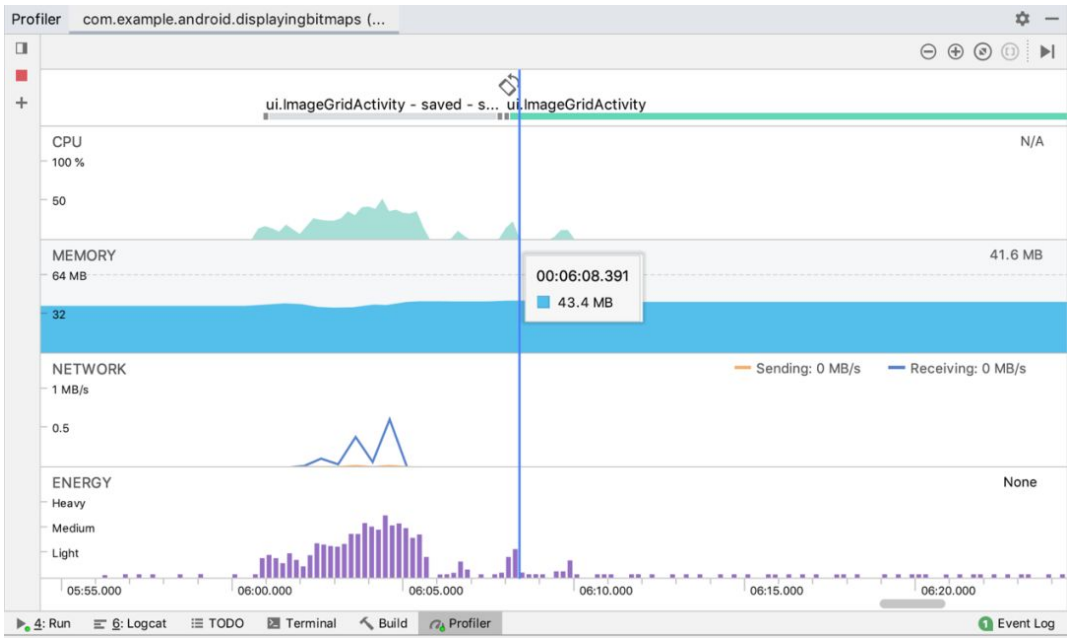
使用 Android Studio Profiler 工具解析应用的内存和 CPU 使用数据

谷歌开发者
已认证的官方帐号
已关注

19 人赞同了该文章

为了帮助开发者开发出更加轻快高效的应用，我们在 Android Studio 3.0 以及更高版本中加入了 Android Profiler 工具，用于应用的 CPU、内存、网络和能耗分析。



在 Android Profiler 提供的这四种性能数据中，绝大多数场景下我们都更关心 CPU 和内存的使用情况。本文将介绍对应的两种分析工具 —— Memory Profiler 和 CPU Profiler。

Memory Profiler

许多开发者使用 Memory Profiler，是希望发现和定位内存泄漏问题。在介绍 Memory Profile 如何解决这一问题之前，我想先明确 "内存泄漏" 这一概念。无论您当前是否了解内存泄漏，都将帮助我更好地解释 Memory Profile 的工作原理。

内存泄漏

什么是内存泄漏?

通常我们认为，在运行的程序中，如果一个无法访问的对象却仍然占用着内存空间，即为此对象造成了内存泄漏。如果您使用过 C 语言或 C++ 的指针，您会很熟悉这个概念。

但是在 Kotlin 和 Java 的世界中，事情有些许不同。因为这两种语言是运行在 Java 虚拟机 (JVM) 中的。在 JVM 中，有个重要的概念，就是垃圾回收 (GC)。当垃圾回收运行时，虚拟机会首先识别 GC Root。GC Root 是一个可以从堆外部访问的对象，它可以是本地变量或运行中的线程等。虚拟机会识别所有可以从 GC Root 访问的对象，它们将会被保留。而其他无法从 GC root 访问的对象，则会被认为是垃圾并回收掉。

所以，一般意义上的内存泄漏在 JVM 中并不存在。在 JVM 中的内存泄漏通常是指: 内存中含有那些再也不会被使用、但是仍然能够访问的对象。

Activity 和 Fragment 泄漏检测

在 Android 应用中，应当尤为警惕 Activity 和 Fragment 对象的泄漏，因为这两种对象通常都会占用很多内存。在 Android 3.6 中，Memory Profiler 加入了自动检查 Activity 和 Fragment 中的内存泄漏的功能。使用这一功能非常的简单:

- 首先，您需要在 Memory Profiler 中保存 Heap Dump，点击下图所示按钮

- 在 Heap Dump 加载完成后，勾选 "Activity/Fragment Leaks" 选框:

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...

此时如果有检查到 Activity 或 Fragment 的泄漏，就会在界面中显示出来。

Memory Profiler 通过以下几种场景来判断泄漏是否发生:

- 当我们销毁了一个 Activity 的实例后，这个实例就再也不会被使用了。此时如果仍然有这个 Activity 的引用，Memory Profiler 就会认为它已经泄漏；
- Fragment 的实例应当与一个 Fragment Manager 相关联，如果我们看到一个 Fragment 没有关联任何一个 Fragment Manager，而且它依然被引用时，也可以认为有泄漏发生。

不过要注意的是，针对 Fragment 有个特别的情况: 如果您载入的 Heap Dump 的时机，刚好介于 Fragment 被创建和被使用的时间之间，就会造成 Memory Profiler 误报；相同情况也会发生在 Fragment 被缓存但是没有被复用的时候。

其他内存泄漏检测

Memory Profiler 也可以用于检查其他类型的泄漏，它提供了许多信息，用于帮助您识别内存泄漏是否发生。

当您拿到一段 Heap Dump 之后，Memory Profiler 会展示出类的列表。对于每个类，"Allocation" 这一列显示的是它的实例数量。而在它右边则依次是 "Native Size"、"Shallow Size" 和 "Retained Size":

这几组数据分别意味着什么呢？下面我会通过一个例子来说明。

我们用下图来表示某段 Heap Dump 记录的应用内存状态。注意红色的节点，在这个示例中，这个节点所代表的对象从我们的工程中引用了 Native 对象:

这种情况不太常见，但在 Android 8.0 之后，使用 Bitmap 便可能产生此类情景，因为 Bitmap 会把像素信息存储在原生内存中来减少 JVM 的内存压力。

先从 "Shallow Size" 讲起，这列数据其实非常简单，就是对象本身消耗的内存大小，在上图中，即为红色节点自身所占内存。

而 "Native Size" 同样也很简单，它是类对象所引用的 Native 对象 (蓝色节点) 所消耗的内存大小:

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...

"Retained Size" 稍复杂些，它是下图中所有橙色节点的大小:

由于一旦删除红色节点，其余的橙色节点都将无法被访问，这时候它们就会被 GC 回收掉。从这个角度上讲，它们是被红色节点所持有的，因此被命名为 "Retained Size"。

还有一个前面没有提到的数据维度。当您点击某个类名，界面中会显示这个类实例列表，这里有一列新数据 —— "Depth":

"Depth" 是从 GC Root 到达这个实例的最短路径，图中的这些数字就是每个对象的深度 (Depth):

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...

一个对象离 GC Root 越近，它就越有可能与 GC Root 有多条路径相连，也就越可能在垃圾回收中被保存下来。

以红色节点为例，如果从其左边来的任何一个引用被破坏，红色节点就会变成不可访问的状态并且被垃圾回收回收掉。而对于右边的蓝色节点来说，如果您希望它被垃圾回收，那您需要把左右两边的路径都破坏才行。

值得警惕的是，如果您看到某个实例的 "Depth" 为 1 的话，这意味着它直接被 GC root 引用，同时也意味着它永远不会被自动回收。

下面是一个示例 Activity，它实现了 LocationListener 接口，高亮部分代码 "requestLocationUpdates" 将会使用当前 Activity 实例来注册 locationManager。如果您忘记注销，这个 Activity 就会泄漏。它将永远都待在内存里，因为位置管理器是一个 GC root，而且永远都存在:

您能在 Memory Profiler 中查看这一情况。点击一个实例，Memory Profiler 将会打开一个面板来显示谁正在引用这个实例:

我们可以看到位置管理器中的 mListener 正在引用这个 Activity。您可以更进一步，通过引用面板导航至堆的引用视图，它可以让您验证这条引用链是否是您所预期的，也能帮您理解代码中是否有泄漏以及哪里有泄漏

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···

CPU Profiler

和 Memory Profiler 类似，CPU Profiler 提供了从另一个角度记录和分析应用关键性能数据的方法。

使用 CPU Profiler，首先要产生一些 CPU 的使用记录:

- 进入 Android Studio 中的 CPU Profiler 界面，在您的应用已经部署的前提下，点击 "Record" 按钮；
- 在应用中进行您想要分析的操作；
- 返回 CPU Profiler，点击 "Stop" 按钮。

由于最终呈现的数据是基于线程组织的，所以去观察数据之前，您应该确认是否选择了正确的线程:

我们这里所获得的 CPU 使用记录信息，其实是一个 System Trace 实例的调用栈集合 (下文统称 "调用栈")。而就算是很短的 CPU 使用记录，也会包含巨量的信息，同时这些信息也是人无法读懂的。所以 CPU Profiler 提供了一些工具来可视化这些数据。

Call Chart

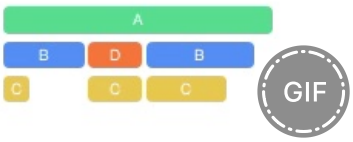
在 CPU Profiler 界面下半部，有四个标签页，分别对应四个不同的数据图表，它们分别是: Call Chart、Flame Chart、Top Down 和 Bottom Up。其中的 Call Chart 可能是最直白的一个，它基本上就是一个调用栈的重新组织和可视化呈现:

Call Chart 横轴就是时间线，用来展示方法开始与结束的确切时间，纵轴则自上而下展示了方法间调用和被调用的关系。Call Chart 已经比原数据可读性高很多，但它仍然不方便发现那些运行时间很长的代码，这时我们便需要使用 Flame Chart。

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···

Flame Chart 提供了一个调用栈的聚合信息。与 Call Chart 不同的是，它的横轴显示的是百分比数值。由于忽略了时间线信息，Flame Chart 可以展示每次调用消耗时间占用整个记录时长的百分比。同时纵轴也被对调了，在顶部展示的是被调用者，底部展示的是调用者。此时的图表看起来越往上越窄，就好像火焰一样，因此得名:

Flame Chart 是基于 Call Chart 来重新组织信息的。从 Call Chat 开始，合并相同的调用栈，以耗时由长至短对调用栈进行排序，就获得了 Flame Chart:



Call Chart

对比两种图表不难看出，左边的 Call Chart 有详细的时间信息，可以展示每次调用是何时发生的；右边的 Flame Chart 所展示的聚合信息，则有助于发现一个总耗时很长的调用路径:

Top Down Tree

前面介绍的两种图表，可以帮助我们从两种角度纵览全局。而如果我们需要更精确的时间信息，就需要使用 Top Down Tree。在 CPU Profiler 中，Top Down 选项卡展示的是一个数据表格，为了便于理解其中各组数据的意义，接下来我们会尝试构建一个 Top Down Tree。

构建一个 Top Down Tree 并不复杂。以 Flame Chart 为基础，您只需要从调用者开始，持续添加被调用者作为子节点，直到整个 Flame Chart 被遍历一遍，您就获得了一个 Top Down Tree:

Flame Chart

Top Down

对于每个节点，我们关注三个时间信息:

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...

- Children Time —— 调用其他方法的时间；
- Total Time —— 前面两者时间之和。

有了 Top Down Tree，我们能轻易将这三组信息归纳到一个表格之中:

下面我们来看一看这些时间信息是怎么计算的。左手边是和前面一样的 Flame Chart 示例。右边则是一个 Top Down Tree。

我们从 A 节点开始:

- A 消耗了 1 秒钟来运行自己的代码，所以 Self Time 是 1；
- 然后它消耗了 9 秒中去调用其他方法，这意味着它的 Children Time 是 9；
- 这样就一共消耗了 10 秒钟，Total Time 是 10；
- B 和 D 以此类推...

值得注意的是，D 节点只是调用了 C，自己没做任何事，这种情况在方法封装时很常见。所以 D 的 Children Time 和 Total Time 都是 2。

下面是表格完全展开的状态。当您在 Android Studio 中分析应用时，CPU Profiler 会完成上面所有的计算，您只要理解这些数字是怎么产生的即可:

对比左右两边: Flame Chart 比较便于发现总耗时很长的调用链，而 Top Down Tree 则方便观察其中每一步所消耗的精确时间。作为一个表格，Top Down Tree 也支持按单独维度进行排序，这点同样非常实用。

Bottom Up Tree

当您希望方便地找到某个方法的调用栈时，Bottom Up Tree 就派上用场了。"树" 如其名，Bottom Up Tree 从底部开始构建，这样我们就能通过在节点上不断添加调用者来反向构建出树。由于每个独立节点都可以构建出一棵树，所以这里其实是森林 (Forest):

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...

Flame Chart Bottom Up

让我们再做些计算来搞定这些时间信息。

表格有四行，因为我们有四个树在森林中。从节点 C 开始:

- Self Time 是 4 + 2 = 6 秒钟；
- C 没有调用其他方法，所以 Children Time 是 0；
- 前面两者相加，总时间为 6 秒钟。

看起来与 Top Bottom Tree 别无二致。接下来展开 C 节点，计算 C 的调用者 B 和 D 的情况。

在计算 B 和 D 节点的相关时间时，情况与前面的 Top Bottom Tree 有所不同:

- 由于我们在构建基于 C 节点的 Bottom Up Tree，所以所有时间信息也都是基于 C 节点的。这时我们在计算 B 的 Self Time 时，应当计算 C 被 B 调用的时间，而不是 B 自身执行的时间，这里是 4 秒；对于 D 来说，则是 2 秒。
- 由于只有 B 和 D 调用 C 的方法，它们的 Total Time 之和应与 C 的 Total Time 相等。

下一个树是 B 节点的 Bottom Up Tree，它的 Self Time 是 3 秒，Children Time 是用来调用其他方法的时间，这里只有 C，所以是 2 秒。Total Time 永远都是前两者之和。下面便是整个表格展开的样子:

当您想要观察某个方法如何被调用，比如这个 nanoTime() 方法时，您可以使用 Bottom Up Tree 并观察 nanoTime 方法的子节点列表，通过右边的时间数据，您可以找到那个您所感兴趣的调用:

备忘表

前面介绍了四种不同的数据图表，并且还详细解释了一些数据是如何被计算出来的。如果您觉得头绪太多很难记住，没关系，下面这个简明的备忘表就是为您准备的:

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ···

总结

本文介绍了 Android Studio Profiler 中的两种数据分析工具。

其中 Memory Profiler 可以自动检测 Activity 和 Fragment 的内存泄漏，而通过了解和使用 Memory Profiler 中数据分析功能提供的数据，也可以发现和解决其他类型的内存泄漏问题。

有关 CPU Profiler 则介绍了 Call Chart、Flame Chart、Top Down、Bottom Up 这四种维度的数据呈现。

希望这些内容能够帮助您更加了解 Android Profiler。如仍有疑问，欢迎在下方留言。也欢迎通过 Android Studio 反馈使用中遇到的问题。

您也可以通过视频回顾 2019 Android 开发者峰会演讲 —— 读懂 Android Studio 分析工具数据:

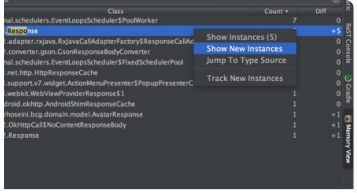
https://v.qq.com/x/page/m3020p5ndvs.html
v.qq.com

点击这里即刻阅读更多应用性能相关内容

发布于 06-24

Android Studio

▲ 赞同 19 ▼ 2 条评论 分享 喜欢 收藏 申请转载 ...



兑一说Android Studio和IDEA中一个很有用的内存调试插件

和秋z　　发表于极乐科技

Android 性能优化之内存优化

前言 Android App优化这个问题，我相信是Android开发者一个永恒的话题。本篇文章也不例外，也是来讲解一下Android内存优化。那么本篇文章有什么不同呢？ 本篇文章主要是从最基础的Android系...

未来

Android Studio环境下Opencv导入Android方法

本机环境： Android Studio 2.3 gradle 3.3 Opencv for Android 3.2 之前经过一番艰难的操作，用以下方法可以将Opencv for Android 成功导入至Android Studio项目中。 1、官网上下载opencv ...

胡国盛

Android Studio项目创建和模拟器配置

Wayne...　　发表于程序员实验...

2 条评论　　切换为时间排序

写下你的评论...

银滩尹　　08-17

请教楼主 android studio simpleperf 符号表怎么导入呀

赞

peitie　　09-04

好巧吖，最近刚好看到T profiler，又刚好接触mevalet维护工作[可怜]

赞