# 概述

# 分类

## MySQL HeatWave

## MySQL Enterprise

## MySQL Standard Edition

## MySQL Classic Edition

## MySQL Cluster CGE

## MySQL Embedded（OEM/ISV）

# 安装配置

# 架构

参考：<https://cloud.tencent.com/developer/article/2119477>

# 原理

## Server层

## 存储引擎

参考：

<https://blog.csdn.net/lzb348110175/article/details/106555504>

Secondary Engine：<http://mysql.taobao.org/monthly/2020/11/04/>

一些流行引擎存储格式简介：<http://mysql.taobao.org/monthly/2017/10/04/>

### InnoDB

### MyISAM

### Archive

### Blackhole

### CSV

参考：

CSV 引擎详解：<http://mysql.taobao.org/monthly/2021/10/05/>

### temptable

### TokuDB

参考：<https://article.itxueyuan.com/0e3Xkp>

### Federated

### heap

### rocksdb

### Spider

参考：<https://cloud.tencent.com/developer/article/1992931>

## 主要数据结构

参考：

语法解析调用栈：<https://www.cnblogs.com/jkin/p/16848540.html>

### THD

代码位置：sql/sql\_class.h

包含处理用户请求时需要的相关数据，每个连接会有一个线程来处理，在一些高层函数中，此数据结构常被当作第一个参数传递。

THD类继承了三个父类：

MDL\_context\_owner：作为Metadata Lock 拥有者相关的接口，其中元数据边界控制（进/退），元数据信息通知等，它是一个虚类。

Query\_arena：管理语句元素的类，主要针对指语句（包括store procedure）解释成抽象语法树后的节点

Open\_tables\_state：管理线程打开和加锁的表的状态，保持了一个临时表信息和锁信息，子类有Open\_tables\_backup

除这些外，还包括与锁相关的数据结构如MDL\_context，接口语法语义相关如LEX，日志和事务以及执行数据和优化参数等。

NET net; // 客户连接描述符

Protocol \*protocol; // 当前的协议

Protocol\_text protocol\_text; // 普通协议

Protocol\_binary protocol\_binary; // 二进制协议

HASH user\_vars; //用户变量的hash值

String packet; // 网络IO时所用的缓存

String convert\_buffer; // 字符集转换所用的缓存

struct sockaddr\_in remote; //客户端socket地址

THR\_LOCK\_INFO lock\_info; // 当前线程的锁信息

THR\_LOCK\_OWNER main\_lock\_id; // 在旧版的查询中使用

THR\_LOCK\_OWNER \*lock\_id; //若非main\_lock\_id, 指向游标的lock\_id

pthread\_mutex\_t LOCK\_thd\_data;

//thd的mutex锁，保护THD数据（thd->query, thd->query\_length）不会被其余线程访问到

Statement\_map stmt\_map;

//prepared statements和stored routines 会被重复利用

int insert(THD \*thd, Statement \*statement); // statement的hash容器

class Statement::

LEX\_STRING name;

LEX \*lex; //语法树描述符

bool set\_db(const char \*new\_db, size\_t new\_db\_len)

void set\_query(char \*query\_arg, uint32 query\_length\_arg);

{

pthread\_mutex\_lock(&LOCK\_thd\_data);

set\_query\_inner(query\_arg, query\_length\_arg);

pthread\_mutex\_unlock(&LOCK\_thd\_data);

}

### Lex

MySQL通过THD的父类字段Statement::lex保存解析出来的语法树信息。struct LEX继承于Query\_tables\_list。LEX中涉及的查询语法树的核心成员包括：

unit：语法树顶层节点；

select\_lex：顶层第一个select语句节点；

all\_selects\_list：通过link\_next和link\_prev构成所有select子句的双向链表；

query\_tables：通过next\_global和prev\_global构成所有talbe的双向链表；

query\_tables\_last：指向query\_tables中最后一个元素；

注：其他成员不作描述，在sql/sql\_lex.h源码文件中。

#### select语法树

##### 核心数据结构

select语法树由SELECT\_LEX\_UNIT（即st\_select\_lex\_unit）节点和SELECT\_LEX（即st\_select\_lex）节点组成。两个class都继承自st\_select\_lex\_node。SELECT\_LEX的核心成员如下：

table\_list：通过next\_local串联起来的表结构，也就是from后面的表；

top\_join\_list：将from后面的表根据jion语法规则生成的jion树；

where：where语句后面的条件表示树；

group\_list：group分组条件；

having：having条件；

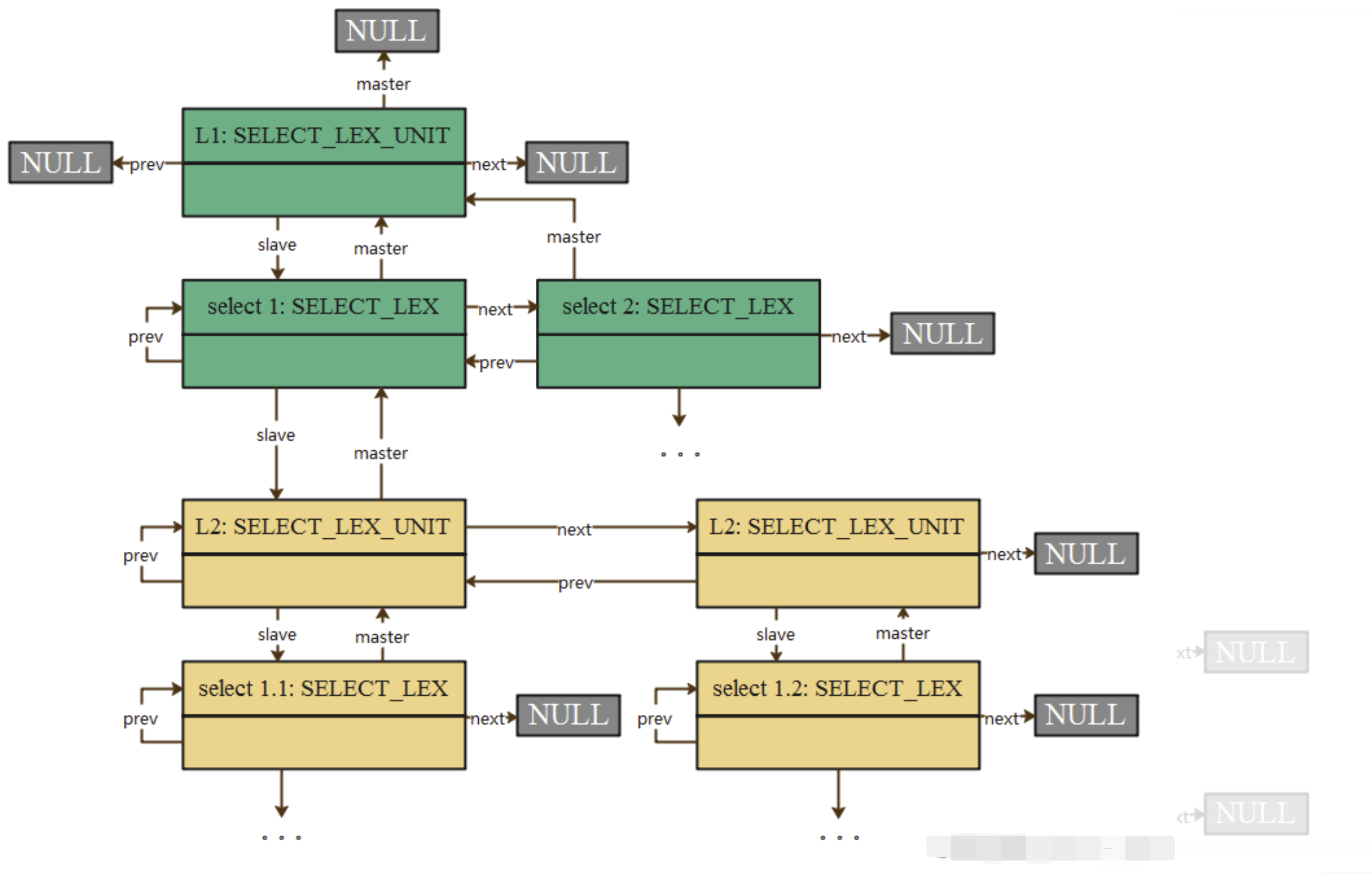
item\_list：select语句后面的字段列表；

order\_list：order排序方式；

select\_limit，offset\_limit：行限制条件；

##### 语法树结构

查询语句的语法树结构如下：



整个语法树由SELECT\_LEX\_UNIT作为顶层，下面挂unit的多个select语句；每个select子句的顶层SELECT\_LEX\_UNIT挂到所属的select主句上面。THD::lex->unit指向顶层L1；THD::lex->select\_lex指向顶层select 1。

#### from子句

##### 解析原理

整个from表格解析规则如下：

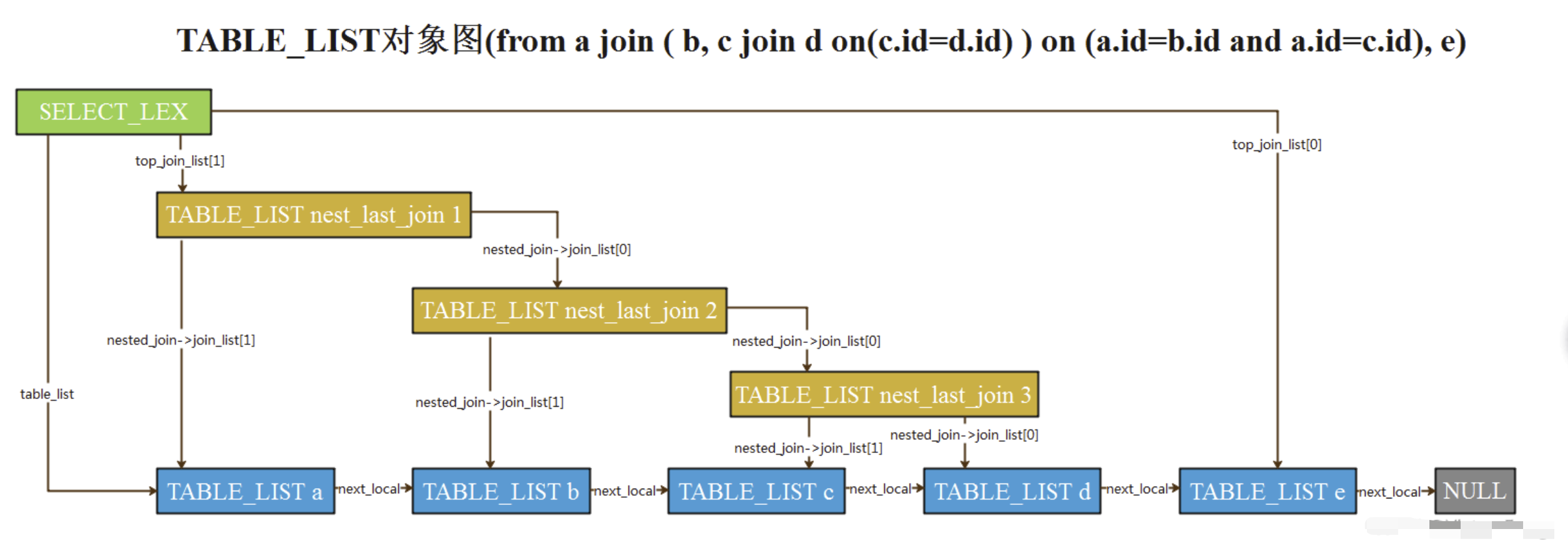
规则1：JOIN表达式：=（JOIN表达式 | JOIN嵌套）JOIN（JOIN表达式 | JOIN嵌套）

规则2：JOIN嵌套：=’(’ （表引用 | JOIN表达式）（，表引用 | ，JOIN表达式）\* ‘)’ | 表引用

也就是说一个jion表达式是由join表达式或join嵌套组成；一个join嵌套可以是一个表引用，或者多个表引用或join表达式通过“,”相隔、且首尾括号包围。

##### 语法树结构

from后面的表解析成TABLE\_LIST结构，所有的TABLE\_LIST按如下方式串联：



#### where语句

where语句都是抽象类Item的子类一层一层组成语法树结构。

### NET

**代码位置：**sql/mysql\_com.h

网络连接描述符，对内部数据包进行了封装，是client和server之间的通信协议。

Vio \*vio; //底层的网络I/O socket描述符

unsigned char \*buff,\*buff\_end,\*write\_pos,\*read\_pos; //缓存相关

unsigned long remain\_in\_buf,length, buf\_length, where\_b;

unsigned long max\_packet,max\_packet\_size; //当前值;最大值

unsigned int pkt\_nr,compress\_pkt\_nr; //当前（未）压缩包的顺序值

my\_bool compress; //是否压缩

unsigned int write\_timeout, read\_timeout, retry\_count; //最大等待时间

unsigned int \*return\_status; //thd中的服务器状态

unsigned char reading\_or\_writing;

unsigned int last\_errno; //返回给客户端的错误号

unsigned char error;

### TABLE

参考：

TABLE信息的生命周期：<http://mysql.taobao.org/monthly/2022/01/04/>

**代码位置：**sql/table.h

数据库表描述符，分成TABLE和TABLE\_SHARE两部分。

handler \*file; //指向这张表在storage engine中的handler的指针

THD \*in\_use;

Field \*\*field;

uchar \*record[2];

uchar \*write\_row\_record;

uchar \*insert\_values;

key\_map covering\_keys;

key\_map quick\_keys, merge\_keys;

key\_map keys\_in\_use\_for\_query;

key\_map keys\_in\_use\_for\_group\_by;

key\_map keys\_in\_use\_for\_order\_by;

KEY \*key\_info;

HASH name\_hash; //数据域名字的hash值

MEM\_ROOT mem\_root; //内存块

LEX\_STRING db;

LEX\_STRING table\_name;

LEX\_STRING table\_cache\_key;

enum db\_type db\_type //当前表的storage engine类型

enum row\_type row\_type //当前记录是定长还是变长

uint primary\_key;

uint next\_number\_index; //自动增长key的值

bool is\_view ;

bool crashed;

### FIELD

**代码位置：**sql/field.h

域描述符，是各种字段的抽象基类。

uchar \*ptr; // 记录中数据域的位置

uchar \*null\_ptr; // 记录 null\_bit 位置的byte

TABLE \*table; // 指向表的指针

TABLE \*orig\_table; // 指向原表的指针

const char \*\*table\_name, \*field\_name;

LEX\_STRING comment;

key\_map key\_start, part\_of\_key, part\_of\_key\_not\_clustered;

key\_map part\_of\_sortkey;

enum utype { NONE,DATE,SHIELD,NOEMPTY,CASEUP,PNR,BGNR,PGNR,YES,NO,REL,

CHECK,EMPTY,UNKNOWN\_FIELD,CASEDN,NEXT\_NUMBER,

INTERVAL\_FIELD,BIT\_FIELD, TIMESTAMP\_OLD\_FIELD, CAPITALIZE, BLOB\_FIELD,

TIMESTAMP\_DN\_FIELD, TIMESTAMP\_UN\_FIELD, TIMESTAMP\_DNUN\_FIELD};

…..

virtual int store(const char \*to, uint length,CHARSET\_INFO \*cs)=0;

inline String \*val\_str(String \*str) { return val\_str(str, str); }

### Utility API Calls API

各种核心的工具，例如内存分配，字符串操作或文件管理。标准C库中的函数只使用了很少一部分，C++中的函数基本没用。

void \*my\_malloc(size\_t size, myf my\_flags)

//对malloc的封装

size\_t my\_write(File Filedes, const uchar \*Buffer, size\_t Count, myf MyFlags)

//对write的封装

### Preprocessor Macros处理器宏

Mysql中使用了大量的C预编译，随编译参数的不同最终代码也不同。

#define max(a, b) ((a) > (b) ? (a) : (b)) //得出两数中的大者

do \

{ \

char compile\_time\_assert[(X) ? 1 : -1] \

\_\_attribute\_\_ ((unused)); \

} while(0)

使用gcc的attribute属性指导编译器行为

• Global variables 全局变量

• configuration settings

• server status information

• various data structures shared among threads

主要包括一些全局的设置，服务器信息和部分线程间共享的数据结构。

struct system\_status\_var global\_status\_var; //全局的状态信息

struct system\_variables global\_system\_variables; //全局系统变量

# 源码

源码结构（MySQL-5.5.0-m2）

BUILD: 内含在各个平台、各种编译器下进行编译的脚本。如compile-pentium-debug表示在pentium架构上进行编译的脚本。

client: 客户端工具，如mysql, mysqladmin之类。

cmd-line-utils: readline, libedit工具。

config: 给aclocal使用的配置文件。

dbug: 提供一些调试用的宏定义。

extra: 提供innochecksum，resolveip等额外的小工具。

include: 包含的头文件。

libmysql: 库文件，生产libmysqlclient.so。

libmysql\_r: 线程安全的库文件，生成libmysqlclient\_r.so。

libservices: 5.5.0中新加的目录，实现了打印功能。

man: 手册页。

mysql-test: mysqld的测试工具一套。

mysys: 为跨平台计，MySQL自己实现了一套常用的数据结构和算法，如string, hash等。

netware: 在netware平台上进行编译时需要的工具和库。

plugin: mysql以插件形式实现的部分功能。

pstack: 异步栈追踪工具。

regex: 正则表达式工具。

Scripts: 提供脚本工具，如mysql\_install\_db等

sql: mysql主要代码，将会生成mysqld文件。

sql-bench: 一些评测代码。

sql-common: 存放部分服务器端和客户端都会用到的代码。

storage: 存储引擎所在目录，如myisam, innodb, ndb等。

这个目录包含了所谓的MySQL存储引擎（storage engine）。存储引擎是数据库系统的核心，封装了数据库文件的操作，是数据库系统是否强大最重要的因素。MySQL实现了一个抽象接口层，叫做handler（sql/handler.h），其中定义了接口函数，比如：ha\_open，ha\_index\_end，ha\_create等等，存储引擎需要实现这些接口才能被系统使用。对于具体每种引擎的特点，推荐大家去看mysql的在线文档: <http://dev.mysql.com/doc/refman/5.1/en/storage-engines.html>。

Strings: string库。

support-files: my.cnf示例配置文件。

tests: 测试文件所在目录。

unittest: 单元测试。

vio: virtual io系统，是对network io的封装。

win: 给windows平台提供的编译环境。

zip: zip库工具

## client

## include

## plugin

参考：

<https://blog.csdn.net/fpcc/article/details/120596120>

<https://www.cnblogs.com/zhedan/p/12518091.html>

### audit\_log

### auth

### autl\_ldap

### binlog\_utils\_udf

### clone

### connection\_ctrol

### data\_masking

### ddl\_rewriter

### fulltext

### froup\_replication

### innodb\_memcached

### keyring

### keyring\_udf

### keyring\_vault

### password\_validation

### percona-udf

### pfs\_table\_plugin

### procfs

### rewiter

### semisync

### tokudb-backup-plugin

### udf\_services

### version\_token

### x

## router

### harness

### http

### io

### json\_schema\_embedder

### keepalive

### metadata\_cache

### mock\_server

### mysql\_protocol

### openssl

### plugin\_info

### protobuf

### rest\_api

### rest\_metadata\_cache

### rest\_router

### rest\_routing

### router

### routing

## scripts

## sql

### auth

### binlong

### chagestreams

### conn\_handler

### daemon\_proxy\_keyring

### dd

#### cache

#### impl

#### raw

#### system\_views

#### tables

#### types

#### upgrade

#### dd.cc

#### distionary\_impl.cc

#### distionary\_impl.h

#### object\_key.h

#### properties\_impl.cc

#### properties\_impl.h

#### sdi\_api.cc

#### sdi\_file.cc

#### sdi\_file.h

#### sdi\_impl.h

#### sdi\_tablespace.cc

#### sdi\_tablespace.h

#### sdi\_utils.h

#### sdi.cc

#### sdi.h

操作持久化层（磁盘）的接口，主要是Storage\_adapter调用

#### string\_type.cc

#### system\_registry.cc

#### system\_registry.h

#### transaction\_impl.cc

#### transaction\_impl.h

#### utils.cc

#### utils.h

### gis

### join\_optimizer

#### access\_path.cc

##### NewSortAccessPath

##### GetUsedTables

##### CreateIteratorFromAccessPath

根据前面构造的AccessPath，构造对应的迭代器Iterator。

##### FindTablesToGetRowidFor

##### ExpandFilterAccessPaths

功能：

调用：sql/join\_optimizer/join\_optimizer.cc/FindBestQueryPlan

###### WalkAccessPaths

###### ConditionFromFilterPredicates

#### access\_path.h

#### bit\_utils.h

#### estimate\_selectivity.cc

#### estimate\_selectivity.h

#### explain\_access\_path.cc

##### ExplainAccessPath

##### PrintQueryPlan

功能：主要是opt\_explain.cc中接口的实现（获取AccessPath相关信息）

调用：

ExplainIterator（sql/opt\_explain.cc）

SELECT\_LEX\_UNIT::optimize（sql/sql\_union.cc）

#### explain\_access\_path.h

#### hypergraph.cc

#### hypergraph.h

#### join\_optimizer.cc

##### FindBestQueryPlan

功能：如果开启了超图算法优化，则调用FindBestQueryPlan，否则调用左深树优化FindBestQueryPlan。

调用：

JOIN::optimize（sql/sql\_optimizer.cc）

-> FindBestQueryPlan

#### join\_optimizer.h

#### make\_join\_hypergraph.cc

#### make\_join\_hypergraph.h

#### materialize\_path\_parameters.h

#### print\_utils.cc

#### print\_utils.h

#### subgraph\_enumeration.h

#### walk\_access\_paths.h

##### WalkAccessPaths

调用：

FindSingleAccessPathOfType（sql/join\_optimizer/access\_path.cc）

FindTablesToGetRowidFor（sql/join\_optimizer/access\_path.cc）

ExpandFilterAccessPaths（sql/join\_optimizer/access\_path.cc）

FindBestQueryPlan（sql/join\_optimizer/join\_optimizer.cc）

-> WalkAccessPaths

WalkAccessPaths：遍历下面的每个访问路径path，通过前序或后序遍历为每个访问路径调用func()。如果func()返回true,则遍历不会深入到当前路径的child。对于后序遍历,当调用func()时，child已经被遍历，因此跳过它们为时已晚，并且func()的返回值实际上被忽略。

join参数表示查询块属于哪个查询块，因为路径本身并不是隐式的.

该函数将跟踪它在整个树中的变化（在MATERIALIZE或STREAM访问路径中），并向func()回调提供正确的值。

如果策略是ENTIRE\_QUERY\_BLOCK，则仅由WalkAccessPaths()本身调用；如果不是，它仅用于func()回调，也可以将其设置为nullptr。

### locks

### memory

### partitioning

### raii

### regexp

### resourcegroups

### server\_somponent

### abstract\_query\_plan.cc

主要是Table\_access（表访问session）的封装接口。

### abstract\_query\_plan.h

#### class Table\_access

### aggregate\_check.cc

封装了聚合函数检查的接口：

Distinct\_check

Group\_check

### aggregate\_check.h

#### class Distinct\_check

#### class Group\_check

### auto\_thd.cc

### auto\_thd.h

### basic\_istream.cc

### basic\_istream.h

### basic\_ostream.cc

### basic\_ostream.h

### basic\_row\_iterator.h

封装了基本的迭代器Iterator操作，继承关系如下：

RowIterator（row\_iterator.h）

-> TableRowIterator（row\_iterator.h）

-> TableScanIterator

#### class TableScanIterator

#### class IndexScanIterator

#### class IndexRangeScanIterator

#### class SortBufferIterator

#### class SortBufferIndirectIterator

#### class SortFileIterator

#### class SortFileIndirectIterator

#### class FakeSingleRowIterator

#### class UnqualifiedCountIterator

#### class ZeroRowsIterator

#### class ZeroRowsAggregatedIterator

#### class FollowTailIterator

#### class TableValueConstructorIterator

### binlog\_istream.cc

### binlog\_istream.h

### binlog\_ostream.cc

### binlog\_ostream.h

### binlog\_reader.cc

### binlog\_reader.h

### binlog.cc

### binlog.h

### bka\_iterator.cc

BKA join迭代器实现。

### bka\_iterator.h

#### BKAIterator

### bootstrap.cc

### bootstrap.h

#### run\_bootstrap\_thread

### bounded\_queue.h

### check\_stack.cc

### check\_stack.h

#### check\_stack\_overrun

### client\_settings.h

### clone\_handler.cc

### clone\_handler.h

### cmp\_varlen\_keys.h

### command\_serviece.cc

### comp\_creator.h

### composite\_iterators.cc

### composite\_iterators.h

定义了基本的一些迭代器，比如Filter、Limit、Offset等。

#### class FilterIterator

#### class LimitOffsetIterator

#### class AggregateIterator

#### class NestedLoopIterator

#### class CacheInvalidatorIterator

#### class MaterializeIterator

#### class StreamingIterator

#### class TemptableAggregateIterator

#### class MaterializedTableFunctionIterator

#### class WeedoutIterator

#### class RemoveDuplicatesIterator

#### class NestedLoopSemiJoinWithDuplicateRemovalIterator

#### class WindowingIterator

#### class BufferingWindowingIterator

#### class MaterializeInformationSchemaTableIterator

#### class AppendIterator

### create\_def.cc

### create\_field.cc

### create\_field.h

### current\_thd.cc

### current\_thd.h

### dd\_sp.cc

### dd\_sp.h

### dd\_sql\_view.cc

### dd\_sql\_view.h

### dd\_table\_shared.cc

### dd\_table\_shared.h

### debug\_io\_misc.h

### debug\_lock\_order.cc

### debug\_sync.cc

### debug\_sync.h

### debug\_values.cc

### debug\_values.h

### derror.cc

### derror.h

### dynamic\_ids.cc

Server id的接口封装。

#### pack\_dynamic\_ids

#### unpack\_dynamic\_ids

### dynamic\_ids.h

### enum\_query\_type.h

#### enum enum\_query\_type

说明：查询类型，这个在Item的print的时候会用到，比如derived table，有时候需要打印别名不需要打印表达式。

### error\_handler.cc

### error\_handler.h

#### Internal\_error\_handler

##### Dummy\_error\_handler

##### Set\_var\_error\_handler

##### Drop\_table\_error\_handler

##### MDL\_deadlock\_and\_lock\_abort\_error\_handler

##### View\_error\_handler

##### No\_such\_table\_error\_handler

##### Ignore\_error\_handler

##### Strict\_error\_handler

##### Functional\_index\_error\_handler

##### Tablespace\_name\_error\_handler

##### Key\_length\_error\_handler

##### Info\_schema\_error\_handler

##### Foreign\_key\_error\_handler

##### Ignore\_warnings\_error\_handler

##### Ignore\_json\_syntax\_handler

### event\_data\_object.cc

### event\_data\_object.h

### event\_db\_repository.cc

### event\_db\_repository.h

### event\_parse\_data.cc

### event\_parse\_data.h

### event\_queue.cc

### event\_queue.h

### event\_scheduler.cc

### event\_scheduler.h

### events.cc

### events.h

### field.cc

### field.h

### field\_utils.cc

### field\_utils.h

### filesort.cc

file-sort算子的实现。

### filesort.h

### float\_sompare.h

### gen\_keyword\_list.cc

### gen\_lex\_hash.cc

### gen\_lex\_token.cc

### geometry\_rtree.cc

### gis\_bg\_traits.h

### gstram.cc

### gstram.h

### handler.cc

### handler.h

### hash\_join\_buffer.cc

### hash\_join\_buffer.h

### hash\_join\_chunk.cc

### hash\_join\_chunk.h

### hash\_join\_iterator.cc

### hash\_join\_iterator.h

### init.cc

### init.h

### inplace\_vector.h

### instrusive\_list\_iterator.h

### item\_buff.cc

### item\_cmpfunc.cc

### item\_cmpfunc.h

### item\_create.cc

### item\_create.h

### item\_func.cc

### item\_func.h

### item\_geofunc\_buffer.cc

### item\_geofunc\_internal.cc

### item\_geofunc\_relchecks\_bgwrap.cc

### item\_geofunc\_relchecks.cc

### item\_geofunc\_setops.cc

### item\_geofunc.cc

### item\_inetfunc.cc

### item\_json\_func.cc

### item\_json\_func.h

### item\_pfs\_func.h

### item\_regexp\_func.h

### item\_row.cc

### item\_row.h

### item\_strfunc.cc

### item\_strfunc.h

### item\_subselect.cc

### item\_subselect.h

### item\_timefunc.cc

### item\_xmlfunc.cc

### item\_xmlfunc.h

### item.cc

### item.h

### json\_type.h

### json\_binary.cc

### json\_binary.h

### json\_diff.cc

### json\_diff.h

### json\_dom.cc

### json\_dom.h

### json\_path.cc

### json\_path.h

### json\_schema.cc

### json\_schema.h

### json\_syntax\_check.cc

### json\_syntax\_check.h

### key\_spec.cc

### key\_spec.h

### key.cc

### key.h

### keycaches.cc

### keycaches.h

### keyring\_service.cc

### lex\_symbol.h

### lex.h

### lex\_yystype.h

### lock.cc

### lock.h

### lock\_tables\_list.cc

### lock\_tables\_list.h

### locking\_service\_udf.cc

### locking\_service.cc

### locking\_service.h

### log\_event.cc

### log\_resource.cc

### log\_resource.h

### log.cc

### log.h

### main.cc

main函数的封装，调用mysqld\_main，启动mysqld（mysqld.cc）进程。

### malloc\_allocator.h

#### class Malloc\_allocator

### mdl\_context\_backup.cc

### mdl\_context\_backup.h

### mdl.cc

### mdl.h

### mem\_root\_allocator.h

#### class Mem\_root\_allocator

### mem\_root\_array.h

### merge\_many\_buff.h

### merge\_sort.h

### message.h

### message.mc

### message.rc

### mf\_iocache.cc

### migrate\_keyring.cc

### my\_decimal.cc

### my\_decimal.h

### mydsqld\_daemon.cc

#### mysqld\_daemonize

说明：mysqld.cc调用

### mydsqld\_daemon.h

### mysqld\_thd\_manager.cc

### mysqld\_thd\_manager.h

### mysqld.cc

### mysqld.h

### named\_pipe.cc

### named\_pipe.h

### nested\_join.h

### nt\_servc.cc

### nt\_servc.h

### opt\_constconstantcache.cc

### opt\_constconstantcache.h

### opt\_constconstants.cc

### opt\_constconstants.h

### opt\_costmodel.cc

### opt\_costmodel.h

### opt\_explain\_format.cc

### opt\_explain\_format.h

### opt\_explain\_json.cc

### opt\_explain\_json.h

### opt\_explain\_traditional.cc

### opt\_explain\_traditional.h

### opt\_explain.cc

### opt\_explain.h

### opt\_hints.cc

### opt\_hints.h

### opt\_range.cc

### opt\_range.h

### opt\_statistics.cc

### opt\_statistics.h

### opt\_sum.cc

### opt\_trace\_context.h

### opt\_trace.cc

### opt\_trace.h

### options\_mysqld.h

### options\_parser.cc

### options\_parser.h

### pack\_rows.cc

### pack\_rows.h

### parse\_file.cc

### parse\_file.h

### parse\_location.h

### parse\_tree\_column\_attrs.h

### parse\_tree\_handler.cc

### parse\_tree\_handler.h

### parse\_tree\_helpers.cc

### parse\_tree\_helpers.h

### parse\_tree\_hints.cc

### parse\_tree\_hints.h

### parse\_tree\_items.cc

### parse\_tree\_items.h

### parse\_tree\_node\_base.cc

### parse\_tree\_node\_base.h

### parse\_tree\_nodes.cc

### parse\_tree\_nodes.h

### parse\_tree\_partitions.cc

### parse\_tree\_partitions.h

### parse\_tree\_windows.cc

### parse\_tree\_windows.h

### parse\_service.cc

### parse\_yystype.h

### partition\_element.h

### partition\_info.cc

### partition\_info.h

#### struct PARTITION\_ITERATOR

#### class Parser\_partition\_info

#### class partition\_info

### persisted\_variable.cc

### persisted\_variable.h

### pfs\_batch\_mode.h

#### class PFSBatchMode

### plugin\_table.h

#### class Plugin\_table

##### class Plugin\_view

#### class Plugin\_tablespace

### protocol\_callback.cc

### protocol\_callback.h

#### class Protocol\_callback

### protocol\_classic.cc

### protocol\_classic.h

#### class Protocol\_classic

##### class Protocol\_text

###### class Protocol\_binary

### protocol.h

#### class Protocol

### psi\_memory\_key.cc

#### register\_server\_memory\_keys

### psi\_memory\_key.h

### query\_options.h

### query\_result.cc

### query\_result.h

#### class Query\_result

##### class Query\_result\_interceptor

###### class Query\_result\_to\_file

class Query\_result\_export

class Query\_result\_dump

##### class Query\_result\_send

##### class Query\_dumpvar

##### class Query\_result\_subquery

### record\_buff.h

#### class Record\_buffer

### record.cc

#### IndexScanIterator

IndexScan迭代器的实现（在basic\_row\_iterator.h中定义）。

#### create\_table\_access\_path

构造不同的AccessPath（调用sql/join\_optimizer/access\_path.h）：

##### NewIndexRangeScanAccessPath

##### NewFollowTailAccessPath

##### NewTableScanAccessPath

##### SetCostOnTableAccessPath

说明：sql\_executor.cc

#### init\_table\_iterator

主要是构造AccessPath，然后据此构造Iterator，最后调用迭代器的Init函数完成初始化。

### record.h

### ref\_row\_iterator.h

索引使用类型（访问方式）为ref的时候，需要使用的迭代器。

#### class RefIterator

#### class RefOrNullIterator

#### class EQRefIterator

#### class ConstIterator

#### class FullTextSearchIterator

#### class DynamicRangeIterator

#### class PushedJoinRefIterator

#### class AlternativeIterator

### replication.h

### restart\_monitor\_win.cc

### restart\_monitor\_win.h

### row\_iterator.h

### row\_iterator.h

说明：这里主要是封装了Row相关的迭代器，父类是RowIterator和TableRowIterator，基本的迭代器都是从这里继承（在basic\_row\_iterator.h中定义）。

#### class RowIterator

##### class TableRowIterator

### rpl\_applier\_reader.cc

### rpl\_applier\_reader.h

### rpl\_async\_conn\_failover.cc

### rpl\_async\_conn\_failover.h

### rpl\_async\_conn\_failover\_add\_managed\_udf.cc

### rpl\_async\_conn\_failover\_add\_managed\_udf.h

### rpl\_async\_conn\_failover\_add\_source\_udf.cc

### rpl\_async\_conn\_failover\_add\_source\_udf.h

### rpl\_async\_conn\_failover\_delete\_managed\_udf.cc

### rpl\_async\_conn\_failover\_delete\_managed\_udf.h

### rpl\_async\_conn\_failover\_delete\_source\_udf.cc

### rpl\_async\_conn\_failover\_delete\_source\_udf.h

### rpl\_async\_conn\_failover\_table\_operations.cc

### rpl\_async\_conn\_failover\_table\_operations.h

### rpl\_binlog\_sender.cc

### rpl\_binlog\_sender.h

### rpl\_channel\_credentials.cc

### rpl\_channel\_credentials.h

### rpl\_channel\_service\_interface.cc

### rpl\_channel\_service\_interface.h

### rpl\_commit\_stage\_manager.cc

### rpl\_commit\_stage\_manager.h

### rpl\_constants.h

### rpl\_context.cc

### rpl\_context.h

### rpl\_filter.cc

### rpl\_filter.h

### rpl\_group\_replication.cc

### rpl\_group\_replication.h

### rpl\_gtid\_execution.cc

### rpl\_gtid\_misc.cc

### rpl\_gtid\_mode.cc

### rpl\_gtid\_mutex\_cond\_array.cc

### rpl\_gtid\_owned.cc

### rpl\_gtid\_persist.cc

### rpl\_gtid\_persist.h

### rpl\_gtid\_set.cc

### rpl\_gtid\_sid\_map.cc

### rpl\_gtid\_specification.cc

### rpl\_gtid\_state.cc

### rpl\_gtid.h

### rpl\_handler.cc

### rpl\_handler.h

### rpl\_info\_dummy.cc

### rpl\_info\_factory.cc

### rpl\_info\_file.cc

### rpl\_info\_file.h

### rpl\_info\_handler.cc

### rpl\_info\_handler.h

### rpl\_info\_table\_access.cc

### rpl\_info\_table\_access.h

### rpl\_info\_table.cc

### rpl\_info\_table.h

### rpl\_info\_values.cc

### rpl\_info\_values.h

### rpl\_info.cc

### rpl\_info.h

### rpl\_injector.cc

### rpl\_injector.h

### rpl\_io\_monitor.cc

### rpl\_io\_monitor.h

### rpl\_log\_encryption.cc

### rpl\_log\_encryption.h

### rpl\_master.cc

### rpl\_master.h

### rpl\_mi.cc

### rpl\_mi.h

### rpl\_msr.cc

### rpl\_msr.h

### rpl\_mts\_submode.cc

### rpl\_mts\_submode.h

### rpl\_mysql\_connect.cc

### rpl\_mysql\_connect.h

### rpl\_record.cc

### rpl\_record.h

### rpl\_reporting.cc

### rpl\_reporting.ch

### rpl\_rli\_pdb.cc

### rpl\_rli\_pdb.h

### rpl\_rli.cc

### rpl\_rli.h

### rpl\_slave\_commit\_order\_manager.cc

### rpl\_slave\_commit\_order\_manager.h

### rpl\_slave\_until\_options.cc

### rpl\_slave\_until\_options.h

### rpl\_slave.cc

### rpl\_slave.ch

### rpl\_sys\_key\_access.cc

### rpl\_sys\_key\_access.h

### rpl\_sys\_table\_access.cc

### rpl\_sys\_table\_access.h

### rpl\_table\_access.cc

### rpl\_table\_access.h

### rpl\_tblmap.cc

### rpl\_tblmap.h

### rpl\_transaction\_ctx.cc

### rpl\_transaction\_ctx.h

### rpl\_transaction\_write\_set\_ctx.cc

### rpl\_transaction\_write\_set\_ctx.h

### rpl\_trx\_boundary\_parser.cc

### rpl\_trx\_boundary\_parser.h

### rpl\_trx\_tracking.cc

### rpl\_trx\_tracking.h

### rpl\_utility.cc

### rpl\_utility.h

### rpl\_write\_set\_handler.cc

### rpl\_write\_set\_handler.h

### rules\_table\_service.cc

### rules\_table\_service.h

### rwlock\_scoped\_lock.cc

### rwlock\_scoped\_lock.h

### sd\_notify.cc

### sd\_notify.h

### sdi\_utils.cc

### sdi\_utils.h

### select\_lex\_visitor.cc

### select\_lex\_visitor.h

### session\_tracker.cc

### session\_tracker.h

### set\_var.cc

### set\_var.h

### signal\_handler.cc

### sort\_param.h

### sorting\_iterator.cc

### sorting\_iterator.h

### sp\_cache.cc

### sp\_cache.h

### sp\_head.cc

### sp\_head.h

### sp\_instr.cc

### sp\_instr.h

### sp\_pcontext.cc

### sp\_pcontext.h

### sp\_rcontext.cc

### sp\_rcontext.h

### sp.cc

### spatial.cc

### spatial.h

### sql\_admin.cc

### sql\_admin.h

### sql\_alter\_instance.cc

### sql\_alter\_instance.h

### sql\_alter.cc

### sql\_alter.h

### sql\_array.h

### sql\_audit.cc

### sql\_audit.h

### sql\_backup\_lock.cc

### sql\_backup\_lock.h

### sql\_base.cc

### sql\_base.h

### sql\_binlog.cc

### sql\_binlog.h

### sql\_bitmap.h

### sql\_bootstrap.cc

### sql\_bootstrap.h

### sql\_call.cc

### sql\_call.h

### sql\_check\_constraint.cc

### sql\_check\_constraint.h

### sql\_class.cc

### sql\_class.h

### sql\_client.cc

### sql\_client.h

### sql\_cmd\_ddl\_table.cc

### sql\_cmd\_ddl\_table.h

### sql\_cmd\_ddl.h

### sql\_cmd\_dml.h

### sql\_cmd\_srs.cc

### sql\_cmd\_srs.h

### sql\_cmd.h

### sql\_component.cc

### sql\_component.h

### sql\_connector.cc

### sql\_connector.h

### sql\_const\_folding.cc

### sql\_const\_folding.h

### sql\_const.h

### sql\_constraint.cc

### sql\_constraint.h

### sql\_cursor.cc

### sql\_cursor.h

### sql\_data\_change.cc

### sql\_data\_change.h

### sql\_db.cc

### sql\_db.h

### sql\_delete.cc

### sql\_delete.h

### sql\_derived.cc

### sql\_derived.h

### sql\_digest\_stream.h

### sql\_digest.cc

### sql\_diagest.h

### sql\_do.cc

### sql\_do.h

### sql\_exception\_handler.cc

### sql\_exception\_handler.h

### sql\_exchange.h

### sql\_executor.cc

### sql\_executor.h

### sql\_get\_diagnostics.cc

### sql\_get\_diagnostics.h

### sql\_handler.cc

### sql\_handler.h

### sql\_help.cc

### sql\_help.h

### sql\_hints.yy

### sql\_import.cc

### sql\_import.h

### sql\_initialize.cc

### sql\_initialize.h

### sql\_insert.cc

### sql\_insert.h

### sql\_join\_buffer.cc

### sql\_join\_buffer.h

### sql\_lex\_hash.cc

### sql\_lex\_hash.h

### sql\_lex\_hints.cc

### sql\_lex\_hints.h

### sql\_lex.cc

### sql\_lex.h

### sql\_list.cc

### sql\_list.h

### sql\_load.cc

### sql\_load.h

### sql\_locale.cc

### sql\_locale.h

### sql\_manager.cc

### sql\_manager.h

### sql\_opt\_exec\_shared.h

### sql\_optimizer.cc

### sql\_optimizer.h

### sql\_parse.cc

### sql\_parse.h

### sql\_partition\_admin.cc

### sql\_partition\_admin.h

### sql\_partition.cc

### sql\_partition.h

### sql\_planner.cc

### sql\_planner.h

### sql\_plist.h

### sql\_plugin\_ref.h

### sql\_plugin\_services.h

### sql\_plugin\_var.cc

### sql\_plugin\_var.h

### sql\_plugin.cc

### sql\_plugin.h

### sql\_prepare.cc

### sql\_prepare.h

### sql\_profile.cc

### sql\_profile.h

### sql\_query\_rewrite.cc

### sql\_query\_rewrite.h

### sql\_reload.cc

### sql\_reload.h

### sql\_rename.cc

### sql\_rename.h

### sql\_resolver.cc

### sql\_resolver.h

### sql\_resatrt\_server.cc

### sql\_resatrt\_server.h

### sql\_rewrite.cc

### sql\_rewrite.h

### sql\_select.cc

### sql\_select.h

### sql\_show\_processlist.cc

### sql\_show\_processlist.h

### sql\_show\_status.cc

### sql\_show\_status.h

### sql\_show.cc

### sql\_show.h

### sql\_signal.cc

### sql\_signal.h

### sql\_sort.h

### sql\_state.cc

### sql\_system\_table\_check.h

### sql\_table.cc

### sql\_table.h

### sql\_tablespace.cc

### sql\_tablespace.h

### sql\_thd\_api.cc

### sql\_thd\_internal\_api.cc

### sql\_thd\_internal\_api.h

### sql\_time.cc

### sql\_time.h

### sql\_tmp\_table.cc

说明：主要是临时表的一些操作。

### sql\_tmp\_table.h

### sql\_trigger.cc

### sql\_trigger.h

### sql\_truncate.cc

### sql\_truncate.h

### sql\_udf.cc

### sql\_udf.h

### sql\_union.cc

### sql\_union.h

### sql\_update.cc

### sql\_update.h

### sql\_view.cc

### sql\_view.h

### sql\_yacc.yy

### srs\_fetcher.h

### srv\_session\_info\_service.cc

### srv\_session\_service.cc

### srv\_session.cc

### srv\_session.h

### ssl\_acceptor\_cntext\_data.cc

### ssl\_acceptor\_cntext\_data.h

### ssl\_acceptor\_cntext\_iterator.cc

### ssl\_acceptor\_cntext\_iterator.h

### ssl\_acceptor\_cntext\_operator.cc

### ssl\_acceptor\_cntext\_operator.h

### ssl\_acceptor\_context\_status.h

### ssl\_init\_callbacl.cc

### ssl\_init\_callbacl.h

### ssl\_wrapper\_service.cc

### stateless\_alllocator.cc

### stateless\_alllocator.h

### stream\_ciper.cc

### stream\_ciper.h

### strfunc.cc

### strfunc.h

### string\_serviece.cc

### string\_serviece.h

### sys\_vars\_resource\_mgr.cc

### sys\_vars\_resource\_mgr.h

### sys\_vars\_shared.h

### sys\_var.cc

### sys\_var.h

### system\_variables.cc

### system\_variables.h

### table\_cache.cc

### table\_cache.h

### table\_column\_iterator.h

### table\_function.cc

### table\_function.h

### table\_trigger\_dispatcher.cc

### table\_trigger\_dispatcher.h

### table\_trigger\_field\_support.h

### table.cc

### table.h

### tc\_log.cc

### tc\_log.h

### temp\_table\_param.h

### thd\_raii.h

### thr\_malloc.cc

### thr\_malloc.h

### time\_zone\_common.cc

### time\_zone\_common.h

### timing\_iterator.h

#### TimingIterator

在MySQL 8.0之前，在查询执行计划中检测步骤是不可行的，因为代码是分散的和异构的。从MySQL 8.0开始，在查询执行计划中检测步骤是微不足道的，因为每个步骤本质上都是一样的：一个迭代器。由于迭代器是一个接口，因此可以透明地将每个真正的迭代器包装在一个定时迭代器（sql/timing\_iterator.h）中：

template <class RealIterator>

bool TimingIterator<RealIterator>::Init() {

++m\_num\_init\_calls;

steady\_clock::time\_point start = now(); /\* start time \*/

bool err = m\_iterator.Init(); /\* real iterator \*/

steady\_clock::time\_point end = now(); /\* end time \*/

m\_time\_spent\_in\_first\_row += end - start;

m\_first\_row = true;

return err;

}

TimingIterator这是包装和测量真实迭代器的挂钟时间的类的一部分。在执行查询时，MySQL直接执行真正的迭代器。对于EXPLAIN ANALYZE，MySQL将每个真正的迭代器包装在a中TimingIterator（并丢弃结果集）。

### transaction\_info.cc

### transaction\_info.h

### transaction.cc

### transaction.h

### trigger\_chain.cc

### trigger\_chain.h

### trigger\_creation\_ctx.cc

### trigger\_creation\_ctx.h

### trigger\_def.h

### trigger.cc

### trigger.h

### tzfile.h

### tztime.cc

### tztime.h

### udf\_example.cc

### udf\_example.def

### udf\_registration\_impl.h

### udf\_service\_impl.cc

### udf\_service\_impl.h

### udf\_service\_utils.cc

### udf\_service\_utils.h

### uniques.cc

### uniques.h

### val\_int\_compare.h

### visible\_fields.h

### window\_lex.h

### window.cc

### window.h

### xa\_aux.h

### xa.cc

### xa.h

## sql-common

## storage

## strings

## vio

# 程序

## 概述

## 使用MySQL程序

## 服务器和服务器启动程序

### mysqld

### mysqld\_safe

### mysql.server

### mysqld\_multi

## 安装相关程序

### comp\_err

### mysql\_secure\_installation

### mysql\_ssl\_rsa\_setup

### mysql\_tzinfo\_to\_sql

### mysql\_upgrade

## 客户端程序

### mysql

### mysqladmin

### mysqlcheck

### mysqldump

### mysqlimport

### mysqlpump

### mysqlshow

### mysqlslap

## 管理和实用程序

### ibd2sdi

### innochecksum

### myisam\_ftdump

### myisamchk

### myisamlog

### myisampack

### mysql\_config\_editor

### mysql\_migrate\_keyring

### mysqlbinlog

### mysqldumpslow

## 程序开发实用程序

### mysql\_config

### my\_print\_defaults

## 杂项程序

### lz4\_decompress

### perror

### zlib\_decompress

## 环境变量

## MySQL中的Unix信号处理

# 调试

## 调试Server端

如果你正在使用MySQL中的一些非常新的功能，可以尝试使用该选项运行mysqld--skip-new（这会禁用所有新的、可能不安全的功能）。参考：[如果MySQL不断崩溃该怎么办？](https://dev.mysql.com/doc/refman/8.0/en/crashing.html)

如果mysqld无法启动，请验证是否存在my.cnf影响你设置的文件！可以使用mysqld --print-defaultsmy.cnf检查my.cnf的参数，通过在启动的时候设置mysqld --no-defaults使用它们。

如果mysqld开始耗尽CPU、内存或者进程“挂起”，可以使用mysqladmin processlist status来查明是否有人正在执行一个需要很长时间的查询。如果遇到性能问题或新客户端无法连接时出现问题，在某些窗口中运行mysqladmin -i10 processlist status是个好办法。

命令mysqladmin debug将有关正在使用的锁、已用内存和查询使用情况的一些信息转储到MySQL日志文件。这可能有助于解决一些问题。即使你没有编译MySQL进行调试，这个命令也提供了一些有用的信息！

OPTIMIZE TABLE如果问题是某些表变得越来越慢，应该尝试使用或 myisamchk优化表。请参阅[MySQL服务器管理](https://dev.mysql.com/doc/refman/8.0/en/server-administration.html)。还应该使用来检查慢速查询EXPLAIN。

### 编译MySQL进行调试

如果你遇到一些非常具体的问题，你可以随时尝试调试MySQL。为此，你必须使用该-DWITH\_DEBUG=1选项配置MySQL。你可以通过执行以下命令来检查MySQL是否已通过调试进行编译：mysqld --help。如果该--debug标志与选项一起列出，则说明你已启用调试。mysqladmin ver还会以mysql ... –debug的格式将mysqld版本列出来。

如果当你使用CMake选项配置mysqld-DWITH\_DEBUG=1时停止崩溃，你可能发现了MySQL中的编译器错误或计时错误。-g在这种情况下，你可以尝试使用CMAKE\_C\_FLAGS和CMake选项进行添加CMAKE\_CXX\_FLAGS，而不是使用-DWITH\_DEBUG=1。如果mysqld死掉了，至少可以使用gdb附加到它，或者在core文件上使用gdb来找出发生了什么。

当你配置MySQL进行调试时，你会自动启用许多额外的安全检查功能来监视mysqld的运行状况。如果他们发现“意外”的情况，则会将一个条目写入 mysqld\_safestderr并将其定向到错误日志！这也意味着，如果你在使用 MySQL 时遇到一些意外问题并且正在使用源代码发行版，那么你应该做的第一件事就是配置MySQL进行调试。

在Windows MySQL发行版中，mysqld.exe默认情况下编译时支持跟踪文件。

### 创建跟踪文件

如果mysqld服务器没有启动或者很容易崩溃，可以尝试创建跟踪文件来查找问题。

为此，你必须有一个已编译并支持调试的mysqld。你可以通过执行mysqld -V来检查。如果版本号以结尾-debug，则编译时支持跟踪文件。（在Windows上，调试服务器名为mysqld-debug而不是mysqld。）

mysql-server-8.0.24/bin/mysqld Ver 8.0.26-17-debug for Linux on x86\_64 (Source distribution)

通过跟踪日志启动mysqld服务器（Unix：/tmp/mysqld.trace，Windows：\mysqld.trace）：

$> mysqld --debug

在Windows上，你还应该使用该--standalone标志来不将mysqld作为服务启动。在控制台窗口中，使用以下命令：

C:\> mysqld-debug --debug --standalone

之后，你可以在第二个控制台窗口中使用mysql.exe命令行工具来重现问题。你可以使用 mysqladmin shutdown停止mysqld服务器。

跟踪文件可能会变得非常大！要生成较小的跟踪文件，你可以使用如下调试选项：

mysqld --debug=d,info,error,query,general,where:O,/tmp/mysqld.trace

这只会将带有最有趣标签的信息打印到跟踪文件中。

如果你提交错误，请仅将跟踪文件中的那些行添加到错误报告中，以指示似乎出现问题的位置。如果找不到错误的地方，请打开错误报告并将整个跟踪文件上传到报告中，以便MySQL开发人员查看。

跟踪文件是DBUG由Fred Fish的软件包制作的。请参见“[DBUG包”](https://dev.mysql.com/doc/refman/8.0/en/dbug-package.html)。

### 使用WER和PDB创建Windows故障转储

程序数据库文件（带有后缀pdb）包含在MySQL的ZIP Archive调试二进制文件和测试套件发行版中。这些文件提供了在出现问题时调试MySQL安装的信息。这是从标准MSI或Zip文件中单独下载的。

说明：

PDB文件位于标有“ZIP Archive Debug Binaries & Test Suite”的单独文件中。

PDB文件包含有关 mysqld可创建更详细的跟踪和转储文件的其他工具的更详细信息。你可以将它们与WinDbg或Visual Studio一起使用来调试mysqld。

要使用WinDbg，请安装完整的Windows驱动程序工具包 (WDK) 或安装独立版本。

说明：

.exe和.pdb文件必须完全匹配（版本号和MySQL服务器版本）；否则，或者WinDBG在尝试加载符号时会出错。

1、要生成小型转储mysqld.dmp，设置my.ini文件[mysqld]配置项下面的core-file，在设置这些变更后重新启动MySQL服务器。

2、创建一个目录来存放生成的文件，例如 c:\symbols

3、使用查找GUI或从命令行确定Windbg.exe可执行文件的路径，例如： dir /s /b windbg.exe –a。常见的默认值是C:\Program Files\Debugging Tools for Windows (x64)\windbg.exe

4、启动windbg.exe，为mysqld.exe、mysqld.pdb、mysqld.dmp提供路径和源码信息。或者，从WinDbg GUI传入每个路径。例如：

windbg.exe -i "C:\mysql-8.0.34-winx64\bin\"^

-z "C:\mysql-8.0.34-winx64\data\mysqld.dmp"^

-srcpath "E:\ade\mysql\_archives\8.0\8.0.34\mysql-8.0.34"^

-y "C:\mysql-8.0.34-winx64\bin;SRV\*c:\symbols\*http://msdl.microsoft.com/download/symbols"^

-v -n -c "!analyze -vvvvv"

说明：

Windows 命令行处理器会删除字符^和换行符，因此请确保空格保持完整。

### gdb下调试mysqld

在大多数系统上，如果mysqld崩溃，你还可以从gdb启动mysqld以获取更多信息。

如果希望使用Linux上的一些较旧的gdb版本调试mysqld线程，则必须run --one-thread。在这种情况下，一次只能有一个线程处于活动状态。

在gdb下运行mysqld时，NPTL线程（Linux上的新线程库）可能会导致问题。一些现象是：

* mysqld在启动期间挂起（在写入之前ready for connections）。
* mysqld在调用pthread\_mutex\_lock()或pthread\_mutex\_unlock()期间崩溃。

在这种情况下，你应该在启动gdb之前在shell中设置以下环境变量：

LD\_ASSUME\_KERNEL=2.4.1

export LD\_ASSUME\_KERNEL

当在gdb下运行mysqld时，你应该通过--skip-stack-trace禁用堆栈跟踪， 以便能够捕获gdb中的段错误。

使用mysqld--gdb的选项安装中断处理程序处理SIGINT（需要通过^C停止mysqld以设置断点）并禁用堆栈跟踪和core文件处理。

如果你一直进行大量新连接，那么在gdb下调试MySQL会非常困难，因为gdb不会为旧线程释放内存。你可以通过启动mysqld时，将thread\_cache\_size设置为等于max\_connections + 1 的值来来避免此问题。在大多数情况下，只需使用--thread\_cache\_size=5'就会有很大帮助！

如果你想在mysqld因SIGSEGV信号而终止时在Linux上获取核心转储，你可以使用该选项启动mysqld--core-file。这个核心文件可用于进行回溯，可以帮助你找出mysqld死亡的原因：

$> gdb mysqld core

gdb> backtrace full

gdb> quit

如果你在Linux上使用gdb，则应.gdb在当前目录中安装一个包含以下信息的文件：

set print sevenbit off

handle SIGUSR1 nostop noprint

handle SIGUSR2 nostop noprint

handle SIGWAITING nostop noprint

handle SIGLWP nostop noprint

handle SIGPIPE nostop

handle SIGALRM nostop

handle SIGHUP nostop

handle SIGTERM nostop noprint

以下是如何调试mysqld 的 示例：

$> gdb /usr/local/libexec/mysqld

gdb> run

...

backtrace full # Do this when mysqld crashes

如果mysqld挂起，你可以尝试使用一些系统工具，例如strace或 /usr/proc/bin/pstack来检查mysqld挂起的位置。

strace /tmp/log libexec/mysqld

如果你使用的是PerlDBI接口，可以通过使用trace方法或设置 DBI\_TRACE环境变量来打开调试信息。

### 使用堆栈跟踪

在某些操作系统上，如果mysqld意外终止，错误日志会包含堆栈跟踪。你可以使用它来找出mysqld 死亡的位置（也许还有原因）。要获得堆栈跟踪，你不能使用gcc选项编译mysqld-fomit-frame-pointer。请参见“[编译MySQL以进行调试](https://dev.mysql.com/doc/refman/8.0/en/compiling-for-debugging.html)”。

错误日志中的堆栈跟踪如下所示：

mysqld got signal 11;

Attempting backtrace. You can use the following information

to find out where mysqld died. If you see no messages after

this, something went terribly wrong...

stack\_bottom = 0x41fd0110 thread\_stack 0x40000

mysqld(my\_print\_stacktrace+0x32)[0x9da402]

mysqld(handle\_segfault+0x28a)[0x6648e9]

/lib/libpthread.so.0[0x7f1a5af000f0]

/lib/libc.so.6(strcmp+0x2)[0x7f1a5a10f0f2]

mysqld(\_Z21check\_change\_passwordP3THDPKcS2\_Pcj+0x7c)[0x7412cb]

mysqld(\_ZN16set\_var\_password5checkEP3THD+0xd0)[0x688354]

mysqld(\_Z17sql\_set\_variablesP3THDP4ListI12set\_var\_baseE+0x68)[0x688494]

mysqld(\_Z21mysql\_execute\_commandP3THD+0x41a0)[0x67a170]

mysqld(\_Z11mysql\_parseP3THDPKcjPS2\_+0x282)[0x67f0ad]

mysqld(\_Z16dispatch\_command19enum\_server\_commandP3THDPcj+0xbb7[0x67fdf8]

mysqld(\_Z10do\_commandP3THD+0x24d)[0x6811b6]

mysqld(handle\_one\_connection+0x11c)[0x66e05e]

如果跟踪的函数名称解析失败，则跟踪包含的信息较少：

mysqld got signal 11;

Attempting backtrace. You can use the following information

to find out where mysqld died. If you see no messages after

this, something went terribly wrong...

stack\_bottom = 0x41fd0110 thread\_stack 0x40000

[0x9da402]

[0x6648e9]

[0x7f1a5af000f0]

[0x7f1a5a10f0f2]

[0x7412cb]

[0x688354]

[0x688494]

[0x67a170]

[0x67f0ad]

[0x67fdf8]

[0x6811b6]

[0x66e05e]

较新版本的glibc堆栈跟踪函数还会打印相对于对象的地址。在glibc基于 的系统（Linux）上，插件内意外退出的跟踪类似于：

plugin/auth/auth\_test\_plugin.so(+0x9a6)[0x7ff4d11c29a6]

要将相对地址 ( +0x9a6) 转换为文件名和行号，请​​使用以下命令：

$> addr2line -fie auth\_test\_plugin.so 0x9a6

auth\_test\_plugin

mysql-trunk/plugin/auth/test\_plugin.c:65

addr2line实用程序是Linux上的软件包的一部分binutils。

在Solaris上，过程类似。Solaris printstack()已经打印相对地址：

plugin/auth/auth\_test\_plugin.so:0x1510

要翻译，请使用以下命令：

$> gaddr2line -fie auth\_test\_plugin.so 0x1510

mysql-trunk/plugin/auth/test\_plugin.c:88

Windows 已经打印了地址、函数名称和行：

000007FEF07E10A4 auth\_test\_plugin.dll!auth\_test\_plugin()[test\_plugin.c:72]

### 使用服务器日志查找mysqld错误原因

注意，在启用通用查询日志的情况下启动mysqld之前，你应该使用myisamchk检查所有表。请参阅MySQL服务器管理。

如果mysqld死机或挂起，你应该在启用常规查询日志的情况下启动mysqld。请参见“通用查询日志”。当mysqld 再次终止时，你可以检查日志文件的末尾以查找终止mysqld的查询。

如果你使用默认的一般查询日志文件，该日志将存储在数据库目录中，因为host\_name.log在大多数情况下，它是日志文件中杀死mysqld的最后一个查询，但如果可能，你应该通过重新启动mysqld并执行找到的查询来验证这一点从mysql命令行工具。如果这有效，你还应该测试所有未完成的复杂查询。

EXPLAIN你还可以在所有SELECT需要很长时间的语句上尝试该命令，以确保mysqld正确使用索引。请参见“EXPLAIN 语句”。

你可以通过启动mysqld并启用慢查询日志来 找到执行时间较长的查询。请参见“慢查询日志”。

如果你mysqld restarted在错误日志（通常是名为host\_name.err的文件）中找到该文本，你可能发现了导致mysqld失败的查询 。如果发生这种情况，你应该使用myisamchk检查所有表（请参阅MySQL服务器管理），并测试MySQL日志文件中的查询以查看是否有失败。如果发现这样的查询，请先尝试升级到最新的MySQL版本。

如果你使用系统变量集启动了mysqldmyisam\_recover\_options，MySQL会自动检查并尝试修复MyISAM被标记为“未正确关闭”或“崩溃”的表。hostname.err如果发生这种情况，MySQL会在文件 中写入一个条目'Warning: Checking table ...'，后跟Warning: Repairing table是否需要修复表。如果你收到大量此类错误，而mysqld之前没有意外死亡，则说明出现了问题，需要进一步调查。请参见“服务器命令选项”。

当服务器检测到MyISAM表损坏时，它会将附加信息写入错误日志，例如源文件的名称和行号以及访问表的线程列表。示例：Got an error from thread\_id=1, mi\_dynrec.c:368. 这是包含在错误报告中的有用信息。

如果mysqld确实意外死亡， 这不是一个好兆头，但在这种情况下，你不应该调查这些Checking table...消息，而应该尝试找出mysqld死亡的原因。

#### DBUG\_ENTER

#### DBUG\_RETURN

#### DBUG\_VOID\_RETURN

#### DBUG\_EXECUTE

#### DBUG\_EXECUTE\_IF

参考：<https://developer.aliyun.com/article/40903>

DEBUG\_EXECUTE\_IF主要用于当设置了某个关键字key时，执行后面的代码。可以简单的表示为：

DEBUG\_EXECUTE\_IF(key, code)

例如在open\_and\_lock\_tables函数中：

5527 if (open\_tables(thd, &tables, &counter, flags, prelocking\_strategy))

5528 goto err;

5529

5530 DBUG\_EXECUTE\_IF("sleep\_open\_and\_lock\_after\_open", {

5531 const char \*old\_proc\_info= thd->proc\_info;

5532 thd->proc\_info= "DBUG sleep";

5533 my\_sleep(6000000);

5534 thd->proc\_info= old\_proc\_info;});

5535

5536 if (lock\_tables(thd, tables, counter, flags))

5537 goto err;

执行如下语句让其生效：

set session debug=”+d, sleep\_open\_and\_lock\_after\_open”

这时候当进入这个函数时，在执行完open\_tables语句后，就会设置thd的状态为DBUG sleep，再sleep 6秒钟，然后再调用lock\_tables函数。

另外我们在写test case时，也可能用到用户层的锁，使用SQL FUNCTION的方法实现：

GET\_LOCK(str, timeout)

—尝试获取一个名为str的锁，等待timeout，返回1表示获取锁成功

IS\_FREE\_LOCK(str)

—检查名为str的锁是否已经被释放

IS\_USED\_LOCK(str)

—检查名为str的锁是否已经被占用，如果是，返回占有该锁的线程ID，否则返回NULL

RELEASE\_LOCK(str)

—释放名为str的锁

#### DBUG\_EVALUATE

#### DBUG\_EVALUATE\_IF

DBUG\_EVALUATE\_IF主要用于设置了某个关键字key时，结果设置为a1,否则设置为a2。可以简单的表示为：

DBUG\_EVALUATE\_IF (key, a1, a2)

#### DBUG\_PRINT

#### DBUG\_PUSH

#define DBUG\_PUSH(a1) \_db\_push\_(a1)

#### DBUG\_POP

#define DBUG\_POP() \_db\_pop\_()

#### DBUG\_SET

#define DBUG\_SET(a1) \_db\_set\_(a1)

#### DBUG\_SET\_INITIAL

#define DBUG\_SET\_INITIAL(a1) \_db\_set\_init\_(a1)

#### DBUG\_PROCESS

#define DBUG\_PROCESS(a1) \_db\_process\_(a1)

#### DBUG\_FILE

#define DBUG\_FILE \_db\_fp\_()

#### DBUG\_DUMP

#define DBUG\_DUMP(keyword, a1, a2) \_db\_dump\_(\_\_LINE\_\_, keyword, a1, a2)

#### DBUG\_END

#define DBUG\_END() \_db\_end\_()

#### DBUG\_LOCK\_FILE

#define DBUG\_LOCK\_FILE \_db\_lock\_file\_()

#### DBUG\_UNLOCK\_FILE

#define DBUG\_UNLOCK\_FILE \_db\_unlock\_file\_()

#### DBUG\_ASSERT

#define DBUG\_ASSERT(A) assert(A)

#### DBUG\_EXPLAIN

#define DBUG\_EXPLAIN(buf, len) \_db\_explain\_(0, (buf), (len))

#### DBUG\_EXPLAIN\_INITIAL

#define DBUG\_EXPLAIN\_INITIAL(buf, len) \_db\_explain\_init\_((buf), (len))

### 如果遇到表损坏，则创建测试用例

以下过程适用于MyISAM表。

如果遇到损坏的MyISAM 表或者mysqld在某些更新语句后总是失败，可以通过执行以下操作来测试问题是否可重现：

1、使用mysqladmin shutdown停止MySQL守护进程。

2、对表进行备份，以防止修复造成不良后果的极少数情况。

3、使用myisamchk -s database/\*.MYI 检查所有表。使用myisamchk -r database/ table.MYI修复任何损坏的表 。

4、对表进行第二次备份。

5、如果需要更多空间，从MySQL数据目录中删除（或移走）所有旧日志文件。

6、启动mysqld并启用二进制日志。如果想找到导致mysqld崩溃的语句，应该在启用通用查询日志的情况下启动服务器。

7、当表崩溃时，停止mysqld服务器。

8、恢复备份。

9、在不启用二进制日志的情况下重新启动mysqld服务器。

10、重新执行语句mysqlbinlog binary-log-file | mysql。二进制日志保存在MySQL数据库目录中，名称为. hostname-bin.NNNNNN

11、如果表再次损坏或者可以使用上述命令让mysqld死掉，那么就发现了一个可重现的错误。

## 调试Client端

为了能够使用集成调试包调试 MySQL 客户端，你应该使用-DWITH\_DEBUG=1。请参见“[MySQL 源配置选项](https://dev.mysql.com/doc/refman/8.0/en/source-configuration-options.html)”。

在运行客户端之前，你应该设置 MYSQL\_DEBUG环境变量：

$> MYSQL\_DEBUG=d:t:O,/tmp/client.trace

$> export MYSQL\_DEBUG

这样客户端会在/tmp/client.trace生成对应的trace文件。

如果你自己的客户端代码有问题，你应该尝试连接到服务器并使用可以工作的客户端运行查询。通过在调试模式下运行mysql来完成此操作（假设你已在调试模式下编译了 MySQL）：

$> mysql --debug=d:t:O,/tmp/client.trace

如果你的客户端因某些看起来“合法”的代码而崩溃，你应该检查你的mysql.h包含文件是否与你的MySQL库文件相匹配。mysql.h一个非常常见的错误是将旧 MySQL 安装中的旧文件与新MySQL库一起使用 。

## LOCK ORDER工具

MySQL 服务器是一个多线程应用程序，它使用大量内部锁定和与锁定相关的原语，例如互斥锁、rwlocks（包括 prlocks 和 sxlocks）、条件和文件。在服务器内，与锁相关的对象集随着新功能的实现和代码重构以提高性能而发生变化。与任何使用锁定原语的多线程应用程序一样，当同时持有多个锁时，在执行过程中始终存在遇到死锁的风险。对于 MySQL 来说，死锁的影响是灾难性的，导致服务完全丢失。

从 MySQL 8.0.17 开始，为了能够检测锁获取死锁并强制执行运行时避免死锁，MySQL 支持LOCK\_ORDER工具。这使得锁顺序依赖图能够被定义为服务器设计和服务器运行时检查的一部分，以确保锁获取是非循环的并且执行路径符合该图。

本节提供有关使用该 LOCK\_ORDER工具的信息，但仅限于基础级别。有关完整的详细信息，请参阅https://dev.mysql.com/doc/index-other.html。

该LOCK\_ORDER工具旨在用于调试Server服务器，而不是用于生产用途。

要使用该LOCK\_ORDER工具，请按照以下步骤操作：

1、从源代码构建 MySQL，使用 CMake选项对其进行配置，以便构建包含工具。-DWITH\_LOCK\_ORDER=ON LOCK\_ORDER

笔记

启用该WITH\_LOCK\_ORDER 选项后，MySQL 构建需要 flex程序。

2、要在启用该工具的情况下运行服务器LOCK\_ORDER，请 lock\_order在服务器启动时启用系统变量。LOCK\_ORDER还可以使用 其他几个用于配置的系统变量 。

3、对于 MySQL 测试套件操作，mysql-test-run.pl有一个 --lock-order选项可以控制是否LOCK\_ORDER在测试用例执行期间启用该工具。

LOCK\_ORDER假设 MySQL 已构建为包含工具，则在工具 的配置操作之后描述系统变量LOCK\_ORDER。主要变量是lock\_order，指示是否在运行时启用该 LOCK\_ORDER工具：

* 如果lock\_order禁用（默认），则其他LOCK\_ORDER系统变量不会产生任何影响。
* 如果lock\_order启用，其他系统变量将配置 LOCK\_ORDER要启用的功能。

笔记：

一般来说，该工具的目的是 通过使用选项 LOCK\_ORDER执行 mysql-test-run.pl--lock-order来配置该工具，并将 mysql-test-run.pl设置 LOCK\_ORDER为适当的值。

所有LOCK\_ORDER系统变量都必须在服务器启动时设置。在运行时，它们的值是可见的，但不能更改。

有些系统变量是成对存在的，例如 lock\_order\_debug\_loop和 lock\_order\_trace\_loop。对于此类对，当发生与其关联的条件时，变量按如下方式区分：

* 如果\_debug\_启用该变量，则会引发调试断言。
* 如果\_trace\_启用该变量，则会将错误打印到日志中。

## DEBUG包

参考：<https://www.cnblogs.com/jkin/p/16497783.html>

MySQL的debug版提供了一个专门的DBUG包。通过这个DBUG包，可获取正在执行操作程序的Trace文件。通过控制DBUG开关，可以将MySQL 的任何操作，以及所涉及的调用模块、函数、状态信息记录在Trace文件中。

**方法一：**

DBUG包可以通过使用--debug[=debug\_options]或-#[debug\_options]选项调用程序来使用。如果在没有debug\_options值的情况下指定--debug或-#选项，则大多数MySQL程序使用默认值。服务器默认为d:t:i:o，/tmp/mysqld。大多数客户端程序使用默认的debug选项值d:t:o，/tmp/program\_name.trace。

说明：启动程序的时候添加-debug配置即可。

在shell命令行指定调试控制字符串，示例：

--debug=d:t

--debug=d:f,main,subr1:F:L:t,20

--debug=d,input,output,files:n

--debug=d:t:i:O,\\mysqld.trace

**方法二：**

对于mysqld，也可以通过设置debug系统变量在运行时更改DBUG设置。此变量具有全局值和会话值：

mysql> SET GLOBAL debug = 'debug\_options';

mysql> SET SESSION debug = 'debug\_options';

debug\_options值是一系列冒号分隔的字段，值中的每个字段都由一个强制标志字符组成，前面可选+或-字符，后面可选逗号分隔的修饰符列表

[+|-]flag[,modifier,modifier,...,modifier]

下表介绍标志字符。无法识别的标志字符将被自动忽略。



前导+或-字符和尾随修饰符列表用于标志字符，如d或f，可以对所有适用修饰符或其中一些修饰符启用调试操作：

如果没有前导+或-，则标志值将精确设置为给定的修饰符列表。使用前导+或-，列表中的修饰符将添加到当前修饰符列表或从中减去

以下示例显示了这如何适用于d标志。为所有调试宏启用了空d列表输出。非空列表仅为列表中的宏关键字启用输出。

这些语句将d值设置为给定的修饰符列表：

mysql> SET debug = 'd';

mysql> SELECT @@debug;

+---------+

| @@debug |

+---------+

| d |

+---------+

mysql> SET debug = 'd,error,warning';

mysql> SELECT @@debug;

+-----------------+

| @@debug |

+-----------------+

| d,error,warning |

+-----------------+

前导+或-将当前d值相加或相减：

mysql> SET debug = '+d,loop';

mysql> SELECT @@debug;

+----------------------+

| @@debug |

+----------------------+

| d,error,warning,loop |

+----------------------+

说明：前面一次操作中debug的变量值设置为d,error,warning，在这里通过+d的方式，增加了一个loop信息。

mysql> SET debug = '-d,error,loop';

mysql> SELECT @@debug;

+-----------+

| @@debug |

+-----------+

| d,warning |

+-----------+

说明：前面一次操作中debug的变量值设置为d,error,warning,loop，在这里通过-d的方式，减少了error,loop信息。

添加到“所有启用的宏”不会导致任何更改：

mysql> SET debug = 'd';

mysql> SELECT @@debug;

+---------+

| @@debug |

+---------+

| d |

+---------+

mysql> SET debug = '+d,loop';

mysql> SELECT @@debug;

+---------+

| @@debug |

+---------+

| d |

+---------+

禁用所有启用的宏将完全禁用d标志：

mysql> SET debug = 'd,error,loop';

mysql> SELECT @@debug;

+--------------+

| @@debug |

+--------------+

| d,error,loop |

+--------------+

mysql> SET debug = '-d,error,loop';

mysql> SELECT @@debug;

+---------+

| @@debug |

+---------+

| |

+---------+

建议：

set debug = 'd:t:o,/tmp/mysqld.trace';只对当前会话生效，只对当前执行的语句 trace。

官方手册：<https://dev.mysql.com/doc/refman/8.0/en/debugging-mysql.html>

## DEBUG技巧

参考：<https://blog.csdn.net/weixin_43770745/article/details/126931668>