项目技术报告

目录

1 集中式两阶段提交的实现

1.1 协调器 Transaction Manager 的实现

下文协调器等价于事务管理器。

1.1.1 正常流程

协调器维护维护一个所有事物 ID 的结合和每一个事务 ID 当前的状态。如下：

```
// all active transactions
private HashMap<Integer, String> xids = new HashMap<>();
```

协调器维护 4 个重要接口：enlist, start, commit, abort。其中 enlist 由事务参与者调用，其它三个由流程控制器调用。

每个事务有四种状态：INITED, PREPARING, COMMITED, ABORTED & DONE. 四个状态的涵义以及其转变的实现如下。

INITED: 当流程控制器调用 start 接口开启一个事务时，当前事务开始，分配一个事务 ID，并进入 INITED 状态。

PREAPARING: 当流程控制器调用 commit 接口准备提交一个分布式事务时，协调器中控制该事务进入 PREPARING 阶段。然后协调器通知事务参与者准备该事务。

COMMITED: 当协调器收到所有参与者准备成功的消息时，写 LOG 记录该事务为 COMMITED 状态。LOG 记录成功后，该事务进入 COMMITED 状态。

ABORTED & DONE: 在我们的实现中，ABORTED & DONE 在协调器端的状态维护是一样的，即从事务 ID 结合中移除该事务。其具体实现有所不同。当协调器在 PREAPRING 阶段未成功收到所有参与者的成功恢复消息时，通知能通知到的参与者放弃该事务，然后移除该事务，该事务进入 ABORTED 状态； 当协调器收到各个参与者成功提交的消息时，移除该事务 ID，该事务进入 DONE 状态。其实我们不需要区分这两种状态，进入这两种状态之一后，如果有参与者来询问，即是 ABORTED 状态， 通知该参与者放弃该事务。此外，就是 DONE，

无需处理。

1.1.2 宕机处理以及事务恢复

我们维护了事务独一无二的 ID，保证新事物与老事务的 ID 不会重复。

协调器在处理事务的过程中，每当事务状态改变时，将新的事物状态写入 LOG（即将 xids 写入磁盘）。例如当某一事物进入 COMMITED 状态后，如下记录 LOG

```
// log commit with xid
synchronized (xids) {
    xids.put(xid, COMMITTED + "_" + xidRMs.size());
    utils.storeObject(xids, "data/" + xidsStatusPath);
}
```

如果协调器宕机，我们需要在其重启时根据 LOG 的状态恢复事务。 此时我们需要维护 COMMITED 状态的事务的 ID 集合。其它的事务就直接放弃就可以了。讨论一下事务状态为 ABORTED 或者 DONE 时的情况。我们的实现中协调不保存该状态，LOG 记录若有事务 ID，其必为 COMMITTED，INITED 或者 PREPARING。ABORTED 处理与 PREPARING 相同，也无需记录 LOG，直接移除 ID 即可。如果在移除的过程中宕机，则 LOG 里记录的是 PREPARING，恢复时也是一样的。DONE 就是事务已完成，无需处理。需要恢复的事务集合定义如下：

```
// transaction to be recovered after some RMs died or TM died
private HashMap<Integer, Integer> xids_to_be_recovered = new HashMap<>();
```

在恢复的时候，我们将需要恢复的事务加入上面的待恢复事务列表。

然后注册服务，使得参与者能够及时重新连接到协调器。然后我们如下恢复事务。

参与者会定期检查与协调器之间的连接，当发现连接失效时，就会重新连接协调器直到成功。当获得连接后，会调用协调器的 enlist 接口重新加入事务。当协调器收到了 enlist 请求时，我们丰富了 enlist 的接口，该接口会返回当前加入事务的状态。如果 enlist 的事务在待恢复的 xids_to_be_recovered 列表中，就通知该事务已经 COMMITED，让该参与者提交任务。如果不在，我们检查是否在正常事务列表 xids 中，如果不在则返回该事务已经 ABORTED。

我们使用参与者的数目来表示一个事务是否被完全恢复。如果恢复的数目达到了预订的数据，则表示该事务已经完成。这种方法的一个问题是参与者收到提交信息后可能会宕机，当其再次启动来访问时，该事务已经完成并且移除， 该事务就会被不一致的放弃。第二个问题时，有的参与者可能已经提交了，只是信息没接收到，协调器就不会减去对应的数量，则检测不到事务完成，但是实际上事务已经完成。所以我们这里不移除可能已经完全恢复的事务。虽然存了无用的信息，但是能确保更高的准确度。

1.1.3 超时处理

在等待参与者 PREARED 或者 DONE 消息的时候可能超时，在我们的项目中，我们认为发生超时都是参与者宕机了，处理方式如同宕机处理。

1.2 参与者资源管理器的实现

下文中参与者与资源管理器为同一东西。

1.2.1 正常流程

我们沿用提供的代码，做了一些略微的修改。参与者会使用到三种状态。 INITED & PREPARED, COMMITED。参与者自己维护的状态只有 INITED & PREPARED。如下：

```
private HashSet xids = new HashSet();
```

如果一个事务 ID 在列表中，则表明该事务是 INITIED 或者 PREPARED 状态；当事务 COMMITED 之后则移除该事务 ID。

当流程控制器调用资源管理器时，参与者就开始了事务。在这里做了修改以满足需求。我们在读取数据时，应该先询问锁，然后再从磁盘读取数据至影子页面。但是原有的方法如下所示：

```
// TODO should aquire the lock first
RMTable table = getTable(xid, tablename);
ResourceItem item = table.get(key);
if (item != null && !item.isDeleted()) {
    // TODO value has already been read to memory
    table.lock(key, LockManager.READ);
    if (!storeTable(table, new File("data/" + xid + "/" + tablename))) {
        throw new RemoteException("System Error: Can't write table to disk!");
    }
    return item;
}
```

数据 item 已经从磁盘读取，然后再去询问锁。在询问锁的时候虽然会等待其它事务提交。但是之后该事务依然使用了老的数据，而不是其它事务提交后的状态。为了解决这个问题，有一个简单的方法，如下

：

```
// get lock before read from disk
// will lock the un-existed item that maybe inserted later
// can not work if we don't know the key.
if (!lm.lock(xid, tablename + ":" + key.toString(), LockManager.READ))
    throw new RuntimeException();
RMTable table = getTable(xid, tablename);
ResourceItem item = table.get(key);
if (item != null && !item.isDeleted()) {
```

```
        // put lock record
        table.putLock(key, LockManager.READ);
        if (!storeTable(table, new File("data/" + xid + "/" + tablename))) {
            throw new RemoteException("System Error: Can't write table to disk!");
        }
        return item;
}
```

但是该方法也存在一些问题比如要知道 key，以及会锁住根本不存在的数据，导致将来其他事务不能插入。为了解决这个问题，一个解决方法如下

```
// read twice, first to get lock, then to read.
// if the item hasn't been locked by other transactions, just read twice and the results are same
// if the item has been locked by other transactions, then wait for lock and read new result.
// first to get lock
RMTable table = getTable(xid, tablename);
ResourceItem item = table.get(key);
if (item != null && !item.isDeleted()) {
        table.lock(key, LockManager.READ);

        // then to read values
        // remove old value
        Hashtable xidtables = (Hashtable) tables.get(xid); // can not be null
        synchronized (xidtables) {
            xidtables.remove(tablename);
        }
        // read new value
        table = getTable(xid, tablename);
        item = table.get(key);
```

```
        if (!storeTable(table, new File("data/" + xid + "/" + tablename))) {
            throw new RemoteException("System Error: Can't write table to disk!");
        }
        return item;
    }
```

基于现有的实现，我们略微改动，第一次读是为了检测锁，获取锁之后，从内存移除所读内容，再重新从磁盘读取内容。这一部分的验证将在 T_L_RW 测试用例中解释。

对于一次查询多个条目，同样读取两次，第一次读是为了检测锁，获取锁之后，从内存移除所读内容，再重新从磁盘读取内容。 如下：

```
Collection<ResourceItem> result = new ArrayList<>();
// read twice, first to get lock, then to read.
// if the item hasn't been locked by other transactions, just read twice and the results are same
// if the item has been locked by other transactions, then wait for lock and read new result.
// first to get lock
RMTable table = getTable(xid, tablename);
synchronized (table) {
    for (Iterator iter = table.keySet().iterator(); iter.hasNext(); ) {
        Object key = iter.next();
        ResourceItem item = table.get(key);
        if (item != null && !item.isDeleted() && item.getIndex(indexName).equals(indexVal))
{
            table.lock(key, LockManager.READ);
        }
    }
}
```

```
// then to read values
// remove old value
Hashtable xidtables = (Hashtable) tables.get(xid); // can not be null
synchronized (xidtables) {
    xidtables.remove(tablename);
}
// read new value
table = getTable(xid, tablename);
synchronized (table) {
    for (Iterator iter = table.keySet().iterator(); iter.hasNext(); ) {
        Object key = iter.next();
        ResourceItem item = table.get(key);
        if (item != null && !item.isDeleted() && item.getIndex(indexName).equals(indexVal))
{
            // table.lock(key, LockManager.READ); // have been locked
            result.add(item);
        }
    }
    if (!result.isEmpty()) {
        if (!storeTable(table, new File("data/" + xid + "/" + tablename))) {
            throw new RemoteException("System Error: Can't write table to disk!");
        }
    }
}
return result;
```

其它地方基本如同原有提供的实现，没有变化。

1.2.2 宕机处理以及事务恢复

在原有实现的基础上，我们利用协调器 enlist 方法返回的状态来恢复事务。

当资源管理器重启时，会重新连接协调器。然后在调用 enlist 的时候询问当前事务的状态，如果当前事务已经 COMMITED，则资源管理器会提交该任务，释放锁。如果当前事务已经 ABORTED，则资源管理器放弃该任务，释放锁。否则状态则视为为 INITED，等待协调器即事务管理器的管理。

在两阶段提交的过程中，如果资源管理器是在 PREARE 的时候宕机，则返回的状态是 ABORTED，可以直接放弃任务。如果资源管理器是在 COMMIT 的时候宕机，则返回的状态是 COMMITED，可以直接提交任务。在现在的实现中，如果资源管理器在两阶段提交开始前（即客户端调用 commit 之前）就宕机了，则返回的状态是 INITED。该事务现在还能继续运行，并且持有锁。直到客户端调用 commit 之后，事务管理器开始两阶段提交，会检测到该事务管理器之前的已不能连接的尸体，然后会通知当前资源管理器以及其它资源管理器放弃事务。最终达到结果一致。

1.2.3 超时处理

在我们的项目中，我们认为发生超时都是参与者宕机了，处理方式如同宕机处理。

1.3 测试功能实现

协调器和参与的对测试接口的实现均和已有的资源管理器实现一样。使用宕机标志 dieTime 来记录该在什么时候宕机，然后再事务进行到对应进度的时候宕机。

我们保留原来的 Client.java 为 SimpleClient.java。我们可以在该客户端写相应的测试代码来测试各个服务。为了更方便的测试，我们参考了 UCI 提供的测试方法[1]。具体的做法可以参

[1] https://www.ics.uci.edu/~cs223/projects/projects2.html

考项目的 README。

1.4 部署说明

在原有的代码基础上，Java rmi registry 的注册地址是 ip:port/RMIName。所以我们并不需要为每一个 RM，TM，WC 分配一个端口。只要使用 RMIName 来区分各个服务即可，所以相应的接口配置文件。同时因为所有的 RMs 使用相同的实现，再为每一个 RM 写一个启动类，略显多余，所以我们删除了多余的启动类。

最终所有的启动方法，测试方法都封装在对应的脚本文件中，详见项目 README 介绍。

2 测试用例及说明

2.1 基本设计

我们的测试方式详见项目 README。考虑项目内容，可将测试分为以下几部分。

2.1.2 事务基本功能

2-4 个测试用例,

- 启动事务
- 提交事务
- 放弃事务
- 输入异常

2.1.2 基本业务逻辑

增删改查极其组合业务，包含数据 Flight, Room, Car, Customer。可以添加修改删除 Flight, Room, Car, Customer。用户可以预订或者取消预订 Flight, Room, Car。8-16 个测试用例。

- 增加 Flight, Room, Car, Customer。
- 删除 Flight, Room, Car, Customer。

- 查看 Flight, Room, Car, Customer。
- 修改 Flight, Room, Car。
- 用户预订 Flight, Room, Car。
- 用户取消预订 Flight, Room, Car。
- 输入异常
    - 异常 key，数据

2.1.3 并发测试

当并发执行以上事务逻辑，能够正确执行。基于两阶段锁测试。随机选择基本事务。5-10 个测试用例

- 读读共享: T_L_RR
- 读写等待: T_L_RW
- 写读等待: T_L_WR
- 写写等待: T_L_WW
- 死锁，后者放弃事务
    - 读读写写死锁: T_L_RRWW
    - 2 数据，读写读写死锁: T_L_RWRW
    - 2 数据，写读写读死锁: T_L_WRWR
    - 2 数据，写写写写死锁: T_L_WWW

2.1.4 宕机测试

基于两阶段提交进行测试。随机选择基本事务。基于现有 WC 接口。 7-14 个测试用例。

- TM 宕机
    - 两阶段提交前宕机，事务失败. T_TM_DIE
    - 开启事务后直到写 COMMIT log 前宕机（After INITED），事务失败
        - PREPARE 之前，PREPARE 之后. T_TM_DBC
    - 写 COMMIT log 之后宕机，事务成功. T_TM_DAC
    - 事务 ABORT 的时候宕机，同写 COMMIT log 前，事务失败. no condition to test.
- RM 宕机，多个 RM，随机选择宕机

- 与当前事务无关。事务成功 D_RM_DIE
- 与当前事务有关
  - PREPARE 前宕机（已被 TM 通知过 PREPARE），事务失败 D_RM_DBP
  - PREPARED 后宕机，（未成功发送 prepared 消息至 TM，事务失败 D_RM_DAP
  - COMMIT log 前宕机过程中宕机（已成功发送 prepared 消息至 TM），事务成功 D_RM_DBC
  - COMMIT log 后宕机，事务成功. no need to test.

2.2 重要测试用例说明

我们使用了一些现有的测试用例[2]。该测试用例已经包含许多内容。我们在此基础上考虑其忽略的部分或者其与我们实现有冲突的部分，或者一些对现有功能的探索。

2.2.1 用例：T_L_WR，并发测试

该测试用例主要用于测试并发执行的过程中的读写等待情况：对于一个数据 d，事务 A 首先获取 WRITE 锁，然后事务 B 在申请 d 的 READ 锁的时候就要等待事务 A 提交或者放弃，然后释放锁。事务 B 拿到 d 的 READ 锁之后，从数据库读出现有的 d 值。其核心过程如下

1 call addCars xid "SFO" 300 40

1 return true

2 call queryCarsPrice xid "SFO"

1 call commit xid

1 return true

2 return 40

首先线程 1 的事务获取了 car 的写锁，将其价格修改为 40. 然后线程 2 的事务要读取 car 的价格，这时候就需要等待线程 1 提交。等到线程 1 提交后，线程 2 获取锁，读到修

[2] https://www.ics.uci.edu/~cs223/projects/projects2.html

改后的价格 40.

结合 1.2.1 资源管理的实现，我们一定到在从数据库读取数据前先申请锁。在 UCI 的测试用例中没有包含对读取内容的检验，这也恰恰是一个忽略点。

2.2.2 用例：T_L_RRWW，并发测试

该测试用例主要用于测试并发执行的过程中的死锁处理情况：对于一个数据 d，事务 A 首先获取了 READ 锁，然后事务 B 又获取了 d 的 READ 锁，接着事务 A 又想获取 d 的 WRITE 锁，这时候事务 A 必须等待事务 B 释放 d 的 READ 锁，接着事务 B 并没有提交而是想要获取 d 的 WRITE 锁，这时候就发生了死锁。事务 A 和 B 中，必须有一个被放弃而使得另一个事务获取锁并正确执行。其核心过程如下

1 call queryCarsPrice xid "SFO"

1 return 30

2 call queryCarsPrice xid "SFO"

2 return 30

1 call addCars xid "SFO" 300 40

2 call addCars xid "SFO" 300 50

1 except transaction.TransactionAbortedException

2 return true

或者

1 call queryCarsPrice xid "SFO"

1 return 30

2 call queryCarsPrice xid "SFO"

2 return 30

1 call addCars xid "SFO" 300 40

2 call addCars xid "SFO" 300 50

2 except transaction.TransactionAbortedException

1 return true

但是事实上线程 1 和 2 的事务都检测到了死锁。究其原因，现在的锁管理器检测死锁的方法是根据时间的，当超过一定时间后仍然不能获取锁，则当前事务就会抛出死锁。根据该实现分析以上测试用例，在测试用例中，1 线程执行第 5 行时，必须等待获取 WRITE 锁，然后 2 线程紧接着执行第 6 行时，必须等待获取 WRITE 锁。 之后第 7 行无论时执行线程 1 还是线程 2 都会因为等不到锁而抛出死锁。但是紧接着第 8 行不会获得锁，因为在另一个线程释放锁之前，本线程也已经到时死锁了。在两个用例中，第 6 行之后，2 个线程几乎同时要检测出死锁，但是一般线程 1 早等了一点点，会先抛出死锁，但是由于释放锁耗时较长（相比于两个事务先后申请锁的时长），线程 1 还没释放锁，线程 2 也抛出了死锁。

基于现有的锁管理器，如果想要测试死锁，需要如下测试：

1 call queryCarsPrice xid "SFO"

1 return 30

2 call queryCarsPrice xid "SFO"

2 return 30

1 call addCars xid "SFO" 300 40

1 except transaction.TransactionAbortedException

2 call addCars xid "SFO" 300 50

2 return true

第 5 行，线程 1 等待线程 2 释放锁，但是第 6 行需要线程 1 执行，线程 2 不能执行而被阻塞。所以线程 1 等了一段时间后会抛出死锁，放弃事务。接着线程 2 就可以征程执行。

基于现有的锁管理器实现的死锁机制并不能通过我们通常意义上的死锁测试。我预想的死锁是根据锁的情况推到出来的。现有的执行逻辑为 T_L_RRWW_PASS 和 T_L_RRWW_WAIT 这样的情况，其实 T_L_RRWW_PASS 因该有一个事务能不被放弃，T_L_RRWW_WAIT 实际上

不是死锁，只是等了很久。这个在现实的业务中，算是可以商榷的，等待超时记为死锁可以解决一些情况。

2.2.3 用例：Liti3，业务逻辑测试。

该用例原本是一个锁测试，但是在我们的归类里可以归为业务逻辑测试。原测试核心过程如下：

1 call reserveItinerary xid "John" (347,3471) "Stanford" false true

1 return true

2 call queryCustomerBill xid "John"

1 call commit xid

1 return true

2 return

用例设计者认为第 1 行应当有用户或者账单相关的 WRITE 锁，导致第 3 行线程 2 申请 READ 锁时等待。

但是在我们的实现中当用户预订的时候，会在 Reservation 的表中增加新的 reservation 记录，并不会影响 customer 的使用。 当 customer 查询的账单的时候，由于 1 线程事务并没有提交，数据库 reservation 中并没有该用户的记录，所以不会发生锁等待情况，应当直接可以返回查不到账单，也就是账单数额为 0 .

为了通过该测试，我们认为用户预订的时候是在改变用户的信息，所以添加申请用户 WRITE 锁的代码。如下：

```
// lock customer,
// This is for test case Liti3. The case think the reservation action should modify
//     the info of customers and get the WRITE lock.
// But, in our implementations, all new reservations are stored in Reservations table.
```

```
// We just need the READ lock for customers.
try {
    rmCustomers.update(xid, rmCustomers.getID(), custName, cust); // just to set WRITE lock for the test
} catch (DeadlockException e) {
    // dead lock happened, quit this transaction
    abort(xid);
    throw new TransactionAbortedException(xid, "This transaction cause dead lock: " + e.getMessage());
}
```

每次预订的时候，我们实际上不会修改用户的信息，我们添加额外的 WRITE 锁请求。因为项目要求中说“RESERVATIONS 表可以和 CUSTOMERS 表合并,也可以分开”。我们选择的是分开处理，但如果是合并处理，就会锁定用户的条目，从而测试用例是合理的。