# cqlsh代码流程

## start入口

bin/cqlsh

-main-

->main(\*read\_options(sys.argv[1:], os.environ))

->def main(options, hostname, port):

->shell.cmdloop()

->cmdloop(self): //进入了cmd loop

## select入口

bin/cqlsh

1. def do\_select(self, parsed):

 tracing\_was\_enabled = self.tracing\_enabled

      ksname = parsed.get\_binding('ksname')

        stop\_tracing = ksname == 'system\_traces' or (ksname is None and self.current\_keyspace == 'system\_traces')

        self.tracing\_enabled = self.tracing\_enabled and not stop\_tracing

        statement = parsed.extract\_orig()

        with\_default\_limit = parsed.get\_binding('limit') is None

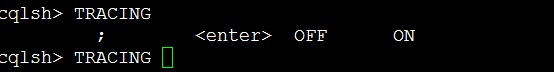
        if with\_default\_limit:

            statement = "%s LIMIT %d;" % (statement[:-1], DEFAULT\_SELECT\_LIMIT)

        self.perform\_statement(statement, with\_default\_limit=with\_default\_limit) #进入查询过程

        self.tracing\_enabled = tracing\_was\_enabled

可以使用tracing 开关，在shell命令行通过如下命令打开



1. self.perform\_statement(statement, decoder=ErrorHandlingSchemaDecoder, with\_default\_limit=with\_default\_limit)；
2. perform\_statement\_untraced(self, statement, decoder=None, with\_default\_limit=False)

untraced 的状态下解析代码；

1. self.cursor.execute(statement, decoder=decoder) #执行代码；

->self.cursor = self.conn.cursor()

-> self.conn = cql.connect(hostname, port, user=username, password=password,

                       cql\_version=cqlver, transport=transport); #cqlsh打开的时候连接集群

1. import cql 是在 lib/cql-internal-only-1.4.1.zip 模块里面实现的；
2. open  connection.py ；

-> def cursor(self):

        if not self.open\_socket:

            raise ProgrammingError("Connection has been closed.")

        curs = self.cursorclass(self)

        curs.compression = self.compression

        curs.consistency\_level = self.consistency\_level

        return curs

1. open cursor.py

-> def execute(self, cql\_query, params={}, decoder=None, consistency\_level=None):

        # note that 'decoder' here is actually the decoder class, not the

        # instance to be used for decoding. bad naming, but it's in use now.

        if isinstance(cql\_query, unicode):

            raise ValueError("CQL query must be bytes, not unicode")

        self.pre\_execution\_setup() #执行环节设置

        prepared\_q = self.prepare\_inline(cql\_query, params)

        cl = consistency\_level or self.consistency\_level

        response = self.get\_response(prepared\_q, cl) #获得执行结果

        return self.process\_execution\_results(response, decoder=decoder)

1. 下载代码需要深入分析

查找.get\_response的分析过程

1. $ grep 'get\_response' ./\*   
   ./cursor.py: response = self.get\_response(prepared\_q, cl)   
   ./cursor.py: response = self.get\_response\_prepared(prepared\_query, params, cl)   
   ./native.py: def get\_response(self, query, consistency\_level):   
   ./native.py: def get\_response\_prepared(self, prepared\_query, params, consistency\_level):   
   ./thrifteries.py: def get\_response(self, cql\_query, consistency\_level):  # 经过测试，该文件实现了前面的功能；  
   ./thrifteries.py: def get\_response\_prepared(self, prepared\_query, params, consistency\_level):
2. ./thrifteries.py: def get\_response(self, cql\_query, consistency\_level):

 def get\_response(self, cql\_query, consistency\_level):

        compressed\_q, compress = self.compress\_query\_text(cql\_query)  //语句进行了压缩

        print "thrift\_getresponse"

        cl = getattr(ConsistencyLevel, consistency\_level)

        if **self.use\_cql3\_methods: #cassandra 默认为cql3.0**

            doquery = self.\_connection.client.execute\_cql3\_query #获得执行方法

            return self.handle\_cql\_execution\_errors(doquery, compressed\_q, compress, cl)  //

        else: #查看方法执行过程

            doquery = self.\_connection.client.execute\_cql\_query

            return self.handle\_cql\_execution\_errors(doquery, compressed\_q, compress)

1. def handle\_cql\_execution\_errors(self, executor, \*args, \*\*kwargs):

        try:

            return executor(\*args, \*\*kwargs) //调用executor方法执行出来结果，所有需要关注executor方法

        except InvalidRequestException, ire:

            raise cql.ProgrammingError("Bad Request: %s" % ire.why)

        except SchemaDisagreementException, sde:

            raise cql.IntegrityError("Schema versions disagree, (try again later).")

        except UnavailableException:

            raise cql.OperationalError("Unable to complete request: one or "

                                       "more nodes were unavailable.")

        except TimedOutException:

            raise cql.OperationalError("Request did not complete within rpc\_timeout.")

        except TApplicationException, tapp:

1. doquery = self.\_connection.client.execute\_prepared\_cql3\_query

[ lib]$ grep -r 'prepare\_cql3\_query' ./cql-1.4.1/\*   
./cql-1.4.1/cql/thrifteries.py: doquery = self.\_connection.client.prepare\_cql3\_query   
./cql-1.4.1/cql/cassandra/Cassandra.py: def prepare\_cql3\_query(self, query, compression):   
./cql-1.4.1/cql/cassandra/Cassandra.py: def prepare\_cql3\_query(self, query, compression):   
./cql-1.4.1/cql/cassandra/Cassandra.py: self.send\_prepare\_cql3\_query(query, compression)   
./cql-1.4.1/cql/cassandra/Cassandra.py: return self.recv\_prepare\_cql3\_query()   
./cql-1.4.1/cql/cassandra/Cassandra.py: def send\_prepare\_cql3\_query(self, query, compression):   
./cql-1.4.1/cql/cassandra/Cassandra.py: self.\_oprot.writeMessageBegin('prepare\_cql3\_query', TMessageType.CALL, self.\_seqid)   
./cql-1.4.1/cql/cassandra/Cassandra.py: args = prepare\_cql3\_query\_args()   
./cql-1.4.1/cql/cassandra/Cassandra.py: def recv\_prepare\_cql3\_query(self, ):   
./cql-1.4.1/cql/cassandra/Cassandra.py: result = prepare\_cql3\_query\_result()   
./cql-1.4.1/cql/cassandra/Cassandra.py: raise TApplicationException(TApplicationException.MISSING\_RESULT, "prepare\_cql3\_query failed: unknown result");   
./cql-1.4.1/cql/cassandra/Cassandra.py: self.\_processMap["prepare\_cql3\_query"] = Processor.process\_prepare\_cql3\_query   
./cql-1.4.1/cql/cassandra/Cassandra.py: def process\_prepare\_cql3\_query(self, seqid, iprot, oprot):   
./cql-1.4.1/cql/cassandra/Cassandra.py: args = prepare\_cql3\_query\_args()   
./cql-1.4.1/cql/cassandra/Cassandra.py: result = prepare\_cql3\_query\_result()   
./cql-1.4.1/cql/cassandra/Cassandra.py: result.success = self.\_handler.prepare\_cql3\_query(args.query, args.compression)   
./cql-1.4.1/cql/cassandra/Cassandra.py: oprot.writeMessageBegin("prepare\_cql3\_query", TMessageType.REPLY, seqid)   
./cql-1.4.1/cql/cassandra/Cassandra.py:class prepare\_cql3\_query\_args:   
./cql-1.4.1/cql/cassandra/Cassandra.py: oprot.writeStructBegin('prepare\_cql3\_query\_args')   
./cql-1.4.1/cql/cassandra/Cassandra.py:class prepare\_cql3\_query\_result:   
./cql-1.4.1/cql/cassandra/Cassandra.py: oprot.writeStructBegin('prepare\_cql3\_query\_result')

1. open cql-1.4.1/cql/cassandra/Cassandra.py

 def execute\_cql3\_query(self, query, compression, consistency):

    """

    Parameters:

     - query

     - compression

     - consistency

    """

    print "execute\_cql3\_query(self, query, compression, consistency)"

    self.send\_execute\_cql3\_query(query, compression, consistency)  #发送请求

    return self.recv\_execute\_cql3\_query()  #接受请求

1. def send\_execute\_cql3\_query(self, query, compression, consistency):

    self.\_oprot.writeMessageBegin('execute\_cql3\_query', TMessageType.CALL, self.\_seqid)

    args = execute\_cql3\_query\_args()

    args.query = query

    args.compression = compression

    args.consistency = consistency

    args.write(self.\_oprot)

    self.\_oprot.writeMessageEnd()

    self.\_oprot.trans.flush()

消息内容如下

| string | 32bit | 32bit |

| execute\_cql3\_query'| TMessageType.CAL| self.\_seqid |

-------------------args----------------------------------

end

将请求类型发送到thrift端口，有固定的通信协议。

1. grep -r 'writeMessage' ./\*

./lib/thrift/protocol/TProtocol.py: def writeMessageBegin(self, name, ttype, seqid):

./lib/thrift/protocol/TProtocol.py: def writeMessageEnd(self):

./lib/thrift/protocol/TJSONProtocol.py: def writeMessageBegin(self, name, request\_type, seqid):

./lib/thrift/protocol/TJSONProtocol.py: def writeMessageEnd(self):

./lib/thrift/protocol/TJSONProtocol.py: def writeMessageBegin(self, name, request\_type, seqid):

./lib/thrift/protocol/TJSONProtocol.py: def writeMessageEnd(self):

./lib/thrift/protocol/TBinaryProtocol.py: def writeMessageBegin(self, name, type, seqid):

./lib/thrift/protocol/TBinaryProtocol.py: def writeMessageEnd(self):

使用了thrift接口写入端口信息

1. vim ./lib/thrift/protocol/TBinaryProtocol.py

def writeMessageBegin(self, name, type, seqid):

if self.strictWrite:

self.writeI32(TBinaryProtocol.VERSION\_1 | type)

self.writeString(name)

self.writeI32(seqid)

else:

self.writeString(name)

self.writeByte(type)

self.writeI32(seqid)

1. vim lib/cql-1.4.1/cql/cassandra/Cassandra.py

def write(self, oprot):

if oprot.\_\_class\_\_ == TBinaryProtocol.TBinaryProtocolAccelerated and self.thrift\_spec is not None and fastbinary is n

ot None:

oprot.trans.write(fastbinary.encode\_binary(self, (self.\_\_class\_\_, self.thrift\_spec)))

return

oprot.writeStructBegin('execute\_cql3\_query\_args')

if self.query is not None:

oprot.writeFieldBegin('query', TType.STRING, 1)

oprot.writeString(self.query)

oprot.writeFieldEnd()

if self.compression is not None:

oprot.writeFieldBegin('compression', TType.I32, 2)

oprot.writeI32(self.compression)

oprot.writeFieldEnd()

if self.consistency is not None:

oprot.writeFieldBegin('consistency', TType.I32, 3)

oprot.writeI32(self.consistency)

oprot.writeFieldEnd()

oprot.writeFieldStop()

oprot.writeStructEnd()

# thrift 研究

## Hello.thrift

1. namespace java service.demo

service Hello{

string helloString(1:string para)

i32 helloInt(1:i32 para)

bool helloBoolean(1:bool para)

void helloVoid()

string helloNull()

}

1. thrift --gen java Hello.thrift

使用 Thrift 工具编译 Hello.thrift，就会生成相应的 Hello.java 文件。

该文件包含:

1. Hello.thrift 文件中描述的服务 Hello 的接口定义，即 Hello.Iface 接口;
2. 服务调用的底层通信细节;
3. 客户端的调用逻辑 Hello.Client ;
4. 服务器端的处理逻辑 Hello.Processor.
5. HelloServiceImpl.java

创建服务器端实现代码，将 HelloServiceImpl 作为具体的处理器传递给 Thrift 服务器，代码如下:

import org.apache.thrift.TException;

public class HelloServiceImpl implements Hello.Iface {

@Override

public boolean helloBoolean(boolean para) throws TException {

return para;

}

@Override

public int helloInt(int para) throws TException {

try {

Thread.sleep(20000);

} catch (InterruptedException e) {

e.printStackTrace();

}

return para;

}

@Override

public String helloNull() throws TException {

return null;

}

@Override

public String helloString(String para) throws TException {

return para;

}

@Override

public void helloVoid() throws TException {

System.out.println("Hello World");

}

}

1. HelloServiceServer.java

创建服务器端实现代码，将 HelloServiceImpl 作为具体的处理器传递给 Thrift 服务器，代码如下：package service.server;

import org.apache.thrift.TProcessor;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TBinaryProtocol.Factory;

import org.apache.thrift.server.TServer;

import org.apache.thrift.server.TThreadPoolServer;

import org.apache.thrift.transport.TServerSocket;

import org.apache.thrift.transport.TTransportException;

import service.demo.Hello;

import service.demo.HelloServiceImpl;

public class HelloServiceServer {

/\*\*

\* 启动 Thrift 服务器

\* @param args

\*/

public static void main(String[] args) {

try {

// 设置服务端口为 7911

TServerSocket serverTransport = new TServerSocket(7911);

// 设置协议工厂为 TBinaryProtocol.Factory

Factory proFactory = new TBinaryProtocol.Factory();

// 关联处理器与 Hello 服务的实现

TProcessor processor = new Hello.Processor(new HelloServiceImpl());

TServer server = new TThreadPoolServer(processor, serverTransport,

proFactory);

System.out.println("Start server on port 7911...");

server.serve();

} catch (TTransportException e) {

e.printStackTrace();

}

}

}

1. HelloServiceClient.java

package service.client;

import org.apache.thrift.TException;

import org.apache.thrift.protocol.TBinaryProtocol;

import org.apache.thrift.protocol.TProtocol;

import org.apache.thrift.transport.TSocket;

import org.apache.thrift.transport.TTransport;

import org.apache.thrift.transport.TTransportException;

import service.demo.Hello;

public class HelloServiceClient {

/\*\*

\* 调用 Hello 服务

\* @param args

\*/

public static void main(String[] args) {

try {

// 设置调用的服务地址为本地，端口为 7911

TTransport transport = new TSocket("localhost", 7911);

transport.open();

// 设置传输协议为 TBinaryProtocol

TProtocol protocol = new TBinaryProtocol(transport);

Hello.Client client = new Hello.Client(protocol);

// 调用服务的 helloVoid 方法

client.helloVoid();

transport.close();

} catch (TTransportException e) {

e.printStackTrace();

} catch (TException e) {

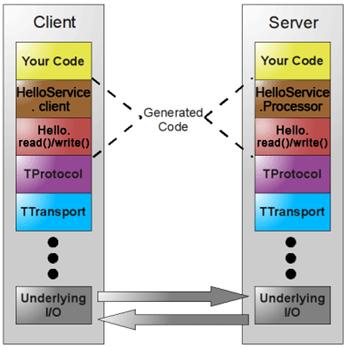
e.printStackTrace();

}

}

}

## 架构图



Thrift 服务器包含用于绑定协议和传输层的基础架构，它提供阻塞、非阻塞、单线程和多线程的模式运行在服务器上，可以配合服务器 / 容器一起运行，可以和现有的 J2EE 服务器 /Web 容器无缝的结合。

## Server 端启动、服务时序图



## Client 端调用服务时序图



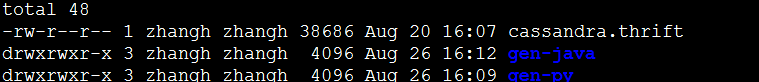
# cassandra thrift

定义的文件为：interface/ cassandra.thrift ，使用

thrift --gen py cassandra.thrift

thrift --gen Java cassandra.thrift

生成了相关代码如下：



Java 端作为服务器端代码，Python端作为客户端代码。实现了thrift接口实现过程。现在需要查看服务端的读数据请求过程，通过上文知道，py端最后发送的请求如下：

self.send\_execute\_cql3\_query(query, compression, consistency)

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

self.\_oprot.writeMessageBegin('execute\_cql3\_query', TMessageType.CALL, self.\_seqid)

现在需要查看execute\_cql3\_query在cassandra-server的实现过程,具体过程如下：

## execute\_cql3\_query

1. \src\java\org\apache\cassandra\thrift\CassandraServer.java 相当于上文中的 HelloServiceImpl.java，请求在该处具体的实现过程：

public CqlResult execute\_cql3\_query(ByteBuffer query, Compression compression, ConsistencyLevel cLevel)

{

validateCQLVersion(3);

try

{

String queryString = uncompress(query, compression); //解压了查询语句

if (startSessionIfRequested())

{

Tracing.instance.begin("execute\_cql3\_query",

ImmutableMap.of("query", queryString)); //重新建立了连接

}

else

{

logger.debug("execute\_cql3\_query");

}

ThriftClientState cState = state();

return cState.getCQLQueryHandler().process(queryString,

cState.getQueryState(),

new QueryOptions(ThriftConversion.fromThrift(cLevel),

Collections.<ByteBuffer>emptyList())).toThriftResult();

// process实现了处理过程，

}

catch (RequestExecutionException e)

{

throw ThriftConversion.rethrow(e);

}

catch (RequestValidationException e)

{

throw ThriftConversion.toThrift(e);

}

finally

{

Tracing.instance.stopSession();

}

}

## QueryProcessor.java

\src\java\org\apache\cassandra\cql3\QueryProcessor.java 实现了处理过程

public ResultMessage process(String queryString, QueryState queryState, QueryOptions options)

throws RequestExecutionException, RequestValidationException

{

CQLStatement prepared = getStatement(queryString, queryState.getClientState()).statement; //将查询语句转换为statement

if (prepared.getBoundTerms() != options.getValues().size())

throw new InvalidRequestException("Invalid amount of bind variables");

return processStatement(prepared, queryState, options); //此处处理

}

public static ResultMessage processStatement(CQLStatement statement, QueryState queryState,QueryOptions options)

throws RequestExecutionException, RequestValidationException

{

logger.trace("Process {} @CL.{}", statement, options.getConsistency());

ClientState clientState = queryState.getClientState();　　//获得执行状态

statement.checkAccess(clientState); //检查权限

statement.validate(clientState); //

ResultMessage result = statement.execute(queryState, options); #提交执行语句

return result == null ? new ResultMessage.Void() : result; //返回执行结果

}

## SelectStatement.java

src\java\org\apache\cassandra\cql3\statements\SelectStatement.java

public ResultMessage.Rows execute(QueryState state, QueryOptions options) throws RequestExecutionException, RequestValidationException

{

QueryPager pager = QueryPagers.pager(command, cl, options.getPagingState());

if (parameters.isCount)

return pageCountQuery(pager, variables, pageSize, now, limit); //count(\*) 处理count\*

List<Row> page = pager.fetchPage(pageSize);

ResultMessage.Rows msg = processResults(page, variables, limit, now); //一般的执行结果

if (!pager.isExhausted())

msg.result.metadata.setHasMorePages(pager.state());

return msg;

}

## SelectStatement.java

public static List<Row> read(List<ReadCommand> commands, ConsistencyLevel consistency\_level)//读数据流程