Clojure STM简介与代码分析

xumingmingv

Why STM?

“

就像垃圾回收机制使我们不用再手动管理内存——从而避免了一堆微妙的bug，STM则经常被认为是对于另一个非常容易出错的编程实践的系统性简化：手动锁管理。

”

—— 《Clojure编程》

STM特性

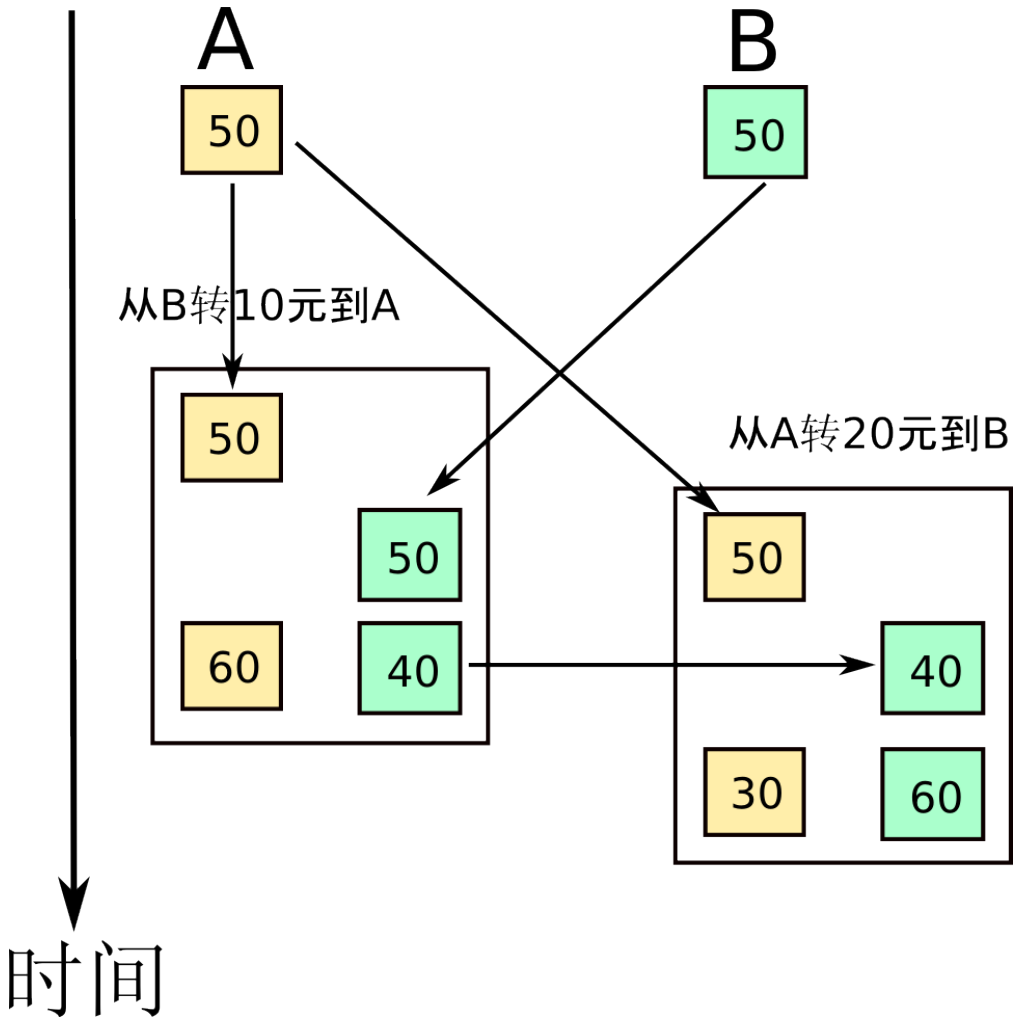
- 原子性(Atomicity): 对于一个事务里面的所有ref的操作要么都成功，要么都失败。
- 一致性(Consistency): 在事务发生前后ref的数据一致性是得到保证的
- 隔离性(Isolation): 如果多个事务同时操作一个ref，一个事务不会看到其它事务对一个ref的修改。

太难懂了！来点通俗的！

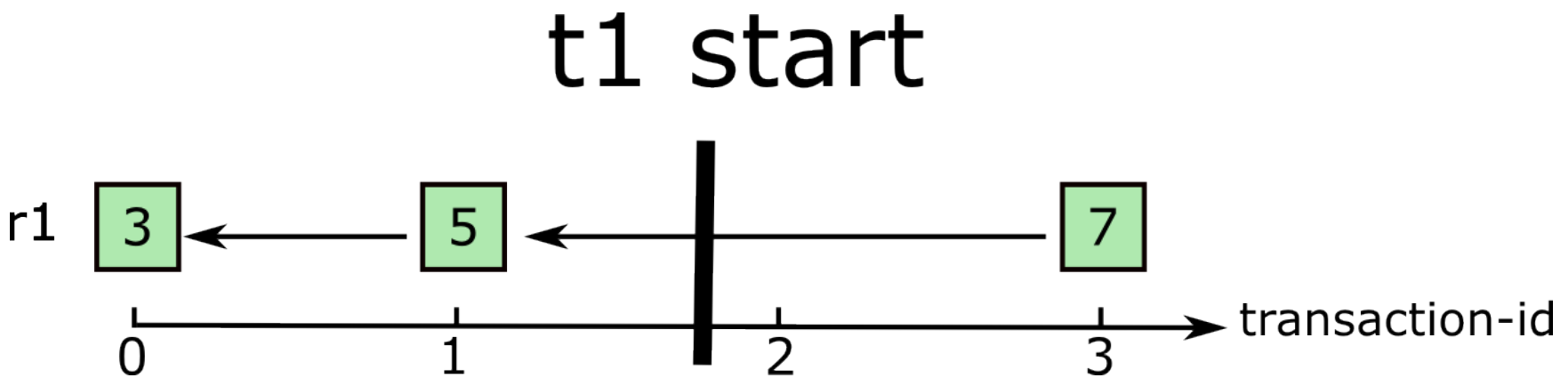
举个简单例子

- 有A、B两人，各有50元，两人之间会发生金钱来往。
- 原子性：A转10元给B。从A扣10元、向B加10元，两个要么同时成功，要么同时失败。
- 一致性：两个人总钱数只能是100元，多了少了都是不一致。

(不)隔离性



Clojure STM: MVCC



STM相关源码文件

- clojure/core.clj
- clojure/lang/Ref.java
- clojure/lang/LockingTransaction.java

Clojure的STM接口

```
;; Clojure STM API
;; dosync划定事务边界
(dosync
 ;; 把r1修改成给定值
 (ref-set r1 1)
 ;; 以r2的当前值作为参数调用inc,
 ;; 返回值作为r2的新值
 (alter r2 inc)
 ;; 用法跟alter类似，以r3当前值
 ;; 来调用inc，返回值作为r3新值
 ;; 并发性能比ref-set, alter
 ;; 要好，但是不是所有场景可用
 (commute r3 inc)
 ;; 确保在当前事务执行过程中，r4
 ;; 不会被别的事务修改
 (ensure r4)
 ;; 获取r5的事务内的值
 @r5)
```

用Clojure来写银行转账例子

```
27
28 ;; 银行转账例子: 从a转10元到b
29 (let [a (ref 50)
30       b (ref 50)]
31   (dosync (alter a + 10)
32           (alter b - 10))
33   [@a @b])
34
```

dosync

```
344 Object run(Callable fn) throws Exception{
345     boolean done = false;
346     Object ret = null;
347     /**
348      * 所有上了写锁的ref
349      */
350     ArrayList<Ref> locked = new ArrayList<Ref>();
351     /**
352      * 所有要发的notify消息
353      */
354     ArrayList<Notify> notify = new ArrayList<Notify>();
355
356     for(int i = 0; !done && i < RETRY_LIMIT; i++)
357         {
358         System.out.println("[tx:" + this.hashCode() + "] round:" + i);
359         try
360             {
361             getReadPoint();
362             if(i == 0)
363                 {
364                 startPoint = readPoint;
365                 startTime = System.nanoTime();
366                 }
367             info = new Info(RUNNING, startPoint);
368             ret = fn.call();
369             //make sure no one has killed us before this point, and can't from now on
370             // 这里先compareset一下，确保没有别的线程已经把我们干掉了(改了status状态)，一旦状态改成COMMITING之后别人就干不了我们了
371             if(info.status.compareAndSet(RUNNING, COMMITTING))
372                 {
373                 for(Map.Entry<Ref, ArrayList<CFn>> e : commutes.entrySet())
374                     {
375                     Ref ref = e.getKey();
```

我们dosync里面的代码被包装成一个匿名函数，而匿名函数是Callable的实例

整个包在一个for循环里面，RETRY_LIMIT是做大的重试次数

我们的代码在这里被调用，ref-set, alter commute等等调用都发生在这里

ref-set -- 执行阶段

- 首先检查当前线程上有没有正在运行的事务
  - 如果没有的话会抛出异常
  - `ref-set, alter, commute`等等这些方法只能在事务内(`dosync`)内部调用
- 获取ref上的写锁。如果获取写锁超时，那么整个事务重试。
- 检查事务开始之后别的事务是否修改过ref，如果改过，那么当前事务重试。

ref-set

- 检查是否有其它事务*正在*修改这个ref，如果有，那么开始早的事务继续运行，另外一个重试
- 在ref上标记 —— 告诉别的事务我正在操作这个ref
- 释放写锁
- 把ref的事务内值更新为传入值

ref-set – 提交阶段

- 对于所有的被ref-set的ref，获取写锁
  - Ref要被提交的值已经在vals里面了
- 提交ref的新值
- 释放锁

ref-set冲突

```
34
35 ;; ref-set冲突，同一时间只有一个事务能获取到一个ref
36 ;; 的锁，从而可以对ref进行修改，其它的事务则必须重试
37 (let [r (ref 1)
38       t1 (future (dosync_xumm ["t1"]
39                               (Thread/sleep 1000)
40                               (ref-set r 101)))
41       t2 (future (dosync_xumm ["t2"]
42                               (Thread/sleep 5000)
43                               (ref-set r 201)))]
44   [@t1 @t2 @r])
45
```

alter

```
227
228 /**
229  * 设置ref的值: (ref-set ref val)
230  * @param val
231  * @return
232  */
233 public Object set(Object val){
234     return LockingTransaction.getEx().doSet(this, val);
235 }
236
```

```
250 /**
251  * alter其实就是ref-set只不过提供的参数不是最终的newv，而是fn + args
252  * @param fn
253  * @param args
254  * @return
255  */
256 public Object alter(IFn fn, ISeq args) {
257     LockingTransaction t = LockingTransaction.getEx();
258     return t.doSet(this, fn.applyTo(RT.cons(t.doGet(this), args)));
259 }
260
```

alter

- alter方法传进来的不是ref的新值，而是一个函数，以及一些参数，ref的旧值会作为函数的第一个参数，调用的返回值则作为ref的新值。

commute -- 执行阶段

- commute可以看成特定场景下，对于ref-set的性能优化方案：
  - 多个事务对ref的修改的顺序对于最终ref的状态值无影响(commutative)。
  - (= (+ 1 2 3) (+ 1 3 2))
- 给ref加读锁
- 获取ref值作为事务内值。
- 计算ref的新的事务内的值
- 保存传入的函数以及参数

commute – 提交阶段

- 获取写锁
- 以ref的最新值重新调用传入函数，计算ref的新值 – 最终要被提交的值。
- 释放锁

commute绝不冲突

```
47 ;; commute不冲突，多个事务可以同时操作一个ref
48 ;; 事务也不会重试
49 (let [r (ref 1)
50       t1 (future (dosync
51                   (Thread/sleep 1000)
52                   (commute r + 100)))
53       t2 (future (dosync
54                   (Thread/sleep 5000)
55                   (commute r + 200)))]
56   [@t1 @t2 @r])
```

commute VS alter(ref-set)

```
58 ;; commute比alter(ref-set)的并发性能要好
59 (time (let [r (ref 1)
60               futures (for [i (range 100)]
61                         (future (dosync_xumm [(str i)]
62                                              (alter r inc))))]
63           (doseq [f futures]
64             (deref f))
65           @r))
66
67 (time (let [r (ref 1)
68               futures (for [i (range 100)]
69                         (future (dosync_xumm [(str i)]
70                                              (commute r inc))))]
71           (doseq [f futures]
72             (deref f))
73           @r))
74
```

ensure – 执行阶段

- 给ref上读锁
- 如果在事务开始之后别的事务写过这个ref，那么释放读锁并重试。
- 检查是否有事务正在对这个ref进行修改，如果有，释放读锁，并且检查对ref修改的事务是否是当前事务
  - 如果是，那么成功返回
  - 否则停止当前事务并重试

ensure – 提交阶段

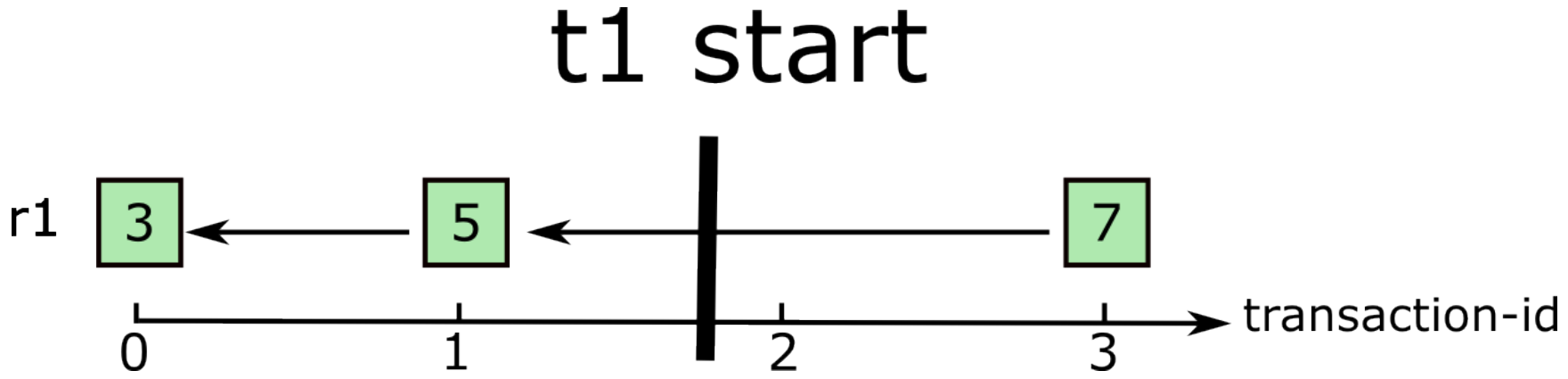
- 释放读锁

ensure

```
76 ;; ensure保证ref在事务执行过程中不被修改
77 (let [r (ref 1)
78       f1 (future (dosync_xumm ["f1"]
79                               (ensure r)
80                               (Thread/sleep 1000)))
81
82       f2 (future (dosync_xumm ["f2"]
83                               (ref-set r 2)))]
84   [@f1 @f2])
85
```

deref

- 如果当前有事务在运行，那么获取ref在事务内的值。
- 否则直接获取它的最新值
  - 遍历ref的MVCC链，找到离当前事务开始之前、并且离开始时间最近的那个版本。



只读事务也可能重试

```
199 ;; 只读事务也可能会重试
200 (let [r (ref 1)
201       f1 (future (dosync_xumm ["f1"]
202                               (Thread/sleep 2000)
203                               @r))
204       f2 (future (dosync_xumm ["f2"]
205                               (ref-set r 2)))]
206   [@f1 @f2])
207
```

探查MVCC – ref-history-count

```
206
207 ;; 探查ref-history-count
208 (let [r (ref 1)
209       f1 (future (dosync_xumm ["f1"]
210                   (Thread/sleep 100)
211                   (ref-set r 1)))
212       f2 (future (dosync_xumm ["f2"]
213                   (Thread/sleep 200)
214                   (ref-set r 2)))]
215   [@f1 @f2]
216   (println "ref-history-count: " (ref-history-count r)))
217
```

ref-history-count会是多少？2？

其实。。。

```
206
207 ;; 探查ref-history-count
208 (let [r (ref 1)
209       f1 (future (dosync_xumm ["f1"]
210                   (Thread/sleep 100)
211                   (ref-set r 1)))
212       f2 (future (dosync_xumm ["f2"]
213                   (Thread/sleep 200)
214                   (ref-set r 2)))]
215   [@f1 @f2]
216   (println "ref-history-count: " (ref-history-cour
 - 6.6k core.clj          Clojure Paredit Undo-Tree nREPL
304 user>
305 user>
306 user>
307 ref-history-count:  0
```

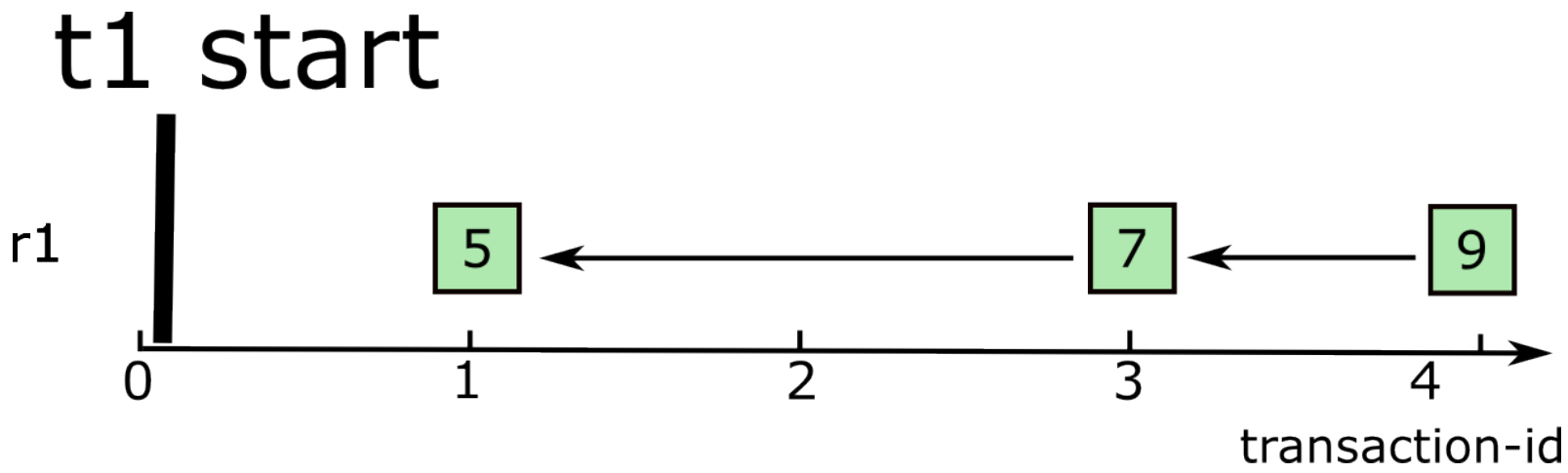
探查MVCC – ref-history-count

- Clojure STM对MVCC做了优化，ref的value链只在如下两种情况下增长：
  - min-history > 0
  - 产生过“读失败”(read faults)

min-history > 0

```
217
218 ;; 由于min-history > 0, 所以ref-history-count会增加
219 (let [r (ref 1 :min-history 1)]
220   @(future (dosync_xumm ["f1"]
221                         (Thread/sleep 1000)
222                         (ref-set r :f1)
223                         @r))
224   (println (ref-history-count r)))
225
```

read faults



read faults

```
226 ;; 由于有读失败，所以ref-history-count会增加
227 (let [r (ref 1)
228       f1 (future (dosync_xumm ["f1"]
229                               (Thread/sleep 1000)
230                               @r))
231       f2 (future (dosync_xumm ["f2"]
232                               (ref-set r 2)))
233       f3 (future (dosync_xumm ["f3"]
234                               (Thread/sleep 1500)
235                               (ref-set r 2)))]
236   [@f1 @f2 @f3]
237   (println "ref-history-count: " (ref-history-count r)))
```

STM避免了典型的并发问题

- 死锁
- 活锁

并发控制问题：死锁

- ref-set, alter都是排他的，不会有两个事务因为ref-set，alter去请求同一个ref的写锁
- commute会按照固定的顺序来请求写锁

```
148 /**
149  * 一个commit其实就是一个函数调用，每个ref可以对应多个commute
150  */
151 final TreeMap<Ref, ArrayList<CFn>> commutes = new TreeMap<Ref, ArrayList<CFn>>();
152
```

并发控制问题: 活锁

“一个长事务始终得不到执行的机会 ——”

因为总是获取不到锁，但是不死锁。

两个事务同时去修改一个ref，Clojure STM会优先让开始运行比较早的那个事务运行，一个事务再长，最终总会获得运行的机会 —— 等它足够老，老到可以获取到所有它想获得的锁。

```
;; 一个再长的事务都不会被"活锁"
(let [fw (FileWriter. "/tmp/stm.log")
      r (ref 1)
      exit? (atom false)]
  (binding [*out* fw]
    ;; 添加一个watcher，这样在LONG这个事务执行
    ;; 完成的时候会把exit?置为true
    (add-watch r :key
               (fn [k tr old-value new-value]
                 (when (= "LONG-VALUE" new-value)
                   (println "Long transaction successfully executed!")
                   (reset! exit? true))))


    ;; 执行一个很长的事务
    (future (dosync_xumm ["LONG"]
                         (ref-set r "LONG-VALUE")
                         (Thread/sleep 2000)))

    ;; 不断的生成新的事务，直到exit?为true
    (loop [i 0]
      (when-not @exit?
        (future (dosync_xumm [(str i)]
                             (ref-set r i)
                             (Thread/sleep 100)))
        (Thread/sleep 100)
        (recur (inc i))))))
```

参考资料

- (修改过的)Clojure STM源代码:
  - https://github.com/xumingming/clojure/blob/master/src/clj/clojure/core.clj
  - https://github.com/xumingming/clojure/blob/master/src/jvm/clojure/lang/Ref.java
  - https://github.com/xumingming/clojure/blob/master/src/jvm/clojure/lang/LockingTransaction.java
- R. Mark Volkmann有关STM的文章:
  - http://java.ociweb.com/mark/stm/article.html
- Demo代码下载地址
  - https://github.com/xumingming/stm-demo

Q & A