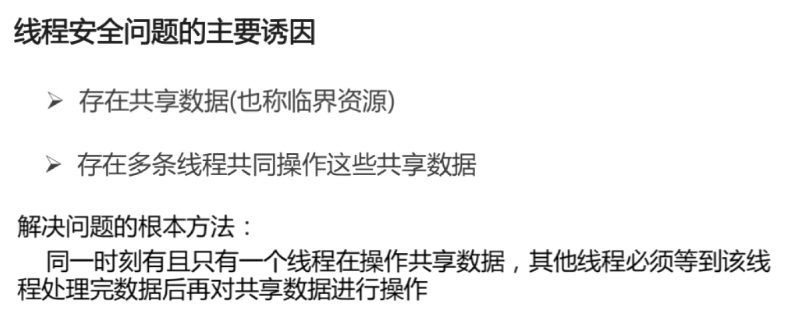
# 线程安全与同步

## 线程安全问题



## 什么情况下需要同步

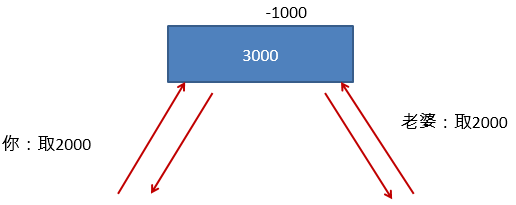
当多线程并发, 有多段代码同时执行时, 我们希望某一段代码执行的过程中CPU不要切换到其他线程工作. 这时就需要同步.

如果两段代码是同步的, 那么同一时间只能执行一段, 在一段代码没执行结束之前, 不会执行另外一段代码.

解决线程安全问题

前提：

如果我们创建的多个线程，存在着共享数据，那么就有可能出现线程的安全问题：当其中一个线程操作共享数据时，还未操作完成，另外的线程就参与进来，导致对共享数据的操作出现问题？



解决方式：

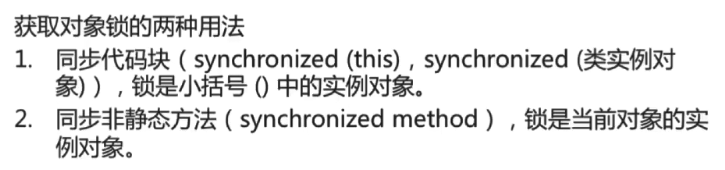
要求一个线程操作共享数据时，只有当其完成操作完成共享数据，其它线程才有机会执行共享数据。

## Synchronized

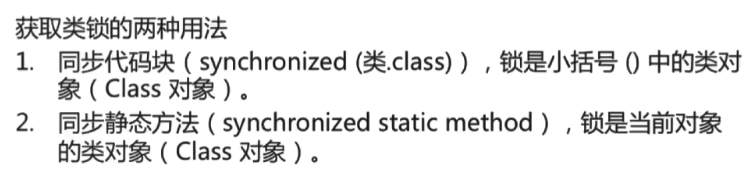
### 锁的本质



### 获取对象锁



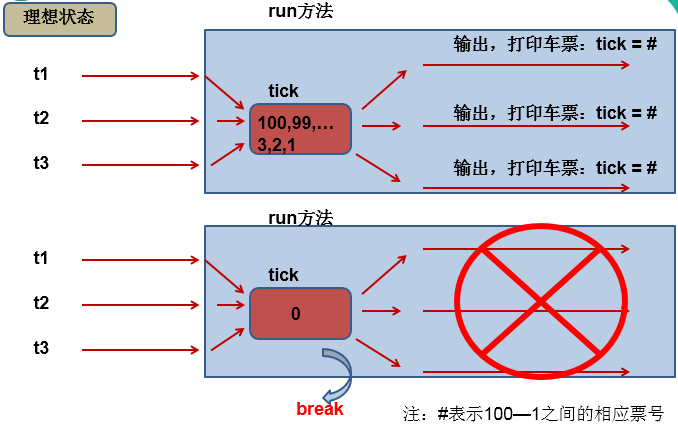
### 获取类锁

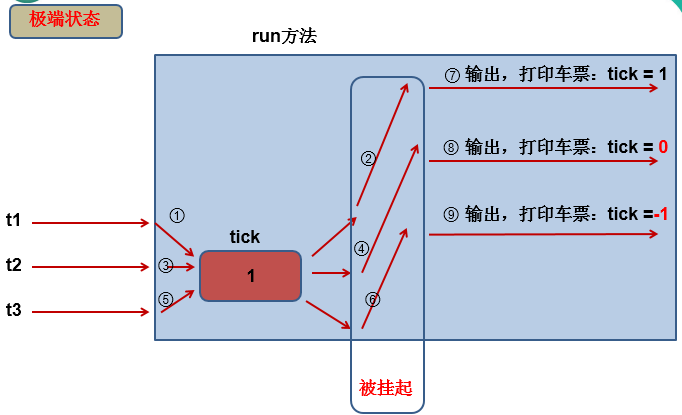


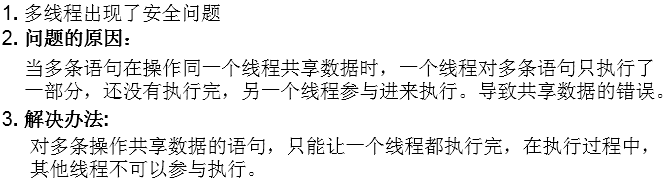
# 取票问题

## 模拟火车站售票程序

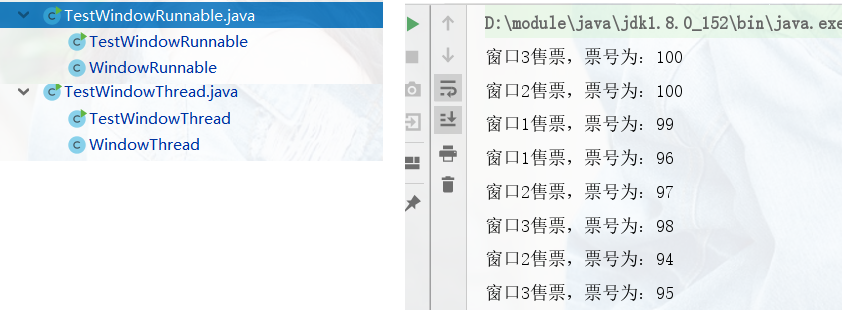
模拟火车站售票程序，开启三个窗口售票。TestWindow1.java







## 原始Runnable和Thread存在线程安全



## 方式一：Synchronized同步代码块：

### 概念

谁是共享数据？

**synchronized**(同步监视器){

//操作共享数据的代码

}

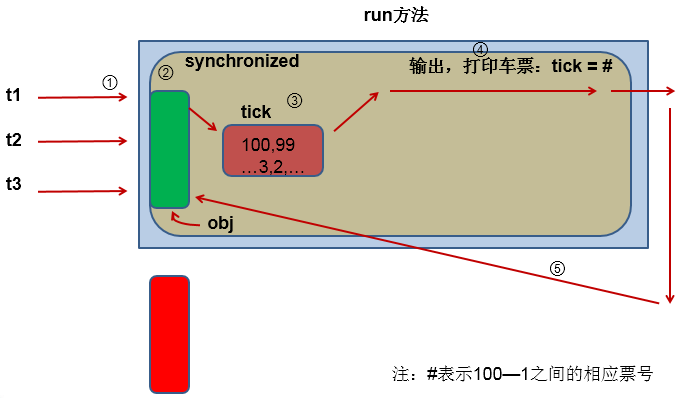
注：

1.同步监视器：俗称锁，任何一个类的对象都可以才充当锁。要想保证线程的安全，必须要求所有的线程共用同一把锁！

2.使用实现Runnable接口的方式创建多线程的话，同步代码块中的锁，可以考虑是this。如果使用继承Thread类的方式，慎用this!

对于静态方法而言，使用当前类本身充当锁

3.共享数据：多个线程需要共同操作的变量。 明确哪部分是操作共享数据的代码。



### 同步Runnable

**class** WindowRunnableSynchronizedCode **implements** Runnable {

**int** ticket = 100;// 共享

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

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

**synchronized** (**this**) {//this表示当前对象，本题中即为w

**if** (ticket > 0) {

**try** {

Thread.*currentThread*().*sleep*(10);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName()

+ "售票，票号为：" + ticket--);

}

}

}

}

}

**public** **class** TestWindowRunnableSynchronizedCode {

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

WindowRunnableSynchronizedCode w = **new** WindowRunnableSynchronizedCode();

Thread t1 = **new** Thread(w);

Thread t2 = **new** Thread(w);

Thread t3 = **new** Thread(w);

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

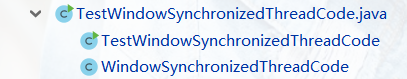
t2.start();

t3.start();

}

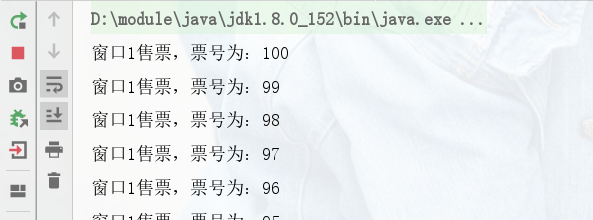
}

### 同步Thread



### 结果

票号按照顺序输出出



## 方式二：Synchronized同步方法：

### 概念

//非静态的同步方法的锁对象是神马?

//答:非静态的同步方法的锁对象是this

//静态的同步方法的锁对象是什么?

//是该类的字节码对象

将操作共享数据的方法声明为synchronized

**public** **synchronized** **void** show (String name){

….

}

//操作共享数据的代码

注：

1.对于非静态的方法而言，使用同步的话，默认锁为：this。如果使用在继承的方式实现多线程的话，慎用！

2..对于静态的方法，如果使用同步，默认的锁为：当前类本身。以单例的懒汉式为例。 Class clazz = Singleton.class

### 同步Runnable

**class** WindowRunnableSynchronizedMethod **implements** Runnable {

**int** ticket = 100;// 共享数据

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

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

show();

}

}

**public** **synchronized** **void** show() {

**if** (ticket > 0) {

**try** {

Thread.*currentThread*().*sleep*(10);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() + "售票，票号为："

+ ticket--);

}

}

}

**public** **class** TestWindowRunnableSynchronizedMethod {

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

WindowRunnableSynchronizedMethod w = **new** WindowRunnableSynchronizedMethod();

Thread t1 = **new** Thread(w);

Thread t2 = **new** Thread(w);

Thread t3 = **new** Thread(w);

t1.setName("窗口1");

t2.setName("窗口2");

t3.setName("窗口3");

t1.start();

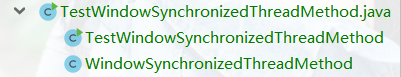
t2.start();

t3.start();

}

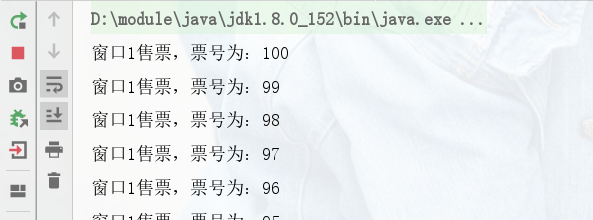
}

### 同步Thread



### 结果

票号按照顺序输出出



# 线程通信

## 线程通信

Java.lang.Object提供的这三个方法只有在synchronized方法或synchronized代码块中才能使用，否则会报java.lang.IllegalMonitorStateException异常



## wait() 方法

### 概述

在当前线程中调用方法： 对象名.wait()

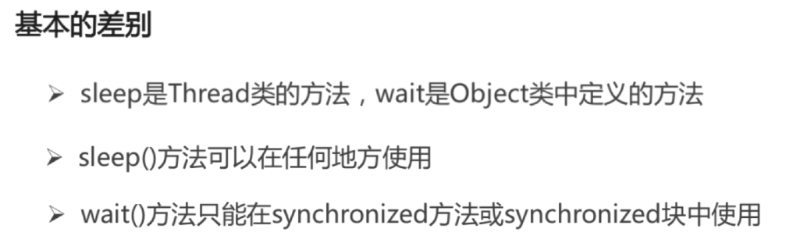
使当前线程进入等待（某对象）状态 ，直到另一线程对该对象发出 notify (或notifyAll) 为止。

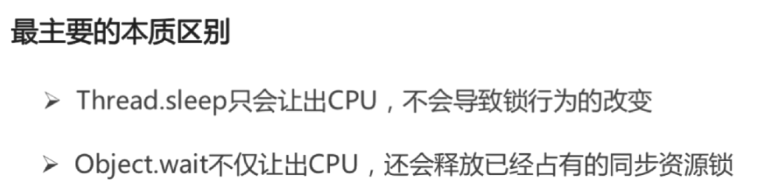
调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

调用此方法后，当前线程将释放对象监控权 ，然后进入等待

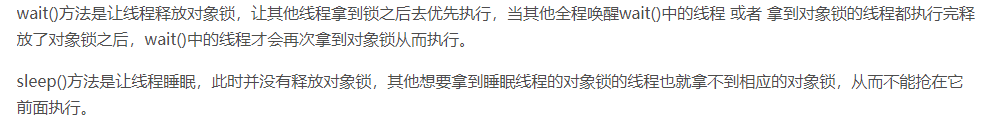
在当前线程被notify后，要重新获得监控权，然后从断点处继续代码的执行。

### Sleep 和 wait 的区别



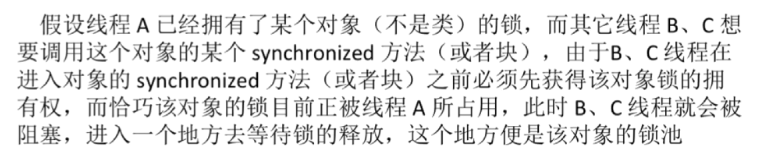


### 只有Wait对锁有影响



## 锁池和等待池

### 锁池EntryList



### 等待池WaitSet

调用进入等待池

## notify()/notifyAll()

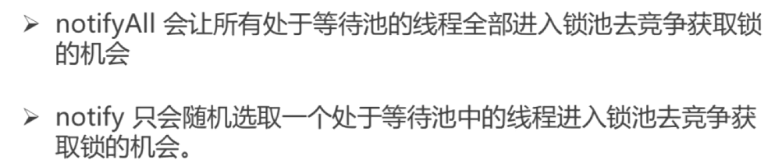
### 概述

在当前线程中调用方法： 对象名.notify()

功能：唤醒等待该对象监控权的一个线程。

调用方法的必要条件：当前线程必须具有对该对象的监控权（加锁）

### 区别



# 线程通信

## 使用两个线程打印 1-100.

**使用两个线程打印 1-100. 线程1, 线程2 交替打印**

//线程通信。如下的三个关键字使用的话，都得在同步代码块或同步方法中。

//wait():一旦一个线程执行到wait()，就释放当前的锁。

//notify()/notifyAll():唤醒wait的一个或所有的线程

//使用两个线程打印 1-100. 线程1, 线程2 交替打印

**class** PrintNum **implements** Runnable {

**int** num = 1;

Object obj = **new** Object();

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

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

**synchronized** (obj) {

obj.notify();

**if** (num <= 100) {

**try** {

Thread.*currentThread*().*sleep*(10);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

System.*out*.println(Thread.*currentThread*().getName() + ":"

+ num);

num++;

} **else** {

**break**;

}

**try** {

obj.wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

}

**public** **class** TestCommunication {

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

PrintNum p = **new** PrintNum();

Thread t1 = **new** Thread(p);

Thread t2 = **new** Thread(p);

t1.setName("甲");

t2.setName("乙");

t1.start();

t2.start();

}

}

## 经典例题：生产者/消费者问题

3. **经典例题：生产者/消费者问题**

* 生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，店员一次只能持有固定数量的产品(比如:20），如果生产者试图生产更多的产品，店员会叫生产者停一下，如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，如果店中有产品了再通知消费者来取走产品。
* 这里可能出现两个问题：
* 生产者比消费者快时，消费者会漏掉一些数据没有取到。
* 消费者比生产者快时，消费者会取相同的数据。

/\*

\* 生产者/消费者问题

\* 生产者(Productor)将产品交给店员(Clerk)，而消费者(Customer)从店员处取走产品，

\* 店员一次只能持有固定数量的产品(比如:20），如果生产者试图生产更多的产品，店员会叫生产者停一下，

\* 如果店中有空位放产品了再通知生产者继续生产；如果店中没有产品了，店员会告诉消费者等一下，

\* 如果店中有产品了再通知消费者来取走产品。

分析：

1.是否涉及到多线程的问题？是！生产者、消费者

2.是否涉及到共享数据？有！考虑线程的安全

3.此共享数据是谁？即为产品的数量

4.是否涉及到线程的通信呢？存在这生产者与消费者的通信

\*/

**class** Clerk{//店员

**int** product;

**public** **synchronized** **void** addProduct(){//生产产品

**if**(product >= 20){

**try** {

wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}**else**{

product++;

System.*out*.println(Thread.*currentThread*().getName() + ":生产了第" + product + "个产品");

notifyAll();

}

}

**public** **synchronized** **void** consumeProduct(){//消费产品

**if**(product <= 0){

**try** {

wait();

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}**else**{

System.*out*.println(Thread.*currentThread*().getName() + ":消费了第" + product + "个产品");

product--;

notifyAll();

}

}

}

**class** Producer **implements** Runnable{//生产者

Clerk clerk;

**public** Producer(Clerk clerk){

**this**.clerk = clerk;

}

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

System.*out*.println("生产者开始生产产品");

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

**try** {

Thread.*currentThread*().*sleep*(100);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

clerk.addProduct();

}

}

}

**class** Consumer **implements** Runnable{//消费者

Clerk clerk;

**public** Consumer(Clerk clerk){

**this**.clerk = clerk;

}

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

System.*out*.println("消费者消费产品");

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

**try** {

Thread.*currentThread*().*sleep*(10);

} **catch** (InterruptedException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

clerk.consumeProduct();

}

}

}

**public** **class** TestProduceConsume {

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

Clerk clerk = **new** Clerk();

Producer p1 = **new** Producer(clerk);

Consumer c1 = **new** Consumer(clerk);

Thread t1 = **new** Thread(p1);//一个生产者的线程

Thread t3 = **new** Thread(p1);

Thread t2 = **new** Thread(c1);//一个消费者的线程

t1.setName("生产者1");

t2.setName("消费者1");

t3.setName("生产者2");

t1.start();

t2.start();

t3.start();

}

}

## 模拟银行取钱的问题

**模拟银行取钱的问题**

1.定义一个Account类

1）该Account类封装了账户编号（String）和余额（double）两个属性

2）设置相应属性的getter和setter方法

3）提供无参和有两个参数的构造器

4）系统根据账号判断与用户是否匹配，需提供hashCode()和equals()方法的重写

2.提供一个取钱的线程类

1）提供了Account类的account属性和double类的取款额的属性

2）提供带线程名的构造方法

3）run()方法中提供取钱的操作

3.在主类中创建线程进行测试。考虑线程安全问题。

