

# volatile不保证原子性以及解决方案

## 1、原子性的定义

什么是原子性,什么是原子性操作?

举个例子:

A想要从自己的帐户中转1000块钱到B的帐户里。那个从A开始转帐,到转帐结束的这一个过程,称之为一个事务。在这个事务里,要做如下操作:

- 1. 从A的帐户中减去1000块钱。如果A的帐户原来有3000块钱,现在就变成2000块钱了。
- 2. 在B的帐户里加1000块钱。如果B的帐户如果原来有2000块钱,现在则变成3000块钱了。

如果在A的帐户已经减去了1000块钱的时候,忽然发生了意外,比如停电什么的,导致转帐事务意外终止了,而此时B的帐户里还没有增加1000块钱。那么,我们称这个操作失败了,要进行回滚。回滚就是回到事务开始之前的状态,也就是回到A的帐户还没减1000块的状态,B的帐户的原来的状态。此时A的帐户仍然有3000块,B的帐户仍然有2000块。

我们把这种要么一起成功(A帐户成功减少1000,同时B帐户成功增加1000),

要么一起失败(A帐户回到原来状态,B帐户也回到原来状态)的操作叫原子性操作。

如果把一个事务可看作是一个程序,它要么完整的被执行,要么完全不执行。这种特性就叫原子性。

## 2、volatile不保证原子性代码验证

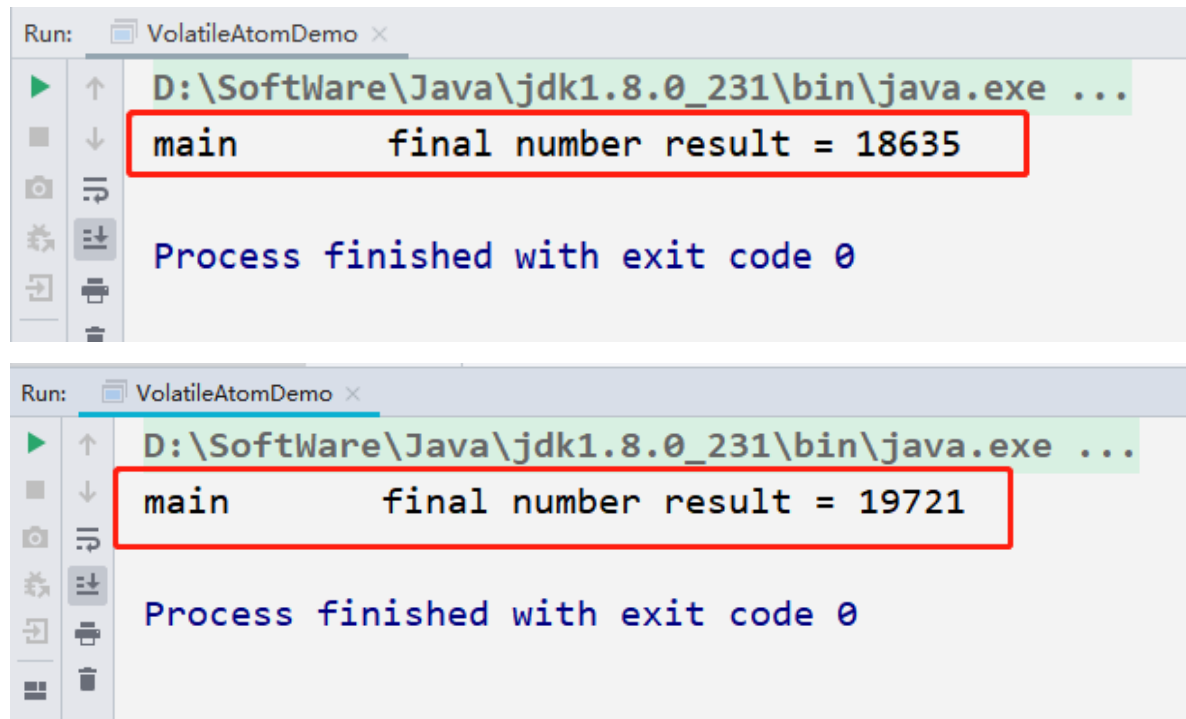
```
package com.hyq.test;public class VolatileAtomDemo {
    // volatile不保证原子性
    // 原子性: 保证数据一致性、完整性
    volatile int number = 0;
    public void addPlusPlus() {
        number++;
    }
    public static void main(String[] args) {
        VolatileAtomDemo volatileAtomDemo = new VolatileAtomDemo();
        for (int j = 0; j < 20; j++) {
            new Thread(() -> {
                for (int i = 0; i < 1000; i++) {
                    volatileAtomDemo.addPlusPlus();
                }
            }, String.valueOf(j)).start();
        }
        // 后台默认两个线程: 一个是main线程, 一个是gc线程
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        // 如果volatile保证原子性的话, 最终的结果应该是20000
        // 但是每次程序执行结果都不等于20000
        System.out.println(Thread.currentThread().getName() +
```

```

        "\t final number result = " +
        volatileAtomDemo.number);
    }
}

```

代码执行结果如下：多次执行结果证明volatile不保证原子性



### 3、volatile不保证原子性原理分析

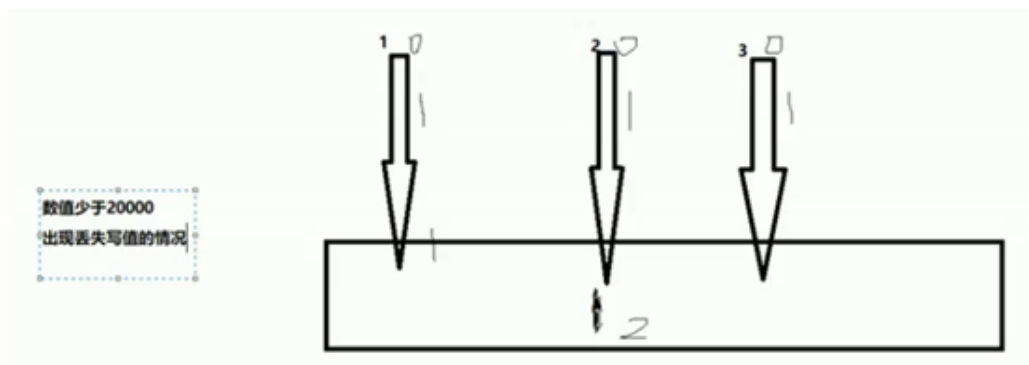
number++被拆分成3个指令

执行GETFIELD拿到主内存中的原始值number

执行IADD进行加1操作

执行PUTFIELD把工作内存中的值写回主内存中

当多个线程并发执行PUTFIELD指令的时候，会出现写回主内存覆盖问题，所以才会导致最终结果不为20000，volatile不能保证原子性。



### 4、解决volatile不保证原子性问题

a、方法前加synchronized解决

```
public synchronized void addPlusPlus() {
    number++;
}
```

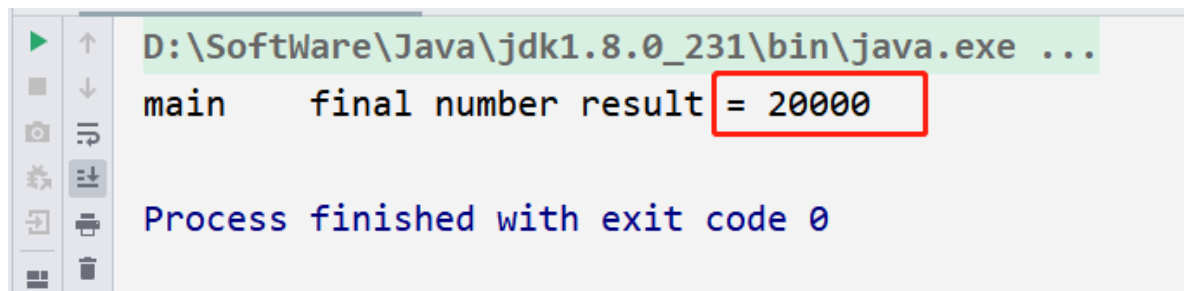
#### b、加锁解决

```
Lock lock = new ReentrantLock();
public void addPlusPlus() {
    lock.lock();
    number++;
    lock.unlock();
}
```

#### c、原子类解决

```
package com.hyq.test;
import java.util.concurrent.atomic.AtomicInteger;
public class volatileSolveAtomDemo {
    // 原子Integer类型，保证原子性
    private AtomicInteger atomicNumber = new AtomicInteger();
    // 底层通过CAS保证原子性
    public void addPlusPlus() {
        atomicNumber.getAndIncrement();
    }
    public static void main(String[] args) {
        volatileSolveAtomDemo volatileSolveAtomDemo = new
        volatileSolveAtomDemo();
        for (int j = 0; j < 20; j++) {
            new Thread(() -> {
                for (int i = 0; i < 1000; i++) {
                    volatileSolveAtomDemo.addPlusPlus();
                }
            }, String.valueOf(j)).start();
        }
        // 后台默认两个线程：一个是main线程，一个是gc线程
        while (Thread.activeCount() > 2) {
            Thread.yield();
        }
        // 因为volatile不保证原子性，所以选择原子类AtomicInteger来解决volatile不保证原子
        // 性问题
        // 最终每次程序执行结果都等于20000
        System.out.println(Thread.currentThread().getName() +
            "\tfinal number result = " +
            volatileSolveAtomDemo.atomicNumber.get());
    }
}
```

代码执行结果如下：多次执行结果证明原子类是可以解决volatile不保证原子性问题



The screenshot shows a Java IDE's console window. On the left is a vertical toolbar with icons for running, stepping through, and other debugging actions. The main area of the console displays the following text:

```
D:\SoftWare\Java\jdk1.8.0_231\bin\java.exe ...  
main    final number result = 20000  
  
Process finished with exit code 0
```

The first line, representing the command prompt, is highlighted in light green. The output line, showing the final result of the program, has the value '20000' enclosed in a red rectangular box.