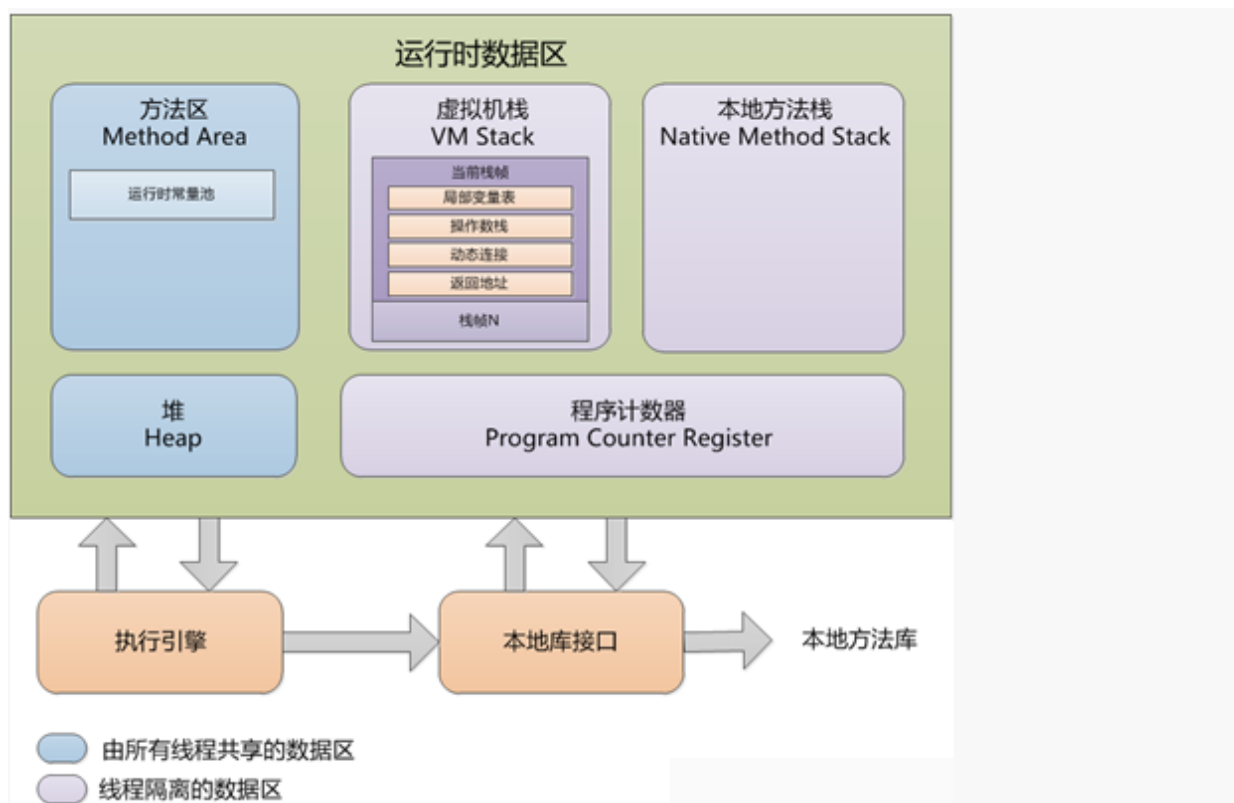


android 内存泄漏分析指北

java 垃圾回收介绍:

Java 虚拟机运行所管理的内存包括以下几个运行时的数据区域
如下图:



程序计数器: 一块比较小的内存区域, 可以看作是当前线程所执行的字节码的行号指示器。且每个线程都有一个独立的程序计数器。

java 虚拟机栈: 线程私有的, 描述的是java 方法执行的内存模型, 每个线程执行的时候都会创建一个栈帧用于储存 局部变量、操作数栈、动态链接、方法出口、等信息。一个方法的调用到执行结束的过程就对应一个栈帧在虚拟机栈中的入栈到处栈的过程。虚拟机局部变量表中存放了编译可知的各种局部数据类型 (`boolean`、`byte`、`char`、`short`、`int`、`float`、`long`、`double`)、对象引用、返回地址。

本地方法栈: 和虚拟机栈类似, 其中虚拟机栈为执行java 方法服务, 而本地方法栈为虚拟机使用的 native 的方法服务
java 堆: java 虚拟机所管理的内存中最大的一块, 且其是被所有的线程共享的一块内存区域, 在虚拟机启动的时候创建。该区域的唯一目的就是来存放对象实例的。java 堆是垃圾啊回收管理的主要区域, 因此在很多的时候被叫做"GC堆"

方法区: 和java 堆一样是各个线程共享的内存区域, 用来存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码和数据。

运行中常量池: 为方法区的一部分。class 文件中除了有类的版本、字段、方法、接口等描述信息外、还有一项信息是常量池、用来存放编译期生产的各种字面量和符号引用。

GC 时那些内存需要释放：

首先对于程序计数器、虚拟机栈、本地方法栈 这3个区域都是随线程而生、随线程而亡。栈中的栈帧随着方法的进入和退出执行着如栈和出栈的操作。每一个栈帧中分配的内存基本上在类结构确定下来的时候已经是可知的了。所以在这几个区域就不需要考虑内存回收的问题，因为在方法结束，或者线程结束的时候内存自然就会回收了。

但是java 堆和方法区确不一样，如一个接口中的多个实现类需要的内存可能不一样，一个方法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行的情况下才可以知道会创建那些对象，这部分的内存的分配和回收也是动态的，垃圾回收所关注的也这一部分的内容。

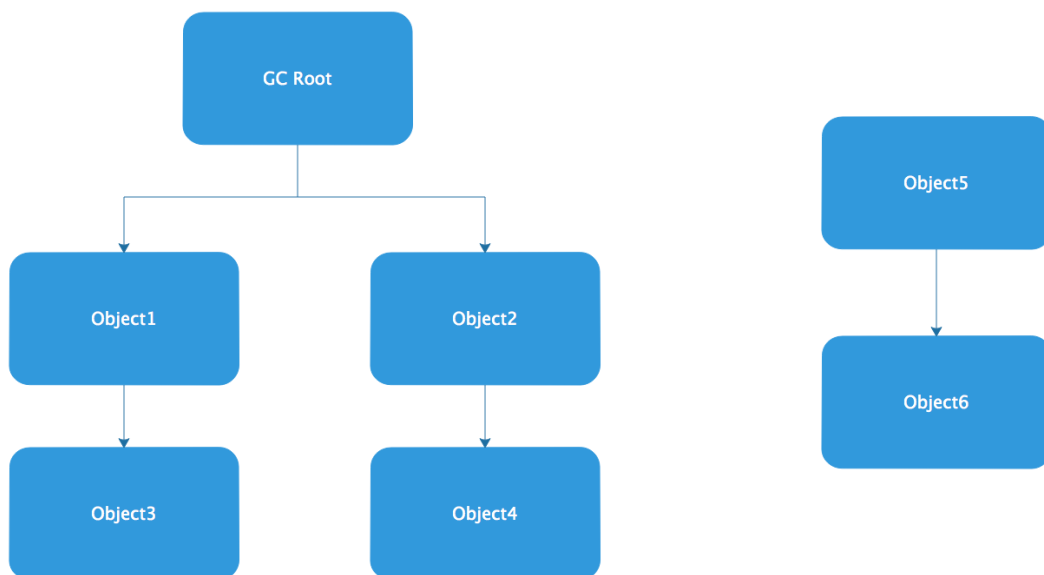
确定哪些对象是存活的，那些已经“死去”（即不可能被任何途径使用的对象）

方式：

A、引用计数法： 给对象添加一个引用计数器，每当有一个地方引用它时，计数器就加1；当引用失效时就减1；任何时刻计数器为0的对象是就不可能再被使用的。（主流的 **JAVA** 虚拟机并没有使用引用计数法来管理内存，其中主要的原因是它很难解决对象的相互循环引用的问题）。

B、可达性分析法： 通过一系列的称为 **GC Roots** 的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到 **GC Roots** 没有任何引用链相连时（即 **GC Roots** 到这个对象不可达），则证明这个对象是不可用的。

如下图论：



如上图所示，object1~object4对GC Root都是可达的，说明不可被回收，object5和object6对GC Root节点不可达，说明其可以被回收

可作为GC roots 的对象包括：

虚拟机栈中的应用的对象、方法区中的静态属性引用的对象、方法区中常量引用的对象、本地方法中jni 引用的对象。

各中引用：

强引用： 只要强引用还存在，垃圾收集器将永远不会回收掉被引用的对象。

软引用： 用来描述一些还有用但并非必须的对象、在系统将要发生内存溢出异常之前，将会把这些对象列进对象回收范围中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出的异常。

弱引用： 也是用来描述非必须对象，其强度比软引用更弱些，被弱引用关联的对象只能生存到下一次垃圾收集发生之前，当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联

的对象。

虚引用：一种最弱的引用关系、一个对象是否有引用存在完全不会影响其生存时间、且无法通过虚引用活着一个对象的实例。一个虚引用唯一的目的就是在这个对象在被收集器回收的时候能够收到一个系统通知。

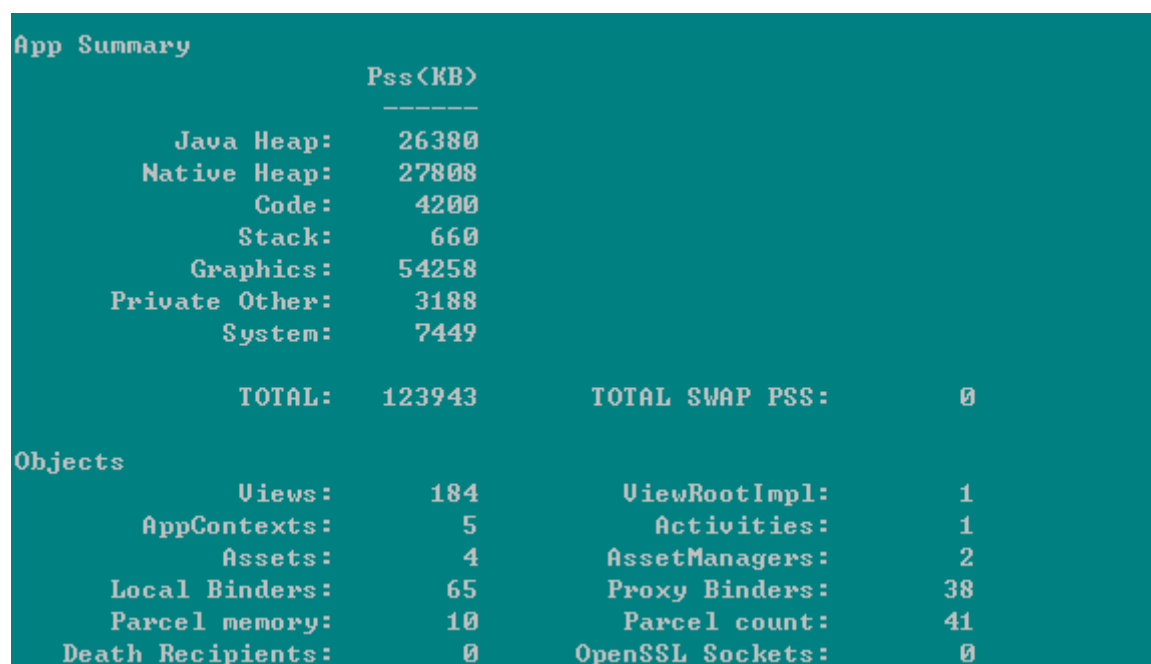
参考：深入理解java 虚拟机第二版

内存泄漏检测：

1、**Dump App 的 meminfo**：在一个App 中存在着很多个Activity，其实我们只需要一个一个界面去检查其是否存在Activity泄漏的情况，

利用 `adb shell dumpsys meminfo <package name>` 来 dump 该 app 的内存信息，该信息中有包含当前 app 中所未释放的 **activity** 的数量，以及 **view** 的个数，如下图所示，在进入一个界面之前检查下 **activity** 数量和 **view** 的数量，在退出该界面后检查下在查看一边 **activity** 和 **view** 的数量，对比进入和退出后activity 和view 的数量是否有差异。

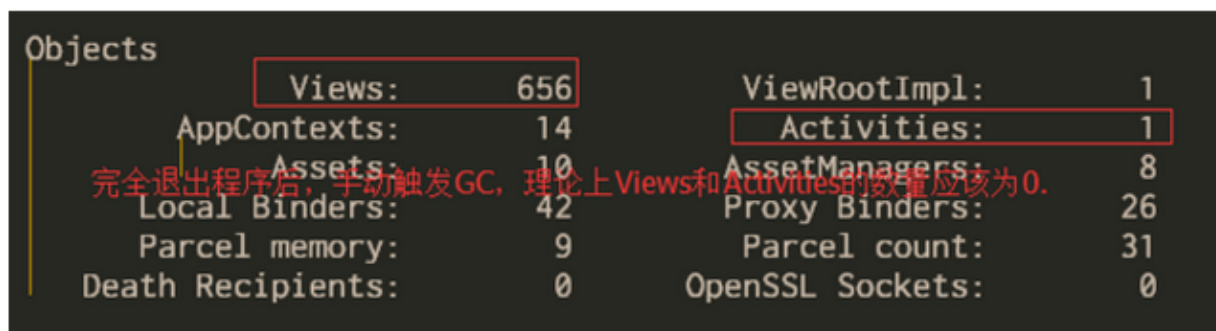
如下图：目前存在着1个activity 184 个 view。因此我们如果要检查某个activity 是否存在泄漏，我们只需要在进入该activity 之前dump 下该信息，然后在进入该activity，进行一系列操作，然后点击back键，退出。



The screenshot shows the output of the 'adb shell dumpsys meminfo' command. It is divided into two main sections: 'App Summary' and 'Objects'.

App Summary	
	Pss(KB)
Java Heap:	26380
Native Heap:	27808
Code:	4200
Stack:	660
Graphics:	54258
Private Other:	3188
System:	7449
TOTAL:	123943
TOTAL SWAP PSS:	0

Objects			
Views:	184	ViewRootImpl:	1
AppContexts:	5	Activities:	1
Assets:	4	AssetManagers:	2
Local Binders:	65	Proxy Binders:	38
Parcel memory:	10	Parcel count:	41
Death Recipients:	0	OpenSSL Sockets:	0



This screenshot shows the 'Objects' section of the memory dump after a manual garbage collection (GC) was triggered. Red boxes highlight the 'Views' and 'Activities' counts, which have decreased from 184 to 656 and 1 to 1 respectively. A red text overlay explains that after fully exiting the program and manually triggering GC, the theoretical counts for Views and Activities should be 0.

Objects			
Views:	656	ViewRootImpl:	1
AppContexts:	14	Activities:	1
Assets:	10	AssetManagers:	8
Local Binders:	42	Proxy Binders:	26
Parcel memory:	9	Parcel count:	31
Death Recipients:	0	OpenSSL Sockets:	0

完全退出程序后，手动触发GC，理论上Views和Activities的数量应该为0.

ps: 在 dump meminfo 信息的时候，需要等内存稳定下然后进行 dump，或者通过手动 gc (利用 android 的 monitor 工具手动 gc 按钮)后进行 dump 操作。

2、跑自动化测试脚本，或者跑Monkey 查看内存的增长曲线（测试检测的方式）。

3、LeakCanary Square 公司开源作品，使用方便，可以直接定位到泄漏的对象，并且给出调用链。

内存泄漏分析，相关工具使用：

分析内存泄漏，第一步得复现问题，然后抓取 **hprof** 文件。

1、**androidStudio Hprof** 分析 **Activity** 泄漏：利用 **Android studio** 的 **Monitors** 的工具抓取 点击 **Start Allocation Tracking** 抓取 **hprof** 文件。具体怎样使用该工具 可参考如下链接。

hprof 分析

例如如下分析结果，存在 **HandlerActivity** 存在泄漏。然后直接参考上面的链接，找出对应引起泄漏的点。下图是 **HandlerActivity** 的内部类释放不了造成 **Activity** 泄漏的。

The screenshot displays the Android Studio interface with the Memory Monitor tool open. The 'Class Name' list on the left shows various system classes. The 'Instance' list in the center shows several instances of **HandlerActivity** (e.g., 0 = {HandlerActivity@378687672}, 1 = {HandlerActivity@378751672}, etc.). On the right, the 'Analysis Results' panel is expanded, showing a list of 'Leaked Activities' with their memory addresses (e.g., 0 = {HandlerActivity@378687672 (0x169250b8)}).

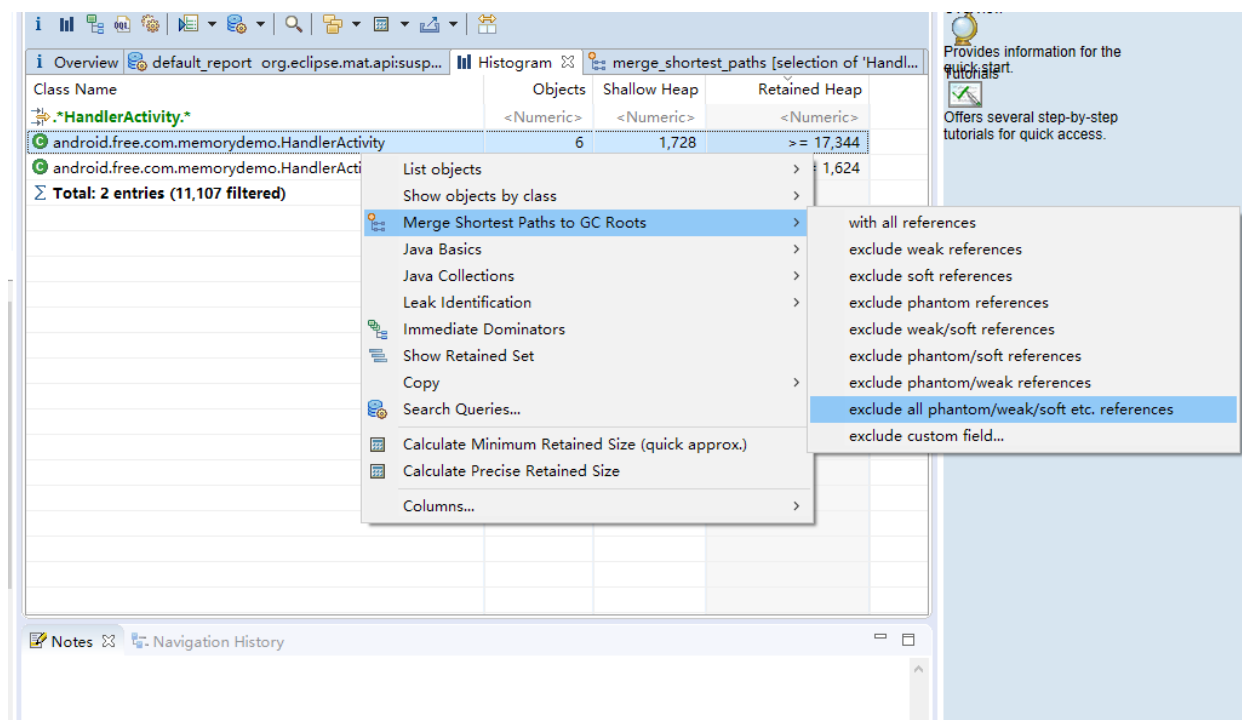
2、**MAT** 使用：**MAT** 工具打开 **hprof** 文件时，需要先利用 **hprof-conv** 工具将 **hprof** 文件转换下,利用如下命令。

```
hprof-conv demo.hprof demo_conv.hprof
```

打开 **hprof** 文件如下：搜索关键字可以搜索出相关对象对象的个数，所占的内存（如下图）。

Class Name	Objects	Shallow Heap	Retained Heap
	<Numeric>	<Numeric>	<Numeric>
android.free.com.memorydemo.HandlerActivity	6	1,728	>= 17,344
android.free.com.memorydemo.HandlerActivity\$InnerThread	6	816	>= 1,624
Total: 2 entries (11,107 filtered)	12	2,544	

右键 Merge shortest paths to gc roots 选择 exclude all phantom/weak/soft etc references 可以定位到 gc root 这样就可以确定是哪个对象没有释放导致 HandlerActivity 不能释放。



对比两个 hprof 文件，查看某个对象的个数对比如下图：


```

        Toast.makeText(HandlerActivity.this,
            "versionCode="+versionCode,Toast.LENGTH_LONG).show();
    }

```

在上面的例子中 `DeviceUtil` 持有了 `Activity` 从而导致其释放不了。

解决方案： 将 `Activity context` 修改成 `Application context` . 因为 `Application` 时全局的，生命周期时和app 生命周期一样的。

非静态内部类造成内存泄漏：

```

public class HandlerActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler);
        startThread();
    }

    class InnerThread extends Thread{
        @Override
        public void run() {
            super.run();
            int index = 0;
            while (index < Integer.MAX_VALUE){
                index ++;
                try {
                    sleep(200);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        }
    }

    private void startThread(){
        InnerThread thread = new InnerThread();
        thread.start();
    }
}

```

如上面代码中 `InnerThread` 会隐式的持有外部类的引用，因为在这里 `InnerThread` 线程的生命周期超过了 `Activity` 的生命周期，当 `finish` 当前 `Activity` 时 `InnerThread` 并不会停止且持有了当前 `Activity` 从而导致 `HandlerActivity` 泄漏。

解决方案： 将内部类修改为静态内部类的方式。如下：

```

static class InnerThread extends Thread{
    @Override
    public void run() {

```

```

        super.run();
        int index = 0;
        while (index < Integer.MAX_VALUE){
            index ++;
            try {
                sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

```

匿名内部类造成的内存泄漏

```

public class HandlerActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_handler);
        //startThread();
        Handler handler = new Handler(){
            @Override
            public void handleMessage(Message msg) {
                final int what = msg.what;
                if(what == 0){
                    Toast.makeText(HandlerActivity.this,"receive message",Toa
st.LENGTH_LONG).show();
                }
                super.handleMessage(msg);
            }
        };

        Message message = handler.obtainMessage(0);
        handler.sendMessageDelayed(message,1000*10);
    }
}

```

在这里Message 会在主线程中存在10s，且Message 会持有handler 对象，而handler 会隐式持有Activity，从而导致Activity 泄漏。

修改方案： 将匿名内部类修改成静态内部类即可，参考上面的例子。

Native 泄漏：

在android 中 Native 泄漏都大部分是通过 jni 调用 Native 方法，所以只需要检查 Java 端调用 JNI 的地方即可。

总结：一般来说，内存泄漏都是因为泄漏对象的引用被传递到该对象的范围之外，或者说内存泄漏是因为持有对象的长期引用，导致对象无法被 GC 回收。为了避免这种情况，我们可以选择在对象生命周期结束的时候，解除绑定，将引用置为空，或者使用弱引用。