# MySQL自定义存储引擎

MySQL服务器采用模块化的方式构建，很多功能通过插件来实现，存储引擎也是如此，存储引擎负责管理MySQL的数据存储和索引管理。MySQL服务器通过定义好的API与存储引擎进行通信。每个存储引擎是一个类，该类的每个实例通过特殊的handler接口与MySQL服务器通信 。对于每一个链接线程，MySQL会为每一个表创建一个与该表对应的handler的实例，即：MySQL中，存储引擎是针对表的。例如：如果三个连接都开始使用同一个表，则需要创建三个处理程序实例。一旦处理程序实例被创建，MySQL服务器向处理程序发出命令来执行数据存储和检索任务，例如打开表，操纵行和管理索引等。

自定义存储引擎可建以渐进的方式进行：开发人员可以从只读存储引擎开始，随后慢慢支持INSERT，UPDATE以及DELETE，甚至在以后添加对索引，事务等先进操作的支持。

实现新的存储引擎的最简单的方法是从复制和修改EXAMPLE存储引擎开始。该存储引擎由文件ha\_example.cc和 ha\_example.h实现，它们位于MySQL的源码目录： storage/example中。

存储引擎也是一个插件，有关为插件添加系统变量、状态变量、配置文件中的配置项相关的知识，在MySQL插件中学习。接下来讨论存储引擎的实现。

MySQL中，MySQL服务器层与存储引擎交互的基础数据结构和类都在handler.h中定义。现在我们进行分析，并结合具体的存储引擎实现进行讨论。

1、is\_supported\_system\_table

handlerton用于代表一个存储引擎，每个存储引擎有一个该类的实例，注意，该类与每张表有关的处理程序（每个TABLE数据结构会有一个处理程序实例，TABLE实例在每个链接的每个表上创建一个）不同。该实例由MySQL服务层在插件初始化的时候创建，然后会以该实例的地址为参数调用我们对应的存储引擎实现的初始化函数，通常而言，为了方便的获取我们存储引擎对应的handlerton实例，通常会在ha\_xxx.cc中定义一个静态指针，并在存储引擎实现的初始化函数中设置该指针的值为MySQL服务层创建并传递下来的handlerton实例的地址，例如，对于InnoDB如下：在storage/innobase/handler/ha\_innodb.cc中：

119 static struct handlerton\* innodb\_hton\_ptr;

2923

static int innobase\_init(void \*p) /\*!< in: InnoDB handlerton \*/

{

    ......

    DBUG\_ENTER("innobase\_init");

    handlerton \*innobase\_hton= (handlerton\*) p;

    innodb\_hton\_ptr = innobase\_hton;

    innobase\_hton->state = SHOW\_OPTION\_YES;

innobase\_hton->db\_type= DB\_TYPE\_INNODB;

.........

//一系列的初始化,随着分析的深入，剩下的内容慢慢探讨。

}

现在我们看看handlerton的数据成员以及成员函数，分析一下每个成员的意义。

struct handlerton

{

SHOW\_COMP\_OPTION state;

enum legacy\_db\_type db\_type;

uint slot;

uint savepoint\_offset;

int (\*close\_connection)(handlerton \*hton, THD \*thd);

int (\*savepoint\_set)(handlerton \*hton, THD \*thd, void \*sv);

int (\*savepoint\_rollback)(handlerton \*hton, THD \*thd, void \*sv);

bool (\*savepoint\_rollback\_can\_release\_mdl)(handlerton \*hton, THD \*thd);

int (\*savepoint\_release)(handlerton \*hton, THD \*thd, void \*sv);

int (\*commit)(handlerton \*hton, THD \*thd, bool all);

int (\*rollback)(handlerton \*hton, THD \*thd, bool all);

int (\*prepare)(handlerton \*hton, THD \*thd, bool all);

int (\*recover)(handlerton \*hton, XID \*xid\_list, uint len);

int (\*commit\_by\_xid)(handlerton \*hton, XID \*xid);

int (\*rollback\_by\_xid)(handlerton \*hton, XID \*xid);

void \*(\*create\_cursor\_read\_view)(handlerton \*hton, THD \*thd);

void (\*set\_cursor\_read\_view)(handlerton \*hton, THD \*thd, void \*read\_view);

void (\*close\_cursor\_read\_view)(handlerton \*hton, THD \*thd, void \*read\_view);

handler \*(\*create)(handlerton \*hton, TABLE\_SHARE \*table, MEM\_ROOT \*mem\_root);

void (\*drop\_database)(handlerton \*hton, char\* path);

int (\*panic)(handlerton \*hton, enum ha\_panic\_function flag);

int (\*start\_consistent\_snapshot)(handlerton \*hton, THD \*thd);

bool (\*flush\_logs)(handlerton \*hton);

bool (\*show\_status)(handlerton \*hton, THD \*thd, stat\_print\_fn \*print, enum ha\_stat\_type stat);

uint (\*partition\_flags)();

uint (\*alter\_table\_flags)(uint flags);

int (\*alter\_tablespace)(handlerton \*hton, THD \*thd, st\_alter\_tablespace \*ts\_info);

int (\*fill\_is\_table)(handlerton \*hton, THD \*thd, TABLE\_LIST \*tables,

class Item \*cond,

enum enum\_schema\_tables);

uint32 flags; /\* global handler flags \*/

int (\*binlog\_func)(handlerton \*hton, THD \*thd, enum\_binlog\_func fn, void \*arg);

void (\*binlog\_log\_query)(handlerton \*hton, THD \*thd,

enum\_binlog\_command binlog\_command,

const char \*query, uint query\_length,

const char \*db, const char \*table\_name);

int (\*release\_temporary\_latches)(handlerton \*hton, THD \*thd);

enum log\_status (\*get\_log\_status)(handlerton \*hton, char \*log);

enum handler\_create\_iterator\_result

(\*create\_iterator)(handlerton \*hton, enum handler\_iterator\_type type,

struct handler\_iterator \*fill\_this\_in);

int (\*discover)(handlerton \*hton, THD\* thd, const char \*db,

const char \*name,

uchar \*\*frmblob,

size\_t \*frmlen);

int (\*find\_files)(handlerton \*hton, THD \*thd,

const char \*db,

const char \*path,

const char \*wild, bool dir, List<LEX\_STRING> \*files);

int (\*table\_exists\_in\_engine)(handlerton \*hton, THD\* thd, const char \*db,

const char \*name);

int (\*make\_pushed\_join)(handlerton \*hton, THD\* thd,

const AQP::Join\_plan\* plan);

const char\* (\*system\_database)();

bool (\*is\_supported\_system\_table)(const char \*db,

const char \*table\_name,

bool is\_sql\_layer\_system\_table);

uint32 license; /\* Flag for Engine License \*/

void \*data; /\* Location for engines to keep personal structures \*/

};

如上，其中的成员都是针对整个存储引擎的，而不是基于每个表的基础工作方法。

虽然有40多个成员，但是很多并不是必须的。只有少数几个是必须的。

作者：许富博

版权所有，文章以学习和交流为主，切勿用于商业用途。

限于本人水平有限，欢迎大家随时指正，联系方式：

[xufubobo@gmail.com](mailto:xufubobo@gmail.com)

[xufubobo@163.com](mailto:xufubobo@163.com)

[1332841493@qq.com](mailto:1332841493@qq.com)