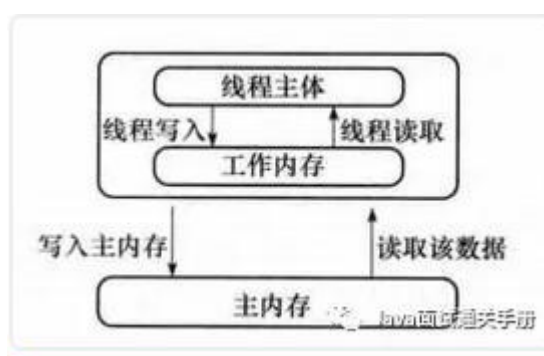


Java 多线程学习 (三) volatile 关键字

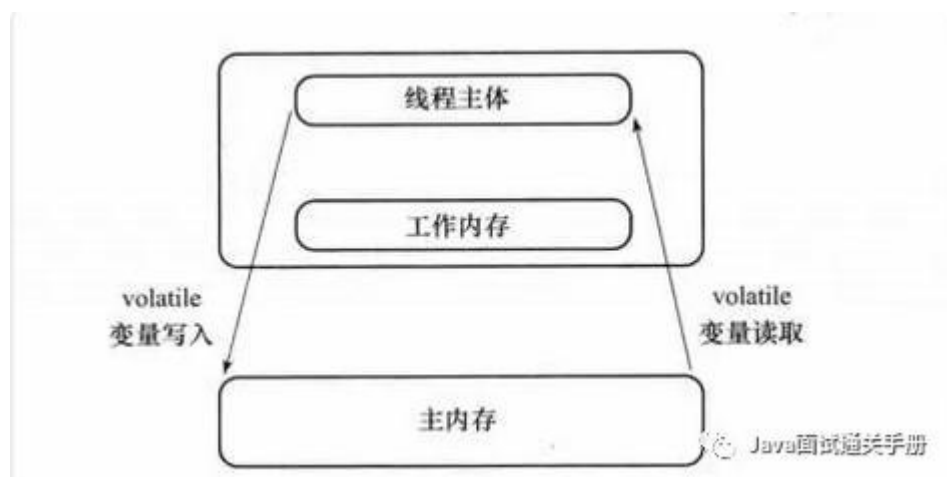
一 简介

Java 中的“volatile 关键字”关键字：

在 JDK1.2 之前，Java 的内存模型实现总是从**主存（即共享内存）**读取变量，是不需要进行特别的注意的。而在当前的 Java 内存模型下，线程可以把变量保存**本地内存**（比如机器的寄存器）中，而不是直接在主存中进行读写。这就可能造成一个线程在主存中修改了一个变量的值，而另外一个线程还继续使用它在寄存器中的变量值的拷贝，造成**数据的不一致**。



要解决这个问题，就需要把**变量声明为 volatile**，这就指示 JVM，这个变量是不稳定的，每次使用它都到**主存**中进行读取。



二 volatile 关键字的可见性

volatile 修饰的成员变量在每次被线程访问时，都强迫从**主存（共享内存）**中重读该成员变量的值。而且，当成员变量发生变化时，**强迫线程将变化值回写到主存（共享内存）**。这样在任何时刻，**两个不同的线程总是看到某个成员变量的同一个值**，这样也就保证了同步数据的**可见性**。

实例 1:

```
public class RunThread extend Thread{
    private boolean isRunning = true;
    int m;
    public boolean isRunning() {
        return isRunning;
    }
    public void setRunning(boolean isRunning) {
        this.isRunning = isRunning;
    }
    @Override
    public void run() {
        System.out.println("进入 run 了");
        while (isRunning == true) {
            int a=2;
            int b=3;
            int c=a+b;
            m=c;
        }
        System.out.println(m);
        System.out.println("线程被停止了! ");
    }
}
```

```
public class RunTest {
    public static void main(String[] args) throws InterruptedException {
        RunThread thread = new RunThread();
        thread.start();
        Thread.sleep(1000);
        thread.setRunning(false);
        System.out.println("已经赋值为 false");
    }
}
```

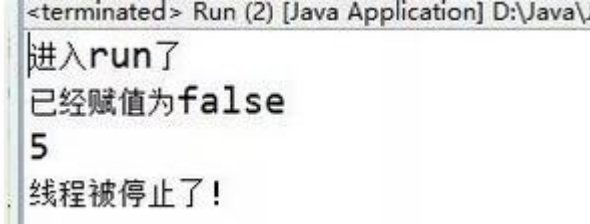


RunThread 类中的 isRunning 变量没有加上 **volatile 关键字**时, 运行以上

代码会出现**死循环**，这是因为 isRunning 变量虽然被修改但是没有被写到**主存**中，这也就导致该线程在**本地内存**中的值一直为 true，这样就导致了死循环的产生。

解决办法也很简单：isRunning 变量前加上 **volatile 关键字**即可。

```
volatile private boolean isRunning = true;
```



注意!!!

假如你把 while 循环代码里加上任意一个输出语句或者 sleep 方法你会发现**死循环也会停止**，不管 isRunning 变量是否被加上了上 volatile 关键字。

原因：JVM 会尽力保证内存的可见性，即便这个变量没有加同步关键字。换句话说，只要 CPU 有时间，JVM 会尽力去保证变量值的更新。这种与 volatile 关键字的不同在于，**volatile 关键字会强制的保证线程的可见性**。而不加这个关键字，**JVM 也会尽力去保证可见性**，但是如果 CPU 一直有其他的事情在处理，它也没办法。最开始的代码，一直处于死循环中，CPU 处于一直占用的状态，这个时候 CPU 没有时间，JVM 也不能强制要求 CPU 分点时间去取最新的变量值。而**加了输出或者 sleep 语句之后，CPU 就有可能有时间去保证内存的可见性，于是 while 循环可以被终止。**

三 volatile 关键字能保证原子性吗？

volatile 无法保证对变量原子性的(非原子性操作一定不是原子性的,如: ++i; s= x+y 等)。

实例 2:

```
public class MyThread extends Thread {  
    volatile public static int count;  
    private static void addCount() {  
        for (int i = 0; i < 100; i++) {  
            count=i;  
        }  
        System.out.println("count=" + count);  
    }  
}
```

```

    }
    @Override
    public void run() {
        addCount();
    }
}

public class Run {
    public static void main(String[] args) {
        MyThread[] mythreadArray = new MyThread[100];
        for (int i = 0; i < 100; i++) {
            mythreadArray[i] = new MyThread();
        }
        for (int i = 0; i < 100; i++) {
            mythreadArray[i].start();
        }
    }
}

```

上面的“count=i;”是一个原子操作，但是运行结果大部分都是正确结果 99，但是也有部分不是 99 的结果。

```

count=99
count=99
count=99
count=99
count=99
count=99
count=74
count=99
count=99
count=99
count=99
count=99
count=99
count=99
count=99

```

解决办法：

使用 **synchronized 关键字加锁**。(这只是一种方法, Lock 和 AtomicInteger

原子类都可以，因为之前学过 `synchronized` 关键字，所以我们使用 `synchronized` 关键字的方法），这样运行输出的 `count` 就都为 99 了，所以要保证数据的原子性还是要使用 `synchronized` 关键字。

四 `synchronized` 关键字和 `volatile` 关键字比较

(1) `volatile` 关键字是线程同步的轻量级实现，所以 `volatile` 性能肯定比 `synchronized` 关键字要好。但是 `volatile` 关键字只能用于变量而 `synchronized` 关键字可以修饰方法以及代码块。`synchronized` 关键字在 JavaSE1.6 之后进行了主要包括为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁以及其它各种优化之后执行效率有了显著提升，实际开发中使用 `synchronized` 关键字还是更多一些。

(2) 多线程访问 `volatile` 关键字不会发生阻塞，而 `synchronized` 关键字可能会发生阻塞

(3) `volatile` 关键字能保证数据的可见性，但不能保证数据的原子性。`synchronized` 关键字两者都能保证。

(4) `volatile` 关键字用于解决变量在多个线程之间的可见性，而 `synchronized` 关键字解决的是多个线程之间访问资源的同步性。