

1 mysql架构

MYSQL架构:

- * 1 客户端 (navicate\mysql客户端\jdbc)
- * 2 服务器端
 - (连接器(连接池) - show full processlist、
 - 解析器(语法解析、词法解析)、
 - 优化器(cbo成本优化\rbo规则优化) - 可进行相关优化、
 - 执行器、
 - 查询缓存)
- * 3 存储引擎层(myisam\innodb\memory)
- * 4 文件层

2 mysql存储引擎

- * myisam和innodb区别
- * 1 myisam .frm .myi .myd innodb .frm .idb
- * 2 myisam不支持事务 innodb支持事务
- * 3 myisam不支持行锁 innodb支持行锁(行锁锁定的是索引, 如果没有索引, 锁定的是表)
- * 4 myisam不支持外键 innodb支持外键
- * 5 myisam只缓存索引 innodb缓存索引和数据, 所以innodb需要的内存空间更大, 效率更高
- * 6 有主键的情况下, myisam查询顺序就是插入顺序, 而innodb的查询顺序是根据主键排序的
 - * myisam在执行select语句时会给表加读锁, 在执行更新语句时加写锁。
 - * lock tables table_name read;
 - * lock tables table_name write;
 - * unlock tables;
 - * show open tables; -- 查看表锁的情况
 - * innodb-insert语句是间隙(有主键的情况下, 如果没有主键, 锁表)

3 mysql事务

- * 事务的隔离级别--数据库所有
- * 事务的传播属性--spring才有
- * 只有innodb才有事务
- * InnoDB中虽然是repeatable read隔离级别, 但是解决了幻读问题, 通过Next-key Locks+MVCC的方式
- * a(undo-log)
i(lock+mvcc)
d(redo-log)
c(aid)
 - * wal日志(write ahead log)
 - * redo(innodb): 可以实现事务持久性, (在事务提交前, 只要将redo-log持久化即可, 不需要将数据持久化)。当系统崩溃时, 虽然数据没有持久化, 但是
 - * redo已经持久化了, 可以进行数据恢复。(两阶段提交、0-copy) ib_logfile(饭店记账的问题)
 - * 磁盘->kernel space->user space
 - * 0 commit->log buffer(每秒写入)->os buffer(fsync)->磁盘
 - * 1 commit(每个操作)->os buffer->磁盘(默认是这个)
 - * 2 commit(每个操作)->os buffer(每秒调用fsync)->磁盘
 - * undo(innodb): 可以实现事务原子性和mvcc, 在操作任何数据之前, 先将数据备份到一个地方(存储备份数据的地方叫undolog),
 - * 然后再对数据进行修改。如果执行了rollback, 则利用undolog进行数据的恢复
 - * binlog: 主从复制
 - *

- * 事务更新的流程
- * 获取数据->是否在内存中(存在直接返回, 不存在从磁盘中读取内存)->更改数据(内存中)->写入redo
处于prepare->写入bin->提交事务, commit

4 mysql锁

- 表锁
- * 行锁
 - * 排他锁 **for update**,
 - * **delete** 删除一条记录, 先对记录加排他锁, 再执行删除
 - * **update** 如果被更新的列, 修改前后没有导致存储空间发生变化, 那么会先给记录加排他锁, 然后再进行修改
 - * 如果被更新的列, 修改前后导致了存储空间发生了变化, 那么会先给记录加排他锁, 然后删除记录, 再**insert**
 - * **insert** 插入一条记录时, 会增加间隙锁
 - * 共享锁 (S锁) **-select * from user where id = 1 lock in share mode**
 - * 间隙锁
 - * **next-key lock**
 - * 意向共享锁 (innodb)
 - * 意向排他锁 (innodb)
 - * 自增锁
 - * **select**语句默认不加锁
 - *
 - *
 - *
 - * **rc**: 锁定的只有行 (mvcc+行锁)
 - * **rr**: 锁定的是间隙锁-锁定的索引 (mvcc+next-key lock)

5 mysql索引

- * **mysql调优**
- * **1** 打开慢日志开关, 观察一段时间, 会把慢日志写入指定慢日志文件
- * **2 explain** 查看sql情况
- * **3 show profile** 查看具体的资源消耗情况
- * **4 mysql参数调优 (dba做)**
- *
- *
- * 排序有两种形式: 索引排序 **using index** 和 文件排序 **using filesort**
- * 排序调优 -- 可以调大排序内存参数
- * **mysql**从以下几点梳理
- * **1** 存储引擎 (存储方式)
- * **2** 事务 (acid) **-a(redo\lock\undo) binlog**
- * **3** 索引
- * **4** 优化
- * **5 explain**
- * **id select_type table type possible_keys key ke_len ref rows extra**
- * **id**
- * 表示执行的顺序, 规则如下:
- * **1 id**相同: 执行顺序从上到下
- * **2 id**不同: 值越大, 越先执行
- * **select_type :**
- * **simple**: 简单的查询、查询中不包含自查询或者union
- * **primary**: 子查询中, 最外层的查询
- * **subquery**: 在**select**或**where**列表中包含了子查询
- * **derived**: 在**from**列表中包含了子查询被标记为**derived** (衍生), **MySQL**会递归执行这些子查询, 把结果放到临时表中

- * union:如果第二个select出现在UNION之后, 则被标记为UNION, 如果union包含在from子句的子查询中, 外层select被标记为derived
- * union result: UNION 的结果
- * type:
 - * system:表只有一条记录
 - * const: 只有一个匹配行, 用于主键或者唯一索引
 - * eq_ref: 多表连接中使用primary key或者 unique key作为关联条件
 - * ref:非唯一索引扫描, 返回匹配某个单独值的所有行(查询条件使用了普通索引), 一般用于=查询
 - * rang: 检索给定范围的行, 一般用在between < > in等查询
 - * index: 遍历整个索引树
 - * all: 遍历整个表
 - * possible_keys:可能使用到的索引, 一个或者多个(但不一定实际被使用)
 - * key:实际使用的索引, 如果为null, 则没有使用索引, 查询中若使用了覆盖索引, 则该索引仅出现在key列表中
 - * key_len:索引中使用的字节数
 - * ref:显示索引的那一列被使用, 如果可能的话, 是一个常数。那些列或者常量被用于查找索引列上的值(组合索引中, 有那些列被真正使用)
 - * rows:MySQL根据表统计信息及索引选用情况, 估算的找到所需的记录所需要读取的行数
 - * extra:
 - * using filesort: MySQL中无法利用索引完成排序操作称为“文件排序”
 - * using index: 使用索引覆盖
 - * using temporary: mysql使用临时表来处理查询, 常见于排序、子查询、分组查询, 查询返回的数据量太大需要建立一个临时表存储数据, 出现这个sql应该优化
 - * using where: mysql服务器层使用where过滤数据

6 mysql命令

- * SQL命令
- * 1 mysql -h 127.0.0.1 -P 3306 -u root -p(链接数据库)
- * 2 show databases(查看所有数据库)
- * 3 use test(切换至test库)
- * 4 show tables(查看所有表)
- * 5 desc a(查看a表的字段)
- * 6 show index from a(查看a表中的索引)
- * 7 truncate table a(清空表数据)
- * 8 drop table a(删除表)
- * 9 create table a(id int not null auto_increment, name varchar(10), primary key (id))(创建表)
- * 10 ALTER TABLE a ADD INDEX idx_name(name(5))(创建索引)
- * CREATE INDEX idx_name ON a(name(5))
- * 11 alter table a drop index idx_name(删除索引)
- * drop index inx_name on a
- * 12 show variables like '%buffer%'(查看变量)
- * 13 show profiles(没有开启, 返回空)
- * +-----+-----+-----+-----+
- * | Query_ID | Duration | Query |
- * +-----+-----+-----+-----+
- * | 1 | 0.00102925 | select * from a |
- * +-----+-----+-----+-----+
- * 14 set profiling =1(开启) set profiling =0(关闭)
- * 15 show profile [all|cpu|默认是duration] for query 1(查看13中返回的查询列表中的操作耗时详情)
- * +-----+-----+-----+-----+
- * | Status | Duration |
- * +-----+-----+-----+-----+
- * | starting | 0.000036 |

```

* | checking permissions | 0.000011 |
* | opening tables      | 0.000783 |
* | init                | 0.000014 |
* | System lock         | 0.000012 |
* | optimizing          | 0.000004 |
* | statistics          | 0.000011 |
* | preparing           | 0.000009 |
* | executing           | 0.000002 |
* | Sending data        | 0.000021 |
* | end                 | 0.000002 |
* | query end           | 0.000004 |
* | closing tables      | 0.000005 |
* | freeing items       | 0.000087 |
* | cleaning up         | 0.000030 |
* +-----+-----+
* 16 show processlist(查看该数据库有多少个连接, 以及连接情况)
* +-----+-----+-----+-----+-----+-----+-----+-----+
-----+
* | Id      | User | Host          | db  | Command | Time | State | Info
* |-----+-----+-----+-----+-----+-----+-----+-----+
-----+
* | 303612 | root | localhost:63631 | NULL | Sleep   | 3592 |      | NULL
* |-----+-----+-----+-----+-----+-----+-----+-----+
* | 303623 | root | localhost:63659 | test | Query   | 0    | init  | show
processlist |
* | 304005 | root | localhost:64953 | test | Sleep   | 2812 |      | NULL
* |-----+-----+-----+-----+-----+-----+-----+-----+
* | 305822 | root | localhost:54144 | NULL | Sleep   | 8    |      | NULL
* |-----+-----+-----+-----+-----+-----+-----+-----+
* | 305823 | root | localhost:54147 | test | Sleep   | 5    |      | NULL
* |-----+-----+-----+-----+-----+-----+-----+-----+
* +-----+-----+-----+-----+-----+-----+-----+-----+
-----+
* 17 select version();查看MYSQL版本
* 18 help show;查看show的所有情况
*
* 19 begin(后面不需要跟transaction)/start transaction
* 20 set session autocommit=on/off
* 21 commit/rollback
*
* 22 show global status like 'Innodb_page_size';(查看innodb存储引擎页的大小)

```

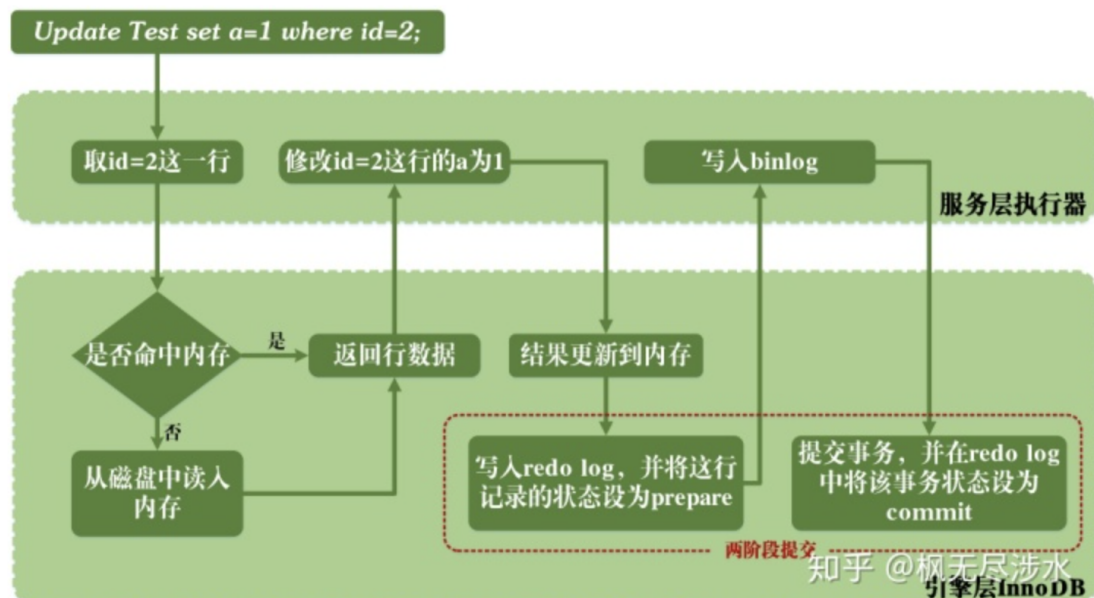
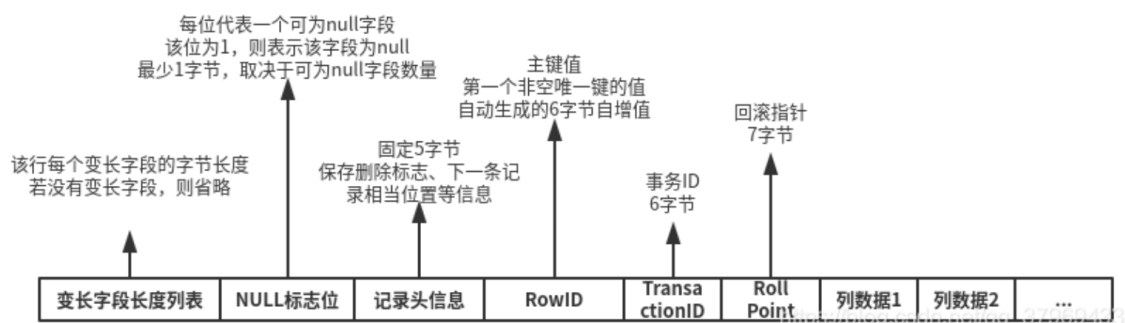
7 innodb行格式和页格式

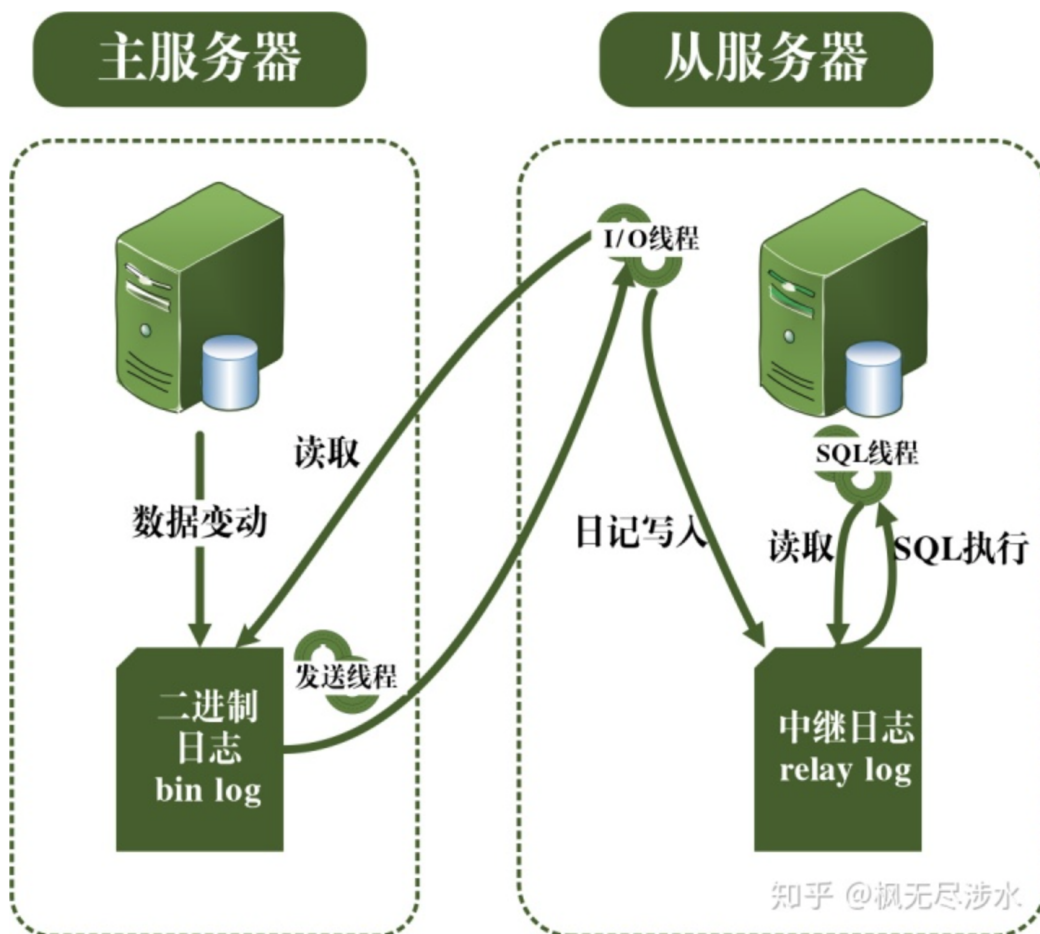
- * 局部性原理: 空间局部性和时间局部性
- * 内存和磁盘: 页为单位进行数据交换的, 一页大小是4k或者8K(根据操作系统不一样), innodb存储引擎, 一次读取16k的数据
- * 内存和cpu: 缓存行, 缓存行的大小是64个字节, 8个long型
- *
 - * innodb行格式
 - * 一行记录可以以不同的格式存储在innodb中, 行格式分别是compact redundant dynamic compressed
 - * compact行格式
 - * 变长字段长度列表 | null标志位 | 记录头信息 | 列1 | 列2 |
 - * innodb的行记录, 除了blobs之外, 大小是65535个字节。

- * 记录头信息中有下一个记录的指针
- *
- * 记录的真实数据除了我们自定义的列的数据外，还有三个隐藏列
- * **row_id** 非必须 6个字节 唯一记录id
- * **transaction_id** 必须 6个字节 事务id--最近一次修改这个记录的事务id是那个
- * **roll_pointer** 必须 7个字节 回滚指针
- * 版本链
- * **readview(m_ids,min_trx_id,max_trx_id,creator_id)**
- *
- *
- * **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，说白了就是已经开启还没有提交或者回滚的事务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。
- * 有了这部分知识，上面Session1在生成查询语句的时候就会附带生成一个ReadView{ m_ids[100]}对象，当查询的时候就把最新的事务id拿去和m_ids里面的内容对比，如果发现存在100，那么就跟随这roll_pointer走到第99次事务继续对比，发现99号事务已经不是活跃的，说明99号已经提交，那么就读取99号的数据显示出来。
- * 有了这个ReadView，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：
- *
- * 如果被访问版本的transaction_id属性值与ReadView中的creator_trx_id值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
- * 如果被访问版本的transaction_id属性值小于ReadView中的min_trx_id值，表明生成该版本的事务在当前事务生成ReadView前已经提交，所以该版本可以被当前事务访问。
- * 如果被访问版本的transaction_id属性值大于ReadView中的max_trx_id值，表明生成该版本的事务在当前事务生成ReadView后才开启，所以该版本不可以被当前事务访问。
- * 如果被访问版本的transaction_id属性值在ReadView的min_trx_id和max_trx_id之间，那就需要判断一下 transaction_id属性值是不是在m_ids列表中，如果在，说明创建ReadView时生成该版本的事务还是活跃 的，该版本不可以被访问；如果不在，说明创建ReadView时生成该版本的事务已经被提交，该版本可以被访问。
- *
- * 如果一行数据65536个字节，会出现行溢出（一行数据超过一页大小）
- * **mvcc**:
- * 在使用rc、rr两种隔离级别的事务在执行普通的select操作时访问记录的版本链的过程中，可以使不同事务的读-写操作并发执行，从而提升系统性能。
- * **rc rr**这两个隔离级别的最大不通时，生成readview的实际不通，rc在每一次进行select操作前都会生成readview，而rr只在第一次进行select
- * 操作前生成一个readview，之后的查询操作都重复使用这个readview

名称	中文名	占用空间	简单描述
File Header	文件头部	38字节	页的一些通用信息
Page Header	页面头部	56字节	数据页专有的一些信息
Infimum + Supremum	最小记录和最大记录	26字节	两个虚拟的行记录
User Records	用户记录	不确定	实际存储的行记录内容
Free Space	空闲空间	不确定	页中尚未使用的空间
Page Directory	页面目录	不确定	页中的某些记录的相对位置
File Trailer	文件尾部	8字节	校验页是否完整

https://blog.csdn.net/java_ssehehe





之所以需要实现主从复制，实际上是由实际应用场景所决定的。主从复制能够带来的好处有：

参考文章：<https://blog.csdn.net/xiewenfeng520/article/details/99715680>

mvcc参考文章：https://blog.csdn.net/qg_31821675/article/details/71135933

主从复制、undo\redo\binlog：https://blog.csdn.net/weixin_42513602/article/details/112270444

关于MySQL的InnoDB的MVCC原理，很多朋友都能说个大概：

每行记录都含有两个隐藏列，分别是记录的创建时间与删除时间

每次开启事务都会产生一个全局自增ID

在RR隔离级别下

INSERT -> 记录的创建时间 = 当前事务ID，删除时间 = NULL

DELETE -> 记录的创建时间不动，删除时间 = 当前事务ID

UPDATE -> 将记录复制一次

老记录的创建时间不动，删除时间 = 当前事务ID

新记录的创建时间 = 当前事务ID，删除时间 = NULL

SELECT -> 返回的记录需要满足两个条件：

创建时间 <= 当前事务ID (记录是在当前事务之前或者由当前事务创建的)

删除时间 == NULL || 删除时间 > 当前事务ID (记录是在当前事务之后被删除的)

但实际上，这个描述是很不严格的，问题有以下几点：

1. 每条记录含有的隐藏列不是两个而是三个

它们分别是：

DB_TRX_ID, 6byte, 创建这条记录/最后一次更新这条记录的事务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, 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

另外, 每条记录的[头信息 \(record header \)](#)里都有一个专门的bit (**deleted flag**) 来表示当前记录是否已经被删除

2. 记录的历史版本是放在专门的rollback segment里 (undo log)

UPDATE非主键语句的效果是

老记录被复制到rollback segment中形成undo log, DB_TRX_ID和DB_ROLL_PTR不动

新记录的DB_TRX_ID = 当前事务ID, DB_ROLL_PTR指向老记录形成的undo log

这样就能通过DB_ROLL_PTR找到这条记录的历史版本。如果对同一行记录执行连续的update操作, 新记录与undo log会组成一个链表, 遍历这个链表可以看到这条记录的变迁)

3. MySQL的一致性读, 是通过一个叫做[read view](#)的结构来实现的

read_view中维护了系统中活跃事务集合的快照, 这些活跃事务ID的**最小值为up_limit_id, *最大值为low_limit_id*** (不要搞反了!!!)

附上源码注释以便于理解

```
trx_id_t low_limit_id; // The read should not see any transaction with trx id >= this value. In
other words, this is the "high water mark".
trx_id_t up_limit_id; // The read should see all trx
ids which are strictly smaller (<) than this value. In other words, this is the "low water mark".
```

SELECT操作返回结果的可见性是由以下规则决定的：

DB_TRX_ID < up_limit_id -> 此记录的最后一次修改在read_view创建之前, 可见

DB_TRX_ID > low_limit_id -> 此记录的最后一次修改在read_view创建之后, 不可见 -> 需要用DB_ROLL_PTR查找undo log(此记录的上一次修改), 然后根据undo log的DB_TRX_ID再计算一次可见性

up_limit_id <= DB_TRX_ID <= low_limit_id -> 需要进一步检查read_view中是否含有DB_TRX_ID

DB_TRX_ID ∉ read_view -> 此记录的最后一次修改在read_view创建之前, 可见

DB_TRX_ID ∈ read_view -> 此记录的最后一次修改在read_view创建时尚未保存, 不可见 -> 需要用DB_ROLL_PTR查找undo log(此记录的上一次修改), 然后根据undo log的DB_TRX_ID再从头计算一次可见性

经过上述规则的决议, 我们得到了这条记录相对read_view来说, 可见的结果。

此时, 如果这条记录的delete_flag为true, 说明这条记录已被删除, 不返回。

如果delete_flag为false, 说明此记录可以安全返回给客户端

4. 用MVCC这一种手段可以同时实现RR与RC隔离级别

它们的不同之处在于：

RR：read view是在**first touch read**时创建的，也就是执行事务中的第一条SELECT语句的瞬间，后续所有的SELECT都是复用这个read view，所以能保证每次读取的一致性（可重复读的语义）

RC：每次读取，都会创建一个新的read view。这样就能读取到其他事务已经COMMIT的内容。

所以对于InnoDB来说，RR虽然比RC隔离级别高，但是开销反而相对少。

补充：RU的实现就简单多了，不使用read view，也不需要管什么DB_TRX_ID和DB_ROLL_PTR，直接读取最新的record即可。

- InnoDB存储引擎中，每行数据包含了一些隐藏字段 DATA_TRX_ID，DATA_ROLL_PTR，DB_ROW_ID，DELETE BIT
- DATA_TRX_ID 字段记录了数据的创建和删除时间，这个时间指的是对数据进行操作的事务的id
- DATA_ROLL_PTR 指向当前数据的undo log记录，回滚数据就是通过这个指针
- DELETE BIT位用于标识该记录是否被删除，这里的不是真正的删除数据，而是标志出来的删除。真正意义的删除是在mysql进行数据的GC，清理历史版本数据的时候。