如何回答B+树?

第一部分，回答B+树结构的通用情况

第一点，叶子节点和非叶子结点
B+树有叶子节点和非叶子节点。
叶子节点存放的是记录数据，非叶子节点仅存放索引，因此数据量相同的情况下，相比即存储索引又存储记录的B树，B+树的非叶子节点可以存放更多的索引，因此B+树可以比B树更加矮胖，查询底层节点的磁盘IO次数会更少。一次IO读入的节点数更多

第二点，插入和删除效率

B+树有大量的冗余节点，所以删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不移动非叶子节点，这样删除就非常快。甚至在删除根节点的时候，由于存在冗余 的节点，所以不会发生复杂的树变形。

B+树的插入也是一样，有冗余的节点，插入可能存在节点的分裂（如果节点饱和），但是最多涉及一条路径，而且B+树会自动平衡，不需要更多的复杂算法。
所以总的来说，B+树的删除和插入效率更高。

第三点，范围查询

B+树所有的叶子节点间还有一个链表进行连接，可以实现快速的范围查找

第二部分，回答MySQL中的B+树

MySQL的InnoDB存储引擎，就是采用B+树作为索引的数据结构。

第一点：InnoDB使用的B+树有一些特别的点
1，B+树的叶子节点之间是用双向链表进行连接的，这样的好处是即可以向右遍历，也能向左遍历。
2，**B+树节点内容是数据页**，数据页里存放用户的记录以及各种信息，每个数据页默认大小是16K。

第二点：聚集索引和辅助索引

InnoDB根据索引类型不同，分为聚集索引和辅助索引，它们的主要区别在于，**聚集索引的叶子节点存放的是实际数据，也就是完整的用户记录，而辅助索引的叶子节点存放的是辅助索引值和主键值，不包含完整的用户记录。**

表的数据都是存放在聚集索引的叶子节点里，所以InnoDB存储引擎一定会为表创建一个聚集索引，且由于数据在屋里上只会保存一份，所以**聚集索引只能有一个，而辅助索引可以创建多个。**

B+树与其他数据结构的比较，为什么不使用其他结构

红黑树、AVL树

1. 红黑树在插入和删除节点时，为了维护其特性，都需要复杂的重建树的过程。
2. 而且随着插入元素的增多，树的高度会变得很高，这就意味着磁盘IO操作次数变多，会影响整体数据查询的效率。

> **AVL 树和红黑树基本都是存储在内存中才会使用的数据结构**

Hash索引和B+树区别是什么?

1. B+树可以进行范围查询，Hash索引不能。
2. B+树支持联合索引的最左侧原则，Hash索引不支持。
3. B+树支持order by排序，Hash索引不支持。
4. Hash索引在等值查询上比B+树效率更高。
5. B+树使用like 进行模糊查询的时候，like后面（比如%开头）的话可以起到优化的作用，Hash索引根本无法进行模糊查询。

b+树为什么这么设计

第一，能在尽可能少的磁盘I/O操作中完成查询工作。
第二，要能高效地查询某个记录，也能高效地执行范围查找。

InnoDB里的B+ 树是如何进行查询的?

在InnoDB里的B+树中的每个节点都是一个数据页。

在查询记录时，从根节点开始，通过二分法快速定位到符合页内范围包含查询值的页，定位到叶子节点的数据页之后，又会在该页内通过二分法快速定位记录所在的那个槽，也就是记录分组，最后在分组内进行遍历查找记录。

找到记录之后，就根据查询语句使用的索引分为不同的情况。

如果某个查询语句使用了二级索引，但是查询的数据不是主键值，这时在二级索引数的记录中找到主键值后，需要去聚簇索引中获得数据行，这个过程就叫作「回表」，也就是说要查两个 B+ 树才能查到数据。

不过，当查询的数据是主键值时，因为只在二级索引就能查询到，不用再去聚簇索引查，这个过程就叫作「索引覆盖」，也就是只需要查一个 B+ 树就能找到数据

B树和B+树的区别，一般哪个层数更高

结构上的差别

1，B+树的叶子节点才会存放实际的数据（而B树存放的是索引+ 记录），而非叶子节点只会存放索引。
2，B+树的所有索引都会在叶子节点出现，叶子节点之间构成一个有序链表
3，非叶子节点中有多少个子节点，就会有多少个索引。

性能上的差别

1，单点 查询

> B树进行单个索引查询时，最快可以在O（1）的时间代价内就查到了，而从平均时间代价来看，会比B+树稍快一些。因为B+树得搜索到最底层的叶子节点才能拿到记录。
> 但是B树的查询波动会比较大，因为节点即存索引有存记录，所以 有时候访问到非叶子节点就可以找到索引，而有时需要访问到叶子节点才能找到索引。
>
> B+树的非叶子节点不存放实际的记录数据，仅存放索引，因此数据量相同的情况下，相比即存储索引又存储记录的B树，B+树的非叶子节点可以存放更多的索引，**因此B+树可以比B树更加矮胖**，查询底层节点的磁盘IO次数会更少。

2，插入和删除效率

> B+树有大量的冗余节点，这样使得删除一个节点的时候，可以直接从叶子节点中删除，甚至可以不移动非叶子节点，这样删除就非常快。甚至在删除根节点的时候，由于存在冗余 的节点，所以不会发生复杂的树变形。

> 而B树没有冗余节点，删除节点的时候很复杂，删除根节点中的数据时，可能涉及复杂的树的变形。
>
> B+树的插入也是一样，有冗余的节点，插入可能存在节点的分裂（如果节点饱和），但是增多涉及一条路径，而且B+树会自动平衡，不需要更多的复杂算法。
> 所以总的来说，B+树的删除和插入效率更高。

3，范围查询

>B+树所有的叶子节点间还有一个链表进行连接，这种涉及对范围查找非常有帮助
>而B树没有将所有叶子节点用链表串联起来的结构，因此只能通过树的遍历来完成范围查询，这会涉及多个节点的磁盘IO操作，范围查询效率不如B+树。
>
>因此，对于存在大量范围检索的场景，适合使用B+树，比如数据库。而对于大量的单个索引查询的场景，可以考虑B树，比如MangoDB

都有哪些维度可以进行数据库调优?

- 索引失效、没有充分利用到索引——`索引建立`
- 关联查询太多JOIN （设计缺陷或不得已的需求）——`SQL优化`，通过`SQL等价变换`提升查询效率，直白一点就是说，换一种查询写法执行效率可能更高。
- 数据过多——`分库分表`

索引调优

首先我们要明确索引的作用就是加快数据的检索，避免全表扫描。
那我们就得清楚使用索引有什么注意的事项，才能进行优化。
主要分为三个维度

第一、索引建立的原则以及索引不适合哪些场景。

索引列要考虑数据区分度。如果重复数据太多就不适合索引列，比如性别
数据量少不适合加索引，增加索引维护的工作量
频繁更新的字段也不适合去建立索引，因为更新数据的同时，索引也需要进行更新维护，造成较多性能开销。

第二、导致索引失效的场景

1，对索引字段进行计算，函数或者自动类型转换会导致索引失效
2，字符串类型的索引不加引号，比如字符串的内容是123，但是在查询时没有加引号，底层就会把这个字符串类型的字段转换成整形的，也就是对字段进行类型转换，就会导致索引失效。
3，使用不等于进行查询
4，is null可以使用索引，但是is not null只有在查询字段仅是索引字段时才能使用索引
5，左模糊匹配和左右模糊匹配不行
6，or前后存在非索引的列，索引失效
7，联合索引不符合最左匹配原则，
8，联合索引中范围条件后面的列用不上索引。比如联合索引(a,b,c) where a = xxx and b > xxx and c= xxx,这时候c是用不上索引的。

第三、索引的一些潜在原则

1，要利用好覆盖索引，减少回表的次数
2，最左匹配原则，防止联合索引失效
3，了解索引下推，能够有效减少回表次数。

> 联合索引的字段会先在存储引擎中进行筛选，再回表检查是否符合查询条件。

SQL优化（慢SQL优化）

Join优化

利用好小表驱动大表的原则，并且被驱动表要建好索引。
具体步骤是要先遍历驱动表，所以就要求驱动表的数据量较小，然后，驱动表遍历出的每一行记录都要去被驱动表中根据on条件字段进行搜索，由于被驱动表上建立了条件字段的索引，所以每次搜索就只需要在索引树上扫描记录就行，性能较高。

order by优化（调大sort_buffer，避免外部排序， 建联合索引，取消排序过程）

1. order by 的基本原理其实就是 MySQL 会给每个线程分配一块内存也就是 sort_buffer 用于排序，sort_buffer 中存储的是 select 涉及到的所有的字段，可以称为全字段排序吧。排序这个动作，可能在内存中完成，也可能需要使用外部排序，这取决于排序所需的内存和 sort_buffer 的大小，由参数 `sort_buffer_size` 决定。如果要排序的数据量小于 sort_buffer_size，排序就在内存中完成。但如果排序数据量太大，内存放不下，就需要利用磁盘临时文件来辅助排序。
2. 这里其实可以优化下，只存放排序相关的字段，而不是 select 涉及的所有字段，这样 sort_buffer 中存放的东西就多一点，就尽可能避免使用磁盘进行外部排序，或者说使得划分的磁盘文件相对变少，减少磁盘访问。这种排序称为 rowid 排序。 如果表中单行的长度超过 `max_length_for_sort_data` 定义的值，那 MySQL 就认为单行太大（那么数据量肯定就越大，sort_buffer 可能不够用），由全字段排序改为 rowid 排序。

以上是我们说的关于 order by 的两个参数优化，还可以根据索引进行一些优化

1. 以 `select a, b, c from table where a = xxxx order by b` 为例，我们为查询条件 a 和排序条件 b 建立联合索引，联合索引就是 a 是从小到大绝对有序的，如果 a 相同，再按 b 从小到大排序，这样就不需要排序了，直接避免了排序这个操作。
2. 还可以进一步优化，由于联合索引 (a, b) 中没有 c 的值，所以从联合索引树上获取符合条件的对应主键 id 后，还需要回表查询取出 a b c 的值，这个回表查询的过程可以通过建立 (a,b,c) 覆盖索引来避免。

数据量大（分库分表）

一、水平分表（以字段为依据，按照一定策略（hash、range等），将一个表中的数据拆分到多个表中。）
应用场景就是系统并发量还没有太高，只是单表的数据量太多，影响了SQL的效率，加重了CPU的负担，以至于成为瓶颈。
水平分表后，表的数据量少了，单次执行SQL的效率高，自然减轻了CPU的负担。

二、水平分库（以字段为依据，按照一定策略（hash、range等），将一个库中的数据拆分到多个库中。）
应用场景是系统的绝对并发量上来了，分表难以从根本上解决问题，并且还没有明显的业务归属来垂直分库。
水平分库后，库多了，IO和CPU的压力就能缓解

三、垂直分表（以字段为依据，按照字段的活跃性，将表中字段拆到不同的表（主表和扩展表）中。）
列表页和详情页
应用场景是系统的绝对并发量并没有上来，表的记录并不多，只是字段比较多，并且热点数据和非热点数据在一起，单行数据所需存储的空间较大。以至于数据库缓存的数据行减少，查询时会去读磁盘数据产生大量的随机IO，产生IO瓶颈。

可以用列表页和详情页来帮助理解。垂直分表的拆分原则是将热点数据（可能会冗余经常一起查询的数据）放在一起作为主表，非热点数据放在一起作为扩展表。这样更多的热点数据就能被缓存下来，进而减少了随机读IO。

拆了之后，要想获得全部数据就需要关联两个表来取数据。但记住，千万别用join，因为join不仅会增加CPU负担并且会将两个表耦合在一起（必须在一个数据库实例上）。
关联数据，应该在业务Service层做文章，分别获取主表和扩展表数据然后用关联字段关联得到全部数据。

四、垂直分库（以表为依据，按照业务归属不同，将不同的表拆分到不同的库中。）

应用场景是系统绝对并发量上来了，并且可以抽象出单独的业务模块。
分析：到这一步，基本上就可以服务化了。例如，随着业务的发展一些公用的配置表、字典表等越来越多，这时可以将这些表拆到单独的库中，甚至可以服务化。再有，随着业务的发展孵化出了一套业务模式，这时可以将相关的表拆到单独的库中，甚至可以服务化。

分库分表可能遇到的问题

事务问题：需要用分布式事务啦

跨节点Join的问题：解决这一问题可以分两次查询实现，（串行）

跨节点的count,order by,group by以及聚合函数问题：分别在各个节点上得到结果后在应用程序端进行合并。（并行）

数据迁移，容量规划，扩容等问题

ID问题：数据库被切分后，不能再依赖数据库自身的主键生成机制啦，最简单可以考虑UUID，建一个序列号表，雪花ID

SQL优化的一般步骤?

一、通过show status命令去了解各种SQL的执行频率。
比如更新操作和查询操作的执行频率，就能了解当前数据库的应用是更新为主还是查询为主。
另外也可以了解事务提交和回顾的情况，如果回滚操作过于频繁，可能意味着应用程序编写存在问题。

二、定位执行效率较低的 sql 语句

- 通过慢查询日志定位那些执行效率较低的 sql 语句
  然后利用explain查看sql是否使用了索引，或者连接表的顺序

  > explain详解

> **explain会出现几种索引扫描类型？出现filesort的场景**
>
> **结合Explain分析Sql语句的索引使用情况**

- show processlist实时定位问题（分析加锁情况）
  慢查询日志在查询结束以后才记录，所以在应用反映执行效率出现问题的时候慢查询日志并不能定位问
  题，可以使用 show processlist 命令查看当前 mysql 在进行的线程，包括线程的状态、是否锁表等，可
  以实时的查看 sql 的执行情况，同时对一些锁表操作进行优化。避免死锁问题

b+树一般是多少层

一、先计算非叶子节点能够存放多少数据量

非叶子节点里面存的是主键值 + 指针，我们假设主键的类型是 BigInt，长度为 8 字节，而指针大小在 InnoDB 中设置为 6 字节，这样一共 14 字节。

为了方便行文，这里我们把一**个主键值 + 一个指针**称为一个单元，这样的话，一页或者说一个非叶子节点能够存放 16384 / 14=1170 个这样的单元。

二、再计算叶子节点能够存放的数据量

简单按照一行记录的数据大小为 1k 来算的话（实际上现在很多互联网业务数据记录大小通常就是 1K 左右），一页或者说一个叶子节点可以存放 16 行这样的数据。

所以两层高的B+树能存放的数据量就是

1170（一个非叶子节点中的指针数） * 16（一个叶子节点中的行数）= 18720 行数据。

三层高（2千万条数据）

1170 * 1170 * 16 = 21902400

mysql了解么?

先回答整个MySQL的架构（mysql发一个sql请求，到磁盘查出来需要经过哪里步骤？）

> keywords: 连接器、Server层、存储引擎层

MySQL的架构从上到下，也就是一条SQL语句到达MySQL服务端，要先与连接器建立连接，并检查权限；

权限通过就交给下层的Server层，Server层包括分析器，优化器，执行器。

先经过分析器进行语法解析，生成有效的解析树发送给优化器，优化器生成执行计划，然后交付给执行器执行。

执行器就会调用接口与存储引擎层进行交互，从存储引擎层获取所需要的数据记录。

然后回答InnoDB

MySQL的默认存储引擎是InnoDB，这得益于其高效的索引实现和事务的实现。

底层结构使用的是B+树，能够在较少的IO情况下，实现高效的索引记录并进行范围查询。

实现了的事务ACID特性。利用了undo log实现原子性，锁+MVCC实现隔离性，redo log来实现持久性，最终通过原子性，隔离性和持久性来保证事务的一致性。

MyISAM 和 InnoDB 的区别?

> keywords : 事务，count，锁，索引即数据， 主键索引树和辅助索引树的不同

一、InnoDB支持事务，MyISAM不支持事务

二、InnoDB 支持 MVCC(多版本并发控制)，MyISAM 不支持

三、select count(*) from table时，也就是统计表中的记录时，MyISAM更快，因为它有一个变量保存了整个表的总行数，可以直接读取，InnoDB就需要全表扫描。但如果统计条件中加入了where查询，那就都需要全表扫描。

四、InnoDB支持表、行级锁，而MyISAM支持表级锁。

五、InnoDB的数据文件本身就是索引文件，而MyISAM的索引和数据是分开的。

六、MyISAM使用的是非聚簇索引，非聚簇索引的两棵B+树（主键索引树和辅助索引树）看上去没什么不同，节点的结构完全一致只是存储的内容不同而已，主键索引B+树的节点存储了主键，辅助键索引B+树存储了辅助键。表数据存储在独立的地方，这两颗B+树的叶子节点都使用一个地址指向真正的表数据，对于表数据来说，这两个键没有任何差别。由于索引树是独立的，通过辅助键检索无需访问主键的索引树。

InnoDB的主键索引是聚集索引，叶子节点存放的是完整的数据记录。而辅助索引树只是存放辅助索引键以及主键。

> **Memory**存储引擎也称**HEAP**存储引擎,所有的数据都保存在内存中,如果**MySQL**服务重启，数据将会丢失,但是表的结构会保存下来

事务相关(数据库的特性)

数据库事务特性，为什么要保证ACID

- 原子性： 事务作为一个整体被执行，包含在其中的对数据库的操作要么全部都执行，要么都不执行。

  > 整体执行

- 一致性： 指在事务开始之前和事务结束以后，数据不会被破坏，假如A账户给B账户转10块钱，不管成功与否，A和B的**总金额是不变的。**

  > 事务前后总金额不变

- 隔离性： 多个事务并发访问时，事务之间是相互隔离的，一个事务不应该被其他事务干扰，多个并发事务之间要相互隔离。

  > 事务互不干扰

- 持久性： 表示事务完成提交后，该事务对数据库所作的操作更改，将持久地保存在数据库之中。

  > 持久到磁盘

为什么需要实现ACID?

事务是由有限的数据库操作序列构成，这些操作要么全部完成，要么全部不执行，是不可分割的。所以为了保证事务的正确执行，我们就需要实现事务的ACID特性。

MySQL是怎么实现事务的ACID的

- 原子性：是使用 undo log来实现的，如果事务执行过程中出错或者用户执行了rollback，系统通过undo log日志,以及回滚指针形成的版本链，返回事务开始的状态。

  > 在对数据库进行修改时，innoDB引擎除了会产生redo log，还会产生undo log。**InnoDB实现回滚，靠的是undo log：当事务对数据库进行修改时，InnoDB会生成对应的undo log；如果事务执行失败导致事务需要回滚，就利用undo log中的信息将数据回滚到修改之前的样子。**

- 持久性：使用 redo log来实现，只要redo log日志持久化了，当系统崩溃，即可通过redo log把数据恢复。（引出两阶段提交）

  > 当做数据修改的时候，不仅在内存中操作，还会在redo log中记录这次操作。当事务提交的时候，会将redo log日志进行刷盘(redo log一部分在内存中，一部分在磁盘上)。当数据库宕机重启的时候，会将redo log中的内容恢复到数据库中，再根据undo log和binlog内容决定回滚数据还是提交数据。

- 隔离性：通过锁以及MVCC,使事务相互隔离开。（引出事务的隔离级别）
- 一致性：通过回滚(原子性)、恢复（持久性），以及并发情况下的隔离性，从而实现一致性。

事务的持久性只和redo log有关吗？除了redo log还有保障持久性的技术吗？——两阶段提交

两段式提交，就是我们先把这次更新写入到redo log中，并设redo log为prepare状态，然后再写入bin log,写完bin log之后再提交事务，并设redo log为commit状态。也就是把redo log拆成了prepare和commit两段！

为什么需要两阶段提交?

为了保证redo log和bin log数据的安全一致性。只有在这两个日志文件逻辑上高度一致了。你才能放心地使用redo log帮你将数据库中的状态恢复成crash之前的状态，使用bin log实现数据备份、恢复、以及主从复制。

而两阶段提交的机制可以保证这两个日志文件的逻辑是高度一致的。没有错误、没有冲突。

两阶段提交为什么能够实现崩溃恢复?

根据两阶段提交，崩溃恢复时的判断规则是这样的：

1，如果 redo log 里面的事务是完整的，也就是已经有了 commit 标识，则直接提交

2，如果 redo log 里面的事务处于 prepare 状态，则判断对应的事务 binlog 是否存在并完整

- a. 如果 bin log 存在并完整（XID一致），则提交事务；

  > 当MySQL写完redolog并将它标记为prepare状态时，并且会在redolog中记录一个XID，它全局唯一的标识着这个事务。写入binlog，其结束的位置上也有一个XID。

  也就是说，如果数据库在写入bin log之后，redo log状态修改为commit前发生崩溃，此时redo log里面的事务仍然是prepare状态，而bin log存在并完整。这时候即使数据库崩溃了，事务仍然会被正常提交。

  在主从复制的场景下，因为bin log已经写入成功了，这样之后就会被从库同步过去。但是实际上主库并没有完成这个更新操作，所以为了主备一致，在主库上需要提交这个事务。 -

- b. 如果不完整（XID不一致），则回滚事务。

  也就是说，如果数据库在写入redo log（prepare）阶段之后，在写入bin log之前，系统崩溃了。此时的redo log里面的事务处于 prepare 状态，bin log 还没写，所以崩溃的时候，这个事务会回滚。

  为什么需要回滚事务？因为 bin log 还没有写入，之后从库进行同步的时候，无法执行这个操作，那如果我们主库上继续执行这个操作的话显然就会导致主备不一致，所以在主库上需要回滚这个事务。并且，由于 binlog 还没写，所以也就不会传到备库，从而避免主备不一致的情况。

所以，其实可以看出来，处于 prepare 阶段的 redo log 加上完整的 bin log，就能保证数据库的崩溃恢复了。

bin log 和redo log的区别

1，**适用对象不同**

- binlog是MySQL的Server层实现了，所有存储引擎都可以使用
- redo log 是InnoDB特有的

2，**写入内容不同**

- binlog是逻辑日志（数据记录），记录的是这个语句的原始逻辑，对系统的每个操作、每个修改都保存起来，比如“给id = 1的这一行的age字段加1”

  > undo log也是逻辑日志

- redo log是物理日志（数据页的修改），记录的是“在某个数据页上做了什么修改”

3，**写入方式不同**

- binlog是追加写入的。“追加写” 是指 bin log 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
- redo log 是循环写的，空间固定会被用完。redo log 只会记录未刷入磁盘的日志，已经刷入磁盘的数据都会从 redo log 这个有限大小的日志文件里删除。

4，**具备崩溃恢复的能力**

- binlog保存的是全量的日志，那就会导致一个问题，就是没有标志去标识binlog中的哪些数据已经刷入磁盘了。那么当数据库崩溃重启后，只通过binlog，数据库是无法判断哪条记录已经写入磁盘，就无法确定具体该恢复哪些记录。
- redo log只要是刷入磁盘的数据，都会从redo log中被抹掉，数据库重启后，直接把redo log中的数据恢复到内存就可以了。

事务的隔离级别

读未提交（Read Uncommitted） -> 脏读

读已提交（Read Committed） - > 不可重复读

可重复读（Repeatable Read） -> 幻读 - > 间隙锁

串行化（Serializable）

Mysql默认的事务隔离级别是可重复读(Repeatable Read)

乐观锁要在什么隔离级别下才有效?

读提交，RR会读到以前的数据。

如何实现可重复读？读提交?

重头戏就是读取已提交和可重复读是如何实现的?

通过MVCC（多版本并发控制），具体实现需要通过Undo log版本链和ReadVIew视图机制。

对于这两个隔离级别，**数据库会为每个事务创建一个视图 (ReadView)，访问的时候以视图的逻辑结果为准**:

- 在 “读取已提交” 隔离级别下，这个视图是在每个 SQL 语句开始执行的时候创建的。所以每次读操作都是读取最新的行数据版本，而这最新的数据行版本很可能是某个事务进行了修改操作后提交

的，所以可能会发生多次读取同一行数据，但是前后读取的数据不一致的情况。这就是不可重复读现象，所以提交读不能避免不可重复度现象。
- 在 “可重复读” 隔离级别下，这个视图是在事务启动时就创建的，整个事务存在期间都用这个视图（这就是为什么说在可重复读隔离级别下，一个事务执行过程中看到的数据，总是跟这个事务在启动时看到的数据是一致的）

那么问题了就来了，**已经执行了这么多的操作，事务该如何重新回到之前视图记录的状态**?

这就是 undo log 版本链做的事

什么是MVCC

B+ 索引树上对应的记录只会有一个最新版本，但是 InnoDB 可以根据 undo log 得到数据的历史版本。同一条记录在系统中可以存在多个版本，就是数据库的多版本并发控制（MVCC）

什么是undo log 版本链?

在 MySQL 中，每条记录在更新的时候都会同时记录一条回滚操作（也就是 undo log），当前记录上的最新值，通过回滚操作，都可以得到前一个状态的值。

简单理解，**undo log 就是每次操作的反向操作**，比如比如当前事务执行了一个插入 id = 100 的记录的操作，那么 undo log 中存储的就是删除 id = 100 的记录的操作。

InnoDB 存储引擎中每条行记录其实都拥有两个隐藏的字段：trx_id 和 roll_pointer，每次修改行记录都会更新 trx_id 和 roll_pointer 这两个隐藏字段，之前的多个数据快照对应的 undo log 会通过 roll_pointer 指针串联起来，从而形成一个版本链。

ReadView 机制

undo log版本链就是把数据记录的多个历史版本串联起来，那ReadView 机制就是用来判断当前事务能够看见哪些版本的。

> 一个 ReadView 主要包含如下几个部分：
>
> - `m_ids`：生成 ReadView 时有哪些事务在执行但是还没提交的（称为 “**活跃事务**”），这些活跃事务的 id 就存在这个字段里
> - `min_trx_id`：m_ids 里最小的值
> - `max_trx_id`：生成 ReadView 时 InnoDB 将分配给下一个事务的 ID 的值（事务 ID 是递增分配的，越后面申请的事务 ID 越大）
> - `creator_trx_id`：当前创建 ReadView 事务的 ID
>
> 当一个事务读取某条数据时，就会按照如下规则来决定当前事务能读取到什么数据：
>
> 1. 如果当前数据的 row_trx_id 小于 min_trx_id，那么表示这条数据是在当前事务开启之前，其他的事务就已经将该条数据修改了并提交了事务(事务的 id 值是递增的)，所以当前事务能读取到。
> 2. 如果当前数据的 row_trx_id **大于等于 max_trx_id**，那么表示在当前事务开启以后，过了一段时间，系统中有新的事务开启了，并且新的事务修改了这行数据的值并提交了事务，所以当前事务肯定是不能读取到的，因此这是后面的事务修改提交的数据。
> 3. 如果当前数据的 row_trx_id 处于 min_trx_id 和 max_trx_id 的范围之间，又需要分两种情况：
>
>   （a）row_trx_id 在 m_ids 数组中，那么当前事务不能读取到。为什么呢？row_trx_id 在 m_ids 数组中表示的是和当前事务在同一时刻开启的事务，修改了数据的值，并提交了事务，所以不能让当前事务读取到；

> (b) row_trx_id 不在 m_ids 数组中，那么当前事务能读取到。row_trx_id 不在 m_ids 数组中表示的是在当前事务开启之前，其他事务将数据修改后就已经提交了事务，所以当前事务能读取到。
>
> **注意：如果 row_trx_id 等于当前事务的 id，那表示这条数据就是当前事务修改的，那当前事务肯定能读取到啊。**

幻读

幻读就是一个事务在前后两次查询同一个范围的时候，前后两次查询的记录数量不一致。

举例

假设有A和B两个事务同时在进行。A先查询数据库中账户余额大于100万的记录，共有5条，B也按照相同的搜索条件查询出5条记录。
接下来， 事务A插入了一条余额超过100万的账号，并提交了事务，此时数据库中账户余额超过100万的记录就有6条。
然后事务B再次查询账户余额大于100万的记录就会有6条。这就产生了幻读。
因为同一个事务的相同查询，记录数应当不变。

快照读下通过MVCC解决幻读

在可重复读隔离级别下，**普通的查询是快照读，是不会看到别的事务插入的数据的。**

可重复读隔离级是由 MVCC（多版本并发控制）实现的，实现的方式是启动事务后，在执行第一个查询语句后，会创建一个视图，然后后续的查询语句都用这个视图，「快照读」读的就是这个视图的数据，视图你可以理解为版本数据，这样就使得每次查询的数据都是一样的。

MySQL 里除了普通查询是快照读，其他都是**当前读**，比如update、insert、delete，这些语句执行前都会查询最新版本的数据，然后再做进一步的操作。还有下面两种查询方式

> select...lock in share mode (共享读锁)
> select...for update
> update, delete, insert

这很好理解，假设你要 update 一个记录，另一个事务已经 delete 这条记录并且 提交事务了，这样不是会产生冲突吗，所以 update 的时候肯定要知道最新的数据。

另外，`select ... for update` 这种查询语句是当前读，每次执行的时候都是读取最新的数据。

因此，要讨论「可重复读」隔离级别的幻读现象，是要建立在「当前读」的情况下。

临键锁 Next-Lock Key解决当前读下的幻读

Innodb 引擎为了解决「可重复读」隔离级别使用「当前读」而造成的幻读问题，就引入了 next-key 锁，就是记录锁和间隙锁的组合。不仅对扫描到的行进行加锁，还对行之间的间隙进行加锁，这样就能杜绝新数据的插入和更新。

- 记录锁，锁的是记录本身；
- 间隙锁，锁的就是两个值之间的空隙，以防止其他事务在这个空隙间插入新的数据，从而避免幻读现象。

> 需要注意的是，next-key lock 锁的是索引，而不是数据本身，所以如果 update 语句的 where 条件没有用到索引列，那么就会全表扫描，在一行行扫描的过程中，不仅给行加上了行锁，还给行两边的空隙也加上了间隙锁，相当于锁住整个表，然后直到事务结束才会释放锁。
>
> 所以在线上千万不要执行没有带索引的 update 语句，不然会造成业务停滞

日志

物理日志和逻辑日志

逻辑日志：可以简单理解为记录的就是SQL语句
物理日志：因为MySQL数据最终是保存在数据页中的，物理日志记录的就是数据页的变更

bin log

是什么?

用于记录数据库执行的写入性操作信息，以二进制的形式保存在磁盘中。binlog是逻辑日志，并且是由Server层进行存储的，使用任何存储引擎的MySQL数据库都会记录binlog日志。

应用场景

binlog日志主要应用于主从复制，还有数据恢复。

binlog是什么时候刷盘的?

对于InnoDB存储引擎而言，一个更新操作只有在事务提交时才会记录binlog，此时记录还在内存中，那么binlog是什么时候刷到磁盘中的呢？**mysql通过sync_binlog参数控制binlog的刷盘时机**，取值范围是0-N：

0：不去强制要求，由系统自行判断何时写入磁盘；
1：每次commit的时候都要将binlog写入磁盘；
N：每N个事务，才会将binlog写入磁盘。

从上面可以看出，s**ync_binlog最安全的是设置是1，这也是MySQL 5.7.7之后版本的默认值。**但是设置一个大一些的值可以提升数据库性能，因此实际情况下也可以将值适当调大，牺牲一定的一致性来获取更好的性能。

binlog日志格式

`binlog` 日志有三种格式，分别为 `STATMENT`、 `ROW` 和 `MIXED`。

- **Statement** 模式：只记录执行的 SQL，不需要记录每一行数据的变化

  优点：**不需要记录每一行数据的变化，减少了 `binlog` 日志量，节约了 `IO`，从而提高了系统性能**；

  缺点：正是由于 Statement 模式只记录 SQL，所以如果一些 SQL 中包含了函数，那么可能会出现执行结果不一致的情况。比如说 uuid() 函数，每次执行的时候都会生成一个随机字符串，在 master 中记录了 uuid，当同步到 slave 之后，再次执行，就获取到另外一个结果了。
  所以使用 Statement 格式会出现一些数据一致性问题。

- Row 格式：不记录 SQL 语句上下文相关信息，仅仅只需要记录某一条记录被修改成什么样子了。

  优点：Row 格式的日志内容会非常清楚的记录下每一行数据修改的细节，这样就不会出现 Statement 中存在的那种数据无法被正常复制的情况。避免出现数据一致性问题。

  缺点：日志量太大了，特别是批量 update、整表 delete、alter 表等操作，由于要记录每一行数据的变化，此时会产生大量的日志，大量的日志也会带来 IO 性能问题。

- MIXED：基于 `STATMENT` 和 `ROW` 两种模式的混合复制(`mixed-based replication, MBR`)，

  在 Mixed 模式下，系统会自动判断该用 Statement 还是 Row：一般的语句修改使用 Statement 格式保存 binlog；

  对于一些 **Statement 无法准确完成主从复制的操作**，则采用 Row 格式保存 binlog。

> 在`MySQL 5.7.7`之前，默认的格式是`STATEMENT`，`MySQL 5.7.7`之后，默认值是`ROW`。日志格式通过`binlog-format`指定。

redo log

为什么需要redo log

> keywords: 持久化，减少刷盘次数。

我们都知道，事务的四大特性里面有一个是**持久性**，具体来说就是**只要事务提交成功，那么对数据库做的修改就被永久保存下来了，不可能因为任何原因再回到原来的状态。**

那么`mysql`是如何保证持久性的呢？最简单的做法是在每次事务提交的时候，将该事务涉及修改的数据页全部刷新到磁盘中。但是这么做会有严重的性能问题，主要体现在两个方面：

1. 因为`Innodb`是以`页`为单位进行磁盘交互的，而一个事务很可能只修改一个数据页里面的几个字节，这个时候将完整的数据页刷到磁盘的话，太浪费资源了！
2. 一个事务可能涉及修改多个数据页，并且这些数据页在物理上并不连续，使用随机IO写入性能太差！

因此`mysql`设计了`redo log`，**具体来说就是只记录事务对数据页做了哪些修改**，这样就能完美地解决性能问题了(相对而言文件更小并且是顺序IO)。

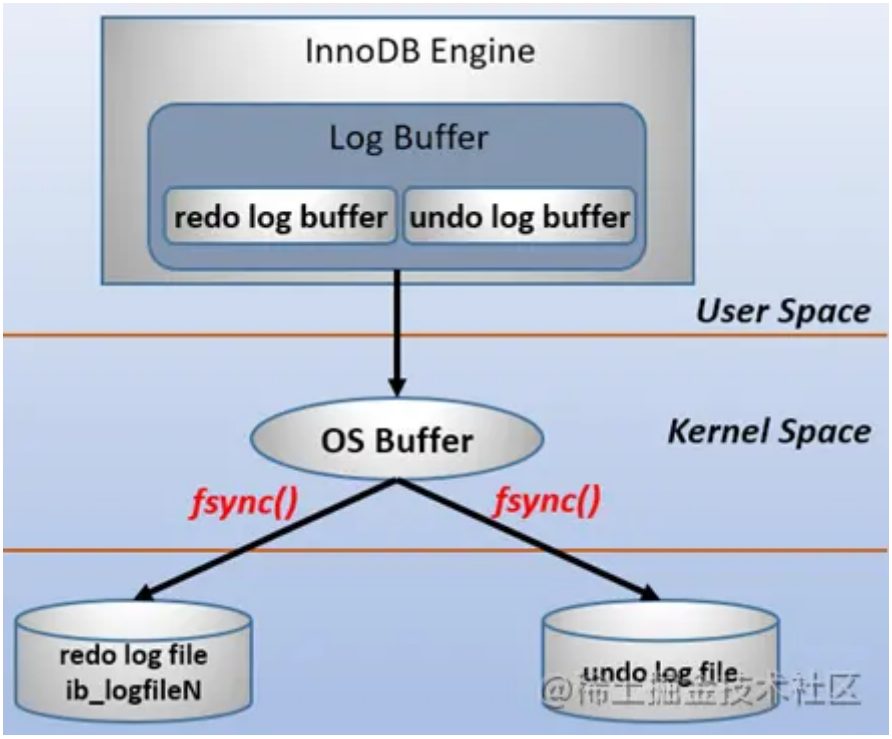
redo log基本概念（如何将redolog的数据同步到磁盘）

> keywords:WAL技术，先写日志，再写磁盘、。

`redo log`包括两部分：一个是内存中的日志缓冲(`redo log buffer`)，另一个是磁盘上的日志文件(`redo log file`)。

`mysql`每执行一条`DML`语句，先将记录写入`redo log buffer`，后续某个时间点再一次性将多个操作记录写到`redo log file`。这种**先写日志，再写磁盘**的技术就是`MySQL`里经常说到的`WAL(Write-Ahead Logging)`技术。

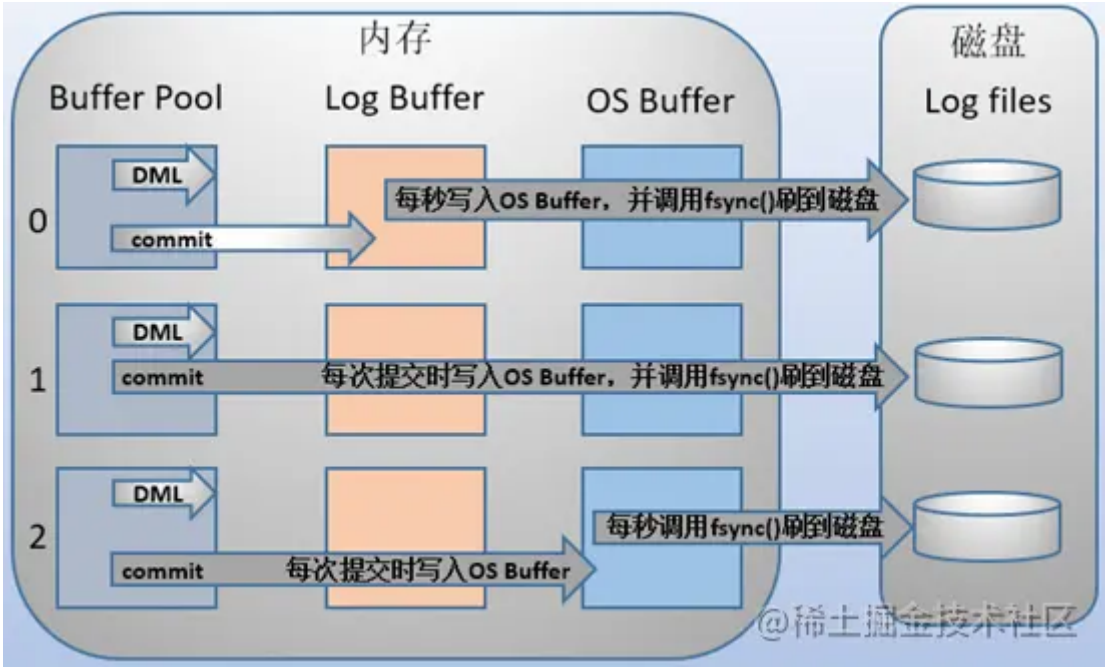
在计算机操作系统中，用户空间(`user space`)下的缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统内核空间(`kernel space`)缓冲区(`OS Buffer`)。因此，`redo log buffer`写入`redo log file`实际上是先写入`OS Buffer`，然后再通过系统调用`fsync()`将其刷到`redo log file`中，过程如下：



redo log buffer写入redo log file的时机

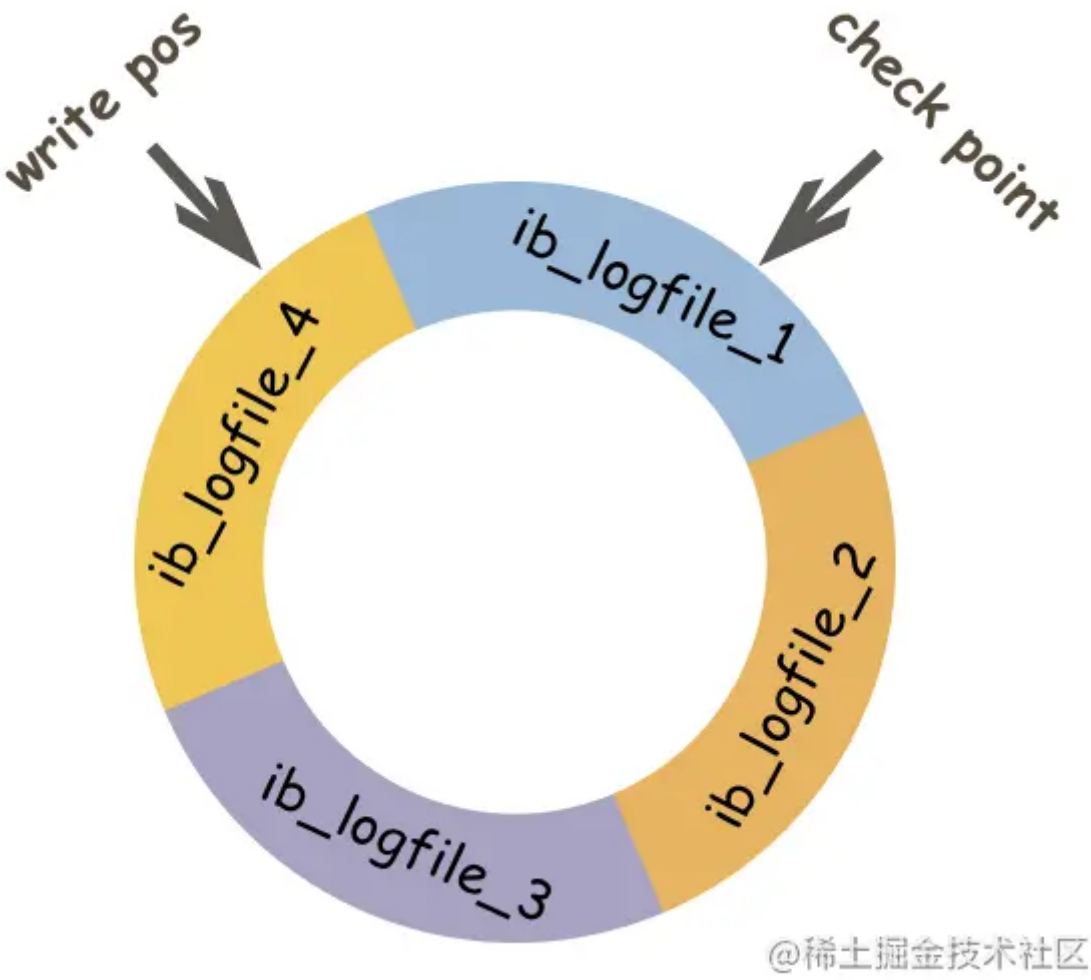
`mysql` 支持三种将 `redo log buffer` 写入 `redo log file` 的时机，可以通过 `innodb_flush_log_at_trx_commit` 参数配置，各参数值含义如下：

参数值	含义
0（延迟写）	事务提交时不会将 `redo log buffer` 中日志写入到 `os buffer`，而是每秒写入 `os buffer` 并调用 `fsync()` 写入到 `redo log file` 中。也就是说设置为0时是(大约)每秒刷新写入到磁盘中的，当系统崩溃，会丢失1秒钟的数据。
1（实时写，实时刷）	事务每次提交都会将 `redo log buffer` 中的日志写入 `os buffer` 并调用 `fsync()` 刷到 `redo log file` 中。这种方式即使系统崩溃也不会丢失任何数据，但是因为每次提交都写入磁盘，IO的性能较差。
2（实时写，延迟刷）	每次提交都仅写入到 `os buffer`，然后是每秒调用 `fsync()` 将 `os buffer` 中的日志写入到 `redo log file`。



redo log记录形式及数据恢复过程

`redo log`实际上记录数据页的变更，而这种变更记录是没必要全部保存，因此`redo log`实现上采用了大小固定，循环写入的方式，当写到结尾时，会回到开头循环写日志。如下图：



同时我们很容易得知，**在innodb中，既有`redo log`需要刷盘，还有`数据页`也需要刷盘，`redo log`存在的意义主要就是降低对`数据页`刷盘的要求。**

在上图中，`write pos`表示`redo log`当前记录的`LSN`(逻辑序列号)位置，`check point`表示**数据页更改记录**刷盘后对应`redo log`所处的`LSN`(逻辑序列号)位置。`write pos`到`check point`之间的部分是`redo log`空着的部分，用于记录新的记录；`check point`到`write pos`之间是`redo log`待落盘的数据页更改记录。

当`write pos`追上`check point`时，会先推动`check point`向前移动，空出位置再记录新的日志。

启动`innodb`的时候，不管上次是正常关闭还是异常关闭，总是会进行恢复操作。因为`redo log`记录的是数据页的物理变化，因此恢复的时候速度比逻辑日志(如`binlog`)要快很多。

重启`innodb`时，首先会检查磁盘中数据页的`LSN`，如果数据页的`LSN`小于日志中的`LSN`，则会从`checkpoint`开始恢复。

还有一种情况，在宕机前正处于`checkpoint`的刷盘过程，且数据页的刷盘进度超过了日志页的刷盘进度，此时会出现数据页中记录的`LSN`大于日志中的`LSN`，这时超出日志进度的部分将不会重做，因为这本身就表示已经做过的事情，无需再重做。

undo log

数据库事务四大特性中有一个是**原子性**，具体来说就是 **原子性是指对数据库的一系列操作，要么全部成功，要么全部失败，不可能出现部分成功的情况。**

实际上，**原子性**底层就是通过`undo log`实现的。

`undo log`主要记录了数据的逻辑变化，比如一条`INSERT`语句，对应一条`DELETE`的`undo log`，对于每个`UPDATE`语句，对应一条相反的`UPDATE`的`undo log`，这样在发生错误时，就能回滚到事务之前的数据状态。

同时，`undo log`也是`MVCC`(多版本并发控制)实现的关键。

undo log和redo log是mysql独有的技术吗?

不是，Oracle也有。

relay log

从服务器I/O线程将主服务器的二进制日志（binlog）读取过来记录到从服务器本地文件（relay log），然后SQL线程会读取relay-log日志的内容并应用到从服务器，从而使从服务器和主服务器的数据保持一致

索引

索引的分类（单列，组合）

单列索引(普通索引，唯一索引，主键索引)、**组合索引**（联合索引/多列索引）

单列索引

- 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。（唯一指数据唯一）
- 普通索引: 基本的索引类型，没有唯一性的限制，允许为NULL值 ，可创建多个

组合索引（联合索引/多列索引）

联合索引是指对表上的多个列进行索引，联合索引也是一棵B+树

- **覆盖索引**:一种特殊的联合索引,即是你查询的字段的所有数据都在索引上，不需要再进行一次回表查询，这样的索引即为覆盖索引。

MySQL 聚集索引 (主键索引) 和非聚集索引 (辅助索引/普通索引) 的区别

聚集索引：（找到了索引就找到了需要的数据，那么这个索引就是聚簇索引）

聚集索引一般都是加在主键上的。聚集索引就是按照每张表的主键构造一棵 B+ 树，同时叶子节点中存放的即为表中一行一行的数据，所以聚集索引的叶子节点也被称为数据节点。也就是说，聚集索引能够在 B+ 树索引的叶子节点上直接找到数据。

可以这么说：在聚集索引中，索引即数据，数据即索引

非聚集索引：（找到了索引但没找到数据，需要根据索引上的值(主键)再次回表查询,非聚簇索引也叫做辅助索引。）

非聚集索引也叫做辅助索引，和聚集索引的最大区别就在于，辅助索引的叶子节点并不包含行记录的全部数据。

辅助索引的叶子节点包含的是：每行数据的辅助索引键 + 该行数据对应的聚集索引键，也就是主键ID

当通过辅助索引来寻找数据时，InnoDB 存储引擎会先遍历辅助索引的 B+ 树，通过叶子节点获得某个辅助索引键对应的聚集索引键，然后再通过聚集索引来找到一个完整的行记录。

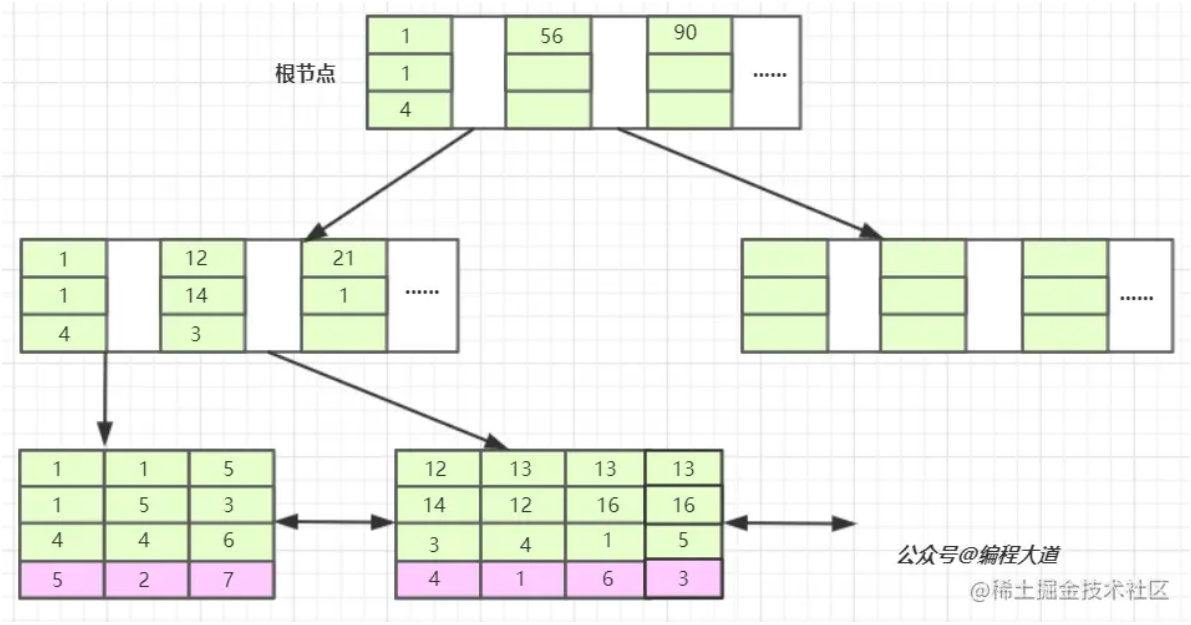
> MyisAM引擎没有聚簇索引。主键索引树和辅助索引树都是非聚集索引。

联合索引

联合索引的存储结构

`联合索引（b，c，d）`也会生成一个索引树，同样是B+树的结构，只不过它的data部分存储的是联合索引所在行的主键值（紫色）。

对于联合索引来说只不过比单值索引多了几列，而这些索引列全都出现在索引树上。对于联合索引，存储引擎会首先根据第一个索引列排序，如果第一列相等则再根据第二列排序，依次类推。

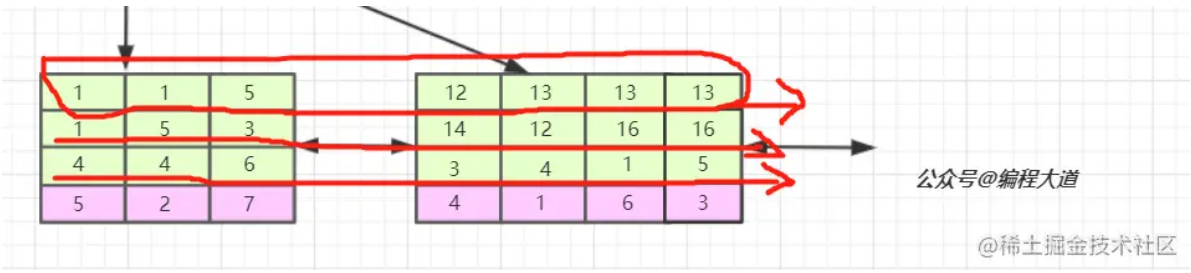


最左前缀匹配原则

之所以会有最左前缀匹配原则和联合索引的索引构建方式及存储结构是有关系的。

首先我们创建的`index_bcd(b,c,d)`索引，相当于创建了(b)、（b、c）（b、c、d）三个索引，看完下面你就知道为什么相当于创建了三个索引。

我们看，联合索引是首先使用多列索引的第一列构建的索引树，用上面idx_t1_bcd(b,c,d)的例子就是优先使用b列构建，当b列值相等时再以c列排序，若c列的值也相等则以d列排序。我们可以取出索引树的叶子节点看一下。



索引的第一列也就是b列可以说是从左到右单调递增的，但我们看c列和d列并没有这个特性，它们只能在b列值相等的情况下这个小范围内递增，如第一叶子节点的第1、2个元素和第二个叶子节点的后三个元素。 由于联合索引是上述那样的索引构建方式及存储结构，所以联合索引只能从多列索引的第一列开始查找。所以如果你的查找条件不包含b列如（c,d）、(c）、(d)是无法应用缓存的，以及跨列也是无法完全用到索引如(b,d)，只会用到b列索引。

```
select * from T1 where b = 12 and c = 14 and d = 3;-- 全值索引匹配 三列都用到
select * from T1 where b = 12 and c = 14 and e = 'xml';-- 应用到两列索引
select * from T1 where b = 12 and e = 'xml';-- 应用到一列索引
select * from T1 where b = 12  and c >= 14 and e = 'xml';-- 应用到bc两列列索引及索引条件下推优化
select * from T1 where b = 12  and d = 3;-- 应用到一列索引  因为不能跨列使用索引 没有c列 连不上
select * from T1 where c = 14  and d = 3;-- 无法应用索引，违背最左匹配原则
```

联合索引有必要得用区分度最高的列作为最左字段吗?

有必要。在创建联合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，其次区分度高的能缩小查询范围，所以结合实际场景，选择合适的多列索引的顺序

```
(a,b,c)的联合索引和(c,b,a)的联合索引，对于a,b,c三个字段的等值查询，都能利用上吗
比如字段a的取值是0-100，b是0-1000，c是0-10000，三个字段的联合索引要以什么顺序建，为什么

Mysql 建立联合索引时，基数大的排在前面。因为基数大的话，字段数据的区分度高。
```

> **不要在小基数字段上建立索引**
> 索引基数是指这个字段在表里总共有多少个不同的值，比如一张表总共100万行记录，其中有个性别字段，
> 其值不是男就是女，那么该字段的基数就是2。
> 如果对这种小基数字段建立索引的话，还不如全表扫描了，因为你的索引树里就包含男和女两种值，根本没
> 法进行快速的二分查找，那用索引就没有太大的意义了。
> 一般建立索引，尽量使用那些基数比较大的字段，就是值比较多的字段，那么才能发挥出B+树快速二分查
> 找的优势来。

前缀索引

前缀索引了解吗，为什么要建前缀索引

前缀索引就是选取字段的前几个字节建立索引。首先，InnoDB 限制了每列索引的最大长度不能超过 767 字节，所以，对于某些比较长的字段，如果确实有建立索引的必要，使用前缀索引不仅能够避免索引长度超过限制，而且相对于普通索引来说，占用的空间和查询成本更小。

> 避免索引过长，减少占用空间和查询成本

前缀索引可能产生什么问题?

第一个，使用前缀索引可能会增加记录扫描次数与回表次数，影响性能。针对这一点呢，其实前缀索引长度的选取还是很重要的，可能前缀定义的长一点，就能够大幅减少记录扫描次数和回表次数，所以，在建立前缀索引的时候，我们需要在占用空间和搜索效率之间做一个权衡

> 也就是选取的长度不合理，导致数据的区分度不高。增加了扫描的次数

第二个，使用前缀索引其实就没法用覆盖索引对查询性能的优化了，因为 InnoDB 并不能确定前缀索引的定义是否截断了完整信息，就算是完全踩中了前缀索引，InnoDB 还得回表确认一次到底是不是满足条件了。

> 无法使用覆盖索引原则

建立前缀索引需要注意的是什么?

事实上，**我们在建立前缀索引时关注的是区分度，区分度越高，意味着重复的键值越少，所以区分度越高越好。** 对于索引来说，什么是区分度呢，很简单，就是这个索引上有多少个不同的值。建立出来的索引上拥有越多不同的值，那么这个索引的区分度就越高。 因此，我们可以通过统计索引上有多少个不同的值来判断要使用多长的前缀

前缀索引的区分度不够高怎么办——Hash

字段(假设这个字段名是 a)超级超级长，远大于 InnoDB 的限制 767 字节，普通索引肯定是不可能了，前缀索引就算是长度定义成 767 都还是存在区分度不高的情况，但是又存在根据这个字段进行查询的挺频繁的一个需求。

一个很常见的解决手段就是 Hash。

对这个超长字段 a 进行 hash(假设命名为 a_hash) 存入数据库，然后对这个 hash 值建立索引，由于 hash 值同样可能存在冲突，也就是说两个不同的 a 通过 Hash 函数得到的结果可能是相同的，所以我们在查询语句的 where 部分还需要进行一次精确判断

假设输入的字段是 input_a， `select * from user where hash(input_a) = a_hash and input_a = a;` 不过使用 Hash 这种方式有个众所周知的缺点，那就是不支持范围查询了，只能等值查询。

设计索引的原则

- 为那些经常作为查询条件的字段建立索引，频繁更新的字段不建索引
- 避免为“大字段”建立索引，减少索引占用更多存储空间
- 选择区分度大的字段作为索引，也就是基数较大的，基数太小（性别2），优化器会倾向于全表扫描
- 不要建太多索引，优先考虑扩展索引而不是创建索引。因为索引也需要维护

索引有哪些优缺点?

优点:

1,唯一索引可以保证数据库表中每一行的数据的唯一性

2,索引可以加快数据查询速度，减少查询时间

缺点:

1,创建索引和维护索引要耗费时间

2,索引需要占物理空间，除了数据表占用数据空间之外，每一个索引还要占用一定的物理空间

3,对表中的数据进行增、删、改的时候，索引也要动态的维护。

三大范式

第一范式（1NF）：要求数据库表的每一列都是不可分割的原子数据项。

> `系名/系主任`这一列数据的时候感觉这并不合符我们数据库的设计理念，因为它完全可以拆分为两个列的。

第二范式：需要确保数据库表中的每一列都和主键相关，而不能只与主键的某一部分相关（主要针对联合主键而言）。

> 举例说明：
>
> | 订单号 | 产品号 | 产品数量 | 产品折扣 | 产品价格 | 订单金额 | 订单时间 |
> | --- | --- | --- | --- | --- | --- | --- |
> | 2008003 | 205 | 100 | 0.9 | 8.9 | 2870 | 20080103 |
> | 2008003 | 206 | 200 | 0.8 | 9.9 | 2870 | 20080103 |
> | 2008005 | 207 | 200 | 0.75 | 10 | 2000 | 20080203 |
> | 2008006 | 207 | 400 | 0.85 | 12 | 4800 | 20080206 |
> | 2008007 | 207 | 1000 | 0.88 | 14 | 14000 | 20080209 |
> | 2008008 | 210 | 240 | 0.95 | 8 | 12255 | 20100423 |
> | 2008008 | 211 | 300 | 0.75 | 8 | 12255 | 20100423 |
> | 2008008 | 212 | 350 | 0.8 | 15.9 | 12255 | 20100423 |
>
> 在上图所示的情况中，同一个订单中可能包含不同的产品，因此主键必须是“订单号”和“产品号”联合组成，
>
> 但可以发现，产品数量、产品折扣、产品价格与“订单号”和“产品号”都相关，但是订单金额和订单时间仅与“订单号”相关，与“产品号”无关，

第三范式：需要确保数据表中的每一列数据都和主键直接相关，而不能间接相关。

> 举例说明：
>
> | 学号 | 姓名 | 性别 | 家庭人口 | 班主任姓名 | 班主任性别 | 班主任年龄 |
> | --- | --- | --- | --- | --- | --- | --- |
> | 20150001 | 李白 | 男 | 3口人 | 陈洁 | 女 | 35 |
> | 20150002 | 杜甫 | 男 | 2口人 | 陈洁 | 女 | 35 |
> | 20150003 | 王维 | 男 | 4口人 | 陈洁 | 女 | 35 |
> | 20150004 | 白居易 | 男 | 3口人 | 李丽 | 女 | 32 |
> | 20150005 | 刘禹锡 | 男 | 4口人 | 李丽 | 女 | 32 |
> | 20150006 | 李清照 | 女 | 5口人 | 王安 | 男 | 29 |
> | 20150007 | 苏轼 | 男 | 2口人 | 南林 | 男 | 34 |
> | 20150008 | 屈原 | 男 | 4口人 | 南林 | 男 | 34 |
> | 20150009 | 陶渊明 | 男 | 1口人 | 王安 | 男 | 29 |
>
> 上表中，所有属性都完全依赖于学号，所以满足第二范式，但是“班主任性别”和“班主任年龄”直接依赖的是“班主任姓名”，

主从复制

形式

一主一从

一主多从

一主一从和一主多从是我们现在见的最多的主从架构，使用起来简单有效，不仅可以实现 HA，而且还能读写分离，进而提升集群的并发能力。

多主一从

多主一从可以将多个 MySQL 数据库备份到一台存储性能比较好的服务器上。

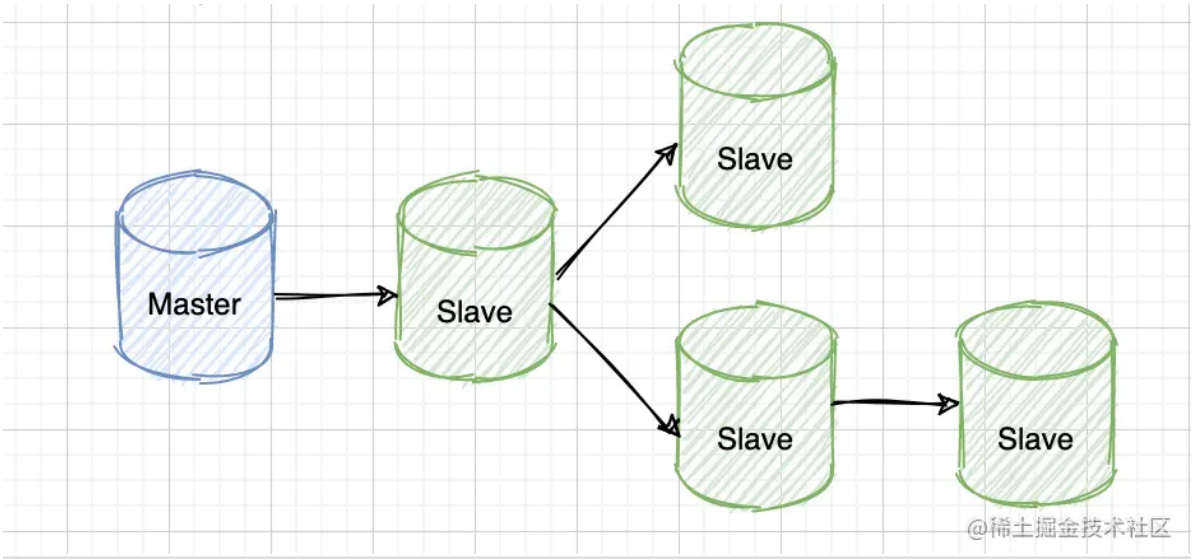
> MySQL 可不可以多主多从? 可以，用binlog同步

双主复制

双主复制，也就是可以互做主从复制，每个 master 既是 master，又是另外一台服务器的 salve。这样任何一方所做的变更，都会通过复制应用到另外一方的数据库中。

> Mysql 双主相互备份是如何解决循环复制的?
>
> 假设A，B 相互备份，A产生的binlog 中携带 serverId=1，发送到B，B解析后执行，执行完毕也会产生binlog，此时B的binlog serverId 仍然等于1，发送会A后，A看到serverId=1为自己发送的binlog，直接丢弃；同理B方向类似。
>
> Mysql 三主如何同步？会造成无线同步循环吗?
>
> 环形结构，每个节点接受来自前序节点的写入，并将这些写入（加上自身的写入）转发给后序节点。
>
> 在这个结构中，写请求需要通过多个节点才能到达所有副本，即中间节点需要转发从其他节点接收到的数据变更。
>
> 为了防止循环同步问题，每个节点需要赋予一个唯一标识符，在binlog日志中的每个写请求都标记已经通过的节点标识符。如果某个节点收到了包含自身标识符的数据更改，表明该更新请求已被处理过，因此会忽略此变更请求，避免重复转发。

级联复制



级联复制模式下，部分 slave 的数据同步不连接主节点，而是连接**从节点**。

因为如果主节点有太多的从节点，就会损耗一部分性能用于 replication ，那么我们可以让 3~5 个从节点连接主节点，其它从节点作为二级或者三级与从节点连接，这样不仅可以**缓解**主节点的压力，并且对**数据一致性**没有负面影响。

原理

MySQL 主从复制是基于主服务器在二进制日志binlog日志，跟踪所有对数据库的更改操作。因此，要进行复制，必须在主服务器上启用二进制日志。

每个从服务器接收来自主服务器的binlog数据。当从服务器连接到主服务器时，从服务器会通知主服务器，从服务器日志中读取最后一个更新成功的位置。

从服务器接收从那时发生起的任何更新，并在主机上执行相同的更新。然后封锁等待主服务器通知的更新。

从服务器执行备份不会干扰主服务器，在备份过程中主服务器可以继续处理更新。

MySQL的主从复制过程

主从复制分了五个步骤进行：

步骤一：主库的更新事件(update、insert、delete)被写到binlog(所有DDL和DML产生的日志)

步骤二：当从服务器连接主服务器时，主服务器会创建一个 log dump 线程，用于发送 binlog 的内容。在读取 binlog 的内容的操作中，会对象主节点上的 binlog **加锁**，当读取完成并发送给从服务器后解锁。

步骤三：当从节点上执行 `start slave` 命令之后，从节点会创建一个 IO 线程用来连接主节点，请求主库中**更新** binlog。IO 线程接收主节点 binlog dump 进程发来的更新之后，保存到 relay-log 中。

步骤四：从节点 创建SQL 线程负责读取 realy-log 中的内容，**解析**成具体的操作执行，将主库的DDL和DML操作事件重放，最终保证主从数据的一致性。

类型

异步复制

这种模式下，主节点**不会主动推送数据**到从节点，主库在执行完客户端提交的事务后会立即将结果返给给客户端，并不关心从库是否已经接收并处理。

不用等到从库接收到并写到relay log 中才返回成功信息给客户端。

存在的问题：新主节点缺失数据

主节点如果崩溃掉了，此时主节点上已经提交的事务可能并没有传到从节点上，如果此时，强行将从提升为主，可能导致新主节点上的数据不完整。

同步复制

当主库执行完一个事务，然后所有的从库都复制了该事务并**成功执行完**才返回成功信息给客户端。

存在的问题：性能较差

因为需要等待所有从库执行完该事务才能返回成功信息，所以全同步复制的性能必然会收到严重的影响。

半同步复制

介于异步复制和全同步复制之间，主库在执行完客户端提交的事务后不是立刻返回给客户端，而是等待**至少一个从库接收到并写到** relay log 中才返回成功信息给客户端(只能保证主库的 Binlog 至少传输到了一个从节点上)，否则需要等待直到**超时时间**然后切换成异步模式再提交。

优点：比异步复制安全，比全同步复制性能好

相对于异步复制，半同步复制提高了数据的**安全性**，一定程度的保证了数据能成功备份到从库，同时它也造成了一定程度的延迟，但是比全同步模式延迟要低，这个延迟最少是一个 TCP/IP 往返的时间。所以，半同步复制最好在**低延时**的网络中使用。

半同步模式不是 MySQL 内置的，从 `MySQL 5.5` 开始集成，需要 master 和 slave 安装插件开启半同步模式。

延迟复制

在异步复制的基础上，人为设定主库和从库的数据**同步延迟时间**，即保证数据延迟至少是这个参数。

复制的方式（对应binlog的三种格式）

语句复制（statement）

基于语句的复制相当于**逻辑复制**，即二进制日志中记录了操作的语句，通过这些语句在从数据库中重放来实现复制。

这种方式简单，二进制文件小，传输带宽占用小。但是基于语句更新依赖于其它因素，比如插入数据时利用了时间戳或者其他函数。重放的时候数据会不一致。

因此在开发当中，我们应该尽量将业务逻辑逻辑放在**代码层**，而不应该放在 MySQL 中，不易拓展。

特点:

- 传输效率高，减少延迟。
- 在从库更新不存在的记录时，语句赋值不会失败。而行复制会导致失败，从而更早发现主从之间的不一致。
- 设表里有一百万条数据，一条sql更新了所有表，基于语句的复制仅需要发送一条sql，而基于行的复制需要发送一百万条更新记录

行数据复制

基于行的复制相当于**物理复制**，即二进制日志中记录的实际更新数据的每一行。

这样导致复制的压力比较大，日志占用的空间大，传输带宽占用大。但是这种方式比基于语句的复制要更加**精确**。

特点:

- 不需要执行查询计划。
- 不知道执行的到底是什么语句。
- 例如一条更新用户总积分的语句，需要统计用户的所有积分再写入用户表。如果是基于语句复制的话，从库需要再一次统计用户的积分，而基于行复制就直接更新记录，无需再统计用户积分。

混合类型的复制

一般情况下，默认采用**基于语句**的复制，一旦发现基于语句无法精确复制时，就会采用基于行的复制。

存在的问题

延迟

当主库的 TPS 并发较高的时候，由于主库上面是多线程写入的，而从库的SQL线程是单线程的，导致从库SQL可能会跟不上主库的**处理速度**。

解决方法:

- 网络方面：尽量保证主库和从库之间的**网络稳定**，延迟较小；
- 硬件方面：从库**配置更好**的硬件，提升随机写的性能；
- 配置方面：尽量使 MySQL 的操作在**内存中完成**，减少磁盘操作。或升级 MySQL5.7 版本使用并行复制，提高SQL线程数
- 建构方面：把一台从服务器单独作为备份使用，而不提供查询。查询负载下来了，执行relay log里面的SQL效率就能提高了，或增加从服务器

数据丢失

当主库宕机后，数据可能丢失。

解决方法:

使用**半同步**复制，可以解决数据丢失的问题。

为什么要做主从同步？主从复制的作用

> 数据安全，性能提升，扩展性高，负载均衡

主服务器出现问题，可以**切换**到从服务器；

可以进行数据库层面的**读写分离**；

可以在从数据库上进行日常**备份**。

还可以保证：

1. 数据更安全：做了**数据冗余**，不会因为单台服务器的宕机而丢失数据；
2. 性能大大提升：一主多从，不同用户从不同数据库读取，**性能提升**；
3. 扩展性更优：流量增大时，可以方便的**增加**从服务器，不影响系统使用；
4. 负载均衡：一主多从相当于分担了主机任务，做了**负载均衡**。

应用场景

横向扩展

将工作负载**分发**到各 Slave 节点上，从而提高系统性能。

在这个场景下，所有的写(write)和更新(update)操作都在 Master 节点上完成；所有的读(read)操作都在 Slave 节点上完成。通过**增加更多**的 Slave 节点，便能提高系统的读取速度。

数据安全

数据从 Master 节点复制到 Slave 节点上，在 Slave 节点上可以**暂停**复制进程。可以在 Slave 节点上**备份**与 Master 节点对应的数据，而不用影响 Master 节点的运行。

数据分析

实时数据可以在 Master 节点上创建，而分析这些数据可以在 Slave 节点上进行，并且不会对 Master 节点的性能产生影响。

拆分访问

可以把几个不同的从服务器，根据公司的**业务**进行拆分。通过拆分可以帮助减轻主服务器的压力，还可以使数据库对外部用户浏览、内部用户业务处理及 DBA 人员的备份等**互不影响**。

读写分离的数据库，是否有必要建立聚簇与非聚簇索引?

很有必要，从库用来做查询，主库用来做插入等操作，这种情况下，从库建立索引增加查询速度，而建立索引，插入会相应变慢，所以主库不建立索引

#

SQL注入

sql注入到底有哪些危害?

1. 核心数据泄露

大部分攻击者的目的是为了赚钱，说白了就是获取到有价值的信息拿出去卖钱，比如：用户账号、密码、手机号、身份证信息、银行卡号、地址等敏感信息。

他们可以注入类似这样的语句：

```
-1; select * from user;--
复制代码
```

就能轻松把用户表中所有信息都获取到。

所以，建议大家对这些敏感信息加密存储，可以使用`AES`对称加密。

2. 删库跑路

也不乏有些攻击者不按常理出牌，sql注入后直接把系统的表或者数据库都删了。

他们可以注入类似这样的语句：

```
-1; delete from user;--
复制代码
```

以上语句会删掉user表中所有数据。

```
-1; drop database test;--
复制代码
```

以上语句会把整个test数据库所有内容都删掉。

正常情况下，我们需要控制线上账号的权限，只允许DML（data manipulation language）数据操纵语言语句，包括：select、update、insert、delete等。

不允许DDL（data definition language）数据库定义语言语句，包含：create、alter、drop等。

也不允许DCL（Data Control Language）数据库控制语言语句，包含：grant,deny,revoke等。

DDL和DCL语句只有dba的管理员账号才能操作。

顺便提一句：如果被删表或删库了，其实还有补救措施，就是从备份文件中恢复，可能只会丢失少量实时的数据，所以一定有备份机制。

3. 把系统搞挂

他们可以注入类似这样的语句：

```
-1;锁表语句;--
```

把表长时间锁住后，可能会导致数据库连接耗尽。

这时，我们需要对数据库线程做监控，如果某条sql执行时间太长，要邮件预警。此外，合理设置数据库连接的超时时间，也能稍微缓解一下这类问题。

从上面三个方面，能看出sql注入问题的危害真的挺大的，我们一定要避免该类问题的发生，不要存着侥幸的心理。如果遇到一些不按常理出票的攻击者，一旦被攻击了，你可能会损失惨重。

如何防止sql注入

1. 使用预编译机制

尽量用预编译机制，少用字符串拼接的方式传参，它是sql注入问题的根源。

> `preparestatement`预编译机制会在sql语句执行前，对其进行语法分析、编译和优化，其中参数位置使用占位符`?`代替了。
>
> 当真正运行时，**传过来的参数会被看作是一个纯文本**，不会重新编译，不会被当做sql指令。
>
> 这样，即使入参传入sql注入指令如：
>
> ```
> id; select 1 --
> ```
>
> 最终执行的sql会变成：
>
> ```
> select * from test1 order by 'id; select 1 --' limit 1,20
> ```
>
> 这样就不会出现sql注入问题了。

2. 要对特殊字符转义

有些特殊字符，比如：`%`作为`like`语句中的参数时，要对其进行转义处理。

3. 要捕获异常

需要对所有的异常情况进行捕获，切忌接口直接返回异常信息，因为有些异常信息中包含了sql信息，包括：库名，表名，字段名等。攻击者拿着这些信息，就能通过sql注入随心所欲的攻击你的数据库了。目前比较主流的做法是，有个专门的网关服务，它统一暴露对外接口。用户请求接口时先经过它，再由它将请求转发给业务服务。这样做的好处是：能统一封装返回数据的返回体，并且如果出现异常，能返回统一的异常信息，隐藏敏感信息。此外还能做限流和权限控制。

4. 使用代码检测工具

使用sqlMap等代码检测工具，它能检测sql注入漏洞。

5. 要有监控

需要对数据库sql的执行情况进行监控，有异常情况，及时邮件或短信提醒。

6. 数据库账号需控制权限

对生产环境的数据库建立单独的账号，只分配`DML`相关权限，且不能访问系统表。切勿在程序中直接使用管理员账号。

7. 代码review

建立代码review机制，能找出部分隐藏的问题，提升代码质量。

8. 使用其他手段处理

对于不能使用预编译传参时，要么开启`druid`的`filter`防火墙，要么自己写代码逻辑过滤掉所有可能的注入关键字。

（todo）Mysql有哪几种锁，乐观锁与悲观锁的区别，行锁与表锁的区别，间隙锁的作用与应用场景

MySQL的锁

分类

全局锁

全局锁可以使整个数据库处于只读状态，这时其他线程DML，DDL操作，都会被阻塞。

全局锁应用场景是什么?

- 全局锁**主要用于全库的备份逻辑**，这样在备份数据库期间，不会因为数据或表结构的更新，而出现备份文件的数据与预期不一致。

加全局锁又会带来什么缺点呢?

- 加上全局锁，意味着整个数据库都是只读状态。

  那么如果数据库里有很多数据，备份就会花费很多的时间，关键是备份期间，业务只能读数据，而不能更新数据，这样会造成业务停滞

既然备份数据库数据的时候，使用全局锁会影响业务，那有什么其他方式可以避免?

- InnoDB的存储引擎可以在可重复读的隔离级别下，开启事务进行备份操作。
- MyISAM因为不支持事务，所以只能用全局锁进行备份。

表级锁

MySQL 里面表级别的锁有这几种：

- 表锁；
- 元数据锁（MDL）；
- 意向锁；
- AUTO-INC 锁（自增锁）

一、表锁

有两种模式：

- 表共享读锁 （Table Read Lock）：不会阻塞其他用户对同一表的读请求，但会阻塞对同一表的写请求；
- 表独占写锁 （Table Write Lock）：会阻塞其他用户对同一表的读和写操作；

> 默认情况下，写锁比读锁具有更高的优先级：当一个锁释放时，这个锁会优先给写锁队列中等候的获取锁请求，然后再给读锁队列中等候的获取锁请求

二、元数据锁

MDL 是为了保证当用户对表执行 CRUD 操作时，防止其他线程对这个表结构做了变更。相反，也可以在进行表结构变更操作时，防止其他线程执行CRUD操作。

> 当有线程在执行 select 语句（ 加 MDL 读锁）的期间，如果有其他线程要更改该表的结构（ 申请 MDL 写锁），那么将会被阻塞，直到执行完 select 语句（ 释放 MDL 读锁）。

> 反之，当有线程对表结构进行变更（加 MDL 写锁）的期间，如果有其他线程执行了 CRUD 操作（申请 MDL 读锁），那么就会被阻塞，直到表结构变更完成（释放 MDL 写锁）。

MDL 会有什么问题?

申请 MDL 锁的操作会形成一个队列，队列中**写锁获取优先级高于读锁**，一旦出现 MDL 写锁等待，会阻塞后续该表的所有 CRUD 操作。

而数据库中的长事务，也就是一直事务一直不提交，后续如果我们要进行表结构的变更，可能就会导致MDL写锁等待。

所以为了能安全的对表结构进行变更，在对表结构变更前，先要看看数据库中的长事务，是否有事务已经对表加上了 MDL 读锁，如果可以考虑 kill 掉这个长事务，然后再做表结构的变更。

三、意向锁

在使用 InnoDB 引擎的表里对某些记录加上「共享锁」之前，也就是加行锁前，需要先在表级别加上一个「意向共享锁」；
在使用 InnoDB 引擎的表里对某些纪录加上「独占锁」之前，也就是加行锁前，需要先在表级别加上一个「意向独占锁」；
也就是，当执行插入、更新、删除操作，需要先对表加上「意向独占锁」，然后对该记录加独占锁。
而普通的 select 是不会加行级锁的，普通的 select 语句是利用 MVCC 实现一致性读，是无锁的。

不过，select 也是可以对记录加共享锁（in share mode）和独占锁(for update)的

意向锁的作用? 为了快速判断表里是否有记录被加锁。

独占表锁和独占行锁是不能共存的。

如果没有「意向锁」，那么加「独占表锁」时，就需要遍历表里所有记录，查看是否有记录存在独占锁，这样效率会很慢。

那么有了「意向锁」，由于在对记录加独占锁前，先会加上表级别的意向独占锁，那么在加「独占表锁」时，直接查该表是否有意向独占锁，如果有就意味着表里已经有记录被加了独占锁，这样就不用去遍历表里的记录。申请[独占表锁]的线程会被阻塞。

> 注意：申请意向锁的动作是数据库完成的，就是说，当申请一行的行锁的时候，数据库会自动先开始申请表的意向锁，不需要我们程序员使用代码来申请。

四、AUTO-INC 锁

- **定义**

  在为某个字段声明 AUTO_INCREMENT 属性时，之后可以在插入数据时，可以不指定该字段的值，数据库会自动给该字段赋值递增的值，这主要是通过 AUTO-INC 锁实现的。

  自增锁是特殊的表锁机制，锁不是事务提交后才释放，而是在执行插入语句后就释放。

  在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 AUTO_INCREMENT 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。

- **AUTO-INC 锁有什么问题?**

  在插入数据时，会加一个表级别的 AUTO-INC 锁，然后为被 `AUTO_INCREMENT` 修饰的字段赋值递增的值，等插入语句执行完成后，才会把 AUTO-INC 锁释放掉。

  那么，一个事务在持有 AUTO-INC 锁的过程中，其他事务的如果要向该表插入语句都会被阻塞，从而保证插入数据时，被 `AUTO_INCREMENT` 修饰的字段的值是连续递增的。

  所以， AUTO-INC 锁再对大量数据进行插入的时候，会影响插入性能，因为另一个事务中的插入会被阻塞。

  因此， 在 MySQL 5.1.22 版本开始，InnoDB 存储引擎提供了一种**轻量级的锁**来实现自增。

- **如何提高自增的性能？轻量级锁和AUTO-INC锁如何选取?**

> 轻量级锁，性能好，但是并发场景下自增值可能不是连续的。
>
> AUTO-INC，保证自增值连续，但是性能较差。

一样也是在插入数据的时候，会为被 `AUTO_INCREMENT` 修饰的字段加上轻量级锁，**然后给该字段赋值一个自增的值，就把这个轻量级锁释放了，而不需要等待整个插入语句执行完后才释放锁。**

InnoDB 存储引擎提供了个 innodb_autoinc_lock_mode 的系统变量，是用来控制选择用 AUTO-INC 锁，还是轻量级的锁。

- 当 innodb_autoinc_lock_mode = 0，就采用 AUTO-INC 锁；
- 当 innodb_autoinc_lock_mode = 2，就采用轻量级锁；
- 当 innodb_autoinc_lock_mode = 1，这个是默认值，两种锁混着用，如果能够确定插入记录的数量就采用轻量级锁，不确定时就采用 AUTO-INC 锁。

不过，当 innodb_autoinc_lock_mode = 2 是性能最高的方式，但是会带来一定的问题。因为并发插入的存在，在每次插入时，自增长的值可能不是连续的，**这在有主从复制的场景中是不安全的。**

行锁

记录锁
记录锁就是为某行记录加锁，它封锁该行的索引记录。

间隙锁（Gap Locks）
它锁定一段范围内的索引记录。使用间隙锁锁住的是一个区间，而不仅仅是这个区间中的每一条数据。

Next-Key Lock：
Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。

行锁是怎么解决幻读问题的?

> 普通select - > MVCC
>
> 当前读 select .... in share mode；select .... for update； 使用临键锁

这样，当你执行 select * from user where name = 'Jack' for update ；（name是非唯一索引）的时候，就不止是给数据库中已有的 n 个记录加上了行锁，还同时加了 n + 1 个间隙锁（这两个合起来也成为 Next-Key Lock 临键锁）。也就是说，在数据库一行行扫描的过程中，不仅扫描到的行加上了行锁，还给行两边的空隙也加上了锁。这样就确保了无法再插入新的记录。

其主要通过两个方面实现这个目的：

- 防止间隙内有新数据被插入
- 防止已存在的数据，更新成间隙内的数据（例如防止numer=3的记录通过update变成number=5）

> 这里多提一嘴，update、delete 语句用不上索引是很恐怖的。
>
> 对非索引字段进行 select .. for update、update 或者 delete 操作，由于没有索引，走全表查询，就会对所有行记录 以及 所有间隔 都进行上锁。而对于索引字段进行上述操作，只有索引字段本身和附近的间隔会被加锁。

总结下 MySQL 解决幻读的手段：

隔离级别：可重复读

- 快照读 MVCC + 当前读 Next-Lock Key(只在可重复读隔离级别下生效)

隔离级别：SERIALIZABLE

- 在这个隔离级别下，事务在读操作时，先加表级别的共享锁，直到事务结束才释放；事务在写操作时，先加表级别的排它锁，直到事务结束才释放。也就是说，串行化锁定了整张表，幻读不存在的

Inner join与left join区别，左连接、右连接、内连接、外连接的区别

Mysql查询优化器机制

关系型数据库与非关系型数据库区别

sql 语句的执行顺序

MySQL 处理重复数据

MySQL 处理重复数据 | 菜鸟教程 (runoob.com)

Mysql产生死锁的原因及解决方案

Drop delete truncate的比较

Mysql如何实现分页查询？，深分页优化

主键为什么自增，不自增是否可行

> 为什么主键用自增不用UUID

mysql事务回滚及提交的原理

Mysql三级封锁协议

Mysql数据存储形式

浅谈Mysql数据存储 - 掘金 (juejin.cn)