**ava内存管理与多线程**

**1. 什么是线程？什么是进程？同一进程下的线程共享**

**线程：**程序在执行过程中，能够执行程序代码的一个执行单元，一个线程可以创建和撤销另一个线程;同一个进程中的多个线程之间可以并发执行。在[**Java**](http://lib.csdn.net/base/java)语言中有4种状态：运行、就绪、挂起、结束。

**进程：**指一段正在执行的程序。线程有时也被称为轻量级进程，它是程序执行的最小单元，一个进程可以拥有多个线程，各个线程之间共享程序的内存空间及一些进程级的资源，但是各个线程拥有自己的栈空间。

**进程的作用和定义：**进程是为了提高CPU的执行效率，减少因为程序等待带来的CPU空转以及其他计算机软硬件资源的浪费而提出来的。进程是为了完成用户任务所需要的程序的一次执行过程和为其分配资源的一个基本单位，是一个具有独立功能的程序段对某个数据集的一次执行活动。

**线程和进程的区别：**   
A. 地址空间和其它资源：进程间相互独立，同一进程的各线程间共享。某进程内的线程在其它进程不可见。   
B. 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性。   
C. 调度和切换：线程上下文切换比进程上下文切换要快得多。   
D. 在多线程OS中，进程不是一个可执行的实体。   
E. 线程是进程的一部分，所以线程有的时候被称为是轻权进程或者轻量级进程。

**线程与进程资源分配：**   
线程共享的内容包括： 进程代码段、进程的公有数据(利用这些共享的数据，线程很容易的实现相互之间的通讯)、进程打开的文件描述符、信号的处理器、进程的当前目录和、进程用户ID、进程组ID。   
线程独有的内容包括： 线程ID 、寄存器组的值 、线程的堆栈 、错误返回码 、线程的信号屏蔽码 。

**2. Java的内存机制**

Java 把内存划分成两种：一种是栈内存，另一种是堆内存。   
**堆和栈相同点：**   
栈(stack)与堆(heap)都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。   
**堆和栈区别：**   
栈的优势是，存取速度比堆要快，仅次于直接位于CPU中的寄存器。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。另外，栈数据可以共享。堆的优势是可以动态地分配内存大小，生存期也不必事先告诉编译器，Java的垃圾收集器会自动收走这些不再使用的数据。但缺点是，由于要在运行时动态分配内存，存取速度较慢。

在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配，当在一段代码块定义一个变量时，Java 就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java 会自动释放掉为该变量分配的内存空间，该内存空间可以立即被另作它用。

堆内存用来存放由 new 创建的对象和数组，在堆中分配的内存，由 Java 虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或者对象之后，还可以在栈中定义一个特殊的变量，让栈中的这个变量的取值等于数组或对象在堆内存中的 首地址，栈中的这个变量就成了数组或对象的引用变量，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或者对象，引用变量就相当于是为数组或者对象 起的一个名称。引用变量是普通的变量，定义时在栈中分配，引用变量在程序运行到其作用域之外后被释放。而数组和对象本身在堆中分配，即使程序运行到使用 new 产生数组或者对象的语句所在的代码块之外，数组和对象本身占据的内存不会被释放，数组和对象在没有引用变量指向它的时候，才变为垃圾，不能在被使用，但仍 然占据内存空间不放，在随后的一个不确定的时间被垃圾回收器收走（释放掉）。

**3. java中变量在内存中的分配**

**（1）类变量（static修饰的变量）**：在程序加载时系统就为它在堆中开辟了内存，堆中的内存地址存放于栈以便于高速访问。静态变量的生命周期–一直持续到整个”系统”关闭

**（2）实例变量：**当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量（比如说类实例），然后根据零散的堆内存地址，通过哈希[**算法**](http://lib.csdn.net/base/datastructure)换算为一长串数字以表征这个变量在堆中的”物理位置”。 实例变量的生命周期–当实例变量的引用丢失后，将被GC（垃圾回收器）列入可回收“名单”中，但并不是马上就释放堆中内存

**（3）局部变量：**局部变量，由声明在某方法，或某代码段里（比如for循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放

**4. JVM内存分配**   
JVM 将内存区域划分为： Method Are（Non-Heap）（方法区）,Heap（堆）,Program Counter Register（程序计数器）,VM Stack（虚拟机栈，也有翻译成JAVA 方法栈的）,Native Method Stack（本地方法栈）。   
**方法区和堆是线程共享的，虚拟机栈，程序计数器和本地方法栈是非线程共享的。**   
一般性的 Java 程序的工作过程：一个 Java 源程序文件，会被编译为字节码文件（以 class 为扩展名），每个java程序都需要运行在自己的JVM上，然后告知 JVM 程序的运行入口，再被 JVM 通过字节码解释器加载运行。那么程序开始运行后，都是如何涉及到各内存区域的呢？   
概括地说来，JVM初始运行的时候都会分配好方法区和堆，而JVM每遇到一个线程，就为其分配一个程序计数器,虚拟机栈和本地方法栈，当线程终止时，三者（虚拟机栈，本地方法栈和程序计数器）所占用的内存空间也会被释放掉。这也是为什么我把内存区域分为线程共享和非线程共享的原因，非线程共享的那三个区域的生命周期与所属线程相同，而线程共享的区域与JAVA程序运行的生命周期相同，所以这也是系统垃圾回收的场所只发生在线程共享的区域（实际上对大部分虚拟机来说知发生在Heap上）的原因。

**5. Java实现多线程，创建并启动线程的过程**

**（1）定义线程：**   
1）扩展java.lang.Thread类。 2）实现java.lang.Runnable接口。

**（2）实例化线程：**   
1）如果是扩展java.lang.Thread类的线程，则直接new即可。   
2）如果是实现了java.lang.Runnable接口的类，则用Thread的构造方法：   
Thread(Runnable target)   
Thread(Runnable target, String name)   
Thread(ThreadGroup group, Runnable target)   
Thread(ThreadGroup group, Runnable target, String name)   
Thread(ThreadGroup group, Runnable target, String name, long stackSize)

**（3）启动线程：**在线程的Thread对象上调用start()方法，而不是run()或者别的方法。

**通过继承Thread类创建线程：**   
public class ThreadDemo02 extends Thread{   
public void run(){   
System.out.println(“线程启动！”);   
}   
public static void main(String[] args) {   
ThreadDemo02 thread = new ThreadDemo02();   
thread.start();   
}   
}   
**通过实现Runnable接口创建线程：**   
public class ThreadDemo03 implements Runnable{   
public void run() {   
System.out.println(“线程启动02”);   
}   
public static void main(String[] args) {   
Thread thread01 = new Thread(new ThreadDemo03());   
thread01.start();   
}   
}   
**start()方法和run()方法的区别：**   
（1）启动一个线程是start()方法，启动线程之后start()方法会去调用run方法内容。   
（2）start是创建并启动一个线程，而run是要运行线程中的代码。   
（3）run()方法 : 在本线程内调用该Runnable对象的run()方法，可以重复多次调用；   
start()方法 : 启动一个线程，调用该Runnable对象的run()方法，不能多次启动一个线程；

**7. 在JAVA中，有六个不同的地方可以存储数据**

**（1） 寄存器（register）**：这是最快的存储区，因为它位于不同于其他存储区的地方——处理器内部。但是寄存器的数量极其有限，所以寄存器由编译器根据需求进行分配。你不能直接控制，也不能在程序中感觉到寄存器存在的任何迹象。   
**（2） 堆栈（stack）**：位于通用RAM中，但通过它的“堆栈指针”可以从处理器哪里获得支持。堆栈指针若向下移动，则分配新的内存；若向上移动，则释放那些内存。这是一种快速有效的分配存储方法，仅次于寄存器。创建程序时候，JAVA编译器必须知道存储在堆栈内所有数据的确切大小和生命周期，因为它必须生成相应的代码，以便上下移动堆栈指针。这一约束限制了程序的灵活性，所以虽然某些JAVA数据存储在堆栈中——特别是对象引用，但是JAVA对象不存储其中。   
**（3）堆（heap）**：一种通用性的内存池（也存在于RAM中），用于存放所以的JAVA对象。堆不同于堆栈的好处是：编译器不需要知道要从堆里分配多少存储区域，也不必知道存储的数据在堆里存活多长时间。因此，在堆里分配存储有很大的灵活性。当你需要创建一个对象的时候，只需要new写一行简单的代码，当执行这行代码时，会自动在堆里进行存储分配。当然，为这种灵活性必须要付出相应的代码。用堆进行存储分配比用堆栈进行存储存储需要更多的时间。   
**（4）静态存储（static storage）**：这里的“静态”是指“在固定的位置”。静态存储里存放程序运行时一直存在的数据。你可用关键字static来标识一个对象的特定元素是静态的，但JAVA对象本身从来不会存放在静态存储空间里。   
**（5）常量存储（constant storage）**：常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会被改变。有时，在[**嵌入式**](http://lib.csdn.net/base/embeddeddevelopment)系统中，常量本身会和其他部分分割离开，所以在这种情况下，可以选择将其放在ROM中   
**（6）非RAM存储**：如果数据完全存活于程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在。

**8.**[**Java**](http://lib.csdn.net/base/java)**concurrent包下的4个类，选出差别最大的一个 (C)**   
A. Semaphore B. ReentrantLock C. Future D. CountDownLatch   
别的类都处理线程间的关系，处理并发机制，但Future 只用于获取线程结果。   
Future是个接口，表示获取一个正在指定的线程的结果。对该线程有取消和判断是否执行完毕等操作。   
CountDownLatch 是个锁存器，他表示我要占用给定的多少个线程且我优先执行，我执行完之前其他要使用该资源的都要等待。   
Semaphore，就像是一个许可证发放者，也像一个[**数据库**](http://lib.csdn.net/base/mysql)连接池。证就这么多，如果池中的证没换回来，其他人就不能用。   
ReentrantLock 和 synchronized一样，用于锁定线程。

[**Java**](http://lib.csdn.net/base/java)**垃圾收集器与类的finalize()方法总结**

* 1． 垃圾收集器的工作目标是回收已经无用的对象的内存空间，从而避免内存渗漏体的产生，节省内存资源，避免程序代码的崩溃。
* 2．垃圾收集器判断一个对象的内存空间是否无用的标准是：如果该对象不能再被程序中任何一个”活动的部分”所引用，此时我们就说，该对象的内存空间已经无用。所谓”活动的部分”，是指程序中某部分参与程序的调用，正在执行过程中，尚未执行完毕。
* 3．垃圾收集器线程虽然是作为低优先级的线程运行，但在系统可用内存量过低的时候，它可能会突发地执行来挽救内存资源。当然其执行与否也是不可预知的。
* 4．垃圾收集器不可以被强制执行，但程序员可以通过调用System. gc方法来建议执行垃圾收集器。
* 5．不能保证一个无用的对象一定会被垃圾收集器收集，也不能保证垃圾收集器在一段Java语言代码中一定会执行。因此在程序执行过程中被分配出去的内存空间可能会一直保留到该程序执行完毕，除非该空间被重新分配或被其他方法回收。由此可见，完全彻底地根绝内存渗漏体的产生也是不可能的。但是请不要忘记，Java的垃圾收集器毕竟使程序员从手工回收内存空间的繁重工作中解脱了出来。设想一个程序员要用C或C++来编写一段10万行语句的代码，那么他一定会充分体会到Java的垃圾收集器的优点！
* 6．同样没有办法预知在一组均符合垃圾收集器收集标准的对象中，哪一个会被首先收集。
* 7．循环引用对象不会影响其被垃圾收集器收集。
* 8．可以通过将对象的引用变量（reference variables，即句柄handles）初始化为null值，来暗示垃圾收集器来收集该对象。但此时，如果该对象连接有事件监听器（典型的AWT组件），那它还是不可以被收集。所以在设一个引用变量为null值之前，应注意该引用变量指向的对象是否被监听，若有，要首先除去监听器，然后才可以赋空值。
* 9．每一个对象都有一个finalize( )方法，这个方法是从Object类继承来的。
* 10．finalize()方法用来回收内存以外的系统资源，就像是文件处理器和网络连接器。该方法的调用顺序和用来调用该方法的对象的创建顺序是无关的。换句话说，书写程序时该方法的顺序和方法的实际调用顺序是不相干的。请注意这只是finalize()方法的特点。
* 11．每个对象只能调用finalize( )方法一次。如果在finalize()方法执行时产生异常（exception），则该对象仍可以被垃圾收集器收集。
* 12．垃圾收集器跟踪每一个对象，收集那些不可到达的对象（即该对象没有被程序的任何”活的部分”所调用），回收其占有的内存空间。但在进行垃圾收集的时候，垃圾收集器会调用finalize()方法，通过让其他对象知道它的存在，而使不可到达的对象再次”复苏”为可到达的对象。既然每个对象只能调用一次finalize()方法，所以每个对象也只可能”复苏”一次。
* 13．finalize( )方法可以明确地被调用，但它却不能进行垃圾收集。
* 14．finalize( )方法可以被重载（overload），但只有具备初始的finalize( )方法特点的方法才可以被垃圾收集器调用。
* 15．子类的finalize( )方法可以明确地调用父类的finalize()方法，作为该子类对象的最后一次适当的操作。但Java编译器却不认为这是一次覆盖操作（overriding），所以也不会对其调用进行检查。
* 16．当finalize( )方法尚未被调用时，System. runFinalization( )方法可以用来调用finalize()方法，并实现相同的效果，对无用对象进行垃圾收集。
* 17．当一个方法执行完毕，其中的局部变量就会超出使用范围，此时可以被当作垃圾收集，但以后每当该方法再次被调用时，其中的局部变量便会被重新创建。
* 18．Java语言使用了一种”标记交换区的垃圾收集算法”。该算法会遍历程序中每一个对象的句柄，为被引用的对象做标记，然后回收尚未做标记的对象。所谓遍历可以简单地理解为”检查每一个”。
* 19．Java语言允许程序员为任何方法添加finalize(   
  )方法，该方法会在垃圾收集器交换回收对象之前被调用。但不要过分依赖该方法对系统资源进行回收和再利用，因为该方法调用后的执行结果是不可预知的。通过以上对垃圾收集器特点的了解，你应该可以明确垃圾收集器的作用，和垃圾收集器判断一块内存空间是否无用的标准。简单地说，当你为一个对象赋值为null并且重新定向了该对象的引用者，此时该对象就符合垃圾收集器的收集标准。

**内存管理:**在java中内存分配是由关键字new申请，有GC(垃圾收集器)负责释放，GC将程序中对象引用当作一个有向图，当某个对象不在有向图内时，该对象将被释放回收，注意遍历图是需要消耗资源，当GC运行频率过大，将导致性能严重下降。

**典型内存泄漏原因：**   
1.全局集合 2.缓存 3.ClassLoader   
内存泄漏标志是OutOfMemoryError，产生该信号有2个原因：1.确实是内存不够用，可以通过增加JVM堆解决 2.由内存泄漏造成

**Clone**   
1.Object类的clone方法是protected,如果想通过调用该方法实现克隆，该类需要实现Cloneable接口，否则将抛出CloneNotSupportedException,因为，javac需要程序显示指定能够执行Clone方法，该方法默认实现是浅克隆   
2.覆盖Object类中的clone方法

**一、内存管理原理**

　　在java中，有java程序、虚拟机、操作系统三个层次，其中java程序与虚拟机交互，而虚拟机与操作系统间交互！这就保证了java程序的平台无关性！下面我们从程序运行前，程序运行中、程序运行内存溢出三个阶段来说一下内存管理原理！

　　1、程序运行前：JVM向操作系统请求一定的内存空间，称为初始内存空间！程序执行过程中所需的内存都是由java虚拟机从这片内存空间中划分的。

 　  2、程序运行中：java程序一直向java虚拟机申请内存，当程序所需要的内存空间超出初始内存空间时，java虚拟机会再次向操作系统申请更多的内存供程序使用！

　　3、内存溢出：程序接着运行，当java虚拟机已申请的内存达到了规定的最大内存空间，但程序还需要更多的内存，这时会出现内存溢出的错误！

　　至此可以看出，Java 程序所使用的内存是由 Java 虚拟机进行管理、分配的。Java 虚拟机规定了 Java 程序的初始内存空间和最大内存空间，开发者只需要关心 Java 虚拟机是如何管理内存空间的，而不用关心某一种操作系统是如何管理内存的。

**二、内存空间逻辑划分**

　　JVM 会把申请的内存从逻辑上划分为三个区域，即：方法区、堆与栈。

**方法区**：方法区默认最大容量为64M，Java虚拟机会将加载的java类存入方法区，保存类的结构（属性与方法），类静态成员等内容。

**堆**：默认最大容量为64M，堆存放对象持有的数据，同时保持对原类的引用。可以简单的理解为对象属性的值保存在堆中，对象调用的方法保存在方法区。

**栈（虚拟机栈）**：栈默认最大容量为1M，在程序运行时，每当遇到方法调用时，Java虚拟机就会在栈中划分一块内存称为栈帧（Stack frame），栈帧中的内存供局部变量（包括基本类型与引用类型）使用，当方法调用结束后，Java虚拟机会收回此栈帧占用的内存。

**运行时常量池**：是方法区的一部分。用于存放编译期生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法去的运行时常量池中。

**程序计数器**是一块较小的内存空间，它的作用可以看做是当前线程所执行的字节码的行号指示器。在虚拟机的概念模型里（仅是概念模型，各种虚拟机可能会通过一些更高效的方式去实现），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。如果线程正在执行的是一个Java方法，这个计数器

记录的是正在执行的虚拟机字节码指令的地址；如果正在执行的是Natvie方法，这个计数器值则为空（Undefined）。**此内存区域是唯一一个在Java虚拟机规范中没有规定任何OutOfMemoryError情况的区域。**

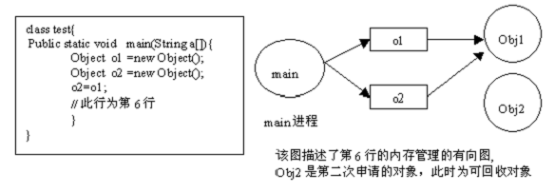
**本地方法栈**（Native Method Stacks）与虚拟机栈所发挥的作用是非常相似的，其区别不过是虚拟机栈为虚拟机执行Java方法（也就是字节码）服务，而本地方法栈则是为虚拟机使用到的Native方法服务。虚拟机规范中对本地方法栈中的方法使用的语言、使用方式与数据结构并没有强制规定，因此具体的虚拟机可以自由实现它

**Java内存管理机制**

在C++ 语言中，如果需要动态分配一块内存，程序员需要负责这块内存的整个生命周期。从申请分配、到使用、再到最后的释放。这样的过程非常灵活，但是却十分繁琐，程序员很容易由于疏忽而忘记释放内存，从而导致内存的泄露。 Java 语言对内存管理做了自己的优化，这就是垃圾回收机制。 Java 的几乎所有内存对象都是在堆内存上分配（基本数据类型除外），然后由 GC （ garbage collection）负责自动回收不再使用的内存。

上面是Java 内存管理机制的基本情况。但是如果仅仅理解到这里，我们在实际的项目开发中仍然会遇到内存泄漏的问题。也许有人表示怀疑，既然 Java 的垃圾回收机制能够自动的回收内存，怎么还会出现内存泄漏的情况呢？这个问题，我们需要知道 GC 在什么时候回收内存对象，什么样的内存对象会被 GC 认为是“不再使用”的。

Java中对内存对象的访问，使用的是引用的方式。在 Java 代码中我们维护一个内存对象的引用变量，通过这个引用变量的值，我们可以访问到对应的内存地址中的内存对象空间。在 Java 程序中，这个引用变量本身既可以存放堆内存中，又可以放在代码栈的内存中（与基本数据类型相同）。 GC 线程会从代码栈中的引用变量开始跟踪，从而判定哪些内存是正在使用的。如果 GC 线程通过这种方式，无法跟踪到某一块堆内存，那么 GC 就认为这块内存将不再使用了（因为代码中已经无法访问这块内存了）。



通过这种有向图的内存管理方式，当一个内存对象失去了所有的引用之后，GC 就可以将其回收。反过来说，如果这个对象还存在引用，那么它将不会被 GC 回收，哪怕是 Java 虚拟机抛出 OutOfMemoryError 。

**Java内存泄露**

一般来说内存泄漏有两种情况。一种情况如在C/C++ 语言中的，在堆中的分配的内存，在没有将其释放掉的时候，就将所有能访问这块内存的方式都删掉（如指针重新赋值）；另一种情况则是在内存对象明明已经不需要的时候，还仍然保留着这块内存和它的访问方式（引用）。第一种情况，在 Java 中已经由于垃圾回收机制的引入，得到了很好的解决。所以， Java 中的内存泄漏，主要指的是第二种情况。

可能光说概念太抽象了，大家可以看一下这样的例子：

Vector v = **new** Vector( 10 );

**for** ( **int** i = 1 ;i < 100 ; i ++ ){

Object o = **new** Object();

v.**add**(o);

o = null ;

}

在这个例子中，代码栈中存在Vector 对象的引用 v 和 Object 对象的引用 o 。在 For 循环中，我们不断的生成新的对象，然后将其添加到 Vector 对象中，之后将 o 引用置空。问题是当 o 引用被置空后，如果发生 GC ，我们创建的 Object 对象是否能够被 GC 回收呢？答案是否定的。因为， GC 在跟踪代码栈中的引用时，会发现 v 引用，而继续往下跟踪，就会发现 v 引用指向的内存空间中又存在指向 Object 对象的引用。也就是说尽管 o 引用已经被置空，但是 Object 对象仍然存在其他的引用，是可以被访问到的，所以 GC 无法将其释放掉。如果在此循环之后， Object 对象对程序已经没有任何作用，那么我们就认为此 Java 程序发生了内存泄漏。

尽管对于C/C++ 中的内存泄露情况来说， Java 内存泄露导致的破坏性小，除了少数情况会出现程序崩溃的情况外，大多数情况下程序仍然能正常运行。但是，在移动设备对于内存和 CPU都有较严格的限制的情况下， Java 的内存溢出会导致程序效率低下、占用大量不需要的内存等问题。这将导致整个机器性能变差，严重的也会引起抛出 OutOfMemoryError ，导致程序崩溃。

**一般情况下内存泄漏的避免**

在不涉及复杂数据结构的一般情况下，Java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度。我们有时也将其称为“对象游离”。

例如：

**public** **class** **FileSearch**{

**private** **byte** [] content;

**private** File mFile;

**public** **FileSearch**(File file){

mFile = file;

}

**public** **boolean** **hasString**(String str){

**int** size = getFileSize(mFile);

content = **new** **byte** [size];

loadFile(mFile, content);

String s = **new** String(content);

**return** s.contains(str);

}

}

在这段代码中，FileSearch 类中有一个函数 hasString ，用来判断文档中是否含有指定的字符串。流程是先将mFile 加载到内存中，然后进行判断。但是，这里的问题是，将 content 声明为了实例变量，而不是本地变量。于是，在此函数返回之后，内存中仍然存在整个文件的数据。而很明显，这些数据我们后续是不再需要的，这就造成了内存的无故浪费。

要避免这种情况下的内存泄露，要求我们以C/C++ 的内存管理思维来管理自己分配的内存。第一，是在声明对象引用之前，明确内存对象的有效作用域。在一个函数内有效的内存对象，应该声明为 local 变量，与类实例生命周期相同的要声明为实例变量……以此类推。第二，在内存对象不再需要时，记得手动将其引用置空。

**复杂数据结构中的内存泄露问题**

在实际的项目中，我们经常用到一些较为复杂的数据结构用于缓存程序运行过程中需要的数据信息。有时，由于数据结构过于复杂，或者我们存在一些特殊的需求（例如，在内存允许的情况下，尽可能多的缓存信息来提高程序的运行速度等情况），我们很难对数据结构中数据的生命周期作出明确的界定。这个时候，我们可以使用Java 中一种特殊的机制来达到防止内存泄露的目的。

之前我们介绍过，Java 的 GC 机制是建立在跟踪内存的引用机制上的。而在此之前，我们所使用的引用都只是定义一个“ Object o; ”这样形式的。事实上，这只是 Java 引用机制中的一种默认情况，除此之外，还有其他的一些引用方式。通过使用这些特殊的引用机制，配合 GC 机制，就可以达到一些我们需要的效果。

**Java中的几种引用方式**

Java中有几种不同的引用方式，它们分别是：强引用、软引用、弱引用和虚引用。下面，我们首先详细地了解下这几种引用方式的意义。

**强引用**

在此之前我们介绍的内容中所使用的引用 都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必不可少的生活用品，垃圾回收器绝不会回收它。当内存空 间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题。

**软引用（SoftReference ）**

SoftReference 类的一个典型用途就是用于内存敏感的高速缓存。 SoftReference  的原理是：在保持对对象的引用时保证在  JVM  报告内存不足情况之前将清除所有的软引用。关键之处在于，垃圾收集器在运行时可能会（也可能不会）释放软可及对象。对象是否被释放取决于垃圾收集器的算法 以及垃圾收集器运行时可用的内存数量。

**弱引用（WeakReference ）**

WeakReference 类的一个典型用途就是规范化映射（ canonicalized mapping ）。另外，对于那些生存期相对较长而且重新创建的开销也不高的对象来说，弱引用也比较有用。关键之处在于，垃圾收集器运行时如果碰到了弱可及对象，将释放  WeakReference  引用的对象。然而，请注意，垃圾收集器可能要运行多次才能找到并释放弱可及对象。

**虚引用（PhantomReference ）**

PhantomReference 类只能用于跟踪对被引用对象即将进行的收集。同样，它还能用于执行  pre-mortem  清除操作。 PhantomReference  必须与  ReferenceQueue  类一起使用。需要  ReferenceQueue  是因为它能够充当通知机制。当垃圾收集器确定了某个对象是虚可及对象时， PhantomReference  对象就被放在它的  ReferenceQueue  上。将  PhantomReference  对象放在  ReferenceQueue  上也就是一个通知，表明  PhantomReference  对象引用的对象已经结束，可供收集了。这使您能够刚好在对象占用的内存被回收之前采取行动。 Reference与 ReferenceQueue 的配合使用。