

## 43 | 软件事务内存：借鉴数据库的并发经验

2019-06-06 王宝令



很多同学反馈说，工作了挺长时间但是没有机会接触并发编程，实际上我们天天都在写并发程序，只不过并发相关的问题都被类似Tomcat这样的Web服务器以及MySQL这样的数据库解决了。尤其是数据库，在解决并发问题方面，可谓成绩斐然，它的事务机制非常简单易用，能用Java里面的锁、原子类十条街。技术无边界，很显然要借鉴一下。

其实很多编程语言都有从数据库的事务管理中获得灵感，并且总结出了一个新的并发解决方案：软件事务内存（**Software Transactional Memory**，简称**STM**）。传统的数据库事务，支持4个特性：原子性（**Atomicity**）、一致性（**Consistency**）、隔离性（**Isolation**）和持久性（**Durability**），也就是大家常说的**ACID**，STM由于不涉及到持久化，所以只支持**ACI**。

STM的使用很简单，下面我们以经典的转账操作为例，看看用STM该如何实现。

### 用STM实现转账

我们曾经在[《05 | 一不小心就死锁了，怎么办？》](#)这篇文章中，讲到了并发转账的例子，示例代码如下。简单地使用 `synchronized` 将 `transfer()` 方法变成同步方法并不能解决并发问题，因为还存在死锁问题。

```

class UnsafeAccount {
    //余额
    private long balance;
    //构造函数
    public UnsafeAccount(long balance) {
        this.balance = balance;
    }
    //转账
    void transfer(UnsafeAccount target, long amt){
        if (this.balance > amt) {
            this.balance -= amt;
            target.balance += amt;
        }
    }
}

```

该转账操作若使用数据库事务就会非常简单，如下面的示例代码所示。如果所有SQL都正常执行，则通过 **commit()** 方法提交事务；如果SQL在执行过程中有异常，则通过 **rollback()** 方法回滚事务。数据库保证在并发情况下不会有死锁，而且还能保证前面我们说的原子性、一致性、隔离性和持久性，也就是**ACID**。

```

Connection conn = null;
try{
    //获取数据库连接
    conn = DriverManager.getConnection();
    //设置手动提交事务
    conn.setAutoCommit(false);
    //执行转账SQL
    .....
    //提交事务
    conn.commit();
} catch (Exception e) {
    //出现异常回滚事务
    conn.rollback();
}

```

那如果用STM又该如何实现呢？Java语言并不支持STM，不过可以借助第三方的类库来支持，[Multiverse](#)就是个不错的选择。下面的示例代码就是借助Multiverse实现了线程安全的转账操作，相比较上面线程不安全的UnsafeAccount，其改动并不大，仅仅是将余额的类型从 long 变成了 TxnLong，将转账的操作放到了 atomic()->{} 中。

```
class Account{
    //余额
    private TxnLong balance;
    //构造函数
    public Account(long balance){
        this.balance = StmUtils.newTxnLong(balance);
    }
    //转账
    public void transfer(Account to, int amt){
        //原子化操作
        atomic()->{
            if (this.balance.get() > amt) {
                this.balance.decrement(amt);
                to.balance.increment(amt);
            }
        };
    }
}
```

一个关键的atomic()方法就把并发问题解决了，这个方案看上去比传统的方案的确简单了很多，那它是如何实现的呢？数据库事务发展了几十年了，目前被广泛使用的是MVCC（全称是Multi-Version Concurrency Control），也就是多版本并发控制。

MVCC可以简单地理解为数据库事务在开启的时候，会给数据库打一个快照，以后所有的读写都是基于这个快照的。当提交事务的时候，如果所有读写过的数据在该事务执行期间没有发生过变化，那么就可以提交；如果发生了变化，说明该事务和有其他事务读写的数据冲突了，这个时候是不可以提交的。

为了记录数据是否发生了变化，可以给每条数据增加一个版本号，这样每次成功修改数据都会增加版本号的值。MVCC的工作原理和我们曾经在[《18 | StampedLock: 有没有比读写锁更快的锁？》](#)中提到的乐观锁非常相似。有不少STM的实现方案都是基于MVCC的，例如知名的Clojure STM。

下面我们就用最简单的代码基于MVCC实现一个简版的STM，这样你会对STM以及MVCC的工作原理有更深入的认识。

## 自己实现STM

我们首先要做的，就是让Java中的对象有版本号，在下面的示例代码中，VersionedRef这个类的作用就是将对象value包装成带版本号的对象。按照MVCC理论，数据的每一次修改都对应着一个唯一的版本号，所以不存在仅仅改变value或者version的情况，用不变性模式就可以很好地解决这个问题，所以VersionedRef这个类被我们设计成了不可变的。

所有对数据的读写操作，一定是在一个事务里面，TxnRef这个类负责完成事务内的读写操作，读写操作委托给了接口Txn，Txn代表的是读写操作所在的当前事务，内部持有的curRef代表的是系统中的最新值。

```

//带版本号的对象引用
public final class VersionedRef<T> {
    final T value;
    final long version;

    //构造方法
    public VersionedRef(T value, long version) {
        this.value = value;
        this.version = version;
    }
}

//支持事务的引用
public class TxnRef<T> {
    //当前数据，带版本号
    volatile VersionedRef curRef;

    //构造方法
    public TxnRef(T value) {
        this.curRef = new VersionedRef(value, 0L);
    }

    //获取当前事务中的数据
    public T getValue(Txn txn) {
        return txn.get(this);
    }

    //在当前事务中设置数据
    public void setValue(T value, Txn txn) {
        txn.set(this, value);
    }
}

```

**STMTxn**是**Txn**最关键的一个实现类，事务内对于数据的读写，都是通过它来完成的。**STMTxn**内部有两个**Map**：**inTxnMap**，用于保存当前事务中所有读写的数据的快照；**writeMap**，用于保存当前事务需要写入的数据。每个事务都有一个唯一的事务ID **txnId**，这个**txnId**是全局递增的。

**STMTxn**有三个核心方法，分别是读数据的**get()**方法、写数据的**set()**方法和提交事务的**commit()**方法。其中，**get()**方法将要读取数据作为快照放入**inTxnMap**，同时保证每次读取的数据都是一个版本。**set()**方法会将要写入的数据放入**writeMap**，但如果写入的数据没被读取过，也会将其放入 **inTxnMap**。

至于**commit()**方法，我们为了简化实现，使用了互斥锁，所以事务的提交是串行的。**commit()**方法的实现很简单，首先检查**inTxnMap**中的数据是否发生过变化，如果没有发生变化，那么就将**writeMap**中的数据写入（这里的写入其实就是**TxnRef**内部持有的**curRef**）；如果发生过变化，那么就不能将**writeMap**中的数据写入了。

```
//事务接口
public interface Txn {
    <T> T get(TxnRef<T> ref);
    <T> void set(TxnRef<T> ref, T value);
}

//STM事务实现类
public final class STMTxn implements Txn {
    //事务ID生成器
    private static AtomicLong txnSeq = new AtomicLong(0);

    //当前事务所有的相关数据
    private Map<TxnRef, VersionedRef> inTxnMap = new HashMap<>();
    //当前事务所有需要修改的数据
    private Map<TxnRef, Object> writeMap = new HashMap<>();
    //当前事务ID
    private long txnId;
    //构造函数，自动生成当前事务ID
    STMTxn() {
        txnId = txnSeq.incrementAndGet();
    }

    //获取当前事务中的数据
    @Override
    public <T> T get(TxnRef<T> ref) {
        //将需要读取的数据，加入inTxnMap
        if (!inTxnMap.containsKey(ref)) {
            inTxnMap.put(ref, ref.curRef);
        }
        return (T) inTxnMap.get(ref).value;
    }

    //在当前事务中修改数据
    @Override
```

```

public <T> void set(TxnRef<T> ref, T value) {
    //将需要修改的数据，加入inTxnMap
    if (!inTxnMap.containsKey(ref)) {
        inTxnMap.put(ref, ref.curRef);
    }
    writeMap.put(ref, value);
}

//提交事务
boolean commit() {
    synchronized (STM.commitLock) {
        //是否校验通过
        boolean isValid = true;
        //校验所有读过的数据是否发生过变化
        for(Map.Entry<TxnRef, VersionedRef> entry : inTxnMap.entrySet()){
            VersionedRef curRef = entry.getKey().curRef;
            VersionedRef readRef = entry.getValue();
            //通过版本号来验证数据是否发生过变化
            if (curRef.version != readRef.version) {
                isValid = false;
                break;
            }
        }
        //如果校验通过，则所有更改生效
        if (isValid) {
            writeMap.forEach((k, v) -> {
                k.curRef = new VersionedRef(v, txnId);
            });
        }
        return isValid;
    }
}

```

下面我们来模拟实现Multiverse中的原子化操作atomic()。atomic()方法中使用了类似于CAS的操作，如果事务提交失败，那么就重新创建一个新的事务，重新执行。

```

@FunctionalInterface
public interface TxnRunnable {
    void run(Txn txn);
}

//STM
public final class STM {
    //私有化构造方法
    private STM() {
        //提交数据需要用到的全局锁
        static final Object commitLock = new Object();
        //原子化提交方法
        public static void atomic(TxnRunnable action) {
            boolean committed = false;
            //如果没有提交成功，则一直重试
            while (!committed) {
                //创建新的事务
                STMTxn txn = new STMTxn();
                //执行业务逻辑
                action.run(txn);
                //提交事务
                committed = txn.commit();
            }
        }
    }
}

```

就这样，我们自己实现了**STM**，并完成了线程安全的转账操作，使用方法和**Multiverse**差不多，这里就不赘述了，具体代码如下面所示。



```
class Account {  
    //余额  
    private TxnRef<Integer> balance;  
    //构造方法  
    public Account(int balance) {  
        this.balance = new TxnRef<Integer>(balance);  
    }  
    //转账操作  
    public void transfer(Account target, int amt){  
        STM.atomic((txn)->{  
            Integer from = balance.getValue(txn);  
            balance.setValue(from-amt, txn);  
            Integer to = target.balance.getValue(txn);  
            target.balance.setValue(to+amt, txn);  
        });  
    }  
}
```

## 总结

**STM**借鉴的是数据库的经验，数据库虽然复杂，但仅仅存储数据，而编程语言除了有共享变量之外，还会执行各种I/O操作，很显然I/O操作是很难支持回滚的。所以，**STM**也不是万能的。目前支持**STM**的编程语言主要是函数式语言，函数式语言里的数据天生具备不可变性，利用这种不可变性实现**STM**相对来说更简单。

另外，需要说明的是，文中的“自己实现**STM**”部分我参考了[Software Transactional Memory in Scala](#)这篇博文以及[一个GitHub项目](#)，目前还很粗糙，并不是一个完备的MVCC。如果你对这方面感兴趣，可以参考[Improving the STM: Multi-Version Concurrency Control](#)这篇博文，里面讲到了如何优化，你可以尝试学习下。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

# Java 并发编程实战

全面系统提升你的并发编程能力

王宝令

资深架构师



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言



我的腿腿

我公司用的就是这个解决并发问题的，才知道是这种技术

2019-06-06

👍 2



Geek\_89bbab

上面的STMTxn实现是不是有问题？只能每次get都是获取到最开始快照的值。

例如 `v = tx.get(ref); // v=10`

`tx.set(ref, v+20); // v=30`

`v = tx.get(ref); // v = 10` 这里应该期望的是 30,但是实际却得到10.

我觉得 正确的STMTxn实现方法应该像老师给出的链接里面的这个的实现方法

<https://github.com/epam-mooc/stm-java/blob/master/src/concurrency/stm/Transaction.java>

2019-06-08

👍 1



QQ怪

哔，打卡，涨知识了

2019-06-06

👍 0



有铭

老师，关系数据库也是有死锁的，只是他们往往实现了死锁检测机制，死锁到一定时间就会强制解锁

2019-06-06

👍 0



黄海峰

代码里硬是没看到哪里修改了version。。

2019-06-06

👍 0

作者回复

只创建新的版本，永远不会去修改

2019-06-10



张三

打卡！这篇高质量！

2019-06-06

👍 0



爱吃回锅肉的瘦子

涨见识了，谢谢老师。

2019-06-06

👍 0