原子类是如何利用 CAS 保证线程安全的?

原子类的**作用**和锁有类似之处，是为了**保证并发情况下线程安全**。不过原子类相比于锁，有一定的优势:

粒度更细：原子变量可以把竞争范围缩小到变量级别，通常情况下，锁的粒度都要大于原子变量的粒度。
效率更高：除了高度竞争的情况之外，使用原子类的效率通常会比使用同步互斥锁的效率更高，因为原子类底层利用了 CAS 操作，不会阻塞线程。

6种原子类:

类型	具体类
Atomic* 基本类型原子类	AtomicInteger、AtomicLong、AtomicBoolean
Atomic*Array 数组类型原子类	AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray
Atomic*Reference 引用类型原子类	AtomicReference、AtomicStampedReference、AtomicMarkableReference
Atomic*FieldUpdater 升级类型原子类	AtomicIntegerfieldupdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater
Adder 累加器	LongAdder、DoubleAdder
Accumulator 积累器	LongAccumulator、DoubleAccumulator

Atomic\ 基本类型原子类:

包括三种，分别是 AtomicInteger、AtomicLong 和 AtomicBoolean。

AtomicInteger：它是对于 int 类型的封装，我们可以不用基本类型 int，也不使用包装类型 Integer，而是直接使用 AtomicInteger，这样一来就自动具备了原子能力，使用起来非常方便。

AtomicInteger常用方法：

```
public final int get() //获取当前的值
public final int getAndSet(int newValue) //获取当前的值，并设置新的值
public final int getAndIncrement() //获取当前的值，并自增
public final int getAndDecrement() //获取当前的值，并自减
public final 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，则以原子方式将该值更新为输入值（update）
```

Array 数组类型原子类：

保证数组每一个元素都具备原子性。

它一共分为 3 种，分别是：

- 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。

Atomic\Reference 引用类型原子类：

引用类型原子类，可以让一个对象保证原子性。这样一来，AtomicReference 的能力明显比 AtomicInteger 强，因为一个对象里可以包含很多属性。

在这个类别之下，除了 AtomicReference 之外，还有：

- AtomicStampedReference：它是对 AtomicReference 的升级，在此基础上还加了时间戳，用于解决 CAS 的 ABA 问题。
- AtomicMarkableReference：和 AtomicReference 类似，多了一个绑定的布尔值，可以用于表示该对象已删除等场景。

Atomic\FieldUpdater 原子更新器：

一共有三种，分别是。

- AtomicIntegerFieldUpdater：原子更新整形的更新器；
- AtomicLongFieldUpdater：原子更新长整形的更新器；
- AtomicReferenceFieldUpdater：原子更新引用的更新器。

比如是整型的 int，实际它并不具备原子性，利用 Atomic*FieldUpdater，如果它是整型的，就使用 AtomicIntegerFieldUpdater 把已经声明的变量进行升级，这样一来这个变量就拥有了 CAS 操作的能力。

为什么一不开始就声明 AtomicInteger?

答：出于以下两种原因：

1.历史原因，有些变量已经广泛运用，修复成本很高。

2.该变量偶尔需要用到它的原子性，那么久没必要直接变为 AtomicInteger，因为 AtomicInteger 比普通的变量更加耗费资源。

AtomicIntegerFieldUpdater示例？？？

Adder 加法器：

它里面有两种加法器，分别叫作 LongAdder 和 DoubleAdder。

Accumulator 积累器：

最后一种叫 Accumulator 积累器，分别是 LongAccumulator 和 DoubleAccumulator。

这两种原子类我们会在后面的课时中展开介绍。

以 AtomicInteger 为例，分析在 Java 中如何利用 CAS 实现原子操作？

首先看AtomicInteger 的getAndAdd方法：

```
//JDK 1.8实现
public final int getAndAdd(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta);
}
```

里面使用了 Unsafe 这个类，并且调用了 unsafe.getAndAddInt 方法

- Unsafe 类介绍：

> Unsafe 其实是 CAS 的核心类，它提供了硬件级别的原子操作，我们可以利用它直接操作内存数据。

Unsafe 的 objectFieldOffset方法得到当前这个原子类的 value 的**偏移地址**（英文Offset ，通过native 实现）， Unsafe 就是根据内存偏移地址获取数据的原值的，这样我们就能通过 Unsafe 来实现 CAS 了，具体CAS实现看下面代码：

- Unsafe 类的getAndAddInt方法介绍：

```
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;
}
```

getIntVolatile：获取var1 中的 var2 偏移，这是个 native 方法。

var1：操作的对象；

var2 ： 偏移量，借助它就可以获取到 value 的数值（获取到的是原子类的值）

var5：value 的数值（CAS的预期值）

var5 + var4：希望修改的数值，var4就是我们希望原子类所改变的数值，比如可以传入 +1，也可以传入 -1。（就是如果没有改动情况，最终插入的值）

compareAndSwapInt ： var5是之前获取到的原子类var2的值，var5在运算出var5 + var4过程中，如果var5变化了，和var2不一样了，就修改失败，否则修改成功。

什么是偏移地址：

有效地址=基地址+偏移地址

原子类和 volatile 有什么异同?

但是value++ 语句volatile不能保证线程安全，因为value++ 不仅有可见性，还有原子性问题。

所以volatile可以用来修饰 boolean 类型的标记位，因为对于标记位来讲，直接的赋值操作本身就是具备原子性的，再加上 volatile 保证了可见性，那么就是线程安全的了。

对于value++等需要存在原子性问题场景还是使用原子类。

所以一个数值操作的线程安全不仅要所有线程可见（volatile 可以实现）还要有原子性。

原子类和 synchronized 的异同点?

背后原理的不同：

synchronized 使用的是悲观锁。

原子类使用的是乐观锁（利用了 CAS 操作）。

使用范围的不同：

synchronized 可以修饰一个方法，又可以修饰一段代码，可以非常灵活地去控制它的应用范围。

原子变量的粒度是比较小的，可以是变量级别的。synchronized 也可以做到变量级别，但是有一点杀鸡焉用牛刀的感觉。

线程开销不同：

synchronized使用的悲观锁是比较重量级的，但开销是固定的。

原子类使用的乐观锁短期内的开销不大，但是随着时间的增加，它的开销也是逐步上涨的。

适用场景：

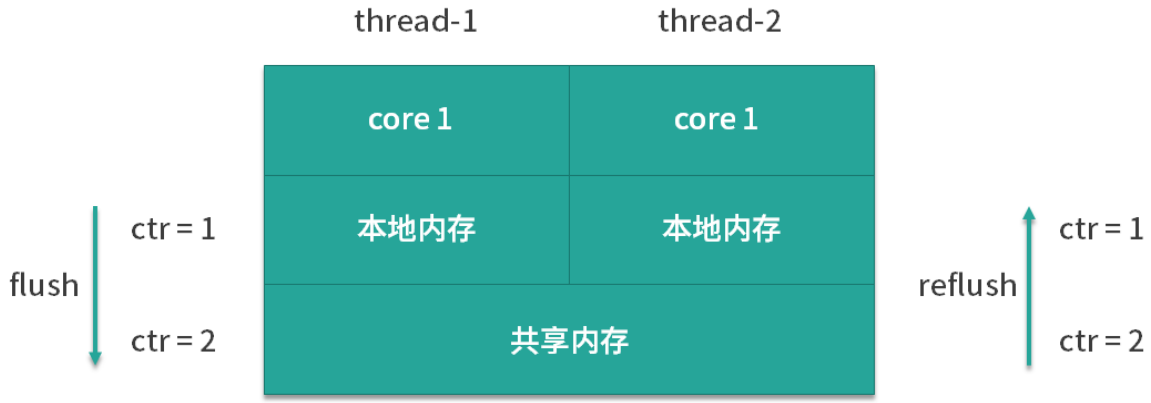
竞争非常激烈的情况下，推荐使用 synchronized；而在竞争不激烈的情况下，使用原子类会得到更好的效果。（synchronized 的性能随着 JDK 的升级，也在不断优化，现在使用了锁的晋升机制，先是无锁，再慢慢升为偏向锁、轻量级锁、重量级锁）

Atomic 原子类、Adder 加法器、Accumulator 积累器的对比：

> Adder 加法器、Accumulator 积累器是Java 8 引入的，是相对比较新的类。

AtomicLong 存在的问题以及和LongAdder 的对比：

AtomicLong高并发下性能并不好，因为如下图，每一次AtomicLong 的数值有变化的时候，它都需要进行 flush 和 refresh，比如原始值是0， core 1 把它改成 1 的话，要把最新结果flush 到下方的共享内存，再到右侧去往上 refresh 到core 2的本地内存，对于核心 2 而言，它才能感知到这次变化。由于竞争很激烈，这样的 flush 和 refresh 操作耗费了很多资源，而且 CAS 也会经常失败。



LongAdder 带来的改进和原理：

高并发下 LongAdder 比 AtomicLong 效率更高，因为LongAdder 引入了分段累加的概念，内部一共有两个参数参与计数：第一个叫作 base，它是一个变量，第二个是 Cell[]，是一个数组。竞争不激烈的情况下，可以直接把累加结果改到 base 变量上。

竞争激烈的时候，LongAdder 会把不同线程计算出 hash 值来对应到不同的 Cell 上进行修改，降低了冲突的概率，然后 Cell 累计求和，并加上 base，形成最终的总和。代码如下：

```
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;
}
```

如何选择LongAdder 和AtomicLong：

累加和减操作情况下选择LongAdder，因为LongAdder 吞吐量要大得多，大约是 AtomicLong 的十倍。

利用 CAS 比如 compareAndSet 等操作的话，就需要使用 AtomicLong 来完成。（LongAdder 没有这些操作）

Accumulator 其实就是一个更通用版本的 Adder，LongAdder 的 API 只有对数值的加减，而 LongAccumulator 提供了自定义的函数操作。

举例：

```java
public class LongAccumulatorDemo {

    public static void main(String[] args) throws InterruptedException {
        LongAccumulator accumulator = new LongAccumulator((x, y) -> x + y, 0);
        ExecutorService executor = Executors.newFixedThreadPool(8);
        IntStream.range(1, 10).forEach(i -> executor.submit(() -
> accumulator.accumulate(i)));

        Thread.sleep(2000);
        System.out.println(accumulator.getThenReset());
    }
}
```

这段代码使用LongAccumulator实现 0+1+2+3+...+8+9=45 计算。

Accumulator还可以实现下面的功能；

```java
LongAccumulator counter = new LongAccumulator((x, y) -> x + y, 0);
LongAccumulator result = new LongAccumulator((x, y) -> x * y, 0);
LongAccumulator min = new LongAccumulator((x, y) -> Math.min(x, y), 0);
LongAccumulator max = new LongAccumulator((x, y) -> Math.max(x, y), 0);
```

这时你可能会有一个疑问：在这里为什么不用 for 循环呢？比如说我们之前的例子，从 0 加到 9，我们直接写一个 for 循环不就可以了吗？

答：因为for是串行的，而LongAccumulator 一大优势就是**可以利用线程池**来为它工作，多个线程之间是可以并行计算的，效率要比之前的串行高得多，但是加的顺序是不固定的，最终的结果是确定的。

LongAccumulator 的适用场景：

1. 需要大量的计算，并且当需要并行计算的时候，我们可以考虑使用 LongAccumulator。如果计算量大，需要提高计算的效率时，我们则可以利用线程池，再加上 LongAccumulator 来配合的话，就可以达到并行计算的效果，效率非常高。

   > 计算量不大，或者串行计算就可以满足需求的时候，可以使用 for 循环

2. 计算的执行顺序不关键，只要求最终的结果正确的可以用Accumulator 。