参考：

深入浅出CAS（占小狼）

大白话聊聊Java并发面试问题之Java 8如何优化CAS性能？【石杉的架构笔记】

前言

CAS是啥呢，即compare and swap（比较并替换），之前介绍AQS时就有提过调用lock时会用CAS操作将那个值变更，那么这个比较并替换的操作到底有什么讲究呢，为什么要去使用呢。

并发安全性问题和一般解决方案

这里给出一个案例，一个数据初始化为0，我们通过并发的方式，让20个线程对这个数据操作加1，我们想得到最好结果是20。我们看我们的测试代码。

```
public class HelloWorld {

	public static int a = 0;

	public static void increatment() {
		a++;
		System.out.println("结果：" + a);
	}
}
```

```
public class FirstDemo implements Runnable {

	public void count() {
		HelloWorld.increatment();
	}

	@Override
	public void run() {
		count();
	}

	public static void main(String[] args) {
		for (int i = 0; i < 20; i++) {
```

```
                new Thread(new FirstDemo()).start();
            }
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("a的最终结果是：" + HelloWorld.a);
        }

}
```

这个样子结果是咋样呢。

```
结果：4
结果：4
结果：4
结果：4
结果：5
结果：4
结果：6
结果：7
结果：8
结果：9
结果：10
结果：12
结果：14
结果：13
结果：11
结果：16
结果：15
结果：17
结果：19
结果：18
a的最终结果是：19
```

这是一个结果，关注最后一句，至少我不管能不能有正确的，至少这次运行结果就是与我们预期的不一致的，术语就是线程不安全性。

针对这个我相信大多数人都是一个解决方案的，就是synchronized，就像这样改造。

```
public static synchronized void increatment() {
            a++;
            System.out.println("结果：" + a);
        }
```

这样这个方法就被加锁了，这样20个线程就得老老实实排队执行，我们看结果。

```
结果：1
结果：2
```

```
结果：3
结果：4
结果：5
结果：6
结果：7
结果：8
结果：9
结果：10
结果：11
结果：12
结果：13
结果：14
结果：15
结果：16
结果：17
结果：18
结果：19
结果：20
a的最终结果是：20
```

哇，单步结果按序打印，结果也是我们想要的20，好像是皆大欢喜的结果，但是不觉得奇怪吗，我们在这里用多线程的初衷是啥，我们是想通过并发的方式快速得到结果20，但是因为不安全，最终加了锁。加锁代表了啥，变相的变成了串行，这样一步步算到20，我何不直接for循环一个线程干呢？所以基于轻量级的并发安全性问题我们不是说无脑用synchronized就行了，你得明白用并发的初衷是啥。

既然这里我们想通过并发快速算到20，难道真的只能冒险得到错误答案吗，当然不是，所以才有CAS操作嘛。

Atomic原子类及其实现原理

说了那么久的CAS操作，也没说具体的实现，这里先介绍一种CAS操作--Atomic原子类，先看看代码实现。

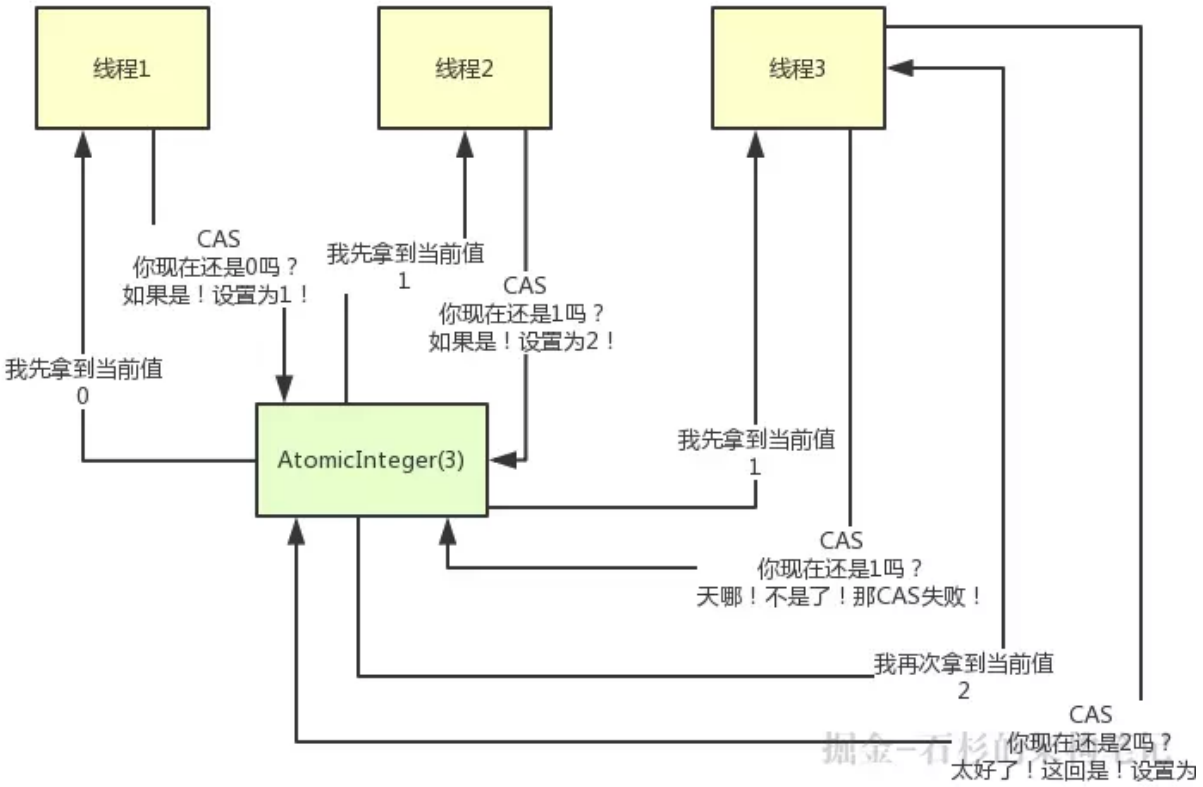
```
public class HelloWorld {

    public static AtomicInteger a = new AtomicInteger(0);

    public static void increatment() {
        a.addAndGet(1);
        System.out.println("结果：" + a);
    }
}
```

这里我们把原本的int改装成了AtomicInteger，然后我们看结果。

```
结果：4
结果：4
结果：4
结果：4
结果：5
结果：6
结果：7
结果：8
结果：10
结果：9
结果：11
结果：12
结果：13
结果：15
结果：16
结果：17
结果：15
结果：19
结果：20
结果：19
a的最终结果是：20
```

可以看到，计算的单步打印过程还是很凌乱的，但这不是CAS有问题，是我们打印的问题，并发的情况下可能有多个打印的同时获取的a值是一样的，恰恰说明了CAS的快速，同时结果的准确又展现了它的能力，这才是并发应有的姿态，这样简单的需求下我们还要用到synchronized的排队，岂不是很笨重。

那么提到的这个AtomicInteger的原子类的实现原理是啥呢，我们赋一张图协助理解。



好！现在我们对照着上面的图，来看一下这整个过程：

- 首先第一步，我们假设线程一咔嚓一下过来了，然后对AtomicInteger执行incrementAndGet()操作，他底层就会先获取AtomicInteger当前的值，这个值就是0。
- 此时没有别的线程跟他抢！他也不管那么多，直接执行原子的CAS操作，问问人家说：兄弟，你现在值还是0吗？
- 如果是，说明没人修改过啊！太好了，给我累加1，设置为1。于是AtomicInteger的值变为1！
- 接着线程2和线程3同时跑了过来，因为底层不是基于锁机制，都是无锁化的CAS机制，所以他们俩可能会并发的同时执行incrementAndGet()操作。
- 然后俩人都获取到了当前AtomicInteger的值，就是1
- 接着线程2抢先一步发起了原子的CAS操作！注意，CAS是原子的，此时就他一个线程在执行！

- 然后线程2问：兄弟，你现在值还是1吗？如果是，太好了，说明没人改过，我来改成2
- 好了，此时AtomicInteger的值变为了2。关键点来了：现在线程3接着发起了CAS操作，但是他手上还是拿着之前获取到的那个1啊！
- 线程3此时会问问说：兄弟，你现在值还是1吗？
- 噩耗传来！！！这个时候的值是2啊！线程3哭泣了，他说，居然有人在这个期间改过值。算了，那我还是重新再获取一次值吧，于是获取到了最新的值，值为2。
- 然后再次发起CAS操作，问问，现在值是2吗？是的！太好了，没人改，我抓紧改，此时AtomicInteger值变为3！

嗯，虽然有点脸红，毕竟上面这段是照搬别人的，但写的是真的形象，读书人的事怎么能说是抄呢。

Atomic原子类的源码解析

我们就以AtomicInteger为例吧。

```
*/
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;
```

这个Unsafe类是CAS的核心类，由于Java无法直接访问底层系统，需要通过native方法访问，Unsafe相当于一个后门，基于该类就可以直接

操作特定内存的数据。

valueOffset代表内存偏移地址，Unsafe就是根据内存偏移地址去获取数据的。

value也就是我们操作的数据了，用volatile修饰能够保证多线程之间的内存可见性（volatile的整理后期会奉上）。

接下来我们来看看AtomicInteger如何去实现并发下的累加操作的。

```
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}

//unsafe.getAndAddInt
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        var5 = this.getIntVolatile(var1, var2);
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
    return var5;
}
```

上面的是AtomicInteger中的方法，下面的是调用的Unsafe中的方法。举个例子。假设线程A和线程B同时执行getAndAdd操作（分别跑在不同CPU上）：

1. AtomicInteger里面的value原始值为3，即主内存中AtomicInteger的value为3，根据Java内存模型，线程A和线程B各自持有一份value的副本，值为3；
2. 线程A通过getIntVolatile(var1, var2)拿到value值3，这时线程A被挂起；
3. 线程B也通过getIntVolatile(var1, var2)方法获取到value值3，运气好，线程B没有被挂起，并执行compareAndSwapInt方法比较内存值也为3，成功修改内存值为2；
4. 这时线程A恢复，执行compareAndSwapInt方法比较，发现自己手里的值(3)和内存的值(2)不一致，说明该值已经被其它线程提

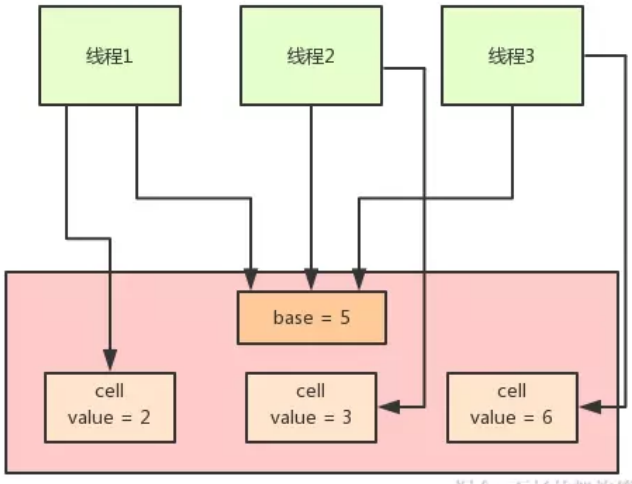
前修改过了，那只能重新来一遍了；

5. 重新获取value值，因为变量value被volatile修饰，所以其它线程对它的修改，线程A总是能够看到，线程A继续执行compareAndSwapInt进行比较替换，直到成功。

到这为止，更深层次的暂时就不去了解了，再往下就涉及到调用native方法了。

Atomic原子类的问题以及Java8的优化

其实想一想就能明白这种机制的问题，抢占式的修改，然后其它线程检测修改重新获取处理，那么万一有个线程无限抢占不到不就会无限自转了吗。针对出现这样的问题，Java8推出了一个新的类LongAdder，它采用分段CAS以及自动分段迁移的方式提升了CAS的性能。



大体意思就是所有线程先来操作base里面的值，然后发现更新的线程太多，就会开始实施分段CAS机制，也就是搞了一堆的cell给每个部分的线程去操作，这样就大大降低了自转的风险，同时它还提了自动分段迁移的机制，就是某个cell的CAS失败后会自动去找另一个cell的值继续操作，最后呢，把base值和cell的值加起来返回就是最终值了。简

单的说就是原本只有一个base，大家都来抢，有个弱小的家伙总抢不到，这时候我们多分担多来几个点，这样大家就不用聚在一起抢一个地方了，可以分流去抢其它点了，这时候弱小的家伙机会就多了点，要这样还抢不到那也没法子，可能太弱小了，所以只是大大降低了自转的风险。

CAS的缺点

用一个术语说就是ABA的问题，什么意思呢？

线程1读到的值是A，在准备赋值时检查还是A，这是后按照CAS机制就会去将值A改掉，但这样确定就是这样吗？

比如线程2在之前将值改为了B，但是线程3又把B改回了A，那么这时候线程1检查的A还是那个A吗？显然不是，那根据不一致原则，理应是重新获取A再重新处理，而这边只会认为没有变动直接修改。

当然Java并发包也提供了一个类用来处理这个问题AtomicStampedReference，他通过控制变量的版本来保证CAS的正确性，就和以前上学时一个班有两个同名时，老师为了区别比如会用“大”+名字，小+“名字”来进行区分。