**21、描述一下JVM加载class文件的原理机制？ 参考 第9题**  
答：

JVM中类的装载是由类加载器（ClassLoader）和它的子类来实现的，Java中的类加载器是一个重要的Java运行时系统组件，它负责在运行时查找和装入类文件中的类。  
由于Java的跨平台性，经过编译的Java源程序并不是一个可执行程序，而是一个或多个类文件。当Java程序需要使用某个类时，JVM会确保这个类已经被加载、连接（验证、准备和解析）和初始化。类的加载是指把类的.class文件中的数据读入到内存中，通常是创建一个字节数组读入.class文件，然后产生与所加载类对应的Class对象。加载完成后，Class对象还不完整，所以此时的类还不可用。当类被加载后就进入连接阶段，这一阶段包括验证、准备（为静态变量分配内存并设置默认的初始值）和解析（将符号引用替换为直接引用）三个步骤。最后JVM对类进行初始化，包括：1)如果类存在直接的父类并且这个类还没有被初始化，那么就先初始化父类；2)如果类中存在初始化语句，就依次执行这些初始化语句。  
类的加载是由类加载器完成的，类加载器包括：根加载器（BootStrap）、扩展加载器（Extension）、系统加载器（System）和用户自定义类加载器（java.lang.ClassLoader的子类）。从Java 2（JDK 1.2）开始，类加载过程采取了父亲委托机制（PDM）。PDM更好的保证了Java平台的安全性，在该机制中，JVM自带的Bootstrap是根加载器，其他的加载器都有且仅有一个父类加载器。类的加载首先请求父类加载器加载，父类加载器无能为力时才由其子类加载器自行加载。JVM不会向Java程序提供对Bootstrap的引用。下面是关于几个类加载器的说明：

* *Bootstrap：一般用本地代码实现，负责加载JVM基础核心类库（rt.jar）；*
* *Extension：从java.ext.dirs系统属性所指定的目录中加载类库，它的父加载器是Bootstrap；*
* *System：又叫应用类加载器，其父类是Extension。它是应用最广泛的类加载器。它从环境变量classpath或者系统属性java.class.path所指定的目录中记载类，是用户自定义加载器的默认父加载器。*

 Java中的所有类，都需要由类加载器装载到JVM中才能运行。类加载器本身也是一个类，而它的工作就是把class文件从硬盘读取到内存中。在写程序的时候，我们几乎不需要关心类的加载，因为这些都是隐式装载的，除非我们有特殊的用法，像是反射，就需要显式的加载所需要的类。

　　类装载方式，有两种   
    　　1.隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中，  
    　　2.显式装载， 通过class.forname()等方法，显式加载需要的类   
  　　隐式加载与显式加载的区别：两者本质是一样?

     Java类的加载是动态的，它并不会一次性将所有类全部加载后再运行，而是保证程序运行的基础类(像是基类)完全加载到jvm中，至于其他类，则在需要的时候才加载。这当然就是为了节省内存开销。

  　Java的类加载器有三个，对应Java的三种类:（java中的类大致分为三种：   1.系统类   2.扩展类 3.由程序员自定义的类 ）

     Bootstrap Loader  // 负责加载**系统类** (指的是内置类，像是String，对应于C#中的System类和C/C++标准库中的类)  
            |   
          - - ExtClassLoader   // 负责加载**扩展类**(就是继承类和实现类)  
                          |   
                      - - AppClassLoader   // 负责加载应用类(**程序员自定义的类**)

 三个加载器各自完成自己的工作，但它们是如何协调工作呢？哪一个类该由哪个类加载器完成呢？为了解决这个问题，Java采用了委托模型机制。

委托模型机制的工作原理很简单：当类加载器需要加载类的时候，先请示其Parent(即上一层加载器)在其搜索路径载入，如果找不到，才在自己的搜索路径搜索该类。这样的顺序其实就是加载器层次上自顶而下的搜索，因为加载器必须保证基础类的加载。之所以是这种机制，还有一个安全上的考虑：如果某人将一个恶意的基础类加载到jvm，委托模型机制会搜索其父类加载器，显然是不可能找到的，自然就不会将该类加载进来。

      我们可以通过这样的代码来获取类加载器:

ClassLoader loader = ClassName.class.getClassLoader();

ClassLoader ParentLoader = loader.getParent();

注意一个很重要的问题，就是Java在逻辑上并不存在BootstrapKLoader的实体！因为它是用C++编写的，所以打印其内容将会得到null。

前面是对类加载器的简单介绍，它的原理机制非常简单，就是下面几个步骤:

1.装载:查找和导入class文件;

2.连接:

      (1)检查:检查载入的class文件数据的正确性;

      (2)准备:为类的静态变量分配存储空间;

      (3)解析:将符号引用转换成直接引用(这一步是可选的)

3.初始化:初始化静态变量，静态代码块。

      这样的过程在程序调用类的静态成员的时候开始执行，所以静态方法main()才会成为一般程序的入口方法。类的构造器也会引发该动作。

**22、char 型变量中能不能存贮一个中文汉字，为什么？**

在计算机内，定点数有3种表示法：原码、反码和补码

[原码]二进制定点表示法，即最高位为符号位，“0”表示正，“1”表示负，其余位表示数值的大小。

[反码]表示法规定：正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外。

[补码]表示法规定：正数的补码与其原码相同；负数的补码是在其反码的末位加1。

计算机数据是用补码表示的，正数的补码就是它的原码，而负数的补码是将其原码（取绝对值后的原码）取反加1得到的。

  
答：char类型可以存储一个中文汉字，因为Java中使用的编码是Unicode（不选择任何特定的编码，直接使用字符在字符集中的编号，这是统一的唯一方法），一个char类型占2个字节（16比特），所以放一个中文是没问题的。

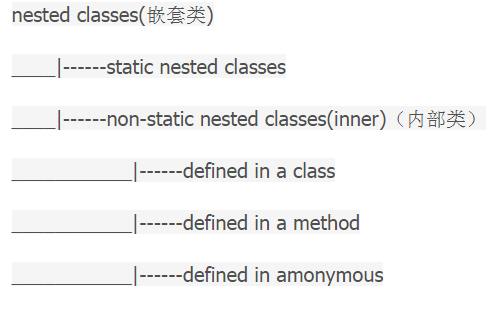
***补充：****使用Unicode意味着字符在JVM内部和外部有不同的表现形式，在JVM内部都是Unicode，当这个字符被从JVM内部转移到外部时（例如存入文件系统中），需要进行编码转换。所以Java中有字节流和字符流，以及在字符流和字节流之间进行转换的转换流，如InputStreamReader和OutputStreamReader，这两个类是字节流和字符流之间的适配器类，承担了编码转换的任务；对于C程序员来说，要完成这样的编码转换恐怕要依赖于union（联合体/共用体）共享内存的特征来实现了。*

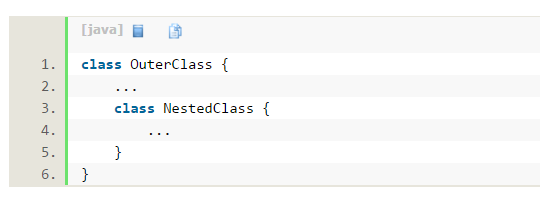
**23、抽象类（abstract class）和接口（interface）有什么异同？**  
答：抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是private、默认、protected、public的，而接口中的成员全都是public的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

**24、静态嵌套类(Static Nested Class)和内部类（Inner Class）的不同？**

为什么要使用内部类？

1.封装性更好，代码可读性高，方便维护。

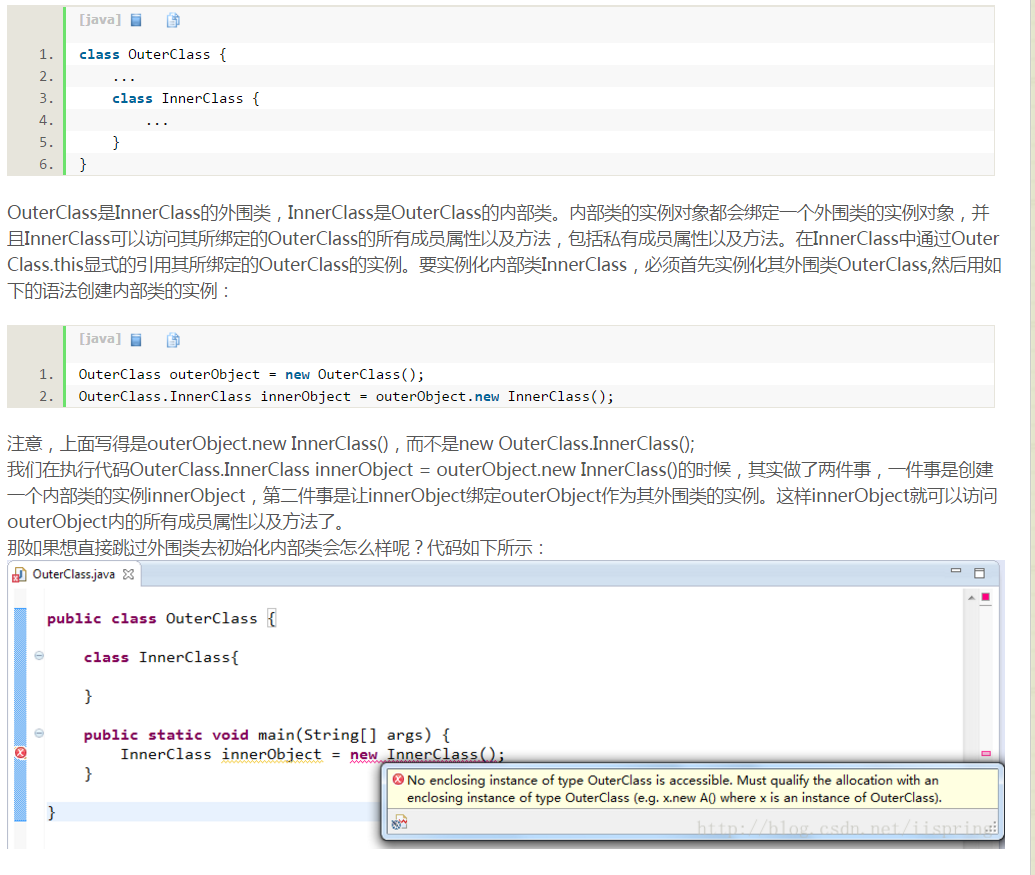
  
答： 内部类实例化时，必须先实例化外部类。内部类的实例化对象需要绑定一个外围类的实例化对象，而静态嵌套类的实例化对象不能也无法绑定外围类的实例化对象。



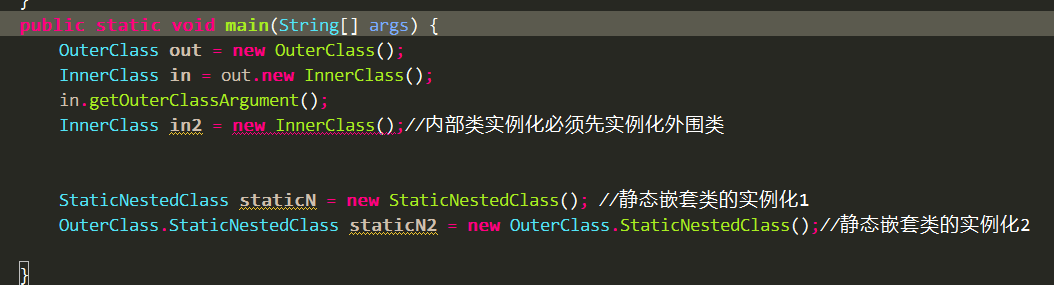
那么在上面的例子中我们称OuterClass为外围类（enclosing class），里面的那个类称之为嵌套类（Nested Class）.

**嵌套类可以分为两种，静态的和非静态的，即静态嵌套类和非静态嵌套类。非静态嵌套类又叫做内部类（Inner Class）。我们通常所说的静态内部类其实是不严格的，严格的说应该叫做静态嵌套类(Static Nested Class)。**

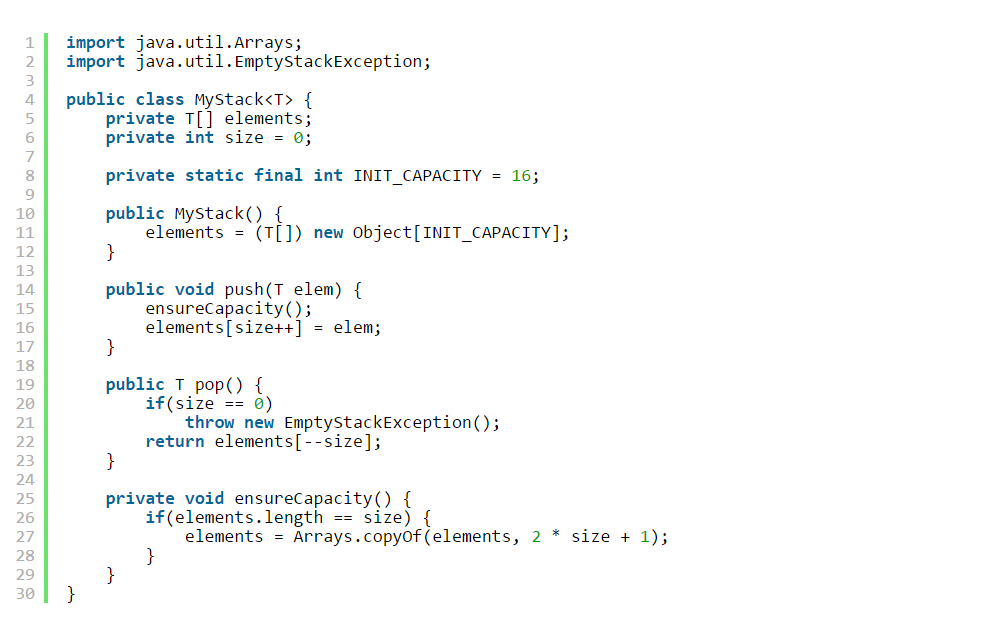
虽然内部类和静态嵌套类都属于嵌套类，但是二者有本质区别：内部类的实例化对象需要绑定一个外围类的实例化对象，而静态嵌套类的实例化对象不能也无法绑定外围类的实例化对象。







**25、Java 中会存在内存泄漏吗，请简单描述。**  
答：理论上Java因为有垃圾回收机制（GC）不会存在内存泄露问题（这也是Java被广泛使用于服务器端编程的一个重要原因）；然而在实际开发中，可能会存在无用但可达的对象，这些对象不能被GC回收，因此也会导致内存泄露的发生。例如hibernate的Session（一级缓存）中的对象属于持久态，垃圾回收器是不会回收这些对象的，然而这些对象中可能存在无用的垃圾对象，如果不及时关闭（close）或清空（flush）一级缓存就可能导致内存泄露。下面例子中的代码也会导致内存泄露。



上面的代码实现了一个栈（先进后出（FILO））结构，乍看之下似乎没有什么明显的问题，它甚至可以通过你编写的各种单元测试。然而其中的pop方法却存在内存泄露的问题，当我们用pop方法弹出栈中的对象时，该对象不会被当作垃圾回收，即使使用栈的程序不再引用这些对象，因为栈内部维护着对这些对象的过期引用（obsolete reference）。在支持垃圾回收的语言中，内存泄露是很隐蔽的，这种内存泄露其实就是无意识的对象保持。如果一个对象引用被无意识的保留起来了，那么垃圾回收器不会处理这个对象，也不会处理该对象引用的其他对象，即使这样的对象只有少数几个，也可能会导致很多的对象被排除在垃圾回收之外，从而对性能造成重大影响，极端情况下会引发Disk Paging（物理内存与硬盘的虚拟内存交换数据），甚至造成OutOfMemoryError。

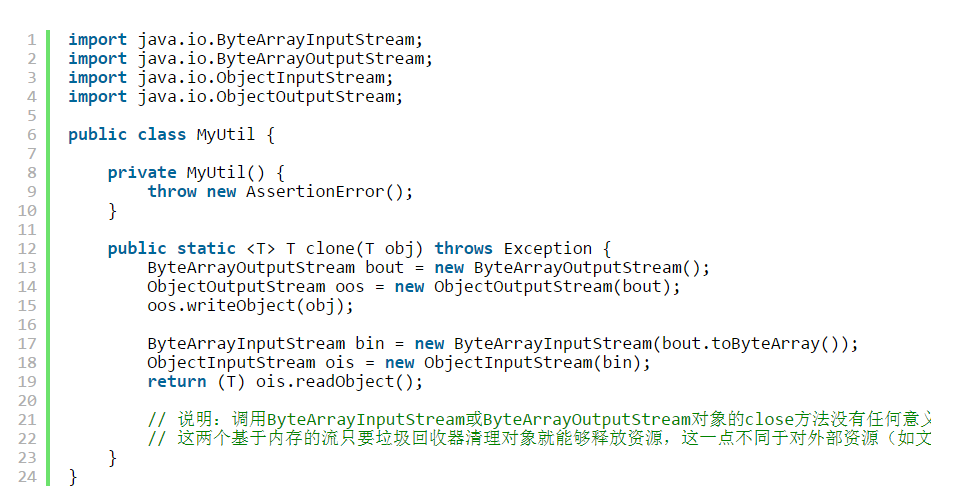
**26、抽象的（abstract）方法是否可同时是静态的（static）,是否可同时是本地方法（native），是否可同时被synchronized修饰？**  
答：都不能。抽象方法需要子类重写，而静态的方法是无法被重写的，因此二者是矛盾的。本地方法是由本地代码（如C代码）实现的方法，而抽象方法是没有实现的，也是矛盾的。synchronized和方法的实现细节有关，抽象方法不涉及实现细节，因此也是相互矛盾的。

**27、阐述静态变量和实例变量的区别。**  
答：静态变量是被static修饰符修饰的变量，也称为类变量，它属于类，不属于类的任何一个对象，一个类不管创建多少个对象，静态变量在内存中有且仅有一个拷贝；实例变量必须依存于某一实例，需要先创建对象然后通过对象才能访问到它。静态变量可以实现让多个对象共享内存。

***补充：****在Java开发中，上下文类和工具类中通常会有大量的静态成员。*

**28、是否可以从一个静态（static）方法内部发出对非静态（non-static）方法的调用？**  
答：不可以，静态方法只能访问静态成员，因为非静态方法的调用要先创建对象，在调用静态方法时可能对象并没有被初始化。

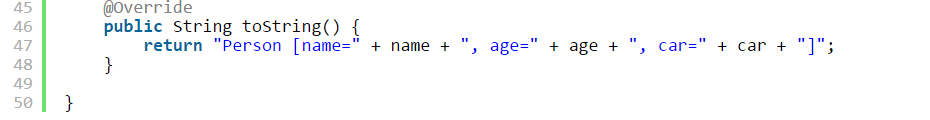
**29、如何实现对象克隆？**  
答：有两种方式：  
1). 实现Cloneable接口并重写Object类中的clone()方法；深克隆的话，必须在重写的clone方法中，克隆那些引用的对象  
2). 实现Serializable接口，通过对象的序列化和反序列化实现克隆，可以实现真正的深度克隆，代码如下。

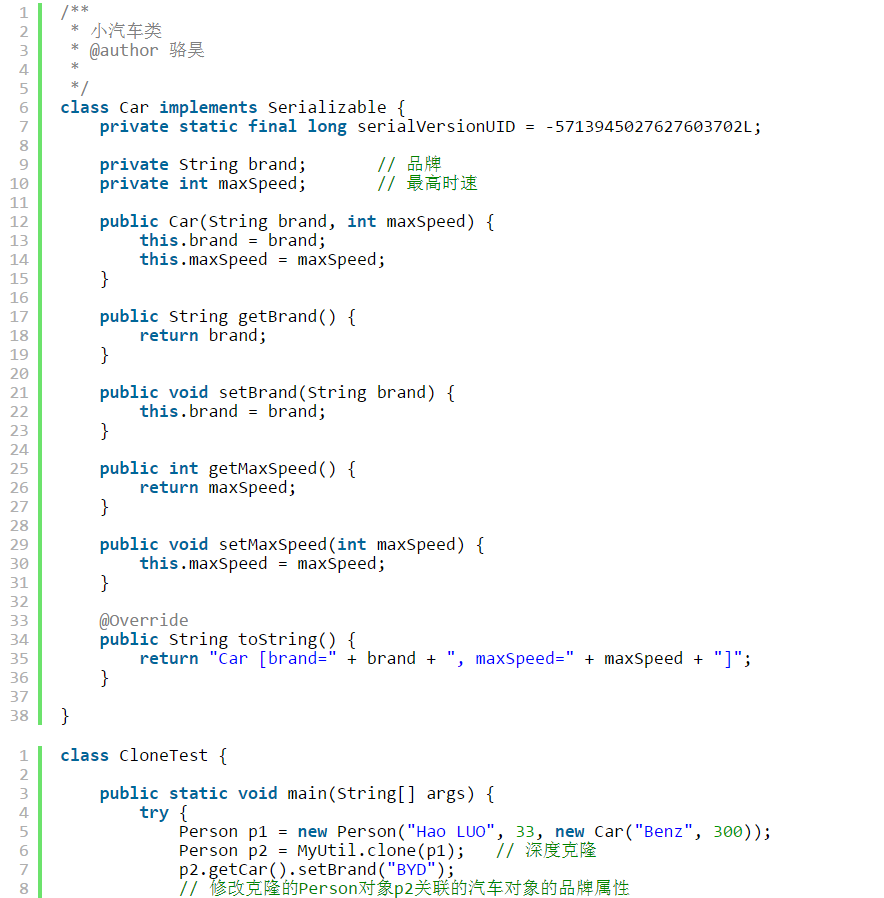


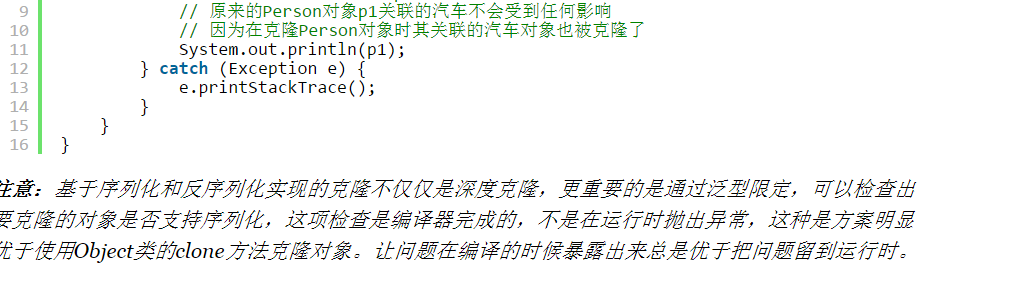
// 说明：调用ByteArrayInputStream或ByteArrayOutputStream对象的close方法没有任何意义

// 这两个基于内存的流只要垃圾回收器清理对象就能够释放资源，这一点不同于对外部资源（如文件流）的释放









30、GC是什么？为什么要有GC？  
答：GC是垃圾收集的意思，内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，

1.Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。

2.要请求垃圾收集，可以调用下面的方法之一：System.gc() 或Runtime.getRuntime().gc() ，但JVM可以屏蔽掉显示的垃圾回收调用。  
3.垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。

4.垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

在Java诞生初期，垃圾回收是Java最大的亮点之一，因为服务器端的编程需要有效的防止内存泄露问题，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于Android系统中垃圾回收的不可预知性。

***补充：****垃圾回收机制有很多种，包括：分代复制垃圾回收、标记垃圾回收、增量垃圾回收等方式。标准的Java进程既有栈又有堆。栈保存了原始型局部变量，堆保存了要创建的对象。Java平台对堆内存回收和再利用的基本算法被称为标记和清除，但是Java对其进行了改进，采用“分代式垃圾收集”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：  
- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。  
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。  
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。*

