图文详解 66 道 MySQL 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。

基础

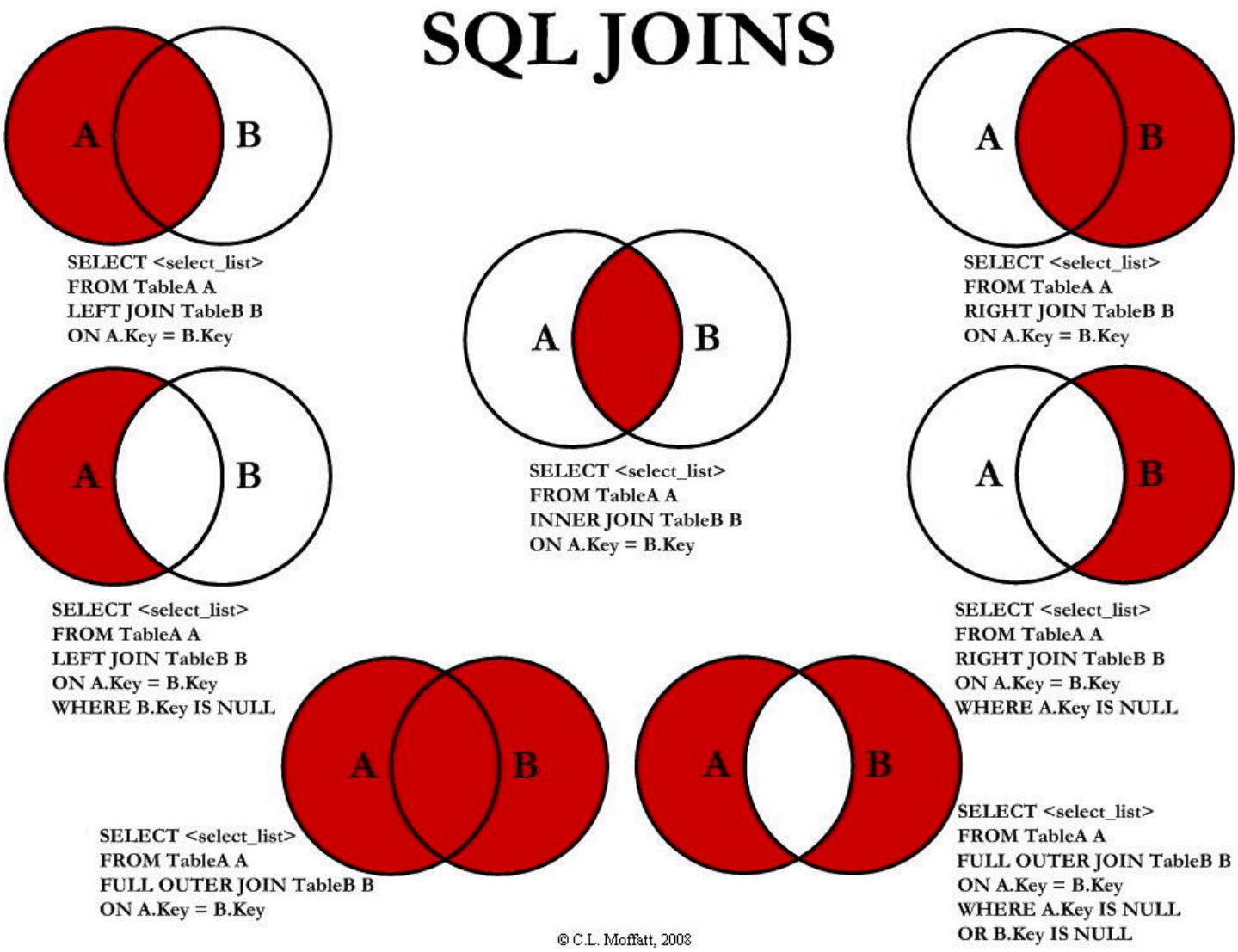
作为 SQL Boy，基础部分不会有人不会吧？面试也不怎么问，基础掌握不错的小伙伴可以跳过这一部分。当然，可能会现场写一些 SQL 语句，SQ 语句可以通过牛客、LeetCode、LintCode 之类的网站来练习。

1. 什么是内连接、外连接、交叉连接、笛卡尔积呢？

- 内连接（inner join）：取得两张表中满足存在连接匹配关系的记录。
- 外连接（outer join）：不只取得两张表中满足存在连接匹配关系的记录，还包括某张表（或两张表）中不满足匹配关系的记录。
- 交叉连接（cross join）：显示两张表所有记录一一对应，没有匹配关系进行筛选，它是笛卡尔积在 SQL 中的实现，如果 A 表有 m 行，B 表有 n 行，那么 A 和 B 交叉连接的结果就有 m*n 行。
- 笛卡尔积：是数学中的一个概念，例如集合 A={a,b}，集合 B={1,2,3}，那么 A ✖ B=`{<a,o>,<a,1>,<a,2>,<b,0>,<b,1>,<b,2>,}`。

2. 那 MySQL 的内连接、左连接、右连接有有什么区别？

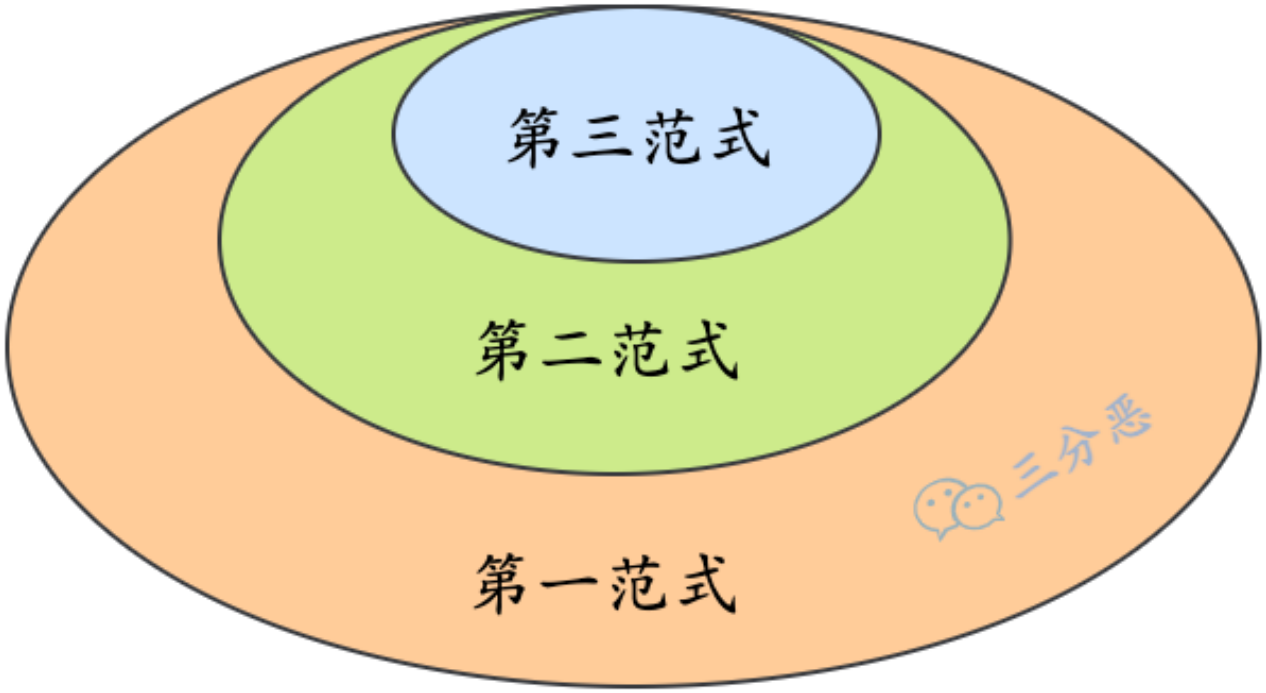
MySQL 的连接主要分为内连接和外连接，外连接常用的有左连接、右连接。



MySQL-joins-来源菜鸟教程

- inner join 内连接，在两张表进行连接查询时，只保留两张表中完全匹配的结果集
- left join 在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
- right join 在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

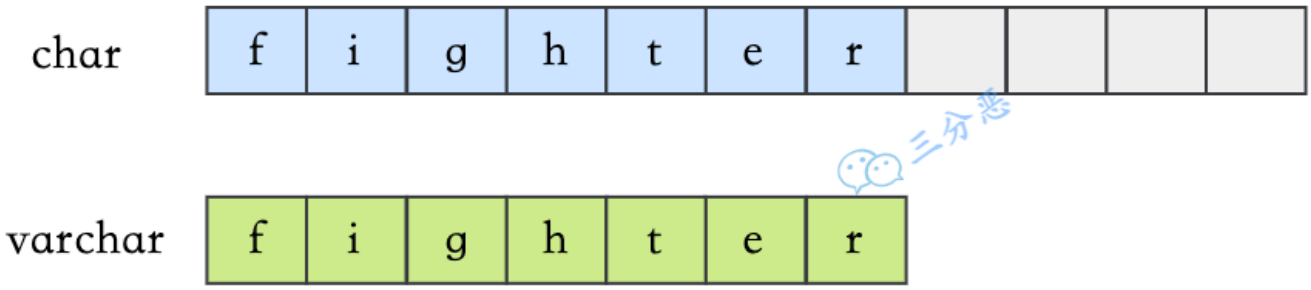
3.说一下数据库的三大范式?



- 第一范式：数据表中的每一列（每个字段）都不可以再拆分。例如用户表，用户地址还可以拆分成国家、省份、市，这样才是符合第一范式的。
- 第二范式：在第一范式的基础上，非主键列完全依赖于主键，而不能是依赖于主键的一部分。例如订单表里，存储了商品信息（商品价格、商品类型），那就需要把商品 ID 和订单 ID 作为联合主键，才满足第二范式。
- 第三范式：在满足第二范式的基础上，表中的非主键只依赖于主键，而不依赖于其他非主键。例如订单表，就不能存储用户信息（姓名、地址）。

三大范式的作用是为了控制数据库的冗余，是对空间的节省，实际上，一般互联网公司的设计都是反范式的，通过冗余一些数据，避免跨表跨库，利用空间换时间，提高性能。

4.varchar 与 char 的区别?



char:

- char 表示定长字符串，长度是固定的;
- 如果插入数据的长度小于 char 的固定长度时，则用空格填充;
- 因为长度固定，所以存取速度要比 varchar 快很多，甚至能快 50%，但正因为其长度固定，所以会占据多余的空间，是空间换时间的做法;
- 对于 char 来说，最多能存放的字符个数为 255，和编码无关

varchar:

- varchar 表示可变长字符串，长度是可变的;

- 插入的数据是多长，就按照多长来存储；
- varchar 在存取方面与 char 相反，它存取慢，因为长度不固定，但正因如此，不占据多余的空间，是时间换空间的做法；
- 对于 varchar 来说，最多能存放的字符个数为 65532

日常的设计，对于长度相对固定的字符串，可以使用 char，对于长度不确定的，使用 varchar 更合适一些。

5.blob 和 text 有什么区别?

- blob 用于存储二进制数据，而 text 用于存储大字符串。
- blob 没有字符集，text 有一个字符集，并且根据字符集的校对规则对值进行排序和比较

6.DATETIME 和 TIMESTAMP 的异同?

相同点:

1. 两个数据类型存储时间的表现格式一致。均为 `YYYY-MM-DD HH:MM:SS`
2. 两个数据类型都包含「日期」和「时间」部分。
3. 两个数据类型都可以存储微秒的小数秒（秒后 6 位小数秒）

区别:

DATETIME 和 TIMESTAMP 的区别

1. 日期范围：DATETIME 的日期范围是 `1000-01-01 00:00:00.000000` 到 `9999-12-31 23:59:59.999999`；TIMESTAMP 的时间范围是`1970-01-01 00:00:01.000000` UTC `到 ``2038-01-09 03:14:07.999999` UTC
2. 存储空间：DATETIME 的存储空间为 8 字节；TIMESTAMP 的存储空间为 4 字节
3. 时区相关：DATETIME 存储时间与时区无关；TIMESTAMP 存储时间与时区有关，显示的值也依赖于时区
4. 默认值：DATETIME 的默认值为 null；TIMESTAMP 的字段默认不为空(not null)，默认值为当前时间(CURRENT_TIMESTAMP)

7.MySQL 中 in 和 exists 的区别？

MySQL 中的 in 语句是把外表和内表作 hash 连接，而 exists 语句是对外表作 loop 循环，每次 loop 循环再对内表进行查询。我们可能认为 exists 比 in 语句的效率要高，这种说法其实是不准确的，要区分情景：

1. 如果查询的两个表大小相当，那么用 in 和 exists 差别不大。
2. 如果两个表中一个较小，一个是大表，则子查询表大的用 exists，子查询表小的用 in。

3. not in 和 not exists：如果查询语句使用了 not in，那么内外表都进行全表扫描，没有用到索引；而 not extsts 的子查询依然能用到表上的索引。所以无论那个表大，用 not exists 都比 not in 要快。

8.MySQL 里记录货币用什么字段类型比较好?

货币在数据库中 MySQL 常用 Decimal 和 Numric 类型表示，这两种类型被 MySQL 实现为同样的类型。他们被用于保存与货币有关的数据。

例如 salary DECIMAL(9,2)，9(precision)代表将被用于存储值的总的小数位数，而 2(scale)代表将被用于存储小数点后的位数。存储在 salary 列中的值的范围是从-9999999.99 到 9999999.99。

DECIMAL 和 NUMERIC 值作为字符串存储，而不是作为二进制浮点数，以便保存那些值的小数精度。

之所以不使用 float 或者 double 的原因：因为 float 和 double 是以二进制存储的，所以有一定的误差。

9.MySQL 怎么存储 emoji😊?

MySQL 可以直接使用字符串存储 emoji。

但是需要注意的，utf8 编码是不行的，MySQL 中的 utf8 是阉割版的 utf8，它最多只用 3 个字节存储字符，所以存储不了表情。那该怎么办?

需要使用 utf8mb4 编码。

```
alter table blogs modify content text CHARACTER SET utf8mb4 COLLATE
utf8mb4_unicode_ci not null;
```

10.drop、delete 与 truncate 的区别?

三者都表示删除，但是三者有一些差别：

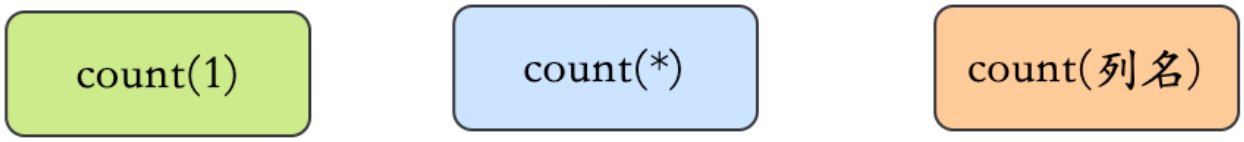
delete	truncate	drop
类型	属于 DML	属于 DDL
回滚	可回滚	不可回滚
删除内容	表结构还在，删除表的全部或者一部分数据行	表结构还在，删除表中的所有数据
删除速度	删除速度慢，需要逐行删除	删除速度快

因此，在不再需要一张表的时候，用 drop；在想删除部分数据行时候，用 delete；在保留表而删除所有数据的时候用 truncate。

11.UNION 与 UNION ALL 的区别？

- 如果使用 UNION ALL，不会合并重复的记录行
- 效率 UNION 高于 UNION ALL

12.count(1)、count(*) 与 count(列名) 的区别？



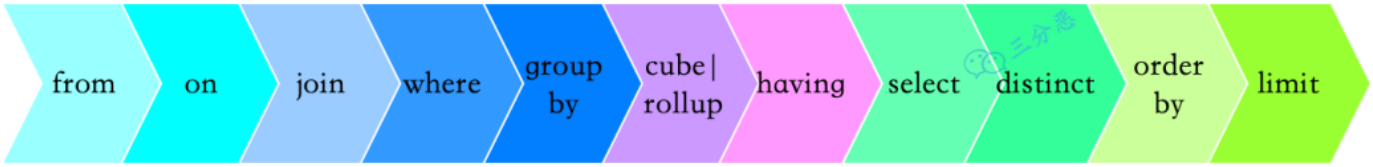
执行效果：

- count(*)包括了所有的列，相当于行数，在统计结果的时候，不会忽略列值为 NULL
- count(1)包括了忽略所有列，用 1 代表代码行，在统计结果的时候，不会忽略列值为 NULL
- count(列名)只包括列名那一列，在统计结果的时候，会忽略列值为空（这里的空不是只空字符串或者 0，而是表示 null）的计数，即某个字段值为 NULL 时，不统计。

执行速度：

- 列名为主键，count(列名)会比 count(1)快
- 列名不为主键，count(1)会比 count(列名)快
- 如果表多个列并且没有主键，则 count（1） 的执行效率优于 count（*）
- 如果有主键，则 select count（主键）的执行效率是最优的
- 如果表只有一个字段，则 select count（*）最优。

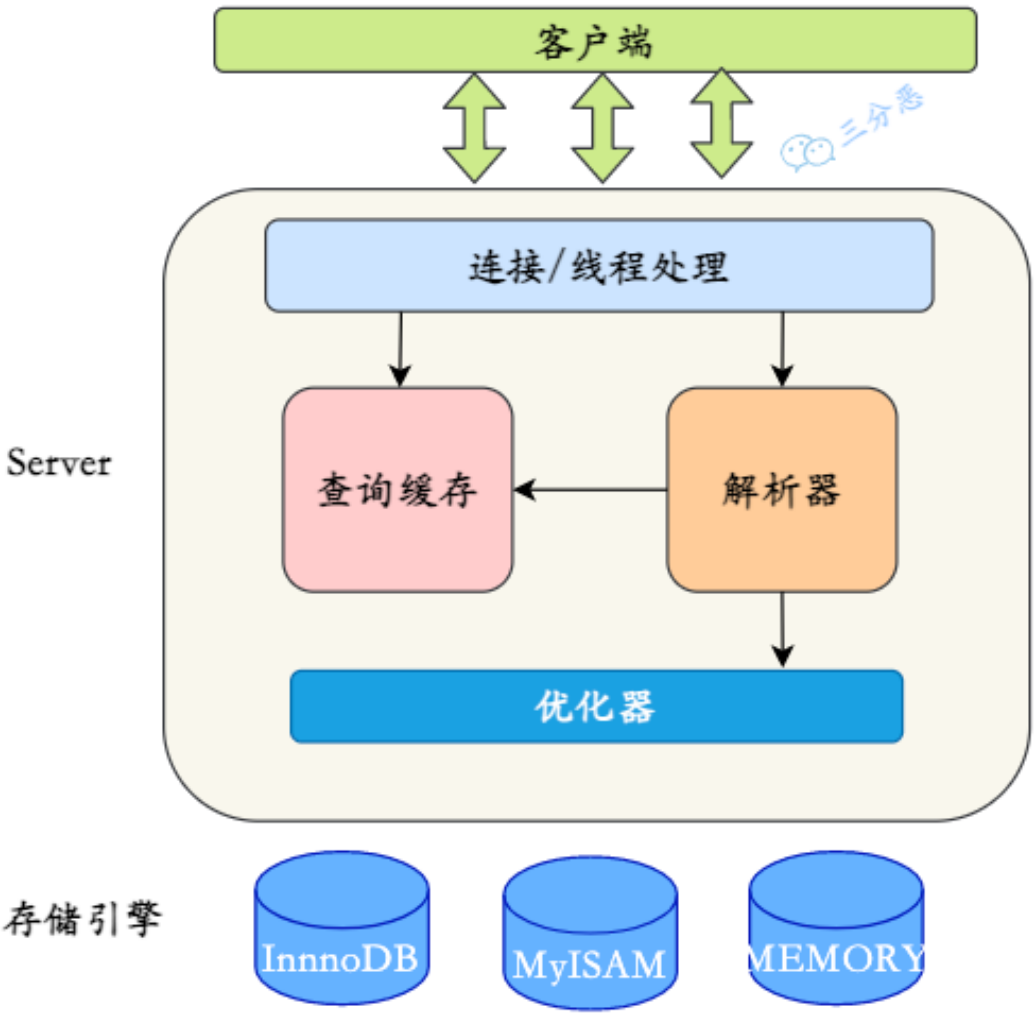
13.一条 SQL 查询语句的执行顺序？



1. **FROM**：对 FROM 子句中的左表<left_table>和右表<right_table>执行笛卡儿积（Cartesianproduct），产生虚拟表 VT1
2. **ON**：对虚拟表 VT1 应用 ON 筛选，只有那些符合<join_condition>的行才被插入虚拟表 VT2 中
3. **JOIN**：如果指定了 OUTER JOIN（如 LEFT OUTER JOIN、RIGHT OUTER JOIN），那么保留表中未匹配的行作为外部行添加到虚拟表 VT2 中，产生虚拟表 VT3。如果 FROM 子句包含两个以上表，则对上一个连接生成的结果表 VT3 和下一个表重复执行步骤 1）~步骤 3），直到处理完所有的表为止
4. **WHERE**：对虚拟表 VT3 应用 WHERE 过滤条件，只有符合<where_condition>的记录才被插入虚拟表 VT4 中
5. **GROUP BY**：根据 GROUP BY 子句中的列，对 VT4 中的记录进行分组操作，产生 VT5
6. **CUBE|ROLLUP**：对表 VT5 进行 CUBE 或 ROLLUP 操作，产生表 VT6
7. **HAVING**：对虚拟表 VT6 应用 HAVING 过滤器，只有符合<having_condition>的记录才被插入虚拟表 VT7 中。
8. **SELECT**：第二次执行 SELECT 操作，选择指定的列，插入到虚拟表 VT8 中
9. **DISTINCT**：去除重复数据，产生虚拟表 VT9
10. **ORDER BY**：将虚拟表 VT9 中的记录按照<order_by_list>进行排序操作，产生虚拟表 VT10。11）
11. **LIMIT**：取出指定行的记录，产生虚拟表 VT11，并返回给查询用户

数据库架构

14.说说 MySQL 的基础架构?



MySQL 逻辑架构图主要分三层：

- 客户端：最上层的服务并不是 MySQL 所独有的，大多数基于网络的客户端/服务器的工具或者服务都有类似的架构。比如连接处理、授权认证、安全等等。
- Server 层：大多数 MySQL 的核心服务功能都在这一层，包括查询解析、分析、优化、缓存以及所有的内置函数（例如，日期、时间、数学和加密函数），所有跨存储引擎的功能都在这一层实现：存储过程、触发器、视图等。
- 存储引擎层：第三层包含了存储引擎。存储引擎负责 MySQL 中数据的存储和提取。Server 层通过 API 与存储引擎进行通信。这些接口屏蔽了不同存储引擎之间的差异，使得这些差异对上层的查询过程透明。

15.一条 SQL 查询语句在 MySQL 中如何执行的？

- 先检查该语句`是否有权限`，如果没有权限，直接返回错误信息，如果有权限会先查询缓存(MySQL8.0 版本以前)。

- 如果没有缓存，分析器进行`语法分析`，提取 sql 语句中 select 等关键元素，然后判断 sql 语句是否有语法错误，比如关键词是否正确等等。
- 语法解析之后，MySQL 的服务器会对查询的语句进行优化，确定执行的方案。
- 完成查询优化后，按照生成的执行计划`调用数据库引擎接口`，返回执行结果。

存储引擎

16.MySQL 有哪些常见存储引擎？

主要存储引擎以及功能如下：

功能	MyISAM	MEMORY	InnoDB
存储限制	256TB	RAM	64TB
支持事务	No	No	Yes
支持全文索引	Yes	No	Yes
支持树索引	Yes	Yes	Yes
支持哈希索引	No	Yes	Yes
支持数据缓存	No	N/A	Yes
支持外键	No	No	Yes

MySQL5.5 之前，默认存储引擎是 MyISAM，5.5 之后变成了 InnoDB。

> InnoDB 支持的哈希索引是自适应的，InnoDB 会根据表的使用情况自动为表生成哈希索引，不能人为干预是否在一张表中生成哈希索引。

> MySQL 5.6 开始 InnoDB 支持全文索引。

17.那存储引擎应该怎么选择？

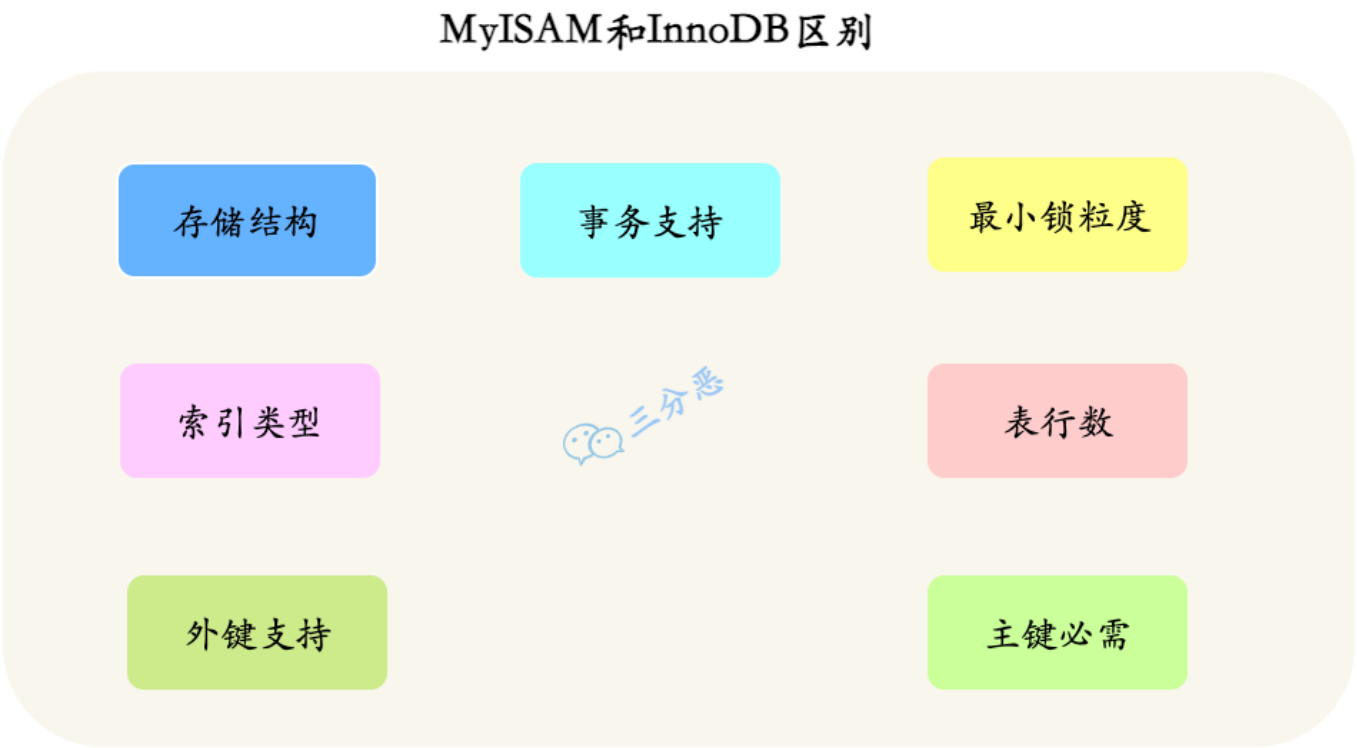
大致上可以这么选择：

- 大多数情况下，使用默认的 InnoDB 就够了。如果要提供提交、回滚和恢复的事务安全（ACID 兼容）能力，并要求实现并发控制，InnoDB 就是比较靠前的选择了。
- 如果数据表主要用来插入和查询记录，则 MyISAM 引擎提供较高的处理效率。
- 如果只是临时存放数据，数据量不大，并且不需要较高的数据安全性，可以选择将数据保存在内存的 MEMORY 引擎中，MySQL 中使用该引擎作为临时表，存放查询的中间结果。

使用哪一种引擎可以根据需要灵活选择，因为存储引擎是基于表的，所以一个数据库中多个表可以使用不同的引擎以满足各种性能和实际需求。使用合适的存储引擎将会提高整个数据库的性能。

18.InnoDB 和 MyISAM 主要有什么区别？

PS:MySQL8.0 都开始慢慢流行了，如果不是面试，MyISAM 其实可以不用怎么了解。



1. 存储结构：每个 MyISAM 在磁盘上存储成三个文件；InnoDB 所有的表都保存在同一个数据文件中（也可能是多个文件，或者是独立的表空间文件），InnoDB 表的大小只受限于操作系统文件的大小，一般为 2GB。

2. 事务支持：MyISAM 不提供事务支持；InnoDB 提供事务支持事务，具有事务(commit)、回滚(rollback)和崩溃修复能力(crash recovery capabilities)的事务安全特性。

3 最小锁粒度：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。

4. 索引类型：MyISAM 的索引为聚簇索引，数据结构是 B 树；InnoDB 的索引是非聚簇索引，数据结构是 B+树。

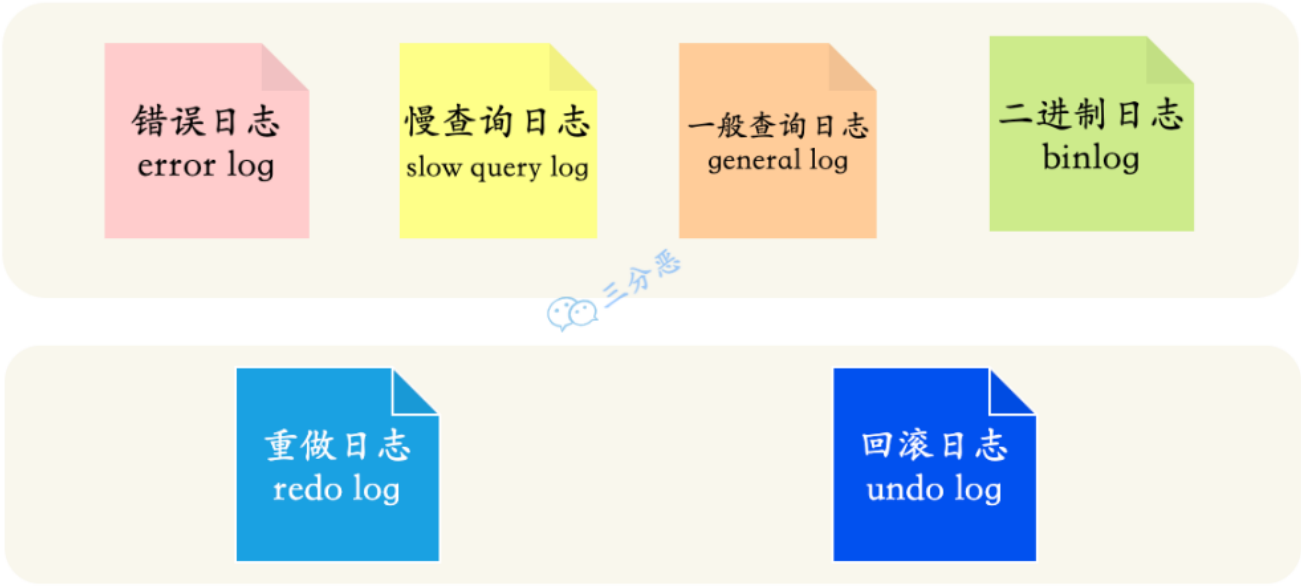
5. 主键必需：MyISAM 允许没有任何索引和主键的表存在；InnoDB 如果没有设定主键或者非空唯一索引，**就会自动生成一个 6 字节的主键(用户不可见)**，数据是主索引的一部分，附加索引保存的是主索引的值。

6. 表的具体行数：MyISAM 保存了表的总行数，如果 select count(*) from table;会直接取出出该值；InnoDB 没有保存表的总行数，如果使用 select count(*) from table；就会遍历整个表;但是在加了 wehre 条件后，MyISAM 和 InnoDB 处理的方式都一样。

7. 外键支持：MyISAM 不支持外键；InnoDB 支持外键。

日志

19.MySQL 日志文件有哪些？分别介绍下作用？



MySQL 日志文件有很多，包括：

- **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）：慢查询日志是用来记录执行时间超过 long_query_time 这个变量定义的时长的查询语句。通过慢查询日志，可以查找出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所有对 MySQL 数据库请求的信息，无论请求是否正确执行。
- **二进制日志**（bin log）：关于二进制日志，它记录了数据库所有执行的 DDL 和 DML 语句（除了数据查询语句 select、show 等），以事件形式记录并保存在二进制文件中。

还有两个 InnoDB 存储引擎特有的日志文件：

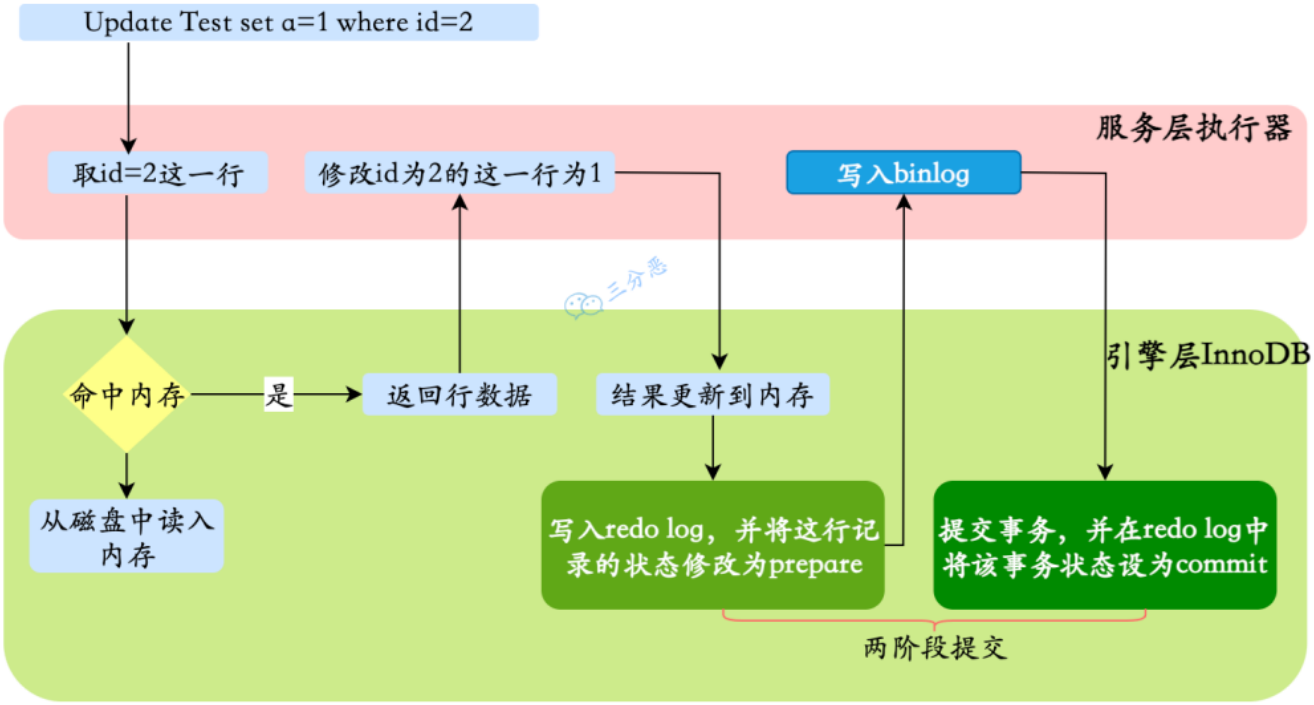
- **重做日志**（redo log）：重做日志至关重要，因为它们记录了对于 InnoDB 存储引擎的事务日志。
- **回滚日志**（undo log）：回滚日志同样也是 InnoDB 引擎提供的日志，顾名思义，回滚日志的作用就是对数据进行回滚。当事务对数据库进行修改，InnoDB 引擎不仅会记录 redo log，还会生成对应的 undo log 日志；如果事务执行失败或调用了 rollback，导致事务需要回滚，就可以利用 undo log 中的信息将数据回滚到修改之前的样子。

20.binlog 和 redo log 有什么区别？

- bin log 会记录所有与数据库有关的日志记录，包括 InnoDB、MyISAM 等存储引擎的日志，而 redo log 只记 InnoDB 存储引擎的日志。
- 记录的内容不同，bin log 记录的是关于一个事务的具体操作内容，即该日志是逻辑日志。而 redo log 记录的是关于每个页（Page）的更改的物理情况。
- 写入的时间不同，bin log 仅在事务提交前进行提交，也就是只写磁盘一次。而在事务进行的过程中，却不断有 redo ertry 被写入 redo log 中。
- 写入的方式也不相同，redo log 是循环写入和擦除，bin log 是追加写入，不会覆盖已经写的文件。

21.一条更新语句怎么执行的了解吗？

更新语句的执行是 Server 层和引擎层配合完成，数据除了要写入表中，还要记录相应的日志。



1. 执行器先找引擎获取 ID=2 这一行。ID 是主键，存储引擎检索数据，找到这一行。如果 ID=2 这一行所在的数据页本来就在内存中，就直接返回给执行器；否则，需要先从磁盘读入内存，然后再返回。
2. 执行器拿到引擎给的行数据，把这个值加上 1，比如原来是 N，现在就是 N+1，得到新的一行数据，再调用引擎接口写入这行新数据。
3. 引擎将这行新数据更新到内存中，同时将这个更新操作记录到 redo log 里面，此时 redo log 处于 prepare 状态。然后告知执行器执行完成了，随时可以提交事务。
4. 执行器生成这个操作的 binlog，并把 binlog 写入磁盘。
5. 执行器调用引擎的提交事务接口，引擎把刚刚写入的 redo log 改成提交（commit）状态，更新完成。

从上图可以看出，MySQL 在执行更新语句的时候，在服务层进行语句的解析和执行，在引擎层进行数据的提取和存储；同时在服务层对 binlog 进行写入，在 InnoDB 内进行 redo log 的写入。

不仅如此，在对 redo log 写入时有两个阶段的提交，一是 binlog 写入之前`prepare`状态的写入，二是 binlog 写入之后`commit`状态的写入。

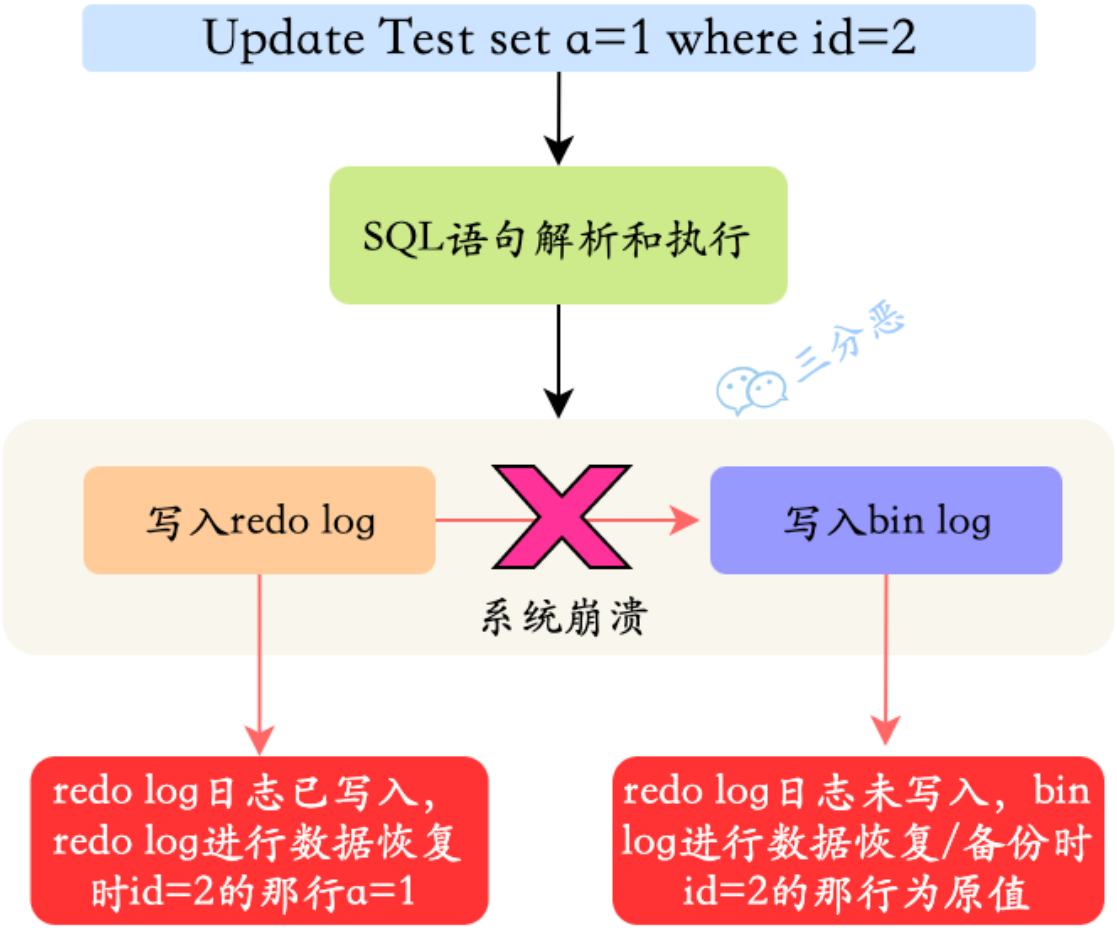
22.那为什么要两阶段提交呢？

为什么要两阶段提交呢？直接提交不行吗？

我们可以假设不采用两阶段提交的方式，而是采用“单阶段”进行提交，即要么先写入 redo log，后写入 binlog；要么先写入 binlog，后写入 redo log。这两种方式的提交都会导致原先数据库的状态和被恢复后的数据库的状态不一致。

先写入 redo log，后写入 binlog：

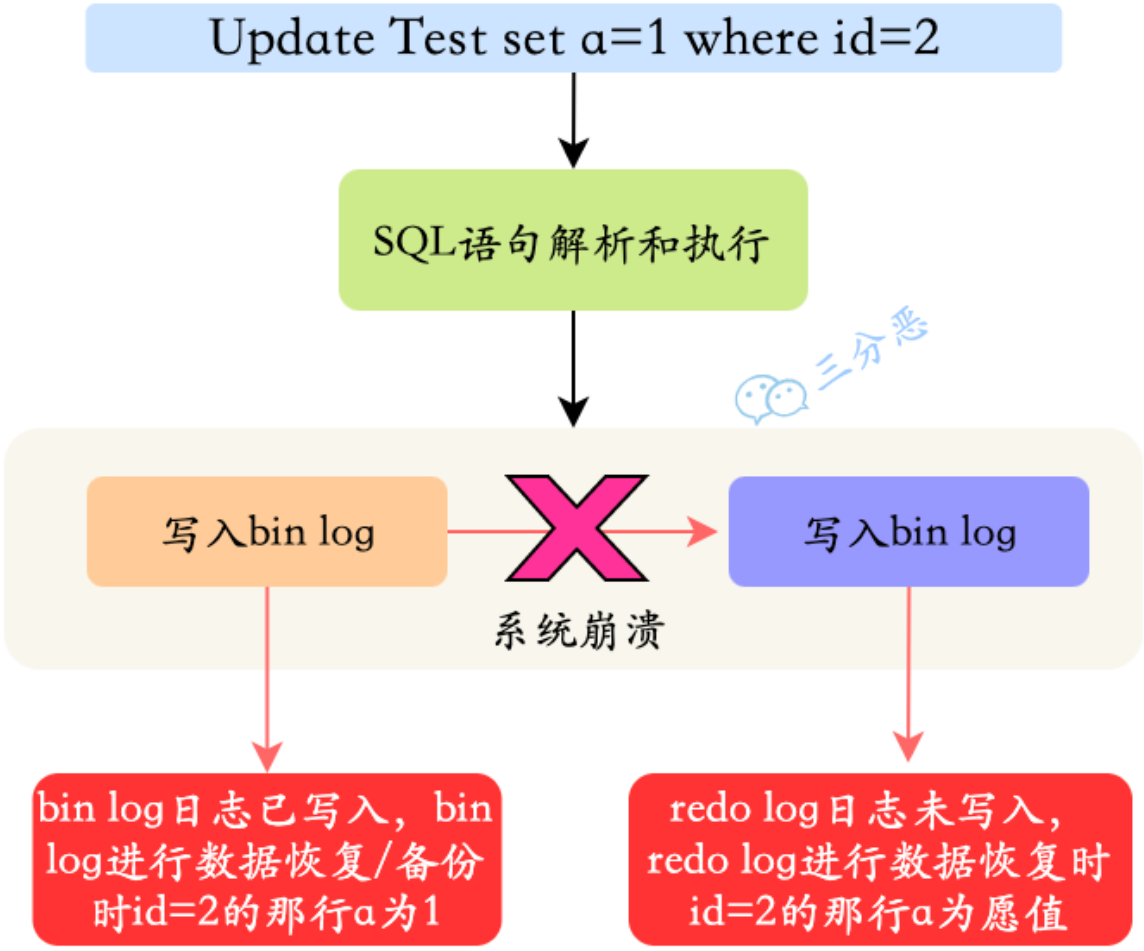
在写完 redo log 之后，数据此时具有`crash-safe`能力，因此系统崩溃，数据会恢复成事务开始之前的状态。但是，若在 redo log 写完时候，binlog 写入之前，系统发生了宕机。此时 binlog 没有对上面的更新语句进行保存，导致当使用 binlog 进行数据库的备份或者恢复时，就少了上述的更新语句。从而使得`id=2`这一行的数据没有被更新。



先写redo log，后写bin log的问题

先写入 binlog，后写入 redo log：

写完 binlog 之后，所有的语句都被保存，所以通过 binlog 复制或恢复出来的数据库中 id=2 这一行的数据会被更新为 a=1。但是如果在 redo log 写入之前，系统崩溃，那么 redo log 中记录的这个事务会无效，导致实际数据库中`id=2`这一行的数据并没有更新。

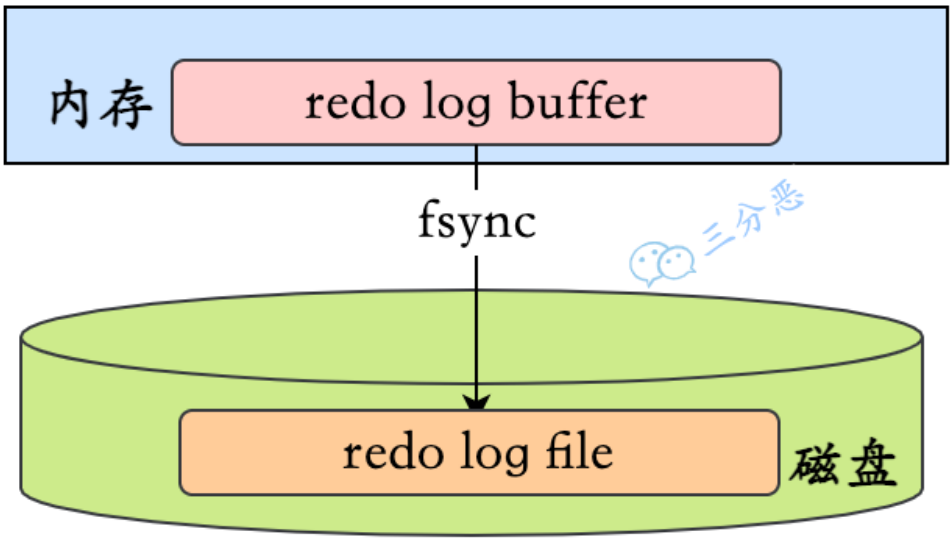


先写bin log，后写redo log的问题

简单说，redo log 和 binlog 都可以用于表示事务的提交状态，而两阶段提交就是让这两个状态保持逻辑上的一致。

23.redo log 怎么刷入磁盘的知道吗?

redo log 的写入不是直接落到磁盘，而是在内存中设置了一片称之为`redo log buffer`的连续内存空间，也就是`redo 日志缓冲区`。



什么时候会刷入磁盘?

在如下的一些情况中，log buffer 的数据会刷入磁盘:

- log buffer 空间不足时

log buffer 的大小是有限的，如果不停的往这个有限大小的 log buffer 里塞入日志，很快它就会被填满。如果当前写入 log buffer 的 redo 日志量已经占满了 log buffer 总容量的大约一半左右，就需要把这些日志刷新到磁盘上。

- 事务提交时

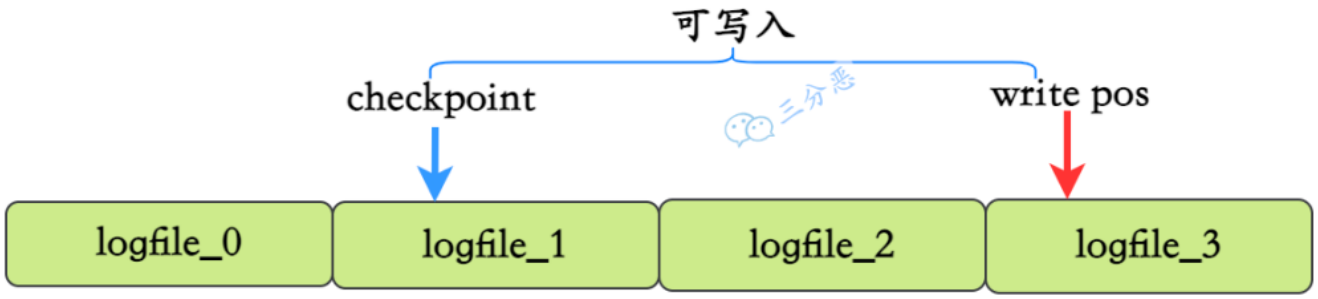
在事务提交时，为了保证持久性，会把 log buffer 中的日志全部刷到磁盘。注意，这时候，除了本事务的，可能还会刷入其它事务的日志。

- 后台线程输入

有一个后台线程，大约每秒都会刷新一次`log buffer`中的`redo log`到磁盘。

- 正常关闭服务器时
- 触发 **checkpoint** 规则

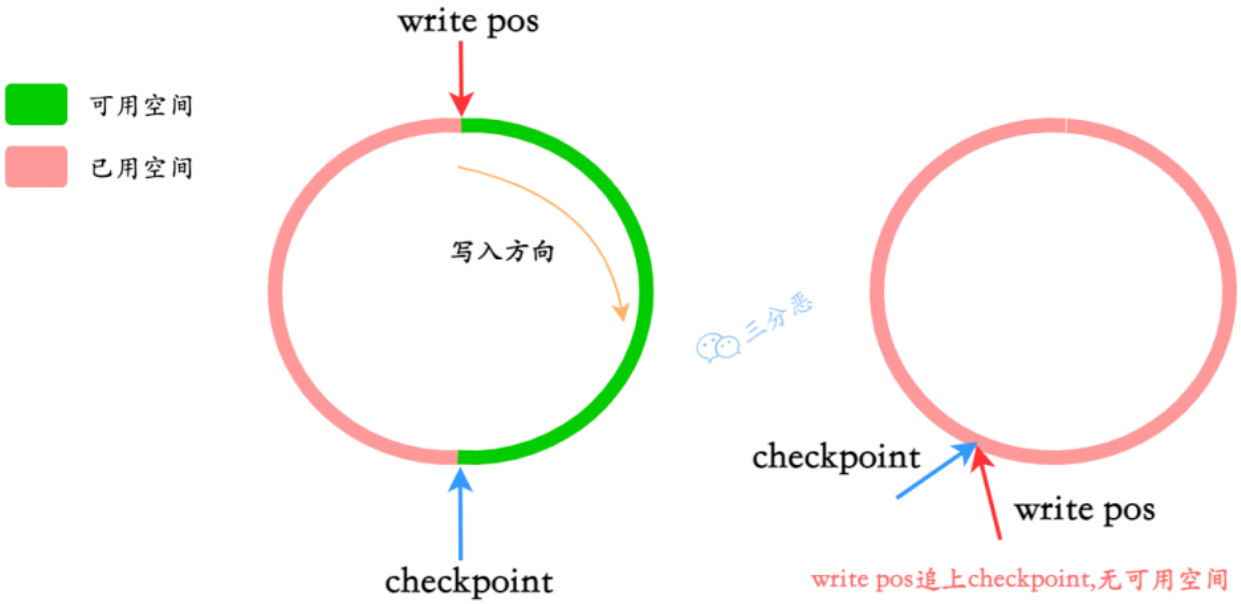
重做日志缓存、重做日志文件都是以块（**block**）的方式进行保存的，称之为重做日志块（**redo log block**）,块的大小是固定的 512 字节。我们的 redo log 它是固定大小的，可以看作是一个逻辑上的**log group**，由一定数量的**log block** 组成。



它的写入方式是从头到尾开始写，写到末尾又回到开头循环写。

其中有两个标记位置：

`write pos`是当前记录的位置，一边写一边后移，写到第 3 号文件末尾后就回到 0 号文件开头。`checkpoint`是当前要擦除的位置，也是往后推移并且循环的，擦除记录前要把记录更新到磁盘。



当`write_pos`追上`checkpoint`时，表示 redo log 日志已经写满。这时候就不能接着往里写数据了，需要执行`checkpoint`规则腾出可写空间。

所谓的**checkpoint 规则**，就是 checkpoint 触发后，将 buffer 中日志页都刷到磁盘。

SQL 优化

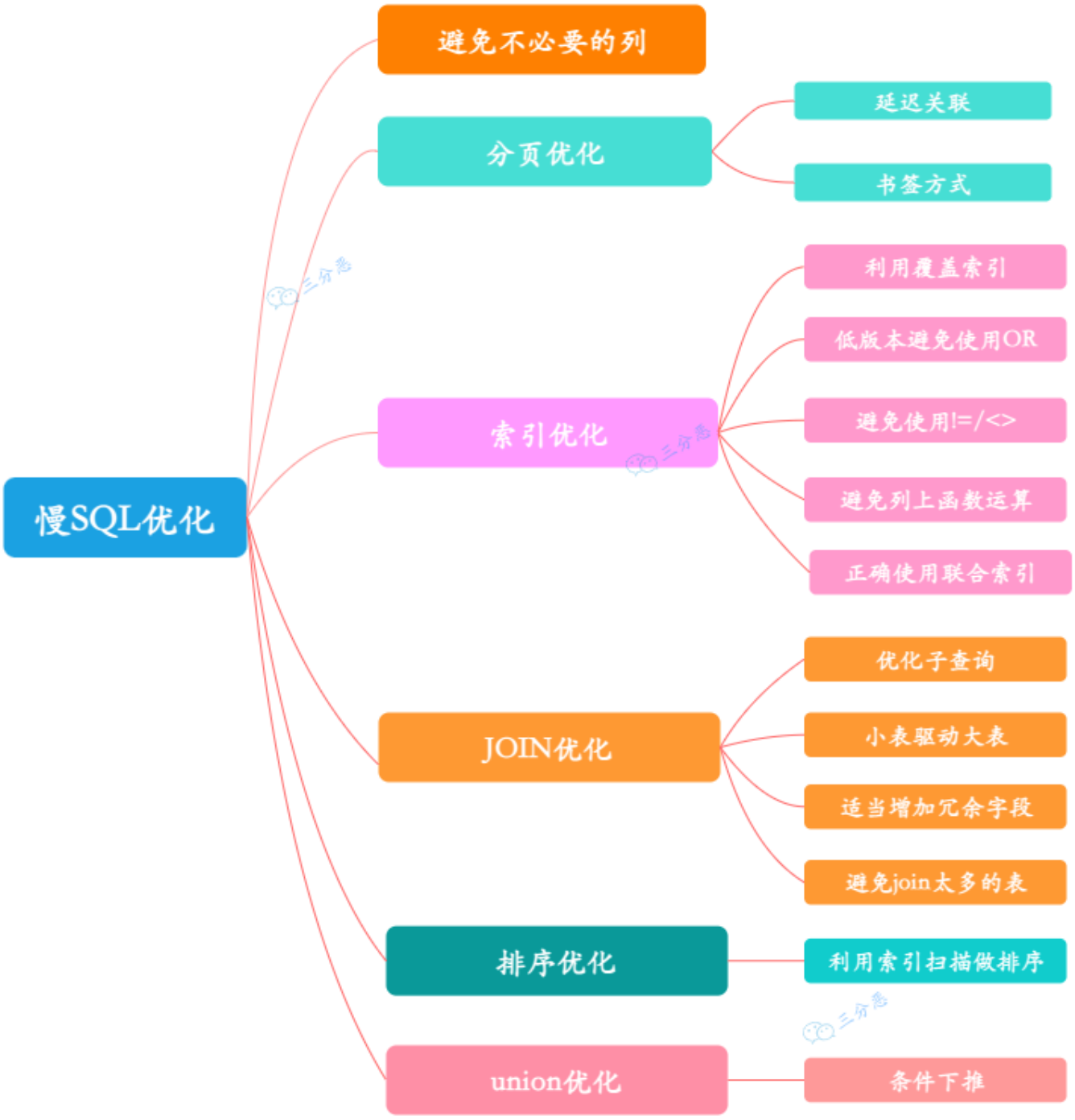
24.慢 SQL 如何定位呢？

慢 SQL 的监控主要通过两个途径：

- **慢查询日志**：开启 MySQL 的慢查询日志，再通过一些工具比如 mysqldumpslow 去分析对应的慢查询日志，当然现在一般的云厂商都提供了可视化的平台。
- **服务监控**：可以在业务的基建中加入对慢 SQL 的监控，常见的方案有字节码插桩、连接池扩展、ORM 框架过程，对服务运行中的慢 SQL 进行监控和告警。

25.有哪些方式优化慢 SQL?

慢 SQL 的优化，主要从两个方面考虑，SQL 语句本身的优化，以及数据库设计的优化。



避免不必要的列

这个是老生常谈，但还是经常会出的情况，SQL 查询的时候，应该只查询需要的列，而不要包含额外的列，像`slect *`这种写法应该尽量避免。

分页优化

在数据量比较大，分页比较深的情况下，需要考虑分页的优化。

例如：

```
select * from table where type = 2 and level = 9 order by id asc limit 190289,10;
```

优化方案：

- 延迟关联

先通过 where 条件提取出主键，在将该表与原数据表关联，通过主键 id 提取数据行，而不是通过原来的二级索引提取数据行

例如：

```
select a.* from table a,
 (select id from table where type = 2 and level = 9 order by id asc limit 190289,10 ) b
 where a.id = b.id
```

- 书签方式

书签方式就是找到 limit 第一个参数对应的主键值，根据这个主键值再去过滤并 limit

例如：

```
 select * from table where id >
 (select * from table where type = 2 and level = 9 order by id asc limit 190
```

索引优化

合理地设计和使用索引，是优化慢 SQL 的利器。

利用覆盖索引

InnoDB 使用非主键索引查询数据时会回表，但是如果索引的叶节点中已经包含要查询的字段，那它没有必要再回表查询了，这就叫覆盖索引

例如对于如下查询：

```
select name from test where city='上海'
```

我们将被查询的字段建立到联合索引中，这样查询结果就可以直接从索引中获取

```
alter table test add index idx_city_name (city, name);
```

低版本避免使用 or 查询

在 MySQL 5.0 之前的版本要尽量避免使用 or 查询，可以使用 union 或者子查询来替代，因为早期的 MySQL 版本使用 or 查询可能会导致索引失效，高版本引入了索引合并，解决了这个问题。

避免使用 != 或者 <> 操作符

SQL 中，不等于操作符会导致查询引擎放弃查询索引，引起全表扫描，即使比较的字段上有索引

解决方法：通过把不等于操作符改成 or，可以使用索引，避免全表扫描

例如，把`column<>'aaa'，改成column>'aaa' or column<'aaa'`，就可以使用索引了

适当使用前缀索引

适当地使用前缀所云，可以降低索引的空间占用，提高索引的查询效率。

比如，邮箱的后缀都是固定的“`@xxx.com`”，那么类似这种后面几位为固定值的字段就非常适合定义为前缀索引

```
alter table test add index index2(email(6));
```

PS:需要注意的是，前缀索引也存在缺点，MySQL 无法利用前缀索引做 order by 和 group by 操作，也无法作为覆盖索引

避免列上函数运算

要避免在列字段上进行算术运算或其他表达式运算，否则可能会导致存储引擎无法正确使用索引，从而影响了查询的效率

```
select * from test where id + 1 = 50;
select * from test where month(updateTime) = 7;
```

正确使用联合索引

使用联合索引的时候，注意最左匹配原则。

JOIN 优化

优化子查询

尽量使用 Join 语句来替代子查询，因为子查询是嵌套查询，而嵌套查询会新创建一张临时表，而临时表的创建与销毁会占用一定的系统资源以及花费一定的时间，同时对于返回结果集比较大的子查询，其对查询性能的影响更大

小表驱动大表

关联查询的时候要拿小表去驱动大表，因为关联的时候，MySQL 内部会遍历驱动表，再去连接被驱动表。

比如 left join，左表就是驱动表，A 表小于 B 表，建立连接的次数就少，查询速度就被加快了。

```
select name from A left join B ;
```

适当增加冗余字段

增加冗余字段可以减少大量的连表查询，因为多张表的连表查询性能很低，所有可以适当的增加冗余字段，以减少多张表的关联查询，这是以空间换时间的优化策略

避免使用 JOIN 关联太多的表

《阿里巴巴 Java 开发手册》规定不要 join 超过三张表，第一 join 太多降低查询的速度，第二 join 的 buffer 会占用更多的内存。

如果不可避免要 join 多张表，可以考虑使用数据异构的方式异构到 ES 中查询。

排序优化

利用索引扫描做排序

MySQL 有两种方式生成有序结果：其一是对结果集进行排序的操作，其二是按照索引顺序扫描得出的结果自然是有序的

但是如果索引不能覆盖查询所需列，就不得不每扫描一条记录回表查询一次，这个读操作是随机 IO，通常会比顺序全表扫描还慢

因此，在设计索引时，尽可能使用同一个索引既满足排序又用于查找行

例如：

```
--建立索引（date,staff_id,customer_id）
select staff_id, customer_id from test where date = '2010-01-01' order by
staff_id,customer_id;
```

只有当索引的列顺序和 ORDER BY 子句的顺序完全一致，并且所有列的排序方向都一样时，才能够使用索引来对结果做排序

UNION 优化

条件下推

MySQL 处理 union 的策略是先创建临时表，然后将各个查询结果填充到临时表中最后再来做查询，很多优化策略在 union 查询中都会失效，因为它无法利用索引

最好手工将 where、limit 等子句下推到 union 的各个子查询中，以便优化器可以充分利用这些条件进行优化

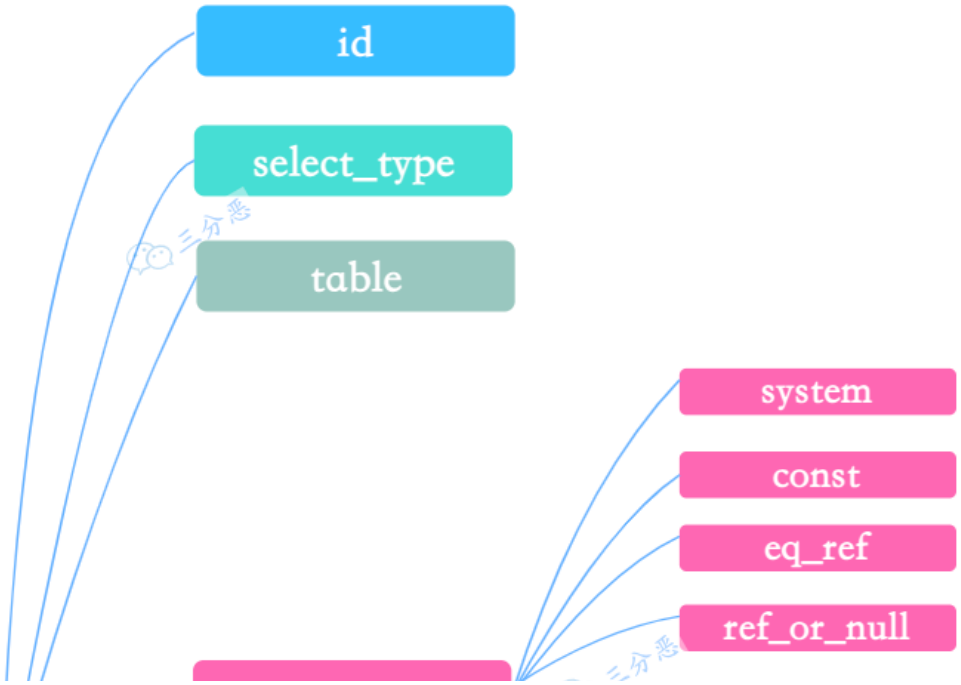
此外，除非确实需要服务器去重，一定要使用 union all，如果不加 all 关键字，MySQL 会给临时表加上 distinct 选项，这会导致对整个临时表做唯一性检查，代价很高。

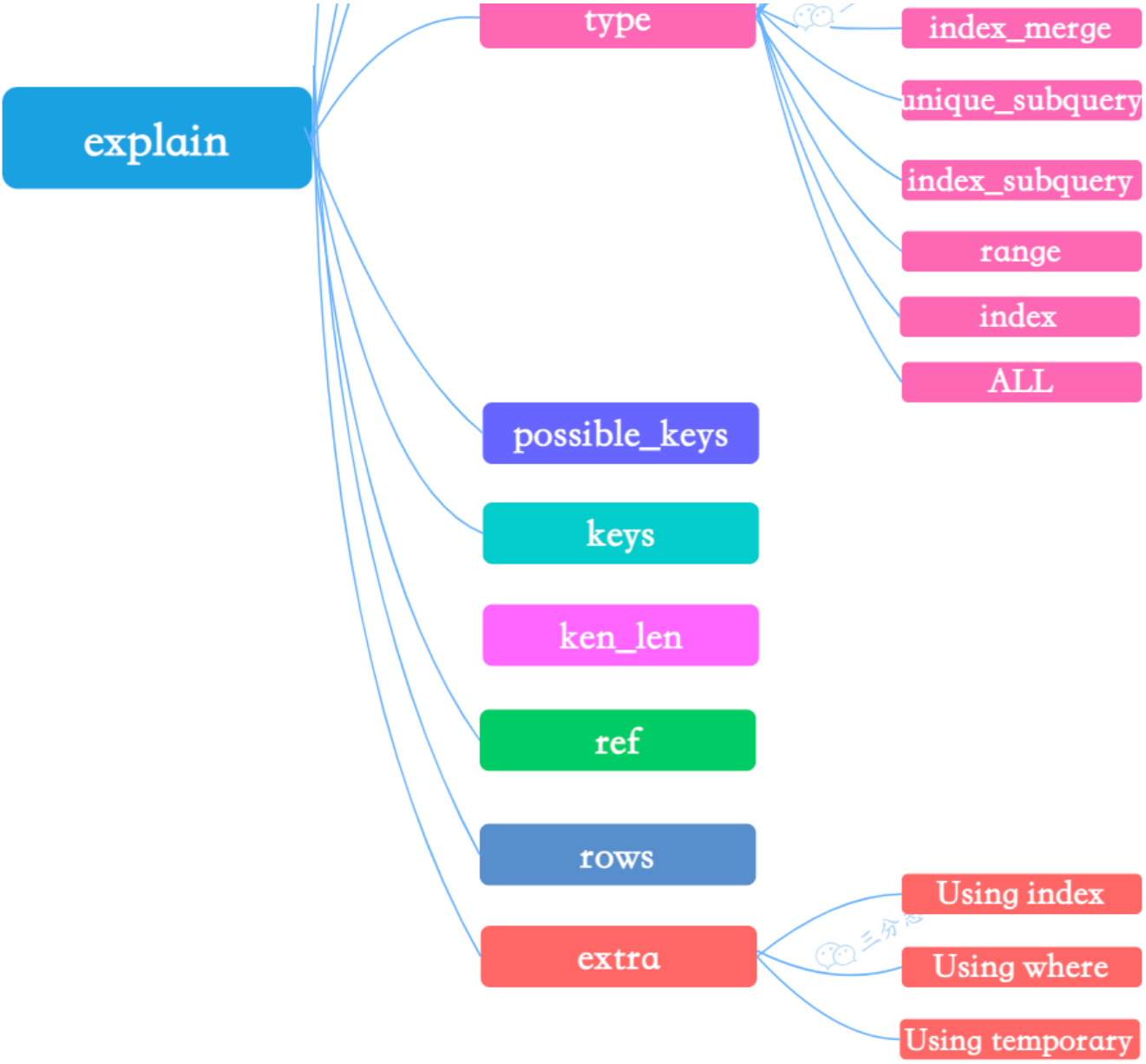
26.怎么看执行计划（explain），如何理解其中各个字段的含义？

explain 是 sql 优化的利器，除了优化慢 sql，平时的 sql 编写，也应该先 explain，查看一下执行计划，看看是否还有优化的空间。

直接在 select 语句之前增加`explain`关键字，就会返回执行计划的信息。

```
mysql> explain select name from student;
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
| id | select_type | table | type | possible_keys | key  | key_len | ref  | rows | Extra |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
|  1 | SIMPLE      |student| ALL  | NULL          | NULL | NULL    | NULL |    2 | NULL  |
+----+-------------+-------+------+---------------+------+---------+------+------+-------+
```





1. **id** 列：MySQL 会为每个 select 语句分配一个唯一的 id 值
2. **select_type** 列，查询的类型，根据关联、union、子查询等等分类，常见的查询类型有 SIMPLE、PRIMARY。
3. **table** 列：表示 explain 的一行正在访问哪个表。
4. **type** 列：最重要的列之一。表示关联类型或访问类型，即 MySQL 决定如何查找表中的行。

性能从最优到最差分别为：system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL

- system

`system`：当表仅有一行记录时(系统表)，数据量很少，往往不需要进行磁盘 IO，速度非常快

- const

`const`：表示查询时命中 `primary key` 主键或者 `unique` 唯一索引，或者被连接的部分是一个常量(`const`)值。这类扫描效率极高，返回数据量少，速度非常快。

- eq_ref

`eq_ref`：查询时命中主键`primary key` 或者 `unique key`索引， `type` 就是 `eq_ref`。

- ref_or_null

`ref_or_null`：这种连接类型类似于 ref，区别在于 `MySQL`会额外搜索包含`NULL`值的行。

- index_merge

`index_merge`：使用了索引合并优化方法，查询使用了两个以上的索引。

- unique_subquery

`unique_subquery`：替换下面的 `IN`子查询，子查询返回不重复的集合。

- index_subquery

`index_subquery`：区别于`unique_subquery`，用于非唯一索引，可以返回重复值。

- range

`range`：使用索引选择行，仅检索给定范围内的行。简单点说就是针对一个有索引的字段，给定范围检索数据。在`where`语句中使用 `bettween...and`、`<`、`>`、`<=`、`in` 等条件查询 `type` 都是`range`。

- index

`index`：`Index` 与`ALL` 其实都是读全表，区别在于`index`是遍历索引树读取，而`ALL`是从硬盘中读取。

- ALL

就不用多说了，全表扫描。

6. **possible_keys** 列：显示查询可能使用哪些索引来查找，使用索引优化 sql 的时候比较重要。
7. **key** 列：这一列显示 mysql 实际采用哪个索引来优化对该表的访问，判断索引是否失效的时候常用。
8. **key_len** 列：显示了 MySQL 使用
9. **ref** 列：ref 列展示的就是与索引列作等值匹配的值，常见的有：const（常量），func，NULL，字段名。
10. **rows** 列：这也是一个重要的字段，MySQL 查询优化器根据统计信息，估算 SQL 要查到结果集

需要扫描读取的数据行数，这个值非常直观显示 SQL 的效率好坏，原则上 rows 越少越好。

11. **Extra** 列：显示不适合在其它列的额外信息，虽然叫额外，但是也有一些重要的信息：

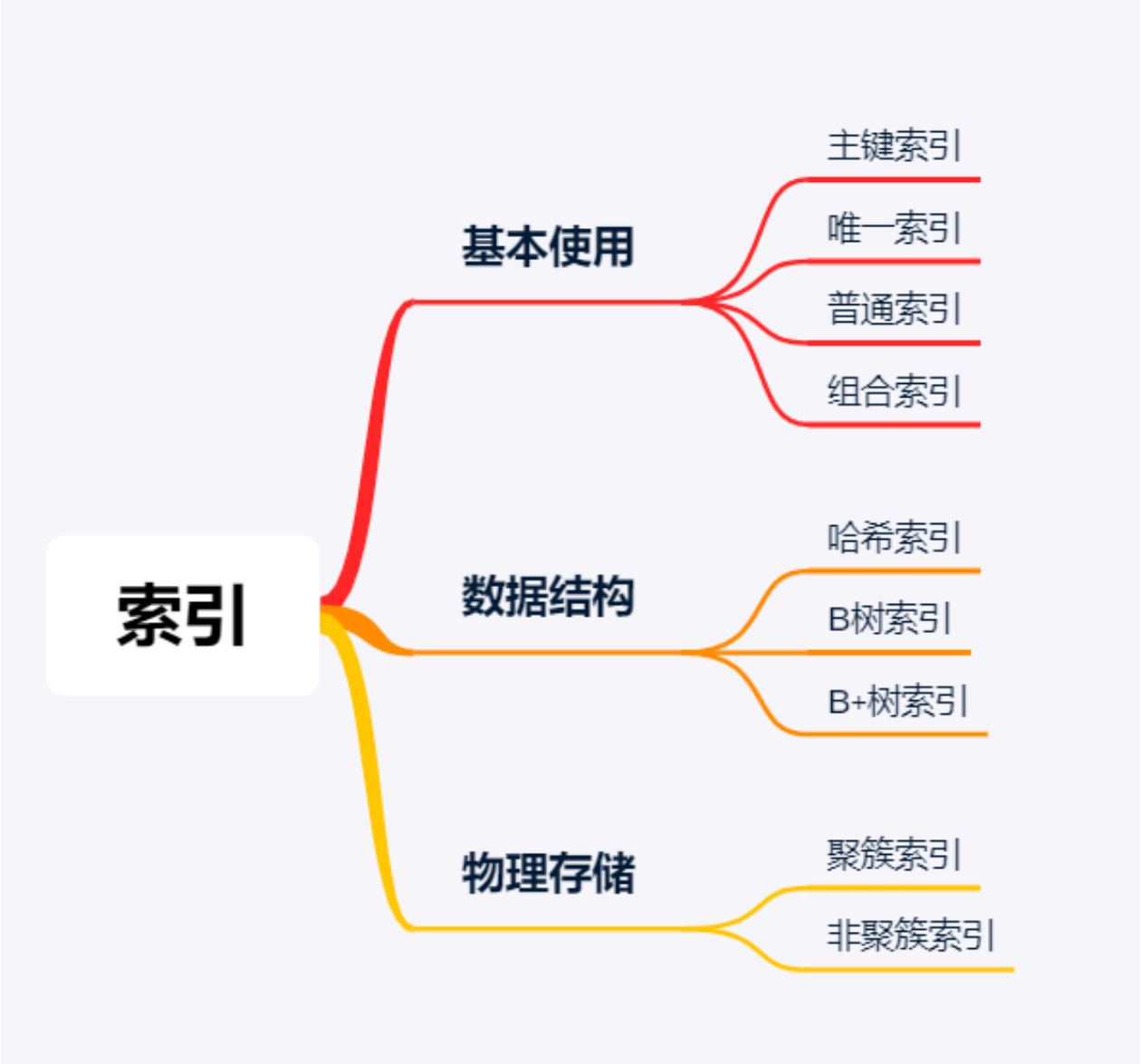
- Using index：表示 MySQL 将使用覆盖索引，以避免回表
- Using where：表示会在存储引擎检索之后再进行过滤
- Using temporary ：表示对查询结果排序时会使用一个临时表。

索引

索引可以说是 MySQL 面试中的重中之重，一定要彻底拿下。

27.能简单说一下索引的分类吗？

从三个不同维度对索引分类：



例如从基本使用使用的角度来讲：

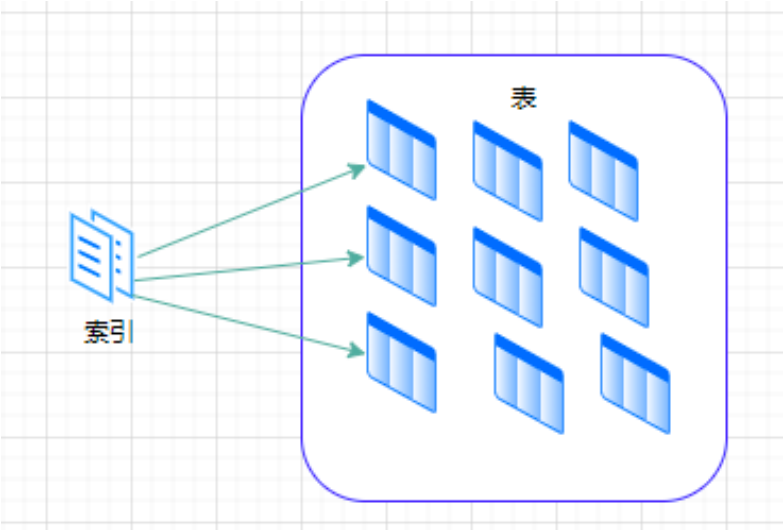
- 主键索引：InnoDB 主键是默认的索引，数据列不允许重复，不允许为 NULL，一个表只能有一个主键。
- 唯一索引：数据列不允许重复，允许为 NULL 值，一个表允许多个列创建唯一索引。
- 普通索引：基本的索引类型，没有唯一性的限制，允许为 NULL 值。
- 组合索引：多列值组成一个索引，用于组合搜索，效率大于索引合并

28.为什么使用索引会加快查询？

传统的查询方法，是按照表的顺序遍历的，不论查询几条数据，MySQL 需要将表的数据从头到尾遍历一遍。

在我们添加完索引之后，MySQL 一般通过 BTREE 算法生成一个索引文件，在查询数据库时，找到索引文件进行遍历，在比较小的索引数据里查找，然后映射到对应的数据，能大幅提升查找的效率。

和我们通过书的目录，去查找对应的内容，一样的道理。



29.创建索引有哪些注意点?

索引虽然是 sql 性能优化的利器，但是索引的维护也是需要成本的，所以创建索引，也要注意:

1. 索引应该建在查询应用频繁的字段

在用于 where 判断、 order 排序和 join 的(on)字段上创建索引。

2. 索引的个数应该适量

索引需要占用空间；更新时候也需要维护。

3. 区分度低的字段，例如性别，不要建索引。

离散度太低的字段，扫描的行数降低的有限。

4. 频繁更新的值，不要作为主键或者索引

维护索引文件需要成本；还会导致页分裂，IO 次数增多。

5. 组合索引把散列性高(区分度高)的值放在前面

为了满足最左前缀匹配原则

6. 创建组合索引，而不是修改单列索引。

组合索引代替多个单列索引（对于单列索引，MySQL 基本只能使用一个索引，所以经常使用多个条件查询时更适合使用组合索引）

7. 过长的字段，使用前缀索引。当字段值比较长的时候，建立索引会消耗很多的空间，搜索起来也会很慢。我们可以通过截取字段的前面一部分内容建立索引，这个就叫前缀索引。
8. 不建议用无序的值(例如身份证、UUID)作为索引

当主键具有不确定性，会造成叶子节点频繁分裂，出现磁盘存储的碎片化

30.索引哪些情况下会失效呢?

- 查询条件包含 or，可能导致索引失效
- 如果字段类型是字符串，where 时一定用引号括起来，否则会因为隐式类型转换，索引失效
- like 通配符可能导致索引失效。
- 联合索引，查询时的条件列不是联合索引中的第一个列，索引失效。
- 在索引列上使用 mysql 的内置函数，索引失效。
- 对索引列运算（如，+、-、*、/），索引失效。
- 索引字段上使用（！= 或者 < >，not in）时，可能会导致索引失效。
- 索引字段上使用 is null， is not null，可能导致索引失效。
- 左连接查询或者右连接查询查询关联的字段编码格式不一样，可能导致索引失效。
- MySQL 优化器估计使用全表扫描要比使用索引快,则不使用索引。

31.索引不适合哪些场景呢?

- 数据量比较少的表不适合加索引
- 更新比较频繁的字段也不适合加索引
- 离散低的字段不适合加索引（如性别）

32.索引是不是建的越多越好呢?

当然不是。

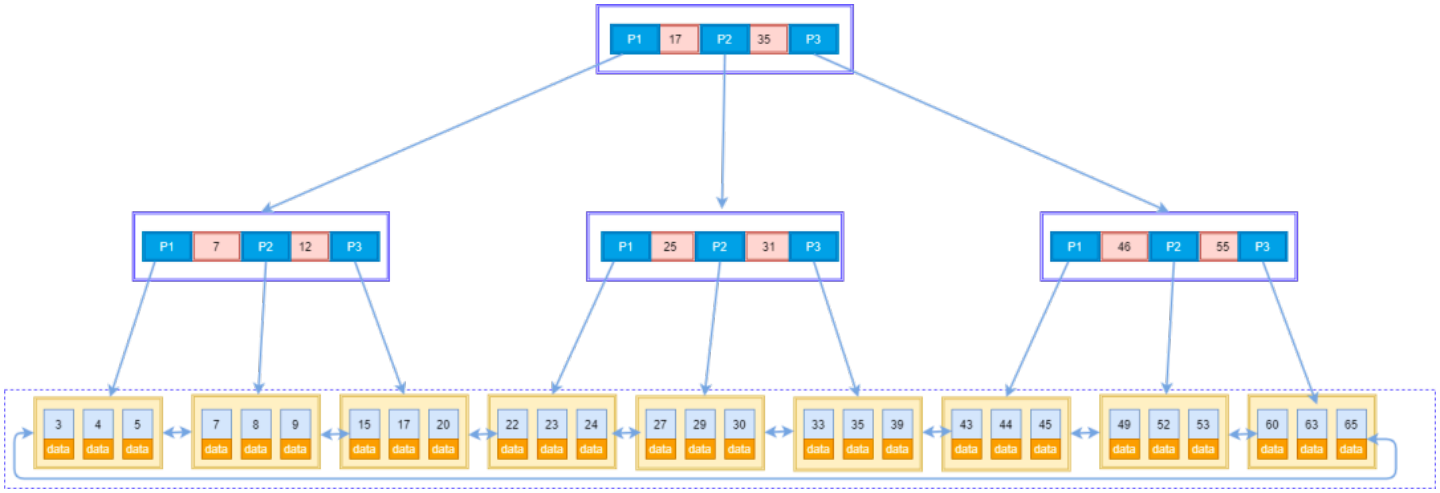
- 索引会占据磁盘空间
- **索引虽然会提高查询效率，但是会降低更新表的效率。**比如每次对表进行增删改操作，MySQL 不仅要保存数据，还有保存或者更新对应的索引文件。

33.MySQL 索引用的什么数据结构了解吗?

MySQL 的默认存储引擎是 InnoDB，它采用的是 B+树结构的索引。

- B+树：只有叶子节点才会存储数据，非叶子节点只存储键值。叶子节点之间使用双向指针连接，

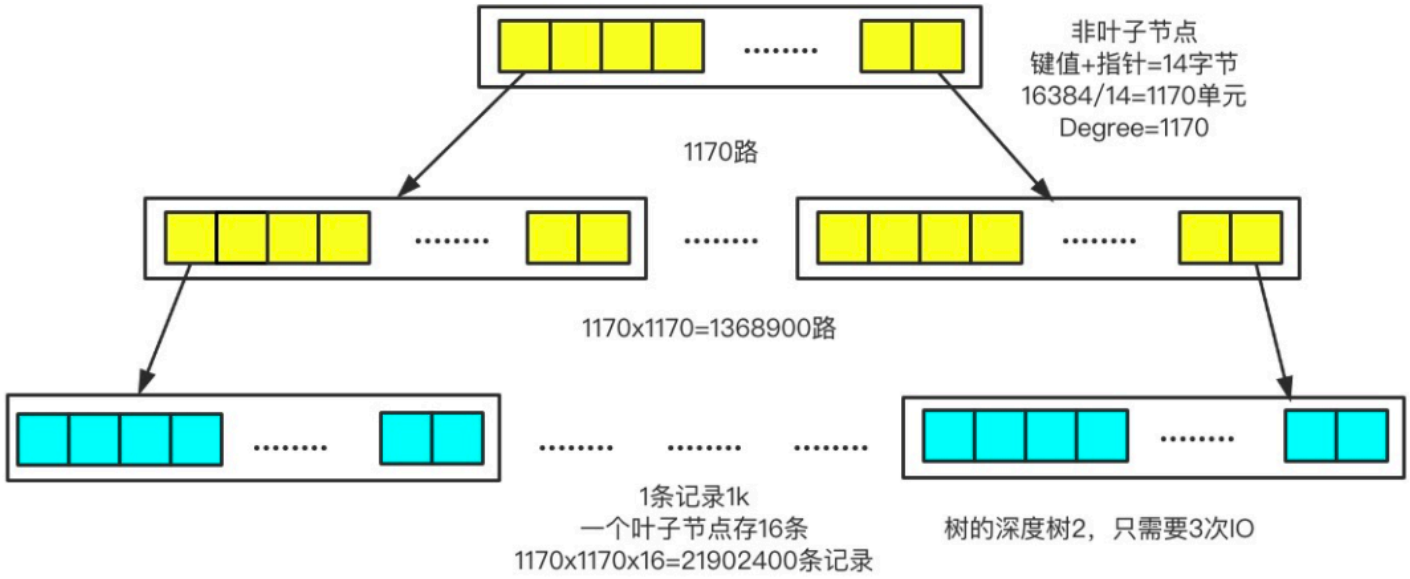
最底层的叶子节点形成了一个双向有序链表。



在这张图里，有两个重点：

- 最外面的方块，的块我们称之为一个磁盘块，可以看到每个磁盘块包含几个数据项（粉色所示）和指针（黄色/灰色所示），如根节点磁盘包含数据项 17 和 35，包含指针 P1、P2、P3，P1 表示小于 17 的磁盘块，P2 表示在 17 和 35 之间的磁盘块，P3 表示大于 35 的磁盘块。真实的数据存在于叶子节点即 3、4、5……、65。非叶子节点只不存储真实的数据，只存储指引搜索方向的数据项，如 17、35 并不真实存在于数据表中。
- 叶子节点之间使用双向指针连接，最底层的叶子节点形成了一个双向有序链表，可以进行范围查询。

34.那一棵 B+树能存储多少条数据呢？



假设索引字段是 bigint 类型，长度为 8 字节。指针大小在 InnoDB 源码中设置为 6 字节，这样一共 14 字节。非叶子节点(一页)可以存储 16384/14=1170 个这样的 单元(键值+指针)，代表有 1170 个指针。

树深度为 2 的时候，有 1170^2 个叶子节点，可以存储的数据为 1170*1170*16=**21902400**。

在查找数据时一次页的查找代表一次 IO，也就是说，一张 2000 万左右的表，查询数据最多需要访问 3 次磁盘。

所以在 InnoDB 中 B+ 树深度一般为 1-3 层，它就能满足千万级的数据存储。

35.为什么要用 B+ 树，而不用普通二叉树?

可以从几个维度去看这个问题，查询是否够快，效率是否稳定，存储数据多少，以及查找磁盘次数。

> 为什么不用普通二叉树?

普通二叉树存在退化的情况，如果它退化成链表，相当于全表扫描。平衡二叉树相比于二叉查找树来说，查找效率更稳定，总体的查找速度也更快。

> 为什么不用平衡二叉树呢?

读取数据的时候，是从磁盘读到内存。如果树这种数据结构作为索引，那每查找一次数据就需要从磁盘中读取一个节点，也就是一个磁盘块，但是平衡二叉树可是每个节点只存储一个键值和数据的，如果是 B+ 树，可以存储更多的节点数据，树的高度也会降低，因此读取磁盘的次数就降下来啦，查询效率就快。

36.为什么用 B+ 树而不用 B 树呢?

B+相比较 B 树，有这些优势:

- 它是 B Tree 的变种，B Tree 能解决的问题，它都能解决。

B Tree 解决的两大问题: 每个节点存储更多关键字; 路数更多

- 扫库、扫表能力更强

如果我们要对表进行全表扫描，只需要遍历叶子节点就可以 了，不需要遍历整棵 B+Tree 拿到所有的数据。

- B+Tree 的磁盘读写能力相对于 B Tree 来说更强，IO 次数更少

根节点和枝节点不保存数据区， 所以一个节点可以保存更多的关键字，一次磁盘加载的关键字更多，IO 次数更少。

- 排序能力更强

因为叶子节点上有下一个数据区的指针，数据形成了链表。

- 效率更加稳定

B+Tree 永远是在叶子节点拿到数据，所以 IO 次数是稳定的。

37.Hash 索引和 B+ 树索引区别是什么?

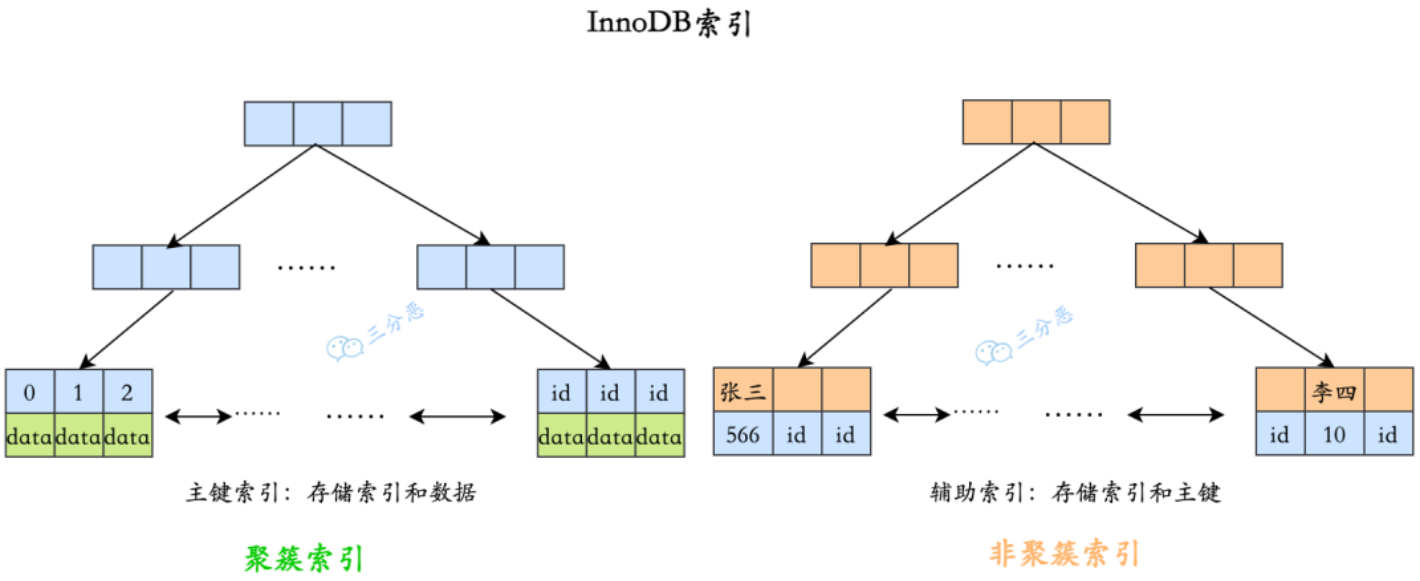
- B+ 树可以进行范围查询，Hash 索引不能。
- B+ 树支持联合索引的最左侧原则，Hash 索引不支持。
- B+ 树支持 order by 排序，Hash 索引不支持。
- Hash 索引在等值查询上比 B+ 树效率更高。
- B+ 树使用 like 进行模糊查询的时候，like 后面（比如 % 开头）的话可以起到优化的作用，Hash 索引根本无法进行模糊查询。

38.聚簇索引与非聚簇索引的区别?

首先理解聚簇索引不是一种新的索引，而是而是一种**数据存储方式**。聚簇表示数据行和相邻的键值紧凑地存储在一起。我们熟悉的两种存储引擎——MyISAM 采用的是非聚簇索引，InnoDB 采用的是聚簇索引。

可以这么说：

- 索引的数据结构是树，聚簇索引的索引和数据存储在一棵树上，树的叶子节点就是数据，非聚簇索引索引和数据不在一棵树上。



- 一个表中只能拥有一个聚簇索引，而非聚簇索引一个表可以存在多个。
- 聚簇索引，索引中键值的逻辑顺序决定了表中相应行的物理顺序；索引，索引中索引的逻辑顺序与

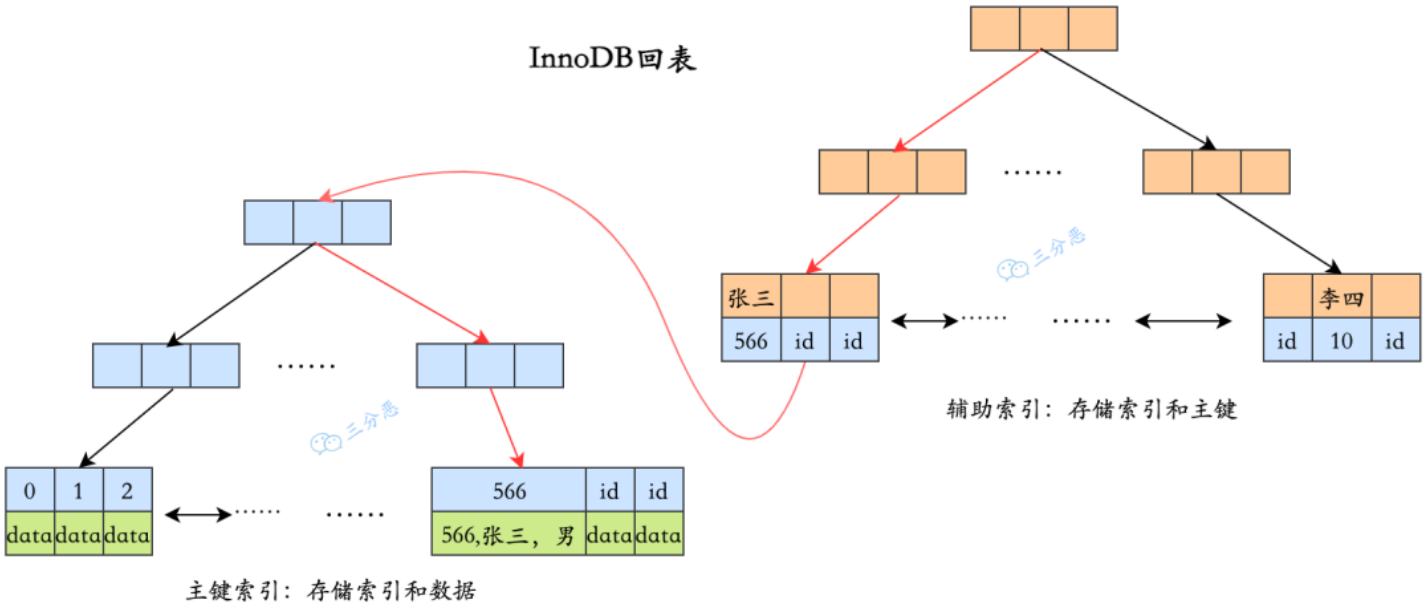
磁盘上行的物理存储顺序不同。

- 聚簇索引：物理存储按照索引排序；非聚集索引：物理存储不按照索引排序；

39.回表了解吗？

在 InnoDB 存储引擎里，利用辅助索引查询，先通过辅助索引找到主键索引的键值，再通过主键值查出主键索引里面没有符合要求的数据，它比基于主键索引的查询多扫描了一棵索引树，这个过程就叫回表。

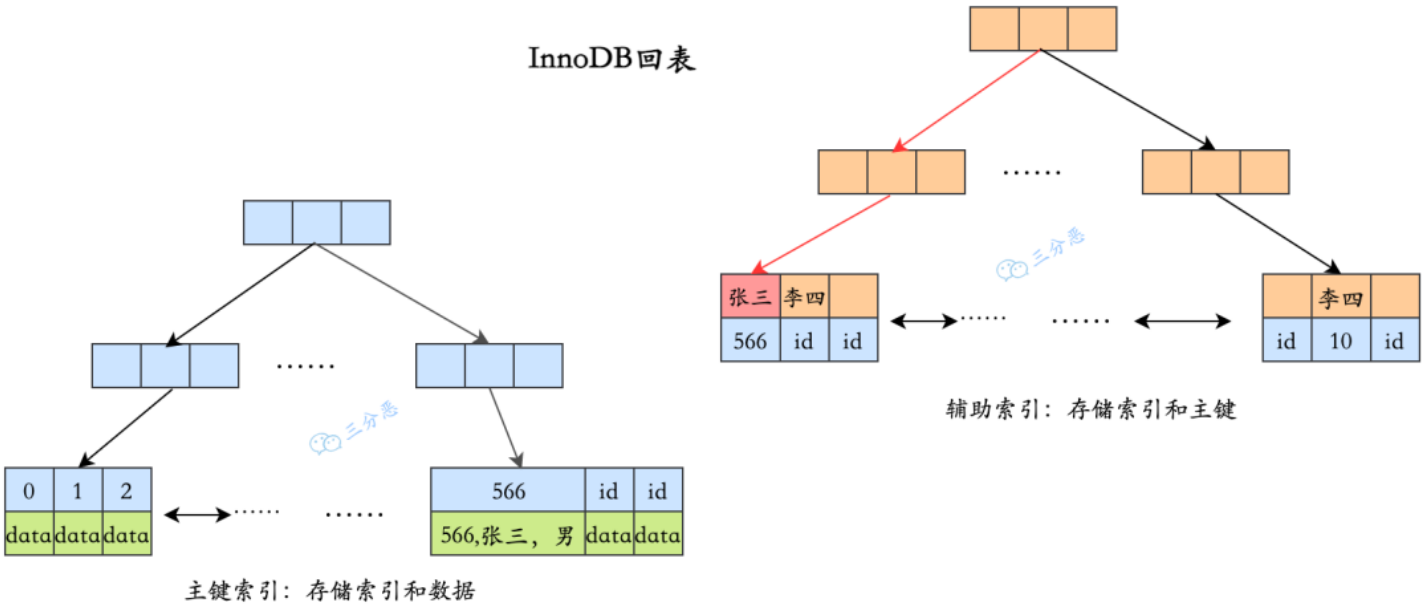
例如：`select \* from user where name = '张三';`



40.覆盖索引了解吗？

在辅助索引里面，不管是单列索引还是联合索引，如果 select 的数据列只用辅助索引中就能够取得，不用去查主键索引，这时候使用的索引就叫做覆盖索引，避免了回表。

比如，`select name from user where name = '张三';`



41.什么是最左前缀原则/最左匹配原则？

注意：最左前缀原则、最左匹配原则、最左前缀匹配原则这三个都是一个概念。

最左匹配原则：在 InnoDB 的联合索引中，查询的时候只有匹配了前一个/左边的值之后，才能匹配下一个。

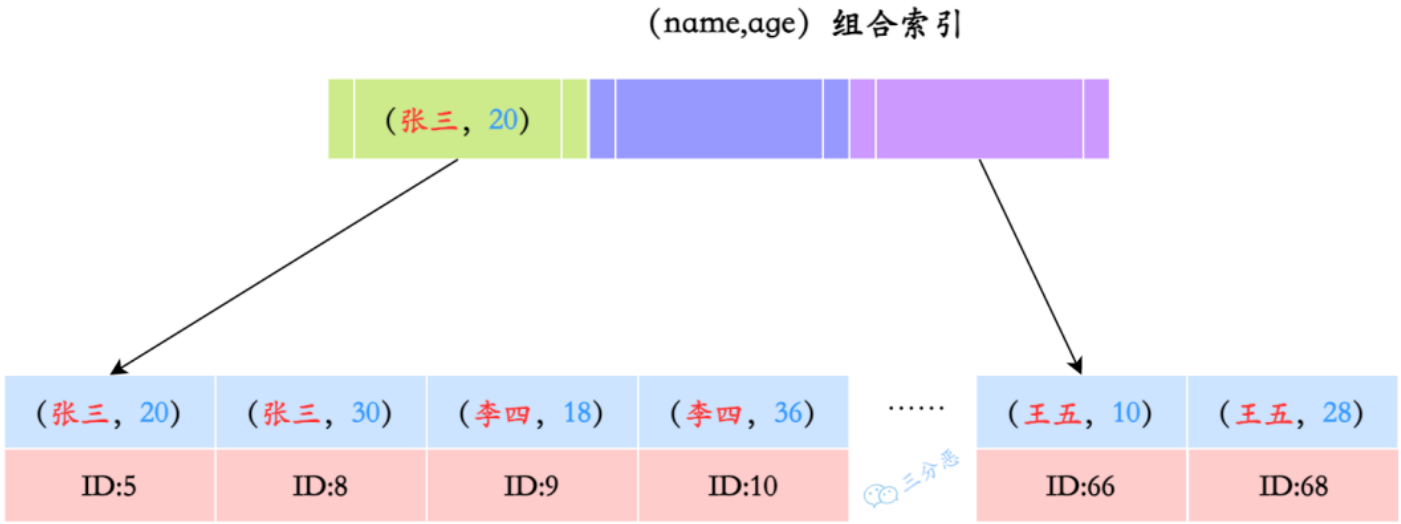
根据最左匹配原则，我们创建了一个组合索引，如 (a1,a2,a3)，相当于创建了（a1）、(a1,a2)和(a1,a2,a3) 三个索引。

为什么不从最左开始查，就无法匹配呢？

比如有一个 user 表，我们给 name 和 age 建立了一个组合索引。

```
ALTER TABLE user add INDEX comidx_name_phone (name,age);
```

组合索引在 B+Tree 中是复合的数据结构，它是按照从左到右的顺序来建立搜索树的 (name 在左边，age 在右边)。



从这张图可以看出来，name 是有序的，age 是无序的。当 name 相等的时候， age 才是有序的。

这个时候我们使用 `where name= ‘张三‘ and age = ‘20 ‘` 去查询数据的时候， B+Tree 会优先比较 name 来确定下一步应该搜索的方向，往左还是往右。如果 name 相同的时候再比较 age。但是如果查询条件没有 name，就不知道下一步应该查哪个 节点，因为建立搜索树的时候 name 是第一个比较因子，所以就没用上索引。

42.什么是索引下推优化?

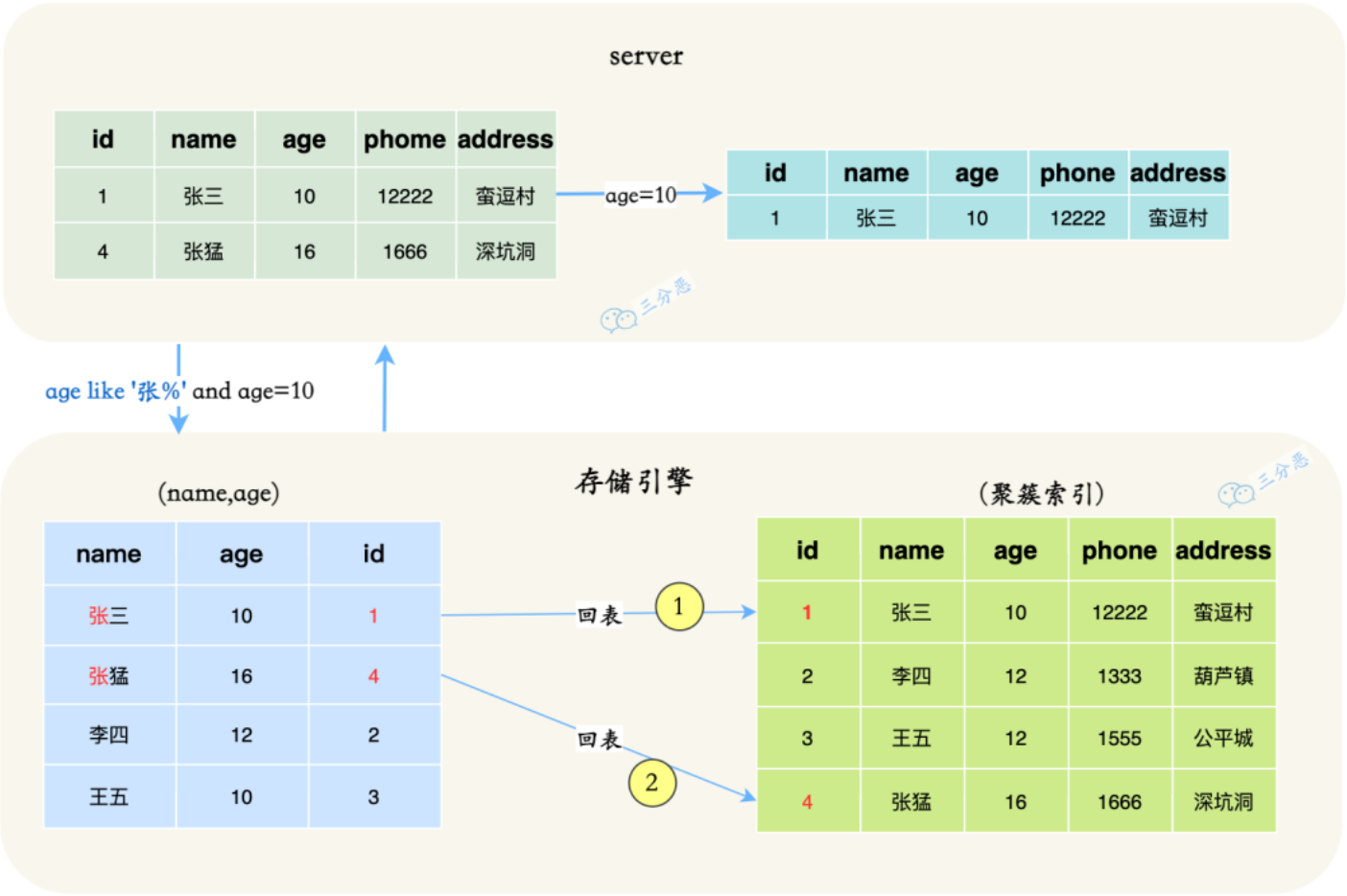
索引条件下推优化 `（Index Condition Pushdown (ICP)）` 是 MySQL5.6 添加的，用于优化数据查询。

- 不使用索引条件下推优化时存储引擎通过索引检索到数据，然后返回给 MySQL Server，MySQL Server 进行过滤条件的判断。
- 当使用索引条件下推优化时，如果存在某些被索引的列的判断条件时，MySQL Server 将这一部分判断条件下推给存储引擎，然后由存储引擎通过判断索引是否符合 MySQL Server 传递的条件，只有当索引符合条件时才会将数据检索出来返回给 MySQL 服务器。

例如一张表，建了一个联合索引（name, age），查询语句：`select * from t_user where name like '张%' and age=10;`，由于 `name` 使用了范围查询，根据最左匹配原则：

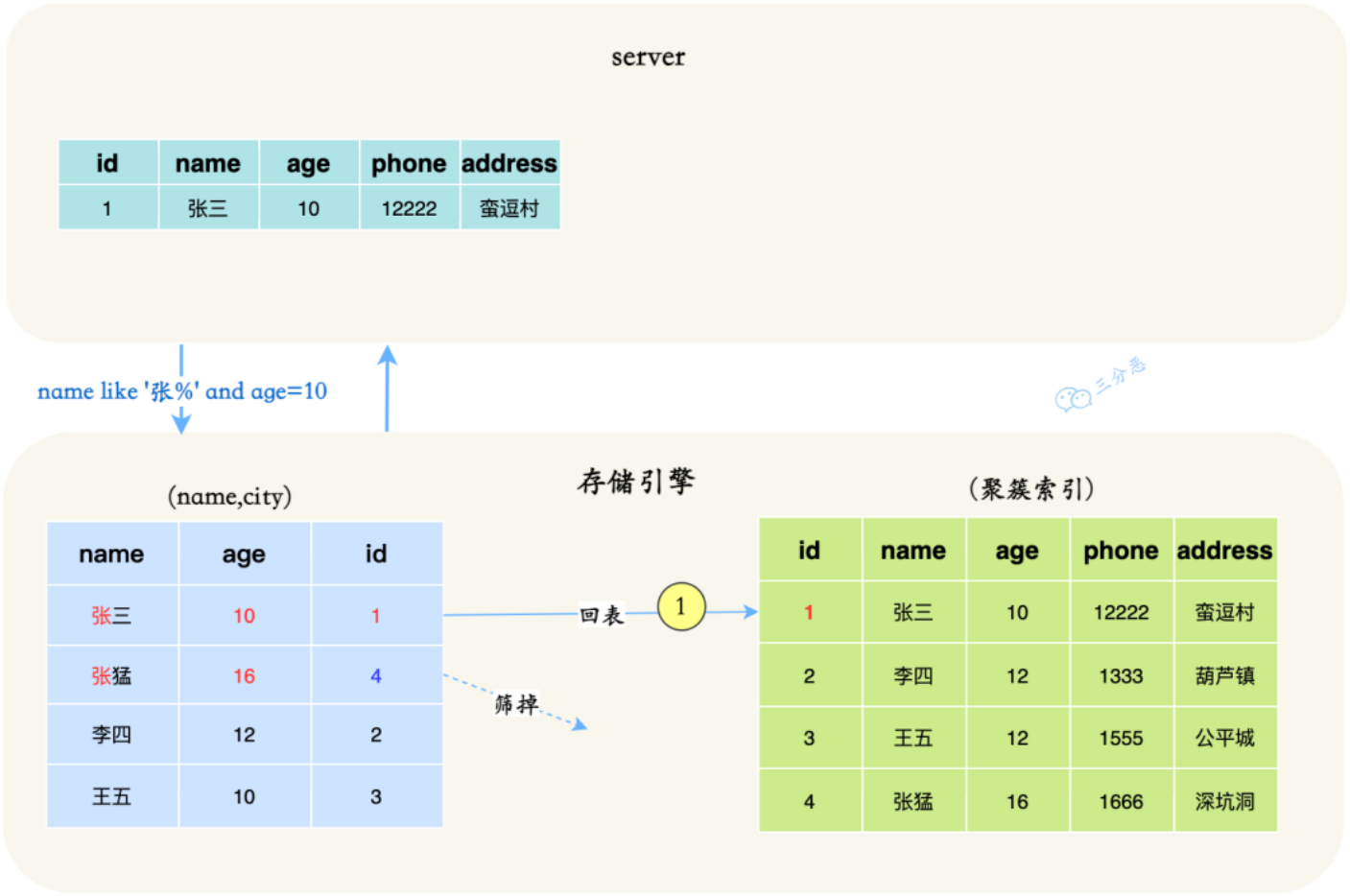
不使用 ICP，引擎层查找到 `name like '张%'` 的数据，再由 Server 层去过滤 `age=10` 这个条件，这样一来，就回表了两次，浪费了联合索引的另外一个字段 `age`。

没有使用ICP



但是，使用了索引下推优化，把 where 的条件放到了引擎层执行，直接根据`name like '张%' and age=10`的条件进行过滤，减少了回表的次数。

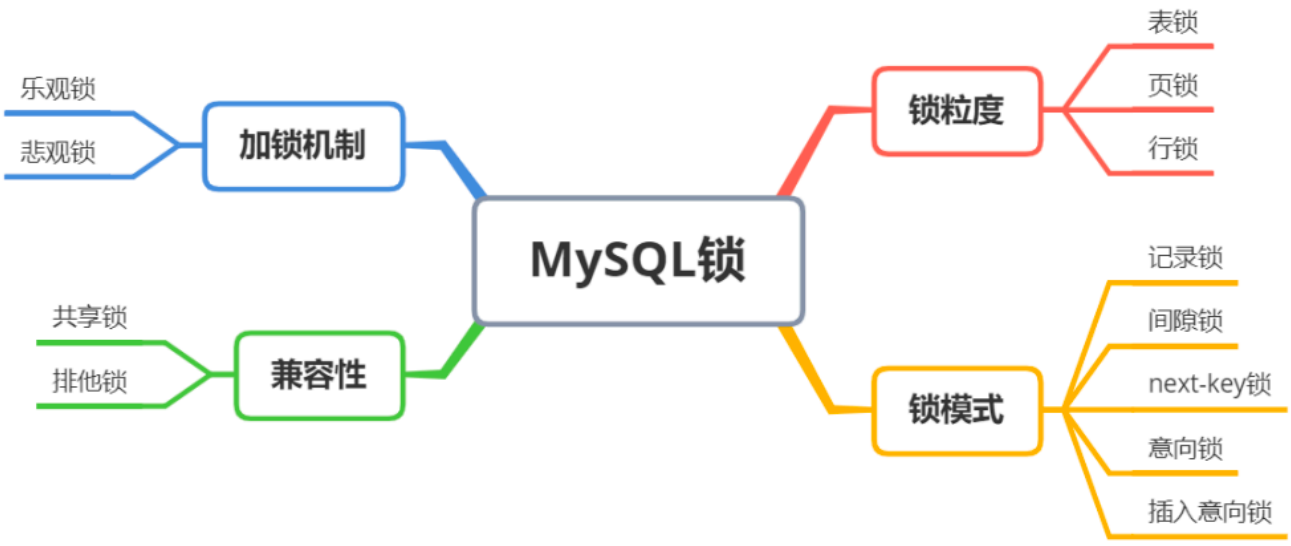
使用ICP



索引条件下推优化可以减少存储引擎查询基础表的次数，也可以减少 MySQL 服务器从存储引擎接收数据的次数。

锁

43.MySQL 中有哪几种锁，列举一下？



如果按锁粒度划分，有以下 3 种：

- 表锁：开销小，加锁快；锁定力度大，发生锁冲突概率高，并发度最低;不会出现死锁。
- 行锁：开销大，加锁慢；会出现死锁；锁定粒度小，发生锁冲突的概率低，并发度高。
- 页锁：开销和加锁速度介于表锁和行锁之间；会出现死锁；锁定粒度介于表锁和行锁之间，并发度

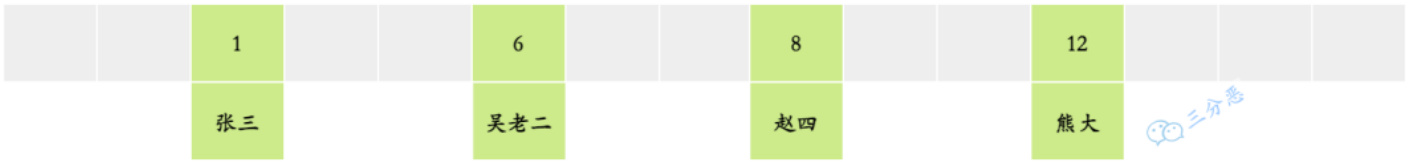
一般

如果按照兼容性，有两种，

- 共享锁（S Lock）,也叫读锁（read lock），相互不阻塞。
- 排他锁（X Lock），也叫写锁（write lock），排它锁是阻塞的，在一定时间内，只有一个请求能执行写入，并阻止其它锁读取正在写入的数据。

44.说说 InnoDB 里的行锁实现?

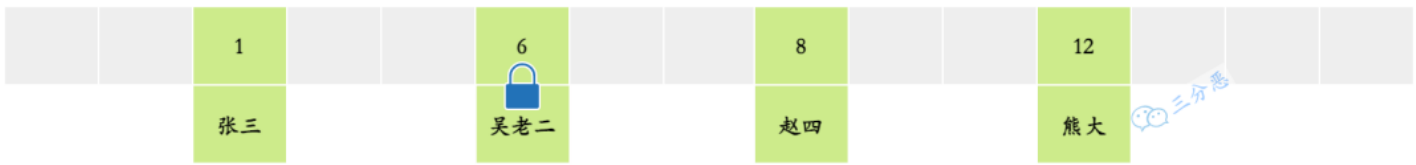
我们拿这么一个用户表来表示行级锁，其中插入了 4 行数据，主键值分别是 1,6,8,12，现在简化它的聚簇索引结构，只保留数据记录。



InnoDB 的行锁的主要实现如下：

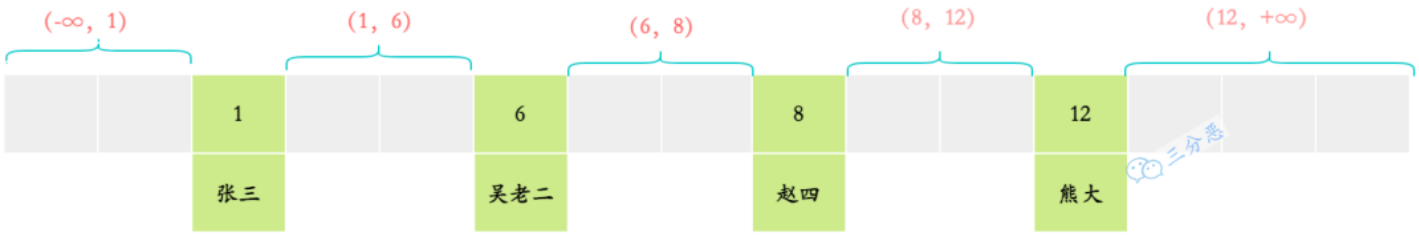
- **Record Lock 记录锁**

记录锁就是直接锁定某行记录。当我们使用唯一性的索引(包括唯一索引和聚簇索引)进行等值查询且精准匹配到一条记录时，此时就会直接将这条记录锁定。例如`select * from t where id =6 for update;`就会将`id=6`的记录锁定。



- **Gap Lock 间隙锁**

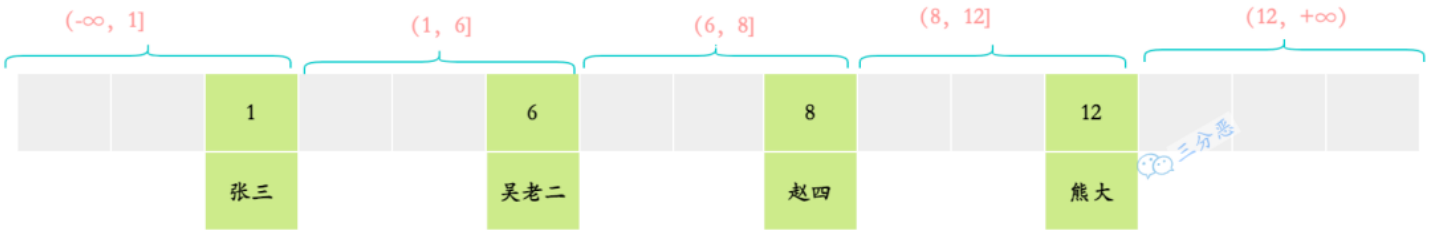
间隙锁(Gap Locks) 的间隙指的是两个记录之间逻辑上尚未填入数据的部分,是一个左开右开空间。



间隙锁就是锁定某些间隙区间的。当我们使用用等值查询或者范围查询，并且没有命中任何一个`record`，此时就会将对应的间隙区间锁定。例如`select * from t where id =3 for update;`或者`select * from t where id > 1 and id < 6 for update;`就会将(1,6)区间锁定。

- **Next-key Lock 临键锁**

临键指的是间隙加上它右边的记录组成的左开右闭区间。比如上述的(1,6]、(6,8]等。



临键锁就是记录锁(Record Locks)和间隙锁(Gap Locks)的结合，即除了锁住记录本身，还要再锁住索引之间的间隙。当我们使用范围查询，并且命中了部分`record`记录，此时锁住的就是临键区间。注意，临键锁锁住的区间会包含最后一个 record 的右边的临键区间。例如`select * from t where id > 5 and id <= 7 for update;`会锁住(4,7]、(7,+∞)。mysql 默认行锁类型就是`临键锁(Next-Key Locks)`。当使用唯一性索引，等值查询匹配到一条记录的时候，临键锁(Next-Key Locks)会退化成记录锁；没有匹配到任何记录的时候，退化成间隙锁。

> 间隙锁(Gap Locks) 和 临键锁(Next-Key Locks) 都是用来解决幻读问题的，在 已提交读（READ COMMITTED） 隔离级别下， 间隙锁(Gap Locks) 和 临键锁(Next-Key Locks) 都会失效！

上面是行锁的三种实现算法，除此之外，在行上还存在插入意向锁。

- **Insert Intention Lock 插入意向锁**

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意向锁 ，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。这种类型的锁命名为 Insert Intention Locks ，也就是插入意向锁 。

假如我们有个 T1 事务，给(1,6)区间加上了意向锁，现在有个 T2 事务，要插入一个数据，id 为 4，它会获取一个（1,6）区间的插入意向锁，又有有个 T3 事务，想要插入一个数据，id 为 3，它也会获取一个（1,6）区间的插入意向锁，但是，这两个插入意向锁锁不会互斥。

45.意向锁是什么知道吗?

意向锁是一个表级锁，不要和插入意向锁搞混。

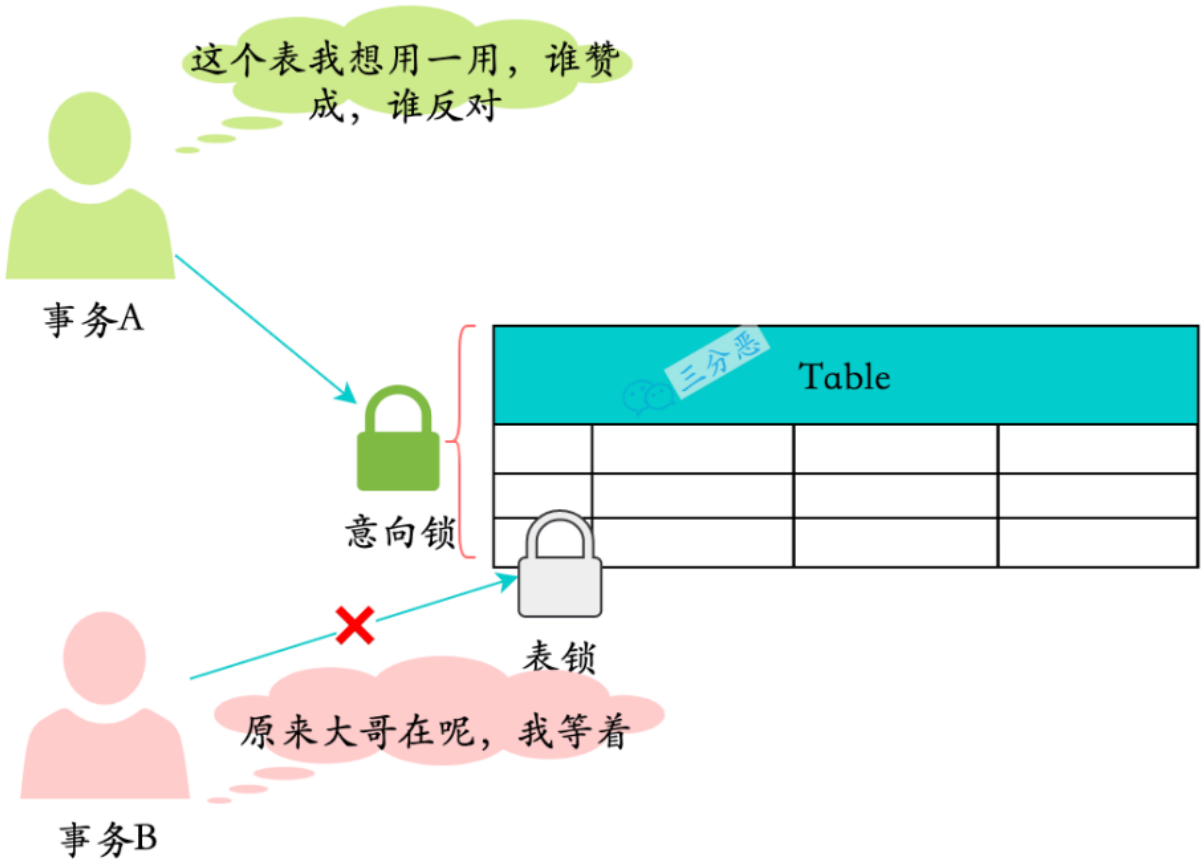
意向锁的出现是为了支持 InnoDB 的多粒度锁，它解决的是表锁和行锁共存的问题。

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。

假如没有意向锁，那么我们就得遍历表中所有数据行来判断有没有行锁；

有了意向锁这个表级锁之后，则我们直接判断一次就知道表中是否有数据行被锁定了。

有了意向锁之后，要执行的事务 A 在申请行锁（写锁）之前，数据库会自动先给事务 A 申请表的意向排他锁。当事务 B 去申请表的互斥锁时就会失败，因为表上有意向排他锁之后事务 B 申请表的互斥锁时会被阻塞。



46.MySQL 的乐观锁和悲观锁了解吗?

- 悲观锁（Pessimistic Concurrency Control）：

悲观锁认为被它保护的数据是极其不安全的，每时每刻都有可能被改动，一个事务拿到悲观锁后，其他任何事务都不能对该数据进行修改，只能等待锁被释放才可以执行。

数据库中的行锁，表锁，读锁，写锁均为悲观锁。

- **乐观锁（Optimistic Concurrency Control）**

乐观锁认为数据的变动不会太频繁。

乐观锁通常是通过在表中增加一个版本(version)或时间戳(timestamp)来实现，其中，版本最为常用。

事务在从数据库中取数据时，会将该数据的版本也取出来(v1)，当事务对数据变动完毕想要将其更新到表中时，会将之前取出的版本 v1 与数据中最新的版本 v2 相对比，如果 v1=v2，那么说明在数据变动期间，没有其他事务对数据进行修改，此时，就允许事务对表中的数据进行修改，并且修改时 version 会加 1，以此来表明数据已被变动。

如果，v1 不等于 v2，那么说明数据变动期间，数据被其他事务改动了，此时不允许数据更新到表中，一般的处理办法是通知用户让其重新操作。不同于悲观锁，乐观锁通常是由开发者实现的。

47.MySQL 遇到过死锁问题吗，你是如何解决的？

排查死锁的一般步骤是这样的：

（1）查看死锁日志 show engine innodb status;

（2）找出死锁 sql

（3）分析 sql 加锁情况

（4）模拟死锁案发

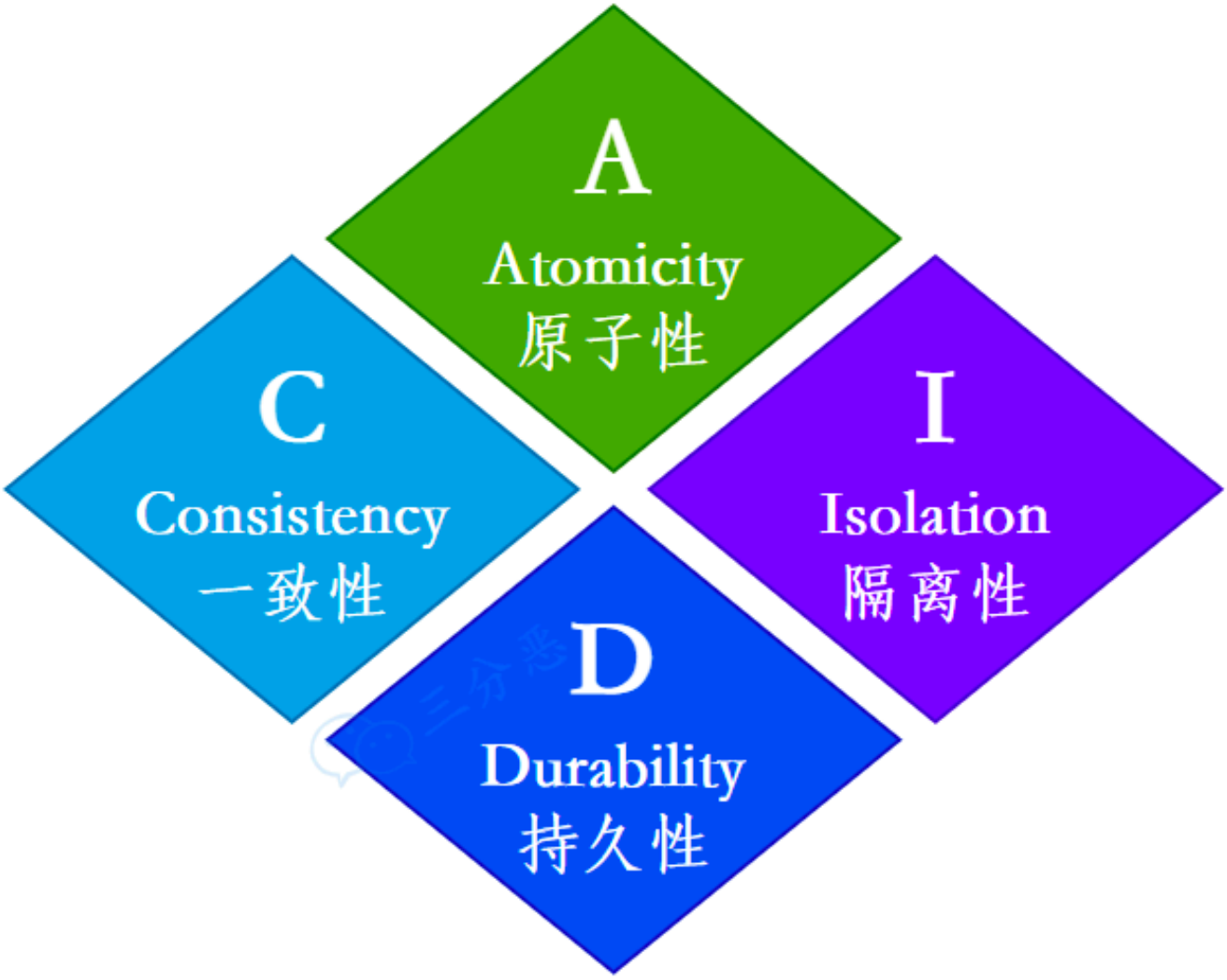
（5）分析死锁日志

（6）分析死锁结果

当然，这只是一个简单的流程说明，实际上生产中的死锁千奇百怪，排查和解决起来没那么简单。

事务

48.MySQL 事务的四大特性说一下？

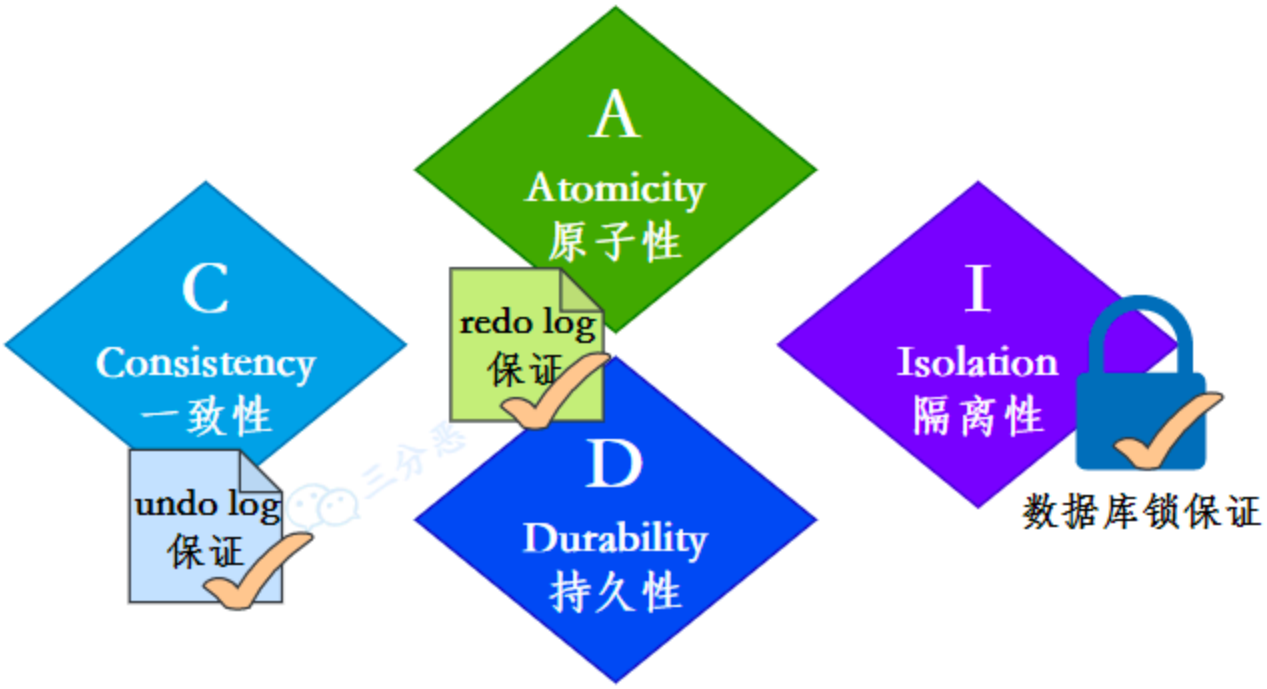


- 原子性：事务作为一个整体被执行，包含在其中的对数据库的操作要么全部被执行，要么都不执行。
- 一致性：指在事务开始之前和事务结束以后，数据不会被破坏，假如 A 账户给 B 账户转 10 块钱，不管成功与否，A 和 B 的总金额是不变的。
- 隔离性：多个事务并发访问时，事务之间是相互隔离的，即一个事务不影响其它事务运行效果。简言之，就是事务之间是进水不犯河水的。
- 持久性：表示事务完成以后，该事务对数据库所作的操作更改，将持久地保存在数据库之中。

49.那 ACID 靠什么保证的呢？

- 事务的隔离性是通过数据库锁的机制实现的。
- 事务的一致性由 undo log 来保证：undo log 是逻辑日志，记录了事务的 insert、update、deltete 操作，回滚的时候做相反的 delete、update、insert 操作来恢复数据。
- 事务的原子性和持久性由 redo log 来保证：redolog 被称作重做日志，是物理日志，事务提交的

时候，必须先将事务的所有日志写入 redo log 持久化，到事务的提交操作才算完成。



50.事务的隔离级别有哪些？MySQL 的默认隔离级别是什么？

事务的四个隔离级别

- 读未提交（Read Uncommitted）
- 读已提交（Read Committed）
- 可重复读（Repeatable Read）
- 串行化（Serializable）

MySQL 默认的事务隔离级别是可重复读 (Repeatable Read)。

51.什么是幻读，脏读，不可重复读呢？

- 事务 A、B 交替执行，事务 A 读取到事务 B 未提交的数据，这就是脏读。
- 在一个事务范围内，两个相同的查询，读取同一条记录，却返回了不同的数据，这就是不可重复读。
- 事务 A 查询一个范围的结果集，另一个并发事务 B 往这个范围中插入 / 删除了数据，并静悄悄地提交，然后事务 A 再次查询相同的范围，两次读取得到的结果集不一样了，这就是幻读。

不同的隔离级别，在并发事务下可能会发生的问题：

隔离级别	脏读	不可重复读	幻读
Read Uncommited 读取未提交	是	是	是
Read Commited 读取已提交	否	是	否
Repeatable Read 可重复读	否	否	是
Serialzable 可串行化	否	否	否

52.事务的各个隔离级别都是如何实现的？

读未提交

读未提交，就不用多说了，采取的是读不加锁原理。

- 事务读不加锁，不阻塞其他事务的读和写
- 事务写阻塞其他事务写，但不阻塞其他事务读；

读取已提交&可重复读

读取已提交和可重复读级别利用了`ReadView`和`MVCC`，也就是每个事务只能读取它能看到的版本（ReadView）。

- READ COMMITTED：每次读取数据前都生成一个 ReadView
- REPEATABLE READ ：在第一次读取数据时生成一个 ReadView

串行化

串行化的实现采用的是读写都加锁的原理。

串行化的情况下，对于同一行事务，`写`会加`写锁`，`读`会加`读锁`。当出现读写锁冲突的时候，后访问的事务必须等前一个事务执行完成，才能继续执行。

53.MVCC 了解吗？怎么实现的？

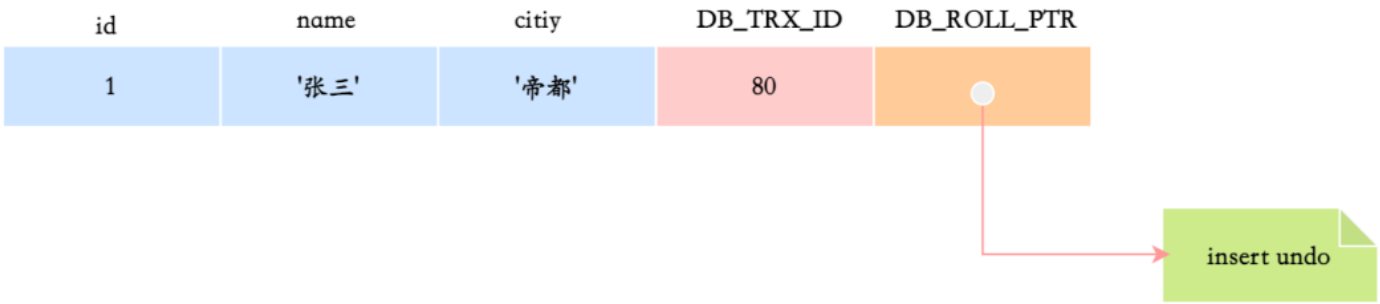
MVCC(Multi Version Concurrency Control)，中文名是多版本并发控制，简单来说就是通过维护数据历史版本，从而解决并发访问情况下的读一致性问题。关于它的实现，要抓住几个关键点，**隐式字段、undo 日志、版本链、快照读&当前读、Read View**。

版本链

对于 InnoDB 存储引擎，每一行记录都有两个隐藏列**DB_TRX_ID、DB_ROLL_PTR**

- `DB_TRX_ID`，事务 ID，每次修改时，都会把该事务 ID 复制给`DB_TRX_ID`；
- `DB_ROLL_PTR`，回滚指针，指向回滚段的 undo 日志。

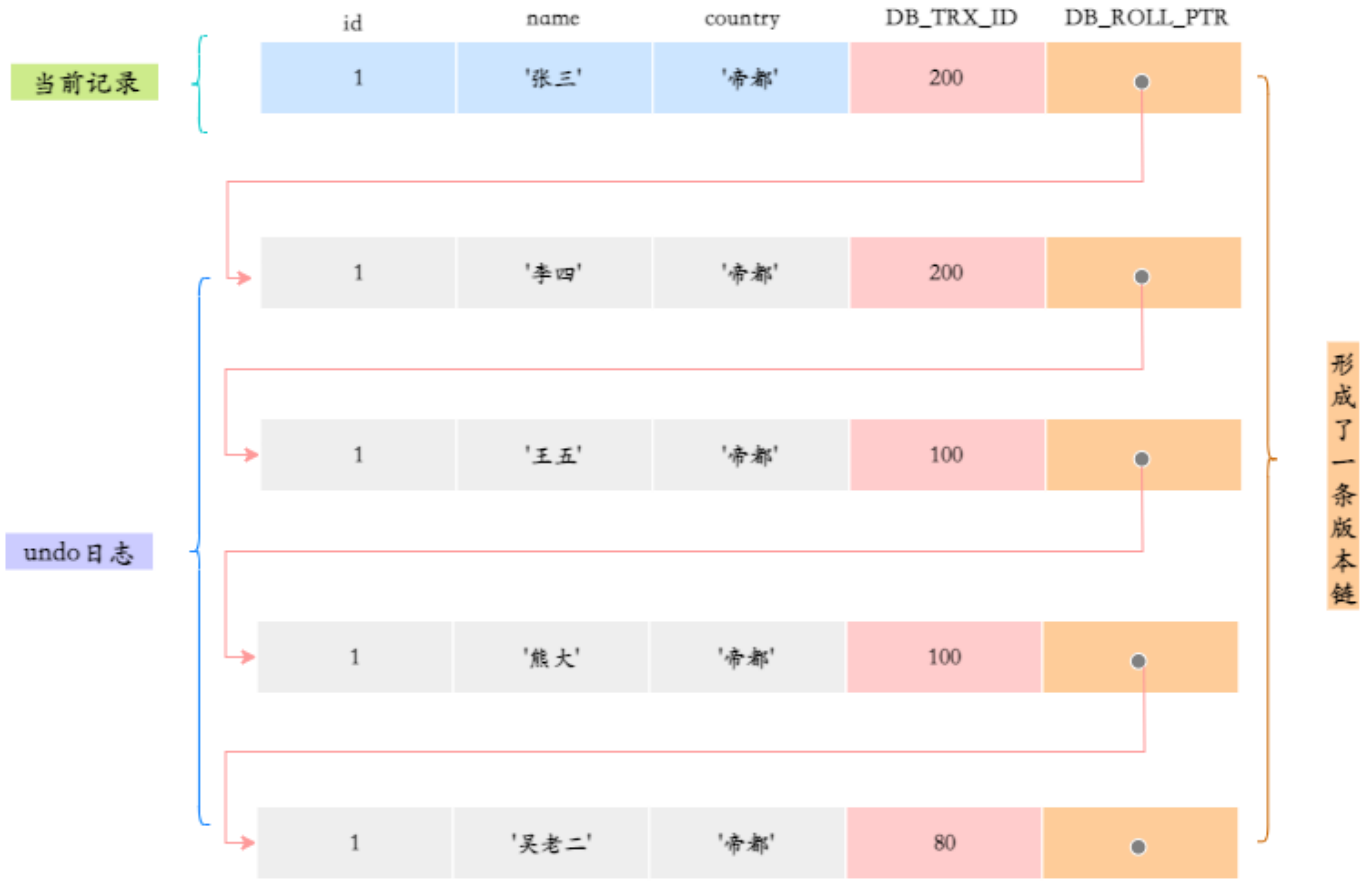
假如有一张`user`表，表中只有一行记录，当时插入的事务 id 为 80。此时，该条记录的示例图如下：



接下来有两个`DB_TRX_ID`分别为`100`、`200`的事务对这条记录进行`update`操作，整个过程如下：

发生时间编号	TRX 100	TRX 200
①	BEGIN	
②		BEGIN
③		
④	UPDATE hero SET name='李四' WHERE id=1	
⑤	UPDATE hero SET name='王五' WHERE id=1	
⑥	COMMIT	UPDATE hero SET name='熊大' WHERE id=1
⑦		UPDATE hero SET name='吴老二' WHERE id=1
⑧		COMMIT

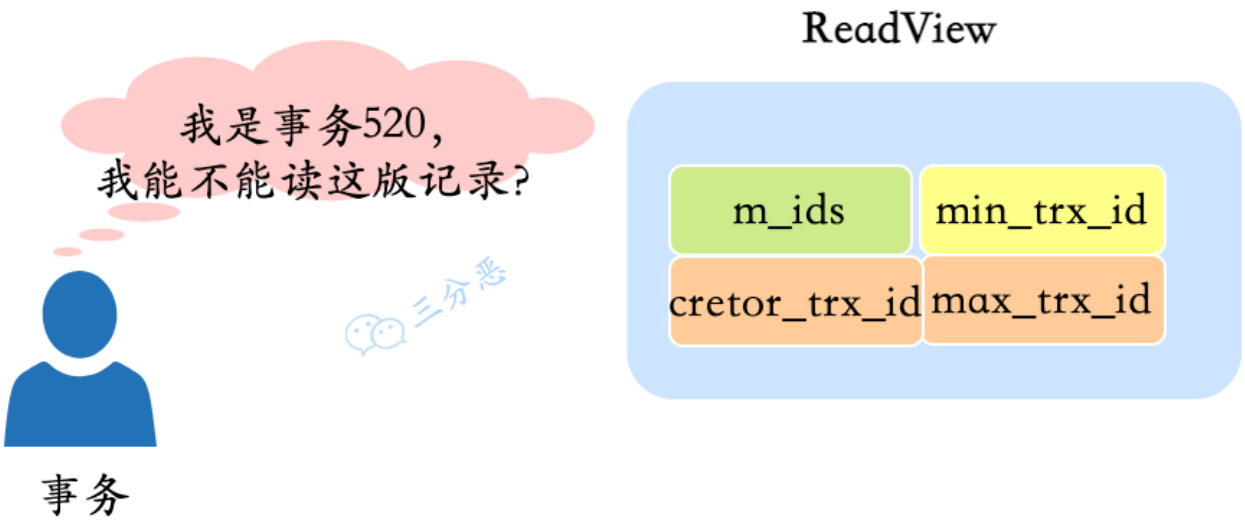
由于每次变动都会先把 undo 日志记录下来，并用 DB_ROLL_PTR 指向 undo 日志地址。因此可以认为，对该条记录的修改日志串联起来就形成了一个 版本链，版本链的头节点就是当前记录最新的值。如下：



ReadView

> 对于 **Read Committed** 和 **Repeatable Read** 隔离级别来说，都需要读取已经提交的事务所修改的记录，也就是说如果版本链中某个版本的修改没有提交，那么该版本的记录时不能被读取的。所以需要确定在 **Read Committed** 和 **Repeatable Read** 隔离级别下，版本链中哪个版本是能被当前事务读取的。于是就引入了 **ReadView** 这个概念来解决这个问题。

Read View 就是事务执行快照读时，产生的读视图，相当于某时刻表记录的一个快照，通过这个快照，我们可以获取：



- 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。
- 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 DB_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 DB_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 DB_TRX_ID 属性值大于 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 DB_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。

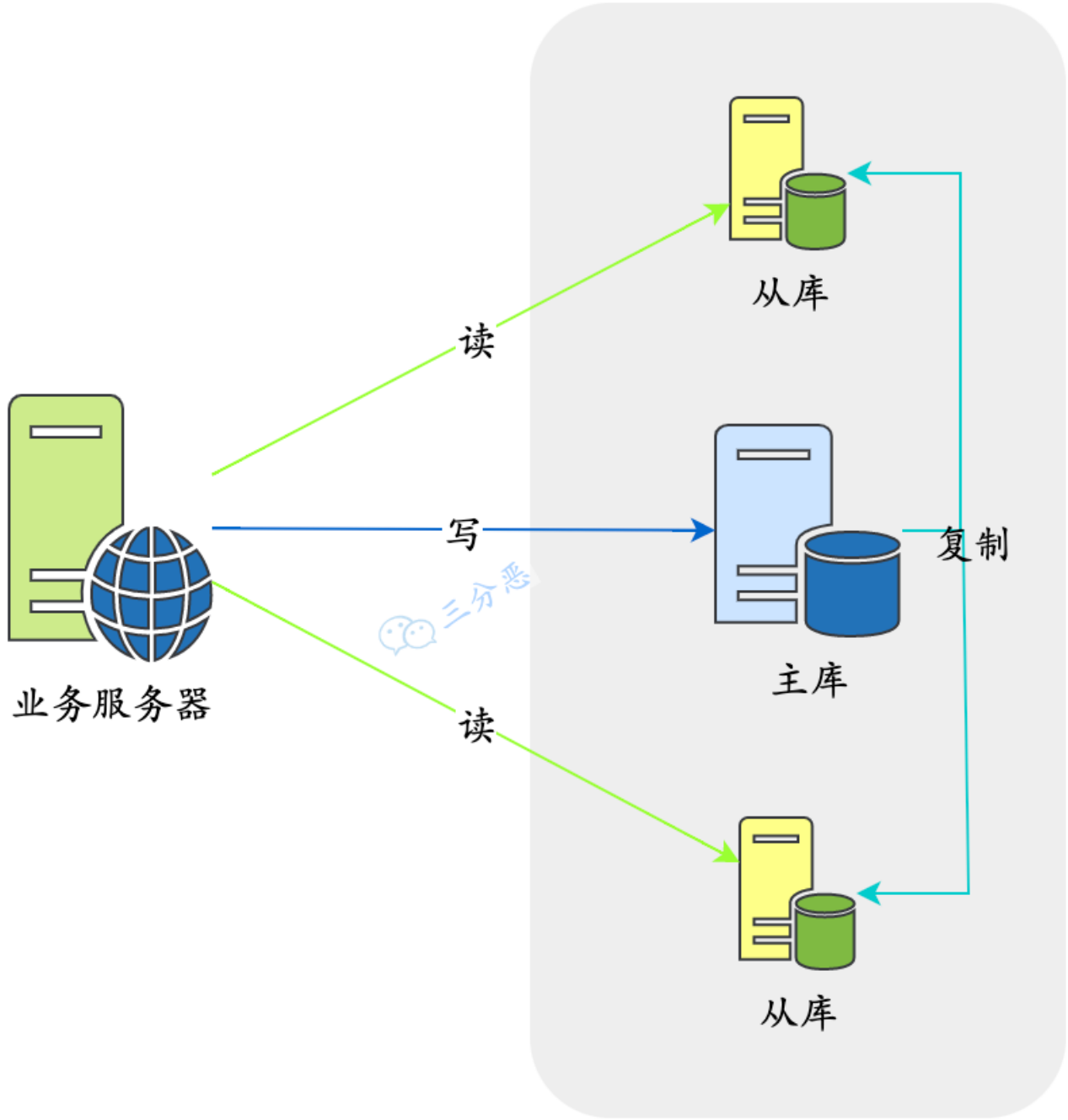
在 MySQL 中， READ COMMITTED 和 REPEATABLE READ 隔离级别的的一个非常大的区别就是它们生成 ReadView 的时机不同。

READ COMMITTED 是每次读取数据前都生成一个 **ReadView**，这样就能保证自己每次都能读到其它事务提交的数据；REPEATABLE READ 是在第一次读取数据时生成一个 **ReadView**，这样就能保证后续读取的结果完全一致。

高可用/性能

54.数据库读写分离了解吗？

读写分离的基本原理是将数据库读写操作分散到不同的节点上，下面是基本架构图：



读写分离的基本实现是:

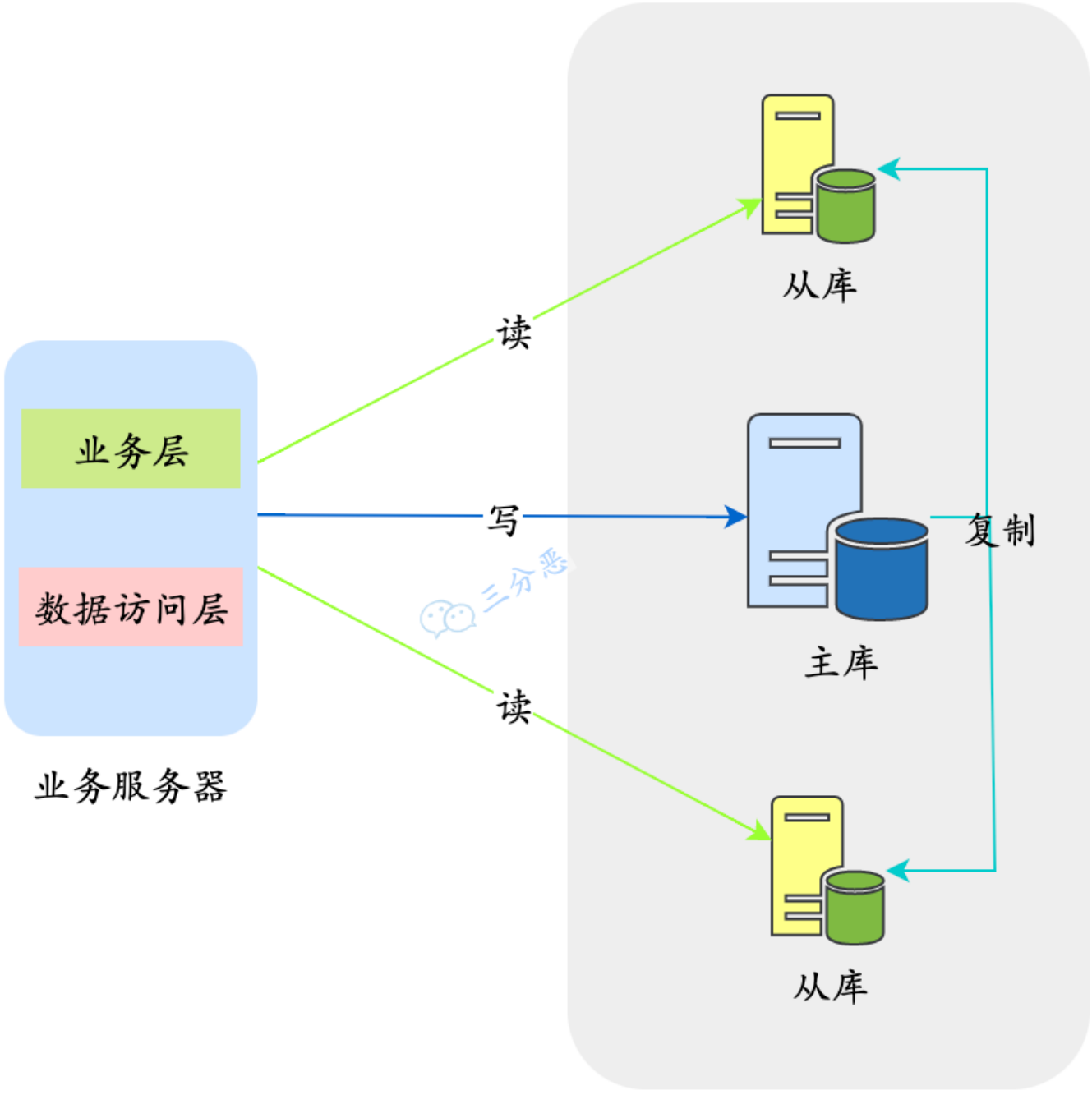
- 数据库服务器搭建主从集群，一主一从、一主多从都可以。
- 数据库主机负责读写操作，从机只负责读操作。
- 数据库主机通过复制将数据同步到从机，每台数据库服务器都存储了所有的业务数据。
- 业务服务器将写操作发给数据库主机，将读操作发给数据库从机。

55.那读写分离的分配怎么实现呢？

将读写操作区分开来，然后访问不同的数据库服务器，一般有两种方式：程序代码封装和中间件封装。

1．程序代码封装

程序代码封装指在代码中抽象一个数据访问层（所以有的文章也称这种方式为"中间层封装"），实现读写操作分离和数据库服务器连接的管理。例如，基于 Hibernate 进行简单封装，就可以实现读写分离：



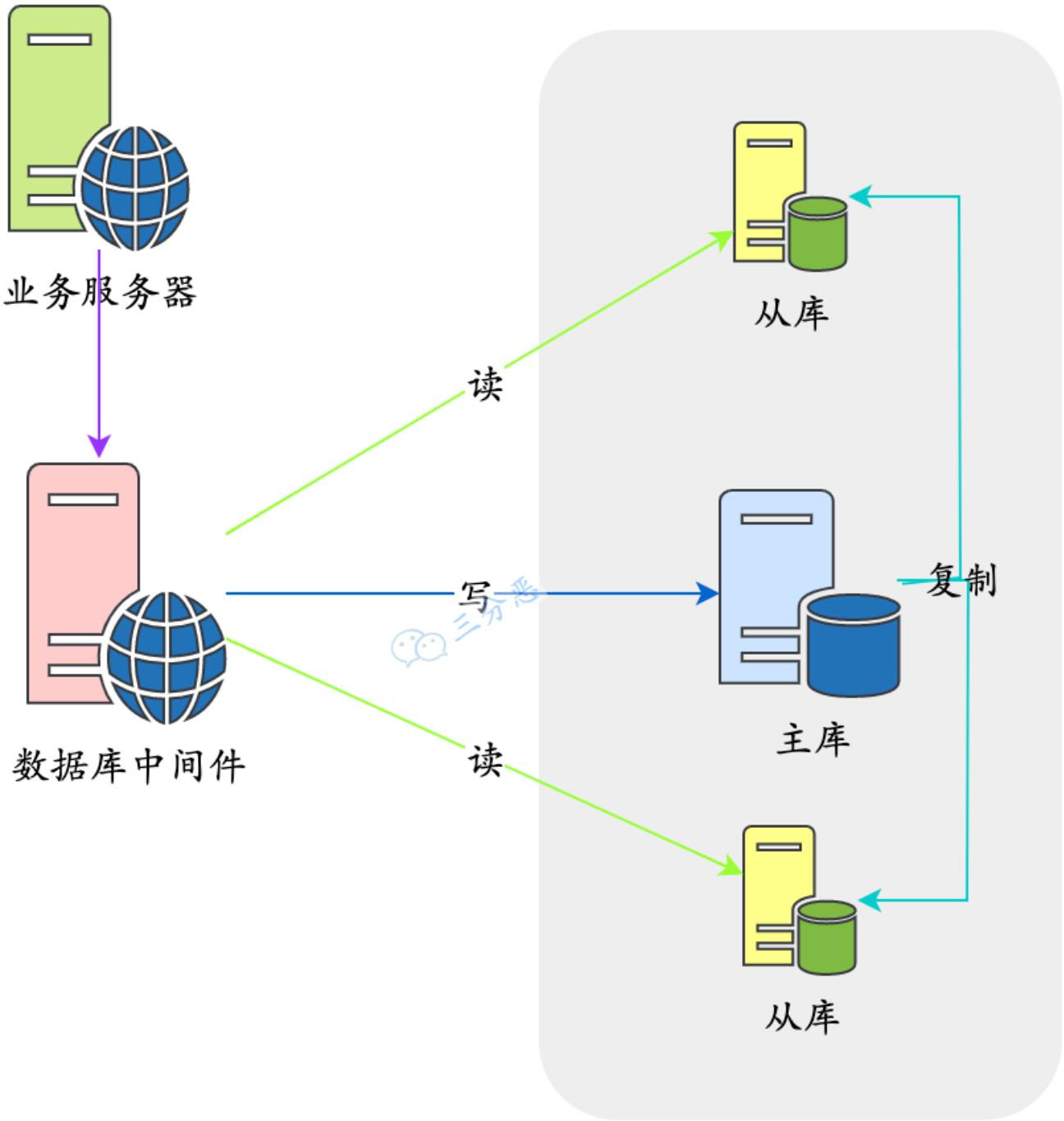
目前开源的实现方案中，淘宝的 TDDL (Taobao Distributed Data Layer, 外号：头都大了）是比较有名的。

2．中间件封装

中间件封装指的是独立一套系统出来，实现读写操作分离和数据库服务器连接的管理。中间件对业务服务器提供 SQL 兼容的协议，业务服务器无须自己进行读写分离。

对于业务服务器来说，访问中间件和访问数据库没有区别，事实上在业务服务器看来，中间件就是一个数据库服务器。

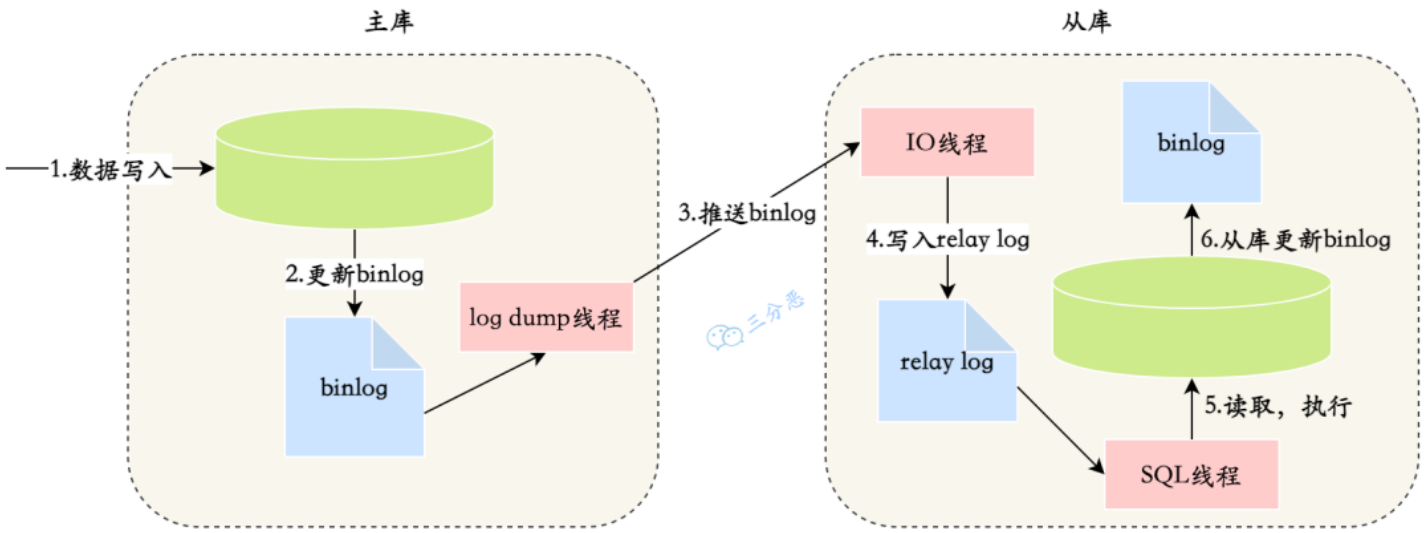
其基本架构是：



56.主从复制原理了解吗?

- master 数据写入，更新 binlog
- master 创建一个 dump 线程向 slave 推送 binlog
- slave 连接到 master 的时候，会创建一个 IO 线程接收 binlog，并记录到 relay log 中继日志中
- slave 再开启一个 sql 线程读取 relay log 事件并在 slave 执行，完成同步

- slave 记录自己的 binglog



57.主从同步延迟怎么处理?

主从同步延迟的原因

一个服务器开放N个链接给客户端来连接的，这样有会有大并发的更新操作，但是从服务器的里面读取binlog 的线程仅有一个，当某个 SQL 在从服务器上执行的时间稍长 或者由于某个 SQL 要进行锁表就会导致，主服务器的 SQL 大量积压，未被同步到从服务器里。这就导致了主从不一致， 也就是主从延迟。

主从同步延迟的解决办法

解决主从复制延迟有几种常见的方法:

1. 写操作后的读操作指定发给数据库主服务器

例如，注册账号完成后，登录时读取账号的读操作也发给数据库主服务器。这种方式和业务强绑定，对业务的侵入和影响较大，如果哪个新来的程序员不知道这样写代码，就会导致一个 bug。

2. 读从机失败后再读一次主机

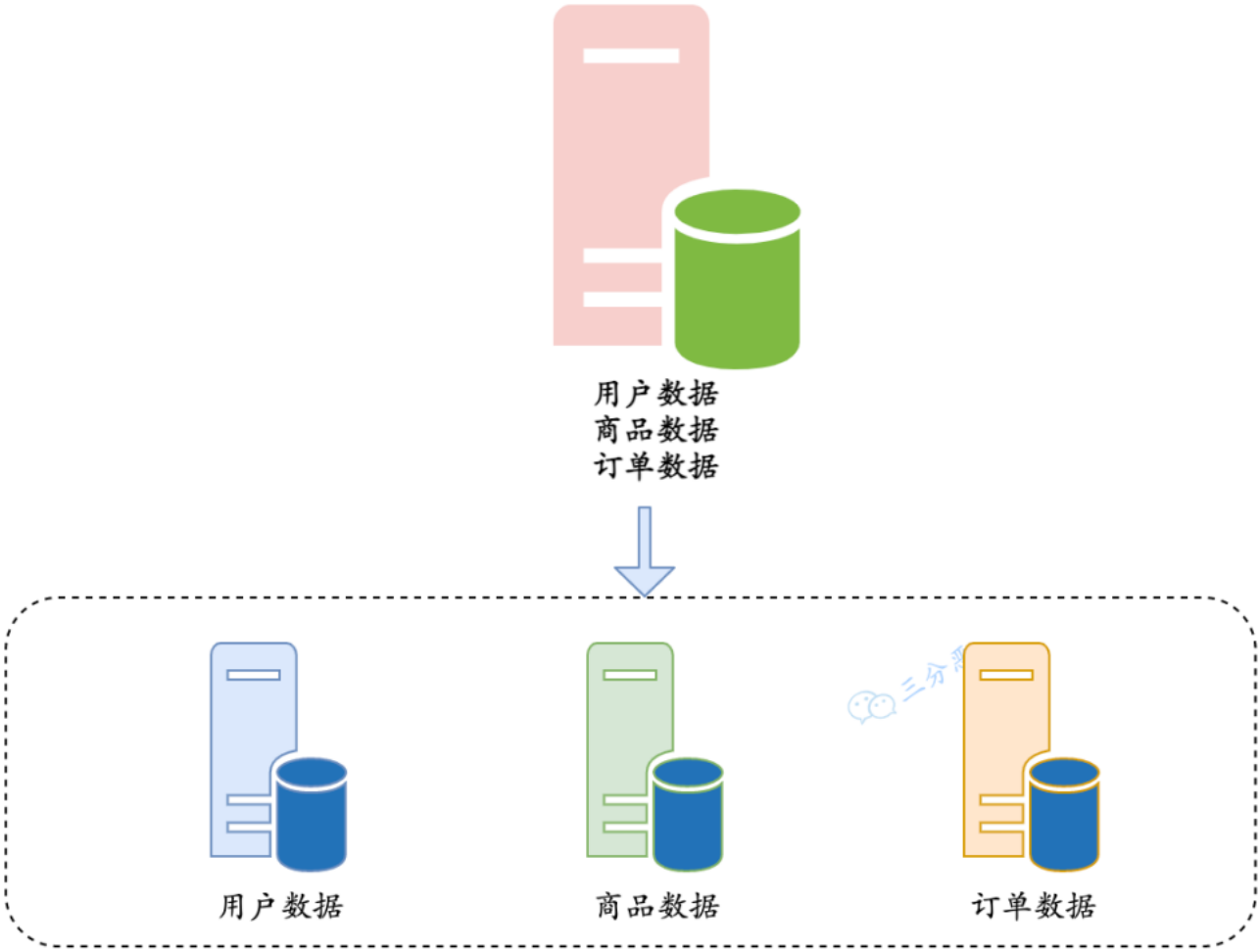
这就是通常所说的 "二次读取"，二次读取和业务无绑定，只需要对底层数据库访问的 API 进行封装即可，实现代价较小，不足之处在于如果有很多二次读取，将大大增加主机的读操作压力。例如，黑客暴力破解账号，会导致大量的二次读取操作，主机可能顶不住读操作的压力从而崩溃。

3. 关键业务读写操作全部指向主机，非关键业务采用读写分离

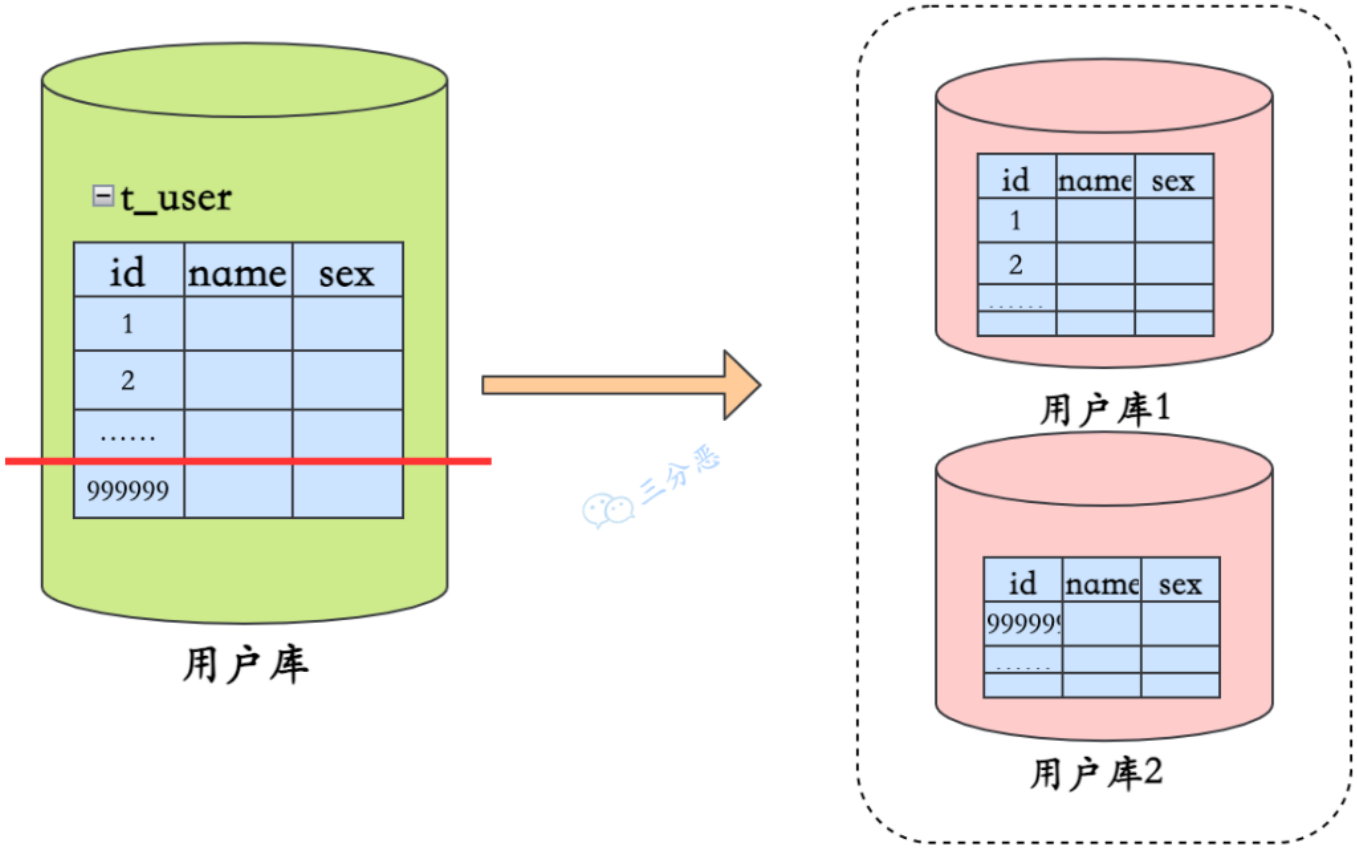
例如，对于一个用户管理系统来说，注册 + 登录的业务读写操作全部访问主机，用户的介绍、爱好、等级等业务，可以采用读写分离，因为即使用户改了自己的自我介绍，在查询时却看到了自我介绍还是旧的，业务影响与不能登录相比就小很多，还可以忍受。

58.你们一般是怎么分库的呢？

- 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。

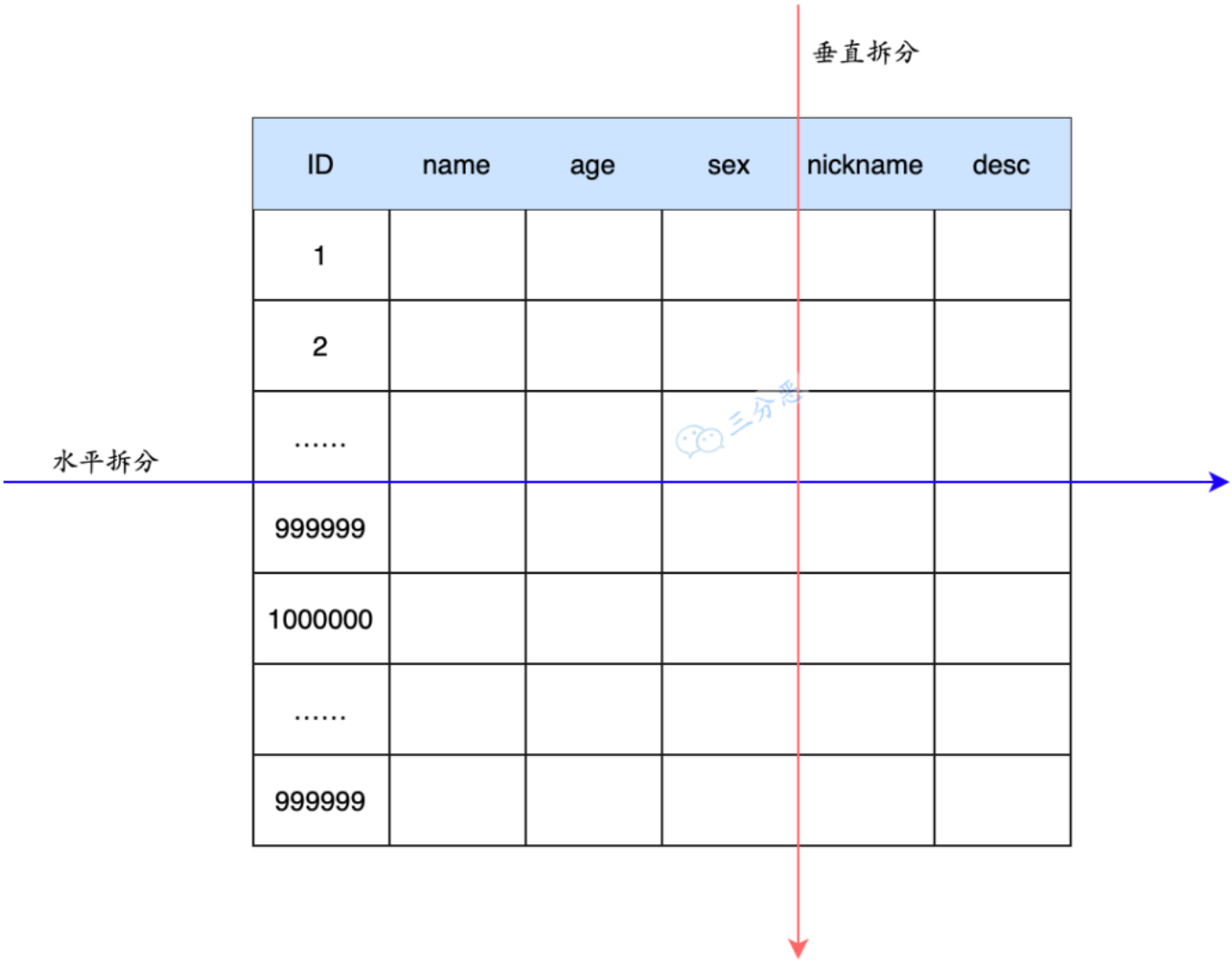


- 水平分库：以字段为依据，按照一定策略（hash、range 等），将一个库中的数据拆分到多个库中。



59.那你们是怎么分表的?

- 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。
- 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。



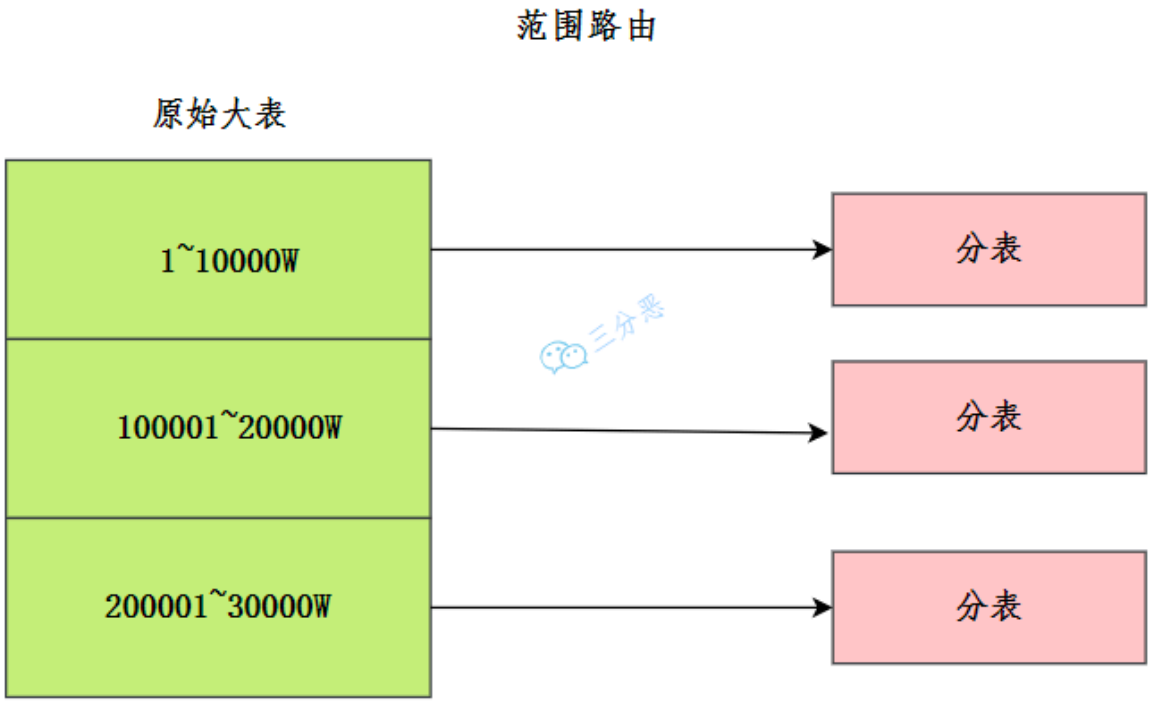
60.水平分表有哪几种路由方式?

什么是路由呢？就是数据应该分到哪一张表。

水平分表主要有三种路由方式：

- **范围路由**：选取有序的数据列 （例如，整形、时间戳等） 作为路由的条件，不同分段分散到不同的数据库表中。

我们可以观察一些支付系统，发现只能查一年范围内的支付记录，这个可能就是支付公司按照时间进行了分表。

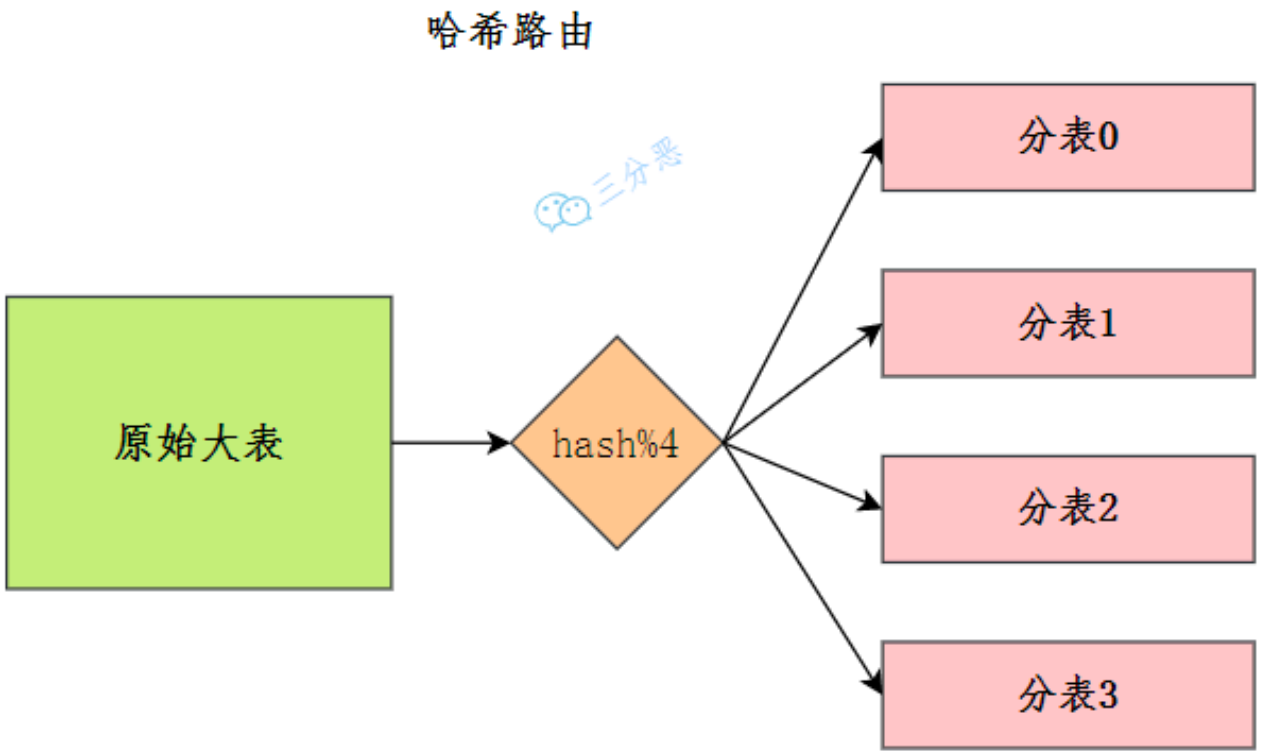


范围路由设计的复杂点主要体现在分段大小的选取上，分段太小会导致切分后子表数量过多，增加维护复杂度；分段太大可能会导致单表依然存在性能问题，一般建议分段大小在 100 万至 2000 万之间，具体需要根据业务选取合适的分段大小。

范围路由的优点是可以随着数据的增加平滑地扩充新的表。例如，现在的用户是 100 万，如果增加到 1000 万，只需要增加新的表就可以了，原有的数据不需要动。范围路由的一个比较隐含的缺点是分布不均匀，假如按照 1000 万来进行分表，有可能某个分段实际存储的数据量只有 1000 条，而另外一个分段实际存储的数据量有 900 万条。

- **Hash 路由**：选取某个列（或者某几个列组合也可以）的值进行 Hash 运算，然后根据 Hash 结果分散到不同的数据库表中。

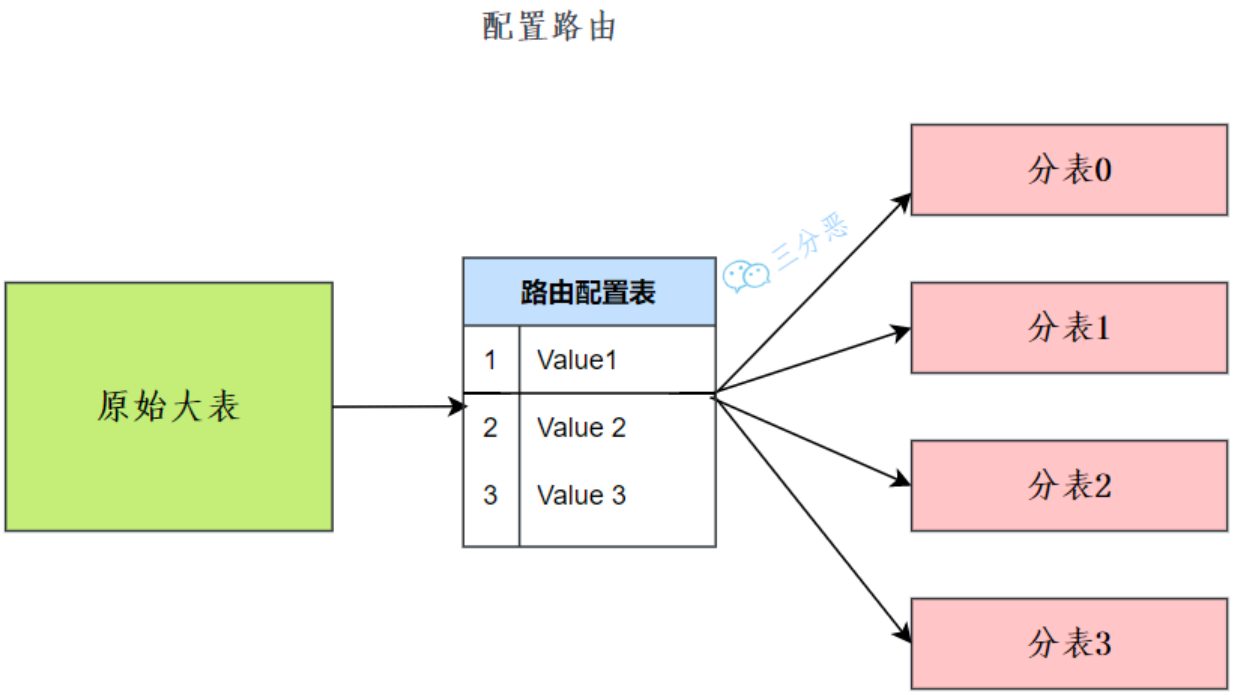
同样以订单 id 为例，假如我们一开始就规划了 4 个数据库表，路由算法可以简单地用 id % 4 的值来表示数据所属的数据库表编号，id 为 12 的订单放到编号为 50 的子表中，id 为 13 的订单放到编号为 61 的字表中。



Hash 路由设计的复杂点主要体现在初始表数量的选取上，表数量太多维护比较麻烦，表数量太少又可能导致单表性能存在问题。而用了 Hash 路由后，增加子表数量是非常麻烦的，所有数据都要重分布。Hash 路由的优缺点和范围路由基本相反，Hash 路由的优点是表分布比较均匀，缺点是扩充新的表很麻烦，所有数据都要重分布。

- **配置路由**：配置路由就是路由表，用一张独立的表来记录路由信息。同样以订单 id 为例，我们新增一张 order_router 表，这个表包含 orderjd 和 tablejd 两列，根据 orderjd 就可以查询对应的 table_id。

配置路由设计简单，使用起来非常灵活，尤其是在扩充表的时候，只需要迁移指定的数据，然后修改路由表就可以了。



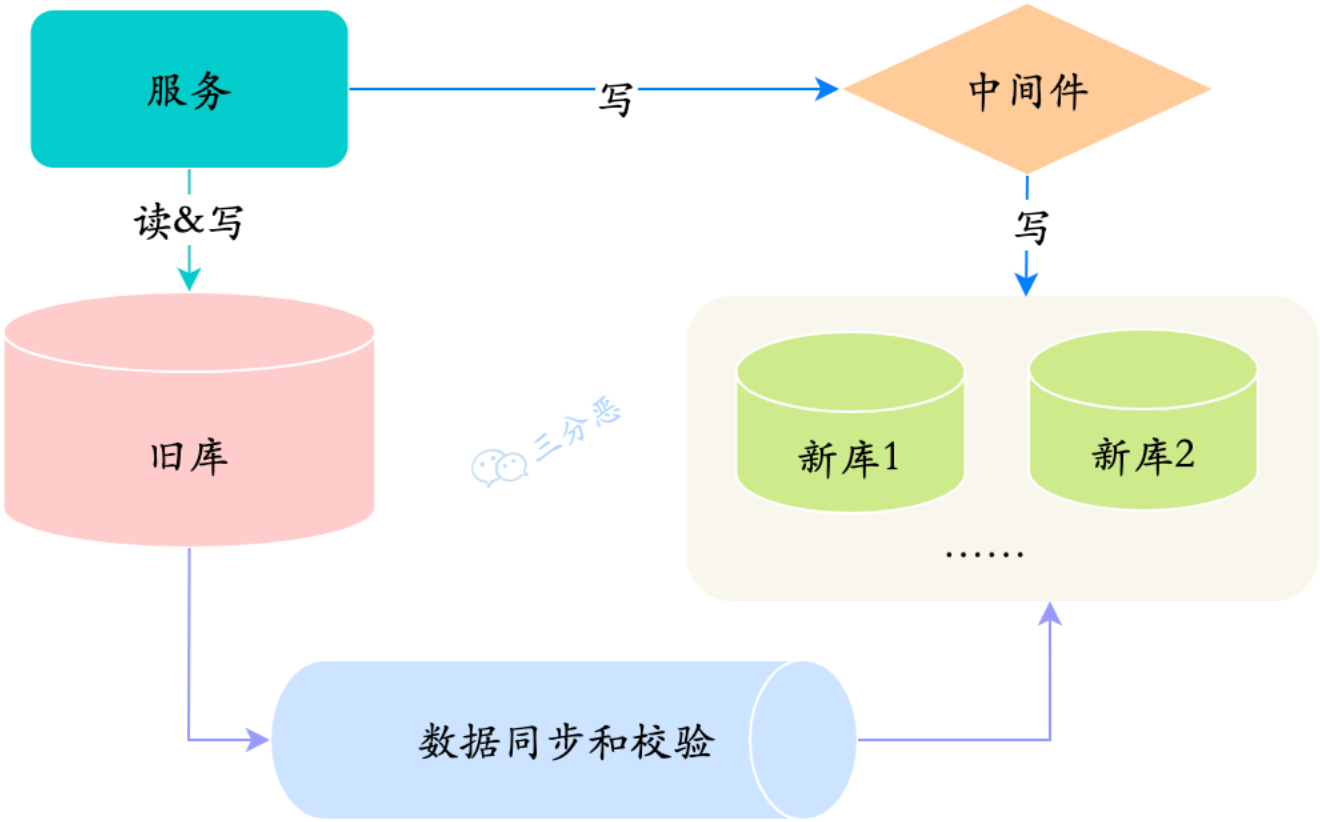
配置路由的缺点就是必须多查询一次，会影响整体性能；而且路由表本身如果太大（例如，几亿条数据），性能同样可能成为瓶颈，如果我们再次将路由表分库分表，则又面临一个死循环式的路由算法选择问题。

61.不停机扩容怎么实现？

实际上，不停机扩容，实操起来是个非常麻烦而且很有风险的操作，当然，面试回答起来就简单很多。

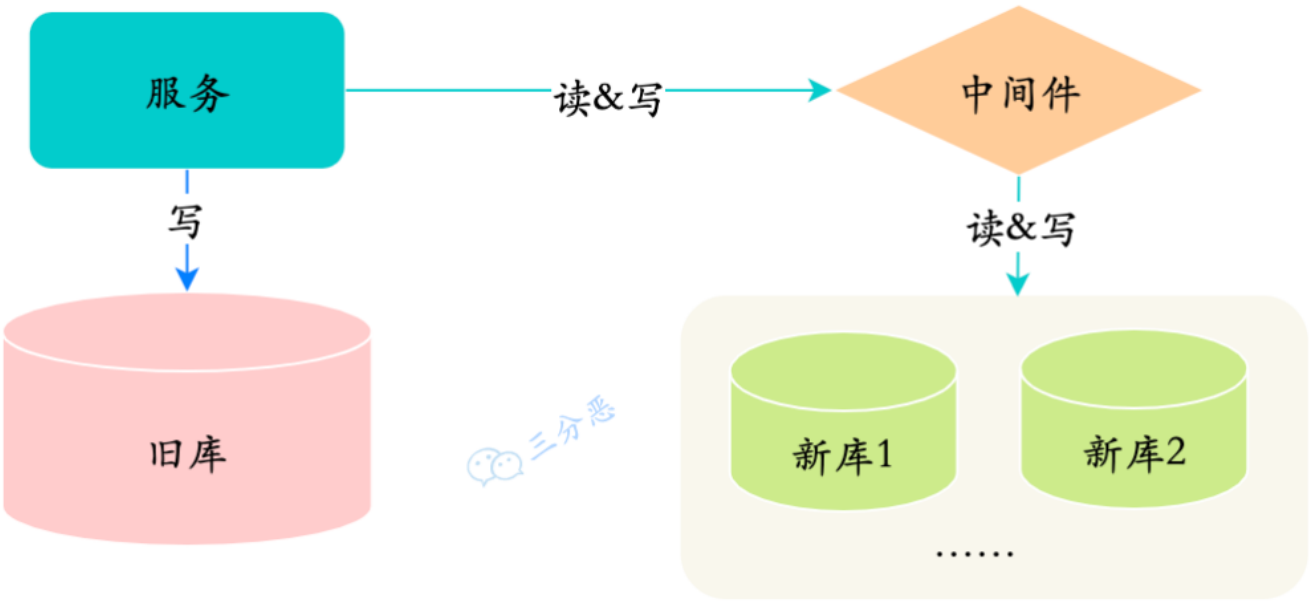
- **第一阶段：在线双写，查询走老库**

1. 建立好新的库表结构，数据写入久库的同时，也写入拆分的新库
2. 数据迁移，使用数据迁移程序，将旧库中的历史数据迁移到新库
3. 使用定时任务，新旧库的数据对比，把差异补齐



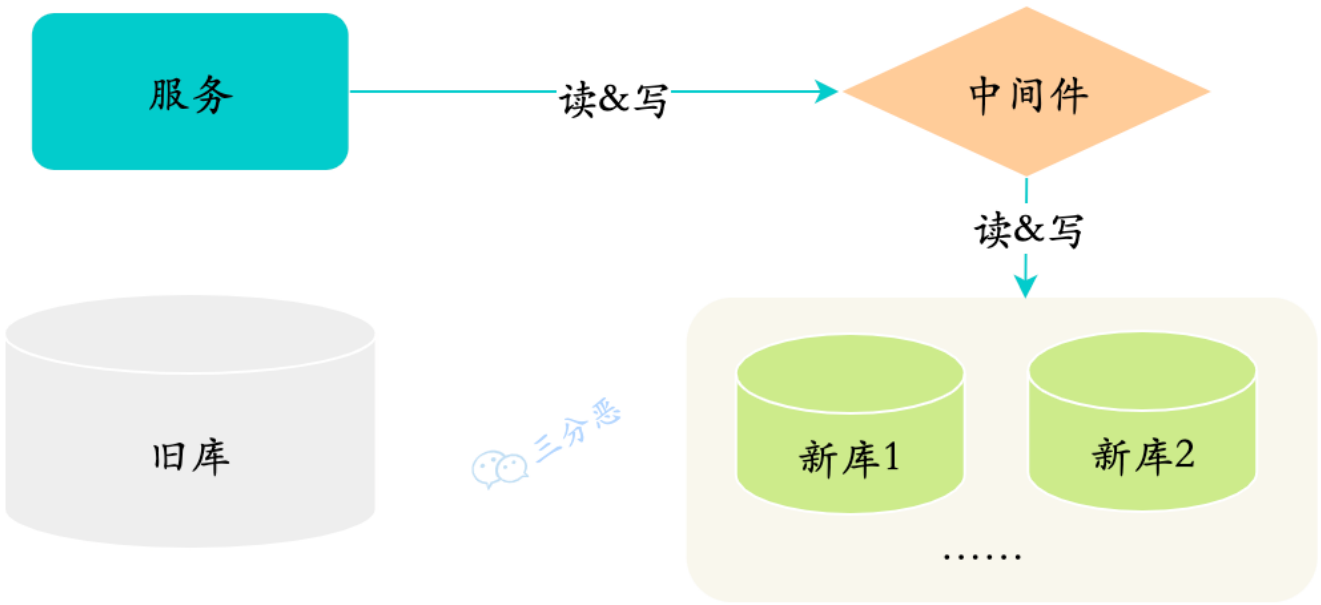
- 第二阶段：在线双写，查询走新库

1. 完成了历史数据的同步和校验
2. 把对数据的读切换到新库



- 第三阶段：旧库下线

1. 旧库不再写入新的数据
2. 经过一段时间，确定旧库没有请求之后，就可以下线老库



62.常用的分库分表中间件有哪些?

- sharding-jdbc
- Mycat

63.那你觉得分库分表会带来什么问题呢?

从分库的角度来讲:

- **事务的问题**

使用关系型数据库，有很大一点在于它保证事务完整性。

而分库之后单机事务就用不上了，必须使用分布式事务来解决。

- **跨库 JOIN 问题**

在一个库中的时候我们还可以利用 JOIN 来连表查询，而跨库了之后就无法使用 JOIN 了。

此时的解决方案就是**在业务代码中进行关联**，也就是先把一个表的数据查出来，然后通过得到的结果再去查另一张表，然后利用代码来关联得到最终的结果。

这种方式实现起来稍微比较复杂，不过也是可以接受的。

还有可以**适当的冗余一些字段**。比如以前的表就存储一个关联 ID，但是业务时常要求返回对应的 Name 或者其他字段。这时候就可以把这些字段冗余到当前表中，来去除需要关联的操作。

还有一种方式就是**数据异构**，通过 binlog 同步等方式，把需要跨库 join 的数据异构到 ES 等存储结构中，通过 ES 进行查询。

从分表的角度来看:

- **跨节点的 count,order by,group by 以及聚合函数问题**

只能由业务代码来实现或者用中间件将各表中的数据汇总、排序、分页然后返回。

- **数据迁移，容量规划，扩容等问题**

数据的迁移，容量如何规划，未来是否可能再次需要扩容，等等，都是需要考虑的问题。

- **ID 问题**

数据库表被切分后，不能再依赖数据库自身的主键生成机制，所以需要一些手段来保证全局主键唯一。

1. 还是自增，只不过自增步长设置一下。比如现在有三张表，步长设置为 3，三张表 ID 初始值分别是 1、2、3。这样第一张表的 ID 增长是 1、4、7。第二张表是 2、5、8。第三张表是 3、6、9，这样就不会重复了。

2. UUID，这种最简单，但是不连续的主键插入会导致严重的页分裂，性能比较差。
3. 分布式 ID，比较出名的就是 Twitter 开源的 sonwflake 雪花算法

运维

64.百万级别以上的数据如何删除？

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。

1. 所以我们想要删除百万数据的时候可以先删除索引
2. 然后删除其中无用数据
3. 删除完成后重新创建索引创建索引也非常快

65.百万千万级大表如何添加字段？

当线上的数据库数据量到达几百万、上千万的时候，加一个字段就没那么简单，因为可能会长时间锁表。

大表添加字段，通常有这些做法：

- 通过中间表转换过去

创建一个临时的新表，把旧表的结构完全复制过去，添加字段，再把旧表数据复制过去，删除旧表，新表命名为旧表的名称，这种方式可能回丢掉一些数据。

- 用 pt-online-schema-change

`pt-online-schema-change` 是 percona 公司开发的一个工具，它可以在线修改表结构，它的原理也是通过中间表。

- 先在从库添加 再进行主从切换

如果一张表数据量大且是热表（读写特别频繁），则可以考虑先在从库添加，再进行主从切换，切换后再将其他几个节点上添加字段。

66.MySQL 数据库 cpu 飙升的话，要怎么处理呢？

排查过程：

（1）使用 top 命令观察，确定是 mysqld 导致还是其他原因。

（2）如果是 mysqld 导致的，show processlist，查看 session 情况，确定是不是有消耗资源的 sql 在运行。

（3）找出消耗高的 sql，看看执行计划是否准确， 索引是否缺失，数据量是否太大。

处理：

（1）kill 掉这些线程 (同时观察 cpu 使用率是否下降)，

（2）进行相应的调整 (比如说加索引、改 sql、改内存参数)

（3）重新跑这些 SQL。

其他情况：

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

没有什么使我停留——除了目的，纵然岸旁有玫瑰、有绿荫、有宁静的港湾，我是不系之舟。

系列内容：

- 面渣逆袭 Java SE 篇👍
- 面渣逆袭 Java 集合框架篇👍
- 面渣逆袭 Java 并发编程篇👍
- 面渣逆袭 JVM 篇👍
- 面渣逆袭 Spring 篇👍

- 面渣逆袭 Redis 篇👍
- 面渣逆袭 MyBatis 篇👍
- 面渣逆袭 MySQL 篇👍
- 面渣逆袭操作系统篇👍
- 面渣逆袭计算机网络篇👍

图文详解 66 道 MySQL 面试高频题，这次吊打面试官，我觉得稳了（手动 dog）。整理：沉默王二，戳转载链接，作者：三分恶，戳原文链接。