# 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:**

|  |
| --- |
| **public** **class** ThreadTrain **implements** Runnable {  **private** **int** trainCount = 100;  @Override  **public** **void** run() {  **while** (trainCount > 0) {  **try** {  Thread.*sleep*(50);  } **catch** (Exception e) {  }  sale();  }  }  **public** **void** sale() {  **if** (trainCount > 0) {  System.***out***.println(Thread.*currentThread*().getName() + ",出售第" + (100 - trainCount + 1) + "张票");  trainCount--;  }  }  **public** **static** **void** main(String[] args) {  ThreadTrain threadTrain = **new** ThreadTrain();  Thread t1 = **new** Thread(threadTrain, "①号");  Thread t2 = **new** Thread(threadTrain, "②号");  t1.start();  t2.start();  }  } |

**运行结果:**

**一号窗口和二号窗口同时出售火车第九九张,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。**

问:什么是多线程同步

**答:当多个线程共享同一个资源,不会受到其他线程的干扰。**

## 内置的锁

Java提供了一种内置的锁机制来支持原子性

每一个Java对象都可以用作一个实现同步的锁，称为内置锁，线程进入同步代码块之前自动获取到锁，代码块执行完成正常退出或代码块中抛出异常退出时会释放掉锁

内置锁为互斥锁，即线程A获取到锁后，线程B阻塞直到线程A释放锁，线程B才能获取到同一个锁

内置锁使用synchronized关键字实现，synchronized关键字有两种用法：

1.修饰需要进行同步的方法（所有访问状态变量的方法都必须进行同步），此时充当锁的对象为调用同步方法的对象

2.同步代码块和直接使用synchronized修饰需要同步的方法是一样的，但是锁的粒度可以更细，并且充当锁的对象不一定是this，也可以是其它对象，所以使用起来更加灵活

## 同步代码块synchronized

|  |
| --- |
| 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:**

|  |
| --- |
| **public** **void** sale() {  **synchronized** (**this**) {  **if** (trainCount > 0) {  System.***out***.println(Thread.*currentThread*().getName() + ",出售第" + (100 - trainCount + 1) + "张票");  trainCount--;  }  }  } |

## 同步方法

### 什么是同步方法？

答：在方法上修饰synchronized 称为同步方法

**代码样例**

|  |
| --- |
| **public** **synchronized** **void** sale() {  **if** (trainCount > 0) {  System.***out***.println(Thread.*currentThread*().getName() + ",出售第" + (100 - trainCount + 1) + "张票");  trainCount--;  }  } |

### 同学们思考问题？同步方法使用的是什么锁？

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:

|  |
| --- |
| **class** Thread009 **implements** Runnable {  **private** **int** trainCount = 100;  **private** Object oj = **new** Object();  **public** **boolean** flag = **true**;  **public** **void** run() {  **if** (flag) {  **while** (trainCount > 0) {  **synchronized** (**this**) {  **try** {  Thread.*sleep*(10);  } **catch** (Exception e) {  // **TODO**: handle exception  }  **if** (trainCount > 0) {  System.***out***  .println(Thread.*currentThread*().getName() + "," + "出售第" + (100 - trainCount + 1) + "票");  trainCount--;  }  }  }  } **else** {  **while** (trainCount > 0) {  sale();  }  }  }  **public** **synchronized** **void** sale() {  **try** {  Thread.*sleep*(10);  } **catch** (Exception e) {  // **TODO**: handle exception  }  **if** (trainCount > 0) {  System.***out***.println(Thread.*currentThread*().getName() + "," + "出售第" + (100 - trainCount + 1) + "票");  trainCount--;  }  }  }  **public** **class** Test009 {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Thread009 threadTrain = **new** Thread009();  Thread t1 = **new** Thread(threadTrain, "窗口1");  Thread t2 = **new** Thread(threadTrain, "窗口2");  t1.start();  Thread.*sleep*(40);  threadTrain.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:

|  |
| --- |
| **public** **static** **void** sale() {  **synchronized** (ThreadTrain3.**class**) {  **if** (*trainCount* > 0) {  System.***out***.println(Thread.*currentThread*().getName() + ",出售第" + (100 - *trainCount* + 1) + "张票");  *trainCount*--;  }  }  } |

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:

|  |
| --- |
| **class** Thread009 **implements** Runnable {  **private** **int** trainCount = 100;  **private** Object oj = **new** Object();  **public** **boolean** flag = **true**;  **public** **void** run() {  **if** (flag) {  **while** (trainCount > 0) {  **synchronized** (oj) {  **try** {  Thread.*sleep*(10);  } **catch** (Exception e) {  // **TODO**: handle exception  }  sale();  }  }  } **else** {  **while** (trainCount > 0) {  sale();  }  }  }  **public** **synchronized** **void** sale() {  **synchronized** (oj) {  **try** {  Thread.*sleep*(10);  } **catch** (Exception e) {  }  **if** (trainCount > 0) {  System.***out***.println(Thread.*currentThread*().getName() + "," + "出售第" + (100 - trainCount + 1) + "票");  trainCount--;  }  }  }  }  **public** **class** Test009 {  **public** **static** **void** main(String[] args) **throws** InterruptedException {  Thread009 threadTrain = **new** Thread009();  Thread t1 = **new** Thread(threadTrain, "窗口1");  Thread t2 = **new** Thread(threadTrain, "窗口2");  t1.start();  Thread.*sleep*(40);  threadTrain.flag = **false**;  t2.start();  }  }} |

# Threadlocal

## 什么是Threadlocal

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)；

# 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

可见性也就是说一旦某个线程修改了该被volatile修饰的变量，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，可以立即获取修改之后的值。

在Java中为了加快程序的运行效率，对一些变量的操作通常是在该线程的寄存器或是CPU缓存上进行的，之后才会同步到主存中，而加了volatile修饰符的变量则是直接读写主存。

Volatile 保证了线程间共享变量的及时可见性，但不能保证原子性

|  |
| --- |
| 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特性

　　1.保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：Java内存模型）来完成。

　　2.禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl $0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。

volatile 性能：

　　volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

## [Volatile与Synchronized](https://blog.csdn.net/SEU_Calvin/article/details/52370068)区别

（1）从而我们可以看出volatile虽然具有可见性但是并不能保证原子性。

（2）性能方面，synchronized关键字是防止多个线程同时执行一段代码，就会影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized。

但是要注意volatile关键字是无法替代synchronized关键字的，因为volatile关键字无法保证操作的原子性。

# 重排序

## 数据依赖性

如果两个操作访问同一个变量，且这两个操作中有一个为写操作，此时这两个操作之间就存在数据依赖性。数据依赖分下列三种类型：

|  |  |  |
| --- | --- | --- |
| 名称 | 代码示例 | 说明 |
| 写后读 | a = 1;b = a; | 写一个变量之后，再读这个位置。 |
| 写后写 | a = 1;a = 2; | 写一个变量之后，再写这个变量。 |
| 读后写 | a = b;b = 1; | 读一个变量之后，再写这个变量。 |

上面三种情况，只要重排序两个操作的执行顺序，程序的执行结果将会被改变。

前面提到过，编译器和处理器可能会对操作做重排序。编译器和处理器在重排序时，会遵守数据依赖性，编译器和处理器不会改变存在数据依赖关系的两个操作的执行顺序。

注意，这里所说的数据依赖性仅针对单个处理器中执行的指令序列和单个线程中执行的操作，不同处理器之间和不同线程之间的数据依赖性不被编译器和处理器考虑。

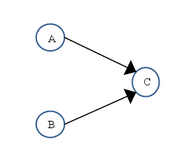
## as-if-serial语义

s-if-serial语义的意思指：不管怎么重排序（编译器和处理器为了提高并行度），（单线程）程序的执行结果不能被改变。编译器，runtime 和处理器都必须遵守as-if-serial语义。

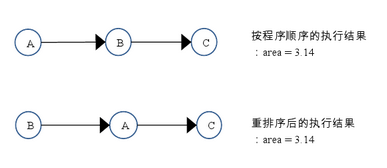
为了遵守as-if-serial语义，编译器和处理器不会对存在数据依赖关系的操作做重排序，因为这种重排序会改变执行结果。但是，如果操作之间不存在数据依赖关系，这些操作可能被编译器和处理器重排序。为了具体说明，请看下面计算圆面积的代码示例：

|  |
| --- |
| double pi = 3.14; //A  double r = 1.0; //B  double area = pi \* r \* r; //C |

上面三个操作的数据依赖关系如下图所示：



如上图所示，A和C之间存在数据依赖关系，同时B和C之间也存在数据依赖关系。因此在最终执行的指令序列中，C不能被重排序到A和B的前面（C排到A和B的前面，程序的结果将会被改变）。但A和B之间没有数据依赖关系，编译器和处理器可以重排序A和B之间的执行顺序。下图是该程序的两种执行顺序：



as-if-serial语义把单线程程序保护了起来，遵守as-if-serial语义的编译器，runtime 和处理器共同为编写单线程程序的程序员创建了一个幻觉：单线程程序是按程序的顺序来执行的。as-if-serial语义使单线程程序员无需担心重排序会干扰他们，也无需担心内存可见性问题。

## 程序顺序规则

根据happens- before的程序顺序规则，上面计算圆的面积的示例代码存在三个happens- before关系：

1. A happens- before B；

2. B happens- before C；

3. A happens- before C；

这里的第3个happens- before关系，是根据happens- before的传递性推导出来的。

这里A happens- before B，但实际执行时B却可以排在A之前执行（看上面的重排序后的执行顺序）。在第一章提到过，如果A happens- before B，JMM并不要求A一定要在B之前执行。JMM仅仅要求前一个操作（执行的结果）对后一个操作可见，且前一个操作按顺序排在第二个操作之前。这里操作A的执行结果不需要对操作B可见；而且重排序操作A和操作B后的执行结果，与操作A和操作B按happens- before顺序执行的结果一致。在这种情况下，JMM会认为这种重排序并不非法（not illegal），JMM允许这种重排序。

在计算机中，软件技术和硬件技术有一个共同的目标：在不改变程序执行结果的前提下，尽可能的开发并行度。编译器和处理器遵从这一目标，从happens- before的定义我们可以看出，JMM同样遵从这一目标。

## 重排序对多线程的影响

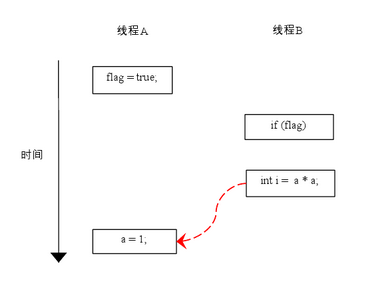
现在让我们来看看，重排序是否会改变多线程程序的执行结果。请看下面的示例代码：

|  |
| --- |
| class ReorderExample {  int a = 0;  boolean flag = false;  public void writer() {  a = 1; //1  flag = true; //2  }  Public void reader() {  if (flag) { //3  int i = a \* a; //4  ……  }  }  } |

flag变量是个标记，用来标识变量a是否已被写入。这里假设有两个线程A和B，A首先执行writer()方法，随后B线程接着执行reader()方法。线程B在执行操作4时，能否看到线程A在操作1对共享变量a的写入？

答案是：不一定能看到。

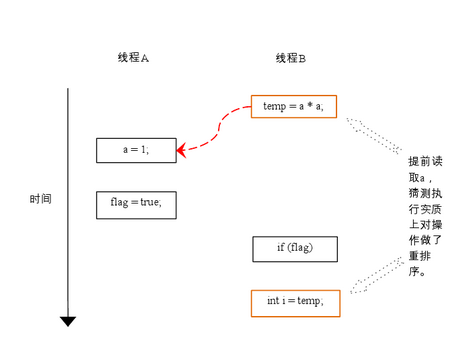
由于操作1和操作2没有数据依赖关系，编译器和处理器可以对这两个操作重排序；同样，操作3和操作4没有数据依赖关系，编译器和处理器也可以对这两个操作重排序。让我们先来看看，当操作1和操作2重排序时，可能会产生什么效果？请看下面的程序执行时序图：



如上图所示，操作1和操作2做了重排序。程序执行时，线程A首先写标记变量flag，随后线程B读这个变量。由于条件判断为真，线程B将读取变量a。此时，变量a还根本没有被线程A写入，在这里多线程程序的语义被重排序破坏了！

※注：本文统一用红色的虚箭线表示错误的读操作，用绿色的虚箭线表示正确的读操作。

下面再让我们看看，当操作3和操作4重排序时会产生什么效果（借助这个重排序，可以顺便说明控制依赖性）。下面是操作3和操作4重排序后，程序的执行时序图：



在程序中，操作3和操作4存在控制依赖关系。当代码中存在控制依赖性时，会影响指令序列执行的并行度。为此，编译器和处理器会采用猜测（Speculation）执行来克服控制相关性对并行度的影响。以处理器的猜测执行为例，执行线程B的处理器可以提前读取并计算a\*a，然后把计算结果临时保存到一个名为重排序缓冲（reorder buffer ROB）的硬件缓存中。当接下来操作3的条件判断为真时，就把该计算结果写入变量i中。

从图中我们可以看出，猜测执行实质上对操作3和4做了重排序。重排序在这里破坏了多线程程序的语义！

在单线程程序中，对存在控制依赖的操作重排序，不会改变执行结果（这也是as-if-serial语义允许对存在控制依赖的操作做重排序的原因）；但在多线程程序中，对存在控制依赖的操作重排序，可能会改变程序的执行结果。