第十九讲：性能分析及 handlerton 存储引擎接口设计

知春路遇上八里桥

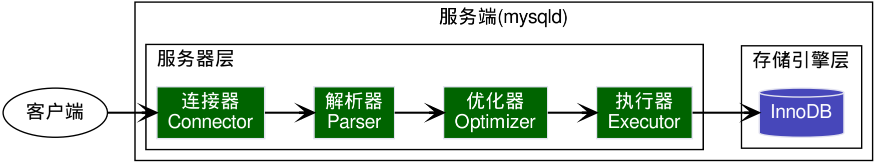
<2024-07-22 Mon>

1. 前情提要
2. 追溯日志
3. 性能分析
4. PFS 实现
5. 存储引擎的接口层

1

前情提要

本节内容

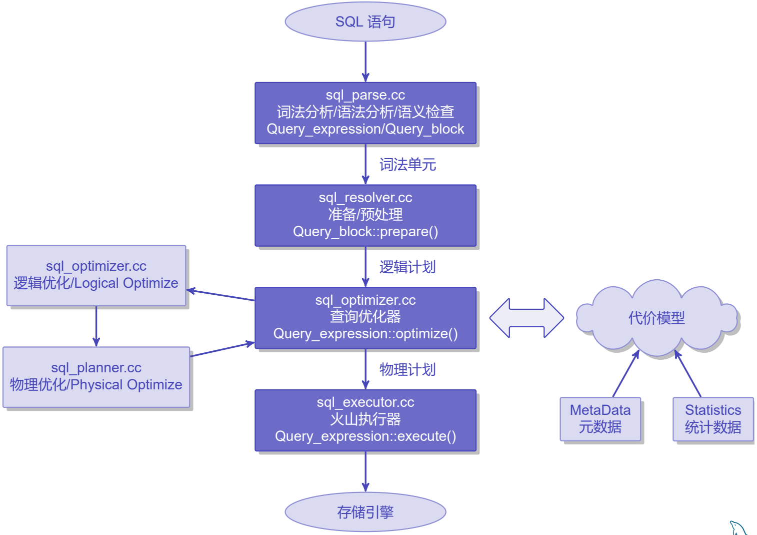


- 连接器
  - ☑ 连接管理器 Connection Manager
  - ☑ 线程管理器 Thread Manager
  - ☑ 用户模块 User Module
- 解析器
  - ☑ 网络模块 Net Module
  - ☑ 派发模块 Commander Dispatcher
  - ☑ 词法分析 Lexical Analysis
  - ☑ 语法分析 Syntax Analysis
- 优化器
  - ☑ 准备模块 Prepare Module
  - ☑ 追踪日志 Optimizer Trace
  - ☑ 优化模块 Optimize Module
- 执行器
  - ☑ 火山模型 Volcano Model
  - ☑ 执行模块 Execution Module

2

追溯日志

服务层架构梳理


SQL 语句
sql_parse.cc
词法分析/语法分析/语义检查
Query_expression/Query_block
词法单元
sql_resolver.cc
准备/预处理
Query_block::prepare()
逻辑计划
sql_optimizer.cc
逻辑优化/Logical Optimize
sql_optimizer.cc
查询优化器
Query_expression::optimize()
代价模型
sql_planner.cc
物理优化/Physical Optimize
物理计划
sql_executor.cc
火山执行器
Query_expression::execute()
MetaData
元数据
Statistics
统计数据
存储引擎

Parser 解析器

- parse_sql() 函数进入 SQL 解析器
  - 词法分析，产生 token
  - 语法分析，产生抽象语法树
- 通过 BISON 生成解析树 PT
  - Parse Tree
- 语法分析还构建词法单元 LU
  - Lexical Unit
  - Query_expression 表示查询表达式
  - Query_block 表示查询块

```
T@10: | | | >parse_sql
T@10: | | | | >THD::convert_string
...
T@10: | | | | >MEM_ROOT::AllocSlow
T@10: | | | | | enter: root: 0x7fff300aed10
T@10: | | | | | >MEM_ROOT::AllocBlock
T@10: | | | | | | >char* PFS_instr_name::str
T@10: | | | | | | <char* PFS_instr_name::str
T@10: | | | | | <MEM_ROOT::AllocBlock
T@10: | | | | <MEM_ROOT::AllocSlow
T@10: | | | | >THD::convert_string
T@10: | | | | >find_udf
...
T@10: | | | | >Query_block::add_table_to_list
T@10: | | | | | >MEM_ROOT::AllocSlow
T@10: | | | | | | enter: root: 0x7fff300aed10
T@10: | | | | | | >MEM_ROOT::AllocBlock
T@10: | | | | | | | >char* PFS_instr_name::str
T@10: | | | | >Query_block::add_joined_table
T@10: | | | <parse_sql
```

Resolver 准备模块

```
T@10: | | | >mysql_execute_command
...
T@10: | | | | >bool Sql_cmd_dml::execute
T@10: | | | | | >bool Sql_cmd_dml::prepare
T@10: | | | | | | >check_table_access
T@10: | | | | | | >open_tables_for_query
...
T@10: | | | | | | >Query_block::prepare
T@10: | | | | | | | >Query_block::setup_tables
T@10: | | | | | | | >setup_fields
T@10: | | | | | | | >Query_block::setup_conds
T@10: | | | | | | | >Query_block::apply_local_transforms
T@10: | | | | | | | | >Query_block::simplify_joins
T@10: | | | | | | | | <Query_block::simplify_joins
T@10: | | | | | | | | >build_bitmap_for_nested_joins
T@10: | | | | | | | | <build_bitmap_for_nested_joins
T@10: | | | | | | | <Query_block::apply_local_transforms
T@10: | | | | | | | opt: steps: ending struct
T@10: | | | | | | | opt: join_preparation: ending struct
T@10: | | | | | | | opt: (null): ending struct
T@10: | | | | | | <Query_block::prepare
T@10: | | | | | <bool Sql_cmd_dml::prepare
```

- 准备阶段是预处理词法单元
  - `mysql_execute_command()` 执行命令
  - `Sql_cmd_dml::execute()` SELECT 语句
- 各种 prepare 函数叠加调用
  - `Sql_cmd_dml::prepare()`
  - `Query_block::prepare()`
- 单个查询块的处理包括
  - `setup_tables()` 设置表
  - `setup_fields()` 设置列
  - `setup_conds()` 设置条件
  - `apply_local_transforms()` 本地变换
- 准备阶段生成结果是逻辑计划
  - Logical Plan
- 准备阶段对应 Opt_trace 的阶段是
  - join_preparation
  - 注意 join_preparation 可能会嵌套

Optimizer 优化器

- Query_expression::optimize() 是优化器的入口，最终调用 JOIN::optimize() 执行优化
- 优化器核心目的是计算不同策略的 cost 值，最终选取 cost 值最小的执行计划
- 在计算 JOIN 次序时如果搜索使用次序排列效率较低，故使用贪心算法 greedy_search()

```
T@10: | | | | | >Query_expression::optimize
T@10: | | | | | | >Query_block::optimize
T@10: | | | | | | | >JOIN::optimize
...
T@10: | | | | | | | | >optimize_cond
T@10: | | | | | | | | >JOIN::make_join_plan
T@10: | | | | | | | | | >int ha_innobase::info_low
T@10: | | | | | | | | | >JOIN::make_sum_func_list
T@10: | | | | | | | | | >get_quick_record_count
T@10: | | | | | | | | | | >test_quick_select
...
T@10: | | | | | | | | | | >Optimize_table_order::greedy_search
T@10: | | | | | | | | | | | >Optimize_table_order::best_extension_by_limited_search
part_plan; idx: 0  best: DBL_MAX  atime: 0  itime: 0  count: 1
     POSITIONS:
      BEST_REF: employees(9,276023,1)
T@10: | | | | | | | | | | | | >Optimize_table_order::best_access_path
...
T@10: | | | | | | | | | | | | | opt: cost: 2.81809
T@10: | | | | | | | | | | | | | opt: chosen: 1
...
```

Executor 执行器

- Query_expression::execute() 是执行器的入口，处理执行计划并从存储引擎中读取数据
- 执行器采用火山模型，执行器对应 Opt_trace 的 join_execution 阶段

```
T@10: | | | | | >Query_expression::execute
T@10: | | | | | | >THD::send_result_metadata
T@10: | | | | | | | >bool Protocol_classic::start_result_metadata
T@10: | | | | | | | >bool Protocol_classic::send_field_metadata
T@10: | | | | | | | >bool Protocol_classic::end_row
...
T@10: | | | | | | >InitIndexRangeScan
T@10: | | | | | | | >handler::ha_index_init
T@10: | | | | | | | | >int ha_innobase::index_init
T@10: | | | | | | | | | >ha_innobase::change_active_index
T@10: | | | | | | | | | | >dict_index_t* ha_innobase::innobase_get_index
... 读取/发送数据若干次
T@10: | | | | | | >int IndexRangeScanIterator::Read
T@10: | | | | | | | >handler::ha_multi_range_read_next
T@10: | | | | | | | | >int handler::multi_range_read_next
T@10: | | | | | | | | | >int handler::read_range_first
T@10: | | | | | | | | | | >handler::ha_index_first
T@10: | | | | | | | | | | | >int ha_innobase::index_first
T@10: | | | | | | | | | | | | >int ha_innobase::index_read
T@10: | | | | | | | | | | | | | >row_search_mvcc
...
T@10: | | | | | | >bool Query_result_send::send_data
T@10: | | | | | | | >THD::send_result_set_row
...
```

3

性能分析

Profile 配置

```
mysql> set profiling=1;
Query OK, 0 rows affected, 1 warning (0.00 sec)

mysql> \. exe02.sql
... 省略执行结果输出

mysql> show profiles\G
*************** 1. row ***************
Query_ID: 1
Duration: 0.00713825
   Query: select
  x.dept_no,
  e.first_name,
  e.last_name
from
  dept_manager x
  left join
    employees e
      using (emp_no)
where
  x.dept_no < 'd004'
1 row in set, 1 warning (0.01 sec)
```

- 开启 profiling 后执行查询语句
- 通过 show profiles 查看 Query_ID

  ```
  show profile cpu, ipc for query 1;
  show profile memory for query 1;
  ```

```
mysql> show profile for query 1;
+----------------------+----------+
| Status               | Duration |
+----------------------+----------+
| starting             | 0.000921 |
| checking permissions | 0.000066 |
| checking permissions | 0.000135 |
| Opening tables       | 0.007945 |
| end                  | 0.000027 |
| query end            | 0.000042 |
| closing tables       | 0.000036 |
| freeing items        | 0.000117 |
| cleaning up          | 0.000057 |
+----------------------+----------+
9 rows in set, 1 warning (0.00 sec)
```

Profile 结果查询和分析

- 通过 Query_ID 查询 information_schema.profiling 表获取执行耗时

```
mysql> select query_id, seq, state, duration, source_file, source_line
    ->   from information_schema.profiling where query_id = 1;
+----------+-----+--------------------------------+----------+----------------------+-------------+
| query_id | seq | state                          | duration | source_file          | source_line |
+----------+-----+--------------------------------+----------+----------------------+-------------+
|        1 |   2 | starting                       | 0.000692 | NULL                 |        NULL |
|        1 |   3 | Executing hook on transaction  | 0.000054 | rpl_handler.cc       |        1477 |
|        1 |   4 | starting                       | 0.000142 | rpl_handler.cc       |        1479 |
|        1 |   5 | checking permissions           | 0.000041 | sql_authorization.cc |        2146 |
|        1 |   6 | checking permissions           | 0.000039 | sql_authorization.cc |        2146 |
|        1 |   7 | Opening tables                 | 0.000697 | sql_base.cc          |        5797 |
|        1 |   8 | init                           | 0.000061 | sql_select.cc        |         772 |
|        1 |   9 | System lock                    | 0.000171 | lock.cc              |         332 |
|        1 |  10 | optimizing                     | 0.000171 | sql_optimizer.cc     |         354 |
|        1 |  11 | statistics                     | 0.001304 | sql_optimizer.cc     |         694 |
|        1 |  12 | preparing                      | 0.000531 | sql_optimizer.cc     |         778 |
|        1 |  13 | executing                      | 0.002607 | sql_union.cc         |        1670 |
|        1 |  14 | end                            | 0.000042 | sql_select.cc        |         805 |
|        1 |  15 | query end                      | 0.000024 | sql_parse.cc         |        4881 |
|        1 |  16 | waiting for handler commit     | 0.000151 | handler.cc           |        1610 |
|        1 |  17 | closing tables                 | 0.000190 | sql_parse.cc         |        4944 |
|        1 |  18 | freeing items                  | 0.000164 | sql_parse.cc         |        5413 |
|        1 |  19 | cleaning up                    | 0.000060 | sql_parse.cc         |        2489 |
+----------+-----+--------------------------------+----------+----------------------+-------------+
18 rows in set, 1 warning (0.01 sec)
```

performance_schema

- `show profiles` 在后续 MySQL 中逐渐弃用，取而代之的是 performance_schema

```
'SHOW PROFILES' is deprecated and will be removed in a future release.
Please use Performance Schema instead
```

- 在配置文件中启用 performance_schema 参数，打开后将会收集所有用户的执行历史事件

```
[mysqld]
performance_schema=ON
```

- performance_schema 提供了多种表来存储监控数据，具体分为以下几类
  - Setup 表：用于配置 performance_schema 的行为，比如哪些监控器被启用
    1. setup_instruments 表来启用或禁用特定的监控器 (instruments)
  - Instances 表：包含有关监控实例（如线程、表等）的信息，例如：xxx_instances
  - Events 表：包含关于 MySQL 服务器事件（如语句执行、等待事件等）的信息，例如:
    1. events_waits_summary_by_instance：提供关于等待事件的汇总信息
    2. events_statements_summary_by_digest：提供关于 SQL 语句执行情况的汇总信息，按语句摘要分组
    3. events_transactions_summary_by_state：提供关于事务状态的汇总信息
  - Consumer 表：定义了如何消费（存储或汇总）performance_schema 中的数据

performance_schema 参数配置

1. 配置 setup_actors，开启监控的用户

```
select * from performance_schema.setup_actors;
```

2. 配置 setup_instruments，开启对 ‘statements’ 和 ‘stage’ 名称的配置

```
update performance_schema.setup_instruments set ENABLED = 'YES', TIMED = 'YES'
 where name like '%statement/%';
update performance_schema.setup_instruments set ENABLED = 'YES', TIMED = 'YES'
 where name like '%stage/%';
```

3. 配置 setup_consumer，启用 events_statements_*， events_stages_* 开头的事件类型消费

```
update performance_schema.setup_consumers set ENABLED = 'YES'
 where name like '%events_statements_%';
update performance_schema.setup_consumers set ENABLED = 'YES'
 where name like '%events_stages_%';
```

performance_schema 执行并收集结果

- 查询事件的 EVENT_ID

```
select EVENT_ID,
  truncate(TIMER_WAIT / 1000000000000, 6) as Duration,
  SQL_TEXT
  from performance_schema.events_statements_history_long
  where SQL_TEXT like '%empl%'\G
```

- 查询耗时结果

```
select
  event_name as Stage,
  truncate(TIMER_WAIT / 1000000000000, 6) as Duration
  from performance_schema.events_stages_history_long
  where NESTING_EVENT_ID = 422;
```

performance_schema 查询收集结果

```
mysql> select event_name as Stage,
    ->   truncate(TIMER_WAIT / 1000000000000, 6) as Duration
    ->   from performance_schema.events_stages_history_long
    ->   where NESTING_EVENT_ID = 422;
+------------------------------------------------+----------+
| Stage                                          | Duration |
+------------------------------------------------+----------+
| stage/sql/starting                             |   0.0010 |
| stage/sql/Executing hook on transaction begin. |   0.0000 |
| stage/sql/starting                             |   0.0001 |
| stage/sql/checking permissions                 |   0.0000 |
| stage/sql/checking permissions                 |   0.0000 |
| stage/sql/Opening tables                       |   0.0011 |
| stage/sql/init                                 |   0.0000 |
| stage/sql/System lock                          |   0.0002 |
| stage/sql/optimizing                           |   0.0002 |
| stage/sql/statistics                           |   0.0018 |
| stage/sql/preparing                            |   0.0006 |
| stage/sql/executing                            |   0.0028 |
| stage/sql/end                                  |   0.0000 |
| stage/sql/query end                            |   0.0000 |
| stage/sql/waiting for handler commit           |   0.0001 |
| stage/sql/closing tables                       |   0.0002 |
| stage/sql/freeing items                        |   0.0001 |
| stage/sql/cleaning up                          |   0.0000 |
+------------------------------------------------+----------+
18 rows in set (0.01 sec)
```

```
mysql> select EVENT_ID,
  -> truncate(TIMER_WAIT / 1000000000000, 6)
  ->   as Duration,
  -> SQL_TEXT
  -> from
  -> events_statements_history_long
  -> where SQL_TEXT like '%empl%';
...
****************** 2. row ******************
EVENT_ID: 422
Duration: 0.0091
SQL_TEXT: select
  x.dept_no,
  e.first_name,
  e.last_name
from
  dept_manager x
  left join
    employees e
      using (emp_no)
where
  x.dept_no < 'd004'
```

4

PFS 实现

performance_schema

- performance_schema [1] 的实现主要位于 ☞ storage/perfschema/ 目录下
  - 该目录下有很多代码，这些函数一般是 PSI_xxx 或 PFS_xxx 作为前缀
- performance_schema 使用了多种数据结构来存储和管理监控数据
  - 包括哈希表、链表、位图等
  - 这些数据结构的选择和实现在很大程度上决定了 performance_schema 的性能和可扩展性
- 事件的采集通常是通过在 MySQL 代码的关键位置插入钩子 (hooks) 来实现的
  - 这些钩子在特定事件发生时被触发
  - 调用 performance_schema 的相关函数来记录事件信息
  - 例如在之前说的在线程创建是有 PFS 的钩子 ☞ storage/perfschema/pfs.cc

```
extern "C" {
static void *pfs_spawn_thread(void *arg) {
  auto *typed_arg = (PFS_spawn_thread_arg *)arg;
  void *user_arg;
  void *(*user_start_routine)(void *);
```

- 收集到的数据后，用户可以根据需求进行汇总和整理
  - performance_schema 提供了 SQL 查询接口，以便用户可以方便地查询和分析

[1] https://dev.mysql.com/doc/dev/mysql-server/8.0.37/PAGE_PFS.html

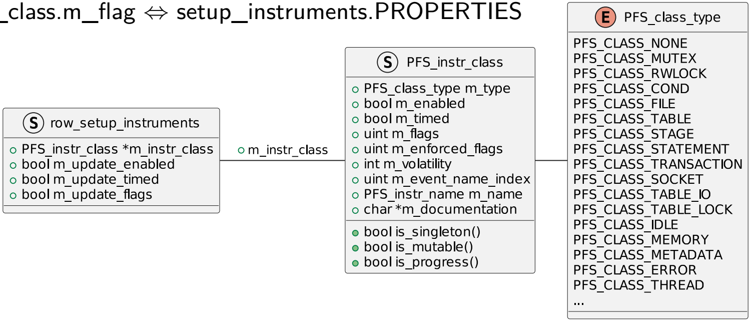
setup_instruments 表创建

- 表定义代码 ☞ storage/perfschema/table_setup_instruments.cc

```
Plugin_table table_setup_instruments::m_table_def(
    /* Schema name */
    "performance_schema",
    /* Name */
    "setup_instruments",
    /* Definition */
    "  NAME VARCHAR(128) not null,\n"
    "  ENABLED ENUM ('YES', 'NO') not null,\n"
    "  TIMED ENUM ('YES', 'NO'),\n"
    "  PROPERTIES SET('singleton', 'progress', 'user', 'global_statistics', "
    "'mutable', 'controlled_by_default') not null,\n"
    "  FLAGS SET('controlled'),\n"
    "  VOLATILITY int not null,\n"
    "  DOCUMENTATION LONGTEXT,\n"
    "  PRIMARY KEY (NAME) USING HASH\n",
    /* Options */
    " ENGINE=PERFORMANCE_SCHEMA",
    /* Tablespace */
    nullptr);
```

setup_instruments 表数据结构

- setup_instruments [2] 表中的行数据主要是通过 row_setup_instruments 类来存取的
- 元数据主要存储在 pfs_instr_class 对象中
- pfs_instr_class 对象描述了 instrument 的信息，该类和 setup_instruments 表对应如下:
    - pfs_instr_class.m_name ⇔ setup_instruments.NAME
    - pfs_instr_class.m_enabled ⇔ setup_instruments.ENABLED
    - pfs_instr_class.m_flag ⇔ setup_instruments.PROPERTIES
    - …



S row_setup_instruments
- ○ PFS_instr_class *m_instr_class
- ○ bool m_update_enabled
- ○ bool m_update_timed
- ○ bool m_update_flags

○ m_instr_class

S PFS_instr_class
- ○ PFS_class_type m_type
- ○ bool m_enabled
- ○ bool m_timed
- ○ uint m_flags
- ○ uint m_enforced_flags
- ○ int m_volatility
- ○ uint m_event_name_index
- ○ PFS_instr_name m_name
- ○ char *m_documentation
- ● bool is_singleton()
- ● bool is_mutable()
- ● bool is_progress()

E PFS_class_type
- PFS_CLASS_NONE
- PFS_CLASS_MUTEX
- PFS_CLASS_RWLOCK
- PFS_CLASS_COND
- PFS_CLASS_FILE
- PFS_CLASS_TABLE
- PFS_CLASS_STAGE
- PFS_CLASS_STATEMENT
- PFS_CLASS_TRANSACTION
- PFS_CLASS_SOCKET
- PFS_CLASS_TABLE_IO
- PFS_CLASS_TABLE_LOCK
- PFS_CLASS_IDLE
- PFS_CLASS_MEMORY
- PFS_CLASS_METADATA
- PFS_CLASS_ERROR
- PFS_CLASS_THREAD
- ...

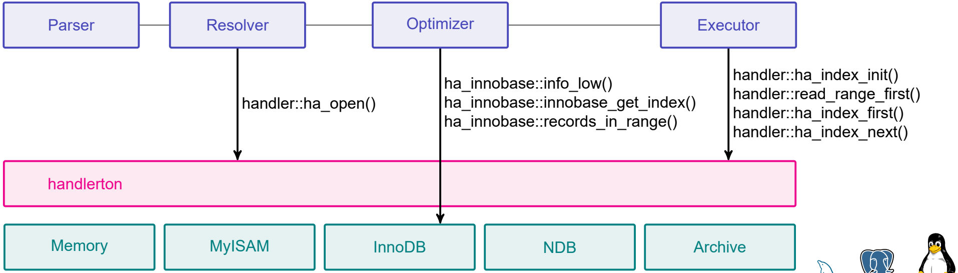
[2] https://dev.mysql.com/doc/refman/8.0/en/performance-schema-setup-instruments-table.html

5

存储引擎的接口层

handlerton

- handlerton 代表了 MySQL 存储引擎的接口层，(handlerton = table handler + singleton)
  - 存储引擎早期就叫表处理器 (table handler)
  - handlerton 是一个单例 (singleton) 的数据结构，每个存储引擎只用一个实例
  - handlerton 结构就是 MySQL 服务器与这些存储引擎之间交互的桥梁
  - 可以通过实现 handlerton 的接口来创建自己的存储引擎，并通过插件方式嵌入到 MySQL 中
- MySQL 支持多种存储引擎，每种存储引擎都提供了不同特性
  - 例如：存储机制、锁定级别、事务支持等
  - 常见的存储引擎包括：Innodb，MyISAM



Parser

Resolver

Optimizer

Executor

handler::ha_open()

ha_innobase::info_low()
ha_innobase::innobase_get_index()
ha_innobase::records_in_range()

handler::ha_index_init()
handler::read_range_first()
handler::ha_index_first()
handler::ha_index_next()

handlerton

Memory

MyISAM

InnoDB

NDB

Archive

handlerton 实现

- handlerton [3] 包含了以下几个关键部分
  - 状态信息：如存储引擎的名称、是否支持事务、是否支持外键等
  - 函数指针：指向存储引擎实现的函数
    1. 从表的创建、删除、打开、关闭
    2. 行的插入、删除、更新、读取
    3. 索引的创建、删除、查询等
  - 代码中实现 `handler` 抽象类，使用 `ha_` 作为大部分函数名称前缀
- handlerton 在 ☞ sql/handler.h 中声明，在 ☞ sql/handler.cc 中实现

```
struct handlerton {
  /**
    Historical marker for if the engine is available or not.
  */
  SHOW_COMP_OPTION state;

  /**
    Historical number used for frm file to determine the correct storage engine.
    This is going away and new engines will just use "name" for this.
  */
  enum legacy_db_type db_type;
```

[3] https://dev.mysql.com/doc/dev/mysql-server/8.0.37/structhandlerton.html

handlerton 的实例 xxx_hton

- handlerton 的实例通常静态定义在 ha_xxx.cc 中，服务层通过函数指针来调用

```
// 通常定义成
static handlerton { ... } xxx_hton;
// 具体的 hton 实例例如
extern handlerton *myisam_hton;
extern handlerton *heap_hton;
extern handlerton *temptable_hton;
extern handlerton *innodb_hton;
```

- 例如：InnoDB 的 hton 实例，可以通过 gdb 打印其内容

```
(gdb) p *innodb_hton
$10 = {
  state = SHOW_OPTION_YES,
  db_type = DB_TYPE_INNODB,
  slot = 1,
  savepoint_offset = 0,
  close_connection = 0x55555a661380 <innobase_close_connection(handlerton*, THD*)>,
  kill_connection = 0x55555a66181f <innobase_kill_connection(handlerton*, THD*)>,
  pre_dd_shutdown = 0x55555a654ba1 <innodb_pre_dd_shutdown(handlerton*)>,
  savepoint_set = 0x55555a661170 <innobase_savepoint(handlerton*, THD*, void*)>,
  savepoint_rollback = 0x55555a660c46 <innobase_rollback_to_savepoint(handlerton*, THD*, void*)>,
  ...
```

结束