

AnDevCon Boston 2014

Putting Your App on a Memory Diet



The Problem You Are Thinking Of

```
E/AndroidRuntime(2065): FATAL EXCEPTION: main
E/AndroidRuntime(2065): java.lang.OutOfMemoryError
E/AndroidRuntime(2065):     at android.graphics.Bitmap.nativeCreate(Native Method)
E/AndroidRuntime(2065):     at android.graphics.Bitmap.createBitmap(Bitmap.java:605)
E/AndroidRuntime(2065):     at android.graphics.Bitmap.createBitmap(Bitmap.java:551)
E/AndroidRuntime(2065):     at android.graphics.Bitmap.createScaledBitmap(Bitmap.java:437)
E/AndroidRuntime(2065):     at android.graphics.BitmapFactory.finishDecode(BitmapFactory.java:618)
E/AndroidRuntime(2065):     at android.graphics.BitmapFactory.decodeStream(BitmapFactory.java:593)
E/AndroidRuntime(2065):     at
android.graphics.BitmapFactory.decodeResourceStream(BitmapFactory.java:445)
E/AndroidRuntime(2065):     at
android.graphics.BitmapFactory.decodeResource(BitmapFactory.java:468)
...
```



The Problem You Are Not Thinking Of

- System RAM
 - The bigger your app, the more likely it is to get kicked out of RAM once it moves to the background
 - Rationale: the bigger they are, the bigger the benefit for getting rid of 'em
 - Converse: the smaller your footprint, the longer you'll live, and the better your multitasking will work



Root Cause #1: Limited Heap Size

- Originally 16MB
 - (no, that's not a typo)
- Varies by Device
 - OS level
 - Screen resolution
- Not Directly Controllable by App
 - ...though we will discuss some indirect controls



The Green Mantra

- Reduce
 - Consume less of the resource in the first place
 - Example: bulk consumer packaging
- Reuse
 - Apply the resource to another problem when done with the first
 - Example: shipping eBay sale using Amazon box
- Recycle
 - Return resource to the manufacturing stream
 - Example: consumer paper, plastic recycling programs



The #A4C739 Mantra

- Reduce
- **Reorder**
 - When you allocate has impacts on what you can allocate
- Reuse
- Recycle



The #A4C739 Mantra

- **Reckon**
 - The first step on the road to recovery is to admit that you have a problem... then measure it
- Reduce
- Reorder
- Reuse
- Recycle



The #A4C739 Mantra

- Reckon
- Reduce
- Reorder
- Reuse
- Recycle
- **Cheat**



Android Memory: Reckon



Reckon: System RAM

- Why Do We Care?
 - Lower system RAM usage = somewhat less likely for Android to terminate our process
 - Improves multitasking for users



Reckon: System RAM

- What Can We Do About It?
 - GC!
 - Dalvik allocates space for our heap from system RAM on a page-by-page basis
 - Dalvik will release those pages if we allow lots of data to be garbage-collected
 - Less code (DEX, .so)
 - Stop cheatin'



Reckon: System RAM

- Process Stats
 - Available in Android 4.4
 - Gives you average, peak PSS over period
 - Proportional Set Size = how much system RAM to blame your process for
 - Also shows how much time your process ran, how much time services in that process ran



Reckon: System RAM

- procstats
 - Same backing data as Process Stats, but with command-line goodness
- meminfo
 - Available on older versions of Android
 - PSS per process, grouped by importance



Reckon: Heap Usage

- Runtime Information
 - `ActivityManager#getMemoryClass()`
 - Returns number of MB for max heap size
 - Exception: large heap (but that's cheatin')
 - `android.os.Debug` methods
 - `getNativeHeapSize()`
 - `getNativeHeapAllocatedSize()`
 - `getNativeHeapFreeSize()`
 - Not as useful as you might think...



Reckon: Heap Usage

- Memory Analysis Tool (MAT)
 - Used to examine heap dumps
 - From DDMS
 - From `Debug#dumpHprofData()`
 - Available as standalone GUI or as Eclipse plugin
 - Uses
 - Understanding what's using up your heap
 - Finding memory leaks



Android Memory: Reduce



Reduce: Scale All Your Caches

- Use `getMemoryClass()`
 - Plus algorithm for capping your caches to certain portion of possible heap
 - On top of using weak or soft references
- Beware Multiple Caches
 - Library A has a cache, Library B has a cache, etc.
 - Sum of all caches must be reasonable, beyond any individual cache



Reduce: Load What You Need, Not What You Have

- Avoid massive scrolling lists
 - User can't find anything anyway
 - Steer user towards searching large data sets
- Load bitmaps as needed
 - Example: avatars for a list
 - And pay attention to row recycling and such
- Easy on the POJOs
 - ...or at least close your cursors after copying



Reduce: Ponder Bitmap Sizes

- Load the Size You Need
- Example: ListView rows
 - Load thumbnails, using inSampleSize on BitmapFactory.Options
 - Only load full-size image for the ones the user clicks upon and brings up some sort of details fragment/activity/whatever



Reduce: Ponder Bitmap Pixels

- Default: ARGB_8888
 - 4 bytes per pixel
- Alternative: RGB_565
 - 2 bytes per pixel = half the memory for same resolution
 - No transparency
- Example: ListView rows
 - Thumbnails perhaps can get away with less color depth



Reduce: Simpler UI/Progressive Enhancement

- Think Web apps
 - Opt into different page rendering depending on browser capabilities
 - Usually same broad-brush features for the user, but missing “sizzle” or optional features
- Treat low-memory devices differently
 - Steer the user away from things that may blow out your available heap
 - Problem: how best to package this
 - “Hey! Get a real phone!” unlikely to prove popular



Android Memory: Reorder



Root Cause #2: Non-Compacting GC

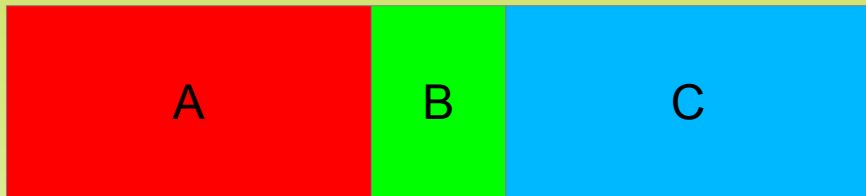
- Only combines adjacent free blocks into larger free block
 - Does not move objects in memory to try to coalesce free blocks
- Net: heap fragmentation
 - OutOfMemoryError = no block big enough to satisfy request
 - Think defragmenting hard drives



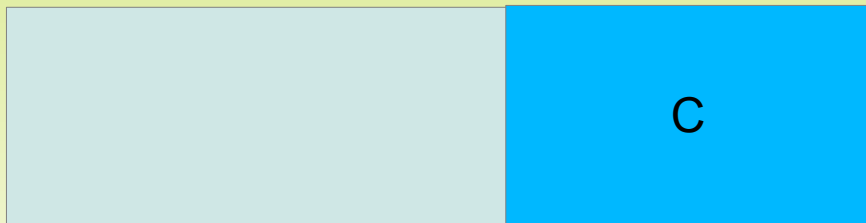
Root Cause #2: Non-Compacting GC



In the beginning, there was heap,
and it was good...



The story of “The Three Little Blocks
and the Big Bad OutOfMemoryError”



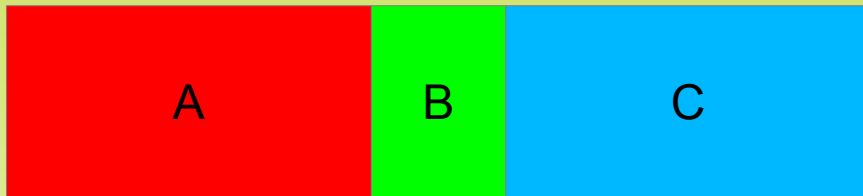
Trash compactor



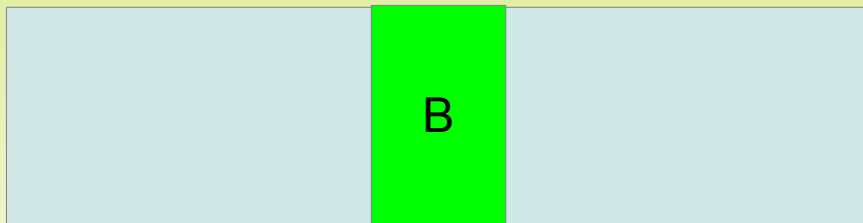
Root Cause #2: Non-Compacting GC



In the beginning, there was heap,
and it was good...



The story of “The Three Little Blocks
and the Big Bad OutOfMemoryError”



Where, o where has my heap space
gone? O where, o where can it be?



Reorder: Allocate Long-Lived Stuff Earlier

- CIY (Compact It Yourself)
 - Allocate bitmap or other object pools early
 - Objective: minimize long-lived objects fragmenting the heap
 - Only really useful for memory buffers that can be reused, rather than recycled



Reorder: MAT

- Know Your Heap
 - What are you allocating and holding onto long-term?
 - IOW, what are your key GC roots?
 - Static data members
 - Threads
 - Long-running services
 - What can you do to minimize what they reference?



Reorder: Let Your Process Go Poof

- Nuke Your Entire Heap From Orbit
 - It's the only way to be sure that you're going to get a nice clean heap again
- Avoid the “Must Keep Process Running” Syndrome
 - Or use a second process for focused heap for long-running stuff (but that's cheatin')



Android Memory: Reuse



Reuse: Object Pools

- Pools
 - Collection of some common resource (objects, threads, etc.)
 - Access patterns to acquire and release
 - Use the resource in between
 - Pre-allocated minimum pool contents
 - Cap on maximum pool size
 - Grow as need to limit, release resources to shrink
 - Acquire blocks if needed for other thread to release



Reuse: Object Pools

- Rationale
 - Avoid heap fragmentation!
 - Slight heap compacting effect via pre-allocation
 - Reduce GC processing time
 - Minimize constructor CPU time
- Counter-Arguments
 - Shades of `malloc()` and `free()`



Reuse: Object Pools

- Framework-Enforced
 - SensorEvent, etc.
- Framework-Encouraged
 - TypedArray, etc.
- Custom
 - Stormpot
 - Commons-Pool
 - A zillion blog posts



Reuse: inBitmap

- BitmapFactory.Options field
- Specifies Bitmap to reuse
 - API Level 19+: must be same size or larger than bitmap to be loaded
 - API Level 18 and lower: must match exactly
- Great for thumbnails, other scenarios with constant bitmap sizes
 - Usually integrated into bitmap libraries



Android Memory: Recycle



Recycle: MAT and Leaks

- Look for Leak Candidates
 - Instances of your classes that should not be around
 - Bitmaps and byte arrays
- Trace Your GC Roots
 - “GC Roots” = “what the @#\$%&! is holding onto this &%&\$ thing?!?”



Recycle: Choose the Right Context

- Beware of a Custom Application Class
 - Singleton, so anything it holds onto cannot be GC'd
- Use Application in the Right Places
 - If you are holding onto things in static data members that might be associated with a Context, use Application, as it is “pre-leaked”
 - Failure to do so: leak activities, etc.



Recycle: onTrimMemory()

- Called On Your Components
 - Activity, Service, ContentProvider, Application, etc.
- Objective: Release Some Heap Space
 - In hopes that Dalvik can release some of that heap back to the OS



Recycle: onTrimMemory()

- You Are Safe (But Please Be Kind to Others)
 - TRIM_MEMORY_RUNNING_MODERATE
 - TRIM_MEMORY_RUNNING_LOW
 - TRIM_MEMORY_RUNNING_CRITICAL
- You Are Invisible
 - TRIM_MEMORY_UI_HIDDEN



Recycle: onTrimMemory()

- Your Time is Running Out
 - TRIM_MEMORY_BACKGROUND
 - TRIM_MEMORY_MODERATE
 - TRIM_MEMORY_COMPLETE
 - Also available as onLowMemory() for sub-API Level 14 devices



Recycle: Watch Your Threads

- Threads = GC Roots
 - Anything reachable by a thread cannot be GC'd
- Tips
 - Leaking threads = leaking heap
 - Ensure threads in pools null out data members
 - Beware the everlasting service!
 - Its threads, and anything else, cannot be GC'd
 - Also screws up multitasking, etc.



Android Memory: Cheat



Cheat: Request Large Heap

- `android:largeHeap` in `<application>`
- **Probably** gives you a larger heap on API Level 11+
 - Depends a bit on device capabilities
- Use `getLargeMemoryClass()` to determine how large your heap is



Cheat: Multiple Processes

- Reason #1: More Heap
 - Use second process for specific memory-intensive operations
 - Workaround for pre-Honeycomb devices
- Reason #2: Focused Heap
 - Use second process for long-running background services
 - May be able to accomplish same basic end with `onTrimMemory()`



Cheat: NDK

- Native allocations do not count against Dalvik heap
- Use native code for memory-intensive operations
 - Particularly where you could gain some performance from native code
 - Example: image processing



The Costs of Cheating

- Larger system RAM consumption
 - More likely to get blamed by OS, users
 - More likely to have background process terminated
- Multiple processes = IPC
 - CPU and battery consumption
 - Keep your protocol relatively coarse-grained



Summary

- Reckon: measure twice, cut if needed
- Reduce: what you don't use can't hurt you
- Reorder: avoid fragmentation (the heap kind)
- Reuse: object pools FTW!
- Recycle: this is why we used managed code
- Cheat: try not to, #kthxbye

