3、CAS&Atomic 原子操作详解

什么是原子操作？如何实现原子操作？

什么是原子性？相信很多同学在工作中经常使用事务，事务的一大特性就是原子性（事务具有 ACID 四大特性），一个事务包含多个操作，这些操作要么全部执行，要么全都不执行。

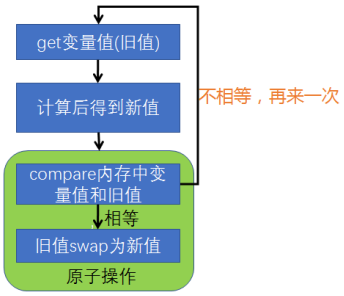
并发里的原子性和原子操作是一样的内涵和概念，假定有两个操作 A 和 B 都包含多个步骤，如果从执行 A 的线程来看，当另一个线程执行 B 时，要么将 B 全部执行完，要么完全不执行 B，执行 B 的线程看 A 的操作也是一样的，那么 A 和 B 对彼此来说是原子的。

实现原子操作可以使用锁，锁机制，满足基本的需求是没有问题的了，但是有的时候我们的需求并非这么简单，我们需要更有效，更加灵活的机制，synchronized 关键字是基于阻塞的锁机制，也就是说当一个线程拥有锁的时候，访问同一资源的其它线程需要等待，直到该线程释放锁，

这里会有些问题：首先，如果被阻塞的线程优先级很高很重要怎么办？其次，如果获得锁的线程一直不释放锁怎么办？同时，还有可能出现一些例如死锁之类的情况，最后，其实锁机制是一种比较粗糙，粒度比较大的机制，相对于像计数器这样的需求有点儿过于笨重。为了解决这个问题，Java 提供了 Atomic 系列的原子操作类。

这些原子操作类其实是使用当前的处理器基本都支持 CAS 的指令，比如 Intel 的汇编指令 cmpxchg，每个厂家所实现的具体算法并不一样，但是原理基本一样。每一个 CAS 操作过程都包含三个运算符：一个内存地址 V，一个期望的值 A 和一个新值 B，操作的时候如果这个地址上存放的值等于这个期望的值 A，则将地址上的值赋为新值 B，否则不做任何操作。

CAS 的基本思路就是，如果这个地址上的值和期望的值相等，则给其赋予新值，否则不做任何事儿，但是要返回原值是多少。自然 CAS 操作执行完成时，在业务上不一定完成了，这个时候我们就会对 CAS 操作进行反复重试，于是就有了循环 CAS。很明显，循环 CAS 就是在一个循环里不断的做 cas 操作，直到成功为止。Java 中的 Atomic 系列的原子操作类的实现则是利用了循环 CAS 来实现。



CAS 实现原子操作的三大问题

ABA 问题。

因为 CAS 需要在操作值的时候，检查值有没有发生变化，如果没有发生变化则更新，但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。在变量前面追加上版本号，每次变量更新的时候把版本号加 1，那么 A→B→A 就会变成 1A→2B→3A。举个通俗点的例子，你倒了一杯水放桌子上，干了点别的事，然后同事把你水喝了又给你重新倒了一杯水，你回来看水还在，拿起来就喝，如果你不管水中间被人喝过，只关心水还在，这就是 ABA 问题。

如果你是一个讲卫生讲文明的小伙子，不但关心水在不在，还要在你离开的时候水被人动过没有，因为你是程序员，所以就想起了放了张纸在旁边，写上初始值 0，别人喝水前麻烦先做个累加才能喝水。

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。

当对一个共享变量执行操作时，我们可以使用循环 CAS 的方式来保证原子操作，但是对多个共享变量操作时，循环 CAS 就无法保证操作的原子性，这个时候就可以用锁。

还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量 i＝2，j=a，合并一下 ij=2a，然后用 CAS 来操作 ij。从 Java 1.5 开始，JDK 提供了 AtomicReference 类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行 CAS 操作。

Jdk 中相关原子操作类的使用

参见代码，包 cn.tulingxueyuan.cas 下

AtomicInteger

•int addAndGet（int delta）：以原子方式将输入的数值与实例中的值（AtomicInteger 里的 value）相加，并返回结果。

•boolean compareAndSet（int expect，int update）：如果输入的数值等于预期值，则以原子方式将该值设置为输入的值。

•int getAndIncrement()：以原子方式将当前值加 1，注意，这里返回的是自增前的值。

•int getAndSet（int newValue）：以原子方式设置为 newValue 的值，并返回旧值。

AtomicIntegerArray

主要是提供原子的方式更新数组里的整型，其常用方法如下。

•int addAndGet（int i，int delta）：以原子方式将输入值与数组中索引 i 的元素相加。

•boolean compareAndSet（int i，int expect，int update）：如果当前值等于预期值，则以原子方式将数组位置 i 的元素设置成 update 值。

需要注意的是，数组 value 通过构造方法传递进去，然后 AtomicIntegerArray 会将当前数组复制一份，所以当 AtomicIntegerArray 对内部的数组元素进行修改时，不会影响传入的数组。

更新引用类型

原子更新基本类型的 AtomicInteger，只能更新一个变量，如果要原子更新多个变量，就需要使用这个原子更新引用类型提供的类。Atomic 包提供了以下 3 个类。

AtomicReference

原子更新引用类型。

AtomicStampedReference

利用版本戳的形式记录了每次改变以后的版本号，这样的话就不会存在 ABA 问题了。这就是 AtomicStampedReference 的解决方案。AtomicMarkableReference 跟 AtomicStampedReference 差不多， AtomicStampedReference 是使用 pair 的 int stamp 作为计数器使用，AtomicMarkableReference 的 pair 使用的是 boolean mark。还是那个水的例子， AtomicStampedReference 可能关心的是动过几次， AtomicMarkableReference 关心的是有没有被人动过，方法都比较简单。

AtomicMarkableReference：

原子更新带有标记位的引用类型。可以原子更新一个布尔类型的标记位和引用类型。构造方法是 AtomicMarkableReference（V initialRef，booleaninitialMark）。

原子更新字段类

如果需原子地更新某个类里的某个字段时，就需要使用原子更新字段类，Atomic 包提供了以下 3 个类进行原子字段更新。

要想原子地更新字段类需要两步。第一步，因为原子更新字段类都是抽象类，每次使用的时候必须使用静态方法 newUpdater()创建一个更新器，并且需要设置想要更新的类和属性。第二步，更新类的字段（属性）必须使用 public volatile 修饰符。

AtomicIntegerFieldUpdater：

原子更新整型的字段的更新器。

AtomicLongFieldUpdater：

原子更新长整型字段的更新器。

AtomicReferenceFieldUpdater：

原子更新引用类型里的字段。

LongAdder

JDK1.8 时，java.util.concurrent.atomic 包中提供了一个新的原子类：LongAdder。根据 Oracle 官方文档的介绍，LongAdder 在高并发的场景下会比它的前辈————AtomicLong 具有更好的性能，代价是消耗更多的内存空间。

AtomicLong 是利用了底层的 CAS 操作来提供并发性的，调用了 **Unsafe** 类的 **getAndAddLong** 方法，该方法是个 **native** 方法，它的逻辑是采用自旋的方式不断更新目标值，直到更新成功。

在并发量较低的环境下，线程冲突的概率比较小，自旋的次数不会很多。但是，高并发环境下，N 个线程同时进行自旋操作，会出现大量失败并不断自旋的情况，此时 **AtomicLong** 的自旋会成为瓶颈。

这就是 **LongAdder** 引入的初衷——解决高并发环境下 **AtomicLong** 的自旋瓶颈问题。

AtomicLong 中有个内部变量 **value** 保存着实际的 long 值，所有的操作都是针对该变量进行。也就是说，高并发环境下，value 变量其实是一个热点，也就是 N 个线程竞争一个热点。

```
private volatile long value;
```

LongAdder 的基本思路就是***分散热点***，将 value 值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的那个值进行 CAS 操作，这样热点就被分散了，冲突的概率就小很多。如果要获取真正的 long 值，只要将各个槽中的变量值累加返回。

这种做法和 ConcurrentHashMap 中的“分段锁”其实就是类似的思路。

LongAdder 提供的 API 和 **AtomicLong** 比较接近，两者都能以原子的方式对 long 型变量进行增减。

但是 **AtomicLong** 提供的功能其实更丰富，尤其是 **addAndGet**、**decrementAndGet**、**compareAndSet** 这些方法。

addAndGet、**decrementAndGet** 除了单纯的做自增自减外，还可以立即获取增减后的值，而 **LongAdder** 则需要做同步控制才能精确获取增减后的值。如果业务需求需要精确的控制计数，做计数比较，**AtomicLong** 也更合适。

另外，从空间方面考虑，**LongAdder** 其实是一种“空间换时间”的思想，从这一点来讲 **AtomicLong** 更适合。

总之，低并发、一般的业务场景下 AtomicLong 是足够了。如果并发量很多，存在大量写多读少的情况，那 LongAdder 可能更合适。适合的才是最好的，如果

真出现了需要考虑到底用 AtomicLong 好还是 LongAdder 的业务场景，那么这样的讨论是没有意义的，因为这种情况下要么进行性能测试，以准确评估在当前业务场景下两者的性能，要么换个思路寻求其它解决方案。

对于 **LongAdder** 来说，内部有一个 base 变量，一个 Cell[]数组。

base 变量：非竞态条件下，直接累加到该变量上。

Cell[]数组：竞态条件下，累加个各个线程自己的槽 Cell[i]中。

```
transient volatile Cell[] cells;

transient volatile long base;
```

所以，最终结果的计算应该是

```
public long sum() {
    Cell[] as = cells; Cell a;
    long sum = base;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

$$value = base + \sum_{i=0}^{n} Cell[i]$$

在实际运用的时候，只有从未出现过并发冲突的时候，base 基数才会使用到，一旦出现了并发冲突，之后所有的操作都只针对 Cell[]数组中的单元 Cell。

```
public void add(long x) {
    Cell[] as; long b, v; int m; Cell a;
    if ((as = cells) != null || !casBase(b = base, val: b + x)) {
        boolean uncontended = true;
        if (as == null || (m = as.length - 1) < 0 ||
            (a = as[getProbe() & m]) == null ||
            !(uncontended = a.cas(v = a.value, val: v + x)))
            longAccumulate(x, fn: null, uncontended);
    }
}
```

而 LongAdder 最终结果的求和，并没有使用全局锁，返回值不是绝对准确的，因为调用这个方法时还有其他线程可能正在进行计数累加，所以只能得到某个时刻的近似值，这也就是 **LongAdder** 并不能完全替代 **LongAtomic** 的原因之一。

而且从测试情况来看，线程数越多，并发操作数越大，LongAdder 的优势越大，线程数较小时，AtomicLong 的性能还超过了 LongAdder。

其他新增

除了新引入 LongAdder 外，还有引入了它的三个兄弟类：**LongAccumulator**、**DoubleAdder**、**DoubleAccumulator**。

LongAccumulator 是 LongAdder 的增强版。LongAdder 只能针对数值的进行加减运算，而 LongAccumulator 提供了自定义的函数操作。

通过 LongBinaryOperator，可以自定义对入参的任意操作，并返回结果（LongBinaryOperator 接收 2 个 long 作为参数，并返回 1 个 long）。

LongAccumulator 内部原理和 LongAdder 几乎完全一样。

DoubleAdder 和 DoubleAccumulator 用于操作 double 原始类型。

本文链接：

http://note.youdao.com/noteshare?id=b004370f0a4f61a9c81a81d4fae25453&sub=67D3563DF51E4CCC818399C5B57F855D