day16【volatile关键字、原子性】

今日内容

- volatile关键字
- 原子性
- synchronized关键字

教学目标

- [ ] 能够说出volatile关键字的作用
- [ ] 能够掌握原子类AtomicInteger的使用
- [ ] 能够理解原子类的工作机制
- [ ] 能够使用同步代码块解决线程安全问题

第一章 volatile关键字

1.1 什么是volatile关键字

- volatile是一个"变量修饰符"，它只能修饰"成员变量"，它能强制线程每次从主内存获取值，并能保证此变量不会被编译器优化。
- volatile能解决变量的可见性、有序性。
- volatile不能解决变量的原子性。

1.2 volatile解决可见性

- 将1.3的线程类MyThread做如下修改：

> 1. 线程类：

```
public class MyThread extends Thread {
    public static volatile int a = 0;//增加volatile关键字
    @Override
    public void run() {
        System.out.println("线程启动，休息2秒...");
        try {
            Thread.sleep(1000 * 2);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("将a的值改为1");
        a = 1;
        System.out.println("线程结束...");
    }
}
```

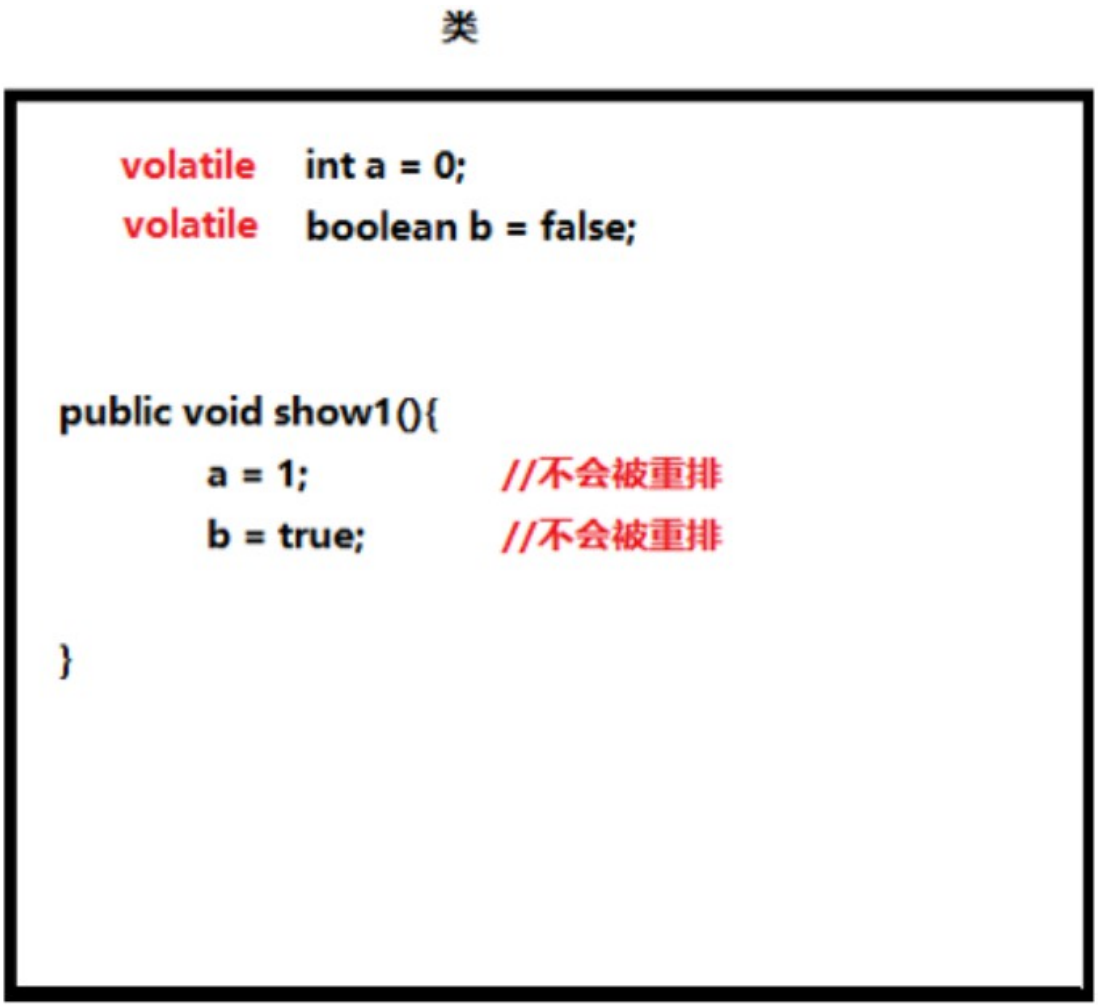
> 1. 测试类

```
public class Demo {
    public static void main(String[] args) {
        //1.启动线程
        MyThread t = new MyThread();
        t.start();

        //2.主线程继续
        while (true) {
            if (MyThread.a == 1) {
                System.out.println("主线程读到了a = 1");
            }
        }
    }
}
```

当变量被修饰为volatile时，会迫使线程每次使用此变量，都会去主内存获取，保证其可见性

1.3 volatile解决有序性

- 当变量被修饰为volatile时，会禁止代码重排



1.4 volatile不能解决原子性

- 对于示例1.5，加入volatile关键字并不能解决原子性：

> 1. 线程类：

```
public class MyThread extends  Thread {
    public static volatile int a = 0;

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            //线程1：取出a的值a=0(被暂停)
            a++;
            //写回
        }
        System.out.println("修改完毕！");
    }
}
```

> 1. 测试类：

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        //1.启动两个线程
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();

        Thread.sleep(1000);
        System.out.println("获取a最终值：" + MyThread.a);//最终结果仍然不正确。

    }
}
```

所以，volatile关键字只能解决"变量"的可见性、有序性问题，并不能解决原子性问题

第二章 原子类

2.1 原子类概述

- 在java.util.concurrent.atomic包下定义了一些对“变量”操作的“原子类”：

  1).java.util.concurrent.atomic.AtomicInteger：对int变量操作的“原子类”;

  2).java.util.concurrent.atomic.AtomicLong：对long变量操作的“原子类”;

  3).java.util.concurrent.atomic.AtomicBoolean：对boolean变量操作的“原子类”;

  它们可以保证对“变量”操作的：原子性、有序性、可见性。

2.2 AtomicInteger类示例

- 我们可以通过AtomicInteger类，来看看它们是怎样工作的

> 1. 线程类：

```
public class MyThread extends  Thread {
    //public static volatile int a = 0;//不直接使用基本类型变量

    //改用"原子类"
    public static AtomicInteger a = new AtomicInteger(0);

    @Override
    public void run() {
        for (int i = 0; i < 10000; i++) {
            // a++;
            a.getAndIncrement();//先获取，再自增1：a++
        }
        System.out.println("修改完毕！");
    }
}
```

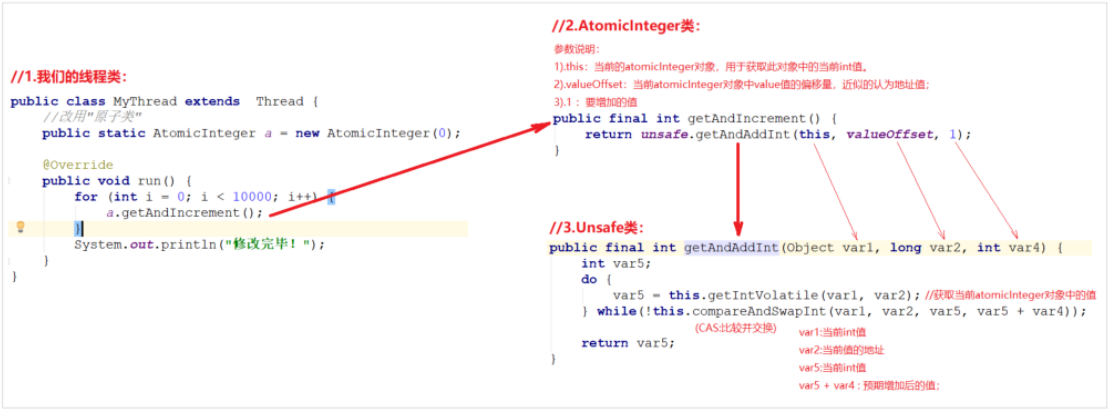
> 1. 测试类：

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        //1.启动两个线程
        MyThread t1 = new MyThread();
        MyThread t2 = new MyThread();

        t1.start();
        t2.start();

        Thread.sleep(1000);
        System.out.println("获取a最终值：" + MyThread.a.get());
    }
}
```

我们能看到，无论程序运行多少次，其结果总是正确的！

2.3 AtomicInteger类的工作原理-CAS机制

- 先来看一下调用过程：



- 在Unsafe类中，调用了一个：compareAndSwapInt()方法，此方法的几个参数：
  - var1：传入的AtomicInteger对象
  - var2：AtommicInteger内部变量的偏移地址
  - var5：之前取出的AtomicInteger中的值；
  - var5 + var4：预期结果

  此方法使用了一种"比较并交换(Compare And Swap)"的机制，它会用var1和var2先获取内存中AtomicInteger中的值，然后和传入的，之前获取的值var5做一下比较，也就是比较当前内存的值和预期的值是否一致，如果一致就修改为var5 + var4，否则就继续循环，再次获取AtomicInteger中的值，再进行比较并交换，直至成功交换为止。
- compareAndSwapInt()方法是"线程安全"的。
- 我们假设两个线程交替运行的情况，看看它是怎样工作的：
  - 初始AtomicInteger的值为0
  - 线程A执行：var5 = this.getIntVolatile(var1,var2);获取的结果为：0
  - 线程A被暂停
  - 线程B执行：var5 = this.getIntVolatile(var1,var2);获取的结果为：0
  - 线程B执行：this.compareAndSwapInt(var1,var2,var5,var5 + var4)
  - 线程B成功将AtomicInteger中的值改为1
  - 线程A恢复运行，执行：this.compareAndSwapInt(var1,var2,var5,var5 + var4)

    此时线程A使用var1和var2从AtomicInteger中获取的值为：1，而传入的var5为0，比较失败，返回false，继续循环。
  - 线程A执行：var5 = this.getIntVolatile(var1,var2);获取的结果为：1
  - 线程A执行：this.compareAndSwapInt(var1,var2,var5,var5 + var4)

    此时线程A使用var1和var2从AtomicInteger中获取的值为：1，而传入的var5为1，比较成功，将其修改为var5 + var4，也就是2，将AtomicInteger中的值改为2，结束。
- CAS机制也被称为：乐观锁。因为大部分比较的结果为true，就直接修改了。只有少部分多线程并发的情况会导致CAS失败，而再次循环。

2.4 AtomicIntegerArray类示例

- 常用的数组操作的原子类： 1).java.util.concurrent.atomic.AtomicIntegetArray:对int数组操作的原子类。

  2).java.util.concurrent.atomic.AtomicLongArray：对long数组操作的原子类。

  3).java.utio.concurrent.atomic.AtomicReferenceArray：对引用类型数组操作的原子类。
- 数组的多线程并发访问的安全性问题：

> 1. 线程类：

```
public class MyThread extends Thread {
    private static int[] intArray = new int[1000];//不直接使用数组

    @Override
    public void run() {
        for (int i = 0; i < arr.length(); i++) {
            intArray[i]++;
        }
    }
}
```

> 1. 测试类：

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();//创建1000个线程，每个线程为数组的每个元素+1
        }

        Thread.sleep(1000 * 5);//让所有线程执行完毕

        System.out.println("主线程休息5秒醒来");
        for (int i = 0; i < MyThread.intArray.length(); i++) {
            System.out.println(MyThread.intArray[i]);
        }
    }
}
```

正常情况，数组的每个元素最终结果应为：1000，而实际打印：

```
1000
1000
1000
1000
999
999
999
999
999
999
999
999
1000
1000
1000
1000
```

可以发现，有些元素并不是1000.

- 为保证数组的多线程安全，改用AtomicIntegerArray类，演示：

> 1. 线程类：

```
public class MyThread extends Thread {
    private static int[] intArray = new int[1000];//定义一个数组
    //改用原子类，使用数组构造
    public static AtomicIntegerArray arr = new AtomicIntegerArray(intArray);
    @Override
    public void run() {
        for (int i = 0; i < arr.length(); i++) {
            arr.addAndGet(i, 1);//将i位置上的元素 + 1
        }
    }
}
```

1. 测试类：

```
public class Demo {
    public static void main(String[] args) throws InterruptedException {
        for (int i = 0; i < 1000; i++) {
            new MyThread().start();
        }
        Thread.sleep(1000 * 5);//让所有线程执行完毕

        System.out.println("主线程休息5秒醒来");
        for (int i = 0; i < MyThread.arr.length(); i++) {
            System.out.println(MyThread.arr.get(i));
        }
    }
}
```

先在能看到，每次运行的结果都是正确的。

第三章 synchronized关键字

3.1 多行代码的原子性问题

- 之前的AtomicInteger类只能保证"变量"的原子性操作，而对多行代码进行"原子性"操作，使用AtomicInteger类就不能达到效果了。
- 我们通过一个案例，演示线程的安全问题：

  电影院要卖票，我们模拟电影院的卖票过程。假设要播放的电影是 “葫芦娃大战奥特曼”，本次电影的座位共100个(本场电影只能卖100张票)。

  我们来模拟电影院的售票窗口，实现多个窗口同时卖 “葫芦娃大战奥特曼”这场电影票(多个窗口一起卖这100张票)需要窗口，采用线程对象来模拟；需要票，Runnable接口子类来模拟。

模拟票：

```
public class Ticket implements Runnable {
    private int ticket = 100;
    /*
     * 执行卖票操作
```

```
     */
    @Override
    public void run() {
        //每个窗口卖票的操作
        //窗口 永远开启
        while (true) {
            if (ticket > 0) {//有票 可以卖
                //出票操作
                //使用sleep模拟一下出票时间
                try {
                    Thread.sleep(100);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                //获取当前线程对象的名字
                String name = Thread.currentThread().getName();
                System.out.println(name + "正在卖:" + ticket--);
            }
        }
    }
}
```

测试类：

```
public class Demo {
    public static void main(String[] args) {
        //创建线程任务对象
        Ticket ticket = new Ticket();
        //创建三个窗口对象
        Thread t1 = new Thread(ticket, "窗口1");
        Thread t2 = new Thread(ticket, "窗口2");
        Thread t3 = new Thread(ticket, "窗口3");

        //同时卖票
        t1.start();
        t2.start();
        t3.start();
    }
}
```

结果中有一部分这样现象：

发现程序出现了两个问题：

1. 相同的票数,比如5这张票被卖了两回。
2. 不存在的票，比如0票与-1票，是不存在的。

这种问题，几个窗口(线程)票数不同步了，这种问题称为线程不安全。

> 线程安全问题都是由全局变量及静态变量引起的。而每个线程操作这个变量都需要很多步骤：获取变量的值、打印变量的值、更改变量的值，而一个线程在执行某一步骤时都可能被暂停，而另一个线程会执行，这同样会导致多个线程访问同一个变量，最终导致这个变量的值不准确。

3.2 synchronized关键字概述

- synchronized关键字：表示“同步”的。它可以对“多行代码”进行“同步”——将多行代码当成是一个完整的整体，一个线程如果进入到这个代码块中，会全部执行完毕，执行结束后，其它线程才会执行。这样可以保证这多行的代码作为完整的整体，被一个线程完整的执行完毕。
- synchronized被称为“重量级的锁”方式，也是“悲观锁”——效率比较低。
- synchronized有几种使用方式： a).同步代码块

  b).同步方法【常用】

当我们使用多个线程访问同一资源的时候，且多个线程中对资源有写的操作，就容易出现线程安全问题。

要解决上述多线程并发访问一个资源的安全性问题:也就是解决重复票与不存在票问题，Java中提供了同步机制(**synchronized**)来解决。

根据案例简述：

> 窗口1线程进入操作的时候，窗口2和窗口3线程只能在外等着，窗口1操作结束，窗口1和窗口2和窗口3才有机会进入代码去执行。也就是说在某个线程修改共享资源的时候，其他线程不能修改该资源，等待修改完毕同步之后，才能去抢夺CPU资源，完成对应的操作，保证了数据的同步性，解决了线程不安全的现象。

3.3 同步代码块

- **同步代码块**：`synchronized`关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

格式:

```
synchronized(同步锁){
      需要同步操作的代码
}
```

同步锁:

对象的同步锁只是一个概念,可以想象为在对象上标记了一个锁.

1. 锁对象 可以是任意类型。
2. 多个线程对象 要使用同一把锁。

> 注意:在任何时候,最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着(BLOCKED)。

使用同步代码块解决代码：

```
public class Ticket implements Runnable{
    private int ticket = 100;

    Object lock = new Object();
    /*
     * 执行卖票操作
     */
    @Override
    public void run() {
```

```
        //每个窗口卖票的操作
        //窗口 永远开启
        while(true){
            synchronized (lock) {
                if(ticket>0){//有票 可以卖
                    //出票操作
                    //使用sleep模拟一下出票时间
                    try {
                        Thread.sleep(50);
                    } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                    }
                    //获取当前线程对象的名字
                    String name = Thread.currentThread().getName();
                    System.out.println(name+"正在卖:"+ticket--);
                }
            }
        }
    }
}
```

当使用了同步代码块后，上述的线程的安全问题，解决了。