# 数据库_系统原理

## 事务

### 概念

- 事务 是指将一组操作 视为一个 不可分割的 逻辑单元，使其满足 ACID 特性，可以通过 commit 提交事务，也可以通过 rollback 回滚事务！
- Mysql 默认采用 **自动提交**模式 **AUTOCOMMIT**

  执行 SQL 语句后就会马上执行 commit 操作！如果要显示 开启一个事务，需要使用 begin 或 start transaction 。

  也就是说，如果不显示执行 start transaction 开启一个事务，那么每个查询操作都会被当作一个事务并自动提交！
- 可以执行*set autocommit = 0 禁止当前会话自动提交*！此时，所有查询操作都是在一个事务中，直到显示地执行 COMMIT 或者 ROLLBACK 回滚，则该事务结束！

### ACID 特性

#### 原子性

- 原子性（Atomic）

  事务是一个不可分割的工作单元，事务中的所有操作，要么都成功提交，要么都失败回滚！
- 事务的回滚可用 **回滚日志（Undo Log）**来实现，日志中**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！

#### 一致性

- 一致性（Consistency）

  数据库 总是从一个 一致性状态 转移到 另一个一致性状态 —— 类似 **能量守恒**！

  ```
  例如：银行转账 A账户 -> B账户
          A账户 ：转出 1000     B账户：必须收到 1000 才能算数据保持一致
  ```

#### 隔离性

- 隔离性（Isolation）

  一个事务 所做的修改，在最终提交之前，对其他事务不可见

  ```
  例如：银行转账 A -> B，若 A 有 100 元，现在需要向 B 转账 20 元！
  则，事务1 对 A 扣款后，在向 B 汇款之前，如果又来了一个转账事务2，则，事务2 看不见 事务1 扣除的 20 元！
  ```

#### 持久性

- 持久性（Durability）

  一旦事务提交，事务所做的修改就会保存到数据库中，即使数据库崩溃，事务执行的结果也不丢失！

  系统发生崩溃后，可以用 重做日志（Redo Log）进行恢复，从而持久化事务执行结果！

```
重做日志（Redo Log）记录的是 数据页 的 物理修改！
回滚日志（Undo Log）记录的是 数据库 的 修改操作！
```

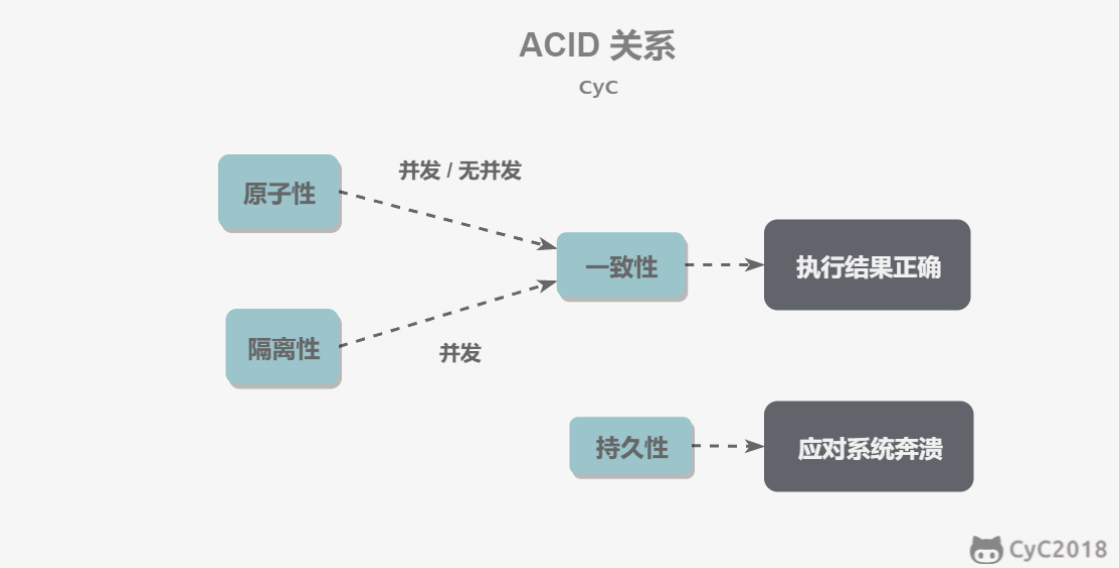
### ACID 关系

- 原子性 和 隔离性 是 一致性的基础！

  单线程情况下，事务串行执行，此时只要能保证事务是原子性不可分割，就能保证一致性！

  多线程情况下，事务并行执行，此时除了保证原子性，还需要保证 隔离性，才能满足一致性！

- 持久性 是为了 应对 系统崩溃的 状况！

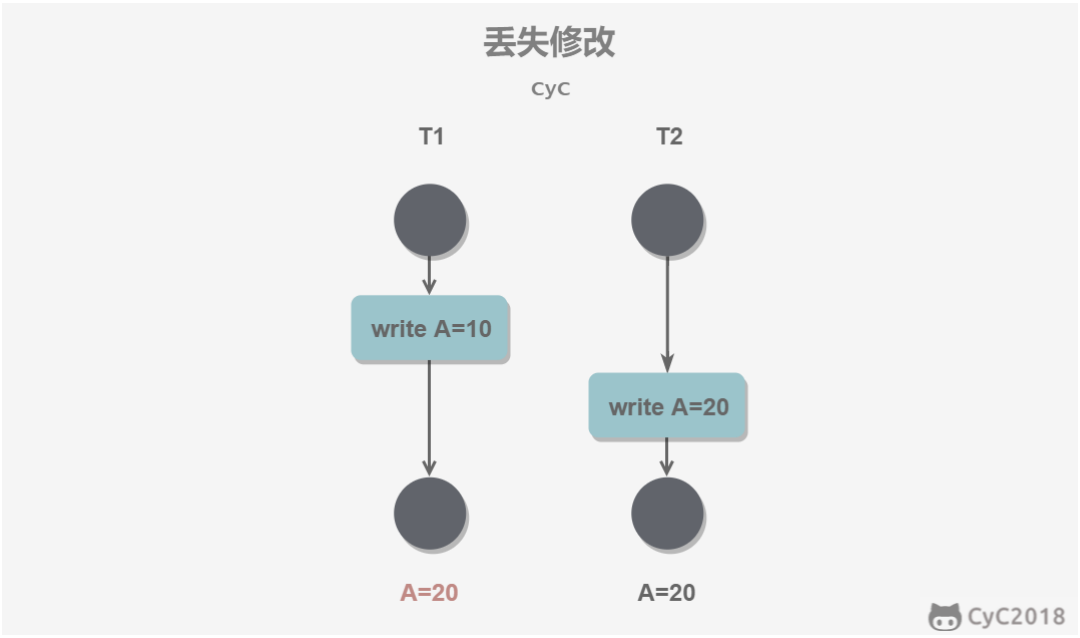



### 并发一致性

- 在并发情况下，事务的隔离性很难保证，因此会出现并发一致性问题！
- 常见有 丢失修改、脏读、不可重读、幻读 等四个并发一致性问题！

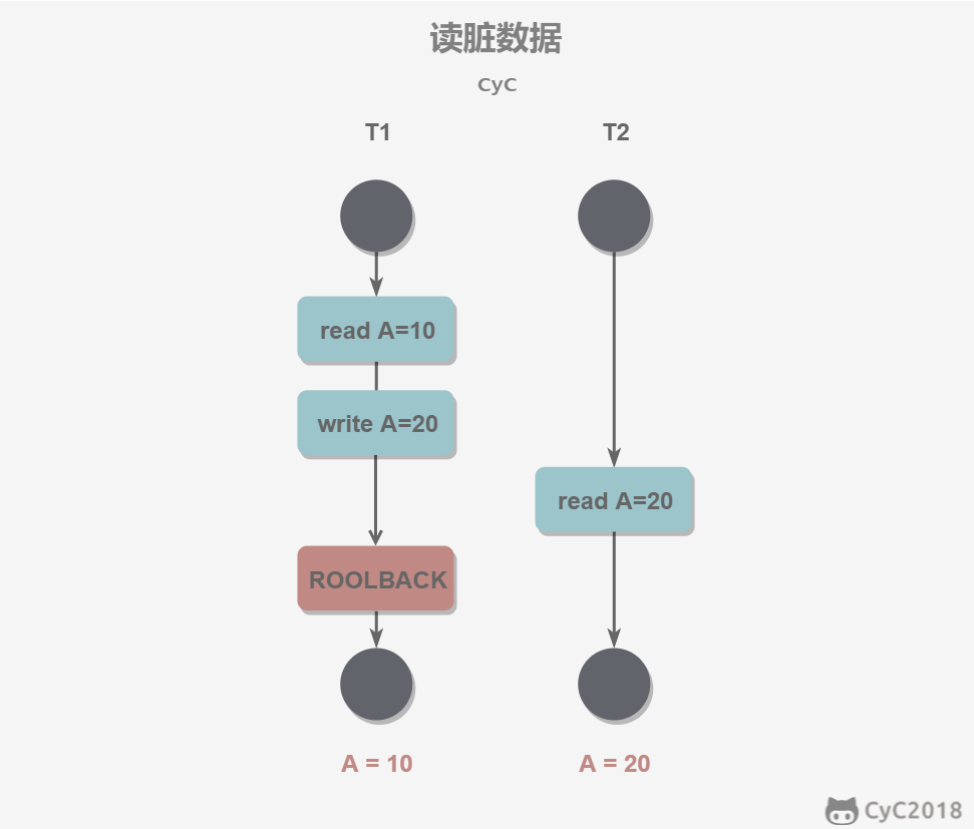
### 丢失修改

- 在 事务A 未完成修改之前，事务B 获取同一个数据进行修改，事务A 修改完成之后，会被事务B 的修改结果覆盖！



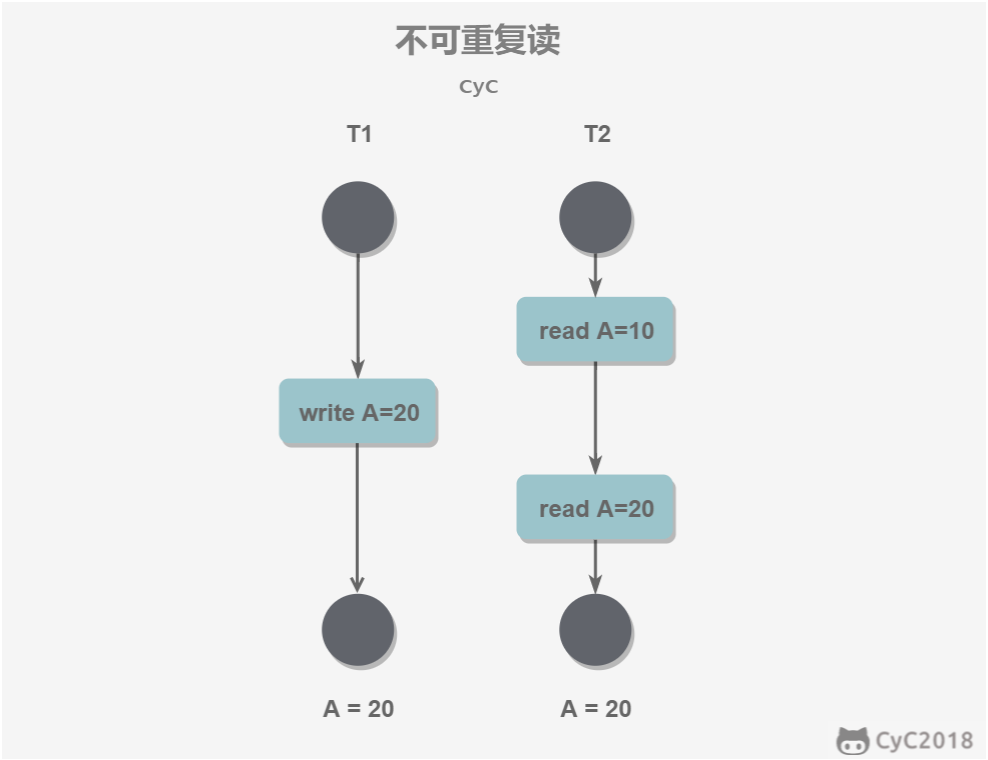

### 脏读

- 事务 B 读取到 事务A 回滚之前修改的数据，则事务 B 读取到了脏数据！



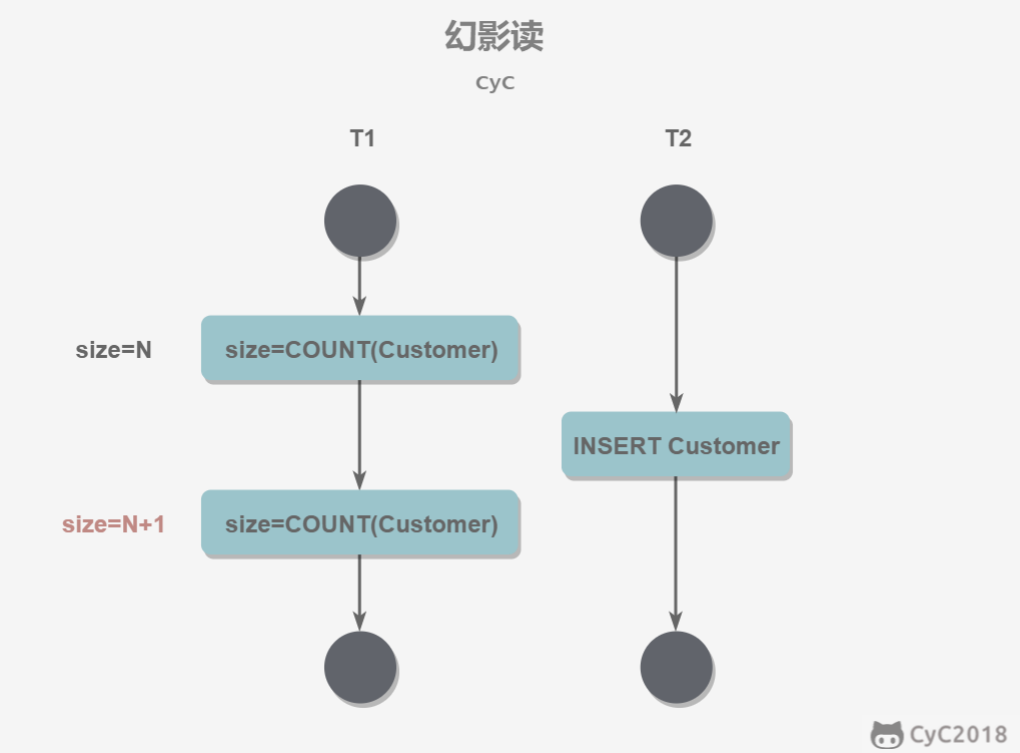

**不可重读**

- 事务A 连续两次 获取相同的资源，两次获取的结果不同！这期间，数据被其他事务修改过！




**幻读**

- 事务 A 执行相同的范围查询，两次得出的结果集不相同！有其他事务，对该范围的数据执行了 插入操作！




### 总结

- “更新丢失” 应该在 应用程序中完全避免，而不是单靠数据库的事务控制器解决！

  应用程序对要更新的数据加上必要的锁来避免更新丢失问题，因此，防止丢失更新是应用的责任

  （在程序中加上 互斥锁 Synchronized/Lock 或者 乐观锁CAS）
- “脏读、不可重读、幻读”，其实都是数据库读一致性问题，必须由数据库提供一定的事务隔离机制来解决。

### 事务隔离级别

- 产生并发一致性问题，主要是因为破坏了 事务的 隔离性！为了解决并发一致性问题，数据库中定义了 四种事务的隔离级别：

  级别由低到高：读取未提交、读取已提交、可重复读、串行化！

| 读数据一致性及允许的并发副作用 / 隔离级别 | 读数据一致性 | 脏读 | 不可重复读 | 幻读 |
| --- | --- | --- | --- | --- |
| 未提交读（Read uncommitted） | 最低级别，只能保证不读取物理上损坏的数据 | 是 | 是 | 是 |
| 已提交读（Read committed） | 语句级 | 否 | 是 | 是 |
| 可重复读（Repeatable read） | 事务级 | 否 | 否 | 是 |
| 可序列化（Serializable） | 最高级别，事务级 | 否 | 否 | 否 |

- Mysql 默认的隔离级别为 可重复读！

### 读未提交

- 读未提交（Read Uncommitted）

  事务之间没有隔离性，事务中的修改，即使还未提交，对其他事务也是可见的！
- 优点

  这个级别，没有并发控制，相对来说性能较高！
- 缺点

当事务读取到未提交（已回滚）的数据，则是脏读！

#### 读已提交

- 读已提交（Read committed）

  事务提交之前，事务的修改对其他事务不可见！其他事务只能看到本事务提交后的结果！
- 优点：

  可避免脏读
- 缺点

  会出现 不可重读、幻读等并发一致性问题！

#### 可重复读

- 可重复读（Repeatable Read）

  该级别保证了事务多次读取同一个记录的结果是一致的。
- 优点

  解决了不可重读、脏读的问题。
- 缺点

  依旧不能解决 幻读问题

#### 串行化

- 串行化（Serializable）

  强制事务串行化执行，隔离级别最高，避免了幻读！
- 问题

  串行化会在读取的每一个行数据上都加锁，所以可能导致大量的超时和锁争用问题。
- 实际应用中，很少使用这个隔离级别，只有在非常需要确保数据的一致性而且能接受串行执行的情况下，才会考虑使用该级别！

  mysql 中的 innodb 并没有使用串行化级别解决幻读问题，而是通过 MVCC+Next-Key Locks 解决幻读！

#### 隔离级别实现

- 未提交读 级别 无需控制事务提交，直接读取当前行的最新状态，要求最低，不需要使用特殊的机制！
- MVCC 是 MySQL 的 InnoDB 实现 Read Committed 和 Repeatable Read 这两种隔离级别的一种方式！
- 可串行化隔离级别 需要对 查询范围内的行 全部加锁，单纯使用 MVCC 无法办到！

## 锁

- 事务的隔离性，主要是由锁来实现！

#### 锁粒度

- 锁粒度

  锁粒度分为 **行锁** 和 **表锁** 和 页面锁！

  锁粒度是在 `并发的性能 和 数据安全性 之间寻求平衡` ！

#### 表锁

- 是 mysql 中最基本的锁策略，锁粒度最大，会**锁定整张表**！
- 优点

  *开销小、加锁快、不会出现死锁（因为一次性获取了所需的所有资源）*
- 缺点

  *发生锁冲突的概率最大，并发度最低*！

  一个用户在对表执行写操作时，会阻塞其他用户的读写操作！

  只有当没有写操作时，其他读取的用户才能获得读锁，并且读锁之间互不阻塞！

#### 行锁

- 锁粒度最低，只会**锁住 一条记录**，非不是整张表！
- 优点

  *锁粒度最小，发生锁冲突的概率最低，并发度最高！*
- 缺点

  *开销大（频繁的加锁），加锁慢，会出现死锁！*

#### 页面锁

- 对页面加锁！
- 锁开销、锁定粒度介于表锁和行锁之间，并发度一般！
- 可能会出现死锁。

#### MySQL 的锁粒度

- MySQL 的锁机制较为简单，MySQL 的显著特点就是，不同的存储引擎支持不同的锁机制！
  - *MyISAM*、 MEMORY 存储引擎*采用了 表级锁*
  - BDB 存储引擎采用了 页面锁
  - **InnoDB 既支持 表级锁，也支持 行级锁，默认采用行级锁！**

#### 锁类型

- 数据库的**锁类型**分为：**共享锁（读锁）**、**排他锁（写锁）**
- **读锁** 又称为 共享锁（Shared），简写 S 锁！

  *读锁与读锁之间不会相互阻塞*，允许多个用户可以同时读取同一个资源！
- **写锁** 又称为 排他锁（Exclusive），简写 X 锁！

  *一个写锁 会 阻塞 其他的读锁和写锁，一个 写锁 或 读锁，也能阻塞其他的写锁*！

  写锁与写锁、写锁与读锁，之间相互阻塞！只有当写锁执行完毕，其他的读/写锁才能获取资源！
- 兼容性：

| | X | S |
|---|---|---|
| X | 不兼容 | 不兼容 |
| S | 不兼容 | 兼容 |

### InnoDB 的锁

#### 介绍

- 锁粒度上：**InnoDB 同时支持 表级锁和行级锁**！

  为允许 两种锁粒度 共存，实现高效的**多锁粒度机制**，InnoDB **引入了 表级的 意向锁**：*意向共享锁*、*意向排他锁*！

- 锁类型上：**读写锁**

#### 意向锁

- 作用：

  **在 表锁和行锁 共存的情况下，能高效的为表加上表锁，而不需要逐行记录排查！**

  > 在为某一行数据加锁前，尝试获取该表的意向锁，获取成功后，在表上标志 意向锁标识，表明该表中已经存在 X/S 的行级锁！
  > 当有其他事务想要对该 表加 S/X 锁时，就不需要逐行检测，而是直接查看 表上的意向锁标识！
  > 如果表上的意向锁标识 与 自己所加的锁冲突，则阻塞等待，否则对该表加锁！

- **意向共享锁（IS）**

  *表示 事务打算给 数据行加 行共享锁*！即：事务在给一个数据行 加 行共享锁 前必须先获得该表的 IS 锁！

- **意向排他锁（IX）**

  *表示 事务打算给 数据行加 行排他锁*！即：事务在给一个数据行 加 行排他锁 前必须先获得该表的 IX 锁！

#### InnoDB 锁兼容性

- 如果一个事务请求的锁模式 与 当前的锁模式兼容，InnoDB 就将请求的锁授予 该事务；反之，如果两者不兼容，该事务就要等待锁释放！

表20-6 InnoDB行锁模式兼容性列表

| 是否兼容 / 请求锁模式 / 当前锁模式 | X | IX | S | IS |
|---|---|---|---|---|
| X | 冲突 | 冲突 | 冲突 | 冲突 |
| IX | 冲突 | 兼容 | 冲突 | 兼容 |
| S | 冲突 | 冲突 | 兼容 | 兼容 |
| IS | 冲突 | 兼容 | 兼容 | 兼容 |

- 注意：

  **意向锁 在加 行级锁之前， InnoDB 自动加上的**，不需要用户干预！

### 封锁协议

#### 三级封锁协议

1. 一级封锁协议

   事务 A 修改某个数据之前，必须加上 排他锁（X），直到事务结束 才释放锁！

   > 解决：“丢失修改” 问题：
   > 因为同一个数据的两个 修改事务 在执行时间上，不可能有交叉！
   > 即：排他锁 和 排他锁 不兼容；不可能有两个事务同时对同一个数据进行修改！

2. 二级封锁协议

   在一级封锁协议的基础上，事务 A 读取某个数据之前，必须加上 共享锁（S），直到 数据读取结束，才释放锁！

```
解决："脏读"问题：
    因为同一个数据的 读取和修改事务 在执行时间上，不可能有交叉！
    即：排他锁 和 共享锁 不兼容；不可能出现 某个事务读取 数据的同时，该数据被其他事务修改！
```

3. 三级封锁协议

   在二级封锁协议的基础上，事务 A 读取某个数据之前，必须加上共享锁（S），直到 事务结束，才释放锁！

```
解决："不可重读"问题：
    因为共享锁的范围扩大到事务级别，在一个事务内，不可能出现其他事务对该数据的写操作！
    即：在一个事务内，多次读取同一个记录，其结果始终不会改变！
```

**两段锁协议**

- 两段锁：指每个事务的执行分为 加锁 和 解锁 分为两个阶段进行！

  也称为 生长阶段 和 衰退阶段！

- **加锁阶段：只封锁资源**

  事务对任何数据进行读、写操作之前，要先申请该资源的锁！

  读操作时 申请 S锁，写操作时 申请 X锁！如果加锁不成功，则事务进入等待状态，直到加锁成功才继续执行！

  因此：加锁阶段，并不要求一次性全部加锁！

- **解锁阶段：只解锁资源**

  当事务释放一个锁之后，事务进入解锁阶段，该阶段只能继续对 已加锁的资源 解锁，并且 不能再进行 加锁操作！

- 注意：

  1. 在对任何数据进行读、写操作之前，要申请并获得对该数据的封锁！
  2. 每个事务中，所有的加锁请求 都 先于所有的解锁请求！
  3. 两段协议可能产生死锁！

- *两段锁协议* vs *一次性封锁*

  - 一次性封锁，表示一次性获取所需资源的所有锁，否则不能继续执行，它破坏了 产生死锁的 "请求与保持条件"， 是防止死锁的策略！

    一次性封锁 遵循了 两段锁协议！

  - 而 两段锁协议 并不要求事务必须一次性将所有要使用的资源全部加锁，因此，两段锁协议可能产生死锁！

- 两段锁协议的 加锁和解锁 序列应如下：

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

- 可证明：*事务遵循 两段锁协议，是保证事务 可串行化调度的 充分条件！*

```
但 可串行化调度 时，并不一定 遵循两段锁协议。如下：
lock-x(A)...unlock(A)...lock-s(B)...unlock(B)...lock-s(C)...unlock(C)
```

**隐式/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 再需要的时候自动加锁，并且所有的锁都是在同一个时刻被释放，这是隐式锁定！

- InnoDB 存储引擎 支持显示锁定：

```
# ----------------------------------------------#
# 表锁

# 表锁 加读锁
local table(s) table_1 read [local];
# 表锁 加写锁
local table(s) table_1 write;
# 表锁 解锁方式
unlock table(s);
# ----------------------------------------------#
# 行锁

# 首先设定禁止自动提交
set autocommit=0;

# 行锁：共享锁（S）：lock in share mode
SELECT ... LOCK IN SHARE MODE;
# 行锁：排他锁（X）：for update
SELECT ... FOR UPDATE;

# 手动提交
commit;
# ----------------------------------------------#
```

## MVCC

- 数据库 的 锁机制能够 解决多事务执行时的并发一致性问题。在实际场景中，往往是 读多写少，并且读写锁互斥，此时，使用锁 不但会耗费系统资源，还会阻塞事务的执行，降低 MySQL 的性能！
- 于是，MySQL 采用无锁的并发控制机制 MVCC 解决 Read Committed 和 Repeatable Read 这两个隔离级别的**并发读一致性问题**！（MySQL 默认工作在 Repeatable Read 级别）

### 介绍

- MVCC（Multi Version Concurrency Control）多版本控制并发

  它为某每个行 维护了一个 行记录版本链，当某事务查询数据时，便从版本链中为其找出最合适的行记录版本！

  当某事修改行记录时，InnoDB 便将数据库中当前的 行记录 保存到该行的版链中，并在数据库中更新该行记录！

- MVCC 的实现机制

  MVCC 维护了 *事务ID、行记录的版本号、保存行记录版本链的 undo log 回滚日志* 以及 *维护当前活跃事务的 read view* ！

- InnoDB 的逻辑存储结构

  InnoDB 的数据保存在 表空间中，表空间中包含了多种 segment（段）：数据段、索引段、回滚段！

  - InnoDB 以 B+树 存储数据，其中，叶子节点（Leaf Node Segment）就是数据段，非叶子结点（Non-Leaf Node Segment）就是索引段！
  - 回滚段 用于 存储回滚日志（Undo log）。回滚日志中记录了 各个行的多个版本数据！

    用于 快照读 和 事务失败后的数据回滚。

    MySQL 会在 合适的时机清理 Undo log！

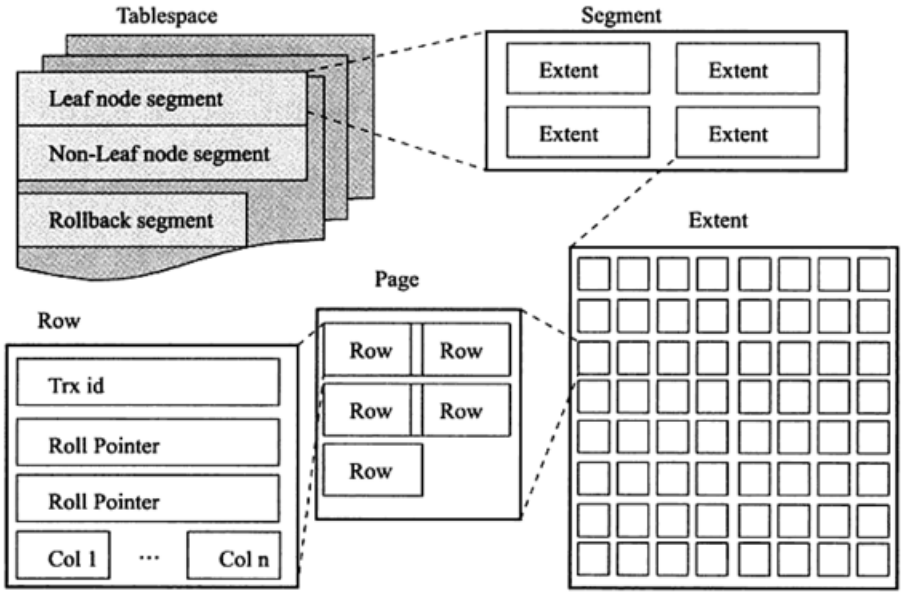



图 4-1　InnoDB 逻辑存储结构

**版本号**

- SYS_ID：系统版本号：每开启一个事务，该版本号就会自动递增！
- TRX_ID：**事务版本号**：事务当前的版本。事务开启时，SYS_ID 表示的版本号！

**表的隐藏字段**

- MySQL 默认为我们的 表 增加了 TRX_ID、ROLL_PTR 、Delete_Flag 等隐藏字段！
  1. **TRX_ID：行记录版本号**

     该行数据 到目前为止，最后 被修改的事务ID
  2. **ROLL_PTR：上一个行记录的地址**

     回滚指针，指向 RollBack Segment 中的 Undo Log 记录。

     该行数据被更新时，MySQL 会将修改之前的内容 记录到 undo log 中！
  3. Delete_Flag

     该行数据的删除标志位！

**Undo Log**

- **undo log（回滚日志）除了 事务失败回滚 之外，还用作 MVCC 行数据的多版本 记录！**
- Undo  Log 在 Rollback Segment 中被*细分为 insert undo log 和 update undo log*
  - insert undo log：仅仅用作事务回滚，一旦事务成功提交，该 log 就会被丢弃！
  - update undo log：被用作一致性读 和 事务回滚！

    当没有事务 会对该行数据 进行一致性读时，该行 log 被清理！
- *Undo log 的创建*

  每当 事务 更新/修改 行记录时，mysql 都会 复制一份数据副本，并将其保存到 undo log 中。

  同时，修改当前行的 回滚指针（DB_ROLL_PTR）指向 Undo Log 中旧的行记录！

  如此一来，undo log 便形成了该 行记录的 版本链！

**Read View**

- **Read View 用于判断数据行的可见性！**

在 innodb 中，创建一个新事务时，innodb 会为 当前系统中的活跃事务列表 创建一个 read view，该 read view 中保存了系统当前不应该被事务看见的其他事务 id 列表。

- innodb 正是**通过 read view 实现了不同隔离级别的 一致性读!**
- 当 事务读取某行记录时，innodb 会将当前行的 DB_TRX_ID 与 read view 做对比！判断改行是否可见，如果可见，则使用该行数据，否则 根据根据回滚指针 从 undo log 中查找可见的 行记录数据!

**MVCC 一致性读**

1. 当某个事务执行时，innodb 会为生成一个 read view，存放着当前的 依旧活跃的事务（还未提交的事务）
2. 当该事务 查询某行记录时，innodb 首先将 该行数据的 trx_id 与 read view 中的 事务 id 做对比!
    - 若 trx_id < read_view_min_id ，则该行记录 在 本事务执行之前 就已经被其他事务提交了，是可见的!
    - 若 trx_id > read_view_max_id ，则该行记录 在 本事务执行之后 才被其他新事务修改的，不可见!
    - 若 trx_id 大小 处于 read view 区间内，则遍历 read view 查看 trx_id 对应的事务 是否存在于 read view 中!
        1. 如果存在，则说明，该行记录被该事务修改后，还未提交 —— 不可见!
        2. 如果不存在，则说明，该行记录是被事务成功提交后的结果 —— 可见!
3. 若 该行记录不可见，则根据 行记录的 回滚指针，到 undo log 中遍历该行记录的版本链，依旧根据上述判断方法，找出最近最合适的可见版本!

**MVCC 隔离级别实现**

- 由 MVCC 的执行机制可知，判断行记录的可见性，主要取决于 read view 的更新状态!
    - 如果 read view 实时更新，实时记录此时的 活跃的 事务 id，那么 行记录的可见性 就能实时更新!
    - 如果 read view 从开始执行本事务时，一直不变，那么，行记录的可见性 就是一直不变的!

1. *Read Committed 读已提交*

    在**本事务的每次查询中，都会重新生成 read view**，记录最新的 活跃事务，此时，相对于上一次生成的 read view，已成功提交的事务，将不会出现在 本次的 read  view中!

    此时查询数据时，能读取到 该行记录最新提交的内容 —— 这也导致了，两次读取的内容不一致 —— 不可重读!

2. *Repeatable Read 可重复读*

    在**本次事务第一次 查询数据时，就生成一个 read view，直到事务执行结束**!

    它能保证，无论多少次查询数据，所对比的 read view 都是一样的，这也就保证了多次查询数据，得出的行记录版本都是一致的 —— 可重复读!

## InnoDB 行锁实现

**介绍**

- InnoDB 通过**给 索引 加锁**实现 行锁!

```
<!--
 · 注意：
    1. 如果查询条件中没有使用到索引，那么，将使用表锁！（生产环境要注意！！）

    2. 如果表中没有索引，InnoDB 会自动创建隐藏的聚族索引 来对记录加锁！

    3. MySQL 行锁是针对索引加锁的，不是针对记录加锁，因此即使访问不同的行记录，如果使用了相同的索引键，依旧是会出现锁冲突！
    ？？？？我测试的结果是：即便是使用了相同索引 查询不同行，也不会冲突!？？？？

    4. 当表有多个索引时，不同的事务可以使用不同的索引锁定不同的行，不论是使用主键索引、唯一索引 还是 普通索引，InnoDB 都会使用行锁对数据加锁！
-->
```

- InnoDB 的行锁有三种实现：

  *Record Lock*、*Gap Lock*、*Next-Key Lock*

**Record Lock**

- *Record Lock（记录锁）*

  锁定一个记录上的索引，而不是记录本身！

**Gap Lock**

- *Gap Lock（间隙锁）*

  锁定索引之间的间隙（行记录之间的间隙），但是不包含索引本身（不包括行记录本身）！

```
select * from table_1 where id between 10 and 20 for update;
# 则，其他事务 插入 id=15 时，将阻塞！
# 那么 其他事务是不是可以 更新 id=10 以及 id=20 呢？
```

**Next-Key Locks**

- MVCC 不能解决 幻读的问题，而 **Next-Key Locks 正是 MySQL 解决幻读的方法。**
- Record Lock 和 Gap Lock 的组合，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*的区间。

  例如：一个索引包含：10，11，13，20，那么它就锁定以下区间

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

  锁范围：满足条件的第一个记录 到 满足条件的最后一个记录的后一个记录（包括该记录）！

```
set autocommit=0;

# 以下 sql 检索同样的内容，uid=4 到 uid=11 的记录都会被返回！
select * from table_1 where uid >= 4 and uid < 12 for update;
select * from table_1 where uid > 3 and uid <= 11 for update;
select * from table_1 where uid between 4 and 11 for update;

# 在另个一 client 端执行以下内容时将会被阻塞
```

```
delete from table_1 where uid=12;
delete from table_1 where uid=11;
delete from table_1 where uid=4 ;

# 但是执行 uid=3 时不阻塞!
delete from table_1 where uid=3;
delete from table_1 where uid=13;

# 结果：除了锁定满足的条件，还会锁定的范围，还会延续到 查询结果的后一个记录!
# 例如：当表中存在 uid=[3,15] 并且 uid=[20,25]
select * from table_1 where uid between 4 and 15 for update;
# 执行上述结果后，uid=4 到 uid=20 的记录都会被锁定!
# 如果新增 uid=16 则阻塞!
```

## 快照读 vs 当前读

- **对于 update、delete、insert 等写操作，InnoDB 会自动给涉及的涉及的数据集加上 排他锁（Next-Key Locks）！**
- **对于普通的 select ，InnoDB 不会加 任何锁!**

### 快照读

- 单纯的 select 查询数据时，InnoDB 会根据 MVCC 的版本链查询合适的快照，而不需要加锁!

```
select * from table_1 ... ;
```

### 当前读

- 当执行 insert 、update、delete 等操作时，InnoDB **使用 Next-Key Locks 配合 MVCC，使其在 Repeatable Read 隔离级别下，解决 决幻读的问题**!
- 这是因为，*insert、update、delete 等都需要读取当前最新的数据*，而不是版本链中的快照!

  因此，MVCC 并不是完全不用加锁的，而只是避免了普通 select 的加锁操作!

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

- 在进行 select 操作时，可以强制 加锁:

```
# S 共享锁
select * from table where ... lock in share mode;
# X 排他锁
select * from table where ... for update;
```

## 死锁

### 介绍

- 死锁是指**两个或者多个事务在同一个资源上相互占用，并请求锁定对方占用的资源，导致相互等待的现象，若无外力作用，事务都无法推进下去**!

  当多个事务以不同的资源顺序锁定资源时，就可能产生死锁!
- 死锁发生后，只有 *将部分或者完全回滚其中一个事务，才能打破死锁*，大多数情况下只需要重新执行因死锁回滚的事务即可!

**死锁检测**

1. **超时机制**
   - 若多个事务相互等待，当等待时间超过设定的阈值，则通过 **FIFO 的方式，将其中一个事务回滚**，另外的事务就能继续执行下去！

     InnoDB 采用 **innodb_lock_wait_timeout 参数设定超时时间**！
   - 优点：简单
   - 缺点

     通过 FIFO 的方式回滚事务，可能会导致大量的系统开销！

     如果回滚的事务 所占的权重较大，即：该事务执行了大量的更新操作，占用了许多的 undo log，那么回滚这个事务相对于回滚其他事务，就会占用大量的系统时间，并且所有操作重新执行，也会大量的系统资源！
2. **wait-for graph（等待图）**

   相对于 超时检测方案，**wait-for graph 是一种更为主动的死锁检测**方式！

```
<!--
 · wait-for graph 需要维护两个 链表：
    1. 锁的信息链表
    2. 事务等待链表
 · 锁信息链表中存放的是 当前获取到该资源的 事务结点，以及当前正在等待资源的事务结点！
    通过将所有锁信息链表中，正在等待资源的事务结点，指向以获取该资源的事务结点，便能够形成一张事务等待图！
    InnoDB 将某事务阻塞等待资源之前，首先判断加入该事务结点之后，图中是否会产生回路，如果会，则将该事务回滚！

    wait-for graph 的死锁检测通常采用深度优先算法实现，在 InnoDB1.2 之前，都是采用 递归的方式，而从 1.2 版本开始采用了非递归的方式实现，从而提高了 InnoDB 的性能！
-->
```

- 如下例：

  锁信息链 + 事务等待链：

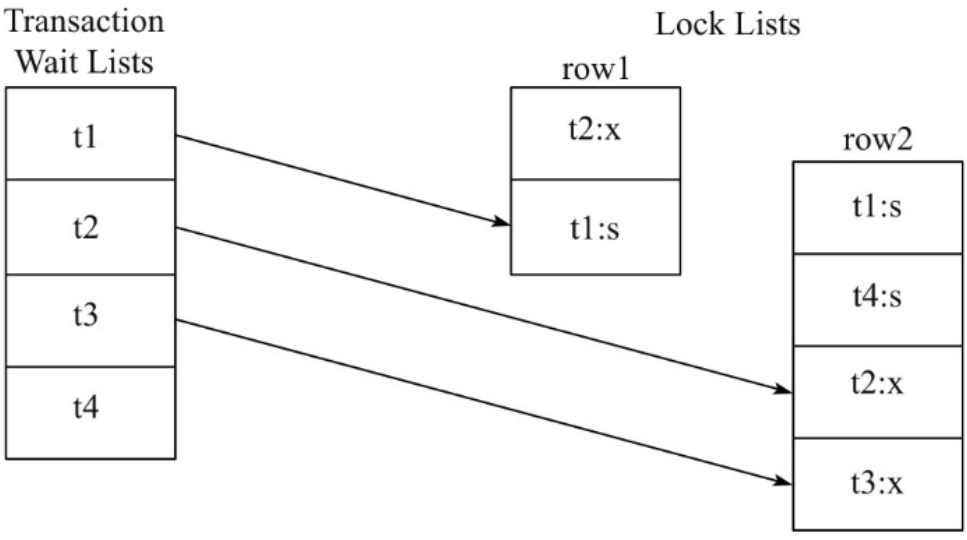



**图6-5 示例事务状态和锁的信息**

将 等待资源的事务结点 指向 已获取资源的事务结点 —— 事务等待图！

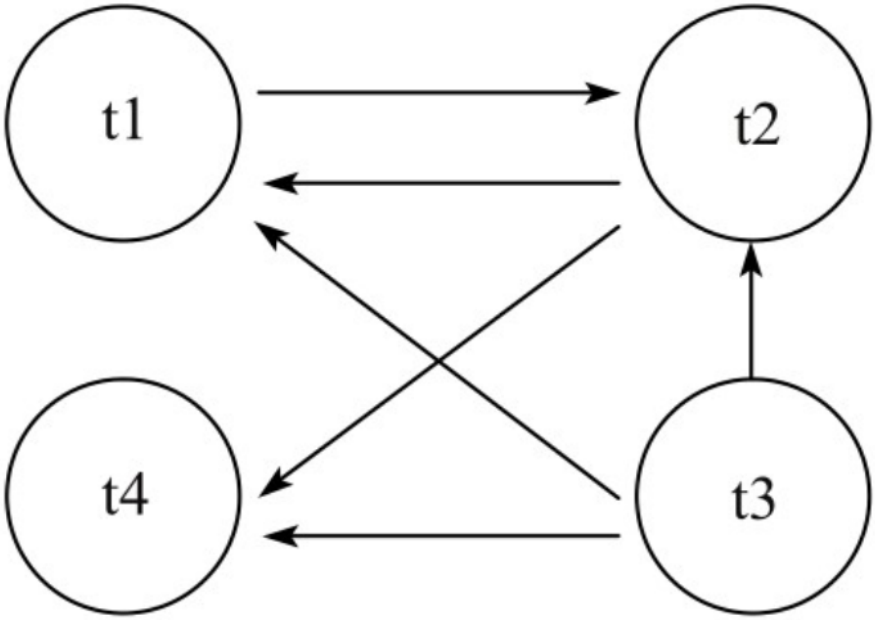



**图6-6 wait-for graph**

图中存在 t1-> t2 的环，说明存在死锁!

**死锁处理**

- Innodb 存储引擎 **检测到 死锁**的循环依赖 后，将**回滚一个事务**，并**返回一个错误**!
- InnoDB 检测到死锁之后，InnoDB 将持有最少 行级排他锁 的事务回滚！这是相对比较简单的死锁回滚算法!

**死锁的两种形式**

1. **AB - BA 形式**：*AB 相互占用对方正在等待的资源*!
2. InnoDB **主动视为死锁**的请况！此时 InnoDB 会回滚 undo log 记录多的事务!

```
# 1、事务 T1 首先获取了 某个 资源A 的 X 锁，并未提交!
# 2、事务 T2 随后，尝试获取包括 资源A 在内的资源的锁 —— 事务 T2 等待!
# 3、此时，T1 尝试在 上述 T2 申请的资源范围内 insert 一个新资源!

# 如果 执行到 步骤3 时 T1 能成功 插入资源，那么，后续当 T2 获取到 T1释放的资源的A 锁之后，还需要向后获得 T1 insert 的新资源 —— 不合理，主动认为是死锁!
```

| 时　　间 | 会话 A | 会话 B |
|---|---|---|
| 1 | BEGIN; | |
| 2 | | BEGIN; |
| 3 | SELECT * FROM t<br>WHERE a = 4 FOR UPDATE; | |
| 4 | | SELECT * FROM t<br>WHERE a <= 4 LOCK IN SHARE MODE;<br>-- 等待 |
| 5 | INSERT INTO t VALUES(3);<br>-- ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction | |
| 6 | | -- 事务获得锁，正常运行 |

上述例子：

会话A中已经对记录4持有了X锁，但是会话A中插入记录3时会导致死锁发生。

这个问题的产生是由于 会话B 中请求记录4的S锁而发生等待，但之前请求的锁对于主键值记录1、2都已经成功，若在事件点5能插入记录，那么会话B在获得记录4持有的S锁后，还需要向后获得记录3的记录，这样就显得有点不合理。

因此InnoDB存储引擎在这里主动选择了死锁，而回滚的是undo log记录大的事务，这与AB-BA死锁的处理方式又有所不同。

## 范式

### 介绍

- **范式：是具有最小冗余的表结构**！
- 不满足范式 的 关系，表中的列常常会有不同维度的信息，例如：学生信息、课程信息、班级信息的高度耦合！

  这常常会产生许多异常，一般有如下四种：

  - 数据冗余：一个同一列中，一个数据出现多次，不同维度之间可能是一对多、多对多的关系！
  - 修改异常：往往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修改！
  - 删除异常：删除一个信息，可能会丢失其他信息。（删除某个课程，可能连带会删除某个学生的所有信息）
  - 插入异常：由于列与列之间是不同维度的耦合关系，所以，插入某个维度信息，如果不存在另一个维度的信息，那么将无法插入！

  如下例：（学生-课程-学院）

| 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 |

- 范式理论是为了解决上述 4 种异常！
- 总共分为 三范式，高级别的范式依赖低级别的方式。

  1NF 是最低级别的范式！

#### 第一范式 1NF

- **每一列 字段 都是不可再分的最小数据单元：要求属性不可再分！**
- 例如：
  - Address = 中国南京

  则可再分为：
  - Country = 中国  City = 南京

#### 第二范式 2NF

- **在 1NF 的基础上**，表中的 **非主键列 不存在对 主键列的 部分依赖**：要求**每个表只描述一种关系**！
- 例如：
  - Order表　字段：订单编号、产品编号、订购日期、价格

则可再分为：Order表 和 Product表

- Order表　字段：订单编号、订购日期
- Product表　字段：产品编号、价格

**第三范式 3NF**

- **在 2NF 的基础上，表中的 列 不存在对 非主键列 的传递依赖**！
- 例如：
  - Order表　字段：订单编号、产品编号、顾客编号、顾客姓名

  Order 表中存在 `顾客姓名` 依赖于 非主键列 `顾客编号` —— 去掉 `顾客姓名` 这一列！
  - Order表　字段：订单编号、产品编号、顾客编号

## SQL vs Nosql

**SQL 优势**

- 复杂查询

  关系型数据库注重的是数据之间的逻辑关系，通过 SQL 语句能够方便的在一个表 以及 多个表之间做复杂的关系查询！
- 事务支持

  使得对于安全性能很高的数据访问要求得以实现。

**Nosql 优势**

- 性能

  NOSQL是基于键值对，可以想象成表中的主键和值的对应关系，不需要经过SQL层的解析，所以性能非常高。
- 可扩展性

  同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

**sql vs nosql**

1. 对于这两类数据库，对方的优势就是自己的弱势，反之亦然。
2. Nosql 数据库慢慢开始具备SQL数据库的一些复杂查询功能，比如MongoDB。
3. 对于事务的支持也可以用一些系统级的原子操作来实现例如乐观锁之类的方法来曲线救国，比如 Redis set nx。

## 完整性约束

```
/*
示例：
    主键约束 primary key
    非空约束 not noll
    唯一性约束 unique
    外键约束：foreign key
    检查约束：check (mysql中不支持)
*/
create table `student`{
    `stu_id` int(11) primary key comment'id',
    `stu_name` varchar(32) not null unique comment'姓名'
};
```

- 详见：5种完整性约束