二

42 AtomicInteger 和 synchronized 的异同点?

在上一课时中，我们说明了原子类和 synchronized 关键字都可以用来保证线程安全，在本课时中，我们首先分别用原子类和 synchronized 关键字来解决一个经典的线程安全问题，给出具体的代码对比，然后再分析它们背后的区别。

代码对比

首先，原始的线程不安全的情况的代码如下所示：

```
public class Lesson42 implements Runnable {

    static int value = 0;

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

        Runnable runnable = new Lesson42();

        Thread thread1 = new Thread(runnable);

        Thread thread2 = new Thread(runnable);

        thread1.start();

        thread2.start();

        thread1.join();

        thread2.join();

        System.out.println(value);

    }

    @Override

    public void run() {

        for (int i = 0; i < 10000; i++) {

            value++;

        }
```

```
    }
}
```

在代码中我们新建了一个 value 变量，并且在两个线程中对它进行同时的自加操作，每个线程加 10000 次，然后我们用 join 来确保它们都执行完毕，最后打印出最终的数值。

因为 value++ 不是一个原子操作，所以上面这段代码是线程不安全的（具体分析详见第 6 讲），所以代码的运行结果会小于 20000，例如会输出 14611 等各种数字。

我们首先给出**方法一**，也就是用原子类来解决这个问题，代码如下所示：

```
public class Lesson42Atomic implements Runnable {

    static AtomicInteger atomicInteger = new AtomicInteger();

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

        Runnable runnable = new Lesson42Atomic();

        Thread thread1 = new Thread(runnable);

        Thread thread2 = new Thread(runnable);

        thread1.start();

        thread2.start();

        thread1.join();

        thread2.join();

        System.out.println(atomicInteger.get());

    }

    @Override

    public void run() {

        for (int i = 0; i < 10000; i++) {

            atomicInteger.incrementAndGet();

        }

    }

}
```

用原子类之后，我们的计数变量就不再是一个普通的 int 变量了，而是 AtomicInteger 类型的对象，并且自加操作也变成了 incrementAndGet 法。由于原子类可以确保每一次的自加操作都是具备原子性的，所以这段程序是线程安全的，所以以上程序的运行结果会始终等于 20000。

下面我们给出**方法二**，我们用 synchronized 来解决这个问题，代码如下所示：

```
public class Lesson42Syn implements Runnable {

    static int value = 0;

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

        Runnable runnable = new Lesson42Syn();

        Thread thread1 = new Thread(runnable);

        Thread thread2 = new Thread(runnable);

        thread1.start();

        thread2.start();

        thread1.join();

        thread2.join();

        System.out.println(value);

    }

    @Override

    public void run() {

        for (int i = 0; i < 10000; i++) {

            synchronized (this) {

                value++;

            }

        }

    }

}
```

它与最开始的线程不安全的代码的区别在于，在 run 方法中加了 synchronized 代码块，就可以非常轻松地解决这个问题，由于 synchronized 可以保证代码块内部的原子性，所以以

上程序的运行结果也始终等于 20000，是线程安全的。

方案对比

下面我们就对这两种不同的方案进行分析。

第一点，我们来看一下它们背后**原理**的不同。

在第 21 课时中我们详细分析了 synchronized 背后的 monitor 锁，也就是 synchronized 原理，同步方法和同步代码块的背后原理会有少许差异，但总体思想是一致的：在执行同步代码之前，需要首先获取到 monitor 锁，执行完毕后，再释放锁。

而我们在第 39 课时中介绍了原子类，它保证线程安全的原理是利用了 CAS 操作。从这一点上看，虽然原子类和 synchronized 都能保证线程安全，但是其实现原理是大有不同的。

第二点不同是**使用范围**的不同。

对于原子类而言，它的使用范围是比较局限的。因为一个原子类仅仅是一个对象，不够灵活。而 synchronized 的使用范围要广泛得多。比如说 synchronized 既可以修饰一个方法，又可以修饰一段代码，相当于可以根据我们的需要，非常灵活地去控制它的应用范围。

所以仅有少量的场景，例如计数器等场景，我们可以使用原子类。而在其他更多的场景下，如果原子类不适用，那么我们就可以考虑用 synchronized 来解决这个问题。

第三个区别是**粒度**的区别。

原子变量的粒度是比较小的，它可以把竞争范围缩小到变量级别。通常情况下，synchronized 锁的粒度都要大于原子变量的粒度。如果我们只把一行代码用 synchronized 给保护起来的话，有一点杀鸡焉用牛刀的感觉。

第四点是它们**性能**的区别，同时也是悲观锁和乐观锁的区别。

因为 synchronized 是一种典型的悲观锁，而原子类恰恰相反，它利用的是乐观锁。所以，我们在比较 synchronized 和 AtomicInteger 的时候，其实也就相当于比较了悲观锁和乐观锁的区别。

从性能上来考虑的话，悲观锁的操作相对来讲是比较重量级的。因为 synchronized 在竞争激烈的情况下，会让拿不到锁的线程阻塞，而原子类是永远不会让线程阻塞的。不过，虽然 synchronized 会让线程阻塞，但是这并不代表它的性能就比原子类差。

因为悲观锁的开销是固定的，也是一劳永逸的。随着时间的增加，这种开销并不会线性增

长。

而乐观锁虽然在短期内的开销不大，但是随着时间的增加，它的开销也是逐步上涨的。

所以从性能的角度考虑，它们没有一个孰优孰劣的关系，而是要区分具体的使用场景。在竞争非常激烈的情况下，推荐使用 synchronized；而在竞争不激烈的情况下，使用原子类会得到更好的效果。

值得注意的是，synchronized 的性能随着 JDK 的升级，也得到了不断的优化。synchronized 会从无锁升级到偏向锁，再升级到轻量级锁，最后才会升级到让线程阻塞的重量级锁。因此synchronized 在竞争不激烈的情况下，性能也是不错的，不需要“谈虎色变”。

上一页 下一页