16. Explain详解（上）

执行计划输出中各列详解

table

不论我们的查询语句有多复杂，里边儿包含了多少个表，到最后也是需要对每个表进行单表访问的，所以设计MySQL的大叔规定EXPLAIN语句输出的每条记录都对应着某个单表的访问方法，该条记录的table列代表着该表的表名。

id

查询语句中每出现一个SELECT关键字，设计MySQL的大叔就会为它分配一个唯一的id值。这个id值就是EXPLAIN语句的第一个列

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+------
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filte
+----+-------------+-------+------------+------+---------------+------+---------+------+------+------
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100
+----+-------------+-------+------------+------+---------------+------+---------+------+------+------
2 rows in set, 1 warning (0.01 sec)
```

可以看到，上述连接查询中参与连接的 s1 和 s2 表分别对应一条记录，但是这两条记录对应的 id 值都是 1 。这里需要大家记住的是，在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的，出现在前边的表表示驱动表，出现在后边的表表示被驱动表。所以从上边的 EXPLAIN 输出中我们可以看出，查询优化器准备让 s1 表作为驱动表，让 s2 表作为被驱动表来执行查询。

```
mysql> EXPLAIN SELECT * FROM s1  UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+---------+------+------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+
3 rows in set, 1 warning (0.00 sec)
```

这个语句的执行计划的第三条记录是个什么鬼？为毛 `id` 值是 `NULL`，而且 `table` 列长的也怪怪的？大家别忘了 `UNION` 子句是干嘛用的，它会把多个查询的结果集合并起来并对结果集中的记录进行去重，怎么去重呢？ `MySQL` 使用的是内部的临时表。正如上边的查询计划中所示， `UNION` 子句是为了把 `id` 为 `1` 的查询和 `id` 为 `2` 的查询的结果集合并起来并去重，所以在内部创建了一个名为 `<union1, 2>` 的临时表（就是执行计划第三条记录的 `table` 列的名称）， `id` 为 `NULL` 表明这个临时表是为了合并两个查询的结果集而创建的。

跟 `UNION` 对比起来， `UNION ALL` 就不需要为最终的结果集进行去重，它只是单纯的把多个查询的结果集中的记录合并成一个并返回给用户，所以也就不需要使用临时表。所以在包含 `UNION ALL` 子句的查询的执行计划中，就没有那个 `id` 为 `NULL` 的记录，如下所示：

select_type

设计MySQL的大叔为每一个SELECT关键字代表的小查询都定义了一个称之为select_type的属性，意思是我们只要知道了某个小查询的select_type属性，就知道了这个小查询在整个大查询中扮演了一个什么角色

SIMPLE

查询语句中不包含UNION或者子查询的查询都算作是SIMPLE类型

需要大家注意的是，由于select_type为SUBQUERY的子查询会被物化，所以只需要执行一遍。

需要大家注意的是，select_type为DEPENDENT SUBQUERY的查询可能会被执行多次。

type

我们前边说过执行计划的一条记录就代表着MySQL对某个表的执行查询时的访问方法，其中的type列就表明了这个访问方法是个啥

- system

当表中只有一条记录并且该表使用的存储引擎的统计数据是精确的，比如MyISAM、Memory，那么对该表的访问方法就是system。

- const

这个我们前边唠叨过，就是当我们根据主键或者唯一二级索引列与常数进行等值匹配时，对单表的访问方法就是const

- eq_ref

在连接查询时，如果被驱动表是通过主键或者唯一二级索引列等值匹配的方式进行访问的（如果该主键或者唯一二级索引是联合索引的话，所有的索引列都必须进行等值比较），则对该被驱动表的访问方法就是eq_ref

- ref

当通过普通的二级索引列与常量进行等值匹配时来查询某个表，那么对该表的访问方法就可能是ref

- range

如果使用索引获取某些范围区间的记录，那么就可能使用到range访问方法，比如下边的这个查询：

- index

当我们可以使用索引覆盖，但需要扫描全部的索引记录时，该表的访问方法就是index

小贴士： 再一次强调，对于使用InnoDB存储引擎的表来说，二级索引的记录只包含索引列和主键列的值，而聚簇索引中包含用户定义的全部列以及一些隐藏列，所以扫描二级索引的代价比直接全表扫描，也就是扫描聚簇索引的代价更低一些。

- ALL

最熟悉的全表扫描

possible_keys和key

在EXPLAIN语句输出的执行计划中，possible_keys列表示在某个查询语句中，对某个表执行单表查询时可能用到的索引有哪些，key列表示实际用到的索引有哪些

key_len

key_len列表示当优化器决定使用某个索引执行查询时，该索引记录的最大长度

- 对于使用固定长度类型的索引列来说，它实际占用的存储空间的最大长度就是该固定值，对于指定字符集的变长类型的索引列来说，比如某个索引列的类型是VARCHAR(100)，使用的字符集是utf8，那么该列实际占用的最大存储空间就是100 × 3 = 300个字节。

- 如果该索引列可以存储NULL值，则key_len比不可以存储NULL值时多1个字节。
- 对于变长字段来说，都会有2个字节的空间来存储该变长列的实际长度。

ref

当使用索引列等值匹配的条件去执行查询时，也就是在访问方法是const、eq_ref、ref、ref_or_null、unique_subquery、index_subquery其中之一时，ref列展示的就是与索引列作等值匹配的东东是个啥，比如只是一个常数或者是某个列

rows

如果查询优化器决定使用全表扫描的方式对某个表执行查询时，执行计划的rows列就代表预计需要扫描的行数，如果使用索引来执行查询时，执行计划的rows列就代表预计扫描的索引记录行数。

filtered

之前在分析连接查询的成本时提出过一个condition filtering的概念，就是MySQL在计算驱动表扇出时采用的一个策略：

- 如果使用的是全表扫描的方式执行的单表查询，那么计算驱动表扇出时需要估计出满足搜索条件的记录到底有多少条。
- 如果使用的是索引执行的单表扫描，那么计算驱动表扇出的时候需要估计出满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND common_field = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+-
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+-
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+-
1 row in set, 1 warning (0.00 sec)
```

从执行计划的 key 列中可以看出来，该查询使用 idx_key1 索引来执行查询，从 rows 列可以看出满足 key1 > 'z' 的记录有 266 条。执行计划的 filtered 列就代表查询优化器预测在这 266 条记录中，有多少条记录满足其余的搜索条件，也就是 common_field = 'a' 这个条件的百分比。此处 filtered 列的值是 10.00 ，说明查询优化器预测在 266 条记录中有 10.00% 的记录满足 common_field = 'a' 这个条件。

17. Explain详解（下）

执行计划输出中各列详解

Extra

顾名思义，Extra列是用来说明一些额外信息的，我们可以通过这些额外信息来更准确的理解MySQL到底将如何执行给定的查询语句。

Json格式的执行计划

我们上边介绍的EXPLAIN语句输出中缺少了一个衡量执行计划好坏的重要属性 —— 成本。不过设计MySQL的大叔贴心的为我们提供了一种查看某个执行计划花费的成本的方式：

- 在EXPLAIN单词和真正的查询语句中间加上FORMAT=JSON。

这样我们就可以得到一个json格式的执行计划，里边儿包含该计划花费的成本

Extented EXPLAIN

最后，设计MySQL的大叔还为我们留了个彩蛋，在我们使用EXPLAIN语句查看了某个查询的执行计划后，紧接着还可以使用SHOW WARNINGS语句查看与这个查询的执行计划有关的一些扩展信息

18. optimizer trace 表的神奇功效

这个功能可以让我们方便的查看优化器生成执行计划的整个过程，这个功能的开启与关闭由系统变量optimizer_trace决定

优化过程大致分为了三个阶段：

- prepare阶段
- optimize阶段
- execute阶段

我们所说的基于成本的优化主要集中在optimize阶段，对于单表查询来说，我们主要关注optimize阶段的"rows_estimation"这个过程，这个过程深入分析了对单表查询的各种执行方案的成本；对于多表连接查询来说，我们更多需要关注"considered_execution_plans"这个过程，这个过程里会写明各种不同的连接方

式所对应的成本。反正优化器最终会选择成本最低的那种方案来作为最终的执行计划，也就是我们使用EXPLAIN语句所展现出的那种方案。

19. InnoDB 的 Buffer Pool

缓存的重要性

通过前边的唠叨我们知道，对于使用InnoDB作为存储引擎的表来说，不管是用于存储用户数据的索引（包括聚簇索引和二级索引），还是各种系统数据，都是以页的形式存放在表空间中的，而所谓的表空间只不过是InnoDB对文件系统上一个或几个实际文件的抽象，也就是说我们的数据说到底还是存储在磁盘上的。但是各位也都知道，磁盘的速度慢的跟乌龟一样，怎么能配得上“快如风，疾如电”的CPU呢？所以InnoDB存储引擎在处理客户端的请求时，当需要访问某个页的数据时，就会把完整的页的数据全部加载到内存中，也就是说即使我们只需要访问一个页的一条记录，那也需要先把整个页的数据加载到内存中。将整个页加载到内存中后就可以进行读写访问了，在进行完读写访问之后并不着急把该页对应的内存空间释放掉，而是将其缓存起来，这样将来有请求再次访问该页面时，就可以省去磁盘IO的开销了。

InnoDB的Buffer Pool

啥是个Buffer Pool

设计InnoDB的大叔为了缓存磁盘中的页，在MySQL服务器启动的时候就向操作系统申请了一片连续的内存，他们给这片内存起了个名，叫做Buffer Pool（中文名是缓冲池）。Buffer Pool也不能太小，最小值为5M(当小于该值时会自动设置成5M)。

Buffer Pool内部组成

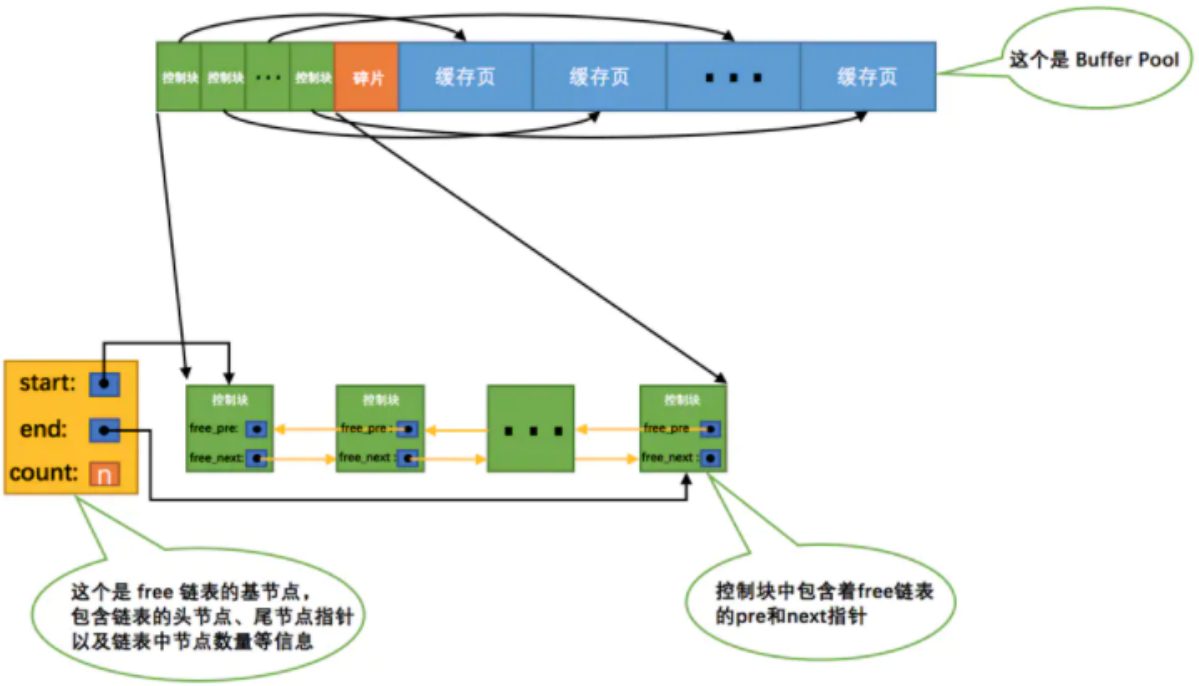
Buffer Pool中默认的缓存页大小和在磁盘上默认的页大小是一样的，都是16KB。

每个缓存页对应的控制信息占用的内存大小是相同的，我们就把每个页对应的控制信息占用的一块内存称为一个 控制块 吧，控制块和缓存页是一一对应的，它们都被存放到 Buffer Pool 中，其中控制块被存放到 Buffer Pool 的前边，缓存页被存放到 Buffer Pool 后边，所以整个 Buffer Pool 对应的内存空间看起来就是这样的：

小贴士：每个控制块大约占用缓存页大小的5%，在MySQL5.7.21这个版本中，每个控制块占用的大小是808字节。而我们设置的innodb_buffer_pool_size并不包含这部分控制块占用的内存空间大小，也就是说InnoDB在为Buffer Pool向操作系统申请连续的内存空间时，这片连续的内存空间一般会比innodb_buffer_pool_size的值大5%左右。

free链表的管理

当我们最初启动 MySQL 服务器的时候，需要完成对 Buffer Pool 的初始化过程，就是先向操作系统申请 Buffer Pool 的内存空间，然后把它划分成若干对控制块和缓存页。但是此时并没有真实的磁盘页被缓存到 Buffer Pool 中（因为还没有用到），之后随着程序的运行，会不断的有磁盘上的页被缓存到 Buffer Pool 中。那么问题来了，从磁盘上读取一个页到 Buffer Pool 中的时候该放到哪个缓存页的位置呢？或者说怎么区分 Buffer Pool 中哪些缓存页是空闲的，哪些已经被使用了呢？我们最好在某个地方记录一下Buffer Pool中哪些缓存页是可用的，这个时候缓存页对应的 控制块 就派上大用场了，我们可以把所有空闲的缓存页对应的控制块作为一个节点放到一个链表中，这个链表也可以被称作 free链表（或者说空闲链表）。刚刚完成初始化的 Buffer Pool 中所有的缓存页都是空闲的，所以每一个缓存页对应的控制块都会被加入到 free链表 中，假设该 Buffer Pool 中可容纳的缓存页数量为 n，那增加了 free链表 的效果图就是这样的：

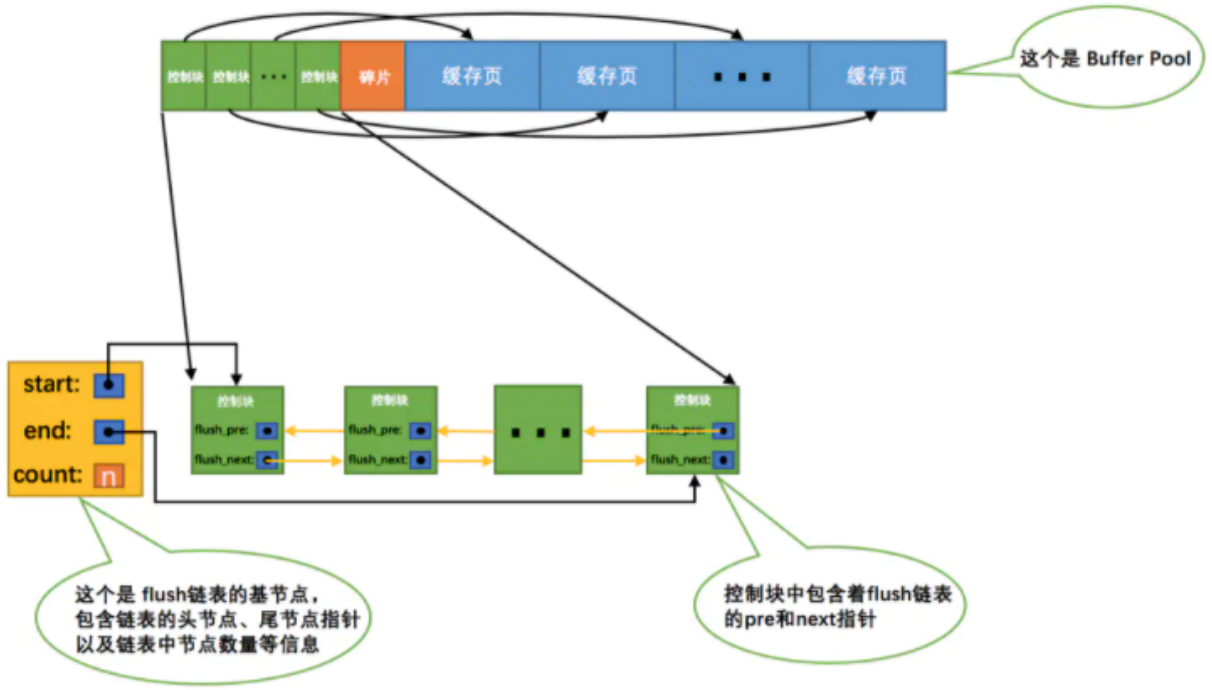


缓存页的哈希处理

我们可以用表空间号 + 页号作为key，缓存页作为value创建一个哈希表，在需要访问某个页的数据时，先从哈希表中根据表空间号 + 页号看看有没有对应的缓存页，如果有，直接使用该缓存页就好，如果没有，那就从free链表中选一个空闲的缓存页，然后把磁盘中对应的页加载到该缓存页的位置。

flush链表的管理

再创建一个存储脏页的链表，凡是修改过的缓存页对应的控制块都会作为一个节点加入到一个链表中，因为这个链表节点对应的缓存页都是需要被刷新到磁盘上的，所以也叫flush链表。链表的构造和free链表差不多，假设某个时间点Buffer Pool中的脏页数量为n，那么对应的flush链表就长这样：



LRU链表的管理

缓存不够的窘境

假设我们一共访问了n次页，那么被访问的页已经在缓存中的次数除以n就是所谓的缓存命中率，我们的期望就是让缓存命中率越高越好

简单的LRU（Least Recently Used）链表

- 如果该页不在Buffer Pool中，在把该页从磁盘加载到Buffer Pool中的缓存页时，就把该缓存页对应的控制块作为节点塞到链表的头部。
- 如果该页已经缓存在Buffer Pool中，则直接把该页对应的控制块移动到LRU链表的头部。

也就是说：只要我们使用到某个缓存页，就把该缓存页调整到LRU链表的头部，这样LRU链表尾部就是最近最少使用的缓存页喽～ 所以当Buffer Pool中的空闲缓存页使用完时，到LRU链表的尾部找些缓存页淘汰就OK啦

划分区域的LRU链表

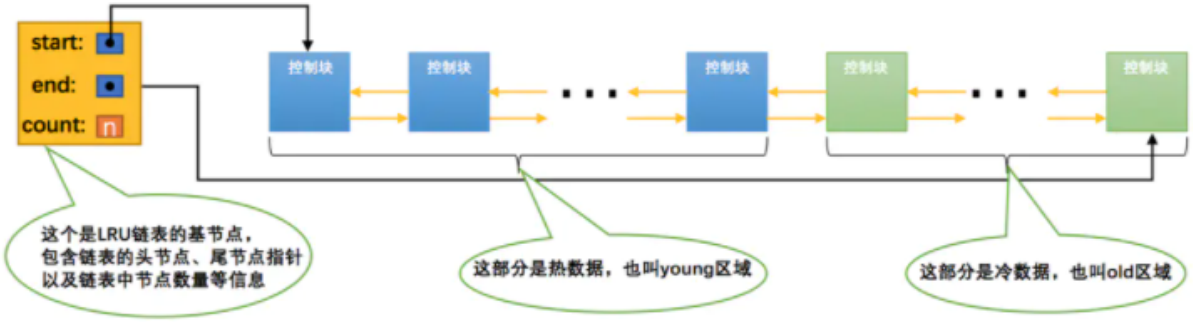
1）预读：1.线性预读　　2.随机预读

2）一些需要扫描全表的查询语句

因为有这两种情况的存在，所以设计InnoDB的大叔把这个LRU链表按照一定比例分成两截，分别是：

- 一部分存储使用频率非常高的缓存页，所以这一部分链表也叫做热数据，或者称young区域。
- 另一部分存储使用频率不是很高的缓存页，所以这一部分链表也叫做冷数据，或者称old区域。

LRU链表示意图



针对预读的页面可能不进行后续访问情况的优化：

当磁盘上的某个页面在初次加载到Buffer Pool中的某个缓存页时，该缓存页对应的控制块会被放到old区域的头部

针对全表扫描时，短时间内访问大量使用频率非常低的页面情况的优化

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，那么该页面就不会被从old区域移动到young区域的头部，否则将它移动到young区域的头部

更进一步优化LRU链表

比如只有被访问的缓存页位于young区域的1/4的后边，才会被移动到LRU链表头部，这样就可以降低调整LRU链表的频率，从而提升性能（也就是说如果某个缓存页对应的节点在young区域的1/4中，再次访问该缓存页时也不会将其移动到LRU链表头部）

刷新脏页到磁盘

- 从LRU链表的冷数据中刷新一部分页面到磁盘。
- 从flush链表中刷新一部分页面到磁盘。

Buffer Pool中存储的其它信息

Buffer Pool的缓存页除了用来缓存磁盘上的页面以外，还可以存储锁信息、自适应哈希索引等信息

总结

1. 磁盘太慢，用内存作为缓存很有必要。

2. Buffer Pool本质上是InnoDB向操作系统申请的一段连续的内存空间，可以通过innodb_buffer_pool_size来调整它的大小。

3. Buffer Pool向操作系统申请的连续内存由控制块和缓存页组成，每个控制块和缓存页都是一一对应的，在填充足够多的控制块和缓存页的组合后，Buffer Pool剩余的空间可能产生不够填充一组控制块和缓存页，这部分空间不能被使用，也被称为碎片。

4. InnoDB使用了许多链表来管理Buffer Pool。

5. free链表中每一个节点都代表一个空闲的缓存页，在将磁盘中的页加载到Buffer Pool时，会从free链表中寻找空闲的缓存页。

6. 为了快速定位某个页是否被加载到Buffer Pool，使用表空间号 + 页号作为key，缓存页作为value，建立哈希表。

7. 在Buffer Pool中被修改的页称为脏页，脏页并不是立即刷新，而是被加入到flush链表中，待之后的某个时刻同步到磁盘上。

8. LRU链表分为young和old两个区域，可以通过innodb_old_blocks_pct来调节old区域所占的比例。首次从磁盘上加载到Buffer Pool的页会被放到old区域的头部，在innodb_old_blocks_time间隔时间内访问该页不会把它移动到young区域头部。在Buffer Pool没有可用的空闲缓存页时，会首先淘汰掉old区域的一些页。

9. 我们可以通过指定innodb_buffer_pool_instances来控制Buffer Pool实例的个数，每个Buffer Pool实例中都有各自独立的链表，互不干扰。

10. 自MySQL 5.7.5版本之后，可以在服务器运行过程中调整Buffer Pool大小。每个Buffer Pool实例由若干个chunk组成，每个chunk的大小可以在服务器启动时通过启动参数调整。

11. 可以用下边的命令查看Buffer Pool的状态信息：

```
SHOW ENGINE INNODB STATUS\G
```

20. 事务简介

事务的基本要素：

原子性（Atomicity）、隔离性（Isolation）、一致性（Consistency）和持久性（Durability）

事务的概念

设计数据库的大叔根据这些操作所执行的不同阶段把事务大致上划分成了这么几个状态：

- 活动的（active）

事务对应的数据库操作正在执行过程中时，我们就说该事务处在活动的状态。

- 部分提交的（partially committed）

当事务中的最后一个操作执行完成，但由于操作都在内存中执行，所造成的影响并没有刷新到磁盘时，我们就说该事务处在部分提交的状态。

- 失败的（failed）

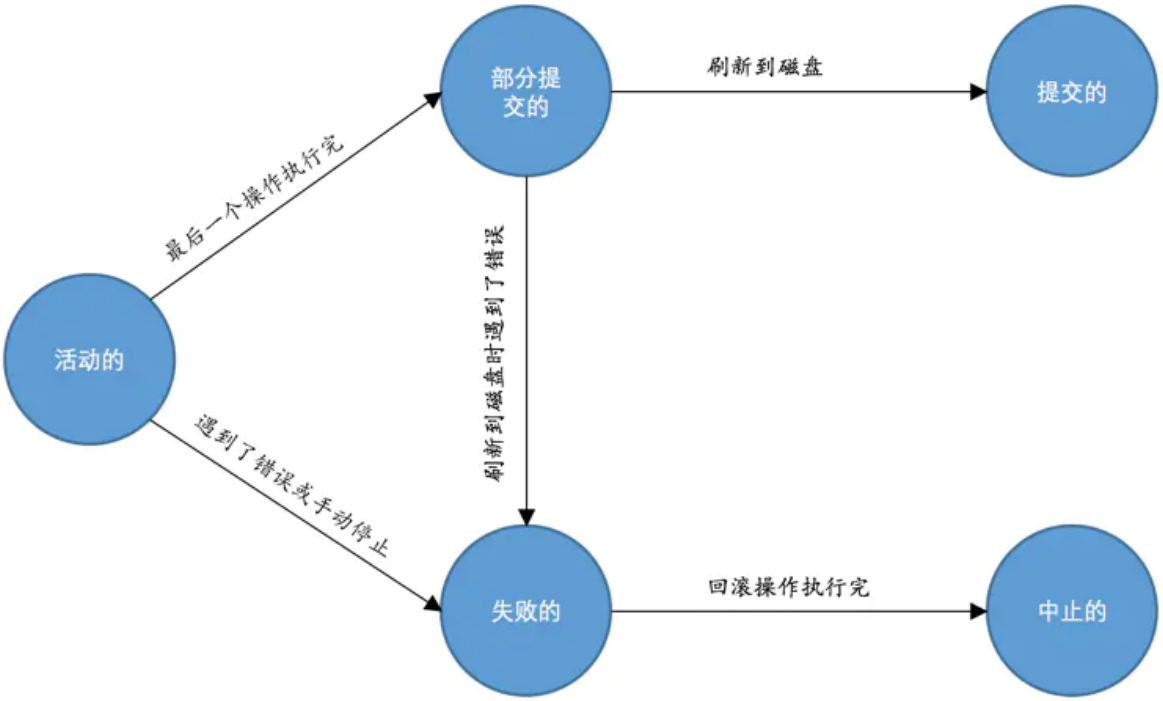
当事务处在活动的或者部分提交的状态时，可能遇到了某些错误（数据库自身的错误、操作系统错误或者直接断电等）而无法继续执行，或者人为的停止当前事务的执行，我们就说该事务处在失败的状态。

- 中止的（aborted）

如果事务执行了半截而变为失败的状态，比如我们前边唠叨的狗哥向猫爷转账的事务，当狗哥账户的钱被扣除，但是猫爷账户的钱没有增加时遇到了错误，从而当前事务处在了失败的状态，那么就需要把已经修改的狗哥账户余额调整为未转账之前的金额，换句话说，就是要撤销失败事务对当前数据库造成的影响。书面一点的话，我们把这个撤销的过程称之为回滚。当回滚操作执行完毕时，也就是数据库恢复到了执行事务之前的状态，我们就说该事务处在了中止的状态。

- 提交的（committed）

当一个处在部分提交的状态的事务将修改过的数据都同步到磁盘上之后，我们就可以说该事务处在了提交的状态。



只有当事务处于提交的或者中止的状态时，一个事务的生命周期才算是结束了。

MySQL中事务的语法

开启事务

- `BEGIN [WORK];`
- `START TRANSACTION;`

后面可以跟修饰符：

`1) READ ONLY`：标识当前事务是一个只读事务，也就是属于该事务的数据库操作只能读取数据，而不能修改数据。

`2) READ WRITE`：标识当前事务是一个读写事务，也就是属于该事务的数据库操作既可以读取数据，也可以修改数据。

`3) WITH CONSISTENT SNAPSHOT`：启动一致性读（先不用关心啥是个一致性读，后边的章节才会唠叨）。

提交事务

`COMMIT [WORK]`

`COMMIT`语句就代表提交一个事务，后边的`WORK`可有可无。

手动中止事务

`ROLLBACK [WORK]`

`ROLLBACK`语句就代表中止并回滚一个事务，后边的`WORK`可有可无类似的

支持事务的存储引擎

MySQL中并不是所有存储引擎都支持事务的功能，目前只有InnoDB和NDB存储引擎支持

自动提交

MySQL中有一个系统变量autocommit
可以看到它的默认值为ON，也就是说默认情况下，如果我们不显式的使用START TRANSACTION或者BEGIN语句开启一个事务，那么每一条语句都算是一个独立的事务，这种特性称之为事务的自动提交。

隐式提交

定义或修改数据库对象的数据定义语言
隐式使用或修改mysql数据库中的表
事务控制或关于锁定的语句
加载数据的语句
关于MySQL复制的一些语句

保存点

就是在事务对应的数据库语句中打几个点，我们在调用ROLLBACK语句时可以指定会滚到哪个点，而不是回到最初的原点。定义保存点的语法如下：

```
SAVEPOINT 保存点名称;
```

当我们想回滚到某个保存点时，可以使用下边这个语句（下边语句中的单词WORK和SAVEPOINT是可有可无的）：

```
ROLLBACK [WORK] TO [SAVEPOINT] 保存点名称;
```

21. redo日志（上）

重做日志(redo log)用来保证事务的持久性，即事务ACID中的D。Redo log的主要作用是用于数据库的崩溃恢复

- redo日志占用的空间非常小
- redo日志是顺序写入磁盘的

redo日志格式

redo日志通用结构

各个部分的详细释义如下：

- `type`：该条`redo`日志的类型。

  在`MySQL 5.7.21`这个版本中，设计`InnoDB`的大叔一共为`redo`日志设计了53种不同的类型，稍后会详细介绍不同类型的`redo`日志。

- `space ID`：表空间ID。

- `page number`：页号。

- `data`：该条`redo`日志的具体内容。

简单的redo日志类型

`redo`日志中只需要记录一下在某个页面的某个偏移量处修改了几个字节的值，具体被修改的内容是啥就好了，设计`InnoDB`的大叔把这种极其简单的`redo`日志称之为物理日志

复杂一些的redo日志类型

redo日志格式小结

redo日志会把事务在执行过程中对数据库所做的所有修改都记录下来，在之后系统崩溃重启后可以把事务所做的任何修改都恢复出来。

Mini-Transaction的概念

设计`MySQL`的大叔把对底层页面中的一次原子访问的过程称之为一个`Mini-Transaction`，简称`mtr`

22. redo日志（下）

redo日志文件格式

我们前边说过log buffer本质上是一片连续的内存空间，被划分成了若干个512字节大小的block。将log buffer中的redo日志刷新到磁盘的本质就是把block的镜像写入日志文件中，所以redo日志文件其实也是由若干个512字节大小的block组成。

每一组由mtr生成的redo日志都有一个唯一的LSN值与其对应，LSN值越小，说明redo日志产生的越早。

flush链表中的LSN

我们知道一个mtr代表一次对底层页面的原子访问，在访问过程中可能会产生一组不可分割的redo日志，在mtr结束时，会把这一组redo日志写入到log buffer中。除此之外，在mtr结束时还有一件非常重要的事情要做，就是把在mtr执行过程中可能修改过的页面加入到Buffer Pool的flush链表。

flush链表中的脏页是按照页面的第一次修改时间从大到小进行排序的。

flush链表中的脏页按照修改发生的时间顺序进行排序，也就是按照oldest_modification代表的LSN值进行排序，被多次更新的页面不会重复插入到flush链表中，但是会更新newest_modification属性的值。

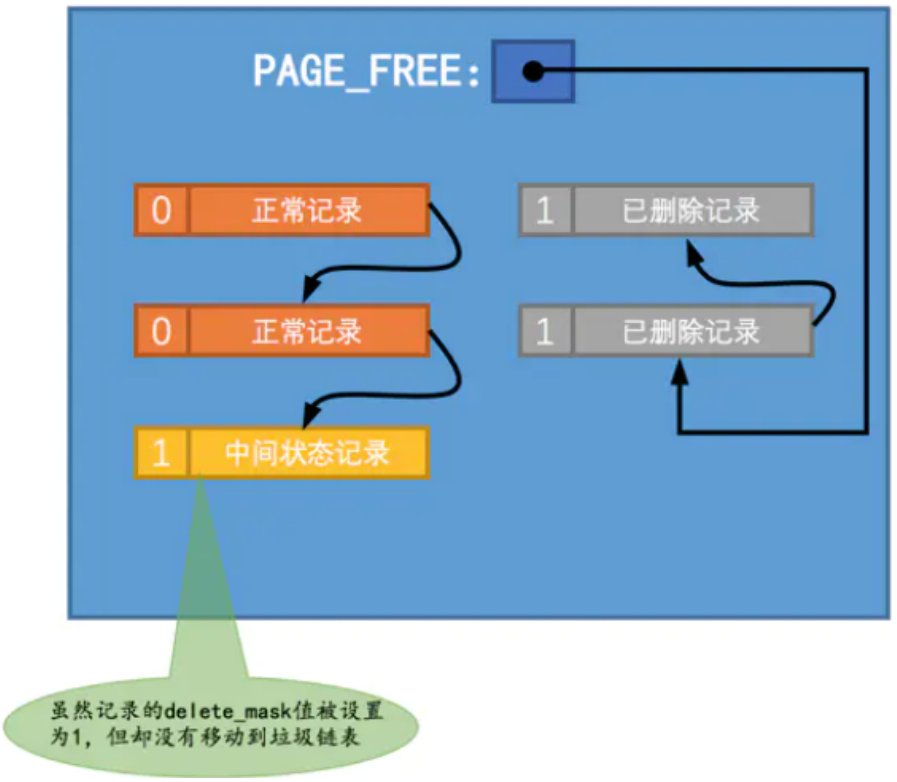
23. undo日志（上）

设计数据库的大叔把这些为了回滚而记录的这些东东称之为撤销日志，英文名为undo log，我们也可以土洋结合，称之为undo日志

DELETE操作对应的undo日志

- 阶段一：仅仅将记录的 `delete_mask` 标识位设置为 `1`，其他的不做修改（其实会修改记录的 `trx_id`、`roll_pointer` 这些隐藏列的值）。设计 `InnoDB` 的大叔把这个阶段称之为 `delete mark`。

  把这个过程画下来就是这样：



ps：中间状态是为了实现MVCC功能

- 阶段二：当该删除语句所在的事务提交之后，会有专门的线程后来真正的把记录删除掉。所谓真正的删除就是把该记录从正常记录链表中移除，并且加入到垃圾链表中，然后还要调整一些页面的其他信息（purge阶段）

UPDATE操作对应的undo日志

在执行`UPDATE`语句时，`InnoDB`对更新主键和不更新主键这两种情况有截然不同的处理方案。

不更新主键的情况

1）就地更新：更新后的列和更新前的列占用的存储空间都一样大

2）先删除掉旧记录，再插入新记录：更新后的列和更新前的列占用的存储空间不一样大

更新主键的情况（分两步）

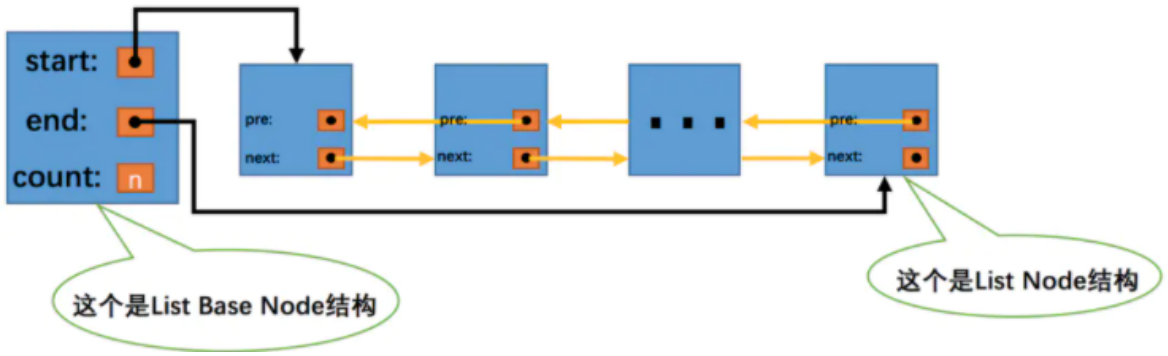
1）将旧记录进行delete mark操作

高能注意：这里是delete mark操作！这里是delete mark操作！这里是delete mark操作！也就是说在UPDATE语句所在的事务提交前，对旧记录只做一个delete mark操作，在事务提交后才由专门的线程做purge操作，把它加入到垃圾链表中。这里一定要和我们上边所说的在不更新记录主键值时，先真正删除旧记录，再插入新记录的方式区分开！

2）根据更新后各列的值创建一条新记录，并将其插入到聚簇索引中（需重新定位插入的位置）。

24. undo日志（下）

通用链表结构



多个事务中的Undo页面链表

为了尽可能提高undo日志的写入效率，不同事务执行过程中产生的undo日志需要被写入到不同的Undo页面链表中。

重用Undo页面

一个Undo页面链表是否可以被重用的条件：

1）该链表中只包含一个Undo页面。

2）该Undo页面已经使用的空间小于整个页面空间的3/4。

3）insert undo链表

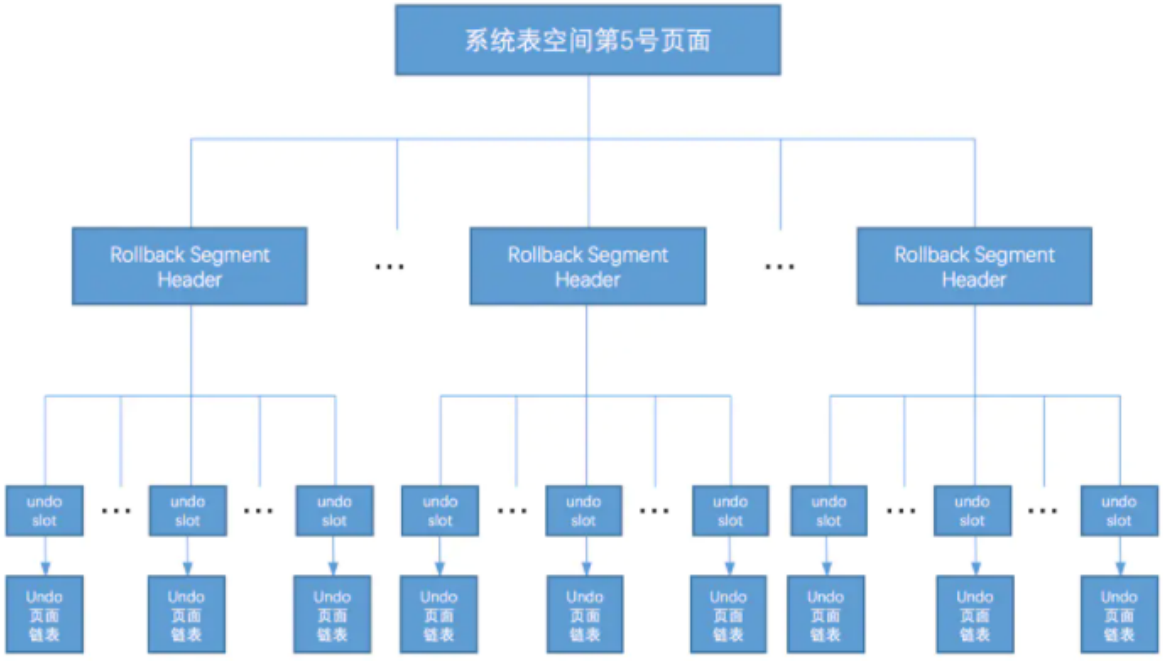
4）update undo链表

回滚段

回滚段的概念

我们现在知道一个事务在执行过程中最多可以分配4个 Undo页面 链表，在同一时刻不同事务拥有的 Undo页面 链表是不一样的，所以在同一时刻系统里其实可以有许许多多个 Undo页面 链表存在。为了更好的管理这些链表，设计 InnoDB 的大叔又设计了一个称之为 Rollback Segment Header 的页面，在这个页面中存放了各个 Undo页面 链表的 frist undo page 的 页号，他们把这些 页号 称之为 undo slot 。我们可以这样理解，每个 Undo页面 链表都相当于是一个班，这个链表的 first undo page 就相当于这个班的班长，找到了这个班的班长，就可以找到班里的其他同学（其他同学相当于 normal undo page ）。有时候学校需要向这些班级传达一下精神，就需要把班长都召集在会议室，这个 Rollback Segment Header 就相当于是一个会议室。

在系统表空间的第5号页面中存储了128个Rollback Segment Header页面地址，每个Rollback Segment Header就相当于一个回滚段。在Rollback Segment Header页面中，又包含1024个undo slot，每个undo slot都对应一个Undo页面链表。我们画个示意图：



总结一下针对普通表和临时表划分不同种类的回滚段的原因：在修改针对普通表的回滚段中的Undo页面时，需要记录对应的redo日志，而修改针对临时表的回滚段中的Undo页面时，不需要记录对应的redo日志。

如果一个事务在执行过程中既对普通表的记录做了改动，又对临时表的记录做了改动，那么需要为这个记录分配2个回滚段。并发执行的不同事务其实也可以被分配相同的回滚段，只要分配不同的undo slot就可以了。

25. 事务隔离级别和MVCC

事务并发执行遇到的问题

- 脏写（`Dirty Write`）

如果一个事务修改了另一个未提交事务修改过的数据，那就意味着发生了脏写

- 脏读（`Dirty Read`）

如果一个事务读到了另一个未提交事务修改过的数据，那就意味着发生了脏读

- 不可重复读（Non-Repeatable Read）

如果一个事务只能读到另一个已经提交的事务修改过的数据，并且其他事务每对该数据进行一次修改并提交后，该事务都能查询得到最新值，那就意味着发生了不可重复读

- 幻读（Phantom）

如果一个事务先根据某些条件查询出一些记录，之后另一个事务又向表中插入了符合这些条件的记录，原先的事务再次按照该条件查询时，能把另一个事务插入的记录也读出来，那就意味着发生了幻读

（幻读强调的是一个事务按照某个相同条件多次读取记录时，后读取时读到了之前没有读到的记录。）

SQL标准中的四种隔离级别

这些问题按照严重性来排一下序：`脏写 > 脏读 > 不可重复读 > 幻读`

4个隔离级别：

- `READ UNCOMMITTED`：未提交读。
- `READ COMMITTED`：已提交读。
- `REPEATABLE READ`：可重复读。
- `SERIALIZABLE`：可串行化。

SQL标准 中规定，针对不同的隔离级别，并发事务可以发生不同严重程度的问题，具体情况如下：

隔离级别	脏读	不可重复读	幻读
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible

MySQL中支持的四种隔离级别

MySQL的默认隔离级别为REPEATABLE READ

如果我们在服务器启动时想改变事务的默认隔离级别，可以修改启动参数transaction-isolation的值

如何设置事务的隔离级别

我们可以通过下边的语句修改事务的隔离级别：

```
SET [GLOBAL|SESSION] TRANSACTION ISOLATION LEVEL level;
```

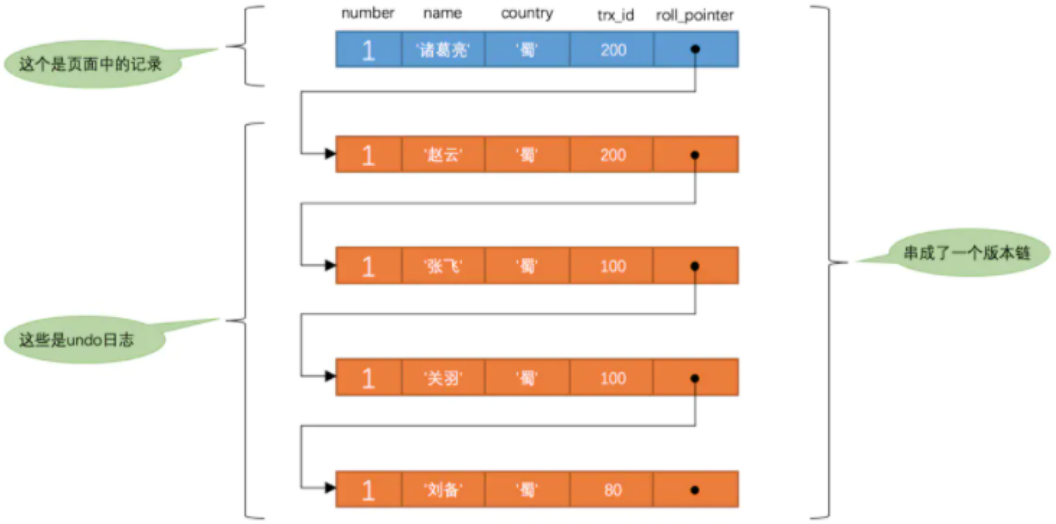
MVCC原理

版本链

我们前边说过，对于使用 InnoDB 存储引擎的表来说，它的聚簇索引记录中都包含两个必要的隐藏列（ row_id 并不是必要的，我们创建的表中有主键或者非NULL的UNIQUE键时都不会包含 row_id 列）：

- trx_id：每次一个事务对某条聚簇索引记录进行改动时，都会把该事务的 事务id 赋值给 trx_id 隐藏列。
- roll_pointer：每次对某条聚簇索引记录进行改动时，都会把旧的版本写入到 undo日志 中，然后这个隐藏列就相当于一个指针，可以通过它来找到该记录修改前的信息。

每次对记录进行改动，都会记录一条 `undo日志` ，每条 `undo日志` 也都有一个 `roll_pointer` 属性（ `INSERT` 操作对应的 `undo日志` 没有该属性，因为该记录并没有更早的版本），可以将这些 `undo日志` 都连起来，串成一个链表，所以现在的情况就像下图一样：



对该记录每次更新后，都会将旧值放到一条 `undo日志` 中，就算是该记录的一个旧版本，随着更新次数的增多，所有的版本都会被 `roll_pointer` 属性连接成一个链表，我们把这个链表称之为 `版本链` ，**版本链的头节点就是当前记录最新的值**。另外，每个版本中还包含生成该版本时对应的 `事务id` ，这个信息很重要，我们稍后就会用到。

ReadView

对于使用 `READ UNCOMMITTED` 隔离级别的事务来说，由于可以读到未提交事务修改过的记录，所以直接读取记录的最新版本就好了；对于使用 `SERIALIZABLE` 隔离级别的事务来说，设计 `InnoDB` 的大叔规定使用加锁的方式来访问记录（加锁是啥我们后续文章中说哈）；对于使用 `READ COMMITTED` 和 `REPEATABLE READ` 隔离级别的事务来说，都必须保证读到已经提交了的事务修改过的记录，也就是说假如另一个事务已经修改了记录但是尚未提交，是不能直接读取最新版本的记录的，核心问题就是：需要判断一下版本链中的哪个版本是当前事务可见的。为此，设计 `InnoDB` 的大叔提出了一个 `ReadView` 的概念，这个 `ReadView` 中主要包含4个比较重要的内容：

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

> 小贴士：注意max_trx_id并不是m_ids中的最大值，事务id是递增分配的。比方说现在有id为1，2，3这三个事务，之后id为3的事务提交了。那么一个新的读事务在生成ReadView时，m_ids就包括1和2，min_trx_id的值就是1，max_trx_id的值就是4。

- `creator_trx_id` ：表示生成该 `ReadView` 的事务的 `事务id` 。

有了这个 `ReadView` ，这样在访问某条记录时，只需要按照下边的步骤判断记录的某个版本是否可见：

- 如果被访问版本的 `trx_id` 属性值与 `ReadView` 中的 `creator_trx_id` 值相同，意味着当前事务在访问它自己修改过的记录，所以该版本可以被当前事务访问。
- 如果被访问版本的 `trx_id` 属性值小于 `ReadView` 中的 `min_trx_id` 值，表明生成该版本的事务在当前事务生成 `ReadView` 前已经提交，所以该版本可以被当前事务访问。
- 如果被访问版本的 `trx_id` 属性值大于或等于 `ReadView` 中的 `max_trx_id` 值，表明生成该版本的事务在当前事务生成 `ReadView` 后才开启，所以该版本不可以被当前事务访问。
- 如果被访问版本的 `trx_id` 属性值在 `ReadView` 的 `min_trx_id` 和 `max_trx_id` 之间，那就需要判断一下 `trx_id` 属性值是不是在 `m_ids` 列表中，如果在，说明创建 `ReadView` 时生成该版本的事务还是活跃的，该版本不可以被访问；如果不在，说明创建 `ReadView` 时生成该版本的事务已经被提交，该版本可以被访问。

如果某个版本的数据对当前事务不可见的话，那就顺着版本链找到下一个版本的数据，继续按照上边的步骤判断可见性，依此类推，直到版本链中的最后一个版本。如果最后一个版本也不可见的话，那么就意味着该条记录对该事务完全不可见，查询结果就不包含该记录。

READ COMMITTED和REPEATABLE READ隔离级别的的一个非常大的区别就是它们生成ReadView的时机不同。

使用READ COMMITTED隔离级别的事务在每次查询开始时都会生成一个独立的ReadView。

REPEATABLE READ —— 在第一次读取数据时生成一个ReadView

对于使用REPEATABLE READ隔离级别的事务来说，只会在第一次执行查询语句时生成一个ReadView，之后的查询就不会重复生成了

MVCC小结

从上边的描述中我们可以看出来，所谓的MVCC（Multi-Version Concurrency Control ，多版本并发控制）指的就是在使用READ COMMITTD、REPEATABLE READ这两种隔离级别的事务在执行普通的SELECT操作时访问记录的版本链的过程，这样子可以使不同事务的读-写、写-读操作并发执行，从而提升系统性能。READ COMMITTD、REPEATABLE READ这两个隔离级别的一个很大不同就是：生成ReadView的时机不同，READ COMMITTD在每一次进行普通SELECT操作前都会生成一个ReadView，而REPEATABLE READ只在第一次进行普通SELECT操作前生成一个ReadView，之后的查询操作都重复使用这个ReadView就好了。

关于purge

大家有没有发现两件事儿：

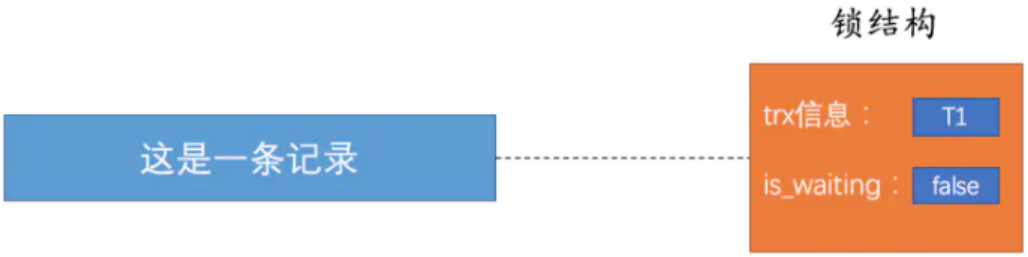
- 我们说 insert undo 在事务提交之后就可以被释放掉了，而 update undo 由于还需要支持 MVCC ，不能立即删除掉。
- 为了支持 MVCC ，对于 delete mark 操作来说，仅仅是在记录上打一个删除标记，并没有真正将它删除掉。

随着系统的运行，在确定系统中包含最早产生的那个 ReadView 的事务不会再访问某些 update undo日志 以及被打了删除标记的记录后，有一个后台运行的 purge线程 会把它们真正的删除掉。关于更多的

26. 锁

解决并发事务带来问题的两种基本方式

当一个事务想对这条记录做改动时，首先会看看内存中有没有与这条记录关联的 `锁结构` ，当没有的时候就会在内存中生成一个 `锁结构` 与之关联。比方说事务 `T1` 要对这条记录做改动，就需要生成一个 `锁结构` 与之关联：

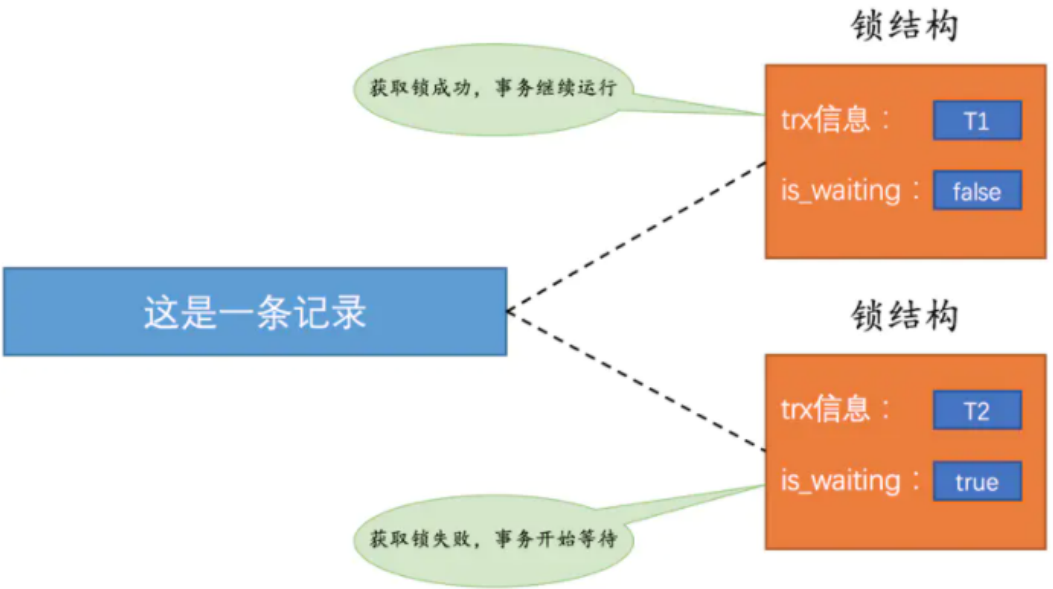


其实在 `锁结构` 里有很多信息，不过为了简化理解，我们现在只把两个比较重要的属性拿了出来：

- `trx信息` ：代表这个锁结构是哪个事务生成的。
- `is_waiting` ：代表当前事务是否在等待。

如图所示，当事务 `T1` 改动了这条记录后，就生成了一个 `锁结构` 与该记录关联，因为之前没有别的事务为这条记录加锁，所以 `is_waiting` 属性就是 `false` ，我们把这个场景就称之为获取锁成功，或者加锁成功，然后就可以继续执行操作了。

在事务 `T1` 提交之前，另一个事务 `T2` 也想对该记录做改动，那么先去看看有没有 `锁结构` 与这条记录关联，发现有一个 `锁结构` 与之关联后，然后也生成了一个 `锁结构` 与这条记录关联，不过 `锁结构` 的 `is_waiting` 属性值为 `true` ，表示当前事务需要等待，我们把这个场景就称之为获取锁失败，或者加锁失败，或者没有成功的获取到锁，画个图表示就是这样：



在事务 T1 提交之后，就会把该事务生成的 锁结构 释放掉，然后看看还有没有别的事务在等待获取锁，发现了事务 T2 还在等待获取锁，所以把事务 T2 对应的锁结构的 is_waiting 属性设置为 false，然后把该事务对应的线程唤醒，让它继续执行，此时事务 T2 就算获取到锁了。效果图就是这样：

读-写或写-读情况：也就是一个事务进行读取操作，另一个进行改动操作。

我们前边说过，这种情况下可能发生脏读、不可重复读、幻读的问题。

> 小贴士：幻读问题的产生是因为某个事务读了一个范围的记录，之后别的事务在该范围内插入了新记录，该事务再次读取该范围的记录时，可以读到新插入的记录，所以幻读问题准确的说并不是因为读取和写入一条相同记录而产生的，这一点要注意一下。

不过各个数据库厂商对 SQL标准 的支持都可能不一样，与 SQL标准 不同的一点就是，MySQL 在 REPEATABLE READ 隔离级别实际上就已经解决了 幻读 问题。

怎么解决 脏读 、 不可重复读 、 幻读 这些问题呢？其实有两种可选的解决方案：

- 方案一：读操作利用多版本并发控制（MVCC），写操作进行 加锁 。

  所谓的 MVCC 我们在前一章有过详细的描述，就是通过生成一个 ReadView，然后通过 ReadView 找到符合条件的记录版本（历史版本是由 undo日志 构建的），其实就像是在生成 ReadView 的那个时刻做了一次时间静止（就像用相机拍了一个快照），查询语句只能读到在生成 ReadView 之前已提交事务所做的更改，在生成 ReadView 之前未提交的事务或者之后才开启的事务所做的更改是看不到的。而写操作肯定针对的是最新版本的记录，读记录的历史版本和改动记录的最新版本本身并不冲突，也就是采用 MVCC 时， 读-写 操作并不冲突。

> 小贴士： 我们说过普通的SELECT语句在READ COMMITTED和REPEATABLE READ隔离级别下会使用到MVCC读取记录。在READ COMMITTED隔离级别下，一个事务在执行过程中每次执行SELECT操作时都会生成一个ReadView，ReadView的存在本身就保证了事务不可以读取到未提交的事务所做的更改，也就是避免了脏读现象；REPEATABLE READ隔离级别下，一个事务在执行过程中只有第一次执行SELECT操作才会生成一个ReadView，之后的SELECT操作都复用这个ReadView，这样也就避免了不可重复读和幻读的问题。

- 方案二：读、写操作都采用 `加锁` 的方式。

  如果我们的一些业务场景不允许读取记录的旧版本，而是每次都必须去读取记录的最新版本，比方在银行存款的事务中，你需要先把账户的余额读出来，然后将其加上本次存款的数额，最后再写到数据库中。在将账户余额读取出来后，就不想让别的事务再访问该余额，直到本次存款事务执行完成，其他事务才可以访问账户的余额。这样在读取记录的时候也就需要对其进行 `加锁` 操作，这样也就意味着 `读` 操作和 `写` 操作也像 `写-写` 操作那样排队执行。

  > 小贴士： 我们说脏读的产生是因为当前事务读取了另一个未提交事务写的一条记录，如果另一个事务在写记录的时候就给这条记录加锁，那么当前事务就无法继续读取该记录了，所以也就不会有脏读问题的产生了。不可重复读的产生是因为当前事务先读取一条记录，另外一个事务对该记录做了改动之后并提交之后，当前事务再次读取时会获得不同的值，如果在当前事务读取记录时就给该记录加锁，那么另一个事务就无法修改该记录，自然也不会发生不可重复读了。我们说幻读问题的产生是因为当前事务读取了一个范围的记录，然后另外的事务向该范围内插入了新记录，当前事务再次读取该范围的记录时发现了新插入的新记录，我们把新插入的那些记录称之为幻影记录。采用加锁的方式解决幻读问题就有那么一丢丢麻烦了，因为当前事务在第一次读取记录时那些幻影记录并不存在，所以读取的时候加锁就有点尴尬 —— 因为你并不知道给谁加锁，没关系，这难不倒设计InnoDB的大叔的，我们稍后揭晓答案，稍安勿躁。

很明显，采用MVCC方式的话，读-写操作彼此并不冲突，性能更高，采用加锁方式的话，读-写操作彼此需要排队执行，影响性能。一般情况下我们当然愿意采用MVCC来解决读-写操作并发执行的问题，但是业务在某些特殊情况下，要求必须采用加锁的方式执行，那也是没有办法的事。

一致性读（Consistent Reads）

事务利用MVCC进行的读取操作称之为一致性读，或者一致性无锁读，有的地方也称之为快照读。所有普通的SELECT语句（`plain SELECT`）在READ COMMITTED、REPEATABLE

READ隔离级别下都算是一致性读

锁定读（Locking Reads）

共享锁和独占锁

我们前边说过，并发事务的 读-读 情况并不会引起什么问题，不过对于 写-写 、 读-写 或 写-读 这些情况可能会引起一些问题，需要使用 MVCC 或者 加锁 的方式来解决它们。在使用 加锁 的方式解决问题时，由于既要允许 读-读 情况不受影响，又要使 写-写 、 读-写 或 写-读 情况中的操作相互阻塞，所以设计 MySQL 的大叔给锁分了个类：

- 共享锁，英文名： Shared Locks ，简称 S锁 。在事务要读取一条记录时，需要先获取该记录的 S锁 。
- 独占锁，也常称 排他锁 ，英文名： Exclusive Locks ，简称 X锁 。在事务要改动一条记录时，需要先获取该记录的 X锁 。

假如事务 T1 首先获取了一条记录的 S锁 之后，事务 T2 接着也要访问这条记录：

- 如果事务 T2 想要再获取一个记录的 S锁 ，那么事务 T2 也会获得该锁，也就意味着事务 T1 和 T2 在该记录上同时持有 S锁 。
- 如果事务 T2 想要再获取一个记录的 X锁 ，那么此操作会被阻塞，直到事务 T1 提交之后将 S锁 释放掉。

所以我们说 S锁 和 S锁 是兼容的， S锁 和 X锁 是不兼容的， X锁 和 X锁 也是不兼容的，画个表表示一下就是这样：

兼容性	X	S
X	不兼容	不兼容
S	不兼容	兼容

锁定读的语句

我们前边说在采用 加锁 方式解决 脏读 、 不可重复读 、 幻读 这些问题时，读取一条记录时需要获取一下该记录的 S锁 ，其实这是不严谨的，有时候想在读取记录时就获取记录的 X锁 ，来禁止别的事务读写该记录，为此设计 MySQL 的大叔提出了两种比较特殊的 SELECT 语句格式：

- 对读取的记录加 S锁 ：

```
SELECT ... LOCK IN SHARE MODE;
```

也就是在普通的 SELECT 语句后边加 LOCK IN SHARE MODE ，如果当前事务执行了该语句，那么它会为读取到的记录加 S锁 ，这样允许别的事务继续获取这些记录的 S锁 （比方说别的事务也使用 SELECT ... LOCK IN SHARE MODE 语句来读取这些记录），但是不能获取这些记录的 X锁 （比方说使用 SELECT ... FOR UPDATE 语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的 X锁 ，那么它们会阻塞，直到当前事务提交之后将这些记录上的 S锁 释放掉。

- 对读取的记录加 X锁 ：

```
SELECT ... FOR UPDATE;
```

也就是在普通的 SELECT 语句后边加 FOR UPDATE ，如果当前事务执行了该语句，那么它会为读取到的记录加 X锁 ，这样既不允许别的事务获取这些记录的 S锁 （比方说别的事务使用 SELECT ... LOCK IN SHARE MODE 语句来读取这些记录），也不允许获取这些记录的 X锁 （比如说使用 SELECT ... FOR UPDATE 语句来读取这些记录，或者直接修改这些记录）。如果别的事务想要获取这些记录的 S锁 或者 X锁 ，那么它们会阻塞，直到当前事务提交之后将这些记录上的 X锁 释放掉。

- INSERT：

一般情况下，新插入一条记录的操作并不加锁，设计InnoDB的大叔通过一种称之为隐式锁的东东来保护这条新插入的记录在本事务提交前不被别的事务访问，更多细节我们后边看哈～

多粒度锁

我们前边提到的锁都是针对记录的，也可以被称之为行级锁或者行锁，对一条记录加锁影响的也只是这条记录而已，我们就说这个锁的粒度比较细；其实一个事务也可以在表级别进行加锁，自然就被称之为表级锁或者表锁，对一个表加锁影响整个表中的记录，我们就说这个锁的粒度比较粗。给表加的锁也可以分为共享锁（S锁）和独占锁（X锁）：

- 意向共享锁，英文名：Intention Shared Lock，简称IS锁。当事务准备在某条记录上加S锁时，需要先在表级别加一个IS锁。

- 意向独占锁，英文名：Intention Exclusive Lock，简称IX锁。当事务准备在某条记录上加X锁时，需要先在表级别加一个IX锁。

总结一下：IS、IX锁是表级锁，它们的提出仅仅为了在之后加表级别的S锁和X锁时可以快速判断表中的记录是否被上锁，以避免用遍历的方式来查看表中有没有上锁的记录，也就是说其实IS锁和IX锁是兼容的，IX锁和IX锁是兼容的。我们画个表来看一下表级别的各种锁的兼容性：

MySQL中的行锁和表锁

其他存储引擎中的锁

对于MyISAM、MEMORY、MERGE这些存储引擎来说，它们只支持表级锁，而且这些引擎并不支持事务，所以使用这些存储引擎的锁一般都是针对当前会话来说的。比方说在Session 1中对一个表执行SELECT操作，就相当于为这个表加了一个表级别的S锁，如果在SELECT操作未完成时，Session 2中对这个表执行UPDATE操作，相当于要获取表的X锁，此操作会被阻塞，直到Session 1中的SELECT操作完成，释放掉表级别的S锁后，Session 2中对这个表执行UPDATE操作才能继续获取X锁，然后执行具体的更新语句。

InnoDB存储引擎中的锁

InnoDB存储引擎既支持表锁，也支持行锁。表锁实现简单，占用资源较少，不过粒度很粗，有时候你仅仅需要锁住几条记录，但使用表锁的话相当于为表中的所有记录都加锁，所以性能比较差。行锁粒度更细，可以实现更精准的并发控制。下边我们详细看一下。

InnoDB中的表级锁

- 表级别的S锁、X锁

在对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，InnoDB存储引擎是不会为这个表添加表级别的S锁或者X锁的。

另外，在对某个表执行一些诸如ALTER TABLE、DROP TABLE这类的DDL语句时，其他事务对这个表并发执行诸如SELECT、INSERT、DELETE、UPDATE的语句会发生阻塞，同理，某个事务中对某个表执行SELECT、INSERT、DELETE、UPDATE语句时，在其他会话中对这个表执行DDL语句也会发生阻塞。这个过程其实是通过在server层使用一

种称之为元数据锁（英文名：Metadata Locks，简称MDL）东东来实现的，一般情况下也不会使用InnoDB存储引擎自己提供的表级别的S锁和X锁。

InnoDB中的行级锁

行锁，也称为记录锁，顾名思义就是在记录上加的锁。不过设计InnoDB的大叔很有才，一个行锁玩出了各种花样，也就是把行锁分成了各种类型。换句话说即使对同一条记录加行锁，如果类型不同，起到的功效也是不同的。

Record Locks：

我们前边提到的记录锁就是这种类型，也就是仅仅把一条记录锁上

Gap Locks：

我们说MySQL在REPEATABLE READ隔离级别下是可以解决幻读问题的，解决方案有两种，可以使用MVCC方案解决，也可以采用加锁方案解决。但是在使用加锁方案解决时有个大问题，就是事务在第一次执行读取操作时，那些幻影记录尚不存在，我们无法给这些幻影记录加上正经记录锁。不过这难不倒设计InnoDB的大叔，他们提出了一种称之为Gap Locks的锁，官方的类型名称为：LOCK_GAP，我们也可以简称为gap锁

这个gap锁的提出仅仅是为了防止插入幻影记录而提出的

Next-Key Locks：

有时候我们既想锁住某条记录，又想阻止其他事务在该记录前边的间隙插入新记录，所以设计InnoDB的大叔们就提出了一种称之为Next-Key Locks的锁，官方的类型名称为：LOCK_ORDINARY，我们也可以简称为next-key锁。

Insert Intention Locks：

我们说一个事务在插入一条记录时需要判断一下插入位置是不是被别的事务加了所谓的gap锁（next-key锁也包含gap锁，后边就不强调了），如果有的话，插入操作需要等待，直到拥有gap锁的那个事务提交。但是设计InnoDB的大叔规定事务在等待的时候也需要在内存中生成一个锁结构，表明有事务想在某个间隙中插入新记录，但是现在在等待。设计InnoDB的大叔就把这种类型的锁命名为Insert Intention Locks，官方的类型名称为：LOCK_INSERT_INTENTION，我们也可以称为插入意向锁。

隐式锁

一个事务对新插入的记录可以不显式的加锁（生成一个锁结构），但是由于事务id这个牛逼的东东的存在，相当于加了一个隐式锁。别的事务在对这条记录加S锁或者X锁时，由于隐式锁的存在，会先帮助当前事务生成一个锁结构，然后自己再生成一个锁结构后进入等待状态。

InnoDB锁的内存结构