Android Developers

Manage Your App's Memory

Random-access memory (RAM) is a valuable resource in any software development environment, but it's even more valuable on a mobile operating system where physical memory is often constrained. Although both the Android Runtime (ART) and Dalvik virtual machine perform routine garbage collection, this does not mean you can ignore when and where your app allocates and releases memory. You still need to avoid introducing memory leaks, usually caused by holding onto object references in static member variables, and release any `Reference` (https://developer.android.com/reference/java/lang/ref/Reference.html) objects at the appropriate time as defined by lifecycle callbacks.

This page explains how you can proactively reduce memory usage within your app. For more information about general practices to clean up your resources when programming in Java, refer to other books or online documentation about managing resource references. If you're looking for information about how to analyze memory in a running app, read Tools for analyzing RAM usage (#AnalyzeRam). For more detailed information about how the Android Runtime and Dalvik virtual machine manage memory, see the Overview of Android Memory Management (https://developer.android.com/topic/performance/memory-overview.html).

In this document

- Monitor Available Memory and Memory Usage
  - Tools for analyzing RAM usage
  - Release memory in response to events
  - Check how much memory you should use
- Use More Efficient Code Constructs
  - Use services sparingly
  - Use optimized data containers
  - Be careful with code abstractions
  - Use nano protobufs for serialized data
  - Avoid memory churn
- Remove Memory-Intensive Resources and Libraries
  - Reduce overall APK size
  - Use Dagger 2 for dependency injection
  - Be careful about using external libraries

See Also

- Overview of Android Memory Management
- Investigating Your RAM Usage
- Reduce APK Size

Monitor Available Memory and Memory Usage

The Android framework and Android Studio can help you analyze and adjust your app's memory usage. The Android framework exposes several APIs that allow your app to reduce its memory usage dynamically during runtime. Android Studio contains several tools that allow you to investigate how your app uses memory.

Tools for analyzing RAM usage

Before you can fix the memory usage problems in your app, you first need to find them. Android Studio and the Android SDK include several tools for analyzing memory usage in your app:

1. The Memory Monitor in Android Studio shows you how your app allocates memory over the course of a single session. The tool shows a graph of available and allocated Java memory over time, including garbage collection events. You can also initiate garbage collection events and take a snapshot of the Java heap while your app runs. The output from the Memory Monitor tool can help you identify points when your app experiences excessive garbage collection events, leading to app slowness.

   For more information about how to use Memory Monitor tool, see Viewing Heap Updates (https://developer.android.com/tools/debugging/debugging-memory.html#ViewHeap).

2. The Allocation Tracker tool in Android Studio gives you a detailed look at how your app allocates memory. The Allocation Tracker records an app's memory allocations and lists all allocated objects within the profiling snapshot. You can use this tool to track down parts of your code that allocate too many objects.

   For more information about how to use the Allocation Tracker tool, see Allocation Tracker Walkthrough (https://developer.android.com/studio/profile/allocation-tracker-walkthru.html).

Release memory in response to events

An Android device can run with varying amounts of free memory depending on the physical amount of RAM on the

device and how the user operates it. The system broadcasts signals to indicate when it is under memory pressure, and apps should listen for these signals and adjust their memory usage as appropriate.

You can use the `ComponentCallbacks2` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html) API to listen for these signals and then adjust your memory usage in response to app lifecycle or device events. The `onTrimMemory()` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html#onTrimMemory(int)) method allows your app to listen for memory related events when the app runs in the foreground (is visible) and when it runs in the background.

To listen for these events, implement the `onTrimMemory()` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html#onTrimMemory(int)) callback in your `Activity` (https://developer.android.com/reference/android/app/Activity.html) classes, as shown in the following code snippet.

```
import android.content.ComponentCallbacks2;
// Other import statements ...

public class MainActivity extends AppCompatActivity
    implements ComponentCallbacks2 {

    // Other activity code ...

    /**
     * Release memory when the UI becomes hidden or when system resources become low.
     * @param level the memory-related event that was raised.
     */
    public void onTrimMemory(int level) {

        // Determine which lifecycle or system event was raised.
        switch (level) {

            case ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN:

                /*
                   Release any UI objects that currently hold memory.

                   The user interface has moved to the background.
                */

                break;

            case ComponentCallbacks2.TRIM_MEMORY_RUNNING_MODERATE:
            case ComponentCallbacks2.TRIM_MEMORY_RUNNING_LOW:
            case ComponentCallbacks2.TRIM_MEMORY_RUNNING_CRITICAL:

                /*
                   Release any memory that your app doesn't need to run.

                   The device is running low on memory while the app is running.
                   The event raised indicates the severity of the memory-related event.
                   If the event is TRIM_MEMORY_RUNNING_CRITICAL, then the system will
                   begin killing background processes.
                */

                break;

            case ComponentCallbacks2.TRIM_MEMORY_BACKGROUND:
            case ComponentCallbacks2.TRIM_MEMORY_MODERATE:
            case ComponentCallbacks2.TRIM_MEMORY_COMPLETE:

                /*
                   Release as much memory as the process can.

                   The app is on the LRU list and the system is running low on memory.
                   The event raised indicates where the app sits within the LRU list.
                   If the event is TRIM_MEMORY_COMPLETE, the process will be one of
                   the first to be terminated.
                */

                break;

            default:
                /*
                  Release any non-critical data structures.

                  The app received an unrecognized memory level value
                  from the system. Treat this as a generic low-memory message.
```

```
                */
                break;
        }
    }
}
```

The `onTrimMemory()` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html#onTrimMemory(int)) callback was added in Android 4.0 (API level 14). For earlier versions, you can use the `onLowMemory()` (https://developer.android.com/reference/android/content/ComponentCallbacks.html#onLowMemory()) callback as a fallback for older versions, which is roughly equivalent to the `TRIM_MEMORY_COMPLETE` (https://developer.android.com/reference/android/content/ComponentCallbacks2.html#TRIM_MEMORY_COMPLETE) event.

Check how much memory you should use

To allow multiple running processes, Android sets a hard limit on the heap size alloted for each app. The exact heap size limit varies between devices based on how much RAM the device has available overall. If your app has reached the heap capacity and tries to allocate more memory, the system throws an `OutOfMemoryError` (https://developer.android.com/reference/java/lang/OutOfMemoryError.html).

To avoid running out of memory, you can to query the system to determine how much heap space you have available on the current device. You can query the system for this figure by calling `getMemoryInfo()` (https://developer.android.com/reference/android/app/ActivityManager.html#getMemoryInfo(android.app.ActivityManager.MemoryInfo)). This returns an `ActivityManager.MemoryInfo` (https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.html) object that provides information about the device's current memory status, including available memory, total memory, and the memory threshold—the memory level below which the system begins to kill processes. The `ActivityManager.MemoryInfo` (https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.html) class also exposes a simple boolean field, `lowMemory` (https://developer.android.com/reference/android/app/ActivityManager.MemoryInfo.html#lowMemory) that tells you whether the device is running low on memory.

The following code snippet shows an example of how you can use the `getMemoryInfo()` (https://developer.android.com/reference/android/app/ActivityManager.html#getMemoryInfo(android.app.ActivityManager.MemoryInfo)). method in your application.

```
public void doSomethingMemoryIntensive() {

    // Before doing something that requires a lot of memory,
    // check to see whether the device is in a low memory state.
    ActivityManager.MemoryInfo memoryInfo = getAvailableMemory();

    if (!memoryInfo.lowMemory) {
        // Do memory intensive work ...
    }
}

// Get a MemoryInfo object for the device's current memory status.
private ActivityManager.MemoryInfo getAvailableMemory() {
    ActivityManager activityManager = (ActivityManager) this.getSystemService(ACTIVITY_SERVI
    ActivityManager.MemoryInfo memoryInfo = new ActivityManager.MemoryInfo();
    activityManager.getMemoryInfo(memoryInfo);
    return memoryInfo;
}
```

Use More Memory-Efficient Code Constructs

Some Android features, Java classes, and code constructs tend to use more memory than others. You can minimize how much memory your app uses by choosing more efficient alternatives in your code.

Use services sparingly

Leaving a service running when it's not needed is **one of the worst memory-management mistakes** an Android app can make. If your app needs a service (https://developer.android.com/guide/components/services.html) to perform work in the background, do not keep it running unless it needs to run a job. Remember to stop your service when it has completed

its task. Otherwise, you can inadvertently cause a memory leak.

When you start a service, the system prefers to always keep the process for that service running. This behavior makes services processes very expensive because the RAM used by a service remains unavailable to other processes. This reduces the number of cached processes that the system can keep in the LRU cache, making app switching less efficient. It can even lead to thrashing in the system when memory is tight and the system can't maintain enough processes to host all the services currently running.

You should generally avoid use of persistent services because of the on-going demands they place on available memory. Instead, we recommend that you use an alternative implementation such as `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html). For more information about how to use `JobScheduler` (https://developer.android.com/reference/android/app/job/JobScheduler.html) to schedule background processes, see Background Optimizations (https://developer.android.com/topic/performance/background-optimization.html).

If you must use a service, the best way to limit the lifespan of your service is to use an `IntentService` (https://developer.android.com/reference/android/app/IntentService.html), which finishes itself as soon as it's done handling the intent that started it. For more information, read Running in a Background Service (https://developer.android.com/training/run-background-service/index.html).

Use optimized data containers

Some of the classes provided by the programming language are not optimized for use on mobile devices. For example, the generic `HashMap` (https://developer.android.com/reference/java/util/HashMap.html) implementation can be quite memory inefficient because it needs a separate entry object for every mapping.

The Android framework includes several optimized data containers, including `SparseArray` (https://developer.android.com/reference/android/util/SparseArray.html), `SparseBooleanArray` (https://developer.android.com/reference/android/util/SparseBooleanArray.html), and `LongSparseArray` (https://developer.android.com/reference/android/support/v4/util/LongSparseArray.html). For example, the `SparseArray` (https://developer.android.com/reference/android/util/SparseArray.html) classes are more efficient because they avoid the system's need to autobox the key and sometimes value (which creates yet another object or two per entry).

If necessary, you can always switch to raw arrays for a really lean data structure.

Be careful with code abstractions

Developers often 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. So if your abstractions aren't supplying a significant benefit, you should avoid them.

For example, enums often require more than twice as much memory as static constants. You should strictly avoid using enums on Android.

Use nano protobufs for serialized data

Protocol buffers (https://developers.google.com/protocol-buffers/docs/overview) are a language-neutral, platform-neutral, extensible mechanism designed by Google for serializing structured data—similar to XML, but smaller, faster, and simpler. If you decide to use protobufs for your data, you should always use nano protobufs in your client-side code. Regular protobufs generate extremely verbose code, which can cause many kinds of problems in your app such as increased RAM use, significant APK size increase, and slower execution.

For more information, see the "Nano version" section in the protobuf readme (https://android.googlesource.com/platform/external/protobuf/+/master/java/README.txt) .

Avoid memory churn

As mentioned previously, garbage collections events don't normally affect your app's performance. However, many garbage collection events that occur over a short period of time can quickly eat up your frame time. The more time that the system spends on garbage collection, the less time it has to do other stuff like rendering or streaming audio.

Often, *memory churn* can cause a large number of garbage collection events to occur. In practice, memory churn describes the number of allocated temporary objects that occur in a given amount of time.

For example, you might allocate multiple temporary objects within a `for` loop. Or you might create new `Paint` (https://developer.android.com/reference/android/graphics/Paint.html) or `Bitmap` (https://developer.android.com/reference/android/graphics/Bitmap.html) objects inside the `onDraw()` (https://developer.android.com/reference/android/view/View.html#onDraw(android.graphics.Canvas)) function of a view. In both cases, the app creates a lot of objects quickly at high volume. These can quickly consume all the available memory in the young generation, forcing a garbage collection event to occur.

Of course, you need to find the places in your code where the memory churn is high before you can fix them. Use the tools discussed in Analyze your RAM usage (#AnalyzeRam)

Once you identify the problem areas in your code, try to reduce the number of allocations within performance critical areas. Consider moving things out of inner loops or perhaps moving them into a Factory (https://en.wikipedia.org/wiki/Factory_method_pattern) based allocation structure.

Remove Memory-Intensive Resources and Libraries

Some resources and libraries within your code can gobble up memory without you knowing it. Overall size of your APK, including third-party libraries or embedded resources, can affect how much memory your app consumes. You can improve your app's memory consumption by removing any redundant, unnecessary, or bloated components, resources, or libraries from your code.

Reduce overall APK size

You can significantly reduce your app's memory usage by reducing the overall size of your app. Bitmap size, resources, animation frames, and third-party libraries can all contribute to the size of your APK. Android Studio and the Android SDK provide multiple tools to help you reduce the size of your resources and external dependencies.

For more information about how to reduce your overall APK size, see Reduce APK Size (https://developer.android.com/topic/performance/reduce-apk-size.html).

Use Dagger 2 for dependency injection

Dependency injection frameworks can simplify the code you write and provide an adaptive environment that's useful for testing and other configuration changes.

If you intend to use a dependency injection framework in your app, consider using Dagger 2 (http://google.github.io/dagger/) . Dagger does not use reflection to scan your app's code. Dagger's static, compile-time implementation means that it can be used in Android apps without needless runtime cost or memory usage.

Other dependency injection frameworks that use reflection tend to initialize processes by scanning your code for annotations. This process can require significantly more CPU cycles and RAM, and can cause a noticeable lag when the app launches.

Be careful about using external libraries

External library code is often not written for mobile environments and can be inefficient when used for work on a mobile client. When you decide to use an external library, you may need to optimize that library for mobile devices. Plan for that work up-front and analyze the library in terms of code size and RAM footprint before deciding to use it at all.

Even some mobile-optimized libraries can cause problems due to differing implementations. For example, one library may use nano protobufs while another uses micro protobufs, resulting in two different protobuf implementations in your app. This can happen with different implementations of logging, analytics, image loading frameworks, caching, and many other things you don't expect.

Although ProGuard (https://developer.android.com/tools/help/proguard.html) can help to remove APIs and resources with the right flags, it can't remove a library's large internal dependencies. The features that you want in these libraries may require lower-level dependencies. This becomes especially problematic when you use an `Activity` (https://developer.android.com/reference/android/app/Activity.html) subclass from a library (which will tend to have wide swaths of dependencies), when libraries use reflection (which is common and means you need to spend a lot of time manually tweaking ProGuard to get it to work), and so on.

Also avoid using a shared library for just one or two features out of dozens. You don't want to pull in a large amount of

code and overhead that you don't even use. When you consider whether to use a library, look for an implementation that strongly matches what you need. Otherwise, you might decide to create your own implementation.