Lab3实验文档

19302010084 程茜

PartA 汉明距离

- **实现：**

先将先将两个数做异或操作再将结果转换为二进制，二进制中1的个数即为所求的汉明距离。

```
 int hmNum=x^y;
        String hmNum10=parseTo01(hmNum);
        int count=0;
        for(int i=0;i<hmNum10.length();i++) {
            if(hmNum10.charAt(i)=='1') {
                count++;
            }
        }
return count;
```

- **计算机系统的可靠性：**

计算机系统的可靠性是在给定的时间内，计算机系统能实施应有功能的能力。冗余是一种提高可靠性很有效的方法，汉明距离就是通过冗余来实现可靠性的。同样地，TCP检验和也是通过冗余来提高可靠性的，TCP检验和在计算时要加上12byte的伪首部，检验范围包括TCP首部及数据部分，接收方收到报文后，就可以根据检验和来判断数据是否有错，如果校验和有差错，则TCP段会被直接丢弃，防止了不正确数据的写入。TCP也通过增加检验和字段提高了端到端传输的可靠性。

PartB

All-or-nothing:

- **实现**:

对于单个事务，如果有对文件内容的updata，则会先写入文件“log0.txt”中，写完之后写一条committed log，如果事务被中断，则会在log中写入一条aborted log。在事务写完之后进行recover，recover会从最后一行开始读取log文件，如果读到commit log，则会将修改写入数据库文件。如果读到abort则不会修改数据库文件：

```
// write
public void write(int index, char ch) {
        // TODO
        //transactionsOutcome.replace(getTransactionId(),"Pending");
        int tId = getTransactionId();
        char preData = (char)(ch-1); // 上一个事务写的字符
        if(tId != -1) {
            log("Change," + tId +","+index+"," + ch+"," + preData);
        }
    }

// recover
char updateDate = '0';
```

```
if (result.startsWith("Committed")) {
    findCommitted = true;
} else {
    if (findCommitted) {
        String[] logContent = result.split(",");
        updateDate = logContent[logContent.length - 2].charAt(0);
        break;
    }
}

if (updateDate != '0') {
    FileWriter fw = new FileWriter(databasePath);
    for (int i = 0; i < 10; i++) {
        fw.write(updateDate);
        fw.write("\n");
    }
    fw.close();
 }else {
    System.out.println("No update");
}
```

- **测试**

  让一个子进程进行两次update，每次update前先read数据库的值，update的值为read的值加1。update两次用时至少需要20秒，在主线程中等待子线程15秒后强行中断子线程，此时子线程应该处于第一次update成功commit而第二次未成功的状态，最终的写入数据库的值应该为1：

```
myAtomicity.begin();
char preData = myAtomicity.read();
char newData = (char) (preData +1);
myAtomicity.update(newData);
myAtomicity.update((char) (newData+1));
myAtomicity.recover();
```

- **运行结果：**

```
Transaction 1 update is finished! value is: 1
Interrupted
final result is 1
```

log内容：

log内容为：类型，事务id，修改行数，更新值，旧值

```
Begin,1
Change,1,0,1,0
Change,1,1,1,0
Change,1,2,1,0
Change,1,3,1,0
Change,1,4,1,0
Change,1,5,1,0
Change,1,6,1,0
Change,1,7,1,0
Change,1,8,1,0
Change,1,9,1,0
Committed,1
Change,1,0,2,1
Change,1,1,2,1
Change,1,2,2,1
Change,1,3,2,1
Change,1,4,2,1
Aborted,1
```

Read-Capture:

- **实现**

  在开始时给每个事务一个唯一的id，这里用的是lab2实现的lock：

```
public void begin() {
    try {
        lock.lock();
        addTransaction( transactionId: transactionsThread.size() + 1, Thread.currentThread().getId());
        System.out.println("Transaction "+getTransactionId()+", is thread "+Thread.currentThread().getId());
        //read();
        lock.unLock();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

具体实现是每个事务在update之前都会**先进行一次read**，然后update的值为read的值**加1**。

- **read**：事务开始update会将自己的状态改为pending，链表versionId用于记录对文件进行修改过的事务id，highWaterMarket用于记录读取过数据库数据的最大的事务id。进行读操作的时候会从versionId从后往前遍历事务vId，如果vId大于当前事物的id则继续往前遍历，反之则判断事务vId的状态，如果正在pending状态则等待该事务直到其Commit或abort，如果abor则继续向前寻找，如果commit则读取该事务修改后的值并判断highWaterMarket是否需要改变：

```
for(int i = versionId.size()-1; i >= 0; i--){
    int vId = versionId.get(i);
    // 修改数据的事务id大于当前事务的id则继续往前寻找
    if(vId > tId){
        continue;
    }
    try {
        int time = (int)(1+Math.random()*5);
        Thread.sleep( millis: time*1000);
        //Thread.sleep(1000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    //等待直到之前的事务结束
    while (transactionsOutcome.get(vId).equals("Pending")){
        try {
            int time = (int)(1+Math.random()*5);
            Thread.sleep( millis: time*1000);
        } catch (InterruptedException e) {
            System.out.println(e.getMessage());
        }
    }
    if(transactionsOutcome.get(vId).equals("Committed") ){
      // 修改highWaterMarket
        highWaterMarket = Math.max(tId,highWaterMarket);
        return transactionValue.get(vId);
    }
```

- **update**：在update时会先判断highWaterMarket是否大于当前事务id，如果大于则会直接abort，反之则进行write操作，这里也会先写入log中。在写完所有行commit之前也会判断highWaterMarket是否大于当前事务id，如果大于则会直接abort，反之则commit（因为可能有后面的事务在当前事务判断了highWaterMarket后但修改事务状态之间进行了读取操作，当前事务也该被abort）：

```
for(int i = 0; i < 10; i++){
    //System.out.println(tId+" verion "+highWaterMarket);
    write(i,ch);
    try {
        Thread.sleep( millis: 1000);
    } catch (InterruptedException e) {
        System.out.println("Interrupted");
        transactionsOutcome.replace(tId,"Aborted");
        abort();
        return;
    }
}
if(highWaterMarket > tId){
    //isAborted = true;
    System.out.println("Transaction "+tId+" abort!");
    abort();
    return;
}
transactionsOutcome.replace(tId,"Pending");
System.out.println("Transaction "+tId+" update is finished! value is: "+ch);
transactionValue.replace(tId,ch);
//log("Change," + tId); //+","+index+"," + ch+"," + preData);
commit();
```

- **运行截图：**

  可以看到事务3在事务1、2之前读取了数据，所以事务1,2被abort，而事务4,5,6均在等待事务3执行完后才读取到数据，事务5和6几乎同时读取到事务4更新后的数据，所以事务5被abort，最后得到的结果为3：

```
Transaction 1, is thread 14
Transaction 2, is thread 16
Transaction 3, is thread 12
Transaction 4, is thread 17
Transaction 5, is thread 13
Transaction 6, is thread 15
Transaction 3 read value is: 0
Transaction 1 read value is: 0
Transaction 1 abort!
Transaction 3 update is finished! value is: 1
Transaction 4 read value is: 1
Transaction 2 read value is: 0
Transaction 2 abort!
Transaction 4 update is finished! value is: 2
Transaction 5 read value is: 2
Transaction 6 read value is: 2
Transaction 5 abort!
Transaction 6 update is finished! value is: 3
finish
final result is 3
```

```
3
3
3
3
3
3
3
3
3
3
```