Tips for Optimizing Android* Application Memory Usage

By Bin Zhu (Intel) (https://software.intel.com/en-us/user/1109850), Updated February 9, 2015 Translate

Introduction

Memory allocation and de-allocation in Android* always comes at a cost. The Chinese saying “Easy to be luxurious from frugal, hard to be frugal from luxurious” and is an appropriate turn of phrase for memory usage.

Let’s imagine a worst-case scenario of compiling an application with millions of lines of code, when suddenly an out of memory (OOM) crash is triggered. You start debugging the application and analyzing the hprof file. Luckily, you locate the root cause and fix the memory killer. But sometimes you are unlucky and find that so many tiny member variables and temporaries are allocating memory that there is no simple fix, meaning you have to refactor code, which involves potential risk, just to save kilobytes or even bytes of memory.

This article introduces Android Memory Management and explains various aspects that play a role in the management system. Additionally, improving memory management, detecting and avoiding memory leaks, and analyzing memory usage are covered.

Android Memory Management

Android uses paging and mmap instead of providing swap space, which means any memory your application touches cannot be paged out unless you release all references.

The Dalvik* Virtual Machine’s heap size for application processes is limited. Applications start up with 2 MB, and the maximum allocation, marked as "largeHeap," is limited to 36 MB (depending on the specific device configuration). Examples of large heap applications are Photo/Video Editor, Camera, Gallery, and Home Screen.

Android stores background application processes in a LRU cache. When the system runs low on memory, it will kill processes according to the LRU strategy, but it will also consider which application is the largest memory consumer. Currently the maximum background process count is 20 (depending on the specific device configuration). If you need your app to live longer in the background, de-allocate unnecessary memory before moving to the background and the Android system will less likely generates error message or even terminates the app.

How to Improve Memory Usage

Android is a worldwide mobile platform and millions of Android developers are dedicated to building stable and scalable applications. Here is a list of tips and best practices for improving memory usage in Android applications:

1. Be careful about using a design pattern with "abstraction". Although from the point of view of design pattern, abstraction can help to build more flexible software architect. In mobile world, abstraction may involve side effect for its extra code to be executed, which will cost more time and memory. Unless abstraction can provide your application a significant benefit, you would be better not to use it.
2. Avoid using "enum". Enum will double memory allocation than ordinary static constant, so do not use it.
3. Try to use the optimized SparseArray, SparseBooleanArray, and LongSparseArray containers instead of HashMap. HashMap allocates an entry object during every mapping which is a memory inefficient action, also the low performance behavior - “autoboxing/unboxing” is spread all over the usage. Instead, SparseArray-like containers map keys into plain array. But please remember that

these optimized containers are not suitable for large numbers of items, when executing add/remove /search actions, they are slower than Hashmap if your data set is over thousands of records.
4. Avoid creating unnecessary objects. Do not allocate memory especially for short-term temporary objects if you can avoid it, and garbage collection will occur less when fewer objects are created.
5. Check the available heap of your application. Invoke ActivityManager::getMemoryClass() to query how many heap(MB) is available for your application. OutofMemory Exception will occur if you try to allocate more memory than is available. If your application declares a "largeHeap" in AndroidManifest.xml, you can invoke ActivityManager::getLargeMemoryClass() to query an estimated large heap size.
6. Coordinate with the system by implementing onTrimMemory() callback. Implement ComponentCallbacks2::onTrimMemory(int) in your Activity/Service/ContentProvider to gradually release memory according to latest system constraints. The onTrimMemory(int) helps overall system response speed, but alsokeep your process alive longer in the system.
   When TRIM_MEMORY_UI_HIDDEN occurs, it means all the UI in your application has been hidden and you need to free UI resources. When your application is foreground, you may receive TRIM_MEMORY_RUNNING[MODERATE/LOW/CRITICAL], or in the background you may receive TRIM_MEMORY_[BACKGROUND/MODERATE/COMPLETE]. You can free non-critical resources based on the strategy to release memory when system memory is tight.

7. Services should be used with caution. If you need a service to run a job in the background, avoid keeping it running unless it's actively performing a task. Try to shorten its lifespan by using an IntentService, which will finish itself when it's done handling the intent. Services should be used with the caution to never leave one running when it's not needed. Worst case, the overall system performance will be poor and users will find your app and uninstall it (if possible).
   But if you are building an app that needs to run for a long period of time, e.g., a music player service, you should split it into two processes: one for the UI and the other for the background service by setting the property "android:process" for your Service in AndroidManifest.xml. The resources in the UI process can be released after hidden, while the background playback service is still running. Keep in mind that the background service process MUST NOT touch any UI; otherwise, the memory allocation will be doubled or tripled!

8. External libraries should be used carefully. External libraries are often written for non-mobile device and can be inefficient in Android. You must take into account the effort in porting and optimizing the library for mobile before you decide to use it. If you are using a library for only one or two features from its thousands of other uses, it may be best to implement it by yourself.
9. Use bitmaps with correct resolution. Load a bitmap at the resolution you need, or scale it down if the original bitmap is a higher resolution.
10. Use Proguard* and zipalign. The Proguard tool removes unused code and obfuscates classes, methods and fields. It will compact your code to reduce required RAM pages to be mapped. The zipalign tool will re-align your APK. More memory will be needed if zipalign not running because resource files cannot mapped from the APK.

How to Avoid Memory Leaks

Use memory carefully with above tips can bring benefit for your application incrementally, and make your application stay longer in system. But all benefit will lost if memory leakage happens. Here are some familiar potential leakage that developer needs to keep in mind.

1. Remember to close the cursor after querying the database. If you want to keep the cursor open long-term, you must use it carefully and close it as soon as the database task finished.
2. Remember to call unregisterReceiver() after calling registerReceiver().
3. Avoid Context leakage. If you declare a static member variable "Drawable" in your Activity, and then call view.setBackground(drawable) in onCreate(), after screen rotate, a new Activity instance will be created and the old Activity instance can never be de-allocated because drawable has set the view as callback and view has a reference to Activity (Context). A leaked Activity instance means a significant amount of memory, which will cause OOM easily.
   There are two ways to avoid this kind of leakage:

   - Do not keep long-lived references to a context-activity. A reference to an activity should have the same life cycle as the activity itself.
   - Try using the context-application instead of a context-activity.

4. Avoid non-static inner classes in an Activity. In Java, non-static anonymous classes hold an implicit reference to their enclosing class. If you're not careful, storing this reference can result in the Activity being retained when it would otherwise be eligible for garbage collection. So instead, use a

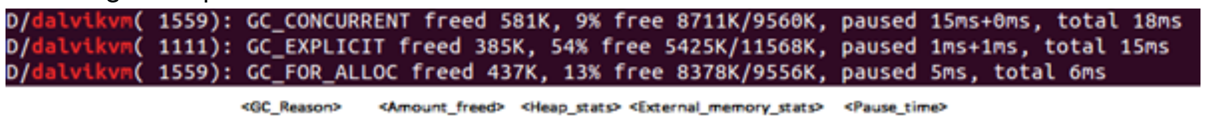
static inner class and make a weak reference to the activity inside.

5. Be careful about using Threads. Threads in Java are garbage collection roots; that is, the Dalvik Virtual Machine (DVM) keeps hard references to all active threads in the runtime system, and as a result, threads that are left running will never be eligible for garbage collection. Java threads will persist until either they are explicitly closed or the entire process is killed by the Android system. Instead, the Android application framework provides many classes designed to make background threading easier for developers:
    - Use Loader instead of a thread for performing short-lived asynchronous background queries in conjunction with the Activity lifecycle.
    - Use Service and report the results back to the Activity using a BroadcastReceiver.
    - Use AsyncTask for short-lived operations.

How to Analyze Memory Usage

To know more about memory usage statistics in online/offline, you can check the Android main log by using logcat command in Android Debug Bridge (ADB), dump memory info for specific package name, or using other tools like Dalvik Debug Monitor Server (DDMS) and Memory Analyzer Tool (MAT), here are some brief introductions about the ways to analyze the memory usage of your application.

1. Understand garbage collection (GC) log messages for the Dalvik Virtual Machine, as shown in the following example and definition:

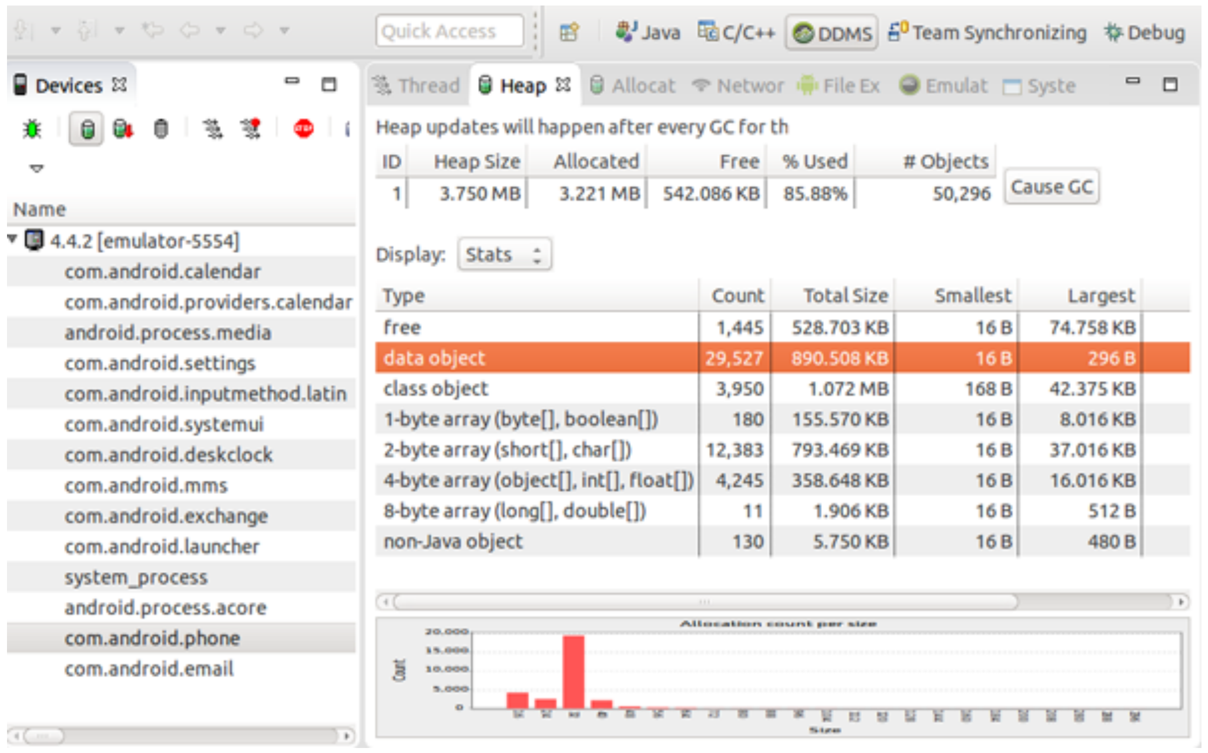


```
D/dalvikvm( 1559): GC_CONCURRENT freed 581K, 9% free 8711K/9560K, paused 15ms+0ms, total 18ms
D/dalvikvm( 1111): GC_EXPLICIT freed 385K, 54% free 5425K/11568K, paused 1ms+1ms, total 15ms
D/dalvikvm( 1559): GC_FOR_ALLOC freed 437K, 13% free 8378K/9556K, paused 5ms, total 6ms
```

    - GC Reason: What triggered the garbage collection and what kind of collection is it? Reasons may include:
        - GC_CONCURRENT: A concurrent garbage collection that frees up memory as your heap begins to fill up.
        - GC_FOR_ALLOC: A garbage collection occurs because your app attempted to allocate memory when your heap was already full, so the system had to stop your app and reclaim memory.
        - GC_HPROF_DUMP_HEAP: A garbage collection that occurs when you create an HPROF file to analyze your heap.
        - GC_EXPLICIT: An explicit garbage collection, such as when you call gc() (which you should avoid calling and instead trust the garbage collector to run when needed).
    - Amount freed: The amount of memory reclaimed from this garbage collection.
    - Heap stats: Percentage free and (number of live objects) / (total heap size).
    - External memory stats: Externally allocated memory on API level 10 and lower (amount of allocated memory) / (limit at which collection will occur).
    - Pause time: Larger heaps will have larger pause times. Concurrent pause times show two pauses: one at the beginning of the collection and another near the end.

    The larger the GC log, the more memory allocation/de-allocation occurred in your application, which also means the user experience is choked.

2. Use DDMS to view heap updates and track allocation.
    It's convenient to check real-time heap allocation of the specific process through DDMS. Try interacting with your application and watch the heap allocation update in the "Heap" tab. This can help you identify which actions are likely using too much memory. The "Allocation Tracker" tab shows all recent allocations, providing information including the type of object, allocated in which thread, class, and file and at which line. For more information about using DDMS for heap analysis, please refer to the References section at the end of this article. The following screenshot is showing a running DDMS which including current processes and memory heap statistics for specific process.



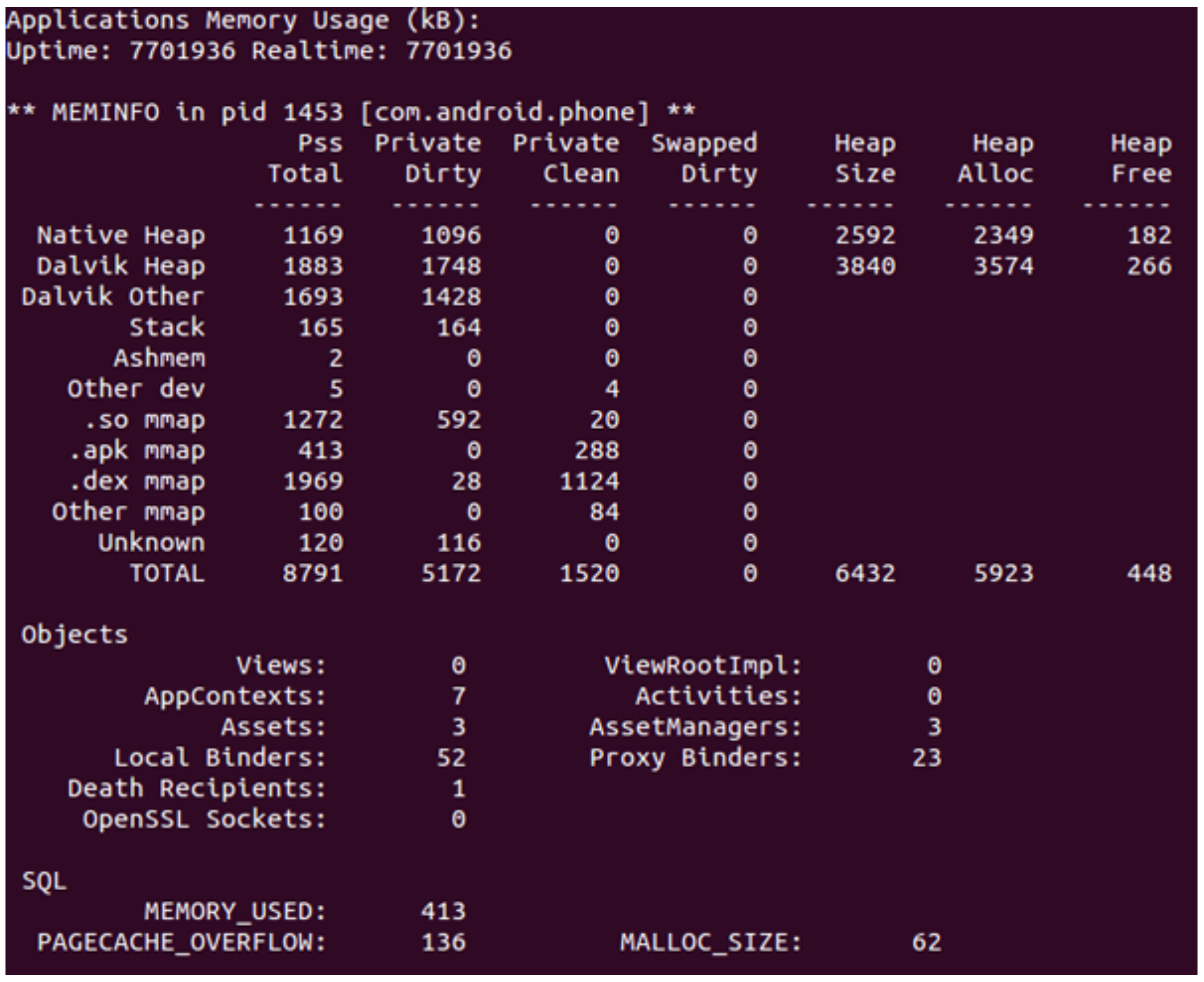
Name
4.4.2 [emulator-5554]
com.android.calendar
com.android.providers.calendar
android.process.media
com.android.settings
com.android.inputmethod.latin
com.android.systemui
com.android.deskclock
com.android.mms
com.android.exchange
com.android.launcher
system_process
android.process.acore
com.android.phone
com.android.email

Heap updates will happen after every GC for th

ID	Heap Size	Allocated	Free	% Used	# Objects
1	3.750 MB	3.221 MB	542.086 KB	85.88%	50,296

Type	Count	Total Size	Smallest	Largest
free	1,445	528.703 KB	16 B	74.758 KB
data object	29,527	890.508 KB	16 B	296 B
class object	3,950	1.072 MB	168 B	42.375 KB
1-byte array (byte[], boolean[])	180	155.570 KB	16 B	8.016 KB
2-byte array (short[], char[])	12,383	793.469 KB	16 B	37.016 KB
4-byte array (object[], int[], float[])	4,245	358.648 KB	16 B	16.016 KB
8-byte array (long[], double[])	11	1.906 KB	16 B	512 B
non-Java object	130	5.750 KB	16 B	480 B

3. View overall memory allocations.
   By executing the adb command: “adb shell dumpsys meminfo <package_name>”, you can see all of your application's current allocations, measured in kilobytes.

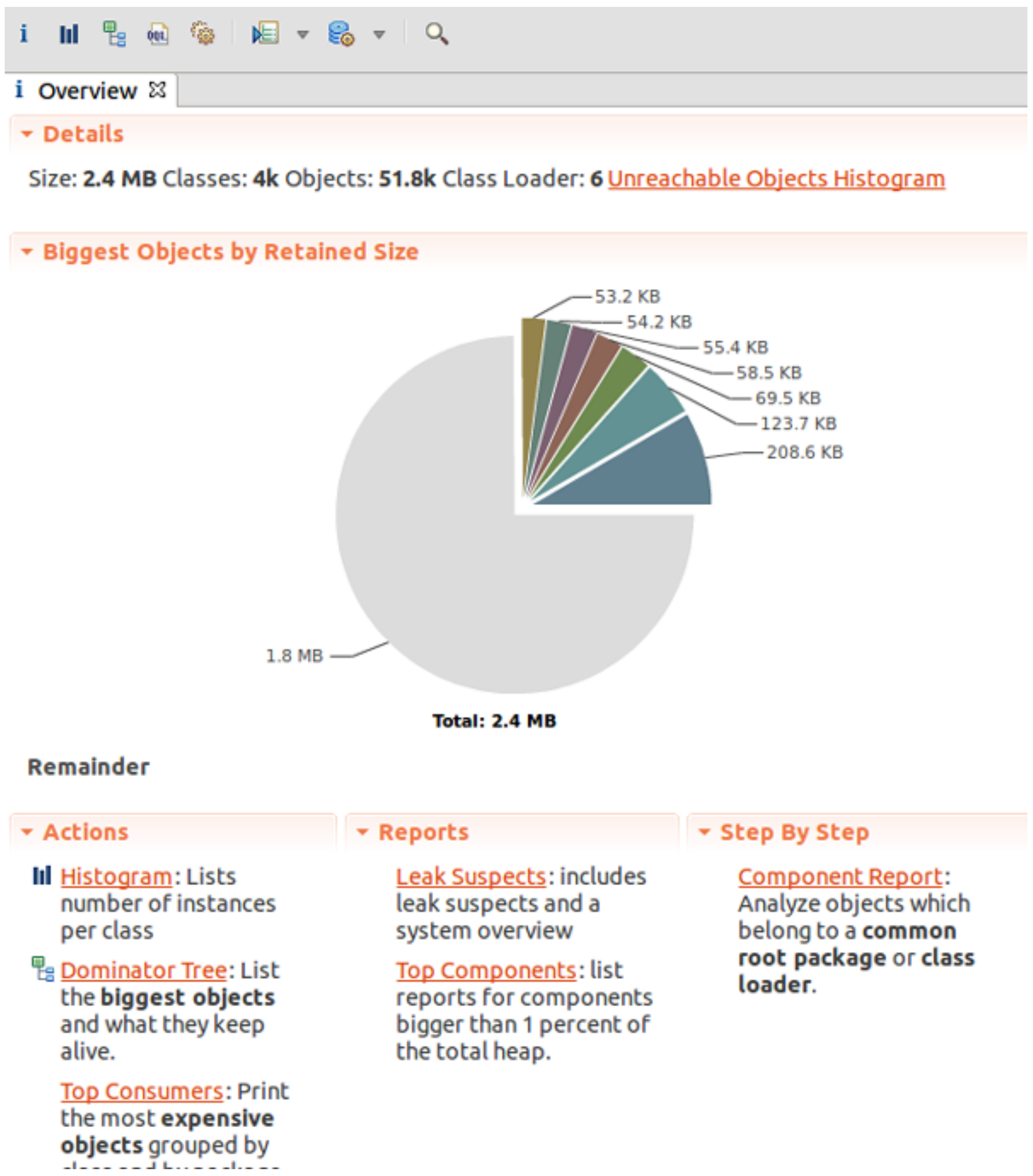


```
Applications Memory Usage (kB):
Uptime: 7701936 Realtime: 7701936

** MEMINFO in pid 1453 [com.android.phone] **
                   Pss  Private  Private  Swapped     Heap     Heap     Heap
                 Total    Dirty    Clean    Dirty     Size    Alloc     Free
                ------   ------   ------   ------   ------   ------   ------
  Native Heap     1169     1096        0        0     2592     2349      182
  Dalvik Heap     1883     1748        0        0     3840     3574      266
 Dalvik Other     1693     1428        0        0
        Stack      165      164        0        0
       Ashmem        2        0        0        0
    Other dev        5        0        4        0
     .so mmap     1272      592       20        0
    .apk mmap      413        0      288        0
    .dex mmap     1969       28     1124        0
   Other mmap      100        0       84        0
      Unknown      120      116        0        0
        TOTAL     8791     5172     1520        0     6432     5923      448

 Objects
               Views:        0         ViewRootImpl:        0
         AppContexts:        7           Activities:        0
              Assets:        3        AssetManagers:        3
       Local Binders:       52        Proxy Binders:       23
    Death Recipients:        1
     OpenSSL Sockets:        0

 SQL
         MEMORY_USED:      413
  PAGECACHE_OVERFLOW:      136          MALLOC_SIZE:       62
```

Generally, you should be concerned with only the "Pss Total" and "Private Dirty" columns. The "Pss Total" includes all Zygote allocations (weighted by their sharing across processes, as described in the PSS definition above). The "Private Dirty" number is the actual RAM committed to your application’s heap, your own allocations, and any Zygote allocation pages that have been modified since forking your app’s process from Zygote.

Additionally, the "ViewRootImpl" shows the number of root views that are active in your process. Each root view is associated with a window, so this can help you identify memory leaks involving dialogs or other windows. The "AppContexts" and "Activities" show the number of application Context and Activity objects that currently live in your process. This can be useful to quickly identify leaked Activity objects that can’t be garbage collected due to static references on them, which is common. These objects often have a lot of other allocations associated with them and are a good way to track large memory leaks.

4. Capture a heap dump and analyze it with the Eclipse* Memory Analyzer Tool (MAT).
   You can directly capture a heap dump by using DDMS or calling Debug::dumpHprofData() in your source code for more precise results. Then you need to use the hprof-conv tool to generate the converted HPROF file. The following screenshot is the memory analyze result showing in the MAT.



Size: **2.4 MB** Classes: **4k** Objects: **51.8k** Class Loader: **6** Unreachable Objects Histogram

Remainder

Histogram: Lists number of instances per class

Dominator Tree: List the **biggest objects** and what they keep alive.

Top Consumers: Print the most **expensive objects** grouped by

Leak Suspects: includes leak suspects and a system overview

Top Components: list reports for components bigger than 1 percent of the total heap.

Component Report: Analyze objects which belong to a **common root package** or **class loader**.

Summary

To build more memory friendly applications, Android developers need to have a basic understanding of Android memory management. Developers should practice efficient memory usage, use analysis tools, and implement the tips provided in this paper. It is better to build a stable and scalable application first, rather than applying fixes during the implementation period.

References

1. http://developer.android.com/training/articles/memory.html (http://developer.android.com/training/articles/memory.html)
2. https://developer.android.com/tools/debugging/debugging-memory.html (https://developer.android.com/tools/debugging/debugging-memory.html)
3. http://developer.android.com/training/articles/perf-tips.html (http://developer.android.com/training/articles/perf-tips.html)
4. http://android-developers.blogspot.co.uk/2009/01/avoiding-memory-leaks.html (http://android-developers.blogspot.co.uk/2009/01/avoiding-memory-leaks.html)
5. http://developer.android.com/tools/debugging/ddms.html (http://developer.android.com/tools/debugging/ddms.html)
6. http://www.curious-creature.com/2008/12/18/avoid-memory-leaks-on-android/comment-page-1/ (http://www.curious-creature.com/2008/12/18/avoid-memory-leaks-on-android/comment-page-1/)
7. https://eclipse.org/mat/ (https://eclipse.org/mat/)

About the Author

Bin Zhu is an application engineer in the Intel® Atom™ Processor Mobile Enabling Team, Developer Relations Division, Software and Solutions Group (SSG). He is responsible for Android app enabling on Intel Atom processors and focuses on multimedia technologies for the X86 Android platforms.

- **Hardware Developers**
  - Resource and Design Center
- **Open Source**
  - 01.org
- **Manage Your Tools**
  - Download Center
- **Stay Up-to-Date**
  - Forums