HSQLDB的事务机制分析

文庆福
2011013239 thssvince@163.com
清华大学软件学院11班

2014年 4月 6日

目录

1 实验简介

1.1 实验背景

数据库事务（transaction）一词看似比较陌生，其实在我们的数据库操作中常常会用到。例如我们在网上购物的时候，但我们点击购买物品并扣款的时候，这时候数据库的操作就应该是一个事务。从用户的账户扣除金额，减少物品数量，生成购买记录等操作应该是一个完整的过程，不可拆分。所以说，数据库事务就是由一系列数据库操作组成的一个完整的逻辑过程，要么完整地执行，要么完全不执行。

事务具有 ACID 四个特性：原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

- 原子性：一个事务（transaction）中的所有操作，要么全部完成，要么全部不完成，不会结束在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
- 一致性：在事务开始之前和事务结束以后，数据库的完整性没有被破坏。这表示写入的资料必须完全符合所有的默认规则，这包含资料的精确度、串联性以及后续数据库可以自发性地完成预定的工作。
- 隔离性：当两个或者多个事务并发访问（此处访问指查询和修改的操作）数据库的同一数据时所表现出的相互关系。事务隔离分为不同级别，包括读未提交（Read uncommitted）、读提交（read committed）、可重复读（repeatable read）和串行化（Serializable）。
- 持久性：在事务完成以后，该事务对数据库所作的更改便持久地保存在数据库之中，并且是完全的。

对事务的以上四大特性，一致性是由程序员保证的，原子性、隔离性与持久性是由 DBMS 来实现，本次实验我们将主要分析数据库的原子性，隔离性和持久性将在接下来的两次实验中再做分析。

1.2 实验环境

- 操作系统：Windows 8 企业版
- JDK: OpenJDK 7（64-Bit）
- HSQLDB: 2.3.1

- IDE: Eclipse Standard(Kepler Service Release 1)

2 HSQLDB 事务机制概述

HSQLDB 中事务处理部分由数据库实例的事务管理器 TransactionManager 进行管理，事务管理器和 session 相互配合共同事务处理的工作。

事务处理的核心是 TransactionManager，TransactionManager 接口提供了一系列事务处理的基本操作供使用，session 类调用 TransactionManager 接口提供的方法并进行封装，运用这些封装方法进行事务处理。HSQLDB 中事务处理的基本单元是操作 action。一个 action 是一个 RowAction 类。它也是事务处理中记录操作过程的基本单位。

HSQLDB 通过出错回滚的机制保证事务的原子性。一旦遇到不可处理的错误，该事务中所有进行过的操作都将进行回滚。HSQLDB 支持三种并发控制模型，分别是默认的两阶段封锁（2PL），多版本并发控制（MVCC）及多版本两阶段封锁（MV2PL）模式。对于每种模式，HSQLDB 支持 SQL 标准中的四个事务隔离级别：未提交读（READ UNCOMMITTED）、已提交读（READ COMMITTED）、可重复读（REPEATABLE READ）和可串行化（SERIALIZABLE）。

2.1 事务管理器

HSQLDB 中 TransactionManager 接口定义了一些列的事务基本操作，下面重点关注三个函数。

```
1  public void beginTransaction(Session session) {
2
3          if (!session.isTransaction) {
4              session.actionTimestamp       = getNextGlobalChangeTimestamp();
5              session.transactionTimestamp = session.actionTimestamp;
6              session.isTransaction         = true;
7
8              transactionCount++;
9          }
10     }
11
12 public boolean commitTransaction(Session session) {
13
14         if (session.abortTransaction) {
15             return false;
16         }
17
```

```
18          writeLock.lock();
19
20          try {
21              int limit = session.rowActionList.size();
22
23              // new actionTimestamp used for commitTimestamp
24              session.actionTimestamp          = getNextGlobalChangeTimestamp();
25              session.transactionEndTimestamp = session.actionTimestamp;
26
27              endTransaction(session);
28
29              for (int i = 0; i < limit; i++) {
30                  RowAction action = (RowAction) session.rowActionList.get(i);
31
32                  action.commit(session);
33              }
34
35              adjustLobUsage(session);
36              persistCommit(session);
37              endTransactionTPL(session);
38          } finally {
39              writeLock.unlock();
40          }
41
42          session.tempSet.clear();
43
44          return true;
45      }
46
47  public void rollback(Session session) {
48
49          session.abortTransaction         = false;
50          session.actionTimestamp          = getNextGlobalChangeTimestamp();
51          session.transactionEndTimestamp = session.actionTimestamp;
52
53          rollbackPartial(session, 0, session.transactionTimestamp);
54          endTransaction(session);
55          writeLock.lock();
56
57          try {
58              endTransactionTPL(session);
59          } finally {
60              writeLock.unlock();
61          }
```

```
62      }
```

- beginTransaction：该方法将对 TransactionManager 做一些初始化工作，以便事务处理能正常工作。
- commitTransaction：该方法提交事务，做一些收尾工作，标明事务处理结束。
- rollback：该方法实现了回滚操作。

’ 可以说，事务管理器是事务机制的核心，事务处理中用到的方法均定义在管理器中。共有三个类实现了 org.hsqldb.TransactionManager 接口：

- TransactionManager2PL
- TransactionManagerMVCC
- transactionManagerMV2PL

这些类分别对应上文提到的三种事务处理模型，隔离级别也由这三种管理器实现。

2.2 事务处理流程

我们知道，HSQLDB 中，SQL 语句的执行是由 org.hsqldb.Session 类控制的，每条语句的执行位于 executeCompiledStatement 方法。这部分的工作流程：

```
1  public Result executeCompiledStatement(Statement cs, Object[] pvals,
2                                         int timeout) {
3          Result r;
4          if (abortTransaction) {
5              rollback(false);
6              return Result.newErrorResult(Error.error(ErrorCode.X_40001));
7          }
8          ...
9          boolean isTX = cs.isTransactionStatement();
10         if (!isTX) {
11             ...
12             r                                  = cs.execute(this);
13             sessionContext.currentStatement = null;
14             return r;
15         }
16         while (true) {
17             ...
18             if (abortTransaction) {
```

```
19                  rollback(false);
20                  sessionContext.currentStatement = null;
21                  return Result.newErrorResult(Error.error(ErrorCode.X_40001));
22              }
23              timeoutManager.startTimeout(timeout);
24              try {
25                  latch.await();
26              } catch (InterruptedException e) {
27                  abortTransaction = true;
28              }
29              boolean abort = timeoutManager.endTimeout();
30              if (abort) {
31                  r = Result.newErrorResult(Error.error(ErrorCode.X_40502));
32                  endAction(r);
33                  break;
34              }
35              if (abortTransaction) {
36                  rollback(false);
37                  sessionContext.currentStatement = null;
38                  return Result.newErrorResult(Error.error(ErrorCode.X_40001));
39              }
40              database.txManager.beginActionResume(this);
41              //          tempActionHistory.add("sql execute " + cs.sql + " "
42              //  + actionTimestamp + " " + rowActionList.size());
43              sessionContext.setDynamicArguments(pvals);
44              r = cs.execute(this);
45              if (database.logger.getSqlEventLogLevel()
46                      >= SimpleLog.LOG_NORMAL) {
47                  database.logger.logStatementEvent(this, cs, pvals, r,
48                                                    SimpleLog.LOG_NORMAL);
49              }
50              lockStatement = sessionContext.currentStatement;
51              //          tempActionHistory.add("sql execute end "
52              //  + actionTimestamp + " " + rowActionList.size());
53              endAction(r);
54              if (abortTransaction) {
55                  rollback(false);
56                  sessionContext.currentStatement = null;
57                  return Result.newErrorResult(Error.error(r.getException(),
58                          ErrorCode.X_40001, null));
59              }
60
61              if (redoAction) {
62                  redoAction = false;
```

```
63                  try {
64                      latch.await();
65                  } catch (InterruptedException e) {
66                      abortTransaction = true;
67                  }
68              } else {
69                  break;
70              }
71          }
72          ...
73          return r;
74      }
```

这里保留了 executeCompiledStatement 方法的框架。我们看到，第 10 行，方法首先判断当前处理的语句是否为事务性的，如果不是，则直接处理，否则进入第 17-72 行的循环： 这里的循环实际上是为第 63 行的 redoAction 设计的，如果没有 redoAction，我们发现，这里的处理过程实际上分为如下几步：

- 判断是否需要中断事务。
- 等待并发事务的结束。
- 再次判断是否需要中断当前事务（因为处理并发事务可能会引发新异常）。
- 执行语句。
- 第三次判断是否需要中断当前事务（因为执行中可能引发异常）。
- 判断是否需要重新执行当前语句（对于 INSERT 和 DELETE 语句，可能会由于并发事务的冲突导致失败，这将在下次实验中详细分析）

由此，我们知道，事务处理的总体流程是，在任何可能产生异常的地方进行检测，如果遇到异常，则执行回滚操作。

3 事务的原子性

3.1 新建事务

首先，在默认状况下，HSQLDB 会将一个 session 的 isAutoCommit 域设置为 true，这样实际上每一条语句都被视为一个 transaction。

```
1  if (sessionContext.depth == 0
2          && (isAutoCommit || cs.isAutoCommitStatement())) {
```

```
3       try {
4           if (r.isError()) {
5               rollback(false);
6           } else {
7               commit(false);
8           }
9       } catch (Exception e) {
10          currentStatement = null;
11          return Result.newErrorResult(
12          Error.error(ErrorCode.X_40001));
13      }
14  }
```

在每一个 executeCompiledStatement 方法中，由于 isAutoCommit 被置为 true，故每一次都会进行 commit 操作。在 commit 方法中，首先会调用 database.txManager 的 commitTransaction 方法提交此次事务。后面会对 commitTransaction 操作进行详细分析。

接下来的 endTransaction 方法将会清除该 transaction 的各个记录。包括 savepoints, rowActionList, transaction tables, transactionNavigators 等。另外，这里会将只读属性、隔离模式、锁声明等都设为默认值。

```
1   private void endTransaction(boolean commit, boolean chain) {
2           sessionContext.savepoints.clear();
3           sessionContext.savepointTimestamps.clear();
4           rowActionList.clear();
5           sessionData.persistentStoreCollection.clearTransactionTables();
6           sessionData.closeAllTransactionNavigators();
7           sessionData.clearLobOps();
8           lockStatement = null;
9           logSequences();
10
11          if (!chain) {
12              sessionContext.isReadOnly = isReadOnlyDefault ? Boolean.TRUE
13                                                            : Boolean.FALSE;
14              setIsolation(isolationLevelDefault);
15          }
16
17          Statement endTX = commit ? StatementSession.commitNoChainStatement
18                                   : StatementSession.rollbackNoChainStatement;
19          if (database.logger.getSqlEventLogLevel() > 0) {
20              database.logger.logStatementEvent(this, endTX, null,
21                                                Result.updateZeroResult,
22                                                SimpleLog.LOG_ERROR);
23          }
```

```
24  /* debug 190
25          tempActionHistory.add("commit ends " + actionTimestamp);
26          tempActionHistory.clear();
27  // */
28      }
```

从这里也可以看出，HSQLDB 中一个 transaction 的关键要素有 savepoints、rowActionList、transactionTables、transactionNavigators、isReadOnly、isolationMode、lockStatement 等。由于设置了 autoCommit 之后，每一行语句运行完毕就 endTransaction，故为了分析 transaction 机制，必须将该域设置为 false。

3.2 事务性语句的执行

3.2.1 插入语句

```
1  CREATE TABLE testtable (id INTEGER );
2  SET AUTOCOMMIT FALSE ;
3  INSERT INTO testtable VALUES (1);
```

采用以上的 SQL 语句为例执行，插入语句的执行入口位于 org.hsqldb.StatementInsert 类的 getResult 方法。该方法在初始化后（包括初始化存储器和列信息、准备数据等），并在检查约束和触发器后，调用 org.hsqldb.Table 类的 insertSingleRow 方法处理插入单行操作。

```
1  Row insertSingleRow(Session session, PersistentStore store, Object[] data,
2                      int[] changedCols) {
3      ...
4      Row row = (Row) store.getNewCachedObject(session, data, true);
5      session.addInsertAction(this, store, row, changedCols);
6      return row;
7  }
```

insertSingleRow 也会做一些检查，这里，store.getNewCachedObject 方法（实际上是接口的具体实现）会根据表的类型（MEMORY, TEXT, CACHED），新建一个具体类的 Row 对象，并将相应的数据装入 Row。这里以 Memory Table 为例：

```
1  public CachedObject getNewCachedObject(Session session, Object object,
2      boolean tx) {
3      int id;
4      synchronized ( this ) {
5          id = rowIdSequence ++;
6      }
```

```
7       Row row = new RowAVL(table , (Object []) object , id , this );
8       if (tx) {
9           RowAction action = new RowAction (session , table ,
10                                            RowAction .ACTION_INSERT , row ,
11                                            null );
12          row. rowAction = action;
13      }
14      return row;
15  }
```

第 3-6 行生成一个表中行的唯一的 id。

第 7 行利用 RowAVL 类的构造方法得到行对象 row。

第 8-13 行给这个行添加了 RowAction 对象，标明当前执行的是插入操作。其中 RowAction 类继承自 RowActionBase 类，表示一个行（Row）的插入、删除、回滚或提交的行为及相关操作。RowAction 类别包括 ACTION_NONE、ACTION_INSERT、ACTION_DELETE 等。

在得到填充过数据和 Action 信息的 Row 对象后，insertSingleRow 方法将调用 session 对象的 addInsertAction 方法将行插入，这个方法会调用 TransactionManager 的同名方法 addInsertAction 执行插入操作。和之前提到的 beginAction 方法一样，这里的 ddInsertAction 也随事务管理器的不同而不同。取其主要部分分析：

```
1   public void addInsertAction(Session session, Table table,
2                               PersistentStore store, Row row,
3                               int[] changedColumns) {
4       RowAction action = row.rowAction;
5       if (action == null) {
6           throw Error.runtimeError(ErrorCode.GENERAL_ERROR,
7                                    "null insert action ");
8       }
9       store.indexRow(session, row);
10      session.rowActionList.add(action);
11      row.rowAction = null;
12  }
```

第 9 行，该方法调用 store 对象的 indexRow 方法，将 row 插入到表中。

第 10 行，将这个 action 加入 session 对象的 rowActionList 中，为之后的回滚做好准备。

3.2.2 删除语句

执行如下 SQL 语句调试分析：

```
DELETE FROM testtable
```

DELETE 语句是在 org.hsqldb.StatementDML 类中处理的，和 INSERT 语句一样，在删除前，还需要做一些准备工作，包括生成 navigator，检查触发器等。最终删除数据的操作在该类的 delete 方法中执行：

```
1  int delete(Session session , Table table ,
2          RowSetNavigatorDataChange navigator ) {
3      int rowCount = navigator .getSize ();
4      ......
5      for ( int i = 0; i < navigator .getSize (); i++) {
6      Row row = navigator . getNextRow ();
7      Table currentTable = (( Table) row.getTable ());
8      session. addDeleteAction (currentTable , row , null );
9      ......
10     }
11     ......
12     return rowCount;
13 }
```

我们看到，这里的行为与 org.hsqldb.Table 类的 insertSingleRow 的行为类似，首先准备好 row 对象，然后调用 session 对象的 addDeleteAction 方法删除行。这个方法同样会调用事务管理器的同名方法继续执行。

这里，不同的管理器采取了不同的方法进行处理。由于代码都很简单，这里不再列出，仅列出三种管理器的策略：

- 2PL: 从表中删除，并记录此 Action
- MVCC: 只记录此 Action，不删除物理数据
- MV2PL: 只记录此 Action，不删除物理数据

3.3 事务的保存

分别执行如下 SQL 语句调试分析：

SAVEPOINT name;

RELEASE SAVEPOINT name;

保存点是由 org.hsqldb.Session 类的 savepoint 方法实现的：

```
1  public synchronized void savepoint(String name) {
2      int index = sessionContext.savepoints.getIndex(name);
3      if (index != -1) {
4          sessionContext.savepoints.remove(name);
5          sessionContext.savepointTimestamps.remove(index);
6      }
```

```
7         sessionContext.savepoints.add(name,
8                                         ValuePool.getInt(rowActionList.size()));
9         sessionContext.savepointTimestamps.addLast(actionTimestamp);
10    }
```

我们看到，这里首先删除可能已经存在的同名保存点。同时记录当前保存点的名称和 rowActionList 的长度组成的键值对至 savepoints 这个 hash 表中。接下来记录当前 action 的时间戳。

在删除保存点时，是在 org.hsqldb.Session 类的 releaseSavepoint 方法中处理：

```
1   public synchronized void releaseSavepoint (String name) {
2       // remove this and all later savepoints
3       int index = sessionContext . savepoints .getIndex(name );
4       if (index < 0) {
5           throw Error.error( ErrorCode .X_3B001 , name );
6       }
7       while ( sessionContext . savepoints .size () > index) {
8           sessionContext . savepoints .remove( sessionContext . savepoints .size ()
9                                                   - 1);
10          sessionContext . savepointTimestamps . removeLast ();
11      }
12  }
```

显然，此方法不仅删除给定的保存点，还删除其后的所有保存点。

3.4 事务的回滚

回滚分为两类，一类是回滚整个事务，另一类是回滚至某个保存点。无论是哪一种回滚，都将调用某个事务管理器的 rollbackPartial 方法进行回滚：

回滚整个事务，将调用 rollbackPartial(session, 0, session.transactionTimestamp);

对于回滚至某个保存点，将调用 rollbackPartial(session, start, timestamp);

rollbackPartial 方法用于回滚从 rowActionList 指定下标和某个时间戳开始的全部 Action。

三个管理器的实现不尽相同，但流程基本是一致的，这里以 2PL 为例进行说明：

```
1   public void rollbackPartial(Session session, int start, long timestamp) {
2       int limit = session.rowActionList.size();
3       if (start == limit) {
4           return;
5       }
6       for (int i = limit - 1; i >= start; i--) {
7           RowAction action = (RowAction) session.rowActionList.get(i);
8           if (action == null || action.type == RowActionBase.ACTION_NONE
```

```
9                   || action.type == RowActionBase.ACTION_DELETE_FINAL) {
10              continue;
11          }
12          Row row = action.memoryRow;
13          if (row == null) {
14              row = (Row) action.store.get(action.getPos(), false);
15          }
16          if (row == null) {
17              continue;
18          }
19          action.rollback(session, timestamp);
20          int type = action.mergeRollback(session, timestamp, row);
21          action.store.rollbackRow(session, row, type, txModel);
22      }
23      session.rowActionList.setSize(start);
24  }
```

对于列表中指定范围内的每个 RowAction，得到其 Row 对象（第 18-26 行）。

接下来，调用 action 对象的 rollback 方法，标记 action 的 commitTimestamp等字段。

第 30 行，action 对象的 mergeRollback 方法，合并部分 Action 对象。

第 32 行，回滚操作最终在 store 对象中的 rollbackRow 方法中执行：

```
1   public void rollbackRow(Session session, Row row, int changeAction,
2       int txModel) {
3       switch ( changeAction ) {
4           case RowAction . ACTION_DELETE :
5               if (txModel == TransactionManager .LOCKS) {
6                   (( RowAVL) row ). setNewNodes ( this );
7                   indexRow(session , row );
8               }
9               break ;
10          case RowAction . ACTION_INSERT :
11              if (txModel == TransactionManager .LOCKS) {
12                  delete(session , row );
13                  remove(row.getPos ());
14              }
15              break ;
16      ......
17      }
18  }
```

显然，这里的操作是显然的，对于删除，将行重新插入；对于插入，删除行。

然而，MVCC 模型的处理略有不同，之前提到，对于 MVCC/MV2PL，删除操作并

未真正执行，因此，MVCC/MV2PL 的最终回滚是在 commitRow 方法中处理的：

```
1  public void commitRow(Session session, Row row, int changeAction,
2                        int txModel) {
3      Object[] data = row.getData();
4      switch (changeAction) {
5          case RowAction.ACTION_DELETE:
6          database.logger.writeDeleteStatement(session, (Table) table,
7                                               data);
8          break;
9      case RowAction.ACTION_INSERT:
10         database.logger.writeInsertStatement(session, row,
11                                              (Table) table);
12         break;
13     case RowAction.ACTION_INSERT_DELETE:
14         // INSERT + DELETE
15         break;
16     case RowAction.ACTION_DELETE_FINAL:
17         delete(session, row);
18         break;
19     }
20 }
```

从名字看，这个方法用于处理事务的提交。这里先分析跟回滚有关的部分：对于 MVCC/MV2PL，插入操作的回滚会被置为 ACTION_DELETE_FINAL，因此，会在这里执行真正的删除操作。

3.5 事务的提交

事务的提交通过 COMMIT 语句执行。这是在各个管理器的 commitTransaction 方法中处理的。和之前一样， commitTransaction 方法的实现也不尽相同，大致流程如下：

1. 调用 endTransaction 方法，置 session 的事务状态为 false，同时事务计数器减一。

2. 遍历每一个 Action，执行 commit。对于 2PL，从刚才列出的代码里可以发现，插入和删除只需记录日志即可。对于 MVCC/MV2PL，还需要进行额外的操作，persistCommit 方法会填充 action 里的信息，这样再调用commitRow 的时候就会顺利删除了。

3. 最后，调用 endTransactionTPL 方法根据管理器的实现方法做一些收尾工作，例如释放锁等。

4 问题总结

4.1 HSQLDB实现了哪几种隔离级别？

HSQLDB 实现了四个隔离级别：

- 可串行化（SERIALIZABLE）
- 可重复读（REPEATABLE READ）
- 已提交读（READ COMMITTED）
- 未提交读（READ UNCOMMITTED）

4.2 在每一个事务中，数据操作时如何进行的？

见 3.2 节

4.3 如何实现回滚和提交操作？

见 3.4 节 和 3.5 节

4.4 保存点是如何实现的？

见 3.3 节