

主讲老师: Fox

有道云链接: <http://note.youdao.com/noteshare?>

id=60dbd623e3b9c51e673d2ba45722a1c2&sub=56070C4289CD4693A544AAA0E7FB9F91

什么是 CAS

CAS应用

CAS源码分析

CAS缺陷

ABA问题及其解决方案

什么是ABA问题

ABA问题的解决方案

什么是 CAS

CAS (Compare And Swap, 比较并交换), 通常指的是这样一种原子操作: 针对一个变量, 首先比较它的内存值与某个期望值是否相同, 如果相同, 就给它赋一个新值。

CAS 的逻辑用伪代码描述如下:

```
1 if (value == expectedValue) {  
2     value = newValue;  
3 }
```

以上伪代码描述了一个由比较和赋值两阶段组成的复合操作, **CAS 可以看作是它们合并后的整体——一个不可分割的原子操作, 并且其原子性是直接在硬件层面得到保障的。**

CAS可以看做是乐观锁(对比数据库的悲观、乐观锁)的一种实现方式, Java原子类中的递增操作就通过CAS自旋实现的。

CAS是一种无锁算法, 在不使用锁(没有线程被阻塞)的情况下实现多线程之间的变量同步。

CAS应用

在 Java 中, CAS 操作是由 Unsafe 类提供支持的, 该类定义了三种针对不同类型变量的 CAS 操作, 如图

```
public final native boolean compareAndSwapObject(Object var1, long var2, Object var4, Object var5);  
public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);  
public final native boolean compareAndSwapLong(Object var1, long var2, long var4, long var6);
```

它们都是 native 方法，由 Java 虚拟机提供具体实现，这意味着不同的 Java 虚拟机对它们的实现可能会略有不同。

以 compareAndSwapInt 为例，Unsafe 的 compareAndSwapInt 方法接收 4 个参数，分别是：对象实例、内存偏移量、字段期望值、字段新值。该方法会针对指定对象实例中的相应偏移量的字段执行 CAS 操作。

```
1 public class CASTest {
2
3     public static void main(String[] args) {
4         Entity entity = new Entity();
5
6         Unsafe unsafe = UnsafeFactory.getUnsafe();
7
8         long offset = UnsafeFactory.getFieldOffset(unsafe, Entity.class, "x");
9
10        boolean successful;
11
12        // 4个参数分别是：对象实例、字段的内存偏移量、字段期望值、字段新值
13        successful = unsafe.compareAndSwapInt(entity, offset, 0, 3);
14        System.out.println(successful + "\t" + entity.x);
15
16        successful = unsafe.compareAndSwapInt(entity, offset, 3, 5);
17        System.out.println(successful + "\t" + entity.x);
18
19        successful = unsafe.compareAndSwapInt(entity, offset, 3, 8);
20        System.out.println(successful + "\t" + entity.x);
21    }
22 }
23
24 public class UnsafeFactory {
25
26     /**
27      * 获取 Unsafe 对象
28      * @return
29      */
30     public static Unsafe getUnsafe() {
31         try {
32             Field field = Unsafe.class.getDeclaredField("theUnsafe");
33             field.setAccessible(true);
34             return (Unsafe) field.get(null);
35         } catch (Exception e) {
36             e.printStackTrace();
37         }
38         return null;
39     }
40 }
```

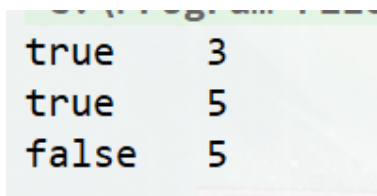
```

39  }
40
41  /**
42   * 获取字段的内存偏移量
43   * @param unsafe
44   * @param clazz
45   * @param fieldName
46   * @return
47   */
48  public static long getFieldOffset(Unsafe unsafe, Class clazz, String fieldName) {
49      try {
50          return unsafe.objectFieldOffset(clazz.getDeclaredField(fieldName));
51      } catch (NoSuchFieldException e) {
52          throw new Error(e);
53      }
54  }
55  }

```

测试

针对 entity.x 的 3 次 CAS 操作，分别试图将它从 0 改成 3、从 3 改成 5、从 3 改成 8。执行结果如下：



```

true      3
true      5
false     5

```

CAS源码分析

Hotspot 虚拟机对 compareAndSwapInt 方法的实现如下：

```

1  #unsafe.cpp
2  UNSAFE_ENTRY(jboolean, Unsafe_CompareAndSwapInt(JNIEnv *env, jobject unsafe, jobject obj, jlong offset, jint e, jint x))
3  UnsafeWrapper("Unsafe_CompareAndSwapInt");
4  oop p = JNIHandles::resolve(obj);
5  // 根据偏移量，计算value的地址
6  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
7  // Atomic::cmpxchg(x, addr, e) cas逻辑 x:要交换的值 e:要比较的值
8  //cas成功，返回期望值e，等于e,此方法返回true
9  //cas失败，返回内存中的value值，不等于e，此方法返回false
10 return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
11 UNSAFE_END

```

核心逻辑在Atomic::cmpxchg方法中，这个根据不同操作系统和不同CPU会有不同的实现。这里我们以linux_64x的为例，查看Atomic::cmpxchg的实现

```
1 #atomic_linux_x86.inline.hpp
2 inline jint Atomic::cmpxchg (jint exchange_value, volatile jint* dest, jint compare_value) {
3     //判断当前执行环境是否为多处理器环境
4     int mp = os::is_MP();
5     //LOCK_IF_MP(%4) 在多处理器环境下，为 cmpxchgl 指令添加 lock 前缀，以达到内存屏障的效果
6     //cmpxchgl 指令是包含在 x86 架构及 IA-64 架构中的一个原子条件指令，
7     //它会首先比较 dest 指针指向的内存值是否和 compare_value 的值相等，
8     //如果相等，则双向交换 dest 与 exchange_value，否则就单方面地将 dest 指向的内存值交给exchange_value。
9     //这条指令完成了整个 CAS 操作，因此它也被称为 CAS 指令。
10    __asm__ volatile (LOCK_IF_MP(%4) "cmpxchgl %1,(%3)"
11        : "=a" (exchange_value)
12        : "r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)
13        : "cc", "memory");
14    return exchange_value;
15 }
```

cmpxchgl的详细执行过程：

首先，输入是"r" (exchange_value), "a" (compare_value), "r" (dest), "r" (mp)，表示compare_value存入eax寄存器，而exchange_value、dest、mp的值存入任意的通用寄存器。嵌入式汇编规定把输出和输入寄存器按统一顺序编号，顺序是从输出寄存器序列从左到右从上到下以"%0"开始，分别记为%0、%1...%9。也就是说，输出的eax是%0，输入的exchange_value、compare_value、dest、mp分别是%1、%2、%3、%4。

因此，cmpxchg %1,(%3)实际上表示cmpxchg exchange_value,(dest)

需要注意的是cmpxchg有个隐含操作数eax，其实际过程是先比较eax的值(也就是compare_value)和dest地址所存的值是否相等，

输出是"a" (exchange_value)，表示把eax中存的值写入exchange_value变量中。

Atomic::cmpxchg这个函数最终返回值是exchange_value，也就是说，如果cmpxchgl执行时compare_value和dest指针指向内存值相等则会使得dest指针指向内存值变成exchange_value，最终eax存的compare_value赋值给了exchange_value变量，即函数最终返回的值是原先的compare_value。此时Unsafe_CompareAndSwapInt的返回值(jint) (Atomic::cmpxchg(x, addr, e)) == e就是true，表明CAS成功。如果cmpxchgl执行时compare_value和(dest)不等则会把当前dest指针指向内存的值写入eax，最终输出时赋值给exchange_value变量作为返回值，导致(jint)(Atomic::cmpxchg(x, addr, e)) == e得到false，表明CAS失败。

现代处理器指令集架构基本上都会提供 CAS 指令，例如 x86 和 IA-64 架构中的 cmpxchgl 指令和 comxchgg 指令，sparc 架构中的 cas 指令和 casx 指令。

不管是 Hotspot 中的 Atomic::cmpxchg 方法，还是 Java 中的 compareAndSwapInt 方法，它们本质上都是对相应平台的 CAS 指令的一层简单封装。CAS 指令作为一种硬件原语，有着天然的原子性，这也正是 CAS 的价值所在。

CAS缺陷

CAS 虽然高效地解决了原子操作，但是还是存在一些缺陷的，主要表现在三个方面：

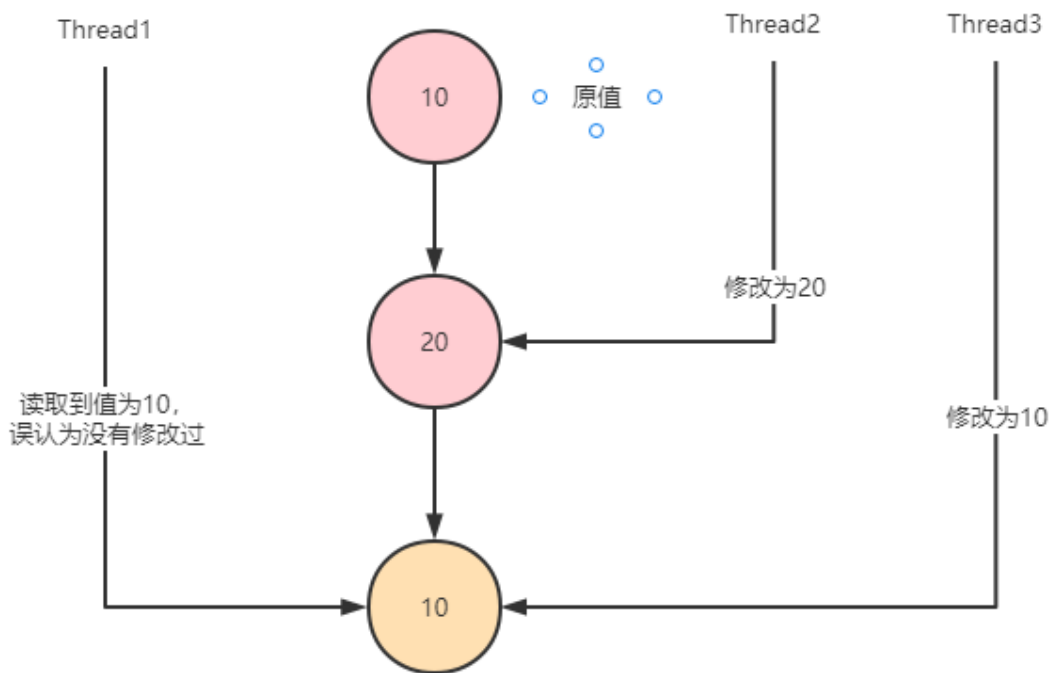
- 自旋 CAS 长时间地不成功，则会给 CPU 带来非常大的开销
- 只能保证一个共享变量原子操作
- ABA 问题

ABA问题及其解决方案

CAS算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

什么是ABA问题

当有多个线程对一个原子类进行操作的时候，某个线程在短时间内将原子类的值A修改为B，又马上将其修改为A，此时其他线程不感知，还是会修改成功。



测试

```
1 @Slf4j
2 public class ABATest {
3
4     public static void main(String[] args) {
5         AtomicInteger atomicInteger = new AtomicInteger(1);
6
7         new Thread(()->{
8             int value = atomicInteger.get();
9             log.debug("Thread1 read value: " + value);
10
11             // 阻塞1s
```

```

12  LockSupport.parkNanos(1000000000L);
13
14  // Thread1通过CAS修改value值为3
15  if (atomicInteger.compareAndSet(value, 3)) {
16  log.debug("Thread1 update from " + value + " to 3");
17  } else {
18  log.debug("Thread1 update fail!");
19  }
20  }, "Thread1").start();
21
22  new Thread(()->{
23  int value = atomicInteger.get();
24  log.debug("Thread2 read value: " + value);
25  // Thread2通过CAS修改value值为2
26  if (atomicInteger.compareAndSet(value, 2)) {
27  log.debug("Thread2 update from " + value + " to 2");
28
29  // do something
30  value = atomicInteger.get();
31  log.debug("Thread2 read value: " + value);
32  // Thread2通过CAS修改value值为1
33  if (atomicInteger.compareAndSet(value, 1)) {
34  log.debug("Thread2 update from " + value + " to 1");
35  }
36  }
37  }, "Thread2").start();
38  }
39  }

```

Thread1不清楚Thread2对value的操作，误以为value=1没有修改过

```

va\jdk1.8.0_181\bin\java.exe" ...
1] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread1 read value: 1
2] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread2 read value: 1
2] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread2 update from 1 to 2
2] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread2 read value: 2
2] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread2 update from 2 to 1
1] DEBUG com.tuling.jucdemo.atomic.ABATest - Thread1 update from 1 to 3

```

ABA问题的解决方案

数据库有个锁称为乐观锁，是一种基于数据版本实现数据同步的机制，每次修改一次数据，版本就会进行累加。

同样，Java也提供了相应的原子引用类AtomicStampedReference<V>

```

public class AtomicStampedReference<V> {

    private static class Pair<T> {
        final T reference;
        final int stamp;
        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;

```

reference即我们实际存储的变量，stamp是版本，每次修改可以通过+1保证版本唯一性。这样就可以保证每次修改后的版本也会往上递增。

```

1  @Slf4j
2  public class AtomicStampedReferenceTest {
3
4      public static void main(String[] args) {
5          // 定义AtomicStampedReference Pair.reference值为1, Pair.stamp为1
6          AtomicStampedReference atomicStampedReference = new
            AtomicStampedReference(1,1);
7
8          new Thread(()->{
9              int[] stampHolder = new int[1];
10             int value = (int) atomicStampedReference.get(stampHolder);
11             int stamp = stampHolder[0];
12             log.debug("Thread1 read value: " + value + ", stamp: " + stamp);
13
14             // 阻塞1s
15             LockSupport.parkNanos(1000000000L);
16             // Thread1通过CAS修改value值为3
17             if (atomicStampedReference.compareAndSet(value, 3, stamp, stamp+1)) {
18                 log.debug("Thread1 update from " + value + " to 3");
19             } else {
20                 log.debug("Thread1 update fail!");
21             }
22             }, "Thread1").start();
23
24             new Thread(()->{
25                 int[] stampHolder = new int[1];
26                 int value = (int)atomicStampedReference.get(stampHolder);
27                 int stamp = stampHolder[0];
28                 log.debug("Thread2 read value: " + value+ ", stamp: " + stamp);
29                 // Thread2通过CAS修改value值为2
30                 if (atomicStampedReference.compareAndSet(value, 2, stamp, stamp+1)) {
31                     log.debug("Thread2 update from " + value + " to 2");
32

```



```

33 // do something
34
35 value = (int) atomicStampedReference.get(stampHolder);
36 stamp = stampHolder[0];
37 log.debug("Thread2 read value: " + value + ", stamp: " + stamp);
38 // Thread2通过CAS修改value值为1
39 if (atomicStampedReference.compareAndSet(value, 1, stamp, stamp+1)) {
40 log.debug("Thread2 update from " + value + " to 1");
41 }
42 }
43 }, "Thread2").start();
44 }
45 }

```

Thread1并没有成功修改value

```

mic.AtomicStampedReferenceTest - Thread2 read value: 1, stamp: 1
mic.AtomicStampedReferenceTest - Thread1 read value: 1, stamp: 1
mic.AtomicStampedReferenceTest - Thread2 update from 1 to 2
mic.AtomicStampedReferenceTest - Thread2 read value: 2, stamp: 2
mic.AtomicStampedReferenceTest - Thread2 update from 2 to 1
mic.AtomicStampedReferenceTest - Thread1 update fail!

```

补充: AtomicMarkableReference可以理解为上面AtomicStampedReference的简化版, 就是不关心修改过几次, 仅仅关心是否修改过。因此变量mark是boolean类型, 仅记录值是否有过修改。

```

public class AtomicMarkableReference<V> {
    private static class Pair<T> {
        final T reference;
        final boolean mark;
        private Pair(T reference, boolean mark) {
            this.reference = reference;
            this.mark = mark;
        }
        static <T> Pair<T> of(T reference, boolean mark) { return new Pair<T>(reference, mark); }
    }
    private volatile Pair<V> pair;
}

```