链滴

无锁算法：CAS 原理

作者：zhanwh9

原文链接：https://ld246.com/article/1543212684107

来源网站：链滴

在 Java 8 之后版本的 ConcurrentHashMap 中，在一些特定场景会采用 CAS 来进行无锁并发操作。

乐观锁和悲观锁

独占锁是一种悲观锁，synchronized 就是一种独占锁，它假设最坏的情况，并且只有在确保其他线不会造成干扰的情况下执行，会导致其他所有需要锁的线程挂起，等待持有锁的线程释放锁。另一个加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作。如果因冲突而失败就重试，直到成功为止。

锁的代价

锁是用来做并发最简单的方式，当然其代价也是最高的。内核态的锁的时候需要操作系统进行一次上文切换，加锁、释放锁会导致比较多的上下文切换和调度延时，等待锁的线程会被挂起直至锁释放。上下文切换的时候，CPU 之前缓存的指令和数据都将失效，对性能有很大的损失。操作系统对多线程锁进行判断就像两姐妹在为同一个玩具而争吵，然后操作系统就是能决定他们谁能拿到玩具的父母，是很慢的。用户态的锁虽然避免了这些问题，但是其实他们只是在没有真实的竞争时才有效。

Java 在 JDK 1.5 之前都是靠 synchronized 关键字保持同步的，这种通过使用一致的锁定协议来协调共享状态的访问，可以确保无论哪个线程持有守护变量的锁，都采用独占的方式来访问这些变量，如出现多个线程同时访问锁，那么一些线程将会被挂起，当线程恢复执行时，必须等待其他线程执行完们的时间片以后才能被调度执行，在挂起和恢复执行过程中存在很大的开销。锁还存在着其他一些缺，当一个线程正在等待锁时，它不能做任何事。如果一个线程在持有锁的情况下被延迟执行，那么所需要这个锁的线程都无法执行下去。如果被阻塞的线程优先级高，而持有锁的线程优先级低，将会导优先级反转（Priority Inversion）。

总的来说，使用锁来实现线程同步的缺点如下：

- 产生竞争时，线程被阻塞等待，无法做到线程实时响应。
- dead lock，死锁
- live lock。
- 优先级反转。
- 使用不当，造成性能下降。

无锁算法

既然使用锁来进行同步会有上述的代价，因此，无锁算法应运而生。无锁编程，即不用锁的情况下实多线程之间的变量同步，也就是说在没有线程被阻塞的情况下实现变量的同步，所以也叫非阻塞同步Non-blocking Synchronization）。实现非阻塞同步的方案称为“无锁编程算法”（Non-blocking a gorithm）。

当然在部分情况下，目前来看，无锁编程并不能替代 lock。

CAS（比较与交换，Compare And Swap）是一种有名的无锁算法。CAS 的语义是“我认为 V 的值该为 A，如果是，那么将 V 的值更新为 B，否则不修改并告诉 V 的值实际为多少”。CAS 是项乐观技术，当多个线程尝试使用 CAS 同时更新同一个变量时，只有其中一个线程能更新变量的值，而其线程都失败，失败的线程不会被挂起，而是被告知这次竞争中失败，并可以再次尝试。CAS 有 3 个作，内存值 V，旧的预期值 A，要修改的新值 B。当且仅当预期值 A 和内存 V 相同时，将内存值 V 改为 B，否则什么都不做。

java.util.concurrent.atomic 中的 AtomicXXX，都使用了这些底层的 JVM 支持作为数字类型的引用

型提供一种高效的 CAS 操作，而在 java.util.concurrent 中的大多数类在实现时都直接或间接的使用这些原子变量类，这些原子变量都调用了 sun.misc.Unsafe 库里面的 CAS 算法，用 CPU 指令来实无锁自增。

```
public final int getAndIncrement() {
  for (;;) {
    int current = get();
    int next = current + 1;
    if (compareAndSet(current, next))
      return current;
  }
}
```

因而在大部分情况下，java 中使用 Atomic 包中的 incrementAndGet 的性能比用 synchronized 高几倍。