# 并发包java.util.concurrent.\*

线程的相关类都在该包下

<https://www.cnblogs.com/jjfan0327/p/6866488.html>

Java内存模型和ConcurrentHashMap 1.7源码分析

# Atomic原子

例如2000个线程计算同时加1000次，每次加1，那么最后的结果应该是20000。

## 方式一

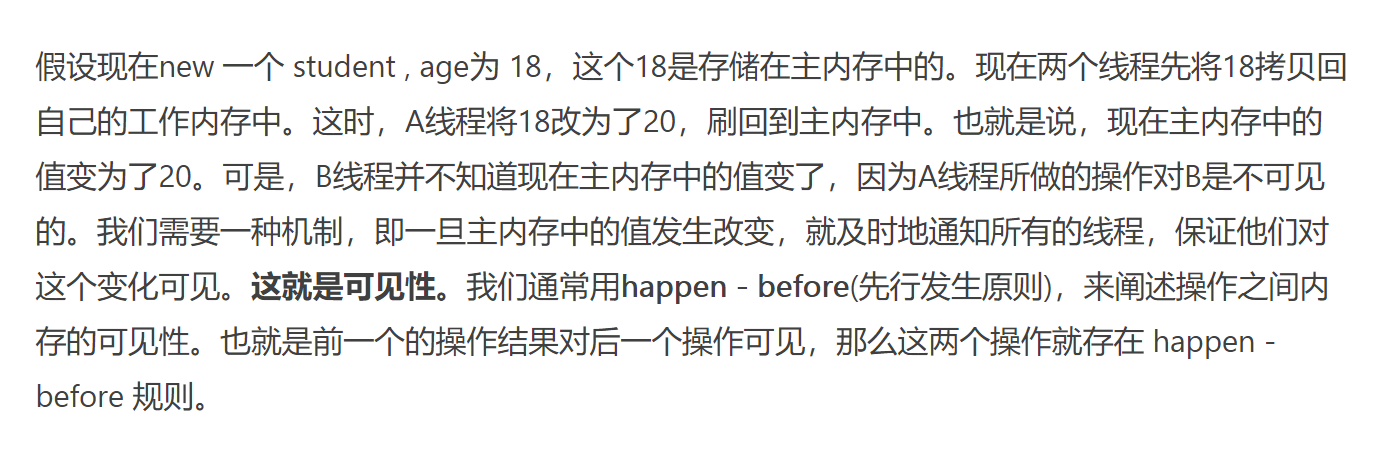
|  |
| --- |
| **package** com.huayu.test4;  **public** **class** AtomicIntegerTest {    **private** **static** **final** **int** ***THREADS\_CONUT*** = 20;    **public** **static** **int** *count* = 0;  **public** **static** **void** increase() {  *count*++;  }    **public** **static** **void** main(String[] args) {  Thread[] threads = **new** Thread[***THREADS\_CONUT***];  **for** (**int** i = 0; i < ***THREADS\_CONUT***; i++) {  threads[i] = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 1000; i++) {  *increase*();  }  }  });  threads[i].start();  }  **while** (Thread.*activeCount*() > 1) {  Thread.*yield*();  }  System.***out***.println(*count*);  }  } |

这种方式计算的结果肯定不对，而且每次结果也不相同。

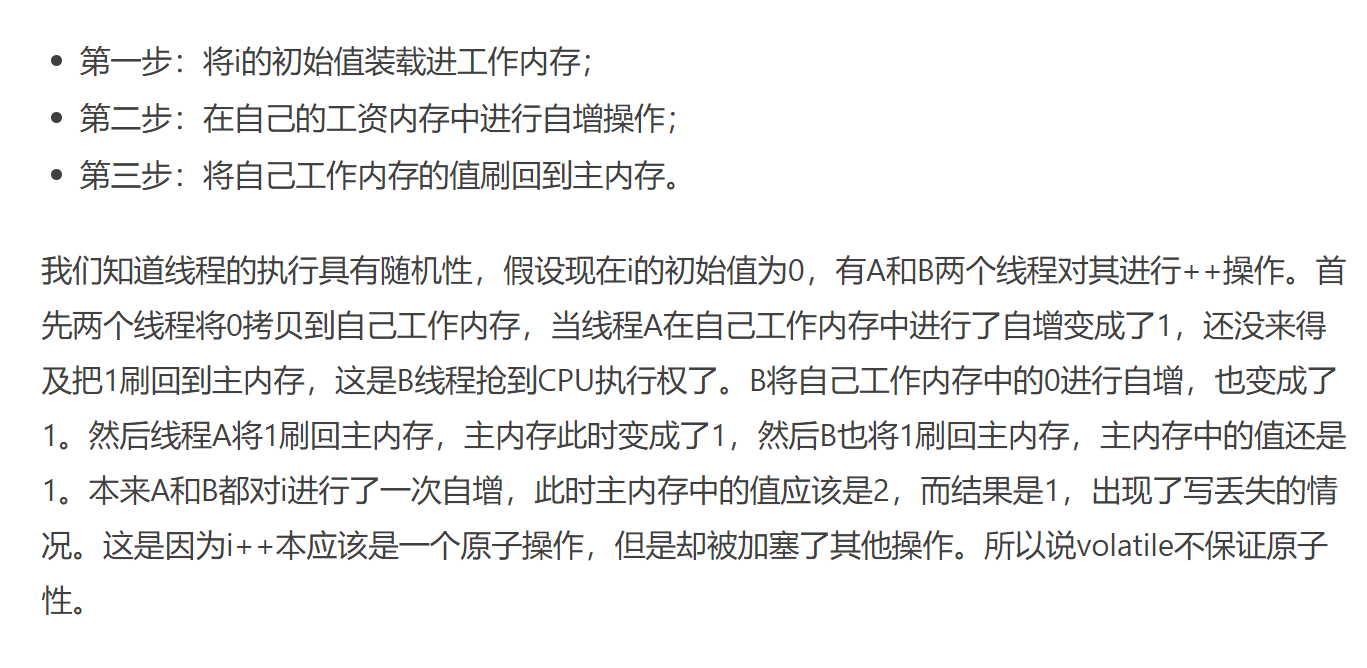
## 方式二，用volatile

|  |
| --- |
| count用volatile修饰，同样也是有问题的，和方式一一样，为什么？Volatile修改的变量在修改时不是对其他线程可见的吗。Volatile有可见性，但是没有原子性。 |

可见性：



原子性：



## 方式三

|  |
| --- |
| **package** com.huayu.test4;  **import** java.util.concurrent.atomic.AtomicInteger;  **public** **class** AtomicIntegerTest {    **private** **static** **final** **int** ***THREADS\_CONUT*** = 20;      **public** **static** AtomicInteger *count* = **new** AtomicInteger();    **public** **static** **void** increase() {  *count*.incrementAndGet();  }    **public** **static** **void** main(String[] args) {  Thread[] threads = **new** Thread[***THREADS\_CONUT***];  **for** (**int** i = 0; i < ***THREADS\_CONUT***; i++) {  threads[i] = **new** Thread(**new** Runnable() {  @Override  **public** **void** run() {  **for** (**int** i = 0; i < 1000; i++) {  *increase*();  }  }  });  threads[i].start();  }    **while** (Thread.*activeCount*() > 1) {  Thread.*yield*();  }  System.***out***.println(*count*);  }  } |

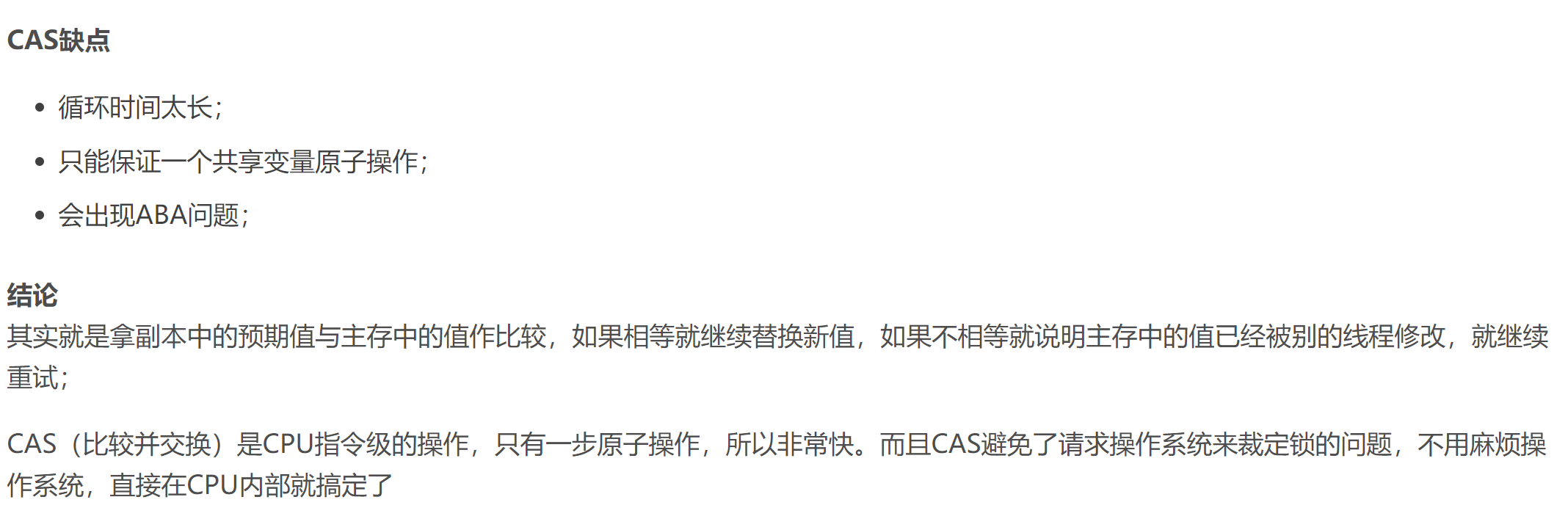
除了使用AtomicInteger之外，还可以在increase方法上加锁synchronized，显然加锁的方式不好，太重量级了。

AtomicInteger为什么可以？因为采用了CAS算法。

## CAS算法

Compare and Swap，即比较再交换

例如主内存中的值为50，线程1与线程2同时对这个数进行修改，那么线程1和线程2 同时读取到该数50，然后线程1进行了+1并且刷新到了主内存中，此时线程2也进行 了修改+1，也要刷新到主内存中，如果操作成功，两次操作相当于成功了一次。结果还 是51.但是CAS算法保证线程2在刷新到主内存之前进行一次比较，就是比较当时取出 的数50和现在主内存中的数据是否一致，如果一直就刷新到内存中，如果不一致则不 能刷新到主内存中，重新取数进行操作。线程1的操作流程也是如此。所有的线程操作 都是这样，先比较再交换。



## AtomicBoolean

# 多线程锁机制



# ForkJoinPool合并和分叉

Fork就是将任务分开执行，分成一个一个的小任务进行执行

Join就是将执行结束的子任务的结果合并。