## [Android内存监控与分析](http://blog.csdn.net/allan_shore_ma/article/details/78340931)

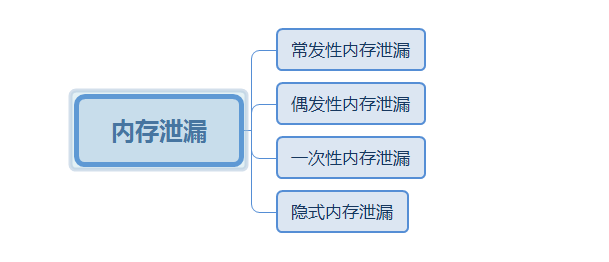
### 内存泄露与内存溢出

#### 内存泄漏

1. 定义

内存泄漏（Memory Leak）是指程序中己动态分配的堆内存由于某种原因程序未释放或无法释放，造成系统内存的浪费，导致程序运行速度减慢甚至系统崩溃等严重后果。

1. 分类



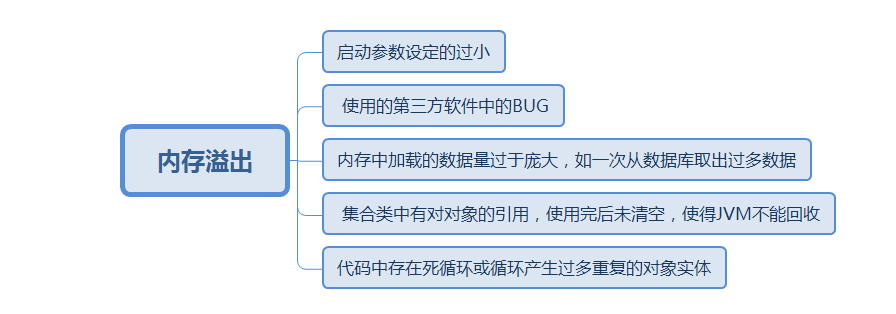
1. 常发性内存泄漏：发生内存泄漏的代码会被多次执行到，每次被执行的时候都会导致一块内存泄漏。
2. 偶发性内存泄漏：发生内存泄漏的代码只有在某些特定环境或操作过程下才会发生。常发性和偶发性是相对的。对于特定的环境，偶发性的也许就变成了常发性的。所以测试环境和测试方法对检测内存泄漏至关重要。
3. 一次性内存泄漏：发生内存泄漏的代码只会被执行一次，或者由于算法上的缺陷，导致总会有一块仅且一块内存发生泄漏。比如，在类的构造函数中分配内存，在析构函数中却没有释放该内存，所以内存泄漏只会发生一次。
4. 隐式内存泄漏：程序在运行过程中不停的分配内存，但是直到结束的时候才释放内存。严格的说这里并没有发生内存泄漏，因为最终程序释放了所有申请的内存。但是对于一个服务器程序，需要运行几天，几周甚至几个月，不及时释放内存也可能导致最终耗尽系统的所有内存。所以，我们称这类内存泄漏为隐式内存泄漏。

#### 内存溢出

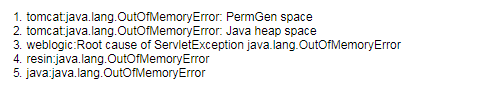
1. 定义

用户在对其数据缓冲区操作时，超过了其缓冲区的边界；尤其是对缓冲区写操作时，缓冲区的溢出很可能导致程序的异常。

1. 分类



常见错误提示：



解决办法：1）增加jvm的内存大小； 2）优化程序，及时地释放没用对象，释放内存空间

### Java内存管理与垃圾回收

谈到android内存分析以及原理，对象的引用与回收，我们必须先了解java内存管理、垃圾回收（GC）机制和Android内存管理，熟悉其内在的联系。

#### Java内存管理机制

Java虚拟机在执行程序时把它管理的内存分为若干数据区域，这些数据区域分布情况如图：



###### 方法区

方法区是各个线程所共享的内存区域，用于存储已被虚拟机加载类信息、常亮、静态变量、即时编译器编译后的代码等数据。

运行时常量池：运行时常量池是方法区的一部分，Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池，用于存放编译期生成的各种字面量和符号引用。

###### 虚拟机栈

Java虚拟机栈：线程私有的，其生命周期和线程一致，每个方法执行时都会创建一个栈帧用于存储局部变量表、操作数栈、动态链接、方法出口等信息。当方法被调用时，栈帧在JVM栈中入栈，当方法执行完成时，栈帧出栈。

局部变量表中存储着方法的相关局部变量，包括各种基本数据类型，对象的引用，返回地址等。需要注意的是局部变量表是在编译时就已经确定好的，方法运行所需要分配的空间在栈帧中是完全确定的，在方法的生命周期内都不会改变。

###### 本地方法栈

与虚拟机栈功能类似，只不过虚拟机栈为虚拟机执行Java方法服务，而本地方法栈则为使用到的Native方法服务。

###### 堆

**虚拟机管理内存中最大的一块，被所有线程共享，该区域用于存放对象实例，几乎所有的对象都在该区域分配。Java堆是内存回收的主要区域，因此很多时候也被称作“GC堆”。**从内存回收角度看，由于现在的收集器大都采用分代收集算法，所以Java堆还可以细分为：新生代和老年代，再细分一点的话可以分为Eden空间、From Survivor空间、To Survivor空间等。根据Java虚拟机规范规定，Java堆可以处于物理上不连续的空间，只要逻辑上是连续的就行。**如果在堆中没有内存完成实例分配，并且也无法再扩展时，将会抛出OutofMemoryError异常。**

###### 程序计数器

一块较小内存区域，指向当前所执行的字节码。如果线程正在执行一个Java方法，这个计数器记录正在执行的虚拟机字节码指令的地址，如果执行的是Native方法，这个计算器值为空。

###### Java对象访问方式

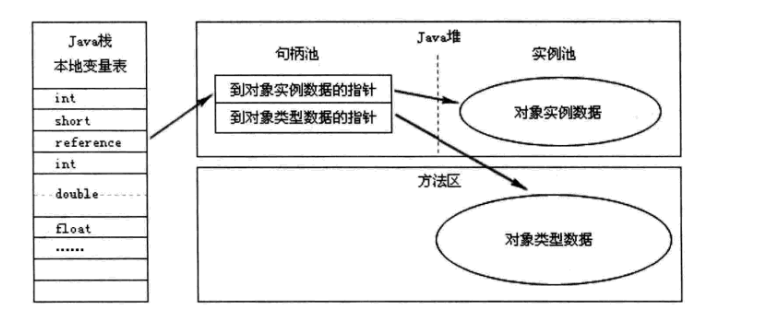
一般来说，一个Java的引用访问涉及到3个内存区域：JVM栈，堆，方法区。以最简单的本地变量引用：Object objRef = new Object()为例：

1. Object objRef 表示一个本地引用，存储在JVM栈的本地变量表中，表示一个reference类型数据；
2. new Object()作为实例对象数据存储在堆中；
3. 堆中还记录了能够查询到此Object对象的类型数据（接口、方法、field、对象类型等）的地址，实际的数据则存储在方法区中；

在Java虚拟机规范中，只规定了指向对象的引用，对于通过reference类型引用访问具体对象的方式并未做规定，不过目前主流的实现方式主要有两种：

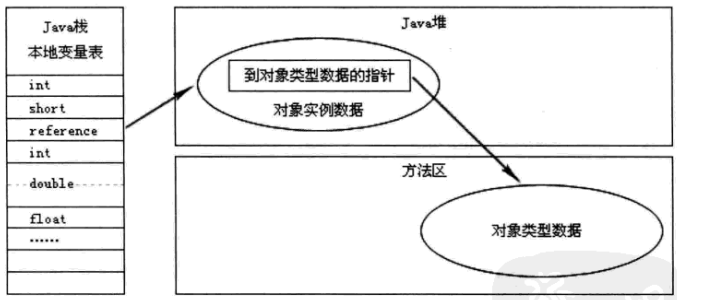
1. **通过句柄访问**

通过句柄访问的实现方式中，JVM堆中会划分单独一块内存区域作为句柄池，句柄池中存储了对象实例数据（在堆中）和对象类型数据（在方法区中）的指针。这种实现方法由于用句柄表示地址，因此十分稳定**。**



1. **通过直接指针访问**

通过直接指针访问的方式中，reference中存储的就是对象在堆中的实际地址，在堆中存储的对象信息中包含了在方法区中的相应类型数据。这种方法最大的优势是速度快，在HotSpot虚拟机中用的就是这种方式。

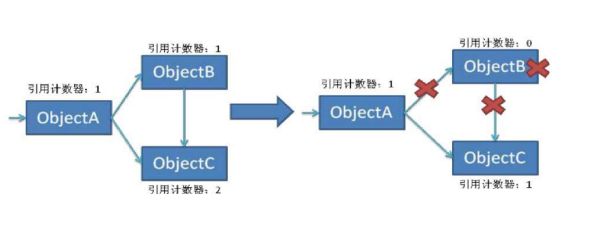


#### 垃圾回收（GC）机制

JVM通过GC来回收堆和方法区中的内存，这个过程是自动执行的。说到Java GC机制，其主要完成3件事：确定哪些内存需要回收；确定什么时候需要执行GC；如何执行GC。JVM主要采用收集器的方式实现GC，主要的收集器有引用计数收集器和跟踪收集器。

###### 引用计数收集器

引用计数器采用分散式管理方式，通过计数器记录对象是否被引用。当计数器为0时，说明此对象已经不再被使用，可进行回收，如图所示：



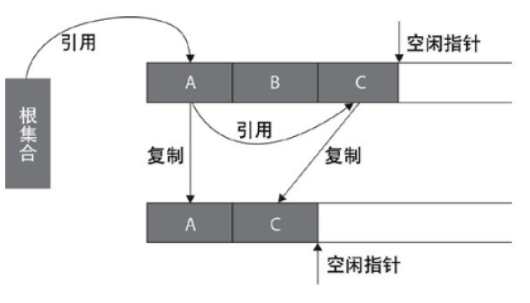
在上图中，ObjectA释放了对ObjectB的引用后，ObjectB的引用计数器变为0，此时可回收ObjectB所占有的内存。

但是引用计数器需要在每次对象赋值时进行引用计数器的增减，有一定消耗。另外，引用计数器对于循环引用的场景没有办法实现回收。例如在上面的例子中，如果ObjectB和ObjectC互相引用，那么即使ObjectA释放了对ObjectB和ObjectC的引用，也无法回收ObjectB、ObjectC

###### 跟踪收集器

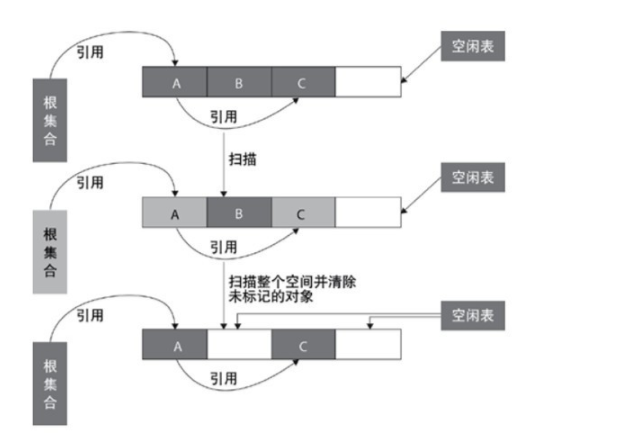
由于引用计数器的循环引用缺陷问题，后来出现了跟踪收集器，它采用的为集中式的管理方式，会全局记录数据引用的状态。基于一定条件的触发（例如定时、空间不足时），执行时需要从根集合来扫描对象的引用关系，这可能会造成应用程序暂停。主要有复制（Copying）、标记-清除（Mark-Sweep）和标记-压缩（Mark-Compact）三种实现算法。

1. **复制（Copying）**



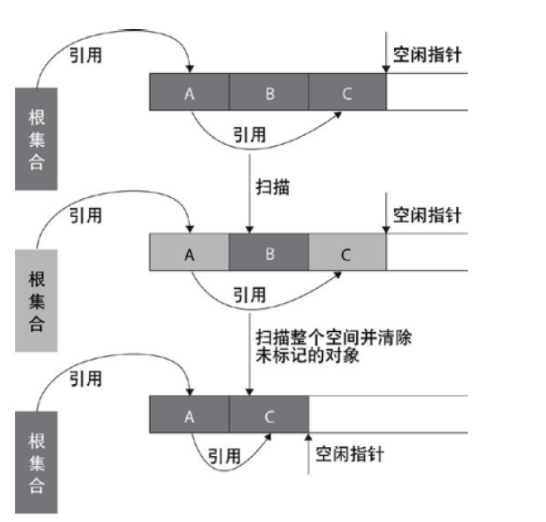
该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成 一个对象面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于copying算法的垃圾收集就从根集中扫描活动对象，并将每个活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。一种典型的基于coping算法的垃圾回收是stop-and-copy算法，它将堆分成对象面和空闲区域面，在对象面与空闲区域面的切换过程中，程序暂停执行。

1. **标记-清除（Mark-Sweep）**



标记-清除算法采用从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，如上图所示。标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

1. **标记-压缩（Mark-Compact）**



标记-压缩算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。在基于Compacting算法的收集器的实现中，一般增加句柄和句柄表。

1. **分代搜索算法**

当前商业虚拟机都采用这个“分代收集”算法（Generation Collection），它根据对象存活周期的不同将内存划分为几块，一般是把java堆分为新生代和老年代，根据各个年代的特点选用不同的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，因此可以选用“复制算法”，此时只需要付出少量存活对象的复制成本即可；对于老年代，因为对象存活率较高、也没有额外空间为期分配担保，就必须使用“标记-清除”或“标记-整理”算法来进行回收。

### Android内存管理

### 内存监控与分析工具