MySQL 知识点 1　　digest time：2021-10-01

数据库三大范式是什么?

- ***每个列都不可以再拆分***。
- ***非主键列完全依赖于主键***，而不能是依赖于主键的一部分。
- **非主键列只依赖于主键**，不依赖于其他非主键。

mysql 有关权限的表都有哪几个?

MySQL 服务器**通过权限表来控制用户对数据库的访问**，权限表存放在 mysql 数据库里，由 mysql_install_db 脚本初始化。这些权限表分别 **user，db，table_priv**，**columns_priv 和 host**。下面分别介绍一下这些表的结构和内容：user 权限表：记录允许连接到服务器的用户帐号信息，里面的权限是全局级的。db 权限表：记录各个帐号在各个数据库上的操作权限。table_priv 权限表：记录数据表级的操作权限。columns_priv 权限表：记录数据列级的操作权限。host 权限表：配合 db 权限表对给定主机上数据库级操作权限作更细致的控制。**这个权限表不受 GRANT 和 REVOKE 语句的影响。**

MySQL 的 binlog 有有几种录入格式? 分别有什么区别?

有三种格式，**statement，row 和 mixed**。statement 模式下，每一条会修改数据的 sql 都会记录在 binlog 中。不需要记录每一行的变化，（**statement 模式**）**减少了 binlog 日志量，节约了 IO，提高性能**。由于 sql 的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。row 级别下，不记录 sql 语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是

可以全部记下来但是由于很多操作，会导致大量行的改动(比如 alter table)，因此（**row 模式**）**文件保存的信息太多，日志量太大。**mixed，一种折中的方案，**普通操作使用 statement 记录，当无法使用 statement 的时候使用 row。**

此外，新版的 MySQL 中对 row 级别也做了一些优化，当表结构发生变化的时候，会记录语句而不是逐行记录。

mysql 有哪些数据类型?

- 1、整数类型，包括 TINYINT、SMALLINT、MEDIUMINT、INT、BIGINT，分别表示 1 字节、2 字节、3 字节、4 字节、8 字节整数。任何整数类型都可以加上 UNSIGNED 属性，表示数据是无符号的，即非负整数。长度：整数类型可以被指定长度，例如：INT(11) 表示长度为 11 的 INT 类型。**长度在大多数场景是没有意义的，它不会限制值的合法范围，只会影响显示字符的个数**，而且需要和 UNSIGNED ZEROFILL 属性配合使用才有意义。例子：假定类型设定为 INT(5)，属性为 UNSIGNED ZEROFILL，如果用户插入的数据为 12 的话，那么数据库实际存储数据为 00012。
- 2、实数类型，包括 FLOAT、DOUBLE、DECIMAL。**DECIMAL 可以用于存储比 BIGINT 还大的整型，能存储精确的小数。**而 FLOAT 和 DOUBLE 是有取值范围的，并支持使用标准的浮点进行近似计算。计算时 FLOAT 和 DOUBLE 相比 DECIMAL 效率更高一些，DECIMAL 你可以理解成是用字符串进行处理。
- 3、字符串类型，包括 VARCHAR、CHAR、TEXT、BLOBVARCHAR 用于存储可变长字符串，它比定长类型更节省空间。VARCHAR 使用额外 1 或 2 个字节存储字符串长度。**列长度小于 255 字节时，使用 1 字节表示，否则使用 2 字节表示。**VARCHAR 存储的内容超出设置的长度时，内容会被截断。CHAR 是定长的，根据定义的字符串长度分配足够的空

间。CHAR 会根据需要使用空格进行填充方便比较。CHAR 适合存储很短的字符串，或者所有值都接近同一个长度。CHAR 存储的内容超出设置的长度时，内容同样会被截断。

- 4、枚举类型（ENUM），把不重复的数据存储为一个预定义的集合。有时可以使用 ENUM 代替常用的字符串类型。ENUM 存储非常紧凑，会把列表值压缩到一个或两个字节。ENUM 在内部存储时，其实存的是整数。尽量避免使用数字作为 ENUM 枚举的常量，因为容易混乱。排序是按照内部存储的整数
- 5、日期和时间类型，尽量使用 **timestamp，空间效率高于 datetime**，用整数保存时间戳通常不方便处理。**如果需要存储微妙，可以使用 bigint 存储。**

MyISAM 索引与 InnoDB 索引的区别?

- **InnoDB 索引是聚簇索引**，MyISAM 索引是非聚簇索引。
- **InnoDB 的主键索引的叶子节点存储着行数据，因此主键索引非常高效。**
- **MyISAM 索引的叶子节点存储的是行数据地址，需要再寻址一次才能得到数据。**
- InnoDB 非主键索引的叶子节点存储的是主键和其他带索引的列数据，因此查询时做到覆盖索引会非常高效。

InnoDB 引擎的 4 大特性（TODO）

- 插入缓冲（insert buffer)
- 二次写(double write)
- 自适应哈希索引(ahi)
- 预读(read ahead)

什么是索引?

- 索引是一种特殊的文件(InnoDB 数据表上的**索引是表空间的一个组成部分**)，它们包含着对数据表里所有记录的引用指针。
- 索引是一种数据结构。数据库索引，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。索引的实现通常使用 B 树及其变种 B+树。
- 更通俗的说，**索引就相当于目录。**为了方便查找书中的内容，通过对内容建立索引形成目录。索引是一个文件，它是要占据物理空间的。

索引有哪些优缺点?

- 索引的优点：可以大大加快数据的检索速度，这也是创建索引的最主要的原因。通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。
- 索引的缺点：**时间方面**：创建索引和维护索引要耗费时间，具体地，当对表中的数据进行**增加、删除和修改的时候，索引也要动态的维护**，会降低增/改/删的执行效率；**空间方面**：**索引需要占物理空间。**

索引有哪几种类型?

- **主键索引**: 数据列不允许重复，**不允许为 NULL**，一个表只能有一个主键。
- **唯一索引**: 数据列不允许重复，**允许为 NULL 值**，一个表允许多个列创建唯一索引。可以通过 ALTER TABLE table_name ADD UNIQUE index_name (column); 创建唯一索引可以通过 ALTER TABLE table_name **ADD UNIQUE index_name** (column1,column2); 创建唯一组合索引
- **普通索引**: 基本的索引类型，没有唯一性的限制，允许为 NULL 值。可以通过 ALTER TABLE table_name ADD INDEX index_name (column);创建普通索引可以通过 ALTER

TABLE table_name ADD INDEX index_name(column1, column2, column3);创建组合索引。

- **全文索引（TODO）**： 是目前搜索引擎使用的一种关键技术。可以通过 ALTER TABLE table_name ADD FULLTEXT (column);创建全文索引

索引的数据结构（b 树，hash）

- 索引的数据结构和具体存储引擎的实现有关，在 MySQL 中使用较多的索引有 **Hash 索引，B+树索引**等，而我们经常使用的 InnoDB 存储引擎的默认索引实现为：B+树索引。对于哈希索引来说，底层的数据结构就是哈希表，因此**在绝大多数需求为单条记录查询的时候，可以选择哈希索引，查询性能最快**；其余大部分场景，建议选择 BTree 索引。

- **1. B 树索引**

- mysql 通过存储引擎取数据，基本上 90%的人用的就是 InnoDB 了，按照实现方式分，InnoDB 的索引类型目前只有两种：BTREE（B 树）索引和 HASH 索引。B 树索引是 Mysql 数据库中使用最频繁的索引类型，基本所有存储引擎都支持 BTree 索引。通常我们说的索引不出意外指的就是（B 树）索引（实际是用 B+树实现的，因为在查看表索引时，mysql 一律打印 BTREE，所以简称为 B 树索引）

- **2. B+tree 性质**

- n 棵子 tree 的节点包含 n 个关键字，不用来保存数据而是**保存数据的索引。**

  所有的**叶子结点中包含了全部关键字的信息**，即指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接。

所有的非终端结点可以看成是索引部分，结点中仅含其子数中的最大（或最小）关键字。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

B+树有 2 个头指针，一个是树的根节点，一个是最小关键码的叶节点。

- **3. 哈希索引**

- 简要说下，类似于数据结构中简单实现的 HASH 表（散列表）一样，当我们在 mysql 中用哈希索引时，主要就是通过 Hash 算法（常见的 Hash 算法有直接定址法、平方取中法、折叠法、除数取余法、随机数法），将数据库字段数据转换成定长的 Hash 值，与这条数据的行指针一并存入 Hash 表的对应位置；**如果发生 Hash 碰撞（两个不同关键字的 Hash 值相同），则在对应 Hash 键下以链表形式存储。**当然这只是简略模拟图。

索引的基本原理

- 索引用来快速地寻找那些具有特定值的记录。如果没有索引，一般来说执行查询时遍历整张表。

  索引的原理很简单，就是把无序的数据变成有序的查询把创建了索引的列的内容进行排序对排序结果生成倒排表在倒排表内容上拼上数据地址链在查询的时候，先拿到倒排表内容，再取出数据地址链，从而拿到具体数据

索引算法有哪些?

- 索引算法有 BTree 算法和 Hash 算法
- 1. BTree 算法

- BTree 是最常用的 mysql 数据库索引算法，也是 mysql 默认的算法。因为它**不仅可以被用在=,>,>=,<,<=和 between 这些比较操作符上，而且还可以用于 like 操作符**，只要它的查询条件是一个不以通配符开头的常量。
- 2. Hash 算法
- Hash **Hash 索引只能用于对等比较**，例如=。由于是一次定位数据，不像 BTree 索引需要从根节点到枝节点，最后才能访问到页节点这样多次 IO 访问，所以检索效率远高于 BTree 索引。

• 索引设计的原则?

- 适合索引的列是出现在 where 子句中的列，或者连接子句中指定的列。
- 基数较小的类，索引效果较差，没有必要在此列建立索引
- 使用短索引，如果对长字符串列进行索引，应该指定一个前缀长度，这样能够节省大量索引空间
- 不要过度索引。索引需要额外的磁盘空间，并降低写操作的性能。在修改表内容的时候，索引会进行更新甚至重构，索引列越多，这个时间就会越长。所以只保持需要的索引有利于查询即可。

创建索引的原则

索引虽好，但也不是无限制的使用，最好符合一下几个原则最左前缀匹配原则，组合索引非常重要的原则，**mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配**，比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d 是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d 的顺序可以任意调整。较频繁作为查询条件的字段才去创建索引更新频繁字段不适合创建索引若是不能有效区分数

据的列不适合做索引列(如性别，男女未知，最多也就三种，区分度实在太低)尽量的扩展索引，不要新建索引。比如表中已经有 a 的索引，现在要加(a,b)的索引，那么只需要修改原来的索引即可。定义有外键的数据列一定要建立索引。对于那些查询中很少涉及的列，重复值比较多的列不要建立索引。对于定义为 text、image 和 bit 的数据类型的列不要建立索引。

创建索引时需要注意什么?

- **非空字段**：应该指定列为 NOT NULL，除非你想存储 NULL。在 mysql 中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用 0、一个特殊的值或者一个空串代替空值；
- **取值离散大的字段（区分度大的字段）**：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过 count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
- **索引字段越小越好**：**数据库的数据存储以页为单位**一页存储的数据越多一次 IO 操作获取的数据越大效率越高。

使用索引查询一定能提高查询的性能吗?

- 通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

  索引需要空间来存储，也需要定期维护， 每当有记录在表中增减或索引列被修改时，索引本身也会被修改。 这意味着每条记录的 INSERT，DELETE，UPDATE 将为此多付出 4，5 次的磁盘 I/O。 因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反

应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况:

- 基于一个范围的检索，一般查询返回结果集小于表中记录数的 30%
- 基于非唯一性索引的检索

百万级别或以上的数据如何删除？（先删除索引再删除数据）

- 关于索引：由于索引需要额外的维护成本，因为索引文件是单独存在的文件,所以当我们对数据的增加,修改,删除,都会产生额外的对索引文件的操作,这些操作需要消耗额外的 IO,会降低增/改/删的执行效率。所以，在我们删除数据库百万级别数据的时候，查询 MySQL 官方手册得知删除数据的速度和创建的索引数量是成正比的。所以**我们想要删除百万数据的时候可以先删除索引**（此时大概耗时三分多钟）然后删除其中无用数据（此过程需要不到两分钟）删除完成后重新创建索引(此时数据较少了)创建索引也非常快，约十分钟左右。与之前的直接删除绝对是要快速很多，更别说万一删除中断,一切删除会回滚。那更是坑了。

什么是最左前缀原则？什么是最左匹配原则？

顾名思义，就是最左优先，在创建多列索引时，要根据业务需求，where 子句中使用最频繁的一列放在最左边。

最左前缀匹配原则，非常重要的原则，mysql 会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如 a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d 是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d 的顺序可以任意调整。

=和 in 可以乱序，比如 a = 1 and b = 2 and c = 3 建立(a,b,c)索引可以任意顺序，mysql 的查询优化器会帮你优化成索引可以识别的形式

B 树和 B+树的区别

- 在 B 树中，你可以将键和值存放在内部节点和叶子节点；但**在 B+树中，内部节点都是键，没有值，叶子节点同时存放键和值。**

  B+树的叶子节点有一条链相连，而 B 树的叶子节点各自独立。

使用 B 树的好处（热点数据的查询效率）

- B 树可以在内部节点同时存储键和值，因此，**把频繁访问的数据放在靠近根节点的地方将会大大提高热点数据的查询效率。**这种特性使得 B 树在特定数据重复多次查询的场景中更加高效。

使用 B+树的好处（方便全数据遍历+内存页获取更多的键）

- 由于 B+树的内部节点只存放键，不存放值，因此，一次读取，可以在内存页中获取更多的键，有利于更快地缩小查找范围。 B+树的叶节点由一条链相连，因此，当需要进行一次全数据遍历的时候，B+树只需要使用 O(logN)时间找到最小的一个节点，然后通过链进行 O(N)的顺序遍历即可。而 B 树则需要对树的每一层进行遍历，这会需要更多的内存置换次数，因此也就需要花费更多的时间

什么是聚簇索引（数据和索引在一块）？何时使用聚簇索引与非聚簇索引?

- 聚簇索引：将**数据存储与索引放到了一块，找到索引也就找到了数据**
- 非聚簇索引：将数据存储于索引分开结构，索引结构的叶子节点指向了数据的对应行，

非聚簇索引一定会回表查询吗（覆盖查询不会）？

- 不一定，这涉及到查询语句所要求的字段是否全部命中了索引，如果全部命中了索引，那么就不必再进行回表查询。

联合索引是什么？为什么需要注意联合索引中的顺序?

- MySQL 可以使用多个字段同时建立一个索引，叫做联合索引。在联合索引中，如果想要命中索引，需要按照建立索引时的字段顺序挨个使用，否则无法命中索引。建立联合索引的时候应该注意索引列的顺序，一般情况下，**将查询需求频繁或者字段选择性高的列放在前面。**

什么是数据库事务（不可分割的数据库操作序列）？

- **事务是一个不可分割的数据库操作序列**，也是数据库并发控制的基本单位，其执行的结果必须使数据库从一种一致性状态变到另一种一致性状态。事务是逻辑上的一组操作，要么都执行，要么都不执行。

事物的四大特性(ACID)介绍一下?

- 原子性： 事务是最小的执行单位，不允许分割。事务的原子性确保动作要么全部完成，要么完全不起作用；
- 一致性： 执行事务前后，数据保持一致，多个事务对同一个数据读取的结果是相同的；

- 隔离性： 并发访问数据库时，一个用户的事务不被其他事务所干扰，各并发事务之间数据库是独立的；
- 持久性： 一个事务被提交之后。它对数据库中数据的改变是持久的，即使数据库发生故障也不应该对其有任何影响。

什么是脏读？幻读？不可重复读？

脏读(Drity Read)：读到未提交的数据。

不可重复读(Non-repeatable read):一次事务中相同的两次查询结果不一致。

幻读(Phantom Read):在一个事务的两次查询中数据笔数不一致。

什么是事务的隔离级别？MySQL 的默认隔离级别是什么？

READ-UNCOMMITTED**(读取未提交)**： 最低的隔离级别，允许读取尚未提交的数据变更，可能会导致脏读、幻读或不可重复读。

READ-COMMITTED**(读取已提交**)： 允许读取并发事务已经提交的数据，可以阻止脏读，但是幻读或不可重复读仍有可能发生。

REPEATABLE-READ(**可重复读)**： 对同一字段的多次读取结果都是一致的，除非数据是被本身事务自己所修改，可以阻止脏读和不可重复读，但幻读仍有可能发生。

SERIALIZABLE(**可串行化**)： 最高的隔离级别，完全服从 ACID 的隔离级别。所有的事务依次逐个执行，这样事务之间就完全不可能产生干扰，也就是说，该级别可以防止脏读、不可重复读以及幻读。

Mysql 默认采用的 REPEATABLE_READ 隔离级别 ；

Oracle 默认采用的 READ_COMMITTED 隔离级别

隔离级别与锁的关系（TODO）

- 在 Read Uncommitted 级别下，读取数据不需要加共享锁，这样就不会跟被修改的数据上的排他锁冲突
- 在 Read Committed 级别下，读操作需要加共享锁，但是在语句执行完以后释放共享锁；
- 在 Repeatable Read 级别下，读操作需要加共享锁，但是在事务提交之前并不释放共享锁，也就是必须等待事务执行完毕以后才释放共享锁。
- SERIALIZABLE 是限制性最强的隔离级别，因为该级别锁定整个范围的键，并一直持有锁，直到事务完成。

按照锁的粒度分数据库锁有哪些？（页级锁、行级、表级）

- 行级锁：**加锁粒度最小，开销最大。行级锁分为共享锁和排他锁。**特点：开销大，加锁慢；会出现死锁；锁定粒度最小，发生**锁冲突的概率最低**，并发度也最高。
- 表级锁：**开销小，加锁快；不会出现死锁**；锁定粒度大，发出**锁冲突的概率最高**，并发度最低。
- 页级锁：**锁定粒度介于行级锁和表级锁中间**，**一次锁定相邻的一组记录。**

从锁的类别上分 MySQL 都有哪些锁呢?

从锁的类别上来讲，有共享锁和排他锁。共享锁: 又叫做读锁。 当用户要进行数据的读取时，对数据加上共享锁（需要加上 lock in share mode）。共享锁可以同时加上多个。排

他锁：又叫做写锁。 当用户要进行数据的写入时，对数据加上排他锁。排他锁只可以加一个，他和其他的排他锁，共享锁都相斥。

InnoDB 存储引擎的锁的算法有哪三种?

- **Record lock**：单个行记录上的锁
- **Gap lock**：间隙锁，锁定一个范围，不包括记录本身
- **Next-key lock**：临键锁，record+gap 锁定一个范围，包含记录本身

什么是死锁？怎么解决？ (约定顺序、升级粒度、一次锁定、乐观锁或分布式锁)

定义：两个或多个事务在同一资源上相互占用，并请求锁定对方的资源，从而导致恶性循环的现象。

解决：**约定以相同的顺序访问表**；

一次锁定所需要的所有资源；

升级锁定颗粒度，通过表级锁定来减少死锁产生的概率；

用分布式事务锁或者使用乐观锁

数据库的乐观锁和悲观锁是什么？怎么实现的?

悲观锁：**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。在查询完数据的时候就把事务锁起来，直到提交事务。实现方式：使用数据库中的锁机制

乐观锁：**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。**在修改数据

的时候把事务锁起来，**通过 version 的方式来进行锁定。**实现方式：乐一般会使用版本号机制或 CAS 算法实现。

大表数据查询，怎么优化?

- 优化 shema、sql 语句+索引;
- 第二加缓存，memcached, redis;
- 主从复制，读写分离;
- 垂直拆分，根据你模块的耦合度，将一个大的系统分为多个小的系统，也就是分布式系统
- 水平切分，针对数据量大的表，这一步最麻烦，最能考验技术水平，要选择一个合理的 sharding key，为了有好的查询效率，表结构也要改动，做一定的冗余，应用也要改，sql 中尽量带 sharding key，将数据定位到限定的表上去查，而不是扫描全部的表

超大分页怎么处理?

数据库层面,修改为 select * from table where id in (**select id from table where age > 20 limit 1000000,10**).这样虽然也 load 了一百万的数据,但是由于索引覆盖,要查询的所有字段都在索引中,减少回表。

同时如果 ID 连续的好,我们还可以 select * from table where id > 1000000 limit 10,效率也是不错的,优化的可能性有许多种,但是核心思想都一样,就是减少 load 的数据从需求的角度减少这种请求…主要是不做类似的需求(直接跳转到几百万页之后的具体某一页.只允许逐页查看或者按照给定的路线走,这样可预测,可缓存)以及防止 ID 泄漏且连续被人恶意攻击

为什么要尽量设定一个主键?

主键是数据库确保数据行在整张表唯一性的保障，即使业务上本张表没有主键，也建议添加一个自增长的 ID 列作为主键。设定了主键之后，在后续的删改查的时候可能更加快速以及确保操作数据范围安全。

主键使用自增 ID 还是 UUID?

推荐使用自增 ID，不要使用 UUID。

InnoDB 存储引擎中，主键索引是作为聚簇索引存在的，如果主键索引是自增 ID，那么只需要不断向后排列即可，如果是 UUID，由于到来的 ID 与原来的大小不确定，会造成非常多的数据插入，数据移动，然后导致产生很多的内存碎片，进而造成插入性能的下降。

关于主键是聚簇索引，如果没有主键，InnoDB 会选择一个唯一键来作为聚簇索引，**如果没有唯一键，会生成一个隐式的主键。**

字段为什么要求定义为 not null?

null 值会占用更多的字节，且会在程序中造成很多与预期不符的情况。

如果要存储用户的密码散列（固定长度），应该使用什么字段进行存储?

密码散列，盐，用户身份证号等固定长度的字符串应该使用 char 而不是 varchar 来存储，这样可以节省空间且提高检索效率。

数据库结构优化?

需要考虑**数据冗余、查询和更新的速度、字段的数据类型**是否合理等多方面的内容。

将字段很多的表分解成多个表：对于字段较多的表，如果有些字段的使用频率很低，可以

将这些字段分离出来形成新表。因为当一个表的数据量很大时，会由于使用频率低的字段的存在而变慢。

增加中间表：对于需要经常联合查询的表，可以建立中间表以提高查询效率。通过建立中间表，将需要通过联合查询的数据插入到中间表中，然后将原来的联合查询改为对中间表的查询。

增加冗余字段：设计数据表时应尽量遵循范式理论的规约，尽可能的减少冗余字段，让数据库设计看起来精致、优雅。但是，合理的加入冗余字段可以提高查询速度。表的规范化程度越高，表和表之间的关系越多，需要连接查询的情况也就越多，性能也就越差。

MySQL 数据库 cpu 飙升到 500%的话他怎么处理?

当 cpu 飙升到 500%时，先用操作系统命令 top 命令观察是不是 mysqld 占用导致的，如果不是，找出占用高的进程，并进行相关处理。

如果是 mysqld 造成的， show processlist，看看里面跑的 session 情况，是不是有消耗资源的 sql 在运行。找出消耗高的 sql，看看执行计划是否准确， index 是否缺失，或者实在是数据量太大造成。

一般来说，肯定要 kill 掉这些线程(同时观察 cpu 使用率是否下降)，等进行相应的调整(比如说加索引、改 sql、改内存参数)之后，再重新跑这些 SQL。

也有可能是每个 sql 消耗资源并不多，但是突然之间，有大量的 session 连进来导致 cpu 飙升，这种情况就需要跟应用一起来分析为何连接数会激增，再做出相应的调整，比如说限制连接数等。

MySQL 主从复制解决的问题（故障切换、负载均衡、读写分离、日常备份）？

- 数据分布：随意开始或停止复制，并在不同地理位置分布数据备份
- 负载均衡：降低单个服务器的压力
- 高可用和故障切换：帮助应用程序避免单点失败
- 升级测试：可以用更高版本的 MySQL 作为从库

MySQL 主从复制工作原理（binlog 重放）？

- 在主库上把数据更改记录到二进制日志
- 从库将主库的日志复制到自己的**中继日志**
- 从库读取中继日志的事件，将其**重放**到从库数据中