**多线程之间实现同步**

# 课程目标

理解线程安全？

synchronized用法

死锁

Java内存模型

Vlolatile

ThreadLock

# 什么是线程安全？

## 为什么有线程安全问题？

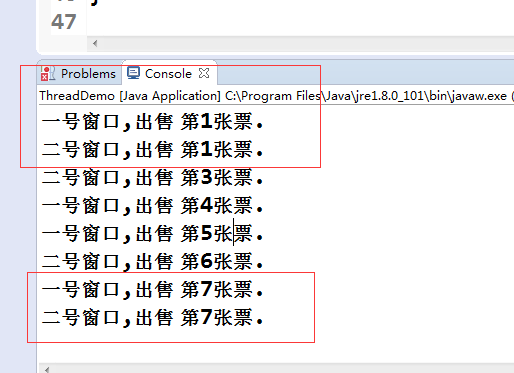
当多个线程同时共享，同一个**全局变量或静态变量**，做写的操作时，可能会发生数据冲突问题，也就是线程安全问题。但是做读操作是不会发生数据冲突问题。

**案例:需求现在有100张火车票，有两个窗口同时抢火车票，请使用多线程模拟抢票效果。**

**代码:**

|  |
| --- |
| class ThreadTrain1 implements Runnable {  private int count = 100;  private static Object *oj* = new Object();  @Override  public void run() {  while (count > 0) {  try {  Thread.*sleep*(50);  } catch (Exception e) {  // TODO: handle exception  }  sale();  }  }  public void sale() {  // 前提 多线程进行使用、多个线程只能拿到一把锁。  // 保证只能让一个线程 在执行 缺点效率降低  // synchronized (oj) {  // if (count > 0) {  System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");  count--;  // }  // }  }  }  public class ThreadDemo {  public static void main(String[] args) {  ThreadTrain1 threadTrain1 = new ThreadTrain1();  Thread t1 = new Thread(threadTrain1, "①号窗口");  Thread t2 = new Thread(threadTrain1, "②号窗口");  t1.start();  t2.start();  }  } |

**运行结果:**



**一号窗口和二号窗口同时出售火车第一张和第七张,部分火车票会重复出售。**

**结论发现，多个线程共享同一个全局成员变量时，做写的操作可能会发生数据冲突问题。**

# 线程安全解决办法:

**问:如何解决多线程之间线程安全问题?**

**答:使用多线程之间同步synchronized或使用锁(lock)。**

**问:为什么使用线程同步或使用锁能解决线程安全问题呢？**

**答:将可能会发生数据冲突问题(线程不安全问题)，只能让当前一个线程进行执行。代码执行完成后释放锁，让后才能让其他线程进行执行。这样的话就可以解决线程不安全问题。**

**问:什么是多线程之间同步？**

**答:当多个线程共享同一个资源,不会受到其他线程的干扰。**

## 同步代码块

什么是同步代码块？

答:就是将可能会发生线程安全问题的代码，给包括起来。

**synchronized(同一个数据){**

**可能会发生线程冲突问题**

**}**

就是同步代码块

synchronized(对象)//这个对象可以为任意对象

{

    需要被同步的代码

}

对象如同锁，持有锁的线程可以在同步中执行

没持有锁的线程即使获取CPU的执行权，也进不去

同步的前提：

1，必须要有两个或者两个以上的线程

2，必须是多个线程使用同一个锁

必须保证同步中只能有一个线程在运行

好处：解决了多线程的安全问题

弊端：多个线程需要判断锁，较为消耗资源、抢锁的资源。

**代码样例:**

|  |
| --- |
| **private static Object oj = new Object();**  **public void sale() {**  **// 前提 多线程进行使用、多个线程只能拿到一把锁。**  **// 保证只能让一个线程 在执行 缺点效率降低**  **synchronized (*oj*) {**  **if (count > 0) {**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");**  **count--;**  **}**  **}**  **}** |

## 同步函数

什么是同步函数？

答：在方法上修饰synchronized 称为同步函数

**代码样例**

|  |
| --- |
| **public synchronized void sale() {**  **if (trainCount > 0) {**  **try {**  **Thread.*sleep*(40);**  **} catch (Exception e) {**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");**  **trainCount--;**  **}**  **}** |

**同学们思考问题？同步函数用的是什么锁？**

答：同步函数使用this锁。

证明方式: 一个线程使用同步代码块(this明锁),另一个线程使用同步函数。如果两个线程抢票不能实现同步，那么会出现数据错误。

代码:

|  |
| --- |
| **package com.itmayiedu;**  **class ThreadTrain2 implements Runnable {**  **private int count = 100;**  **public boolean flag = true;**  **private static Object *oj* = new Object();**  **@Override**  **public void run() {**  **if (flag) {**  **while (count > 0) {**  **synchronized (this) {**  **if (count > 0) {**  **try {**  **Thread.*sleep*(50);**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");**  **count--;**  **}**  **}**  **}**  **} else {**  **while (count > 0) {**  **sale();**  **}**  **}**  **}**  **public synchronized void sale() {**  **// 前提 多线程进行使用、多个线程只能拿到一把锁。**  **// 保证只能让一个线程 在执行 缺点效率降低**  **// synchronized (oj) {**  **if (count > 0) {**  **try {**  **Thread.*sleep*(50);**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售第" + (100 - count + 1) + "票");**  **count--;**  **}**  **// }**  **}**  **}**  **public class ThreadDemo2 {**  **public static void main(String[] args) throws InterruptedException {**  **ThreadTrain2 threadTrain1 = new ThreadTrain2();**  **Thread t1 = new Thread(threadTrain1, "①号窗口");**  **Thread t2 = new Thread(threadTrain1, "②号窗口");**  **t1.start();**  **Thread.*sleep*(40);**  **threadTrain1.flag = false;**  **t2.start();**  **}**  **}** |

## 静态同步函数

答：什么是静态同步函数？

方法上加上static关键字，使用synchronized 关键字修饰 或者使用类.class文件。

静态的同步函数使用的锁是 该函数所属字节码文件对象

可以用 getClass方法获取，也可以用当前 类名.class 表示。

代码样例:

|  |
| --- |
| **synchronized (ThreadTrain.class) {**  **System.*out*.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - *trainCount* + 1) + "张票.");**  ***trainCount*--;**  **try {**  **Thread.*sleep*(100);**  **} catch (Exception e) {**  **}**  **}** |

**总结：**

**synchronized 修饰方法使用锁是当前this锁。**

**synchronized 修饰静态方法使用锁是当前类的字节码文件**

# 多线程死锁

## 什么是多线程死锁？

答:同步中嵌套同步,导致锁无法释放

代码:

|  |
| --- |
| **package** com.itmayiedu;  **class** ThreadTrain6 **implements** Runnable {  // 这是货票总票数,多个线程会同时共享资源  **private** **int** trainCount = 100;  **public** **boolean** flag = **true**;  **private** Object mutex = **new** Object();  @Override  **public** **void** run() {  **if** (flag) {  **while** (**true**) {  **synchronized** (mutex) {  // 锁(同步代码块)在什么时候释放？ 代码执行完， 自动释放锁.  // 如果flag为true 先拿到 obj锁,在拿到this 锁、 才能执行。  // 如果flag为false先拿到this,在拿到obj锁，才能执行。  // 死锁解决办法:不要在同步中嵌套同步。  sale();  }  }  } **else** {  **while** (**true**) {  sale();  }  }  }  /\*\*  \*  \* **@methodDesc**: 功能描述:(出售火车票)  \* **@author**: 余胜军  \* **@param**:  \* **@createTime**:2017年8月9日 下午9:49:11  \* **@returnType**: void  \* **@copyright**:上海每特教育科技有限公司  \*/  **public** **synchronized** **void** sale() {  **synchronized** (mutex) {  **if** (trainCount > 0) {  **try** {  Thread.*sleep*(40);  } **catch** (Exception e) {  }  System.***out***.println(Thread.*currentThread*().getName() + ",出售 第" + (100 - trainCount + 1) + "张票.");  trainCount--;  }  }  }  }  **public** **class** DeadlockThread {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  ThreadTrain6 threadTrain = **new** ThreadTrain6(); // 定义 一个实例  Thread thread1 = **new** Thread(threadTrain, "一号窗口");  Thread thread2 = **new** Thread(threadTrain, "二号窗口");  thread1.start();  Thread.*sleep*(40);  threadTrain.flag = **false**;  thread2.start();  }  } |

# 多线程有三大特性

**原子性、可见性、有序性**

## 什么是原子性

即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。

一个很经典的例子就是银行账户转账问题：   
比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。这2个操作必须要具备原子性才能保证不出现一些意外的问题。

我们操作数据也是如此，比如i = i+1；其中就包括，读取i的值，计算i，写入i。这行代码在[Java](http://lib.csdn.net/base/java" \o "Java 知识库)中是不具备原子性的，则多线程运行肯定会出问题，所以也需要我们使用同步和lock这些东西来确保这个特性了。

原子性其实就是保证数据一致、线程安全一部分，

## 什么是可见性

当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

若两个线程在不同的cpu，那么线程1改变了i的值还没刷新到主存，线程2又使用了i，那么这个i值肯定还是之前的，线程1对变量的修改线程没看到这就是可见性问题。

## 什么是有序性

程序执行的顺序按照代码的先后顺序执行。

一般来说处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。如下：

int a = 10; //语句1

int r = 2; //语句2

a = a + 3; //语句3

r = a\*a; //语句4

则因为重排序，他还可能执行顺序为 2-1-3-4，1-3-2-4  
但绝不可能 2-1-4-3，因为这打破了依赖关系。  
显然重排序对单线程运行是不会有任何问题，而多线程就不一定了，所以我们在多线程编程时就得考虑这个问题了。

# Java内存模型

共享内存模型指的就是Java内存模型(简称JMM)，**JMM决定一个线程对共享变量的写入时,能对另一个线程可见**。从抽象的角度来看，JMM定义了线程和主内存之间的抽象关系：**线程之间的共享变量存储在主内存（main memory）中，每个线程都有一个私有的本地内存（local memory），本地内存中存储了该线程以读/写共享变量的副本**。本地内存是JMM的一个抽象概念，并不真实存在。它涵盖了缓存，写缓冲区，寄存器以及其他的硬件和编译器优化。



从上图来看，线程A与线程B之间如要通信的话，必须要经历下面2个步骤：

1. 首先，线程A把本地内存A中更新过的共享变量刷新到主内存中去。

2. 然后，线程B到主内存中去读取线程A之前已更新过的共享变量。

下面通过示意图来说明这两个步骤：   


如上图所示，本地内存A和B有主内存中共享变量x的副本。假设初始时，这三个内存中的x值都为0。线程A在执行时，把更新后的x值（假设值为1）临时存放在自己的本地内存A中。当线程A和线程B需要通信时，线程A首先会把自己本地内存中修改后的x值刷新到主内存中，此时主内存中的x值变为了1。随后，线程B到主内存中去读取线程A更新后的x值，此时线程B的本地内存的x值也变为了1。

从整体来看，这两个步骤实质上是线程A在向线程B发送消息，而且这个通信过程必须要经过主内存。JMM通过控制主内存与每个线程的本地内存之间的交互，来为java程序员提供内存可见性保证。

**总结：什么是Java内存模型：java内存模型简称jmm，定义了一个线程对另一个线程可见。共享变量存放在主内存中，每个线程都有自己的本地内存，当多个线程同时访问一个数据的时候，可能本地内存没有及时刷新到主内存，所以就会发生线程安全问题。**

# Volatile

## 什么是Volatile

Volatile 关键字的作用是变量在多个线程之间可见。

代码:

|  |
| --- |
| class ThreadVolatileDemo extends Thread {  public boolean flag = true;  @Override  public void run() {  System.*out*.println("开始执行子线程....");  while (flag) {  }  System.*out*.println("线程停止");  }  public void setRuning(boolean flag) {  this.flag = flag;  }  }  public class ThreadVolatile {  public static void main(String[] args) throws InterruptedException {  ThreadVolatileDemo threadVolatileDemo = new ThreadVolatileDemo();  threadVolatileDemo.start();  Thread.*sleep*(3000);  threadVolatileDemo.setRuning(false);  System.*out*.println("flag 已经设置成false");  Thread.*sleep*(1000);  System.*out*.println(threadVolatileDemo.flag);  }  } |

运行结果:

C:\Users\ADMINI~1\AppData\Local\Temp\ksohtml\wpsB78E.tmp.jpg

已经将结果设置为fasle为什么？还一直在运行呢。

原因:线程之间是不可见的，读取的是副本，没有及时读取到主内存结果。

解决办法使用Volatile关键字将解决线程之间可见性, 强制线程每次读取该值的时候都去“主内存”中取值

## Volatile非原子性

注意: Volatile非原子性

|  |
| --- |
| **public class VolatileNoAtomic extends Thread {**  **private static volatile int *count*;**  **// private static AtomicInteger count = new AtomicInteger(0);**  **private static void addCount() {**  **for (int i = 0; i < 1000; i++) {**  ***count*++;**  **// count.incrementAndGet();**  **}**  **System.*out*.println(*count*);**  **}**  **public void run() {**  ***addCount*();**  **}**  **public static void main(String[] args) {**  **VolatileNoAtomic[] arr = new VolatileNoAtomic[100];**  **for (int i = 0; i < 10; i++) {**  **arr[i] = new VolatileNoAtomic();**  **}**  **for (int i = 0; i < 10; i++) {**  **arr[i].start();**  **}**  **}**  **}** |

运行结果:

|  |
| --- |
| C:\Users\ADMINI~1\AppData\Local\Temp\ksohtml\wpsB79F.tmp.jpg |

结果发现 数据不同步，因为Volatile不用具备原子性。

## 使用AtomicInteger原子类

AtomicInteger是一个提供原子操作的Integer类，通过线程安全的方式操作加减。

|  |
| --- |
| **public class VolatileNoAtomic extends Thread {**  **static int *count* = 0;**  **private static AtomicInteger *atomicInteger* = new AtomicInteger(0);**  **@Override**  **public void run() {**  **for (int i = 0; i < 1000; i++) {**  **//等同于i++**  ***atomicInteger*.incrementAndGet();**  **}**  **System.*out*.println(*count*);**  **}**  **public static void main(String[] args) {**  **// 初始化10个线程**  **VolatileNoAtomic[] volatileNoAtomic = new VolatileNoAtomic[10];**  **for (int i = 0; i < 10; i++) {**  **// 创建**  **volatileNoAtomic[i] = new VolatileNoAtomic();**  **}**  **for (int i = 0; i < volatileNoAtomic.length; i++) {**  **volatileNoAtomic[i].start();**  **}**  **}**  **}** |

## volatile与synchronized区别

仅靠volatile不能保证线程的安全性。（原子性）

①volatile轻量级，只能修饰变量。synchronized重量级，还可修饰方法

②volatile只能保证数据的可见性，不能用来同步，因为多个线程并发访问volatile修饰的变量不会阻塞。

synchronized不仅保证可见性，而且还保证原子性，因为，只有获得了锁的线程才能进入临界区，从而保证临界区中的所有语句都全部执行。多个线程争抢synchronized锁对象时，会出现阻塞。

**线程安全性**

线程安全性包括两个方面，①可见性。②原子性。

从上面自增的例子中可以看出：仅仅使用volatile并不能保证线程安全性。而synchronized则可实现线程的安全性。

# ThreadLoca

## 什么是ThreadLoca

ThreadLocal提高一个线程的局部变量，访问某个线程拥有自己局部变量。

当使用ThreadLocal维护变量时，ThreadLocal为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其它线程所对应的副本。

ThreadLocal的接口方法

ThreadLocal类接口很简单，只有4个方法，我们先来了解一下：

* void set(Object value)设置当前线程的线程局部变量的值。
* public Object get()该方法返回当前线程所对应的线程局部变量。
* public void remove()将当前线程局部变量的值删除，目的是为了减少内存的占用，该方法是JDK 5.0新增的方法。需要指出的是，当线程结束后，对应该线程的局部变量将自动被垃圾回收，所以显式调用该方法清除线程的局部变量并不是必须的操作，但它可以加快内存回收的速度。
* protected Object initialValue()返回该线程局部变量的初始值，该方法是一个protected的方法，显然是为了让子类覆盖而设计的。这个方法是一个延迟调用方法，在线程第1次调用get()或set(Object)时才执行，并且仅执行1次。ThreadLocal中的缺省实现直接返回一个null。

案例:创建三个线程，每个线程生成自己独立序列号。

代码:

|  |
| --- |
| **class Res {**  **// 生成序列号共享变量**  **public static Integer *count* = 0;**  **public static ThreadLocal<Integer> *threadLocal* = new ThreadLocal<Integer>() {**  **protected Integer initialValue() {**  **return 0;**  **};**  **};**  **public Integer getNum() {**  **int count = *threadLocal*.get() + 1;**  ***threadLocal*.set(count);**  **return count;**  **}**  **}**  **public class ThreadLocaDemo2 extends Thread {**  **private Res res;**  **public ThreadLocaDemo2(Res res) {**  **this.res = res;**  **}**  **@Override**  **public void run() {**  **for (int i = 0; i < 3; i++) {**  **System.*out*.println(Thread.*currentThread*().getName() + "---" + "i---" + i + "--num:" + res.getNum());**  **}**  **}**  **public static void main(String[] args) {**  **Res res = new Res();**  **ThreadLocaDemo2 threadLocaDemo1 = new ThreadLocaDemo2(res);**  **ThreadLocaDemo2 threadLocaDemo2 = new ThreadLocaDemo2(res);**  **ThreadLocaDemo2 threadLocaDemo3 = new ThreadLocaDemo2(res);**  **threadLocaDemo1.start();**  **threadLocaDemo2.start();**  **threadLocaDemo3.start();**  **}**  **}** |

## ThreadLoca实现原理

ThreadLoca通过map集合

Map.put(“当前线程”,值)；

# 练习题

## [设计4个线程，其中两个线程每次对j增加1，另外两个线程对j每次减少1。写出程序。](file:///D:\%E4%B8%8B%E8%BD%BD\Java%E9%9D%A2%E8%AF%95%E5%AE%9D%E5%85%B82017.doc#_Toc266110562)