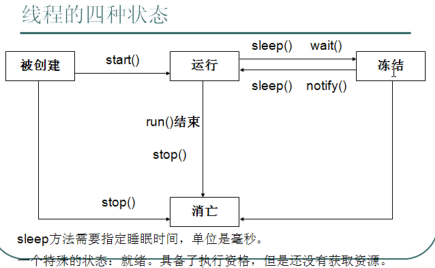
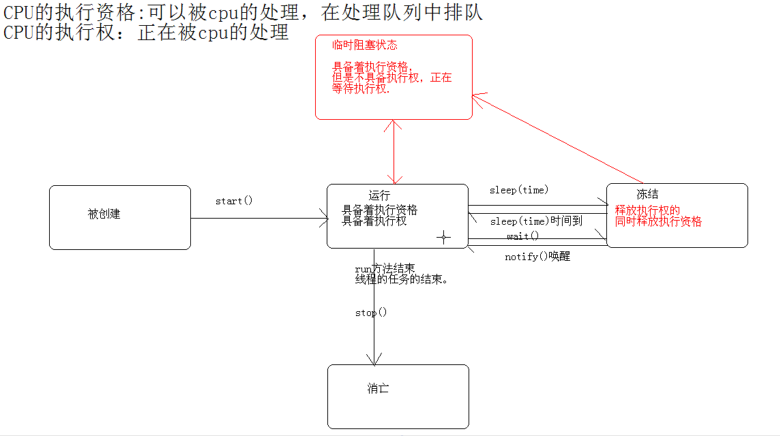
线程间的通信

1. **线程间通信（通讯）： 多个线程在处理同一个资源，但任务不同，就需要利用线程间通信。**
2. 线程的状态：





（1） 当**run方法执行完后**，线程会自动消亡，当然也可以利用**stop方法**使线程**强制消亡**(注意stop方法不安全，已经过时了，建议自己通过标志位控制线程结束。)；

（2） **sleep（time）方法**必须指定时间，时间一到自动会进入运行状态；

（3） **wait()方法**既可以指定时间，也可以不指定时间，如果不指定时间，则必须由**notify（） 方法**唤醒线程，否则一直等待。

（4） **CPU的执行资格：在CPU待处理的队列中排队；**

**CPU的执行权：正在被CPU处理**。

（5） **冻结状态**：释放CPU执行权的同时，同时释放CPU执行资格。

（6） **临时阻塞状态**：具备CPU执行资格，但不具有CPU执行权，正在等待获取执行权。

（7） 如果划分更细的话，可以把**sleep和wait**分别化为一种状态。

1. **等待/唤醒机制：**
2. **监视器： 就是所说的锁。**
3. **涉及到的方法：**
4. **wait(); 使当前线程处于冻结状态，被wait的线程会存储到线程池（等待集）中;**
5. **notify(); 唤醒该监视器对象所监视的线程池中的任意的一个线程（随机）；**
6. **notifyAll（）：唤醒该监视器对象所监视的线程池中的所有线程。**
7. **这些方法都必须定义在同步中，因为这些方法是用于操作线程状态的方法；而且必须要明确到底操作的是哪个锁上的线程,即必须明确线程由那个监视器监视。这些方法其实就是监视器对象的操作方法。**
8. **为什么操作线程的方法wait（）、notify（）、notifyAll（）定义在了Object类中？**

**答：因为这些方法就是监视器的方法，监视器就是锁，锁可以是任意的对象，如 Object obj = new Object（）；既然可以是任意对象，那么这些方法需要定义在最顶层的类中，即定义在Object类中。任意对象都可以作为同步锁。**

1. **线程停止的方法**：
   1. run方法结束，线程自动结束；
   2. 利用stop方法（**已经过时了**）。
   3. 通过设置**循环标志**来，必要时需要使用**interrupt方法**，强制激活某个线程。

（1）怎么控制线程的任务结束呢？

答：任务中设置有**循环结构**，只要控制住循环就可以**控制住任务**，即可以**控制住线程**。while（flag）{} 若flag为true，则为永久循环，若另flag为false即可结束循环。 **控制循环通常使用定义标志来完成**。

（2）如果线程处于**冻结状态**，无法读取标记时，怎样结束线程呢？

利用**interrupt（）**方法，可以将线程从**冻结状态**强制回到拥有cpu资格的运行状态，但是这种强制动作会**抛出异常InterruptedException**，需要在catch中进行**异常处理**。interrupt方法：



1. **线程间通信的示例：**

**分析： 先定义一个资源类，然后定义出多个不同的线程任务对象都操作这个资源类的同一个资源对象，显然需要把资源对象作为参数传进任务对象中，然后把这两个任务对象作为参数传进Thread类，从而创建两个线程，最后启动线程。**

**示例： 创建一个资源类，具有属性名字和年龄，并设置一个flag作为标志位，控制线程的状态。再创建两个线程，一个是Input一个是Output，完成功能：输入一个，便输出一个。**

**创建资源类：**

**public** **class** Resource {

**private** **boolean** flag ;

**private** String name ;

**private** **int** age;

**public** Resource(**boolean** flag) {

**super**();

**this**.flag = flag;

}

**public** **synchronized** **void** set(String name,**int** age) {

**if**(flag) {

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

}

}

**this**.name = name;

**this**.age = age;

System.***out***.println("输入 ： "+**this**.name + "\t"+ **this**.age);

flag = !flag;

**this**.notify();

}

**public** **synchronized** **void** get() {

**if** (!flag) {

**try** {

**this**.wait();

} **catch** (InterruptedException e) {

}

}

System.***out***.println("输出 ："+**this**.name + "\t"+**this**.age);

flag = !flag;

**this**.notify();

} }

创建两个线程Input和Output：

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

Resource resource = **new** Resource(**false**);

Thread input = **new** Thread(**new** Runnable() {

@Override

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

**for** (**int** i = 0; i < 20; i++) {

**if**(i % 2 == 0) {

resource.set("小明 : "+i, 10);

}**else** {

resource.set("小红 : "+i, 22);

}

}

}

});

Thread output = **new** Thread(**new** Runnable() {

@Override

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

**for** (**int** i = 0; i < 20; i++) {

resource .get();

}

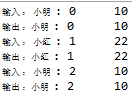
}

});

input.start();

output.start();

}

**结果为：**

**存在问题：**

**问题1：利用if（flag）进行判断，只能判断一次，如果线程wait了，则执行了其他线程，其他线程notify这个线程时，如果flag仍为true，则本应该再次wait，结果却执行了，从而执行了不该执行的任务。**

**改进方法：if（flag）=🡺 while(flag)，就会一直判断，直到false才会往下执行。**

**问题2：如果第一个线程已经wait了，第二个线程执行到notify时，仍然把自己唤醒，而不是把另外一个线程唤醒，该线程判断while（flag）时，又wait，则就会出现两个线程都阻塞，从而出现了死锁。这是由while –notify导致的死锁。**

**改进方法：while（flag）- notify => while(flag) –notifyAll,，利用了notifyAll就确保了一定能够把对方线程唤醒。**

1. **多生产者—多消费者模式与单生产者-单消费者模式：**

**（1）将if换成while，解决了线程获取执行权后，再次判断flag，决定是否执行；if只能判断一次，判断后如果满足if句则线程wait，等到唤醒后直接执行下面的代码，不再进行判断，从而导致不该运行的线程运行了；利用while可以解决这个问题，因为while是一直循环，直到不满足才会执行下面的代码。While + notify会导致程序死锁。**

**（2） notifyAll解决了保证能够唤醒对方的线程，而notify只可以唤醒一个线程，可能仍然把本方的线程唤醒，然后本方判断flag时又wait，导致所有的线程都冻结。**

1. **单生产者和单消费者模式：**

**资源：**

**public class Resource2 {**

**String name;**

**boolean flag;**

**int num = 0;**

**public Resource2(String name, boolean flag) {**

**super();**

**this.name = name;**

**this.flag = flag;**

**}**

**public synchronized void produce() {**

**if(flag) {**

**try {**

**this.wait();**

**} catch (InterruptedException e) {}**

**}**

**System.out.println("生产的"+this.name+"的编号是"+(++this.num));**

**flag = true;**

**this.notify();**

**}**

**public synchronized void consume() {**

**if(!flag) {**

**try {**

**this.wait();**

**} catch (InterruptedException e) {}**

**}**

**System.out.println("消费的"+this.name+"的编号是"+(this.num));**

**flag = false;**

**this.notify();**

**}**

**}**

**生产任务线程类：**

**public class Producer implements Runnable {**

**private Resource2 r;**

**public Producer(Resource2 r) {**

**this.r = r;**

**}**

**public void run() {**

**for (int i = 0; i < 20; i++) {**

**r.produce();**

**}**

**}}**

**消费任务线程类：**

**public class Consumer implements Runnable{**

**private Resource2 r;**

**public Consumer(Resource2 r) {**

**this.r = r;**

**}**

**public void run() {**

**for(int i = 0;i < 20;i++) {**

**r.consume();**

**}}}**

**主方法：**

**Resource2 r = new Resource2("烤鸭",false);**

**new Thread(new Producer(r)).start();**

**new Thread(new Consumer(r)).start();**

1. **多生产者和多消费者模式：**

**资源类：**

**public class Resource2 {**

**String name;**

**boolean flag;**

**int num = 0;**

**public Resource2(String name, boolean flag) {**

**super();**

**this.name = name;**

**this.flag = flag;**

**}**

**public synchronized void produce() {**

**while(flag) {**

**try {**

**this.wait();**

**} catch (InterruptedException e) {}**

**}**

**System.out.println(Thread.currentThread().getName()+"生产的"+this.name+"的编号是"+(++this.num));**

**flag = true;**

**this.notifyAll();**

**}**

**public synchronized void consume() {**

**while(!flag) {**

**try {**

**this.wait();**

**} catch (InterruptedException e) {}**

**}**

**System.out.println(Thread.currentThread().getName()+"消费的......."+this.name+"的编号是"+(this.num));**

**flag = false;**

**this.notifyAll();**

**}**

**}**

**任务线程类1：**

**public class Producer implements Runnable {**

**private Resource2 r;**

**public Producer(Resource2 r) {**

**this.r = r;**

**}**

**public void run() {**

**while(true)**

**r.produce();**

**}}**

**任务线程类2：**

**public class Consumer implements Runnable{**

**private Resource2 r;**

**public Consumer(Resource2 r) {**

**this.r = r;**

**}**

**public void run() {**

**while(true)**

**r.consume();**

**}}**

**主方法： Resource2 r = new Resource2("烤鸭",false);**

**Producer pr = new Producer(r);**

**Consumer co = new Consumer(r);**

**new Thread(pr,"生产机器1").start();**

**new Thread(pr,"生产机器2").start();**

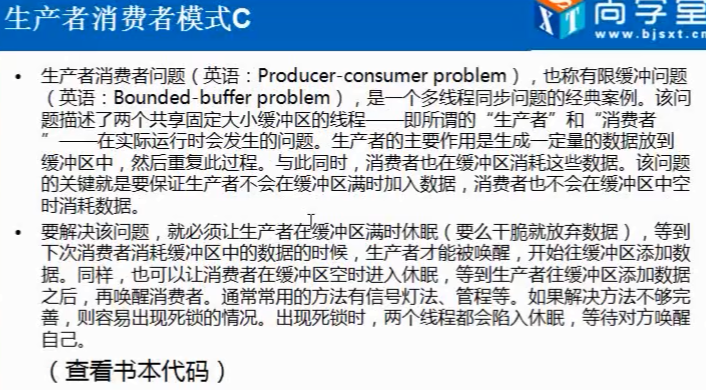
**new Thread(pr,"生产机器1").start();**

**new Thread(co,"消费机器1").start();**

**new Thread(co,"消费机器2").start();**

**new Thread(co,"消费机器3").start();**

1. **生产者与消费者模式：**



1. **多窗口售票的问题分析：不涉及到线程通信的问题，各自销售自己的。**

**多个窗口同时售票，但是票号是连续的，唯一的，所以是多窗口共享火车票这一个资源。共享的数据资源的处理方法是：把共享的数据资源封装成一个类，并封装该资源的属性和方法，比如火车票，单独封装成一个类，具有座号的属相，同时具有售出的方法，该方法应该是同步的，因为会有多个线程同时调用。多个窗口就是多个线程，把同一个数据资源对象传进每个线程中，由于火车票的售出方法本身就是同步的，所以是线程安全的。**

1. **生产者-消费者模式问题：涉及到生产线程和销售线程间的通信问题，生产线程间或销售线程间不需要通信。**

**这样的话，线程就分为两类：**

1. **生产线程2、消费线程。**

**利用信号灯的方法区分生产还是消费。该信号灯应该封装在产品类中，因为是需要生产还是销售，是产品自己的特性。比如：信号灯flag ：true时，需要生产，false时，需要消费。**

**分析：由于生产者和消费者操作的是同一个资源，即共享数据，那么就需要把操作对象封装单独的类，例如生产汽车，生产厂有多个，销售厂也有多个，但是都是操作汽车，汽车被生产或被销售是汽车的方法，所以把汽车的属性及方法封装到汽车类中。多个生产厂都会调用生产方法，多个销售厂也都会调用销售方法，所以生产方法和销售都必须是同步的。**

**特殊的问题是：产品只有先被生产，才能被销售。必须对产品的数量进行控制。即不能无限生产，也不能无限销售。涉及到线程间的通信问题。当生产一定数量时，生产厂必须停止生产，呼唤销售厂进行销售；同理销售完时，必须停止销售，呼唤生产厂进行生产。**

1. **单生产者-单消费者模式：**

**生产厂和销售厂都只有1个。生产者生产1个，便缓缓消费者消费这1个。也就是生产一个消费一个。**

**重点分析资源Car的生产和消费方法。**

**public** **class** Car {

**private** String name;

**private** **int** maxNum;

**private** **int** num = 0;

**public** **boolean** flag = **true**;//默认先生产

**public** Car() {

}

**public** Car(String name, **int** maxNum) {

**super**();

**this**.name = name;

**this**.maxNum = maxNum;

}

**public** **synchronized** **void** produce() {

**while**(!flag) {**//信号灯flag为true时，需要生产，就跳出循环，为false，就wait，释放了锁，等待被唤醒。不能换成if语句，因为如果为false，则此线程wait，释放了锁，然后被另外一个生产线程唤醒，就不再判断信号灯flag，就去生产了。但是此时flag仍然为false，本是不应该生产的。从而造成错误。**

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

}

**if** (**this**.num < **this**.maxNum) {//对后续有用，再此处无用

System.***out***.println(Thread.*currentThread*().getName()+"生产第 "+ ++**this**.num +" 个Car");

**this**.flag = !**this**.flag;//不再允许生产

**this**.notifyAll();//呼唤消费者去消费

}

}

**public** **synchronized** **void** consume() {

**while**(flag) {

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

}

**if** (**this**.num > 0) {

System.***out***.println(Thread.*currentThread*().getName()+"消售第 "+ **this**.num-- +" 个Car");

**this**.flag = !**this**.flag;//不允許銷售了，这句话会造成所有的销售线程wait。所以在此处就不要再调用wait方法了，且不能调用wait方法，因为先wait，后面notiyAll就不会执行了，从而可能所有的线程都wait，从而死锁了。

**this**.notifyAll();

}

}

}

**生产者：**

**public** **class** Producer **implements** Runnable {

**private** Car car;

**public** Producer(Car car) {

**super**();

**this**.car = car;

}

@Override

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

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

car.produce();

}

}

**消费者：**

**public** **class** Consumer **implements** Runnable {

**private** Car car;

**public** Consumer(Car car) {

**super**();

**this**.car = car;

}

@Override

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

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

car.consume();

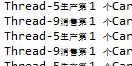
}

}

Car car = **new** Car("奔馳",200);

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**结果：**

**当然也可以控制生产m个消耗m个，交替进行。如何控制呢？**

**只有生产了m个，才把flag取反，关闭生产，允许销售；同理只有消耗了，m个关闭消耗，允许生产。**

**public** **synchronized** **void** produce() {

**while**(!flag) {

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

}

**if** (**this**.num < **this**.maxNum) {

System.***out***.println(Thread.*currentThread*().getName()+"生产第 "+ ++**this**.num +" 个Car");

**if (this.num == 5) {//只有生产5个才关闭生产，允许消费**

**this.flag = !this.flag;**

**}**

**this**.notifyAll();

}

}

**public** **synchronized** **void** consume() {

**while**(flag) {

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

**if** (**this**.num > 0) {

System.***out***.println(Thread.*currentThread*().getName()+"消售第 "+ **this**.num-- +" 个Car");

**if** (**this**.num == 0) {**//只有消耗完，才关闭消费，允许生产**

**this**.flag = !**this**.flag;

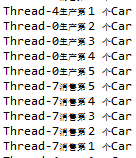
}

**this**.notifyAll();

}

}

结果：



1. **多生产者-多消费者模式：**

**生产厂和销售厂都有多个。**

**和上面差不多。**

**利用maxNum控制生产的最大量。完成的任务是：生产到maxNum量时，停止所有的生产，然后消费，等到消费完后再停止消费，允许生产。即0—》numMax—》0…….**

**资源：**

**public** **class** Car {

**private** String name;

**private** **int** maxNum;

**private** **int** num = 0;

**public** **boolean** flag = **true**;//默认先生产

**public** Car() {

}

**public** Car(String name, **int** maxNum) {

**super**();

**this**.name = name;

**this**.maxNum = maxNum;

}

**public** **synchronized** **void** produce() {

**while**(!flag) {

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

}

**if** (**this**.num < **this**.maxNum) {

System.***out***.println(Thread.*currentThread*().getName()+"生产第 "+ ++**this**.num +" 个Car");

**if** (**this**.num == **this**.maxNum) {

**this**.flag = !**this**.flag;//不再允许生产

}

**this**.notifyAll();

}

}

**public** **synchronized** **void** consume() {

**while**(flag) {

**try** {

**this**.wait();

**this**.notifyAll();

} **catch** (InterruptedException e) {}

}

**if** (**this**.num > 0) {//產品還有允許銷售

System.***out***.println(Thread.*currentThread*().getName()+"消售第 "+ **this**.num-- +" 个Car");

**if** (**this**.num == 0) {

**this**.flag = !**this**.flag;//不允許銷售了,把flag取反已经相当于是让所有的生产线程wait了，没有必要再次wait。

}

**this**.notifyAll();

}

}

}

生产者：

**private** Car car;

**public** Producer(Car car) {

**super**();

**this**.car = car;

}

@Override

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

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

car.produce();

}

}

消费者：

**public** **class** Consumer **implements** Runnable {

**private** Car car;

**public** Consumer(Car car) {

**super**();

**this**.car = car;

}

@Override

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

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

car.consume();

}

}

测试：

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

Car car = **new** Car("奔馳",2000);

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Producer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**new** Thread(**new** Consumer(car)).start();

**new** Thread(**new** Consumer(car)).start();

}

}

结果：**就是多个生产者交替完成了2000个，然后多个消费者交替消耗了2000个。**

**flag很关键，控制着生产和消耗。**