

Filter By Markets

- Networks (/blog /term/Networks)
- Multimedia (/blog /term/Multimedia)
- Retail (/blog/term/Retail)
- Automotive (/blog /term/Automotive)
- IoT (/blog/term/IoT)

Receive new Insights

Enter your email address

Subscribe

Filter by Tags

- Heavy Reading, Light Reading, Embedded Android Technology Trends, Wireless Networks Technology Trends, Android Devices At Home, Android For Business, Infographics, Android Medical Device Development, Android Labs, Demos, connected cars, Security, Wireless Technology For Business (https://hsc.com /Blog/Tag/Wireless-Technology-For-Business)

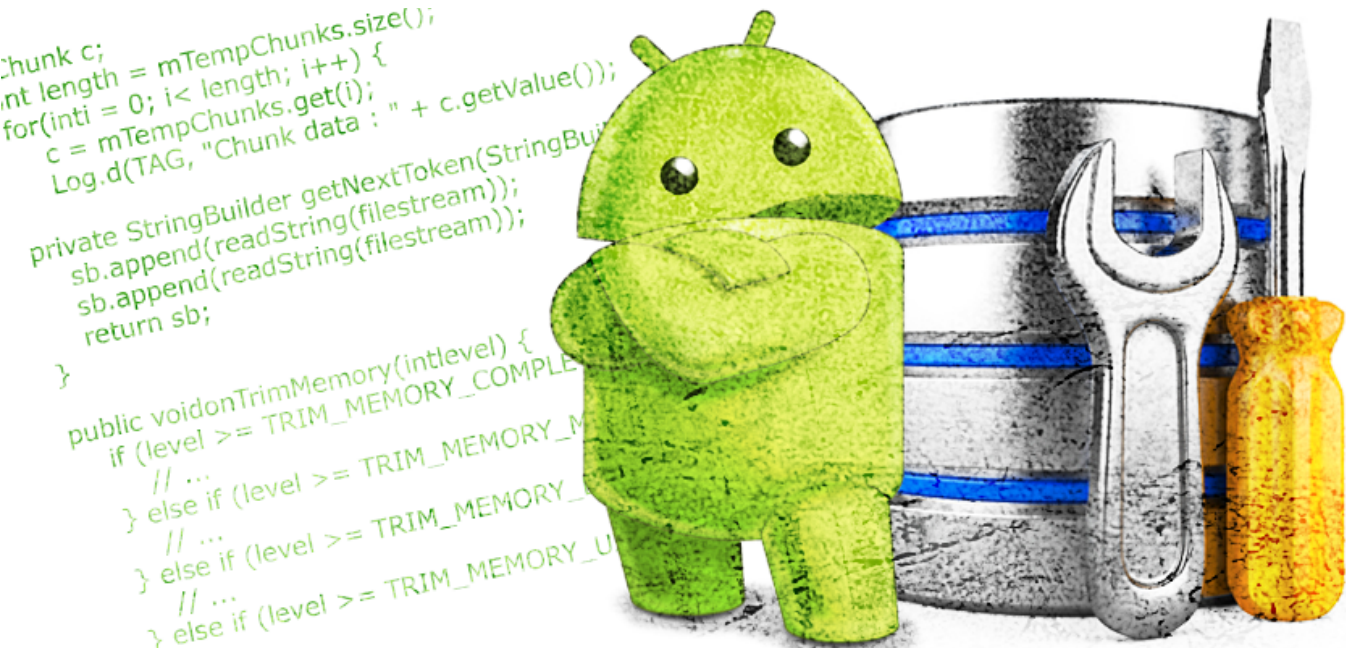
All Entries (https://hsc.com /Blog)

- May 2017 (https://hsc.com /Blog/Year/2017/Month/5) (2)
- April 2017 (https://hsc.com /Blog/Year/2017/Month/4) (2)
- March 2017 (https://hsc.com /Blog/Year/2017/Month/3) (2)
- February 2017 (https://hsc.com /Blog/Year /2017/Month/2) (2)
- January 2017 (https://hsc.com /Blog/Year/2017/Month/1) (2)
- December 2016 (https://hsc.com /Blog/Year /2016/Month/12) (2)
- November 2016 (https://hsc.com /Blog/Year /2016/Month/11) (2)
- October 2016 (https://hsc.com /Blog/Year /2016/Month/10) (1)
- September 2016 (https://hsc.com /Blog/Year /2016/Month/9) (2)
- August 2016 (https://hsc.com /Blog/Year/2016/Month/8) (2)
- July 2016 (https://hsc.com /Blog/Year/2016/Month/7) (1)
- June 2016 (https://hsc.com /Blog/Year/2016/Month/6) (1)

Best Practices For Memory Optimization on Android (https://hsc.com/Blog/Best-Practices-For-Memory-Optimization-on-Android-1)

June 26, 2014

- Embedded Android Technology Trends, Heavy Reading (https://hsc.com/Blog/Tag/Heavy-Reading)
- 4 Comments (https://hsc.com/Blog/Best-Practices-For-Memory-Optimization-on-Android-1#comments)



(/Portals/0/images/Blog/uploads/2014/06/Android-Memory-Optimization.png)

For those tracking the evolution of Android (<http://blog.hsc.com/android/technology-trends/embedded-systems-history/>), it is evident that the future of the Android based ecosystem goes far beyond just phones and tablets. The OS is already making its way into a host of other smart devices (<http://blog.hsc.com /android/android-devices-at-home/embedded-android-devices-quick-look-todays-smart-technology-powered-android-os/>), like Google Glass (<http://blog.hsc.com/android/technology-trends/google-glass-development/>) for example, in a movement toward what's being called "the internet of things" (<http://blog.hsc.com/android/technology-trends/internet-of-things/>) or IoT. Developing a new OEM product based on Android as an embedded OS makes a lot of sense compared to say, only using Linux as we have covered before (<http://blog.hsc.com/android/technology-trends/android-vs-linux/>). However, getting Android to actually work effectively on diverse platforms is quite challenging. While phones and tablets are getting very powerful (with quad core processors and 2+GB RAM having become the de facto standard) this is certainly not the case with many other IoT devices where due to cost margins, the need of the day is still lower powered CPUs and lesser RAM (as RAM is an expensive part of any device BOM). While there are many mechanisms to reduce Android footprint and reduce memory overhead (such as headless Android mode, low memory Android configurations, etc.) ensuring that the application code also effectively uses available memory is important. This article covers best practices for memory usage.

Android & Ram

There are two kinds of memories when it comes to Android: Clean RAM and Dirty RAM.

Clean RAM

Unlike PCs, Android does not offer swap space for memory, however it does use paging and memory-mapping. Any files or resources which are present on the disk, such as code, are kept in mmap'ed pages. Android knows these pages can be recovered from the disk, so they can be paged out if the system needs memory somewhere else.

Dirty RAM

Dirty RAM is the memory that cannot be paged out. It can be expensive, especially when running in a background process. Most of the memory in a running application is dirty memory and this is the one you should watch out for.

In order to optimize the memory usage, Android tries to share some framework resources or common classes in memory across processes. So whenever a device boots up, a process called zygote loads the common framework code. Every new application process is then forked from the zygote process so it is able to access all the shared RAM pages loaded by it. While investigating an application's RAM usage, it is

- May 2016 (https://hsc.com/Blog/Year/2016/Month/5) (2)
- March 2016 (https://hsc.com/Blog/Year/2016/Month/3) (1)
- February 2016 (https://hsc.com/Blog/Year/2016/Month/2) (2)
- December 2015 (https://hsc.com/Blog/Year/2015/Month/12) (1)
- September 2015 (https://hsc.com/Blog/Year/2015/Month/9) (1)
- July 2015 (https://hsc.com/Blog/Year/2015/Month/7) (1)
- May 2015 (https://hsc.com/Blog/Year/2015/Month/5) (1)
- April 2015 (https://hsc.com/Blog/Year/2015/Month/4) (1)
- March 2015 (https://hsc.com/Blog/Year/2015/Month/3) (1)
- February 2015 (https://hsc.com/Blog/Year/2015/Month/2) (1)
- January 2015 (https://hsc.com/Blog/Year/2015/Month/1) (3)
- December 2014 (https://hsc.com/Blog/Year/2014/Month/12) (4)
- November 2014 (https://hsc.com/Blog/Year/2014/Month/11) (2)
- October 2014 (https://hsc.com/Blog/Year/2014/Month/10) (3)
- September 2014 (https://hsc.com/Blog/Year/2014/Month/9) (5)
- August 2014 (https://hsc.com/Blog/Year/2014/Month/8) (4)
- July 2014 (https://hsc.com/Blog/Year/2014/Month/7) (2)
- June 2014 (https://hsc.com/Blog/Year/2014/Month/6) (2)
- May 2014 (https://hsc.com/Blog/Year/2014/Month/5) (6)
- April 2014 (https://hsc.com/Blog/Year/2014/Month/4) (4)
- March 2014 (https://hsc.com/Blog/Year/2014/Month/3) (7)
- February 2014 (https://hsc.com/Blog/Year/2014/Month/2) (7)
- January 2014 (https://hsc.com/Blog/Year/2014/Month/1) (4)
- December 2013 (https://hsc.com/Blog/Year/2013/Month/12) (6)
- November 2013 (https://hsc.com/Blog/Year/2013/Month/11) (6)
- October 2013 (https://hsc.com/Blog/Year/2013/Month/10) (6)
- July 2013 (https://hsc.com/Blog/Year/2013/Month/7) (1)
- May 2013 (https://hsc.com/Blog/Year/2013/Month/5) (1)
- March 2013 (https://hsc.com/Blog/Year/2013/Month/3) (3)
- August 2012 (https://hsc.com

important to keep shared memory usage in mind since we should only be looking at the private dirty memory that is being used by our application. This is reflected by USS (unique set size) and PSS (proportional set size) in 'meminfo.'

Another important thing to keep in mind when investigating opportunities for memory optimization is that Android divides the application processes based on running vs cached processes. A running process is the foremost application running on the device or an application with a service running actively in the background. All other launched applications will go into the list of cached processes to allow for easier and faster switching between applications. For example, if an application is launched and then the user presses the 'Home' button, then that application's process will be added in the list of cached processes, that is if it does not have a running service. Be aware that the system will kill one or more cached processes if it needs more memory for any running process. Cached processes can be killed in LRU (least recently used) order. However there are some other options also, like killing whichever cached process will give the maximum memory gain for the system.

Every application on Android has a maximum heap size limit (varies for each device). You can check the `getMemoryClass()` API of `ActivityManager` service, it will tell you the maximum heap size available for any application on a device. Most devices running Android 2.3 or later will return this size as 24MB or higher. For example, the size on a Galaxy S3 is 64MB, whereas on a Nexus5 device, it is 192MB. Android will always start an application process with an average heap size and will then grow it up to the maximum limit on that device for an app. If an application reaches the maximum heap size and needs to allocate more memory, the system will throw an `OutOfMemoryError`.

Memory Optimization, Best Practices For Android

So what can you do to keep your system from running out of memory? Read on for some general guidelines for improving the memory usage and overall performance of Android.

Avoid Creating Unnecessary Objects

The basic rule of thumb here is that garbage collection is not free. The more objects an application allocates, the more frequently garbage collector will be forced to run – which eats up resources, needs to boost user experience and responsiveness. Temporary objects can also hurt. A large number of small allocations can also cause heap fragmentation.

For Example:

```
List<Chunk>mTempChunks = new ArrayList<Chunk>();

for(int i=0; i< 10000; i++) {

    mTempChunks.add(new Chunk(i));

}

for(int i= 0; i<mTempChunks.size(); i++) {

    Chunk c = mTempChunks.get(i);

    Log.d(TAG, "Chunk data: " + c.getValue());

}
```

In the second loop of the code snippet above, we are creating a new chunk object for each iteration of the loop. So it will essentially create 10,000 objects of type 'Chunk' and occupy a lot of memory. Imagine, if we have just missed the routine GC cycle before creating these objects, then these objects would lay around until next GC.

The same code can be written like the version below, instead:

```
Chunk c;

int length = mTempChunks.size();

for(int i= 0; i< length; i++) {

    c = mTempChunks.get(i);

    Log.d(TAG, "Chunk data: " + c.getValue());

}
```

Just one object!

contact us (https://hsc.com/Contact/Contact-Our-Team&enquiry=Best-Practices-For-Memory-Optimization-on-Android->Hughes-Systique-Corp.->Blog)

/Blog/Year/2012/Month/8) (1)
July 2012 (https://hsc.com
/Blog/Year/2012/Month/7) (1)
March 2012 (https://hsc.com
/Blog/Year/2012/Month/3) (1)
October 2010
(https://hsc.com/Blog/Year
/2010/Month/10) (1)
July 2010 (https://hsc.com
/Blog/Year/2010/Month/7) (1)
December 2009
(https://hsc.com/Blog/Year
/2009/Month/12) (1)
November 2009
(https://hsc.com/Blog/Year
/2009/Month/11) (1)
August 2009 (https://hsc.com
/Blog/Year/2009/Month/8) (3)
January 2009
(https://hsc.com/Blog/Year
/2009/Month/1) (1)
December 2008
(https://hsc.com/Blog/Year
/2008/Month/12) (1)
April 2008 (https://hsc.com
/Blog/Year/2008/Month/4) (1)
January 2007 (https://hsc.com
/Blog/Year/2007/Month/1) (1)
August 2006 (https://hsc.com
/Blog/Year/2006/Month/8) (1)

Recent Blog Posts
Future Innovations on POS and
Mobile Payments
(https://hsc.com
/Blog/Future-Innovations-
on-POS-and-Mobile-
Payments)
5/18/2017

How to Use Machine Learning
To Understand Your
Customers Better
(https://hsc.com/Blog/How-
to-Use-Machine-Learning-
To-Understand-
Your-Customers-Better)
5/4/2017

Role of Machine Learning in
Accelerating Digital
Transformation
(https://hsc.com/Blog/Role-
of-Machine-Learning-
in-Accelerating-Digital-
Transformation)
4/26/2017

How NFV (Network Function
Virtualization) Will Accelerate
Mobile Cloud Adoption
(https://hsc.com/Blog/How-
NFV-Network-Function-
Virtualization-Will-Accelerate-
Mobile-Cloud-Adoption)
4/11/2017

Another Example:

Try to reuse the same object when passing through functions, like below:

```
private StringBuilder getNextToken(StringBuilder sb) {  
  
    sb.append(readString(filestream));  
  
    sb.append(readString(filestream));  
  
    return sb;  
}
```

Notice how the string is appended to same StringBuilder object directly without creating any short term temporary objects for String and StringBuilder.

Be Aware Of Memory Overhead Of The Language
It helps to know the cost and overhead of language constructs we are using.

For example:

An Object with just one int variable takes 16 bytes at minimum in Android:

```
Class Integer {  
private int value;  
}
```

overhead of Object	+	overhead of dlmalloc	+	data
8 bytes		4-8 bytes		n bytes

The result must be 8-byte aligned

contact us (https://hsc.com
/Contact/Contact-
Our-Team?enquiry=Best
Practices For Memory
Optimization on Android >
Hughes Systique Corp. > Blog)

HashMap:

```
classHashMap$HashMapEntry<K, V> {  
  
    final K key;  
  
    V value;  
  
    finalint hash;  
  
    HashMapEntry<K, V> next;  
}
```

Total = 4(Object) + 8(dlmalloc) + 4 * 4(members) = 28 bytes

Aligned total = 32 bytes

contact us (https://hsc.com
/Contact/Contact-
Our-Team?enquiry=Best
Practices For Memory
Optimization on Android >
Hughes Systique Corp. > Blog)

So, every entry in a HashMap would occupy 32 bytes.

Objects Vs. Primitive Types

As mentioned above, since an "Integer" boxed object occupies 4 times as much memory as primitive "int," we should always try to use primitive types where we can. Similarly, a Boolean boxed object occupies much more memory than primitive boolean type. In the example below, the API call returns a primitive "int" value but we have assigned this value to an Integer object. This assignment will perform an autoboxing operation from int to Integer object. For a single call, this may not matter much. However, if we are using it frequently, for example in an inner loop, it might occupy a lot of memory unnecessarily.

```
Integer width = view.getWidth();  
  
|  
  
Autoboxing
```

Integer (16 bytes)	vs	int(4 bytes)
Boolean(16 bytes)	vs	boolean(4 bytes)
vs	bit-field(1 bit)	//even better!

Enums Vs. Ints

Plain and simple – always avoid using enums on Android. Instead, use “static final” variables for constants. Enums usually require more than twice as much memory as static constants.

```
public static enum Things {  
      
    THING_1,  
    THING_2,  
};  
  
dex-file size  
  
+1,112 bytes
```

VS.

```
public static int THING_1 = 1;  
  
public static intTHING_2 = 2;  
  
+128 bytes
```

Avoid Unnecessary Classes/Inner Classes

Every class in Java, including anonymous inner classes which create an object and writes accessor methods internally, uses about 500 bytes of code.

```
button.setOnClickListener(new Runnable() {  
      
    public void run() {  
        // do stuff  
    }  
});
```

So, these kinds of listeners should be unregistered as soon as they are not needed.

Hidden Cost Of Abstractions

In general, writing code with multiple layers of abstraction is considered good programming practice for object-oriented languages. However, the more code that is written, the more execution time and memory it is going to take. So, try not to overdo the layers. Only use abstractions where they provide a significant benefit. For example, in cases where writing a library to be used by other applications, it makes sense to use abstractions to expose only certain areas of functionality.


Beware of Services


Services are useful for running operations in the background, but they are very expensive. You should never keep a service running unless absolutely required. The best way to automatically manage service lifecycle is to use an IntentService, which will finish itself after its work is done. For other services, it's the application developer's responsibility to make sure that stopService or stopSelf is being called after work is done.

Release Memory When User Interface Becomes Hidden

When the user navigates to a different activity, release the resources associated with that activity in onPause and onStop callbacks. These resources are generally a network or database connection, a broadcast receiver, etc.

If the user navigates to a different application and all the UI components of the app are hidden, the app receives onTrimMemory() callback in all activities if Android system needs to kill any cached process to reclaim some memory for a running process. Listen for the TRIM_MEMORY_UI_HIDDEN level and release the UI resources here. For example, textviews, imageviews, etc.

 contact us (<https://hsc.com/Contact/Contact-Our-Team?enquiry=BestPracticesForMemoryOptimizationonAndroid>)

 Hughes Systique Corp. > Blog

```
public void onTrimMemory(int level) {  
  
    if (level >= TRIM_MEMORY_COMPLETE) {  
  
        // ...  
  
    } else if (level >= TRIM_MEMORY_MODERATE) {  
  
        // ...  
  
    } else if (level >= TRIM_MEMORY_BACKGROUND) {  
  
        // ...  
  
    } else if (level >= TRIM_MEMORY_UI_HIDDEN) {  
  
        // ...  
  
        Cached  
  
        -----  
  
        Running  
  
    } else if (level >= TRIM_MEMORY_RUNNING_CRITICAL) {  
  
        // ...  
  
    } else if (level >= TRIM_MEMORY_RUNNING_LOW) {  
  
        // ...  
  
    } else if (level >= TRIM_MEMORY_RUNNING_MODERATE) {  
  
        // ...  
  
    }  
  
}
```

Optimize Bitmaps Memory Usage

Bitmaps are often the largest RAM user in an application. A bitmap loaded in memory takes much more RAM than the size of the image we see on filesystem because:

bitmap size = width * height * depth (usually 4 bytes)

Keeping it in mind, bitmap should be loaded in RAM only at the size and resolution of the current device's screen. So, we should scale it down if the original bitmap is at higher resolution. On Android 2.3.3 and lower, the backing pixel data for a bitmap was stored in native memory, and there was a need to write finalizers in Java code to free this native memory allocation. Therefore, it used to take more than one GC cycle to free bitmap memory. Hence it was recommended to use recycle on bitmaps after using them to free the memory as soon as possible.

As of Android 3.0 (API level 11) however, the pixel data is stored on the Dalvik heap along with the associated bitmap. So there is no need to call recycle(). But it is still useful to optimize the large amount of memory used by bitmap, and we should try to reuse bitmaps whenever possible. API level 11 introduces the BitmapFactory.Options.inBitmap field. If this option is set, decode methods of BitmapFactory that take the Options object will attempt to reuse an existing bitmap when loading content.

```
final BitmapFactory.Options options = new BitmapFactory.Options();  
  
// inBitmap only works with mutable bitmaps, so force the decoder to return  
// mutable bitmaps.  
options.inMutable = true;  
  
if (reusableBitmaps.size() > 0) {  
  
    // Try to find a bitmap to use for inBitmap.  
    Bitmap inBitmap = reusableBitmaps.remove();  
  
    if (inBitmap != null) {  
  
        // If a suitable bitmap has been found, set it as the value of  
        // inBitmap.  
        options.inBitmap = inBitmap;  
    }  
}  
  
...  
  
BitmapFactory.decodeFile(filename, options);  
  
}
```

 contact us (<https://hsc.com/Contact/Contact-Our-Team?enquiry=BestPracticesForMemoryOptimizationonAndroid>)

 Hughes Systique Corp. > Blog

Use Optimized Datacontainers

Android has provided a few optimized data containers in SDK and support libraries, such as SparseArray, SparseBooleanArray, and ArrayMap. These containers can be a replacement for HashMap where the keys are of primitive type like int, Boolean, and so on. Since HashMap needs an Integer object for storing ints, it occupies a lot more memory than is actually needed, especially if we have large number of entries in our map. ArrayMap also consumes less memory, however it is slower in access than HashMap, so it should be only used when working with smaller number of elements, like < 100.

Here are some examples of optimized data containers which we can use as replacement for their HashMap equivalents:

HashMap	Array Class
<Integer, Object>	SparseArray
<Integer, Boolean>	SparseBooleanArray
<Integer, Integer>	SparseIntArray
<Integer, Long>	SparseLongArray
<Long, Object>	LongSparseArray

Use raw arrays, like int[], in performance-critical sections of the code or where we are working with hundreds of thousands of elements at a time, if possible.

Proguard And Zipalign


The ProGuard tool shrinks, optimizes, and obfuscates the code by removing unused code and renaming classes, fields and methods with semantically obscure names. ProGuard can make the code more compact, requiring fewer RAM pages to be mapped. But you should be aware of how ProGuard works before using it in the application. For example, by default, ProGuard will strip out native JNI functions, dynamically loaded classes or methods, and code which is part of some library internally referenced by another library in the project. So, it is important to configure ProGuard config file to add rules for keeping all the required classes and methods in the project.

While preparing a release build of the application, it is always important to run the ZipAlign tool on the APK to have it re-aligned. This is necessary in order to maximize our static code and resources to be mmapped by Android. Eclipse probably already does that automatically, however we should take care if we are building the APKs on our own using Ant.

To recap, some general performance tips for memory optimization:

- > Try to avoid static variables or objects as much as possible if they are not final constants. Static variables pose the threat of having references in other classes, which we might forget about and thus cause a memory leak.
- > Prefer static methods over virtual methods where any of the member fields of object are not accessed. The static invocations are faster because they save dalvik look up of the method.
- > Avoid internal getters and setters. Direct field access is much faster in Android than virtual method lookup. If an app is not exposing APIs as libraries, it should avoid using accessors.
- > A common mistake is to save "Context" objects everywhere in the application. If the developer forgets to free even one reference of a context, it increases the chance of a whole activity leak.
- > Match the calls of registration and un-registration of the listeners, receivers in corresponding pairs in activity lifecycle. If a broadcast receiver was registered in onStart method, then it should be unregistered in onStop method only. The same goes for onCreate-onDestroy and onResume-onPause.
- > In Eclipse or Android Studio, ADT comes with lint analysis tool. It provides tips and tricks for optimizing the application code at the time of compilation. Always pay attention to these tips and warnings and try to incorporate them if possible.
- > For passing information between components of same application, avoid using IPC mechanisms like Intents and ContentProviders if possible. Work with handlers, listener interfaces instead.



 contact us (<https://hsc.com/Contact/Contact-Our-Team?enquiry=BestPracticesForMemoryOptimizationonAndroid> > Hughes Systique Corp. > Blog)

Tools For Measuring Android Memory Usage

To analyze an application's memory usage on Android, there are several memory profiling tools available. The Android SDK provides two main ways of profiling the memory usage of an app: the Allocation Tracker tab in DDMS, and heap dumps. The Allocation Tracker is useful when we want to get a sense of what kinds of allocations are happening over a given time period, but it doesn't provide any information about the overall state of the application's heap.

To collect a heap dump:

- > Run the app
- > Select the app in DDMS
- > Press "Dump HPROF File" button
- > Save file
- > Run hprof-convhprof-conv heap-original.hprof heap-converted.hprof

Note – DDMS version integrated into Eclipse does hprof conversion automatically, so this step is not required.

To analyze a heap dump, we can use standard tools like Eclipse Memory Analyzer(MAT) or jhat.

When analyzing the heap dump, look for memory leaks caused by:

- > Long-lived references to an Activity, Context, View, Drawable, and other objects that may hold a reference to the container Activity or Context.
- > Non-static inner classes (such as a Runnable, which can hold the Activity instance).
- > Large objects accumulating over time collect multiple heap dumps at different intervals and compare them. MAT Tool's Leak Suspects Report is particularly useful in this area.

Meminfo And Procstats

To observe how an application is divided between different types of RAM, we can use the following adb command:


adb shell dumpsys meminfo <package_name>

This command lists all the current allocations of an application, measured in kilobytes. The important details are the memory used by USS (Private Dirty + Private Clean) and PSS total. It will also show the number of activities currently running, number of view objects allocated and binder objects shared between processes.

Kitkat (4.4) introduced a new service called Procstats to help better understand the memory usage on a device. There is a UI screen found under the Developer Options menu so that you can look at the memory usage by all applications. To run Procstats from command line:

adb shell dumpsys procstats <package_name>

(Do you find this article interesting? You may want to check out our [Embedded Android \(/Services/Product-Engineering-Services/Embedded-Android\)](#) pages to read more about what we do in this space.)

 contact us (<https://hsc.com/Contact/Contact-Our-Team?enquiry=BestPracticesForMemoryOptimizationonAndroid> > Hughes Systique Corp. > Blog)

Hi all, you are correct. That specific example is incorrect. It is infact, not creating a new object each time in the loop. Thanks for pointing it out.

Reply

Dec 15, 2016
Abu

Your articles will help me in memory utilization each application. Great article !

Reply

Apr 12, 2015
silinik

"In the second loop of the code snippet above, we are creating a new chunk object for each iteration of the loop. So it will essentially create 10,000 objects of type 'Chunk' and occupy a lot of memory. "

Are you sure that
Chunk c = mTempChunks.get(i);
creates a new object every loop iteration? I think that we just get a reference to the previously created object

Reply

Apr 08, 2015
Ian Ni-Lewis

Great article, lots of good advice here! A couple of quibbles:

1. In your first example, unless I'm greatly mistaken, you are not in fact creating a new object each time through the loop. You are creating a local reference to an existing object. The local reference consumes a few bytes on the stack for a very short period of time. Technically it's a root, so if gc kicks off in the middle of your loop it will add a negligible bit of overhead to the mark phase, but otherwise it's harmless.
2. Changes to the allocator make your object size calculations invalid on Android 5.0+. However, your larger point still stands, because the new runtime allocates in blocks whose size is a multiple of 16. That may change in the future, though...

Reply

Add Comment

Join the Discussion...

 contact us (<https://hsc.com/Contact/Contact-Our-Team?enquiry=BestPracticesForMemoryOptimizationonAndroid> > Hughes Systique Corp. > Blog)

