

Lab 4 Report

姓名：戚晓睿



学号：1811412

GitHub: <https://github.com/NickSkyyy/SimpleDB>

GitCommitPic: see it below



Commits on May 25, 2020

Lab 4 without extra credit finished.
NickSkyyy committed 2 days ago

 f1cd01a 



Commits on May 23, 2020

Dead Lock Test finished.
NickSkyyy committed 4 days ago



 5d60df4 

Commits on May 20, 2020

Exercise 4 finished. ...
NickSkyyy committed 7 days ago



 faf17fe 

Transaction unit test passed.
NickSkyyy committed 7 days ago

 36c3232 



Commits on May 19, 2020

Exercise 2 use time checker. ...
NickSkyyy committed 8 days ago

 43ced5b 

Commits on May 18, 2020

Exercise 1 finished.
NickSkyyy committed 9 days ago

 5b2968b 

1 设计思路

本次Lab的核心内容是对事务（Transaction）的处理，包括脏页（Dirty Page）的更新、页锁（Lock）信息的存取等等，整体难度较高，需要对事务ACID的特点有所把握，并在处理代码时保持较为清晰的思路。主要改动均在 `BufferPool.java` 文件当中，以及新增加的类 `Lock.java` 用于存储页锁信息，并处理对锁的更删改查等一系列操作。

思路不清晰的时候会因为比较繁琐的内容卡住，花了3天进行debug，加上一段时间的理论课学习，完成此次Lab之后对事务和锁有了新的认识，下面将进行更加详细的讲解。

2 详细介绍

2.1 Lock.java

此次实验中，增加了新的类 `Lock.java` 进行页锁信息的保存和处理，内部包含如下三个数据结构：

```
1 private Map<PageId, List<TransactionId>> rLock; // read lock (PageId key)
2 private Map<PageId, TransactionId> wLock; // write lock (PageId key)
3 private Map<TransactionId, List<Pair<TransactionId, PageId>>> wait; //
  waitList
```

根据 Permissions 提供的访问权限不同，rLock 记录读锁 (Shared Lock) 信息，wLock 记录写锁 (Exclusive Lock) 信息，wait 记录多线程运行时**事务之间的交叉等待情况**。类内提供对于锁更删改查操作的接口，将在具体练习处做具体说明。

2.2 Lock.grantLock()

在之前的Lab中，BufferPool.getPage() 方法**并没有同时获取锁信息**；本次，我们在获取Page之前添加如下代码：

```
1  bool f = locks.grantLock(tid, pid, perm);
2  while (!f) {
3      ...
4      f = locks.grantLock(tid, pid, perm);
5  }
```

根据资料，模拟多线程处理用类似 while(true) 的循环，内置布尔变量 f 进行锁信息的获取，针对获取情况做不同的实际处理。取锁情况分为两种，根据参数 perm 分为**请求读锁和请求写锁**。

读锁有如下情形：

序号	情形	处理
1	其他事务拥有对页的写锁	等待
2	本事务没有对页的读锁（不论有无写锁）	上锁返回
3	本事务拥有对页的读锁	命中返回

对应代码片段：

```
1  if (perm == Permissions.READ_ONLY) {
2      // read only
3      // 写锁不空且不为tid
4      if (admin != null && !tid.equals(admin))
5      {
6          Pair<TransactionId, PageId> tp = new Pair<>(admin, pid);
7          if (!waitList.contains(tp)) waitList.add(tp);
8          wait.put(tid, waitList);
9          return false;
10     }
11     // 写锁为空或者拥有tid的写锁
12     if (!permList.contains(tid))
13         permList.add(tid);
14     rLock.put(pid, permList);
15 }
```

写锁情况更加复杂一些，有如下情形：

序号	情形	处理
1	本事务拥有对页的写锁	命中返回
2	其他事务拥有对页的写锁	等待
3	页写锁为空，拥有超过1个读锁	等待
4	页写锁为空，仅有1个本事务的读锁	升级返回
5	页写锁为空，仅有1个非本事务的读锁	等待
6	其他	上锁返回

对应代码片段：

```

1  else {
2      // write
3      // 拥有tid的写锁
4      if (tid.equals(admin)) return true;
5      // 写锁不为空且不是tid的写锁
6      if (admin != null) {
7          Pair<TransactionId, PageId> tp = new Pair<>(admin, pid);
8          if (!waitList.contains(tp)) waitList.add(tp);
9          wait.put(tid, waitList);
10         return false;
11     }
12     // 写锁为空
13     // 拥有超过1个读锁
14     if (permList.size() > 1) {
15         for (int i = 0; i < permList.size(); i++) {
16             TransactionId pTid = permList.get(i);
17             Pair<TransactionId, PageId> tp = new Pair<>(pTid, pid);
18             if (!waitList.contains(tp) && !tid.equals(pTid))
19                 waitList.add(tp);
20             wait.put(tid, waitList);
21             return false;
22         }
23         // 只有1个读锁，且不是tid的读锁
24         if (permList.size() == 1 && !permList.contains(tid)) {
25             TransactionId pTid = permList.get(0);
26             Pair<TransactionId, PageId> tp = new Pair<>(pTid, pid);
27             if (!waitList.contains(tp) && !tid.equals(pTid)) waitList.add(tp);
28             wait.put(tid, waitList);
29             return false;
30         }
31         wLock.put(pid, tid);
32     }

```

练习1至此基本完成，查锁（getLock）、删锁（unLock）只是对数据结构操作，比较简单因此不做详细介绍。

2.3 BufferPool.evictPage()

在之前的Lab中，`BufferPool.evictPage()` 方法采用的是最长记录驱逐政策（Time Order Policy），利用 `order` 数组记录写入`BufferPool`的时间顺序并进行页的驱逐；本次Lab要求，**对于脏页不能驱逐**，因此加入额外判定，

```
1  for (int i = 0; i < order.size(); i++) {
2      pid = order.get(i);
3      Page p = pool.get(pid);
4      if (p.isDirty() != null) continue;
5      f = true;
6      break;
7  }
```

这里有一种错误类型初见端倪，对于 `try catch` 匹配块，`catch`部分的捕捉需要格外注意，如果类内方法需要对异常进行抛出处理，则`catch`的时候不需要进行格外的操作，**且需要特别捕捉相应的异常，而不能用普适的 `Exception e`**。

2.4 BufferPool.transactionComplete()

事务完成时有两种情况，提交修改（Commit）和回滚（Rollback / Abort），该方法利用布尔参数 `commit` 判断处理情形。

Commit时，应当提交修改信息写入磁盘；Abort时，应当从磁盘中取出原有文件再放入cache中。二者的共性是，都需要将**事务拥有的锁信息进行剔除**。

```
1  if (commit) {
2      flushPages(tid);
3      ...
4  }
5  else {
6      Set<PageId> key = pool.keySet();
7      Iterator<PageId> it = key.iterator();
8      while (it.hasNext()) {
9          PageId pid = it.next();
10         Page p = pool.get(pid);
11         if (p.isDirty() != null && p.isDirty().equals(tid)) {
12             DbFile df =
Database.getCatalog().getDatabaseFile(pid.getTableId());
13             Page oldP = df.readPage(pid);
14             pool.put(pid, oldP);
15         }
16     }
17 }
```

而我在处理的时候，在Commit部分**额外加入了一个对BufferPool的处理**，代码如下：

```

1 // delete pool info
2 Set<PageId> key = pool.keySet();
3 Iterator<PageId> it = key.iterator();
4 while (it.hasNext()) {
5     PageId pid = it.next();
6     if (holdsLock(tid, pid)) {
7         releasePage(tid, pid);
8         if (locks.isEmpty(pid)) {
9             it.remove();
10            order.remove(pid);
11        }
12    }
13 }

```

这样处理的好处是，及时清空cache的信息以便后续操作，会让整体性能提高一些。至此，练习1-4均已完成。特别注明，BTree的相关测试也可以正常通过运行。

2.5 Lock.isDead()

检查死锁，这是整个Lab4最难的部分。首先要对死锁的定义有所明确：

当某一个事务所等待的对象，直接或间接的等待该事务时，形成死锁

考虑到SimpleDB的设计，我使用如下的数据结构：

```

1 private Map<TransactionId, List<Pair<TransactionId, PageId>>> wait; //
   waitList

```

在2.1部分也已经进行了说明，这里记录的信息是， tid_i 正在因为 pid 的访问而等待 tid_x 。例如，

事务	拥有	等待
tid1	pid1(R)	pid2(W)
tid2	pid2(R)	pid3(W)
tid3	pid3(R)	pid1(W)

注：拥有列表中**R**代表读锁权限，**W**代表写锁权限；等待列表中，二者代表申请的锁类型。

根据上表形成了死锁类型 $tid1 \rightarrow tid2 \rightarrow tid3 \rightarrow tid1$ ，这是我们的代码应该检测出的内容。利用有向图判环的算法，dfs配合三值标记数组即可完成。

```

1 for (int i = 0; i < waitList.size(); i++) {
2     Pair<TransactionId, PageId> tp = waitList.get(i);
3     TransactionId nextTid = tp.getKey();
4     //System.out.println("Now trace with Pair<" + tid.toString() + ", " +
   nextTid.toString() + ">");
5     if (!mark.containsKey(nextTid) || mark.get(nextTid) == 0) {
6         // 未加入mark序列（未被访问），在序列且标记为0
7         boolean f = isDead(nextTid, mark);
8         if (f) return true;
9     }
10    if (!tid.equals(nextTid) && mark.get(nextTid) == -1) return true;
11 }

```

为了提高效率，不让每次调用 `BufferPool.getPage()` 的时候都重判全图，我设计了**两个方法**：

```
1 public synchronized boolean isDead(TransactionId tid, Map<TransactionId, Integer> mark) {}
2 public synchronized boolean isDead(TransactionId tid, PageId pid, Map<TransactionId, Integer> mark) {}
```

前者是用于dfs遍历中间步骤，后者用于死锁检查的入口部分，**只检查与初始事务和页相关的其他tid**即可。这样能够提高系统效率，减少不必要的重复查环。

2.6 Transaction System Test

至此，DeadLock的测试通过完成，但是因为Transacion System Test总是通过不了，因此debug了2-3天。考虑到很多细节部分，设计了如下的各种方法：

```
1 public synchronized void unwait(TransactionId tid, PageId pid) {}
2 public boolean isEmpty(PageId pid) {}
```

当然，这些方法只是杯水车薪。通过**对测试源代码**的解读，逐渐发现问题所在：

```
1 } catch (TransactionAbortedException te) {
2     //System.out.println("thread " + tr.getId() + " killed");
3     //give someone else a chance: abort the transaction
4     tr.transactionComplete(true);
5     latch.stillParticipating();
6 }
```

上述代码是测试源代码中的 `try catch` 块，运行时，当打开行2的测试时，**发现并没有执行此处的 `catch`**，因此每次abort时，都不会执行这里提供的complete方法。

检查到这个地方的时候已经过了3天，慢慢debug发现了问题，**在之前的Lab中，会习惯性使用 `try catch` 块**，这样的习惯会导致在insert或者delete的过程中，由于死锁判断抛出的 `TransactionAbortException` 被**提前截断**，因此无法传递至System Test当中，后来将catch中的通用Exception改为Transaction专用异常处理，并取消 `e.printStackTrace()` 对异常的打印，即可完成整个System Test。

3 附加说明

由于没有做Tuple层面的锁，附加题1跳过不讲。

3.1 Timeout VS Dependency Graphs

在未做到练习4时，`BufferPool.getPage()` 方法中我采用的是超时 (Timeout) 策略，代码如下：

```
1 boolean f = locks.grantLock(tid, pid, perm);
2 int cnt = 0;
3 while (!f) {
4     if (++cnt == 3) throw new TransactionAbortException();
5     Thread.sleep(200);
6     f = locks.grantLock(tid, pid, perm);
7 }
```

对于很多出现死锁的测试，需要跑到我设定的超时时间才能完成测试，消耗时间大概在设定时间的cnt倍。

使用死锁检测和异常抛出后，相对的使用时间**明显变少**。对应代码如下，`Lock.isDead()` 方法如2.5所介绍：

```
1  boolean f = locks.grantLock(tid, pid, perm);
2  while (!f) {
3      if (locks.isDead(tid, pid, null)) {
4          isTAE = true;
5          break;
6      }
7      Thread.sleep(200);
8      f = locks.grantLock(tid, pid, perm);
9  }
```

3.2 Abort Yourself VS Others

死锁出现时，abort策略分两种，终止当前事务请求（Yourself）和终止与当前事务相关的其他事务（Others）。我在本次Lab中采取的策略是**前者**，因为这样**能够最大限度保留用户操作**，减少重启事务的过程。

本次abort的代码如下：

```
1  // BufferPool.transactionComplete()
2      locks.deleteLocks(tid);
3
4  // Lock.deleteLocks()
5  ...
6      // 经过abort, tid不需要再等待任何事务的完成
7      if (tid.equals(pTid)) {
8          it2.remove();
9          continue;
10     }
11     // 查找其余的tid是否在等待该事务的完成
12     List<Pair<TransactionId, PageId>> waitList = wait.get(pTid);
13     if (waitList == null || waitList.isEmpty()) {
14         it2.remove();
15         continue;
16     }
17     ...
```

在 `Lock` 类里面设计了对应的方法方便abort策略的实现，通过传参 `tid` 区分要删除的事务锁信息。

如果要调整成为Abort Others策略，则只需要将上述代码（伪代码，BufferPool当中的）改为：

```
1  for (每一个当前事务T需要等待的事务tid)
2      locks.deleteLocks(tid);
```

则可以完成对其他相关事务的除锁操作，这样的话，相对的事务需要重启，**重启的代价更高**，得不偿失。