#Java中的内存泄漏

1.Java内存回收机制

不论哪种语言的内存分配方式，都需要返回所分配内存的真实地址，也就是返回一个指针到内存块的首地址。Java中对象是采用new或者反射的方法创建的，这些对象的创建都是在堆（Heap）中分配的，所有对象的回收都是由Java虚拟机通过垃圾回收机制完成的。GC为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控，Java会使用有向图的方法进行管理内存，实时监控对象是否可以达到，如果不可到达，则就将其回收，这样也可以消除引用循环的问题。在Java语言中，判断一个内存空间是否符合垃圾收集标准有两个：一个是给对象赋予了空值null，以下再没有调用过，另一个是给对象赋予了新值，这样重新分配了内存空间。

2.Java内存泄漏引起的原因

内存泄漏是指无用对象（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，从而造成内存空间的浪费称为内存泄漏。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。

Java内存泄漏的根本原因是什么呢？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漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是Java中内存泄漏的发生场景。具体主要有如下几大类：

1、静态集合类引起内存泄漏：

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。

例如

```
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。

例如：

```
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漏的一个常见问题，单例对象在初始化后将在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的引用，那么这个对象将不能被JVM正常回收，导致内存泄漏，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