> 二狗：为什么要使用线程池？直接new个线程不是很舒服？

如果我们在方法中直接new一个线程来处理，当这个方法被调用频繁时就会创建很多线程，不仅会消耗系统资源，还会降低系统的稳定性，一不小心**把系统搞崩了**，就可以**直接去财务那结帐**了。

如果我们合理的使用线程池，则可以避免把系统搞崩的窘境。总得来说，使用线程池可以带来以下几个好处：

1. 降低资源消耗。通过重复利用已创建的线程，降低线程创建和销毁造成的消耗。
2. 提高响应速度。当任务到达时，任务可以不需要等到线程创建就能立即执行。
3. 增加线程的可管理型。线程是稀缺资源，使用线程池可以进行统一分配，调优和监控。

> 二狗：线程池的核心属性有哪些？

threadFactory（线程工厂）：用于创建工作线程的工厂。

corePoolSize（核心线程数）：当线程池运行的线程少于 corePoolSize 时，将创建一个新线程来处理请求，即使其他工作线程处于空闲状态。

workQueue（队列）：用于保留任务并移交给工作线程的阻塞队列。

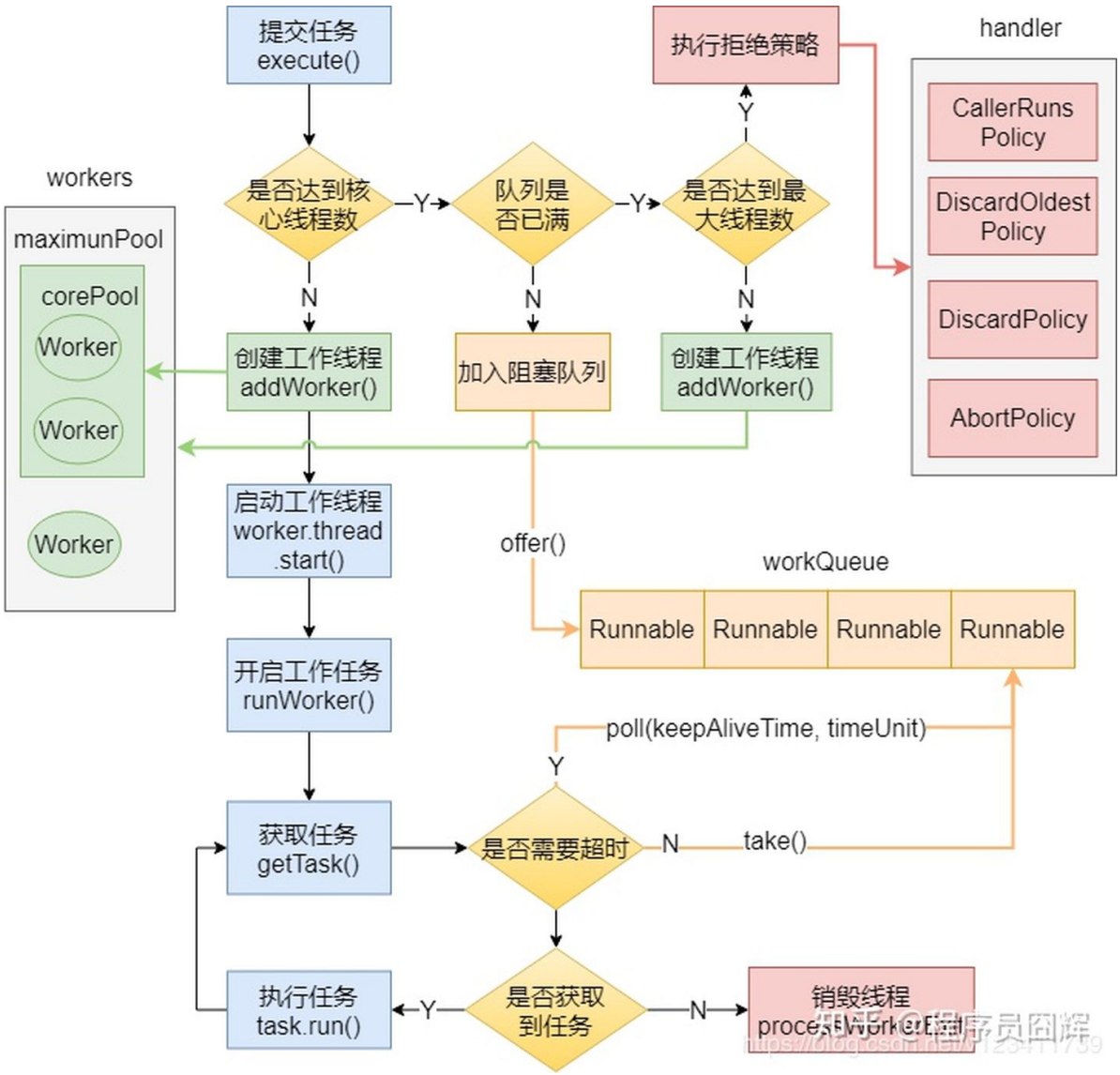
maximumPoolSize（最大线程数）：线程池允许开启的最大线程数。

handler（拒绝策略）：往线程池添加任务时，将在下面两种情况触发拒绝策略：1）线程池运行状态不是 RUNNING；2）线程池已经达到最大线程数，并且阻塞队列已满时。

keepAliveTime（保持存活时间）：如果线程池当前线程数超过 corePoolSize，则多余的线程空闲时间超过 keepAliveTime 时会被终止。

> 二狗：说下线程池的运作流程

我给你画张图吧。



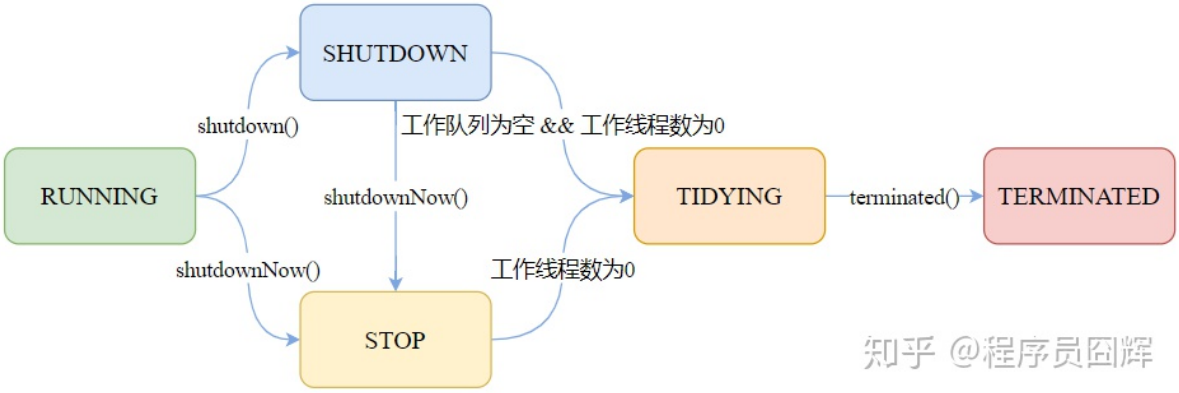
> 二狗：（尼玛，这图也太香了，收藏收藏）线程池中的各个状态分别代表什么含义？

线程池目前有5个状态：

- RUNNING：接受新任务并处理排队的任务。
- SHUTDOWN：不接受新任务，但处理排队的任务。
- STOP：不接受新任务，不处理排队的任务，并中断正在进行的任务。
- TIDYING：所有任务都已终止，workerCount 为零，线程转换到 TIDYING 状态将运行 terminated() 钩子方法。
- TERMINATED：terminated() 已完成。

> 二狗：这几个状态之间是怎么流转的？

我再给你画个图，看好了！



> 二狗：（这图也不错，收藏就对了）线程池有哪些队列？

常见的阻塞队列有以下几种：

ArrayBlockingQueue：基于数组结构的有界阻塞队列，按先进先出对元素进行排序。

LinkedBlockingQueue：基于链表结构的有界/无界阻塞队列，按先进先出对元素进行排序，吞吐量通常高于 ArrayBlockingQueue。Executors.newFixedThreadPool 使用了该队列。

SynchronousQueue：不是一个真正的队列，而是一种在线程之间移交的机制。要将一个元素放入 SynchronousQueue 中，必须有另一个线程正在等待接受这个元素。如果没有线程等待，并且线程池的当前大小小于最大值，那么线程池将创建一个线程，否则根据拒绝策略，这个任务将被拒绝。使用直接移交将更高效，因为任务会直接移交给执行它的线程，而不是被放在队列中，然后由工作线程从队列中提取任务。只有当线程池是无界的或者可以拒绝任务时，该队列才有实际价值。Executors.newCachedThreadPool使用了该队列。

PriorityBlockingQueue：具有优先级的无界队列，按优先级对元素进行排序。元素的优先级是通过自然顺序或 Comparator 来定义的。

> 二狗：使用队列有什么需要注意的吗？

使用有界队列时，需要注意线程池满了后，被拒绝的任务如何处理。

使用无界队列时，需要注意如果任务的提交速度大于线程池的处理速度，可能会导致内存溢出。

> 二狗：线程池有哪些拒绝策略？

常见的有以下几种：

AbortPolicy：中止策略。默认的拒绝策略，直接抛出 RejectedExecutionException。调用者可以捕获这个异常，然后根据需求编写自己的处理代码。

DiscardPolicy：抛弃策略。什么都不做，直接抛弃被拒绝的任务。

DiscardOldestPolicy：抛弃最老策略。抛弃阻塞队列中最老的任务，相当于就是队列中下一个将要被执行的任务，然后重新提交被拒绝的任务。如果阻塞队列是一个优先队列，那么“抛弃最旧的”策略将导致抛弃优先级最高的任务，因此最好不要将该策略和优先级队列放在一起使用。

CallerRunsPolicy：调用者运行策略。在调用者线程中执行该任务。该策略实现了一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将任务回退到调用者（调用线程池执行任务的主线程），由于执行任务需要一定时间，因此主线程至少在一段时间内不能提交任务，从而使得线程池有时间来处理完正在执行的任务。

> 二狗：线程只能在任务到达时才启动吗？

默认情况下，即使是核心线程也只能在新任务到达时才创建和启动。但是我们可以使用 prestartCoreThread（启动一个核心线程）或 prestartAllCoreThreads（启动全部核心线程）方法来提前启动核心线程。

> 二狗：核心线程怎么实现一直存活？

阻塞队列方法有四种形式，它们以不同的方式处理操作，如下表。

	抛出异常	返回特殊值	一直阻塞	超时退出
插入	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除	remove()	poll()	take()	poll(time,unit)
检查	element()	peek()	不可用	不可用

核心线程在获取任务时，通过阻塞队列的 take() 方法实现的一直阻塞（存活）。

> 二狗：非核心线程如何实现在 keepAliveTime 后死亡？

原理同上，也是利用阻塞队列的方法，在获取任务时通过阻塞队列的 poll(time,unit) 方法实现的在延迟死亡。

> 二狗：非核心线程能成为核心线程吗？

虽然我们一直讲着核心线程和非核心线程，但是其实线程池内部是不区分核心线程和非核心线程的。只是根据当前线程池的工作线程数来进行调整，因此看起来像是有核心线程于非核心线程。

> 二狗：如何终止线程池？

终止线程池主要有两种方式：

shutdown：“温柔”的关闭线程池。不接受新任务，但是在关闭前会将之前提交的任务处理完毕。

shutdownNow：“粗暴”的关闭线程池，也就是直接关闭线程池，通过 Thread#interrupt() 方法终止所有线程，不会等待之前提交的任务执行完毕。但是会返回队列中未处理的任务。

> 二狗：（粗暴？实在是太刺激了，不过我喜欢）那我再问问你，Executors 提供了哪些创建线程池的方法？

newFixedThreadPool：固定线程数的线程池。corePoolSize = maximumPoolSize，keepAliveTime为0，工作队列使用无界的LinkedBlockingQueue。适用于为了满足资源管理的需求，而需要限制当前线程数量的场景，适用于负载比较重的服务器。

newSingleThreadExecutor：只有一个线程的线程池。corePoolSize = maximumPoolSize = 1，keepAliveTime为0， 工作队列使用无界的LinkedBlockingQueue。适用于需要保证顺序的执行各个任务的场景。

newCachedThreadPool： 按需要创建新线程的线程池。核心线程数为0，最大线程数为Integer.MAX_VALUE，keepAliveTime为60秒，工作队列使用同步移交 SynchronousQueue。该线程池可以无限扩展，当需求增加时，可以添加新的线程，而当需求降低时会自动回收空闲线程。适用于执行很多的短期异步任务，或者是负载较轻的服务器。

newScheduledThreadPool：创建一个以延迟或定时的方式来执行任务的线程池，工作队列为DelayedWorkQueue。适用于需要多个后台线程执行周期任务。

newWorkStealingPool：JDK 1.8 新增，用于创建一个可以窃取的线程池，底层使用 ForkJoinPool 实现。

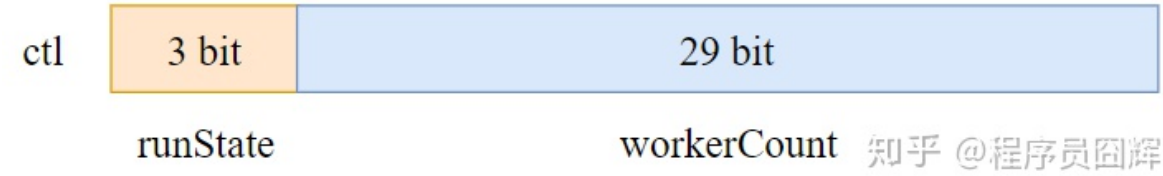
> 二狗：线程池里有个 ctl，你知道它是如何设计的吗?

ctl 是一个打包两个概念字段的原子整数。

1）workerCount：指示线程的有效数量；

2）runState：指示线程池的运行状态，有 RUNNING、SHUTDOWN、STOP、TIDYING、TERMINATED 等状态。

int 类型有32位，其中 ctl 的低29为用于表示 workerCount，高3位用于表示 runState，如下图所示。



例如，当我们的线程池运行状态为 RUNNING，工作线程个数为3，则此时 ctl 的原码为：1010 0000 0000 0000 0000 0000 0000 0011

> 二狗：（这小子看来由偷偷准备了，看来得拿出压箱底的题目了）ctl 为什么这么设计？有什么好处吗?

个人认为，ctl 这么设计的主要好处是将对 runState 和 workerCount 的操作封装成了一个原子操作。

runState 和 workerCount 是线程池正常运转中的2个最重要属性，线程池在某一时刻该做什么操作，取决于这2个属性的值。

因此无论是查询还是修改，我们必须保证对这2个属性的操作是属于“同一时刻”的，也就是原子操作，否则就会出现错乱的情况。如果我们使用2个变量来分别存储，要保证原子性则需要额外进行加锁操作，这显然会带来额外的开销，而将这2个变量封装成1个 AtomicInteger 则不会带来额外的加锁开销，而且只需使用简单的位操作就能分别得到 runState 和 workerCount。

由于这个设计，workerCount 的上限 CAPACITY = (1 << 29) - 1，对应的二进制原码为：0001 1111 1111 1111 1111 1111 1111 1111（不用数了，29个1）。

通过 ctl 得到 runState，只需通过位操作：ctl & ~CAPACITY。

~（按位取反），于是“~CAPACITY”的值为：1110 0000 0000 0000 0000 0000 0000 0000，只有高3位为1，与 ctl 进行 & 操作，结果为 ctl 高3位的值，也就是 runState。

通过 ctl 得到 workerCount 则更简单了，只需通过位操作：c & CAPACITY。

> 二狗：小伙子不错不错，那我最后问一个，在我们实际使用中，线程池的大小配置多少合适?

要想合理的配置线程池大小，首先我们需要区分任务是计算密集型还是I/O密集型。

对于计算密集型，设置 线程数 = CPU数 + 1，通常能实现最优的利用率。

对于I/O密集型，网上常见的说法是设置 线程数 = CPU数 * 2 ，这个做法是可以的，但个人觉得不是最优的。

在我们日常的开发中，我们的任务几乎是离不开I/O的，常见的网络I/O（RPC调用）、磁盘I/O（数据库操作），并且I/O的等待时间通常会占整个任务处理时间的很大一部分，在这种情况下，开启更多的线程可以让 CPU 得到更充分的使用，一个较合理的计算公式如下：

线程数 = CPU数 * CPU利用率 * (任务等待时间 / 任务计算时间 + 1)

例如我们有个定时任务，部署在4核的服务器上，该任务有100ms在计算，900ms在I/O等待，则线程数约为：4 * 1 * (1 + 900 / 100) = 40个。

当然，具体我们还要结合实际的使用场景来考虑。

概述

CAS（Compare-and-Swap），即比较并替换，是一种实现并发算法时常用到的技术，Java并发包中的很多类都使用了CAS技术。CAS也是现在面试经常问的问题，本文将深入的介绍CAS的原理。

案例

介绍CAS之前，我们先来看一个例子。

```
/**
 * @author joonwhee
 * @date 2019/7/6
 */
public class VolatileTest {

    public static volatile int race = 0;

    private static final int THREADS_COUNT = 20;

    public static void increase() {
        race++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++) {
```

```
                        increase();
                    }
                }
            });
            threads[i].start();
        }

        while (Thread.activeCount() > 1) {
            Thread.yield();
        }
        System.out.println(race);
    }
}
```

这个例子有些网友反馈会进入死循环，我后面也发现了，在IDEA的RUN模式下确实会陷入死循环，通过Thread.currentThread().getThreadGroup().list(); 代码可以打印出当前的线程情况如下：

```
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
    Thread[Monitor Ctrl-Break,5,main]
```

可以看到，除了Main方法线程后，还有一个Monitor Ctrl-Break线程，这个线程是IDEA用来监控Ctrl-Break中断信号的线程。

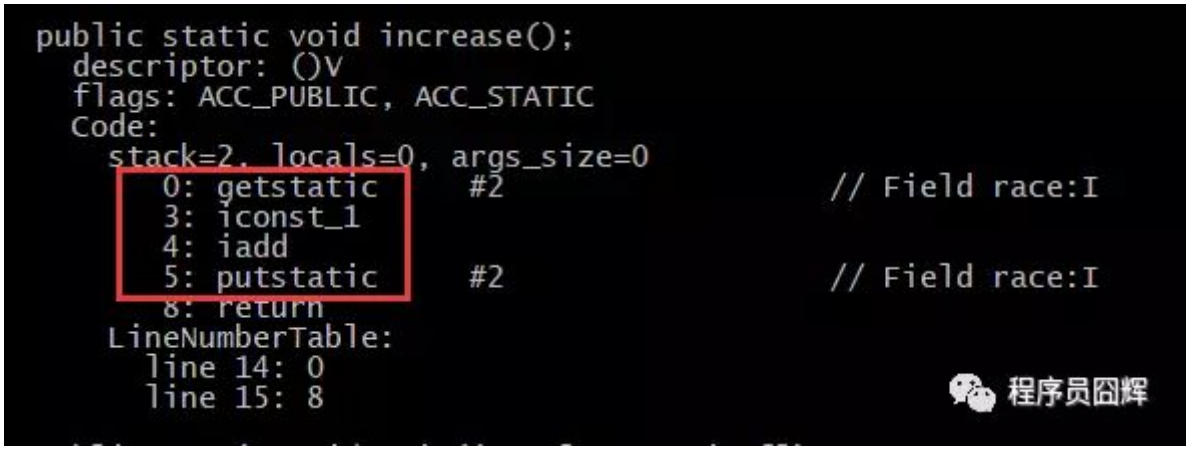
解决死循环的办法：如果是IDEA，可以使用DEBUG模式运行就可以，或者使用下面这段代码。

```
import java.util.concurrent.CountDownLatch;

/**
 * @author joonwhee
 * @date 2019/7/6
 */
public class VolatileTest {

    public static volatile int race = 0;

    private static final int THREADS_COUNT = 20;

    private static CountDownLatch countDownLatch = new
CountDownLatch(THREADS_COUNT);

    public static void increase() {
        race++;
    }

    public static void main(String[] args) throws InterruptedException {
        Thread[] threads = new Thread[THREADS_COUNT];
        for (int i = 0; i < THREADS_COUNT; i++) {
            threads[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int i = 0; i < 10000; i++) {
                        increase();
                    }
                    countDownLatch.countDown();
                }
```

```
            });
            threads[i].start();
        }
        countDownLatch.await();
        System.out.println(race);
    }
}
```

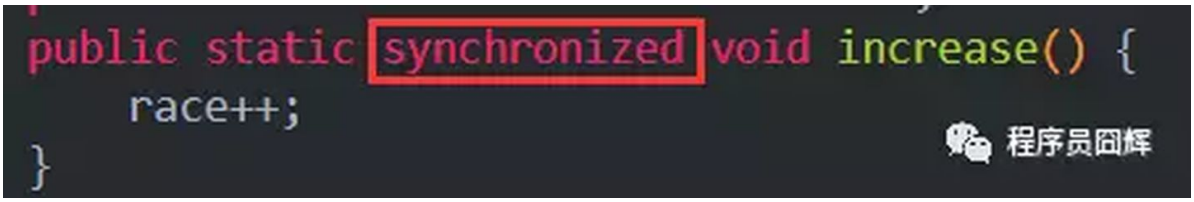
上面这个例子在volatile关键字详解文中用过，我们知道，运行完这段代码之后，并不会获得期望的结果，而且会发现每次运行程序，输出的结果都不一样，都是一个小于200000的数字。

通过分析字节码我们知道，这是因为volatile只能保证可见性，无法保证原子性，而自增操作并不是一个原子操作（如下图所示），在并发的情况下，putstatic指令可能把较小的race值同步回主内存之中，导致我们每次都无法获得想要的结果。那么，应该怎么解决这个问题了？



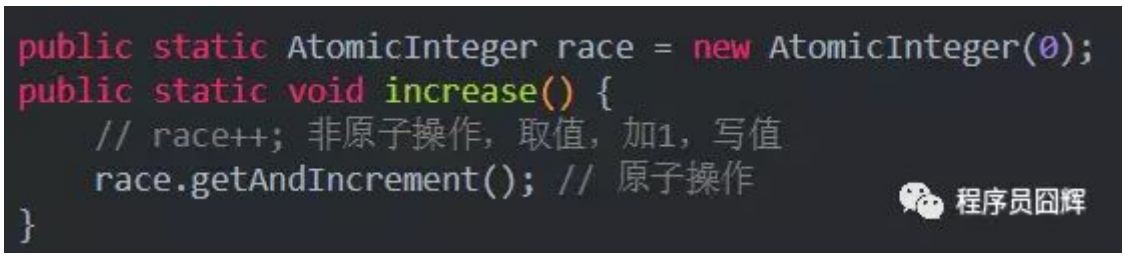
解决方法：

首先我们想到的是用synchronized来修饰increase方法。



使用synchronized修饰后，increase方法变成了一个原子操作，因此是肯定能得到正确的结果。但是，我们知道，每次自增都进行加锁，性能可能会稍微差了点，有更好的方案吗？

答案当然是有的，这个时候我们可以使用Java并发包原子操作类（Atomic开头），例如以下代码。



我们将例子中的代码稍做修改：race改成使用AtomicInteger定义，“race++”改成使用“race.getAndIncrement()”，AtomicInteger.getAndIncrement()是原子操作，因此我们可以确保每次都可以获得正确的结果，并且在性能上有不错的提升（针对本例子，在JDK1.8.0_151下运行）。

通过方法调用，我们可以发现，getAndIncrement方法调用getAndAddInt方法，最后调用的是compareAndSwapInt方法，即本文的主角CAS，接下来我们开始介绍CAS。

```
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}
```



```
/**
 * Atomically adds the given value to the current value of a field
 * or array element within the given object <code>o</code>
 * at the given <code>offset</code>.
 *
 * @param o object/array to update the field/element in
 * @param offset field/element offset
 * @param delta the value to add
 * @return the previous value
 * @since 1.8
 */
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);  // 拿到内存位置的最新值
    } while (!compareAndSwapInt(o, offset, v, v + delta));  // CAS修改成功才跳出循环
    return v;
}
```


getAndAddInt方法解析：拿到内存位置的最新值v，使用CAS尝试修将内存位置的值修改为目标值v+delta，如果修改失败，则获取该内存位置的新值v，然后继续尝试，直至修改成功。

CAS是什么?

CAS是英文单词**CompareAndSwap**的缩写，中文意思是：比较并替换。CAS需要有3个操作数：内存地址V，旧的预期值A，即将要更新的目标值B。

CAS指令执行时，当且仅当内存地址V的值与预期值A相等时，将内存地址V的值修改为B，否则就什么都不做。整个比较并替换的操作是一个原子操作。

源码分析

上面源码分析时，提到最后调用了compareAndSwapInt方法，接着继续深入探讨该方法，该方法在Unsafe中对应的源码如下。

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env,
        jobject unsafe, jobject obj, jlong offset, jint e, jint x))  // CAS
  UnsafeWrapper("Unsafe_CompareAndSwapInt");
  oop p = JNIHandles::resolve(obj);
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END
```


可以看到调用了“Atomic::cmpxchg”方法，“Atomic::cmpxchg”方法在linux_x86和windows_x86的实现如下。

linux_x86的实现：

```
// linux_x86
inline jint     Atomic::cmpxchg    (jint     exchange_value,
          volatile jint*     dest, jint     compare_value) {
  int mp = os::is_MP();
  __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
                    : "=a" (exchange_value)
                    : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
                    : "cc", "memory");
  return exchange_value;
}
```


windows_x86的实现：

```
215
216   // windows_x86
217   inline jint     Atomic::cmpxchg    (jint     exchange_value,
218                   volatile jint*     dest, jint     compare_value) {
219     // alternative for InterlockedCompareExchange
220     int mp = os::is_MP();
221     __asm {
222       mov edx, dest
223       mov ecx, exchange_value
224       mov eax, compare_value
225       LOCK_IF_MP(mp)
226       cmpxchg dword ptr [edx], ecx
227     }
228   }
```


Atomic::cmpxchg方法解析：

mp是“os::is_MP()”的返回结果，“os::is_MP()”是一个内联函数，用来判断当前系统是否为多处理器。

1. 如果当前系统是多处理器，该函数返回1。
2. 否则，返回0。

LOCK_IF_MP(mp)会根据mp的值来决定是否为cmpxchg指令添加lock前缀。

1. 如果通过mp判断当前系统是多处理器（即mp值为1），则为cmpxchg指令添加lock前缀。
2. 否则，不加lock前缀。

这是一种优化手段，认为单处理器的环境没有必要添加lock前缀，只有在多核情况下才会添加lock前缀，因为lock会导致性能下降。cmpxchg是汇编指令，作用是比较并交换操作数。

intel手册对lock前缀的说明如下：

1. 确保对内存的读-改-写操作原子执行。在Pentium及Pentium之前的处理器中，带有lock前缀的指令在执行期间会锁住总线，使得其他处理器暂时无法通过总线访问内存。很显然，这会带来昂贵的开销。从Pentium 4，Intel Xeon及P6处理器开始，intel在原有总线锁的基础上做了一个很有意义的优化：如果要访问的内存区域（area of memory）在lock前缀指令执行期间已经在处理器内部的缓存中被锁定（即包含该内存区域的缓存行当前处于独占或以修改状态），并且该内存区域被完全包含在单个缓存行（cache line）中，那么处理器将直接执行该指令。由于在指令执行期间该缓存行会一直被锁定，其它处理器无法读/写该指令要访问的内存区域，因此能保证指令执行的原子性。这个操作过程叫做缓存锁定（cache locking），缓存锁定将大大降低lock前缀指令的执行开销，但是当多处理器之间的竞争程度很高或者指令访问的内存地址未对齐时，仍然会锁住总线。
2. 禁止该指令与之前和之后的读和写指令重排序。
3. 把写缓冲区中的所有数据刷新到内存中。

上面的第1点保证了CAS操作是一个原子操作，第2点和第3点所具有的内存屏障效果，保证了CAS同时具有volatile读和volatile写的内存语义。

CAS的缺点：

CAS虽然很高效的解决了原子操作问题，但是CAS仍然存在三大问题。

1. 循环时间长开销很大。
2. 只能保证一个共享变量的原子操作。
3. ABA问题。

循环时间长开销很大：我们可以看到getAndAddInt方法执行时，如果CAS失败，会一直进行尝试。如果CAS长时间一直不成功，可能会给CPU带来很大的开销。

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来保证原子性。

什么是ABA问题？ABA问题怎么解决？

CAS 的使用流程通常如下：1）首先从地址 V 读取值 A；2）根据 A 计算目标值 B；3）通过 CAS 以原子的方式将地址 V 中的值从 A 修改为 B。

但是在第1步中读取的值是A，并且在第3步修改成功了，我们就能说它的值在第1步和第3步之间没有被其他线程改变过了吗？

如果在这段期间它的值曾经被改成了B，后来又被改回为A，那CAS操作就会误认为它从来没有被改变过。这个漏洞称为CAS操作的“ABA”问题。Java并发包为了解决这个问题，提供了一个带有标记的原子引用类“AtomicStampedReference”，它可以通过控制变量值的版本来保证CAS的正确性。因此，在使用CAS前要考虑清楚“ABA”问题是否会影响程序并发的正确性，如果需要解决ABA问题，改用传统的互斥同步可能会比原子类更高效。