

AtomicInteger源码分析与并发漫谈

2015-09-25 10:23 feiying 0 阅读 101

java中讨论并发编程的时候，通常使用int i进行讨论，

一种情况是在多线程程序中，如果对i进行多次读取和赋值操作，如果什么也不加的话，在我赋值之后，其它线程不能立刻识别这个最新的i的值；

其主要原因是因为，在class编译生成汇编指令的时候，一般会做代码优化，也就是i直接从寄存器中读取，不再从内存拿数据；

因为java的赋值语句编译成cpu汇编指令也是一句，这里就是一个可见性的问题，

java中虽然有全局代码优化的开关，但是你要将整体java优化关闭，那java代码性能会拖得非常慢，

可以缩小范围，对这个int使用volatile关键字，告知编译的时候，别给这个int做优化，该从内存中拿东西我就拿东西；

java的volatile关键字其实本意和并发真的无关，他原本是从C++的舶来品，其主要的目的就是做禁止局部变量的编译优化，

除了从内存中拿数据跳过寄存器之外，还有很多在汇编指令集的优化，都是由volatile关键字来做的，

现在把volatile关键字看成和并发相关的关键字，实际是比较局限的见解；

可见性可以保证，并不能说明你程序就可以支撑并发场景，

原子性也得保证，最典型的是i++,i++最终编译成各种cpu汇编指令一般为3条指令，3条指令"同进退"这才叫原子性；

上述可见性中的给i赋值是1条指令，从CPU的特性来看，一般1条指令肯定是原子的，即使通过cpu的时钟周期打算，他也会在cpu级别加锁来保证1条汇编指令不能被其他线程打扰

因此i++这条java语句肯定不是原子性的，

这个时候，一般会使用synchronized关键字来包装这个i，

或者觉得想对锁细粒度的操控，也可以使用lock包中的ReentrantLock类，这个类级别和synchronized一样，但是更灵活的对锁进行设置和操控

但是，从操作系统的角度来看，synchronized这种级别的锁相当于调用操作系统的mutex_lock，这就是相当于让当前线程放入内核的等待队列中等待被唤醒

相当于有一个放弃cpu时间片休眠的过程，一直到对象的锁没有了，又被唤醒，抢到对象的锁，能抢到当前线程开始执行，抢不到继续休眠

上述的休眠-唤醒-休眠-唤醒 ... 的过程非常的耗时，效率很低，一般一次会错过几万个cpu时钟周期

而AtomicInteger 类就是java和cpu级别的CAS指令结合的典型案例，下面是两个程序的对比，差距惊人：

```
public class AtomicIntegerTest {
    private int value;

    public AtomicIntegerTest(int value) {
        this.value = value;
    }

    public synchronized int increase() {
        return value++;
    }

    public static void main(String args[]) {
        long start = System.currentTimeMillis();

        AtomicIntegerTest test = new AtomicIntegerTest(0);
        for (int i = 0; i < 1000000; i++) {
            test.increase();
        }
        long end = System.currentTimeMillis();
        System.out.println("time elapse:" + (end - start));

        long start1 = System.currentTimeMillis();
        AtomicInteger atomic = new AtomicInteger(0);
        for (int i = 0; i < 1000000; i++) {
            atomic.incrementAndGet();
        }
        long end1 = System.currentTimeMillis();
        System.out.println("time elapse:" + (end1 - start1));
    }
}
```

1. ←

<terminated> AtomicInteger
time elapse:61

2

time elapse:28

应用服务器技术讨论圈

一个是61秒，一个是28秒；

几乎在一倍左右的差距；

分析一下 AtomicInteger 类是怎么做的：

1.compareAndSet方法

典型的CAS的操作方法，调用JNI底层的cpu指令，

对于这部分的代码封装在sun的包中，并且根据不同cpu的特性，调用不同的JNI代码，

顾名思义，这个类也叫做unsafe

2.原子增加和减少方法


```

/**
 * Atomically adds the given value to the current value.
 *
 * @param delta the value to add
 * @return the previous value
 */
public final int getAndAdd(int delta) {
    for (;;) {
        int current = get();
        int next = current + delta;
        if (compareAndSet(current, next))
            return current;
    }
}

/**
 * Atomically increments by one the current value.
 *
 * @return the updated value
 */
public final int incrementAndGet() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return next;
    }
}

/**
 * Atomically decrements by one the current value.
 *
 * @return the updated value
 */
public final int decrementAndGet() {
    for (;;) {
        int current = get();
        int next = current - 1;
        if (compareAndSet(current, next))
            return next;
    }
}

```

 应用服务器技术讨论圈

这些方法整体调用思路是包装在一个while循环中，

a.通过get方法拿到最新的值，

b.通过计算增加减少1或者n

c.调用CAS方法：如果a这一步拿到的最新值确实是"最新的值"，这个时刻立刻cpu级别进行CAS锁定，将这块内存地址赋成第二步b的值；

注意c这一步判断加赋值是原子性的，而不是a,b,c三步是原子性的，所以看到如果不成的话，就套了一个while循环，不行再试；

通过这个也可以发现，CAS并不是想象中的那么完美，在竞争压力极大的情况下，这种循环会消耗CPU大量的压力，导致cpu干不了正事；

但CAS的操作，可比线程休眠，唤醒要强得太多太多，一个相当于在用户态一直没有出来，一个去到内核态进行排队再到用户态，二者的数量级不是一种；

CAS的锁也叫乐观锁，之所以乐观，就是因为他觉得压力没那么大，觉得一下子就可以拿到；

3.get方法和volatile关键字

其实整个AtomicInteger类的整体架构是非常简单的

下载《开发者大全》

下载 (/download/dev.apk)



它引自sun包中的unsafe实现，用以调用不同cpu的JNI的CAS操作；

AtomicInteger类包装的实际的值就是value，这个属性是它的实际的值，AtomicInteger是包装类

如下：

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // 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;

    /**
     * Creates a new AtomicInteger with the given initial value.
     *
     * @param initialValue the initial value
     */
    public AtomicInteger(int initialValue) {
        value = initialValue;
    }

    /**
     * Creates a new AtomicInteger with initial value {@code 0}.
     */
    public AtomicInteger() {
    }

    /**
     * Gets the current value.
     *
     * @return the current value
     */
    public final int get() {
        return value;
    }
}
```



应用服务器技术讨论圈

而get方法需要引起注意，为什么get方法可以获得最新的值呢？

其原因是value采用的是volatile，因为看到上述的多个方法都会调用这个get，这也就是为什么使用volatile的原因，保证可见性；

整体总结：

AtomicInteger 类的原理是 volatile + CAS；

分享：

阅读 101 0

下载《开发者大全》

下载 (/download/dev.apk)



应用服务器技术讨论圈 更多文章
东方通加码大数据业务 拟募资8亿收购微智信业 (/html/308/201504/206211355/1.html)
玩转Netty – 从Netty3升级到Netty4 (/html/308/201504/206233287/1.html)
金蝶中间件2015招聘来吧！Come on！ (/html/308/201505/206307460/1.html)
GlassFish 4.1 发布，J2EE 应用服务器 (/html/308/201505/206323120/1.html)
Tomcat对keep-alive的实现逻辑 (/html/308/201505/206357679/1.html)

猜您喜欢
那些红极一时的BBS大佬们，你们还好吗？ (/html/190/201506/206898169/1.html)
基于MySQL的高可用可扩展架构探讨 (/html/313/201308/10000205/1.html)
我和尚学堂：那些不得不说的事儿 (/html/462/201602/405723954/1.html)
HTML5游戏前端开发秘籍 - 腾讯ISUX (/html/155/201405/200827095/1.html)
听说 libevent 的并发工作做得很好？ (/html/138/201606/2650918178/1.html)