1.事务四大特性（ACID）原子性、一致性、隔离性、持久性？

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，因此事务的操作如果成功就必须要完全应用到数据库，如果操作失败则不能对数据库有任何影响。

一致性（Consistency）

事务开始前和结束后，数据库的完整性约束没有被破坏。比如 A 向 B 转账，不可能 A 扣了钱，B 却没收到。

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。

同一时间，只允许一个事务请求同一数据，不同的事务之间彼此没有任何干扰。比如 A 正在从一张银行卡中取钱，在 A 取钱的过程结束前，B 不能向这张卡转账。

关于事务的隔离性数据库提供了多种隔离级别，稍后会介绍到。 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，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

2.事务的并发？事务隔离级别，每个级别会引发什么问题，MySQL 默认是哪个级别？

从理论上来说，事务应该彼此完全隔离，以避免并发事务所导致的问题，然而，那样会对性能产生极大的影响，因为事务必须按顺序运行，**在实际开发中，为了提升性能，事务会以较低的隔离级别运行， 事务的隔离级别可以通过隔离事务属性指定。**

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，结果因此本事务先后两次读到的数据结果会不一致。

3、**幻读**：幻读解决了不重复读，保证了同一个事务里，查询的结果都是事务开始时的状态（一致性）。

例如：事务 T1 对一个表中所有的行的某个数据项做了从“1”修改为“2”的操作 这时事务 T2 又对这个表中插入了一行数据项，而这个数据项的数值还是为“1”并且提交给数据库。 而操作事务 T1 的用户如果再查看刚刚修改的数据，会发现还有跟没有修改一样，其实这行是从事务 T2 中添加的，就好像产生幻觉一样，这就是发生了幻读。

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。

事务的隔离级别

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

读未提交：另一个事务修改了数据，但尚未提交，而本事务中的 SELECT 会读到这些未被提交的数据**脏读**

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，结果因此本事务先后两次读到的数据结果会不一致。

可重复读：在同一个事务里，SELECT 的结果是事务开始时时间点的状态，因此，同样的 SELECT 操作读到的结果会是一致的。但是，会有**幻读**现象

串行化：最高的隔离级别，在这个隔离级别下，不会产生任何异常。并发的事务，就像事务是在一个个按照顺序执行一样

MySQL 默认的事务隔离级别为 repeatable-read

MySQL 支持 4 中事务隔离级别.

事务的隔离级别要得到底层数据库引擎的支持，而不是应用程序或者框架的支持.

Oracle 支持的 2 种事务隔离级别：READ_COMMITED , SERIALIZABLE

补充：

1. SQL 规范所规定的标准，不同的数据库具体的实现可能会有些差异
2. MySQL 中默认事务隔离级别是“可重复读”时并不会锁住读取到的行

- **事务隔离级别：未提交读时**，写数据只会锁住相应的行。
- **事务隔离级别为：可重复读时**，写数据会锁住整张表。
- **事务隔离级别为：串行化时**，读写数据都会锁住整张表。

隔离级别越高，越能保证数据的完整性和一致性，但是对并发性能的影响也越大，鱼和熊掌不可兼得啊。**对于多数应用程序，可以优先考虑把数据库系统的隔离级别设为 Read Committed，它能够避免脏读取，而且具有较好的并发性能**。尽管它会导致不可重复读、幻读这些并发问题，在可能出现这类问题的个别场合，可以由应用程序采用悲观锁或乐观锁来控制。

3.MySQL 常见的三种存储引擎（InnoDB、MyISAM、MEMORY）的区别？

MySQL 存储引擎中的 MyISAM 和 InnoDB 区别详解

blog.csdn.net/lc0817/arti…

MySQL 存储引擎之 MyISAM 和 Innodb 总结性梳理

www.cnblogs.com/kevingrace/…

MySQL 存储引擎 MyISAM 与 InnoDB 如何选择

MySQL 有多种存储引擎，每种存储引擎有各自的优缺点，可以择优选择使用：`MyISAM、InnoDB、MERGE、MEMORY(HEAP)、BDB(BerkeleyDB)、EXAMPLE、FEDERATED、ARCHIVE、CSV、BLACKHOLE`。

虽然 MySQL 里的存储引擎不只是 MyISAM 与 InnoDB 这两个，但常用的就是两个。

两种存储引擎的大致区别表现在：

- **InnoDB 支持事务，MyISAM 不支持**，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了。
- **MyISAM 适合查询以及插入为主的应用。**
- **InnoDB 适合频繁修改以及涉及到安全性较高的应用。**
- InnoDB 支持外键，MyISAM 不支持。
- **从 MySQL5.5.5 以后，InnoDB 是默认引擎。**
- InnoDB 不支持 FULLTEXT 类型的索引。
- **InnoDB 中不保存表的行数**，如 `select count(*) from table` 时，InnoDB 需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count(*)语句包含 where 条件时 MyISAM 也需要扫描整个表。
- 对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引。
- `DELETE FROM table` 时，**InnoDB 不会重新建立表，而是一行一行的 删除，效率非常慢。MyISAM 则会重建表。**
- InnoDB 支持行锁（某些情况下还是锁整表，如 `update table set a=1 where user like '%lee%'`。

关于 MySQL 数据库提供的两种存储引擎，MyISAM 与 InnoDB 选择使用：

- **INNODB 会支持一些关系数据库的高级功能，如事务功能和行级锁，MyISAM 不支持。**
- **MyISAM 的性能更优，占用的存储空间少**，所以，选择何种存储引擎，视具体应用而定。
- **如果你的应用程序一定要使用事务，毫无疑问你要选择 INNODB 引擎。**但要注意，INNODB 的行级锁是有条件的。在 where 条件没有使用主键时，照样会锁全表。比如 DELETE FROM mytable 这样的删除语句。
- **如果你的应用程序对查询性能要求较高，就要使用 MyISAM 了。MyISAM 索引和数据是分开的，而且其索引是压缩的，可以更好地利用内存。**所以它的**查询性能明显优于 INNODB**。压缩后的索引也能节约一些磁盘空间。**MyISAM 拥有全文索引的功能，这可以极大地优化 LIKE 查询的效率。**

有人说 MyISAM 只能用于小型应用，其实这只是一种偏见。如果数据量比较大，这是需要通过升级架构来解决，比如分表分库，而不是单纯地依赖存储引擎。

现在一般都是选用 innodb 了，主要是 MyISAM 的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM 对于读写密集型应用一般是不会去选用的。

MEMORY 存储引擎

MEMORY 是 MySQL 中一类特殊的存储引擎。它使用存储在内存中的内容来创建表，而且数据全部放在内存中。这些特性与前面的两个很不同。

每个基于 MEMORY 存储引擎的表实际对应一个磁盘文件。该文件的文件名与表名相同，类型为 frm 类型。该文件中只存储表的结构。而其数据文件，都是存储在内存中，这样有利于数据的快速处理，提高整个表的效率。值得注意的是，服务器需要有足够的内存来维持 MEMORY 存储引擎的表的使用。如果不需要了，可以释放内存，甚至删除不需要的表。

MEMORY 默认使用哈希索引。速度比使用 B 型树索引快。当然如果你想用 B 型树索引，可以在创建索引时指定。

注意，**MEMORY 用到的很少，因为它是把数据存到内存中，如果内存出现异常就会影响数据。如果重启或者关机，所有数据都会消失。**因此，基于 **MEMORY 的表的生命周期很短，一般是一次性的。**

4.MySQL 的 MyISAM 与 InnoDB 两种存储引擎在，事务、锁级别，各自的适用场景？

事务处理上方面

- **MyISAM：强调的是性能**，每次查询具有原子性,其执行数度比 InnoDB 类型更快，但是**不提供事务支持**。
- **InnoDB：提供事务支持事务，外部键等高级数据库功能**。 具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全(transaction-safe (ACID compliant))型表。

锁级别

- **MyISAM：只支持表级锁**，用户在操作 MyISAM 表时，select，update，delete，insert 语句都会给表自动加锁，如果加锁以后的表满足 insert 并发的情况下，可以在表的尾部插入新的数据。
- **InnoDB**：支持事务和行级锁，是 innodb 的最大特色。行锁大幅度提高了多用户并发操作的新能。但是 InnoDB 的行锁，只是在 WHERE 的主键是有效的，非主键的 WHERE 都会锁全表的。

5.查询语句不同元素（where、jion、limit、group by、having 等等）执行先后顺序？

1.查询中用到的关键词主要包含六个，并且他们的顺序依次为 `select--from--where--group by--having--order by`

其中 select 和 from 是必须的，其他关键词是可选的，这六个关键词的执行顺序与 sql 语句的书写顺序并不是一样的，而是按照下面的顺序来执行

- from:需要从哪个数据表检索数据

- where:过滤表中数据的条件
- group by:如何将上面过滤出的数据分组
- having:对上面已经分组的数据进行过滤的条件
- select:查看结果集中的哪个列，或列的计算结果
- order by :按照什么样的顺序来查看返回的数据

2.from 后面的表关联，是自右向左解析 而 where 条件的解析顺序是自下而上的。

也就是说，在写 SQL 文的时候，尽量把数据量小的表放在最右边来进行关联（用小表去匹配大表），而把能筛选出小量数据的条件放在 where 语句的最左边（用小表去匹配大表）

6.什么是临时表，临时表什么时候删除？

时表可以手动删除：

```
DROP TEMPORARY TABLE IF EXISTS temp_tb;
```

临时表只在当前连接可见，当关闭连接时，MySQL 会自动删除表并释放所有空间。因此在不同的连接中可以创建同名的临时表，并且操作属于本连接的临时表。

创建临时表的语法与创建表语法类似，不同之处是**增加关键字 TEMPORARY**，如：

```
CREATE TEMPORARY TABLE tmp_table (

    NAME VARCHAR (10) NOT NULL,

    time date NOT NULL

);



select * from tmp_table;
```

7.MySQL B+Tree 索引和 Hash 索引的区别？

- Hash 索引结构的特殊性，其检索效率非常高，索引的检索可以一次定位;
- B+树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的 IO 访问;

那为什么大家不都用 Hash 索引而还要使用 B+树索引呢？

Hash 索引

1. Hash 索引仅仅能满足"="，"IN"和"<=>"查询，不能使用范围查询,因为经过相应的 Hash 算法处理之后的 Hash 值的大小关系，并不能保证和 Hash 运算前完全一样；
2. Hash 索引无法被用来避免数据的排序操作，因为 Hash 值的大小关系并不一定和 Hash 运算前的键值完全一样；
3. Hash 索引不能利用部分索引键查询，对于组合索引，Hash 索引在计算 Hash 值的时候是组合索引键合并后再一起计算 Hash 值，而不是单独计算 Hash 值，所以通过组合索引的前面一个或几个索引键进行查询的时候，Hash 索引也无法被利用；
4. Hash 索引在任何时候都不能避免表扫描，由于不同索引键存在相同 Hash 值，所以即使取满足某个 Hash 键值的数据的记录条数，也无法从 Hash 索引中直接完成查询，还是要回表查询数据；
5. Hash 索引遇到大量 Hash 值相等的情况后性能并不一定就会比 B+树索引高。

B+Tree 索引

MySQL 中，只有 HEAP/MEMORY 引擎才显示支持 Hash 索引。

常用的 InnoDB 引擎中默认使用的是 B+树索引，它会实时监控表上索引的使用情况，如果认为建立哈希索引可以提高查询效率，则自动在内存中的“自适应哈希索引缓冲区”建立哈希索引（**在 InnoDB 中默认开启自适应哈希索引**），通过观察搜索模式，MySQL 会利用 index key 的前缀建立哈希索引，如果一个表几乎大部分都在缓冲池中，那么建立一个哈希索引能够加快等值查询。

B+树索引和哈希索引的明显区别是：

如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据；

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索；

同理，**哈希索引没办法利用索引完成排序**，以及 like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；

哈希索引也不支持多列联合索引的最左匹配规则；

B+树索引的关键字检索效率比较平均，不像 B 树那样波动幅度大，**在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题**。

在大多数场景下，都会有范围查询、排序、分组等查询特征，用 B+树索引就可以了。

8.sql 查询语句确定创建哪种类型的索引，如何优化查询

- **性能优化过程中，选择在哪个列上创建索引是最重要的步骤之一**，可以考虑使用索引的主要有两种类型的列：在 where 子句中出现的列，在 join 子句中出现的列。
- 考虑列中值的分布，索引的列的基数越大，索引的效果越好。
- 使用短索引，如果**对字符串列进行索引，应该指定一个前缀长度**，可节省大量索引空间，提升查询速度。
- **利用最左前缀**,顾名思义，就是最左优先，在多列索引，有体现:(ALTER TABLE people ADD INDEX lname_fname_age (lame,fname,age);)，所谓最左前缀原则就是先要看第一列，在第一列满足的条件下再看左边第二列，以此类推

- **不要过度建索引，只保持所需的索引**。每个额外的索引都要占用额外的磁盘空间，并**降低写操作的性能**。
- 在修改表的内容时，索引必须进行更新，有时可能需要重构，因此，**索引越多，所花的时间越长**。
- **MySQL** 只对一下操作符才使用索引：`<,<=,=,>,>=,between,in`
- 以及某些时候的 **like**(不以通配符%或_开头的情形)。

9.聚集索引和非聚集索引区别？

聚合索引(clustered index) / 非聚合索引(nonclustered index)

根本区别

聚集索引和非聚集索引的根本区别是**表记录的排列顺序和与索引的排列顺序是否一致**。

聚集索引

聚集索引表记录的排列顺序和索引的排列顺序一致，所以查询效率快，只要找到第一个索引值记录，其余就连续性的记录在物理也一样连续存放。聚集索引对应的缺点就是修改慢，因为为了保证表中记录的物理和索引顺序一致，在记录插入的时候，会对数据页重新排序。

聚集索引类似于新华字典中用拼音去查找汉字，拼音检索表于书记顺序都是按照 a~z 排列的，就像相同的逻辑顺序于物理顺序一样，当你需要查找 a,ai 两个读音的字，或是想一次寻找多个傻(sha)的同音字时，也许向后翻几页，或紧接着下一行就得到结果了。

非聚集索引

非聚集索引制定了表中记录的逻辑顺序，但是记录的物理和索引不一定一致，两种索引都采用 B+树结构，非聚集索引的叶子层并不和实际数据页相重叠，而采用叶子层包含一个指向表中的记录在数据页中的指针方式。非聚集索引层次多，不会造成数据重排。

非聚集索引类似在新华字典上通过偏旁部首来查询汉字，检索表也许是按照横、竖、撇来排列的，但是由于正文中是 a~z 的拼音顺序，所以就类似于逻辑地址于物理地址的不对应。同时适用的情况就在于分组，大数目的不同值，频繁更新的列中，这些情况即不适合聚集索引。

10.有哪些锁（乐观锁悲观锁），select 时怎么加排它锁？

悲观锁（Pessimistic Lock）

悲观锁的特点是先获取锁，再进行业务操作，即“悲观”的认为获取锁是非常有可能失败的，因此要先确保获取锁成功再进行业务操作。**通常所说的“一锁二查三更新”即指的是使用悲观锁。通常来讲在数据库上的悲观锁需要数据库本身提供支持，即通过常用的 select … for update 操作来实现悲观锁**。当数据库执行 select for update 时会获取被 select 中的数据行的行锁，因此其他并发执行的 select for update 如果试图选中同一行则会发生排斥（需要等待行锁被释放），因此达到锁的效果。select for update 获取的行锁会在当前事务结束时自动释放，因此必须在事务中使用。

这里需要注意的一点是不同的数据库对 select for update 的实现和支持都是有所区别的，例如 oracle 支持 select for update no wait，表示如果拿不到锁立刻报错，而不是等待，MySQL 就没有 no wait 这个选项。另外 **MySQL 还有个问题是 select for update 语句执行中所有扫描过的行都会被锁上，这一点很容易造成问题。因此如果在 MySQL 中用悲观锁务必要确定走了索引，而不是全表扫描。**

乐观锁（Optimistic Lock）

乐观锁，也叫乐观并发控制，它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，那么当前正在提交的事务会进行回滚。

乐观锁的特点先进行业务操作，不到万不得已不去拿锁。即“乐观”的认为拿锁多半是会成功的，因此在进行完业务操作需要实际更新数据的最后一步再去拿一下锁就好。

乐观锁在数据库上的实现完全是逻辑的，不需要数据库提供特殊的支持。一般的做法是在需要锁的数据上增加一个版本号，或者时间戳，然后按照如下方式实现：

乐观锁（给表加一个版本号字段） 这个并不是乐观锁的定义，给表加版本号，是数据库实现乐观锁的一种方式。

```
1\. SELECT data AS old_data, version AS old_version FROM …;2\. 根据获取的数据进行业务操作，得到 new_data 和 new_version3\. UPDATE SET data = new_data, version = new_version WHERE version = old_versionif (updated row > 0) {

    // 乐观锁获取成功，操作完成

} else {

    // 乐观锁获取失败，回滚并重试

}
```

乐观锁在不发生取锁失败的情况下开销比悲观锁小，但是一旦发生失败回滚开销则比较大，因此适合用在取锁失败概率比较小的场景，可以提升系统并发性能

乐观锁还适用于一些比较特殊的场景，例如在业务操作过程中无法和数据库保持连接等悲观锁无法适用的地方。

总结

悲观锁和乐观锁是数据库用来保证数据并发安全防止更新丢失的两种方法，例子在 `select ... for update` 前加个事务就可以防止更新丢失。悲观锁和乐观锁大部分场景下差异不大，一些独特场景下有一些差别，一般我们可以从如下几个方面来判断。

- **响应速度**：如果需要非常高的响应速度，**建议采用乐观锁方案**，成功就执行，不成功就失败，不需要等待其他并发去释放锁。
-
- **冲突频率**：如果冲突频率非常高，**建议采用悲观锁**，保证成功率，如果冲突频率大，乐观锁会需要多次重试才能成功，代价比较大。
-

-

  重试代价：如果重试代价大，建议采用悲观锁。

-

11.非关系型数据库和关系型数据库区别，优势比较？

非关系型数据库的优势：

1. 性能

NOSQL 是基于键值对的，可以想象成表中的主键和值的对应关系，而且不需要经过 SQL 层的解析，所以性能非常高。

2. 可扩展性

同样也是因为基于键值对，数据之间没有耦合性，所以非常容易水平扩展。

关系型数据库的优势：

1. 复杂查询

可以用 SQL 语句方便的在一个表以及多个表之间做非常复杂的数据查询。

2. 事务支持

使得对于安全性能很高的数据访问要求得以实现。

总结

对于这两类数据库，对方的优势就是自己的弱势，反之亦然。

NOSQL 数据库慢慢开始具备 SQL 数据库的一些复杂查询功能，比如 MongoDB。

对于事务的支持也可以用一些系统级的原子操作来实现例如乐观锁之类的方法来曲线救国，比如 Redis set nx。

12.数据库三范式，根据某个场景设计数据表？

- 所有字段值都是不可分解的原子值。
- 在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。
- 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。

第一范式(确保每列保持原子性)

第一范式是最基本的范式。如果数据库表中的所有字段值都是不可分解的原子值，就说明该数据库表满足了第一范式。

第一范式的合理遵循需要根据系统的实际需求来定。比如某些数据库系统中需要用到“地址”这个属性，本来直接将“地址”属性设计成一个数据库表的字段就行。但是如果系统经常会访问“地址”属性中的“城市”部分，那么就非要将“地址”这个属性重新拆分为省份、城市、详细地址等多个部分进行存储，这样在对地址中某一部分操作的时候将非常方便。这样设计才算满足了数据库的第一范式，如下表所示。

上表所示的用户信息遵循了第一范式的要求，这样在对用户使用城市进行分类的时候就非常方便，也提高了数据库的性能。

第二范式(确保表中的每列都和主键相关)

第二范式在第一范式的基础之上更进一层。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）。**也就是说在一个数据库表中，一个表中只能保存一种数据，不可以把多种数据保存在同一张数据库表中。**

比如要设计一个订单信息表，因为订单中可能会有多种商品，所以要将订单编号和商品编号作为数据库表的联合主键。

第三范式(确保每列都和主键列直接相关,而不是间接相关)

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。**

比如在设计一个订单数据表的时候，可以将客户编号作为一个外键和订单表建立相应的关系。而不可以在订单表中添加关于客户其它信息（比如姓名、所属公司等）的字段。

13.数据库的读写分离、主从复制，主从复制分析的 7 个问题？

主从复制的几种方式

同步复制

- 所谓的同步复制，意思是 master 的变化，必须等待 slave-1,slave-2,...,slave-n 完成后才能返回。 这样，显然不可取，也不是 MySQL 复制的默认设置。比如，在 WEB 前端页面上，用户增加了条记录，需要等待很长时间。

异步复制

- 如同 AJAX 请求一样。master 只需要完成自己的数据库操作即可。至于 slaves 是否收到二进制日志，是否完成操作，不用关心,MySQL 的默认设置。

半同步复制

- master 只保证 slaves 中的一个操作成功，就返回，其他 slave 不管。 这个功能，是由 google 为 MySQL 引入的。

主从复制分析的 7 个问题

问题 1：master 的写操作，slaves 被动的进行一样的操作，保持数据一致性，那么 slave 是否可以主动的进行写操作？

假设 slave 可以主动的进行写操作，slave 又无法通知 master，这样就导致了 master 和 slave 数据不一致了。因此 **slave 不应该进行写操作**，至少是 slave 上涉及到复制的数据库不可以写。实际上，**这里已经揭示了读写分离的概念**。

问题 2：主从复制中，可以有 N 个 slave,可是这些 slave 又不能进行写操作，要他们干嘛？

以实现数据备份。

类似于高可用的功能，一旦 master 挂了，可以让 slave 顶上去，同时 slave 提升为 master。

异地容灾，比如 master 在北京，地震挂了，那么在上海的 slave 还可以继续。

主要用于实现 scale out,**分担负载,可以将读的任务分散到 slaves 上。**

【很可能的情况是，一个系统的读操作远远多于写操作，因此写操作发向 master，读操作发向 slaves 进行操作】

问题 3：主从复制中有 master,slave1,slave2,...等等这么多 MySQL 数据库，那比如一个 JAVA WEB 应用到底应该连接哪个数据库?

当 然，我们在应用程序中可以这样，`insert/delete/update` 这些更新数据库的操作，用 `connection(for master)`进行操作，`select` 用 `connection(for slaves)`进行操作。那我们的应用程序还要完成怎么从 slaves 选择一个来执行 select，例如**使用简单的轮循算法。**

这样的话，相当于应用程序完成了 SQL 语句的路由，而且与 MySQL 的主从复制架构非常关联，一旦 master 挂了，某些 slave 挂了，那么应用程序就要修改了。能不能让应用程序与 MySQL 的主从复制架构没有什么太多关系呢？

找一个组件，application program 只需要与它打交道，用它来完成 MySQL 的代理，实现 SQL 语句的路由。

MySQL proxy 并不负责，怎么从众多的 slaves 挑一个？可以交给另一个组件(比如 haproxy)来完成。

这就是所谓的 `MySQL READ WRITE SPLITE，MySQL` 的读写分离。

问题 4：如果 MySQL proxy , direct , master 他们中的某些挂了怎么办？

总统一般都会弄个副总统，以防不测。同样的，可以给这些关键的节点来个备份。

问题 5：当 master 的二进制日志每产生一个事件，都需要发往 slave，如果我们有 N 个 slave,那是发 N 次，还是只发一次？

如果只发一次，发给了 slave-1，那 slave-2,slave-3,...它们怎么办？

显 然，应该发 N 次。实际上，**在 MySQL master 内部，维护 N 个线程，每一个线程负责将二进制日志文件发往对应的 slave**。master 既要负责写操作，还的维护 N 个线程，负担会很重。**可以这样，slave-1 是 master 的从，slave-1 又是 slave-2,slave-3,...的主**，同时 slave-1 不再负责 select。**slave-1 将 master 的复制线程的负担，转移到自己的身上。这就是所谓的多级复制的概念。**

问题 6：当一个 select 发往 MySQL proxy，可能这次由 slave-2 响应，下次由 slave-3 响应，这样的话，就无法利用查询缓存了。

应该找一个共享式的缓存，比如 memcache 来解决。将 slave-2,slave-3,...这些查询的结果都缓存至 mamcache 中。

问题 7：随着应用的日益增长，读操作很多，我们可以扩展 slave，但是如果 master 满足不了写操作了，怎么办呢？

scale on ?更好的服务器？ 没有最好的，只有更好的，太贵了。。。

scale out？ 主从复制架构已经满足不了。

可以分库【垂直拆分】，分表【水平拆分】。

14.使用 explain 优化 sql 和索引？

对于复杂、效率低的 sql 语句，我们通常是使用 explain sql 来分析 sql 语句，这个语句可以打印出，语句的执行。这样方便我们分析，进行优化

- **table**：显示这一行的数据是关于哪张表的
- **type**：这是重要的列，显示连接使用了何种类型。从最好到最差的连接类型为 `const、eq_reg、ref、range、index` 和 `ALL`
- **all**: full table scan ;MySQL 将遍历全表以找到匹配的行；
- **index**： index scan; index 和 all 的区别在于 index 类型只遍历索引；
- **range**：索引范围扫描，对索引的扫描开始于某一点，返回匹配值的行，常见与 `between ，< ,>`等查询；
- **ref**：非唯一性索引扫描，返回匹配某个单独值的所有行，常见于使用非唯一索引即唯一索引的非唯一前缀进行查找；
- **eq_ref**：唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配，常用于主键或者唯一索引扫描；

- **const**，system：当 MySQL 对某查询某部分进行优化，并转为一个常量时，使用这些访问类型。如果将主键置于 where 列表中，MySQL 就能将该查询转化为一个常量。
- **possible_keys**：显示可能应用在这张表中的索引。如果为空，没有可能的索引。可以为相关的域从 WHERE 语句中选择一个合适的语句
- **key**：**实际使用的索引。如果为 NULL，则没有使用索引**。很少的情况下，MySQL 会选择优化不足的索引。这种情况下，可以在 SELECT 语句中使用 USE INDEX（indexname）来强制使用一个索引或者用 IGNORE INDEX（indexname）来强制 MySQL 忽略索引
- **key_len**：**使用的索引的长度。在不损失精确性的情况下，长度越短越好**
- **ref**：显示索引的哪一列被使用了，如果可能的话，是一个常数
- **rows**：MySQL 认为必须检查的用来返回请求数据的行数
- **Extra**：关于 MySQL 如何解析查询的额外信息。将在表 4.3 中讨论，但这里可以看到的坏的例子是 Using temporary 和 Using filesort，意思 MySQL 根本不能使用索引，结果是检索会很慢。

15.MySQL 慢查询怎么解决？

- **slow_query_log** 慢查询开启状态。
- **slow_query_log_file** 慢查询日志存放的位置（这个目录需要 MySQL 的运行帐号的可写权限，一般设置为 MySQL 的数据存放目录）。
- **long_query_time** 查询超过多少秒才记录。

16.什么是 内连接、外连接、交叉连接、笛卡尔积等？

内连接

内连接查询操作列出与连接条件匹配的数据行，它使用比较运算符比较被连接列的 列值。

内连接分三种：

1.等值连接：在连接条件中使用等于号(=)运算符比较被连接列的列值，其查询结果中列出被连接表中的所有列，包括其中的重复列。

例，下面使用等值连接列出 authors 和 publishers 表中位于同一城市的作者和出版社：

```
SELECT * FROM authors AS a INNER JOIN publishers AS p ON a.city=p.city
```

2.不等连接： 在连接条件使用除等于运算符以外的其它比较运算符比较被连接的 列的列值。这些运算符包括>、>=、<=、<、!>、!<和<>。

3.自然连接：在连接条件中使用等于(=)运算符比较被连接列的列值，但它使用选择列表指出查询结果集合中所包括的列，并删除连接表中的重复列。

例，在选择列表中删除 authors 和 publishers 表中重复列(city 和 state)：

```
SELECT a.*,p.pub_id,p.pub_name,p.country FROM authors AS a INNER JOIN publishers
AS p ON a.city=p.city
```

外连接

外连接，返回到查询结果集合中的不仅包含符合连接条件的行，**而且还包括左表(左外连接或左连接)**、右表(**右外连接或右连接**)或两个边接表(全外连接)中的所有数据行。

- **left join**(左联接) 返回包括左表中的所有记录和右表中联结字段相等的记录。
- **right join**(右联接) 返回包括右表中的所有记录和左表中联结字段相等的记录。

例如 1：

```
SELECT a.*,b.* FROM luntan LEFT JOIN usertable as b ON a.username=b.username
```

例如 2：

```
SELECT a.*,b.* FROM city as a FULL OUTER JOIN user as b ON a.username=b.username
```

交叉连接

交叉连接不带 WHERE 子句，它返回被连接的两个表所有数据行的“笛卡尔积”，返回到结果集合中的数据行数等于第一个表中符合查询条件的数据行数乘以第二个表中符合查询条件的数据行数。

例，titles 表中有 6 类图书，而 publishers 表中有 8 家出版社，则下 列交叉连接检索到的记录数将等于 6*8=48 行。

例如：

```
SELECT type,pub_name FROM titles CROSS JOIN publishers  ORDER BY type
```

笛卡尔积

笛卡尔积是两个表每一个字段相互匹配，去掉 `where` 或者 `inner join` 的等值 得出的结果就是笛卡尔积。**笛卡尔积也等同于交叉连接**。

总结

- 内连接: 只连接匹配的行。
- 左外连接: 包含左边表的全部行（不管右边的表中是否存在与它们匹配的行），以及右边表中全部匹配的行。
- 右外连接: 包含右边表的全部行（不管左边的表中是否存在与它们匹配的行），以及左边表中全部匹配的行。
- 全外连接: 包含左、右两个表的全部行，不管另外一边的表中是否存在与它们匹配的行。
- 交叉连接 生成笛卡尔积—它不使用任何匹配或者选取条件，而是直接将一个数据源中的每个行与另一个数据源的每个行都一一匹配。

17.MySQL 都有什么锁，死锁判定原理和具体场景，死锁怎么解决？

MySQL 都有什么锁

MySQL 有三种锁的级别：**页级、表级、行级**。

- **表级锁**：开销小，加锁快；**不会出现死锁**；锁定粒度大，发生锁冲突的概率最高,并发度最低。
- **行级锁**：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。
- **页面锁**：开销和加锁时间界于表锁和行锁之间；**会出现死锁**；锁定粒度界于表锁和行锁之间，并发度一般

什么情况下会造成死锁

- 所谓死锁: 是指两个或两个以上的进程在执行过程中。
- 因争夺资源而造成的一种互相等待的现象,若无外力作用,它们都将无法推进下去。
- 此时称系统处于死锁状态或系统产生了死锁,这些永远在互相等竺的进程称为死锁进程。
- 表级锁不会产生死锁.所以解决死锁主要还是针对于最常用的 InnoDB。

死锁的关键在于：两个(或以上)的 Session 加锁的顺序不一致。

那么对应的**解决死锁问题的关键就是**：让不同的 session 加锁有次序。

死锁的解决办法

- 查出的线程杀死 kill

```
SELECT trx_MySQL_thread_id FROM information_schema.INNODB_TRX;
```

- 设置锁的超时时间

Innodb 行锁的等待时间，单位秒。可在会话级别设置，RDS 实例该参数的默认值为 50（秒）。

生产环境不推荐使用过大的 `innodb_lock_wait_timeout` 参数值

该参数支持在会话级别修改，方便应用在会话级别单独设置某些特殊操作的行锁等待超时时间，如下：

```
set innodb_lock_wait_timeout=1000; —设置当前会话 Innodb 行锁等待超时时间，单位秒。
```

18.varchar 和 char 的使用场景？

char 的长度是不可变的，而 varchar 的长度是可变的。

定义一个 char[10]和 varchar[10]。

如果存进去的是‘csdn’,那么 char 所占的长度依然为 10，除了字符‘csdn’外，后面跟六个空格，varchar 就立马把长度变为 4 了，取数据的时候，char 类型的要用 trim()去掉多余的空格，而 varchar 是不需要的。

char 的存取数度还是要比 varchar 要快得多，因为其长度固定，方便程序的存储与查找。

char 也为此付出的是空间的代价，因为其长度固定，所以难免会有多余的空格占位符占据空间，**可谓是以空间换取时间效**率。

varchar 是**以空间效率为首位**。

char 的存储方式是：对英文字符（ASCII）占用 1 个字节，对一个汉字占用两个字节。

varchar 的存储方式是：对每个英文字符占用 2 个字节，汉字也占用 2 个字节。

两者的存储数据都非 unicode 的字符数据。

19.MySQL 高并发环境解决方案？

MySQL 高并发环境解决方案 分库 分表 分布式 增加二级缓存。。。。。

需求分析：互联网单位 每天大量数据读取，写入，并发性高。

- **现有解决方式**：水平分库分表，由单点分布到多点数据库中，从而降低单点数据库压力。
- **集群方案**：解决 DB 宕机带来的单点 DB 不能访问问题。
- **读写分离策略**：极大限度提高了应用中 Read 数据的速度和并发量。无法解决高写入压力。

20.数据库崩溃时事务的恢复机制（REDO 日志和 UNDO 日志）？

[转]MySQL REDO 日志和 UNDO 日志

www.cnblogs.com/Bozh/archiv…

Undo Log

Undo Log 是为了实现事务的原子性，在 MySQL 数据库 InnoDB 存储引擎中，还用了 Undo Log 来实现多版本并发控制(简称：MVCC)。

事务的原子性(Atomicity)事务中的所有操作，要么全部完成，要么不做任何操作，不能只做部分操作。如果在执行的过程中发生了错误，要回滚(Rollback)到事务开始前的状态，就像这个事务从来没有执行过。

原理 Undo Log 的原理很简单，**为了满足事务的原子性，在操作任何数据之前，首先将数据备份到一个地方（这个存储数据备份的地方称为 UndoLog）**。然后进行数据的修改。如果出现了错误或者用户执行了 ROLLBACK 语句，**系统可以利用 Undo Log 中的备份将数据恢复到事务开始之前的状态**。

-

之所以能同时保证原子性和持久化，是因为以下特点：

- 更新数据前记录 Undo log。
- 为了保证持久性，必须将数据在事务提交前写到磁盘。只要事务成功提交，数据必然已经持久化。
- **Undo log 必须先于数据持久化到磁盘**。如果在 G,H 之间系统崩溃，**undo log 是完整的， 可以用来回滚事务**。
- 如果在 A-F 之间系统崩溃,因为数据没有持久化到磁盘。所以磁盘上的数据还是保持在事务开始前的状态。

缺陷：每个事务提交前将数据和 Undo Log 写入磁盘，这样会导致大量的磁盘 IO，因此性能很低。

如果能够将数据缓存一段时间，就能减少 IO 提高性能。但是这样就会丧失事务的持久性。因此引入了另外一种机制来实现持久化，即 Redo Log。

Redo Log

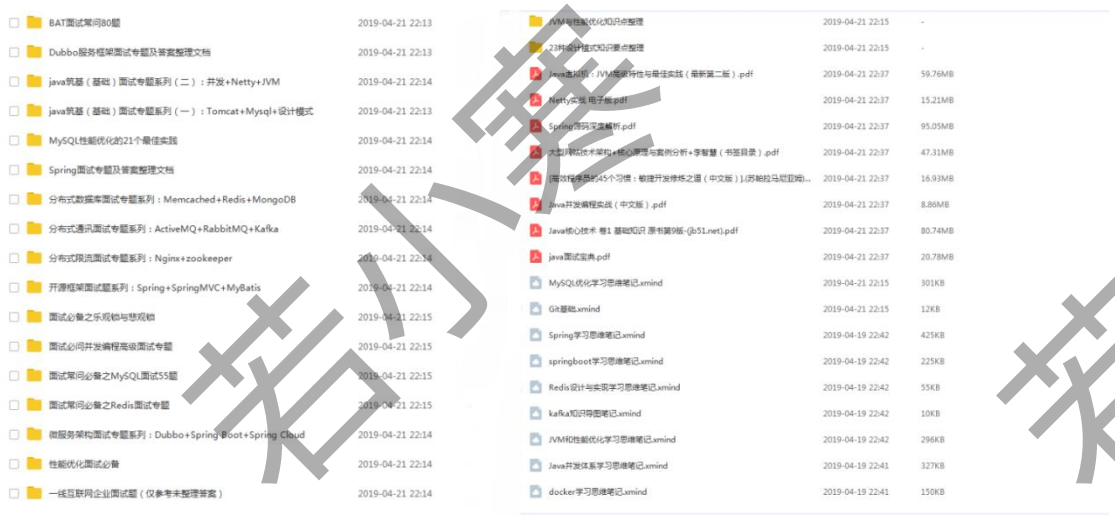
- 原理和 **Undo Log** 相反，**Redo Log 记录的是新数据的备份。在事务提交前，只要将 Redo Log 持久化即可，不需要将数据持久化。当系统崩溃时，虽然数据没有持久化，但是 Redo Log 已经持久化。系统可以根据 Redo Log 的内容，将所有数据恢复到最新的状态**。

读者福利

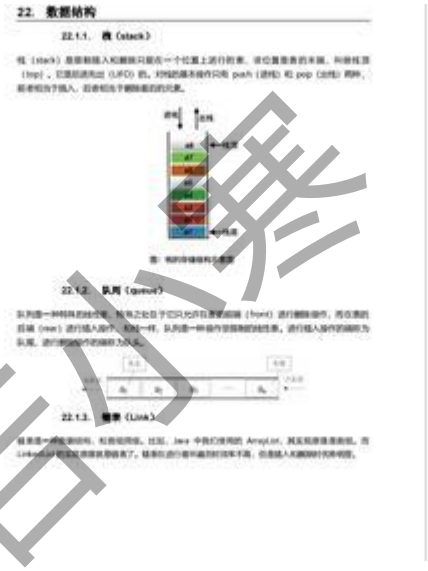
针对于还会准备免费的 Java 架构学习资料（里面有高可用、高并发、高性能及分布式、Jvm 性能调优、MyBatis，Netty,Redis,Kafka,Mysql,Zookeeper,Tomcat,Docker,Dubbo,Nginx 等多个知识点的架构资料）

为什么某些人会一直比你优秀，是因为他本身就很优秀还一直在持续努力变得更优秀，而你是不是还在满足于现状内心在窃喜！**希望读到这的您能转发和关注下，以后还会更新技术干货，谢谢您的支持！**

资料免费领取方式：加 QQ 交流群：7087014573 免费获取！


BAT面试常问80题 2019-04-21 22:13
Dubbo服务框架面试专题及答案整理文档 2019-04-21 22:13
java筑基（基础）面试专题系列（二）：并发+Netty+JVM 2019-04-21 22:14
java筑基（基础）面试专题系列（一）：Tomcat+Mysql+设计模式 2019-04-21 22:13
MySQL性能优化的21个最佳实践 2019-04-21 22:14
Spring面试专题及答案整理文档 2019-04-21 22:14
分布式数据库面试专题系列：Memcached+Redis+MongoDB 2019-04-21 22:14
分布式通讯面试专题系列：ActiveMQ+RabbitMQ+Kafka 2019-04-21 22:14
分布式限流面试专题系列：Nginx+zookeeper 2019-04-21 22:14
开源框架面试题系列：Spring+SpringMVC+MyBatis 2019-04-21 22:14
面试必备之乐观锁与悲观锁 2019-04-21 22:15
面试必问并发编程高级面试专题 2019-04-21 22:15
面试常问必备之MySQL面试55题 2019-04-21 22:15
面试常问必备之Redis面试专题 2019-04-21 22:15
微服务架构面试专题系列：Dubbo+Spring Boot+Spring Cloud 2019-04-21 22:14
性能优化面试必备 2019-04-21 22:14
一线互联网企业面试题（仅参考未整理答案） 2019-04-21 22:14
JVM和性能优化知识点整理 2019-04-21 22:15 -
23种设计模式知识要点整理 2019-04-21 22:15 -
Java虚拟机：JVM高级特性与最佳实践（最新第二版）.pdf 2019-04-21 22:37 59.76MB
Netty实战 电子版.pdf 2019-04-21 22:37 15.21MB
Spring源码深度解析.pdf 2019-04-21 22:37 95.05MB
大型网站技术架构+核心原理与案例分析+李智慧（书签目录）.pdf 2019-04-21 22:37 47.31MB
[高效程序员的45个习惯：敏捷开发修炼之道（中文版）].(苏帕拉马尼亚姆)... 2019-04-21 22:37 16.93MB
Java开发编程实战（中文版）.pdf 2019-04-21 22:37 8.86MB
Java核心技术 卷1 基础知识 原书第9版-(jb51.net).pdf 2019-04-21 22:37 80.74MB
java面试宝典.pdf 2019-04-21 22:37 20.78MB
MySQL优化学习思维笔记.xmind 2019-04-21 22:15 301KB
Git基础.xmind 2019-04-21 22:15 12KB
Spring学习思维笔记.xmind 2019-04-19 22:42 425KB
springboot学习思维笔记.xmind 2019-04-19 22:42 225KB
Redis设计与实现学习思维笔记.xmind 2019-04-19 22:42 55KB
kafka知识导图笔记.xmind 2019-04-19 22:42 10KB
JVM和性能优化学习思维笔记.xmind 2019-04-19 22:42 296KB
Java并发体系学习思维笔记.xmind 2019-04-19 22:41 327KB
docker学习思维笔记.xmind 2019-04-19 22:41 150KB


2. JVM
3. JAVA集合
4. JAVA多线程并发
5. JAVA基础
6. Spring 原理
7. 微服务
8. Netty NIO/RPC
9. 网络
10. 日志
11. Zookeeper
12. Kafka
13. RabbitMQ
14. Hbase
15. MongoDB
16. Cassandra
17. 设计模式
18. 负载均衡
19. 数据库
20. 一致性算法
21. JAVA算法
22. 数据结构
23. 加密算法
24. 分布式缓存
25. Hadoop
26. Spark


22. 数据结构
22.1.1. 栈（stack）
22.1.2. 队列（queue）
22.1.3. 链表（Link）


22.1.4. 散列表（Hash Table）
22.1.5. 排序二叉树

更多笔记分享

Dubbo服务框架知识点PDF文档整理

Java架构进阶150道面试题PDF文档整理

Java架构师面试技术点整理及学习笔记

Java面试高频试题汇总（整理答案适合1-5年开发）

Java面试知识点体系PDF文档整理

Spring全家桶知识点PDF文档整理

JVM与性能优化知识点整理

Redis学习笔记


Spring AOP源码讲解


大型网站数据瓶颈之数据库分库分表方案实践


大战618 决胜双十一高并发秒杀系统解密


互联网四大法宝之缓存与并发编程


面试必问Redis进阶问题讲解


面试必问—zk分布式锁


面试必问-服务器推送技术


面试必问-深入理解JVM


轻松搞定AOP面试从Spring热插件实战开始


手写SpringMVC框架