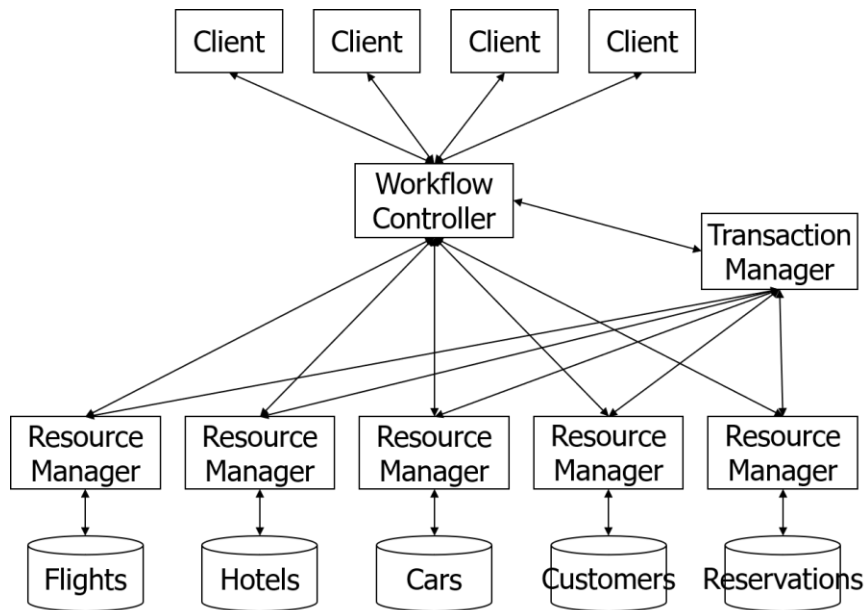


《分布式旅游预定系统》技术报告

谢思豪 19212010037

一. 概述

本系统主要分为三个模块：资源管理器（Resource Manger, RM），事务管理器（Transaction Manager, TM）和流程控制器（Workflow Controller, WC）。其中，资源管理器负责实际数据的增删改查与持久化；事务管理器负责管理事务，保证事务的 ACID 特性，即使系统出现故障，事务的 ACID 特性仍会得到保证；流程控制器用于整合资源管理器与事务管理器的功能，对外提供业务接口，是用户与系统交互的唯一入口。系统各模块间交互逻辑如下图所示：



27

二. 资源管理器 (RM)

RM 负责实际数据的增删改查与持久化，并对外提供操作数据的接口（如 query, insert, delete 等）以及事务相关的接口（如 prepare, commit 等）。在 RM 中，数据以 ResourceItem 的形式存储在 HashTable 之中。课程提供的 Source Code 中已经实现了 RM 的绝大部分功能，本项目主要进行了四点改动：

1. 创建与实际数据表对应的实体类

本质上，每个数据表中存储的数据都为 ResourceItem，但不同的实体拥有不同的属性，所以需要为每个实体创建对应的继承自 ResourceItem 的实体类。根据项目说明，本系统共包含五个实体类：Flight, Hotel, Car, Customer, Reservation。每个实体类均继承自 ResourceItem 并拥有自己独特的属性与主键。考虑到 Customer 和 Reservation 的主键不同，为了便于后续操作，本系统将他们独立开来。

2. 将 RManager***.java 代码归入 ResourceManagerImpl.java 之中

不同的 RM 管理着不同的数据表。虽然不同的数据表中存放的实体不同，但是通过合理的抽象与封装，不同的 RM 操作其管理的数据表的逻辑完全一致。因而，在本系统中，不同的 RM 的唯一区别在于其名字不同。所以，我们认为没必要为每个 RM 新建一个单独的类，而是将实例化不同 RM 的逻辑统一归入

ResourceManagerImpl.java 之中，并通过传入不同的 rmiName 属性来区分不同的 RM 实例。通过这种方式，减少了冗余代码，降低了系统的复杂度，更利于管理。

3. 保证各 RM 之间的隔离性

不同的 RM 复用了同一份代码（即 ResourceManagerImpl.java），因而，我们应保证 RM 之间的隔离性。但是我们在测试的过程中发现，ResourceManagerImpl.java 中对于事务日志的保存逻辑并没有保证各 RM 之间的隔离性，都统一写入了文件 transactions.log 之中。这导致的最直接问题是各 RM 之间的事务日志相互干扰。为了解决这一问题，我们在每个 RM 的事务日志文件名中加入了 RM 的名字以区别不同的 RM 事务日志。例如，对于 RMFlights 来说，其事务日志存储于文件 RMFlights_transactions.log 之中。

4. 保证事务状态发生改变后立即写入事务日志中

RM 事务日志用于保存 RM 当前的事务状态并用于系统故障后的恢复。因此，每当事务的状态发生改变时（如新建事务，事务提交等），应立即把该变化写入事务日志之中。但我们测试时发现，ResourceManagerImpl.java 中 commit 方法和 abort 方法的实现并未做到这一点。更准确的说，当 RM commit 或 abort 成功后，并未将这一信息立即写入日志之中。为了解决这一问题，我们在 ResourceManagerImpl.java 的 commit 方法和 abort 方法中分别加入了写入 RM 事务日志的代码逻辑。具体改动如下所示：

```
@@ -609,6 +626,7 @@ public void commit(int xid) throws InvalidTransactionException, RemoteException
609 626
610 627     synchronized (xids) {
611 628         xids.remove(new Integer(xid));
629 +        storeTransactionLogs(xids);
612 630     }
613 631 }
614 632

@@ -635,40 +653,7 @@ public void abort(int xid) throws InvalidTransactionException, RemoteException {
635 653
636 654     synchronized (xids) {
637 655         xids.remove(new Integer(xid));
656 +        storeTransactionLogs(xids);
638 657     }
639 658 }
```

三 . 事务管理器 (TM)

TM 负责保证事务的 ACID 特性。所有事务开始之前都需要先从 TM 中拿到事务 ID (xid) 后才能进行后续的操作。同时，TM 中还利用 Map 维护了每个事务所涉及的所有 RM，原子性的保证就是通过保证一个事务涉及的所有 RM 最终都执行了相同的事务操作 (commit 或 abort) 来实现的。

TM 的一个很重要的功能是保证当系统出现故障后事务的 ACID 仍能被保证。为了做到这一点，TM 在一个事务的状态发生变化时，都会立即将这一变化写入磁盘中。例如，当有 RM 通过 enlist 接口通知 TM 它所参与的事务 id 时，TM 会记录这一信息并写入磁盘之中。再比如，当用户希望提交事务并且事务可以提交时，TM 会将相应的事务状态改为 committed 并写入磁盘后再返回提交成功的信息。TM 通过这种先记入日志再返回结果的方式，保证了所有 TM 承诺的操作最终都会被完成（即使系统出现故障）。

本系统共考虑了八种异常情况

1. RM 在 enlist 后 fail (AfterEnlist)

由于 RM 在 enlist 后就 fail 了，所以用户无法完成后续的对数据的操作，因而 abort 整个事务。

2. RM 在 prepare 前 fail (BeforePrepare)

由于 RM 的 prepare 未成功，所以 TM 应返回 commit 失败的信息，并 abort 整个事务。

3. RM 在 prepare 后 fail (AfterPrepare)

虽然在这种情形下，RM 已经 prepared 了，但由于代码中的实现方式使得 RM 无法通知 TM 其已经“prepare OK”了，所以对于 TM 来说，RM 的 prepare 操作是失败的，所以我们选择给用户返回 commit 失败的信息并 abort 整个事务。

4. RM 在 commit 前 fail (BeforeCommit)

RM 的 commit 函数被调用，说明整个事务已经进入了 commit 状态，所以这个事务涉及的所有 RM 都必须成功的执行 commit 操作。若 RM 在 commit 前 fail，那 TM 则不会将此 RM 从其对应的事务的 RM 列表中移除，而是在 RM 重启后再次调用它的 commit 函数，直到成功为止。

5. RM 在 abort 前 fail (BeforeAbort)

与第 4 点类似，RM 的 abort 函数被调用说明整个事务已经进入了 abort 状态，则该事务涉及的所有 RM 都应该成功执行 abort 操作。若 RM 在 abort 前 fail 了，则 TM 会在 RM 重启后再次调用 abort 函数，直到成功为止。

6. TM 在 commit 前 fail (BeforeCommit)

根据定义，TM 在收到所有 RM 的“prepare OK”消息后，并在将日志中的事务状态改为 committed 之前失败。在这种情况下，客户端感知到 TM 失败并告诉用户事务未成功提交。而对于 TM 来说，由于其未能将日志中事务的状态改成 committed，所以在其重启后会选择 abort 这个事务。这个操作是和用户得到的反馈一致的。

7. TM 在 commit 后 fail (AfterCommit)

与第 6 点类似，但此种情况下 TM 已将磁盘中事务的状态改为了 committed，所以，事务最终将会被 commit。在 TM 重启后，它会调用此事务涉及的所有 RM 的 commit 接口，并保证所有的 RM 最终都成功 commit。

8. 整个系统 fail

若整个系统 fail，当 TM 重启时，它能从日志中获取系统 fail 前的所有事务状态，对于已经进入 committed/aborted 状态的事务来说，TM 会等到涉及的 RM 重启成功后调用 RM 对应的 commit/abort 接口。而对于系统 fail 前还处于执行中的事务，TM 会将它们的状态改为 aborted，并调用所有涉及到的 RM 的 abort 接口将事务 abort。通过这种方式，可以保证所有事务的 ACID。

四 . 流程控制器 (WC)

WC 的主要作用在于对外提供业务接口，是用户和系统交互的唯一入口，也是用户所能感受到的唯一组件。WC 的实现并不复杂，所有业务接口的实现逻辑都类似：即通过调用 RM/TM 提供的基础接口以完成对应的业务逻辑。每个业务接口的行为在 WorkflowController.java 文件中都有详细定义，根据定义编写代码即可，这里就不再赘述。

而对于系统测试类接口，我们主要做了两个重构：一是将 dieRM*** (String who) 接口统一成了 dieRM (String who, String dieTime) 接口；二是将 dieTM*** () 接口统一成了

dieTM(String dieTlme)接口。考虑合并这些接口的原因是因为这些接口的唯一不同在于 dieTime, 而 dieTime 完全可以使用参数的形式传入, 没必要在接口名中硬编码 dieTime。通过这种合并, 可以减少很多不必要的重复代码, 使得代码更加清晰, 更容易维护。

五. 测试用例的编写

为了使得测试用例逻辑清晰且易于维护, 本项目在 src 目录下新建了 test 文件夹用来专门存放测试用例, 并提供了编译运行测试用例所需的 Makefile。本项目的测试用例基于 Junit 框架编写, 共分为四个部分, 分别为:

1. NormalTest.java

在该文件中, 我们假设系统没有发生故障, 针对 WC 提供的所有业务接口进行了功能测试, 保证所有的接口行为都是正确的。

2. RMFailureTest.java

在该文件中, 针对 RM 的五种不同 fail 时间进行了测试, 以保证在所有情况下事务的 ACID 都能得到保证。

3. TMFailureTest.java

在该文件中, 针对 TM 的两种不同 fail 时间进行了测试, 以保证在所有情况下事务的 ACID 都能得到保证。

4. SystemFailureTest.java

在该文件中, 我们针对整个系统都发生故障的情形进行了测试, 以保证在系统故障的情况下事务的 ACID 依然能得到保证。

关于测试用例的详细说明请参见《分布式旅游预订系统》测试用例及说明。

六. 总结

就我个人而言, 我觉得这是一个很好的课程实践项目。做完这个课程实践项目, 我最大的收获是对分布式事务有了更深刻的认识。通过实现一个具体的系统, 使我对事务的 ACID 特性的理解不止停留于理论层面, 而是深入到了原理与实现层面, 并掌握了一些基本的分布式事务管理技术, 如两阶段锁, 两阶段提交等。

我觉得本项目的最大难点在于对整个系统架构的理解, 即每一模块分别负责哪些职责, 以及这些模块如何协作以构成一个功能完整的系统。弄懂整体架构以后, 由于实现部分最难的锁和 RM 已经提供了绝大部分代码, 所以在系统的实现上并不会特别大的困难。

最后, 我想对本课程实践项目提出两点个人建议, 一是目前源码中存在大量的多余代码 (如定义在接口中的字段默认是 public static final 的, 不需要显式写在代码中), 可以考虑对提供的 Source Code 进行重构, 以消除这些多余代码。二是目前给的 Source Code 中提供了锁和资源管理器的绝大部分代码, 而这一部分正是实现的难点之一, 可以考虑将这一部分 (在提供必要的指导下) 也留给学生实现。