Java 多线程开发之原子变量与 CAS 算法（二）

一、前言

在上篇[《Java 多线程开发之 volatile（一）》文章中介绍了 volatile 的相关内容，它是一个轻量级的锁，但不支持原子操作。

本篇将介绍原子操作相关内容。

二、基本概念

2.1 CAS 算法

CAS (Compare-And-Swap) 是一种硬件对并发的支持，针对多处理器操作而设计的处理器中的一种特殊指令，用于管理对共享数据的并发访问。

CAS 是一种无锁的非阻塞算法的实现，该算法包含了 3 个操作数

```
需要读写的内存值 V
进行比较的值 A
拟写入的新值 B
```

当且仅当 V 的值等于 A 时， CAS 通过原子方式用新值 B 来更新 V 的值，否则不会执行任何操作。

三、演示与分析

3.1 案例演示

```
public class CASTest {

    public static void main(String[] args) {

        DemoRunnable dr = new DemoRunnable();

        for (int i = 0; i < 10; i++) {
            new Thread(dr).start();
        }
    }
}

class DemoRunnable implements Runnable {

    private volatile int count = 0;

    @Override
    public void run() {
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("子线程-count:" + (++count));
    }

}
```

打印结果：

```
子线程-count:1
子线程-count:1
子线程-count:2
子线程-count:5
子线程-count:7
子线程-count:6
子线程-count:8
子线程-count:4
子线程-count:4
子线程-count:3
```

从结果我们可以发现 count 的值没有加到 10，且出现多个线程累加 count 值重复的问题，即出现了线程安全问题。

3.2 问题分析

其实，在 Java 语言中执行 ++ 操作实际上在底层被拆分为三个步骤，即“读-改-写”。

笔者在初学 Java 时遇到如下一个问题：

```
int i = 10;
i = i++;
System.out.println(i);
```

打印的结果依然为 10。

因为 ***i = i++*** 在底层操作如下：

```
int temp = i; // 读
i = i + 1;    // 改
i = temp;     // 写
```

同理，在上文的代码中，***System.out.println("子线程-count:" + (++count)); ***同样被拆分为多个步骤，无法保证操作的原子性，从而当多个线程修改 count 时出现对旧值重复累加操作，进而出现打印相同结果的问题。

四、解决方案

在 java.util.concurrent.atomic 包中，提供了很多支持原子操作的类。如：AtomicBoolean、AtomicInteger、AtomicLong、AtomicReference 等。

这些类底层通过 CAS 算法实现。

修改 DemoRunnable：

```
class DemoRunnable implements Runnable {

    private volatile AtomicInteger ai = new AtomicInteger();

    @Override
    public void run() {

        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("子线程-count:" + (ai.incrementAndGet()));
    }

}
```

将 int count 改成 AtomicInteger ai。

代码执行 N 次都能正常将 ai 的值累加成 10 。

五、模拟 **CAS** 算法

现在，根据上文介绍的 CAS 算法步骤来模拟 CAS 算法的实现。

```
public class CompareAndSwapTest {

    public static void main(String[] args) {

        CompareAndSwap cas = new CompareAndSwap();

        for (int i = 0; i < 10; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    int expectedValue = cas.get();
                    boolean result = cas.compareAndSet(expectedValue,(int)
(Math.random() * 100));
                    System.out.println("结果：" + result);
                }
            }).start();
        }
    }
}

class CompareAndSwap {

    private int value;

    /**
     * 获取内存值
     *
     * @return
     */
    public synchronized int get() {
        return value;
    }

    /**
     * 设置
     *
     * @param expectedValue 预估值
     * @param newValue      新值
     *
     * @return
```

```
     */
    public synchronized boolean compareAndSet(int expectedValue, int
newValue) {
        return expectedValue == compareAndSwap(expectedValue, newValue);
    }

    /**
     * 比较
     *
     * @param expectedValue 预估值
     * @param newValue      新值
     *
     * @return
     */
    private synchronized int compareAndSwap(int expectedValue, int
newValue) {
        int oldValue = value;

        if (oldValue == expectedValue) {
            value = newValue;
        }

        return oldValue;
    }
}
```

注意：此处模拟代码使用 **synchronized** 关键字只是模拟演示效果，并非 **Java** 底层具体实现方式。