Atomic原子类介绍

所谓原子类说简单点就是具有原子/原子操作特征的类。 并发包 java.util.concurrent 的原子类都存放在java.util.concurrent.atomic下。

分为4类： 基本类型： 原子方式更新基本类型: AtomicInteger：整形原子类 AtomicLong：长整型原子类 AtomicBoolean：布尔型原子类

数组类型： 原子方式更新数组中某个元素 AtomicIntegerArray：整型数组原子类 AtomicLongArray：长整型数组原子类 AtomicReferenceArray ： 引用类型数组原子类

引用类型： AtomicReference：引用类型原子类 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

对象的属性修改类型： AtomicIntegerFieldUpdater:原子更新整型字段的更新器 AtomicLongFieldUpdater：原子更新长整型字段的更新器 AtomicStampedReference ： 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决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：原子更新带有标记的引用类型。该类将 boolean 标记与引用关联起来，也可以解决使用 CAS 进行原子更新时可能出现的 ABA 问题。

```
public class AtomicIntegerDefectDemo {
    public static void main(String[] args){
        defectOfABA();
    }
    static void defectOfABA(){
        final AtomicInteger atomicInteger = new AtomicInteger(1);

        Thread coreThread = new Thread(
                ()-> {
                    final int currentValue = atomicInteger.get();
                    System.out.println(Thread.currentThread().getName() + " ------
-------currentVakue = " + currentValue );
                    //模拟其他处理业务花费时间
                    try{
                        Thread.sleep(300);
                    }catch (InterruptedException e){
                        e.printStackTrace();
                    }

                    boolean casResult = atomicInteger.compareAndSet(1,2);
                    System.out.println(Thread.currentThread().getName() + " ------
currentValue=" + currentValue + ", finalValue=" + atomicInteger.get() + ",
compareAndSet Result=" + casResult);
                }
        );
        coreThread.start();

        Thread amateurThread = new Thread(
```

```
                () -> {
                    int currentValue = atomicInteger.get();
                    boolean casResult = atomicInteger.compareAndSet(1,2);
                    System.out.println(Thread.currentThread().getName()
                            + " ------ currentValue=" + currentValue
                            + ", finalValue=" + atomicInteger.get()
                            + ", compareAndSet Result=" + casResult);
                    currentValue = atomicInteger.get();
                    casResult = atomicInteger.compareAndSet(2,1);
                    System.out.println(Thread.currentThread().getName()
                            + " ------ currentValue=" + currentValue
                            + ", finalValue=" + atomicInteger.get()
                            + ", compareAndSet Result=" + casResult);
                }
        );
        amateurThread.start();

    }
}
```

输出为:

```
Thread-0 -------------currentVakue = 1
Thread-1 ------ currentValue=1, finalValue=2, compareAndSet Result=true
Thread-1 ------ currentValue=2, finalValue=1, compareAndSet Result=true
Thread-0 ------ currentValue=1, finalValue=2, compareAndSet Result=true
```

基本类型原子类

介绍

三类基本类型原子类方法基本相同，以AtomicInteger为例:

```
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。
```

常见方法使用:

```
public class AtomicIntegerTest {
    public static void main(String[] args){
        int temValue = 0;
```

```
        AtomicInteger i = new AtomicInteger(0);
        temValue = i.getAndSet(3);
        System.out.println("temValue: " + temValue + "; i:" + i);
        temValue = i.getAndIncrement();
        System.out.println("temValue: " + temValue + "; i:" + i);
        temValue = i.getAndAdd(5);
        System.out.println("temValue: " + temValue + "; i:" + i);
    }
}
```

多线程使用基本类型原子类的优势。基本类型原子类不用加锁就可以保证线程安全。

Atomic安全原理分析

```
private static final Unsafe unsafe = Unsafe.getUnsafe();
private static final long valueOffset;

static{
    try {
        valueOffset =
unsafe.objectFieldOffset(AtomicInteger.class.getDeclaredField("value"));
    } catch (Exception ex) {
        throw new Error(ex);
    }
}

private volatile int value;
```

AtomicInteger 类主要利用 CAS (compare and swap) + volatile 和 native 方法来保证原子操作，从而避免 synchronized 的高开销，执行效率大为提升. CAS的原理是拿期望的值和原本的一个值作比较，如果相同则更新成新的值。UnSafe 类的 objectFieldOffset() 方法是一个本地方法，这个方法是用来拿到“原来的值”的内存地址。另外 value 是一个volatile变量，在内存中可见，因此 JVM 可以保证任何时刻任何线程总能拿到该变量的最新值。

数组类型原子类

介绍

使用原子方式更新数组中的某个元素

提供的方法：

```
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。
```

常见方法使用

```
public class AtomicIntegerArrayTest {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int temvalue = 0;
        int[] nums = { 1, 2, 3, 4, 5, 6 };
        AtomicIntegerArray i = new AtomicIntegerArray(nums);
        for (int j = 0; j < nums.length; j++) {
            System.out.println(i.get(j));
        }
        temvalue = i.getAndSet(0, 2);
        System.out.println("temvalue:" + temvalue + ";  i:" + i);
        temvalue = i.getAndIncrement(0);
        System.out.println("temvalue:" + temvalue + ";  i:" + i);
        temvalue = i.getAndAdd(0, 5);
        System.out.println("temvalue:" + temvalue + ";  i:" + i);
    }
}
```

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 AtomicStampedReference：原子更新引用类型里的字段原子类
AtomicMarkableReference：原子更新带有标志位的引用类型

AtomicReference类使用示例

```
public class AtomicReferenceTest {
    public static void main(String[] args){
        AtomicReference<Person> ar = new AtomicReference<Person>();
        Person person = new Person("SnailClimb",22);
        ar.set(person);
        Person updatePerson = new Person("Daisy",20);
```

```
        ar.compareAndSet(person,updatePerson);
        System.out.println(ar.get().getName());
        System.out.println(ar.get().getAge());
    }
}
class Person{
    private String name;
    private int age;

    public Person(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

AtomicStampedReference使用示例

类似上述

```
public class AtomicStampedReferenceDemo {
    public static void main(String[] args) {
        // 实例化、取当前值和 stamp 值
        final Integer initialRef = 0, initialStamp = 0;
        final AtomicStampedReference<Integer> asr = new AtomicStampedReference<>
(initialRef, initialStamp);
        System.out.println("currentValue=" + asr.getReference() + ",
currentStamp=" + asr.getStamp());

        // compare and set
        final Integer newReference = 666, newStamp = 999;
        final boolean casResult = asr.compareAndSet(initialRef, newReference,
initialStamp, newStamp);
```

```
        System.out.println("currentValue=" + asr.getReference()
                + ", currentStamp=" + asr.getStamp()
                + ", casResult=" + casResult);

        // 获取当前的值和当前的 stamp 值
        int[] arr = new int[1];
        final Integer currentValue = asr.get(arr);
        final int currentStamp = arr[0];
        System.out.println("currentValue=" + currentValue + ", currentStamp=" +
currentStamp);

        // 单独设置 stamp 值
        final boolean attemptStampResult = asr.attemptStamp(newReference, 88);
        System.out.println("currentValue=" + asr.getReference()
                + ", currentStamp=" + asr.getStamp()
                + ", attemptStampResult=" + attemptStampResult);

        // 重新设置当前值和 stamp 值
        asr.set(initialRef, initialStamp);
        System.out.println("currentValue=" + asr.getReference() + ",
currentStamp=" + asr.getStamp());

        // [不推荐使用，除非搞清楚注释的意思了] weak compare and set
        // 困惑！weakCompareAndSet 这个方法最终还是调用 compareAndSet 方法。[版本:
jdk-8u191]
        // 但是注释上写着 "May fail spuriously and does not provide ordering
guarantees,
        // so is only rarely an appropriate alternative to compareAndSet."
        // todo 感觉有可能是 jvm 通过方法名在 native 方法里面做了转发
        final boolean wCasResult = asr.weakCompareAndSet(initialRef, newReference,
initialStamp, newStamp);
        System.out.println("currentValue=" + asr.getReference()
                + ", currentStamp=" + asr.getStamp()
                + ", wCasResult=" + wCasResult);
    }
}
```

AtomicMarkableReference使用示例

类似上述

```
public class AtomicMarkableReferenceDemo {
    public static void main(String[] args) {
        // 实例化、取当前值和 mark 值
        final Boolean initialRef = null, initialMark = false;
        final AtomicMarkableReference<Boolean> amr = new AtomicMarkableReference<>
(initialRef, initialMark);
        System.out.println("currentValue=" + amr.getReference() + ", currentMark="
+ amr.isMarked());
```

```
        // compare and set
        final Boolean newReference1 = true, newMark1 = true;
        final boolean casResult = amr.compareAndSet(initialRef, newReference1,
initialMark, newMark1);
        System.out.println("currentValue=" + amr.getReference()
                + ", currentMark=" + amr.isMarked()
                + ", casResult=" + casResult);

        // 获取当前的值和当前的 mark 值
        boolean[] arr = new boolean[1];
        final Boolean currentValue = amr.get(arr);
        final boolean currentMark = arr[0];
        System.out.println("currentValue=" + currentValue + ", currentMark=" +
currentMark);

        // 单独设置 mark 值
        final boolean attemptMarkResult = amr.attemptMark(newReference1, false);
        System.out.println("currentValue=" + amr.getReference()
                + ", currentMark=" + amr.isMarked()
                + ", attemptMarkResult=" + attemptMarkResult);

        // 重新设置当前值和 mark 值
        amr.set(initialRef, initialMark);
        System.out.println("currentValue=" + amr.getReference() + ", currentMark="
+ amr.isMarked());

        // [不推荐使用，除非搞清楚注释的意思了] weak compare and set
        // 困惑！weakCompareAndSet 这个方法最终还是调用 compareAndSet 方法。[版本:
jdk-8u191]
        // 但是注释上写着 "May fail spuriously and does not provide ordering
guarantees,
        // so is only rarely an appropriate alternative to compareAndSet."
        // todo 感觉有可能是 jvm 通过方法名在 native 方法里面做了转发
        final boolean wCasResult = amr.weakCompareAndSet(initialRef,
newReference1, initialMark, newMark1);
        System.out.println("currentValue=" + amr.getReference()
                + ", currentMark=" + amr.isMarked()
                + ", wCasResult=" + wCasResult);
    }
}
```

对象的属性修改类型原子类

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类型原子类。

AtomicIntegerFieldUpdater:原子更新整形字段的更新器 AtomicLongFieldUpdater：原子更新长整形字段的更新器 AtomicStampedReference ： 原子更新带有版本号的引用类型。该类将整数值与引用关联起来，可用于解决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

要想原子的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。并且需要设置想要的类和属性。第二步，更新的对象属性必须使用public volatile修饰符。

```
public class AtomicIntegerFieldUpdaterTest {
    public static void main(String[] args) {
        AtomicIntegerFieldUpdater<User> a =
AtomicIntegerFieldUpdater.newUpdater(User.class, "age");

        User user = new User("Java", 22);
        System.out.println(a.getAndIncrement(user));// 22
        System.out.println(a.get(user));// 23
    }
}

class User {
    private String name;
    public volatile int age;

    public User(String name, int age) {
        super();
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

}
```