谈一谈并发CAS（Compare and Swap）实现

题目标签

学习时长：20分钟

题目难度：中等

知识点标签：CAS

题目描述

谈一谈并发CAS（Compare and Swap）实现

题目解决

1. 什么是乐观锁与悲观锁?

悲观锁

总是假设最坏的情况，每次读取数据的时候都默认其他线程会更改数据，因此需要进行加锁操作，当其他线程想要访问数据时，都需要阻塞挂起。悲观锁的实现：

- 传统的关系型数据库使用这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁；
- Java里面的同步`synchronized`关键字的实现。

乐观锁

乐观锁，其实就是一种思想，总是认为不会产生并发问题，每次读取数据的时候都认为其他线程不会修改数据，所以不上锁，但是在更新的时候会判断一下在此期间别的线程有没有修改过数据，乐观锁适用于读操作多的场景，这样可以提高程序的吞吐量。实现方式：

- CAS实现：Java中java.util.concurrent.atomic包下面的原子变量使用了乐观锁的一种CAS实现方式，CAS分析看下节。
- 版本号控制：一般是在数据表中加上一个数据版本号version字段，表示数据被修改的次数，当数据被修改时，version值会加一。当线程A要更新数据值时，在读取数据的同时也会读取version值，在提交更新时，若刚才读取到的version值为当前数据库中的version值相等时才更新，否则重试更新操作，直到更新成功

乐观锁适用于读多写少的情况下（多读场景），悲观锁比较适用于写多读少场景

2. 乐观锁的实现方式-CAS（Compare and Swap），CAS（Compare and Swap）实现原理

背景

在jdk1.5之前都是使用`synchronized`关键字保证同步，`synchronized`保证了无论哪个线程持有共享变量的锁，都会采用独占的方式来访问这些变量,导致会存在这些问题：

- 在多线程竞争下，加锁、释放锁会导致较多的上下文切换和调度延时，引起性能问题
- 如果一个线程持有锁，其他的线程就都会挂起，等待持有锁的线程释放锁。
- 如果一个优先级高的线程等待一个优先级低的线程释放锁，会导致优先级倒置，引起性能风险

为了优化悲观锁这些问题，就出现了乐观锁：

假设没有并发冲突，每次不加锁操作同一变量，如果有并发冲突导致失败，则重试直至成功。

CAS（Compare and Swap）原理

CAS 全称是 compare and swap(比较并且交换)，是一种用于在多线程环境下实现同步功能的机制，其也是无锁优化，或者叫自旋，还有自适应自旋。

在jdk中，`CAS`加`volatile`关键字作为实现并发包的基石。没有CAS就不会有并发包，java.util.concurrent中借助了CAS指令实现了一种区别于synchronized的一种乐观锁。

乐观锁的一种典型实现机制（CAS）：

乐观锁主要就是两个步骤：

- 冲突检测
- 数据更新

当多个线程尝试使用CAS同时更新同一个变量时，只有一个线程可以更新变量的值，其他的线程都会失败，失败的线程并不会挂起，而是告知这次竞争中失败了，并可以再次尝试。

在不使用锁的情况下保证线程安全，CAS实现机制中有重要的三个操作数：

- **需要读写的内存位置(V)**
- **预期原值(A)**
- **新值(B)**

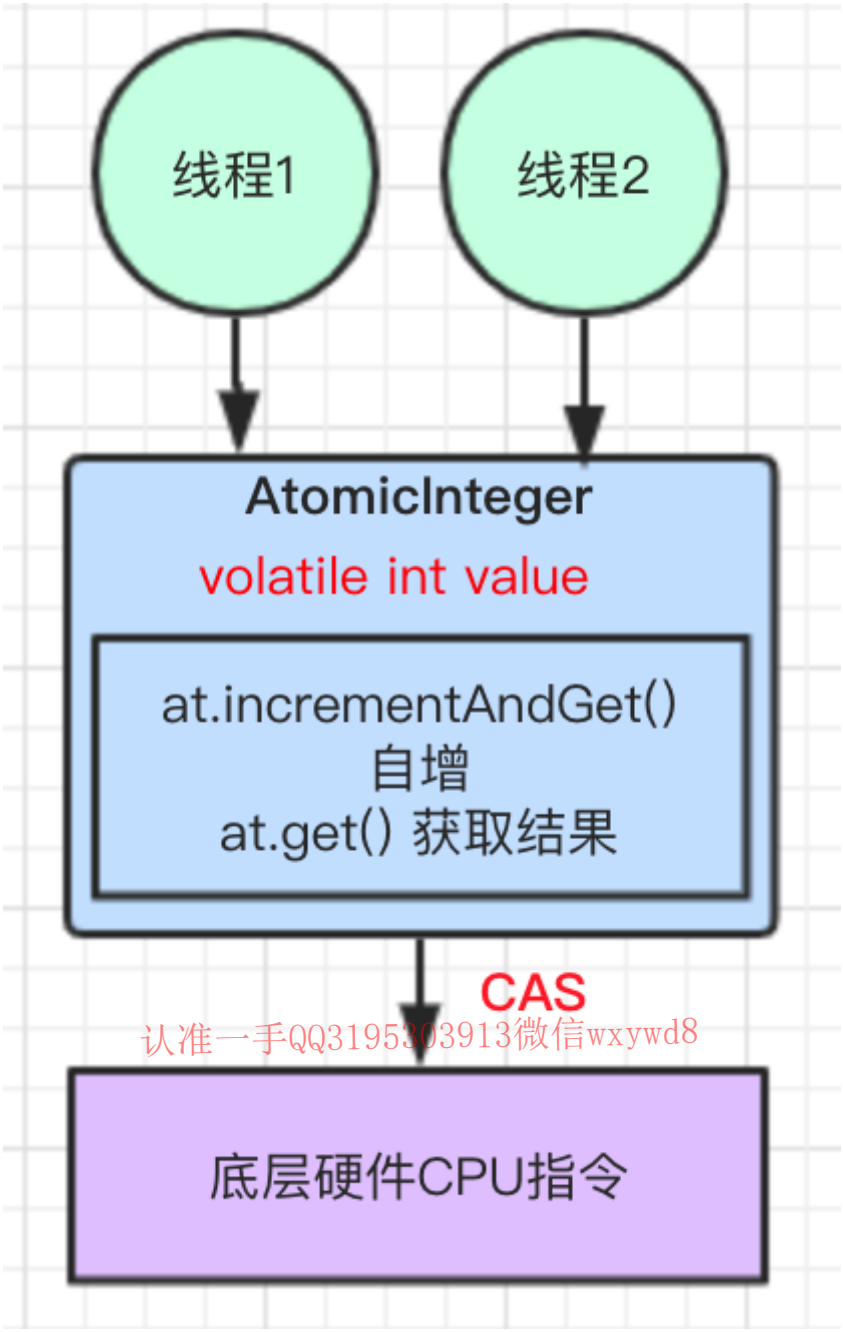
首先先读取需要读写的内存位置(V)，然后比较需要读写的内存位置(V)和预期原值(A)，如果内存位置与预期原值的A相匹配，那么将内存位置的值更新为新值B。如果内存位置与预期原值的值不匹配，那么处理器不会做任何操作。无论哪种情况，它都会在 CAS 指令之前返回该位置的值。具体可以分成三个步骤：

- **读取（需要读写的内存位置(V)）**
- **比较（需要读写的内存位置(V)和预期原值(A)）**
- **写回（新值(B)）**

3. CAS在JDK并发包中的使用

在JDK1.5以上 java.util.concurrent(JUC java并发工具包)是基于CAS算法实现的，相比于synchronized独占锁，堵塞算法，CAS是非堵塞算法的一种常见实现，使用乐观锁JUC在性能上有了很大的提升。

CAS如何在不使用锁的情况下保证线程安全，看并发包中的原子操作类AtomicInteger::getAndIncrement()方法(相当于i++的操作)：




```
// AtomicInteger中
//value的偏移量
private static final long valueOffset;
//获取值
private volatile int value;
//设置value的偏移量
static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }
//增加1
public final int getAndIncrement() {
        return unsafe.getAndAddInt(this, valueOffset, 1);
    }
```

- 首先value必须使用了volatile修饰，这就保证了他的可见性与有序性
- 需要初始化value的偏移量
- unsafe.getAndAddInt通过偏移量进行CAS操作，每次从内存中读取数据然后将数据进行+1操作，然后对原数据，+1后的结果进行CAS操作，成功的话返回结果，否则重试直到成功为止。

```
//unsafe中
public final int getAndAddInt(Object var1, long var2, int var4) {
    int var5;
    do {
        //使用偏移量获取内存中value值
        var5 = this.getIntVolatile(var1, var2);
       //比较并value加+1
    } while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));
    return var5;
}
```

JAVA实现CAS的原理，unsafe::compareAndSwapInt是借助C来调用CPU底层指令实现的。下面是sun.misc.Unsafe::compareAndSwapInt()方法的源代码：

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


4. CAS的缺陷

ABA问题

在多线程场景下CAS会出现ABA问题，例如有2个线程同时对同一个值(初始值为A)进行CAS操作，这三个线程如下

线程1，期望值为A，欲更新的值为B
线程2，期望值为A，欲更新的值为B

线程3，期望值为B，欲更新的值为A

- 线程1抢先获得CPU时间片，而线程2因为其他原因阻塞了，线程1取值与期望的A值比较，发现相等然后将值更新为B，
- 这个时候出现了线程3，线程3取值与期望的值B比较，发现相等则将值更新为A
- 此时线程2从阻塞中恢复，并且获得了CPU时间片，这时候线程2取值与期望的值A比较，发现相等则将值更新为B，虽然线程2也完成了操作，但是线程2并不知道值已经经过了**A->B->A**的变化过程。

ABA问题带来的危害：
小明在提款机，提取了50元，因为提款机问题，有两个线程，同时把余额从100变为50
线程1（提款机）：获取当前值100，期望更新为50，
线程2（提款机）：获取当前值100，期望更新为50，
线程1成功执行，线程2某种原因block了，这时，某人给小明汇款50
线程3（默认）：获取当前值50，期望更新为100，

这时候线程3成功执行，余额变为100，
线程2从Block中恢复，获取到的也是100，compare之后，继续更新余额为50！！！
此时可以看到，实际余额应该为100（100-50+50），但是实际上变为了50（100-50+50-50）这就是ABA问题带来的成功提交。

解决方法

- AtomicStampedReference 带有时间戳的对象引用来解决ABA问题。这个类的compareAndSet方法作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

```
public boolean compareAndSet(
               V      expectedReference,//预期引用
               V      newReference,//更新后的引用
             int    expectedStamp, //预期标志
             int    newStamp //更新后的标志

)
```

- 在变量前面加上版本号，每次变量更新的时候变量的版本号都+1，即A->B->A就变成了1A->2B->3A

循环时间长开销大

自旋CAS（不成功，就一直循环执行，直到成功）如果长时间不成功，会给CPU带来极大的执行开销。

解决方法：

- 限制自旋次数，防止进入死循环
- JVM能支持处理器提供的pause指令那么效率会有一定的提升，

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

解决方法：

- 如果需要对多个共享变量进行操作，可以使用加锁方式(悲观锁)保证原子性，
- 可以把多个共享变量合并成一个共享变量进行CAS操作。