

# 线程安全概念

## 竞态条件和临界区

多个线程访问了相同的资源，向这些资源做了写操作时，对执行顺序有要求。下面是一段有线程安全问题的代码，从代码可知我们的预期结果应该是result: 70000，结果却是result: 63040，并且执行多次结果有可能不一致。

```
public class UnsafeThreadDemo {
    private volatile int i = 0;

    public void incr() {
        i++;
    }

    public int getI() {
        return i;
    }

    public static void main(String[] args) throws InterruptedException {
        UnsafeThreadDemo unsafeThreadDemo = new UnsafeThreadDemo();
        for (int i = 0; i < 7; i++) {
            new Thread(){
                @Override
                public void run() {
                    for (int j = 0; j < 10000; j++) {
                        unsafeThreadDemo.incr();
                    }
                }
            }.start();
        }
        Thread.sleep(2000);
        System.out.println("result: " + unsafeThreadDemo.getI());
    }
}
```

这是因为incr()方法中的i++操作不是一个原子操作，这一点我们可以从incr()的字节码中看出。实现i++需要三步，首先获取i的旧值，然后将旧值加1在写入i变量。

```

public void incr();
descriptor: ()V
flags: ACC_PUBLIC
Code:
    stack=3, locals=1, args_size=1
        0: aload_0
        1: dup
        2: getfield      #2                // Field i:I
        5: iconst_1
        6: iadd
        7: putfield      #2                // Field i:I
       10: return

```

1. 临界区：incr方法内部就是临界区域，关键部分代码的多线程并发执行，会对执行结果产生影响。
2. 竞态条件：可能发生在临界区域内的特殊条件。多线程执行incr方法中的i++关键代码时，产生了竞态条件。

## 共享资源

1. 如果一段代码是线程安全的，则它不包含竞态条件。只有当多个线程更新共享资源时，才会发生竞态条件。
2. 栈封闭时，不会在线程之间共享变量，都是线程安全的。
3. 局部对象引用本身不共享，但是引用的对象存储在共享堆中。如果方法内创建的对象，只是在方法中传递，并且不对其他线程可用，那么也是线程安全的。

```

public void someMethod(){
    LocalObject localObject = new LocalObject();
    localObject.callMethod();
    method2(localObject);
}

public void method2(LocalObject localObject){
    localObject.setValue("value");
}

```

局部变量~~~引用

## 不可变对象

1. 创建不可变的共享对象来保证对象在线程间共享时不会被修改，从而实现线程安全。实例被创建，value变量就不能再被修改，这就是不可变性。

```

public class Demo{
    private int value = 0;
    public Demo(int value){
        this.value = value;
    }
    public int getValue(){
        return this.value;
    }
}

```

## 原子操作的定义

1. 原子操作可以是一个步骤，也可以是多个操作步骤，但是其顺序不可以被打乱，也不可以被切割而知执行其中的一部分(不可中断性)。
2. 将整个操作视作一个整体，资源在该次操作中保持一致，这是原子性的核心特征。
3. 其本质是对数据一致性的要求。



## CAS机制

1. Compare and swap比较和交换。属于硬件原语，处理器提供了基本内存操作的原子性保证。
2. CAS操作需要输入两个数值，一个旧值A(期望操作前的值)和一直新值B，在操作期间先比较旧值有没有发生变化，如果没有发生变化，才交换成新值，发生变化则不交换。
3. JAVA中的sun.misc.Unsafe类，提供了compareAndSwapint()和compareAndSwapLong()等几个方法实现CAS。



接下来用CAS机制来更正上面例子的线程不安全。多次运行结果为70000。

```
public class CASDemo {
    private volatile int i = 0;

    private static Unsafe unsafe;

    private static Long valueOffset;

    static {
        try {
            Field theUnsafe = Unsafe.class.getDeclaredField("theUnsafe");
            theUnsafe.setAccessible(true);
            unsafe = (Unsafe)theUnsafe.get(null);
            valueOffset = unsafe.objectFieldOffset(CASDemo.class.getDeclaredField("i"));
        } catch (NoSuchFieldException e) {
            e.printStackTrace();
        }
    }
}
```

```

        } catch (IllegalAccessEception e) {
            e.printStackTrace();
        }
    }

    public void incr() {
        int current;
        int newValue;
        do {
            current = unsafe.getIntVolatile(this, valueOffset);
            newValue = current + 1;
        } while (!unsafe.compareAndSwapInt(this, valueOffset, current, newValue));
    }

    public int getI() {
        return i;
    }

    public static void main(String[] args) throws InterruptedException {
        CASDemo casDemo = new CASDemo();
        for (int i = 0; i < 7; i++) {
            new Thread(){
                @Override
                public void run() {
                    for (int j = 0; j < 10000; j++) {
                        casDemo.incr();
                    }
                }
            }.start();
        }
        Thread.sleep(2000);
        System.out.println("result: " + casDemo.getI());
    }
}

```

## JUC包内的原子操作封装类

AtomicBoolean: 原子更新布尔类型

AtomicInteger: 原子更新整型

AtomicLong: 原子更新长整型

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。

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。

AtomicReferenceFieldUpdater: 原子更新引用类型里的字段。

AtomicReference: 原子更新引用类型。

AtomicStampedReference: 原子更新带有版本号引用类型

AtomicMarkableReference: 原子更新带有标记位的引用类型。

#### 1.8更新

更新器: DoubleAccumulator、LongAccumulator

计数器: DoubleAdder、LongAdder

计数器增强版, 高并发下性能更好

频繁更新但不太频繁读取的汇总统计信息时使用

分成多个操作单元, 不同线程更新不同的单元

只有需要汇总的时候才计算所有单元的操作

接下来我们比较一下加锁, 原子类和更新器的性能。我们在2秒内测试三种方法执行的次数, 结果可知DoubleAdder性能最好, Atomic原子类次之, 最后是同步加锁。

```
public class SynUtilPerformanceDemo {

    private static Long aLong = 0L;

    private static AtomicLong atomicLong = new AtomicLong(0);

    private static DoubleAdder doubleAdder = new DoubleAdder();

    public synchronized void add() {
        aLong++;
    }

    public Long get() {
        return aLong;
    }

    public static void testSync() throws InterruptedException {
        SynUtilPerformanceDemo synUtilPerformanceDemo = new SynUtilPerformanceDemo();
        long startTime = System.currentTimeMillis();
        do {
            for (int i = 0; i < 7; i++) {
                new Thread() {
                    @Override
                    public void run() {
                        synUtilPerformanceDemo.add();
                    }
                }.start();
            }
        } while (System.currentTimeMillis() - startTime <= 2000);
        Thread.sleep(3000);
        System.out.println("Synchronize execute " + aLong + " times");
    }
}
```

```

public static void testAtomic() throws InterruptedException {
    SynUtilPerformanceDemo synUtilPerformanceDemo = new SynUtilPerformanceDemo();
    long startTime = System.currentTimeMillis();
    do {
        for (int i = 0; i < 7; i++) {
            new Thread() {
                @Override
                public void run() {
                    synUtilPerformanceDemo.atomicLong.incrementAndGet();
                }
            }.start();
        }
        while (System.currentTimeMillis() - startTime <= 2000);
        Thread.sleep(3000);
        System.out.println("Atomic execute " + synUtilPerformanceDemo.atomicLong.get() + "
times");
    }

    public static void testDoubleAdder() throws InterruptedException {
        SynUtilPerformanceDemo synUtilPerformanceDemo = new SynUtilPerformanceDemo();
        long startTime = System.currentTimeMillis();
        do {
            for (int i = 0; i < 7; i++) {
                new Thread() {
                    @Override
                    public void run() {
                        synUtilPerformanceDemo.doubleAdder.add(1);
                    }
                }.start();
            }
            while (System.currentTimeMillis() - startTime <= 2000);
            Thread.sleep(3000);
            System.out.println("DoubleAdder execute " +
synUtilPerformanceDemo.doubleAdder.doubleValue() + " times");
        }

        public static void main(String[] args) throws InterruptedException {
            testSync();
            testAtomic();
            testDoubleAdder();
        }
    }
}

```

```

Synchronize execute 7889 times
Atomic execute 12236 times
DoubleAdder execute 11032.0 times

```

## CAS的三个问题

1. 循环CAS，自旋的实现让所有线程都处于高频运行，争抢CPU执行时间的状态。如果操作长时间不成功，会带来很大的CPU资源消耗。

2. 仅针对单个变量的操作，不能用于多个变量来实现原子操作。

3. ABA问题。(无法体现出数据的变动)。

下面我们来看一个ABA问题的例子。我们自己实现了一个简单的栈，其中用原子更新引用类来更新栈顶指针。

执行Test类我们预期输出栈的元素应该是ACDB，而结果却是B。这是因为线程thread2在弹出A元素后，压入了CD元素，又压入了A元素。在这之后thread1执行，根据CAS机制判断旧值还是A所以直接将栈顶指针更新为B，它看不到数据的更新过程。使用带版本的原子引用后，每次元素经过操作后，版本都会不同，因此CAS操作会失败重试，直到拿到最新结果，因此执行结果符合预期。

```
public class Node {

    public final String item;

    public Node next;

    public Node(String item) {
        this.item = item;
    }
}

public class Stack {

    AtomicReference<Node> top = new AtomicReference<>();

    public void push(Node node) {
        Node oldTop;
        do {
            oldTop = top.get();
            node.next = oldTop;
        } while (!top.compareAndSet(oldTop, node));
    }

    public Node pop(int time) throws InterruptedException {
        Node newTop;
        Node oldTop;
        do {
            oldTop = top.get();
            if (oldTop == null) {
                return null;
            }
            newTop = oldTop.next;
            TimeUnit.SECONDS.sleep(time);
        } while (!top.compareAndSet(oldTop, newTop));
        return oldTop;
    }
}

public class NoABAStack {

    AtomicStampedReference<Node> top = new AtomicStampedReference<>(null, 0);

    public void push(Node node) {
        Node oldTop;
```

```

        int v;
        do {
            v = top.getStamp();
            oldTop = top.getReference();
            node.next = oldTop;
        } while (!top.compareAndSet(oldTop, node, v, v + 1));
    }

    public Node pop(int time) throws InterruptedException {
        Node newTop;
        Node oldTop;
        int v;
        do {
            v = top.getStamp();
            oldTop = top.getReference();
            if (oldTop == null) {
                return null;
            }
            newTop = oldTop.next;
            TimeUnit.SECONDS.sleep(time);
        } while (!top.compareAndSet(oldTop, newTop, v, v + 1));
        return oldTop;
    }
}

public class Test {
    public static void main(String[] args) throws InterruptedException {
        Stack stack = new Stack();
        //NoABASTack stack = new NoABASTack();
        stack.push(new Node("B"));
        stack.push(new Node("A"));

        Thread thread1 = new Thread() {
            @Override
            public void run() {
                try {
                    stack.pop(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        };
        thread1.start();

        Thread thread2 = new Thread() {
            @Override
            public void run() {
                Node A = null;
                try {
                    A = stack.pop(0);
                    stack.push(new Node("C"));
                    stack.push(new Node("D"));
                    stack.push(A);
                }
            }
        };
        thread2.start();
    }
}

```



```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
};
thread2.start();

TimeUnit.SECONDS.sleep(5);
Node node;
while (null != (node = stack.pop(0))) {
    System.out.println(node.item);
}
}
```