技术小黑屋

一个有态度的技术分享媒体

- RSS

Search

Navigate…

- 博客
- 存档
- 关于
- 文章订阅

Google IO：Android内存管理主题演讲记录

Nov 2nd, 2014

翻出了3年前的Google IO大会的主题演讲 Google IO 2011 Memory management for Android Apps，该演讲介绍了Android系统在垃圾回收上的变化和如何发现并内存泄露以及如何管理Android中的内存。本演讲对开发者还是有很大的帮助。

目前全文内容为英文，但是大部分很容易理解。以下内容在原有基础上，进行了分段整理,更加便于阅读。

视频

演讲全文内容

Hi everybody,

My name’s Patrick Dubroy and today I’m going to talk to you about memory management for Android. So I’m really happy to see so many people here who care about memory management, especially near the end of the day.

So let’s get started. So I’m sure you all remember this device. This is the T-Mobile G1. Hugo talked about it in the keynote yesterday. It was released about two and a half years ago. So, is there anybody here who actually developed on the G1? All right, impressive. That’s about maybe 40% of the room. So you may remember that the G1 came with 192 megabytes of RAM. And in fact, most of that was used up by the system. There wasn’t a whole lot left over for apps.

Fast forward a few years later, we have the Motorola Xoom released just a couple of months ago. The Xoom comes with a gigabyte of RAM. Now some people might hear this and think, OK, my troubles are over. I never have to worry about memory management again. And of course, given that we have a whole room here, you guys are all smart people and you realize that that’s not true. And there are a couple of reasons for this. First of all, the Xoom has six and a half times the resolution that the G1 had. So you’ve got six and a half times as many pixels on screen. That means you’re probably going to need to use more memory. You got more bitmaps for example. The other thing is that on tablets, you really want to create a new kind of application. You know, the rich, immersive applications, like this

YouTube app, for example. These are going to take a lot of memory. There's tons of bitmaps in here. Those use up a lot of memory. Also, the Xoom we're talking about a pretty new device. This is basically bleeding edge hardware. Most your customers are not going to be using something that's only two months old. So of course, you want to support people who are using older hardware as well.

Finally, maybe you're all familiar with Parkinson's Law, which says that work always take as much time as you have. And really, it's kind of the same for software. So, no matter how much memory you have, you're going to find a way to use it and wish you had just a little bit more.

What I want to talk about today are basically two different things. First of all, I want to cover some of the changes that we've made in Gingerbread and Honeycomb that affect how your app uses memory. That's your cameo. All right, so as I was saying, there are two different things I want to cover today. So first of all, I want to talk about some of the changes we've made in Gingerbread and Honeycomb that affect how your apps use memory. And basically, memory management in general. In the second half of the talk I want to talk about some tools you can use to better understand how your app is using memory. And if you have memory leaks,how you can figure out where those memory leaks are.

So just to set expectations for this talk, I'm going to make them some assumptions about the stuff you've done and it'll really help you get the most out of this if you're familiar with these concepts. So I'm hoping that you've all written Android apps before. And it looked like about half the room had developed on the G1, so that's probably true. I hope that most of you have heard of the Dalvik heap. You have a basic idea of what I'm talking about when I talk about heap memory. I'm sure you're familiar with the garbage collector. You have a basic idea hopefully of what garbage collection is and how it works. And probably, most of you have seen an OutOfMememoryError before and you have a basic idea of why you get it and what you can do to deal with it.

So first, let's talk about heap size. So you may know that in Android, there's a hard limit on your application's heap size. And there's a couple reasons for this. So first of all, one of the great features of Android is that it has full multitasking. So you actually have multiple programs running at once. And so obviously, each one can't use all of your devices memory. We also don't want a runaway app to just start getting bigger and bigger and bloating the entire system. You always want your dialer to work, your launcher the work, that sort of thing. So there's this hard limit on heap size and if your application needs to allocate more memory and you've gone up to that heap size limit already,then you're basically going to get an out of memory error. So this heap size limit is device dependent. It's changed a lot over the years. On the G1 it was 16 megabytes. On the Xoom it's now 48 megabytes. So it's a little bit bigger.

If you're writing an app and you want to know, OK, how much heap space do I have available? You know, maybe you want to decide how much stuff to keep in a cache for example. There's a method you can use in ActivityManager, getMemoryClass that will return an integer value in megabytes, which is your heap size. Now these limits were designed assuming that you know almost any app that you would want to build on Android should be able to fit under these limits.

Of course, there are some apps that are really memory intensive. And as I said, on the tablet, we really want to build almost a new class of application. It's quite a different than the kind of things you were building on phones. So we thought, if

someone wants to build an image editor, for example, on the Xoom, they should be able to do that. But an image editor's a really memory intensive application. It's unlikely that you could build a good one that used less than 48 megabytes of heap. So in Honeycomb we've added a new option that allows applications like this to get a larger heap size. Basically, , you can put something in your AndroidManifest, largeHeap equals true. And that will allow your application to use more heap. And similarly, there's a method you can use to determine how much memory you have available to you. The ActivityManager getLargeMemoryClass method again, will return an integer value of this large heap size.

Now before we go any further, I want to give a big warning here. You know, this is not something you should be doing just because you got an out of memory error, or you think that your app deserves a bigger heap. You're not going to be doing yourself any favors because your app is going to perform more poorly because bigger heap means you're going to spend more time at garbage collection. Also, your users are probably going to notice because all their other apps are getting kicked out of memory. It's really something you want to reserve for when you really understand OK, I'm using tons of memory and I know exactly why I'm using that memory, and I really need to use that memory. That's the only time that you should be using this large heap option. So I mentioned garbage collection. And that it takes longer when you have a bigger heap. Let's talk a little bit about garbage collection. So I just want to go through a quick explanation here of what garbage collection is doing. So basically, you have a set of objects. First of all, let's say these blue objects here, these are the objects in your heap. And they form a kind of graph. They've got references to each other. Some of those objects are alive, some of them are not used anymore. So what the GC does is it starts from a set of objects that we call the roots. These are the objects that the GC knows is alive.

For example, variables that are alive on a thread stack, J and I global references, we treat objects in the zygote as heap, or as roots as well. So basically, the GC starts with those objects and starts visiting the other objects. And basically, traversing through the whole graph to find out which objects are referenced directly or indirectly from the GC roots. At the end of this process, you've got some objects left over, which the GC never visited. Those are your garbage. They can be collected. So it's a pretty simple concept. And you can see why when I said that you have bigger heaps you're going to have larger pause times. Because the garbage collector basically has to traverse your entire live set of objects. If you're using say the large heap option and you've got 256 megs of heap, well, that's a lot of memory for the garbage collector to walk over. You're going to see longer pause times with that.

We have some good news though. In Gingerbread, there have been some great changes to the garbage collector that make things a lot better. So in Gingerbre- sorry, pre-Gingerbread, the state of the garbage collector was that we had to stop the world collector. So what this means is that basically, when a garbage collection is in progress, your application is stopped. All your application threads are completely stopped while the collection is proceeding. This is a pretty standard things. These pauses generally tend to be pretty short. What we found was that pause times as heaps were getting bigger, these were getting to be a little bit too long. So we were seeing stuff up 50 to 100 milliseconds. And if you're trying to build a really responsive app that kind of pause time is not really acceptable.

So in Gingerbread, we now have a concurrent garbage collector. It does most of its

work concurrently, which means that your application is not stopped for the duration of the garbage collection. Basically, we have another thread that's running at the same time as your application that can perform garbage collection work. You'll see basically two short pauses. One at the beginning of a collection and one near the end. But these pause times are going to be much, much lower. Usually you'll see two, three, four, or five milliseconds for your pause time. So that's a significant improvement. Pause times about 10% of what they used to be. So that's a really good change that we have in Gingerbread.

Now if you're building memory heavy apps, there's a good chance you're using a lot of bitmaps. We found that in a lot of apps you have maybe 50 or 75% of your heap is taken up by bitmaps. And in Honeycomb because you're going to be developing on tablets, this gets even worse. Because your images are bigger to fill the screen. So before Honeycomb, the way we managed bitmaps was this. So the blue area up here is the Dalvik heap and this yellow object is a bitmap object. Now bitmap objects are always the same size in the heap no matter what their resolution is. The backing memory for the bitmap is actually stored in another object. So the pixel data is stored separately. Now before Honeycomb what we did was this pixel data was actually native memory. It was allocated using malloc outside the Dalvik heap. And this had a few consequences. If you wanted to free this memory you could either call recycle, which would free the memory synchronously. But if you didn't call recycle and you were waiting for your bitmap to get garbage collected,we had to rely on the finalizer to free the backing memory for the bitmap. And if you're familiar with finalization, you probably know that it's an inherently unreliable process. Just by its nature it takes several collections, usually for finalization to complete. So this can cause problems with bitmap heavy app as you had to wait for several garbage collections before your pixel data was reclaimed. And this could be a lot of memory because bitmap pixel data is quite a significant portion of the heap. This also made things harder to debug. If you were using standard memory analysis tools like the Eclipse Memory Analyzer, it couldn't actually see this native memory. You would see this tiny bitmap object. Sure, but that doesn't tell you very much. You don't mind if you have a 10 by 10 bitmap. But if you have a 512 by 512 bitmap it's a big difference. Finally, the other problem that we had with this approach was that it required full stop the world garbage collections in order to reclaim the backing memory, assuming that you didn't call recycle,that is.

The good news is in Honeycomb we've changed the way this works. And the bitmap pixel data is now allocated inside the Dalvik heap. So this means it can be freed synchronously by the GC on the same cycle that your bitmap gets collected. It's also easier to debug because you can see this backing memory in standard analysis tools like Eclipse Memory Analyzer. And I'm going to do a demo in a few minutes and you'll see really, how much more useful this is when you can see that memory. Finally, this strategy is more amenable to concurrent and partial garbage collections, which means we can generally keep those pause times down. So those are the two biggest changes that we've introduced in Gingerbread and Honeycomb that affect how your apps use memory.

And now I want to dive in to some tools that you can use to better understand how much memory your app's using. And if you have memory leaks, better understanding where those leaks are and generally, how your app is using memory. The most basic tool you can use for understanding your apps memory usage is to look at your log messages. So these are the log messages that you see in DDMS in the logcat view. You can also see them at the command line using adb logcat. And every time a

garbage collection happens in your process, you're going to see a message that looks something like this one. And I just want to go through the different parts of this message, so you can better understand what it's telling you.

The first thing we have is the reason for the garbage collection. Kind of what triggered it and what kind of collection is it. This one here was a concurrent collection. So a concurrent collection is triggered by basically, as your heap starts to fill up, we kick off our concurrent garbage collection so that it can hopefully complete before your heap gets full.

Other kinds of collections that you'll see in the log messages. GC for malloc is one of them. That's what happens when say, we didn't complete the concurrent collection in time and your application had to allocate more memory. The heap was full, so we had to stop and do a garbage collection.

You'll see GC external alloc, which is for externally allocated memory, like the bitmap pixel data which I mentioned. It's also used for NIO direct byte buffers. Now this external memory as I mentioned, has gone away in Honeycomb. Basically everything is allocated inside the Dalvik heap now. So you won't see this in your log messages in Honeycomb and later.

You'll also see a message if you do an HPROF, if you create an HPROF profile. And finally, the last one I want to mention is GC explicit. You'll see this generally when you're calling system.gc, which is something that you know you really should avoid doing. In general, you should trust in the garbage collector. We've got some information also about the amount of memory that was freed on this collection. There's some statistics about the heap. So the heap in this case, was 65% free after the collection completed. There's about three and a half megs of live objects and the total heap size here is listed as well. It's almost 10 megs, 9,991 K. There's some information about externally allocated memory, which is the bitmap pixel data and also, NIO direct byte buffers. The two numbers here, the first number is the amount of external memory that your app has allocated. The second number is a sort of soft limit. When you've allocated that much memory, we're going to kick off a GC. Finally, you'll see the pause times for that collection. And this is where you're going to see the effect of your heap size. Larger heaps are going to have larger pause times.

The good news is for a concurrent collection, you're going to see these pause times generally pretty low. Concurrent collections are going to show two pause times. There's one short pause at the beginning of the collection and one most of the way through. Non-concurrent collections you'll see a single pause time, and this is generally going to be quite a bit higher. So looking at your log messages is a really basic way to understand how much memory your app is using. But it doesn't really tell you, where am I using that memory? What objects are using this memory?

And the best way to do that is using heap dumps. So a heap dump is basically a binary file that contains information about all of the objects in your heap. You can create a heap dump using DDMS by clicking on the icon, this somewhat cryptic icon. I think that mentioned it in the previous talk. There's also an API for creating heap dumps. In general, I find using DDMS is fine. There are times when you want to create a heap dump at a very, very specific point in time. Maybe when you're trying to track down a memory leak. So it can be helpful to use that API. You may need to convert the heap dump to the standard HPROF format. You'll only need to do that if you're using the standalone version of DDMS. If you're using

the Eclipse plug-in, the ADT plug-in, it will automatically convert it. But the conversion is pretty simple. There's a tool in the Android SDK, which you can use to do it. And after you've converted it to the standard HPROF format, you can analyze it with standard heap analysis tools, like MAT or jhat.

And I'm going to show an example of MAT, which is the shorter way of saying the Eclipse Memory Analyzer. And before I jump into the demo, I want to talk about memory leaks. So there's kind of a misconception that in a managed run time, you can't have memory leaks. And I'm sure you guys know that's not true. Having a garbage collector does not prevent memory leaks. A memory leak in a managed runtime is a little bit different though, than a memory leak in C or C++. Basically, a leak is when you have a reference to an unused object that's preventing that object from being garbage collected. And sometimes you can have a reference to a single object, but that object points to a bunch of other objects. And basically, that single reference is preventing a large group of objects from being collected.

One thing to watch out for in Android. I see people sometimes and I've done this myself, accidentally create a memory leak by holding a long lived reference to an activity. So you need to be really careful with that and maybe it's you're holding a reference to the context and that's what happens. You can also do it by keeping a long lived reference to a view or to a drawable, because these will also hold a reference to the activity that they were originally in. And the reason that this is a problem, the reason this causes a memory leak is this. So you've got your activity, it contains a viewgroup, a linearlayout or something, and it contains some views. And we've got a reference from the framework to the currently visible activity.

But in Android, when you have a rotation event, so you rotate your device, what we do is actually build up a new view hierarchy because you need to load new resources, you may have a brand new layout for landscape or portrait, you may have differently sized icons or bitmaps. And then we basically remove the reference to the old view hierarchy and point to the new one. And the idea is that this old view hierarchy sure get garbage collected. But if you're holding a reference to that, you're going to prevent it from getting garbage collected. And that's why it's a problem to hold the long lived reference to an activity or even to a view because in fact, the arrows connecting these objects should be going in both directions. Because you've got pointers all the way up. So if you do have a memory leak, a really good way to figure out where it is using the Eclipse Memory Analyzer.

I'm going to do a demo of that, but I want to first cover some of the concepts behind the Memory Analyzer, so that when I do the demo you'll better understand what I'm showing you. So the Eclipse Memory Analyzer can be downloaded from the eclipse.org site. It comes in a couple of flavors. There's an Eclipse plug-in version, there's also a standalone version. I'm going to be demonstrating the standalone version. I just personally prefer not to have Eclipse have all these different plug-ins. I kind of like to keep things a little bit separate. But they're basically the same. Now, Memory Analyzer has some important concepts that you'll see a lot.

It talks about shallow heap and retained heap. So the shallow heap of an object is just how large is this object, it's size and bytes. It's really simple. So let's say that all of these objects are 100 bytes. So they're shallow heap is 100 bytes. It's easy. The retained heap is something different. Basically, the

retained heap says, if I have an object here and I were to free this object, what other objects is it pointing to? And could those be freed at the same time? And so you calculate the retained heap in terms of, what is the total size of objects that could be freed by freeing this one object? So maybe it's best to understand with an example. So this object down on the right-hand side in yellow, this guy doesn't point to any other objects. So his retained size is pretty easy to calculate. His retained heap is 100. This guy on top, he has a pointer to one other object. But he's not holding that object alive. There are other pointers to that same object. So this guy's retained heap is also just 100 bytes. Because if we were to remove this object, it's not going to free up any other objects. The object down at the end however, it's basically keeping all the other objects alive. So its retained heap is 400 because if we could free that object, we could free all the other objects well, on this slide anyway. So you might be wondering, how do you go about calculating the retain heap?

So you're going to see this in Memory Analyzer. And actually, knowing how it calculates the retained heap is quite useful. So the Memory Analyzer uses a concept called the denominator tree. This is a concept from graph theory. Basically, if you have a node A and a node B, A is said to be the dominator of B if every path to B goes through A. And so you might see how that could help us figure out what the retained heap of an object is. So another example here. So let's start with A. It's kind of the root. B and C are only accessible through A. So it's pretty straightforward. They're children of A and the dominator tree. E is also only accessible through C. So it's a child of C in the dominator tree. D is a little bit interesting here. D can be accessed through B or C, but A is on every path to D. So that means that A is the parent of D and the dominator tree. And now you're going to see this dominator tree concept also pop up in Memory Analyzer in its UI. And it can be really helpful for tracking down memory leaks.

So let's jump in and do a demo of debugging and memory leak with MAT. So what I'm going to use for this demo is the Honeycomb gallery's sample application. It's a simple application that comes with the Android SDK the basically just demonstrates some of the features of Honeycomb. And really, all it is is a little app the lets you page through some photos. Pretty simple. Now I've done something kind of naughty here. I've introduced a memory leak into this application. And I'll show you how I've done that. Sorry, I better switch to the slides again.

So you'll see here I have the source code, an excerpt of the source code from the activity. And so what I've done here is I've introduced this inner class called leaky. And this is not a static inner class. So you may know that if you create a non-static inner class, it actually keeps a reference to the enclosing object. And this is because from a non-static inner class, you can actually refer to the instance variables of the enclosing object. So it's going to retain a reference to the activity here. That's fine as long as this object doesn't live longer than the activity. But I've got this static field and statics live longer than any particular instance. And in my on create method, what I've done is instantiated the leaky object and stored it into the static field. So if you want to be able to visualize this, I basically got my view hierarchy that starts with the main activity. I've instantiated this leaky object and he has a reference to the main activity because that was its enclosing class. Finally, I have the main activity class, which is conceptually a different area of memory than any particular instance. And there's a static variable pointing to the leaky object. So maybe you can see how this is going to cause a memory leak when I rotate the screen. So let's jump in and take a look at this memory leak. So if you want to

figure out whether you have a memory leak, one of the easiest ways is to just kind of look at your log messages. So I'm just going to do that. I'm going to do it at the command line. I can just type logcat. And I want to restrict it to the particular process that I've got running here. I don't want to see all of the log messages on the system. So I'm just going to grab on the process ID. There we see a bunch a log messages, including some garbage collection messages. And the number you want to look at is basically the first number here in the 9805K. The first number in your heap size. This is the amount of live objects in the system. And if you're looking for a memory leak, that's what you want to look at. So I'm going to flip through some of the photos here. And you'll see that that number stays pretty constant. We're up to 9872. But basically, the heap usage is pretty constant. Now when I rotate this device, we're going to be a bunch of garbage collections happen. That heap usage goes up and it doesn't go down again. So we're now up to 12 megs of heap. So we leaked about two and a half megs. So whenever you see your memory go up in kind of a step function like that, it steps up and just never goes back down, that's a good sign you have a memory leak.

So once you know that you have a leak, what you'll want to do is create a heap dump, so you can go about debugging it. So I'm going to do that right now. I'll open up DDMS. You just need to select the process that you care about and click on this icon up in the toolbar that says dump HPROF file. That'll create a heap dump. It takes a few seconds because it's dumping basically a huge binary file out to disk. And then I can just save it in a file called dump.hprof. And then, because I'm using this standalone version of DDMS here, I need to convert this file. As I mentioned, if you're using the ADT plug-in for Eclipse and using DDMS in there, you don't need to go through this conversion step. But it's really simple. Now that I've converted it, I can open up the Eclipse Memory Analyzer and take a look at this heap dump. So there's not much to see in the Memory Analyzer until you've opened up a heap dump, which we can do just from the file menu. Open heap dump. And I'll open up this converted heap dump, which I just created. Doesn't take very long for it to load up.

And the first thing you'll see is this pie chart. This is showing the biggest objects in the system by retained size. Now this alone doesn't really tell us too much. You can see that down in the bottom left here, when I mouse over the various slices of the pie, it's telling me what kind of object I've got. But that doesn't really tell us too much. If we want to get some more information, you want to look down here. There are two views. The histogram view and the dominator tree. And these are the ones that I find most useful and I'm going to show to you. Let's take a look at the dominator tree. You remember the concept I explained. This is how it can be useful in tracking down a memory leak. So what we've got here is basically a list of instances or a list of objects in this system organized. There's a column here. Organized by the amount of retained heap. So when you've got a memory leak, looking at the amount of retained heap is often a good way to look at things because that's going to have the biggest effect on how much memory you're using. And chances are, if you've noticed that you've got a leak, you're leaking a significant amount. So let me just Xoom in here. Hopefully you guys can see this a bit better. So at the very top of the list we have the resources class. That's not too surprising because our resources we have to load lots of bitmaps. That's going to hold lots of memory alive. That's fine. These two bitmap objects are interesting. I've got these two large bitmaps, more than two and a half megs each. It's funny because that sounds about like the amount of memory that I was leaking. So if I want to investigate a bit further, I can right click on one of these objects and choose path to GC roots. And I'll

chose excluding weak references because I want to see what's keeping that object alive. And a weak reference is not going to keep it alive. So this opened up a new tab and what do you know? It actually points right to my leak. So when you're creating leaks in your application, make sure you name it something really helpful like this so you can find it easily.

AUDIENCE: [LAUGHTER]

PATRICK DUBROY: So some of you might have noticed this, that if there's only a single path to this object, because that's all I can see here, why didn't this leak object show up in the dominator tree? I mentioned that the dominator tree should show you the largest objects by their amount of retained heap. And well this is a single object that's responsible for retaining the bitmap. So the reason for that is that the Eclipse Memory Analyzer, when it calculates the dominator tree, it actually doesn't treat weak references separately. It basically just treats them like a normal reference. So you'll see that if I actually right click on this guy again and say path to GC roots, and say with all references, then there's actually another path to this object. But it's a weak reference. Generally you don't need to be too concerned about weak references because they're not going to prevent your object from being garbage collected. But that's why the leak object didn't show up in the dominator tree. So the dominator tree is one really, really useful way of tracking down a memory leak. Another thing I like to use is the histogram view. So I mentioned that in Android, it's common to leak memory by keeping long lived references to an activity. So you may want to actually go and look at the number instances of your main activity class that you have.

And the histogram view lets you do that. So the histogram view just shows a list of all the classes in its system and right now it's sorted based on the amount of shallow heap occupied by classes in the system. So at the very top there, we see we have byte arrays. And the reason for this is that byte arrays are now the backing memory for pixel data. And you know, this is a perfect example of why it's really useful that we now have the pixel data inside the heap. Because if you're using this on Gingerbread or earlier, you're not going to see byte arrays at the top. Because that memory with allocated in native memory. So we could also, if we were concerned about these byte array objects, we might want to right click on it and say list objects with incoming references. And we've got our two large byte array objects here. We can right click on one and say, path to GC roots, excluding weak references. So this guy looks to have several paths, which keep it alive. Nothing looks out of the ordinary to me. And when you're trying to find a memory leak, there's not really a magic answer for how you find a leak. You really have to understand your system and understand what objects are alive, why they're alive, during the various parts of your application. But you'll see if I look at this other byte array object, and again, do path to GC roots excluding weak references, well, I've found my leak again. So this was another way that I might have found this if it weren't so obvious from the dominator tree.

The histogram view can also help us look for our activity instances. So there's a lot of classes obviously in the system. Our activity is not here. There's 2,200 classes. But luckily, Eclipse Memory Analyzer has this handy little filter view at the top. You can just start typing a regular expression. And it'll return you all the classes that match that. So here we've got our main activity. And it tells us that there are actually two instances of this main activity. And that should kind of be a red flag. Normally you should expect to see only a single instance of your

main activity alive.Now I mentioned during the screen rotation, we build up a new view hierarchy, there's going to be a brief time where there's two instances alive. But for the most part, you should expect to see one here. So I might think, OK, this is a red flag.

Let's take a look. So I can right click on this object and list objects with incoming references. So I want to look at what instances do I have and what's pointing to them? And so I've got two instances here. If I right click on one of them and choose path to GC roots, excluding weak references, I've again, found my memory leak. And in looking at this, I might realize that, oh, I really didn't intend to do this. I didn't mean to keep this reference there. So that's another way that you could have found the leak. So now that we've discovered where our memory leak is, why don't we actually go ahead and fix it. So in this case, the problem was that we had a non-static inner class. So we could fix this by making it a static inner class. And then it wouldn't actually keep a reference to the enclosing activity. The other thing we could do is actually just not store it in a static variable. So it's fine if this leaky object has a reference to the activity, as long as it doesn't live longer than the activity. So let's do that. Let's just make this a regular instance variable and not a static. So then I can go in here recompile this and push it to the device. And hopefully, we should see that our memory leak has been eliminated. Sorry, what we actually want to do is look at our log output in order to see how much memory we're using. So I'm just going to fire up the process here, take a look at the process ID. And again, just do adb logcat just on that process. So as I page through the photos again, we see lots of GC messages. When I rotate, we're going to see the memory usage goes up for a minute there. But after a few collections, it does go back down to its previous value. So we've successfully eliminated the leak there. And this is great. You always want to eliminate memory leaks.

So that's an example of using the Eclipse Memory Analyzer to debug a memory leak. Eclipse Memory Analyzer is a really powerful tool. It's a little bit complex. It actually took me quite a while to figure out that these were the two best tools for the job. So you really want to watch out for these memory leaks. So I gave an example here of retaining a long lived reference to an activity. If you've got our context, a view, a drawable, all of these things you need to watch out for. Don't hold long lived references to those. It can also happen with non-static inner classes, which is what I demonstrated there as well. Runnable is actually one that can bite you sometimes. You know, you create a new runnable. You have a deferred event that's going to run in like five minutes. If user rotates the screen that deferred runnable is going to hold your previous activity instance alive for five minutes. So that's not good.

You also want to watch out for caches. Sometimes you have a cache and you want to keep memory alive, so that you can load images faster let's say. But you may inadvertently hold things alive too long. So that covers basically, the core parts of the Eclipse Memory Analyzer, and gives you a basic understanding of memory leaks. If you'd like to get more information about Memory Analyzer, the download link you can find on the eclipse.org/mat site. Markus Kohler who's one of the original team members of Eclipse Memory Analyzer, he has a blog called the Java Performance Blog. This is really great. He's got tons of great articles on there about MAT and different ways you can use it to understand your applications memory usage.

I've also got an article that I wrote on the Android Developer Blog called memory

analysis for Android applications. It covers a lot of the same stuff that I did in my demo here. And Romain Guy also has a good article on avoiding memory leaks in Android. So I hope that's been helpful, I hope you guys have a better understanding now of how you can figure out your apps memory usage.

And I've talked about two of the biggest changes that we've made in Gingerbread and Honeycomb that affect how your apps use memory. Thanks.

[APPLAUSE]

So I can take questions from the floor if anyone has any. Or you all want to get out and get to a pub and have a beer?

AUDIENCE: Hi, you mentioned that if you use NIO in Honeycomb your objects are going to be not in native memory and now they're going to be managed memory. How does that affect performance if you're doing a NIO, is that going to be any slower, like very intense on network?

PATRICK DUBROY: No, I mean it shouldn't affect. So I should say that there is still a way to allocate native memory for your NIO byte buffers. I'm not that familiar with the NIO APIs, but I believe there's a way in JNI you can allocate your own memory. So in that case, you'll still be using native memory. But either way, it's just memory. It's just allocated in a different place. So there's nothing that makes the Dalvik heap memory slower than other memory.

AUDIENCE: So you're saying how in Honeycomb the bitmaps are stored in the Dalvik heap, but in previous versions to that it was stored on native memory. Does that mean that bitmaps had a different amount of heap size? Or is that still all counted in the 16 or 24 megabytes that previous versions had?

PATRICK DUBROY: Yeah, good question. The accounting limits are still the same. That was accounted for previously. You might have noticed if you ever ran into your heap limit, you would be looking at your heap size and like, I haven't hit the limit yet, why am I'm getting out of memory? That was actually accounted for, so it was your total heap size plus the amount of externally allocated memory that was your limit. So that hasn't changed.

AUDIENCE: Hello. I have a question on when does the garbage collector kicks in. Is is when a number of objects in memory or the size of the heap?

PATRICK DUBROY: Well, it depends on what kind of garbage collection you're talking about. The concurrent garbage collector -

AUDIENCE: Yeah, the concurrent. Yes.

PATRICK DUBROY: Yeah, so that I believe is the amount of basically, how full your heap is getting.

AUDIENCE: Because I noticed that when you do a lot of [INAUDIBLE] provide operations, so you have like [INAUDIBLE] list of operations, the garbage collector kicks in. But actually don't collect any objects because you're just filling in the array of objects that you want to insert into a database. And that's grow quite quickly. And that tends to slow down a bit, the application without actually solving any heap size.

PATRICK DUBROY: Yeah, I’m not sure if the GC looks at- so you’re basically saying, I guess, that the collector is kicking in. It’s not actually able to collect anything, so it shouldn’t-

AUDIENCE: But it keeps trying.

PATRICK DUBROY: Yeah, it should be smart enough. Yeah, I don’t believe we actually look at those kind of statistics yet. But I mean it seems reasonable. Yeah.

AUDIENCE: I was wondering if you guys have some plans for making a profiler for applications or more tools for analyzing memory and all that stuff?

PATRICK DUBROY: No plans that I know of. Is there anything in particular that you need? I mean I think the Eclipse Memory Analyzer is a really powerful tool and I use it in my day-to-day work quite a bit. So I’ve certainly never found that it it was missing certain features that I needed.

AUDIENCE: Yeah, probably because there are some old versions from Android that show memory leaks or something. But for example, on Eclair, there were some stuff with the- something there.

PATRICK DUBROY: Yeah, I mean we don’t have any immediate plans I don’t think to running specific tools.

AUDIENCE: OK, thank you.

PATRICK DUBROY: Oh, sorry I’ve been- yeah.

AUDIENCE: To my understanding, the native part of a bitmap memory before was actually an instance of the SKIA library, of one of the SKIA library bitmap classes. So is this still there or is it gone now that there is no more native memory allocated?

PATRICK DUBROY: No, SKIA is still part of this stack there. Basically at the point where SKIA calls out to allocate memory, we actually just call back into the VM and allocate the memory there rather than calling malloc. So it’s still basically the same mechanism, but the memory’s just coming from a different place.

AUDIENCE: OK. AUDIENCE: I thought that when I was using my application, I checked the heap size. While using the application the heap size was not significantly going up. But the amount of memory used by the application, which is listed in the applications tab under the running applications is going up significantly. Sometimes even doubling. I know that this is a different heap that is shown there. It’s actually the process heap, right? Can you tell me what the background of that is that this is shown there because might like- I don’t have a memory leak and users complain about my application leaking memory. Because for the user it looks like it’s leaking memory.

PATRICK DUBROY: Right. Because you’re saying there’s stuff that’s attributed to your process that are showing up in the- basically, in system memory?

AUDIENCE: Yeah. So it’s showing the system memory in the applications tab, which is not really linked to my heap memory. So that is going up, but I can only control the heap memory. If I don’t have a native application I cannot control

everything else.

PATRICK DUBROY: I mean there are going to be various things in the system that are going to get larger. For example, like your JIT code caches. As the JIT kicks in and is allocating memory, like it needs to store the compiled code somewhere. So there's definitely other parts of this system that allocate memory that's going to kind of get charged to your application. But I can't think of why. I can't think of anything that would be out of the ordinary really that should cause problems.

AUDIENCE: But do you know if this will be changed maybe in the future? That this number is not shown there because for me, it doesn't make sense to show this number to the end user because he doesn't understand what it means.

PATRICK DUBROY: I see. Where is he seeing the number?

AUDIENCE: In the running applications tab. If he goes to settings, running applications, he can see the memory usage per application and that's actually the system memory.

PATRICK DUBROY: I see. Yeah, I'm not sure what our plans are with that. Sorry. I can take a look and I'm not actually sure where it's getting that number from.

AUDIENCE: OK, thanks.

AUDIENCE: My question's about reasonable expectations of out of memory errors. Is it possible to completely eliminate them? We've been working for a while in getting rid of all the out of memory errors and down to one in about every 17,000 sessions. Should we keep troubleshooting. I mean, I'd like to get it down to zero, but is that reasonable or?

PATRICK DUBROY: So there are certain scenarios where if you're really close to your memory limit, so if your applications live memory size is really close to that limit, the garbage collector's fundamentally kind of asynchronous. So if you're really close to the limit, there can be times where you're just trying to allocate so fast that the garbage collector can't keep up. So you can be actually sort of out running the garbage collector. So certainly it's possible to build applications that never see an out of memory error. But on the other hand, there are certain types of applications that are going to be running really, really close to the limits. One thing you can use if you have caches or things that you can free up, there are several ways to figure out that you're getting close to the heap memory limit. I believe there's a callback you can get notification that we're getting low on memory. Although, the name escapes me. But you can also look at that, the Activity Manager, get memory class to get a sense of how much memory you have available on the system. And you know, maybe you can keep like smaller caches or leave the initialize objects rather than initializing them all in the constructor or something like that. It really depends on the application whether you expect to be running close to that heap limit or not.

AUDIENCE: You recommended not to call system.gc manually if you can help it. Is there any way to reliably free bitmap memory pre-Honeycomb?

PATRICK DUBROY: Yes. Pre-Honeycomb?

AUDIENCE: Yes.

PATRICK DUBROY: You can call recycle on the bitmap.

AUDIENCE: Yeah, but it can still take several passes apparently.

PATRICK DUBROY: No. If you call recycle that will immediately free the backing memory. The bitmap itself, that's like 80 bytes or something.

AUDIENCE: There are also bitmaps like drawables that you can't manually recycle the bitmaps that the drawable object creates.

PATRICK DUBROY: OK.

AUDIENCE: The backing bitmaps for those.

PATRICK DUBROY: I see. No, I mean there are still some cases I guess where system.gc is the right approach.

[UNINTELLIGIBLE PHRASE]

PATRICK DUBROY: OK, which objects are you talking about in -

AUDIENCE: My experience is when I have image drawables that are used some where in my layout and I know they're no longer needed. Some of them are fairly large and it seems like -

PATRICK DUBROY: You can call recycle on those I believe.

AUDIENCE: OK. My experience is that it will cause other problems when I do that.

PATRICK DUBROY: If you're still using them, then you can't - I mean, you can only recycle that when you're not using it.

AUDIENCE: Sure. OK.

AUDIENCE: For native code that uses a lot of mallocs, what's the best way to manage that memory?

PATRICK DUBROY: That's a very good question. When you've got native code, I mean mostly what I was covering here was managing memory from the Dalvik side of things. I don't know that I have any real pointers. I mean that's one of the reasons why programming in a managed runtime is very nice. Is that you don't have to deal with manually managing your memory. I don't have any great advice for that.

AUDIENCE: Does the app that calls into the native libraries, is it aware of, at least on an aggravate level, how much memory is being used or is it completely a separate -

PATRICK DUBROY: I don't believe there's any way to account for if you're calling into the library and it's calling malloc. I don't know that there's any way to account for that memory from your application side.

AUDIENCE: But that garbage collector will run when you start allocating memory, will it not?

PATRICK DUBROY: It'll run when you start allocating like objects in Dalvik. It

doesn’t have any knowledge of calls to malloc.

AUDIENCE: You’ll just get an out of memory or a failed malloc if you-

PATRICK DUBROY: Yeah. Sure. It’s going to be the same mechanisms as any C or C++ program. Malloc is going to return a null pointer. Yes?

AUDIENCE: [UNINTELLIGIBLE PHRASE]

PATRICK DUBROY: Pardon me?

AUDIENCE: [UNINTELLIGIBLE PHRASE] PATRICK DUBROY: Oh, OK. That’s news to me. Malloc can’t fail on Android.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

PATRICK DUBROY: I see. OK.

AUDIENCE: Can you repeat that?

PATRICK DUBROY: Romain tells me that malloc can’t fail on Android.

AUDIENCE: [UNINTELLIGIBLE PHRASE]

PATRICK DUBROY: I see. So I think this is the old Linux lazy- yeah. It’ll successfully allocate the virtual memory, but Linux can actually hand out more virtual memory than it can actually commit. So you can get problems. Like when your system is totally, totally out of native memory, you’re going to see crashes.

AUDIENCE: So native memory is completely separate from anything Dalvik?

PATRICK DUBROY: Yes. Well, I mean, sorry, I should say, like Dalvik is still allocating its own memory like for the heap through the native mechanisms. So it’s reserving the same virtual memory pages that other applications are using.

AUDIENCE: But if your system memory is-

PATRICK DUBROY: Yeah, if your system memory is out, you’re in trouble.

AUDIENCE: But Dalvik won’t get a notice say, hey, better start garbage collecting?

PATRICK DUBROY: Well, no.

AUDIENCE: The flag for using larger heap, does that require a permission, like users permission or something like that?

PATRICK DUBROY: I can’t remember whether we added that or not. I don’t think that it does.

AUDIENCE: Like the whole- could it been like a permission thing? But if it’s not then-

PATRICK DUBROY: Yeah, I mean the idea I think is that- yeah, you’re right. I mean it can affect the system as a whole because you’re going to have apps that

are using a lot more memory, which is why I gave that big warning, that this is not something that you should be using unless you know that you really need it.

AUDIENCE: Yeah. But [INAUDIBLE]. OK.

PATRICK DUBROY: I don’t think there’s a permission for it, though.

AUDIENCE: What if the app kind of runs in the background for weeks at a time? So I do everything I can to simulate a leak, click everywhere I can, but I see the leaks if the app runs two or three days and then I get [INAUDIBLE].

PATRICK DUBROY: One thing you could try is if you can use the APIs to determine how much free memory you have. I don’t know if there’s any way you can actually kind of notice in your application that it started leaking. But you could write out an HPROF file when you notice that you’ve gotten to a certain point, your heap is getting smaller and smaller. So there is some debug information there that you could use. So if you have like some beta testers, who could actually send you these dumps, then you could do that. So write out the HPROF file to SD card or something.

AUDIENCE: So maybe I can just write an HPROF file every -

PATRICK DUBROY: I wouldn’t do that. I mean they’re quite large. You don’t want to be doing that on a regular basis. But if you detect that things have gone really, really wrong and you’re about to die, in an alpha version or something for testing that’s one way you could do it. But I definitely wouldn’t recommend putting an app in the market that’s dumping like very large files to the SD card for no reason.

AUDIENCE: OK.

PATRICK DUBROY: OK, Thanks a lot.

下载

http://pan.baidu.com/s/1i3zLPjf

- 程序员如何着装才能提升逼格
- 那些让程序员爱不释手的马克杯
- 程序员的数学【一本为程序员朋友们写的数学书】

Posted by androidyue Nov 2nd, 2014 Android

« Note for Google IO Memory Management For Android

评论 最新 最早 最热

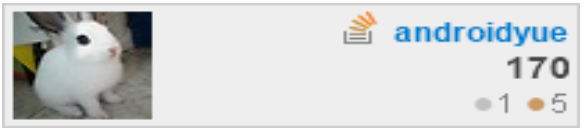
还没有评论，沙发等你来抢

社交帐号登录: 微博 QQ 人人 豆瓣 更多»

不留下点什么吗

编译--执行

技术小黑屋正在使用多说



Recent Posts

- Google IO：Android内存管理主题演讲记录
- Note for Google IO Memory Management for Android
- 效率脚本：删除已经合并的git分支
- 人生苦短，让你的Git飞起来吧
- 译文：理解Java中的弱引用

Google+

Blogrolls

- 友情链接
- 捐赠 Donate!
- 超性价比服务器
- 国内最好的免费CDN图床

Copyright © 2014 - androidyue - Powered by Octopress