优秀借鉴

Java实现CAS的原理 | Java程序员进阶之路

美团终面：CAS确定完全不需要锁吗?

1、什么是CAS

CAS是 *Compare-And-Swap*（比较并交换）的缩写，是一种**轻量级的同步机制**，主要用于实现多线程环境下的无锁算法和数据结构，保证了并发安全性。它可以在**不使用锁**（如synchronized、Lock）的情况下，对共享数据进行线程安全的操作。

CAS操作主要有三个参数：要更新的内存位置、期望的值和新值。CAS操作的执行过程如下：

1. 首先，获取要更新的**内存位置的值**，记为var。
2. 然后，将**期望值expected**与**var**进行比较，如果两者相等，则将**内存位置的值var**更新为新值new。
3. 如果两者不相等，则说明有其他线程修改了**内存位置的值var**，此时CAS操作失败，需要重新尝试。

2、原理相关的Unsafe类

前面提到，CAS是一种原子操作。那么Java是怎样来使用CAS的呢？我们知道，在Java中，如果一个方法是native的，那Java就不负责具体实现它，而是交给底层的JVM使用c或者c++去实现。

Unsafe类是JDK提供的一个**不安全的类**，它提供了一些底层的操作，包括内存操作、线程调度、对象实例化等。它的作用是让Java可以在底层**直接操作内存**，从而提高程序的效率。但是，由于Unsafe类是不安全的，所以只有JDK开发人员才能使用它，普通开发者不建议使用。它里面大多是一些 *native* 方法，其中就有几个关于CAS的：

```
boolean compareAndSwapObject(Object o, long offset,Object expected,
Object x);
boolean compareAndSwapInt(Object o, long offset,int expected,int x);
boolean compareAndSwapLong(Object o, long offset,long expected,long
x);
```

具体的实现过程如下：

1. 调用 *compareAndSwapInt* 、 *compareAndSwapLong* 或 *compareAndSwapObject* 方法时，会传入三个参数，分别是需要修改的变量V、期望的值A和新值B。
2. 方法会先读取变量V的当前值，如果当前值等于期望的值A，则使用新值B来更新变量V，否则不做任何操作。
3. 方法会返回更新操作是否成功的标志，如果更新成功，则返回true，否则返回false。

由于CAS操作是基于底层硬件支持的原子性指令来实现的，所以它可以保证操作的**原子性和线程安全性**，同时也可以避免使用锁带来的性能开销。因此，CAS操作广泛应用于并发编程中，比如实现无锁数据结构、实现线程安全的计数器等。

3、原子操作类解析

前面说到Unsafe类中的几个支持CAS的方法，而在Java中便有这么一些类应用到了这些方法，其中一类便是原子操作相关类，在juc包中的atomic包下。

这里我们以 *AtomicInteger* 类的 *getAndAdd(int delta)* 方法为例，来看看Java是如何实现原子操作的：

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

这里的U其实就是一个 *Unsafe* 对象：

```
private static final jdk.internal.misc.Unsafe U =
jdk.internal.misc.Unsafe.getUnsafe();
```

所以其实 *AtomicInteger* 类的 *getAndAdd(int delta)* 方法是调用 *Unsafe* 类的方法来实现的：

```java
@HotSpotIntrinsicCandidate
public final int getAndAddInt(Object o, long offset, int delta) {
    int v;
    do {
        v = getIntVolatile(o, offset);
    } while (!weakCompareAndSetInt(o, offset, v, v + delta));
    return v;
}
```

前面我们讲到，CAS是“无锁”的基础，它允许更新失败。所以经常会与while循环搭配，在失败后不断去重试。这里声明了一个v，也就是要返回的值。从 `getAndAddInt` 来看，它返回的应该是原来的值，而新的值的 `v + delta` 。

这里使用的是**do-while循环**。这种循环不多见，它的目的是**保证循环体内的语句至少会被执行一遍**。这样才能保证return 的值 `v` 是我们期望的值。

```java
public final boolean weakCompareAndSetInt(Object o, long offset, int
expected, int x) {
    return compareAndSetInt(o, offset, expected, x);
}

public final native boolean compareAndSetInt(Object o, long offset,
int expected, int x);
```

可以看到，最终其实是调用的我们之前说到了CAS `native` 方法。那为什么要经过一层 `weakCompareAndSetInt` 呢?

根据深入浅出 Java 多线程中介绍可知，这是与volatile关键字有关的。 `weakCompareAndSet` 操作仅保留了 `volatile` 自身变量的特性，而除去了happens-before规则带来的内存语义。也就是说， `weakCompareAndSet` **无法保证处理操作目标的volatile变量外的其他变量的执行顺序(编译器和处理器为了优化程序性能而对指令序列进行重新排序)，同时也无法保证这些变量的可见性**。这在一定程度上可以提高性能。

4、ABA问题

4.1、何为ABA

ABA问题指在CAS操作过程中，如果变量的值被改为了 A、B、再改回 A，而CAS操作是能够成功的，这时候就可能导致程序出现意外的结果。

在高并发场景下，使用CAS操作可能存在ABA问题，也就是在一个值被修改之前，先被其他线程修改为另外的值，然后再被修改回原值，此时CAS操作会认为这个值没有被修改过，导致数据不一致。

4.2、解决方案

为了解决ABA问题，Java中提供了 `AtomicStampedReference` 类，该类通过使用**版本号**的方式来解决ABA问题。每个共享变量都会关联一个版本号，CAS操作时需要同时检查值和版本号是否匹配。因此，如果共享变量的值被改变了，版本号也会发生变化，即使共享变量被改回原来的值，版本号也不同，因此CAS操作会失败。

下面是一个使用 `AtomicStampedReference` 类解决ABA问题的示例代码：

```
public void test() {
    AtomicStampedReference<Integer> atomicStampedRef = new AtomicStampedReference<>(1, 0);
    // 重现ABA问题
    int oldStamp = atomicStampedRef.getStamp();
    int oldValue = atomicStampedRef.getReference();
    // 将值从 1 改为 2，并使版本号自增
    atomicStampedRef.compareAndSet(oldValue, 2, atomicStampedRef.getStamp(), atomicStampedRef.getStamp() + 1);
    // 将值从 2 改回 1，并使版本号自增
    atomicStampedRef.compareAndSet(2, 1, atomicStampedRef.getStamp(), atomicStampedRef.getStamp() + 1);

    // 使用旧版本号修改时，即使值和旧值一样，但版本号已经发生了变化，导致修改失败
    atomicStampedRef.compareAndSet(2, 1, oldStamp, oldStamp + 1);
}
```

在上面的示例中，通过 `getStamp()` 和 `getReference()` 方法分别获取共享变量的版本号和值，然后使用 `compareAndSet()` 方法进行CAS操作，每次操作都会更新版本号。这样，就可以避免ABA问题。

5、CPU空转

5.1、为什么出现CPU空转

除了ABA问题，CAS操作还可能会受到自旋时间过长的影响，因为如果某个线程一直在自旋等待，会浪费CPU资源。

5.2、解决方案

为了解决上述问题，可以采用自适应自旋锁的方式，即在前几次重试时采用忙等待的方式，后面则使用阻塞等待的方式，避免浪费CPU资源。

```
public class SpinLock {
    private AtomicReference<Thread> owner = new AtomicReference<>();
```

```
    private int count;

    public void lock() {
        Thread currentThread = Thread.currentThread();
        // 已经获取了锁
        if (owner.get() == currentThread) {
            count++;
            return;
        }
        // 自旋等待获取锁
        while (!owner.compareAndSet(null, currentThread)) {
            // 自适应自旋
            if (count < 10) {
                count++;
            } else {
                // 阻塞等待
                LockSupport.park(currentThread);
            }
        }
    }

    public void unlock() {
        Thread currentThread = Thread.currentThread();
        // 当前线程持有锁
        if (owner.get() == currentThread) {
            if (count > 0) {
                count--;
            } else {
                // 释放锁
                owner.compareAndSet(currentThread, null);
                // 唤醒其他线程
                LockSupport.unpark(currentThread);
            }
        }
    }
}
```

这里解释一下上面代码：

在设计时使用了 `AtomicReference` 来保存当前持有锁的线程对象，这样可以保证线程安全。

当一个线程**请求获取锁**时，如果当前线程已经持有锁，则将计数器加1，否则使用CAS操作来获取锁。这样可以避免了使用synchronized关键字或者ReentrantLock等锁的实现机制。

当线程**获取锁失败**时，使用自旋等待的方式，这样可以避免线程进入阻塞状态，避免了线程上下文切换的开销。当重试次数小于10时，使用自旋等待的方式，当重试次数大于10时，则使用阻塞等待的方式。这样可以在多线程环境下保证线程的公平性和效率。

在**释放锁**时，如果计数器大于0，则将计数器减1，否则将锁的拥有者设为null，唤醒其他线程。这样可以确保在有多个线程持有锁的情况下，正确释放锁资源，并唤醒其他等待线程，保证线程的正确性和公平性。

6、应用场景

CAS在多线程并发编程中被广泛应用，它通常用于实现**乐观锁和无锁算法**。以下是CAS的一些应用场景：

1. **线程安全计数器**：由于CAS操作是原子性的，因此CAS可以用来实现一个线程安全的计数器；
2. **队列**：在并发编程中，队列经常用于多线程之间的数据交换。使用CAS可以实现无锁的非阻塞队列（Lock-Free Queue）；
3. **数据库并发控制**：乐观锁就是通过CAS实现的，它可以在数据库并发控制中保证多个事务同时访问同一数据时的一致性；
4. **自旋锁**：自旋锁是一种非阻塞锁，当线程尝试获取锁时，如果锁已经被其他线程占用，则线程不会进入休眠，而是一直在自旋等待锁的释放。自旋锁的实现可以使用CAS操作；
5. **线程池**：在多线程编程中，线程池可以提高线程的使用效率。使用CAS操作可以避免对线程池的加锁，从而提高线程池的并发性能。

7、CAS真的完全没加锁吗？

上面说过，CAS是一个无锁机制的操作，底层是通过Unsafe类使用**native本地方法**进行的CAS操作，但是大家有没有想过这样的问题：**硬件层面CAS又是如何保证原子性的呢？真的完全没加锁吗？**

拿比较常见的x86架构的CPU来说，其实 CAS 操作通常使用 `cmpxchg` 指令实现的。

可是为啥 `cmpxchg` 指令能保证原子性呢？主要是有以下几个方面的保障：

1. `cmpxchg` 指令是一条原子指令。**在 CPU 执行 `cmpxchg` 指令时，处理器会自动锁定总线，防止其他 CPU 访问共享变量，然后执行比较和交换操作，最后释放总线。**
2. `cmpxchg` 指令在执行期间，**CPU 会自动禁止中断**。这样可以确保 CAS 操作的原子性，避免中断或其他干扰对操作的影响。
3. `cmpxchg` 指令是硬件实现的，可以保证其原子性和正确性。CPU 中的硬件电路确保了 cmpxchg 指令的正确执行，以及对共享变量的访问是原子的。

所以，在操作系统层面，CAS还是会加锁的，通过加锁的方式锁定总线，避免其他CPU访问共享变量。