**参考：**[**http://www.blogjava.net/xylz/archive/2010/07/08/325587.html**](http://www.blogjava.net/xylz/archive/2010/07/08/325587.html)

[**《深入浅出 Java Concurrency》目录**](http://www.blogjava.net/xylz/archive/2010/07/08/325587.html)

这是一份完整的Java 并发整理笔记，记录了我最近几年学习Java并发的一些心得和体会。

1. [J.U.C 整体认识](http://www.blogjava.net/xylz/archive/2010/06/30/324915.html)
2. [原子操作 part 1 从AtomicInteger开始](http://www.blogjava.net/xylz/archive/2010/07/01/324988.html)
3. [原子操作 part 2 数组、引用的原子操作](http://www.blogjava.net/xylz/archive/2010/07/02/325079.html)
4. [原子操作 part 3 指令重排序与happens-before法则](http://www.blogjava.net/xylz/archive/2010/07/03/325168.html)
5. [原子操作 part 4 CAS操作](http://www.blogjava.net/xylz/archive/2010/07/04/325206.html)
6. [锁机制 part 1 Lock与ReentrantLock](http://www.blogjava.net/xylz/archive/2010/07/05/325274.html)
7. [锁机制 part 2 AQS](http://www.blogjava.net/xylz/archive/2010/07/06/325390.html)
8. [锁机制 part 3 加锁的原理 (Lock.lock)](http://www.blogjava.net/xylz/archive/2010/07/07/325410.html)
9. [锁机制 part 4 锁释放与条件变量 (Lock.unlock And Condition)](http://www.blogjava.net/xylz/archive/2010/07/08/325540.html)
10. [锁机制 part 5 闭锁 (CountDownLatch)](http://www.blogjava.net/xylz/archive/2010/07/09/325612.html)
11. [锁机制 part 6 CyclicBarrier](http://www.blogjava.net/xylz/archive/2010/07/12/325913.html)
12. [锁机制 part 7 信号量 (Semaphore)](http://www.blogjava.net/xylz/archive/2010/07/13/326021.html)
13. [锁机制 part 8 读写锁 (ReentrantReadWriteLock) (1)](http://www.blogjava.net/xylz/archive/2010/07/14/326080.html)
14. [锁机制 part 9 读写锁 (ReentrantReadWriteLock) (2)](http://www.blogjava.net/xylz/archive/2010/07/15/326152.html)
15. [锁机制 part 10 锁的一些其它问题](http://www.blogjava.net/xylz/archive/2010/07/16/326246.html)
16. [并发容器 part 1 ConcurrentMap (1)](http://www.blogjava.net/xylz/archive/2010/07/19/326527.html)
17. [并发容器 part 2 ConcurrentMap (2)](http://www.blogjava.net/xylz/archive/2010/07/20/326584.html)
18. [并发容器 part 3 ConcurrentMap (3)](http://www.blogjava.net/xylz/archive/2010/07/20/326661.html)
19. [并发容器 part 4 并发队列与Queue简介](http://www.blogjava.net/xylz/archive/2010/07/21/326723.html)
20. [并发容器 part 5 ConcurrentLinkedQueue](http://www.blogjava.net/xylz/archive/2010/07/23/326934.html)
21. [并发容器 part 6 可阻塞的BlockingQueue (1)](http://www.blogjava.net/xylz/archive/2010/07/24/326988.html)
22. [并发容器 part 7 可阻塞的BlockingQueue (2)](http://www.blogjava.net/xylz/archive/2010/07/27/327265.html)
23. [并发容器 part 8 可阻塞的BlockingQueue (3)](http://www.blogjava.net/xylz/archive/2010/07/30/327582.html)
24. [并发容器 part 9 双向队列集合 Deque](http://www.blogjava.net/xylz/archive/2010/08/12/328587.html)
25. [并发容器 part 10 双向并发阻塞队列 BlockingDeque](http://www.blogjava.net/xylz/archive/2010/08/18/329227.html)
26. [并发容器 part 11 Exchanger](http://www.blogjava.net/xylz/archive/2010/11/22/338733.html)
27. [并发容器 part 12 线程安全的List/Set CopyOnWriteArrayList/CopyOnWriteArraySet](http://www.blogjava.net/xylz/archive/2010/11/23/338853.html)
28. [线程池 part 1 简介](http://www.blogjava.net/xylz/archive/2010/12/19/341098.html)
29. [线程池 part 2 Executor 以及Executors](http://www.blogjava.net/xylz/archive/2010/12/21/341281.html)
30. [线程池 part 3 Executor 生命周期](http://www.blogjava.net/xylz/archive/2011/01/04/342316.html)
31. [线程池 part 4 线程池任务拒绝策略](http://www.blogjava.net/xylz/archive/2011/01/08/342609.html)
32. [线程池 part 5 周期性任务调度](http://www.blogjava.net/xylz/archive/2011/01/10/342738.html)
33. [线程池 part 6 线程池的实现及原理 (1)](http://www.blogjava.net/xylz/archive/2011/01/18/343183.html)
34. [线程池 part 7 线程池的实现及原理 (2)](http://www.blogjava.net/xylz/archive/2011/02/11/344091.html)
35. [线程池 part 8 线程池的实现及原理 (3)](http://www.blogjava.net/xylz/archive/2011/02/13/344207.html)
36. [线程池 part 9 并发操作异常体系](http://www.blogjava.net/xylz/archive/2011/07/12/354206.html)
37. [并发总结 part 1 死锁与活跃度](http://www.blogjava.net/xylz/archive/2011/12/29/365149.html)
38. [并发总结 part 2 常见的并发场景](http://www.blogjava.net/xylz/archive/2011/12/29/367480.html)
39. [并发总结 part 3 常见的并发陷阱](http://www.blogjava.net/xylz/archive/2011/12/30/367592.html)
40. [并发总结 part 4  性能与伸缩性](http://www.blogjava.net/xylz/archive/2011/12/31/367641.html)
41. [捕获Java线程池执行任务抛出的异常](http://www.blogjava.net/xylz/archive/2013/08/05/402405.html)

[**深入浅出 Java Concurrency (1) : J.U.C的整体认识**](http://www.blogjava.net/xylz/archive/2010/06/30/324915.html)

去年年底有一个[Guice](http://www.blogjava.net/xylz/archive/2009/12/22/306955.html)的研究计划，可惜由于工作“繁忙”加上实际工作中没有用上导致“无疾而终”，最终只是完成了Guice的初步学习教程，深入的研究没有继续进行下去。

最近一直用的比较多的就是java.util.concurrent(J.U.C)，实际上这块一直也没有完全深入研究，这次准备花点时间研究下Java里面整个并发体系。初步的设想包括比较大的方面（包括硬件、软件、思想以及误区等等），因此可能会持续较长的时间。这块内容也是Java在多线程方面引以为豪的一部分，深入这一部分不仅对整个Java体系有更深的了解，也对工作、学习的态度有很多帮助。

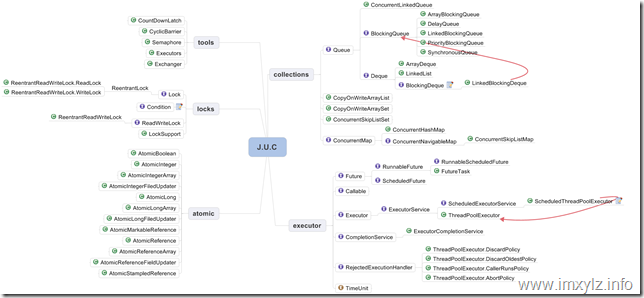
从深入浅出入手，大体内容包括以下几个方面：

（1）J.U.C的API：包括完整的类库结构和样例分析。

（2）J.U.C的硬件原理以及软件思想：这部分也将就自己个人对硬件与程序语言的一些认识与大家分享，主要以总结前人的经验和所谓的理论来进行一些描述。

（3）J.U.C的误区和常见陷阱：包括对J.U.C的一些设计思想和使用上的原则进行说明，同时对可能犯的错误进行一些总结。

下面的图是J.U.C完整的API。[完整的图片地址在这里](http://www.blogjava.net/images/blogjava_net/xylz/WindowsLiveWriter/JavaConcurrent_FB25/J.U.C_2.png)。

[](http://www.blogjava.net/images/blogjava_net/xylz/Windows-Live-Writer/-Java-Concurrency-1--J.U.C_9314/J.U.C_2.png)

完整的MindManger图形可以从下面的地址得到（注意，我是用MindManger 8写的，低版本没有测试是否能正常显示）。

[J.U.C完整的MindManger图形](http://www.blogjava.net/Files/xylz/J.U.C.zip)

在接下来的系列文章中，我们将根据这张API的图形进行完整的分析和研究。

[**深入浅出 Java Concurrency (2): 原子操作 part 1**](http://www.blogjava.net/xylz/archive/2010/07/01/324988.html)

从相对简单的Atomic入手（java.util.concurrent是基于Queue的并发包，而Queue，很多情况下使用到了Atomic操作，因此首先从这里开始）。很多情况下我们只是需要一个简单的、高效的、线程安全的递增递减方案。注意，这里有三个条件：简单，意味着程序员尽可能少的操作底层或者实现起来要比较容易；高效意味着耗用资源要少，程序处理速度要快；线程安全也非常重要，这个在多线程下能保证数据的正确性。这三个条件看起来比较简单，但是实现起来却难以令人满意。

通常情况下，在Java里面，++i或者--i不是线程安全的，这里面有三个独立的操作：获得变量当前值，为该值+1/-1，然后写回新的值（me:操作系统的指令操作）。在没有额外资源可以利用的情况下，只能使用加锁才能保证读-改-写这三个操作时“原子性”的。

Doug Lea在未将[backport-util-concurrent](http://backport-jsr166.sourceforge.net/)合并到[JSR 166](http://jcp.org/en/jsr/detail?id=166)里面来之前，是采用纯Java实现的，于是不可避免的采用了synchronized关键字。

public final synchronized void set(int newValue);

public final synchronized int getAndSet(int newValue);

public final synchronized int incrementAndGet();

同时在变量上使用了volatile （后面会具体来讲volatile到底是个什么东东）来保证get()的时候不用加锁。尽管synchronized的代价还是很高的，但是在没有JNI的手段下纯Java语言还是不能实现此操作的。

JSR 166提上日程后，backport-util-concurrent就合并到JDK 5.0里面了，在这里面重复使用了现代CPU的特性来降低锁的消耗。本章的最后小结中会谈到这些原理和特性。在此之前先看看API的使用。

一切从java.util.concurrent.atomic.AtomicInteger开始。

int addAndGet(int delta)  
          以原子方式将给定值与当前值相加。 实际上就是等于线程安全版本的i =i+delta操作。

boolean compareAndSet(int expect, int update)  
          如果当前值 == 预期值，则以原子方式将该值设置为给定的更新值。 如果成功就返回true，否则返回false，并且不修改原值。

int decrementAndGet()  
          以原子方式将当前值减 1。 相当于线程安全版本的--i操作。

int get()  
          获取当前值。

int getAndAdd(int delta)  
          以原子方式将给定值与当前值相加。 相当于线程安全版本的t=i;i+=delta;return t;操作。

int getAndDecrement()  
          以原子方式将当前值减 1。 相当于线程安全版本的i--操作。

int getAndIncrement()  
          以原子方式将当前值加 1。 相当于线程安全版本的i++操作。

int getAndSet(int newValue)  
          以原子方式设置为给定值，并返回旧值。 相当于线程安全版本的t=i;i=newValue;return t;操作。

int incrementAndGet()  
          以原子方式将当前值加 1。 相当于线程安全版本的++i操作。

void lazySet(int newValue)  
          最后设置为给定值。 延时设置变量值，这个等价于set()方法，但是由于字段是volatile类型的，因此次字段的修改会比普通字段（非volatile字段）有稍微的性能延时（尽管可以忽略），所以如果不是想立即读取设置的新值，允许在“后台”修改值，那么此方法就很有用。如果还是难以理解，这里就类似于启动一个后台线程执行修改新值的任务，原线程就不等待修改结果立即返回（这种解释其实是不正确的，但是可以这么理解）。

void set(int newValue)  
          设置为给定值。 直接修改原始值，也就是i=newValue操作。

boolean weakCompareAndSet(int expect, int update)  
          如果当前值 == 预期值，则以原子方式将该设置为给定的更新值。JSR规范中说：以原子方式读取和有条件地写入变量但*不* 创建任何 happen-before 排序，因此不提供与除 weakCompareAndSet 目标外任何变量以前或后续读取或写入操作有关的任何保证。大意就是说调用weakCompareAndSet时并不能保证不存在happen-before的发生（也就是可能存在指令重排序导致此操作失败）。但是从Java源码来看，其实此方法并没有实现JSR规范的要求，最后效果和compareAndSet是等效的，都调用了unsafe.compareAndSwapInt()完成操作。

下面的代码是一个测试样例，为了省事就写在一个方法里面来了。

http://www.blogjava.net/Images/OutliningIndicators/None.gifpackage xylz.study.concurrency.atomic;  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gifimport java.util.concurrent.atomic.AtomicInteger;  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gifimport org.junit.Test;  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gifimport static org.junit.Assert.\*;  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockStart.gifpublic class AtomicIntegerTest {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif    @Test  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif    public void testAll() throws InterruptedException{  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        final AtomicInteger value = new AtomicInteger(10);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.compareAndSet(1, 2), false);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.get(), 10);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertTrue(value.compareAndSet(10, 3));  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.get(), 3);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        value.set(0);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        //  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.incrementAndGet(), 1);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.getAndAdd(2),1);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.getAndSet(5),3);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.get(),5);  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        //  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        final int threadSize = 10;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        Thread[] ts = new Thread[threadSize];  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif        for (int i = 0; i < threadSize; i++) {  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif            ts[i] = new Thread() {  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif                public void run() {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif                    value.incrementAndGet();  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif                }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            };  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        //  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif        for(Thread t:ts) {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            t.start();  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif        for(Thread t:ts) {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            t.join();  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        //  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        assertEquals(value.get(), 5+threadSize);  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
http://www.blogjava.net/Images/OutliningIndicators/None.gif

 由于这里例子比较简单，这里就不做过多介绍了。  
AtomicInteger和AtomicLong、AtomicBoolean、AtomicReference差不多，这里就不介绍了。在下一篇中就介绍下数组、字段等其他方面的原子操作。  
 参考资料：  
(1)<http://stackoverflow.com/questions/2443239/java-atomicinteger-what-are-the-differences-between-compareandset-and-weakcompar>  
(2)<http://stackoverflow.com/questions/1468007/atomicinteger-lazyset-and-set>

[**深入浅出 Java Concurrency (3): 原子操作 part 2**](http://www.blogjava.net/xylz/archive/2010/07/02/325079.html)

在这一部分开始讨论数组原子操作和一些其他的原子操作。

**AtomicIntegerArray/AtomicLongArray/AtomicReferenceArray**的API类似，选择有代表性的AtomicIntegerArray来描述这些问题。

**int get(int i)**

获取位置 i 的当前值。很显然，由于这个是数组操作，就有索引越界的问题（IndexOutOfBoundsException异常）。

对于下面的API起始和AtomicInteger是类似的，这种通过方法、参数的名称就能够得到函数意义的写法是非常值得称赞的。在[《重构：改善既有代码的设计》](http://www.china-pub.com/12901)和[《代码整洁之道》](http://www.china-pub.com/196266)中都非常推崇这种做法。

**void set(int i, int newValue)**  
**void lazySet(int i, int newValue)  
int getAndSet(int i, int newValue)  
boolean compareAndSet(int i, int expect, int update)  
boolean weakCompareAndSet(int i, int expect, int update)  
int getAndIncrement(int i)  
int getAndDecrement(int i)  
int getAndAdd(int i, int delta)  
int incrementAndGet(int i)  
int decrementAndGet(int i)  
int addAndGet(int i, int delta)**

整体来说，数组的原子操作在理解上还是相对比较容易的，这些API就是有多使用才能体会到它们的好处，而不仅仅是停留在理论阶段。

现在关注字段的原子更新。

**AtomicIntegerFieldUpdater<T>/AtomicLongFieldUpdater<T>/AtomicReferenceFieldUpdater<T,V>**是基于反射的原子更新字段的值。

相应的API也是非常简单的，但是也是有一些约束的。

（1）字段必须是volatile类型的！在后面的章节中会详细说明为什么必须是volatile，volatile到底是个什么东西。

（2）字段的描述类型（修饰符public/protected/default/private）是与调用者与操作对象字段的关系一致。也就是说调用者能够直接操作对象字段，那么就可以反射进行原子操作。但是对于父类的字段，子类是不能直接操作的，尽管子类可以访问父类的字段。

（3）只能是实例变量，不能是类变量，也就是说不能加static关键字。

（4）只能是可修改变量，不能是final变量，因为final的语义就是不可修改。实际上final的语义和volatile是有冲突的，这两个关键字不能同时存在。

（5）对于**AtomicIntegerFieldUpdater**和**AtomicLongFieldUpdater**只能修改int/long类型的字段，不能修改其包装类型（Integer/Long）。如果要修改包装类型就需要使用**AtomicReferenceFieldUpdater**。

在下面的例子中描述了操作的方法。

http://www.blogjava.net/Images/OutliningIndicators/None.gifpackage xylz.study.concurrency.atomic;   
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gifimport java.util.concurrent.atomic.AtomicIntegerFieldUpdater;   
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockStart.gifpublic class AtomicIntegerFieldUpdaterDemo {   
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif   class DemoData{  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif       public volatile int value1 = 1;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif       volatile int value2 = 2;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif       protected volatile int value3 = 3;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif       private volatile int value4 = 4;  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif   }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif    AtomicIntegerFieldUpdater<DemoData> getUpdater(String fieldName) {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        return AtomicIntegerFieldUpdater.newUpdater(DemoData.class, fieldName);  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif    void doit() {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        DemoData data = new DemoData();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        System.out.println("1 ==> "+getUpdater("value1").getAndSet(data, 10));  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        System.out.println("3 ==> "+getUpdater("value2").incrementAndGet(data));  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        System.out.println("2 ==> "+getUpdater("value3").decrementAndGet(data));  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        System.out.println("true ==> "+getUpdater("value4").compareAndSet(data, 4, 5));  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif    public static void main(String[] args) {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        AtomicIntegerFieldUpdaterDemo demo = new AtomicIntegerFieldUpdaterDemo();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        demo.doit();  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockEnd.gif}   
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gif

在上面的例子中DemoData的字段value3/value4对于AtomicIntegerFieldUpdaterDemo类是不可见的，因此通过反射是不能直接修改其值的。

**AtomicMarkableReference**类描述的一个<Object,Boolean>的对，可以原子的修改Object或者Boolean的值，这种数据结构在一些缓存或者状态描述中比较有用。这种结构在单个或者同时修改Object/Boolean的时候能够有效的提高吞吐量 。

**AtomicStampedReference**类维护带有整数“标志”的对象引用，可以用原子方式对其进行更新。对比**AtomicMarkableReference**类的<Object,Boolean>，**AtomicStampedReference**维护的是一种类似<Object,int>的数据结构，其实就是对对象（引用）的一个并发计数。但是与**AtomicInteger**不同的是，此数据结构可以携带一个对象引用（Object），并且能够对此对象和计数同时进行原子操作。

在后面的章节中会提到“ABA问题”，而**AtomicMarkableReference/AtomicStampedReference**在解决“ABA问题”上很有用**。**

原子操作的使用大概就是这么多，大体来说还算是比较清晰的，在下一个章节中，将对原子操作进行总结，重点介绍下原子操作的原理和设计思想。

[**深入浅出 Java Concurrency (4): 原子操作 part 3 指令重排序与happens-before法则**](http://www.blogjava.net/xylz/archive/2010/07/03/325168.html)

在这个小结里面重点讨论原子操作的原理和设计思想。

由于在下一个章节中会谈到锁机制，因此此小节中会适当引入锁的概念。

在[Java Concurrency in Practice](http://www.amazon.com/exec/obidos/ASIN/0321349601/ref=nosim/none0b69)中是这样定义线程安全的：

***当多个线程访问一个类时，如果不用考虑这些线程在运行时环境下的调度和交替运行，并且不需要额外的同步及在调用方代码不必做其他的协调，这个类的行为仍然是正确的，那么这个类就是线程安全的。***

显然只有资源竞争时才会导致线程不安全，因此***无状态对象永远是线程安全的***。

原子操作的描述是： 多个线程执行一个操作时，其中***任何一个线程要么完全执行完此操作，要么没有执行此操作的任何步骤***，那么这个操作就是原子的。

枯燥的定义介绍完了，下面说更枯燥的理论知识。

**指令重排序**

Java语言规范规定了JVM线程内部维持顺序化语义，也就是说只要程序的最终结果等同于它在严格的顺序化环境下的结果，那么指令的执行顺序就可能与代码的顺序不一致。这个过程通过叫做指令的重排序。指令重排序存在的意义在于：JVM能够根据处理器的特性（CPU的多级缓存系统、多核处理器等）适当的重新排序机器指令，使机器指令更符合CPU的执行特点，最大限度的发挥机器的性能。

程序执行最简单的模型是按照指令出现的顺序执行，这样就与执行指令的CPU无关，最大限度的保证了指令的可移植性。这个模型的专业术语叫做顺序化一致性模型。但是现代计算机体系和处理器架构都不保证这一点（因为人为的指定并不能总是保证符合CPU处理的特性）。

我们来看最经典的一个案例。

http://www.blogjava.net/Images/OutliningIndicators/None.gifpackage xylz.study.concurrency.atomic;   
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockStart.gifpublic class ReorderingDemo {   
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif    static int x = 0, y = 0, a = 0, b = 0;   
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif    public static void main(String[] args) throws Exception {   
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif        for (int i = 0; i < 100; i++) {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            x=y=a=b=0;  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif            Thread one = new Thread() {  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif                public void run() {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif                    a = 1;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif                    x = b;  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif                }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            };  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif            Thread two = new Thread() {  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockStart.gif                public void run() {  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif                    b = 1;  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif                    y = a;  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif                }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif            };  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            one.start();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            two.start();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            one.join();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            two.join();  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif            System.out.println(x + " " + y);  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif        }  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedSubBlockEnd.gif    }   
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockEnd.gif}  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gif

在这个例子中one/two两个线程修改区x,y,a,b四个变量，在执行100次的情况下，可能得到(0 1)或者（1 0）或者（1 1）。事实上按照JVM的规范以及CPU的特性有很可能得到（0 0）。当然上面的代码大家不一定能得到（0 0），因为run()里面的操作过于简单，可能比启动一个线程花费的时间还少，因此上面的例子难以出现（0,0）。但是在现代CPU和JVM上确实是存在的。由于run()里面的动作对于结果是无关的，因此里面的指令可能发生指令重排序，即使是按照程序的顺序执行，数据变化刷新到主存也是需要时间的。假定是按照a=1;x=b;b=1;y=a;执行的，x=0是比较正常的，虽然a=1在y=a之前执行的，但是由于线程one执行a=1完成后还没有来得及将数据1写回主存（这时候数据是在线程one的堆栈里面的），线程two从主存中拿到的数据a可能仍然是0（显然是一个过期数据，但是是有可能的），这样就发生了数据错误。

在两个线程交替执行的情况下数据的结果就不确定了，在机器压力大，多核CPU并发执行的情况下，数据的结果就更加不确定了。

**Happens-before法则**

Java存储模型有一个happens-before原则，就是如果动作B要看到动作A的执行结果（无论A/B是否在同一个线程里面执行），那么A/B就需要满足happens-before关系。

在介绍happens-before法则之前介绍一个概念：JMM动作（Java Memeory Model Action），Java存储模型动作。一个动作（Action）包括：变量的读写、监视器加锁和释放锁、线程的start()和join()。后面还会提到锁的。

happens-before完整规则：

*（1）同一个线程中的每个Action都happens-before于出现在其后的任何一个Action(me：即线程内部是按照顺序执行的)。*

*（2）对一个监视器的解锁happens-before于每一个后续对同一个监视器的加锁。*

*（3）对volatile字段的写入操作happens-before于每一个后续的同一个字段的读操作(me:先写后读)。*

*（4）Thread.start()的调用会happens-before于启动线程里面的动作。*

*（5）Thread中的所有动作都happens-before于其他线程检查到此线程结束或者Thread.join（）中返回或者Thread.isAlive()==false。*

*（6）一个线程A调用另一个另一个线程B的interrupt（）都happens-before于线程A发现B被A中断（B抛出异常或者A检测到B的isInterrupted（）或者interrupted()）。*

*（7）一个对象构造函数的结束happens-before与该对象的finalizer的开始*

*（8）如果A动作happens-before于B动作，而B动作happens-before与C动作，那么A动作happens-before于C动作。*

**volatile语义**

到目前为止，我们多次提到volatile，但是却仍然没有理解volatile的语义。

volatile相当于synchronized的弱实现，也就是说volatile实现了类似synchronized的语义，却又没有锁机制。它确保对volatile字段的更新以可预见的方式告知其他的线程。

volatile包含以下语义：

（1）Java 存储模型不会对valatile指令的操作进行重排序：这个保证对volatile变量的操作时按照指令的出现顺序执行的。

（2）volatile变量不会被缓存在寄存器中（只有拥有线程可见）或者其他对CPU不可见的地方，每次总是从主存中读取volatile变量的结果。也就是说对于volatile变量的修改，其它线程总是可见的，并且不是使用自己线程栈内部的变量。也就是在happens-before法则中，对一个valatile变量的写操作后，其后的任何读操作理解可见此写操作的结果。

尽管volatile变量的特性不错，但是volatile并不能保证线程安全的，也就是说volatile字段的操作不是原子性的，volatile变量只能保证可见性（一个线程修改后其它线程能够理解看到此变化后的结果），要想保证原子性，目前为止只能加锁！

volatile通常在下面的场景：

http://www.blogjava.net/Images/OutliningIndicators/None.gifvolatile boolean done = false;  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/None.gif…  
http://www.blogjava.net/Images/OutliningIndicators/None.gif  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockStart.gif    while( ! done ){  
http://www.blogjava.net/Images/OutliningIndicators/InBlock.gif        dosomething();  
http://www.blogjava.net/Images/OutliningIndicators/ExpandedBlockEnd.gif    }

应用volatile变量的三个原则：

*（1）写入变量不依赖此变量的值，或者只有一个线程修改此变量*

*（2）变量的状态不需要与其它变量共同参与不变约束*

*（3）访问变量不需要加锁*

**这一节理论知识比较多，但是这是很多章节的基础，在后面的章节中会多次提到这些特性。**

本小节中还是没有谈到原子操作的原理和思想，在下一节中将根据上面的一些知识来介绍原子操作。

参考资料：

（1）[Java Concurrency in Practice](http://www.amazon.com/exec/obidos/ASIN/0321349601/ref=nosim/none0b69)

（2）[正确使用 Volatile 变量](http://www.ibm.com/developerworks/cn/java/j-jtp06197.html)

[**深入浅出 Java Concurrency (5): 原子操作 part 4**](http://www.blogjava.net/xylz/archive/2010/07/04/325206.html)

在JDK 5之前Java语言是靠synchronized关键字保证同步的，这会导致有锁（后面的章节还会谈到锁）。

锁机制存在以下问题：

（1）在多线程竞争下，加锁、释放锁会导致比较多的上下文切换和调度延时，引起性能问题。

（2）一个线程持有锁会导致其它所有需要此锁的线程挂起。

（3）如果一个优先级高的线程等待一个优先级低的线程释放锁会导致优先级倒置，引起性能风险。

volatile是不错的机制，但是volatile不能保证原子性。因此对于同步最终还是要回到锁机制上来。

独占锁是一种悲观锁，synchronized就是一种独占锁，会导致其它所有需要锁的线程挂起，等待持有锁的线程释放锁。而另一个更加有效的锁就是乐观锁。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。

**CAS 操作**

上面的乐观锁用到的机制就是CAS，Compare and Swap。

CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。

**非阻塞算法 （nonblocking algorithms）**

*一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。*

现代的CPU提供了特殊的指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 compareAndSet() 就用这些代替了锁定。

拿出AtomicInteger来研究在没有锁的情况下是如何做到数据正确性的。

*private volatile int value;*

首先毫无以为，在没有锁的机制下可能需要借助volatile原语，保证线程间的数据是可见的（共享的）。

这样才获取变量的值的时候才能直接读取。

*public final int get() {  
        return value;  
    }*

然后来看看++i是怎么做到的。

*public final int incrementAndGet() {  
    for (;;) {  
        int current = get();  
        int next = current + 1;  
        if (compareAndSet(current, next))  
            return next;  
    }  
}*

在这里采用了CAS操作，每次从内存中读取数据然后将此数据和+1后的结果进行CAS操作，如果成功就返回结果，否则重试直到成功为止。

而compareAndSet利用JNI来完成CPU指令的操作。

*public final boolean compareAndSet(int expect, int update) {     
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);  
    }*

整体的过程就是这样子的，利用CPU的CAS指令，同时借助JNI来完成Java的非阻塞算法。其它原子操作都是利用类似的特性完成的。

而整个J.U.C都是建立在CAS之上的，因此对于synchronized阻塞算法，J.U.C在性能上有了很大的提升。参考资料的文章中介绍了如果利用CAS构建非阻塞计数器、队列等数据结构。

CAS看起来很爽，但是会导致“ABA问题”。

CAS算法实现一个重要前提需要取出内存中某时刻的数据，而在下时刻比较并替换，那么在这个时间差类会导致数据的变化。

比如说一个线程one从内存位置V中取出A，这时候另一个线程two也从内存中取出A，并且two进行了一些操作变成了B，然后two又将V位置的数据变成A，这时候线程one进行CAS操作发现内存中仍然是A，然后one操作成功。尽管线程one的CAS操作成功，但是不代表这个过程就是没有问题的。如果链表的头在变化了两次后恢复了原值，但是不代表链表就没有变化。因此前面提到的原子操作AtomicStampedReference/AtomicMarkableReference就很有用了。这允许一对变化的元素进行原子操作。

参考资料：

（1）[非阻塞算法简介](http://www.ibm.com/developerworks/cn/java/j-jtp04186/)

（2）[流行的原子](https://www.ibm.com/developerworks/cn/java/j-jtp11234/)