ANDROID PERFORMANCE

Best practices in performance for Android

TRAINING ROADMAP: OVERVIEW

- Profiling
- Services usage recommendations
- Memory usage recommendations
- Code design recommendations
- Views recommendations

This training covers major aspects of performance on Android. The goal of this training is to inform students about best performance practices on Android.

Pre-requisites:

- Readiness to learn new

PROFILING

- What is Profiling?
  - Analyzing behavior in runtime
  - CPU utilization
  - Memory and I/O analysis
  - Measurement of time and space
- Why do we need it?
  - Avoid guesswork
  - Quickly identify performance pitfalls

PROFILING: TOOLS

- System info (CPU/GPU/FPS)
- Heap view
- Memory alloc
- Method profiling
- Logcat
- File explorer

Android DDMS	Android studio
Examine threads	Allocation tracker
Network traffic	Memory monitor
View hierarchy	Android Memory Viewer
Emulate calls	GPU Monitor
	CPU Monitor

PROFILING: TOOLS

Android DDMS	Android studio
Heap view	Allocation tracker
Memory alloc	Memory monitor
Method profiling	

PROFILING: DDMS METHOD PROFILING

- Sample-based profiling
  - Interrupt VM at sampling frequency at collection call stacks
  - Overhead proportional to frequency
- Trace-based profiling
  - Trace entry and exit of every method
  - High overhead
- Start Record, Perform Action, Stop Record
- Timeline vs Traceview panel views

PROFILING: MEMORY TRACKING

Heap view	Allocation tracker
Total available heap space	List of allocations
Total space allocated	Allocation order
Total space free	Allocation size
% Used	Class/ Stacktrace
Total # Objects	Thread id
Object allocation per size	Allocated in

AGREEMENT

- You should consider RAM constraints throughout all phases of development, including during app design.

- You should apply the following techniques while designing and implementing your app to make it more memory efficient.

SERVICES: USE SERVICES SPARINGLY

- If your app needs a service to perform work in the background, do not keep it running unless it's actively performing a job, stop it when its work is done.
- When service is started, the system keeps the process for that service running - RAM used by the service can’t be used by anything else or paged out.
- **This reduces the number of cached processes that the system can keep in the LRU cache, making app switching less efficient.**

SERVICES: USE SERVICES SPARINGLY: INTENT SERVICE

- The best way to limit the lifespan of your service is to use an **IntentService**, which finishes itself as soon as it's done handling the intent that started it.
- Leaving a service running when it’s not needed is one of the worst memory-management mistakes an Android app can make.
- Don’t be greedy by keeping a service for your app running. Not only will it increase the risk of your app performing poorly due to RAM constraints, but users will discover such misbehaving apps and uninstall them.

RELEASE MEMORY: UI HIDDEN

- When the user navigates to a different app and your UI is no longer visible - **release any resources that are used by only your UI.**
- Releasing UI resources at this time can significantly increase the system's capacity for cached processes, which has a direct impact on the quality of the user experience.

RELEASE MEMORY: UI HIDDEN IMPLEMENTATION

- To be notified when the user exits your UI, implement the **onTrimMemory()** callback in your Activity classes.
- You should use this method to listen for the TRIM_MEMORY_UI_HIDDEN level, which indicates your UI is now hidden from view and you should free resources that only your UI uses.

RELEASE MEMORY: UI HIDDEN IMPLEMENTATION

- Your app receives the onTrimMemory() callback with TRIM_MEMORY_UI_HIDDEN only when all the UI components of your app process become hidden from the user.
- Implement onStop() to release activity resources such as a network connection or to unregister broadcast receivers, you usually should not release your UI resources until you receive **onTrimMemory(TRIM_MEMORY_UI_HIDDEN)**.

RELEASE MEMORY: UI HIDDEN IMPLEMENTATION

- TRIM_MEMORY_RUNNING_MODERATE - app is running and not considered killable, but the device is running low on memory and the system is actively killing processes in the LRU cache.
- TRIM_MEMORY_RUNNING_LOW - app is running and not considered killable, but the device is running much lower on memory so you should release unused resources to improve system performance (impacts your app's performance).
- TRIM_MEMORY_RUNNING_CRITICAL - app is still running, but the system has already killed most of the processes in the LRU cache, you should release all non-critical resources now. If the system cannot reclaim sufficient amounts of RAM, it will clear all of the LRU cache and begin killing processes that the system prefers to keep alive, such as those hosting a running service.

RELEASE MEMORY: UI HIDDEN IMPLEMENTATION

- TRIM_MEMORY_BACKGROUND - your process is near the beginning of the LRU list, app process is not at a high risk of being killed, the system may already be killing processes in the LRU cache. Should release resources that are easy to recover so your process will remain in the list and resume quickly when the user returns to your app.
- TRIM_MEMORY_MODERATE - your process is near the middle of the LRU list. If the system becomes further constrained for memory, there's a chance your process will be killed.
- TRIM_MEMORY_COMPLETE - your process is one of the first to be killed if the system does not recover memory now. You should release everything that's not critical to resuming your app state.

RELEASE MEMORY: MEMORY LIMITS

- Each device has a different amount of RAM available to the system and thus provides a different heap limit for each app.
- You can call **getMemoryClass()** to get an estimate of your app's available heap in megabytes.
- If your app tries to allocate more memory than is available here, it will receive an **OutOfMemoryError**.
- You can request a larger heap size by setting the **largeHeap** attribute to "true" in the manifest <application> tag. If you do so, you can call getLargeMemoryClass() to get an estimate of the large heap size.

RELEASE MEMORY: MEMORY LIMITS

- Never request a large heap simply because you've run out of memory and you need a quick fix.
- Using the extra memory will increasingly be to the detriment of the overall user experience because garbage collection will take longer and system performance may be slower when task switching or performing other common operations.

MEMORY: BITMAPS LOADING

- When you load a bitmap, keep it in RAM only at the resolution you need for the current device's screen, scaling it down if the original bitmap is a higher resolution.

MEMORY: OPTIMIZED DATA CONTAINERS

- Take advantage of optimized containers in the Android framework, such as SparseArray,SparseBooleanArray, and LongSparseArray.
- Advantages:
  - Avoid using extra entry object for each mapping.
  - Avoid autoboxing for keys and sometimes for values.
- Disadvantages:
  - Use binary search for lookups, instead of hashing. Slower for large number of entries , > 1000, compared to Hashmaps

MEMORY: MEMORY OVERHEAD

- Overhead of the language and libraries you are using, and keep this information in mind when you design your app, from start to finish.
- Enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.
- Every class in Java (including anonymous inner classes) uses about 500 bytes of code.
- Every class instance has 12-16 bytes of RAM overhead.
- Putting a single entry into a HashMap requires the allocation of an additional entry object that takes 32 bytes

MEMORY: ABSTRACTIONS MEMORY CONSUMPTIONS

- Developers use abstractions simply as a "good programming practice," because abstractions can improve code flexibility and maintenance.
- However, abstractions come at a significant cost:
  - Generally they require a fair amount more code that needs to be executed, requiring more time and more RAM for that code to be mapped into memory.
  - If your abstractions aren't supplying a significant benefit, you should avoid them.

MEMORY & PERFOMANCE: DI

- Using a dependency injection framework may be attractive because they can simplify the code you write and provide an adaptive environment that's useful for testing and other configuration changes.
- However, these frameworks tend to perform a lot of process initialization by scanning your code for annotations, which can require significant amounts of your code to be mapped into RAM even though you don't need it.

MEMORY & PERFOMANCE: EXTERNAL LIBRARIES

- External library code is often not written for mobile environments and can be inefficient when used for work on a mobile client.
- At the very least, when you decide to use an external library, you should assume you are taking on a significant porting and maintenance burden to optimize the library for mobile.

MEMORY & PERFOMANCE: PROGUARD

- The ProGuard tool shrinks, optimizes, and obfuscates your code by removing unused code and renaming classes, fields, and methods with semantically obscure names.
- Using ProGuard can make your code more compact, requiring fewer RAM pages to be mapped.

MEMORY & PERFOMANCE: BEST PRACTICES

- Object creation is never free. As you allocate more objects in your app, you will force a periodic garbage collection, creating little "hiccups" in the user experience.
- If you don't need to access an object's fields, make your method static.
  - Invocations will be about 15%-20% faster.
  - You can tell from the method signature that calling the method can't alter the object's state.

MEMORY & PERFOMANCE: BEST PRACTICES

- Consider the following declaration at the top of a class:
  - final static int intVal = 42;
  - final static String strVal = “I ❤️ Android & Luxoft”;
- In native languages like C++ it's common practice to use getters (i = getCount()) instead of accessing the field directly (i = mCount).
  - this is a bad idea on Android. Virtual method calls are expensive, more so than instance field lookups. Follow common oop practices, have getters and setters in the public interface, but within a class always access fields directly.

MEMORY & PERFOMANCE: BEST PRACTICES

- As a rule of thumb, floating-point is about 2x slower than integer on Android-powered devices.
- In speed terms, there's no difference between float and double on the more modern hardware. Space-wise, double is 2x larger.
- As with desktop machines, assuming space isn't an issue, you should prefer double to float.

MEMORY & PERFOMANCE: BEST PRACTICES

- RecyclerView (replaces ListView and GridView)
  - setHasFixedSize
  - Data updates (notifyDataSetChanged)
  - Layout updates (notifyItemChanged / Inserted / Removed)
- <include> tag (easy code reuse of common layouts)
- <merge> tag (avoid extra ViewGroups on inflation)
- TextView Compound Drawables (add images to TextView)