# 参考

1. 主

https://developer.aliyun.com/article/3009

1. 辅

<https://blog.csdn.net/carson_ho/article/details/79407707>

<https://hio.oppo.com/app/module/detail?tkh_id=2326360&itm_id=35&mod_id=262> oppo课程

<https://www.jianshu.com/p/ac00e370f83d>

<https://juejin.im/entry/56d64b9e816dfa005943a55c>

# 概述

内存管理的目的就是让我们在开发中怎么有效的避免我们的应用出现内存泄漏的问题。内存泄漏简单粗俗的讲，就是该被释放的对象没有释放，一直被某个或某些实例所持有却不再被使用导致 GC 不能回收。

该文从 java 内存泄漏的基础知识开始，并通过具体例子来说明 Android 引起内存泄漏的各种原因，以及如何利用工具来分析应用内存泄漏，最后再做总结。

# Java内存分配策略

## 2.1 Java三种内存分配策略

Java程序运行时的内存分配策略有三种，分别是静态，栈式，和堆式分配。对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区），栈区和堆区。

### 静态存储区（方法区）：

主要存放静态数据，全局static数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。

### （2） 栈区：

当方法被执行时，方法体内的局部变量都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。

### （3） 堆区：

又称动态内存分配，通常就是指程序运行时直接new出来的内存。这部分内存在不使用时将会由Java的GC来负责回收。

## 堆和栈的区别

**栈：**在方法体内定义的（局部变量）一些基本类型的变量和对象的引用变量都是在方法的栈内存中分配的。当在一段方法块中定义了一个变量时，Java就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。

**堆：**堆内存用来存放所有由new创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由Java垃圾回收器来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，该变量的取值等于数组或者对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变量。可以通过该引用变量访问堆中的对象或数组。

|  |
| --- |
| **public class** HeapStack {  **public class** Sample{  **int s1**=0;  Sample **mSample1**=**new** Sample();   **public void** method(){  **int** s2=1;  Sample mSample2=**new** Sample();  }  }  Sample **mSample3**=**new** Sample(); } |

Sample 类的局部变量 s2 和引用变量 mSample2 都是存在于栈中，但 mSample2 指向的对象是存在于堆上的。

mSample3 指向的对象实体存放在堆上，包括这个对象的所有成员变量 s1 和 mSample1，而它自己存在于栈中。

## 结论：

**局部变量**的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。--因为它们属于方法中的变量，生命周期随方法而结束。

**成员变量**全部存储于堆中（包括基本数据类型，引用和引用的对象实体）--因为它们属于类，类对象终究要被new出来使用的。

了解了 Java 的内存分配之后，我们再来看看 Java 是怎么管理内存的。

# Java如何管理内存

## 3.1 有向图

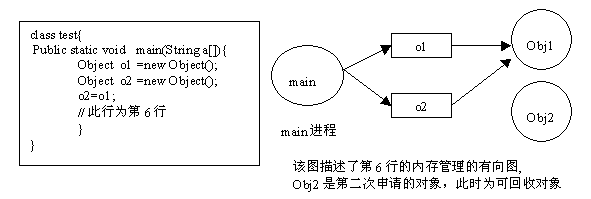
Java内存管理就是对象的分配和释放问题。在Java中，程序员需要通过关键字new为每个对象申请内存空间（基本类型除外），所有的对象都在堆（Heap）中分配空间。另外，对象的释放是由GC决定和执行的。在Java中，内存的分配是由程序完成的，而内存的释放是由GC完成的，这种收支两条线的方法确实简化了程序员的工作。但同时，它也加重了JVM的工作。这也是Java程序运行速度较慢的原因之一。因为GC为了能够正确释放对象，GC必须监控每一个对象的运行状态，包括对象的申请，引用，被引用·，赋值等，GC都需要进行监控。

监视对象状态是为了更加准确地，及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引用对象。另外，每个线程对象都可以作为一个图的起始顶点，例如大多程序从main进程开始执行，那么该图就是以main进程顶点开始的一颗根数。在这个有向图中，根顶点可达的对象都是有效对象，GC将不回收这些对象。如果某个对象（连通子图）与这个根顶点不可达（注意，该图为有向图），那么我们认为这个（这些）对象不再被引用，可以被GC回收。

## 3.2 例子

以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每个时刻，都有一个有向图表示JVM的内存分配情况。以下右图，就是左边程序运行到第6行的示意图。



1. **GC：**Java使用有向图的方式进行内存管理，可以消除引用循环的问题，例如有三个对象，相互引用，只要它们和根进程不可达，则GC也是可以回收它们的。这种方式的优点是内存管理的精度很高，但效率较低。
2. **计数器：**另外一种常用的内存管理技术是使用计数器，例如COM模型采用计数器方式管理构件，它与有向图相比，精度很低（很难处理循环引用问题），但执行效率很高。

# Java中的内存泄露

## 4.1 Java内存泄露定义

在Java中，内存泄露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；

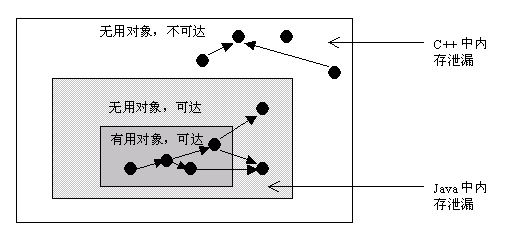
**其次**，这些对象是无用的，即程序以后不会再使用这些对象。

如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄露，这些对象不会被GC所回收，然而它却占用内存。

## 4.2 Java和C++内存泄露对比

在C++中，内存泄露的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，可知对于C++，程序员需要自己管理边和顶点，而对于Java程序员只需要管理边就可以了（不需要管理顶点的释放）。通过该方式，Java提高了编程的效率。



因此，通过以上分析，可知在Java中也有内存泄露，但范围比C++要小一些。因为Java从语言上保证，任何对象都是可达的，所有的不可达对象都由GC管理。

## 4.3 GC的透明性

对于程序员来说，GC基本是透明的，不可见的。虽然，我们只有几个函数可以访问GC，例如运行GC的System.gc()，但是根据Java语言规范定义，该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级较低。JVM调用GC的策略也有很多种，有的是内存使用达到一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，不需要关心这些。除非在一些特定的场合，GC的执行影响应用程序的性能，例如对于基于Web的实时系统，如网络游戏等，用户不希望GC突然中断应用程序执行而进行垃圾回收，那么我们需要调整GC的参数，让GC能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun提供的HotSpot JVM就支持该特性。

## 4.4 例子

同样给出一个 Java 内存泄漏的典型例子，

|  |
| --- |
| **public void** Leak1(){  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，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。

# 5. Android常见内存泄露

参考：

1. 主：<https://blog.csdn.net/carson_ho/article/details/79407707>
2. <https://developer.aliyun.com/article/3009#slide-4>

## （1） 集合类

### 【1】 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

集合类添加元素后，当该元素指向的对象无用时，集合仍引用着该对象，导致该对象不可被回收，从而导致内存泄漏。

### 【2】 解决方案

<https://blog.csdn.net/carson_ho/article/details/79407707>

集合类添加集合元素对象后，在使用后必须从集合中删除。

由于1个集合中有许多元素，故最简单的方法 = 将集合清空&设置为null。

### 【3】 示例

<https://blog.csdn.net/carson_ho/article/details/79407707>

存在泄漏的代码：

|  |
| --- |
| // 通过 循环申请Object 对象 & 将申请的对象逐个放入到集合List List<Object> objectList = new ArrayList<>(); for (int i = 0; i < 10; i++) {  Object o = new Object();  objectList.add(o);  o = null; } // 虽释放了集合元素引用的本身：o=null // 但集合List 仍然引用该对象，故垃圾回收器GC 依然不可回收该对象 |

解决：下面标黄代码

|  |
| --- |
| // 通过 循环申请Object 对象 & 将申请的对象逐个放入到集合List List<Object> objectList = new ArrayList<>(); for (int i = 0; i < 10; i++) {  Object o = new Object();  objectList.add(o);  o = null; } // 虽释放了集合元素引用的本身：o=null // 但集合List 仍然引用该对象，故垃圾回收器GC 依然不可回收该对象  // 释放objectList objectList.clear(); objectList = null; |

## （2） static成员变量

### 【1】 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

被 static 关键字修饰的成员变量的生命周期 = 应用程序的生命周期。

若使被 static 关键字修饰的成员变量 引用耗费资源过多的实例（如Context），则容易出现该成员变量的生命周期 > 引用实例生命周期的情况，当引用实例需结束生命周期销毁时，会因静态变量的持有而无法被回收，从而出现内存泄露。

### 【2】 解决方案

<https://blog.csdn.net/carson_ho/article/details/79407707>

1. 尽量避免 static 成员变量引用资源耗费过多的实例（如 Context）

若需引用 Context，则尽量使用Applicaiton的Context

1. 使用 弱引用（WeakReference） 代替 强引用 持有实例

### 【3】 示例

<https://blog.csdn.net/carson_ho/article/details/79407707>

|  |
| --- |
| **public class** MainActivity **extends** AppCompatActivity {   *// 定义1个静态变量* **private static** Context *sContext*;   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);   *// 引用的是Activity的context  sContext* = **this**;  *// 当Activity需销毁时，由于sContext = 静态 & 生命周期 = 应用程序的生命周期，故 Activity无法被回收，从而出现内存泄露* } } |

### 【4】 典型的例子：单例模式

#### 《1》 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

单例模式 由于其**静态特性**，其生命周期的长度 = 应用程序的生命周期。

若1个对象已不需再使用，而单例对象还持有该对象的引用，那么该对象将不能被正常回收 从而导致内存泄漏。

#### 《2》 解决方案

<https://blog.csdn.net/carson_ho/article/details/79407707>

让 单例模式引用的对象的生命周期 = 应用的生命周期。

#### 《3》 示例

<https://blog.csdn.net/carson_ho/article/details/79407707>

存在泄露的代码：

|  |
| --- |
| *// 创建单例时，有时需传入一个Context // 若传入的是Activity的Context，此时单例则持有该Activity的引用 // 由于单例一直持有该Activity的引用（直到整个应用生命周期结束），即使该Activity退出，该Activity的内存也不会被回收 // 特别是一些庞大的Activity，此处非常容易导致OOM* **public class** SingletonClass {   **private static** SingletonClass *sInstance*;   **private** Context **mContext**;   **private** SingletonClass(Context context) {  **this**.**mContext** = context; *// 传递的是Activity的context* }   **public** SingletonClass getInstance(Context context) {  **if** (*sInstance* == **null**) {  *sInstance* = **new** SingletonClass(context);  }  **return** *sInstance*;  } } |

解决：应传递Application的Context，因Application的生命周期 = 整个应用的生命周期【下面标黄代码】

|  |
| --- |
| **public class** SingletonClass {   **private static** SingletonClass *sInstance*;   **private** Context **mContext**;   **private** SingletonClass(Context context) {**this**.**mContext** = context.getApplicationContext(); *// 传递的是Application的context* }   **public** SingletonClass getInstance(Context context) {  **if** (*sInstance* == **null**) {  *sInstance* = **new** SingletonClass(context);  }  **return** *sInstance*;  } } |

#### 《4》 实例

图形用户界面, 应用程序, Word

描述已自动生成

改成这样解决内泄。

## （3） 非静态内部类/匿名类

### 【1】 前提知识

<https://blog.csdn.net/carson_ho/article/details/79407707>

非静态内部类/匿名类 默认持有 外部类的引用；而静态内部类则不会。

### 【2】 常见情况

#### 《1》 静态变量指向非静态内部类对象

##### 1） 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

若 静态变量（其生命周期 = 应用的生命周期）指向非静态内部类所创建的实例，会因 **非静态内部类默认持有外部类的引用** 而导致外部类无法释放，最终 造成内存泄露。

##### 2） 解决方案

<https://blog.csdn.net/carson_ho/article/details/79407707>

* 将非静态内部类设置为：静态内部类（静态内部类默认不持有外部类的引用）
* 该内部类抽取出来封装成一个单例
* 尽量 避免 非静态内部类所创建的实例 = 静态

##### 3） 示例

<https://blog.csdn.net/carson_ho/article/details/79407707>

|  |
| --- |
| *// 背景：在启动频繁的Activity中，为了避免重复创建相同的数据资源，会在Activity内部创建一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据* **public class** MainActivity **extends** AppCompatActivity {   *// 非静态内部类的实例的引用设置为静态* **public static** InnerClass *innerClass*;   @Override  **protected void** onCreate(Bundle savedInstanceState) {  **super**.onCreate(savedInstanceState);   *// 保证非静态内部类的实例只有1个，即单例* **if** (*innerClass* == **null**) {  *innerClass* = **new** InnerClass();  }  }   *// 非静态内部类的定义* **private class** InnerClass {  *//...* } }  *// 造成内存泄露的原因： // 当MainActivity销毁时，因非静态内部类单例的引用（innerClass）的生命周期=应用App的生命周期 且 非静态内部类单例持有外部类MainActivity的引用，故MainActivity无法被GC回收，从而导致内存泄漏* |

#### 《2》 多线程

##### 1） 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

当多线程的使用方法为非静态内部类/匿名类时【即自定义Thread类为非静态内部类/匿名类】：若工作线程正在处理任务&外部类需销毁，由于工作线程实例持有外部类引用，将使得外部类无法被垃圾回收器（GC）回收，从而造成内存泄漏。

##### 2） 解决方案

<https://blog.csdn.net/carson_ho/article/details/79407707>

造成内存泄露的原因有2个关键条件：

1. 存在 “工作线程实例 持有外部类引用” 的引用关系
2. 工作线程实例的生命周期 > 外部类的生命周期，即工作线程仍在运行 而 外部类需销毁

所以解决方案思路：使得上述任一条件不成立 **即可。具体有2个解决方案：**

1. 静态内部类
2. 当外部类结束生命周期时，强制结束线程

##### 3） 示例

<https://blog.csdn.net/carson_ho/article/details/79407707>

多线程主要使用的是：AsyncTask、实现Runnable接口 & 继承Thread类。

3者内存泄露的原理相同，此处以继承Thread类 为例说明。

###### 内存泄漏写法

代码：

方式1：新建Thread子类（内部类）

|  |
| --- |
| public class MainActivity1 extends AppCompatActivity {   public static final String *TAG* = "carson：";   @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.*activity\_main*);   *// 通过创建的内部类 实现多线程* new MyThread().start();   }   *// 自定义的Thread子类* private class MyThread extends Thread {  @Override  public void run() {  try {  Thread.*sleep*(5000);  Log.*d*(*TAG*, "执行了多线程");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

方式2：匿名Thread内部类

|  |
| --- |
| public class MainActivity2 extends AppCompatActivity {   public static final String *TAG* = "carson：";   @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.*activity\_main*);   *// 通过匿名内部类 实现多线程* new Thread() {  @Override  public void run() {  try {  Thread.*sleep*(5000);  Log.*d*(*TAG*, "执行了多线程");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }.start();  } } |

分析：

// 工作线程Thread类属于非静态内部类 / 匿名内部类，运行时默认持有外部类的引用

// 当工作线程运行时，若外部类MainActivity需销毁

// 由于此时工作线程类实例持有外部类的引用，将使得外部类无法被垃圾回收器（GC）回收，从而造成 内存泄露

###### 解决

解法1：静态内部类

|  |
| --- |
| */\*\*  \* 解决方式1：静态内部类  \* 原理：静态内部类 不默认持有外部类的引用，从而使得 “工作线程实例 持有 外部类引用” 的引用关系 不复存在  \* 具体实现：将Thread的子类设置成 静态内部类  \*/* public class MainActivity1 extends AppCompatActivity {   public static final String *TAG* = "carson：";   @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.*activity\_main*);   *// 通过创建的内部类 实现多线程* new MyThread().start();   }   *// 分析1：自定义Thread子类  // 设置为：静态内部类* private static class MyThread extends Thread {  @Override  public void run() {  try {  Thread.*sleep*(5000);  Log.*d*(*TAG*, "执行了多线程");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } } |

解法2：当外部类结束生命周期时，强制结束线程

|  |
| --- |
| */\*\*  \* 解决方案2：当外部类结束生命周期时，强制结束线程  \* 原理：使得 工作线程实例的生命周期 与 外部类的生命周期 同步  \* 具体实现：当 外部类（此处以Activity为例） 结束生命周期时（此时系统会调用onDestroy（）），强制结束线程（调用stop（））  \*/* public class MainActivity2 extends AppCompatActivity {   public static final String *TAG* = "carson：";   Thread thread;   @Override  public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.*activity\_main*);   *// 通过匿名内部类 实现多线程* thread = new Thread() {  @Override  public void run() {  try {  Thread.*sleep*(5000);  Log.*d*(*TAG*, "执行了多线程");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  };  thread.start();  }   @Override  protected void onDestroy() {  super.onDestroy();  thread.stop();  *// 外部类Activity生命周期结束时，强制结束线程* } } |

#### 《3》 消息传递机制：Handler

<https://www.jianshu.com/p/ed9e15eff47a>

<https://developer.aliyun.com/article/3009#slide-4>

##### x） 泄露原因

见图形用户界面, 应用程序

描述已自动生成

##### x） 解决方案

###### 1》 静态内部类（+弱引用）

A） 方案

A》 静态内部类：将Handler的子类设置成静态内部类。

B》弱引用：此外，还可**使用WeakReference弱引用**

**持有外部类**，保证外部类能被回收。

B） 原理

A》 静态内部类

静态内部类不默认持有外部类的引用，从而使得 “未被处理 / 正处理的消息 -> Handler实例 -> 外部类” 的引用关系 不存在。

B》 弱引用

弱引用的对象拥有短暂的生命周期，在垃圾回收器

线程扫描时，一旦发现了具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。

C） 示例

|  |
| --- |
| public class SampleActivity extends Activity {   */\*\*  \* Instances of static inner classes do not hold an implicit  \* reference to their outer class.  \*/* private static class MyHandler extends Handler {  private final WeakReference<SampleActivity> mActivity;   public MyHandler(SampleActivity activity) {  mActivity = new WeakReference<SampleActivity>(activity);  }   @Override  public void handleMessage(Message msg) {  SampleActivity activity = mActivity.get();  if (activity != null) {  *// ...* }  }  }   private final MyHandler mHandler = new MyHandler(this);   */\*\*  \* Instances of anonymous classes do not hold an implicit  \* reference to their outer class when they are "static".  \*/* private static final Runnable *sRunnable* = new Runnable() {  @Override  public void run() { */\* ... \*/* }  };   @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);   *// Post a message and delay its execution for 10 minutes.* mHandler.postDelayed(*sRunnable*, 1000 \* 60 \* 10);   *// Go back to the previous Activity.  // 制造泄漏条件* finish();  } } |

D） 项目实例

例1：oppo-智能家居 uwb项目

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成



该Handler用于不停地接收和发送手机壳连接相关消息，并更新进度条ui。

###### 2》 当外部类结束生命周期时，清空Handler内消息队列

A） 方案

<https://www.jianshu.com/p/ed9e15eff47a>

当 外部类（此处以Activity为例） 结束生命周期时（此时系统会调用onDestroy（）），清除 Handler消息队列里的所有消息（调用removeCallbacksAndMessages(null)）。

<https://developer.aliyun.com/article/3009#slide-4>

前面所说的，创建一个静态Handler内部类，然后对 Handler 持有的对象使用弱引用，这样在回收时也可以回收 Handler 持有的对象，但是这样做虽然避免了 Activity 泄漏，不过 Looper 线程的消息队列中还是可能会有待处理的消息，所以我们在 Activity 的 Destroy 时或者 Stop 时应该移除消息队列 MessageQueue 中的消息。

下面几个方法都可以移除 Message：

|  |
| --- |
| public final void removeCallbacks(Runnable r);  public final void removeCallbacks(Runnable r, Object token);  public final void removeCallbacksAndMessages(Object token);  public final void removeMessages(int what);  public final void removeMessages(int what, Object object); |

B） 原理

<https://www.jianshu.com/p/ed9e15eff47a>

不仅使得 “未被处理 / 正处理的消息 -> Handler实例 -> 外部类” 的引用关系 不复存在，同时使得 Handler的生命周期（即 消息存在的时期）与外部类的生命周期 同步。

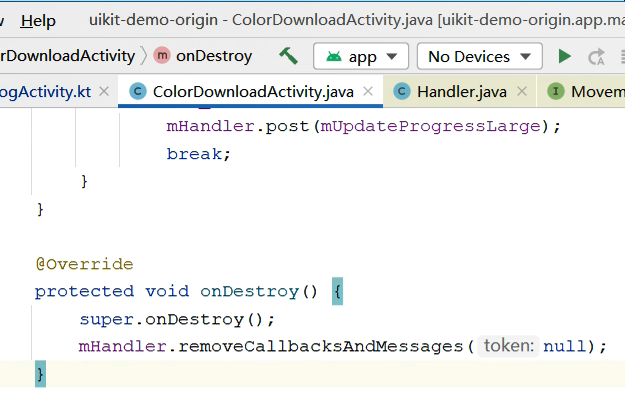
C） 示例

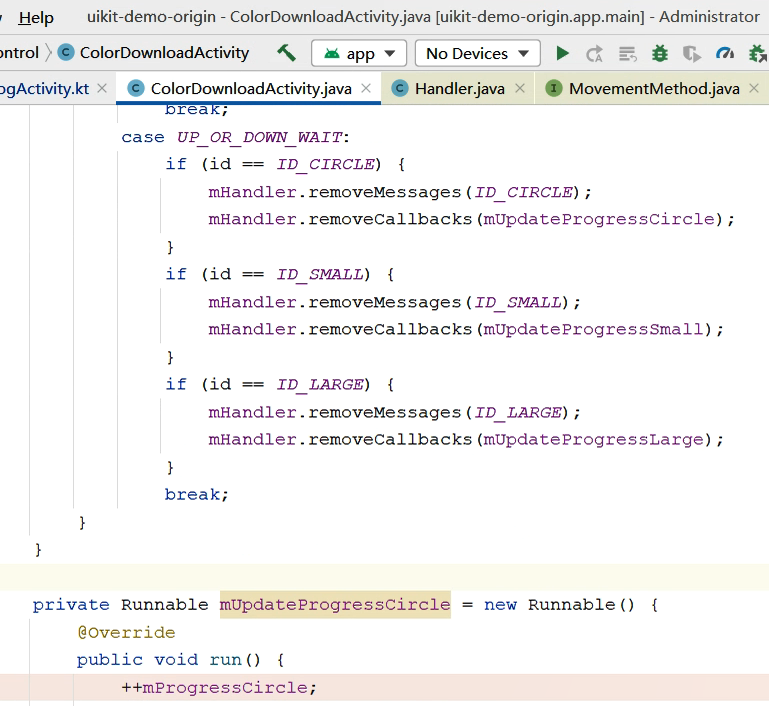
<https://www.jianshu.com/p/ed9e15eff47a>

|  |
| --- |
| @Override protected void onDestroy() {  super.onDestroy();  mHandler.removeCallbacksAndMessages(null);  *// 外部类Activity生命周期结束时，同时清空消息队列 & 结束Handler生命周期* } |

D） 项目实例

例1：oppo的uikit\_demo





以上ColorDownloadActivity为“用Handler控制下载ui显示”的功能：



## （4） 资源对象使用后未关闭

### 【1】 泄露原因

<https://blog.csdn.net/carson_ho/article/details/79407707>

对于资源的使用（如 广播BroadcastReceiver、文件流File、数据库游标Cursor、图片资源Bitmap等），若在Activity销毁时未及时关闭 / 注销这些资源，则这些资源将不会被回收，从而造成内存泄漏。

### 【2】 解决方案

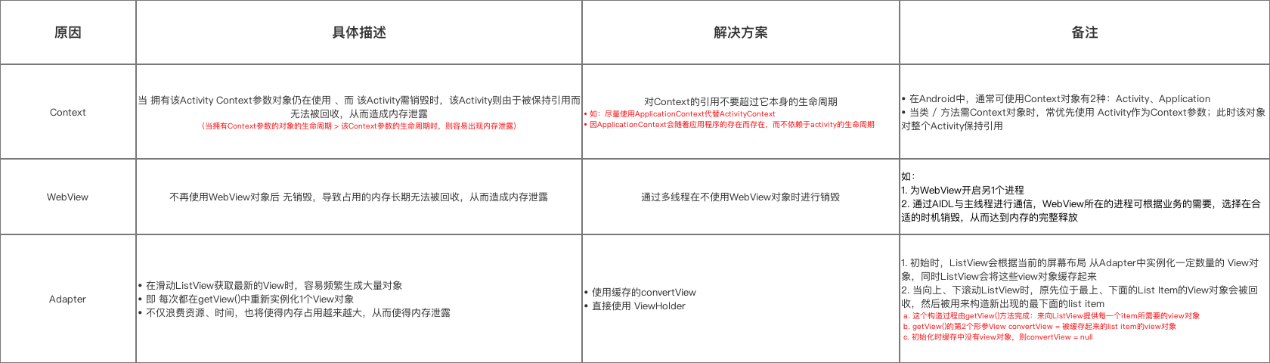
<https://blog.csdn.net/carson_ho/article/details/79407707>

在Activity销毁时及时关闭/注销资源。

|  |
| --- |
| *// 对于 广播BraodcastReceiver：注销注册* unregisterReceiver();  *// 对于 文件流File：关闭流* InputStream/OutputStream.close();  *// 对于 数据库游标cursor：使用后关闭游标* cursor.close();  *// 对于 图片资源Bitmap：Android分配给图片的内存只有8M，若1个Bitmap对象占内存较多，当它不再被使用时，应调用recycle()回收此对象的像素所占用的内存；最后再赋为null* Bitmap.recycle(); Bitmap = **null**;  *// 对于 属性动画：将动画设置成无限循环播放repeatCount = “infinite”后，在Activity退出时记得停止动画* |

## （x） 其他

除了上述4种常见情况，还有一些日常的使用会导致内存泄露。主要包括：Context、WebView、Adapter，具体介绍如下



# 6. 内存泄漏和内存溢出

## 参1：<https://blog.csdn.net/carson_ho/article/details/79549417>

内存泄露容易使得应用程序发生内存溢出，即 OOM

内存溢出 简介：

文本

中度可信度描述已自动生成

## 参2：oppo课程

<https://hio.oppo.com/app/module/detail?tkh_id=2326360&itm_id=35&mod_id=262>

文本, 信件

描述已自动生成

文本

描述已自动生成

# 7. 引用的使用

<https://yq.aliyun.com/articles/3009?spm=a2c4e.11153940.0.0.72d32b94X9SZW3&p=1&do=login&accounttraceid=28c9141ffe5342af95247141b45ebb54doni>

前面提到了 WeakReference，所以这里就简单的说一下 Java 对象的几种引用类型。

Java对引用的分类有 Strong reference, SoftReference, WeakReference, PhatomReference 四种。



在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且生命周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

假设我们的应用会用到大量的默认图片，比如应用中有默认的头像，默认游戏图标等等，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生OutOfMemory异常。这时，我们可以考虑使用软/弱引用技术来避免这个问题发生。以下就是高速缓冲器的雏形：

首先定义一个HashMap，保存软引用对象。

private Map <String, SoftReference<Bitmap>> imageCache = new HashMap <String, SoftReference<Bitmap>> ();

再来定义一个方法，保存Bitmap的软引用到HashMap。



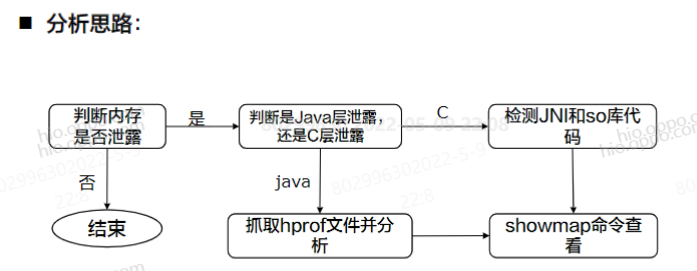
使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从而避免内存达到上限，避免Crash发生。

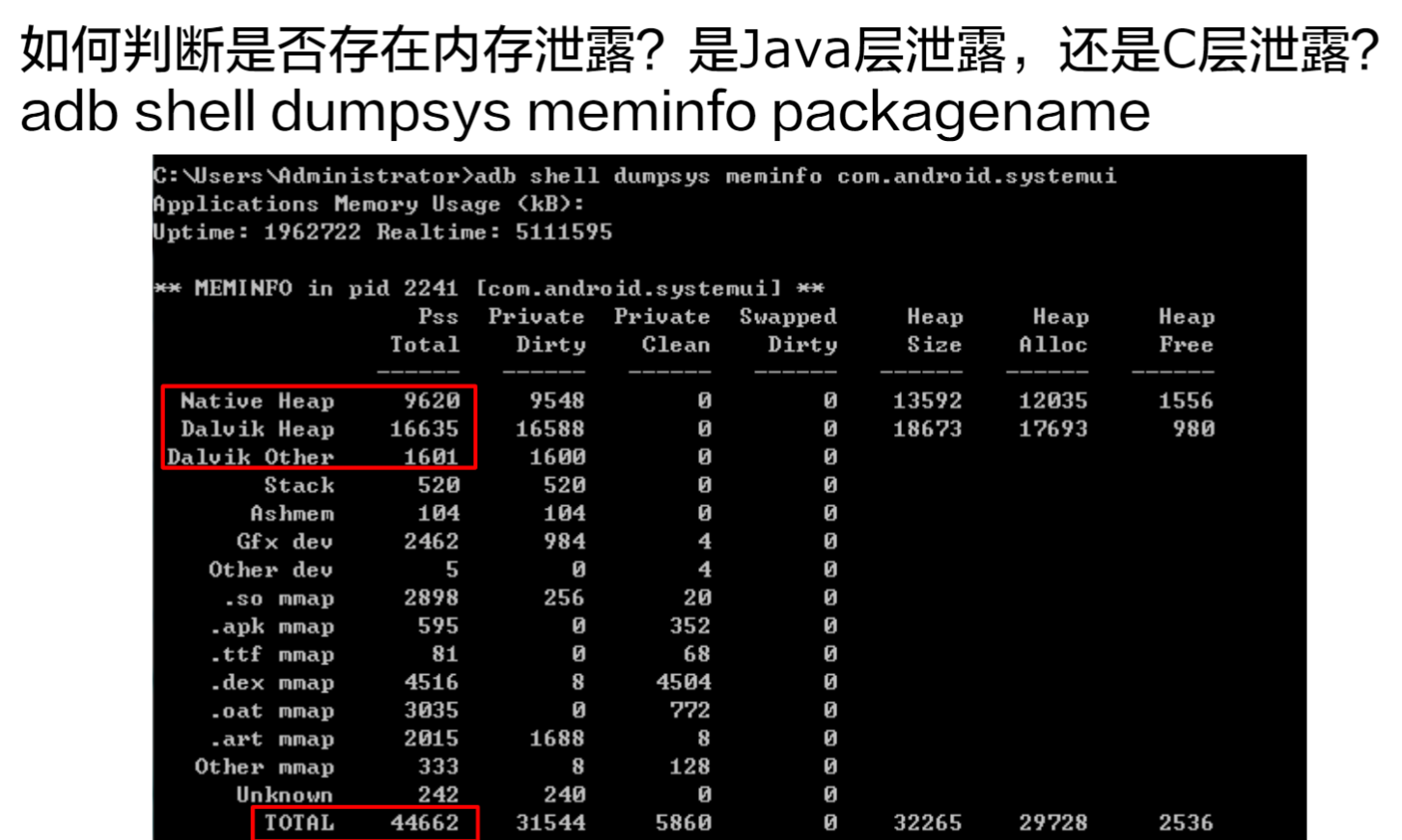
如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

# 8. 内存泄露分析思路

<https://hio.oppo.com/app/module/detail?tkh_id=2326360&itm_id=35&mod_id=262>





# 9. 内存泄露分析工具

<https://blog.csdn.net/carson_ho/article/details/79407707>

<https://developer.aliyun.com/article/3009#slide-5>

（1） MAT

（2） LeakCanary

（3） AS的Memory Profiler【原本的Memory Monitor】

<https://zhuanlan.zhihu.com/p/33858521>

（4） Heap Viewer

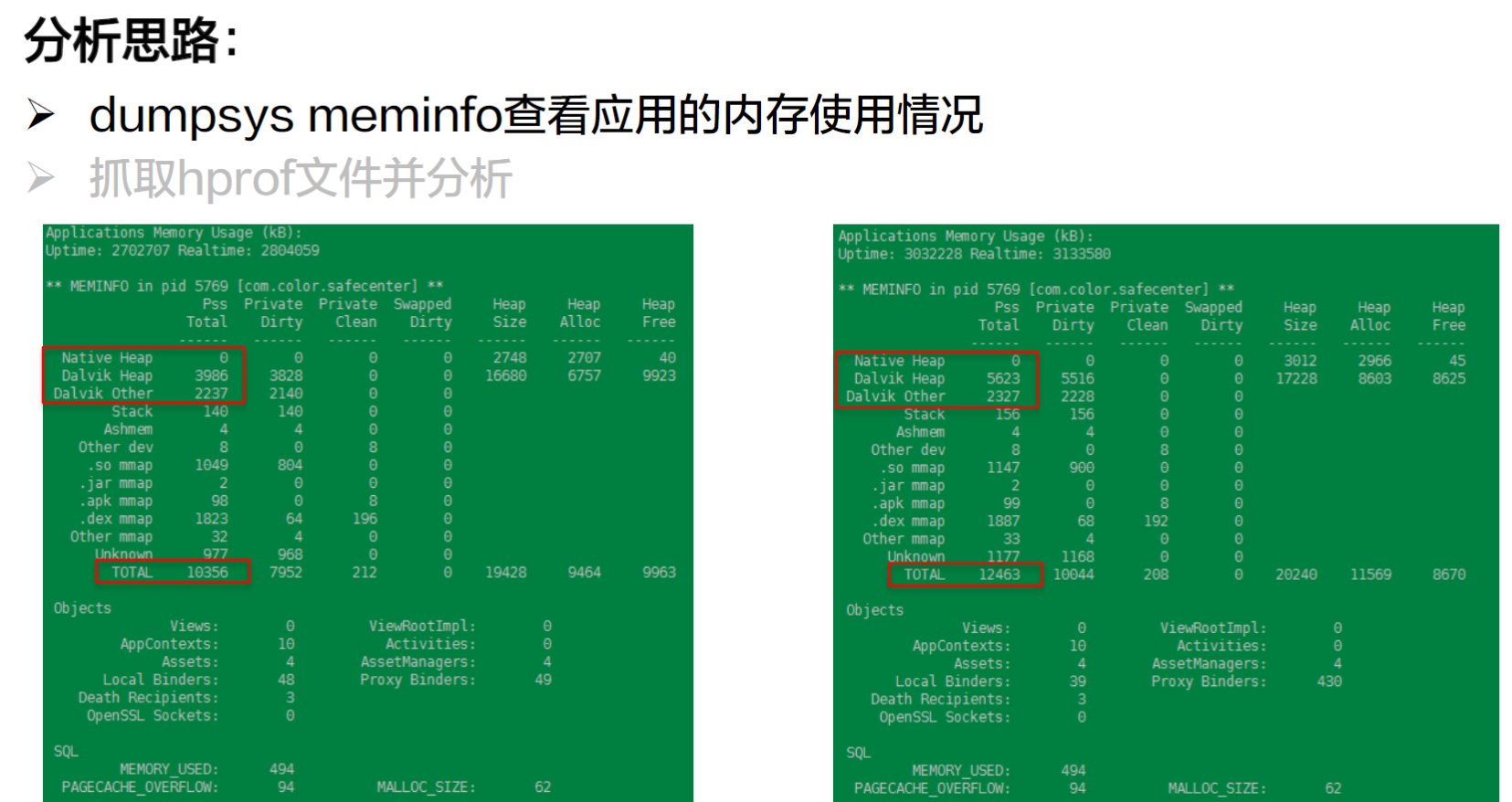
（5） Allocation Tracker

（6） YourKit

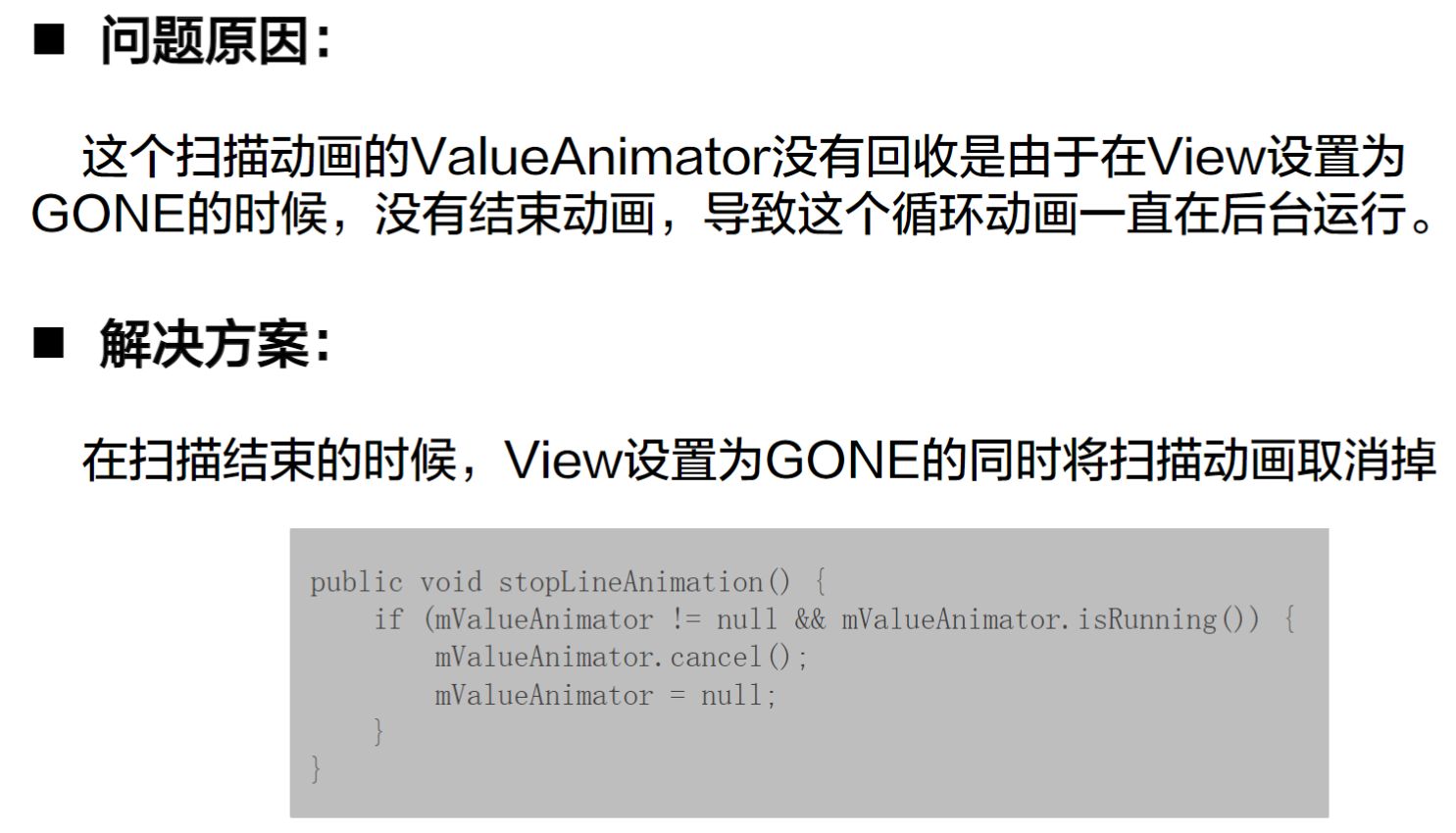
# 10. 内存泄露项目实例

<https://hio.oppo.com/app/module/detail?tkh_id=2326360&itm_id=35&mod_id=262>









# 11. 总结

## 参1：<https://developer.aliyun.com/article/3009>

（1） 对Activity等组件的引用应该控制在Activity的生命周期之内；如果不能就考虑getApplicationContext或者getApplication，以避免Activity被外部长生命周期的对象引用而泄露。

（2） 尽量不要在静态变量或者静态内部类中使用非静态外部成员变量（包括context），即使要使用，也要考虑适时把外部成员变量置空；也可以在内部类中使用弱引用来引用外部类的变量。

（3） 对于生命周期比Activity长的内部类对象，并且内部类中使用了外部类的成员变量，可以这样做避免内存泄露：

【1】 将内部类改为静态内部类

【2】 静态内部类中使用弱引用来引用外部类的成员变量

（4） Handler持有的引用对象最好使用弱引用，资源释放时也可以清空Handler里面的消息。比如在Activity onStop或者onDestroy的时候，取消掉该Handler对象的Message和Runnable。

（5） 在Java的实现过程中，也要考虑其对象释放，最好的方法是在不使用某对象时，显示地将此对象赋值为null，比如使用完Bitmap后先调用recycle（），再赋值为null，清空对图片等资源有直接引用或者间接引用的数组（使用array.clear()；array=null）等，最好遵循谁创建谁释放的原则。

（6） 正确关闭资源，对于使用了BroadcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源，应该在Activity销毁时及时关闭或注销。

（7） 保持对对象生命周期的敏感，特别注意单例，静态对象，全局性集合等的生命周期。

## 参2：

<https://hio.oppo.com/app/module/detail?tkh_id=2326360&itm_id=35&mod_id=262>

