与Reference相关的原子类

# 与Reference有关的5个原子类

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

## AtomicXxx<V>、AtomicXxxArray<E>、AtomicXxxFieldUpdater<T,V>

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

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

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

对于Reference是可以对所有类型都可以进行操作，而Integer、Long算是其中两个特例。

都是对应AtomicXxx、AtomicXxxArray、AtomicXxxFieldUpdater<T>，AtomicXxx是直接对Xxx类型进行原子操作，而AtomicXxxArray是一个数组，通过索引单独对数组中的每一个Xxx进行原子操作，AtomicXxxFieldUpdater<T>是通过对T类中的Xxx变量进行原子操作。

# AtomicReference<V>

## 继承关系介绍

public class AtomicReference<V> extends Object implements Serializable

**An object reference that may be updated atomically.** See the java.util.concurrent.atomic package specification for description of the properties of atomic variables.

Type Parameters V - The type of object referred to by this reference

## 构造函数

**AtomicReference**()

Creates a new AtomicReference with null initial value.

**AtomicReference**(V initialValue)

Creates a new AtomicReference with the given initial value.

## 一般方法

### get与set方法、**lazySet**

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

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

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

### **getAndAccumulate**

V **getAndAccumulate**(V x, BinaryOperator<V> accumulatorFunction)

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

### **getAndSet**

V **getAndSet**(V newValue)

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

### **getAndUpdate与**updateAndGet

V **getAndUpdate**(UnaryOperator<V> updateFunction)

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

V **updateAndGet**(UnaryOperator<V> updateFunction)

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

### V **accumulateAndGet**(V x, BinaryOperator<V> accumulatorFunction)

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

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

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

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

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

# AtomicReferenceArray

## 继承关系

public class **AtomicReferenceArray**<E> extends Object implements Serializable



An array of object references in which elements may be updated atomically. See the **java.util.concurrent.atomic** package specification for description of the properties of atomic variables.

Type Parameters: E - The base class of elements held in this array

## 构造函数

**AtomicReferenceArray**(**E[] array**)

Creates a new AtomicReferenceArray with the same length as, and all elements copied from, the given array.

**AtomicReferenceArray**(**int length**)

Creates a new AtomicReferenceArray of the given length, with all elements initially null.

## 一般方法

与**AtomicReference**相比，方法都多了一个int index，表示数组的索引。

# AtomicReferenceFieldUpdater<T,V>

## 继承关系

public abstract class AtomicReferenceFieldUpdater<T,V> extends Object

## 功能介绍

A **reflection-based utility** that enables atomic updates to designated volatile reference fields of designated classes. This class is designed for use in atomic data structures in which several reference fields of the same node are independently subject to atomic updates. For example, a tree node might be declared as

**class Node {**

**private volatile Node left, right;**

**private static final AtomicReferenceFieldUpdater<Node, Node> leftUpdater =**

**AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "left");**

**private static AtomicReferenceFieldUpdater<Node, Node> rightUpdater =**

**AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "right");**

**Node getLeft() { return left; }**

**boolean compareAndSetLeft(Node expect, Node update) {**

**return leftUpdater.compareAndSet(this, expect, update);**

**}**

**// ... and so on**

**}**

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

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

通过静态方法**newUpdater**创建更新实例对象。

例如：

**private static final AtomicReferenceFieldUpdater<Node, Node> leftUpdater =**

**AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "left");**

**private static AtomicReferenceFieldUpdater<Node, Node> rightUpdater =**

**AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "right");**

## 一般方法

与AtomicReference相比，方法多了个参数:T obj。

# AtomicMarkableReference<V>

## 继承关系

public class **AtomicMarkableReference<V>** extends Object



Type Parameters: V - The type of object referred to by this reference

## 功能

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

Implementation note: This implementation **maintains markable references** by creating internal objects representing "boxed**" [reference, boolean] pairs**.

## 构造方法

AtomicMarkableReference(V initialRef, boolean initialMark)

Creates a new **AtomicMarkableReference** with the given initial values.

## 一般方法

### get方法

V get(**boolean**[] markHolder)

Returns the current values of both the reference and the mark.

### getReference

V getReference() Returns the current value of the reference.

### set

void set(V newReference, boolean newMark)

Unconditionally sets the value of both the reference and mark.

### **isMarked**

boolean isMarked() Returns the current value of the mark.

### **attemptMark**

boolean attemptMark(V expectedReference, boolean newMark)

Atomically sets the value of the mark to the given update value if the current reference is == to the expected reference.

### **compareAndSet**

boolean compareAndSet(V expectedReference, V newReference, boolean expectedMark, boolean newMark)

Atomically sets the value of both the reference and mark to the given update values if the current reference is == to the expected reference and the current mark is equal to the expected mark.

### **weakCompareAndSet**

boolean **weakCompareAndSet**(V expectedReference, V newReference, boolean expectedMark, boolean newMark)

**Atomically** sets the value of both the reference and mark to the given update values if the current reference is == to the expected reference and the current mark is equal to the expected mark.

# AtomicStampedReference<V>

## 继承关系介绍

public class **AtomicStampedReference<V>** extends Object



## 功能介绍

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

Implementation note: This implementation maintains stamped references by creating internal objects representing **"boxed" [reference, integer] pairs**.

在 Java 中，**AtomicStampedReference<E>**也实现了这个作用，它通过包装**[E,Integer]的元组**来对对象标记**版本戳 stamp**，从而避免 ABA 问题。

Atomically sets the value of both the reference and stamp to the given update values **if the current reference is == to the expected reference and the current stamp is equal to the expected stamp.只有两个值都相等的时候，才会设置新的值。**

## 构造函数

**AtomicStampedReference**(V initialRef, int initialStamp)

Creates a new AtomicStampedReference with the given initial values.

## 一般方法

### attemptStamp

boolean **attemptStamp**(V expectedReference, int newStamp)

Atomically sets the value of the stamp to the given update value if the current reference is == to the expected reference.

### get

V **get**(int[] stampHolder)

Returns the current values of both the reference and the stamp.

### getReference

V **getReference**()

Returns the current value of the reference.

### getStamp

int getStamp()

Returns the current value of the stamp.

### set

void set(V newReference, int newStamp)

Unconditionally sets the value of both the reference and stamp.

### **compareAndSet**

boolean **compareAndSet**(V expectedReference, V newReference, int expectedStamp, int newStamp)

Atomically sets the value of both the reference and stamp to the given update values **if the current reference is == to the expected reference and the current stamp is equal to the expected stamp.**

### **weakCompareAndSet**

boolean **weakCompareAndSet**(V expectedReference, V newReference, int **expectedStamp**, int **newStamp**)

Atomically sets the value of both the reference and stamp to the given update values **if the current reference is == to the expected reference** **and the current stamp is equal to the expected stamp.**

## 示例：

AtomicStampedReference<Integer> asr = new AtomicStampedReference<Integer>(10, 1);

System.out.println(asr.getStamp());//1

System.out.println(asr.getReference());//10

System.out.println(asr.compareAndSet(10, 100, 2, 3));//false reference和Integer都不变

// System.out.println(asr.compareAndSet(10, 100, 1, 3));true

System.out.println(asr.getStamp());//1 //100

System.out.println(asr.getReference());//10//3

# CAS的ABA问题

因为CAS需要在操作值的时候检查下值有没有发生变化，如果没有发生变化则更新，但是如果一个值原来是A，变成了B，又变成了A，那么使用CAS进行检查时会发现它的值没有发生变化，操作成功，但是实际上却变化了。

**ABA问题的解决思路就是使用版本号**。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A－B－A 就会变成1A-2B-3A。

从Java1.5开始JDK的**atomic**包里提供了一个类**AtomicStampedReference**来解决ABA问题。这个类的**compareAndSet方法**作用是首先检查当前引用是否等于预期引用，并且当前标志是否等于预期标志，如果全部相等，则以原子方式将该引用和该标志的值设置为给定的更新值。

CAS 算法实现一个重要前提是需要取出内存中某时刻的数据，而在下一时刻把取出后的数据和内存中原始数据比较并替换，那么在这个时间差内会导致数据的变化。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。

ABA 问题带来的隐患，各种乐观锁的实现中通常都会用**版本号 version** 来对记录或对象标记，避免并发操作带来的问题。

在 Java 中，**AtomicStampedReference<E>**也实现了这个作用，它通过包装**[E,Integer]的元组**来对对象标记**版本戳 stamp**，从而避免 ABA 问题。