

第7章 Java中的13个原子操作类

当程序更新一个变量时，如果多线程同时更新这个变量，可能得到期望之外的值，比如变量*i*=1，A线程更新*i*+1，B线程也更新*i*+1，经过两个线程操作之后可能*i*不等于3，而是等于2。因为A和B线程在更新变量*i*的时候拿到的*i*都是1，这就是线程不安全的更新操作，通常我们会使用synchronized来解决这个问题，synchronized会保证多线程不会同时更新变量*i*。

而Java从JDK 1.5开始提供了java.util.concurrent.atomic包(以下简称Atomic包)，这个包中的原子操作类提供了一种用法简单、性能高效、线程安全地更新一个变量的方式。

因为变量的类型有很多种，所以在Atomic包里一共提供了13个类，属于4种类型的原子更新方式，分别是原子更新基本类型、原子更新数组、原子更新引用和原子更新属性(字段)。

Atomic包里的类基本都是使用Unsafe实现的包装类。

7.1 原子更新基本类型类

使用原子的方式更新基本类型，Atomic包提供了以下3个类。

- AtomicBoolean: 原子更新布尔类型。

- AtomicInteger: 原子更新整型。

- AtomicLong: 原子更新长整型。

以上3个类提供的方法几乎一模一样，所以本节仅以AtomicInteger为例进行讲解，AtomicInteger的常用方法如下。

- int addAndGet(int delta): 以原子方式将输入的数值与实例中的值(AtomicInteger里的value)相加，并返回结果。

- 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的值。

- int getAndIncrement(): 以原子方式将当前值加1，注意，这里返回的是自增前的值。

- void lazySet(int new Value): 最终会设置成new Value，使用lazySet设置值后，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。关于该方法的更多信息可以参考并发编程网翻译的一篇文章《AtomicLong.lazySet是如何工作的？》，文章地址是“<http://ifeve.com/how-does-atomiclong-lazysset-work/>”。

- int getAndSet(int new Value): 以原子方式设置为new Value的值，并返回旧值。

AtomicInteger示例代码如代码清单7-1所示。

代码清单7-1 AtomicIntegerTest.java

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicIntegerTest {
    static AtomicInteger ai = new AtomicInteger(1);
    public static void main(String[] args) {
        System.out.println(ai.getAndIncrement());
        System.out.println(ai.get());
    }
}
```

输出结果如下。

```
1
2
```

那么getAndIncrement是如何实现原子操作的呢？让我们一起分析其实现原理，getAndIncrement的源码如代码清单7-2所示。

代码清单7-2 AtomicInteger.java

```
public final int getAndIncrement() {
    for (;;) {
        int current = get();
        int next = current + 1;
        if (compareAndSet(current, next))
            return current;
    }
}
public final boolean compareAndSet(int expect, int update) {
    return unsafe.compareAndSwapInt(this, valueOffset, expect, update);
}
```

源码中for循环体的第一步先取得AtomicInteger里存储的数值，第二步对AtomicInteger的当前数值进行加1操作，关键的第三步调用compareAndSet方法来进行原子更新操作，该方法先检查当前数值是否等于current，等于意味着AtomicInteger的值没有被其他线程修改过，则将AtomicInteger的当前数值更新成next的值，如果不等compareAndSet方法会返回false，程序会进入for循环重新进行compareAndSet操作。

Atomic包提供了3种基本类型的原子更新，但是Java的基本类型里还有char、float和double

等。那么问题来了，如何原子的更新其他的基本类型呢？Atomic包里的类基本都是使用Unsafe实现的，让我们一起看一下Unsafe的源码，如代码清单7-3所示。

代码清单7-3 Unsafe.java

```
/**
 * 如果当前数值是expected, 则原子的将Java变量更新成x
 * @return 如果更新成功则返回true
 */
public final native boolean compareAndSwapObject(Object o,
                                                    long offset,
                                                    Object expected,
                                                    Object x);

public final native boolean compareAndSwapInt(Object o, long offset,
                                                int expected,
                                                int x);

public final native boolean compareAndSwapLong(Object o, long offset,
                                                  long expected,
                                                  long x);
```

通过代码，我们发现Unsafe只提供了3种CAS方法：compareAndSwapObject、compareAndSwapInt和compareAndSwapLong，再看AtomicBoolean源码，发现它是先把Boolean转换成整型，再使用compareAndSwapInt进行CAS，所以原子更新char、float和double变量也可以用类似的思路来实现。

7.2 原子更新数组

通过原子的方式更新数组里的某个元素，Atomic包提供了以下4个类。

- 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里的元素。
- AtomicIntegerArray类主要是提供原子的方式更新数组里的整型，其常用方法如下。

- int addAndGet(int i, int delta): 以原子方式将输入值与数组中索引i的元素相加。

- boolean compareAndSet(int i, int expect, int update): 如果当前值等于预期值，则以原子方式将数组位置i的元素设置成update值。

以上几个类提供的方法几乎一样，所以本节仅以AtomicIntegerArray为例进行讲解，AtomicIntegerArray的使用实例代码如代码清单7-4所示。

代码清单7-4 AtomicIntegerArrayTest.java

```
public class AtomicIntegerArrayTest {  
    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]);  
    }  
}
```

以下是输出的结果。

需要注意的是, 数组value通过构造方法传递进去, 然后AtomicIntegerArray会将当前数组复制一份, 所以当AtomicIntegerArray对内部的数组元素进行修改时, 不会影响传入的数组。

7.3 原子更新引用类型

原子更新基本类型的AtomicInteger, 只能更新一个变量, 如果要原子更新多个变量, 就需要使用这个原子更新引用类型提供的类。Atomic包提供了以下3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。可以原子更新一个布尔类型的标记位和引用类型。构造方法是AtomicMarkableReference(V initialRef, boolean initialMark)。

以上几个类提供的方法几乎一样, 所以本节仅以AtomicReference为例进行讲解, AtomicReference的使用示例代码如代码清单7-5所示。

代码清单7-5 AtomicReferenceTest.java

```
public class AtomicReferenceTest {
    public static AtomicReference<user> atomicUserRef = new
        AtomicReference<user>();
    public static void main(String[] args) {
        User user = new User("conan", 15);
        atomicUserRef.set(user);
        User updateUser = new User("Shinichi", 17);
        atomicUserRef.compareAndSet(user, updateUser);
        System.out.println(atomicUserRef.get().getName());
        System.out.println(atomicUserRef.get().getOld());
    }
    static class User {
        private String name;
        private int old;
        public User(String name, int old) {
            this.name = name;
            this.old = old;
        }
        public String getName() {
            return name;
        }
        public int getOld() {
            return old;
        }
    }
}
```

```
}  
    }  
}
```

代码中首先构建一个user对象，然后把user对象设置进AtomicReferenc中，最后调用compareAndSet方法进行原子更新操作，实现原理同AtomicInteger里的compareAndSet方法。代码执行后输出结果如下。

```
Shinichi  
17
```

7.4 原子更新字段类

如果需原子地更新某个类里的某个字段时,就需要使用原子更新字段类, Atomic包提供了以下3个类进行原子字段更新。

·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。

·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。

要想原子地更新字段类需要两步。第一步,因为原子更新字段类都是抽象类,每次使用的时候必须使用静态方法newUpdater()创建一个更新器,并且需要设置想要更新的类和属性。第二步,更新类的字段(属性)必须使用public volatile修饰符。

以上3个类提供的方法几乎一样,所以本节仅以AtomicIntegerFieldUpdater为例进行讲解,AtomicIntegerFieldUpdater的示例代码如代码清单7-6所示。

代码清单7-6 AtomicIntegerFieldUpdaterTest.java

```
public class AtomicIntegerFieldUpdaterTest {
    // 创建原子更新器,并设置需要更新的对象类和对象的属性
    private static AtomicIntegerFieldUpdater<User> a = AtomicIntegerFieldUpdater.
        newUpdater(User.class, "old");
    public static void main(String[] args) {
        // 设置柯南的年龄是10岁
        User conan = new User("conan", 10);
        // 柯南长了一岁,但是仍然会输出旧的年龄
        System.out.println(a.getAndIncrement(conan));
        // 输出柯南现在的年龄
        System.out.println(a.get(conan));
    }
    public 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 int getOld() {  
            return old;  
        }  
    }  
}
```

代码执行后输出如下。

```
10  
11
```

7.5 本章小结

本章介绍了JDK中并发包里的13个原子操作类以及原子操作类的实现原理, 读者需要熟悉这些类和使用场景, 在适当的场合下使用它。

第8章 Java中的并发工具类

在JDK的并发包里提供了几个非常有用的并发工具类。CountDownLatch、CyclicBarrier和

Semaphore工具类提供了一种并发流程控制的手段，Exchanger工具类则提供了在线程间交换数

据的一种手段。本章会配合一些应用场景来介绍如何使用这些工具类。

8.1 等待多线程完成的CountDownLatch

CountDownLatch允许一个或多个线程等待其他线程完成操作。

假如有这样一个需求:我们需要解析一个Excel里多个sheet的数据,此时可以考虑使用多线程,每个线程解析一个sheet里的数据,等到所有的sheet都解析完之后,程序需要提示解析完成。在这个需求中,要实现主线程等待所有线程完成sheet的解析操作,最简单的做法是使用join()方法,如代码清单8-1所示。

代码清单8-1 JoinCountDownLatchTest.java

```
public class JoinCountDownLatchTest {
    public static void main(String[] args) throws InterruptedException {
        Thread parser1 = new Thread(new Runnable() {
            @Override
            public void run() {
                // ...
            }
        });
        Thread parser2 = new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("parser2 finish");
                // ...
            }
        });
        parser1.start();
        parser2.start();
        parser1.join();
        parser2.join();
        System.out.println("all parser finish");
    }
}
```

join用于让当前执行线程等待join线程执行结束。其实现原理是不停检查join线程是否存活,如果join线程存活则让当前线程永远等待。其中,wait(0)表示永远等待下去,代码片段如下。

```
while (isAlive()) {
    wait(0);
}
```

直到join线程中止后，线程的this.notifyAll()方法会被调用，调用notifyAll()方法是在JVM里实现的，所以在JDK里看不到，大家可以查看JVM源码。

在JDK 1.5之后的并发包中提供的CountDownLatch也可以实现join的功能，并且比join的功能更多，如代码清单8-2所示。


代码清单8-2 CountDownLatchTest.java

```
public class CountDownLatchTest {
    static CountDownLatch c = new CountDownLatch(2);
    public static void main(String[] args) throws InterruptedException {
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(1);
                c.countDown();
                System.out.println(2);
                c.countDown();
            }
        }).start();
        c.await();
        System.out.println("3");
    }
}
```

CountDownLatch的构造函数接收一个int类型的参数作为计数器，如果你想等待N个点完成，这里就传入N。

当我们调用CountDownLatch的countDown方法时，N就会减1，CountDownLatch的await方法会阻塞当前线程，直到N变成零。由于countDown方法可以用在任何地方，所以这里说的N个点，可以是N个线程，也可以是1个线程里的N个执行步骤。用在多个线程时，只需要把这个CountDownLatch的引用传递到线程里即可。

如果有某个解析sheet的线程处理得比较慢，我们不可能让主线程一直等待，所以可以使用另外一个带指定时间的await方法——await(long time, TimeUnit unit)，这个方法等待特定时间后，就会不再阻塞当前线程。join也有类似的方法。

 **注意** 计数器必须大于等于0, 只是等于0时候, 计数器就是零, 调用await方法时不会阻塞当前线程。CountDownLatch不可能重新初始化或者修改CountDownLatch对象的内部计数器的值。一个线程调用countDown方法happen-before, 另外一个线程调用await方法。

8.2 同步屏障CyclicBarrier

CyclicBarrier的字面意思是可循环使用(Cyclic)的屏障(Barrier)。它要做的事情是, 让一组线程到达一个屏障(也可以叫同步点)时被阻塞, 直到最后一个线程到达屏障时, 屏障才会开门, 所有被屏障拦截的线程才会继续运行。

8.2.1 CyclicBarrier简介

CyclicBarrier默认的构造方法是CyclicBarrier(int parties)，其参数表示屏障拦截的线程数量，每个线程调用await方法告诉CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。示例代码如代码清单8-3所示。

代码清单8-3 CyclicBarrierTest.java

```
public class CyclicBarrierTest {
    staticCyclicBarrier c = new CyclicBarrier(2);
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    c.await();
                } catch (Exception e) {
                }
                System.out.println(1);
            }
        }).start();
        try {
            c.await();
        } catch (Exception e) {
        }
        System.out.println(2);
    }
}
```

因为主线程和子线程的调度是由CPU决定的，两个线程都有可能先执行，所以会产生两种输出，第一种可能输出如下。

```
1
2
```

第二种可能输出如下。

```
2
1
```

如果把new CyclicBarrier(2)修改成new CyclicBarrier(3), 则主线程和子线程会永远等待, 因为没有第三个线程执行await方法, 即没有第三个线程到达屏障, 所以之前到达屏障的两个线程都不会继续执行。

CyclicBarrier还提供一个更高级的构造函数CyclicBarrier(int parties, Runnable barrier-Action), 用于在线程到达屏障时, 优先执行barrierAction, 方便处理更复杂的业务场景, 如代码清单8-4所示。

代码清单8-4 CyclicBarrierTest2.java

```
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierTest2 {
    static CyclicBarrier c = new CyclicBarrier(2, new A());
    public static void main(String[] args) {
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    c.await();
                } catch (Exception e) {
                }
                System.out.println(1);
            }
        }).start();
        try {
            c.await();
        } catch (Exception e) {
        }
        System.out.println(2);
    }
    static class A implements Runnable {
        @Override
        public void run() {
            System.out.println(3);
        }
    }
}
```

因为CyclicBarrier设置了拦截线程的数量是2, 所以必须等代码中的第一个线程和线程A都执行完之后, 才会继续执行主线程, 然后输出2, 所以代码执行后的输出如下。

8.2.2 CyclicBarrier的应用场景

CyclicBarrier可以用于多线程计算数据，最后合并计算结果的场景。例如，用一个Excel保存了用户所有银行流水，每个Sheet保存一个账户近一年的每笔银行流水，现在需要统计用户的日均银行流水，先用多线程处理每个sheet里的银行流水，都执行完之后，得到每个sheet的日均银行流水，最后，再用CyclicBarrier用这些线程的计算结果，计算出整个Excel的日均银行流水，如代码清单8-5所示。

代码清单8-5 BankWaterService.java

```
import java.util.Map.Entry;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;
/**
 * 银行流水处理服务类
 *
 * @author ftf
 *
 */
public class BankWaterService implements Runnable {
    /**
     * 创建4个屏障，处理完之后执行当前类的run方法
     */
    private CyclicBarrier c = new CyclicBarrier(4, this);
    /**
     * 假设只有4个sheet，所以只启动4个线程
     */
    private Executor executor = Executors.newFixedThreadPool(4);
    /**
     * 保存每个sheet计算出的银流结果
     */
    private ConcurrentHashMap<String, Integer> sheetBankWaterCount = new
        ConcurrentHashMap<String, Integer>();
    private void count() {
        for (inti = 0; i < 4; i++) {
            executor.execute(new Runnable() {
                @Override
                public void run() {
                    // 计算当前sheet的银流数据，计算代码省略
                    sheetBankWaterCount
                        .put(Thread.currentThread().getName(), 1);
                    // 银流计算完成，插入一个屏障
                }
            });
        }
    }
}
```

```

        c.await();
    } catch (InterruptedException |
            BrokenBarrierException e) {
        e.printStackTrace();
    }
}

});
}

}

@Override
public void run() {
    int result = 0;
    // 汇总每个sheet计算出的结果
    for (Entry<String, Integer> sheet : sheetBankWaterCount.entrySet()) {
        result += sheet.getValue();
    }
    // 将结果输出
    sheetBankWaterCount.put("result", result);
    System.out.println(result);
}

public static void main(String[] args) {
    BankWaterService bankWaterCount = new BankWaterService();
    bankWaterCount.count();
}
}

```

使用线程池创建4个线程，分别计算每个sheet里的数据，每个sheet计算结果是1，再由BankWaterService线程汇总4个sheet计算出的结果，输出结果如下。

8.2.3 CyclicBarrier和CountDownLatch的区别

CountDownLatch的计数器只能使用一次，而CyclicBarrier的计数器可以使用reset()方法重置。所以CyclicBarrier能处理更为复杂的业务场景。例如，如果计算发生错误，可以重置计数器，并让线程重新执行一次。

CyclicBarrier还提供其他有用的方法，比如getNumberWaiting方法可以获得CyclicBarrier阻塞的线程数量。isBroken()方法用来了解阻塞的线程是否被中断。代码清单8-5执行完之后会返回true，其中isBroken的使用代码如代码清单8-6所示。

代码清单8-6 CyclicBarrierTest3.java

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
public class CyclicBarrierTest3 {
    static CyclicBarrier c = new CyclicBarrier(2);
    public static void main(String[] args) throws InterruptedException,
        BrokenBarrierException {
        Thread thread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    c.await();
                } catch (Exception e) {
                }
            }
        });
        thread.start();
        thread.interrupt();
        try {
            c.await();
        } catch (Exception e) {
            System.out.println(c.isBroken());
        }
    }
}
```

输出如下所示。

```
true
```


8.3 控制并发线程数的Semaphore

Semaphore(信号量)是用来控制同时访问特定资源的线程数量, 它通过协调各个线程, 以保证合理的使用公共资源。

多年以来, 我都觉得从字面上很难理解Semaphore所表达的含义, 只能把它比作是控制流量的红绿灯。比如××马路要限制流量, 只允许同时有一百辆车在这条路上行使, 其他的都必须在路口等待, 所以前一百辆车会看到绿灯, 可以开进这条马路, 后面的车会看到红灯, 不能驶入××马路, 但是如果前一百辆中有5辆车已经离开了××马路, 那么后面就允许有5辆车驶入马路, 这个例子里说的车就是线程, 驶入马路就表示线程在执行, 离开马路就表示线程执行完成, 看见红灯就表示线程被阻塞, 不能执行。

1.应用场景

Semaphore可以用于做流量控制, 特别是公用资源有限的应用场景, 比如数据库连接。假如有一个需求, 要读取几万个文件的数据, 因为都是IO密集型任务, 我们可以启动几十个线程并发地读取, 但是如果读到内存后, 还需要存储到数据库中, 而数据库的连接数只有10个, 这时我们必须控制只有10个线程同时获取数据库连接保存数据, 否则会报错无法获取数据库连接。这个时候, 就可以使用Semaphore来做流量控制, 如代码清单8-7所示。

代码清单8-7 SemaphoreTest.java

```
public class SemaphoreTest {
    private static final int THREAD_COUNT = 30;
    private static ExecutorService threadPool = Executors
        .newFixedThreadPool(THREAD_COUNT);
    private static Semaphore s = new Semaphore(10);
    public static void main(String[] args) {
        for (inti = 0; i< THREAD_COUNT; i++) {
            threadPool.execute(new Runnable() {
                @Override
                public void run() {
                    try {
                        s.acquire();
```



```
        System.out.println("save data");
        s.release();
    } catch (InterruptedException e) {
    }
    }
    });
}
threadPool.shutdown();
}
```

在代码中，虽然有30个线程在执行，但是只允许10个并发执行。Semaphore的构造方法Semaphore(int permits)接受一个整型的数字，表示可用的许可证数量。Semaphore(10)表示允许10个线程获取许可证，也就是最大并发数是10。Semaphore的用法也很简单，首先线程使用Semaphore的acquire()方法获取一个许可证，使用完之后调用release()方法归还许可证。还可以用tryAcquire()方法尝试获取许可证。

2.其他方法

Semaphore还提供一些其他方法，具体如下。

- intavailablePermits():返回此信号量中当前可用的许可证数。
- intgetQueueLength():返回正在等待获取许可证的线程数。
- booleanhasQueuedThreads():是否有线程正在等待获取许可证。
- void reducePermits(int reduction):减少reduction个许可证，是个protected方法。
- Collection getQueuedThreads():返回所有等待获取许可证的线程集合，是个protected方法。

8.4 线程间交换数据的Exchanger

Exchanger(交换者)是一个用于线程间协作的工具类。Exchanger用于进行线程间的数据交换。它提供一个同步点, 在这个同步点, 两个线程可以交换彼此的数据。这两个线程通过exchange方法交换数据, 如果第一个线程先执行exchange()方法, 它会一直等待第二个线程也执行exchange方法, 当两个线程都到达同步点时, 这两个线程就可以交换数据, 将本线程生产出来的数据传递给对方。

下面来看一下Exchanger的应用场景。

Exchanger可以用于遗传算法, 遗传算法里需要选出两个人作为交配对象, 这时候会交换两人的数据, 并使用交叉规则得出2个交配结果。Exchanger也可以用于校对工作, 比如我们需要将纸制银行流水通过人工的方式录入成电子银行流水, 为了避免错误, 采用AB岗两人进行录入, 录入到Excel之后, 系统需要加载这两个Excel, 并对两个Excel数据进行校对, 看看是否录入一致, 代码如代码清单8-8所示。

代码清单8-8 ExchangerTest.java

```
public class ExchangerTest {
    private static final Exchanger<String>exgr = new Exchanger<String>();
    private static ExecutorService threadPool = Executors.newFixedThreadPool(2);
    public static void main(String[] args) {
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String A = "银行流水A"; // A录入银行流水数据
                    exgr.exchange(A);
                } catch (InterruptedException e) {}
            }
        });
        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String B = "银行流水B"; // B录入银行流水数据
                    String A = exgr.exchange("B");
                    System.out.println("A和B数据是否一致:" + A.equals(B) + ", A录入的是:"
```

```
        + A + ", B录入是:" + B);
    } catch (InterruptedException e) {
    }
}
});
threadPool.shutdown();
}
}
```

如果两个线程有一个没有执行exchange()方法, 则会一直等待, 如果担心有特殊情况发生, 避免一直等待, 可以使用exchange(V x, long timeout, TimeUnit unit)设置最大等待时长。

8.5 本章小结

本章配合一些应用场景介绍JDK中提供的几个并发工具类，大家记住这个工具类的用途，一旦有对应的业务场景，不妨试试这些工具类。