一、代码逻辑

1、cdb_test.cc ----main()

- cdb_test.cc----**handle_line()**
- cdb_test.cc----add_history()
- cdb_test.cc----q.find(":load")(判断输入的语句是否是包含load，从文件中加载指令，否)
- **cdb_test.cc----Schema_Cache**
-
  - **getDefaultDatabaseForConnection(建立连接)**
  - **executeQuery**

```
static bool handle_line(ProxyState& ps, const std::string& q, bool pp=true)
{
  if (q == "\\q") {
    std::cerr << "Goodbye!\n";
    return false;
  }

  add_history(q.c_str());

  // handle meta inputs 以:load name 标志从文件输入
  if (q.find(":load") == 0) {
  //从文件中读取SQl语句
  }

  static SchemaCache schema_cache;
  try {
      const std::string &default_db =
          getDefaultDatabaseForConnection(ps.getConn());

      std::cout << "在函数handle_line中\ndefault_db: " + default_db <<
std::endl;
      const EpilogueResult &epi_result =executeQuery(ps, q, default_db,
&schema_cache, pp);
```

参数：const ProxyState &ps, const stdstring &q,const stdstring &default_db, SchemaCache *const schema_cache, bool pp（SQl语句重写，执行，获得结果）

- assert（schema_cache)

- **queryPreamble（ps, q, &qr, &out_queryz, schema_cache, default_db)**

3、rewrite_util.cc----queryPreamble

rewrite类存储重写之后的SQl语句或解密后的结果，通过调用Rewriter::rewrite函数对SQL语句重写，再创建一个QueryRewrite对象。

```
*qr = std::unique_ptr\<QueryRewrite>( new QueryRewrite(Rewriter::rewrite(ps, q, schema, default_db)));


if (cryptdbDirective(q)) { // 不需要进行重写
    output = Rewriter::handleDirective(analysis, ps, q);
} else {
    output = Rewriter::dispatchOnLex(analysis, ps, q); // 进行重写
    if (!output) {
        output = new SimpleOutput(mysql_noop());
    }
}
```

调用dispatchOnLex进行重写

5、rewrite_main.cc----Rewriter::dispatchOnLex

- analysis.hh----getDatabaseName(return db_name:db)

- new query_parse(a.getdatabasename(),query) 生成query_parse指针，存储lex对象

```
if (noRewrite(*lex))
{
    return new SimpleOutput(query);//不重写
}
else if (dml_dispatcher->canDo(lex))
```

```
{
    const SQLHandler &handler = dml_dispatcher->dispatch(lex);
    AssignOnce<LEX *> out_lex;
}
else if (ddl_dispatcher->canDo(lex))
{
  const SQLHandler &handler = ddl_dispatcher->dispatch(lex);
    LEX *const out_lex = handler.transformLex(a, lex, ps);
        // HACK.
    const std::string &original_query =
        lex->sql_command != SQLCOM_LOCK_TABLES ? query : "do 0";
  return new
DDLOutput(original_query,lex_to_query(out_lex),std::move(a.deltas));
    }
else
{
    return NULL;
 }
```

分dml，ddl和不重写三种类型，Dispatcher根据lex的结构，为SQL语句分配handler，不同类型的SQL语句有不同的handler，用来处理sql语句的加密。加密以后的结果则放置在executor中.

1561944267772例子是create 是ddl，

- transformLex时跳转到**ddl_handler.cc----transformLex**

（return this->**rewriteAndUpdate**(a, lex, ps);rewriteandupdata很复杂 最后return new lex

- DDLOutput(original_query,lex_to_query(out_lex),std::move(a.deltas));这里又跳转到**rewrite_main.cc----lex_to_query**，输出语句

ddl系列类型的典型处理流程包含了两部分，一是对SQL语句进行加密，二是以delta类来基于数据库的变化，这个delta在next阶段会写入到本地的embedded数据库中。例如CREATE TABLE语句，delta需要记录添加的表的名字， 表有多少列，每列分别采用什么样的加密算法。这些功能全都实现在rewriteAndUpdate函数中了。

DMLhandler:在transformlext函数内部，包含了gather和rewrite两个函数，首先获得rewrite plain，然后根据rewrite plain多lex的内部成员进行改写，最后返回的executor，这里的executor类型是ShowTableExecutor。

```
 LEX *DMLHandler::transformLex(Analysis &analysis, LEX *lex,
                               const ProxyState &ps) const
{
    this->gather(analysis, lex, ps);
    return this->rewrite(analysis, lex, ps);

}
```

- rewrite函数：从fieldmeta到valuemeta

  至此4的rewrite全过程调用完毕，回到3，继续rewrite_util.cc---queryPreamble
- updatastaleness
- usesembeddedDB
- beforeQuery
- getQuery(将重写结果从*qr存入out_queryz中)

  至此3的queryPreamble全过程调用完毕，回到2，继续rewrite_main.cc----executeQuery
- assert（qr)
- 循环判断SQL语句是否为对数据库进行变更，新建或是删除，如果是就新建一个连接

```
 for (auto it : out_queryz) {
        if (true == pp) {
            prettyPrintQuery(it); // 输出重写后的SQL语句，以QUERY:为标识
        }
        if ((strncmp(toLowerCase(it).c_str(), "use", 3) == 0) ||
              (strncmp(toLowerCase(it).c_str(), "create database", 15) == 0)
||
              (strncmp(toLowerCase(it).c_str(), "drop database", 13) == 0)
              ) {
          std::cout << "SQL语句执行: " + it << std::endl;
          TEST_Sync(ps.getConn()->execute(it, &dbres,
                                          qr->output-
>multipleResultSets()),
                           "failed to execute query!");
        }
 }
```

本例子中最后输出三个

> INSERT INTO remote_db.generic_prefix_remoteQueryCompletion (begin, complete, embedded_completion_id, reissue) VALUES (TRUE, FALSE,167 , FALSE);create table

> table_VXRNZIQJSE (WXFOYISCAJoDET BIGINT(8) unsigned, QXGKNHUDRPoOPE BIGINT(11) unsigned, KAQHLORKVDoAGG VARBINARY(256), cdb_saltHBVIVJZHMR BIGINT(8) unsigned not null) ENGINE=InnoDBUPDATE remote_db.generic_prefix_remoteQueryCompletion　SET complete = TRUE　WHERE embedded_completion_id = 167;* assert(res.successs()) 进行解密以及输出解密之后的结果

- pi_result =queryEpilogue(ps, *qr.get(), res, q, default_db, pp);
- assert(epi_result.res_type.success());

最后返回epi_result，**executeQuery函数总结如下**

- **queryPreamble：sql语句重写**
- **ps.getConn()->execute：语句执行**
- **queryEpilogue：数据集解密**
  至此2的executeQuery全过程调用完毕，实现了语句的重写与执行，以及结果的解密，回到1，继续cdb_test.cc ----main()
- **EpilogueResult**

二、元数据存储

SchemaCache是对SchemaInfo进行的简单封装，SchemaInfo继承了MappedDBMeta，通过Key-Value的形式保存了当前CryptDB中创建了的所有数据库信息。

1561863787080

```
class SchemaCache {
public:
    SchemaCache() : no_loads(true), id(randomValue() % UINT_MAX) {}


const SchemaInfo &getSchema(const std::unique_ptr<Connect> &conn,
                            const std::unique_ptr<Connect> &e_conn);
void updateStaleness(const std::unique_ptr<Connect> &e_conn,
                     bool staleness);
bool initialStaleness(const std::unique_ptr<Connect> &e_conn);
bool cleanupStaleness(const std::unique_ptr<Connect> &e_conn);
void lowLevelCurrentStale(const std::unique_ptr<Connect> &e_conn);
void lowLevelCurrentUnstale(const std::unique_ptr<Connect> &e_conn);


private:
    std::unique_ptr<const SchemaInfo> schema;
```

```
    bool no_loads;           //设置state状态
    const unsigned int id;   //随机ID
};
```

首先就是获得一个id随机的SchemaCache结构，内部包含了一个空的SchemaInfo成员。如果之前已经建立了数据库，或者数据库下已经有一些表，则需要读取embedded MySQL中的元数据，**反序列化**，用内存的数据结构来表示。

三、加密算法相关

加密层——util/onions.hh

- 通过枚举类型表示各种加密类型

```
typedef enum onion {
    oDET,
    oOPE,
    oAGG,
    oSWP,
    oPLAIN,
    oBESTEFFORT,
    oINVALID,
} onion;
```

```
enum class SECLEVEL {
    INVALID,
    PLAINVAL,
    OPE,
    DETJOIN,
    DET,
    SEARCH,
    HOM,
    RND,
};
```

```
第一块的一种加密方案，要用下面的多种算法洋葱加密，安全性越低的算法，越早加密。最外层基本都是RND。
```

```
static onionlayout STR_ONION_LAYOUT = {
    {oDET, stdvector<SECLEVEL>({SECLEVELDETJOIN, SECLEVEL::DET,
                                SECLEVEL::RND})},
```

{oOPE, stdvector<SECLEVEL>({SECLEVELOPE, SECLEVEL::RND})},

* Enclayer层继承自LeafDBMeta，定义了newCreateField，encrypt，decrypt，decryptUDF函数，其中decryrptUDF函数用来调整洋葱层结构。
* 具体的加密算法的实现继承Enclayer：HOM，SEARCH等等

加密层的管理——main/CryptoHandlers.cc

*
加密层的创建依靠LayerFactory结构，不同的加密层有自己的factory。而这些factory又通过EncLayerFactory类来实现管理。LayerFactory系列的类，主要提供了create和deserialize函数，前者用于在内存中直接创建加密层，后者用于对磁盘读取的数据做反序列化来创建加密层

> - RNDFactory: outputs a RND layer - RND layers: RND_int for blowfish, RND_str for AES -DETFactory: outputs a DET layer - DET layers: DET_int, DET_str -OPEFactory: outputs a OPE layer - OPE layers: OPE_int, OPE_str, OPE_dec -HOMFactory: outputs a HOM layer - HOM layers: HOM (for integers), HOM_dec (for decimals)

*
其他helper包括序列化helper，Factory implementions和其他（prng_pad）

*
serialize函数实现了加密层的序列化

*
反序列化功能在LayerFactory管理类中实现

*
加解密函数encrypt和decrypt，是对上面介绍的crypto目录中的底层库的封装。由于这里处理的都是item类型，所以需要进行item类型和普通数据类型的互相转换

*
decryptUDF用于返回一个UDF函数，做洋葱层次调整。

decryptUDF函数用于洋葱层次的调整。举例来说，当一个查询需要使用where xx=xx的条件时，需要使用洋葱层次DET，而如果此时洋葱的实际层次是RND，则需要在MySQL端执行解密函数，剥掉RND层。这个操作通过UDF来完成，而decryptUDF就是用来生成这个UDF语句的。

*
newCreateField用来处理加密带来的数据类型的变化。比如原来是整数类型，经过了Pailliar的加密，就变成了二进制字符串类型。

newCreateField函数是为了处理数据类型的变化：数据经过加密算法的处理，其数据类型和数据长度会发生变化，加密层的newCreateField要能够返回加密以后的数据类型。这种类型的信息封装在Create_field类里面了，这也是MySQL的parser中定义的类，具体细节不在此展开。

CryptDB需要对MySQL的parser中的LEX结构中的Item类型做加密，底层加密库不能直接处理Item，所以在EncLayer中要做一个封装，这部分的内容主要实现在main/CryptoHandlers.cc，用于处理数据类型的转化。此外，EncLayer还需要处理序列化，UDF返回等功能。为了辅助加密层类型的使用，设计了LayerFactory系列的类，用于构造加密层类，这个构造分为普通构造和反序列化构造。这些factory类又通过EncLayerFactory类型来进行统一管理。通过这些机制，底层的加密库就和CryptDB的实现连接起来了，CryptDB会调用封装好的EncLayer，而不直接使用底层的加密库。

四、元数据管理结构

数据库与其中的表—DatabaseMeta，通过std:map类型保存

（map中的key是对IdentityMetaKey类型，是对table1的封装，value是TableMeta类型，代表一个表的元数据）

在CryptDB中，每个明文的表名都被替换成了密文的表名。其中明文的表名被封装成了*IdentityMetaKey*，存储在DatabaseMeta内部的Map中作为key，加密替换以后的表名则存储在*TableMeta*中的成员*anon_table_name*中。这样，在通过明文的表名做Key，找到对应的TableMeta类型的value时，可以从其类成员anno_table_name得到加密的表名。明文和密文的对应关系就是这样存储的。

表与其中对应的列—TableMeta

counter，每次加一，记录每一列的顺序，传递到新建的FieldMeta结构中，这样一个TableMeta下的FieldMeta就可以根据这个counter的值进行排序了。

每一个原始列对应多个洋葱层列—FieldMeta中的onion_layout

Field每部包含多个洋葱加密模型，每个洋葱代表了加密表中的一个列，洋葱层也有顺序

onion_layout是一个map结构，key是洋葱类型，value是一个vector，表示洋葱的各个层次。

例如：

```
onionlayout NUM_ONION_LAYOUT = {
   {oDET, stdvector<SECLEVEL>({SECLEVELDETJOIN, SECLEVEL::DET,
                               SECLEVEL::RND})},
   {oOPE, stdvector<SECLEVEL>({SECLEVELOPE, SECLEVEL::RND})},
   {oAGG, stdvector<SECLEVEL>({SECLEVELHOM})}
};
```

没有map，直接继承DBMeta。

OnionMeta代表了一个洋葱。在CryptDB中，一个洋葱有很多的层次，每个层次代表一次加密，原始列的数据被这个洋葱中的多个层次依次进行加密。加密以后的列有列名，通过这里的onionname成员来记录。uniq_count成员则是用于onionmeta的排序，之前已经做过介绍。layers成员是通过vector类型来对加密层次进行记录。序列化和反序列化的函数和前面的类似，这里不再给出。

最后的Enclayer

EncLayer代表了一个加密层次的抽象，所以其首先应该有加密和解密函数，用于对数据做加解密。在这个层次，数据的加解密的对象是*Item*，这是一个MySQL的parser中定义的类型，代表了解析以后的SQL语句的语法树中的一个节点。

补充

* /*补充分析一下CryptoHandlers.cc中有关OPE的代码*/

继承Enclayer有三个类，OPE_init、OPE_tinyint、OPE_mediumint、OPE_str(1,4一样；2,3一样)

```
class OPE_int : public EncLayer {
public:
    OPE_int(Create_field * const cf, const std::string &seed_key);

     serialize and deserialize
    std::string doSerialize() const {return key;}
    OPE_int(unsigned int id, const std::string &serial);

    SECLEVEL level() const {return SECLEVEL::OPE;}
    std::string name() const {return "OPE_int";}
    Create_field * newCreateField(const Create_field * const cf,
                                  const std::string &anonname = "")
        const;

    Item *encrypt(const Item &p, uint64_t IV) const;
    Item * decrypt(Item * const c, uint64_t IV) const;



private:
    std::string const key;
     HACK.
    mutable OPE ope;
    static const size_t key_bytes = 16;
    static const size_t plain_size = 4;
```

```
    static const size_t ciph_size = 8;
};
```

```
class OPE_str : public EncLayer {
public:
    OPE_str(Create_field * const cf, const std::string &seed_key);

    // serialize and deserialize
    std::string doSerialize() const {return key;}
    OPE_str(unsigned int id, const std::string &serial);

    SECLEVEL level() const {return SECLEVEL::OPE;}
    std::string name() const {return "OPE_str";}
    Create_field * newCreateField(const Create_field * const cf,
                                  const std::string &anonname = "")
        const;

    Item *encrypt(const Item &p, uint64_t IV) const;
    Item * decrypt(Item * const c, uint64_t IV) const
        __attribute__((noreturn));

private:
    std::string const key;
    // HACK.
    mutable OPE ope;
    static const size_t key_bytes = 16;
    static const size_t plain_size = 4;
    static const size_t ciph_size = 8;
};
```

FieldMeta中的onion_layout确定了加密洋葱层，在OnionMeta的vector中，Enlayer层实现具体的加密算法，CryptoHandlers.cc中是对底层加密算法代码（crypto文件夹中的）的封装

- 序列化与反序列化

序列化 (Serialization)是将对象的状态信息转换为可以存储或传输的形式的过程。在序列化期间，对象将其当前状态写入到临时或持久性存储区。以后，可以通过从存储区中读取或反序列化对象的状态，重新创建该对象。

CREATE DATABASE db创建一个数据库db的时候，CryptDB会生成一个DatabaseMeta结构

来表示这个新的数据库，并把这个信息序列化以后写入到embedded MySQL中。(在CryptDB中，需要对加密过程进行记录：比如某个表的原始名字和加密以后的名字，表中有多少列，每列用了什么样的加密算法。这些信息被记录在mysql-proxy端的embedded MySQL中。)

2019年11月 刘一静