Chapter 5 第5章 原子操作

本章将介绍另一种同步控制方式——原子操作。原子操作也可以保证多线程环境下的共享变量操作的正确性。

5.1 原子性

如果一个操作是原子的，则表示该操作要么全做，要么全不做。这意味着原子操作将作为一个不可分割的整体完成，在执行完毕前不会被任何其他任务或事件中断。

在同步控制方式中，原子操作是指不会被线程调度机制打断的操作，这种操作一旦开始，就一直运行到结束，中间不会有任何上下文切换。

使用原子操作后，一般不需要对临界区加锁，可以实现多线程程序的同步控制。

在单处理器系统中，能够在单条指令中完成的操作都可以认为具有原子操作的潜质，因为中断只能发生于指令之间，单条指令的执行不会发生其他指令的介入，例如 a=0。然而，需要把单条指令和单条语句的概念区分开，不是所有的程序设计语言中的一条语句都具有原子操作的潜质，例如，“a++”操作是一条语句，它不是一个原子操作，因为该操作可以细分为 3 个步骤完成：

(1) 取出 a 的值；

(2) 执行加 1 操作；

(3) 将结果存回 a。

每个步骤对应一个操作，在多线程环境下，在某两个步骤之间可能存在其他操作的介入，所以 a++ 操作不是原子操作。

比较并交换(compare and swap，CAS)操作是基本的原子操作之一，现在几乎所有的 CPU 都支持 CAS 操作。CAS 操作的算法描述如下：

```
int compare_and_swap (Mem m, int oldval, int newval) {
   int old_reg_val = m;
   if (old_reg_val == oldval)
      m = newval;
   return old_reg_val;
}
```

其中，Mem 是一种抽象表示，可以代表内存位置，也可以代表一种引用类型，oldval 为初始值，newval 为更新后的值。在方法 compare_and_swap()的执行过程中，首先取出内存单元或引用变量的值 m，然后和 oldval 值进行比较，如果相同，则将内存位置设置为新值 newval，否则仍然为 oldval。

java. util. concurrent. atomic 包中提供了 AtomicBoolean、AtomicInteger、AtomicLong、AtomicIntegerArray 和 AtomicReference 等原子类。这些原子类提供了一种无锁的、线程安全的访问方式，每个类都提供了对于相应类型变量进行原子更新的方法。

5.2　基本类型的原子操作

基本类型的原子操作类包括 AtomicInteger、AtomicBoolean、AtomicLong，并不是每种基本类型都提供了基本类型原子操作。

下面以原子整型类 AtomicInteger 为例说明基本类型原子类的方法和使用。类 AtomicInteger 的定义形式如下：

```
public class AtomicInteger extends Number implements Serializable
```

从上面的定义可以看出，AtomicInteger 从类 Number 继承，并实现了 Serializable 接口。当程序中需要以原子方式增加或减少整数值时，通常需要用到此类。

该类有两个构造方法：

```
//初始化一个 AtomicInteger 类对象，初始值为 0
• public AtomicInteger()
//用给定的初始值 initialValue 初始化 AtomicInteger 类对象
• public AtomicInteger(int initialValue)
```

该类的常用方法如表 5-1 所示，其中，addAndGet()、getAndAdd()和 getAndSet()都是比较常用的方法，方法 lazySet()中的 lazy 表示“慵懒”的意思，该方法就像一个懒汉一样，并不是马上去做 set 操作，而是以慵懒的方式稍后再执行 set 操作。

表 5-1　类 AtomicInteger 的常用方法

方　法	含　义
int addAndGet(int delta)	在原值的基础上增加 delta
boolean compareAndSet(int expect, int update)	如果当前值等于 expect 的值，则采用原子方式更新为 update 的值
int decrementAndGet()	采用原子方式在原值的基础上减 1
double doubleValue()	将 AtomicInteger 类对象的值转换为 double 类型
float floatValue()	将 AtomicInteger 类对象的值转换为 float 类型
int get()	得到当前值
int getAndAdd(int delta)	采用原子方式在当前值的基础上增加指定的值
int getAndDecrement()	采用原子方式在原值的基础上减 1
int getAndIncrement()	采用原子方式在原值的基础上增 1
int getAndSet(int newValue)	采用原子方式设定为给定的新值，并返回原来的值
int incrementAndGet()	采用原子方式在原值的基础上增 1

续表

方　法	含　义
int intValue()	以 int 类型返回值
void lazySet(int newValue)	最后设置为给定的值 newValue
long longValue()	将 AtomicInteger 类对象的值转换为 long 类型
void set(int newValue)	设置为给定的值

下面通过例子演示类 AtomicInteger 的用法。

【例 5-1】 使用两个线程对同一个整型变量进行原子更新，更新时，在每个线程中分别对整型变量执行 100 次增 1 操作。

【解题分析】

如果两个线程对同一个整型变量进行操作时不施加任何控制，则会出错，读者可以自行尝试。对整型变量进行原子更新可以使用类 AtomicInteger，增 1 操作可以使用方法 getAndIncrement()实现。

【程序代码】

```
//Counter.java
package book.ch5.IntegerAtomic;
import java.util.concurrent.atomic.AtomicInteger;
public class Counter {
    //定义原子整型变量 ia
    private AtomicInteger ia = new AtomicInteger();
    //调用方法 getAndIncrement()进行原子更新
    public void increase(){
        ia.getAndIncrement();
    }
    //读取数据时要使用原子操作方法 get()
    public int get(){
        return ia.get();
    }
}
//Worker.java
package book.ch5.IntegerAtomic;
public class Worker extends Thread {
    Counter counter;
    Worker(Counter counter) {
        this.counter = counter;
    }
    public void run() {
        for(int i=0; i<100; i++){
            counter.increase();
        }
    }
```

```
}
//Index.java
package book.ch5.IntegerAtomic;
public class Index {
    public static void main(String[] args) {
        //定义类对象 counter,是两个线程同时操作的对象
        Counter counter = new Counter();
        //定义两个线程,将同一个 counter 对象作为参数传入线程
        Thread t1 = new Worker(counter);
        Thread t2 = new Worker(counter);
        //启动两个线程
        t1.start();
        t2.start();
        //通过 join()方法让主线程等待线程 t1 和 t2 的完成
        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("counter.get() = " + counter.get());
    }
}
```

【程序分析】

当使用两个线程同时操作同一个对象时,要将同一个对象传入线程,本例将类 Counter 对象实例分别传入了线程 t1 和 t2。

【运行结果】

程序运行结果如图 5-1 所示。

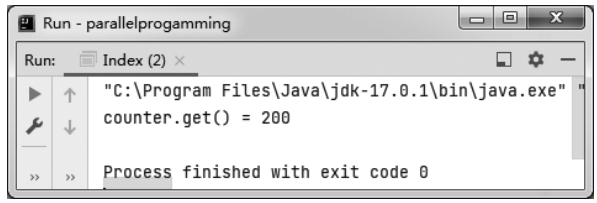

图 5-1　运行结果

【相关讨论】

从运行结果可以看出,两个线程各执行了 100 次增 1 操作,结果为 200。虽然两个线程同时对同一个变量进行了操作,但变量可以原子性的更新,以保证增加后的值为 200。

5.3 引用类型的原子操作

引用类型的原子操作使用类 AtomicReference 创建，定义的形式如下：

```
public class AtomicReference<V> extends Object implements Serializable
```

其中，V 用来指明对象引用的类型，可以根据自定义的类型指定。

类 AtomicReference 有两个构造方法，如下所示：

```
//用初始值 null 创建类 AtomicReference 的实例
• public AtomicReference()
//用初始值 initialValue 创建类 AtomicReference 的实例，其中，类型由参数 V 指定
• public AtomicReference(V initialValue)
```

第二个构造方法在生成对象实例时，会把一个 V 类型的对象实例 initialValue 放到原子引用对象中，这不会让对象本身是线程安全的，而是对该对象的获取和设置操作是线程安全的。

类 AtomicReference 的常用方法如表 5-2 所示。

表 5-2 类 AtomicReference 的常用方法

方 法	含 义
boolean compareAndSet(V expect, V update)	如果当前值等于预期值 expect，则以原子方式将该值设置为给定的更新值 update
V get()	获取当前值
V getAndSet(V newValue)	以原子方式设置为给定的新值 newValue，并返回旧值
void lazySet(V newValue)	采用慵懒的方式设置为给定值 newValue
void set(V newValue)	设置为给定值 newValue
V getAndUpdate(UnaryOperator<V> updateFunction)	以原子方式使用 updateFunction 的结果更新当前值，返回更新前的值
V updateAndGet(UnaryOperator<V> updateFunction)	以原子方式使用 updateFunction 的结果更新当前值，返回更新后的值
V getAndAccumulate(V x, BinaryOperator<V> accumulatorFunction)	以原子方式使用 accumulatorFunction 的结果更新当前值，返回更新前的值
V accumulateAndGet(V x, BinaryOperator<V> accumulatorFunction)	以原子方式使用 accumulatorFunction 的结果更新当前值，返回更新后的值

在后面四个方法的参数中，UnaryOperator 和 BinaryOperator 都是功能接口，其中，UnaryOperator 用于处理单个操作数，BinaryOperator 可以处理两个操作数，它们都会返回与操作数类型相同的结果，可以通过 Lambda 表达式的形式作为参数传递。

【例 5-2】 有 5 个选手参加一个抢答节目，谁先抢到，答题权就归谁。

【解题分析】

5 个选手可以生成 5 个线程，对引用变量进行原子更新，谁能更新成功则说明谁抢到了答题权。

【程序代码】

```
//Resource.java
package book.ch5.ar;
public class Resource {
    public Resource(){
    }
    public Resource(String name){
    }
}
//Player.java
package book.ch5.ar;
import java.util.concurrent.atomic.AtomicReference;
public class Player extends Thread{
    int id;
    Resource resource;
    AtomicReference<Resource> ar;
    public Player(int id, Resource r, AtomicReference<Resource> ar){
        this.id = id;
        resource = r;
        this.ar = ar;
    }
    public void run(){
        if(ar.compareAndSet(resource, new Resource(this.getName())))
            System.out.println(this.id + "号选手抢到了答题权");
        else
            System.out.println(this.id + "号选手没抢到");
    }
}
//Index.java
package book.ch5.ar;
import java.util.concurrent.atomic.AtomicReference;
public class Index {
    public static void main(String[] args){
        Resource resource = new Resource();
        AtomicReference<Resource> ar = new AtomicReference<Resource>(resource);
        for(int i=1; i<=5; i++){
            new Player(i, resource, ar).start();
        }
    }
}
```

【运行结果】

程序运行结果如图 5-2 所示。

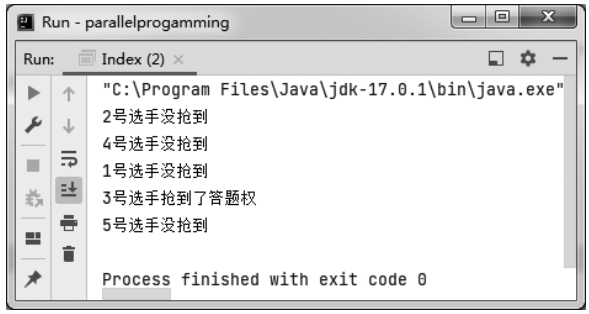

图 5-2 运行结果

【相关讨论】

从运行结果可以看出，在 3 号选手抢到答题权后，由于已经对 ar 进行了更新，故其他线程无法再次执行更新操作。

5.4 ABA 问题

原子操作对于无阻塞的操作有一定的优势，但它是不是“完美”的？显然不是的，本节将介绍使用原子操作时可能出现的 ABA 问题。

为了帮助读者理解 ABA 问题，首先通过一个生活中的例子进行说明。例如，你准备去打篮球，在去打篮球之前，你在宿舍晾了一杯水，准备打球回来后喝掉，然后你出去了，碰巧你的舍友在你不在的这段时间回到宿舍，他也刚刚打完球，口渴得厉害，直接拿起你晾好的水喝了半杯，喝完后，舍友又帮你把水倒满了，你回来后看到晾着的水以为还是你之前晾的那杯，想也没想就直接喝光。虽然水杯仍然是满的，但是水已经不是你出去之前晾的那一杯了。可以将原来满杯的水叫作 A，舍友喝完又满杯的水叫作 B，你回来后看到的水叫作 A，这就是 ABA 问题。

CAS 操作一般比较某一个对象引用的当前值和期望值，如果当前值和期望值相等，则将其替换为新值。CAS 操作的核心是看某一个值是否已经改变，也就是说，要保证在对某一变量进行操作时该变量没有被其他线程改变过，只要没有改变，就可以更新。如果期间某一线程对该值进行了改变，然后又恢复了原值，则这种情况就是 ABA 问题。

例如，有多个线程可能同时对某一个变量 x 进行更新，x 的初始值是 A，线程 1 现要将 x 的值替换为 D，它执行了 CAS 操作 CAS(A，D)。但在程序并行执行的过程中，可能会发生这样一种情况：线程 2 将 x 的值由 A 改写为 B，然后又改写为 A。等到线程 1 执行 CAS 操作时，发现当前值 A 与期望值 A 相同，则更新为 D，看似是正确的，但实际上更新后的 A 值与之前的 A 值的程序执行环境已经改变，如果继续执行，则可能对程序造成影响。

下面通过一个具体应用的例子进行说明，以栈结构为例，如图 5-3(a)所示，堆栈中有两个元素 x 和 z，其中 x 为栈顶元素，线程 2 想在栈里只有两个元素的情况下将栈顶元素 x 通过 CAS 操作更新为 t，正确的结果应该如图 5-3(b)所示。如果线程 1 和 2 同时进行操作，在线程 2 即将对该堆栈进行操作之前，线程 1 获得了执行的机会，将栈顶元素 x 出栈，然后将 y 和 x 重新推入栈内，如图 5-3(c)所示，当线程 2 执行时，发现栈顶元素仍然为 x，则更新为 t，如图 5-3(d)所示。

ABA 问题的出现在于 CAS 操作只是简单地比较了当前值和期望值，没有考虑到有可能出现的中

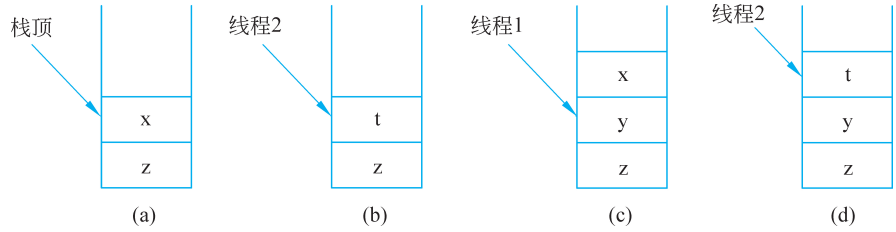

图 5-3　ABA 问题演示

间变化。在 Java 语言中，ABA 问题可以通过使用类 AtomicStampedReference 解决，这个类维护了一个类似于版本号的整数引用，每次更新后都会更新 stamp，可以避免 ABA 问题的出现。

【例 5-3】　通过原子引用操作模拟 ABA 问题。

【解题分析】

为了模拟 ABA 问题，令一个线程 a 将一个整型数的值由 0 更改为 1，然后马上将其值由 1 更改为 0，另一个线程 b 与线程 a 同时启动，观察 ABA 问题。

【程序代码】

```
//ABAThread.java
package book.ch5.aba;
import java.util.concurrent.atomic.AtomicReference;
//通过扩展类 Thread 创建类 ABAThread
public class ABAThread extends Thread {
    //原子引用类型对象 ar
    AtomicReference<Integer> ar;
    public ABAThread(AtomicReference<Integer> ar){
        this.ar = ar;
    }
    @Override
    public void run(){
        //将 ar 的值使用原子操作由 0 更新为 1,此后再将其值由 1 更新为 0
        ar.compareAndSet(0, 1);
        System.out.println("已经将值由 0 改为 1");
        ar.compareAndSet(1, 0);
        System.out.println("已经将值由 1 改为 0");
    }
}
//NormalThread.java
package book.ch5.aba;
import java.util.concurrent.atomic.AtomicReference;
//定义类 NormalThread
public class NormalThread extends Thread{
    //原子引用类型对象 ar
    AtomicReference<Integer> ar;
    //在构造方法中,对 ar 进行赋值
```

```
    public NormalThread(AtomicReference<Integer> ar){
        this.ar = ar;
    }
    @Override
    public void run(){
        //线程运行前先休眠 1ms,给 ABAThread 对象切换 0 和 1 的值留出时间
        try {
            sleep(1);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
          //验证是否更新成功
        if(ar.compareAndSet(0, 1))
            System.out.println("更新成功");
        else
            System.out.println("更新失败");
    }
}
//Index.java
package book.ch5.aba;
import java.util.concurrent.atomic.AtomicReference;
public class Index {
    public static void main(String[] args){
        //原子引用类型 ar 定义及初始化
        AtomicReference<Integer> ar = new AtomicReference<Integer>(0);
        //abaThread 线程对象定义,将原子引用 ar 作为参数传递给相应的对象
        ABAThread abaThread = new ABAThread(ar);
        //normalThread 线程对象定义
        NormalThread normalThread = new NormalThread(ar);
        //启动两个线程
        abaThread.start();
        normalThread.start();
    }
}
```

【程序分析】

该程序同时启动了两个线程 abaThread 和 normalThread,但是 normalThread 线程由于启动后休息了 0.001s,让 abaThread 线程对象有机会将 ar 的值由 0 变为 1,然后由 1 变为 0,虽然 ar 的值依然为 0,但是已经经历了一次状态的改变。当 normalThread 线程再次执行时,会检测到 0,然后更新为 1,并显示更新成功。

【运行结果】

程序运行结果如图 5-4 所示。

【相关讨论】

从运行结果可以看出,当线程 ABAThread 执行了 ABA 操作后,normalThread 仍然能够成功更新,并不知道之前在 ar 值上发生的变化。

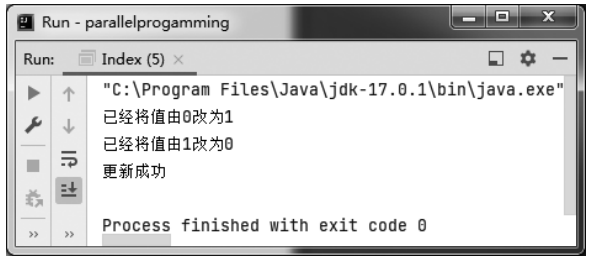

图 5-4　运行结果

5.5　扩展的原子引用类型

本节将分别对扩展的原子引用类型进行介绍，扩展的原子引用类型在引用类型的基础上加上了一些额外的标记。

5.5.1　类 AtomicMarkableReference

类 AtomicMarkableReference 是一个线程安全的类，该类封装了一个对象的引用 reference 和一个布尔型值 mark，可以原子性地对这两个值同时进行更新。该类的定义形式如下：

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

其中，V 为泛型，通常为需要标记的原子操作的类型。

类 AtomicMarkableReference 的构造方法如下：

```
AtomicMarkableReference(V initialRef, boolean initialMark)
```

该构造方法使用给定的初始值 initialRef 对引用 reference 进行赋值，使用 initialMark 对标记 mark 进行赋值。

例如，定义一个初始值为 null，未标记的 AtomicMarkableReference 对象为 amr。

```
AtomicMarkableReference<Node> amr=
                new AtomicMarkableReference<Node>(null, false);
```

其中，Node 为用户自定义类。

类 AtomicMarkableReference 的常用方法如表 5-3 所示。

表 5-3　类 AtomicMarkableReference 的常用方法

方　法	说　明
boolean attemptMark(V expectedReference, boolean newMark)	如果当前引用与 expectedReference 的值相同，则原子性地设定标记的值为 newMark 值
boolean compareAndSet (V expectedReference, V newReference, boolean expectedMark, boolean newMark)	如果当前引用与 expectedReference 相同，并且当前标记值和 expectedMark 值相同，则原子性地更新引用和标记为新值 newReference 和 newMark