二

62 volatile 的作用是什么？与 synchronized 有什么异同？

本课时我们主要介绍 volatile 的作用和适用场景，以及它与 synchronized 有什么异同。

volatile 是什么

首先我们就来介绍一下 volatile，它是 Java 中的一个关键字，是一种同步机制。当某个变量是共享变量，且这个变量是被 volatile 修饰的，那么在修改了这个变量的值之后，再读取该变量的值时，可以保证获取到的是修改后的最新的值，而不是过期的值。

相比于 synchronized 或者 Lock，volatile 是更轻量的，因为使用 volatile 不会发生上下文切换等开销很大的情况，不会让线程阻塞。但正是由于它的开销相对比较小，所以它的效果，也就是能力，相对也小一些。

虽然说 volatile 是用来保证线程安全的，但是它做不到像 synchronized 那样的同步保护，volatile 仅在很有限的场景中才能发挥作用，所以下面就让我们来看一下它的适用场景，我们会先给出不适合使用 volatile 的场景，再给出两种适合使用 volatile 的场景。

volatile 的适用场合

不适用：a++

首先我们就来看一下不适合使用 volatile 的场景，volatile 不适合运用于需要保证原子性的场景，比如更新的时候需要依赖原来的值，而最典型的就是 a++ 的场景，我们仅靠 volatile 是不能保证 a++ 的线程安全的。代码如下所示：

```
public class DontVolatile implements Runnable {

    volatile int a;

    AtomicInteger realA = new AtomicInteger();

    public static void main(String[] args) throws InterruptedException {
```

```
        Runnable r =  new DontVolatile();
        Thread thread1 = new Thread(r);
        Thread thread2 = new Thread(r);
        thread1.start();
        thread2.start();
        thread1.join();
        thread2.join();
        System.out.println(((DontVolatile) r).a);
        System.out.println(((DontVolatile) r).realA.get());
    }
    @Override
    public void run() {
        for (int i = 0; i < 1000; i++) {
            a++;
            realA.incrementAndGet();
        }
    }
}
```

在这段代码中，我们有一个 volatile 修饰的 int 类型的 a 变量，并且下面还有一个原子类的 realA，原子类是可以保证线程安全的，所以我们就用它来和 volatile int a 做对比，看一看它们实际效果上的差别。

在 main 函数中，我们新建了两个线程，并且让它们运行。这两个线程运行的内容就是去执行 1000 次的累加操作，每次累加操作会对 volatile 修饰的变量 a 进行自加操作，同时还会对原子类 realA 进行自加操作。当这两个线程都运行完毕之后，我们把结果给打印出来，其中一种运行结果如下：

```
1988
2000
```

会发现最终的 a 值和 realA 值分别为 1988 和 2000。可以看出，即便变量 a 被 volatile 修

饰了，即便它最终一共执行了 2000 次的自加操作（这一点可以由原子类的最终值来印证），但是依然有一些自加操作失效了，所以最终它的结果是不到 2000 的，这就证明了 volatile 不能保证原子性，那么它究竟适合运用于什么场景呢？

适用场合1：布尔标记位

如果某个共享变量自始至终只是被各个线程所赋值或读取，而没有其他的操作（比如读取并在此基础上进行修改这样的复合操作）的话，那么我们就可以使用 volatile 来代替 synchronized 或者代替原子类，因为赋值操作自身是具有原子性的，volatile 同时又保证了可见性，这就足以保证线程安全了。

一个比较典型的场景就是布尔标记位的场景，例如 volatile boolean flag。因为通常情况下，boolean 类型的标记位是会被直接赋值的，此时不会存在复合操作（如 a++），只存在单一操作，就是去改变 flag 的值，而一旦 flag 被 volatile 修饰之后，就可以保证可见性了，那么这个 flag 就可以当作一个标记位，此时它的值一旦发生变化，所有线程都可以立刻看到，所以这里就很适合运用 volatile 了。

我们来看一下代码示例：

```
public class YesVolatile1 implements Runnable {

    volatile boolean done = false;

    AtomicInteger realA = new AtomicInteger();

    public static void main(String[] args) throws InterruptedException {

        Runnable r =  new YesVolatile1();

        Thread thread1 = new Thread(r);

        Thread thread2 = new Thread(r);

        thread1.start();

        thread2.start();

        thread1.join();

        thread2.join();

        System.out.println(((YesVolatile1) r).done);

        System.out.println(((YesVolatile1) r).realA.get());

    }

    @Override
```

```
    public void run() {
        for (int i = 0; i < 1000; i++) {
            setDone();
            realA.incrementAndGet();
        }
    }
    private void setDone() {
        done = true;
    }
}
```

这段代码和前一段代码非常相似，唯一不同之处在于，我们把 volatile int a 改成了 volatile boolean done，并且在 1000 次循环的操作过程中调用的是 setDone() 方法，而这个 setDone() 方法就是把 done 这个变量设置为 true，而不是根据它原来的值再做判断，例如原来是 false，就设置成 true，或者原来是 true，就设置成 false，这些复杂的判断是没有的，setDone() 方法直接就把变量 done 的值设置为 true。那么这段代码最终运行的结果如下：

```
true
2000
```

无论运行多少次，控制台都会打印出 true 和 2000，打印出的 2000 已经印证出确实是执行了 2000 次操作，而最终的 true 结果证明了，在这种场景下，volatile 起到了保证线程安全的作用。

第二个例子区别于第一个例子最大的不同点就在于，第一个例子的操作是 a++，这是个复合操作，不具备原子性，而在本例中的操作仅仅是把 done 设置为 true，这样的赋值操作本身就是具备原子性的，所以在这个例子中，它是适合运用 volatile 的。

适用场合 2：作为触发器

那么下面我们再来看第二个适合用 volatile 的场景：作为触发器，保证其他变量的可见性。

下面是 Brian Goetz 提供的一个经典例子：

```
Map configOptions;

char[] configText;

volatile boolean initialized = false;

. . .

// In thread A

configOptions = new HashMap();

configText = readConfigFile(fileName);

processConfigOptions(configText, configOptions);

initialized = true;

. . .

// In thread B

while (!initialized)

  sleep();

// use configOptions
```

在这段代码中可以看到，我们有一个 map 叫作 configOptions，还有一个 char 数组叫作 configText，然后会有一个被 volatile 修饰的 boolean initialized，最开始等于 false。再下面的这四行代码是由线程 A 所执行的，它所做的事情就是初始化 configOptions，再初始化 configText，再把这两个值放到一个方法中去执行，实际上这些都代表了初始化的行为。那么一旦这些方法执行完毕之后，就代表初始化工作完成了，线程 A 就会把 initialized 这个变量设置为 true。

而对于线程 B 而言，它一开始会在 while 循环中反复执行 sleep 方法（例如休眠一段时间），直到 initialized 这个变量变成 true，线程 B 才会跳过 sleep 方法，继续往下执行。重点来了，一旦 initialized 变成了 true，此时对于线程 B 而言，它就会立刻使用这个 configOptions，所以这就要求此时的 configOptions 是初始化完毕的，且初始化的操作的结果必须对线程 B 可见，否则线程 B 在执行的时候就可能报错。

你可能会担心，因为这个 configOptions 是在线程 A 中修改的，那么在线程 B 中读取的时候，会不会发生可见性问题，会不会读取的不是初始化完毕后的值？如果我们不使用 volatile，那么确实是存在这个问题的。

但是现在我们用了被 volatile 修饰的 initialized 作为触发器，所以这个问题被解决了。根据 happens-before 关系的单线程规则，线程 A 中 configOptions 的初始化 happens-before 对

initialized 变量的写入，而线程 B 中对 initialzed 的读取 happens-before 对 configOptions 变量的使用，同时根据 happens-before 关系的 volatile 规则，线程 A 中对 initialized 的写入为 true 的操作 happens-before 线程 B 中随后对 initialized 变量的读取。

如果我们分别有操作 A 和操作 B，我们用 hb(A, B) 来表示 A happens-before B。而 Happens-before 是有可传递性质的，如果hb(A, B)，且hb(B, C)，那么可以推出hb(A, C)。所以根据上面的条件，我们可以得出结论：线程 A 中对于 configOptions 的初始化 happens-before 线程 B 中 对于 configOptions 的使用。所以对于线程 B 而言，既然它已经看到了 initialized 最新的值，那么它同样就能看到包括 configOptions 在内的这些变量初始化后的状态，所以此时线程 B 使用 configOptions 是线程安全的。这种用法就是把被 volatile 修饰的变量作为触发器来使用，保证其他变量的可见性，这种用法也是非常值得掌握的，可以作为面试时的亮点。

volatile 的作用

上面我们分析了两种非常典型的用法，那么就来总结一下 volatile 的作用，它一共有两层作用。

第一层的作用是保证可见性。Happens-before 关系中对于 volatile 是这样描述的：对一个 volatile 变量的写操作 happen-before 后面对该变量的读操作。

这就代表了如果变量被 volatile 修饰，那么每次修改之后，接下来在读取这个变量的时候一定能读取到该变量最新的值。

第二层的作用就是禁止重排序。先介绍一下 as-if-serial语义：不管怎么重排序，（单线程）程序的执行结果不会改变。在满足 as-if-serial 语义的前提下，由于编译器或 CPU 的优化，代码的实际执行顺序可能与我们编写的顺序是不同的，这在单线程的情况下是没问题的，但是一旦引入多线程，这种乱序就可能会导致严重的线程安全问题。用了 volatile 关键字就可以在一定程度上禁止这种重排序。

volatile 和 synchronized 的关系

下面我们就来看一下 volatile 和 synchronized 的关系：

相似性：volatile 可以看作是一个轻量版的 synchronized，比如一个共享变量如果自始至终只被各个线程赋值和读取，而没有其他操作的话，那么就可以用 volatile 来代替 synchronized 或者代替原子变量，足以保证线程安全。实际上，对 volatile 字段的每次读取或写入都类似于“半同步”——读取 volatile 与获取 synchronized 锁有相同的内存语义，而写入 volatile 与释放 synchronized 锁具有相同的语义。

不可代替：但是在更多的情况下，volatile 是不能代替 synchronized 的，volatile 并没有提供原子性和互斥性。

性能方面：volatile 属性的读写操作都是无锁的，正是因为无锁，所以不需要花费时间在获取锁和释放锁上，所以说它是高性能的，比 synchronized 性能更好。

小结

最后总结一下，本课时主要介绍了 volatile 是什么，以及它不适用的场景和两种非常典型的适用场景；然后我们介绍了 volatile 的两点作用，第一点是保证可见性，第二点是禁止重排序；最后我们分析了 volatile 和 synchronized 的关系。

上一页 下一页