

一，问题引入

1.需求

模拟网站访问量统计，假设现在有100个用户，每个人访问10次。

```
/**
 * @author yhd
 * @createtime 2020/10/3 19:05
 * @description 模拟网站访问量统计
 * 假设现在有100个用户，每个人访问10次。
 */
public class DemoA {

    /**
     * 模拟用户访问
     */
    public static int count = 0;
    public static void request() {
        try {
            count++;
            TimeUnit.SECONDS.sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        //开始时间
        long startTime = System.currentTimeMillis();

        int size = 100;
        CountDownLatch latch = new CountDownLatch(size);

        //模拟100个用户
        for (int i = 0; i < size; i++) {
            new Thread(() -> {
                //模拟每个用户访问10次
                for (int j = 0; j < 10; j++) {
                    request();
                }
                latch.countDown();
            }, "用户" + String.valueOf(i)).start();
        }

        latch.await();

        //结束时间
        long endTime = System.currentTimeMillis();
        System.out.println(Thread.currentThread().getName()+"统计完成：耗时" +
            (endTime - startTime) / 1000 + "s"+" 访问次数: "+count);
    }

    //结果：main统计完成：耗时10s 访问次数： 805
}
```

```
}
```

我们发现输出结果始终是错误的。

1.分析问题出在哪里？

count++ 操做实际上是由3步来完成的！（jvm执行引擎）

- 1.获取count的值，记作A：A=count
- 2.将A的值+1，得到B：B=A+1
- 3.将B的值赋给count

如果有A，B两个线程同时执行count++，他们同时执行到上面步骤的第一步，得到的count是一样的，3步操做结束后，count只加1，导致count结果不正确！

2.怎么解决结果不正确的问题？

对count操做的时候，我们让多个线程排队处理，多个线程同时到达request()方法的时候，只能允许一个线程进去操做其他线程在外面等着，等里面的处理完毕之后，外面等着的再进去一个，这样操做的count++就是排队进行的，结果一定是正确的。

3.怎么实现排队效果？

Java中的synchronized和ReentrantLock都可以实现对资源枷锁，保证并发正确性，多线程情况下可以保证被锁住的资源被“串行”访问。

2.代码改进（线程安全）

```
/**
 * @author yhd
 * @createtime 2020/10/3 19:05
 * @description 模拟网站访问量统计
 * 假设现在有100个用户，每个人访问10次。
 */
public class DemoA {

    /**
     * 模拟用户访问
     */
    public static int count = 0;
    public synchronized static void request() {
        try {
            count++;
            TimeUnit.MILLISECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        //开始时间
        long startTime = System.currentTimeMillis();

        int size = 100;
        CountDownLatch latch = new CountDownLatch(size);

        //模拟100个用户
        for (int i = 0; i < size; i++) {
```

```

        new Thread(() -> {
            //模拟每个用户访问10次
            for (int j = 0; j < 10; j++) {
                request();
            }
            latch.countDown();
        }, "用户" + String.valueOf(i)).start();
    }

    latch.await();

    //结束时间
    long endTime = System.currentTimeMillis();
    System.out.println(Thread.currentThread().getName()+"统计完成：耗时" +
        (endTime - startTime) + "ms"+" 访问次数： "+count);
    }

    //结果：main统计完成：耗时5665ms 访问次数：1000
}

```

结果正确了，可是耗时很长。

1.耗时太长的原因是什么呢？

程序中的request()方法使用synchronized关键字修饰，保证了并发情况下，request方法同一时刻只允许同一个线程进入，request加锁相当于串行执行了，count的结果和我们预期的一致，只是消耗的时间太长了。

2.如何解决耗时长的问题？

count++操做实际上是由三步来完成的！

- 1.获取count的值，记作A：A=count
- 2.将A的值+1，得到B：B=A+1
- 3.将B的值赋给count

升级第三步的实现：

- 1.获取锁
- 2.获取一下count最新值，记作LV
- 3.判断LV是否等于A，如果相等，则将B的值赋值给count，并返回true，否则返回false。
- 4.释放锁

3.再次优化（执行时间）

```

/**
 * @author yhd
 * @createtime 2020/10/3 19:05
 * @description 模拟网站访问量统计
 * 假设现在有100个用户，每个人访问10次。
 */
public class DemoA {

    /**
     * 模拟用户访问
     */
    public volatile static int count = 0;

```

```

public static void request() {
    try {
        while (!compareAndSwap((getCount()), count + 1)) {
        }
        TimeUnit.MILLISECONDS.sleep(5);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

/**
 * 比较count的预期值与当前值，如果相等将新值赋值给count，并返回true，否则返回false。
 *
 * @param expectCount 预期值
 * @param newCount    新值
 * @return count+1;
 */
public synchronized static boolean compareAndSwap(int expectCount, int
newCount) {
    if (getCount() == expectCount) {
        count = newCount;
        return true;
    }
    return false;
}

/**
 * 获取count的值
 *
 * @return
 */
public static int getCount() {
    return count;
}

public static void main(String[] args) throws InterruptedException {
    //开始时间
    long startTime = System.currentTimeMillis();

    int size = 100;
    CountDownLatch latch = new CountDownLatch(size);

    //模拟100个用户
    for (int i = 0; i < size; i++) {
        new Thread(() -> {
            //模拟每个用户访问10次
            for (int j = 0; j < 10; j++) {
                request();
            }
            latch.countDown();
        }, "用户" + String.valueOf(i)).start();
    }

    latch.await();

    //结束时间
    long endTime = System.currentTimeMillis();

```

```

        System.out.println(Thread.currentThread().getName() + "统计完成: 耗时" +
            (endTime - startTime) + "ms" + " 访问次数: " + count);
    }

    //结果: main统计完成: 耗时88ms 访问次数: 1000
}

```

此时保证了数据正确的同时，效率也得到了极大的提升。

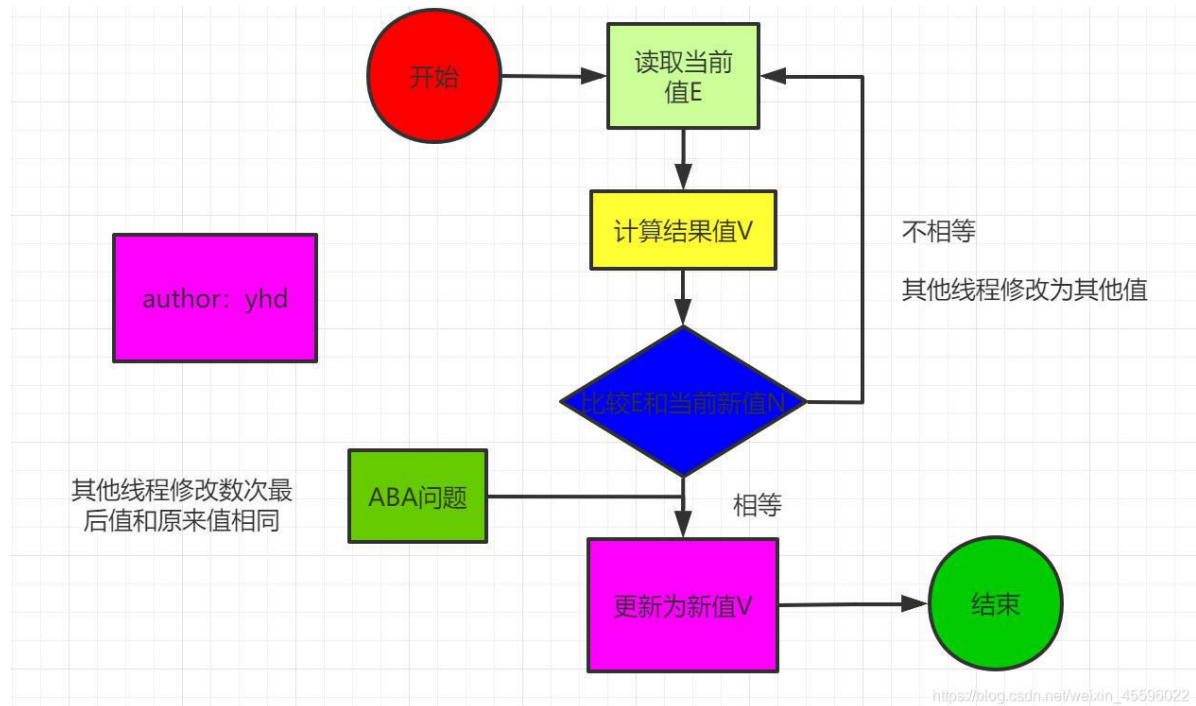
二，java中对CAS的支持

CAS 全称“CompareAndSwap”，中文翻译过来为“比较并替换”

1.定义

CAS操作包含三个操作数——内存位置（V）、期望值（A）和新值（B）。如果内存位置的值与期望值匹配，那么处理器会自动将该位置值更新为新值。否则，处理器不作任何操作。无论哪种情况，它都会在CAS指令之前返回该位置的值。

（CAS在一些特殊情况下仅返回CAS是否成功，而不提取当前值）CAS有效的说明了“我认为位置V应该包含值A；如果包含该值，则将B放到这个位置；否则，不要更改该位置的值，只告诉我这个位置现在的值即可。”



2.如何使用jdk提供的对cas的支持?

java中提供了对CAS操作的支持，具体在sun.misc.unsafe类中，声明如下：

```

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);

```

参数var1: 表示要操作的对象
参数var2: 表示要操作对象中属性地址的偏移量
参数var4: 表示需要修改数据的期望的值
参数var5: 表示需要修改为的新值

3.cas的实现原理

CAS通过调用JNI的代码实现, JNI: java Native Interface, 允许java调用其它语言。而compareAndSwapxxx系列的方法就是借助“C语言”来调用cpu底层指令实现的。
以常用的Intel x86平台来说, 最终映射到的cpu的指令为“cmpxchg”, 这是一个原子指令, cpu执行此命令时, 实现比较并替换的操作!

4.cmpxchg怎么保证多核心下的线程安全?

系统底层进行CAS操作的时候, 会判断当前系统是否为多核心系统, 如果是就给“总线”加锁, 只有一个线程会对总线加锁成功, 加锁成功之后会执行CAS操作, 也就是说CAS的原子性是平台级别的!

5.cas存在的问题

1.ABA问题

线程1准备用CAS将变量的值由A替换为B, 在此之前, 线程2将变量的值由A替换为C, 又由C替换为A, 然后线程1执行CAS时发现变量的值仍然为A, 所以CAS成功。但实际上这时的现场已经和最初不同了, 尽管CAS成功, 但可能存在潜藏的问题。

举例: 一个小偷, 把别人家的钱偷了之后又还了回来, 还是原来的钱吗, 你老婆出轨之后又回来, 还是原来的老婆吗? ABA问题也一样, 如果不好好解决就会带来大量的问题。最常见的就是资金问题, 也就是别人如果挪用了你的钱, 在你发现之前又还了回来。但是别人却已经触犯了法律。

2.循环时间长开销大

3.只能保证一个共享变量的原子操作

6.代码模拟ABA问题

```
/**
 * @author yhd
 * @createtime 2020/10/3 21:17
 * 演示ABA问题
 */
public class CasAbaDemo {

    //cas共享变量
    public static AtomicInteger a = new AtomicInteger(1);

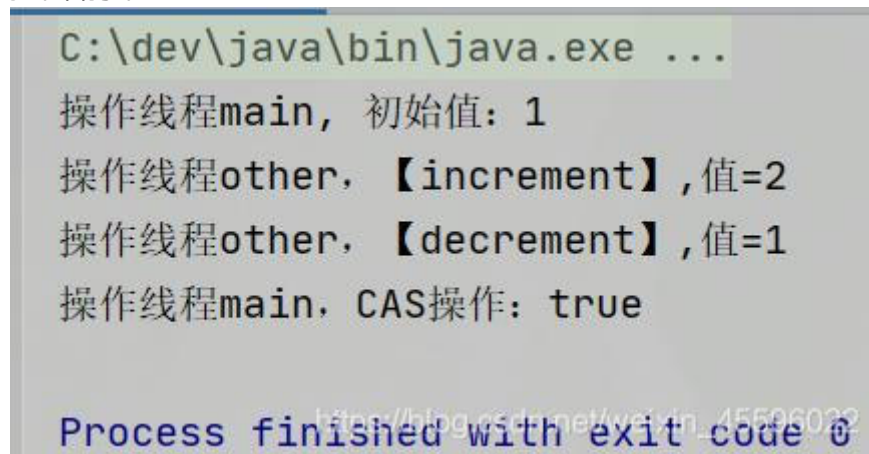
    public static void main(String[] args) {
        //主线程
        new Thread(() -> {
            System.out.println("操作线程" + Thread.currentThread().getName() + ",
初始值: " + a.get());
            try {
                //期望值
                int expectNum = a.get();
                //新值
                int newNum = a.get() + 1;
```

```

        //将cpu执行权让给干扰线程
        Thread.sleep(1000);
        boolean flag = a.compareAndSet(expectNum, newNum);
        System.out.println("操作线程" + Thread.currentThread().getName() +
", CAS操作: " + flag);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}, "main"
).start();
//干扰线程
new Thread(() -> {
    try {
        //将cpu的执行权先交给主线程
        Thread.sleep(20);
        //+1
        a.incrementAndGet();
        System.out.println("操作线程" + Thread.currentThread().getName() +
", 【increment】,值=" + a.get());
        //-1
        a.decrementAndGet();
        System.out.println("操作线程" + Thread.currentThread().getName() +
", 【decrement】,值=" + a.get());
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}, "other").start();
}
}

```

控制台打印:



```

C:\dev\java\bin\java.exe ...
操作线程main, 初始值: 1
操作线程other, 【increment】,值=2
操作线程other, 【decrement】,值=1
操作线程main, CAS操作: true
Process finished with exit code 0

```

7.jdk对于cas问题的解决

1.如何解决ABA问题

解决ABA最简单的方案就是给值加一个修改版本号，每次值变化，都会修改它的版本号，CAS操作时都去对比此版本号。

java中ABA解决方法（AtomicStampedReference）

AtomicStampedReference主要包含一个对象引用及一个可以自动更新的整数“stamp”的pair对象来解决ABA问题。

```

/**
 * @author yhd
 * @createtime 2020/10/3 21:17
 * 解决ABA问题
 */
public class CasAbaDemo {

    //cas共享变量
    public static AtomicStampedReference<Integer> a = new
AtomicStampedReference(1, 1);

    public static void main(String[] args) {
        //主线程
        new Thread(() -> {
            System.out.println("操作线程" + Thread.currentThread().getName() + ",
初始值: " + a.getReference());
            try {
                //期望值
                int expectNum = a.getReference();
                //新值
                int newNum = a.getReference() + 1;
                //期望版本
                int expectVersion = a.getStamp();
                //新的版本
                int newVersion = a.getStamp() + 1;
                //将cpu执行权让给干扰线程
                Thread.sleep(1000);
                boolean flag = a.compareAndSet(expectNum, newNum, expectVersion,
newVersion);
                System.out.println("操作线程" + Thread.currentThread().getName() +
", CAS操作: " + flag);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "main"
).start();
        //干扰线程
        new Thread(() -> {
            try {
                //将cpu的执行权先交给主线程
                Thread.sleep(20);
                //+1
                a.compareAndSet(a.getReference(), a.getReference() + 1,
a.getStamp(), a.getStamp() + 1);
                System.out.println("操作线程" + Thread.currentThread().getName() +
", 【increment】,值=" + a.getReference());
                //-1
                a.compareAndSet(a.getReference(), a.getReference() - 1,
a.getStamp(), a.getStamp() + 1);
                System.out.println("操作线程" + Thread.currentThread().getName() +
", 【decrement】,值=" + a.getReference());
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }, "other").start();
    }
}

```


2.AtomicStampedReference

1.内部类Pair

```
private static class Pair<T> {
    final T reference; //数据引用
    final int stamp; //版本号
    private Pair(T reference, int stamp) {
        this.reference = reference;
        this.stamp = stamp;
    }
    //创建一个新的pair对象
    static <T> Pair<T> of(T reference, int stamp) {
        return new Pair<T>(reference, stamp);
    }
}
```

将数据和版本号封装成一个对象进行比较，懂的人自然懂。

2.compareAndSet ()

```
public boolean compareAndSet(V expectedReference,
                             V newReference,
                             int expectedStamp,
                             int newStamp) {
    Pair<V> current = pair;
    return
        //期望引用和当前引用一致
        expectedReference == current.reference &&
        //期望版本与当前版本一致
        expectedStamp == current.stamp &&
        (
            //表示版本号对应上的同时要修改的值与原来的值相等，
            //就没有必要创建一个新的Pair对象了
            (newReference == current.reference &&
             newStamp == current.stamp)
            ||
            //否则的话，创建新的对象
            casPair(current, Pair.of(newReference, newStamp)));
}
```

3.casPair ()

```
private boolean casPair(Pair<V> cmp, Pair<V> val) {
    return UNSAFE.compareAndSwapObject(this, pairOffset, cmp, val);
}
```

实际上调用的还是unsafe类里面的compareAndSwapObject ()。

