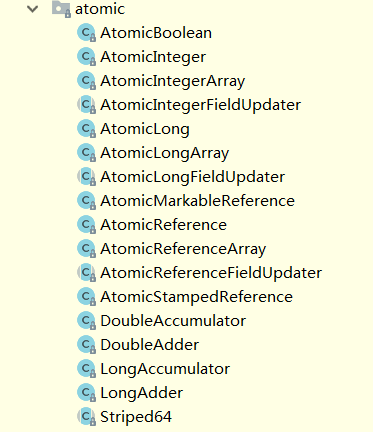
## JUC学习之Atomic

### Atomic是什么

顾名思义，Atomic是原子类。原子类，那里面的操作都符合原子类了。打开JUC包（JDK1.8版本），我们发现里面有这些东西：



相比之前的版本，JDK1.8里面Atomic包里多了DoubleAccumulator，DoubleAdder，LongAccumulater，LongAdder几个类，据说这几个新加类进一步提升了Double,Long原子类的性能，个人尚未使用过，留待以后讨论。

### Atomic能用来干嘛

目前只用了AtomicInteger，通常被用来做计数器，计数器有什么用，用来限流啊。统计单位时间（通常是1S）的请求数量，到了这个数值就不再接受请求，而是把请求放到下一秒钟去处理，是常见的操作。

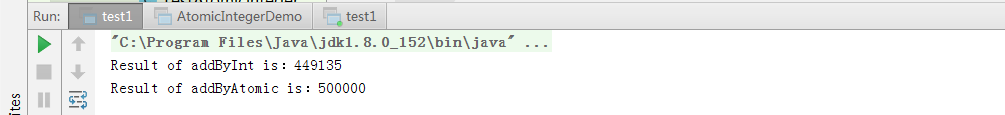
大家也看到了，Atomic类里面实现的都是简单类型，像Boolean,Int,Long这种，基本上我们在并发场景操作以上类型的数据就可以考虑用对应的原子类了。

好了，我们看一下计数器基本实现。

**public static void** main(String[] args) {  
 **final** MyInc myInc = **new** MyInc();  
 **for** (**int** i = 1; i <= 50; i++) {  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 **for** (**int** j = 1; j <= 10000; j++) {  
 myInc.addByInt();  
 myInc.addByAtomic();  
 }  
 }  
 }).start();  
 }  
 *//需要等new的50个线程执行结束* **while** (Thread.*activeCount*() > 2) {  
 Thread.*yield*();  
 }  
 System.***out***.println(**"Result of addByInt is："** + myInc.**num**);  
 System.***out***.println(**"Result of addByAtomic is："** + myInc.**atomicInteger**);  
}

**public class** MyInc {  
 **volatile int num** = 0; *//volatile不能保证原子性* AtomicInteger **atomicInteger** = **new** AtomicInteger();  
  
 **public void** addByInt() {  
 **num**++;  
 }  
  
 **public void** addByAtomic() {  
 **atomicInteger**.getAndIncrement();  
 }  
}

运行后结果：



### 为啥要用原子类

就是由于int的自增运算是非原子性的啊。

从上面的结果我们也看到了，没有得到预期的结果500000，这里我们可以看一下什么是原子操作。

根据周志明JVM一书的描述。原子操作有如下八种，这里照搬一下原书的定义：

lock（锁定）：作用于主内存的变量，它把一个变量标识为一条线程独占的状态。

unlock（解锁）：作用于主内存的变量，它把一个处于锁定状态的变量释放出来，释放后的变量才可以被其他线程锁定。

read（读取）：作用于主内存的变量，它把一个变量的值从主内存传输到线程的工作内存中，以便以后的load动作使用。

load（载入）：作用于工作内存的变量，它把read操作从主内存中得到的变量值放入工作内存的变量副本中。

use（使用）：作用于工作内存的变量，它把工作内存中一个变量的值传递给执行引擎，每当虚拟机遇到一个需要使用变量的值的字节码指令时将会执行这个操作。

assign（赋值）：作用于工作内存的变量，它把一个从执行引擎接收的值赋给工作内存的变量，每当虚拟机遇到一个给变量赋值的字节码指令时执行这个操作。

store（存储）：作用于工作内存的变量，它把工作内存中一个变量的值传送到主内存中，以便随后的write操作使用。

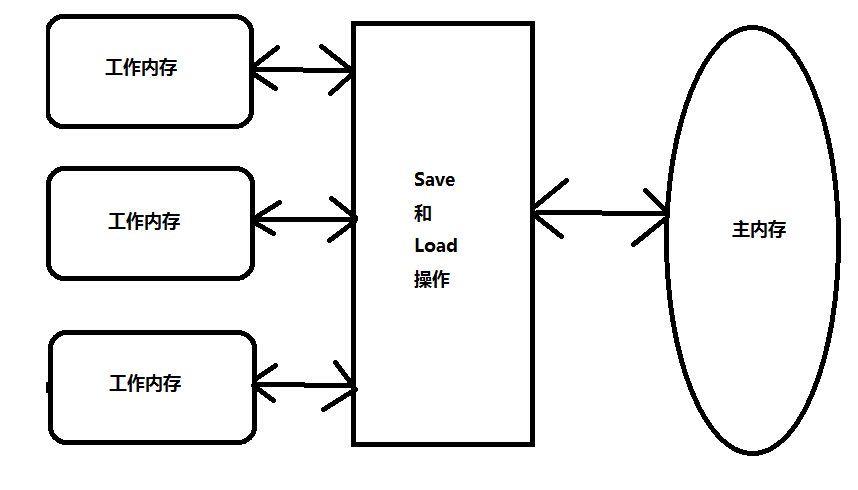
write（写入）：作用于主内存的变量，它把store操作从工作内存中得到的变量的值放入主内存的变量中。

好了，你看到了吧，现在你可以想象到了。int的自增运算至少是有读取，赋值，载入三个原子操作的，没有拿到预期结果，就在于某个线程执行到载入还没完成时的时候另一个线程已经读取了，这样你相当于读取了一个“旧值”。

而为啥原子类就没有这个问题呢，因为CAS机制啊（这里不考虑ABA问题），其实就是循环不断的比较看获取的是“旧值”还是“新值”，如果是旧的就重新读取一次，只有获取到了“新值”才载入主内存，这样保证了以上说到的三步都完成了，下个线程才执行。这样就保证了原子性。

#### 工作内存和主内存

这里简单说下上面提到的工作内存和主内存，他们大概是这样的关系：



基本上可以把工作内存看成是单个Java线程持有的内存，主内存是机器里还能够利用的物理内存。Java的这种内存设计方式从根本上决定了多线程需要处理数据一致性问题，或者说线程为啥需要同步就是因它而产生。