链滴

CAS 与 Auomic 原子类

作者：DongXiaokai0819

原文链接：https://ld246.com/article/1571234161860

来源网站：链滴

<h3 id="toc_h3_0">CAS与Auomic原子类</h3>
<h4 id="toc_h4_1">什么是原子操作？</h4>
<blockquote>
"原子操作(atomic operation)是不需要synchronized"，这是多线程编程的老生常谈了。所谓原操作是指不会被线程调度机制打断的操作；这种操作一旦开始，就一直运行到结束，中间不会有任何ontext switch （切换到另一个线程）。
</blockquote>
<h4 id="toc_h4_2">Java如何实现原子操作？</h4>

用锁，synchronized内置锁、显示锁
CAS 实现

<h4 id="toc_h4_3">synchronized的缺点</h4>

在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，非常消耗资源，同时也能带来死锁或者其它安全问题。
一个线程持有的锁很长时间不释放。
如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，优先级高的反而不得到执行，产生性能问题。

<h4 id="toc_h4_4">什么是CAS？</h4>
<blockquote>
在计算机科学中，比较和交换（Conmpare And Swap）是用于实现多线程同步的原子指令。将内存位置的内容与给定值进行比较，只有在相同的情况下，将该内存位置的内容修改为新的给定值。这是作为单个原子操作完成的。 原子性保证新值基于最新信息计算; 如果该值在同一时间被另一个线更新，则写入将失败。 操作结果必须说明是否进行替换; 这可以通过一个简单的布尔响应（这个变体常称为比较和设置），或通过返回从内存位置读取的值来完成。
</blockquote>
一个CAS涉及到以下操作：
<blockquote>
我们假设内存中的原数据V，旧的预期值A，需要修改的新值B。

比较 A 与 V 是否相等。（比较）
如果比较相等，将 B 写入 V。（交换）
返回操作是否成功。

</blockquote>
<h5 id="toc_h5_5">CAS 自旋</h5>
基于CAS的自旋就是典型的乐观锁，程序执行时，某个线程从共享内存中取值V并建一个副本A对A进行计算后将新的值保存为B，然后对A值和内存中的V值进行比较，如果A等于V，则认为内存中V值没有被其他线程修改过，可以将新值B赋给内存中的V，否则，认为内存中已被其他的线程修改，重新执行上述计算步骤，即！！！ 先从内存中取得V的副本A，对A进行计算完成后将结果保存为B，将A与内存中的V进行比较，如果不相同则继续循环跳到 ！！！ 处，直到线程本地副本A值等于内存V为止。
<h5 id="toc_h5_6">CAS的缺点</h5>

ABA问题。因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化但是实际上却变化了。ABA问题的解决思路就是使用版本号。Java现在提供了带有版本号的Atomic。
循环时间长开销大。自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。
只能保证一个共享变量的原子操作。

<h4 id="toc_h4_7">Java中采用CAS原理实现的原子类有哪些？</h4>

<h5 id="toc_h5_8">基本类型</h5>
使用原子的方式更新基本类型

AtomicInteger：整形原子类
AtomicLong：长整型原子类
AtomicBoolean：布尔型原子类

<h5 id="toc_h5_9">数组类型</h5>
使用原子的方式更新数组里的某个元素

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

<h5 id="toc_h5_10">引用类型</h5>

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可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。
AtomicMarkableReference ：原子更新带有标记位的引用类型，传入Boolean值，表示是否改过，为了解决可能出现的ABA问题

<h5 id="toc_h5_11">字段类型</h5>

AtomicIntegerFieldUpdater：原子更新整形字段的更新器
AtomicLongFieldUpdater：原子更新长整形字段的更新器
AtomicReferenceFieldUpdater： 原子更新引用类型字段的更新器。

<h4 id="toc_h4_12">AtomicInteger</h4>
<h5 id="toc_h5_13">AtomicInteger的常用方法</h5>

int get() //获取当前的值
int getAndSet(int newValue)//获取当前的值，并设置新的值
int getAndIncrement()//获取当前的值，并自增，i++
int getAndDecrement() //获取当前的值，并自减, i--
int incrementAndGet() //先自增，然后获取自增后的值, ++i
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，则以原子方将该值设置为输入值（update）
void lazySet(int newValue)//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程在后的一小段时间内还是可以读到旧的值。

<h5 id="toc_h5_14">使用实例</h5>
<pre><code>   static AtomicInteger atomicInteger = new AtomicInteger(10);

public static void main(String[] args) {
    System.out.println(atomicInteger.getAndIncrement());//先获取值，再自增
    System.out.println(atomicInteger.incrementAndGet());//先自增，再获取值
    System.out.println(atomicInteger.get());
}

</code></pre>

大家可以自己测试一下上述程序的输出。
下面我们多启动几个线程，来测试一下Atomic类的原子性。
<pre><code>   static AtomicInteger atomicInteger = new AtomicInteger(10);

private static class TestAtomicInteger extends Thread{
    private AtomicInteger atomicInteger;
    TestAtomicInteger(AtomicInteger atomicInteger){
        this.atomicInteger = atomicInteger;
    }
    @Override
    public void run() {
        Random random = new Random();
        try {
            Thread.sleep(random.nextInt(1000));//让每个线程休眠不同的时间
            int andIncrement = atomicInteger.getAndIncrement();
            System.out.println(Thread.currentThread().getId() +"号线程的值为：" + andIncrement);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}

public static void main(String[] args) {
    for (int i=0; i<30; i++){//启动三十个线程
        new TestAtomicInteger(atomicInteger).start();
    }
}

</code></pre>

我们启动了三十个线程，并让不同的线程休眠了不同的时间，我们来看一下输出。
<pre><code>33号线程的值为：11//第一次运行的时候出现了这种情况。
26号线程的值为：10//只能说这个33号线程太争气了。。。
15号线程的值为：12
25号线程的值为：13
13号线程的值为：14
21号线程的值为：15
37号线程的值为：16
18号线程的值为：17
24号线程的值为：18
30号线程的值为：19
16号线程的值为：20
31号线程的值为：21
39号线程的值为：22
22号线程的值为：23
28号线程的值为：24
....
</code></pre>
我们可以看到，并没有产生任何的数据冲突。
<h5 id="toc_h5_15">AtomicInteger原理</h5>
AtomicInteger类的部分源码
<pre><code>   // 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;

</code></pre>

<blockquote>
<p>AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操
，从而避免 synchronized 的高开销，执行效率大为提升。<br>
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。UnSafe 类的 objectFiel
Offset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址，返回值是 valueOffse
。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该
量的最新值。</p>
</blockquote>
<h4 id="toc_h4_16">其它的Atomic类</h4>
<h5 id="toc_h5_17">AtomicArray的简单使用</h5>
<pre><code>    static int[] value = new int[] { 1, 2 };
    static AtomicIntegerArray atomicIntegerArray= new AtomicIntegerArray(value);
    public static void main(String[] args) {
        atomicIntegerArray.getAndSet(0, 3);//第一个参数为数组的下标，第二个参数为要赋予的新值
        System.out.println(atomicIntegerArray.get(0));//获取下标为0的值
        System.out.println(value[0]);//只改变了对象的引用，原对象的值是不变的
    }
</code></pre>
<h5 id="toc_h5_18">AtomicReference的简单使用</h5>
<pre><code>    static AtomicReference&amp;lt;UserInfo&amp;gt; atomicReference = new AtomicRefe
ence&amp;lt;UserInfo&amp;gt;();
    public static void main(String[] args) {
        UserInfo user = new UserInfo("Mark", 15);//要修改的实体的实例
        atomicReference.set(user);


    UserInfo updateUser = new UserInfo("Bill", 17);//要变化的新实例
    atomicReference.compareAndSet(user, updateUser);//先比较，再赋值
    System.out.println(atomicReference.get().getName());
    System.out.println(atomicReference.get().getAge());
    System.out.println(user.getName());//改变的是对象的引用而不是对象本身
    System.out.println(user.getAge());
}

//定义一个实体类
static class UserInfo {
    private String name;
    private int age;
    public UserInfo(String name, int age) {
        this.name = name;
```

```
        this.age = age;
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }
}

</code></pre>

<h5 id="toc_h5_19">UseAtomicStampedReference的简单使用</h5>
<ul>
<li>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
可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
。</li>
</ul>
<p><code>public AtomicStampedReference(V initialRef, int initialStamp) { pair = Pair.of(initia
Ref, initialStamp); }</code><br>
以上为UseAtomicStampedReference的构造方法，创建实例对象时，需将引用以及对应的初始版本
传入。</p>
<p>该类的api有</p>
<ol>
<li>V getReference()</li>
<li>int getStamp() 获取版本号</li>
<li>public boolean compareAndSet(V  expectedReference, V  newReference, int expectedSt
mp, int newStamp) 如果当前引用 等于 预期值并且 当前版本戳等于预期版本戳, 将更新新的引用和
的版本戳到内存</li>
<li>boolean attemptStamp(V expectedReference, int newStamp) 如果当前引用 等于 预期引用,
将更新新的版本戳到内存</li>
<li>void set(V newReference, int newStamp)  设置当前引用的新引用和版本戳</li>
</ol>
<pre><code>    static AtomicStampedReference&amp;lt;String&amp;gt; stampedReference =
        new AtomicStampedReference&amp;lt;&amp;gt;("Test",0); //初始值，跟初始化的版本号


public static void main(String[] args) throws InterruptedException {
    final int oldStamp = stampedReference.getStamp();//获取初始的版本号
    final String oldReferenc = stampedReference.getReference();//获取原来的值
    System.out.println("原值为：" + oldReferenc+"  版本号为：  "+oldStamp);

    Thread rightStampThread = new Thread(() -&amp;gt; System.out.println(Thread.currentThread()
getId()
            +"当前变量值："+oldReferenc+"当前版本戳："+oldStamp
            +" 赋值是否成功：  " + stampedReference.compareAndSet(oldReferenc, oldReferenc+"
ava",
                    oldStamp, oldStamp+1)));

    Thread errorStampThread = new Thread(() -&amp;gt; {
        String reference = stampedReference.getReference();
        System.out.println(Thread.currentThread().getId()
                +"当前变量值："+reference+"当前版本戳："+ stampedReference.getStamp()
                +" 赋值是否成功：  " + stampedReference.compareAndSet(reference, reference+"C",
```

```
                    oldStamp, oldStamp+1));

        });

        rightStampThread.start();
        rightStampThread.join();// 等待线程结束
        errorStampThread.start();
        errorStampThread.join();
        System.out.println("原值为：" + stampedReference.getReference()+"    版本号为：  "+stamp
dReference.getStamp());
}

</code></pre>

<p>输出如下：</p>
<pre><code>原值为：Test    版本号为：  0
12当前变量值：Test当前版本戳：0 赋值是否成功：  true
13当前变量值：TestJava当前版本戳：1 赋值是否成功：  false
原值为：TestJava    版本号为：  1
</code></pre>
<p><strong>参考自：</strong><br>
JAVA 中的 CAS ：  <a href="https://ld246.com/forward?goto=https%3A%2F%2Fjuejin.im%2F
ost%2F5a75db20f265da4e826320a9" target="_blank" rel="nofollow ugc">https://juejin.im/p
st/5a75db20f265da4e826320a9</a><br>
CAS自旋：  <a href="https://ld246.com/forward?goto=https%3A%2F%2Fblog.csdn.net%2Fsin
t_28028941%2Farticle%2Fdetails%2F53539775" target="_blank" rel="nofollow ugc">https://b
og.csdn.net/sinat_28028941/article/details/53539775</a><br>
JAVA CAS实现原理与使用：  <a href="https://ld246.com/forward?goto=https%3A%2F%2Fblog
csdn.net%2Fu011506543%2Farticle%2Fdetails%2F82392338" target="_blank" rel="nofollow u
c">https://blog.csdn.net/u011506543/article/details/82392338</a><br>
JavaGuide：  <a href="https://ld246.com/forward?goto=https%3A%2F%2Fsnailclimb.top%2FJ
vaGuide%2F%23%2F%3Fid%3D%25e5%25b9%25b6%25e5%258f%2591" target="_blank" rel=
nofollow ugc">https://snailclimb.top/JavaGuide/#/?id=%e5%b9%b6%e5%8f%91</a></p>
```