ゼミ: Learning Concurrent Programming in Scala: p224-

久野　靖*

2015.12.11

1　入れ子トランザクション

□ 2章で「synchronizedの中にまたsynchronizedを入れられる」

- このような機能は複数モジュールを組み合わせる時必須
- 例:　送金モジュールが記録のためロギングモジュールを呼ぶ
- どちらのモジュールも内部でsynchronizedを使っている可能性

□ atomic文も同様に入れ子にできる

- あるモジュールのトランザクション内で別モジュールのトランザクションを含んだ機能を呼び出すことはやりたい
- どこの中でどんなトランザクションとか知らずに済む→相互依存性が小さい

□ 具体例として、前にやったNodeは低レベル過ぎ

- ここでは整列リストのクラスTSortedListを例題に使う
- 内部では整数が昇順に並んでいる
- トランザクション参照headを内部に持ちここから連鎖
- toStringメソッドにより文字列化

```
package org.learningconcurrency
import scala.concurrent._
import scala.concurrent.stm._
import ExecutionContext.Implicits.global
import scala.annotation.tailrec

object NodeTrans2 extends App {
  case class Node(elem: Int, next: Ref[Node]) {
    def append(n: Node): Unit = atomic {
        implicit txn =>
      val oldNext = next()
      next() = n
      n.next() = oldNext
    }
    def nextNode: Node = next.single()
  }
  def nodeToString(n: Node): String = atomic {
      implicit txn =>
    val b = new StringBuilder
    var curr = n
    while(curr != null) {
      b ++= s"${curr.elem}, "
      curr = curr.next()
    }
    b.toString
  }
  class TSortedList {
    val head = Ref[Node](null)
    override def toString: String = atomic {
        implicit txn =>
      val h = head()
      nodeToString(h)
    }
    def insert(x: Int): this.type = atomic {
        implicit txn =>
      @tailrec def insert(n: Node): Unit = {
        if(n.next() == null || n.next().elem > x)
          n.append(new Node(x, Ref(null)))
        else insert(n.next())
      }
      if(head() == null || head().elem > x)
        head() = new Node(x, Ref(head()))
      else insert(head())
      this
    }
  }
  val sortedList = new TSortedList
  val f = Future {
    sortedList.insert(1); sortedList.insert(4) }
  val g = Future {
    sortedList.insert(2); sortedList.insert(3) }
  for(_ <- f; _ <- g) log(s"sorted list - $sortedList")
}
```

□ メソッドnodeToStringは内部で別のトランザクションを持つ→入れ子トランザクション

- nodeToString内のatomicは新しいトランザクションを開始するのでなく既に走っているトランザクションの一部となる。帰結:
- 内側のatomicがロールバックしたときは内側のatomicが再実行されるのではなく外側のtoStringのatomicがロールバック(トランザクションは動的スコープ)
- 同様に内側がcommitしても外側がcommitするまで確定しない

□ リストに要素を(正しい位置に)挿入するメソッドinsertを作る

*筑波大学ビジネスサイエンス系

- insert は head も内部のノードも変更する可能性あり→トランザクションとして実行
- 特殊ケース： リストが空、ないし先頭要素の手前に挿入
- それ以外は末尾再帰で

```
  def insert(x: Int): this.type = atomic {
      implicit txn =>
    @tailrec def insert(n: Node): Unit = {
      if(n.next() == null || n.next().elem > x)
        n.append(new Node(x, Ref(null)))
      else insert(n.next())
    }
    if(head() == null || head().elem > x)
      head() = new Node(x, Ref(head()))
    else insert(head())
    this
  }
}
```

□ 先のコードではトランザクションコンテキスト txn は外側のものを内側のメソッド内から参照していた

- 内側のメソッドにしない場合は渡す必要

```
@tailrec
final def insert(n: Node, x: Int)
    (implicit txn: InTxn):Unit = {
  if(n.next() == null || n.next().elem > x)
    n.append(new Node(x, Ref(null)))
  else insert(n.next(), x)
}
```

□ 別案として implicit txn をやめてもいいが、その場合 @tailrec の中でトランザクション開始することに

- 少し効率が悪いかも知れないが意味的には同じ

```
@tailrec
final def insert(n: Node, x: INt):Unit = atomic {
    implict txn =>
  if(n.next() == null || n.next().elem > x)
    n.append(new Node(x, Ref(null)))
  else insert(n.next(), x)
}
```

□ テスト用コードはこんな感じ

```
val sortedList = new TSortedList
val f = Future {
  sortedList.insert(1); sortedList.insert(4) }
val g = Future {
  sortedList.insert(2); sortedList.insert(3) }
for(_ <- f; _ <- g) log(s"sorted list - $sortedList")
```

□ Future のスケジュールに関わらず 1,2,3,4 の順

- 並行動作する整列リストができ、実装は逐次版とほぼ同じ
- 並行性にあまり煩わされずに並行データ構造が作れる

□ あと 1 つ： 例外が起きたら？ 次に扱う

2 トランザクションと例外

□ このまでのところ、例外だとどうなるかは明確でない

- ロールバックもコミットもあり得る。ScalaSTM ではデフォルトでは前者

□ たとえば先のリストを並行優先順位キューに使おうとしたとする

- いつでも head のところに最小要素がある

□ 今までのところ挿入しかなかったので、先頭 N 個を取る pop を追加する

```
def pop(xs: TSortedList, n: Int): Unit = atomic {
    implicit txn =>
  var left = n
  while(left > 0) {
    xs.head() = xs.head().next()
    left -= 1
  }
}
val lst = new TSortedList
lst.insert(4).insert(9).insert(1).insert(16)
Future { pop(lst, 2) } foreach {
  case _ => log(s"removed 2 elements, list = $lst")
}
```

□ 取りすぎたら当然 NullPointerException

- onComplete で例外を補足してみると取り除いたはずのものは戻っている←ロールバックの効果

```
Future { pop(lst, 3) } onComplete {
  case Failure(t) => log(s"whoa $t: $lst")
}
```

□ 入れ子の外側でロールバックしたら内側も戻る

```
Future {
  atomic { implicit txn =>
    pop(lst, 1)
    sys.error("")
  }
} onComplete {
  case Failute(t) => log("$t, $lst")
}
```

□ ScalaSTM 以外の実装で、例外時に commit するものもある

- ScalaSTM では大半の例外では rollback するが…
- control 例外では commit する (break とかの実装に使うので)

```
import scala.util.control.Breaks._
Future {
  breakable {
    atomic { implicit txn =>
      for(n <- List(1, 2, 3)) {
        pop(lst, n)
        break
```

```
      }
    }
  }
  log(s"after remove - $lst")
}
```

□ さらに標準の動作を変更することもできる --- atomic の中で withControlFlowRecognizer を呼ぶ

- パラメタとして Throwable から boolean への部分関数を受け取る→これを使って個々の例外でどっちの扱いにするか決める

```
import scala.util.control._
Future {
  breakable {
    atomic.withControlFlowRecognizer {
      case c: ControlThrowable => false
    } { implicit txn =>
      for(n <- List(1, 2, 3)) {
        pop(lst, n)
        break
      }
    }
  }
  log(s"after remove - $lst")
}
```

□ トランザクション内で投げられた例外を catch で受け止めることもある

- この場合、入れ子トランザクションはアボートされて catch の所へいく

```
val lst = new TSortedList
lst.insert(4).insert(9).insert(1).insert(16)
atomic { implicit txn =>
  pop(lst, 2)
  log(s"lst = $lst")
  try { pop(lst, 3) }
  cat { case e: Exception => log(s"$e") }
  pop(lst, 1)
}
log(s"result - $lst")
```

□ これをやってみると、最初の log は 2 回実行される

- 例外が最初におきると外と中のトランザクションがロールバック (入れ子トランザクションの最適化のためフラット化している)
- 2 回目は想定通りに実行される

□ トランザクション内のトランザクションの意味が分かる例題

- 場合によっては空のとき例外を返すのでなく insert を待つとかも