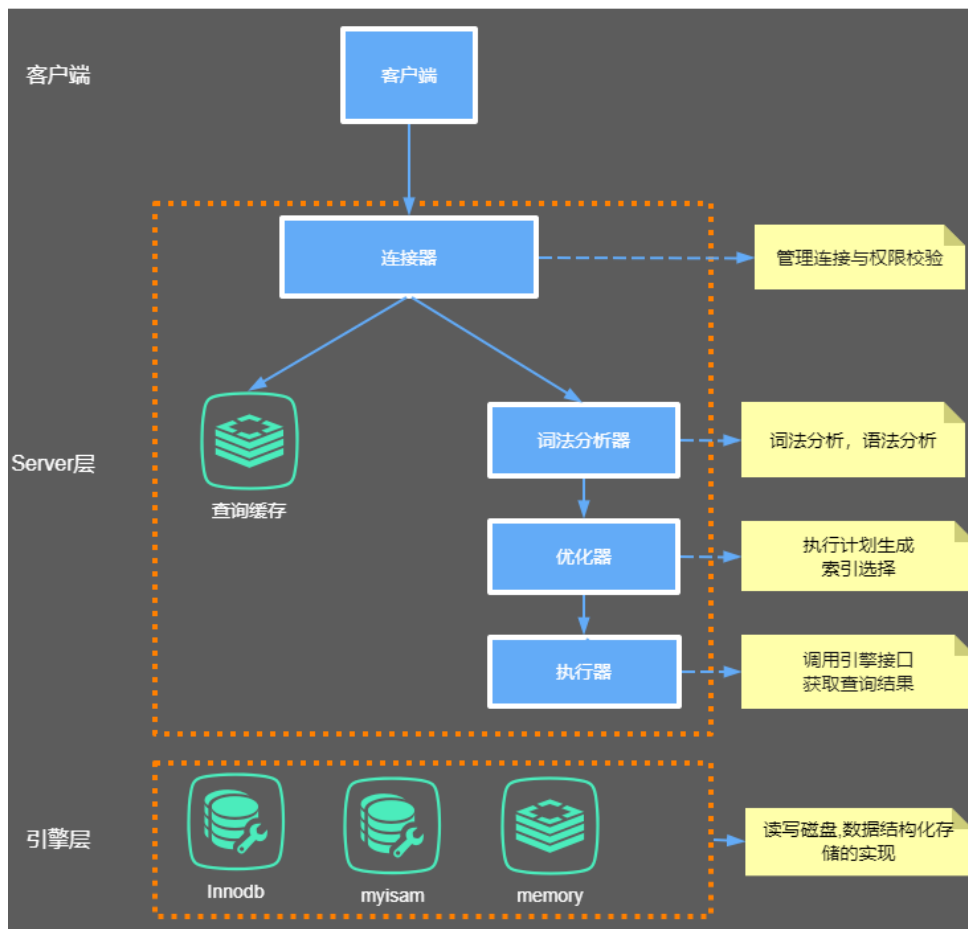


MySQL的内部组件结构



大体来说，MySQL 可以分为 Server 层和存储引擎层两部分。

Server层

主要包括连接器、查询缓存、分析器、优化器、执行器等，涵盖 MySQL 的大多数核心服务功能，以及所有的内置函数（如日期、时间、数学和加密函数等），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、视图等。

存储引擎层

存储引擎层负责数据的存储和提取。其架构模式是插件式的，支持 InnoDB、MyISAM、Memory 等多个存储引擎。现在最常用的存储引擎是 InnoDB，它从 MySQL 5.5.5 版本开始成为了默认存储引擎。也就是说如果我们在 create table 时不指定表的存储引擎类型，默认会给你设置存储引擎为 InnoDB。

下面我们来看下 Server 层的连接器、查询缓存、分析器、优化器、执行器分别主要干了哪些事情。

连接器

我们知道由于 MySQL 是开源的，他有非常多种类的客户端：navicat, mysql front, jdbc, SQLyog 等非常丰富的客户端，包括各种编程语言实现的客户端连接程序，这些客户端要向 mysql 发起通信都必须先跟 Server 端建立通信连接，而建立连接的工作就是有连接器完成的。

第一步，你会先连接到这个数据库上，这时候接待你的就是连接器。连接器负责跟客户端建立连接、获取权限、维持和管理连接。连接命令一般是这么写的：

```
1 [root@192 ~]# mysql -h host[数据库地址] -u root[用户] -p root[密码] -P 3306
```

连接命令中的 mysql 是客户端工具，用来跟服务端建立连接。在完成经典的 TCP 握手后，连接器就要开始认证你的身份，这个时候用的就是你输入的用户名和密码。

- 1、如果用户名或密码不对，你就会收到一个 "Access denied for user" 的错误，然后客户端程序结束执行。

2、如果用户名密码认证通过，连接器会到权限表里面查出你拥有的权限。之后，这个连接里面的权限判断逻辑，都将依赖于此时读到的权限。

这就意味着，一个用户成功建立连接后，即使你用管理员账号对这个用户的权限做了修改，也不会影响已经存在连接的权限。修改完成后，只有再新建的连接才会使用新的权限设置。

查询缓存

连接建立完成后，你就可以执行 select 语句了。执行逻辑就会来到第二步：查询缓存。

MySQL 拿到一个查询请求后，会先到查询缓存看看，之前是不是执行过这条语句。之前执行过的语句及其结果可能会以 key-value 对的形式，被直接缓存在内存中。key 是查询的语句，value 是查询的结果。如果你的查询能够直接在这个缓存中找到 key，那么这个 value 就会被直接返回给客户端。

如果语句不在查询缓存中，就会继续后面的执行阶段。执行完成后，执行结果会被存入查询缓存中。你可以看到，如果查询命中缓存，MySQL 不需要执行后面的复杂操作，就可以直接返回结果，这个效率会很高。

大多数情况查询缓存就是个鸡肋，为什么呢？

因为查询缓存往往弊大于利。查询缓存的失效非常频繁，只要有对一个表的更新，这个表上所有的查询缓存都会被清空。因此很可能你费劲地把结果存起来，还没使用呢，就被一个更新全清空了。对于更新压力大的数据库来说，查询缓存的命中率会非常低。

一般建议大家在静态表里使用查询缓存，什么叫静态表呢？就是一般我们极少更新的表。比如，一个系统配置表、字典表，那这张表上的查询才适合使用查询缓存。好在 MySQL 也提供了这种“按需使用”的方式。你可以将 my.cnf 参数 query_cache_type 设置成 DEMAND。

```
1 my.cnf
2 #query_cache_type有3个值 0代表关闭查询缓存OFF，1代表开启ON，2（DEMAND）代表当sql语句中有SQL_CACHE关键词时才缓存
3 query_cache_type=2
```

这样对于默认的 SQL 语句都不使用查询缓存。而对于你确定要使用查询缓存的语句，可以用 SQL_CACHE 显式指定，像下面这个语句一样：

```
1 mysql> select SQL_CACHE * from test where ID=5;
```

查看当前mysql实例是否开启缓存机制

```
1 mysql> show global variables like "%query_cache_type%";
```

mysql 8.0已经移除了查询缓存功能

分析器

如果没有命中查询缓存，就要开始真正执行语句了。首先，MySQL 需要知道你要做什么，因此需要对 SQL 语句做解析。

分析器先会做“词法分析”。你输入的是由多个字符串和空格组成的一条 SQL 语句，MySQL 需要识别出里面的字符串分别是什么，代表什么。

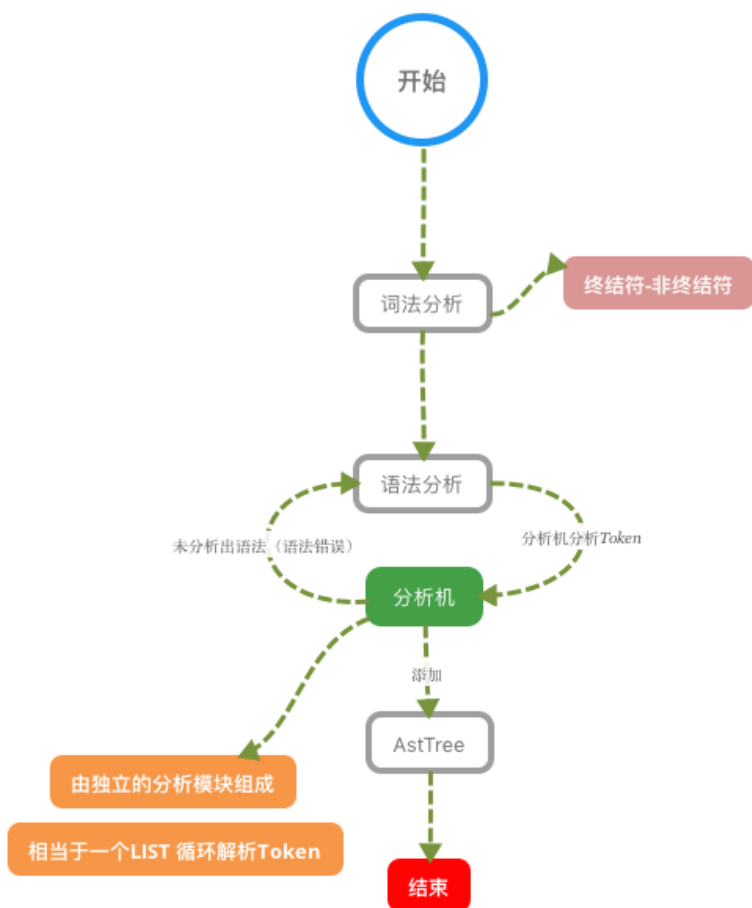
MySQL 从你输入的"select"这个关键字识别出来，这是一个查询语句。它也要把字符串“T”识别成“表名 T”，把字符串“ID”识别成“列 ID”。

做完了这些识别以后，就要做“语法分析”。根据词法分析的结果，语法分析器会根据语法规则，判断你输入的这个 SQL 语句是否满足 MySQL 语法。

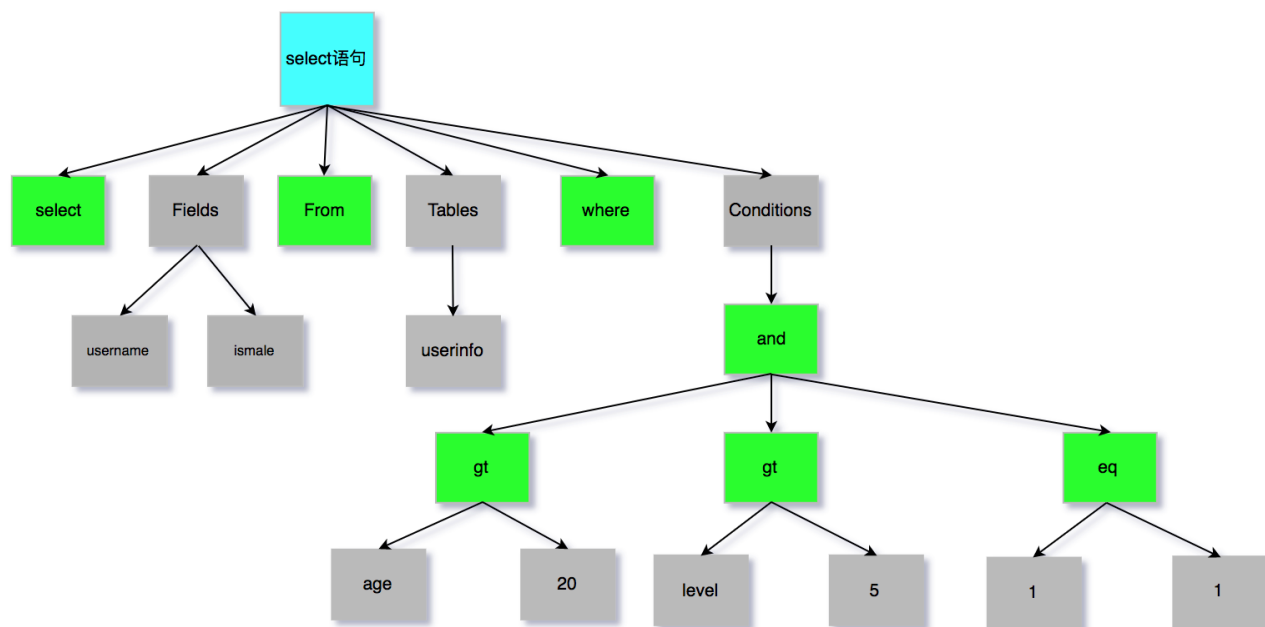
如果你的语句不对，就会收到“You have an error in your SQL syntax”的错误提醒，比如下面这个语句 from 写成了"rom"。

```
1 mysql> select * fro test where id=1;
2 ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'fro test where id=1' at line 1
```

下图是分析器对sql的分析过程步骤：



SQL语句经过分析器分析之后，会生成一个这样的语法树



至此我们分析器的工作任务也基本圆满了。接下来进入到优化器

优化器

经过了分析器，MySQL 就知道你要做什么了。在开始执行之前，还要先经过优化器的处理。

优化器是在表里面有多个索引的时候，决定使用哪个索引；或者在一个语句有多表关联（join）的时候，决定各个表的连接顺序；以及一些mysql自己内部的优化机制。

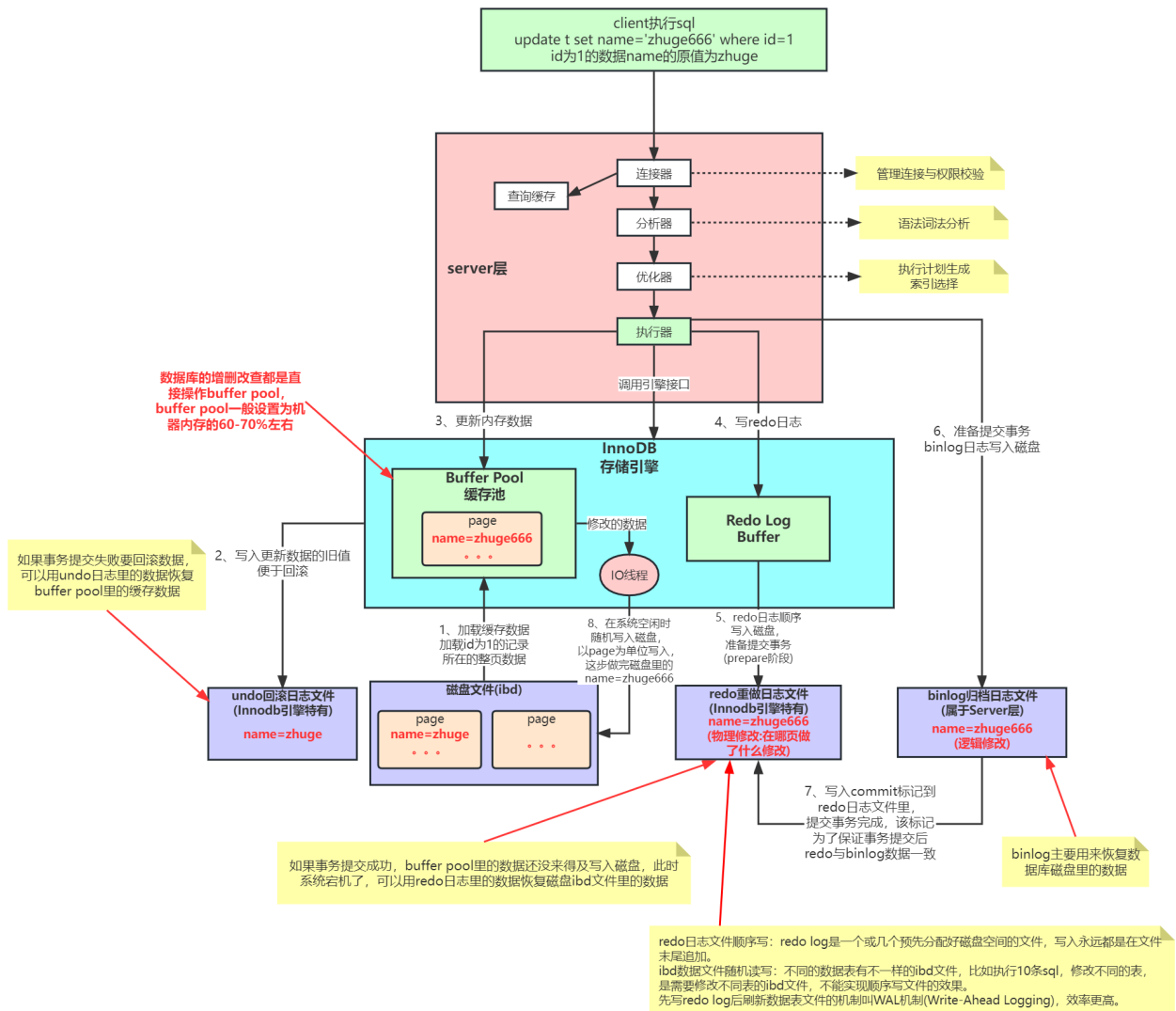
执行器

开始执行的时候，要先判断一下你对这个表 T 有没有执行查询的权限，如果没有，就会返回没有权限的错误，如下所示 (在工程实现上，如果命中查询缓存，会在查询缓存返回结果的时候，做权限验证)。

```
1 mysql> select * from test where id=10;
```

如果有权限，就打开表继续执行。打开表的时候，执行器就会根据表的引擎定义，去使用这个引擎提供的接口。

Innodb底层原理与Mysql日志机制



redo log重做日志关键参数

innodb_log_buffer_size: 设置redo log buffer大小参数，默认16M，最大值是4096M，最小值为1M。

```
1 show variables like '%innodb_log_buffer_size';
```

innodb_log_group_home_dir: 设置redo log文件存储位置参数，默认值为`./`，即innodb数据文件存储位置，其中的 `ib_logfile0` 和 `ib_logfile1` 即为redo log文件。

```
1 show variables like '%innodb_log_group_home_dir';
```

`ib_logfile0`

`ib_logfile1`

innodb_log_files_in_group: 设置redo log文件的个数，命名方式如: ib_logfile0, iblogfile1... iblogfileN。默认2个，最大100个。

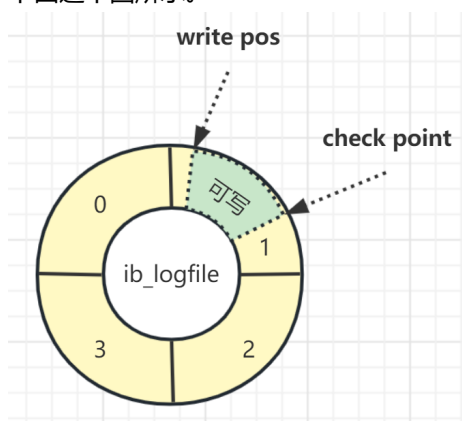
```
1 show variables like '%innodb_log_files_in_group%';
```

innodb_log_file_size: 设置单个redo log文件大小，默认值为48M。最大值为512G，注意最大值指的是整个 redo log系列文件之和，即(`innodb_log_files_in_group * innodb_log_file_size`)不能大于最大值512G。

```
1 show variables like '%innodb_log_file_size%';
```

redo log 写入磁盘过程分析:

redo log 从头开始写，写完一个文件继续写另一个文件，写到最后一个文件末尾就又回到第一个文件开头循环写，如下面这个图所示。



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数据文件里。

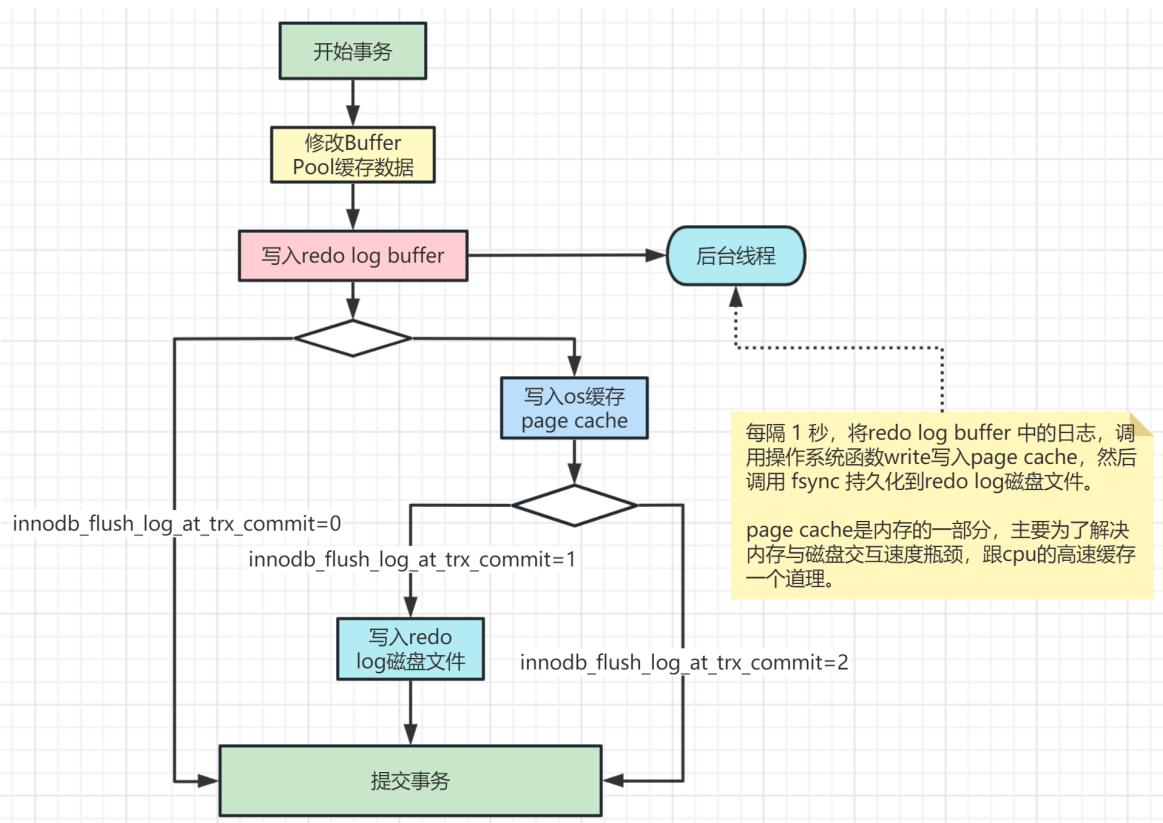
write pos 和 checkpoint 之间的部分就是空着的可写部分，可以用来记录新的操作。如果 write pos 追上 checkpoint，表示redo log写满了，这时候不能再执行新的更新，得停下来先擦掉一些记录，把 checkpoint 推进一下。

innodb_flush_log_at_trx_commit: 这个参数控制 redo log 的写入策略，它有三种可能取值：

- 设置为0: 表示每次事务提交时都只是把 redo log 留在 redo log buffer 中，数据库宕机可能会丢失数据。
- 设置为1(默认值): 表示每次事务提交时都将 redo log 直接持久化到磁盘，数据最安全，不会因为数据库宕机丢失数据，但是效率稍微差一点，线上系统推荐这个设置。
- 设置为2: 表示每次事务提交时都只是把 redo log 写到操作系统的缓存page cache里，这种情况如果数据库宕机是不会丢失数据的，但是操作系统如果宕机了，page cache里的数据还没来得及写入磁盘文件的话就会丢失数据。

InnoDB 有一个后台线程，每隔 1 秒，就会把 redo log buffer 中的日志，调用 操作系统函数 write 写到文件系统的 page cache，然后调用操作系统函数 fsync 持久化到磁盘文件。

redo log写入策略参看下图：



```

1 # 查看innodb_flush_log_at_trx_commit参数值:
2 show variables like 'innodb_flush_log_at_trx_commit';
3 # 设置innodb_flush_log_at_trx_commit参数值(也可以在my.ini或my.cnf文件里配置):
4 set global innodb_flush_log_at_trx_commit=1;

```

binlog二进制归档日志

binlog二进制日志记录保存了所有执行过的修改操作语句, 不保存查询操作。如果 MySQL 服务意外停止, 可通过二进制日志文件排查, 用户操作或表结构操作, 从而来恢复数据库数据。

启动binlog记录功能, 会影响服务器性能, 但如果需要恢复数据或主从复制功能, 则好处则大于对服务器的影响。

```

1 # 查看binlog相关参数
2 show variables like '%log_bin%';

```

```
mysql> show variables like '%log_bin%';
```

Variable_name	Value
log_bin	OFF
log_bin_basename	
log_bin_index	
log_bin_trust_function_creators	OFF
log_bin_use_vl_row_events	OFF
sql_log_bin	ON

MySQL5.7 版本中, binlog默认是关闭的, 8.0版本默认是打开的。上图中log_bin的值是OFF就代表binlog是关闭状态, 打开binlog功能, 需要修改配置文件my.ini(windows)或my.cnf(linux), 然后重启数据库。

在配置文件中的[mysqld]部分增加如下配置

```

1 # log-bin设置binlog的存放位置, 可以是绝对路径, 也可以是相对路径, 这里写的相对路径, 则binlog文件默认会放在data数据目录下
2 log-bin=mysql-binlog
3 # Server Id是数据库服务器id, 随便写一个数都可以, 这个id用来在mysql集群环境中标记唯一mysql服务器, 集群环境中每台mysql服务器的id不能一样, 不加启动会报错

```





```

4 server-id=1
5 # 其他配置
6 binlog_format = row # 日志文件格式，下面会详细解释
7 expire_logs_days = 15 # 执行自动删除binlog日志文件的天数， 默认为0， 表示不自动删除
8 max_binlog_size = 200M # 单个binlog日志文件的大小限制，默认为 1GB

```

重启数据库后我们再去看data数据目录会多出两个文件，第一个就是binlog日志文件，第二个是binlog文件的索引文件，这个文件管理了所有的binlog文件的目录。

 mysql-binlog.000001

 mysql-binlog.index

当然也可以执行命令查看有多少binlog文件

```
1 show binary logs;
```

```
1 show variables like '%log_bin%';
```

```
mysql> show variables like '%log_bin%';
```

Variable_name	Value
log_bin	ON
log_bin_basename	D:\dev\mysql-5.7.25-winx64\data\mysql-binlog
log_bin_index	D:\dev\mysql-5.7.25-winx64\data\mysql-binlog.index
log_bin_trust_function_creators	OFF
log_bin_use_vl_row_events	OFF
sql_log_bin	ON

1 log_bin: binlog日志是否打开状态

2 log_bin_basename: 是binlog日志的基本文件名，后面会追加标识来表示每一个文件，binlog日志文件会滚动增加

3 log_bin_index: 指定的是binlog文件的索引文件，这个文件管理了所有的binlog文件的目录。

4 sql_log_bin: sql语句是否写入binlog文件，ON代表需要写入，OFF代表不需要写入。如果想在主库上执行一些操作，但不复制到slave库上，可以通过修改参数sql_log_bin来实现。比如说，模拟主从同步复制异常。

binlog 的日志格式

用参数 binlog_format 可以设置binlog日志的记录格式，mysql支持三种格式类型：

- STATEMENT：基于SQL语句的复制，每一条会修改数据的sql都会记录到master机器的bin-log中，这种方式日志量小，节约IO开销，提高性能，但是对于一些执行过程中才能确定结果的函数，比如UUID()、SYSDATE()等函数如果随sql同步到slave机器去执行，则结果跟master机器执行的不一样。
- ROW：基于行的复制，日志中会记录成每一行数据被修改的形式，然后在slave端再对相同的数据进行修改记录下每一行数据修改的细节，可以解决函数、存储过程等在slave机器的复制问题，但这种方式日志量较大，性能不如Statement。举个例子，假设update语句更新10行数据，Statement方式就记录这条update语句，Row方式会记录被修改的10行数据。
- MIXED：混合模式复制，实际就是前两种模式的结合，在Mixed模式下，MySQL会根据执行的每一条具体的sql语句来区分对待记录的日志形式，也就是在Statement和Row之间选择一种，如果sql里有函数或一些在执行时才知道结果的情况，会选择Row，其它情况选择Statement，推荐使用这一种。

binlog写入磁盘机制

binlog写入磁盘机制主要通过 sync_binlog 参数控制，默认值是 0。

- 为0的时候，表示每次提交事务都只 write 到page cache，由系统自行判断什么时候执行 fsync 写入磁盘。虽然性能得到提升，但是机器宕机，page cache里面的 binlog 会丢失。
- 也可以设置为1，表示每次提交事务都会执行 fsync 写入磁盘，这种方式最安全。
- 还有一种折中方式，可以设置为N(N>1)，表示每次提交事务都write 到page cache，但累积N个事务后才 fsync 写入磁盘，这种如果机器宕机会丢失N个事务的binlog。

发生以下任何事件时，binlog日志文件会重新生成：

- 服务器启动或重新启动
- 服务器刷新日志，执行命令flush logs
- 日志文件大小达到 max_binlog_size 值，默认值为 1GB

删除 binlog 日志文件

```
1 删除当前的binlog文件
2 reset master;
3 # 删除指定日志文件之前的所有日志文件，下面这个是删除6之前的所有日志文件，当前这个文件不删除
4 purge master logs to 'mysql-binlog.000006';
5 # 删除指定日期前的日志索引中binlog日志文件
6 purge master logs before '2023-01-21 14:00:00';
```

查看 binlog 日志文件

可以用mysql自带的命令工具 mysqlbinlog 查看binlog日志内容

```
1 # 查看bin-log二进制文件（命令行方式，不用登录mysql）
2 mysqlbinlog --no-defaults -v --base64-output=decode-rows D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000007
3
4 # 查看bin-log二进制文件（带查询条件）
5 mysqlbinlog --no-defaults -v --base64-output=decode-rows D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000007 start-datetime="2023-01-21 00:00:00" stop-datetime="2023-02-01 00:00:00" start-position="5000" stop-position="20000"
```

执行mysqlbinlog命令

```
1 mysqlbinlog --no-defaults -v --base64-output=decode-rows D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000007
```

查出来的binlog日志文件内容如下：

```
1
2 /*!50530 SET @@SESSION.PSEUDO_SLAVE_MODE=1*/;
3 /*!50003 SET @@OLD_COMPLETION_TYPE=@@COMPLETION_TYPE,COMPLETION_TYPE=0*/;
4 DELIMITER /*!*/;
5 # at 4
6 #230127 21:13:51 server id 1 end_log_pos 123 CRC32 0x084f390f Start: binlog v 4, server v 5.7.25-log created 230127 21:13:51 at startup
7 # Warning: this binlog is either in use or was not closed properly.
8 ROLLBACK/*!*/;
9 # at 123
10 #230127 21:13:51 server id 1 end_log_pos 154 CRC32 0x672ba207 Previous-GTIDs
11 # [empty]
12 # at 154
13 #230127 21:22:48 server id 1 end_log_pos 219 CRC32 0x8349d010 Anonymous_GTID last_committed=0 sequence_number=1 rbr_only=yes
14 /*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED/*!*/;
15 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
16 # at 219
17 #230127 21:22:48 server id 1 end_log_pos 291 CRC32 0xbf49de02 Query thread_id=3 exec_time=0 error_code=0
18 SET TIMESTAMP=1674825768/*!*/;
19 SET @@session.pseudo_thread_id=3/*!*/;
20 SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
21 SET @@session.sql_mode=1342177280/*!*/;
22 SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
23 /*!\C utf8 *//*!*/;
```



```

24 SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=33/*!*/;
25 SET @@session.lc_time_names=0/*!*/;
26 SET @@session.collation_database=DEFAULT/*!*/;
27 BEGIN
28 /*!*/;
29 # at 291
30 #230127 21:22:48 server id 1 end_log_pos 345 CRC32 0xc4ab653e Table_map: `test`.`account` mapped to number 99
31 # at 345
32 #230127 21:22:48 server id 1 end_log_pos 413 CRC32 0x54a124bd Update_rows: table id 99 flags: STMT_END_F
33 ### UPDATE `test`.`account`
34 ### WHERE
35 ### @1=1
36 ### @2='lilei'
37 ### @3=1000
38 ### SET
39 ### @1=1
40 ### @2='lilei'
41 ### @3=2000
42 # at 413
43 #230127 21:22:48 server id 1 end_log_pos 444 CRC32 0x23355595 Xid = 10
44 COMMIT/*!*/;
45 # at 444
46 ...

```

能看到里面有具体执行的修改伪sql语句以及执行时的相关情况。

binlog日志文件恢复数据

用binlog日志文件恢复数据其实就是回放执行之前记录在binlog文件里的sql，举一个数据恢复的例子

```

1 # 先执行刷新日志的命令生成一个新的binlog文件mysql-binlog.000008，后面我们的修改操作日志都会记录在最新的这个文件里
2 flush logs;
3 # 执行两条插入语句
4 INSERT INTO `test`.`account` (`id`, `name`, `balance`) VALUES ('4', 'zhuge', '666');
5 INSERT INTO `test`.`account` (`id`, `name`, `balance`) VALUES ('5', 'zhuge1', '888');
6 # 假设现在误操作执行了一条删除语句把刚新增的两条数据删掉了
7 delete from account where id > 3;

```

现在需要恢复被删除的两条数据，我们先查看binlog日志文件

```

1 mysqlbinlog --no-defaults -v --base64-output=decode-rows D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000008

```

文件内容如下：

```

1 .....
2 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
3 # at 219
4 #230127 23:32:24 server id 1 end_log_pos 291 CRC32 0x4528234f Query thread_id=5 exec_time=0 error_code=0
5 SET TIMESTAMP=1674833544/*!*/;
6 SET @@session.pseudo_thread_id=5/*!*/;
7 SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
8 SET @@session.sql_mode=1342177280/*!*/;

```

```

9 SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
10 /*!\C utf8 *//*!*/;
11 SET @@session.character_set_client=33,@@session.collation_connection=33,@@session.collation_server=33/*!*/;
12 SET @@session.lc_time_names=0/*!*/;
13 SET @@session.collation_database=DEFAULT/*!*/;
14 BEGIN
15 /*!*/;
16 # at 291
17 #230127 23:32:24 server id 1 end_log_pos 345 CRC32 0x7482741d Table_map: `test`.`account` mapped to number 99
18 # at 345
19 #230127 23:32:24 server id 1 end_log_pos 396 CRC32 0x5e443cf0 Write_rows: table id 99 flags: STM T_END_F
20 ### INSERT INTO `test`.`account`
21 ### SET
22 ### @1=4
23 ### @2='zhuge'
24 ### @3=666
25 # at 396
26 #230127 23:32:24 server id 1 end_log_pos 427 CRC32 0x8a0d8a3c Xid = 56
27 COMMIT/*!*/;
28 # at 427
29 #230127 23:32:40 server id 1 end_log_pos 492 CRC32 0x5261a37e Anonymous_GTID last_committed=1 sequence_number=2 rbr_only=yes
30 /*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED*//*!*/;
31 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
32 # at 492
33 #230127 23:32:40 server id 1 end_log_pos 564 CRC32 0x01086643 Query thread_id=5 exec_time=0 error_code=0
34 SET TIMESTAMP=1674833560/*!*/;
35 BEGIN
36 /*!*/;
37 # at 564
38 #230127 23:32:40 server id 1 end_log_pos 618 CRC32 0xc26b6719 Table_map: `test`.`account` mapped to number 99
39 # at 618
40 #230127 23:32:40 server id 1 end_log_pos 670 CRC32 0x8e272176 Write_rows: table id 99 flags: STM T_END_F
41 ### INSERT INTO `test`.`account`
42 ### SET
43 ### @1=5
44 ### @2='zhuge1'
45 ### @3=888
46 # at 670
47 #230127 23:32:40 server id 1 end_log_pos 701 CRC32 0xb5e63d00 Xid = 58
48 COMMIT/*!*/;
49 # at 701
50 #230127 23:34:23 server id 1 end_log_pos 766 CRC32 0xa0844501 Anonymous_GTID last_committed=2 sequence_number=3 rbr_only=yes
51 /*!50718 SET TRANSACTION ISOLATION LEVEL READ COMMITTED*//*!*/;
52 SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
53 # at 766

```

```

54 #230127 23:34:23 server id 1 end_log_pos 838 CRC32 0x687bdf88 Query thread_id=7 exec_time=0 error_code=0
55 SET TIMESTAMP=1674833663/*!*/;
56 BEGIN
57 /*!*/;
58 # at 838
59 #230127 23:34:23 server id 1 end_log_pos 892 CRC32 0x4f7b7d6a Table_map: `test`.`account` mapped to number 99
60 # at 892
61 #230127 23:34:23 server id 1 end_log_pos 960 CRC32 0xc47ac777 Delete_rows: table id 99 flags: STMT_END_F
62 ### DELETE FROM `test`.`account`
63 ### WHERE
64 ### @1=4
65 ### @2='zhuge'
66 ### @3=666
67 ### DELETE FROM `test`.`account`
68 ### WHERE
69 ### @1=5
70 ### @2='zhuge1'
71 ### @3=888
72 # at 960
73 #230127 23:34:23 server id 1 end_log_pos 991 CRC32 0x386699fe Xid = 65
74 COMMIT/*!*/;
75 SET @@SESSION.GTID_NEXT= 'AUTOMATIC' /* added by mysqlbinlog */ /*!*/;
76 DELIMITER ;
77 # End of log file
78 . . . . .

```

找到两条插入数据的sql，每条sql的上下都有BEGIN和COMMIT，我们找到第一条sql BEGIN前面的文件位置标识 at 219(这是文件的位置标识)，再找到第二条sql COMMIT后面的文件位置标识 at 701

我们可以根据文件位置标识来恢复数据，执行如下sql：

```

1 mysqlbinlog --no-defaults --start-position=219 --stop-position=701 --database=test D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000009 | mysql -uroot -p123456 -v test
2
3 # 补充一个根据时间来恢复数据的命令，我们找到第一条sql BEGIN前面的时间戳标记 SET TIMESTAMP=1674833544，再找到第二条sql COMMIT后面的时间戳标记 SET TIMESTAMP=1674833663，转成datetime格式
4 mysqlbinlog --no-defaults --start-datetime="2023-1-27 23:32:24" --stop-datetime="2023-1-27 23:34:23" --database=test D:/dev/mysql-5.7.25-win64/data/mysql-binlog.000009 | mysql -uroot -p123456 -v test

```

被删除数据被恢复！

注意：如果要恢复大量数据，比如程序员经常说的删库跑路的话题，假设我们把数据库所有数据都删除了要怎么恢复了，如果数据库之前没有备份，所有的binlog日志都在的话，就从binlog第一个文件开始逐个恢复每个binlog文件里的数据，这种一般不太可能，因为binlog日志比较大，早期的binlog文件会定期删除的，所以一般不可能用binlog文件恢复整个数据库的。

一般我们推荐的是每天(在凌晨后)需要做一次全量数据库备份，那么恢复数据库可以用最近的一次全量备份再加上备份时间点之后的binlog来恢复数据。

备份数据库一般可以用mysqldump 命令工具

```

1 mysqldump -u root 数据库名 > 备份文件名; #备份整个数据库
2 mysqldump -u root 数据库名 表名字 > 备份文件名; #备份整个表
3
4 mysql -u root test < 备份文件名 #恢复整个数据库，test为数据库名称，需要自己先建一个数据库test

```

为什么会有redo log和binlog两份日志呢？

因为最开始 MySQL 里并没有 InnoDB 引擎。MySQL 自带的引擎是 MyISAM，但是 MyISAM 没有 crash-safe 的能力，binlog 日志只能用于归档。而 InnoDB 是另一个公司以插件形式引入 MySQL 的，既然只依靠 binlog 是没有 crash-safe 能力的，所以 InnoDB 使用另外一套日志系统——也就是 redo log 来实现 crash-safe 能力。

有了 redo log，InnoDB 就可以保证即使数据库发生异常重启，之前提交的记录都不会丢失，这个能力称为 crash-safe。

undo log回滚日志

InnoDB对undo log文件的管理采用段的方式，也就是回滚段（rollback segment）。每个回滚段记录了 1024 个 undo log segment，每个事务只会使用一个undo log segment。

在MySQL5.5的时候，只有一个回滚段，那么最大同时支持的事务数量为1024个。在MySQL 5.6开始，InnoDB支持最大 128个回滚段，故其支持同时在线的事务限制提高到了 128*1024。

- 1 innodb_undo_directory: 设置undo log文件所在的路径。该参数的默认值为"./"，即innodb数据文件存储位置，目录下ibdata1文件就是undo log存储的位置。
- 2 innodb_undo_logs: 设置undo log文件内部回滚段的个数，默认值为128。
- 3 innodb_undo_tablespaces: 设置undo log文件的数量，这样回滚段可以较为平均地分布在多个文件中。设置该参数后，会在路径innodb_undo_directory看到undo为前缀的文件。

undo log日志什么时候删除

新增类型的，在事务提交之后就可以清除掉了。

修改类型的，事务提交之后不能立即清除掉，这些日志会用于mvcc。只有当没有事务用到该版本信息时才可以清除。

为什么Mysql不能直接更新磁盘上的数据而且设置这么一套复杂的机制来执行SQL了？

因为来一个请求就直接对磁盘文件进行随机读写，然后更新磁盘文件里的数据性能可能相当差。

因为磁盘随机读写的性能是非常差的，所以直接更新磁盘文件是不能让数据库抗住很高并发的。

Mysql这套机制看起来复杂，但它可以保证每个更新请求都是**更新内存BufferPool**，然后**顺序写日志文件**，同时还能保证各种异常情况下的数据一致性。

更新内存的性能是极高的，然后顺序写磁盘上的日志文件的性能也是非常高的，要远高于随机读写磁盘文件。

正是通过这套机制，才能让我们的MySQL数据库在较高配置的机器上每秒可以抗下几千甚至上万的读写请求。

错误日志

Mysql还有一个比较重要的日志是错误日志，它记录了数据库启动和停止，以及运行过程中发生任何严重错误时的相关信息。当数据库出现任何故障导致无法正常使用时，建议首先查看此日志。

在MySQL数据库中，错误日志功能是默认开启的，而且无法被关闭。

- ```
1 # 查看错误日志存放位置
2 show variables like '%log_error%';
```

## 通用查询日志

通用查询日志记录用户的所有操作，包括启动和关闭MySQL服务、所有用户的连接开始时间和截止时间、发给

MySQL 数据库服务器的所有 SQL 指令等，如select、show等，无论SQL的语法正确还是错误、也无论SQL执行成功

还是失败，MySQL都会将其记录下来。

通用查询日志用来还原操作时的具体场景，可以帮助我们准确定位一些疑难问题，比如重复支付等问题。

**general\_log**：是否开启日志参数，默认为OFF，处于关闭状态，因为开启会消耗系统资源并且占用磁盘空间。一般不建议开启，只在需要调试查询问题时开启。

**general\_log\_file**：通用查询日志记录的位置参数。

```
1 show variables like '%general_log%';
2 # 打开通用查询日志
3 SET GLOBAL general_log=on;
```

```
1 文档: 07-VIP-Innodb底层原理与Mysql日志机制深入剖析
2 链接: http://note.youdao.com/noteshare?id=f030268c54f18d2116837f8f3ef045bf&sub=4D84722956BB4A4BBB6C42A1D337BCAA
```