Research on Memory Leakage Monitoring Based on Android Mobile Platform

Di Zhou[1], Peixing Yang[1]
1.College of Computer Science and Technology , Nanjing University of Aeronautics and Astronautics

Key Words: Memory Leakage, Monitoring, Control, Android

Abstract. An excellent Android application shall not only have perfect function, but also have good user experience. Actually, performance is an important factor influencing user experience. Memory leakage is a common performance problem in Android development, and DDMS with a good memory monitoring tool Heap in Android tools can detect the memory change of a process. By virtue of this tool, we can roughly test whether an application has memory leakage probability, thus to explain the whole process of a memory leakage problem, namely from problem discovery to analysis & location and to final solution.

Introduction

AndroidApp is based on virtual machines and the memory management thereof is realized by Dalvik, and GC untimely collects the memory garbage, for example: after one activity is finished, the reference object of the memory thereof will be calculated according to the collection algorithm when GC collects memory garbage at next time; if this part of the memory belongs to the collectable object, then the garbage object will be collected at the same time; if we suspect that an operation or an interface has memory leakage, then it is necessary to repeatedly execute the operation or repeatedly open and close the interface. Theoretically, each ESC operation is corresponding to large memory freeing, so if there is any memory leakage, as shown in the following figure, the memory will be increasingly occupied when the reading interface of Reader is repeatedly opened and closed; in other words, at each time for closing the activity, the memory that shall be freed is not freed.

Memory Leakage Monitoring

2.1 Memory leakage caused by reference freeing failure

Such Android memory leakage as the memory leakage caused by registration cancelation failure is more serious than pure Java memory leakage, because other Android programs may reference the object (e.g. registration mechanism) of our Android program. Even though our Android program is stopped, other reference program still references a certain object of our Android program and the leaked memory still cannot be collected as memory garbage. Example 1: if we want to monitor the telephone service in the system in LockScreen in order to obtain some information (e.g. signal strength), then we can define a PhoneStateListener object in LockScreenand this object is registered in TelephonyManager service at the same time. For LockScreen object, a LockScreen object shall be created when LockScreen needs to be displayed, and this LockScreen object will be freed when LockScreen disappears. However, if we forget to cancel PhoneStateListener object already registered when freeing LockScreen object, LockScreen will not be collected as memory garbage. If LockScreen is continuously displayed and closed, OutOfMemory will be finally caused due to the collection failure of many LockScreen objects and system_process will be accordingly hung up. Although some system programs can automatically cancel relevant registration (such cancellation is certainly untimely), we still need to clearly cancel relevant registration in our programs, and all registrations shall be canceled at the end of program.

Memory leakage caused by object clearing failure: we usually add some object references to a

set, but we fail to remove relevant reference from the set when we don't need the object, thus making the set become larger and larger. For a static set, the corresponding consequence is more serious. Memory leakage caused by unclosed resource object: the resource object (e.g. Cursor, File) usually occupies some buffers, so if such objects are not used, it is necessary to timely close them in order to timely collect the buffers to the memory, wherein such buffers not only exist inside Java virtual machine, but also exist outside Java virtual machine. If we only set the reference as null rather than close these objects, memory leakage will be usually caused. For such resource object as SQLiteCursor (if we don't close finalize(), this function will automatically call close()), if we don't close such object, the system will close it during thecollecting process of the system, but the efficiency is too low. Therefore, when the resource object is not used, it is necessary to call thecorrespondingclose() function to close the object and then set it as null. When exiting from a program, please be sure that our resource object has been closed.

Database query operation is often implemented in programs, but Cursor is frequently not closed after use. If our query result set is small and the memory consumption cannot be easily discovered, the memory problem can appear only under the condition of long-time frequent operations. As a result, difficulty and risk will be brought to future test and troubleshooting.

2.2 Memory pressure caused by some bad codes

Some codes cannot cause memory leakage, but they either fail to effectively and timely free the memory not in use or fail to effectively use the existing object but frequently apply for new memory, thus significantly influencing memory collection and allocation. As a result, the virtual machine will be easily forced to allocate more memory to the application process and memory consumption will be unnecessarily occupied.

Bitmap doesn't call collect(), so when Bitmap object is not used, we shall firstly call collect() for memory freeing and then set it as null. Viewed from the source code of collect(), the main memory of Bitmap shall be freed immediately after collect() is called, but the test result shows that the main memory is not freed immediately. In my opinion, such call shall significantly accelerate the freeing process of the main memory of Bitmap.

The cached convertView is not used for constructing Adapter. In this article, the construction of ListViewBaseAdapter is taken as an example, and the following method is provided for BaseAdapter: public View getView(int position, View convertView, ViewGroup parent), thus to provide the view object needed by each item to ListView. In the initial stage, ListViewinstantiates a certain quantity of view objects in BaseAdapter according to the present screen layout and meanwhile caches these view objects. When ListView is upwards rolled, the view object of the list item originally located at the uppermost part will be collected and then used for constructing the new nethermost list item. The above construction process is achieved by getView() method, and the second formal parameterView convertView of getView() is namely the view object of the cached list item (if there is no view object in the cache during the initialization, then convertView shall benull).

According to above analysis, if we newly instantiate one View object in getView() at each time rather than use convertView, we will waste time and cause memory garbage and bring difficulty to garbage collection. If the memory garbage is not timely collected, the virtual machine has to allocate more memory to the application process and accordingly causes unnecessary memory consumption. Additionally, ListView process for collecting view object of list item can be checked.

Memory Leakage Control

Memory leakage avoidance notices for Android: collect() shall be called to free the memory when Bitmap object is not used. We know that the cause of the memory leakage in Java language is as follows: reference is adopted for accessing the memory object in Java, and the object no longer referenced will be collected by the garbage collector, but if the object is referenced, then it will not be collected by the garbage collector, even though the virtual machine is abnormal. Therefore, when some objects are not needed in the memory, we still can reserve relevant memory and the reference thereof. In consideration of the above reasons, when using the object in an Activity, we need to

notice that the life cycle of the object shall not be more than that of the Activity in order to avoid the resource waste caused by database Cursor closure failure.

In order to make full use of contentview and reduce view object creation, the static class can be used to process and optimize getview process method: ddms->heapsize->adtaobject->total size. In Android program, Context is often used for many operations such as resource loading and accessing, and this is why each Widget accepts one Context parameter in its constructor. In a normal Android application program, you can find two types of Context, namely: Activity and Application. However, the first type of Context is usually introduced into the class and the method needing one Context, and this means that View can reference the whole Activity and possess all contents of the Activity, namely: the whole View layer and all resources thereof. Therefore, if you "leak" Context ("leak" means that you reserve one reference and prevent GC from collecting memory garbage), you will leak more memory. If you are not careful enough, an Activity can be easily leaked. When the screen direction is changed, the system will usually destroy the present Activity, create a new Activity and maintain its state. In this way, Android will reload UI of the application program from the resources. If your application program has large Bitmap and you don't want to reload it for each rotation, the simplest way is to save the program in a static field. When one Drawable is added to one View, View will regard it as one callback and set it on the Drawable. The above code snippet means that Drawable has one TextView reference, but TextView has Activity (Context type) reference; in other words, Drawable has more object references (this depends on your codes). This is one of the examples that Context can be most easily leaked, and such case is processed in Home Screen source code as follows (search nbindDrawables()): when an Activity is destroyed, it is necessary to set callback of the saved Drawable as null. Interestingly, there are still many terrible Context leakage cases in which the application program can rapidly and completely consume the memory.

There are two simple methods for avoiding the memory leakage related to Context. The simplest solution is to prevent Context from exceeding its own scope. Besides the static reference given in the codes of the above example, the implicitreferences of the internal and external classes are also very dangerous. The second solution is to adopt the other Context type ---- Application, and this Context has the life cycle as long as that of the application program and is independent of the life cycle of an Activity. If you intend to reserve an object for a long time and need one Context, please remember to adopt Application object. You can call Context.getApplicationContext() or Activity.getApplication() to easily get an Application object. Additionally, the long-time reference of Context-Activity shall not be kept (for a Activity reference, it is necessary to ensure the life cycle as long as that of the Activity), and you can try to adopt Context-Application to replace Context-Activity. If you don't want to control the life cycle of the internal class, it is necessary to adopt static internal class rather than non-static internal class in an Activity, and meanwhile create a weak reference therein. The solution for such case is to adopt a static internal class which has the WeakReference of the external class, and similarly toViewRoot and its Winner class, GC (garbage collector) cannot solve memory leakage problem.

After logcat check, there is no any abnormality in item XXX code except the following one: INFO/ActivityManager(209): Process com.xxx (pid 18396) has died.This sentence indicates that the process has died and the Application has been restarted and all classes have been initialized. The process is preliminarily speculated to be killed due to memory leakage. For checking memory leakage, it is necessary to open ddms and update heap view and then test the Application. In the end, we find that the heap space occupation is continuously increased when two interfaces are alternatelyswitched, and this finally causes the process to be restarted.

Conclusion

Memory leakage will reduce available memory and accordingly reduce computer performance. In the most terrible case, excessive available memory will be allocated finally, thus causing all or part of equipment to stop normal operation or causing application program crash. Memory leakage may be not serious and even can be detected by normal measures. In modern operating system, the

normal memory occupied by an application program will be freed at the end of program, thus indicating that the memory leakage in a short-time running application program will not cause serious consequence.

Reference

[1] Jie He, YishuangGeng, KavehPahlavan, Toward Accurate Human Tracking: Modeling Time-of-Arrival for Wireless Wearable Sensors in Multipath Environment, IEEE Sensor Journal, 14(11), 3996-4006, Nov. 2014

[2] Na Lu, Caiwu Lu, Zhen Yang, YishuangGeng, Modeling Framework for Mining Lifecycle Management, Journal of Networks, 9(3), 719-725, Jan. 2014

[3] YishuangGeng, KavehPahlavan, On the accuracy of rf and image processing based hybrid localization for wireless capsule endoscopy, IEEE Wireless Communications and Networking Conference (WCNC), Mar.2015

[4] Guanxiong Liu, YishuangGeng, KavehPahlavan, Effects of calibration RFID tags on performance of inertial navigation in indoor environment, 2015 International Conference on Computing, Networking and Communications (ICNC), Feb. 2015