JVM概念：Java虚拟机，是通过软件模拟Java字节码指令集实现的，JVM只保留了pc寄存器，其他虚拟机有很多寄存器。
默认虚拟机 HotSpot

Java内存区域划分
以下内存划分基于JVM HotSpot，虚拟机不同Java内存区域划分可能也有些差异。
线程私有区域：程序计数器、Java虚拟机栈、本地方法栈
线程共享区域：GC堆、方法区、运行时常量池
线程私有指的是每个线程都有独立的该区域，彼此之间完全隔离互不影响。
线程共享指的是所有线程共享该区域。
程序计数器(线程私有)：
作用：若执行的是java方法，则记录当前方法的执行行号；若为native方法，则计数器值为空。因虚拟机同一时刻只能有一个线程(多核CPU同一时刻只有单核)执行，多线程情况下，虚拟机中通过多个线程不断轮流切换并分配处理器时间来实现高并发，为了实现线程的连续执行故需要程序计数器记录其行号，每个线程之间的程序计数器都是独立的，互不影响。
该区域是唯一一个不会产生OOM(内存溢出)异常的区域。
Java虚拟机栈(线程私有)：
Java虚拟机栈对应的是Java方法的内存模型。
作用：
1.每一个方法在创建时同时会创建一个栈帧用于存放局部变量表(决定虚拟机栈开辟空间大小)、方法出口等信息，每一个方法的执行过程都对应着栈帧在虚拟机栈的入栈和出栈过程。

2.生命周期与线程相同。创建线程时会同时创建此线程的虚拟机栈，线程执行结束，虚拟机栈与线程一同被收回。

局部变量表：存放了编译器可知的各种基本数据类型(8大基本类型)、对象引用(无论是哪个类型的对象引用均占4个字节)。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，在执行期间不会改变局部变量表的大小。

该区域会产生两种异常：栈溢出和内存溢出。

若线程请求的栈深度大于虚拟机所允许的深度(-Xss设置栈容量)，将会抛出StackOverFlowError异常。

虚拟机动态扩展时无法申请到足够的内存，会抛出OOM异常(OutOfMemoryError异常)

本地方法栈(线程私有)

该区域作用与Java虚拟机栈一样，只不过操作对象是本地方法。在HotSpot虚拟机中，Java虚拟机栈和本地方法栈是同一块区域。

线程共享区域：

GC堆(线程共享)--GC堆是JVM所管理的最大内存的空间。

GC堆是所有线程共享的一块区域，在JVM启动时创建。主要存储对象实例和数组，垃圾回收主要区域。根据JVM规范规定的内容，Java堆可以处于物理上不连续的内存空间中。Java堆在主流的虚拟机中都是可扩展的(-Xmx设置最大值，-Xms设置最小值)

如果在堆中没有足够的内存完成实例分配并且堆也无法再扩展时，会产生OOM。

方法区(线程共享)：

方法区与Java堆一样，是各个线程共享的内存区域。它用于存储已被虚拟机加载的类信息、常量(字面量与符号引用)、静态变量等数据。在

JDK8以前的HotSpot虚拟机中，方法区也被称为"永久代"，JDK8之后称为元空间。

永久代不意味着数据进入方法区就永久存在，此区间的内存回收主要是针对常量池的回收以及对类型的卸载。

当方法去无法满足内存分配需求时，将抛出OOM异常。

运行时常量池(方法区的一部分)

存放字面量和符号引用。

字面量：字符串(JDK1.7后移动到堆中)、final常量、基本数据类型的值

符号引用：类和结构的完全限定名(包名.类名)、字段的名称和描述符、方法的名称和描述符(方法的完全名称：包名+类名 权限修饰符)。

Java堆溢出

-Xmx设置堆最大值，-Xms设置堆最小值

若设置的最大值与最小值相等，表明当前堆不可扩展。

Java堆中出现OOM异常有两个原因：

1.内存泄漏：无用对象无法被GC

2.内存溢出：内存中的对象确实还改存活，但由于堆内存空间不够而产生的异常。

那么如何判断到底是内存泄漏还是内存溢出呢?

若扩展内存能解决的是内存溢出，反之为内存泄漏。

虚拟机栈和本地方法栈溢出

-Xss设置栈容量

会出现两个异常：栈溢出和内存溢出。

当线程请求的栈深度大于虚拟机所允许的深度就会报StackOverFlow异常(常见于单线程情况下)---如递归没有写出口

若虚拟机在拓展栈时无法申请到足够的空间，则会抛出OOM异常。(常见于多线程情况下)

回收Java堆

如何判断对象已死?

JVM采用的是可达性分析法。该方法思路为，设定常量或局部变量或静态变量或本地方法的引用对象为GC Roots，判断任意对象GC Roots是否可达，若可达表明该对象存活。若不可达对该对象进行第一次标记并进行第一次筛选，若该对象未覆写finalize()方法或finalize()方法已被调用，则该对象已死；若该对象覆写了finalize()方法且未调用，则调用finalize()方法，若当前对象与GC Roots重新建立连接，则该对象被移除回收集合，该对象自我拯救成功，若依旧未建立连接则该对象已死。

JVM何时GC是由JVM控制的，程序员不能直接控制，但我们可以通过以下两种法师获得GC时间：

1.虚引用　2.finalize()(因只能调用一次)

注意：finalize()放啊只能被调用一次，若该对象已被调用过且被放入了回收集合，那么该对象直接可判断为已死。

JDK1.2之后，Java引用有以下四种：

强引用：一般代码中常见的都为强引用，如new出来的对象都是强引用。强引用在内存不够用时也不会对其进行回收。

软引用：软引用存储的是有用但不必须的对象。对于仅被软引用指向的对象，若内存够用则不进行回收，若即将要发生内存溢出，则对该部分对象进行垃圾回收。SoftReference

弱引用：弱引用存储的也是有用但不必须的对象。对于仅被弱引用指向的对象，无论内存够不够用，再一次GC时都会被回收。

WeakReference

虚引用：虚引用的存在与对象生存关系毫无关系，若对象只被虚引用指向则不能产生对象，使用虚引用的唯一目的就是返回GC时间。

PhantomReference

引用计数法(Python)：引用一次加1，解除引用减1，当引用次数为0时表明该对象已死。但该方法不能解决循环引用的问题，故JVM使用的是可达性分析。

方法区回收(元空间)

永生代几乎不会产生垃圾回收。该区域垃圾回收主要针对废弃常量和无用的类。

对常量的回收：若内存够用就不回收，若没有任何一个引用指向该常量且内存不够用时对其回收。

判断无用的类需满足以下三个要点：

1.Java堆中没有一个该类的实例对象存在；

2.该类的ClassLoader(类加载器被回收);

3.该类对应的Class对象没有在任何其他地方被引用，无法在任何地方通过反射访问该类的方法

注意：JVM对同时满足这三点的无用类可以进行回收，但不是必然，因在大量使用反射、动态代理等场景都需要JVM具备类卸载的功能来防止永久代的溢出。

垃圾回收算法

JVM采用分代收集算法

即根据对象的生命周期将其划分为新生代和老年代。

新生代有“朝生夕死”的特性。即存活时间短，每次GC都会有大部分对象死亡，少部分对象存活。故使用复制算法进行新生代的垃圾回收。

复制算法思路：

将新生代内存划分为一块较大的Eden区，两块较小的From区和To区，Eden:From:Eden=8：1：1。当Eden区满了，第一次发生GC，此时标记Eden区已死对象，将存活对象复制到From区，回收Eden区。再次

GC时，标记Eden区和From区已死对象，将这两个区的存活对象复制到To区，将这两个区回收；往返如此，即新生代内存每次利用率为90%，效率高。

注意：当有对象在From区和To区移动超过一定次数(默认15次)时移动至老年代。

老年代对象存活时间较长，不适合再使用复制算法，因使用复制算法会使大量算法在From区和To区之间移动，较低效率。故老年代垃圾回收使用“标记-整理”算法。

“标记-整理算法”思路：

对内存中已死对象进行标记，回收的同时将存活对象向内存一端移动。解决了碎片空间的问题。

面试题：Minor GC和Full GC的区别?

MInor GC又称为新生代GC，指的是发生在新生代的垃圾收集。因为新生代对象大多具备朝生夕死的特性，因此MInor GC(采用复制算法)非常频繁，一般回收速度也非常快。

Full GC又称为老年代GC，指发生在老年代的垃圾收集。出现了Full GC，经常会伴随至少一次的Minor GC(但也并非绝对，在某些收集器中就有直接进行Full GC的策略选择过程)。Major GC的速度一般会比Minor GC慢10倍以上。

Java的内存模型(JMM)---基于线程的内存模型(多线程)

分为主内存和工作内存。主内存主要是线程共享区域，如GC堆、方法区、运行时常量池；工作内存是每个线程所私有的，如程序计数器、Java虚拟机栈、本地方法栈。

JMM定义的主要目标是为了定义程序中各个变量的访问规则(JVM如何从主内存中取出变量以及如何从工作内存写回等细节)。此处的变量包括实例字段、静态字段与数组元素。

注意：

1.JMM规定所有变量必须存储在主内存中

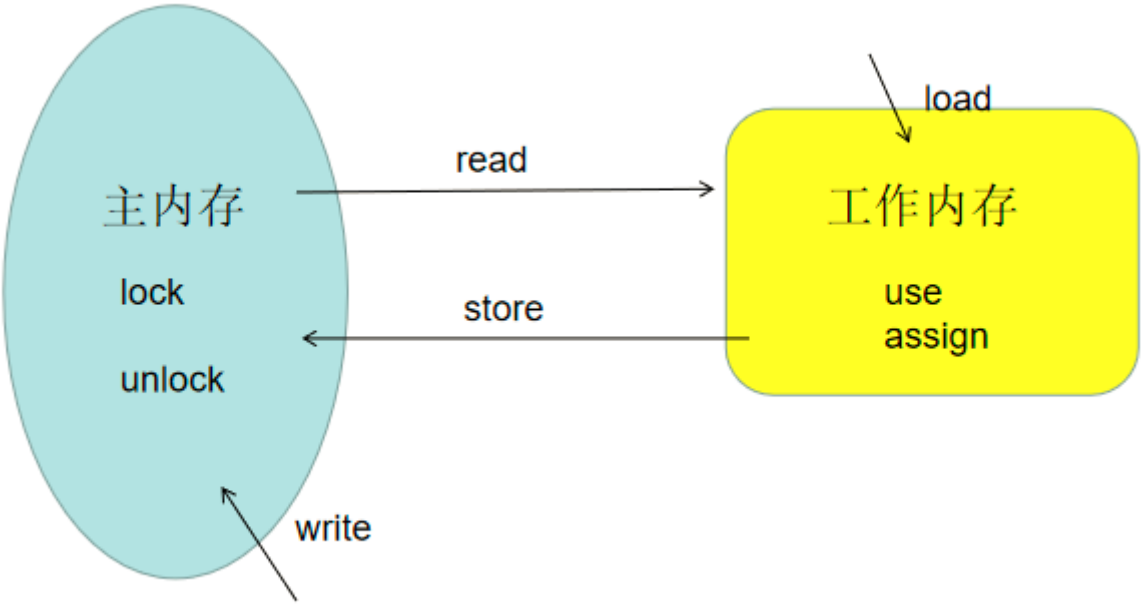
2.每个线程都有自己的工作内存，线程的工作内存中保存了该线程使用到的变量的主内存副本。

3.线程对变量的所有操作(读取、赋值)等都必须在工作内存中进行，不能直接读写主内存变量。

4.不同线程之间也无法直接访问彼此的工作内存变量，线程间变量值的传递均需要通过主内存来完成。

线程不安全的原因：因不同线程之间无法直接访问彼此的工作内存，变量的读写都需从主内存获得，高并发情况下，若某一线程对变量进行修改并写回主内存，此时系统调度到另一线程，该线程还未重新读取主线程中变量的最新值(也就是说若某一线程对变量值进行修改，其他线程不能立即知道)，此种情况就会导致变量值有误，线程不安全。

主内存与工作内存间的相互操作：



Java内存模型的三大特性：

1.原子性。基本数据类型的访问读写是具备原子性的(即要么都发生要么都不发生)。若想扩大原子性范围，可通过synchronized关键字或Lock体系实现。

i++/i--需要通过synchronized关键字或Lock体系保证多行代码的原子性，故线程同步相当于保证其可见性和原子性。

2.可见性。当一个线程修改了共享变量的值时，其他线程立刻得知。volatile、synchronized、final这三个关键字可保证可见性。

3.有序性。在线程内部看线程操作是有序的，原因JVM虽会进行优化，存在指令重排，但同一变量若赋值多次，必定按照写入顺序执行；在线程内部看其他线程操作是无序的，原因是指令重排。

JMM具备先天的有序性，即不需要任何手段就能够保证有序性，这个也通常称为happens-before(先行发生原则)。若两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证有序性，虚拟机可随意对它们进行重排序。

happens-before原则：

1.程序次序规则：针对同一变量而言，按照写入顺序执行。

2.锁定规则：unlock操作发生在lock操作之后。

3.volatile规则：对一个变量的写操作一定发生在读操作之前。

4.传递规则：若操作A先行发生于操作B,操作B先行发生于操作C，那么操作A一定先行发生于操作C。

5.线程启动规则：一个线程的启动操作一定发生在该线程的所有操作之前。

6.线程终端规则：对线程interrupt()方法的调用一定发生在程序检测到中断异常之前

7.线程终结规则：线程的所有操作都发生于线程终结操作之前。可使用Thread.join()方法结束、Thread.isAlive()的返回值检测线程是否终止

8.对象终结规则：一个对象的finalize()方法的使用在初始化该对象操作之后

注意：要想并发程序正确运行，必须同时保证原子性、可见性和有序性，只要有任意一个没保证则会导致并发程序运行有误。

volatile：只能保证可见性，不能保证原子性和有序性

关键字volatile是Java提供的最轻量级的同步机制。

volatile变量的特殊规则：

1.保证此变量对所有线程的可见性

若变量倍volatiel关键字修饰，则表明当该变量被某一线程修改时其他线程会立刻知道。普通变量无法做到可见性。

为什么被volatile关键字修饰就可保证可见性呢?

因happens-before原则中对于volatile变量规定对于一个变量的写操作一定发生在读操作之前。普通变量没有此规定。

2.禁止指令重排

volatie修饰的变量在程序执行过程中不会提前执行也不会推迟执行，一定会在程序那行执行，且它保证了执行到该句时改行前的所有代码已执行完毕且执行结果对改行之后的代码可见。禁止指令重排指的是整体上以该行为界限，一定是先执行该行之前的代码，再执行改行，最后执行改行之后的代码，但改行之前代码和之后代码在执行时是根据JVM优化决定执行顺序的。

注意：volatile变量在各个线程中是一致的，但在高并发情况下依旧是不安全的，因volatile不能保证原子性，若遇到非原子性操作，必须结合synchronized或Lock体系进行约束。

只需volatile就可保证结果准确的两种情况：

1.运算结果不依赖变量的当前值，或能够确保只有单一的线程修改的变量的值

2.变量不需要与其他状态变量共同参与不变约束

双锁检验锁模式：

懒汉式单例模式

```
public class Singleton{
 private static volatile Singleton singleton;
 private Singleton(){}
 public static Singleton print(){
 if(singleton==null){
 synchronized(Singleton.class){
 if(singleton==null){
 singleton=new Singleton();
 }
  }
  }
  return singleton;
  }
}
public class Test{
 public static void main(String[] args){
 Singleton singleton=Singleton.print();
 System.out.println(singleton);
  }
}
```

代码singleton=new Singleton()；new一个对象可看作三个指令：

1.在堆上开辟空间；

2.属性初始化；

3.引用指向对象。

若变量没有使用volatile关键字修饰，则此处可能发生指令重排，当执行顺序是1->3->2时(指令重排是2和3的执行顺序不确定)，当一个线程执行完1和3，此时已有引用指向该对象，即singleton!=null，但还未来得及执行指令2时，由于系统调度此时执行线程2，那么此时因singleton!=null，会直接返回singleton，而此时属性值为空，因还未来得及初始化。
那么要如何解决这个问题呢?
对变量添加volatile关键字，一是保证了可见性；而是禁止指令重排，保证在返回对象之前一定初始化对象完毕。

数据结构尽量保证空间复杂度最小，算法尽量保证时间复杂度最小。