Android 内存泄漏调试

一、概述

Java 编程中经常容易被忽视，但本身又十分重要的一个问题就是内存使用的问题。Android 应用主要使用Java 语言编写，因此这个问题也同样会在Android 开发中出现。本文不对Java 编程问题做探讨，而是对于在Android 中，特别是应用开发中的此类问题进行整理。

二、Android(Java)中常见的容易引起内存泄漏的不良代码

Android 主要应用在嵌入式设备当中，而嵌入式设备由于一些众所周知的条件限制，通常都不会有很高的配置，特别是内存是比较有限的。如果我们编写的代码当中有太多的对内存使用不当的地方，难免会使得我们的设备运行缓慢，甚至是死机。为了能够使得Android 应用程序安全且快速的运行，Android 的每个应用程序都会使用一个专有的Dalvik 虚拟机实例来运行，它是由Zygote 服务进程孵化出来的，也就是说每个应用程序都是在属于自己的进程中运行的。一方面，如果程序在运行过程中出现了内存泄漏的问题，仅仅会使得自己的进程被kill 掉，而不会影响其他进程（如果是system_process 等系统进程出问题的话，则会引起系统重启）。另一方面Android 为不同类型的进程分配了不同的内存使用上限，如果应用进程使用的内存超过了这个上限，则会被系统视为内存泄漏，从而被kill 掉。Android 为应用进程分配的内存上限如下所示：

位置： /ANDROID_SOURCE/system/core/rootdir/init.rc 部分脚本

```
# Define the oom_adj values for the classes of processes that can be
# killed by the kernel. These are used in ActivityManagerService.
setprop ro.FOREGROUND_APP_ADJ 0
setprop ro.VISIBLE_APP_ADJ 1
setprop ro.SECONDARY_SERVER_ADJ 2
setprop ro.BACKUP_APP_ADJ 2
setprop ro.HOME_APP_ADJ 4
setprop ro.HIDDEN_APP_MIN_ADJ 7
setprop ro.CONTENT_PROVIDER_ADJ 14
setprop ro.EMPTY_APP_ADJ 15
# Define the memory thresholds at which the above process classes will
# be killed. These numbers are in pages (4k).
setprop ro.FOREGROUND_APP_MEM 1536
setprop ro.VISIBLE_APP_MEM 2048
setprop ro.SECONDARY_SERVER_MEM 4096
setprop ro.BACKUP_APP_MEM 4096
setprop ro.HOME_APP_MEM 4096
setprop ro.HIDDEN_APP_MEM 5120
```

```
setprop ro.CONTENT_PROVIDER_MEM 5632
setprop ro.EMPTY_APP_MEM 6144
# Write value must be consistent with the above properties.
# Note that the driver only supports 6 slots, so we have HOME_APP at the
# same memory level as services.
write /sys/module/lowmemorykiller/parameters/adj 0,1,2,7,14,15
write /proc/sys/vm/overcommit_memory 1
write /proc/sys/vm/min_free_order_shift 4
write /sys/module/lowmemorykiller/parameters/minfree
1536,2048,4096,5120,5632,6144
# Set init its forked children's oom_adj.
write /proc/1/oom_adj -16
```

正因为我们的应用程序能够使用的内存有限，所以在编写代码的时候需要特别注意内存使用问题。如下是一些常见的内存使用不当的情况。

（一）查询数据库没有关闭游标

描述：

程序中经常会进行查询数据库的操作，但是经常会有使用完毕Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

示例代码：

```
Cursor cursor = getContentResolver().query(uri ...);
if (cursor.moveToNext()) {
... ...
}
```

修正示例代码：

```
Cursor cursor = null;
try {
cursor = getContentResolver().query(uri ...);
if (cursor != null && cursor.moveToNext()) {
... ...
}
} finally {
if (cursor != null) {
try {
cursor.close();
} catch (Exception e) {
//ignore this
}
}
}
```

（二）构造Adapter时，没有使用缓存的convertView

描述：

以构造ListView 的BaseAdapter 为例，在BaseAdapter 中提高了方法：public View getView(int position, View convertView, ViewGroup parent)来向ListView提供每一个item 所需要的view 对象。初始时ListView 会从BaseAdapter 中根据当前的屏幕布局实例化一定数量的view 对象，同时ListView 会将这些view 对象缓存起来。当向上滚动ListView 时，原先位于最上面的list item 的view 对象会被回收，然后被用来构造新出现的最下面的list item。这个构造过程就是由getView()方法完成的，getView()的第二个形参View convertView 就是被缓存起来的list item 的view 对象(初始化时缓存中没有view对象则convertView 是null)。

由此可以看出，如果我们不去使用convertView，而是每次都在getView()中重新实例化一个View 对象的话，即浪费资源也浪费时间，也会使得内存占用越来越大。ListView 回收listitem 的view 对象的过程可以查看：

android.widget.AbsListView.java --> void addScrapView(View scrap) 方法。

示例代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
View view = new Xxx(...);
... ...
return view;
}
```

修正示例代码：

```
public View getView(int position, View convertView, ViewGroup parent) {
View view = null;
if (convertView != null) {
view = convertView;
populate(view, getItem(position));
...
} else {
view = new Xxx(...);
...
}
return view;
}
```

(三) Bitmap对象不在使用时调用recycle()释放内存

描述：

有时我们会手工的操作Bitmap 对象，如果一个Bitmap 对象比较占内存，当它不在被使用的时候，可以调用Bitmap.recycle()方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。可以看一下代码中的注释：

```
/**
* Free up the memory associated with this bitmap's pixels, and mark the
* bitmap as "dead", meaning it will throw an exception if getPixels() or
* setPixels() is called, and will draw nothing. This operation cannot be
* reversed, so it should only be called if you are sure there are no
```

```
* further uses for the bitmap. This is an advanced call, and normally need
* not be called, since the normal GC process will free up this memory when
* there are no more references to this bitmap.
*/
```

（四）释放对象的引用

描述：

这种情况描述起来比较麻烦，举两个例子进行说明。

示例A：

假设有如下操作

```
public class DemoActivity extends Activity {
... ...
private Handler mHandler = ...
private Object obj;
public void operation() {
obj = initObj();
...
[Mark]
mHandler.post(new Runnable() {
public void run() {
useObj(obj);
}
});
}
}
```

我们有一个成员变量obj，在operation()中我们希望能够将处理obj 实例的操作post 到某个线程的MessageQueue 中。在以上的代码中，即便是mHandler 所在的线程使用完了obj所引用的对象，但这个对象仍然不会被垃圾回收掉，因为DemoActivity.obj 还保有这个对象的引用。所以如果在DemoActivity 中不再使用这个对象了，可以在[Mark]的位置释放对象的引用，而代码可以修改为：

```
... ...
public void operation() {
obj = initObj();
...
final Object o = obj;
obj = null;
mHandler.post(new Runnable() {
public void run() {
useObj(o);
}
}
}
... ...
```

示例B:

假设我们希望在锁屏界面(LockScreen)中，监听系统中的电话服务以获取一些信息(如信号强度等)，则可以在LockScreen 中定义一个PhoneStateListener 的对象，同时将它注册到TelephonyManager 服务中。对于LockScreen 对象，当需要显示锁屏界面的时候就会创建一个LockScreen 对象，而当锁屏界面消失的时候LockScreen 对象就会被释放掉。

但是如果在释放LockScreen 对象的时候忘记取消我们之前注册的PhoneStateListener 对象，则会导致LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量的LockScreen 对象没有办法被回收而引起OutOfMemory,使得system_process 进程挂掉。

总之当一个生命周期较短的对象A,被一个生命周期较长的对象B 保有其引用的情况下，在A 的生命周期结束时，要在B 中清除掉对A 的引用。

（五）其他

Android 应用程序中最典型的需要注意释放资源的情况是在Activity 的生命周期中，在onPause()、onStop()、onDestroy()方法中需要适当的释放资源的情况。由于此情况很基础，在此不详细说明，具体可以查看官方文档对Activity 生命周期的介绍，以明确何时应该释放哪些资源。

三、内存监测工具DDMS --> Heap

无论怎么小心，想完全避免bad code 是不可能的，此时就需要一些工具来帮助我们检查代码中是否存在会造成内存泄漏的地方。Android tools 中的DDMS 就带有一个很不错的内存监测工具Heap(这里我使用eclipse 的ADT 插件，并以真机为例，在模拟器中的情况类似)。用Heap 监测应用进程使用内存情况的步骤如下：

1. 启动eclipse 后，切换到DDMS 透视图，并确认Devices 视图、Heap 视图都是打开的；
2. 将手机通过USB 链接至电脑，链接时需要确认手机是处于“USB 调试”模式，而不是作为“Mass Storage”；
3. 链接成功后，在DDMS 的Devices 视图中将会显示手机设备的序列号，以及设备中正在运行的部分进程信息；
4. 点击选中想要监测的进程，比如system_process 进程；
5. 点击选中Devices 视图界面中最上方一排图标中的“Update Heap”图标；
6. 点击Heap 视图中的“Cause GC”按钮；
7. 此时在Heap 视图中就会看到当前选中的进程的内存使用量的详细情况［如图所示］。

说明：

a) 点击“Cause GC”按钮相当于向虚拟机请求了一次gc 操作；

b) 当内存使用信息第一次显示以后，无须再不断的点击“Cause GC”，Heap 视图界面会定时刷新，在对应用的不断的操作过程中就可以看到内存使用的变化；

c) 内存使用信息的各项参数根据名称即可知道其意思，在此不再赘述。

如何才能知道我们的程序是否有内存泄漏的可能性呢。这里需要注意一个值：Heap 视图中部有一个Type 叫做data object，即数据对象，也就是我们的程序中大量存在的类类型的对象。在data object 一行中有一列是“Total Size”，其值就是当前进程中所有Java 数据对象的内存总量，一般情况下，这个值的大小决定了是否会有内存泄漏。可以这样判断：

a) 不断的操作当前应用，同时注意观察data object 的Total Size 值；

b) 正常情况下Total Size 值都会稳定在一个有限的范围内，也就是说由于程序中的的代码

良好，没有造成对象不被垃圾回收的情况，所以说虽然我们不断的操作会不断的生成很多对象，而在虚拟机不断的进行GC 的过程中，这些对象都被回收了，内存占用量会会落到一个稳定的水平；

c) 反之如果代码中存在没有释放对象引用的情况，则data object 的Total Size 值在每次GC后不会有明显的回落，随着操作次数的增多Total Size 的值会越来越大，直到到达一个上限后导致进程被kill 掉。

d) 此处已system_process 进程为例，在我的测试环境中system_process 进程所占用的内存的data object 的Total Size 正常情况下会稳定在2.2~2.8 之间，而当其值超过3.55 后进程就会被kill。

总之，使用DDMS 的Heap 视图工具可以很方便的确认我们的程序是否存在内存泄漏的可能性。

四、内存分析工具MAT(MemoryAnalyzer Tool)

如果使用DDMS 确实发现了我们的程序中存在内存泄漏，那又如何定位到具体出现问题的代码片段，最终找到问题所在呢？如果从头到尾的分析代码逻辑，那肯定会把人逼疯，特别是在维护别人写的代码的时候。这里介绍一个极好的内存分析工具-- Memory Analyzer Tool(MAT)。

MAT 是一个Eclipse 插件，同时也有单独的RCP 客户端。官方下载地址、MAT 介绍和详细的使用教程请参见：www.eclipse.org/mat，在此不进行说明了。另外在MAT 安装后的帮助文档里也有完备的使用教程。在此仅举例说明其使用方法。我自己使用的是MAT 的eclipse 插件，使用插件要比RCP 稍微方便一些。

使用MAT 进行内存分析需要几个步骤，包括：生成.hprof 文件、打开MAT 并导入.hprof 文件、使用MAT 的视图工具分析内存。以下详细介绍。

(一) 生成.hprof文件

生成.hprof 文件的方法有很多，而且Android 的不同版本中生成.hprof 的方式也稍有差别，我使用的版本的是2.1，各个版本中生成.prof 文件的方法请参考：
http://android.git.kernel.org/?p=platform/dalvik.git;a=blob_plain;f=docs/heapprofiling.html;hb=HEAD。

1. 打开eclipse 并切换到DDMS 透视图，同时确认Devices、Heap 和logcat 视图已经打开了；
2. 将手机设备链接到电脑，并确保使用“USB 调试”模式链接，而不是“Mass Storage“模式；
3. 链接成功后在Devices 视图中就会看到设备的序列号，和设备中正在运行的部分进程；
4. 点击选中想要分析的应用的进程，在Devices 视图上方的一行图标按钮中，同时选中“Update Heap”和“Dump HPROF file”两个按钮；
5. 这是DDMS 工具将会自动生成当前选中进程的.hprof 文件，并将其进行转换后存放在sdcard 当中，如果你已经安装了MAT 插件，那么此时MAT 将会自动被启用，并开始对.hprof 文件进行分析；

注意：第4 步和第5 步能够正常使用前提是我们需要有sdcard，并且当前进程有向sdcard 中写入的权限(WRITE_EXTERNAL_STORAGE)，否则.hprof 文件不会被生成，在logcat 中会显

示诸如
ERROR/dalvikvm(8574): hprof: can't open /sdcard/com.xxx.hprof-hptemp: Permission denied.的信息。
如果我们没有sdcard，或者当前进程没有向sdcard 写入的权限（如system_process）， 那我们可以这样做：
6. 在当前程序中，例如framework 中某些代码中，可以使用android.os.Debug 中的：
public static void dumpHprofData(String fileName) throws IOException
方法，手动的指定.hprof 文件的生成位置。例如：

```
xxxButton.setOnClickListener(new View.OnClickListener() {
public void onClick(View view) {
android.os.Debug.dumpHprofData("/data/temp/myapp.hprof");
... ...
}
}
```

上述代码意图是希望在xxxButton 被点击的时候开始抓取内存使用信息，并保存在我们指定的位置：/data/temp/myapp.hprof，这样就没有权限的限制了，而且也无须用sdcard。但要保证/data/temp 目录是存在的。这个路径可以自己定义，当然也可以写成sdcard 当中的某个路径。

（二）使用MAT导入.hprof文件

1. 如果是eclipse 自动生成的.hprof 文件，可以使用MAT 插件直接打开（可能是比较新的ADT才支持）；
2. 如果eclipse 自动生成的.hprof 文件不能被MAT 直接打开， 或者是使用android.os.Debug.dumpHprofData()方法手动生成的.hprof 文件，则需要将.hprof 文件进行转换，转换的方法：
例如我将.hprof 文件拷贝到PC 上的/ANDROID_SDK/tools 目录下，并输入命令hprofconv xxx.hprof yyy.hprof，其中xxx.hprof 为原始文件，yyy.hprof 为转换过后的文件。转换过后的文件自动放在/ANDROID_SDK/tools 目录下。OK，到此为止，.hprof 文件处理完毕，可以用来分析内存泄露情况了。
3. 在Eclipse 中点击Windows->Open Perspective->Other->Memory Analyzer，或者打Memory Analyzer Tool 的RCP。在MAT 中点击File->Open File，浏览并导入刚刚转换而得到的.hprof文件。

（三）使用MAT的视图工具分析内存

导入.hprof 文件以后，MAT 会自动解析并生成报告，点击Dominator Tree，并按Package分组，选择自己所定义的Package 类点右键，在弹出菜单中选择List objects->With incoming
references。这时会列出所有可疑类，右键点击某一项，并选择Path to GC Roots -> exclude weak/soft references，会进一步筛选出跟程序相关的所有有内存泄露的类。据此，可以追踪到代码中的某一个产生泄露的类。
MAT 的界面如下图所示。
具体的分析方法在此不做说明了，因为在MAT 的官方网站和客户端的帮助文档中有十分详尽的介绍。

了解MAT 中各个视图的作用很重要，例如 www.eclipse.org/mat/about/screenshots.php 中介绍的。

总之使用MAT 分析内存查找内存泄漏的根本思路，就是找到哪个类的对象的引用没有被释放，找到没有被释放的原因，也就可以很容易定位代码中的哪些片段的逻辑有问题了。