链滴

【JUC】CAS 底层原理

作者：openshell

原文链接：https://ld246.com/article/1643861891298

来源网站：链滴

比较并交换（compare and set）

CAS翻译成中文即：**比较并交换**，他是一条CPU原语操作（保证读写的原子性），底层基于c/c++实，直接通过指针操作内存实现。通过使用CAS原语能够解决并发更新数据的问题，不用额外加锁去保线程安全。

CAS的使用

以下例子，通过AtomicInteger类的两个方法阐述CAS的使用。

getAndIncrement()

有个面试高频考点，i++线程安全吗?

实际上在多线程环境下，基于JMM模型，每个线程都有各自的本地内存，线程工作时会从主内存拷贝份自己需要的数据，在值回写主内存时容易引发线程安全问题。

比如两个线程t1,t2同时进行 i++操作，

1. t1与t2同时从主内存拷贝了i=0的值到本地内存
2. t1率先完成了两次++操作，此时主内存值为2。t2因故挂起未执行，且t2的本地内存值为0。
3. t2被唤醒，执行1次++操作，回写主内存，　**将主内存的值覆盖为1**，至此引发线程安全问题。

有同学可能要讲，加上 volatile保证可见性，就不会有上述问题了，那我们继续看看：

首先i++实际上是由四条指令组成的

第一条指令 getfield ： 取出n这个元素

第二条指令 iconst_1 ： 把常量1压入栈

第三条指令 iadd ： 把n和常量1进行相加

第四条指令 putfield ： 把数据刷回主内存

t1与t2即便能读取到最新的值，但盖不住i++操作非原子，上述地三条指令若同时被执行，则同样引线程安全问题。

volatile能够保证安全性，但是不能保证原子性，它仅能保证读到最新的数据。

跑代码看看：

```
@Data
public class VolatileTestData {
    /**
     * 普通int类型
     */
    private int number = 0;
    /**
     * 原子类int
     */
    private AtomicInteger atomicInteger = new AtomicInteger();

    /**
     * 使用++操作，线程不安全。
     *
     * @return: void
     * @author openshell
     * @date: 2022/2/3 10:48
     */
    public void add() {
        number++;
    }

    /**
     * 基于CPU原语，CAS原理进行自增，线程安全。
     *
     * @return: void
     * @author openshell
     * @date: 2022/2/3 10:48
     */
    public void atomicAdd() {
        atomicInteger.getAndIncrement();
    }
}

@Test
    public void testAtomicity() {
        VolatileTestData volatileTestData = new VolatileTestData();
        for (int i = 0; i < 20; i++) {
            new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    //调用i++方法自增
                    volatileTestData.add();
                    //使用AtomicInteger自增（CAS）
                    volatileTestData.atomicAdd();
```

```
                }
            }).start();
        }
        //注：
        //1.Thread.activeCount()用于返回当前线程的线程组中活动线程的数量,返回的值只是一个估计，因为当此方法遍历内部数据结构时，线程数可能会动态更改。
        //2.此处无内部数据结构故无影响
        //3.IntelliJ IDEA执行用户代码的时候，实际是通过反射方式去调用，而与此同时会创建一个Monitor Ctrl-Break 用于监控目的，故线程会多一个。
        while (Thread.activeCount() > 2) {
            System.out.println("线程数量：" + Thread.activeCount());
            //线程回退到可运行状态，将cpu时间交给同级或者更高优先级的线程
            Thread.yield();
        }
        System.out.println("普通i++的执行结果：" + volatileTestData.getNumber());
        System.out.println("Atomically increments的结果：" + volatileTestData.getAtomicInteger());
    }
```

执行结果：

```
线程数量：4
线程数量：4
普通i++的执行结果：18306
Atomically increments的结果：20000
```

多次运行i++操作均无法满足期望值，原子整型的计算结果总是正确的。

compareAndSet()

运行：

```
AtomicInteger atomicInteger = new AtomicInteger(5);
        System.out.println(atomicInteger.compareAndSet(5, 2021) + "\t 当前值为：" + atomicInteger.get());
        System.out.println(atomicInteger.compareAndSet(5, 1024) + "\t 当前值为：" + atomicInteger.get());
```

输出：

```
true     当前值为：2021
false     当前值为：2021
```

compareAndSet源码：

若当前值等于期望值，则原子的更新给定的值。

```
/**
     * Atomically sets the value to the given updated value
     * if the current value {@code ==} the expected value.
     *
     * @param expect the expected value
     * @param update the new value
     * @return {@code true} if successful. False return indicates that
```

```
     * the actual value was not equal to the expected value.
     */
    public final boolean compareAndSet(int expect, int update) {
        return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
    }
```

该方法调用了unsafe类，unsafe位于jre下的rt.jar，由于该类脱离了jvm的管理，直接操作内存，可带来不可预估的错误，故取名为unsafe。

CAS的缺点

尽管cas算法保证了数据读写的原子性，但仍存在一些问题。

- ABA问题

eg:有三个线程t1 t2 t3，同时修改变量i,t1与t2的功能是将i改为B，t3的功能是将i改为A：

1. t1执行成功，i=B,t2被挂起
2. t3被唤醒，将i改为A
3. t2被唤醒，此时值为A，t2仍然执行i=B的操作，t2并不知道第二步的发生，至此产生ABA问题

解决办法：CAS属于乐观锁，可以在使用时增加版本号，解决以上问题。

代码示例：

```
public class ABAResolveTest {

    public static void main(String[] args) {
        testStamp();
    }

    private static void testStamp() {
        AtomicStampedReference<Integer> atomicStampedReference = new AtomicStampedRe
erence<>(1, 1);

        new Thread(()->{
            int[] stampHolder = new int[1];
            //返回当前相同引用的值和信号量
            int value = atomicStampedReference.get(stampHolder);
            int stamp = stampHolder[0];
            System.out.println("thread 1 read value: " + value + ", stamp: " + stamp);

            // 阻塞1s
            LockSupport.parkNanos(1000000000L);

            if (atomicStampedReference.compareAndSet(value, 3, stamp, stamp + 1)) {
                System.out.println("thread 1 update from " + value + " to 3");
            } else {
                System.out.println("thread 1 update fail!");
            }
        }).start();

        new Thread(()->{
            int[] stampHolder = new int[1];
```

```
            int value = atomicStampedReference.get(stampHolder);
            int stamp = stampHolder[0];
            System.out.println("thread 2 read value: " + value + ", stamp: " + stamp);
            if (atomicStampedReference.compareAndSet(value, 2, stamp, stamp + 1)) {
                System.out.println("thread 2 update from " + value + " to 2");

                // do sth

                value = atomicStampedReference.get(stampHolder);
                stamp = stampHolder[0];
                System.out.println("thread 2 read value: " + value + ", stamp: " + stamp);
                if (atomicStampedReference.compareAndSet(value, 1, stamp, stamp + 1)) {
                    System.out.println("thread 2 update from " + value + " to 1");
                }
            }
        }).start();
    }
}
```

- 循环可能导致CPU开销过大

在unsafe源码中可以看出，若未能按照期望值更新，该方法会一直循环下去，这对CPU的开销是极大。

可以通过增加次数限制解决该问题。

- 只能保证一个共享变量的原子操作

如果要保证一个代码块的原子性，CAS则不支持。此时，可以通过加锁，或者将多个共享变量转化为个执行CAS。

参考文章：死磕 java并发包之AtomicStampedReference源码分析（ABA问题详解） - 掘金 (juejin.c)