

事务隔离级别（4种）

事务隔离级别指的是一个事务对数据的修改与另一个并行的事务的隔离程度，当多个事务同时访问相同数据时，如果没有采取必要的隔离机制，就可能发生以下问题：

Serializable(串行化)：一个事务在执行过程中完全看不到其他事物对数据库所做的更新（事务执行的时候不允许别的事务并发执行，事务只能一个接着一个地执行，而不能并发执行）

Repeatable Read (可重复读)：一个事务在执行过程中可以看到其它事务已经提交的新插入的记录，但是不能看到其它事务对已有记录的更新

Read Committed (读已提交数据)：一个事务在执行过程中可以看到其它事务已经提交的新插入的记录，而且能看到其它事务已经提交的对已有记录的更新。

Read Uncommitted (读未提交数据)：一个事务在执行过程中可以看到其它事务没有提交的新插入的记录的更新，而且能看其它事务没有提交到对已有记录的更新

数据库的四大特征：

1) 原子性 (Atomicity)：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚。

2) 一致性 (Consistency)：一个事务执行之前和执行之后都必须处于一致性状态。

3) 隔离性 (Isolation)：隔离性是当多个用户并发访问数据库时，比如操作同一张表时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。

4) 持久性 (Durability)：持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的。

事务的并发问题

1、脏读：事务 A 读取了事务 B 更新的数据，然后 B 回滚操作，那么 A 读取到的数据是脏数据

2、不可重复读：事务 A 多次读取同一数据，事务 B 在事务 A 多次读取的过程中，对数据作了更新并提交，导致事务 A 多次读取同一数据时，结果不一致。

3、幻读：系统管理员 A 将数据库中所有学生的成绩从具体分数改为 ABCDE 等级，但是系统管理员 B 就在这个时候插入了一条具体分数的记录，当系统管理员 A 改结束后发现还有一条记录没有改过来，就好像发生了幻觉一样，这就叫幻读。

小结：不可重复读的和幻读很容易混淆，不可重复读侧重于修改，幻读侧重于新增或删除。解决不可重复读的问题只需锁住满足条件的行，解决幻读需要锁表

MySQL 事务隔离级别

事务隔离级别	脏读	不可重复读	幻读
读未提交 (read-uncommitted)	是	是	是
读已提交 (read-committed)	否	是	是
可重复读 (repeatable-read)	否	否	是
串行化 (serializable)	否	否	否

第一类丢失更新、脏读、不可重复读、第二类丢失更新、幻读

第一类丢失更新

撤销一个事务的时候，把其它事务已提交的更新数据覆盖了。这是完全没有事务隔离级别造成的。如果事务被提交，另一个事务被撤销，那么会连同事务 1 所做的更

新也被撤销。

脏读 (Dirty Read)

如果一个事务对数据进行了更新，但事务还没有提交，另一个事务就可以“看到”该事务没有提交的更新结果。这样就造成的问题就是，如果第一个事务回滚，那么第二个事务在此之前所“看到”的数据就是一笔脏数据。

不可重复读取 (Non-Repeatable Read)

不可重复读取是指同一个事务在整个事务过程中对同一笔数据进行读取，每次读取结果都不同。如果事务 1 在事务 2 的更新操作之前读取一次数据，在事务 2 的更新操作之后再读取同一笔数据一次，两次结果是不同的。所以 Read Uncommitted 也无法避免不可重复读取的问题。

第二类丢失更新

它和不可重复读本质上是同一类并发问题，通常将它看成不可重复读的特例。当两个或多个事务查询相同的记录，然后各自基于查询的结果更新记录时会造成第二类丢失更新问题。**每个事务不知道其它事务的存在，最后一个事务对记录所做的更改将覆盖其它事务之前对该记录所做的更改。**

幻读 (Phantom Read)

幻读是指同样一个查询在整个事务过程中多次执行后，查询所得的结果集是不一样的。幻读针对的是多笔记录。在 Read Uncommitted 隔离级别下，不管事务 2 的插入操作是否提交，事务 1 在插入操作执行之前和之后执行相同的查询，取得的结果集是不同的，所以 Read Uncommitted 同样无法避免幻读。

无事务隔离级别存在：第一类丢失更新、脏读、不可重复读、第二类丢失更新和幻读问题。

Read Uncommitted 存在：脏读、不可重复读、第二类丢失更新和幻读问题。

Read committed 存在： 不可重复读、第二类丢失更新和幻读问题。

Repeatable Read 存在： 幻读问题。

Serializable : 不存在问题。

MySQL 存储引擎 (4 种)

MyISAM

- ◆ 索引只有一种
 - 被索引字段值作为索引数据，叶子节点还包含了该记录数据页地址
- ◆ 不支持事务，没有 undo log 和 redo log
- ◆ 仅支持表锁
- ◆ 不支持外键
- ◆ 会保存表的总行数

InnoDB

- ◆ 索引分为聚簇索引与二级索引
 1. 聚簇索引：主键值作为索引数据，叶子节点还包含了所有字段数据
 2. 二级索引：被索引的字段值作为索引数据，叶子节点还包含了主键值
- ◆ 支持事务
 3. 通过 undo log 支持事务回滚、当前读（多版本查询）
 4. 通过 redo log 实现持久性
 5. 通过两阶段提交实现一致性

6. 通过当前读、锁实现隔离性

- ◆ 支持行锁、间隙锁
- ◆ 支持外键

MEMORY

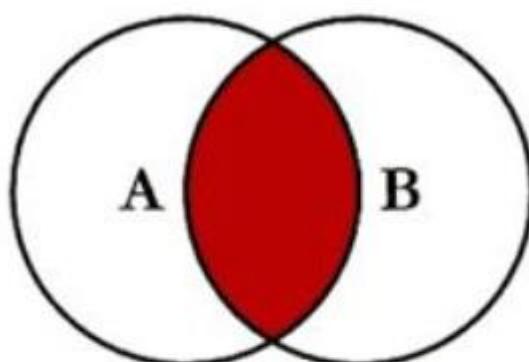
memory 使用存在内存中的内容来创建表。每个 MEMORY 表实际对应一个磁盘文件，格式是.frm。MEMORY 类型的表访问非常快，因为它到数据是放在内存中的，并且默认使用 HASH 索引，但是一旦服务器关闭，表中的数据就会丢失，但表还会继续存在。

MERGE

merge 存储引擎是一组 MyISAM 表的组合，这些 MyISAM 表结构必须完全相同，MERGE 表中并没有数据，对 MERGE 类型的表可以进行查询、更新、删除的操作，这些操作实际上是对内部的 MyISAM 表进行操作。

Mysql 七种连接

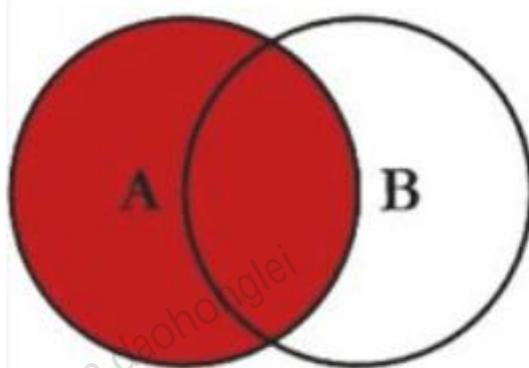
1. 内连接



```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
858883985@qq.com daohonglei  
SELECT * FROM tbl_dept a INNER JOIN tbl_emp b on a.id = b.deptId
```

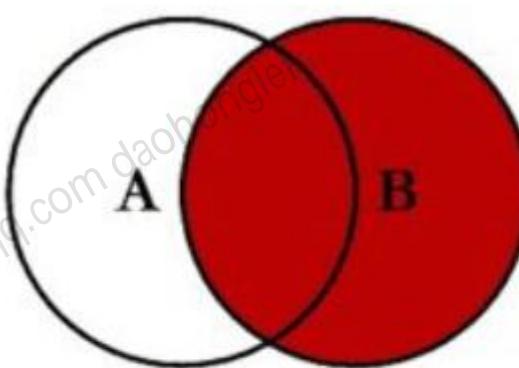
2. 左连接



```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```

```
858883985@qq.com daohonglei  
SELECT * FROM tbl_dept a LEFT JOIN tbl_emp b on a.id = b.deptId
```

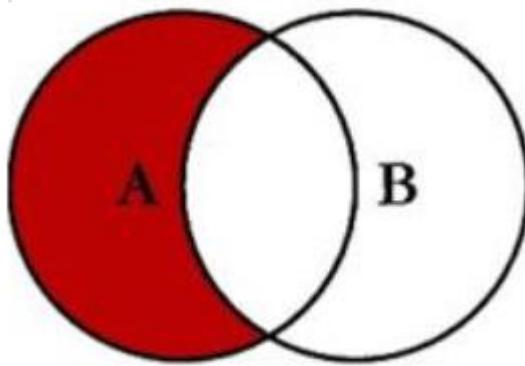
3. 右连接



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```

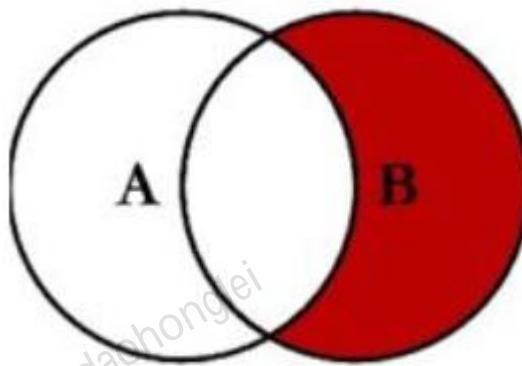
```
858883985@qq.com daohonglei  
SELECT * FROM tbl_dept a RIGHT JOIN tbl_emp b on a.id = b.deptId
```

4. 左外连接



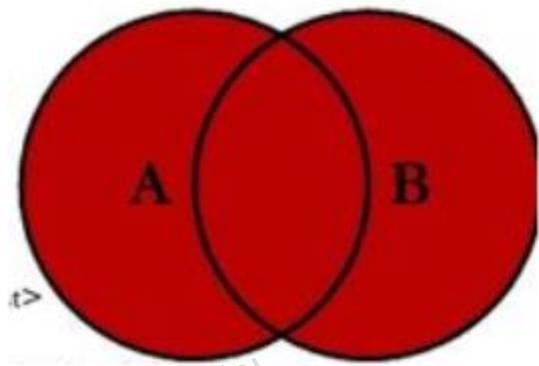
```
SELECT * FROM tbl_dept a LEFT JOIN tbl_emp b ON a.id = b.deptId WHERE  
b.deptId IS NULL
```

5. 右外连接



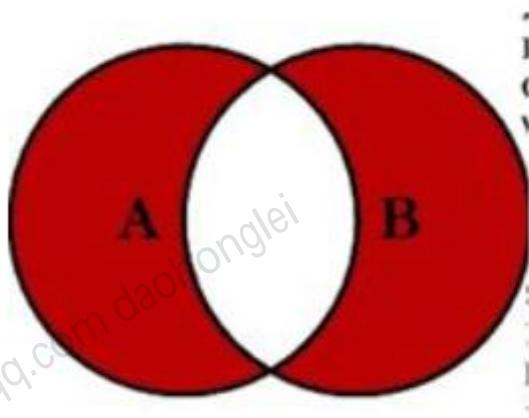
```
SELECT * FROM tbl_dept a RIGHT JOIN tbl_emp b ON a.id = b.deptId WHERE a.id  
IS NULL
```

6. 全连接



```
SELECT* FROMtbl_dept aLEFT JOIN tbl_emp b ON a.id = b.deptId UNION SELECT*  
FROMtbl_dept aRIGHT JOIN tbl_emp b ON a.id = b.deptId
```

7. 全外连接

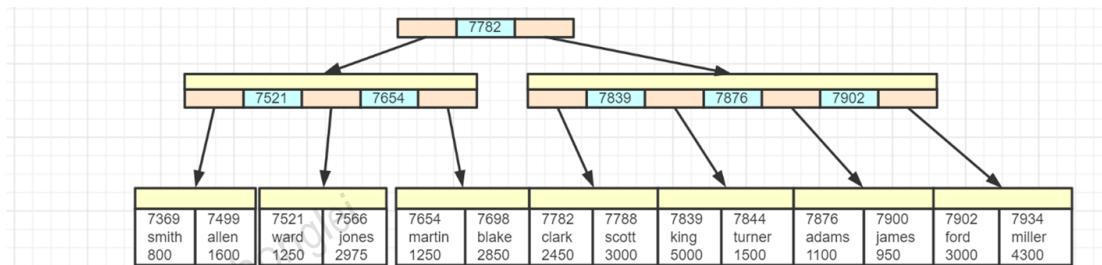


```
SELECT* FROMtbl_dept aLEFT JOIN tbl_emp b ON a.id = b.deptId WHERE  
b.deptId is NULL UNION SELECT* FROMtbl_dept aRIGHT JOIN tbl_emp b ON a.id  
= b.deptId WHERE a.id is NULL
```

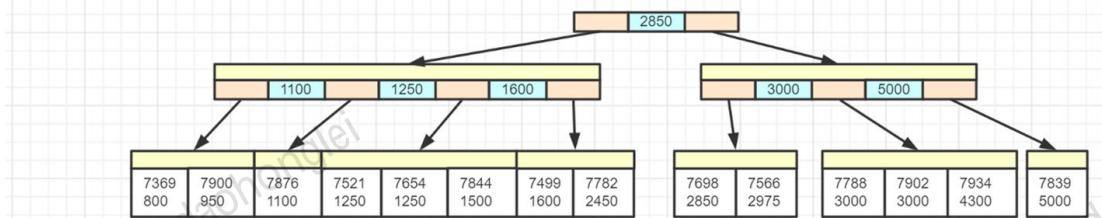
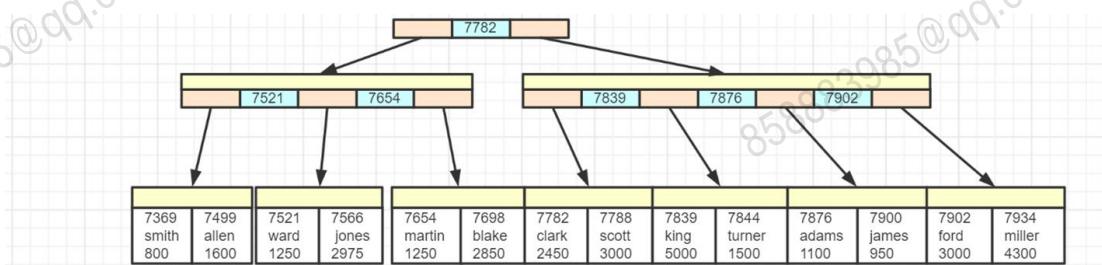
不同存储引擎索引区别

InnoDB

- 聚簇索引：主键值作为索引数据，叶子节点还包含了所有字段数据

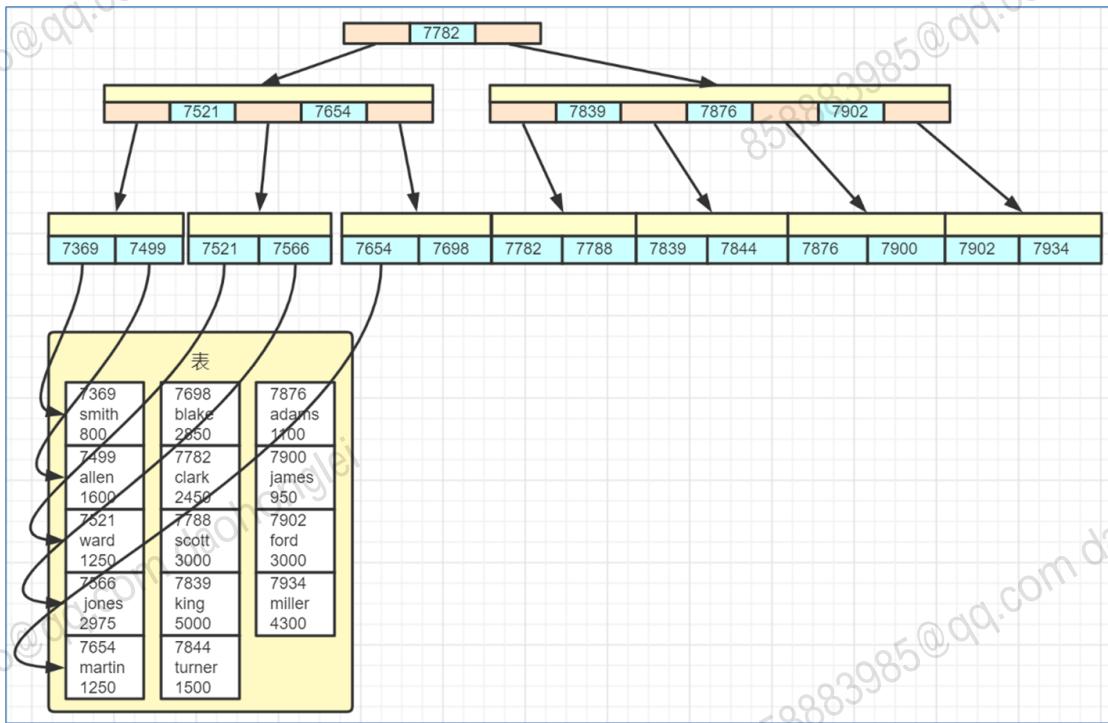


- 二级索引：被索引的字段值作为索引数据，叶子节点还包含了主键值



MyISAM

- 被索引字段值作为索引数据，叶子节点还包含了该记录数据页地址



为什么 mysql 的索引使用 B+树而不是 B 树呢

1. B+树更适合外部存储(一般指磁盘存储),由于内节点(非叶子节点)不存储 data,所以一个节点可以存储更多的内节点, 每个节点能索引的范围更大更精确。也就是说使用 B+树单次磁盘 IO 的信息量相比较 B 树更大, IO 效率更高。
2. mysql 是关系型数据库, 经常会按照区间来访问某个索引列, B+树的叶子节点间按顺序建立了链指针, 加强了区间访问性, 所以 B+树对索引列上的区间范围查询很友好。而 B 树每个节点的 key 和 data 在一起, 无法进行区间查找。

为什么 MySQL 采用 B+ 树索引

哈希索引

- 理想时间复杂度为 O(1)

- 适用场景：适用于等值查询的场景，内存数据的索引
- 典型实现：Redis, MySQL 的 memory 引擎

平衡二叉树索引

- 查询和更新的时间复杂度都是 $O(\log(n))$
- 适用场景：内存数据的索引，但不适合磁盘数据的索引，可以认为树的高度决定了磁盘 I/O 的次数，百万数据树高约为 20

BTree 索引

- BTree 其实就是 n 叉树，分叉多意味着节点中的孩子（key）多，树高自然就降低了
- 分叉数由页大小和行（包括 key 与 value）大小决定
 - 假设页大小为 16k，每行 40 个字节，那么分叉数就为 $16k / 40 \approx 410$
 - 而分叉为 410，则百万数据树高约为 3，仅 3 次 I/O 就能找到所需数据
- 局部性原理：每次 I/O 按页为单位读取数据，把多个 key 相邻的行放在同一页中（每页就是树上一个节点），能进一步减少 I/O

B+ 树索引

- 在 BTree 的基础上做了改进，索引上只存储 key，这样能进一步增加分叉数，假设 key 占 13 个字节，那么一页数据分叉数可以到 1260，树高可以进一步下降为 2

BTree key 及 value 在每个节点上，无论叶子还是非叶子节点，而 B+Tree 普通节点只存 key，叶子节点才存储 key 和 value

B+Tree 叶子节点用链表连接，可以方便范围查询及全表遍历

无论 BTree 还是 B+Tree，每个叶子节点到根节点距离都相同，B+Tree 必须到达叶子节点才能找到 value

索引、单列索引、复合索引、主键、唯一索引、聚簇索引、非聚簇索引、唯一聚簇索引 的区别与联系

索引

数据库只做两件事情：存储数据、检索数据。而索引是在你存储的数据之外，额外保存一些路标（一般是 B+树），以减少检索数据的时间。所以索引是主数据衍生的附加结构。

一张表可以建立任意多个索引，每个索引可以是任意多个字段的组合。索引可能会提高查询速度（如果查询时使用了索引），但一定会减慢写入速度，因为每次写入时都需要更新索引，所以索引只应该加在经常需要搜索的列上，不要加在写多读少的列上。

单列索引 与 复合索引

只包含一个字段的索引叫做单列索引，包含两个或以上字段的索引叫做复合索引（或组合索引）。建立复合索引时，字段的顺序极其重要。

下面这个 SQL 语句在 列 X, 列 Y, 列 Z 上建立了一个复合索引。

```
CREATE INDEX 索引名 ON 表名(列名 X, 列名 Y, 列名 Z);
```

其实这相当于建立了三个索引，分别是：1、单列索引（列 X） 2、复合索引（列 X, 列 Y） 3、复合索引（列 X, 列 Y, 列 Z）。

如何理解呢？

我们可以把多个字段组合的索引比喻成一个老式的纸质电话簿，三列分别是：

姓 - 名 - 电话号码

电话簿中的内容先按照姓氏的拼音排序，相同姓氏再按名字的拼音排序，这

相当于在（姓，名）上建立了一个复合索引。你可以通过这个索引快速找到所有具有特定姓氏的人的电话号码，也可以快速找到具有特定 姓-名 组合的人的电话号码。然而，想象一下，如果你想找到某个特定名字的人，其实这个索引是没有用的，你只能从头到尾遍历整个电话簿。

唯一索引与主键

唯一索引是在表上一个或者多个字段组合建立的索引，这个（或这几个）字段的值组合起来在表中不可以重复。一张表可以建立任意多个唯一索引，但一般只建立一个。

主键是一种特殊的唯一索引，区别在于，唯一索引列允许 null 值，而主键列不允许为 null 值。一张表最多建立一个主键，也可以不建立主键。

聚簇索引、非聚簇索引、主键

在《数据库原理》一书中是这么解释聚簇索引和非聚簇索引的区别：聚簇索引的叶子节点就是数据节点，而非聚簇索引的叶子节点仍然是索引节点，只不过有指向对应数据块的指针。

怎么理解呢？

聚簇索引的顺序，就是数据在硬盘上的物理顺序。一般情况下主键就是默认的聚簇索引。

一张表只允许存在一个聚簇索引，因为真实数据的物理顺序只能有一种。如果一张表上还没有聚簇索引，为它新创建聚簇索引时，就需要对已有数据重新进行排序，所以对表进行修改速度较慢是聚簇索引的缺点，对于经常更新的列不宜建立聚簇索引。

聚簇索引性能最好，因为一旦具有第一个索引值的记录被找到，具有连续索

引值的记录也一定物理地紧跟其后。一张表只能有一个聚簇索引，所以非常珍贵，必须慎重设置，一般要根据这个表最常用的 SQL 查询方式选择某个（或多个）字段作为聚簇索引（或复合聚簇索引）。

聚簇索引默认是主键，如果表中没有定义主键，InnoDB[1]会选择一个唯一的非空索引代替（“唯一的非空索引”是指列不能出现 null 值的唯一索引，跟主键性质一样）。如果没有这样的索引，InnoDB 会隐式地定义一个主键来作为聚簇索引。

聚簇索引与唯一索引

严格来说，聚簇索引不一定是唯一索引，聚簇索引的索引值并不要求是唯一的，唯一聚簇索引才是！在一个有聚簇索引的列上是可以插入两个或多个相同值的，这些相同值在硬盘上的物理排序与聚簇索引的排序相同，仅此而已。

MySQL Explain 详解

```
explain select e.no, e.name from emp e left join dept d on e.dept_no = d.no  
where e.name like 'Jef%' and d.name = '研发部';
```

	id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
	1	SIMPLE	d	ref	name loc	name loc	62	const	1	Using index condition
	1	SIMPLE	e	range	name index	name index	62	null	376	Using index condition; Using where; Using join b...

id	SELECT 识别符。这是 SELECT 的查询序列号 我的理解是 SQL 执行的顺序的标识，SQL 从大到小的执行 1. id 相同时，执行顺序由上至下 2. 如果是子查询，id 的序号会递增，id 值越大优先级越高，越先被执行 3. id 如果相同，可以认为是一组，从上往下顺序执行；在所有组中，id 值越大，优先级越高，越先执行
select_type	(1) SIMPLE(简单 SELECT，不使用 UNION 或子查询等) (2) PRIMARY(子查询中最外层查询，查询中若包含任何复杂的子部)

	<p>分, 最外层的 select 被标记为 PRIMARY)</p> <p>(3) UNION(UNION 中的第二个或后面的 SELECT 语句)</p> <p>(4) DEPENDENT UNION(UNION 中的第二个或后面的 SELECT 语句, 取决于外面的查询)</p> <p>(5) UNION RESULT(UNION 的结果, union 语句中第二个 select 开始后面所有 select)</p> <p>(6) SUBQUERY(子查询中的第一个 SELECT, 结果不依赖于外部查询)</p> <p>(7) DEPENDENT SUBQUERY(子查询中的第一个 SELECT, 依赖于外部查询)</p> <p>(8) DERIVED(派生表的 SELECT, FROM 子句的子查询)</p> <p>(9) UNCACHEABLE SUBQUERY(一个子查询的结果不能被缓存, 必须重新评估外链接的第一行)</p>
table	显示这一步所访问数据库中表名称（显示这一行的数据是关于哪张表的）, 有时不是真实的表名字, 可能是简称, 例如上面的 e, d, 也可能是第几步执行的结果的简称
type	<p>对表访问方式, 表示 MySQL 在表中找到所需行的方式, 又称“访问类型”。</p> <p>常用的类型有: ALL、index、range、ref、eq_ref、const、system、NULL (从左到右, 性能从差到好)</p> <p>ALL: Full Table Scan, MySQL 将遍历全表以找到匹配的行</p> <p>index: Full Index Scan, index 与 ALL 区别为 index 类型只遍历索引树</p> <p>range: 只检索给定范围的行, 使用一个索引来选择行</p> <p>ref: 表示上述表的连接匹配条件, 即哪些列或常量被用于查找索引列上的值, 用到了非唯一索引</p> <p>eq_ref: 类似 ref, 区别就在使用的索引是唯一索引, 对于每个索引键值, 表中只有一条记录匹配, 简单来说, 就是多表连接中使用 primary key 或者 unique key 作为关联条件</p> <p>const、system: 当 MySQL 对查询某部分进行优化, 并转换为一个常量时, 使用这些类型访问。如将主键置于 where 列表中, MySQL 就能将该查询转换为一个常量, system 是 const 类型的特例, 当查询的表只有一行的情况下, 使用 system</p> <p>NULL: MySQL 在优化过程中分解语句, 执行时甚至不用访问表或索引, 例如从一个索引列里选取最小值可以通过单独索引查找完成。</p>
possible_keys	指出 MySQL 能使用哪个索引在表中找到记录, 查询涉及到的字段上若存在索引, 则该索引将被列出, 但不一定被查询使用 (该查询可以利用

	<p>的索引，如果没有任何索引显示 null)</p> <p>该列完全独立于 EXPLAIN 输出所示的表的次序。这意味着在 possible_keys 中的某些键实际上不能按生成的表次序使用。</p> <p>如果该列是 NULL，则没有相关的索引。在这种情况下，可以通过检查 WHERE 子句看是否它引用某些列或适合索引的列来提高你的查询性能。如果是这样，创造一个适当的索引并且再次用 EXPLAIN 检查查询</p>
key	<p>key 列显示 MySQL 实际决定使用的键（索引），必然包含在 possible_keys 中</p> <p>如果没有选择索引，键是 NULL。要想强制 MySQL 使用或忽视 possible_keys 列中的索引，在查询中使用 FORCE INDEX、USE INDEX 或者 IGNORE INDEX。</p>
key_len	<p>表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度</p> <p>(key_len 显示的值为索引字段的最大可能长度，并非实际使用长度，即 key_len 是根据表定义计算而得，不是通过表内检索出的)</p> <p>不损失精确性的情况下，长度越短越好</p>
ref	<p>列与索引的比较，表示上述表的连接匹配条件，即哪些列或常量被用于查找索引列上的值</p>
rows	<p>估算出结果集行数，表示 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</p>
filtered	<p>显示了通过条件过滤出的行数的百分比估计值。</p>
Extra	<p>该列包含 MySQL 解决查询的详细信息,有以下几种情况:</p> <p>Using where: 不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示 mysql 服务器将在存储引擎检索行后再进行过滤</p> <p>Using temporary: 表示 MySQL 需要使用临时表来存储结果集，常见于排序和分组查询，常见 group by ; order by</p> <p>Using filesort: 当 Query 中包含 order by 操作，而且无法利用索引完成的排序操作称为“文件排序”</p> <p>-- 测试 Extra 的 filesort explain select * from emp order by name;</p> <p>Using join buffer: 改值强调了在获取连接条件时没有使用索引，并且需要连接缓冲区来存储中间结果。如果出现了这个值，那应该注意，根据查询的具体情况可能需要添加索引来改进能。</p> <p>Impossible where: 这个值强调了 where 语句会导致没有符合条件的行（通过收集统计信息不可能存在结果）。</p> <p>Select tables optimized away: 这个值意味着仅通过使用索引，优</p>

化器可能仅从聚合函数结果中返回一行
No tables used: Query 语句中使用 from dual 或不含任何 from 子句
--- explain select now() from dual;

--all:全表扫描，一般情况下出现这样的 sql 语句而且数据量比较大的话那么就需要进行优化。

```
explain select * from emp;
```

--index: 全索引扫描这个比 all 的效率要好，主要有两种情况，一种是当前的查询时覆盖索引，即我们需要的数据在索引中就可以索取，或者是使用了索引进行排序，这样就避免数据的重排序

```
explain select empno from emp;
```

--range: 表示利用索引查询的时候限制了范围，在指定范围内进行查询，这样避免了 index 的全索引扫描，适用的操作符： =, <>, >, >=, <, <=, IS NULL, BETWEEN, LIKE, or IN()

```
explain select * from emp where empno between 7000 and 7500;
```

--index_subquery: 利用索引来关联子查询，不再扫描全表

```
explain select * from emp where emp.job in (select job from t_job);
```

--unique_subquery: 该连接类型类似与 index_subquery, 使用的是唯一索引

```
explain select * from emp e where e.deptno in (select distinct deptno from dept);
```

--index_merge: 在查询过程中需要多个索引组合使用，没有模拟出来

--ref_or_null: 对于某个字段即需要关联条件，也需要 null 值的情况下，查询优化器会选择这种访问方式

```
explain select * from emp e where e.mgr is null or e.mgr=7369;
```

--ref: 使用了非唯一性索引进行数据的查找

```
create index idx_3 on emp(deptno);
```

```
explain select * from emp e,dept d where e.deptno =d.deptno;
```

--eq_ref : 使用唯一性索引进行数据查找

```
explain select * from emp,emp2 where emp.empno = emp2.empno;s
```

--const: 这个表至多有一个匹配行，

```
explain select * from emp where empno = 7369;
```

--system: 表只有一行记录（等于系统表），这是 const 类型的特例，平时不会出现

导致 MySQL 索引失效的几种常见写法

```
CREATE TABLE `user`(  
    `id` int(5) unsigned NOT NULL AUTO_INCREMENT,
```

```
 `create_time` datetime NOT NULL,  
 `name` varchar(5) NOT NULL,  
 `age` tinyint(2) unsigned zerofill NOT NULL,  
 `sex` char(1) NOT NULL,  
 `mobile` char(12) NOT NULL DEFAULT "",  
 `address` char(120) DEFAULT NULL,  
 `height` varchar(10) DEFAULT NULL,  
 PRIMARY KEY (`id`),  
 KEY `idx_createtime` (`create_time`) USING BTREE,  
 KEY `idx_name_age_sex` (`name`,`sex`,`age`) USING BTREE,  
 KEY `idx_height` (`height`) USING BTREE,  
 KEY `idx_address` (`address`) USING BTREE,  
 KEY `idx_age` (`age`) USING BTREE  
) ENGINE=InnoDB AUTO_INCREMENT=261 DEFAULT CHARSET=utf8;
```

1、使用!= 或者 <> 导致索引失效

```
SELECT * FROM `user` WHERE `name` != '冰峰';
```

2、类型不一致导致的索引失效(隐式转换)

```
SELECT * FROM `user` WHERE height= 175;
```

3、函数导致的索引失效

```
SELECT * FROM `user` WHERE DATE(create_time) = '2020-09-03';
```

4、运算符导致的索引失效

```
SELECT * FROM `user` WHERE age - 1 = 20;
```

5、OR 引起的索引失效

```
SELECT * FROM `user` WHERE `name` = '张三' OR height = '175';
```

6、模糊搜索导致的索引失效

```
SELECT * FROM `user` WHERE `name` LIKE '%冰';
```

7、NOT IN、NOT EXISTS 导致索引失效

```
SELECT s.* FROM `user` s WHERE NOT EXISTS (SELECT * FROM `user` u WHERE  
u.name = s.name AND u.name = '冰峰')  
  
SELECT * FROM `user` WHERE `name` NOT IN ('冰峰');
```

查询条件使用 not in 时，如果是主键则走索引，如果是普通索引，则索引失效。

8、is null 时正常走索引，使用 is not null 时，不走索引

```
SELECT * FROM `user` WHERE address IS NOT NULL
```

1. 全值匹配我最爱
2. 最佳左前缀法则 —— 如果索引了多列，要遵守最左前缀法则。指的是查询从索引的最左前列开始并且不跳过索引中的列。
3. 不在索引列上做任何操作（计算、函数、(自动or手动)类型转换），会导致索引失效而转向全表扫描
4. 存储引擎不能使用索引中范围条件右边的列
5. 尽量使用覆盖索引(只访问索引的查询(索引列和查询列一致))，减少select *
6. mysql 在使用不等于(!= 或者<>)的时候无法使用索引会导致全表扫描
7. is null ,is not null 也无法使用索引
8. like以通配符开头('%abc...')mysql索引失效会变成全表扫描的操作 —— 问题：解决like '%字符串%'时索引不被使用的方法？？
9. 字符串不加单引号索引失效
10. 少用or,用它来连接时会索引失效

索引用于排序时的例子

多列排序需要用组合索引

- create index first_idx on big_person(first_name);
- create index last_idx on big_person(last_name);
- explain select * from big_person order by last_name, first_name limit 10;
- alter table big_person drop index first_idx;

- alter table big_person drop index last_idx;
- create index last_first_idx on big_person(last_name,first_name);

多列排序需要遵循最左前缀原则

- explain select * from big_person order by last_name, first_name limit 10;
- explain select * from big_person order by first_name, last_name limit 10;
- explain select * from big_person order by first_name limit 10;

多列排序升降序需要一致

- explain select * from big_person order by last_name desc, first_name desc limit 10;
- explain select * from big_person order by last_name desc, first_name asc limit 10;

索引用于 where 筛选例子

模糊查询需要遵循字符串最左前缀原则

- explain SELECT * FROM big_person WHERE first_name LIKE 'dav%' LIMIT 5;
- explain SELECT * FROM big_person WHERE last_name LIKE 'dav%' LIMIT 5;
- explain SELECT * FROM big_person WHERE last_name LIKE '%dav' LIMIT 5;

组合索引需要遵循最左前缀原则

- create index province_city_county_idx on big_person(province,city,county);
- explain SELECT * FROM big_person WHERE province = '上海' AND city='宜兰县' AND county='中西区';
- explain SELECT * FROM big_person WHERE county='中西区' AND city='宜兰县'

AND province = '上海';

- explain SELECT * FROM big_person WHERE city='宜兰县' AND county='中西区';
- explain SELECT * FROM big_person WHERE county='中西区';

函数及计算问题

- create index birthday_idx on big_person(birthday);
- explain SELECT * FROM big_person WHERE ADDDATE(birthday,1)='2005-02-10'; 失效
- explain SELECT * FROM big_person WHERE birthday=ADDDATE('2005-02-10',-1); 生效

隐式类型转换问题

- create index phone_idx on big_person(phone);
- explain SELECT * FROM big_person WHERE phone = 13000013934;
- explain SELECT * FROM big_person WHERE phone = '13000013934';

索引条件下推

哪些条件能利用索引

- explain SELECT * FROM big_person WHERE province = '上海';
- explain SELECT * FROM big_person WHERE province = '上海' AND city='嘉兴市';
- explain SELECT * FROM big_person WHERE province = '上海' AND city='嘉兴市' AND county='中西区';
- explain SELECT * FROM big_person WHERE province = '上海' AND county='中

西区';

MySQL 执行条件判断的时机有两处：引擎层（包括了索引实现）和服务层

- 上面第 4 条 SQL 中仅有 province 条件能够利用索引，在引擎层执行
- 但 county 条件仍然要交给服务层处理
- 在 5.6 之前，服务层需要判断所有记录的 county 条件，性能非常低
- 5.6 以后，引擎层会先根据 province 条件过滤，满足条件的记录才在服务层处理 county 条件

索引条件下推

- SELECT * FROM big_person WHERE province = '上海' AND county='中西区';
- SET optimizer_switch = 'index_condition_pushdown=off';

例子：

- 有一个联合索引(a,b,c)，当查询 where a=x and c=x2，根据最左匹配原则，只有 a 会用得上，对于 c 不同版本就有区别了：
- 在 MySQL5.6 之前，就是根据二级索引找出每一条 a=x 的叶子节点的主键，然后拿着主键去回表，根据回表拿到对应行 c 的值，将不符合 c=x2 的行过滤掉
- 在 MySQL5.6 引入了索引下推优化，找到 a=x 后，因为在二级索引中就有 c 的值了，MySQL Server 就会将 c=x2 这个索引条件下推给存储引擎去做，所以在二级索引内就判断该行是否符合 c=x2，不符合直接就跳过了，这样就少了很多次回表

更多例子

二级索引覆盖的例子

- explain SELECT * FROM big_person WHERE province = '上海' AND city='宜兰县'
' AND county= '中西区';
- explain SELECT id,province,city,county FROM big_person WHERE province = '
上海' AND city='宜兰县' AND county='中西区';

表连接需要在连接字段上建立索引

不要迷信网上说法，具体情况具体分析

- create index first_idx on big_person(first_name);
- explain SELECT * FROM big_person WHERE first_name > 'Jenni'; **失效**
- explain SELECT * FROM big_person WHERE first_name > 'Willia'; **生效**
- explain select * from big_person where id = 1 or id = 190839;
- explain select * from big_person where first_name = 'David' or last_name =
'Thomas';
- explain select * from big_person where first_name in ('Mark', 'Kevin', 'David');
- explain select * from big_person where first_name not in ('Mark',
'Kevin', 'David'); **失效**
- explain select id from big_person where first_name not in ('Mark',
'Kevin', 'David'); **生效**

以上实验基于 5.7.27，其它如 !=、is null、is not null 是否使用索引都会跟版本、实际数据相关，以优化器结果为准

Mysql8 主从复制

1. 授权给从机服务器

```
CREATE USER 'slave1'@'192.168.157.155' IDENTIFIED WITH
```

```
caching_sha2_password BY 'Slave@123';
```

```
GRANT REPLICATION SLAVE ON *.* TO 'slave1'@'192.168.157.155';
```

```
FLUSH PRIVILEGES;
```

```
SHOW MASTER STATUS;
```

2. 修改主库配置文件

开启 binlog , 并设置 server-id

3. 查看主服务器当前二进制日志名和偏移量

```
show master status;
```

4. 修改从库配置文件

注意从机这里只需要配置一下 server-id 即可

5. 使用命令来配置从机

```
CHANGE REPLICATION SOURCE TO
```

```
SOURCE_HOST='192.168.157.154',SOURCE_PORT=3306,SOURCE_USER
```

```
='slave1',SOURCE_PASSWORD='Slave@123',SOURCE_LOG_FILE='mysql-
```

```
bin.000002',SOURCE_LOG_POS=157,GET_SOURCE_PUBLIC_KEY=1;
```

```
STOP REPLICA;
```

```
START REPLICA;
```

```
SHOW SLAVE STATUS;
```

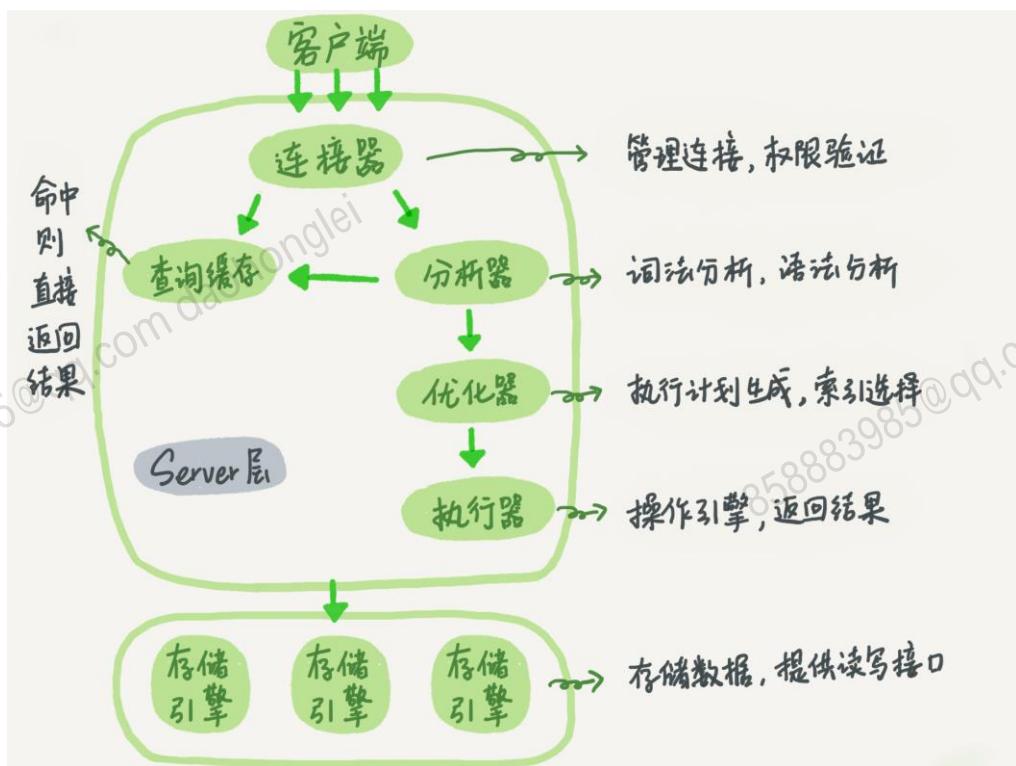
6. 启动 slave 进程

```
start slave;
```

7.启动之后查看从机状态:

show slave status\G;

一条 sql 查询语句是如何执行的



1. 连接：连接器会校验用户的账号和密码，验证通过后，然后会去权限表获取用户拥有的权限。

长连接：数据库连接成功后，如果客户端一直有请求，则会一直使用同一个连接。

短连接：每次执行完很少的几次连接后，会自动断开。下次查询会再重建一个。

2. 查询缓存：建立连接后，会优先查询缓存，若对应缓存存在，则直接返回结果，查询缓存已 key->value 的形式存储在内存中，key 为查询的 sql，value 为查询的结果。

若有对一个表进行更新，那么这个表的所有查询缓存均会失效。因此，**查询缓存弊往往大于利，不建议使用。（mysql8.0 及之后版本均不在支持查询缓存）**

3. 分析器：分析器先做 **词法分析**，识别出 SQL 语句中的字符串分别是什么，代表什么。

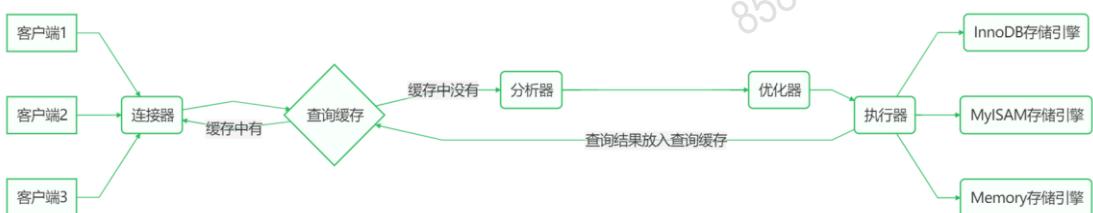
再做 **语法分析**，根据语法规则，判断 SQL 是否满足 MySQL 语法规则。

4. 优化器：如在表里存在多个索引时，决定具体哪个索引；在具体执行 SQL 时，决定执行的先后顺序（join 关联多个表时，先执行 A 表的 where 条件或是 B 表的）

5. 执行器：**开始执行语句**，先判断是否有对执行表的权限，然后根据表的引擎定义去使用引擎所提供的接口。

执行 SQL 语句 `select * from user where id = 1` 时发生了什么

1. 连接器：负责建立连接、检查权限、连接超时时间由 `wait_timeout` 控制，默认 8 小时
2. 查询缓存：会将 SQL 和查询结果以键值对方式进行缓存，修改操作会以表单位导致缓存失效
3. 分析器：词法、语法分析
4. 优化器：决定用哪个索引，决定表的连接顺序等
5. 执行器：根据存储引擎类型，调用存储引擎接口
6. 存储引擎：数据的读写接口，索引、表都在此层实现



redo log

redo log 的作用主要是实现 ACID 中的持久性，保证提交的数据不丢失

- 它记录了事务提交的变更操作，服务器意外宕机重启时，利用 redo log 进行回放，重新执行已提交的变更操作
- 事务提交时，首先将变更写入 redo log，事务就视为成功。至于数据页（表、索引）上的变更，可以放在后面慢慢做
 - 数据页上的变更宕机丢失也没事，因为 redo log 里已经记录了
 - **数据页在磁盘上位置随机，写入速度慢，redo log 的写入是顺序的速度快**
- 它由两部分组成，内存中的 redo log buffer，磁盘上的 redo log file
 - redo log file 由一组文件组成，当写满了会循环覆盖较旧的日志，这意味着不能无限依赖 redo log，更早的数据恢复需要 binlog
 - buffer 和 file 两部分组成意味着，写入了文件才真正安全，同步策略由下面的参数控制
 - innodb_flush_log_at_trx_commit
 - ◆ 0 - 每隔 1s 将日志 write and flush 到磁盘
 - ◆ 1 - 每次事务提交将日志 write and flush （默认值）
 - ◆ 2 - 每次事务提交将日志 write page cache，每隔 1s flush 到磁盘，这意味着 write 意味着写入操作系统缓存，如果 MySQL 挂了，而操作系统没挂，那么数据不会丢失

MySQL InnoDB 的关键特性

插入缓存

两次写

自适应哈希索引

异步 IO

刷新邻近页

insert buffer

insert buffer 只适用于 non-unique secondary indexes 也就是说只能用在非唯一的索引上，原因如下

1、primary key 是按照递增的顺序进行插入的，平常插入聚簇索引一般也顺序的，非

随机 IO

2 写唯一索引要检查记录是不是存在，所以在修改唯一索引之前，必须把修改的记录相关的索引页读出来才知道是不是唯一、这样 Insert buffer 就没意义了，要读出来(随机 IO)

insert buffer 的原理

对于为非唯一索引，辅助索引的修改操作并非实时更新索引的叶子页，而是把若干对同一页面的更新缓存起来做，合并为一次性更新操作，减少 IO，转随机 IO 为顺序 IO，这样可以避免随机 IO 带来性能损耗，提高数据库的写性能

先判断要更新的这一页在不在缓冲池中

a、若在，则直接插入；

b、若不在，则将 index page 存入 Insert Buffer，按照 Master Thread 的调度规则来合并非唯一索引和索引页中的叶子结点

insert buffer 的缺点

1、可能导致数据库宕机后实例恢复时间变长。如果应用程序执行大量的插入和更新操作，且涉及非唯一的聚集索引，一旦出现宕机，这时就有大量内存中的插入缓冲区

数据没有合并至索引页中，导致实例恢复时间会很长

2、在写密集的情况下，插入缓冲会占用过多的缓冲池内存 (innodb_buffer_pool) ，

默认情况下最大可以占用 1/2，这在实际应用中会带来一定的问题

3、insert buffer 无法进行控制，for different workloads and hardware

configuration，特别是在 SSD 盛行的今天

两次写和 redo log 的区别

1.两次写特性能加强 MySQL 的可靠性。在脏页写入到表之前，先将一份副本写入硬盘，成功之后再写入表。这样做的目的是预防脏页写入表时，发生最坏情况（例如：掉电），导致页损坏，也就是说至少部分数据无法读取。两次写避免了这个问题。当最坏情况发生，服务重新启动时，系统检测页发生了损坏，那么会读取两次写的副本来恢复损坏的页。这样的系统更加可靠。

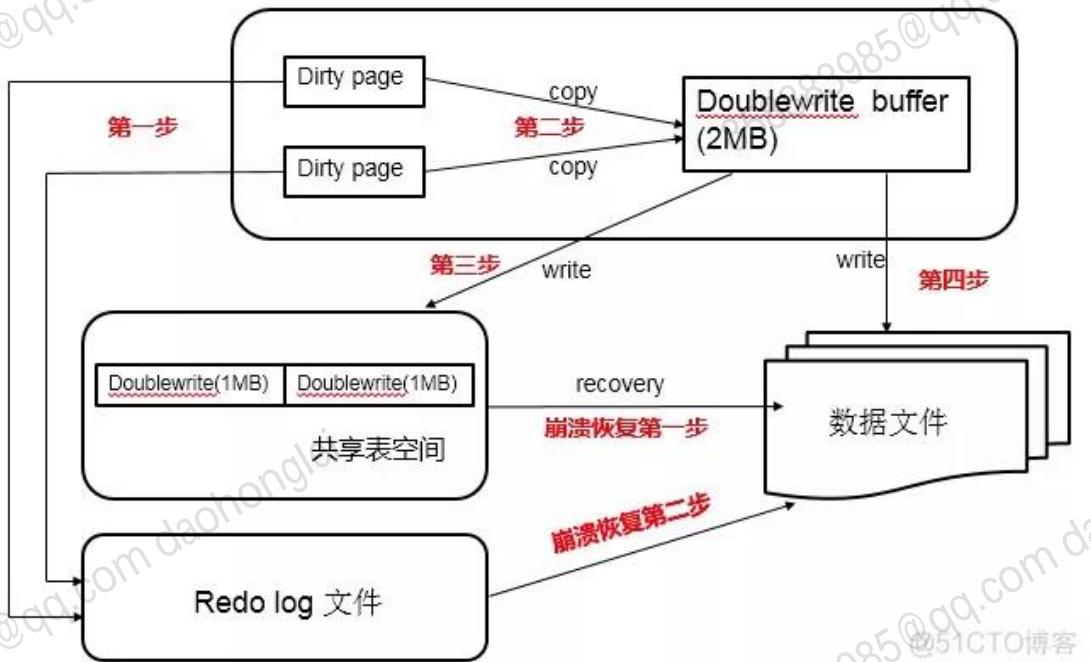
2.redo log 是当脏页写入前，先写入到 redo 日志。当发生掉电时，缓存内的部分脏页会丢失，在系统恢复后，将从 redo log 中恢复缓存未写入磁盘的数据。

这看起来两者有点相似，再掉电后都有恢复数据的功能。不过两者的功能还是有较大区别。

a.两次写所存储的数据较小，它的重点是预防马上要写入的数据页，在故障中损坏后的恢复。也就是说重点是修复物理存储介质上损坏的数据页。

b.redo log 重点是故障中，缓存脏页未能写入磁盘的数据，能通过 redo log 找到丢失的数据得以完整写入到磁盘。

总结：两次写解决的是脏页损坏不可用的问题，redo log 解决的是脏页没损坏但是脏页数据丢失的问题。



redo log 和 binlog 区别与作用

这两种日志有以下三点不同。

1. redo log 是 InnoDB 引擎特有的；binlog 是 MySQL 的 Server 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“**在某个物理数据页上做了什么修改**”；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 1”。
3. redo log 是循环写的，空间固定会用完；binlog 是可以追加写入的。“追加写”是指 binlog 文件写到一定大小后会切换到下一个，并不会覆盖以前的日志。

update 语句时的内部流程：

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=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。

2. 执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。

3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，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 改成提交(commit)状态，更新完成。

redo log 刷盘时机

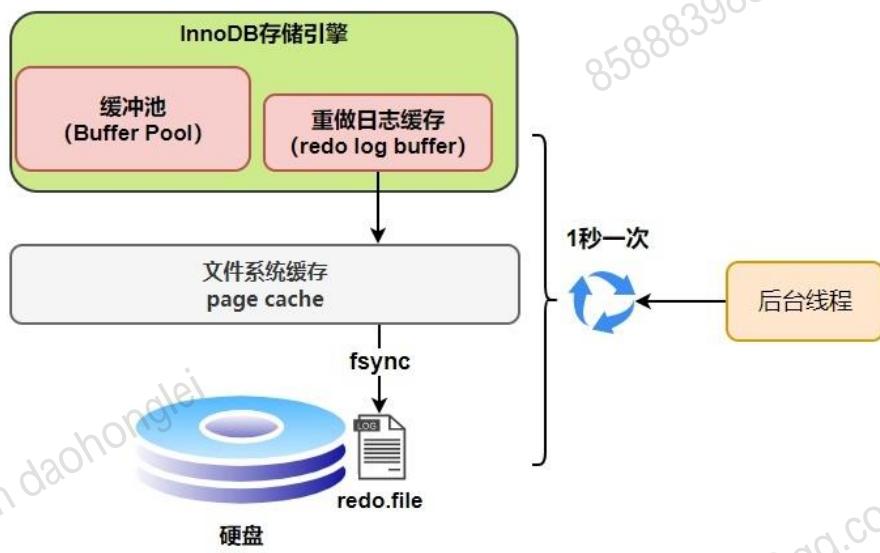
一次写入 512 个字节，也就是一个扇区，扇区是写入的最小单位，因此可以保证写入必定是成功的，因此重做日志不需要 doublewrite

InnoDB 存储引擎为 redo log 的刷盘策略提供了

`innodb_flush_log_at_trx_commit` 参数，它支持三种策略

- 设置为 0 的时候，表示每次事务提交时不进行刷盘操作
- 设置为 1 的时候，表示每次事务提交时都将进行刷盘操作（默认值）
- 设置为 2 的时候，表示每次事务提交时都只把 redo log buffer 内容写入 page cache

另外 InnoDB 存储引擎有一个后台线程，每隔 1 秒，就会把 redo log buffer 中的内容写到文件系统缓存（page cache），然后调用 `fsync` 刷盘。

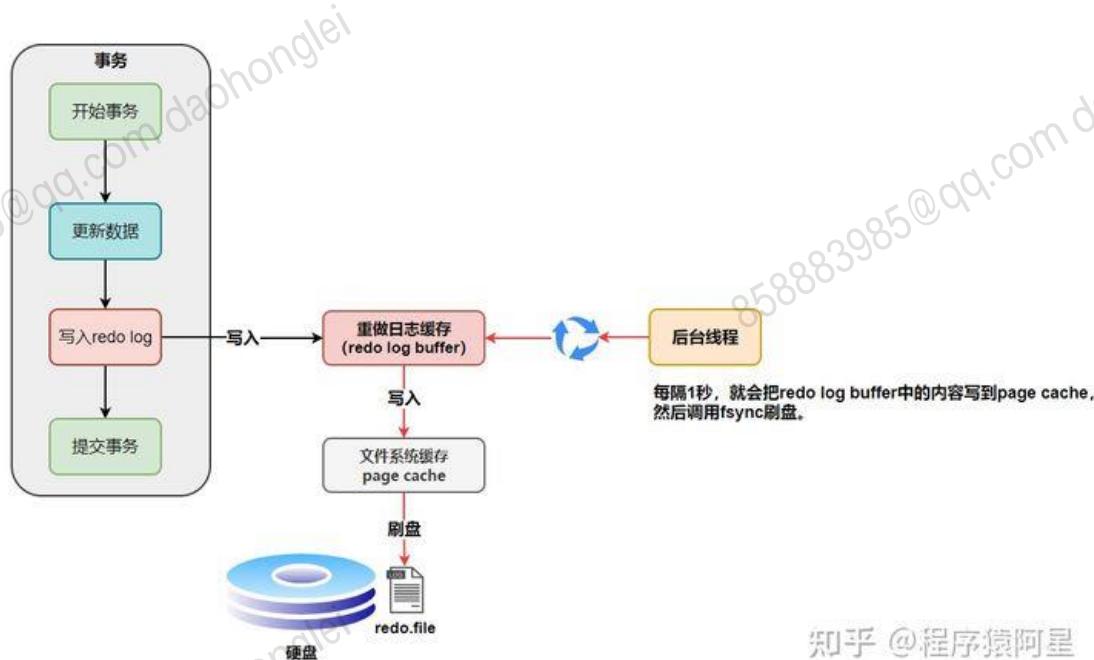


知乎 @程序猿阿星

也就是说，一个没有提交事务的 redo log 记录，也可能会刷盘。

为什么呢？

因为在事务执行过程 redo log 记录是会写入 redo log buffer 中，这些 redo log 记录会被后台线程刷盘。



知乎 @程序猿阿星

除了后台线程每秒 1 次的轮询操作，还有一种情况，当 redo log buffer 占用

的空间即将达到 `innodb_log_buffer_size` 一半的时候，后台线程会主动刷盘。

InnoDB 磁盘数据页的存储结构

原理

- 数据库最终所有的数据（包括我们建的各种表和表里的数据）都是要存放在磁盘上的文件的
- 然后在文件里存放的物理格式就是数据页

准备

当前有表如下

```
create table index_demo{  
    c1 int,  
    c2, int,  
    c3 char(1),  
    primary key (c1)  
} ROW_FORMAT = COMPACT;
```

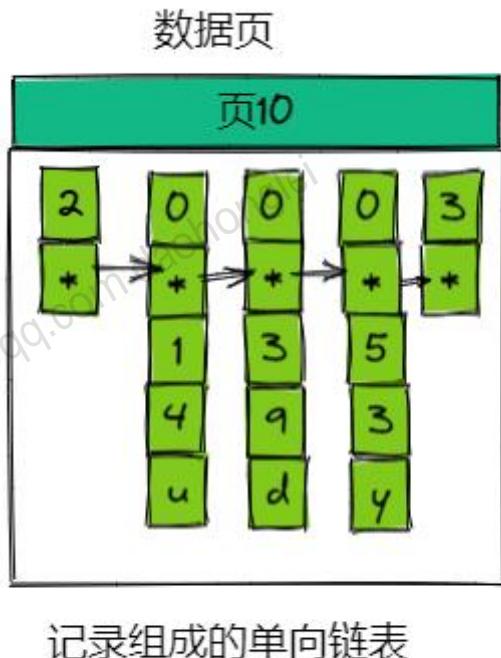
数据页内部的结构

- 我们知道数据是一行一行插入的，**当需要插入数据时，会先申请一个数据页，然后将它按照一定的格式写入到数据页中**，其行列格式由 `ROW_FORMAT = COMPACT` 指定
- 数据页内部的数据行将按照**主键值从小到大串联成一个单向链表**

举个例子：

- 比如现在我们插入 3 条记录 1、3、5，那么数据页内部如下：

从上面可以看到，里面是一行一行的数据：



记录组成的单向链表

- 刚开始的一行是起始行，它的行类型是 2，就是最小的一行，然后它有一个指针指向了下一行数据
- 每一行数据都有自己每个字段的值，然后每一行通过一个指针不停的指向下一个数据，普通的数据行的类型都是 0，而且它们是按照主键大小组织的单向链表
- 最后一行是一个类型为 3 的，就是代表最大的一行

页空间不足了！！！

- 假设你不停地往表里插入数据，那么刚开始就是不停的在一个数据页插入数据。但随着数据越来越多，超出了一个数据页的容量，这个时候就必须再申请一个数据页来存放新数据行了
- 但是此时会遇到一个问题：索引运作的一个核心机制就是要求你后一个数据页的主键

值都大于前一个数据页的主键值。

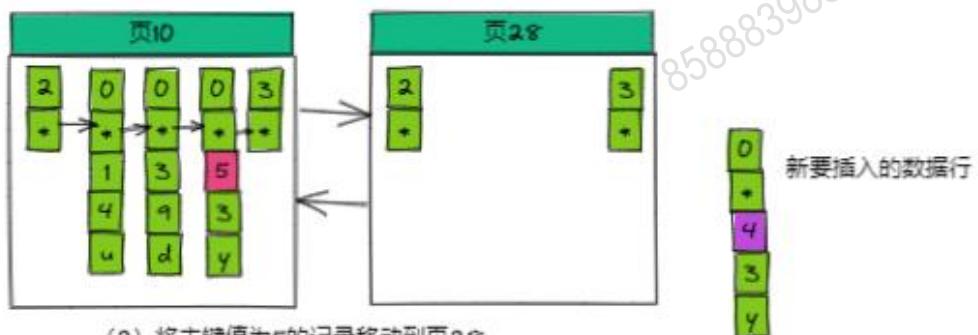
- 如果你的主键是自增的，那还可以保证这一点，因为你新插入后一个数据页的主键值一定都大于前一个数据页的主键值
- 但是有时候你的主键并不是自增的，所以可能会出现你后一个数据页的主键值里，有的主键是小于前一个数据页的主键值的。
- 对于第二种情况，这个时候就会出现一个过程，叫做**页分裂**。就是万一你的主键值都是你自己设置的，那么在增加一个新的数据页的时候，实际上会把前一个数据页里主键值较大的，挪动到新的数据页里来，然后把你新插入的主键值较小的数据挪动到上一个数据页里去，保证新数据页里的主键值一定都比上一个数据页里的主键值大。

举个例子

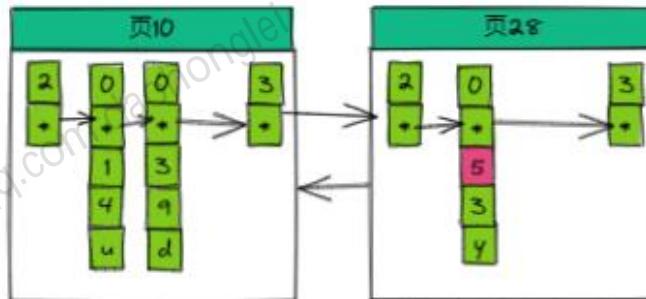
接着上面，我们又想插入一行记录 4：

- 但是发现页空间不够了，所以会再申请一个空页
- 新申请的数据页可能不是连续的，它们是通过维护上一个页和下一个页的编号而建立了链接关系（比如新申请了一个页 28 而不是页 11）
- 因为新插入记录的主键值 4 < 上一页的最大主键值 5。而我们需要保证**上一页的主键值必须大于下一页的主键值**，所以需要进行一些移动。

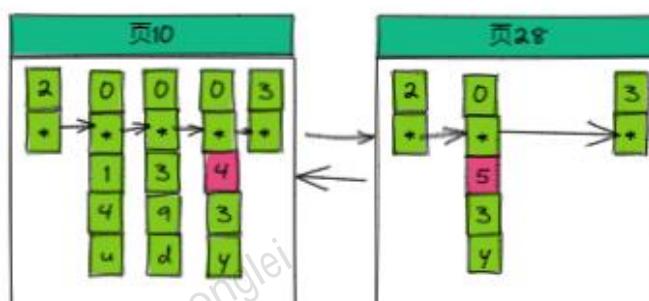
(1) 发现上一页的最大主键是5, $5 > 4$



(2) 将主键值为5的记录移动到页28



(3) 将主键值为4的记录插入到页10



上面就是一个页分裂的过程，核心目标是保证下一个数据页里的主键都比上一个数据页里的主键值要大

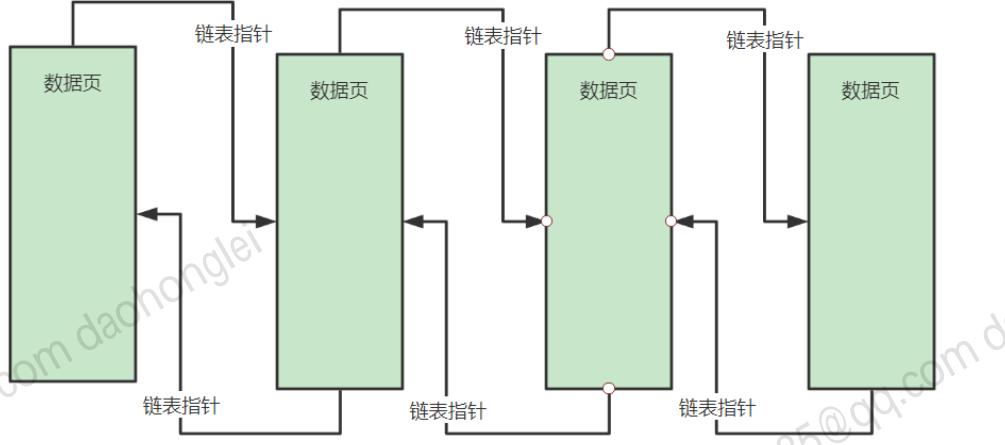
总结：在你不停的往表里插入数据的时候，会搞出来一个一个的数据页，如果你的主键不是自增的，那么就可能有一个数据行的挪动过程，保证你下一个数据页的主键值都大于上一个数据页的主键值

数据页之间是怎么存储的

但是上面可以发现有两个问题：

数据页在磁盘文件里是怎么存储的呢？

- 大量的数据页是一页一页的存放的，然后两个相邻的数据页之间会采用**双向链表**的格式相互引用。



那么上图中在磁盘文件里到底是怎么弄出来的呢？

- 其实一个数据页在磁盘文件里就是一段数据，可能是二进制或者别的特殊格式的数据，然后数据页里包含两个指针，一个指针指向自己上一个数据页的物理地址，一个指向执行自己下一个数据页的物理地址。如下：

```

DataPage: xx=xx, xx=xx, linked_list_pre_pointer=15367,
linked_list_next_pointer=34126 ||

DataPage: xx=xx, xx=xx, linked_list_pre_pointer=23789,
linked_list_next_pointer=46589 ||

DataPage: xx=xx, xx=xx, linked_list_pre_pointer=33198,
linked_list_next_pointer=55681

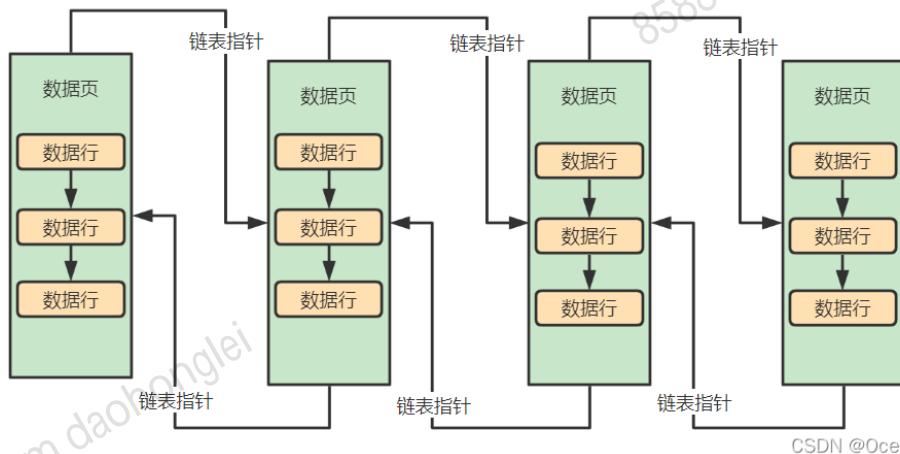
```

- 每个数据页在磁盘文件里都是连续的一段数据
 - 每个数据页里，可以认为就是 DataPage 打头到||符号的一段磁盘里的连续的数据。
 - 每个数据页，都有一个指针指向自己上一个数据页在磁盘文件里的起始物理位置，

比如 `linked_list_pre_pointer=15367`, 就是指向了上一个数据页在磁盘文件里的起始物理位置, 那个 15367 可以认为就是在磁盘文件里的 `position` 或者 `offset`

- 同理, 也有一个指针指向自己下一个数据页的物理位置。
- 也就是说, 一个磁盘文件里的多个数据页通过 **指针** 组成了一个双向链表。

然后, 数据页内部会有数据行, 数据行按照**主键值**从小到大串联成一个串联成一个**单向链表**表, 如下图:



CSDN @Ocean&&Star

页目录 (每个数据页都有自己的页目录)

问题的引出

从上面我们可以知道, 数据页内部的数据行是按照**主键值**组织成的**单向链表**。

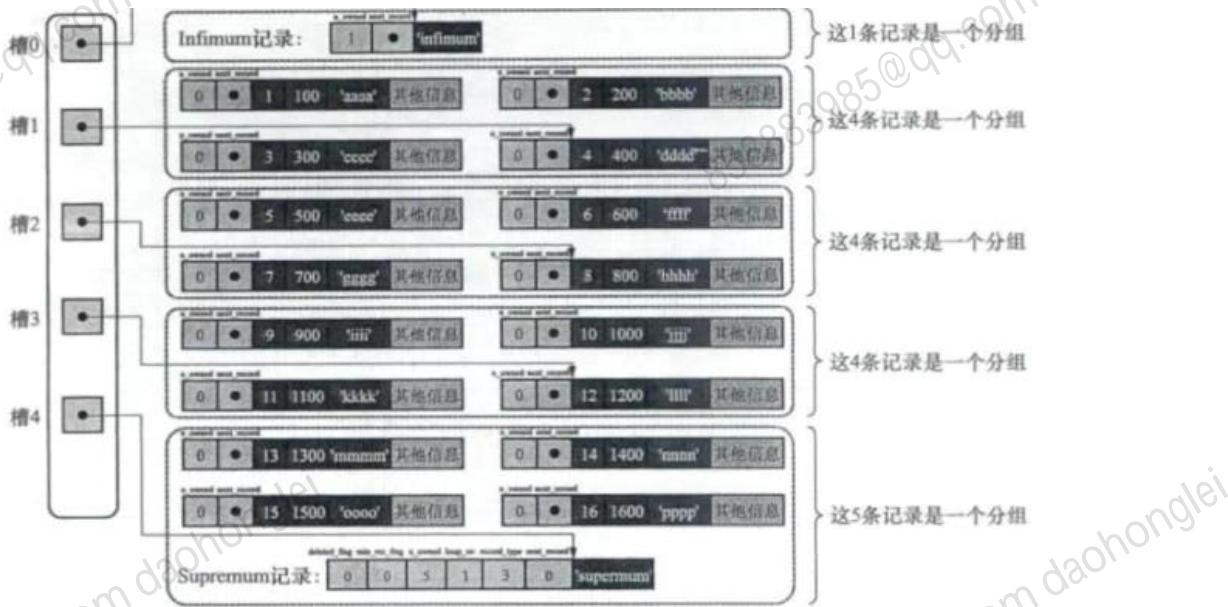
如果已经定位到了某一页, 现在我们想要在这个数据页中根据**主键值**查找某条记录, 怎么找呢?

- 一般来说, 直接遍历就可以找到。但是当单向链表太长了, 那么性能就很低了。
- 怎么办呢? 这就引入了**页目录**, 每个数据页都有自己的页目录。

页目录的制作过程

- 目录制作过程如下：

- 数据页都会将**行数据**划分成几个组
- 每个组中最大的那条**记录**在页面中的**地址偏移量**会被单独提取出来，按照顺序存储到靠近页尾部的地方
- 这个地方就是“**页目录**”。页目录中的这些地址偏移量叫做**槽**。
- 各个槽之间是挨着的，而且它们代表的记录的主键值从小到大排序。

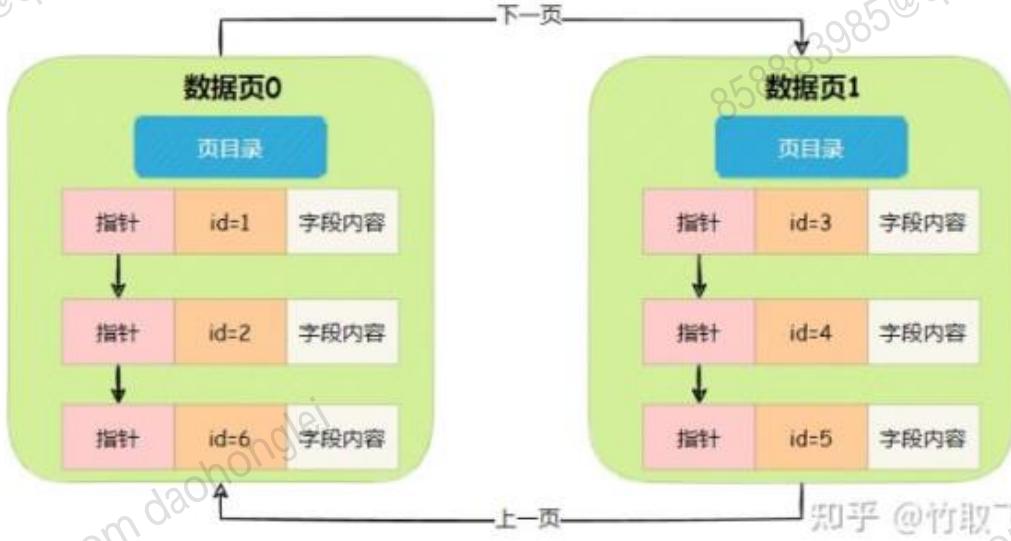


- 这样我们就能快速找到主键对应的数据行的实际存储位置

- 首先先到数据页的

页目录里根据主键进行二分查找，这样就可以迅速**定位到主键对应的数据是在哪个槽位里**

- 然后到那个槽位里去，遍历槽位里每一行数据，就能快速找到那个主键对应的数据了。



那么假设你是要根据**非主键**的其他字段查找呢？

- 这时可能是没有办法使用主键的页目录进行二分查找的
- 只能进入到数据页里，根据单向链表依次遍历查找。
- 此时性能会很差。

主键目录

问题的引出

现在我们要根据主键来查找某条索引，但是现在很多数据页，那应该怎么定位到对应主键在哪个数据页上呢？

从上面我们知道：

- 数据页页号可能不是连续的
- 数据页之间是双向链表结构

在目前情况下：

- 无论是根据主键来找还是非主键来找，实际上都是没有什么取巧的方法
- 因为一个表里所有数据页都是组成双向链表的，此时就只能从第一个数据页开始遍历所有数据页。从第一个数据页开始，你需要先把第一个数据页从磁盘上读取到内存

buffer pool 的缓存页里来。

- 然后你就在第一个数据页对应的缓存页里，按照上述办法查找，假设是根据主键查找的，你可以在数据页的页目录里二分查找，假设你要是根据其他字段查找的，只能是根据数据页内部的单向链表来遍历查找
- 假设第一个数据页没找到你要的那条数据呢？没办法，就只能根据双向链表去找下一个数据页，然后读取到 buffer pool 的缓存页里去，然后按照一样的方法在一个缓存页内部查找那条数据。
- 以此类推，循环往复。

也就是说：

- 如果没有建立索引，就只能对数据库进行全表扫描，就是根据双向链表依次把磁盘上的数据页加载的缓存页里去，然后在一个缓存页内部来查找那条数据。
- 最坏的情况下，你需要把所有数据页里的每条数据都遍历一遍才行，才能找到你需要的那条数据，这就是全表扫描

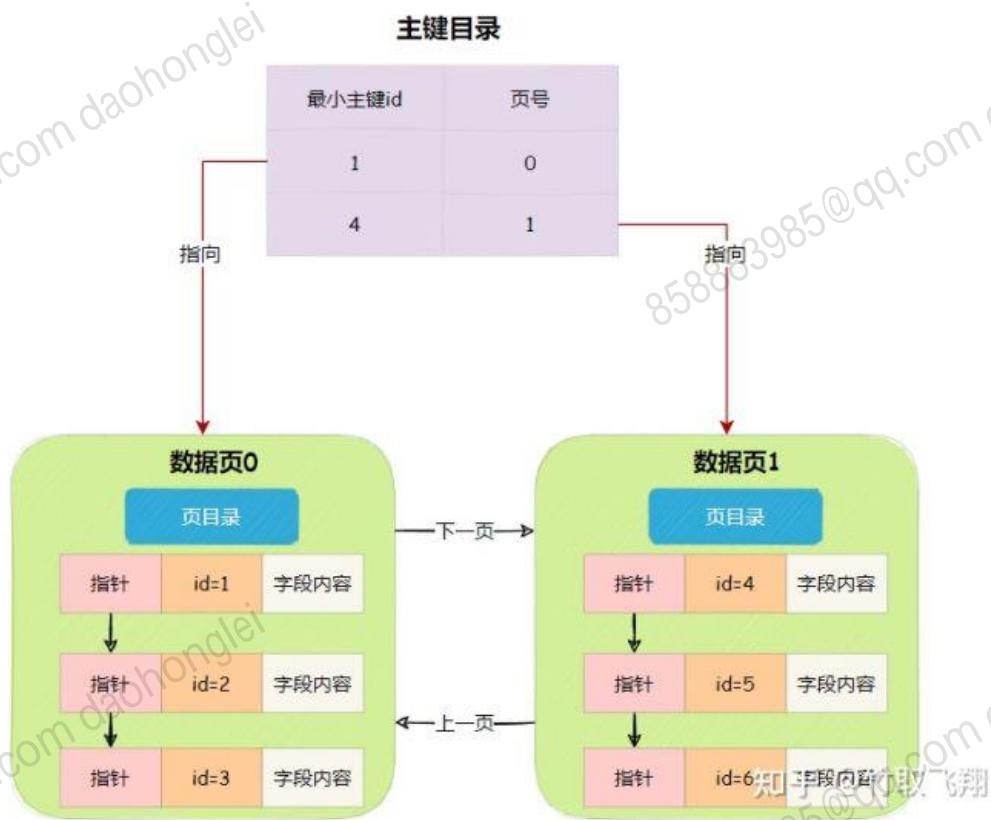
怎么改进呢？

- 和页目录一样，可以为所有的页建立一个目录
- 由于数据页的编号可能不是连续的，为快速根据主键值定位页目录，需要给它们建立一个目录，每个目录项包括两个部分：
 - 页的用户记录中的最小主键值 key
 - 页号 page_no
- 这些目录项在物理上是连续顺序存储的，所以我们可以用二分法快速定位主键所在页目录编号
- 这个东西就叫做主键目录。 主键目录就是**把每个数据页的页号，还有数据页的最小的**

主键值放在一起，组成一个索引的目录。

有了主键目录的加持，那找数据就非常快了，过程如下

- 二分查找主键目录，找到对应的数据页
- 进入数据页，二分查找数据页目录，找到对应的行数据



B+树的诞生

聚簇索引

新的问题

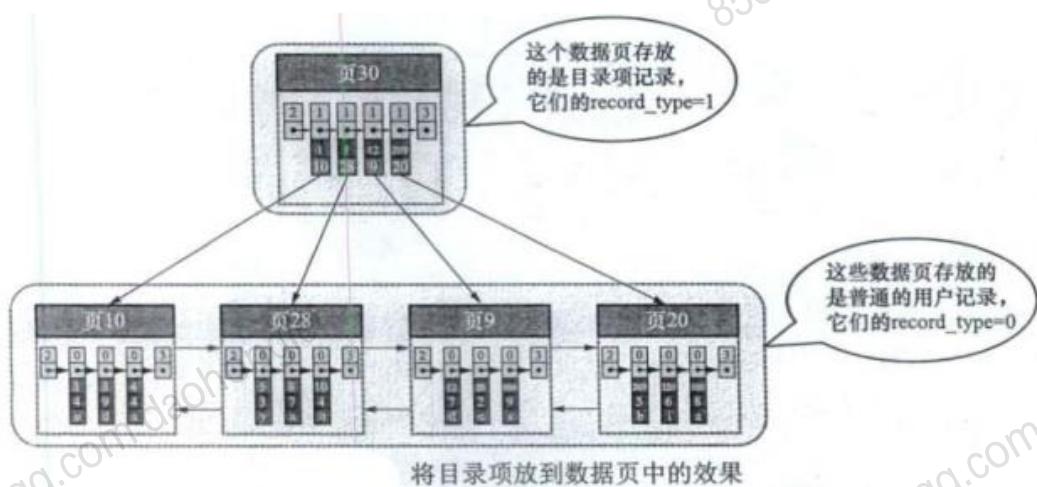
上面的所有目录项必须在物理上连续存储，但是：

- InnoDB 使用页作为管理存储空间的基本单位，也就是最多只有 16KB 的连续存储空间。如果页目录大小超过了 16KB 怎么办？
- 我们经常需要对记录进行增删改操作，如果我们把页 28 中的记录都删除，那么需要

移除目录项 2，也就意味着我们需要把目录项 2 后的目录项向前移动一下，这样就太慢了；如果不移除目录项 2，又会浪费太多空间。怎么办？

InnoDB 是怎么处理的？

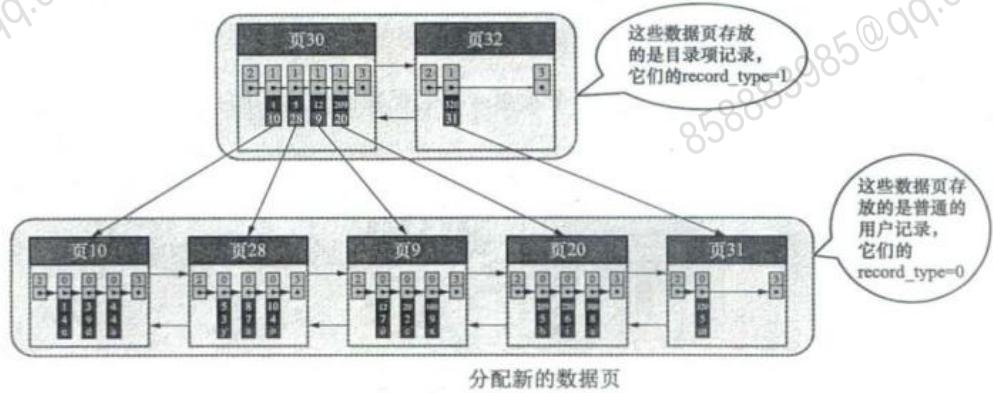
- 解决方案：使用之前存储用户记录的数据页来存储目录项，这样就可以解决第一个问题了
- 那么又有一个问题了：怎么区分一条记录是用户记录还是目录项记录呢？行记录的头信息中有一项叫做 record_type，其取值为 0 时表示用户记录，取值为 1 时表示目录项记录，这样就区分了



如果一个页不能存储更多目录项了，怎么办？

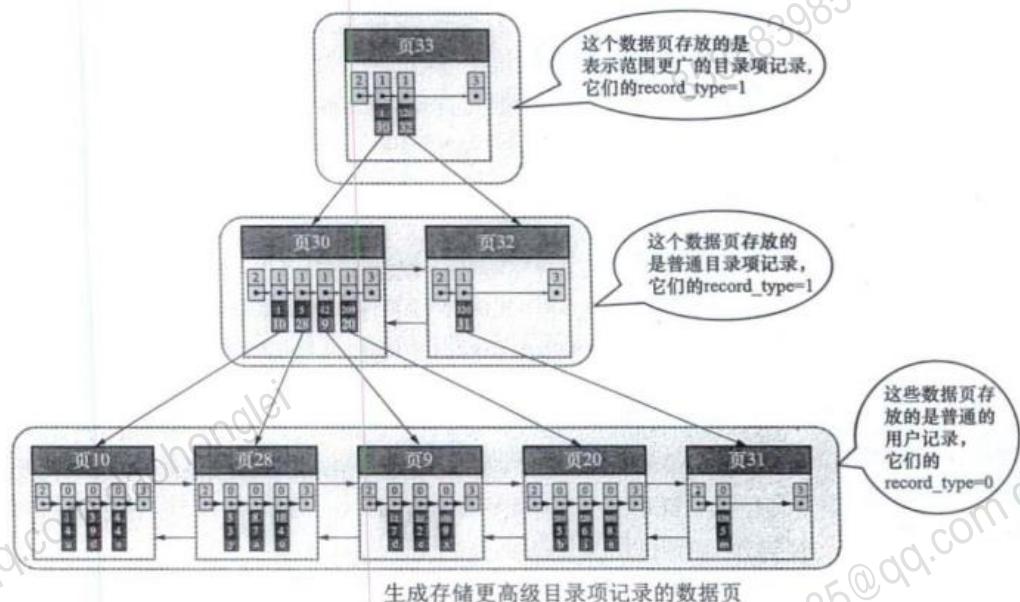
就会申请一个新的页来存储目录项

但是问题是：这样页在存储空间中并不是连续的，如果目录项记录页太多了，又会浪费存储空间



怎么办呢？解决方法是为这些存储目录项记录的页再生成一个更高级的目录

...随着表中记录的增加，这个目录的层级会继续增加。这就是 B+树，其叶子节点存储的是真正的行数据，其非叶子节点(也叫做内节点)存储的是目录项记录



B+树的特点：

- 使用记录主键值的大小进行记录和页的排序。也就是说：
 - 页(用户记录页、目录项记录页)中的记录按照主键的大小顺序形成了一个单向链表
 - 用户记录页根据页中记录的主键大小形成了一个双向链表
 - 目录项记录页分成不同的层级，在同一层级中的页也是根据页中主键大小顺序形成一个双向链表

■ 这意味着聚簇索引只能在搜索条件是主键值时才能发挥作用

- B+树的叶子节点存储的是完整的用户记录

这样的 B+数就叫做聚簇索引。

实际上 B+树的形成过程如下：

- 每当为某个表创建一个 B+树索引时，都会为这个索引创建一个根节点页面。
- 最开始表中没有数据时，每个 B+数索引对应的根节点中是空的
- 随着表中插入数据，先把用户记录存储到这个根节点中
- 当根节点可用节点用完了又插入记录，此时会先把根节点中所有记录复制到一个新分配的页中，然后对这个新页进行页分裂操作，得到另一个新页。这时新插入的记录会根据键值的大小分配到页 a 或者页 b 中。根节点此时就升级为目录项记录页

这个过程中，可以看出，一个 B+树索引的根节点自创建起(页号)就不会改变，此后每次需要用到这个索引时，都会从固定的地方取出根节点的页号，从而访问这个索引

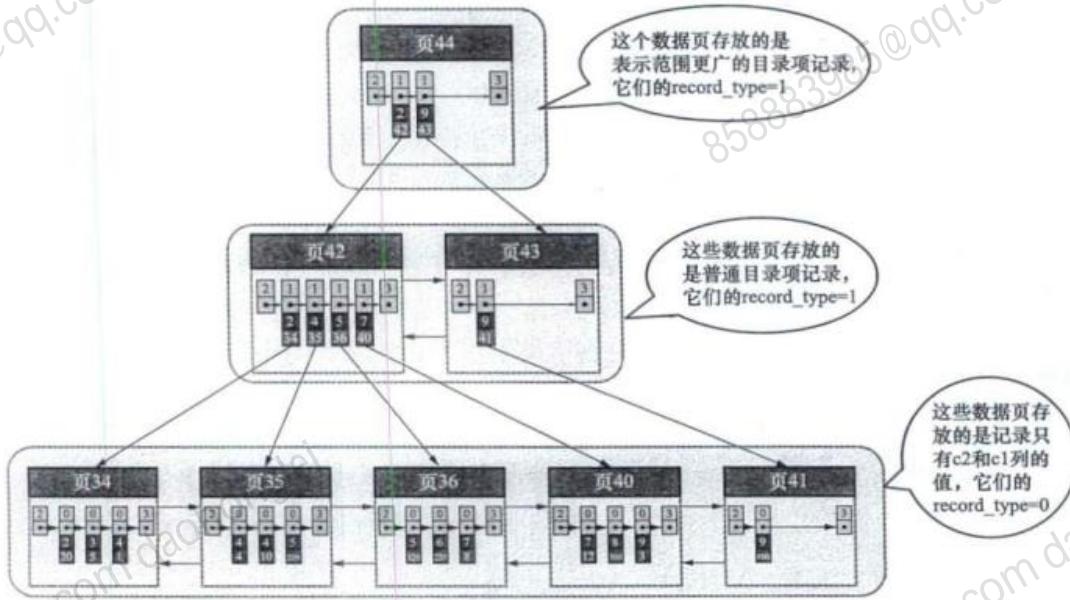
InnoDB 必须至少要有一个聚簇索引

- 如果表设置了主键，则主键就是聚簇索引
- 如果表没有主键，则会默认第一个 NOT NULL，且唯一 (UNIQUE) 的列作为聚簇索引
- 以上都没有，则会默认创建一个隐藏的 row_id 作为聚簇索引

二级索引

如果需要用别的列作为搜索条件怎么办？

我们可以多建几颗 B+数，并且不同的 B+树中的数据采用不同的排序规则。如下我们用 c2 的大小作为排序规则：



这个 B+树和上面 B+数不同的是：

- 它使用记录 c2 列的大小进行记录和页的排序
 - 页(叶子节点和非叶子节点)中记录是按照 c2 的大小顺序形成的单向链表
 - 用户记录页是根据页中记录的 c2 列大小顺序形成的双向链表
 - 目录项记录页分成不同的层级，在同一层级中的页也是根据页中 c2 大小顺序形成一个双向链表
- B+数的叶子节点存储的并不是完整的用户记录，而是 c2 列+主键这两个列的值
- 目录项记录中不再是主键+页号，而是 c2 列+页号

因为 c2 不满足唯一性约束，所以满足搜索条件的可能有多个，因此，我们先根据这个 B+ 树通过 c2 值定位**主键**，然后再用**主键**去聚簇索引中定位完整的用户记录(这个过程叫做**回表**)，找到一条记录之后再返回当前 B+树继续找下一个 c2 值，然后定位主键，再回表找到对应叶子节点...

为什么需要回表而不是把完整的用户记录放到二级索引的叶子节点呢？为了节省空间，我们不可能每建立一颗 B+树就把所有的用户记录复制一遍。

这种以非主键列的大小为排序规则而建立的 B+树需要执行的回表操作才能定位到完整的

用户记录，所以这种 B+树也叫做**二级索引/辅助索引**。

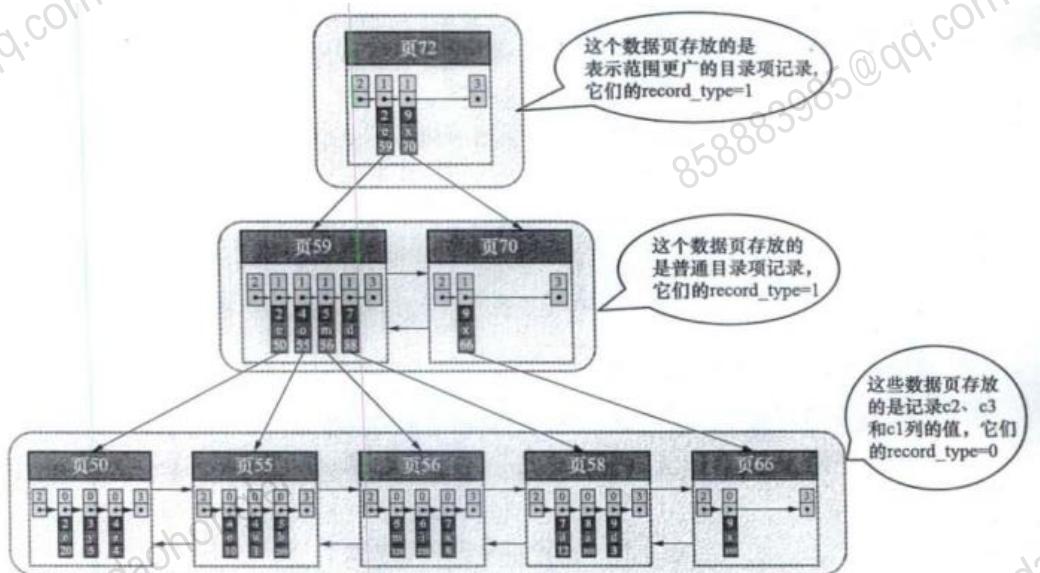
二级索引和聚簇索引使用的是一样的记录行格式，只是二级索引存储的列是不完整的

联合索引

我们也可以同时为多个列建立索引。比如以 c2 和 c3 的大小排序，也就是

- 先把各个记录和页按照 c2 进行排序
- 如果 c2 相同，在按照 c3 进行排序

这样的索引叫做**联合索引**。联合索引树的特点：



- 每个目录项记录都是由 c2、c3、页号这 3 部分组成，各个记录先按照 c2 排序，如果 c2 相同，再按照 c3 排序
- 叶子节点的用户记录由 c2、c3、主键 c1 组成

提问

问题：没有索引的时候，数据库是查找数据的

- 数据页的物理存储结构：数据页之间是组成双向链表的，数据页内部的数据行是组成单向链表的，每个数据页内根据主键做了一个页目录
- 然后一般来说，在没有索引的情况下，所有的数据查询，其实在物理层面都是全表扫

描，依次扫描每个数据页内部的每个数据行。

- 但是这个速度非常慢，所以一般肯定是不能让查询走全表扫描的。
- 因此正在在数据库中的查询，必须要用索引来加速查询的执行。

问题：用户记录和目录项记录有什么不同

- 目录项记录的 record_type=1，用户记录的 record_type=0
- 目录项记录只有主键值和页编号两个列，用户记录的列是用户自己定义的，可能有很多，另外还有 InnoDB 自己添加的隐藏列
- 目录项记录的 min_rec_flag=1，用户记录的 min_rec_flag=0

除了上面几点外，这两者就没有差别了：

- 它们用到都是数据页，页的组成结构也是一样的；
- 都会为主键值生成**页目录**，这样就可以按照主键值快速查找了

小结

- InnoDB 存储印象的索引是一颗 B+树，完整的用户记录都存储在 B+树第 0 层的叶子节点，其他层次的节点都属于内节点，内节点存储的是目录项记录。
- InnoDB 的索引分为两种：
 - 聚簇索引：以主键值大小作为页和记录的排序规则，在叶子节点处存储的记录包含了表中所有列
 - 二级索引：以索引列的大小为页和记录的排序规则，在叶子节点处存储的记录包含索引列 + 主键
- 每个索引都对应一颗 B+树
 - InnoDB 存储引擎会为主键自动建立聚簇索引
 - 我们可以为感兴趣的列建立二级索引。

- 如果向通过二级索引查找完整的用户记录，需要执行回表操作，也就是在通过二级索引找到主键值之后，再到聚簇索引中查找完整的回表记录
- B+中的每层记录都是按照索引列的值从小到大的顺序排序组成的双向链表，而且每个表每记录都是按照索引列顺序形成单向链表
- 通过索引查找记录时，是从 B+树的跟节点开始一层一层往下搜索的，由于每个页面中的记录都划分成了若干个组，每个组中的索引列值最大的记录在页内的偏移量都被当做槽依次存放到页目录中，这样就可以二分快速定位了

MySQL 调优之 innodb_buffer_pool_size 大小设置

<https://blog.csdn.net/sunny05296/article/details/78916775>

MySQL 调优之 innodb_buffer_pool_size 大小设置

相关查看命令

```
sql> show global variables like 'innodb_buffer_pool_size';
sql> show global status like 'Innodb_buffer_pool_pages_data';
sql> show global status like 'Innodb_page_size';
```

或

```
sql> use mysql;
sql> select @@innodb_buffer_pool_size;
```

....

```
MariaDB [(none)]> show global variables like 'innodb_buffer_pool_size';
```

Variable_name	Value

```
+-----+  
| innodb_buffer_pool_size | 268435456 |  
+-----+
```

1 row in set (0.00 sec)

```
MariaDB [(none)]> show global status like 'Innodb_buffer_pool_pages_data';
```

```
+-----+-----+  
| Variable_name          | Value   |  
+-----+-----+  
| Innodb_buffer_pool_pages_data | 6082    |  
+-----+-----+
```

1 row in set (0.00 sec)

```
MariaDB [(none)]> show global status like 'Innodb_buffer_pool_pages_total';
```

```
+-----+-----+  
| Variable_name          | Value   |  
+-----+-----+  
| Innodb_buffer_pool_pages_total | 16383  |  
+-----+-----+
```

1 row in set (0.00 sec)

```
MariaDB [(none)]> show global status like 'Innodb_page_size';
```

```
+-----+-----+  
| Variable_name          | Value   |  
+-----+-----+
```

```
| Innodb_page_size | 16384 |
```

```
+-----+-----+
```

```
1 row in set (0.00 sec)
```

```
MariaDB [(none)]>
```

官方对这个几个参数的解释：

Innodb_buffer_pool_pages_data

The number of pages in the InnoDB buffer pool containing data. The number includes both dirty and clean pages.

Innodb **buffer** pool 缓存池中包含数据的页的数目，包括脏页。单位是 page。

Innodb_buffer_pool_pages_total

The total size of the InnoDB buffer pool, in pages.

innodb buffer pool 的页总数目。单位是 page。

Innodb_page_size

InnoDB page size (default 16KB). Many values are counted in pages; the page size enables them to be easily converted to bytes
编译的 InnoDB 页大小 (默认 16KB)。

调优参考计算方法：

`val = Innodb_buffer_pool_pages_data / Innodb_buffer_pool_pages_total * 100%`

`val > 95%` 则考虑增大 innodb_buffer_pool_size，建议使用物理内存的 75%

`val < 95%` 则考虑减小 innodb_buffer_pool_size，建议设置为：

`Innodb_buffer_pool_pages_data * Innodb_page_size * 1.05 / (1024*1024*1024)G`

注意：

my.cnf 配置文件修改参数时支持: innodb_buffer_pool_size = 2G , 也支持:

innodb_buffer_pool_size = 2147483648, 既支持单位字节、也支持单位 G 或 M。

但 session 中修改仅支持单位字节的方式、不支持 G、M 单位方式，即仅支持:

innodb_buffer_pool_size = 2147483648.

设置命令: set global innodb_buffer_pool_size = 2097152; //缓冲池字节大小, 单位

kb, 如果不设置, 默认为 128M

设置要根据自己的实际情况来设置, 如果设置的值不在合理的范围内, 并不是设置越大越

好, 可能设置的数值太大体现不出优化效果, 反而造成系统的 swap 空间被占用, 导致操

作系统变慢, 降低 sql 查询性能。

修改配置文件的调整方法, 修改 my.cnf 配置:

innodb_buffer_pool_size = 2147483648 #设置 2G

innodb_buffer_pool_size = 2G #设置 2G

innodb_buffer_pool_size = 500M #设置 500M

MySQL5.7 及以后版本, 改参数时动态的, 修改后, 无需重启 MySQL, 但是低版本, 静

态的, 修改后, 需要重启 MySQL。

MySQL 的 Buffer Pool 的 free 链表、flush 链表、LRU 链表

<https://blog.csdn.net/zht245648124/article/details/127030881>

1、数据库启动的时候, 是如何初始化 Buffer Pool 的呢?

前面我们已经讲过 MySQL 数据库的 Buffer Pool 到底长成什么样, 其实简单点说,

Buffer Pool 里面就是会包含很多个缓存页, 同时每个缓存页还有一个描述数据, 也可以

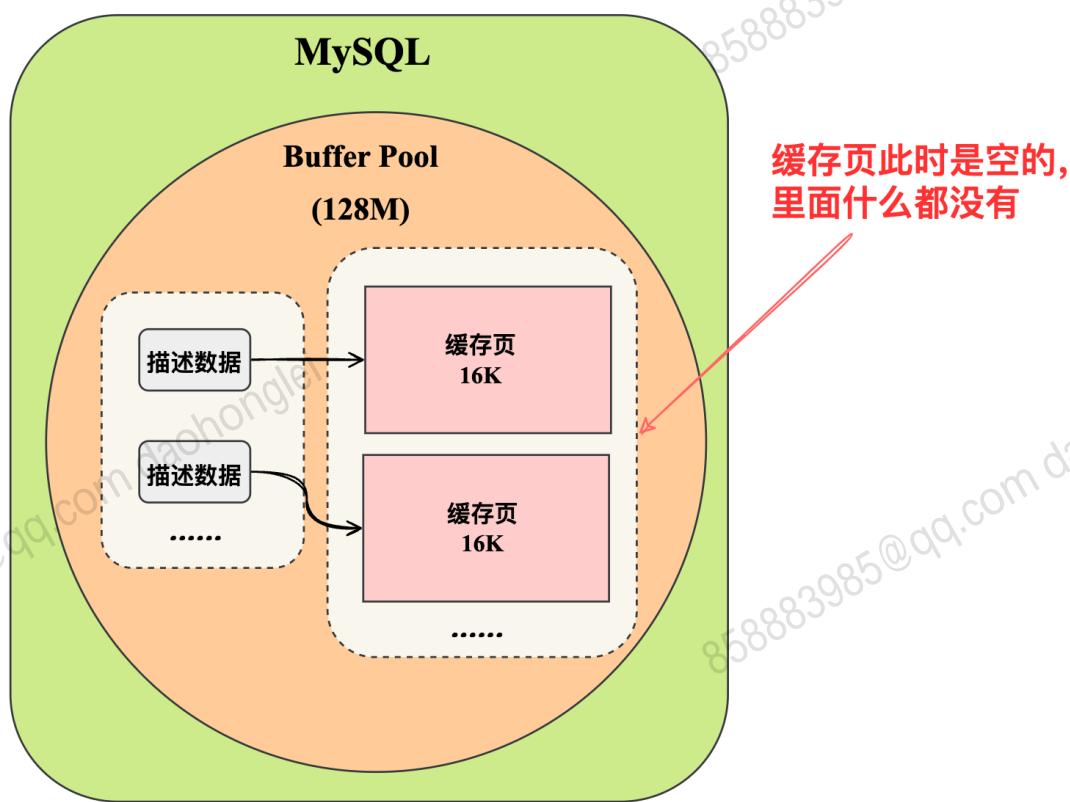
叫做是控制数据, 但是一般大家都叫做描述数据, 或者缓存页的元数据。

那么在数据库启动的时候，他是如何初始化 Buffer Pool 的呢？

其实这个也很简单，数据库只要一启动，就会按照你设置的 Buffer Pool 大小，稍微再加大一点，去找操作系统申请一块内存区域，作为 Buffer Pool 的内存区域。

然后当内存区域申请完毕之后，数据库就会按照默认的缓存页的 16KB 的大小以及对应的 800 个字节左右的描述数据的大小，在 Buffer Pool 中划分出来一个一个的缓存页和一个一个的他们对应的描述数据。

然后当数据库把 Buffer Pool 划分完毕之后，看起来就是之前我们看到的那张图了，如下图所示。



CSDN @zht245648124

只不过这个时候，Buffer Pool 中的一个一个的缓存页都是空的，里面什么都没有，要等数据库运行起来之后，当我们要对数据执行增删改查的操作的时候，才会把数据对应的页从磁盘文件里读取出来，放入 Buffer Pool 中的缓存页中。

2、Buffer Pool 的 free 链表

2.1 我们怎么知道哪些缓存页是空闲的呢？

接着来看下一个问题，当数据库运行起来之后，我们的系统肯定会不停的执行增删改查的

操作，此时就需要不停的从磁盘上读取一个一个的数据页放入 Buffer Pool 中的对应的缓

存页里去，把数据缓存起来，那么以后就可以在内存里对这个数据执行增删改查了。

但是此时在从磁盘上读取数据页放入 Buffer Pool 中的缓存页的时候，必然涉及到一个问

题，那就是哪些缓存页是空闲的？

因为默认情况下磁盘上的数据页和缓存页是一一对应起来的，都是 16KB，一个数据页对

应一个缓存页。数据页只能加载到空闲的缓存页里，所以 MySql 必须要知道 Buffer Pool

中哪些缓存页是空闲的状态？

MySQL 数据库会为 Buffer Pool 设计了一个 free 链表，是一个双向链表数据结构，这个

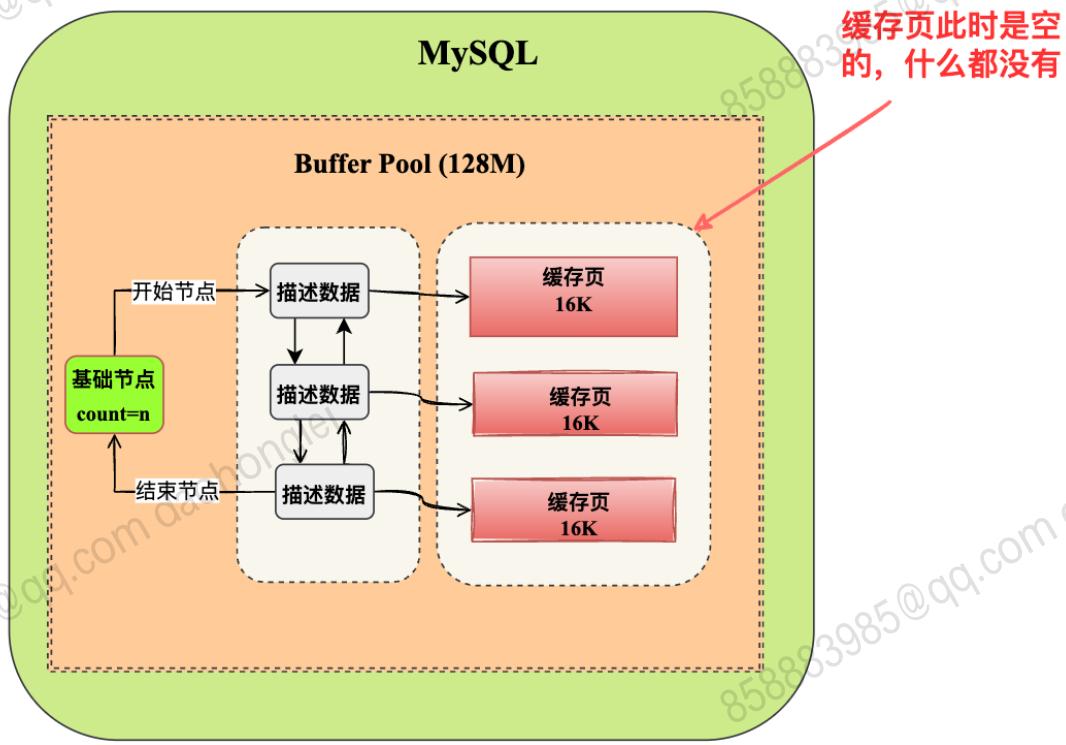
free 链表里，每个节点就是一个空闲的缓存页的描述数据块的地址，也就是说，只要你一

个缓存页是空闲的，那么它的描述数据块就会被放入这个 free 链表中。

刚开始数据库启动的时候，所有的缓存页都是空闲的，因为此时可能是一个空的数据库，

一条数据都没有，所以此时所有缓存页的描述数据块，都会被放入这个 free 链表中。

假设数据此时总共有 3 个数据页，那么它的 free 链表如下图所示：



CSDN @zht245648124

大家可以看到上面出现了一个 free 链表，这个 free 链表里面就是各个缓存页的描述数据块，只要缓存页是空闲的，那么他们对应的描述数据块就会加入到这个 free 链表中，每个节点都会双向链接自己的前后节点，组成一个双向链表。

除此之外，这个 free 链表有一个基础节点，它会引用链表的头节点和尾节点，里面还存储了链表中有多少个描述数据块的节点，也就是有多少个空闲的缓存页。

这里要给大家讲明白一点，这个 free 链表，其实就是咱们所学的数据结构中的双向链表。它本身其实就是在 Buffer Pool 里的描述数据块组成的，你可以认为是每个描述数据块里都有两个指针，一个是 free_pre，一个是 free_next，分别指向自己的上一个 free 链表的节点，以及下一个 free 链表的节点。

通过 Buffer Pool 中的描述数据块的 free_pre 和 free_next 两个指针，就可以把所有的描述数据块串成一个 free 链表。上面为了画图需要，假如数据库总共有 3 个空闲的描述数据

块，展示他们之间的指针引用关系。

对于 free 链表而言，只有一个基础节点是不属于 Buffer Pool 的，它是 40 字节大小的一个节点，里面就存放了 free 链表的头节点的地址，尾节点的地址，还有 free 链表里当前有多少个节点。

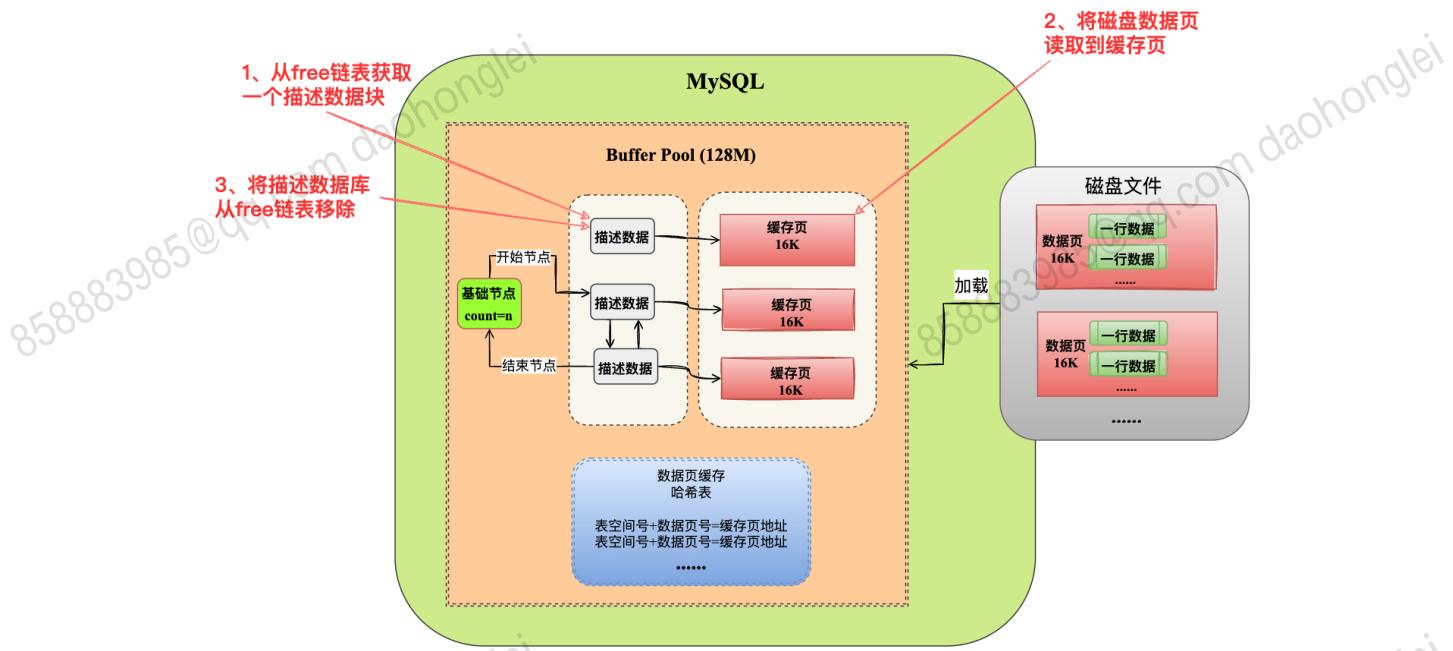
2.2 如何将磁盘上的页读取到 Buffer Pool 的缓存页中去？

好了，有了 free 链表，当需要把磁盘上的数据页读取到 Buffer Pool 中的缓存页里去的时候，是怎样一个操作过程呢？

其实有了 free 链表之后，这个操作就很简单了。

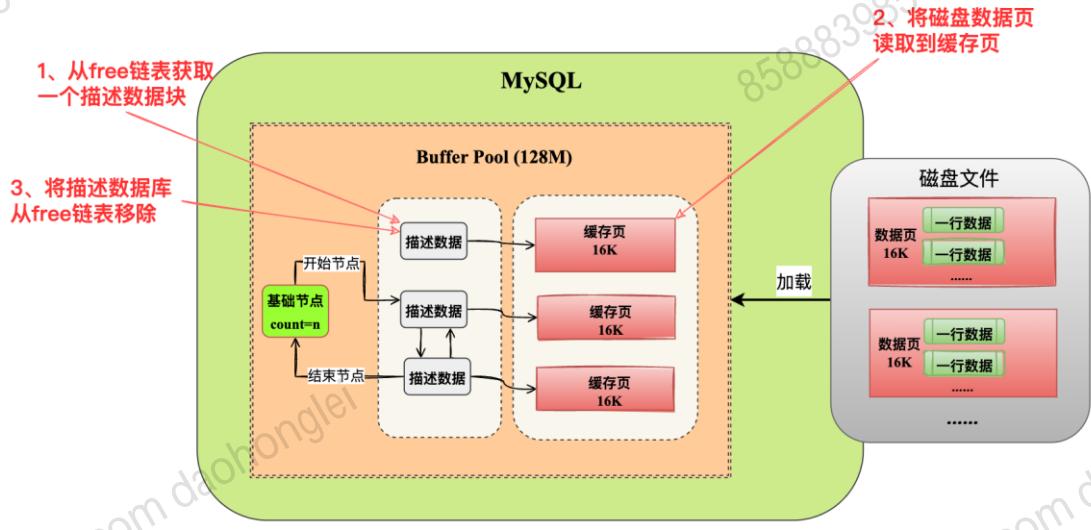
首先，我们需要从 free 链表里获取一个描述数据块，然后就可以获取到这个描述数据块对应的空闲缓存页；

接着我们就可以把磁盘上的数据页读取到对应的缓存页里去，同时把相关的一些描述数据写入缓存页的描述数据块里去，比如这个数据页所属的表空间之类的信息，如下图所示：



最后把那个描述数据块从 free 链表里去除就可以了。移除后，链表变化如下图所示：

CSDN @zht245648124



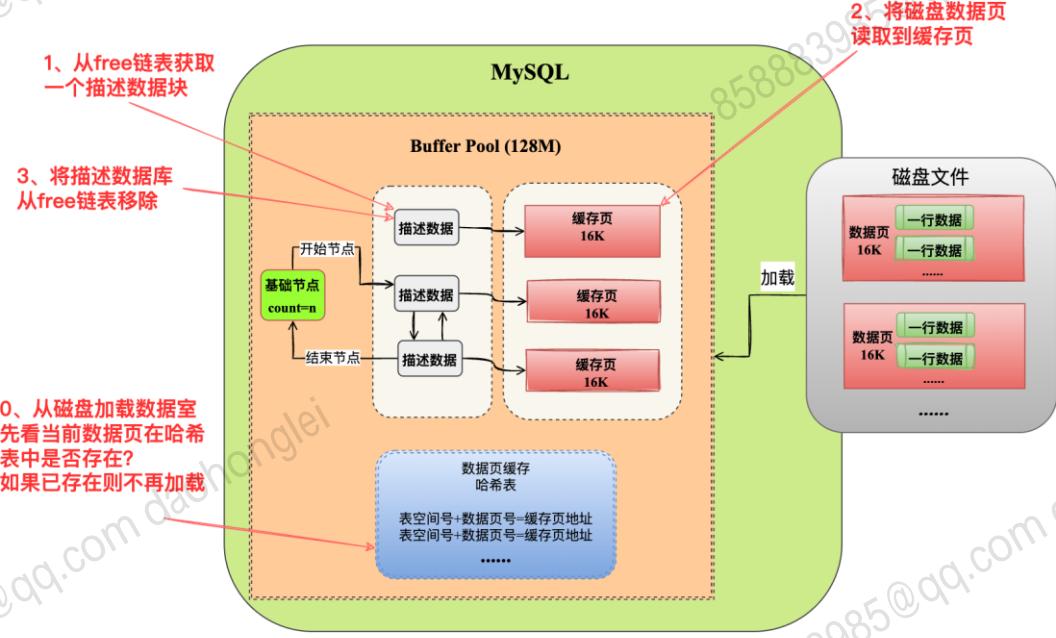
2.3 MySQL 怎么知道某个数据页有没有被缓存？

接着我们来看下一个问题：MySQL 怎么知道某个数据页有没有被缓存？

我们在执行增删改查的时候，肯定是先看看这个数据页有没有被缓存，如果没被缓存就走

上面的逻辑，从 free 链表中找到一个空闲的缓存页，从磁盘上读取数据页写入缓存页，写入描述数据，从 free 链表中移除这个描述数据块。

但是如果数据页已经被缓存了，那么就会直接使用了。所以其实数据库还会有一个哈希表数据结构，他会用表空间号+数据页号，作为一个 key，然后缓存页的地址作为 value。当你你要使用一个数据页的时候，通过“表空间号+数据页号”作为 key 去这个哈希表里查一下，如果没有就读取数据页，如果已经有了，就说明数据页已经被缓存了，如下图所示：



CSDN @zht245648124

MySQL 引入了一个数据页缓存哈希表的结构，也就是说，每次你读取一个数据页到缓存之后，都会在这个哈希表中写入一个 key-value 对，key 就是表空间号+数据页号，value 就是缓存页的地址，那么下次如果你再使用这个数据页，就可以从哈希表里直接读取出来它已经被放入一个缓存页了。

{表空间号+数据页号 : 控制块的地址}

3、Buffer Pool 的 flush 链表

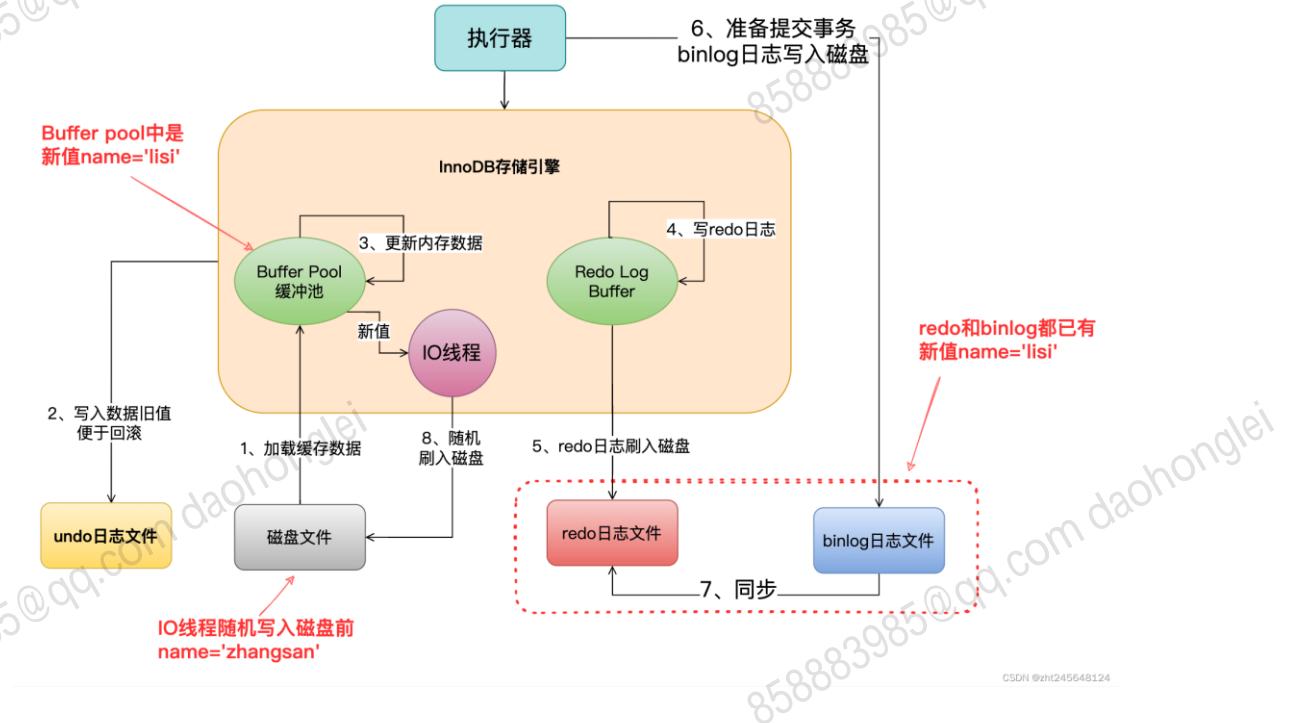
3.1 为什么会有脏数据页？

前面的章节我们也讲过，你要更新的数据页都会在 Buffer Pool 的缓存页里，供你在内存中直接执行增删改的操作。比如前面讲的如下更新操作：

```
update users set name='lisi' where id=2
```

mysql 肯定会去更新 Buffer Pool 的缓存页中的数据，此时一旦更新了缓存页中的数据，那么缓存页里的数据和磁盘上的数据页里的数据，是不是就不一致了？

这个时候，我们就说缓存页是脏数据，脏页，如下图。这个前面已经讲的很明白，这里不做过多重复讲解。



3.2 脏页怎么刷回磁盘呢？

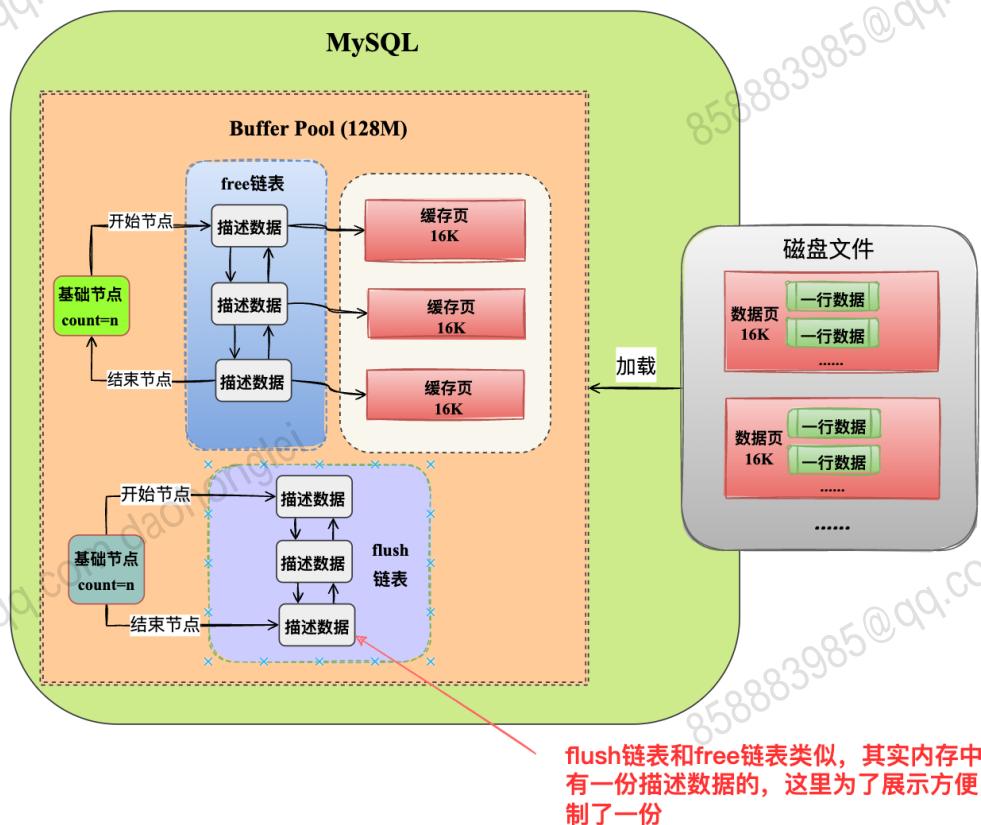
我们都是知道一点的，最终这些在内存里更新的脏页的数据，都是要被刷新回磁盘文件的。

但是这里就有一个问题了，不可能所有的缓存页都刷回磁盘的，因为有的缓存页可能是因为查询的时候被读取到 Buffer Pool 里去的，可能根本没修改过！那 mysql 是怎么判断哪些缓存页需要刷回磁盘的呢？

所以 mysql 数据库在这里引入了另外一个跟 free 链表类似的 flush 链表。

flush 链表也是由描述数据块组成，凡是被修改过的缓存页，都会把它的描述数据块加入到 flush 链表中去。flush 的意思就是这些都是脏页，后续都是要 flush 刷新到磁盘上去的。

所以 flush 链表的结构如下图所示，跟 free 链表几乎是一样的，只不过是脏页对应的描述数据块组成的链表，这里就不详细介绍介绍了。



CSDN @zht245648124

好多人看上图认为 flush 链表是把描述数据库又复制了一份，这是错误的！

flush 链表和 free 链表类似，这里方便展示，单独复制一份描述数据，只是方便大家理解。

其实内存中仅有一份描述数据。

3.3 flush 链表的总结

讲到这里，大家都应该明白了，当 mysql 更新缓存页的时候，通过变换缓存页中的描述数据块的 flush 链表的指针，就可以把脏页的描述数据块组成一个双向链表，也就是 flush 链表，而且 flush 链表的基础节点会指向起始节点和尾巴节点。

通过这个 flush 链表，就可以记录下来哪些缓存页是脏页了！

4、Buffer Pool 的 LRU 链表

4.1 当 Buffer Pool 中缓存页不够了怎么办？

之前我们已经给大家讲解了 Buffer Pool 中的缓存页的划分，包括 free 链表的使用，然后磁盘上的数据页是如何加载到缓存页里去的，包括对缓存页修改之后，形成内存里的脏页，

flush 链表是如何用来记载脏数据页的。

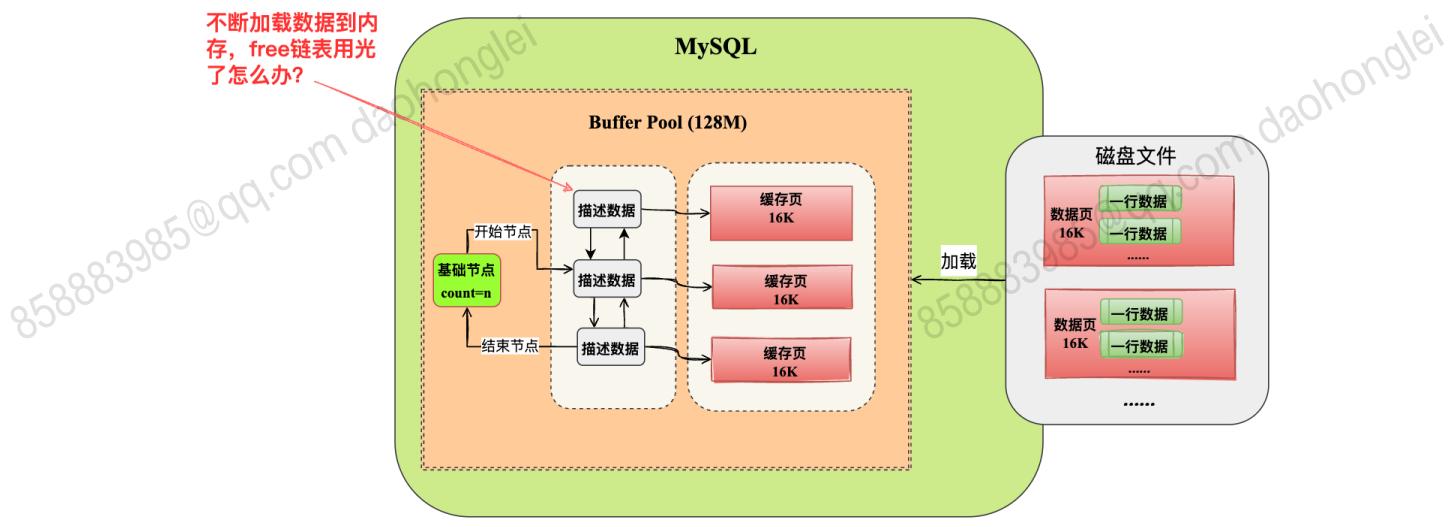
我们接着来分析 Buffer Pool 的工作原理，首先来思考一个问题，当你要执行 CRUD 操作的时候，无论是查询数据，还是修改数据，实际上都会把磁盘上的数据页加载到缓存页里来，这个大家都是没有问题的吧？

那么在加载数据到缓存页的时候，必然是要加载到空闲的缓存页里去的，所以必须要从 free 链表中找一个空闲的缓存页，然后把磁盘上的数据页加载到那个空闲的缓存页里去。

那么大家通过之前的学习肯定都知道了，随着你不停的把磁盘上的数据页加载到空闲的缓存页里去，free 链表中的空闲缓存页是不是会越来越少？因为只要你把一个数据页加载到一个空闲缓存页里去，free 链表中就会减少一个空闲缓存页。

所以，当你不停的把磁盘上的数据页加载到空闲缓存页里去，free 链表中不停的移除空闲缓存页，迟早有那么一瞬间，你会发现 free 链表中已经没有空闲缓存页了。

这个时候，当你还要加载数据页到一个空闲缓存页的时候，怎么办呢？如下图所示：



4.2 如果要淘汰掉一些缓存数据，淘汰谁？

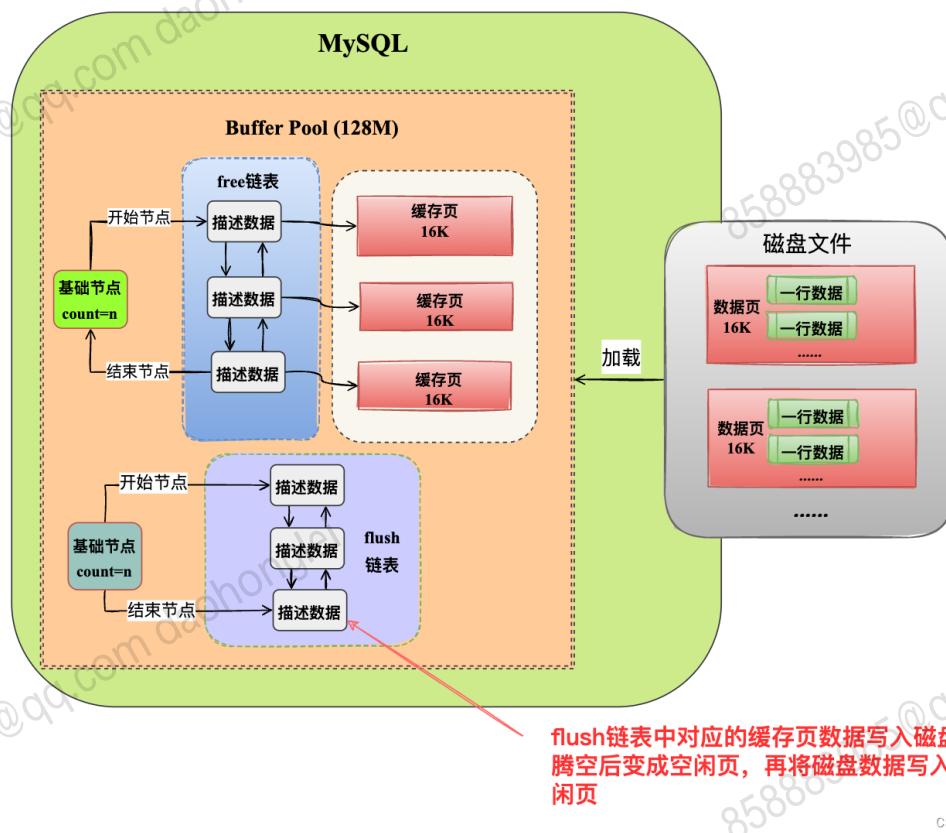
针对上述的问题，大家来思考下一个问题，如果所有的缓存页都被塞了数据了，此时无法

从磁盘上加载新的数据页到缓存页里去了，那么此时你只有一个办法，就是淘汰掉一些缓存页。

那什么叫淘汰缓存页呢？

顾名思义，你必须把一个缓存页里被修改过的数据，给他刷到磁盘上的数据页里去，然后这个缓存页就可以清空了，让他重新变成一个空闲的缓存页。

接着你再把磁盘上你需要的新的数据页加载到这个腾出来的空闲缓存页中去，如下图。



那么下一个问题来了，如果要把一个缓存页里的数据刷入磁盘，腾出来一个空闲缓存页，

那么应该把哪个缓存页的数据给刷入磁盘呢？

4.3 缓存命中率概念

要解答这个问题，我们就得引入一个缓存命中率的概念。

假设现在有两个缓存页，一个缓存页的数据，经常会被修改和查询，比如在 100 次请求中，有 30 次都是在查询和修改这个缓存页里的数据。那么此时我们可以说这种情况下，缓存

命中率很高，为什么呢？因为 100 次请求中，30 次都可以操作缓存，不需要从磁盘加载数据，这个缓存命中率就比较高了。

另外一个缓存页里的数据，就是刚从磁盘加载到缓存页之后，被修改和查询过 1 次，之后 100 次请求中没有一次是修改和查询这个缓存页的数据的，那么此时我们就说缓存命中率有点低，因为大部分请求可能还需要走磁盘查询数据，他们要操作的数据不在缓存中。所以针对上述两个缓存页，假设此时让你做一个抉择，要把其中缓存页的数据刷入到磁盘去，腾出来一个空闲的缓存页，此时你会选择谁？

那还用想么，当然是选择第二个缓存页刷入磁盘中了！因为第二个缓存页，压根儿就没什人来使用他里面的数据，结果这些数据还空占据了一个缓存页，这不是白白浪费缓存页吗？

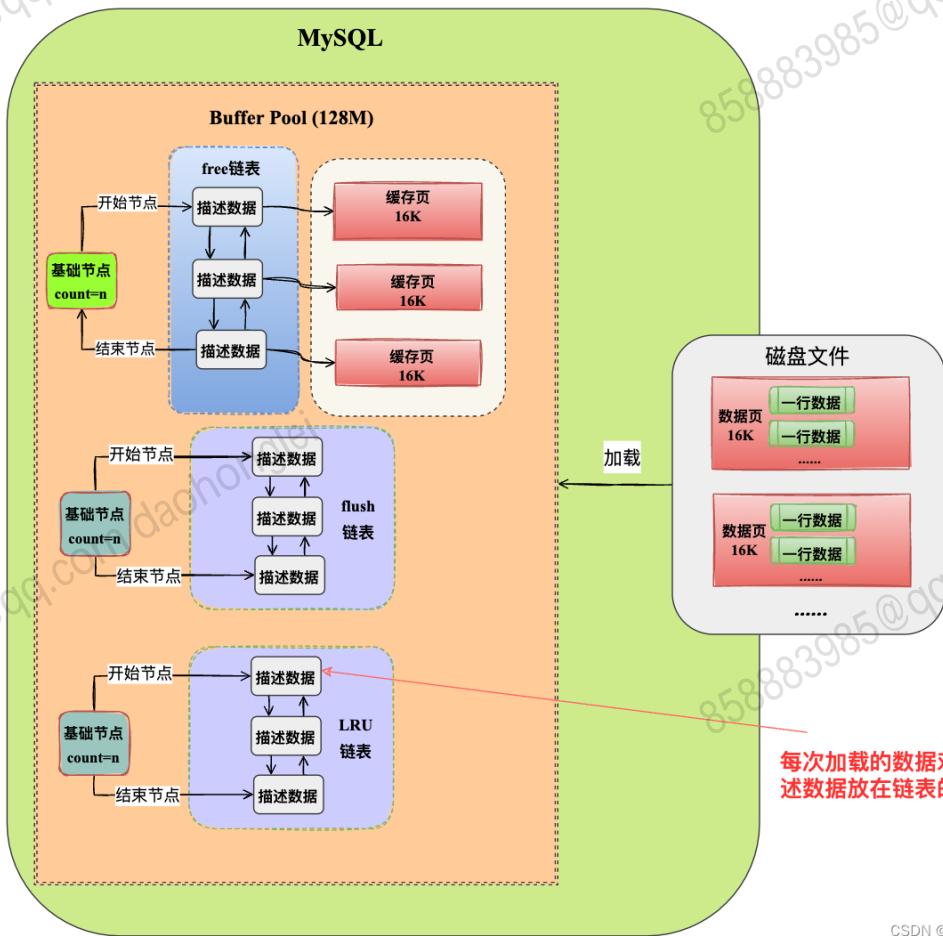
4.4 引入 LRU 链表来判断哪些缓存页是不常用的

接着我们就要解决下一个问题了，就是你怎么知道哪些缓存页经常被访问，哪些缓存页很少被访问？

此时就要引入一个新的 LRU 链表了，这个所谓的 LRU 就是 Least Recently Used，最近最少使用的意思。

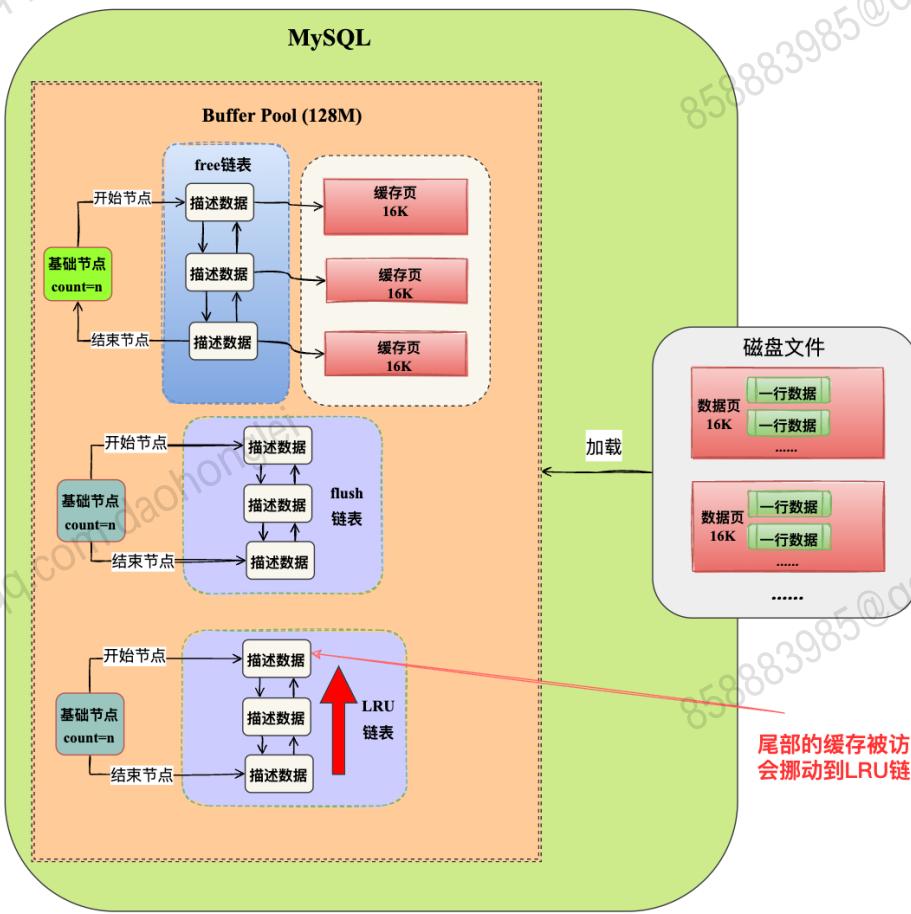
通过这个 LRU 链表，我们可以知道哪些缓存页是最近最少被使用的，那么当你缓存页需要腾出来一个刷入磁盘的时候，不就可以选择那个 LRU 链表中最近最少被使用的缓存页了么？这个 LRU 链表大致是怎么个工作原理呢？

简单来说，我们看下图，假设我们从磁盘加载一个数据页到缓存页的时候，就把这个缓存页的描述数据块放到 LRU 链表头部去，那么只要有数据的缓存页，他都会在 LRU 里了，而且最近被加载数据的缓存页，都会放到 LRU 链表的头部去。



CSDN @zht245648124

然后假设某个缓存页的描述数据块本来在 LRU 链表的尾部，后续你只要查询或者修改了这个缓存页的数据，也要把这个缓存页挪动到 LRU 链表的头部去，也就是说最近被访问过的缓存页，一定在 LRU 链表的头部，如下图所示：



CSDN @zht245648124

那么这样的话，当你的缓存页没有一个空闲的时候，你是不是要找出来那个最近最少被访问的缓存页去刷入磁盘？此时你就直接在 LRU 链表的尾部找到一个缓存页，它一定是最近最少被访问的那个缓存页！

然后你就把 LRU 链表尾部的那个缓存页刷入磁盘中，然后把你需要的磁盘数据页加载到腾出来的空闲缓存页中就可以了！

mysql 的 LRU 队列

MySQL 的缓冲池 (`innodb_buffer_pool_instances`, `innodb_buffer_pool_size`) 是通过 LRU 算法来进行管理的。即最频繁使用的页在 LRU 列表的最前端，而最少使用的页在 LRU 列表的尾端。当缓冲池不能存放新读取到的页时，将首先释放 LRU 列表尾端的页。

在 INNODB 引擎中，缓冲池页的大小默认为 16KB。

在 INNODB 引擎中，有一个 midpoint 位置。新读取到的页，虽然是最新访问的页，但并不直接放到 LRU 队列的首部，而是放到 LRU 列表的 midpoint 位置。

midpoint 位置是通过参数 innodb_old_blocks_pct 来控制的。表示新读取的页插入到 LRU 列表尾端的 37% 的位置。在 INNODB 中，把 midpoint 之后的列表称为 old 列表，之前的列表称为 new 列表。可以简单的理解为 new 列表中的页都是活跃的热点数据。

为什么不把读入的页直接放入 LRU 队列的首部呢？是因为如此做可能会把大量频繁使用的数据页换出 LRU 队列，带来再次磁盘读取的 IO 压力。

```
show variables like 'innodb_old_blocks_pct';  
  
mysql> show variables like 'innodb_%blocks_pct'\G;  
***** 1. row *****  
Variable_name: innodb_old_blocks_pct  
Value: 37
```

CSDN @请叫我曾阿牛

数据页在 midpoint 位置，什么情况下会进入 NEW 列表呢？是通过参数 innodb_old_blocks_time 来控制的。该参数表示，页读取到 midpoint 位置后需要等多久才会被加入到 LRU 队列的热端。

```
show variables like 'innodb_old_blocks_time';  
  
mysql> show variables like 'innodb_%blocks_time'\G;  
***** 1. row *****  
Variable_name: innodb_old_blocks_time  
Value: 1000
```

CSDN @请叫我曾阿牛

重新调整两个参数的值：

```
mysql> set global innodb_old_blocks_pct=20;  
  
mysql> set global innodb_old_blocks_time=500;
```

数据页在 LRU 队列上的移动规则：

当数据库刚启动的时候，LRU 列表是空的，所有的页都放在 FREE 列表中。当需要重

缓冲池中分页时，首先从 FREE 列表中查找是否有可用的空闲页，如果有则将该页从 FREE 列表中删除，放入 LRU 列表中；如果没有，则根据 LRU 算法，淘汰 LRU 列表尾部的页，将该内存空间分配给新的页。

当页从 LRU 列表的 OLD 部分加入到 NEW 部分时，称此时发生的操作为 page made young；如果因为 innodb_old_blocks_time 的设置导致页没有从 old 部分移动到 new 部分的操作，称为 page not made young。

通过命令 show engine innodb status\G;可以查看 LRU 队列的使用情况：

```
-----  
BUFFER POOL AND MEMORY  
-----  
  
Total large memory allocated 137428992  
  
Dictionary memory allocated 140621  
  
Buffer pool size 8192  
  
Free buffers 7912  
  
Database pages 280  
  
Old database pages 0  
  
Modified db pages 0  
  
Pending reads 0  
  
Pending writes: LRU 0, flush list 0, single page 0  
  
Pages made young 0, not young 0  
  
0.00 youngs/s, 0.00 non-youngs/s  
  
Pages read 245, created 35, written 40
```

0.00 reads/s, 0.00 creates/s, 0.00 writes/s

Buffer pool hit rate 988 / 1000, young-making rate 0 / 1000 not 0 / 1000

Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead

0.00/s

LRU len: 280, unzip_LRU len: 0

I/O sum[0]:cur[0], unzip sum[0]:cur[0]

Buffer pool size: 表示当前缓冲池中内存页的数量，内存池的大小=Buffer pool

size*16KB

Free buffers: 表示当前 FREE 列表中页的数量；

Database pages: LRU 列表中页的数量；

Modified db pages: 显示了脏页的数量；

Pages made young 0, not young 0 : 表示是否发生了页在 LRU 队列上的移动；

0.00 youngs/s, 0.00 non-youngs/s: 表示每秒两类操作发生的次数；

Buffer pool hit rate: 表示缓冲池的命中率，正常情况下命中率如果低于 95%，则需要观察是否因为全表扫描引起了 LRU 队列被污染的问题

Per second averages calculated from the last 18 seconds: 表示 show engine

innodb status\G;打印的是过去 18 秒内的数据库状态。

观察缓冲池的运行状态：

```
select pool_id,hit_rate,pages_made_young,pages_not_made_young from  
information_schema.innodb_buffer_pool_stats\G;
```

```
mysql> select pool_id,hit_rate,pages_made_young,pages_not_made_young from information_schema.innodb_buffer_pool_stats\G;  
***** 1. row *****  
    pool_id: 0  
    hit_rate: 0  
  pages_made_young: 0  
pages_not_made_young: 0  
1 row in set (0.00 sec)
```

CSDN @请叫我曾阿牛

查看 LRU 列表中每个页的具体信息：

```
select table_name,space,page_number,page_type from  
information_schema.innodb_buffer_page_lru where space=1;
```

```
mysql> select table_name,space,page_number,page_type from information_schema.innodb_buffer_page_lru where space=1;  
Empty set (0.00 sec) CSDN @请叫我曾阿牛
```

脏页的刷新：

脏页既存在于 LRU 列表中，也存在于 Flush 列表中。LRU 列表用来管理缓冲池中页的可用性，Flush 列表用来管理将页刷新回磁盘，二者互不影响。

预读失效

先来说说 MySQL 的预读机制。程序是有空间局部性的，靠近当前被访问数据的数据，在未来很大概率会被访问到。所以，MySQL 在加载数据页时，会提前把它相邻的数据页一并加载进来，目的是为了减少磁盘 IO。

但是可能这些被提前加载进来的数据页，并没有被访问，相当于这个预读是白做了，这就是预读失效。

如果使用简单的 LRU 算法，就会把预读页放到 LRU 链表头部，而当 Buffer Pool 空间不够的时候，还需要把末尾的页淘汰掉。

如果这些预读页如果一直不会被访问到，就会出现一个很奇怪的问题，不会被访问的预读页却占用了 LRU 链表前排的位置，而末尾淘汰的页，可能是频繁访问的页，这样就大大降低了缓存命中率。

怎么解决预读失效而导致缓存命中率降低的问题？

我们不能因为害怕预读失效，而将预读机制去掉，大部分情况下，局部性原理还是成立的。要避免预读失效带来影响，最好就是让预读的页停留在 Buffer Pool 里的时间要尽可能的

短，让真正被访问的页才移动到 LRU 链表的头部，从而保证真正被读取的热数据留在

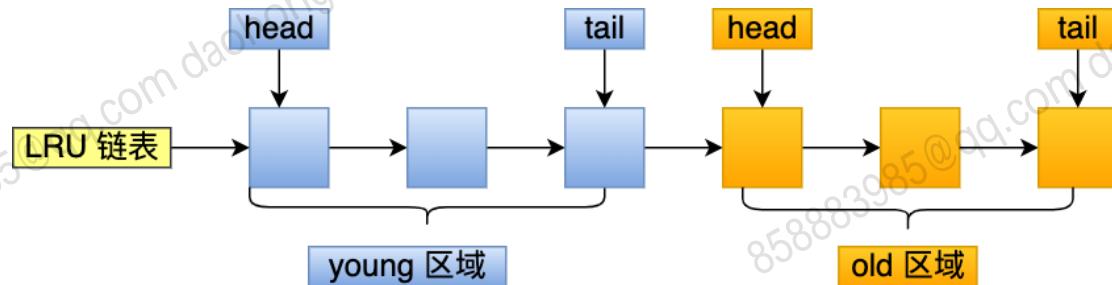
Buffer Pool 里的时间尽可能长。

那到底怎么才能避免呢？

MySQL 是这样做的，它改进了 LRU 算法，将 LRU 划分了 2 个区域：old 区域和 young

区域。

young 区域在 LRU 链表的前半部分，old 区域则是在后半部分，如下图：



old 区域占整个 LRU 链表长度的比例可以通过 `innodb_old_blocks_pc` 参数来设置，默认

是 37，代表整个 LRU 链表中 young 区域与 old 区域比例是 63:37。

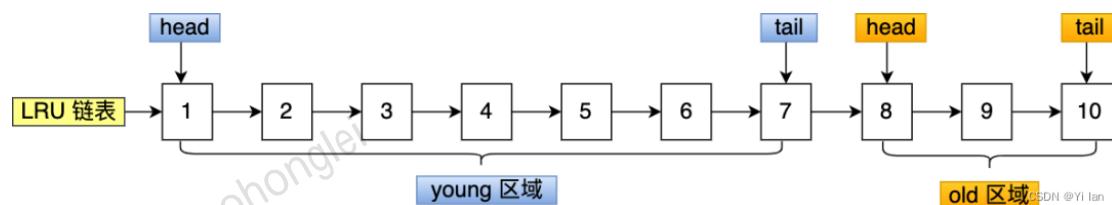
划分这两个区域后，预读的页就只需要加入到 old 区域的头部，当页被真正访问的时候，

才将页插入 young 区域的头部。如果预读的页一直没有被访问，就会从 old 区域移除，

这样就不会影响 young 区域中的热点数据。

例子，假设有一个长度为 10 的 LRU 链表，其中 young 区域占比 70 %，old 区域占比

20 %。如下：



过程说明：

假如我们有一个 15 号页被预读了，这个页号会被插入到 old 区域头部，而 old 区域 10 号页给淘汰

如果 15 号页一直没有被访问到，那么就不会占用 young 区域的位置，而且会给 young 区域的数据更早被淘汰

如果 15 号页被预读后，立刻被访问了，那么就会将它插入到 young 区域的头部，young 区域末尾的页（7 号），会被挤到 old 区域，作为 old 区域的头部，这个过程并不会有页被淘汰。

Buffer Pool 污染

当某一个 SQL 语句扫描了大量的数据时，在 Buffer Pool 空间比较有限的情况下，可能会将 Buffer Pool 里的所有页都替换出去，导致大量热数据被淘汰了，等这些热数据又被再次访问的时候，由于缓存未命中，就会产生大量的磁盘 IO，MySQL 性能就会急剧下降，这个过程被称为 Buffer Pool 污染。

注意，Buffer Pool 污染并不只是查询语句查询出了大量的数据才出现的问题，即使查询出来的结果集很小，也会造成 Buffer Pool 污染。

比如，在一个数据量非常大的表，执行了这条语句：

```
select * from t_user where name like "%ian%";
```

可能这个查询出来的结果就几条记录，但是由于这条语句会发生索引失效，所以这个查询过程是全表扫描的，接着会发生如下的过程：

- 从磁盘读到的页加入到 LRU 链表的 old 区域头部；
- 当从页里读取行记录时，也就是页被访问的时候，就要将该页放到 young 区域头部；
- 接下来拿行记录的 name 字段和字符串 ian 进行模糊匹配，如果符合条件，就加入到

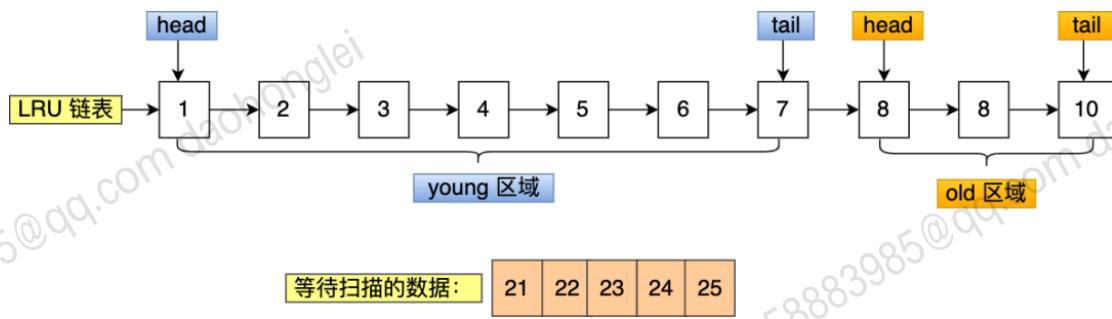
结果集里；

- 如此往复，直到扫描完表中的所有记录。

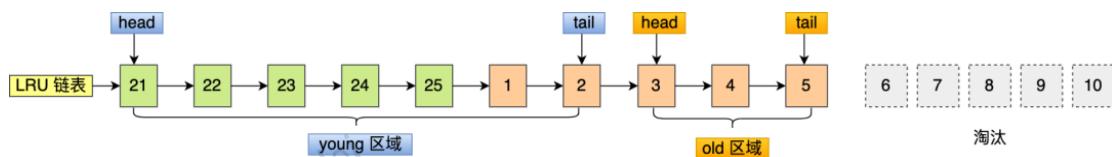
经过这一番折腾，原本 young 区域的热点数据都会被替换掉。

举个例子，假设需要批量扫描：21, 22, 23, 24, 25 这五个页，这些页都会被逐一访问

(读取页里的记录)。



在批量访问这些数据的时候，会被逐一插入到 young 区域头部。



可以看到，原本在 young 区域的热点数据 6 和 7 号页都被淘汰了，这就是 Buffer Pool 污染的问题。

怎么解决出现 Buffer Pool 污染而导致缓存命中率下降的问题？

像前面这种全表扫描的查询，很多缓冲页其实只会被访问一次，但是它却只因为被访问了一次而进入到 young 区域，从而导致热点数据被替换了。

LRU 链表中 young 区域就是热点数据，只要我们提高进入到 young 区域的门槛，就能有效地保证 young 区域里的热点数据不会被替换了。

MySQL 是这样做的，进入到 young 区域条件增加了一个停留在 old 区域的时间判断。

具体是这样做的，在对某个处在 old 区域的缓存页进行第一次访问时，就在它对应的控制

块中记录下来这个访问时间：

- 如果后续的访问时间与第一次访问的时间在某个时间间隔内，那么该缓存页就不会被从 old 区域移动到 young 区域的头部；
- 如果后续的访问时间与第一次访问的时间不在某个时间间隔内，那么该缓存页移动到 young 区域的头部；

这个间隔时间是由 `innodb_old_blocks_time` 控制的，默认是 1000 ms。

也就说，只有同时满足「被访问」与「在 old 区域停留时间超过 1 秒」两个条件，才会被插入到 young 区域头部，这样就解决了 Buffer Pool 污染的问题。

另外，MySQL 针对 young 区域其实做了一个优化，为了防止 young 区域节点频繁移动到头部。young 区域前面 1/4 被访问不会移动到链表头部，只有后面的 3/4 被访问了才会。

MySQL 脏页刷盘流程

1. 什么是脏页

InnoDB 更新语句，是先查询到指定记录到内存缓冲区，先写重做日志，然后更新内存缓冲区数据，也并不会立即将数据页刷新到磁盘上，而是放到一个链表（Flush 链表）里。这样就会导致内存数据页和磁盘数据页的数据不一致的情况。这种数据不一致的数据页成为脏页。当脏页写入到磁盘后（flush），数据一致性后称为干净页

2. 关于 Innodb 刷盘的策略

对于数据更新操作，存储引擎会将数据页先加载到内存缓冲池，然后修改内存中该数据页的内容。这样就会产生脏页，脏页需要刷新到磁盘才能保证对数据表的更新

被持久化。但是如果更新一条记录就需要将一个页刷盘一次，则这个开销就有点太大。太多次 IO 操作非常影响性能。所以存储引擎对于内存数据页的修改，不是一有修改就会刷盘，而是达到一定的阈值才会去刷盘。那么就会产生一个问题，如果此时数据库宕机了，则内存的的脏数据页会没有完成刷盘就丢失了。这样就会导致有些更新被丢失了。因此 InnoDB 存储引擎引入了 redo 日志，在数据库对内存数据页的更新后会先将更新日志写入 redo 日志的内存缓冲区中，当 redo 日志的内存缓冲区中的日志写入 redo 日志的磁盘文件上，就表示这个操作完成了持久化。然后存储引擎在合适的时间再将数据脏页刷回磁盘。可能你会问， redo 日志的内存缓冲区中的日志写入 redo 日志的磁盘文件上，也是需要 IO 啊，为什么不直接将数据页写入磁盘呢？**因为 redo 日志写入磁盘文件这个过程时顺序存储的（在磁盘文件的尾部追加即可）**。而 IO 刷盘是随机存储的（需要寻找刷新的磁盘位置）。所以 redo 日志写入磁盘的速度远远快于数据的刷盘。

3. 什么时候会 flush 脏页

- a. 磁盘上的 redo log 日志文件写满时：redo log 大小固定，写完后会循环覆盖写入，写满后当有新内容要写入时，系统必须停止所有的更新操作，将 checkpoint 向前推进到新的位置，但是在推进之前必须将覆盖部分的所有脏页都 flush 到磁盘上
- b. 内存不足需要淘汰数据页。如果淘汰的是脏页，则需要将这些脏页刷盘。
- c. 系统空闲的时候后台会定期 flush 适量的脏页到磁盘。
- d. MySQL 正常关闭时，会把所有脏页都 flush 到磁盘。
- e. 当脏数据页太多时，也会触发将脏数据页刷新回磁盘。该机制可由参数 innodb_max_dirty_pages_pct 控制，比如将其设置为 75，表示，当 Buffer Pool 中的脏数据页达到整体缓存的 75% 时，触发刷新的动作。现实情况是该参数默认值为 0。

以此来禁用 Buffer Pool 早期的刷新行为。

```
show global variables like 'innodb_max_dirty_pages_pct';
```

4. flush 对系统性能的影响

前两种会造成系统性能问题。第三种系统空闲所以不会有性能问题。第四种要关闭自然也不用考虑性能问题。

undo log 的作用

在 mysql 中， undo log 日志的作用主要有两个：

1、提供回滚操作【undo log 实现事务的原子性】

我们在进行数据更新操作的时候，不仅会记录 redo log，还会记录 undo log，如果因为某些原因导致事务回滚，那么这个时候 mysql 就要执行回滚（rollback）操作，利用 undo log 将数据恢复到事务开始之前的状态。

此外，undo log 会产生 redo log，也就是 undo log 的产生会伴随着 redo log 的产生，这是因为 undo log 也需要持久保护。

如我们执行下面一条删除语句：

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

那么此时 undo log 会记录一条对应的 insert 语句【反向操作的语句】，以保证在事务回滚时，将数据还原回去。

再比如我们执行一条 update 语句：

```
update user set name= "李四" where id = 1;  
-- 修改之前 name=张三
```

此时 undo log 会记录一条相反的 update 语句，如下：

```
update user set name = "张三" where id = 1;
```

如果这个修改出现异常，可以使用 undo log 日志来实现回滚操作，以保证事务的一致性。

2、提供多版本控制(mvcc) 【undo log 实现多版本并发控制 (mvcc)】

mvcc，即多版本控制。在 mysql 数据库 innodb 存储引擎中，用 undo log 来实现多版本并发控制(mvcc)。当读取的某一行被其他事务锁定时，它可以从 undo log 中分析出该行记录以前的数据版本是怎样的，从而让用户能够读取到当前事务操作之前的数据【快照读】。

下面解释一下什么是快照读，与之对应的还有一个是---当前读。

快照读：sql 读取的数据是快照版本【可见版本】，也就是历史版本，不用加锁，普通的 select 就是快照读。

当前读：sql 读取的数据是最新版本。通过锁机制来保证读取的数据无法通过其他事务进行修改 update、delete、insert、select ... lock in share mode、select ... for update 都是当前读。

<https://www.cnblogs.com/sxrtb/p/16603155.html>

https://blog.csdn.net/weixin_49238443/article/details/125958607

<https://www.zhihu.com/question/460304592/answer/2407448183>

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。所以我们

先来看看这个三个 point 的概念

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
- 实际还有一个删除 flag 隐藏字段, 既记录被更新或删除并不代表真的删除, 而是删除 flag 变了

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

如上图, 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

undo log 主要分为两种:

- insert undo log

代表事务在 insert 新记录时产生的 undo log, 只在事务回滚时需要, 并且在事务提交后可以被立即丢弃

- update undo log

事务在进行 update 或 delete 时产生的 undo log; 不仅在事务回滚时需要，在快照读时也需要；所以不能随便删除，只有在快速读或事务回滚不涉及该日志时，对应的日志才会被 purge 线程统一清除

purge

- 从前面的分析可以看出，为了实现 InnoDB 的 MVCC 机制，更新或者删除操作都只是设置一下老记录的 deleted_bit，并不真正将过时的记录删除。
- 为了节省磁盘空间，InnoDB 有专门的 purge 线程来清理 deleted_bit 为 true 的记录。为了不影响 MVCC 的正常工作，purge 线程自己也维护了一个 read view（这个 read view 相当于系统中最老活跃事务的 read view）；如果某个记录的 deleted_bit 为 true，并且 DB_TRX_ID 相对于 purge 线程的 read view 可见，那么这条记录一定是可以被安全清除的。

对 MVCC 有帮助的实质是 update undo log , undo log 实际上就是存在 rollback segment 中旧记录链，它的执行流程如下：

一、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

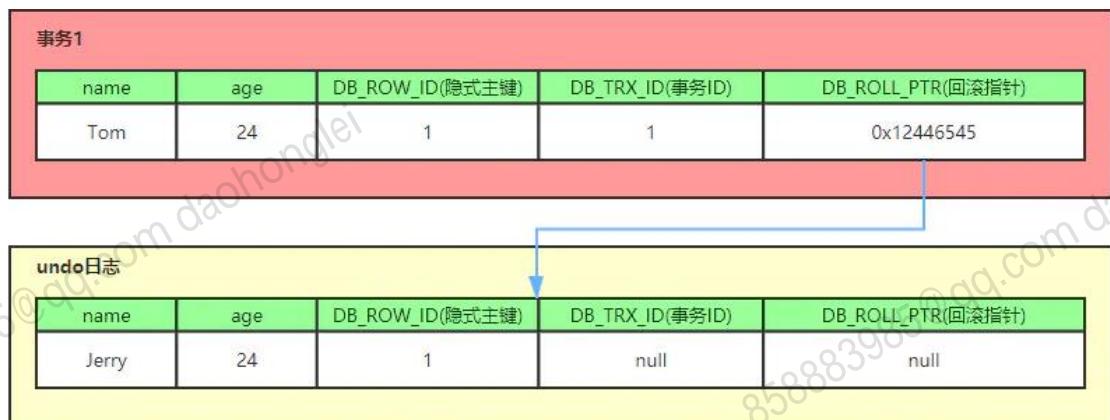
name	age	DB_ROW_ID(隐式主键)	DB_TRX_ID(事务ID)	DB_ROLL_PTR(回滚指针)
Jerry	24	1	null	null

二、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



三、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

从上面，我们就可以看出，不同事务或者相同事务的对同一记录的修改，会导致该记录的 undo log 成为一条记录版本线性表，既链表，undo log 的链首就是最新的旧记录，链尾就是最早的旧记录（当然就像之前说的该 undo log 的节点可能是会 purge 线程清除掉，向图中的第一条 insert undo log，其实在事务提交之后可能就被删除丢失了，不过这里为了演示，所以还放在这里）

Read View(读视图)

什么是 Read View?

什么是 Read View, 说白了 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快照, 记录并维护系统当前活跃事务的 ID(当每个事务开启时, 都会被分配一个 ID, 这个 ID 是递增的, 所以最新的事务, ID 值越大)

所以我们知道 Read View 主要是用来做可见性判断的, 即当我们某个事务执行快照读的时候, 对该记录创建一个 Read View 读视图, 把它比作条件用来判断当前事务能够看到哪个版本的数据, 既可能是当前最新的数据, 也有可能是该行记录的 undo log 里面的某个版本的数据。

Read View 遵循一个可见性算法, 主要是将要被修改的数据的最新记录中的 DB_TRX_ID (即当前事务 ID) 取出来, 与系统当前其他活跃事务的 ID 去对比 (由 Read View 维护) , 如果 DB_TRX_ID 跟 Read View 的属性做了某些比较, 不符合可见性, 那就通过 DB_ROLL_PTR 回滚指针去取出 Undo Log 中的 DB_TRX_ID 再比较, 即遍历链表的 DB_TRX_ID (从链首到链尾, 即从最近的一次修改查起) , 直到找到满足特定条件的 DB_TRX_ID, 那么这个 DB_TRX_ID 所在的旧记录就是当前事务能看见的最新老版本
那么这个判断条件是什么呢?

```

157     /** Check whether the changes by id are visible.
158     @param[in]      id      transaction id to check against the view
159     @param[in]      name    table name
160     @return whether the view sees the modifications of id. */
161     bool changes_visible(
162         trx_id_t           id,
163         const table_name_t& name) const
164         MY_ATTRIBUTE((warn_unused_result))
165     {
166         ut_ad(id > 0);
167
168         if (id < m_up_limit_id || id == m_creator_trx_id) {
169
170             return(true);
171
172         check_trx_id_sanity(id, name);
173
174         if (id >= m_low_limit_id) {
175
176             return(false);
177
178         } else if (m_ids.empty()) {
179
180             return(true);
181
182         }
183
184         const ids_t::value_type*      p = m_ids.data();
185
186         return(!std::binary_search(p, p + m_ids.size(), id));
187     }

```

如上，它是一段 MySQL 判断可见性的一段源码，即 changes_visible 方法（不完全哈，但能看出大致逻辑），该方法展示了我们拿 DB_TRX_ID 去跟 Read View 某些属性进行怎样的比较

在展示之前，我先简化一下 Read View，我们可以把 Read View 简单的理解成有三个全局属性

trx_list (名字我随便取的)

一个数值列表，用来维护 Read View 生成时刻系统正活跃的事务 ID

up_limit_id

记录 trx_list 列表中事务 ID 最小的 ID

low_limit_id

ReadView 生成时刻系统尚未分配的下一个事务 ID，也就是目前已出现过的事务 ID

的最大值+1

- 首先比较 DB_TRX_ID < up_limit_id, 如果小于，则当前事务能看到 DB_TRX_ID 所在的记录，如果大于等于进入下一个判断
- 接下来判断 DB_TRX_ID 大于等于 low_limit_id , 如果大于等于则代表 DB_TRX_ID 所在的记录在 Read View 生成后才出现的，那对当前事务肯定不可见，如果小于则进入下一个判断
- 判断 DB_TRX_ID 是否在活跃事务之中，`trx_list.contains(DB_TRX_ID)`，如果在，则代表我 Read View 生成时刻，你这个事务还在活跃，还没有 Commit，你修改的数据，我当前事务也是看不见的；如果不在，则说明，你这个事务在 Read View 生成之前就已经 Commit 了，你修改的结果，我当前事务是能看见的

整体流程

我们在了解了隐式字段，undo log，以及 Read View 的概念之后，就可以来看看 MVCC 实现的整体流程是怎么样了

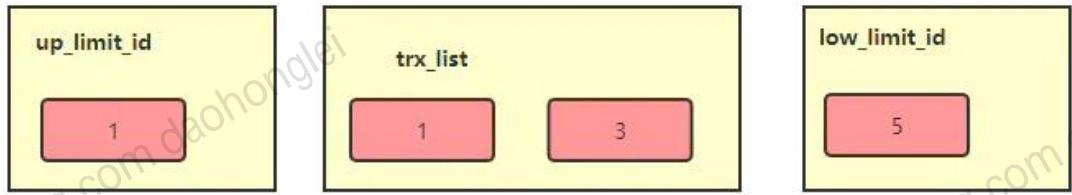
整体的流程是怎么样的呢？我们可以模拟一下

- 当事务 2 对某行数据执行了快照读，数据库为该行数据生成一个 Read View 读视图，假设当前事务 ID 为 2，此时还有事务 1 和事务 3 在活跃中，事务 4 在事务 2 快照读前一刻提交更新了，所以 Read View 记录了系统当前活跃事务 1, 3 的 ID，维护在一个列表上，假设我们称为 `trx_list`

事务1	事务2	事务3	事务4
事务开始	事务开始	事务开始	事务开始
...	修改且已提交
进行中	快照读	进行中	

- Read View 不仅仅会通过一个列表 `trx_list` 来维护事务 2 执行快照读那刻系统正活跃

的事务 ID，还会有两个属性 up_limit_id (记录 trx_list 列表中事务 ID 最小的 ID) , low_limit_id(记录 trx_list 列表中事务 ID 最大的 ID，也有人说快照读那刻系统尚未分配的下一个事务 ID 也就是目前已出现过的事务 ID 的最大值+1，我更倾向于后者；所以在这里例子中 up_limit_id 就是 1, low_limit_id 就是 $4 + 1 = 5$, trx_list 集合的值是 1,3, Read View 如下图



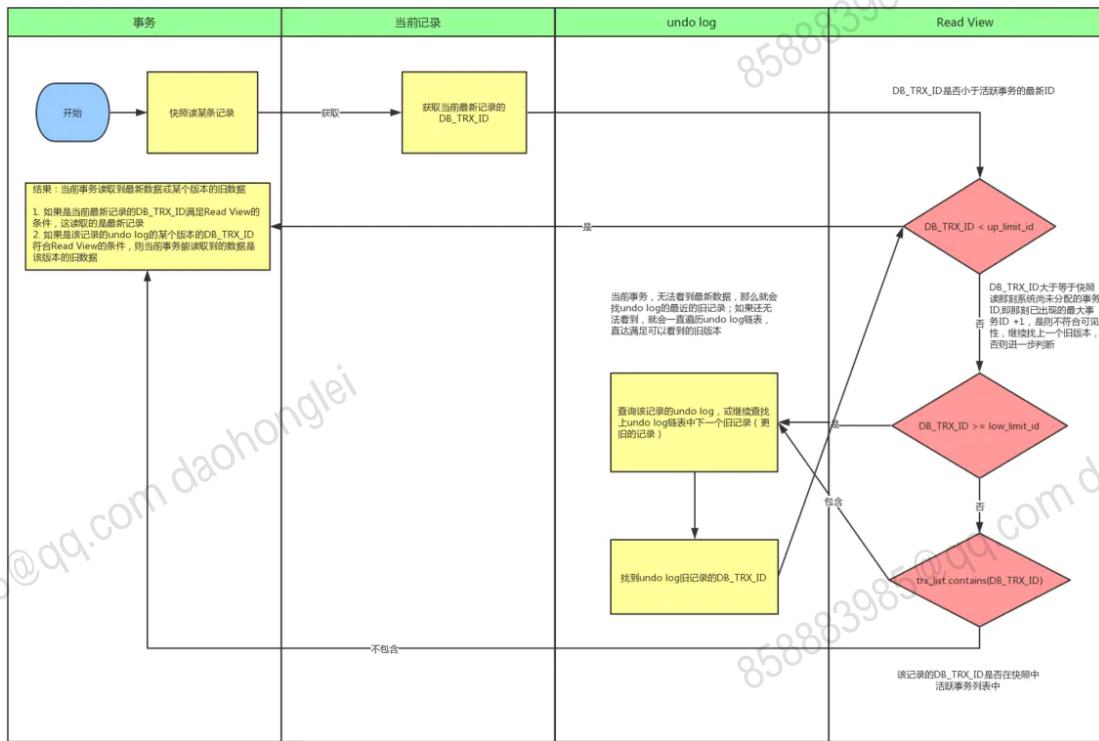
- 我们的例子中，只有事务 4 修改过该行记录，并在事务 2 执行快照读前，就提交了事务，所以当前该行当前数据的 undo log 如下图所示；我们的事务 2 在快照读该行记录的时候，就会拿该行记录的 DB_TRX_ID 去跟 up_limit_id,low_limit_id 和活跃事务 ID 列表(trx_list)进行比较，判断当前事务 2 能看到该记录的版本是哪个。

事务2		因为当前事务2还没修改字段，仅查，所以最新数据的事务ID依然标记的是事务4的ID		
字段	DB_ROW_ID(隐式主键)	DB_TRX_ID(事务ID)	DB_ROLL_PTR(回滚指针)	
A	1	4	0x6546123	

undo日志		这是事务4修改前的版本 (由某个事务完成，所以事务ID不明)		
字段	DB_ROW_ID(隐式主键)	DB_TRX_ID(事务ID)	DB_ROLL_PTR(回滚指针)	
X	1	某个事务ID	0x12446545	

- 所以先拿该记录 DB_TRX_ID 字段记录的事务 ID 4 去跟 Read View 的的 up_limit_id 比较，看 4 是否小于 up_limit_id(1)，所以不符合条件，继续判断 4 是否大于等于 low_limit_id(5)，也不符合条件，最后判断 4 是否处于 trx_list 中的活跃事务，最后发现事务 ID 为 4 的事务不在当前活跃事务列表中，符合可见性条件，所以事务 4 修改后提交的最新结果对事务 2 快照读时是可见的，所以事务 2 能读到的最新数据记录是事

务 4 所提交的版本，而事务 4 提交的版本也是全局角度上最新的版本



- 也正是 Read View 生成时机的不同，从而造成 RC,RR 级别下快照读的结果的不同

RC,RR 级别下的 InnoDB 快照读有什么不同？

正是 Read View 生成时机的不同，从而造成 RC,RR 级别下快照读的结果的不同

- 在 RR 级别下的某个事务的对某条记录的第一次快照读会创建一个快照及 Read View，将当前系统活跃的其他事务记录起来，此后在调用快照读的时候，还是使用的是同一个 Read View，所以只要当前事务在其他事务提交更新之前使用过快照读，那么之后的快照读使用的都是同一个 Read View，所以对之后的修改不可见；即 RR 级别下，快照读生成 Read View 时，Read View 会记录此时所有其他活动事务的快照，这些事务的修改对于当前事务都是不可见的。而早于 Read View 创建的事务所做的修改均是可见。
- 而在 RC 级别下的，事务中，每次快照读都会新生成一个快照和 Read View，这就是我们在 RC 级别下的事务中可以看到别的事务提交的更新的原因。

总之在 RC 隔离级别下，是每个快照读都会生成并获取最新的 Read View；而在 RR 隔离级别下，则是同一个事务中的第一个快照读才会创建 Read View，之后的快照读获取的都是同一个 Read View。

readView

readView（读视图）是快照读 SQL 执行时 MVCC 提取数据的依据，记录并维护正在活动的事务（未提交的事务）

四个核心字段

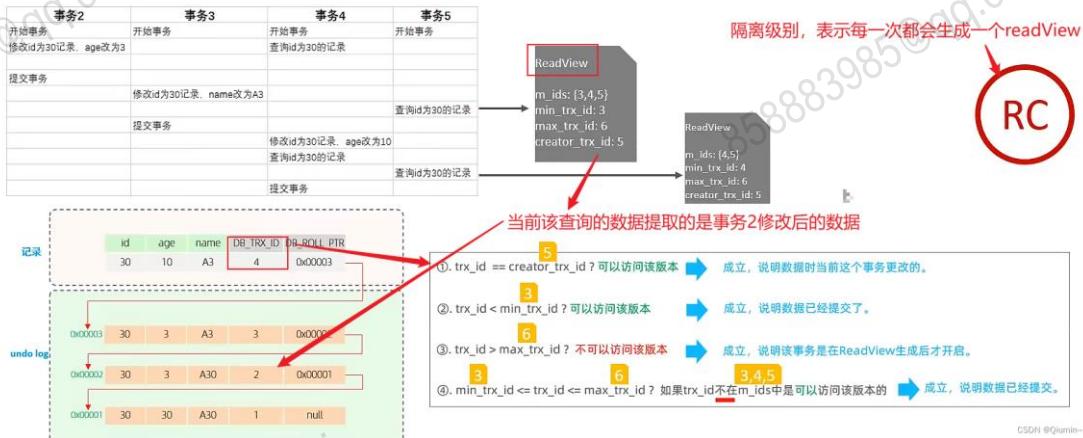
字段	含义
m_ids	当前活跃事务的 id
min_trx_id	活跃事务中最小的 id
max_trx_id	预分配事务 id，最大事务 id+1
creator_trx_id	readView 创建者事务的 id

版本链事务提取的规则



CSDN @Qiumin-

演示



锁

全局锁

用作全量备份时，保证表与表之间的数据一致性

- flush tables with read lock;

使用全局读锁锁定所有数据库的所有表。这时会阻塞其它所有 DML 以及 DDL 操作，这样可以避免备份过程中的数据不一致。接下来可以执行备份，最后用 unlock tables 来解锁

但这属于比较重的操作，可以使用 --single-transaction 参数来完成不加锁的一致性备份
(仅针对 InnoDB 引擎的表)

- mysqldump --single-transaction -uroot -p test > 1.sql

表级锁 (InnoDB)

1. 表锁

- 语法：加锁 lock tables 表名 read/write，解锁 unlock tables

- 缺点：粒度较粗，在 InnoDB 引擎很少使用

2. 元数据锁，即 metadata-lock (MDL)，**主要是为了避免 DML 与 DDL 冲突**

- 加元数据锁的几种情况

- ◆ lock tables read/write，类型为 SHARED_READ_ONLY 和 SHARED_NO_READ_WRITE
- ◆ alter table，类型为 EXCLUSIVE，与其它 DML 都互斥
- ◆ select, select ... lock in share mode，类型为 SHARED_READ
- ◆ insert, update, delete, select for update，类型为 SHARED_WRITE

- DML 的元数据锁之间不互斥
- select object_type,object_schema,object_name,lock_type,lock_duration
from performance_schema.metadata_locks;

3. IS (意向共享) 与 IX (意向排他)，**主要是避免 DML 与表锁冲突**

- DML 主要目的是加行锁，为了让表锁不用检查每行数据是否加锁，加意向锁（表级）来减少表锁的判断，意向锁之间不会互斥

- 由 DML 语句添加，例如 select ... lock in share mode 会加 IS 锁，insert, update, delete, select ... for update 会加 IX 锁

- select
object_schema,object_name,index_name,lock_type,lock_mode,lock_data
from performance_schema.data_locks

行级锁 (InnoDB)

种类

- a) 行锁 – 在 RC RR 下, 锁住的是行, 防止其他事务对此行 update 或 delete,
- b) 间隙锁 – 在 RR 下, 锁住的是间隙, 防止其他事务在这个间隙 insert 产生幻读
- c) 临键锁 – 在 RR 下, 锁住的是前面间隙+行, 特定条件下可优化为行锁

注意

它们锁定的其实都是索引上的行与间隙, 根据索引的有序性来确定间隙

```
create table t (id int primary key, name varchar(10), age int, key (name));

insert into t values(1, 'zhangsan', 18);

insert into t values(2, 'lisi', 20);

insert into t values(3, 'wangwu', 21);

insert into t values(4, 'zhangsan', 17);

insert into t values(8, 'zhang', 18);

insert into t values(12, 'zhang', 20);
```

间隙锁

```
begin;

select * from t where id = 9 for update;

select object_schema, object_name, index_name, lock_type, lock_mode, lock_data
from performance_schema.data_locks where object_name='t';
```

```
update t set age=100 where id = 8;
```

```
update t set age=100 where id = 12;
```

```
insert into t values(10,'aaa',18);

临键锁和记录锁

begin;

select * from t where id >= 8 for update;

select object_schema,object_name,index_name,lock_type,lock_mode,lock_data
from performance_schema.data_locks where object_name='t';

insert into t values(7,'aaa',18);

update t set age=100 where id = 8;

insert into t values(10,'aaa',18);

update t set age=100 where id = 12;

insert into t values(13,'aaa',18);
```

https://blog.csdn.net/m0_57752520/article/details/124852971

锁

锁是计算机协调多个进程或线程并发访问某一个资源的机制，为了保证数据并发访问的一致性和安全性。锁对数据而言显得重要，也更加复杂了。

分类 【按锁的粒度分】

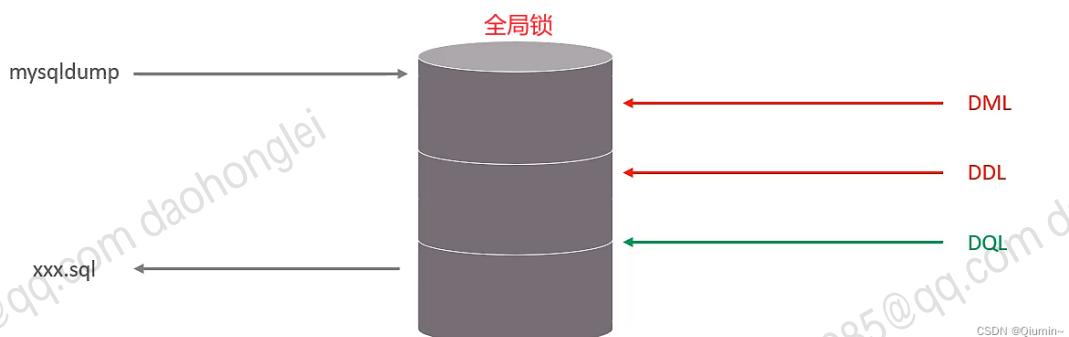
- 全局锁：锁定数据库中的所有表。
- 表级锁：每次操作锁住整张表。

- 表锁
- 元数据锁
- 意向锁
- 行级锁：每次操作锁住对应的行数据。
 - 行锁
 - 间隙锁
 - 临建锁

1 全局锁

全局锁的粒度是最大的，锁住的是整个数据库实例，加锁后整个实例处于只读状态【即查询】，其他语句都处于阻塞状态

典型的使用场景：全库备份时，阻塞所有的 DML、DDL 语句，保证数据的完整性



加锁全局锁： flush tables with read lock;

释放全局锁： unlock tables;

特点

数据库中加全局锁，是一个比较重的操作，存在一下问题：

- 如果在主库上备份，那么在备份期间都不能执行更新操作。
- 如果在从库上备份，那么在备份期间从库不能执行主库的同步过来的二进制文件。会导致主从延迟。

在 innodb 引擎中，可以在备份时加上参数 --single-transaction 参数来完成不加锁的一致性数据备份

```
mysqldump --single-transaction -uroot -p123456 itcast>itcast.sql #备份到  
itcast.sql 文件
```

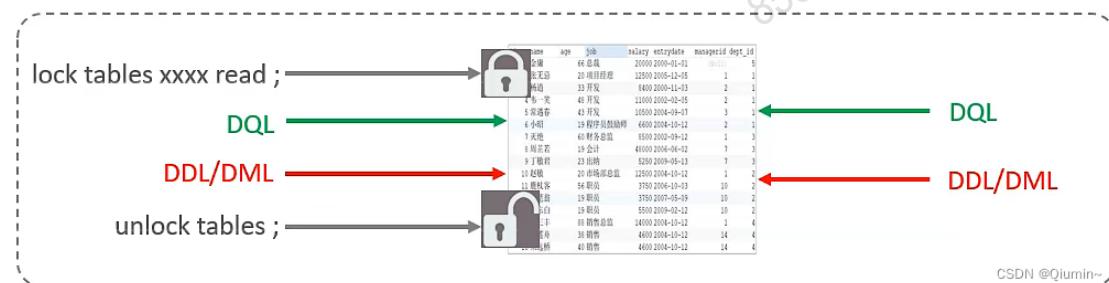
2 表级锁

锁整张表

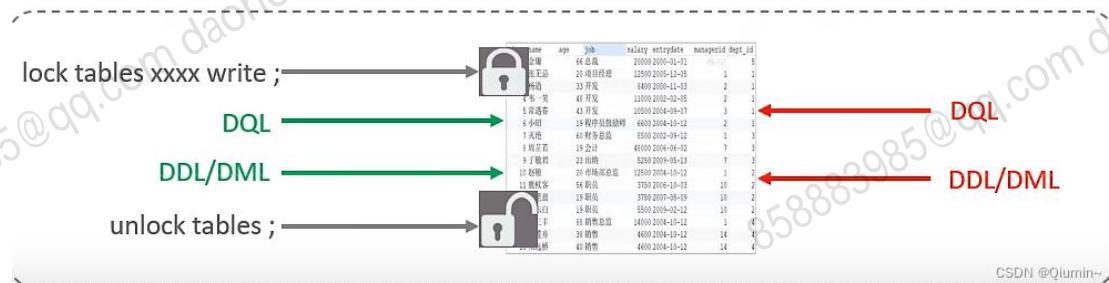
(一) 表锁

对于表锁分为：

- 表级共享锁 (read lock)



- 表级占写锁 (write lock)



语法：

- 加锁： lock tables 表名 read/write 【加都锁、加写锁】
- 释放锁： unlock tables

读锁不会阻塞其他客户端的读操作，但会阻塞写操作。写锁不但阻塞其他客户端的写操作也会阻塞读操作，自己的客户端不阻塞

(二) 元数据锁

MDL 加锁过程是系统自动控制，无需显示声明，在访问一张表的时候会自动加上。MDL 锁主要作用是维护表元数据的一致性，在表上有活动事务的时候，不可以对元数据进行写入操作。**为了避免 DML 与 DDL 语句的冲突，保证读写的正确性。**

mysql5.5 中引入了 MDL，**当对一张表数据进行增删改查的时候加 MDL 共享读锁或共享写锁；当对表结构进行变更时加 MDL 非共享写锁（排他锁）**

对应的 SQL	锁类型	说明
select	share_read	与 share_read、share_write 兼容，与 exclusive 互斥
insert、update、delete	share_write	与 share_read、share_write 兼容，与 exclusive 互斥
alter	exclusive	与以上都互斥

查看元数据锁

```
select object_type,object_schema,object_name,lock_type,lock_duration  
from performance_schema.metadata_locks;
```

(三) 意向锁

意向锁主要解决表锁和行锁的冲突问题，这样表锁不需逐行检查是否加了行锁。

1. 意向共享锁 (IS)：由语句 select ...lock inshare mode 添加，与读锁兼容与写锁互斥
2. 意向排它锁 (IX)：由 insert、update、delete、select ... for update 添加，与读锁与写锁都互斥

查看意向锁

select

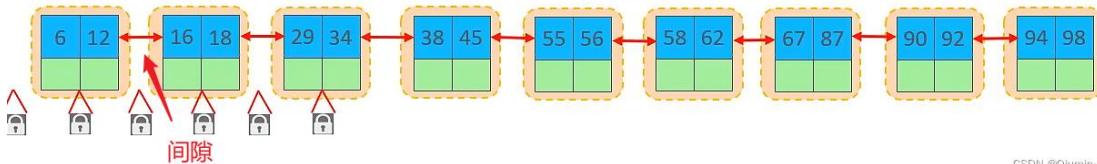
```
object_schema,object_name,index_name,lock_type,lock_mode,lock_data from  
perfomance_schema.data_locks;
```

3 行级锁

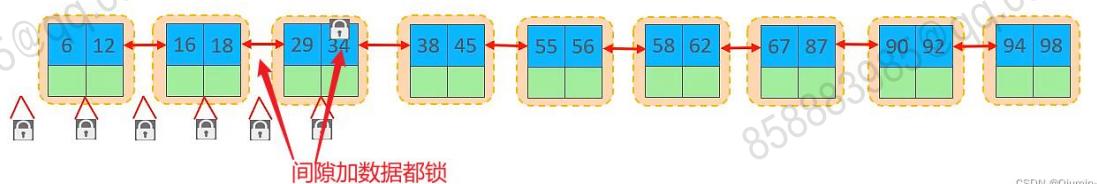
行级锁，每次操作时是锁住行数据，锁粒度最小，发生冲突的概率最低，并发度最高，应用在 innodb 存储引擎中

行锁是针对索引加锁的而不是对记录加锁

- 行锁 (Record lock)：在 RC、RR 隔离级别下都支持
- 间隙锁 (Gap lock)：锁定索引记录间隙（不含该记录），确保索引记录的间隙不变，防止其他事务在这个间隙插入值，产生幻读。在 RR 隔离级别下支持。



- 临建锁 (Next-Key lock)：行锁和间隙锁的组合，同时锁住数据，以及数据前面的间隙，在 RR 隔离级别下支持。



(一) 行锁

innodb 实现类一下两种类型的行锁

- 共享锁 (S)：允许事务读，阻止其他事务加排他锁。
- 排他锁 (X)：允许获得排他锁的事务操作，其他事务阻塞排他锁和共享锁都不能加。

	共享锁	排他锁
--	-----	-----

共享锁 (S)	兼容	冲突
排他锁 (X)	冲突	冲突

- insert、update、delete、select ... for update 自动加排它锁。
- select ... lock in share mode 加共享锁。
- select ... 不加任何锁

注意：需针对索引加锁，否则会升级为表锁

查看行锁情况

```
select
object_schema,object_name,index_name,lock_type,lock_mode,lock_data from
perfomance_schema.data_locks;
```

(二) 间隙锁、临建锁

默认情况下，innodb 在 RR 事务隔离级别运行，innodb 使用 nest-key 锁进行搜索和索引扫描，以防止幻读

- 索引上的等值查询（唯一索引），给不存在的记录加锁时，优化为间隙锁。
- 索引上的等值查询（普通索引），向右遍历时最后一个值不满足需求时，临建锁退化为间隙锁。
- 索引上的范围查询（唯一索引），会访问到不满足条件的第一个值为止。

注意：间隙锁的唯一目的就是防止其他事务插入间隙。**间隙锁可以共存，一个事务采用的间隙锁不会阻止另一个事务在同一间隙上加间隙锁。**

InnoDB 表级锁

在绝大部分情况下都应该使用行锁，因为事务和行锁往往是选择 InnoDB 的理由，但个别

情况下也使用表级锁；

- 1) 事务需要更新大部分或全部数据，表又比较大，如果使用默认的行锁，不仅这个事务执行效率低，而且可能造成其他事务长时间等待和锁冲突；
- 2) 事务涉及多个表，比较复杂，很可能引起死锁，造成大量事务回滚。

如：

LOCK TABLE user READ; 读锁锁表

LOCK TABLE user WRITE; 写锁锁表

事务执行...

COMMIT/ROLLBACK; 事务提交或者回滚

UNLOCK TABLES; 本身自带提交事务，释放线程占用的所有表锁

意向共享锁 & 意向排它锁

使用表锁的时候，设计一个效率问题：

要获取一张表的共享锁 S 或排它锁 X，最起码得确定，这张表没有被其他事务获取过 X 锁。

意向锁解决上面的问题：当要获取表的 X 锁时，不需要再检查表中哪些行锁（X 或 S）被占用，只需要检查 IS 锁和 IX 锁即可！

意向共享锁（IS 锁）：事务计划给记录加行共享锁，事务在给一行记录加共享锁前，必须先取得该表的 IS 锁。

意向排他锁（IX 锁）：事务计划给记录加行排他锁，事务在给一行记录加排他锁前，必须先取得该表的 IX 锁。

	X	IX	S	IS
X	Conflict	Conflict	Conflict	Conflict
IX	Conflict	兼容	Conflict	兼容
S	Conflict	Conflict	兼容	兼容
IS	Conflict	兼容	兼容	兼容

CSDN @_素伦

自增锁 (AI) AUTO-INC Lock 是一种特殊的表级锁，发生在 AUTO_INCREMENT 约束下的插入操作，在完成对自增长插入的 sql 语句后立即释放。

锁	S	X	IS	IX	AI
S	兼容	冲突	兼容	冲突	冲突
X	冲突	冲突	冲突	冲突	冲突
IS	兼容	冲突	兼容	兼容	兼容
IX	冲突	冲突	兼容	兼容	兼容
AI	冲突	冲突	兼容	兼容	冲突

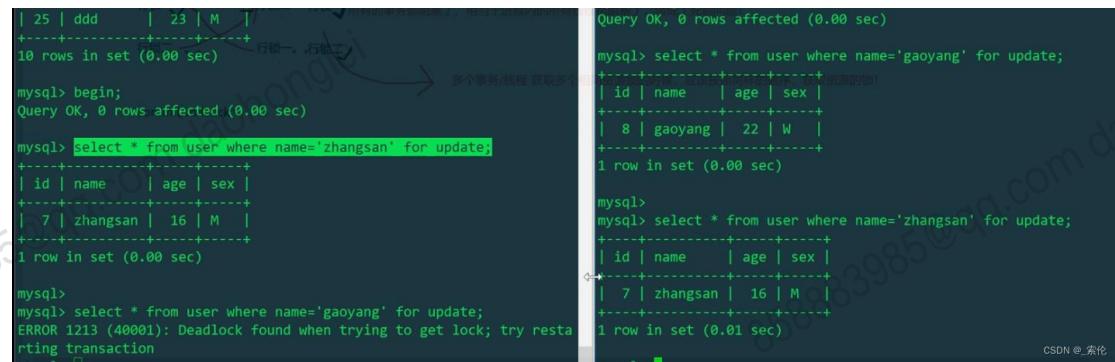
- 1、意向锁是由 InnoDB 存储引擎获取行锁之前自己获取的
- 2、意向锁之间都是兼容的，不会产生冲突
- 3、意向锁存在的意义是为了更高效的获取表锁（表格中的 X 和 S 指的是表锁，不是行锁！！！）
- 4、意向锁是表级锁，协调表锁和行锁的共存关系。主要目的是显示事务正在锁定某行或者试图锁定某行。

死锁

MyISAM 表锁是 deadlock free 的，这是因为 MyISAM 总是一次获得所需的全部锁，

要么全部满足，要么等待，因此不会出现死锁。但在 InnoDB 中，除单个 SQL 组成的事
务外，锁是逐步获得的，即锁的粒度比较小，这就决定了在 InnoDB 中发生死锁是可能的。

```
mysql> select * from test_dead_lock where id=1 for update;  
  
ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting  
transaction
```



```
| 25 | ddd      | 23 | M   |  
+----+-----+-----+  
10 rows in set (0.00 sec)  
  
mysql> begin;  
Query OK, 0 rows affected (0.00 sec)  
  
mysql> select * from user where name='zhangsan' for update;  
+----+-----+-----+  
| id | name  | age | sex |  
+----+-----+-----+  
| 7  | zhangsan | 16 | M  |  
+----+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>  
mysql> select * from user where name='gaoyang' for update;  
+----+-----+-----+  
| id | name  | age | sex |  
+----+-----+-----+  
| 8  | gaoyang | 22 | W  |  
+----+-----+-----+  
1 row in set (0.00 sec)  
  
mysql>  
mysql> select * from user where name='zhangsan' for update;  
+----+-----+-----+  
| id | name  | age | sex |  
+----+-----+-----+  
| 7  | zhangsan | 16 | M  |  
+----+-----+-----+  
1 row in set (0.01 sec)
```

死锁问题一般都是我们自己的应用造成的，和多线程编程的死锁情况相似，大部分都
是由于我们多个线程在获取多个锁资源的时候，获取的顺序不同而导致的死锁问题。因此
我们应用在对数据库的多个表做更新的时候，不同的代码段，应对这些表按相同的顺序进
行更新操作，以防止锁冲突导致死锁问题。

锁的优化建议

- 1.尽量使用较低的隔离级别
- 2.设计合理的索引并尽量使用索引访问数据，使加锁更加准确，减少锁冲突的机会提高并发能力
- 3.选择合理的事务大小，小事务发生锁冲突的概率小
- 4.不同的程序访问一组表时，应尽量约定以相同的顺序访问各表，对一个表而言，尽可能以固定的顺序存取表中的行。这样可以大大减少死锁的机会
- 5.尽量用相等条件访问数据，这样可以避免间隙锁对并发插入的影响
- 6.不要申请超过实际需要的锁级别

7.除非必须，查询时不要显示加锁

MySQL 的锁机制

锁在 MySQL 中是非常重要的一部分，锁对 MySQL 的数据访问并发有着举足轻重的影响。锁涉及到的知识篇幅也很多，所以要啃完并消化到自己的肚子里，是需要静下心好好反反复复几遍地细细品味。本文是对锁的一个大概的整理，一些相关深入的细节，还是需要找到相关书籍来继续夯实。

1.锁的认识

1.1 锁的解释

计算机协调多个进程或线程并发访问某一资源的机制。

1.2 锁的重要性

在数据库中，除传统计算资源（CPU、RAM、I/O 等）的争抢，数据也是一种供多用户共享的资源。如何保证数据并发访问的一致性，有效性，是所有数据库必须要解决的问题。锁冲突也是影响数据库并发访问性能的一个重要因素，因此锁对数据库尤其重要。

1.3 锁的缺点

加锁是消耗资源的，锁的各种操作，包括获得锁、检测锁是否已解除、释放锁等，都会增加系统的开销。

1.4 简单的例子

现如今网购已经特别普遍了，比如淘宝双十一活动，当天的人流量是千万及亿级别的，但商家的库存是有限的。系统为了保证商家的商品库存不发生超卖现象，会对商品的库存进行锁控制。当有用户正在下单某款商品最后一件时，系统会立马对该件商品进行锁

定，防止其他用户也重复下单，直到支付动作完成才会释放（支付成功则立即减库存售罄，支付失败则立即释放）。

2. 锁的类型

2.1 表锁

种类

读锁 (read lock) , 也叫共享锁 (shared lock) , 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 (select)

写锁 (write lock) , 也叫排他锁 (exclusive lock) , 当前操作没完成之前，会阻塞其它读和写操作 (update、insert、delete)

存储引擎默认锁 MyISAM

特点

1. 对整张表加锁
2. 开销小
3. 加锁快
4. 无死锁
5. 锁粒度大，发生锁冲突概率大，并发性低

结论

1. 读锁会阻塞写操作，不会阻塞读操作
2. 写锁会阻塞读和写操作

建议

MyISAM 的读写锁调度是写优先，这也是 MyISAM 不适合做写为主表的引擎，因为写锁以后，其它线程不能做任何操作，大量的更新使查询很难得到锁，从而造成永远阻塞。

2.2 行锁

种类

读锁 (read lock) , 也叫共享锁 (shared lock) , 允许一个事务去读一行，阻止

其他事务获得相同数据集的排他锁

写锁 (write lock) , 也叫排他锁 (exclusive lock) , 允许获得排他锁的事务更新数据, 阻止其他事务取得相同数据集的共享锁和排他锁

意向共享锁 (IS) , 一个事务给一个数据行加共享锁时, 必须先获得表的 IS 锁

意向排它锁 (IX) , 一个事务给一个数据行加排他锁时, 必须先获得该表的 IX 锁

存储引擎默认锁 InnoDB

特点

1. 对一行数据加锁 2. 开销大 3. 加锁慢 4. 会出现死锁 5. 锁粒度小, 发生锁冲突概率最高, 并发性高

事务并发带来的问题

1. 更新丢失 解决: 让事务变成串行操作, 而不是并发的操作, 即对每个事务开始---对读取记录加排他锁 2. 脏读 解决: 隔离级别为 Read uncommitted
3. 不可重读 解决: 使用 Next-Key Lock 算法来避免 4. 幻读 解决: 间隙锁 (Gap Lock)

2.3 页锁

开销、加锁时间和锁粒度介于表锁和行锁之间, 会出现死锁, 并发处理能力一般 (此锁不做多介绍)

3. 如何上锁?

3.1 表锁

隐式上锁 (默认, 自动加锁自动释放)

select //上读锁

insert、update、delete //上写锁

显式上锁 (手动)

lock table tableName read;//读锁 lock table tableName write;//写锁

解锁 (手动)

unlock tables;//所有锁表

session01	session02
lock table teacher read;//上读锁	
select * from teacher; //可以正常读取	select * from teacher; //可以正常读取
update teacher set name = 3 where id =2;//报错因被上读锁不能写操作	update teacher set name = 3 where id =2;//被阻塞
unlock tables;//解锁	
	update teacher set name = 3 where id =2;//更新操作成功
session01	session02
lock table teacher write;//上写锁	
select * from teacher; //可以正常读取	select * from teacher; //被阻塞
update teacher set name = 3 where id =2;//可以正常更新操作	update teacher set name = 4 where id =2;//被阻塞
unlock tables;//解锁	
	select * from teacher; //读取成功
	update teacher set name = 4 where id =2;//更新操作成功

3.2 行锁

隐式上锁 (默认, 自动加锁自动释放)

select //不会上锁

insert、update、delete //上写锁

显式上锁 (手动)

select * from tableName lock in share mode; //读锁 select * from tableName for update; //写锁

解锁 (手动)

1. 提交事务 (commit) 2. 回滚事务 (rollback) 3. kill 阻塞进程

session01	session02
begin;	
select * from teacher where id = 2 lock in share mode;//上读锁	
	select * from teacher where id = 2;//可 以正常读取
update teacher set name = 3 where id =2;// 可以更新操作	update teacher set name = 5 where id =2;//被阻塞
commit;	
	update teacher set name = 5 where id =2;//更新操作成功

session01	session02
begin;	
select * from teacher where id = 2 for	

update://上写锁	
	select * from teacher where id = 2;//可以正常读取
update teacher set name = 3 where id =2;// 可以更新操作	update teacher set name = 5 where id =2;//被阻塞
rollback;	
	update teacher set name = 5 where id =2;//更新操作成功

为什么上了写锁，别的事务还可以读操作？因为 InnoDB 有 MVCC 机制（多版本并发控制），可以使用快照读，而不会被阻塞。

4.行锁的实现算法

4.1 Record Lock 锁

单个行记录上的锁 Record Lock 总是会去锁住索引记录，如果 InnoDB 存储引擎表建立的时候没有设置任何一个索引，这时 InnoDB 存储引擎会使用隐式的主键来进行锁定

4.2 Gap Lock 锁

当我们用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB 会给出符合条件的已有数据记录的索引加锁，对于键值在条件范围内但并不存在的记录。优点：解决了事务并发的幻读问题 不足：因为 query 执行过程中通过范围查找的话，他会锁定一个范围内所有的索引键值，即使这个键值并不存在。间隙锁有一个致命的弱点，就是当锁定一个范围键值之后，即使某些不存在的键值也会被无辜的锁定，而造成锁定的时候无法插入锁定键值范围内任何数据。在某些场景下这可能会对性能造成很大的危害。

4.3 Next-key Lock 锁

同时锁住数据+间隙锁 在 Repeatable Read 隔离级别下，Next-key Lock 算法是默认的行记录锁定算法。

4.4 行锁的注意点

1. 只有通过索引条件检索数据时，InnoDB 才会使用行级锁，否则会使用表级锁(索引失效，行锁变表锁)
2. 即使是访问不同行的记录，如果使用的是相同的索引键，会发生锁冲突
3. 如果数据表建有多个索引时，可以通过不同的索引锁定不同的行

5.如何排查锁？

5.1 表锁

查看表锁情况

```
show open tables;
```

Database	Table	In_use	Name_locked
information_schema	VIEW_TABLE_USAGE	0	0
information_schema	VIEW_ROUTINE_USAGE	0	0
information_schema	VIEWS	0	0
information_schema	TRIGGERS	0	0
information_schema	ST_UNITS_OF_MEASURE	0	0
mysql	tablespaces	0	0
mysql	time_zone_transition_type	0	0
information_schema	TABLES	0	0
mysql	plugin	0	0
mysql	catalogs	0	0
information_schema	COLUMN_STATISTICS	0	0
mysql	events	0	0
information_schema	COLUMNS	0	0
mysql	servers	0	0
mysql	slave_master_info	0	0
information_schema	PARAMETERS	0	0
mysql	time_zone_leap_second	0	0
mysql	collations	0	0
mysql	character_sets	0	0
school	subject	0	0
mysql	password_history	0	0
mysql	global_grants	0	0
mysql	role_edges	0	0
mysql	proxies_priv	0	0
mysql	column_type_elements	0	0
mysql	slow_log	0	0
mysql	user	0	0
mysql	parameters	0	0
information_schema	SCHEMATA	0	0
performance_schema	session_status	0	0
mysql	db	0	0
mysql	foreign_keys	0	0
mysql	innodb_table_stats	0	0
mysql	foreign_key_column_usage	0	0
mysql	slave_worker_info	0	0
mysql	view_routine_usage	0	0
mysql	index_column_usage	0	0
mysql	schemata	0	0
mysql	triggers	0	0
mysql	index_partitions	0	0
mysql	columns_priv	0	0
mysql	time_zone_name	0	0
mysql	engine_cost	0	0
information_schema	SHOW_STATISTICS	0	0
mysql	default_roles	0	0
mysql	gtid_executed	0	0
mysql	time_zone_transition	0	0

表锁分析

show status like 'table%';

```
[mysql]> show status like 'table%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Table_locks_immediate | 183 |
| Table_locks_waited | 0 |
| Table_open_cache_hits | 266 |
| Table_open_cache_misses | 29 |
| Table_open_cache_overflows | 0 |
+-----+
```

1. table_locks_waited 出现表级锁定争用而发生等待的次数（不能立即获取锁的次数，每等待一次值加 1），此值高说明存在着较严重的表级锁争用情况 2.
- table_locks_immediate 产生表级锁定次数，不是可以立即获取锁的查询次数，每立即获取锁加 1

5.2 行锁

行锁分析

```
show status like 'innodb_row_lock%';
```

```
[mysql]> show status like 'innodb_row_lock%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Innodb_row_lock_current_waits | 1 |
| Innodb_row_lock_time | 0 |
| Innodb_row_lock_time_avg | 0 |
| Innodb_row_lock_time_max | 0 |
| Innodb_row_lock_waits | 1 |
+-----+
```

1. innodb_row_lock_current_waits //当前正在等待锁定的数量 2.
- innodb_row_lock_time //从系统启动到现在锁定总时间长度 3.
- innodb_row_lock_time_avg //每次等待所花平均时间 4.
- innodb_row_lock_time_max //从系统启动到现在等待最长的一次所花时间 5.
- innodb_row_lock_waits //系统启动后到现在总共等待的次数

information_schema 库

1. innodb_lock_waits 表 2. innodb_locks 表 3. innodb_trx 表

优化建议

1. 尽可能让所有数据检索都通过索引来完成，避免无索引行锁升级为表锁
2. 合理设计索引，尽量缩小锁的范围
3. 尽可能较少检索条件，避免间隙锁
4. 尽量控制事务大小，减少锁定资源量和时间长度
5. 尽可能低级别事务隔离

6.死锁

6.1 解释

指两个或者多个事务在同一资源上相互占用，并请求锁定对方占用的资源，从而导致恶性循环的现象

6.2 产生的条件

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，在没有使用完之前，不能强行剥夺
4. 循环等待条件：多个进程之间形成的一种互相循环等待的资源的关系

6.1 解决

等待，直到超时 (innodb_lock_wait_timeout=50s)，自动回滚事务。

发起死锁检测，主动回滚一条事务，让其他事务继续执行

(innodb_deadlock_detect=on)。

1. 查看死锁：show engine innodb status \G
2. 自动检测机制，超时自动回滚代价较小的事务 (innodb_lock_wait_timeout 默认 50s)
3. 人为解决，kill 阻塞进程 (show processlist)
4. wait for graph 等待图（主动检测）

6.1 如何避免

1. 加锁顺序一致，尽可能一次性锁定所需的数据行
2. 尽量基于 primary key (主键) 或 unique key 更新数据
3. 单次操作数据量不宜过多，涉及表尽量少
4. 减少表上索引，减少锁定资源
5. 尽量使用较低的隔离级别
6. 尽量使用相同条件访问数据，这样可以避免间隙锁对并发的插入影响
7. 精心设计索引，尽量使用索引访问数据
8. 借助相关工具：pt-deadlock-logger

7. 乐观锁与悲观锁

悲观锁与乐观锁区别		
	悲观锁	乐观锁
概念	假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作	假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性
实现机制	查询时直接锁住记录使得其它事务不能查询，更不能更新	提交更新时检查版本或者时间戳是否符合
实现方式	表锁、行锁	使用version或者timestamp进行比较
实现者	数据库本身	开发者
适用场景	并发量大	并发量小

7.1 悲观锁

解释：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作

实现机制：表锁、行锁等

实现层面：数据库本身

适用场景：并发量大

7.2 乐观锁

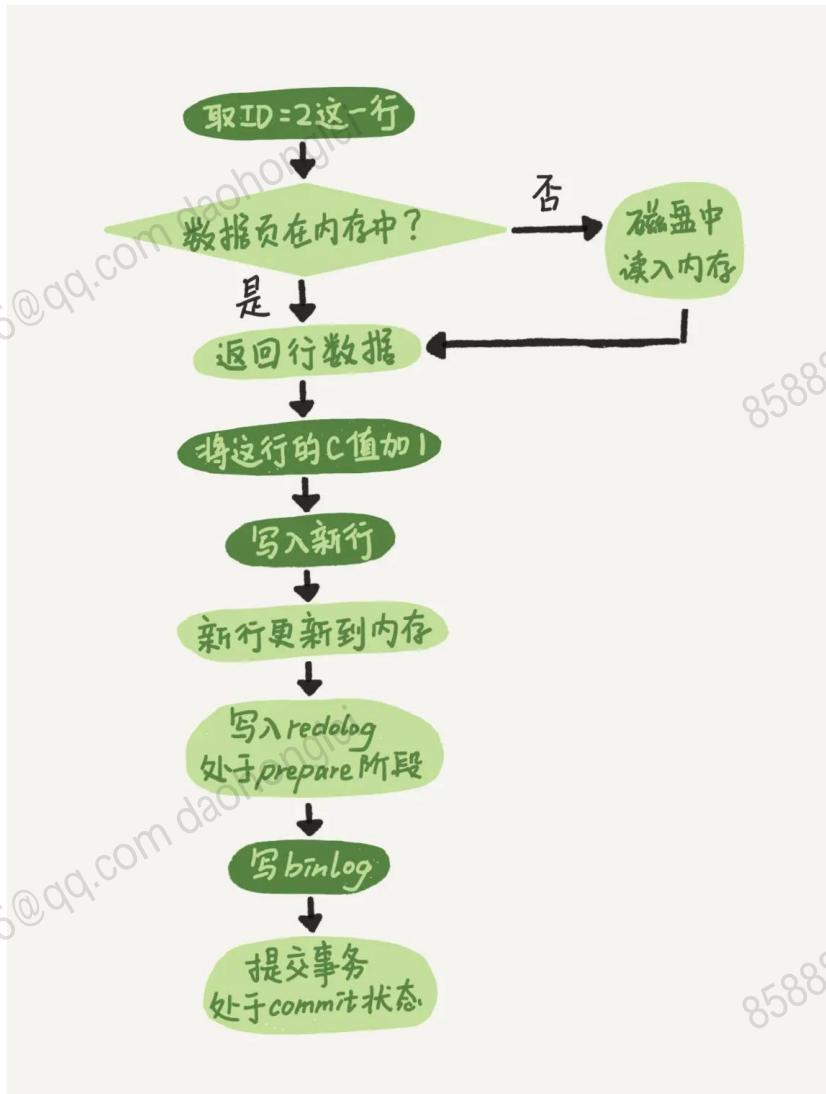
解释：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性

实现机制：提交更新时检查版本号或者时间戳是否符合

实现层面：业务代码

适用场景：并发量小

数据库两阶段提交



两阶段流程

从图中可以看出，在最后提交事务的时候，需要有 3 个步骤：

- 写入 redo log，处于 prepare 状态
- 写 binlog
- 修改 redo log 状态为 commit

ps: redo log 的提交分为 prepare 和 commit 两个阶段，所以称之为**两阶段提交**

为什么需要两阶段提交？

假设当前 ID=2 的行，字段 c 的值是 0，再假设执行 update 语句过程中在写完第一个日志后，第二个日志还没有写完期间发生了 crash，会出现什么情况呢？

1. 先写 redo log 后写 binlog。假设在 redo log 写完，binlog 还没有写完的时候，MySQL 进程异常重启。由于我们前面说过的，redo log 写完之后，系统即使崩溃，仍然能够把数据恢复回来，所以恢复后这一行 c 的值是 1。但是由于 binlog 没写完就 crash 了，这时候 binlog 里面就没有记录这个语句。因此，之后备份日志的时候，存起来的 binlog 里面就没有这条语句。然后你会发现，如果需要用这个 binlog 来恢复临时库的话，由于这个语句的 binlog 丢失，这个临时库就会少了这一次更新，恢复出来的这一行 c 的值就是 0，与原库的值不同。
2. 先写 binlog 后写 redo log。如果在 binlog 写完之后 crash，由于 redo log 还没写，崩溃恢复以后这个事务无效，所以这一行 c 的值是 0。但是 binlog 里面已经记录了“把 c 从 0 改成 1”这个日志。所以，在之后用 binlog 来恢复的时候就多了一个事务出来，恢复出来的这一行 c 的值就是 1，与原库的值不同。
可以看到，如果不使用“两阶段提交”，那么数据库的状态就有可能和用它的日志恢复出来的库的状态不一致。

SQL 优化

1 insert 优化

批量插入

```
insert into tb_user values(1,'haha'),(2,'heihei'),(3,'xixi');
```

手动提交事务

```
start transaction; --开启事务手动提交

insert into tb_user values(1,'haha'),(2,'heihei'),(3,'xixi');

insert into tb_user values(1,'haha'),(2,'heihei'),(3,'xixi');

insert into tb_user values(1,'haha'),(2,'heihei'),(3,'xixi');

commit; --提交
```

主键顺序插入

- 主键顺序插入 【效率较高】
- 主键乱序插入
- 大批数据插入：如果需要大批数据的插入，使用 insert 语句插入的性能就会很低，此时可以使用 mysql 的 load 指令来进行插入。

```
#连接数据库服务时加上参数 --local-infile

mysql --local-infile -u root -p

#设置全局参数 local_infile 为 1， 开启从本地加载文件导入数据的开关

set global local_infile=1

#执行 load 指令将准备好的数据加载到表结构中，'表名'字段以 , 分隔 '以换行分隔行数

据 \n '

load data local infile '文件的地址' into table '表名' fields terminated by ',' lines
terminated by '\n';
```

2 主键优化

列分裂

- 页可以为空，也可以填充一半，填充 100%，每个页包含了 $2-n$ 行数据（一行数据过大时就会行溢出），根据主键排序的，如果是乱序插入的话，就可能会发生列分裂。

列合并

- 当删除一行数据时，其实不会立即删除，而是进行标记该空间已经被删除了可以被申请，当该页数据占比小于或等于 50% 时，innodb 就会寻找最近的页看是否可以将其进行合并。

主键的设计原则

1. 满足业务要求的情况下，主键的长度尽量小一些。
2. 插入时尽量顺数插入，是用 Auto_increment 自增。
3. 尽量避免对主键的操作。

3 order by 优化

- using filesort：通过表的索引或者全文扫描，读取满足条件的行，然后在排序缓存区进行排序操作，所有不是通过索引直接返回排序结果的排序都叫 filesort 排序。
- using index：通过有序索引顺序扫描直接返回有序数据，就是 using index，不需要额外的排序，效率高。

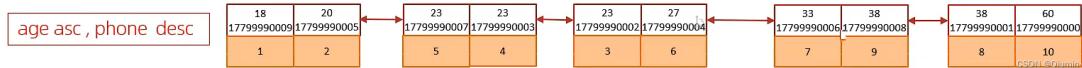
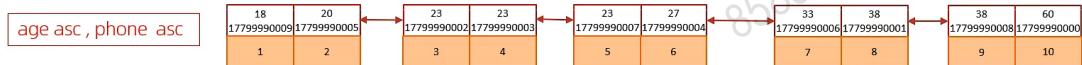
```
#通过没有创建索引的字段进行排序
```

```
select id,age,phone from user order by age,phone; #using filesort
```

```
#带索引排序
```

```
select id,age,phone from user order by age,phone; #using index, 效率高，同为升序或同为降序，没有指定默认为升序
```

图示：



说明：

- 根据排序字段建立连个索引，遵循最左前缀法则
- 尽量使用覆盖索引
- 要想一个升序一个降序，在创建索引时声明，默认为升序，如果没有声明一个升序一个降序还是会有额外的排序即 using filesort

```
create index idx_id_code on city (ID asc , CountryCode desc );
```

4 group by 优化

```
#创建索引  
create index idx_user_sex_name on user(sex,name);  
  
#分组  
select id name sex from user group by sex; #使用到了索引，效率高
```

遵循最左前缀法则，可以覆盖索引。

5 limit 优化

当数据量大时就需要用到分页，而分页越到后面就会越慢。如将 100000 到 100010 分为一页。这时需要用覆盖索引 加子查询 的办法来优化.

6 count 优化

count() 是一个聚合函数，一行一行的判断，如果 count 函数的参数不是 null，累计就

加 1，最后统计累计值

count 的几种用法

- count (主键) : innodb 引擎会遍历整张表，把每一行的主键 id 值都取出来，返回给服务层，服务层拿到主键后，直接按行进行累加（主键不可能为 null，所以不用判断）。
- count (字段) : 没有 not null 约束，会遍历整张表将该字段都取出返回服务层，服务层进行判断不为 null 就加 1；有 not null 约束时，遍历整张表取出该字段返回服务层，服务不需判断，直接按行累加。
- count (1) : 遍历整张表，但不取值，服务层对于返回的每一行放一个数字 1 进去，直接按行进行累加。
- count (*) : innodb 做了优化，不取值，服务层直接按行累加。

按查询效率排序: count (字段) < count(主键) < count(1)、count(*) 【尽量使用 count (*)】

7 update 优化

update 操作时，条件尽量为带有索引的字段，防止行锁升级为表锁

```
updata user set name='hha' where id=1
```

#由于主键有索引，所以此时是行锁，锁的是 id=1 这一行记录，其他线程可以操作 id!=1 的记录

```
update user set sex='女' where name='haha'
```

#由于 name 字段没有索引，所以此时锁的是整个表，其他线程此时都不能操作该表，并发性极低

innodb 的行锁是针对索引加的锁，不是针对记录加的锁，并且该索引不能失效，否则行锁也会升级为表锁

Mysql 优化

1.硬件优化：4核cpu，64位操作系统，32g内存，固态硬盘，组建磁盘阵列

(RAID0、RAID1、RAID5、RAID10、RAID50)

2.Mysql 自身：

a.最新的稳定版 Mysql

b.Sql 语句优化：

1.用 exist 替代 in

2.合理的构建索引

3.尽量去除表连接操作

4.尽量不要在 where 里面使用不等于符号，或者是进行 null 值判断，这会导致全表扫描

5.尽量减少操作到的列的数目，尽量明确写出要查询的列，少用 select *，可以使用覆盖索引，减少 IO 流

6.确保 group by 和 order by 中的表达式只涉及表中的一个列，确保 group by 和 order by 列上有索引

7.确保每条查询语句上都能用到索引，不用扫描全表

8.复杂 Sql 语句拆分成多个简单语句

总结：尽量使用索引，减少 IO 流

3.缓存数据库：在 redis、memcache 等中缓存部分数据

快照读与当前读

当前读，即读取最新提交的数据

- select ... for update
- insert、update、delete，都会按最新提交的数据进行操作

快照读，读取某一个快照建立时（可以理解为某一时间点）的数据

快照读主要体现在 select 时，不同隔离级别下，select 的行为不同

- 在 Serializable 隔离级别下 - 普通 select 也变成当前读
- 在 RC 隔离级别下 - 每次 select 都会建立新的快照
- 在 RR 隔离级别下
 - a) 事务启动后，首次 select 会建立快照
 - b) 如果事务启动选择了 with consistent snapshot，事务启动时就建立快照
 - c) 基于旧数据的修改操作，会重新建立快照

select into from 和 insert into select 的用法和区别

select into from 和 insert into select 都是用来复制表，两者的主要区别为：select into from 要求目标表不存在，因为在插入时会自动创建。insert into select from 要求目标表存在

下面分别介绍两者语法

一、INSERT INTO SELECT 语句

1、语句形式为：

Insert into Table2(field1,field2,...) select value1,value2,... from Table1

2、注意地方：

- (1) 要求目标表 Table2 必须存在，并且字段 field,field2...也必须存在
- (2) 注意 Table2 的主键约束，如果 Table2 有主键而且不为空，则 field1, field2... 中必须包括主键
- (3) 注意语法，不要加 values，和插入一条数据的 sql 混了，不要写成:Insert into Table2(field1,field2,...) values (select value1,value2,... from Table1)
- (4) 由于目标表 Table2 已经存在，所以我们除了插入源表 Table1 的字段外，还可以插入常量。

二、SELECT INTO FROM 语句

1、语句形式为：

```
SELECT vale1, value2 into Table2 from Table1
```

要求目标表 Table2 不存在，因为在插入时会自动创建表 Table2，并将 Table1 中指定字段数据复制到 Table2 中。

MySQL 数据库的备份与恢复

一 数据备份的重要性

- 1. 备份的主要目的是灾难恢复。
- 2. 在生产环境中，数据的安全性至关重要。
- 3. 任何数据的丢失都可能产生严重的后果。
- 4. 造成数据丢失的原因：①程序错误 ②人为操作错误 ③运算错误 ④灾难（如火灾，地震等）和盗窃

二 数据库备份的分类和备份策略

数据库备份的分类

1) 物理备份

对数据库操作系统的物理文件（如数据文件，日志文件等）的备份。

物理备份的方法：

- 1.冷备份（脱机备份）：是在关闭数据库的时候进行的。
- 2.热备份（联机备份）：数据库处于运行状态，依赖于数据库的日志文件。
- 3.温备份：数据库锁定表格（不可写入但可读）的状态下进行备份操作。

2) 逻辑备份

对数据库逻辑组件（如：表等数据库对象）的备份

- 1.以sql语句的形式，把库，表结构，表数据保存下来。

数据库的备份策略

- 完全备份（全量备份）：每次对数据库进行完整的备份。
- 差异备份：备份自从上次完全备份之后被修改过的文件。
- 增量备份：只有在上次完全备份或者增量备份后被修改的文件才会被备份。



三 常见的备份方法

物理冷备（完全备份）：

备份时数据库处于关闭状态，直接打包数据库文件

备份速度快，恢复时也是最简单的

专用备份工具 mydump 或 mysqlhotcopy （完全备份，逻辑备份）：

mysqlldump 常用的逻辑备份工具（导出为 sql 脚本）

mysqlhotcopy 仅拥有备份 MyISAM 和 ARCHIVE 表

启用二进制日志进行增量备份（增量备份）

进行增量备份，需要刷新二进制日志

第三方工具备份

免费的 MySQL 热备份软件 Percona XtraBackup（阿里云的工具：dts，支持热迁移）

四 MySQL 完全备份

- 完全备份是对整个数据库、数据库结构和文件结构的备份
- 保存的是备份完成时刻的数据库
- 是差异备份与增量备份的基础

完全备份的优缺点

优点：备份与恢复操作简单方便

缺点：①数据存在大量的重复 ②占用大量的备份空间 ③备份与恢复时间长

完全备份的方法

1) 物理冷备份与恢复

1.1 关闭 MySQL 数据库

1.2 使用 tar 命令直接打包数据库文件夹

1.3 直接替换现有 MySQL 目录即可

2) mysqldump 备份与恢复

2.1 MySQL 自带的备份工具，可方便实现对 MySQL 的备份

2.2 可以将指定的库、表导出为 SQL 脚本

2.3 使用命令 mysql 导入备份的数据

五 操作演示

物理冷备份

完全备份

先关闭数据库，之后打包备份

```
systemctl stop mysqld      #先关闭服务  
mkdir /backup/              #创建备份目录  
  
rpm -q xz                  #使用 xz 工具进行压缩，检查 xz 工具是否已安装  
yum install xz -y           #如果没安装，可以先 yum 安装  
tar Jcf /backup/mysql_all_$(date +%F).tar.xz /usr/local/mysql/data #打包数据库文  
件。/usr/local/mysql/data 为数据库文件存放目录  
  
cd /backup/                #切换到备份目录  
ls                         #查看目录内容  
tar tf mysql_all_时间.tar.xz #查看 tar 包内的文件
```

```
mysql> show databases;
+-----+
| Database      |
+-----+
| information_schema |
| hjh           |
| lhr           |
| mysql          |
| performance_schema |
| sys            |
+-----+
6 rows in set (0.01 sec)
```

查看当前有哪些数据库

CSDN @澄江映秋月

```
[root@localhost ~]# systemctl stop mysqld
[root@localhost ~]# mkdir /backup/
[root@localhost ~]# rpm -q xz
xz-5.2.2-1.el7.x86_64
[root@localhost ~]# tar Jcf /backup/mysql_all_$(date +%F).tar.xz /usr/local/mysql/data
tar: 从成员名中删除开头的"/"
[root@localhost ~]# cd /backup/
[root@localhost backup]# ls
mysql_all_2022-10-28.tar.xz
[root@localhost backup]# tar tf mysql_all_2022-10-28.tar.xz
/usr/local/mysql/data/
/usr/local/mysql/data/ibdata1
/usr/local/mysql/data/ib_logfile1
/usr/local/mysql/data/ib_logfile0
/usr/local/mysql/data/auto.cnf
/usr/local/mysql/data/mysql/
/usr/local/mysql/data/mysql/db.opt
/usr/local/mysql/data/mysql/db_error
```

关闭服务，创建备份目录

将数据库文件存放目录进行打包

查看tar包内的文件

CSDN @澄江映秋月

完全恢复

将数据库迁移到另一台主机，测试完全恢复。

#主机 A，使用 scp 命令将 tar 包传给另一台主机 B

```
scp /backup/mysql_all_2022-06-05.tar.xz 192.168.72.70:/opt
```

##主机 B 的操作##

```
systemctl stop mysqld #关闭 mysql
```

```
cd /opt/
```

```
mkdir /opt/bak/ #创建备份目录
```

```
tar Jxf mysql_all_2022-06-05.tar.xz -C /opt/bak/ #将 tar 包解压到备份目录
```

```
cd /opt/bak/ #切换到 tar 包的解压目录
```

```
\cp -af /usr/local/mysql/data/ /usr/local/mysql #将 data 目录复制到  
/usr/local/mysql/目录下，覆盖原有文件  
  
systemctl start mysqld #启动 mysql  
  
mysql -u root -p #登录数据库查看  
  
show databases;  
  
use yuji;  
  
show tables;  
  
select * from class;
```

逻辑备份（使用 mysqldump 工具）

mysqldump 是常用的逻辑备份工具。

mysqldump 可以将指定的库、表导出为 SQL 脚本。

mysqldump 完全备份

1) 完全备份一个或多个完整的库（包括库中所有的表）

```
mysqldump -uroot -p[密码] --databases 库名1 [库名2].. >/备份路径/备份文件  
名.sql#导出的就是数据库脚本文件
```

示例：

```
mysqldump -u root -p --databases kkk > /opt/mysql_bak/yuji.sql #完全备份一个  
库 kkk  
  
mysqldump -u root -p --databases kkk market > /opt/mysql_bak/kkk-market.sql  
#完全备份多个库，kkk 库和 market 库
```

2) 完全备份 MySQL 服务器中所有的库 (包括库中所有的表)

```
mysqldump -u root -p[密码] --all-databases > / 备份路径/备份文件名.sql
```

示例:

```
mysqldump -u root -p --all-databases > /opt/mysql_bak/all.sql #完全备份所有  
的库
```

3) 完全备份指定库中的部分表

注意: 只备份表, sql 语句中只有对表的操作, 没有对库的操作。恢复时需要人为确认库存在。

```
mysqldump -u root -p[密码] [-d] 库名 表名 1 [表名 2] ... > /备份路径/备份文件名.sql  
#使用 “-d”选项, 说明只保存数据库的表结构  
#不使用 “-d” 选项, 说明表数据也进行备份
```

4) 查看备份文件

备份文件中保存的是 sql 语句。即以 sql 语句的形式, 把库、表结构、表数据保存下来。

```
cd /opt/mysql_bak  
cat kkk-market.sql | grep -v "^\-\-" |grep -v "^\/" |grep -v "^\$"
```

mysqldump 完全恢复

先启动 mysql

```
systemctl start mysqld
```

1) 恢复数据库

先删除数据库, 之后进行恢复。

```
##删除数据库 kkk##  
mysql -u root -p -e 'drop database kkk;'
```

#"e"选项，用于指定连接 MySQL 后执行的命令，命令执行完后自动退出

```
mysql -u root -p -e 'SHOW DATABASES;'      #查看当前有哪些数据库
```

##恢复数据库 kkk##

```
mysql -u root -p < /opt/mysql_bak/kkk.sql    #重定向输入库文件
```

```
mysql -u root -p -e 'SHOW DATABASES;'      #查看当前有哪些数据库
```

2) 恢复数据表

当备份文件中只包含表的备份，而不包含创建的库的语句时，执行导入操作时必须指定库名，且目标库必须存在。

##备份 kkk 库中的 class 表##

```
mysqldump -uroot -p kkk class > /opt/mysql_bak/kkk_class.sql
```

##删除 kkk 库中的 class 表##

```
mysql -u root -p -e 'drop table kkk.class;'
```

```
mysql -u root -p -e 'show tables from kkk;'    #查看 yuji 库中的表，已无 class 表
```

##恢复 kkk 库中的 class 表##

```
mysql -u root -p kkk < /opt/mysql_bak/kkk_class.sql  #重定向导入备份文件，必须指定库名，且目标库必须存在
```

```
mysql -u root -p -e 'show tables from kkk;'
```

六 MySQL 增量备份与恢复

1. MySQL 增量备份介绍

使用 mysqldump 进行完全备份存在的问题

备份数据中有重复数据

备份时间与恢复时间过长

增量备份是什么：

是自上一次备份后增加/变化的文件或者内容

增量备份的特点

没有重复数据，备份量不大，时间短

恢复需要上次完全备份及完全备份之后所有的增量备份才能恢复，而且要对所有增量备份进行逐个反推恢复

2. MySQL 增量备份的方法

- MySQL 没有提供直接的增量备份方法
- 可通过 MySQL 提供的二进制日志间接实现增量备份
- MySQL 二进制日志对备份的意义
 - 二进制日志保存了所有更新或者可能更新数据库的操作
 - 二进制日志在启动 MySQL 服务器后开始记录，并在文件达到 max_binlog_size 所设置的大小或者接收到 flush logs 命令后重新 创建新的日志文件
 - 只需定时执行 flush logs 方法重新创建新的日志，生成二进制文件序列，并及时把这些日志保存到安全的地方就完成了一个时间段的增量备份

3. MySQL 数据库增量恢复

一般恢复

将所有备份的二进制日志内容全部恢复

基于位置恢复

数据库在某一时间点可能既有错误的操作也有正确的操作

可以基于精准的位置跳过错误的操作

基于时间点恢复

跳过某个发生错误的时间点实现数据恢复

七 增量备份与恢复 演示

1. 开启二进制日志功能

```
vim /etc/my.cnf

[mysqld]

log-bin=mysql-bin #开启二进制日志。如果使用相对路径，则保存在
/usr/local/mysql/data/目录下

binlog_format = MIXED #可选，指定二进制日志(binlog)的记录格式为 MIXED

server-id = 1
```

```
systemctl restart mysqld
```

```
ls -l /usr/local/mysql/data/mysql-bin.*
```

----- 以下是注释 -----

#二进制日志(binlog)有 3 种不同的记录格式:

STATEMENT (基于 SQL 语句)、ROW(基于行)、MIXED(混合模式)，默认格式是

STATEMENT

STATEMENT (基于 SQL 语句)：记录修改的 sql 语句。高并发的情况下，记录操作的 sql

语句时可能顺序会有错误，导致恢复数据时，数据丢失或有误差。效率高，但数据可能有误差。

ROW(基于行)：记录每一行数据，准确，但恢复时效率低。

MIXED(混合模式)：正常情况下使用 STATEMENT，高并发的情况下会智能地切换到

ROW。

2. 可每周对数据库或表进行完全备份

```
mysqldump -u root -p kkk class > /bak/kkk_class_$(date +%F).sql
```

```
mysqldump -u root -p --databases kkk > /bak/kkk_$(date +%F).sql
```

3. 可每天进行增量备份操作，生成新的二进制日志文件

```
mysqladmin -u root -p flush-logs
```

<https://github.com/daohonglei/javaStereotypedWriting>

<https://gitee.com/daohonglei/javaStereotypedWriting>