39 原子类是如何利用 CAS 保证线程安全的?

本课时主要讲解原子类是如何利用 CAS 保证线程安全的。

什么是原子类？原子类有什么作用?

要想回答这个问题，首先我们需要知道什么是原子类，以及它有什么作用。

在编程领域里，原子性意味着“一组操作要么全都操作成功，要么全都失败，不能只操作成功其中的一部分”。而 java.util.concurrent.atomic 下的类，就是具有原子性的类，可以原子性地执行添加、递增、递减等操作。比如之前多线程下的线程不安全的 i++ 问题，到了原子类这里，就可以用功能相同且线程安全的 getAndIncrement 方法来优雅地解决。

原子类的作用和锁有类似之处，是为了保证并发情况下线程安全。不过原子类相比于锁，有一定的优势：

- 粒度更细：原子变量可以把竞争范围缩小到变量级别，通常情况下，锁的粒度都要大于原子变量的粒度。
- 效率更高：除了高度竞争的情况之外，使用原子类的效率通常会比使用同步互斥锁的效率更高，因为原子类底层利用了 CAS 操作，不会阻塞线程。

6 类原子类纵览

下面我们来看下一共有哪些原子类，原子类一共可以分为以下这 6 类，我们来逐一介绍：

类型	具体类
Atomic* 基本类型原子类	AtomicInteger、AtomicLong、AtomicBoolean
Atomic*Array 数组类型原子类	AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

类型	具体类
Atomic*Reference 引用类型原子类	AtomicReference、AtomicStampedReference、AtomicMarkableReference
Atomic*FieldUpdater 升级类型原子类	AtomicIntegerfieldupdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater
Adder 累加器	LongAdder、DoubleAdder
Accumulator 积累器	LongAccumulator、DoubleAccumulator

Atomic\ 基本类型原子类

首先看到第一类 Atomic*，我们把它称为基本类型原子类，它包括三种，分别是 AtomicInteger、AtomicLong 和 AtomicBoolean。

我们来介绍一下最为典型的 AtomicInteger。对于这个类型而言，它是对于 int 类型的封装，并且提供了原子性的访问和更新。也就是说，我们如果需要一个整型的变量，并且这个变量会被运用在并发场景之下，我们可以不用基本类型 int，也不使用包装类型 Integer，而是直接使用 AtomicInteger，这样一来就自动具备了原子能力，使用起来非常方便。

AtomicInteger 类常用方法

AtomicInteger 类有以下几个常用的方法：

- public final int get() //获取当前的值

因为它本身是一个 Java 类，而不再是一个基本类型，所以要想获取值还是需要一些方法，比如通过 get 方法就可以获取到当前的值。

- public final int getAndSet(int newValue) //获取当前的值，并设置新的值

接下来的几个方法和它平时的操作相关：

- public final int getAndIncrement() //获取当前的值，并自增
- public final int getAndDecrement() //获取当前的值，并自减

- public final int getAndAdd(int delta) //获取当前的值，并加上预期的值

这个参数就是我想让当前这个原子类改变多少值，可以是正数也可以是负数，如果是正数就是增加，如果是负数就是减少。而刚才的 getAndIncrement 和 getAndDecrement 修改的数值默认为 +1 或 -1，如果不能满足需求，我们就可以使用 getAndAdd 方法来直接一次性地加减我们想要的数值。

- boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将该值更新为输入值（update）

这个方法也是 CAS 的一个重要体现。

Array 数组类型原子类

下面我们来看第二大类 Atomic*Array 数组类型原子类，数组里的元素，都可以保证其原子性，比如 AtomicIntegerArray 相当于把 AtomicInteger 聚合起来，组合成一个数组。这样一来，我们如果想用一个每一个元素都具备原子性的数组的话， 就可以使用 Atomic*Array。

它一共分为 3 种，分别是：

- AtomicIntegerArray：整形数组原子类；
- AtomicLongArray：长整形数组原子类；
- AtomicReferenceArray ： 引用类型数组原子类。

Atomic\Reference 引用类型原子类

下面我们介绍第三种 AtomicReference 引用类型原子类。AtomicReference 类的作用和 AtomicInteger 并没有本质区别， AtomicInteger 可以让一个整数保证原子性，而 AtomicReference 可以让一个对象保证原子性。这样一来，AtomicReference 的能力明显比 AtomicInteger 强，因为一个对象里可以包含很多属性。

在这个类别之下，除了 AtomicReference 之外，还有：

- AtomicStampedReference：它是对 AtomicReference 的升级，在此基础上还加了时间戳，用于解决 CAS 的 ABA 问题。
- AtomicMarkableReference：和 AtomicReference 类似，多了一个绑定的布尔值，可以用于表示该对象已删除等场景。

Atomic\FieldUpdater 原子更新器

第四类我们将要介绍的是 Atomic\FieldUpdater，我们把它称为原子更新器，一共有三种，分别是。

- AtomicIntegerFieldUpdater：原子更新整形的更新器；
- AtomicLongFieldUpdater：原子更新长整形的更新器；
- AtomicReferenceFieldUpdater：原子更新引用的更新器。

如果我们之前已经有了一个变量，比如是整型的 int，实际它并不具备原子性。可是木已成舟，这个变量已经被定义好了，此时我们有没有办法可以让它拥有原子性呢？办法是有的，就是利用 Atomic*FieldUpdater，如果它是整型的，就使用 AtomicIntegerFieldUpdater 把已经声明的变量进行升级，这样一来这个变量就拥有了 CAS 操作的能力。

这里的非互斥同步手段，是把我们已经声明好的变量进行 CAS 操作以达到同步的目的。那么你可能会想，既然想让这个变量具备原子性，为什么不在一开始就声明为 AtomicInteger？这样也免去了升级的过程，难道是一开始设计的时候不合理吗？这里有以下几种情况：

第一种情况是出于历史原因考虑，那么如果出于历史原因的话，之前这个变量已经被声明过了而且被广泛运用，那么修改它成本很高，所以我们可以利用升级的原子类。

另外还有一个使用场景，如果我们在大部分情况下并不需要使用到它的原子性，只在少数情况，比如每天只有定时一两次需要原子操作的话，我们其实没有必要把原来的变量声明为原子类型的变量，因为 AtomicInteger 比普通的变量更加耗费资源。所以如果我们有成千上万个原子类的实例的话，它占用的内存也会远比我们成千上万个普通类型占用的内存高。所以在这种情况下，我们可以利用 AtomicIntegerFieldUpdater 进行合理升级，节约内存。

下面我们看一段代码：

```
public class AtomicIntegerFieldUpdaterDemo implements Runnable{

    static Score math;

    static Score computer;

    public static AtomicIntegerFieldUpdater<Score> scoreUpdater = AtomicIntegerField

            .newUpdater(Score.class, "score");

    @Override

    public void run() {

        for (int i = 0; i < 1000; i++) {

            computer.score++;
```

```
            scoreUpdater.getAndIncrement(math);
        }
    }
    public static class Score {
        volatile int score;
    }
    public static void main(String[] args) throws InterruptedException {
        math =new Score();
        computer =new Score();
        AtomicIntegerFieldUpdaterDemo2 r = new AtomicIntegerFieldUpdaterDemo2();
        Thread t1 = new Thread(r);
        Thread t2 = new Thread(r);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("普通变量的结果："+ computer.score);
        System.out.println("升级后的结果："+ math.score);
    }
}
```

这段代码就演示了这个类的用法，比如说我们有两个类，它们都是 Score 类型的，Score 类型内部会有一个分数，也叫作 core，那么这两个分数的实例分别叫作数学 math 和计算机 computer，然后我们还声明了一个 AtomicIntegerFieldUpdater，在它构造的时候传入了两个参数，第一个是 Score.class，这是我们的类名，第二个是属性名，叫作 score。

接下来我们看一下 run 方法，run 方法里面会对这两个实例分别进行自加操作。

第一个是 computer，这里的 computer 我们调用的是它内部的 score，也就是说我们直接调用了 int 变量的自加操作，这在多线程下是线程非安全的。

第二个自加是利用了刚才声明的 scoreUpdater 并且使用了它的 getAndIncrement 方法并且传入了 math，这是一种正确使用AtomicIntegerFieldUpdater 的用法，这样可以线程安全地进行自加操作。

接下来我们看下 main 函数。在 main 函数中，我们首先把 math 和 computer 定义了出来，然后分别启动了两个线程，每个线程都去执行我们刚才所介绍过的 run 方法。这样一来，两个 score，也就是 math 和 computer 都会分别被加 2000 次，最后我们在 join 等待之后把结果打印了出来，这个程序的运行结果如下：

普通变量的结果：1942 升级后的结果：2000

可以看出，正如我们所预料的那样，普通变量由于不具备线程安全性，所以在多线程操作的情况下，它虽然看似进行了 2000 次操作，但有一些操作被冲突抵消了，所以最终结果小于 2000。可是使用 AtomicIntegerFieldUpdater 这个工具之后，就可以做到把一个普通类型的 score 变量进行原子的自加操作，最后的结果也和加的次数是一样的，也就是 2000。可以看出，这个类的功能还是非常强大的。

下面我们继续看最后两种原子类。

Adder 加法器

它里面有两种加法器，分别叫作 LongAdder 和 DoubleAdder。

Accumulator 积累器

最后一种叫 Accumulator 积累器，分别是 LongAccumulator 和 DoubleAccumulator。

这两种原子类我们会在后面的课时中展开介绍。

以 AtomicInteger 为例，分析在 Java 中如何利用 CAS 实现原子操作?

让我们回到标题中的问题，在充分了解了原子类的作用和种类之后，我们来看下 AtomicInteger 是如何通过 CAS 操作实现并发下的累加操作的，以其中一个重要方法 getAndAdd 方法为突破口。

getAndAdd方法

这个方法的代码在 Java 1.8 中的实现如下：

```
//JDK 1.8实现
```

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}
```

可以看出，里面使用了 Unsafe 这个类，并且调用了 unsafe.getAndAddInt 方法。所以这里需要简要介绍一下 Unsafe 类。

Unsafe 类

Unsafe 其实是 CAS 的核心类。由于 Java 无法直接访问底层操作系统，而是需要通过 native 方法来实现。不过尽管如此，JVM 还是留了一个后门，在 JDK 中有一个 Unsafe 类，它提供了硬件级别的原子操作，我们可以利用它直接操作内存数据。

那么我们就来看一下 AtomicInteger 的一些重要代码，如下所示：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;
    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
    private volatile int value;
    public final int get() {return value;}
    ...
}
```

可以看出，在数据定义的部分，首先还获取了 Unsafe 实例，并且定义了 valueOffset。我们往下看到 static 代码块，这个代码块会在类加载的时候执行，执行时我们会调用 Unsafe

的 objectFieldOffset 方法，从而得到当前这个原子类的 value 的偏移量，并且赋给 valueOffset 变量，这样一来我们就获取到了 value 的偏移量，它的含义是在内存中的偏移地址，因为 Unsafe 就是根据内存偏移地址获取数据的原值的，这样我们就能通过 Unsafe 来实现 CAS 了。

value 是用 volatile 修饰的，它就是我们原子类存储的值的变量，由于它被 volatile 修饰，我们就可以保证在多线程之间看到的 value 是同一份，保证了可见性。

接下来继续看 Unsafe 的 getAndAddInt 方法的实现，代码如下：

```
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
    return var5;
}
```

首先我们看一下结构，它是一个 do-while 循环，所以这是一个死循环，直到满足循环的退出条件时才可以退出。

那么我们来看一下 do 后面的这一行代码 var5 = this.getIntVolatile(var1, var2) 是什么意思。这是个 native 方法，作用就是获取在 var1 中的 var2 偏移处的值。

那传入的是什么呢？传入的两个参数，第一个就是当前原子类，第二个是我们最开始获取到的 offset，这样一来我们就可以获取到当前内存中偏移量的值，并且保存到 var5 里面。此时 var5 实际上代表当前时刻下的原子类的数值。

现在再来看 while 的退出条件，也就是 compareAndSwapInt 这个方法，它一共传入了 4 个参数，这 4 个参数是 var1、var2、var5、var5 + var4，为了方便理解，我们给它们取了新了变量名，分别 object、offset、expectedValue、newValue，具体含义如下：

- 第一个参数 object 就是将要操作的对象，传入的是 this，也就是 atomicInteger 这个对象本身；
- 第二个参数是 offset，也就是偏移量，借助它就可以获取到 value 的数值；
- 第三个参数 expectedValue，代表“期望值”，传入的是刚才获取到的 var5；
- 而最后一个参数 newValue 是希望修改的数值 ， 等于之前取到的数值 var5 再加上

> var4，而 var4 就是我们之前所传入的 delta，delta 就是我们希望原子类所改变的数值，比如可以传入 +1，也可以传入 -1。

所以 compareAndSwapInt 方法的作用就是，判断如果现在原子类里 value 的值和之前获取到的 var5 相等的话，那么就把计算出来的 var5 + var4 给更新上去，所以说这行代码就实现了 CAS 的过程。

一旦 CAS 操作成功，就会退出这个 while 循环，但是也有可能操作失败。如果操作失败就意味着在获取到 var5 之后，并且在 CAS 操作之前，value 的数值已经发生变化了，证明有其他线程修改过这个变量。

这样一来，就会再次执行循环体里面的代码，重新获取 var5 的值，也就是获取最新的原子变量的数值，并且再次利用 CAS 去尝试更新，直到更新成功为止，所以这是一个死循环。

我们总结一下，Unsafe 的 getAndAddInt 方法是通过循环 + CAS 的方式来实现的，在此过程中，它会通过 compareAndSwapInt 方法来尝试更新 value 的值，如果更新失败就重新获取，然后再次尝试更新，直到更新成功。

总结

在本课时我们首先介绍了原子类的作用，然后对 6 类原子类进行了介绍，分别是 Atomic* 基本类型原子类、Atomic*Array 数组类型原子类*、*Atomic*Reference 引用类型原子类、Atomic*FieldUpdater 升级类型原子类、Adder 加法器和 Accumulator 积累器。

然后我们对它们逐一进行了展开介绍，了解了它们的基本作用和用法，接下来我们以 AtomicInteger 为例，分析了在 Java 中是如何利用 CAS 实现原子操作的。

我们从 getAndAdd 方法出发，逐步深入，最后到了 Unsafe 的 getAndAddInt 方法。所以通过源码分析之后，我们也清楚地看到了，它实现的原理是利用自旋去不停地尝试，直到成功为止。

上一页 下一页