### 什么是Java中的内存泄露

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

由于java的JVM引入了垃圾回收机制，垃圾回收器会自动回收不再使用的对象，了解JVM回收机制的都知道JVM是使用引用计数法和可达性分析算法来判断对象是否是不再使用的对象，**本质都是判断一个对象是否还被引用**。那么对于这种情况下，由于代码的实现不同就会出现很多种内存泄漏问题（让JVM误以为此对象还在引用中，无法回收，造成内存泄漏）。

### 一个简单的内存泄露的例子

下面给出了一个简单的内存泄露的例子。在这个例子中，我们循环申请Object对象，并将所申请的对象放入一个Vector中，如果我们仅仅释放引用本身，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。

Vector v=new Vector(10);

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

{

    Object o=new Object();

    v.add(o);

    o=null;

}

//此时，所有的Object对象都没有被释放，因为变量v引用这些对象。

### java中内存泄露8种情况的总结

#### 静态集合类

如HashMap、LinkedList等等。如果这些容器为静态的，那么它们的生命周期与程序一致，则容器中的对象在程序结束之前将不能被释放，从而造成内存泄漏。简单而言，长生命周期的对象持有短生命周期对象的引用，尽管短生命周期的对象不再使用，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。

#### 2、**各种连接，如数据库连接、网络连接和IO连接等**

在对数据库进行操作的过程中，首先需要建立与数据库的连接，当不再使用时，需要调用close方法来释放与数据库的连接。只有连接被关闭后，垃圾回收器才会回收对应的对象。否则，如果在访问数据库的过程中，对Connection、Statement或ResultSet不显性地关闭，将会造成大量的对象无法被回收，从而引起内存泄漏。

#### 3、变量不合理的作用域

一般而言，一个变量的定义的作用范围大于其使用范围，很有可能会造成内存泄漏。另一方面，如果没有及时地把对象设置为null，很有可能导致内存泄漏的发生。

public class UsingRandom {

private String msg;

public void receiveMsg(){

readFromNet();// 从网络中接受数据保存到msg中

saveDB();// 把msg保存到数据库中

}

}

**如上面这个伪代码**，通过readFromNet方法把接受的消息保存在变量msg中，然后调用saveDB方法把msg的内容保存到数据库中，此时msg已经就没用了，由于msg的生命周期与对象的生命周期相同，此时msg还不能回收，因此造成了内存泄漏。

实际上这个msg变量可以放在receiveMsg方法内部，当方法使用完，那么msg的生命周期也就结束，此时就可以回收了。还有一种方法，在使用完msg后，把msg设置为null，这样垃圾回收器也会回收msg的内存空间。

#### 内部类持有外部类

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持有外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

#### 改变哈希值

当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为的参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，造成内存泄露

#### 举个例子-看你能否找出内存泄漏

import java.util.Arrays;

public class Stack {

private Object[] elements;

private int size = 0;

private static final int DEFAULT\_INITIAL\_CAPACITY = 16;

public Stack() {

elements = new Object[DEFAULT\_INITIAL\_CAPACITY];

}

public void push(Object e) {

ensureCapacity();

elements[size++] = e;

}

public Object pop() {

if (size == 0)

throw new EmptyStackException();

return elements[--size];

}

private void ensureCapacity() {

if (elements.length == size)

elements = Arrays.copyOf(elements, 2 \* size + 1);

}

}

1**原因分析**

上述程序并没有明显的错误，但是这段程序有一个内存泄漏，随着GC活动的增加，或者内存占用的不断增加，程序性能的降低就会表现出来，严重时可导致内存泄漏，但是这种失败情况相对较少。

代码的主要问题在pop函数，当进行大量的pop操作时，由于引用未进行置空，gc是不会释放的

如果栈先增长，在收缩，那么从栈中弹出的对象将不会被当作垃圾回收，即使程序不再使用栈中的这些队象，他们也不会回收，因为栈中仍然保存这对象的引用，俗称过期引用，这个内存泄露很隐蔽。

2**解决方法**

public Object pop() {

if (size == 0)

throw new EmptyStackException();

Object result = elements[--size];

elements[size] = null;

return result;

}

一旦引用过期，清空这些引用，将引用置空。

#### 7、**缓存泄漏**

内存泄漏的另一个常见来源是缓存，一旦你把对象引用放入到缓存中，他就很容易遗忘，对于这个问题，可以使用WeakHashMap代表缓存，此种Map的特点是，当除了自身有对key的引用外，此key没有其他引用那么此map会自动丢弃此值

**1代码示例**

package com.ratel.test;

import java.util.HashMap;

import java.util.Map;

import java.util.WeakHashMap;

import java.util.concurrent.TimeUnit;

public class MapTest {

static Map wMap = new WeakHashMap();

static Map map = new HashMap();

public static void main(String[] args) {

init();

testWeakHashMap();

testHashMap();

}

public static void init(){

String ref1= new String("obejct1");

String ref2 = new String("obejct2");

String ref3 = new String ("obejct3");

String ref4 = new String ("obejct4");

wMap.put(ref1, "chaheObject1");

wMap.put(ref2, "chaheObject2");

map.put(ref3, "chaheObject3");

map.put(ref4, "chaheObject4");

System.out.println("String引用ref1，ref2，ref3，ref4 消失");

}

public static void testWeakHashMap(){

System.out.println("WeakHashMap GC之前");

for (Object o : wMap.entrySet()) {

System.out.println(o);

}

try {

System.gc();

TimeUnit.SECONDS.sleep(20);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("WeakHashMap GC之后");

for (Object o : wMap.entrySet()) {

System.out.println(o);

}

}

public static void testHashMap(){

System.out.println("HashMap GC之前");

for (Object o : map.entrySet()) {

System.out.println(o);

}

try {

System.gc();

TimeUnit.SECONDS.sleep(20);

} catch (InterruptedException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

System.out.println("HashMap GC之后");

for (Object o : map.entrySet()) {

System.out.println(o);

}

}

}

/\*\* 结果

String引用ref1，ref2，ref3，ref4 消失

WeakHashMap GC之前

obejct2=chaheObject2

obejct1=chaheObject1

WeakHashMap GC之后

HashMap GC之前

obejct4=chaheObject4

obejct3=chaheObject3

Disconnected from the target VM, address: '127.0.0.1:51628', transport: 'socket'

HashMap GC之后

obejct4=chaheObject4

obejct3=chaheObject3

\*\*/

当init函数执行完成后，局部变量字符串引用weakd1,weakd2,d1,d2都会消失，此时只有静态map中保存中对字符串对象的引用，可以看到，调用gc之后，hashmap的没有被回收，而WeakHashmap里面的缓存被回收了。

#### 8.监听器和回调

内存泄漏第三个常见来源是监听器和其他回调，如果客户端在你实现的API中注册回调，却没有显示的取消，那么就会积聚。需要确保回调立即被当作垃圾回收的最佳方法是只保存他的若引用，例如将他们保存成为WeakHashMap中的键。