1.JVM
JVM内存模型：PC，虚拟机栈，本地方法栈，Java堆，方法区
PC：字节码解释器工作时就是通过改变这个计数器的值来选取下一条需要执行的字节码指令，分支、循环、跳转、异常处理、线程恢复等基础功能都需要依赖这个计数器来完成。
虚拟机栈：每个方法被执行的时候都会创建一个栈帧(Stack Frame)用于存储局部变量表、操作栈、动态链接方法、方法出口等信息。每一个方法被调用直至执行完成的过程，就对应着一个栈帧在虚拟机栈中从入栈到出栈的过程。局部变量表所需的内存空间在编译期间完成分配。当进入一个方法时，这个方法需要在帧中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变。
本地方法栈：虚拟机栈为虚拟机执行Java方法(也就是字节码)服务，而本地方法栈则是为虚拟机使用到的Native方法服务。
如果从内存分配的角度看，线程共享的Java堆中可能划分出多个线程私有的分配缓冲区(ThreadLocalAllocationBuffer)。
Java堆：几乎所有的对象和数组都是在堆中分配空间的，分为老生代和新生代
新生代可分为eden，survivor space 0，survivor space 1
方法区：它用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据，其信息大部分来自class文件。
在Hot spot虚拟机中，方法区也叫永久区，但是也会被GC，GC主要有两类：对常量池的回收，对类元数据的回收
**如果VM确认所有该类的实例都被回收并且装载该类的类加载器也被回收了，那么就回收该类的元数据
**运行时常量池(Runtime Constant Pool)是方法区的一部分。(不一定)
Java语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中常量池的内容才能进入方法区运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用得比较多的便是 String 类的 intern()方法。
JVM参数：设置最大堆内存Xmx，最小堆内存Xms，新生代大小Xmn，老年代大小PermSize，线程栈大小Xss，新生代eden和s0空间大小比例以及老年代和新生代的空间大小比例

垃圾回收算法
（1）引用计数法：缺点是无法处理循环引用问题
（2）标记-清除法：标记所有从根节点开始的可达对象，清除所有未被标记的对象
缺点是会造成内存空间不连续，不连续的内存空间的工作效率低于连续的内存空间，不容易分配内存
（3）复制算法：将内存空间分成两块，每次将正在使用的内存中的存活对象复制到未使用的内存块中，之后清除正在使用的内存块
算法效率高，但是代价是将系统内存折半。适用于新生代（存活对象少，垃圾对象多）
（4）标记-压缩算法：标记-清除的改进，清除未标记的对象时还将所有的存活对象压缩到内存的一端，之后，清理边界外所有空间
既避免碎片产生，又不需要两块同样大小的内存块，性价比高。适用于老年代
（5）分代

JVM--标记-清除算法Mark-Sweep - andy521zhu的个人空间 - 开源中国社区
首先是mutator和collector，这两个名词经常在垃圾收集算法中出现，collector指的就是垃圾收集器，而mutator是指除了垃圾收集器之外的部分，比如说我们应用程序本身。mutator的职责一般是NEW(分配内存),READ(从内存中读取内容),WRITE(将内容写入内存)，而collector则就是回收不再使用的内存来供mutator进行NEW操作的使用。
第二个基本概念是关于mutator roots(mutator根对象)，mutator根对象一般指的是分配在堆内存之外，可以直接被mutator直接访问到的对象，一般是指静态/全局变量以及Thread-Local变量(在Java中，存储在java.lang.ThreadLocal中的变量和分配在栈上的变量 - 方法内部的临时变量等都属于此类).
第三个基本概念是关于可达对象的定义，从mutator根对象开始进行遍历，可以被访问到的对象都称为是可达对象。这些对象也是mutator(你的应用程序)正在使用的对象。

垃圾回收器的类型
（1）线程数：串行，并行　　　　并行：开启多个线程同时进行垃圾回收，缩短GC停顿时间
（2）工作模式：并发，独占　　　　并发：垃圾回收线程和应用程序线程交替工作
（3）碎片处理：压缩，非压缩
（4）分代：新生代，老年代

CMS：Concurrent Mark Sweep 并发标记清除，减少GC造成的停顿时间
过程：初始标记，并发标记，重新标记，并发清理，并发重置

2.多线程
JAVA多线程和并发基础面试问答（转载） - 海 子 - 博客园
生产者-消费者模式：Java BlockingQueue Example implementing Producer Consumer Problem | JournalDev
Java并发编程：线程间协作的两种方式：wait、notify、notifyAll和Condition - 海 子 - 博客园

java.util.concurrent.BlockingQueue的特性是：当队列是空的时，从队列中获取或删除元素的操作将会被阻塞，或者当队列是满时，往队列里添加元素的操作会被阻塞。
阻塞队列不接受空值，当你尝试向队列中添加空值的时候，它会抛出NullPointerException。
阻塞队列的实现都是线程安全的，所有的查询方法都是原子的并且使用了内部锁或者其他形式的并发控制。
BlockingQueue接口是java collections框架的一部分，它主要用于实现生产者-消费者问题。

线程调度器是一个操作系统服务，它负责为Runnable状态的线程分配CPU时间。一旦我们创建一个线程并启动它，它的执行便依赖于线程调度器的实现。时间分片是指将可用的CPU时间分配给可用的Runnable线程的过程。分配CPU时间可以基于线程优先级或者线程等待的时间。线程调度并不受到Java虚拟机控制，所以就是说不要让你的程序依赖于线程的优先级。
线程之间是如何通信的？当线程间是可以共享资源时，线程间通信是协调它们的重要的手段。Object类中wait()\notify()\notifyAll()方法可以用于线程间通信关于资源的锁的状态。
如何确保线程安全？在Java中可以有很多方法来保证线程安全——同步，使用原子类(atomic concurrent classes)，实现并发锁（Lock），使用volatile关键字，使用不变类和线程安全类。

Lock，设置文件共享锁，Lock 确保当一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻止），直到该对象被释放。

volatile 并不能保证线程安全　　jvm虚拟机栈->线程栈->方法栈
jvm有一个内存区域是jvm虚拟机栈，每一个线程运行时都有一个线程栈，线程栈保存了线程运行时候变量值信息。当线程访问某一个对象时候值的时候，首先通过对象的引用找到对应在堆内存的变量的值，然后把堆内存变量的具体值load到线程本地内存中，建立一个变量副本，之后线程就不再和对象在堆内存变量值有任何关系，而是直接修改副本变量的值，在修改完之后的某一个时刻（线程退出之前），自动把线程变量副本的值回写到对象在堆中变量。这样在堆中的对象的值就产生变化了。

当一个线程访问 object 的一个synchronized(this)同步代码块时，它就获得了这个 object 的对象锁。结果，其它线程对该 object 对象所有同步代码部分的访问都被暂时阻塞。
synchronized方法控制对类成员变量的访问：每个类实例对应一把锁，每个 synchronized 方法都必须获得调用该方法的类实例的锁方能执行，否则所属线程阻塞，方法一旦执行，就独占该锁，直到从该方法返回时才将锁释放，此后被阻塞的线程方能获得该锁，重新进入可执行状态。这种机制确保了同一时刻对于每一个类实例，其所有声明为 synchronized 的成员函数中至多只有一个处于可执行状态(因为至多只有一个能够获得该类实例对应的锁)，从而有效避免了类成员变量的访问冲突(只要所有可能访问类成员变量的方法均被声明为 synchronized)。同步函数使用的锁是this，静态同步函数的锁是该类的字节码文件对象。
synchronized块是这样一个代码块，其中的代码必须获得对象syncObject (如前所述，可以是类实例或类)的锁方能执行，具体机制同前所述。利用同步代码块可以解决线程安全问题。

等待唤醒机制:
wait:将同步中的线程处于冻结状态。释放了执行权，释放了资格，同时将线程对象存储到线程池中。
notify:唤醒线程池中某一个等待线程。 notifyAll:唤醒的是线程池中的所有线程。
****notify和notifyAll到底哪个线程先执行？！**
1:这些方法都需要定义在同步中。
2:这些方法必须要标示所属的锁。要知道A锁上的线程被wait了，那这个线程就相当于处于A锁的线程池中，只能A锁的notify醒。
3:这三个方法都定义在 Object 类中。
为什么操作线程的方法定义在 Object 类中？因为这三个方法都需要定义同步内,并标示所属的同步锁,既然被锁调用,而锁又可以是任意对象,那么能被任意对象调用的方法一定定义在 Object 类中。

wait 和 sleep 区别，从执行权和锁上来分析:
wait:可以指定时间也可以不指定时间。不指定时间,只能由对应的 notify 或者 notifyAll来唤醒。
wait:线程会释放执行权,而且线程会释放锁。
sleep:必须指定时间,时间到自动从冻结状态blocked转成运行状态(临时阻塞状态)。 ****到底哪个状态？！**
sleep:线程会释放执行权,但不是不释放锁。

①NEW：这种情况指的是，通过New关键字创建了Thread类（或其子类）的对象
②RUNNABLE：这种情况指的是Thread类的对象调用了start()方法，这时的线程就等待时间片轮转到自己这，以便获得CPU；第二种情况是线程在处于RUNNABLE状态时并没有运行完自己的run方法，时间片用完之后回到RUNNABLE状态；还有种情况就是处于BLOCKED状态的线程结束了当前的BLOCKED状态之后重新回到RUNNABLE状态。
③RUNNING：这时的线程指的是获得CPU的RUNNABLE线程，RUNNING状态是所有线程都希望获得的状态。
④DEAD：处于RUNNING状态的线程，在执行完run方法之后，就变成了DEAD状态了。
⑤BLOCKED：这种状态指的是处于RUNNING状态的线程，出于某种原因，比如调用了sleep方法、等待用户输入等而让出当前的CPU给其他的线程。

处于RUNNABLE状态的线程变为BLOCKED状态的原因，除了该线程调用了sleep方法、等待输入原因外，还有就是在当前线程中调用了其他线程的join方法、当访问一个对象的方法时，该方法被锁定等。
相应的，当处于BLocked状态的线程在满足以下条件时就会由该状态转到RUNNABLE状态，这些条件是：sleep的线程醒来（sleep的时间到了）、获得了用户的输入、调用了join的其他线程结束、获得了对象锁。
一般情况下，都是处于RUNNABLE的线程和处于RUNNING状态的线程，互相切换，直到运行完run方法，线程结束，进入DEAD状态。

2.集合框架
List:有序(元素存入集合的顺序和取出的顺序一致),元素都有索引。元素可以重复。
|--ArrayList:底层的数据结构是数组,线程不同步,ArrayList 替代了 Vector,查询元素的速度非常快。
|--LinkedList:底层的数据结构是链表,线程不同步,增删元素的速度非常快。
|--Vector:底层的数据结构就是数组,线程同步的,Vector 无论查询和增删都巨慢。
可变长度数组的原理: 当元素超出数组长度,会产生一个新数组,将原数组的数据复制到新数组中,再将新的元素添加到 新数组中。
ArrayList:是按照原数组的50%延长。构造一个初始容量为 10 的空列表。 Vector:是按照原数组的100%延长。

Set 接口中的方法和 Collection 中方法一致的。Set 接口取出方式只有一种,迭代器。
|--HashSet:底层数据结构是哈希表,线程是不同步的。无序,高效;
　　|--LinkedHashSet:有序,hashset 的子类。
|--TreeSet:对 Set 集合中的元素的进行指定顺序的排序。不同步。TreeSet 底层的数据结构就是二叉树。

HashSet 集合保证元素唯一性:通过元素的 hashCode 方法 和 equals 方法完成的。当元素的 hashCode 值相同时,才继续判断元素的 equals 是否为 true。如果为 true,那么视为相同元素,不存。如果为 false,那么存储。如果 hashCode 值不同,那么不判断 equals,从而高对象比较的速度。

对于 ArrayList 集合,判断元素是否存在,或者删元素底层依据都是 equals 方法。
对于 HashSet 集合,判断元素是否存在,或者删除元素,底层依据的是 hashCode 方法和 equals 方法。

3.常用的设计模式
（1）单例模式：实现方式很多，最常见的方式是将构造函数设置为private，类中保存一个private的静态单例对象，然后新建一个public static的函数例如getInstance，在这个方法中返回静态单例对象。如果考虑到延迟加载，为了在多线程环境保持单例需要同步关键字，但是这样做反而会增加时间消耗。最好的实现方式：使用内部类来维护单例的实例。
当StaticSingleton被加载的时候，其内部类并不会被初始化，所以实例也不会被初始化。而当getInstance方法被调用时，才会加载SingletonHolder，从而初始化instance。同时，由于实例的建立是在类加载时完成的，天生对多线程友好，getInstance方法不需要使用同步关键字。（类加载自身处理了多线程环境下的同步问题）

```
//双重判断的方式
class Single{
    private static Single s = null;
    private Single(){}
    public static Single getInstance(){ //锁是
谁?字节码文件对象;
        if(s == null){
            synchronized(Single.class){
                if(s == null)
                    s = new Single();
            }
        }
        return s;
    }
}
```

```
public class StaticSingleton {
    private StaticSingleton() {
    }

    private static class SingletonHolder {
        private static StaticSingleton
instance = new StaticSingleton();
    }

    public static StaticSingleton
getInstance() {
        return SingletonHolder.instance;
    }
}
```

（2）代理模式：
适用场景：（1）延迟加载，对真实对象进行封装；（2）网络代理，RMI；（3）安全代理，屏蔽客户端直接访问真实对象
延迟加载的核心思想：在真正需要某个组件的时候才去对它进行加载，其他时候使用代理对象即可，这样可以有效据提高系统启动速度。
多线程编程模式中的Future模式就是使用了代理模式，一般情况下都会有个接口，真实对象和代理对象都实现了这个接口。
动态代理：使用字节码动态生成加载技术，在运行时生成并加载类。应用场景：在运行时动态生成并加载代理类，与静态代理相比，这样做的好处就是不用为每个真实对象生成一个形式上一样的封装类，如果要修改就都要修改。
工具：JDK自带，CGLib，Javasist
（3）享元模式：如果在一个系统中存在多个相同的对象，那么只需要共享一份对象的拷贝，而不必为每一次使用都创建新的对象。
类似对象池，但是不同的是前者保存的对象是不可以相互替换的，而后者可以。
（4）装饰器模式：通过委托机制，复用系统中的各个组件，在运行时，可以将这些功能组件进行叠加，使其拥有所有这些组件的功能。
被装饰者是系统的核心组件，装饰者可以在被装饰者的方法前后加上特定的前置处理和后置处理，增强被装饰者的功能。
使用场景：JDK中的IO框架、输出html内容
（5）观察者模式：在单线程中使某一个对象及时得知自身所依赖的状态变化。实现将观察值添加到被观察者维护的观察者列表即可。

4.Java的4中引用类型
强引用：JVM宁愿抛出OOM也不会将它回收，可能导致内存泄露
软引用：当内存空间不足的时候才会去回收软引用的对象
弱引用：在系统GC时，弱引用的对象一定会被回收，软弱引用适合保存那些可有可无的缓存数据
虚引用：虚引用跟没有引用差不多，即使强引用对象还存在，get方法总是返回null，它最大的作用是跟踪对象回收，清理被销毁对象的相关资源
WeakHashMap适用场景：如果系统需要一张很大的map表，map中的表项作为缓存之用，即使没能从map中拿到数据也没关系的情况下。一旦内存不足的时候，weakhashmap会将没有被引用的表项清除掉，从而避免内存溢出。它是实现缓存的一种特别好的方式。
实现：Entry<K,V> extends WeakReference<K> implements Map.Entry<K,V>
Entry继承了WeakReference，并在构造方法中构造了key的弱引用
**如果希望WeakHashMap能够自动清理数据就不要在系统的其他地方强引用WeakHashMap的key，否则，这些key不会被回收。

5.类加载过程

================================ 常见面试问题 ================================
1.equals和==的区别：前者是由对象的equals方法决定的，后者是判断两个对象指向的内存空间的地址是否相同

2.string.intern方法，在JDK6中常量池是方法区的一部分，在JDK7及以上常量池放到了Java堆中。
intern方法调用时首先去常量池中找这个字符串，如果有就将该字符串的引用返回，如果没有就创建该字符串放到池中然后返回引用
String s = “11”; //在常量池中创建字符串11，并把它的引用赋值给s
String s = new String(“11”); //在常量池中创建字符串11，在堆中创建一个对象c，它指向常量池中的字符串，而s指向c

3.执行顺序:(优先级从高到低)静态代码块>mian 方法>构造代码块>构造方法。
其中静态代码块只执行一次。 构造代码块在每次创建对象是都会执行。

4.final 1:这个关键字是一个修饰符,可以修饰类,方法,变量。 2:被 final 修饰的类是一个最终类,不可以被继承。 3:被 final 修饰的方法是一个最终方法,不可以被覆盖。 4:被 final 修饰的变量是一个常量,只能赋值一次。

5.抽象类和接口的区别:
1:抽象类只能被继承,而且只能单继承。 接口需要被实现,而且可以多实现。
2:抽象类中可以定义非抽象方法,子类可以直接继承使用。 接口中都是抽象方法,需要实现类去实现。
3:抽象类使用的是 is a 关系。 接口使用的 like a 关系。
4:抽象类的成员修饰符可以自定义。
接口中的成员修饰符是固定的。全都是 public 的。

6.如果内部类被静态修饰,相当于外部类,会出现访问局限性,只能访问外部类中的静态成员。
注意：如果内部类中定义了静态成员,那么该内部类必须是静态的。内部类编译后的文件名为:“外部类名$内部类名.java”。
为什么内部类可以直接访问外部类中的成员呢?
那是因为内部中都持有一个外部类的引用。这个是引用是 外部类名.this。内部类可以定义在外部类中的成员位置上,也可以定义在外部类中的局部位置上。 当内部类被定义在局部位置上,只能访问局部中被 final 修饰的局部变量。

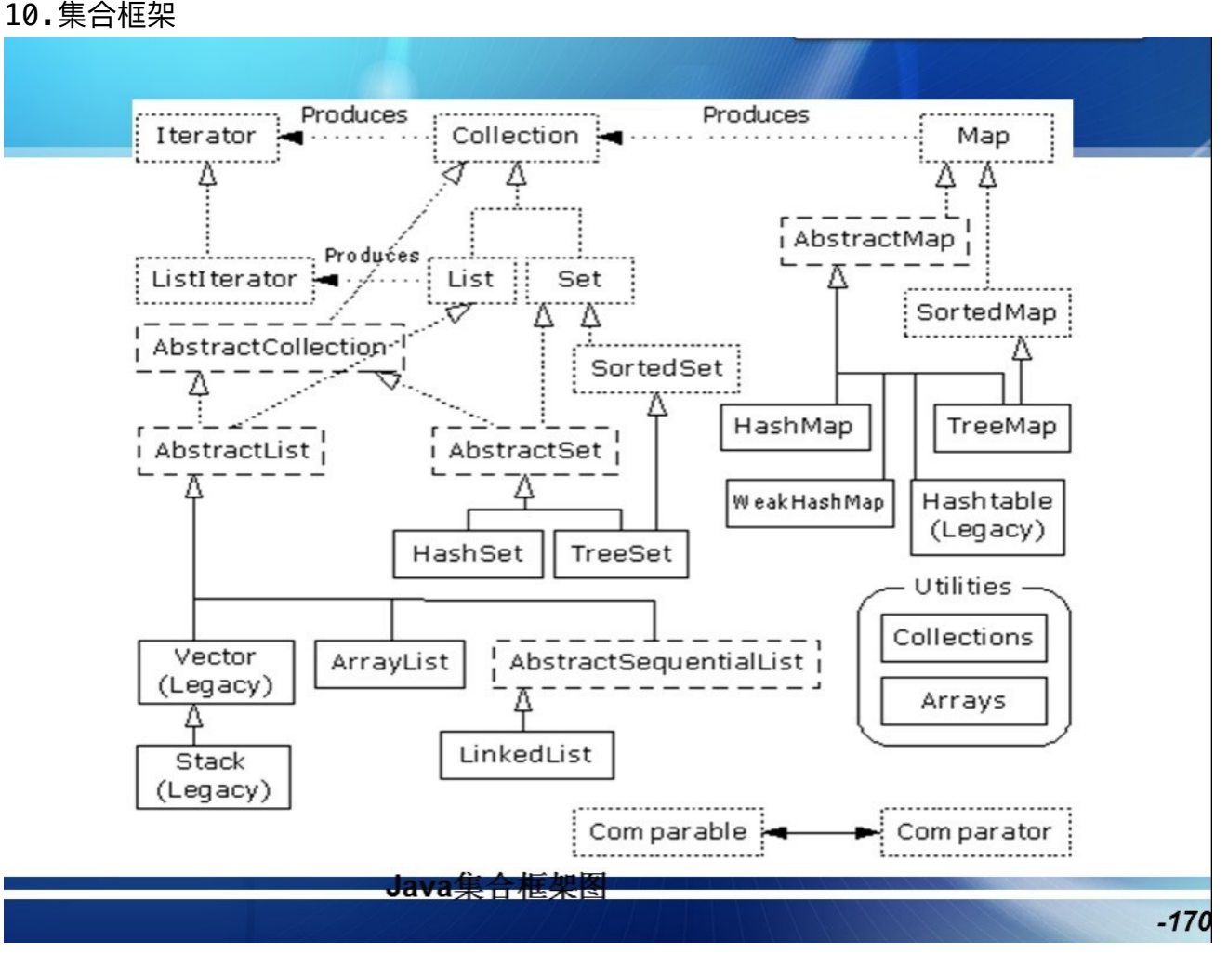
7.Hashmap和Hashtable的区别：（1）hashtable是线程安全的；（2）hashtable不允许key或者value为null，hash map可以；（3）在内部算法上，它们对key的hash算法和hash值到内存索引的映射算法不同。

8.hashcode和equals方法
**在一个运行的进程中，相等的对象必须要有相同的哈希码；不同的对象可以有相同的哈希码；
（1）无论你何时实现equals方法，你必须同时实现hashCode方法，；
（2）永远不要把哈希码误用作为key，哈希冲突是很常见的事情； hashmap中的contains方法的实现！
（3）哈希码可变，hashcode并不保证在不同的应用执行中得到相同的结果；
（4）在分布式应用中不要使用哈希码。
替代哈希码：SHA1，加密的哈希码，160位密钥，冲突几乎是不可能的。

9.RuntimeException与其他Exception的区别
Error体系 ：Error类体系描述了Java运行系统中的内部错误以及资源耗尽的情形。应用程序不应该抛出这种类型的对象（一般是由虚拟机抛出）。如果出现这种错误，除了尽力使程序安全退出外，在其他方面是无能为力的。所以，在进行程序设计时，应该更关注Exception体系。

Exception体系包括RuntimeException体系和其他非RuntimeException的体系 ：
① RuntimeException：RuntimeException体系包括错误的类型转换、数组越界访问和试图访问空指针等等。处理RuntimeException的原则是：如果出现RuntimeException，那么一定是程序员的错误。例如，可以通过检查数组下标和数组边界来避免数组越界访问异常。
②其他非RuntimeException（IOException等等）：这类异常一般是外部错误，例如试图从文件尾后读取数据等，这并不是程序本身的错误，而是在应用环境中出现的外部错误。

10.集合框架



Java集合框架图

** 生产者消费者代码

```
public class ProducerConsumerTest {
    public static void main(String[] args) {
        PublicResource resource = new PublicResource();
        new Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
        new Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
        new Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
    }
}

/**
 * 生产者线程，负责生产公共资源
 */
class ProducerThread implements Runnable {
    private PublicResource resource;

    public ProducerThread(PublicResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        while (true){
            try {
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.increase();
        }
    }
}

/**
 * 消费者线程，负责消费公共资源
 */
class ConsumerThread implements Runnable {
    private PublicResource resource;

    public ConsumerThread(PublicResource resource) {
        this.resource = resource;
    }

    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.decrease();
        }
    }
}

/**
 * 公共资源类
 */
class PublicResource {
    private int number = 0;
    private int size = 10;

    /**
     * 增加公共资源
     */
    public synchronized void increase() {
        while (number >= size) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        number++;
        System.out.println("生产了1个，总共有" + number);
        notifyAll();
    }

    /**
     * 减少公共资源
     */
    public synchronized void decrease() {
        while (number <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        number--;
        System.out.println("消费了1个，总共有" + number);
        notifyAll();
    }
}
```

** 实现LRU Cache

```
public class LRUCache {
    private class Node{
        Node prev;
        Node next;
        int key;
        int value;

        public Node(int key, int value) {
            this.key = key;
            this.value = value;
            this.prev = null;
            this.next = null;
        }
    }

    private int capacity;
    private HashMap<Integer, Node> hs = new HashMap<Integer, Node>();
    private Node head = new Node(-1, -1);
    private Node tail = new Node(-1, -1);

    public LRUCache(int capacity) {
        this.capacity = capacity;
        tail.prev = head;
        head.next = tail;
    }

    public int get(int key) {
        if( !hs.containsKey(key)) {
            return -1;
        }

        // remove current
        Node current = hs.get(key);
        current.prev.next = current.next;
        current.next.prev = current.prev;

        // move current to tail
        move_to_tail(current);

        return hs.get(key).value;
    }

    public void set(int key, int value) {
        if( get(key) != -1) {
            hs.get(key).value = value;
            return;
        }

        if (hs.size() == capacity) {
            hs.remove(head.next.key);
            head.next = head.next.next;
            head.next.prev = head;
        }

        Node insert = new Node(key, value);
        hs.put(key, insert);
        move_to_tail(insert);
    }

    private void move_to_tail(Node current) {
        current.prev = tail.prev;
        tail.prev = current;
        current.prev.next = current;
        current.next = tail;
    }
}
```