[https://mp.weixin.qq.com/s?\_\_biz=MzA5MTkxMDQ4MQ==&mid=2648933181&idx=1&sn=a1e254365d405cdc2e3b8372ecda65ee&chksm=88621b03bf159215ca696c9f81e228d0544a7598b03fe30436babc95c6a95e848161f61b868c&token=743622661&lang=zh\_CN&scene=21#wechat\_redirect](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933181&idx=1&sn=a1e254365d405cdc2e3b8372ecda65ee&chksm=88621b03bf159215ca696c9f81e228d0544a7598b03fe30436babc95c6a95e848161f61b868c&token=743622661&lang=zh_CN&scene=21" \l "wechat_redirect)

# JUC中的原子类

## 本文主要内容

1. JUC中的原子类介绍
2. 介绍基本类型原子类
3. 介绍数组类型原子类
4. 介绍引用类型原子类
5. 介绍对象属性修改相关原子类

## **预备知识**

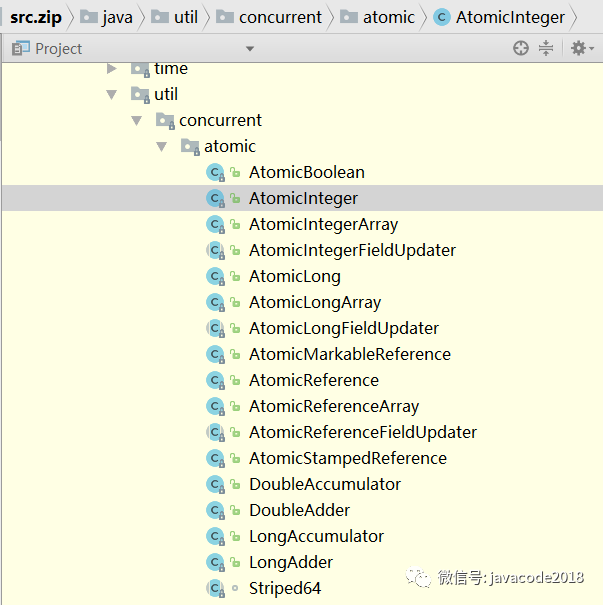
JUC中的原子类都是都是依靠**volatile**、**CAS**、**Unsafe**类配合来实现的，需要了解的请移步：  
[volatile与Java内存模型](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933088&idx=1&sn=f1d666dd799664b1989c77441b9d12c5&chksm=88621adebf1593c83501ac33d6a0e0de075f2b2e30caf986cf276cbb1c8dff0eac2a0a648b1d&token=2041017112&lang=zh_CN&scene=21" \l "wechat_redirect)  
[java中的CAS](http://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933166&idx=1&sn=15e614500676170b76a329efd3255c12&chksm=88621b10bf1592064befc5c9f0d78c56cda25c6d003e1711b85e5bfeb56c9fd30d892178db87&scene=21" \l "wechat_redirect)  
[JUC底层工具类Unsafe](https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933173&idx=1&sn=80eb550294677b0042fc030f90cce109&chksm=88621b0bbf15921d2274a7bf6afde912fec02a4c3ade9cfb50d03cdce73e07e33d08d35a3b27&token=1771071046&lang=zh_CN&scene=21" \l "wechat_redirect)

## **JUC中原子类介绍**

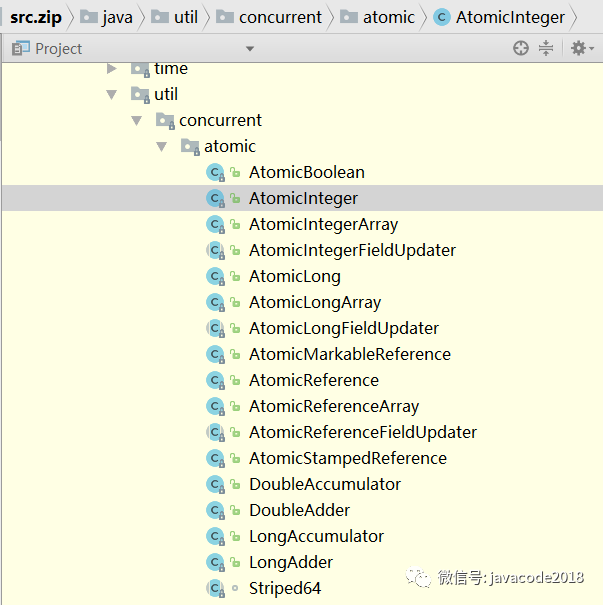
**什么是原子操作？**

**atomic** 翻译成中文是原子的意思。在化学上，我们知道原子是构成一般物质的最小单位，在化学反应中是不可分割的。**在我们这里 atomic 是指一个操作是不可中断的。即使是在多个线程一起执行的时候，一个操作一旦开始，就不会被其他线程干扰，所以，所谓原子类说简单点就是具有原子操作特征的类，原子操作类提供了一些修改数据的方法，这些方法都是原子操作的，在多线程情况下可以确保被修改数据的正确性**。

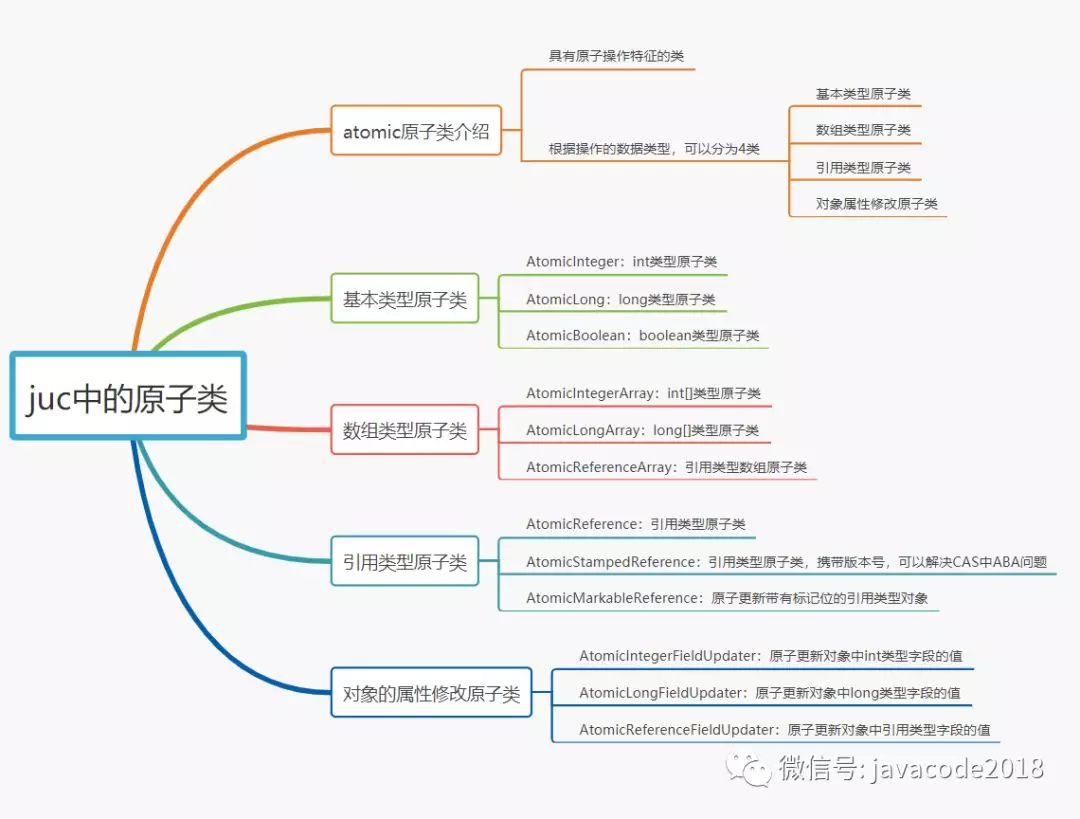
JUC中对原子操作提供了强大的支持，这些类位于java.util.concurrent.atomic包中，如下图：



JUC中对原子操作提供了强大的支持，这些类位于java.util.concurrent.atomic包中，如下图：



## **JUC中原子类思维导图**



## 基本类型原子类

使用原子的方式更新基本类型

* AtomicInteger：int类型原子类
* AtomicLong：long类型原子类
* AtomicBoolean ：boolean类型原子类

上面三个类提供的方法几乎相同，这里以 AtomicInteger 为例子来介绍。

**AtomicInteger 类常用方法**

**public final int get()** //获取当前的值

**public final int getAndSet(int newValue)**//获取当前的值，并设置新的值

**public final int getAndIncrement()**//获取当前的值，并自增

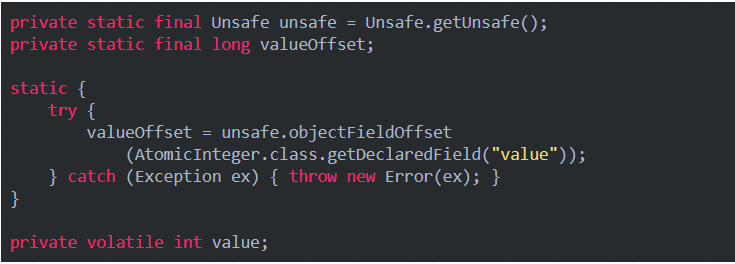
**public final int getAndDecrement()** //获取当前的值，并自减

**public final int getAndAdd(int delta)** //获取当前的值，并加上预期的值

**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）

**public final void lazySet(int newValue)**//最终设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

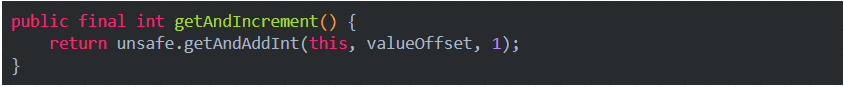
**部分源码**



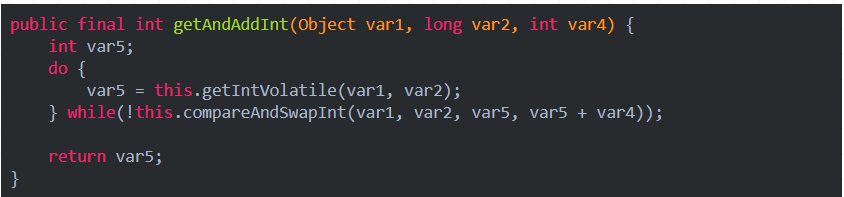
2个关键字段说明：

* value：使用volatile修饰，可以确保value在多线程中的可见性。
* valueOffset：value属性在AtomicInteger中的偏移量，通过这个偏移量可以快速定位到value字段，这个是实现AtomicInteger的关键。

**getAndIncrement源码：**



内部调用的是Unsafe类中的getAndAddInt方法，我们看一下getAndAddInt源码：



说明：

this.getIntVolatile：可以确保从主内存中获取变量最新的值。

compareAndSwapInt：CAS操作，CAS的原理是拿期望的值和原本的值作比较，如果相同则更新成新的值，可以确保在多线程情况下只有一个线程会操作成功，不成功的返回false。

上面有个do-while循环，compareAndSwapInt返回false之后，会再次从主内存中获取变量的值，继续做CAS操作，直到成功为止。

getAndAddInt操作相当于线程安全的count++操作，如同：

synchronize(lock){

count++;

}

count++操作实际上是被拆分为3步骤执行：

1. 获取count的值，记做A：A=count
2. 将A的值+1，得到B：B = A+1
3. 让B赋值给count：count = B

多线程情况下会出现线程安全的问题，导致数据不准确。

synchronize的方式会导致占时无法获取锁的线程处于阻塞状态，性能比较低。CAS的性能比synchronize要快很多。

**示例**

使用AtomicInteger实现网站访问量计数器功能，模拟100人同时访问网站，每个人访问10次，代码如下：



输出：



通过输出中可以看出incrementAndGet在多线程情况下能确保数据的正确性。

## **数组类型原子类介绍**

使用原子的方式更新数组里的某个元素，可以确保修改数组中数据的线程安全性。

AtomicIntegerArray：整形数组原子操作类

AtomicLongArray：长整形数组原子操作类

AtomicReferenceArray ：引用类型数组原子操作类

上面三个类提供的方法几乎相同，所以我们这里以 AtomicIntegerArray 为例子来介绍。

**AtomicIntegerArray 类常用方法**

**public final int get(int i)** //获取 index=i 位置元素的值

**public final int getAndSet(int i, int newValue)**//返回 index=i 位置的当前的值，并将其设置为新值：newValue

**public final int getAndIncrement(int i)**//获取 index=i 位置元素的值，并让该位置的元素自增

**public final int getAndDecrement(int i)** //获取 index=i 位置元素的值，并让该位置的元素自减

**public final int getAndAdd(int delta)** //获取 index=i 位置元素的值，并加上预期的值

**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 index=i 位置的元素值设置为输入值（update）

**public final void lazySet(int i, int newValue)**//最终 将index=i 位置的元素设置为newValue,使用 lazySet 设置之后可能导致其他线程在之后的一小段时间内还是可以读到旧的值。

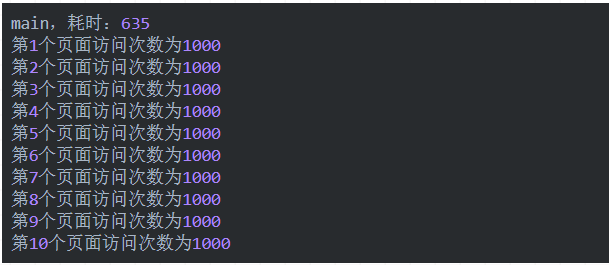
**示例**

统计网站页面访问量，假设网站有10个页面，现在模拟100个人并行访问每个页面10次，然后将每个页面访问量输出，应该每个页面都是1000次，代码如下：





输出：



说明：

代码中将10个页面的访问量放在了一个int类型的数组中，数组大小为10，然后通过AtomicIntegerArray来操作数组中的每个元素，可以确保操作数据的原子性，每次访问会调用incrementAndGet，此方法需要传入数组的下标，然后对指定的元素做原子+1操作。输出结果都是1000，可以看出对于数组中元素的并发修改是线程安全的。如果线程不安全，则部分数据可能会小于1000。

其他的一些方法可以自行操作一下，都非常简单。

## **引用类型原子类介绍**

基本类型原子类只能更新一个变量，如果需要原子更新多个变量，需要使用 引用类型原子类。

* **AtomicReference**：引用类型原子类
* **AtomicStampedRerence**：原子更新引用类型里的字段原子类
* **AtomicMarkableReference** ：原子更新带有标记位的引用类型

**AtomicReference** 和 **AtomicInteger** 非常类似，不同之处在于 **AtomicInteger**是对整数的封装，而**AtomicReference**则是对应普通的对象引用，它可以确保你在修改对象引用时的线程安全性。在介绍**AtomicReference**的同时，我们先来了解一个有关原子操作逻辑上的不足。

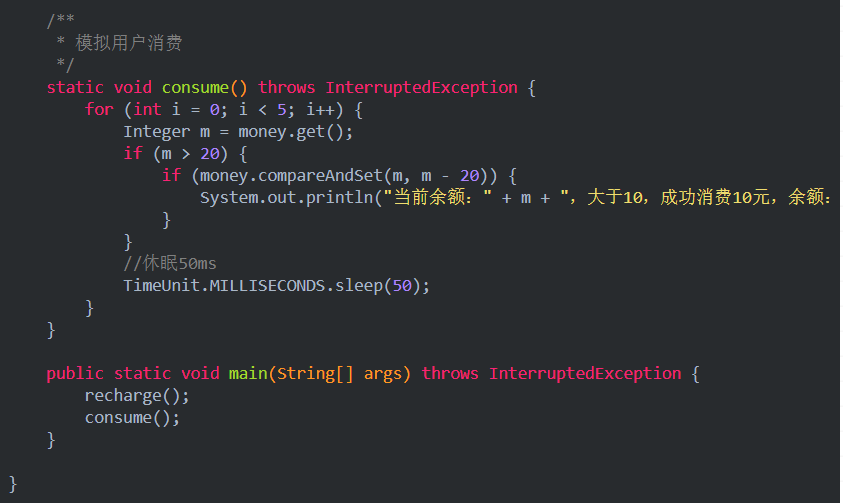
### ABA问题

之前我们说过，线程判断被修改对象是否可以正确写入的条件是对象的当前值和期望值是否一致。这个逻辑从一般意义上来说是正确的，但是可能出现一个小小的例外，就是当你获得当前数据后，在准备修改为新值钱，对象的值被其他线程连续修改了两次，而经过这2次修改后，对象的值又恢复为旧值，这样，当前线程就无法正确判断这个对象究竟是否被修改过，这就是所谓的ABA问题，可能会引发一些问题。

**举个例子**

有一家蛋糕店，为了挽留客户，决定为贵宾卡客户一次性赠送20元，刺激客户充值和消费，但条件是，每一位客户只能被赠送一次，现在我们用AtomicReference来实现这个功能，代码如下：





输出：



从输出中可以看到，这个账户被先后反复多次充值。其原因是账户余额被反复修改，修改后的值和原有的数值19一样，使得CAS操作无法正确判断当前数据是否被修改过（是否被加过20）。虽然这种情况出现的概率不大，但是依然是有可能出现的，因此，当业务上确实可能出现这种情况时，我们必须多加防范。JDK也为我们考虑到了这种情况，使用**AtomicStampedReference**可以很好地解决这个问题。

### 使用AtomicStampedReference解决ABA的问题

AtomicReference无法解决上述问题的根本原因是，对象在被修改过程中丢失了状态信息，比如充值20元的时候，需要同时标记一个状态，用来标注用户被充值过。因此我们只要能够记录对象在修改过程中的状态值，就可以很好地解决对象被反复修改导致线程无法正确判断对象状态的问题。

AtomicStampedReference正是这么做的，他内部不仅维护了对象的值，还维护了一个时间戳（我们这里把他称为时间戳，实际上它可以使用任何一个整形来表示状态值），当AtomicStampedRefernce对应的数值被修改时，除了更新数据本身外，还必须要更新时间戳。当AtomicStampedReference设置对象值时，对象值及时间戳都必须满足期望值，写入才会成功。因此，即使对象值被反复读写，写回原值，只要时间戳发生变量，就能防止不恰当的写入。

AtomicStampedRerence的几个Api在AtomicReference的基础上新增了有关时间戳的信息。

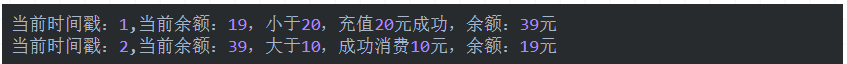
//比较设置，参数依次为：期望值、写入新值、期望时间戳、新时间戳  
public boolean compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp);  
//获得当前对象引用  
public V getReference();  
//获得当前时间戳  
public int getStamp();  
//设置当前对象引用和时间戳  
public void set(V newReference, int newStamp);

现在我们使用AtomicStampedRerence来修改一下上面充值的问题，代码如下：





输出：



结果正常了。

**关于这个时间戳的，在数据库修改数据中也有类似的用法，比如2个编辑同时编辑一篇文章，同时提交，只允许一个用户提交成功，提示另外一个用户：博客已被其他人修改，如何实现呢？**

博客表：t\_blog（id,content,stamp)，stamp默认值为0，每次更新+1

A、B 二个编辑同时对一篇文章进行编辑，stamp都为0，当点击提交的时候，将stamp和id作为条件更新博客内容，执行的sql如下：

**update t\_blog set content = 更新的内容,stamp = stamp+1 where id = 博客id and stamp = 0;**

这条update会返回影响的行数，只有一个会返回1，表示更新成功，另外一个提交者返回0，表示需要修改的数据已经不满足条件了，被其他用户给修改了。这种修改数据的方式也叫乐观锁。

## **对象的属性修改原子类介绍**

如果需要原子更新某个类里的某个字段时，需要用到对象的属性修改原子类。

* AtomicIntegerFieldUpdater：原子更新整形字段的值
* AtomicLongFieldUpdater：原子更新长整形字段的值
* AtomicReferenceFieldUpdater ：原子更新应用类型字段的值

**要想原子地更新对象的属性需要两步：**

**第一步，因为对象的属性修改类型原子类都是抽象类，所以每次使用都必须使用静态方法 newUpdater()创建一个更新器，并且需要设置想要更新的类和属性;**

**第二步，更新的对象属性必须使用 public volatile 修饰符。**

**上面三个类提供的方法几乎相同，所以我们这里以AtomicReferenceFieldUpdater为例子来介绍。**

调用AtomicReferenceFieldUpdater静态方法newUpdater创建AtomicReferenceFieldUpdater对象

public static <U, W> AtomicReferenceFieldUpdater<U, W> newUpdater(Class<U> tclass, Class<W> vclass, String fieldName)

说明:

三个参数

tclass：需要操作的字段所在的类  
vclass：操作字段的类型  
fieldName：字段名称

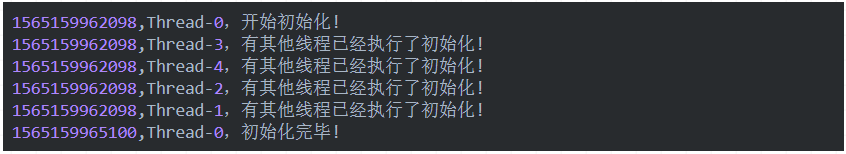
**示例**

多线程并发调用一个类的初始化方法，如果未被初始化过，将执行初始化工作，要求只能初始化一次

代码如下：



输出:



说明：

isInit属性必须要volatille修饰，可以确保变量的可见性

可以看出多线程同时执行init()方法，只有一个线程执行了初始化的操作，其他线程跳过了。多个线程同时到达updater.compareAndSet，只有一个会成功。