Java并行(3):可见性重访之锁、Volatile与原子变量

1. 过期数据

我们在前面讨论JMM时便已提出“同步之道，外炼‘互斥’，内修‘可见’”的法门。过去，我不注意内存可见性的时候，程序里长满了小红疙瘩：

```
public class RedPimple extends Thread{
        private boolean done;
        private int value;

        @Override
        public void run() {
                while(!done) //A
                        Thread.yield();
                System.out.println(value); //D
        }

        public void done() {
                done = true;
        }

        public void setValue(int value){
                this.value = value;
        }

        public static void main(String[] args) {
                RedPimple r = new RedPimple();
                r.start();
                r.setValue(1); //B
                r.done(); //C
        }
}
```

上面的代码有什么问题？即使运行它千百遍，你可能也察觉不出有什么问题。但是，问题确实存在。

- 病灶一：r线程理论上存在无限循环的可能（这里有两个线程，main线程和r线程）。因为没有任何同步的措施，main线程中C动作的效果何时对于r线程的A可见是不可知的。他肯能在done值已经被改之后的一段

时间里仍然读到过期数据，最极端的情况，A一直读到的都是过期数据false。

- 病灶二：理论上可能打印出0。这就更匪夷所思了，main线程里不是有B hb C么？没错，是有B hb C，但是不管是B还是C，都和A、D没有hb关系，理论上存在这样的执行序列C A D B，它是合法的，C依旧可以宣称看到了B的效果。（这是从JMM理论上论证是允许的，实际情况取决于JMM掩盖之下的你的机器的MM）

这就是“过期数据”的隐患。尽管在这个例子里，问题好像还没那么严重，无非皮肤上出点小丘疹而已。但在实际的编程中，过期数据的危害是不容小视的。

2. 锁的可见性

后来，我用了“锁牌香皂”，小红疙瘩真的就不见了！看这里看这里……

```
public class NoRedPimple extends Thread {
        private boolean done;
        private int value;

@Override
public void run() {
        boolean tmp = false;
        while (!tmp) {
                synchronized (this) {
                        tmp = done; //A
                }

                Thread.yield();
        }

        synchronized (this) {
                System.out.println(value);  //D

        }
}

synchronized public void done() {
        done = true;
}

synchronized public void setValue(int value) {
```

```
                this.value = value;
        }

        public static void main(String[] args) {
                NoRedPimple r = new NoRedPimple();
                r.start();
                r.setValue(1); //B
                r.done(); //C
        }
}
```

正如这个例子所示，锁不仅仅有“互斥”的功能，而且还保证了内存可见性。运用我们在JMM里提到的理论，被标号的三个操作这次有了明确的HB关系，这下就不会有小红疙瘩了。

更进一步看锁的可见性：如果有两个线程t1，t2，t1有动作序列A B C U，其中U为放锁操作，t2有动作序列L D E F，L为加锁操作（同一锁），那么在执行中如果有U tb L，那么这两个线程的执行序列必为 A B C U L D E F，再无其他可能。这就保证了“在放锁前对t1可见的值，B获得锁后同样可见”。

3. Volatile

虽然上面用锁解决了过期数据的问题，但似乎有些大材小用了吧？代码不那么好看，治好了疹子，却留了一脸麻子。Java早已为消费者考虑到了这一点，volatile就是一种轻量级的同步，它可以保证“可见性”，但不保证“互斥”。

```
public class NoRedPimple extends Thread {
        private volatile boolean done;
        private volatile int value;

        @Override
        public void run() {
                while (!done) {//A
                        Thread.yield();
                }
                System.out.println(value); //D
        }

        public void done() {
                done = true;
```

```
        }

        public void setValue(int value) {
                this.value = value;
        }

        public static void main(String[] args) {
                NoRedPimple r = new NoRedPimple();
                r.start();
                r.setValue(1); //B
                r.done(); //C
        }
}
```

是不是简洁很多？结合JMM中对volatile读写的规定（SW第二条规则），上面的代码完全符合我们的要求，没有过期数据。那么，什么时候可以用volatile呢？必须同时满足以下三条：

- 不依赖自己：写变量时并不依赖变量的当前值，或者：可以保证只有一个writer
- 不依赖别人：变量不与其他状态共同组成invariant。
- 访问变量时，没有其他原因需要加锁。

只解释一下第一条，可能很多人对“不依赖自己的当前值”不太理解，举个简单的例子：count++， 这个就叫依赖当前值。为什么要有这样的限制？因为，volatile不保证count++是原子的，即我们所说的“互斥执行”，虽然我们过去的例子都把一条代码当作一个动作，但相信你知道，一条代码在CPU那里多半不会是一条指令，比如count++其实会分解为load-modify-store三个更小的动作，如果这样的操作有多个线程在做，是极易出错的。（鉴于这个问题过于经典，就此打住）。所以，第一条规则实际的意思就是“要么只有一个writer，怎么写随你便；要么可以多个writer，但不能是count++这种依赖当前值的写”。

4. 原子变量

Java 1.5 一声炮响，给我们送来了java.util.concurrency包，这个包并行功能强大，工具齐全，我们以后讨论会经常用到。原子变量也是此包提供的工具之一。顾名思义，原子变量，即支持“原子更新”，它更多地被用在“非阻塞算法”和“lock-free算法”中，其实我很想现在讨论非阻塞算法，两次面试都被

考到，但抬头看看标题已经写了“可见性重访”，还是不跑题了，以后有机会再和大家讨论这个topic。

而除了“原子更新”的好处外，原子变量还提供了与volatile相同的内存语义，所以volatile所能保证的可见性，在原子变量这里同样可以。

```
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;

public class NoRedPimple extends Thread {
        private AtomicBoolean done = new AtomicBoolean(false);
        private AtomicInteger value = new AtomicInteger(0);

        @Override
        public void run() {
                while (!done.get()) {//A
                        Thread.yield();
                }
                System.out.println(value.get()); //D
        }

        public void done() {
                done.set(true);
        }

        public void setValue(int value) {
                this.value.set(value);
        }

        public static void main(String[] args) {
                NoRedPimple r = new NoRedPimple();
                r.start();
                r.setValue(1); //B
                r.done(); //C
        }
}
```

当然，我们这里使用原子变量，也是“大材小用”了的，原子变量的NB之处在于其原子性的CAS（compare-and-set）操作，由此可以完成volatile所不能的“check-and-act”动作，“可见性”只不过是其稍带脚支持的功能而已。我们以后再讨论他的CAS、由他构建的“非阻塞算法”以及“非阻塞算法”和用一般锁构建的“阻塞算法”的比较。

主要参考资料：

1. JSL 第三版

2. Java Concurrency in Practice