参考：<http://blog.csdn.net/u011116672/article/details/51068828>

[**Java并发编程系列之十九：原子操作类**](http://blog.csdn.net/u011116672/article/details/51068828)

版权声明：本文为博主原创文章，转载请注明出处。

**原子操作类简介**

当更新一个变量的时候，多出现数据争用的时候可能出现所意想不到的情况。这时的一般策略是使用synchronized解决，因为synchronized能够保证多个线程不会同时更新该变量。然而，从jdk 5之后，提供了粒度更细、量级更轻，并且在多核处理器具有高性能的原子操作类。因为原子操作类把竞争的范围缩小到单个变量上，这可以算是粒度最细的情况了。

原子操作类相当于泛化的volatile变量，能够支持原子读取-修改-写操作。比如AtomicInteger表示一个int类型的数值，提供了get和set方法，这些volatile类型的变量在读取与写入上有着相同的内存语义。原子操作类共有13个类，在java.util.concurrent.atomic包下，可以分为四种类型的原子更新类：原子更新基本类型、原子更新数组类型、原子更新引用和原子更新属性。

下面将分别介绍这四种原子操作类。

**原子更新基本类型**

使用原子方式更新基本类型，共包括3个类：

* AtomicBoolean：原子更新布尔变量
* AtomicInteger：原子更新整型变量
* AtomicLong：原子更新长整型变量

具体到每个类的源代码中，提供的方法基本相同，这里以AtomicInteger为例进行说明。AtomicInteger提供的部分方法如下：

public class AtomicInteger extends Number implements java.io.Serializable {

//返回当前的值

public final int get() {

return value;

}

//原子更新为新值并返回旧值

public final int getAndSet(int newValue) {

return unsafe.getAndSetInt(this, valueOffset, newValue);

}

//最终会设置成新值

public final void lazySet(int newValue) {

unsafe.putOrderedInt(this, valueOffset, newValue);

}

//如果输入的值等于预期值，则以原子方式更新为新值

public final boolean compareAndSet(int expect, int update) {

return unsafe.compareAndSwapInt(this, valueOffset, expect, update);

}

//原子自增

public final int getAndIncrement() {

return unsafe.getAndAddInt(this, valueOffset, 1);

}

//原子方式将当前值与输入值相加并返回结果

public final int getAndAdd(int delta) {

return unsafe.getAndAddInt(this, valueOffset, delta);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

为了说明AtomicInteger的原子性，这里代码演示多线程对一个int值进行自增操作，最后输出结果，代码如下：

package com.rhwayfun.concurrency.r0405;

import java.util.concurrent.atomic.AtomicInteger;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class AtomicIntegerDemo {

private static AtomicInteger atomicInteger = new AtomicInteger(0);

public static void main(String[] args){

for (int i = 0; i < 5; i++){

new Thread(new Runnable() {

public void run() {

//调用AtomicInteger的getAndIncement返回的是增加之前的值

System.out.println(atomicInteger.getAndIncrement());

}

}).start();

}

System.out.println(atomicInteger.get());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24

输出结果如下：

0   
1   
2   
3   
4   
5   
5

可以看到在多线程的情况下，得到的结果是正确的，但是如果仅仅使用int类型的成员变量则可能得到不同的结果。这里的关键在于getAndIncrement是原子操作，那么是如何保证的呢？

getAndIncrement方法的源码如下：

public final int getAndIncrement() {

return unsafe.getAndAddInt(this, valueOffset, 1);

}

public final int getAndAddInt(Object var1, long var2, int var4) {

int var5;

do {

var5 = this.getIntVolatile(var1, var2);

} while(!this.compareAndSwapInt(var1, var2, var5, var5 + var4));

return var5;

}

public final native boolean compareAndSwapInt(Object var1, long var2, int var4, int var5);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

到这里可以发现最终调用了native方法来保证更新的原子性。

**原子更新数组**

通过原子更新数组里的某个元素，共有3个类：

* AtomicIntegerArray：原子更新整型数组的某个元素
* AtomicLongArray：原子更新长整型数组的某个元素
* AtomicReferenceArray：原子更新引用类型数组的某个元素

AtomicIntegerArray常用的方法有：

* int addAndSet(int i, int delta)：以原子方式将输入值与数组中索引为i的元素相加
* boolean compareAndSet(int i, int expect, int update)：如果当前值等于预期值，则以原子方式更新数组中索引为i的值为update值

示例代码如下：

package com.rhwayfun.concurrency.r0405;

import java.util.concurrent.atomic.AtomicIntegerArray;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class AtomicIntegerArrayDemo {

static int[] value = new int[]{1, 2};

static AtomicIntegerArray ai = new AtomicIntegerArray(value);

public static void main(String[] args){

ai.getAndSet(0,3);

System.out.println(ai.get(0));

System.out.println(value[0]);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19

运行结果是:

3   
1

数组value通过构造的方式传入AtomicIntegerArray中，实际上AtomicIntegerArray会将当前数组拷贝一份，所以在数组拷贝的操作不影响原数组的值。

**原子更新引用类型**

需要更新引用类型往往涉及多个变量，早atomic包有三个类：

* AtomicReference：原子更新引用类型
* AtomicReferenceFieldUpdater：原子更新引用类型里的字段
* AtomicMarkableReference：原子更新带有标记位的引用类型。

下面以AtomicReference为例进行说明：

package com.rhwayfun.concurrency.r0405;

import java.util.concurrent.atomic.AtomicReference;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class AtomicReferenceDemo {

static class User{

private String name;

private int id;

public User(String name, int id) {

this.name = name;

this.id = id;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

}

public static AtomicReference<User> ar = new AtomicReference<User>();

public static void main(String[] args){

User user = new User("aa",11);

ar.set(user);

User newUser = new User("bb",22);

ar.compareAndSet(user,newUser);

System.out.println(ar.get().getName());

System.out.println(ar.get().getId());

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47

运行结果为：

bb   
22

可以看到user被成功更新。

**原子更新字段类**

如果需要原子更新某个类的某个字段，就需要用到原子更新字段类，可以使用以下几个类：

* AtomicIntegerFieldUpdater：原子更新整型字段
* AtomicLongFieldUpdater：原子更新长整型字段
* AtomicStampedReference：原子更新带有版本号的引用类型。

要想原子更新字段，需要两个步骤：

1. 每次必须使用newUpdater创建一个更新器，并且需要设置想要更新的类的字段
2. 更新类的字段（属性）必须为public volatile

下面的代码演示如何使用原子更新字段类更新字段：

package com.rhwayfun.concurrency.r0405;

import java.util.concurrent.atomic.AtomicIntegerFieldUpdater;

/\*\*

\* Created by rhwayfun on 16-4-5.

\*/

public class AtomicIntegerFieldUpdaterDemo {

//创建一个原子更新器

private static AtomicIntegerFieldUpdater<User> atomicIntegerFieldUpdater =

AtomicIntegerFieldUpdater.newUpdater(User.class,"old");

public static void main(String[] args){

User user = new User("Tom",15);

//原来的年龄

System.out.println(atomicIntegerFieldUpdater.getAndIncrement(user));

//现在的年龄

System.out.println(atomicIntegerFieldUpdater.get(user));

}

static class User{

private String name;

public volatile int old;

public User(String name, int old) {

this.name = name;

this.old = old;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getOld() {

return old;

}

public void setOld(int old) {

this.old = old;

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48

输出的结果如下：

15   
16

至此，我们知道了如何使用原子操作类在不同场景下的基本用法。