# Mysql基本架构

连接器：身份认证和权限相关(登录MySQL的时候)。

查询缓存：执行查询语句的时候，会先查询缓存（MySQL 8.0 版本后移除，因为这个功能不太实用）。

分析器：没有命中缓存的话，SQL 语句就会经过分析器，分析器就是要先看你的 SQL 语句要干嘛，再检查你的 SQL 语句语法是否正确。

优化器：按照 MySQL 认为最优的方案去执行。

执行器：执行语句，然后从存储引擎返回数据。

简单来说 MySQL 主要分为 Server 层和存储引擎层：

Server层：主要包括连接器、查询缓存、分析器、优化器、执行器等，所有跨存储引擎的功能都在这一层实现，比如存储过程、触发器、视图，函数等，还有一个通用的日志模块 binglog 日志模块。

存储引擎：主要负责数据的存储和读取，采用可以替换的插件式架构，支持 InnoDB、MyISAM、Memory 等多个存储引擎，其中 InnoDB 引擎有自有的日志模块 redolog 模块。现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始就被当做默认存储引擎了。

# Server组件

## 连接器

连接器主要负责将 mysql 客户端和服务端建立连接，连接成功后，会获取当前连接用户的权限。这里获取到的权限对整个连接都有效，一旦连接成功后，如果使用管理员账号对该用户更改权限，当前连接中的拥有的权限保持不变，只有等到下次重新连接才会更新权限。

## 查询缓存

连接成功后，即开始要正式执行 select 语句了，但是在执行查询之前，mysql 会去看下有没有该条语句的缓存内容，如果有缓存直接从缓存中读取并返回数据，不再执行后面的步骤了，结束查询操作。

如果没有缓存则继续往后执行，并将执行结果和语句保存在缓存中。

注意在 mysql8 后已经没有查询缓存这个功能了，因为这个缓存非常容易被清空掉，命中率比较低。只要对表有一个更新，这个表上的所有缓存就会被清空，因此你刚缓存下来的内容，还没来得及用就被另一个更新给清空了。

## 分析器

既然没有查到缓存，就需要开始执行 sql 语句了，在执行之前肯定需要先对 sql 语句进行解析。分析器主要对 sql 语句进行语法和语义分析，检查单词是否拼写错误，还有检查要查询的表或字段是否存在。

如果分析器检测出有错误就会返回类似 "You have an error in your sql" 这样的错误信息，并结束查询操作。

## 优化器

通过分析器之后，mysql 就算是理解了你要执行的操作了。通常对于同一个 sql 语句，mysql 内部可能存在多种执行方案，比如存在多个索引时，该选择哪个索引，多个表关联查询时，怎么确认各个表的连接顺序。

这些方案的执行结果都一样，但是执行效率不一样，所以 mysql 在执行之前需要尝试找出一个最优的方案来，这就是优化器的主要工作。但是 mysql 也会有选择错误方案的时候，这里暂不细说，留到后面再解释原因。

## 执行器

经过优化器选定了一个方案后，执行器就按照选定的方案执行 sql 语句。前面我们有讲过，在连接器中会读取当前用户的权限，连接器中只是获取权限而已，并没有对权限进行判断和校验。

所以在执行器中，在执行语句之前会判断权限，如果没有对应的权限则会直接返回并提示没有相关权限。

这里你可能会问，为什么不在连接器中就直接判断权限呢，这里我觉得可能是因为 mysql 要查询的表并不一定仅限于 sql 语句中字面上的那些表，有的时候可能需要经过分析器和优化器之后才能确定到底要怎么执行，所以权限校验放在执行器中是有道理的。

注意如果是在前面的查询缓存中查到缓存之后，也会在返回结果前做权限校验的。

权限校验通过之后，就继续打开表，调用存储引擎提供的接口去查询并返回结果集数据。

到这里，一条查询 sql 语句就执行结束了。

# 语句分析

## 查询语句

说了以上这么多，那么究竟一条 sql 语句是如何执行的呢？其实我们的 sql 可以分为两种，一种是查询，一种是更新（增加，更新，删除）。我们先分析下查询语句，语句如下：

select \* from tb\_student A where A.age='18' and A.name=' 张三 ';

结合上面的说明，我们分析下这个语句的执行流程：

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，在 MySQL8.0 版本以前，会先查询缓存，以这条 sql 语句为 key 在内存中查询是否有结果，如果有直接缓存，如果没有，执行下一步。

通过分析器进行词法分析，提取 sql 语句的关键元素，比如提取上面这个语句是查询 select，提取需要查询的表名为 tb\_student,需要查询所有的列，查询条件是这个表的 id='1'。然后判断这个 sql 语句是否有语法错误，比如关键词是否正确等等，如果检查没问题就执行下一步。

接下来就是优化器进行确定执行方案，上面的 sql 语句，可以有两种执行方案：

a.先查询学生表中姓名为“张三”的学生，然后判断是否年龄是 18。

b.先找出学生中年龄 18 岁的学生，然后再查询姓名为“张三”的学生。

那么优化器根据自己的优化算法进行选择执行效率最好的一个方案（优化器认为，有时候不一定最好）。那么确认了执行计划后就准备开始执行了。

进行权限校验，如果没有权限就会返回错误信息，如果有权限就会调用数据库引擎接口，返回引擎的执行结果。

## 更新语句

以上就是一条查询 sql 的执行流程，那么接下来我们看看一条更新语句如何执行的呢？sql 语句如下：

update tb\_student A set A.age='19' where A.name=' 张三 ';

我们来给张三修改下年龄，在实际数据库肯定不会设置年龄这个字段的，不然要被技术负责人打的。其实条语句也基本上会沿着上一个查询的流程走，只不过执行更新的时候肯定要记录日志啦，这就会引入日志模块了，MySQL 自带的日志模块式 binlog（归档日志） ，所有的存储引擎都可以使用，我们常用的 InnoDB 引擎还自带了一个日志模块 redo log（重做日志），我们就以 InnoDB 模式下来探讨这个语句的执行流程。流程如下：

•先查询到张三这一条数据，如果有缓存，也是会用到缓存。

•然后拿到查询的语句，把 age 改为 19，然后调用引擎 API 接口，写入这一行数据，InnoDB 引擎把数据保存在内存中，同时记录 redo log，此时 redo log 进入 prepare 状态，然后告诉执行器，执行完成了，随时可以提交。

•执行器收到通知后记录 binlog，然后调用引擎接口，提交 redo log 为提交状态。

•更新完成。

这里肯定有同学会问，为什么要用两个日志模块，用一个日志模块不行吗?

这是因为最开始 MySQL 并没与 InnoDB 引擎( InnoDB 引擎是其他公司以插件形式插入 MySQL 的) ，MySQL 自带的引擎是 MyISAM，但是我们知道 redo log 是 InnoDB 引擎特有的，其他存储引擎都没有，这就导致会没有 crash-safe 的能力(crash-safe 的能力即使数据库发生异常重启，之前提交的记录都不会丢失)，binlog 日志只能用来归档。

并不是说只用一个日志模块不可以，只是 InnoDB 引擎就是通过 redo log 来支持事务的。那么，又会有同学问，我用两个日志模块，但是不要这么复杂行不行，为什么 redo log 要引入 prepare 预提交状态？这里我们用反证法来说明下为什么要这么做？

•先写 redo log 直接提交，然后写 binlog，假设写完 redo log 后，机器挂了，binlog 日志没有被写入，那么机器重启后，这台机器会通过 redo log 恢复数据，但是这个时候 bingog 并没有记录该数据，后续进行机器备份的时候，就会丢失这一条数据，同时主从同步也会丢失这一条数据。

•先写 binlog，然后写 redo log，假设写完了 binlog，机器异常重启了，由于没有 redo log，本机是无法恢复这一条记录的，但是 binlog 又有记录，那么和上面同样的道理，就会产生数据不一致的情况。

如果采用 redo log 两阶段提交的方式就不一样了，写完 binglog 后，然后再提交 redo log 就会防止出现上述的问题，从而保证了数据的一致性。那么问题来了，有没有一个极端的情况呢？假设 redo log 处于预提交状态，binglog 也已经写完了，这个时候发生了异常重启会怎么样呢？这个就要依赖于 MySQL 的处理机制了，MySQL 的处理过程如下：

•判断 redo log 是否完整，如果判断是完整的，就立即提交。

•如果 redo log 只是预提交但不是 commit 状态，这个时候就会去判断 binlog 是否完整，如果完整就提交 redo log, 不完整就回滚事务。

这样就解决了数据一致性的问题。

# 总结

MySQL 主要分为 Server 层和引擎层，Server 层主要包括连接器、查询缓存、分析器、优化器、执行器，同时还有一个日志模块（binlog），这个日志模块所有执行引擎都可以共用，redolog只有InnoDB有。

引擎层是插件式的，目前主要包括，MyISAM，InnoDB，Memory 等。

SQL等执行过程分为两类，一类对于查询等过程如下：权限校验🡪查询缓存🡪分析器🡪优化器🡪权限校验🡪执行器🡪引擎

对于更新等语句执行流程如下：分析器🡪权限校验🡪执行器🡪引擎🡪redo log prepare🡪binlog🡪redo log commit