参考：

MySQL实战45讲

前言

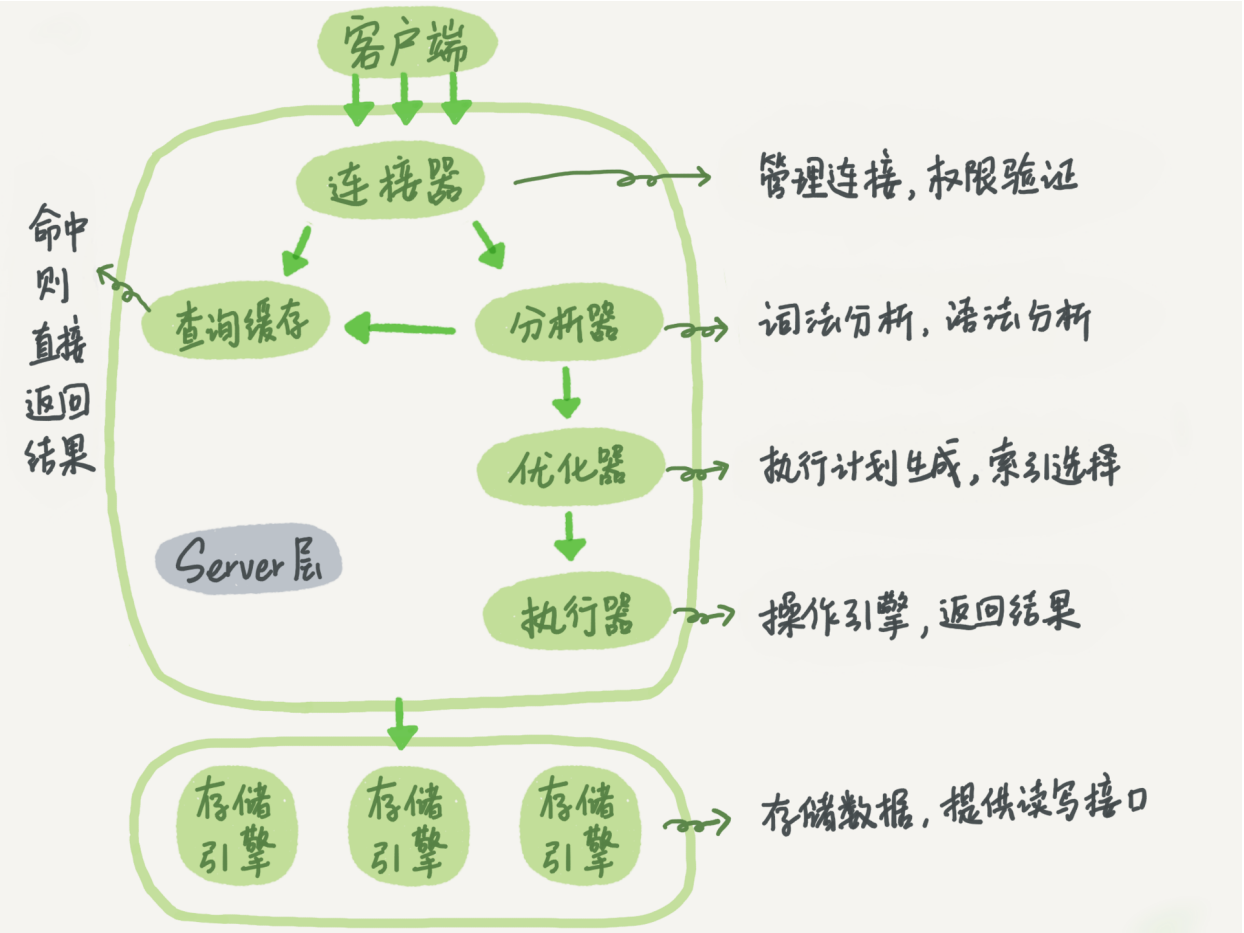
一直在写SQL语句，然而SQL语句在MySQL中事怎么执行的呢？我们知道MySQL是由客户端，服务端，存储引擎三部分组成的，存储引擎是存储数据的核心，所以我们在客户端写的一条SQL语句肯定最终经历的步骤肯定是由客户端到服务端再到存储引擎的（关于存储引擎就不在该篇叙述了）。那么就让我们详细看看这个步骤吧。

一条查询语句的执行流程

例如，表A中有字段id，我们写一条查询语句：

```
select * from T where id = 10;
```

我们使用者关注点就是这条语句写错了没，目测没问题，点击执行，我们就能得到结果，现在我们要做的是化简为繁，去了解这条语句背后的经历，上演一部小蝌蚪找妈妈的大电影。



直接把林晓斌大牛的图拿过来，这是MySQL的基本架构示意图，我们就围绕这张图详细去了解下。

客户端没啥说的，我们从服务端开始说，它包括了连接器，查询缓存，分析器，优化器，执行器等等，这些基本涵盖了MySQL的大多数核心服务功能了。内置函数，存储过程，触发器，视图等也是这边实现的。

存储引擎负责数据存储和提取，支持InnoDB，MyISAM，Memory等多种存储引擎，InnoDB是最常用的也是MySQL5.5.5版本开始的默认存储引擎，

连接器

从图中看，连接器是访问服务端的第一步，不难理解，服务端在那，第一步肯定要和它建立连接，负责接待的就是连接器。

连接命令一般按以下格式：

mysql -h$ip -P$port -u$user -p

密码则在在该命令生效后输入，也可以接在该命令后，但不安全。然后用户名和密码通过验证后就会读取用户权限，本次连接里涉及到的权限问题皆依赖该次读取为主，顾一旦连接，就算修改了用户权限，只要未重新连接，均在该次连接不予生效。

```
Microsoft Windows [版本 10.0.17134.523]
(c) 2018 Microsoft Corporation。保留所有权利。

C:\Users\pc>mysql -h127.0.0.1 -P3306 -uroot -p
Enter password: ******
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 3
Server version: 5.7.20 MySQL Community Server (GPL)

Copyright (c) 2000, 2017, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

登陆成功后会有类似mysql>这样的mysql命令行标志出现。

关于命令show processlist的使用，能够查看连接的状态，类似下图

```
mysql> show processlist
    -> ;
+----+------+-----------+------+---------+------+----------+-----------------+
| Id | User | Host      | db   | Command | Time | State    | Info            |
+----+------+-----------+------+---------+------+----------+-----------------+
|  3 | root | localhost | NULL | Query   |    0 | starting | show processlist |
+----+------+-----------+------+---------+------+----------+-----------------+
1 row in set (0.01 sec)

mysql> use test;
Database changed
mysql> show processlist;
+----+------+-----------+------+---------+------+----------+-----------------+
| Id | User | Host      | db   | Command | Time | State    | Info            |
+----+------+-----------+------+---------+------+----------+-----------------+
|  3 | root | localhost | test | Query   |    0 | starting | show processlist |
+----+------+-----------+------+---------+------+----------+-----------------+
1 row in set (0.00 sec)
```

客户端过长时间没动静的话，mysql也会采取措施直接断开，该时间由参数wait_timeout控制，默认的是8小时。

长连接和短连接的概念，长连接指连接成功后，客户端持续有请求，一直使用一个连接，短连接指每次执行完很少的几次查询就断开连接下次查询再重新建立一个。

建立连接是个复杂的过程，所以生产上一般建议多使用长连接，但使用长连接有个问题，因为连接一直存在会一直消耗内存，所以如果连接存在时间过长，总会出现内存溢出的现象，针对该问题有如下思路可做考虑：

1.定期断开长连接。按时间定期也可以做判断，执行过一个占内存大的查询；

2.只限MySQL5.7以及以上版本，在执行完一个较大操作后，可以执行mysql_reset_connection初始化连接资源，该操作不需要重新连接和权限验证，仅仅是恢复到刚刚连接的状态。

查询缓存

按照图中第二步，就是查询缓存了，之前做过的查询结果会以key-value的型式暂存在内存中，key对应查询语句，value就是结果，如果新的一条语句进来，命中了某个key，就可以直接取得结果，而不需要进行后面的复杂过程了。

然而，却不建议使用，对某个表的更新会让这个表的所有查询缓存失效，所以说正常情况查询缓存的失效是非常频繁的，不过也有例外的，配置表这种不常更改的表是可以考虑用查询缓存的，MySQL也提供了这种按需的方式。然而实际中，有个叫redis的东西，做缓存更牛逼，说实在话，就算查询缓存命中那还是访问了MySQL，生产中尽量能避免访问数据库的就应该尽量避免。当然也是按需，生产中redis的维护毕竟是要额外费用的，如本人曾参加的某个项目组，甲方就因维护费用问题拒绝使用redis。

那如何按需呢，我们把参数query_cache_type设置成2(MySQL配置文件my.ini中)，这样对于默认的SQL就不会使用查询缓存，而要使用就像这样写：

```
select SQL_CACHE * from T where id = 10;
```

当然MySQL8已经删掉查询缓存了，也不需要考虑了。

分析器

按照图中第三步，现在就走到分析器了，如其名，这一步肯定是检查SQL语句的，当然没有查询缓存的话，这就是第二步。

分析器总的来说干了两件事，词法分析和语法分析。就以上述的sql为例：

```
select * from T where id = 10;
```

select被识别，这是一个查询语句，“T”被识别成“表名 T”，“id”被识别成“列名 id”。

语法分析则检查该SQL是否符合MySQL的语法。

优化器

到了这一步，MySQL已经知道你要干什么了，也确保了你要干的事情是能执行的了，然后这一步干了什么呢，就是选出执行的最优方案，比如多个索引时决定使用哪个索引，多表关联（join）时决定表的连接顺序。举个例子：

```
select * from t1 join t2 using(id) where t1.c = 10 and t2.d = 20;
```

对于MySQL来说，这条语句它是有两个方案的。

1.从t1取出c=10的记录的id并根据id关联t2，再判断t2的d是否为20；

2.从t2取出d=20的记录的id并根据id关联t1，再判断t1的c是否为10；

这两个方案的效率最终肯定是不一样的，而优化器就是要决定使用哪个方案更佳。

执行器

这一步就是最后的关键一步了，走了那么多步的SQL语句，是骡子是马这一步都要去执行了。首先会进行权限认证，如果用户对表不具备查询权限，会返回权限错误（补充一点，查询缓存命中也会做权限认证）。

通过权限认证，执行器就会打开表，根据表的引擎定义，使用这个引擎的接口。

以上述SQL为例，id字段无索引，那么流程如下：

1.调用InnoDB引擎的接口取该表第一行，判断id是否为10，不是则跳过，是则放入结果集；

2.继续取下一行，判断，直到最后一行；

3.执行器将所有满足条件的结果集返回给客户端。

若有索引，上述步骤不变，取行改为取满足条件的第一行，下一行。

数据库的慢查询日志中有一个rows_examined字段，表示该语句执行扫描了多少行。

一条更新语句的执行流程

大体步骤可以说是与上述的查询是一致的，那我们直切更新操作的核心，两个日志：redo log和binlog。为啥MySQL可以恢复到半个月内

任意一秒的状态，就是依靠这两个日志起到的作用。

举个例子：

update T set c = c + 1 where id = 20;

这是一个很简单的更新语句，将T表中id为20的c值加1，我们眼见的是执行完提交事务后，表数据更新了就对了，殊不知这其中也是做了复杂讲究的。

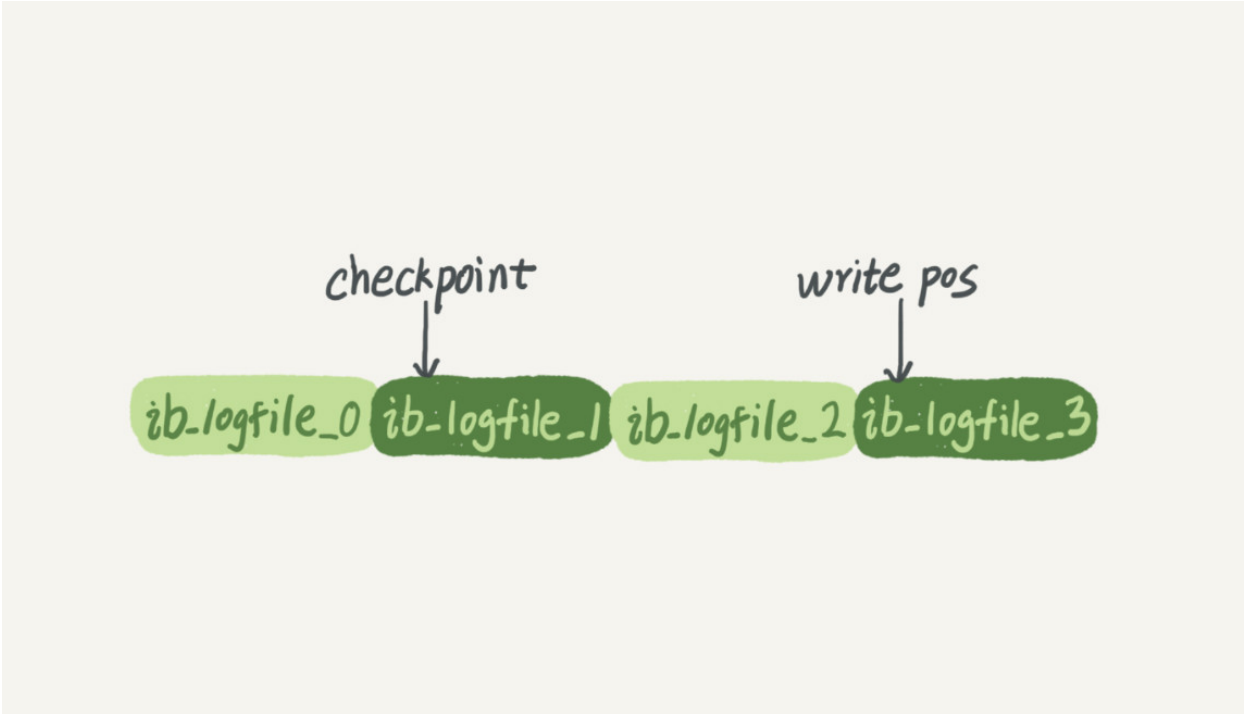
redo log

林晓斌老师针对redo log以《孔乙己》中的故事切入，很是生动形象，这里就不详细叙述了，直接切入重点。

MySQL更新我们知道是要改数据了，如果每一次更新都直接写入磁盘，然后磁盘要找到数据，再更新，这个过程IO成本、查找成本都很高。所以MySQL就引入了先日志再写入的技术，专业的说叫WAL（Write-Ahead Logging）。

redo log就是这么一项技术，怎么个过程呢，每次更新时，InnoDB会把记录写道redo log，并更新内存，宏观上讲，这时候已经算是更新完成了。然后在适当的时间，InnoDB会将这个操作记录更新到磁盘里。

并且redo log的大小是一开始就定好了的，就像黑板一样，从头写到尾，满了，这时候再回到头擦掉从新从头写到尾。比如，redo log配置了一组四个文件，每个大小1GB，那总共就可以记录4GB的操作，以下图为例：



write pos是当前记录的位置，一边写一边向后移动，写到3号文件末尾就会回到0号文件的开始。checkpoint是当前要擦除的位置，也是向后移动并且循环的，擦除前要确保数据已经更新到数据文件。若是write pos追赶上了checkpoint那得等待checkpoint擦除出一定空间才会继续。

形象的说，这里面相当于有三个人分工，A不断把外界的东西一条条写上去，B则定期合适的时间把未处理的内容更新到数据文件，C则不断把黑板上已更新的数据擦除。

正因为redo log的功效，InnoDB就能保证数据库异常重启，之前提交的记录不会丢失，这个能力称作crash-safe。

binlog

针对上述的redo log，这是引擎层的日志，是InnoDB带过来的，像MyISAM就没有，对于Server层来说，也有一个重要的日志就是binlog。

那么为啥要用两套日志呢，或者准确的说，用InnoDB时为啥要同时使用redo log和binlog呢，这就要从两者功能来叙述，redo log我们知

道有个重要的能力就是crash-safe，这是只做归档的binlog所做不到的。同样的，归档也是很重要的，这又是redo log做不到的。

对比两者其实可以总结出三点不同。

1.redo log是InnoDB引擎自带的，binlog则是Server层实现的，所有引擎共有；

2.redo log是物理日志，记录的是“在某个数据页上做了什么修改”，binlog是逻辑日志，记录的是这个语句的原始逻辑，比如“给id=2这一行的c字段加1”（不就是语句本身吗）。

3.redo log是循环写的，空间固定，binlog是追加写的，所有日志不会被覆盖。

以上述更新语句为例，我们在流程中看一下这两个日志是如何起作用的。

update T set c = c + 1 where id = 20;

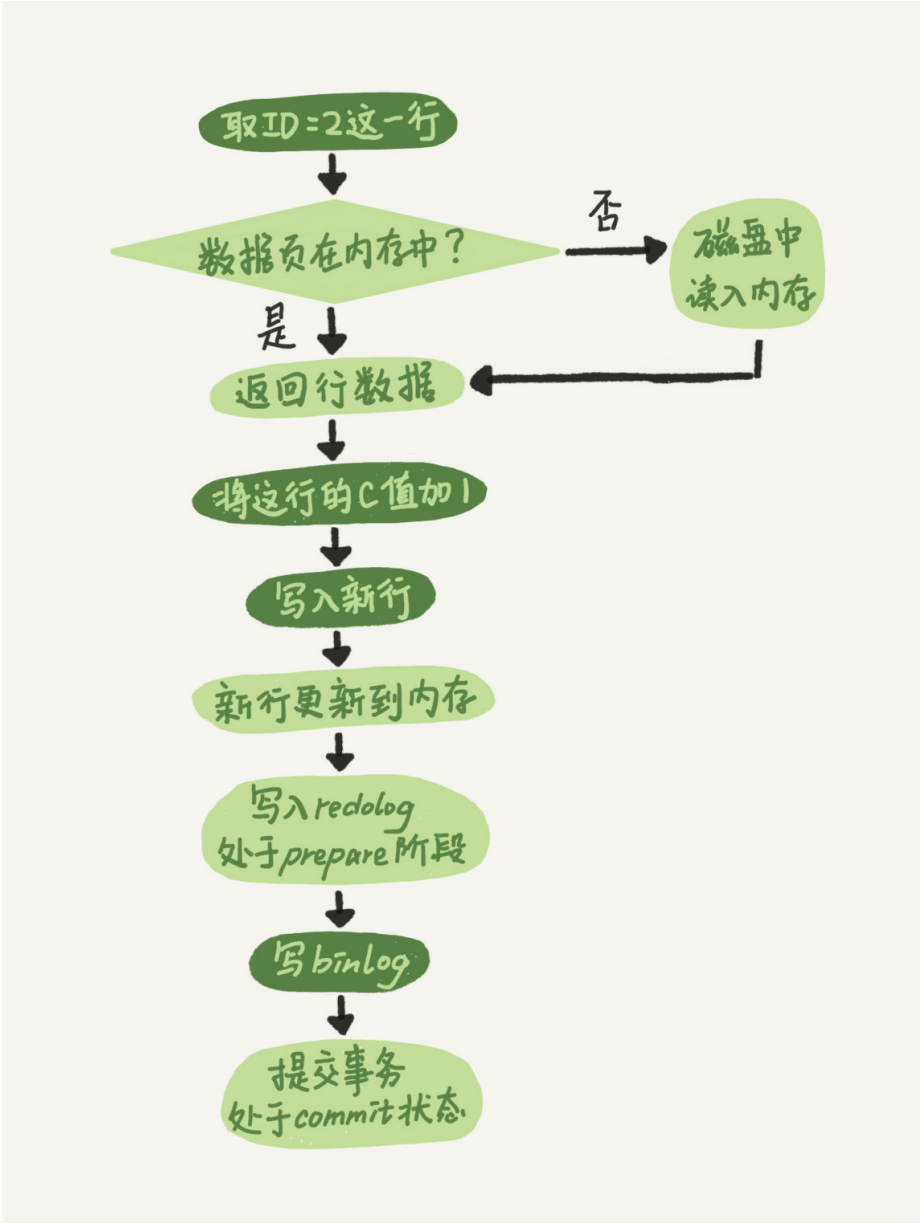
1.执行器先找引擎取id=20这一行，如果id=20这一行的数据页本就在内存中，直接返回给执行器，否则，从磁盘读入内存再返回；

2.执行器拿到引擎给的行数据，把c值加1，再调用引擎接口写入这行新数据；

3.引擎将这行数据更新到内存中，同时记录到redo log中，此时redo log处于prepare状态，然后就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并写入磁盘；

5.执行器调用引擎的提交事务接口，引擎将刚刚写入的redo log改成commit状态，更新完成。



针对redo log的prepare和commit两阶段提交的解读

这个问题就涉及到如何保证redo log和binlog两份日志的逻辑一致。我们用反正法来解读下，按一般思路，两个日志总得又先后吧。

1.先写redo log，若写完redo log，MySQL却异常重启了，但因为redo log已经写完，所以我们可以恢复，但由于binlog没有写完就crash了，这时候binlog就没有记录这条语句，后期如果用binlog恢复数据库就会失去这条数据，而redo log已经处理完，原本是有的。

2.先写binlog，如果binlog写完，发生crash，后期恢复是有的，然后当下，由于redo log并未记录，数据库是不应该有这条数据的。

所以无论哪个日志先写，其实都可能因为crash问题造成数据的错乱。这里就推出redo log的两阶段的方式，由于先只是prepare阶段，比如prepare后crash无所谓，redo log这条记录并不生效，若写完binlog后crash也无所谓，redo log 和binlog都已经有记录