HSQLDB 的并发控制机制分析

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

2014年 4月 13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，上次实验我们主要分析了数据库的原子性，本次实验我们将主要分析数据库的隔离性和持久性。

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 Java的多线程机制

Java 语言提供了对于线程很好的支持。对于方法重入的保护，信号量（semaphore）和临界区（critical section）机制的实现都非常简洁。可以很容易的实现多线程间的同步操作从而保护关键数据的一致性。

Java 中内置了对于对象并发访问的支持，每一个对象都有一个监视器（monitor），同时只允许一个线程持有监视器从而进行对对象的访问，那些没有获得监视器的线程必须等待直到持有监视器的线程释放监视器。对象通过 synchronized 关键字来声明线程必须获得监视器才能进行对自己的访问。

synchronized 声明仅仅对于一些较为简单的线程间同步问题比较有效，对于复杂的同步问题，比如带有条件的同步问题， Java 提供了另外的解决方法，wait/notify/notifyAll。获得对象监视器的线程可以通过调用该对象的 wait 方法主动释放监视器，等待在该对象的线程等待队列上，此时其他线程可以得到监视器从而访问该对象，之后可以通过调用 notify/notifyAll 方法来唤醒先前因调用 wait 方法而等待的线程。一般情况下，对于 wait/notify/notifyAll 方法的调用都是根据一定的条件来进行的，比如：经典的生产者/消费者问题中对于队列空、满的判断。使用 wait/notify/notifyAll 可以很容易的实现 POSIX 中的一个线程间的高级同步技术：条件变量。

Java 中实现多线程可以使用两种方式：

- 继承Thread 类 写一个自定义的类继承 Thread 类，然后重写 run() 方法。使用的时候new 一个继承了 Thread 的类的实例，调用其 start() 方法。比如：

```
1 class MyThread extends Thread{
2     public void run(){}
3 }
4 MyThread myThread = new MyThread();
5 myThread.start();
```

- 实现 Runnable 接口 写一个自定义的类实现 Runnable 接口，然后实现 run() 方法。使用的时候实例化实现了Runnable 的类，然后将这一实例作为参数传到 Thread 的构造方法 Thread(Runnable r) 中，从而创建 Thread 实例，然后调用其 start() 方法即可。比如：

```
1 class MyRunnable implements Runnable {
2 public void run(){}
```

```
3  }
4  MyRunnable myRunnable = new MyRunnable();
5  Thread thread = new Thread(myRunnable);
6  thread.start();
```

在 hsqldb 中 ServerConnection 类实现了 Runnable 接口，ServerThread 类（Server 类的内部类）继承了 Thread 类。

Java 的两个关于多线程的关键字：

- synchronized 关键字限定同步语句块。一个对象可以有多个同步语句块，JVM 给每个对象一个内部锁，当有线程试图进入同步语句块时，JVM 检查该对象的锁是否打开，若是，则允许该线程进入，并将锁锁住，当线程退出同步语句块时，锁再打开。如果检查时锁是锁住的，则当前线程被阻塞起来，进入该对象的等待集进行等待。如果线程调用 notify 方法，则从对象的等待集中取出一个线程进入就绪态接受 JVM 调度。
- volatile 关键字在一定程度上保证了线程间的数据一致性。由于 JVM 对不同线程分配了各自的寄存器、堆栈、缓存等独立资源，线程对共享数据的修改不一定能及时反映到主存中 (有可能被缓存起来但没有写回内存)，因此不同线程读取相同数据时有可能出现不一致的情况。volatile 关键字的使用保证了这个问题不会出现，对被 volatile 关键字修饰的变量的任何修改都将直接写回内存，由于各个线程之间内存资源是共享的，线程间的数据一致性就得到了一定的保证。
- wait 方法显式调用，用于将当前线程放入被使用该对象（包含该 wait 调用的代码所在的对象）的等待集。然后当前线程进入阻塞态。
- notify 方法用于从对象的等待集中唤醒一个线程重新进入就绪态。
- sleep 方法用于使当前线程进入阻塞态开始睡眠，当睡眠时间到则线程进入就绪态接受 JVM 调度。

3 HSQLDB的多线程

org.hsqldb.server.Server 类中，run 方法会调用 handleConnection 方法处理每一个传入连接。handleConnection 方法将创建 ServerConnection 对象，并将 ServerConnection 加入线程组。因为 ServerConnection 类实现了 Runnable 接口，因此接下来调用 ServerConnection 类的 start 方法即可启动新的线程。ServerConnection 类的 run 方法首先调用 init 方法进行一些初始化工作，包括设定流、创建 session 等。run 方法的核心部分代码如下：

```
while ( keepAlive ) {
    msgType = dataInput .readByte ();
    if (msgType < ResultConstants . MODE_UPPER_LIMIT ) {
        receiveResult (msgType );
    } else {
        receiveOdbcPacket (( char ) msgType );
    }
}
```

可见，对每个请求，都有一个线程对其处理。在线程管理方面，HSQLDB 大量使用了锁来完成，这将在接下来进行详细分析。

4 HSQLDB的隔离机制

HSQLDB 支持 SQL92 标准中的四个隔离级别：未提交读（READ UNCOMMITTED）、已提交读（READ COMMITTED）、可重复读（REPEATABLE READ）和可串行化（SERIALIZABLE）。

	脏读	不可重复读	幻象
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

在 HSQLDB 中虽然定义了下面四种隔离级别，但却只是作为 READ COMMITTED 和 SERIALIZABLE 两种隔离级别来处理的，默认隔离级别是 READ COMMITTED。

```
int TX_READ_UNCOMMITTED = 1;
int TX_READ_COMMITTED   = 2;
int TX_REPEATABLE_READ  = 4;
int TX_SERIALIZABLE     = 8;
```

```
static String getIsolationString(int isolationLevel) {
    switch (isolationLevel) {
        case SessionInterface.TX_READ_UNCOMMITTED :
        case SessionInterface.TX_READ_COMMITTED :
            StringBuffer sb = new StringBuffer();
            sb.append(Tokens.T_READ).append(' ');
            sb.append(Tokens.T_COMMITTED);
            return sb.toString();
```

```
            case SessionInterface.TX_REPEATABLE_READ :
            case SessionInterface.TX_SERIALIZABLE :
            default :
                return Tokens.T_SERIALIZABLE;
        }
    }
```

5 HSQLDB的2PL协议

5.1 概述

两阶段封锁协议是保证冲突可串行性的一个并发控制协议。该协议要求事务分两个阶段提出加锁和解锁申请，第一阶段是增长阶段，事务可以获得锁但不能释放锁，第二阶段是缩减阶段，事务可以释放锁但不能获得锁。

两阶段封锁的一个变体是严格连接段封锁协议，它要求事务持有的所有排他锁必须在事务提交后方可释放。另一个变体是强两阶段封锁协议，它要求事务提交之前不释放任何锁。锁转换机制可以使得在需要的时候将共享锁升级为排他锁或将排他锁降级为共享锁，从而获得更灵活的并发机制。不过锁升级只能在增长阶段，锁降级只能在缩减阶段。

HSQLDB 中实现的是强两阶段封锁协议，在commit 之前不释放任何锁，只有在执行commit 的过程中，通过把当前session 占有的锁从TransactionManager2PL 的 tableWriteLocks 和 tableReaderLocks 中移除而释放其排他写锁和共享读锁。

5.2 加锁

开始执行每个 Action 前，HSQLDB 会首先执行 beginAction 方法。

```
public void beginAction (Session session , Statement cs) {
    ......
    writeLock .lock ();
    try {
        ......
        boolean canProceed = setWaitedSessionsTPL (session , cs);
        if ( canProceed ) {
            if (session.tempSet.isEmpty ()) {
                lockTablesTPL (session , cs);
            } else {
                setWaitingSessionTPL (session );
```

```
12                 }
13             }
14         } finally {
15             writeLock .unlock ();
16         }
17     }
```

首先，代码段加锁，防止重入。

这里有两个方法：setWaitedSessionsTPL，setWaitingSessionTPL。第 8 行调用 setWaitedSessionsTPL 方法检查锁，并判断能否处理当前语句。

```
1  boolean setWaitedSessionsTPL (Session session , Statement cs) {
2      session.tempSet.clear ();
3      ......
4      HsqlName [] nameList = cs. getTableNamesForWrite ();
5      for ( int i = 0; i < nameList .length; i++) {
6          HsqlName name = nameList[i];
7          ......
8          Session holder = (Session) tableWriteLocks .get(name );
9          if (holder != null && holder != session) {
10             session.tempSet.add(holder );
11         }
12         Iterator it = tableReadLocks .get(name );
13         while (it.hasNext ()) {
14             holder = (Session) it.next ();
15             if (holder != session) {
16                 session.tempSet.add(holder );
17             }
18         }
19     }
20     nameList = cs. getTableNamesForRead ();
21     ......
22     for ( int i = 0; i < nameList .length; i++) {
23         HsqlName name = nameList[i];
24         ......
25         Session holder = (Session) tableWriteLocks .get(name );
26         if (holder != null && holder != session) {
27             session.tempSet.add(holder );
28         }
29     }
30     if (session.tempSet.isEmpty ()) {
31         return true ;
32     }
33     if ( checkDeadlock (session , session.tempSet )) {
```

```
34            return true ;
35        }
36        session.tempSet.clear ();
37        session. abortTransaction = true ;
38        return false ;
39    }
```

第 4-19 行处理要写的表，把所有正在读写本语句将要写的表的 session 均加入到 tempSet 中。

第 20-29 行处理要读的表，把所有正在写本语句将要读的表的 session 加入到 tempSet 中。

第 30-32 行，如果 tempSet 为空，则可以处理。

第 33-35 行，如果没有死锁，则可以处理。死锁的处理将在稍后分析。

第 36-38 行是不能处理的情形，将中断事务。

回到 setWaitedSessionsTPL 方法，如果能处理，则判断 tempSet 是否为空。若 tempSet 为空，则表明能正常处理，调用 lockTablesTPL 方法给表加锁：

```
1   void lockTablesTPL (Session session , Statement cs) {
2       ......
3       HsqlName [] nameList = cs. getTableNamesForWrite ();
4       for ( int i = 0; i < nameList .length; i++) {
5           HsqlName name = nameList[i];
6           ......
7           tableWriteLocks .put(name , session );
8       }
9       nameList = cs. getTableNamesForRead ();
10      for ( int i = 0; i < nameList .length; i++) {
11          HsqlName name = nameList[i];
12          ......
13          tableReadLocks .put(name , session );
14      }
15  }
```

这里主要是遍历要读和写的表名，将表名和 session 的键值对加入锁表中。值得注意的是，这里的锁表是定义在 database 级别上的。

如果 setWaitedSessionsTPL 方法中，tempSet 不为空，表明还需等待其他会话结束，但无死锁。因此，调用 setWaitingSessionTPL 方法，将当前 session 加入互斥会话的 waitingSessions 中，并设置当前会话的等待计数器：

```
1   void setWaitingSessionTPL (Session session) {
2       int count = session.tempSet.size ();
3       for ( int i = 0; i < count; i++) {
4           Session current = (Session) session.tempSet.get(i);
```

```
            current. waitingSessions .add(session );
        }
        session.tempSet.clear ();
        session.latch.setCount(count );
    }
```

加锁处理完毕后，会有三种结果：

1. 加锁成功，latch 不会阻塞当前语句的执行。
2. 加锁失败，但没有死锁，latch 会组则当前语句的执行。
3. 加锁失败，有死锁，调用 rollback 方法回滚。

5.3 释放锁

事务完成（提交或回滚）后，都会调用 TransactionManager2PL.endTransactionTPL()方法。该方法将 unlockedCount 的初值设为 0，调用 unlockTablesTPL()方法释放锁。

```
void endTransactionTPL(Session session) {

    unlockTablesTPL(session);

    final int waitingCount = session.waitingSessions.size();

    if (waitingCount == 0) {
        return;
    }

    resetLocks(session);
    resetLatches(session);
}
```

unlockTablesTPL()方法这里遍历两个锁表 tableWriteLocks 和 tableReadLocks，删掉所有和当前 session 关联的锁。

```
void unlockTablesTPL(Session session) {
    Iterator it = tableWriteLocks.values().iterator();
    while (it.hasNext()) {
        Session s = (Session) it.next();
        if (s == session) {
            it.remove();
        }
```

```
        }
        it = tableReadLocks.values().iterator();
        while (it.hasNext()) {
            Session s = (Session) it.next();
            if (s == session) {
                it.remove();
            }
        }
    }
```

之后再调用 resetLocks 来释放锁：

```
    void resetLocks(Session session) {
        final int waitingCount = session.waitingSessions.size();
        for (int i = 0; i < waitingCount; i++) {
            Session current = (Session) session.waitingSessions.get(i);
            current.tempUnlocked = false;
            long count = current.latch.getCount();
            if (count == 1) {
                boolean canProceed = setWaitedSessionsTPL(current,
                    current.sessionContext.currentStatement);
                if (canProceed) {
                    if (current.tempSet.isEmpty()) {
                        lockTablesTPL(current,
                                      current.sessionContext.currentStatement);
                        current.tempUnlocked = true;
                    }
                }
            }
        }
        for (int i = 0; i < waitingCount; i++) {
            Session current = (Session) session.waitingSessions.get(i);
            if (current.tempUnlocked) {
                //
            } else if (current.abortTransaction) {
                //
            } else {
                // this can introduce additional waits for the sessions
                setWaitedSessionsTPL(current,
                                     current.sessionContext.currentStatement);
            }
        }
    }
```

首先，方法遍历所有等待中的 session，如果某一 session 的 latch.getCount()== 1 则说明只有当前处理 session 阻塞了该 session。 这样，像 beginAction 方法那样调用 setWaitedSessionsTPL 方法重新判断锁的情况。如果没有死锁，则对这个 session 加锁，并置其 tempUnlocked 字段为 true。

然后，再次遍历所有等待中的 session，如果上一轮没有处理（即 tempUnlocked为 false），则再次检查死锁。释放锁的最后一个步骤是调用 resetLathcesMidTransaction 方法。该方法将对所有等待中的 session 执行 setWaitingSessionTPL 方法处理。目的是重设和它相关的 session 的 waitingSessions 及计数器。

最后 endTransactionTPL 调用 resetLatches 方法重设每个等待 session 的等待情况和计数器。

5.4 死锁判断

HSQLDB 会在 checkDeadlock 方法中检测死锁情况：

```
1  boolean checkDeadlock(Session session, OrderedHashSet newWaits) {
2      int size = session.waitingSessions.size();
3      for (int i = 0; i < size; i++) {
4          Session current = (Session) session.waitingSessions.get(i);
5          if (newWaits.contains(current)) {
6              return false;
7          }
8          if (!checkDeadlock(current, newWaits)) {
9              return false;
10         }
11     }
12     return true;
13 }
```

这个方法接受两个参数，一个是要判断的 session A，另一个是新加入的互斥的 session 列表。

这个方法的做法很直观：如果所有直接或简介等待 session A 的 session 在互斥列表中，这就产生了循环依赖，因此会造成死锁。

6 问题总结

6.1 Java 是如何实现线程之间的同步和通信的？

Java 线程间同步和通信既可以通过简单的 synchronized 和 volatile 关键字实现，还有 wait/notify/notifyAll, sleep 等，具体可以见第 2 节的分析。

6.2 HSQLDB 是如何采用多线程机制实现并发的？

见 3 节

6.3 HSQLDB 中的隔离是如何保证的？

在 2PL 协议中，隔离是通过锁的机制来保证的。一些较高隔离级别，在事务结束后释放锁，而低级别的隔离在语句结束后就释放锁。具体可以第 4 节和第 5 节。

6.4 HSQLDB 的锁协议是怎样实现的？

HSQLDB 会在每条事务性语句执行前加锁，加锁的规则满足共享锁和排它锁的相容性关系。释放锁则根据隔离级别的不同在不同时机进行。

6.5 HSQLDB 中可能出现死锁吗？怎样预防或者解决？

正如 5.4 节分析中所说一样，如果当 session A 与 session B 产生循环依赖时，就会出现死锁。HSQLDB 会在加锁前判断是否会死锁，如果会发生死锁，则回滚当前事务。