

Memory Monitor

Android Monitor (<https://developer.android.com/tools/help/android-monitor.html>) provides a Memory Monitor so you can more easily monitor app performance and memory usage to find deallocated objects, locate memory leaks, and track the amount of memory the connected device is using. The Memory Monitor reports how your app allocates memory and helps you to visualize the memory your app uses. It lets you:

- Show a graph of available and allocated Java memory over time.
- Show garbage collection (GC) events over time.
- Initiate garbage collection events.
- Quickly test whether app slowness might be related to excessive garbage collection events.
- Quickly test whether app crashes may be related to running out of memory.

In this document

- Memory Monitor Workflow
 - Garbage collection roots and dominator trees
 - Memory leak and use analysis
 - Memory management for different virtual machines
- Displaying a Running App in the Memory Monitor
- Forcing a Garbage Collection Event
- Taking a Snapshot of the Java Heap and Memory Allocation

See also

- Managing Your App's Memory
- Addressing Garbage Collection Issues
- Investigating Your RAM Usage

Memory Monitor Workflow

To profile and optimize memory use, the typical workflow is to run your app and do the following:

1. Profile the app using the Memory Monitor to find out whether undesirable garbage collection event patterns might be causing performance problems.
2. If you see many garbage collection events in a short amount of time, dump the Java heap to identify candidate object types that get or stay allocated unexpectedly or unnecessarily.
3. Start allocation tracking to determine where any problems are happening in your code.

The Java heap data shows in real-time what types of objects your application has allocated, how many, and their sizes on the heap. Viewing the heap helps you to:

- Get a sense of how your app allocates and frees memory.
- Identify memory leaks.

Allocation tracking records app memory allocations and lists all allocations for the profiling cycle, including the call stack, size, and allocating code. It helps you to:

- Identify where many similar object types, from roughly the same call stack, are allocated and deallocated over a very short period of time.
- Find the places in your code that may contribute to inefficient memory use.

Garbage collection roots and dominator trees

When you dump the Java heap, the Memory Monitor creates an Android-specific Heap/CPU Profiling (HPROF) file that you can view in the HPROF Viewer. The HPROF Viewer indicates a garbage collection root with the 📌 icon (and a depth of zero) and a dominator with the 🏠 icon.

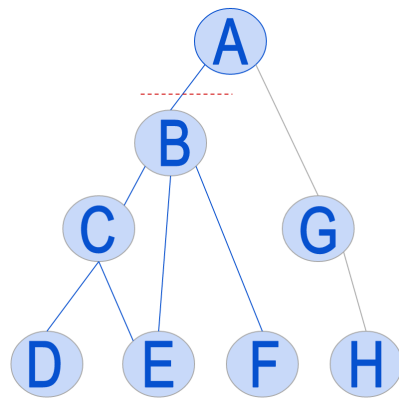
There are several kinds of garbage collection roots in Java:

- references on the stack
- Java Native Interface (JNI) native objects and memory
- static variables and functions
- threads and objects that can be referenced

- classes loaded by the bootstrap loader
- finalizers and unfinalized objects
- busy monitor objects

The HPROF file provides the list of roots to the HPROF Viewer.

A dominator tree traces paths to objects created by the app. An object dominates another object if the only way to reach the other object is, directly or indirectly, through the dominator object. When you examine objects and paths created by an app in an effort to optimize memory use, try to remove objects that are no longer needed. You can release a dominator object to release all subordinate objects. For example, in the following figure, if you were to remove object B, that would also release the memory used by the objects it dominates, which are objects C, D, E, and F. In fact, if objects C, D, E, and F were marked for removal, but object B was still referring to them, that could be the reason that they weren't released.



Memory leak and use analysis

An app performs better if it uses memory efficiently and releases the memory when it's no longer needed. Memory leaks that are large or that grow over time are the most important to correct.

One way to optimize memory usage is to analyze large arrays. For example, can you reduce the size of individual elements in the array to save memory?

Another area that deserves attention is objects that the app no longer needs but continues to reference. You can gather heap dumps over different periods of time and compare them to determine if you have a growing memory leak, such as an object type that your code creates multiple times but doesn't destroy. These objects could be part of a growing array or an object tree, for example. To track down this problem, compare the heap dumps and see if you have a particular object type that continues to have more and more instances over time.

Continually growing object trees that contain root or dominator objects can prevent subordinate objects from being garbage-collected. This issue is a common cause of memory leaks, out-of-memory errors, and crashes. Your app could have a small number of objects that are preventing a large number of subordinate objects from being destroyed, so it runs out of memory quickly. To find these issues, get a heap dump and examine the amount of memory held by root and dominator objects. If the memory is substantial, you've likely found a good place to start optimizing your memory use.

As you start narrowing down memory issues, you should also use the Allocation Tracker to get a better understanding of where your memory-hogging objects are allocated. The Allocation Tracker can be valuable not only for looking at specific uses of memory, but also for analyzing critical code paths, such as loading and scrolling. For example, tracking allocations when flinging a list in your app allows you to see all of the allocations that need to be done for that behavior, what thread they are on, and where they came from. This information is extremely valuable for tightening up these paths to reduce the work they need and improve the overall smoothness of the UI.

It's useful to examine your algorithms for allocations that are unnecessary or that create the same object many times instead of reusing them. For example, do you create temporary objects and variables within recursive loops? If so, try creating an object or variable before the loop for use within the loop. Otherwise, your app might needlessly allocate many objects and variables, depending on the number of recursions.

It's important to perform allocation tests on portions of your code that create the most and largest objects, as those areas offer the most optimization opportunities. In addition to unit tests, you should test your app with production-realistic data loads, especially those algorithms that are data-driven. Also, make sure to account for the app caching and startup phase, which can sometimes be slow; allocation analysis is best done after that phase to produce accurate results.

After you optimize code, be sure to test that it worked. You need to test under different load conditions and also without running the Memory Monitor tools. Compare results before and after optimization to make sure that

performance has actually improved.

Memory management for different virtual machines

Android Monitor uses the Virtual Machine (VM) that the device or emulator uses:

- Android 4.3 (API level 18) and lower uses the Dalvik VM.
- In Android 4.4 (API level 19), the Android RunTime (ART) VM is an option, while the Dalvik VM is the default.
- Android 5.0 (API level 21) and higher uses the ART VM.

The VM handles garbage collection. The Dalvik VM uses a mark-and-sweep scheme for garbage collection. The ART VM uses a generational scheme, combined with mark-and-sweep when memory needs a more thorough garbage collection, such as when memory becomes excessively fragmented. The logcat Monitor displays some messages that indicate the type of garbage collection that occurred and why.

Memory Monitor results can vary between the different VMs. As a result, if you’re supporting both VMs, you might want to test with both. In addition, the VMs available for different API levels can have different behavior. For example, the Dalvik VM in Android 2.3 (API level 10) and lower uses externally allocated memory while higher versions allocate in the Dalvik heap only.


You can’t reconfigure the Dalvik and ART VMs to tune performance. Instead, you should examine your app code to determine how to improve its operation, for example, reducing the size of very large arrays.

There are programmatic ways to manipulate when the VM performs garbage collection, although it’s not a best practice. These techniques can be specific to the VM. For more information, see [Addressing Garbage Collection \(GC\) Issues](https://developer.android.com/guide/practices/verifying-apps-art.html#GC_Migration) (https://developer.android.com/guide/practices/verifying-apps-art.html#GC_Migration) and [Investigating Your RAM Usage](https://developer.android.com/tools/debugging/debugging-memory.html) (<https://developer.android.com/tools/debugging/debugging-memory.html>).

The ART VM adds a number of performance, development, and debugging improvements over the Dalvik VM. For more information, see [ART and Dalvik](https://source.android.com/devices/tech/dalvik/index.html) (<https://source.android.com/devices/tech/dalvik/index.html>).

Displaying a Running App in the Memory Monitor

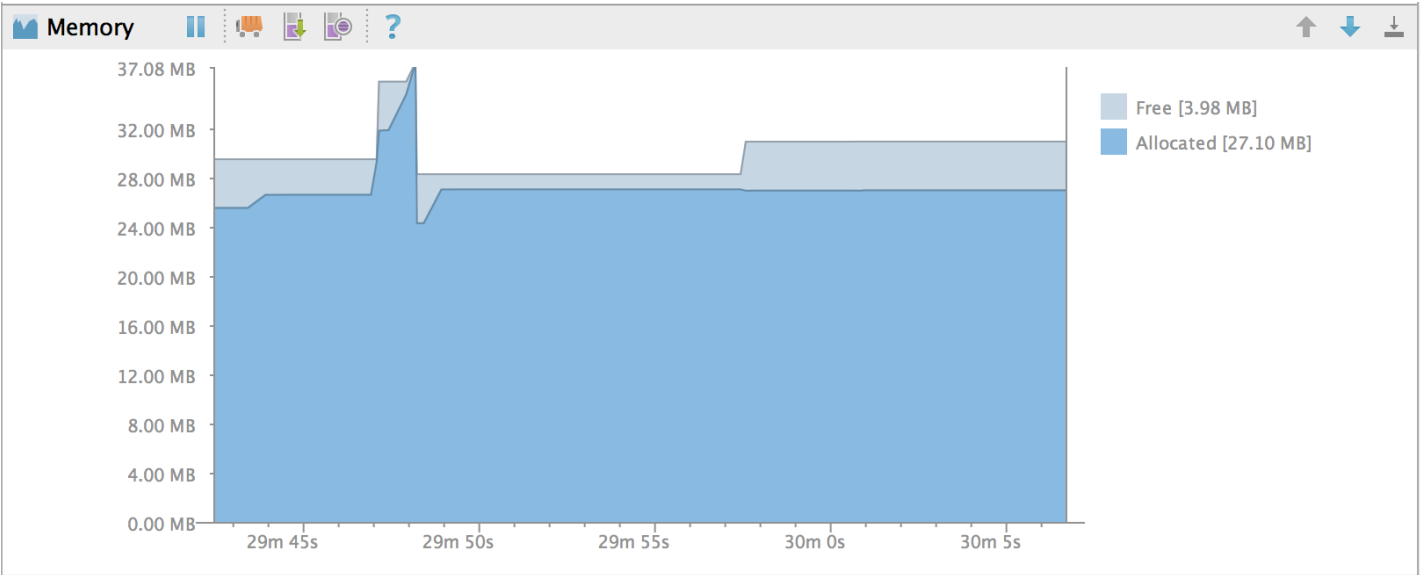
To display an app running on a particular device or emulator in the Memory Monitor:

1. Meet the prerequisites and dependencies (<https://developer.android.com/tools/help/am-basics.html#byb>).
2. Open an app project.
3. Run the app (<https://developer.android.com/tools/building/building-studio.html#RunningApp>) on a hardware device or emulator.
4. Display Android Monitor (<https://developer.android.com/tools/help/am-basics.html#displaying>).
5. Click the **Monitors** tab and display the Memory Monitor (<https://developer.android.com/tools/help/am-basics.html#rearranging>).
6. Enable the Memory Monitor by clicking Pause  to deselect it.

In the graph, the y-axis displays the free and allocated RAM in megabytes. The x-axis shows the time elapsed; it starts with seconds, and then minutes and seconds, and so on. The amount of free memory, measured in megabytes, is shown in a light color, and allocated memory is a darker color. When there’s a sharp drop in allocated memory, that indicates a garbage collection event.

To force a garbage collection event, click Initiate GC .

In the following figure, the VM initiated the first garbage collection event, while the developer forced the second.




7. Interact with your app and watch how it affects memory usage in the Memory Monitor. You can identify garbage collection patterns for your app and determine whether they're healthy and what you expect.

The graph can show you potential issues:

- Excessive garbage collection events slow down the app.
- The app runs out of memory, which causes it to crash.
- Potential memory leaks.

For example, you might see the following signs of problems:

- Your app is static, but you see memory being allocated in the monitor.
- You see spikes of memory allocations in the monitor, but you don't think there's any app logic to cause this behavior.

8. To stop the Memory Monitor, click Pause  again to select it.

Forcing a Garbage Collection Event

Normally, VMs perform garbage collection only when absolutely needed, since it's expensive. However, it can be useful to force garbage collection in certain circumstances. For example, when locating memory leaks, if you want to determine whether a large object was successfully released already, you can initiate garbage collection much more aggressively than usual.

To force a garbage collection event:

- While the Memory Monitor is running (`#displaying`), click Initiate GC .

Taking a Snapshot of the Java Heap and Memory Allocation

You can take snapshots while the Memory Monitor is running or paused:

- To take and display a snapshot of the Java heap, see HPROF Viewer and Analyzer (<https://developer.android.com/tools/help/am-hprof.html>).
- To take and display a snapshot of memory allocation, see Allocation Tracker (<https://developer.android.com/tools/help/am-allocation.html>).