Android reduce memory usage programmatically

Random Access Memory (RAM) is a valuable resource in any software development environment, but it is even more valuable in a mobile operating system where physical memory is often limited. While both Android Runtime (ART) and Dalvik's virtual machine do routine garbage collection, it doesn't mean you can ignore when and where your app allocates and releases memory. You still need to avoid entering memory leaks, usually caused by keeping on subject references in static member variables, and to release any reference objects at the aging time as defined by life cycle callbacks. This page explains how you can proactively reduce your memory usage in your app. For information on how the Android operating system manages memory, see the Android Memory Management Review. Monitoring available memory and memory usage Before you can fix problems with the use of memory in the app, you first have to find them. The memory profile in Android Studio will help you find and diagnose memory problems as follows: See how your app distributes memory over time. The memory profile shows a real-time graph of how much memory the app uses, the number of java objects allocated, and when garbage is collected. Initiate garbage collection events and take a snapshot of the Java heap while the app is running. Lock up the app's memory distribution, then inspect all the highlighted objects, review the stack tracing for each selection, and go to the appropriate code in the Android Studio editor. Release memory in response to events described in the Android Memory Management Review, Android can recover memory from your app in several ways or kill your app completely if necessary to free up memory for critical tasks. To further balance the system's memory and avoid having to kill the application process, you can implement the ComponentCallbacks2 interface in Activity classes. The onTrimMemory callback method allows your app to listen to memory events when your app is in the foreground or in the background, and then release objects in response to the app's lifecycle or system events indicating that the system needs to restore memory. For example, you can implement onTrimMemory () callbacks to respond to various memory-related events, as shown here: import android.content.ComponentCallbacks2 / Other import statements ... MainActivity Class : AppCompatActivity (), ComponentCallbacks2 // Another activity code ... / q - Release memory when the user interface becomes hidden or when system resources become low. @param level of the event, with the memory of the ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN that was raised. The user interface has faded into the background. */ } } ComponentCallbacks2.TRIM_MEMORY_RUNNING_LOW, ComponentCallbacks2.TRIM_MEMORY_RUNNING_CRITICAL - Release any memory your app doesn't need to run. The device doesn't have memory while the app is running. The raised event indicates the severity of the memory-related event. If the event TRIM_MEMORY_RUNNING_CRITICAL, the system will start killing the background processes. The questions and ComponentCallbacks2.TRIM_MEMORY_BACKGROUND ComponentCallbacks2.TRIM_MEMORY_MODERATE, ComponentCallbacks2.TRIM_MEMORY_COMPLETE - - Release as much memory as the process can do. The app is on the LRU list and the system works with low memory. The raised event indicates where the app is on the LRU list. If the event TRIM_MEMORY_COMPLETE, the process will be one of the first to be terminated. B/ - - Release any not critical data structures. The application received an unrecognized memory level value from the system. Treat this as a birth message with low memory. Import android.content.ComponentCallbacks2; Other import statements ... MainActivity's public class extends the action of AppCompatActivity, implements ComponentCallbacks2 @param /ComponentCallbacks2.TRIM_MEMORY_UI_HIDDEN/ Another activity code ... // Release memory when the user interface becomes hidden or when the system's resources become low. The user interface has faded into 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 doesn't have memory while the app is running. The raised event indicates the severity of the memory-related event. If the event TRIM_MEMORY_RUNNING_CRITICAL, the system will start killing the 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 do. The app is on the LRU list and the system works with low memory. The raised event indicates where the app is on the LRU list. If the event TRIM_MEMORY_COMPLETE, the process will be one of the first to be terminated. /break; default: / - Release any not critical data structures. The application received an unrecognized memory level value from the system. Treat this as a birth message with low memory. /break; The onTrimMemory callback was added to Android 4.0 (API level 14). For earlier versions, you can use onLowMemory, which is roughly equivalent to TRIM_MEMORY_COMPLETE events. Check how much memory you should use to allow a few Processes, Android sets a tight limit on the size of the heap, different for each application. The exact heap size limit varies between devices depending on how much RAM the device is available overall. If your app has reached heap capacity and tries to highlight more memory, the system throws OutOfMemoryError. To avoid memory loss, you can request a system to determine how much space in the pile you have on your current device. You can request the system for this number by calling getMemoryInfo. This returns the ActivityManager.MemoryInfo object, which provides information about the device's current memory status, including available memory, shared memory, and memory threshold - the level of memory at which the system begins to kill processes. ActivityManager.MemoryInfo also provides a simple boolean, lowMemory, that tells you whether the device works with low memory. The following piece of code shows an example of how you can use getMemoryInfo in an application. fun doSomethingMemoryIntensive () before doing something that requires a lot of memory, / Check whether the device is in a low memory state. If (!getAvailableMemory ().. lowMemory) / / / Have intense memory work ... Private Pleasure getAvailableMemory (): ActivityManager.MemoryInfo - val activityManager - getSystemService (Context.ACTIVITY_SERVICE) as ActivityManager return ActivityManager.MemoryInfo (zgt;)... ActivityManager.MemoryInfo memoryInfo - getAvailableMemory(); if (!memoryInfo.lowMemory) - / / Intensive memory work ... Get a MemoryInfo object for your device's current state of mind. Private activityManager.MemoryInfo getAvailableMemory (ActivityManager) this.getSystemService (ACTIVITY_SERVICE); ActivityManager.MemoryInfo memoryInfo - new ActivityManager.MemoryInfo activityManager.getMemoryInfo (memoryInfo); Bring back the memoryInfo; Using more efficient memory code builds some Android features, Java classes and code designs tend to use more memory than others. You can minimize the amount of memory your app uses by choosing more effective alternatives in the code. Using services sparingly leaving the service running when it doesn't need to is one of the worst memory management mistakes an Android app can make. If an app needs a service to do work in the background, don't keep it on a workday unless it needs to do a job. Don't forget to stop the service when it has completed its task. Otherwise, you may inadvertently cause a memory leak. When you start a service, the system prefers to keep the process always This behavior makes service processes very expensive because the RAM used by the service remains inaccessible to other processes. This reduces the number of cached processes the system can store in the LRU cache, making switching applications less efficient. This can even lead to thrashing in the system when memory is limited and the system cannot maintain enough processes to accommodate all the services currently running. As a general rule, you should avoid using permanent services because of the constant requirements that they are subject to accessible memory. Instead, we recommend using an alternative implementation such as JobScheduler. For more information on how to use JobScheduler to plan background processes, see If you need to use the service, the best way to limit your lifespan is to use IntentService, which shuts down as soon as it's done processing the intent that started it. For more information, read Start in the Helpline. Using optimized data containers Some classes provided by the programming language are not optimized for use on mobile devices. For example, a shared implementation of HashMap can be a completely inefficient memory because each display requires a separate login object. The Android structure includes several optimized data containers, including SparseArray, SparseBooleanArray and LongSparseArray. For example, SparseArray classes are more effective because they avoid the need for an automatic key box and sometimes a value (which creates another object or two on the record). 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 good programming practice because abstractions can improve code flexibility and maintenance. However, abstractions cost a considerable amount: they usually require a fair amount of more code to be executed, requiring more time and more RAM for that code to be displayed in memory. So if your abstractions don't bring a significant advantage, you should avoid them. Using lightweight proto-puffs for serialized data protocol buffers is a neutral language, neutral with a platform, diluted by a mechanism developed by Google to serialize structured data, like XML, but smaller, faster, and easier. If you decide to use protobufs for your data, you should always use lightweight proto-puffs in the code on the client's side. Regular proto-puffs generate extremely verbose code, which can cause a lot of problems in such as increased RAM use, a significant increase in the size of the APK, and slower execution. For more information, see Lite version of the section in protobuf readme. Avoid memory outflows As mentioned earlier, garbage collection events do not affect the performance of the application. However, many garbage collection events that occur over a short period of time can quickly be quickly battery, and slightly increase the time frame set up due to the necessary interactions between the garbage collector and the app streams. The more time the system spends on garbage disposal, the faster the battery drains. Often, memory outflows can lead to a large number of garbage collection events. In practice, memory outflows describe the number of time objects allocated that occur at a given time. For example, you can identify several time objects in a loop. Or you can create new Paint or Bitmap objects inside the onDraw () view function. In both cases, the app quickly creates multiple objects at high volume. They can quickly consume all available memory in the younger generation, forcing a garbage collection event to occur. Of course, you need to find places in the code where the outflow of memory is high before you can fix them. To do this, you should use the memory profile in Android Studio. Once you've identified problem areas in the code, try to reduce the number of allocations in critical performance areas. Consider moving things from internal cycles or possibly moving them to a factory-based distribution structure. Another possibility is to assess whether object pools are useful if used. With an object pool, instead of dropping a copy of an object on the floor, you release it into the pool as soon as you no longer need it. The next time you want an instance of this type, you can buy it from the pool, not highlight it. A thorough performance assessment is important to determine whether an object pool is appropriate in a given situation. There are times when object pools can impair performance. Although pools avoid allotment, they introduce other overheads. For example, maintaining a pool usually involves synchronization that has a negligible overhead. In addition, cleaning a sample of a syfused object (to avoid memory leaks) during release, and then initiating it during purchase may have non-zero overheads. Finally, keeping more objects in a pool than we would like also puts a burden on GK. While object pools reduce the number of GC calls, they eventually increase the amount of work that needs to be done for each call, as this is proportional to the number of live (achievable) bytes. Some resources and libraries in the code can gobble up memory without knowing it. The total size of your APK, including third-party libraries or built-in resources, can affect the amount of memory your app consumes. You can improve your application's memory consumption by removing any redundant, unnecessary, or bloated components, resources, or libraries from the code. Reducing the overall size of APK can significantly reduce memory usage by reducing the overall size of the application. Bitmap size, resources, animation frames and third-party libraries can the size of your APK. Android Studio and Android SDK provide a few tools to help you reduce your resources and external dependencies. These tools support modern code compression techniques, such as R8 compilation. (Android Studio 3.3 and lower use of ProGuard instead of R8 compilation.) For more information on how to reduce the overall size of the APK, see a guide on how to reduce the size of your app. Using Dagger 2 for dependency framework for dependency injection can simplify the code you write and provide an adaptive environment that is useful for testing and other configuration changes. If you're going to use a dependency injection framework in your app, consider using Dagger 2. The dagger does not use reflection to scan the application code. The static implementation of Dagger, a compilation of time means that it can be used in Android applications without unnecessary time running time or using memory. Other dependency injection frameworks that use reflection tend to initiate processes by scanning the code on annotations. This process can require significantly more CPU and RAM cycles, and can cause a noticeable lag in the launch of the application. Be careful with using external libraries External library code is often not written for mobile environments and can be ineffective when used to work on a mobile client. When you decide to use an external library, you may need to optimize this library for mobile devices. Plan this work for the foreground and analyze the library in terms of code size and RAM before deciding to use it at all. Even some mobile optimized libraries can cause problems due to different implementations. For example, one library may use lightweight protoboofs, while another uses microprobes, resulting in two different protobutu implementations in the application. This can happen with different logs, analytics, framework, image downloads, caching, and many other things you don't expect. While ProGuard can help remove APIs and resources with the right flags, it can't remove large internal library dependencies. The features that are required in these libraries may require lower-level dependencies. This becomes particularly problematic when you use the Activity subclass from the library (which usually has wide dependency strips), when libraries use reflection (which is common and means you need to spend a lot of time manually setting up ProGuard to make it work), and so on. Also, avoid using a shared library for just one two functions out of dozens. You don't want to pull in a lot of code and overheads that you don't even use. If you're thinking about using a library, look for an implementation that strongly matches what you need. Otherwise, you can create your own implementation. Implementation. Implementation. reduce memory usage android app programmatically