事务保障原理

相关面试题

1. 什么WAL？有什么优点?
2. 什么是ChangeBuffer?
3. change buffer为什么仅针对非唯一普通索引页?
4. binlog的主要作用是什么?
5. binlog有哪些记录格式?
6. 为什么 redo log 具有 crash-safe 的能力，而binlog 却没有?
7. 什么是redolog的二阶段提交？为什么要使用二阶段提交?
8. mysql是如何实现崩溃恢复的?
9. undolog的作用有哪些?
10. 唯一索引值不存在时，会锁住哪些数据?
11. 什么是当前读？什么是快照读?
12. Read View的作用是什么?
13. Mysql在RR下有几种解决幻读问题的方式?
14. redolog可以独完成崩溃恢复吗?

什么是事务

数据库的**事务（Transaction）**是一种机制、一个操作序列，包含了一组数据库操作命令。事务把所有的命令作为一个整体一起向系统提交或撤销操作请求，即这一组数据库命令要么都执行，要么都不执行，因此事务是一个不可分割的工作逻辑单元。

通常情况下，事务可以看成是一组DML语句，要么同时成功，要么同时失败。

事务具有 4 个特性，即原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），这 4 个特性通常简称为 ACID。

1. 原子性

> 当前事务的操作要么全部成功要么全部失败。

原子性是由`undo log`来保证的，undolog记录着数据修改之前的值。比如我们insert一条语句，undolog就会记录一条delete语句，我们update一条语句，undolog就会记录一条对应的原来数据的update语句，如果回滚就会利用到undolog日志的内容。

2. 一致性

> 我们使用事务的目的就是为例保证一致性，而原子性、隔离性、持久性都是为了保证一致性的。

3. 隔离性

> 对数据进行修改的所有并发事务是彼此隔离的，这表明事务必须是独立的，它不应以任何方式依赖于或影响其他事务。

事务的隔离性是通过锁机制或MVCC实现的。mysql支持四种隔离级别。

> **Read uncommitted(读未提交)**
>
> **Read committed(读提交)**
>
> **Repeatable read(可重复读取)**
>
> **Serializable(串行化)**

隔离级别	脏读	不可重复读	幻读
RU	有	有	有
RC	无	有	有
RR	无	无	**可能有**
S	无	无	无

事务的隔离级别越高，安全性越好，但是并发性能就会越低。隔离性底层是由锁或mvcc来实现的，所谓隔离性只是屏蔽了加锁的细节。

4. 持久性

持久性是由redolog来保证的，如果我们需要修改数据，MySQL会先把这个数据所在的页找到，加载到内存当中，将对应的数据修改了，为了防止我们内存刚修改完MySQL服务器就挂了，就无法记录到硬盘当中了，所以MySQL会把操作记录到redolog日志当中，redolog是顺序写入的，写入的速度很快，即使MySQL数据库挂了，也可以通过redolog进行恢复。

MySQL的WAL机制

WAL全称为Write-Ahead Logging，预写日志系统。其主要是指MySQL在执行写操作的时候并不是立刻更新到磁盘上，而是先记录在日志中，之后在合适的时间更新到磁盘中。日志主要分为undo log、redo log、binlog。
当内存数据页跟磁盘数据页内容不一致的时候，我们成这个内存页为“脏页”。内存数据写入磁盘后，内存和磁盘上的数据页内容就一致了，称为“干净页”。
MySQL真正使用WAL的原因是：磁盘的写操作是随机IO，比较耗性能，所以如果把每一次的更新操作都先写入log中，那么就成了顺序写操作，实际更新操作由后台线程再根据log异步写入。这样对于client端，延迟就降低了。并且，由于顺序写入大概率是在一个磁盘块内，这样产生的IO次数也大大降低。所以WAL的核心在于将随机写转变为了顺序写，降低了客户端的延迟，提升了吞吐量。
WAL 其实也是这两种思路的一种实现，一方面 WAL 中记录事务的更新内容，通过 WAL 将随机的脏页写入变成顺序的日志刷盘，另一方面，WAL 通过 buffer 的方式改单条磁盘刷入为缓冲批量刷盘，再者从 WAL 数据到最终数据的同步过程中可以采用并发同步的方式。这样极大提升数据库写入性能，因此，WAL 的写入能力决定了数据库整体性能的上限，尤其是在高并发时。

change buffer

在MySQL中数据分为内存和磁盘两个部分；在buffer pool中缓存热的数据页和索引页，减少磁盘读；通过change buffer就是为了缓解磁盘写的一种手段。

change buffer就是在非唯一普通索引页不在buffer pool中时，对页进行了写操作的情况下，先将记录变更缓冲，等未来数据被读取时，再将 change buffer 中的操作merge到原数据页的技术。在MySQL5.5之前，叫插入缓冲(insert buffer)，只针对insert做了优化；现在对delete和update也有效，叫做写缓冲(change buffer)。

当需要更新一个数据页时，如果数据页在内存中就直接更新。如果数据页不在内存中。在不影响数据一致性的前提下，InooDB 会将这些更新操作缓存在 change buffer 中，这样就不需要从磁盘中读入这个数据页了。在下次查询需要访问这个数据页的时候，将数据页读入内存，然后执行 change buffer 中与这个页有关的操作。通过这种方式就能保证这个数据逻辑的正确性。

> change buffer为什么仅针对非唯一普通索引页?
>
> 对于唯一索引，所有的更新操作都要先判断这个操作是否违反唯一性约束。而这必须要将数据页读入内存才能判断。如果都已经读入到内存了，那直接更新内存会更快，就没必要使用 change buffer 了。因此，唯一索引的更新就不能使用 change buffer，实际上也只有普通索引可以使用。

相关变量

```
show variables like '%change_buffer%';
```

二进制日志(bin log)

记录所有更改数据的语句，可用于数据复制。属于物理日志，基于MySQL服务层的，所有的存储引擎都有binglog，通过追加的方式记录文件，可以通过max_binlog_size来设置binlog文件的大小，满了会创建新的文件。

虽然binlog记录了所有的操作，但是由于binlog无法判断哪些数据已经刷盘，索引MySQL服务宕机了之后无法通过binlog恢复，

binlog日志有三种形式：

Statement：基于sql语句的复制，不需要记录每一行的变化，减少binlog的日志量，节省IO，提高性能。但是需要记录sql上下文相关的信息。

Row：基于行的复制，不需要记录sql语句上下文的信息，仅需要记录每一行数据被修改成了什么，但是可能会导致大量的日志。

Mixed：混合模式复制，前两种的结合。涉及到一些函数的sql，binlog无法记录sql，就需要Row的格式，MySQL根据执行的具体的sql自动选择记录的格式。

binlog有两个常用的使用场景：

- 主从复制：mysql replication在master端开启binlog,master把它的二进制日志传递给slaves来达到master-slave数据一致的目的。
- 数据恢复：通过mysqlbinlog工具来恢复数据。**注意区别宕机和崩溃恢复**

相关变量

```
# 是否开启
show variables like 'log_bin'
# 单个二进制日志文件的最大值
show variables like 'max_binlog_size'
# InnoDB会将所有未提交的binLog写到一个缓存中，等事务提交后再将缓存刷新到文件。缓存大小有参数
binlog_cache_size控制。**会话级别**
show variables like 'binlog_cache_size'
# 二进制日志的格式：Statement、Row、Mixed
show variables like 'binlog_format'
```

重做日志（redo log）

> redo是InnoDB引擎特有的。**记录着事务里对数据的修改**。

redo log主要用于MySQL服务器异常重启的情况，可以用来恢复尚未写入磁盘的数据，因为MySQL进行更新操作时候采用了异步写入磁盘的技术，写入内存就返回可能导致数据丢失，这时候redo log就可以起作用了。

redolog可以判断哪些数据已经写入了磁盘，写入磁盘之后就会删除redolog里面的日志，所以是可以一直写的。如果redolog写入失败，说明这次的操作失败，事务也就不会提交。redolog内部结构是基于页的，记录了这个页字段的变化。

对比binlog和redolog

	redo log	**bin log**
作用	用于崩溃恢复	主从复制和数据恢复
实现方式	Innodb存储引擎实现	Server层实现，所有的存储引擎都可以使用
记录方式	循环写的方式记录，写到结尾时，会回到开头循环写日志	通过追回的方式记录，当文件尺寸大于配置值时，后续的日志记录到新的文件中
文件大小	redolog的大小是固定的	通过配置参数max_binlog_size设置每个binlog文件大小
crash-safe能力	有	没有
日志类型	逻辑日志	物理日志

为什么 redo log 具有 crash-safe 的能力，是 binlog 无法替代的?

第一点：redo log 可确保 innoDB 判断哪些数据已经刷盘，哪些数据还没有刷盘。

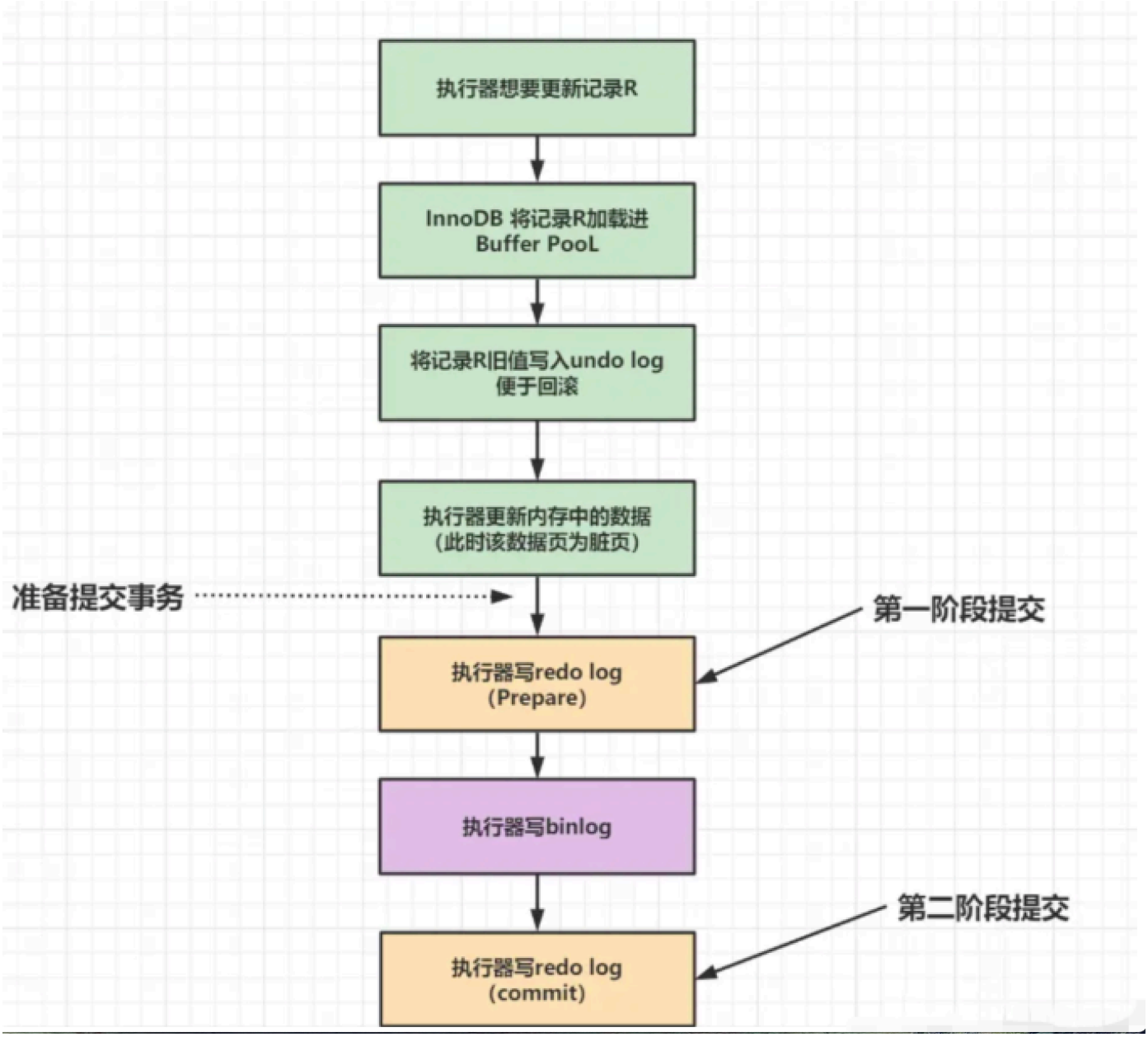
redo log 和 binlog 有一个很大的区别就是，一个是循环写，一个是追加写。也就是说 redo log 只会记录未刷盘的日志，已经刷入磁盘的数据都会从 redo log 这个有限大小的日志文件里删除。binlog 是追加日志，保存的是全量的日志。
当数据库 crash 后，想要恢复未刷盘但已经写入 redo log 和 binlog 的数据到内存时，binlog 是无法恢复的。虽然 binlog 拥有全量的日志，但没有一个标志让 innoDB 判断哪些数据已经刷盘，哪些数据还没有。
但 redo log 不一样，只要刷入磁盘的数据，都会从 redo log 中抹掉，因为是循环写！数据库重启后，直接把 redo log 中的数据都恢复至内存就可以了。

第二点：如果 redo log 写入失败，说明此次操作失败，事务也不可能提交

redo log 每次更新操作完成后，就一定会写入日志，如果写入失败，说明此次操作失败，事务也不可能提交。
redo log 内部结构是基于页的，记录了这个页的字段值变化，只要crash后读取redo log进行重放，就可以恢复数据。
这就是为什么 redo log 具有 crash-safe 的能力，而 binlog 不具备。

什么是两阶段提交?

> MySQL 将 redo log 的写入拆成了两个步骤：prepare 和 commit，中间再穿插写入binlog，这就是"两阶段提交"。



而两阶段提交就是让这两个状态保持逻辑上的一致。redolog 用于恢复主机故障时的未更新的物理数据，binlog 用于备份操作。两者本身就是两个独立的个体，要想保持一致，就必须使用分布式事务的解决方案来处理。

为什么需要两阶段提交呢?

> 如果不用两阶段提交的话，可能会出现这样情况
> 先写 `redo log`，`crash` 后 `bin log` 备份恢复时少了一次更新，与当前数据不一致。
> 先写 `bin log`，`crash` 后，由于 `redo log` 没写入，事务无效，所以后续 `bin log` 备份恢复时，数据不一致。
> 两阶段提交就是为了保证 `redo log` 和 `binlog` 数据的安全一致性。只有在这两个日志文件逻辑上高度一致了才能放心的使用。

在恢复数据时，redolog 状态为 commit 则说明 binlog 也成功，直接恢复数据；如果 redolog 是 prepare，则需要查询对应的 binlog事务是否成功，决定是回滚还是执行。

mysql是如何实现崩溃恢复的?

首先比较重要的一点是，在写入redo log时，会顺便记录XID，即当前事务id。在写入binlog时，也会写入XID。因此存在以下四种情况：

```
如果在写入redo log之前崩溃，那么此时redo log与binlog中都没有，是一致的情况，崩溃也无所谓。
如果在写入redo log prepare阶段后立马崩溃，之后会在崩恢复时，由于redo log没有被标记为commit，由于此时是在prepare阶段crash，所以redolog中还没有commit。于是拿着redo log中的XID去bin log中查找，此时肯定是找不到的，那么执行回滚操作。
如果在写入bin log后立马崩溃，在恢复时，由redo log中的XID可以找到对应的bin log，这个时候直接提交即可。
如果在写入redo log的commit后crash，说明提交成，直接恢复。
```

总的来说，在崩溃恢复后，只要redo log不是处于commit阶段，那么就拿着redo log中的XID去binlog中寻找，找得到就提交，否则就回滚。在这样的机制下，两阶段提交能在崩溃恢复时，能够对提交中断的事务进行补偿，来确保redo log与binlog的数据一致性。

redolog可以独完成崩溃恢复吗?

回滚日志（undo log）

在数据库事务开始之前，MYSQL会去记录更新前的数据到undo log文件中。如果事务回滚或者数据库崩溃时，可以利用undo log日志中记录的日志信息进行回退。同时也可以提供多版本并发控制下的读(MVCC)。

如我们执行下面一条删除语句：

```
delete from book where id = 1;
```

那么此时undo log会生成一条与之相反的insert 语句【反向操作的语句】，在需要进行事务回滚的时候，直接执行该条sql，可以将数据完整还原到修改前的数据，从而达到事务回滚的目的。

> delete -> insert
>
> insert -> delete
>
> update -> update
>
> update book set name = "三国" where id = 1; ---修改之前name=西游记
>
> update book set name = "西游记" where id = 1;

对比redolog和undolog

> undo log实现事务的原子性
> undo log记录的是事务[开始前]的数据状态，记录的是更新之前的值
> undo log实现事务的原子性(提供回滚)
> redo log实现事务的持久性
> redo log记录的是事务[完成后]的数据状态，记录的是更新之后的值
> redo log实现事务的持久性(保证数据的完整性)

锁机制

> 共享锁：共享锁(shared lock)也称为读锁(read lock)。共享锁是共享的，或者说是相互不阻塞的。多个连接在同一时刻可以同时读取同一个资源，而不相互干扰。
>
> 排他锁：排他锁(exclusive lock)也称为写锁(write lock)。写锁是排他的，也就是一个写锁会阻塞其他的写锁和读锁。

表级锁

表锁是MySQL中最基本的锁策略，并且也是开销最小的策略。表锁会锁住整张表，在对表进行写操作(插入、删除、更新等)前，需要先获取写锁，它会阻塞其他用户对该表的所有读写操作。只有没有写锁时，其他读取的用户才能获取读锁。读锁之前互相不会造成阻塞。

写锁的优先级高于读锁，因此一个写锁的请求可能会被插入到读锁队列前面，但是读锁是不能插入到写锁的前面的。

```
-- 对表加读锁
lock tables ...... read;
-- 对表加写锁
lock tables ...... write;
-- 释放锁
unlock tables;
```

读锁测试：读锁不会阻塞读请求，写请求会正常阻塞。

> - 窗口1：对user表加读锁。
> - 窗口2：对user表全表读取，正常返回全表数据。
> - 窗口2：修改user表中id=6的数据，阻塞。读写冲突。
> - 窗口1：修改user表中id=6的数据，报错。
> - 窗口1：释放读锁，窗口2的更新数据执行成功。

写锁测试：会阻塞读请求和写请求

> - 窗口1：对user表添加写锁。
> - 窗口2：全表查询user表，阻塞。
> - 窗口1：更新user表中的id=6的数据，更新成功。
> - 窗口1：释放锁，窗口2的全表查询返回最新数据。

注意

> MyISAM默认是表级锁不支持行级锁。
>
> InnoDB默认用的是行级锁也支持表级锁。

记录锁record lock

对于InnoDB 在RR(MySQL默认隔离级别) 而言，对于 update、delete 和 insert 语句， 会自动给涉及数据集加排它锁（X）；

对于普通 select 语句，innodb 不会加任何锁。如果想在select操作的时候加上 S锁 或者 X锁，需要我们手动加锁。

共享锁：共享锁(shared lock)也称为读锁(read lock)。共享锁是共享的，或者说是相互不阻塞的。多个连接在同一时刻可以同时读取同一个资源，而不相互干扰。

对于共享锁而言，对当前行加**共享锁**，不会阻塞其他事务对同一行的读请求，但会阻塞对同一行的写请求。只有当读锁释放后，才会执行其它事物的写操作。

```
select ..... lock in share mode;
```

排他锁(exclusive lock)也称为写锁(write lock)。写锁是排他的，也就是一个写锁会阻塞其他的写锁和读锁(也要获取锁才会阻塞)。

对于排它锁而言，会阻塞其他事务对同一行的读和写操作，只有当写锁释放后，才会执行其它事务的读写操作。

```
select ..... for update;
```

间隙锁gap lock

间隙锁 是 **Innodb 在 RR(可重复读) 隔离级别** 下为了解决`幻读问题`时引入的锁机制。**间隙锁是innodb中行锁的一种**。

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**。

举例来说，假如emp表中只有101条记录，其empid的值分别是1,2,...,100,101，下面的SQL：

```
SELECT * FROM emp WHERE empid > 100 FOR UPDATE
```

当我们用条件检索数据，并请求共享或排他锁时，InnoDB不仅会对符合条件的empid值为101的记录加锁，也会对empid大于101（这些记录并不存在）的“间隙”加锁。

这个时候如果你插入empid等于102的数据的，如果那边事物还没有提交，那你就会处于等待状态，无法插入数据。

> **注意**间隙锁只在可重复读隔离级别中存在。

临键锁（Next-Key Locks）

Next-key锁是记录锁和间隙锁的组合，它指的是加在某条记录以及这条记录前面间隙上的锁。

也可以理解为一种特殊的**间隙锁**。通过**临建锁**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。

> 唯一索引等值查询：
>
> - 当查询的记录是存在的，next-key lock 会退化成「记录锁」。
> - 当查询的记录是不存在的，next-key lock 会退化成「间隙锁」。
>
> 非唯一索引等值查询：
>
> - 当查询的记录存在时，除了会加 next-key lock 外，还额外加间隙锁，也就是会加两把锁。
> - 当查询的记录不存在时，只会加 next-key lock，然后会退化为间隙锁，也就是只会加一把锁。

> 非唯一索引和主键索引的范围查询的加锁规则不同之处在于：
>
> - 唯一索引在满足一些条件的时候，next-key lock 退化为间隙锁和记录锁。
> - 非唯一索引范围查询，next-key lock 不会退化为间隙锁和记录锁。

假设有如下表：

id主键, age 普通索引

id	age	name
1	10	aa
5	25	bb
10	35	cc
15	45	dd

该表中 id列潜在的临键锁有：
(-∞, 1],
(1, 5],
(5, 10],
(10, 15],
(15, +∞],

该表中 age 列潜在的临键锁有：
(-∞, 10],
(10, 25],
(25, 35],
(35, 45],
(45, +∞],
在事务 A 中执行如下命令：

```
-- 根据非唯一索引列 UPDATE 某条记录
UPDATE table SET name = Vladimir WHERE age = 24;
-- 或根据非唯一索引列 锁住某条记录
SELECT * FROM table WHERE age = 24 FOR UPDATE;
```

不管执行了上述 SQL 中的哪一句，之后如果在事务 B 中执行以下命令，则该命令会被阻塞：

```
INSERT INTO table VALUES(100, 26, 'ee');
```

事务 A 在对 age 为 24 的列进行 UPDATE 操作的同时，也获取了 (24, 32] 这个区间内的临键锁。

意向锁 intention lock

意向锁（Intention Lock），又称I锁。针对表锁。
意向锁表示某个事务正在锁定一行或者将要锁定一行，表明一个意图。它分为意向共享锁（IS）和意向排他锁（IX）：

一个事务对一张表的某行添加共享锁前，必须获得对该表一个IS锁或者优先级更高的锁。
一个事务对一张表的某行添加排他锁之前，它必须对该表获取一个IX锁。
意向锁属于表锁，它不与innodb中的行锁冲突，任意两个意向锁之间也不会产生冲突，但是会与表锁（S锁和X锁）产生冲突，如下表：

兼容性　S锁　X锁
IS锁　兼容　冲突
IX锁　冲突　冲突
表锁和行锁已经保证了事务的隔离性，确保数据一致，那么为什么还要使用意向锁呢?
意向锁是在当事务加表锁时发挥作用。比如一个事务想要对表加排他锁，如果没有意向锁的话，那么该事务在加锁前需要判断当前表的每一行是否已经加了锁，如果表很大，遍历每行进行判断需要耗费大量的时间。如果使用意向锁的话，那么加表锁前，只需要判断当前表是否有意向锁即可，这样加快了对表锁的处理速度。
意向锁是有存储引擎自己维护的，是内部机制，用户无法操作意向锁。

MVCC

> `MVCC`，全称`Multi-Version Concurrency Control`，即多版本并发控制。MVCC是一种并发控制的方法，一般在数据库管理系统中，实现对数据库的并发访问，在编程语言中实现事务内存。
>
> 多版本控制: 指的是一种提高并发的技术。最早的数据库系统，只有读读之间可以并发，读写，写读，写写都要阻塞。引入多版本之后，**只有写写之间相互阻塞，其他三种操作都可以并行**，这样大幅度提高了InnoDB的并发度。在内部实现中，与Postgres在数据行上实现多版本不同，InnoDB是在undolog中实现的，通过undolog可以找回数据的历史版本。找回的数据历史版本可以提供给用户读(按照隔离级别的定义，有些读请求只能看到比较老的数据版本)，也可以在回滚的时候覆盖数据页上的数据。在InnoDB内部中，会记录一个全局的活跃读写事务数组，其主要用来判断事务的可见性。
> MVCC是一种多版本并发控制机制。

MVCC在MySQL InnoDB中的实现主要是为了提高数据库并发性能，用更好的方式去处理读-写冲突，做到即使有读写冲突时，也能做到不加锁，非阻塞并发读

什么是当前读和快照读?

- 当前读
  像select lock in share mode(`共享锁`), select for update ; update, insert ,delete(`排他锁`)这些操作都是一种当前读，为什么叫当前读？就是它读取的是记录的最新版本，读取时还要保证其他并发事务不能修改当前记录，会对读取的记录进行加锁
- 快照读
  像`不加锁`的select操作就是快照读，即不加锁的非阻塞读；快照读的前提是隔离级别不是串行级别，串行级别下的快照读会退化成当前读；之所以出现快照读的情况，是基于提高并发性能的考虑，快照读的实现是基于多版本并发控制，即MVCC,可以认为MVCC是行锁的一个变种，但它在很多情况下，避免了加锁操作，降低了开销；既然是基于多版本，即快照读可能读到的并不一定是数据的最新版本，而有可能是之前的历史版本

说白了MVCC就是为了实现读-写冲突不加锁，而这个读指的就是`快照读`, 而非当前读，当前读实际上是一种加锁的操作，是悲观锁的实现

当前读，快照读和MVCC的关系

- 准确的说，MVCC多版本并发控制指的是 “维持一个数据的多个版本，使得读写操作没有冲突” 这么一个概念。仅仅是一个理想概念

- 而在MySQL中，实现这么一个MVCC理想概念，我们就需要MySQL提供具体的功能去实现它，而快照读就是MySQL为我们实现MVCC理想模型的其中一个具体非阻塞读功能。而相对而言，当前读就是悲观锁的具体功能实现
- 要说的再细致一些，快照读本身也是一个抽象概念，再深入研究。MVCC模型在MySQL中的具体实现则是由`3个隐式字段`，`undo日志`，`Read View`等去完成的。

MVCC带来的好处

多版本并发控制（MVCC）是一种用来解决`读-写冲突`的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 所以MVCC可以为数据库解决以下问题

- 在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能
- 同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题

总之，MVCC就是因为大牛们，不满意只让数据库采用悲观锁这样性能不佳的形式去解决读-写冲突问题，而提出的解决方案，所以在数据库中，因为有了MVCC，所以我们可以形成两个组合：

- `MVCC + 悲观锁`
  MVCC解决读写冲突，悲观锁解决写写冲突
- `MVCC + 乐观锁`
  MVCC解决读写冲突，乐观锁解决写写冲突

这种组合的方式就可以最大程度的提高数据库并发性能，并解决读写冲突，和写写冲突导致的问题

MVCC的实现原理

MVCC的目的就是多版本并发控制，在数据库中的实现，就是为了解决`读写冲突`，它的实现原理主要是依赖记录中的`3个隐式字段`，`undo日志`，`Read View`来实现的。

隐式字段

每行记录除了我们自定义的字段外，还有数据库隐式定义的`DB_TRX_ID`,`DB_ROLL_PTR`,`DB_ROW_ID`等字段

`DB_TRX_ID`
6byte，最近修改(`修改/插入`)事务ID：记录创建这条记录/最后一次修改该记录的事务ID

`DB_ROLL_PTR`
7byte，回滚指针，指向这条记录的上一个版本（存储于rollback segment里）

`DB_ROW_ID`
6byte，隐含的自增ID（隐藏主键），如果数据表没有主键，InnoDB会自动以`DB_ROW_ID`产生一个聚簇索引

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	1	0x12446545

如图，`DB_ROW_ID`是数据库默认为该行记录生成的唯一隐式主键，`DB_TRX_ID`是当前操作该记录的事务ID，而`DB_ROLL_PTR`是一个回滚指针，用于配合undo日志，指向上一个旧版本

undo日志

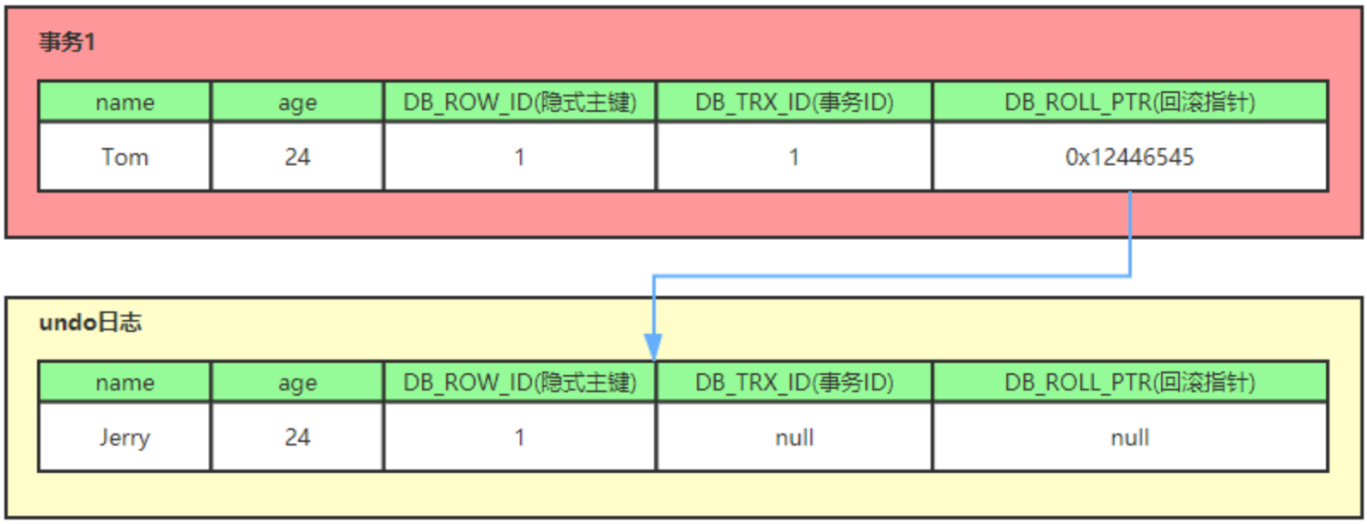
比如一个有个事务插入person表插入了一条新记录，记录如下，`name`为Jerry，`age`为24岁，`隐式主键`是1，`事务ID`和`回滚指针`，我们假设为NULL

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

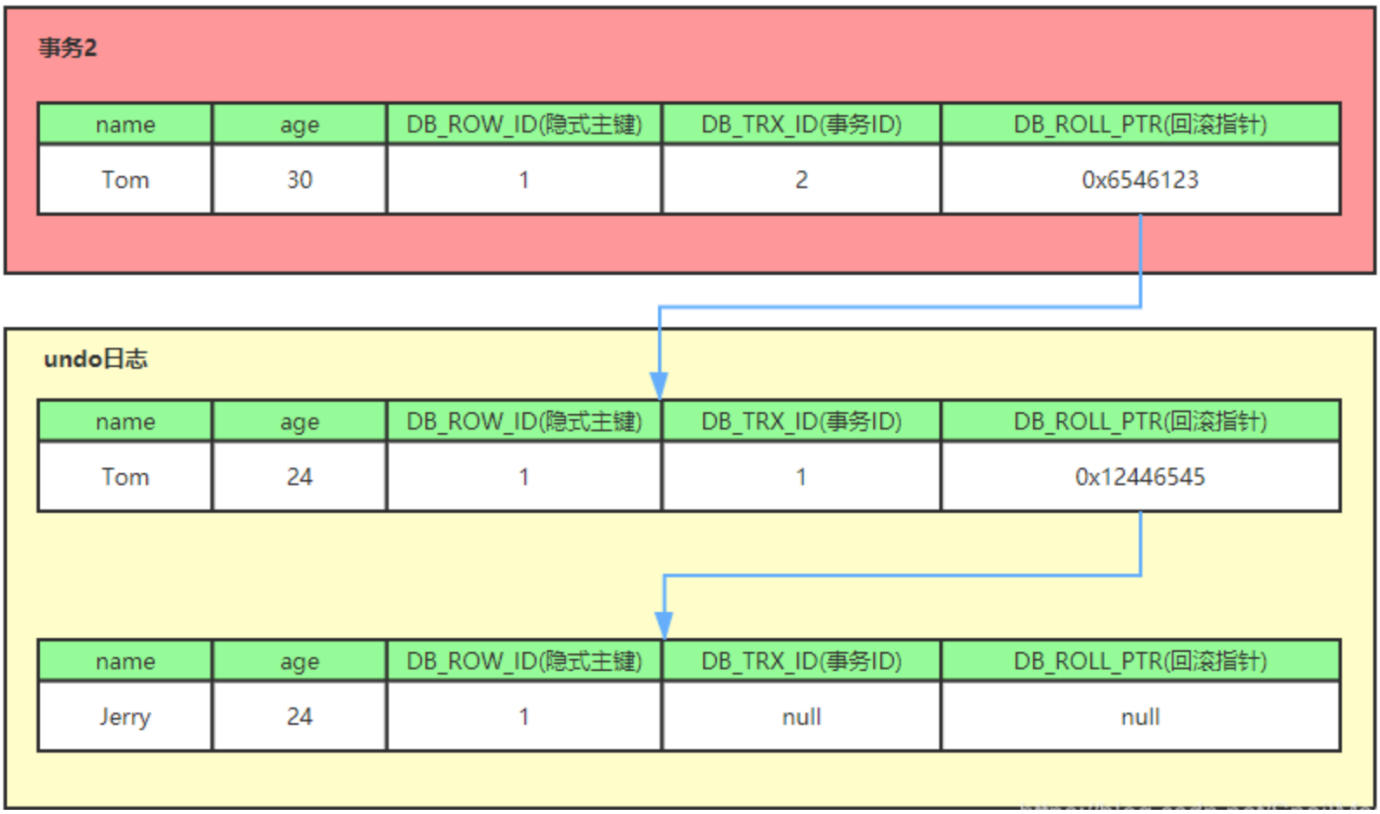
现在来了一个`事务1`对该记录的`name`做出了修改，改为Tom

- 在`事务1`修改该行(记录)数据时，数据库会先对该行加`排他锁`
- 然后把该行数据拷贝到`undo log`中，作为旧记录，既在`undo log`中有当前行的拷贝副本
- 拷贝完毕后，修改该行`name`为Tom，并且修改隐藏字段的事务ID为当前`事务1`的ID, 我们默认从`1`开始，之后递增，回滚指针指向拷贝到`undo log`的副本记录，既表示我的上一个版本就是它
- 事务提交后，释放锁



又来了个`事务2`修改`person表`的同一个记录，将`age`修改为30岁

- 在`事务2`修改该行数据时，数据库也先为该行加锁
- 然后把该行数据拷贝到`undo log`中，作为旧记录，发现该行记录已经有`undo log`了，那么最新的旧数据作为链表的表头，插在该行记录的`undo log`最前面
- 修改该行`age`为30岁，并且修改隐藏字段的事务ID为当前`事务2`的ID, 那就是`2`，回滚指针指向刚刚拷贝到`undo log`的副本记录
- 事务提交，释放锁



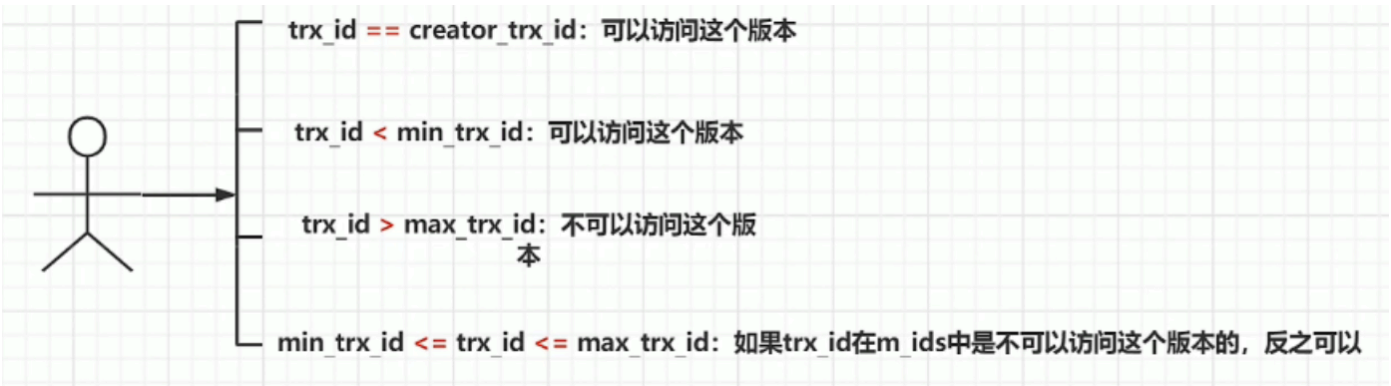
读视图 Read View

当我们用select读取数据时，这一时刻的数据会有很多个版本（例如上图有四个版本），但我们并不知道读取哪个版本，这时就靠readview来对我们进行读取版本的限制，**通过readview我们才知道自己能够读取哪个版本**。

在一个readview快照中主要包括以下这些字段：

m_ids	表示在生成`ReadView`时当前系统中活跃的读写事务的`事务id`列表
min_trx_id	表示在生成`ReadView`时当前系统中活跃的读写事务中最小的`事务id`，也就是`m_ids`中的最小值
max_trx_id	表示生成`ReadView`时系统中应该分配给下一个事务的`id`值
creator_trx_id	表示生成该`ReadView`的事务的`事务id`

readview如何判断版本链中的哪个版本可用呢?



对应以下四种情况，trx_id表示要读取的事务id

（1）如果要读取的事务id等于进行读操作的事务id，说明是我读取我自己创建的记录，可以读。

（2）如果要读取的事务id小于最小的活跃事务id，说明要读取的事务已经提交，那么可以读取。

（3）max_trx_id表示生成readview时，分配给下一个事务的id，如果要读取的事务id大于max_trx_id，说明该id已经不在该readview版本链中了，故无法访问。

（4）m_ids中存储的是活跃事务的id，如果要读取的事务id不在活跃列表，那么就可以读取，反之不行。

MVCC如何实现RC和RR的隔离级别

（1）RC的隔离级别下，每个快照读都会生成并获取最新的readview。

（2）RR的隔离级别下，只有在同一个事务的第一个快照读才会创建readview，之后的每次快照读都使用的同一个readview，所以每次的查询结果都是一样的。

幻读问题

```
快照读：通过mvcc，RR的隔离级别解决了幻读问题，因为每次使用的都是同一个readview。
当前读：通过next-key锁（行锁+gap锁），RR隔离级别并不能解决幻读问题。
```