必须会手写的代码（可以的话打开notpad++）

生产者消费者代码

Synchronized版本  
**public class** PcBySynchronized {  
 **public static void** main(String[] args) {  
 Data data = **new** Data();  
 *//生产者线程* **new** Thread(()->{  
 **try** {  
 **for** (**int** i = 0; i <10 ; i++) {  
 data.inCrement();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"A"**).start();  
 *//消费者线程* **new** Thread(()->{  
 **try** {  
 **for** (**int** i = 0; i <10 ; i++) {  
 data.deCrement();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"B"**).start();  
 }  
}  
*/\*  
\* 资源类，用于提供资源，也就是变量+需要的方法（data和生产的方法，消费的方法）  
\* \*/***class** Data {  
 **public int data** = 0;  
 *//生产的方法  
 //判断、等待、业务、通知* **public synchronized void** inCrement() **throws** InterruptedException {  
 *//此处不能使用if，防止虚假唤醒* **while** (**data** != 0) {  
 **this**.wait();  
 }  
 **data**++;  
 System.***out***.println(**"线程生产"** + Thread.*currentThread*().getName() + **"==>"** + **data**);  
 **this**.notifyAll();  
 }  
  
 *//消费的方法  
 //判断、等待、业务、通知* **public synchronized void** deCrement() **throws** InterruptedException {  
 *//此处不能使用if，防止虚假唤醒* **while** (**data** == 0) {  
 **this**.wait();  
 }  
 **data**--;  
 System.***out***.println(**"线程消费"** + Thread.*currentThread*().getName() + **"==>"** + **data**);  
 **this**.notifyAll();  
 }  
}

Lock版本

**public class** PcByLock {  
 **public static void** main(String[] args) {  
 DataLock data = **new** DataLock();  
  
 **new** Thread(()->{  
 **try** {  
 **for** (**int** i = 0; i <10 ; i++) {  
 data.inCrement();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"A"**).start();  
 **new** Thread(()->{  
 **try** {  
 **for** (**int** i = 0; i <10 ; i++) {  
 data.deCrement();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 },**"B"**).start();  
 }  
}  
  
**class** DataLock {  
 **public int data** = 0;  
 Lock **lock** = **new** ReentrantLock();  
 Condition **condition** = **lock**.newCondition();  
 *//生产的方法  
 //判断、等待、业务、通知* **public void** inCrement() **throws** InterruptedException {  
 *//此处不能使用if，防止虚假唤醒* **lock**.lock();  
 **try** {  
 **while** (**data** != 0){  
 **condition**.await();  
 }  
 **data**++;  
 System.***out***.println(**"线程生产"** + Thread.*currentThread*().getName() + **"==>"** + **data**);  
 **condition**.signalAll();  
 } **catch** (Exception e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
 }  
  
 *//消费的方法  
 //判断、等待、业务、通知* **public void** deCrement() **throws** InterruptedException {  
 *//此处不能使用if，防止虚假唤醒* **lock**.lock();  
 **try** {  
 **while** (**data** ==0){  
 **condition**.await();  
 }  
 **data**--;  
 System.***out***.println(**"线程消费"** + Thread.*currentThread*().getName() + **"==>"** + **data**);  
 **condition**.signalAll();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 }  
 }  
}

注意事项

While防止虚假唤醒：如果使用if会存在虚假唤醒，因为if值判断一次，可能在这之前有多个线程阻塞在if条件判断里，如果notifyAll会导致唤醒多个线程，而所有通过if条件判断的线程都会执行，而实际只能一条线程执行，就有其他线程被虚假唤醒，而while能防止虚假唤醒是因为即使线程获得cpu也会再进行一次条件判断，防止虚假唤醒。

单例模式，双重检测锁

**public class** SingleModle {  
 **private** SingleModle(){  
  
 }  
 *//使用valitale的原因，  
 /\*  
 \* 因为创建singleModle的过程分为三步，1开辟空间，2构造方法初始化，3将引用赋给singleModle  
 \* 这三步是没有关系的，可以进行指令重排序，如果是132的执行顺序，一旦3执行完就认为对象已经存在  
 \* 可能会导致一个没有初始化的对象就被返回出去，  
 \* 使用volatile可以避免这种情况  
 \*  
 \* \*/* **private static volatile** SingleModle *singleModle* = **null**;  
  
 */\*  
 \* 为什么需要双重检（两次if）测锁，因为在第一次初始化对象的时候可能会有多个对象同时执行“singleModle == null”  
 \* 导致多个线程进入，如果不再多一次if判断，可能会创建多个对象  
 \*  
 \* \*/* **public** SingleModle newInstance(){  
 **if** (*singleModle* == **null**){  
 **synchronized** (SingleModle.**class**){  
 **if** (*singleModle* == **null**){  
 *singleModle* = **new** SingleModle();  
 }  
 }  
 }  
 **return** *singleModle*;  
 }  
}

死锁

**package** com.example.juc.code.deadLock;  
  
**public class** deadlock {  
 **public static void** main(String[] args) {  
 *//先获取黑再获取白* **new** Thread(**new** chopsticks(**"black"**,**"white"**)).start();  
 *//先获取白再获取黑* **new** Thread(**new** chopsticks(**"white"**,**"black"**)).start();  
 }  
}  
  
**class** chopsticks **implements** Runnable{  
 *//两个筷子，一黑一白，必须都获取才可以吃饭* **private** String **chopsticks1** ;  
 **private** String **chopsticks2** ;  
  
  
 **public** chopsticks(String chopsticks1, String chopsticks2) {  
 **this**.**chopsticks1** = chopsticks1;  
 **this**.**chopsticks2** = chopsticks2;  
 }  
  
 **public void** run() {  
 **synchronized** (**chopsticks1**){  
 System.***out***.println(Thread.*currentThread*().getName()+**"获得"**+**chopsticks1**+**"筷子"**);  
 **try** {  
 Thread.*sleep*(1000);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 }  
 **synchronized**(**chopsticks2**){  
 System.***out***.println(Thread.*currentThread*().getName()+**"获得"**+**chopsticks2**+**"筷子"**);  
 };  
 }  
 }  
}

快速排序

原理：<https://www.e-learn.cn/content/java/1545532>

public class Qsort {  
  
 private static void sort(int[] arr,int low ,int hight){  
 if (low>=hight){  
 return;  
 }  
 int i,j,temp,t;  
 i = low;  
 j = hight;  
 //temp是基准位置  
 temp = arr[low];  
  
 while (i<j){  
 //从后向前看，如果后面比基准位大则前移  
 while (temp <= arr[j] && i<j ){  
 j--;  
 }  
 //从前向后看，如果前面比基准位小，表示不用变，则后移  
 while ( temp >= arr[i] && i<j){  
 i++;  
 }  
  
 //如果满足条件则交换  
 if (i<j) {  
 t = arr[j];  
 arr[j] = arr[i];  
 arr[i] = t;  
 }  
 }  
 //将temp和arr[low]进行交换，前面已经temp = arr[low]，现在执行下面两个就可以  
 arr[low] = arr[i];  
 arr[i] = temp;  
 //递归调用左半数组  
 *sort*(arr, low, j-1);  
 //递归调用右半数组  
 *sort*(arr, j+1, hight);  
 }  
  
 public static void main(String[] args) {  
 int[] arr = {5,5,6,98,65,2,6,44,89,51,94,15,1};  
 *sort*(arr,0,arr.length-1);  
 for (int i = 0; i < arr.length; i++) {  
 System.*out*.println(arr[i]);  
 }  
  
 }