

MySQL InnoDB 存储引擎

你了解MVCC么？它是怎么工作的？

InnoDB和MyISAM的区别？

InnoDB支持事务，提供了行级锁和外键的约束。适合处理大容量的数据库系统，由于行锁的粒度小，所以并发度比MyISAM高。不支持全文检索，需要扫描全表。存在自适应hash索引。

MyISAM不支持事务，仅支持表锁，适合读多写少的业务场景，MyISAM引擎保存了表的行数，不需要进行全表扫描。

问题

1、如果你要删除一个表里面的前 10000 行数据，有以下三种方法可以做到：第一种，直接执行 delete from T limit 10000; 第二种，在一个连接中循环执行 20 次 delete from T limit 500; 第三种，在 20 个连接中同时执行 delete from T limit 500。**你会选择哪一种方法呢？为什么呢？**

最好使用第二种方式，

第一种方式，单个语句的占用时间最长，锁的时间也比较长；大事务还会导致主从复制延迟

第三种方式，会造成锁冲突

日志

(脏页) Mysql为什么会在运行时抖一下？

孔乙己

在InnoDB 在处理更新语句的时候，只做了写日志这一个磁盘操作。

这个日志叫作 redo log （重做日志），也就是《孔乙己》里咸亨酒店掌柜用来记账的粉板，在更新内存写完 redo log 后，就返回给客户端，本次更新成功。

♡ 做下类比的话，掌柜记账的账本是数据文件，记账用的粉板是日志文件（redo log），掌柜的记忆就是内存。

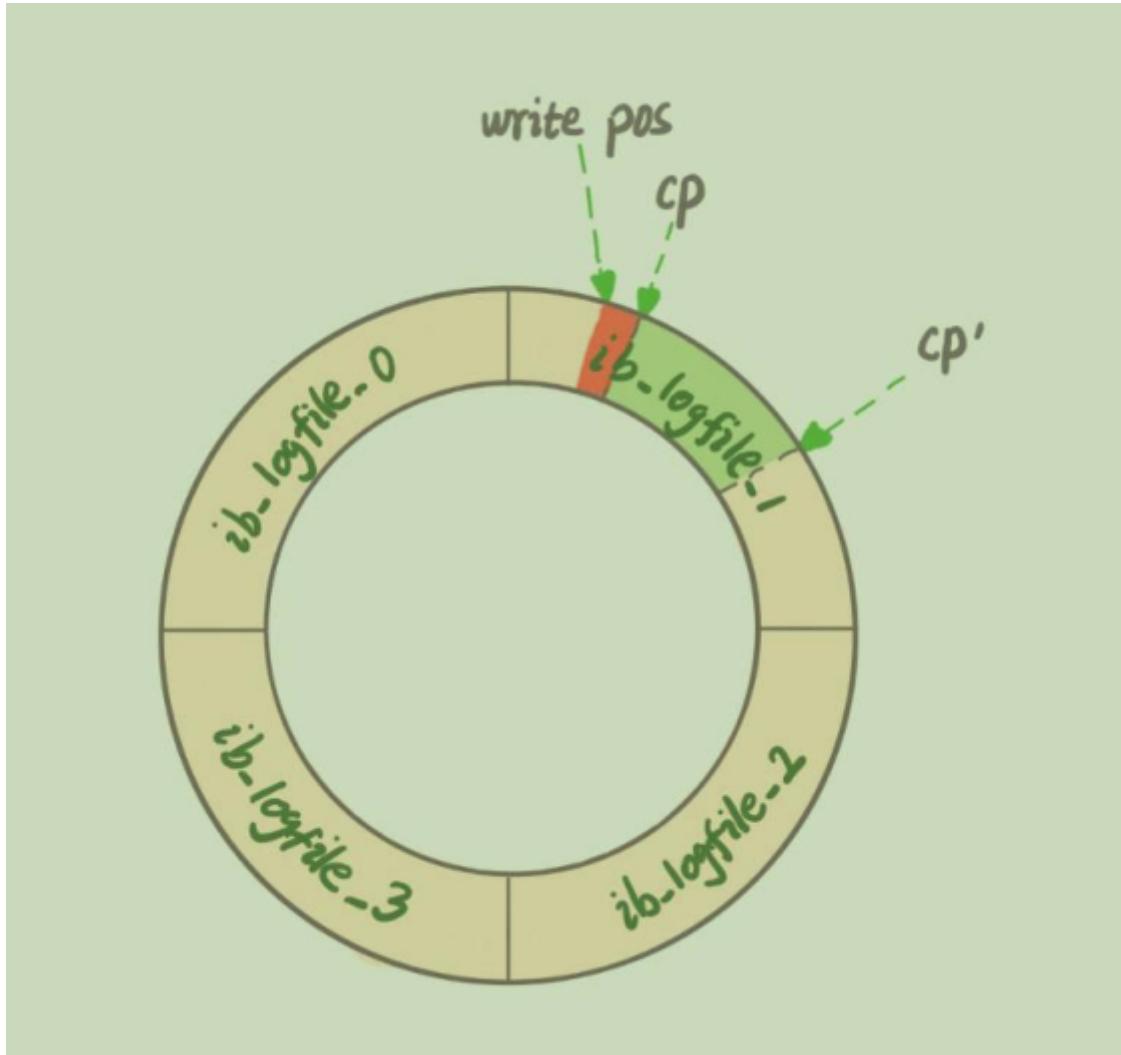
脏页、干净页

- 当内存中的数据页和磁盘数据页内容不一致时，称为脏页；
- 一致时，称为干净页

(一) redo log 满了，在刷脏页

在刷脏页，当redo log 满时，停止更新操作，把checkpoint 往前推进，redo log 留出空间可以继续写。

具体流程：当redo log 满时，checkpoint位置从 cp推近到cp'，将这段之间的日志flush 到磁盘上，之后从这段区域就可以在写入日志了



(二) 系统内存不足

先将脏页写到磁盘，InnoDB用缓冲池（buffer pool）管理内存，缓冲池中的内存页有**三种状态**：

- 还没有使用
- 使用了并且是干净页
- 使用了并且是脏页

当没有内存时，就必须去缓冲池中去申请一个内存，

- 如果淘汰的是一个干净页那么直接释放复用，
- 如果是脏页那么必须先刷到磁盘，变成干净页复用。

(三) 在MySQL空闲的时候

(四) 在MySQL正常关闭的时候，把所有的脏页刷到磁盘中

以上4种方式对性能的影响：

1. 一个查询要淘汰的脏页个数太多，导致查询的相应时间增长。
2. 日志写满，更新全部堵住，写性能为0

InnoDB刷脏页的控制策略

innodb_io_capacity 设置IO的磁盘能力，这样InnoDB才能知道全力刷脏页的时候，可以刷多快。

InnoDB 的刷盘速度就是要参考这两个因素

- 脏页比例：不要让它接近75%
- redo log 写盘速度

刷脏页策略

一旦一个查询请求在执行过程中，先flush 掉一个脏页时，这个查询要比平时慢一些。因为在刷脏页时，遇到脏页时会继续蔓延刷掉挨着的脏页。

通过设置下面的参数，设置每次刷脏页是否只刷自己的。

`innodb_flush_neighbors`

值为0表示不找邻居，值为1表示找邻居

- 在机械硬盘的时候，刷邻居这种优化可以减少随机IO，提升系统性能。
- 使用SSD（固态硬盘）IOPS比较高时，建议把设置为不找邻居。

在MySQL 8.0 中 `innodb_flush_neighbors` 设置默认为0；只刷自己。

WAL

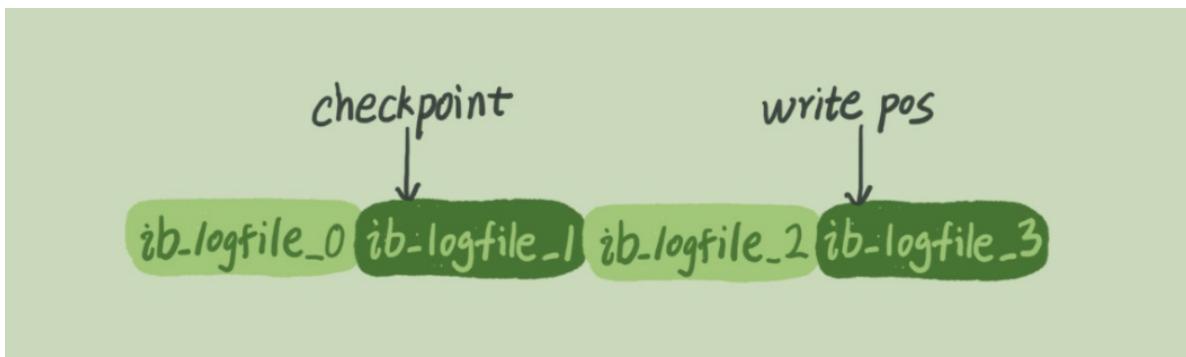
WAL (write ahead logging) **先写日志，在写磁盘。**

数据库将随机写转换成顺序写，大大提升了数据库的性能。

redo log 和binlog

(一) redo log (粉板) InnoDB 引擎层特有的日志。

redo log 是固定大小的，比如可以配置一组4个文件，每个文件的大小是1GB，那么这块粉板总共记录了4GB的操作。从头开始写，写到末尾又回到开头循环写，如下图所示



- `write pos` **是当前记录的位置**，一边写一边往后移，写到3号文件末尾后就回到0号文件开头。
- `checkpoint` **是当前要擦除的位置**，也是往后推移并且循环的，擦除记录前要把记录更新到数据库中。

`write pos` 和`checkpoint` 中间空着的部分，可以用来记录新的操作。如果`write pos` 追上`checkpoint`，表示粉板满了，这时候不执行新的更新，停下来先擦除一些记录(刷)，把`checkpoint`推进一下。

有了redo log，InnoDB就可以在发生故障时，之前提交的记录不会丢失，这个能力称为**crash-safe**。

(二) binlog 归档日志

service 层日志

- 一是用于复制。master 把它的binlog 文件传输给 slave 来达到主从一致性。
- 二是用于数据恢复。例如还原备份后，可以重新执行binlog 文件使得数据库保持最新的状态。

binlog日志可以选择三种模式

- statement：基于SQL语句的复制，每一条修改都会记录下来。

| 产生的binlog 比较小，会造成主从数据不一致。

- row：基于行的复制，不记录每一条具体执行的SQL，而是记录哪条数据被修改了，以及修改后的样子。

| 产生的binlog 比较大。主从复制不会造成不一致。

- mixed：混合复制模式，同时使用statement和row模式。现根据 statement模式保存 binlog，如果statement模式无法复制的操作会使用row模式保存binlog

| MySQL 5.7 之前默认是 statement，高版本默认是 ROW模式。可以清楚的记录每行数据修改的细节

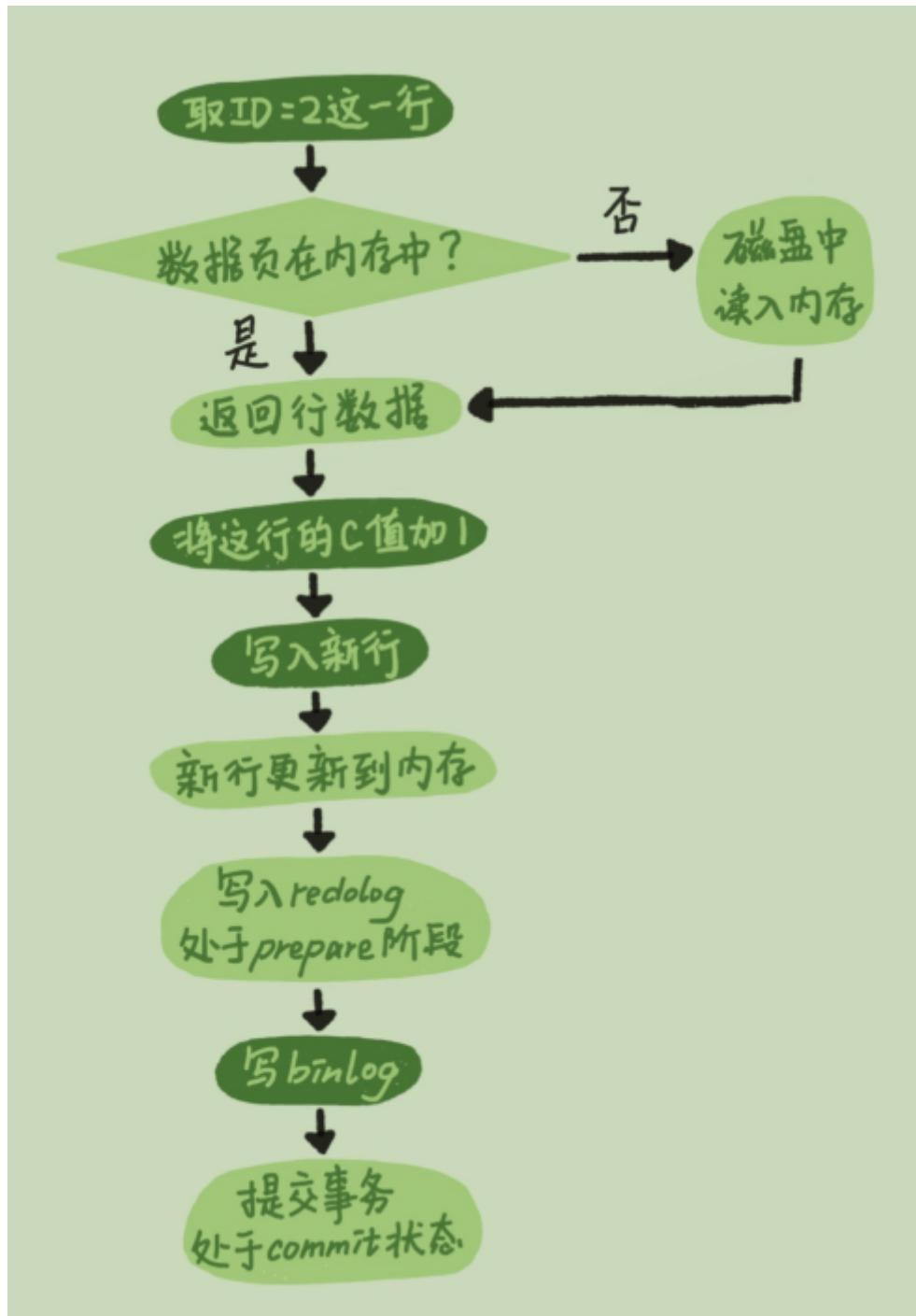
执行器和InnoDB引擎在执行简单update 语句的内部流程。

| 比如“给 ID=2 这一行的 c 字段加 1 ”。

1. 执行器先找引擎取ID = 2这一行。ID 是主键，引擎直接用树搜索找到这一行。如果ID这一行所在的数据页在内存中，直接返回给执行器。否则从磁盘中读取到内存，然后在返回。
2. 执行器就会拿到引擎给的这行数据，把这个值加1，得到新的一行数据，再调用引擎接口写入这行新数据。
3. 引擎将这行数据更新到内存中，同时将更新操作记录更新到redo log 里面，此时redo log 处于 prepare 状态，然后告知执行器执行完成，随时可以提交事务。
4. 执行器生成这个操作的binlog，并把binlog 写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 日志改成 commit 状态，更新完成。

prepare 和 commit 这就是两阶段提交

浅色框表示是在 InnoDB 内部执行的，深色框表示是在执行器中执行的。



为什么需要两阶段提交？

所谓的两阶段提交：先在InnoDB存储引擎写redo log使其进入prepare状态，再在执行器写binlog，随后在存储引擎提交事务，使redo log处于commit状态。

如果不使用两阶段提交，可能会出现两种情况：

一、先写redo log 再写binlog。这种情况会造成，当写完redo log 后，MySQL异常重启，导致没有把日志写到binlog 文件中，在恢复数据时，因为只把数据写到了redo log 日志文件中，binlog 日志文件中没有写入更新。导致在用binlog 恢复数据时，与原库的数据不同。

二、先写binlog 再写 redo log。如果在binlog 写完之后crash，由于 redo log 没有写，崩溃恢复这个事务无效，所以这一行的值为0。但是 binlog里面已经记录了把c从0改为1这个日志，所以在恢复的时候多了一个事务出来，与原库值不同。

综上所述：如果不使用两阶段提交，那么数据库的状态就有可能和它的日志恢复出来的库状态不一致。

redo log 和binlog有什么不同？

1. redo log 是InnoDB引擎特有的； binlog 是MySQL service 层实现的，所有引擎都可以使用
2. **redo log 是物理日志**，记录的是在某个页做了什么修改； **binlog 是逻辑日志**，记录的是这个语句的原始逻辑，比如给ID=2这一行的C字段加一。
3. redo log 是循环写的，空间固定会用完； binlog 是追加写入的，写到一定大小会切换到下一个，并不会覆盖日志。

事务

隔离级别

隔离级别越高，效率越低。

读未提交 一个事务还未提交时，它所做的更改可以被另一个事务看到

读提交 一个事务提交后，它所做的更改才可以被其他事物看到。

可重复读 一个事务执行过程中看到的数据，总是跟这个事务启动时看到的事务是一致的。

串行化 对同一行数据加锁，按照事务串行化执行。

读提交和可重复读的区别？

事务A	事务B
启动事务 查询得到值1	启动事务
	查询得到值1
	将1改成2
查询得到值V1	
	提交事务B
查询得到值V2	
提交事务A	
查询得到值V3	

未提交读 V1为2， V2为2， V3为2

读提交 V1为1， V2、 V3为2

可重复读 V1为1， V2为1， V3为2 可重复读隔离级别，保证了事务在执行期间看到的数据是一致的。

串行化 事务B在执行更改操作会被锁住，直到事务A提交后才可以继续执行。

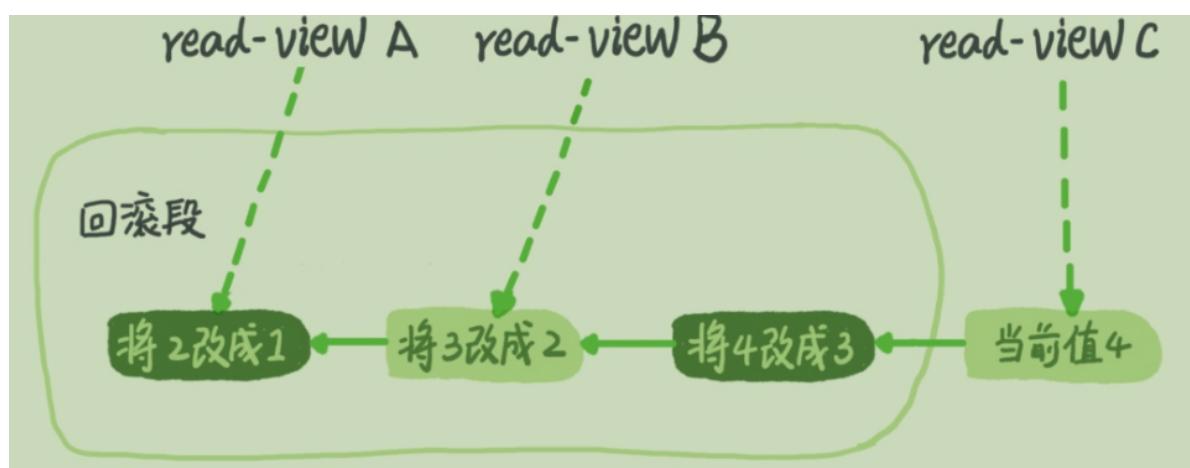
实际上，数据库会创建一个视图，访问的时候以视图的逻辑结果为准。

- 可重复读：这个视图在事务启动时创建，整个事务存在期间都用这个视图。
- 读提交：在每个SQL语句开始执行的时候被创建
- 读未提交：直接返回记录上的最新值，没有视图概念。

事务隔离的实现

在MySQL中，每条记录在更新的时候都会同时记录一条回滚操作。记录的最新值，通过回滚操作都可以得到前一个状态的值。

假设一个值从 1 被按顺序改成了 2、3、4，在回滚日志里面就会有类似下面的记录。



当前值是 4，但是在查询这条记录的时候，不同时刻启动的事务会有不同的 read-view。如图中看到的，在视图 A、B、C 里面，这一个记录的值分别是 1、2、4，同一条记录在系统中可以存在多个版本，就是数据库的多版本并发控制（MVCC）。对于 read-view A，要得到 1，就必须将当前值依次执行图中所有的回滚操作得到。

回滚日志何时删除？

系统会判断，当没有事务在需要用到这些回滚日志时，回滚日志会被删除。

什么时候才不需要了呢？

就是当系统里没有比这个回滚日志更早的read-view 的时候。

为什么不建议使用长事务？

长事务意味着系统里面会存在很老的事务视图。由于这些事务随时可能访问数据库里面的任何数据，所以事务提交之前，数据库里面它可能用到的回滚记录都必须保留，这样会导致**大量占用存储空间**。

另一方面 长事务还占用锁资源。

事务的启动方式

- 显示启动事务语句，begin，start transaction。提交 commit 回滚 rollback
- set autocommit = 0，关掉自动提交。意味着如果你执行了一个 select 语句，这个事务就启动了，并且不会自动提交。这个事务持续存在直到你主动执行了 commit 或者 rollback。或者断开连接。

 **会导致长事务**

建议使用 set autocommit = 1，通过显示语句启动事务。

但是会出现多一次交互的问题:即每次都要主动执行一次 begin 。但是如果执行 commit work and chain ，则是提交事务并自动启动下一个事务，这样就省去了begin 语句的开销。

可重复读和提交读区别

一致性视图-事务隔离级别

begin/start transaction 命令并不是一个事务的起点，在执行他们之后的第一个操作InnoDB表的语句，事务才真正启动。如果你想马上启动一个事务，可以使用 start transaction with consistent snapshot 这个命令。

一致性视图

一个数据版本，对于一个事务视图来说，除了自己更新总是可见外，有如下三种情况

- 版本未提交，不可见
- 版本已提交，但是在视图创建后提交的，不可见
- 版本已提交，而且是在视图创建前提交的，可见。

事务 A、B、C 的执行流程

图1

事务A	事务B	事务C
start transaction with consistent snapshot;		
	start transaction with consistent snapshot;	
		update t set k=k+1 where id=1;
	update t set k=k+1 where id=1; select k from t where id=1;	
select k from t where id=1; commit;		
	commit;	

事务 C 没有显式地使用 begin/commit，表示这个 update 语句本身就是一个事务，语句完成的时候会自动提交。事务 B 在更新了行之后查询；事务 A 在一个只读事务中查询，并且时间顺序上是在事务 B 的查询之后。

事务 B 查到的 k 的值是 3，而事务 A 查到的 k 的值是 1

在MySQL 中，有两个视图的概念

- 一个是 view。它是用于查询语句定义的虚拟表，在调用的时候执行查询语句并生成结果。创建视图的语法 create view，而他的查询方法和表一样
- 另一个是InnoDB 在实现MVCC时用到的一致性视图。即 consistent read view，用于支持读提交RC (read committed)、可重复读

(repeatable read) 隔离级别的实现。他没有物理结构，作用是事务执行期间用来定义“我能看到什么数据”

"快照" 在MVCC 里是怎么工作的？

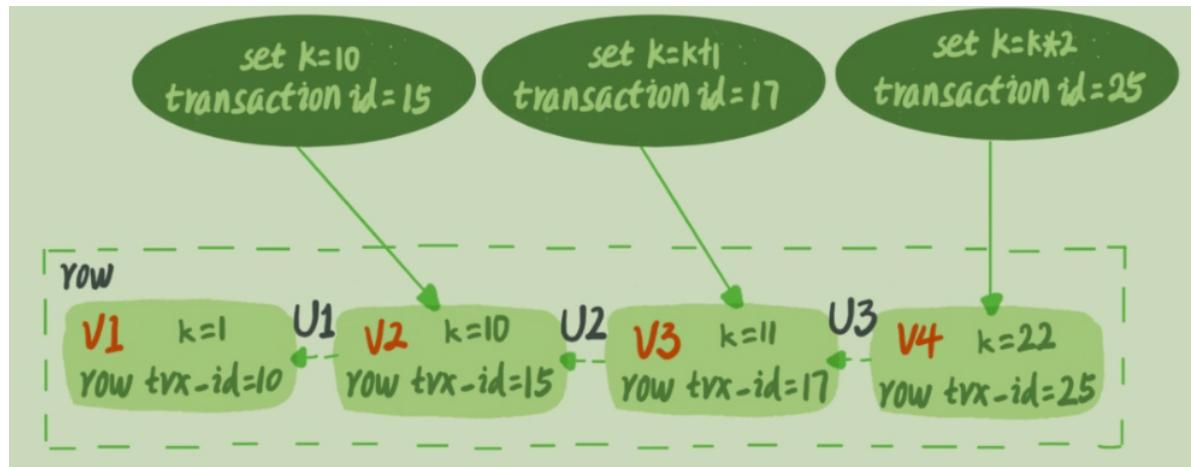
在可重复读隔离级别下，事务在启动的时候就拍个快照，这个快照是基于整库的。

InnoDB 里面每个事务有一个唯一的事务ID，叫做transcation id，他在事务开始的时候向InnoDB的事务系统申请的，是按申请顺序严格递增的。

每行数据也是有多个版本的，每次事务更新数据的时候，都会生成一个新的数据库版本，并且把 transcation id 赋值给这个**数据库版本的事务 ID，记为 row trx_id**。同时旧的数据库版本要保留，并且在新的数据库版本中，能够有信息可以直接拿到它。

数据表中的一行记录，其实可能有多个版本 (row)，每个版本有自己的 row trx_id

图2



图中虚线框里是同一行数据的 4 个版本，当前最新版本是 V4，k 的值是 22，它是被 transaction id 为 25 的事务更新的，因此它的 row trx_id 也是 25。

图中的虚线箭头 (U1、U2、U3) 就是Undo log

一个事务只需要在启动的时候声明说：以我启动的时刻为准，如果一个数据库版本是在我启动之前生成的，就认；如果是我启动以后生成的，我就不认，我必须找到它的上一个版本。如果上一个版本也不可见，那就继续往前找。**如果是这个事务自己更新的数据，他自己还是要认的**

视图数组、高水位

- InnoDB为每个事务构造了一个数组，用来保存这个事务的启动瞬间，当前正在“活跃”的所有事务ID。“活跃”指的是，启动了还没提交。
- 数组里面事务ID 的最小值记为低水位，当前系统里面已经创建过的事务ID的最大值加1记为高水位。

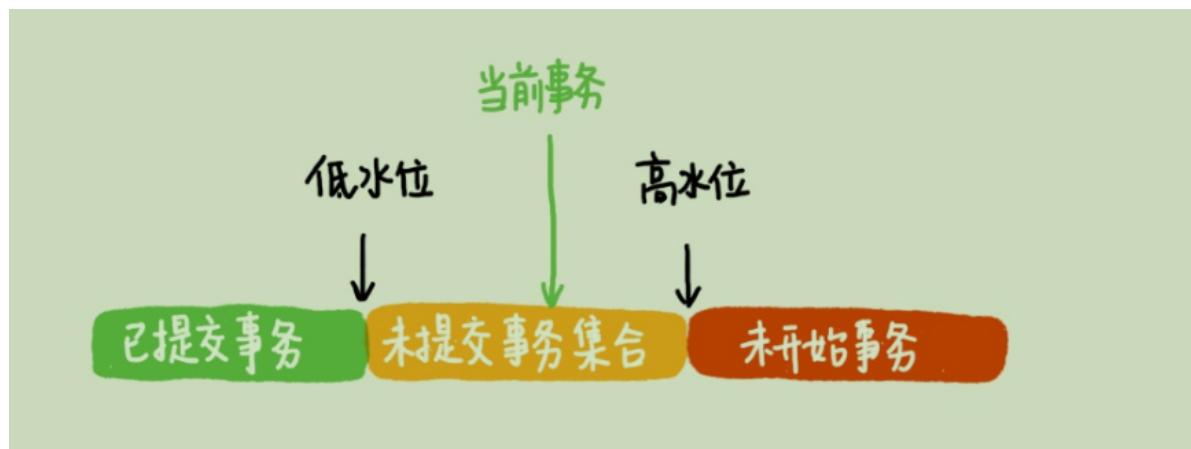
视图数组

视图数组和高水位，就组成了当前事务的一致性视图 read-view。

数据库版本的可见性规则，就是基于数据的row trx_id 和这个一致性视图的对比结果得到的

这个**视图数组**把所有的 row trx_id 分成了几种不同的情况。

数据库版本可见性规则



1. 如果落在绿色部分，表示这个版本已提交的事务或者是当前事务自己生成的，这个事务是可见的
2. 如果落在红色部分，表示这个版本是由将来启动的事务生成的，是不可见的。
3. 如果落在黄色部分，

1. 若 row trx_id 在数组中，表示这个版本是由还没提交的事务生成的，**不可见**
2. 若 row trx_id 不在数组中，表示这个版本是已经提交了的事务生成的，**可见**

比如，对于图 2 中的数据来说，如果有一个事务，它的低水位是 18，那么当它访问这一行数据时，就会从 V4 通过 U3 计算出 V3，所以在它看来，这一行的值是 11。

InnoDB利用了所有数据都有多个版本的这个特性，实现了秒级创建快照的能力。

当前读

- 更新数据都是先读后写的，而这个读，只能读当前的值，称为“当前读”(current read)。

⌚ 事务可重复读的能力是怎么实现的？

- 可重复读核心就是一致性读(consistent read)；
- 而事务更新数据时，只能用当前读。如果当前的记录的行锁被其他事务占用的话，就要进入锁等待。

读提交和可重复读的逻辑的区别是？

- 在可重复读隔离级别下，只需要在事务开始的时候创建一致性视图，之后事务的其他查询都共用这个一致性视图。
- 在读提交隔离级别下，每个语句执行前都会重新算出一个新视图。

总结

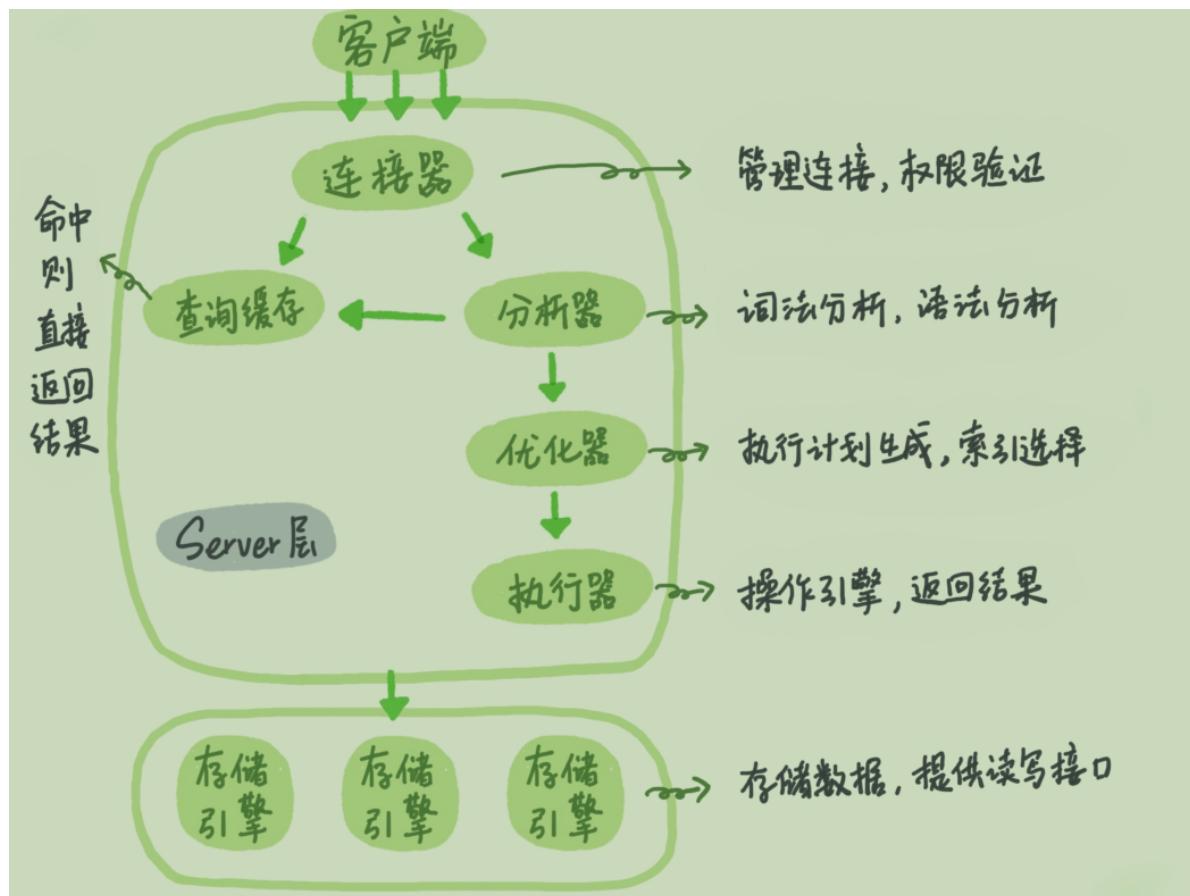
InnoDB 的行数据有多个版本，每个数据版本有自己的 row trx_id，每个事务或者语句都有自己的一致性视图。普通查询语句是一致性读，一致性读会根据 row trx_id 和一致性视图确定数据版本的可见性。

- 对于可重复读，查询只承认在事务启动前就已经提交完成的数据；
- 对于读提交，查询只承认在语句启动前就已经提交完成的数据；

而当前读，总是读取已经提交完成的最新版本。

你知道执行一条语句的流程么？

首先看一下流程图



连接器、查询缓存、分析器、优化器、执行器、存储引擎

(一) 连接器

- 首先对客户端与服务器建立连接（使用TCP）对输入账号密码进行验证，然后验证权限是否通过。如果已经连接上，管理员对已连接的这个账户修改权限不会立即生效而是等到下一次连接才会生效。
- 客户端连接成功后，如果长时间没动静，连接器会将其断开。默认是8小时。

使用长连接会导致MySQL内存涨的非常快，这是因为MySQL在执行过程中临时使用的内存是管理在连接对象里面的。这些资源只有在断开才会被释放。

如何解决长连接导致的OOM？

- 定期断开连接

- 使用MySQL5.7以上版本，通过执行mysql_reset_connection 重新初始化连接资源。这个过程不需要重连和权限验证，会将连接恢复到刚刚创建时的状态。

(二) 查询缓存

MySQL拿到一个查询请求时，先到缓存中去看看，如果命中直接返回给客户端。但是大多数情况下查询缓存并不是很理想，适合于静态的表。在MySQL8.0以上版本去除掉查询缓存。

(三) 分析器

先做词法分析，识别SQL语句分别是什么，代表什么意思。

然后做语法分析，根据词法分析的结果语法分析器将根据语法规则，判断这个SQL是否合法。

(四) 优化器

如果表中有多个索引的话会决定使用哪个索引，或者多表关联决定表的连接顺序

(五) 执行器

判断这个表有没有执行的权限，如果有权限就去执行对应引擎的接口。

索引

索引都包含哪些

(一) 按逻辑分类

- **唯一索引** 数据列不允许重复，允许为空，可以有多个唯一索引
- **主键索引** 一张表只有一个主键，不允许为空，不允许重复
- **普通索引**
- **全文索引** MyISAM支持全文索引，用于解决模糊查询效率低的问题。

(二) 按物理分类

聚簇索引

按照每张表的主键构建一颗B+数，叶子节点存放整张表的行记录。

优点

- 数据访问更快，因为聚簇索引将索引和数据保存在同一个B+树上。
- 对主键的排序查找和范围查找速度更快

缺点

- 插入速度严重依赖于插入顺序，按照主键的顺序插入，如果不按主键顺序插入会造成页分裂，严重影响性能
- 更新主键的代价很高，因此将会导致被更新的行移动

非聚簇索引

- 非聚簇索引总是需要二次查找，先查找到主键值，再根据主键值找到数据行的数据页。
- 叶子节点不包含行记录的全部数据
- 会造成回表问题

为什么不用有序数组、hash、二叉树实现？

hash：首先说以下hash索引，它是以键值对的形式存储的，底层使用数组+链表实现。经过哈希函数计算出key存放的位置，当哈希函数计算出来的值相同时，会使用拉链法存放在数组的节点上。因为哈希值是无序的，所以不能范围查询。适合等值查询，例如Memcached。

有序数组 有序数组适合等值查询和范围查询，因为是有序的，可以使用二分法对信息进行查找，时间复杂度为 $O(\log n)$ ，但是更新慢，适合静态数据存储。

二叉树搜索树实现 二叉搜索树的特点是左子树 < 父节点 < 右子树。查询和更新的时间复杂度都是 $O(\log n)$ 但是它的树高限制了对磁盘的访问效率。例如：

- 1棵100万节点的AVL树，树高20，一次查询需要访问20个数据块，对于机械磁盘，读一个数据块需要10ms左右的时间，也就是说对于一个100万行的表需要 $20 * 10 = 200\text{ms}$ 的时间，查询效率低。
- **需要一个N叉树来解决磁盘访问效率低的问题。**

N叉树 假如树高为4的情况下，考虑到树根的数据库总是在内存中，所以查找一个数据最多需要3次磁盘访问，树的第二层也有很大概率在内存中。

怎么查看索引是否生效？什么情况下索引会失效？

你知道有哪些种类的索引？

你平时是怎么进行SQL优化的？

什么是聚簇索引和非聚簇索引？

主键索引又被称为聚簇索引，非主键索引被称为二级索引。

在InnoDB引擎中，聚集索引就是按照每张表的主键构造一棵B+树，同时**叶子节点中存放的即为整张表的行记录数据**，也将聚集索引的叶子节点称为数据页。

聚簇索引的好处是：

- 对于主键的排序查找和范围查找速度非常快。

聚簇索引和非聚簇索引的主要区别是：

- 聚簇索引的叶子节点是数据节点，
- **而非聚簇索引的叶子节点存储的仍然是索引节点**，只不过有指向对应数据块的指针。
- 非聚簇索引会造成回表问题

区别这两者的区别就是来对比InnoDB和MYISAM的数据结构了。假如我们有一个表原始数据如下所示：

row number	col1	col2
0	99	8
1	12	56
2	3000	62
...
9997	18	8
9998	4700	13
9999	3	93

什么是回表？主键索引和非主键索引的区别？

InnoDB引擎的**主键索引**存储的是**行数据**，**二级索引的叶子节点**存储的是**索引数据**以及**对应的主键**，索引回表就是根据索引进行条件查询，回到主键索引树进行搜索的过程。

回表问题涉及到**主键索引**和**普通索引**的问题。

- 如果使用**主键索引**查询一个数据，那么在查询的时候只需要根据 ID 搜索这棵B+树。
- 使用**普通索引**，会先根据普通索引查到主键ID的值，在根据ID主键索引查询一次，这就是回表问题。

基于**非主键索引**的查询会多扫描一颗索引树，因此我们在查询的时候尽量使用**主键索引**

怎么解决回表问题

建立联合索引进行覆盖索引。

建表

```
mysql> create table T (
    -> ID int primary key,
    -> k int NOT NULL DEFAULT 0,
    -> s varchar(16) NOT NULL DEFAULT '',
    -> index k(k))
    -> engine=InnoDB;
```

```
Query OK, 0 rows affected (0.42 sec)
```

```
mysql> insert into T values(100,1, 'aa'),(200,2,'bb'),  
(300,3,'cc'),(500,5,'ee'),(600,6,'ff');
```

```
Query OK, 5 rows affected (0.09 sec)
```

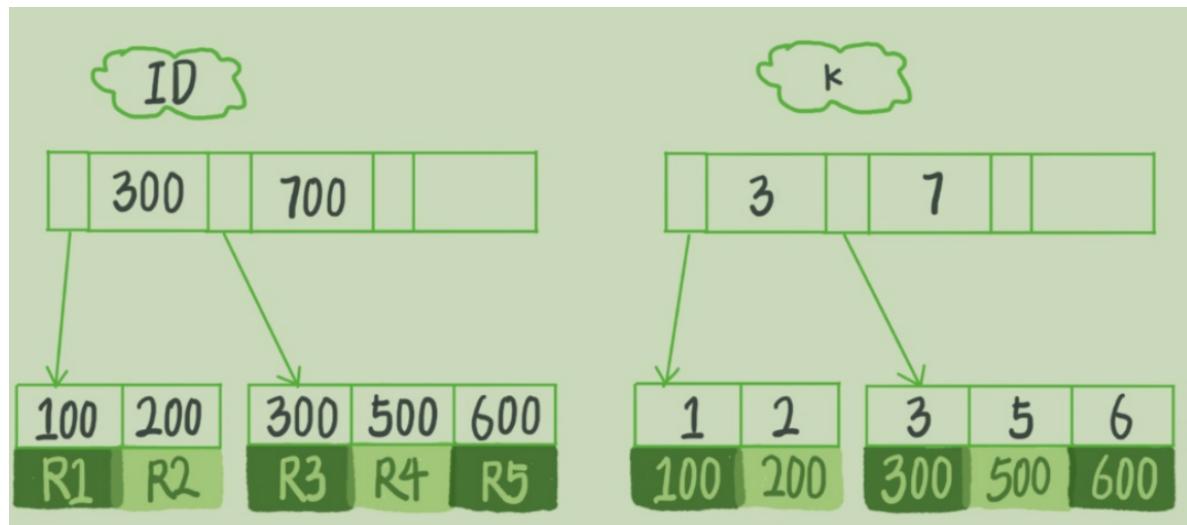
```
Records: 5 Duplicates: 0 Warnings: 0
```

```
mysql> select * from T;
```

ID	k	s
100	1	aa
200	2	bb
300	3	cc
500	5	ee
600	6	ff

```
5 rows in set (0.00 sec)
```

表结构



我们一起来看看这条 SQL 查询语句的执行流程：

1. 在 **k** 索引树上找到 **k=3** 的记录，取得 **ID = 300**；
2. 再到 **ID** 索引树查到 **ID=300** 对应的 **R3**；
3. 在 **k** 索引树取下一个值 **k=5**，取得 **ID=500**；
4. 再回到 **ID** 索引树查到 **ID=500** 对应的 **R4**；
5. 在 **k** 索引树取下一个值 **k=6**，不满足条件，循环结束

回到主键索引树搜索的过程，我们称为**回表**

```

mysql> explain select ID from T where k between 3 and 5;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE    | T     | index | k           | k   | 4      | NULL | 5  | Using where; Using index |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)

mysql> explain select * from T where k between 3 and 5;
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE    | T     | range | k           | k   | 4      | NULL | 2  | Using where |
+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

```

使用了覆盖索引，这时只需要查找ID值，而ID值已经在K索引树上，可以直接查找结果，不需要回表。

最左前缀原则

比如说一个学生表中有id、name、age字段，现在需要查找姓名为李四的人并且年龄为23的人，可以使用name_age 联合索引。当要查找姓名为李四的人可以使用最左前缀匹配索引。但是要查找年龄为23 的人索引就失效了

前缀索引（字符串创建索引的方法？）

使用场景：邮箱

可以指定给字段加多长的索引值，比如：

```
mysql> alter table suser add index index2(email(6));
```

如果没有定义好前缀索引，会增加回表次数，增加扫描行

使用前缀索引，定义好长度，就可以做到既节省空间，又不用增加太多的查询成本。

如何确定要使用多长的前缀索引？

再确认索引时，要关注的区分度，区分度越高，重复键值越少，可以通过判断索引上有多少个不同的值来判断要使用多长的前缀。

(一) 倒序存储

比如存储身份证时，可以把身份证号倒过来存储。后6位没有向开头那种地址码的重复逻辑。

```
mysql> select field_list from t where id_card =  
reverse('input_id_card_string');
```

使用reverse 函数

(二) hash字段

使用crc32()得到校验码填到这个新字段，校验码可能得到冲突。

倒序存储和hash字段的异同点？

占用空间：hash 字段存储占用空间大，因为要增加一个字段存储crc32()函数；倒叙存储方式在主键索引上，不消耗存储空间。

CPU消耗：倒序方式需要额外调用一次reverse()方法，hash字段方式需要调用一次crc32()。按照这两个比较来说，reverse 消耗的CPU资源要少一些。

查询效率：使用hash 字段的查询效率要稳定一些，查询效率维持在 O (1) 级别，倒序存储的方式使用前缀索引，有可能增加扫描的行数。

总结

- 创建完整的索引比较占用空间
- 创建前缀索引，节省空间，但是会增加扫描的行数
- 倒序存储，在创建前缀索引，用于绕过字符串前缀的区分度高的场景
- hash字段 索引，查询稳定，有额外的存储和计算消耗，不支持范围查询。

索引下推

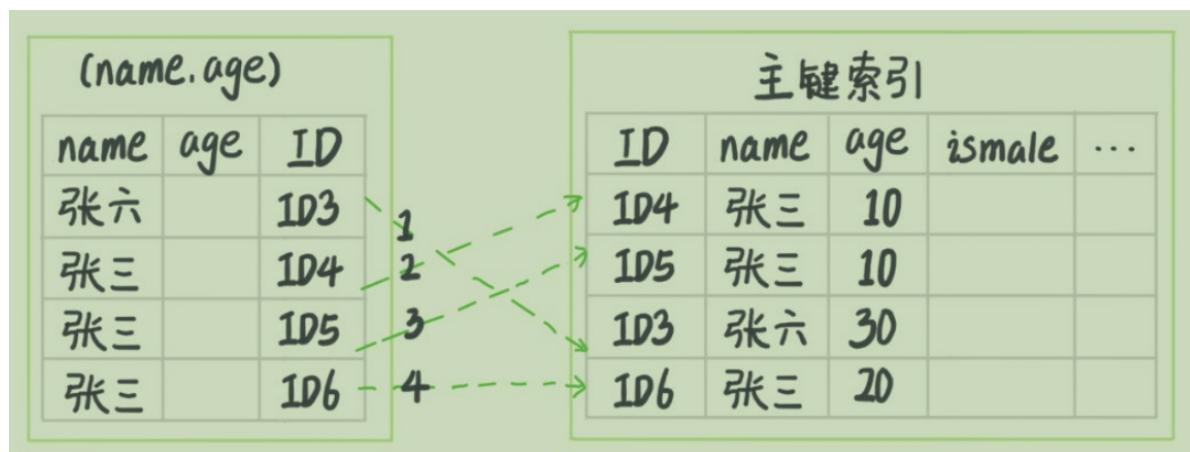
在MySQL5.6之后，引入了索引下推优化，可以在遍历索引的过程中，对索引包含的字段先判断，直接过滤掉不满足条件的记录，减少回表次数。

假如我们有一个用户表，并且使用用户的name, age两个字段建立联合索引，name在没有索引下推的功能，执行下面的sql，执行的流程如下图所示：

```
select * from tuser where name like '张%' and age=10  
and ismale=1;
```

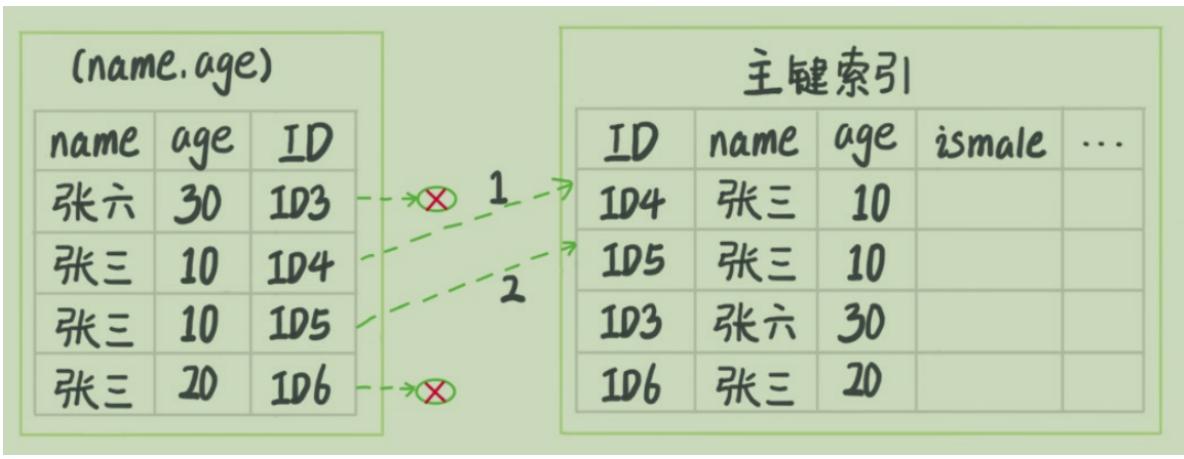
```
//添加name、age联合索引  
create index idx_name_age on tuser(name,age);
```

无索引下推



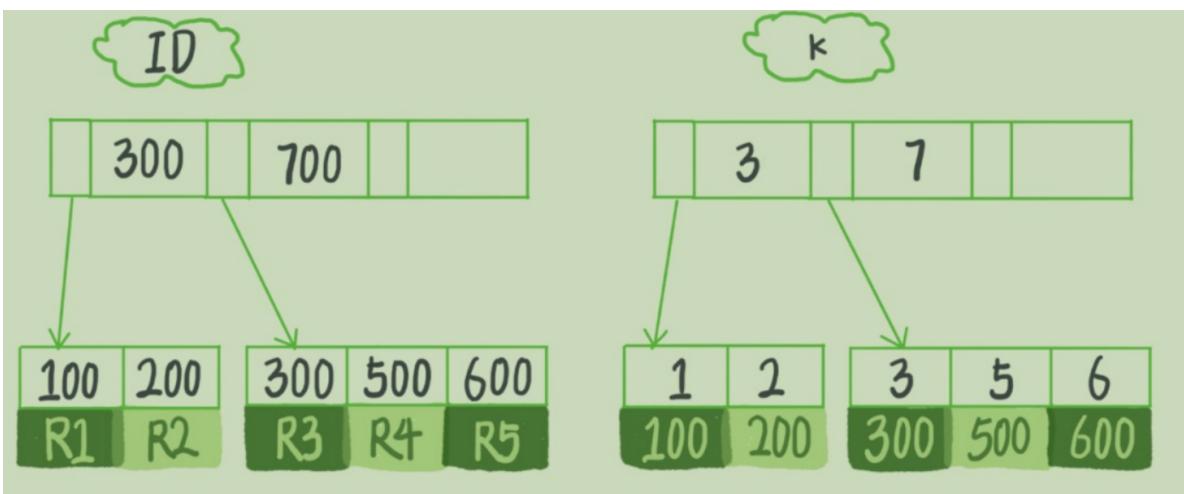
当比较第一个索引字段like '张%' 就会筛选出4行数据，后面不会根据age进行比较，而是直接获取主键值，然后回表查询，回表后在比较age、ismale 是否符合要求。

有索引下推



使用索引下推会根据你的age 再次进行比较，发现有不符合要求的直接过滤掉，符合条件的才会进行回表查询，这样就减少了不必要的回表。

主键使用自增ID还是UUID? 能说明原因么?



主键尽量使用自增ID，使用自增ID的话，新增一个数据会放在 R5 的记录的最后面插入一条记录。

如果添加是无序的 B+树 需要在逻辑上挪动后面的数据，空出一个位置。如果此时 R5 所在的记录已经满了，根据B+ 树算法，会申请一个新的数据页，然后挪动部分数据过去，这个过程称为**页分裂**，性能会受很大影响。

另一方面是存储空间，UUID需要16个字节的大小，会占用更多的存储空间。

什么场景使用非主键作为索引呢？

- 只有一个索引
- 该索引必须是唯一索引，就是典型的KV场景

如何重建索引 ❤

```
alter table t engine = InnoDB
```

为什么要重建索引？

索引可能会删除，或者页分裂的原因，导致数据页有空洞，重建索引的过程会创建一些新的索引，把数据按顺序插入，这样页的利用率最高，也就是索引更紧凑，更省空间。

MySQL选错索引

force

force index(a) 让优化器强制使用此索引

analyze table t 命令

- 发现 explain 的结果预估的 rows 值跟实际情况差距比较大，**用来重新统计索引信息。**

扫描行数是影响执行代价的因素之一（临时表、是否排序）。扫描的行数越少，磁盘的访问次数越少，消耗的CPU越少。

ChangeBuffer

普通索引和唯一索引的区别？

查找方面

- 对于普通索引来说，需要继续向后查找，直到碰到第一个不满足条件的记录停止检索

- 对于唯一索引，由于索引具有唯一性，查找到记录就会停止检索

性能差距微乎其微

更新方面

- 对于唯一索引来说，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。必须先将其读入到内存中才能判断，如果读入到内存就没有必要使用change buffer 了。唯一索引跟新没有必要使用change buffer。

要更新的目标页在内存中，唯一索引和普通索引的性能差距微乎其微。只是唯一索引多了一个判断是否唯一性。

不在内存中，对于唯一索引来说，需要将数据页读入到内存中，判断有没有冲突，然后插入这个值。对于普通索引来说，则是将更新记录在change buffer 中，语句就执行结束了。

数据从磁盘读入到内存设计IO的访问，时间成本高。change buffer因为减少了磁盘的随机访问，所以对更新性能有所提高。

change buffer

- 如果数据页在内存中，直接更新。没有在内存中，在不影响数据一致性的情况下，InnoDB引擎将这些更新到ChangeBuffer 中，这样就可以减少磁盘的访问次数。在下次查询的时候在change buffer 中进行读取。
- 适合于写多读少的业务场景：账单类、日志类
- 在写入后立即做查询，反而change buffer 起到了副作用。

change buffer 和redo log 的区别？

- redo log 减少了随机写磁盘的消耗（转成顺序写）。
- change buffer 减少了随机读磁盘的IO消耗。

merge

- 将change buffer 的操作应用到原数据页，得到的最新结果称为 merge。

merge的执行流程

1. 从磁盘读入数据页到内存
2. 从change buffer 里找出这个数据页的change buffer 记录，依次应用，得到新的数据页。
3. 写redo log。这个redo log 包含了数据的变更和change buffer 的改变。

将更新操作记录到 change buffer 中，减少读磁盘，执行的语句会得到明显的提升。

问题

如果某次写入使用了 change buffer 机制，之后主机异常重启，是否会丢失 change buffer 和数据。

不会丢失，在事务提交的时候，已经把change buffer 的记录到redo log 中，所以在崩溃恢复过程中，change buffer 也能找回来。

锁

全局锁

Flush tables with read lock (**FTWRL**)。让整个数据库处于只读的状态。

使用场景

- 全库逻辑备份，将整个库每个表都select 出来存成文本。

使用官方自带的逻辑备份工具mysqldump。导数据之前启动一个事务，来确保拿到一个一致性视图。由于MVCC的支持，这个过程中事务是可以正常更新。

为什么要使用FTWRL?

- 因为MyISAM 引擎不支持事务，如果备份中更新，总是能取到最新的数据，保证不了数据的一致性。

为什么不使用 set global readonly = true 的方式呢？

- 在有一些系统中。readonly的值用来做逻辑判断，比如用于判断一个库是主库还是从库。
- 使用FTWRL命令 如果客户端发生异常断开，那么MySQL 会释放这个全局锁，整个库回到可以正常跟新的状态。而 设置readonly 后，客户端发生异常，则数据库会一直保持 readonly 状态，这样会导致整个库长时间处于不可写状态，风险高。

表级锁

一共有两种：表锁、元数据锁（meta data lock）MDL

表锁

语法是 lock tables . . . read/write , unlock tables 主动释放锁，也可以在客户端断开时自动释放

lock tables语法除了会限制别的线程读写外，也限定了本线程接下来的操作对象。

举个例子，如果在某个线程 A 中执行 lock tables t1 read, t2 write; 这个语句，则其他线程写 t1、读写 t2 的语句都会被阻塞。同时，线程 A 在执行 unlock tables 之前，也只能执行读 t1、读写 t2 的操作。连写 t1 都不允许，自然也不能访问其他表。

MDL

- 不需要显示使用，在访问一个表的时候被自动加上。保证读写的正确性。

| 在MySQL 5.5 版本中引入了MDL，当对一个表执行增删改查操作时，加MDL读锁。

| 当对表结构变更操作的时候加MDL写锁。

读锁之间不互斥，多线程同时对一张表增删改查

读写锁互斥、写锁互斥，用来保证变更表结构的安全性。因此如果有两个线程同时给一个表加字段，其中一个要等到另一个执行完才能开始执行。

| 例如如下测试。MySQL 5.6

session A	session B	session C	session D
begin; select * from t limit 1;			
	select * from t limit 1;		
		alter table t add f int; (blocked)	
			select * from t limit 1; (blocked)

我们可以看到 session A 先启动，这时候会对表 t **加一个 MDL 读锁**。
由于 session B 需要的也是MDL 读锁，**因此可以正常执行**。

之后 **session C 会被 blocked**，是因为 **session A 的 MDL 读锁还没有释放**，而 session C 需要 MDL 写锁，因此只能被阻塞。

如果只有 session C 自己被阻塞没什么关系，但是之后所有要在表 t 上新申请 MDL 读锁的请求也会被 session C 阻塞。前面我们说了，所有对表的增删改查操作都需要先申请 MDL 读锁，就都被锁住，等于这个表现在完全不可读写了。

④ 事务中的MDL 锁，在语句执行开始时申请，但是语句结束后不会马上释放，而是等到整个事务提交之后在释放。

如何安全地给小表加字段？

给一个表加字段，或者修改字段，或者加索引，需要扫描全表的数据。

在alter table 语句里面设定等待时间，如果在这个等待时间里面能够拿到MDL 写锁最好，拿不到也不要阻塞后面的业务语句，先放弃。之后开发人员或者DBA在通过重试命令重复这个过程。

行锁

两阶段协议

- 在InnoDB 事务中，行锁是在需要的时候被加上的，但并不是不需要就立即释放，而是等到事务结束时才释放。这就是两阶段协议。

如果你的事务中需要锁住多个行，要把最可能造成锁冲突、最可能影响并发度的锁尽量往后放。

举个例子：假设现在你负责一个电影票交易业务，顾客A在电影院B购票。涉及到以下操作

1. 从顾客A账户余额中扣除电影票钱
2. 给电影院B的账户余额增加这张电影票的票价
3. 记录一条交易记录。

在这三个事务中，最容易发生事务冲突的是业务2，如果同时有另一个顾客C要在影院B买票。因为他们要更新同一个影院账户的余额，需要修改同一行数据。

根据两段锁协议，不管你怎样安排语句顺序，所有的操作需要的行锁都是在事务提交的时候才释放的。所以如果你把语句2安排到最后，这样影院账户余额这一行的锁时间就最少。这样就很大程度地减少了事务之间的锁等待，提升了并发度。

为什么服务器CPU接近100%?

主要原因是死锁和死锁检测。

死锁

当并发系统中**不同线程出现循环资源依赖**, 涉及的线程都在等待别的线程释放资源时, 就会导致这几个线程都进入无限等待的状态。

解决死锁的策略

- 直接进入等待, 直到超时。超时时间可以通过设置 `innodb_lock_wait_timeout` 来设置。
- 发起死锁检测, 发现死锁后, 主动回滚死锁链条中的某一个事务, 让其他事务继续执行。设置参数 `innodb_deadlock_detect` 为 `on`, 表示开启逻辑。

第一种策略: InnoDB中 `innodb_lock_wait_timeout` 默认为50s, 意味着当出现死锁时, 需要等待50s才会超时退出。对于大部分在线服务是无法接受的。但是又不能设置一个很小的值, **因为不能判断是死锁还是锁等待?**

第二种策略:  主动死锁检测, InnoDB引擎中默认开启 `innodb_deadlock_detect` 为 `on`, 但是检测死锁的过程中**大量消耗CPU**。每个新来的被堵住的线程, 都要判断会不会由于自己的加入导致了死锁, 这是一个时间复杂度为 $O(n)$ 的操作。

如何解决主动死锁检测带来的性能问题?

1. 临时关闭死锁检测
2. 控制并发度。使用MQ这种中间件, 控制数据库中并发线程数。

行锁和表锁的区别?

行锁粒度小。行锁只锁定一行, 并发度高, 加锁开销大, 加锁慢, 会发生死锁

表锁粒度大, 不会发生死锁。并发度低, MyISAM 和InnoDB都支持表锁。

重建表

为什么表数据删掉一半，表文件大小不变？

InnoDB引擎包含两部分：

- 表结构定义和数据。

MySQL8.0 之前表结构是存在 .frm 后缀文件里

MySQL 8.0 之后，把表结构定义在系统数据表中，因为表结构定义很占空间。

innodb_file_per_table

控制表数据放在单独的文件或者共享表空间。

- OFF 表示的是，表的数据放在系统共享表空间，也就是跟数据字典放在一起；
- ON 表示的是，每个 InnoDB 表数据存储在一个以 .ibd 为后缀的文件中

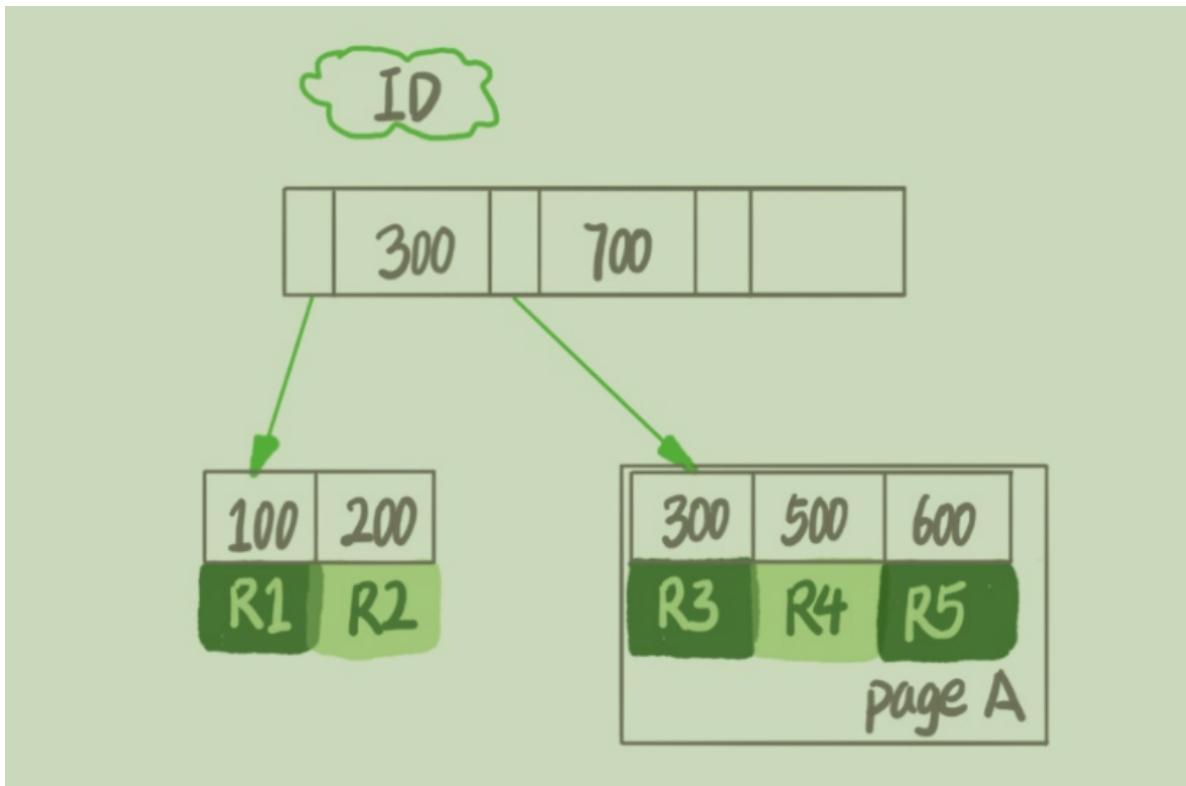
| MySQL 5.6.6 开始默认为ON

一个单独的文件便于管理，通过`drop table` 命令，系统就会直接删除这个文件。而放在共享表空间内，即使表删除掉了，空间也不会回收的。因为只是在删除的数据上做了个标记。

| 回收表空间 `drop table`

数据删除流程

B+树索引示意图



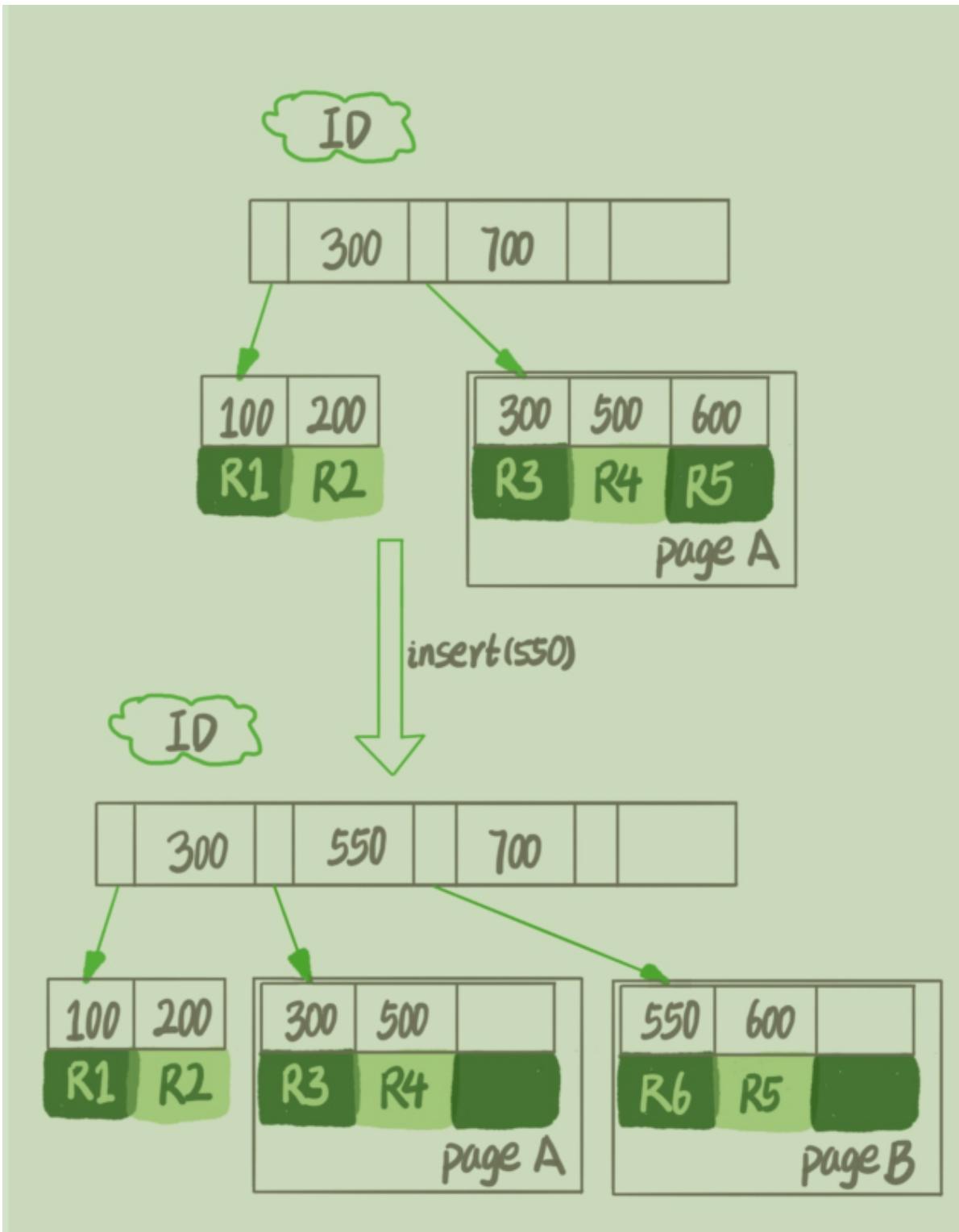
假如我们要删除R4这个数据，InnoDB引擎只会把R4这个记录标记为删除，如果之后在要插入一个ID在300 和 600 之间会复用这个位置。磁盘文件的大小并不会缩小。

InnoDB引擎是按页存储的。

delete

如果使用delete 命令把整个表的数据删除，所有的数据页都会标记为可复用，但是磁盘不会变小。

插入数据造成空洞



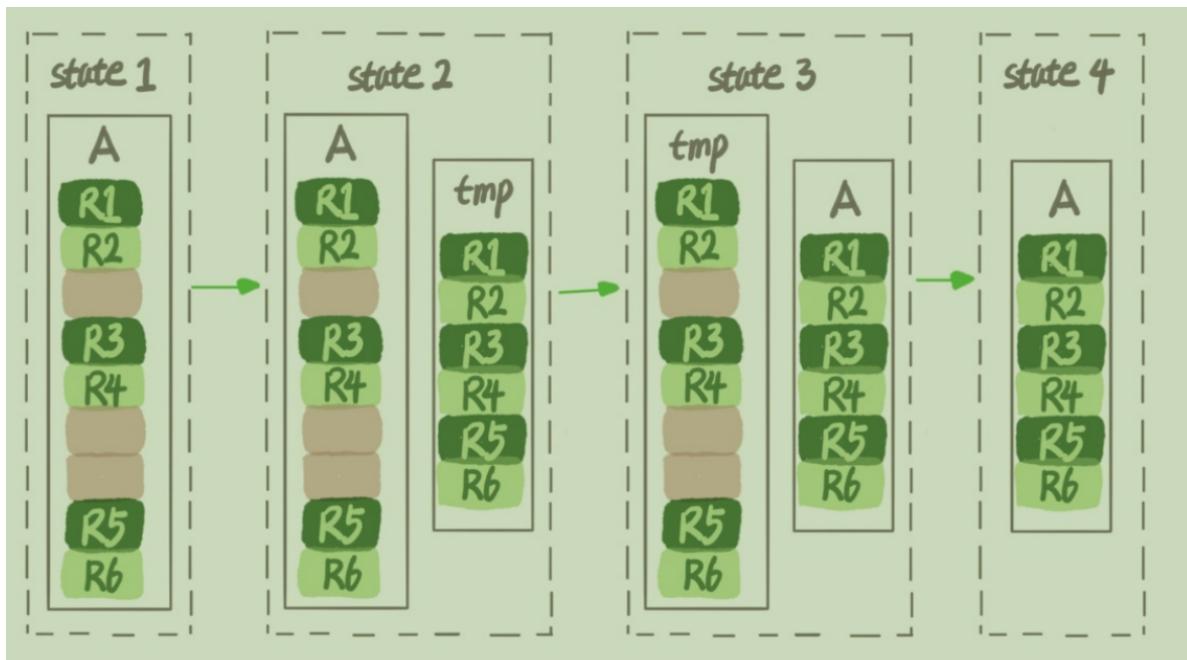
实际上不止一个空洞。

更新索引上的值，可以理解为删除一个旧值，在插入一个新值。这会造成空洞。

重建表

在MySQL 5.5 版本之前MySQL 会自动完成转存数据、交换表名、删除旧表的操作。

```
alter table A engine=InnoDB
```



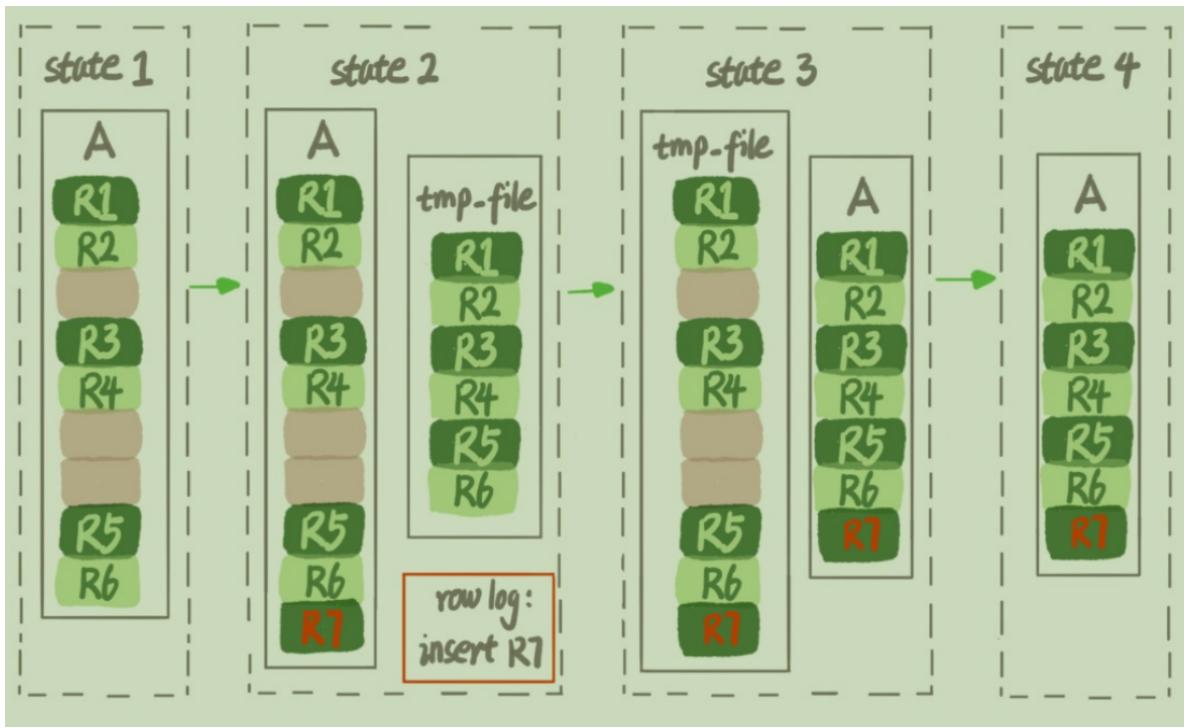
花费时间最长的是往临时表插入数据的过程。

在整个流程中是不能DDL的

MySQL 5.6 版本引入了Online DDL

流程

1. 建立一个临时文件，扫描A 主键的所有数据页
2. 用数据页中表A的记录生成B+树，存储到临时文件中
3. 生成临时文件的过程中，将所有对A的操作记录在一个row log 日志文件中，对应图state2 的状态。
4. 临时文件生成完成后，将日志文件中操作应用到临时文件，得到一个逻辑数据与表A相同的数据文件，对应state3的状态。
5. 用临时文件替换表A的数据文件。



optimize table、analyze table 和 alter table 这三种方式重建表的区别。

- 从MySQL 5.6 版本开始，`alter table t engine=InnoDB (recreate)` 默认就是上图4的流程了。
- `analyze table` 不是重建表，只是对表的索引信息做重新统计，没有修改数据，在这个过程加了MDL读锁
- `optimize table` 等于 `recreate+analyze`

count(*)为什么这么慢？

- MyISAM 把一个表的总行数存在磁盘上，效率很高
- InnoDB需要把数据一行一行的从引擎中读取出来，然后累计计数。

为什么InnoDB不把表的总行数存在磁盘上？

因为InnoDB存在MVCC 多版本，不能确定现在是哪个版本。

总结

- MyISAM表虽然`count (*)` 很快 但是不支持事务
- `show table status` 命令虽然返回很快，但是不准确
- InnoDB表会直接遍历全表，虽然很准确，但是效率低下

解决办法

自己计数，

(一) Redis

使用Redis 缓存，可能会导致丢失更新，当异常重启后从数据库中查询出来存入到Redis中。但是会存在数据不一致的情况。有以下两种情况

这两种情况都存在数据不一致的情况，因为先插入数据库，读Redis 计数时，计数没有+1

第二种情况是先Redis+1，但是数据还没有存储在数据库中，都会导致数据不一致

时刻	会话A	会话B
T1		
T2	插入一行数据R;	
T3		读Redis计数; 查询最近100条记录;
T4	Redis 计数加1;	

时刻	会话A	会话B
T1		
T2	Redis 计数加1;	
T3		读Redis计数; 查询最近100条记录;
T4	插入一行数据R;	
T5		

(二) 数据库

利用InnoDB支持事务的特性，把计数值单独放在一张表中。

时刻	会话A	会话B
T1		
T2	begin; 表C中计数值加1;	
T3		begin; 读表C计数值; 查询最近100条记录; commit;
T4	插入一行数据R commit;	

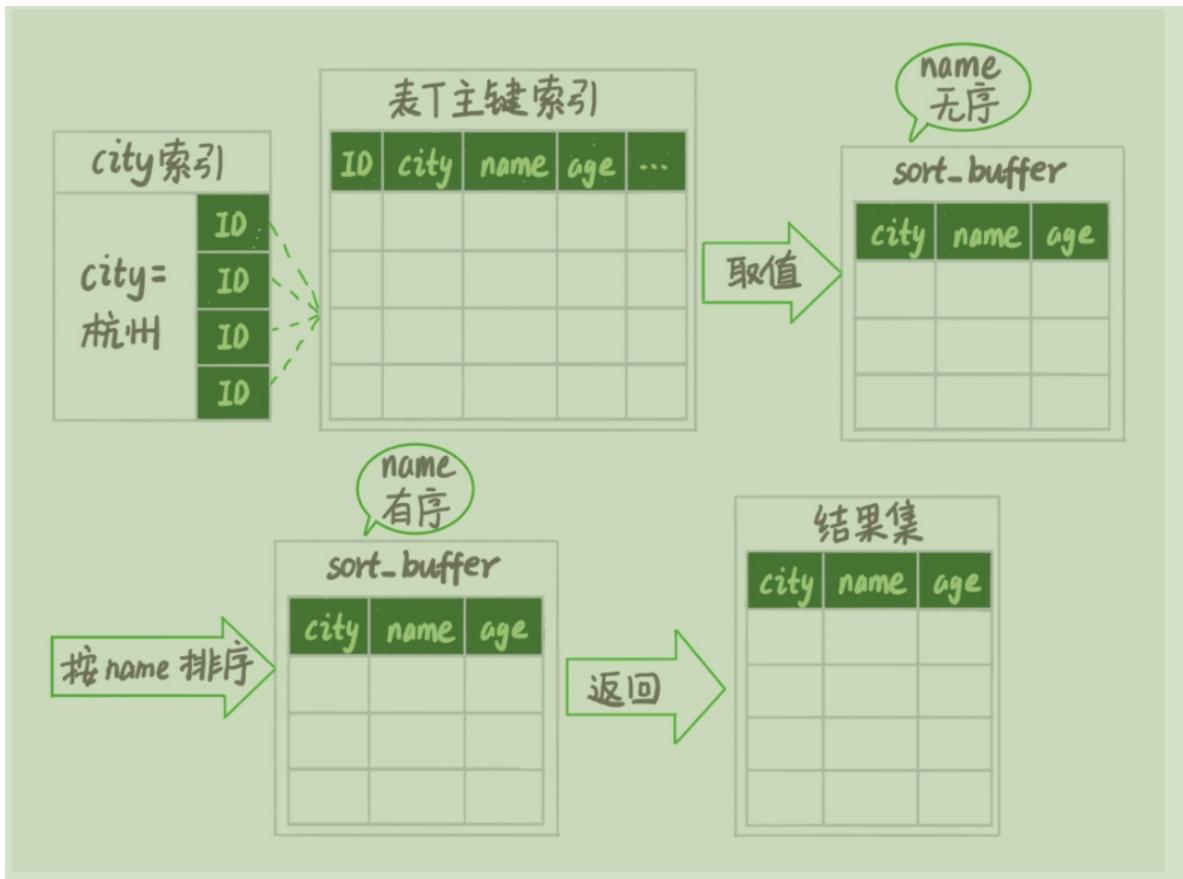
Order By 是怎么工作的?

全字段排序

```
mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE     | T      | NULL       | ref  | city          | city | 51    | const | 4000 |   100.00 | Using index condition; Using filesort |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Extra 字段下面 using filesort 使用了排序，称为sort_buffer

全排序流程



1. 初始化sort_buffer, 确定放入name、 city、 age 这三个字段
2. 从索引city 找到第一个满足city=' 杭州 ' 条件的主键 id , 也就是图中的 ID_X
3. 到主键 id 索引取出整行, 取 name 、 city 、 age 三个字段的值, 存入 sort_buffer 中;
4. 从索引 city 取下一个记录的主键 id ;
5. 重复步骤 3 、 4 直到 city 的值不满足查询条件为止, 对应的主键 id 也就是图中的 ID_Y ;
6. 对 sort_buffer 中的数据按照字段 name 做 快速排序;
7. 按照排序结果取前 1000 行返回给客户端。

按照name 进行排序, 可能在内存中完成, 也可能需要使用外部排序, 这取决于排序所需的内存和参数 sort_buffer_size.

sort_buffer_size

sort_buffer_size , 就是 MySQL 为排序开辟的内存 (sort_buffer) 的大小。

- 如果要排序的数据量小于 sort_buffer_size , 排序就在内存中完成。

- 但如果排序数据量太大，内存放不下，则不得不利用磁盘临时文件辅助排序。

外部排序使用归并排序算法。

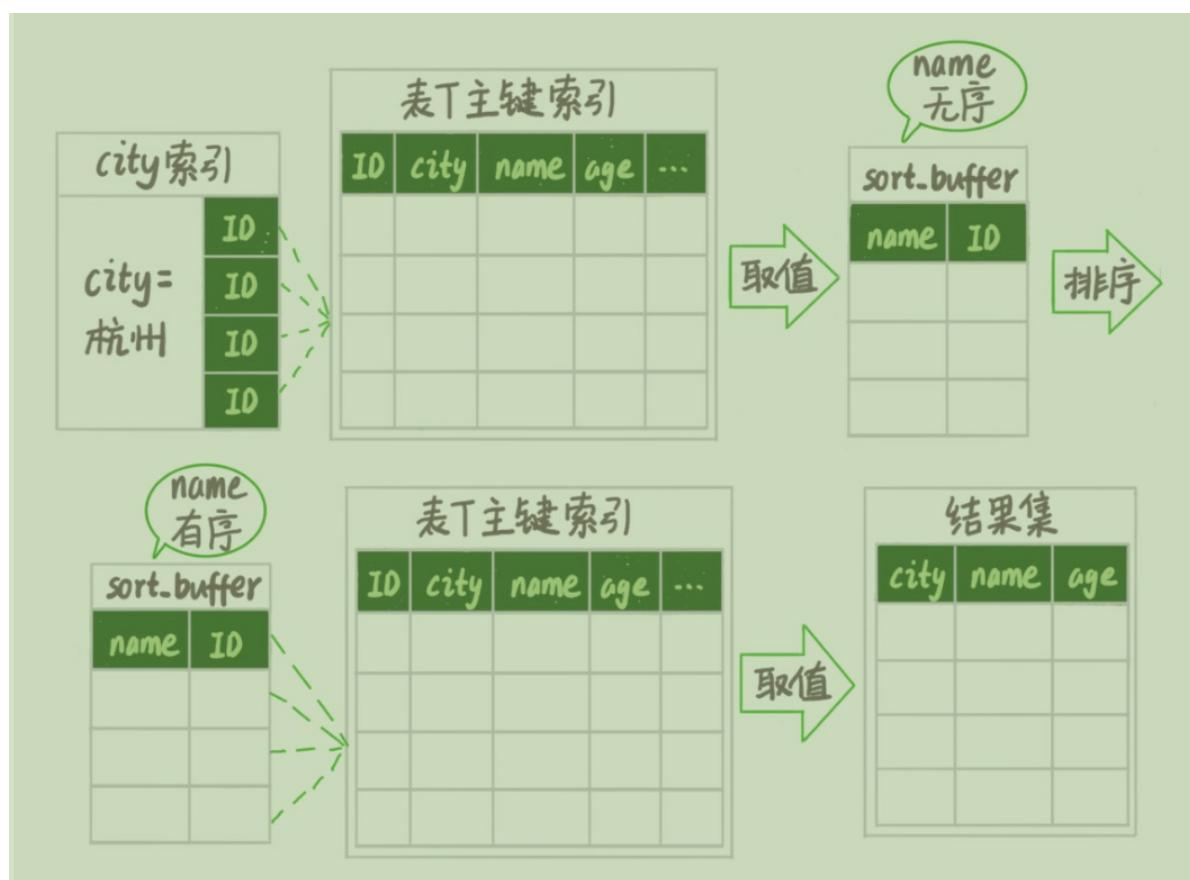
rowId排序

如果 sort_buffer 里面要放的字段数太多，这样内存里能够同时放下的行数很少，要分成很多个临时文件，**排序的性能会很差**。所以如果单行很大，这个方法效率不够好。

max_length_for_sort_data

max_length_for_sort_data，是 MySQL 中专门控制用于排序的行数据的长度的一个参数。它的意思是，如果单行的长度超过这个值，MySQL 就认为单行太大，要换一个算法。

row_id排序流程图



1. 初始化 sort_buffer，确定放入两个字段，即 name 和 id；
2. 从索引 city 找到第一个满足 city=' 杭州 ' 条件的主键 id，也就是图中的 ID_X；

3. 到主键 id 索引取出整行，取 name 、 id 这两个字段，存入 sort_buffer 中；
4. 从索引 city 取下一个记录的主键 id ；
5. 重复步骤 3 、 4 直到不满足 city=' 杭州 ' 条件为止，也就是图中的 ID_Y ；
6. 对 sort_buffer 中的数据按照字段 name 进行排序；
7. 遍历排序结果，取前 1000 行，并按照 id 的值回到原表中取出 city 、 name 和 age 三个字段返回给客户端。

rowid 排序多访问了一次表 t 的主键索引，就是步骤 7

全字段排序 VS rowid 排序

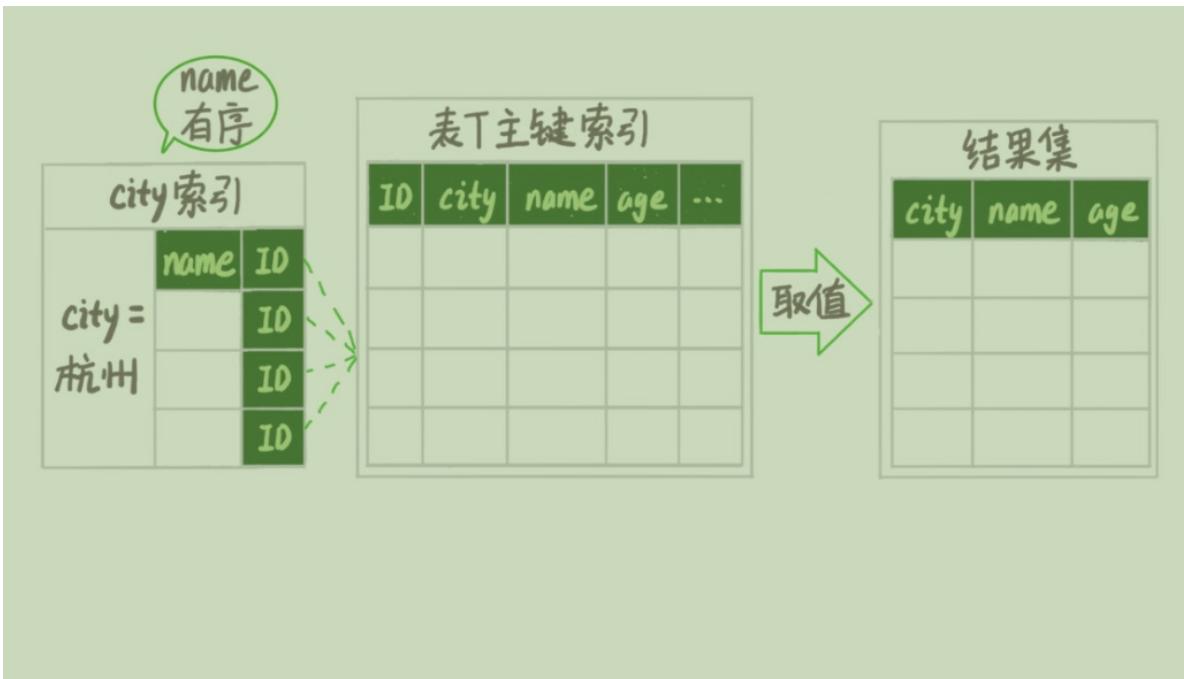
如果 MySQL 实在是担心排序内存太小，会影响排序效率，才会采用 rowid 排序算法，这样排序过程中一次可以排序更多行，但是需要再回到原表去取数据。

如果 MySQL 认为内存足够大，会优先选择全字段排序，把需要的字段都放到 sort_buffer 中，这样排序后就会直接从内存里面返回查询结果了，不用再回到原表去取数据。造成回表问题

如果内存够，就要多利用内存，尽量减少磁盘访问。

rowid 排序会要求回表多造成磁盘读，因此不会被优先选择。

更好的解决办法：使用覆盖索引



1. 从索引 (city,name) 找到第一个满足 city=' 杭州 ' 条件的主键 id ;
2. 到主键 id 索引取出整行，取 name 、 city 、 age 三个字段的值，作为结果集的一部分直接返回；
3. 从索引 (city,name) 取下一个记录主键 id ; alter table t add index city_user(city, name);
4. 重复步骤 2 、 3 ，直到查到第 1000 条记录，或者是不满足 city=' 杭州 ' 条件时循环结束。

```
mysql> explain select city, name,age from T where city='杭州' order by name limit 1000;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | partitions | type | possible_keys | key | key_len | ref | rows | filtered | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | T | NULL | ref | city,city_user | city_user | 51 | const | 4000 | 100.00 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
```

Mysql 实现功能

随机产生 3 个字段

```
select * from test order by rand() limit 3;
```

QA

1、当 MySQL 去更新一行，但是要修改的值跟原来的值是相同的，这时候 MySQL 会真的去执行一次修改吗？

会的，MySQL 会认真的执行更新语句。

可以用一致性读(快照读)来验证这个问题。每次事务执行都会创建一个快照，在这个快照中进行数据操作。

假设当前表的值为 (1, 2)

SessionA	SessionB
begin;select *from t where id = 1;	
	update t set a = 3 where id =1
update t set a = 3 where id =1;Query OK, 0 row addeected(0.00 sec) Rows matched: 1 Changed:0 Warnings:0	
select *from t where id = 1; //返回(1,3)	

Session A 的更新语句，返回OK，表示每次更新都会先老老实实的执行SQL 语句，该加锁加锁、该更新的更新。

2.

MySQL 是怎么控制并发的访问资源?

MySQL的死锁是怎么发生的? 怎么解决死锁问题?

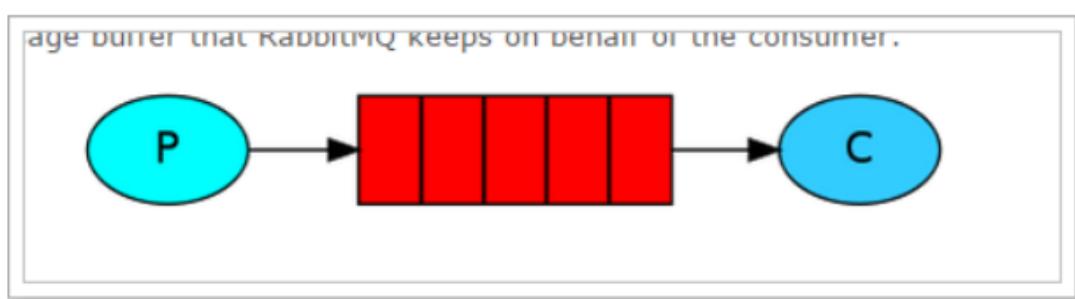
说一说MySQL的主从复制?

说说分库分表? 怎么分?

RabbitMq

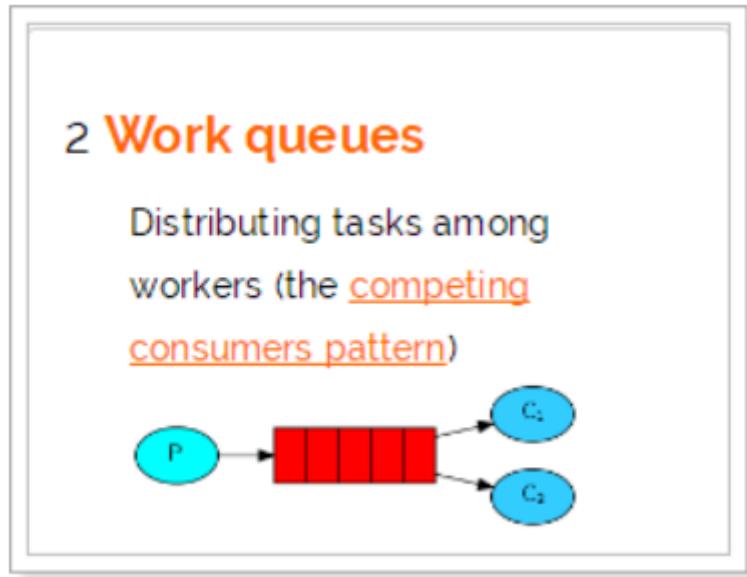
消息类型

基本消息模型



一个生产者一个消费者

work消息模型

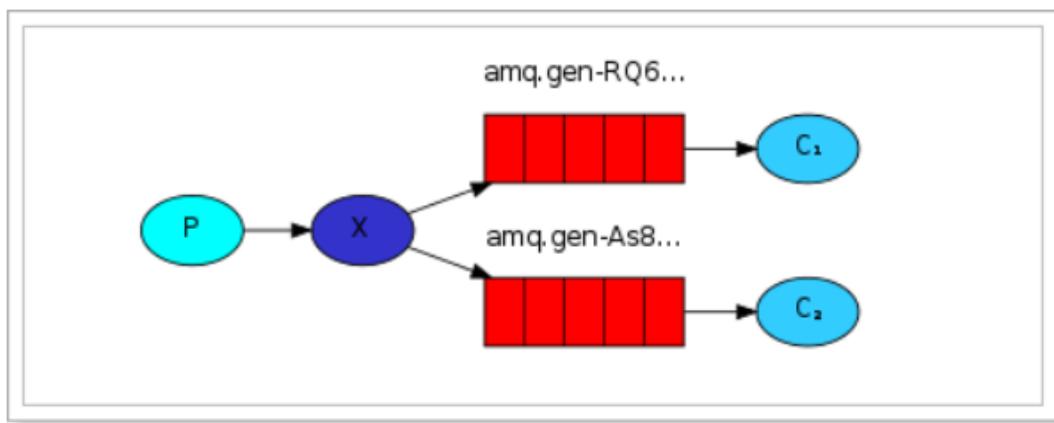


一个生产者两个消费者。

能者多劳

消费越快的人，消费的越多，通过配置basicQos方法prefetchCount = 1设置。

发送订阅模型



- 一个生产者，多个消费者
- 每个消费者都有自己的一个队列
- 生产者将消息发送到交换机（Exchange），绑定队列到交换机上
- 生产者发送消息，经过交换机到达队列，实现一个消息被多个消费者消费

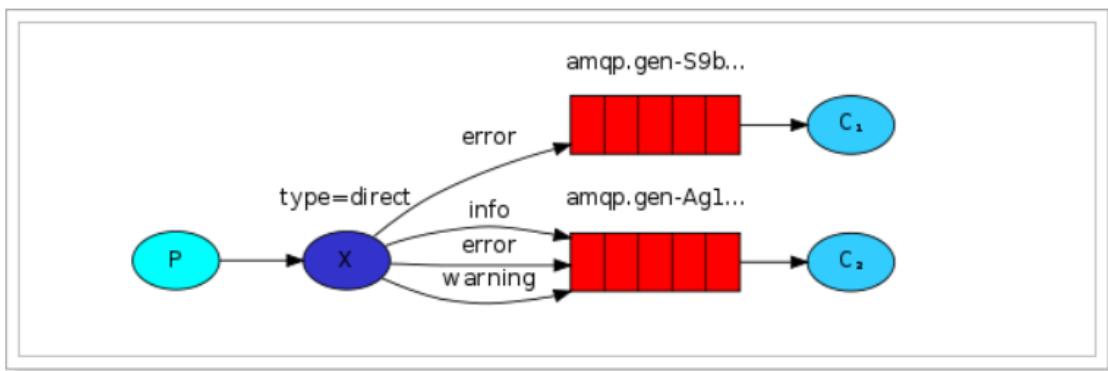
Exchange 类型有以下几种

- fanout 广播，将消息交给所有绑定的队列
- direct 定向，把消息交给符合要求的 routing key 队列
- Topic 通配符，把消息交给符合 routing pattern (路由模式) 队列

Fanout 广播

- 可以有多个消费者，每个消费者有自己的queue (队列)
- 每个队列绑定到交换机上
- 生产者发送消息到交换机，由交换机发布到已绑定上的队列
- 队列的消费者拿到消息进行消费，实现一条消息被多个消费者消费。

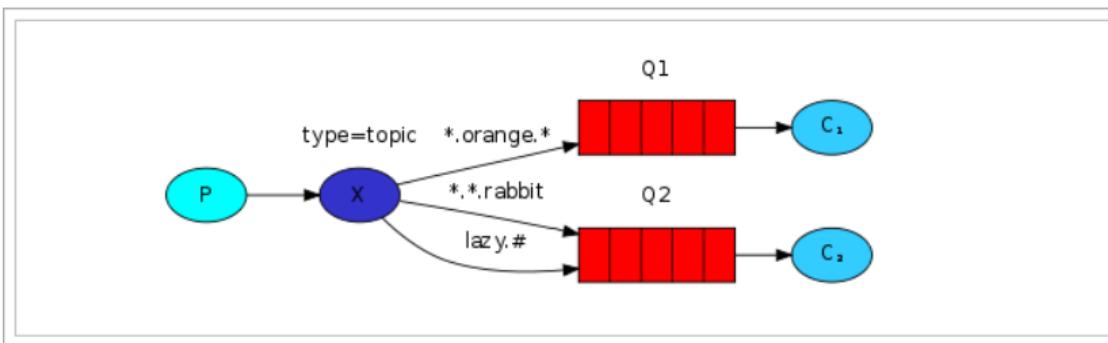
Direct 定向接受消息



Exchange 会指定一个routing key 进行绑定，消息的发送方向
Exchange 发送消息也必须指定消息的routing key 。

该模式可以把消息进行分类传输，比如重要的消息发送到执行的 routing key，次要的消息发送的另外一个routing key。

Topic 使用通配符选择消息队列



- '#' 匹配一个或多个词
- '*' 匹配一个词

消息确认机制

手动ACK

接受消息后，不会发送ACK，需要手动调用。

自送ACK

消息一旦接受，消费者自动发送ACK，如果消费者发生异常消息也会被消费。

如何确保数据的可靠性？

1. 事务机制，开启事务支持，如果消息投递失败，回滚事务。
2. 发送方确认机制，将信道设置成confirm 模式，消费者消费后，rabbit就会给生产者一个确认
3. 持久化交换机、队列、消息。
4. 手动确认消息，

保证消息不重复消费（幂等性）

- 根据Id去数据库查询，如果存在就不插入。不存在直接插入
- 利用redis 原子性机制实现，redis 是单线程的。
- 生产者每次发送的时候传输一个订单ID,消费者消费的时候先去redis 查询一下，是否被消费。

保证消息的顺序性

rabbitmq 拆分多个queue，每个queue一个consumer。然后这个 consumer内部用内存队列排队，然后分发给不同的worker来处理。

如何解决消息积压？

- 首先解决消费者那边的问题，确保恢复消费速度，然后将现有的消费者停掉
- 新建一个Topic，partition 是原来的10倍，临时建立好原先10倍或20倍的queue 数量
- 然后写一个临时分发数据的consumer 程序，这个程序部署上去消费积压的数据。
- 临时征用10倍的机器来部署 consumer，每一批consumer 消费一个临时的queue 数据
- 这种做法是临时将queue 和consumer 资源扩大10倍，以正常速度的10倍去消费数据
- 等消费完积压数据后，恢复原来的系统架构，重新用原先的 consumer 消费消息

Mybatis

mybatis接口的工作原理

Mapper接口是没有实现类的，且接口中的方法是不能重载的，每个接口的全限定类名+方法名可定位一个MapperStatement。Mybatis运行时会使用动态代理为Mapper 接口生成代理对象proxy，代理对象会拦截接口方法，转而执行 MapperStatement所代表的SQL，然后将SQL执行结果返回。

并发编程

基础知识

守护线程和用户线程的区别？

- 用户线程运行在前台，执行具体的任务。
- 守护线程运行在后台，守护前台线程。

如 main 方法就是一个前台线程，在启动的同时JVM在后台启动多个守护线程如垃圾回收器。

用户线程运行在前台，执行具体的任务，如main方法

当用户线程结束，JVM退出。守护线程并不会影响JVM的退出。

并行和并发的区别

并行指同一时刻多线程同时运行

并发指同一时间段内多线程交替运行

线程和进程

- **根本区别**：进程是操作系统资源调度的基本单位，而线程是处理器任务调度和执行的基本单位
- **资源开销**：因为进程之间是独立的，进程间切换会有较大的消耗；每个线程之间都有独立的运行栈和程序计数器（PC），线程之间切换的消耗小。
- **包含关系**：一个进程有多个线程，多个线程共同完成；线程是进程的一部分，所以也叫轻量级进程
- **内存分配**：同一进程的线程共享本进程的地址空间和资源。而进程之间的地址空间和资源是相互独立的。
- **影响关系**：一个进程崩溃后，在保护模式下不会对其他线程影响；但是一个线程崩溃后整个进程死掉。多进程要比多线程健壮。
- **执行进程** 每个独立的进程有程序的入口、顺序执行的序列、程序的出口。但是线程不能独立的运行，必须依存在应用程序中。

什么是上下文切换？

多线程编程中一般线程的数量大于CPU的数量，CPU在同一时刻只能执行一个线程。此时就需要利用时间片轮转依次来执行每个线程，线程切换的过程中会消耗一部分时间，如果过多的上下文切换会导致利用率降低。

总的来说：当前任务在执行完CPU时间片切换到另一个任务前先保存自己的状态，以便下次切换回来可以加载整个任务的状态。**任务从保存到加载的过程就是一次上下文切换。**

死锁、活锁、饥饿

死锁

- 两个或两个以上的进程在执行过程中，由于竞争资源造成阻塞。若无外力的作用下，他们将无法推进下去。

形成死锁的4个条件

- 互斥条件
- 请求与保持条件
- 不剥夺条件
- 循环等待条件。

活锁

- 比如两个线程间互相礼貌的推让，自己都不占用资源，最后两个线程都无法使用资源

饥饿

- 在非公平锁中会出现饥饿的现象，比如某一个线程长时间获取不到锁。

创建线程的4种方式

- 继承Thread
- 实现Callable
- 实现Runnable
- 使用 Executors 工具类创建线程池

start() 和run()的区别？

为什么我们调用 start() 方法时会执行 run() 方法，为什么我们不能直接调用 run() 方法？

- new 一个线程线程进入新建状态，调用start()方法，线程进入就绪状态，当分配到时间片就可以运行了。start()执行线程的准备工作，然后自动执行run()方法里的内容。

- 直接运行 run方法，会把run() 方法当成一个main 下面的一个普通的方法去执行，并不会在某个线程中去执行，所以他不是多线程工作。

总结

- 调用start 方法会让线程进入就绪状态，而调用run 方法只是一个普通的方法，还是在主线程中执行。

并发编程的3个重要特征

原子性 synchronized 保证了代码片段的原子性

可见性：对工作线程里的变量进行修改，另外的线程可以立即看到修改后最新值。volatile 可以保证共享变量的可见性

有序性：代码在执行过程的先后顺序，volatile 保证了指令的重排序。

synchronized

谈谈synchronized 的理解

synchronized 用于解决多线程并发的安全性，用于线程间的同步，在早期的版本中，锁的效率不高，依靠操作系统底层来实现的，用户态和内核态的切换需要消耗时间。在jdk6以后JVM对synchronized 进行了优化，加入了自旋锁、偏向锁、轻量级锁、锁粗化等技术来减少锁操作的开销。

synchronized关键字的三种使用方式

- 修饰方法
- 修饰静态方法
- 修饰代码块

synchronized 关键字加到 static 静态方法和 synchronized(Class) 代码块上都是给Class 类加锁。加到实例方法上是给实例对象加锁。

synchronized 和ReentrantLock的区别？

synchronized 是java自带的关键字，是非公平锁、加锁简单；使用 mark word 标记锁的状态，有一个锁升级优化的过程。都是可重入锁

ReentrantLock 是JUC下面的包，可以实现公平和非公平锁，需要手动加锁、解锁。否则会造成死锁。底层采用unsafe.park方法。都是可重入锁。使用更灵活，绑定多个condition。

volatile

volatile的关键是重排序和内存可见性。在java内存模型中分为工作内存和主内存，volatile 关键字保证了每次查询都是在主内存中查询，将工作内存中的数据强制刷新到主内存。在程序运行过程中编译器和处理器会对指令进行优化，代码的执行顺序未必是编写代码的顺序。volatile 关键字可以进制指令重排序，保证程序的正确执行。

ThreadLocal

每一个线程底层使用一个ThreadLocalMap 存储了很多ThreadLocal，用于存储每一个线程变量的副本。把数据的可见性范围限制在一个线程内，采用空间换时间的方式保证了并发安全。

存在内存泄漏问题

ThreadLocalMap 中使用的key 为弱引用，而value 是强引用。在垃圾回收时，key将被回收，value 不会被清理掉。这样key 就会为空，导致 value 永远不会被回收，此时会造成内存泄漏问题。在调用 set get remove 方法时，会清理掉 key 为null 的记录。使用完ThreadLocal 方法后，最好手动调用 remove 方法防止内存泄漏。

强软弱虚

强引用：new 出来的对象都是强引用

软引用：当内存不足时回收

若引用：发现及回收

虚引用：

线程池

线程池中 submit()和 execute()方法有什么区别？

- submit 用于接收返回值，可以用futureTask 方法接收，通过get()方法获取返回值，get()方法会阻塞线程直到任务完成
- execute 没有返回值，无法判断任务是否执行成功

Linux常用命令

查看cpu利用率

top

vmstat

```
[root@999 ~]# vmstat
procs      -----memory-----  -----swap-----  -----io-----  -----system-----  -----cpu-----
r b    swpd   free   buff   cache   si   so    bi    bo   in   cs us sy id wa st
0 0      0 823592 141300 634048    0    0     0     1   21   25  0  0 100  0  0
```

查询所有java进程的pid 命令

- JPS

显示内存的使用情况

- free

查询负载

uptime

查找命令

grep命令 命令是一种强大的文本搜索工具

格式： grep [option] pattern [file] 可使用 —help 查看更多参数。 使用实例：

```
ps -ef | grep sshd
```

查找指定 ssh 服务进程

```
ps -ef | grep sshd | grep -v grep
```

查找指定服务进程，排除 grep 本身

```
grep -n 'hello' a.txt
```

从文件中查找关键词，并显示行号

find命令

find 命令在目录结构中搜索文件，并对搜索结果执行指定的操作。 使用实例：
`find . -name "*.log" -ls` 在当前目录查找以.log 结尾的文件，并显示详细信息。
`find /root/ -perm 777` 查找/root/目录下权限为 777 的文件
`find . -size +100M` 查找当前目录大于 100M 的文件

Locate命令

locate 让使用者可以很快速的搜寻档案系统内是否有指定的档案。其方法是先建立一个包括系统内所有档案名称及路径的数据库。之后当寻找时就只需查询这个数据库（/var/lib/locatedb）。Linux 系统自动创建这个数据库， 默认每天自动更新一次，所以使用 locate 命令查不到最新变动过的文件。为了避免这种情况，可以在使用 locate 之前，先使用 updatedb 命令，手动更新数据库。 yum -y install mlocate 使用实例：
`locate /etc/sh` 搜索 etc 目录下所有以 sh 开头的文件
`locate pwd` 查找和 pwd 相关的所有文件

