

一、事务

概念

ACID

1. 原子性 (Atomicity)
2. 一致性 (Consistency)
3. 隔离性 (Isolation)
4. 持久性 (Durability)

ACID是由什么保证的

AUTO COMMIT

二、并发一致性问题(破坏隔离性)

丢失更新

读脏数据

不可重复读

幻影读

幻读和不可重复读的区别:

三、封锁

封锁粒度

封锁类型

1. 读写锁
2. 意向锁

例如:

意向锁的并发性

总结

封锁协议

1. 三级封锁协议
2. 两段锁协议

MySQL 隐式与显示锁定

四、隔离级别

未提交读 (READ UNCOMMITTED)

提交读 (READ COMMITTED)

可重复读 (REPEATABLE READ)

可串行化 (SERIALIZABLE)

五、多版本并发控制

基本思想

版本号

Undo 日志

ReadView

快照读与当前读

1. 快照读
2. 当前读

六、Next-Key Locks

Record Locks 记录锁

Gap Locks 间隙锁

Next-Key Locks 临键锁

七、关系数据库设计理论

函数依赖

异常

范式

1. 第一范式 (1NF)
2. 第二范式 (2NF)
3. 第三范式 (3NF)

八、ER 图

实体的三种联系

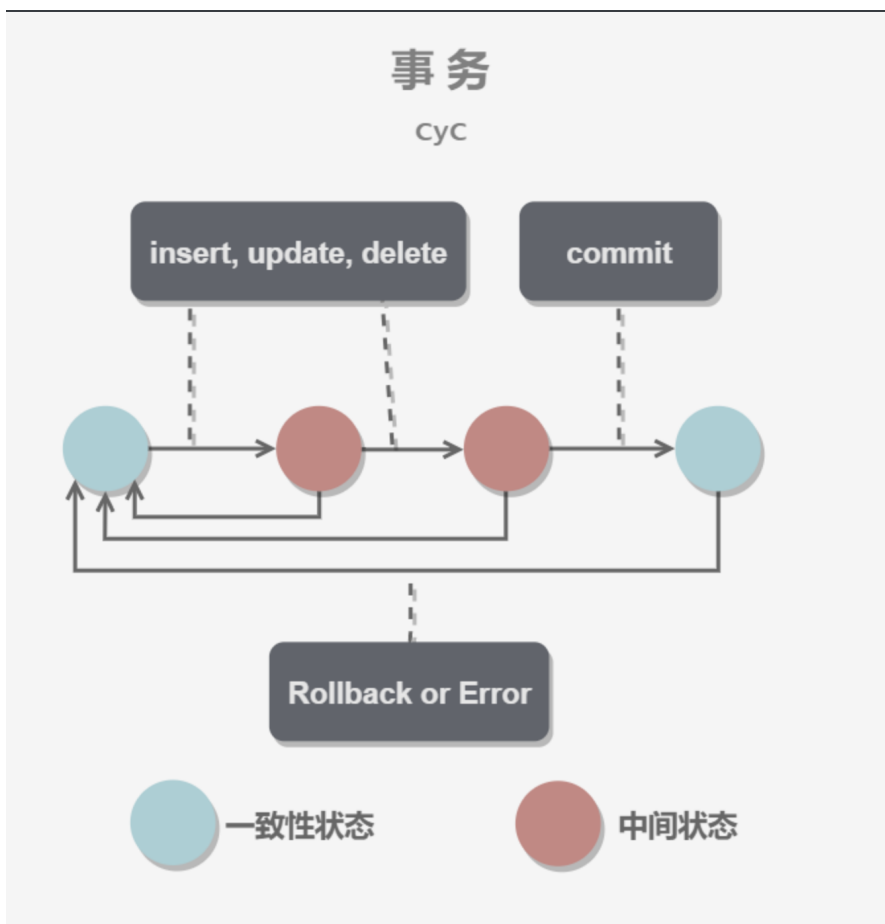
表示出现多次的关系

联系的多向性

一、事务

概念

事务指的是满足 ACID 特性的一组操作，可以通过 Commit 提交一个事务，也可以使用 Rollback 进行回滚。

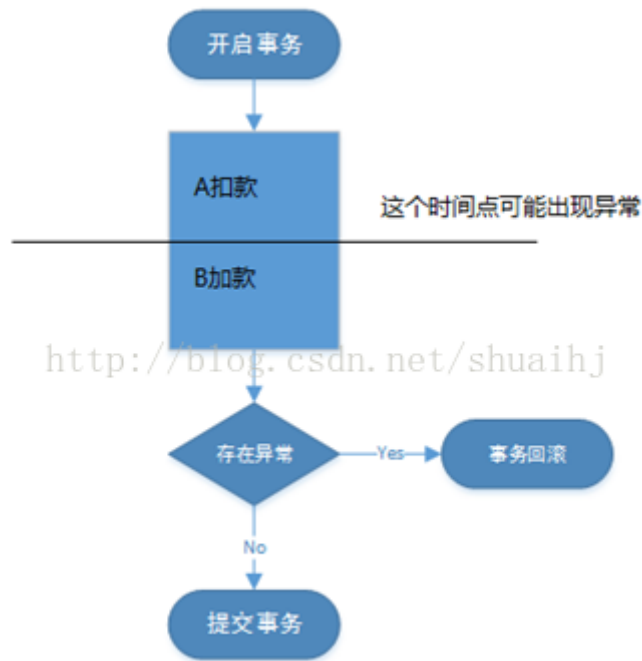


ACID

1. 原子性 (Atomicity)

事务被视为不可分割的最小单元，事务的所有操作要么全部提交成功，要么全部失败回滚。

回滚可以用回滚日志 (Undo Log) 来实现，回滚日志记录着事务所执行的修改操作，在回滚时反向执行这些修改操作即可。



在事务中的扣款和加款两条语句，要么都执行，要么就都不执行。否则如果只执行了扣款语句，就提交了，此时如果突然断电，A账号已经发生了扣款，B账号却没收到加款，就会引起纠纷。

在数据库管理系统（DBMS）中，默认情况下一条SQL就是一个单独事务，事务是自动提交的。只有显式的使用`start transaction`开启一个事务，才能将一个代码块放在事务中执行。保障事务的原子性是数据库管理系统的责任，为此许多数据源采用日志机制

2. 一致性（Consistency）

数据库在事务执行前后都保持一致性状态，数据库的完整性约束没有被破坏。在一致性状态下，所有事务对同一个数据的读取结果都是相同的。

例如: 对银行转账事务，不管事务成功还是失败，应该保证事务结束后ACCOUNT表中aaa和bbb的存款总额为2000元。

保障事务的一致性，可以从以下两个层面入手

2.1 数据库机制层面

使用约束: 数据库层面的一致性，在一个事务执行之前和之后，数据会符合你设置的约束（唯一约束，外键约束,Check约束等)和触发器设置。比如转账，则可以使用CHECK约束两个账户之和等于2000来达到一致性目的

2.2 业务层面

对于业务层面来说，一致性是保持业务的一致性。这个业务一致性需要由开发人员进行保证。当然，很多业务方面的一致性，也可以通过转移到数据库机制层面进行保证。

3. 隔离性（Isolation）

一个事务所做的修改在最终提交以前，对其它事务是不可见的。多个事务并发访问时，事务之间是隔离的，一个事务不应该影响其它事务运行效果。

事务之间的相互影响分为几种，分别为：脏读，不可重复读，幻读，丢失更新,这些问题可以通过不同的隔离级别和MVCC来保证

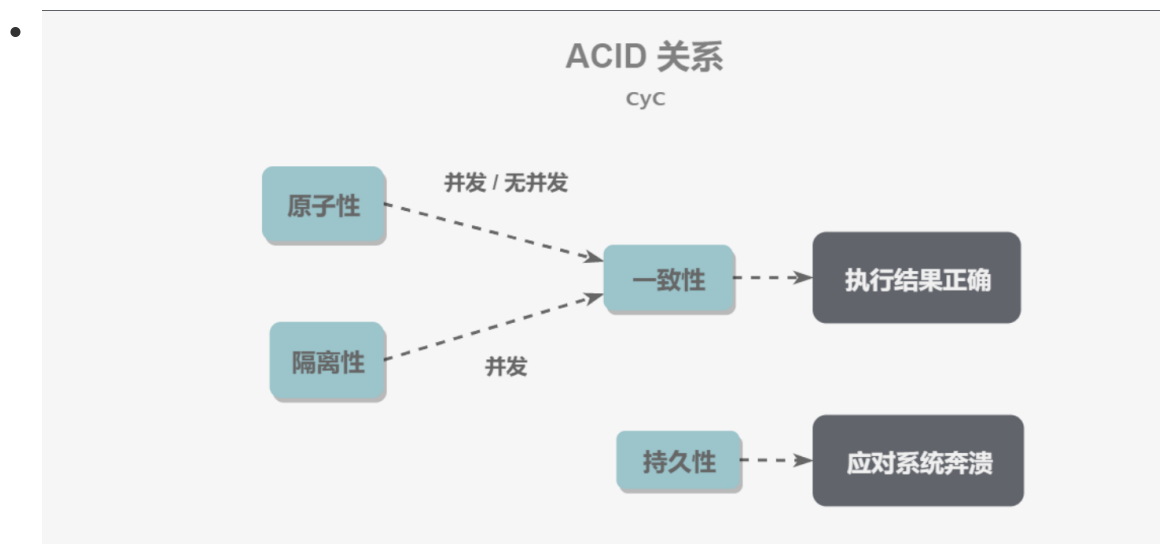
4. 持久性 (Durability)

一旦事务提交，则其所做的修改将会永远保存到数据库中。即使系统发生崩溃，事务执行的结果也不能丢失。

系统发生崩溃可以用重做日志 (Redo Log) 进行恢复，从而实现持久性。与回滚日志记录数据的逻辑修改不同，重做日志记录的是数据页的物理修改。

事务的 ACID 特性概念简单，但不是很好理解，主要是因为这几个特性不是一种平级关系：

- 只有满足一致性，事务的执行结果才是正确的。
- 在无并发的情况下，事务串行执行，隔离性一定能够满足。此时只要能满足原子性，就一定能满足一致性。
- 在并发的情况下，多个事务并行执行，事务不仅要满足原子性，还需要满足隔离性，才能满足一致性。
- 事务满足持久化是为了能应对系统崩溃的情况。



ACID是由什么保证的

- A原子性由undo log日志保证，它记录了需要回滚的日志信息，事务回滚时撤销已经执行成功的sql
- C一致性一般由代码层面来保证
- I隔离性由MVCC来保证
- D持久性由内存+redolog来保证，mysql修改数据同时在内存和redo log记录这次操作，事务提交的时候通过 redo log刷盘，宕机的时候可以从redo log恢复

AUTOCOMMIT

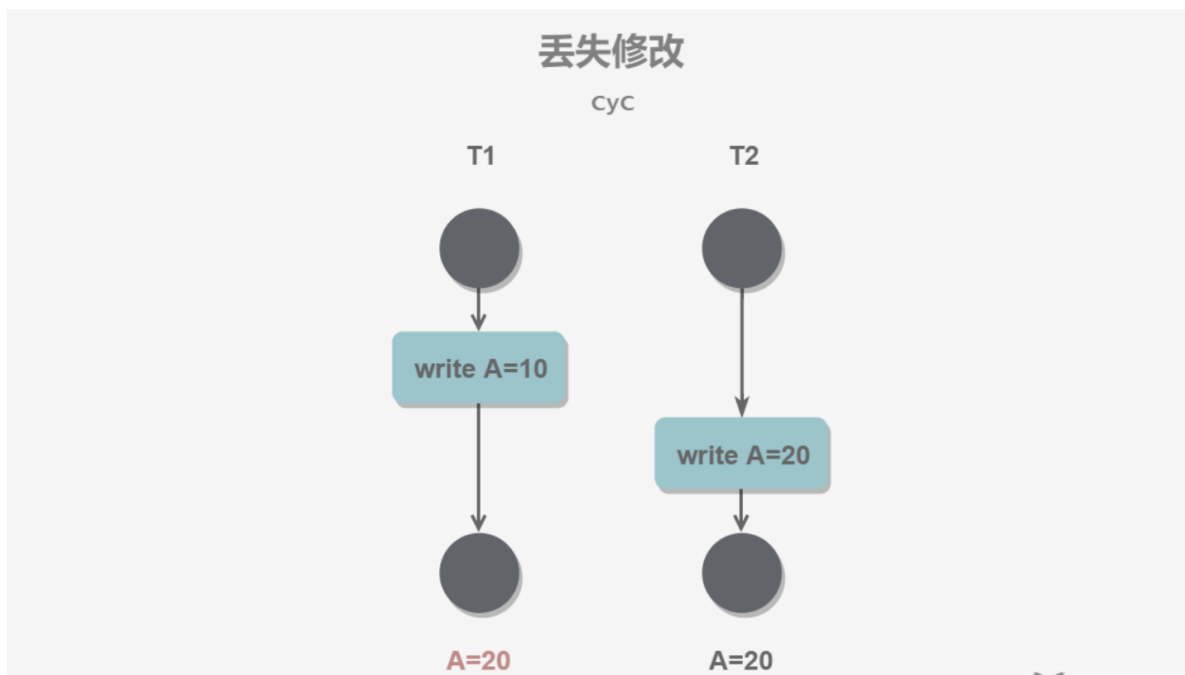
MySQL 默认采用自动提交模式。也就是说，如果不显式使用 `START TRANSACTION` 语句来开始一个事务，那么每个查询操作都会被当做一个事务并自动提交。

二、并发一致性问题(破坏隔离性)

在并发环境下，事务的隔离性很难保证，因此会出现很多并发一致性问题。

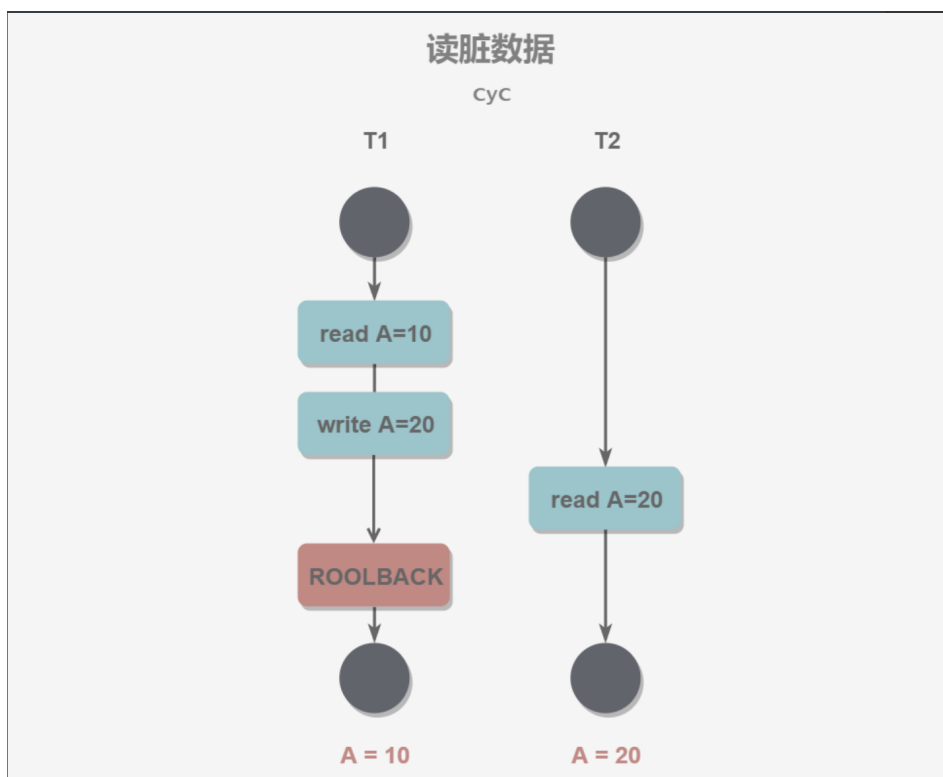
丢失更新

丢失更新指一个事务的更新操作被另外一个事务的更新操作替换。一般在现实生活中常会遇到，例如： T_1 和 T_2 两个事务都对一个数据进行修改， T_1 先修改并提交生效， T_2 随后修改， T_2 的修改覆盖了 T_1 的修改。



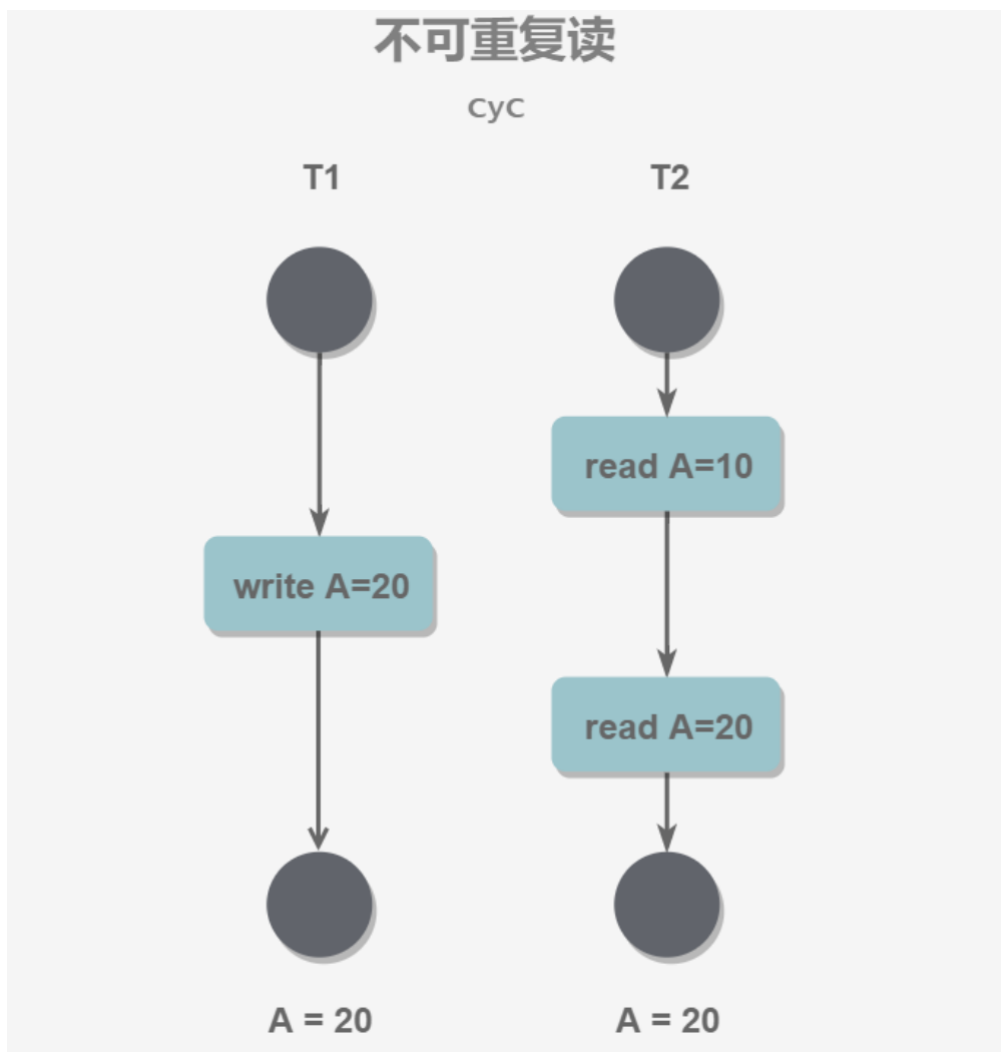
读脏数据

读脏数据指在不同的事务下，当前事务可以读到另外事务未提交的数据。例如： T_1 修改一个数据但未提交， T_2 随后读取这个数据。如果 T_1 撤销了这次修改，那么 T_2 读取的数据是脏数据。



不可重复读

不可重复读指在一个事务内多次读取同一数据集合。在这一事务还未结束前，另一事务也访问了该同一数据集合并做了修改，由于第二个事务的修改，**第一次事务的两次读取的数据可能不一致**。例如： T_2 读取一个数据， T_1 对该数据做了修改。如果 T_2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。

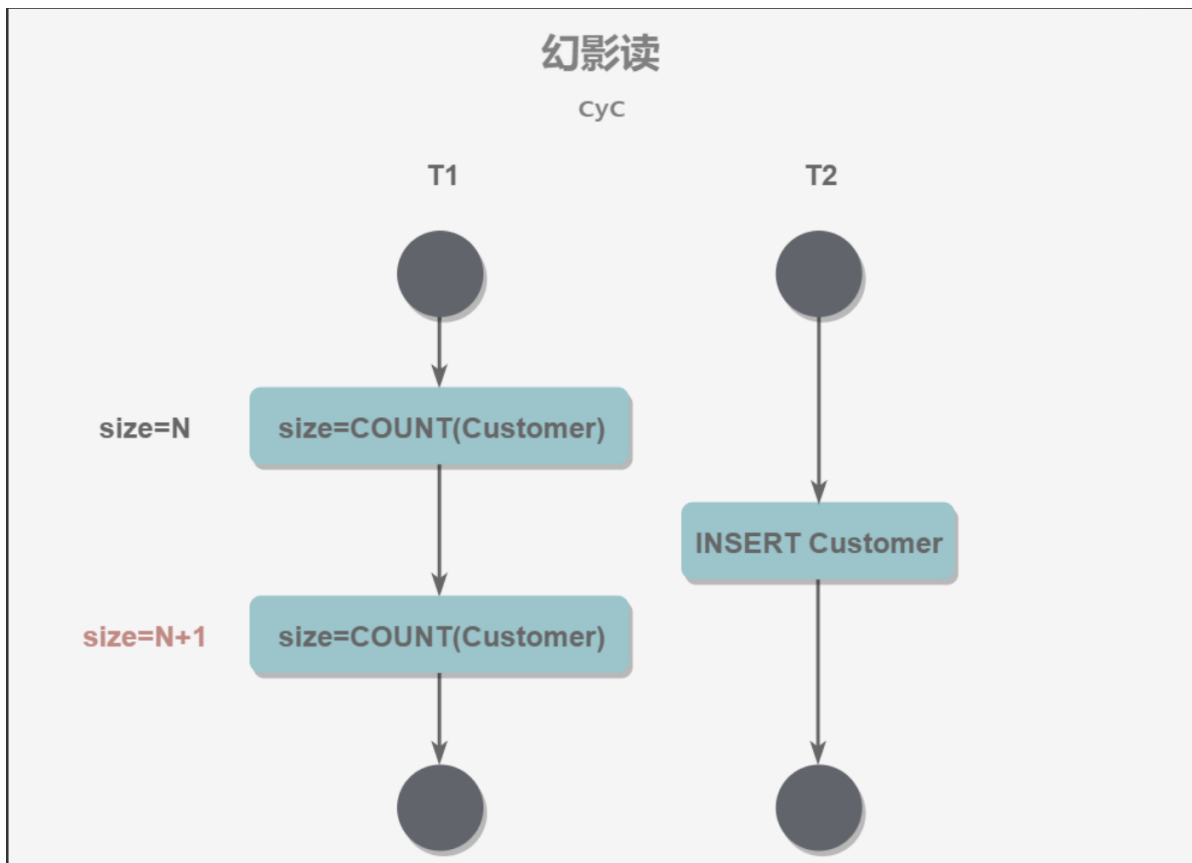


幻影读

幻影读也是指一个事务两次读取的结果不一致，例如 T_1 读取某个范围的数据， T_2 在这个范围内**插入**新的数据， T_1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。

幻读和不可重复读的区别:

- 不可重复读是针对一个事务对一条记录的**更新(UPDATE)**, 导致另外一个事务两次这个数据的结果不一致.
- 而幻影读是一个事务新**插入了一条记录(INSERT)**, 导致另一个事务读到了以前没有的数据



产生并发不一致性问题的主要原因是**破坏了事务的隔离性**，解决方法是通过**并发控制**来保证隔离性。并发控制可以通过封锁来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了**事务的隔离级别**，让用户以一种更轻松的方式处理并发一致性问题。

三、封锁

封锁粒度

MySQL 中提供了两种封锁粒度：**行级锁以及表级锁**。(MyISAM则只有表级锁)

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁、以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统吞吐量就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

封锁类型

1. 读写锁

- 互斥锁（Exclusive），简称为 X 锁，又称写锁。
- 共享锁（Shared），简称为 S 锁，又称读锁。

有以下两个规定：

- 一个事务对数据对象 A 加了 X 锁，就可以对 A 进行读取和更新。加锁期间**其它事务不能对 A 加任何锁**。
- 一个事务对数据对象 A 加了 S 锁，可以对 A 进行读取操作，但是不能进行更新操作。加锁期间**其它事务能对 A 加 S 锁，但是不能加 X 锁**。

锁的兼容关系如下：



2. 意向锁

InnoDB 支持 **多粒度锁 (multiple granularity locking)**，它允许 **行级锁** 与 **表级锁** 共存，而**意向锁** 就是其中的一种 表锁。

在存在行级锁和表级锁的情况下，事务 T 想要对表 A 加 X 锁，就需要先检测是否有其它事务对表 A 或者表 A 中的任意一行加了锁，那么就需要对表 A 的每一行都检测一次，这是非常耗时的。

意向锁在原来的 X/S 锁之上引入了 IX/IS，IX/IS 都是**表锁**，用来表示一个事务想要在表中的某个数据行上加 X 锁或 S 锁。

- 意向共享锁 (Intention Shared lock, IS)：事务有意向对表中的某些行加共享锁 (S锁)

```
-- 事务要获取某些行的 S 锁，必须先获得表的 IS 锁。  
SELECT column FROM table ... LOCK IN SHARE MODE;  
复制代码
```

- 意向排他锁 (Intention Exclusive lock, IX)：事务有意向对表中的某些行加排他锁 (X锁)

```
-- 事务要获取某些行的 X 锁，必须先获得表的 IX 锁。  
SELECT column FROM table ... FOR UPDATE;  
复制代码
```

即：意向锁是有数据引擎自己维护的，用户无法手动操作意向锁，在为数据行加共享 / 排他锁之前，InnoDB 会先获取该数据行所在在数据表的对应意向锁。

如果另一个任务试图在该表级别上应用共享或排它锁，则受到由第一个任务控制的表级别意向锁的阻塞。第二个任务在锁定该表前不必检查各个页或行锁，而只需检查表上的意向锁。

通过引入意向锁，事务 T 想要对表 A 加 X 锁，只需要先检测是否有其它事务对表 A 加了 X/IX/S/IS 锁，如果加了就表示有其它事务正在使用这个表或者表中某一行的锁，因此事务 T 加 X 锁失败。

各种锁的兼容关系如下：

这里的排他X / 共享S锁指的都是表锁！！！意向锁不会与行级的共享S / 排他X锁互斥！！！！

意向锁兼容关系

CyC

	X	IX	S	IS
X	⊘	⊘	⊘	⊘
IX	⊘		⊘	
S	⊘	⊘		
IS	⊘			

CyC2018

解释如下：

- 任意 IS/IX 锁之间都是兼容的，因为它们只表示想要对表加锁，而不是真正加锁；
- 这里兼容关系针对的是表级锁，而表级的 IX 锁和行级的 X 锁兼容，两个事务可以对两个数据行加 X 锁。（事务 T_1 想要对数据行 R_1 加 X 锁，事务 T_2 想要对同一个表的数据行 R_2 加 X 锁，两个事务都需要对该表加 IX 锁，但是 IX 锁是兼容的，并且 IX 锁与行级的 X 锁也是兼容的，因此两个事务都能加锁成功，对同一个表中的两个数据行做修改。）

例如：

事务 A 获取了表中某一行的排他锁,事务 B 想要获取表的共享锁：

因为**共享锁与排他锁互斥**，所以事务 B 在视图对表加共享锁的时候，必须保证：

- 当前没有其他事务持有表的排他锁。
- 当前没有其他事务持有表中**任意一行**的排他锁。

为了检测是否满足第二个条件，事务 B 必须在确保表不存在任何排他锁的前提下，**去检测表中的每一行是否存在排他锁**。很明显这是一个效率很差的做法，但是有了**意向锁**之后，情况就不一样了：

在事务A获取排它锁后,表中存在两把锁：表上的**意向排他锁IX**与数据行上的**排他锁X**，此时，事务 B 想要获取表的共享锁，事务B检测到事务 A 持有 表的**IX**锁，就可以得知事务A 必然持有该表中某些数据行的**排他锁**，那么事务 B 对表的加锁请求就会被排斥（阻塞），而**无需去检测表中的每一行数据是否存在排他锁**。

意向锁的并发性

意向锁不会与行级的共享 S/ 排他锁X互斥！！

正因为如此，意向锁并不会影响到**多个事务对不同数据行加排他锁时的并发性**（不然我们直接用普通的表锁就行了）。

在事务B因为检测到IX锁而被排斥后，事务C也想获取表中某一行的**排他锁**：

1. 事务 C 申请表的**意向排他锁**。

2. 事务 C 检测到 事务 A 持有表的意向排他IX锁。
3. 因为意向锁之间并不互斥，所以 事务 C 获取到了表的意向排他锁。
4. 因为C要修改的数据行上不存在任何排他锁，最终事务 C成功获取到了该数据行上的排他锁。

总结

1. InnoDB 支持 多粒度锁，特定场景下，行级锁可以与表级锁共存。
2. 意向锁之间互不排斥，但除了 IS 与 S 兼容外，意向锁会与 共享锁 / 排他锁 互斥。
3. IX, IS是表级锁，不会和行级的X, S锁发生冲突。只会和表级的X, S发生冲突。
4. 意向锁在保证并发性的前提下，实现了 行锁和表锁共存 且 满足事务隔离性 的要求。

封锁协议

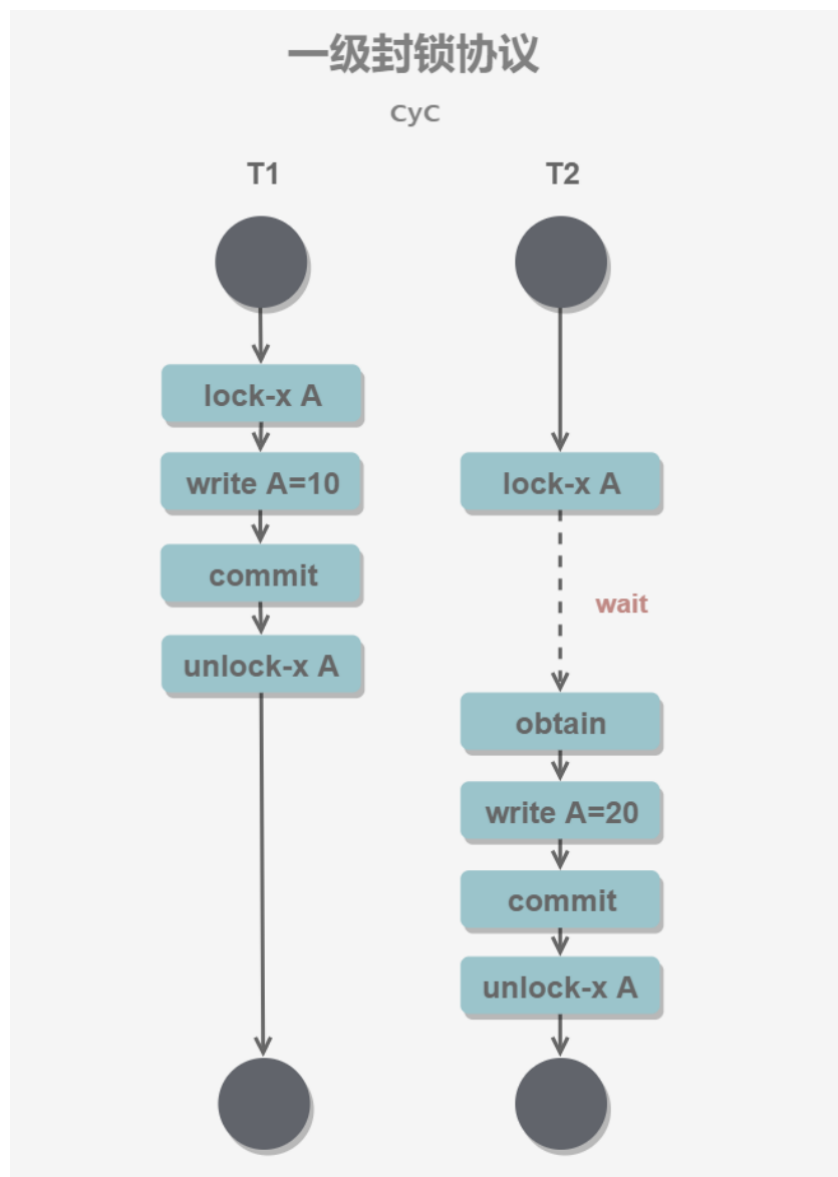
1. 三级封锁协议

一级封锁协议

事务 T 要修改数据 A 时必须加 X 锁，直到 T 结束才释放锁。

可以解决丢失修改问题，因为不能同时有两个事务对同一个数据进行修改，那么事务的修改就不会被覆盖。

但是由于没有对读做任何限制,仍然会出现读脏,不可重复读,幻读等问题

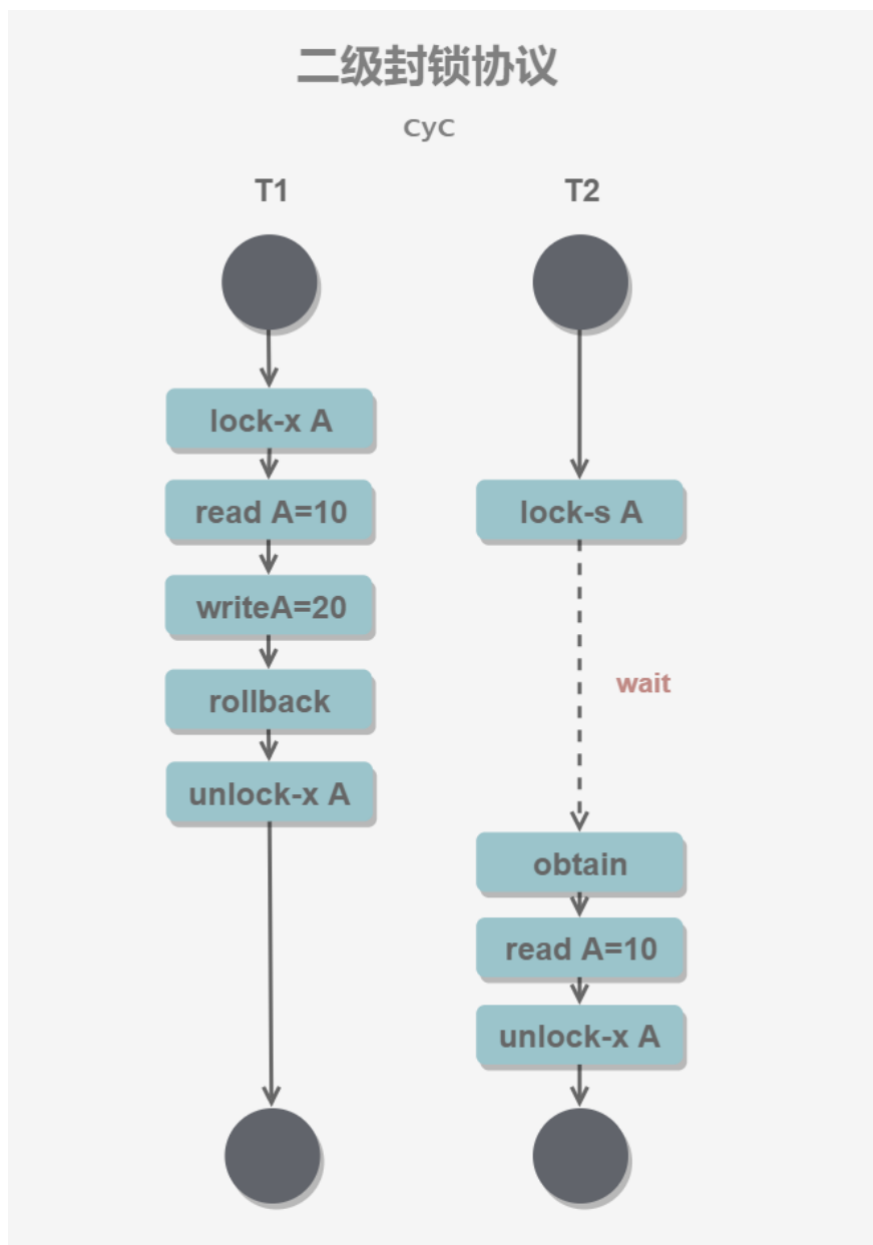


二级封锁协议

在一级的基础上，要求读取数据 A 时必须加 S 锁，读取完马上释放 S 锁。

可以解决读脏数据问题，因为如果一个事务在对数据 A 进行修改，根据 1 级封锁协议，会加 X 锁，那么就不能再加 S 锁了，也就是不会读入数据。

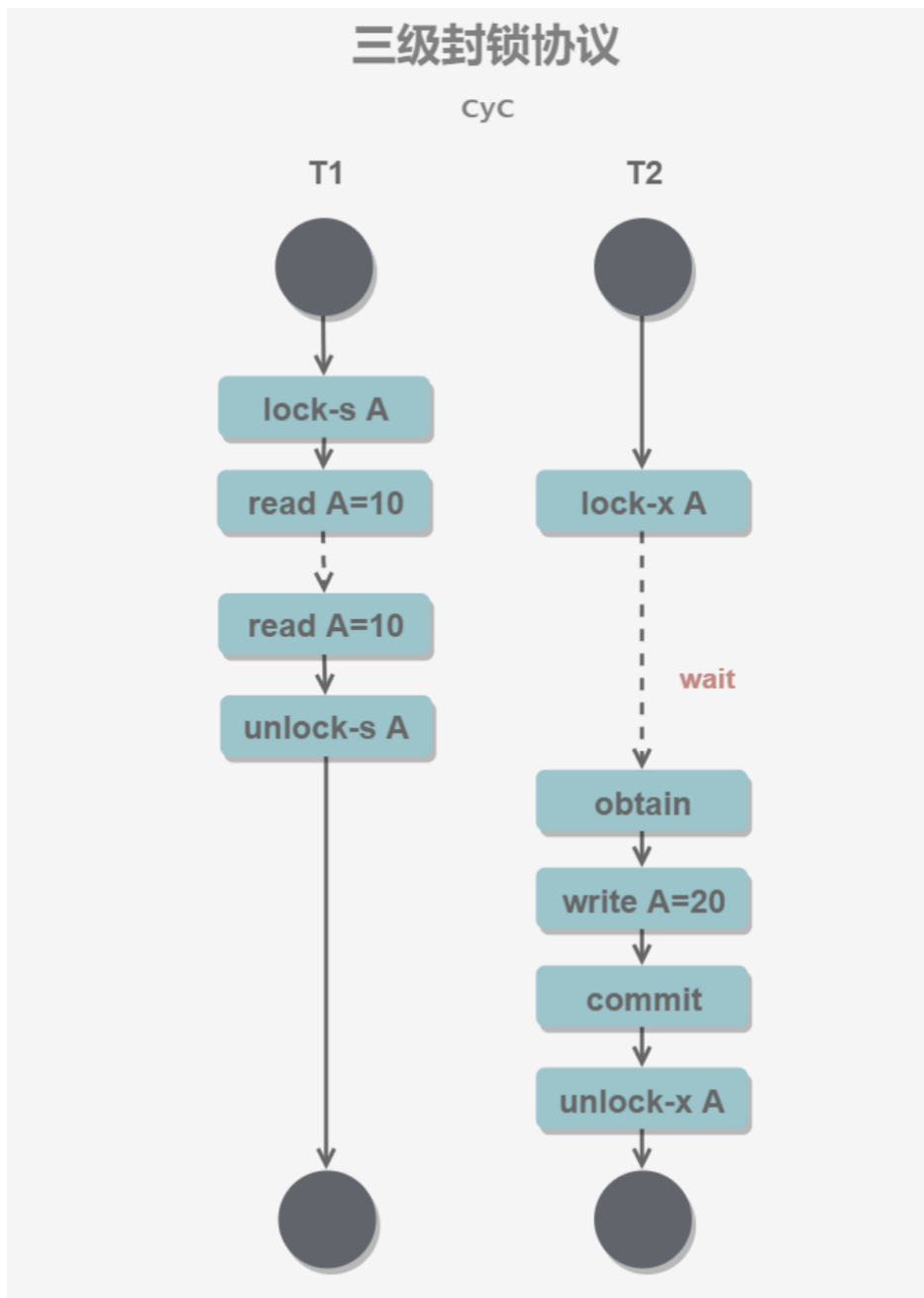
由于读之后马上就释放S锁,所以仍然会有不可重复读和幻读问题



三级封锁协议

在二级的基础上，要求读取数据 A 时必须加 S 锁，直到事务结束了才能释放 S 锁。

可以解决不可重复读的问题，因为读 A 时，其它事务不能对 A 加 X 锁，从而避免了在读的期间数据发生改变。



2. 两段锁协议

加锁和解锁分为两个阶段进行。

可串行化调度是指，通过并发控制，使得并发执行的事务结果与某个串行执行的事务结果相同。串行执行的事务互不干扰，不会出现并发一致性问题。

事务遵循两段锁协议是保证可串行化调度的充分条件。例如以下操作满足两段锁协议，它是可串行化调度。

```
lock-x(A) ... lock-s(B) ... lock-s(C) ... unlock(A) ... unlock(C) ... unlock(B)
```

但不是必要条件，例如以下操作不满足两段锁协议，但它还是可串行化调度。

```
lock-x(A) ... unlock(A) ... lock-s(B) ... unlock(B) ... lock-s(C) ... unlock(C)
```

MySQL 隐式与显示锁定

MySQL 的 InnoDB 存储引擎采用两段锁协议，会根据隔离级别在需要的时候自动加锁，并且所有的锁都是在同一时刻被释放，这被称为隐式锁定。

InnoDB 也可以使用特定的语句进行显示锁定：

```
SELECT ... LOCK IN SHARE MODE;  
SELECT ... FOR UPDATE;
```

四、隔离级别

并发控制可以通过封锁来实现，但是封锁操作需要用户自己控制，相当复杂。数据库管理系统提供了事务的隔离级别，让用户以一种更轻松的方式处理并发一致性问题。

未提交读（READ UNCOMMITTED）

事务中的修改，即使没有提交，对其它事务也是可见的。

提交读（READ COMMITTED）

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。**解决读脏问题**

可重复读（REPEATABLE READ）

保证在同一个事务中多次读取同一数据的结果是一样的。**解决不可重复读问题**

可串行化（SERIALIZABLE）

强制事务串行执行，这样多个事务互不干扰，不会出现并发一致性问题。**解决幻读问题**

该隔离级别需要加锁实现，因为要使用加锁机制保证同一时间只有一个事务执行，也就是保证事务串行执行。

隔离级别能解决的并发一致性问题

CyC

	脏读	不可重复读	幻影读
未提交读	×	×	×
提交读	✓	×	×
可重复读	✓	✓	×
可串行化	✓	✓	✓

CyC2018

五、多版本并发控制

多版本并发控制（Multi-Version Concurrency Control, MVCC）是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。而未提交读隔离级别总是读取最新的数据行，要求很低，无需使用 MVCC。可串行化隔离级别需要对所有读取的行都加锁，单纯使用 MVCC 无法实现。

基本思想

在封锁一节中提到，加锁能解决多个事务同时执行时出现的并发一致性问题。在实际场景中读操作往往多于写操作，因此又引入了读写锁来避免不必要的加锁操作，例如读和读没有互斥关系。读写锁中读和写操作仍然是互斥的，而 MVCC 利用了多版本的思想，写操作更新最新的版本快照，而读操作去读旧版本快照，没有互斥关系，这一点和 CopyOnWrite 类似。

在 MVCC 中事务的修改操作（DELETE、INSERT、UPDATE）会为数据行新增一个版本快照。

脏读和不可重复读最根本的原因是事务读取到其它事务未提交的修改。在事务进行读取操作时，为了解决脏读和不可重复读问题，MVCC 规定只能读取已经提交的快照。当然一个事务可以读取自身未提交的快照，这不算是脏读。

版本号

- 系统版本号 SYS_ID：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。
- 事务版本号 TRX_ID：事务开始时的系统版本号。

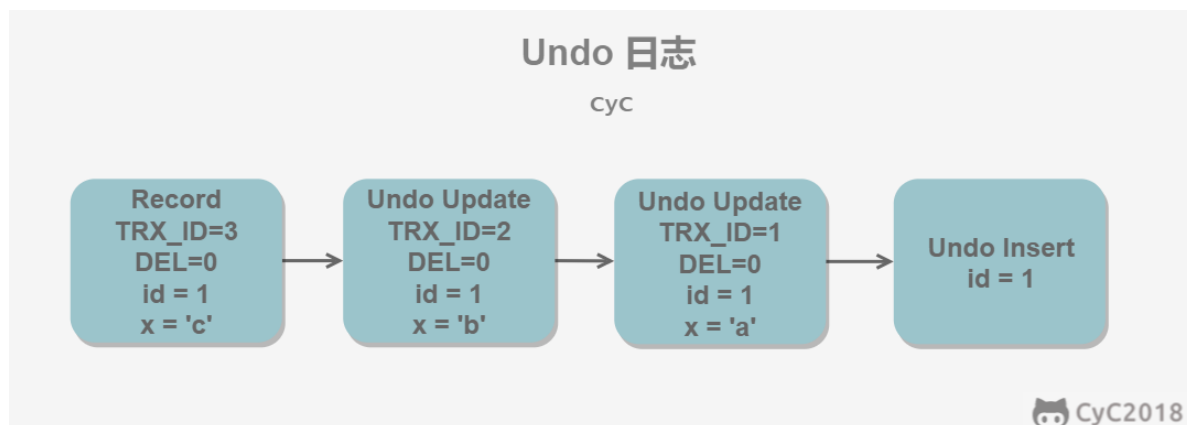
Undo 日志

MVCC 的多版本指的是多个版本的快照，快照存储在 Undo 日志中，该日志通过回滚指针 ROLL_PTR 把一个数据行的所有快照连接起来。

例如在 MySQL 创建一个表 t，包含主键 id 和一个字段 x。我们先插入一个数据行，然后对该数据行执行两次更新操作。

```
INSERT INTO t(id, x) VALUES(1, "a");
UPDATE t SET x="b" WHERE id=1;
UPDATE t SET x="c" WHERE id=1;
```

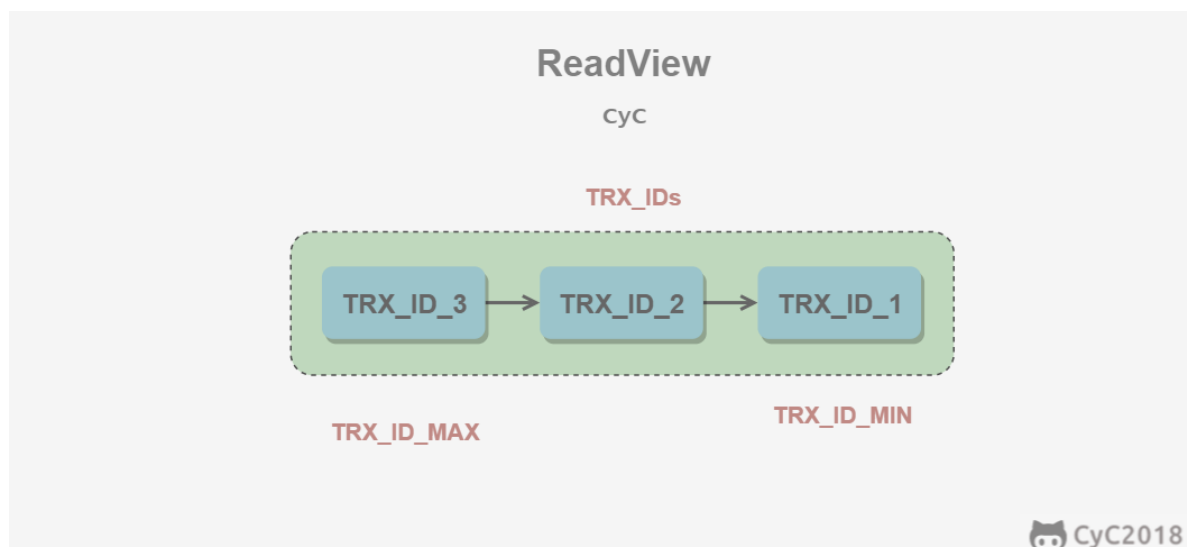
因为没有使用 `START TRANSACTION` 将上面的操作当成一个事务来执行，根据 MySQL 的 AUTOCOMMIT 机制，每个操作都会被当成一个事务来执行，所以上面的操作总共涉及到三个事务。快照中除了记录事务版本号 TRX_ID 和操作之外，还记录了一个 bit 的 DEL 字段，用于标记是否被删除。



INSERT、UPDATE、DELETE 操作会创建一个日志，并将事务版本号 TRX_ID 写入。DELETE 可以看成是一个特殊的 UPDATE，还会额外将 DEL 字段设置为 1。

ReadView

MVCC 维护了一个 ReadView 结构，主要包含了当前系统未提交的事务列表 **TRX_IDs** {**TRX_ID_1**, **TRX_ID_2**, ...}, 还有该列表的最小值 **TRX_ID_MIN** 和 **TRX_ID_MAX**。



在进行 SELECT 操作时，根据数据行快照的 TRX_ID 与 TRX_ID_MIN 和 TRX_ID_MAX 之间的关系，从而判断数据行快照是否可以使用：

- $TRX_ID < TRX_ID_MIN$ ，表示该数据行快照是在当前所有未提交事务之前进行更改的，因此可以使用。
- $TRX_ID > TRX_ID_MAX$ ，表示该数据行快照是在事务启动之后被更改的，因此不可使用。
- $TRX_ID_MIN \leq TRX_ID \leq TRX_ID_MAX$ ，需要根据隔离级别再进行判断：
 - 提交读：如果 TRX_ID 在 TRX_IDs 列表中，表示该数据行快照对应的事务还未提交，则该快照不可使用。否则表示已经提交，可以使用。
 - 可重复读：都不可以使用。因为如果可以使用的話，那么其它事务也可以读到这个数据行快照并进行修改，那么当前事务再去读这个数据行得到的值就会发生改变，也就是出现了不可重复读问题。

在数据行快照不可使用的情况下，需要沿着 Undo Log 的回滚指针 ROLL_PTR 找到下一个快照，再进行上面的判断。

快照读与当前读

1. 快照读

MVCC 的 SELECT 操作是快照中的数据，不需要进行加锁操作。

```
SELECT * FROM table ...;
```

2. 当前读

MVCC 其它会对数据库进行修改的操作（INSERT、UPDATE、DELETE）需要进行加锁操作，从而读取最新的数据。可以看到 MVCC 并不是完全不用加锁，而只是避免了 SELECT 的加锁操作。

```
INSERT;  
UPDATE;  
DELETE;
```


在进行 SELECT 操作时，可以强制指定进行加锁操作。以下第一个语句需要加 S 锁，第二个需要加 X 锁。

```
SELECT * FROM table WHERE ? lock in share mode;
SELECT * FROM table WHERE ? for update;
```

六、Next-Key Locks

Next-Key Locks 是 MySQL 的 InnoDB 存储引擎的一种锁实现。

MVCC 不能解决幻影读问题，Next-Key Locks 就是为了解决这个问题而存在的。在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

行锁在 InnoDB 中是基于索引实现的，所以一旦某个加锁操作没有使用索引，那么该锁就会退化为表锁。

Record Locks 记录锁

顾名思义，记录锁就是为某行记录加锁，它封锁该行的索引记录（锁定一个记录上的索引，而不是记录本身）

如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚簇索引，因此 Record Locks 依然可以使用。

Gap Locks 间隙锁

间隙锁基于非唯一索引，它锁定一段范围内的索引记录。间隙锁基于下面将会提到的 Next-Key Locking 算法，请务必牢记：使用间隙锁锁住的是一个区间，而不仅仅是这个区间中的每一条数据。

锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就不能在 t.c 中插入 15, 因为 10-20 之间的记录都会被锁住, 但是 10 和 20 两条记录并不会被锁住。

```
SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE;
```

Next-Key Locks 临键锁

Next-Key 可以理解作为一种特殊的间隙锁，也可以理解作为一种特殊的算法。通过临键锁可以解决幻读的问题。每个数据行上的非唯一索引列上都会存在一把临键锁，当某个事务持有该数据行的临键锁时，会锁住一段左开右闭区间的数据。需要强调的一点是，InnoDB 中行级锁是基于索引实现的，临键锁只与非唯一索引列有关，在唯一索引列（包括主键列）上不存在临键锁。

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录上的索引，也锁定索引之间的间隙。它锁定一个前开后闭区间，例如一个索引包含以下值：10, 11, 13, and 20，那么就需要锁定以下区间：

```
(-∞, 10]
(10, 11]
(11, 13]
(13, 20]
(20, +∞)
```

七、关系数据库设计理论

函数依赖

记 $A \rightarrow B$ 表示 A 函数决定 B，也可以说 B 函数依赖于 A。

如果 $\{A_1, A_2, \dots, A_n\}$ 是关系的一个或多个属性的集合，该集合函数决定了关系的其它所有属性并且是**最小**的，那么该集合就称为键码(也就是联合主键-)。

对于 $A \rightarrow B$ ，如果能找到 A 的真子集 A' ，使得 $A' \rightarrow B$ ，那么 $A \rightarrow B$ 就是部分函数依赖，否则就是完全函数依赖。

对于 $A \rightarrow B, B \rightarrow C$ ，则 $A \rightarrow C$ 是一个传递函数依赖。

异常

以下的学生课程关系的函数依赖为 $\{Sno, Cname\} \rightarrow \{Sname, Sdept, Mname, Grade\}$ ，键码为 $\{Sno, Cname\}$ 。也就是说，**确定学生和课程之后，就能确定其它信息**。

StuNo	StuName	StuDept	ManagerName	CourseName	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

不符合范式的关系，会产生很多异常，主要有以下四种异常：

- 冗余数据：例如 学生-2 出现了两次。
- 修改异常：修改了一个记录中的信息，但是另一个记录中相同的信息却没有被修改。
- 删除异常：删除一个信息，那么也会丢失其它信息。例如删除了 课程-1 需要删除第一行和第三行，那么 学生-1 的信息就会丢失。
- 插入异常：例如想要插入一个学生的信息，如果这个学生还没选课，那么就无法插入。

范式

范式理论是为了解决以上提到四种异常。

高级别范式的依赖于低级别的范式，1NF 是最低级别的范式。

1. 第一范式 (1NF)

属性不可分。

2. 第二范式 (2NF)

每个非主属性完全函数依赖(也包括传递依赖)于键码。

可以通过分解来满足。

分解前

Sno	Sname	Sdept	Mname	Cname	Grade
1	学生-1	学院-1	院长-1	课程-1	90
2	学生-2	学院-2	院长-2	课程-2	80
2	学生-2	学院-2	院长-2	课程-1	100
3	学生-3	学院-2	院长-2	课程-2	95

以上学生课程关系中，{Sno, Cname} 为键码，有如下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname
- Sno, Cname-> Grade

Grade 完全函数依赖于键码，它没有任何冗余数据，每个学生的每门课都有特定的成绩。

Sname, Sdept 和 Mname 都部分依赖于键码，当一个学生选修了多门课时，这些数据就会出现多次(例如上表中的Sno为2的学生)，造成大量冗余数据。

分解后

关系-1

Sno	Sname	Sdept	Mname
1	学生-1	学院-1	院长-1
2	学生-2	学院-2	院长-2
3	学生-3	学院-2	院长-2

有以下函数依赖：

- Sno -> Sname, Sdept
- Sdept -> Mname

关系-2

Sno	Cname	Grade
1	课程-1	90
2	课程-2	80
2	课程-1	100
3	课程-2	95

有以下函数依赖：

- Sno, Cname -> Grade

3. 第三范式 (3NF)

非主属性不传递函数依赖于键码。

上面的 关系-1 中存在以下传递函数依赖：

- Sno -> Sdept -> Mname

可以进行以下分解：

关系-11

Sno	Sname	Sdept
1	学生-1	学院-1
2	学生-2	学院-2
3	学生-3	学院-2

关系-12

Sdept	Mname
学院-1	院长-1
学院-2	院长-2

八、ER 图

Entity-Relationship，有三个组成部分：实体、属性、联系。

用来进行关系型数据库系统的概念设计。

实体的三种联系

包含一对一，一对多，多对多三种。

- 如果 A 到 B 是一对多关系，那么画个带箭头的线段指向 B；
- 如果是一对一，画两个带箭头的线段；
- 如果是多对多，画两个不带箭头的线段。

下图的 Course 和 Student 是一对多的关系。

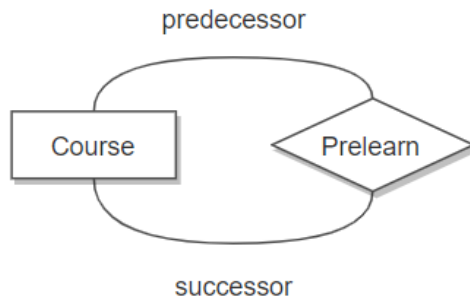


 CyC2018

表示出现多次的关系

一个实体在联系出现几次，就要用几条线连接。

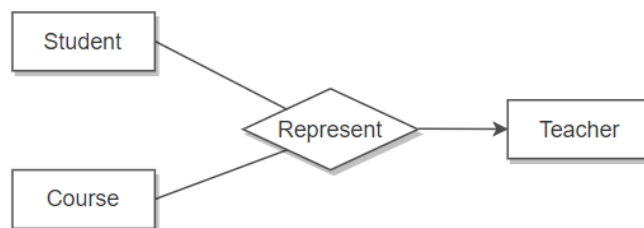
下图表示一个课程的先修关系，先修关系出现两个 Course 实体，第一个是先修课程，后一个是后修课程，因此需要用两条线来表示这种关系。



CyC2018

联系的多向性

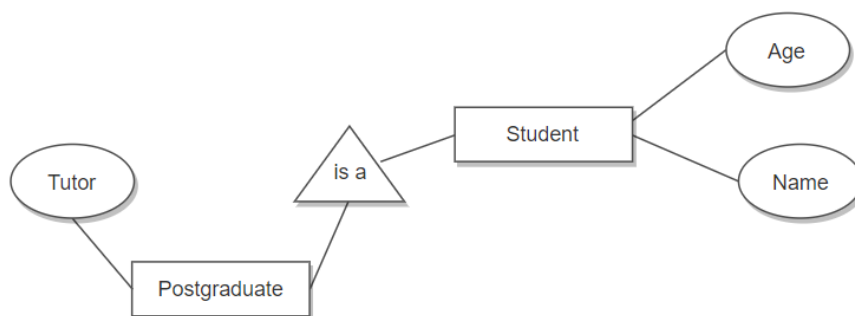
虽然老师可以开设多门课，并且可以教授多名学生，但是对于特定的学生和课程，只有一个老师教授，这就构成了一个三元联系。



CyC2018

表示子类

用一个三角形和两条线来连接类和子类，与子类有关的属性和联系都连到子类上，而与父类和子类都有关的连到父类上。

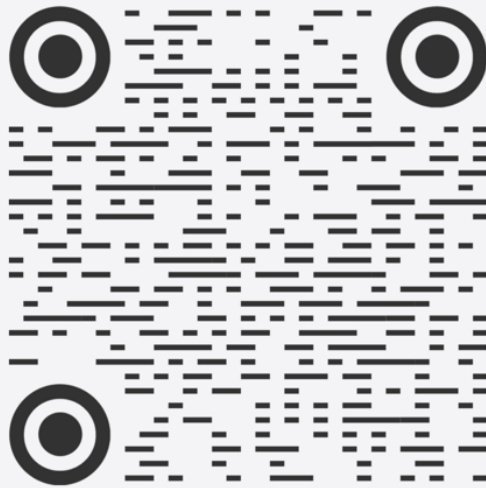


CyC2018

参考资料

- AbrahamSilberschatz, HenryF.Korth, S.Sudarshan, 等. 数据库系统概念 [M]. 机械工业出版社, 2006.
- 施瓦茨. 高性能 MySQL(第3版)[M]. 电子工业出版社, 2013.
- 史嘉权. 数据库系统概论[M]. 清华大学出版社有限公司, 2006.
- [The InnoDB Storage Engine](#)
- [Transaction isolation levels](#)
- [Concurrency Control](#)
- [The Nightmare of Locking, Blocking and Isolation Levels!](#)
- [Database Normalization and Normal Forms with an Example](#)

- [The basics of the InnoDB undo logging and history system](#)
- [MySQL locking for the busy web developer](#)
- [深入浅出 MySQL 和 InnoDB](#)
- [InnoDB 中的事务隔离级别和锁的关系](#)



公众号 CyC2018
帮助你快速学习成长
并拿到心仪的 Offer
回复 CyC 即可领取学习资料