ISSN 1673-9418 CODEN JKYTA8
Journal of Frontiers of Computer Science and Technology
1673-9418/2000/00(00)-0000-00
doi: 10.3778/j.issn.1673-9418.

E-mail: fcst@vip.163.com
http://www.ceaj.org
Tel: +86-10-89056056

基于 Framework 层插桩构建关键路径对 Android 应用性能分析

薛海龙[1,2], 陈渝[3], 雷蕾[4],王丹[5]

1. 北京工业大学 计算机学院, 北京市 100124
2. 清华大学 计算机系, 北京市 100084
3. 清华大学 计算机系, 北京市 100084
4. 北京工业大学 计算机学院, 北京市 100124
5. 北京工业大学 计算机学院, 北京市 100124

Performance Analysis of Android Application Based on Framework Pile Construction

XUE HaiLong[1,2], CHEN Yu[3], LEI Lei[4] ,WANG Dan[5]

1. Computer Science and Technology Department, Beijing University of Technology, 100124, China
2. Computer Science and Technology Department, Tsinghua University, 100124, China
3. Computer Science and Technology Department, Tsinghua University, 100124, China
4. Computer Science and Technology Department, Beijing University of Technology, 100124, China
5. Computer Science and Technology Department, Beijing University of Technology, 100124, China

+ Corresponding author: Phn: +86-18810360877, E-mail: 847036983@qq.com

Author. Title. Journal of Frontiers of Computer Science and Technology, 2000, 0(0): 1-000.

Abstract: This paper presents a method based on asynchronous tracking to realize the performance detection and analysis of Android application response in order to solve the problem that the Android system response performance is low and the complexity of the program execution process is increased which caused by asynchronous programming to improve the user experience. Through the key code in the Android system's Framework layer, this paper tracks the asynchronous thread execution process and constructs the critical path across the asynchronous call boundary. Through the analysis of the key path, the main reason for the response performance can be obtained. The experimental results show that the expectation has been achieved.

* Supported by Beijing Natural Science Foundation,No.4173072 (北京市自然科学基金资助项目,编号.4173072).
Received 2000-00, Accepted 2000-00.

Key words: Android; Performance Analysis; Framework; Log; Critical Path

摘 要: 为解决开发人员多采用异步编程方式来提高用户体验而引发的 Android 系统响应性能低下、程序执行过程复杂性增加的问题，提出一种基于异步跟踪实现 Android 应用响应性能检测与分析的方法。通过对 Android 系统的 Framework 层中关键代码进行插桩，完成对程序异步线程执行过程的跟踪，并构建出跨越异步调用边界的关键路径。通过对关键路径的分析，获得响应性能的主要原因。实验结果表明，达到了预期目标。

关键词: Android；性能分析；Framework；Log；关键路径

文献标志码：A 中图分类号：TP391.1

1 引言

Android 系统以其突出的市场占有率和开放的特性吸引了大量的开发人员，上百万类型丰富而且创意独特的应用程序应运而生。然而，用户在享受丰富多彩应用的同时，较差的响应性能一直影响着使用体验。例如，应用的启动时间过长，不能在短时间内迅速对用户输入进行反馈等。在功能相似的 Android 应用中，响应性能差的往往最先被淘汰。面对竞争如此激烈的 Android 应用市场，开发人员迫切的需要研制有效的手段来解决这个问题。

Android 系统的程序开发框架使用单线程模型来处理用户输入事件，应用启动时，系统创建一个 UI 线程来运行应用程序 ,这个线程主要负责处理生命周期事件、用户输入事件和显示更新事件。所有的事件默认情况下在 UI 线程中顺序执行。因此，响应性能低下的原因往往是由于在 UI 线程执行耗时操作导致的，例如磁盘 I/O、数据库的读取或者网络访问等，这些操作会导致 UI 线程阻塞，使得显示不能及时更新，或者用户陷入长时间的等待中。为避免出现这个问题，移动开发采用了异步编程技术，将这些耗时操作放到 UI 线程以外的工作线程中去执行。由于 Android 提供了多种异步编程模型供开发人员使用，例如 HandlerThread、IntentService、AsyncTask、ThreadPool 以及新建 Thread 类来执行异步操作。因此，使用异步编程技术可在一定程度上提升系统性能。

然而，从异步编程模型的特点可知，采用该模型的程序往往会在多条线程中异步执行，使得它的执行过程变得十分复杂。如果使用了这些编程模型，性能却没有达到预期的效果，开发人员就很难找到性能异常的根源所在。所以需要研究帮助开发人员分析程序在多条线程中的执行情况的有效方法。

2012 年，微软的 Lenin Ravindranath 等人为了帮助 Windows 手机应用开发人员了解相应应用在真实使用过程中的性能瓶颈和运行失败原因，提出了一种在真实使用过程中跟踪应用性能的检测分析方法。该方法关注以用户对 UI 的操作开始，以完成由操作触发的应用程序中的所有同步和异步任务结束的用户事务上，并认为识别异步调用代码的性能瓶颈需要正确地跟踪跨异步边界的因果关系。在实现上，该方法首先确定构建用户事务的执行跟踪所需的全部信息，包括 UI 操作事件信息、线程执行信息、异步调用因果关系、线程同步信息、UI 更新信息以及未处理的异常。然后通过引入两个自定义的库，对 Applications 层 App 的二进制代码进行动态插桩，向开发人员展示异步程序执行的全

部流程，最终构建出程序执行的关键路径，给开发人员指向可能导致响应性能异常的根源。

利用这种方法，Lenin Ravindranath 等人开发了 AppInsight，它准确地向开发人员展示了异步程序执行的整体流程，并构建出程序执行关键路径帮助开发人员识别性能瓶颈和异常根源。AppInsight 是基于 Windows 编程框架中的指定接口实现的，因此只能对 Windows 平台上的应用进行性能监测与分析，并不能直接应用到 Android 应用性能的监测分析上。

Facebook 公司在这个方法的基础上，针对于 Android 编程框架的特点，开发出了一套远程性能监控工具，用于对自己所开发的应用在真实用户的使用情况进行监测。FaceBook 官方称利用此方法确实解决了一些性能问题。

然而，以上对 App 的二进制源码进行动态插桩的办法只能作用于 App 自身，即只有被测的 App 启动之后，才可以监测用户的行为操作得到插入的 Log，App 启动的过程中的异步性能瓶颈是没办法检测到的。针对这个问题，本文提出对 Android 系统的 Framework 层进行静态插桩来捕捉异步任务处理的各项信息，以监测到 App 启动之后，并覆盖到 App 启动过程中的所有异步执行流程。本文的主要贡献：

（1）提出在 Framework 层进行静态插桩对应用程序层的所有异步任务的执行进行的跟踪方法。该方法可以有针对性地在 Framework 层的 API 接口中找到其对应的关键代码，通过一次插桩就可以解决所有应用程序的监测，相比较对应用程序的插桩是更轻量级的，减轻了性能分析的复杂性。

（2）在实现策略上，本文是通过对异步任务上的所有事务信息进行插桩后，找到处理流程的一条关键路径。该路径上包括触发用户事务的关键事件。如果存在异步任务的处理会有新线程开始的时间戳，之后就是异步任务开始处理的过程，处理完成后会通过 Handler 发送消息给主线程进行界面的刷新。通过对关键路径的分析，可帮助开发人员找到应用程序的异步任务性能瓶颈。

2 Android 系统结构概述及其异步处理任务

2.1 FrameWork 层

Android 的系统架构和其操作系统一样，采用了分层的架构，分为四层，从高层到低层分别是应用程序层、应用程序框架层（FrameWork）、系统运行库层和 Linux 核心层，如图 1 所示。

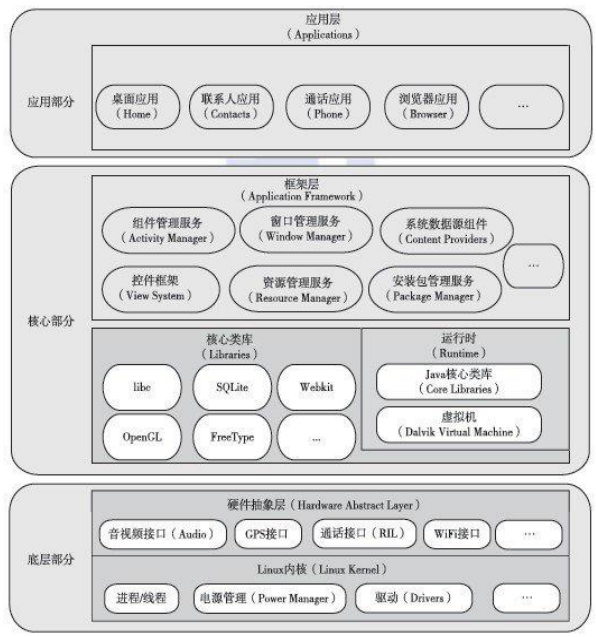

Fig.1 Android System Structure

图 1 Android 系统结构

Framework 层提供给 Android 开发人员一系列的服务和 API 的接口，然后将一些基本功能实现，通过接口提供给上层调用，可以重复调用。应用程序层的所有 App 都是通过调用 Framework 层的各种 API 来实现业务需求的。

2.2 异步任务处理机制

Android 程序的大多数代码操作都必须在主线程执行，例如系统事件、输入事件、程序回调服务，UI 绘制等。那么在上述事件或者方法中插入的代码也将执行在主线程中执行。一旦在主线程里面添加了操作复杂的代码，这些代码会影响应用程序的响应性能。因此，为了提高用户体验或者避免 ANR（Application is not responding）通常都会把一些耗时的任务放在子线程中去处理。

对于异步执行的任务，则不需要等待返回结果，而是等任务处理完成后，再通知界面刷新。在 Android 中开启异步任务的方式主要有 Thread、HandlerThread、IntentService、AsyncTask 和 ThreadPool 这五种，开发人员可以针对不同的应用场景选择不同的异步处理方式。

子线程和主线程的消息传递使用 Android 中的异步消息处理机制：Handler。Handler 是谷歌封装好的一个消息处理接口，它可以绑定在主线程或者子线程中，当异步任务处理完成后可以利用 Handler 发送一个消息出去，主线程接收到这个消息之后就说明异步任务已经执行完成了进而可以刷新界面，这就实现了异步消息的传递。

3 关键路径构建及分析

本文通过分析用户事务期间所对应的关键路径来判断性能瓶颈。为方便下面的叙述，下面先给出这些术语的定义。

定义 1（用户事务） 用户事务指用户对 UI 的操作开始，并完成操作触发的所有异步任务结束。例如在图 2 中，用户事务从 onClick 事件开始，并在 UI update 事件结束。

定义 2（关键路径） 关键路径是指用户事务中的瓶颈路径，从而更改关键路径的任何部分的长度将改变用户感知的持久性。关键路径以用户操纵事件开始，并以 UI 更新事件结束。在图 2 中，从①到⑦的整个路径构成事务的关键路径。

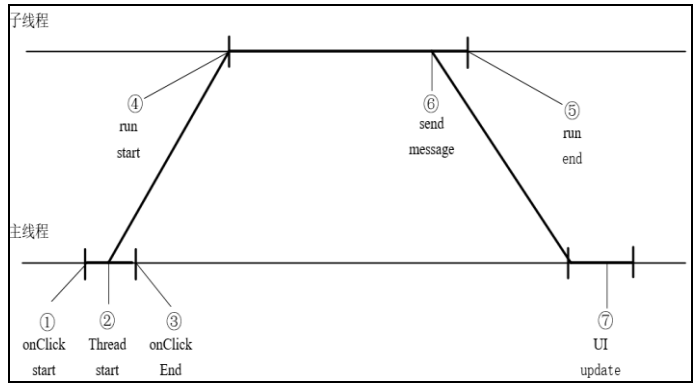

Fig.2 Critical Path

图 2 关键路径

本文通过对这条关键路径上的节点所对应 FrameWork 层中的源码进行插桩来确定每个节点运行的时间点和整个用户事务的运行时间，并通过 Logcat 输出我们的插桩信息。

鉴于 Android 自带的 Logcat 功能非常强大，利用它可以得到我们的插桩 Log，并通过进程号和一些筛选条件可以过滤到大部分的无关 Log，然后通过打印 Log 所在的线程 ID，可以一目了然的得到用户事务的关键路径。

本文对应用程序的迭代过程中的所有用户事务构建关键路径，并对关键路径的运行时间进行对比分析。如果某个版本的一个用户事务的运行时间明显过长，可以确定此处是有问题，即是性能瓶颈，然后把这个问题反馈给开发人员，让开发人员对代码进行优化。

4 事件插桩

由于关键路径以用户操纵事件开始，并以 UI 更新事件结束，下面介绍事件的插桩。

4.1 点击事件插桩

Android 中点击事件分为四类：(1) 匿名内部类；(2) 自定义事件监听类；(3) Activity 继承 View.OnClickListener，由 Activity 实现 OnClick(View view)方法；(4) 在 XML 文件中“显示指定按钮的 onClick 属性，这样点击按钮时会利用反射的方式调用对应 Activity 中的 Click()方法。

无论哪种写法，经过分析源码，其最终都会调用 Framework 中 View.java 文件中的 performClick 函数，所以我们在这个位置插桩就可以得到点击事件触发和结束的临界点，如图 2 中的①和③所示。

4.2 异步消息处理的插桩

Android 中异步消息处理使用最广泛的就是 Handler 机制，Handler 处理消息的流程如图 3 所示，包括以下四个要素：(1) Message：消息，理解为线程间通讯的数据单元；(2) Message Queue：消息队列，用来存放通过 Handler 发布的消息，按照先进先出执行；(3) Handler：Handler 是 Message 的主要处理者，负责将 Message 添加到消息队列以及对消息队列中的 Message 进行处理；(4) Looper：循环器，扮演 Message Queue 和 Handler 之间桥梁的角色，循环取出 Message Queue 里面的 Message，并交付给相应的 Handler 进行处理。

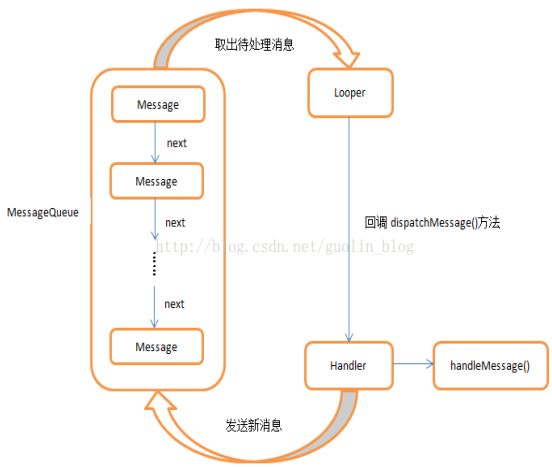

Fig.3 Handler DispatchMessage Process

图 3 Handler 处理消息流程

经过对 Handler 源码的分析，我们确定对于异步消息处理的插桩需要分为两个部分：子线程的 sendMessage 和主线程的 handlerMessage，找到这个关系就可以确定主线程和子线程的因果关系。

4.2.1 sendMessage

在应用层子线程发送消息主要有 send 和 post 两种方式，其中 send 包括 sendEmptyMessage(int what)、sendEmptyMessageAtTime(int what,long uptimeMillis)、sendEmptyMessageDelayed (int what, long delayMillis)、sendMessage (Message msg)四种；而 post 包括 post(Runnable)、postAtTime(Runnable long) 和 postDelayed (Runable long) 三种。跟踪这几种方法的调用栈发现最终调用的会是 sendMessageAtTime 和 sendMessageAtFrontOfQueue 其中一个函数，而这两个函数最终返回的都是 enqueueMessage，所以我们选择在 enqueueMessage 的关键位置进行插桩，得到如图 2 中的⑥。

4.2.2 handlerMessage

应用层处理消息是在绑定 Handler 的线程中进行的 handlerMessage 处理 MessageQueue 中的消息都会调用 FrameWork 层的的 dispatchMessage，因此我们选择在 dispatchMessage 的关键位置进行插桩，得到如图 2 所示的⑦。

4.3 子线程插桩

Android 中开启子线程主要有 Thread、HandlerThread、IntentService、AsyncTask 和 ThreadPool 五种方式，针对不同的应用场景开发人员可以选择不同的开启方式。其中 AsyncTask 和 ThreadPool 都是开启一个线程池来处理异步任务，而这种方式开启子线程其实就是封装了一个

Thread，因此这三种方式可以归为一种来讨论，所以这五种方式可以分为三类 :Thread、HandlerThread 和 IntentService。

4.3.1 Thread

Android 用 Thread 开启子线程的方式和 Java 中的相似，有继承 Thread 和实现 Runnable 接口两种方式，Runnable 接口里只有一个无参无返回值的 run()方法，而 Thread 类也是 实现了 Runnable 接口并重写了 run()方法，所以我们主要分析 Thread 类来找关键位置插桩。

开发人员在应用层 start 开启子线程是调用的 libcore 库中 Thread 类的 start 方法，接着转调 VMThread 的 native 方法 create，然后转到 native 层去创建子线程并设置属性，创建成功后在调用 Thread 类的 run 方法执行开发人员自定义的异步任务。在这个创建的过程中，我们需要关注的有两个地方：线程 start 和线程 run。线程 start 是主线程调用的，这就是子线程开启的始点，如图 2 所示的②。线程 run 的过程是子线程处理异步任务的过程，找到子线程 run 的起始点，就得到了如图 2 所示的④和⑤。

4.3.2 HandlerThead

HandlerThread 本质上是一个 Thread 对象，只不过其内部帮我们创建了该线程的 Looper 和 MessageQueue。开发人员使用 HandlerThread 很简单，只需要新建一个对象，让它 start，并在在 handlerMessage 中处理异步任务。

我们分析源码知道这里调用的 start 仍然会调用是 Thread 的 start 方法来创建新的线程，但是创建完成之后不会调用 Thread 的 run 方法，而直接调用 HandlerThread 的 run 方法，因此我们需要在 HandlerThread 的 run 方法里插桩来分辨我们开启的是一个 HandlerThread。它处理异步任务是在 handlerMessage 中进行的，这就可以利用之前消息处理的插桩结合线程 ID 可以得到它处理异步任务的临界点。

4.3.3 IntentService

IntentService 是 Looper，Handler，Service 的集合体，IntentService 是继承于 Service 并处理异步请求的一个类，在 IntentService 内有一个工作线程来处理耗时操作，启动 IntentService 的方式和启动传统 Service 一样。IntentService 可以自动开启一个 HandleThread，并自动调用 IntentService 中的 onHandleIntent 方法来处理异步任务。

既然 IntentService 是开启了一个 HandlerThread，那我们之前插的桩都可以直接使用，只是 IntentService 处理异步任务和之前的方式都不同，它是自定义了一个 onHandleIntent 方法，因此我们需要捕捉它处理异步任务的临界点就需要在这个方法里插桩。

4.4 界面刷新插桩

Android 中的任何一个布局、任何一个控件其实都是直接或间接继承自 View 实现的，当然自定义控件也不例外，所以说这些 View 应该都具有相同的绘制流程与机制才能显示到屏幕上。经过总结发现每一个 View 的绘制过程都必须经历三个最主要的过程，也就是 measure、layout 和 draw。而整个 View 树的绘图流程是在 ViewRootImpl 类的 performTraversals 方法开始的，所以需要监测界面刷新的流程就需要在 performTraversals 处插桩。

5 实验与分析

为了测试此方法的可行性和有效性，本文设计了相关的测试用例，并对实际 Android 项目中使用。

5.1 实验环境

5.1.1 软件环境

我们的所有实验都在 OPENTHOS 系统上完成的。OPENTHOS 是将 Android 5.1 原生系统移植到 PC 端并实现多窗口、多任务等一系列功能的开源操作系统，是由清华大学、清华同方和一铭公司共同联合开发的。

5.1.2 硬件环境

我们把 OPENTHOS 系统安装在清华同方 T45 笔记本上进行我们的实验与分析。T45 的配置如下：

CPU：Intel 酷睿 i5 6200U

内存：4GB

磁盘：500G

显卡：2GB 独立显卡

5.2 集合所有异步编程模型的实例

在上文我们已经介绍过，在 Android 异步编程模型中有五种开启子线程的方式，为了验证本文我们插桩方案的有效性，我们需要对每一种进行测试，基于此我自己写了一个测试用例。本文设计的这个测试用例的主要功能是根据一个图片的网络地址，开启一个子线程下载这个图片并通知主线程刷新界面，将图片显示到界面上，我分别用 Thread、HandlerThread 、 IntentService 、 AsyncTask 、 ThreadPool 实现这个功能，通过我们的插桩得到异步任务处理的关键路径。图 4 所示就是一条关键路径，其中前两列是日期和时间，第三列是进程 ID，第四列是线程 ID、第五列和第六列是插桩的 Log 标签，第七列是主要的 Log 信息。

如图 4 是 Thread 的关键路径， 第 1 行和第 4 行是用户点击事件的临界点，对应图 2 的①和③；第 3 行是主线程开启子线程时调用的 Thread 的 start 方法，对应的是图 2 的②；第 5 行和第 9 行是子线程处理耗时任务的临界点，对应图 2 的④和⑤；第 6 行是子线程处理完异步消息后向主线程发送消息通知主线程刷新界面，对应图 2 的⑥；第 7、8、10、11 行是主线程处理子线程发来的消息并进行界面刷新的过程，对应图 2 的⑦。

```
1.06-30 15:07:33.006  5180  5180 I aaaaaaaa: onClick()  start5180  main
2.06-30 15:07:33.006  5180  5180 D aaaaaaaa: currentThread is 1  main
3.06-30 15:07:33.006  5180  5180 D aaaaaaaa: Thread start
4.06-30 15:07:33.006  5180  5180 I aaaaaaaa: onClick()  end5180  main
5.06-30 15:07:33.007  5180  5219 D aaaaaaaa: new Thread start
6.06-30 15:07:33.076  5180  5219 I aaaaaaaa: enqueueMessage  start  5219
Thread-169  main  1
7.06-30 15:07:33.076  5180  5180 I aaaaaaaa: dispatchMessage start  5180
8.06-30 15:07:33.076  5180  5180 I aaaaaaaa: dispatchMessage end  5180
9.06-30 15:07:33.076  5180  5219 D aaaaaaaa: new Thread end
10.06-30 15:07:33.081  5180  5180 I aaaaaaaa: draw start
11.06-30 15:07:33.083  5180  5180 I aaaaaaaa: draw end
```

Fig.4 Thread

图 4 Thread

图 5 是 HandlerThread 的关键路径，它与 Thread 的最大区别就是子线程在 dispatchMessage 中处理异步任务，所以第 4 行的 Log 会显示这是一个 HandlerThread，它的异步任务是在第 7 行和第 8 行的地方处理的。

```
1.06-30 15:07:39.688  5180  5180 I aaaaaaaa: onClick()  start5180  main
2.06-30 15:07:39.688  5180  5180 D aaaaaaaa: currentThread is 1  main
3.06-30 15:07:39.688  5180  5180 D aaaaaaaa: Thread start
4.06-30 15:07:39.689  5180  5226 I aaaaaaaa: new HandlerThread run  start
5.06-30 15:07:39.690  5180  5180 I aaaaaaaa: enqueueMessage  start  5180  main
handlerThread_test  172
6.06-30 15:07:39.690  5180  5180 I aaaaaaaa: onClick()  end5180  main
7.06-30 15:07:39.690  5180  5226 I aaaaaaaa: dispatchMessage start  5226
8.06-30 15:07:39.718  5180  5226 I aaaaaaaa: dispatchMessage end  5226
9.06-30 15:07:39.718  5180  5226 I aaaaaaaa: enqueueMessage  start  5226
handlerThread_test  main  1
10.06-30 15:07:39.718  5180  5180 I aaaaaaaa: dispatchMessage start  5180
11.06-30 15:07:39.718  5180  5180 I aaaaaaaa: dispatchMessage end  5180
12.06-30 15:07:39.731  5180  5180 I aaaaaaaa: draw start
13.06-30 15:07:39.733  5180  5180 I aaaaaaaa: draw end
```

Fig.5 HandlerThread

图 5 HandlerThread

图 6 是 IntentService 的关键路径，上文提到它其实是封装了一个 HandlerThread，所以对 HandlerThread 的插桩也存在，而且 IntentService 是在 onHandleIntent 中处理异步任务，通过对 onHandleIntent 的插桩得到图 6 的第 8 条和第 12 条

Log。

```
1.06-30 15:07:44.046  5180  5180 I aaaaaaaa: onClick()  start5180  main
2.06-30 15:07:44.050  5180  5180 D aaaaaaaa: currentThread is 1  main
3.06-30 15:07:44.050  5180  5180 D aaaaaaaa: Thread start
4.06-30 15:07:44.051  5180  5233 I aaaaaaaa: new HandlerThread run  start
5.06-30 15:07:44.051  5180  5180 I aaaaaaaa: onClick()  end5180  main
6.06-30 15:07:44.051  5180  5180 I aaaaaaaa: enqueueMessage  start  5180  main
IntentService[MyIntentService]  174
7.06-30 15:07:44.051  5180  5233 I aaaaaaaa: dispatchMessage start  5233
8.06-30 15:07:44.051  5180  5233 D aaaaaaaa: IntentService new Thread start
5233
9.06-30 15:07:44.085  5180  5233 I aaaaaaaa: enqueueMessage  start  5233
IntentService[MyIntentService]  main  1
10.06-30 15:07:44.086  5180  5180 I aaaaaaaa: dispatchMessage start  5180
11.06-30 15:07:44.086  5180  5180 I aaaaaaaa: dispatchMessage end  5180
12.06-30 15:07:44.086  5180  5233 D aaaaaaaa: IntentService new Thread end  5233
13.06-30 15:07:44.086  5180  5233 I aaaaaaaa: dispatchMessage end  5233
14.06-30 15:07:44.098  5180  5180 I aaaaaaaa: draw start
15.06-30 15:07:44.099  5180  5180 I aaaaaaaa: draw end
```

Fig.6 IntentService

图 6 IntentService

图 7 和图 8 分别是 AsyncTask 和 ThreadPool 的关键路径 ,这两种异步处理的方式和 Thread 是一致的，所以得到的关键路径和 Thread 无很大的区别。

```
1.06-30 15:07:48.350  5180  5180 I aaaaaaaa: onClick()  start5180  main
2.06-30 15:07:48.351  5180  5180 D aaaaaaaa: currentThread is 1  main
3.06-30 15:07:48.351  5180  5180 D aaaaaaaa: Thread start
4.06-30 15:07:48.351  5180  5180 I aaaaaaaa: onClick()  end5180  main
5.06-30 15:07:48.352  5180  5237 D aaaaaaaa: new Thread start
6.06-30 15:07:48.392  5180  5237 I aaaaaaaa: enqueueMessage  start  5237
AsyncTask #5  main  1
7.06-30 15:07:48.392  5180  5180 I aaaaaaaa: dispatchMessage start  5180
8.06-30 15:07:48.393  5180  5180 I aaaaaaaa: dispatchMessage end  5180
9.06-30 15:07:48.393  5180  5237 D aaaaaaaa: new Thread end
10.06-30 15:07:48.401  5180  5180 I aaaaaaaa: draw start
11.06-30 15:07:48.404  5180  5180 I aaaaaaaa: draw end
```

Fig.7 AsyncTask

图 7 AsyncTask

```
1.06-30 15:07:52.757  5180  5180 I aaaaaaaa: onClick()  start5180  main
2.06-30 15:07:52.758  5180  5180 D aaaaaaaa: currentThread is 1  main
3.06-30 15:07:52.758  5180  5180 D aaaaaaaa: Thread start
4.06-30 15:07:52.758  5180  5180 I aaaaaaaa: onClick()  end5180  main
5.06-30 15:07:52.758  5180  5241 D aaaaaaaa: new Thread start
6.06-30 15:07:52.795  5180  5241 I aaaaaaaa: enqueueMessage  start  5241
pool-1-thread-1  main  1
7.06-30 15:07:52.795  5180  5180 I aaaaaaaa: dispatchMessage start  5180
8.06-30 15:07:52.795  5180  5180 I aaaaaaaa: dispatchMessage end  5180
9.06-30 15:07:52.795  5180  5241 D aaaaaaaa: new Thread end
10.06-30 15:07:52.798  5180  5180 I aaaaaaaa: draw start
11.06-30 15:07:52.799  5180  5180 I aaaaaaaa: draw end
```

Fig.8 ThreadPool

图 8 ThreadPool

得到以上的关键路径之后，为了对比当异步处理出现问题影响响应性能的实例，我们让子线程都 sleep 2 秒，这样得到的关键路径如图 9 所示，可以发现第 5 行开始子线程处理异步任务，第 7 条处理完成的时间比图 6.1 延长了 1.993 秒，这就说明这次的异步处理任务是有性能问题的（其余的几种异步方式结果都是类似的，就不再赘述）。

```
1.07-03 16:41:10.624  3654  3654 I aaaaaaaa: onClick()  start3654  main
2.07-03 16:41:10.624  3654  3654 D aaaaaaaa: currentThread is 1  main
3.07-03 16:41:10.624  3654  3654 D aaaaaaaa: Thread start
4.07-03 16:41:10.624  3654  3654 I aaaaaaaa: onClick()  end3654  main
5.07-03 16:41:10.624  3654  3769 D aaaaaaaa: new Thread start
6.07-03 16:41:12.686  3654  3769 I aaaaaaaa: enqueueMessage  start  3769
Thread-175  main  1
7.07-03 16:41:12.686  3654  3769 D aaaaaaaa: new Thread end
8.07-03 16:41:12.686  3654  3654 I aaaaaaaa: dispatchMessage start  3654
9.07-03 16:41:12.686  3654  3654 I aaaaaaaa: dispatchMessage end  3654
10.07-03 16:41:12.689  3654  3654 I aaaaaaaa: draw start
11.07-03 16:41:12.689  3654  3654 I aaaaaaaa: draw end
```

Fig.9 Thrad Performance Exception

图 9 Thread 性能异常

5.3 OPENTHOS 的 StartMenu 实例

OPENTHOS 在开发的过程中一直碰到 StartMenu 卡顿的问题 ,尤其是在安装的几十个应用的时候，点击 StartMenu 之后需要 2 秒左右才能显示出来，为了分析这个问题我们利用我们的方法在 OPENTHOS 中进行插桩并分析原因。

本文对初期版本和近期的分别进行分析，在对初期版本进行插桩分析的时候发现 StartMenu 启动的时候并没有开启子线程，也就是说启动的过程中所有的任务都是在主线程完成的。带着这个问题对 StartMenu 的源码进行了分析 ,发现在启动的过程中是需要查询数据库把所有已经安装的应用显示出来的，而查询数据库这样耗时的任务在主线程执行，势必会影响性能。

```
1.06-30 20:42:24.929  4038  4038 D aaaaaaaa: Thread start
2.06-30 20:42:24.931  4038  4038 D aaaaaaaa: currentThread is 1  main
3.06-30 20:42:24.931  4038  4057 D aaaaaaaa: new Thread start
4.06-30 20:42:24.931  4038  4057 I aaaaaaaa: enqueueMessage  start   4057
Thread-228  main  1
5.06-30 20:42:24.931  4038  4057 D aaaaaaaa: new Thread end
6.06-30 20:43:35.937  4038  4038 I aaaaaaaa: dispatchMessage start   4038
7.06-30 20:43:35.954  4038  4038 I aaaaaaaa: dispatchMessage end  4038
8.06-30 20:43:35.954  4038  4038 I aaaaaaaa: draw start
9.06-30 20:43:35.995  4038  4038 I aaaaaaaa: draw end
```

Fig.10 StartMenu Analysis 1

图 10 StartMenu 分析 1

接下来，对近期的版本进行分析，插桩得到的 Log 信息如图 10 所示 ,第 3 行是子线程处理耗时任务，即查询数据库的开始点，第 5 行是结束点，但是可以发现这本应该是一个时间段，结果却显示是一个时间点，而且子线程只发送了一条消息就结束了，这绝对是不应该的。带着这样的问题，我们分析源码并梳理 StartMenu 的逻辑关系，发现启动的时候开启的子线程仍然没有处理耗时任务，而只是进行了一个消息的传递，这和初期的版本区别并不大。所以我们和开发人员一起重新整理 StartMenu 的逻辑流程，对代码进行调整，得到最新的版本。如图 11 所示，这样耗时的任务就在子线程中执行，所以性能也提升了不少。

```
1.06-30 18:12:55.121  3082  3082 D aaaaaaaa: Thread start
2.06-30 18:12:55.122  3082  3462 D aaaaaaaa: new Thread start
3.06-30 18:12:55.143  3082  3462 I aaaaaaaa: enqueueMessage  start   3462
Thread-148  main  1
4.06-30 18:12:55.143  3082  3462 D aaaaaaaa: new Thread end
5.06-30 18:12:55.168  3082  3082 I aaaaaaaa: enqueueMessage  start   3082  main
main  1
6.06-30 18:12:55.169  3082  3082 I aaaaaaaa: dispatchMessage start   3082
7.06-30 18:12:55.169  3082  3082 I aaaaaaaa: dispatchMessage end  3082
8.06-30 18:12:55.169  3082  3082 I aaaaaaaa: draw start
9.06-30 18:12:55.179  3082  3082 I aaaaaaaa: draw end
```

Fig.10 StartMenu Analysis 2

图 10 StartMenu 分析 2

6 结论

1）对 Android 的 FrameWork 层进行插桩，把插桩得到的 Log 信息保存下来进行离线分析，以发现异步处理任务的关键路径，可以帮助移动应用程序的开发人员监控和诊断他们的应用程序的性能。

2）对 FrameWork 层进行插桩是轻量级的，只需要一次插桩就可以获得关键路径，有更好的通用性，对于迭代发展的版本之间的性能对比尤其明显。

3）对 FrameWork 层进行插桩是可发现程序中影响性能的瓶颈所在，并向开发人员提供反馈。

References:

[1] Ravindranath L, Padhye J, Agarwal S, et al. AppInsight: Mobile App Performance Monitoring in the Wild[C]//OSDI. 2012, 12: 107-120.

[2] Kang Y, Zhou Y, Xu H, et al. PersisDroid: Android Performance Diagnosis via Anatomizing Asynchronous Executions[J]. arXiv preprint arXiv:1512.07950, 2015.

[3] Ravindranath L, Nath S, Padhye J, et al. Automatic and scalable fault detection for mobile applications[C]//Proceedings of the 12th annual international conference on Mobile systems, applications, and services. ACM, 2014: 190-203.

[4] Lu Ji-xiang, LI Ying.Research on performance optimization of Android application [J]. Mechanical Design and Manufacturing Engineering,2003,42(3):82-85.

[5] Chen Peng.Study and implementation of performance monitoring system based on Android application [D] .Guangzhou: South China University of Technology, 2015.

附中文参考文献：

[1] 陆继翔，李映. Android 应用程序的性能优化分析与研究[J]. 机械设计与制造工程, 2013, 42(3): 82-85.

[2] 陈鹏. 基于 Android 应用的性能监控系统的研究与实现[D]. 广州：华南理工大学, 2015.

XUE HaiLong was born in Linqing City, Shandong Province in 1991. He is studying M.S degree in Computer Science and Technology Department from Beijing University of Technology. His research interests include Performance Analysis of Android Application.

薛海龙(1991–)，男，山东省临清市人,就读于北京工业大学研究生部计算机学院 ,主要研究领域为 Android 性能。

CHEN Yu was born in Chongqing City in 1972. He is an associate professor at Tsinghua University. His research interests include Operating system, distributed system; pervasive computing, parallel computing.

陈渝(1972-)，男，重庆市人，清华大学副教授 ,主要研究领域为操作系统,分布式系统 ;普适计算 ,并行计算。

LEI Lei was born in Sanhe City, Hebei Province in 1992. She is studying M.S degree in Computer Science and Technology Department from Beijing University of Technology. Her research interests include Performance Analysis of Android Application.

雷蕾(1992-)，女,河北省三河市人,就读于北京工业大学研究生部计算机学院，主要研究领域为 Android 性能。

WANG Dan was born in 1969. She is a professor at Beijing University of Technology. Her research interests include software analysis, trusted software.

王丹(1969 -),女,北京工业大学教授,主要研究领域为软件分析,可信软件。