# Java常见内存泄漏例子及详解

Java的一个重要特性就是通过垃圾收集器（GC）自动管理内存的回收，而不需要程序员自己来释放内存。理论上java中所有不会再被利用的对象所占用的内存都可以被GC回收，但是java也存在内存泄露，但他的表现与C++不同。

# Java中的内存管理

要了解java中的内存泄露，首先就得知道java中的内存是如何管理的。

在java中，我们通常使用new为对象分配内存，而这些内存空间都在堆（Heap）上。下面看一个示例：

public class Simple{

public static void main（String[] args）{

Object obj1 = new Object();

Object obj2 = new Object();

obj2 = obj1

//此时obj2分配的内存空间是可以被清理的

}

}

Java使用有向图的方式进行内存管理：

object1

object2

在有向图中，我们叫做obj1是可达的，obj2就是不可达的，显然不可达的可以被清理。

内存的释放，也即清理那些不可达的对象，是由GC决定和执行的，所以GC会监控每一个对象的状态，包括申请、引用、被应用和赋值等。释放对象的根本原则就是对像不会再被使用：

* 给对象赋予空值null，之后再没有调用过
* 另一个是给对象赋予了新值，这样重新分配了内存空间

通常会认为在堆上分配对象的代价比较大，但是GC却优化了这一操作：c++中，在堆

上分配一块内存，会查找一块适用的内存加以分配，如果对象销毁，这块内存就可以重用；而java中，就像一条长的带子，没分配一个新的对象，java的“堆指针”就向后移动到尚未分配的区域。所以java分配内存的效率可与c++媲美。

但是这种工作方式有一个问题：如果频繁的申请内存，资源将会耗尽。这时GC就会介

入进来，他会回收空间，并使堆中的对象排列更紧凑。这样，就始终会有足够大的内存

空间可以分配。

Gc清理时的引用计数方式：当引用连接至新对象时，引用计数+1；当某个引用离开作用域或被设置为null时，引用计数-1，gc发现这个计数为0时，就回收其占用的内存。这个开销会在引用程序的整个生命周期发生，并且不能处理循环引用的情况。所以这种方式只是用来说明gc的工作方式，而不会被任何一种jvm应用。

多数GC采用一种自适应的清理方式（加上其他附加的用于提升速度的技术），主要依据是找出任何“活”的对象，然后采用“自适应的、分代的、停止-复制、标记-清理”式的垃圾回收期。

# Java中的内存泄露

Java中的内存泄露，广义并通俗的说，就是：不再会被使用的对象的内存不能被回收，就是内存泄露。

Java中的内存泄露与c++中的表现有所不同。在C++中，所有被分配了内存的对象，不再使用后，都必须程序员手动的释放他们。所以每个类，都会含有一个析构函数，作用就是完成清理工作，如果我们忘记了某些对象的释放，就会造成内存泄露。

但是在java中，我们不用（也没办法）自己释放内存，无用对象由gc自动清理，这也极大的简化了我们的编程工作。但实际有时候一些不会再被使用的对象，在gc看来不能被释放，就会造成内存泄露。

我们知道，对象都是有生命周期的，有的长、有的短，如果长生命周期的对象持有短生命周期的引用，就很可能会出现内存泄露。我们举一个简单的例子：

Public class Simple{

Object object;

Public void method1(){

object = new Object();

//其他代码

}

}

这里的object1示例，其实我们期望它只作用于method1()方法中，且其他地方不会再用到它，但是，当method1()方法执行完成后，object对象所分配的内存不会马上被认为是可以被释放的对象，只有再Simple类创建的对象被释放后才会被释放，严格的说，这就是一种内存泄露。解决方法就是将object作为method1()方法中的局部变量。当然，如果一定要这么写，可以改为这样：

Public class Simple{

Object object ;

Public void method1(){

Object = new Object();

//其他代码

Object = null;

}

}

这样，之前new Object()分配的内存就可以被GC回收。

到这里，java的内存泄露应该都比较清楚了。下面再进一步说明：

* 在堆中分配的内存，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），这是针对c++等语言的，java中的gc会帮我们处理这种情况，所以我们无需担心）
* 在内存对象明明已经不需要的时候，还保留着这块内存和它的访问方式（引用），这是所有语言都有可能会出现的内存泄露的方式。编程时如果不小心，我们很容易发生这种情况，如果不太严重，可能就只是短暂的内存泄露。

# 一些容易发生内存泄露的例子和解决方法

像上面例子中 的情况很容易发生，也是我们最容易忽略并引发内存泄露的情况，解决的原则就是尽量减小对象的作用域（比如android studio中，上面的代码就会发出警告，并给出的建议是将类的成员变量改写为方法内部的局部变量）以及手动设置null值。

至于作用域，需要在我们编写代码时多注意；null值的手动设置，我们可以看一下java容器LinkedList的源码的删除指定节点的内部方法：

//删除指定节点并返回被删除的元素值

E unlink（Node<E> x）{

//获取当前值和前后节点

Final E element = x.item;

Final Node<E> next = x.next;

Final Node<e> prev = x.prev;

If(prev==null){

First = next;

}else{

Prev.next = next;

x.prev = null;

}

If(next==null){

Last = prev;

}else{

Next.prev = prev;

x.next = null;

}

x.item = null;

size--;

modCount++;

return element;

}

除了修改节点间的关联关系，我们还要做的就是赋值为null的操作，不管gc何时会开始清理，我们都应及时的将无用的对象标记为可被清理的对象。

我们知道java容器ArrayList是数组实现的，如果我们要为其写一个pop（）（弹出）的方法，可能会是这样的：

Public E pop(){

If(size==0){

Return null;

}else{

Return (E)elementData[--size];

}

}

写法很简洁，但这里却会造成内存溢出：elementData[size-1]依然持有E类型对象的引用，并且暂时不能被GC回收。我们可以如下修改：

Public E pop(){

If（size==0）{

Return null;

}else{

E e = (E)elementData[--size];

elementData[size]=null;

return e;

}

}

我们写代码并不能一味的追求简洁，首要是保证其正确性。

# 容器使用时的内存泄露

在很多文章中可能看到一个如下内存泄露例子：

Vector v = new Vector();

For(int i=1;i<100;i++){

Object o = new Object();

v.add(o);

o=null;

}

可能很多人一开始并不理解，下面我们将上面的代码完整以下就好理解了：

Void method(){

Vector v = new Vector();

For(int i=1;i<100;i++){

Object o = new Object();

v.add(o);

o=null;

}

//堆vector 的操作

//与vector无关的其他操作

}

这里的 内存泄露指的是在对vector操作完成之后，执行下面与vector无关的代码时，如果发生了GC，这一系列的object是无法被回收的，而此处的内存泄露可能是短暂的，因为在整个methdo()方法执行完成后，那些对象还是可以被回收。这里要解决很简单，手动赋值null即可：

Void method(){

Vector v = new Vector();

For(int i=1;i<100;i++){

Object o = new Object();

v.add(o);

o=null;

}

//堆vector 的操作

Vector = null;

//与vector无关的其他操作

}

上面的vector已经过时了，不过只是使用老的例子来做内存泄露的介绍。我们使用容器时很容易发生内存泄露，就如上面的例子，不过上例中，容器是方法内的局部变量，造成的内存泄露影响可能不算很大（但我们也应该避免），但是，如果这个容器作为一个类的成员变量，甚至是一个静态（static）的成员变量时，就要更加注意内存泄露了。

下面也是一种使用容器时可能会发生的错误：

Public class CollectionMemory{

Public static void main(String[] args){

Set<MyObject> objects = new LinkedHashSet<MyObject>();

objects.add(new Myobject());

objects.add(new Myobject());

objects.add(new Myobject());

objects.add(new Myobject());

System.Out.println(objects.size());

While(true){

objects.add(new Myobject());

}

}

}

Class MyObject{

List<String> list = new ArrayList<String>(99999);

}

运行上面的代码将很快报错。

如果足够了解java的容器，上面的错误是不可能发生的。

容器Set只存放唯一的元素，是通过对象的equals()方法来比较的，但是java中所有类都直接或间接继承至Object类，Object的equals()方法比较的是对象的地址，上例中，就会一直添加元素知道内存溢出。

所以上例严格的说是容器的错误使用导致的内存溢出。

就Set而言，remove()方法也是通过equals()方法来删除匹配的元素的，如果一个对象确实提供了正确equals()方法，但是切记不要在修改这个对象后使用remove(o)，这也可能会发生内存泄露。

# 各种提供了close方法的对象

比如数据库连接dataSource.getConnection(),网络连接（socket）和io连接，以及使用其他框架的时候，除非其显示的调用了其close()方法（或类似方法）将其连接关闭，否则是不会被gc回收的。其实原因依然是长生命周期对象持有短生命周期对象的引用。

可能很多人使用或Hibernate，我们操作数据库时，通过sessionfactory获取一个session：

Session session = sessionFactory.openSession();

完成后我们必须调用close()方法关闭：

Session.close();

SessionFactory就是一个长生命周期的对象，而session相对是个短生命周期的对象，但是框架这么设计是合理的：它并不清楚我们要使用session到多久，于是只能提供一个方法让我们自己决定何时不再使用。

# 单例模式导致的内存泄露