**Atomic包**

# 什么是原子类？有什么作用？

## 不可分割

我们来说在化学反应中一般来讲，原子是不可分割的，我们分子啊，比如说水，经过电解呢，会生成氧气和氢气，说明分子是比较容易分解的，这就会牵扯到化学方程式，但是我们的化学方程式一般都是牵扯到分子级别的，所以原子的一个特点就是不太容易被分割。

在编程领域的原子操作它是不可中断的，即便是在多线程的情况下，也可以保证。

## 一个操作是不可中断的，即便是多线程的情况下也可以保证。

那么我们说你原子下面好像不是不可中断的，好像也可以分割嘛。对不对，由中子，质子，下面还有夸克，还有量子，到底什么是不可分割的？我这边可以确定的就是我和我的做一刻钟也不能分割。

## 3、java.util.concurrent.atomic

而在我们的java中啊，和原子相关的都放在了我们的java.util.concurrent.atomic这个包里，这个包里面都是具有原子特性的类。

## 原子类的作用和锁类似，是为了保证在并发情况下线程安全，不过原子类相比于锁，有一定的优势：

### 粒度更细：原子变量可以把竞争范围缩小到变量级别，这是我们可以获得的最细粒度的情况了，通常锁的粒度都要大于原子变量的粒度。

因为我们的原子变量可以把竞争范围缩小到变量级别，比如说我们使用一个AtomicInteger，那么它本身是一个变量，合适我们可以获取到最细的粒度了，而我们使用锁通常会锁住好几行代码，甚至锁住某个方法，那这种情况下他的粒度都要大于我们原子变量的粒度，那么对锁而言，其实对于粒度呢，一般是越小越好，这样可以提高我们的并发效率，避免锁住无用代码，

### 效率更高：通常，使用原子类的效率会比使用锁的效率更高，除了高度竞争的情况。

通常情况下原子类的效率是比锁高得，但是呢，他也有例外，原子类在高度竞争的情况下，那他的效率是不如我们的锁的，所以呢，要根据具体的场景去进行选择，到后面我们会讲原子类在高度竞争的情况下他的效率会降低。在这里我们主要需要知道的就是原子类就是有这两大特点。

# 6类原子类纵览

## Atomic\*基本类型原子类

1. AtomicInteger
2. AtomicLong
3. AtomicBool

## Atomic\*Array数组类型原子类

1. AtomicIntegerArray
2. AtomicLongArray
3. AtomicReferenceArray

## 3、Atomic\*Reference引用类型原子类

1、AtomicReference

2、AtomicStampedReference

3、AtomicMarkableReference

## 4、Atomic\*FieldUpdater升级类型原子类

1、AtomicIntegerfiledupdater

2、AtomicLongFiledUpdater

3、AtomicReferenceFiledUpdater

## 5、Adder累加器

1、LongAdder

2、DoubleAdder

## 6、Accemulator累加器

1、LongAccumulator

2、DoubleAccumulator

# Atomic\*基本类型原子类，以AtomicInteger为例

## AtomicInteger：整形原子类

### 1、概念

它是对int或者是说对Integer的一种封装，封装好了以后就提供了原子的更新和访问操作，他背后呢是基于CAS技术。在这里我们可以把它理解为有一系列指令，这个指令可以在不被打断的情况下，去更新某个值，并且在更新的时候呢会去检查，和我预期的是否一致。

### 常用方法。

1. public final int get() //获取当前的值
2. public final int getAndSet(int newValue) //获取当前的值，并设置新的值
3. public final int getAndIncrement() //获取当前的值，并自增

increment的意思是增加，他这里就体现了它原子性的特点，这个increment我们如果不用原子类来写的话，比如说我们写成a++的形式，那这a++本身不是一个原子操作，它内部会分为三个步骤：先读取，然后+，最后呢再写回去，那这种情况下是可能被打断的。等会我们会去演示，但是我们在使用原子类的getAndIncrement()之后，他就会把整个操作变成原子的，在此期间谁都打断不了，就可以保证线程安全，

1. public final int getAndDecrement() //获取当前值，并自减。
2. Public final int getAndAdd(int delta) //获取当前的值，并加上预期的值。
3. Boolean compareAndSet(int expect,int update) //如果输入的数值等于预期值，则以原子方式将该值设置为输入值(update)

## AtomicLong：长整型原子类

## AtomicBoolean：布尔型原子类

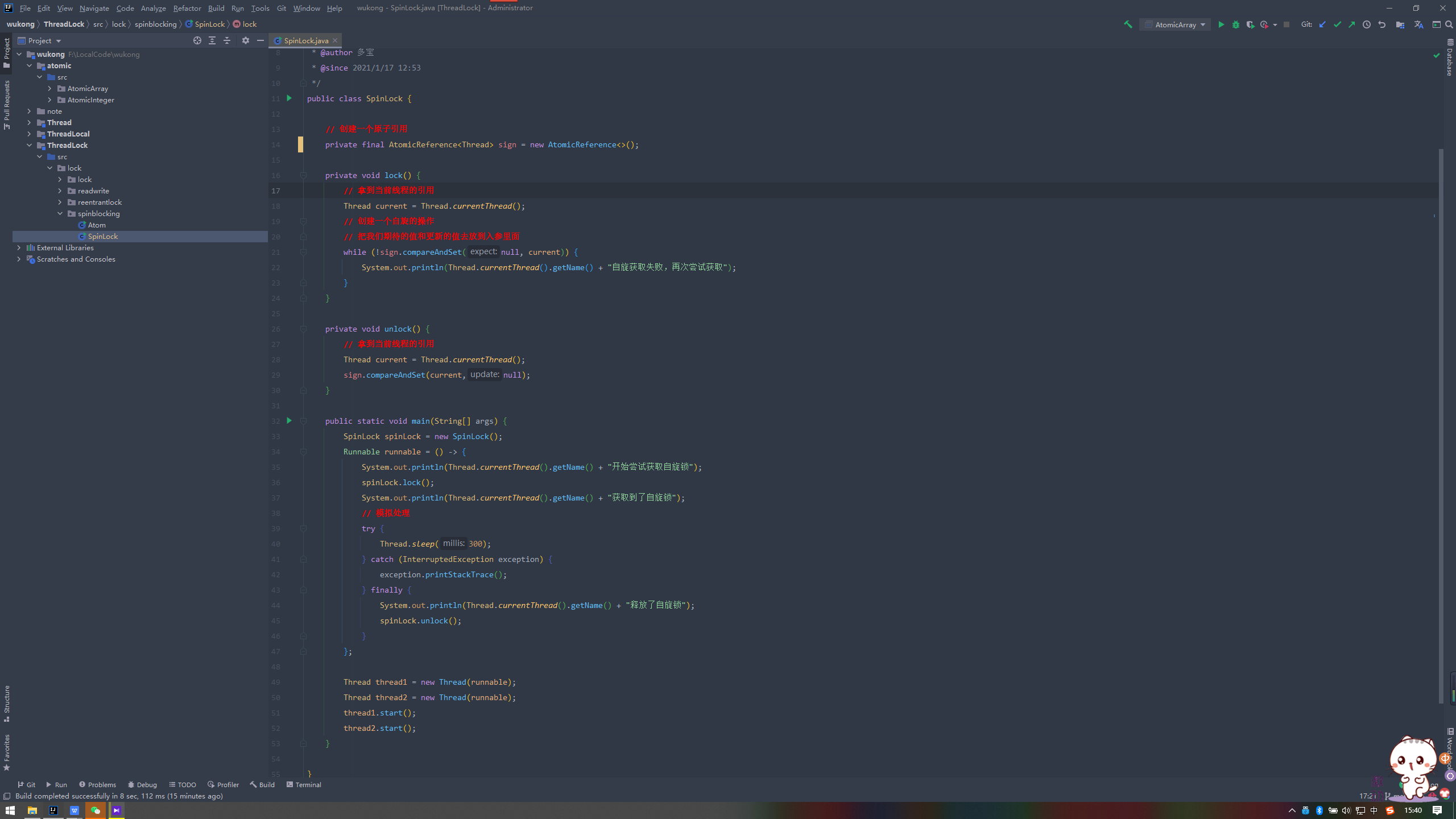
# Atomic\*Array数组类型原子类

# Atomic\*Reference引用类型原子类

## 1、作用

AtomicReference:AtomicReference类的作：，和AtomicInteger并没有本质区别，AtomicInteger可以让一个整数保证了原子性，而AtomicReference可以让一个对象保证原子性，当然，AtomicReference的功能明显比AtomicInteger强，因为一个对象里可以包含很多属性，用法和AtomincInteger类似。

## 2、使用(以自旋锁为例)



在自旋锁中，加锁的时候，我们将线程作为原子引用类型的泛型，我们在使用原子引用类型的compareAndSet()的时候，我们要判断当前原子引用的线程是不是空，如果是空我们将线程设置为当前线程。

其实原子引用类型就是针对对象做原子操作，而并非是单独的针对基本类型的封装。

# 把普通变量升级为原子类：用AtomicIntegerFieldUpdater升级原有变量。

## AtomicIntegerFieldUpdater对普通变量进行升级

### 思想

AtomicIntegerFieldUpdater还是比较厉害的，我们用了它之后呢，可以对普通的变量进行升级。比如说我们之前已经是生命过一个对象，这个对象里面啊有一个int这个类型的一个数值，那这个int明显他是基本类型，所以它是不具备原子性的，那么在这种情况下，木已成舟，因为类已经被定义好了，那么我们也不方便去改变这个类，此时又想对他这里面的一个变量进行原子操作，怎么办呢？在这种场景下啊，我们就可以利用我们的AtomicIntegerFieldUpdater，相当于使用它来辅助我们自己的一个普通的对象，让它拥有一个原子的能力。

那你会说啊，哎，你这样是不是多此一举？你既然知道这里面有一个变量会被多线程访问，而且呢，还有可能是并发+的，那你一开始就声明成原子的不好吗？你一开始就把他生命成一个原子的AtomicInteger，那这样一来，你就不需要费时费力去对它进行升级了，这种情况呢也是对的。如果我们在某一个类中某一个变量经常被并发的读写，而我们希望他是个原子的话，可以在最开始就声明为是一个原子类型。

但是我们的AtomicIntegerFieldUpdater他有自己的使用场景。

第一个适用场景就是说：我们如果一个类不是我们自己编写，我们没有权利修改它，那么我们可以去适用AtomicIntegerFieldUpdater。

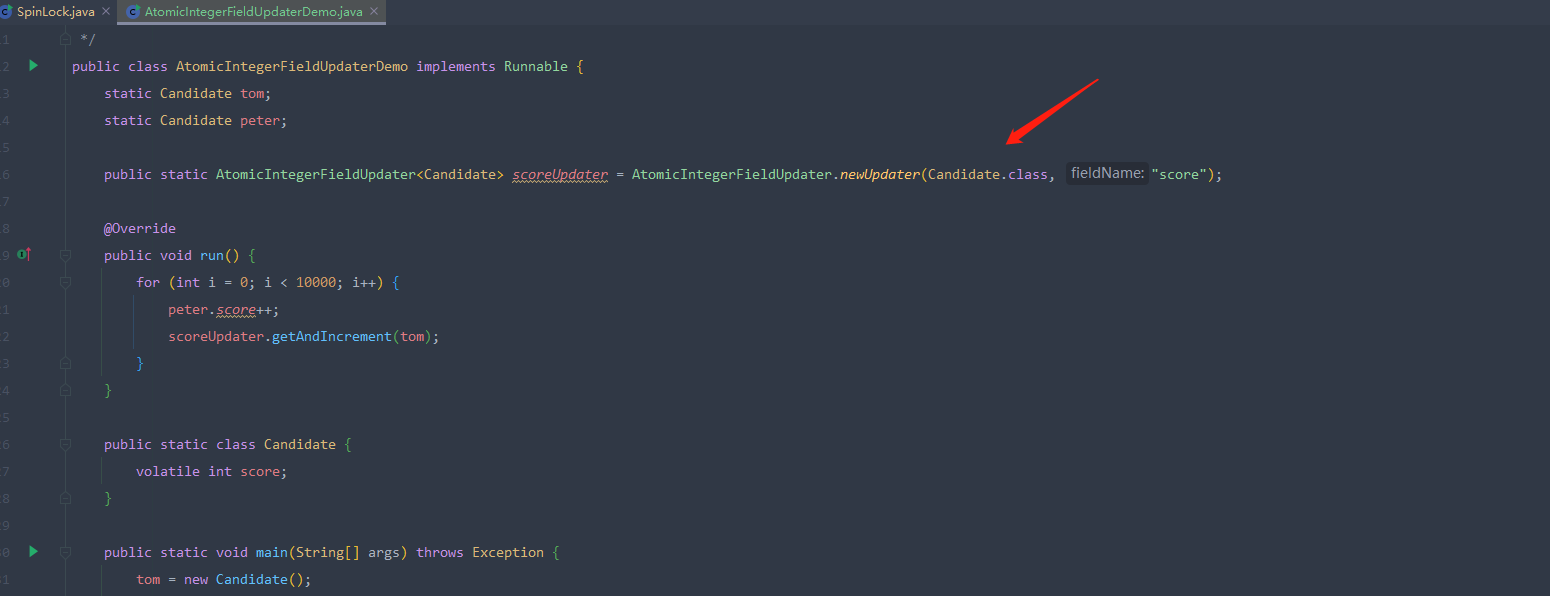
第二个适用场景就是说可能我们偶尔需要原子的get-set操作。这种情况不是特别多，那么如果我们的业务场景是，比如说每天只有是在0点的时候，才会对它进行一个原子的更新。并且这个时候还存在并发问题。我们必须要保持原子性的话，我们就必须对它进行升级。但是除了晚上0点的那一个时刻之外，其他时刻并不存在线程并发访问的一个情况，所以在其他时刻我们只需要把它当做是一个普通变量就可以了。

那么如果是说在这种情况下，也就是说我们大部分情况下希望它是一个普通的类型，只有很少数的情况下才需要这种普通类型具备原子性，那么我们就没有必要在这个类中去嵌入一个额外的原子对象，因为实际上我们生成原子对象，因为我们实际上生成原子对象并且对它进行操作的话，也是有资源消耗的，一个AtomicInteger，它所带来的消耗远比我们的基本类型要大，包括内存、包括操作时候的消耗，所以如果我们这样的类呢有很多个，比如说成千上万个，那么这个时候内存的占用就是一个值得考虑的问题，所以在这种情况下我们就可以把我们类中的数值，声明为一个普通类型，比如说int，那么这样的资源占用是最小的，那么我们需要原子操作的时候呢，我们再去利用AtomicIntegerFieldUpdater，去对它进行升级，尤其是当数量庞大之后，这个节约的资源就是可观的。这样一来就优化了性能。

### 用法，代码演示

### AtomicIntegerFieldUpdater注意点

1. 可见范围



# Adder累加器

# Accumulator累加器