

# Part B实验文档

---

19302010074 张诗涵

## 实现思路

---

### update方式

- 考虑到ppt中也有提到, 不要对存储文件的唯一副本进行修改, 我认为在修改时不应该直接修改 database文件, 而是应该选择开一个新的副本(或首先在内存中修改), 例如在"#db1"中修改, 如果能commit就用副本替代原件(如果是内存中写的话, 则是将其写入db 文件覆盖内容); 如果abort处理也更加简单, 放弃这份副本即可 (内存中更新的话更简单、直接不处理就好了)
- 这样保证了在未commit前不会影响到原件, 即使是后来的修改者read到的也是最新版被commit 的数据, 而不会出现读到了之前修改者实际并未commit的数据

### Write-Ahead-Log

- 理解: wal主要是为了给那些做到一半尚未commit也未abort的事务擦屁股善后(即没有outcome类 log的事务), 因为对于那些有结果的事务, 它们都已经完成 了自己的责任, 如果是需要abort就已经完成了回滚的义务, 只有那些中途断电的自己无法回滚, 只好由recover来帮助完成回滚、并为其添加outcome+abort的log
- 结构: LogFile类用ArrayList管理迄今为止的所有Log(即Log类对象), Log类记录了 type(BEGIN,CHANGE,OUTCOME三类), operation (NEW\_TRANSACTION/ABORT/COMMIT/PUT等), workId(即watermark), choices(选项字段, 主要是put操作需要记录是将第几行从什么值修改为了什么值)
- 记录: 在MyAtomicity的每一个操作(开始更新、更新某行、结束)之前都要进行Log的记录, 记录本次操作type、watermark、operation, 如果是put则还需要 选项字段(详细内容及结构见上), 从保存的文件恢复LogFile对象, 使用提供的addLog接口插入新 Log, LogFile类会在内部调用save()函数将本次结果保存进磁盘。
- Recover: 从保存的文件恢复LogFile对象, 从最后向前遍历, 如果该log type为OUTCOME则将其 加入finishIds, 并对每条判断如果其不在finishId中, 且type为CHANGE 就要进行回滚, 同时如果它还不在于loserId(所有中断事务的集合)中的话就将其加入。最后对loserId进行遍历, 为其中每条都添加OUTCOME(ABORT)记录表示 已经完成回滚善后

### Read-Capture

- 理解: read-capture是一种乐观锁的理念, 即认为大概率不会出现干扰。
- 运行的主线程中管理着watermark的静态全局变量, 每个atom试图进行修改操作时都必须获取一个 watermark
- 获取watermark的方式是将当前watermark + 1并返回
- 任何update操作最后提交前需要比较自身watermark是否与当前最高watermark一致, 若一致则 commit, 不一致则abort
- commit时就正式打开db文件覆盖其原内容
- abort时就放弃本次所有内存内修改部分, 同时休眠一段时间(我设置为了9000ms)然后再重新调用自身再尝试写入。
- 强制休眠一段时间是因为之前abort一定是和一个新修改者冲突了, 如果此时立刻retry, 则大概率自己又会干扰到对方、对方再次retry干扰自己..... 造成互相钳制, 因此一次abort后必须进行休眠。

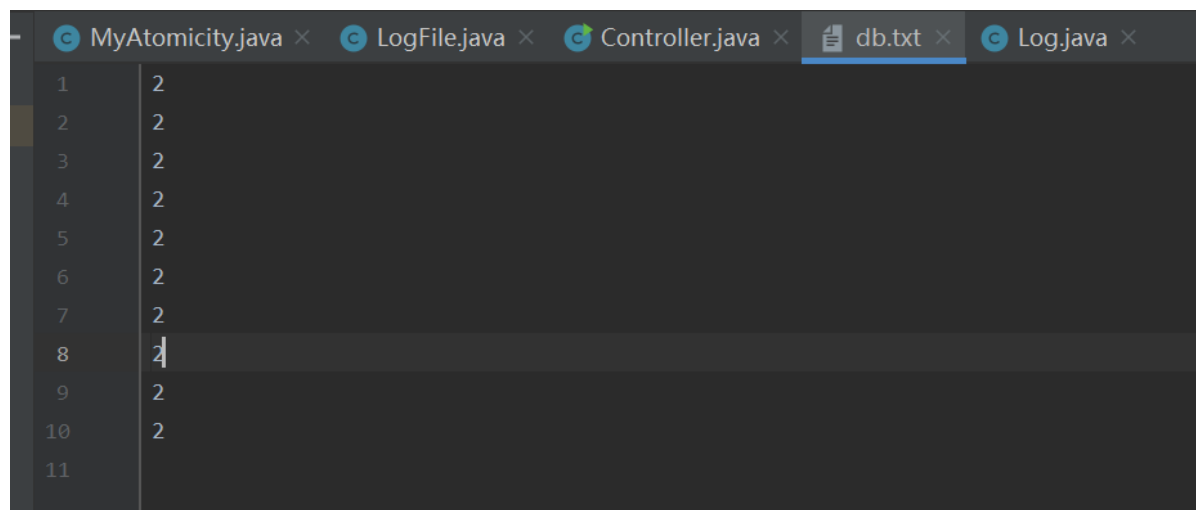
## 与提供模板的不同

- 我将recover函数搬到了用于总控的Controller类中，因为个人理解wal是为了保证all-or-nothing，而假设未曾断电，则每个正在修改的atom都有职责保证自身事务的原子性，因此不可能出现一个事务做到一半而没有结果的情况，只有中途断电，这个事务无法自己完成回滚，于是只好在下次运行时通过log的记录来补足回滚的步骤、完成原子性。
- 将write(int line, char ch)移除了，因为我们实际上没有对某行单独修改的操作，且在内存中修改也可以通过sleep完成强制休眠1000ms的要求，没有集成出一个函数的必要。
- 理论上recover()应该在运行后自动调用，但是为了测试断电后恢复前后log的区别，采用了输入指令调用的方式

## 测试过程&截图

- 运行Controller类的main函数即可开始操作
  - 输入的指令格式为<command> <atom\_id> <to\_write\_ch>(第一个为操作，第二个为要使用的atom号，第二个为要写入的character)
1. 首先测试能否正确更新

```
"D:\IntelliJ IDEA 2019.2.2\jbr\bin\java.exe" ...
Zsh: please input command(<command> <atom_id> <to_write_ch>)
put 1 2
Zsh: please input command(<command> <atom_id> <to_write_ch>)
Zsh: transaction commit
check
BEGIN NEW_TRANSACTION 1
CHANGE PUT 1 (0,0,2)
CHANGE PUT 1 (1,0,2)
CHANGE PUT 1 (2,0,2)
CHANGE PUT 1 (3,0,2)
CHANGE PUT 1 (4,0,2)
CHANGE PUT 1 (5,0,2)
CHANGE PUT 1 (6,0,2)
CHANGE PUT 1 (7,0,2)
CHANGE PUT 1 (8,0,2)
CHANGE PUT 1 (9,0,2)
OUTCOME COMMIT 1
Zsh: please input command(<command> <atom_id> <to_write_ch>)
```



The image shows a screenshot of an IDE with five tabs: MyAtomicity.java, LogFile.java, Controller.java, db.txt, and Log.java. The db.txt tab is active, displaying a list of 11 lines, each containing the number 2. The lines are numbered 1 through 11 on the left margin.

Line Number	Content
1	2
2	2
3	2
4	2
5	2
6	2
7	2
8	2
9	2
10	2
11	2

2. 然后测试两个线程并发时前者是否放弃，又是否成功重做

```
Zsh: please input command(<command> <atom_id> <to_write_ch>)
put 1 4
Zsh: please input command(<command> <atom_id> <to_write_ch>)
put 2 6
Zsh: please input command(<command> <atom_id> <to_write_ch>)
Zsh: transaction abort
Zsh: transaction commit
Retry
Zsh: transaction commit
check
BEGIN NEW_TRANSACTION 1
CHANGE PUT 1 (0,0,2)
CHANGE PUT 1 (1,0,2)
CHANGE PUT 1 (2,0,2)
CHANGE PUT 1 (3,0,2)
CHANGE PUT 1 (4,0,2)
CHANGE PUT 1 (5,0,2)
CHANGE PUT 1 (6,0,2)
CHANGE PUT 1 (7,0,2)
CHANGE PUT 1 (8,0,2)
CHANGE PUT 1 (9,0,2)
OUTCOME COMMIT 1
BEGIN NEW_TRANSACTION 1
CHANGE PUT 1 (0,2,4)
CHANGE PUT 1 (1,2,4)
CHANGE PUT 1 (2,2,4)
BEGIN NEW_TRANSACTION 2
```

```
BEGIN NEW_TRANSACTION 2
CHANGE PUT 2 (0,2,6)
CHANGE PUT 1 (3,2,4)
CHANGE PUT 2 (1,2,6)
CHANGE PUT 1 (4,2,4)
CHANGE PUT 2 (2,2,6)
CHANGE PUT 1 (5,2,4)
CHANGE PUT 2 (3,2,6)
CHANGE PUT 1 (6,2,4)
CHANGE PUT 2 (4,2,6)
CHANGE PUT 1 (7,2,4)
CHANGE PUT 2 (5,2,6)
CHANGE PUT 1 (8,2,4)
CHANGE PUT 2 (6,2,6)
CHANGE PUT 1 (9,2,4)
CHANGE PUT 2 (7,2,6)
OUTCOME ABORT 1
CHANGE PUT 2 (8,2,6)
CHANGE PUT 2 (9,2,6)
OUTCOME COMMIT 2
BEGIN NEW_TRANSACTION 3
CHANGE PUT 3 (0,6,4)
CHANGE PUT 3 (1,6,4)
CHANGE PUT 3 (2,6,4)
CHANGE PUT 3 (3,6,4)
CHANGE PUT 3 (4,6,4)
CHANGE PUT 3 (5,6,4)
CHANGE PUT 3 (6,6,4)
```

```

BEGIN NEW_TRANSACTION 3
CHANGE PUT 3 (0,6,4)
CHANGE PUT 3 (1,6,4)
CHANGE PUT 3 (2,6,4)
CHANGE PUT 3 (3,6,4)
CHANGE PUT 3 (4,6,4)
CHANGE PUT 3 (5,6,4)
CHANGE PUT 3 (6,6,4)
CHANGE PUT 3 (7,6,4)
CHANGE PUT 3 (8,6,4)
CHANGE PUT 3 (9,6,4)
OUTCOME COMMIT 3
Zsh: please input command(<command> <atom_id> <to_write_ch>)
put 3 6
Zsh: please input command(<command> <atom_id> <to_write_ch>)

```

从上图可以看到（第一个完整的更新流1是第一次测试，这里不用讨论），而后来的1受到了2的干扰，因此必须abort，同时2成功commit，在失败操作休眠一段事件后，之前abort的事务1进行了retry，重新获得了新的watermark3并成功更新并commit

此时db中的 数据记录为4

1	4
2	4
3	4
4	4
5	4
6	4
7	4
8	4
9	4
10	4
11	

3. 最后测试是否断电后会回滚上次中断的事务(上图中可看到最后一次更新未到结果即断电)

- 未recover前

```

OUTCOME COMMIT 3
BEGIN NEW_TRANSACTION 4
CHANGE PUT 4 (0,4,6)
CHANGE PUT 4 (1,4,6)
CHANGE PUT 4 (2,4,6)
Zsh: please input command(<command> <atom_id> <to_write_ch>)
recover
Zsh: please input command(<command> <atom_id> <to_write_ch>)

```

- recover后

```

OUTCOME COMMIT 3
BEGIN NEW_TRANSACTION 4
CHANGE PUT 4 (0,4,6)
CHANGE PUT 4 (1,4,6)
CHANGE PUT 4 (2,4,6)
OUTCOME ABORT 4
Zsh: please input command(<command> <atom_id> <to_write_ch>)

```

- 从上两图中可以看到已经完成了回滚以及添加outcome的log