# Lock - 锁

## 1.概述

在JDK1.5中，提供了Lock，相对synchronized，更加的灵活和精细。

实现类：ReentrantLock - 重入锁。锁在释放之后允许被其他线程抢占或者被一个线程重复抢占。

非重入锁：即线程在强到这个锁之后只能执行一次代码，而且这个锁是一次性的使用完之后其他线程也不能再次使用。

### 1.1.ReentrantLock

测试Lock锁单纯使用ReentrantLock对象跟直接使用synchronized的效果是一样的，只不过前者需要新建出来后者是一个关键字。

ReentrantLock是排他锁，排他锁在同一时刻仅有一个线程可以进行访问，实际上独占锁是一种相对比较保守的锁策略，在这种情况下任何“读/读”、“读/写”、“写/写”操作都不能同时发生，这在一定程度上降低了吞吐量。

package cn.tedu.lock;

import java.util.concurrent.locks.Lock;  
import java.util.concurrent.locks.ReentrantLock;  
import java.util.concurrent.locks.ReentrantReadWriteLock;  
  
public class LockDemo {  
  
 static int *i* = 0;  
  
 public static void main(String[] args) throws InterruptedException {  
  
 //加锁  
 Lock lock = new ReentrantLock();  
  
 new Thread(new Add(lock)).start();  
 new Thread(new Add(lock)).start();  
  
 /\*  
 main所在的类是一个线程类 - 主线程  
 启动2个Add线程，在Add线程启动和准备过程中，  
 主线程会抢占资源继续执行的  
 让主线程阻塞  
 \*/  
 Thread.*sleep*(3000);  
 System.*out*.println(*i*);  
 }  
  
}  
  
class Add implements Runnable {  
  
 private Lock lock;  
  
 public Add(Lock lock) {  
  
 this.lock = lock;  
 }  
  
 @Override  
 public void run() {  
 //加锁  
 lock.lock();  
 for (int i = 0; i < 100000; i++) {  
  
 LockDemo.*i*++;  
 }  
 //解锁  
 lock.unlock();  
  
 }  
}

### 1.2.ReentrantReadWriteLock - 读写锁

分为读锁和写锁 。在使用的时候需要先创建ReentrantReadWriteLock，通过这个对象获取读锁或者写锁，之后再加锁解锁或者解锁。

ReentrantReadWriteLock 读写锁（乐观锁，无罪假设）

读操作之间不存在数据竞争问题，如果”读/读”操作能够以共享锁的方式进行，那会进一步提升性能。因此引入了ReentrantReadWriteLock，顾名思义，ReentrantReadWriteLock 是Reentrant （可重入）Read（读）Write（写）Lock（锁），我们下面称它为读写锁。

读写锁内部又分为读锁和写锁，读锁可以在没有写锁的时候被多个线程同时持有，写锁是独占的。读锁和写锁分离从而提升程序性能，读写锁主要应用于读多写少的场景。

**package** day15.xiancheng;

**import** java.util.Arrays;

**import** java.util.concurrent.locks.ReentrantReadWriteLock;

//测试线程共享冲突，ReentrantReadWriteLock语句

//解决逻辑，让两个线程抢锁，抢到锁的执行完程序再把锁让出，

**public** **class** Test5\_ReentrantReadWriteLock {

**static** **char**[] *a*= {'\*','\*','\*','\*','\*'};

**static** **char** *c*='-';

**static** ReentrantReadWriteLock *lock* =

**new** ReentrantReadWriteLock();

**public** **static** **void** main(String[] args) {

//创建Thread对象使用匿名子类，t1

Thread t1 = **new** Thread() {

@Override

**public** **void** run() {

/\*

\* 死循环不停地修改数组数据，

\* 步调一致关键字不能写在死循环，要在修改完数组的值之后将锁释放

\*/

**while**(**true**) {

//步调一致关键字，括号内填入数组变量

//synchronized(a) {

*lock*.writeLock().lock();

**for** (**int** i = 0; i < *a*.length; i++) {

*a*[i]=*c*;

}

*lock*.writeLock().unlock();

//}

//使用三元运算，判断如果c是'-'改为'\*'否则改为'-'

*c*=(*c*=='-'?'\*':'-') ;

}

}

};

//创建Thread对象使用匿名子类，t2

Thread t2 = **new** Thread() {

@Override

**public** **void** run() {

**int** index=0;

//死循环显示数组数据

**while**(**true**) {

//synchronized(a) {

*lock*.readLock().lock();

System.***out***.println(Arrays.*toString*(*a*));

//}

*lock*.readLock().unlock();

}

}

};

t1.start();

t2.start();

}

}

### 1.3.锁的公平和非公平原则

在资源有限的情况下，线程之间实际上执行的次数并不均等，这种现象称之为非公平原则；在公平策略下，线程并不能直接抢占资源而是抢占入队顺序，此时线程之间的实际执行次数是大致相等的，把这种策略称之为公平策略。

相对而言，非公平的效率更高，在Lock中，如果不指定默认是非公平策略 synchronized是非公平的。

如果需要开启公平原则也十分的简单，只需在锁对象的构造方法里传入布尔值即可。

*/\*\**

*\* Creates an instance of {****@code*** *ReentrantLock} with the  
 \* given fairness policy.  
 \*  
 \** ***@param*** *fair {****@code*** *true} if this lock should use a fair ordering policy  
 \*/*public ReentrantLock(boolean fair) {  
 sync = fair ? new FairSync() : new NonfairSync();  
}

## 2.其他可对线程资源控制的类

### 2.1.CountDownLatch：闭锁/线程递减锁

对线程进行计数，在计数归零之前线程会陷入阻塞；直到计数归零为止，才会放开阻塞。一组线程结束之后开启另一组线程。

package cn.tedu.lock;

import java.util.concurrent.CountDownLatch;  
  
public class CountDownLatchDemo {  
  
 public static void main(String[] args) throws InterruptedException {  
  
 CountDownLatch countDownLatch = new CountDownLatch(7);  
  
 new Thread(new Teacher(countDownLatch)).start();  
 new Thread(new Teacher(countDownLatch)).start();  
 new Thread(new Student(countDownLatch)).start();  
 new Thread(new Student(countDownLatch)).start();  
 new Thread(new Student(countDownLatch)).start();  
 new Thread(new Student(countDownLatch)).start();  
 new Thread(new Student(countDownLatch)).start();  
  
 countDownLatch.await();  
  
 System.*out*.println("考试开始");  
  
 }  
  
}  
  
class Teacher implements Runnable {  
  
 private CountDownLatch countDownLatch;  
  
 public Teacher(CountDownLatch countDownLatch) {  
 this.countDownLatch = countDownLatch;  
 }  
  
 @Override  
 public void run() {  
  
 try {  
  
 Thread.*sleep*((long) Math.*random*() \* 10000);  
 System.*out*.println("考官到达现场~~~");  
 countDownLatch.countDown();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
}  
  
class Student implements Runnable {  
  
 private CountDownLatch countDownLatch;  
  
 public Student(CountDownLatch countDownLatch) {  
 this.countDownLatch = countDownLatch;  
 }  
  
 @Override  
 public void run() {  
  
 try {  
 Thread.*sleep*((long) Math.*random*() \* 10000);  
 System.*out*.println("学生到达现场~~~");  
 countDownLatch.countDown();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

### 2.2.CyclicBarrier：栅栏

对线程进行计数，在计数归零之前线程会陷入阻塞；直到计数归零为止，才会放开阻塞。这一组线程到达同一个点之后再分别继续执行。

package cn.tedu.lock;

import java.util.concurrent.CyclicBarrier;  
  
public class CyclicBarrierDemo {  
  
 public static void main(String[] args) {  
  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(7);  
  
 new Thread(new Runner(cyclicBarrier),"1号").start();  
 new Thread(new Runner(cyclicBarrier),"2号").start();  
 new Thread(new Runner(cyclicBarrier),"3号").start();  
 new Thread(new Runner(cyclicBarrier),"4号").start();  
 new Thread(new Runner(cyclicBarrier),"5号").start();  
 new Thread(new Runner(cyclicBarrier),"6号").start();  
 new Thread(new Runner(cyclicBarrier),"7号").start();  
  
 }  
  
}  
  
class Runner implements Runnable {  
  
 private CyclicBarrier cyclicBarrier;  
  
 public Runner(CyclicBarrier cyclicBarrier) {  
 this.cyclicBarrier = cyclicBarrier;  
 }  
  
 @Override  
 public void run() {  
  
 try {  
  
 Thread.*sleep*((long) (Math.*random*() \* 10000));  
 String name = Thread.*currentThread*().getName();  
  
 System.*out*.println(name + "运动员到了起跑线~~~");  
  
 cyclicBarrier.await();  
 /\*  
 这个方法在阻塞的同时会减少一个计数  
 当计数归零的时候，会放开阻塞  
 \*/  
 System.*out*.println(name + "运动员跑了出去~~~");  
  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 }  
}

### 2.3.Exchanger：交换机

用于交换两个线程之间的信息。

package cn.tedu.lock;

import java.util.concurrent.Exchanger;  
  
public class ExchangerDemo {  
  
 public static void main(String[] args) {  
  
 Exchanger<String> exchanger = new Exchanger<>();  
  
 new Thread(new Producer(exchanger)).start();  
 new Thread(new Consumer(exchanger)).start();  
  
 }  
  
}  
  
class Producer implements Runnable {  
  
 private Exchanger<String> exchanger;  
  
 public Producer(Exchanger<String> exchanger) {  
 this.exchanger = exchanger;  
 }  
  
 @Override  
 public void run() {  
  
 String info = "商品";  
  
 try {  
 String msg = exchanger.exchange(info);  
 System.*out*.println("生产者收到消费者换来的：" + msg);  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
  
}  
class Consumer implements Runnable {  
  
 private Exchanger<String> exchanger;  
  
 public Consumer(Exchanger<String> exchanger) {  
 this.exchanger = exchanger;  
 }  
  
 @Override  
 public void run() {  
  
 String info = "钱";  
  
 try {  
 String msg = exchanger.exchange(info);  
 System.*out*.println("消费者收到生产者换来的：" + msg);  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
  
}

### 2.4.Semaphore：信号量

线程在获取信号之后执行代码，而在信号被全部占用之后后来的线程需要阻塞，直到前面的线程释放信号，阻塞的线程才能获取信号执行逻辑。实际过程中，信号量的作用是用于限流。

package cn.tedu.lock;

import java.util.concurrent.Semaphore;  
  
public class SemaphoreDemo {  
  
 public static void main(String[] args) {  
  
 Semaphore semaphore = new Semaphore(5);  
  
 for (int i = 0; i < 7; i++) {  
  
 new Thread(new Eater(semaphore)).start();  
 }  
  
 }  
  
}  
  
class Eater implements Runnable {  
  
 private Semaphore semaphore;  
  
 public Eater(Semaphore semaphore) {  
 this.semaphore = semaphore;  
 }  
  
 @Override  
 public void run() {  
  
 try {  
  
 semaphore.acquire();  
 System.*out*.println("来了一波客人，占用了一张桌子~~~");  
  
 Thread.*sleep*((long) (Math.*random*() \* 10000));  
 System.*out*.println("客人买单离开，空出一张桌子~~~");  
 // 释放1个信号，被阻塞的线程就可以获取信号执行代码  
 semaphore.release();  
  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

# Java原子性操作

原子性操作实际上是保证属性的原子性，底层是基于CAS+volatile来实现的 。

## 1. volatile的可见性

volatile是Java中的关键字之一，是Java中提供的用于保证线程间通信的轻量级机制保证线程的可见性 - 一个线程对堆内存中的数据进行改变，其他线程能够立即察觉到这个改变，此时就说是线程之间是可见的。

计算机生成了可选文字:
new Thread(new Runnable() 
public void run() { 
while (d ． i 
= 19 ） 
new Thread(new Runnable() 
public void run() { 
59 ； 
A 
B 
Stack 
le 
while(i 
Stack 
不 可 见 
Heap 
d 
5B 

volatile不保证线程的原子性 - 线程的执行过程不可拆分，换言之，就是线程在执行过程中不被打断不被抢占，这就是原子性(加锁实际上就是保证原子性)。不保证原子性也就意味着会产生线程安全问题。

### 1.2.volatile的可见性代码测试

package cn.tedu.atomic;

public class VolatileDemo {  
  
 public static void main(String[] args) {  
  
 Data d = new Data();  
 d.i = 10;  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 System.*out*.println("A线程启动");  
 while (d.i == 10) ;  
 System.*out*.println("A线程结束");  
 }  
 }).start();  
  
 new Thread(new Runnable() {  
 @Override  
 public void run() {  
 try {  
 // 保证A线程充分启动  
 Thread.*sleep*(2000);  
 System.*out*.println("B线程启动");  
 d.i = 50;  
 System.*out*.println("B线程结束");  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }).start();  
  
 }  
  
}  
class Data {  
 volatile int i;  
}

## 2. Java指令重排

### 2.1.指令重排的影响

指令重排是指指令没有按照预定顺序调用执行而是在底层产生了所谓的"优化"导致调用顺序发生了改变。指令重排不能违背happen-before(先发生)原则(指令重排不能改变依赖顺序)。在多线程中，产生指令重排就可能导致结果产生二相性。

计算机生成了可选文字:
在 这 几 步 过 程 中 ， 会 有 不 到 万 分 之 一 的 概 率 产 生 指 令 重 排 
CPU 
编 译 
code 
Class 
转 化 操 作 系 统 能 解 忻 
理 解 的 语 言 
调 度 
指 令 
class 文 件 本 质 上 是 丿 VM 
根 据 固 定 规 则 编 译 出 来 
的 一 套 通 用 的 解 释 文 件 

#### 2.1.1.案例1

不影响

1: double pi = 3.14;

2: int r = 5;

3: double area = pi \* r \* r;

正常顺序：1 2 3

指令重排：2 1 3

最终计算结果不受影响

#### 2.1.1.案例2

double -> a\* 2^b

int -> 不需要转化为指数

产生影响

1: int i = 5;

2: int j = i \* i;

3: i = i + 3;

正常顺序：1 2 3

指令重排：1 3 2

最终计算结果会出现错误。

### 2.2.产生指令重排的原因

JVM本身内存分配有关系 - JVM在分配内存的时候一般是以4个字节为单位进行计算分配。

浮点型在存储和计算的时候要依靠不同的精度指数集来进行计算，导致浮点型在转化为指数来存储的时候会比较麻烦。

## 3.volatile关键字总结

volatile是Java中提供的用于进行线程间通信的轻量级机制。

保证可见性 - 一个线程对主内存中的数据进行了改变，其他线程能够立即感知到这个改变

不保证原子性 - 线程的执行过程不可拆分，即线程在执行的时候不会被抢占

禁止指令重排 - 指令重排是指指令没有按照预定顺序调用执行而是在底层产生了所谓的"优化"导致调用顺序发生了改变。指令重排不能违背happen-before(先发生)原则(指令重排不能改变依赖顺序)。

## 4.Java提供的原子性类

在多线程中，产生指令重排就可能导致结果产生二相性，原子性类在底层采取了CAS+volatile机制来保证属性的数据安全。

### 4.1.代码测试案例

使用两个线程对同一个整数进行自增操作，使用Java提供的原子性整数类可以保证每条线程执行的时候不被打断，从何保证线程并发数据安全性。

package cn.tedu.atomic;

import java.util.concurrent.CountDownLatch;  
import java.util.concurrent.atomic.AtomicInteger;  
  
public class AtomicDemo {  
  
 //public static int i = 0;  
   
 //原子性整数类，保证在执行的时候不被打断  
 public static AtomicInteger *atomicInteger* = new AtomicInteger();  
  
 public static void main(String[] args)  
 throws InterruptedException {  
  
 CountDownLatch countDownLatch = new CountDownLatch(2);  
  
 new Thread(new Demo01(countDownLatch)).start();  
 new Thread(new Demo01(countDownLatch)).start();  
   
 //闭锁，等待两个线程执行结束后才可以继续执行主线程  
 countDownLatch.await();  
 System.*out*.println(*atomicInteger*.get());  
 }  
  
}  
  
class Demo01 implements Runnable {  
  
 CountDownLatch countDownLatch;  
  
 public Demo01(CountDownLatch countDownLatch) {  
 this.countDownLatch = countDownLatch;  
 }  
  
 @Override  
 public void run() {  
  
 for (int i = 0; i < 100000; i++) {  
  
 //AtomicDemo.i++;  
 AtomicDemo.*atomicInteger*.getAndIncrement();  
 }  
 //循环结束后闭锁-1  
 countDownLatch.countDown();  
 }  
}  
  
class Demo02 implements Runnable {  
  
 CountDownLatch countDownLatch;  
  
 public Demo02(CountDownLatch countDownLatch) {  
 this.countDownLatch = countDownLatch;  
 }  
  
 @Override  
 public void run() {  
  
 for (int i = 0; i < 100000; i++) {  
  
 //AtomicDemo.i++;  
 AtomicDemo.*atomicInteger*.getAndIncrement();  
 }  
 //循环结束后闭锁-1  
 countDownLatch.countDown();  
 }  
}