java.util.concurrent.atomic包介绍

# atom、atomicity、atomic、atomiclly、

**atom** 英 [ˈætəm] 美 [ˈætəm] n. 原子;原子能;微粒，微量

**atomic** 英[əˈtɒmɪk] 美[əˈtɑ:mɪk] adj. **原子的;** 原子能的，原子武器的; 极微的;

atomically英 [ə'tɒmɪklɪ] 美 [ə'tɒmɪklɪ] adv. 原子级的;利用原子能地

[例句]Each atomic cluster is made up of neutrons and protons. 原子簇是由中子和质子构成的。

**atomicity**英 [ˌætəmˈɪsɪti] 美 [ˌætəˈmɪsɪti] n. 原子数;原子价;原子化合力;原子力

事务的ACID中A就是**Atomicity**。

**accumulate** 英 [əˈkju:mjəleɪt] 美 [əˈkjumjəˌlet] vt.& vi.堆积，积累

vi. （数量）逐渐增加，（质量）渐渐提高

**accumulator**英 [əˈkju:mjəleɪtə(r)] 美 [əˈkjumjəˌletɚ]

n. <英>蓄电池，积聚者;蓄势器;收集器;蓄电池组

# java.util.concurrent.atomic包的类

## 简单原子类

* AtomicBoolean ：原子更新布尔类型
* AtomicInteger： 原子更新整型
* AtomicLong: 原子更新长整型
* AtomicReference ：原子更新引用类型。(任意类型)解决了CAS的“**只能保证一个共享变量的原子操作**”的问题。

## 数组原子类

* AtomicBooleanArray ：原子更新布尔类型数组的元素
* AtomicIntegerArray ：原子更新整型数组里的元素
* AtomicLongArray :原子更新长整型数组里的元素
* AtomicReferenceArray : 原子更新引用类型数组的元素

## 字段更新器

* AtomicIntegerFieldUpdater:原子更新整型的字段的更新器
* AtomicLongFieldUpdater：原子更新长整型字段的更新器
* AtomicReferenceFieldUpdater ：原子更新引用类型里的字段

## 特殊原子类

* **AtomicStampedReference**:原子更新带有版本号的引用类型。该类将整型数值与引用关联起来，可用于原子的更新数据和数据的版本号，**可以解决使用CAS进行原子更新时可能出现的ABA问题**。
* AtomicMarkableReference：原子更新带有标记位的引用类型。**可以原子更新一个布尔类型的标记位和应用类型**

# java.util.concurrent.atomic包介绍-Description

Package **java.util.concurrent.atomic** Description

**java.util.concurrent.atomic包提供了一系列原子类。**这些类可以保证多线程环境下，当某个线程在执行 atomic 的方法时，不会被其他线程打断，而别的线程就像**自旋锁**一样，一直等到该方法执行完成，才由 JVM 从等待队列中选择一个线程执行。**Atomic 类在软件层面上是非阻塞的，它的原子性其实是在硬件层面上借助相关的指令来保证的**。

A small toolkit of classes **that support lock-free thread-safe programming on single variables**. In essence, the classes in this package extend **the notion of volatile values, fields, and array elements** to those that also provide an atomic conditional update operation of the form:

boolean **compareAndSet**(expectedValue, updateValue);

This method (which varies in argument types across different classes) atomically sets a variable to the updateValue if it currently holds the **expectedValue**, reporting true on success. The classes in this package also contain methods to get and unconditionally set values, as well as a weaker conditional atomic update operation **weakCompareAndSet** described below.

The specifications of these methods enable implementations to employ efficient machine-level atomic instructions that are available on contemporary processors. However on some platforms, support may entail some form of internal locking. Thus the methods are not strictly guaranteed to be **non-blocking** -- a thread may block transiently before performing the operation.

Instances of classes **AtomicBoolean**, **AtomicInteger**, **AtomicLong**, and **AtomicReference** each provide access and updates to a single variable of the corresponding type. Each class also provides appropriate utility methods for that type. For example, classes **AtomicLong** and **AtomicInteger** provide atomic increment methods. One application is to generate sequence numbers, as in:

class Sequencer {

private final AtomicLong sequenceNumber = **new AtomicLong(0);**

public long next() {

return **sequenceNumber.getAndIncrement();**

}

}

It is straightforward to define new utility functions that, like **getAndIncrement**, apply a function to a value atomically. For example, given some transformation

**long transform(long input)**

write your utility method as follows:

long getAndTransform(AtomicLong var) {

long prev, next;

do {

prev = var.get();

next = transform(prev);

} while (!var.compareAndSet(prev, next));

return prev; // return next; for transformAndGet

}

The memory effects for accesses and updates of atomics generally follow the rules for volatiles, as stated in The Java Language Specification (17.4 Memory Model):

### **get** has the memory effects of reading a volatile variable.

### **set** has the memory effects of writing (assigning) a volatile variable.

### **lazySet** has the memory effects of writing (assigning) **a volatile variable** except that it permits reorderings with subsequent (but not previous) memory actions that do not themselves impose reordering constraints with ordinary non-volatile writes. Among other usage contexts, lazySet may apply when nulling out, for the sake of garbage collection, a reference that is never accessed again.

### **weakCompareAndSet** atomically reads and conditionally writes a variable but does not create any happens-before orderings, so provides no guarantees with respect to previous or subsequent reads and writes of any variables other than the target of the weakCompareAndSet.

### **compareAndSet** and all other read-and-update operations such as getAndIncrement have the memory effects of both reading and writing volatile variables.

In addition to **classes** representing **single values**, this package contains **Updater** classes that can be used to obtain **compareAndSet** operations on any selected volatile field of any selected class. **AtomicReferenceFieldUpdater**, **AtomicIntegerFieldUpdater**, and **AtomicLongFieldUpdater** are reflection-based utilities that provide access to the associated field types. These are mainly of use in atomic data structures in which several volatile fields of the same node (for example, the links of a tree node) are independently subject to atomic updates. These classes enable greater flexibility in how and when to use atomic updates, **at the expense of** more awkward reflection-based setup, less convenient usage, and weaker guarantees.

The **AtomicIntegerArray**, **AtomicLongArray**, and **AtomicReferenceArray** classes further extend atomic operation support to arrays of these types. These classes are also notable in providing volatile access semantics for their array elements, which is not supported for ordinary arrays.

The atomic classes also support method **weakCompareAndSet**, which has limited applicability. On some platforms, the weak version may be more efficient than compareAndSet in the normal case, but differs in that any given invocation of the **weakCompareAndSet** method may return false spuriously (that is, for no apparent reason). A false return means only that the operation may be retried if desired, relying on the guarantee that repeated invocation when the variable holds expectedValue and no other thread is also attempting to set the variable will eventually succeed. (Such spurious failures may for example be due to memory contention effects that are unrelated to whether the expected and current values are equal.) Additionally **weakCompareAndSet** does not provide ordering guarantees that are usually needed for synchronization control. However, the method may be useful for updating counters and statistics when such updates are unrelated to the other happens-before orderings of a program. When a thread sees an update to an atomic variable caused by a weakCompareAndSet, it does not necessarily see updates to any other variables that occurred before the **weakCompareAndSet**. This may be acceptable when, for example, updating performance statistics, but rarely otherwise.

The **AtomicMarkableReference** class associates a single boolean with a reference. For example, this bit might be used inside a data structure to mean that the object being referenced has logically been deleted. The AtomicStampedReference class associates an integer value with a reference. This may be used for example, to represent version numbers corresponding to series of updates.

**Atomic classes** are designed primarily as building blocks for implementing **non-blocking** data structures and related infrastructure classes. The **compareAndSet** method is not a general replacement for locking. It applies only when critical updates for an object are confined to a single variable.

**Atomic classes** are not general purpose replacements for **java.lang.Integer** and **related classes**. They do not define methods such as equals, hashCode and compareTo. (Because atomic variables are expected to be mutated, they are poor choices for hash table keys.) Additionally, classes are provided only for those types **that are commonly useful in intended applications**. For example, there is no atomic class for representing byte. In those infrequent cases where you would like to do so, you can use an **AtomicInteger** to hold byte values, and **cast** appropriately. You can also hold floats using **Float.floatToRawIntBits(float) and Float.intBitsToFloat(int)** conversions, and doubles using **Double.doubleToRawLongBits(double) and Double.longBitsToDouble(long)** conversions.

这里只是为使用比较频繁的类型定义了原子类，如AtomicInteger。对于Byte类型，可以利用AtomicInteger实现，并转换。对于Float和Double也可以先转为int类型，然后再转为Float和Double。

# 原子类：AtomicXxx分成5部分

注意：8种基本数据类型中，只提供了3中AtomicXxx原子类：**AtomicInteger**、**AtomicLong**、**AtomicBoolean**。对于byte、short、char、float、double都可以通过AtomicInteger实现原子操作。其中，AtomicInteger与AtomicLong对应数组AtomicIntegerArray和AtomicLongArray。

java.util.concurrent.atomic包中的所有原子类主要分成这几部分：

### **AtomicBoolean** (easy)

### 与**Integer**有关的

**AtomicInteger、AtomicIntegerArray、AtomicIntegerFieldUpdater**<T>

### 与Long有关

**AtomicLong、AtomicLongArray、AtomicLongFieldUpdater**<T>

### 与**Reference有关的(重要)**

**AtomicReference<V>、AtomicReferenceArray**<E>、**AtomicReferenceFieldUpdater**<T,V>、**AtomicStampedReference<V>、AtomicMarkableReference**<V>

### **Adder、Accumulator**

LongAdder、LongAccumulator、DoubleAdder、DoubleAccumulator

**AtomicBoolean** A boolean value that may be updated **atomically**.

**AtomicInteger** An int value that may be updated atomically.

**AtomicIntegerArray** An int array in which elements may be updated atomically.

**AtomicIntegerFieldUpdater**<T> A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes.

**AtomicLong** A long value that may be updated atomically.

**AtomicLongArray** A long array in which elements may be updated atomically.

**AtomicLongFieldUpdater**<T> A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes.

**AtomicMarkableReference**<V> An AtomicMarkableReference maintains an object reference along with a mark bit, that can be updated atomically.

**AtomicReference**<V> An object reference that may be updated atomically.

**AtomicReferenceArray**<E> An array of object references in which elements may be updated atomically.

**AtomicReferenceFieldUpdater**<T,V> A reflection-based utility that enables atomic updates to designated volatile reference fields of designated classes.

**AtomicStampedReference<V>** An AtomicStampedReference maintains an object reference along with an integer "stamp", that can be updated atomically.

**DoubleAccumulator** One or more variables that together maintain a running double value updated using a supplied function.

**DoubleAdder** One or more variables that together maintain an initially zero double sum.

**LongAccumulator** One or more variables that together maintain a running long value updated using a supplied function.

**LongAdder** One or more variables that together maintain an initially zero long sum.

# AtomicInteger 与AtomicIntegerArray API介绍

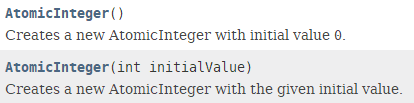
## 继承关系介绍

public class **AtomicInteger** extends **Number** implements **Serializable**

**An int value that may be updated atomically.** See the java.util.concurrent.atomic package specification for description of the properties of atomic variables. An **AtomicInteger** is used in applications such as **atomically incremented counters**, and cannot be used as a replacement for an Integer. However, this class does extend Number to allow uniform access by tools and utilities that deal with numerically-based classes.

通常应用:自增计数器**atomically incremented counters**

## 构造函数



## 一般方法

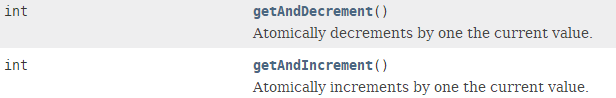
### get和set、**getAndSet**

int **get**() Gets the current value.

void **set**(int newValue) Sets to the given value.

int **getAndSet**(int newValue) **Atomically** sets to the given value and returns the old value.

### getAndDecrement() 和 getAndIncrement()：**获取原来的值**



### decrementAndGet和incrementAndGet：获取的是新值。

int **decrementAndGet**() Atomically decrements by one the current value.

int **incrementAndGet**() Atomically increments by one the current value.

### **compareAndSet**

boolean **compareAndSet**(int expect, int update)

Atomically sets the value to the given updated value if the current value == the expected value.

### **addAndGet和getAndAdd**

int addAndGet(int delta) Atomically adds the given value to the current value.

int getAndAdd(int delta) Atomically adds the given value to the current value.

### **getAndAccumulate和accumulateAndGet**

int **getAndAccumulate**(int x, **IntBinaryOperator** accumulatorFunction)

Atomically updates the current value with the results of applying the given function to the current and given values, returning the previous value.

int **accumulateAndGet**(int x, **IntBinaryOperator** accumulatorFunction)

Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.

### **doubleValue和floatValue**

double **doubleValue**()

Returns the value of this AtomicInteger as a double after a widening primitive conversion.

float **floatValue**()

Returns the value of this AtomicInteger as a float after a widening primitive conversion.

### String toString() Returns the String representation of the current value.

### 继承自Number的方法

**byteValue**, **shortValue**

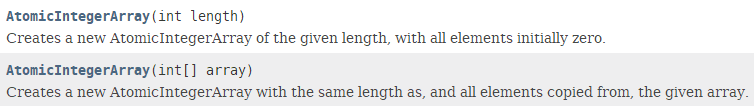
## AtomicIntegerArray API

### 继承关系

public class **AtomicIntegerArray** extends **Object** implements **Serializable**

**An int array in which elements may be updated atomically.**

### 构造函数



### 一般方法：**增加索引参数int index**

**与AtomicInteger的区别**：在AtomicInteger的所有方法中多了一个参数(第一个参数)，int index，表示要**操作的数组索引**。因此无论AtomicInteger还是**AtomicIntegerArray**一次都是对一个**int 值**进行操作。

# AtomicIntegerFieldUpdater <T> 更新器

## 继承关系

public abstract class **AtomicIntegerFieldUpdater**<T> extends **Object**



Type Parameters: T - The type of the object holding the updatable field

## 功能介绍

对指定类的实例对象的**volatile** 修饰的指定Integer变量进行原子更新操作。

基于**反射机制**的。

**A reflection-based utility that enables atomic updates to designated volatile int fields of designated classes**. This class is designed for use in atomic data structures in which several fields of the same node are independently subject to atomic updates.

**Note** that the guarantees of the **compareAndSet** method in this class are weaker than in other atomic classes. Because this class cannot ensure that all uses of the field are appropriate for purposes of atomic access, it can guarantee atomicity only with respect to other invocations of **compareAndSet** and **set** **on the same updater.**

**只能保证compareAndSet** and **set发生在同一个updater上实现原子性。**

## 创建实例方法

利用静态方法获取实例：

static <U> **AtomicIntegerFieldUpdater**<U> **newUpdater**(Class<U> tclass, String fieldName)

Creates and returns an updater for objects with the given field.

示例:

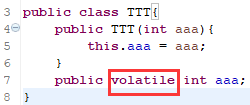
AtomicIntegerFieldUpdater<TTT> afu =

AtomicIntegerFieldUpdater.*newUpdater*(TTT.**class**, "aaa");

TTT tt = **new** TTT(100);

System.***out***.println(afu.addAndGet(tt, 10));//110

注意:更新的变量必须是**volatile**修饰的。



## 一般方法

与AtomicInteger相比，前面多了一个参数**T obj**。

# AtomicInteger、AtomicIntegerArray、AtomicIntegerFieldUpdater<T>三者的区别与联系

### AtomicInteger是最基本的，其只含有一个Integer值，并对该值进行**原子操作**。

### AtomicIntegerArray是一个数组，其包含了多个Integer值，但是同时只能对一个Integer值进行操作，所以相比AtomicInteger的方法都多了个**参数int index**。

### AtomicIntegerFieldUpdater<T>是一个**更新器Updater**，利用了**反射机制**的原理，对指定T类型的对象实例的属性变量进行原子操作，该属性变量必须是volatile修饰的**Integer类型或int**。

# AtomicLong 与 AtomicLongArray API

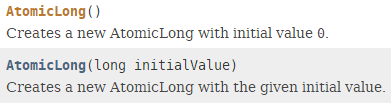
## AtomicLong继承关系

public class **AtomicLong** extends **Number** mplements **Serializable**

A long value that may be updated atomically. See the java.util.concurrent.atomic package specification for description of the properties of atomic variables. An AtomicLong is used in applications such as **atomically incremented sequence numbers**, and cannot be used as a replacement for a Long. However, this class does extend Number to allow uniform access by tools and utilities that deal with numerically-based classes.

应用场景：**atomically incremented sequence numbers**

## AtomicLong构造函数

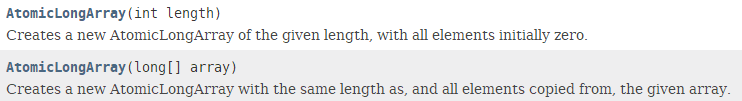


## 与AtomicInteger类似，省略。

## AtomicLongArray

public class AtomicLongArray extends Object implements Serializable

A long array in which elements may be updated atomically. See the java.util.concurrent.atomic package specification for description of the properties of atomic variables.



AtomicLongArray的方法与AtomicLong相比，都是多了一个参数，数组的索引int index。

# AtomicLongFieldUpdater<T>

与AtomicIntegerFieldUpdater<T>类似。

## 继承关系

public abstract class **AtomicLongFieldUpdater**<T> extends Object

A reflection-based utility that enables atomic updates to designated volatile long fields of designated classes. This class is designed for use in atomic data structures in which several fields of the same node are independently subject to atomic updates.

Note that the guarantees of the **compareAndSet** method in this class are weaker than in other atomic classes. Because this class cannot ensure that all uses of the field are appropriate for purposes of atomic access, it can guarantee atomicity only with respect to other invocations of compareAndSet and set on the same updater.

## 创建实例方法

static <U> **AtomicLongFieldUpdater<U>** **newUpdater**(Class<U> tclass, String fieldName)

Creates and returns an updater for objects with the given field.

## 一般方法

相对于AtomicLong来说，多个参数T obj。

# AtomicBoolean API

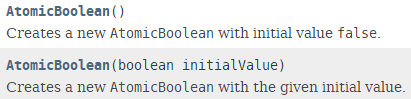
## 继承关系

public class **AtomicBoolean** extends Object implements Serializable

A boolean value that may be updated atomically. See the java.util.concurrent.atomic package specification for description of the properties of atomic variables. An **AtomicBoolean** is used in applications such as **atomically updated flags**, and cannot be used as a replacement for a Boolean.

通常应用：**原子性更新flag**。**atomically updated flags**

## 构造函数



## 一般方法

### get、set、**getAndSet、lazySet**

boolean get() Returns the current value.

void set(boolean newValue) **Unconditionally** sets to the given value.

boolean **getAndSet**(boolean newValue)

Atomically sets to the given value and returns the previous value.

void **lazySet**(boolean newValue) Eventually sets to the given value

### **compareAndSet/ weakCompareAndSet**

boolean **compareAndSet**(boolean expect, boolean update)

Atomically sets the value to the given updated value if the current value == the expected value.

boolean **weakCompareAndSet**(boolean expect, boolean update)

Atomically sets the value to the given updated value if the current value == the expected value.

May fail spuriously and does not provide ordering guarantees, so is only rarely an appropriate alternative to compareAndSet.

# LongAdder API

## 继承关系介绍

public class **LongAdder** extends **Number** implements Serializable



**since:1.8**

## 功能介绍

One or more variables that together maintain an initially zero long sum. When updates (method add(long)) are contended **across threads**, the set of variables may grow dynamically to reduce contention. Method **sum()** (or, equivalently, longValue()) returns the current total combined across the variables maintaining the sum.

**This class is usually preferable to AtomicLong when multiple threads update a common sum that is used for purposes such as collecting statistics**, not for fine-grained synchronization control. Under low update contention, the two classes have similar characteristics. But under high contention, expected throughput of this class is significantly higher, at the expense of higher space consumption.

LongAdders can be used with a ConcurrentHashMap to maintain a scalable frequency map (a form of histogram or multiset). For example, to add a count to a ConcurrentHashMap<String,LongAdder> freqs, initializing if not already present, you can use freqs.computeIfAbsent(k -> new **LongAdder()).increment();**

This class extends Number, but does not define methods such as equals, hashCode and compareTo because instances are expected to be mutated, and so are not useful as collection keys.

## 构造函数



## 一般方法

### void **add**(long x) Adds the given value.

### **increment和decrement**

void increment() Equivalent to add(1).

void decrement() Equivalent to add(-1).

### long sum() Returns the current sum.

### void **reset**() Resets variables maintaining the sum to zero.

### long **sumThenReset**() Equivalent in effect to sum() followed by reset().

intValue、longValue、floatValue、DoubleValue

### 继承Number的

byteValue, shortValue

# LongAccumulator

## 继承关系

public class **LongAccumulator** extends **Number** implements **Serializable**

## 功能介绍

One or more variables that together maintain a running long value updated using a supplied function. When updates (method **accumulate(long)**) are contended across threads, the set of variables may grow dynamically to reduce contention. Method **get()** (or, equivalently, longValue()) returns the current value across the variables maintaining updates.

This class **is usually preferable to** **AtomicLong** when multiple threads update a common value that is used for purposes such as collecting statistics, not for fine-grained synchronization control. Under low update contention, the two classes have similar characteristics. But under high contention, expected throughput of this class is significantly higher, at the expense of higher space consumption.

The order of accumulation within or across threads is not guaranteed and cannot be depended upon, so this class is only applicable to functions for which the order of accumulation does not matter. The supplied accumulator function should be **side-effect-free**, since it may be re-applied when attempted updates fail due to contention among threads. The function is applied with the current value as its first argument, and the given update as the second argument. For example, to maintain **a running maximum value**, you could supply Long::max along with **Long.MIN\_VALUE** as the identity.

Class **LongAdder** provides analogs of the functionality of this class for the common special case of maintaining counts and sums. The call new **LongAdder()** is equivalent to new **LongAccumulator((x, y) -> x + y, 0L.**

This class extends Number, but does not define methods such as equals, hashCode and compareTo because instances are expected to be mutated, and so are not useful as collection keys.

Since: 1.8

## 构造函数

**LongAccumulator**(LongBinaryOperator accumulatorFunction, long identity)

Creates a new instance using the given accumulator function and identity element.

## 一般方法

### void **accumulate**(long x) Updates with the given value.

### long get() Returns the current value.

### void reset() Resets variables maintaining updates to the identity value.

### long getThenReset() Equivalent in effect to get() followed by reset().

### intValue、longValue、floatValue、doubleValue

### Number的byteValue, shortValue

# DoubleAdder 与DoubleAccumulator API

## DoubleAdder继承关系

public class **DoubleAdder** extends **Number** implements **Serializable**



Since: 1.8

## 功能介绍

One or more variables that together maintain an initially zero double sum. When updates (method add(double)) are contended across threads, the set of variables may grow dynamically to reduce contention. Method **sum()** (or, equivalently doubleValue()) returns the current total combined across the variables maintaining the sum. The order of accumulation within or across threads is not guaranteed. Thus, this class may not be applicable if numerical stability is required, especially when combining values of substantially different orders of magnitude.

**This class is usually preferable to alternatives when multiple threads update a common value that is used for purposes such as summary statistics** that are frequently updated but less frequently read.

This class extends Number, but does not define methods such as equals, hashCode and compareTo because instances are expected to be mutated, and so are not useful as collection keys.

## 构造方法

DoubleAdder() Creates a new adder with initial sum of zero.

## 方法与LongAdder类似，省略。

## DoubleAccumulator

public class **DoubleAccumulator** extends **Number** implements Serializable

特性、方法与LongAccumulator类相似。