链滴

Atomic 原子类的使用及其原理

作者：vcjmhg
原文链接：https://ld246.com/article/1627910297195
来源网站：链滴

概述

Atomic，在化学中原子指的是不可分割的实体。同样的在并发体系中，原子类则是所有操作都具有子性的，也就是说它的一个操作一旦开始，就不会被其他线程干扰。同时原子类是"并java并发体系"，无锁方案的重要组成部分。

在之前的文章中--"并发知识梳理"，这篇文章中我们提出了累加器问题：

当多个线程同时访问下边这个累加方法时，会出现最终结果小于实际累加值的情况，并且每次执行的候最终结果都是不确定的。

```
public class UnsafeSequence {
    private int value = 0;
    public int getNext() {
        return ++value;
    }
}
```

当时分析后，我们知道造成这种不确定性的原因在于我们这个累加器的加法操作++value不具有原子导致的，如果我们的加法操作使用原子操作则可以避免这个问题。

使用

在下面的代码中，我们将原来的 long 型变量 count 替换为了原子类 AtomicLong，原来的 count + 1 替换成了 count.getAndIncrement()，仅需要这两处简单的改动就能使 add10K() 方法变成线程安的，原子类的使用还是挺简单的。

```
public class SafeSequence{
  AtomicLong count = new AtomicLong(0);
  public long getNext() {
     return count.getAndIncrement();
  }
}
```

question 好像很神奇，但它内部是如何保证 getAndIncrement()是原子操作的呢?

CAS原理

其实其内部原理非常简单，都是通过CAS（Compare And Sweep，比较并交换）指令来实现。

CAS 的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。可能说起来比较抽象，们结合getAndIncrement()源码来看：

```
public final long getAndIncrement() {
    return U.getAndAddLong(this, VALUE, 1L);
}
```

U.getAndAddLong() 方法的源码如下，该方法首先会在内存中读取共享变量的值，之后循环调用compareAndSwapLong()方法来尝试设置共享变量的值，直到成功为止。compareAndSwapLong() 是个 native 方法，只有当内存中共享变量的值等于 expected 时，才会将共享变量的值更新为 x，并且回 true；否则返回 fasle。compareAndSwapLong 的语义和 CAS 指令的语义的差别仅仅是返回值同而已。

```
public final long getAndAddLong(Object o, long offset, long delta) {
    long v;
    do {
        v = getLongVolatile(o, offset);
    } while (!weakCompareAndSetLong(o, offset, v, v + delta));
    return v;
}

//当从内存中读取的o.offset的值和expected值相同时，将o.offset的值更新为x
native boolean compareAndSwapLong(  Object o, long offset, long expected,  long x);
```

使用 CAS 来解决并发问题，一般都会伴随着自旋，而所谓自旋，其实就是循环尝试。上边的代码也例外，

```
do {
    读取旧变量的值
} while (compareAndSet(expect, newV);
```

在自旋过程中，如果CAS设置失败，会重新读取变量的值，再次进行CAS操作，直到成功。

通过CAS+自旋可以实现以无锁的方式更新变量的值，但可能带了一个ABA问题，什么是ABA问题呢?

简单来说，第一个线程取到了变量 x 的值 A，然后巴拉巴拉干别的事，总之就是只拿到了变量 x 的值。这段时间内第二个线程也取到了变量 x 的值 A，然后把变量 x 的值改为 B，然后巴拉巴拉干别的事最后又把变量 x 的值变为 A （相当于还原了）。在这之后第一个线程终于进行了变量 x 的操作，但此时变量 x 的值还是 A，所以 compareAndSet 操作是成功。

可能大多数情况下我们并不关心 ABA 问题，例如数值的原子递增，但也不能所有情况下都不关心，如原子化的更新对象很可能就需要关心 ABA 问题，因为两个 A 虽然相等，但是第二个 A 的属性可已经发生变化了。所以在使用 CAS 方案的时候，一定要先 check 一下。

Java中常用的原子类

基本类型原子类

- AtomicInteger：整型原子类
- AtomicLong：长整型原子类
- AtomicBoolean ： 布尔型原子类

上面三个类提供的方法几乎相同，所以我们这里以 AtomicInteger 为例子来介绍。

```
public final int get() //获取当前的值
public final int getAndSet(int newValue)//获取当前的值，并设置新的值
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
boolean compareAndSet(int expect, int update) //如果输入的数值等于预期值，则以原子方式将值设置为输入值（update）
public final void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其线程在之后的一小段时间内还是可以读到旧的值。
```

数组类型原子类

- 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

这些类提供的方法和原子化的基本数据类型的区别仅仅是：每个方法多了一个数组的索引参数，所以里也不再赘述了。其常用方法如下：

public final int get(int i) //获取 index=i 位置元素的值
public final int getAndSet(int i, int newValue)//返回 index=i 位置的当前的值，并将其设置为新：newValue
public final int getAndIncrement(int i)//获取 index=i 位置元素的值，并让该位置的元素自增
public final int getAndDecrement(int i) //获取 index=i 位置元素的值，并让该位置的元素自减
public final int getAndAdd(int i, int delta) //获取 index=i 位置元素的值，并加上预期的值
boolean compareAndSet(int i, int expect, int update) //如果输入的数值等于预期值，则以原子方将 index=i 位置的元素值设置为输入值（update）
public final void lazySet(int i, int newValue)//最终 将index=i 位置的元素设置为newValue,使用 laySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

引用类型原子类

- AtomicReference：引用类型原子类
- AtomicStampedReference：原子更新带有版本号的引用类型。该类将整数值与引用关联起来，用于解决原子的更新数据和数据的版本号，可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。
- AtomicMarkableReference ：原子更新带有标记的引用类型。该类将 boolean 标记与引用关联来，本质就是它的版本号只有两个，true和false，这个类只能减少ABA问题的发生，但不能从根本上决ABA问题。

前面引用类型原子类可能出现ABA问题，如何解决该问题呢?

其实也很简单，只要我们在更新版本的时候增加一个版本号就可以解决，基于此设计了AtomicStampdReference

也就是说在使用boolean compareAndSet(V expectedReference，V newReference，int expecteStamp)增加了一个版本字段字段newStam变成了boolean compareAndSet(V expectedReference,V newReference，int expectedStamp,int newStam)。

总结

本文总结了常用的原子类，及其内部实现的原理--CAS原理。Java 提供的原子类能够解决一些简单的子性问题，但你可能会发现，上面我们所有原子类的方法都是针对一个共享变量的，如果你需要解决个变量的原子性问题，建议还是使用互斥锁方案。原子类虽好，但使用要慎之又慎。

引用

1. 《 原子类：无锁工具类的典范》
2. 《Atomic原子类总结》