# 一、What-是什么

## 1、事务简介

事务(Transaction)是访问并可能更新数据库中各种数据项的一个程序执行单元。是一个对数据库操作的序列，是一个不可分割的工作单一个事务由一组SQL语句组成。

**事务的作用：**保障数据的正确且可靠。

### 没有事务的问题?

转账例子：A账户给B账户转账一个100元。在这种交易的过程中，有哪些问题：


- 交易中如何同时保证，A账户总金额减少100元，B账户总金额增加100元？ A
- A账户如果同时在和C账户交易，如何让这两笔交易互不影响？ I
- 如果交易完成时数据库突然崩溃，如何保证交易数据不丢失，即保存在数据库中？ D
- 如何在支持大量交易的同时，保证数据的合法性(没有钱凭空产生或消失)？ C

**综上，事务需要满足：**

- 数据库事务可以包含一个或多个数据库操作,但这些操作构成一个逻辑上的整体。
- 构成逻辑整体的这些数据库操作,要么全部执行成功,要么全部不执行。
- 构成事务的所有操作,要么全都对数据库产生影响,要么全都不产生影响,即不管事务是否执行成功,数据库总能保持一致性状态。
- 以上即使在数据库出现故障以及并发事务存在的情况下依然成立。

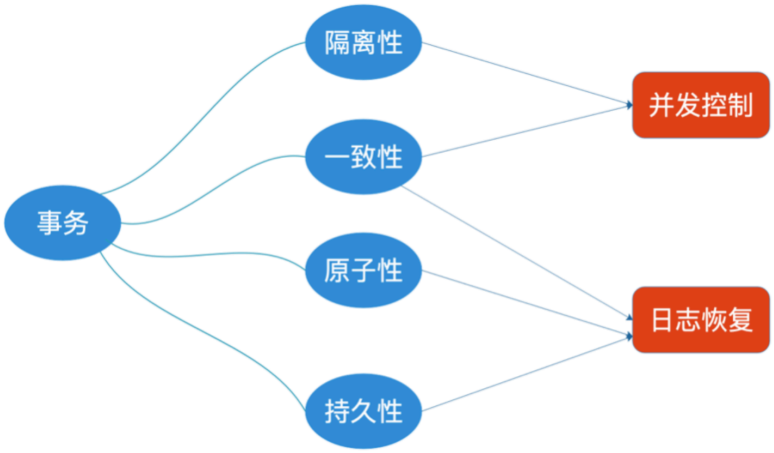
### ACID特性

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。任何事务机制在实现时，都应该考虑事务

- **原子性（atomicity）**：事务的最小工作单元，一个不可再分割的工作单位；事务中的操作要么都做，要么都不做。要么全成功，要么
- **一致性（consistency）**：事务必须是使数据库从一个一致性状态变到另一个一致性状态，事务的中间状态是不能被观察到的，保证库。 依赖原子性和隔离性。
- **隔离性（isolation）**：不同事务之间互不影响，一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的级别：读未提交(read uncommitted)、读已提交(read committed，解决脏读)、可重复读(repeatable read，解决虚读)、串行化(serializ
- **持久性（durability）**：持久性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接

**对数据一致性的破坏主要来自两个方面：**

- 事务的并发执行
- 事务故障或系统故障

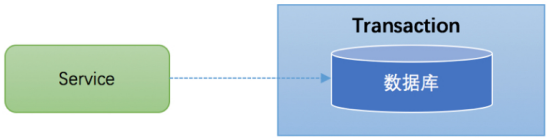



可以看出，原子性、隔离性、一致性的根本问题，是不同的事务同时对同一份数据(A账户)进行写操作(修改、删除、新增)，如果事务中

的。

# 二、How-怎么用

本地事务(Local Transaction)是比较常见的业务场景；即应用服务操作单一数据库。




## 1、sql事务操作

```sql
begin; -- 开始一个事务
-- 伪sql，仅作示意
update table set A = A - 100;
update table set B = B + 100;
-- 其他读写操作
commit; -- 提交事务
```

## 2、JDBC编程操作

java.sql.Connection对象来开启、关闭或者提交事务。

```java
Connection conn = ... //获取数据库连接
conn.setAutoCommit(false); //开启事务
try{
    //...执行增删改查sql
    conn.commit(); //提交事务
}catch (Exception e) {
   conn.rollback();//事务回滚
}finally{
    conn.close();//关闭链接
 }
```

## 3、Spring 容器做法

**2.1）配置事务管理器**

spring提供了一个PlatformTransactionManager接口，其有2个重要的实现类，DataSourceTransactionManager（本地事务，其实内部了JTA规范，使用XA协议进行两阶段提交）。

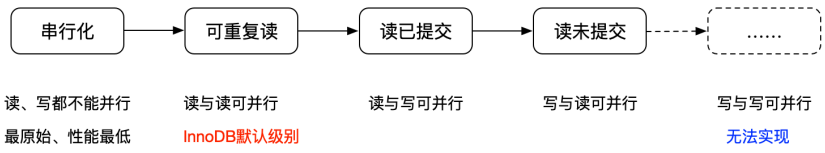
**2.2）注解方式**

在需要开启的事务的bean的方法上添加@Transitional注解。

# 三、How-深入研究

## 1、事务的隔离级别

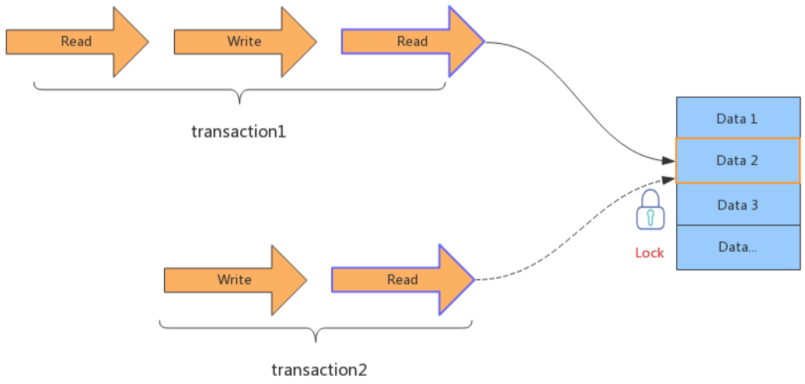
整个隔离级别的发展史，就是对mysql事务性能的优化史，研究读与写如何并行，如下：




**并行处理措施：不加锁、排它锁、读写锁、乐观锁**

**1）排它锁**

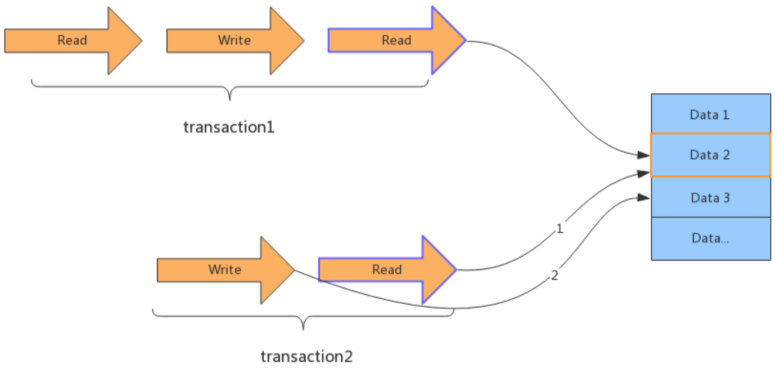
引入锁之后就可以支持并发处理事务，如果事务之间涉及到相同的数据项时，会使用排他锁，或叫互斥锁，先进入的事务独占数据项以在整个事务1结束之前，锁是不会被释放的，所以，事务2必须等到事务1结束之后开始。




2）读写锁

读和写操作:读读、写写、读写、写读。

读写锁就是进一步细化锁的颗粒度，区分读操作和写操作，让读和读之间不加锁，这样下面的两个事务就可以同时被执行了。




### 1.1 串行化（Serializable）

所有的数据库的读或者写操作都为串行执行。

当前隔离级别下只支持单个请求同时执行，所有的操作都需要队列执行。所以种隔离级别下所有的数据是最稳定的，但是性能也是最差

**各隔离级别产生的问题**

| 隔离级别 | 脏读（Dirty Read） | 不可重复读（NonRepeatable Read） | 幻读（Phantom Read） |
| --- | --- | --- | --- |
| 未提交读（Read uncommitted） | 可能 | 可能 | 可能 |
| 已提交读（Read committed） | 不可能 | 可能 | 可能 |
| 可重复读（Repeatable read） | 不可能 | 不可能 | 可能 |
| 可串行化（Serializable ） | 不可能 | 不可能 | 不可能 |

### 1.2 可重复读（Repeatable Read/RR）

一个事务读可以读取到其他事务提交的数据。

但是在RR隔离级别下，当前读取此条数据只可读取一次。

事务2执行到一半时，事务1 成功提交，会产生幻读，读出来的数据并不一定就是最新的数据，不论读取多少次，数据仍然是第一次读取

举例：幻读问题常发生在插入时。一个事务期间，第二次count的结果和第一次的不一致。

| 发生时间 | SessionA | SessionB |
| --- | --- | --- |
| 1 | begin; | |
| 2 | select count (*) from user;　(1) | |
| 3 | | insert into user (2,'李四'); |
| 4 | select count (*) from user;　(2) | |
| 事务执行过程中，数据插入了一条数据，导致第二次count的结果和第一次的不一致，影响了业务正常执行校验和后续逻辑。 | | |

### 1.3 读已提交（Read Committed/RC）

一个事务读取到另一个事务已提交的修改数据。

会出现幻读、不可重复读等问题，即导致在当前事务的不同时间读取同一条数据获取的结果不一致。

举例：不可重复读问题常发生在更新时。一个事务期间，两次查询的数据不一样。

| 发生时间 | SessionA | SessionB |
| --- | --- | --- |
| 1 | begin; | |
| 2 | select * from user where id=1;（张三） | |
| 3 | | update user set name='李四' where id=1;(默认隐式提交事务) |
| 4 | select * from user where id=1;（李四） | |
| 事务执行过程中，正在查询的数据发生了更新，导致第二次查询出来的结果和第一次的不一致，影响了业务正常执行校验和后续逻辑。 | | |

### 1.4 读未提交（Read UnCommitted/RU）

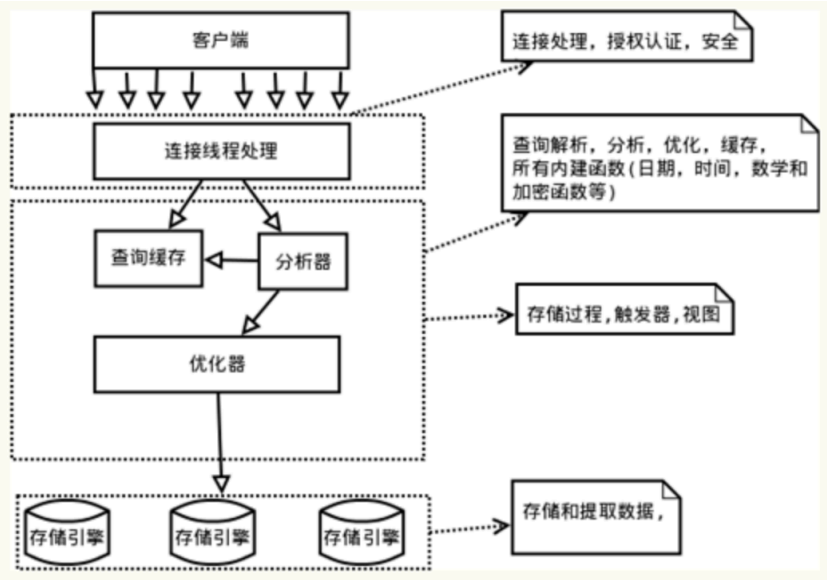
一个事务可以读取到另一个事务未提交的数据。

这种隔离级别岁最不安全的一种，会出现幻读、不可重复读、脏读等所有问题，因为未提交的事务是存在回滚的情况，一旦回滚，本次

| 发生时间 | SessionA | SessionB |
| --- | --- | --- |
| 1 | begin; | begin; |
| 2 | | update user set name='李四' where id=1;(默认隐式提交事务) |
| 3 | select * from user where id=1;（李四） | |
| 4 | commit; | |
| 5 | | rollback; |
| 事务执行过程中，正在查询的数据发生了更新，读取到最新数据，提交了事务；而更新数据的事务缺发生了回滚，此时读到了脏数据，直接影响了业务的逻辑。 | | |

## 2、Mysql架构原理

### 2.1 Mysql服务器的逻辑架构

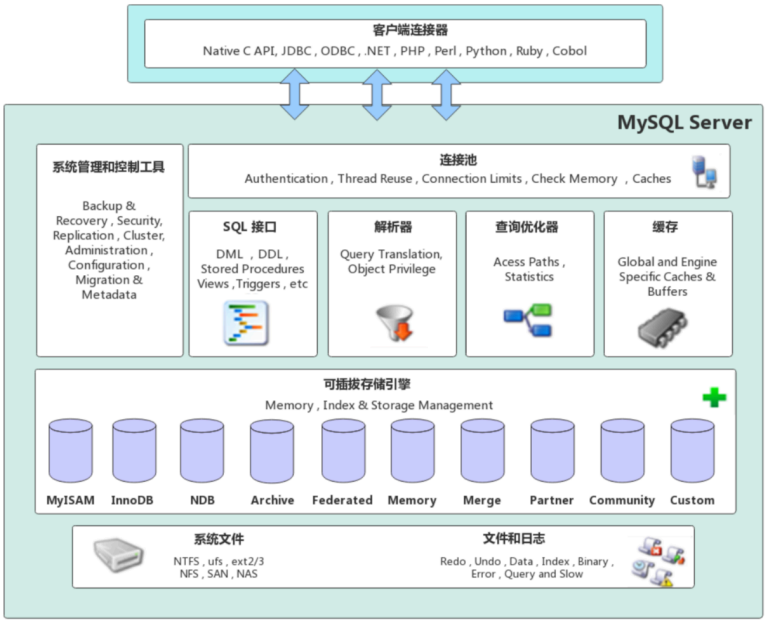



**MySQL服务器逻辑架构从上往下可以分为三层：**

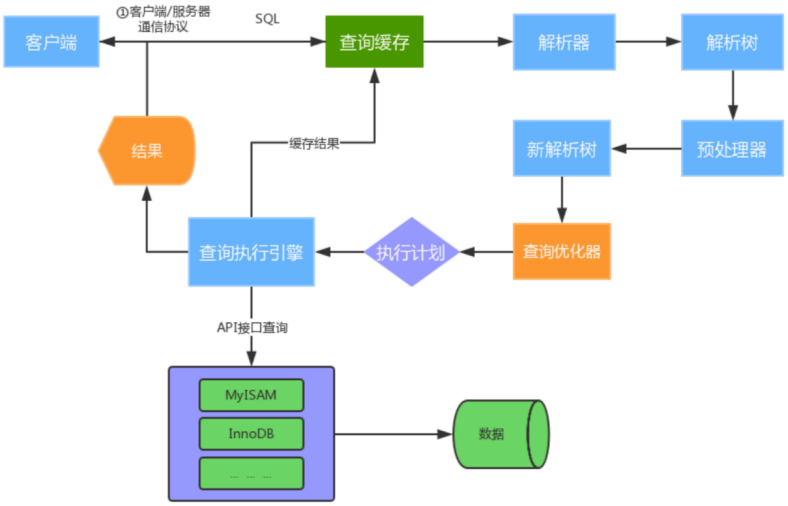
- 第一层：处理客户端连接、授权认证等。
- 第二层：服务器层，负责查询语句的解析、优化、缓存以及内置函数的实现、存储过程等。
- 第三层：存储引擎，负责MySQL中数据的存储和提取。

MySQL中服务器层不管理事务，事务是由存储引擎实现的。MySQL支持事务的存储引擎有InnoDB、NDB Cluster等，其中InnoDB的使

### 2.2 Mysql体系架构




### 2.3 Mysql运行机制




**1）首先建立连接(Connectors&Connection Pool)**

通过客户端/服务器通信协议与MySQL建立连接。MySQL 客户端与服务端的通信方式是“ 半双工 ”。对于每一个 MySQL 的连接，时刻

**通讯机制:**

全双工:能同时发送和接收数据，例如平时打电话。

半双工:指的某一时刻，要么发送数据，要么接收数据，不能同时。例如早期对讲机

单工:只能发送数据或只能接收数据。例如单行道

**2）查询缓存(Cache&Buffer)**

这是MySQL的一个可优化查询的地方，如果开启了查询缓存且在查询缓存过程中查询到完全相同的SQL语句，则将查询结果直接返回给器进行语法语义解析，并生成“解析树”。

## 3、ACID实现原理（undo和redo）

MySQL的日志有很多种，如二进制日志、错误日志、查询日志、慢查询日志等，此外InnoDB存储引擎还提供了两种事务日志：redo log

### 3.1 Undo Log

**1）介绍**

Undo意为撤销或取消，以撤销操作为目的，返回指定某个状态的操作。

**Undo Log属于逻辑日志，记录一个变化过程，记录下相反的操作**。例如执行一个delete，undo log会记录一个insert;执行一个update，

数据库事务开始之前，会将要修改的记录存放到 Undo 日志里，当事务回滚时或者数据库崩溃时，可以利用 Undo 日志，撤销未提交事

Undo Log在事务开始前产生;事务在提交时，并不会立刻删除undo log，innodb会将该事务对应的undo log放入到删除列表中，后面会

undo log采用段的方式管理和记录。在innodb数据文件中包含一种rollback segment回滚段，内部包含1024个undo log segment。可以

当事务执行update时，其生成的undo log中会包含被修改行的主键(以便知道修改了哪些行)、修改了哪些列、这些列在修改前后的值等

**2）作用**

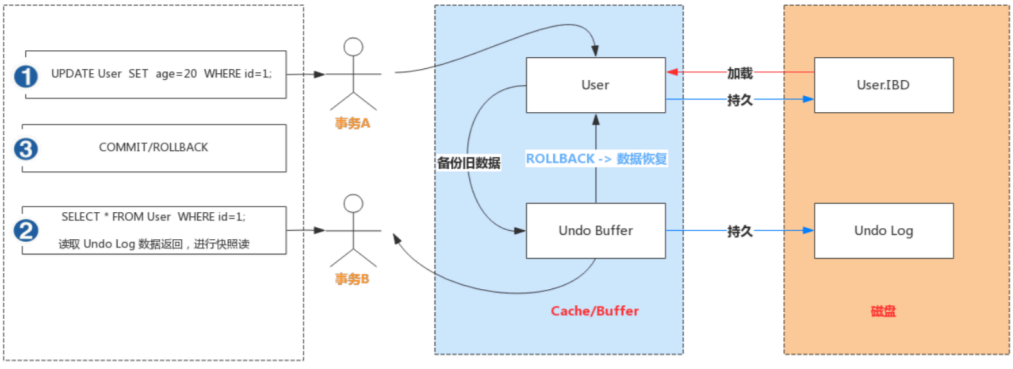
- **实现事务的原子性**

Undo Log 是为了实现事务的原子性而出现的产物。事务处理过程中，如果出现了错误或者用户执行了 ROLLBACK 语句，MySQL 可以

- **实现多版本并发控制(MVCC)**

Undo Log 在 MySQL InnoDB 存储引擎中用来实现多版本并发控制。事务未提交之前，Undo Log 保存了未提交之前的版本数据，Undo

事务A手动开启事务，执行更新操作，首先会把更新命中的数据备份到 Undo Buffer 中。 事务B手动开启事务，执行查询操作，会读取



### 3.2 Redo Log

1）介绍

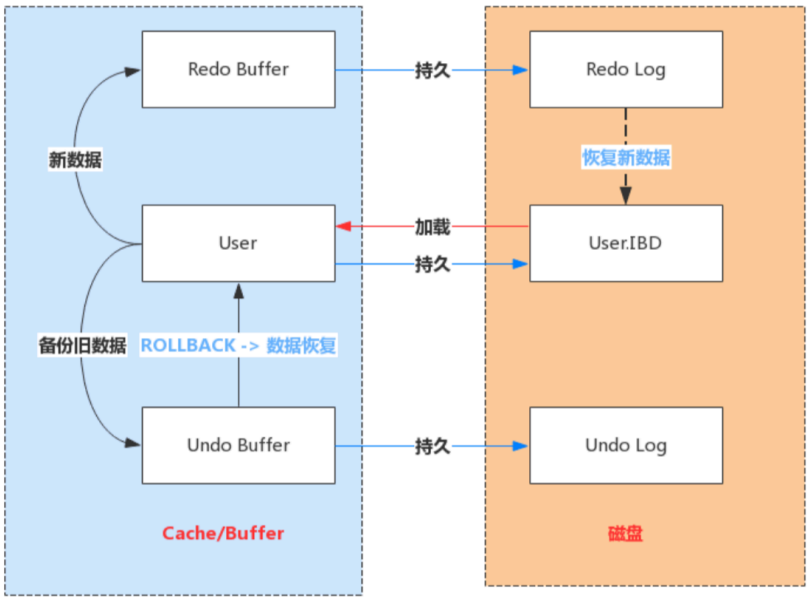
Redo顾名思义就是重做。以恢复操作为目的，在数据库发生意外时重现操作。

Redo Log指事务中修改的任何数据，将最新的数据备份存储的位置(Redo Log)，被称为重做日志。

随着事务操作的执行，就会生成Redo Log，在事务提交时会将产生 Redo Log写入Log Buffer，并不是随着事务的提交就立刻写入磁盘文件。等事
就可以重用(被覆盖写入)。

**2）Redo Log工作原理**

Redo Log 是为了实现事务的持久性而出现的产物。防止在发生故障的时间点，尚有脏页未写入表 的 IBD 文件中，在重启 MySQL 服务的时候，根




### 3.3 BinLog

1）介绍

Redo Log 是属于InnoDB引擎所特有的日志，而MySQL Server也有自己的日志，即 Binary log(二进制日志)，简称Binlog。**Binlog是记录**
**SHOW这类操作**。Binlog日志是以事件形式记录，还包含语句所执行的消耗时间。开启Binlog日志有以下两个最重要的使用场景：

- **主从复制**：在主库中开启Binlog功能，这样主库就可以把Binlog传递给从库，从库拿到 Binlog后实现数据恢复达到主从数据一致性。
- **数据恢复**：通过mysql binlog工具来恢复数据。

Binlog文件名默认为“主机名_binlog-序列号”格式，例如oak_binlog-000001，也可以在配置文件 中指定名称。文件记录模式有STATEM

**2）Binlog文件结构**

MySQL的binlog文件中记录的是对数据库的各种修改操作，用来表示修改操作的数据结构是Log event。不同的修改操作对应的不同的lo
binlog文件的内容就是各种Log event的集合。结构如下：

| | |
|---|---|
| timestamp 4字节 | 事件开始的执行时间 |
| Event Type 1字节 | 指明该事件的类型 |
| server_id 1字节 | 服务器的server ID |
| Event size 4字节 | 该事件的长度 |
| Next_log pos 4字节 | 固定4字节下一个event的开始位置 |
| Flag 2字节 | 固定2字节 event flags |
| Fixed part | 每种Event Type对应结构体固定的结构部分 |
| Variable part | 每种Event Type对应结构体可变的结构部分 |

**3）Binlog写入机制**

- 根据记录模式和操作触发event事件生成log event(事件触发执行机制)
- 将事务执行过程中产生log event写入缓冲区，每个事务线程都有一个缓冲区

Log Event保存在一个binlog_cache_mngr数据结构中，在该结构中有两个缓冲区，一个是 stmt_cache，用于存放不支持事务的信息;另

- 事务在提交阶段会将产生的log event写入到外部binlog文件中。 不同事务以串行方式将log event写入binlog文件中，所以一个事务包

binlog文件中是连续的，中间不会插入其他事务的log event。

# 4、MVCC

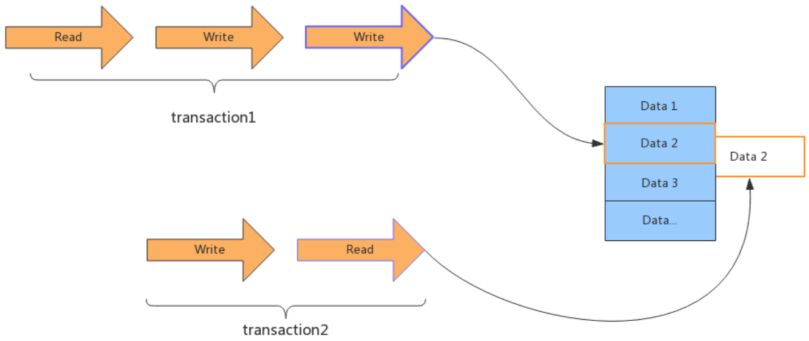
## 4.1 介绍

MVCC（Multi-Version Concurrency Control ， 多版本并发控制），也就是Copy on Write的思想。

MVCC除了支持读和读并行，还支持读和写、写和读 的并行，但为了保证一致性，写和写是无法并行的。

MVCC指的就是在使用 READ COMMITTD 、REPEATABLE READ 这两种隔离级别的事务在执行普通的 SEELCT 操作时访问记录的版本链

在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的的一个非常大的区别就是它们生成 ReadView 的时机不同。在 R
提交后的数据是处于当前的可见状态。而 REPEATABLE READ 中，在当前事务第一次查询时生成当前的 ReadView，并且当前的 ReadView
同事务内的多次 select 数据是一致的，但这个数据不一定是最新的。




## 4.2 MVCC实现原理

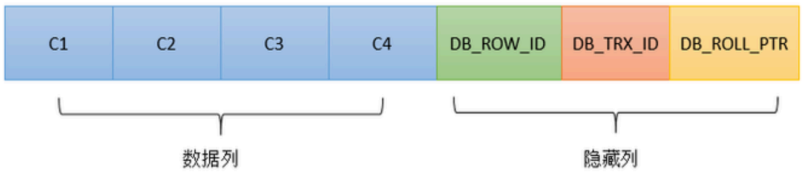
MVCC最大的好处是读不加锁，读写不冲突。在读多写少的系统应用中，读写不冲突是非常重要的，极大的提升系统的并发性能，目前

在 MVCC 并发控制中，读操作可以分为两类: 快照读(Snapshot Read)与当前读 (Current Read)。

- 快照读：读取的是记录的快照版本(有可能是历史版本)，不用加锁。(select)
- 当前读：读取的是记录的最新版本，并且当前读返回的记录，都会加锁，保证其他事务不会再并发 修改这条记录。(select... for upd

具体的实现是，在数据库的每一行中，添加额外的三个字段：

- DB_TRX_ID – 记录插入或更新该行的最后一个事务的事务 ID
- DB_ROLL_PTR – 指向改行对应的 undolog 的指针
- DB_ROW_ID – 单调递增的行 ID，他就是 AUTO_INCREMENT 的主键 ID

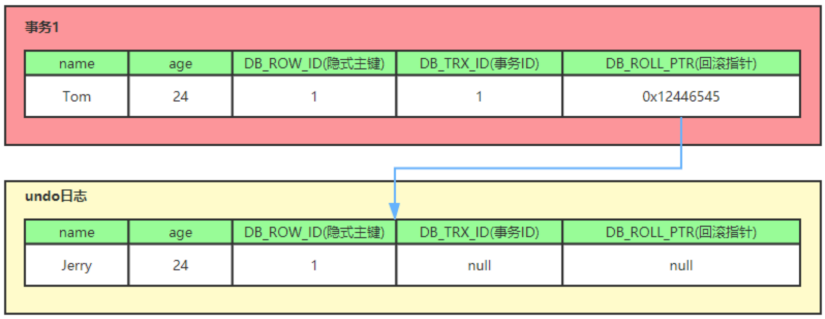



## 4.3 MVCC案例理解

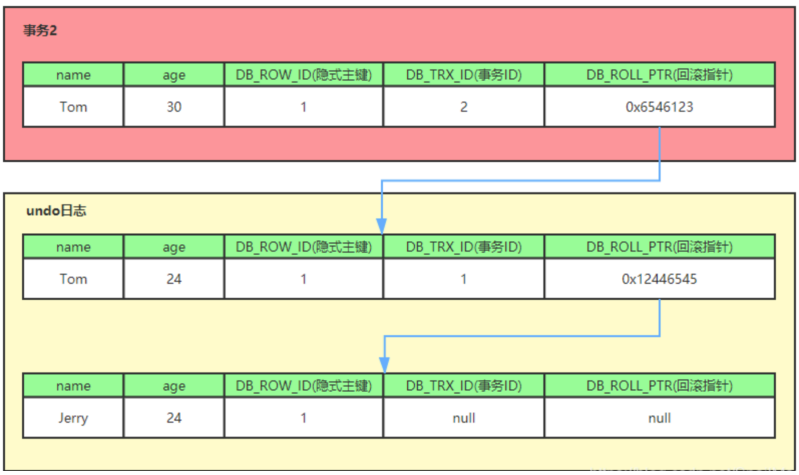
person表的某条记录

| name | age | DB_ROW_ID(隐式主键) | DB_TRX_ID(事务ID) | DB_ROLL_PTR(回滚指针) |
|---|---|---|---|---|
| Jerry | 24 | 1 | null | null |

事务1:




事务2：




Read View就是事务进行快照读操作的时候生产的读视图(Read View)，在该事务执行的快照读的那一刻，会生成数据库系统当前的一个ID, 这个ID是递增的，所以最新的事务，ID值越大)。

## 5、锁

### 5.1 锁分类

**1）从操作的粒度划分**

可分为表级锁、行级锁和页级锁。

- **表级锁**：每次操作锁住整张表。锁定粒度大，发生锁冲突的概率最高，并发度最低。应用在 MyISAM、InnoDB、BDB 等存储引擎中。
- **行级锁**：每次操作锁住一行数据。锁定粒度最小，发生锁冲突的概率最低，并发度最高。应 用在InnoDB 存储引擎中。
- **页级锁**：每次锁定相邻的一组记录，锁定粒度界于表锁和行锁之间，开销和加锁时间界于表 锁和行锁之间，并发度一般。应用在BDE

**2）从操作的类型划分**

可分为读锁和写锁。

- **读锁(S锁)**：共享锁，针对同一份数据，多个读操作可以同时进行而不会互相影响。
- **写锁(X锁)**：排他锁，当前写操作没有完成前，它会阻断其他写锁和读锁。

IS锁、IX锁:意向读锁、意向写锁，属于表级锁，S和X主要针对行级锁。在对表记录添加S或X锁之 前，会先对表添加IS或IX锁。

- S锁:事务A对记录添加了S锁，可以对记录进行读操作，不能做修改，其他事务可以对该记录追加 S锁，但是不能追加X锁，需要
- X锁:事务A对记录添加了X锁，可以对记录进行读和修改操作，其他事务不能对记录做读和修改操 作。

**3）从操作的性能划分**

可分为乐观锁和悲观锁。

- **乐观锁**：一般的实现方式是对记录数据版本进行比对，在数据更新提交的时候才会进行冲突检测，如果发现冲突了，则提示错误信息

表内设计version字段，更新时用来当条件。

- **悲观锁**：在对一条数据修改的时候，为了避免同时被其他人修改，在修改数据之前先锁定，再修改的控制方式。

共享锁和排他锁是悲观锁的不同实现，但都属于悲观锁范畴。

### 5.2 死锁

所谓死锁：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下为死锁进程。

由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就

**产生死锁的四个必要条件：**

（1） **互斥条件**：一个资源每次只能被一个进程使用。

（2） **请求与保持条件**：一个进程因请求资源而阻塞时，对已获得的资源保持不放。

（3） **不剥夺条件**：进程已获得的资源，在末使用完之前，不能强行剥夺。

（4） **循环等待条件**：若干进程之间形成一种头尾相接的循环等待资源关系。

**1）产生原因**

- 事务之间资源访问顺序不同

  用户A--》A表(表锁)--》B表(表锁)

  用户B--》B表(表锁)--》A表(表锁)

  循环等待，死循环

- 并发修改同一条记录

  共享锁转为排它锁

  事务A 查询一条纪录，然后更新该条纪录;此时事务B 也更新该条纪录，这时事务B 的排他锁由于 事务A 有共享锁，必须等A 释放共锁，因为事务A 需要排他锁来做更新操作。但是，无法授予该锁请求，因为事务B 已经 有一个排他锁请求，并且正在等待事务A 释放其共享

- 索引不当，导致全表扫描

  如果在事务中执行了一条没有索引条件的查询，引发全表扫描，把行级锁上升为全表记录锁定(等价于表级锁)，多个这样的事务执行

- 锁等待时间过长，超时

**2）解决方法**