MySQL

基础

数据库范式

三大范式

第一范式（1NF）：**表中每一列都是不可分割的原子项。**字段只能是一个值，不能再分为多个其他的字段了

第二范式（2NF）：**在满足1NF的基础上，消除了非主属性对主属性的部分依赖。**主要是说在联合主键的情况下，非主键列不能只依赖于主键的一部分，要完全依赖所有主键。

学号、课程、分数

第三范式（3NF）：**在满足2NF的基础上，消除了非主属性之间的传递依赖**，要求表中的非主键属性之间没有传递依赖关系。

举例：学生表（学号，身份证号，姓名）

意义

减少数据冗余：同样的信息被多次存储。**例如，在学生选课表中（学号、课程号、姓名），学生的姓名会随着他选课的增多而重复存储。**

修改异常：如果需要修改某个信息，可能需要在多个地方进行修改，而且如果不小心遗漏了某个地方，就**会导致数据不一致。例如，如果需要更新某个学生的姓名，就必须逐行更改各个包含该学生姓名的记录进行更新。**

删除异常：如果删除了某个实体与其他实体的关联，就可能导致该实体的信息丢失。**例如，如果某个学生退选了所有课程，那么同时也会丢失关于这个学生的姓名等信息。**

插入异常：如果某个实体尚未与其他实体建立关联，就无法将其插入到数据库中。**例如，如果有一个新学生还没有选课，那么就无法在表中插入他的姓名。**

主键与外键的区别

主键(主码)：主键用于唯一标识一个元组，**不能有重复，不允许为空。一个表只能有一个主键。**

外键(外码)：外键用来和其他表建立联系用，**外键是另一表的主键，外键是可以有重复的，可以是空值。一个表可以有多个外键。**

外键的优缺点

数据库设计的步骤

什么是关系型数据库与非关系型数据库

原理

MySQL的执行流程

- **连接器**：mysql是基于tcp协议传输数据的，所以要进行tcp三次握手与服务器建立连接
- **Server缓存**：如果是查询语句，就先会根据sql语句先去缓存中查找结果，命中直接返回结果，未命中则继续执行。因为命中率极低，MySQL 8.0 版本直接将查询缓存删掉了，
- **解析器**：通过分析器进行**词法分析和语法分析**，提取出SQL语句中的关键字，判断SQL语法是否正确，并构建出语法树
- **预处理器和优化器**：进入预处理阶段，先**检查语句中表或字段是否存在，将select的*扩展成所有的列**，在优化阶段，优化器会**基于查询成本，选择合适的索引**，
- **执行器**：通过执行器在存储引擎进行数据存储或读取，返回结果。

InnoDB Compat行格式

MySQL 5.1 版本之后，行格式默认设置成 Compact。

记录的额外信息

- **变长字段长度列表**：记录各个变长字段varchar(n)真实占用字节数
- **NULL值列表**：最少1字节，二进制位1和0分别表示字段是否为NULL。NULL 值列表必须用整数个字节的位表示如果使用的二进制位个数不足整数个字节，则在字节的高位补0。**NULL值列表会占用 1 字节空间，当表中所有字段都定义成 NOT NULL，行格式中就不会有 NULL值列表，这样可节省 1 字节的空间。**
- **记录头信息**：包含多个字段，delete_mask、delete_mask，record_type等

记录的真实数据

包含三个隐藏字段和记录中每个列的值

- row_id 如果我们建表的时候指定了主键或者唯一约束列，那么就没有 row_id 隐藏字段了。如果既没有指定主键，又没有唯一约束，那么 InnoDB 就会为记录添加 row_id 隐藏字段。row_id不是必需的，占用 6 个字节。
- trx_id 这个数据是由哪个事务生成的，占用 6 个字节
- roll_pointer 记录上一个版本的指针。占用 7 个字节

> 为什么「变长字段长度列表」以及NULL值列表 的信息要按照逆序存放?

这样可以使记录中位置靠前的字段和它们对应的字段长度信息在内存中的距离更近，被CPU读取时更有可能在同一页中，可能会提高高速缓存的命中率

> varchar(n) 中 n 最大取值为多少?

一行变长字段长度列表和 NULL 值列表和数据列最大65535字节。

65535B - 1B空值列表 - 2B变长字段长度 = 65532B

因此，n字符数最大为65532

> 行溢出后，MySQL 是怎么处理的?

一行记录的数据列最大65535字节，而MSQL数据页固定16kB，因此可能出现行溢出的情况。

如果一个数据页存不了一条记录，InnoDB 存储引擎会自动将溢出的数据存放到「溢出页」中。然后真实数据处用 20 字节存储指向溢出页的地址

存储引擎

5.5版本之后mysql默认使用InnoDB存储引擎

MyISAM与InnoDB存储引擎有什么区别?

- MyISAM 不支持**外键**，而 InnoDB 支持。
- MyISAM 不提供**事务**支持。InnoDB 提供事务支持
- **MyISAM只支持表级锁**，更新时会锁住整张表，导致其它查询和更新都会被阻塞，**InnoDB支持行级锁。**
- 读写不能并发，**InnoDB性能更强**

怎么查看SQL执行计划

在查询语句最前面加个 explain 命令，这样就会输出这条 SQL 语句的执行计划，包括使用了哪些索引

```
explain select * from blog where id = 1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	blog	(Null)	const	PRIMARY	PRIMARY	4	const	1	100.00	(Null)

索引

什么是索引?

索引就是一种**将数据库中的记录按照特殊形式存储的一种有序的数据结构。**就类似一本书的目录（举例），它维护了表中某些列的值以及对应的行指针。能够显著地提高数据查询的效率。

索引有哪些种类?

从数据结构的角度

- **B+树索引**:所有数据存储在叶子节点，复杂度为O(logn)，适合范围查询。
- **哈希索引**:适合等值查询，检索效率高，一次到位
- **全文索引**: MyISAM 和 InnoDB 中都支持使用全文索引，一般在文本类型char，text，varchar类型上创建

从物理存储的角度

- **聚簇存储**： **B+Tree 的叶子节点存放的是实际数据**，所有完整的用户记录都存放在主键索引的 B+Tree 的叶子节点里； 聚簇索引在创建表时自动创建，默认就是主键索引，聚簇索引只能有一个。
- **二级存储**： **B+Tree 的叶子节点存放的是主键值**，而不是实际数据，**要进行回表。** 手动创建的索引都是是二级存储的。

基本类型分类

- **主键索引**:建立在主键字段上的索引，一张表只能有一个主键索引，主键值不允许有空值。
- **唯一索引**:索引列中的值必须是唯一的，一张表可以有多个，但是允许为空值
- **普通索引**:MySQL中基本索引类型，允许空值和重复值
- **联合索引**:为多个字段创建的索引，使用时遵循最左前缀原则

覆盖索引

在查询时使用了二级索引，**如果查询的列能在二级索引里查询的到，那么就不需要回表，这个过程就是覆盖索引。**如果查询的数据不在二级索引里，就会先检索二级索引，找到对应的叶子节点，**获取到主键值后，然后再检索主键索引，就能查询到数据了，这个过程就是回表。**

索引下推

在联合索引遍历过程中，对联合索引中包含的字段先做判断，直接过滤掉不满足条件的记录，减少回表次数。

InnoDB 数据页查找，B+树查找过程

InnoDB 的数据是按「数据页」为单位来读写的，数据页的默认大小是 16KB

每一页包含一个**页目录**，通过通过页目录进行**二分查找**找到对应的记录。时间复杂度为O(logn)

在B+树中每个节点都是一个数据页，它们通过指针构成一个树结构，其中非叶子节点中的页是索引页，存的是索引，也就是其他页的定位信息。叶子节点是数据页，里面存的是实际的数据。

找到对应记录的定位信息后就可以定位到下次层中的页节点，直到叶子节点，然后根据页目录二分查找到对应记录

讲一讲索引数据结构，为什么使用B+树?

可以作为索引的数据结构有很多种，但最好的数据结构应在**尽可能少的磁盘IO完成查询，且能够高效的查询一个记录和范围查询。**

哈希表是键值对的集合，通过键(key)即可快速取出对应的值(value)，查询效率是O(1)，**但是会产生碰撞冲突，通过拉链法解决又会影响性能，而且哈希表不支持范围查询。**

二叉搜索树是一种天然的二分结构，利用二分查找定位数据，但是它存在一种极端的情况，就会导致二分查找树**退化成一个链表**，此时查询复杂度就会从 O(logn)降低为 O(n)。

平衡二叉树。左右子树高度之差不超过1，它解决了二分查找树退化成链表的问题。但是它**本质上还是一个二叉树**，每个节点只能有 2 个子节点，随着元素的增多，**树的高度会越来越高，磁盘io次数也越多。**

（红黑树与平衡二叉树相似，红黑树并不追求严格的平衡，而是大致的平衡。正因如此，红黑树的**查询效率稍有下降**，但插入和删除操作效率大大提高，插入和删除节点时只需进行 O(1) 次数的旋转和变色操作，即可保持基本平衡状态，而不需要像 AVL 树一样进行 O(logn) 次数的旋转操作。）

B树和B+树，都是通过多叉平衡树的方式，会将树的高度变矮，所以这两个数据结构非常适合检索存于磁盘中的数据。

- **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，相比存储只存索引又存记录的 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「矮胖」**，查询底层节点的磁盘 I/O次数会更少。
- B 树的检索的过程相当于对范围内的每个节点的关键字做二分查找，可能还没有到达叶子节点，检索就结束了。而 **B+ 树的检索效率就很稳定**了，任何查找都是从根节点到叶子节点的过程，叶子节点的顺序检索很明显
- **B+ 树节点之间用双向链表连接了起来，遍历链表即可进行范围查询**，而 B 树要进行范围查询，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 I/O 操作，范围查询效率不如 B+ 树。

综上，B+ 树与 B 树相比，具备**更少的 IO 次数、更稳定的查询效率和更适于范围查询**这些优势。

页分裂

聚簇索引与二级索引比较

二级索引更新代价小，但是需要回表查询效率低

聚簇索引更新代价大，但查询效率高

索引优缺点

优点:

提高了查询效率

通过创建唯一性索引，保证数据每一行的唯一性

缺点:

- 需要**占用物理空间**，数据数量越大，索引占用空间越大;
- 会**降低表的增删改的效率**，因为每次增删改数据时都要修改索引，B+ 树为了保证索引有序性，都需要进行动态维护，而且数据量越大，性能损失越大。

什么时候需要/不需要索引?

以下情况需要创建索引

- **经常使用where查询**的字段，建立索引提高查询效率
- **被频繁查询的字段**：我们创建索引的字段应该是查询操作非常频繁的字段。
- **频繁需要排序的字段**：索引已经排序，这样查询可以利用索引的排序，加快排序查询时间。
- **需要唯一性约束时**，必须对其创建唯一性索引

以下情况不需要创建索引

- **表记录太少**，使用索引优化不大

- **很少被where指定查询的列**不需要建立索引，创建索引可能没有明显的性能提升。
- **频繁更新的字段**不用创建索引，由于要维护 B+Tree的有序性，那么就需要频繁维护B+树，这个过程是会影响数据库性能的
- **字段中存在大量重复数据且分布平均**，不需要创建索引，比如性别字段，只有男女，如果数据库表中，男女的记录分布均匀，那么无论搜索哪个值都要查找一半的数据。

最左匹配原则与索引下推

最左匹配原则

联合索引B+树叶子节点数据中只有第一个索引字段是有序的，后面的字段都是局部有序的，也就是只有在前面字段都相等的情况下才是有序的。因此联合索引查询有一个最左匹配原则

最左匹配原则 **就是要求查询条件中的列必须按照联合索引中列的顺序从左到右依次出现，才能有效利用索引。联合索引查询中如果某个索引字段没有出现或者匹配失败，那么这个索引字段以及后面的索引字段都不会被利用到，而是通过当前查询到的id进行回表查询，然后再通过这些没判定过的字段进行过滤，增加了回表查询的工作量。**

注意：

- 范围查询会让后面的字段匹配失败，而左模糊或左右模糊匹配会让所有字段匹配失败，因为第一个字段无法匹配成功。
- 查询条件中的字段的顺序对匹配无影响，因为mysql的优化器会自动优化顺序选择索引。

讲一下索引下推

索引下推是mysql5.6推出的 联合索引 **查询优化方案**，在5.6版本之前的联合索引查询中，如果某些字段最左匹配原则匹配失败，那么这些字段就不会走索引，而是在进行回表查询后，将查询到的数据交给server层通过这些查询条件进行进一步过滤，才得出最终的结果。而索引下推做出了优化，将查询条件下推到存储引擎层面进行处理，也就是直接通过索引字段的值进行判断是否符合条件进行筛选，然后再去执行回表查询。

- 减少了回表查询的次数，降低了磁盘IO开销。
- 避免将不符合条件的数据行读取到内存中，从而减少不必要的数据读取和传输。

索引优化方式?

覆盖索引优化

覆盖索引即需要查询的字段正好是索引的字段，那么直接根据该索引，就可以查到数据了，而无需回表查询。，可以避免回表查询聚簇索引的操作，减少了大量的磁盘IO操作。

假设我们现在有个业务需要查询商品的名称、价格（举例），我们**可以建立一个联合索引**，即「商品ID、名称、价格」作为一个联合索引。因为叶子节点索引中存在我要的这些字段，查询将不会再次检索主键索引，而**避免回表。**

前缀索引优化

前缀索引顾名思义就是**使用某个字段中字符串的前几个字符建立索引，减小索引字段大小，可以增加一个索引页中存储的索引值，B+树的层次也就更少，有效提高索引的查询速度。**通过可以为一些大字符串的字段建立前缀索引，但前缀索引无法用作覆盖索引；

使用自增主键

对于主键索引，尽量使用自增主键，因为每次**插入一条新记录主键都是最大的，插入时都是追加操作，不需要重新移动数据**，因此这种插入数据的方法效率非常高。当页面写满，就会自动开辟一个新页面。

如果我们使用非自增主键，由于每次插入主键的索引值都是随机的，就可能会**插入到现有数据页中间的某个位置**，造成页分裂问题，这将不得不移动其它数据，甚至需要从一个页面复制数据到另外一个页面。 页分裂 **还有可能会造成大量的内存碎片，导致索引结构不紧凑，从而影响查询效率。**

防止索引失效

使用sql查询语句时防止索引失效而全表扫描。

索引失效

索引失效有哪些情况

- **最左匹配原则**：联合索引查询不符合**最左匹配原则**可能会有部分索引列失效，比如等值查询某些字段缺失或使用了<、>、like等范围查询，都会导致这个索引字段后面的字段失效
- **左模糊或左右模糊查询**：当我们使用**左或者左右模糊匹配**的时候，也就是 `like %xx` 或者 `like %xx%`这两种方式都会造成索引失效，不符和最左匹配原则；
- **or查询**：在 WHERE 子句中，如果在 OR 条件有字段不是索引，那只能全表扫描。
- **使用函数**：当我们在查询条件中**对索引列使用函数**，就会导致索引失效。`where length(name)=6;`因为索引保存的是索引字段的原始值，而不是经过函数计算后的值，自然就没办法走索引了
- **出现隐式类型转换**：查询条件中如果列类型不一致，就可能出现隐式类型转换，比如索引字段是整数，而将它与字符串进行比较。

事务

说说ACID/事务的特性

原子性（Atomicity）：事务的所有操作是一个整体，要么全部成功，要么全部失败，不能只完成一部分

一致性（Consistency）：执行事务前后，数据保持一致，例如转账业务中，无论事务是否成功，转账者和收款人的总额应该是不变的；

隔离性（Isolation）：数据库允许多个并发事务同时对其数据进行读写和修改的能力。不会出现多个事务并发执行而导致数据不一致的情况。

持久性（Durability）：事务处理结束后，对数据的修改就是永久的。

并发事务会引发什么问题?

脏读：如果一个事务修改了数据还没有提交，此时另一个数据读到了这个未提交但是修改过的数据，就意味着发生了「脏读」现象。

不可重复读：在一个事务内多次读取同一个数据，另一个事务修改了数据，导致出现前后两次读到的数据不一样的情况，就意味着发生了「不可重复读」现象。

幻读：在一个事务内多次查询某一范围的数据，另一个事务新增或删除了数据，导致出现前后两次查询到的记录数量不一样的情况，就意味着发生了「幻读」现象。

事务的隔离级别，InnoDB使用哪种级别?

- **读未提交（read uncommitted）**，指一个事务还没提交时，它做的变更就能被其他事务看到；
- **读提交（read committed）**，指一个事务提交之后，它做的变更才能被其他事务看到；**避免了脏读**
- **可重复读（repeatable read）**，指一个事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的。**避免了不可重复读**
- **串行化（serializable）**，串行化隔离级别下，数据库系统确保事务之间的并发执行具有串行化的效果，即每个事务都像是按顺序执行一样，不存在并发执行的情况

InnoDB存储引擎使用的是可重复读级别

说一下你了解的MVCC

MVCC，**多版本并发控制**，是**读已提交**和**可重复读**事务隔离级别**无锁**的底层实现方式。主要由两部分组成一个是`undo log`版本控制以及`readview`快照隔离。

当一个事务每次对数据记录进行修改后，就会把旧版本的数据存进`undo`日志中，通过`roll_pointer`回滚指针连接起来构成一个链表，于是就可以通过它找到修改前的记录。

当事务读取数据时，通过快照可以判断当前版本的数据记录是否可以读取，

（readview快照记录了创建当前快照时活跃中的事务id列表以及最小和最大的活跃事务id。当事务读取数据记录时会创建一个快照，并通过行记录中的**trx_id即最后修改该记录的事务id**与**快照中活跃的id列表**进行判断来决定是否可读。

- 如果trx_id小于当前活跃事务的最小id，表明创建这个版本的记录的事务已经提交了，可以读取这个数据
- 如果trx_id大于当前活跃事务的最大id，表明这个版本记录是在创建快照后生成的，不可以读取这个数据
- 如果trx_id在最小和最大活跃事务的id列表之间的话，就要判断它是否在活跃事务id列表中，如果在的话表明这个事务还未提交，不能读取，否则可以读取。）

如果不能读取的话要**沿着undo log版本链往下找一个旧的记录，进行同样的判断，直到查找到一个合适的版本的记录。**

读提交和可重复读隔离级别的实现区别就在于生成read view的时机不同。

- 读提交隔离级别是在**每次读取数据前都会重新生成一个** `Read View`，保证了每次读取到的数据都是别的事务已经提交后的数据。
- 可重复读隔离级别**是启动事务时生成一个 Read View，**然后整个事务期间都在用这个 Read View。保证了事务中每次重复读取得到的数据都是一样的。

幻读是怎么解决的?

- 针对**快照读**（普通 select 语句），是**通过 MVCC 方式读取旧版本的数据解决了幻读**，因为可重复读隔离级别下，事务执行过程中看到的数据，一直跟这个事务启动时看到的数据是一致的，即使中途有其他事务插入了一条数据，是查询不出来这条数据的，所以就很好了避免幻读问题。
- 针对**当前读**（select ... for update 等语句），因为当前读只会读取最新数据，**通过 next-key lock（记录锁+间隙锁）方式解决了幻读**，因为当执行 select ... for update 语句的时候，会加上 next-key lock，如果有其他事务在 next-key lock 锁范围内插入了一条记录，那么这个插入语句就会被阻塞，无法成功插入，所以就很好了避免幻读问题。

可重复读隔离级别完全解决了幻读吗?

通过mvcc以及临建锁并没有完全解决幻读问题，还有一些特殊情况。

- **当前事务通过update，修改了另一个事务insert的新数据。**导致新数据的undo log版本链中的trx_id = 当前事务id。由于trx_id == creator_id，就让事务误以为这条数据是自己创建的，所以导致ReadView可以看见这条数据了。
- **如果当前事务第一次快照读之后，另一个事务插入了一条数据，而当前事务又进行当前读**，那么就可以读取到这条新插入的记录。

锁

怎么查询加锁情况

我们可以通过 `select * from performance_schema.data_locks\G;` 这条语句，查看事务执行 SQL 过程中加了什么锁。

查询了performance_schema数据库中的data_locks表信息。

表级锁

表级锁锁住的是一整张表，锁粒度最大，**并发程度很低，但不会出现死锁问题**

元数据锁（MDL）

对数据库表进行操作时，会自动给这个表加上元数据锁，防止crud操作时，变更表的结构，或者变更表的结构时，进行crud操作。

意向锁有什么作用

当我们需要给一个表加表锁的时候，我们需要根据去判断表中有没有数据行被锁定，以确定是否能加成功，通过意向锁这个表级锁可以快速判断表里是否有记录被加锁**。当对行记录加上共享锁或独占锁时，需要在表级别加上意向共享锁或意向独占锁。

AUTO-INC 自增锁

当表中主键设置成自增主键后，**在插入数据时，会加一个表级别的 AUTO-INC 锁**，然后为自增主键赋自增的值，执行完成后，才会把 AUTO-INC 锁释放掉。从而**保证插入数据时字段的值是连续递增的。**

行级锁

锁住某些记录或某些间隙，行级锁锁的粒度很小，**事务的并发度很高，但会出现死锁。**

- **记录锁（record lock）**：锁住的是一条索引记录。防止记录被其他事务修改，解决不可重复读的问题，事务提交时释放。而且记录锁是有 S 锁和 X 锁之分的
- **间隙锁**：锁住的是一个范围，只存在于可重复读隔离级别，防止记录的解决可重复读隔离级别下幻读的现象。间隙锁之间是兼容的，两个事务可以同时持有包含共同间隙范围的间隙锁，并不存在互斥关系。
- **Next-Key Lock**：Next-Key Lock临键锁，是 Record Lock + Gap Lock 的组合，锁定一个范围，并且锁定记录本身。next-key lock 即能保护该记录，又能阻止其他事务将新纪录插入到被保护记录前面的间隙中。
- **插入意向锁**：一个事务在插入一条记录的时候，需要判断插入位置是否已被其他事务加了间隙锁（next-key lock 也包含间隙锁）。如果有的话，插入操作就会发生阻塞，直到拥有间隙锁的那个事务提交为止，在此期间会生成一个插入意向锁，表明有事务想在某个区间插入新记录，但是现在处于等待状态。

表级锁和行级锁了解吗？有什么区别?

表级锁锁住的是一整张表，锁粒度最大，并发程度很低，触发锁冲突的概率最高，但不会出现死锁问题

行级锁锁住某些记录或某些间隙，行级锁锁的粒度很小，并发度很高，大大减少数据库操作的冲突，但加锁的开销也最大，还会出现死锁。

mysql加锁规则

加锁规则要看具体的where条件，等值查询和范围查询会决定加临键锁还是记录锁还是间隙锁。

普通的查询操作不会加锁，共享读会加S型记录锁，当前读会加X型记录锁，

不仅是查询会加行锁，修改操作也会加行锁，而插入操作加插入意向锁，

行级锁使用注意（锁全表）

InnoDB 的行锁是针对索引字段加的锁。在执行 update、delete、select ... for update 等具有加锁性质的语句，**如果没走索引而进行全表扫描的话，会对每一个记录加 next-key 锁，相当于把整个表锁住了，当前事务无法执行其他需要加锁的语句，这是挺严重的问题。**

mysql中的死锁，以及如何解决?

mysql中的死锁是因为多个事务互相操作已经被锁定的资源，都等待对方锁的释放以至无法继续执行。

- 首先事务a修改了记录1，对其加了记录锁
- 此时事务b修改了记录2，也对其加了记录锁
- 然后事务a去修改记录2，而记录2已经被事务b锁定，所以事务a只能阻塞
- 事务b这时也去修改记录1，而记录1已经被事务a锁定，所以事务b也会阻塞

日志

Update语句执行流程（含日志）

- 执行器首先要查询出要修改的记录，先会从Buffer Pool缓存中找
- 开启事务，InnoDB 层**更新记录前**，首先要记录相应的 undo log，将其记录到buffer pool缓存中的undo log日志页中并标记为脏页，同时用redo log日志记录对undo log日志页的修改。

- 然后更新内存并标记为脏页，**更新记录后**将页面的修改记录到 redo log 里面，后续由后台线程选择一个合适的时机将脏页写入到磁盘。这就是 **WAL 技术**
- 更新语句执行完成后，server层开始记录该语句对应的 binlog
- 最后，两阶段提交事务。

mysql有哪些日志文件?

- **错误日志（error log）**：错误日志文件对MySQL的启动、运行、关闭过程进行了记录，能帮助定位MySQL问题。
- **慢查询日志（slow query log）**：**记录了执行时间超过指定阈值的查询语句。**通过慢查询日志，可以知道哪些查询语句的执行效率很低，以便进行优化。
- **一般查询日志（general log）**：一般查询日志**记录了客户端对MySQL服务器的所有SQL记录**，无论请求是否正确执行。
- **二进制日志（bin log）**：**记录了所有更改数据库数据和表结构的SQL语句**

还有两个 InnoDB 存储引擎特有的日志文件：

- 重做日志（redo log）
- 回滚日志（undo log）

undo log

回滚日志，记录了旧版本的数据信息

每当对数据进行修改前，会把旧版本的数据信息记录到undo log中。**两大作用：**

- **实现事务回滚，保障事务的原子性。**事务处理过程中，如果出现了错误或者用户执行了 ROLLBACK 语句，MySQL 可以利用 undo log 中的历史数据执行相反的操作将数据恢复到事务开始之前的状态。
- **实现 MVCC（多版本并发控制）关键因素之一。**MySQL 在执行快照读（普通 select 语句）的时候，会根据事务的 Read View 里的信息，顺着 undo log 的版本链找到满足其可见性的旧的记录。控制数据记录对事务的可见性，从而解决事务并发中的不可重复读问题。

Buffer Pool 缓存

Buffer Pool是InnoDB的缓存，减少了磁盘IO次数，提高了读写效率

- 当读取数据时，如果数据存在于 Buffer Pool 中，客户端就会直接读取 Buffer Pool 中的数据，否则再去磁盘中读取。
- 当修改数据时，如果数据存在于 Buffer Pool 中，那直接修改 Buffer Pool 中数据所在的页，然后将其页设置为脏页（该页的内存数据和磁盘上的数据已经不一致），为了减少磁盘I/O，不会立即将脏页写入磁盘，**后续由后台线程选择一个合适的时机将脏页写入到磁盘。**

redo log

redolog 是重做日志，记录了事务对某页数据的修改，当事务对某页的数据进行修改后，会将这个页的修改**以 redo log 的形式记录下来**（对 XXX 表空间中的 YYY 数据页 ZZZ 偏移量的地方做了AAA 更新）

- **确保事务的持久化。**每次修改数据的时候，并不会立即修改磁盘中的数据，而是先修改当前的Buffer Pool缓存的数据页，后台再慢慢的将缓存中的脏页数据同步的磁盘。而缓存是基于内存，如果断电重启，**那么脏页数据就丢失了**，但数据库重启后通过redo log就可以读取到修改的操作，重做所有已记录的事务操作，这些已提交的记录就不会丢失，从而确保了事务的持久化，**为mysql提供了崩溃恢复的能力。**

redo log 要写到磁盘，数据也要写磁盘，为什么不直接写入数据

因为Redo Log日志是以追加的形式写入的，是顺序写，写入磁盘时是顺序IO，性能较高。而数据页的刷盘操作通常是随机写，也就是随机IO，性能较差

bin log

它**记录了数据库中所有的执行语句**，包括DDL和DML语句，保存在二进制文件中。它与redo log很相似，它就相当于**数据库备份**，**可以用来进行主从复制。**

bin log与redo log有什么区别?

- **层次和存储引擎**：bin log是mysql server层的日志，**所有存储引擎都可以使用**，而redo log是innoDB存储引擎特有的，**只记录InnoDB存储引擎的日志。**
- **记录内容**：bin log是**逻辑日志**，记录的是一个事务执行的具体操作语句，而redo log是**物理日志**，记录的是每个数据页的更改情况。
- **写入方式不同**：bin log是**追加写**，写满一个文件，就创建一个新的文件继续写，不会覆盖以前的日志，保存的是全量的日志。redo log 是**循环写**，日志空间大小是固定，全部写满就从头开始覆盖写，保存未被刷入磁盘的脏页日志。
- **作用不同**：redo log用于掉电等故障恢复，bin log用于备份恢复、主从复制等。

主从复制（定义，原理）

主从复制是用来建立一个与主数据库完全一样的数据库环境，即从数据库。

原理

binlog 二进制文件，记录了数据可执行的所有 SQL 语句。主从同步的目标就是把主数据库的 binlog 文件中的 SQL 语句复制到从数据库，让其在从数据的 relaylog 文件中再执行一次这些 SQL 语句即可。

需要三个线程：

- **binlog输出线程**：每当有从库连接到主库时，主库都会创建一个binlog输出线程然后发送 binlog内 容到从库
- **从库IO接收线程：**当 `START SLAVE` 语句在从库开始执行之后，从库创建一个 IO 线程，该线程连接到主库并接收binlog 里面的更新记录到从库中的relay log中继日志中
- **从库 SQL 线程**：读取中继日志回放binlog，更新数据，实现主从数据库一致性

为什么要两阶段提交？两阶段提交过程

事务提交的时候，将提交过程分为了两个阶段，从而保证重做日志和二进制日志都能正常写入磁盘，从而保证主从数据库的一致性。

分库分表

Redis

认识Redis

Redis是什么?

Redis 是一种基于内存的数据库系统，对数据的读写操作都是在内存中完成，因此读写速度非常快，使用键值对存储数据，并支持多种数据结构.

关系型数据库和非关系型数据库有什么区别?

存储方式：关系型数据库以表格的形式**结构化存储数据。**非关系型数据库没有固定的格式

性能：关系型数据库在处理大量数据时可能会面临性能瓶颈，非关系型数据库处理能力更强。

数据一致性：关系型数据库支持事务，可以保证数据的一致性和可靠性

灵活性：结构化查询语言（SQL）进行数据查询和操作，支持复杂的数据查询和分析。非关系型数据库更灵活，不受数据格式限制。

常用数据类型及应用场景

string

内部有**int和SDS动态字符串**实现，常用于：

- 缓存对象：存对象的json，set user:1 '{"name":"xiaolin", "age":18}'
- 常规计数：
- 分布式锁：

list

内部由**双向列表和压缩列表实现（3.2版本quicklist）**，常用于：

- 消息队列

hash

内部由**哈希表和压缩列表**实现

常用于缓存一个对象的各种属性

- 缓存对象：set user name:zs age:10
- 购物车：用户 id 为 key，商品 id 为 field，商品数量为 value

set

内部由**哈希表和整数集合**实现，常用于：

- 点赞：因为不能重复点赞，为每个文章存点赞用户id，sadd article:1 user:1
- 共同关注：集合运算，交集计算

zset

内部由**哈希表和跳表（3.2版本listpack）**实现，常用于：

- 排行榜：
- 电话姓名排序（字典排序）：

跳表原理

跳表利用了二分查找的特性，就是一个多层级的链表，上一层的节点个数大概是下一层的一半，每次查找元素的时候从最顶层开始找到第一个大于等于查找值的节点，如果值不相等，就向下一层往回查找，到小于等于查找值的节点，然后顺序查找，重复这个过程。

redis为什么那么快?

基于内存：Redis是一个内存数据库，所有数据都存储在内存中，读写操作在内存中完成，避免了磁盘I/O带来的延迟

单线程模型：redis是单线程模型，所有核心操作都是在一个线程中进行的，没有多线程竞争锁的性能消耗，也没有线程切换的开销。

io多路复用：通过iO多路复用，通过一个单独的线程管理多个连接，减少了线程创建和销毁的开销，降低了系统资源的占用。

redis线程模型

redis是单线程吗?

Redis 单线程指的是「**接收客户端请求->解析请求 ->进行数据读写等操作->发送数据给客户端**」这个过程是由一个线程（主线程）来完成的，这也是我们常说Redis 是单线程的原因。但是，Redis 程序并不是单线程的，Redis 在启动的时候，是会启动后台线程

为什么redis后面要有多线程?

持久化

AOF日志原理/优缺点

追加写操作：当有写操作（增删改）发生时，Redis会将这些写操作以文本的形式追加到AOF文件末尾。

文件同步：Redis会通过系统调用将AOF文件的内容强制刷写到磁盘上，以保证数据的持久性。

恢复数据：在Redis重启的时候，会通过加载AOF文件中保存的写操作来恢复数据，重建内存中的数据状态。

文件重写：当AOF文件变得过大时，会触发AOF文件的重写。后台子线程会将当前内存中的数据重写到一个新的AOF文件中，并且优化写入操作，减小AOF文件的体积。

RDB快照原理

redis可以将当前内存中的数据通过子线程保存在文件当中，每次重启redis，会通过加载RDB文件将数据重新读入内存中。

redis默认使用的是RDB持久化

AOF与RDB的比较

时效性与性能

AOF持久化**时效性**更好，相比于RDB定期快照记录，AOF持久化会实时追加写到AOF文件中。避免了数据丢失的风险。每次写操作都要记录到AOF文件中，文件的体积很大，而且对**性能**有影响。

RDB持久化定期进行的，所以对redis的性能影响小，但如果发生故障还没来得及持久化可能会有数据的丢失

混合持久化机制

RDB 优点是**数据恢复速度快，但是快照的频率不好把握**。频率太低，丢失的数据就会比较多，频率太高，就会影响性能。

AOF 优点是**丢失数据少，但是数据恢复不快。**

为了集成了两者的优点， Redis 4.0 提出了混合使用 AOF 日志和内存快照，也叫混合持久化，AOF文件的**前半部分是 RDB 格式的全量数据，后半部分是 AOF 格式的增量数据。既保证了 Redis 重启速度，又降低数据丢失风险**

功能篇

过期删除怎么实现的?

Redis 会把该 key 带上过期时间存储到一个**过期字典**（expires dict）中，也就是说「过期字典」保存了数据库中所有 key 的过期时间。

字典实际上是哈希表，哈希表的最大好处就是让我们可以用 O(1) 的时间复杂度来快速查找。当我们查询一个 key 时，Redis 首先检查该 key 是否存在于过期字典中：

- 如果不在，则正常读取键值；
- 如果存在，则会获取该 key 的过期时间，然后与当前系统时间进行比对，如果比系统时间大，那就没有过期，否则判定该 key 已过期

三种删除策略

定时删除：在设置key的过期时间时，同时设置一个定时事件，当时间到达时，由事件处理器自动执行 key 的删除操作。

- 内存可以被尽快地释放
- 占用相当一部分 CPU 时间

惰性删除：不主动删除过期键，每次从数据库访问 key 时，都检测 key 是否过期，如果过期则删除该 key

- 对 CPU 时间最友好。
- 如果过期 key 一直没有被访问，它所占用的内存就不会释放，造成了一定的内存空间浪费

定期删除：每隔一段时间「随机」从数据库中取出一定数量的 key 进行检查，并删除其中的过期key

- 减少了删除操作对 CPU 的影响，同时减少了过期键对空间的无效占用。
- 内存清理方面没有定时删除效果好，同时没有惰性删除使用的系统资源少。

Redis 选择「惰性删除+定期删除」这两种策略配和使用，以求在合理使用 CPU 时间和避免内存浪费之间取得平衡。

内存淘汰策略

当redis内存满了后，内存淘汰策略删除符合条件的 key。具体有8中内存淘汰策略

- 默认策略不进行数据淘汰，直接返回错误

在过期时间数据中进行淘汰：

- 随机淘汰设置了过期时间的任意键值
- 优先淘汰更早过期的键值
- 淘汰所有设置了过期时间的键值中，最久未使用的键值（lru）
- 淘汰所有设置了过期时间的键值中，最少使用的键值(lfu)

在所有数据中进行淘汰：

- 随机淘汰任意键值
- 淘汰整个键值中最久未使用的键值（lru）
- 淘汰整个键值中最少使用的键值。(lfu)

3.Redis持久化怎么处理过期键的?

RDB方式再生成RDB文件时，会直接忽略已过期的键

AOF方式下，对于已过期键写入redis命令时，会加入一条del命令显示删除该键

4.主从模式下怎么处理过期键的?

高可用

1.Redis 如何实现服务高可用?

主从复制、哨兵、集群

2.主从复制（意义、策略）

如果数据只存储在一台服务器上，硬盘出现了故障，可能数据就都丢失了。要避免这种单点故障，最好的办法是将数据备份到其他服务器上。**主从复制**解决了多台服务器之间的数据一致性问题。

主从复制策略

首先建立连接进行第一次同步，也是**全量同步**，主服务器会执行 bgsave 命令来生成 RDB 文件，然后把文件发送给从服务器

完成第一次同步后，双方之间就会维护一个 TCP 连接。主服务器可以继续将写操作命令传播给从服务器，即**增量同步**

增量同步失败可能是因为网络断开，从服务器与主服务器断开连接，而从服务器需要的同步缓冲已经被覆盖了。然而主服务器任何时候都可以发起全量同步，其主要策略就是无论如何首先会尝试进行增量同步，如果失败则会要求 slave 进行全量同步。

3.哨兵模式原理

哨兵其实是主从模式下一个运行在特殊模式下的 Redis 进程，所以它也是一个节点。

监控

哨兵可以监控所有主从节点，判断它们是否正常运行。如果监控到主节点无响应，哨兵就会判断其为**主观下线**。然后向其它哨兵发起投票，达到某一票数后，主节点就会被标记为**客观下线**

选主

然后就通过vote投票算法从所有节点中再选出一个主节点。更改相关配置

通知

哨兵会通过发布者/订阅者模式通知客户端主节点已经发生变化。客户端就可以跟新主节点进行通信了。

缓存问题

什么是缓存雪崩、缓存击穿、缓存穿透，各自怎么解决?

缓存雪崩

大量缓存数据在同一时间过期或者Redis故障宕机，大量请求直接访问数据库，从而导致数据库压力骤增，严重导致数据库宕机。

- **均匀设置过期时间**，避免数据都同一时间失效。
- **热点数据永不过期**，而是让热点数据缓存“永久有效”，并将更新缓存的工作交由后台线程定时更新。
- **启动服务熔断机制**，暂停业务应用对数据库的访问，直接返回错误

缓存击穿

某个热点数据过期了，此时大量的请求访问了该热点数据，就无法从缓存中读取，直接访问数据库，**数据库很容易就被高并发的请求冲垮**

同雪崩

- **热点数据永不过期**，而是让热点数据缓存“永久有效”，并将更新缓存的工作交由后台线程定时更新。
- **启动服务熔断机制**，暂停业务应用对数据库的访问，直接返回错误
- **限流**，限流机制则可以限制对某个热点数据的访问频率，以减轻数据库压力。

缓存穿透

当大量请求查询不存在的数据，缓存未命中，查询数据库，但是数据库不能回写缓存，导致不存在的数据的一直会去访问数据库，导致数据库崩溃。

- **使用布隆过滤器快速判断数据是否存在**，避免通过查询数据库来判断数据是否存在。
- **回写缓存空值**，当某个请求查询的数据不存在时，将空对象或者null放入缓存，以防止相同的请求频繁查询数据库。
- **限流**，限流机制则可以限制数据库的访问频率，以减轻数据库压力。

布隆过滤器简单原理

有一个哈希表，首先所有的元素的值都为0，有多个哈希函数，写入数据的时候，通过这几个哈希函数计算出多个下标，将这几个下标对应位置1。读取数据时可以先读取到对应位置的地方的值，只要有一个位置为0，就说明数据不在集合中，全为1就说明在集合中

优点:

- **节省内存**：相比于使用散列表或集合等数据结构，布隆过滤器占用的内存较少，因为它只需要维护位数组。
- **快速查询**：布隆过滤器的查询操作非常快速，通常只需要几个哈希函数的计算和位的检查。

缺点:

- **误判率**：布隆过滤器可能会产生误判，本来不存在的，误判未存在。多个数据修改了同一个下标的值

- **不支持删除**：由于布隆过滤器的位数组只能设置为1，不能删除元素。如果需要删除元素，需要重新构建布隆过滤器。

NoSQL