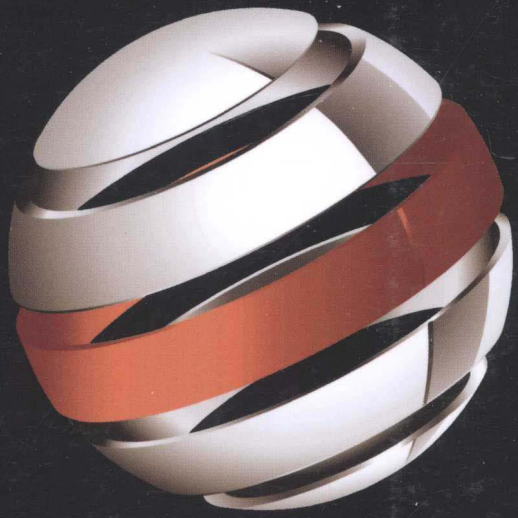
TURING 图灵程序设计丛书 移动开发系列

Apress®

- 全面剖析Android应用性能优化技巧
- 详尽的代码示例供你举一反三
- 开发优秀的Android应用必备指南



Pro Android Apps
Performance Optimization

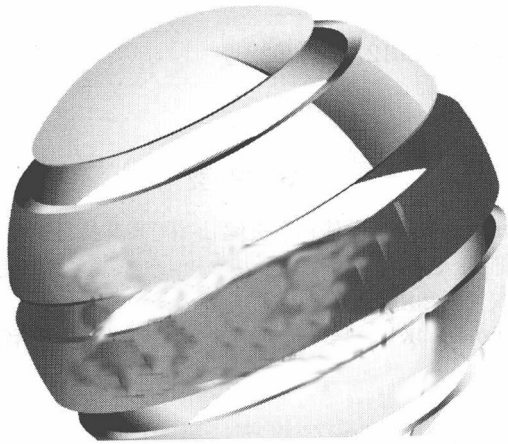
Android应用性能优化

[法] Hervé Guihot 著
白龙 译

人民邮电出版社
POSTS & TELECOM PRESS

更多资源请访问稀酷客(www.ckook.com)

TURING 图灵程序设计丛书 移动开发系列



Pro Android Apps
Performance Optimization

Android应用性能优化

[法] Hervé Guihot 著
白龙 译

人民邮电出版社
北 京

更多资源请访问稀酷客(www.ckook.com)

图书在版编目（CIP）数据

Android应用性能优化 /（法）埃尔韦（Hervé ,G.）著 ; 白龙译. -- 北京 : 人民邮电出版社, 2012.10
（图灵程序设计丛书）
书名原文: Pro Android Apps Performance Optimization
ISBN 978-7-115-27241-6

Ⅰ. ①A… Ⅱ. ①埃… ②白… Ⅲ. ①移动终端－应用程序－程序设计 Ⅳ. ①TN929.53

中国版本图书馆CIP数据核字(2012)第232522号

内 容 提 要

本书主要介绍如何调优 Android 应用，以使应用更健壮并提高其执行速度。内容包括用 Java、NDK 优化应用，充分利用内存以使性能最大化，尽最大可能节省电量，何时及如何使用多线程，如何使用基准问题测试代码，如何优化 OpenGL 代码和使用 Renderscript 等。

本书面向熟悉 Java 和 Android SDK 的想要进一步学习如何用本地代码优化应用性能的 Android 开发人员。

图灵程序设计丛书

Android应用性能优化

◆ 著 [法] Hervé Guihot
译 白 龙
责任编辑 毛倩倩
执行编辑 丁晓昀

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号
· 邮编 100061 电子邮件 315@ptpress.com.cn
网址 http://www.ptpress.com.cn
北京鑫正大印刷有限公司印刷

◆ 开本：800×1000 1/16
印张：15
字数：355千字 2012年10月第1版
印数：1-4 000册 2012年10月北京第1次印刷

著作权合同登记号 图字：01-2012-1734号

ISBN 978-7-115-27241-6

定价：49.00元

读者服务热线：(010)51095186转604 印装质量热线：(010)67129223

反盗版热线：(010)67171154

更多资源请访问稀酷客(www.ckook.com)

版权声明

Original English language edition, entitled *Pro Android Apps Performance Optimization* by Hervé Guihot , published by Apress, 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705 USA.

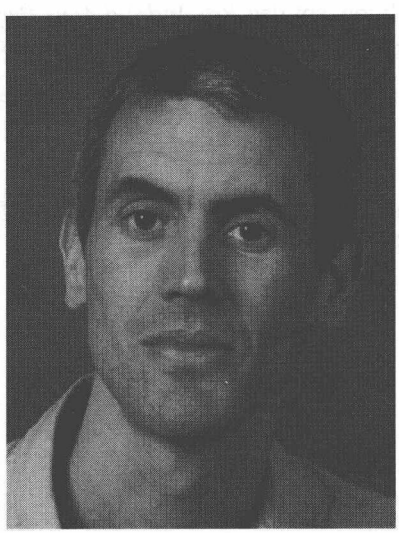
Copyright ©2012 by Hervé Guihot. Simplified Chinese-language edition copyright © 2012 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Apress L.P.授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

更多资源请访问稀酷客(www.ckook.com)

作者简介



Hervé Guihot 20年前通过Amstrad CPC464开始学计算机。尽管CPC464让他着迷绿色屏幕的设备（问问他用的啥手机），不过由于Android成为了流行的应用开发平台，并且是唯一能把Hervé的两个最爱（软件和甜点）搭配在一起的平台，因此Hervé开始了在Android上的工作。在互动与数字电视的领域里工作多年后，他现在关注的是让Android运行在更多的设备上，促使更多的开发者利用Android的强大功能。Hervé 目前在联发科技公司（MTK，www.mediatek.com）担任软件工程经理。联发科技公司是一家业界领先的提供无线通信和数字多媒体解决方案的芯片设计公司。他在布列塔尼的雷恩第一大学计算机与传播高等教育学院获取了电信工程学学士学位，有时你会发现他在18号大街和格雷罗大街的交叉口处的商店排队买法国长条泡芙（éclair）[①]。

① 旧金山的Tartine Bakeryhttp://www.yelp.com/biz/tartine-bakery-san-francisco-2。——编者注

更多资源请访问稀酷客(www.ckook.com)

译 者 序

人是一种两栖动物，同时生活在两个世界里；已有的和自己建造的世界——物质、生命和意识的世界以及符号的世界。

——奥尔德斯·赫胥黎

现在Android智能手机和平板电脑普及率很高，它们极大地改变了我们的生活，从摸索世界的孩童到安度晚年的老人，都在享受智能移动设备和移动网络带给他们的乐趣，享受那些织有梦想感觉的程序，这是另一个世界，但不是梦。这一切都是我们这些平凡而又有激情的程序员带来的，或许你会带来更多？这本书会帮上忙的。

现在市面上讲Android的书实在是汗牛充栋，但以优化为主题的几乎没有。如果只讲某部分优化，可能很多人都可以写出几篇文章，难得的是照顾到 Android的方方面面，这得需要技术的一定广度和深度。本书从性能的视角，带着大家重新审视了Android平台和SDK，把Android中的宝藏都给挖掘出来了。本书也不是大全式的著作，很多地方并没特别深入，不过日常应用基本够用。稍微有点缺憾的就是网络方面着墨不多，如果能再补充成为一章就好了， 当然这部分资料其实也挺多，大家可以自行寻找。稍微啰嗦点其他的，要想提高性能其实还得要懂点反编译知识，原因你懂的。

了解了这么多性能优化的手段，可能你会迫不及待地想大显身手，但作者也泼了点冷水，以下几个注意事项反复提醒了很多遍。

- 确定：请确定这个是问题；请确定这个是你（开发者）的问题。换句话说，搞清需求，有数据佐证，和其他同事（包括产品美工等）沟通确定过。
- 确保：请确保优化后进行充分测试，确保有效果，没问题。某种意义上讲应用程序就是靠评价吃饭的，切记不要搞砸。

这两点其实就是规范和流程的问题。在我看来性能优化其实大体上分为两类：一种是非常规使用的优化，另一种是常规使用的非劣化的方式（包括正确的算法，了解并合理使用各种辅助工具以及API等）。很多人往往在没有了解后一种的情况下就匆忙使用第一种方式，结果导致很多问题；其实后一种才是首先要做的，这步应该叫修正而不是优化。作者说优化像是门艺术，可我觉得其实就是经验，还没那么神秘，不管怎样它们都需要大量实践。

翻译本书其实还有些奇妙的缘分，在动笔翻译后才注意到作者居然是前公司的同事，记忆中

更多资源请访问稀酷客(www.ckook.com)

曾见过一面。感谢图灵社区这个大家一起学习的平台，感谢图灵的傅志红老师、李鑫老师在翻译过程中的大力帮助和指导。感谢同事郑珂帮忙审稿。感谢大胖老师帮忙推敲其中的几句译文，耐心地回答我在使用图灵社区 Markdown编辑器时碰到的菜鸟问题，他翻译的《番茄工作法图解》是工作的性能优化，值得阅读。感谢我的妻子钱茹，谢谢她一直以来的支持，还有我的儿子，我希望以后能够有更多的时间陪你们。

——白龙

更多资源请访问稀酷客(www.ckook.com)

前　言

Android已经融入了寻常百姓的生活中。当今世界，手机正从功能时代进化到智能时代，同时又诞生了令人爱不释手的平板电脑。目前，应用程序开发者的可选择平台主要就是Android和iOS。Android降低了甚至可以说是打破了移动开发的门槛，应用程序开发者编写Android应用程序只需要一台计算机就够了（当然还要有一些编程知识）。工具都是免费的，几乎每个人都能写出数百万人会用的应用。Android可以运行在各种设备上，从平板到电视。开发者关键要做的就是保证应用可以顺利地在这些设备上运行，而且比竞争对手的还好。对应用程序开发人员而言，Android开发的门槛已经很低了，你会发现，在许多情况下，自己不过是想要在日益增长的Android应用程序市场上分一杯羹而已。赖以谋生、实现明星梦，或者只是想使世界变得更美好……无论你编写程序所为何求，性能问题都是其中的关键。

要想阅读本书，最好能事先对Android应用程序开发基础有所了解，由此方能利用本书的妙诀良方让程序跑得更快。尽管借助Android工具和在线文档可以很容易地创建应用程序，但性能优化（有时简直更像是一门艺术而不是科学）却无定法可循。不管要优化的程序是已有的，还是从头编写的。本书的目的就是要帮你找到简便的优化方法，以便使程序在几乎所有Android设备上都能取得不错的性能。Android允许开发人员使用Java、C/C++，甚至汇编语言，所以，无论是更好地利用CPU特性，还是针对特定问题使用合适的编程语言，相信你可以用多种不同的方法来优化性能。

第1章　优化Java代码。毫无疑问，你的第一个Android应用程序基本都是用Java开发的。在这一章，你会了解到，选择算法要比实现算法更重要。你还将学习如何利用简单的技术（如缓存和减少内存分配）来极大地优化应用程序。此外，你还将学习让应用程序随时能够保持响应的方法，这是一个非常重要的性能指标。此外还将介绍高效使用数据库的方法。

第2章　更进一步（或者说更底层，得看谈话对象）领略Android NDK。尽管自从Android 2.2以后Java代码可以即时编译为机器码，但某些方法用C代码实现可以获得更棒的结果。NDK还可以让你轻松地将现有代码移植到Android，而无需用Java重写一遍。

第3章　底层的汇编语言。大多数应用程序开发很少用到汇编语言，但汇编语言能充分利用各个平台的专有指令集。虽然这会增加复杂度和维护成本，但却是非常强大的优化秘诀。汇编代码通常仅限于应用程序的某些特定部分，但不应忽略它的优点，仔细而有针对性的优化可以取得巨大成效。

第4章　探讨如何使用更少的内存来提高性能。除了学习在代码中使用较少内存的简单方法，

更多资源请访问稀酷客(www.ckook.com)

你还将了解到，由于CPU的设计方式，内存分配方式和内存访问也会对性能有直接影响。

第5章 如何在Android应用程序中使用多线程，以便保持随时响应，为越来越多可以同时运行多线程的Android设备提升性能。

第6章 测量应用程序性能的基础知识。除了可以用API来测量时间外，一些Android工具还可以方便地查看应用程序执行时间耗费的具体情况。

第7章 确保应用程序合理使用电量的一些方法。许多Android设备都由电池供电，因而节电非常重要，没人愿意使用过于耗电的应用。通过本章所述方法，可以不必牺牲Android程序的特性就能最大限度地减少功耗。

第8章 一些完善应用程序布局和优化OpenGL渲染的基本技术。

第9章 RenderScript。它是Honeycomb引入的一个相对较新的Android组件。RenderScript为性能而生，从首次发布以来已经有不少改进。本章介绍如何在应用程序中使用RenderScript，顺便学习RenderScript定义的许多API。

我希望你喜欢上这本书，并在里面找到许多有用的技巧。你会发现，很多技术不独适用于Android，还可以用在很多其他平台上，例如iOS。就个人而言，我偏好汇编语言，希望能借着Android平台的快速发展以及其对NDK汇编语言的支持，能使Android吸引到更多的开发者。至少，他们可以学到一门新技术。但是，良好的设计和算法常常可以满足所有性能优化的需求，这才是关键。祝你好运，我期待着你的Android应用程序！

更多资源请访问稀酷客(www.ckook.com)

致　　谢

感谢Apress的出版团队：Steve Anglin、Corbin Collins、Jim Markham、Jill Steinberg。与他们一起工作非常开心，我真心把他们推荐给所有的作者。

我还要感谢以下几位技术评审：Charles Cruz、Shane Kirk、Eric Neff。他们提供了宝贵的反馈意见，指出不少我看了十几遍都没发现的错误。

由于忙于写书，所以难得和一些朋友相聚，也要向你们致谢：Marcely、Mathieu、Marilen、Maurice、Katie、Maggie、Jean-René、Ruby、Greg、Aline、Amy、Gilles，我保证会弥补这一切。最后，我非常感激以下三位：Eddy Derick、Fabrice Bernard、Jean-Louis Gassée。脚趾受伤、行李箱摔坏，我欠你们的情。

更多资源请访问稀酷客(www.ckook.com)

目　录

更多资源请访问稀酷客(www.ckook.com)

更多资源请访问稀酷客(www.ckook.com)

更多资源请访问稀酷客(www.ckook.com)

第 1 章 Java 代码优化

许多 Android 应用开发者都有着丰富的 Java 开发经验。自从 1995 年问世以来，Java 已经成为一种非常流行的编程语言。虽然一些调查显示，在与其他语言（比如 Objective-C 或 C#）的竞争中，Java 已光芒不再，但它们还是不约而同地把 Java 排为第一流行的语言。当然，随着移动设备的销量超过个人电脑，以及 Android 平台的成功（2011 年 12 月平均每天激活 70 万部），Java 在今天的市场上扮演着比以往更重要的角色。

移动应用与 PC 应用在开发上有着明显的差异。如今的便携式设备已经很强大了，但在性能方面还是落后于个人电脑。例如，一些基准测试显示，四核 Intel Core i7 处理器的运行速度大约是三星 Galaxy Tab 10.1 中的双核 Nvidia Tegra 2 处理器的 20 倍。

注意 基准测试结果不能全信，因为它们往往只测量系统的一部分，不代表典型的使用场景。

本章将介绍一些确保 Java 应用在 Android 设备上获得高性能的办法（无论其是否运行于最新版本的 Android）。我们先来看看 Android 是如何来执行代码的，然后再品评几个著名数列代码的优化技巧，包括如何利用最新的 Android API。最后再介绍几个提高应用响应速度和高效使用数据库的技巧。

在深入学习之前，你应该意识到代码优化不是应用开发的首要任务。提供良好的用户体验并专注于代码的可维护性，这才是你首要任务。事实上，代码优化应该最后才做，甚至完全可能不用去做。不过，良好的优化可以使程序性能直接达到一个可接受的水平，因而也就无需再重新审视代码中的缺陷并耗费更多的精力去解决它们。

1.1 Android 如何执行代码

Android 开发者使用 Java，不过 Android 平台不用 Java 虚拟机（VM）来执行代码，而是把应用编译成 Dalvik 字节码，使用 Dalvik 虚拟机来执行。Java 代码仍然编译成 Java 字节码，但随后 Java 字节码会被 dex 编译器（dx，SDK 工具）编译成 Dalvik 字节码。最终，应用只包含 Dalvik 字节码，而不是 Java 字节码。

例如，代码清单 1-1 是包含类定义的计算斐波那契数列第 n 项的实现。斐波那契数列的定义

更多资源请访问稀酷客(www.ckook.com)

如下：

$F_0 = 0$

$F_1 = 1$

$F_n = F_{n-2} + F_{n-1}$ (n>1)

代码清单 1-1　简单的斐波那契数列递归实现

```
public class Fibonacci {
    public static long computeRecursively (int n)
    {
        if (n > 1) return computeRecursively(n-2) + computeRecursively(n-1);
        return n;
    }
}
```

注意　微小优化：当 n 等于 0 或 1 时直接返回 n，而不是另加一个 if 语句来检查 n 是否等于 0 或 1。

Android 应用也称为 apk，因为应用被打包成带有 apk 扩展名（例如，APress.apk）的文件，这是一个简单的压缩文件。classes.dex 文件就在这个压缩文件里，它包含了应用的字节码。Android 的工具包中有名为 dexdump 的工具，可以把 classes.dex 中的二进制代码转化为使人易读的格式。

提示　apk 文件只是个简单的 ZIP 压缩文件，可以使用常见的压缩工具（如 WinZip 或 7-Zip）来查看 apk 文件的内容。

代码清单1-2显示了对应的 Dalvik 字节码。

代码清单 1-2　Fibonacci.computeRecursively 的 Dalvik 字节码的可读格式

```
002548:                        |[002548] com.apress.proandroid.Fibonacci.computeRecursively:(I)J
002558: 1212                   |0000: const/4 v2, #int 1 // #1
00255a: 3724 1100              |0001: if-le v4, v2, 0012 // +0011
00255e: 1220                   |0003: const/4 v0, #int 2 // #2
002560: 9100 0400              |0004: sub-int v0, v4, v0
002564: 7110 3d00 0000         |0006: invoke-static {v0},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
00256a: 0b00                   |0009: move-result-wide v0
00256c: 9102 0402              |000a: sub-int v2, v4, v2
002570: 7110 3d00 0200         |000c: invoke-static {v2},
    Lcom/apress/proandroid/Fibonacci;.computeRecursively:(I)J
002576: 0b02                   |000f: move-result-wide v2
002578: bb20                   |0010: add-long/2addr v0, v2
00257a: 1000                   |0011: return-wide v0
00257c: 8140                   |0012: int-to-long v0, v4
00257e: 28fe                   |0013: goto 0011 // -0002
```

更多资源请访问稀酷客(www.ckook.com)

在“|”左边的本地代码中，除了第一行（用于显示方法名），每行冒号右边是一个或多个16位的字节码单元[①]，冒号左边的数字指定了它们在文件中的绝对位置。“|”右边的可读格式中，冒号左边是绝对位置转换为方法内的相对位置或标签，冒号右边是操作码助记符及后面不定个数的参数。例如，地址0x00255a的两字节码组合3724 1100翻译为`if-le v4, v2, 0012 // +0011`，意思是说“如果虚拟寄存器v4的值小于等于虚拟寄存器v2的值，就跳转到标签0x0012，相当于跳过17（十六进制的11）个字节码单元”。术语“虚拟寄存器”是指实际上非真实的硬件寄存器，也就是Dalvik虚拟机使用的寄存器。[②]

通常情况下，你不必看应用的字节码。在平台是Android 2.2（代号Froyo）和更高版本的情况下尤其如此，因为在Android 2.2中引入了实时（JIT）编译器。Dalvik JIT编译器把Dalvik字节码编译成本地代码，这可以明显加快执行速度。JIT编译器（有时简称JIT）可以显著提高性能，因为：

- 本地代码直接由CPU执行，而不必由虚拟机解释执行；
- 本地代码可以为特定架构予以优化。

谷歌的基准测试显示，Android 2.2的代码执行速度比Android 2.1快2到5倍。虽说代码的具体功能会对结果产生很大影响，但可以肯定的是，使用Android 2.2及更高版本会显著提升速度。

对于无JIT的Android 2.1或更早的版本而言，优化策略的选用可能会因此受到很大影响。如果打算针对运行Android 1.5（代号Cupcake）、1.6（代号Donut），或2.1（代号éclair）的设备开发，你要先仔细地审查应用在这些环境下需要提供哪些功能。此外，这些运行Android早期版本的旧设备是没新设备强劲的。尽管运行Android 2.1和更早版本的设备所占的市场份额正在萎缩，但直到2011年12月，其数量仍占大约12%。可选的策略有3条：

- 不予优化，因为应用在这些旧设备上运行得相当缓慢；
- 限制应用中Android API等级为最低8级，让它只能安装在Android 2.2或更高版本上；
- 即使没有JIT编译器，也要针对旧设备优化，给用户以舒畅的体验。也就是说禁掉那些非常耗CPU资源的功能。

提示 *在应用的manifest配置里可以用`Android:vmSafeMode`启用或禁用JIT编译器。默认是启用的（如果平台有JIT）。这个属性是Android 2.2引入的。*

现在可以在真实平台上运行代码了，看看它是如何执行的。如果你熟悉递归和斐波那契数列，可能已经猜到，这段代码运行得不会很快。没错！在三星Galaxy Tab 10.1上，计算第30项斐波那契数列花了约370毫秒。禁用JIT编译器之后需要大约440毫秒。如果把这个功能加到计算器程序里，用户会感觉难以忍受，因为结果不能“马上”计算出来。从用户的角度来看，如果可以在100毫秒或更短的时间内计算完成，那就是瞬时计算。这样的响应时间保证了顺畅的用户体

① java指令是16位的，可以参考jvm和dex指令集。——译者注

② sun的jvm是基于栈的虚拟机，而dalvik是基于寄存器的虚拟机。——译者注

更多资源请访问稀酷客(www.ckook.com)

验，这才是我们要达到的目标。

1.2　优化斐波那契数列

我们要做的首次优化是消除一个方法调用，如代码清单 1-3 所示。由于这是递归实现，去掉方法中的一个调用就会大大减少调用的总数。例如，computeRecursively (30) 产生了 2 692 537 次调用，而 computeRecursivelyWithLoop (30) 产生的调用“只有”1 346 269 次。然而，这样优化过的性能还是无法接受，因为前面我们把响应时间的标准定为 100 毫秒或者更少，而 computeRecursivelyWithLoop (30) 却花了 270 毫秒。

代码清单 1-3　优化递归实现斐波那契数列

```
public class Fibonacci {
    public static long computeRecursivelyWithLoop (int n)
       {
        if (n > 1) {
            long result = 1;
            do {
                result += computeRecursivelyWithLoop(n-2);
                n--;
            } while (n > 1);
            return result;
        }
        return n;
    }
}
```

注意　这不是一个真正的尾递归优化。

1.2.1　从递归到迭代

第二次优化会换成迭代实现。递归算法在开发者当中的名声不太好，尤其是在没多少内存可用的嵌入式系统开发者中，主要是因为递归算法往往要消耗大量栈空间。正如我们刚才看到的，它产生了过多的方法调用。即使性能尚可，递归算法也有可能导致栈溢出，让应用崩溃。因此应尽量用迭代实现。代码清单 1-4 是斐波那契数列的迭代实现。

代码清单 1-4　斐波那契数列的迭代实现

```
public class Fibonacci {
    public static long computeIteratively (int n)
       {
        if (n > 1) {
            long a = 0, b = 1;
            do {
                long tmp = b;
                b += a;
```

更多资源请访问稀酷客(www.ckook.com)

```
                a = tmp;
            } while (--n > 1);
            return b;
        }
        return n;
    }
}
```

由于斐波那契数列的第 n 项其实就是前两项之和，所以一个简单的循环就可以搞定。与递归算法相比，这种迭代算法的复杂性也大大降低，因为它是线性的。其性能也更好，computeIteratively (30) 花了不到 1 毫秒。由于其线性特性，你可以用这种算法来计算大于 30 的项。例如，computeIteratively (50000)，只要 2 毫秒就能返回结果。根据这个推测，你应该能猜出 computeIteratively (500000) 大概会花 20 至 30 毫秒。

虽然这样已经达标了，但相同的算法稍加修改后还可以更快，如代码清单 1-5 所示。这个新版本每次迭代计算两项，迭代总数少了一半。因为原算法的迭代次数可能是奇数，所以 a 和 b 的初始值要做相应的修改：该数列开始时如果 n 是奇数，则 a= 0，b = 1；如果 n 是偶数，则 a = 1，b= 1 (Fib(2)= 1)。

代码清单 1-5 修改后的斐波那契数列的迭代实现

```
public class Fibonacci {
    public static long computeIterativelyFaster (int n)
    {
        if (n > 1) {
            long a, b = 1;
            n--;
            a = n & 1;
            n /= 2;
            while (n-- > 0) {
                a += b;
                b += a;
            }
            return b;
        }
        return n;
    }
}
```

结果表明此次修改的迭代版本速度比旧版本快了一倍。

虽然这些迭代实现速度很快，但它们有个大问题：不会返回正确结果。问题在于返回值是 long 型，它只有 64 位。在有符号的 64 位值范围内，可容纳的最大的斐波那契数是 7 540 113 804 746 346 429，或者说是斐波那契数列第 92 项。虽然这些方法在计算超过 92 项时没有让应用崩溃，但是因为出现溢出，结果却是错误的，斐波那契数列第 93 项会变成负的！递归实现实际上有同样的限制，但得耐心等待才能得到最终的结论。

注意 在 Java 所有基本类型（boolean 除外）中，long 是 64 位、int 是 32 位、short 是 16 位。所有整数类型都是有符号的。

更多资源请访问稀酷客(www.ckook.com)

1.2.2 BigInteger

Java 提供了恰当的类来解决这个溢出问题：java.math.BigInteger。BigInteger 对象可以容纳任意大小的有符号整数，类定义了所有基本的数学运算（除了一些不太常用的）。代码清单 1-6 是 computeIterativelyFaster 的 BigInteger 版本。

提示　java.math 包除了 BigInteger 还定义了 BigDecimal，而 java.lang.Math 提供了数学常数和运算函数。如果应用不需要双精度（double precision），使用 Android 的 FloatMath 性能会更好（虽然不同平台的效果不同）。

代码清单 1-6　BigInteger 版本的 Fibonacci.computeIterativelyFaster

```
public class Fibonacci {
    public static BigInteger computeIterativelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            BigInteger a, b = BigInteger.ONE;
            n--;
            a = BigInteger.valueOf(n & 1);
            n /= 2;
            while (n-- > 0) {
                a = a.add(b);
                b = b.add(a);
            }
            return b;
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE;
    }
}
```

这个实现保证正确，不再会溢出。但它又出现了新问题，速度再一次降了下来，变得相当慢：计算 computeIterativelyFasterUsingBigInteger (50000) 花了 1.3 秒。表现平平的原因有以下三点：

- BigInteger 是不可变的；
- BigInteger 使用 BigInt 和本地代码实现；
- 数字越大，相加运算花费的时间也越长。

由于 BigInteger 是不可变的，我们必须写 a = a.add(b)，而不是简单地用 a.add(b)，很多人误以为 a.add(b)相当于 a += b，但实际上它等价于 a + b。因此，我们必须写成 a = a.add(b)，把结果值赋给 a。这里有个小细节是非常重要的：a.add(b)会创建一个新的 BigInteger 对象来持有额外的值。

由于目前 BigInteger 的内部实现，每分配一个 BigInteger 对象就会另外创建一个 BigInt 对象。在执行 computeIterativelyFasterUsingBigInteger 过程中，要分配两倍的对象：调用 computeIterativelyFasterUsingBigInteger(50000)时约创建了 100 000 个对象（除了其中的 1 个对象外，其他所有对象立刻变成等待回收的垃圾）。此外，BigInt 使用本地代码，而从 Java 使用 JNI 调用本地代码会产生一定的开销。

更多资源请访问稀酷客(www.ckook.com)

第三个原因是指非常大的数字不适合放在一个 64 位 long 型值中。例如，第 50 000 个斐波那契数为 347 111 位长。

注意 未来 Android 版本的 BigInteger 内部实现（BigInteger.java）可能会改变。事实上，任何类的内部实现都有可能改变。

基于性能方面的考虑，在代码的关键路径上要尽可能避免内存分配。无奈的是，有些情况下分配是不可避免的。例如，使用不可变对象（如 BigInteger）。下一种优化方式则侧重于通过改进算法来减少分配数量。基于斐波那契 Q-矩阵，我们有以下公式：

$$F_{2n-1} = F_n^2 + F_{n-1}^2$$

$$F_{2n} = (2F_{n-1} + F_n) * F_n$$

这可以用 BigInteger 实现（保证正确的结果），如代码清单 1-7 所示。

代码清单 1-7 斐波那契数列使用 BigInteger 的快速归实现

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigInteger (int n)
    {
        if (n > 1) {
            int m = (n / 2) + (n & 1); // 较为晦涩，是否该有个更好的注释?
            BigInteger fM = computeRecursivelyFasterUsingBigInteger(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigInteger(m - 1);
            if ((n & 1) == 1) {
                // F(m)^2 + F(m-1)^2
                return fM.pow(2).add(fM_1.pow(2)); // 创建了 3 个 BigInteger 对象
            } else {
                // (2*F(m-1) + F(m)) * F(m)
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // 创建了 3 个对象
            }
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE; // 没有创建 BigInteger
    }
    public static long computeRecursivelyFasterUsingBigIntegerAllocations(int n) {
        long allocations = 0;
        if (n > 1) {
            int m = (n / 2) + (n & 1);
            allocations += computeRecursivelyFasterUsingBigIntegerAllocations(m);
            allocations +=  computeRecursivelyFasterUsingBigIntegerAllocations(m - 1);
            // 创建的 BigInteger 对象多于 3 个
            allocations += 3;
        }
        return allocations; // 当调用 computeRecursivelyFasterUsingBigInteger(n)时，创建 BigInteger
                            对象的近似数目
    }
}
```

调用 computeRecursivelyFasterUsingBigInteger (50000) 花费了 1.6 秒左右，这表明最新实

更多资源请访问稀酷客(www.ckook.com)

现实际上是慢于已有的最快迭代实现。拖慢速度的罪魁祸首是要分配大约 200 000 个对象（几乎立即标记为等待回收的垃圾）。

注意　实际分配数量比 computeRecursivelyFasterUsingBigIntegerAllocations 返回的估算值少。因为 BigInteger 的实现使用了预分配对象，BigInteger.ZERO、BigInteger.ONE 或 BigInteger.TEN，有些运算没必要分配一个新对象。这需要在 Android 的 BigInteger 实现一探究竟，看看它到底创建了多少个对象。

尽管这个实现慢了点，但它毕竟是朝正确的方向迈出了一步。值得注意的是，即使我们需要使用 BigInteger 确保正确性，也不必用 BigInteger 计算所有 n 的值。既然基本类型 long 可容纳小于等于 92 项的结果，我们可以稍微修改递归实现，混合 BigInteger 和基本类型，如代码清单 1-8 所示。

代码清单 1-8　斐波那契数列使用 BigInteger 和基本类型 long 的快速递归实现

```
public class Fibonacci {
    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndPrimitive(int n)
    {
        if (n > 92) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndPrimitive(m - 1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM); // shiftLeft(1)乘以 2
            }
        }
        return BigInteger.valueOf(computeIterativelyFaster(n));
    }
    private static long computeIterativelyFaster(int n)
    {
        // 见代码清单 1-5 实现
    }
}
```

调用 computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)花了约 73 毫秒，创建了约 11 000 个对象：略微修改下算法，速度就快了约 20 倍，创建对象数则仅是原来的 1/20，很惊人吧！通过减少创建对象的数量，进一步改善性能是可行的，如代码清单 1-9 所示。Fibonacci 类首次加载时，先快速生成预先计算的结果，这些结果以后就可以直接使用。

代码清单 1-9　斐波那契数列使用 BigInteger 和预先计算结果的快速递归实现

```
public class Fibonacci {
    static final int PRECOMPUTED_SIZE= 512;
    static BigInteger PRECOMPUTED[] = new BigInteger[PRECOMPUTED_SIZE];
```

更多资源请访问稀酷客(www.ckook.com)

```
    static {
        PRECOMPUTED[0] = BigInteger.ZERO;
        PRECOMPUTED[1] = BigInteger.ONE;
        for (int i = 2; i < PRECOMPUTED_SIZE; i++) {
            PRECOMPUTED[i] = PRECOMPUTED[i-1].add(PRECOMPUTED[i-2]);
        }
    }
    public static BigInteger computeRecursivelyFasterUsingBigIntegerAndTable(int n)
    {
        if (n > PRECOMPUTED_SIZE - 1) {
            int m = (n / 2) + (n & 1);
            BigInteger fM = computeRecursivelyFasterUsingBigIntegerAndTable (m);
            BigInteger fM_1 = computeRecursivelyFasterUsingBigIntegerAndTable (m - 1);
            if ((n & 1) == 1) {
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                return fM_1.shiftLeft(1).add(fM).multiply(fM);
            }
        }
        return PRECOMPUTED[n];
    }
}
```

这个实现的性能取决于 PRECOMPUTED_SIZE：更大就更快。然而，内存使用量可能会成为新问题。由于许多 BigInteger 对象创建后保留在内存中，只要加载了 Fibonacci 类，它们就会占用内存。我们可以合并代码清单 1-8 和代码清单 1-9 的实现，联合使用预计算和基本类型。例如，0 至 92 项可以使用 computeIterativelyFaster，93 至 127 项使用预先计算结果，其他项使用递归计算。作为开发人员，你有责任选用最恰当的实现，它不一定是最快的。你要权衡各种因素：

- 应用是针对哪些设备和 Android 版本；
- 资源（人力和时间）。

你可能已经注意到，优化往往使源代码更难于阅读、理解和维护，有时几个星期或几个月后你都认不出自己的代码了。出于这个原因，关键是要仔细想好，你真正需要怎样的优化以及这些优化究竟会对开发产生何种影响（短期或长期的）。强烈建议你先实现一个能运行的解决方案，然后再考虑优化[①]（注意备份之前能运行的版本）。最终，你可能会意识到优化是不必要的，这就节省了很多时间。另外，有些代码不易被水平一般的人所理解，注意加上注释，同事会因此感激你。另外，当你在被旧代码搞蒙时，注释也能勾起你的回忆。我在代码清单 1-7 中的少量注释就是例证。

注意 所有实现忽略了一个事实——n 可以是负数。我是特意这样做的。不过，你的代码（至少在所有的公共 API 中）应该在适当时抛出 IllegalArgumentException 异常。

① 你可以使用版本控制，高德纳名言“过早的优化是噩梦之源”。——译者注

更多资源请访问稀酷客(www.ckook.com)

1.3　缓存结果

如果计算代价过高，最好把过去的结果缓存起来，下次就可以很快取出来。使用缓存很简单，通常可以转化为代码清单 1-10 所示的伪代码。

代码清单 1-10　使用缓存

```
result = cache.get(n); // 输入参数 n 作为键
if (result == null) {
    // 如果在缓存中没有 result 值，就计算出来填进去
    result = computeResult(n);
    cache.put(n, result); // n 作为键，result 是值
}
return result;
```

快速递归算法计算斐波那契项包含许多重复计算，可以通过将函数计算结果缓存[①]（memoization）起来的方法来减少这些重复计算。例如，计算第 50 000 项时需要计算第 25 000 和第 24 999 项。计算 25 000 项时需要计算第 12 500 项和第 12 499 项，而计算第 24 999 项还需要 12 500 项与 12 499 项！代码清单 1-11 是个更好的实现，它使用了缓存。

如果你熟悉 Java，你可能打算使用一个 HashMap 充当缓存，它可以胜任这项工作。不过，Android 定义了 SparseArray 类，当键是整数时，它比 HashMap 效率更高。因为 HashMap 使用的是 java.lang.Integer 对象，而 SparseArray 使用的是基本类型 int。因此使用 HashMap 会创建很多 Integer 对象，而使用 SparseArray 则可以避免。

代码清单 1-11　使用 BigInteger 的快速递归实现，用了基本类型 long 和缓存

```
public class Fibonacci {
    public static BigInteger computeRecursivelyWithCache (int n) {
        SparseArray<BigInteger> cache = new SparseArray<BigInteger>();
        return computeRecursivelyWithCache(n, cache);
    }
    private static BigInteger computeRecursivelyWithCache (int n, SparseArray<BigInteger> cache) {
        if (n > 92) {
            BigInteger fN = cache.get(n);
            if (fN == null) {
                int m = (n / 2) + (n & 1);
                BigInteger fM = computeRecursivelyWithCache(m, cache);
                BigInteger fM_1 = computeRecursivelyWithCache(m - 1, cache);
                if ((n & 1) == 1) {
                    fN = fM.pow(2).add(fM_1.pow(2));
                } else {
                    fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
                }
                cache.put(n, fN);
            }
            return fN;
```

① 请参见 http://en.wikipedia.org/wiki/Memoization。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
        }
        return BigInteger.valueOf(iterativeFaster(n));
    }

    private static long iterativeFaster (int n) {
        //见代码清单 1-5 的实现
    }
}
```

测量结果表明，computeRecursivelyWithCache(50000)用了约 20 毫秒，或者说比computeRecursivelyFasterUsingBigIntegerAndPrimitive(50000)快了约 50 毫秒。显然，差异随着 n 的增长而加剧，当 n 等于 200 000 时两种方法分别用时 50 毫秒和 330 毫秒。

因为创建了非常少的 BigInteger 对象，BigInteger 的不可变性在使用缓存时就不是什么大问题了。但请记住，当计算 F_n时仍创建了三个 BigInteger 对象（其中两个存在时间很短），所以使用可变大整数仍会提高性能。

尽管使用 HashMap 代替 SparseArray 会慢一些，但这样的好处是可以让代码不依赖 Android，也就是说，你可以在非 Android 的环境（无 SparseArray）使用完全相同的代码。

注意 Android 定义了多种类型的稀疏数组（sparse array）：SparseArray（键为整数，值为对象）、SparseBooleanArray（键为整数，值为 boolean）和 SparseIntArray（键为整数，值为整数）。

1.3.1 android.util.LruCache

值得一提的另一个类是 android.util.LruCache<K, V>，这个类是 Android 3.1（代号 Honeycomb MR1）引入的，可以在创建时定义缓存的最大长度。另外，还可以通过覆写 sizeof()方法改变每个缓存条目计算大小的方式。因为 android.util.LruCache 只能在 Android 3.1 及更高版本上使用，如果针对版本低于 3.1 的 Android 设备，则仍然必须使用不同的类来实现自己的应用缓存。由于目前的 Android 3.1 设备占有率不高，这种情况很有可能出现。替代方案是继承 java.util.LinkedHashMap 覆写 removeEldestEntry。LRU（Least Recently Used）缓存先丢弃最近最少使用的项目。在某些应用中，可能需要完全相反的策略，即丢弃缓存中最近最多使用的项目。Android 现在没有这种 MruCache 类，考虑到 MRU 缓存不常用，这并不奇怪。

当然，缓存是用来存储信息而不是计算结果的。通常，缓存被用来存储下载数据（如图片），但仍需严格控制使用量。例如，覆写 LruCache 的的 sizeof 方法不能简单地以限制缓存中的条目数为准则。尽管这里简要地讨论了 LRU 和 MRU 策略，你仍可以在缓存中使用不同的替代策略，最大限度地提高缓存命中率。例如，缓存可以先丢弃那些重建开销很小的项目，或者干脆随机丢弃项目。请以务实的态度设计缓存。简单的替换策略（如 LRU）可以产生很好的效果，把手上资源留给更重要的问题。

我们用了几个不同的技术优化斐波那契数列的计算。虽然每种技术都有其优点，却没有一个

更多资源请访问稀酷客(www.ckook.com)

实现是最佳选择。往往最好的结果是结合多种不同的技术，而不是只依赖于其中之一。例如，更快的实现可以使用预计算、缓存机制，甚至采用不同的公式。（提示：当 n 是 4 的倍数，会发生什么？）怎样在 100 毫秒内计算出 $F_{Integer.MAX\_VALUE}$?

1.4　API 等级

上述 LruCache 类是一个很好的例子，它让你知道需要了解目标平台的 API 等级。Android 大约每 6 个月发布一个新版本，随之发布的新 API 也只适用于该版本。试图调用不存在的 API 将导致崩溃，不仅会让用户失望，开发者也会感到羞愧。例如，在 Android 1.5 设备上调用 `Log.wtf(TAG, "really?")` 会使应用崩溃，因为 Log.wtf 是 Android 2.2（API 等级 8）引入的，这是个可怕的错误。表 1-1 列出了不同 Android 版本的性能改进情况。

表1-1　Android版本

API等级	版　本	代　号	重大的性能改进
1	1.0	Base	
2	1.1	Base 1.1	
3	1.5	Cupcake	相机启动时间、图像采集时间、更快的GPS定位、支持NDK
4	1.6	Donut	
5	2.0	éclair	图形
6	2.0.1	éclair 0.1	
7	2.1	éclair MR1	
8	2.2	Froyo	V8 Javascript引擎（浏览器）、JIT编译器、内存管理
9	2.3.0、2.3.1、2.3.2	Gingerbread	并发垃圾收集器、事件分布、更好的OpenGL驱动程序
10	2.3.3 2.3.4	Gingerbread MR1	
11	3.0	Honeycomb	Renderscript、动画、2D图形硬件加速、多核支持
12	3.1	Honeycomb MR1	LruCache、废弃部分硬件加速的view、新Bitmap.setHasAlpha() API
13	3.2	Honeycomb MR2	
14	4.0	Ice Cream Sandwich	Media效果（变换滤镜），2D图形硬件加速（必需）

不过，支持某种目标设备的决策依据通常并不是要使用的 API，而是在于打算进入什么样的市场。例如，如果你的目标主要是平板电脑，而不是手机，就可以只考虑 Honeycomb。这样做，会限制应用只能有一小部分 Android 用户，因为 Honeycomb 截至 2011 年 12 月的占有率只有 2.4% 左右，而且并不是所有平板电脑都支持 Honeycomb。（例如，Barnes & Noble 的 Nook 采用的是 Android 2.2，而 Amazon 的 Kindle Fire 使用的是 Android 2.3。）因此，支持较旧的 Android 版本仍有意义。

Android 团队知道这个问题，它们发布了 Android 兼容包，可以通过 SDK 更新。这个软件包有一个静态库，包含了一些 Android 3.0 引入的新 API，也就是分化的 API。然而，此兼容包只包

更多资源请访问稀酷客(www.ckook.com)

含 Honeycomb 分化的 API，不支持其他 API。这样的兼容包是例外情况，而不是通则。通常情况下，在特定的 API 等级引入的 API 是不可以在较低 API 等级上用的，开发人员要认真考虑 API 等级。

你可以使用 Build.VERSION.SDK_INT 获得 Android 平台的 API 等级。具有讽刺意味的是，这个字段是在 Android 1.6（API 等级 4）引入的，所以试图获取版本号，也会导致程序在 Android 1.5 或更早版本下崩溃。另一种选择是使用 Build.VERSION.SDK，它是 API 等级 1 引入的。但这一字段现在已经废弃，版本字符串也没有归档（虽然很容易理解它们是如何被创建的）。

提示 可以用反射来检查是否存在 SDK_INT 字段（也就是判断该平台是不是 Android 1.6 或更高版本）。参见 Class.forName("Android.os.Build$VERSION").getField("SDK")。

应用中的 manifest 清单文件应使用<uses-sdk >元素指定以下两个重要的信息：

- 应用所需的最低 API 等级（android:minSdkVersion）；
- 应用期望的 API 等级（android:targetSdkVersion）。

也可以限制最高的 API 等级（android:maxSdkVersion），但不推荐使用此属性。指定 maxSdkVersion 可能导致 Android 更新后，应用自动被卸载。[①]所针对平台的 API 等级应是你的应用实际测试通过的等级。

默认情况下，最小的 API 等级设置为 1，即应用兼容所有 Android 版本。指定 API 等级大于 1 可防止应用安装到旧设备上。例如，Android minSdkVersion ="4"可确保使用 Build.VERSION.SDK_INT 没有任何崩溃的风险。最低 API 等级并不是一定要指定为应用可以用的最高 API 等级，只要确保要调用的特定 API 确实存在即可，如代码清单 1-12 所示。

代码清单 1-12 调用 Honeycomb 中的 SparseArray 方法（API 等级 11）

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
    sparseArray.removeAt(1); // API 等级 11 及以上
} else {
    int key = sparseArray.keyAt(1); // 默认实现慢一些
    sparseArray.remove(key);
}
```

这类代码很常用，它既可以使用最适当的 API 来获取最好性能，也可以在旧的平台上（可能使用了较慢的 API）正常运行。

Android 也使用这些属性做其他的事情，比如决定应用是否应在屏幕兼容模式运行。如果 minSdkVersion 设置为 3 或更低，targetSdkVersion 并没有设置为 4 或更高，应用将在屏幕兼容模式运行。这样应用就不能在平板电脑上全屏显示，会使它很难使用。平板电脑只是最近才流行开来，很多应用还没有为它更新，所以发现程序在大屏幕上显示不正确并不鲜见。

① 升级后的系统版本大于应用指定的最高版本。——译者注

更多资源请访问稀酷客(www.ckook.com)

注意　Android Market 使用 minSdkVersion 和 maxSdkVersion 属性来筛选可供特定设备下载安装的应用，其他属性也同样可以用来筛选。此外，Android 定义了两个版本的屏幕兼容模式，它们的行为有所不同。请参阅 http://d.android.com/guide 页面中“Supporting Multiple Screens”中的完整描述。

如果不想用类似代码清单 1-12 所示的代码来检查版本号，也可以直接使用反射（reflection）来确认平台上是否有特定方法。虽然这是一种更清晰、更安全的实现，不过反射会使代码变慢，因此，在性能至关重要的地方应尽量避免使用反射。替代的办法是在静态初始化块里调用 `Class.forName()`和 `Class.getMethod()`确认指定方法是否存在，在性能要求高的地方只调用 `Method.invoke()`就好了。

1.4.1　版本分化

Android 的许多版本（到目前为止最高 API 等级为 14[①]）割裂了目标市场，这使得出现了越来越多的类似代码清单 1-12 的代码。不过在现实中，主流设备上运行的 Android 版本并不多。截至 2011 年 12 月，连接到 Android Market 的设备中，2.*x* 版本超过 95%。尽管运行 Android 1.6 及更早版本的设备也还有，不过不优化它们以节约资源也是合情合理的。

运行 Android 的设备数量越来越多，列出目前接近 200 个移动电话型号[②]，其中美国就有 80 个。虽然列出的设备都是手机或平板电脑，它们仍然在许多方面不同：屏幕分辨率、实体物理键盘、硬件图形加速器以及处理器。支持各种配置，甚至只支持其中一个子集，对应用开发来说也是挑战。因此，要充分了解目标市场的状况，以便把努力的重点放在重要的功能和优化上。

注意　并没有列出所有的 Android 设备，甚至忽略了一些国家，例如印度及其使用的运行 Android 2.2 的双卡 Spice MI270。

谷歌电视设备（2010 年首次在美国由罗技和索尼公司发布）的技术与手机或平板电脑没什么大的不同，主要是在于人机交互方式的差异。如果要支持这些电视设备，主要难点之一要了解应用在电视上的使用方式。例如，应用可以在电视上提供更多的社交体验，游戏可以支持同时多人模式，而这功能对手机来说却没有多大用处。

1.5　数据结构

不同的斐波那契数列实现证明，好的算法和数据结构是实现快速应用的关键。Android 和 Java 定义了许多数据结构，你需要有丰富的知识，快速地为任务选择适用的数据结构。选择适当的数

① 翻译本书时又有更新。——译者注

② 翻译本书时数量已经增加。——译者注

更多资源请访问稀酷客(www.ckook.com)

据结构是最先要解决的事项。

java.util 包中最常见的数据结构，如图 1-1 所示。

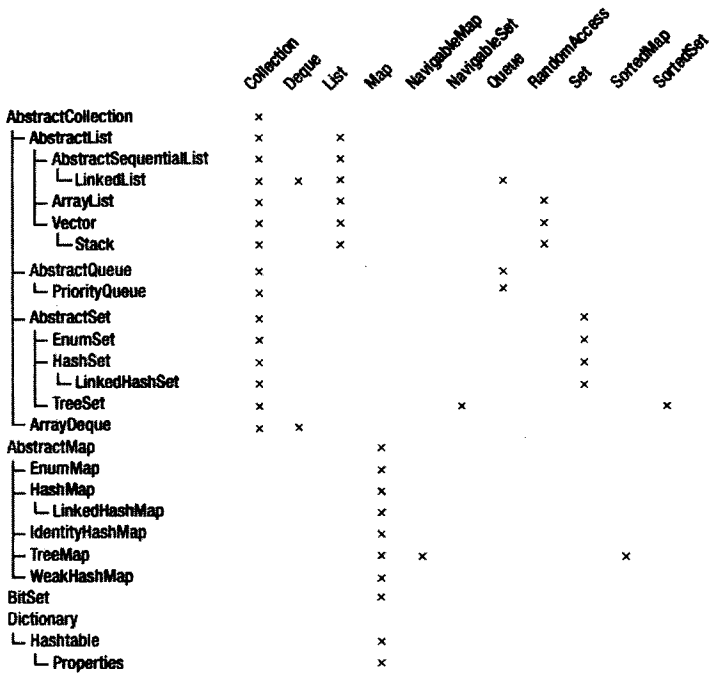


图 1-1 java.util 包中的数据结构

Android 在这些数据结构中增加了自身的一些实现，一般是为了解决性能问题。

- LruCache
- SparseArray
- SparseBooleanArray
- SparseIntArray
- Pair

注意 Java 还定义了 Arrays 类和 Collections 类。这两个类只包含静态方法，分别操作数组和集合。例如，使用 Arrays.sort 对数组排序，使用 Arrays.binarySearch 在有序数组中搜索值。

虽然上述斐波那契数列实现有一个用了内部缓存（基于稀疏数组），但是缓存只是暂时的，计算出最终结果后立即变成待回收的垃圾。你也可以使用 LruCache 保存最终结果，如代码清单 1-13 所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单 1-13　使用 LruCache 存储斐波那契数列项

```
int maxSize = 4 * 8 * 1024 * 1024; // 32 兆位
LruCache<Integer, BigInteger> cache = new LruCache<Integer, BigInteger> (maxSize) {
    protected int sizeOf (Integer key, BigInteger value) {
        return value.bitLength(); // 和对象占用的位数接近
    }
};
…
int n = 100;
BigInteger fN = cache.get(n);
if (fN == null) {
    fN = Fibonacci. computeRecursivelyWithCache(n);
    cache.put(n, fN);
}
```

每当需要选择一个数据结构来解决问题时，最好将选择范围缩小到只有几个类，因为通常每个类为特定目的或为提供特定的服务而优化。例如，如果你不需要在数据结构内部处理同步，应该选择 ArrayList 而不是 Vector。[①]当然，你完全可以创建自己的数据结构类，无论是从零开始（继承 Object），还是继承现有的类。

注意　在代码清单 1-11 所示的 computeRecursivelyWithCache 中，你能解释 LruCache 为什么不是内部缓存的好选择吗？

如果你使用基于散列的数据结构（例如 HashMap），而且键是自定义的对象，确保你正确地覆写了类定义中的 equal 和 hashCode 方法。hashCode 的低劣实现可以轻易将散列的收益化为乌有。

提示　请参考 http://d.android.com/reference/java/lang/Object.html，那里有实现 hashCode()的一个很好的示例。

尽管对于许多嵌入式程序开发者来说是不自然的，但也要考虑在程序中将一种数据结构转换成另一种：在某些情况下，因为使用了更好的算法，这种转换得到的性能提升远大于转换开销。一个常见的例子是将集合转为数组，可能是有序的。这种转换要创建新的对象，显然需要内存。在内存受限的设备上，这样的内存分配可能并不总会成功，会抛出一个 OutOfMemoryError 异常。Java 语言规范说了两件事：

- Error 类及其子类是普通程序中抛出的异常，它们通常是不期望能够恢复的。
- 精密的程序可能希望抓住这异常并试图从错误异常中恢复。

如果内存分配仅是优化的一部分，作为一个老练的应用开发者，你可以提供一个备用的机制（例如，使用原始数据结构的算法，尽管速度较慢），然后捕获 OutOfMemoryError 异常，这样做是

---

① Vector 内部有锁，影响性能。——译者注

更多资源请访问稀酷客(www.ckook.com)

有价值的，它可以让你程序跑在更多的设备上。这种可选的优化让代码难以维护，但给了你一个更大的腾挪余地。

注意 与直觉相反，并非所有异常都是 Exception 的子类。所有异常都是 Throwable 的子类（只有 Exception 和 Error 是它的直接子类）。

一般情况下，你应该很熟悉 java.util 和 android.util 包，几乎所有的组件依赖这两个工具箱。每当新的 Android 版本发布，你都应该特别注意这些包的修改（添加和修改的类）并参考“API 的变化报告”，详见 http://d.android.com/sdk。我们将在第 5 章讨论 java.util.concurrent 中更多的数据结构。

1.6 响应能力

应用的性能不仅仅在乎于速度，也要能让用户真正感觉到快才行。例如，显得更快的方法有，应用可以延迟创建对象，直到需要时才创建，称为推迟初始化的技术。另外，在开发过程中，你很有可能要在关注性能的地方侦测执行缓慢的代码。

下面的类是大多数 Android Java 应用的基石：

- Application
- Activity
- Service
- ContentProvider
- BroadcastReceiver
- Fragment（Android 3.0 及以上）
- View

在这些类中要特别注意的是所有的 onSomething()方法，它们由主线程调用，比如 onStart()和 onFocusChanged()。主线程也称为 UI 线程，可以说应用就在其中运行。在主线程中运行所有代码是可以做到的，但不推荐这样做。主线程里包括如下要处理的事情：

- 按键事件接收（例如，View.onKeyDown() 和 Activity.onKeyLongPress()）；
- 绘制 View（View.onDraw()）；
- 产生生命周期事件（例如，Activity.onCreate()）。

注意 许多方法都被设计为由主线程予以调用。当你覆写方法时，要确认它是如何将被调用的。Android 的文档并不没有明确说明方法是否该从主线程调用。

更多资源请访问稀酷客(www.ckook.com)

在一般情况下，无论事件从系统本身还是从用户处发生，主线程都在不断接收正在发生的事件的通知。应用只有一个主线程，因此，所有的事件都按顺序处理。很显然，这样一来响应能力会受到负面影响：在队列中的第一个事件处理完之前，后面的事件都是挂起的，一次只能处理一个。如果前面事件的处理需要很长时间才能完成，那么后续其他事件需要等很长时间才轮到处理。

一个简单的例子是从主线程调用 computeRecursivelyWithCache。虽然当 n 很小时还算快，但随着 n 的增长调用会变得越来越慢。当 n 值非常大时，你肯定会碰到 Android 上臭名昭著的应用没有响应（Application Not Responding，ANR）对话框。如果 Android 检测到应用没有响应，也就是说当 Android 检测到输入事件在 5 秒钟内没有被处理，或者 BroadcastReceiver 在 10 秒内没有执行完毕，这个对话框就会跳出来。当发生这种情况，用户的可选项只有“等待”或“强制关闭”应用（这可能会导致你的应用迈向被卸载的结局）。

为所有 Activity 优化启动序列是非常重要的，包括以下调用：

- onCreate
- onStart
- onResume

当然，这个序列发生在创建 Activity 时，实际上它发生的可能比你想象的更频繁。当配置发生变化时，当前 Activity 被销毁，并创建一个新实例，会调用以下序列：

- onPause
- onStop
- onDestroy
- onCreate
- onStart
- onResume

这个序列完成得很快，用户也会很快再次使用你的应用。最常见的配置变化之一是，设备的方向发生旋转。

注意　应用可以在 manifest 文件里指定每个 Activity 元素的 Android:configChanges 属性，让它只接受自己想处理的配置变化。这会导致调用 Activity 的 onConfigurationChanged()，而不是销毁。

Activity 的 onCreate() 方法一般会包含调用 setContentView 或任何其他负责展开资源的方法。因为展开资源是一个开销相对较大的操作，所以你可以通过降低布局（Layout，XML 文件定义应用的外观）复杂性来使资源展开加快。几个降低布局复杂性的步骤如下。

- 使用 RelativeLayout 代替嵌套 LinearLayouts，尽可能保持“扁平化”的布局。此外，减少创建的对象数量，也会让事件的处理速度加快。
- 使用 ViewStub 推迟对象创建（见 1.6.1 节）。

更多资源请访问稀酷客(www.ckook.com)

注意 因为可能有许多项目在列表中，所以要特别注意 ListView 布局。你可以使用 SDK 的 layoutopt 工具来分析布局。

优化的基本原则是保持应用的持续响应，让主线程尽可能快地完成任务。这句话也常有另一个说法，在主线程当中尽可能少做事情。在大多数情况下，你可以通过把操作转移到另一个线程或推迟操作来加快应用响应速度，这两种技术通常并不会使代码更难维护。把任务转移到另一个线程之前，一定要确保你已充分了解任务执行太慢的原因。如果响应慢的原因是坏的算法或实现，就去修改它们，将任务转移到另一个线程只是欲盖弥彰而已。

1.6.1 推迟初始化

拖延还是有其可取之处。通常的做法是在组件的 onCreate()方法中执行所有初始化。虽然这样做可行，但这意味着 onCreate()需要较长的时间才能结束。这一点对应用的 Activity 尤为重要，onStart()直到 onCreate()完成后才会被调用（同样，onResume()只有在 onStart()完成后才会被调用），任何延迟都会导致应用需要较长时间才能启动，用户最终可能会感到难以忍受。

例如，Android 使用 android.view.ViewStub 来推迟初始化，它可以在运行时展开资源。当 View- Stub 需要展现时，它被相应的资源展开替换，自己就成为等待垃圾回收的对象。

由于内存分配需要花时间，等到对象真正需要时才进行分配，也是一个很好的选择。当某个对象并不是立即就要使用时，推迟创建对象有着很明显的好处。代码清单 1-14 是推迟初始化的示例，它是基于代码清单 1-13 写的。为了避免总是检查对象是否为空，考虑使用工厂方法模式。

代码清单 1-14 推迟分配缓存

```
int n = 100;
if (cache == null) {
    // createCache 分配缓存对象，可以从许多地方调用它
    cache = createCache();
}
BigInteger fN = cache.get(n);
if (fN == null) {
    fN = Fibonacci. computeRecursivelyWithCache(n);
    cache.put(n, fN);
}
```

请参阅第 8 章学习如何在 XML Layout 中使用 android.view.ViewStub，以及如何推迟资源展开。

1.6.2 StrictMode

写程序时，你应该始终假定下列两种情况：

- 网络很慢（你正在试图连接的服务器甚至可能没有响应）；
- 文件系统的访问速度很慢。

更多资源请访问稀酷客(www.ckook.com)

结论就是，不应该在主线程内进行网络操作或访问文件系统，因为缓慢的操作会拖累系统的响应能力。虽然在开发环境中，你可能永远不会遇到任何网络问题或任何文件系统的性能问题，但用户可能不像你那么幸运。

注意　SD 卡并不都具有相同“速度”，如果应用在很大程度上依赖外部存储设备的性能，那么你应该确保在来自不同制造商的各种 SD 卡上测试过你的应用。

Android 有实用工具来帮助应用检测这类缺陷。它提供的 StrictMode 是检测不良行为的良好工具。通常情况下，在应用启动时，即当 onCreate()被调用时，启用 StrictMode，如代码清单 1-15 所示。

代码清单 1-15　在应用中启用 StrictMode

```
public class MyApplication extends Application {
    @Override
    public void onCreate () {
        super.onCreate();

        StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()
        .detectCustomSlowCalls()// API 等级 11，使用 StrictMode.noteSlowCode
        .detectDiskReads()
        .detectDiskWrites()
        .detectNetwork()
        .penaltyLog()
        .penaltyFlashScreen()// API 等级 11
        .build());

         // 其实和性能无关，但如果使用 StrictMode，最好也定义 VM 策略
         StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()
         .detectLeakedSqlLiteObjects()
         .detectLeakedClosableObjects()// API 等级 11
         .setClassInstanceLimit(Class.forName("com.apress.proandroid.SomeClass", 100)// API 等级 11
         .penaltyLog()
         .build());
    }
}
```

StrictMode 是 Android 2.3 引入的，在 Android 3.0 中加入了更多的功能，所以应该确保选择了正确的 Android 版本，让代码跑在适当的 Android 平台上，如代码清单 1-12 所示。

Android 3.0 中引入的需要特别留意的方法包括 detectCustomSlowCall()和 noteSlowCall()，它们都是用来检测应用中执行缓慢的代码或潜在缓慢的代码。代码清单 1-16 说明了如何将代码标记为潜在缓慢的代码。

代码清单 1-16　标记潜在缓慢的代码

```
public class Fibonacci {
    public static BigInteger computeRecursivelyWithCache(int n) {
       StrictMode.noteSlowCall("computeRecursivelyWithCache");// 消息可以带任何信息
         SparseArray<BigInteger> cache = new SparseArray<BigInteger>();
```

更多资源请访问稀酷客(www.ckook.com)

```
            return computeRecursivelyWithCache(n, cache);
        }
        ...
    }
```

从主线程调用 computeRecursivelyWithCache 执行时间过长，如果 StrictMode Thread 策略配置为检测缓慢调用时，会出现如下日志：

```
StrictMode policy violation; ~duration=21121 ms:
android.os.StrictMode$StrictModeCustomViolation: policy=31 violation=8 msg=computeRecursivelyWithCache
```

Android 提供了一些辅助方法，可以在主线程里进行临时磁盘读写，如代码清单 1-17 所示。

代码清单 1-17 修改线程策略，临时允许磁盘读取

```
StrictMode.ThreadPolicy oldPolicy = StrictMode.allowThreadDiskReads();
 // 从磁盘读取数据
 StrictMode.setThreadPolicy(oldPolicy);
```

目前没有临时允许网络访问的方法，但实在没有理由在主线程中允许这种访问，即使是暂时的，也没有合适的方式知道访问是否很快。有人可能会说，也没有合理的方式知道磁盘访问将是否是快速的，但那是另一场争论。

注意 只在开发阶段启用 StrictMode，发布应用时，记得要禁用它。如果你使用 detectAll() 方法去建立策略总是可行的，那将来更可行，未来的 Android 版本会检测出更多的不良行为。

1.7 SQLite

大多数应用都不会是 SQLite 的重度使用者，因此，不用太担心与数据库打交道时的性能。不过，在优化应用中 SQLite 相关的代码时，需要了解几个概念：

- SQLite 语句
- 事务
- 查询

注意 本节不是 SQLite 完整指南，而是提供几个关注点，确保有效地使用数据库。想要找完整的指南，请参阅 www.sqlite.org 和 Android 的在线文档。

本节所涵盖的优化不会使代码难以阅读和维护，所以要养成使用它们的习惯。

1.7.1 SQLite 语句

一开始，SQL 语句是简单的字符串，例如：

更多资源请访问稀酷客(www.ckook.com)

- CREATE TABLE cheese (name TEXT, origin TEXT)
- INSERT INTO cheese VALUES ('Roquefort', 'Roquefort-sur-Soulzon')

第一条语句将创建一个名为“cheese”的表，表有“name”和“origin”两列；第二条语句在表中插入新行。因为这些语句是简单的字符串，需要解释或编译才可以执行。当你执行 SQLite 的语句时，如代码清单 1-18 所示的 SQLiteDatabase.execSQL，SQLite 内部是编译执行的。

代码清单 1-18　执行简单的 SQLite 语句

```
SQLiteDatabase db = SQLiteDatabase.create(null); // 数据库在内存中
 db.execSQL("CREATE TABLE cheese (name TEXT, origin TEXT)");
 db.execSQL("INSERT INTO cheese VALUES ('Roquefort', 'Roquefort-sur-Soulzon')");
 db.close(); // 操作完成后记得关闭数据库
```

注意　许多 SQLite 的相关方法会抛出异常。

事实证明，执行 SQLite 的语句可能需要相当长的一段时间。除了编译，语句本身可能还需要创建。由于 String 是不可改变的，这可能会和之前在 computeRecursivelyFasterUsingBigInteger 中创建大量 BigInteger 对象时出现同样的性能问题。我们现在只关注 INSERT 语句的性能。毕竟，表只会创建一次，但会添加、修改或删除许多行。

如果我们想建立一个全面的奶酪数据库[①]（谁不想呢？），需要加入许多 INSERT 语句才能结束，如代码清单 1-19 所示。每个 INSERT 语句都会创建一个 String 并调用 execSQL，每个奶酪添加到数据库中时，SQL 语句会在内部解析。

代码清单 1-19　建立全面的奶酪数据库

```
public class Cheeses {
    private static final String[] sCheeseNames = {
        "Abbaye de Belloc",
        "Abbaye du Mont des Cats",
        ...
       "Vieux Boulogne"
    };
    private static final String[] sCheeseOrigins = {
         "Notre-Dame de Belloc",
         "Mont des Cats",
         ...
         "Boulogne-sur-Mer"
    };
    private final SQLiteDatabase db;
    public Cheeses () {
        db = SQLiteDatabase.create(null); // 内存数据库
        db.execSQL("CREATE TABLE cheese (name TEXT, origin TEXT)");
```

① 作者非常喜欢奶酪，例子都是奶酪名字和原产地，后面会继续奶酪做例子，感兴趣的读者可以自己搜索。

——译者注

更多资源请访问稀酷客(www.ckook.com)

```
    }
    public void populateWithStringPlus () {
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            String sql = "INSERT INTO cheese VALUES(\"" + name + "\",\"" + origin + "\")";
            db.execSQL(sql);
        }
    }
}
```

在三星 Galaxy Tab 10.1 上，添加 650 条奶酪数据到内存数据库用时 393 毫秒，平均一条 0.6 毫秒。

一个显而易见的优化方法是，加快要执行的 SQL 语句字符串的创建速度。在这种情况下，使用+运算符来连接字符串不是最有效的方法，而使用 StringBuilder 对象，或调用 String.format 可以提高性能。代码清单 1-20 给出了两种新方法。它们只是优化了传递给 execSQL 的字符串创建速度，这两种方法本身并不算是与 SQL 相关的优化。

代码清单 1-20　采用更快的方法创建 SQL 语句字符串

```
public void populateWithStringFormat () {
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        String sql = String.format("INSERT INTO cheese VALUES(\"%s\",\"%s\")", name, origin);
        db.execSQL(sql);
    }
}

public void populateWithStringBuilder () {
    StringBuilder builder = new StringBuilder();
    builder.append("NSERT INTO cheese VALUES(\"");
    int resetLength = builder.length();
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        builder.setLength(resetLength); // 复位位置
        builder.append(name).append("\",\"").append(origin).append("\")");// 链式调用
        db.execSQL(builder.toString());
    }
}
```

增加相同数量的奶酪条目，String.format 版本用时 436 毫秒，而 StringBuilder 的版本用时 371 毫秒。String.format 版本比原始版本慢，StringBuilder 版本仅快了一点点。

尽管这三种方法创建字符串的方式不同，但都做了相同的事情，都调用了 execSQL，execSQL 完成实际的语句编译（解析）[①]。因为所有的语句都非常相似（仅是奶酪的名字和原产地不同），所以可以使用 compileStatement 让语句在循环外只编译一次。实现如代码清单 1-21 所示。

① 参见 SQLite 的介绍，内部有虚拟机，处理字节码指令。——译者注

更多资源请访问稀酷客(www.ckook.com)

代码清单 1-21　SQLite 语句的编译

```
public void populateWithCompileStatement () {
    SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese VALUES(?,?)");
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        stmt.clearBindings();
        stmt.bindString(1, name); // 替换第一个问号为 name
        stmt.bindString(2, origin); // 替换第二个问号为 origin
        stmt.executeInsert();
    }
}
```

因为只进行一次语句编译，而不是 650 次，并且绑定值是比编译更轻量的操作，所以这种方法明显快多了，建立数据库只用了 268 毫秒。这样还使代码更具可读性。

Android 还提供了其他的 API，使用 ContentValues 对象把值插入到数据库中，它基本上包含了列名和值之间的绑定信息。实现如代码清单 1-22 所示，实际上非常近似 populateWithCompileStatement，"INSERT INTO cheese VALUES"字符串甚至不作为 INSERT 语句的一部分出场，暗示了这部分是靠调用 db.insert()实现的。

但是，这个实现用了 352 毫秒，性能低于 populateWithCompileStatement。

代码清单 1-22　使用 ContentValues 填充数据库

```
public void populateWithContentValues () {
    ContentValues values = new ContentValues();
    int i = 0;
    for (String name : sCheeseNames) {
        String origin = sCheeseOrigins[i++];
        values.clear();
        values.put("name", name);
        values.put("origin", origin);
        db.insert("cheese", null, values);
    }
}
```

最快的实现，也是最灵活的，因为它允许在语句中有更多的选择。例如，你可以使用 INSERT OR FAIL 或 INSERT OR IGNORE 来替换简单的 INSERT。

注意　Android 3.0 的 android.database 和 android.database.sqlite 包发生了许多变化。例如，Activity 类中的 managedQuery、startManagingCursor 和 stopManagingCursor 方法已废弃，由 CursorLoader 取而代之。

Android 也定义了一些可以提高性能的类。例如，可以使用 DatabaseUtils.InsertHelper 在数据库中插入多行，这样只需编译一次 INSERT 语句。它目前和 populateWithCompileStatement 的实现方案类似，但我们所关注的选项（例如，FAIL 或 ROLLBACK）不提供与之同样的灵活性。

更多资源请访问稀酷客(www.ckook.com)

题外话，和性能无关，你也可以使用 DatabaseUtils 类的静态方法来简化实现。

1.7.2 事务

上述例子中并没有显式创建任何事务，但会自动为每个插入操作创建一个事务，并在每次插入后立即提交。显式创建事务有以下两个基本特性：

- 原子提交
- 性能更好

抛开对性能的追求，第一个特性是很重要的。原子提交意味着数据库的所有修改都完成或都不做。事务不会只提交部分修改。在这个例子中，我们可以考虑把插入 650 个奶酪条目当作一个事务。要么成功建立完整的奶酪列表，要么没有插入任何奶酪条目，不会只建立部分列表。实现如代码清单 1-23 所示。

代码清单 1-23 在事务中插入所有的奶酪

```
public void populateWithCompileStatementOneTransaction () {
    try {
        db.beginTransaction();
        SQLiteStatement stmt = db.compileStatement("INSERT INTO cheese VALUES(?,?)");
        int i = 0;
        for (String name : sCheeseNames) {
            String origin = sCheeseOrigins[i++];
            stmt.clearBindings();
            stmt.bindString(1, name); // 替换第一个问号为 name
            stmt. bindString(2, origin);// 替换第二个问号为 origin
            stmt.executeInsert();
        }
        db.setTransactionSuccessful(); // 删除这一调用不会提交任何改动!
    } catch (Exception e) {
        // 在这里处理异常
    } finally {
        db.endTransaction(); //必须写在 finally 块
    }
}
```

新实现了用了 166 毫秒，改进相当大（快了约 100 毫秒）。有人会争论说这两种实现对于大多数应用来说都是可以接受的，因为同时这么快速地插入这么多行很少见。事实上，大多数应用通常会在某一时刻访问很多行，这可能是在响应一些用户操作。最重要的一点是，该数据库是在内存中，而不是保存到持久存储（SD 卡或内部闪存）上的。在数据库工作时，大量的时间花费在访问持久性存储上（读/写），这比访问易失性记忆体慢得多。通过测量内部持久存储上的数据库的操作，可以确定单个事务的性能。建立在持久存储上的数据库，如代码清单 1-24 所示。

代码清单 1-24 在持久存储上的创建数据库

```
public Cheeses (String path) {
    // 路径可能已经由 getDatabasePath("fromage.db")创建了

    // 你也可以调用 mkdirs 确保路径存在
```

更多资源请访问稀酷客(www.ckook.com)

```
        // File file = new File(path);
        // File parent = new File(file.getParent());
        // parent.mkdirs();

        db = SQLiteDatabase.openOrCreateDatabase(path, null);
        db.execSQL("CREATE TABLE cheese (name TEXT, origin TEXT)");
    }
```

当数据库在持久存储，而不是在内存中时，调用 populateWithCompileStatement 需要花 34 秒的时间，每行（52 毫秒），而调用 populateWithCompileStatementOneTransaction 时间不到 200 毫秒。无需多言，一次性事务是解决这类问题的较好办法。这些数据显然取决于要用的存储类型，外部 SD 卡上存储的数据库会更慢，更应该采用一次性事务。

注意　*在存储上创建数据库时，确保父目录已经存在。参见 Context.getDatabasePath 和 File.mkdirs 的文档获取更多信息。为方便起见，可以使用 SQLiteOpenHelper 来代替手动创建数据库。*

1.7.3　查询

我们可以用限制数据库访问的方式来加快查询速度，尤其是对存储中的数据库。数据库查询仅会返回一个 Cursor（游标）对象，然后用它来遍历结果。代码清单 1-25 给出了两种方法来遍历所有的行。第一种方法是创建 Cursor 获取数据库中的两列数据，而第二种方法创建的 Cursor 只获取第一列。

代码清单 1-25　遍历所有的行

```
public void iterateBothColumns () {
    Cursor c = db.query("cheese", null, null, null, null, null, null);
    if (c.moveToFirst()) {
        do {
        } while (c.moveToNext());
    }
    c.close(); // 结束时（或在出现异常时）切记要关闭 Cursor
}

public void iterateFirstColumn () {
    Cursor c = db.query("cheese", new String[]{"name", null, null, null, null, null}); // 唯一区别
    if (c.moveToFirst()) {
        do {
        } while (c.moveToNext());
    }
    c.close();
}
```

和预想的一样，因为没有从第二列读取数据，第二种方法快了不少，两方法分别用时 61 毫秒和 23 毫秒（使用多个事务时）。迭代所有行时，将所有行作为一个事务处理会更快，这种情况

更多资源请访问稀酷客(www.ckook.com)

下 iterateBothColumns 和 iterateFirstColumn 各用时 11 毫秒和 7 毫秒。你可以看到，只读取需要的数据才是上上之选。调用查询时选择正确的参数，可以使性能有可观的提升。如果只需要一定数量的行，指定调用查询的限制参数，则可以进一步减少数据库的访问时间。

提示 考虑使用 SQLite 的 FTS（全文检索）扩展，它支持更多高级搜索特性（使用索引）。参阅 www.sqlite.org/fts3.html 获取更多信息。

1.8 总结

几年前，Java 由于性能问题而广受诟病，但现在情况已大有改观。每次发布新版本 Android 时，Dalvik 虚拟机（包括它的 JIT 编译器）的性能都会有所提升。代码可以编译为本地代码，从而利用最新的 CPU 架构，而不必重新编译。虽然实现很重要，但最重要的还是慎选数据结构和算法。好算法可以弥补差的实现，甚至不需要优化就可以使应用流畅运行；而坏算法无论你在实现上花费多少精力，其结果还是会很糟糕。

最后，不要牺牲响应能力。这可能会加大应用开发的难度，但响应顺畅是应用成功的关键。

更多资源请访问稀酷客(www.ckook.com)

第2章 NDK 入门

Android的原生开发套件（NDK）是SDK的辅助工具，可以用它把Android应用的一部分或全部用本地代码实现。字节码需要由虚拟机解释，而本地代码可以由设备处理器直接执行，没有任何中间步骤，执行速度更快，有时快很多。Dalvik的JIT编译器可以将字节码编译为本地代码，减少应用字节码的解释次数（理想情况下仅有一次），而直接使用自己生成的本地代码，可以让应用运行得更快。使用NDK时，是在开发环境中将代码编译为本地代码，而不是在Android设备上。你可能会奇怪，为什么需要操心NDK呢？Dalvik的JIT编译器就可以动态生成本地代码了，只用Java SDK写应用不就行了么？本章将阐述使用NDK的缘由和各种使用方式。

使用本地代码和NDK的方式有以下两种：

- 应用的一部分用Java编写，另一部分用C/C++编写；
- 应用全部用C/C++写。

注意 NDK是在Android 1.5时引入的。今天，很少有设备仍运行Android 1.5或更早的版本了，因此使用NDK及C/C++编写应用的一部分是没问题的。不过，如果用C/C++来写整个应用，则需要Android 2.3或更高版本。

本章先介绍NDK的组成部分。然后，学习在Android应用中如何混合使用C/C++与Java，以及如何确保针对目标平台进行代码优化。最后，我们将深入探讨一个新类，即Android 2.3推出的NativeActivity，可以用它写出纯C/C++的应用，最后是一个在C/C++使用传感器的简单示例。

2.1 NDK里有什么

NDK是为应用开发本地代码而诞生的一套工具。所有文件都放在一个单独的目录中，你可以从http://d.android.com/sdk/ndk下载压缩包。例如，Windows版的NDK r6b包含以下目录：

- build
- docs
- platforms

更多资源请访问稀酷客(www.ckook.com)

- samples
- sources
- tests
- toolchains

NDK 目录的根目录中还包含以下几个文件：

- documentation.html
- GNUmakefile
- ndk-build
- ndk-gdb
- ndk-stack
- README.txt
- RELEASE.txt

和 http://d.android.com 的 SDK 相比，NDK 的文档少得可怜，用你常用的网页浏览器打开 documentation.html，就可以进入了 NDK 的世界。README.TXT 文件也最好先看一下，会有些帮助。

NDK 是以下 6 个组件的集合：

- 文档
- 头文件
- C/C++文件
- 预编译库
- 编译、链接、分析和调试代码的工具
- 示例应用程序

就定义而言，本地代码是具体到某个架构的。例如，Intel CPU 不接受 ARM 指令，反之亦然。因此，NDK 包括多个平台以及不同版本的预编译库。NDK r7 支持以下三个应用程序二进制接口（ABI）：

- armeabi
- armeabi-v7a
- x86

注意 NDK 不支持 ARMv6 的 ABI。

大多数人都已经对 x86 耳熟能详，它表示 Intel 架构，这种架构非常普及。armeabi 和 armeabi-v7a 这两个名字你可能不是很熟悉，但你能找到许多基于 ARM 芯片的产品，从洗衣机到 DVD 播放机，在 Android 诞生之前，你应该使用基于 ARM 的设备很长时间了。仅 2011 年第二季度，基于 ARM 的芯片出货量就达到 2 亿，其中手机和平板电脑 1.1 亿，其他消费电子和嵌入式设备 0.8 亿。

更多资源请访问稀酷客(www.ckook.com)

术语“armeabi”代表ARM嵌入式应用程序二进制接口，而v5和v7a是指两种不同的架构。ARM架构最初版本是v1，最新的是v7。每种架构都有一系列处理器核心在使用：ARM7/ARM9/ARM10核心使用v5、Cortex系列使用v7。Cortex系列包括A5、A8、A9以及即将面世的A15，目前的智能手机和平板电脑主要使用A8和A9。

ARM11系列处理器核心使用ARMv6架构，一些Android设备使用了基于ARM11的芯片，不过Android NDK不支持这个架构。表2-1列出了一些Android设备及其架构。

表2-1　一些Android的设备及其架构

设 备 名	制 造 商	CPU	处理器系列
Blade	中兴	高通 MSM7227	ARM11
LePhone	联想	高通 Snapdragon	基于Cortex A8
Nexus S	三星	三星 Hummingbird	Cortex A8
Xoom	摩托罗拉	Nvidia Tegra 2	Cortex A9 （双核）
Galaxy Tab（7"）	三星	三星 Hummingbird	Cortex A8
Galaxy Tab 10.1	三星	Nvidia Tegra 2	Cortex A9 （双核）
Revue （机顶盒）	罗技	CE4150 （Sodaville）	Intel Atom
NSZ-GT1 （蓝光播放器）	索尼	CE4170 （Sodaville）	Intel Atom

MIPS科技宣布，运行Android 2.2的MIPS智能手机早在2011年6月就通过了Android兼容性测试，但官方Android NDK仍然不支持MIPS ABI。[①]截至目前，ARM仍然是Android设备的主导架构。

注意　所有在2010年发布的谷歌电视设备（罗技机顶盒，索尼电视和蓝光播放器）都是基于英特尔CE4100。但谷歌电视平台目前还不支持NDK。

NDK经常更新，应该总是使用最新的版本。新版本一般会通过提供更好的编译器或优化更好的预编译库来提高性能，同时也会修复以前版本中的bug。当发布应用的更新版本时，即使你只修改了应用的Java部分，也最好用最新版本的NDK重新编译C/C++代码。不过，请确保你的C/C++代码通过了测试！表2-2显示了NDK各版的修订内容。

表2-2　Android NDK的修订版本

修订版本	日　期	功　能
1	2009年6月	Android 1.5 NDK，发布版本1，支持ARMv5TE指令，GCC 4.2.1
2	2009年9月	Android 1.6 NDK，发布版本1，增加OpenGL ES 1.1原生库支持
3	2010年3月	增加了OpenGL ES 2.0原生库支持，GCC 4.4.0
4b	2010年6月	用ndk-build工具简化构建系统，用ndk-gdb简化调试过程，增加了armeabi-v7a（Thumb-2、VFP、NEON高级SIMD）的支持，增加了本地代码访问Bitmap对象像素缓冲区的API

① NDK r8已支持MIPS。——译者注

更多资源请访问稀酷客(www.ckook.com)

（续）

修订版本	日　期	功　能
5c	2011年6月	更多的本地API（真的很多！），添加预编译库支持，GCC 4.4.3，修复了r5（2010年12月）及5b（2011年1月）的问题
6b	2011年8月	增加了x86 ABI的支持，增加新的ndk-stack工具用于调试，修复r6（2011年7月）的问题
7	2011年11月	基于OpenMAX 1.0.1的本地多媒体API，基于OpenSL 1.0.1的本地声音API，新C++运行时（gabi++和gnustl_shared），支持STLport的RTTI
8	2012年5月	增加MIPS支持[①]

2.2 混合使用 Java 和 C/C++代码

从 Java 调用 C/C++函数其实很简单，只需以下几个步骤。

(1) 必须在 Java 代码中声明本地方法。

(2) 需要实现 Java 本地接口（JNI）粘合层。

(3) 必须创建 Android makefile 文件。

(4) 必须用 C/C++实现本地方法。

(5) 必须编译本地库。

(6) 必须加载本地库。

在 NDK 中做这些非常容易。我们将逐步完成这些工作，本节结束时，你就会了解混合 Java 和 C/C++的基础知识。我们会在后面的小节详细讨论 Android makefile 更复杂的细节，让你可以更好地优化代码。目前，Android NDK 可以安装在 Linux、MacOS X 和 Windows（Cygwin，或者不用 Cygwin 的 NDK r7）上，尽管总体操作基本一致，但具体步骤可能会略有不同。以下步骤假定已经创建了一个 Android 项目，接下来需要添加本地代码。

2.2.1 声明本地方法

代码清单 2-1 所示的第一步只做了一件事。

代码清单 2-1　声明 Fibonacci.java 中的本地方法

```
public class Fibonacci {
    public static native long recursiveNative (int n); // 注意 native 关键字
}
```

在 Java 代码中只是使用 native 关键字声明本地方法，并没有实现。上面所示的方法是公共的，但本地方法可以是公共的，也可以是受保护的、私有的或包作用域的，就像任何其他 Java 方法一样。同样，本地方法不必是静态方法，也不是非得用基本类型。从调用者角度来看，本地方法就像任何其他方法一样，一旦声明了就可以在 Java 代码中调用，会顺利通过编译。不过，如果应用运行并调用 Fibonacci.recursiveNative，它就会崩溃并抛出 UnsatisfiedLinkError 异

① 此条为译者添加。——编者注

更多资源请访问稀酷客(www.ckook.com)

常。这在意料之中，因为除了声明函数外没有做其他任何事情，函数的实现部分实际并不存在。

在声明了本地方法之后，就可以开始编写 JNI 粘合层了。

2.2.2 实现 JNI 粘合层

Java 通过 JNI 框架调用库里的 C/C++方法。可以利用 JDK（Java 开发工具包）建立 JNI 粘合层。首先，需要在 C/C++头文件中声明要实现的函数。你不必亲手写这个头文件，可以（而且应该）使用 JDK 的 javah 工具生成。

在命令行终端，只需更改当前目录为应用目录，并调用 javah 创建所需要的头文件。在应用的 jni 目录创建头文件。如果 jni 目录不存在，必须在创建头文件前先创建这个目录。假设项目保存在~/workplace/MyFibonacciApp，命令执行如下：

```
cd ~/workspace/MyFibonacciApp
mkdir jni
javah -classpath bin -jni -d jni com.apress.proandroid.Fibonacci
```

注意　必须提供类的完整名称。如果 javah 返回“Class com.apress.proandroid.Fibonacci not found”错误，确保用-classpath 指定了正确目录，并检查类名是否正确。-d 选项指定保存头文件的路径。由于 javah 需要使用 Fibonacci.class，所以执行命令之前应该先编译 Java 应用。[①]

现在头文件已经创建，它有个不讨人喜欢的文件名 com_apress_proandroid_Fibonacci.h，存放在~/workplace/ MyFibonacciApp/jni 目录中，文件内容如代码清单 2-2 所示。不要直接修改这个文件，如果需要更新文件版本（例如，要在 Java 文件里重命名或添个新的本地方法），可以用 javah 再创建一次。

代码清单 2-2　JNI 头文件

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/*Header for class com_apress_proandroid_Fibonacci */

#ifndef _Included_com_apress_proandroid_Fibonacci
#define _Included_com_apress_proandroid_Fibonacci
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     com_apress_proandroid_Fibonacci
 * Method:    recursiveNative
 * Signature: (I)J
 */
JNIEXPORT jlong JNICALL
```

---

① 不要在 Cygwin 里使用 javah，可能会出现错误，另外 jdk7 需要指定 android.jar 的路径。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
Java_com_apress_proandroid_Fibonacci_recursiveNative
   (JNIEnv *, jclass, jint);

#ifdef __cplusplus
}
#endif
#endif
```

C 头文件本身做不了任何事情。现在需要在创建的 com_apress_proandroid_Fibonacci.c 文件中实现 Java_com_apress_proandroid_Fibonacci_recursiveNative 函数，如代码清单 2-3 所示。

代码清单 2-3 JNI C 源文件

```
#include "com_apress_proandroid_Fibonacci.h"

/*
 * 类:     com_apress_proandroid_Fibonacci
 * 方法:    recursiveNative
 * 签名: (I)J
 */
jlong JNICALL
Java_com_apress_proandroid_Fibonacci_recursiveNative
   (JNIEnv *env, jclass clazz, jint n) {
     return 0; // 现在只是一个空实现，先返回 0
}
```

JNI 层的所有函数有一个共同点：它们的第一个参数都是 JNIEnv*类型（JNIEnv 对象的指针）。JNIEnv 的对象是 JNI 环境本身，使用它可以与虚拟机交互（如果需要的话）。第二个参数，当方法被声明为静态时为 jclass 类型，否则为 jobject 类型。

提示 试试用 javah 的-stubs 选项生成 C 文件（javah -classpath bin -stubs com_apress_proandroid_Fibonacci -d jni）。你很可能会得到这个错误信息："Error: JNI does not require stubs, please refer to the JNI documentation"，尽管有这个错误信息，但在老版本 JDK 中还是会生成 C 文件。[①]

2.2.3 创建 Makefile

此时，你肯定可以把这个 C++文件用 NDK 的 GCC 编译器编译为库，但 NDK 提供了一个工具——ndk-build，可以帮你完成这些工作。接下来看看这个工具都做什么，ndk-build 工具会用到两个由你创建的文件：

- Application.mk（可选）
- Android.mk

应该在应用的 jni 目录（JIN 头文件和源文件已经在里面了）创建这两个文件。只需参照现有

① 新版本不会生成，stub 参数不起作用。——译者注

更多资源请访问稀酷客(www.ckook.com)

已经定义了这两个文件的项目即可创建它们。NDK 在 samples 目录中包含了使用本地代码的应用示例，hello-jni 是最简单的一个。由于 Application.mk 是一个可选的文件，并不是每个示例都有。先不用操心性能问题，使用非常简单的 Application.mk 和 Android.mk 尽可能快地创建应用程序。尽管 Application.mk 是可选的，可以不用它，我们还是先看下这个文件的一个基础版本，如代码清单 2-4 所示。

代码清单 2-4　Application.mk 文件的一个基础版本，指定了一种 ABI

```
APP_ABI := armeabi-v7a
```

这个 Application.mk 只指定了构建一个版本的库，这个版本针对的是 Cortex 系列处理器。如果没有提供 Application.mk，将会创建仅针对 armeabi ABI (ARMv5)这一个版本的库，这相当于定义代码清单 2-5 所示的 Application.mk 文件。

代码清单 2-5　指定 armeabi 作为唯一的 ABI 的 Application.mk 文件

```
APP_ABI := armeabi
```

Android.mk 最简单的形式也略有些冗长，语法复杂是一方面，另一方面是要声明最终生成的库。代码清单 2-6 是 Android.mk 的基础版本。

代码清单 2-6　Android.mk 的基础版本

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := fibonacci

LOCAL_SRC_FILES := com_apress_proandroid_Fibonacci.c

include $(BUILD_SHARED_LIBRARY)
```

Android.mk 文件起始处定义了本地路径，就是 Android.mk 所在位置。Android NDK 提供了一些宏，供你在 makefile 中使用，这里我们使用 my-dir 宏，它返回最后包含的 makefile 文件的路径。在我们的例子中，最后包含的 makefile 就是在~/workplace/MyFibonacciApp/jni 的目录下的 Android.mk，因此 LOCAL_PATH 被设置为~/MyFibonacciApp。

第二行只是清除 LOCAL_PATH 之外的所有 LOCAL_*XXX* 变量，如果忘记了这行，变量可能会被错误定义。想让构建开始于可预见的状态，切记不要忘记在定义一个模块之前把这行包含进来。

LOCAL_MODULE 定义了模块的名称，这也是生成库的名称。例如，如果 LOCAL_MODULE 设置为 fibonacci，那么共享库的名称就是 libfibonacci.so。LOCAL_SRC_FILES 列出所有需要编译的文件，我们的例子中只有 com_apress_proandroid_Fibonacci.c（JNI 粘合层），还没有真正实现 Fibonacci 函数。每当添加一个新文件，切记将它添加到 LOCAL_SRC_FILES 队列中，否则将不会编译到库中。

最后，当所有的变量定义好后，需要包含指定了构建库的规则的文件。在此例中要编译共享库，因此使用了 include $(BUILD_SHARED_LIBRARY)

虽然看起来有些复杂，不过当前只需要关心 LOCAL_MODULE 和 LOCAL_SRC_FILES 的定义，文件

更多资源请访问稀酷客(www.ckook.com)

的其余部分照抄样板即可。

如需更多有关这些 makefile 文件的详细信息，请参阅本章的 2.3 节和 2.4 节。

2.2.4 实现本地函数

现在 makefile 已经定义，我们需要实现新建的 fibonacci.c 中的函数，如代码清单 2-7 所示，然后从粘合层调用函数，如代码清单 2-8 所示。fibonacci.c 中的函数在调用之前需要先声明，因此还要创建一个新的头文件，如代码清单 2-9 所示。添加 fibonacci.c 到文件列表里，利用 Android.mk 编译。

代码清单 2-7 实现 fibonacci.c 的新函数

```
#include "fibonacci.h"

uint64_t recursive (unsigned int n) {
    if (n > 1) return recursive(n-2) + recursive(n-1);
    return n;
}
```

代码清单 2-8 从粘合层调用函数

```
#include "com_apress_proandroid_Fibonacci.h"
#include "fibonacci.h"

/*
 * Class:         com_apress_proandroid_Fibonacci
 * Method:    recursiveNative
 * Signature:    (I)J
 */

jlong JNICALL
Java_com_apress_proandroid_Fibonacci_recursiveNative
   (JNIEnv *env, jclass clazz, jint n) {
     return recursive(n);
}
```

代码清单 2-9 头文件 fibonacci.h

```
#ifndef _FIBONACCI_H_
#define _FIBONACCI_H_

#include <stdint.h>

extern uint64_t recursive (unsigned int n);

#endif
```

注意 请确保使用了正确的类型，在 C/C++代码中 jlong 是 64 位。使用定义良好的类型，针对占用位数不同使用合适的类型，如 uint64_t 或 int32_t。[①]

① C99 规范定义了 stdint.h，包含了这些类型，gcc 和 vs2010 也都有。——译者注

更多资源请访问稀酷客(www.ckook.com)

有些人可能认为，使用多个文件增加了不必要的复杂性，一切都可以在粘合层实现，即用单个文件，而不是3个或4个（fibonacci.h、fibonacci.c、com_apress_proandroid_Fibonacci.c甚至还有com_apress_proandroid_Fibonacci.h）。虽然这在技术上是可行的，如代码清单2-10所示，但不推荐这样做。这样会将粘合层与本地函数实现捆绑在一起，使得本地函数代码难以在非 Java 程序中重用。例如，你可能要在iOS程序中重用相同的C/C++头文件和实现。让JNI待在粘合层，让粘合层中只有JNI。

虽然你可能想在包含列表里删除JNI头文件，但请把它留下，因为它可以确保函数实现和Java层定义保持一致。（你应该还记得我们之前说过，只要 Java 中有相关变化，就用 javah 重新生成头文件。）

代码清单2-10　文件合三为一

```
#include "com_apress_proandroid_Fibonacci.h"
#include <stdint.h>

static uint64_t recursive (unsigned int n) {
     if (n > 1) return recursive(n-2) + recursive(n-1);
     return n;
}
/*
 * Class:     com_apress_proandroid_Fibonacci
 * Method:    recursiveNative
 * Signature: (I)J
 */
jlong JNICALL
Java_com_apress_proandroid_Fibonacci_recursiveNative
   (JNIEnv *env, jclass clazz, jint n) {
     return recursive(n);
}
```

2.2.5　编译本地库

现在C实现完成了，最后，在应用程序的JNI目录使用ndk-build命令构建共享库。

提示　修改 PATH 环境变量，把 NDK 目录加进去，这样你就可以很方便的使用 ndk-build 或其他脚本文件，而不用指定命令的全路径。[①]

最终，在lib/armeabi目录中生成了libfibonacci.so共享库。你可能还需要刷新Eclipse项目，以显示新创建的库。如果此时编译并运行应用程序，应用程序调用了Fibonacci.recursiveNative，它会再次崩溃并抛出UnsatisfiedLinkError异常。这是个典型错误，许多开发者压根没想起来要在 Java 代码中显式地加载共享库：虚拟机不能未卜先知，必须告诉它需要加载什么库。这可以通过调用System.loadLibrary()实现，如代码清单2-11所示。

① 除了NDK根目录，建议再加上toolchain里的bin目录，可以直接使用objdump等工具。——译者注

更多资源请访问稀酷客(www.ckook.com)

代码清单 2-11 在静态初始化块中加载库

```
public class Fibonacci {
    static {
        System.loadLibrary("fibonacci"); // 加载 libfibonacci.so
    }

    public static native long recursiveNative (int n);
}
```


2.2.6 加载本地库

在静态初始化块内调用 System.loadLibrary 加载本地库，是加载库的最简单的方法。这种块中的代码是在虚拟机加载类时执行的，此时还没调用过任何方法。如果类中有多个方法，但并不是所有的方法都需要把所有的东西初始化（例如，共享库加载），那就会成为一个潜在的性能问题，但种问题并不多见。换句话说，静态初始化块可能会显著增加开销，这是你在用某些特定函数时需要避免的，如代码清单 2-12 所示。

代码清单 2-12 在静态初始化块载入库

```
public class Fibonacci {
    static {
        System.loadLibrary("fibonacci"); // 加载 libfibonacci.so
        // 这里如果做过多费时的事情会将推迟执行 superFast
    }
    public static native long recursiveNative (int n);

    public long superFast (int n) {
        return 42;
    }
}
```

注意 加载库所花费的时间也取决于库本身（例如，其大小和方法数）。

到目前为止，我们已学到混合 Java 和 C/C++的基本知识。本地代码可以提高性能，但 C/C++代码的编译方式也会对性能产生影响。事实上，有许多编译选项，优化效果可能很大程度上取决于使用了哪些选项。

到现在为止我们知道的 Application.mk 和 Android.mk 的 makefile 都是非常基本的，后面两节介绍更多的选项。

2.3 Application.mk

代码清单 2-4 所示 Application.mk 文件只是最基础的版本。这个文件还可以指定更多东西，你可能需要在应用中指定其他一些选项。表 2-3 显示了可以在 Application.mk 定义的各种变量。

更多资源请访问稀酷客(www.ckook.com)

表2-3　Application.mk的变量

变　量	含　义
APP_PROJECT_PATH	项目路径
APP_MODULES	模块编译列表
APP_OPTIM	设置程序为“release”或“debug”版
APP_CFLAGS	C/C++编译器选项
APP_CXXFLAGS	废弃，用APP_CPPFLAGS替代
APP_CPPFLAGS	C++编译器选项
APP_BUILD_SCRIPT	除了jni/Android.mk外，使用其他构建脚本
APP_ABI	编译代码输出的ABI列表
APP_STL	指定使用哪种C++标准模板库，可选择“system”、“stlport_static”、“stlport_shared”或“gnustl_static”
STLPORT_FORCE_REBUILD	如果打算用代码构建STLport而不是使用预编译库，设置为true

你需要关注一下几个对性能有影响的变量：

- APP_OPTIM
- APP_CFLAGS
- APP_CPPFLAGS
- APP_STL
- APP_ABI

APP_OPTIM是可选的，可以设置为“release”或“debug”。如果没有设置，就会根据应用是否为调试模式（在应用程序的manifest文件里设置android:debuggable为true）自动进行设置：如果应用程序设为调试，APP_OPTIM则为“debug”，否则为“release”。当要调试应用时，创建debug版的库是有意义的，默认的行为适用于大多数情况，因此一般不需要在Application.mk显式设置APP_OPTIM。

APP_CFLAGS（C/C++）和APP_CPPFLAGS（只C++）指定传给编译器的选项。这两个选项不一定要指定优化代码的参数，它们可以用于指定include路径以方便找到包含文件（例如，APP_CFLAGS += - I$(LOCAL_PATH)/myincludefiles）。详尽的编译器选项代码清单可参阅gcc文档。与性能相关的最典型选项是O*x*系列，其中*x*指定优化级别，从不优化的0到优化级别3（或者-Os）。不过，在大多数情况下，仅设置APP_OPTIM为“release”，或不设置APP_OPTIM应该就可以了，构建系统可以替你选择优化级别，它产生的结果是可以接受的。

APP_STL用于指定应用应该使用哪个标准库。例如，NDK r6定义了以下4个可选的值：

- system
- stlport_static
- stlport_shared
- gnustl_static

更多资源请访问稀酷客(www.ckook.com)

每个库都有其优缺点。

- 只有 gnustl_static 支持 C++异常和运行时类型信息（RTTI）。STLport 库在 NDK r7 之后才添加了 RTTI 的支持。
- 如果多个本地共享库使用了 C++库，要使用 stlport_shared。（记住：通过调用 System.loadLibrary("stlport_shared")显式加载该库）。
- 如果应用只有一个共享库，最好 stlport_static（以避免动态加载库）。

你可以在 APP_CPPFLAGS 加入-fexceptions 来支持 C++异常，加入-frtti 以支持 RTTI。

2.3.1 为（几乎）所有设备优化

如果应用的性能在很大程度上取决于 C++库的性能，使用不同的库测试应用，选择最好的一个。选择时可不能光看性能，你必须考虑其他因素，如应用的最终大小或 C++库的特性（例如 RTTI）。

我们上面编译的库（libfibonacci.so）构建的是 armeabi ABI，现在出现了以下两个问题：

- 如果只是与 armeabi v7a ABI 兼容，就不能为 Cortex 系列处理器做优化；
- 本地代码与 x86 ABI 不兼容。

Cortex 系列处理器比老的 ARMv5 架构的处理器更加强劲。原因之一是 ARMv7 架构定义了新指令，ARMv5 的库用不了这些指令。编译器生成 armeabi ABI 的库，就要保证不能使用 ARMv5 处理器无法执行的指令。即使库兼容基于 Cortex 的设备，也不能充分利用 CPU 的优势，因而也不会充分发挥其性能潜力。

注意 ARMv7 架构比 ARMv5 强大有很多原因，指令集仅是其中之一。想知道 ARM 各种架构的更多信息，请访问 ARM 网站（http://www.arm.com）。

第二个问题更严重一些，使用 ARM ABI 构建的库不能用在 x86 设备上。如果本地代码功能是应用必需的，那么应用将不能在任何基于 Intel 的 Android 设备上工作。在我们的例子中，System.loadLibrary("Fibonacci")将会失败并抛出 UnsatisfiedLinkError 异常，表示无法加载库。

其实这两个问题很好解决，APP_ABI 可以设置多个需要编译本地代码的 ABI，如代码清单 2-13 所示。通过指定多个 ABI，不仅可以保证为所有架构编译本地代码，而且还为它们逐一进行了优化。

代码清单 2-13 Application.mk 指定三种 ABI

```
APP_ABI := armeabi armeabi-v7a x86
```

使用这个新 Application.mk 重新编译，项目会包含三个子目录。除了 armeabi，包含的两个新子目录名为 armeabi-v7a 和 x86。顾名思义，这两个新目录就是对应应用现在支持的两种新 ABI。这三个目录中每一个都包含了名为 libfibonacci.so 的文件。

更多资源请访问稀酷客(www.ckook.com)

提示　在编辑 Application.mk 或 Android.mk 之后，使用 ndk-build -B V=1 强制重新构建库，显示构建命令。这样，你总是可以在构建过程中验证改动是否达到预期效果。

现在应用文件大了不少，因为它包含三个“相同”的库，每个针对不同的 ABI。Android 的包管理器将在应用安装时，确定在设备上安装哪个库。Android 系统定义了主要 ABI 和可选的次要 ABI。主要 ABI 是首选的 ABI，即包管理器将首先针对主要 ABI 安装库。如果定义了次要 ABI，且主要 ABI 找不到对应的库时，会安装次要 ABI 对应的库。例如，基于 Cortex 的 Android 设备应定义 armeabi v7a 为主要 ABI，armeabi 为次要 ABI。表 2-4 显示了所有设备的主要 ABI 和次要 ABI。

表2-4　主要和次要的ABI

Android系统	主要ABI	次要ABI
基于ARMv5	armeabi	未定义
基于ARMv7	armeabi-v7a	armeabi
基于x86	x86	未定义

次要 ABI 提供了在新 Android 设备与旧应用之间保持兼容性的手段，就像 ARMv7 ABI 全面向前兼容 ARMv5。

注意　Android 系统可能在将来定义更多的主要和次要 ABI，例如，假若 ARM 公司设计了新的 ARMv8 架构，它会向前兼容 ARMv7 和 ARMv5。

2.3.2　支持所有设备

这样做还是有问题。尽管事实上，应用现在可以支持 NDK 所能支持的所有 ABI，Android 可以（几乎非常肯定）被移植到新架构上。例如，前面提到的 MIPS 手机。虽然 Java 宣称“编写一次，随处运行”（字节码是平台独立的，代码不用重新编译就可以支持新平台），但本地代码是针对目标平台，我们生成的 3 个库并不能兼容基于 MIPS 的 Android 系统。可以用以下两种方法解决这个问题：

- 只要 NDK 增加新的 ABI，就重新编译库并发布更新。
- 保留一个 Java 版本实现，用在软件包管理器安装本地代码失败时。

第一个解决方案相当容易，因为它仅涉及安装新 NDK，修改应用的 Application.mk，再重新编译并发布更新（比如在 Android Market）。不过，官方 Android NDK 不是总支持所有 Android 已经移植过的或将被移植的 ABI。因此，建议你还是实现第二个解决方案，提供一个 Java 实现。

更多资源请访问稀酷客(www.ckook.com)

注意 MIPS 科技提供了一个单独的 NDK 的，它允许你构建 MIPS ABI 库。访问 http://developer.mips.com/android 获取更多信息。

代码清单 2-14 给出了加载库失败时，如何用默认的 Java 实现替代。

代码清单 2-14 提供默认的 Java 实现

```
public class Fibonacci {
    private static final boolean useNative;
    static {
        boolean success;
        try {
            System.loadLibrary("fibonacci"); // 加载 libfibonacci.so
            success = true;
        } catch (Throwable e) {
            success = false;
        }
        useNative = success;
    }

     public static long recursive (int n) {
        if (useNative) return recursiveNative(n);
        return recursiveJava(n);
    }

    private static long recursiveJava (int n) {
        if (n > 1) return recursiveJava(n-2) + recursiveJava(n-1);
        return n;
    }

    private static native long recursiveNative (int n);
}
```

另一种设计方案是使用策略模式：

- 定义一个策略接口；
- 定义实现了接口的两个类（使用本地代码，另一种仅使用 Java）；
- 以 System.loadLibrary()的结果为依据实例化适当的类。

代码清单 2-15 是这种变通设计的实现。

代码清单 2-15 使用策略模式提供默认的 Java 实现

```
// FibonacciInterface.java

public interface FibonacciInterface {
     public long recursive (int n);
}

// Fibonacci.java

public final class FibonacciJava implements FibonacciInterface {
     public long recursive(int n) {
         if (n > 1) return recursive(n-2)+recursive(n-1);
```

更多资源请访问稀酷客(www.ckook.com)

```
        return n;
    }
}

// FibonacciNative.java

public final class FibonacciNative implements FibonacciInterface {
    static {
        System.loadLibrary("fibonacci");
    }

    public native long recursive (int n);
}

// Fibonacci.java

public class Fibonacci {
    private static final FibonacciInterface fibStrategy;
    static {
        FibonacciInterface fib;
        try {
            fib = new FibonacciNative();
        } catch (Throwable e) {
            fib = new FibonacciJava();
        }
        fibStrategy = fib;
    }

    public static long recursive (int n) {
        return fibStrategy.recursive(n);
    }
}
```

注意　由于本地函数在 FibonacciNative.java 里声明，而不在 Fibonacci.java，因此需要再次创建本地库，这回使用 com_apress_proandroid_FibonacciNative.c 和 com_apress_proandroid_FibonacciNative.h。Java_com_apress_proandroid_FibonacciNative_recursiveNative 是 Java 调用的函数名。使用以前的库会触发 UnsatisfiedLinkError 异常。

这两个实现就性能而言仅有细微的差别，可以说实现方式基本无影响，可以忽略这部分性能干扰：

- 第一个实现每次调用 recursive()方法都要测试一次；
- 第二个实现需要在静态初始化块里创建一个对象，调用 recursive()时使用的是虚函数。

从设计的角度看，还是建议使用策略模式，原因是：

- 只需一次性选择正确的策略，不用冒忘记 if(useNative)检查的风险；
- 可以很容易地改变策略，仅需修改几行代码；
- 在不同的文件中实现各种策略，维护起来更容易；
- 如果在策略接口添加方法，会强制你在所有实现中都实现这个方法。

更多资源请访问稀酷客(www.ckook.com)

你可以看到，配置 Application.mk 可不是不值一提的任务。不过，你很快就会知道，在大部分情况下，所有应用都用着相同的参数，开发人员通常会复制现有 Application.mk 到新的应用中。

2.4 Android.mk

Application.mk 用于指定整个应用的共同变量，Android.mk 用来指定想构建什么模块以及如何构建，具体细节比较琐碎。表 2-5 列出了 NDK r6 中可用的变量。

表2-5 Android.mk可以定义的变量

变 量	含 义
LOCAL_PATH	Android.mk的路径，可以设置为$(call my-dir)
LOCAL_MODULE	模块名称
LOCAL_MODULE_FILENAME	重新定义库的名称（可选）
LOCAL_SRC_FILES	模块要编译的文件列表
LOCAL_CPP_EXTENSION	重新定义C++源文件的扩展名（默认是.cpp）
LOCAL_C_INCLUDES	追加到include搜索路径的路径列表
LOCAL_CFLAGS	C和C++文件的编译器选项
LOCAL_CXXFLAGS	废弃，用LOCAL_CPPFLAGS替代
LOCAL_CPPFLAGS	C++文件编译器选项
LOCAL_STATIC_LIBRARIES	模块链接的静态库列表
LOCAL_SHARED_LIBRARIES	模块运行时依赖的共享库列表
LOCAL_WHOLE_STATIC_LIBRARIES	和LOCAL_STATIC_LIBRARIES相似，不过使用的是--whole-archive选项
LOCAL_LDLIBS	其他链接选项代码清单（例如，用-lGLESv2链接OpenGL ES 2.0库）
LOCAL_ALLOW_UNDEFINED_SYMBOLS	这个变量设置为true（默认为false）允许未定义的符号
LOCAL_ARM_MODE	编译的指令模式（ARM或Thumb）
LOCAL_ARM_NEON	允许使用NEON高级SIMD指令/内联
LOCAL_DISABLE_NO_EXECUTE	禁用NX位（默认是false，即启用NX）
LOCAL_EXPORT_CFLAGS LOCAL_EXPORT_CPPFLAGS LOCAL_EXPORT_C_INCLUDES LOCAL_EXPORT_LDLIBS	导出变量到依赖此模块的模块（就是在LOCAL_STATIC_LIBRARY或LOCAL_SHARED_LIBRARY里列出此模块）
LOCAL_FILTER_ASM	允许执行shell命令以过滤汇编文件

同样，我们主要关注那几个对性能有影响的变量：

- LOCAL_CFLAGS
- LOCAL_CPPFLAGS
- LOCAL_ARM_MODE
- LOCAL_ARM_NEON
- LOCAL_DISABLE_NO_EXECUTE

更多资源请访问稀酷客(www.ckook.com)

LOCAL_CFLAGS 和 LOCAL_CPPFLAGS 类似于 APP_CFLAGS 和 APP_CPPFLAGS，但只作用于当前模块，而定义在 Application.mk 的选项则用于所有模块。强烈建议不要在 Android.mk 设置优化级别，而是由 Application.mk 的 APP_OPTIM 选项决定。

LOCAL_ARM_MODE 可用来强制编译为 ARM 模式，即用 32 位指令。虽然代码密度比起 Thumb 模式（16 位指令）或许差些，不过 ARM 模式往往比 Thumb 代码快，对性能有提高。例如，Android 自身的 Skia 库明确使用 ARM 模式编译。显然，这仅适用于 ARM ABI，也就是 armeabi 和 armeabi-v7a。例如，如果你要只使用 ARM 模式编译某些特定文件，可以把它们加上.arm 后缀后放入 LOCAL_SRC_FILES，例如把 file.c 改成 file.c.arm。

LOCAL_ARM_NEON 指定是否可以在代码中使用高级 SIMD 指令或内联，编译器是否可以在本地代码中生成 NEON 指令。虽然 NEON 指令可以明显改善性能，可 NEON 指令只有 ARMv7 架构支持，而且还是可选组件。因此，NEON 不能运行在所有设备上。例如，三星 Galaxy Tab 10.1 不支持 NEON，但三星 Nexus S 可以。和 LOCAL_ARM_MODE 一样，可以针对单个文件处理 NEON 支持，只要使用.neon 后缀即可。第 3 章涵盖了 NEON 扩展，并提供了示例代码。

提示　可以在 LOCAL_SRC_FILES 中同时加入.arm 和.neon 后缀，例如，file.c.arm.neon。如果同时使用这两个后缀，需要将.arm 放在前面，否则该文件不会编译。

LOCAL_DISABLE_NO_EXECUTE 本身对性能没有任何影响。但是，专家级开发者使用动态生成代码时（很有可能获得更好的性能）会对禁用 NX 位很感兴趣。这种情况比较罕见，你可能永远不会在 Android.mk 指定默认启用 NX 位选项。禁用 NX 位也被认为存在安全风险。

Android.mk 可以指定多个模块，每个模块可以使用不同的选项和不同的源文件。代码清单 2-16 的 Android.mk 文件中编译两个模块，每个都有不同的选项。

代码清单 2-16　在 Android.mk 中指定两个模块

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := fibonacci
LOCAL_ARM_MODE := thumb
LOCAL_SRC_FILES := com_apress_proandroid_Fibonacci.c fibonacci.c
include $(BUILD_SHARED_LIBRARY)

include $(CLEAR_VARS)
LOCAL_MODULE := fibonarmcci
LOCAL_ARM_MODE := arm
LOCAL_SRC_FILES := com_apress_proandroid_Fibonacci.c fibonacci.c
include $(BUILD_SHARED_LIBRARY)
```

类似 Application.mk，Android.mk 也可以在许多方面进行配置。为这两个文件中的变量选择正确的值，可能是获得良好性能的关键点，而不必耗费更多资源进行高级和复杂的优化。随着 NDK 不断发展，应该参考最新的联机文档，在新版本中可能增加了新的变量，废弃了另一些变量。当新

更多资源请访问稀酷客(www.ckook.com)

NDK 发布时，特别是更新包含新版本的编译器时，建议用新 NDK 重新编译应用，并发布更新。

注意 用不同的工具链（即新的 SDK 或 NDK）重新编译后，一定要再次测试应用程序。

2.5 使用 C/C++改进性能

通过前面几节的学习，你知道了如何结合 Java 和 C/C++代码。你可能觉得使用 C/C++总是比用 Java 性能更好，但事实并非如此。本地代码不是所有性能问题的解决方案。其实，调用本地代码有可能会导致性能下降。这似乎有些出人意料，但从 Java 空间切换到本地空间也不可能没有任何开销。Dalvik JIT 编译器也可以生成本地代码，它可能相当于或优于你自己生成的本地代码。

下面考虑第 1 章 Fibonacci.computeIterativelyFaster()方法的 C 实现，如代码清单 2-17 所示。

代码清单 2-17 斐波那契数列迭代版本的 C 实现

```
uint64_t computeIterativelyFaster (unsigned int n) {
    if (n > 1) {
        uint64_t a, b = 1;
        n--;
        a = n & 1;
        n /= 2;
        while (n-- > 0) {
            a += b;
            b += a;
        }
        return b;
    }
    return n;
}
```

你可看到，C 实现和 Java 非常类似，唯一的区别是使用了无符号类型。你还可以看到代码清单 2-18 所示的 Dalvik 字节码，它非常类似代码清单 2-19 所示的 NDK 的 objdump 工具生成的 ARM 本地代码。此外，NDK 的 objdump 工具还可以用来反编译二进制文件（目标文件、库或可执行文件），并显示汇编助记符。这个工具非常像 dexdump，基本上执行相同的操作，但 dexdump 处理的是.dex 文件（例如，应用的 classes.dex 文件）。

注意 使用 objdump 的-d 选项反编译文件，例如，objdump -d libfibonacci.so。不带任何选项或参数执行 objdump，会列出所有支持的选项。NDK 有两个不同版本的 objdump，一个用于 ARM ABI，另一个用于 x86 ABI。

代码清单 2-18 Fibonacci.iterativeFaster 的 Dalvik 字节码

```
0008e8:                 |[0008e8] com.apress.proandroid.Fibonacci.iterativeFaster:(I)J
0008f8: 1215            |0000: const/4 v5, #int 1 // #1
```

更多资源请访问稀酷客(www.ckook.com)

```
0008fa: 3758 1600  |0001: if-le v8, v5, 0017 // +0016
0008fe: 1602 0100  |0003: const-wide/16 v2, #int 1 // #1
000902: d808 08ff  |0005: add-int/lit8 v8, v8, #int -1 // #ff
000906: dd05 0801  |0007: and-int/lit8 v5, v8, #int 1 // #01
00090a: 8150       |0009: int-to-long v0, v5
00090c: db08 0802  |000a: div-int/lit8 v8, v8, #int 2 // #02
000910: 0184       |000c: move v4, v8
000912: d808 04ff  |000d: add-int/lit8 v8, v4, #int -1 // #ff
000916: 3c04 0400  |000f: if-gtz v4, 0013 // +0004
00091a: 0425       |0011: move-wide v5, v2
00091c: 1005       |0012: return-wide v5
00091e: bb20       |0013: add-long/2addr v0, v2
000920: bb02       |0014: add-long/2addr v2, v0
000922: 0184       |0015: move v4, v8
000924: 28f7       |0016: goto 000d // -0009
000926: 8185       |0017: int-to-long v5, v8
000928: 28fa       |0018: goto 0012 // -0006
```

代码清单 2-19　用 C 实现的 iterativeFaster 的 ARM 汇编代码

```
00000410 <iterativeFaster>:
 410:  e3500001   cmp  r0, #1  ; 0x1
 414:  e92d0030   push  {r4, r5}
 418:  91a02000   movls r2,  r0
 41c:  93a03000   movls r3,  #0  ;  0x0
 420:  9a00000e   bls  460 <iterativeFaster+0x50>
 424:  e2400001   sub  r0, r0, #1  ; 0x1
 428:  e1b010a0   lsrs  r1, r0, #1
 42c:  03a02001   moveq r2,  #1  ;  0x1
 430:  03a03000   moveq r3,  #0  ;  0x0
 434:  0a000009   beq  460 <iterativeFaster+0x50>
 438:  e3a02001   mov  r2, #1  ; 0x1
 43c:  e3a03000   mov  r3, #0  ; 0x0
 440:  e0024000   and  r4, r2, r0
 444:  e3a05000   mov  r5, #0  ; 0x0
 448:  e0944002   adds  r4, r4, r2
 44c:  e0a55003   adc  r5, r5, r3
 450:  e0922004   adds  r2, r2, r4
 454:  e0a33005   adc  r3, r3, r5
 458:  e2511001   subs  r1, r1, #1  ; 0x1
 45c:  1afffff9   bne  448 <iterativeFaster+0x38>
 460:  e1a01003   mov  r1, r3
 464:  e1a00002   mov  r0, r2
 468:  e8bd0030   pop  {r4, r5}
 46c:  e12fff1e   bx  lr
```

注意　请参阅http://infocenter.arm.com 获取 ARM 指令集完整文档。

汇编代码是由 CPU 执行的。Dalvik 字节码看起来很像汇编代码（尽管汇编代码更紧凑），由此推断，Dalvik 的 JIT 编译器生成的本地代码应该是相当接近代码清单 2-19 所示的汇编代码。此

更多资源请访问稀酷客(www.ckook.com)

外，即使字节码和汇编代码非常不同，Dalvik JIT 编译器生成的本地代码仍有可能会和 NDK 生成的汇编代码非常相似。

现在，为了比较这些方法，我们需要运行一些测试。实际的性能评测需要实验证据，下面将测试和比较 4 个项目：

- 没有 JIT 编译器的 Java 实现；
- 有 JIT 编译器的 Java 实现；
- 本地实现（debug 版）；
- 本地实现（release 版）。

测试的框架（Fibonacci.java）如代码清单 2-20 所示，图 2-1 和图 2-2 显示了测试结果。

代码清单 2-20 测试框架

```
static {
    System.loadLibrary("fibonacci_release"); // 使用两个库
    System.loadLibrary("fibonacci_debug");
}

private static final int ITERATIONS = 1000000;

private static long testFibonacci (int n)
{
    long time = System.currentTimeMillis();
    for (int i = 0; i < ITERATIONS; i++) {
        // 调用 iterativeFaster(n), iterativeFasterNativeRelease(n) 或 interativeFasterNativeDebug(n)
        callFibonacciFunctionHere(n);
    }
    time = System.currentTimeMillis() - time;
    Log.i("testFibonacci", String.valueOf(n) + " >> Total time: " + time + " milliseconds");
}

private static void testFibonacci ()
{
    for (int i = 0; i < 92; i++) {
        testFibonacci(i);
    }
}

private static native long iterativeFasterNativeRelease (int n);

private static native long iterativeFasterNativeDebug (int n);
```

图 2-1 是上面 4 项测试的用时（单位：毫秒）。图 2-2 展示了以开启 JIT 编译器的 Java 版本作为基准的 4 项实现的相对性能。

我们可以得出以下结论。

- Dalvik 的 JIT 编译器可以显著提高性能。（JIT 版本比没有 JIT 的版本快 3 到 6 倍。）
- 本地实现并不总是比启用 JIT 的 Java 版本速度更快。
- 在本地空间花费更多的时间，可以摊薄 Java 和本地空间的过渡成本。

更多资源请访问稀酷客(www.ckook.com)

谷歌自己的测试结果显示，Dalvik JIT 编译器可以让 CPU 密集型代码的性能提升 5 倍，我们自己的测试结果确认了这个结果。性能增益依赖代码完成的工作，不应该总是假定这样的比率。如果仍想让应用运行在没有 JIT 版本的 Android（Android 2.1 或更早版本）的旧设备上，这个测量是很重要的。在某些情况下，在旧设备上提供可接受的用户体验，使用本地代码是唯一的选择。

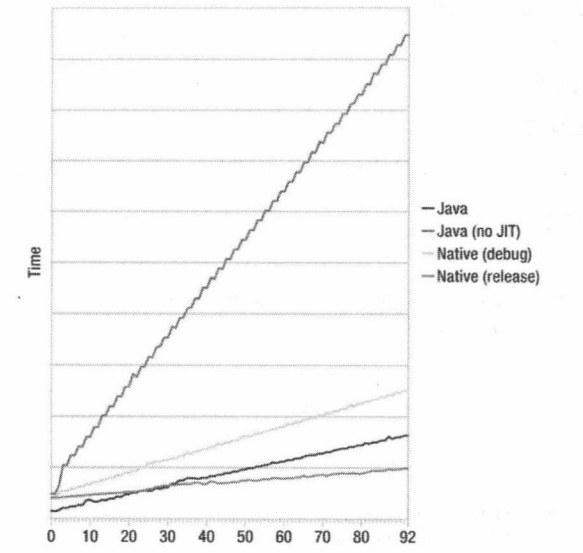


图 2-1　iterativeFaster()不同实现的性能

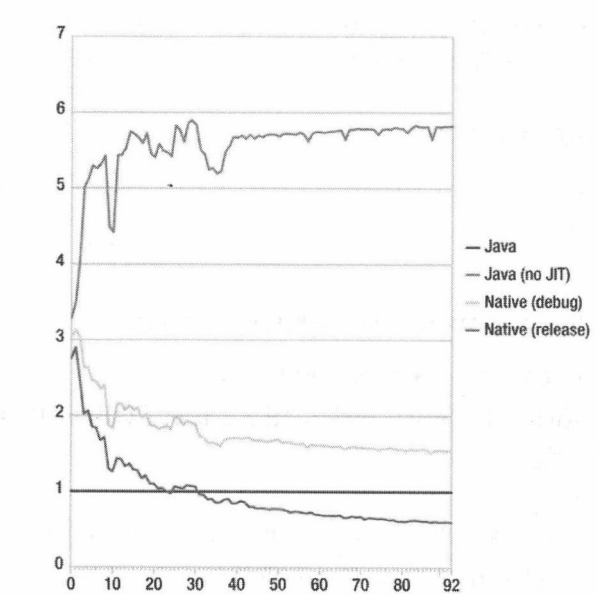


图 2-2　iterativeFaster()相对于启用了 JIT 的 Java 实现的不同实现的性能

更多资源请访问稀酷客(www.ckook.com)

更多有关 JNI 的信息

我们使用的 JNI 粘合层是非常简单的，因为它只是调用另一个 C 函数。不幸的是，使用非原始类型和从本地代码访问 Java 对象或类的域或方法等更复杂的事情都不是这么简单。另一方面，在 JNI 粘合层要做的事情是非常机械的。

1. 字符串

在 Java 和 C/C++使用 String，常会导致性能问题。Java 的 String 使用 16 位 Unicode 字符（UTF-16），而许多 C/C++函数使用 char *作字符串用（即 C/C++中的字符串大部分情况下用 ASCII 或 UTF-8）。怀旧的开发者，甚至可能为了混淆而使用 EBCDIC 编码。也就是说，Java 字符串必须转换成 C/C++字符串才可以使用。代码清单 2-21 是一个简单的示例。

代码清单 2-21 Java 本地方法使用字符串和 JNI 粘合层

```
// Java (Myclass.java 文件中)
public class MyClass {
    public static native void doSomethingWithString (String s);
}

// JNI 粘合层 (C 文件中)
void JNICALL Java_com_apress_proandroid_MyClass_doSomethingWithString
  (JNIEnv *env, jclass clazz, jstring s)
{
    const char* str = (*env)->GetStringUTFChars(env, s, NULL);
    if (str != NULL) {
        // 用 str 字符串在这里做些事情

        // 记得释放字符串，不要犯内存泄漏这个常见错误
        (*env)->ReleaseStringUTFChars(env, s, str);
    }
}
```

JNI 提供了多种方法来处理字符串，它们都用相同的方式完成大量工作：

- Java 字符串必须被转换成 C/C++字符串；
- C/C++字符串必须释放。

表 2-6 显示了 JNI 提供的各种字符串获取/释放方法，并附有简短的说明。

表2-6 JNI中获取/释放字符串的方法

获 取	释 放	说 明
GetStringChars	ReleaseStringChars	获取一个UTF-16字符串（可能需要内存分配）的指针
GetStringUTFChars	ReleaseStringUTFChars	获取一个UTF-8字符串（可能需要内存分配）的指针
GetStringCritical	ReleaseStringCritical	获取一个UTF-16字符串（可能需要内存分配，GetStringCritical和ReleaseStringCritical调用之间做的事情有限制。[①]）的指针
GetStringRegion	n/a	复制字符串的一部分到预先分配的缓冲区(UTF-16格式,没有内存分配)
GetStringUTFRegion	n/a	复制字符串的一部分到预先分配的缓冲区（UTF-8格式，没有内存分配）

① 具体参考规定，如不能进行再次 jni 调用，不能阻塞。——译者注

更多资源请访问稀酷客(www.ckook.com)

内存分配从来不是免费午餐，应该在代码中尽可能使用 GetStringRegion 和 GetStringUTFRegion。这样做，可以得到以下好处：

- 避免可能的内存分配；
- 复制 String 要用的一部分到预先分配的缓冲区（可能在栈上）；
- 不需要释放字符串，避免了忘记释放字符串的风险。

注意　请参阅 JNI 在线文档，NDK 的 jni.h 头文件获取有关其他字符串函数的更多信息。

2. 访问域或方法

你可以从 JNI 粘合层内访问 Java 对象或类的的域和方法，但它不像访问域或调用 C++对象或类的函数那样简单，要通过 id 访问 Java 对象或类的域和方法。访问域或者调用方法时需要：

- 得到这一域或方法的 id；
- 使用 JNI 函数设置/获取域或调用方法。

代码清单 2-22 是一个示例。

代码清单 2-22　在 JNI 粘合层里修改域和调用方法

```
// Java（在 MyClass.java 中）

public class MyClass {
    static {
        System.loadLibrary("mylib");
    }

    public static int someInteger = 0;

    public static native void sayHelloToJNI();

    public static void helloFromJNI() {
        Log.i("MyClass", "Greetings! someInteger=" + someInteger);
    }
}

// JNI 的粘合层（C 文件中）

void JNICALL Java_com_apress_proandroid_MyClass_sayHelloToJNI
  (JNIEnv *env, jclass clazz) {
    // 获取 someInteger 域和 helloFromJNI 方法的 id
    jfieldID someIntegerId = (*env)->GetStaticFieldID(env, clazz, "someInteger", "I");
    jfieldID helloFromJNIId = (*env)->GetStaticMethodID(env, clazz, "helloFromJNI", " ()V");

    // 增加 someInteger
    jint value = (*env)->GetStaticIntField(env, clazz, someIntegerId);
    (*env)->SetStaticIntField(env, clazz, value + 1);

    // 调用 helloFromJNI
    (*env)->CallStaticVoidMethod(env, clazz, helloFromJNIId);
}
```

更多资源请访问稀酷客(www.ckook.com)

出于性能方面的考量，不想每次访问域或调用方法时都去获取一次域或方法的 id。虚拟机加载类时域和方法的 id 就被设置，只要类被加载，该 id 就有效。如果类由虚拟机卸载并再次加载，新的 id 可能与旧的不同。也就是说，高效的方式是在类被加载时获取 id，即在静态初始化块中获取，如代码清单 2-23 所示。

代码清单 2-23 域/方法 id 只获取一次

```
// Java (MyClass.java 中)

public class MyClass {
    static {
        System.loadLibrary("mylib ");
        getIds();//加载类时，只用一次获取到 id
    }

    public static int someInteger = 0;
    public static native void sayHelloToJNI();
    public static void helloFromJNI() {
        Log.i("MyClass", "Greetings! someInteger=" + someInteger);
    }

    private static native void getIds();
}

// JNI 的粘合层 (C 文件中)

static jfieldID someIntegerId;
static jfieldID helloFromJNIId;

void JNICALL Java_com_apress_proandroid_MyClass_sayHelloToJNI
    (JNIEnv *env, jclass clazz) {
    // 不需要在这里拿 id 了

    // 增加 someInteger
    jint value = (*env)->GetStaticIntField(env, clazz, someIntegerId);
    (*env)->SetStaticIntField(env, clazz, value + 1);

    // 调用 helloFromJNI
    (*env)->CallStaticVoidMethod(env, clazz, helloFromJNIId);
}

void JNICALL Java_com_apress_proandroid_MyClass_getIds
    (JNIEnv *env, jclass clazz) {
    // 获取 someInteger 域和 helloFromJNI 方法的 id
    someIntegerId = (*env)->GetStaticFieldID(env, clazz, "someInteger", "I");
    helloFromJNIId = (*env)->GetStaticMethodID(env, clazz, "helloFromJNI", " ()V");
}
```

JNI 定义了大量用于访问域和调用方法的函数。例如，访问整数域和访问布尔域，是用两个不同的函数完成两种操作。同样，调用静态方法和非静态方法也定义了不同的函数。

注意 请参阅 JNI 在线文档和 NDK 的 jni.h 头文件中获取函数的完整列表。

更多资源请访问稀酷客(www.ckook.com)

Android 定义了自己的一套函数和数据结构来访问本机代码中最常使用的类。例如，android/bitmap.h（NDK 4b引入）定义的API可以访问位图对象的像素缓冲区：

- AndroidBitmap_getInfo
- AndroidBitmap_lockPixels
- AndroidBitmap_unlockPixels

NDK r5引入了许多新的API，应用开发人员可以使用本地代码访问Android的Java框架的部分，不用依赖JNI特性（例如JNIEnv、jclass、jobject）。

2.6 本地Acitivity

到目前为止，我们已经看到了如何在单一应用中混合使用Java和C/C++。Android 2.3更进一步，定义了NativeActivity类，它可以用C/C++编写整个应用，但你仍然可以通过JNI访问整个Android的Java框架。

注意　*没必要用NativeActivity编写应用中的所有Activity。例如，可以写包含两个Activity的应用：一个NativeActivity、一个ListActivity。*

如果你喜欢读源文件或头文件，而不是一堆的正式文档，那这对你而言或许是一种享受。事实上，大部分的文档包含在头文件中，可以在NDK的platforms/android-9/arch-arm/usr/include/android目录中找到。表2-7列出了本地Activity使用的头文件。

表2-7　本地Activity使用的头文件

头 文 件	内　容
api-level.h	定义__ANDROID_API__
asset_manager.h	资源管理API
asset_manager_jni.h	转换Java对象（AssetManager）为本地对象的API
bitmap.h	位图API
configuration.h	配置API
input.h	输入API（设备、按键、手势等）
keycodes.h	所有按键代码（如AKEYCODE_SEARCH）的定义
log.h	日志
API looper.h	处理事件的便捷API
native_activity.h	很多东西开始的地方
native_window.h	窗口API
native_window_jni.h	将Java对象（Surface）转换为本地对象的API
obb	Opaque Binary Blob API（见Java的StorageManager）
rect	ARect类型的定义
sensor	传感器API（加速度计、陀螺仪等）
storage_manager.h	存储管理API
window.h	窗口参数（见Java的WindowManager.LayoutParams）

更多资源请访问稀酷客(www.ckook.com)

创建本地 Activity 很简单。第一步是定义应用的 manifest 文件，以便通知 Android 你要使用本地 Activity。你需要在应用的 AndroidManifest.xml 为每个本地 Activity 指定以下内容：

- 要实例化的类；
- 加载库及其入口函数。

第一项和其他非本地 Activity 实际上没有什么不同。当创建 Activity 时，Android 需要实例化一个适当的类，这就是<activity>标签内的 android:name 指定的。在大多数情况下，应用不必继承 NativeActivity 类，基本上使用 android.app.NativeActivity 类进行实例化就可以了。不过，你可以随意实例化自己创建的继承自 NativeActivity 的类。

第二项是让 Android 知道，用什么库包含 Activity 的本地代码，以便让它在 Activity 创建时自动加载。这些信息通过名字/值对的元数据传给 Android：在 android.app.lib_name 中设置，值为指定库的名字，去掉 lib 前缀或.so 后缀。另外，你还可以用名字/值的元数据指定库的入口点，在 android.app.func_name 设置，值为函数名。默认情况下，该函数名为 ANativeActivity_onCreate。

代码清单 2-24 是一个 manifest 文件示例。NativeActivity 是 Android 2.3 引入的，所以最低的 SDK 版本设置为 9，Activity 的本地代码放在 libmyapp.so 中。

代码清单 2-24 本地应用的 AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
package="com.apress.proandroid"
      android:versionCode="1"
      android:versionName="1.0">
    <uses-sdk android:minSdkVersion="9" />

    <application android:icon="@drawable/icon"
                 android:label="@string/app_name"
                 android:hasCode="false">
        <activity android:name="android.app.NativeActivity"
                  android:label="@string/app_name">
            <meta-data android:name="android.app.lib_name"
                       android:value="myapp" />
            <meta-data android:name="android.app.func_name"
                       android:value="ANativeActivity_onCreate" />
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

注意 如果应用不包含任何 Java 代码，你可以在<application>标签中设置 android:hasCode 为 false。

现在启动这个应用会出现 libmyapp.so 不存在的错误。因此，下一步就是构建这个缺失的库。像往常一样使用 NDK 的 ndk-build 工具。

更多资源请访问稀酷客(www.ckook.com)

2.6.1 构建缺失的库

你必须定义 Application.mk 文件以及 Android.mk 文件。当使用本地的 Activity 时，Android.mk 中的不同之处如代码清单 2-25 所示。你还需要另一个文件——myapp.c（如代码清单 2-25 所示），其中包含应用的实现。

代码清单 2-25　本地应用程序的 Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := myapp
LOCAL_SRC_FILES := myapp.c
LOCAL_LDLIBS    := -llog -landroid
LOCAL_STATIC_LIBRARIES := android_native_app_glue

include $(BUILD_SHARED_LIBRARY)
$(call import-module,android/native_app_glue)
```

它和我们之前用的 Android.mk 的区别有：

- 链接的共享库；
- 链接的静态库。

既然本地 Activity 使用 Android 本地应用的 API，就需要添加-landroid 到 LOCAL_LDLIBS。根据要用到的 API 可能会链接更多的库。例如，-llog 链接日志库，可以使用 logcat 调试工具。

Android NDK 提供了简单的方法来创建本地应用，且已经在 NDK 的 native_app_glue 模块中实现。要使用这个模块，你不仅需要把它添加到 LOCAL_STATIC_LIBRARIES，而且需要用 import-module 函数宏将它导入到项目中，如代码清单 2-24 中的最后一行所示。

注意　native_app_glue 模块在 NDK 里实现，源代码位于 android-ndk-r7/sources/android/native_app_glue 目录。可以自由修改其实现，使用修改后的版本作为静态库链接到应用。

代码清单 2-26 是一个应用示例，在单个文件 myapp.c 中实现，它监听以下事件：

- 应用事件（类似于 Java 中的 onStart 方法）；
- 输入事件（按键、运动）；
- 加速度计；
- 陀螺仪（基于回调）。

此应用没做任何有意义的事情，只是打开传感器并说明如何处理传感器事件。在这个特定的例子中，传感器值通过调用__android_log_print 打印出来。你可以把这个应用作为基本框架，在需要时使用它。

更多资源请访问稀酷客(www.ckook.com)

代码清单 2-26 myapp.c 的实现

```
#include <android_native_app_glue.h>
#include <android/sensor.h>
#include <android/log.h>
#define TAG "myapp"

typedef struct {
    // 加速度计
    const ASensor* accelerometer_sensor;
    ASensorEventQueue* accelerometer_event_queue;
    // 陀螺仪
    const ASensor* gyroscope_sensor;
    ASensorEventQueue* gyroscope_event_queue;
} my_user_data_t;

static int32_t on_key_event (struct android_app* app, AInputEvent* event)
{
    // 使用 AKeyEvent_xxx API
    return 0; // 如果想处理这个事件就改为 1
}

static int32_t on_motion_event (struct android_app* app, AInputEvent* event) {
    // 使用 AMotionEvent_xxx API
    return 0; // 如果想处理这个事件就改为 1
}

// 这里只是检查事件类型，并调用相应的功能
static int32_t on_input_event (struct android_app* app, AInputEvent* event)
{
    int32_t type = AInputEvent_getType(event);
    int32_t handled = 0;

    switch (type) {
        case AINPUT_EVENT_TYPE_KEY:
            handled = on_key_event(app, event);
            break;

        case AINPUT_EVENT_TYPE_MOTION:
            handled = on_motion_event(app, event);
            break;
    }
    return handled;
}

// 有些函数尚未实现
static void on_input_changed (struct android_app* app) {}
static void on_init_window (struct android_app* app) {}
static void on_term_window (struct android_app* app) {}
static void on_window_resized (struct android_app* app) {}
static void on_window_redraw_needed (struct android_app* app) {}
static void on_content_rect_changed (struct android_app* app) {}

// 在这里打开传感器
static void on_gained_focus (struct android_app* app) {
    my_user_data_t* user_data = app->userData;
```

更多资源请访问稀酷客(www.ckook.com)

```
    if (user_data->accelerometer_sensor != NULL) {
        ASensorEventQueue_enableSensor(
            user_data->accelerometer_event_queue,
            user_data->accelerometer_sensor);
        ASensorEventQueue_setEventRate(
            user_data->accelerometer_event_queue,
            user_data->accelerometer_sensor, 1000000L/60);
    }
    if (user_data->gyroscope_sensor != NULL) {
        ASensorEventQueue_enableSensor(
            user_data->gyroscope_event_queue,
            user_data->gyroscope_sensor);
        ASensorEventQueue_setEventRate(
            user_data->gyroscope_event_queue,
            user_data->gyroscope_sensor, 1000000L/60);
    }
}

// 失去焦点时禁用传感器
static void on_lost_focus (struct android_app* app)
{   my_user_data_t* user_data = app->userData;
    if (user_data->accelerometer_sensor != NULL) {
        ASensorEventQueue_disableSensor(
            user_data->accelerometer_event_queue,
            user_data->accelerometer_sensor);
    }
    if (user_data->gyroscope_sensor != NULL) {
        ASensorEventQueue_disableSensor(
            user_data->gyroscope_event_queue,
            user_data->gyroscope_sensor);
    }
}

// 更多函数在这里实现...
static void on_config_changed (struct android_app* app) {}
static void on_low_memory (struct android_app* app) {}
static void on_start (struct android_app* app) {}
static void on_resume (struct android_app* app) {}
static void on_save_state (struct android_app* app) {}
static void on_pause (struct android_app* app) {}
static void on_stop (struct android_app* app) {}
static void on_destroy (struct android_app* app) {}

// 这里只是检查命令，调用正确的函数
static void on_app_command (struct android_app* app, int32_t cmd) {
    switch (cmd) {
        case APP_CMD_INPUT_CHANGED:
            on_input_changed(app);
            break;
        case APP_CMD_INIT_WINDOW:
            on_init_window(app);
            break;
        case APP_CMD_TERM_WINDOW:
            on_term_window(app);
            break;
        case APP_CMD_WINDOW_RESIZED:
            on_window_resized(app);
```

更多资源请访问稀酷客(www.ckook.com)

```
            break;
        case APP_CMD_WINDOW_REDRAW_NEEDED:
            on_window_redraw_needed(app);
            break;
        case APP_CMD_CONTENT_RECT_CHANGED:
            on_content_rect_changed(app);
            break;
        case APP_CMD_GAINED_FOCUS:
            on_gained_focus(app);
            break;
        case APP_CMD_LOST_FOCUS:
            on_lost_focus(app);
            break;
        case APP_CMD_CONFIG_CHANGED:
            on_config_changed(app);
            break;
        case APP_CMD_LOW_MEMORY:
            on_low_memory(app);
            break;
        case APP_CMD_START:
            on_start(app);
            break;
        case APP_CMD_RESUME:
            on_resume(app);
            break;
        case APP_CMD_SAVE_STATE:
            on_save_state(app);
            break;
        case APP_CMD_PAUSE:
            on_pause(app);
            break;
        case APP_CMD_STOP:
            on_stop(app);
            break;
        case APP_CMD_DESTROY:
            on_destroy(app);
            break;
    }
}

// 用户定义的循环 id
#define LOOPER_ID_USER_ACCELEROMETER    (LOOPER_ID_USER + 0)
#define LOOPER_ID_USER_GYROSCOPE                (LOOPER_ID_USER + 1)

// 可以一次获取 8 个事件
#define NB_SENSOR_EVENTS 8
static int gyroscope_callback (int fd, int events, void* data)
{
   // 不推荐这里加入任何日志，因为得到的可能比期望的多很多...
    __android_log_write(ANDROID_LOG_INFO, TAG, "gyroscope_callback");
    return 1;
}

static void list_all_sensors (ASensorManager* sm)
{
    ASensorList list;
    int i, n;
```

更多资源请访问稀酷客(www.ckook.com)

```
    n = ASensorManager_getSensorList(sm, & list);
    for (i = 0; i < n; i++) {
        const ASensor* sensor = list[i];
        const char* name = ASensor_getName(sensor);
        const char* vendor = ASensor_getVendor(sensor);
        int type = ASensor_getType(sensor);
        int min_delay = ASensor_getMinDelay(sensor);
        float resolution = ASensor_getResolution(sensor);

        __android_log_print(
            ANDROID_LOG_INFO, TAG, "%s (%s) %d %d %f",name, vendor, type, min_delay, resolution);
    }
}

// 这里开始工作…
void android_main (struct android_app* state)
{
    my_user_data_t user_data;
    ASensorManager* sm = ASensorManager_getInstance();

    app_dummy(); // 不要忘了调用这个

    // 只列出设备上的所有传感器
    list_all_sensors(sm);

    state->userData = & user_data;
    state->onAppCmd = on_app_command;
    state->onInputEvent = on_input_event;

    // 加速度计
    user_data.accelerometer_sensor =
       ASensorManager_getDefaultSensor(sm, ASENSOR_TYPE_ACCELEROMETER);
    user_data.accelerometer_event_queue = ASensorManager_createEventQueue(
        sm, state->looper, LOOPER_ID_USER_ACCELEROMETER, NULL, NULL);

    // 陀螺仪 (基于回调)
    user_data.gyroscope_sensor =
        ASensorManager_getDefaultSensor(sm, ASENSOR_TYPE_GYROSCOPE);
    user_data.gyroscope_event_queue = ASensorManager_createEventQueue(
        sm, state->looper, LOOPER_ID_USER_GYROSCOPE, gyroscope_callback, NULL);

    while (1) {
        int ident;
        int events;
        struct android_poll_source* source;

        while ((ident = ALooper_pollAll(-1, NULL, &events, (void**)&source)) >= 0) {
            // 开始是“标准”的事件
            if ((ident == LOOPER_ID_MAIN) || (ident == LOOPER_ID_INPUT)) {
                //source 不会是空的，但无论如何要检查一下
                if (source != NULL) {
                    // 这将调用 on_app_command 或 on_input_event
                    source->process(source->app, source);
                }
            }
            // 加速度计事件
            if (ident == LOOPER_ID_USER_ACCELEROMETER) {
```

更多资源请访问稀酷客(www.ckook.com)

```
                ASensorEvent sensor_events[NB_SENSOR_EVENTS];
                int i, n;
                while ((n = ASensorEventQueue_getEvents(
                    user_data.accelerometer_event_queue, sensor_events,
NB_SENSOR_EVENTS)) > 0) {
                    for (i = 0; i < n; i++) {
                        ASensorVector* vector = & sensor_events[i].vector;
                        __android_log_print(
                            ANDROID_LOG_INFO, TAG,
                            "%d accelerometer x=%f y=%f z=%f", i, vector->x, vector->y,
vector->z);
                    }
                }
            }

            // 这里处理其他事件
            // 不要忘记检查是否该返回
            if (state->destroyRequested != 0) {
                ASensorManager_destroyEventQueue(sm,
user_data.accelerometer_event_queue);
                ASensorManager_destroyEventQueue(sm, user_data.gyroscope_event_queue);
                return;
            }
        }
        // 当所有的事件已处理完，在这里进行渲染
    }
}
```

2.6.2 替代方案

创建本地应用的另一种方式是实现本地版本的 onCreate，这样你不仅要初始化应用，而且要定义所有其他回调（即等同于 onStart、onResume，等等）。这正是 native_app_glue 模块实现的功能，它可以简化开发。此外，native_app_glue 模块能够保证在一个独立的线程中处理某些事件，让应用保持响应。如果你决定定义自己的 onCreate 实现，就不需要链接 native_app_glue 库，也不用实现 android_main，而是实现 ANativeActivity_onCreate，如代码清单 2-27 所示。

代码清单 2-27 实现 ANativeActivity_onCreate

```
#include <android/native_activity.h>

void ANativeActivity_onCreate (ANativeActivity* activity, void* savedState, size_t savedStateSize)
{
    // 在这里设置所有回调
    activity->callbacks->onStart = my_on_start;
    activity->callbacks->onResume = my_on_resume;
    ...
    // 设置 activity->instance 为一些 instance 定义的数据
    activity->instance = my_own_instance;
//和 userData 相似
//这里没有事件循环，它只是简单地返回，NativeActivity 将调用你定义的回调函数
}
```

更多资源请访问稀酷客(www.ckook.com)

虽然这些看起来简单，当你需要监听其他一些事件（如传感器），并在屏幕上绘制东西时就会复杂很多。

提示　因为 native_app_glue 模块实现了 ANativeActivity_onCreate，如果使用这个模块就不要在 manifest 文件中改变库的入口点的名字。

新的 NativeActivity 类本身不会提高性能。这仅仅是一个使本地应用开发变得更容易的机制。事实上，你可以使用相同的机制，在旧的 Android 版本上写本地应用。尽管应用的某部分（或全部）是由 C/C++编写的，但它仍运行在 Dalvik 虚拟机上，仍依赖 NativeActivity 的 Java 类。

2.7　总结

本章介绍了使用本地代码提高性能的办法。精心雕琢的本地代码很少会降低性能，而使用 NDK 的原因也不只是提升性能。下面总结了使用 NDK 的理由：

- 重用现有代码，而不是在 Java 中重写一切；
- 编写新的代码，在其他不支持 Java 的平台上使用；
- 要在没有 JIT 编译器（Android 2.1 或更早版本）的旧 Android 设备上运行应用，那本地代码是提供顺畅用户体验的唯一出路；
- 即使 Android 设备上已有 JIT 编译器，在应用中使用本地代码还是可以提升用户体验。

前两个原因非常重要，在某些情况下，你可能需要为了它们牺牲性能。（前提是不会影响到用户体验，或至少不会超出用户的忍耐范围。）和许多开发者一样，你可用的资源有限，如果想让尽可能多的人采用你开发的应用，不要局限在单一平台，那不会使你的收益最大化。

更多资源请访问稀酷客(www.ckook.com)

第3章 NDK 进 阶

第2章讲述了如何建立使用Android NDK的项目，还有如何在Android应用中使用C/C++代码。多数情况下，这些就足够用了，不过有时为了进一步压榨性能，需要更深层次的优化。

本章，你需要亲自动手实践，学习如何使用低阶语言来和CPU对话，倾听这颗动力引擎的轰鸣，感受马达的转动，这可不是仅使用纯C/C++代码就能做到的。本章的第一部分给出利用汇编优化函数的实例，并概述了ARM指令集。第二部分给出利用GCC编译器支持的C扩展来提高应用性能的实例。最后，本章总结了一些易见成效的优化代码的简单小技巧。

最新的Android NDK支持的ABI包括armeabi、armeabi-v7a和x86[①]，但本章侧重于前两者，Android主要部署在ARM设备上。如果你打算写汇编代码，应该首选ARM。虽然第一个Google TV设备是基于Intel的，但Google TV还不支持NDK。

3.1 汇编

NDK支持在Android应用里使用C/C++。第2章展示了C/C++代码编译后的汇编代码，以及如何使用objdump-d反汇编文件(目标文件或库)。在代码清单3-1中再看一遍computeIteratively-Faster的ARM汇编代码。[②]

代码清单3-1 C实现的computeIterativelyFaster的ARM汇编代码

```
00000410 < computeIterativelyFaster>:
 410:   e3500001        cmp     r0, #1  ; 0x1
 414:   e92d0030        push    {r4, r5}
 418:   91a02000        movls   r2, r0
 41c:   93a03000        movls   r3, #0  ; 0x0
 420:   9a00000e        bls     460 <computeIterativelyFaster+0x50>
 424:   e2400001        sub     r0, r0, #1      ; 0x1
 428:   e1b010a0        lsrs    r1, r0, #1
 42c:   03a02001        moveq   r2, #1  ; 0x1
 430:   03a03000        moveq   r3, #0  ; 0x0
 434:   0a000009        beq     460 < computeIterativelyFaster+0x50>
 438:   e3a02001        mov     r2, #1  ; 0x1
```

① 最新的r8已支持mips。——译者注

② gcc explorer http://gcc.godbolt.org/ 可以在网上看反汇编。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
43c:   e3a03000        mov     r3, #0  ; 0x0
440:   e0024000        and     r4, r2, r0
444:   e3a05000        mov     r5, #0  ; 0x0
448:   e0944002        adds    r4, r4, r2
44c:   e0a55003        adc     r5, r5, r3
450:   e0922004        adds    r2, r2, r4
454:   e0a33005        adc     r3, r3, r5
458:   e2511001        subs    r1, r1, #1      ; 0x1
45c:   1afffff9        bne     448 < computeIterativelyFaster+0x38>
460:   e1a01003        mov     r1, r3
464:   e1a00002        mov     r0, r2
468:   e8bd0030        pop     {r4, r5}
46c:   e12fff1e                bx      lr
```

NDK不仅支持在应用中使用C和C++代码，还能让你直接写汇编代码。严格说来，支持编写汇编代码不是NDK独有的功能，而是Android NDK内置的GCC编译器的功能。因此，本章所学的几乎所有东西，还可以应用到其他项目上，例如iOS平台上的应用。

你可以看到，阅读汇编代码挺费力，更不用说动手写。不过，在理解了汇编代码之后，你将会如庖丁解牛般深入程序的肌理，优化应用游刃有余，另外还可以在需要时拿出来炫耀一下。

先用下面三个简单的例子来热热身：

- 计算最大公约数；
- 转换颜色格式；
- 并行按8位计算平均值。

这几个例子就是汇编新手也能完成，但从它们身上可总结出汇编优化的重要原则。这些例子只介绍了可用指令集的一个子集，后面会有ARM指令集的更完整的介绍，以及功能强大的ARM SIMD指令简介。最后，你会学到动态检查CPU具体可用功能的方法，如果用到的特性并不是所有设备都具备时，这一步必不可少。

3.1.1　最大公约数

两个非零整数的最大公约数（GCD）是它们都可以整除的最大正整数。例如，10和55的最大公约数是5。计算两整数最大公约数的函数实现如代码清单3-2所示。

代码清单3-2　最大公约数简单的实现

```
unsigned int gcd (unsigned int a, unsigned int b) {
    // a和b必须都不等于0 （否则，就干看着无限循环吧!）

    while (a != b) {
        if (a > b) {
            a -= b;
        } else {
            b -= a;
        }
    }
    return a;
}
```

更多资源请访问稀酷客(www.ckook.com)

如果在应用的 Application.mk 文件里定义了 APP_ABI 支持 x86、armeabi、armeabi-v7 架构，会得到三种不同的库，反编译时也要生成对应的三份不同的汇编代码。不过，因为还可以选择编译 armeabi 和 armeabi-v7a ABI 为 ARM 或 Thumb 模式，实际上要看 5 份汇编代码。

提示 NDK r7 开始可以设置 APP_ABI 为 "all"（APP_ABI := all），从而不用为每个库指定独立的 ABI。如果 NDK 支持了新的 ABI，只需执行 ndk-build，无需修改 Application.mk。

代码清单 3-3 是 x86 的汇编代码、代码清单 3-4 是 armv5、代码清单 3-5 是 ARMv7。因为不同版本的编译器可能输出不同的代码，你看到的代码可能与示例显示的略有不同。生成的代码也依赖于优化级别及其他设置选项。

代码清单 3-3 x86 汇编代码

```
00000000 <gcd>:
   0:    8b 54 24 04     mov    0x4(%esp),%edx
   4:    8b 44 24 08     mov    0x8(%esp),%eax
   8:    39 c2           cmp    %eax,%edx
   a:    75 0a           jne    16 <gcd+0x16>
   c:    eb 12           jmp    20 <gcd+0x20>
   e:    66 90           xchg   %ax,%ax
  10:    29 c2           sub    %eax,%edx
  12:    39 d0           cmp    %edx,%eax
  14:    74 0a           je     20 <gcd+0x20>
  16:    39 d0           cmp    %edx,%eax
  18:    72 f6           jb     10 <gcd+0x10>
  1a:    29 d0           sub    %edx,%eax
  1c:    39 d0           cmp    %edx,%eax
  1e:    75 f6           jne    16 <gcd+0x16>
  20:    f3 c3           repz ret
```

如果熟悉 x86 汇编助记符，可以看到这段代码大量使用了跳转指令（jne、jmp、je、jb）。此外，大多数指令是 16 位的（例如，f3 c3），有些是 32 位的。

注意 请确保使用正确版本的 objdump 反编译目标文件和库。例如，使用 ARM 版本的 objdump 反编译 x86 的目标文件，会出现如下消息：

```
arm-linux-androideabi-objdump: Can't disassemble for architecture UNKNOWN!
```

代码清单 3-4 RMv5 汇编代码（ARM 模式）

```
00000000 <gcd>:
   0:    e1500001            cmp    r0, r1
   4:    e1a03000            mov    r3, r0
   8:    0a000004            beq    20 <gcd+0x20>
   c:    e1510003            cmp    r1, r3
  10:    30613003            rsbcc    r3, r1, r3
```

更多资源请访问稀酷客(www.ckook.com)

```
  14:    20631001            rsbcs   r1, r3, r1
  18:    e1510003            cmp    r1, r3
  1c:    1afffffa            bne    c <gcd+0xc>
  20:    e1a00001            mov    r0, r1
  24:    e12fff1e            bx    lr
```

代码清单 3-5　RMv7a 汇编代码（ARM 模式）

```
00000000 <gcd>:
   0:    e1500001            cmp    r0, r1
   4:    e1a03000            mov    r3, r0
   8:    0a000004            beq    20 <gcd+0x20>
   c:    e1510003            cmp    r1, r3
  10:    30613003            rsbcc   r3, r1, r3
  14:    20631001            rsbcs   r1, r3, r1
  18:    e1510003            cmp    r1, r3
  1c:    1afffffa            bne    c <gcd+0xc>
  20:    e1a00001            mov    r0, r1
  24:    e12fff1e            bx    lr
```

从以上结果可看出，ARM 模式下 GCC 编译器编译代码清单 3-2 的代码，armeabi 和 armeabi -v7a 两个 ABI 生成的汇编代码是相同的。虽然编译器通常会利用新 ABI 加入的新指令编译，但并不总是如此。

因为除了 ARM 模式，你还可以选择 Thumb 模式，所以接下来看看在 Thumb 模式下生成的汇编代码。代码清单 3-6 是 ARMv5 汇编代码（Application.mk 设置 armeabi ABI），而代码清单 3-7 是 ARMv7 汇编代码（Application.mk 设置 armeabi-v7a ABI）。

代码清单 3-6　RMv5 汇编代码（Thumb 模式）

```
00000000 <gcd>:
   0:    1c03          adds    r3, r0, #0
   2:    428b          cmp    r3, r1
   4:    d004          beq.n    10 <gcd+0x10>
   6:    4299          cmp    r1, r3
   8:    d204          bcs.n    14 <gcd+0x14>
   a:    1a5b          subs    r3, r3, r1
   c:    428b          cmp    r3, r1
   e:    d1fa          bne.n    6 <gcd+0x6>
  10:    1c08          adds    r0, r1, #0
  12:    4770          bx    lr
  14:    1ac9          subs    r1, r1, r3
  16:    e7f4          b.n    2 <gcd+0x2>
```

在代码清单 3-6 中的所有指令都是 16 位（如 e7f4，上述最后一条指令），12 条指令共需要 24 字节空间。

代码清单 3-7　RMv7 体系汇编代码（Thumb 模式）

```
00000000 <gcd>:
   0:    4288          cmp    r0, r1
   2:    4603          mov    r3, r0
   4:    d007          beq.n    16 <gcd+0x16>
```

更多资源请访问稀酷客(www.ckook.com)

```
 6:    4299          cmp    r1, r3
 8:    bf34          ite    cc
 a:    ebc1 0303     rsbcc    r3, r1, r3
 e:    ebc3 0101     rsbcs    r1, r3, r1
12:    4299          cmp    r1, r3
14:    d1f7          bne.n    6 <gcd+0x6>
16:    4608          mov    r0, r1
18:    4770          bx    lr
1a:    bf00          nop
```

这次的两份清单不同了，ARMv5 架构使用的是 Thumb 指令集（全部是 16 位指令），ARMv7 架构却支持 Thumb2 指令集，指令分 16 位或 32 位。

其实代码清单 3-7 看起来和代码清单 3-5 还是有很多相似之处的。最大的区别是，代码清单 3-7 用了 ite（if-then-else）指令，事实上，ARM 模式指令总长度为 40 字节，而 Thumb2 只有 28 字节。

注意 尽管 ARM 架构是主流的，但读一读 x86 汇编代码也不会有什么损失。

通常情况下，GCC 编译器的工作非常理想，大可对它生成的代码放心。不过，如果自己写的 C/C++代码变成了应用中的一个瓶颈，就需要仔细检查编译器生成的汇编代码，来决定是否需要手写。很多时候，编译器生成的代码质量很高，你不会比它做得更好。话虽这么说，在有些情况下，如果对指令集了解透彻，苦干加巧干，可以做得比编译器更好。

注意 切记首先考虑修改 C/C++代码来得到更好的性能，这可比用汇编代码重写容易多了。

事实上，GCD 函数可以换种实现，速度更快，指令也更紧凑，如代码清单 3-8 所示。

代码清单 3-8 手工打造的汇编代码

```
.global gcd_asm
.func gcd_asm

gcd_asm:
    cmp   r0, r1
    subgt r0, r0, r1
    sublt r1, r1, r0
    bne   gcd_asm
    bx    lr
.endfunc
.end
```

不包括最后的函数返回指令，该算法的核心实现只有 4 条指令。从测量结果也能看出执行速度变快了。注意对比代码清单 3-8 CMP 指令的单次调用和代码清单 3-7 中的两次调用。

此代码可复制到名为 gcd_asm.S 的文件中，并添加到 Android.mk 编译文件列表中。此文件使用 ARM 指令，显然不能编译为 x86 ABI。在应确保该文件与相应的 ABI 兼容时才可以放入 Android.mk 的编译文件列表中。代码清单 3-9 是为此修改 Android.mk 的示例。

更多资源请访问稀酷客(www.ckook.com)

代码清单 3-9　Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := chapter3
LOCAL_SRC_FILES := gcd.c

ifeq ($(TARGET_ARCH_ABI),armeabi)
LOCAL_SRC_FILES += gcd_asm.S
endif

ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
LOCAL_SRC_FILES += gcd_asm.S
endif

include $(BUILD_SHARED_LIBRARY)
```

gcd_asm.S 既然已用汇编代码编写，生成的目标文件应该和源文件看起来极其相似。代码清单 3-10 是反汇编代码，事实上，反汇编代码和源代码几乎相同。

代码清单 3-10　反编译 gcd_asm 代码

```
00000000 <gcd_asm>:
   0:    e1500001        cmp    r0, r1
   4:    c0400001        subgt    r0, r0, r1
   8:    b0411000        sublt    r1, r1, r0
   c:    1afffffb        bne    0 <gcd_asm>
  10:    e12fff1e        bx    lr
```

注意　汇编器可能在某些情况下会替换一些指令，所以你可能会观察到你写的代码与反汇编代码之间有细微的差别。

通过简化汇编代码，我们取得了更好的效果，可维护性也没变差。

3.1.2　色彩转换

图形程序中常见的操作是把颜色从一种格式转换为另一种。例如，颜色用有四个 8 位通道（透明，红色，绿色和蓝色）的 32 位值表示，可以转换为用有三个通道（红色 5 位，6 位为绿色，蓝色 5 位，没有 alpha）的 16 位值表示。这两种格式一般分别称为 ARGB8888 和 RGB565。代码清单 3-11 是这种转换的详细实现。

代码清单 3-11　色彩转换功能函数的实现

```
unsigned int argb888_to_rgb565 (unsigned int color) {
    /*
        input:  aaaaaaaarrrrrrrrggggggggbbbbbbbb
        output: 0000000000000000rrrrrggggggbbbbb
    */
```

更多资源请访问稀酷客(www.ckook.com)

```
    return
        /* red   */ ((color >> 8) & 0xF800) |
        /* green */ ((color >> 5) & 0x07E0) |
        /* blue  */ ((color >> 3) & 0x001F);
}
```

再啰嗦一次，有 5 份汇编代码可供分析。代码清单 3-12 是 x86 汇编代码、代码清单 3-13 是 ARM 模式的 ARMv5、代码清单 3-14 是 ARM 模式下的 ARMv7、代码清单 3-15 是在 Thumb 模式下的 ARMv5、代码清单 3-16 是在 Thumb 模式下的 ARMv7。

代码清单 3-12 x86 汇编代码

```
00000000 <argb8888_to_rgb56577>:
   0:    8b 54 24 04             mov    0x4(%esp),%edx
   4:    89 d0                   mov    %edx,%eax
   6:    89 d1                   mov    %edx,%ecx
   8:    c1 e8 05                shr    $0x5,%eax
   b:    c1 e9 08                shr    $0x8,%ecx
   e:    25 e0 07 00 00          and    $0x7e0,%eax
  13:    81 e1 00 f8 00 00       and    $0xf800,%ecx
  19:    c1 ea 03                shr    $0x3,%edx
  1c:    09 c8                   or     %ecx,%eax
  1e:    83 e2 1f                and    $0x1f,%edx
  21:    09 d0                   or     %edx,%eax
  23:    c3                      ret
```

代码清单 3-13 ARMv5 汇编代码（ARM 模式）

```
00000000 <argb8888_to_rgb565>:
   0:    e1a022a0                lsr    r2, r0, #5
   4:    e1a03420                lsr    r3, r0, #8
   8:    e2022e7e                and    r2, r2, #2016    ; 0x7e0
   c:    e2033b3e                and    r3, r3, #63488    ; 0xf800
  10:    e1a00c00                lsl    r0, r0, #24
  14:    e1823003                orr    r3, r2, r3
  18:    e1830da0                orr    r0, r3, r0, lsr #27
  1c:    e12fff1e                bx    lr
```

代码清单 3-14 ARMv7 汇编代码（ARM 模式）

```
00000000 <argb8888_to_rgb565>:
   0:    e7e431d0                ubfx    r3, r0, #3, #5
   4:    e1a022a0                lsr    r2, r0, #5
   8:    e1a00420                lsr    r0, r0, #8
   c:    e2022e7e                and    r2, r2, #2016    ; 0x7e0
  10:    e2000b3e                and    r0, r0, #63488    ; 0xf800
  14:    e1820000                orr    r0, r2, r0
  18:    e1800003                orr    r0, r0, r3
  1c:    e12fff1e                bx    lr
```

代码清单 3-15 ARMv5 汇编代码（Thumb 模式）

```
00000000 <argb8888_to_rgb565>:
   0:    23fc          movs    r3, #252
   2:    0941          lsrs    r1, r0, #5
```

更多资源请访问稀酷客(www.ckook.com)

```
   4:    00db          lsls    r3, r3, #3
   6:    4019          ands    r1, r3
   8:    23f8          movs    r3, #248
   a:    0a02          lsrs    r2, r0, #8
   c:    021b          lsls    r3, r3, #8
   e:    401a          ands    r2, r3
  10:    1c0b          adds    r3, r1, #0
  12:    4313          orrs    r3, r2
  14:    0600          lsls    r0, r0, #24
  16:    0ec2          lsrs    r2, r0, #27
  18:    1c18          adds    r0, r3, #0
  1a:    4310          orrs    r0, r2
  1c:    4770          bx      lr
  1e:    46c0          nop             (mov r8, r8)
```

代码清单 3-16　ARMv7 汇编代码（Thumb 模式）

```
00000000 <argb888_to_rgb565>:
   0:    0942          lsrs    r2, r0, #5
   2:    0a03          lsrs    r3, r0, #8
   4:    f402 62fc     and.w    r2, r2, #2016    ; 0x7e0
   8:    f403 4378     and.w    r3, r3, #63488    ; 0xf800
   c:    4313          orrs    r3, r2
   e:    f3c0 00c4     ubfx    r0, r0, #3, #5
  12:    4318          orrs    r0, r3
  14:    4770          bx      lr
  16:    bf00          nop
```

根据生成多少指令来判断，Thumb 模式的 ARMv5 代码似乎是效率最低的。不过统计指令条数不是判断代码快慢的准确方法。为了得到较接近的估计时间值，还要计入每条指令花费的时钟周期。例如，执行“orr r3, r2”指令只需要一个周期。对现在的 CPU 来说，计算最终会需要多少个时钟周期非常困难，因为 CPU 可以在每个时钟周期执行几条指令，甚至在某些情况下，为了最大化吞吐量还会乱序执行。

注意　参阅 Cortex-A9 技术参考手册，了解更多有关指令周期计时的信息。

现在，可以用 UBFX 和 BFI 指令来实现同样的转换函数，这个版本略有不同，如代码清单 3-17 所示。

代码清单 3-17　手工打造的汇编代码

```
.global argb8888_ro_rgb565_asm
.func argb8888_ro_rgb565_asm

argb8888_ro_rgb565_asm:
    // r0=aaaaaaaarrrrrrrrggggggggbbbbbbbb
    // r1=undefined (scratch register)

    ubfx r1, r0, #3, #5

    // r1=00000000000000000000000000bbbbb
```

更多资源请访问稀酷客(www.ckook.com)

```
    lsr r0, r0, #10

    // r0=0000000000aaaaaaaarrrrrrrrgggggg

    bfi r1, r0, #5, #6

    // r1=00000000000000000000ggggggbbbbb

    lsr r0, r0, #9

    // r0=0000000000000000000aaaaaaaarrrrr

    bfi r1, r0, #11, #5

    // r1=0000000000000000rrrrrggggggbbbbb

    mov r0, r1

    // r0=0000000000000000rrrrrggggggbbbbb

    bx    lr
.endfunc
.end
```

此代码使用 UBFX 和 BFI 指令（在 ARMv6T2 架构才有），不能编译为 armeabi ABI（ARMV5），显然也不能编译为 x86 ABI。

如代码清单 3-9 所示，Android.mk 应确保该文件只在对应的 ABI 编译文件列表里。代码清单 3-18 里把 rgb.c 和 rgb_asm.S 加到了编译文件列表中。

代码清单 3-18 Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := chapter3
LOCAL_SRC_FILES := gcd.c rgb.c

ifeq ($(TARGET_ARCH_ABI),armeabi)
LOCAL_SRC_FILES += gcd_asm.S
endif

ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
LOCAL_SRC_FILES += gcd_asm.S rgb_asm.S
endif

include $(BUILD_SHARED_LIBRARY)
```

如果把 rgb_asm.S 加到 armeabi ABI 的编译文件列表中，就会得到以下错误：

```
Error: selected processor does not support `ubfx r1,r0,#3,#5'
Error: selected processor does not support `bfi r1,r0,#5,#6'
Error: selected processor does not support `bfi r1,r0,#11,#5'
```

更多资源请访问稀酷客(www.ckook.com)

3.1.3　并行计算平均值

在下面这个例子中，我们把每个 32 位值看作 4 个独立的 8 位值并按字节计算这样两个值的平均值。例如，0x10FF3040 和 0x50FF7000 的平均值是 0x30FF5020 （0x10 和 0x50 的平均值为 0x30，0xFF 的和 0xFF 的平均值是 0xFF）。函数实现如代码清单 3-19 所示。

代码清单 3-19　并行计算平均值函数的实现

```
unsigned int avg8 (unsigned int a, unsigned int b)
{
    return
        ((a >> 1) & 0x7F7F7F7F) +
        ((b >> 1) & 0x7F7F7F7F) +
        (a & b & 0x01010101);
}
```

与前两个例子一样，共有 5 份汇编代码如代码清单 3-20 到代码清单 3-24 所示。

代码清单 3-20　x86 汇编代码

```
00000000 <avg8>:
   0:    8b 54 24 04           mov    0x4(%esp),%edx
   4:    8b 44 24 08           mov    0x8(%esp),%eax
   8:    89 d1                 mov    %edx,%ecx
   a:    81 e1 01 01 01 01     and    $0x1010101,%ecx
  10:    d1 ea                 shr    %edx
  12:    21 c1                 and    %eax,%ecx
  14:    81 e2 7f 7f 7f 7f     and    $0x7f7f7f7f,%edx
  1a:    d1 e8                 shr    %eax
  1c:    8d 14 11              lea    (%ecx,%edx,1),%edx
  1f:    25 7f 7f 7f 7f        and    $0x7f7f7f7f,%eax
  24:    8d 04 02              lea    (%edx,%eax,1),%eax
  27:    c3                    ret
```

代码清单 3-21　ARMv5 汇编代码（ARM 模式）

```
00000000 <avg8>:
   0:    e59f301c              ldr    r3, [pc, #28]    ; 24 <avg8+0x24>
   4:    e59f201c              ldr    r2, [pc, #28]    ; 28 <avg8+0x28>
   8:    e0003003              and    r3, r0, r3
   c:    e0033001              and    r3, r3, r1
  10:    e00200a0              and    r0, r2, r0, lsr #1
  14:    e0830000              add    r0, r3, r0
  18:    e00220a1              and    r2, r2, r1, lsr #1
  1c:    e0800002              add    r0, r0, r2
  20:    e12fff1e              bx     lr
  24:    01010101              .word  0x01010101
  28:    7f7f7f7f              .word  0x7f7f7f7f
```

ARMv5 MOV 指令不能将值复制到寄存器，需要改用 LDR 指令来复制 0x01010101 到寄存器 r3。同理，用 LDR 指令来复制 0x7f7f7f7f 到 r2。

更多资源请访问稀酷客(www.ckook.com)

代码清单 3-22 ARMv7 体系汇编代码（ARM 模式）

```
00000000 <avg8>:
   0:   e3003101        movw    r3, #257    ; 0x101
   4:   e3072f7f        movw    r2, #32639    ; 0x7f7f
   8:   e3403101        movt    r3, #257    ; 0x101
   c:   e3472f7f        movt    r2, #32639    ; 0x7f7f
  10:   e0003003        and     r3, r0, r3
  14:   e00200a0        and     r0, r2, r0, lsr #1
  18:   e0033001        and     r3, r3, r1
  1c:   e00220a1        and     r2, r2, r1, lsr #1
  20:   e0830000        add     r0, r3, r0
  24:   e0800002        add     r0, r0, r2
  28:   e12fff1e        bx      lr
```

不用 LDR 指令复制 0x01010101 到 r3，ARMv7 代码使用两个 MOV 指令：第一个，MOVW 用来复制 16 位值（0x0101）到 r3 的低 16 位，而第二个 MOVT 是复制 0x0101 到 r3 的高 16 位。执行这两个指令后，r3 就会包含值 0x01010101。其余部分的汇编代码看起来和 ARMv5 一样。

代码清单 3-23 ARMv5 汇编代码（Thumb 模式）

```
00000000 <avg8>:
   0:   b510            push    {r4, lr}
   2:   4c05            ldr     r4, [pc, #20]    (18 <avg8+0x18>)
   4:   4b05            ldr     r3, [pc, #20]    (1c <avg8+0x1c>)
   6:   4004            ands    r4, r0
   8:   0840            lsrs    r0, r0, #1
   a:   4018            ands    r0, r3
   c:   400c            ands    r4, r1
   e:   1822            adds    r2, r4, r0
  10:   0848            lsrs    r0, r1, #1
  12:   4003            ands    r3, r0
  14:   18d0            adds    r0, r2, r3
  16:   bd10            pop     {r4, pc}
  18:   01010101        .word   0x01010101
  1c:   7f7f7f7f        .word   0x7f7f7f7f
```

代码使用 r4 寄存器，需要保存到栈上之后恢复。

代码清单 3-24 ARMv7 体系汇编代码（Thumb 模式）

```
00000000 <avg8>:
   0:   f000 3301       and.w   r3, r0, #16843009    ; 0x1010101
   4:   0840            lsrs    r0, r0, #1
   6:   400b            ands    r3, r1
   8:   f000 307f       and.w   r0, r0, #2139062143    ; 0x7f7f7f7f
   c:   0849            lsrs    r1, r1, #1
   e:   1818            adds    r0, r3, r0
  10:   f001 317f       and.w   r1, r1, #2139062143    ; 0x7f7f7f7f
  14:   1840            adds    r0, r0, r1
  16:   4770            bx      lr
```

更多资源请访问稀酷客(www.ckook.com)

Thumb2 汇编代码更紧凑，仅用一条指令就把 0x01010101 和 0x7f7f7f7f 复制到 r3 和 r0。

在决定写优化的汇编代码之前，先要停下来想想如何优化 C 代码。经过短暂思考，你可能得到如代码清单 3-25 所示的结果。

代码清单 3-25　更快地并行计算平均值的函数

```
unsigned int avg8_faster (unsigned int a, unsigned int b)
{
    return (((a ^ b) & 0xFEFEFEFE) >> 1) + (a & b);
}
```

与 C 代码的第一个版本比，这段代码更紧凑，看起来速度更快。第一个版本使用了两个>>、4 个 & 和两个+操作（共 8 个“基本”操作），而新版本只使用了 5 个“基本”的操作。直觉上判断第二个实现应该会更快，事实的确如此。

代码清单 3-26 是相应的 ARMv7 Thumb 汇编代码。

代码清单 3-26　ARMv7 汇编代码（Thumb 模式）

```
00000000 <avg8_faster>:
   0:    ea81 0300      eor.w    r3, r1, r0
   4:    4001           ands     r1, r0
   6:    f003 33fe      and.w    r3, r3, #4278124286    ; 0xfefefefe
   a:    eb01 0053      add.w    r0, r1, r3, lsr #1
   e:    4770           bx       lr
```

这个更快的实现产生了更快、更紧凑的代码（不包括函数返回指令，从 8 条减到 4 条）。

这听起来很了不起，不过仔细查看 ARM 指令集，还可以找到 UHADD8 指令，它执行无符号的按字节加，最后将结果减半，这恰好正是我们想要的计算结果。接下来，更快的版本也出炉了，如代码清单 3-27 所示。

代码清单 3-27　手工打造的汇编代码

```
.global avg8_asm
.func avg8_asm

avg8_asm:
    uhadd8 r0, r0, r1
    bx     lr
.endfunc
.end
```

还可以采用其他“并行指令”。如 UHADD16，类似按字节加的 UHADD8，不过它是按半字加。这些指令可以显著提高性能，但生成利用这些指令的代码对现在的编译器来说还是个艰难的任务，要使用它们往往还得自己动手。

注意　ARMv6 架构中引入并行指令，它们无法编译为 armeabi ABI（ARMV5）。

更多资源请访问稀酷客(www.ckook.com)

整段地用汇编代码写函数，的确是个漫长的磨难。大多数情况下，只有函数的一部分会受益于汇编代码，而其余完全可以用 C/C++来写。GCC 编译器支持在 C/C++中混合使用汇编，如代码清单 3-28 所示。

代码清单 3-28 汇编代码与 C 代码混合

```
unsigned int avg8_fastest (unsigned int a, unsigned int b)
{
#if defined(__ARM_ARCH_7A__)
    unsigned int avg;

    asm("uhadd8 %[average], %[val1], %[val2]"
        : [average] "=r" (avg)
        : [val1] "r" (a), [val2] "r" (b));
    return avg;
#else
   return (((a ^ b) & 0xFEFEFEFE) >> 1) + (a & b); // default generic implementation
#endif
}
```

注意 访问http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html获取有关扩展 ASM 的详细信息，访问http://gcc.gnu.org/onlinedocs/gcc/Constraints.html获取约定的更多细节。一条 asm()语句可以包含多条指令。

更新后的 Android.mk 见代码清单 3-29。

代码清单 3-29 Android.mk

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := chapter3
LOCAL_SRC_FILES := gcd.c rgb.c avg8.c

ifeq ($(TARGET_ARCH_ABI),armeabi)
LOCAL_SRC_FILES += gcd_asm.S
endif

ifeq ($(TARGET_ARCH_ABI),armeabi-v7a)
LOCAL_SRC_FILES += gcd_asm.S rgb_asm.S avg8_asm.S
endif

include $(BUILD\_SHARED\_LIBRARY)
```

这个例子表明，有时候，需要丰富的指令集知识才能获得最佳的性能。由于 Android 设备大多是基于 ARM 架构的，你应该重点关注 ARM 指令集。ARM 网站上提供的 ARM 文档（infocenter.arm.com）质量很高，请充分利用它们。

更多资源请访问稀酷客(www.ckook.com)

3.1.4 ARM 指令

ARM 指令很丰富，我们这里的目标并不是写个描述每个指令的详细文档，表 3-1 是可用的 ARM 指令列表，每条都附有简要说明。

熟悉它们后，就会知道其中有些相对常用，有些虽然比较难懂，但在提高性能时往往表现出色。例如，ADD 和 MOV 实际中随处可见，而 SETEND 指令是生面孔，不过当你需要访问不同字节位置的数据时它会给你意外的惊喜。

注意 如需有关这些指令的详细信息，请参考 ARM 编译器工具链汇编参考文件，可在 http://infocenter.arm.com 找到。

表3-1　ARM指令

指　令	说　明
ADC	进位加法
ADD	加法
ADR	加载相对程序或相对寄存器地址（短范围）
ADRL（伪指令）	加载相对程序或相对寄存器地址（中等范围）
AND	逻辑“与”
ASR	算术右移
B	跳转
BFC	位域清零
BFI	位域插入
BIC	位清零
BKPT	断点
BL	带链接的跳转
BLX	带链接的跳转，并切换指令集
BX	跳转，并切换指令集
BXJ	跳转，切换到Jazelle
CBZ	比较，如果为0就跳转
CBNZ	比较，如果不为0就跳转
CDP	协处理器数据处理操作
CDP2	协处理器数据处理操作
CHKA	检查数组
CLREX	清除独占
CLZ	前导零计数
CMN	与负值比较
CMP	比较
CPS	更改处理器状态

更多资源请访问稀酷客(www.ckook.com)

（续）

指　令	说　明
DBG	调试指示
DMB	数据内存屏障
DSB	数据同步屏障
ENTERX	将状态更改为ThumbEE
EOR	异或
HB	处理程序跳转，跳转到指定处理程序
HBL	处理程序跳转，跳转到指定处理程序
HBLP	处理程序跳转，跳转到指定处理程序
HBP	处理程序跳转，跳转到指定处理程序
ISB	指令同步屏障
IT	条件判断
LDC	加载协处理器
LDC2	加载协处理器
LDM	加载多个寄存器
LDR	加载寄存器
LDR（伪指令）	加载寄存器
LDRB	加载一个字节到寄存器
LDRBT	加载一个字节到寄存器，用户模式
LDRD	加载双字（即四字节）到寄存器
LDREX	加载寄存器，独占模式
LDREXB	加载一个字节到寄存器，独占方式
LDREXD	加载双字（即四字节）到寄存器，独占方式
LDREXH	加载半字到寄存器，独占方式
LDRH	加载半字到寄存器
LDRHT	加载半字到寄存器，用户模式
LDRSB	有符号加载一个字节到寄存器
LDRSBT	有符号加载一个字节到寄存器，用户模式
LDRSH	有符号加载半字到寄存器
LDRT	加载寄存器，用户模式
LEAVEX	退出ThumbEE状态
LSL	逻辑左移
LSR	逻辑右移
MAR	从寄存器移动到40位累加器
MCR	从寄存器移动到协处理器
MCR2	从寄存器移动到协处理器
MCRR	从寄存器移动到协处理器
MCRR2	从寄存器移动到协处理器
MIA	带内部累加器的乘法

更多资源请访问稀酷客(www.ckook.com)

（续）

指　令	说　明
MIAPH	带内部累加器的乘法，组合半字（packed halfwords）①
MIAxy	带内部累加器的乘法，半字
MLA	乘加
MLS	乘减
MOV	移动
MOVT	移动到顶部
MOV32（伪指令）	移动32位常数到寄存器
MRA	从40位累加器移动到寄存器
MRC	从协处理器移动到寄存器
MRC2	从协处理器移动到寄存器
MRRC	从协处理器移动到寄存器
MRRC2	从协处理器移动到寄存器
MRS	从PSR移动到寄存器
MRS	从系统协处理器移动到寄存器
MSR	从寄存器移动到PSR
MSR	从寄存器移动到系统协处理器
MUL	乘法
MVN	取反移动
NOP	空操作
ORN	逻辑“或非”
ORR	逻辑“或”
PKHBT	组合半字（底部+顶部）
PKHTB	组合半字（顶部+底部）
PLD	预载数据
PLDW	预载要写入的数据
PLI	预载指令
POP	从栈弹出寄存器
PUSH	将寄存器压入栈
QADD	有符号加，饱和指令②
QADD8	并行有符号加（4×8位），饱和指令
QADD16	并行有符号加（2×16位），饱和指令
QASX	交换半字，有符号加减，饱和指令
QDADD	有符号加倍和加，饱和指令

① 在对一个 word 进行计算时，有些算法将一个 word 分为高低两个半字进行计算，详见 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204ic/Cihffbgc.html。——译者注

② 饱和指令参考 http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0204ic/Cihidceh.html，在多媒体处理常常用到。——译者注

更多资源请访问稀酷客(www.ckook.com)

（续）

指　令	说　明
QDSUB	有符号加倍和减，饱和指令
QSAX	交换半字，有符号减和加，饱和指令
QSUB	有符号减，饱和指令
QSUB8	并行有符号减（4×8位），饱和指令
QSUB16	并行有符号减（2×16位），饱和指令
RBIT	反转位
REV	反转字节（改变字节序）①
REV16	半字中反转字节
REVSH	低半字反转字节并进行符号扩展
RFE	从异常中返回
ROR	寄存器循环右移
RRX	寄存器右移并符号扩展
RSB	反向减法
RSC	带进位反向减法
SADD8	并行有符号加（4×8位）
SADD16	并行有符号加（2×16位）
SASX	交换半字，有符号加和减
SBC	带进位减
SBFX	有符号位域提取
SDIV	有符号除法
SEL	根据APSR GE标记选择字节
SETEND	设置内存访问的端标记
SEV	设置事件
SHADD8	有符号加（4×8位），结果减半
SHADD16	有符号加（2×16位），结果减半
SHASX	交换半字，有符号加和减，结果减半
SHSAX	交换半字，有符号减和加，结果减半
SHSUB8	有符号减（4×8位），结果减半
SHSUB16	有符号减（2×16位），结果减半
SMC	安全监控调用
SMLAxy	有符号乘加
SMLAD	两次有符号乘加
SMLAL	有符号乘加
SMLALxy	有符号乘加
SMLALD	两次有符号长整数乘加
SMLAWy	有符号乘加

① 即大端（big endian）小端（little endian）变换。——译者注

更多资源请访问稀酷客(www.ckook.com)

（续）

指　令	说　明
SMLSD	两次有符号乘减累加
SMLSLD	两次有符号长整数乘减累加
SMMLA	有符号高位字乘加
SMMLS	有符号高位字乘减
SMMUL	有符号高位字乘法
SMUAD	两次有符号乘并相加
SMULxy	有符号乘法
SMULL	有符号乘法
SMULWy	有符号乘法
SMUSD	两次有符号乘并相减
SRS	存储返回状态
SSAT	有符号饱和
SSAT16	有符号饱和，并行半字
SSAX	并行交换半字，有符号减和加
SSUB8	并行有符号按字节减
SSUB16	并行有符号按半字减
STC	存储协处理器
STC2	存储协处理器
STM	存储多个寄存器（见LDM）
STR	存储寄存器（见LDR）
STRB	存储寄存器为一个字节
STRBT	存储寄存器为一个字节，用户模式
STRD	存储寄存器为双字
STREX	存储寄存器，独占方式（见LDREX）
STREXB	存储寄存器为一个字节，独占方式
STREXD	存储寄存器为双字，独占方式
STREXH	存储寄存器为半字，独占方式
STRH	存储寄存器为半字
STRHT	存储寄存器为半字，用户模式
STRT	存储寄存器，用户模式
SUB	减法
SUBS	从异常中返回，无出栈
SVC	超级用户调用
SWP	交换寄存器和内存（v6废弃）
SWPB	交换寄存器和内存（v6废弃）
SXTAB	有符号扩展，带加法
SXTAB16	有符号扩展两个8位值到16位，带加法
SXTAH	有符号扩展半字，带加法

更多资源请访问稀酷客(www.ckook.com)

（续）

指　令	说　明
SXTB	有符号扩展字节
SXTB16	有符号扩展两个8位值到16位
SXTH	有符号扩展半字
SYS	执行系统协处理器指令
TBB	表跳转字节
TBH	表跳转半字
TEQ	相等测试
TST	测试
UADD8	并行无符号加（4×8位）
UADD16	并行无符号加（2×16位）
UASX	交换半字，无符号加和减
UBFX	无符号位域提取
UDIV	无符号除法
UHADD8	无符号加（4×8位），结果减半
UHADD16	无符号加（2×16-bit），结果减半
UHASX	交换半字，无符号加和减，结果减半
UHSAX	交换半字，无符号减和加，结果减半
UHSUB8	无符号减（4×8位），结果减半
UHSUB16	无符号减（2×16位），结果减半
USAD8	差值的绝对值无符号求和r
USADA8	差值的绝对值无符号求和再累加
USAT	无符号饱和
USAT16	无符号饱和，并行半字
USAX	交换半字，无符号减和加
USUB8	并行无符号按字节减
USUB16	并行无符号按半字减
UXTB	无符号用零扩展到字节
UXTB16	无符号用零扩展两个8位值到16位值
UXTH	用零扩展，半字
WFE	等待事件
WFI	等待中断
YIELD	通知

3.1.5 ARM NEON

NEON是Cortex A系列处理器的128位SIMD（单指令多数据流）扩展[①]。如果你了解代码

① Intel MMX指令集也有类似扩展。——译者注

更多资源请访问稀酷客(www.ckook.com)

清单 3-27 里 UHADD8 指令都做了哪些工作，那么你会很容易理解 NEON。NEON 寄存器可看作位向量。例如，128 位的 NEON 寄存器可看作 4 个 32 位整数、8 个 16 位整数，甚至 16 个 8 位整数（UHADD8 指令以同样方式解释，一个 32 位的寄存器可以看作是 4 个 8 位值）。NEON 指令在所有元素上执行相同的操作。

NEON 有以下几个重要特点：

- 单指令可以执行多个操作（毕竟，这是 SIMD 指令本质）；
- 独立的寄存器；
- 独立的流水线。

NEON 指令看起来类似 ARM 指令。例如，VADD 指令在两个向量中加上相应的元素，这与 ADD 指令类似（ADD 指令只是把两个 32 位寄存器相加，并不把它们作为位向量处理）。所有 NEON 指令以字母 V 开始，很容易识别出来。

在代码中使用 NEON 有以下两种基本方式：

- 手写汇编代码中使用；
- 使用在 NDK 的头文件 arm-neon.h 中定义的 NEON 内联函数。

NDK 提供了 NEON（hello-neon）示例代码，你先要熟悉下这些代码。使用 NEON 可以大幅提高性能，不过为了充分利用向量化，可能还需要略微修改算法。

在使用 NEON 指令时，需要在 Android.mk 的 LOCAL_SRC_FILES 变量里给文件名加上.neon 后缀。如果需要把所有文件都编译成支持 NEON，可以直接在 Android.mk 设置 LOCAL_ARM_NEON 为 true。

阅读 Android 的源代码是学习 NEON 的一个很好的途径。例如，SkBlitRow_opts_arm.cpp（位于 external/skia/src/opts 目录）包含了几个使用 NEON 指令、asm()或内联函数的小程序。在同一目录中，你还会发现另一个文件 SkBlitRow_opts_SSE2.cpp，它包含了使用 x86 SIMD 指令的优化程序。Skia 的源代码请参见http://code.google.com/p/skia。

3.1.6 CPU 特性

如你所见，并非所有 CPU 都是完全相同的。即使同一系列的处理器（例如，ARM Cortex 系列处理器），也不是所有的处理器都支持相同的特性，因为有些特性是可选的。例如，并不是所有的 ARMv7 处理器都支持 NEON 或 VFP 扩展。出于这个原因，Android 提供了一些函数，帮助查询程序运行在什么样的处理器上。这些函数定义在 NDK 头文件 cpu-features.h，代码清单 3-30 显示了如何使用这些函数确定是用通用函数还是用 NEON 指令集。

代码清单 3-30　检查 CPU 特性

```
#include <cpu-features.h>

static inline int has_features(uint64_t features, uint64_t mask) {
    return ((features & mask) == mask);
}
```

更多资源请访问稀酷客(www.ckook.com)

```
static void (*my_function)(int* dst, const int* src, int size); // 函数指针
external void neon_function (INT * DST, const int* src, int size); // 定义在其他文件中
extern void default_function(int* dst, const int* src, int size);

int init () {
    AndroidCpuFamily cpu = android_getCpuFamily();
    uint64_t features = android_getCpuFeatures();
    int count = android_getCpuCount(); // 这里忽略
    if (cpu == ANDROID_CPU_FAMILY_ARM) {
        if (has_features(features,
                         ANDROID_CPU_ARM_FEATURE_ARMv7|
                         ANDROID_CPU_ARM_FEATURE_NEON)) {
            my_function = neon_function;
        } else {
            // 这里使用默认函数
            my_function = default_function; // 通用函数
        }
    } else {
        my_function = default_function; // 通用函数
    }
}

void call_function(int* dst, const int* src, int size) {
    // 假设 init()在此之前被调用过，已设置了 my_function 函数指针
    my_function(dst, src, size);
}
```

使用CPU特性函数之前要在Android.mk做以下两件事：

- 在静态库链接清单加入“cpufeatures”（LOCAL_STATIC_LIBRARIES := cpufeatures）；
- 在Android.mk结尾处添加$(call import-module,android/cpufeatures)，导入android/cpufeatures模块。

通常情况下，为了使用最适合的函数进行优化，首先要探测CPU特性。

如果代码依赖VFP扩展，你可能也要检查是否支持NEON。ANDROID_CPU_ARM_FEATURE_VFPv3标志是64位浮点寄存器扩展的最小规格，有16个寄存器（D0到D15）。如果支持NEON，会有32个64位浮点寄存器（D0到D31）。寄存器在NEON和VFP间以别名共享：

- Q0（128位）是D0和D1（都是64位）的别名。
- D0是S0和S1（S寄存器是单精度32位寄存器）的别名。

实际上，寄存器共享和别名这部分细节非常重要，手写汇编时使用寄存器要万分小心。

注意 请参阅NDK的文档，尤其是cpu-features.html，获取更多相关API信息。

3.2 C扩展

Android NDK用的是GCC编译器（NDK r7中为4.4.3版）。你可以使用GNU编译器集支持

更多资源请访问稀酷客(www.ckook.com)

的C扩展。其中特别值得一提且与性能相关的是：

- 内置函数
- 向量指令

注意　请访问http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html 获取 GCC C 扩展的详尽清单。[①]

3.2.1　内置函数

内置（built-in）函数，有时也被称为内联（intrinsics），是由编译器内部进行特别处理的函数。内置函数常常用来构造某些语言支持不了的特性，经常是内联的，也就是说，编译器会替换掉这个函数调用，把它展开为一系列针对目标平台处理并进行典型优化的指令。例如，调用__builtin_clz()函数会生成CLZ指令（如果代码被编译为ARM且CLZ指令可用）。当内置函数没有优化版本，或优化被关闭时，编译器只是调用包含通用实现的函数。

例如，GCC支持以下内置函数：

- __builtin_return_address
- __builtin_frame_address
- __builtin_expect
- __builtin_assume_aligned
- __builtin_prefetch
- __builtin_ffs
- __builtin_clz
- __builtin_ctz
- __builtin_clrsb
- __builtin_popcount
- __builtin_parity
- __builtin_bswap32
- __builtin_bswap64

使用内置函数，可以在保持代码通用性的同时，充分利用某些平台上特有的优化。

3.2.2　向量指令

在C代码使用向量指令很少见。但随着越来越多的CPU支持SIMD指令，在算法中使用向量指令可以让速度激增。

代码清单3-31演示了用vector_size变量属性定义自己的向量类型，并把两个向量相加的方法。

① GCC有很多特性值得了解，熟悉一下编译选项及扩展，可能有意想不到的发现。——译者注

更多资源请访问稀酷客(www.ckook.com)

代码清单 3-31 向量

```
typedef int v4int __attribute__ ((vector_size (16))); // 4个整数的向量（16字节）

void add_buffers_vectorized (int* dst, const int* src, int size) {
    v4int* dstv4int = (v4int*) dst;
    const v4int* srcv4int = (v4int*) src;
    int i;

    for (i = 0; i < size/4; i++) {
        *dstv4int++ += *srcv4int++;
    }

    // 剩下的
    if (size & 0x3) {
        dst = (int*) dstv4int;
        src = (int*) srcv4int;

        switch (size & 0x3) {
            case 3: *dst++ += *src++;
            case 2: *dst++ += *src++;
            case 1:
            default:  *dst += *src;
        }
    }
}
// 简单的实现
void add_buffers (int* dst, const int* src, int size) {
    while (size--) {
        *dst++ += *src++;
    }
}
```

此代码如何编译取决于目标平台是否支持 SIMD 指令，编译器是否可使用这些指令。要想让编译器使用 NEON 指令，只需在 Android.mk 的 LOCAL_SRC_FILES 文件列表里将对应文件名添上.neon 后缀。另外，如果所有文件都要用 NEON 编译，可以设置 LOCAL_ARM_NEON 为 true。

代码清单 3-32 是编译器不用 ARM SIMD 指令（NEON）产生的汇编代码，而代码清单 3-33 是用了 NEON 指令的。（add_buffers 函数以同样的方式编译，第二份代码清单没有列出。）循环在两个列表中加粗显示。

代码清单 3-32 没有 NEON 指令

```
00000000 <add_buffers_vectorized>:
   0:    e92d 0ff0      stmdb    sp!, {r4, r5, r6, r7, r8, r9, sl, fp}
   4:    f102 0803      add.w    r8, r2, #3    ; 0x3
   8:    ea18 0822      ands.w    r8, r8, r2, asr #32
   c:    bf38           it    cc
   e:    4690           movcc    r8, r2
  10:    b08e           sub    sp, #56
  12:    4607           mov    r7, r0
  14:    468c           mov    ip, r1
```

更多资源请访问稀酷客(www.ckook.com)

```
16:    ea4f 08a8      mov.w    r8, r8, asr #2
1a:    9201           str    r2, [sp, #4]
1c:    f1b8 0f00      cmp.w    r8, #0    ; 0x0
20:    4603           mov    r3, r0
22:    460e           mov    r6, r1
24:    dd2c           ble.n    80 <add_buffers_vectorized+0x80>
26:    2500           movs    r5, #0
28:    f10d 0928      add.w    r9, sp, #40    ; 0x28
2c:    462e           mov    r6, r5
2e:    f10d 0a18      add.w    sl, sp, #24    ; 0x18
32:    f10d 0b08      add.w    fp, sp, #8    ; 0x8
36:    197c           adds    r4, r7, r5
38:    3601           adds    r6, #1
3a:    e894 000f      ldmia.w    r4, {r0, r1, r2, r3}
3e:    e889 000f      stmia.w    r9, {r0, r1, r2, r3}
42:    eb0c 0305      add.w    r3, ip, r5
46:    3510           adds    r5, #16
48:    4546           cmp    r6, r8
4a:    cb0f           ldmia    r3!, {r0, r1, r2, r3}
4c:    e88a 000f      stmia.w    sl, {r0, r1, r2, r3}
50:    9b0a           ldr    r3, [sp, #40]
52:    9a06           ldr    r2, [sp, #24]
54:    4413           add    r3, r2
56:    9a07           ldr    r2, [sp, #28]
58:    9302           str    r3, [sp, #8]
5a:    9b0b           ldr    r3, [sp, #44]
5c:    4413           add    r3, r2
5e:    9a08           ldr    r2, [sp, #32]
60:    9303           str    r3, [sp, #12]
62:    9b0c           ldr    r3, [sp, #48]
64:    4413           add    r3, r2
66:    9a09           ldr    r2, [sp, #36]
68:    9304           str    r3, [sp, #16]
6a:    9b0d           ldr    r3, [sp, #52]
6c:    4413           add    r3, r2
6e:    9305           str    r3, [sp, #20]
70:    e89b 000f      ldmia.w    fp, {r0, r1, r2, r3}
74:    e884 000f      stmia.w    r4, {r0, r1, r2, r3}
78:    d1dd           bne.n    36 <add_buffers_vectorized+0x36>
7a:    0136           lsls    r6, r6, #4
7c:    19bb           adds    r3, r7, r6
7e:    4466           add    r6, ip
80:    9901           ldr    r1, [sp, #4]
82:    f011 0203      ands.w    r2, r1, #3    ; 0x3
86:    d007           beq.n    98 <add_buffers_vectorized+0x98>
88:    2a02           cmp    r2, #2
8a:    d00f           beq.n    ac <add_buffers_vectorized+0xac>
8c:    2a03           cmp    r2, #3
8e:    d007           beq.n    a0 <add_buffers_vectorized+0xa0>
90:    6819           ldr    r1, [r3, #0]
92:    6832           ldr    r2, [r6, #0]
94:    188a           adds    r2, r1, r2
96:    601a           str    r2, [r3, #0]
```

更多资源请访问稀酷客(www.ckook.com)

```
  98:    b00e          add    sp, #56
  9a:    e8bd 0ff0     ldmia.w    sp!, {r4, r5, r6, r7, r8, r9, sl, fp}
  9e:    4770          bx     lr
  a0:    6819          ldr    r1, [r3, #0]
  a2:    f856 2b04     ldr.w    r2, [r6], #4
  a6:    188a          adds    r2, r1, r2
  a8:    f843 2b04     str.w    r2, [r3], #4
  ac:    6819          ldr    r1, [r3, #0]
  ae:    f856 2b04     ldr.w    r2, [r6], #4
  b2:    188a          adds    r2, r1, r2
  b4:    f843 2b04     str.w    r2, [r3], #4
  b8:    e7ea          b.n    90 <add_buffers_vectorized+0x90>
  ba:    bf00          nop

00000000 <add_buffers>:
   0:    b470          push    {r4, r5, r6}
   2:    b14a          cbz    r2, 18 <add_buffers+0x18>
   4:    2300          movs    r3, #0
   6:    461c          mov    r4, r3
   8:    58c6          ldr    r6, [r0, r3]
   a:    3401          adds    r4, #1
   c:    58cd          ldr    r5, [r1, r3]
   e:    1975          adds    r5, r6, r5
  10:    50c5          str    r5, [r0, r3]
  12:    3304          adds    r3, #4
  14:    4294          cmp    r4, r2
  16:    d1f7          bne.n    8 <add_buffers+0x8>
  18:    bc70          pop    {r4, r5, r6}
  1a:    4770          bx     lr
```

代码清单 3-33 NEON 指令

```
00000000 <add_buffers_vectorized>:
   0:    b470          push    {r4, r5, r6}
   2:    1cd6          adds    r6, r2, #3
   4:    ea16 0622     ands.w    r6, r6, r2, asr #32
   8:    bf38          it     cc
   a:    4616          movcc    r6, r2
   c:    4604          mov    r4, r0
   e:    460b          mov    r3, r1
  10:    10b6          asrs    r6, r6, #2
  12:    2e00          cmp    r6, #0
  14:    dd0f          ble.n    36 <add_buffers_vectorized+0x36>
  16:    460d          mov    r5, r1
  18:    2300          movs    r3, #0
  1a:    3301          adds    r3, #1
  1c:    ecd4 2b04     vldmia    r4, {d18-d19}
  20:    ecf5 0b04     vldmia    r5!, {d16-d17}
  24:    42b3          cmp    r3, r6
  26:    ef62 08e0     vadd.i32    q8, q9, q8
  2a:    ece4 0b04     vstmia    r4!, {d16-d17}
  2e:    d1f4          bne.n    1a <add_buffers_vectorized+0x1a>
```

更多资源请访问稀酷客(www.ckook.com)

```
30:    011b          lsls    r3, r3, #4
32:    18c4          adds    r4, r0, r3
34:    18cb          adds    r3, r1, r3
36:    f012 0203     ands.w    r2, r2, #3    ; 0x3
3a:    d008          beq.n    4e <add_buffers_vectorized+0x4e>
3c:    2a02          cmp    r2, #2
3e:    4621          mov    r1, r4
40:    d00d          beq.n    5e <add_buffers_vectorized+0x5e>
42:    2a03          cmp    r2, #3
44:    d005          beq.n    52 <add_buffers_vectorized+0x52>
46:    680a          ldr    r2, [r1, #0]
48:    681b          ldr    r3, [r3, #0]
4a:    18d3          adds    r3, r2, r3
4c:    600b          str    r3, [r1, #0]
4e:    bc70          pop    {r4, r5, r6}
50:    4770          bx    lr
52:    6820          ldr    r0, [r4, #0]
54:    f853 2b04     ldr.w    r2, [r3], #4
58:    1882          adds    r2, r0, r2
5a:    f841 2b04     str.w    r2, [r1], #4
5e:    6808          ldr    r0, [r1, #0]
60:    f853 2b04     ldr.w    r2, [r3], #4
64:    1882          adds    r2, r0, r2
66:    f841 2b04     str.w    r2, [r1], #4
6a:    e7ec          b.n    46 <add_buffers_vectorized+0x46>
```

你很快就会注意到，使用 NEON 指令集编译后的循环中的指令少得多。事实上，vldmia 指令从内存加载 4 个整数，vadd.i32 指令执行 4 次加法，vstmia 指令把 4 个整数存储到内存中。这样生成的汇编代码更紧凑，更高效。

使用向量是一把双刃剑，原因如下所述。

- 它们可以让你在使用 SIMD 指令时，同时拥有一份可以编译为任何 ABI 的通用代码，不论这个 ABI 是否支持 SIMD 指令。（代码清单 3-31 可以正常编译为 x86 ABI，因为它不是 NEON 专用的。）
- 当目标设备不支持 SIMD 指令时它们会生成低性能的代码。（add_buffers 函数比它的“向量化”版本简单得多，汇编代码也简单得多，只要看看不用 SIMD 指令时，add_buffers_vectorized 进行了多少次栈的读写就知道了。）

注意　访问 http://gcc.gnu.org/onlinedocs/gcc/Vector-Extensions.html 获取更多关于向量的信息。

3.3 技巧

下面介绍几个做起来相对容易却能提高性能的方法。

更多资源请访问稀酷客(www.ckook.com)

3.3.1 内联函数

因为函数调用的操作代价比较高，内联函数（即直接在调用处用实现替换调用）可以使代码运行得更快。把函数变为内联非常简单，只要在函数定义前加上“inline”关键字就可以了。代码清单 3-30 是内联函数的一个例子。

使用此功能时要小心，它会使代码变得臃肿，不利于使用指令高速缓存。通常情况下，对小的函数使用内联，节省的调用开销是很显著的。

注意 可以使用宏代替。

3.3.2 循环展开

一个经典的优化循环的方法是把它展开，有时可能只展开其中一部分。效果是不确定的，要试一试才知道优化是否成功，因为有可能得不偿失。确保循环体不要太大，这会对指令缓存有负面影响。

代码清单 3-34 是个循环展开的简单示例。

代码清单 3-34 展开循环

```
void add_buffers_unrolled (int* dst, const int* src, int size)
{
    int i;

    for (i = 0; i < size/4; i++) {
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        // GCC 真的不擅长做这个…… 没有生成 LDM/STM 指令
    }
    // 剩余部分
    if (size & 0x3) {
        switch (size & 0x3) {
            case 3: *dst++ += *src++;
            case 2: *dst++ += *src++;
            case 1:
            default:  *dst += *src;
        }
    }
}
```

3.3.3 内存预读取

如果对要访问的具体数据和要执行的指令有一定了解，可以针对这些数据和指令，在使用之

更多资源请访问稀酷客(www.ckook.com)

前进行预加载（或预取）。

把数据从从外部存储器中移动到缓存要花时间，提前给予充分的时间，把数据从外部存储器中的数据转移到缓存中，可以带来更好的性能，因为这样可以提高访问的数据或指令的缓存命中率。

你可以使用下列方式预加载数据：

- GCC 的__builtin_prefetch()；
- 在 ARM 汇编代码中使用 PLD 和 PLDW 指令，还可以使用 PLI 指令（适用于 ARMv7 及以上），来预加载指令。

一些 CPU 会自动预加载内存，所以可能并不总会看到效果。然而，如果更好地了解代码中访问数据的方式，预加载数据仍会有出色的效果。代码清单 3-35 展示了如何利用内置函数实现预加载。

代码清单 3-35　预加载内存

```
void add_buffers_unrolled_prefetch (int* dst, const int* src, int size) {
    int i;

    for (i = 0; i < size/8; i++) {
        __builtin_prefetch(dst + 8, 1, 0); // 准备写
        __builtin_prefetch(src + 8, 0, 0); // 准备读

        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
        *dst++ += *src++;
    }

    // 剩余部分
    for (i = 0; i < (size & 0x7); i++) {
        *dst++ += *src++;
    }
}
```

要小心使用预加载内存，在某些情况下，它会降低性能。缓存总数是固定的，有多少进来就得有多少出去，确保预加载的是必需的代码，否则会让无用数据污染缓存，降低性能。

注意　ARM 支持 PLD、PLDW、PLI 指令，x86 也支持 PREFETCHT0、PREFETCHT1、PREFETCHT2 和 PREFETCHNTA 指令。请参阅 ARM 和 x86 文档获取更多信息。改变__builtin_prefetch() 的最后一个参数，再编译为 x86，看看会用到哪些指令。

更多资源请访问稀酷客(www.ckook.com)

3.3.4 用 LDM/STM 替换 LDR/STD

使用单个 LDM 指令加载多个寄存器比用多个 LDR 指令加载寄存器快得多。同样，使用单个 STM 指令写入多个寄存器也比用多个 STR 指令写寄存器快得多。

编译器往往能够生成 LDM/STM 指令（即使当内存访问在代码中有些分散时），而你应该尽量编写容易被编译器优化的代码，来帮助编译器生成指令。例如，代码清单 3-36 给出了编译器很容易识别和生成 LDM 和 STM 指令（假设使用 ARM ABI）的样例。理想的情况下，访问内存的代码应该尽量组织在一起，尽可能让编译器生成更好的汇编代码。

代码清单 3-36 生成 LDM 和 STM 指令的样例

```
unsigned int a, b, c, d;
  // 假设 src 和 dst 为指向 int 的指针

  // read source values
  a = *src++;
  b = *src++;
  c = *src++;
  d = *src++;

  // 使用 a、b、 c 和 d 工作

  // 把值写到 dst 缓冲区
  *dst++ = a;
  *dst++ = b;
  *dst++ = c;
  *dst++ = d;
```

注意 展开循环和内联函数，还可以帮助编译器更容易生成 LDM 或 STM 指令。

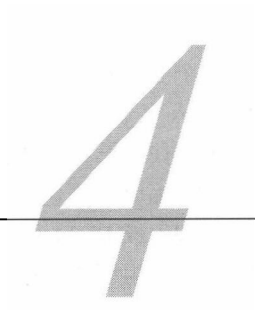
不幸的是，GCC 编译器在生成 LDM 和 STM 指令方面做得并不总是很好。如果你认为使用 LDM 和 STM 指令会大大提高性能，那么审读生成的汇编代码，然后自己手改写汇编代码。

3.4 总结

C 语言之父，Dennis Ritchie（DMR）说过，C 兼具汇编语言的力量和便捷等特点。本章介绍了在某些情况下，为了达到预期的效果，汇编语言是少不了的。汇编是个非常强大的语言，它可以直面机器的特性，但正因如此，针对特定架构的汇编语言会使代码很难维护。不过，汇编代码或内置函数通常会用于性能的关键处，占用很少的代码，因此，维护相对来说还是较容易的。如果你合理地把应用划分为不同部分，分别交给 Java、C/C++和汇编来实现，最终结果会让你惊喜，同时也给用户留下深刻印象。

更多资源请访问稀酷客(www.ckook.com)

第4章 高效使用内存



应用生存期的绝大部分时间都用于处理内存中的数据。虽然许多开发者都意识到在手机或平板这类设备上要尽可能少使用内存，但并非所有人都认识到了内存使用对性能的影响。本章将讨论如何选择正确的数据类型及如何在内存中排布数据，最终提高应用的性能。此外，我们会温习一下 Java 常常被忽视的基本特性：使用垃圾收集和引用的内存管理。

4.1 说说内存

无论分配给应用多少内存，它都不会满足。

Android 设备（如手机或平板电脑）和传统的电脑有两个很大的差异：

- 物理内存大小
- 虚拟内存交换能力

通常情况下，今天的电脑基本都有几个 GB 的 RAM（难见只有 1GB 或更少的），但在常见 Android 设备中 512MB RAM 就算很大了。[①]雪上加霜的是，电脑上使用的内存交换能力，可以给系统和应用程序有更多内存可用的假象，这个是 Android 设备没有的。例如，即使系统只有 256MB 的 RAM，应用程序仍然可以在高达 4GB 的内存空间中寻址，Android 应用根本不会享有这种待遇，因而必须小心应用的内存使用量。

注意 可以在 Android Market 中找到启用交换分区的应用，但它们需要 root 权限和修改过的 Linux 内核。[②]通常假定应用运行在不具备内存交换功能的 Android 设备上。

Java 语言定义的基本类型的大小，还有对应的本地类型，如表 4-1 所示。如果你之前主要用的是 C/C++，那么要格外注意以下两件事：

- Java 的 char 是 16 位（UTF-16）；
- Java 的 long 是 64 位（通常 C 的 long 是 32 位、long long 是 64 位）。

---

① 现在高端 1GB RAM 已经比较常见了，手机的发展速度很快，256MB 到 1GB 没花多少时间。——译者注

② 打开 swap 支持。——译者注

更多资源请访问稀酷客(www.ckook.com)

表4-1 Java基本类型

Java基本类型	本地类型	大 小
boolean	jboolean	8位（取决于VM）
byte	jbyte	8位
char	jchar	16位
short	jshort	16位
int	jint	32位
long	jlong	64位
float	jfloat	32位
double	jdouble	64位

Android 设备和其上应用使用的内存是有限的，使用尽可能少的内存，既是经验也是常识。此外，除了减少碰到 OutOfMemoryError 异常的风险，使用较少的内存也可以提升性能。

性能主要取决于以下三个因素：

- CPU 如何操纵特定的数据类型；
- 数据和指令需要占用多少存储空间；
- 数据在内存中的布局。

我们会逐一讲解这几项，查看本地代码和 Java 代码，测量示例的执行时间，最后比较几个简单的提升性能的算法实现。

4.2 数据类型

其实我们在前面章节已经介绍过 CPU 如何操纵特定的数据类型，回顾一下在第 2 章中本地代码版本的 computeIterativelyFaster()，它用了两条加法指令来求两个 64 位整数的和，如代码清单 4-1 所示。

代码清单 4-1 两个 64 位整数相加

```
448:  e0944002   adds  r4, r4, r2
44c:  e0a55003   adc  r5, r5, r3
```

ARM 寄存器宽是 32 位，两个 64 位整数相加需要两条指令，低 32 位整数存储在寄存器 r4 中，高 32 位存储在寄存器 r5 中。两个 32 位整数相加只要一条指令就够了。

现在，让我们考虑非常简单的 C 函数，它只是返回传入的两个 32 位整数参数的和，如代码清单 4-2 所示。

代码清单 4-2 两个值的和

```
int32_t add_32_32 (int32_t value1, int32_t value2) {
    return value1 + value2;
}
```

更多资源请访问稀酷客(www.ckook.com)

函数的汇编代码如代码清单 4-3 所示。

代码清单 4-3　汇编代码

```
000016c8 <add_32_32>:
    16c8:  e0810000    add  r0, r1, r0
    16cc:  e12fff1e     bx  lr
```

不出所料，两个数相加只要一条指令（bx lr 是 return）。现在，我们写个与 add_32_32 类似的新函数，但 value1 和 value2 的类型不同。例如，add_16_16 是两个 int16_t 值相加，如代码清单 4-4 所示；add_16_32 是 int16_t 和 int32_t 相加，如代码清单 4-5 所示。

代码清单 4-4　add_16_16 的 C 和汇编代码

```
int16_t add_16_16 (int16_t value1, int16_t value2) {
    return value1 + value2;
}

000016d0 <add_16_16>:
    16d0:  e0810000    add  r0, r1, r0
    16d4:  e6bf0070    sxth  r0, r0
    16d8:  e12fff1e    bx  lr
```

代码清单 4-5　add_16_32 的 C 和汇编代码

```
int32_t add_16_32 (int16_t value1, int32_t value2) {
    return value1 + value2;
}

000016dc <add_16_32>:
    16dc:  e0810000    add  r0, r1, r0
    16e0:  e12fff1e    bx  lr
```

你也看到了，两个 16 位数相加需要一条额外的指令，用来把结果从 16 位转到 32 位。代码清单 4-6 又给出 5 个函数，基本上是重复相同的算法，但使用了不同的数据类型。

代码清单 4-6　更多的汇编代码

```
000016e4 <add_32_64>:
    16e4:  e0922000    adds  r2, r2, r0
    16e8:  e0a33fc0    adc  r3, r3, r0, asr #31
    16ec:  e1a00002    mov  r0, r2
    16f0:  e1a01003    mov  r1, r3
    16f4:  e12fff1e    bx  lr

000016f8 <add_32_float>:
    16f8:  ee070a10    fmsr  s14, r0
    16fc:  eef87ac7    fsitos  s15, s14
    1700:  ee071a10    fmsr  s14, r1
    1704:  ee777a87    fadds  s15, s15, s14
    1708:  eefd7ae7    ftosizs s15, s15
    170c:  ee170a90    fmrs  r0, s15
    1710:  e12fff1e    bx  lr
```

更多资源请访问稀酷客(www.ckook.com)

```
00001714 <add_float_float>:
    1714:  ee070a10    fmsr  s14, r0
    1718:  ee071a90     fmsr  s15, r1
    171c:  ee377a27     fadds  s14, s14, s15
    1720:  ee170a10     fmrs  r0, s14
    1724:  e12fff1e    bx  lr

00001728 <add_double_double>:
    1728:  ec410b16     vmov  d6, r0, r1
    172c:  ec432b17     vmov  d7, r2, r3
    1730:  ee366b07     faddd  d6, d6, d7
    1734:  ec510b16     vmov  r0, r1, d6
    1738:  e12fff1e    bx  lr

0000173c <add_float_double>:
    173c:  ee060a10     fmsr  s12, r0
    1740:  eeb77ac6     fcvtds  d7, s12
    1744:  ec432b16     vmov  d6, r2, r3
    1748:  ee376b06     faddd  d6, d7, d6
    174c:  ec510b16     vmov  r0, r1, d6
    1750:  e12fff1e    bx  lr
```

注意 生成的汇编代码可能会和这里看到的有所不同，依加法的上下文而定。（内联的代码可能看起来不同，编译器可能会重新排列指令或改变寄存器分配。）

你可以看到，使用较小的类型并不总会提高性能，实际上可能需要更多的指令，如代码清单4-4所示。此外，如果 add_16_16 传入两个 32 位参数，这两个参数必须首先要转换到 16 位才能使用，如代码清单 4-7 所示。再次，sxth 指令执行“符号扩展”操作，将 32 位值转为 16 位。

代码清单 4-7 传入 add_16_16 两个 32 位参数

```
00001754 <add_16_16_from_32_32>:
    1754:  e6bf0070   sxth  r0, r0
    1758:  e6bf1071   sxth  r1, r1
    175c:  eaffffdb  b  16d0 <add_16_16>
```

4.2.1 值的比较

现在，让我们考虑另一个基本函数，它带有两个参数，如果第一个大于第二个返回 1，否则返回 0，如代码清单 4-8 所示。

代码清单 4-8 比较两个值

```
int32_t cmp_32_32 (int32_t value1, int32_t value2) {
    return (value1 > value2) ? 1 : 0;
}
```

更多资源请访问稀酷客(www.ckook.com)

我们再看下此函数的汇编代码及其一系列变种，如代码清单4-9所示。

代码清单4-9　比较两个值的汇编代码

```
00001760 <cmp_32_32>:
    1760:  e1500001   cmp  r0, r1
    1764:  d3a00000   movle  r0, #0  ; 0x0
    1768:  c3a00001   movgt  r0, #1  ; 0x1
    176c:  e12fff1e  bx  lr

00001770 <cmp_16_16>:
    1770:  e1500001   cmp  r0, r1
    1774:  d3a00000   movle  r0, #0  ; 0x0
    1778:  c3a00001   movgt  r0, #1  ; 0x1
    177c:  e12fff1e  bx  lr

00001780 <cmp_16_32>:
    1780:  e1500001   cmp  r0, r1
    1784:  d3a00000   movle  r0, #0  ; 0x0
    1788:  c3a00001   movgt  r0, #1  ; 0x1
    178c:  e12fff1e  bx  lr

00001790 <cmp_32_64>:
    1790:  e92d0030   push  {r4, r5}
    1794:  e1a04000   mov  r4, r0
    1798:  e1a05fc4  asr  r5, r4, #31
    179c:  e1550003   cmp  r5, r3
    17a0:  e3a00000   mov  r0, #0  ; 0x0
    17a4:  ca000004   bgt  17bc <cmp_32_64+0x2c>
    17a8:  0a000001   beq  17b4 <cmp_32_64+0x24>
    17ac:  e8bd0030   pop  {r4, r5}
    17b0:  e12fff1e  bx  lr
    17b4:  e1540002   cmp  r4, r2
    17b8:  9afffffb  bls  17ac <cmp_32_64+0x1c>
    17bc:  e3a00001   mov  r0, #1  ; 0x1
    17c0:  eafffff9  b  17ac <cmp_32_64+0x1c>

000017c4 <cmp_32_float>:
    17c4:  ee070a10   fmsr  s14, r0
    17c8:  eef87ac7  fsitos  s15, s14
    17cc:  ee071a10   fmsr  s14, r1
    17d0:  eef47ac7  fcmpes  s15, s14
    17d4:  eef1fa10  fmstat
    17d8:  d3a00000   movle  r0, #0  ; 0x0
    17dc:  c3a00001   movgt  r0, #1  ; 0x1
    17e0:  e12fff1e  bx  lr

000017e4 <cmp_float_float>:
    17e4:  ee070a10   fmsr s14, r0
    17e8:  ee071a90   fmsr s15,  r1
    17ec:  eeb47ae7   fcmpes  s14, s15
    17f0:  eef1fa10  fmstat
    17f4:  d3a00000   movle  r0, #0  ; 0x0
```

更多资源请访问稀酷客(www.ckook.com)

```
    17f8:  c3a00001   movgt  r0, #1  ; 0x1
    17fc:  e12fff1e   bx  lr

00001800 <cmp_double_double>:
    1800:  ee060a10   fmsr  s12, r0
    1804:  eeb77ac6   fcvtds  d7, s12
    1808:  ec432b16   vmov  d6, r2, r3
    180c:  eeb47bc6   fcmped  d7, d6
    1810:  eef1fa10   fmstat
    1814:  d3a00000   movle  r0, #0  ; 0x0
    1818:  c3a00001   movgt  r0, #1  ; 0x1
    181c:  e12fff1e   bx  lr

00001820 <cmp_float_double>:
    1820:  ee060a10   fmsr  s12, r0
    1824:  eeb77ac6   fcvtds  d7, s12
    1828:  ec432b16   vmov  d6, r2, r3
    182c:  eeb47bc6   fcmped  d7, d6
    1830:  eef1fa10   fmstat
    1834:  d3a00000   movle  r0, #0  ; 0x0
    1838:  c3a00001   movgt  r0, #1  ; 0x1
    183c:  e12fff1e   bx  lr
    1840:  e3a00001   mov  r0, #1  ; 0x1
    1844:  e12fff1e   bx  lr
```

因为执行了很多指令，使用 long 看起来比 short 和 int 慢。同样，只使用 double 及混用 float 和 double，似乎比只用 float 慢。

注意 只用指令数量作为判断代码快慢的依据是不充分的。因为并不是所有指令的执行时间都相同，当今的 CPU 很复杂，不能仅靠计算指令数目推测某个操作要花费多少时间。

4.2.2 其他算法

现在，我们已经看到了不同的数据类型生成的不同的汇编代码，接下来看看处理大量数据的算法会是什么样子。

代码清单 4-10 是三个简单的方法：调用静态 Arrays.sort()方法来排序；找出数组中的最小值；对数组求和。

代码清单 4-10 Java 中的排序、查找、求和

```
private static void sort (int array[]) {
    Arrays.sort(array);
}

private static int findMin (int array[]) {
    int min = Integer.MAX_VALUE;
    for (int e : array) {
        if (e < min) min = e;
```

更多资源请访问稀酷客(www.ckook.com)

```
        }
        return min;
    }

    private static int addAll (int array[]) {
        int sum = 0;
        for (int e : array) {
            sum += e; // 这里会溢出，我们且不管它
        }
        return sum;
    }
```

表4-2是处理一百万个随机元素的数组时，这些函数花费的时间（单位：毫秒）。除了这些，这些方法变种（使用short、long、float和double类型）的结果也在其中。请参阅第6章，了解如何衡量执行时间的更多信息。

表4-2　1 000 000元素数组的执行时间

Java基本类型	sort	findMin	addAll
short	93	27	25
int	753	31	30
long	1240	57	54
float	1080	41	33
double	1358	58	55

通过上表可能总结出以下两条规律：

- short数组排序远快于其他类型数组；
- 处理64位类型（long或double）比处理32位类型慢。

4.2.3　数组排序

16位数组排序速度远远快于32位或64位，原因很简单，它使用的算法不同。int和long数组使用了某些版本的快速排序算法排序，而short数组使用计数排序，它的算法复杂度是线性的。因此使用short类型，是一石二鸟之策：更少的内存消耗（2MB，而不是int的4MB或long的8MB），更快的运行速度。

注意　许多人一直误以为快速排序通常是最高效的排序算法。可以参考Android源代码Arrays.java，看看每种类型的数组是如何排序的。

不太常见的解决方法是使用多个数组来存储数据。例如，如果应用需要存储0到10万之间的大量整数，那么你可能会使用32位的数组来存储，因为short类型的最大区间是–32 768到32 767。根据数值的分布情况，可能有许多值是小于等于32 767的。在这种情况下，使用两个数组可能会更好：一个是0到32 767之间（即short数组），另一个是大于32 767（int数组）。这

更多资源请访问稀酷客(www.ckook.com)

样多少有些复杂，但内存节省和潜在的性能提升会让你觉得这样做是值得的，甚至会帮你优化某些方法的实现。例如，寻找最大元素的方法，可能只需要遍历 int 数组。（只有 int 数组是空的时候才遍历 short 数组。）

在代码清单 4-9 和表 4-2 中没有列出的类型是 boolean。事实上，boolean 值数组排序是毫无意义的。不过有些场合需要存储大量 boolean 值，然后通过索引来检索。这种情况下，可以创建一个数组。虽然这样做可行，但会浪费很多位数据空间，boolean 值实际上只要 1 位（非真即假）就可以表示了，但每个条目都分配了 8 位。为此，人们定义了 BitSet 类：它可以在数组中存储 boolean 值（要指定索引来检索），同时内存占用最小（每个条目占一位）。如果你看了 BitSet 类的公有方法及其在 BitSet.java 中的实现，可能会注意到下面几件事情。

- BitSet 的后端实现是个 long 数组。为达到更好的性能，可以使用 int 数组。（测试显示换成 int 数组会有 10%的性能提升）。
- 代码中的一些注释指出，有些地方应该可以做得更好（例如，从 FIXME 开始的注释[①]）。
- 你可能不需要这个类的所有功能。[②]

出于以上这些原因，为了提高性能实现自己的类（或基于 BitSet.java）是可行的。

4.2.4 定义自己的类

代码清单 4-11 是一个非常简单的实现，如果数组创建后不再增长，所需操作只是设置或获取某个位的值（例如，实现自己的 Bloom 过滤器），可以用这个实现。比较这个简单的实现与 BitSet，测试结果显示性能提高了约 50%。我们可以更进一步提高性能，用简单的数组替代 SimpleBitSet 类：直接使用数组比使用 SimpleBitSet 对象快了约 50%（即使用数组的速度比用 BitSet 对象的快 4 倍）。但这种做法实际上违背了面向对象的封装原则，使用时要谨慎。

代码清单 4-11 定义自己的 BitSet 类

```
public class SimpleBitSet {
    private static final int SIZEOF_INT = 32;
    private static final int OFFSET_MASK = SIZEOF_INT - 1; // 0x1F

    private int[] bits;

    SimpleBitSet(int nBits) {
        bits = new int[(nBits + SIZEOF_INT - 1) / SIZEOF_INT];
    }

    void set(int index, boolean value) {
        int i = index / SIZEOF_INT;
        int o = index & OFFSET_MASK;
        if (value) {
```

① 有时，Java 及很多源码中会留下这种注释，以便将来调整，不过我查了 java.util.BitSet 1.6 的源码，没有找到 FIXME。——译者注

② 这个理由稍有点牵强，因为这是 SDK 自带的，非第三方实现，不太建议因为这个理由重新发明轮子，除非原类实现的功能对你的需要来讲实在太多。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
            bits[i] |= 1 << o; // 置该位为 1
        } else {
            bits[i] &= ~(1 << o); // 置该位为 0
        }
    }

    boolean get(int index) {
        int i = index / SIZEOF_INT;
        int o = index & OFFSET_MASK;
        return 0 != (bits[i] & (1 << o));
    }
}
```

另外，如果大部分位被设为相同的值，可以考虑使用 SparseBooleanArray 以节省内存（可能会付出些性能代价）。再提一下，你可以使用第 2 章中讨论的策略模式，使更换实现更加方便。

总而言之，这些示例和技术可以概括为如下几条规则。

- 处理大量数据时，使用可以满足要求的最小数据类型。例如，基于性能和空间的考量，选择 short 数组而不是 int 数组。如果对精度要求不高（如果需要，使用 FloatMath 类），使用 float 而不是 double。
- 避免类型转换。尽量保持类型一致，尽可能在计算中使用单一类型。
- 如果有必要取得更好的性能，推倒重来，但要认真处理。

当然，这些规则不是一成不变的。比如，你可能会发现在某些情况下，类型转换后性能的提升超过了开销。优化的态度要务实，好钢要花在刀刃上，解决那些真正的问题。

多半情况下，使用较少内存的原则是值得遵守的。此外，这样会留出更多的内存给其他任务，使用较少的内存还可以提高性能，因为这样可以提高 CPU 缓存的使用效率，更快地访问数据或指令。

4.3　访问内存

如前所述，操纵较大类型的数据代价较高，因为用到了指令较多。直观地说，指令越多性能越差，CPU 需要做很多额外的工作。此外，代码和数据都驻留在内存中，访问内存本身也有开销。

因为访问内存会产生一些开销，CPU 会把最近访问的内存内容缓存起来，无论是内存读还是写。事实上，CPU 通常使用两级缓存：

- 一级缓存（L1）
- 二级缓存（L2）

L1 缓存的速度较快，但比 L2 小。例如，L1 缓存可能是 64KB（32KB 的数据缓存、32KB 指令缓存），而 L2 缓存可能是 512KB。

注意　一些处理器可能还有 3 级缓存，通常几兆字节，但目前的嵌入式设备上还没有。

当数据或指令在缓存中找不到时，就是缓存未命中。这时需要从内存中取出数据或指令。缓

更多资源请访问稀酷客(www.ckook.com)

存未命中有以下几种情况：

- 指令缓存读未命中；
- 数据缓存读未命中；
- 写未命中。

第一种缓存未命中最关键，因为 CPU 要一直等到从内存中读出指令，才可以继续执行。第二种缓存未命中几乎和第一种同样重要，尽管 CPU 仍可能执行其他不依赖要读取数据的指令，但这种情况只会在 CPU 指令乱序执行时出现。最后一种缓存未命中的重要性最低，CPU 通常可以继续执行指令。你几乎无法控制写未命中，不过无须过于担心，只要关注前两种类型就好了，它俩是应该极力避免的缓存未命中情况。

缓存行的大小

除了总大小，缓存的另一个重要属性是其行大小。缓存是由行组成的，每行包含几个字节。例如，Cortex A8 的 L1 缓存的缓存行大小为 64 个字节（16 个字）。缓存和缓存行背后的思想是局部性原则：如果应用读取或写入到一个特定的地址，将来很可能会再次读取或写入到相同的地址，或非常邻近的地址。例如，代码清单 4-10 的 `findMin()`和 `addAll()`方法的实现中，这种行为显而易见。

没有简单的方法告知应用缓存和缓存行的大小。不过知道利用缓存和一些缓存的工作原理可以帮助你写出更好的代码，获得更佳的性能。下面的提示可以帮助你利用缓存，而不必诉诸底层的优化手段（如第 3 章的 PLD 和 PLI 汇编指令）。以下方法可以减少指令缓存未命中的几率。

- 在 Thumb 模式下编译本地库。这不保证会使代码速度更快，因为 Thumb 指令码比 ARM 指令码执行得慢（可能要执行更多的指令）。如何用 Thumb 模式编译本地库的更多信息请参阅第 2 章。
- 保持代码相对密集。虽然不能保证密集的 Java 代码会产生密集的机器码，但这往往是可行的。

以下方法可以减少数据缓存读未命中几率。

- 在有大量数据存储在数组中时，使用尽可能小的数据类型。
- 选择顺序访问而不是随机访问。最大限度地重用已在缓存中的数据，防止数据从缓存中清除后再次载入。

注意 *现代 CPU 都能够自动预取内存，以避免或者说至少限制了缓存未命中情况的发生。*

如前所述，在应用性能的关键处使用这些技巧，这种代码通常只占一小部分。一方面，在 Thumb 模式下编译是一个简单的优化，并没有真正提高维护成本。另一方面，从长远来看，编写密集的代码可能会使事情变得更加复杂。没有一个放之四海而皆准的优化方法，要权衡各种优化手段。

更多资源请访问稀酷客(www.ckook.com)

虽然不必细到控制缓存数据的进进出出，但如何组织和使用数据最终会影响缓存的使用，进而影响性能。某些情形下，虽然会提高复杂性和维护成本，但还是需要将数据以特定方式排布，以提升缓存命中率。

4.4　排布数据

这一节会再次打破封装的原则。假设应用需要存储以某种方式排序的记录，每个记录包含两个字段：id 和 value。遵循面向对象的方法是定义一个记录类，如代码清单 4-12 所示。

代码清单 4-12　记录类

```
public class Record {
    private final short id;
    private final short value;
    // 可能会有更多数据

    public Record(short id, short value) {
        this.id = id;
        this.value = value;
    }

    public final short getId() {
        return id;
    }

    public final short getValue() {
        return value;
    }

    public void doSomething() {
        // 在这里做些事
    }
}
```

现在，Record 类定义完毕，应用可以简单地分配一个数组，在数组中保存记录，并提供一些访问方法，如代码清单 4-13 所示。

代码清单 4-13　保存记录

```
public class MyRecords {
    private Record[] records;
    int nbRecords;

    public MyRecords (int size) {
        records = new Record[size];
    }

    public int addRecord (short id, short value) {
        int index;
        if (nbRecords < records.length) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            index = nbRecords;
            records[nbRecords] = new Record(id, value);
            nbRecords++;
        } else {
            index = -1;
        }
        return index;
    }

    public void deleteRecord (int index) {
        if (index < 0) {
            // 这里抛出异常—无效的参数
        }
        if (index < nbRecords) {
            nbRecords--;
            records[index] = records[nbRecords];
            records[nbRecords] = null; // 不要忘记删除引用
        }
    }

    public int sumValues (int id) {
        int sum = 0;
        for (int i = 0; i < nbRecords; i++) {
            Record r = records[i];
            if (r.getId() == id) {
                sum += r.getValue();
            }
        }
        return sum;
    }

    public void doSomethingWithAllRecords () {
        for (int i = 0; i < nbRecords; i++) {
            records[i].doSomething();
        }
    }
}
```

这一切都可以正常运行，设计也非常简洁。不过，在实际运行代码前，有些缺陷现在还无法看到。

- 每添加一条新的记录到数组中，就要创建一个新的对象。尽管对象是轻量级的，但内存分配还是会产生开销，而这是可以避免。
- 如果 id 和 value 是公有的，就可以避免 getId()和 getValue()的开销。

修改 Record 类，使 id 和 value 变成公有的，这就是小菜一碟。略作修改的 sumValues()，如代码清单 4-14 所示。

代码清单 4-14 修改过的 sumValues()

```
public int sumValues (int id) {
    int sum = 0;
    for (Record r : records) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            if (r.id == id) {
                sum += r.value;
            }
        }
        return sum;
    }
```

不过，仅这样做并不能减少分配记录对象的数量，还是要创建记录对象并添加到数组中。

注意　你可以在 C/C++中避免分配对象，但在 Java 中所有对象实际上都是引用，必须通过 new 操作符创建。

因为所有对象都是在堆中分配的，在数组中只能存储对象的引用。不过，你可以修改 MyRecords 类，使用 short 数组来避免分配，修改后的类如代码清单 4-15 所示。

代码清单 4-15　修改后的 MyRecords 类，使用 short 数组

```
public class MyRecords {
    private short[] records;
    int nbRecords;

    public MyRecords (int size) {
        records = new short[size * 2];
    }

    public int addRecord (short id, short value) {
        int index;
        if (nbRecords < records.length) {
            index = nbRecords;
            records[nbRecords * 2] = id;
            records[nbRecords * 2 + 1] = value;
            nbRecords++;
        } else {
            index = -1;
        }
        return index;
    }

    public void deleteRecord (int index) {
        if (index < 0) {
            // 这里抛出异常-无效的参数
        }
        if (index < nbRecords) {
            nbRecords--;
            records[index * 2] = records[nbRecords * 2];
            records[index * 2 + 1] = records[nbRecords * 2 + 1];
        }
    }

    public int sumValues (int id) {
        int sum = 0;
```

更多资源请访问稀酷客(www.ckook.com)

```
        for (int i = 0; i < nbRecords; i++) {
            if (records[i * 2] == id) {
                sum += records[i * 2 + 1];
            }
        }
        return sum;
    }

    public void doSomethingWithAllRecords () {
        Record r = new Record(0, 0);
        for (int i = 0; i < nbRecords; i++) {
            r.id = records[i * 2];
            r.value = records[i * 2 + 1];
            r.doSomething();
        }
    }
}
```

接下来考虑下后续的情形，你会发现 MyRecords 类的使用情形如下：

- sumValues()的调用次数往往比 doSomethingWillAllRecords()多得多；
- 数组中只有寥寥几个记录共享相同的 id。

这说明了 id 字段往往比 value 字段读得更频繁。鉴于这一额外的信息，你可能会想出以下提高性能的解决方案：使用两个数组，而不是一个；让所有的 id 并拢，当 sumValues()依次遍历数组的 id 时来最大限度地提高缓存命中率。第一个数组只包含记录的 id，而第二个数组只包含记录的 value。这样，sumValues()执行时更多的 id 记录会进到缓存中，单个缓存行中 id 的记录数是之前的两倍。

代码清单 4-16 是新的 MyRecords 实现。

代码清单 4-16 修改 MyRecords 类使用两个数组

```
public class MyRecords {
    private short[] recordIds; // 第一个数组是 id
    private short[] recordValues; // 第二个数组是 value
    int nbRecords;

    public MyRecords (int size) {
        recordIds = new short[size];
        recordValues = new short[size];
    }

    public int addRecord (short id, short value) {
        int index;
        if (nbRecords < recordIds.length) {
            index = nbRecords;
            recordIds[nbRecords] = id;
            recordValues[nbRecords] = value;
            nbRecords++;
        } else {
            index = -1;
        }
```

更多资源请访问稀酷客(www.ckook.com)

```
        return index;
    }

    public void deleteRecord (int index) {
        if (index < 0) {
            // 这里抛出异常-无效的参数
        }
        if (index < nbRecords) {
            nbRecords--;
            recordIds[index] = recordIds[nbRecords];
            recordValues[index] = recordValues[nbRecords];
        }
    }

    public int sumValues (int id) {
        int sum = 0;
        for (int i = 0; i < nbRecords; i++) {
            if (recordIds[i] == id) {
                sum += recordValues[i]; // 如果 id 匹配，读取对应的 value
            }
        }
        return sum;
    }

    public void doSomethingWithAllRecords () {
        Record r = new Record(0, 0);
        for (int i = 0; i < nbRecords; i++) {
            r.id = recordIds[i];
            r.value = recordValues[i];
            r.doSomething();
        }
    }
}
```

这种优化手段并不总能派上用场。例如，上述内容假设 doSomething()不修改 Record 对象，MyRecords 不提供任何方法来从数组中检索记录对象。如果这些假设错了，代码清单 4-15 和代码清单 4-16 的实现就不再等同于代码清单 4-13。

请记住，如果不知道代码的使用方式，就不能正确地优化代码。再敲一次警钟，要本着务实的态度来优化：在找到要解决问题的关键点之前不要优化，针对一种使用方式的优化可能会给其他使用方式带来劣化。

4.5　垃圾收集

Java 的一个非常重要的优点是垃圾收集。不再使用的对象内存会被垃圾收集器释放（或回收）。例如，代码清单 4-15 中，当方法返回时，在 doSomethingWithAllRecords()中分配的 Record 对象便会被回收，其引用也不复存在。有两件非常重要的事情值得注意：

- 还是有可能出现内存泄漏；
- 垃圾收集器会帮你管理内存，它做的不仅仅是释放不用的内存。

更多资源请访问稀酷客(www.ckook.com)

4.5.1 内存泄漏

只有当某个对象不再被引用时，它的内存才会被回收，当该被释放的对象引用仍然存在时就会发生内存泄漏。从 Android 文档中看到的一个典型的例子是，由于屏幕旋转（如竖屏转为横屏），整个 Activity 对象会有泄露。这个典型的例子很容易重现，也是相当严重的，因为 Activity 对象占用相当多的内存（往往包含了多个对象的引用）。避免内存泄漏没有简单的解决方案，但 Android 提供的工具和 API 可以为你提供些帮助。

Eclipse 中的 DDMS 视图中，可以用 Heap 及 Allocation Tracker 跟踪内存使用和分配情况（或直接使用 sdk 工具 ddms）[①]。再提醒一下，这些工具是不会告诉你是否存在内存泄漏的，但你可以用它们来分析应用的内存使用情况，并有希望找出应用中的内存泄漏。

提示 Eclipse 内存分析器可以更好地分析内存使用情况，可以从 http://www.eclipse.org/mat 下载。

Android 2.3 定义了 StrictMode 类，它对检测潜在的内存泄漏有很大帮助。虽然在 Android 2.3 中，StrictMode 的虚拟机策略只能检测 SQLite 对象（如游标）没有关闭时产生的泄露，但在 Android 3.0 及以上版本中，可以检测以下潜在的泄漏：

- Activity 泄漏；
- 其他对象泄漏；
- 对象没有关闭造成的泄漏（至于是哪些对象，可到 Android 文档中查看实现了 Closeable 接口的所有类）。

注意 StrictMode 类在 Android 2.3（API 等级 9）引入，但其虚拟机和线程策略方面的额外功能是在 Android 3.0（API 等级 11）加进来的。例如，Honeycomb 的 StrictMode 线程策略可以在发现有违规操作时使屏幕闪烁。

代码清单 4-17 显示如何在应用中使用 StrictMode 类来检测内存泄漏。此功能仅在开发和测试时开启，应用发布时要禁用。

代码清单 4-17 使用 StrictMode

```
public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
```

① 两个技巧将 sdk 里面用到的工具的路径加入到系统 Path 中，使用“adb logcat -v time> 文件名”可以将 l 日志存下来。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
            StrictMode.VmPolicy.Builder builder = new StrictMode.VmPolicy.Builder();
            builder.detectLeakedSqlLiteObjects();
            if (VERSION.SDK_INT >= Build.VERSION_CODES.HONEYCOMB) {
                builder.detectActivityLeaks().detectLeakedClosableObjects();
            }
            // 或者可以简单地调用 builder.detectAll()

            // 采取措施
            builder.penaltyLog(); // 若有其他措施（如 penaltyDeath()），可以在这合并处理

            StrictMode.VmPolicy vmp = builder.build();
            StrictMode.setVmPolicy(vmp);
        }
    }
```

在这个特定实例中，当可关闭的对象（SQLite 或其他对象）没有关闭时，StrictMode 会检测到违规操作，并只将结果写到日志中。要验证行为，你可以试一下，查询数据库，故意忘记关闭返回的游标，比如略微修改代码清单 1-25。

现在 StrictMode 类功能增强了，建议只使用 detectAll()，随着 Android 版本的更新，它也会将新功能加进来一起检查。

4.5.2　引用

内存释放是垃圾收集器的一个重要的特性，在垃圾收集器中它的作用比在内存管理系统中大得多。关于引用及对象是否被引用对 Java 程序员已是老生常谈了。但很少有人知道引用还分许多种，事实上，Java 定义了 4 种类型的引用：

- 强（Strong）
- 软（Soft）
- 弱（Weak）
- 虚（Phantom）

1. 强引用

强引用是 Java 开发人员最熟悉的。创建这种引用不费吹灰之力，Java 程序员整天都在写。事实上，程序中绝大部分都是用这种引用，如代码清单 4-18 所示。此例创建了两个强引用，一个用于 Integer 对象，另一个用于 BigInteger 对象。

代码清单 4-18　强引用

```
public void printTwoStrings (int n) {
    BigInteger bi = BigInteger.valueOf(n); // 强引用
    Integer i = new Integer(n); // 强引用

    System.out.println(i.toString());
    i = null; // Integer 对象变成可回收的垃圾

    System.out.println(bi.toString());
    bi = null; // 这里 BigInteger 对象可能不会被回收
}
```

更多资源请访问稀酷客(www.ckook.com)

这里有个重要的问题，当把 i 置为 null 时，会使 Integer 对象被回收，设置 bi 为 null 却不会回收。因为 BigInteger.valueOf()方法可以返回一个预先分配的对象（比如 BigInteger.ZERO），设置 bi 为 null 只是删除 BigInteger 对象的一个强引用，但同一对象的强引用依然可能存在。这个方法中还创建了两个强引用，它们可能不像前面那两个显而易见：调用 i.toString()和 bi.toString()会分别创建一个 String 对象的强引用。

注意 严格地说，必须知道 Integer 构造函数的实现，才能确保创建新的 Integer 对象时并没有在其他地方创建强引用，才能保证 i 置为 null 时对象会被回收。

如前文所述，保持无用对象的强引用可能会导致内存泄漏。术语“强引用”我们提了好多遍了，其实 Java 并没有真的定义这样的术语或类。强引用就是“正常”引用，简称为引用。

2. 软、弱、虚引用

软引用和弱引用在本质上是相似的，它们是没有强到足以保持对象不被删除（或回收）的引用。不同之处在于回收时，垃圾收集器处理它们引用的对象的积极程度不同。

对象是软可及的，即存在一个软引用，但没有强引用，当有足够的内存保留对象时，垃圾收集器不会回收它。不过如果垃圾收集器决定需要回收更多的内存，那么它可任意回收软可及对象的内存。这种引用类型适用于缓存，它可以自动删除缓存中的条目。

提示 当使用缓存时，确保你了解它使用的是什么类型的引用。例如，Android 的 LruCache 使用强引用。

弱可及对象，也就是说，存在一个弱引用，但没有强或软引用，下次垃圾回收时基本会被收走。换言之，垃圾收集器会更加积极地回收弱可及对象的内存。这种类型的引用适用于映射，这种映射可以自动删除不再被引用的键，WeakHashMap 类就是这样做的。

注意 垃圾收集器的积极程度取决于实际实现。

虚引用最弱，几乎很少用到。如果你的应用需要知道一个对象什么时候被回收并需要在回收的同时执行一些清理工作，可以用它们。虚引用可用来注册引用队列。

软、弱和虚引用实际上是在对象本身和其他对象间加了一个间接层。例如，你可以创建一个虚引用指向一个软引用，软引用再指向弱引用；不过在实践中，创建的软、弱或虚引用几乎总是直接指向“强”引用。代码清单 4-19 创建了软引用和弱引用，分别关联了不同的引用队列。

代码清单 4-19 引用和引用队列

```
private Integer strongRef;
private SoftReference<Integer> softRef;
private WeakReference<Integer> weakRef;
```

更多资源请访问稀酷客(www.ckook.com)

```
private ReferenceQueue<Integer> softRefQueue = new ReferenceQueue<Integer>();
private ReferenceQueue<Integer> weakRefQueue = new ReferenceQueue<Integer>();

public void reset () {
    strongRef = new Integer(1);
    softRef = new SoftReference<Integer>(strongRef, softRefQueue);
    weakRef = new WeakReference<Integer>(strongRef, weakRefQueue);
}

public void clearStrong () {
    strongRef = null; // 没有强引用，弱及软引用可能还在
}

public void clearSoft () {
    softRef = null; // 没有软引用，强及弱引用可能还在
}

public void clearWeak () {
    weakRef = null; // 没有弱引用，可能强及软引用还在
}

public void pollAndPrint () {
    Reference<? extends Integer> r;
    if ((r = softRefQueue.poll()) != null) {
        do {
            Log.i(TAG, "Soft reference: " + r);
        } while ((r = softRefQueue.poll()) != null);
    } else {
        Log.i(TAG, "Soft reference queue empty");
    }
    if ((r = weakRefQueue.poll()) != null) {
        do {
            Log.i(TAG, "Weak reference: " + r);
        } while ((r = weakRefQueue.poll()) != null);
    } else {
        Log.i(TAG, "Weak reference queue empty");
    }
}

public void gc() {
    System.gc();
}
```

用这份代码实验一下，看看引用进入队列后对程序的影响。如果想充分利用垃圾收集器的内存管理能力，掌握引用非常重要。当需要缓存或映射时，你不必实现类似的内存管理系统。精心规划引用后，大部分工作可以放心地交给垃圾收集器完成。

3. 垃圾收集

垃圾收集可能会在不定的时间触发，你几乎无法控制它发生的时机。有时，你可以通过System.gc()提醒一下Android，但垃圾收集的发生时间最终由Dalvik虚拟机决定。有以下5种情况会触发垃圾收集，这些可以参考logcat中输出的消息。

更多资源请访问稀酷客(www.ckook.com)

- GC_FOR_MALLOC：发生在堆被占满不能进行内存分配时，在分配新对象之前必须进行内存回收。
- GC_CONCURRENT：发生在（可能是部分的）垃圾可供回收时，通常有很多对象可以回收。
- GC_EXPLICIT：显式调用 System.gc()产生的垃圾收集。
- GC_EXTERNAL_ALLOC：Honeycomb 及以上版本不会出现（一切都已在堆中分配）。
- GC_HPROF_DUMP_HEAP：发生在创建 HPROF 文件时。

代码清单 4-20 显示了一些垃圾收集器产生的日志信息。

代码清单 4-20 垃圾收集的讯息

```
GC_CONCURRENT freed 103K, 69% free 320K/1024K, external 0K/0K, paused 1ms+1ms
GC_EXPLICIT freed 2K, 55% free 2532K/5511K, external 1625K/2137K, paused 55ms
```

垃圾收集要花费时间，减少分配/释放对象的数量可以提高性能。在 Android 2.2 和更早的版本中尤其如此，因为垃圾收集发生在应用的主线程，很可能降低响应速度和性能，造成恶劣影响。例如，在即时游戏中会出现丢帧，因为有太多的时间花在垃圾收集上。Android 2.3 有了转机，垃圾收集工作转移到了一个单独的线程。垃圾收集时，还是会对主线程有点影响（暂停 5 毫秒或更少），但比以前的 Android 版本好太多了。一次完整的垃圾收集花了超过 50 毫秒的情况并不少见。试想一下，一个每秒 30 帧的游戏平均在每一帧上要花 33 毫秒进行渲染和显示，这时如果有垃圾回收，在 Android2.3 之前的系统中，游戏会大受影响。

4.6 API

Android 定义了几个 API，你可以用它们来了解系统中还剩多少可用内存和用了多少内存：

- ActivityManager 的 getMemoryInfo()
- ActivityManager 的 getMemoryClass()
- ActivityManager 的 getLargeMemoryClass()
- Debug 的 dumpHprofData()
- Debug 的 getMemoryInfo()
- Debug 的 getNativeHeapAllocatedSize()
- Debug 的 getNativeHeapSize()

提示 *在应用的 manifest 文件中把 android:largeHeap 设为 true，就可以让应用使用更大的堆。这个属性是 Android 3.0 加进来的。需要注意的是，它不保证“更大的堆”真的比常规使用的更大。不要让应用依赖这个配置。*

更多资源请访问稀酷客(www.ckook.com)

代码清单4-21显示了如何使用两个getMemoryInfo()方法。

代码清单4-21　调用getMemoryInfo()

```
ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);

ActivityManager.MemoryInfo memInfo = new ActivityManager.MemoryInfo();
am.getMemoryInfo(memInfo);
// 这里使用 memInfo 带来的数据...

Debug.MemoryInfo debugMemInfo = new Debug.MemoryInfo();
Debug.getMemoryInfo(debugMemInfo);
// 这里使用 debugMemInfo 的数据...

ActivityManager am = (ActivityManager) getSystemService(Context.ACTIVITY_SERVICE);

ActivityManager.MemoryInfo memInfo = new ActivityManager.MemoryInfo();
am.getMemoryInfo(memInfo);
// 这里使用 memInfo 带来的数据...

Debug.MemoryInfo debugMemInfo = new Debug.MemoryInfo();
Debug.getMemoryInfo(debugMemInfo);
// 这里使用 debugMemInfo 的数据...
```

4.7　内存少的时候

你的应用并不能独占平台，它要与许多其他应用及系统作为一个整体来共享资源。因此，当内存不足以分配给所有程序的情况下，Android会要求应用及应用的组件（如Activity或Fragment）“勒紧腰带”。

ComponentCallbacks接口定义了API onLowMemory()，它对所有应用组件都是相同的。当它被调用时，组件基本会被要求释放那些并不会用到的内存。通常情况下，onLowMemory()的实现将释放：

- 缓存或缓存条目（如使用强引用的LruCache）；
- 可以再次按需生成的位图对象；
- 不可见的的布局对象；
- 数据库对象。

删除对象应该小心翼翼，重新创建是需要开销的。如果没有释放出足够的内存可能会导致Android系统更激进的行为（如杀死进程），谁也不能独善其身。如果应用进程被杀掉了，用户下次使用又要从头开始。因此，应用不仅要表现出色，也要释放尽可能多的资源，这样的结果是多赢的，对你的应用和其他应用都有好处。在代码中使用推迟初始化是一个好习惯，它可以让你在实现onLowMemory()时几乎不用改动其他地方的代码。

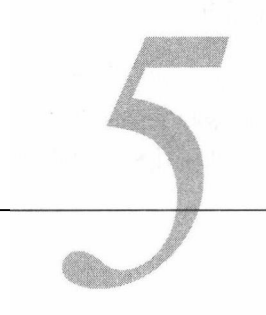
更多资源请访问稀酷客(www.ckook.com)

4.8 总结

内存在嵌入式设备上是稀缺资源。尽管今天的手机和平板电脑的内存越来越多，但这些设备也在运行越来越复杂的系统和应用。有效地使用内存，不仅可以使应用在旧设备上运行时占用较少的内存，还可让程序跑得更快。请记住，应用对内存的需求是无止境的。

更多资源请访问稀酷客(www.ckook.com)

第5章 多线程和同步



第1章介绍了主线程（或UI线程），它处理了大多数的Android事件。即便可以在主线程内执行所有代码，但应用一般都会使用多个线程。事实上，即便你自己没创建，应用启动时也会自动创建并运行几个线程。例如，在Android 3.1的Galaxy Tab 10.1上，使用Eclipse的DDMS视图查看应用时可看到下列线程：

- main
- HeapWorker（执行finalize函数和引用对象清理）
- GC（Garbage Collector，垃圾收集）
- Signal Catcher（捕捉Linux信号进行处理）
- JDWP（Java Debug Wire Protocol，调试协议服务）
- Compiler（JIT compiler，即时编译器）
- Binder Thread #1（Binder通讯）
- Binder Thread #2

到目前为止，我们仅讨论了列表中的第1个——主线程，你可能没想到还有另外7个。不过其他线程用不着管，它们大多数干的都是管家的工作，干的活也不受你的掌控。我们要关注主线程，不在其内部进行耗时长的操作，让应用程序可以快速响应。

应用运行的Android环境版本决定了会派生哪些管家线程。例如，垃圾收集作为一个独立线程只出现在Android 2.3和其后版本中，Android 2.2还没有即时编译器，Android 2.1只生成了6个线程（而不是上面列出的8个）。

本章介绍创建线程、线程间通讯、在线程间安全共享对象的方法，以及在一般情况下，如何写好使用多线程的代码。我们也会讲在Android中使用多线程时会遇到哪些常见陷阱。

5.1 线程

Thread对象，也就是Java定义的Thread类的实例，是自己带有调用栈的执行单位。如代码清单5-1所示，应用可以轻易创建额外的线程。当然，应用可以随意创建线程在主线程之外执行一些操作，很多时候只有这样做才能让应用保持快速响应。

更多资源请访问稀酷客(www.ckook.com)

代码清单 5-1 创建两个线程

```
// run()方法可以被覆写...
Thread thread1 = new Thread("cheese1") {
    @Override
    public void run() {
        Log.i("thread1", "I like Munster");
    }
};

// 或将 Runnable 对象传递给 Thread 构造函数
Thread thread2 = new Thread(new Runnable() {
    public void run() {
        Log.i("thread2", "I like Roquefort");
    }
}, "cheese2");

// 记得调用 start()，否则不会产生任何线程，什么都不会发生
thread1.start();
thread2.start();
```

执行该代码实际上可能会输出不同结果。因为每个线程都是独立的执行单位，两个线程具有相同的默认优先级，即使“I like Munster”线程先启动，系统不会确保它会显示在“I like Roquefort”之前。实际结果取决于系统调度的实现方式。

注意 一个典型的错误是调用 run()方法，而不是 start()。这会让 Thread 对象（或 Runnable 对象，视情况而定）的 run()方法，在当前线程中被调用执行。换言之，没有产生新线程。

上面只是个简单的两线程示例，还没有把需要的线程执行结果反馈回去。虽然有时需要这么做，但我们往往想要的是在不同线程执行操作后的结果。例如，应用可能需要在一个单独的线程计算斐波那契数，让应用保持响应，但最终还是要更新用户界面来显示计算结果。代码清单 5-2 的场景中，mTextView 是应用布局中 TextView 部件的引用，在用户点击按钮时会调用 onClick 方法，或者在点击一般的视图时也会调用（见 Android XML 布局中的 android:onClick 属性）。

代码清单 5-2 计算斐波那契数的工作线程

```
public void onClick (View v) {
    new Thread(new Runnable() {
        public void run() {
            // 注意这里的 "final" 关键字（试试删除它，看看会发生什么现象）
            final BigInteger f = Fibonacci.recursiveFasterPrimitiveAndBigInteger(100000);
            mTextView.post(new Runnable() {
                public void run() {
                    mTextView.setText(f.toString());
                }
            });
        }
    }, "fibonacci").start();
}
```

更多资源请访问稀酷客(www.ckook.com)

虽然这段代码可以正常工作，却绕了不少弯子，也难以阅读和维护。你可能会有简化代码清单 5-2 代码的冲动，换成代码清单 5-3 所示代码。但这是个坏主意，它会直接抛出异常 CalledFromWrong-ThreadException，因为 Android 的 UI 函数只能从 UI 线程调用。异常的描述信息说："只有原来创建了视图树的线程才可以访问它自己的视图。"这样就强制应用只能在 UI 线程中调用 TextView.setText()，例如要将 Runnable 对象传递到 UI 线程中处理。

代码清单 5-3　非 UI 线程的无效调用

```
public void onClick(View v) {
    new Thread(new Runnable() {
        public void run() {
        BigInteger f = Fibonacci.recursiveFasterPrimitiveAndBigInteger(100000);
        mTextView.setText(f.toString()); // 会抛出异常
        }
    }, "fibonacci").start();
}
```

提示　为了方便调试，最好给自己生成的线程起个有意义的名字。如果没有起名字，系统也会给线程自动分配一个新名字（只不过没什么含义）。调用 Thread.getName()可以看到线程的名字。

不管线程是通过什么方式创建，它都有一个优先级。调度器根据优先级来决定调度执行哪个线程，即让哪个线程使用 CPU。调用 Thread.setPriority()可以改变线程优先级，如代码清单 5-4 所示。

代码清单 5-4　设置线程优先级

```
Thread thread = new Thread("thread name") {
    @Override
    public void run() {
        // 在这里做事情
    }
};
thread.setPriority(Thread.MAX_PRIORITY); // 最高优先级 （比 UI 线程高）
thread.start();
```

如果未指定优先级，会使用默认值。Thread 类定义了三个常量：

- MIN_PRIORITY (1)
- NORM_PRIORITY (5)—— 默认优先级
- MAX_PRIORITY (10)

如果应用设置的线程优先级超出取值范围，也就是说，小于 1 或大于 10，那么会抛出 llegal-ArgumentException 异常。

另一种设置线程优先级的方式是基于 Linux 优先级，使用 android.os 包里的 Process.setThreadPriority API。它定义了 8 个优先级：

更多资源请访问稀酷客(www.ckook.com)

- THREAD_PRIORITY_AUDIO (-16)
- THREAD_PRIORITY_BACKGROUND (10)
- THREAD_PRIORITY_DEFAULT (0)
- THREAD_PRIORITY_DISPLAY (-4)
- THREAD_PRIORITY_FOREGROUND (-2)
- THREAD_PRIORITY_LOWEST (19)
- THREAD_PRIORITY_URGENT_AUDIO (-19)
- THREAD_PRIORITY_URGENT_DISPLAY (-8)

还可以使用 Process.THREAD_PRIORITY_LESS_FAVORABLE (+1) 和 Process.THREAD_PRIORITY_MORE_FAVORABLE (-1) 略作调整。例如，要把线程优先级设置比默认值略高，可以设置优先级为 (THREAD_PRIORITY_DEFAULT + THREAD_PRIORITY_MORE_FAVORABLE)。

提示 使用 THREAD_PRIORITY_LESS_FAVORABLE 和 THREAD_PRIORITY_MORE_FAVORABLE，而不是+1和-1，这样你就不用反复记住较高的数字是否意味着更高的优先级。此外，应当避免混用 Thread.setPriority 和 Process.setThreadPriority，这会使代码一团糟。注意，Linux 的优先级是从-20（最高）到 19（最低），而线程的优先级是从 1（最低）到 10（最高）。

要决定改变线程的优先级时，必须非常小心。增加一个线程的优先级可能会加快这个线程的任务执行速度，但也会对其他线程造成负面影响，让它们无法及时获取到 CPU 资源，从而扰乱了整体的用户体验。如果需要这样做，可以考虑使用优先级老化算法。

在 Android 中创建线程并在后台执行非常简单（如代码清单 5-1 所示），但更新用户界面却有些麻烦：因为 View 的方法只能在 UI 线程中调用，所以必须把结果反馈给主线程处理。

5.2 AsyncTask

很多情况下，应用处理顺序如代码清单 5-2 所示：

- 在 UI 线程收到事件；
- 在非 UI 线程中处理相应事件；
- UI 根据处理结果进行刷新。

为了简化这个通用的模式，Android 1.5 及以上版本定义了 AsyncTask 类。AsyncTask 类可以让应用执行后台操作，并通知 UI 线程操作结果。为简单起见，对用户隐藏 Thread、Runnable 以及其他相关对象。代码清单 5-5 所示使用 AsyncTask 类实现代码清单 5-2 的操作序列。

代码清单 5-5 使用 AsyncTask

```
public void onClick (View v) {
    // AsyncTask<启动任务执行的输入参数，后台任务执行的百分比，后台执行任务返回的结果，匿名类
```

更多资源请访问稀酷客(www.ckook.com)

```
        new AsyncTask<Integer, Void, BigInteger>() {
            @Override
            protected BigInteger doInBackground(Integer... params) {
                return Fibonacci.recursiveFasterPrimitiveAndBigInteger(params[0]);
            }

            @Override
            protected void onPostExecute(BigInteger result) {
                mTextView.setText(result.toString());
            }
        }.execute(100000);
    }
```

doInBackground()是需要实现的抽象方法。不覆写 onPostExecute()也是可以的，这个方法就是用来做你想让 AsyncTask 做的事，将结果反馈给 UI。下列 AsyncTask 保护方法会从 UI 线程调用：

- onPreExecute()
- onProgressUpdate(Progress… values)
- onPostExecute(Result result)
- onCancelled()
- onCancelled(Result result)（Android 3.0 API）

onProgressUpdate()要在 doInBackground()方法中紧接着 publishProgress()之后调用，用来在 UI 上更新进度。典型的例子是后台下载文件时刷新前台进度条。代码清单 5-6 是可以下载多个文件的方法。

代码清单 5-6　下载多个文件

```
AsyncTask<String, Object, Void> task = new AsyncTask<String, Object, Void>() {

    private ByteArrayBuffer downloadFile(String urlString, byte[] buffer) {
        try {
            URL url = new URL(urlString);
            URLConnection connection = url.openConnection();
            InputStream is = connection.getInputStream();
            //Log.i(TAG, "InputStream: " + is.getClass().getName()); // 如果想知道
            //is = new BufferedInputStream(is); // 这行可选
            ByteArrayBuffer baf = new ByteArrayBuffer(640 * 1024);
            int len;
            while ((len = is.read(buffer)) != -1) {
                baf.append(buffer, 0, len);
            }
            return baf;
        } catch (MalformedURLException e) {
            return null;
        } catch (IOException e) {
            return null;
        }
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
    @Override
    protected Void doInBackground(String... params) {
        if (params != null && params.length > 0) {
            byte[] buffer = new byte[4 * 1024]; // 试试不同的大小 （例如，1会严重降低性能）
            for (String url : params) {
                long time = System.currentTimeMillis();
                ByteArrayBuffer baf = downloadFile(url, buffer);
                time = System.currentTimeMillis() - time;
                publishProgress(url, baf, time);
            }
        } else {
            publishProgress(null, null);
        }
        return null; // 即使不关心任何结果，也要返回东西
    }

    @Override
    protected void onProgressUpdate(Object... values) {
        // values[0] 是 URL (String) , values[1] 是缓冲区 (ByteArrayBuffer) , values[2] 是时长
        String url = (String) values[0];
        ByteArrayBuffer buffer = (ByteArrayBuffer) values[1];
        if (buffer != null) {
            long time = (Long) values[2];
            Log.i(TAG, "Downloaded " + url + " (" + buffer.length() + " bytes) in " + time + "
                  milliseconds");
        } else {
            Log.w(TAG, "Could not download " + url);
        }

        // 看情况更新 UI
    }
};

String url1 = "http://www.google.com/index.html";
String url2 = "http://d.android.com/reference/android/os/AsyncTask.html";
task.execute(url1, url2);
//task.execute("http://d.android.com/resources/articles/painless-threading.html"); // 试下这个会抛
                                                                                    出异常
```

在代码清单 5-6 中的例子只是下载文件到内存（ByteArrayBuffer 对象）。如果你想存储文件，也要在 UI 以外的线程执行该操作。另外，此例也演示了一个接一个下载文件的方法。看应用需求，并行下载多个文件也许会更好。

注意 必须在 UI 线程中创建 AsyncTask 对象，只能执行一次（即调用 execute()一次）。

实际上，doInBackground()调用时机取决于 Android 版本。Android 1.6 之前，任务是串行执行的，只需一个后台线程。从 Android 1.6 开始，线程池取代了单个的后台线程，线程池允许并行执行多个任务，以提升性能。然而，并行执行多个任务，如果没有正确同步或任务执行顺序不符合开发者的预期，那很可能会导致严重的问题。因此，Android 团队计划在 Honeycomb 之后，

更多资源请访问稀酷客(www.ckook.com)

恢复到默认情况下只有一个后台线程的模式。为了继续让应用可以并行执行任务，在 Honeycomb 中添加了一个新的 API executeOnExecutor()，为开发者提供缓冲时间来更新应用。这个新的 API 可以与 AsyncTask.SERIAL_EXECUTOR 串行执行或 AsyncTask.THREAD_POOL_EXECUTOR 并行执行一起使用。

未来的变化表明，并行执行需要精心设计和全面的测试。Android 团队可能低估了切换到 Android 1.6 中的线程池的潜在问题或高估了应用处理问题的能力，促使他们决定在 Honeycomb 之后恢复到单线程模式。很多经验丰富的开发者在灵活采用并行多任务的手段提升性能的同时，还能提高应用的整体性能。

AsyncTask 类可以简化处理后台任务和用户界面更新的代码，但它并能完全取代 Android 定义的线程之间的基本通信类。

5.3 Handler 和 Looper

Android 在 android.os 包中定义了两个类，它们通常是多线程应用线程间通信的基石：

- Handler
- Looper

创建的 AsyncTask 对象隐藏了 Handler 和 Looper 的细节，在某些情况下，还是要直接跟 Handler 和 Looper 打交道，比如把 Runnable 对象传递到主线程之外的线程。

5.3.1 Handler

代码清单 5-2 道出一点 Handler 和 Looper 合作的玄机：使用 Handler 发送 Runnable 对象到 Looper 的消息队列中。应用的主线程已经有一个消息队列，不必显式创建。不过，自己创建的线程不带消息队列和消息循环，如果需要只能自己创建。代码清单 5-7 显示了如何创建带有 Looper 的线程。

代码清单 5-7　带有消息队列的线程类

```
public class MyThread extends Thread {
    private static final String TAG = "MyThread";
    private Handler mHandler;

    public MyThread(String name) {
        super(name);
    }

    public Handler getHandler() {
        return mHandler;
    }

    @Override
    public void run() {
        Looper.prepare(); // 把 Looper 绑定到此线程
```

更多资源请访问稀酷客(www.ckook.com)

```
        mHandler = new Handler() {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    // 这里处理消息
                }
            }
        };
        // Handler 绑定到此线程的 Looper

        Looper.loop();  // 不要忘了调用 loop()去启动消息循环

        // 只有循环停止了（比如调用了 Looper.quit()），Looper 才会返回
    }
}
```

注意 Handler 对象在 run()方法中创建，因为它需要被绑定到指定的 Looper，这个 Looper 就是在 run()方法中调用 Looper.prepare()创建的。因此在线程产生之前，调用 getHandler()将返回 null。

在线程运行起来之后，就可以发送 Runnable 对象或消息到消息队列，如代码清单 5-8 所示。

代码清单 5-8 发送 Runnable 对象及消息

```
MyThread thread = new MyThread("looper thread");
thread.start();

// 过一会儿
Handler handler = thread.getHandler();
// 小心：如果处理程序尚未初始化，这里会返回 null!

// 发送 runnable 对象
handler.post(new Runnable() {
    public void run() {
        Log.i(TAG, "Where am I? " + Thread.currentThread().getName());
    }
});

// 发送消息
int what = 0; // 定义自己的值
int arg1 = 1;
int arg2 = 2;
Message msg = Message.obtain(handler, what, arg1, arg2);
handler.sendMessage(msg);

// 另一条消息
what = 1;
msg = Message.obtain(handler, what, new Long(Thread.currentThread().getId()));
handler.sendMessageAtFrontOfQueue(msg);
```

更多资源请访问稀酷客(www.ckook.com)

提示　使用 Message.obtain()或 Handler.obtainMessage() API 来获得 Message 对象，因为它们从全局的消息池中取出 Message 对象，这比每次单独创建新实例获取 Message 效率更高。利用这些 API 设置消息中携带的各种字段也很容易。

5.3.2 Looper

Android 提供了 HandlerThread 类来帮助更方便地使用 Looper 的线程，它也利于避免潜在的竞争状态，如代码清单 5-8 提到的，尽管线程已经启动，getHandler()也可能返回 null。代码清单 5-9 示范了如何使用 HandlerThread 类。

代码清单 5-9　使用 HandlerThread 类

```
public class MyHandlerThread extends HandlerThread {
    private static final String TAG = "MyHandlerThread";
    private Handler mHandler;

    public MyHandlerThread(String name) {
        super(name);
    }

    public Handler getHandler() {
        return mHandler;
    }

    @Override
    public void start() {
        super.start();
        Looper looper = getLooper(); // 这里会一直阻塞到线程的 Looper 对象初始化结束
        mHandler = new Handler(looper) {
            @Override
            public void handleMessage(Message msg) {
                switch (msg.what) {
                    // 这里处理消息
                }
            }
        };
    }
}
```

由于 Handler 对象是在 start()方法而不是在 run()方法创建的，只要在 start()之后就可以用 getHandler()获取，不会出现任何竞争状态。

5.4 数据类型

我们已经知道两种产生线程的方法，使用 Thread 和 AsyncTask 类。如果两个或多个线程访问相同的数据，就需要确保数据类型支持并发访问。Java 语言在 java.util.concurrent 包中定义了许

更多资源请访问稀酷客(www.ckook.com)

多用于并发的类：

- ArrayBlockingQueue
- ConcurrentHashMap
- ConcurrentLinkedQueue
- ConcurrentSkipListMap
- ConcurrentSkipListSet
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- DelayQueue
- LinkedBlockingDeque
- LinkedBlockingQueue
- PriorityBlockingQueue
- SynchronousQueue

你要根据应用需求，谨慎选择数据类型。此外，它们支持的并发实现并不一定意味着操作是原子的。事实上，许多操作都不是原子的，也没有照原子操作设计。例如，ConcurrentSkipListMap类的putAll()方法就不是原子的。并发实现仅仅是指当多个线程访问时，数据结构不会被破坏。

同步、易变、内存模型

如果想在多个线程之间共享对象，但没有实现任何细粒度的锁机制，可以使用synchronized关键字确保数据访问是线程安全的，如代码清单5-10所示。

代码清单5-10 使用synchronized关键字

```
public class MyClass {
    private int mValue;

    public MyClass(int n) {
        mValue = n;
    }

    public synchronized void add (int a) {
        mValue += a;
    }

    public synchronized void multiplyAndAdd (int m, int a) {
        mValue = mValue * m + a;
    }
}
```

代码清单5-7中的add()和multiplyAndAdd()两个方法是同步的。这意味着两件事情：

- 如果一个线程正在执行同步的方法，其他线程试图调用同一对象的同步方法必须要等到前一个线程完成；

更多资源请访问稀酷客(www.ckook.com)

- 当同步的方法退出后，所有其他线程可看到对象的更新状态。

第一条很直观，第二条也好理解，不过还有些需要解释。事实上，Java 的内存模型就是这样，一个线程中变量的修改可能不会立即对其他线程可见。事实上，也许会永远不可见。考虑代码清单 5-11 的代码：如果一个线程调用 MyClass.loop()，将来的某个时间点，另一个线程调用了 Myclass.setValue(100)，第一个线程可能仍然不会终止，可能永远循环下去，不断打印出超过 100 的值，这是由 Java 语言的内存模型造成的。

代码清单 5-11　Java 内存模型的影响

```
public class MyClass {
    private static final String TAG = "MyClass";
    private static int mValue = 0;

    public static void setValue(int n) {
        mValue = n;
    }

    public static void loop () {
        while (mValue != 100) {
            try {
                Log.i(TAG,  "Value is " + mValue);
                Thread.sleep(1000);
            } catch (Exception e) {
                // 忽略
            }
        }
    }
}
```

上述问题有两种解决方案：

- 使用 synchronized 关键字，如代码清单 5-12 所示；
- 使用 volatile 关键字，如代码清单 5-13 所示。

代码清单 5-12　添加 synchronized 关键字

```
public class MyClass {
    private static final String TAG = "MyClass";
    private static int mValue = 0;

    public static synchronized void setValue(int n) {
        mValue = n;
    }

    public static synchronized int getValue() {
        return mValue;
    }

    public static void loop () {
        int value;
        while ((value = getValue()) != 100) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            try {
                Log.i(TAG,  "Value is " + value);
                Thread.sleep(1000);
            } catch (Exception e) {
                // 忽略
            }
        }
    }
}
```

代码清单 5-13 添加 volatile 关键字

```
public class MyClass {
    private static final String TAG = "MyClass";
    private static volatile int mValue = 0; // 添加 volatile 关键字，删除同步关键字

    public static void setValue(int n) {
        mValue = n; // 如果是 mValue += n 语句，因为不是原子操作，还得使用 synchronized
    }

    public static void loop () {
        while (mValue != 100) {
            try {
                Log.i(TAG, "Value is " + mValue);
                Thread.sleep(1000);
            } catch (Exception e) {
                // 忽略
            }
        }
    }
}
```

注意 你需要了解哪些语句是原子的。例如，value++不是原子的，而 value= 1 是。volatile 关键字只能解决变量声明是原子的那些并发问题，这一点很重要。如果变量不是原子的，就必须使用 synchronized 关键字。

可以用同步语句提高并发性和吞吐量，如代码清单 5-14 所示，但不要同步整个方法。在这些情况下，只需要保护必要的（也就是正在修改 mValue）部分，日志消息放到同步区之外。你还可以用一个对象作为锁，而不是 this。

代码清单 5-14 同步语句

```
public class MyOtherClass {
    private static final String TAG = "MyOtherClass";
    private int mValue;
    private Object myLock = new Object(); // 用这个对象而不是 this

    public MyClass(int n) {
        mValue = n;
```

更多资源请访问稀酷客(www.ckook.com)

```
    }

    public void add (int a) {
        synchronized (myLock) {
            mValue += a;
        }
        Log.i(TAG, "add: a=" + a); //不必加锁
    }

    public void multiplyAndAdd (int m, int a) {
        synchronized (myLock) {
            mValue = mValue * m + a;
        }
        Log.i(TAG, " multiplyAndAdd: m=" + m + ", a=" + a); // 不必加锁
    }
}
```

使用方法或语句同步是保证类支持并发访问的最简单方法。但是，它可能会降低吞吐量，并不是所有东西都要加锁保护，更糟糕的是，它可能会导致死锁（Dead Lock）。事实上，往往发生死锁的情况是，在一个同步代码块中调用另一个对象的方法，而那个对象已经被锁住并等待之前用到的对象解锁。

提示　不要在 synchronized 块内调用另一个对象的方法，除非能保证不会发生死锁。通常情况下，只有亲手写其他对象的类的代码才敢确定不会发生死锁。

在一般情况下，如果不能确保代码能否正常工作，最好避免从不同的线程访问同一个对象。Java 和 Android 定义的类是很好的老师，你可以参考这些类的实现，了解各种可以利用的技术。此外，为简化开发，Java 定义了许多支持线程安全或并发编程的类，可以在此基础上编写自己的算法。

5.5　并发

java.util.concurrent.atomic 和 java.util.concurrent.locks 包还定义了很多类。java.util.concurrent.atomic 包含以下类：

- AtomicBoolean
- AtomicInteger
- AtomicIntegerArray
- AtomicIntegerFieldUpdater（抽象类）
- AtomicLong
- AtomicLongArray
- AtomicLongFieldUpdater（抽象类）
- AtomicMarkableReference

更多资源请访问稀酷客(www.ckook.com)

- AtomicReference
- AtomicReferenceArray
- AtomicReferenceFieldUpdater（抽象类）
- AtomicStampedReference

这些类需要略作解释，它们只定义了些原子方式更新值的方法。例如，AtomicInteger 类定义了 addAndGet()方法，此方法在 AtomicInteger 对象的当前值上加一个值，并返回更新后的值。在这个包中定义的抽象类供内部使用，很少会被直接用在应用的代码中。

除 java.util.concurrent 包中的 CountDownLatch、CyclicBarrier 以及 Semaphore 外，java.util.concurrent.locks 包还定义了同步的辅助工具：

- AbstractOwnableSynchronizer（抽象类，API 等级 5 引入）
- AbstractQueuedLongSynchronizer（抽象类，API 等级 9 引入）
- AbstractQueuedLongSynchronizer（API 等级 9 引入）
- AbstractQueuedSynchronizer（抽象类）
- AbstractQueuedSynchronizer.ConditionObject
- LockSupport
- ReentrantLock
- ReentrantReadWriteLock
- ReentrantReadWriteLock.ReadLock
- ReentrantReadWriteLock.WriteLock

这些类在 Android 应用中不常用。或许其中你最有可能用到的是的 ReentrantReadWriteLock 类，连同 ReentrantReadWriteLock.ReadLock 和 ReentrantReadWriteLock.WriteLock，它们允许多个读线程同时访问数据（只要没写线程修改数据），写线程同时只存在一个。当多个线程同时只读访问相同的数据时，要想最大化吞吐量，通常可以用这个对象。

一般来说，线程之间共享数据容易滋生吞吐量、并发等问题。同步会变得相当复杂，你要对这一领域知识和代码有扎实深刻的理解，才能保证共享数据不发生问题。调试同步问题可能会让人抓狂，所以你要先使用最简单直接的方式，然后逐步考虑吞吐量等优化。要注重质量甚于优化。

5.6 多核

最近，基于多核架构的 Android 设备已经越来越多了。例如，三星 Galaxy Tab 10.1 和摩托罗拉 Xoom 平板电脑都使用了双核处理器（Cortex A9 核心）。多核处理器不同于单核处理器，可以同时执行多个线程。可以说，在理论上，双核心处理器的性能是单核心的两倍（其他一切没变，比如时钟频率）。尽管多核优化并不像听起来那么容易，它有一些难点，但应用绝对可以充分利用今天的多核处理器的优势。[①]双核 CPU 的设备包括：

① 翻译时双核心已经差不多算中高端主流了，四核心设备已经问世了。——译者注

更多资源请访问稀酷客(www.ckook.com)

- 三星 Galaxy Tab 10.1
- 摩托罗拉 Xoom
- 摩托罗拉 Phonton 4G
- 摩托罗拉 Droid 3
- HTC EVO 3D
- LG Optimus 2X
- 三星 Galaxy Nexus

在许多情况下，不用操心设备有多少个内核。像在单核上创建多线程一样，使用 Thread 对象或者 AsyncTask，将工作分派给不同线程就足够了。如果处理器是多核的，这些线程会运行在不同核心上，这个行为对应用是透明的。

但可能有时要最有效地使用 CPU 来获得可接受的性能，要特别为多核定制算法。

为了达到最佳的性能，应用需要先检查有几个核心，这可以通过 RunTime.availableProcessors()方法得到，如代码清单 5-15 所示。

代码清单 5-15　获取处理器核心数量

```
// Galaxy Tab 10.1 或 BeBox Dual603 返回 2，Nexus S 或 Logitech Revue 返回 1
final int proc = Runtime.getRuntime().availableProcessors();
```

通常情况下，“可用处理器”数量是 1 或 2，未来的产品会采用四核心 CPU。目前 Android 的笔记本电脑可能已使用四核心架构。根据应用的发布时机，你可能先只需关注 1 或 2 核心的 CPU，以后再发布针对更多核的更新。

注意　假设核心数量可能并不总是 2 的幂。

5.6.1　为多核修改算法

在第 1 章介绍的斐波那契算法可以利用多核优势。首先，从代码清单 5-16（实现和第 1 章的代码清单 1-7 相同）所示的分治算法开始。

代码清单 5-16　斐波那契分治算法

```
public class Fibonacci
{
    public static BigInteger recursiveFasterBigInteger (int n)
    {
        if (n > 1) {
            int m = (n / 2) + (n & 1);

            // 两个简单的子问题
            BigInteger fM = recursiveFasterBigInteger(m);
            BigInteger fM_1 = recursiveFasterBigInteger(m - 1);
```

更多资源请访问稀酷客(www.ckook.com)

```
            // 合并结果，计算出原问题的解
            if ((n & 1) == 1) {
                // F(m)^2 + F(m-1)^2
                return fM.pow(2).add(fM_1.pow(2));
            } else {
                // (2*F(m-1) + F(m)) * F(m)
                return fM_1.shiftLeft(1).add(fM).multiply(fM);
            }
        }
        return (n == 0) ? BigInteger.ZERO : BigInteger.ONE;
    }
}
```

这个分治算法做了以下事情：

- 将原问题划分成更简单的子问题；
- 然后将子问题的结果合并处理，算出原问题的解。

由于两个子问题是独立的，它们可以并行处理而不需要大量同步。Java 语言定义了 ExecutorService 接口，你可以用实现这个接口的几个类来调度任务。如代码清单 5-17 所示，使用 Executors 类的 factory 方法来创建线程池。

代码清单 5-17 使用 ExecutorService

```
public class Fibonacci {
    private static final int proc = Runtime.getRuntime().availableProcessors();
    private static final ExecutorService executorService =
Executors.newFixedThreadPool(proc + 2);

    public static BigInteger recursiveFasterBigInteger (int n) {
        // 参阅代码清单 5-16 的实现
    }

    public static BigInteger recursiveFasterBigIntegerAndThreading (int n) {
        int proc = Runtime.getRuntime().availableProcessors();
        if (n < 128 || proc <= 1) {
            return recursiveFasterBigInteger(n);
        }

        final int m = (n / 2) + (n & 1);
        Callable<BigInteger> callable = new Callable<BigInteger>() {
            public BigInteger call() throws Exception {
                return recursiveFasterBigInteger(m);
            }
        };
        Future<BigInteger> ffM = executorService.submit(callable); // 尽早提交第一个任务

        callable = new Callable<BigInteger>() {
            public BigInteger call() throws Exception {
                return recursiveFasterBigInteger(m-1);
            }
        };
        Future<BigInteger> ffM_1 = executorService.submit(callable); // 提交第二个任务
```

更多资源请访问稀酷客(www.ckook.com)

```
        // 得到各部分结果，进行合并
        BigInteger fM, fM_1, fN;

        try {
            fM = ffM.get(); // 获取第一个子问题的结果（阻塞调用）
        } catch (Exception e) {
            // 如果有异常，在当前线程计算 fM
            fM = recursiveFasterBigInteger(m);
        }
        try {
            fM_1 = ffM_1.get(); // 获取第二个子问题的结果（阻塞调用）
        } catch (Exception e) {
            // 如果有异常，在当前线程计算 fM
            fM_1 = recursiveFasterBigInteger(m-1);
        }

        if ((n & 1) != 0) {
            fN = fM.pow(2).add(fM_1.pow(2));
        } else {
            fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
        }

        return fN;
    }
}
```

你可以看到，代码可读性变差了。此外，这个实现仍然基于一段低性能的代码：就像我们在第1章中所看到的，两个子问题最终会计算很多相同的斐波那契数。更好的实现方式是缓存已经计算出的斐波那契数，这样能节省大量时间。代码清单5-18的实现非常类似，但使用了缓存。

代码清单5-18　使用ExecutorService和缓存

```
public class Fibonacci {
    private static final int proc = Runtime.getRuntime().availableProcessors();
    private static final ExecutorService executorService = Executors.newFixedThreadPool(proc + 2);

    private static BigInteger recursiveFasterWithCache (int n, Map<Integer, BigInteger> cache)
    {
        // 参考代码清单1-11实现（稍微有点不同，原来采用的是SparseArray）
    }

    public static BigInteger recursiveFasterWithCache (int n)
    {
        HashMap<Integer , BigInteger> cache = new HashMap<Integer , BigInteger>();
        return recursiveFasterWithCache(n, cache);
    }

    public static BigInteger recursiveFasterWithCacheAndThreading (int n) {
        int proc = Runtime.getRuntime().availableProcessors();
        if (n < 128 || proc <= 1) {
            return recursiveFasterWithCache (n);
        }
```

更多资源请访问稀酷客(www.ckook.com)

```
            final int m = (n / 2) + (n & 1);
            Callable<BigInteger> callable = new Callable<BigInteger>() {
                public BigInteger call() throws Exception {
                    return recursiveFasterWithCache (m);
                }
            };
            Future<BigInteger> ffM = executorService.submit(callable);

            callable = new Callable<BigInteger>() {
                public BigInteger call() throws Exception {
                    return recursiveFasterWithCache (m-1);
                }
            };
            Future<BigInteger> ffM_1 = executorService.submit(callable);

            // 得到各部分结果，进行合并
            BigInteger fM, fM_1, fN;

            try {
                fM = ffM.get(); // 获取第一个子问题的结果（阻塞调用）
            } catch (Exception e) {
                // 如果有异常，在当前线程计算 fM
                fM = recursiveFasterBigInteger(m);
            }
            try {
                fM_1 = ffM_1.get(); // 获取第二个子问题的结果（阻塞调用）
            } catch (Exception e) {
                // 如果有异常，在当前线程计算 fM
                fM_1 = recursiveFasterBigInteger(m-1);
            }

            if ((n & 1) != 0) {
                fN = fM.pow(2).add(fM_1.pow(2));
            } else {
                fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
            }

            return fN;
        }
    }
```

5.6.2 使用并发缓存

注意，此实现存在一个问题，每个子问题使用自己的缓存对象，因此有些值被重复计算。为了让两个子问题共享一个缓存，我们需要把 SparseArray 对象改为可以从不同线程并发访问的对象。代码清单 5-19 是个示例，使用了 ConcurrentHashMap 对象作缓存。

代码清单 5-19 使用 ExecutorService 和单一缓存

```
public class Fibonacci {
    private static final int proc = Runtime.getRuntime().availableProcessors();
```

更多资源请访问稀酷客(www.ckook.com)

```
private static final ExecutorService executorService = Executors.newFixedThreadPool(proc + 2);

private static BigInteger recursiveFasterWithCache (int n, Map<Integer, BigInteger> cache) {
    // 参考代码清单 1-11 实现 （稍微有点不同，原来采用的是 SparseArray）
}

public static BigInteger recursiveFasterWithCache (int n) {
    HashMap<Integer , BigInteger> cache = new HashMap<Integer , BigInteger>();
    return recursiveFasterWithCache(n, cache);
}

public static BigInteger recursiveFasterWithCacheAndThreading (int n) {
    int proc = Runtime.getRuntime().availableProcessors();
    if (n < 128 || proc <= 1) {
        return recursiveFasterWithCache (n);
    }

    final ConcurrentHashMap<Integer, BigInteger> cache = new ConcurrentHashMap<Integer,
        BigInteger>(); // 并发访问正常

    final int m = (n / 2) + (n & 1);

    Callable<BigInteger> callable = new Callable<BigInteger>() {
        public BigInteger call() throws Exception {
            return recursiveFasterWithCache (m, cache); // 第一项和第二项任务共享同一份缓存
        }
    };
    Future<BigInteger> ffM = executorService.submit(callable);

    callable = new Callable<BigInteger>() {
        public BigInteger call() throws Exception {
            return recursiveFasterWithCache (m-1, cache); // 第一项和第二项任务共享同一份缓存
        }
    };
    Future<BigInteger> ffM_1 = executorService.submit(callable);

    // 得到各部分结果，进行合并
    BigInteger fM, fM_1, fN;

    try {
        fM = ffM.get(); //获取第一个子问题的结果（阻塞调用）
    } catch (Exception e) {
        // 如果有异常，在当前线程计算 fM
        fM = recursiveFasterBigInteger(m);
    }
    try {
        fM_1 = ffM_1.get();  // 获取第二个子问题的结果（阻塞调用）
    } catch (Exception e) {
        //如果有异常，在当前线程计算 fM
        fM_1 = recursiveFasterBigInteger(m-1);
    }

    if ((n & 1) != 0) {
        fN = fM.pow(2).add(fM_1.pow(2));
```

更多资源请访问稀酷客(www.ckook.com)

```
        } else {
            fN = fM_1.shiftLeft(1).add(fM).multiply(fM);
        }

        return fN;
    }
}
```

注意 recursiveFasterWithCache 的第二个参数是个 map，它可以是任何实现了 Map 接口的类的对象，例如 ConcurrentHashMap 或 HashMap 对象。SparseArray 对象不是 map。

即使把问题分解为子问题并将子问题分派到不同线程，可能性能也不一定有提升。有可能是数据之间有依赖，不得不进行同步，线程可能会花费一些或大部分时间在等待数据。此外，性能提升也不一定如你所愿。虽然相对于单核心，理论上期望双核心处理器性能是两倍，四核处理器是四倍，但现实给出的是另一番景象。

实践中，最好使用多线程执行无关的任务（避免了同步需求），或者执行只偶尔或频率低的定期同步。例如，视频游戏通常会使用一个游戏逻辑线程和一个渲染线程。渲染线程需要每秒 30 或 60 次（渲染每帧）读取逻辑线程操作的数据，并可以相对迅速地复制一份渲染一帧所需的数据，所以阻塞访问只是短短的一瞬。

5.7 Activity 生命周期

你创建的线程不会自动获知 Activity 生命周期中的变化。例如，调用 Activity 的 onStop()方法使得 Activity 不可见，或者调用 Activity 的 onDestroy()方法都不会自动给你创建的线程发通知。这意味着需要做额外的工作让自己的线程与应用的生命周期保持同步。代码清单 5-20 给出了 Activity 已被销毁后，AsyncTask 仍在运行的示例。

代码清单 5-20 在后台线程中计算斐波那契数，根据结果更新用户界面

```
public class MyActivity extends Activity {
    private TextView mResultTextView;
    private Button mRunButton;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // 布局包含 TextView 和 Button
        mResultTextView = (TextView) findViewById(R.id.resultTextView); // 显示结果处
        mRunButton = (Button) findViewById(R.id.runButton); // 开始计算的按钮
    }

    public void onClick (View v) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            new AsyncTask<Integer, Void, BigInteger>() {
                @Override
                protected void onPreExecute() {
                    // 禁用按钮，让用户一下只能启动一次计算
                    mRunButton.setEnabled(false);
                }

                @Override
                protected void onCancelled() {
                    // 按钮被再次启用，让用户启动下一次计算
                    mRunButton.setEnabled(true);
                }

                @Override
                protected BigInteger doInBackground(Integer... params) {
                    return Fibonacci.recursiveFasterPrimitiveAndBigInteger(params[0]);
                }

                @Override
                protected void onPostExecute(BigInteger result) {
                    mResultTextView.setText(result.toString());
                    // 按钮被再次启用，让用户启动下一次计算
                    mRunButton.setEnabled(true);
                }
            }.execute(100000); // 为了简单起见，这里的参数用了硬编码
        }
    }
```

这个例子做了两件简单的事情，当用户按下按钮时：

- 在单独的线程计算斐波那契数；
- 按钮在计算过程中被禁用，计算完成后才可以再次使用，让用户一下只能计算一次。

表面上看起来这样是正确的。但是如果用户在计算期间旋转了设备，Activity 会销毁并再次创建（我们这里假设 manifest 文件没有指定 Acitvity 自行处理方向变化。）。MyActivity 的当前实例通过 onPause()、onStop()、onDestroy()顺序调用，而新的实例通过 onCreate()、onStart()、onResume()顺序调用。当这一切发生时，AsyncTask 线程不会受到任务影响，不知道方向发生变化，一直运行到计算完成。目前看起来是这是正确的，也似乎是人们所期望的。

不过，意料之外的事情发生了：旋转屏幕后，按钮变成可用了。这很容易解释，因为方向变化后的按钮实际上是个新的按钮，它在 onCreate()中创建，默认情况下是启用的。因此，用户这时可以启动第二个计算，而第一个仍在进行中。虽然相对无害，但这不符合用户界面的定义，即计算正在进行时要禁用按钮。

5.7.1　传递信息

如果要修复这个错误，新的 Activity 实例要知道计算是否已经在进行，这样在 onCreate()创建新按钮时可以禁用它。代码清单 5-21 是修改版，可以把信息传给新 MyActivity 实例。

更多资源请访问稀酷客(www.ckook.com)

代码清单 5-21 把一个 Activity 实例的信息传给另一个

```
public class MyActivity extends Activity {
    private static final String TAG = "MyActivity";

    private TextView mResultTextView;
    private Button mRunButton;
    private AsyncTask<Integer, Void, BigInteger> mTask; // 会把对象传给其他实例

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // 在这里添加日志消息，获知现在创建的是哪个 MyActivity 实例
        Log.i(TAG, "MyActivity instance is " + MyActivity.this.toString());
        Log.i(TAG, "onCreate() called in thread " + Thread.currentThread().getId());

        // 布局包含 TextView 和 Button
        mResultTextView = (TextView) findViewById(R.id.resultTextView); // 这里显示结果
        mRunButton = (Button) findViewById(R.id.runButton); // 开始计算的按钮

        // 得到 onRetainNonConfigurationInstance()返回的对象
        mTask = (AsyncTask<Integer, Void, BigInteger>) getLastNonConfigurationInstance();
        if (mTask != null) {
            mRunButton.setEnabled(false); // 计算仍在进行中，禁用按钮
        }
    }

    @Override
    public Object onRetainNonConfigurationInstance() {
        return mTask; // 如果计算在进行会返回非 null 对象
    }

    public void onClick (View v) {
        // 保持 AsyncTask 对象的引用
        mTask = new AsyncTask<Integer, Void, BigInteger>() {
            @Override
            protected void onPreExecute() {
                // 按钮被禁用，让用户一下只能开始一次计算
                mRunButton.setEnabled(false);
            }

            @Override
            protected void onCancelled() {
                // 按钮被再次启用，让用户可以启动下一次计算
                mRunButton.setEnabled(true);
                mTask = null;
            }

            @Override
            protected BigInteger doInBackground(Integer... params) {
                return Fibonacci.recursiveFasterPrimitiveAndBigInteger(params[0]);
            }
```

更多资源请访问稀酷客(www.ckook.com)

```
            @Override
            protected void onPostExecute(BigInteger result) {
                mResultTextView.setText(result.toString());
                // 按钮被再次启用，让用户可以启动下一次计算
                mRunButton.setEnabled(true);
                mTask = null;

                // 添加日志消息，可以知道什么时候计算完成
                Log.i(TAG, "Computation completed in " + MyActivity.this.toString());
                Log.i(TAG, "onPostExecute () called in thread " + Thread.currentThread().getId());
            }
        }.execute(100000); // 简单起见，硬编码值
    }
}
```

注意　onRetainNonConfigurationInstance()在API等级11的Fragment API中或为旧平台提供的Android兼容包中被建议废弃。这里是为了简单起见，你会发现还有一些代码示例使用此方法。不过，你应该使用Fragment API编写新应用。

如果执行这段代码，你会发现，当旋转屏幕后，只要计算没有完成，按钮就仍然是禁用的。这似乎解决了在代码清单5-20中遇到的问题。不过你可能会注意到一个新问题：如果在计算过程中旋转设备，等到计算完成，按钮也不会被启用，尽管已经调了onPostExecute()方法。这是个大问题，按钮永远不能再次启用了！此外，计算的结果没有反馈到用户界面上。（这个问题代码清单5-20也有，所以你可能会在发现按钮问题之前注意到。）

这个很好解释（但也可能不那么显而易见，如果你是Java新手）：onPostExecute和onCreate（第一个Activity被销毁了，但主线程仍是相同的）在相同的线程中调用，onPostExecute用到的mResultTextView和mRunButton对象实际上属于第一个MyActivity实例，而不属于新的实例。在新AsyncTask对象创建时，声明的匿名内部类与它的外围类的实例发生关联（这就是为什么AsyncTask创建的对象可以直接引用MyActivity内部声明的字段，如mResultTextView和mTask），因此它不会访问新的MyActivity实例字段。基本上，代码清单5-21中的代码在计算过程中旋转设备时会碰到两个主要缺陷：

- 按钮永远不会再次启用，结果永远不会出现；
- 之前的Activity有泄露，因为mTask保持了外围类实例的引用（因此设备旋转后存在两个MyActivity实例）。

5.7.2　记住状态

解决这个问题的方法之一，是让新MyActivity实例知道正处在计算过程中，并能重新开始计算。前面的计算可以在onStop()或onDestroy()中使用AsyncTask.cancel()API取消。代码清单5-22是个参考实现。

更多资源请访问稀酷客(www.ckook.com)

代码清单 5-22 记住计算过程

```
public class MyActivity extends Activity {
    private static final String TAG = "MyActivity";

    private static final String STATE_COMPUTE = "myactivity.compute";

    private TextView mResultTextView;
    private Button mRunButton;
    private AsyncTask<Integer, Void, BigInteger> mTask;

    @Override
    protected void onStop() {
        super.onStop();
        if (mTask != null) {
            mTask.cancel(true); // 尽管现在取消了，线程也不会马上停止
        }
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        // 如果被调用，它会确保在 onStop()之前调用
        super.onSaveInstanceState(outState);
        if (mTask != null) {
            outState.putInt(STATE_COMPUTE, 100000); // 简单起见，把值硬编码
        }
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        // 在这里添加日志消息，知道现在创建的是哪个 MyActivity 实例
        Log.i(TAG, "MyActivity instance is " + MyActivity.this.toString());
        Log.i(TAG, "onCreate() called in thread " + Thread.currentThread().getId());

        // 布局包含 TextView 和 Button
        mResultTextView = (TextView) findViewById(R.id.resultTextView); // 这里显示结果
        mRunButton = (Button) findViewById(R.id.runButton); // 开始计算的按钮

        // 确保检查了 savedInstanceState 是否为空
        if (savedInstanceState != null && savedInstanceState.containsKey(STATE_COMPUTE)){
            int value = savedInstanceState.getInt(STATE_COMPUTE);
            mTask = createMyTask().execute(value); // 按钮会在 onPreExecute()中禁用
        }
    }

    // AsyncTask 的创建挪到私有方法，它现在可以从两个地方创建
    private AsyncTask<Integer, Void, BigInteger> createMyTask() {
        return new AsyncTask<Integer, Void, BigInteger>() {
            @Override
            protected void onPreExecute() {
                // 按钮被禁用，让用户一下只能开始一次计算
```

更多资源请访问稀酷客(www.ckook.com)

```
            mRunButton.setEnabled(false);
        }

        @Override
        protected void onCancelled() {
            // 按钮再次被启用，让用户启动下一次计算
            mRunButton.setEnabled(true);
            mTask = null;
        }

        @Override
        protected BigInteger doInBackground(Integer... params) {
            return Fibonacci.recursiveFasterPrimitiveAndBigInteger(params[0]);
        }

        @Override
        protected void onPostExecute(BigInteger result) {
            mResultTextView.setText(result.toString());
            // 按钮再次被启用，让用户启动下一次计算
            mRunButton.setEnabled(true);
            mTask = null;

            // 添加日志消息，记录计算的时间
            Log.i(TAG, "Computation completed in " + MyActivity.this.toString());
            Log.i(TAG, "onPostExecute () called in thread " + Thread.currentThread().getId());
        }
    };
}

public void onClick (View v) {
    // 保持 AsyncTask 对象的引用
    mTask = createMyTask.execute(100000); // 为简单起见，这里把值硬编码
}
}
```

通过这个实现，我们基本上可以告诉新实例，以前的实例在销毁时正在计算数值。新实例将重新开始计算，用户界面也进行相应的刷新。

设备不光能通过旋转造成配置变化，其他事件也能产生配置变化。比如，语言环境的变化或连接外部键盘。虽然谷歌电视设备可能不会旋转（至少现在是这样），但你还是应该针对谷歌电视设备考虑配置变化的情况，因为其他事件仍有可能发生配置变化。此外，将来可能会加入新事件，这些事件也会导致配置变化。

注意　onSaveInstanceState()并不会被调用。基本上当 Android 认为确实需要时才会调用它。请参阅 Android 的文档获取更多信息。

取消 AsyncTask 对象并不一定意味着线程会立即停止。实际行为取决于以下两件事情：

- 该任务是否已已经开始；
- 传递给 cancel()的参数（true 或 false）

更多资源请访问稀酷客(www.ckook.com)

调用 AsyncTask.cancel()会在 doInBackground()返回之后触发调用 onCancelled()，而不是 onPostExecute()。因为 doInBackground()可能在 onCancelled()之前完成，所以可以在 doInBackground()中定期调用 AsyncTask.isCancelled()来检查，以便尽早返回。说些与本例无关的题外话，这样会使代码有点难以维护，因为你不停地在 AsyncTask 的相关调用（isCancelled()）和代码的真实任务（这是应该和 AsyncTask 无关的）之间进行切换。

注意 Activity 被销毁时线程并不总会被打断。可以使用 Activity.isChangingConfiguration()和 Activity.isFinishing()API 去多了解发生了什么事，作出相应计划。例如，在代码清单 5-22 中，我们可以决定在 onStop()中只有当 isFinishing()返回 true 时才取消任务。

一般情况下，当 Activity 被暂停或停止时，至少要把它的后台线程暂停。这可以防止应用占用其他 Activity 亟需的资源（CPU、内存、内部存储）。

提示 参考 http://code.google.com/p/shelves 上的书架代码和 http://code.google.com/p/apps-for-android 的照片流代码，那有更多在 Activity 实例间保存状态信息的示例。

5.8 总结

使用多线程可以让代码效率更高，维护其实也不难，甚至适用于使用单线程的内部设备。然而，多线程也增加了应用的复杂性，尤其涉及同步以及为了良好的用户体验要保持应用状态时。你应该了解应用中使用多线程的后果，因为这很容易失控，让调试变得相当困难。虽然有时使用多线程不是件简单的事，但它无疑可以显著提高应用的性能。多核架构很快就会成为主流，在应用中添加多线程支持，绝对会让大多数用户受益。

更多资源请访问稀酷客(www.ckook.com)

第6章 性能评测和剖析

为了确定是否需要优化，优化后是否有提升，针对性能进行度量是非常必要的。

在大多数情况下，性能使用函数完成操作所花费的时间作为测量依据。例如，游戏的性能往往用每秒渲染多少帧进行度量，这个直接由渲染帧所需要的时间决定：为达到每秒 60 帧的恒定速率，每帧渲染和显示的时间应小于 16.67 毫秒示。另外，我们在第 1 章中讨论过，小于 100 毫秒的响应时间才会让人有瞬时的感觉。

本章会学习各种在应用中测量时间的方法。你还会学到如何使用剖析工具——Traceview，跟踪 Java 代码和本地代码，并轻松找出应用中的瓶颈。最后，你会了解到 Android 的日志机制及使用日志过滤机制的方法。

6.1 时间测量

操作或操作序列的完成时间是优化代码时所需的关键信息。不知道在某件事情上花了多少时间，是无法度量优化效果的。Java 和 Android 提供了以下 API，让应用可以测量时间及性能：

- `System.currentTimeMillis`
- `System.nanoTime`
- `Debug.threadCpuTimeNanos`
- `SystemClock.currentThreadTimeMillis`
- `SystemClock.elapsedRealtime`
- `SystemClock.uptimeMillis`

通常情况下，应用需要调用这些方法两次，只调用一次是没用的。要测量时间，应用需要获取起始时间和结束时间，用它们之间的差值来度量性能。这里要先说明一下，一秒钟有 1 000 000 000 纳秒，换句话说纳秒是一秒的十亿分之一。

注意 即使有些方法返回时间用纳秒表示，但这并不意味着精度是纳秒级的。实际精度取决于平台，设备之间可能会有所不同。同样，`System.currentTimeMillis()`返回的毫秒数也不保证毫秒的精度。

更多资源请访问稀酷客(www.ckook.com)

代码清单 6-1 给出了典型的用法。

代码清单 6-1 测量时间

```
long startTime = System.nanoTime();

// 这里是待测量的操作

long duration = System.nanoTime() - startTime;

System.out.println("Duration: " + duration);
```

重要的细节是，实际上代码清单 6-1 没有用任何 Android API，这份测量代码只用了 java.lang.System、java.lang.String 和 java.io.PrintStream 包。因此，类似的代码可以用在其他 Java 应用上，并不一定是 Android 程序。另一方面，Debug 和 SystemClock 类是 Android 独有的。

尽管 System.currentTimeMillis()可以作为测量时间的手段，但不建议使用这种方法，原因如下：

- 其精度和准确度可能不够；
- 更改系统时间会影响结果。

最好使用 System.nanoTime()，因为它提供了更好的精度和准确度。

6.1.1 System.nanoTime()

System.nanoTime()没有定义参考时间，它只能用来测量时间间隔，如代码清单 6-1 所示。获取时间(时钟)要用 System.currentTimeMillis(),它的返回值是 UTC 时间 1970 年 1 月 1 日 00:00:00 到现在的毫秒数。

代码清单 6-2 显示了如何用 System.nanoTime()测量完成时间。

代码清单 6-2 用 System.nanoTime()测量时间

```
private void measureNanoTime() {
    final int ITERATIONS = 100000;
    long total = 0;
    long min = Long.MAX_VALUE;
    long max = Long.MIN_VALUE;

    for (int i = 0; i < ITERATIONS; i++) {
        long startTime = System.nanoTime();
        long time = System.nanoTime() - startTime;
        total += time;
        if (time < min) {
            min = time;
        }
        if (time > max) {
            max = time;
        }
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
    Log.i(TAG, "Average time: " + ((float)total / ITERATIONS) + " nanoseconds");
    Log.i(TAG, " Minimum: " + min);
    Log.i(TAG, " Maximum: " + max);
}
```

三星 Galaxy Tab 10.1 上的平均时间大约是 750 纳秒。

注意　调用 System.nanoTime()消耗的时间取决于具体设备和实现。

因为调度器最终负责调度在 CPU 上运行的线程，需要测量的操作可能会被它中断几次，来让出 CPU 时间给别的线程。因此，测量结果可能包括一些执行其他代码的时间，这可能会得出不正确的时间测量结果，产生误导。

可以使用 Android 的 Debug.threadCpuTimeNanos()方法测量代码的执行时间，这个方法更好些。

6.1.2　Debug.threadCpuTimeNanos()

因为 Debug.threadCpuTimeNanos()只测量在当前线程中所花费的时间，所以它的结果更准确。不过，如果要测量的部分运行在多个线程上，只调用一次 Debug.threadCpuTimeNanos()不会给出准确的估值，必须在所有涉及的线程调用此方法并把结果相加。

代码清单 6-3 是 Debug.threadCpuTimeNanos()的使用示例。用法和 System.nanoTime()一样，只用于测量时间间隔。

代码清单 6-3　使用 Debug.threadCpuTimeNanos()

```
long startTime = Debug.threadCpuTimeNanos();
// 警告：这可能会返回-1，如果系统不支持此操作

// 暂停 1 秒钟（其他线程会在这段时间内调度运行）
try {
    TimeUnit.SECONDS.sleep(1);
    // 和 Thread.sleep(1000)是一样的;
} catch (InterruptedException e) {
    e.printStackTrace();
}

long duration = Debug.threadCpuTimeNanos() - startTime;

Log.i(TAG, "Duration: " + duration + " nanoseconds");
```

虽然因为调用 TimeUnit.SECONDS.sleep()，整段代码要花 1 秒钟，但实际上执行代码花费的时间要少得多。事实上，在三星 Galaxy Tab 10.1 上运行这段代码，执行时间只有大约 74 微秒。这是可以预料到的，因为两次调用 Debug.threadCpuTimeNanos()都没做什么事，只是让线程暂停了 1 秒钟。

更多资源请访问稀酷客(www.ckook.com)

注意 参阅 TimeUnit 类的文档，它提供了不同时间单位之间的转换方法及与线程相关的操作（如 Thread.join()和 Object.wait()）。

当然，你也可以在应用的C代码中使用“标准”C时间函数来测量，如代码清单6-4所示。

代码清单6-4 使用C时间函数

```
#include <time.h>

void foo() {
    double duration;
    time_t time = time(NULL);

    // 要测量的东西放在这里

    duration = difftime(time(NULL), time); // 持续秒数
}
```

6.2 方法调用跟踪

一旦确定在哪儿花费了过多时间，就需要了解更多细节，找出罪魁祸首的方法。可以利用跟踪工具创建方法跟踪文件；之后用 Traceview 工具进行分析。

6.2.1 Debug.startMethodTracing()

Android 提供了 Debug.startMethodTracing()方法来创建跟踪文件，然后用 Traceview 工具调试和分析应用。Debug.startMethodTracing()方法有 4 个变种：

- startMethodTracing()
- startMethodTracing(String traceName)
- startMethodTracing(String traceName, int bufferSize)
- startMethodTracing(String traceName, int bufferSize, int flags)

traceName 参数指定写入跟踪信息的文件名。（如果该文件已经存在，它会被截断。）你要确保应用有权限写这个文件（默认情况下，该文件会创建在 sdcard 目录下，否则要给出绝对路径。）。buffersize 参数指定跟踪文件的最大大小。跟踪信息会占用大量的空间，根据存储容量可能要限制它的大小，要尽量使用合理的值（默认是 8MB）。目前，Android 只定义了一个标志，Debug.TRACE_COUNT_ALLOCS，这个标志参数可以设置为 0 或 Debug.TRACE_COUNT_ALLOCS（添加 Debug.startAllocCounting()的结果跟踪内存分配的数量和总大小）。Android 还提供了 Debug.stopMethodTracing()方法，从名字也能猜到就是停止跟踪的方法。用法和之前的时间度量非常类似，如代码清单6-5所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单 6-5　启用跟踪

```
Debug.startMethodTracing("/sdcard/awesometrace.trace");

// 需要跟踪的操作
BigInteger fN = Fibonacci.computeRecursivelyWithCache(100000);

Debug.stopMethodTracing();

//现在在/mnt/sdcard 目录下会有名为 awesometrace.trace 的文件，在 Eclipse DDMS 中可以取到
```

跟踪文件保存在 Android 设备（或模拟器）中，所以需要复制文件到主机上，例如在 Eclipse 中使用 DDMS 或用“adb pull”命令。[①]

6.2.2　使用 Traceview 工具

Android SDK 中有一个 Traceview 的工具，它可以分析这些跟踪文件并给出图形化的结果展示，如图 6-1 所示。你可以在 SDK 的 tools 目录下找到 Traceview 工具，只需在命令行下输入 `traceview awesometrace.trace` 就可以启动 Traceview。

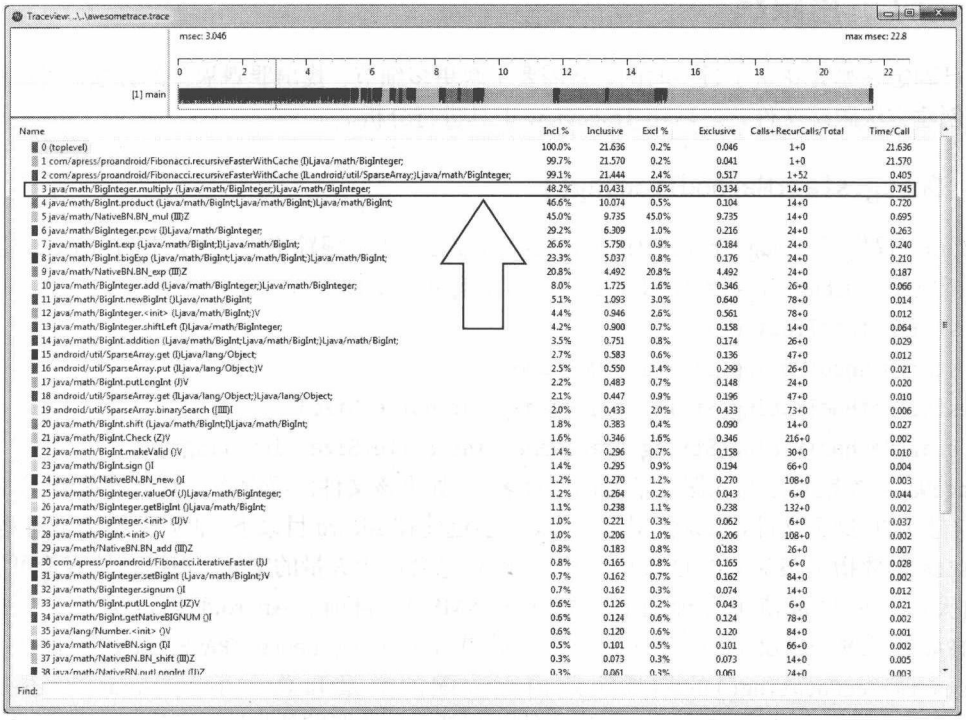


图 6-1　Traceview 窗口

① ddms 也可以单独运行。——译者注

更多资源请访问稀酷客(www.ckook.com)

Traceview 的信息包含了所有的函数调用，以及调用执行时间和调用次数，有以下 7 列内容。

- Name：方法名。
- Incl %：此方法中占的时间百分比（包含子方法）。
- Inclusive：此方法所花毫秒数（包含子方法）。
- Excl %：此方法所占时间百分比（不包含子方法）。
- Exclusive：此方法所花毫秒数（不包含子方法）。
- Calls+RecurCalls/Total：调用和递归调用次数 。
- Time/Call：平均每次调用时间。

例如，`BigInteger.multiply()`总共调用 14 次，共花 10.431 毫秒，平均每次调用花了 745 微秒。因为 VM 启用跟踪时会减缓运行速度，不要把这个时间值当作最终的结果。这些时间值只是为了确定哪个方法或运行方式速度更快。

如果点击一个方法的名称，Traceview 会告诉该方法的更详细的信息，如图 6-2 所示。这包括：

- Parents（此方法的调用者）;
- Children（此方法调用的方法）;
- Parents while recursive（递归时此方法的调用者）;
- Children while recursive（递归时此方法调用的方法）。

从图 6-2 中可以看到，大部分时间都用在了以下 4 个方法上：

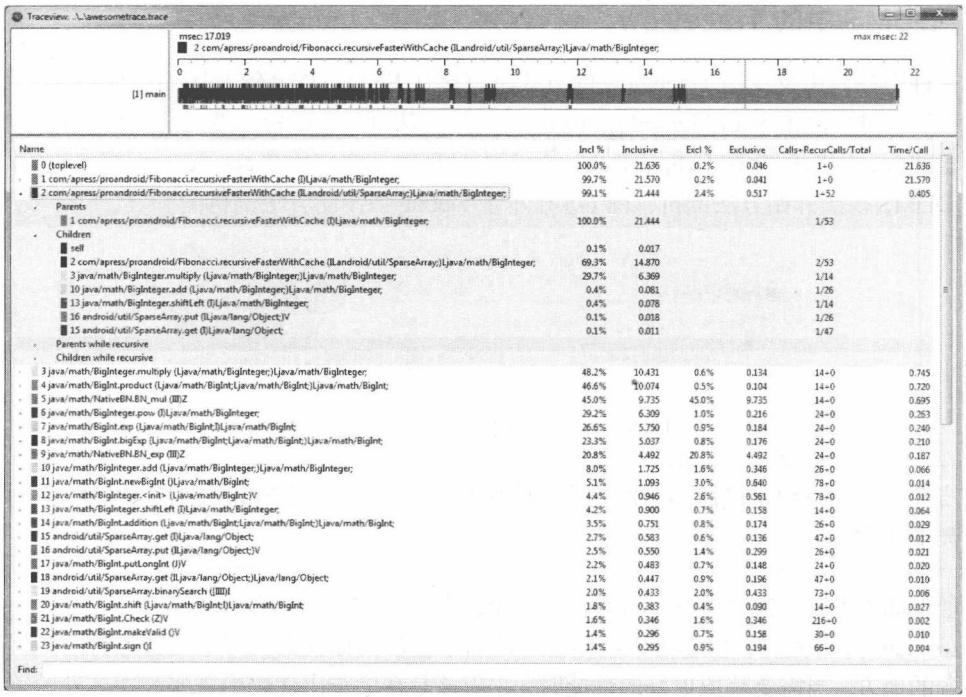


图 6-2 方法的详细信息

更多资源请访问稀酷客(www.ckook.com)

- BigInteger.multiply()
- BigInteger.pow()
- BigInteger.add()
- BigInteger.shiftLeft()

尽管我们在在第 1 章中已经确定了瓶颈所在，Traceview 也可以让你很快确定哪些地方不需要进一步花精力研究。在这个例子中，可以很快看出，大部分时间都花在 BigInteger.multiply() 上，BigInteger.pow() 紧随其后。这并不奇怪，乘法本来就比 BigInteger.add() 加法和 BigInteger.shiftLeft() 移位复杂。

在窗口的顶部，可以看到主线程的时间线。选定时间线的某个区域，双击时间尺度可以放大和缩小。熟悉 Traceview 工具，看看如何从一个方法转到另一个。

提示　这很容易，点击方法的名字即可！

因为启用跟踪时 JIT 编译器是禁用的，得到的结果可能有一定误导性。事实上，你可能也会想到：方法可以被 Dalvik 的 JIT 编译器编译为机器码，它在真实场景中执行需要花费的时间会更少。此外，跟踪不会显示本地函数耗费的时间。例如，图 6-1 显示出调用了 NativeBN.BN_mul() 和 NativeBN.BN_exp()，但如果你点击这些方法，却看不见调用到的其他方法。

6.2.3　DDMS 中的 Traceview

另一种跟踪调用及使用 Traceview 的方式是直接从 Eclipse DDMS 视图中生成跟踪文件。选定一个进程后，可以点击 Start Method Profiling（开始方法剖析）图标，然后再次单击就停止剖析。一旦停止剖析，跟踪就会在 Eclipse Debug 视图中呈现，就像使用 Traceview 一样。图 6-3 显示了如何从 DDMS 视图开始方法剖析，图 6-4 显示了 Debug 视图的方法剖析图。

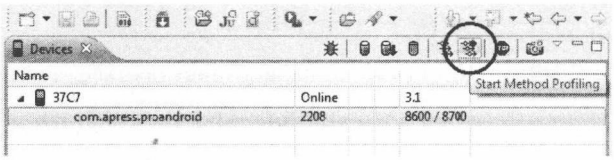


图 6-3　在 DDMS 视图中开始方法剖析

提示　当结束调试和剖析时，记住要删除跟踪文件。可以使用 Eclipse DDMS 视图从设备中删除文件。

正如在图 6-4 中看到的那样，可以显示多个线程的时间线。

Traceview 并不完美，但它可以给你很好的启示，帮助你找出实际中真正执行到的代码和可能是瓶颈的地方。当需要获取更好的性能时，它应该是帮助查出代码中需要改善之处的首选工具之一。

更多资源请访问稀酷客(www.ckook.com)

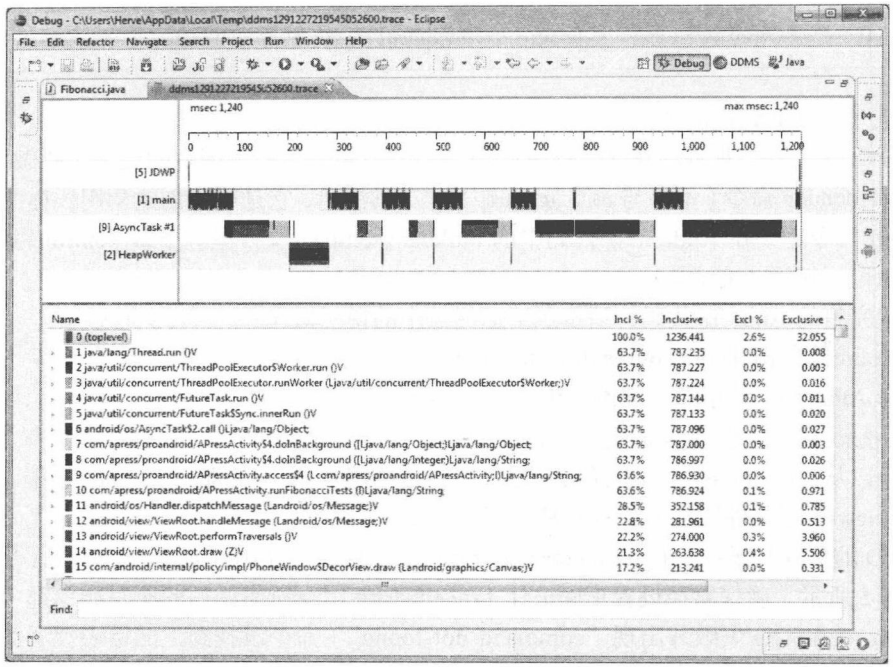


图 6-4 在 Debug 视图中的方法剖析

6.2.4 本地方法跟踪

除了用 `startMethodTracing()`API 剖析 Java 方法，Android 还支持本地方法跟踪（包括内核）。本地方法利用 QEMU 进行跟踪。这一节中会学习如何生成 QEMU 的跟踪文件，如何将其转换成 Traceview 可以处理的文件等。

做以下两件事生成 QEMU 的跟踪信息：

- 利用-trace 选项启动模拟器（例如，“emulator -trace mytrace -avd myavd”）；
- 启动/停止本地跟踪，可以通过调用 `Debug.startNativeTracing()`和 `Debug.stopNativeTracing()`，也可以按 F9 键（第一次开始跟踪，第二次停止跟踪）。

在主机上 AVD 的跟踪目录下，有个 mytrace 目录，其中有以下几个 QEMU 模拟器跟踪文件：

- qtrace.bb
- qtrace.exc
- qtrace.insn
- qtrace.method
- qtrace.pid
- qtrace.static

更多资源请访问稀酷客(www.ckook.com)

注意　QEMU 是开源的模拟器，参阅http://wiki.qemu.org获取更多信息。

1. 生成 Traceview 用的跟踪文件

要像分析 Java 方法那样使用 Traceview，需生成 Traceview 可以理解的文件。要做到这一点，要使用 `tracedmdump` 命令（不要与 `dmtracedump` SDK 工具混淆，它是用来创建调用栈树图的工具）。`tracedmdump` 命令定义在 Android 源代码中的 build/envsetup.sh 文件里。下载 Android 源代码才能使用这个命令，编译 Android。

要下载完整的 Android 代码，请按照下面网页中的说明操作：

http://source.android.com/source/downloading.html。

要编译 Android，请按照下面网页中的说明操作：

http://source.android.com/source/building.html

你也可以从 Android 源代码编译仿真器，而不是使用 SDK 中的那个。Android 编译完成后，创建 Traceview 跟踪文件用到的工具就都有了。

在 AVD 的跟踪目录中运行 `tracedmdump mytrace`，这会创建用 Traceview 可以打开的跟踪文件，如图 6-5 所示。确保路径设置正确，让 `tracedmdump` 运行所需调用的所有命令都可以执行成功。如果 `tracedmdump` 失败并出现“command not found”（命令没找到）的错误提示消息，很可能是因为路径设置不正确。例如，`tracedmdump` 会调用 `post_trace`，它位于 out/host/linux-x86/bin 目录。

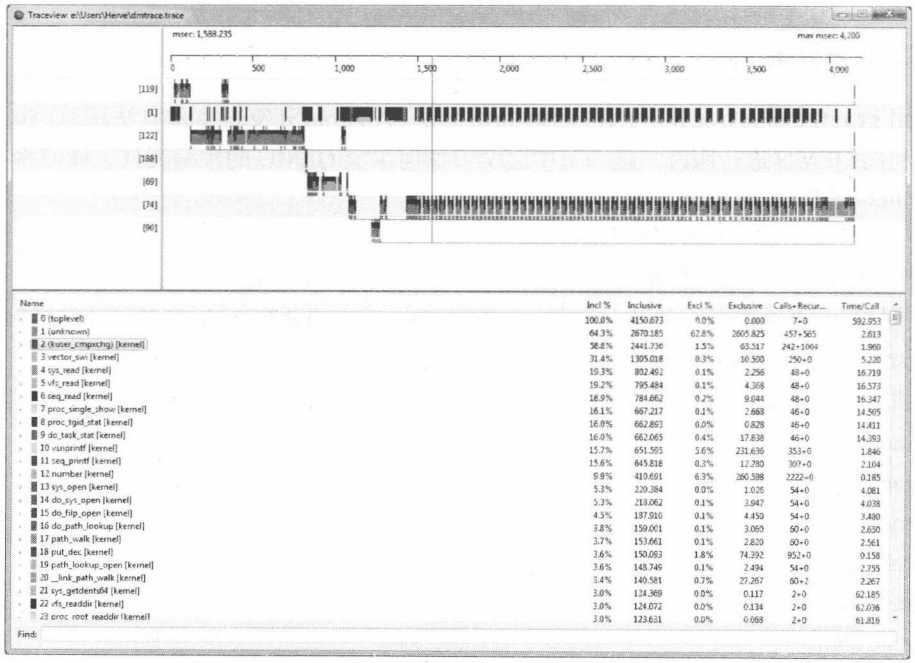


图 6-5　本地跟踪与 Traceview

更多资源请访问稀酷客(www.ckook.com)

虽然用户界面相同，但图 6-5 显示的是本地函数列表，如 vsnprintf()和 sys_open()，分别为 # 10 和 # 13。

tracedmdump 创建了以下两个文件表示相同的数据：

- dmtrace
- dmtrace.html

第一个文件供 Traceview 使用，而第二个可以在任何网页浏览器打开，包括 Lynx。[①]

注意 许多用户使用 tracedump 碰到问题时，错误消息并不总是描述得很清楚。如果你遇到一个错误，请在互联网上搜索错误描述和场景，很可能有人碰到了同样的问题，并公布了解决办法。

有时候，如果应用能够实时产生一份可读的情况描述会对工作大有帮助。日志消息在复杂的调试工具发明之前就已经有很长的历史了，许多开发者非常喜欢用日志调试或剖析应用。

6.3 日志

我们可以使用 Log 类打印消息到 LogCat。除了 Java 传统的日志机制，如 System.out.println()，Android 还定义了 6 个日志等级，每个都有自己对应的方法：

- verbose (Log.v)
- debug (Log.d)
- info (Log.i)
- warning (Log.w)
- error (Log.e)
- assert (Log.wtf)

例如，调用 Log.v(TAG,"my message") 相当于调用 Log.println(Log.VERBOSE, TAG, "my message")。

注意 Log.wtf()方法在 API 等级 8 中引入，但 Log.ASSERT 从 API 等级 1 就有了。如果你想用 ASSERT 日志级别，要保证与旧 Android 设备的兼容性，使用 Log.println(Log.ASSERT,...)，而非 Log.wtf(...)。

你也可以在 Eclipse 中使用 LogCat（Window → Show View → LogCat）或（同时）在终端（adb logcat，或者在 adb shell 中输入 logcat）[②]，查看应用运行时产生的消息。

---

① 纯文本浏览器。——译者注

② 在较新的 SDK 中终端可以重定向到 logcat 到文件 adb logcat -vtime > log.txt）产生的消息。——译者注

更多资源请访问稀酷客(www.ckook.com)

由于会显示许多消息，而且很多都不是自己应用的，就需要创建过滤器，只显示与自己应用相关的输出。可以根据标签、优先级和PID来过滤消息。在Eclipse中，可以使用Create Filter（创建过滤器）功能，如图6-6所示。

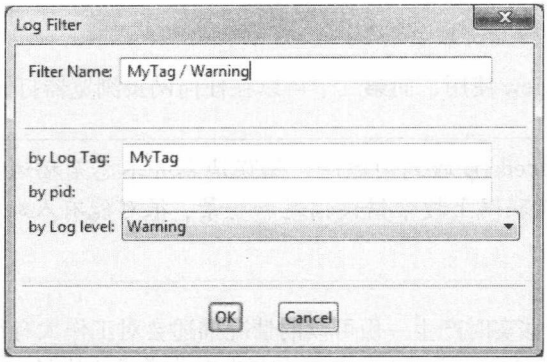


图6-6　用Eclipse创建LogCat过滤器

Eclipse目前不支持创建多个标签的过滤器，如果想这样做，就只能用adb logcat了。例如，要显示标签“MyTag”并且优先级在“Debug”以上（就是Debug、Info、Warning、Error和Assert）日志消息，同时也要显示标签“MyOtherTag”且优先级在“Warning”以上，可以键入：

```
adb logcat MyTag:D MyOtherTag:W *:S
```

不要忘记*:S部分，它表示滤掉所有其他的日志消息（S就代表Silent）。

日志函数在NDK中也可以使用，可以用LogCat来记录C/C++代码的消息。在NDK的android/log.h定义的函数如下：

- __android_log_write
- __android_log_print
- __android_log_vprint
- __android_log_assert

例如，和Log.i("MyTag", "Hello")对应的方法是__android_log_write(ANDROID_LOG_INFO, "MyTag", "Hello")。

这些都是Android内置的工具函数，可能会让代码有点罗嗦，建议为这些函数做个包装。事实上，这正是Android的源代码中cutils/log.h文件的宏LOGI和LOGE所做的，它们分别相当于Log.i和Log.e。你要认真设计包装函数，让它们也可以用在非Android系统上。[①]

6.4　总结

测量性能的方式很简单，但它是优化过程中关键的部分，Android提供了简易而强大的工具

① 可以参照Android系统代码那样处理上层，在底层输出时替换成专门针对目标设备的代码。——译者注

更多资源请访问稀酷客(www.ckook.com)

进行辅助。无论是跟踪 Java 还是本地方法，Traceview 都是最有用的工具之一，但请记住，只有真正的设备上的实际测量才能给出准确的答案，因为 Traceview 会禁用 Dalvik 的 JIT 编译器。虽然这些主题相当琐碎，但它们非常重要，Android 提供的各种工具你应该了然于胸，信手拈来。你还要记得经常检查新工具、新版本 SDK，看有没有提升剖析、测量和调试功能的新特性。记住先找瓶颈，再优化代码，做到物尽其用。

更多资源请访问稀酷客(www.ckook.com)

第7章 延长电池续航时间

电池虽小，地位却非常重要。Android 移动设备使用电池，应用做任何事情都要用电。大多数设备都是夜里在家充电，而白天很少有机会给电池充电，因此很多消费者期望电池最少能坚持 12 个小时。一些典型使用方式耗电很快：例如，在 Google I/O 大会上有充电平台，因为这些设备需要持续使用的时间比平时要长。

尽管应用有时看起来没做多少事，但实际上可能会很耗电，运行不了多久就会把电量榨干，使设备开机不到半天就没电了。那些被归为“电池杀手”的应用，最终的宿命就是被删除 →得差评 →不挣钱。因此作为开发者要尽量少用电量，合理使用电池。

本章会探讨如何测量电池的使用量，以及既可以省电，又不影响用户体验的方法，同时利用网络、定位和传感器这些吸引人特性。此外还会介绍如何高效使用更多 Android 内部组件如广播接收器、警报、唤醒锁。

7.1 电池

不同的设备电池容量也不同。通常，手机和平板电脑的电池容量的单位是 mAh（毫安小时）。表 7-1 是在第 2 章中提到的设备的电池容量。

注意 安培，以法国化学家安德烈·马里·安培（André-Marie Ampère）的名字命名的电流国际单位，常简写为“amp.”。1 安培小时等于 3600 库仑、1 安培秒等于 1 库仑、1 毫安小时等于 3.6 库仑。库仑，是以查尔斯·奥古斯丁·库仑（Charles-Augustin de Coulomb）的名字命名的国际单位，很少在消费产品中用到。

表7-1 一些Android设备的电池容量

设 备	制 造 商	电池容量
Blade	中兴	1250 mAh
Lephone	联想	1500 mAh
Nexus S	三星	1500 mAh

更多资源请访问稀酷客(www.ckook.com)

（续）

设 备	制 造 商	电池容量
Xoom	摩托罗拉	6500 mAh
Galaxy Tab（7"）	三星	4000 mAh
Galaxy Tab 10.1	三星	7000 mAh
Revue（set-top box）	罗技	n/a（无电池）
NSZ-GT1（Blu-ray player）	索尼	n/a（无电池）

事实上，平板电脑使用的电池容易大明显说明屏幕非常耗电。Android 提供了方法，可以让用户知道应用和系统组件耗费电量的近似值。图 7-1 显示了三星 Galaxy Tab 10.1 弹射愤怒小鸟打猪头时耗费了多少电量。

图 7-1 电池使用

在这个屏幕截图中两个项目遥遥领先：Screen 和 Wi-Fi。这两个组件很耗电，设备提供了些选项让用户对它们的使用方式进行配置。例如，用户可以调节屏幕的亮度（手动或根据所显示的图像自动[①]），设定多长时间没有活动后屏幕关闭，屏幕关闭时 Wi-Fi 也关闭。例如，Wi-Fi 连接如果在屏幕关闭后立即断开，那可能只占电池总使用量的百分之几。

注意 这里用的三星 Galaxy Tab 10.1 是只有 Wi-Fi 的版本。显示的其他项与设备相关，例如“手机待机“或”语音通话”。

① 应该还有根据环境光线感应。——译者注

更多资源请访问稀酷客(www.ckook.com)

虽然用户可以自己主动管理电池的使用，但还是有着自身的限制。最终，设备使用多少电量严重依赖所有应用都做了什么，也就是取决于应用是如何设计和实现的。

应用通常会实现以下功能：

- 执行代码（显而易见）；
- 数据传输（上传和下载，使用 Wi-Fi、EDGE、3G，4G）；
- 追踪位置（通过网络或 GPS）；
- 使用传感器（加速度计、陀螺仪）；
- 渲染图像（使用 GPU 或 CPU）；
- 唤醒以执行各种任务。

在学习如何最大限度地减少耗电量之前，我们先要有方法来衡量应用耗用了多少电量。

测量电池用量

不幸的是，开发者无法使用精确测量要用的大部分电气设备。幸好，Android 提供了可以获得电池使用信息的 API。虽然没有 getBatteryInfo()这样的 API，但获取电池信息可以通过检索固定的 Intent，也就是一直会广播出来的 Intent，如代码清单 7-1 所示。

代码清单 7-1　显示电池信息的 Activity

```
import static android.os.BatteryManager.*;
// 注意这里的 static 关键字（不知道它干什么？删掉看看）

public class BatteryInfoActivity extends Activity {
    private static final String TAG = "BatteryInfo";

    private BroadcastReceiver mBatteryChangedReceiver;
    private TextView mTextView; // 布局容器 TextView，用来显示电池信息

    private static String healthCodeToString(int health) {
        switch (health) {
        //case BATTERY_HEALTH_COLD: return "Cold"; // API 等级 11 才有
        case BATTERY_HEALTH_DEAD: return "Dead";
        case BATTERY_HEALTH_GOOD: return "Good";
        case BATTERY_HEALTH_OVERHEAT: return "Overheat";
        case BATTERY_HEALTH_OVER_VOLTAGE: return "Over voltage";
        case BATTERY_HEALTH_UNSPECIFIED_FAILURE: return "Unspecified failure";
        case BATTERY_HEALTH_UNKNOWN:
        default: return "Unknown";
        }
    }

    private static String pluggedCodeToString(int plugged) {
        switch (plugged) {
        case 0: return "Battery";
        case BATTERY_PLUGGED_AC: return "AC";
        case BATTERY_PLUGGED_USB: return "USB";
        default: return "Unknown";
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
}

private static String statusCodeToString(int status) {
    switch (status) {
    case BATTERY_STATUS_CHARGING: return "Charging";
    case BATTERY_STATUS_DISCHARGING: return "Discharging";
    case BATTERY_STATUS_FULL: return "Full";
    case BATTERY_STATUS_NOT_CHARGING: return "Not charging";
    case BATTERY_STATUS_UNKNOWN:
    default: return "Unknown";
    }
}
private void showBatteryInfo(Intent intent) {
    if (intent != null) {
        int health = intent.getIntExtra(EXTRA_HEALTH, BATTERY_HEALTH_UNKNOWN);
        String healthString = "Health: " + healthCodeToString(health);
        Log.i(TAG, healthString);

        int level = intent.getIntExtra(EXTRA_LEVEL, 0);
        int scale = intent.getIntExtra(EXTRA_SCALE, 100);
        float percentage = (scale != 0) ? (100.f * (level / (float)scale)) : 0.0f;
        String levelString = String.format("Level: %d/%d (%.2f%%)", level, scale, percentage);
        Log.i(TAG, levelString);

        int plugged = intent.getIntExtra(EXTRA_PLUGGED, 0);
        String pluggedString = "Power source: " + pluggedCodeToString(plugged);
        Log.i(TAG, pluggedString);

        boolean present = intent.getBooleanExtra(EXTRA_PRESENT, false);
        String presentString = "Present? " + (present ? "Yes" : "No");
        Log.i(TAG, presentString);

        int status = intent.getIntExtra(EXTRA_STATUS, BATTERY_STATUS_UNKNOWN);
        String statusString = "Status: " + statusCodeToString(status);
        Log.i(TAG, statusString);

        String technology = intent.getStringExtra(EXTRA_TECHNOLOGY);
        String technologyString = "Technology: " + technology;
        Log.i(TAG, technologyString);

        int temperature = intent.getIntExtra(EXTRA_STATUS, Integer.MIN_VALUE);
        String temperatureString = "Temperature: " + temperature;
        Log.i(TAG, temperatureString);

        int voltage = intent.getIntExtra(EXTRA_VOLTAGE, Integer.MIN_VALUE);
        String voltageString = "Voltage: " + voltage;
        Log.i(TAG, voltageString);

        String s = healthString + "\n";
        s += levelString + "\n";
        s += pluggedString + "\n";
        s += presentString + "\n";
        s += statusString + "\n";
```

更多资源请访问稀酷客(www.ckook.com)

```
            s += technologyString + "\n";
            s += temperatureString + "\n";
            s += voltageString;
            mTextView.setText(s);

            // 注意，StringBuilder 对象会更高效

            int id = intent.getIntExtra(EXTRA_ICON_SMALL, 0);
            setFeatureDrawableResource(Window.FEATURE_LEFT_ICON, id);
        } else {
            String s = "No battery information";
            Log.i(TAG, s);
            mTextView.setText(s);

            setFeatureDrawable(Window.FEATURE_LEFT_ICON, null);
        }
    }

    private void showBatteryInfo() {
        // 不需要接收器
        Intent intent = registerReceiver(null, new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
        showBatteryInfo(intent);
    }

    private void createBatteryReceiver() {
        mBatteryChangedReceiver = new BroadcastReceiver() {

            @Override
            public void onReceive(Context context, Intent intent) {
                showBatteryInfo(intent);
            }
        };
    }

    /** 当 Activity 第一次创建时调用.*/
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        requestWindowFeature(Window.FEATURE_LEFT_ICON);
        setContentView(R.layout.main);

        mTextView = (TextView) findViewById(R.id.battery);

        showBatteryInfo(); // 不需要接收器
    }

    @Override
    protected void onPause() {
        super.onPause();

        // 如果应用没在前台运行，注销接收器以节省功耗
        unregisterReceiver(mBatteryChangedReceiver);
    }
```

更多资源请访问稀酷客(www.ckook.com)

```
    @Override
    protected void onResume() {
        super.onResume();
        if (mBatteryChangedReceiver == null) {
            createBatteryReceiver();
        }
        registerReceiver(mBatteryChangedReceiver,
            new IntentFilter(Intent.ACTION_BATTERY_CHANGED));
    }

    @Override
    public void onLowMemory() {
        super.onLowMemory();
        unregisterReceiver(mBatteryChangedReceiver);
        mBatteryChangedReceiver = null;
    }
}
```

通过这段代码可以知道，电池信息是 Intent 的额外信息的一部分。这个 Activity 要得到改变的通知，因此在 onResume()注册了广播接收器。但是，该通知的唯一目的是用新的电池信息更新用户界面，Acitvity 只需要在前台和用户打交道时得到通知，所以我们在 onPause()中注销了广播接收器。

注意 *另一种实现方式是，将注册放到 onStart()，注销放到 onStop()中。为了节省更多电量，通常在 onResume()和 onPause()中进行注册与注销。*

如果需要知道当前电池信息，但不想用通知更新的方式，可以调用 registerReceiver()并把接收器参数设为 null，无需注册任何广播接收器，就能得到包含电池信息的 Intent。

测量电池用量，建议在应用启动时获取电池当前电量，运行一段时间，在退出时再次获取电池电量。两电量之间的差异并不表示应用真实的耗电量，因为有其他应用同时运行，所以需要有更好的方法测量单个应用的耗电量。比如，计算出在电池电量耗光之前，应用还可以运行多久。

7.2 禁用广播接收器

为了节省电量，应用应避免执行做无用功的代码。在上述例子中，在用户界面不在前台时更新 TextView 的文本毫无意义，只会浪费电量。

除了前面看到的 ACTION_BATTERY_CHANGED Intent 包含电池信息，Android 还定义了应用可以使用的 4 个广播 Intent：

- ACTION_BATTERY_LOW
- ACTION_BATTERY_OKAY
- ACTION_POWER_CONNECTED
- ACTION_POWER_DISCONNECTED

更多资源请访问稀酷客(www.ckook.com)

虽然不可以在应用的 manifest 文件中声明接收器去接收 ACTION_BATTERY_CHANGED 广播的 Intent（接收器只能显式地通过调用 registerReceiver()注册），其他的 Intent 是可以直接在 manifest 文件中注册的，如代码清单 7-2 所示。

代码清单 7-2 在 manifest 文件中声明广播接收器

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="com.apress.proandroid.ch07" android:versionCode="1" android:versionName="1.0">
    <uses-sdk android:minSdkVersion="8" />

    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".BatteryInfoActivity"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <receiver android:name=".BatteryReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BATTERY_LOW" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.BATTERY_OKAY" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.ACTION_POWER_CONNECTED" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.ACTION_POWER_DISCONNECTED" />
            </intent-filter>
        </receiver>

    </application>
</manifest>
```

广播接收器的一个实现如代码清单 7-3 所示。在这里，我们定义了单独的 BatteryReceiver 广播接收器，由它处理所有 4 个动作。

代码清单 7-3 BatteryReceiver 实现

```
public class BatteryReceiver extends BroadcastReceiver {
    private static final String TAG = "BatteryReceiver";

    @Override
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        String text;

        // 4个动作都在此处理

        if (Intent.ACTION_BATTERY_LOW.equals(action)) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            text = "Low power";
        } else if (Intent.ACTION_BATTERY_OKAY.equals(action)) {
            text = "Power okay (not low anymore)";
        } else if (Intent.ACTION_POWER_CONNECTED.equals(action)) {
            text = "Power now connected";
        } else if (Intent.ACTION_POWER_DISCONNECTED.equals(action)) {
            text = "Power now disconnected";
        } else {
            return;
        }

        Log.i(TAG, text);
        Toast.makeText(context, text, Toast.LENGTH_SHORT).show();
    }
}
```

你可能已经看出，这个应用有个严重缺陷。事实上，这 4 个动作当中只要有一个发生，应用就会启动（如果它尚未启动）。虽然这可能是期望的行为，但在许多情况下，你可能希望应用有不同的处理。例如这种情况，应用在前台运行时显示 Toast 消息是有意义的；如果在后台时还出现 Toast 消息，就会干扰其他应用，损害用户体验。

当应用不在运行或在后台运行时，我们想禁用这些 Toast 消息。这可以采用以下两种方法。

- 在应用中加入标记变量，在 Activity 的 onResume()中置为 true、在 onPause()中置为 false，并修改接收器的 onReceive()方法，检查标记的值。
- 只有当应用在前台运行时才可以启用广播接收器。

第一种方法虽然可行，但只要 4 种动作之一被触发，应用就会启动。最终会导致执行了不必要的指令，相当于执行 no-op（无操作）指令，白白耗费电量。此外，如果应用包含多个 Activity，修改标记会影响到多个文件。

第二种方法就好多了，因为我们可以确保只有在真正需要时才执行指令，就不会浪费电池电量。为了达到这一目标，我们需要在应用中做两件事情：

- 广播接收器默认是禁用的；
- 广播接收器必须在 onResume()中启用，在 onPause()被禁用。

禁用和启用广播接收器

代码清单 7-4 演示了如何在应用的代码清单文件中禁用广播接收器。

代码清单 7-4 在 manifest 文件中禁用广播接收器

```
…
 <receiver android:name=".BatteryReceiver" android:enabled="false" >
 …
```

注意 <application>元素有其自己的 enabled 属性。广播接收器会在<application>属性和<receiver>属性设置为 true 时被启用，在其中之一设置为 false 时被禁用。

更多资源请访问稀酷客(www.ckook.com)

代码清单 7-5 演示了如何在 onResume()和 onPause()中启用和禁用广播接收器。

代码清单 7-5　启用和禁用广播接收器

```
public class BatteryInfoActivity extends Activity {

    …

    private void enableBatteryReceiver(boolean enabled) {
        PackageManager pm = getPackageManager();
        ComponentName receiverName = new ComponentName(this, BatteryReceiver.class);
        int newState;

        if (enabled) {
            newState = PackageManager.COMPONENT_ENABLED_STATE_ENABLED;
        } else {
            newState = PackageManager.COMPONENT_ENABLED_STATE_DISABLED;
        }

        pm.setComponentEnabledSetting(receiverName, newState, PackageManager.DONT_KILL_APP);
    }

    …

    @Override
    protected void onPause() {
        super.onPause();
        unregisterReceiver(mBatteryChangedReceiver);

        enableBatteryReceiver(false); // 电池接收器现在被禁用

        // 应用没有在前台时注销接收器以省功耗
    }

    @Override
    protected void onResume() {
        super.onResume();
        if (mBatteryChangedReceiver == null) {
            createBatteryReceiver();
        }
        registerReceiver(mBatteryChangedReceiver, new
            IntentFilter(Intent.ACTION_BATTERY_CHANGED));

        enableBatteryReceiver(true);  // 电池接收器现在已启用
    }

    …
}
```

在只有真正需要时启用广播接收器，否则一直开着会浪费很多电。开发应用时这很容易被忽视，要特别注意，只在需要时启用接收器。

更多资源请访问稀酷客(www.ckook.com)

7.3 网络

许多 Android 应用在设备和服务器之间或在设备之间传输数据。像获取电池状态一样，应用需要获取设备上的网络连接信息。ConnectivityManager 类提供了 API，供应用调用以访问网络信息。Android 设备通常有多个数据连接：

- Bluetooth
- Ethernet
- Wi-Fi
- WiMAX
- 移动网络（EDGE、UMTS、LTE）

代码清单 7-6 演示了如何获得正在活动的连接以及所有连接信息。

代码清单 7-6 网络信息

```
private void showNetworkInfoToast() {
    ConnectivityManager cm = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);

    // 只显示活动连接
    NetworkInfo info = cm.getActiveNetworkInfo();
    if (info != null) {
        Toast.makeText(this, "Active: " + info.toString(), Toast.LENGTH_LONG).show();
    }

    // 显示所有连接
    NetworkInfo[] array = cm.getAllNetworkInfo();
    if (array != null) {
        String s = "All: ";
        for (NetworkInfo i: array) {
            s += i.toString() + "\n";
        }
        Toast.makeText(this, s, Toast.LENGTH_LONG).show();
    }
}
```

注意 应用需要设置 ACCESS_NETWORK_STATE 权限才能获取网络信息。

为了最大限度地延长电池的使用时间，我们需要知道如下事情：

- 后台数据设置；
- 数据传输频度。

7.3.1 后台数据

用户能在设置中指定是否允许后台数据传输，想必也是为了延长电池寿命。如果应用不在前

更多资源请访问稀酷客(www.ckook.com)

台也需要执行数据传输时，应该要检查该标志，如代码清单 7-7 所示。服务通常在启动数据传输时检查该设置。

代码清单 7-7　检查后台数据设置

```
private void transferData(byte[] array) {
    ConnectivityManager cm = (ConnectivityManager)
        getSystemService(Context.CONNECTIVITY_SERVICE);
    boolean backgroundDataSetting = cm.getBackgroundDataSetting();
    if (backgroundDataSetting) {
        // 传输数据
    } else {
        // 遵从设置，不传数据
    }
}
```

因为这是自愿的检查，应用实际上可以不管它，依然传输数据。但这违背了用户意愿，减慢前台的数据传输，并影响电池寿命，这种行为可能会导致你的应用最终被用户卸载。

想要在后台数据设置发生变化时得到通知，应用可以注册接收器，可以在 Java 代码中显式使用 ConnectivityManager.ACTION_BACKGROUND_DATA_SETTING_CHANGED 字符串建立 Intent 过滤器，或在应用的 manifest 文件中使用 android.net.conn.BACKGROUND_DATA_SETTING_CHANGED。此设置是为了控制后台数据传输，但我们还是推荐在 onResume()中禁用、在 onPause()中启用广播接收器。

注意　在 Android 4.0 中，getBackgroundDataSetting()方法已过时，会始终返回 true；行为也发生变化，不允许后台数据传输时，网络会断开。

7.3.2　数据传输

传输速率的差异非常大，从小于每秒 100Kb 的 GPRS 数据连接到每秒几 Mb 的 LTE 或 Wi-Fi 连接都有。除了连接类型，NetworkInfo 类还指定了连接的子类型，当连接类型是 TYPE_MOBILE 时就需要用到它。Android 定义了如下连接子类型（在 TelephonyManager 类中）：

- NETWORK_TYPE_GPRS （API 等级 1）
- NETWORK_TYPE_EDGE （API 等级 1）
- NETWORK_TYPE_UMTS （API 等级 1）
- NETWORK_TYPE_CDMA （API 等级 4）
- NETWORK_TYPE_EVDO_0 （API 等级 4）
- NETWORK_TYPE_EVDO_A （API 等级 4）
- NETWORK_TYPE_1xRTT （API 等级 4）
- NETWORK_TYPE_HSDPA （API 等级 5）
- NETWORK_TYPE_HSUPA （API 等级 5）

更多资源请访问稀酷客(www.ckook.com)

- NETWORK_TYPE_HSPA（API 等级 5）
- NETWORK_TYPE_IDEN （API 等级 8）
- NETWORK_TYPE_EVDO_B （API 等级 9）
- NETWORK_TYPE_LTE （API 等级 11）
- NETWORK_TYPE_EHRPD （API 等级 11）
- NETWORK_TYPE_HSPAP （API 等级 13）

如果创建和部署了新技术，也会增加新的子类型。例如，API 等级 11 加入了 LTE，API 等级 13 加入了 HSPAP。如果代码依赖这些值，确保应用处理了增加未知新值的情况，否则可能会导致应用无法传输数据。增加新的子类型时，你应该更新代码，所以要留意每个 Android SDK 的新版本发布。API 等级 13 的变更列表可以参见 http://d.android.com/sdk/api_diff/13/changes.html。

应用都喜欢更快的连接，这毋庸置疑。即使 3G 芯片比 Wi-Fi 芯片耗电低，但 Wi-Fi 的速率可以让数据在较短时间内完成传输，从而降低电量消耗。

注意 由于现在数据套餐流量有限（例如，每月 2GB 要 30 美元），通常用户会首选 Wi-Fi 连接。此外，应用可以用 NetworkInfo.isRoaming()来判断该设备目前是否在给定网络上漫游。因为漫游会增大花销，所以当 isRoaming()返回 true 时应避免数据传输。

表 7-2 显示了 T-Mobile G1 手机（也称为 HTC Dream，或 Era G1）各个组件的的电量消耗。虽然这手机现在看来有些过时（2008 年底发布），但这些数字依然揭示了每个组件电量消耗状况。

表7-2 Android G1 Phone功耗（来源：Google I/O 2009）

组 件	功 耗
闲置、飞行模式（关掉无线通讯）	2 mA
闲置、3G打开	5 mA
闲置、EDGE打开	5 mA
闲置、Wi-Fi打开	12 mA
显示（LCD）	90 mA（最低亮度：70 mA；最高亮度：110 mA）
CPU（100%负载）	110 mA
传感器	80 mA
GPS	85 mA
3G（最大传输速度）	150 mA
EDGE（最大传输速度）	250 mA
Wi-Fi （最大传输速度）	275 mA

由于各设备之间差异很大，所以需要知道应用会使用多少电量。G1 的电池是 1150 mAh，应用下载和播放视频（例如 YouTube）约 3 小时会耗光电量，假设它使用 3G 连接：3G 用 150 mA、CPU 用 90 mA、LCD 用 90 mA，共 330 mA，也许会坚持 3.5 小时（前提是手机没有运行其他应用）。

更多资源请访问稀酷客(www.ckook.com)

如果能控制数据的传输类型，就可以先压缩数据，再传输到设备上。虽然解压缩数据耗费 CPU，也多用了些电量，但传输速度大大加快，数据通讯设备（比如 3G 和 Wi-Fi）可以很快再次关闭，从而延长了电池寿命。通常的做法是：

- 使用 GZIP 压缩文本数据，使用 GZIPInputStream 类访问数据；
- 如果可能的话，使用 JPEG 而不是 PNG 格式的图像文件；
- 使用匹配设备分辨率的资源（比如，不必为 96 x 54 大小的显示空间下载 1920×1080 的图片）。

连接（例如 EDGE）越慢，压缩就越重要，这样可以减少通信设备打开的时间。

Android 在越来越多的设备上运行，从手机到平板电脑，从机顶盒到上网本，为所有这些设备生成相应资源可不是件容易的事。但是使用正确的资源，可以大大提高电池的使用寿命，会使应用更上一层楼。除了省电，上传下载越快，应用的响应也越快。

7.4　位置

任何房地产经纪人都反复向你强调一件事：位置。Android 也是，它可以让应用知道设备现在的具体位置（它不会告诉应用该设备是否在一个好学区里，但我想肯定将来会有应用这么干的）。代码清单 7-8 演示了如何向系统的定位服务请求位置更新。

代码清单 7-8　接收位置更新

```
private void requestLocationUpdates() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

    List<String> providers = lm.getAllProviders();

    if (providers != null && ! providers.isEmpty()) {
        LocationListener listener = new LocationListener() {

            @Override
            public void onLocationChanged(Location location) {
                Log.i(TAG, location.toString());
            }

            @Override
            public void onProviderDisabled(String provider) {
                Log.i(TAG, provider + " location provider disabled");
            }

            @Override
            public void onProviderEnabled(String provider) {
                Log.i(TAG, provider + " location provider enabled");
            }

            @Override
            public void onStatusChanged(String provider, int status, Bundle extras) {
                Log.i(TAG, provider + " location provider status changed to " + status);
            }
        };

        for (String name : providers) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            Log.i(TAG, "Requesting location updates on " + name);
            lm.requestLocationUpdates(name, 0, 0, listener);
        }
    }
}
```

注意 应用程序需要设置ACCESS_COARSE_LOCATION或者ACCESS_FINE_LOCATION权限才可以获取位置信息。GPS 定位服务需要 ACCESS_FINE_LOCATION 权限，而通过网络定位需要ACCESS_COARSE_LOCATION 或 ACCESS_FINE_LOCATION 权限。

这段代码有严重缺陷，不过你还是先运行一下，在应用的 onCreate()方法中，可看到此代码对电池寿命的影响。

在 Galaxy Tab 10.1 上，可以看到：

- 有 3 种定位服务方式（网络、GPS、被动定位服务）；
- GPS 定位信息更新频繁（每秒 1 次）；
- 网络定位更新频率低一些（每 45 秒 1 次）。

如果让应用运行一段时间，就会看到电池电量掉得比平时快得多。此代码的 3 个主要缺陷是：

- 没有注销位置监听器；
- 位置更新得太频繁；
- 同时使用了多种位置服务。

幸运的是，所有这些漏洞可以很容易地修复。如何使用位置服务取决于应用的需求，没有适合所有应用的万能解决方案，某个应用中的缺陷可能是另一个的特性。还要注意，通常也没有适用于所有用户的单一解决方案：你应考虑用户的需求，在应用中提供不同选项来满足用户需要。比如，一些用户可能会愿意牺牲电池寿命来频繁地更新位置，而其他人宁愿限制更新次数，以确保设备电量不会一个上午就消耗完。

7.4.1 注销监听器

调用 removeUpdates()可以注销监听器，如代码清单 7-9 所示。像处理广播接收器那样，可以在 onPause()中注销它，但应用需要改动其他许多地方。长时间监听位置更新将消耗大量的电量，应让应用程序设法得到它需要的信息后停止监听。在某些情况下，让用户强制固定某个位置是个好办法。

代码清单 7-9 禁用位置监听器

```
private void disableLocationListener(LocationListener listener) {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    lm.removeUpdates(listener);
}
```

更新频率可以用 requestLocationUpdates()调整。

更多资源请访问稀酷客(www.ckook.com)

7.4.2　更新频率

LocationManager 类定义了 5 个 requestLocationUpdates()方法，所有方法都有两个常见参数：minTime 和 minDistance。第一个参数 minTime 指定通知的最小时间间隔，以毫秒为单位。这仅是用来提示系统节省电力，实际的更新时间间隔未必与指定的相同。显然，值越大越省电。第二个参数 minDistance 指定通知的最小间隔距离，以米为单位。值越大也越省电，应用不用处理所有的位置更新也就执行了更少的指令。代码清单 7-10 演示了如何注册监听更新的最小时间间隔为 1 小时、最小间隔距离为 100 米。

代码清单 7-10　不频繁地接收位置更新

```
private void requestLocationUpdates() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

    List<String> providers = lm.getAllProviders();

    if (providers != null) {
        for (String name : providers) {
            LocationListener listener = new LocationListener() {
                …
            };

            Log.i(TAG, "Requesting location updates on " + name);
            lm.requestLocationUpdates(name, DateUtils.HOUR_IN_MILLIS * 1, 100, listener);
        }
    }
}
```

选择合适的时间间隔更像门艺术，这取决于应用的需求。导航类应用通常需要频繁地更新，但徒步者使用同样一款程序可能就不必更新得那么频繁。此外，你可能要在某些情况下牺牲一点精度来节省电力。给用户提供选项让他们自己调整，选择最适合的行为。

7.4.3　多种位置服务

如前所述，Android 提供了多种位置服务：

- GPS（使用卫星的全球定位系统）
- 网络（用 Cell-ID 来进行基站定位，Wi-Fi 服务地址）
- 被动（API 等级 8 加入）

GPS 定位服务（LocationManager.GPS_PROVIDER）通常是最准确的，水平精度约 10 米（11 码）。网络定位不如 GPS 准确，精度取决于可以用多少服务点来计算设备的位置。例如，我的日志中显示网络方式的定位精度是 48 米，大约是一个足球场的宽度。

GPS 带来高精度的同时，在时间和电量上的耗费也高。使用 GPS 服务“确定”位置需要同时锁定多个卫星的信号，在开阔场地需要几秒钟，如果设备在室内的话，定位或许只能徒劳无功，无法锁定信号（类似车载 GPS 无法在地下停车场中获得卫星信号）。例如，在房间里 GPS 启用后

更多资源请访问稀酷客(www.ckook.com)

花了 35 秒才初次确定位置，相同的测试在室外只花了 5 秒。通过网络只需 5 秒便初次确定位置，无论室内外。辅助 GPS（AGPS）通常会提供更快的定位，但实际时间取决于设备的缓存信息和网络状况。

注意 进行这些测试的三星 Galaxy Tab 10.1 是仅有 Wi-Fi 的版本。如果用上基站定位会更快。

接收多种定位服务的更新，不必限制为一个。事实上，你可能会使用多种定位来提高精度。

被动定位服务是最节省电量的一种。如果应用使用被动定位服务，表示它想知道位置更新信息但不想主动获取。换句话说，你的应用会等待其他应用、服务或系统组件发出定位请求，而后和其他监听器一起接收更新。代码清单 7-11 显示了如何获得被动的位置更新。为了测试应用是否接收到更新，打开另一个用到位置服务的应用，如 Maps。

代码清单 7-11 接受被动的位置更新

```
private void requestPassiveLocationUpdates() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    LocationListener listener = new LocationListener() {

        @Override
        public void onLocationChanged(Location location) {
            Log.i(TAG, "[PASSIVE] " + location.toString());

            // 这里只关心 GPS 更新

            if (LocationManager.GPS_PROVIDER.equals(location.getProvider())) {

                // 如果关心精度，需要调用 hasAccuracy()
                // （对海拔和方位角也是一样）

                if (location.hasAccuracy() && (location.getAccuracy() < 10.0f)) {
                    // 这里做事
                }
            }
        }

        @Override
        public void onProviderDisabled(String provider) {
            Log.i(TAG, "[PASSIVE] " + provider + " location provider disabled");
        }

        @Override
        public void onProviderEnabled(String provider) {
            Log.i(TAG, "[PASSIVE] " + provider + " location provider enabled");
        }

        @Override
        public void onStatusChanged(String provider, int status, Bundle extras) {
            Log.i(TAG, "[PASSIVE] " + provider + " location provider status changed to " + status);
```

更多资源请访问稀酷客(www.ckook.com)

```
        }
    };

    Log.i(TAG, "Requesting passive location updates");
    lm.requestLocationUpdates(LocationManager.PASSIVE_PROVIDER,
        DateUtils.SECOND_IN_MILLIS * 30, 100, listener);
}
```

如果使用这段代码，并开关 Wi-Fi 或 GPS，你会发现应用并没得到服务开启、关闭或状态变化的通知。这通常并不重要，但可能迫使你使用其他方法关注这些服务的变化。

一个好的权衡策略是注册网络定位服务，它比 GPS 省电，同时注册被动定位，以便可以从 GPS 得到更准确的位置信息。

注意　应用需要 ACCESS_FINE_LOCATION 权限来使用被动位置服务，即便它只接收来自网络的位置更新。这里的问题是，可能会引起用户对隐私的担忧。目前还没有办法只接收从网络获取的和只请求 ACCESS_COARSE_LOCATION 许可的被动更新。

7.4.4　筛选定位服务

既然我们的关注点是电池寿命，如果不能选择被动服务，应用可能要滤除高功耗的服务。代码清单 7-12 给出如何可以获得所有定位服务的功耗要求。

代码清单 7-12　定位服务功耗

```
private static String powerRequirementCodeToString(int powerRequirement) {
    switch (powerRequirement) {
    case Criteria.POWER_LOW: return "Low";
    case Criteria.POWER_MEDIUM: return "Medium";
    case Criteria.POWER_HIGH: return "High";
    default: return String.format("Unknown (%d)", powerRequirement);
    }
}

private void showLocationProvidersPowerRequirement() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);

    List<String> providers = lm.getAllProviders();

    if (providers != null) {
        for (String name : providers) {
            LocationProvider provider = lm.getProvider(name);
            if (provider != null) {
                int powerRequirement = provider.getPowerRequirement();
                Log.i(TAG, name + " location provider power requirement: " +
                    powerRequirementCode- ToString(powerRequirement));
            }
        }
```

更多资源请访问稀酷客(www.ckook.com)

```
    }
}
```

注意 和预想的一样，被动定位的功耗是未知的。

不过由于应用可能有非常具体的需求，首先尽可能用精度最高的定位服务。例如，应用可能需要使用坐标位置来报告速度信息。代码清单 7-13 演示了如何创建 Criteria 对象，并找出合适的定位服务。

代码清单 7-13 使用 Criteria 找出定位服务

```
private LocationProvider getMyLocationProvider() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    Criteria criteria = new Criteria();
    LocationProvider provider = null;

    // 这里定义自己的 criteria
    criteria.setAccuracy(Criteria.ACCURACY_COARSE);
    criteria.setAltitudeRequired(true);
    criteria.setBearingAccuracy(Criteria.NO_REQUIREMENT); // API 等级 9
    criteria.setBearingRequired(false);
    criteria.setCostAllowed(true); // 更希望用户能够设置
    criteria.setHorizontalAccuracy(Criteria.ACCURACY_LOW); // API 等级 9
    criteria.setPowerRequirement(Criteria.POWER_LOW);
    criteria.setSpeedAccuracy(Criteria.ACCURACY_MEDIUM); // API 等级 9
    criteria.setSpeedRequired(false);
    criteria.setVerticalAccuracy(Criteria.NO_REQUIREMENT); // API 等级 9

    List<String> names = lm.getProviders(criteria, false); // 只有完全匹配

    if ((names != null) && ! names.isEmpty()) {
        for (String name : names) {
            provider = lm.getProvider(name);
            Log.d(TAG, "[getMyLocationProvider] " + provider.getName() + " " + provider);
        }
        provider = lm.getProvider(names.get(0));
    } else {
        Log.d(TAG, "Could not find perfect match for location provider");

        String name = lm.getBestProvider(criteria, false); // 不必完全匹配

        if (name != null) {
            provider = lm.getProvider(name);
            Log.d(TAG, "[getMyLocationProvider] " + provider.getName() + " " + provider);
        }
    }

    return provider;
}
```

更多资源请访问稀酷客(www.ckook.com)

LocationManager.getProviders()和LocationManager.getBestProvider()差异相当明显。虽然getProviders()将只返回完全匹配的，getBestProvider()会先寻找完全匹配，如果找不到合适的就返回一个放宽条件的。标准依下列次序放宽：

- 电力要求
- 精度
- 方位角
- 速度
- 海拔高度

这个顺序不一定会符合你的具体要求，可能需要专门开发算法，寻找正确的定位服务。此外，该算法可能依赖于当前电池状态：如果电量低时应用未必会放松对电力的限制。

7.4.5　最后已知位置

在决定向定位服务注册位置监听器之前，你可能首先要检查位置是否已知（在系统缓存中）。LocationManager类定义了getLastKnownLocation()方法，它返回指定服务的最后已知位置，如果没有已知位置记录就返回null。虽然这个位置可能是过时的，但它往往是一个很好的起点，因为此位置可以即时获得，调用此方法也不用启动定位服务。即使注册了位置监听器，应用通常也要首先检索最后的已知位置，以增强响应性，因为通常得过几秒钟才能收到位置更新。代码清单7-14给出了如何获取最后已知位置的方法。

代码清单7-14　获取最后已知位置

```
private Location getLastKnownLocation() {
    LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
    List<String> names = lm.getAllProviders();
    Location location = null;

    if (names != null) {
        for (String name : names) {
            if (! LocationManager.PASSIVE_PROVIDER.equals(name)) {

                Location l = lm.getLastKnownLocation(name);

                if ((l != null) && (location == null || l.getTime() > location.getTime())) {
                    location = l;

                    /*
                     * 警告：GPS和网络定位服务的时钟可能没有同步，所以不推荐比较时间。 我们不必获
                       取最近期的位置
                     *
                     */
                }
            }
        }
```

更多资源请访问稀酷客(www.ckook.com)

```
    }

    return location;
}
```

GPS 只关心卫星信号，Android 设备还有其他类型的传感器，它们可让 Android 应用比传统电脑应用增添了很多趣味。

7.5 传感器

传感器太有意思了，所有人都爱不释手。传感器使用方式和定位服务类似：应用向特定的传感器注册监听器，获得更新通知。代码清单 7-15 演示了如何注册设备加速度计的监听器。

代码清单 7-15 向加速度计注册监听器

```
private void registerWithAccelerometer() {
    SensorManager sm = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

    List<Sensor> sensors = sm.getSensorList(Sensor.TYPE_ACCELEROMETER);

    if (sensors != null && ! sensors.isEmpty()) {
        SensorEventListener listener = new SensorEventListener() {

            @Override
            public void onAccuracyChanged(Sensor sensor, int accuracy) {
                Log.i(TAG, "Accuracy changed to " + accuracy);
            }

            @Override
            public void onSensorChanged(SensorEvent event) {
                /*
                 * 加速度计：3 个值的数组
                 *
                 * event.values [0] = x 轴的加速度减去 Gx
                 * event.values [1] = y 轴的加速度减去 Gy
                 * event.values [2] = z 轴的加速度减去 Gz
                 * /

                Log.i(TAG, String.format("x:%.2f y:%.2f z:%.2f ", event.values[0], event.values[1],
                    event.values[2]));

                // 在这里做些有意思的事情
            }
        };

        // 只需选择第一个
        Sensor sensor = sensors.get(0);
        Log.d(TAG, "Using sensor " + sensor.getName() + " from " + sensor.getVendor());

        sm.registerListener(listener, sensor, SensorManager.SENSOR_DELAY_NORMAL);
    }
}
```


更多资源请访问稀酷客(www.ckook.com)

和定位服务一样，Android 也可以让应用指定什么样的频度去获取传感器更新。位置服务用的是毫秒，只能指定以下 4 个值之一作为更新频率：

- SENSOR_DELAY_NORMAL
- SENSOR_DELAY_UI
- SENSOR_DELAY_GAME
- SENSOR_DELAY_FASTEST

三星 Galaxy Tab10.1（使用 InvenSense 的 MPL 加速度计）加速度计的 NORMAL、UI、GAME 和 FASTEST 的延迟分别约为 180、60、20、10 毫秒。这些数字因不同设备而异，更新越快需要越多的电量。例如，在 Android G1 手机上，使用 NORMAL 延迟需 10mA，而 FASTEST 延迟需 90mA（UI：15mA、GAME：80mA）。

像位置服务一样，降低通知频率是省电的最好办法。由于每个设备不同，应用可以测量这 4 种延迟通知的频率，选择兼顾用户体验和省电的那一个。另一种策略是（可能并不适用于所有应用），当发现值不常变化时，使用 NORMAL 或 UI 延迟，当发现有突然变化时，切换到 GAME 或 FASTEST 延迟。这种策略可以给某些应用带来可以接受的结果，并会延长电池寿命。

像其他监听器一样，当不需要通知时，要禁用传感器的监听器。可以用 SensorManager 的 unregisterListener()方法达到此目的。

7.6　图形

应用花费了很多时间在屏幕上画东西。无论是使用 GPU 渲染的 3D 游戏还是使用 CPU 的日历程序，都想只以最少的代价来在屏幕上展示期望的结果，以延长电池寿命。

如前所述，CPU 非全速时使用的电量少一些。现代的 CPU 使用动态调整频率和电压来节省电力和减少发热量。这两种技术通常一起使用，称为 DVFS 技术（动态电压和频率调整，Dynamic Voltage and Frequency Scaling），Linux 的内核、Android 及现代处理器都支持这种技术。

同样，现代 GPU 能够关闭内部组件，从一个完整核心到一个独立的管线，甚至可以在两帧渲染之间。

虽然你不能直接控制电压和频率，或将内部组件断电，但可以控制应用渲染的方式。流畅的帧速率通常是大多数应用首先要达到的，同时不要忘了降低功耗。尽管在 Android 设备上的帧速率通常有上限（例如，每秒 60 帧），但就算应用已达到最大帧速率，优化渲染程序还是会有效果的。除了可能降低能耗，你也为其后台运行的应用留出更多的空间，提供了更好的整体用户体验。

例如，一个典型缺陷是在活动壁纸中忘记调用 onVisibilityChanged()方法。事实上，壁纸可以是不可见的，但很容易被忽视，持续绘制壁纸会消耗很多电量。

第 8 章会介绍如何优化渲染的技巧。

更多资源请访问稀酷客(www.ckook.com)

7.7 提醒

应用可能出于某种原因，需要不时地被唤醒，去执行一些操作。典型的例子有 RSS 阅读器程序，每隔 30 分钟会被唤醒一次，下载 RSS feed，以便在程序启动之后让用户看到最新的内容；进行跟踪的应用每隔 5 分钟发送一条消息到其中一个联系人。代码清单 7-16 显示了如何创建提醒，唤醒应用，启动打印一条消息的服务，随后终止。代码清单 7-17 是服务的实现。

代码清单 7-16 设置提醒

```
private void setupAlarm(boolean cancel) {
    AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);

    Intent intent = new Intent(this, MyService.class);

    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);

    if (cancel) {
        am.cancel(pendingIntent); // 如果 Intent 匹配上，取消所有的提醒
    } else {
        long interval = DateUtils.HOUR_IN_MILLIS * 1;
        long firstInterval = DateUtils.MINUTE_IN_MILLIS * 30;

        am.setRepeating(AlarmManager.RTC_WAKEUP, firstInterval, interval, pendingIntent);
        // 使用 am.set(…) 进行非重复唤醒的调度
    }
}
```

代码清单 7-17 服务的实现

```
public class MyService extends Service {
    private static final String TAG = "MyService";

    @Override
    public IBinder onBind(Intent intent) {
        // 客户端不能绑定这个服务，返回 null
        return null;
    }

    @Override
    public void onStart(Intent intent, int startId) {
        super.onStart(intent, startId);

        Log.i(TAG, "Alarm went off - Service was started");

        stopSelf(); // 完成时记得调用 stopSelf() 释放资源
    }
}
```

类似前面看到的传感器事件监听器，在这里有个值对功耗的影响很大。下面是 AlarmManager.RTC_WAKEUP 的值，Android 定义了 4 种类型的提醒：

更多资源请访问稀酷客(www.ckook.com)

- ELAPSED_TIME
- ELAPSED_TIME_WAKEUP
- RTC
- RTC_WAKEUP

RTC 和 ELAPSED_TIME 类型的区别仅在于时间表示方式，RTC 和 RTC_WAKEUP 是 RTC 类型，用的是 Unix 时间戳，ELAPSED_TIME 和 ELAPSED_TIME_WAKEUP 是 ELAPSED_TIME 类型，是以系统启动为起点计算的时间，这 4 种类型的单位都是毫秒。

这里的_WAKEUP 后缀很关键。RTC 或 ELAPSED_TIME 对提醒时间时，如果设备在休眠，只有到下一次设备被唤醒时才会提交动作，而 RTC_WAKEUP 或 ELAPSED_TIME_WAKEUP 到时会把设备唤醒。显然，即使被唤醒的应用不做任何事情，不断唤醒也是很耗电的：

- 设备被唤醒，开始运行应用；
- 其他（行为正确的）提醒，等待设备唤醒时提交动作。

你可以看到，这会引发连锁反应。其他提醒可能很耗电，比如它们最终会触发 3G 数据传输，而你的应用则是造成这一切的元凶。

很少有应用真正需要到提醒时间时强行唤醒设备。当然闹钟这类程序会需要这种功能，但大多是等到设备唤醒（最常见的是用户主动唤醒设备）时才工作。和位置服务类似，使用被动方式会延长电池寿命。

调度提醒

多数情况下，应用需要在将来某一刻安排提醒到时，但对时间要求并不是很严格。为此 Android 定义了 AlarmManager.setInexactRepeating()，它的参数和其“兄弟”setRepeating()基本相同。主要区别是在系统如何安排提醒到时：Android 可以调整实际的触发时间，同时触发多个提醒（可能来自多个应用）。这种提醒更节能，系统也避免了出现不必要的唤醒。Android 定义了 5 个提醒间隔：

- INTERVAL_FIFTEEN_MINUTES
- INTERVAL_HALF_HOUR
- INTERVAL_HOUR
- INTERVAL_HALF_DAY
- INTERVAL_DAY

这些值用毫秒表示（例如，INTERVAL_HOUR 等于 3 600 000），setInexactRepeating()只接受这些间隔参数，用来创建非精确的提醒。传递任何其他值给 setInexactRepeating()都相当于调用 setRepeating()。代码清单 7-18 显示了如何使用不精确的提醒。

代码清单 7-18　设置不精确提醒

```
private void setupInexactAlarm(boolean cancel) {
    AlarmManager am = (AlarmManager) getSystemService(Context.ALARM_SERVICE);
```

更多资源请访问稀酷客(www.ckook.com)

```
    Intent intent = new Intent(this, MyService.class);

    PendingIntent pendingIntent = PendingIntent.getService(this, 0, intent, 0);

    if (cancel) {
        am.cancel(pendingIntent); // 会取消所有 Intent 和这个匹配的提醒
    } else {
        long interval = AlarmManager.INTERVAL_HOUR;
        long firstInterval = DateUtils.MINUTE_IN_MILLIS * 30;

        am.setInexactRepeating(AlarmManager.RTC, firstInterval, interval, pendingIntent);
    }
}
```

提示 没有 setInexact()方法。如果应用只想安排一次不精确的提醒，调用 setInexactRepeating()，到时后取消提醒。

显然，最好的结果是所有应用都使用这种提醒，而不用精确的触发提醒。为了尽可能节电，应用还可以让用户配置提醒的频度，因为有些人发现较长的时间间隔并不会对用户体验有负面影响。

7.8 WakeLock

在某些情况下，一些应用程序即使用户长时间不与设备交互，也要阻止设备进入休眠状态，来保持良好的用户体验。最简单的例子（也是最可能遇到的），就是当用户观看设备上的视频或电影时。这种情况下，CPU 需要做视频解码，同时屏幕保持开启，让用户能够观看。此外，视频播放时屏幕不能变暗。

Android 为这种场景设计了 WakeLock 类，如代码清单 7-19 所示。

代码清单 7-19 创建 WakeLock

```
private void runInWakeLock(Runnable runnable, int flags) {
    PowerManager pm = (PowerManager) getSystemService(Context.POWER_SERVICE);

    PowerManager.WakeLock wl = pm.newWakeLock(flags, "My WakeLock");

    wl.acquire();

    runnable.run();

    wl.release();
}
```

注意 应用需要 WAKE_LOCK 的权限来使用 WakeLock 对象。

更多资源请访问稀酷客(www.ckook.com)

系统的行为取决于创建 WakeLock 对象时传入的标记（flags）参数。Android 的定义了以下标记：

- PARTIAL_WAKE_LOCK（CPU 开）
- SCREEN_DIM_WAKE_LOCK（CPU 开、暗色显示）
- SCREEN_BRIGHT_WAKE_LOCK（CPU 开、明亮显示）
- FULL_WAKE_LOCK（CPU 开、明亮显示、键盘开）

这些标记可以结合使用：

- ACQUIRE_CAUSES_WAKEUP（打开屏幕和键盘）；
- ON_AFTER_RELEASE（WakeLock 释放后继续保持屏幕和键盘开启片刻）。

虽然使用它们非常简单，但如果不释放 WakeLock 会造成大问题。有缺陷的应用忘记释放 WakeLock，导致一直显示很长一段时间，很快耗光电量。一般情况下，要尽快释放 WakeLock。例如，应用在视频播放时获得 WakeLock，在暂停时最好释放，当用户继续播放视频时才再次获取。当应用暂停时也要释放 WakeLock，恢复应用时（如果视频还在放）再次获得。正如所见，要处理很多不同场景，应用很容易出现错误。

防止出现问题

为了防止问题出现，建议使用带超时的 WakeLock.acquire()版本，它会在超过时限后释放 WakeLock。例如，播放视频的应用可以使用视频长度作为 WakeLock 超时时间。

另外，如果用到的屏幕是和 Activity 中的 View 关联的，也可以在布局文件中使用 XML 属性 android:keepScreenOn。使用这种方法的好处是，不必冒着忘记释放 WakeLock 的风险，交由系统处理，也不需要在 manifest 文件中设置权限。代码清单 7-20 显示了如何在线性布局中使用该元素属性。

代码清单 7-20　XML 属性 keepScreenOn

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:keepScreenOn="true"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >

…

</LinearLayout>
```

提示　android:keepScreenOn 可用在任何 View 上。只要可见的 View 指定了要保持的屏幕，屏幕就会一直保留。你还可以用 View.setKeepScreenOn()方法控制是否要保持屏幕。

更多资源请访问稀酷客(www.ckook.com)

WakeLocks 尽管会导致问题，但它有时还是必要的。如果你用了它们，确保想清楚：何时获取、何时释放。另外，最好完全理解应用的生命周期，并确保有详尽的测试用例。有 bug 时，出问题的只能是 WakeLocks！

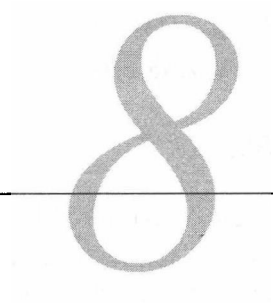
7.9 总结

用户通常不会注意到应用是否延长了电池寿命。但是，如果不做，更有可能会被注意到。因为单个应用可以让所有其他人的努力化为泡影，所有应用都需要通力配合，最大限度地延长电池的使用时间。用户会卸载耗电的应用，要合理使用电池，但也应当给用户配置选项的自由，因为不同的用户会有不同的需求。给你的用户配置的权利。[①]

① 关于使用网络和节电的文章可以参考 http://www.research.att.com/articles/featured_stories/2011_03/201102_Energy_efficient，很有价值的一篇文章。——译者注

更多资源请访问稀酷客(www.ckook.com)

第8章 图 形



很多时候，你都得花大量时间琢磨应用该是什么样子。无论是使用标准 Android 小部件的电子邮件应用还是使用 OpenGL ES 的游戏，应用的界面是用户浏览 Android Market[①] 或 Amazon Appstore 首先在意的。

有些应用看起来不错，但图形很久才显示出来或刷新慢吞吞，这样怎么能取得巨大成功？同样，那些画质极好但帧速较低的游戏也注定将被用户淘汰。用户的评价最终会影响应用是否成功，因此不要做华而不实的玩意儿。

本章将介绍一些基本方法，利用各种技术和工具优化布局以及优化 OpenGL ES 渲染，提高帧率降低功耗。

8.1 布局优化

你应该已经熟悉 XML 布局和 setContentView()方法了。这方法的典型用法如代码清单 8-1 所示。许多人觉得设计布局很简单，尤其是使用 Eclipse 的可视布局设计工具，这很容易让人迷失而忘了优化布局。本节提供了几个简单的方法，可以简化布局、加快布局展开。

代码清单 8-1 典型的 setContentView()调用

```
public class MyActivity extends Activity {
    private static final String TAG = "MyActivity";

    /** 当 Activity 第一次创建时调用。*/
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        //调用 setContentView 展开并设定布局 (main.xml 中定义)
        setContentView(R.layout.main);

        ...
    }
```

① 现为 Google Play 商店。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
    ...
}
```

虽然这个调用非常简单，让我们揭开 setContentView()的盖子，看看里面发生了什么：

- Android 读取应用的资源数据（APK 文件内，存储在内部存储器或 SD 卡中）；
- 解析资源数据，展开布局；
- 布局展开成为 Activity 的顶层视图。

此调用花费的时间取决于布局的复杂性：资源数据越大解析越慢，而更多的类也让布局实例化变慢。

当在 Eclipse 中创建 Android 项目后，会生成默认的布局 main.xml，如代码清单 8-2 所示。TextView 的文字定义在 strings.xml 中。

代码清单 8-2 默认布局

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</LinearLayout>
```

使用这个布局来调用 setContentView()（如代码清单 8-1 所示），在三星 Galaxy Tab 10.1 上大约需要 17 毫秒的时间。这相当快。但是，其中只有两个类会被实例化（LinearLayout 和 TextView），在 XML 文件中也只指定了极少数的属性，所以这是非常简单的布局，不代表真实的典型 Android 应用程序。

注意 可以写代码创建布局，不过通常首选 XML。

在默认布局中加入多个部件后，总共达到了 30 个部件（包括 ScrollView、EditText 和 ProgressBar），调用 setContentView()花费的时间超过 163 毫秒。

可以看到，随着部件数量的增长，展开布局所花费的时间几乎呈线性增长。此外，调用 setContentView()几乎占用了从 onCreate()开始到 onResume()结束之间所有时间的 99%。

你可以在布局中增删部件来测量时间开销。使用 Eclipse 中可视化 XML 文件布局视图很容易。如果已定义了应用的布局，你应该做的第一件事是测量需要花多少时间展开。布局通常在 Activity 的 onCreate()方法中展开，它花费的时间会直接影响 Activity，也就是应用的启动时间。因此，建议尽量减少布局展开花费的时间。

更多资源请访问稀酷客(www.ckook.com)

为了实现这一目标，有多种技术可选，其中大多数是基于同样的原则：减少创建对象数量。你可以用不同的布局达到同样的视觉效果，消除不必要的对象，或推迟创建对象。

8.1.1 相对布局

线性布局一般是开发人员开始学习时用到的第一种布局。事实上，这种布局是代码清单 8-1 所示的默认布局的一部分，因此也是开发人员熟悉的第一个 ViewGroup。这个布局很简单，可以这样理解，线性布局基本上是水平或垂直对齐的部件容器。

很多 Android 开发新手会使用嵌套的线性布局来达到预期效果。代码清单 8-3 就是个嵌套的线性布局示例。

代码清单 8-3　嵌套线性布局

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/text1"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str1" android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text2"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str2" android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
            android:id="@+id/text3"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str3" android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text4"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str4" android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

    <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:orientation="horizontal" >
        <TextView
```

更多资源请访问稀酷客(www.ckook.com)

```
            android:id="@+id/text5"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str5" android:textAppearance="?android:attr/textAppearanceLarge" />
        <TextView
            android:id="@+id/text6"
            android:layout_width="wrap_content" android:layout_height="wrap_content"
            android:text="str6" android:textAppearance="?android:attr/textAppearanceLarge" />
    </LinearLayout>

</LinearLayout>
```

这布局的核心是6个文本视图，4个线性布局在这里只是辅助定位。

这个布局暴露了两个问题：

- 嵌套线性布局会深化布局层次，从而导致布局和按键处理变慢；
- 10个对象，4个只用来定位。

用相对布局取代所有线性布局，即可轻松解决这两个问题。如代码清单8-4所示。

代码清单8-4 相对布局

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:layout_alignParentTop="true"
        android:text="@string/str1" android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text2"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text1"
        android:layout_alignParentTop="true"
        android:text="@string/str2" android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text3"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/text1"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:text="@string/str3" android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text4"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text3"
        android:layout_below="@id/text2"
        android:text="@string/str4" android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text5"
        android:layout_alignParentLeft="true"
        android:layout_below="@id/text3"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
```


更多资源请访问稀酷客(www.ckook.com)

```
        android:text="@string/str5" android:textAppearance="?android:attr/textAppearanceLarge" />
    <TextView
        android:id="@+id/text6"
        android:layout_width="wrap_content" android:layout_height="wrap_content"
        android:layout_toRightOf="@id/text5"
        android:layout_below="@id/text4"
        android:text="@string/str6" android:textAppearance="?android:attr/textAppearanceLarge" />

</RelativeLayout>
```

如上所见，所有 6 个文字视图都放在一个相对布局中，因此只创建了 7 个对象，而不是 10 个。布局层次也不深，它只在文字视图的上一层。定位部件的关键在于 layout_***属性。Android 定义了很多这样的属性，你可以用它来确定布局中的各种元素的位置：

- layout_above
- layout_alignBaseline
- layout_alignBottom
- layout_alignLeft
- layout_alignRight
- layout_alignTop
- layout_alignParentBottom
- layout_alignParentLeft
- layout_alignParentRight
- layout_alignParentTop
- layout_alignWithParentIfMissing
- layout_below
- layout_centerHorizontal
- layout_centerInParent
- layout_centerVertical
- layout_column
- layout_columnSpan
- layout_gravity
- layout_height
- layout_margin
- layout_marginBottom
- layout_marginLeft
- layout_marginRight
- layout_marginTop
- layout_row
- layout_rowSpan
- layout_scale

更多资源请访问稀酷客(www.ckook.com)

- layout_span
- layout_toLeftOf
- layout_toRightOf
- layout_weight
- layout_width
- layout_x
- layout_y

注意 有些属性只用于某些类型的布局。例如，layout_column、layout_columnSpan、layout_row、layout_rowSpan 用在网格布局中。

相对布局在条目列表中非常重要，应用通常用它来显示 10 个及以上的项目。

8.1.2 合并布局

另一种减少布局层次的技巧是用<merge />标签来合并布局。往往自己布局的顶层元素是一个 FrameLayout，如代码清单 8-5 所示。

代码清单 8-5 框架布局

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:id="@+id/my_top_layout" >

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</FrameLayout>
```

因为 Activity 内容视图的“父亲”是个 FrameLayout，所以最终会在布局中出现两个 FrameLayout 对象：

- 自己的 FrameLayout；
- Activity 内容视图的“父亲”——另一个 FrameLayout 其中只有一个“孩子”（你自己的 FrameLayout）。

图 8-1 所显示的布局中假设自己的 FrameLayout 有两个子视图：ImageView 和 TextView。

更多资源请访问稀酷客(www.ckook.com)

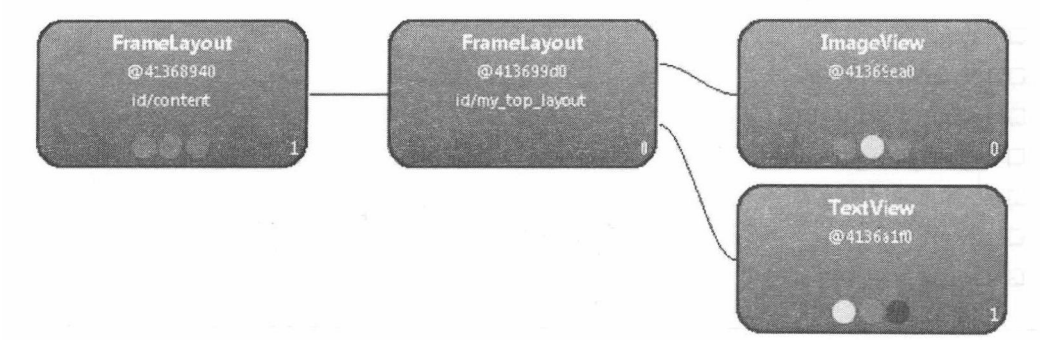


图 8-1　FrameLayout 是另一个 FrameLayout 的“孩子”

这里有两个 FrameLayout，可以用<merge />标签替换自己的 FrameLayout，减少层次。这样，Android 会将<merge />标签中的子标签放入父 FrameLayout。代码清单 8-6 为新的 XML 布局。

代码清单 8-6　合并标签

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android">

    <ImageView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent" />

    <TextView
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="@string/hello" />

</merge>
```

这只是把<FrameLayout />换为<merge />标签。图 8-2 显示了由此产生的布局。

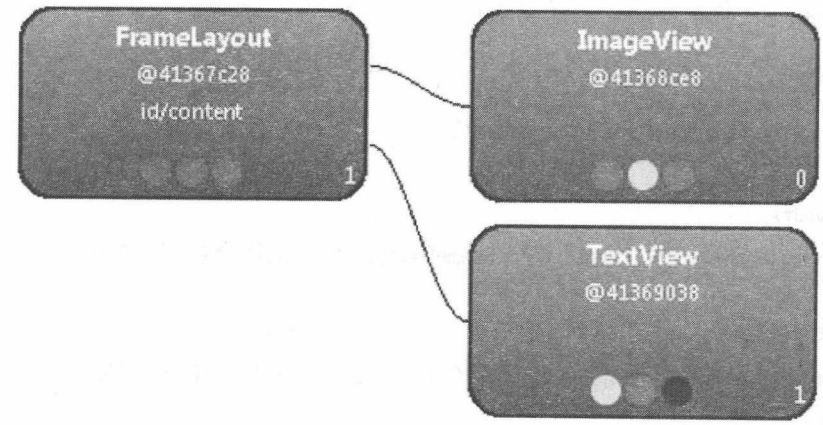


图 8-2　用<merge />替换<FrameLayout />

更多资源请访问稀酷客(www.ckook.com)

8.1.3 重用布局

类似 C 或 C++中的 # include 指令，Android 支持在 XML 布局中使用<include />标签。简而言之，<include />标签包含另一个 XML 的布局，如代码清单 8-7 所示。

<include />标签可用于两个目的：

- 多次使用相同的布局；
- 布局有一个通用的组成部分，或有部分依赖于设备配置（例如，屏幕方向纵置或横置）。

代码清单 8-7 显示如何多次包含布局，同时覆盖包含布局的一些参数。

代码清单 8-7 多次包含布局

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <include android:id="@+id/myid1" android:layout="@layout/mylayout" android:layout_margin="9dip" />
    <include android:id="@+id/myid2" android:layout="@layout/mylayout" android:layout_margin="3dip" />
    <include android:id="@+id/myid3" android:layout="@layout/mylayout" />

</LinearLayout>
```

代码清单 8-8 显示了如何只包含一次布局，但依赖设备的屏幕方向，要么 layout-land/mylayout.xml，要么 layout-port/mylayout.xml（这里假设有两个版本的 mylayout.xml，一个在 res/layout-land 目录，另一个在 res/layout-port）。

代码清单 8-8 根据屏幕方向包含布局

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <include android:id="@+id/myid" android:layout="@layout/mylayout" />

</LinearLayout>
```

当布局包含进来时，还可以覆盖一些布局的参数，如：

- 根视图的 id（android:id）；
- 布局参数（android:layout_*）；

注意 覆盖布局的参数是可选的。只有 android:layout 属性在<include />标签中是必需的。

如代码清单 8-8 所示，在布局展开时，包含的布局是动态处理的。包含是在编译时完成，但 Android 不会知道，包含的是两个布局（layout-land/mylayout.xml 或 layout-port/mylayout.xml）中的哪一个。这不同于 C 或 C++的在编译期由预处理器处理的#include 指令。

更多资源请访问稀酷客(www.ckook.com)

8.1.4 ViewStub

我们在第1章曾介绍过，推迟初始化是个方便的技术，可以推迟实例化，提高性能，还可能会节省内存（如果对象从未创建过）。

为此，Android定义了ViewStub类。ViewStub是轻量级且不可见的视图，当需要时，在自己的布局中可以用它来推迟展开布局。代码清单8-9是代码清单8-8修改后的版本，演示了如何在XML布局中使用ViewStub。

代码清单8-9　XML中的ViewStub

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

  <ViewStub
      android:id="@+id/mystubid"
      android:inflatedId=" @+id/myid"
      android:layout="@layout/mylayout" />
  <|Linearlayout>
```

LineaLayout被展开时，它只会包含一个“孩子”——ViewStub。ViewStub引用的布局是@layout/mylayout，它可能是非常复杂的布局，展开很花时间，不过此时不必展开。要展开mylayout.xml中的布局，你有两个选择，如代码清单8-10和代码清单8-11所示。

代码清单8-10　在代码中展开布局

```
ViewStub stub = (ViewStub) findViewById(R.id.mystubid);
View inflatedView = stub.inflate();// inflatedView 定义在 mylayout.xml 的布局中
```

代码清单8-11　在代码中用setVisibility()展开布局

```
View view = findViewById(R.id.mystubid);
view.setVisibility(View.VISIBLE); // ViewStub 取代展开的布局
view = findViewById(R.id.myid); // 现在要得到展开的视图
```

第一种方式（代码清单8-10）展开布局似乎更方便，有些人可能认为它有一个小问题：代码要知道视图是个ViewStub，需要显式展开。在大多数情况下，这不是个问题，因而建议以这种方式在应用布局中使用ViewStub。

第二种方式（代码清单8-11），因为它不用指明是ViewStub类，所以更通用。然而，如上所示的代码还是清楚地知道它用的是ViewStub，这是因为它使用了两个不同的id：R.id.mystubid和R.id.myid。想完全通用，应该如代码清单8-12所示定义布局，如代码清单8-13所示展开布局。代码清单8-12和代码清单8-9基本相同，区别是前者只创建了一个id（R.id.myid），而不是两个。同样，代码清单8-13与代码清单8-11基本相同，只是用R.id.myid替换了R.id.mystubid。

更多资源请访问稀酷客(www.ckook.com)

代码清单 8-12 在 XML 中 ViewStub 没有覆盖 ID

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:orientation="vertical" >

    <ViewStub
        android:id="@+id/myid"
        android:layout="@layout/mylayout" />

</LinearLayout>
```

代码清单 8-13 用 setVisibility() 展开布局

```
View view = findViewById(R.id.myid);
view.setVisibility(View.VISIBLE); // ViewStub 被展开的布局所替换
view = findViewById(R.id.myid); // 现在要获取展开后的布局
```

事实上，代码清单 8-13 是否有效不受在布局中是否使用 ViewStub 的影响。如果代码的通用性很重要，无论在布局中有没使用 ViewStub，都能正常工作，那么这种方法是首选。不过调用两次 findViewById()会影响性能。为了部分解决这个问题，既然调用 setVisibility(View.VISIBLE)会把 ViewStub 从父容器中删除，你可以在第二次调用 findViewById()之前检查视图是否有“父亲”。对仍然使用 ViewStub 来说，第二次调用 findViewById()还是没优化，它只保证没用 ViewStub 时，只调用一次 findViewById()。修改后的代码见代码清单 8-14。[①]

代码清单 8-14 在可能的情况下只调用 findViewById()一次

```
View view = findViewById(R.id.myid);
view.setVisibility(View.VISIBLE); // ViewStub 被展开的布局替换（如果布局中有的话）
if (view.getParent() == null) {
    // 用了 ViewStub，需要用展开后的新视图替换
    view = findViewById(R.id.myid);
} else {
    // 不做任何事，这个第一次获取到的视图就是我们想要的
}
```

代码清单 8-13 和代码清单 8-14 的做法很少见，你通常不会照做。一般情况下，代码知道布局中使用了 ViewStub 是可以接受的，因此代码清单 8-10 所示的展开 ViewStub 的方法就够用了。

8.2 布局工具

为了协助优化布局，Android SDK 提供了两个易于使用的工具：hierarchyviewer 和 layoutopt。这些工具位于 SDK 的 tools 目录下，这个目录下还有一些其他工具。

① 最好自己手动测试开销到底多大。——译者注

更多资源请访问稀酷客(www.ckook.com)

8.2.1 层级视图

Android SDK 中有个非常有用的工具——hierarchyviewer，可用来查看和分析应用的布局。事实上，图 8-1 和图 8-2 都是用该工具生成的。除了显示应用的详细布局，这个工具还可以得出需要多少时间测量、布局并绘制每个部件，并确定哪些部件需要较长时间来测量、布局和绘制。

hierarchyviewer 可以作为一个独立的工具，或直接在 Eclipse 中使用“Hierarchy View”工作视图。

8.2.2 layoutopt

layoutopt 是 Android SDK 的另一工具，它可以帮助优化布局。此工具分析能够布局文件，提出修改建议，让布局更高效。

例如，使用 layoutopt 分析代码清单 8-5 的布局，得到如下输出结果：

```
The root-level <FrameLayout/> can be replaced with <merge/>
```

事实上，这就是代码清单 8-6 做的事情。针对代码清单 8-6，layoutopt 没有给出任何建议。

提示 *使用最新版本的 layoutopt 工具，以确保得到最佳结果。*

确保在发布应用之前认真处理 layoutopt 报告的所有问题，这些问题通常很容易纠正。未优化的布局会拖慢应用。

8.3 OpenGL ES

对今天的 Android 设备和应用来说，三维渲染是越来越重要的功能了。成为 3D 渲染的专家需要相当长的时间，不过下文将通过一些很容易实现的简单技术，以及需要掌握的一些基本概念，从而使你大体学会使用这项技术。如果你想了解更多 Android 的 OpenGL ES，可以参考 Mike Smithwick 和 Mayank Verma 合著的 *Pro OpenGL ES for Android*。

最近的 Android 设备同时支持 OpenGL ES 1.1 和 OpenGL ES 2.0，而旧设备只支持 OpenGL ES 1.1。截至 2011 年 12 月，在连接到 Android Market（即 Google Play 商店）的设备中，90%都支持 OpenGL ES 2.0。[①]

OpenGL ES 是由 Khronos Group 提出的标准，市面上有几种实现。Android 设备中常见支持 OpenGL ES 的 GPU 有：

- ARM 的 Mali（例如: Mali 400-MP4）
- Imagination Technologies 的 PowerVR（例如: PowerVR SGX543）
- Nvidia 的 GeForce（Nvidia Tegra）

---

① OpenGL ES 3.0 规范 2012 年 8 月发布。——译者注

更多资源请访问稀酷客(www.ckook.com)

- 高通的 Adreno（将此前的 ATI Imageon 从 AMD 处收购，整合到高通自家的 Snapdragon 系列芯片中[①]）

8.3.1 扩展

OpenGL 标准支持扩展，允许在某些 GPU 中纳入新功能。例如，下面是三星 Galaxy Tab 10.1 支持的扩展（基于 NVIDIA Tegra 2）：

- GL_NV_platform_binary
- GL_OES_rgb8_rgba8
- GL_OES_EGL_sync
- GL_OES_fbo_render_mipmap
- GL_NV_depth_nonlinear
- GL_NV_draw_path
- GL_NV_texture_npot_2D_mipmap
- GL_OES_EGL_image
- GL_OES_EGL_image_external
- GL_OES_vertex_half_float
- GL_NV_framebuffer_vertex_attrib_array
- GL_NV_coverage_sample
- GL_OES_mapbuffer
- GL_ARB_draw_buffers
- GL_EXT_Cg_shaders
- GL_EXT_packed_float
- GL_OES_texture_half_float
- GL_OES_texture_float
- GL_EXT_texture_array
- GL_OES_compressed_ETC1_RGB8_texture
- GL_EXT_texture_compression_latc
- GL_EXT_texture_compression_dxt1
- GL_EXT_texture_compression_s3tc
- GL_EXT_texture_filter_anisotropic
- GL_NV_get_text_image
- GL_NV_read_buffer

① 此说随属主流，但却有误，2006 年 AMD 收购了 ATI，但 2009 年初，高通并非收购 Imageon 部门，而是收购的“矢量绘图与 3D 绘图的相关技术及知识产权”，高通现在的 Snapdragon 芯片与 Imageon 没有太大关系，其所用的图形技术更多来源于以上所收购的矢量及 3D 绘图技术专利。——编者注

更多资源请访问稀酷客(www.ckook.com)

- GL_NV_shader_framebuffer_fetch
- GL_NV_fbo_color_attachements
- GL_EXT_bgra
- GL_EXT_texture_format_BGRA8888
- GL_EXT_unpack_subimage
- GL_NV_texture_compression_st3c_update

代码清单 8-15 显示了如何检索设备支持的扩展列表。由于目前没有 Android 模拟器支持 OpenGL ES 2.0，要在真实设备上运行代码。

代码清单 8-15　OpenGL ES 的扩展

```
// 扩展列表作为字符串返回（需要解析来找到指定扩展）

String extensions = GLES20.glGetString(GLES20.GL_EXTENSIONS);

Log.d(TAG, "Extensions: " + extensions);
```

你需要 OpenGL 上下文执行代码清单 8-15 中的代码，获取扩展列表。例如，可以在 `GLSurfaceView.Renderer` 的 `onSurfaceChanged()`方法中执行这段代码。如果在没有 OpenGL 时运行这段代码，那么你会在 Logcat 中看到“Call to OpenGL ES API with no current context”消息。

正如所见，上述扩展可分为多个组：

- GL_OES_*
- GL_ARB_*
- GL_EXT_*
- GL_NV_*

GL_OES_*扩展已由 Khronos OpenGL ES Working Group 批准。虽然它们不是 OpenGL ES 必需的，但也被广泛使用。

GL_ARB_*扩展已由 OpenGL Architecture Review Board 批准。GL_EXT_*扩展由多家厂商商定，而 GL_NV_*扩展是 Nvidia 独有的。查看扩展支持列表，通常可以知道设备的 GPU 是哪个公司提供的：

- ARM 的扩展使用 GL_ARM 前缀；
- Imagination Technologies 的扩展使用 GL_IMG 前缀；
- Nvidia 的扩展使用 GL_NV 前缀；
- Qualcomm 的扩展使用 GL_QUALCOMM、GL_AMD 和 GL_ATI 前缀。

注意　访问 http://www.khronos.org/registry/gles 和http://www.opengl.org/registry可获取更多信息。

因为不是所有设备都支持相同的扩展（又见碎片化），为特定的设备进行优化时，必须非常小心，因为一台设备上正常的程序，可能无法运行在另一台上。典型的例子包括使用非 2 幂次维

更多资源请访问稀酷客(www.ckook.com)

度（Non-Power-Of-Two，NPOT）[①]纹理或纹理压缩格式，在一台设备上测试正常，在另一台就不正常。如果你打算支持非 Android 设备，还要检查那些设备支持的扩展列表。例如，目前苹果的 iPhone/iPod/iPad 设备都是基于 Imagination Technologies 的 PowerVR（它们之间可能会，也可能不会共享相同的扩展），但将来的产品未必还是基于 PowerVR。

8.3.2 纹理压缩

纹理定义了 OpenGL ES 应用的外观。虽然未压缩的纹理使用起来很方便，但这种纹理会迅速让应用变得冗赘不堪。例如，未压缩的 256×256 的 RGBA8888 纹理就会占用 256KB 的内存。

未压缩的纹理对性能有负面影响，它们不像压缩过的能够缓存（因为比较小），而且还要多次访问内存（影响功耗）。使用未压缩纹理的应用体积也更为庞大，需要更多的时间来下载安装。此外，Android 设备的内存通常是有限的，所以应尽可能占用较少内存。

在上面的列表中可看到，下列扩展是三星 Galaxy Tab 10.1 支持的纹理压缩格式：

- GL_OES_compressed_ETC1_RGB8_texture
- GL_EXT_texture_compression_latc
- GL_EXT_texture_compression_dxt1
- GL_EXT_texture_compression_s3tc

ETC1 压缩格式由爱立信（ETC 即 Ericsson Texture Compression）提出，大多数 Android 设备（所有支持 OpenGL ES 2.0 的设备）都支持。因此，如果要支持尽可能多的设备，它是最安全的选择。这种压缩格式使用每像素 4 位，而不是 24（ETC1 不支持 alpha），因此达到 6 倍压缩比：包括 16 字节的头，256 × 256 的 ETC1 纹理占用 32 784 字节。

有多种工具可以创建 ETC1 纹理。Android SDK 中有 etc1tool，它是将 PNG 图像编码压缩为 ETC1 图像的命令行工具。例如，下面一行显示了如何压缩 lassen.png（183KB），后面给出了原始图像和压缩后的差别：

```
etc1tool lassen.png --encode --showDifference lassen_diff.png
```

如果不指定，输出文件名将根据输入文件名来生成。在此例中，输出文件是 lassen.pkm（33KB）。图 8-3 显示了压缩图像和原始图像之间的区别。

注意 *也可以用 etc1tool 把 ETC1 图像解码成 PNG 图像。如需了解 etc1tool 的信息，请参考* http://d.android.com/guide/developing/tools/etc1tool.html。

不幸的是，Android SDK 中的 etc1tool 提供的选项很少。对于那些希望能够调优压缩和视觉质量的开发者，还有另一个工具值得推荐——etcpack。从爱立信网站（http://devtools.ericsson.com/etc）可以下载 etcpack 工具，它包含更多选项：

- 压缩速度（压缩速度与图像质量成反比）

① 参见http://www.opengl.org/wiki/NPOT_Texture。——译者注

更多资源请访问稀酷客(www.ckook.com)

- 误差度量（可感知或不可感知的图形细节）
- 方向

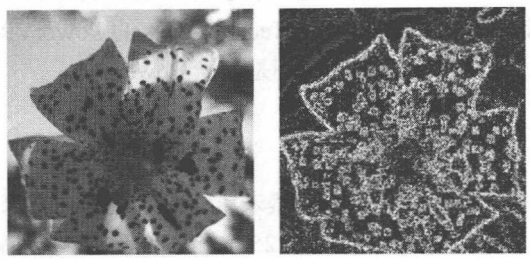


图 8-3　原始图像（左）和压缩后的图像（右）的区别

在决定离线生成 ETC1 纹理时（也就是说，不在 Android 设备上生成），发布的应用一定要采用最高质量的压缩方式。由于人类的眼睛对绿色的敏感度超过红色或蓝色，所以应该选择 perceptual（可感知的）作为误差度量（该算法中绿色接近其原始值，放宽蓝色和红色，降低峰值信噪比）。

虽然区别不明显，但没有理由在发布的应用中使用低质量而不是更高质量的 ETC1 纹理，它们的大小是相同的。尽管产生更高质量的纹理需要花更多的时间，许多开发者选择在开发过程中用低质量的纹理，但应该永远记住在发布应用时一定要换为高品质纹理。

另外，你也可以使用 ARM Mali GPU Compression Tool，从 http://www.malideveloper.com/developer-resources/tools 可以免费获得。此工具提供的选项类似 etcpack 但有图形界面（也有命令行版本）。图 8-4 显示了 ARM Mali GPU Compression Tool 使用之前相同的输入文件的界面。

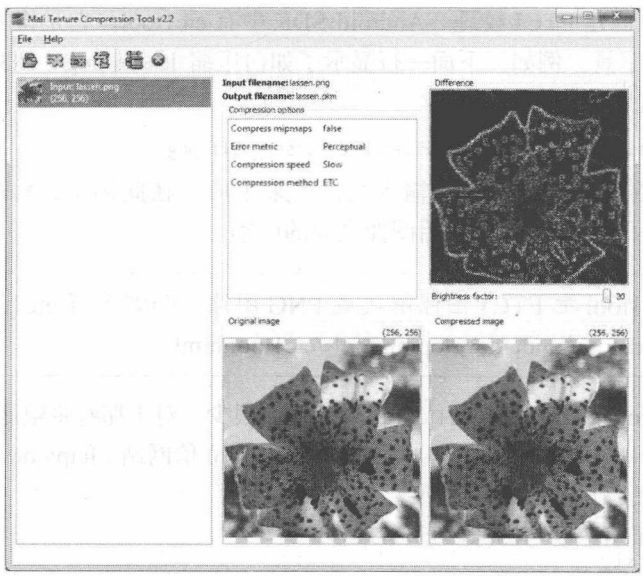


图 8-4　ARM Mali GPU Compression Tool

更多资源请访问稀酷客(www.ckook.com)

图 8-5 显示了高质量（慢）压缩和低质量（快）压缩之间的差异。再一次看到，差异并不明显。

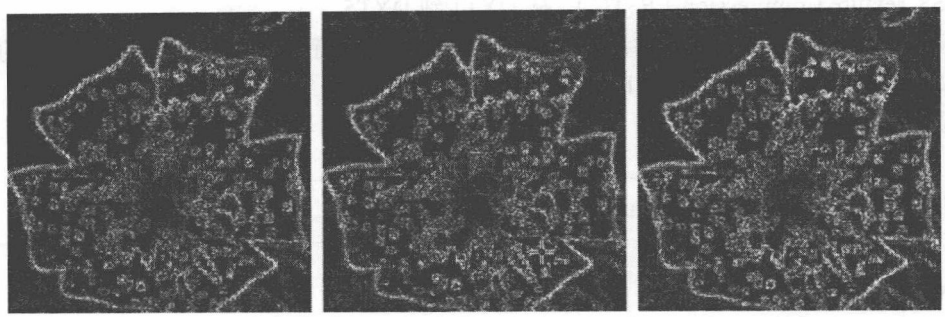


图 8-5 高质量（左）、中等质量（中）、低质量（右）

ARM Mali 开发者网站上还有许多其他工具可用。你可以在 ARM、Imagination Technologies、Nvidia、Qualcomm 的网站上找到许多创作和调试工具：

- http://developer.nvidia.com/tegra-android-development-pack
- http://developer.qualcomm.com/develop/mobile-technologies/graphics-optimization-adreno

例如，Imagination Technologies 也提供了创建压缩纹理的工具 PVRTexTool；Nvidia 网站提供了完整的 SDK（包括 Android SDK、NDK、Eclipse 及相关示例代码）。

注意 你可能需要注册后才能访问这些网站上的各种工具和文档。

从 Android 2.2（API 等级 8）开始，有了如下的类可以帮助使用 ETC1 纹理：

- `android.opengl.ETC1`
- `android.opengl.ETC1Util`
- `android.opengl.ETC1Util.ETC1Texture`

应用可以动态地使用 `ETC1.encodeImage()`和 `ETC1Util.compressTexture()`方法压缩图像。当 ETC1 纹理不能离线使用时，比如，当纹理是基于存储设备上的图片（也许是通讯录中某用户的头像）时，这是非常有用的。

提示 即便 ETC1 不支持透明度（压缩纹理中没有 alpha 分量），你也可以加上另一个只包含透明度信息的单通道纹理，把二者结合起来着色。

1. 其他纹理压缩格式

ETC1 虽然是最常见的纹理压缩格式，还有某些设备支持的另外几种格式。[①]其中包括：

① 伴随 OpenGL ES 3.0 发布，也提出了一个新的纹理压缩格式标准。——译者注

更多资源请访问稀酷客(www.ckook.com)

- PowerVR Texture Compression（PVRTC）
- ATI Texture Compression（ATC 或 ATITC）
- S3 Texture Compression（S3TC），从 DXT1 到 DXT5

你应该根据不同的目标设备尝试不同的压缩格式。由于特定的 GPU 可以优化自己的压缩格式（例如，PowerVR GPU 为 PVRTC 压缩格式优化），专有压缩格式可能会比标准 ETC1 压缩格式效果更好。

注意　还可以使用苹果的 iOS texturetool 产生 PVRTC 纹理。

2. Manifest 清单文件

在 OpenGL 应用的 manifest 文件中，要指定两件事：

- 需要设备支持的 OpenGL 版本号；
- 应用程序支持的纹理压缩格式。

代码清单 8-16 给出如何指定设备支持 OpenGL ES 2.0，以及指明应用支持的纹理压缩格式只有两个：ETC1 和 PVRTC。

代码清单 8-16　Manifest 和 OpenGL

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.apress.proandroid.opengl"
    android:versionCode="1" android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="8" />

    <uses-feature android:glEsVersion="0x00020000" />

    <supports-gl-texture android:name="GL_OES_compressed_ETC1_RGB8_texture" />
    <supports-gl-texture android:name="GL_IMG_texture_compression_pvrtc" />

    ...

</manifest>
```

注意　OpenGL ES 的版本是 16.16 数[①]，所以 0x00020000 即 OpenGL ES 2.0。

当设备连接到 Android Market 时，Android Market 会用此信息来过滤应用。例如，只支持 OpenGL ES 1.1 的设备将无法看到只用 OpenGL ES 2.0 的应用。同样，只支持 ETC1 纹理压缩格式的设备将不会看到只用 PVRTC 和 ATC 纹理的应用程序。

如果应用没在其 manifiest 文件中指定纹理压缩格式，那么 Android Market 将不会用纹理压缩

① 即 16 位主版本.16 位次版本。——译者注

更多资源请访问稀酷客(www.ckook.com)

格式进行过滤（认为应用支持所有的压缩格式）。这可能会导致用户安装后才发现应用不能运行，给出负面评论。

8.3.3 Mipmap

通常，三维场景中出现在背景的对象，在屏幕上不会占用多少空间。例如，屏幕上显示为10×10的对象使用256×256纹理就是浪费资源（内存、带宽）。Mipmap通过提供多层次细节的纹理来解决这个问题，如图8-6所示。

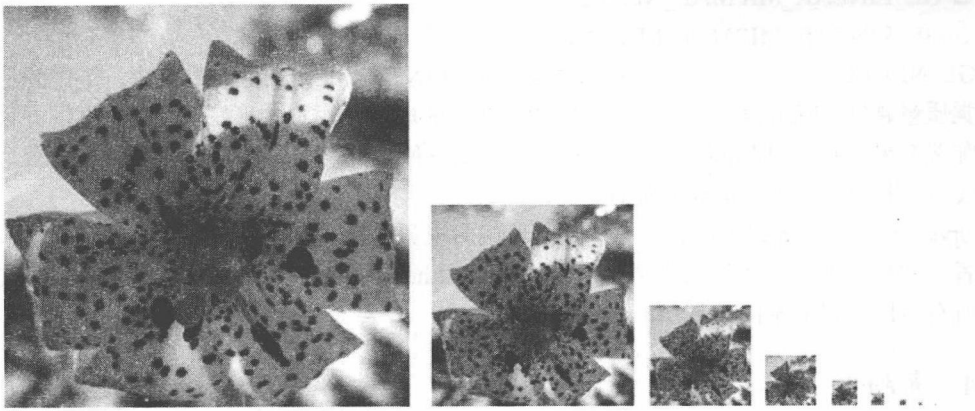


图8-6 从256×256到1×1的Mipmap

虽然Mipmap包会比原始图像多用掉33%的内存，但它不仅可以提升性能，也会提高图像质量。

从图8-6可以看出，每幅图像都是由原始图像派生出来的，差别仅在于细节度不同。显而易见，在1×1纹理版本中看到花朵是不可能的。

ARM Mali GPU Texture Compression Tool可以生成Mipmap。它并不只生成一个.pkm文件，此工具会为所有的细节层次（直到1×1）都生成一个.pkm文件。届时应用要逐一加载这些不同层次的文件，例如，通过调用`ETC1Util.loadTexture()`或`GLES20.glTexImage2D()`加载。

所有设备并不是等同的，它们不必需要有相同的细节层次。例如，分辨率1920×1080（HD分辨率）的Google TV就要比分辨率为800×480（WVGA）的三星Nexus S需要更高层次的细节。因此，Google TV需要256×256的纹理，Nexus S则可能永远不会用到这种纹理，对它来说128×128的纹理就够了。

注意 不要单纯依赖设备的分辨率，要根据应用的需要进行处理。例如，索尼的Google TV（1920×1080分辨率）使用400MHz的PowerVR SGX535，性能却不如三星Galaxy NEXUS（1280×720分辨率）用的384MHz的PowerVR SGX540。

更多资源请访问稀酷客(www.ckook.com)

纹理如何渲染也取决于选择的那个 glTexParameter()方法（比如 glTexParameteri()）设置的纹理参数。例如，可以调用 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, param) 设置缩小功能，param 可以取以下值：

- GL_NEAREST
- GL_LINEAR
- GL_NEAREST_MIPMAP_NEAREST
- GL_NEAREST_MIPMAP_LINEAR
- GL_LINEAR_MIPMAP_NEAREST
- GL_LINEAR_MIPMAP_LINEAR

GL_NEAREST 一般比 GL_LINEAR 快，GL_LINEAR 又比其他四个更快，越慢的函数得到的视觉质量越好。应用程序会选择什么样的参数可能取决于用户的喜好（假如你提供方法让用户自行配置渲染质量），但许多用户不会有耐心尝试各种配置，找出适合自己设备的组合。因此，应用要自己选出最优的 OpenGL 配置。

OpenGL 可以配置很多内容，一些默认值偏向视觉质量而非性能，因此，要熟悉那些应用能够配置的选项。例如，关于纹理方面，http://www.khronos.org/opengles/sdk/docs/man/的文档会告诉你所有可以设置的不同参数。

8.3.4　多APK

由于可能要支持多种纹理压缩格式（尽可能为每个设备都做优化），Mipmap 需要占用较大的空间，应用可能会超过 Android Market（目前为 50M 字节）定义的大小限制。

如果出现这种情况，基本上有三种选择：

- 减少应用的大小，例如只支持 ETC1 纹理压缩；
- 安装应用后从远程服务器下载纹理；
- 生成多个 APK，每个对应一套纹理。

Android Market 可以让应用发布多个 APK，每个针对不同的配置。例如，可以让一个 APK 只使用 ETC1 纹理，另一个用 PVRTC 纹理——即为使用 PowerVR 的 Android 设备优化。这些 APK 共享相同的 Android Market 列表，Android Market 会为每个设备仔细选择适合的 APK。用户不必担心下载并安装的 APK 是否正确，这一切是自动和透明的。

注意　*并非所有的 Android 应用商店都支持此功能，所以如果你打算分发应用到多个商店，还是尽可能用适合所有设备的单一 APK 吧。*

当然，纹理可能不是你需要发布多个 APK 的唯一原因。例如，你可能针对旧设备发布小的 APK，对较新的设备发布带有更多功能且更大的 APK。虽然可以使用多个 APK，但它会使发布流程和维护变得复杂，因此，建议你尽可能尝试发布单一 APK。

更多资源请访问稀酷客(www.ckook.com)

8.3.5 着色

OpenGL ES 2.0 支持 OpenGL ES 着色语言（Shading Language），以取代 OpenGL ES 1.x 中的固定变换函数和分散的管线（pipeline）。这种语言基于 C，可以让你编写自己的顶点（vertex）和片段（fragment）着色来控制 OpenGL 管线。

像 C 程序一样，着色器可以非常简单，也可极其复杂。虽然没有必须遵循的单一规则，但你应该尽可能减少着色器的复杂性，这会非常影响性能。

8.3.6 场景复杂性

显然，渲染复杂的场景比简单的要花的时间长。一个提高帧速率的方法是简化要渲染的场景，同时保持可接受的视觉质量。例如，对于可看到的纹理，较远的物体可以减少细节，使用更少的三角形。简单的对象使用的内存和带宽较少。

8.3.7 消隐

尽管 GPU 擅长几何运算，并能确定要渲染的物体，但应用应该尽力消隐（Culling）视线以外的物体，这样就不会向那些因为不可见而要丢弃的对象发送绘制命令。

消隐对象（甚至三角形）有许多方法，这些已超出本书的范围，帧速率低于预期可能是考虑不周的消隐方法造成的。例如，可以快速消除摄像机（camera）后面的物体。

注意 大多情况下你可以启用背面消隐（Backface Culling），这将不会渲染对象背面的三角形。

8.3.8 渲染模式

默认情况下，不管发生什么变化，OpenGL 都会持续渲染场景。在某些情况下，帧之间的场景可能不会改变，你可能想明确地告诉渲染器只在请求时渲染场景。这可以改变 `GLSurfaceView` 的渲染模式来实现。

可以使用以下两个值之一调用 `setRenderMode()`设置 `GLSurfaceView` 的渲染模式：

- RENDERMODE_CONTINUOUSLY
- RENDERMODE_WHEN_DIRTY

如果渲染模式设置为 RENDERMODE_WHEN_DIRTY，只有创建表面时，或当 `GLSurfaceView.requestRender()`被调用时才渲染场景。

8.3.9 功耗管理

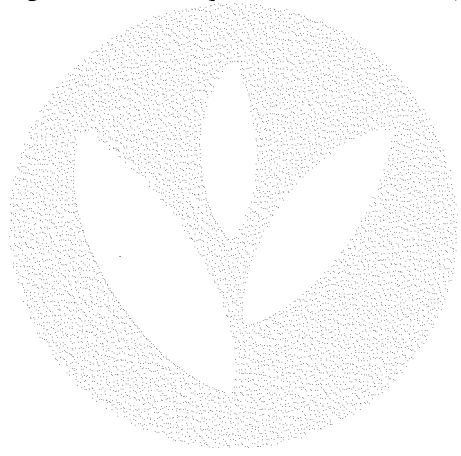
现代 GPU 有个很棒的特性，就是能够休眠，或至少在闲置期间降低频率。例如，GPU 在两帧之间如果有段时间无事可做，可以关闭（部分或全部），降低功耗，从而增加电池寿命。

更多资源请访问稀酷客(www.ckook.com)

即使应用达到可接受的帧率，还可以只为降低功耗进一步优化。渲染帧的速度越快，GPU就越早闲置，电池寿命会更长，用户使用你的程序时间就越长。对于某些应用，使用OpenGL时有个好的延长电池寿命方法，即只有当场景变化才渲染帧（如上面描述的 RENDERMODE_WHEN_DIRTY 渲染模式）。

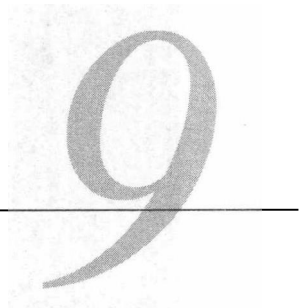
8.4 总结

最新的Android设备具有非常强大的硬件和2D/3D图形能力。虽然一些优化相对几年前来说已经变得不那么重要，但设备支持越来越高的分辨率，OpenGL ES 2.0的普遍支持，用户愈来愈高的期望值，都给我们带来不少新的挑战。本章只触及了 OpenGL ES 的皮毛，介绍了一些容易实现且快速见效的方法。幸运的是，无论对于初学者还是高手，学习OpenGL都有着无数的资源。还记得 GPU 厂商（ARM、Imagination Technologies、Nvidia、Qualcomm）提供的文档吗？那里有各家的GPU优化指南。



更多资源请访问稀酷客(www.ckook.com)

第 9 章 RenderScript



Honeycomb（API 等级 11）引入了 RenderScript，它是一个针对高性能 3D 渲染和计算操作的新框架。虽然 Android 2.1（Eclair）后来也引入了这个框架，但 Honeycomb 是第一个正式将 RenderScript 公之于众的版本。

本章先简要介绍 RenderScript，随后你会学习如何创建简单的脚本，并从 Java 应用调用它。然后，还要学习如何用脚本执行渲染操作，以及如何从 Java 访问脚本的数据。我们先观摩 Android SDK 提供的示例应用 HelloCompute，并比较同一应用的两个不同实现，一个基于 Honeycomb API，另一个基于 Ice Cream Sandwich（Android 4.0）API。本章结尾处我们会浏览一下 Honeycomb 和 Ice Cream Sandwich 的 RenderScript 头文件，以及在脚本中可用的头文件。

注意 本章不是完整的 3D 渲染指南。例如，要想用 RenderScript 的高级功能，得先学习 OpenGL 着色语言（GLSL）。有专门讲述 OpenGL 和 GLSL 的书，掌握 3D 渲染需要多年实践。

9.1 概览

目前使用 RenderScript 的应用并不多见。一图胜千言，让我们先看看使用 RenderScript 的例子。也许最流行的例子莫过于 YouTube 应用及其传送带视图，如图 9-1 所示。为了排除干扰，特意选择了视频尚未加载时的截图，你可以看到每个矩形的样子（每个矩形会包含一个视频缩略图）。另一个例子是 Balls，Android SDK 中的示例代码，使用传感器信息计算屏幕上球的位置，如图 9-2 所示。

注意 要利用示例代码（例如，Balls）创建项目，可以在 Eclipse 中创建新的 Android 项目，并选择“Create project from existing sample”，或创建一个新的 Android 示例项目（新版本的 Eclipse ADT 插件加入的可选项）。RenderScript 示例仅在选择 3.0 或以上为目标平台时可用。可以从 http://code.google.com/p/android-ui-utils/downloads/list 下载传送带示例。

更多资源请访问稀酷客(www.ckook.com)

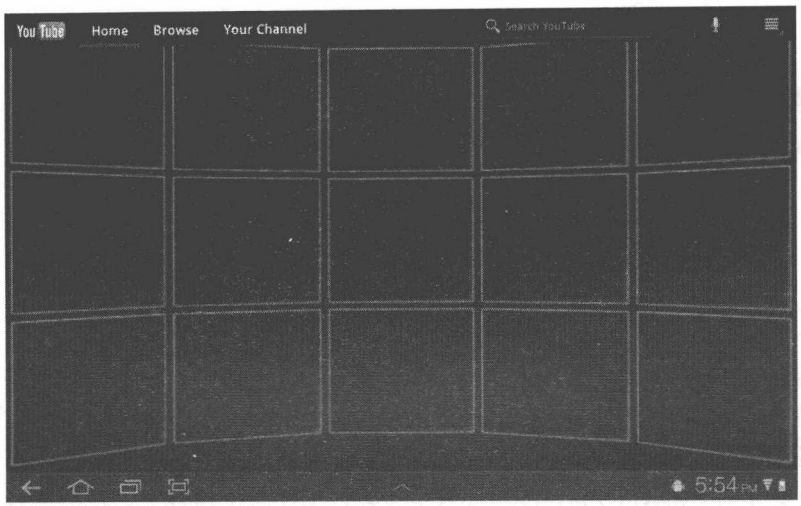


图 9-1　YouTube 的传送带视图

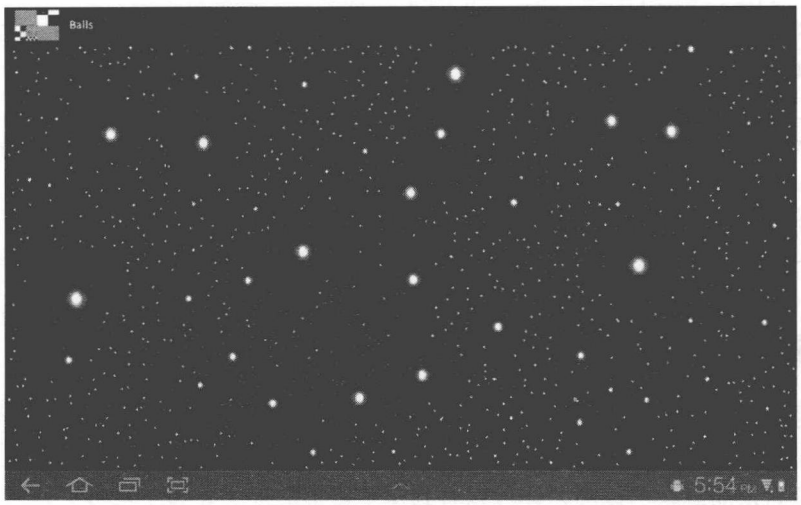


图 9-2　Balls 程序示例

RenderScript 使用 C99 作为开发语言，但并不意味着它会完全取代 Java 应用程序。相反，应用会在必要部分使用 RenderScript， 其他部分用 Java。使用 RenderScript 时，脚本会在主机上被编译为 Low-Level Virtual Machine （ LLVM ）的位码（ 例如， 安装 Eclipse 的 Windows PC ），接着 LLVM 的位码（ bitcode ）将会在 Android 设备上编译为机器码。从这种角度说，它类似在主机上把 Java 代码编译为 Dalvik 字节码，Dalvik 的 JIT 编译器把字节码编译为机器码。

机器码缓存在设备上，因为已编译为机器码，所以脚本后续执行会更快。由于将位码编译为机器码的步骤是在 Android 设备上，机器码可以为设备进行特别优化，甚至包括硬件模块（ 例如，

更多资源请访问稀酷客(www.ckook.com)

GPU），作为开发者并不用操心各种架构和功能。这是与 NDK 的一个主要区别（例如，NDK 的代码中还要检查 NEON 是否可用），这样会大大简化工作流程。

注意 如需更多关于 LLVM 的信息，请访问http://llvm.org，下载并安装各种 LLVM 的工具。LLVM 不是 Android 的专有技术。事实上，苹果的 Xcode 4 也在用它。

在深入 RenderScript 细节之前，我们要先演练一下如何在项目中创建基本的 RenderScript 脚本。由于模拟器不支持 RenderScript 所需的所有功能，你需要真实的 Android 设备来运行 RenderScript。你会经常发现 RenderScript 脚本简称为 RenderScript。

9.2 Hello World

一如既往，每当介绍新框架，第一反应就是用它写“Hello World”，我们用 RenderScript 脚本来写一下。只需在 Android 项目创建文件，命名为 helloworld.rs。代码清单 9-1 演示了如何写个简单的函数，使用 RenderScript 的调试函数输出“Hellow, World”字符串。

代码清单 9-1 “Hello, World”脚本

```
#pragma version(1)
#pragma rs java_package_name(com.apress.proandroid.ch9)

void hello_world() {
    rsDebug("Hello, World", 0); // 没有单独字符串的 API，所以要加 0 来发送给调试输出
}
```

第一行声明了脚本使用的 RenderScript 版本。第二行声明脚本的 Java 反射包名。编译 Android 项目时，系统背后会做很多事情，构建工具会创建以下 3 个文件：

- ScriptC_helloworld.java（在 gen/ 目录）
- helloworld.d（在 gen/ 目录）
- helloworld.bc（在 res/raw/ 目录）

代码清单 9-2 是自动生成的 ScriptC_helloworld.java 文件的内容。你可以在项目的 gen 目录找到这个文件，和自动生成的 R.java 文件在同一个目录。这个文件是脚本反射层的一部分，其中定义的类让你可以调用脚本的函数，与脚本交互。同一目录中，还会找到 helloworld.d，它只是列出 helloworld.bc 文件的依赖关系（也就是说，如果修改任何列在 helloworld.d 依赖的文件，helloworld.bc 会重新生成）。

代码清单 9-2 ScriptC_helloworld.java

```
/*
 * Copyright (C) 2011 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
```


更多资源请访问稀酷客(www.ckook.com)

```
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * This file is auto-generated. DO NOT MODIFY!
 * The source RenderScript file: E:\ws\Chapter 9\src\com\apress\proandroid\ch9\helloworld.rs
 */
package com.apress.proandroid.ch9;

import android.renderscript.*;
import android.content.res.Resources;
/**
 * @hide
 */
public class ScriptC_helloworld extends ScriptC {
    // Constructor
    public  ScriptC_helloworld(RenderScript rs, Resources resources, int id) {
        super(rs, resources, id);
    }

    private final static int mExportFuncIdx_hello_world = 0;
    public void invoke_hello_world() {
        invoke(mExportFuncIdx_hello_world);
    }

}
```

在这个自动生成的文件中要注意三个地方：

- 包名在脚本中是用#pragma 定义的；
- 公共构造函数；
- invoke_hello_world 函数。

这是自动生成文件的一部分，尽管它们很重要，也用不着我们操心。例如，代码清单 9-2 中的私有常量 mExportFuncIdx_hello_world 是脚本中函数的索引，但我们不需知道它的实际值。

要用脚本，必须在 Java 代码中创建 ScriptC_helloworld 实例，这样才能通过调用含反射的方法 invoke_hello_world，调用脚本的 hello_world 函数（参见代码清单 9-3）。

代码清单 9-3　调用 RenderScript 函数

```
public class Chapter9Activity extends Activity {
    /** 当 Activity 第一次创建时调用。*/
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
```

更多资源请访问稀酷客(www.ckook.com)

```
        setContentView(R.layout.main);

        HelloWorldRenderScript();
    }

    private void HelloWorldRenderScript() {
        RenderScript rs = RenderScript.create(this); // 需要 Context 参数

        // 用 res/raw/helloworld.bc 的 helloworld 位码创建脚本
        ScriptC_helloworld helloworldScript = new ScriptC_helloworld(rs, getResources(),
            R.raw.helloworld);

        // 使用含反射的方法调用脚本的 hello_world 函数
        helloworldScript.invoke_hello_world();
    }
}
```

执行此代码会在 Logcat 中输出带有“RenderScript”标签的“Hello, World 0 0x0”。

注意 Eclipse 中使用 DDMS 视图会显示 RenderScript 相关的线程运行在应用中（名为 RSMessageThread）。你还可以看到 Logcat 中有如下输出：“RS Launching thread(s), reported CPU count 2”（如果设备是双核）。学习 RenderScript 时，注意观察“RenderScript”产生的输出。

虽然 helloworld.bc 文件至关重要，因为它包含了脚本的位码，但它的实际内容对你没什么意义。事实上 LLVM 的位码是与脚本平台无关的中间层表示，这个文件也可以反汇编，使用 llvm-dis（LLVM 套件的一部分）可将其转换为可读的 LLVM 汇编语言。代码清单 9-4 是 helloworld.bc 的 LLVM 汇编语言版本。

代码清单 9-4 LLVM 汇编语言（helloworld.bc）

```
; ModuleID = '/home/herve/helloworld.bc'
target datalayout = "e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f32:32:32-f64:64:
64-v64:64:64-v128:128:128-a0:0:64-n32"
target triple = "armv7-none-linux-gnueabi"

@.str = private constant [13 x i8] c"Hello, World\00"

define void @hello_world() nounwind {
  tail call void @_Z7rsDebugPKci(i8* getelementptr inbounds ([13 x i8]* @.str, i32 0, i32 0), i32 0)
nounwind
  ret void
}

declare void @_Z7rsDebugPKci(i8*, i32)

!#pragma = !{!0, !1}
!#rs_export_func = !{!2}
```

更多资源请访问稀酷客(www.ckook.com)

```
!0 = metadata !{metadata !"version", metadata !"1"}
!1 = metadata !{metadata !"java_package_name", metadata !"com.apress.proandroid.ch9"}
!2 = metadata !{metadata !"hello_world"}
```

这个例子没用 RenderScript 做任何渲染。事实上，Activity 依然使用 XML 定义，Activity 的内容仍然是通过调用 `setContentView()`展开布局资源并设置的。在知道了如何创建基本的脚本后，下面看看 RenderScript 如何进行渲染。

9.3 Hello Rendering

RenderScript 渲染有点复杂，不能简单地调用脚本，在向屏幕显示之前要做一些准备工作，如下所述：

- 创建进行渲染的脚本；
- 创建 RenderScriptGL 上下文对象；
- 创建继承 RSSurfaceView 的类；
- 将 Activity 的内容视图设为 RSSurfaceView。

9.3.1 创建渲染脚本

我们的第一个渲染脚本非常简单，它只用一些随机颜色改变背景色。脚本如代码清单 9-5 所示，在 hellorendering.rs 文件中。RenderScript 文件名字很重要，因为它们也同时定义了资源名字（如 R.raw.hellorendering，它是 R.java 文件中定义的一个整数）。

代码清单 9-5　更改背景颜色

```
#pragma version(1)
#pragma rs java_package_name(com.apress.proandroid.ch9)

#include "rs_graphics.rsh"

// 脚本实例创建时自动调用
void init() {
    // 做任何想做的事情
}

int root() {
    float red = rsRand(1.0f);
    float green = rsRand(1.0f);
    float blue = rsRand(1.0f);

    // 清除随机颜色的背景色
    rsgClearColor(red, green, blue, 1.0f); // alpha 为 1.0f，即不透明

    // 每秒 50 帧=每帧 20 毫秒
    return 20;
}
```

更多资源请访问稀酷客(www.ckook.com)

前两行与代码清单 9-1 相同。接下来一行包含 rs_graphics.rsh 头文件，其中定义了 rsgClearColor()。除了 rs_graphics.rsh 外，其他头文件是自动包含的，所以当使用 RenderScript 图形函数时（如 rsgClearColor()或 rsgBindFont()），要显式包含 rs_graphics.rsh 头文件。

init()放在这里只做提示之用，它的实现是空的。init()函数是可选的，如果有的话，创建脚本实例时会自动调用。它被用来在其他子程序执行之前进行初始化。即使不使用渲染的脚本（如代码清单 9-1），也可以加入 init()函数。

root()函数做了渲染工作。这里的实现很简单，只是用随机颜色清除背景。有趣的是，函数的返回值指定了渲染的频率。这个例子中，返回值是 20，表示 20 毫秒刷新一次（即帧率为每秒 50 帧）。

注意 init()和 root()函数是保留函数，并不需要在反射层（此例中为 ScriptC_hellorendering.java）创建 invoke_init()和 invoke_root()方法。

9.3.2 创建RenderScriptGL Context

现在渲染脚本完成了，接下来需要创建 RenderScriptGL 的 Context 对象。这里 RenderScriptGL 的 Context 对象在调用 RSSurfaceView 的 surfaceChanged()方法时创建。HelloRenderingView 实现如代码清单 9-6 所示。

代码清单 9-6 HelloRenderingView.java

```
public class HelloRenderingView extends RSSurfaceView {

    public HelloRenderingView(Context context) {
        super(context);
    }

    private RenderScriptGL mRS;
    private HelloRenderingRS mRender; // 辅助类

    public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
        super.surfaceChanged(holder, format, w, h);
        if (mRS == null) {
            RenderScriptGL.SurfaceConfig sc = new RenderScriptGL.SurfaceConfig();
            mRS = createRenderScriptGL(sc);
            mRender = new HelloRenderingRS();
            mRender.init(mRS, getResources());
        }
        mRS.setSurface(holder, w, h);
}

    @Override
    protected void onDetachedFromWindow() {
        if(mRS != null) {
```

更多资源请访问稀酷客(www.ckook.com)

```
            mRS = null;
            destroyRenderScriptGL();
        }
    }
}
```

代码清单 9-6 中用到了 HelloRenderingRS 辅助类。

9.3.3 展开RSSurfaceView

辅助类的实现如代码清单 9-7 所示。

代码清单 9-7　HelloRenderingRS.java

```
public class HelloRenderingRS {

    public HelloRenderingRS() {
    }

    private Resources mRes;
    private RenderScriptGL mRS;
    private ScriptC_hellorendering mScript;

    public void init(RenderScriptGL rs, Resources res) {
        mRS = rs;
        mRes = res;

        mScript = new ScriptC_hellorendering(mRS, mRes, R.raw.hellorendering);

        mRS.bindRootScript(mScript);
    }
}
```

在这个类中，我们创建真正的 ScriptC_hellorendering 脚本实例，将它绑定到 RenderScriptGL 对象上。不调用 bindRootScript()，不会做任何渲染，因为脚本实例的 root()方法根本不会被调用到。想看渲染的成果却只见到黑色屏幕时，先看一下是否忘记调用了 bindRootScript()。

9.3.4 设置内容视图

现在所需文件都有了，接下来可以创建 HelloRenderingView 实例，并设置 Activity 的内容为该实例，如代码清单 9-8 所示。

代码清单 9-8　Activity

```
public class Chapter9Activity extends Activity {
    static final String TAG = "Chapter9Activity";

    private HelloRenderingView mHelloRenderingView;

    /** 在 Activity 第一次创建时调用。*/
    @Override
```

更多资源请访问稀酷客(www.ckook.com)

```
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        //setContentView(R.layout.main); // 不需要 XML 布局，这里把它注释掉

        HelloWorldRenderScript();

        mHelloRenderingView = new HelloRenderingView(this);
        setContentView(mHelloRenderingView);
    }

    @Override
    protected void onPause() {
        super.onPause();
        mHelloRenderingView.pause(); // 暂停渲染线程
    }

    @Override
    protected void onResume() {
        super.onResume();
        mHelloRenderingView.resume(); // 恢复渲染线程
    }

    private void HelloWorldRenderScript() {
        // 实现参见代码清单 9-3
    }
}
```

文件大部分内容对你来说已是小菜一碟，但这个文件中有个值得注意的地方：Activity 暂停和恢复时，必须通知 RSSurfaceView。如没有调用 RSSurfaceView.pause()和 RSSurfaceView.resume()，渲染线程不会知晓活动的状态，即使 Activity 不可见也会继续进行渲染。

Android 4.0 添加了两个重要的新功能：

- 当创建 Allocation 对象时，应用可以设置 Allocation.USAGE_GRAPHICS_RENDER_TARGET 标志使用屏幕外的缓冲区。
- RSTextureView 可以用在自己的布局中（和其他视图一起工作）。RSSurfaceView 会创建单独的窗口，但 RSTextureView 不会。

9.4 在脚本中添加变量

下面修改一下代码清单 9-5，加深对反射层的理解。到目前为止，我们只看到了一个反射的子程序，代码清单 9-2 的 hello_world。这一节，我们创建新的脚本 hellorendering2.rs，加入更多变量，如代码清单 9-9 所示。

代码清单 9-9 改变背景颜色（第 2 版）

```
#pragma version(1)
#pragma rs java_package_name(com.apress.proandroid.ch9)

#include "rs_graphics.rsh"
```


更多资源请访问稀酷客(www.ckook.com)

```
// init()是空的，这回去掉了

float red = 0.0f; // 初始化为 1.0f

float green; // 故意不初始化

static float blue; // 故意不初始化，静态变量

const float alpha = 1.0f; // 常数

int root() {
    // 清除背景颜色
    blue = rsRand(1.0f);
    rsClearColor(red, green, blue, alpha);

    // 每秒 50 帧 = 每帧 20 毫秒
    return 20;
}
```

正如所见，4 个变量 red、green、blue、alpha 都有不同的定义。虽然在真实脚本中这样做并没什么意义，但可以让我们了解反射层是如何创建的。代码清单 9-10 是 ScriptC_hellorendering2.java 中的类。

代码清单 9-10　ScriptC_hellorendering2.java

```
/*
 * Copyright (C) 2011 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

/*
 * This file is auto-generated. DO NOT MODIFY!
 * The source RenderScript file: E:\ws\Chapter 9\src\com\apress\proandroid\ch9\hellorendering2.rs
 */
package com.apress.proandroid.ch9;

import android.renderscript.*;
import android.content.res.Resources;

/ **
 * @hide
 */
```

更多资源请访问稀酷客(www.ckook.com)

```
public class ScriptC_hellorendering2 extends ScriptC {
    // Constructor
    public  ScriptC_hellorendering2(RenderScript rs, Resources resources, int id) {
        super(rs, resources, id);
        mExportVar_red = 0f;
        mExportVar_alpha = 1f;
    }
    private final static int mExportVarIdx_red = 0;
    private float mExportVar_red;
    public void set_red(float v) {
        mExportVar_red = v;
        setVar(mExportVarIdx_red, v);
    }

    public float get_red() {
        return mExportVar_red;
    }

    private final static int mExportVarIdx_green = 1;
    private float mExportVar_green;
    public void set_green(float v) {
        mExportVar_green = v;
        setVar(mExportVarIdx_green, v);
    }

    public float get_green() {
        return mExportVar_green;
    }

    private final static int mExportVarIdx_alpha = 2;
    private float mExportVar_alpha;
    public float get_alpha() {
        return mExportVar_alpha;
    }
}
```

全局变量 red 在脚本中定义并初始化为零。因此，反射层定义了私有的 mExportVar_red 浮点类型成员，在构造函数中初始化为零，以及设置和获取值的方法：set_red()和 get_red()。

全局变量 green 与 red 十分相似，但它没有被初始化。反射层同样定义了私有的 mExportVar_green 浮点类型成员，以及设置和获取值的方法：set_green()和 get_green()。不过 mExportVar_green 成员并未在构造函数中初始化。

blue 被定义为静态变量，这意味着它不能开放给其他脚本访问。因此，反射层没有针对脚本中的 blue 定义任何成员变量和方法。

最后，alpha 定义为常量，因此反射层在构造函数中对 mExportVar_alpha 成员变量进行初始化，只定义了获取值的方法，get_alpha()。由于 alpha 不变，确实没有必要在 Java 中定义 set_alpha()。

注意 脚本中定义的全局指针会在反射层生成 bind_pointer_name()方法而不是 set_pointer_name()。

更多资源请访问稀酷客(www.ckook.com)

你可以看到，get()方法返回的值为 Java 中最后一次设置的值。例如，get_green()只返回 mExportVar_green，它只能通过调用 set_green()修改。这基本上意味着，如果脚本修改了全局变量 green，这种改变不会通知到 Java 层。这是一个非常值得注意的细节。

现在，我们观摩一下 Android 提供的简单示例代码，来深入了解 RenderScript 的一个很重要的功能。

9.5 HelloCompute

Android 的简单示例程序 HelloCompute 会把彩色图片转为黑白。代码清单 9-11 是其脚本实现（目标平台是 Honeycomb）。

代码清单 9-11　RenderScript mono.rs

```
/*
 * Copyright (C) 2011 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#pragma version(1)
#pragma rs java_package_name(com.android.example.hellocompute)

rs_allocation gIn;
rs_allocation gOut;
rs_script gScript;

const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

void root(const uchar4 *v_in, uchar4 *v_out, const void *usrData, uint32_t x, uint32_t y) {
    float4 f4 = rsUnpackColor8888(*v_in);

    float3 mono = dot(f4.rgb, gMonoMult);

    *v_out = rsPackColorTo8888(mono);
}

void filter() {
    rsForEach(gScript, gIn, gOut, 0); // first: script; second: input allocation; third: output
allocation
}
```

更多资源请访问稀酷客(www.ckook.com)

root()函数把单个像素转为黑白像素。执行以下 4 个步骤：

(1) 将 ARGB 值（* v_in）转换为浮点矢量；

(2) root()函数求 f4.rgb 和 gMonoMult 的点积，返回单个浮点值（亮度）；

(3) 在 mono 矢量中，三个分量是相同的值（点积的结果就是亮度）；

(4) mono 矢量转换为 ARGB 值，此值被复制到* v_out。

注意 gMonoMult 值在 NTSC 标准中定义。

为了让脚本把整幅图像转为黑白，它必须对每个像素点执行一遍 root()函数。这就是 filter()函数调用 rsForEach()做的事情。rsForEach()函数迭代每组输入和输出（第二个和第三个参数）执行脚本（第一个参数）。第四个参数 usrData，这里设置为零，root()函数用不到它。

9.5.1 Allocation

rsForEach()用 rs_allocation 作为参数，它们必须在 Java 中创建，而后传入脚本中。这就是 HelloCompute.java 中的 createScript()除其他事情外要做的工作，如代码清单 9-12 所示。

代码清单 9-12 Allocation

```
private RenderScript mRS;
private Allocation mInAllocation; // 输入分配
private Allocation mOutAllocation; // 输出分配
private ScriptC_mono mScript;

...

private void createScript() {
    mRS = RenderScript.create(this);

    mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn,
                                                Allocation.MipmapControl.MIPMAP_NONE,
                                                Allocation.USAGE_SCRIPT);

    mOutAllocation = Allocation.createTyped(mRS, mInAllocation.getType());

    mScript = new ScriptC_mono(mRS, getResources(), R.raw.mono);
    mScript.set_gIn(mInAllocation);
    mScript.set_gOut(mOutAllocation);
    mScript.set_gScript(mScript);

    mScript.invoke_filter();

    mOutAllocation.copyTo(mBitmapOutRS);
}
```

此例中，根据位图创建输入内存分配，供脚本使用。其他可能用到的包括纹理源（USAGE_

更多资源请访问稀酷客(www.ckook.com)

GRAPHICS _TEXTURE）和图形网格（USAGE_GRAPHICS_VERTEX）。你可以查看 Allocation 类中为创建 Allocation 定义的各种方法。Allocation 可以是一维（例如，数组）、二维（例如，位图）或三维。

9.5.2 rsForEach

rsForEach()函数有 3 种不同用法：

- rsForEach(rs_script script, rs_allocation input, rs_allocation output) (Android 4.0 及以上)
- rsForEach(rs_script script, rs_allocation input, rs_allocation output, const void * usrData)
- rsForEach(rs_script script, rs_allocation input, rs_allocation output, const void * usrData, const rs_script_call_t*)

从代码清单 9-11 中可以看到，第一个参数是脚本。这里会设置要执行哪个 root()函数。第二个和第三个参数指定输入和输出分配。第四个参数（如果有），是用户在脚本中传递的私有数据。Android 不会使用此值，root()函数用不到时可以设置为 0 或任何值。第五个参数（如果有），会指定 root()函数的执行方式。代码清单 9-13 是 rs_script_call_t 结构的定义。

代码清单 9-13　rs_script_call_t

```
enum rs_for_each_strategy {
    RS_FOR_EACH_STRATEGY_SERIAL,
    RS_FOR_EACH_STRATEGY_DONT_CARE,
    RS_FOR_EACH_STRATEGY_DST_LINEAR,
    RS_FOR_EACH_STRATEGY_TILE_SMALL,
    RS_FOR_EACH_STRATEGY_TILE_MEDIUM,
    RS_FOR_EACH_STRATEGY_TILE_LARGE
};

typedef struct rs_script_call {
    enum rs_for_each_strategy strategy;
    uint32_t xStart;
    uint32_t xEnd;
    uint32_t yStart;
    uint32_t yEnd;
    uint32_t zStart;
    uint32_t zEnd;
    uint32_t arrayStart;
    uint32_t arrayEnd;
} rs_script_call_t;
```

你可以配置策略，以及开始和结束参数（适用于所有维）。策略只是个建议，不能保证实现会遵从。

为方便起见，Android 4.0 （Ice Cream Sandwich）定义了 Script.forEach()。不要在代码中直接调用此方法，它只是在反射层中使用。拜新方法所赐，mono.rs 脚本可以简化，代码清单 9-14 是当构建目标设置为 Android 4.0 时脚本的实现。

更多资源请访问稀酷客(www.ckook.com)

代码清单 9-14 RenderScript mono.rs（Android 4.0 版本）

```
/*
 * Copyright (C) 2011 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

#pragma version(1)
#pragma rs java_package_name(com.example.android.rs.hellocompute)

const static float3 gMonoMult = {0.299f, 0.587f, 0.114f};

void root(const uchar4 *v_in, uchar4 *v_out) {
    float4 f4 = rsUnpackColor8888(*v_in);

    float3 mono = dot(f4.rgb, gMonoMult);
    *v_out = rsPackColorTo8888(mono);
}
```

正如所见，filter()函数没有了，gIn、gOut、gScript 也消失了。要想知道为什么 Android 4.0 的示例程序中可以去掉这个函数，你可以到项目构建时自动生成的反射层代码中看看。代码清单 9-15 是新的反射层。

代码清单 9-15 ScriptC_mono.java（Android 4.0 版本）

```
/*
 * Copyright (C) 2011 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
```


更多资源请访问稀酷客(www.ckook.com)

```
/*
 * This file is auto-generated. DO NOT MODIFY!
 * The source Renderscript file:
E:\ws\HelloCompute\src\com\example\android\rs\hellocompute\mono.rs
 */
package com.example.android.rs.hellocompute;

import android.renderscript.*;
import android.content.res.Resources;

/**
 * @hide
 */
public class ScriptC_mono extends ScriptC {
    // Constructor
    public  ScriptC_mono(RenderScript rs, Resources resources, int id) {
        super(rs, resources, id);
        __U8_4 = Element.U8_4(rs);
    }

    private Element __U8_4;
    private final static int mExportForEachIdx_root = 0;
    public void forEach_root(Allocation ain, Allocation aout) {
        // check ain
        if (!ain.getType().getElement().isCompatible(__U8_4)) {
            throw new RSRuntimeException("Type mismatch with U8_4!");
        }
        // check aout
        if (!aout.getType().getElement().isCompatible(__U8_4)) {
            throw new RSRuntimeException("Type mismatch with U8_4!");
        }
        // Verify dimensions
        Type tIn = ain.getType();
        Type tOut = aout.getType();
        if ((tIn.getCount() != tOut.getCount()) ||
            (tIn.getX() != tOut.getX()) ||
            (tIn.getY() != tOut.getY()) ||
            (tIn.getZ() != tOut.getZ()) ||
            (tIn.hasFaces() != tOut.hasFaces()) ||
            (tIn.hasMipmaps() != tOut.hasMipmaps())) {
            throw new RSRuntimeException("Dimension mismatch between input and output parameters!");
        }
        forEach(mExportForEachIdx_root, ain, aout, null);
    }
}
```

显然，invoke_filter()方法不会出现在反射层了，因为脚本中已经没有 filter()了。因为 gIn、gOut、gScript 变量去掉了，反射层也就没有了对应的 set / get 方法了。现在的关键方法是 forEach_root()，在 HelloCompute.java 的 createScript()方法中调用，如代码清单 9-16（Honeycomb 版本见代码清单 9-12）所示。

更多资源请访问稀酷客(www.ckook.com)

代码清单 9-16 createScript()方法（Android 4.0 版本）

```
private void createScript() {
    mRS = RenderScript.create(this);

    mInAllocation = Allocation.createFromBitmap(mRS, mBitmapIn,
                                                Allocation.MipmapControl.MIPMAP_NONE,
                                                Allocation.USAGE_SCRIPT);

    mOutAllocation = Allocation.createTyped(mRS, mInAllocation.getType());

    mScript = new ScriptC_mono(mRS, getResources(), R.raw.mono);

    mScript.forEach_root(mInAllocation, mOutAllocation);

    mOutAllocation.copyTo(mBitmapOut);
}
```

注意 Allocation 必须互相兼容（例如，所有维度都必须是相同大小）。

你可以看到，createScript()现在可以从反射层调用 forEach_root()方法了。因为 Honeycomb 没有 Script.forEach()方法，你可能希望避免使用该方法，继续使用原来 Honeycomb 的方式。虽然它麻烦些，但可以让应用兼容 Android 3.x 设备。

9.5.3 性能

既然 RenderScript 为高性能 3D 渲染和计算操作而生，少不了要把脚本和相应的 Java 实现做比较，看看到底 RenderScript 提升了多少。为此，Java 版本的黑白过滤器实现如代码清单 9-17 所示。

代码清单 9-17 Java 过滤器

```
// mBitmapIn 是输入的位图，mBitmapOut 是输出的位图

int w = mBitmapIn.getWidth();
int h = mBitmapIn.getHeight();
int size = w * h;
int[] pixels = new int[size];

mBitmapIn.getPixels(pixels, 0, w, 0, 0, w, h); // 获得所有像素（都是 32 位整数）

for (int i = 0; i < size; i++) {
    int c = pixels[i]; // 0xAARRGGBB

    // 提取红色，绿色和蓝色分量（都在[0, 255]范围内）
    int r = (c >> 16) & 0xFF;
    int g = (c >> 8)  & 0xFF;
    int b =  c        & 0xFF;

    // 使用整数的近似计算公式
```


更多资源请访问稀酷客(www.ckook.com)

```
        r *= 76;
        g *= 151;
        b *= 29;
        int y = (r + g + b) >> 8; // 亮度

        pixels[i] = y | (y << 8) | (y << 16) | ( c & 0xFF000000);
    }

    mBitmapOut.setPixels(pixels, 0, w, 0, 0, w, h); // 设置输出像素
```

过滤器的 RenderScript 版本在 Galaxy Tab 10.1 上用时约 115 毫秒（使用示例中的位图资源），Java 版本只用了 48 毫秒！无 JIT 版本（在应用 manifest 中设置 `android:vmSafeMode="true"`）花了大约 130 毫秒的时间。结果可能听起来令人惊讶，但你要考虑到创建脚本、分配资源和浮点运算的开销。较为复杂的脚本性能可能会优于其 Java 版本，并会受益于未来更强大的处理器。例如，未来脚本可以运行在 GPU 上，而不是 CPU，会更好地利用并行执行的优势。并非所有脚本都慢，但这个例子说明，使用 RenderScript 可能并不总会让代码更快。

9.6　自带的 RenderScript API

RenderScript 代码不能调用 NDK 或 C 库中的 API，因此只有有限的 API。RenderScript API 定义了 6 个头文件，位于 SDK 的 platform/android-xx/renderscript/include（其中 xx 是 API 等级，如 13）[①]：

- rs_types.rsh
- rs_core.rsh
- rs_cl.rsh
- rs_math.rsh
- rs_graphics.rsh
- rs_time.rsh

在 Ice Cream Sandwich 中，RenderScript API 定义在 12 个文件中. 除了上述 6 个，又加入以下 6 个文件：

- rs_allocation.rsh
- rs_atomic.rsh
- rs_debug.rsh
- rs_matrix.rsh
- rs_object.rsh
- rs_quaternion.rsh

再说下，RenderScript 的在线文档目前相当缺乏，因此建议你直接查看这些头文件。Ice Cream Sandwich 和 Honeycomb 的一个重大区别就是头文件中加入了很多注释。因此，要尽量参考 Android 4.0 RenderScript 的头文件，那里的信息比 Honeycomb 多得多。

① 最新版本又有变化，只有一个 platform-tools/renderscript/include/目录。——译者注

更多资源请访问稀酷客(www.ckook.com)

9.6.1 rs_types.rsh

这个头文件定义了 RenderScript 的基本类型（除了 C99 标准类型 char、short、int、long、float）：

- int8_t
- int16_t
- int32_t
- int64_t
- uint8_t
- uint16_t
- uint32_t
- uint64_t
- uchar （即 uint8_t）
- ushort （即 uint16_t）
- uint （即 uint32_t）
- ulong （即 uint64_t）

除了这些基本类型，这个文件还定义了向量类型：

- float2
- float3
- float4
- double2 （Android 4.0 及以上）
- double3 （Android 4.0 及以上）
- double4 （Android 4.0 及以上）
- char2
- char3
- char4
- uchar2
- uchar3
- uchar4
- short2
- short3
- short4
- ushort2
- ushort3
- ushort4
- int2

更多资源请访问稀酷客(www.ckook.com)

- int3
- int4
- uint2
- uint3
- uint4

没有矩阵的3D框架是不完整的，以下是定义的矩阵：

- rs_matrix2x2（2阶方阵）
- rs_matrix3x3（3阶方阵）
- rs_matrix4x4（4阶方阵）

注意　虽然浮点和整数向量类型都有定义（例如，float2、ushort3、int4），但矩阵类型只用浮点。

Honeycomb MR1（API等级12）引入了四元数类型rs_quaternion（用float4定义）。同时增加了新API去操作新类型。

正如所见，许多类型的层次相对高一些，不过这些类型都可以看成是标量、向量、矩阵。

rs_types.rsh中定义了以下类型，对它们的Java部分来说是不可见的句柄（handle）：

- rs_element
- rs_type
- rs_allocation
- rs_sampler
- rs_script
- rs_mesh
- rs_program_fragment
- rs_program_vertex
- rs_program_raster
- rs_program_store
- rs_font

以下这些类型也暴露在Java层的android.renderscript包中：

- Byte2
- Byte3
- Byte4
- Double2（Android 4.0及以上）
- Double3（Android 4.0及以上）
- Double4（Android 4.0及以上）
- Float2

更多资源请访问稀酷客(www.ckook.com)

- Float3
- Float4
- Int2
- Int3
- Int4
- Long2
- Long3
- Long4
- Matrix2f
- Matrix3f
- Matrix4f
- Short2
- Short3
- Short4
- Element
- Type
- Allocation
- Sampler
- Script
- Mesh
- ProgramFragment
- ProgramVertex
- ProgramRaster
- ProgramStore
- Font

9.6.2 rs_core.rsh

这个头文件在 Honeycomb 中定义了如下函数：

- rsDebug
- rsPackColorTo8888
- rsUnpackColor8888
- rsMatrixSet
- rsMatrixGet
- rsMatrixLoadIdentity
- rsMatrixLoad
- rsMatrixLoadRotate

更多资源请访问稀酷客(www.ckook.com)

- rsMatrixLoadScale
- rsMatrixLoadTranslate
- rsMatrixLoadMultiply
- rsMatrixMultiply
- rsMatrixRotate
- rsMatrixScale
- rsMatrixTranslate
- rsMatrixLoadOrtho
- rsMatrixLoadFrustum
- rsMatrixLoadPerspective
- rsMatrixInverse
- rsMatrixInverseTranspose
- rsMatrixTranspose
- rsClamp

矩阵函数在 Android 4.0 中移到专用的头文件 rs_matrix.rsh 中。

Honeycomb MR1（Android 3.1）引入了四元数类型，rs_core.rsh 加入了新函数：

- rsQuaternionSet
- rsQuaternionMultiply
- rsQuaternionAdd
- rsQuaternionLoadRotateUnit
- rsQuaternionLoadRotate
- rsQuaternionConjugate
- rsQuaternionDot
- rsQuaternionNormalize
- rsQuaternionSlerp
- rsQuaternionGetMatrixUnit

Android 4.0 中，这个头文件又有了很大的变化。事实上，这个文件的 Honeycomb 版本是个大杂烩，包含了矩阵、四元数、调试以及一些数学函数。Android 4.0 中此文件被大幅删改，只定义了以下函数：

- rsSendToClient
- rsSendToClientBlocking
- rsForEach

矩阵、四元数和调试功能被转移到相应的新头文件（rs_matrix.rsh、rs_quaternion.rsh、rs_debug.rsh）中，而数学函数转移到 rs_math.rsh。由于头文件有自动包含功能，从一个头文件移动到另一个，应该不会影响你的脚本。

更多资源请访问稀酷客(www.ckook.com)

9.6.3 rs_cl.rsh

这个头文件，读起来有些晦涩，因为它使用了许多宏来定义函数。它包含了主要“计算”用的 API，如果数学课你都逃掉了，这些看起来有点像天书。

这个头文件定义如下数学函数：

- ❑ acos 反余弦
- ❑ acosh 反双曲余弦
- ❑ acospi acos(x) / pi
- ❑ asin 反正弦
- ❑ asinh 反双曲正弦
- ❑ asinpi asin(x) / pi
- ❑ atan 反正切
- ❑ atan2 两个参数的反正切
- ❑ atanh 反双曲正切
- ❑ atanpi atan(x) / pi
- ❑ atan2pi atan2(x,y) / pi
- ❑ cbrt 立方根
- ❑ ceil 向上取整
- ❑ copysign 将 x 的符号（正负）换为 y 的并返回
- ❑ cos 余弦
- ❑ cosh 双曲余弦
- ❑ cospi cos(pi × x)
- ❑ erf 误差函数
- ❑ erfc 互补误差函数
- ❑ exp e^x
- ❑ exp2 2^x
- ❑ exp10 10^x
- ❑ expm1 $e^x - 1.0$
- ❑ fabs 浮点数绝对值
- ❑ fdim x 和 y 差的正值
- ❑ floor 向下取整
- ❑ fma (x × y) + z, 四舍五入
- ❑ fmax 两个浮点数的最大值
- ❑ fmin 两个浮点数的最小值
- ❑ fmod 模
- ❑ fract x 的小数部分

更多资源请访问稀酷客(www.ckook.com)

- frexp　把浮点数分解为尾数和指数
- hypot　$x^2 + y^2$ 的平方根
- ilogb　整数指数
- ldexp　$x \times 2^y$
- lgamma　gamma 绝对值的自然对数
- log　自然对数（基于 e）
- log10　常用对数（基于 10）
- log2　基于 2 的对数
- log1p　1+x 的自然对数
- logb　x 的指数，$\log_r|x|$ 的主要部分
- mad　(x × y) + z 近似值（速度先于精度）
- modf　分解浮点数为整数和小数
- nextafter　y 方向的最接近 x 的可表示的浮点数值
- pow　x^y
- pown　x^y（y 是整数）
- powr　x^y（x 大于等于 0）
- remainder　求余
- remquo　余和商
- rint　取整到最近的整数（浮点格式）
- rootn　$x^{1/y}$
- round　四舍五入（浮点格式）
- sqrt　平方根
- rsqrt　平方根倒数（即 1.0 / sqrt(x)）
- sin　正弦
- sincos　正弦和余弦
- sinh　双曲正弦
- sinpi　sin(pi × x)
- tan　正切
- tanh　双曲正切
- tanpi　tan(pi × x)
- tgamma　gamma 函数
- trunk　截断浮点值

还定义了几个整数函数：

- abs　绝对值
- clz　前导零的位数
- min　两整数最小值

更多资源请访问稀酷客(www.ckook.com)

- max　　两整数最大值

也定义了其他常用函数：

- clamp　　把数值夹到 low 和 high 之间，即 min(max(x, low), high)
- degrees　　弧度转为度
- mix　　start + (stop - start)*amount （线性混合）
- radians　　度转为弧度
- step　　如果 v 小于 edge 返回 0，否则返回 1.0
- smoothstep　　如果 v 小于等于 edge0 返回 0.0，如果 v 大于等于 edge1 返回 1.0，否则返回 Hermite 插值
- sign sign　　(–1.0, –0.0, +0.0 或 1.0)

最后，这个头文件定义了一些几何函数：

- cross　　叉积
- dot　　点积
- length　　向量长度
- distance　　两点距离
- normalize　　归一化向量（长度为 1.0）

如果熟悉 OpenCL，这些 API 应该都见过。事实上，Honeycomb 的 rs_cl.rsh 头文件就是参考 OpenCL 文档来的，它包含的注解表示引用了 OpenCL 1.1 规范的 6.11.2、6.11.3、6.11.4 和 6.11.5，分别涉及数学、整数、通用和几何函数。代码清单 9-18 从 rs_cl.rsh 摘了一些注释。[①]如果要了解 OpenCL 文档的更多信息，请到 Khronos 网站（www.khronos.org/opencl）自由浏览。

代码清单 9-18　rs_cl.rsh 摘出的包含 OpenCL 引用的注释

```
#ifndef __RS_CL_RSH__
#define __RS_CL_RSH__

...

// Float ops, 6.11.2

...
extern float __attribute__((overloadable)) trunc(float);
FN_FUNC_FN(trunc)

// Int ops (partial), 6.11.3

...
IN_FUNC_IN_IN_BODY(max, (v1 > v2 ? v1 : v2))
FN_FUNC_FN_F(max)
```

① 摘出的这些函数没有带参数信息，如果有不清楚的地方请直接参考头文件，或 http://www.khronos.org/registry/cl/sdk/1.1/docs/man/xhtml/，这里有更详细的解释。——译者注

更多资源请访问稀酷客(www.ckook.com)

```
// 6.11.4

_RS_RUNTIME float __attribute__((overloadable)) clamp(float amount, float low, float high);
...

// 6.11.5
_RS_RUNTIME float3 __attribute__((overloadable)) cross(float3 lhs, float3 rhs);

...

#endif
```

这个头文件还定义了一些转换函数，如将三元素向量转为三整数向量的 convert_int3。

9.6.4　rs_math.rsh

Honeycomb 中，这个头文件定义以下函数：

- rsSetObject
- rsClearObject
- rsIsObject
- rsGetAllocation
- rsAllocationGetDimX
- rsAllocationGetDimY
- rsAllocationGetDimZ
- rsAllocationGetDimLOD
- rsAllocationGetDimFaces
- rsGetElementAt
- rsRand
- rsFrac
- rsSendToClient
- rsSendToClientBlocking
- rsForEach

如前所述，rsForEach()是最重要的函数之一。

值得一提的是 rsSendToClient()和 rsSendToClientBlocking()函数（Android 4.0 中移入 rs_core.rsh），它们可以让脚本把数据发送到 Java 层。想要应用能够从 Java 层接收数据，需要注册消息处理函数，如代码清单 9-19 所示。

代码清单 9-19　消息处理程序

```
RenderScript rs = RenderScript.create(this); // 需要 Context 参数

rs.setMessageHandler(new RenderScript.RSMessageHandler() {

    @Override
```

更多资源请访问稀酷客(www.ckook.com)

```
        public void run() {
            super.run();
            Log.d(TAG, String.valueOf(this.mID) + " " + mData + ", length:" + mLength);
        }
    });
```

Handler 的 mID、mData、mLength 域会包含脚本传来的信息。如果脚本将数据发送到 Java 层，但没有注册处理函数，会抛出异常。

Android 4.0 中 rs_math.rsh 也改了，只定义数学函数：

- rsRand
- rsFrac
- rsClamp
- rsExtractFrustumPlanes （Android 4.0 加入）
- rsIsSphereInFrustum （Android 4.0 加入）
- rsPackColorTo8888
- rsUnpackColor8888

分配函数移到新头文件 rs_allocation.rsh 中，除了 Honeycomb 中的函数，又加入了两个新成员：

- rsAllocationCopy1DRange
- rsAllocationCopy2DRange

对象函数也移到新头文件 rs_object.rsh 中了。

9.6.5 rs_graphics.rsh

这个头文件定义了以下函数：

- rsgBindProgramFragment
- rsgBindProgramStore
- rsgBindProgramVertex
- rsgBindProgramRaster
- rsgBindSampler
- rsgBindTexture
- rsgProgramVertexLoadProjectionMatrix
- rsgProgramVertexLoadModelMatrix
- rsgProgramVertexLoadTextureMatrix
- rsgProgramVertexGetProjectionMatrix
- rsgProgramFragmentConstantColor
- rsgGetWidth
- rsgGetHeight
- rsgAllocationsSyncAll （Android 4.0 加入新的重载函数）

更多资源请访问稀酷客(www.ckook.com)

- rsgDrawRect
- rsgDrawQuad
- rsgDrawQuadTexCoords
- rsgDrawSpriteScreenspace
- rsgDrawMesh
- rsgClearColor
- rsgClearDepth
- rsgDrawText
- rsgBindFont
- rsgFontColor
- rsgMeasureText
- rsgMeshComputeBoundingBox

Android 4.0 又有下列函数添加到 rs_graphics.rsh：

- rsgBindColorTarget
- rsgClearColorTarget
- rsgBindDepthTarget
- rsgClearDepthTarget
- rsgClearAllRenderTargets
- rsgFinish

本章重点不是让你成为 3D 渲染方面的专家，这些函数就不详细赘述了。如果已经熟悉 OpenGL，这些函数上手很容易。如果是新手，那么建议你先熟悉 3D 渲染的基本概念术语，如片元、顶点、网格，然后再进入 RenderScript 的世界。

9.6.6　rs_time.rsh

这个文件定义如下函数：

- rsTime
- rsLocalTime
- rsUptimeMillis
- rsUptimeNanos
- rsGetDt

rsGetDt()函数挺让人感兴趣的，它返回上次调用到现在的秒数（浮点值）。代码清单 9-20 给出其在脚本中的常用法。

代码清单 9-20　调用 rsGetDt()

```
#pragma version(1)
#pragma rs java_package_name(com.yourpackagehere)
```

更多资源请访问稀酷客(www.ckook.com)

```
typedef struct __attribute__((packed, aligned(4))) MyObject {
    float x;
    float other_property;
} MyObject_t;
MyObject_t *object; // Java层要调用 bind_object

int root() {
    float dt = rsGetDt();
    float dx = dt * 10.f; // 每秒10个像素

    object->x += dx; // 对象的新x坐标

    // 做些其他事情，比如画对象

    return 20;
}
```

注意 虽然 root()函数返回 20（即渲染两帧之间用时 20 毫秒或每秒 50 帧），但无法保证帧速率。这是为什么代码要使用 rsGetDt()，而不是假设两帧之间的时间一定是 root()返回的值。

9.6.7 rs_atomic.rsh

这个文件在 Android 4.0 加入的新函数：

- rsAtomicInc
- rsAtomicDec
- rsAtomicAdd
- rsAtomicSub
- rsAtomicAnd
- rsAtomicOr
- rsAtomicXor
- rsAtomicMin
- rsAtomicMax
- rsAtomicCas

从这些名字就可以看出，这些函数的操作都是原子的。

9.7 RenderScript 与 NDK 对比

RenderScript 和 NDK 都是为了提高应用的性能而引入的。虽然它们有共同的目标，但应该仔细斟酌它们的区别。每样东西都有其优缺点，不考虑清楚，往往会在开发过程中卡住，或发现某些实现难以做到，几乎是不可能完成的任务。

更多资源请访问稀酷客(www.ckook.com)

RenderScript 的优势可以概括如下：

- 平台独立（例如 ARM、MIPS、Intel）；
- 易于并行执行；
- 可以使用 CPU、GPU 或其他处理器。

另一方面，RenderScript 确实有一些缺点：

- Android 独有（脚本不能用在其他平台，如 iOS）；
- 学习难度；
- API 数量有限。

RenderScript 的实际性能测试结果都很难得到，它们实在太依赖 Android 版本和硬件设备了。一些脚本的运行速度可能远超 Java 或 NDK 的对应实现，而其他的可能不会。如果因为遗留代码的缘故，或者这样做有更有意义，你大可以在实际中让应用同时使用 RenderScript 和 NDK。

9.8　总结

虽然它似乎仍然是 Honeycomb 带来的略显稚嫩的技术，RenderScript 已经在 Android 4.0 中有所改善，在将来会更好地发挥硬件的优势。[①]尽管时下 ARM 是 Android 移动设备的主要平台，使用 NDK 是合理的选择，可 RenderScript 是独立于平台的，这样可以显著降低成本，给你带来巨大收益。总的说来，RenderScript 挺有前景，并且很可能是将是未来应用的很重要的组成部分，会用在单靠 Java 不能满足性能需求的地方。

① Android 4.03 及 Android 4.1 又增加了

更多资源请访问稀酷客(www.ckook.com)

亚马逊读者评论

“本书详细介绍了优化Android代码的各种规则和技巧，揭开了Android和JAVA核心数据结构的神秘面纱。最值得称道的是，作者展示了使用缓存、SQLite以及延长电池使用寿命的技术，这是每个严谨的开发人员都必须要掌握的内容。”

“市面上这种书并不多见！我想把这本书推荐给所有Android高级程序员。”

Pro Android Apps
Performance Optimization

Android应用性能优化

今天的Android应用开发者经常要想尽办法来提升程序性能。由于应用越来越复杂，这个问题也变得越来越棘手。本书主要介绍如何快速高效地优化应用，让应用变得稳定高效。你将学会利用Android SDK和NDK来混合或单独使用Java、C/C++来开发应用。书中还特别讲解了如下内容：

- 一些OpenGL的优化技术以及RenderScript（Android的新特性）的基础知识；
- 利用SDK来优化应用的Java代码的技巧；
- 通过高效使用内存来提升性能的技巧；
- 延长电池使用时间的技巧；
- 使用多线程的时机及技巧；
- 评测剖析代码的技巧。

把本书的内容学以致用，你的编程技术就会得到关键性的提升，写出的应用就会更为健壮高效，从而广受用户好评，并最终获得成功。

Apress®
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育 @图灵社区
反馈/投稿/推荐邮箱：contact@turingbook.com
热线：(010)51095186转604
分类建议 计算机/移动开发
人民邮电出版社网址：www.ptpress.com.cn

ISBN 978-7-115-27241-6
9 787115 272416 >
ISBN 978-7-115-27241-6
定价：49.00元

更多资源请访问稀酷客(www.ckook.com)