JUC原子类: CAS, Unsafe和原子类详解

JUC中多数类是通过volatile和CAS来实现的，CAS本质上提供的是一种无锁方案，而Synchronized和Lock是互斥锁方案; java原子类本质上使用的是CAS，而CAS底层是通过Unsafe类实现的。

面试问题去理解

- 线程安全的实现方法有哪些?
- 什么是CAS?
- CAS使用示例，结合AtomicInteger给出示例?
- CAS会有哪些问题?
- 针对这这些问题，Java提供了哪几个解决的?
- AtomicInteger底层实现? CAS+volatile
- 请阐述你对Unsafe类的理解?
- 说说你对Java原子类的理解? 包含13个，4组分类，说说作用和使用场景。
- AtomicStampedReference是什么?
- AtomicStampedReference是怎么解决ABA的? 内部使用Pair来存储元素值及其版本号
- java中还有哪些类可以解决ABA的问题? AtomicMarkableReference

CAS

线程安全的实现方法包含:

- 互斥同步: synchronized 和 ReentrantLock
- 非阻塞同步: CAS, AtomicXXXX
- 无同步方案: 栈封闭，Thread Local，可重入代码

什么是*CAS*

CAS的全称为Compare-And-Swap，直译就是对比交换。是一条CPU的原子指令，其作用是让CPU先进行比较两个值是否相等，然后原子地更新某个位置的值，经过调查发现，其实现方式是基于硬件平台的汇编指令，就是说CAS是靠硬件实现的，JVM只是封装了汇编调用，那些AtomicInteger类便是使用了这些封装后的接口。 简单解释：CAS操作需要输入两个数值，一个旧值(期望操作前的值)和一个新值，在操作期间先比较下在旧值有没有发生变化，如果没有发生变化，才交换成新值，发生了变化则不交换。

CAS操作是原子性的，所以多线程并发使用CAS更新数据时，可以不使用锁。JDK中大量使用了CAS来更新数据而防止加锁(synchronized 重量级锁)来保持原子更新。

相信sql大家都熟悉，类似sql中的条件更新一样：update set id=3 from table where id=2。因为单条sql执行具有原子性，如果有多个线程同时执行此sql语句，只有一条能更新成功。

*CAS*使用示例

如果不使用CAS，在高并发下，多线程同时修改一个变量的值我们需要synchronized加锁(可能有人说可以用Lock加锁，Lock底层的AQS也是基于CAS进行获取锁的)。

```
public class Test {
    private int i=0;
    public synchronized int add(){
        return i++;
    }
}
```

java中为我们提供了AtomicInteger 原子类(底层基于CAS进行更新数据的)，不需要加锁就在多线程并发场景下实现数据的一致性。

```
public class Test {
    private  AtomicInteger i = new AtomicInteger(0);
    public int add(){
        return i.addAndGet(1);
    }
}
```

CAS 问题

CAS 方式为乐观锁，synchronized 为悲观锁。因此使用 CAS 解决并发问题通常情况下性能更优。

但使用 CAS 方式也会有几个问题：

ABA问题

因为CAS需要在操作值的时候，检查值有没有发生变化，比如没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时则会发现它的值没有发生变化，但是实际上却变化了。

ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A->B->A就会变成1A->2B->3A。

从Java 1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。这个类的compareAndSet方法的作用是首先检查当前引用是否等于预期引用，并且检查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。

循环时间长开销大

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。如果JVM能支持处理器提供的pause指令，那么效率会有一定的提升。pause指令有两个作用：第一，它可以延迟流水线执行命令(de-pipeline)，使CPU不会消耗过多的执行资源，延迟的时间取决于具体实现的版本，在一些处理器上延迟时间是零；第二，它可以避免在退出循环的时候因内存顺序冲突(Memory Order Violation)而引起CPU流水线被清空(CPU Pipeline Flush)，从而提高CPU的执行效率。

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。

还有一个取巧的办法，就是把多个共享变量合并成一个共享变量来操作。比如，有两个共享变量i = 2，j = a，合并一下ij = 2a，然后用CAS来操作ij。

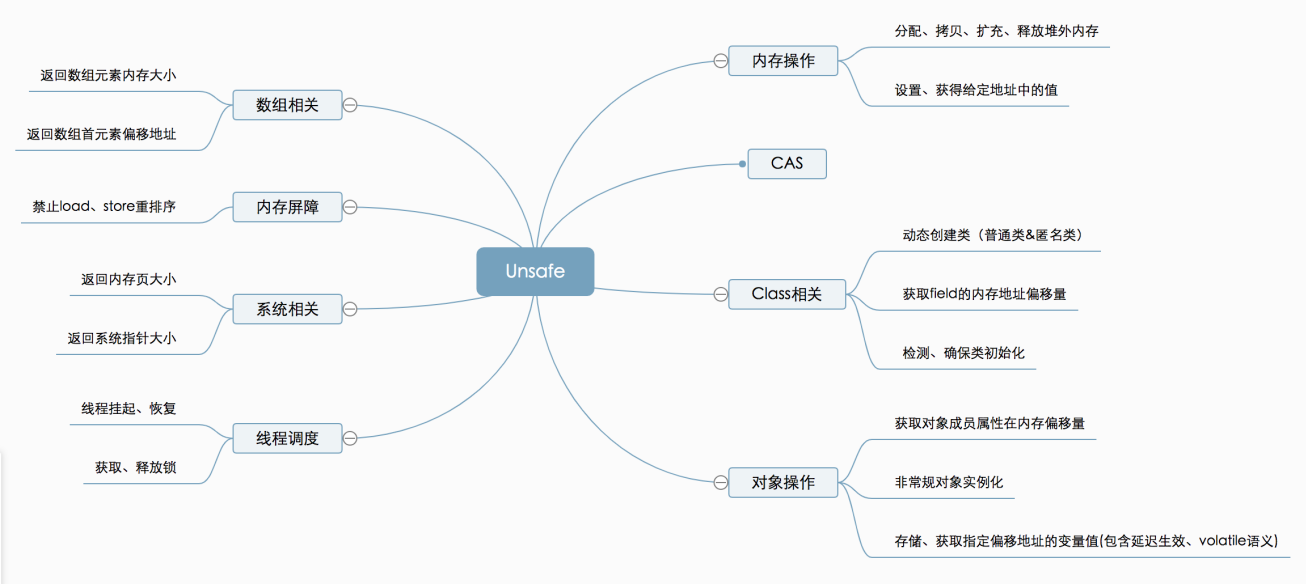
从Java 1.5开始，JDK提供了AtomicReference类来保证引用对象之间的原子性，就可以把多个变量放在一个对象里来进行CAS操作。

UnSafe类详解

Unsafe是位于sun.misc包下的一个类，主要提供一些用于执行低级别、不安全操作的方法，如直接访问系统内存资源、自主管理内存资源等，这些方法在提升Java运行效率、增强Java语言底层资源操作能力方面起到了很大的作用。但由于Unsafe类使Java语言拥有了类似C语言指针一样操作内存空间的能力，这无疑也增加了程序发生相关指针问题的风险。在程序中过度、不正确使用Unsafe类会使得程序出错的概率变大，使得Java这种安全的语言变得不再“安全”，因此对Unsafe的使用一定要慎重。

这个类尽管里面的方法都是 public 的，但是并没有办法使用它们，JDK API 文档也没有提供任何关于这个类的方法的解释。总而言之，对于 Unsafe 类的使用都是受限制的，只有授信的代码才能获得该类的实例，当然 JDK 库里面的类是可以随意使用的。

先来看下这张图，对UnSafe类总体功能：



如上图所示，Unsafe提供的API大致可分为内存操作、CAS、Class相关、对象操作、线程调度、系统信息获取、内存屏障、数组操作等几类，下面将对其相关方法和应用场景进行详细介绍。

Unsafe与CAS

反编译出来的代码：

```
public final int getAndAddInt(Object paramObject, long paramLong, int paramInt)
  {
    int i;
    do
```

```
      i = getIntVolatile(paramObject, paramLong);
    while (!compareAndSwapInt(paramObject, paramLong, i, i + paramInt));
    return i;
  }

  public final long getAndAddLong(Object paramObject, long paramLong1, long paramLong2)
  {
    long l;
    do
      l = getLongVolatile(paramObject, paramLong1);
    while (!compareAndSwapLong(paramObject, paramLong1, l, l + paramLong2));
    return l;
  }

  public final int getAndSetInt(Object paramObject, long paramLong, int paramInt)
  {
    int i;
    do
      i = getIntVolatile(paramObject, paramLong);
    while (!compareAndSwapInt(paramObject, paramLong, i, paramInt));
    return i;
  }

  public final long getAndSetLong(Object paramObject, long paramLong1, long paramLong2)
  {
    long l;
    do
      l = getLongVolatile(paramObject, paramLong1);
    while (!compareAndSwapLong(paramObject, paramLong1, l, paramLong2));
    return l;
  }

  public final Object getAndSetObject(Object paramObject1, long paramLong, Object paramObject2)
  {
    Object localObject;
    do
      localObject = getObjectVolatile(paramObject1, paramLong);
    while (!compareAndSwapObject(paramObject1, paramLong, localObject, paramObject2));
    return localObject;
  }
```

从源码中发现，内部使用自旋的方式进行CAS更新(while循环进行CAS更新，如果更新失败，则循环再次重试)。

又从Unsafe类中发现，原子操作其实只支持下面三个方法。

```
public final native boolean compareAndSwapObject(Object paramObject1, long paramLong, Object
paramObject2, Object paramObject3);

public final native boolean compareAndSwapInt(Object paramObject, long paramLong, int paramInt1,
int paramInt2);

public final native boolean compareAndSwapLong(Object paramObject, long paramLong1, long
paramLong2, long paramLong3);
```

我们发现Unsafe只提供了3种CAS方法：compareAndSwapObject、compareAndSwapInt和compareAndSwapLong。都是native方法。

*Unsafe*底层

不妨再看看Unsafe的compareAndSwap*方法来实现CAS操作，它是一个本地方法，实现位于unsafe.cpp中。

```
UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong
offset, jint e, jint x))
  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
```

可以看到它通过 `Atomic::cmpxchg` 来实现比较和替换操作。其中参数x是即将更新的值，参数e是原内存的值。

如果是Linux的x86，`Atomic::cmpxchg`方法的实现如下：

```
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;
}
```

而windows的x86的实现如下：

```
inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
    int mp = os::isMP(); //判断是否是多处理器
    _asm {
        mov edx, dest
        mov ecx, exchange_value
        mov eax, compare_value
        LOCK_IF_MP(mp)
        cmpxchg dword ptr [edx], ecx
    }
}

// Adding a lock prefix to an instruction on MP machine
// VC++ doesn't like the lock prefix to be on a single line
// so we can't insert a label after the lock prefix.
// By emitting a lock prefix, we can define a label after it.
#define LOCK_IF_MP(mp) __asm cmp mp, 0  \
                       __asm je L0      \
                       __asm _emit 0xF0 \
                       __asm L0:
```

如果是多处理器，为cmpxchg指令添加lock前缀。反之，就省略lock前缀(单处理器会不需要lock前缀提供的内存屏障效果)。这里的lock前缀就是使用了处理器的总线锁(最新的处理器都使用缓存锁代替总线锁来提高性能)。

> cmpxchg(void* ptr, int old, int new)，如果ptr和old的值一样，则把new写到ptr内存，否则返回ptr的值，整个操作是原子的。在Intel平台下，会用lock cmpxchg来实现，使用lock触发缓存锁，这样另一个线程想访问ptr的内存，就会被block住。

*Unsafe*其它功能

Unsafe 提供了硬件级别的操作，比如说获取某个属性在内存中的位置，比如说修改对象的字段值，即使它是私有的。不过 Java 本身就是为了屏蔽底层的差异，对于一般的开发而言也很少会有这样的需求。

举两个例子，比方说：

```
public native long staticFieldOffset(Field paramField);
```

这个方法可以用来获取给定的 paramField 的内存地址偏移量，这个值对于给定的 field 是唯一的且是固定不变的。

再比如说：

```
public native int arrayBaseOffset(Class paramClass);
public native int arrayIndexScale(Class paramClass);
```

前一个方法是用来获取数组第一个元素的偏移地址，后一个方法是用来获取数组的转换因子即数组中元素的增量地址的。

最后看三个方法：

```
public native long allocateMemory(long paramLong);
public native long reallocateMemory(long paramLong1, long paramLong2);
public native void freeMemory(long paramLong);
```

分别用来分配内存，扩充内存和释放内存的。

> 更多相关功能，推荐你看下这篇文章：来自美团技术团队：Java魔法类：Unsafe应用解析 (opens new window)

AtomicInteger

使用举例

以 AtomicInteger 为例，常用 API：

```
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
void lazySet(int newValue): 最终会设置成newValue,使用lazySet设置值后，可能导致其他线程在之后的一小段时间内还是可以读到旧的值。
```

相比 Integer 的优势，多线程中让变量自增：

```
private volatile int count = 0;
// 若要线程安全执行执行 count++，需要加锁
public synchronized void increment() {
    count++;
}
public int getCount() {
    return count;
}
```

使用 AtomicInteger 后：

```
private AtomicInteger count = new AtomicInteger();
public void increment() {
    count.incrementAndGet();
}
// 使用 AtomicInteger 后，不需要加锁，也可以实现线程安全
public int getCount() {
    return count.get();
}
```

源码解析

```
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;
    static {
        try {
            //用于获取value字段相对当前对象的“起始地址”的偏移量
            valueOffset =
unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;

    //返回当前值
    public final int get() {
        return value;
    }

    //递增加detla
    public final int getAndAdd(int delta) {
        //三个参数，1、当前的实例 2、value实例变量的偏移量 3、当前value要加上的数(value+delta)。
        return unsafe.getAndAddInt(this, valueOffset, delta);
    }

    //递增加1
    public final int incrementAndGet() {
        return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
    }
...
}
```

我们可以看到 AtomicInteger 底层用的是volatile的变量和CAS来进行更改数据的。

- volatile保证线程的可见性，多线程并发时，一个线程修改数据，可以保证其它线程立马看到修改后的值
- CAS 保证数据更新的原子性。

延伸到所有原子类：共13个

> JDK中提供了13个原子操作类。

原子更新基本类型

使用原子的方式更新基本类型，Atomic包提供了以下3个类。

- AtomicBoolean: 原子更新布尔类型。
- AtomicInteger: 原子更新整型。
- AtomicLong: 原子更新长整型。

以上3个类提供的方法几乎一模一样，可以参考上面AtomicInteger中的相关方法。

原子更新数组

通过原子的方式更新数组里的某个元素，Atomic包提供了以下的4个类：

- AtomicIntegerArray: 原子更新整型数组里的元素。
- AtomicLongArray: 原子更新长整型数组里的元素。
- AtomicReferenceArray: 原子更新引用类型数组里的元素。 这三个类的最常用的方法是如下两个方法：
- get(int index)：获取索引为index的元素值。
- compareAndSet(int i,E expect,E update): 如果当前值等于预期值，则以原子方式将数组位置i的元素设置为update值。

举个AtomicIntegerArray例子：

```
import java.util.concurrent.atomic.AtomicIntegerArray;

public class Demo5 {
    public static void main(String[] args) throws InterruptedException {
        AtomicIntegerArray array = new AtomicIntegerArray(new int[] { 0, 0 });
        System.out.println(array);
        System.out.println(array.getAndAdd(1, 2));
        System.out.println(array);
    }
}
```

输出结果：

```
[0, 0]
0
[0, 2]
```

原子更新引用类型

Atomic包提供了以下三个类：

- AtomicReference: 原子更新引用类型。
- AtomicStampedReference: 原子更新引用类型, 内部使用Pair来存储元素值及其版本号。
- AtomicMarkableReferce: 原子更新带有标记位的引用类型。

这三个类提供的方法都差不多，首先构造一个引用对象，然后把引用对象set进Atomic类，然后调用compareAndSet等一些方法去进行原子操作，原理都是基于Unsafe实现，但AtomicReferenceFieldUpdater略有不同，更新的字段必须用volatile修饰。

举个AtomicReference例子：

```
import java.util.concurrent.atomic.AtomicReference;

public class AtomicReferenceTest {

    public static void main(String[] args){

        // 创建两个Person对象，它们的id分别是101和102。
        Person p1 = new Person(101);
        Person p2 = new Person(102);
        // 新建AtomicReference对象，初始化它的值为p1对象
        AtomicReference ar = new AtomicReference(p1);
        // 通过CAS设置ar。如果ar的值为p1的话，则将其设置为p2。
        ar.compareAndSet(p1, p2);

        Person p3 = (Person)ar.get();
        System.out.println("p3 is "+p3);
        System.out.println("p3.equals(p1)="+p3.equals(p1));
    }
}

class Person {
    volatile long id;
    public Person(long id) {
        this.id = id;
    }
    public String toString() {
        return "id:"+id;
    }
}
```

结果输出：

```
p3 is id:102
p3.equals(p1)=false
```

结果说明：

- 新建AtomicReference对象ar时，将它初始化为p1。
- 紧接着，通过CAS函数对它进行设置。如果ar的值为p1的话，则将其设置为p2。
- 最后，获取ar对应的对象，并打印结果。p3.equals(p1)的结果为false，这是因为Person并没有覆盖equals()方法，而是采用继承自Object.java的equals()方法；而Object.java中的equals()实际上是调用"=="去比较两个对象，即比较两个对象的地址是否相等。

原子更新字段类

Atomic包提供了四个类进行原子字段更新：

- AtomicIntegerFieldUpdater: 原子更新整型的字段的更新器。
- AtomicLongFieldUpdater: 原子更新长整型字段的更新器。
- AtomicStampedFieldUpdater: 原子更新带有版本号的引用类型。
- AtomicReferenceFieldUpdater: 上面已经说过此处不在赘述。

这四个类的使用方式都差不多，是基于反射的原子更新字段的值。要想原子地更新字段类需要两步:

- 第一步，因为原子更新字段类都是抽象类，每次使用的时候必须使用静态方法newUpdater()创建一个更新器，并且需要设置想要更新的类和属性。
- 第二步，更新类的字段必须使用public volatile修饰。

举个例子：

```
public class TestAtomicIntegerFieldUpdater {

    public static void main(String[] args){
        TestAtomicIntegerFieldUpdater tIA = new TestAtomicIntegerFieldUpdater();
        tIA.doIt();
    }

    public AtomicIntegerFieldUpdater<DataDemo> updater(String name){
        return AtomicIntegerFieldUpdater.newUpdater(DataDemo.class,name);

    }

    public void doIt(){
        DataDemo data = new DataDemo();
        System.out.println("publicVar = "+updater("publicVar").getAndAdd(data, 2));
        /*
            * 由于在DataDemo类中属性value2/value3,在TestAtomicIntegerFieldUpdater中不能访问
            * */
        //System.out.println("protectedVar = "+updater("protectedVar").getAndAdd(data,2));
        //System.out.println("privateVar = "+updater("privateVar").getAndAdd(data,2));

        //System.out.println("staticVar = "+updater("staticVar").getAndIncrement(data));//报
java.lang.IllegalArgumentException
        /*
            * 下面报异常: must be integer
            * */
        //System.out.println("integerVar = "+updater("integerVar").getAndIncrement(data));
        //System.out.println("longVar = "+updater("longVar").getAndIncrement(data));
    }

}

class DataDemo{
    public volatile int publicVar=3;
    protected volatile int protectedVar=4;
    private volatile  int privateVar=5;

    public volatile static int staticVar = 10;
    //public  final int finalVar = 11;

    public volatile Integer integerVar = 19;
    public volatile Long longVar = 18L;

}
```

再说下对于AtomicIntegerFieldUpdater 的使用稍微有一些限制和约束，约束如下：

- 字段必须是volatile类型的，在线程之间共享变量时保证立即可见.eg:volatile int value = 3
- 字段的描述类型(修饰符public/protected/default/private)是与调用者与操作对象字段的关系一致。也就是说调用者能够直接操作对象字段，那么就可以反射进行原子操作。但是对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。
- 只能是实例变量，不能是类变量，也就是说不能加static关键字。

- 只能是可修改变量，不能使final变量，因为final的语义就是不可修改。实际上final的语义和volatile是有冲突的，这两个关键字不能同时存在。
- 对于AtomicIntegerFieldUpdater和AtomicLongFieldUpdater只能修改int/long类型的字段，不能修改其包装类型(Integer/Long)。如果要修改包装类型就需要使用AtomicReferenceFieldUpdater。

再讲讲AtomicStampedReference解决CAS的ABA问题

*AtomicStampedReference*解决*ABA*问题

AtomicStampedReference主要维护包含一个对象引用以及一个可以自动更新的整数"stamp"的pair对象来解决ABA问题。

```
public class AtomicStampedReference<V> {
    private static class Pair<T> {
        final T reference;  //维护对象引用
        final int stamp;  //用于标志版本
        private Pair(T reference, int stamp) {
            this.reference = reference;
            this.stamp = stamp;
        }
        static <T> Pair<T> of(T reference, int stamp) {
            return new Pair<T>(reference, stamp);
        }
    }
    private volatile Pair<V> pair;
    ....

    /**
      * expectedReference : 更新之前的原始值
      * newReference : 将要更新的新值
      * expectedStamp : 期待更新的标志版本
      * newStamp : 将要更新的标志版本
      */
    public boolean compareAndSet(V   expectedReference,
                                 V   newReference,
                                 int expectedStamp,
                                 int newStamp) {
        // 获取当前的(元素值，版本号)对
        Pair<V> current = pair;
        return
            // 引用没变
            expectedReference == current.reference &&
            // 版本号没变
            expectedStamp == current.stamp &&
            // 新引用等于旧引用
            ((newReference == current.reference &&
            // 新版本号等于旧版本号
            newStamp == current.stamp) ||
            // 构造新的Pair对象并CAS更新
            casPair(current, Pair.of(newReference, newStamp)));
    }

    private boolean casPair(Pair<V> cmp, Pair<V> val) {
        // 调用Unsafe的compareAndSwapObject()方法CAS更新pair的引用为新引用
        return UNSAFE.compareAndSwapObject(this, pairOffset, cmp, val);
    }
```

- 如果元素值和版本号都没有变化，并且和新的也相同，返回true；
- 如果元素值和版本号都没有变化，并且和新的不完全相同，就构造一个新的Pair对象并执行CAS更新pair。

可以看到，java中的实现跟我们上面讲的ABA的解决方法是一致的。

- 首先，使用版本号控制；
- 其次，不重复使用节点(Pair)的引用，每次都新建一个新的Pair来作为CAS比较的对象，而不是复用旧的；
- 最后，外部传入元素值及版本号，而不是节点(Pair)的引用。

使用举例

```
private static AtomicStampedReference<Integer> atomicStampedRef =
        new AtomicStampedReference<>(1, 0);
public static void main(String[] args){
    Thread main = new Thread(() -> {
        System.out.println("操作线程" + Thread.currentThread() +",初始值 a = " +
atomicStampedRef.getReference());
        int stamp = atomicStampedRef.getStamp(); //获取当前标识别
        try {
            Thread.sleep(1000); //等待1秒 ，以便让干扰线程执行
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        boolean isCASSuccess = atomicStampedRef.compareAndSet(1,2,stamp,stamp +1);  //此时
expectedReference未发生改变，但是stamp已经被修改了,所以CAS失败
        System.out.println("操作线程" + Thread.currentThread() +",CAS操作结果: " + isCASSuccess);
    },"主操作线程");

    Thread other = new Thread(() -> {
        Thread.yield(); // 确保thread-main 优先执行
atomicStampedRef.compareAndSet(1,2,atomicStampedRef.getStamp(),atomicStampedRef.getStamp() +1);
        System.out.println("操作线程" + Thread.currentThread() +",【increment】 ,值 = "+
atomicStampedRef.getReference());

 atomicStampedRef.compareAndSet(2,1,atomicStampedRef.getStamp(),atomicStampedRef.getStamp() +1);
        System.out.println("操作线程" + Thread.currentThread() +",【decrement】 ,值 = "+
atomicStampedRef.getReference());
    },"干扰线程");

    main.start();
    other.start();
}
```

输出结果：

```
// 输出
> 操作线程Thread[主操作线程,5,main],初始值 a = 2
> 操作线程Thread[干扰线程,5,main],【increment】 ,值 = 2
> 操作线程Thread[干扰线程,5,main],【decrement】 ,值 = 1
> 操作线程Thread[主操作线程,5,main],CAS操作结果: false
```

*java*中还有哪些类可以解决*ABA*的问题*?*

AtomicMarkableReference，它不是维护一个版本号，而是维护一个boolean类型的标记，标记值有修改，了解一下。