(二)全解MySQL：一条SQL语句从诞生至结束的多姿多彩历程！

竹子爱熊猫 LV.5
2022年09月19日 22:47 · 阅读 3609

已关注

引言

> 本文为掘金社区首发签约文章，14天内禁止转载，14天后未获授权禁止转载，侵权必究！

在上篇文章中，我们以《MySQL架构篇》拉开了MySQL数据库的的序幕，上篇文章中将MySQL分层架构中的每一层都进行了详细阐述。而在本篇中，则会进一步站在一条SQL的角度，从SQL的诞生开始，到SQL执行、数据返回等全链路进行分析。

在此之前呢，其实也写过两篇类似的姊妹篇，第一篇讲的是《一个网络请求的神奇之旅！》，在其中化身一个网络请求，切身感受了从浏览器发出后，到响应数据的返回。而在更早的时候，《JVM系列》中也有一篇文章，讲的是《一个Java对象从诞生到死亡的历程》，其中聊到了Java对象是如何诞生的，运行时会经历什么过程？使用结束后又是如何回收的。

> 在本篇中，则会在站在数据库的视角，再次感受“一条SQL多姿多彩的历程”！你如果认真的看完了“一个请求、一个对象、一条SQL”这三部曲后，相信你对于程序开发又会有一个全新的深刻认知。

78 34 收藏

SQL语句都诞生于客户端，主要有两种方式产生一条SQL，一种是由开发者自己手动编写，另一种则是相关的ORM框架自动生成，一般情况下，MySQL运行过程中收到的大部分SQL都是由ORM框架生成的，比如Java中的MyBatis、Hibernate框架等。

同时，SQL生成的时机一般都与用户的请求有关，当用户在系统中进行了某项操作，一般都会产生一条SQL，例如我们在浏览器上输入如下网址：

> `https://juejin.cn/user/862486453028888/posts`

此时，就会先请求掘金的服务器，然后由掘金内部实现中的ORM框架，根据请求参数生成一条SQL，类似于下述的伪SQL：

> `select * from juejin_article where userid = 862486453028888;`

这条SQL大致描述的意思就是：根据用户请求的「作者ID」，在掘金数据库的文章表中，查询该作者的所有文章信息。

从上述这个案例中可以明显感受出来，用户浏览器上看到的数据一般都来自于数据库，而数据库执行的SQL则源自于用户操作，两者是相辅相成的关系，也包括任何写操作（增、删、改），本质上也会被转换一条条SQL，也举个简单的例子：

```json
# 请求网址（Request URL）
https://www.xxx.com/user/register

# 请求参数（Request Param）
{
    user_name : "竹子爱熊猫",
    user_pwd : "123456",
    user_sex : "男",
    user_phone : "18888888888",
```

比如上述这个用户注册的案例，当用户在网页上点击「注册」按钮时，会向目标网站的服务器发送一个 post 请求，紧接着同样会根据请求参数，生成一条 SQL，如下：

```sql 复制代码
insert into table_user (user_name,user_pwd,user_sex,user_phone,....)
    VALUES ("竹子爱熊猫", "123456", "男", "18888888888", ....);
```

也就是说，一条 SQL 的诞生都源自于一个用户请求，在开发程序时，SQL 的大体逻辑我们都会由业务层的编码决定，具体的 SQL 语句则是根据用户的请求参数，以及提前定制好的“SQL 骨架”拼接而成。当然，在 Java 程序或其他语言编写的程序中，只能生成 SQL，而 SQL 真正的执行工作是需要交给数据库去完成的。

二、一条SQL执行前会经历的过程

经过上一步之后，一条完整的 SQL 就诞生了，为了 SQL 能够正常执行，首先会先去获取一个数据库连接对象，上篇关于 MySQL 的架构篇曾聊到过，MySQL 连接层中会维护着一个名为「连接池」的玩意儿，但相信大家也都接触过「数据库连接池」这个东西，比如Java中的 C3P0、Druid、DBCP.... 等各类连接池。

> 那此时在这里可以思考一个问题，为什么数据库自己维护了连接池的情况下，在 MySQL 客户端中还需要再次维护一个数据库连接池呢？接下来一起聊一聊。

2.1、数据库连接池的必要性

众所周知，当要在 Java 中创建一个数据库连接时，首先会去读取配置文件中的连接地址、账号密码等信息，然后根据配置的地址信息，发起网络请求获取数据库连接对象。在这个过程中，由于涉及到了网络请求，那此时必然会先经历 TCP 三次握手的过程，同时获取到连接对象完成 SQL 操作后，又要释放这个数据库连接，此时又需要经历 TCP 四次挥手过程。

> 从上面的描述中可以明显感知出，在Java中创建、关闭数据库连接的过程，过程开销其实比较大，而在程序上线后，又需要频繁进行数据库操作。因此如果每次操作数据库时，都获取新的连接对象，那整个 `Java` 程序至少会有四分之一的时间内在做 `TCP` 三次握手/四次挥手工作，这对整个系统造成的后果可想而知....

也正是由于上述原因，因此大名鼎鼎的「数据库连接池」登场了，「数据库连接池」和「线程池」的思想相同，会将数据库连接这种较为珍贵的资源，利用池化技术对这种资源进行维护。也就代表着之后需要进行数据库操作时，不需要自己去建立连接了，而是直接从「数据库连接池」中获取，用完之后再归还给连接池，以此达到复用的效果。

> 当然，连接池中维护的连接对象也不会一直都在，当长时间未进行 `SQL` 操作时，连接池也会销毁这些连接对象，而后当需要时再次创建，不过何时创建、何时销毁、连接数限制等等这些工作，都交给了连接池去完成，无需开发者自身再去关注。

在Java中，目前最常用的数据库连接池就是阿里的 `Druid` ，一般咱们都会用它作为生产环境中的连接池：

> 目前 Druid 已经被阿里贡献给 Apache 软件基金会维护了~

OK~，回到前面抛出的问题，有了 MySQL 连接池为何还需要在客户端维护一个连接池？

> 对于这个问题，相信大家在心里多少都有点答案了，原因很简单，两者都是利用池化技术去达到复用资源、节省开销、提升性能的目的，只不过针对的方向不同。

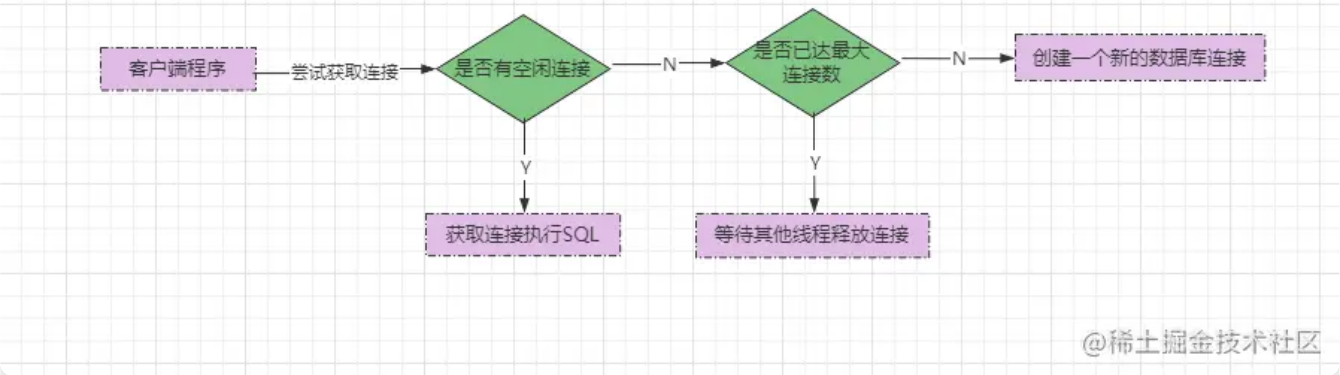
MySQL 的连接池主要是为了实现复用线程的目的，因为每个数据库连接在 MySQL 中都会使用一条线程维护，而每次为客户端分配连接对象时，都需要经历创建线程、分配栈空间....这些繁重的工作，这个过程需要时间，同时资源开销也不小，所以 MySQL 利用池化技术解决了这些问题。

而客户端的连接池，主要是为了实现复用数据库连接的目的，因为每次 SQL 操作都需要经过 TCP 三次握手/四次挥手的过程，过程同样耗时且占用资源，因此也利用池化技术解决了这个问题。

> 其实也可以这样理解， MySQL 连接池维护的是工作线程，客户端连接池则维护的是网络连接。

2.2、SQL执行前会发生的事情

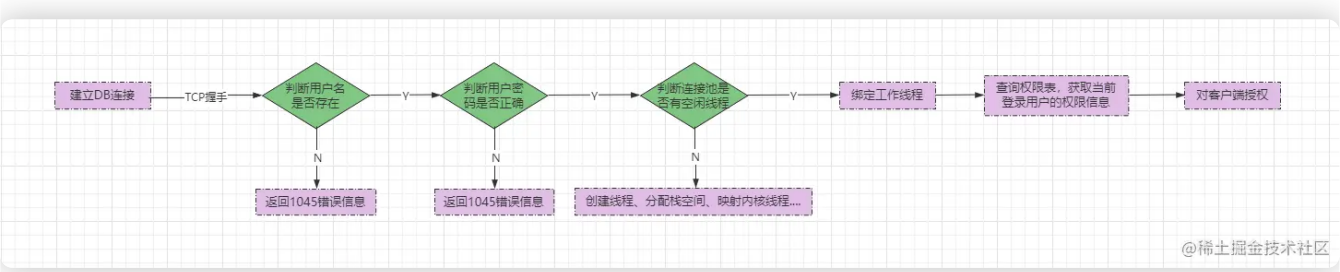
回归本文主题，当完整的 SQL 生成后，会先去连接池中尝试获取一个连接对象，那接下来会发生什么事情呢？如下图：



当尝试从连接池中获取连接时，如果此时连接池中有空闲连接，可以直接拿到复用，但如果没有，则要先判断一下当前池中的连接数是否已达到最大连接数，如果连接数已经满了，当前线程则需要等待其他线程释放连接对象，没满则可以直接再创建一个新的数据库连接使用。

> 假设此时连接池中没有空闲连接，需要再次创建一个新连接，那么就会先发起网络请求建立连接。

首先会经过《TCP的三次握手过程》，对于这块就不再细聊了，毕竟之前聊过很多次了。当网络连接建立成功后，也就等价于在`MySQL`中创建了一个客户端会话，然后会发生下图一系列工作：



- ①首先会验证客户端的用户名和密码是否正确：
  - 如果用户名不存在或密码错误，则抛出`1045`的错误码及错误信息。
  - 如果用户名和密码验证通过，则进入第②步。
- ②判断`MySQL`连接池中是否存在空闲线程：
  - 存在：直接从连接池中分配一条空闲线程维护当前客户端的连接。
  - 不存在：创建一条新的工作线程（映射内核线程、分配栈空间....）。
- ③工作线程会先查询`MySQL`自身的用户权限表，获取当前登录用户的权限信息并授权。

三、一条SQL语句在数据库中是如何执行的?

经过连接层的一系列工作后，接着客户端会将要执行的 SQL 语句通过连接发送过来，然后会进行 MySQL 服务层进行处理，不过根据用户的操作不同， MySQL 执行 SQL 语句时也会存在些许差异，这里是指读操作和写操作，两者 SQL 的执行过程并不相同，下面先来看看 select 语句的执行过程。

3.1、一条查询SQL的执行过程

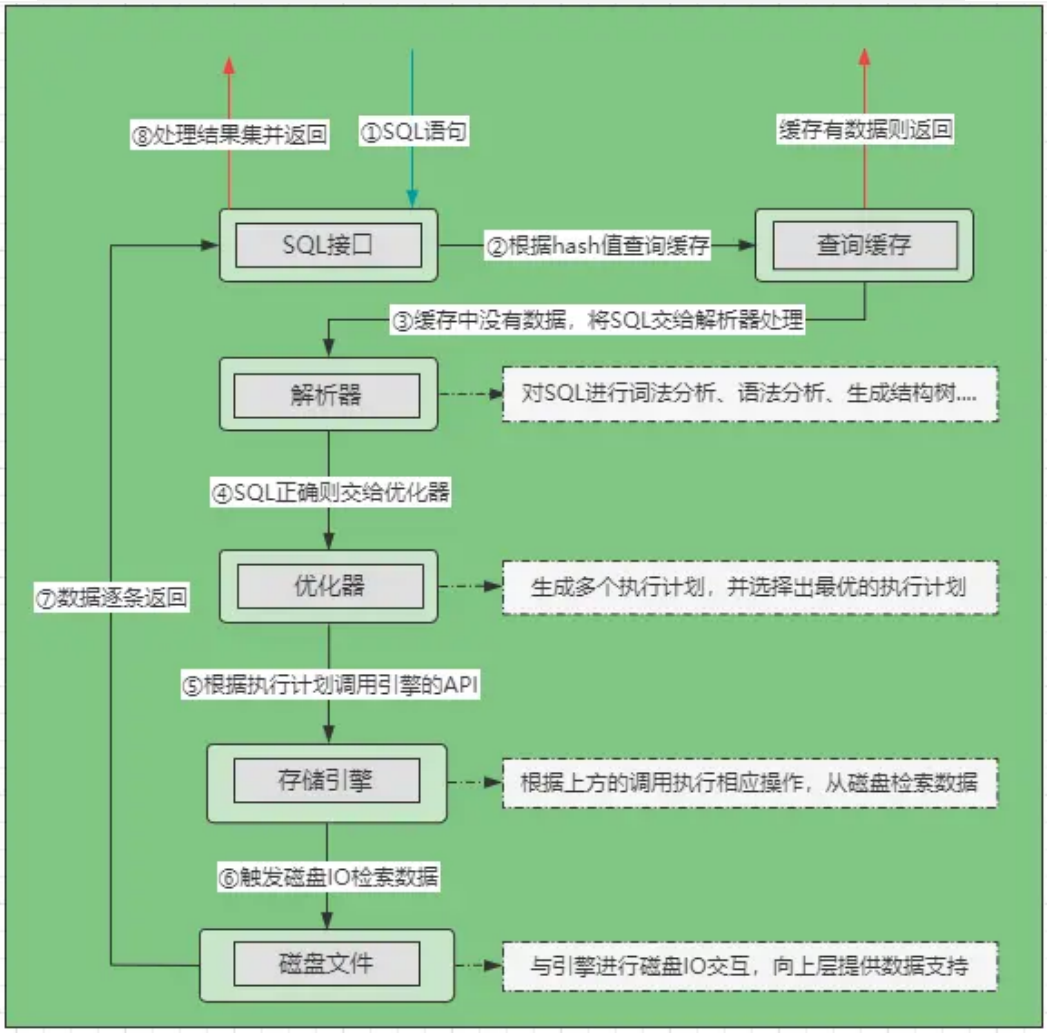
在分析查询 SQL 的执行流程之前，咱们先模拟一个案例，以便于后续分析：

sql 复制代码

```sql
-- SQL语句
SELECT user_id FROM `zz_user` WHERE user_sex = "男" AND user_name = "竹子④号";

-- 表数据
+---------+-------------+----------+-------------+
| user_id | user_name   | user_sex | user_phone  |
+---------+-------------+----------+-------------+
|       1 | 竹子①号     | 男       | 18888888888 |
|       2 | 竹子②号     | 男       | 13588888888 |
|       3 | 竹子③号     | 男       | 15688888888 |
|       4 | 熊猫①号     | 女       | 13488888888 |
|       5 | 熊猫②号     | 女       | 18588888888 |
|       6 | 竹子④号     | 男       | 17777777777 |
|       7 | 熊猫③号     | 女       | 16666666666 |
+---------+-------------+----------+-------------+
```

先上个 SQL 执行的完整流程图，后续再逐步分析每个过程：



- ①先将 SQL 发送给 SQL 接口， SQL 接口会对 SQL 语句进行哈希处理。
- ② SQL 接口在缓存中根据哈希值检索数据，如果缓存中有则直接返回数据。
- ③缓存中未命中时会将 SQL 交给解析器，解析器会判断 SQL 语句是否正确：
  - 错误：抛出 1064 错误码及相关的语法错误信息。
  - 正确：将 SQL 语句交给优化器处理，进入第④步。
- ④优化器根据 SQL 制定出不同的执行方案，并择选出最优的执行计划。
- ⑤工作线程根据执行计划，调用存储引擎所提供的 API 获取数据。
- ⑥存储引擎根据 API 调用方的操作，去磁盘中检索数据（索引、表数据....）。
- ⑦发送磁盘 IO 后，对于磁盘中符合要求的数据逐条返回给 SQL 接口。
- ⑧ SQL 接口会对所有的结果集进行处理（剔除列、合并数据....）并返回。

上述是一个简单的流程概述，一般情况下查询 SQL 的执行都会经过这些步骤，下面再将每一步拆开详细聊一聊。

当客户端将 SQL 发送过来之后，SQL 紧接着会交给 SQL 接口处理，首先会对 SQL 做哈希处理，也就是根据 SQL 语句计算出一个哈希值，然后去「查询缓存」中比对，如果缓存中存在相同的哈希值，则代表着之前缓存过相同 SQL 语句的结果，那此时则直接从缓存中获取结果并响应给客户端。

> 在这里，如果没有从缓存中查询到数据，紧接着会将 SQL 语句交给解析器去处理。

SQL 接口除开对 SQL 进行上述的处理外，后续还会负责处理结果集（稍后分析）。

解析器中会干的工作

解析器收到 SQL 后，会开始检测 SQL 是否正确，也就是做词法分析、语义分析等工作，在这一步，解析器会根据 SQL 语言的语法规则，判断客户端传递的 SQL 语句是否合规，如果不合规就会返回 1064 错误码及错误信息：

> ```
> ERROR 1064 (42000): You have an error in your SQL syntax; check....
> ```

但如果 SQL 语句没有问题，此时就会对 SQL 语句进行关键字分析，也就是根据 SQL 中的 SELECT、UPDATE、DELETE 等关键字，先判断 SQL 语句的操作类型，是读操作还是写操作，然后再根据 FROM 关键字来确定本次 SQL 语句要操作的是哪张表，也会根据 WHERE 关键字后面的内容，确定本次 SQL 的一些结果筛选条件.....。

>

解析了 SQL 语句中的关键字之后，解析器会根据分析出的关键字信息，生成对应的语法树，然后交给优化器处理。

> 在这一步也就相当于Java中的 .java 源代码变为 .class 字节码的过程，目的就是将 SQL 语句翻译成数据库可以看懂的指令。

优化器中会干的工作

经过解析器的工作后会得到一个 SQL 语法树，也就是知道了客户端的 SQL 大体要干什么事情了，接着优化器会对于这条 SQL ，给出一个最优的执行方案，也就是告诉工作线程怎么执行效率最高、最节省资源以及时间。

优化器最开始会根据语法树制定出多个执行计划，然后从多个执行计划中选择出一个最好的计划，交给工作线程去执行，但这里究竟是如何选择最优执行计划的，相信大家也比较好奇，那此时我们结合前面给出的案例分析一下。

```sql
SELECT user_id FROM `zz_user` WHERE user_sex = "男" AND user_name = "竹子④号";
```

先来看看，对于这条 SQL 而言，总共有几种执行方案呢？答案是两种。

- ①先从表中将所有 user_sex="男" 的数据查出来，再从结果中获取 user_name="竹子④号" 的数据。
- ②先从表中寻找 user_name="竹子④号" 的数据，再从结果中获得 user_sex="男" 的数据。

再结合前面给出的表数据，暂且分析一下上述两种执行计划哪个更好呢？

```sql
+---------+-------------+----------+-------------+
| user_id | user_name   | user_sex | user_phone  |
```

```
|       5 | 熊猫②号      | 女       | 18588888888 |
|       6 | 竹子④号      | 男       | 17777777777 |
|       7 | 熊猫③号      | 女       | 16666666666 |
+---------+--------------+----------+-------------+
```

如果按照第①种方案执行，此时会先得到四条 `user_sex="男"` 的数据，然后再从四条数据中查找 `user_name="竹子④号"` 的数据。

如果按照第②中方案执行，此时会直接得到一条 `user_name="竹子④号"` 的数据，然后再判断一下 `user_sex` 是否为"男"，是则直接返回，否则返回空。

> 相较于两种执行方案的过程，前者需要扫一次全表，然后再对结果集逐条判断。而第二种方案扫一次全表后，只需要再判断一次就可以了，很明显可以感知出：第②种执行计划是最优的，因此优化器会给出第②种执行计划。

经过上述案例的讲解后，大家应该能够对优化器的工作进一步理解。但上述案例仅是为了帮助大家理解，实际的 `SQL` 优化过程会更加复杂，例如多表 `join` 查询时，怎么查更合适？单表复杂 `SQL` 查询时，有多条索引可以走，走哪条速度最快....，因此一条 `SQL` 的最优执行计划，需要结合多方面的优化策略来生成，例如 `MySQL` 优化器的一些优化准则如下：

- ❶多条件查询时，重排条件先后顺序，将效率更好的字段条件放在前面。
- ❷当表中存在多个索引时，选择效率最高的索引作为本次查询的目标索引。
- ❸使用分页 `Limit` 关键字时，查询到对应的数据条数后终止扫表。
- ❹多表 `join` 联查时，对查询表的顺序重新定义，同样以效率为准。
- ❺对于 `SQL` 中使用函数时，如 `count()、max()、min()...` ，根据情况选择最优方案。
  - `max()` 函数：走 `B+` 树最右侧的节点查询（大的在右，小的在左）。
  - `min()` 函数：走 `B+` 树最左侧的节点查询。
  - `count()` 函数：如果是 `MyISAM` 引擎，直接获取引擎统计的总行数。
  - `.....`
- ❻对于 `group by` 分组排序，会先查询所有数据后再统一排序，而不是一开始就排序。
- ❼ `.....`

总之，根据 `SQL` 不同，优化器也会基于不同的优化准则选择出最佳的执行计划。但需要牢记的

存储引擎中会干的工作

经过优化器后，会得到一个最优的执行计划，紧接着工作线程会根据最优计划，去依次调用存储引擎提供的 `API`，在上篇文章中提到过，存储引擎主要就是负责在磁盘读写数据的，不同的存储引擎，存储在本地磁盘中的数据结构也并不相同，但这些底层实现并不需要 `MySQL` 的上层服务关心，因为上层服务只需要负责调用对应的 `API` 即可，存储引擎的 `API` 功能都是相同的。

工作线程根据执行计划调用存储引擎的 `API` 查询指定的表，最终也就是会发生磁盘 `IO`，从磁盘中检索数据，当然，检索的数据有可能是磁盘中的索引文件，也有可能是磁盘中的表数据文件，这点要根据执行计划来决定，我们只需要记住，经过这一步之后总能够得到执行结果即可。

> 但有个小细节，还记得最开始创建数据库连接时，对登录用户的授权步骤嘛？当工作线程去尝试查询某张表时，会首先判断一下线程自身维护的客户端连接，其登录的用户是否具备这张表的操作权限，如果不具备则会直接返回权限不足的错误信息。

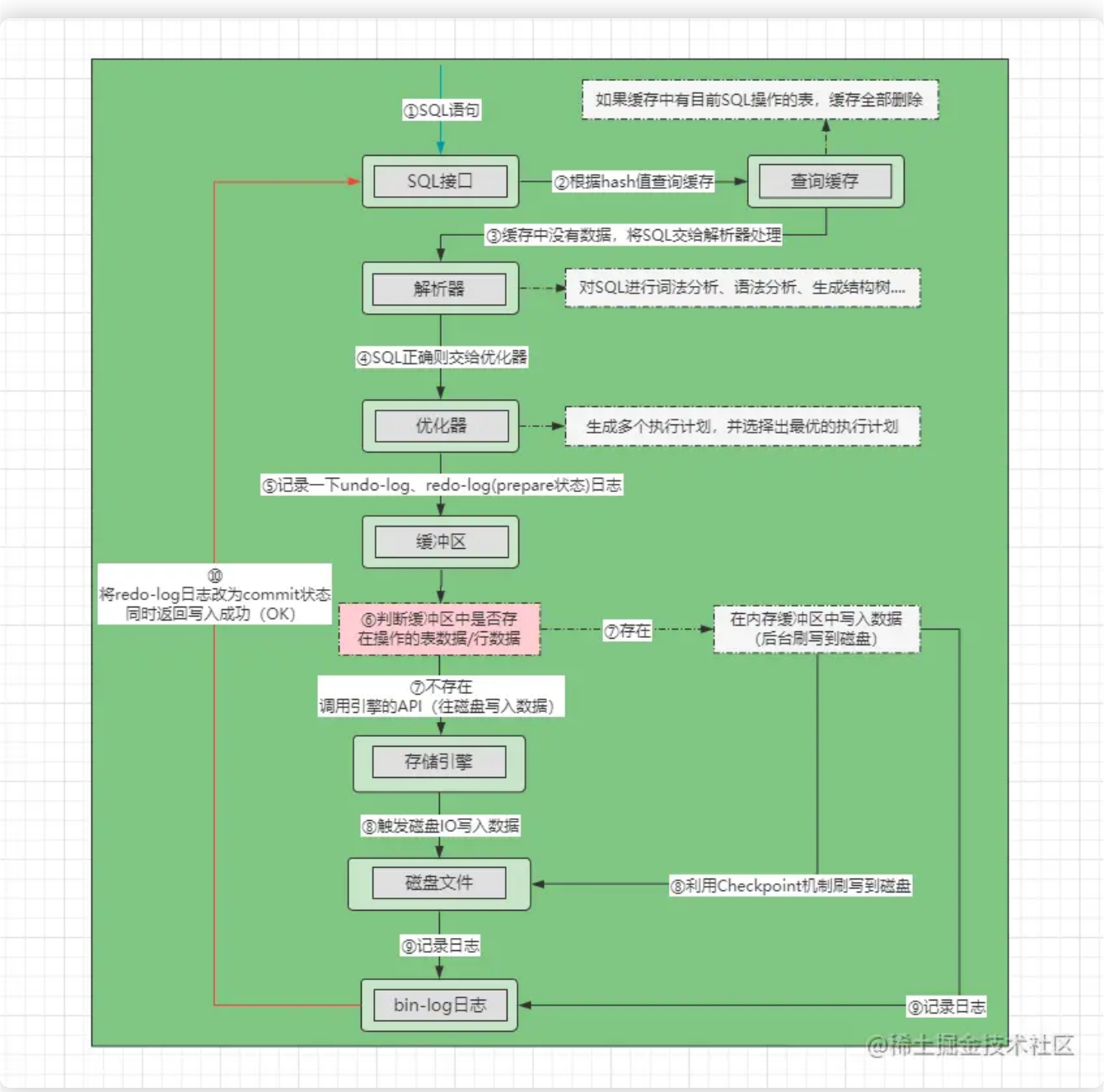
不过存储引擎从磁盘中检索出目标数据后，并不会将所有数据全部得到后再返回，而是会逐条返回给 `SQL` 接口，然后会由 `SQL` 接口完成最后的数据聚合工作，对于这点稍后会详细分析。

> 下来再来看看写入 `SQL` 的执行过程，因为读取和写入操作之间，也会存在些许差异。

3.2、一条写入SQL的执行过程

假设此时要执行下述这一条写入类型的 `SQL` 语句（还是基于之前的表数据）：

上面这条 SQL 是一条典型的修改 SQL，但除开修改操作外，新增、删除等操作也属于写操作，写操作的意思是指会对表中的数据进行更改。同样先上一个完整的流程图：



从上图来看，相较于查询 SQL，写操作的 SQL 执行流程明显会更复杂一些，这里也先简单总结一下每一步流程，然后再详细分析一下其中一些与查询 SQL 中不同的步骤：

- ①先将 SQL 发送给 SQL 接口， SQL 接口会对 SQL 语句进行哈希处理。
- ②在缓存中根据哈希值检索数据，如果缓存中有则将对应表的所有缓存全部删除。
- ③经过缓存后会将 SQL 交给解析器，解析器会判断 SQL 语句是否正确：
  - 错误：抛出 1064 错误码及相关的语法错误信息

- ⑥在缓冲区中查找是否存在当前要操作的行记录或表数据（内存中）：
  - 存在：
    - ⑦直接对缓冲区中的数据进行写操作。
    - ⑧然后利用 `Checkpoint` 机制刷写到磁盘。
  - 不存在：
    - ⑦根据执行计划，调用存储引擎的 `API` 。
    - ⑧发生磁盘 `IO` ，对磁盘中的数据做写操作。
- ⑨写操作完成后，记录 `bin-log` 日志，同时将 `redo-log` 日志中的记录改为 `commit` 状态。
- ⑩将 `SQL` 执行耗时及操作成功的结果返回给 `SQL` 接口，再由 `SQL` 接口返回给客户端。

整个写 `SQL` 的执行过程，前面的一些步骤与查 `SQL` 执行的过程没太大差异，唯一一点不同的在于缓存哪里，原本查询时是从缓存中尝试获取数据。而写操作时，由于要对表数据发生更改，因此如果在缓存中发现了要操作的表存在缓存，则需要将整个表的所有缓存清空，确保缓存的强一致性。

> OK~，除开上述这点区别外，另外多出了唯一性判断、一个缓冲区写入，以及几个写入日志的步骤，接下来一起来聊聊这些。

唯一性判断主要是针对插入、修改语句来说的，因为如果表中的某个字段建立了唯一约束或唯一索引后，当插入/修改一条数据时，就会先检测一下目前插入/修改的值，是否与表中的唯一字段存在冲突，如果表中已经存在相同的值，则会直接抛出异常，反之会继续执行。

> 很简单哈~，接着再来聊聊缓冲区和日志！

其实在上篇中聊到过，由于 `CPU` 和磁盘之间的性能差距实在过大，因此 `MySQL` 中会在内存中设计一个「缓冲区」的概念，主要目的是在于弥补 `CPU` 与磁盘之间的性能差距。

数据/目标表，如果存在则直接对缓冲区中的数据进行操作，然后 MySQL 会在后台以一种名为 Checkpoint 的机制，将缓冲区中更新的数据刷回到磁盘。只有当缓冲区没有找到目标数据时，才会去真正调用存储引擎的 API ，然后发生磁盘 IO ，去对应磁盘中的表数据进行修改。

> 不过值得注意的一点是：虽然缓冲区中有数据时会先操作缓冲区，然后再通过 Checkpoint 机制刷写磁盘，但这两个过程不是连续的！也就是说，当线程对缓冲区中的数据操作完成后，会直接往下走，数据落盘的工作则会交给后台线程。
> 不过虽然两者之间是异步的，但对于人而言，这个过程不会有太大的感知，毕竟 CPU 在运行的时候，都是按纳秒、微秒级作为单位。

但不管数据是在缓冲区还是磁盘，本质上数据更改的动作都是发生在内存的，就算是修改磁盘数据，也是将数据读到内存中操作，然后再将数据写回磁盘。不过在「写 SQL 」执行的前后都会记录日志，这点下面详细聊聊，这也是写 SQL 与读 SQL 最大的区别。

写操作时的日志

执行「读 SQL 」一般都不会有状态，也就是说： MySQL 执行一条 select 语句，几乎不会留下什么痕迹。但这里为什么用几乎这个词呢？因为查询时也有些特殊情况会留下“痕迹”，就是慢查询 SQL ：

> **「慢查询 SQL 」**：查询执行过程耗时较长的 SQL 记录。
> 在执行查询 SQL 时，大多数的普通查询 MySQL 并不关心，但慢查询 SQL 除外，这类 SQL 一般是引起响应缓慢问题的“始作俑者”，所以当一条查询 SQL 的执行时长超过规定的时间限制，就会被“记录在案”，也就是会记录到慢查询日志中。

与「查询 SQL 」恰恰相反，任何一条写入类型的 SQL 都是有状态的，也就代表着只要是会对

录 redo-log 日志。

redo-log 日志是 InnoDB 引擎专属的，主要是为了保证事务的原子性和持久性，这里会将写 SQL 的事务过程记录在案，如果服务器或者 MySQL 宕机，重启时就可以通过 redo_log 日志恢复更新的数据。在「写 SQL 」正式执行之前，就会先记录一条 prepare 状态的日志，表示当前「写 SQL 」准备执行，然后当执行完成并且事务提交后，这条日志记录的状态才会更改为 commit 状态。

> 除开上述的 redo-log、undo-log 日志外，同时还会记录 bin-log 日志，这个日志和 redo-log 日志很像，都是记录对数据库发生更改的 SQL，只不过 redo-log 是 InnoDB 引擎专属的，而 bin-log 日志则是 MySQL 自带的日志。

不过无论是什么日志，都需要在磁盘中存储，而本身「写 SQL 」在磁盘中写表数据效率就较低了，此时还需写入多种日志，效率定然会更低。对于这个问题 MySQL 以及存储引擎的设计者自然也想到了，所以大部分日志记录也是采用先写到缓冲区中，然后再异步刷写到磁盘中。

> 比如 redo-log 日志在内存中会有一个 redo_log 缓冲区中，bin-log 日志也同理，当需要记录日志时，都是先写到内存中的缓冲区。

那内存中的日志数据何时会刷写到磁盘呢？对于这点则是由刷盘策略来决定的，redo-log 日志的刷盘策略由 innodb_flush_log_at_trx_commit 参数控制，而 bin-log 日志的刷盘策略则可以通过 sync_binlog 参数控制：

- innodb_flush_log_at_trx_commit：
  - 0：间隔一段时间，然后再刷写一次日志到磁盘（性能最佳）。
  - 1：每次提交事务时，都刷写一次日志到磁盘（性能最差，最安全，默认策略）。
  - 2：有事务提交的情况下，每间隔一秒时间刷写一次日志到磁盘。
- sync_binlog：
  - 0：同上述 innodb_flush_log_at_trx_commit 参数的 2。

到这里就大致阐述了一下「写 SQL 」执行时，会写的一些日志记录，这些日志在后续做数据恢复、迁移、线下排查时都较为重要，因此后续也会单开一篇详细讲解。

四、一条SQL执行完成后是如何返回的?

一条「读 SQL 」或「写 SQL 」执行完成后，由于 SQL 操作的属性不同，两者之间也会存在差异性，

4.1、读类型的SQL返回

前面聊到过， MySQL 执行一条查询 SQL 时，数据是逐条返回的模式，因为如果等待所有数据全部查出来之后再一次性返回，必然会导致撑满内存。

> 不过这里的返回，并不是指返回客户端，而是指返回 SQL 接口，因为从磁盘中检索出目标数据时，一般还需要对这些数据进行再次处理，举个例子理解一下。

```sql
SELECT user_id FROM `zz_user` WHERE user_sex = "男" AND user_name = "竹子④号";
```

还是之前那条查询 SQL ，这条 SQL 要求返回的结果字段仅有一个 user_id ，但在磁盘中检索数据时，会直接将这个字段单独查询出来吗？并不是的，而是会将整条行数据全部查询出来，如下：

```
+---------+-------------+---------+-------------+
| user_id | user_name   | user_sex | user_phone  |
+---------+-------------+---------+-------------+
|    6    |   竹子④号   |    男    | 17777777777 |
+---------+-------------+---------+-------------+
```

> 还有一点需要牢记：就算没有查询到数据，也会将执行状态、执行耗时这些信息返回给 SQL 接口，然后由 SQL 接口向客户端返回 NULL 。

不过当查询到数据后，在正式向客户端返回之前，还会顺手将结果集放入到缓存中。

4.2、写类型的SQL返回

写 SQL 执行的过程会比读 SQL 复杂，但写 SQL 的结果返回却很简单，写类型的操作执行完成之后，仅会返回执行状态、受影响的行数以及执行耗时，比如：

```sql
UPDATE `zz_user` SET user_sex = "女" WHERE user_id = 6;
```

这条 SQL 执行成功后，会返回 Query OK, 1 row affected (0.00 sec) 这组结果，最终返回给客户端的则只有「受影响的行数」，如果写 SQL 执行成功，这个值一般都会大于 0 ，反之则会等于 0 。

4.3、执行结果是如何返回给客户端的？

对于这个问题的答案其实很简单，由于执行当前 SQL 的工作线程，本身也维护着一个数据库连接，这个数据库连接实际上也维持着客户端的网络连接，如下：

```
mysql> show processlist;
+----+------+-----------------+------+---------+------+-------+------------------+
| Id | User | Host            | db   | Command | Time | State | Info             |
+----+------+-----------------+------+---------+------+-------+------------------+
|  8 | root | localhost:52519 | NULL | Query   |    0 | NULL  | show processlist |
+----+------+-----------------+------+---------+------+-------+------------------+
1 row in set (0.01 sec)

mysql> _
```


当结果集处理好了之后，直接通过 `Host` 中记录的地址，将结果集封装成 `TCP` 数据报，然后返回即可。

> 数据返回给客户端之后，除非客户端主动输入 `exit` 等退出连接的命令，否则连接不会立马断开。

如果要断开客户端连接时，又会经过 `TCP` 四次挥手的过程。

> 不过就算与客户端断开了连接，`MySQL` 中创建的线程并不会销毁，而是会放入到 `MySQL` 的连接池中，等待其他客户端复用当前连接。一般情况下，一条线程在八小时内未被复用，才会触发 `MySQL` 的销毁工作。

五、SQL执行篇总结

看到这里，`SQL` 执行原理篇也走进了尾声，其实 `SQL` 语句的执行过程，实际上也就是 `MySQL` 的架构中一层层对其进行处理，理解了 `MySQL` 架构篇的内容后，相信看 `SQL` 执行篇也不会太难，经过这篇文章的学习后，相信大家对数据库的原理知识也能够进一步掌握，那我们

文章被收录于专栏：

全解MySQL数据库

从MySQL整体架构出发，到SQL优化、MySQL索引、慢查询优化...

关注专栏

相关课程

VIP 高并发秒杀的设计精要与实现

秦二爷 LV.5 1744购买

¥17.94 ~~¥29.9~~ 限时优惠价 · 剩余 0 天

VIP Elasticsearch 从入门到实践

spoofer LV.1 1598购买

¥23.94 ~~¥39.9~~ 限时优惠价 · 剩余 0 天

评论

输入评论（Enter换行，Ctrl + Enter发送）

全部评论 34

最新 最热

BinaryZx LV.2 JY.4 后端开发 @ 自己跳动 2天前

大佬！！！收下我的膝盖！！！

点赞 1

竹子爱熊猫 （作者） 2天前

夸张了夸张了，哈哈哈

点赞 回复

78 34 收藏

膜拜大佬，内容太顶了

点赞 1

竹子爱熊猫 （作者） 10天前

三克油~

点赞 回复

xiaobinqt JY.3 15天前

update 的操作在某些情况下会 小于 0 ?

点赞 1

竹子爱熊猫 （作者） 15天前

写错了抱歉，应该等于0，这里笔误，我更正一下。

点赞 回复

头秃了也拗不过... LV.2 JY.3 PHP开发 26天前

先收藏，后吃灰

点赞 1

竹子爱熊猫 （作者） 25天前

收藏=学会

点赞 回复

StoneDB LV.2 JY.5 数据库架构师 @ 石原... 1月前

关注了，数据库大佬，什么时候来研究一下StoneDB~

点赞 2

竹子爱熊猫 （作者） 1月前

78 34 收藏

下来就仔细研究会儿

1 回复

StoneDB 回复 竹子爱熊猫 1月前

好嘞~我们已经开源啦，欢迎来Github上给我们提issue和PR哦~

github.com

“简单看了一下，StoneDB是自研的一款MySQL存储引擎，但似乎使用了知识网... ”

点赞 回复

大米滔天 LV.1 JY.4 技术负责人 1月前

还文章要鼓励，给你顶一个

点赞 1

竹子爱熊猫 （作者） 1月前

三克油~

点赞 回复

蓝色记忆 LV.4 JY.4 小流码农 1月前

MySQL8是不是把查询缓存去掉了？

点赞 1

竹子爱熊猫 （作者） 1月前

对的，因为查询缓存实际上比较鸡肋，比如一些问题：
①缓存命中率低：几乎大部分SQL都无法从查询缓存中获得数据。
②占用内存高：将大量查询结果放入到内存中，会占用至少几百MB的内存。
③查询时多几步开销：查询表之前会先查一次缓存，查询后会将结果放入缓存。
④缓存维护成本不小，每次更新、插入、删除数据时，都需要清空缓存中的数据。
...

展开

3 回复

点赞 6

竹子爱熊猫 （作者） 1月前

欢迎大佬

1 回复

宁在春 回复 竹子爱熊猫 1月前

大佬你这可是属于倒置了，我目前和大佬这个名词还扯不上一点关系

> “欢迎大佬”

点赞 回复

查看更多回复

芥末拌饭 LV.3 JY.5 后端，大数据 1月前

大佬，关注你了

点赞 3

竹子爱熊猫 （作者） 1月前

谢谢，后续尽量会保持三天一更的频率

点赞 回复

芥末拌饭 回复 竹子爱熊猫 1月前

向大佬学习，期待你分享更多的干货

> “谢谢，后续尽量会保持三天一更的频率”

点赞 回复

查看更多回复

狗霸天 JY.3 后端开发 1月前

很不错！等待大佬的更新

点赞 1

78 34 收藏

点赞 回复

借故 JY.3 1月前

点赞 1

竹子爱熊猫 （作者） 1月前

点赞 回复

unidentifiable LV.4 JY.6 后端小菜鸡 1月前

点赞 1

竹子爱熊猫 （作者） 1月前

点赞 回复

CC_LKL JY.3 1月前

大佬咋又重发了这篇勒 

点赞 1

竹子爱熊猫 （作者） 1月前

是的，因为昨天文章状态变为了审核不通过，所以又重发了一次。

点赞 回复

相关推荐

程序员乔戈里 | 2年前 | Java

78 34 收藏

竹子爱熊猫 | 18天前 | 数据库 · MySQL · Java

(十一)MySQL日志篇之undo-log、redo-log、bin-log.....傻傻分不清!

1150 31 22

摸鱼的春哥 | 9月前 | 前端 · JavaScript

2022，前端的天🐑要怎么变?

17.3w 1678 723

岛上码农 | 5月前 | Flutter · iOS · 前端

让你的聊天气泡丰富多彩!

6069 31 4

前端阿飞 | 11月前 | 前端 · JavaScript

10个常见的前端手写功能，你全都会吗?

14.0w 3456 309

王圣松 | 2年前 | 前端

一位00后前端2年经验的成长历程

3.6w 344 195

阳光是sunny | 7月前 | 前端 · JavaScript

三面面试官：运行 npm run xxx 的时候发生了什么?

14.3w 4552 401

小林coding | 1年前 | MySQL · 后端

完蛋，公司被一条 update 语句干趴了!

3.0w 231 26

程序员依扬 | 3年前 | 面试 · 前端

【1 月最新】前端 100 问：能搞懂 80% 的请把简历给我

59.3w 10340 353

LBJ | 7月前 | 前端 · 后端 · JavaScript

78 34 收藏

10亿级订单系统分库分表设计思路！

2.4w 427 22

Java技术栈 | 1年前 | 程序员

雷军做程序员时写的博客，太牛了。。

26.5w 858 254

捡田螺的小男孩 | 1年前 | 后端 · Java

美团二面：Redis与MySQL双写一致性如何保证?

6.4w 663 134

非优秀程序员 | 11月前 | 前端 · JavaScript

如何用 CSS 中写出超级美丽的阴影效果

99.7w 822 49

前端小溪 | 2月前 | 前端 · 面试 · Vue.js

上海疫情后一个前端的面试心路历程

4.9w 965 229

愣锤 | 4年前 | Vue.js · JavaScript · 前端

Vue 项目里戳中你痛点的问题及解决办法（更新）

14.5w 3699 168

神三元 | 2年前 | JavaScript · 面试

（建议精读）HTTP灵魂之问，巩固你的 HTTP 知识体系

21.5w 4682 178

Sunshine_Lin | 1年前 | 前端 · JavaScript · ECMAScript 6

「万字总结」熬夜总结50个JS的高级知识点，全都会你就是神！！！

9.9w 2754 132

鱼酱 | 1年前 | 前端 · 程序员

一位 23 岁深漂女前端的心路历程丨2021 年中总结

Mysql中，21个写SQL的好习惯，你值得拥有呀

1.4w 312 28