Dalvik虚拟机进程模型分析

周毅敏,陈 榕
(同济大学 基础软件工程中心,上海 200092)

摘 要:Android手机操作系统是Google于2008年推出的智能手机操作系统,它的所有应用都是基于Java语言的,它的类Java虚拟机Dalvik提供了所有应用的运行时环境。Dalvik是一个面向Linux作为嵌入式操作系统设计的虚拟机,尤其是它的面向进程的设计,充分利用了Linux进程管理的特点。介绍了Dalvik所依赖的基础,即Linux操作系统内核中进程管理的一些特性和传统Java程序对进程的控制;进而论述了Dalvik的进程模型的特点,从API和本地代码两个层面具体阐述了进程运行、创建和之间通信的部分细节。文中旨在为Dalvik的研究和应用提供参考。

关键词:Dalvik;虚拟机;结合子;fork;写时复制
中图分类号:TP311.52 **文献标识码**:A **文章编号**:1673-629X(2010)02-0083-04

Analysis about Process in Dalvik Virtual Machine

ZHOU Yi-min, CHEN Rong
(System Software Engineering Centre of Tongji University, Shanghai 200092, China)

Abstract: Android mobile operating system is an OS released at 2008 by Google. All the applications are written by Java language. Its semi-Java virtual machine Dalvik provides the whole environment. However, Dalvik is based on the Linux operating system kernel especially its design of the process management which take advantage of the features of Linux process. This article firstly introduces the process in the kernel of Linux operating system and the process in the traditional Java program and then illustrates the features of Dalvik process and also depicts some details of processes' running and creation and communications between them through the two layers API and native codes. The aim of this article is to provide a reference for Dalvik's research and application.

Key words: Dalvik; virtual machine; zygote; fork; copy-on-write

0 引 言

Android[1]是集操作系统、中间件和关键应用为一体的运行于移动设备上的软件包。Dalvik虚拟机是Google自主开发的代号为Dalvik的Java虚拟机技术。Google自主开发Java虚拟机的目的,是避开Sun公司的授权问题,针对移动手机在保证API[2,3]方面兼容的同时对Java虚拟机[4,5]进行大幅优化,使其占用资源更少、运行效率更高。Dalvik是Android的重要组成部分。Android是一个很大的范畴,其中操作系统部分基于Linux2.6的内核,而很多其他组成部分包括Dalvik的实现与操作系统的内核不可分割。笔者深入研究了Dalvik虚拟机的源代码,发现较为底层的部分直接调用了Linux内核的部分[6]。

收稿日期:2009-03-30;**修回日期**:2009-07-06
基金项目:国家"863"计划资助项目(2001AA113400);国家移动通信产品研究开发专项项目(财政部(财建[2005]182号),信息产业部(信部请函[2005]297号))
作者简介:周毅敏(1980-),男,上海人,硕士研究生,研究方向为嵌入式操作系统、系统软件支撑技术;陈 榕,博士生导师,教授,科泰世纪首席科学家,研究方向为嵌入式系统、构件技术。

1 基本概念

1.1 Linux环境中的进程

和大多数操作系统一样,进程是Linux调度的最小单位。调度采用"有条件抢占式"的方式。进程可以自愿地让出执行机会,也可以在必要的时候抢占执行。这样,既为多任务应用提供了灵活性,又为实时系统提供了便利。Linux的进程结构体(进程控制块,PCB)可谓包罗万象,文件系统、虚拟内存、页面管理、信号应有尽有,使进程资源的控制变得更直接更容易。与有些操作系统创建进程的方式不同的是,Linux使用fork语义而不是creat_proc,这是因为复制进程的效率高于创建进程的效率。即使对应用程序而言fork不如creat_proc易用,但是通过提供一个将复制与创建结合在一起的库函数就可提高易用性。此外,Linux进程间通信使用管道、信号、报文、共享内存[7~9]等多种途径。

1.2 Java 编程环境中的进程

在传统 Java 编程环境中，以 Sun Java 手机平台 Phoneme(https://phoneme.dev.java.net)为例，当虚拟机启动一个 Java 应用后，程序逻辑在操作系统中运行于单进程状态。虽然对应用开发者而言，存在可以使用的多线程模型，但是对操作系统而言这些线程并不可见，它们完全是虚拟机模拟出来的，操作系统所见到的只是单线程的进程[10]。虽然，API 中也提供了创建操作系统意义上的进程的类，即 java.lang.Process 和 java.lang.ProcessBuilder 类，可是似乎没有提供更多的 API 用于进程间通信。

1.3 Dalvik 编程环境中的进程

Dalvik 虚拟机中保留了传统 Java 进程控制的 API，并结合 Linux 操作系统的特点加入了特有的进程控制 API，可以控制信号甚至是 fork 语义的进程创建。事实上在 Android 启动时就已启动了 Dalvik 虚拟机，并且使用 fork 语义创建了系统服务进程。

与 Phoneme 不同的是，Dalvik 虚拟机上的线程是对应于操作系统线程的，而且 Dalvik 中的进程对应于操作系统中的进程。Dalvik 启动的进程与操作系统进程的关系如图 1 所示。

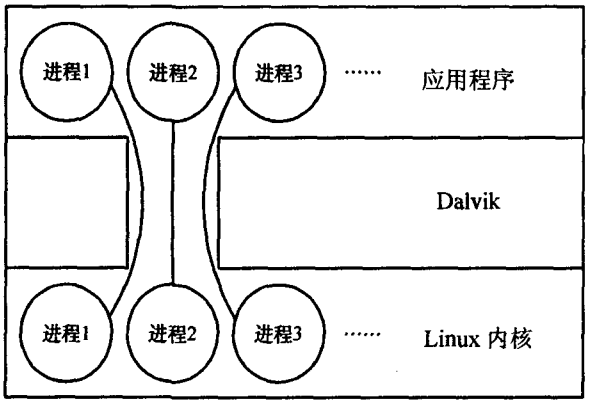

图 1 Dalvik 启动的进程与操作系统进程的关系图

2 Dalvik 运行时的进程模型

2.1 Dalvik 运行时进程关系

Linux 创建进程时，会共享.text 正文段，并且对静态数据采用写时复制(Copy-on-write)的方式操作，这样当一个进程把 Java 运行时所需要的基础类库等加载了以后，再 fork()出来的进程就“继承”了父进程的这些 Java 运行时环境。所以在 Dalvik 中，引入了 zygote(结合子)进程的概念，它加载了 Java 基础类库，在虚拟机启动参数中包含 -Xzygote，并可以创建子进程，此模式称为 Zygote 模式。Dalvik 虚拟机单个实例在 Zygote 模式下运行时的进程关系图如图 2 所示。

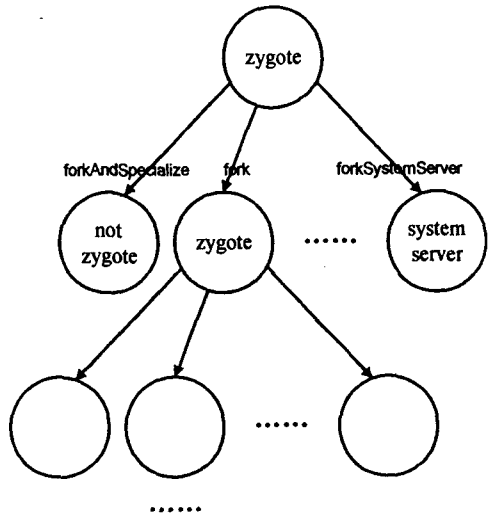

图 2 Dalvik 虚拟机单个实例进程关系图

图 2 中的每个圈代表一个进程，每个箭头表示父进程指向子进程，箭头上的文字表示创建时调用的 Zygote 类的静态方法的名字，圈内的文字代表进程属于哪一类进程。

线程较进程的优点在于占用资源少，而且可以与其它线程共享资源，即共享性好。进程耗费资源，为实现与其它进程共享开销更大。相对进程，线程的缺点在于管理线程非常复杂，要解决麻烦的线程同步问题，通常这是令程序员很头痛的事情。一个线程的崩溃很可能影响到其它线程的正确执行，而进程则不存在这样的问题，因为进程间协作和通信现在已经有很多成熟的模型可以使用。

在 Linux 进程中，线程可以说是一种轻量级的进程，这实际上使进程在这个操作系统上更有优势。在实际应用中，为在效率和易用性、稳定性之间求得平衡，多进程和多线程都要用到。这可能也是 Dalvik 虚拟机在 Java 语言层同时对多进程和多线程提供支持的原因。

Dalvik 直接在 Java API 中提供了操作进程的 API，比如用于控制操作系统进程的 Process 类。除此之外，还提供了一个类 dalvik.system.Zygote 来创建三类进程：zygote 进程，非 zygote 进程，以及系统服务进程。以下是与这三类进程相对应的三个创建进程的静态方法：

* fork()，创建出一个 zygote 进程。
* forkAndSpecialize()，创建出一个非 zygote 进程。
* forkSystemServer()，创建出一个系统服务进程。

其中，zygote 进程可以再 fork 出其它进程，而非 zygote 进程则不能 fork 出其它进程，而系统服务进程

在终止后它的父进程(也就是 fork 出它的进程)也必须终止。

本节是在 Dalvik 的 Java 语言层面研究其语义,下一节将深入虚拟机的源代码,根据函数的调用关系,比较研究以上的三个静态方法。

2.2 Dalvik 进程的创建

这三个方法都是本地方法。

先看 fork 方法,与另两个本地方法不同的是,fork 方法生成的子进程是一个半初始化的进程,它也是 zygote 进程。如果父进程之前已经调用过 addNewHeap,则不再调用它。这里使用的是写时复制技术,因此 zygote 进程是共享同一个堆的。整个初始化到这里就结束了。

forkAndSpecialize 则不同。首先它创建的子进程将不再是一个 zygote 进程(即离开 Zygote 模式,不能再创建子进程),它会启动 HeapWorker 线程(一个主要执行对象的终结函数和引用对象的清理和归队工作的线程)和调试线程。其次,它一定会执行 addNewHeap 函数,创建一个新的堆,因为一个运行特定任务的进程不应该和其它进程共享一个堆。

forkSystemServer 和 forkAndSpecialize 很相似,唯一的不同点是用 forkSystemServer 创建的子进程一旦结束就会立即终止创建它的父进程。

根据 Linux 的 fork 系统调用可以知道,父子进程在 fork 之后各自堆栈互不相关,但是共享程序正文段也就是虚拟机核心库代码,dex 文件除外。这样设计一方面是为了节省内存,另一方面也发挥了 Linux 进程的长处。

fork 方法调用执行的过程如图 3 所示。forkAndSpecialize 方法调用执行的过程如图 4 所示。forkSystemServer 方法调用执行的过程如图 5 所示。

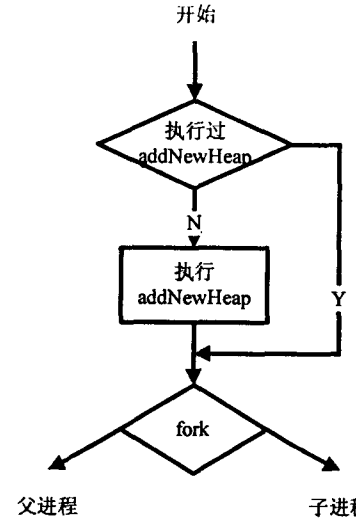

图 3　fork 方法调用

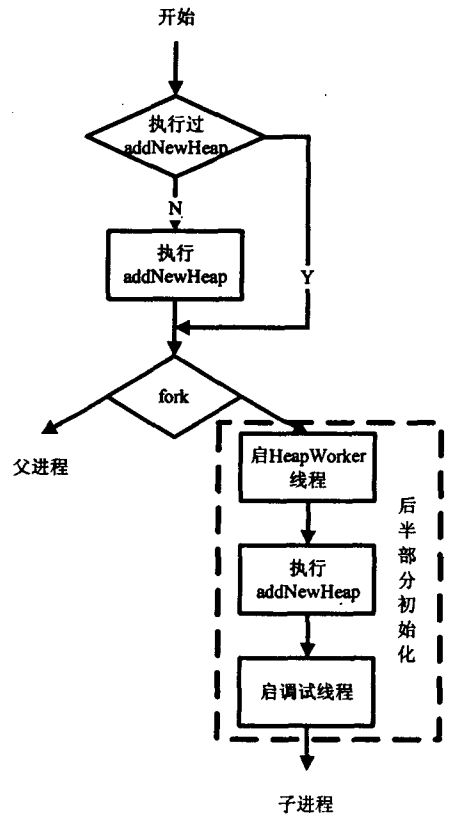

图 4　forkAndSpecialize 方法调用

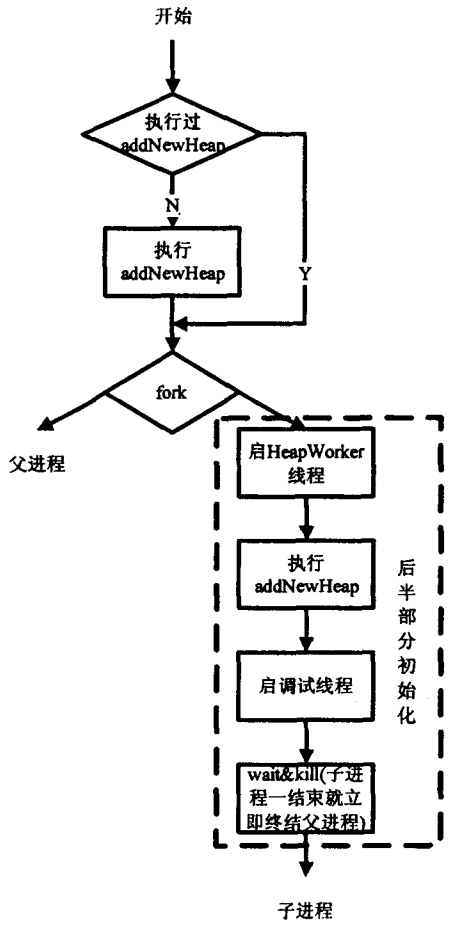

图 5　forkSystemServer 方法调用

3 Dalvik 虚拟机进程间通信关于信号部分

Dalvik 虚拟机暴露的 API 中直接提供了对信号的支持,这体现在 android.os.Process 类中。其中包括了用于终结指定进程的 killProcess(int),以及用于向指定进程发送特定信号的 sendSignal(int,int)。目前可以发送的信号是 SIGNAL_KILL、SIGNAL_QUIT 和 SIGNAL_USR1,Dalvik 的源代码对这些信号都做了特殊处理。

上文对三个创建进程的 API 分析中省略了关于父子进程间通信的信号发送和接收的部分。信号处理也是进程间通信的主要手段,尤其是在 Linux 这样一个以进程为主体的操作系统之上。

以上这三个 API(对应的本地方法的源代码)中父进程都调用了函数 setSignalHandler(),它的实质是将进程中用于接收子进程的 SIGCHLD 信号的处理函数设为 sigchldHandler()。

在这个信号处理函数中,绝大多数部分代码用来根据不同的情况写日志,唯一一处真正用于进程间通信的程序逻辑是下面这段代码:

```
if (pid == gDvm.systemServerPid) { /* 子进程是一个系统服务进程 */
    LOG(.....);
    kill(getpid(), SIGKILL); /* 终止本父进程 */
}
```

即如果子进程是个服务进程,则将本进程(相对于这个子进程来说是父进程)终止。

之前在 fork system server 进程中也有类似的终结父进程的代码,笔者认为并不多余,这是个双保险以保证父进程的正确终结。

当创建非 zygote 进程和 system server 进程时,子进程似乎也有调用 setSignalHandler()。但是后面又调用了 unsetSignalHandler 把这个处理函数删除了。因此,完全符合 API 设计的语义。

如果虚拟机启动时,没有加上"-Xrs"和"-Xnoquithandler"参数,那么在创建非 zygote 进程时,将会触发一个线程运行 signalCatcherThreadStart 函数。这个线程的功能就是在一个无限循环中不断地监听两个信号:SIGQUIT 和 SIGUSR1。当 SIGQUIT 被捕获的时候,就打印 jni 全局参考表的信息;当接收到 SIGUSR1 信号时,会强制进行不回收软引用的垃圾收集,此时,在程序中向进程发送 SIGUSR1 信号,就可使进程进行不回收软引用的垃圾收集。

在 Dalvik 中没有暴露使用管道的 API。因此,在虚拟机初始化时就将管道信号屏蔽,以阻止管道端因读关闭,写入失败而直接退出。

4 结束语

Android 是一个开源的嵌入式操作系统,Dalvik 则是在它之上的核心组件,基于 Dalvik 的 Java 应用打破了 Sun 对 Java 世界的垄断,并使之成为当今最为流行的嵌入式应用开发标准之一。虽然随着 Android 的发展,新版本的不断推出,尚无法预测它的发展方向,因而文中内容可能存在时间和空间上的局限性,但是,对 Dalvik 的深入探索是极具意义的一种尝试,能够给 Dalvik 的使用、移植、研究以及虚拟机设计提供建议和参考,这是撰写本文的主要目的。

参考文献:

[1] 姚昱旻,刘卫国. Android 的架构与应用开发研究[J]. 计算机系统应用,2008,17(11):110-112.

[2] Sun Microsystems Inc. Java ME Technology API Documentation[EB/OL]. 2007. ttp://java.sun.com/javame/reference/apis.jsp.

[3] Google Inc. Android Documentation[EB/OL]. 2007. http://code.google.com/intl/en/android/Documentation.html.

[4] Venners B. 深入 java 虚拟机[M]. 第 2 版. 曹晓钢,蒋 靖 译. 北京:机械工业出版社,2003.

[5] 探矽工作室. 深入嵌入式 Java 虚拟机[M]. 北京:中国铁道出版社,2003.

[6] 陈闵中. Linux 在嵌入式操作系统中的应用[J]. 同济大学学报,2001,5(14):564-566.

[7] 河 秦,王洪涛. Linux 2.6 内核标准教程[M]. 北京:人民邮电出版社,2008.

[8] 李正平,徐 超,陈军宁,等. Linux 2.6 内核进程调度分析[J]. 计算机技术与发展,2006,16(9):82-84.

[9] 毛德操,胡希明. Linux 内核源代码情景分析[M]. 杭州:浙江大学出版社,2001.

[10] Sun Microsystems Inc. Porting Guide Sun Java(TM) Wireless Client Software2.0 Java Platform, Micro Edition[EB/OL]. 2007-05. http://java.sun.com/javame/reference/apis.jsp.

(上接第 82 页)

2007(2):21-23.

[7] 郑 莉,董 渊,张瑞丰. C++语言程序设计[M]. 第 3 版. 北京:清华大学出版社,2004:188-190,293-300.

[8] msdn, VisualC++ DevelopeCentre, Library. Operator Overloading[EB/OL]. 2007-11. http://msdn.microsoft.com/zh-cn/library/5tk49fh2.aspx,2007.11.

[9] Priestley M. Practical Object-orientec Design with UML [M]. 2nd ed(影印版). 北京:清华大学出版社,2004:176-177.