前事务中的操作被取消了并没有执行。

2. 使用保存点

在事务执行过程中设置保存点(Savepoints),可以在事务失败时只回滚到特定的保存点,而不是完全回滚事务。这允许更细粒度的错误恢复,可以在不放弃整个事务的情况下修正错误,具体语句如下。

```
BEGIN;
SAVEPOINT savepoint_name; -- 设置保存点
 -- 执行一系列操作
DO SOMETHING;
 -- 如果某个操作失败
ROLLBACK TO savepoint_name; -- 回滚到保存点
 -- 继续其他操作或最终提交事务
COMMIT;
```

还是以转账为例,设置账户 1 初始余额为 1000 元,账户 2 初始余额为 2000 元,假设账户 1 想要给账户 2 转账 100 元,此时可以开启一个事务,通过 UPDATE 语句将账户 1 的 100 元钱转给账户 2,具体语句如下。

```
BEGIN;
SAVEPOINT before_transfer;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 2;
```

上述语句执行完成后,使用 SELECT 语句查询两个账户的余额,查询结果如下。

```
schooldb = # SELECT * FROM accounts;
account_id |  account_number  | balance
 --------+---------------+----------
         1 | 6214850117715334 |  900.00
         2 | 6214850117715335 | 2100.00
(2 rows)
```

从上述结果可以看出,账户 1 的余额为 900 元,账户 2 的余额为 2100 元,说明转账成功。假设此时发生错误,如账户 2 不存在或转账金额不正确等,需要回滚到 before_transfer 保存点,撤销从账户 1 扣除的金额,就可以将事务回滚到保存点,具体语句如下。

```
ROLLBACK TO before_transfer;
```

上述语句执行成功后,再次使用 SELECT 语句查询两个账户的余额,查询结果如下。

```
schooldb = # SELECT * FROM accounts;
account_id |  account_number  | balance
 --------+---------------+----------
         1 | 6214850117715334 | 1000.00
         2 | 6214850117715335 | 2000.00
(2 rows)
```

从查询结果可以看出,账户 1 的金额是 1000 元且账户 2 的金额是 2000 元,并没有完成转账的功能。这样当账户 2 时发生错误,可以使用 ROLLBACK TO before_transfer;语句回滚到 before_transfer 保存点,这样账户 1 的扣款操作就会被撤销,保持了数据的一致性。

5.1.4 事务的隔离级别

事务的隔离级别定义了一个事务可能受到其他并发事务的影响程度。隔离级别的设置

是为了在并发访问数据库时，平衡数据的正确性和访问速度。SQL 标准定义了 4 种隔离级别，每个级别在并发性和数据一致性之间提供不同的权衡。以下是这 4 种隔离级别，从最低到最高。

1. 读未提交(Read Uncommitted)

在这个级别下，一个事务可以读取另一个事务未提交的数据。这种级别的并发性最高，但是它允许“脏读”(即读取到其他事务未提交的更改)。

2. 读提交(Read Committed)

事务只能读取到其他事务已经提交的数据。这个级别防止了脏读，但是仍然允许“不可重复读”，即在同一事务中，两次读取同一记录可能会得到不同的结果，因为其他事务在这两次读取之间进行了更新并提交。

3. 可重复读(Repeatable Read)

保证在同一个事务内，多次读取同一数据的结果是一致的，即避免了不可重复读。但是这个级别仍然可能出现“幻读”，即事务在读取某个范围的记录时，另一个事务插入了新的记录，导致第一个事务再次读取时会发现之前未见过的新记录。

4. 串行化(Serializable)

串行化是事务的最高隔离级别，它通过锁定涉及的所有行来防止脏读、不可重复读和幻读。这保证了完全的隔离，使得并发执行的事务看起来就像是依次串行执行的。

选择合适的隔离级别是在数据一致性需求和系统性能之间做出的重要权衡。较低的隔离级别提高了并发性能，但降低了数据的一致性保障；而较高的隔离级别虽然提供了更强的数据一致性保护，却以牺牲一定的并发性能为代价。数据库设计者和开发者需要根据具体的应用场景和业务需求，选择最适合的事务隔离级别。

5.2　并发控制

5.2.1　并发问题介绍

在数据库系统中，当多个事务同时执行时，如果没有适当的并发控制机制，就可能出现各种并发问题。这些问题不仅会影响数据的一致性和完整性，还可能导致数据的不一致性，破坏数据库的可靠性。主要的并发问题如下。

1. 脏读

脏读(Dirty Read)发生在一个事务读取了另一个事务未提交的数据时。如果那个事务回滚，它所做的更改就会消失，这意味着第一个事务读到了根本不存在的数据。

实际上，在 openGauss 中，即使设置事务的隔离级别为较低的 READ UNCOMMITTED，也不会导致脏读。这是因为 openGauss 使用多版本并发控制(MVCC)来保证即使在最低隔离级别下也不会出现脏读。脏读主要在理论上讨论，以更好地理解事务隔离级别对并发事务处理的影响。在生产环境中，通常应该避免脏读，因为脏读可能导致数据不一致。

介于上述原因，在 openGauss 中演示脏读可能比较困难，这里仅演示脏读出现的过程。接下来就以前面 accounts 表中的数据为例来演示脏读出现的过程，假设有两个并发事务 T1 和 T2，具体步骤如下。

T1：更新账户 1 的余额但未提交。

```
-- T1 开始
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
```

T2：读取了 T1 修改后的账户 1 余额。

```
-- 设置隔离级别以演示脏读
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
-- T2 开始
BEGIN;
-- 在 T1 提交或回滚之前读取账户 1 的余额
SELECT balance FROM accounts WHERE account_id = 1;
COMMIT;
```

T1：T1 回滚更改。

```
ROLLBACK
```

在上述操作中，如果 T2 在 T1 回滚之前读取了账户 1 的余额，则说明 T2 就发生了脏读。

2. 不可重复读

不可重复读(Non-repeatable Read)发生在一个事务中两次读取同一数据集合时，另一个并发事务更新了这些数据并提交，导致第一个事务两次读取的结果不一致，这主要是由于更新操作造成的。

接下来演示不可重复读，首先将事务隔离级别设置为 READ COMMITTED(这通常是默认级别，可以省略不写)，然后开始一个事务并读取数据，具体步骤如下。

T1：开始一个新事务，读取数据。

```
-- 设置隔离级别以演示不可重复读
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- 事务 1 开始
BEGIN;
SELECT balance FROM accounts WHERE account_id = 1; -- 假设返回 1000.00
```

查询结果：

```
balance
--------------
1000.00
(1 row)
```

T2：开始一个新事务，修改同一条记录的数据，然后提交。

```
-- 事务 2 开始
BEGIN;
UPDATE accounts SET balance = 900 WHERE account_id = 1;
COMMIT;
```

T1：在 T1 中再次读取相同的记录。

```
SELECT balance FROM accounts WHERE account_id = 1;
-- 在 READ COMMITTED 级别下，可能返回 900
COMMIT;
```

查询结果：

```
1.    balance
2.    --------------
3.    900.00
4.    (1 row)
```

在上述操作中，如果 T1 在 T2 提交更新后，再次读取时发现账户 1 的余额已经变了，就说明发生了不可重复读。

需要注意的是，由于 MVCC 的实现，openGauss 的 REPEATABLE READ 以及更高的事务隔离级别提供了对不可重复读的保护，就不会出现不可重复读的现象。

3. 幻读

幻读(Phantom Read)与不可重复读类似，但它涉及插入或删除操作。幻读发生在一个事务重新读取之前查询过的范围时，发现另一个事务插入或删除了符合查询条件的行。这样，第一个事务就会看到之前不存在的“幻影”数据。

接下来演示一下幻读，首先设置事务的隔离级别为 REPEATABLE READ，然后再开启一个事务执行插入操作，具体步骤如下。

T1：设置事务隔离级别为 REPEATABLE READ，然后开始一个事务并进行第一次查询。

```
1.    SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
2.    BEGIN;
3.    SELECT * FROM accounts WHERE balance > 100; -- 假设返回两行数据
```

查询结果：

```
2.    account_id |  account_number  | balance
3.    ---------+---------------+----------
4.             1 | 6214850117715334 |  900.00
5.             2 | 6214850117715335 | 2000.00
6.    (2 rows)
```

T2：开始另一个事务，并插入一条符合之前查询条件的新记录，然后提交。

```
1.    BEGIN;
2.    INSERT INTO accounts (account_id, account_number, balance) VALUES (3, 6214850117715336,
      1500.00);
3.    COMMIT;
```

T1：再次执行相同的查询。

```
1.    SELECT * FROM accounts WHERE balance > 100; -- 在某些数据库系统中,可能返回三行数据
2.    COMMIT;
```

查询结果：

```
1.    account_id |  account_number  | balance
2.    ---------+---------------+----------
3.             1 | 6214850117715334 |  900.00
4.             2 | 6214850117715335 | 1000.00
5.             3 | 6214850117715336 | 1500.00
6.    (3 rows)
```

在上述操作中，如果 T1 在 T2 提交插入操作后再次查询余额大于 100 的账户，会看到

之前未见过的新记录，即发生了幻读。

4. 丢失修改

丢失修改(Lost Update)发生在两个事务都尝试更新同一数据时。一个事务的更新可能会被另一个事务的更新所覆盖，结果是第一个事务的更新就丢失了。

在 openGauss 中，通过默认的事务隔离级别和锁机制，丢失修改(Lost Update)问题是被防止的，所以无法演示出丢失修改，也仅能演示过程。

T1：开始一个事务并查询 accounts 表中 account_id＝1 的余额，假设根据查询结果增加 100 元。

```
1.    BEGIN;
2.    SELECT balance FROM accounts WHERE account_id = 1;
```

T2：再开启另一个事务，查询同一记录的余额，然后更新这个余额，例如，增加 50 元。

```
1.    BEGIN;
2.    SELECT balance FROM accounts WHERE account_id = 1;
3.    UPDATE accounts SET balance = balance + 50 WHERE account_id = 1;
4.    COMMIT;
```

T1：基于最初的查询结果执行更新，假设增加 100 元。

```
1.    UPDATE accounts SET balance = balance + 100 WHERE account_id = 1;
2.    COMMIT;
```

在上述操作中，T2 的更新可能会被 T1 的更新所覆盖，因为 T1 没有考虑到在其事务开始后发生的 T2 更新，这就是典型的丢失修改问题。

通过上述示例说明了并发事务可能带来的问题。在实际应用中，通过设置合适的事务隔离级别，可以有效地控制这些并发问题。具体选择哪种隔离级别取决于应用的需求和对性能的考虑。

5.2.2 锁的分类介绍

锁是数据库管理系统中用来实现并发控制的一种机制，旨在管理不同事务对共享数据的访问。锁的基本原理是当一个事务在访问数据时，它会对数据加锁，以防止其他事务同时访问相同的数据，从而避免数据不一致性问题。

提到锁不得不介绍一下悲观锁和乐观锁，乐观锁和悲观锁是两种常见的并发控制机制，用于处理多个事务同时访问和修改同一资源的情况。

- 乐观锁：乐观锁假设多个事务在并发执行时不会彼此冲突，直到提交数据时才会检查是否有冲突。如果有冲突，则采取回滚等方式解决。相对悲观锁而言，采取了更加宽松的加锁机制，大多是基于数据版本(Version)记录机制实现，如版本号或时间戳等。
- 悲观锁：是指对数据被外界修改持保守态度，因此在整个数据处理过程中总是假设最坏的情况，认为会发生并发冲突，将数据处于锁定状态。悲观锁的实现通常依赖于数据库的锁机制。

了解了乐观锁和悲观锁的原理后，接下来可以根据锁的类型和粒度进行分类，从类型上进行细分，锁可以分为共享锁、排他锁、更新锁；从粒度上来分，锁可以分为行锁、页锁、表

锁、库锁。接下来针对锁的分类进行更详细的讲解。

1. 按锁的类型分类

- 共享锁(Shared Lock)：也称为读锁，允许多个事务同时读取一个资源，但阻止其他事务写入该资源。共享锁是一种乐观锁的实现，适用于读取操作频繁的场景。可以使用 SELECT…FOR SHARE 语句来获取共享锁。
- 排他锁(Exclusive Lock)：也称为写锁，只允许一个事务对数据进行修改(读或写)，阻止其他事务读取或者写入，直到排他锁被释放。排他锁是一种悲观锁的实现，适用于写入操作频繁的场景。可以使用 SELECT…FOR UPDATE 语句来获取排他锁。
- 更新锁(Update Lock)：更新锁是共享锁和排他锁的混合，用于在读取数据的同时防止其他事务修改数据，直到更新锁被释放。更新锁也属于悲观锁，可以使用 SELECT…FOR UPDATE(既是排他锁，也可以理解为更新锁)或 SELECT…FOR NO KEY UPDATE(无键更新锁)语句来获取更新锁。

2. 按锁的粒度分类

- 行锁(Row Lock)：锁定数据表中的特定行。这是最细的锁粒度，可以最大限度地减少锁冲突，但管理这种锁的开销也最大。
- 页锁(Page Lock)：锁定数据表中的页，页是数据库存储结构中的一部分，包含多行数据。
- 表锁(Table Lock)：锁定整个表。这种锁的粒度最大，会锁定表中的所有行，适用于对表执行大量操作的场景。
- 库锁(Database Lock)：锁定整个数据库，这是最大的锁粒度，用得较少，通常在执行数据库级别的操作时使用。

锁的设计和实现是数据库管理系统中解决并发控制问题的关键。正确使用锁可以在保证数据一致性和完整性的同时，提高并发访问的性能。不同的应用场景和需求可能需要不同类型和粒度的锁来平衡数据的安全性和系统的性能。

5.2.3　锁并发控制

锁并发控制是一种通过锁机制来控制多个事务并发访问和修改数据库资源的手段。在 openGauss 中，锁机制用于确保数据的一致性和隔离性。不同类型的锁具有不同的特性和兼容性，如共享锁允许多个事务同时读取资源，而排他锁则只允许一个事务对资源进行修改。此外，意向锁用于表示事务的锁意向，以便其他事务能够做出相应的反应。锁的粒度决定了锁所覆盖的数据范围，而锁的兼容性则决定了不同锁之间是否可以同时持有。

在实际应用中，锁并发控制对于确保数据的安全性和一致性至关重要。例如，在银行转账场景中，通过锁定用户 A 的账户，可以防止其他事务在转账过程中修改该账户，从而确保转账的正确性。

需要注意的是，锁的使用也会带来一定的开销，如锁的获取、释放和等待等操作都会消耗系统资源。因此，在设计数据库系统时，需要根据实际应用场景和需求来选择合适的锁策略和粒度，以平衡并发性能和系统开销之间的关系。

接下来会通过案例的形式来演示不同类型锁的使用。

1. 共享锁

示例：多个事务同时读取同一份数据，但不允许其他事务修改该数据。

T1：查询账户信息。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR SHARE; -- 获取共享锁
COMMIT;
```

T2：查询账户信息。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR SHARE; -- 获取共享锁,与事务 A 兼容
COMMIT;
```

T3：更新账户余额。这里由于 UPDATE 操作会修改数据，因此 openGauss 会自动在相应的行上加上排他锁，以防止其他事务同时进行修改。

```
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 1;
 -- 等待,因为排他锁与共享锁不兼容
COMMIT;
```

在上述例子中，T1 和 T2 可以同时获取共享锁并读取数据，但 T3 尝试获取排他锁来修改数据时会被阻塞，直到 T1 和 T2 释放它们的共享锁。

2. 排他锁

示例：一个事务修改数据，阻止其他事务同时读取或修改该数据。

T1：获取排他锁，更新账户余额。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR UPDATE; -- 获取排他锁
UPDATE accounts SET balance = balance + 100 WHERE account_id = 1
COMMIT;
```

T2：获取排他锁，查询账户信息。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR UPDATE;
 -- 等待,因为排他锁与排他锁不兼容
COMMIT;
```

在上述例子中，T1 获取排他锁后修改数据，T2 尝试获取同一资源的排他锁时会被阻塞，直到 T1 释放锁。

3. 更新锁

示例：一个事务正在读取数据并打算稍后更新，需要阻止其他事务修改这份数据，但允许其他事务继续读取。

T1：获取更新锁，读取账户余额数据，并对账户余额进行更新。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR NO KEY UPDATE; -- 获取更新锁
UPDATE accounts SET balance = balance + 100 WHERE account_id = 1
COMMIT;
```

T2：获取共享锁，读取账户余额。

```
BEGIN;
SELECT * FROM accounts WHERE account_id = 1 FOR SHARE;
-- 可以获取共享锁,因为更新锁与共享锁兼容
COMMIT;
```

T3：获取排他锁，更新账户余额。

```
BEGIN;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 1;
-- 等待,因为排他锁与更新锁不兼容
COMMIT;
```

在上述例子中，T1 获取更新锁后读取数据并计划修改，T2 可以获取共享锁来读取数据，但 T3 尝试获取排他锁来修改数据时会被阻塞。

5.2.4　多版本并发控制

多版本并发控制(Multiversion Concurrency Control，MVCC)是一种广泛使用的并发控制机制，它通过为数据库对象维护不同版本的数据来实现高效的事务隔离。MVCC 允许读操作和写操作并发执行，而不必彼此等待，极大地提高了数据库系统的并发性能。它特别适用于读操作远多于写操作的场景。

1. 工作原理

(1) 数据版本：每当数据被修改时，系统不是直接覆写旧数据，而是创建数据的一个新版本。每个版本都有一个唯一的时间戳或版本号。

(2) 读操作：当执行读操作时，MVCC 允许事务看到数据的一个一致性快照，这通常是事务开始时数据的状态。这意味着读操作可以访问到数据的旧版本，而不会被并发的写操作阻塞。

(3) 写操作：写操作创建数据的新版本，而不影响旧版本的数据，直到新事务提交。这样，不同的事务可以“同时”看到同一数据的不同版本。

2. 优点

(1) 非阻塞读操作：读事务不会被写事务阻塞，因为它们可以访问数据的旧版本。

(2) 减少锁争用：通过减少对共享数据的锁需求，MVCC 可以降低锁争用，提高系统的并发性能。

(3) 实现不同隔离级别：MVCC 可以灵活实现 SQL 标准定义的不同事务隔离级别，如读已提交(Read Committed)、可重复读(Repeatable Read)等。

3. 缺点

(1) 空间开销：因为需要为修改过的数据保留多个版本，MVCC 可能会增加数据存储的开销。

(2) 版本管理：系统必须有效地管理数据的不同版本，包括确定何时可以清理(或“回收”)旧版本的数据，这个过程通常称为垃圾收集或版本清理。

(3) 写操作性能：虽然 MVCC 显著提高了读操作的并发性能，但写操作可能因为需要创建新的数据版本和管理这些版本而产生额外的开销。

MVCC 被许多现代数据库系统采用，如 openGauss、PostgreSQL、MySQL 等。每个系统的 MVCC 实现细节可能不同，但基本原理相似，都是通过为数据对象提供多个版本来支

持高效的并发访问。通过使用 MVCC，数据库系统能够提供强大的并发性能，同时保持严格的事务隔离，确保数据的一致性和完整性。

需要注意的是，MVCC 通常是数据库管理系统内部实现的一部分，而且它的工作原理主要体现在数据版本的管理和访问上，这些都是在数据库的底层操作中自动处理的，无须人为处理。

小结

本章深入探讨了数据库系统中事务管理与并发控制的核心概念，包括事务的 ACID 属性、事务隔离级别以及如何通过锁机制和多版本并发控制（MVCC）来处理多个事务同时访问数据库时可能出现的并发问题。通过理解这些基本原则和技术，开发者可以设计出既保证数据一致性和完整性，又能高效处理并发请求的数据库应用，满足现代应用对数据处理的复杂需求。

习题

1. 请简要说明事务操作包括哪些。
2. 请简要说明并发问题有哪些。