本人博客http://wqtn22.iteye.com/

本人微博http://weibo.com/u/2343837630?wvr=5&

转载请注明出处

目录

[1. 概述 3](#_Toc350168731)

[2. mysqld节点ndb引擎插件启动流程 4](#_Toc350168732)

[3. mysqld节点上对ndb主要对象的初始化 8](#_Toc350168733)

[1. Ndb\_cluster\_connection对象的初始化 8](#_Toc350168734)

[2. Ndb对象的初始化 9](#_Toc350168735)

[3. Ndb\_cluster\_connection对象连接 11](#_Toc350168736)

[4. mysqld节点上的主要工作线程 13](#_Toc350168737)

[1. ndb\_binlog线程 15](#_Toc350168738)

[2. util线程 17](#_Toc350168739)

[3. index线程 17](#_Toc350168740)

[5. mgmd节点启动流程 17](#_Toc350168741)

[6. mgmd节点工作流程 19](#_Toc350168742)

[1. 客户端请求到达过程 19](#_Toc350168743)

[2. 配置检查过程 20](#_Toc350168744)

[3. 配置变更过程 20](#_Toc350168745)

[7. ndbd节点启动流程 21](#_Toc350168746)

[8. ndbd节点消息处理虚拟机工作流程 23](#_Toc350168747)

[9. ndbd节点API接口访问流程 27](#_Toc350168748)

[1. 客户端API节点->服务器ndbd节点消息交互 28](#_Toc350168749)

[2. 服务器ndbd节点事务描述符申请 32](#_Toc350168750)

[3. 服务器ndbd节点事务处理全局事务DBTC部分 33](#_Toc350168751)

[4. 服务器ndbd节点事务处理数据分片管理DBDIH部分 39](#_Toc350168752)

[5. 服务器ndbd节点事务处理本地事务DBLQH部分 41](#_Toc350168753)

[6. 服务器ndbd节点事务处理本地索引查询及锁申请DBACC部分 46](#_Toc350168754)

[7. 服务器ndbd节点事务处理本地记录访问DBTUP部分 48](#_Toc350168755)

[8. 服务器ndbd节点事务处理对gci的验证DBDIH部分 58](#_Toc350168756)

[9. 服务器ndbd节点事务处理提交DBTC部分 59](#_Toc350168757)

[10. 服务器ndbd节点事务本地事务提交DBLQH部分 61](#_Toc350168758)

[11. 服务器ndbd节点事务本地事务完成DBLQH部分 70](#_Toc350168759)

[12. 服务器ndbd节点事务处理中止DBLQH部分 71](#_Toc350168760)

[10. ndbd节点各BLOCK启动流程 75](#_Toc350168761)

[1. 初始化阶段 76](#_Toc350168762)

[2. 配置读取阶段 79](#_Toc350168763)

[3. STTOR Phase 0 82](#_Toc350168764)

[4. STTOR Phase 1 82](#_Toc350168765)

[5. STTOR Phase 2 89](#_Toc350168766)

[6. NDB\_STTOR Phase 1 91](#_Toc350168767)

[7. STTOR Phase 3 92](#_Toc350168768)

[8. NDB\_STTOR Phase 2 93](#_Toc350168769)

[9. STTOR Phase 4 95](#_Toc350168770)

[10. NDB\_STTOR Phase 3 97](#_Toc350168771)

[11. STTOR Phase 5 105](#_Toc350168772)

[12. NDB\_STTOR Phase 4 105](#_Toc350168773)

[13. NDB\_STTOR Phase 5 111](#_Toc350168774)

[14. NDB\_STTOR Phase 6 118](#_Toc350168775)

[15. STTOR Phase 6 121](#_Toc350168776)

[16. STTOR Phase 7 122](#_Toc350168777)

[17. STTOR Phase 8 124](#_Toc350168778)

[18. NDB\_STTOR Phase 7 125](#_Toc350168779)

[19. STTOR Phase 9 125](#_Toc350168780)

[20. STTOR Phase 101 126](#_Toc350168781)

[11. 其它 126](#_Toc350168782)

# 概述

mysql ndb存储引擎由以下三类节点组成：

mysqld（API）节点：查询节点，代理用户到ndbd数据节点的访问，将用户sql转换为对ndbd API的调用，并将结果组成mysql协议返回给用户；

mgmd节点：配置管理节点，用于配置服务、节点id分配、状态收集等；

ndbd节点：数据存储节点，用于数据存储、事务协调、锁管理等。

各类节点运行期通过消息交互，ndb内部定义了完整的消息格式，以及各个消息的处理模块和处理回调，这些处理模块称为BLOCK，从而构建了一套反应式消息处理模型。ndbd节点内部有一个消息处理虚拟机，类似于erlang的消息队列+虚拟机+gen\_server。

mysqld（API）节点/mgmd节点与ndbd节点通信时，BLOCK为-1，表明其客户端的身份。

ndbd节点有如下BLOCK：

BACKUP:负责处理在线备份和检查点。

CMVMI:负责处理ndbd的内核BLOCK、虚拟机、集群任务队列、集群Transporter的配置管理。

DBACC:负责访问控制与锁管理，同时处理数据库索引结构，该结构存储在8k的页中。当新元组插入时，DBTUP负责将元组存储在合适的位置，并返回一个内存索引（元组在内存中的位置引用），DBACC将该元组的主键和内存索引存储在一个哈希表中，与DBTUP类似，DBACC实现了一部分的检查点协议，也实现了undo日志。

DBDICT:负责管理元数据（通过集群master节点），如表和索引定义，一些磁盘数据操作等。数据字典信息将被复制到集群所有ndbd节点的DBDICT中，DBDICT和DBTC是仅有的应用程序可以直接发送请求的BLOCK。

DBDIH:负责提供数据分布（分区）管理服务，负责管理数据分片和副本，处理local和global检查点，也负责管理节点和系统重启。

DBSPJ:NDB7.2加入，实现了NDB内核的多重游标技术，提供了将join下推到数据节点的功能。

DBLQH:负责提供本地节点低级查询句柄，管理本地节点的数据和事务，也作为[本地?]两阶段提交的coordinator。负责部署元组相关的操作，与DBACC（负责管理索引）和DBTUP协同工作（负责管理元组）。

DBTC:全局事务coordinator，在全局级别上处理分布式事务和其它数据操作，与DBLQH相反，DBLQH处理本地数据节点的事务和数据操作。

DBTUP:负责管理集群数据的物理存储，作为元组管理器，也监视元组的变动。

DBTUX:负责管理本地有序索引。

DBUTIL:负责提供一些内部事务和数据操作接口，实现了节点间消息传递的基本操作，也实现了一些公共事务化服务，供其他BLOCK使用。另外，还负责管理备份ID号及其他分布式标识符。

LGMAN:负责磁盘数据表的日志组管理，管理磁盘数据表的undo日志。

NDBCNTR:负责管理其他BLOCK初始化和配置，在ndbd启动过程中，接管QMGR的工作，也负责节点关闭的工作。

NDBFS:负责提供ndbd文件系统抽象层。

PGMAN:负责磁盘数据表的页和缓冲区管理服务。

QMGR:负责逻辑集群管理，通过心跳关键理集群节点成员。当节点失效发生时，QMGR将不断轮训其它数据节点以确定节点真实失效，并将其从集群中剔除。

RESTORE:负责将集群从检查点中恢复。

SUMA:负责处理事件日志和报告，是集群的订阅管理器，同时也是MySQL Cluster集群间复制功能的重要组成部分。

TSMAN:负责磁盘数据表的表空间管理。

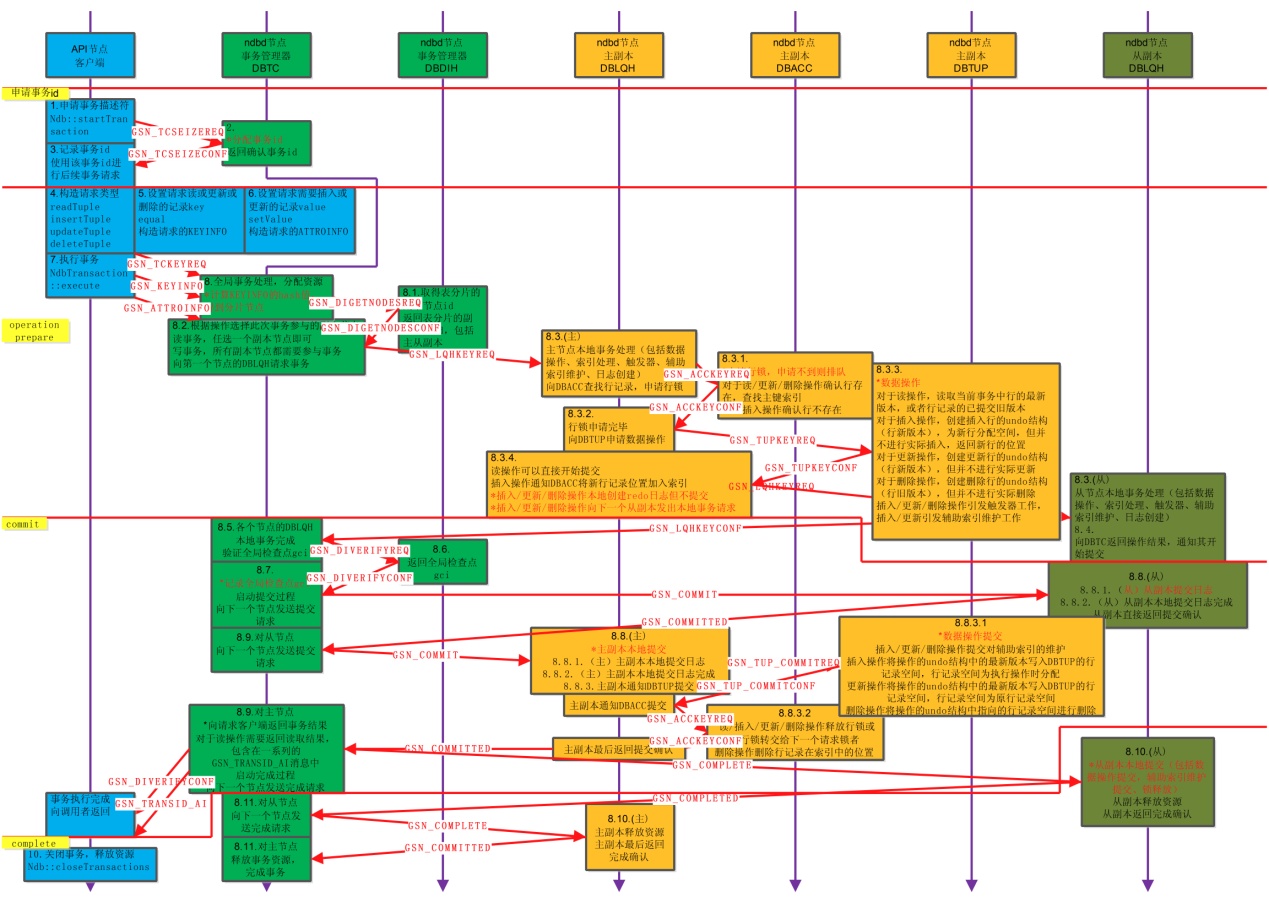
TRIX:负责管理内部触发器和唯一索引，作为功能BLOCK与DBUTIL类似，包含了许多帮助函数用于创建索引及处理节点间消息。

ndb协议交互模式可参考如下链接：

http://dev.mysql.com/doc/ndbapi/en/ndb-internals-ndb-protocol.html

ndb API接口事务处理过程简图（详细过程见第九节）：

紫色垂直线代表时间轴，红色水平线代表阶段，同色流程代表处于同一节点上的BLOCK



# mysqld节点ndb引擎插件启动流程

mysqld调用ndb引擎的启动入口为：

static int ndbcluster\_init(void \*p)

{

//初始化util线程的若干互斥量与条件变量

ndb\_util\_thread.init();

//初始化index线程的若干互斥量与条件变量

ndb\_index\_stat\_thread.init();

//初始化handlerton的各项回调

{

handlerton \*h= ndbcluster\_hton;

h->state= SHOW\_OPTION\_YES;

h->db\_type= DB\_TYPE\_NDBCLUSTER;

h->close\_connection= ndbcluster\_close\_connection;

h->commit= ndbcluster\_commit;

h->rollback= ndbcluster\_rollback;

h->create= ndbcluster\_create\_handler; /\* Create a new handler \*/

h->drop\_database= ndbcluster\_drop\_database; /\* Drop a database \*/

h->panic= ndbcluster\_end; /\* Panic call \*/

h->show\_status= ndbcluster\_show\_status; /\* Show status \*/

h->alter\_tablespace= ndbcluster\_alter\_tablespace; /\* Show status \*/

h->partition\_flags= ndbcluster\_partition\_flags; /\* Partition flags \*/

h->alter\_table\_flags= ndbcluster\_alter\_table\_flags; /\* Alter table flags \*/

h->fill\_is\_table= ndbcluster\_fill\_is\_table;

ndbcluster\_binlog\_init\_handlerton();

h->flags= HTON\_CAN\_RECREATE | HTON\_TEMPORARY\_NOT\_SUPPORTED;

h->discover= ndbcluster\_discover;

h->find\_files= ndbcluster\_find\_files;

h->table\_exists\_in\_engine= ndbcluster\_table\_exists\_in\_engine;

h->make\_pushed\_join= ndbcluster\_make\_pushed\_join;

h->is\_supported\_system\_table= ndb\_is\_supported\_system\_table;

}

//初始化输出和错误流、一些全局互斥量和条件变量、线程本地变量

ndb\_init\_internal();

//分配连接资源并连接到集群，核心初始化函数之一

ndbcluster\_connect(connect\_callback, opt\_ndb\_wait\_connected,

opt\_ndb\_cluster\_connection\_pool,

(global\_opti\_node\_select & 1),

opt\_ndb\_connectstring,

opt\_ndb\_nodeid)

//初始化记录已打开表和已删除表的哈希表

(void) my\_hash\_init(&ndbcluster\_open\_tables,table\_alias\_charset,32,0,0,

(my\_hash\_get\_key) ndbcluster\_get\_key,0,0);

(void) my\_hash\_init(&ndbcluster\_dropped\_tables,table\_alias\_charset,32,0,0,

(my\_hash\_get\_key) ndbcluster\_get\_key,0,0);

//创建ndb\_binlog监听线程，核心初始化函数之一，binlog监听线程监听来自ndbd的更新事件，并录制到本地binlog中

ndbcluster\_binlog\_start()

//启动util线程，核心初始化函数之一，util线程用于建库、建表、获取表统计信息等

ndb\_util\_thread.start()

//启动index线程，核心初始化函数之一，index线程用于将表索引加入cache、定期检查索引更新、删除等

ndb\_index\_stat\_thread.start()

//初始化mysql slave统计量，这些统计量用于跟踪slave行为，在事务提交/回滚、元数据操作等处更新

memset(&g\_slave\_api\_client\_stats, 0, sizeof(g\_slave\_api\_client\_stats));

}

int

ndbcluster\_connect(int (\*connect\_callback)(void), ulong wait\_connected, uint connection\_pool\_size, bool optimized\_node\_select, const char\* connect\_string, uint force\_nodeid)

{

//创建一个全局的Ndb\_cluster\_connection对象g\_ndb\_cluster\_connection，这是ndb的api接口的连接管理对象，一个ndb应用程序应该始于创建一个Ndb\_cluster\_connection对象，通过该对象的connect函数连接到集群mgmd，通过该对象的wait\_until\_ready函数等待连接到达一到多个数据节点。多个Ndb对象可以创建自同一个Ndb\_cluster\_connection对象，以支持多线程，一个Ndb\_cluster\_connection对象仅可绑定到一个mgmd节点，一个mgmd节点仅可连接到一个ndbd集群，一个应用程序可以建立多个Ndb\_cluster\_connection对象，以访问多个ndbd集群，实现应用层数据分区。

g\_ndb\_cluster\_connection=new Ndb\_cluster\_connection(connect\_string, force\_nodeid);

//创建一个全局的Ndb对象g\_ndb，这是ndb的api接口的数据访问对象。任何ndb api程序至少需要使用一个Ndb对象的实例，用于进入库表元数据访问、启动事务、监听事件、进入ndb对象（表空间、undo日志空间等）访问等。通过使用几个Ndb对象，可以实现多线程的应用程序，Ndb对象不可以被多个线程共享，但是一个线程可以使用多个Ndb对象，最多4711个。

g\_ndb= new Ndb(g\_ndb\_cluster\_connection, "sys")

//初始化Ndb对象

g\_ndb->init()

//通过Ndb\_cluster\_connection对象连接到ndb集群的管理节点，分配及确认本节点的节点id、配置参数等，连接过程将不断重试，直至超时或成功

while ((res= g\_ndb\_cluster\_connection->connect(0,0,0)) == 1)

{

if (NdbTick\_CurrentMillisecond() > end\_time)

break;

do\_retry\_sleep(100);

if (abort\_loop)

DBUG\_RETURN(-1);

}

//额外创建connection\_pool\_size个Ndb\_cluster\_connection对象，默认值为0

{

g\_pool\_alloc= connection\_pool\_size;

g\_pool= (Ndb\_cluster\_connection\*\*) my\_malloc(g\_pool\_alloc \* sizeof(Ndb\_cluster\_connection\*), MYF(MY\_WME | MY\_ZEROFILL));

for (uint i= 1; i < g\_pool\_alloc; i++)

{

g\_pool[i]= new Ndb\_cluster\_connection(connect\_string, g\_ndb\_cluster\_connection)) == 0

g\_pool[i]->set\_name(buf);

}

}

//对连接池内的connection\_pool\_size个Ndb\_cluster\_connection对象同样启动连接过程，若连接失败，则通过Ndb\_cluster\_connection::start\_connect\_thread建立connection\_pool\_size个线程异步连接

if (res == 0)

{

connect\_callback();

for (uint i= 0; i < g\_pool\_alloc; i++)

{

int node\_id= g\_pool[i]->node\_id();

if (node\_id == 0)

{

g\_pool[i]->connect(0,0,0);

node\_id= g\_pool[i]->node\_id();

}

}

}

else if (res == 1)

{

//若此次未能成功连接，则启动一个异步连接线程进行连接

for (uint i= 0; i < g\_pool\_alloc; i++)

g\_pool[i]->start\_connect\_thread(i == 0 ? connect\_callback : NULL)

}

else

DBUG\_RETURN(-1);

DBUG\_RETURN(0);

}

int Ndb\_cluster\_connection::start\_connect\_thread(int (\*connect\_callback)(void))

{

m\_impl.m\_connect\_callback= connect\_callback;

if ((r = connect(0,0,0)) == 1)

{

//run\_ndb\_cluster\_connection\_connect\_thread连接mgmd，成功后通过connect\_callback回调通知调用者

m\_impl.m\_connect\_thread=

NdbThread\_Create(run\_ndb\_cluster\_connection\_connect\_thread,

(void\*\*)&m\_impl, 0, "ndb\_cluster\_connection",

NDB\_THREAD\_PRIO\_LOW);

}

else if (r < 0)

DBUG\_RETURN(-1);

else if (m\_impl.m\_connect\_callback)

(\*m\_impl.m\_connect\_callback)();

}

# mysqld节点上对ndb主要对象的初始化

## Ndb\_cluster\_connection对象的初始化

Ndb\_cluster\_connection::Ndb\_cluster\_connection(const char \*connect\_string)

: m\_impl(\* new Ndb\_cluster\_connection\_impl(connect\_string, 0, 0))

{

}

Ndb\_cluster\_connection\_impl::

Ndb\_cluster\_connection\_impl(const char \* connect\_string,

Ndb\_cluster\_connection \*main\_connection,

int force\_api\_nodeid)

{

//初始化各个类成员

//初始化各个互斥量

//初始化各个全局统计量

memset(globalApiStatsBaseline, 0, sizeof(globalApiStatsBaseline));

//创建config获取对象，该对象用于取得配置参数

m\_config\_retriever=

new ConfigRetriever(connect\_string, force\_api\_nodeid,

NDB\_VERSION, NDB\_MGM\_NODE\_TYPE\_API);

if (!m\_main\_connection)

{

//首次初始化经历此路径

//创建连接的数据字典，此处将初始化一个表元数据哈希表，并初始化两个成员用于记录已经失效的表和已经改动的表

m\_globalDictCache = new GlobalDictCache;

//创建连接管理的外观对象TransporterFacade

m\_transporter\_facade= new TransporterFacade(m\_globalDictCache);

}

else

{

assert(m\_main\_connection->m\_impl.m\_globalDictCache != NULL);

m\_globalDictCache = 0;

m\_transporter\_facade=

new TransporterFacade(m\_main\_connection->m\_impl.m\_globalDictCache);

// The secondary connection can't use same nodeid, clear the nodeid

// in ConfigRetriver to avoid asking for the same nodeid again

m\_config\_retriever->setNodeId(0);

}

}

ConfigRetriever::ConfigRetriever(const char \* \_connect\_string,

int force\_nodeid,

Uint32 version,

ndb\_mgm\_node\_type node\_type,

const char \* \_bindaddress,

int timeout\_ms) :

m\_end\_session(true),

m\_version(version),

m\_node\_type(node\_type)

{

//创建mgmd句柄NdbMgmHandle类型对象

m\_handle= ndb\_mgm\_create\_handle();

//设置连接超时

ndb\_mgm\_set\_timeout(m\_handle, timeout\_ms);

//根据命令行参数connect\_string进行实际的初始化工作，以得到mgmd节点的实际connect\_string，即mgmd节点的id、地址:端口，优先级如下：

//1.命令行参数给出的connect\_string

//2.命令行参数给出的配置文件

//3.环境变量NDB\_CONNECTSTRING

//4.环境变量NDB\_HOME指定的目录下的Ndb.cfg文件

//5.工作目录下的Ndb.cfg文件

//6.默认的connect\_string，本机地址

ndb\_mgm\_set\_connectstring(m\_handle, \_connect\_string)

resetError();

DBUG\_VOID\_RETURN;

}

## Ndb对象的初始化

Ndb::Ndb( Ndb\_cluster\_connection \*ndb\_cluster\_connection,

const char\* aDataBase , const char\* aSchema)

: theImpl(NULL)

{

setup(ndb\_cluster\_connection, aDataBase, aSchema);

DBUG\_VOID\_RETURN;

}

void Ndb::setup(Ndb\_cluster\_connection \*ndb\_cluster\_connection,

const char\* aDataBase , const char\* aSchema)

{

//初始化若干成员变量

theError.code = 0;

//初始化连接事务数组，大小为MAX\_NDB\_NODES（49），最多可同时支持MAX\_NDB\_NODES-1（）48个事务

theConnectionArray = new NdbConnection \* [MAX\_NDB\_NODES];

theConnectionArrayLast = new NdbConnection \* [MAX\_NDB\_NODES];

theCommitAckSignal = NULL;

theCachedMinDbNodeVersion = 0;

int i;

for (i = 0; i < MAX\_NDB\_NODES ; i++) {

theConnectionArray[i] = NULL;

theConnectionArrayLast[i] = NULL;

}//for

//设置库名、schema名

theImpl->m\_dbname.assign(aDataBase);

theImpl->m\_schemaname.assign(aSchema);

theImpl->update\_prefix();

//创建事件缓冲区

theEventBuffer= new NdbEventBuffer(this);

//将新建立的Ndb对象加入到其所属的Ndb\_cluster\_connection对象的列表中

theImpl->m\_ndb\_cluster\_connection.link\_ndb\_object(this);

DBUG\_VOID\_RETURN;

}

int Ndb::init(int aMaxNoOfTransactions)

{

//默认的aMaxNoOfTransactions为4，即允许并发创建的事务数，这个值可以进行配置

theInitState = StartingInit;

TransporterFacade \* theFacade = theImpl->m\_transporter\_facade;

theEventBuffer->m\_mutex = theFacade->theMutexPtr;

//构造api节点在ndb集群中的BLOCK引用号，这个引用号由节点id、BLOCK号、动态计算的引用标号，与ndbd节点交互，必须通过BLOCK引用号在各个BLOCK间交换消息，api节点的BLOCK号为-1

const Uint32 tRef = theImpl->open(theFacade);

Uint32 nodeId = refToNode(tRef);

theNdbBlockNumber = refToBlock(tRef);

//设置cache的主节点版本

theFacade->lock\_mutex();

theCachedMinDbNodeVersion = theFacade->getMinDbNodeVersion();

theFacade->unlock\_mutex();

//创建事务空闲列表，aNrOfCon默认为节点数

if((createConIdleList(aNrOfCon)) == -1){

theError.code = 4000;

goto error\_handler;

}

//创建操作空闲列表，aNrOfOp默认为节点数\*2

if((createOpIdleList(aNrOfOp)) == -1){

theError.code = 4000;

goto error\_handler;

}

//初始化事务管理列表。包括prepared、senttransaction、completed

thePreparedTransactionsArray = new NdbTransaction\* [tMaxNoOfTransactions];

theSentTransactionsArray = new NdbTransaction\* [tMaxNoOfTransactions];

theCompletedTransactionsArray = new NdbTransaction\* [tMaxNoOfTransactions];

for (i = 0; i < tMaxNoOfTransactions; i++) {

thePreparedTransactionsArray[i] = NULL;

theSentTransactionsArray[i] = NULL;

theCompletedTransactionsArray[i] = NULL;

}//for

//初始化空闲Signal列表

for (i = 0; i < 16; i++)

tSignal[i] = getSignal();

for (i = 0; i < 16; i++)

releaseSignal(tSignal[i]);

theInitState = Initialised;

}

## Ndb\_cluster\_connection对象连接

int Ndb\_cluster\_connection::connect(int no\_retries,

int retry\_delay\_in\_seconds,

int verbose)

{

return m\_impl.connect(no\_retries, retry\_delay\_in\_seconds, verbose);

}

int Ndb\_cluster\_connection\_impl::connect(int no\_retries,

int retry\_delay\_in\_seconds,

int verbose)

{

//从mgmd节点获取本API节点的id

Uint32 nodeId = m\_config\_retriever->allocNodeId(no\_retries, retry\_delay\_in\_seconds, verbose, alloc\_error);

//从mgmd节点获取本API节点的配置参数

ndb\_mgm\_configuration \* props = m\_config\_retriever->getConfig(nodeId);

//根据配置参数填写API节点的配置变量

configure(nodeId, \*props)

//启动连接管理相关的线程

m\_transporter\_facade->start\_instance(nodeId, props)

//连接成功

m\_transporter\_facade->connected();

}

Uint32 ConfigRetriever::allocNodeId(int no\_retries, int retry\_delay\_in\_seconds, int verbose, int& error)

{

//连接mgmd节点

ndb\_mgm\_connect(m\_handle, 0, 0, verbose)

//向mgmd节点发出”get nodeid”请求，同步等待结果返回或超时

ndb\_mgm\_alloc\_nodeid(m\_handle, m\_version, m\_node\_type, no\_retries == 0);

}

struct ndb\_mgm\_configuration\* ConfigRetriever::getConfig(Uint32 nodeid)

{

//向mgmd节点发出”get config”请求，同步等待结果返回或超时，结果包括system、node、connection配置项

getConfig(m\_handle)

//验证本节点参数是否与本节点实际情况相符，包括本节点主机名、节点类型、远程mgmd节点主机名等

verifyConfig(p, nodeid)

}

int Ndb\_cluster\_connection\_impl::configure(Uint32 nodeId, const ndb\_mgm\_configuration &config)

{

//配置mysqld节点ndb引擎的各项参数，如批量扫描数、批量字节数、批量事务数等、事务超时间隔等

}

int TransporterFacade::start\_instance(NodeId nodeId, const ndb\_mgm\_configuration\* conf)

{

theOwnId = nodeId;

//构造一个Transporter注册结构，它将连接抽象为Transporter，连接可以通过UNIX套接字、TCP套接字、共享内存等方式实现，

theTransporterRegistry = new TransporterRegistry(this, this);

//初始化Transporter注册结构，对于TCP套接字，初始化时将创建一个epollfd，用于轮训各个tcp连接

theTransporterRegistry->init(nodeId)

//初始化集群管理代理结构

theClusterMgr = new ClusterMgr(\*this)

//配置TransporterFacade本身的各项参数，包括创建一个新的NdbMgmHandle句柄，创建一个到本地的环回连接，遍历配置信息中的所有节点的配置，得到全局节点的一致性视图，配置节点的发送缓冲，自动重连、线程优先级等

configure(nodeId, conf)

//启动Transporter管理服务，由于管理各个新建的Transporter

theTransporterRegistry->start\_service(m\_socket\_server)

//启动连接状态检查线程

theReceiveThread = NdbThread\_Create(runReceiveResponse\_C, (void\*\*)this, 0, "ndb\_receive", NDB\_THREAD\_PRIO\_LOW);

//启动数据发送线程，该线程会启动三个线程：

//1.异步数据发送线程，将数据异步发送出去，但这个发送与mysqld的数据访问请求无关，因为mysqld的数据访问是同步过程

//2.连接接收线程，accept到达的连接，并注册入theTransporterRegistry

//3.连接mgmd，获取配置参数

theSendThread = NdbThread\_Create(runSendRequest\_C, (void\*\*)this, 0, "ndb\_send", NDB\_THREAD\_PRIO\_LOW);

//启动本节点的集群监视线程

theClusterMgr->startThread();

}

# mysqld节点上的主要工作线程

初始化过程对ndb\_binlog线程的建立：

int ndbcluster\_binlog\_start()

{

//进行一些初始化检查

//初始化若干互斥量和条件变量

pthread\_mutex\_init(&injector\_mutex, MY\_MUTEX\_INIT\_FAST);

pthread\_cond\_init(&injector\_cond, NULL);

pthread\_mutex\_init(&ndb\_schema\_share\_mutex, MY\_MUTEX\_INIT\_FAST);

//创建binlog监听线程ndb\_binlog，线程入口函数为ndb\_binlog\_thread\_func

if (pthread\_create(&ndb\_binlog\_thread, &connection\_attrib,

ndb\_binlog\_thread\_func, 0))

{

DBUG\_PRINT("error", ("Could not create ndb injector thread"));

pthread\_cond\_destroy(&injector\_cond);

pthread\_mutex\_destroy(&injector\_mutex);

DBUG\_RETURN(-1);

}

ndbcluster\_binlog\_inited= 1;

pthread\_mutex\_lock(&injector\_mutex);

while (!ndb\_binlog\_thread\_running)

pthread\_cond\_wait(&injector\_cond, &injector\_mutex);

pthread\_mutex\_unlock(&injector\_mutex);

if (ndb\_binlog\_thread\_running < 0)

DBUG\_RETURN(-1);

DBUG\_RETURN(0);

}

初始化过程对util和index线程的建立：

Ndb\_util\_thread ndb\_util\_thread;

Ndb\_index\_stat\_thread ndb\_index\_stat\_thread;

Ndb\_util\_thread与Ndb\_index\_stat\_thread继承自Ndb\_component，其start函数指向Ndb\_component::start

int Ndb\_component::start()

{

assert(m\_thread\_state == TS\_INIT);

pthread\_mutex\_lock(&m\_start\_stop\_mutex);

m\_thread\_state= TS\_STARTING;

//创建一个新的线程，线程入口为Ndb\_component\_run\_C

int res= pthread\_create(&m\_thread, &connection\_attrib, Ndb\_component\_run\_C, this);

while (m\_thread\_state == TS\_STARTING)

pthread\_cond\_wait(&m\_start\_stop\_cond, &m\_start\_stop\_mutex);

pthread\_mutex\_unlock(&m\_start\_stop\_mutex);

return m\_thread\_state == TS\_RUNNING ? 0 : 1;

}

void \*Ndb\_component\_run\_C(void \* arg)

{

my\_thread\_init();

Ndb\_component \* self = reinterpret\_cast<Ndb\_component\*>(arg);

//调用Ndb\_component::run\_impl函数

self->run\_impl();

my\_thread\_end();

pthread\_exit(0);

return NULL; // Avoid compiler warnings

}

void Ndb\_component::run\_impl()

{

pthread\_detach\_this\_thread();

pthread\_mutex\_lock(&m\_start\_stop\_mutex);

if (m\_thread\_state == TS\_STARTING)

{

m\_thread\_state= TS\_RUNNING;

pthread\_cond\_signal(&m\_start\_stop\_cond);

pthread\_mutex\_unlock(&m\_start\_stop\_mutex);

//最终调用各个类的do\_run函数

do\_run();

pthread\_mutex\_lock(&m\_start\_stop\_mutex);

}

m\_thread\_state = TS\_STOPPED;

pthread\_cond\_signal(&m\_start\_stop\_cond);

pthread\_mutex\_unlock(&m\_start\_stop\_mutex);

}

## ndb\_binlog线程

pthread\_handler\_t ndb\_binlog\_thread\_func(void \*arg)

{

my\_thread\_init();

//构造一个伪THD作为复制事件监听与binlog构造线程的描述符

thd= new THD;

//进行THD基本的初始化

thd->store\_globals()

lex\_start(thd);

thd->init\_for\_queries();

thd\_set\_command(thd, COM\_DAEMON);

//构造一个到NDB\_REP\_DB（mysql）库（即元数据）的Ndb对象和一个到空库的Ndb对象

s\_ndb= new Ndb(g\_ndb\_cluster\_connection, NDB\_REP\_DB)

i\_ndb= new Ndb(g\_ndb\_cluster\_connection, "")

//通过Ndb::pollEvents等待ndbd的第一个事件

schema\_res= s\_ndb->pollEvents(100, &schema\_gci)

res= i\_ndb->pollEvents(10, &gci);

i\_ndb->flushIncompleteEvents(schema\_gci);

s\_ndb->flushIncompleteEvents(schema\_gci);

//开始真正的事件接收过程

for ( ; !((ndbcluster\_binlog\_terminating ||

binlog\_thread\_state) &&

ndb\_latest\_handled\_binlog\_epoch >= ndb\_get\_latest\_trans\_gci()) &&

binlog\_thread\_state != BCCC\_restart; )

{

res= i\_ndb->pollEvents(tot\_poll\_wait, &gci);

int schema\_res= s\_ndb->pollEvents(tot\_poll\_wait, &schema\_gci);

while (gci > schema\_gci && schema\_res >= 0)

schema\_res= s\_ndb->pollEvents(10, &schema\_gci);

init\_sql\_alloc(&mem\_root, 4096, 0);

//构造一个schema事件句柄

Ndb\_schema\_event\_handler schema\_event\_handler(thd, &mem\_root, g\_ndb\_cluster\_connection->node\_id());

while (pOp != NULL)

{

if (!pOp->hasError())

{

//处理收到的schema事件，将这些事件转换为binlog，对于建表、删表操作等，还需要在本地mysqld执行一遍这些操作

schema\_event\_handler.handle\_event(s\_ndb, pOp);

if (i\_ndb->getEventOperation() == NULL &&

s\_ndb->getEventOperation() == NULL &&

binlog\_thread\_state == BCCC\_running)

binlog\_thread\_state= BCCC\_restart;

}

pOp= s\_ndb->nextEvent();

}

//更新事件监听的统计

updateInjectorStats(s\_ndb, i\_ndb);

//取得数据事件，

NdbEventOperation \*pOp= i\_ndb->nextEvent();

while (pOp != NULL)

{

gci= pOp->getGCI();

inj->new\_trans(thd, &trans);

while ((gci\_op= i\_ndb->getGCIEventOperations(&iter, &event\_types)) != NULL)

{

Ndb\_event\_data \*event\_data= (Ndb\_event\_data \*) gci\_op->getCustomData();

}

//更新mysql.ndb\_apply\_status表的gci位置，这个gci位置用于一个mysql slave从多个mysql master（slave与master均使用ndb引擎）复制binlog时检查一致性时使用

injectApplyStatusWriteRow(trans, gci)

do

{

//处理数据事件，将数据变更事件转换为binlog

if ((unsigned) pOp->getEventType() <

(unsigned) NDBEVENT::TE\_FIRST\_NON\_DATA\_EVENT)

handle\_data\_event(thd, i\_ndb, pOp, &rows, trans,

trans\_row\_count, trans\_slave\_row\_count);

//处理非数据事件

else

handle\_non\_data\_event(thd, pOp, \*rows);

pOp= i\_ndb->nextEvent();

} while (pOp && pOp->getGCI() == gci);

updateInjectorStats(s\_ndb, i\_ndb);

while (trans.good())

{

//提交binlog

trans.commit()

//更新binlog写入位置，记录入mysql.ndb\_binlog\_index表

ndb\_binlog\_index\_table\_\_write\_rows(thd, rows);

}

}

}

}

## util线程

void Ndb\_util\_thread::do\_run()

{

//启动时恢复ndb系统表ndb\_schema，此表不显示在mysql库中，但可以被show create table

//启动时检查其它节点对库表的变动，自动建库、建表（经实测自动建库不会发生）

//建立表事件监听过程

//获取表统计信息

}

## index线程

void Ndb\_index\_stat\_thread::do\_run()

{

//启动时恢复ndb系统表ndb\_index\_stat\_head和ndb\_index\_stat\_sample

//建立对ndb\_index\_stat\_head表的事件监听过程

//将表索引加入cache

//定期检查对索引的更新和删除

}

# mgmd节点启动流程

static int mgmd\_main(int argc, char\*\* argv)

{

//读配置文件，处理命令行选项

ndb\_load\_defaults(NULL, load\_default\_groups,&argc,&argv);

handle\_options(&argc, &argv, my\_long\_options, ndb\_std\_get\_one\_option)

//建立事件日志

g\_eventLogger->setCategory(opt\_logname);

g\_eventLogger->createConsoleHandler();

while (!g\_StopServer)

{

//初始化mgmd提供配置服务的对象MgmtSrvr，包括初始化两个用于记录连入节点类型和地址的数组nodeTypes和m\_connect\_address，在事件监听者Event\_listener m\_event\_listener中加入对cluster\_log监听事件等

mgm= new MgmtSrvr(opts);

//建立mgmd节点的配置管理器对象ConfigManager，设置mgmd节点的节点id，这个节点id应由命令行指派

//检查配置缓存文件，初次初始化时，该文件应不存在，因此从设置的配置文件中生成或从其它mgmd节点处取得，然后保存当前的所有配置到一个配置缓存文件；重启时，若这个缓存文件先前存在，则需要从该文件中加载配置项。因此在重新配置集群时，需要先删除各个mgmd节点的配置缓存文件才可以令新配置生效

//与其他mgmd节点交换配置信息，并进行比较，若配置不同则拒绝启动

mgm->init()

//启动mgmd节点的配置管理服务

mgm->start()

//主线程睡眠

while (!g\_StopServer)

NdbSleep\_MilliSleep(500);

}

}

bool MgmtSrvr::start()

{

//过程与mysqld初始化ndb引擎时类似，创建一个TransporterFacade对象，调用其start\_instance启动若干工作线程

start\_transporter(m\_local\_config)

//初始化配置管理服务，附着在连接接收线程的工作流程中

start\_mgm\_service(m\_local\_config)

//创建一个到本节点的连接

connect\_to\_self()

//为ConfigManger单独启动一个线程，进行配置管理，入口点为ConfigManager::run，该线程主要用于处理与其他mgmd节点的交互，管理配置变更

m\_config\_manager->start()

//启动日志级别管理线程，线程入口为logLevelThread\_C

\_logLevelThread = NdbThread\_Create(logLevelThread\_C, (void\*\*)this, 0, "MgmtSrvr\_Loglevel", NDB\_THREAD\_PRIO\_LOW);

}

bool MgmtSrvr::start\_transporter(const Config\* config)

{

//创建一个TransporterFacade对象

theFacade= new TransporterFacade(0);

//启动若干工作线程，包括：

//连接状态检查线程，异步数据发送线程，连接接收线程，连接mgmd的线程，集群监视线程等

theFacade->start\_instance(\_ownNodeId, config->m\_configValues)

//设置本节点的块引用号

\_ownReference = numberToRef(\_blockNumber, \_ownNodeId);

}

bool MgmtSrvr::start\_mgm\_service(const Config\* config)

{

//初始化mgmd配置服务管理对象

MgmApiService \* mapi = new MgmApiService(\*this);

//将集群配置服务MgmApiService绑定到一个连接接收线程的描述符上，与TransporterFacade::start\_instance创建的连接接收线程不同

//TransporterFacade::start\_instance启动的服务是TransporterService，新到达的会话附着在SocketServer所处的线程中处理，仅用于各个节点间交换一些基础信息，包括节点id、transporter类型，以保持每个节点形成的全局团网视图；此处是MgmApiService，新到达的会话将新创建一个单独的线程处理，用以提供集群配置服务

m\_socket\_server.setup(mapi, &port, m\_opts.bind\_address)

//启动连接接收线程，线程新建会话为MgmApiSession类型，每到达一个新连接，便创建一个新线程为其服务

m\_socket\_server.startServer();

}

# mgmd节点工作流程

## 客户端请求到达过程

客户端请求到达mgmd节点后，由mgmd节点的内部SocketServer对象m\_config\_manager接收连接请求，调用MgmApiService::NewSession生成一个MgmApiSession对象作为此次交互的描述符，并单独创建一个线程，线程最终调用MgmApiSession::runSession()进入服务循环。

MgmApiSession::MgmApiSession(class MgmtSrvr & mgm, NDB\_SOCKET\_TYPE sock, Uint64 session\_id)

: SocketServer::Session(sock), m\_mgmsrv(mgm),

m\_session\_id(session\_id), m\_name("unknown:0")

{

m\_input = new SocketInputStream(sock, SOCKET\_TIMEOUT);

m\_output = new BufferedSockOutputStream(sock, SOCKET\_TIMEOUT);

//此处为MgmApiSession初始化命令parser，该parser实质是一个流式kv提取器，提取指定key（命令名）、value（参数），调用相关处理回调，此处通过一个命令:回调映射数组实现命令匹配及自动处理，数组为const ParserRow<MgmApiSession> commands[]，mysqld节点启动时由，经历调用Ndb\_cluster\_connection\_impl::connect获取其启动信息时，需要调用ndb\_mgm\_alloc\_nodeid/ndb\_mgm\_get\_configuration2从mgmd处取得节点id/节点配置，工作原理即如此，命令key为”get nodeid”/”get config”，在mgmd的处理回调为MgmApiSession::get\_nodeid/MgmApiSession::getConfig

m\_parser = new Parser\_t(commands, \*m\_input, true, true, true);

}

void MgmApiSession::runSession()

{

while(!stop) {

NdbMutex\_Lock(m\_mutex);

//此处是parser自动工作的位置，将匹配到的命令的回调提取出来，调用回调得到结果，返回给客户端

m\_parser->run(ctx, \*this)

NdbMutex\_Unlock(m\_mutex);

//向客户端立即投递数据

m\_output->flush();

}

## 配置检查过程

mgmd请求检查节点 mgmd应答检查节点 mgmd应答检查节点

请求

-GSN\_CONFIG\_CHECK\_REQ-------->

-GSN\_CONFIG\_CHECK\_REQ----------------------------------------------------->

接受：

<- GSN\_CONFIG\_CHECK\_CONF-----

<- GSN\_CONFIG\_CHECK\_CONF--------------------------------------------------

拒绝：

<- GSN\_CONFIG\_CHECK\_REF--------

<- GSN\_CONFIG\_CHECK\_REF-----------------------------------------------------

## 配置变更过程

客户端请求：

客户端 接收请求的mgmd节点 其它mgmd节点

-GSN\_CONFIG\_CHANGE\_REQ-------->

PREPARING

第一阶段提交

-GSN\_CONFIG\_CHANGE\_IMPL\_REQ->

PREPARE

<-GSN\_CONFIG\_CHANGE\_IMPL\_CONF-

COMMITING

第二阶段提交

-GSN\_CONFIG\_CHANGE\_IMPL\_REQ->

COMMIT

<-GSN\_CONFIG\_CHANGE\_IMPL\_CONF-

完成变更

<- GSN\_CONFIG\_CHANGE\_CONF----

mgmd启动初始化变更：

无客户端参与

接收请求的mgmd节点 其它mgmd节点

PREPARING

第一阶段提交

-GSN\_CONFIG\_CHANGE\_IMPL\_REQ->

PREPARE

<-GSN\_CONFIG\_CHANGE\_IMPL\_CONF-

COMMITING

第二阶段提交

-GSN\_CONFIG\_CHANGE\_IMPL\_REQ->

COMMIT

<-GSN\_CONFIG\_CHANGE\_IMPL\_CONF-

完成变更

# ndbd节点启动流程

int real\_main(int argc, char\*\* argv)

{

//进行命令行参数处理

ho\_error=handle\_options(&argc, &argv, my\_long\_options, ndb\_std\_get\_one\_option)

//第一遍进入real\_main 时，先进入angel\_run，进行配置过程，第二遍real\_main 时，将进入ndbd\_run，进行实际的初始化和执行

angel\_run(progname, original\_args, opt\_ndb\_connectstring, opt\_ndb\_nodeid, opt\_bind\_address, opt\_initial, opt\_no\_start, opt\_daemon);

}

void

angel\_run(const char\* progname, const Vector<BaseString>& original\_args, const char\* connect\_str, int force\_nodeid, const char\* bind\_address, bool initial, bool no\_start, bool daemon)

{

//向mgmd取回配置项和节点id，并进行配置工作，设置各项参数，与mysqld节点的初始化相同

retriever.do\_connect(connnect\_retries, connect\_delay, verbose)

const Uint32 nodeid = retriever.allocNodeId(alloc\_retries, alloc\_delay);

ndb\_mgm\_configuration \* config = retriever.getConfig(nodeid);

configure(config, nodeid)

//构建一个管道fds，用于与即将创建的ndbd子进程通信，接收其消息

//创建一个ndbd子进程，带有下列参数：--report\_fd, --allocated\_nodeid, --initial, --nostart，子进程将重新进入real\_main，但是路径进入ndbd\_run

//监听子进程ndbd的退出状态，并报告，若子进程异常退出，则将重启子进程ndbd

}

void ndbd\_run(bool foreground, int report\_fd, const char\* connect\_str, int force\_nodeid, const char\* bind\_address, bool no\_start, bool initial, bool initialstart, unsigned allocated\_nodeid)

{

//创建若干全局对象，包括：

//Configuration

//WatchDog

//ThreadConfig

//SimBlockList

//SocketServer

//Ndbd\_mem\_manager

globalEmulatorData.create();

//向mgmd取回配置项和节点id，并进行配置工作，设置文件系统路径，包括FileSystemPath、BackupDataDir，设置线程运行及多线程配置参数

theConfig->init(no\_start, initial, initialstart)

theConfig->fetch\_configuration(connect\_str, force\_nodeid, bind\_address, allocated\_nodeid);

theConfig->setupConfiguration();

//打印启动信息

systemInfo(\* theConfig, \* theConfig->m\_logLevel);

globalEmulatorData.theThreadConfig->init();

//启动一个watchdog线程

NdbThread\* pWatchdog = globalEmulatorData.theWatchDog->doStart();

//核心初始化函数，初始化内存管理器，该内存管理器是一个基于伙伴系统的管理器，预先为ndbd分配内存，由其他BLOCK使用，当分配内存超过内存管理器的最大容量时，系统便不可用

//主要的内存组成部分包括：

//shared\_mem：共享内存，默认为CFG\_DB\_SGA（8M）

//tupmem：DBTUP页内存，行记录空间内存，默认为CFG\_TUP\_PAGE+CFG\_DB\_INDEX\_MEM

//filepages：redo日志内存，默认为CFG\_DB\_REDO\_BUFFER+NDB\_FILE\_BUFFER\_SIZE

//jbpages：任务队列内存，通过computer\_jb\_pages计算

//sbpages：发送缓冲，默认为CFG\_TOTAL\_SEND\_BUFFER\_MEMORY + CFG\_EXTRA\_SEND\_BUFFER\_MEMORY

//pgman\_pages：磁盘记录页缓冲，默认为CFG\_DB\_DISK\_PAGE\_BUFFER\_MEMORY（64M）/8K + 64 + LCP\_RESTORE\_BUFFER（128个页）

//stpages：元数据schema事务页数，默认为64

init\_global\_memory\_manager(globalEmulatorData, &watchCounter)

//核心初始化函数，将系统配置和内存管理器作为进程上下文，传递给各个BLOCK，创建各个BLOCK的实例，每个BLOCK都会注册其消息回调，使得稍后建立的消息处理模拟器得以工作

globalEmulatorData.theSimBlockList->load(globalEmulatorData);

//开始ndbd各个BLOCK的启动流程

switch(globalData.theRestartFlag){

case initial\_state:

//初始化启动：initial\_start/ initial\_node\_restart

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_CMVMI);

break;

case perform\_start:

//重启：system\_restart/node\_restart

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_CMVMI);

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_STARTING);

break;

default:

assert("Illegal state globalData.theRestartFlag" == 0);

//类似于mysqld与mgmd，初始化TransporterRegistry，管理各个节点到达的连接，同时启动TransporterService，与其它节点交换节点id和节点类型，当有服务到达时，有消息处理虚拟机进行poll，并处理请求，而连接accept由SocketServer服务器进行处理

globalTransporterRegistry.startSending();

globalTransporterRegistry.startReceiving();

globalTransporterRegistry.start\_service(\*globalEmulatorData.m\_socket\_server)

NdbThread\* pSockServ = globalEmulatorData.m\_socket\_server->startServer();

//注册创建的几个线程，包括主线程

globalEmulatorData.theConfiguration->addThread(pTrp, SocketClientThread);

globalEmulatorData.theConfiguration->addThread(pWatchdog, WatchDogThread);

globalEmulatorData.theConfiguration->addThread(pSockServ, SocketServerThread);

NdbThread \*pThis = NdbThread\_CreateObject(0);

Uint32 inx = globalEmulatorData.theConfiguration->addThread(pThis, MainThread);

//进入消息处理虚拟机循环

globalEmulatorData.theThreadConfig->ipControlLoop(pThis, inx);

}

# ndbd节点消息处理虚拟机工作流程

void ThreadConfig::ipControlLoop(NdbThread\*, Uint32 thread\_index)

{

//初始化一个计数器，用于跟踪当前的毫秒

globalData.internalMillisecCounter = NdbTick\_CurrentMillisecond();

while (1)

{

1. //发送缓冲发送数据，配个Transporter都将发送一部分数据

globalTransporterRegistry.performSend();

2. //检查索引连接Transporter的状态是否关闭

if (i++ >= 20)

{

execute\_loop\_constant =

globalEmulatorData.theConfiguration->schedulerExecutionTimer();

min\_spin\_time = globalEmulatorData.theConfiguration->schedulerSpinTimer();

globalData.incrementWatchDogCounter(5);

globalTransporterRegistry.update\_connections();

i = 0;

}

3. //至少执行一次，若处理完计时器，并接收完客户请求后，任务队列仍然为IDLE，方可再进行循环，继续处理

do

{

globalData.incrementWatchDogCounter(2);

4. //处理到期的计时器，检查globalTimeQueue的theShortQueue和theLongQueue，将其中到期的计时器对应的job从JBD移入JBA队列，下一步在处理消息时，这些超时消息将得到处理

scanTimeQueue();

//若有spin计时器处在允许中且未到期，则额外进行一些spin操作，检查连接状态

if (LEVEL\_IDLE == globalData.highestAvailablePrio)

{

timeOutMillis = 10;

if (min\_spin\_time && !yield\_flag)

{

if (spinning)

res2 = NdbTick\_getMicroTimer(&end\_micro);

if (!(res1 + res2))

{

micros\_passed = (Uint32)NdbTick\_getMicrosPassed(start\_micro, end\_micro);

if (micros\_passed < min\_spin\_time)

timeOutMillis = 0;

}

}

}

if (spinning && timeOutMillis > 0 && i++ >= 20)

{

globalData.incrementWatchDogCounter(5);

globalTransporterRegistry.update\_connections();

i = 0;

}

5. //轮询并接受每个Transporter的数据，相当于接受客户请求

globalData.incrementWatchDogCounter(7);

{

bool poll\_flag;

if (yield\_flag)

{

globalEmulatorData.theConfiguration->yield\_main(thread\_index, TRUE);

poll\_flag= globalTransporterRegistry.pollReceive(timeOutMillis);

globalEmulatorData.theConfiguration->yield\_main(thread\_index, FALSE);

res3= NdbTick\_getMicroTimer(&yield\_micro);

}

else

poll\_flag= globalTransporterRegistry.pollReceive(timeOutMillis);

if (poll\_flag)

{

globalData.incrementWatchDogCounter(8);

globalTransporterRegistry.performReceive();

}

yield\_flag= FALSE;

}

spinning = true;

//产生一个NDBFS的GSN\_SEND\_PACKET消息

globalScheduler.postPoll();

} while (LEVEL\_IDLE == globalData.highestAvailablePrio);

6. //进入消息处理过程

exec\_again= 0;

do

{

//处理各类消息的核心驱动函数

globalScheduler.doJob();

停止处理消息条件//若最小通用计时器和最小自旋计时器均到期，则不进入下一轮消息处理循环

if (!execute\_loop\_constant && !min\_spin\_time)

break;

res2= NdbTick\_getMicroTimer(&end\_micro);

if (res2)

break;

micros\_passed = (Uint32)NdbTick\_getMicrosPassed(start\_micro, end\_micro);

tot\_exec\_time += micros\_passed;

//循环每执行8192次，将向CMVMI报告一个GSN\_EVENT\_REP消息

if (no\_exec\_loops++ >= 8192)

{

Uint32 expired\_time =

(Uint32)NdbTick\_getMicrosPassed(statistics\_start\_micro, end\_micro);

statistics\_start\_micro = end\_micro;

globalScheduler.reportThreadConfigLoop(expired\_time, execute\_loop\_constant, &no\_exec\_loops, &tot\_exec\_time, &no\_extra\_loops, &tot\_extra\_time);

}

停止处理消息条件//若执行时间超过通用计时器期限，则不进入下一轮消息处理循环

if (micros\_passed > execute\_loop\_constant ||

停止处理消息条件//若连续处理消息3个循环，则不进入下一轮消息处理循环

(exec\_again > 1))

break;

exec\_again++;

//在消息处理循环中，若仍有空闲时间可用，则利用这段时间接收用户请求，最多进行2次

if (!globalTransporterRegistry.pollReceive(0))

break;

no\_extra\_loops++;

tot\_extra\_time += micros\_passed;

start\_micro = end\_micro;

globalData.incrementWatchDogCounter(8);

globalTransporterRegistry.performReceive();

} while (1);

}

out:

globalData.incrementWatchDogCounter(6);

//退出时清空发送缓冲

globalTransporterRegistry.performSend();

}

void FastScheduler::doJob()

{

register Signal\* signal = getVMSignals();

//取得当前可用的优先级最高的队列

register Uint32 tHighPrio= globalData.highestAvailablePrio;

do{

while ((tHighPrio < LEVEL\_IDLE) && (loopCount < TloopMax))

{

//从优先级最高的队列中取出消息

register Uint32 gsnbnr = theJobBuffers[tHighPrio].retrieve(signal);

if (reg\_bnr > 0) {

Uint32 tJobCounter = globalData.JobCounter;

Uint64 tJobLap = globalData.JobLap;

//取得消息对应的处理BLOCK

SimulatedBlock\* b = globalData.getBlock(reg\_bnr);

theJobPriority[tJobCounter] = (Uint8)tHighPrio;

globalData.JobCounter = (tJobCounter + 1) & 4095;

globalData.JobLap = tJobLap + 1;

//执行消息对应的回调

b->executeFunction(reg\_gsn, signal);

//当前优先级队列不为空，下次仍然从该队列取消息

tHighPrio = globalData.highestAvailablePrio;

} else {

//当前优先级队列为空，下次从优先级更低的队列取消息

tHighPrio++;

globalData.highestAvailablePrio = tHighPrio;

}//if

loopCount++;

}//while

//向DBLQH、DBTC、DBTUP、NDBFS发送GSN\_SEND\_PACKED以批量投递消息，加快消息投递速度

sendPacked();

b\_lqh->executeFunction(GSN\_SEND\_PACKED, signal);

b\_tc->executeFunction(GSN\_SEND\_PACKED, signal);

b\_tup->executeFunction(GSN\_SEND\_PACKED, signal);

b\_fs->executeFunction(GSN\_SEND\_PACKED, signal);

tHighPrio = globalData.highestAvailablePrio;

} while ((getBOccupancy() > MAX\_OCCUPANCY) || ((loopCount < TloopMax) && (tHighPrio < LEVEL\_IDLE)));

theDoJobCallCounter ++;

theDoJobTotalCounter += loopCount;

if (theDoJobCallCounter == 8192) {

reportDoJobStatistics(theDoJobTotalCounter >> 13);

theDoJobCallCounter = 0;

theDoJobTotalCounter = 0;

}

}

# ndbd节点API接口访问流程

该访问流程介绍以如下代码为范例，且认为，所有的提交过程均为成功，一些代码行后标注的数字，为总体阶段标识，这个阶段标识也在下面的代码分析中出现，每个阶段标识出现在代码分析的前部，表明其所属的阶段：

static void test\_insert(Ndb &myNdb)

{

//通过Ndb描述符取得需要操作的表”api\_test”

const NdbDictionary::Dictionary\* myDict= myNdb.getDictionary();

const NdbDictionary::Table \*myTable= myDict->getTable("api\_test");

//开启一个新事务

NdbTransaction \*myTransaction= myNdb.startTransaction(); //1 | 2 | 3

//开启事务上的操作

NdbOperation \*myOperation= myTransaction->getNdbOperation(myTable);

//操作定义为插入一个新行，设置”ATTR1”列属性为1，设置”ATTR2”列属性为2，equal本是一个比较操作，在插入操作中出现时，也会将这个equal是用的属性值作为插入属性，合并到要插入的行中

myOperation->insertTuple(); //4

myOperation->equal("ATTR1", 1); //5

myOperation->setValue("ATTR2", 2); //6

//执行事务

myTransaction->execute(NdbTransaction::Commit) //7 | 8 | 9

//关闭事务

myNdb.closeTransaction(myTransaction); //10

}

## 客户端API节点->服务器ndbd节点消息交互

1阶段

NdbTransaction\* Ndb::startTransaction(const NdbDictionary::Table \*table, const char \* keyData, Uint32 keyLen)

{

NdbTransaction \*trans= startTransactionLocal(0, nodeId, 0);

}

NdbTransaction\*Ndb::startTransactionLocal(Uint32 aPriority, Uint32 nodeId, Uint32 instance)

{

NdbTransaction\* tConnection;

//向ndbd节点连接

tConnection = doConnect(nodeId, instance);

//初始化事务描述符

tConnection->init()

//将事务加入Ndb对象的事务列表中

theTransactionList = tConnection; // into a transaction list.

tConnection->next(tConNext); // Add the active connection object

tConnection->setTransactionId(tFirstTransId);

tConnection->thePriority = aPriority;

//最终返回这个新建立的事务描述符，此时，有一个异步发出的消息GSN\_TCSEIZEREQ，事务管理器的DBTC将处理这个消息，成功后返回GSN\_TCSEIZECONF消息

DBUG\_RETURN(tConnection);

}

NdbTransaction\* Ndb::doConnect(Uint32 tConNode, Uint32 instance)

{

//这里有一个优化的选择连接过程，即若支持优化选择节点，将尝试向一个最优ndbd节点进行连接，否则轮询所有ndbd节点，这样保证了ndbd节点可以均匀的作为事务管理器

NDB\_connect(tConNode, instance)

}

int Ndb::NDB\_connect(Uint32 tNode, Uint32 instance)

{

//获取一个空闲事务描述符

NdbTransaction \* tNdbCon = getNdbCon();

//获取一个空闲消息描述符

NdbApiSignal\* tSignal = getSignal();

//构造一个GSN\_TCSEIZEREQ，该消息将发向前面选择的事务管理器节点的DBTC BLOCK，ndbd的BLOCK类似于erlang的module，兼具运行时进程名的含义。

tSignal->setSignal(GSN\_TCSEIZEREQ, DBTC)

//发送消息到目的地，然后poll返回的消息

tReturnCode= sendRecSignal(tNode, WAIT\_TC\_SEIZE, tSignal, 0, &nodeSequence);

releaseSignal(tSignal);

tNdbCon->setConnectedNodeId(tNode, nodeSequence);

tNdbCon->setMyBlockReference(theMyRef);

//发送及同步等待成功后，关联该事务描述符和事务管理器节点

prependConnectionArray(tNdbCon, tNode);

}

int Ndb::sendRecSignal(Uint16 node\_id, Uint32 aWaitState, NdbApiSignal\* aSignal, Uint32 conn\_seq, Uint32 \*ret\_conn\_seq)

{

//实际的消息发送过程，先将消息推入发送缓冲区，达到发送限度时通过writev将消息发出

theImpl->sendSignal(aSignal, node\_id);

//首先清空发送缓冲区，然后通过PollGuard轮询套接字，当消息从事务管理器返回时，PollGuard在解包的同时，对数据包调用处理回调

poll\_guard.wait\_n\_unlock(WAITFOR\_RESPONSE\_TIMEOUT,node\_id, aWaitState, false);

}

2阶段在事务管理器节点的DBTC中处理

3阶段

API节点在数据到达后通过unpack函数解包，同时选择消息处理回调，消息处理函数为：

void Ndb::handleReceivedSignal(const NdbApiSignal\* aSignal, const LinearSectionPtr ptr[3])

{

const Uint32 tSignalNumber = aSignal->readSignalNumber();

switch (tSignalNumber){

case GSN\_TCKEYCONF:

case GSN\_TCINDXCONF:

…

case GSN\_TCSEIZECONF:

{

tFirstDataPtr = int2void(tFirstData);

tCon = void2con(tFirstDataPtr);

tReturnCode = tCon->receiveTCSEIZECONF(aSignal);

tNewState = NO\_WAIT;

break;

}

…

}

int NdbTransaction::receiveTCSEIZECONF(const NdbApiSignal\* aSignal)

{

//取得DBTC为此次事务分配的事务id

theTCConPtr = (Uint32)aSignal->readData(2);

if (aSignal->getLength() >= 3)

m\_tcRef = aSignal->readData(3);

else

m\_tcRef = numberToRef(DBTC, theDBnode);

theStatus = Connected;

}

4阶段

int NdbOperation::insertTuple()

{

theStatus = OperationDefined;

//标识插入请求

theOperationType = InsertRequest;

tNdbCon->theSimpleState = 0;

theErrorLine = tErrorLine++;

//标识独占锁

theLockMode = LM\_Exclusive;

m\_abortOption = AbortOnError;

return 0;

}

5阶段

//所有的NdbOperation::equal最终调用该函数

int NdbOperation::equal(Uint32 anAttrId, const char\* aValuePassed)

{

equal\_impl(col, aValuePassed);

}

int NdbOperation::equal\_impl(const NdbColumnImpl\* tAttrInfo, const char\* aValuePassed)

{

//根据ndb协议，此处将匹配列及值构造为这次query的GSN\_KEYINFO消息，附加在主请求GSN\_TCKEYREQ之后，对于插入/写请求，还需要将匹配列及值构造为这次query的GSN\_ATTRINFO消息，GSN\_KEYINFO用于选择列，GSN\_ATTRINFO用于更新列

//注意，ndb使用了优化的消息传递机制，仅当GSN\_KEYINFO与GSN\_ATTRINFO的内容太大而不能放入主请求GSN\_TCKEYREQ的参数部分中时，才生成单独的GSN\_KEYINFO与GSN\_ATTRINFO消息，否则，仍旧附着在GSN\_TCKEYREQ中

//注意，发送主请求GSN\_TCKEYREQ时，也会将GSN\_KEYINFO与GSN\_ATTRINFO打包一并发出

}

6阶段

//所有的NdbOperation::setValue最终调用该函数

int NdbOperation::setValue( const NdbColumnImpl\* tAttrInfo, const char\* aValuePassed)

{

//根据ndb协议，此处将匹配列及值构造为这次query的GSN\_ATTRINFO消息，附加在主请求GSN\_TCKEYREQ之后，GSN\_ATTRINFO用于更新列

}

7阶段

//所有的NdbTransaction::execute最终调用该函数，aTypeOfExec在该例中为NdbTransaction::Commit，即事务立即提交，abortOption 为NdbOperation::DefaultAbortOption，forceSend为0

int NdbTransaction::execute(ExecType aTypeOfExec, NdbOperation::AbortOption abortOption, int forceSend)

{

//该例没有blob字段

executeNoBlobs(aTypeOfExec, abortOption, forceSend)

}

int NdbTransaction::executeNoBlobs(NdbTransaction::ExecType aTypeOfExec, NdbOperation::AbortOption abortOption, int forceSend)

{

//构造并发送所有操作的GSN\_TCKEYREQ请求

executeAsynchPrepare(aTypeOfExec, NULL, NULL, abortOption);

//发送完请求后，同步等待事务管理器节点返回，与3阶段相同，此处也通过PollGuard轮询套接字，当消息从事务管理器返回时，PollGuard在解包的同时，对数据包调用处理回调，进入9阶段

int noOfComp = tNdb->sendPollNdb(3 \* timeout, 1, forceSend);

}

void NdbTransaction::executeAsynchPrepare(NdbTransaction::ExecType aTypeOfExec, NdbAsynchCallback aCallback, void\* anyObject, NdbOperation::AbortOption abortOption)

{

NdbOperation\* tOp = theFirstOpInList;

//遍历事务的每个操作，将操作的GSN\_TCKEYREQ|GSN\_KEYINFO|GSN\_ATTRINFO打包写入发送缓冲区，在必要时将数据推出

while (tOp)

{

NdbOperation\* tNextOp = tOp->next();

if (tOp->Status() == NdbOperation::UseNdbRecord)

tReturnCode = tOp->prepareSendNdbRecord(abortOption);

else

tReturnCode= tOp->prepareSend(theTCConPtr, theTransactionId, abortOption);

tOp = tNextOp;

}

}

8阶段在事务管理器节点的DBTC中处理

9阶段

void Ndb::handleReceivedSignal(const NdbApiSignal\* aSignal, const LinearSectionPtr ptr[3])

{

const Uint32 tSignalNumber = aSignal->readSignalNumber();

switch (tSignalNumber){

case GSN\_TCKEYCONF:

case GSN\_TCINDXCONF:

//确认收到结果，更改事务状态

tReturnCode = tCon->receiveTCKEYCONF(keyConf, tLen);

//释放事务资源

completedTransaction(tCon);

…

…

}

10阶段

void Ndb::closeTransaction(NdbTransaction\* aConnection)

{

//释放事务句柄

aConnection->release();

releaseNdbCon(aConnection);

}

## 服务器ndbd节点事务描述符申请

2阶段

void Dbtc::execTCSEIZEREQ(Signal\* signal)

{

//取得客户端API节点的节点id

const NodeId senderNodeId = refToNode(tapiBlockref);

const bool local = senderNodeId == getOwnNodeId() || senderNodeId == 0;

//分配一个空闲事务id，返回GSN\_TCSEIZECONF确认

seizeApiConnect(signal);

apiConnectptr.p->ndbapiConnect = tapiPointer;

apiConnectptr.p->ndbapiBlockref = tapiBlockref;

signal->theData[0] = apiConnectptr.p->ndbapiConnect;

signal->theData[1] = apiConnectptr.i;

signal->theData[2] = reference();

sendSignal(tapiBlockref, GSN\_TCSEIZECONF, signal, 3, JBB);

}

void Dbtc::seizeApiConnect(Signal\* signal)

{

//找到第一个空闲事务id返回给客户端

apiConnectptr.i = cfirstfreeApiConnect; /\* ASSIGN A FREE RECORD FROM \*/

ptrCheckGuard(apiConnectptr, capiConnectFilesize, apiConnectRecord);

cfirstfreeApiConnect = apiConnectptr.p->nextApiConnect;

apiConnectptr.p->nextApiConnect = RNIL;

setApiConTimer(apiConnectptr.i, 0, \_\_LINE\_\_);

apiConnectptr.p->apiConnectstate = CS\_CONNECTED; /\* STATE OF CONNECTION \*/

tc\_clearbit(apiConnectptr.p->m\_flags, ApiConnectRecord::TF\_TRIGGER\_PENDING);

apiConnectptr.p->m\_special\_op\_flags = 0;

}

## 服务器ndbd节点事务处理全局事务DBTC部分

8阶段

void Dbtc::execTCKEYREQ(Signal\* signal)

{

//初始化客户端的api连接描述符ApiConnectRecord

initApiConnectRec(signal, regApiPtr);

//检查表版本

localTabptr.p->checkTable(tcKeyReq->tableSchemaVersion)

//分配一个空闲事务描述符结构TcConnectRecord

seizeTcRecord(signal)

//分配一个空闲缓存结构CacheRecord

seizeCacheRecord(signal)

//取回请求申请的事务描述符结构和缓存结构

TcConnectRecord \* const regTcPtr = tcConnectptr.p;

CacheRecord \* const regCachePtr = cachePtr.p;

//设置事务描述符结构和缓存结构的各项成员，检查是否所有ndbd节点都支持spj

//对于长短请求类型，矫正器请求头部、GSN\_KEYINFO|GSN\_ATTRINFO部分

//常见的请求均为长请求，即GSN\_TCKEYREQ|GSN\_KEYINFO|GSN\_ATTRINFO打包在一起发送到事务管理器节点，取回GSN\_KEYINFO|GSN\_ATTRINFO。

//更新统计量和请求的状态机

//长请求可以直接进行事务处理

if (regCachePtr->isLongTcKeyReq)

{

jam();

/\* Have all the KeyInfo (and AttrInfo), process now \*/

tckeyreq050Lab(signal);

}

}

void Dbtc::tckeyreq050Lab(Signal\* signal)

{

//根据请求的KEYINFO段的值，计算其hash值，默认使用md5，这个hash值将用于分片节点路由

hash(signal); /\* NOW IT IS TIME TO CALCULATE THE HASH VALUE\*/

UintR ThashValue = thashValue;

UintR TdistrHashValue = tdistrHashValue;

DiGetNodesReq \* const req = (DiGetNodesReq \*)&signal->theData[0];

req->tableId = Ttableref;

req->hashValue = TdistrHashValue;

req->distr\_key\_indicator = regCachePtr->distributionKeyIndicator;

req->jamBufferPtr = jamBuffer();

//此处直接调用GSN\_DIGETNODESREQ对应的回调，利用计算出的主键hash值，取得该条记录所处的节点id等数据分片信息

EXECUTE\_DIRECT(DBDIH, GSN\_DIGETNODESREQ, signal,

DiGetNodesReq::SignalLength, 0); //8.1

8.2阶段

//取回记录的分片信息

DiGetNodesConf \* conf = (DiGetNodesConf \*)&signal->theData[0];

UintR Tdata2 = conf->reqinfo;

//将获取的分片信息存入事务描述符TcConnectRecord中，可以发现一个数据分片最多可以有四个副本

UintR Tdata1 = conf->fragId;

UintR Tdata3 = conf->nodes[0];

UintR Tdata4 = conf->nodes[1];

UintR Tdata5 = conf->nodes[2];

UintR Tdata6 = conf->nodes[3];

regCachePtr->fragmentid = Tdata1;

tnodeinfo = Tdata2;

regTcPtr->tcNodedata[0] = Tdata3;

regTcPtr->tcNodedata[1] = Tdata4;

regTcPtr->tcNodedata[2] = Tdata5;

regTcPtr->tcNodedata[3] = Tdata6;

//对于读操作，可以在这些副本节点中任选一个作为参与事务的节点，对于写操作，则需要所有副本节点参与事务

//向DBLQH发出服务请求

attrinfoDihReceivedLab(signal);

}

void Dbtc::attrinfoDihReceivedLab(Signal\* signal)

{

//首先向第一个节点发出请求

Uint16 Tnode = regTcPtr->tcNodedata[0];

Uint32 instanceKey = regTcPtr->lqhInstanceKey;

BlockReference lqhRef;

if(regCachePtr->viaSPJFlag){

//ndbout << "TC:Choosing SPJ." << endl;

lqhRef = numberToRef(DBSPJ, instanceKey, Tnode);

}else{

//ndbout << "TC:Choosing LQH." << endl;

lqhRef = numberToRef(DBLQH, instanceKey, Tnode);

}

packLqhkeyreq(signal, lqhRef);

}

void Dbtc::packLqhkeyreq(Signal\* signal, BlockReference TBRef)

{

sendlqhkeyreq(signal, TBRef);

}

void Dbtc::sendlqhkeyreq(Signal\* signal, BlockReference TBRef)

{

//构造一个巨大的lqhKeyReq结构，用于向目的DBLQH发出请求

sig0 = tcConnectptr.i;

sig2 = regCachePtr->hashValue;

sig4 = cownref;

sig5 = regTcPtr->savePointId;

lqhKeyReq->clientConnectPtr = sig0;

lqhKeyReq->attrLen = tslrAttrLen;

lqhKeyReq->hashValue = sig2;

lqhKeyReq->requestInfo = Tdata10;

lqhKeyReq->tcBlockref = sig4;

lqhKeyReq->savePointId = sig5;

//在请求结构中记录第2个节点以及事务id

sig0 = regCachePtr->tableref + ((regCachePtr->schemaVersion << 16) & 0xFFFF0000);

sig1 = regCachePtr->fragmentid + (regTcPtr->tcNodedata[1] << 16);

sig2 = regApiPtr->transid[0];

sig3 = regApiPtr->transid[1];

sig4 =

(regTcPtr->m\_special\_op\_flags & TcConnectRecord::SOF\_INDEX\_TABLE\_READ) ?

reference() : regApiPtr->ndbapiBlockref;

sig5 = regTcPtr->clientData;

sig6 = regCachePtr->scanInfo;

lqhKeyReq->tableSchemaVersion = sig0;

lqhKeyReq->fragmentData = sig1;

lqhKeyReq->transId1 = sig2;

lqhKeyReq->transId2 = sig3;

lqhKeyReq->scanInfo = sig6;

lqhKeyReq->variableData[0] = sig4;

lqhKeyReq->variableData[1] = sig5;

//在原始的ApiConnectRecord结构中记录此次可请求的节点

regApiPtr->m\_transaction\_nodes.set(regTcPtr->tcNodedata[0]);

regApiPtr->m\_transaction\_nodes.set(regTcPtr->tcNodedata[1]);

regApiPtr->m\_transaction\_nodes.set(regTcPtr->tcNodedata[2]);

regApiPtr->m\_transaction\_nodes.set(regTcPtr->tcNodedata[3]);

//在请求结构中记录第3、4个节点

if (regTcPtr->lastReplicaNo > 1) {

sig0 = (UintR)regTcPtr->tcNodedata[2] +

(UintR)(regTcPtr->tcNodedata[3] << 16);

lqhKeyReq->variableData[nextPos] = sig0;

nextPos++;

}//if

//对于长请求，可以直接向第一个节点的DBLQH发出请求

if (regCachePtr->useLongLqhKeyReq)

{

sendSignal(TBRef, GSN\_LQHKEYREQ, signal, nextPos + LqhKeyReq::FixedSignalLength, JBB, &handle); //8.3

ndbassert( handle.m\_cnt == 0 );

regCachePtr->keyInfoSectionI= RNIL;

regCachePtr->attrInfoSectionI= RNIL;

}

}

8.3-8.4阶段

8.3阶段由DBLQH处理事务请求，完成本地事务部分

8.4阶段由DBLQH返回一条GSN\_LQHKEYCONF消息

8.5阶段

该阶段由DBTC收到GSN\_LQHKEYCONF消息，进行处理

void Dbtc::execLQHKEYCONF(Signal\* signal)

{

const LqhKeyConf \* lqhKeyConf = CAST\_CONSTPTR(LqhKeyConf, signal->getDataPtr());

tlastLqhConnect = lqhKeyConf->connectPtr;

TtcConnectptrIndex = lqhKeyConf->opPtr;

tlastLqhBlockref = lqhKeyConf->userRef;

treadlenAi = lqhKeyConf->readLen;

TcConnectRecord \*localTcConnectRecord = tcConnectRecord;

//取回DBTC本地记录的事务描述符

TcConnectRecord\* const regTcPtr = &localTcConnectRecord[TtcConnectptrIndex];

//取回DBTC本地记录的API连接描述符

regApiPtr.i = TapiConnectptrIndex;

regApiPtr.p = &localApiConnectRecord[TapiConnectptrIndex];

apiConnectptr.i = TapiConnectptrIndex;

apiConnectptr.p = regApiPtr.p;

//验证事务id是否正确

compare\_transid1 = regApiPtr.p->transid[0] ^ Ttrans1;

compare\_transid2 = regApiPtr.p->transid[1] ^ Ttrans2;

compare\_transid1 = compare\_transid1 | compare\_transid2;

if (compare\_transid1 != 0) {

warningReport(signal, 24);

return;

}//if

//此处根据几种情况处理不同分支

//1.是否有发射的触发器

//2.是否为索引表读

//3.操作是否由触发器产生

//4.普通事务

//对于普通事务，进入如下分支：

lqhKeyConf\_checkTransactionState(signal, regApiPtr);

}

void Dbtc::lqhKeyConf\_checkTransactionState(Signal \* signal, Ptr<ApiConnectRecord> regApiPtr)

{

ConnectionState TapiConnectstate = regApiPtr.p->apiConnectstate;

//对于不同的请求，此处会进入不同的状态，正常情况下，读请求将进入CS\_RECEIVING状态，而写请求进入CS\_START\_COMMITTING状态

//CS\_RECEIVING状态下，不需要提交，而是直接向客户端API节点返回GSN\_TCKEYCONF

//CS\_START\_COMMITTING状态下，进行验证和提交

switch (TapiConnectstate) {

case CS\_START\_COMMITTING:

if (TnoOfOutStanding == 0) {

jam();

//进行验证和提交

diverify010Lab(signal);

return;

} else if (TnoOfOutStanding > 0) {

if (regApiPtr.p->tckeyrec == ZTCOPCONF\_SIZE) {

jam();

sendtckeyconf(signal, 0);

return;

}

else if (tc\_testbit(regApiPtr.p->m\_flags,

ApiConnectRecord::TF\_INDEX\_OP\_RETURN))

{

jam();

sendtckeyconf(signal, 0);

return;

}//if

jam();

return;

} else {

TCKEY\_abort(signal, 44);

return;

}//if

return;

case CS\_STARTED:

case CS\_RECEIVING:

if (TnoOfOutStanding == 0) {

jam();

//直接向客户端API节点返回GSN\_TCKEYCONF

sendtckeyconf(signal, 2);

return;

} else {

if (regApiPtr.p->tckeyrec == ZTCOPCONF\_SIZE) {

jam();

sendtckeyconf(signal, 0);

return;

}

else if (tc\_testbit(regApiPtr.p->m\_flags,

ApiConnectRecord::TF\_INDEX\_OP\_RETURN))

{

jam();

sendtckeyconf(signal, 0);

return;

}//if

jam();

}//if

return;

…

}

}

void Dbtc::diverify010Lab(Signal\* signal)

{

UintR TfirstfreeApiConnectCopy = cfirstfreeApiConnectCopy;

ApiConnectRecord \* const regApiPtr = apiConnectptr.p;

signal->theData[0] = apiConnectptr.i;

signal->theData[1] = instance() ? instance() - 1 : 0;

if (regApiPtr->lqhkeyreqrec)

{

if (TfirstfreeApiConnectCopy != RNIL) {

seizeApiConnectCopy(signal);

regApiPtr->apiConnectstate = CS\_PREPARE\_TO\_COMMIT;

\* (EmulatedJamBuffer\*\*)(signal->theData+2) = jamBuffer();

//向DBDIH验证本地gci全局检查点是否正确

EXECUTE\_DIRECT(DBDIH, GSN\_DIVERIFYREQ,

signal, 2 + sizeof(void\*)/sizeof(Uint32), 0); //8.6

//验证完成，开始提交过程

if (signal->theData[3] == 0) {

execDIVERIFYCONF(signal); //8.7

}

return;

}

else

{

jam();

sendtckeyconf(signal, 1);

regApiPtr->apiConnectstate = CS\_CONNECTED;

regApiPtr->m\_transaction\_nodes.clear();

setApiConTimer(apiConnectptr.i, 0,\_\_LINE\_\_);

}

}

## 服务器ndbd节点事务处理数据分片管理DBDIH部分

8.1阶段

//计算行记录的主键对应的数据分片节点

void Dbdih::execDIGETNODESREQ(Signal\* signal)

{

//取回表id、行记录hash值

tabPtr.i = req->tableId;

Uint32 hashValue = req->hashValue;

Uint32 distr\_key\_indicator = req->distr\_key\_indicator;

//根据不同的hash方法，取得记录对应的分片节点

//API节点计算的hash值直接作为分片节点id

if (distr\_key\_indicator)

{

fragId = hashValue;

if (unlikely(fragId >= tabPtr.p->totalfragments))

{

thrjam(jambuf);

conf->zero= 1; //Indicate error;

signal->theData[1]= ZUNDEFINED\_FRAGMENT\_ERROR;

return;

}

}

//API节点计算的hash值通过一层间接映射得到分片节点id

else if (tabPtr.p->method == TabRecord::HASH\_MAP)

{

thrjam(jambuf);

Ptr<Hash2FragmentMap> ptr;

g\_hash\_map.getPtr(ptr, map\_ptr\_i);

fragId = ptr.p->m\_map[hashValue % ptr.p->m\_cnt];

if (unlikely(new\_map\_ptr\_i != RNIL))

{

thrjam(jambuf);

g\_hash\_map.getPtr(ptr, new\_map\_ptr\_i);

newFragId = ptr.p->m\_map[hashValue % ptr.p->m\_cnt];

if (newFragId == fragId)

{

thrjam(jambuf);

newFragId = RNIL;

}

}

}

// API节点计算的hash值通过掩码得到分片节点id，此时分片id实际上是一种线性关系

else if (tabPtr.p->method == TabRecord::LINEAR\_HASH)

{

thrjam(jambuf);

fragId = hashValue & tabPtr.p->mask;

if (fragId < tabPtr.p->hashpointer) {

thrjam(jambuf);

fragId = hashValue & ((tabPtr.p->mask << 1) + 1);

}//if

}

//API节点计算的hash值对总分片数取模，这是最常见的情况

else if (tabPtr.p->method == TabRecord::NORMAL\_HASH)

{

thrjam(jambuf);

fragId= hashValue % tabPtr.p->totalfragments;

}

else

{

thrjam(jambuf);

ndbassert(tabPtr.p->method == TabRecord::USER\_DEFINED);

conf->zero= 1; //Indicate error;

signal->theData[1]= ZUNDEFINED\_FRAGMENT\_ERROR;

return;

}

//通过一个分片表取得表的分片id对应的分片描述符

getFragstore(tabPtr.p, fragId, fragPtr);

//从分片描述符中提取分片对应的副本节点

Uint32 nodeCount = extractNodeInfo(jambuf, fragPtr.p, conf->nodes);

conf->zero = 0;

conf->reqinfo = sig2;

conf->fragId = fragId;

//该函数由Dbtc::tckeyreq050Lab直接调用，因此不需要通过消息返回结果

}

## 服务器ndbd节点事务处理本地事务DBLQH部分

8.3阶段

void Dblqh::execLQHKEYREQ(Signal\* signal)

{

//为请求分配一个TcConnectionrec结构

seizeTcrec();

//将消息携带的LqhKeyReq结构中保存的信息转移到这个新分配的TcConnectionrec结构中

//收集请求中包含的KEYINFO与ATTRINFO，仅考虑长请求的情况

endgettupkeyLab(signal);

}

void Dblqh::endgettupkeyLab(Signal\* signal)

{

prepareContinueAfterBlockedLab(signal);

}

void Dblqh::prepareContinueAfterBlockedLab(Signal\* signal)

{

//连接DBACC，记录key将通过LH3算法转换为本地存储的key，对于插入和删除，还会启动一个到索引hash表的插入和删除过程

exec\_acckeyreq(signal, tcConnectptr);

}

void Dblqh::exec\_acckeyreq(Signal\* signal, TcConnectionrecPtr regTcPtr)

{

//构造一个到DBACC的请求结构

Uint32 sig0, sig1, sig2, sig3, sig4;

sig0 = regTcPtr.p->accConnectrec;

sig1 = fragptr.p->accFragptr;

sig2 = regTcPtr.p->hashValue;

sig3 = regTcPtr.p->primKeyLen;

sig4 = regTcPtr.p->transid[0];

signal->theData[0] = sig0;

signal->theData[1] = sig1;

signal->theData[2] = taccreq;

signal->theData[3] = sig2;

signal->theData[4] = sig3;

signal->theData[5] = sig4;

//本地执行GSN\_ACCKEYREQ，进行主键转换、行锁定等工作

EXECUTE\_DIRECT(refToMain(regTcPtr.p->tcAccBlockref), GSN\_ACCKEYREQ,

signal, 7 + regTcPtr.p->primKeyLen); //8.3.1

//进一步需要向DBTUP请求

execACCKEYCONF(signal);

}

8.3.2阶段

void Dblqh::execACCKEYCONF(Signal\* signal)

{

//此时需要连接DBTUP，DBTUP需要表和分片信息、本地key、操作类型、schema版本号这些信息，DBTUP将直接向DBTC或应用程序发送ATTRINFO（作为返回数据，附着在返回的TRANSID\_AI消息之后）。需要schema版本号的原因在于，一个表可以有两份schema并存，以支持快速alter表等操作

//内存表的处理

if(!regTcPtr->m\_disk\_table)

acckeyconf\_tupkeyreq(signal, regTcPtr, regFragptr.p, localKey1, localKey2, RNIL);

//磁盘表的处理

else

acckeyconf\_load\_diskpage(signal, tcConnectptr, regFragptr.p, localKey1, localKey2);

}

void Dblqh::acckeyconf\_tupkeyreq(Signal\* signal, TcConnectionrec\* regTcPtr, Fragrecord\* regFragptrP, Uint32 lkey1, Uint32 lkey2, Uint32 disk\_page)

{

//构造一个到DBTUP的请求

//本地执行GSN\_TUPKEYREQ请求，进行实际的数据操作

EXECUTE\_DIRECT(tup, GSN\_TUPKEYREQ, signal, TupKeyReq::SignalLength); //8.3.3|8.3.4

}

8.3.3-8.3.4阶段

8.3.3阶段由DBTUP处理数据操作请求，完成本地数据操作部分

8.3.4阶段由DBTUP返回一条GSN\_TUPKEYCONF消息

8.3.5阶段

void Dblqh::execTUPKEYCONF(Signal\* signal)

{

FragrecordPtr regFragptr;

regFragptr.i = tcConnectptr.p->fragmentptr;

c\_fragment\_pool.getPtr(regFragptr);

fragptr = regFragptr;

if (tcConnectptr.p->seqNoReplica == 0) // Primary replica

tcConnectptr.p->noFiredTriggers = tupKeyConf->noFiredTriggers;

tupkeyConfLab(signal);

}

void Dblqh::tupkeyConfLab(Signal\* signal)

{

//取回数据操作返回结果TupKeyConf

const TupKeyConf \* const tupKeyConf = (TupKeyConf \*)&signal->theData[0];

TcConnectionrec \* const regTcPtr = tcConnectptr.p;

//对于简单读或脏读，可以直接进行提交

if (regTcPtr->operation == ZREAD && (regTcPtr->opSimple || regTcPtr->dirtyOp))

{

commitContinueAfterBlockedLab(signal);

return;

}//if

//对于其它写事务

rwConcludedLab(signal);

}

void Dblqh::rwConcludedLab(Signal\* signal)

{

//此时根据操作类型，需要进行日志记录，并向下一个分片副本发送GSN\_LQHKEYREQ，或向DBTC发送GSN\_LQHKEYCONF

//对于读操作，可以不记录日志，并打包消息批量返回给DBTC

if (regTcPtr->operation == ZREAD)

{

packLqhkeyreqLab(signal);

return;

} else {

FragrecordPtr regFragptr = fragptr;

//若不需要对操作进行日志记录，则直接开始向向下一个分片副本发送GSN\_LQHKEYREQ，或向DBTC发送GSN\_LQHKEYCONF

if (regFragptr.p->logFlag == Fragrecord::STATE\_FALSE)

{

if (regTcPtr->dirtyOp == ZTRUE)

{

//最终也调用packLqhkeyreqLab

commitContinueAfterBlockedLab(signal);

return;

} else {

regTcPtr->logWriteState = TcConnectionrec::NOT\_WRITTEN;

packLqhkeyreqLab(signal);

return;

}//if

//若操作需要日志记录，开始日志记录过程

} else {

logLqhkeyreqLab(signal);

return;

}//if

}//if

}

void Dblqh::logLqhkeyreqLab(Signal\* signal)

{

//写日志过程，包括写入日志头，记录key，记录value

checkNewMbyte(signal);

writeLogHeader(signal);

writeKey(signal);

writeAttrinfoLab(signal);

if (regTcPtr->dirtyOp != ZTRUE)

{

//事务操作不会立即提交日志，这个过程需要到提交阶段才可进行

packLqhkeyreqLab(signal);

} else {

//脏操作可以立即进入本地提交阶段，使修改生效

//同时记录日志

writeCommitLog(signal, logPartPtr);

localCommitLab(signal);

}

}

rwConcludedLab的所有分支最终进入packLqhkeyreqLab

//数据操作部分已经执行完成，现在需要将操作请求发送到下个DBLQH或将确认发送到DBTC

void Dblqh::packLqhkeyreqLab(Signal\* signal)

{

//所有副本的DBLQH均以处理完毕，此时向DBTC返回一个GSN\_LQHKEYCONF消息，通知其开始提交过程

if (regTcPtr->nextReplica == ZNIL)

{

sendLqhkeyconfTc(signal, regTcPtr->tcBlockref);

if (! (regTcPtr->dirtyOp || (regTcPtr->operation == ZREAD && regTcPtr->opSimple)))

{

jam();

regTcPtr->transactionState = TcConnectionrec::PREPARED;

releaseOprec(signal);

} else {

jam();

cleanUp(signal);

}//if

return;

}//if

//仍然有剩余副本需要处理操作

LqhKeyReq \* const lqhKeyReq = (LqhKeyReq \*)&signal->theData[0];

UintR Treqinfo;

UintR sig0, sig1, sig2, sig3, sig4, sig5, sig6;

Treqinfo = preComputedRequestInfoMask & regTcPtr->reqinfo;

//取得下一个副本节点id

Uint32 nextNodeId = regTcPtr->nextReplica;

Uint32 nextVersion = getNodeInfo(nextNodeId).m\_version;

//继续向下一个副本的DBLQH发送GSN\_LQHKEYREQ

BlockReference lqhRef = numberToRef(DBLQH, fragptr.p->lqhInstanceKey, regTcPtr->nextReplica);

sendSignal(lqhRef, GSN\_LQHKEYREQ, signal, LqhKeyReq::FixedSignalLength + nextPos, JBB, &handle);

//本地记录的事务装备变更为PREPARED，下次COMMIT提交请求到达时，将提交本地日志

regTcPtr->transactionState = TcConnectionrec::PREPARED;

releaseOprec(signal);

}

8.4阶段

//向DBTC返回GSN\_LQHKEYCONF，通知其开始提交过程

void Dblqh::sendLqhkeyconfTc(Signal\* signal, BlockReference atcBlockref)

{

Thostptr.i = refToNode(atcBlockref);

ptrCheckGuard(Thostptr, chostFileSize, hostRecord);

//这里的发送过程仍然利用了批量发送技巧，只是结果类型变更为了ZLQHKEYCONF

if (refToBlock(atcBlockref) == DBTC)

{

if (Thostptr.p->noOfPackedWordsTc > (25 - LqhKeyConf::SignalLength))

sendPackedSignalTc(signal, Thostptr.p);

} else {

updatePackedList(signal, Thostptr.p, Thostptr.i);

}//if

lqhKeyConf = (LqhKeyConf \*)

&Thostptr.p->packedWordsTc[Thostptr.p->noOfPackedWordsTc];

Thostptr.p->noOfPackedWordsTc += LqhKeyConf::SignalLength;

}

Uint32 ptrAndType = tcConnectptr.i | (ZLQHKEYCONF << 28);

Uint32 tcOprec = tcConnectptr.p->tcOprec;

Uint32 ownRef = cownref;

Uint32 readlenAi = tcConnectptr.p->readlenAi;

Uint32 transid1 = tcConnectptr.p->transid[0];

Uint32 transid2 = tcConnectptr.p->transid[1];

}

## 服务器ndbd节点事务处理本地索引查询及锁申请DBACC部分

8.3.1阶段

void Dbacc::execACCKEYREQ(Signal\* signal)

{

operationRecPtr.i = signal->theData[0]; /\* CONNECTION PTR \*/

fragrecptr.i = signal->theData[1]; /\* FRAGMENT RECORD POINTER \*/

//将请求中包含的操作信息转移到Dbacc的operationRecPtr成员，稍后将对该成员进行一系列处理

initOpRec(signal);

//若key为char类型，则对其进行字符集转换，转换为本地字符集，调用key的字符集对应的strnxfrm函数，最终调用cs->coll->strnxfrm进行字符集转换

if (operationRecPtr.p->tupkeylen && fragrecptr.p->hasCharAttr)

xfrmKeyData(signal);

//此处将使用hash值来查找key对应的内存页号和页内偏移，取出记录，以供下面的上锁处理

OperationrecPtr lockOwnerPtr;

const Uint32 found = getElement(signal, lockOwnerPtr);

Uint32 opbits = operationRecPtr.p->m\_op\_bits;

Uint32 op = opbits & Operationrec::OP\_MASK;

if (found == ZTRUE)

{

switch (op) {

//读/更新/删除/写/扫描/已有的元素，将操作标识上锁

case ZREAD: case ZUPDATE: case ZDELETE: case ZWRITE: case ZSCAN\_OP:

//当前行记录未上锁

if (!lockOwnerPtr.p)

{

if(op == ZWRITE)

{

opbits &= ~(Uint32)Operationrec::OP\_MASK;

opbits |= (op = ZUPDATE);

operationRecPtr.p->m\_op\_bits = opbits; // store to get correct ACCKEYCONF

}

opbits |= Operationrec::OP\_STATE\_RUNNING;

opbits |= Operationrec::OP\_RUN\_QUEUE;

sendAcckeyconf(signal);

//事务操作，需要申请锁

if (! (opbits & Operationrec::OP\_DIRTY\_READ))

{

Uint32 eh = gePageptr.p->word32[tgeElementptr];

operationRecPtr.p->scanBits = ElementHeader::getScanBits(eh);

operationRecPtr.p->hashvaluePart = ElementHeader::getHashValuePart(eh);

operationRecPtr.p->elementPage = gePageptr.i;

operationRecPtr.p->elementContainer = tgeContainerptr;

operationRecPtr.p->elementPointer = tgeElementptr;

operationRecPtr.p->elementIsforward = tgeForward;

//记录上锁动作

eh = ElementHeader::setLocked(operationRecPtr.i);

gePageptr.p->word32[tgeElementptr] = eh;

opbits |= Operationrec::OP\_LOCK\_OWNER;

//将操作记录入锁持有者队列

insertLockOwnersList(signal, operationRecPtr);

//脏读，不需要申请锁

} else {

opbits = Operationrec::OP\_EXECUTED\_DIRTY\_READ;

}//if

operationRecPtr.p->m\_op\_bits = opbits;

return;

//当前行记录已经上锁，将操作加入分片的锁队列中，同时进行死锁检查

} else {

accIsLockedLab(signal, lockOwnerPtr);

return;

}//if

break;

//插入一个已存在的行记录，仅检查操作是否持有锁，若未持有则拒绝请求

case ZINSERT:

insertExistElemLab(signal, lockOwnerPtr);

return;

break;

}//switch

} else if (found == ZFALSE) {

switch (op){

case ZWRITE:

opbits &= ~(Uint32)Operationrec::OP\_MASK;

opbits |= (op = ZINSERT);

//插入一条不存在的行记录，将行记录的key写入索引，稍后在DBTUP为行记录分配空间后，由DBTUP通知DBACC将索引与行记录位置关联起来

case ZINSERT:

opbits |= Operationrec::OP\_INSERT\_IS\_DONE;

opbits |= Operationrec::OP\_STATE\_RUNNING;

opbits |= Operationrec::OP\_RUN\_QUEUE;

operationRecPtr.p->m\_op\_bits = opbits;

insertelementLab(signal);

return;

break;

//读/更新/删除/扫描一条不存在的行记录，立即拒绝请求

case ZREAD: case ZUPDATE: case ZDELETE: case ZSCAN\_OP:

acckeyref1Lab(signal, ZREAD\_ERROR);

return;

}//switch

} else {

acckeyref1Lab(signal, found);

return;

}//if

return;

}

Uint32 Dbacc::getElement(Signal\* signal, OperationrecPtr& lockOwnerPtr)

{

//索引为hash索引，通过树形结构组织，叶节点存储了hash桶，桶内保存了hash值高6+x位相同的行记录key的hash值，在DBTUP中的分片页号和页内偏移

//找到key的hash值在索引中的hash桶号

getdirindex(signal);

tgePageindex = tgdiPageindex;

gePageptr = gdiPageptr;

//进行桶内的key比较，遍历桶内所有key，查找与传入KEYINFO相同的索引记录，若存在这样的记录，则记录这个位置并返回

//在行记录key比较的过程中需要通过Dbacc::readTablePk读取表的主键

//一个hash桶可能跨越多个页

}

## 服务器ndbd节点事务处理本地记录访问DBTUP部分

8.3.3阶段

8.3.4阶段

void Dbtup::execTUPKEYREQ(Signal\* signal)

{

//取出请求包含的数据

TupKeyReq \* tupKeyReq= (TupKeyReq \*)signal->getDataPtr();

//构造一个KeyReqStruct描述符，代理DBTUP处理此次请求，这个结构用于记录操作涉及的行记录的位置、创建的记录版本等

KeyReqStruct req\_struct(this);

Uint32 RoperPtr= tupKeyReq->connectPtr;

Uint32 Rfragptr= tupKeyReq->fragPtr;

c\_operation\_pool.getPtr(operPtr, RoperPtr);

Uint32 TrequestInfo= tupKeyReq->request;

//取出操作描述符、分片描述符

Operationrec \* regOperPtr= operPtr.p;

Fragrecord \* regFragPtr= fragptr.p;

//根据请求内容填充KeyReqStruct描述符

//将此次操作携带的ATTRINFO，即行内容（用于插入/更新/写操作）拷贝入临时缓冲区cinBuffer，今后的插入/更新/写操作会用到这个缓冲区

copyAttrinfo(regOperPtr, &cinBuffer[0], req\_struct.attrinfo\_len, attrInfoIVal);

//根据逻辑页号去的物理页号，PageMap是DBTUP的一项服务用于将逻辑页号映射到物理页号，映射过程的输入参数为分片号及逻辑页号，返回物理页号

regOperPtr->m\_tuple\_location.m\_page\_no= getRealpid(regFragPtr, req\_struct.frag\_page\_id);

//DBTUP的行记录格式包含五个部分：静态定长列/静态变长列/动态定长列/动态变长列/磁盘列，此处取得行记录的定长列的位置由此定位行记录的实际位置，对于插入操作，行记录应该不存在，此函数的结果无关紧要，对于读/更新/删除操作，此函数定位行记录在表分片中实际的页号和页内偏移

//将KeyReqStruct描述符的m\_tuple\_ptr成员定位到操作需要访问的记录位置（分片页号，页内偏移）

setup\_fixed\_part(&req\_struct, regOperPtr, regTabPtr);

//对于读操作，需要先为读取的行记录选择合适的版本，因为事务此前可能已经对本行进行了更新，此处需要读出它更新后的版本而不是原始版本，然进入正式的读取过程

if (Roptype == ZREAD)

{

setup\_read(&req\_struct, regOperPtr, regFragPtr, regTabPtr, disk\_page != RNIL)

handleReadReq(signal, regOperPtr, regTabPtr, &req\_struct)

req\_struct.log\_size= 0;

sendTUPKEYCONF(signal, &req\_struct, regOperPtr);

set\_trans\_state(regOperPtr, TRANS\_IDLE);

return;

}

//对于写操作，需要先将写操作记录在KeyReqStruct描述符的活动操作列表中

if(insertActiveOpList(operPtr, &req\_struct))

{

//对于插入操作，需要为插入操作分配行记录空间，检查触发器，维护辅助索引，通知DBACC新行记录空间的位置

if(Roptype == ZINSERT)

{

do\_insert:

Local\_key accminupdate;

Local\_key \* accminupdateptr = &accminupdate;

//插入操作为新行记录生成一个undo结构，记录行记录的新版本，并为行记录分配空间，accminupdateptr保存分配的行记录空间

handleInsertReq(signal, operPtr, fragptr, regTabPtr, &req\_struct, &accminupdateptr)

terrorCode = 0;

//检查插入触发器

checkImmediateTriggersAfterInsert(&req\_struct,regOperPtr,regTabPtr,disk\_page!= RNIL);

//辅助索引维护

if (!regTabPtr->tuxCustomTriggers.isEmpty())

executeTuxInsertTriggers(signal, regOperPtr, regFragPtr, regTabPtr)

//通知DBACC新分配的行记录空间，允许其进行关联

if (accminupdateptr)

c\_lqh->accminupdate(signal, regOperPtr->userpointer, accminupdateptr);

sendTUPKEYCONF(signal, &req\_struct, regOperPtr);

return;

}

//对于更新操作，需要为更新操作创建新的版本和undo结构，检查触发器，维护辅助索引

if (Roptype == ZUPDATE)

{

//插入操作为新行记录生成一个undo结构，记录行记录的新版本

handleUpdateReq(signal, regOperPtr, regFragPtr, regTabPtr, &req\_struct, disk\_page != RNIL)

//检查更新触发器

checkImmediateTriggersAfterUpdate(&req\_struct, regOperPtr, regTabPtr, disk\_page != RNIL);

//辅助索引维护

if (!regTabPtr->tuxCustomTriggers.isEmpty())

executeTuxUpdateTriggers(signal, regOperPtr, regFragPtr, regTabPtr) != 0))

sendTUPKEYCONF(signal, &req\_struct, regOperPtr);

return;

}//对于删除操作，需要为删除操作创建新的版本和undo结构，检查触发器，维护辅助索引的工作将放到提交过程中做

else if(Roptype == ZDELETE)

{

req\_struct.log\_size= 0;

//删除操作为新行记录生成一个undo结构，记录行记录的原始版本

handleDeleteReq(signal, regOperPtr, regFragPtr, regTabPtr, &req\_struct, disk\_page != RNIL)

terrorCode = 0;

//检查删除触发器

checkImmediateTriggersAfterDelete(&req\_struct, regOperPtr, regTabPtr, disk\_page != RNIL);

//DBTUX辅助索引部分不需要立即维护，而是被延迟到提交过程中

sendTUPKEYCONF(signal, &req\_struct, regOperPtr);

return;

}

//对于刷新操作，需要为之前延迟的插入操作执行真正的插入过程（通过调用handleInsertReq），为之前延迟的更新操作执行真正的更新过程（通过调用handleUpdateReq），刷新操作之后不再允许有其它操作

else if (Roptype == ZREFRESH)

{

do\_refresh:

handleRefreshReq(signal, operPtr, fragptr, regTabPtr, &req\_struct, disk\_page != RNIL)

sendTUPKEYCONF(signal, &req\_struct, regOperPtr);

return;

}

}

}

读取：

bool Dbtup::setup\_read(KeyReqStruct \*req\_struct, Operationrec\* regOperPtr, Fragrecord\* regFragPtr, Tablerec\* regTabPtr, bool disk)

{

do {

//取得操作原先的保存点、是否为脏操作，定位操作描述符

Uint32 savepointId= regOperPtr->savepointId;

bool dirty= req\_struct->dirty\_op;

c\_operation\_pool.getPtr(currOpPtr);

bool sameTrans= c\_lqh->is\_same\_trans(currOpPtr.p->userpointer, req\_struct->trans\_id1, req\_struct->trans\_id2);

//对于脏操作，只能读取已经提交的行记录版本

if(dirty && !sameTrans)

savepointId= 0;

else if(sameTrans)

dirty= false;

//沿着行记录的操作链向前遍历，直到找到某个操作的保存点早于给定的保存点，对于脏操作，找不到这样的保存点，保存点参数为0。找到行记录后，也就意味着找到了行记录的undo结构，即找到行了记录的最晚版本

bool found= find\_savepoint(currOpPtr, savepointId);

Uint32 currOp= currOpPtr.p->op\_struct.op\_type;

bool is\_insert = (bits & Tuple\_header::ALLOC);

//对于脏操作或找不到行记录的最晚版本的情况，必须读取已提交的行记录版本，req\_struct->m\_tuple\_ptr的设置已经由setup\_fixed\_part完成，定位到行记录的已提交版本处，此处不需要作任何额外的工作

if(dirty || !found)

{

/\* Read existing committed tuple \*/

}

//其它情况下，表明已经找到了行记录的最晚版本，令req\_struct->m\_tuple\_ptr指向这个最晚版本

else

{

req\_struct->m\_tuple\_ptr= get\_copy\_tuple(&currOpPtr.p->m\_copy\_tuple\_location);

}

//req\_struct->m\_tuple\_ptr此时已经定位到了行记录的有效位置，读操作handleReadReq执行时便从位置读取行记录

if (regTabPtr->need\_expand(disk))

prepare\_read(req\_struct, regTabPtr, disk);

return true;

} while(0);

return false;

}

int Dbtup::handleReadReq(Signal\* signal, Operationrec\* regOperPtr, Tablerec\* regTabPtr, KeyReqStruct\* req\_struct)

{

Uint32 \*dst;

Uint32 dstLen, start\_index;

const BlockReference sendBref= req\_struct->rec\_blockref;

const Uint32 node = refToNode(sendBref);

//定位记录读取后的输出缓冲区的起始地址

if(node != 0 && node != getOwnNodeId()) {

start\_index= 25;

} else {

start\_index= 3;

}

dst= &signal->theData[start\_index];

dstLen= (MAX\_READ / 4) - start\_index;  
 //此前，req\_struct->m\_tuple\_ptr已经定位了行记录的位置，此时将记录内容拷贝到dst指向的输出缓冲区，完成后dst（即返回的消息缓冲区）内已经保存了记录的内容

int ret = readAttributes(req\_struct, &cinBuffer[0], req\_struct->attrinfo\_len, dst, dstLen, false);

Uint32 TnoOfDataRead= (Uint32) ret;

req\_struct->read\_length += TnoOfDataRead;

//将读取的行记录返回DBTC，行记录被封装在GSN\_TRANSID\_AI消息内部，每条记录对应一条消息

sendReadAttrinfo(signal, req\_struct, TnoOfDataRead, regOperPtr);

}

插入：

int Dbtup::handleInsertReq(Signal\* signal, Ptr<Operationrec> regOperPtr, Ptr<Fragrecord> fragPtr, Tablerec\* regTabPtr, KeyReqStruct \*req\_struct, Local\_key \*\* accminupdateptr)

{

Tuple\_header \*dst;

Tuple\_header \*base= req\_struct->m\_tuple\_ptr, \*org= base;

//从DBTUP的undo空闲页中分配一个undo结构，保存将要写入的数据及其状态，包括如下部分：

//|COPY\_TUPLE\_HEADER|ChangeMask|Tuple\_header|行记录存储空间|

//整个undo结构的大小由表定义记录，在建表或改表时有Dbtup::computeTableMetaData计算，可有CREATE/ALTER TABLE使用

//undo空闲页本身是由ndbd的全局内存管理器结构Ndbd\_mem\_manager分配的，该内存分配器在ndbd初始化时调用init\_global\_memory\_manager函数创建，通过Ndbd\_mem\_manager::init分配所有可用的内存，并构建成内存池，该内存管理结构也是一个伙伴系统分配器

dst= alloc\_copy\_tuple(regTabPtr, &regOperPtr.p->m\_copy\_tuple\_location);

//此时dst中保存了一个空闲的undo结构

//插入操作并未使用原先的req\_struct->m\_tuple\_ptr（其中保存了行记录在DBTUP的实际位置，对于插入操作来说，应当不存在），而是直接使用undo结构保存行记录

tuple\_ptr= req\_struct->m\_tuple\_ptr= dst;

//若插入过程为第一项操作，则直接对操作进行初始化

if(mem\_insert)

{

prepare\_initial\_insert(req\_struct, regOperPtr.p, regTabPtr);

}

//若插入操作是跟在一个删除操作之后的（删除+插入相当于一次更新），则仅需要在前次删除操作的undo结构的基础上进行一系列更新即可，但与前次删除操作使用不同的undo结构

else

{

Operationrec\* prevOp= req\_struct->prevOpPtr.p;

tup\_version= prevOp->tupVersion + 1;

if(!prevOp->is\_first\_operation())

org= get\_copy\_tuple(&prevOp->m\_copy\_tuple\_location);

if (regTabPtr->need\_expand())

{

expand\_tuple(req\_struct, sizes, org, regTabPtr, !disk\_insert);

memset(req\_struct->m\_disk\_ptr->m\_null\_bits+

regTabPtr->m\_offsets[DD].m\_null\_offset, 0xFF,

4\*regTabPtr->m\_offsets[DD].m\_null\_words);

Uint32 bm\_size\_in\_bytes= 4\*(regTabPtr->m\_offsets[MM].m\_dyn\_null\_words);

if (bm\_size\_in\_bytes)

{

Uint32\* ptr =

(Uint32\*)req\_struct->m\_var\_data[MM].m\_dyn\_data\_ptr;

bzero(ptr, bm\_size\_in\_bytes);

\* ptr = bm\_size\_in\_bytes >> 2;

}

}

else

{

memcpy(dst, org, 4\*regTabPtr->m\_offsets[MM].m\_fix\_header\_size);

tuple\_ptr->m\_header\_bits |= Tuple\_header::COPY\_TUPLE;

}

memset(tuple\_ptr->m\_null\_bits+

regTabPtr->m\_offsets[MM].m\_null\_offset, 0xFF,

4\*regTabPtr->m\_offsets[MM].m\_null\_words);

}

//若表为磁盘表，需要为表分配undo日志空间

int res;

if (disk\_insert)

{

Logfile\_client lgman(this, c\_lgman, regFragPtr->m\_logfile\_group\_id);

res= lgman.alloc\_log\_space(regOperPtr.p->m\_undo\_buffer\_space);

if(unlikely(res))

{

terrorCode= res;

goto log\_space\_error;

}

}

//由于此前cinBuffer已经保存了插入请求的ATTRINFO，即行记录内容，此处将其内容拷贝到undo结构中

updateAttributes(req\_struct, &cinBuffer[0], req\_struct->attrinfo\_len)

//为新插入的行记录分配真正的存储空间，根据是否包含定长/变长/磁盘列，选择合适的分配函数，但此时不进行真正的插入工作，插入会延迟到提交阶段才会更新到这个新分配的空间

//DBTUP的行记录空间分配：DBTUP有一个空闲页池c\_page\_pool，每次DBTUP需要为记录分配内存时，首先从此处分配内存，若无内存可用，则向全局内存管理器申请一个8K的页， DBTUP的undo空闲页也通过全局内存管理器分配页。

if(mem\_insert)

{

if (!rowid)

{

if (!varalloc)

{

ptr= alloc\_fix\_rec(&terrorCode, regFragPtr, regTabPtr, &regOperPtr.p->m\_tuple\_location, &frag\_page\_id);

}

else

{

regOperPtr.p->m\_tuple\_location.m\_file\_no= sizes[2+MM];

ptr= alloc\_var\_rec(&terrorCode, regFragPtr, regTabPtr, sizes[2+MM], &regOperPtr.p->m\_tuple\_location, &frag\_page\_id);

}

req\_struct->m\_use\_rowid = true;

}

else

{

regOperPtr.p->m\_tuple\_location = req\_struct->m\_row\_id;

if (!varalloc)

{

ptr= alloc\_fix\_rowid(&terrorCode, regFragPtr, regTabPtr, &regOperPtr.p->m\_tuple\_location, &frag\_page\_id);

}

else

{

regOperPtr.p->m\_tuple\_location.m\_file\_no= sizes[2+MM];

ptr= alloc\_var\_rowid(&terrorCode, regFragPtr, regTabPtr, sizes[2+MM], &regOperPtr.p->m\_tuple\_location, &frag\_page\_id);

}

}

real\_page\_id = regOperPtr.p->m\_tuple\_location.m\_page\_no;

update\_acc = true; /\* Will be updated later once success is known \*/

base = (Tuple\_header\*)ptr;

base->m\_operation\_ptr\_i= regOperPtr.i;

base->m\_header\_bits= Tuple\_header::ALLOC |

(sizes[2+MM] > 0 ? Tuple\_header::VAR\_PART : 0);

}

else

{

if (regTabPtr->need\_shrink() && cmp[0] != cmp[1] &&

handle\_size\_change\_after\_update(req\_struct, base, regOperPtr.p, regFragPtr, regTabPtr, sizes)

req\_struct->m\_use\_rowid = false;

base->m\_header\_bits &= ~(Uint32)Tuple\_header::FREE;

}

if (disk\_insert)

{

Local\_key tmp;

Uint32 size= regTabPtr->m\_attributes[DD].m\_no\_of\_varsize == 0 ? 1 : sizes[2+DD];

int ret= disk\_page\_prealloc(signal, fragPtr, &tmp, size);

regOperPtr.p->op\_struct.m\_disk\_preallocated= 1;

tmp.m\_page\_idx= size;

memcpy(tuple\_ptr->get\_disk\_ref\_ptr(regTabPtr), &tmp, sizeof(tmp));

Local\_key ref = regOperPtr.p->m\_tuple\_location;

ref.m\_page\_no = frag\_page\_id;

Tuple\_header\* disk\_ptr= req\_struct->m\_disk\_ptr;

disk\_ptr->m\_header\_bits = 0;

disk\_ptr->m\_base\_record\_ref= ref.ref();

}

//向上层报告此次为操作分片的空间位置，该位置将有DBACC写入索引

if (update\_acc)

{

Local\_key accKey = regOperPtr.p->m\_tuple\_location;

accKey.m\_page\_no = frag\_page\_id;

\*\* accminupdateptr = accKey;

}

else

\* accminupdateptr = 0; // No accminupdate should be performed

//操作的状态变更为TUPLE\_PREPARED

set\_tuple\_state(regOperPtr.p, TUPLE\_PREPARED);

}

更新：

int Dbtup::handleUpdateReq(Signal\* signal, Operationrec\* operPtrP, Fragrecord\* regFragPtr, Tablerec\* regTabPtr, KeyReqStruct\* req\_struct, bool disk)

{

Tuple\_header \*dst;

Tuple\_header \*base= req\_struct->m\_tuple\_ptr, \*org;

//从DBTUP的undo空闲页中分配一个undo结构，保存将要更新的数据及其状态

dst= alloc\_copy\_tuple(regTabPtr, &operPtrP->m\_copy\_tuple\_location)

//若更新操作为第一项操作，则直接对操作进行初始化

if(operPtrP->is\_first\_operation())

{

org= req\_struct->m\_tuple\_ptr;

tup\_version= org->get\_tuple\_version();

clear\_change\_mask\_info(regTabPtr, change\_mask\_ptr);

}

//若更新操作是跟在其他操作之后的，则仅需要在前次操作的undo结构的基础上进行一系列更新即可，但与前次操作使用不同的undo结构

else

{

Operationrec\* prevOp= req\_struct->prevOpPtr.p;

tup\_version= prevOp->tupVersion;

Uint32 \* rawptr = get\_copy\_tuple\_raw(&prevOp->m\_copy\_tuple\_location);

org= get\_copy\_tuple(rawptr);

copy\_change\_mask\_info(regTabPtr, change\_mask\_ptr, get\_change\_mask\_ptr(rawptr));

}

//由于此前cinBuffer已经保存了更新请求的ATTRINFO，即行记录内容，此处将其内容拷贝到undo结构中

//但此时不进行真正的更新工作，更新会延迟到提交阶段才会更新到这个行记录空间

int retValue = updateAttributes(req\_struct, &cinBuffer[0], req\_struct->attrinfo\_len);

//操作的状态变更为TUPLE\_PREPARED

set\_tuple\_state(operPtrP, TUPLE\_PREPARED);

//更新与插入的不同之处在于，更新不需要从DBTUP中为行记录分配空间，而是复用记录原先占据的空间

}

删除：

int Dbtup::handleDeleteReq(Signal\* signal, Operationrec\* regOperPtr, Fragrecord\* regFragPtr, Tablerec\* regTabPtr, KeyReqStruct \*req\_struct, bool disk)

{

//从DBTUP的undo空闲页中分配一个undo结构，保存将要删除的数据及其状态

Tuple\_header\* dst = alloc\_copy\_tuple(regTabPtr, &regOperPtr->m\_copy\_tuple\_location);

//若更新操作为第一项操作，则保存行记录的原始已提交版本

if (regOperPtr->is\_first\_operation())

{

regOperPtr->tupVersion= req\_struct->m\_tuple\_ptr->get\_tuple\_version();

if (regTabPtr->m\_no\_of\_disk\_attributes)

{

dst->m\_header\_bits = req\_struct->m\_tuple\_ptr->m\_header\_bits;

memcpy(dst->get\_disk\_ref\_ptr(regTabPtr),

req\_struct->m\_tuple\_ptr->get\_disk\_ref\_ptr(regTabPtr),

sizeof(Local\_key));

}

}

//若删除操作是跟在其他操作之后的，则保存前次操作生成的undo版本

else

{

Operationrec\* prevOp= req\_struct->prevOpPtr.p;

regOperPtr->tupVersion= prevOp->tupVersion;

// make copy since previous op is committed before this one

const Tuple\_header\* org = get\_copy\_tuple(&prevOp->m\_copy\_tuple\_location);

Uint32 len = regTabPtr->total\_rec\_size -

Uint32(((Uint32\*)dst) - get\_copy\_tuple\_raw(&regOperPtr->m\_copy\_tuple\_location));

memcpy(dst, org, 4 \* len);

req\_struct->m\_tuple\_ptr = dst;

}

//对于磁盘空间表，还需要分配额外的undo日志空间

if(disk && regOperPtr->m\_undo\_buffer\_space == 0)

{

regOperPtr->op\_struct.m\_wait\_log\_buffer = 1;

regOperPtr->op\_struct.m\_load\_diskpage\_on\_commit = 1;

Uint32 sz= regOperPtr->m\_undo\_buffer\_space=

(sizeof(Dbtup::Disk\_undo::Free) >> 2) +

regTabPtr->m\_offsets[DD].m\_fix\_header\_size - 1;

Logfile\_client lgman(this, c\_lgman, regFragPtr->m\_logfile\_group\_id);

terrorCode= lgman.alloc\_log\_space(sz);

}

//操作的状态变更为TUPLE\_PREPARED

set\_tuple\_state(regOperPtr, TUPLE\_PREPARED);

if (req\_struct->attrinfo\_len == 0)

{

return 0;

}

//处理删除且读语义，读出删除的行并返回给DBTC，正常的删除不会包含此项动作，通过handleReadReq读取行记录

Uint32 RlogSize;

int ret= handleReadReq(signal, regOperPtr, regTabPtr, req\_struct);

if (ret == 0 && (RlogSize= req\_struct->log\_size))

sendLogAttrinfo(signal, req\_struct, RlogSize, regOperPtr);

return ret;

}

//触发器仅在主节点上发起，插入/更新/删除操作的触发器最终通过fireImmediateTriggers发射触发器动作，向DBTC发出一条GSN\_FIRE\_TRIG\_ORD请求

//辅助索引维护动作由各个节点的DBTUP发起，插入/更新操作会立即向DBTUX发出一条GSN\_TUX\_MAINT\_REQ请求，进行辅助索引维护，删除操作则延迟到提交阶段向DBTUX发出一条GSN\_TUX\_MAINT\_REQ请求

//提交时只能针对状态变更为TUPLE\_PREPARED的操作进行

## 服务器ndbd节点事务处理对gci的验证DBDIH部分

8.6阶段

//该函数由Dbtc::diverify010Lab直接调用，因此不需要通过消息返回结果

void Dbdih::execDIVERIFYREQ(Signal\* signal)

{

Uint32 qno = signal->theData[1];

DIVERIFY\_queue & q = c\_diverify\_queue[qno];

//取回本地LQH当前gci

signal->theData[1] = (Uint32)(m\_micro\_gcp.m\_current\_gci >> 32);

signal->theData[2] = (Uint32)(m\_micro\_gcp.m\_current\_gci & 0xFFFFFFFF);

signal->theData[3] = 0;

return;

}

## 服务器ndbd节点事务处理提交DBTC部分

8.7阶段

void Dbtc::execDIVERIFYCONF(Signal\* signal)

{

UintR Tgci\_hi = signal->theData[1];

UintR Tgci\_lo = signal->theData[2];

Uint64 Tgci = Tgci\_lo | (Uint64(Tgci\_hi) << 32);

//为API节点连接描述符创建若干个全局检查点描述符GcpRecordPtr，记录已经更新到的全局检查点位置，将根据事务的全局检查点，将其插入合适的事务队列

commitGciHandling(signal, Tgci);

//开始提交过程

//提交时，要求API连接描述符ApiConnectRecord的状态apiConnectState为CS\_PREPARE\_TO\_COMMIT，到达此处意味着事务已经提交，除非在报告COMMIT消息前出现crash。

TcConnectRecord\* const regTcPtr = &localTcConnectRecord[TtcConnectptrIndex];

tcConnectptr.i = TtcConnectptrIndex;

tcConnectptr.p = regTcPtr;

commit020Lab(signal);

}

void Dbtc::commit020Lab(Signal\* signal)

{

TcConnectRecordPtr localTcConnectptr;

ApiConnectRecord \* const regApiPtr = apiConnectptr.p;

UintR TtcConnectFilesize = ctcConnectFilesize;

TcConnectRecord \*localTcConnectRecord = tcConnectRecord;

localTcConnectptr.p = tcConnectptr.p;

setApiConTimer(apiConnectptr.i, ctcTimer, \_\_LINE\_\_);

UintR Tcount = 0;

//一次一个地向各个分片节点的DBLQH发送提交请求，注意，此处使用了ndbd的异步发送机制，放置一次发出太多数据包，导致其他消息得不到处理

do {

localTcConnectptr.i = localTcConnectptr.p->nextTcConnect;

localTcConnectptr.p->tcConnectstate = OS\_COMMITTING;

//向一个DBLQH发出GSN\_COMMIT请求

Tcount += sendCommitLqh(signal, localTcConnectptr.p);

if (localTcConnectptr.i != RNIL) {

signal->theData[0] = TcContinueB::ZSEND\_COMMIT\_LOOP;

signal->theData[1] = apiConnectptr.i;

signal->theData[2] = localTcConnectptr.i;

sendSignal(cownref, GSN\_CONTINUEB, signal, 3, JBB);

return;

} else {

regApiPtr->apiConnectstate = CS\_COMMIT\_SENT;

return;

}//if

} while (1);

}

Uint32 Dbtc::sendCommitLqh(Signal\* signal, TcConnectRecord \* const regTcPtr)

{

//这里又使用了另外一个发送技巧，为了减少发送次数，这里将请求打包，达到一定量时再一并发送给DBLQH，这对一些频繁出现的消息有效，包括ZCOMMIT，ZCOMPLETE，ZCOMMITTED，ZCOMPLETED，ZLQHKEYCONF等，代表GSN\_COMMIT，GSN\_COMPLETE，GSN\_COMMITTED，GSN\_ ZCOMPLETED，GSN\_ZLQHKEYCONF，DBTC和DBLQH都使用了这个技巧

if (Thostptr.p->noOfPackedWordsLqh > 25 - 5) {

jam();

sendPackedSignalLqh(signal, Thostptr.p); //8.8

} else {

jam();

ret = 1;

updatePackedList(signal, Thostptr.p, Thostptr.i);

}

Tdata[0] |= (ZCOMMIT << 28);

UintR Tindex = Thostptr.p->noOfPackedWordsLqh;

UintR\* TDataPtr = &Thostptr.p->packedWordsLqh[Tindex];

memcpy(TDataPtr, &Tdata[0], len << 2);

Thostptr.p->noOfPackedWordsLqh = Tindex + len;

}

注意：

8.8->8.9，8.10->8.11这两组阶段，代表者DBLQH向DBTC确认提交和完成，每一组又包括了两类实际情况，即正常情况和异常情况。

提交与完成阶段的过程类似，均为DBTC向所有参与事务的DBLQH发出GSN\_COMMIT/ GSN\_COMPLETE消息，若任何DBLQH在中途出现问题，则DBTC通过超时等机制发现该问题后，则转而向剩余DBLQH发送GSN\_COMMITREQ/GSN\_COMPLETEREQ，由DBLQH判断是否可以提交/完成。

正常情况下，DBLQH会向DBTC返回GSN\_COMMITTED/GSN\_COMPLETEED（对应GSN\_COMMIT/ GSN\_COMPLETE）。

异常情况下，DBLQH会向DBTC返回GSN\_COMMITCONF/GSN\_COMPLETECONF（对应GSN\_COMMITREQ/GSN\_COMPLETEREQ）。

这里仅分析正确情况下的流程，即DBLQH会向DBTC返回GSN\_COMMITTED/GSN\_COMPLETEED。

8.9阶段

void Dbtc::execCOMMITTED(Signal\* signal)

{

//提交后首先向API节点返回GSN\_TCKEYCONF

apiConnectptr = localApiConnectptr;

localCopyPtr = sendApiCommit(signal);

//提交后进入完成阶段，这个阶段主要是向各个DBLQH发送GSN\_COMPLETE消息，通知其释放资源

complete010Lab(signal);

}

8.11阶段

void Dbtc::execCOMPLETED(Signal\* signal)

{

//释放事务的各项资源，包括：

//TcConnectRecord事务描述符

//GcpRecord检查点描述符

//TcFiredTriggerData触发器数据

//TcIndexOperation索引操作

releaseTransResources(signal);

}

## 服务器ndbd节点事务本地事务提交DBLQH部分

提交过程为DBTC先向所有从节点发送GSN\_COMMIT，最后向主节点发送GSN\_COMMIT，从节点仅更改本地事务状态，主节点进行数据提交并生效，主节点向DBTC发送GSN\_COMMITTED

8.8阶段

void Dblqh::execCOMMIT(Signal\* signal)

{

//获取gci和事务id

TcConnectionrec \*regTcConnectionrec = tcConnectionrec;

Uint32 ttcConnectrecFileSize = ctcConnectrecFileSize;

Uint32 tcIndex = signal->theData[0];

Uint32 gci\_hi = signal->theData[1];

Uint32 transid1 = signal->theData[2];

Uint32 transid2 = signal->theData[3];

Uint32 gci\_lo = signal->theData[4];

//在当前节点的DBLQH进行提交

commitReqLab(signal, gci\_hi, gci\_lo);

}

void Dblqh::commitReqLab(Signal\* signal, Uint32 gci\_hi, Uint32 gci\_lo)

{

//对于已已经prepared的事务，向DBTC提交其写日志

Uint32 blockNo = refToMain(regTcPtr->tcTupBlockref);

signal->theData[0] = regTcPtr->tupConnectrec;

signal->theData[1] = gci\_hi;

signal->theData[2] = gci\_lo;

EXECUTE\_DIRECT(blockNo, GSN\_TUP\_WRITELOG\_REQ, signal, 3); //8.8.1

//主节点本地提交，使得修改生效

if (regTcPtr->seqNoReplica == 0 || regTcPtr->activeCreat == Fragrecord::AC\_NR\_COPY)

{

localCommitLab(signal); //8.8.3

return;

}//if

//从节点更改事务状态

commitReplyLab(signal); //8.8.2

}

阶段8.8.1

void Dbtup::execTUP\_WRITELOG\_REQ(Signal\* signal)

{

//写入所有操作的日志

while (loopOpPtr.p->prevActiveOp != RNIL) {

jam();

loopOpPtr.i= loopOpPtr.p->prevActiveOp;

c\_operation\_pool.getPtr(loopOpPtr);

}

do {

ndbrequire(get\_trans\_state(loopOpPtr.p) == TRANS\_STARTED);

signal->theData[0] = loopOpPtr.p->userpointer;

signal->theData[1] = gci\_hi;

signal->theData[2] = gci\_lo;

if (loopOpPtr.p->nextActiveOp == RNIL)

{

EXECUTE\_DIRECT(DBLQH, GSN\_LQH\_WRITELOG\_REQ, signal, 3); //8.8.1.1

return;

}

EXECUTE\_DIRECT(DBLQH, GSN\_LQH\_WRITELOG\_REQ, signal, 3);

loopOpPtr.i= loopOpPtr.p->nextActiveOp;

c\_operation\_pool.getPtr(loopOpPtr);

} while (true);

}

阶段8.8.1.1

void Dblqh::execLQH\_WRITELOG\_REQ(Signal\* signal)

{

//完成实际的写入过程，写入日志页

writeCommitLog(signal, regLogPartPtr);

}

从节点提交

阶段8.8.2

void Dblqh::commitReplyLab(Signal\* signal)

{

//从节点事务状态变更为TcConnectionrec::COMMITTED

regTcPtr->transactionState = TcConnectionrec::COMMITTED;

//主节点向DBTC返回GSN\_COMMITTED通知提交过程全部完成

if (regTcPtr->seqNoReplica == 0)

{

sendCommittedTc(signal, clientBlockref);

return;

} else {

sendCommitLqh(signal, clientBlockref);

return;

}//if

}

主节点提交

阶段8.8.3

void Dblqh::localCommitLab(Signal\* signal)

{

commitContinueAfterBlockedLab(signal);

}

void Dblqh::commitContinueAfterBlockedLab(Signal\* signal)

{

if (operation != ZREAD)

{

//写操作需要向DBTUP和DBACC提交，使操作生效，首先通知DBTUP提交，然后通知DBACC提交，按照这个顺序的原因是放置DBTUP的中间状态被提前生效，其它事务出现脏读

TupCommitReq \* const tupCommitReq = (TupCommitReq \*)signal->getDataPtrSend();

//首先通知DBTUP提交

Uint32 sig0 = regTcPtr.p->tupConnectrec;

Uint32 tup = refToMain(regTcPtr.p->tcTupBlockref);

tupCommitReq->opPtr = sig0;

tupCommitReq->gci\_hi = regTcPtr.p->gci\_hi;

tupCommitReq->hashValue = regTcPtr.p->hashValue;

tupCommitReq->diskpage = RNIL;

tupCommitReq->gci\_lo = regTcPtr.p->gci\_lo;

tupCommitReq->transId1 = regTcPtr.p->transid[0];

tupCommitReq->transId2 = regTcPtr.p->transid[1];

EXECUTE\_DIRECT(tup, GSN\_TUP\_COMMITREQ, signal,

TupCommitReq::SignalLength); //8.8.3.1

//然后通知DBACC提交

Uint32 acc = refToMain(regTcPtr.p->tcAccBlockref);

signal->theData[0] = regTcPtr.p->accConnectrec;

EXECUTE\_DIRECT(acc, GSN\_ACC\_COMMITREQ, signal, 1); //8.8.3.2

} else {

//读操作进行向DBACC提交，释放读锁

Uint32 acc = refToMain(regTcPtr.p->tcAccBlockref);

signal->theData[0] = regTcPtr.p->accConnectrec;

EXECUTE\_DIRECT(acc, GSN\_ACC\_COMMITREQ, signal, 1);

}

}

//向DBTC返回GSN\_COMMITED

tupcommit\_conf(signal, regTcPtr.p, regFragptr.p);

}

void Dblqh::tupcommit\_conf(Signal\* signal, TcConnectionrec \* tcPtrP, Fragrecord \* regFragptr)

{

//此处进入commitReplyLab将直接调用sendCommittedTc向DBTC返回GSN\_COMMITTED

if (seqNoReplica == 0 || activeCreat == Fragrecord::AC\_NR\_COPY)

{

commitReplyLab(signal);

return;

}

}

8.8.3.1阶段

void Dbtup::execTUP\_COMMITREQ(Signal\* signal)

{

//取回提交请求结构

TupCommitReq \* const tupCommitReq= (TupCommitReq \*)signal->getDataPtr();

//取回事务在DBTUP保存的操作undo结构

Tuple\_header\* tuple\_ptr= (Tuple\_header\*)

get\_ptr(&page, &regOperPtr.p->m\_tuple\_location, regTabPtr.p);

//此处处理索引相关的维护操作，由于索引维护是以内部触发器方式向DBTUX发出请求的，此处需要执行这些DBTUX相关的请求，完成实际的索引维护

if (!regTabPtr.p->tuxCustomTriggers.isEmpty())

{

if(get\_tuple\_state(regOperPtr.p) == TUPLE\_PREPARED)

{

OperationrecPtr loopPtr = regOperPtr;

goto first;

while(loopPtr.i != RNIL)

{

c\_operation\_pool.getPtr(loopPtr);

executeTuxCommitTriggers(signal, loopPtr.p, regFragPtr.p, regTabPtr.p);

set\_tuple\_state(loopPtr.p, TUPLE\_TO\_BE\_COMMITTED);

loopPtr.i = loopPtr.p->nextActiveOp;

}

}

}

//若行记录有磁盘存储部分，还需要进行磁盘存储部分的维护，将记录写入磁盘页

req\_struct.m\_tuple\_ptr = tuple\_ptr;

Uint32 nextOp = regOperPtr.p->nextActiveOp;

Uint32 prevOp = regOperPtr.p->prevActiveOp;

//断开当前操作与其前后操作的关系，开始提交

regOperPtr.p->nextActiveOp = RNIL;

regOperPtr.p->prevActiveOp = RNIL;

//提交过程

tuple\_ptr->m\_operation\_ptr\_i = RNIL;

//对于删除操作，直接释放其原先的行记录，作为下一个可供分配的行记录即可

if (regOperPtr.p->op\_struct.op\_type == ZDELETE)

dealloc\_tuple(signal, gci\_hi, gci\_lo, page.p, tuple\_ptr,

&req\_struct, regOperPtr.p, regFragPtr.p, regTabPtr.p);

//对于插入/更新/写操作的提交

else if(regOperPtr.p->op\_struct.op\_type != ZREFRESH)

commit\_operation(signal, gci\_hi, gci\_lo, tuple\_ptr, page,

regOperPtr.p, regFragPtr.p, regTabPtr.p);

else

//对于刷新操作的提交

commit\_refresh(signal, gci\_hi, gci\_lo, tuple\_ptr, page,

&req\_struct, regOperPtr.p, regFragPtr.p, regTabPtr.p);

//提交完成，可以重新关联当前操作的前后节点，但当前操作已经失效

if (nextOp != RNIL)

c\_operation\_pool.getPtr(nextOp)->prevActiveOp = prevOp;

if (prevOp != RNIL)

c\_operation\_pool.getPtr(prevOp)->nextActiveOp = nextOp;

//对于更新/删除/写操作，在数据操作阶段分配的undo结构，此时需要释放

//undo结构所在的页维持了一个引用计数，每通过alloc\_copy\_tuple分配一条记录，引用计数加1，每通过free\_copy\_tuple释放一条记录，引用计数减1，引用计数为0时，即可释放undo页

if(!regOperPtr.p->m\_copy\_tuple\_location.isNull())

c\_undo\_buffer.free\_copy\_tuple(&regOperPtr.p->m\_copy\_tuple\_location);

//释放之前分配的操作结构

initOpConnection(regOperPtr.p);

signal->theData[0] = 0;

}

void Dbtup::commit\_operation(Signal\* signal, Uint32 gci\_hi, Uint32 gci\_lo, Tuple\_header\* tuple\_ptr, PagePtr pagePtr, Operationrec\* regOperPtr, Fragrecord\* regFragPtr, Tablerec\* regTabPtr)

{

//行记录的位置由参数tuple\_ptr指定，在调用者Dbtup::execTUP\_COMMITREQ处如下设置：

//Tuple\_header\* tuple\_ptr= (Tuple\_header\*)

// get\_ptr(&page, &regOperPtr.p->m\_tuple\_location, regTabPtr.p);

//对于插入操作，此前已经分配了行记录空间，此时只需要取回即可

//对于更新操作，则复用曾经分配的行记录空间

//取得行记录的最新版本

Tuple\_header \*copy= get\_copy\_tuple(&regOperPtr->m\_copy\_tuple\_location);

//若记录仅包含定长列，则直接拷贝即可

if((mm\_vars+mm\_dyns) == 0)

{

memcpy(tuple\_ptr, copy, 4\*fixsize);

disk\_ptr= (Tuple\_header\*)(((Uint32\*)copy)+fixsize);

}

//若记录还包含变长列，则还要取回变长列的存储位置后进行拷贝

else

{

Var\_part\_ref \*ref= tuple\_ptr->get\_var\_part\_ref\_ptr(regTabPtr);

ref->copyout(&tmp);

memcpy(tuple\_ptr, copy, 4\*fixsize);

if (copy\_bits & Tuple\_header::VAR\_PART)

{

Uint32 \*dst= get\_ptr(&vpagePtr, \*ref);

Var\_page\* vpagePtrP = (Var\_page\*)vpagePtr.p;

Varpart\_copy\*vp =(Varpart\_copy\*)copy->get\_end\_of\_fix\_part\_ptr(regTabPtr);

Uint32 len = vp->m\_len;

memcpy(dst, vp->m\_data, 4\*len);

disk\_ptr = (Tuple\_header\*)(vp->m\_data + len);

}

else

{

ndbassert(tmp.m\_page\_no == RNIL);

disk\_ptr = (Tuple\_header\*)copy->get\_end\_of\_fix\_part\_ptr(regTabPtr);

}

}

//若行记录还包含磁盘存储部分，则需要进行磁盘存储部分的更新，包括磁盘行记录和undo日志

if (regTabPtr->m\_no\_of\_disk\_attributes && (copy\_bits & Tuple\_header::DISK\_INLINE))

{

if(copy\_bits & Tuple\_header::DISK\_ALLOC)

disk\_page\_alloc(signal, regTabPtr, regFragPtr, &key, diskPagePtr, gci\_hi);

if(regTabPtr->m\_attributes[DD].m\_no\_of\_varsize == 0)

{

sz= regTabPtr->m\_offsets[DD].m\_fix\_header\_size;

dst= ((Fix\_page\*)diskPagePtr.p)->get\_ptr(key.m\_page\_idx, sz);

}

else

{

dst= ((Var\_page\*)diskPagePtr.p)->get\_ptr(key.m\_page\_idx);

sz= ((Var\_page\*)diskPagePtr.p)->get\_entry\_len(key.m\_page\_idx);

}

if(! (copy\_bits & Tuple\_header::DISK\_ALLOC))

disk\_page\_undo\_update(diskPagePtr.p, &key, dst, sz, gci\_hi, logfile\_group\_id);

memcpy(dst, disk\_ptr, 4\*sz);

memcpy(tuple\_ptr->get\_disk\_ref\_ptr(regTabPtr), &key, sizeof(Local\_key));

}

//写入行的隐藏字段gci

if (regTabPtr->m\_bits & Tablerec::TR\_RowGCI && update\_gci\_at\_commit)

{

\* tuple\_ptr->get\_mm\_gci(regTabPtr) = gci\_hi;

if (regTabPtr->m\_bits & Tablerec::TR\_ExtraRowGCIBits)

{

Uint32 attrId = regTabPtr->getExtraAttrId<Tablerec::TR\_ExtraRowGCIBits>();

store\_extra\_row\_bits(attrId, regTabPtr, tuple\_ptr, gci\_lo,

}

}

//计算行的校验和

if (regTabPtr->m\_bits & Tablerec::TR\_Checksum)

setChecksum(tuple\_ptr, regTabPtr);

}

Void Dbtup::commit\_refresh(Signal\* signal, Uint32 gci\_hi, Uint32 gci\_lo, Tuple\_header\* tuple\_ptr, PagePtr pagePtr, KeyReqStruct \* req\_struct, Operationrec\* regOperPtr, Fragrecord\* regFragPtr, Tablerec\* regTabPtr)

{

//刷新操作分为两类

switch(regOperPtr->m\_copy\_tuple\_location.m\_file\_no){

case Operationrec::RF\_SINGLE\_NOT\_EXIST:

case Operationrec::RF\_MULTI\_NOT\_EXIST:

break;

//一类为刷新一个已存在的行，类似于更新，通过commit\_operation完成

case Operationrec::RF\_SINGLE\_EXIST:

case Operationrec::RF\_MULTI\_EXIST:

commit\_operation(signal, gci\_hi, gci\_lo, tuple\_ptr, pagePtr,

regOperPtr, regFragPtr, regTabPtr);

return;

default:

ndbrequire(false);

}

//另一类为刷新一个不存在的行，类似于插入的逆操作，但不同于删除，其动作包括两个部分：1.从DBACC中删除索引，通过accremoverow完成；2.类似于删除，删除本地行记录，通过dealloc\_tuple完成

Local\_key key = regOperPtr->m\_tuple\_location;

key.m\_page\_no = pagePtr.p->frag\_page\_id;

c\_lqh->accremoverow(signal, regOperPtr->userpointer, &key);

dealloc\_tuple(signal, gci\_hi, gci\_lo, pagePtr.p, tuple\_ptr,

req\_struct, regOperPtr, regFragPtr, regTabPtr);

}

8.8.3.2阶段

void Dbacc::execACC\_COMMITREQ(Signal\* signal)

{

//实际的提交工作

commitOperation(signal);

//对于删除操作，需要检查是否需要缩减内存空闲页规模

//对于插入操作，需要检查释放需要扩大内存空闲页规模

if((Toperation != ZREAD) && (Toperation != ZSCAN\_OP))

{

fragrecptr.p->m\_commit\_count++;

if (Toperation != ZINSERT) {

if (Toperation != ZDELETE) {

return;

} else {

fragrecptr.p->noOfElements--;

fragrecptr.p->slack += fragrecptr.p->elementLength;

if (fragrecptr.p->slack > fragrecptr.p->slackCheck) {

if (fragrecptr.p->expandCounter > 0) {

if (fragrecptr.p->expandFlag < 2) {

sendSignal(cownBlockref, GSN\_SHRINKCHECK2, signal, 4, JBB);

}//if

}//if

}//if

}//if

} else {

fragrecptr.p->noOfElements++;

fragrecptr.p->slack -= fragrecptr.p->elementLength;

if (fragrecptr.p->slack >= (1u << 31)) {

if (fragrecptr.p->expandFlag == 0) {

sendSignal(cownBlockref, GSN\_EXPANDCHECK2, signal, 3, JBB);

}//if

}//if

}

}

}

void Dbacc::commitOperation(Signal\* signal)

{

//考虑操作持有锁的情形

if (opbits & Operationrec::OP\_LOCK\_OWNER)

{

takeOutLockOwnersList(signal, operationRecPtr);

opbits &= ~(Uint32)Operationrec::OP\_LOCK\_OWNER;

operationRecPtr.p->m\_op\_bits = opbits;

const bool queue = (operationRecPtr.p->nextParallelQue != RNIL ||

operationRecPtr.p->nextSerialQue != RNIL);

//普通的提交路径，锁持有者不在锁队列中，操作也非删除操作，此时直接释放锁

if (!queue && (opbits & Operationrec::OP\_ELEMENT\_DISAPPEARED) == 0)

{

Page8Ptr coPageidptr;

Uint32 tcoElementptr;

Uint32 tmp2Olq;

coPageidptr.i = operationRecPtr.p->elementPage;

tcoElementptr = operationRecPtr.p->elementPointer;

tmp2Olq = ElementHeader::setUnlocked(operationRecPtr.p->hashvaluePart,

operationRecPtr.p->scanBits);

ptrCheckGuard(coPageidptr, cpagesize, page8);

coPageidptr.p->word32[tcoElementptr] = tmp2Olq;

return;

}

//操作持有锁，同时又其它操作在等待该锁，此时将锁转交给下一个等待的操作

else if (queue)

{

lockowner(signal, operationRecPtr, true);

return;

}

//此次为删除操作，后续无等待锁的操作，进行真正的索引删除过程

else

{

commitdelete(signal);

return;

}//if

}

…

}

void Dbacc::commitdelete(Signal\* signal)

{

//索引记录叶节点是一个开放定址的哈希表，若索引记录非最后一条记录，则将最后一条记录覆盖到要删除的位置，并更新undo日志，若索引记录为最后一条记录，则移除即可。

deleteElement(signal);

}

## 服务器ndbd节点事务本地事务完成DBLQH部分

完成过程为DBTC先向所有从节点发送GSN\_COMPLETE，最后向主节点发送GSN\_COMPLETE，主从节点均释放本地资源，主节点向DBTC发送GSN\_COMPLETED

8.10阶段

void Dblqh::execCOMPLETE(Signal\* signal)

{

if ((tcConnectptr.p->transactionState == TcConnectionrec::COMMITTED) &&

(tcConnectptr.p->transid[0] == transid1) &&

(tcConnectptr.p->transid[1] == transid2)) {

TcConnectionrec \* const regTcPtr = tcConnectptr.p;

//从节点在此处进行数据提交，并在本地生效

if (tcConnectptr.p->seqNoReplica != 0 &&

tcConnectptr.p->activeCreat == Fragrecord::AC\_NORMAL)

{

localCommitLab(signal);

return;

}

//主节点在此处向DBTC返回GSN\_COMPLETED

else if (tcConnectptr.p->seqNoReplica == 0)

{

completeTransLastLab(signal);

return;

}

else

{

completeTransNotLastLab(signal);

return;

}

}//if

}

void Dblqh::completeTransLastLab(Signal\* signal)

{

//向DBTC返回GSN\_COMPLETED

sendCompletedTc(signal, clientBlockref);

//释放事务在DBLQH处相关的资源：

//TcConnectionrec结构

//事务id

//DBTUP处分配的操作结构

cleanUp(signal);

}

## 服务器ndbd节点事务处理中止DBLQH部分

中止（abort）过程并未体现在主路径内，此处仅为介绍

中止消息到达各个DBLQH的过程为链式中止，与操作过程相同

void Dblqh::execABORT(Signal\* signal)

{

//取回事务描述符

findTransaction(transid1, transid2, tcOprec, 0)

if (regTcPtr->nextReplica != ZNIL)

sendSignal(TLqhRef, GSN\_ABORT, signal, 4, JBB);

//部署其它中止操作

abortStateHandlerLab(signal);

}

void Dblqh::abortStateHandlerLab(Signal\* signal)

{

abortCommonLab(signal);

}

void Dblqh::abortCommonLab(Signal\* signal)

{

abortContinueAfterBlockedLab(signal);

}

void Dblqh::abortContinueAfterBlockedLab(Signal\* signal)

{

//首先通知DBACC中止

EXECUTE\_DIRECT(DBACC, GSN\_ACC\_ABORTREQ, signal, 2);

//然后通知DBTUP中止

execACC\_ABORTCONF(signal);

}

void Dbacc::execACC\_ABORTREQ(Signal\* signal)

{

abortOperation(signal);

}

void Dbacc::abortOperation(Signal\* signal)

{

//仅考虑锁持有者的中止过程

if (opbits & Operationrec::OP\_LOCK\_OWNER)

{

//若有其它操作等待锁，此时需要将锁转交给等待的操作

if (queue)

{

release\_lockowner(signal, operationRecPtr, false);

}

else

{

//若没有其它操作等待锁，此时仅需释放锁即可

if ((opbits & Operationrec::OP\_ELEMENT\_DISAPPEARED) == 0)

{

Page8Ptr aboPageidptr;

Uint32 taboElementptr;

Uint32 tmp2Olq;

taboElementptr = operationRecPtr.p->elementPointer;

aboPageidptr.i = operationRecPtr.p->elementPage;

tmp2Olq = ElementHeader::setUnlocked(operationRecPtr.p->hashvaluePart,

operationRecPtr.p->scanBits);

ptrCheckGuard(aboPageidptr, cpagesize, page8);

dbgWord32(aboPageidptr, taboElementptr, tmp2Olq);

arrGuard(taboElementptr, 2048);

aboPageidptr.p->word32[taboElementptr] = tmp2Olq;

return;

}

//对于插入操作的中止，需要删除原先分配的索引行记录

else

{

commitdelete(signal);

}//if

}

}

…

}

void Dblqh::execACC\_ABORTCONF(Signal\* signal)

{

//通知DBTUP仅需中止

EXECUTE\_DIRECT(DBTUP, GSN\_TUP\_ABORTREQ, signal, 1);

continueAbortLab(signal);

}

void Dbtup::execTUP\_ABORTREQ(Signal\* signal)

{

do\_tup\_abortreq(signal, 0);

}

void Dbtup::do\_tup\_abortreq(Signal\* signal, Uint32 flags)

{

//首先仅需辅助索引的中止操作

if (!regTabPtr.p->tuxCustomTriggers.isEmpty() &&! (flags & ZSKIP\_TUX\_TRIGGERS))

{

executeTuxAbortTriggers(signal, regOperPtr.p, regFragPtr.p, regTabPtr.p);

OperationrecPtr loopOpPtr;

loopOpPtr.i = regOperPtr.p->nextActiveOp;

while (loopOpPtr.i != RNIL)

{

c\_operation\_pool.getPtr(loopOpPtr);

if (get\_tuple\_state(loopOpPtr.p) != TUPLE\_ALREADY\_ABORTED)

executeTuxAbortTriggers(signal, loopOpPtr.p, regFragPtr.p, regTabPtr.p);

loopOpPtr.i = loopOpPtr.p->nextActiveOp;

}

}

//中止所有的数据变动

bool change = do\_tup\_abort\_operation(signal, tuple\_ptr, regOperPtr.p, regFragPtr.p, regTabPtr.p);

OperationrecPtr loopOpPtr;

loopOpPtr.i = regOperPtr.p->nextActiveOp;

while (loopOpPtr.i != RNIL)

{

c\_operation\_pool.getPtr(loopOpPtr);

if (get\_tuple\_state(loopOpPtr.p) != TUPLE\_ALREADY\_ABORTED)

{

change |= do\_tup\_abort\_operation(signal, tuple\_ptr, loopOpPtr.p, regFragPtr.p, regTabPtr.p);

set\_tuple\_state(loopOpPtr.p, TUPLE\_ALREADY\_ABORTED);

}

loopOpPtr.i = loopOpPtr.p->nextActiveOp;

}

//重设行记录的校验和

if (change && (regTabPtr.p->m\_bits & Tablerec::TR\_Checksum))

setChecksum(tuple\_ptr, regTabPtr.p);

}

Bool Dbtup::do\_tup\_abort\_operation(Signal\* signal, Tuple\_header \*tuple\_ptr, Operationrec\* opPtrP, Fragrecord\* fragPtrP, Tablerec\* tablePtrP)

{

Uint32 bits= tuple\_ptr->m\_header\_bits;

//对于插入/更新/写操作需要undo磁盘列的变更

if (opPtrP->op\_struct.op\_type != ZDELETE)

{

Tuple\_header \*copy= get\_copy\_tuple(&opPtrP->m\_copy\_tuple\_location);

if (opPtrP->op\_struct.m\_disk\_preallocated)

{

Local\_key key;

memcpy(&key, copy->get\_disk\_ref\_ptr(tablePtrP), sizeof(key));

disk\_page\_abort\_prealloc(signal, fragPtrP, &key, key.m\_page\_idx);

}

if(! (bits & Tuple\_header::ALLOC))

{

//对于更新操作，若此前在操作时扩大了行记录空间，此处需要缩减回去，同时需要释放undo结构

if(bits & Tuple\_header::MM\_GROWN)

{

Ptr<Page> vpage;

Uint32 idx= opPtrP->m\_tuple\_location.m\_page\_idx;

Uint32 \*var\_part;

Var\_part\_ref \*ref = tuple\_ptr->get\_var\_part\_ref\_ptr(tablePtrP);

Local\_key tmp;

ref->copyout(&tmp);

idx= tmp.m\_page\_idx;

var\_part= get\_ptr(&vpage, \*ref);

Var\_page\* pageP = (Var\_page\*)vpage.p;

Uint32 len= pageP->get\_entry\_len(idx) & ~Var\_page::CHAIN;

Uint32 sz= var\_part[len-1];

if (sz)

{

pageP->shrink\_entry(idx, sz);

update\_free\_page\_list(fragPtrP, vpage);

}

else

{

free\_var\_part(fragPtrP, vpage, tmp.m\_page\_idx);

tmp.m\_page\_no = RNIL;

ref->assign(&tmp);

bits &= ~(Uint32)Tuple\_header::VAR\_PART;

}

tuple\_ptr->m\_header\_bits= bits & ~Tuple\_header::MM\_GROWN;

change = true;

}

}

//对于插入操作，若此前DBTUP为行记录分配了行记录空间，此时需要释放

else if (opPtrP->is\_first\_operation())

{

change = true;

tuple\_ptr->m\_header\_bits &= ~(Uint32)Tuple\_header::ALLOC;

tuple\_ptr->m\_header\_bits |= Tuple\_header::FREED;

}

}

//对于删除操作，不需要额外的步骤，仅需释放undo结构

else if (opPtrP->is\_first\_operation())

{

if (bits & Tuple\_header::ALLOC)

{

change = true;

tuple\_ptr->m\_header\_bits &= ~(Uint32)Tuple\_header::ALLOC;

tuple\_ptr->m\_header\_bits |= Tuple\_header::FREED;

}

}

return change;

}

# ndbd节点各BLOCK启动流程

进入虚拟机前，ndbd\_run将CMVMI::GSN\_START\_ORD信号发送入JBA队列，若启动时带有nostart选项，则发送一次，否则连续发送二次：

ndbd\_run

…

switch(globalData.theRestartFlag){

case initial\_state: //启动时带有nostart选项

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_CMVMI);

break;

case perform\_start: //启动时没有nostart选项，默认

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_CMVMI);

globalEmulatorData.theThreadConfig->doStart(NodeState::SL\_STARTING);

break;

default:

assert("Illegal state globalData.theRestartFlag" == 0);

}

…

启动虚拟机后，将处理这几个消息，进入BLOCK启动流程，启动阶段与文档启动阶段相同。

## 初始化阶段

CMVMI收到第一条GSN\_START\_ORD后：

Cmvmi::execSTART\_ORD

if(globalData.theStartLevel == NodeState::SL\_NOTHING)

{

for(unsigned int i = 1; i < MAX\_NODES; i++ )

{

//连接所有mgmd节点

if (getNodeInfo(i).m\_type == NodeInfo::MGM)

{

globalTransporterRegistry.do\_connect(i);

}

}

globalData.theStartLevel = NodeState::SL\_CMVMI;

//向QMGR的JBA也发送一条GSN\_START\_ORD消息

sendSignal(QMGR\_REF, GSN\_START\_ORD, signal, 1, JBA);

return ;

}

CMVMI收到第二条GSN\_START\_ORD后：

if(globalData.theStartLevel == NodeState::SL\_CMVMI)

{

globalData.theStartLevel = NodeState::SL\_STARTING;

globalData.theRestartFlag = system\_started;

for(unsigned int i = 1; i < MAX\_NODES; i++ )

{

//与其他非mgmd节点断开连接

if (i != getOwnNodeId() && getNodeInfo(i).m\_type != NodeInfo::MGM)

{

globalTransporterRegistry.do\_disconnect(i);

globalTransporterRegistry.setIOState(i, HaltIO);

}

}

//向DBCNTR的JBA发送GSN\_START\_ORD消息，开始正式的启动过程

sendSignal(NDBCNTR\_REF, GSN\_START\_ORD, signal, 1, JBA);

return;

}

QMGR在收到GSN\_START\_ORD后：

Qmgr::execSTART\_ORD

{

Uint64 now = NdbTick\_CurrentMillisecond();

signal->theData[0] = ZTIMER\_HANDLING;

signal->theData[1] = Uint32(now >> 32);

signal->theData[2] = Uint32(now);

//向QMGR的JBB发送GSN\_CONTINUEB消息，内容为ZTIMER\_HANDLING，触发计时器

sendSignal(QMGR\_REF, GSN\_CONTINUEB, signal, 3, JBB);

}

QMGR在收到GSN\_CONTINUEB后：

Qmgr::execCONTINUEB

case ZTIMER\_HANDLING:

timerHandlingLab(signal);

Qmgr::timerHandlingLab(Signal\* signal)

…

signal->theData[0] = ZTIMER\_HANDLING;

signal->theData[1] = Uint32(TcurrentTime >> 32);

signal->theData[2] = Uint32(TcurrentTime);

//当前各项初始化还未完成，仅仅定期启动一个10ms的计时器

sendSignalWithDelay(QMGR\_REF, GSN\_CONTINUEB, signal, 10, 3);

DBCNTR在收到GSN\_START\_ORD：

Ndbcntr::execSTART\_ORD

c\_missra.execSTART\_ORD(signal);

Ndbcntr::Missra::execSTART\_ORD

signal->theData[0] = NDB\_LE\_NDBStartStarted;

signal->theData[1] = NDB\_VERSION;

signal->theData[2] = NDB\_MYSQL\_VERSION\_D;

//向CMVMI的JBB发送GSN\_EVENT\_REP消息，内容为NDB\_LE\_NDBStartStarted

cntr.sendSignal(CMVMI\_REF, GSN\_EVENT\_REP, signal, 3, JBB);

currentBlockIndex = 0;

sendNextREAD\_CONFIG\_REQ(signal);

Ndbcntr::Missra::sendNextREAD\_CONFIG\_REQ

…

ReadConfigReq \* req = (ReadConfigReq\*)signal->getDataPtrSend();

req->senderData = 0;

req->senderRef = cntr.reference();

req->noOfParameters = 0;

const BlockReference ref = readConfigOrder[currentBlockIndex];

//1ms后向readConfigOrder[currentBlockIndex]发送一个GSN\_READ\_CONFIG\_REQ，进行配置读取过程，这个配置读取过程会按顺序遍历readConfigOrder内的每一个BLOCK，给每一个BLOCK读取配置的机会

cntr.sendSignalWithDelay(ref, GSN\_READ\_CONFIG\_REQ, signal,

1, ReadConfigReq::SignalLength);

…

BLOCK在收到GSN\_READ\_CONFIG\_REQ后，均会通过BLOCK::execREAD\_CONFIG\_REQ函数进行配置读取，配置读取一般通过向mgmd请求获得，然后设置BLOCK本身的各项参数，并向DBCNTR的JBB响应一个GSN\_READ\_CONFIG\_CONF消息：

sendSignal(ref, GSN\_READ\_CONFIG\_CONF, signal, ReadConfigConf::SignalLength, JBB);

而DBCNTR在收到了一个GSN\_READ\_CONFIG\_CONF后，进行如下工作：

Ndbcntr::execREAD\_CONFIG\_CONF

c\_missra.execREAD\_CONFIG\_CONF(signal);

Ndbcntr::Missra::execREAD\_CONFIG\_CONF

//增加currentBlockIndex，重新进入Ndbcntr::Missra::sendNextREAD\_CONFIG\_REQ，通知下一个BLOCK读取配置

currentBlockIndex++;

sendNextREAD\_CONFIG\_REQ(signal);

当Ndbcntr::Missra::sendNextREAD\_CONFIG\_REQ收到所有BLOCK的确认后，进行如下工作：

Ndbcntr::Missra::execSTART\_ORD

currentStartPhase = 0;

for(Uint32 i = 0; i<ALL\_BLOCKS\_SZ; i++){

if(ALL\_BLOCKS[i].NextSP < currentStartPhase)

currentStartPhase = ALL\_BLOCKS[i].NextSP;

}

//进入STTOR启动阶段

currentBlockIndex = 0;

sendNextSTTOR(signal);

CMVMI在收到GSN\_EVENT\_REP后：

Cmvmi::execEVENT\_REP

//该函数主要作为一个消息转发接口，用于向其它订阅者BLOCK广播该消息，同时在日志中记录该消息

//向其它订阅者BLOCK广播该消息

SubscriberPtr ptr;

for(subscribers.first(ptr); ptr.i != RNIL; subscribers.next(ptr))

{

if(ptr.p->logLevel.getLogLevel(eventCategory) < threshold)

continue;

sendSignal(ptr.p->blockRef, GSN\_EVENT\_REP, signal, signal->length(), JBB);

}

//在日志中记录该消息

if(clogLevel.getLogLevel(eventCategory) < threshold)

return;

g\_eventLogger->log(eventReport->getEventType(), signal->theData, signal->getLength(), 0, 0);

1.5.各个BLOCK的配置读取及初始化顺序：

BLOCK顺序：

CMVMI\_REF, NDBFS\_REF, DBINFO\_REF, DBTUP\_REF, DBACC\_REF, DBTC\_REF, DBLQH\_REF, DBTUX\_REF, DBDICT\_REF, DBDIH\_REF, NDBCNTR\_REF, QMGR\_REF, TRIX\_REF, BACKUP\_REF, DBUTIL\_REF, SUMA\_REF, TSMAN\_REF, LGMAN\_REF, PGMAN\_REF, RESTORE\_REF, DBSPJ\_REF

1.6.各个BLOCK的配置读取过程

## 配置读取阶段

当一个READ\_CONFIG\_REQ消息到达一个BLOCK时，这个BLOCK通常向mgmd请求其配置参数，然后设置BLOCK本身的各项参数，同时进行内存分配等任务

2.5. STTOR启动阶段说明

STTOR启动阶段本身是一个框架：

Ndbcntr::Missra::sendNextSTTOR

//这个框架安装如下方式工作：遍历所有的255个STTOR启动阶段，每个阶段都会向对该阶段感兴趣的BLOCK发送一个GSN\_STTOR消息，BLOCK在处理完GSN\_STTOR消息后，同时回复一个GSN\_STTORRY消息，并在该消息中注明，其感兴趣的下一个阶段号

//此处遍历所有的255个STTOR启动阶段

for(; currentStartPhase < 255 ; currentStartPhase++, g\_currentStartPhase = currentStartPhase)

{

//此处遍历每一个BLOCK，注意，遍历的起始BLOCK为currentBlockIndex，每收到一个GSN\_STTORRY消息后，由Ndbcntr::Missra::execSTTORRY加1，以便再次进入此处时直接处理下一个BLOCK

const Uint32 start = currentBlockIndex;

for(; currentBlockIndex < ALL\_BLOCKS\_SZ; currentBlockIndex++)

{

//此处过滤出对该STTOR启动阶段感兴趣的BLOCK

if(ALL\_BLOCKS[currentBlockIndex].NextSP == currentStartPhase)

{

signal->theData[1] = currentStartPhase;

signal->theData[7] = cntr.ctypeOfStart;

const BlockReference ref = ALL\_BLOCKS[currentBlockIndex].Ref;

if (refToBlock(ref) == DBDIH)

signal->theData[7] = cntr.cdihStartType;

//向该BLOCK发送一个GSN\_STTOR消息，通知其进行一个STTOR阶段

cntr.sendSignal(ref, GSN\_STTOR, signal, 8, JBB);

return;

//每当一个BLOCK处理完GSN\_STTOR消息后，将进行本BLOCK的初始化，然后向DBCNTR返回一个GSN\_STTORRY消息，该消息携带了BLOCK感兴趣的所有STTOR启动阶段，如BACKUP：

//Backup::sendSTTORRY(Signal\* signal)

//{

//向DBCNTR通告BACKUP感兴趣的STTOR阶段

// signal->theData[0] = 0;

// signal->theData[3] = 1;

// signal->theData[4] = 3;

// signal->theData[5] = 7;

// signal->theData[6] = 255; // No more start phases from missra

// BlockReference cntrRef = !isNdbMtLqh() ? NDBCNTR\_REF : BACKUP\_REF;

// sendSignal(cntrRef, GSN\_STTORRY, signal, 7, JBB);

//}

//而在DBCNTR收到一个BLOCK返回的GSN\_STTORRY消息后：

//Ndbcntr::Missra::execSTTORRY

// const BlockReference ref = signal->senderBlockRef();

// ndbrequire(refToBlock(ref) == refToBlock(ALL\_BLOCKS[currentBlockIndex].Ref));

// for (Uint32 i = 3; i < 25; i++)

// {

//更新BLOCK感兴趣的下一个阶段

// if (signal->theData[i] > currentStartPhase)

// {

// ALL\_BLOCKS[currentBlockIndex].NextSP = signal->theData[i];

// break;

// }

// }

//继续向下一个BLOCK发送GSN\_STTOR消息

// currentBlockIndex++;

// sendNextSTTOR(signal);

}

}

//一个完整的STTOR启动阶段完成，继续下一个，同时更新当前的STTOR启动阶段号

currentBlockIndex = 0;

NodeState newState(NodeState::SL\_STARTING, currentStartPhase,

(NodeState::StartType)cntr.ctypeOfStart);

cntr.updateNodeState(signal, newState);

//更新过程包括向所有BLOCK的JBB发送一个GSN\_NODE\_STATE\_REP消息：

// sendSignal(ALL\_BLOCKS[i].Ref, GSN\_NODE\_STATE\_REP, signal,

// NodeStateRep::SignalLength, JBB);

if(start != 0)

{

g\_eventLogger->info("Start phase %u completed", currentStartPhase);

signal->theData[0] = NDB\_LE\_StartPhaseCompleted;

signal->theData[1] = currentStartPhase;

signal->theData[2] = cntr.ctypeOfStart;

cntr.sendSignal(CMVMI\_REF, GSN\_EVENT\_REP, signal, 3, JBB);

if (cntr.wait\_sp(signal, currentStartPhase + 1))

{

currentStartPhase++;

g\_currentStartPhase = currentStartPhase;

return;

}

}

}

//STTOR启动阶段全部完成，记录日志并向CMVMI报告NDB\_LE\_NDBStartCompleted事件

g\_eventLogger->info("Node started");

signal->theData[0] = NDB\_LE\_NDBStartCompleted;

signal->theData[1] = NDB\_VERSION;

signal->theData[2] = NDB\_MYSQL\_VERSION\_D;

cntr.sendSignal(CMVMI\_REF, GSN\_EVENT\_REP, signal, 3, JBB);

cntr.updateNodeState(signal, newState);

//向NDBCNTR的JBB发送GSN\_CNTR\_START\_REP消息，进行下一个阶段的处理

NodeReceiverGroup rg(NDBCNTR, cntr.c\_clusterNodes);

signal->theData[0] = cntr.getOwnNodeId();

cntr.sendSignal(rg, GSN\_CNTR\_START\_REP, signal, 1, JBB);

当前所有BLOCK感兴趣的STTOR启动阶段列表：

|  |  |
| --- | --- |
| Kernel Block | Receptive Start Phases |
| NDBFS | 0 |
| DBTC | 1 |
| DBDIH | 1 |
| DBLQH | 1, 4 |
| DBACC | 1 |
| DBTUP | 1 |
| DBDICT | 1, 3 |
| NDBCNTR | 0, 1, 2, 3, 4, 5, 6, 8, 9 |
| CMVMI | 1 (prior to QMGR), 3, 8 |
| QMGR | 1, 7 |
| TRIX | 1 |
| BACKUP | 1, 3, 7 |
| DBUTIL | 1, 6 |
| SUMA | 1, 3, 5, 7, 100 (empty), 101 |
| DBTUX | 1,3,7 |
| TSMAN | 1, 3 (both ignored) |
| LGMAN | 1, 2, 3, 4, 5, 6 (all ignored) |
| PGMAN | 1, 3, 7 (Phase 7 currently empty) |
| RESTORE | 1,3 (only in Phase 1 is any real work done) |

## STTOR Phase 0

由于NDBFS在处理其GSN\_READ\_CONFIG\_REQ时，会创建若干个异步io线程，线程类为AsyncIoThread，最终的处理函数为AsyncIoThread::run，该函数根据请求，处理与文件系统相关的任务，文件系统操作通过调研AsyncFile类的接口完成，该类对应于不同的操作系统有不同的子类，windows下为Win32AsyncFile类，其它为PosixAsyncFile类。

NDBFS

创建ndbd文件系统目录

do\_mkdir(m\_base\_path[FsOpenReq::BP\_FS].c\_str())

DBCNTR

向NDBFS发出GSN\_FSREMOVEREQ请求，NDBFS将通过Ndbfs::execFSREMOVEREQ函数向一个AsyncIoThread发出io请求Request::rmrf，AsyncIoThread最终调用AsyncFile (PosixAsyncFile):: rmrfReq删除文件，删除完毕后，在Ndbfs::execFSREMOVEREQ中会继续调用Ndbfs::report向DBCNTR返回一个GSN\_FSREMOVECONF消息，通知其删除完毕。

if(m\_ctx.m\_config.getInitialStart())

clearFilesystem(signal);

Ndbcntr::clearFilesystem

sendSignal(NDBFS\_REF, GSN\_FSREMOVEREQ, signal, FsRemoveReq::SignalLength, JBA);

c\_fsRemoveCount++;

这个删除过程一次删除一个文件，每次删除完毕后，由NDBFS返回一个GSN\_FSREMOVECONF消息：

Ndbcntr::execFSREMOVECONF

if(c\_fsRemoveCount == CLEAR\_DX + CLEAR\_LCP + CLEAR\_DD){

sendSttorry(signal);

} else {

ndbrequire(c\_fsRemoveCount < CLEAR\_DX + CLEAR\_LCP + CLEAR\_DD);

clearFilesystem(signal);

}

当所有的文件都被删除后，DBCNTR会通告DBCNTR该STTOR阶段完成，发送一个STTORRY消息，否则继续进行文件删除过程

## STTOR Phase 1

除了NDBFS，其余所有的BLOCK都参与了此次启动：

CMVMI

//若需要锁住内存页，则使用mlockall锁住内存

if (m\_ctx.m\_config.lockPagesInMainMemory())

int res = NdbMem\_MemLockAll(1);

//重设watchdog的时间间隔

ndb\_mgm\_get\_int\_parameter(p, CFG\_DB\_WATCHDOG\_INTERVAL, &db\_watchdog\_interval);

update\_watch\_dog\_timer(db\_watchdog\_interval);

//启动定期内存报告

signal->theData[0] = ZREPORT\_MEMORY\_USAGE;

execCONTINUEB(signal);

sendSTTORRY(signal);

DBTUP

//计算实例引用号，组成方式：7位实例号 + 9位BLOCK号 + 16位节点号

cownref = calcInstanceBlockRef(DBTUP);

sendSignal(cntrRef, GSN\_STTORRY, signal, 5, JBB);

DBACC

sendSignal(cntrRef, GSN\_STTORRY, signal, 5, JBB);

DBTC

//初始化诸项变量

csystemStart = SSS\_FALSE;

ctimeOutCheckCounter = 0;

ctimeOutCheckFragCounter = 0;

ctimeOutMissedHeartbeats = 0;

ctimeOutCheckHeartbeat = 0;

ctimeOutCheckLastHeartbeat = 0;

ctimeOutCheckActive = TOCS\_FALSE;

ctimeOutCheckFragActive = TOCS\_FALSE;

sendSignal(c\_sttor\_ref, GSN\_STTORRY, signal, 5, JBB);

DBLQH

//建立对DBTUP、DBACC、LGMAN的引用

cstartPhase = tstartPhase;

c\_tup = (Dbtup\*)globalData.getBlock(DBTUP, instance());

c\_acc = (Dbacc\*)globalData.getBlock(DBACC, instance());

c\_lgman = (Lgman\*)globalData.getBlock(LGMAN);

sendSignal(c\_sttor\_ref, GSN\_STTORRY, signal, 6, JBB);

DBTUX

//建立对DBTUP的引用

c\_tup = (Dbtup\*)globalData.getBlock(DBTUP, instance());

sendSignal(cntrRef, GSN\_STTORRY, signal, 7, JBB);

DBDICT

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 7, JBB);

DBDIH

//创建若干互斥量，包括c\_startLcpMutexHandle、c\_switchPrimaryMutexHandle、c\_fragmentInfoMutex\_lcp，此次仅创建c\_startLcpMutexHandle

createMutexes(signal, 0);

sendSignal(NDBCNTR\_REF, GSN\_READ\_NODESREQ, signal, 1, JBB);

NDBCNTR

//重设启动状态，包括已启动节点数等

c\_start.reset()

//建立对DBLQH、DBDICT、DBTUP、DBACC、DBTC、DBDIH的引用

ndbBlocksPtr.p->blockref = DBLQH\_REF;

ndbBlocksPtr.p->blockref = DBDICT\_REF;

ndbBlocksPtr.p->blockref = DBTUP\_REF;

ndbBlocksPtr.p->blockref = DBACC\_REF;

ndbBlocksPtr.p->blockref = DBTC\_REF;

ndbBlocksPtr.p->blockref = DBDIH\_REF;

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 12, JBB);

QMGR

建立本节点的集群拓扑视图

//初始化各项参数，包括心跳间隔、仲裁超时、仲裁方法、启动分区超时、启动失败超时、连接检查间隔等

initData(signal);

//通知DBDIH进行节点重启过程

startphase1(signal);

//向DBDIH的JBB发送GSN\_DIH\_RESTARTREQ消息

sendSignal(DBDIH\_REF, GSN\_DIH\_RESTARTREQ, signal, DihRestartReq::SignalLength, JBB);

//DBDIH处理GSN\_DIH\_RESTARTREQ消息

Dbdih::execDIH\_RESTARTREQ

cntrlblockref = req->senderRef; //此处将cntrlblockref暂时设置为QMGR

if(m\_ctx.m\_config.getInitialStart())

//对于initial start，则不需要读取任何本地文件，也就不需要DBDIH重启

sendDihRestartRef(signal);

//向QMGR的JBB返回GSN\_DIH\_RESTARTREF消息

sendSignal(cntrlblockref, GSN\_DIH\_RESTARTREF, signal,

DihRestartRef::SignalLength, JBB);

else

//否则需要读取本地GCI文件，进行DBDIH重启，这里将经历一系列的异步处理过程，与NDBFS进行交互：{DBDIH, -GSN\_FSOPENREQ->, NDBFS}，{DBDIH, <-GSN\_FSOPENCONF-, NDBFS}，{DBDIH, -GSN\_FSREADREQ->, NDBFS}，{DBDIH, <-GSN\_FSREADCONF-, NDBFS}，{DBDIH, -GSN\_FSCLOSEREQ->, NDBFS}，{DBDIH, <-GSN\_FSCLOSECONF-, NDBFS}

readGciFileLab(signal);

//确认GCI文件被正确的打开并读取，并向QMGR通告结果

selectMasterCandidateAndSend(signal);

//得到本地最近GCI

DihRestartConf \* conf = CAST\_PTR(DihRestartConf, signal->getDataPtrSend());

conf->unused = getOwnNodeId();

conf->latest\_gci = SYSFILE->lastCompletedGCI[getOwnNodeId()];

//向QMGR的JBB返回GSN\_DIH\_RESTARTREF消息，捎带了本地最近GCI

sendSignal(cntrlblockref, GSN\_DIH\_RESTARTCONF, signal,

DihRestartConf::SignalLength, JBB);

recompute\_version\_info(NodeInfo::DB);

recompute\_version\_info(NodeInfo::API);

recompute\_version\_info(NodeInfo::MGM);

DBDIH在处理完毕后，最终将返回一条GSN\_DIH\_RESTARTREF或GSN\_DIH\_RESTARTCONF消息，无论如何，都将设置启动过程中本地节点的最近GCI，然后通过execCM\_INFOCONF进行一次集群拓扑确认过程

Qmgr::execDIH\_RESTARTREF

const DihRestartRef \* ref = CAST\_CONSTPTR(DihRestartRef, signal->getDataPtr());

c\_start.m\_latest\_gci = 0;

c\_start.m\_no\_nodegroup\_nodes.assign(NdbNodeBitmask::Size, ref->no\_nodegroup\_mask);

execCM\_INFOCONF(signal);

Qmgr::execDIH\_RESTARTCONF

const DihRestartConf \* conf = CAST\_CONSTPTR(DihRestartConf, signal->getDataPtr());

c\_start.m\_latest\_gci = conf->latest\_gci;

c\_start.m\_no\_nodegroup\_nodes.assign(NdbNodeBitmask::Size, conf->no\_nodegroup\_mask);

execCM\_INFOCONF(signal);

Qmgr::execCM\_INFOCONF

//向TRPMAN通告连接所有的ndbd节点

signal->theData[0] = 0; // no answer

signal->theData[1] = 0; // no id

signal->theData[2] = NodeInfo::DB;

sendSignal(TRPMAN\_REF, GSN\_OPEN\_COMORD, signal, 3, JBB);

cpresident = ZNIL;

cpresidentAlive = ZFALSE;

c\_start\_election\_time = NdbTick\_CurrentMillisecond();

//启动定期的启动失败检查计时器，间隔为3s

signal->theData[0] = ZSTART\_FAILURE\_LIMIT;

sendSignalWithDelay(reference(), GSN\_CONTINUEB, signal, 3000, 1);

//开始集群拓扑确认过程，向所有ndbd节点的QMGR的JBB发送GSN\_CM\_REGREQ消息

cmInfoconf010Lab(signal);

for (nodePtr.i = 1; nodePtr.i < MAX\_NDB\_NODES; nodePtr.i++)

ptrAss(nodePtr, nodeRec);

if(getNodeInfo(nodePtr.i).getType() != NodeInfo::DB)

continue;

c\_start.m\_nodes.setWaitingFor(nodePtr.i);

cnoOfNodes++;

if(!c\_connectedNodes.get(nodePtr.i))

continue;

sendCmRegReq(signal, nodePtr.i);

void Qmgr::sendCmRegReq(Signal \* signal, Uint32 nodeId)

CmRegReq \* req = (CmRegReq \*)&signal->theData[0];

req->blockRef = reference();

req->nodeId = getOwnNodeId();

req->version = NDB\_VERSION;

req->mysql\_version = NDB\_MYSQL\_VERSION\_D;

req->latest\_gci = c\_start.m\_latest\_gci;

req->start\_type = c\_start.m\_start\_type;

c\_start.m\_skip\_nodes.copyto(NdbNodeBitmask::Size, req->skip\_nodes);

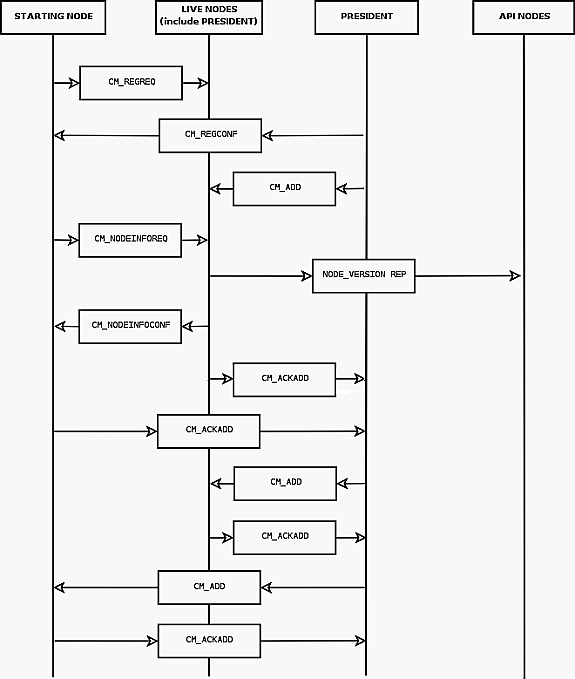
const Uint32 ref = calcQmgrBlockRef(nodeId);

sendSignal(ref, GSN\_CM\_REGREQ, signal, CmRegReq::SignalLength, JBB);

c\_start.m\_regReqReqSent++;

GSN\_CM\_REGREQ消息的处理过程将会非常漫长，中途经历十数轮消息交换，此处借用官方文档的消息交换图进行说明：

Qmgr::execCM\_REGREQ



官方文档的图片有一处错误即LIVE NODES应在向STARTING NODE发送完GSN\_CM\_NODEINFOCONF消息后，接着向PRESIDENT发送一条CM\_ACKADD消息，而不是由PRESIDENT向LIVE NODES发送一条CM\_ACKADD消息。

当这样的消息交互全部完成后，QMGR将会收到最后一轮CM\_ACKADD后，标志着一致的集群拓扑已经得到确认，此时全局有一个ndbd作为PRESIDENT，处理新节点的加入，且每个ndbd都有了其他所有ndbd的信息，此时需要向DBCNTR返回一个GSN\_STTORRY消息

Qmgr::execCM\_ACKADD

case CmAdd::CommitNew:

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 6, JBB);

TRIX

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 5, JBB);

BACKUP

//更新一些配置参数

m\_curr\_disk\_write\_speed = c\_defaults.m\_disk\_write\_speed\_sr;

m\_overflow\_disk\_write = 0;

m\_reset\_disk\_speed\_time = NdbTick\_CurrentMillisecond();

m\_reset\_delay\_used = Backup::DISK\_SPEED\_CHECK\_DELAY;

signal->theData[0] = BackupContinueB::RESET\_DISK\_SPEED\_COUNTER;

//定期重设磁盘速度检查计数器，间隔为Backup::DISK\_SPEED\_CHECK\_DELAY（100ms）

sendSignalWithDelay(reference(), GSN\_CONTINUEB, signal,

Backup::DISK\_SPEED\_CHECK\_DELAY, 1);

sendSignal(cntrRef, GSN\_STTORRY, signal, 7, JBB);

DBUTIL

c\_transId[0] = (number() << 20) + (getOwnNodeId() << 8);

c\_transId[1] = 0;

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 6, JBB);

SUMA

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 10, JBB);

LGMAN

//建立对DBTUP的引用

m\_tup = globalData.getBlock(DBTUP);

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 10, JBB);

PGMAN

//建立对DBTUP、LGMAN、TSMAN的引用

c\_tup = (Dbtup\*)globalData.getBlock(DBTUP, instance());

c\_lgman = (Lgman\*)globalData.getBlock(LGMAN);

c\_tsman = (Tsman\*)globalData.getBlock(TSMAN);

sendSignal(cntrRef, GSN\_STTORRY, signal, 7, JBB);

sendSignal(cntrRef, GSN\_STTORRY, signal, 6, JBB);

RESTORE

//建立对DBLQH、DBTUP的引用

c\_lqh = (Dblqh\*)globalData.getBlock(DBLQH, instance());

c\_tup = (Dbtup\*)globalData.getBlock(DBTUP, instance());

sendSignal(f\_STTOR\_REF, GSN\_STTORRY, signal, 6, JBB);

DBSPJ

//向DBSPJ发送一个GSN\_CONTINUEB消息，该消息将触发DBSPJ的定期释放RT\_SPJ\_DATABUFFER缓存，计时器间隔为100ms

sendSignalWithDelay(reference(), GSN\_CONTINUEB, signal, 1000, 1);

sendSignal(f\_STTOR\_REF, GSN\_STTORRY, signal, 6, JBB);

## STTOR Phase 2

NDBCNTR

Ndbcntr::execSTTOR

startPhase2Lab(signal);

Ndbcntr::startPhase2Lab

//再次向DBDIH发送GSN\_DIH\_RESTARTREQ消息，但此次DBDIH会将消息返回给NDBCNTR而不是QMGR

req->senderRef = reference();

sendSignal(DBDIH\_REF, GSN\_DIH\_RESTARTREQ, signal, DihRestartReq::SignalLength, JBB);

NDBCNTR在收到GSN\_DIH\_RESTARTCONF或GSN\_DIH\_RESTARTREF后将进行不同的处理，最终都将通过ph2Alab进行下一步初始化：

Ndbcntr::execDIH\_RESTARTCONF

const DihRestartConf \* conf = CAST\_CONSTPTR(DihRestartConf, signal->getDataPtrSend());

c\_start.m\_lastGci = conf->latest\_gci;

ctypeOfStart = NodeState::ST\_SYSTEM\_RESTART;

cdihStartType = ctypeOfStart;

ph2ALab(signal);

Ndbcntr::execDIH\_RESTARTREF

ctypeOfStart = NodeState::ST\_INITIAL\_START;

cdihStartType = ctypeOfStart;

ph2ALab(signal);

Ndbcntr::ph2Alab

signal->theData[0] = reference();

sendSignal(QMGR\_REF, GSN\_READ\_NODESREQ, signal, 1, JBB);

QMGR进行的处理，向NDBCNTR通告集群当前的master节点

Qmgr::execREAD\_NODESREQ

ReadNodesConf \* const readNodes = (ReadNodesConf \*)&signal->theData[0];

NdbNodeBitmask tmp = c\_definedNodes;

tmp.bitANDC(c\_clusterNodes);

readNodes->noOfNodes = c\_definedNodes.count();

readNodes->masterNodeId = cpresident;

readNodes->ndynamicId = nodePtr.p->ndynamicId;

tmp.copyto(NdbNodeBitmask::Size, readNodes->inactiveNodes);

sendSignal(TBref, GSN\_READ\_NODESCONF, signal, ReadNodesConf::SignalLength, JBB);

NDBCNTR在收到QMGR的确认后，将向master节点发送

Ndbcntr::execREAD\_NODESCONF

sendCntrStartReq(signal);

CntrStartReq \* req = (CntrStartReq\*)signal->getDataPtrSend();

req->startType = ctypeOfStart;

req->lastGci = c\_start.m\_lastGci;

req->nodeId = getOwnNodeId();

//向master节点请求启动

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_START\_REQ,

signal, CntrStartReq::SignalLength, JBB);

//触发对master节点请求过程进行检查的计时器

signal->theData[0] = ZSTARTUP;

sendSignalWithDelay(reference(), GSN\_CONTINUEB, signal, 1000, 1);

master节点可能在此阶段发生变动，若发生变动或没有准备好启动过程，其将向请求节点返回GSN\_CNTR\_START\_REF消息，若其准备好启动这个节点，则向请求节点返回一个GSN\_CNTR\_START\_CONF消息

若NDBCNTR收到了GSN\_CNTR\_START\_REF消息，且错误码为CntrStartRef::NotMaster，则表明master已经发生了变动，但在收到该消息前，master已经被重新选举出来，此时仅仅需要向新的master再次发送GSN\_CNTR\_START\_REQ消息即可，而错误码为CntrStartRef::StopInProgress时，则表明启动过程出错，不可进行下去

Ndbcntr::execCNTR\_START\_REF

const CntrStartRef \* ref = (CntrStartRef\*)signal->getDataPtr();

switch(ref->errorCode)

case CntrStartRef::NotMaster:

cmasterNodeId = ref->masterNodeId;

sendCntrStartReq(signal);

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_START\_REQ,

signal, CntrStartReq::SignalLength, JBB);

return;

case CntrStartRef::StopInProgress:

progError(\_\_LINE\_\_, NDBD\_EXIT\_RESTART\_DURING\_SHUTDOWN);

若NDBCNTR收到了GSN\_CNTR\_START\_CONF消息，他将更新本地记录的启动GCI和master节点id，并开始启动NDB\_STTOR过程，这个过程也是一个框架类似于STTOR启动框架，只不过消息从GSN\_STTOR变成了GSN\_NDB\_STTOR

Ndbcntr::execCNTR\_START\_CONF

const CntrStartConf \* conf = (CntrStartConf\*)signal->getDataPtr();

cnoStartNodes = conf->noStartNodes;

ctypeOfStart = (NodeState::StartType)conf->startType;

cdihStartType = ctypeOfStart;

c\_start.m\_lastGci = conf->startGci;

cmasterNodeId = conf->masterNodeId;

NdbNodeBitmask tmp;

tmp.assign(NdbNodeBitmask::Size, conf->startedNodes);

c\_startedNodes.bitOR(tmp);

c\_start.m\_starting.assign(NdbNodeBitmask::Size, conf->startingNodes);

m\_cntr\_start\_conf = true;

//启动NDB\_STTOR启动阶段1

ph2GLab(signal);

if (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

//计算下一个要发送的BLOCK

ndbBlocksPtr.i = cndbBlocksCount;

ptrCheckGuard(ndbBlocksPtr, ZSIZE\_NDB\_BLOCKS\_REC, ndbBlocksRec);

sendSignal(ndbBlocksPtr.p->blockref, GSN\_NDB\_STTOR, signal, 22, JBB);

//更新下一个要发送的BLOCK计数器

cndbBlocksCount++;

//每当一个BLOCK处理完毕它的NDB\_STTOR启动阶段后，都会向NDBCNTR返回一个GSN\_NDB\_STTORRY消息

//Ndbcntr::execNDB\_STTORRY

// switch (cstartPhase) {

// case ZSTART\_PHASE\_2:

// 再次向下一个BLOCK发送

// ph2GLab(signal);

// return;

return;

//每当一个NDB\_STTOR阶段完成，都会想NDBCNTR发送GSN\_STTORRY确认

sendSttorry(signal);

5.5参与NDB\_STTOR的BLOCK

初始化时，NDBCNTR设置其ndbBlockRec依序包含如下BLOCK：

DBLQH、DBDICT、DBTUP、DBACC、DBTC、DBDIH

## NDB\_STTOR Phase 1

DBLQH

Dblqh::execNDB\_STTOR

//初始化若干变量

//通过NDBFS打开并初始化DBLQH的FragLog（类型FsOpenReq::S\_FRAGLOG），ndb\_#\_fs/D[8-10]/DBLQH/\*.FragLog

startphase1Lab(signal, /\* dummy \*/ ~0, ownNodeId);

//启动操作统计报告计时器，间隔为10ms

const Uint32 len = c\_Counters.build\_continueB(signal);

signal->theData[0] = ZOPERATION\_EVENT\_REP;

sendSignalWithDelay(cownref, GSN\_CONTINUEB, signal, 10, len);

DBDICT

Dbdict::execNDB\_STTOR

//通过NDBFS打开并初始化DBDICT的SchemaLog（类型FsOpenReq::S\_SCHEMALOG），ndb\_#\_fs/D[1-2]/DBDICT/\*.SchemaLog

initSchemaFile(signal);

Dbdict::initSchemaFile

if (c\_initialStart || c\_initialNodeRestart)

//若启动过程为initial或initial\_node\_restart，则需要重新建立SchemaLog

startWriteSchemaFile(signal);

else

//若启动过程为system\_restart或node\_restart，则需要重新打开SchemaLog

startReadSchemaFile(signal);

DBTUP

Dbtup::execNDB\_STTOR

//初始化若干变量

cownNodeId = ownNodeId;

cownref = calcInstanceBlockRef(DBTUP);

initializeDefaultValuesFrag();

DBACC

Dbacc::execNDB\_STTOR

//无动作

DBTC

Dbtc::execNDB\_STTOR

//加入对其它BLOCK的引用，初始化若干变量

intstartphase1x010Lab(signal);

DBDIH

Dbdih::execNDB\_STTOR

//加入对其它BLOCK的引用，初始化若干变量

## STTOR Phase 3

Dbdict::execSTTOR

//设置当前启动类型

c\_restartType = signal->theData[7];

Ndbcntr::execSTTOR

//启动NDB\_STTOR启动阶段2

startPhase3Lab(signal);

ph3ALab(signal);

if (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

sendSttorry(signal);

## NDB\_STTOR Phase 2

DBLQH

Dblqh::execNDB\_STTOR

startphase2Lab(signal, /\* dummy \*/ ~0);

moreconnectionsLab(signal);

//注意此处，将为所有其他ndbd连接分配一个DBACC和DBTUP的Operationrec

signal->theData[0] = tcConnectptr.i;

//向DBACC请求一个DBACC::Operationrec

sendSignal(caccBlockref, GSN\_ACCSEIZEREQ, signal, 2, JBB);

Dblqh::execACCSEIZECONF

//得到向DBACC请求的DBACC::Operationrec

tcConnectptr.p->accConnectrec = signal->theData[1];

//向DBTUP请求一个DBTUP::Operationrec

sendSignal(ctupBlockref, GSN\_TUPSEIZEREQ, signal, 2, JBB);

Dblqh::execTUPSEIZECONF

//得到向DBTUP请求的DBTUP::Operationrec

tcConnectptr.p->tupConnectrec = signal->theData[1];

//为下一个ndbd分配

tcConnectptr.i = tcConnectptr.p->nextTcConnectrec;

if (tcConnectptr.i != RNIL)

moreconnectionsLab(signal);

sendNdbSttorryLab(signal);

DBDICT

Dbdict::execNDB\_STTOR

//无动作

DBTUP

Dbtup::execNDB\_STTOR

//无动作

DBACC

Dbacc::execNDB\_STTOR

//无动作

DBTC

Dbtc::execNDB\_STTOR

//无动作

DBDIH

Dbdih::execNDB\_STTOR

if (cstarttype == NodeState::ST\_INITIAL\_START)

else if (cstarttype == NodeState::ST\_SYSTEM\_RESTART)

//仅对node\_restart和initial\_node\_restart有效

else if ((cstarttype == NodeState::ST\_NODE\_RESTART) ||

(cstarttype == NodeState::ST\_INITIAL\_NODE\_RESTART))

nodeRestartPh2Lab(signal);

Uint32 lockType = DictLockReq::NodeRestartLock;

Callback c = { safe\_cast(&Dbdih::recvDictLockConf\_nodeRestart), 0 };

sendDictLockReq(signal, lockType, c);

sendSignal(dictMasterRef, GSN\_DICT\_LOCK\_REQ, signal,

DictLockReq::SignalLength, JBB);

Dbdict::execDICT\_LOCK\_REQ

//加DictLockReq::NodeRestartLock锁

lockReq.requestInfo |= UtilLockReq::SharedLock;

res = m\_dict\_lock.lock(this, m\_dict\_lock\_pool, &lockReq, 0);

向DBDIH返回

DictLockConf\* conf = (DictLockConf\*)signal->getDataPtrSend();

conf->userPtr = lockReq.senderData;

conf->lockType = lockReq.extra;

conf->lockPtr = lockReq.senderData;

sendSignal(lockReq.senderRef, GSN\_DICT\_LOCK\_CONF, signal,

DictLockConf::SignalLength, JBB);

Dbdih::execDICT\_LOCK\_CONF

recvDictLockConf(signal);

//得到向DBDICT请求的DictLockConf

const DictLockConf\* conf = (const DictLockConf\*)&signal->theData[0];

lockPtr.p->callback.m\_callbackData = lockPtr.i;

//执行用户回调，此处回调函数为Dbdih::recvDictLockConf\_nodeRestart

execute(signal, lockPtr.p->callback, 0);

Dbdih::recvDictLockConf\_nodeRestart

nodeRestartPh2Lab2(signal);

//向master节点请求准予启动，master向其它ndbd询问，经过一轮判定后，向DBDIH返回结果

StartPermReq \* const req = (StartPermReq \*)&signal->theData[0];

req->blockRef = reference();

req->nodeId = cownNodeId;

req->startType = cstarttype;

sendSignal(cmasterdihref, GSN\_START\_PERMREQ, signal, 3, JBB);

Dbdih::execSTART\_PERMCONF

ndbsttorry10Lab(signal, \_\_LINE\_\_);

以下是对官方文档中本阶段的一些翻译：

The DBDIH kernel block creates the mutexes used by the NDB kernel and reads nodes using the READ\_NODESREQ signal. With the data from the response to this signal, DBDIH can create node lists, node groups, and so forth. For node restarts and initial node restarts, DBDIH also asks the master for permission to perform the restart. The master will ask all “live” nodes if they are prepared to permit the new node to join the cluster. If an initial node restart is to be performed, then all LCPs are invalidated as part of this phase.

DBDIH将创建NDB内核使用的一些互斥量，同时使用READ\_NODESREQ消息获取节点信息。通过这些节点信息，DBDIH将创建节点列表、节点组等。对于node\_restart和initial\_node\_restart，DBDIH还会询问master节点是否允许启动。master节点将询问所有存活节点以允许新节点加入集群。对于initial\_node\_restart，该阶段必须令所有的LCP失效。

LCPs from nodes that are not part of the cluster at the time of the initial node restart are not invalidated. The reason for this is that there is never any chance for a node to become master of a system restart using any of the LCPs that have been invalidated, since this node must complete a node restart—including a local checkpoint—before it can join the cluster and possibly become a master node.

在initial\_node\_restart启动时，节点的LCP还不是集群的一部分，但却没有被失效，原因在于，在system\_restart启动时，使用被失效的LCP的节点不能成为集群master，因此节点必须在其加入集群前和成为master前完成包括包括LCP在内的节点重启

The CMVMI kernel block activates the sending of packed signals, which occurs only as part of database operations. Packing must be enabled prior to beginning any such operations during the execution of the redo log or node recovery phases.

CMVMI将激活打包发送过程，将其作为数据库操作的一部分，该过程必须先于redo日志执行和节点恢复过程

The DBTUX block sets the type of start currently taking place, while the BACKUP block sets the type of restart to be performed, if any (in each case, the block actually sets a variable whose value reflects the type of start or restart). The SUMA block remains inactive during this phase.

DBTUX设置当前的启动类型，BACKUP设置重启类型，SUMA未被激活

The PGMAN kernel block starts the generation of two repeated signals, the first handling cleanup. This signal is sent every 200 milliseconds. The other signal handles statistics, and is sent once per second.

PGMAN产生两个计时器，一个用于清除，间隔200ms，一个用于统计，间隔1s

## STTOR Phase 4

DBLQH

Dblqh::execSTTOR

define\_backup(signal);

DefineBackupReq \* req = (DefineBackupReq\*)signal->getDataPtrSend();

req->backupDataLen = ~0;

//向BACKUP的JBB发送GSN\_DEFINE\_BACKUP\_REQ消息，请求创建一个DefineBackupConf及备份id，在创建本地检查点时使用

sendSignal(backupRef, GSN\_DEFINE\_BACKUP\_REQ, signal,

DefineBackupReq::SignalLength, JBB);

BACKUP在收到GSN\_DEFINE\_BACKUP\_REQ后的处理

Backup::execDEFINE\_BACKUP\_REQ

//取得备份数据长度和备份标志

DefineBackupReq\* req = (DefineBackupReq\*)signal->getDataPtr();

BackupRecordPtr ptr LINT\_SET\_PTR;

if(senderRef == reference())

ptr.p->flags = req->flags;

else

ptr.p->flags = req->flags & ~((Uint32)BackupReq::WAITCOMPLETED);

ptr.p->backupDataLen = req->backupDataLen;

//备份过程将生成三个文件，分表为BackupFormat::CTL\_FILE（元数据）、BackupFormat::UNDO\_FILE| BackupFormat::LOG\_FILE（增量事务）、BackupFormat::DATA\_FILE（数据）

BackupFilePtr files[3];

// backupDataLen为0时，将备份所有数据

if(ptr.p->backupDataLen == 0)

backupAllData(signal, ptr);

return;

//backupDataLen为-1时，仅仅生成一个DefineBackupConf结构返回，包含一个备份id，此场景经历此路径

if(ptr.p->is\_lcp())

getFragmentInfoDone(signal, ptr);

ptr.p->m\_gsn = GSN\_DEFINE\_BACKUP\_CONF;

ptr.p->slaveState.setState(DEFINED);

DefineBackupConf \* conf = (DefineBackupConf\*)signal->getDataPtr();

conf->backupPtr = ptr.i;

conf->backupId = ptr.p->backupId;

sendSignal(ptr.p->masterRef, GSN\_DEFINE\_BACKUP\_CONF, signal,

DefineBackupConf::SignalLength, JBB);

return;

//得到BACKUP返回的DefineBackupConf结构

Dblqh::execDEFINE\_BACKUP\_CONF

{

DefineBackupConf \* conf = (DefineBackupConf\*)signal->getDataPtrSend();

m\_backup\_ptr = conf->backupPtr;

sendsttorryLab(signal);

}

NDBCNTR

Ndbcntr::execSTTOR

startPhase4Lab(signal);

ph4ALab(signal);

ph4BLab(signal);

//启动NDB\_STTOR启动阶段3

if (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

return;

//对于node\_restart和initial\_node\_restart，NDB\_STTOR阶段3的启动过程已经完成

if ((ctypeOfStart == NodeState::ST\_NODE\_RESTART) ||

(ctypeOfStart == NodeState::ST\_INITIAL\_NODE\_RESTART))

sendSttorry(signal);

return;

//对于system\_restart和initial\_start，master节点将等待所有其它节点报告，该函数的实现将在NDB\_STTOR启动阶段3末尾介绍

waitpoint41Lab(signal);

return;

## NDB\_STTOR Phase 3

DBLQH

Dblqh::execNDB\_STTOR

startphase3Lab(signal);

Dblqh::startphase3Lab

switch(cstartType)

case NodeState::ST\_NODE\_RESTART:

case NodeState::ST\_SYSTEM\_RESTART:

//对于node\_restart和system\_restart，以读写方式打开所有的FragLog文件

for (logPartPtr.i = 0; logPartPtr.i < clogPartFileSize; logPartPtr.i++)

{

LogFileRecordPtr locLogFilePtr;

ptrAss(logPartPtr, logPartRecord);

locLogFilePtr.i = logPartPtr.p->firstLogfile;

ptrCheckGuard(locLogFilePtr, clogFileFileSize, logFileRecord);

locLogFilePtr.p->logFileStatus = LogFileRecord::OPEN\_SR\_FRONTPAGE;

openFileRw(signal, locLogFilePtr);

//此处将进行一条异步的redolog恢复过程，假设过程全部为conf类，状态变更和消息传递过程如下：{状态，处理函数}

//{OPEN\_SR\_FRONTPAGE, openSrFrontpageLab }

//{READ\_SR\_FRONTPAGE, readSrFrontpageLab}

//{CLOSING\_SR\_FRONTPAGE, closingSrFrontPage}

//{OPEN\_SR\_LAST\_FILE, openSrLastFileLab}

//{READ\_SR\_LAST\_FILE, readSrLastFileLab}

// |<----|

//{READ\_SR\_LAST\_MBYTE, readSrLastMbyteLab} |循环

// |---->|

// |-全部文件检查完毕->{CLOSING\_SR, closingSrLab}

// |-仍然有待检查的文件->

// {OPEN\_SR\_NEXT\_FILE, openSrNextFileLab}

// |<----------------------------|

// {READ\_SR\_NEXT\_FILE, readSrNextFileLab} |

// |-全部文件检查完毕->{CLOSING\_SR, closingSrLab} |

// |-仍然有待检查的文件----------------------------------------------->|

//redo日志读取完毕后

//Dblqh::closingSrLab

// signal->theData[0] = ZSR\_PHASE3\_START;

// signal->theData[1] = ZSR\_PHASE1\_COMPLETED;

// sendSignal(cownref, GSN\_CONTINUEB, signal, 2, JBB);

//Dblqh::execCONTINUEB

// srPhase3Start(signal);

// 带两个启动阶段都完成后，仅仅需要再启动一个定期检查GCI数目的计时器

// csrPhaseStarted = ZSR\_BOTH\_PHASES\_STARTED;

// signal->theData[0] = ZSR\_GCI\_LIMITS;

// signal->theData[1] = fragptr.i;

// sendSignal(cownref, GSN\_CONTINUEB, signal, 2, JBB);

}//for

break;

case NodeState::ST\_INITIAL\_START:

case NodeState::ST\_INITIAL\_NODE\_RESTART:

//对于initial\_start和initial\_node\_restart，初始化4个FragLog文件

for (logPartPtr.i = 0; logPartPtr.i < clogPartFileSize; logPartPtr.i++)

{

signal->theData[0] = ZINIT\_FOURTH;

signal->theData[1] = logPartPtr.i;

sendSignal(cownref, GSN\_CONTINUEB, signal, 2, JBB);

//Dblqh::execCONTINUEB

// initFourth(signal);

// locLogFilePtr.p->logFileStatus = LogFileRecord::OPEN\_SR\_FOURTH\_PHASE;

// openFileRw(signal, locLogFilePtr);

// 此处将进行一条同步的redolog创建过程，假设过程全部为conf类，状态变更和消息传递过程如下：{状态，处理函数}

//{OPEN\_SR\_FOURTH\_PHASE, openSrFourthPhaseLab}

//{READ\_SR\_FOURTH\_PHASE, readSrFourthPhaseLab}

// |

// initLogpage

// |-有剩余文件->

// {OPEN\_SR\_FOURTH\_NEXT, openSrFourthNextLab}

// {\*\*, openSrFourthZeroSkipInitLab}

// {READ\_SR\_FOURTH\_ZERO, readSrFourthZeroLab}

// {\*\*, readFileInInvalidate(signal, 3)} <----|

// {CLOSE\_SR\_READ\_INVALIDATE\_SEARCH\_FILES, readFileInInvalidate(signal, 4)}|

// {OPEN\_SR\_READ\_INVALIDATE\_SEARCH\_FILES, readFileInInvalidate(signal, 5)} |

// {READ\_SR\_INVALIDATE\_SEARCH\_FILES, invalidateLogAfterLastGCI} |

// |-未找到了最后一个需要失效的页-----------------------------------------------------|

// |-找到了最后一个需要失效的页，遍历所有需要失效的页-> |

// {\*\*, writeFileInInvalidate} |

// {OPEN\_SR\_WRITE\_INVALIDATE\_PAGES, writeFileInInvalidate } <-| |

// {\*\*, writeSinglePage（写入数据令页失效）} | |

// {\*\*, invalidateLogAfterLastGCI} | |

// { WRITE\_SR\_INVALIDATE\_PAGES, writeFileInInvalidate } | |

// |-失效未完成-----------------------------------------------------------| |

// |-失效完成-> |

// {\*\*, exitFromInvalidate } |

// {WRITE\_SR\_INVALIDATE\_PAGES\_UPDATE\_PAGE0, |

// signal->theData[0] = ZSR\_FOURTH\_COMP; |

// signal->theData[1] = logPartPtr.i; |

// sendSignal(cownref, GSN\_CONTINUEB, signal, 2, JBB)} |

// {\*\*, srFourthComp} |

// 完成FragLog的创建/写入/失效 |

// cinitialStartOngoing = ZFALSE; |

// cstartRecReq = SRR\_REDO\_COMPLETE; |

// checkStartCompletedLab(signal); |

// |-当前文件不需要被失效-> |

// {\*\*, readFileInInvalidate(signal, 6)} |

// {CLOSE\_SR\_READ\_INVALIDATE\_SEARCH\_LAST\_FILE, |

// readFileInInvalidate(signal, 7)} |

// {\*\*, readFileInInvalidate(signal, 7)} |

// |-若处理第一个文件-> |

// {\*\*, readFileInInvalidate(signal, 1) } |

// {\*\*, invalidateCloseFile（关闭文件）} |

// {CLOSE\_SR\_READ\_INVALIDATE\_PAGES, |

// readFileInInvalidate(signal, 2)}-------| |

// |-若处理后续文件-> | |

// {\*\*, readFileInInvalidate(signal, 2) } <-----| |

// |-若未打开文件-> |

// {OPEN\_SR\_READ\_INVALIDATE\_PAGES, |

// readFileInInvalidate(signal, 0)} |

// {READ\_SR\_INVALIDATE\_PAGES, |

// invalidateLogAfterLastGCI} |

// |-若已打开文件---------------------------------------------------------->|

// |-无剩余文件，同样进入openSrFourthZeroSkipInitLab->

// {\*\*, openSrFourthZeroSkipInitLab}

}

break;

}

signal->theData[0] = cownref;

//向NDBCNTR请求所有节点信息

sendSignal(NDBCNTR\_REF, GSN\_READ\_NODESREQ, signal, 1, JBB);

此时，将有两条并行执行路径，一条继续进行启动过程，另一条将处理redo日志文件

此处叙述继续启动的过程：

Ndbcntr::execREAD\_NODESREQ

//将节点信息返回给请求者

ReadNodesConf \* const readNodes = (ReadNodesConf \*)&signal->theData[0];

readNodes->noOfNodes = c\_allDefinedNodes.count();

readNodes->masterNodeId = cmasterNodeId;

readNodes->ndynamicId = cdynamicNodeId;

sendSignal(TuserBlockref, GSN\_READ\_NODESCONF, signal,

ReadNodesConf::SignalLength, JBB);

DBLQH收到节点信息后的处理：

Dblqh::execREAD\_NODESCONF

caddNodeState = ZFALSE;

//对于system\_restart和node\_restart，将由异步读取redolog过程完成初始化，此时不需要阻塞

if (cstartType == NodeState::ST\_SYSTEM\_RESTART)

{

sendNdbSttorryLab(signal);

return;

}

else if (cstartType == NodeState::ST\_NODE\_RESTART)

{

m\_sr\_nodes.clear();

m\_sr\_nodes.set(getOwnNodeId());

sendNdbSttorryLab(signal);

return;

}

//对于initial\_start和initial\_node\_restart，将继续阻塞直到所有redolog文件被建立完毕

checkStartCompletedLab(signal);

if (caddNodeState == ZFALSE)

//在调用writeFileInInvalidate的过程中，当redolog创建/写入/失效完成后触发，这使得该过程成为一个同步过程，之后的其他启动阶段可能依赖于这些日志文件

if (cinitialStartOngoing == ZFALSE)

sendNdbSttorryLab(signal);

return;

return;

DBDICT

Dbdict::execNDB\_STTOR

//同DBLQH，仍然是取得节点信息后进行一系列变量初始化

sendSignal(NDBCNTR\_REF, GSN\_READ\_NODESREQ, signal, 1, JBB);

Dbdict::execREAD\_NODESCONF

ReadNodesConf \* const readNodes = (ReadNodesConf \*)&signal->theData[0];

sendNDB\_STTORRY(signal);

DBTUP

Dbtup::execNDB\_STTOR

startphase3Lab(signal, ~0, ~0);

//无动作

DBACC

Dbacc::execNDB\_STTOR

//无动作

DBTC

Dbtc::execNDB\_STTOR

//同DBDICT，仍然是取得节点信息后进行一系列变量初始化

intstartphase3x010Lab(signal);

sendSignal(cndbcntrblockref, GSN\_READ\_NODESREQ, signal, 1, JBB);

//启动一个定期报告事务事件的计时器

const Uint32 len = c\_counters.build\_continueB(signal);

signal->theData[0] = TcContinueB::ZTRANS\_EVENT\_REP;

sendSignalWithDelay(cownref, GSN\_CONTINUEB, signal, 10, len);

Dbtc::execREAD\_NODESCONF

ReadNodesConf \* const readNodes = (ReadNodesConf \*)&signal->theData[0];

ndbsttorry010Lab(signal);

DBDIH

Dbdih::execNDB\_STTOR

cmasterState = MASTER\_IDLE;

if(cstarttype == NodeState::ST\_INITIAL\_START ||

cstarttype == NodeState::ST\_SYSTEM\_RESTART)

cmasterState = isMaster() ? MASTER\_ACTIVE : MASTER\_IDLE;

//非master节点的initial\_start

if (!isMaster() && cstarttype == NodeState::ST\_INITIAL\_START)

ndbStartReqLab(signal, cntrRef);

//初始化SYSFILE对象

initRestartInfo(signal);

//创建GCI文件

initGciFilesLab(signal);

return;

ndbsttorry10Lab(signal, \_\_LINE\_\_);

NDB\_STTOR阶段3完成后，将重新回到STTOR阶段4末尾，master节点的后续工作将在所有其他节点本地redolog就绪后开始：

Ndbcntr::waitpoint41Lab

if (getOwnNodeId() == cmasterNodeId)

cnoWaitrep++;

//此处master节点将等待所有的其它节点启动，自己通过GSN\_CNTR\_WAITREP消息进行等待，当所有其它节点报告后，将通过GSN\_NDB\_STARTREQ消息进行下述启动过程：

//在建立了DBDIH和DBDICT中的数据后，DBDIH将选择redolog的起止分片，然后进行恢复工作：1.从磁盘读入本地检查点；2.为索引和数据执行undolog；3.根据一至多个节点的redolog执行分片redolog以恢复数据

if (cnoWaitrep == cnoStartNodes)

sendSignal(DBDIH\_REF, GSN\_NDB\_STARTREQ, signal, 2, JBB);

else

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_WAITREP, signal, 2, JBB);

Ndbcntr::execCNTR\_WAITREP

CntrWaitRep\* rep = (CntrWaitRep\*)signal->getDataPtr();

Uint32 twaitPoint = rep->waitPoint;

switch (twaitPoint)

case CntrWaitRep::ZWAITPOINT\_4\_1:

//继续等待，当所有非master节点都报告时，将到达全局同步点ZWAITPOINT\_4\_1

waitpoint41Lab(signal);

break;

非master节点首先启动，待所有非master节点启动完成且本地redolog就绪后，master节点将在NDBCNTR的STTOR阶段4末尾处，发起GSN\_NDB\_STARTREQ消息启动，根据官方文档翻译得到system\_restart的启动过程：

1.master节点设置最近GCI作为启动GCI，然后将这个系统文件同步到system\_restart过程中所有其他节点处。

2.接下来master节点将同步所有节点的schema，这由15个步骤组成。此处需要解决一个问题：当一个schema对象在节点可用时被创建，而在节点不可用时被销毁，但一个新对象（甚至可能与被销毁的对象有相同的ID）可能在节点不可用时被创建。为了处理这种情形，有必要在这个启动节点上率先重建所有被其视作存在的对象。之后，在该节点不可用时，其它节点所销毁的对象也将被该节点销毁，这个原则也适用于表对象。最后，在该节点不可用时，其它节点所创建的表也将在该节点上重建。所有这些操作都是本地化的操作。作为该过程的一部分，也有必要确保所有的需要重建的表已经被本地创建，且其相关的数据结构也被所有的BLOCK所建立。

在master节点执行了上述过程后，新的schema文件将被master结点发送到所有其它结点上，其它结点将执行相同的同步过程。

3.所有重启过程中涉及的分片必须有正确的参数内容（派生自DBDIH）。这使得DBDIH发送一些GSN\_START\_FRAGREQ消息到DBLQH，同时也将启动分片的恢复过程，这些分片顺序地从磁盘上读取，并行地进行恢复，这仅能恢复表的主存部分。

4.一旦所有的分片被恢复，一个GSN\_START\_RECREQ消息被发送到所有结点上，而所有的undolog将被执行。

5.LGMAN执行undolog后，将由TSMAN执行恢复工作，TSMAN将遍历表空间的所有扩展表头。

6.接着，需要开始准备redolog的正式恢复过程。redolog的执行包括四个阶段。对于每一个分片的恢复，可能需要执行多个结点的redolog，可以在不同阶段为同一个分片执行redolog来实现这个任务，正如DBDIH发送GSN\_START\_FRAGREQ消息时所做的一样。一个GSN\_EXEC\_FRAGREQ消息在每一个阶段都将被发出，需要在该阶段恢复的分片可以得到处理，这个步骤完成后，一个GSN\_EXEC\_SRREQ消息也将被发送到所有结点上，以通知他们开始执行自己的redolog。

注意：在开始执行第一个redolog前，需要确保DBLQH在STTOR启动阶段4的启动工作已经完成，否则需要等待其完成。

7.在执行redolog前，还需要计算redolog的开始和结束位置。redolog的结束位置将在到达最近GCI时确定

8.在完成redolog的执行后，所有在最近GCI之后写的redolog页将被失效。鉴于redolog的环形特性，可能导致对一个新redolog执行失效操作，而这个新的redolog比已经执行的redolog编号更小。

9.在完成这项阶段后，DBLQH将使用GSN\_START\_RECCONF消息向DBDIH报告。

10.当master节点接收到了所有节点的消息后，它将向NDBCNTR发送一个GSN\_NDB\_STARTCONF消息。

11. GSN\_NDB\_STARTCONF消息标志着NDBCNTR在STTOR阶段4的工作完成，这也是该阶段唯一的重量级工作。

对于几种启动类型在该阶段的工作进行的总结：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| BLOCK | initial\_start | system\_restart | node\_restart | initial\_node\_restart |
|  | 系统级 | 系统级 | 节点级 | 节点级 |
| DBDIH | 非master节点同步创建redolog | 非master节点异步检查redolog | 启动节点异步检查redolog | 启动节点异步创建redolog |
| NDBCNTR | master节点同步创建redolog | master节点同步检查redolog |  |  |

此处官方文档的一些翻译：

The DBLQH block initiates checking of the log files here. Then it obtains the states of the data nodes using the READ\_NODESREQ signal. Unless an initial start or an initial node restart is being performed, the checking of log files is handled in parallel with a number of other start phases. For initial starts, the log files must be initialized; this can be a lengthy process and should have some progress status attached to it.

DBLQH在此处初始化对日志文件的检查，使用GSN\_READ\_NODESREQ获取数据节点的状态。对于node\_restart和system\_restart，日志检查过程将与其他启动阶段同时执行，对于initial\_start和initial\_node\_restart，日志文件必须被同步地初始化，其他启动阶段可能依赖于日志文件

Note

From this point, there are two parallel paths, one continuing to restart and another reading and determining the state of the redo log files.

注意，从此时开始，将有两条并行执行路径，一条继续进行启动过程，另一条将处理redo日志文件

The DBDICT block requests information about the cluster data nodes using the READ\_NODESREQ signal.

DBDICT通过GSN\_READ\_NODESREQ消息请求关于集群数据节点的信息

DBACC resets the system restart flag if this is not a system restart; this is used only to verify that no requests are received from DBTUX during system restart.

若当前不是system\_restart，DBACC将设置为system\_restart，仅仅用于验证在system\_restart时，DBACC并没有接收来自DBTUX的请求

DBTC requests information about all nodes by means of the READ\_NODESREQ signal.

DBTC通过GSN\_READ\_NODESREQ消息请求所有节点的信息

DBDIH sets an internal master state and makes other preparations exclusive to initial starts. In the case of an initial start, the nonmaster nodes perform some initial tasks, the master node doing once all nonmaster nodes have reported that their tasks are completed. (This delay is actually unnecessary since there is no reason to wait while initializing the master node.)

DBDIH将设置master的状态，对于initial\_start，令其它节点的准备过程成为独占式的。对于一个initial\_start，非master节点将执行一些初始化工作，而master节点在所有非master节点完成并报告后才进行初始化（实际上master节点的这种 延迟初始化时没有任何必要的）

For node restarts and initial node restarts no more work is done in this phase. For initial starts the work is done when all nodes have created the initial restart information and initialized the system file.

对于node\_restart和initial\_node\_restart，该阶段启动工作完成，对于initial\_start，当所有节点创建了初始信息和系统文件后，该阶段启动工作完成

For system restarts this is where most of the work is performed, initiated by sending the NDB\_STARTREQ signal from NDBCNTR to DBDIH in the master. This signal is sent when all nodes in the system restart have reached this point in the restart. This we can mark as our first synchronization point for system restarts, designated WAITPOINT\_4\_1.

对于system\_restart，此阶段将完成大部分的工作，在master节点中，NDBCNTR发送GSN\_NDB\_STARTREQ消息到DBDIH，这些消息将被发送到所有节点上，这将是system\_restart中的第一个同步点，称为WAITPOINT\_4\_1

After completing execution of the NDB\_STARTREQ signal, the master sends a CNTR\_WAITREP signal with WAITPOINT\_4\_2 to all nodes. This ends NDB\_STTOR phase 3 as well as (STTOR) Phase 4.

在完成GSN\_NDB\_STARTREQ消息的执行后，master节点会发送一个GSN\_CNTR\_WAITREP消息到所有节点，消息注明WAITPOINT\_4\_2，这将结束NDB\_STTOR阶段3和STTOR阶段4

## STTOR Phase 5

本阶段仅有NDBCNTR触发NDB\_STTOR启动阶段4进行

NDBCNTR

Ndbcntr::execSTTOR

startPhase5Lab(signal);

ph5ALab(signal);

…

## NDB\_STTOR Phase 4

DBDIH

Dbdih::execNDB\_STTOR

switch(typestart)

//initial\_start与system\_restart无工作可做

case NodeState::ST\_INITIAL\_START:

ndbsttorry10Lab(signal, \_\_LINE\_\_);

return;

case NodeState::ST\_SYSTEM\_RESTART:

ndbsttorry10Lab(signal, \_\_LINE\_\_);

return;

//initial\_node\_restart与node\_restart启动START\_MEREQ处理过程

case NodeState::ST\_INITIAL\_NODE\_RESTART:

case NodeState::ST\_NODE\_RESTART:

StartMeReq \* req = (StartMeReq\*)&signal->theData[0];

req->startingRef = reference();

req->startingVersion = 0; // Obsolete

//启动节点向master节点发出START\_MEREQ请求

sendSignal(cmasterdihref, GSN\_START\_MEREQ, signal, StartMeReq::SignalLength, JBB);

return;

}

master节点收到START\_MEREQ请求后进行处理：

Dbdih::execSTART\_MEREQ

//注意，对于master节点来说，这个阶段仅有一个节点允许被启动，其它节点必须排队进入这个过程，因此c\_nodeStartMaster.startNode这个变量代表了对端要求启动的节点，master节点的交互对象也就是这个节点

c\_nodeStartMaster.m\_outstandingGsn = GSN\_COPY\_GCIREQ;

copyGciLab(signal, CopyGCIReq::RESTART\_NR);

if (c\_nodeStartMaster.startNode != RNIL)

c\_COPY\_GCIREQ\_Counter.clearWaitingFor();

c\_COPY\_GCIREQ\_Counter.setWaitingFor(c\_nodeStartMaster.startNode);

//向启动节点拷贝数据

sendCOPY\_GCIREQ(signal, c\_nodeStartMaster.startNode, RNIL);

return;

else

reason = c\_copyGCIMaster.m\_copyReason = c\_copyGCIMaster.m\_waiting[0];

for (Uint32 i = 1; i<CopyGCIMaster::WAIT\_CNT; i++)

c\_copyGCIMaster.m\_waiting[i-1] = c\_copyGCIMaster.m\_waiting[i];

c\_copyGCIMaster.m\_waiting[CopyGCIMaster::WAIT\_CNT-1] = CopyGCIReq::IDLE;

if (reason == CopyGCIReq::IDLE)

return;

sendLoopMacro(COPY\_GCIREQ, sendCOPY\_GCIREQ, RNIL);

sendCOPY\_GCIREQ (signal, specNodePtr.i, RNIL);

master节点向启动节点发送：

Dbdih::sendCOPY\_GCIREQ

const BlockReference ref = calcDihBlockRef(nodeId);

const Uint32 wordPerSignal = CopyGCIReq::DATA\_SIZE; //22字节

const Uint32 noOfSignals = ((Sysfile::SYSFILE\_SIZE32 + (wordPerSignal - 1)) /

wordPerSignal);

CopyGCIReq \* const copyGCI = (CopyGCIReq \*)&signal->theData[0];

for(Uint32 i = 0; i < noOfSignals; i++)

for(Uint32 j = 0; j < wordPerSignal; j++)

//填写启动节点的消息缓冲区，数据来源于sysfileData

copyGCI->data[j] = sysfileData[j+startWord];

//向启动节点发送数据，注意此处的GSN\_COPY\_GCIREQ由master节点发送启动节点

sendSignal(ref, GSN\_COPY\_GCIREQ, signal, 25, JBB);

启动节点收到来自master节点的数据：

Dbdih::execCOPY\_GCIREQ

CopyGCIReq \* const copyGCI = (CopyGCIReq \*)&signal->theData[0];

CopyGCIReq::CopyReason reason = (CopyGCIReq::CopyReason)copyGCI->copyReason;

for(Uint32 i = 0; i<CopyGCIReq::DATA\_SIZE; i++)

cdata[tstart+i] = copyGCI->data[i];

//若拷贝完毕，则启动节点进行本地状态更新，否则要求master节点再进行一轮异步发送过程

if (isdone)

c\_copyGCISlave.m\_expectedNextWord = 0;

else

c\_copyGCISlave.m\_expectedNextWord += CopyGCIReq::DATA\_SIZE;

return;

//此处仅考虑拷贝完毕的情形：

//将得到的数据拷贝至sysfileData

memcpy(sysfileData, cdata, sizeof(sysfileData));

//更新启动节点的节点组信息

setNodeGroups();

c\_copyGCISlave.m\_copyReason = CopyGCIReq::IDLE;

signal->theData[0] = c\_copyGCISlave.m\_senderData;

//向master节点返回确认

sendSignal(c\_copyGCISlave.m\_senderRef, GSN\_COPY\_GCICONF, signal, 1, JBB);

master节点收到来自启动节点的确认后

Dbdih::execCOPY\_GCICONF

//拷贝GCI

startme\_copygci\_conf(signal);

Callback c = { safe\_cast(&Dbdih::lcpBlockedLab),

c\_nodeStartMaster.startNode };

Mutex mutex(signal, c\_mutexMgr, c\_nodeStartMaster.m\_fragmentInfoMutex);

mutex.lock(c, true, true);

//此处通过GSN\_UTIL\_LOCK\_REQ消息向DBUTIL请求lock，得到GSN\_UTIL\_LOCK\_CONF消息确认后，执行器回调Dbdih:: lcpBlockedLab

master节将继续向启动节点拷贝集群信息

Dbdih:: lcpBlockedLab

signal->theData[0] = DihContinueB::ZCOPY\_NODE;

signal->theData[1] = 0;

sendSignal(reference(), GSN\_CONTINUEB, signal, 2, JBB);

Dbdih::execCONTINUEB

copyNodeLab(signal, tableId);

//向启动节点拷贝表信息

tabPtr.p->tabCopyStatus = TabRecord::CS\_COPY\_NODE\_STATE;

signal->theData[0] = DihContinueB::ZPACK\_TABLE\_INTO\_PAGES;

signal->theData[1] = tabPtr.i;

sendSignal(reference(), GSN\_CONTINUEB, signal, 2, JBB);

Dbdih::execCONTINUEB

packTableIntoPagesLab(signal, tableId);

signal->theData[0] = DihContinueB::ZPACK\_FRAG\_INTO\_PAGES;

sendSignal(reference(), GSN\_CONTINUEB, signal, 6, JBB);

Dbdih::execCONTINUEB

packFragIntoPagesLab(signal, &wf);

tableCopyNodeLab(signal, wf->rwfTabPtr);

signal->theData[0] = DihContinueB::ZCOPY\_TABLE\_NODE;

sendSignal(reference(), GSN\_CONTINUEB, signal, 6, JBB);

Dbdih::execCONTINUEB

copyTableNode(signal, &ctn, nodePtr);

for (Uint32 i = 0; i < 16; i++)

//master节点向启动节点拷贝表元数据

sendCopyTable(signal, ctn, calcDihBlockRef(nodePtr.i), reqinfo);

sendSignal(ref, GSN\_COPY\_TABREQ, signal, 21, JBB);

启动节点在收到master节点的表元数据后：

Dbdih::execCOPY\_TABREQ

signal->theData[0] = DihContinueB::ZREAD\_PAGES\_INTO\_TABLE;

signal->theData[1] = tabPtr.i;

sendSignal(reference(), GSN\_CONTINUEB, signal, 2, JBB);

Dbdih::execCONTINUEB

readPagesIntoTableLab(signal, tableId);

//分配表分片

allocFragments(noOfFrags, rf.rwfTabPtr);

signal->theData[0] = DihContinueB::ZREAD\_PAGES\_INTO\_FRAG;

sendSignal(reference(), GSN\_CONTINUEB, signal, 5, JBB);

Dbdih::execCONTINUEB

readPagesIntoFragLab(signal, &rf);

getFragstore(rf->rwfTabPtr.p, rf->fragId, fragPtr);

readFragment(rf, fragPtr);

readReplicas(rf, fragPtr);

//若拷贝完毕

copyTabReq\_complete(signal, rf->rwfTabPtr);

//更新节点信息

//向master节点返回GSN\_COPY\_TABCONF消息，确认表信息读入

sendSignal(cmasterdihref, GSN\_COPY\_TABCONF, signal, 2, JBB);

//若未拷贝完毕，则需要更新本地表

signal->theData[0] = DihContinueB::ZTABLE\_UPDATE;

sendSignal(reference(), GSN\_CONTINUEB, signal, 2, JBB);

启动节点更新过程：

Dbdih::execCONTINUEB

tableUpdateLab(signal, tabPtr);

//创建表文件

createFileRw(signal, filePtr);

master节点收到启动节点的确认GSN\_COPY\_TABCONF消息：

Dbdih::execCOPY\_TABCONF

//对于node\_restart

if (getNodeState().startLevel >= NodeState::SL\_STARTED)

//继续拷贝下一个表，重新进入master节点的Dbdih::execCONTINUEB

signal->theData[0] = DihContinueB::ZCOPY\_NODE;

signal->theData[1] = tableId + 1;

sendSignal(reference(), GSN\_CONTINUEB, signal, 2, JBB);

当master节点的所有表都拷完时：

Dbdih::execCONTINUEB

copyNodeLab(signal, tableId);

dihCopyCompletedLab(signal);

//通知启动节点的DBDICT可以开始启动

sendSignal(ref, GSN\_DICTSTARTREQ,

signal, DictStartReq::SignalLength, JBB);

启动节点的DBDICT收到GSN\_DICTSTARTREQ消息后：

Dbdict::execDICTSTARTREQ

//启动节点开始表元数据同步过程

sendSignal(dictRef, GSN\_GET\_SCHEMA\_INFOREQ, signal, 1, JBB);

**TODO：分析DBDICT对GSN\_GET\_SCHEMA\_INFOREQ的后续处理过程**

启动节点从master节点同步了所有的表元数据之后，返回给master节点一个GSN\_GET\_SCHEMA\_INFOCONF消息，master节点的后续处理：

Dbdih::execDICTSTARTCONF

//此时，启动节点已经从master节点拷贝了所有的DBDIH和DBDICT信息，此时master节点需要将启动节点加入到所有的分布式协议中

nodeDictStartConfLab(signal);

master节点将启动节点加入以下BLOCK的分布式协议中：DBDIH、DBLQH、DBTC、DBDICT、BACKUP、SUMA、DBSPJ

**TODO：分析DBTC对将启动节点加入各分布式协议的处理过程**

最终，master节点将启动节点加入到所有的分布式协议中后，向启动节点返回GSN\_START\_MECONF消息：

Dbdih::execUNBLO\_DICTCONF

const Uint32 wordPerSignal = StartMeConf::DATA\_SIZE;

const int noOfSignals = ((Sysfile::SYSFILE\_SIZE32 + (wordPerSignal - 1)) / wordPerSignal);

Uint32 nodeId = startMe->startingNodeId = c\_nodeStartMaster.startNode;

startMe->startWord = 0;

const Uint32 ref = calcDihBlockRef(c\_nodeStartMaster.startNode);

for(int i = 0; i < noOfSignals; i++){

{

const int startWord = startMe->startWord;

for(Uint32 j = 0; j < wordPerSignal; j++)

startMe->data[j] = sysfileData[j+startWord];

}

sendSignal(ref, GSN\_START\_MECONF, signal, StartMeConf::SignalLength, JBB);

startMe->startWord += wordPerSignal;

}

启动节点在收到master节点的GSN\_START\_MECONF消息后：

Dbdih::execSTART\_MECONF

setNodeActiveStatus();

setNodeGroups();

//node\_restart与initial\_node\_restart在此阶段的工作完成

ndbsttorry10Lab(signal, \_\_LINE\_\_);

回到NDBCNTR中，注意，由于在ph5Alab函数中，额外执行了一条语句cstartPhase = cstartPhase + 1，导致这里讲进入ZSTART\_PHASE\_6标号，这也意味着进入了NDB\_STTOR启动阶段5：

Ndbcntr::execNDB\_STTORRY

ph6ALab(signal);

Some initialization of local checkpoint variables takes place in this phase, and for initial restarts, this is all that happens in this phase.

本阶段将有一些本地检查点变量被初始化，这也是initial\_start在本阶段的主要工作。

For system restarts, all required takeovers are also performed. Currently, this means that all nodes whose states could not be recovered using the redo log are restarted by copying to them all the necessary data from the “live” data nodes.

对于system\_restart，需要额外的接管工作。当前，这意味着，所有不能由redolog所恢复的节点，将会被重启，其所需要的所有数据将拷贝自正常的节点。

For node restarts and initial node restarts, the master node performs a number of services, requested to do so by sending the START\_MEREQ signal to it. This phase is complete when the master responds with a START\_MECONF message, and is described in Section 7.5.22, “START\_MEREQ Handling”.

对于node\_restart和initial\_node\_restart，master节点将发送GSN\_START\_MEREQ消息，来启动一系列服务。master节点响应GSN\_START\_MECONF消息标志着这个阶段完成。

After ensuring that the tasks assigned to DBDIH tasks in the NDB\_STTOR phase 4 are complete, NDBCNTR performs some work on its own.

当DBDIH 在NDB\_STTOR启动阶段4的工作完成后，NDBCNTR将启动一些自己的工作。

For initial starts, it creates the system table that keeps track of unique identifiers such as those used for AUTO\_INCREMENT.

对于initial\_start，master节点将创建一些系统表以记录一些唯一标识符（如AUTO\_INCREMENT）。

Following the WAITPOINT\_4\_1 synchronization point, all system restarts proceed immediately to NDB\_STTOR phase 5, which is handled by the DBDIH block. See Section 7.5.13, “NDB\_STTOR Phase 5”, for more information.

在同步点WAITPOINT\_4\_1之后，DBDIH所有的system\_restart工作立即进入NDB\_STTOR阶段5

GSN\_START\_MEREQ启动，仅对node\_restart和initial\_node\_restart有效：

The first step in handling START\_MEREQ is to ensure that no local checkpoint is currently taking place; otherwise, it is necessary to wait until it is completed. The next step is to copy all distribution information from the master DBDIH to the starting DBDIH. After this, all metadata is synchronized in DBDICT (see Section 7.5.21, “System Restart Handling in Phase 4”).

master节点处理GSN\_START\_MEREQ消息，第一步是确保当前没有本地检查点，否则，需要等待直到该检查点完成。第二步是从master节点的DBDIH处拷贝所有的分布式信息到启动节点的DBDIH。第三步从master节点将所有的元数据同步到启动节点DBDICT中。

After blocking local checkpoints, and then synchronizing distribution information and metadata information, global checkpoints are blocked.

禁用了本地检查点后，将同步集群分布式和元数据信息，全局检查点也将被禁用。

The next step is to integrate the starting node in the global checkpoint protocol, local checkpoint protocol, and all other distributed protocols. As part of this the node status is also updated.

随后将启动节点加入全局检查点协议，本地检查点协议以及其他分布式协议。启动节点状态也将被更新。

After completing this step the global checkpoint protocol is permitted to start again, the START\_MECONF signal is sent to indicate to the starting node that the next phase may proceed.

完成这些工作后，全局检查点将被再次启用，master节点向启动节点响应GSN\_START\_MECONF消息后，启动节点可以继续处理一下的工作。

## NDB\_STTOR Phase 5

NDBCNTR

Ndbcntr::execSTTOR

startPhase5Lab(signal);

ph5ALab(signal);

Ndbcntr::ph5Alab

//启动NDB\_STTOR阶段4，此前已经执行完毕

if (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

return;

//此处增加cstartPhase，其值为6

cstartPhase = cstartPhase + 1;

cinternalStartphase = cstartPhase - 1;

//master节点initial\_start、system\_restart的工作：

if (getOwnNodeId() == cmasterNodeId)

switch(ctypeOfStart)

//对于intial\_start，master节点需要创建所有的系统表

case NodeState::ST\_INITIAL\_START:

beginSchemaTransLab(signal);

return;

//对于system\_restart

case NodeState::ST\_SYSTEM\_RESTART:

waitpoint52Lab(signal);

return;

case NodeState::ST\_NODE\_RESTART:

case NodeState::ST\_INITIAL\_NODE\_RESTART:

break;

case NodeState::ST\_ILLEGAL\_TYPE:

break;

}

}

NdbSttor \* const req = (NdbSttor\*)signal->getDataPtrSend();

switch(ctypeOfStart)

//master节点、非master节点node\_restart、initial\_node\_restart的工作，直接进入NDB\_STTOR启动阶段5

case NodeState::ST\_NODE\_RESTART:

case NodeState::ST\_INITIAL\_NODE\_RESTART:

req->senderRef = reference();

req->nodeId = getOwnNodeId();

req->internalStartPhase = cinternalStartphase;

req->typeOfStart = cdihStartType;

req->masterNodeId = cmasterNodeId;

sendSignal(DBDIH\_REF, GSN\_NDB\_STTOR, signal, NdbSttor::SignalLength, JBB);

return;

//非master节点initial\_start、system\_restart的工作

case NodeState::ST\_INITIAL\_START:

case NodeState::ST\_SYSTEM\_RESTART:

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_5\_2;

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_WAITREP, signal, 2, JBB);

return;

对于几种启动类型在该阶段的工作进行的总结：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | initial\_start | system\_restart | node\_restart | initial\_node\_restart |
|  | 系统级 | 系统级 | 节点级 | 节点级 |
| STTOR启动阶段5 | 本节点DBDIH进入NDB\_STTOR启动阶段4 | 本节点DBDIH进入NDB\_STTOR启动阶段4 | 本节点DBDIH进入NDB\_STTOR启动阶段4 | 本节点DBDIH进入NDB\_STTOR启动阶段4 |
| NDB\_STTOR启动阶段4 |  |  | 启动START\_MEREQ过程拷贝DBDIH（分布式节点信息）和DBDICT（表元数据）信息至本节点 | 启动START\_MEREQ过程拷贝DBDIH（分布式节点信息）和DBDICT（表元数据）信息至本节点 |
|  | master节点创建所有系统表 | master节点等待同步点WAITPOINT\_5\_2 |  |  |
|  | 非master节点向master节点报告同步点WAITPOINT\_5\_2 | 非master节点向master节点报告同步点WAITPOINT\_5\_2 |  |  |
|  | 到达同步点WAITPOINT\_5\_2 | 到达同步点WAITPOINT\_5\_2 |  |  |
|  | 本节点DBDIH进入NDB\_STTOR启动阶段5 | 本节点DBDIH进入NDB\_STTOR启动阶段5 | 本节点DBDIH进入NDB\_STTOR启动阶段5 | 本节点DBDIH进入NDB\_STTOR启动阶段5 |
| NDB\_STTOR启动阶段5 |  |  | 本节点DBDIH进入GSN\_START\_MEREQ启动过程 | 本节点DBDIH进入GSN\_START\_MEREQ启动过程 |
|  |  |  | 启动START\_COPYREQ过程，拷贝表分片数据至本节点 | 启动START\_COPYREQ过程，拷贝表分片数据至本节点 |

initial\_start工作简述：

主要包括：准备表事务，创建哈希映射，创建表，创建日志组、表空间、undolog文件、数据文件，结束表事务，准备schema表项插入事务，schema表项批量插入，结束schema表项插入事务

Ndbcntr::beginSchemaTransLab

sendSignal(DBDICT\_REF, GSN\_SCHEMA\_TRANS\_BEGIN\_REQ, signal,

SchemaTransBeginReq::SignalLength, JBB);

**TODO：分析DBDICT对GSN\_SCHEMA\_TRANS\_BEGIN\_REQ的后续处理过程**

DBDICT返回GSN\_SCHEMA\_TRANS\_BEGIN\_CONF后，系统表创建事务准备完成：

Ndbcntr::execSCHEMA\_TRANS\_BEGIN\_CONF

//创建哈希映射

createHashMap(signal, 0);

sendSignal(DBDICT\_REF, GSN\_CREATE\_HASH\_MAP\_REQ, signal,

CreateHashMapReq::SignalLength, JBB);

**TODO：分析DBDICT对GSN\_CREATE\_HASH\_MAP\_REQ的后续处理过程**

DBDICT返回GSN\_CREATE\_HASH\_MAP\_CONF后，哈希映射创建完成：

Ndbcntr::execCREATE\_HASH\_MAP\_CONF

createSystableLab(signal, 0);

//需要创建的系统表，包括：sys/def/SYSTAB\_0、sys/def/NDB$EVENTS\_0，数据结构为g\_sysTable\_SYSTAB\_0、g\_sysTable\_NDBEVENTS\_0

const SysTable& table = \*g\_sysTableList[index];

//向DBDICT申请建表

sendSignal(DBDICT\_REF, GSN\_CREATE\_TABLE\_REQ, signal,

CreateTableReq::SignalLength, JBB, ptr, 1);

**TODO：分析DBDICT对GSN\_CREATE\_TABLE\_REQ的后续处理过程**

DBDICT返回GSN\_CREATE\_TABLE\_CONF后，一张表创建完成：

Ndbcntr::execCREATE\_TABLE\_CONF

//继续创建其它表

createSystableLab(signal, conf->senderData + 1);

当所有表创建完成后：

Ndbcntr::createSystableLab

//创建其它数据库对象，这些对象根据配置得到，主要是日志组对象和表空间对象

createDDObjects(signal, 0);

const ddentry\* entry = &f\_dd[index];

switch(entry->type)

//创建日志组和表空间对象

case DictTabInfo::LogfileGroup:

case DictTabInfo::Tablespace:

sendSignal(DBDICT\_REF, GSN\_CREATE\_FILEGROUP\_REQ, signal,

CreateFilegroupReq::SignalLength, JBB, ptr, 1);

//创建undolog文件和数据文件对象

case DictTabInfo::Undofile:

case DictTabInfo::Datafile:

sendSignal(DBDICT\_REF, GSN\_CREATE\_FILE\_REQ, signal,

CreateFileReq::SignalLength, JBB, ptr, 1);

**TODO：分析DBDICT对GSN\_CREATE\_FILEGROUP\_REQ的后续处理过程**

**TODO：分析DBDICT对GSN\_CREATE\_FILE\_REQ的后续处理过程**

DBDICT返回GSN\_CREATE\_FILEGROUP\_CONF|GSN\_CREATE\_FILE\_CONF后，一个文件组|文件对象创建完成：

Ndbcntr::execCREATE\_FILEGROUP\_CONF

//继续创建其它对象

createDDObjects(signal, conf->senderData + 1);

Ndbcntr::execCREATE\_FILE\_CONF

//继续创建其它对象

createDDObjects(signal, conf->senderData + 1);

当所有对象创建完毕后：

Ndbcntr::createDDObjects

//完成事务

endSchemaTransLab(signal);

sendSignal(DBDICT\_REF, GSN\_SCHEMA\_TRANS\_END\_REQ, signal,

SchemaTransEndReq::SignalLength, JBB);

**TODO：分析DBDICT对GSN\_SCHEMA\_TRANS\_END\_REQ的后续处理过程**

DBDICT返回GSN\_SCHEMA\_TRANS\_END\_CONF后，schema对象创建事务完成：

Ndbcntr::execSCHEMA\_TRANS\_END\_CONF

startInsertTransactions(signal);

sendSignal(DBTC\_REF, GSN\_TCSEIZEREQ, signal, 2, JBB);

**TODO：分析DBTC对GSN\_TCSEIZEREQ的后续处理过程**

DBTC返回GSN\_TCSEIZECONF后

Ndbcntr::execTCSEIZECONF

crSystab7Lab(signal);

//每次最多处理RowsPerCommit（16）行，在得到DBTC返回的GSN\_TCKEYCONF后，由crSystab8Lab再次进入

for (Tmp = ckey; Tmp <= guard0; Tmp++)

sendSignal(ctcReference, GSN\_TCKEYREQ, signal, TcKeyReq::StaticLength + 6, JBB);

**TODO：分析DBTC对GSN\_TCKEYREQ的后续处理过程**

DBTC返回GSN\_TCKEYCONF后

Ndbcntr::execTCKEYCONF(Signal\* signal)

crSystab8Lab(signal);

//若还有剩余行，则返回crSystab7Lab继续发送

if (ckey < ZSIZE\_SYSTAB)

crSystab7Lab(signal);

return;

//全部处理完毕后，释放事务

sendSignal(ctcReference, GSN\_TCRELEASEREQ, signal, 2, JBB);

**TODO：分析DBTC对GSN\_TCRELEASEREQ的后续处理过程**

DBTC返回GSN\_TCRELEASEONF后

Ndbcntr::execTCRELEASECONF(Signal\* signal)

//同样进入waitpoint52Lab，以进入下一个阶段

waitpoint52Lab(signal);

system\_restart工作简述：

waitpoint52Lab(signal);

initial\_start与system\_restart都将进入waitpoint52Lab：

Ndbcntr::waitpoint52Lab

//由于所有非master节点在initial\_start与system\_restart都将直接向master节点发送消息GSN\_CNTR\_WAITREP，且同步点为CntrWaitRep::ZWAITPOINT\_5\_2

//当所有节点的GSN\_CNTR\_WAITREP到达后，master节点将通知DBDIH开始NDB\_STTOR阶段5，至此，所有启动类型都将进入NDB\_STTOR阶段5

if (cnoWaitrep == cnoStartNodes)

sendSignal(DBDIH\_REF, GSN\_NDB\_STTOR, signal,

NdbSttor::SignalLength, JBB);

DBDIH

Dbdih::execNDB\_STTOR

switch(typestart)

//对于initial\_start及system\_restart，若非master节点，则直接进入NDB\_STTOR启动阶段6，若为master节点，DBDIH检查DBTC是否已经启动

case NodeState::ST\_INITIAL\_START:

case NodeState::ST\_SYSTEM\_RESTART:

if (!isMaster())

ndbsttorry10Lab(signal, \_\_LINE\_\_);

return;

checkLcpStart(signal, \_\_LINE\_\_);

return;

//对于node\_restart与initial\_node\_restart，启动节点的DBDIH产生一个GSN\_START\_COPYREQ消息，从其它节点处拷贝表分片数据到本节点

case NodeState::ST\_NODE\_RESTART:

case NodeState::ST\_INITIAL\_NODE\_RESTART:

StartCopyReq\* req = (StartCopyReq\*)signal->getDataPtrSend();

req->senderRef = reference();

req->senderData = RNIL;

req->flags = StartCopyReq::WAIT\_LCP;

req->startingNodeId = getOwnNodeId();

if (!ndb\_pnr(getNodeInfo(refToNode(cmasterdihref)).m\_version))

sendSignal(cmasterdihref, GSN\_START\_COPYREQ, signal, tartCopyReq::SignalLength, JBB);

else

sendSignal(reference(), GSN\_START\_COPYREQ, signal, StartCopyReq::SignalLength, JBB);

return;

Dbdih::execSTART\_COPYREQ

startTakeOver(signal, startNodeId, takeOverNode, &req);

nr\_start\_fragments(signal, takeOverPtr);

**TODO：分析DBDIH的数据拷贝（takeover）过程**

For initial starts and system restarts this phase means executing a local checkpoint. This is handled by the master so that the other nodes will return immediately from this phase.

对于initial\_start及system\_restart，这个阶段将执行本地检查点，这个过程由master节点处理，其它节点将立即从这个阶段返回。

Node restarts and initial node restarts perform the copying of the records from the primary replica to the starting replicas in this phase. Local checkpoints are enabled before the copying process is begun.

对于node\_restart与initial\_node\_restart，启动节点将从主副本处拷贝所有的记录到本地。拷贝过程开始后，本地检查点将被启用。

Copying the data to a starting node is part of the node takeover protocol. As part of this protocol, the node status of the starting node is updated; this is communicated using the global checkpoint protocol. Waiting for these events to take place ensures that the new node status is communicated to all nodes and their system files.

拷贝数据到启动节点时节点接管协议的一部分。执行协议的过程中，启动节点的状态通过全局检查点协议进行更新。这一步完成后，其它节点将得到新节点的状态和系统文件信息。

After the node's status has been communicated, all nodes are signaled that we are about to start the takeover protocol for this node. Part of this protocol consists of Steps 3 - 9 during the system restart phase as described below. This means that restoration of all the fragments, preparation for execution of the redo log, execution of the redo log, and finally reporting back to DBDIH when the execution of the redo log is completed, are all part of this process.

节点状态广播完毕后，其它节点将为新节点执行接管协议。这个协议由下述的3-9阶段组成，用于system\_restart。这个过程包括恢复所有分片，redolog执行准备，redolog执行，以及redolog最终执行完成后向DBDIH的报告。

After preparations are complete, copy phase for each fragment in the node must be performed. The process of copying a fragment involves the following steps:

准备完毕后，各个节点将执行每个分片的拷贝过程。这个拷贝过程包括下述步骤：

1.The DBLQH kernel block in the starting node is informed that the copy process is about to begin by sending it a PREPARE\_COPY\_FRAGREQ signal.

一个GSN\_PREPARE\_COPY\_FRAGREQ消息将通知启动节点的DBLQH准备分片拷贝过程。

2.When DBLQH acknowledges this request a CREATE\_FRAGREQ signal is sent to all nodes notify them of the preparation being made to copy data to this replica for this table fragment.

完成拷贝这个表分片数据的准备后，启动节点的DBLQH将回复其它所有节点一个GSN\_CREATE\_FRAGREQ消息，通知他们准备完毕。

3.After all nodes have acknowledged this, a COPY\_FRAGREQ signal is sent to the node from which the data is to be copied to the new node. This is always the primary replica of the fragment. The node indicated copies all the data over to the starting node in response to this message.

当所有节点都得到启动节点的确认后，启动节点将向一个节点发送GSN\_COPY\_FRAGREQ消息，以从该节点处拷贝数据。拷贝的总是分片的主副本。主副本节点将分片数据拷贝至启动节点处。

4.After copying has been completed, and a COPY\_FRAGCONF message is sent, all nodes are notified of the completion through an UPDATE\_TOREQ signal.

拷贝过程完成后，主副本节点向启动节点回复一个GSN\_COPY\_FRAGCONF消息，其它节点也将通过GSN\_ UPDATE\_TOREQ消息得到拷贝完成的通知。

5.After all nodes have updated to reflect the new state of the fragment, the DBLQH kernel block of the starting node is informed of the fact that the copy has been completed, and that the replica is now up-to-date and any failures should now be treated as real failures.

当所有节点都得知并更新了分片的新状态后，启动节点的DBLQH便认为这个分片拷贝过程完成了，其拥有的副本已经是最新的，之后的任何失败操作也是真正的失败操作。

6.The new replica is transformed into a primary replica if this is the role it had when the table was created.

如果有新表创建，新副本有可能被转换为主副本。

7.After completing this change another round of CREATE\_FRAGREQ messages is sent to all nodes informing them that the takeover of the fragment is now committed.

完成了上述过程后，启动节点再次将一个新的GSN\_CREATE\_FRAGREQ消息发送到所有节点上，以通知他们，启动节点的接管过程已经提交了。

8.After this, process is repeated with the next fragment if another one exists.

之后，这个拷贝过程将再次重复，目标换成下一个分片。

9.When there are no more fragments for takeover by the node, all nodes are informed of this by sending an UPDATE\_TOREQ signal sent to all of them.

当没有任何分片需要接管后，启动节点向所有其它节点发送一个GSN\_UPDATE\_TOREQ消息。

10.Wait for the next complete local checkpoint to occur, running from start to finish.

等待下一个本地检查点启动，然后同步地完成。

11.The node states are updated, using a complete global checkpoint. As with the local checkpoint in the previous step, the global checkpoint must be permitted to start and then to finish.

一个完整的全局检查点将更新启动节点的状态。与前述本地检查点建立过程一样，全局检查点也必须同步地启动和完成。

12.When the global checkpoint has completed, it will communicate the successful local checkpoint of this node restart by sending an END\_TOREQ signal to all nodes.

当全局检查点完成后，启动节点将向其它节点发送一个GSN\_END\_TOREQ消息，通知node\_restart过程成功地建立了本地检查点。

13.A START\_COPYCONF is sent back to the starting node informing it that the node restart has been completed.

其它节点将向启动节点返回一个GSN\_START\_COPYCONF消息，告知node\_restart过程完成。

14.Receiving the START\_COPYCONF signal ends NDB\_STTOR phase 5. This provides another synchronization point for system restarts, designated as WAITPOINT\_5\_2.

启动节点接收到GSN\_START\_COPYCONF消息后，将结束NDB\_STTOR启动阶段5。这将进入同步点WAITPOINT\_5\_2。

Note

The copy process in this phase can in theory be performed in parallel by several nodes. However, all messages from the master to all nodes are currently sent to single node at a time, but can be made completely parallel. This is likely to be done in the not too distant future.

注意，这个阶段的拷贝过程理论上来说可以并行进行。但是，当前由master向所有节点发送的所有消息都将一次一个地被发出，这个过程完全可以被改作并行的。

In an initial and an initial node restart, the SUMA block requests the subscriptions from the SUMA master node. NDBCNTR executes NDB\_STTOR phase 6. No other NDBCNTR activity takes place.

对于initial\_start和initial\_node\_restart，启动节点的SUMA从主节点的SUMA取得所有订阅。NDBCNTR执行NDB\_STTOR启动阶段6。

## NDB\_STTOR Phase 6

该阶段的启动有些特殊，将由NDB\_STTOR启动阶段5完成后产生的GSN\_NDB\_STTORRY消息触发：

Ndbcntr::execNDB\_STTORRY

ph6ALab(signal);

Ndbcntr::ph6Alab

//对于node\_restart与initial\_node\_restart，直接进入后续处理

if ((ctypeOfStart == NodeState::ST\_NODE\_RESTART) ||

(ctypeOfStart == NodeState::ST\_INITIAL\_NODE\_RESTART))

waitpoint51Lab(signal);

return;

//对于initial\_start与system\_restart，进入ZWAITPOINT\_5\_1同步点

NodeReceiverGroup rg(NDBCNTR, c\_start.m\_starting);

rg.m\_nodes.clear(getOwnNodeId());

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_5\_1;

//此处是向节点组发送

sendSignal(rg, GSN\_CNTR\_WAITREP, signal, 2, JBB);

waitpoint51Lab(signal);

Ndbcntr::waitpoint51Lab

//此处再次增加cstartPhase，其值为7

cstartPhase = cstartPhase + 1;

cinternalStartphase = cstartPhase - 1;

cndbBlocksCount = 0;

ph6BLab(signal);

Ndbcntr::ph6Blab

//启动NDB\_STTOR启动阶段6

if (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

return;

//当所有BLOCK的NDB\_STTOR启动阶段6完成后，对于node\_restart与initial\_node\_restart，直接进入STTOR启动阶段6，而对于initial\_start与system\_restart，则需要进入同步点ZWAITPOINT\_6\_1，再进入STTOR启动阶段6

if ((ctypeOfStart == NodeState::ST\_NODE\_RESTART) ||

(ctypeOfStart == NodeState::ST\_INITIAL\_NODE\_RESTART))

sendSttorry(signal);

return;

waitpoint61Lab(signal);

//仅有initial\_start及system\_restart进入此处：

Ndbcntr::waitpoint61Lab

//此处为initial\_start及system\_restart

//对于master节点，需要等待包括其自己在内的所有节点到齐后，进入同步点ZWAITPOINT\_6\_2，再进入STTOR启动阶段6

if (getOwnNodeId() == cmasterNodeId)

cnoWaitrep6++;

if (cnoWaitrep6 == cnoStartNodes)

NodeReceiverGroup rg(NDBCNTR, c\_start.m\_starting);

rg.m\_nodes.clear(getOwnNodeId());

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_6\_2;

sendSignal(rg, GSN\_CNTR\_WAITREP, signal, 2, JBB);

sendSttorry(signal);

//对于非master节点，将向master节点通知同步点ZWAITPOINT\_6\_1

else

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_6\_1;

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_WAITREP, signal, 2, JBB);

此时master节点向其它所有节点通知同步点ZWAITPOINT\_6\_2到达，其它节点得知此事件后，进入STTOR启动阶段6：

Ndbcntr::execCNTR\_WAITREP

sendSttorry(signal);

至此，所有启动类型\*所有节点类型都进入了STTOR启动阶段6

DBLQH

Dblqh::execNDB\_STTOR

startphase6Lab(signal);

//重设启动阶段和启动类型

cstartPhase = ZNIL;

cstartType = ZNIL;

sendNdbSttorryLab(signal);

DBDICT

Dbdict::execNDB\_STTOR

//重设启动阶段和启动类型

c\_initialStart = false;

c\_systemRestart = false;

c\_initialNodeRestart = false;

c\_nodeRestart = false;

sendNDB\_STTORRY(signal);

DBTUP

Dbtup::execNDB\_STTOR

//标注系统重启完成

csystemStart = SSS\_TRUE;

DBACC

Dbacc::execNDB\_STTOR

//无动作

DBTC

Dbtc::execNDB\_STTOR

//标注系统重启完成

csystemStart = SSS\_TRUE;

DBDIH

Dbdih::execNDB\_STTOR

switch(typestart)

//对于initial\_start及system\_restart，master节点需要启动一个全局检查点

case NodeState::ST\_INITIAL\_START:

case NodeState::ST\_SYSTEM\_RESTART:

if(isMaster())

startGcp(signal);

ndbsttorry10Lab(signal, \_\_LINE\_\_);

return;

//对于node\_restart及initial\_node\_restart，直接进入STTOR启动阶段6

case NodeState::ST\_NODE\_RESTART:

case NodeState::ST\_INITIAL\_NODE\_RESTART:

ndbsttorry10Lab(signal, \_\_LINE\_\_);

return;

Dbdih::startGcp

//启动全局检查点执行过程

signal->theData[0] = DihContinueB::ZSTART\_GCP;

sendSignal(reference(), GSN\_CONTINUEB, signal, 1, JBB);

//启动全局检查点监视计时器

startGcpMonitor(signal);

signal->theData[0] = DihContinueB::ZCHECK\_GCP\_STOP;

sendSignalWithDelay(reference(), GSN\_CONTINUEB, signal, 100, 1);

**TODO：分析DBDIH的全局检查点建立过程**

In this NDB\_STTOR phase, both DBLQH and DBDICT clear their internal representing the current restart type. The DBACC block resets the system restart flag; DBACC and DBTUP start a periodic signal for checking memory usage once per second. DBTC sets an internal variable indicating that the system restart has been completed.

在此阶段，DBLQH与DBDICT将清除他们内部对当前启动类型的记录。DBACC重设重启类型。DBACC和DBTUP启动一个以1s为周期的消息以检查内存使用。DBTC设置一个内部变量以表示system\_restart完成。

## STTOR Phase 6

NDBCNTR

Ndbcntr::execSTTOR

//取得节点组信息

getNodeGroup(signal);

EXECUTE\_DIRECT(DBDIH, GSN\_CHECKNODEGROUPSREQ, signal,

CheckNodeGroups::SignalLength);

Dbdih::execCHECKNODEGROUPSREQ

Uint32 ng = Sysfile::getNodeGroup(getOwnNodeId(), SYSFILE->nodeGroups);

sendSttorry(signal);

DBUTIL

DbUtil::execSTTOR

//获取SYSTAB\_0的表id，主要工作包括：

//1.取得SYSTAB\_0的表id；2.执行硬编码的prepare工作；3.调用connectTc()；4.返回STTORRY。

get\_systab\_tableid(signal);

sendSignal(DBDICT\_REF, GSN\_GET\_TABINFOREQ, signal,

GetTabInfoReq::SignalLength, JBB, ptr,1);

DBDICT返回GSN\_GET\_TABINFO\_CONF后：

DbUtil::execGET\_TABINFO\_CONF

hardcodedPrepare(signal, conf->tableId);

//其它硬编码准备工作

//连接DBTC

connectTc(signal);

while(c\_seizingTransactions.seize(ptr))

signal->theData[0] = ptr.i << 1; // See TcCommitConf

signal->theData[1] = reference();

sendSignal(DBTC\_REF, GSN\_TCSEIZEREQ, signal, 2, JBB);

DbUtil::execTCSEIZECONF

sendSignal(NDBCNTR\_REF, GSN\_STTORRY, signal, 6, JBB);

The NDBCNTR block defines the cluster's node groups, and the DBUTIL block initializes a number of data structures to facilitate the sending keyed operations can be to the system tables. DBUTIL also sets up a single connection to the DBTC kernel block.

NDBCNTR定义了集群节点组，DBUTIL初始化一些数据结构以辅助简化对系统表的操作。DBUTIL也建立一个到DBTC的连接。

## STTOR Phase 7

DBTUX

Dbtux::execSTTOR

c\_statMon.m\_loopIndexId = 0;

statMonSendContinueB(signal);

signal->theData[0] = TuxContinueB::StatMon;

signal->theData[1] = mon.m\_loopIndexId;

//启动一个状态监视计时器

sendSignalWithDelay(reference(), GSN\_CONTINUEB,

signal, mon.m\_loopDelay, 2);

Dbtux::statMonExecContinueB

statMonCheck(signal, mon);

for (uint loop = 0; loop < maxloop; loop++, mon.m\_loopIndexId++)

if (update)

statMonRep(signal, mon);

sendSignal(DBDICT\_REF, GSN\_INDEX\_STAT\_REP,

signal, IndexStatRep::SignalLength, JBB);

mon.m\_loopIndexId++;

break;

statMonSendContinueB(signal);

QMGR

Qmgr::execSTTOR

//QMGR的工作仅与president相关

cactivateApiCheck = 1;

if (cpresident == getOwnNodeId())

switch(arbitRec.method){

case ArbitRec::DISABLED:

break;

case ArbitRec::METHOD\_EXTERNAL:

case ArbitRec::METHOD\_DEFAULT:

handleArbitStart(signal);

break;

Qmgr::handleArbitStart

arbitRec.state = ARBIT\_INIT;

arbitRec.newstate = true;

//启动一个新的continueB线程，线程id将递增，旧有线程将退出

//线程运行函数为：runArbitMgr\_C

//同时启动对API节点的心跳检测

startArbitThread(signal);

BACKUP

Backup::execSTTOR

//设置磁盘写速度

m\_curr\_disk\_write\_speed = c\_defaults.m\_disk\_write\_speed;

//initial\_start启动时，对于master节点，向DBUTIL请求产生下一次的备份id

if(startphase == 7 && g\_TypeOfStart == NodeState::ST\_INITIAL\_START &&

c\_masterNodeId == getOwnNodeId() && !isNdbMtLqh())

createSequence(signal);

req->senderData = RNIL;

req->sequenceId = NDB\_BACKUP\_SEQUENCE;

req->requestType = UtilSequenceReq::Create;

sendSignal(DBUTIL\_REF, GSN\_UTIL\_SEQUENCE\_REQ,

signal, UtilSequenceReq::SignalLength, JBB);

return;

DbUtil::execUTIL\_SEQUENCE\_REQ

runTransaction(signal, transPtr);

SUMA

Suma::execSTTOR

//设置启动状态

//initial\_start启动时，对于master节点，向DBUTIL请求产生SUMA的序列id

if(m\_typeOfStart == NodeState::ST\_INITIAL\_START &&

c\_masterNodeId == getOwnNodeId())

createSequence(signal);

req->senderData = RNIL;

req->sequenceId = SUMA\_SEQUENCE;

req->requestType = UtilSequenceReq::Create;

sendSignal(DBUTIL\_REF, GSN\_UTIL\_SEQUENCE\_REQ,

signal, UtilSequenceReq::SignalLength, JBB);

DBUG\_VOID\_RETURN;

## STTOR Phase 8

CMVMI

Cmvmi::execSTTOR

//设置启动状态

globalData.theStartLevel = NodeState::SL\_STARTED;

sendSTTORRY(signal);

NDBCNTR

此阶段仅用于启动NDB\_STTOR启动阶段7

Ndbcntr::execSTTOR

startPhase8Lab(signal);

ph7ALab(signal);

Ndbcntr::ph7Alab

//启动NDB\_STTOR启动阶段7

while (cndbBlocksCount < ZNO\_NDB\_BLOCKS)

sendNdbSttor(signal);

return;

//对于node\_restart及initial\_node\_restart，立即进入STTOR启动阶段9

if ((ctypeOfStart == NodeState::ST\_NODE\_RESTART) ||

(ctypeOfStart == NodeState::ST\_INITIAL\_NODE\_RESTART))

sendSttorry(signal);

return;

//对于initial\_start及system\_restart，进入同步点ZWAITPOINT\_7\_2

waitpoint71Lab(signal);

Ndbcntr::waitpoint71Lab

if (getOwnNodeId() == cmasterNodeId)

cnoWaitrep7++;

if (cnoWaitrep7 == cnoStartNodes)

NodeReceiverGroup rg(NDBCNTR, c\_start.m\_starting);

rg.m\_nodes.clear(getOwnNodeId());

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_7\_2;

sendSignal(rg, GSN\_CNTR\_WAITREP, signal, 2, JBB);

sendSttorry(signal);

else

signal->theData[0] = getOwnNodeId();

signal->theData[1] = CntrWaitRep::ZWAITPOINT\_7\_1;

sendSignal(calcNdbCntrBlockRef(cmasterNodeId), GSN\_CNTR\_WAITREP, signal, 2, JBB);

master节点通知其他节点共同进入STTOR启动阶段9

## NDB\_STTOR Phase 7

各个BLOCK均无动作

DBDICT

Dbdict::execNDB\_STTOR

sendNDB\_STTORRY(signal);

If this is a system restart, the master node initiates a rebuild of all indexes from DBDICT during this phase.

对于system\_restart，master节点初始化所有的DBDICT索引。

The CMVMI kernel block opens communication channels to the API nodes (including MySQL servers acting as SQL nodes). Indicate in globalData that the node is started.

CMVMI打开到API节点的通道，在globalData中标识节点已启动。

## STTOR Phase 9

NDBCNTR

Ndbcntr::execSTTOR

startPhase9Lab(signal);

cinternalStartphase = cstartPhase - 1;

cndbBlocksCount = 0;

ph8ALab(signal);

sendSttorry(signal);

//重设一些启动变量

resetStartVariables(signal);

cnoStartNodes = 0;

cnoWaitrep6 = cnoWaitrep7 = 0;

## STTOR Phase 101

SUMA

Suma::execSTTOR

//对于node\_restart和initial\_node\_restart，SUMA进入移交过程，进行一次GCP协商，该GCP用作参考点，在集群的事件订阅和复制协议中加入启动节点（在其他节点的事件源和复制协议中加入启动节点）

if (m\_typeOfStart == NodeState::ST\_NODE\_RESTART ||

m\_typeOfStart == NodeState::ST\_INITIAL\_NODE\_RESTART)

c\_startup.m\_wait\_handover= true;

check\_start\_handover(signal);

DBUG\_VOID\_RETURN;

# 其它