**内存溢出和内存泄漏的区别（内存泄漏原因）**

内存溢出 out of memory，是指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory；比如申请了一个integer,但给它存了long才能存下的数，那就是内存溢出。

内存泄露 memory leak，是指程序在申请内存后，无法释放已申请的内存空间，一次内存泄露危害可以忽略，但内存泄露堆积后果很严重，无论多少内存,迟早会被占光。

memory leak会最终会导致out of memory！

内存溢出就是你要求分配的内存超出了系统能给你的，系统不能满足需求，于是产生溢出。   
  
    内存泄漏是指你向系统申请分配内存进行使用(new)，可是使用完了以后却不归还(delete)，结果你申请到的那块内存你自己也不能再访问（也许你把它的地址给弄丢了），而系统也不能再次将它分配给需要的程序。一个盘子用尽各种方法只能装4个果子，你装了5个，结果掉倒地上不能吃了。这就是溢出！比方说栈，栈满时再做进栈必定产生空间溢出，叫上溢，栈空时再做退栈也产生空间溢出，称为下溢。就是分配的内存不足以放下数据项序列,称为内存溢出.   
  
   以发生的方式来分类，内存泄漏可以分为4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。   
  
从用户使用程序的角度来看，内存泄漏本身不会产生什么危害，作为一般的用户，根本感觉不到内存泄漏的存在。真正有危害的是内存泄漏的堆积，这会最终消耗尽系统所有的内存。从这个角度来说，一次性内存泄漏并没有什么危害，因为它不会堆积，而隐式内存泄漏危害性则非常大，因为较之于常发性和偶发性内存泄漏它更难被检测到

**Java内存泄露引起原因**

首先，什么是内存泄露？经常听人谈起内存泄露，但要问什么是内存泄露，没几个说得清楚。内存泄露是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成的内存空间的浪费称为内存泄露。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你Out of memory。  
那么，Java内存泄露根本原因是什么呢？长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景。具体主要有如下几大类：   
1、静态集合类引起内存泄露：   
像HashMap、Vector等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象Object也不能被释放，因为他们也将一直被Vector等引用着。   
例:   
Static 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，循环申请Object 对象，并将所申请的对象放入一个Vector 中，如果仅仅释放引用本身（o=null），那么Vector 仍然引用该对象，所以这个对象对GC 来说是不可回收的。因此，如果对象加入到Vector 后，还必须从Vector 中删除，最简单的方法就是将Vector对象设置为null。

2、当集合里面的对象属性被修改后，再调用remove（）方法时不起作用。

例：   
public static void main(String[] args)   
{   
Set<Person> set = new HashSet<Person>();   
Person p1 = new Person("唐僧","pwd1",25);   
Person p2 = new Person("孙悟空","pwd2",26);   
Person p3 = new Person("猪八戒","pwd3",27);   
set.add(p1);   
set.add(p2);   
set.add(p3);   
System.out.println("总共有:"+set.size()+" 个元素!"); //结果：总共有:3 个元素!   
p3.setAge(2); //修改p3的年龄,此时p3元素对应的hashcode值发生改变 

set.remove(p3); //此时remove不掉，造成内存泄漏

set.add(p3); //重新添加，居然添加成功   
System.out.println("总共有:"+set.size()+" 个元素!"); //结果：总共有:4 个元素!   
for (Person person : set)   
{   
System.out.println(person);   
}   
}

3、监听器   
在java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如addXXXListener()等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

4、各种连接   
比如数据库连接（dataSourse.getConnection()），网络连接(socket)和io连接，除非其显式的调用了其close（）方法将其连接关闭，否则是不会自动被GC 回收的。对于Resultset 和Statement 对象可以不进行显式回收，但Connection 一定要显式回收，因为Connection 在任何时候都无法自动回收，而Connection一旦回收，Resultset 和Statement 对象就会立即为NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在try里面去的连接，在finally里面释放连接。

5、内部类和外部模块等的引用   
内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员A 负责A 模块，调用了B 模块的一个方法如：   
public void registerMsg(Object b);   
这种调用就要非常小心了，传入了一个对象，很可能模块B就保持了对该对象的引用，这时候就需要注意模块B 是否提供相应的操作去除引用。

6、单例模式   
不正确使用单例模式是引起内存泄露的一个常见问题，单例对象在被初始化后将在JVM的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部对象的引用，那么这个外部对象将不能被jvm正常回收，导致内存泄露，考虑下面的例子：   
class A{   
public A(){   
B.getInstance().setA(this);   
}   
....   
}   
//B类采用单例模式   
class B{   
private A a;   
private static B instance=new B();   
public B(){}   
public static B getInstance(){   
return instance;   
}   
public void setA(A a){   
this.a=a;   
}   
//getter...   
}   
显然B采用singleton模式，它持有一个A对象的引用，而这个A类的对象将不能被回收。想象下如果A是个比较复杂的对象或者集合类型会发生什么情况