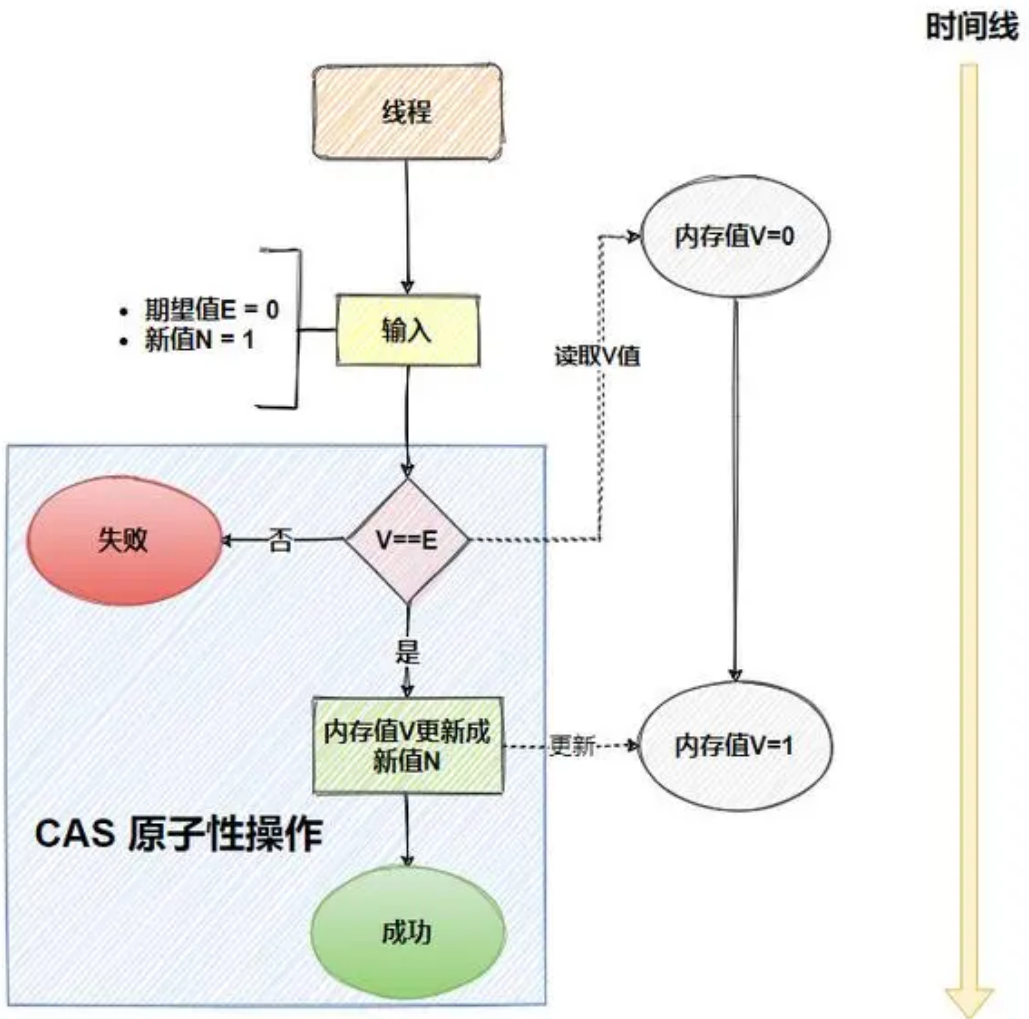
主讲老师：Fox

> 有道笔记： https://note.youdao.com/s/AJECfbdA

1.CAS介绍

什么是 CAS

CAS（Compare And Swap，比较与交换），是非阻塞同步的实现原理，它是CPU硬件层面的一种指令，从CPU层面能保证"比较与交换"两个操作的原子性。CAS指令操作包括三个参数：内存值(内存地址值)V、预期值E、新值N，当CAS指令执行时，当且仅当预期值E和内存值V相同时，才更新内存值为N，否则就不执行更新，无论更新与否都会返回否会返回旧的内存值V，上述的处理过程是一个原子操作。



用Java代码等效实现一下CAS的执行过程：

```java
public class CASDemo {

    // 内存中当前的值
    private volatile int ramAddress;

    /**
     * @param expectedValue 期望值
     * @return newValue 更新的值
     **/
    public synchronized int compareAndSwap(int expectedValue, int newValue) {
        //TODO 模拟直接从内存地址读取到内存中的值
        int oldRamAddress  = accessMemory(ramAddress);
        //内存中的值和期望的值进行比较
        if (oldRamAddress  == expectedValue) {
            ramAddress = newValue;
        }
        return oldRamAddress;
    }

    private int accessMemory(int ramAddress) {
        //TODO  模拟直接从内存地址读取到内存中的值
        return ramAddress;
    }

}
```

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使用

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。

> 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。

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。

```java
public class CASTest {

    public static void main(String[] args) {
        Entity entity = new Entity();

        Unsafe unsafe = UnsafeFactory.getUnsafe();

        long offset = UnsafeFactory.getFieldOffset(unsafe, Entity.class, "x");

        boolean successful;

        // 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
        successful = unsafe.compareAndSwapInt(entity, offset, 0, 3);
        System.out.println(successful + "\t" + entity.x);

        successful = unsafe.compareAndSwapInt(entity, offset, 3, 5);
        System.out.println(successful + "\t" + entity.x);

        successful = unsafe.compareAndSwapInt(entity, offset, 3, 8);
        System.out.println(successful + "\t" + entity.x);
    }
}

public class UnsafeFactory {

    /**
     * 获取 Unsafe 对象
     * @return
     */
    public static Unsafe getUnsafe() {
        try {
            Field field = Unsafe.class.getDeclaredField("theUnsafe");
            field.setAccessible(true);
            return (Unsafe) field.get(null);
        } catch (Exception e) {
            e.printStackTrace();
        }
```

```
        return null;
    }

    /**
     * 获取字段的内存偏移量
     * @param unsafe
     * @param clazz
     * @param fieldName
     * @return
     */
    public static long getFieldOffset(Unsafe unsafe, Class clazz, String fieldName) {
        try {
            return unsafe.objectFieldOffset(clazz.getDeclaredField(fieldName));
        } catch (NoSuchFieldException e) {
            throw new Error(e);
        }
    }
}
```

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：

CAS应用场景

CAS在java.util.concurrent.atomic相关类、Java AQS、CurrentHashMap等实现上有非常广泛的应用。

如下图所示，AtomicInteger的实现中，静态字段valueOffset即为字段value的内存偏移地址，valueOffset的值在AtomicInteger初始化时，在静态代码块中通过Unsafe的objectFieldOffset方法获取。在AtomicInteger中提供的线程安全方法中，通过字段valueOffset的值可以定位到AtomicInteger对象中value的内存地址，从而可以根据CAS实现对value字段的原子操作。

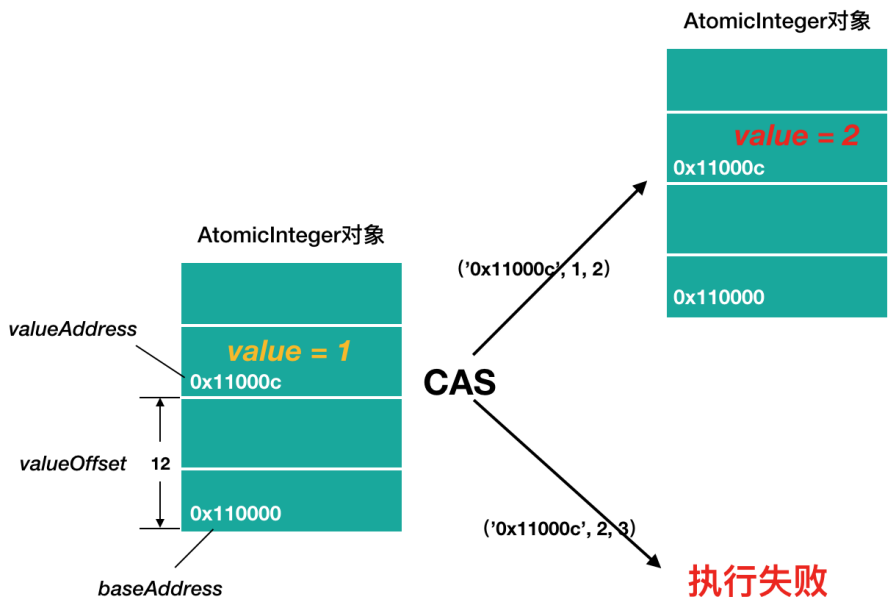
```java
public class AtomicInteger extends Number implements java.io.Serializable {
    private static final long serialVersionUID = 6214790243416807050L;

    // setup to use Unsafe.compareAndSwapInt for updates
    private static final Unsafe unsafe = Unsafe.getUnsafe();
    private static final long valueOffset;

    static {
        try {
            valueOffset = unsafe.objectFieldOffset
                (AtomicInteger.class.getDeclaredField( name: "value"));
        } catch (Exception ex) { throw new Error(ex); }
    }

    private volatile int value;
```

下图为某个AtomicInteger对象自增操作前后的内存示意图，对象的基地址baseAddress=“0x110000”，通过baseAddress+valueOffset得到value的内存地址valueAddress=“0x11000c”；然后通过CAS进行原子性的更新操作，成功则返回，否则继续重试，直到更新成功为止。



CAS源码分析

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

```
#unsafe.cpp
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);
  // 根据偏移量，计算value的地址
  jint* addr = (jint *) index_oop_from_field_offset_long(p, offset);
  // Atomic::cmpxchg(x, addr, e) cas逻辑 x:要交换的值    e:要比较的值
  //cas成功，返回期望值e，等于e,此方法返回true
  //cas失败，返回内存中的value值，不等于e，此方法返回false
  return (jint)(Atomic::cmpxchg(x, addr, e)) == e;
UNSAFE_END2
```

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

```
#atomic_linux_x86.inline.hpp
inline jint     Atomic::cmpxchg    (jint     exchange_value, volatile jint*     dest, jint     compare_value) {
  //判断当前执行环境是否为多处理器环境
  int mp = os::is_MP();
  //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
  //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，
  //它会首先比较 dest 指针指向的内存值是否和 compare_value 的值相等，
  //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。
  //这条指令完成了整个 CAS 操作，因此它也被称为 CAS 指令。
  __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;
}
```

> 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 comxchgq 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：

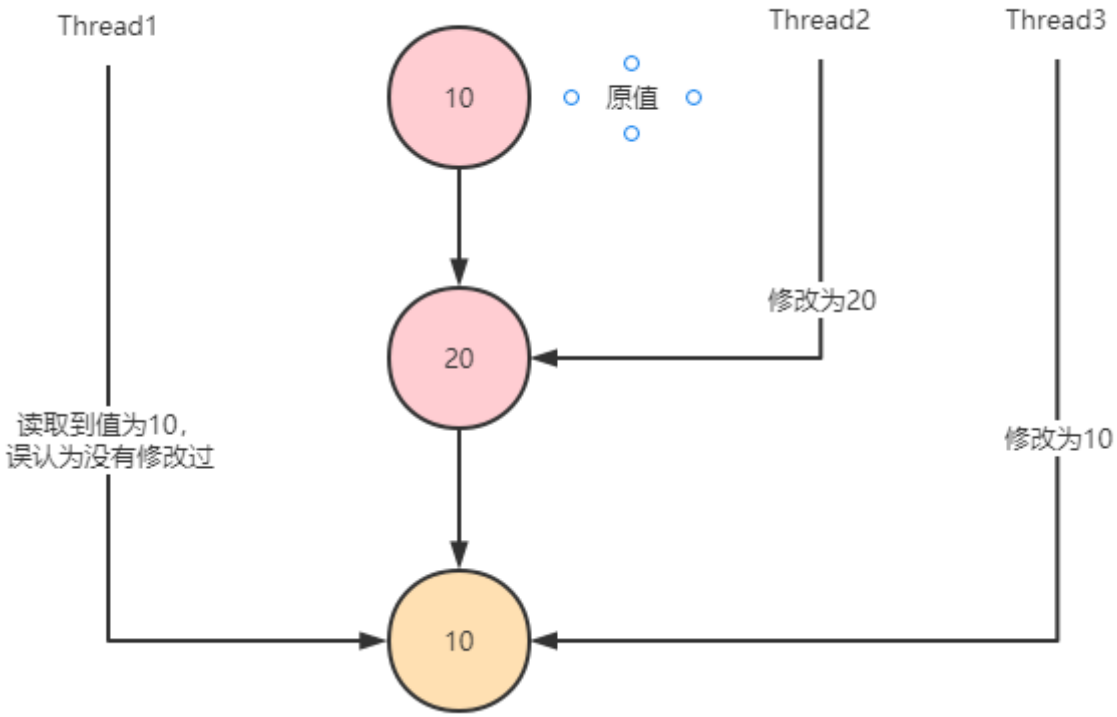
- 自旋 CAS 长时间不成功，则会给 CPU 带来非常大的开销
- 只能保证一个共享变量原子操作
- ABA 问题

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

```java
@Slf4j
public class ABATest {

    public static void main(String[] args) {
        AtomicInteger atomicInteger = new AtomicInteger(1);

        new Thread(()->{
            int value = atomicInteger.get();
            log.debug("Thread1 read value: " + value);

            // 阻塞1s
            LockSupport.parkNanos(1000000000L);

            // Thread1通过CAS修改value值为3
            if (atomicInteger.compareAndSet(value, 3)) {
                log.debug("Thread1 update from " + value + " to 3");
            } else {
                log.debug("Thread1 update fail!");
            }
        },"Thread1").start();

        new Thread(()->{
            int value = atomicInteger.get();
            log.debug("Thread2 read value: " + value);
            // Thread2通过CAS修改value值为2
            if (atomicInteger.compareAndSet(value, 2)) {
                log.debug("Thread2 update from " + value + " to 2");

                // do something
                value = atomicInteger.get();
                log.debug("Thread2 read value: " + value);
                // Thread2通过CAS修改value值为1
                if (atomicInteger.compareAndSet(value, 1)) {
                    log.debug("Thread2 update from " + value + " to 1");
                }
            }
        },"Thread2").start();
    }
```

```
39  }
```

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

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>

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。

```java
@Slf4j
public class AtomicStampedReferenceTest {

    public static void main(String[] args) {
        // 定义AtomicStampedReference    Pair.reference值为1, Pair.stamp为1
        AtomicStampedReference atomicStampedReference = new AtomicStampedReference(1,1);

        new Thread(()->{
            int[] stampHolder = new int[1];
            int value = (int) atomicStampedReference.get(stampHolder);
            int stamp = stampHolder[0];
            log.debug("Thread1 read value: " + value + ", stamp: " + stamp);

            // 阻塞1s
            LockSupport.parkNanos(1000000000L);
            // Thread1通过CAS修改value值为3
            if (atomicStampedReference.compareAndSet(value, 3,stamp,stamp+1)) {
                log.debug("Thread1 update from " + value + " to 3");
            } else {
                log.debug("Thread1 update fail!");
            }
        },"Thread1").start();

        new Thread(()->{
            int[] stampHolder = new int[1];
            int value = (int)atomicStampedReference.get(stampHolder);
            int stamp = stampHolder[0];
            log.debug("Thread2 read value: " + value+ ", stamp: " + stamp);
            // Thread2通过CAS修改value值为2
            if (atomicStampedReference.compareAndSet(value, 2,stamp,stamp+1)) {
                log.debug("Thread2 update from " + value + " to 2");

                // do something

                value = (int) atomicStampedReference.get(stampHolder);
                stamp = stampHolder[0];
                log.debug("Thread2 read value: " + value+ ", stamp: " + stamp);
```

```
                // Thread2通过CAS修改value值为1
                if (atomicStampedReference.compareAndSet(value, 1,stamp,stamp+1)) {
                    log.debug("Thread2 update from " + value + " to 1");
                }
            }
        },"Thread2").start();
    }
}
```

Thread1并没有成功修改value

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。

2.Atomic原子操作类介绍

在并发编程中很容易出现并发安全的问题，有一个很简单的例子就是多线程更新变量i=1,比如多个线程执行i++操作，就有可能获取不到正确的值，而这个问题，最常用的方法是通过Synchronized进行控制来达到线程安全的目的。但是由于synchronized是采用的是悲观锁策略，并不是特别高效的一种解决方案。实际上，在J.U.C下的atomic包提供了一系列的操作简单，性能高效，并能保证线程安全的类去更新基本类型变量，数组元素，引用类型以及更新对象中的字段类型。atomic包下的这些类都是采用的是乐观锁策略去原子更新数据，在java中则是使用CAS操作具体实现。

在java.util.concurrent.atomic包里提供了一组原子操作类：

基本类型：AtomicInteger、AtomicLong、AtomicBoolean；

引用类型：AtomicReference、AtomicStampedRerence、AtomicMarkableReference；

数组类型：AtomicIntegerArray、AtomicLongArray、AtomicReferenceArray

对象属性原子修改器：AtomicIntegerFieldUpdater、AtomicLongFieldUpdater、AtomicReferenceFieldUpdater

原子类型累加器（jdk1.8增加的类）：DoubleAccumulator、DoubleAdder、LongAccumulator、LongAdder、Striped64

原子更新基本类型

以AtomicInteger为例总结常用的方法

```java
//以原子的方式将实例中的原值加1，返回的是自增前的旧值；
public final int getAndIncrement() {
    return unsafe.getAndAddInt(this, valueOffset, 1);
}

//getAndSet(int newValue)：将实例中的值更新为新值，并返回旧值；
public final boolean getAndSet(boolean newValue) {
    boolean prev;
    do {
        prev = get();
    } while (!compareAndSet(prev, newValue));
    return prev;
}

//incrementAndGet() ：以原子的方式将实例中的原值进行加1操作，并返回最终相加后的结果；
public final int incrementAndGet() {
    return unsafe.getAndAddInt(this, valueOffset, 1) + 1;
}

//addAndGet(int delta) ：以原子方式将输入的数值与实例中原本的值相加，并返回最后的结果；
public final int addAndGet(int delta) {
    return unsafe.getAndAddInt(this, valueOffset, delta) + delta;

```

测试

```java
public class AtomicIntegerTest {
    static AtomicInteger sum = new AtomicInteger(0);

    public static void main(String[] args) {

        for (int i = 0; i < 10; i++) {
            Thread thread = new Thread(() -> {
                for (int j = 0; j < 10000; j++) {
                    // 原子自增  CAS
                    sum.incrementAndGet();
                    //TODO
                }
            });
            thread.start();
        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(sum.get());

    }

}
```

incrementAndGet()方法通过CAS自增实现，如果CAS失败，自旋直到成功+1。

思考：这种CAS失败自旋的操作存在什么问题?

原子更新数组类型

AtomicIntegerArray为例总结常用的方法

```java
//addAndGet(int i, int delta)：以原子更新的方式将数组中索引为i的元素与输入值相加；
public final int addAndGet(int i, int delta) {
    return getAndAdd(i, delta) + delta;
}

//getAndIncrement(int i)：以原子更新的方式将数组中索引为i的元素自增加1；
public final int getAndIncrement(int i) {
    return getAndAdd(i, 1);
}

//compareAndSet(int i, int expect, int update)：将数组中索引为i的位置的元素进行更新
public final boolean compareAndSet(int i, int expect, int update) {
    return compareAndSetRaw(checkedByteOffset(i), expect, update);
}
```

测试

```java
public class AtomicIntegerArrayTest {

    static int[] value = new int[]{ 1, 2, 3, 4, 5 };
    static AtomicIntegerArray atomicIntegerArray = new AtomicIntegerArray(value);


    public static void main(String[] args) throws InterruptedException {

        //设置索引0的元素为100
        atomicIntegerArray.set(0, 100);
        System.out.println(atomicIntegerArray.get(0));
        //以原子更新的方式将数组中索引为1的元素与输入值相加
        atomicIntegerArray.getAndAdd(1,5);

        System.out.println(atomicIntegerArray);
    }
}
```

原子更新引用类型

AtomicReference作用是对普通对象的封装，它可以保证你在修改对象引用时的线程安全性。

```
public class AtomicReferenceTest {

    public static void main( String[] args ) {
        User user1 = new User("张三", 23);
        User user2 = new User("李四", 25);
        User user3 = new User("王五", 20);

        //初始化为 user1
        AtomicReference<User> atomicReference = new AtomicReference<>();
        atomicReference.set(user1);

        //把 user2 赋给 atomicReference
        atomicReference.compareAndSet(user1, user2);
        System.out.println(atomicReference.get());

        //把 user3 赋给 atomicReference
        atomicReference.compareAndSet(user1, user3);
        System.out.println(atomicReference.get());

    }

}


@Data
@AllArgsConstructor
class User {
    private String name;
    private Integer age;
}
```

对象属性原子修改器

AtomicIntegerFieldUpdater可以线程安全地更新对象中的整型变量。

```java
public class AtomicIntegerFieldUpdaterTest {

    public static class Candidate {

        volatile int score = 0;

        AtomicInteger score2 = new AtomicInteger();
    }

    public static final AtomicIntegerFieldUpdater<Candidate> scoreUpdater =
            AtomicIntegerFieldUpdater.newUpdater(Candidate.class, "score");

    public static AtomicInteger realScore = new AtomicInteger(0);

    public static void main(String[] args) throws InterruptedException {

        final Candidate candidate = new Candidate();

        Thread[] t = new Thread[10000];
        for (int i = 0; i < 10000; i++) {
            t[i] = new Thread(new Runnable() {
                @Override
                public void run() {
                    if (Math.random() > 0.4) {
                        candidate.score2.incrementAndGet();
                        scoreUpdater.incrementAndGet(candidate);
                        realScore.incrementAndGet();
                    }
                }
            });
            t[i].start();
        }
        for (int i = 0; i < 10000; i++) {
            t[i].join();
        }
        System.out.println("AtomicIntegerFieldUpdater Score=" + candidate.score);
        System.out.println("AtomicInteger Score=" + candidate.score2.get());
        System.out.println("realScore=" + realScore.get());
```

```
39
40          }
41  }
```

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

（1）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

（2）字段的描述类型（修饰符public/protected/default/private）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。

（3）只能是实例变量，不能是类变量，也就是说不能加static关键字。

（4）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。

（5）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。

LongAdder/DoubleAdder详解

AtomicLong是利用了底层的CAS操作来提供并发性的，比如addAndGet方法：

上述方法调用了Unsafe类的getAndAddLong方法，该方法内部是个native方法，它的逻辑是采用自旋的方式不断更新目标值，直到更新成功。

在并发量较低的环境下，线程冲突的概率比较小，自旋的次数不会很多。但是，高并发环境下，N个线程同时进行自旋操作，会出现大量失败并不断自旋的情况，此时AtomicLong的自旋会成为瓶颈。

这就是**LongAdder引入的初衷——解决高并发环境下AtomicInteger，AtomicLong的自旋瓶颈问题。**

性能测试

```java
public class LongAdderTest {

    public static void main(String[] args) {
        testAtomicLongVSLongAdder(10, 10000);
        System.out.println("==================");
        testAtomicLongVSLongAdder(10, 200000);
        System.out.println("==================");
        testAtomicLongVSLongAdder(100, 200000);
    }

    static void testAtomicLongVSLongAdder(final int threadCount, final int times) {
        try {
            long start = System.currentTimeMillis();
            testLongAdder(threadCount, times);
            long end = System.currentTimeMillis() - start;
            System.out.println("条件>>>>>>线程数:" + threadCount + ", 单线程操作计数" + times);
            System.out.println("结果>>>>>>LongAdder方式增加计数" + (threadCount * times) + "次,共计耗时:" + end);

            long start2 = System.currentTimeMillis();
            testAtomicLong(threadCount, times);
            long end2 = System.currentTimeMillis() - start2;
            System.out.println("条件>>>>>>线程数:" + threadCount + ", 单线程操作计数" + times);
            System.out.println("结果>>>>>>AtomicLong方式增加计数" + (threadCount * times) + "次,共计耗时:" + end2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static void testAtomicLong(final int threadCount, final int times) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        AtomicLong atomicLong = new AtomicLong();
        for (int i = 0; i < threadCount; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
```

```
                    for (int j = 0; j < times; j++) {
                        atomicLong.incrementAndGet();
                    }
                    countDownLatch.countDown();
                }
            }, "my-thread" + i).start();
        }
        countDownLatch.await();
    }

    static void testLongAdder(final int threadCount, final int times) throws InterruptedException {
        CountDownLatch countDownLatch = new CountDownLatch(threadCount);
        LongAdder longAdder = new LongAdder();
        for (int i = 0; i < threadCount; i++) {
            new Thread(new Runnable() {
                @Override
                public void run() {
                    for (int j = 0; j < times; j++) {
                        longAdder.add(1);
                    }
                    countDownLatch.countDown();
                }
            }, "my-thread" + i).start();
        }

        countDownLatch.await();
    }
}
```

测试结果：线程数越多，并发操作数越大，LongAdder的优势越明显

低并发、一般的业务场景下AtomicLong是足够了。如果并发量很多，存在大量写多读少的情况，那LongAdder可能更合适。

LongAdder原理

设计思路

AtomicLong中有个内部变量value保存着实际的long值，所有的操作都是针对该变量进行。也就是说，高并发环境下，value变量其实是一个热点，也就是N个线程竞争一个热点。LongAdder的基本思路就是分散热点，将value值分散到一个数组中，不同线程会命中到数组的不同槽中，各个线程只对自己槽中的那个值进行CAS操作，这样热点就被分散了，冲突的概率就小很多。如果要获取真正的long值，只要将各个槽中的变量值累加返回。

LongAdder的内部结构

LongAdder内部有一个base变量，一个Cell[]数组：

base变量：非竞态条件下，直接累加到该变量上

Cell[]数组：竞态条件下，累加个各个线程自己的槽Cell[i]中

```java
/** Number of CPUS, to place bound on table size */
// CPU核数，用来决定槽数组的大小
static final int NCPU = Runtime.getRuntime().availableProcessors();

/**
 * Table of cells. When non-null, size is a power of 2.
 */
 // 数组槽，大小为2的次幂
transient volatile Cell[] cells;

/**
 * Base value, used mainly when there is no contention, but also as
 * a fallback during table initialization races. Updated via CAS.
 */
 /**
 *  基数，在两种情况下会使用：
 *  1. 没有遇到并发竞争时，直接使用base累加数值
 *  2. 初始化cells数组时，必须要保证cells数组只能被初始化一次（即只有一个线程能对cells初始化），
 *  其他竞争失败的线程会将数值累加到base上
 */
transient volatile long base;

/**
 * Spinlock (locked via CAS) used when resizing and/or creating Cells.
 */
transient volatile int cellsBusy;
```

定义了一个内部Cell类，这就是我们之前所说的槽，每个Cell对象存有一个value值，可以通过Unsafe来CAS操作它的值：

LongAdder#add方法

LongAdder#add方法的逻辑如下图：

只有从未出现过并发冲突的时候，base基数才会使用到，一旦出现了并发冲突，之后所有的操作都只针对Cell[]数组中的单元Cell。

如果Cell[]数组未初始化，会调用父类的longAccumelate去初始化Cell[]，如果Cell[]已经初始化但是冲突发生在Cell单元内，则也调用父类的longAccumelate，此时可能就需要对Cell[]扩容了。

这也是LongAdder设计的精妙之处：尽量减少热点冲突，不到最后万不得已，尽量将CAS操作延迟。

Striped64#longAccumulate方法

整个Striped64#longAccumulate的流程图如下：

LongAdder#sum方法

```
/**
* 返回累加的和，也就是"当前时刻"的计数值
* 注意：  高并发时，除非全局加锁，否则得不到程序运行中某个时刻绝对准确的值
*  此返回值可能不是绝对准确的，因为调用这个方法时还有其他线程可能正在进行计数累加，
*  方法的返回时刻和调用时刻不是同一个点，在有并发的情况下，这个值只是近似准确的计数值
*/
public long sum() {
    Cell[] as = cells; Cell a;
    long sum = base;
    if (as != null) {
        for (int i = 0; i < as.length; ++i) {
            if ((a = as[i]) != null)
                sum += a.value;
        }
    }
    return sum;
}
```

由于计算总和时没有对Cell数组进行加锁，所以在累加过程中可能有其他线程对Cell中的值进行了修改，也有可能对数组进行了扩容，所以sum返回的值并不是非常精确的，其返回值并不是一个调用sum方法时的原子快照值。