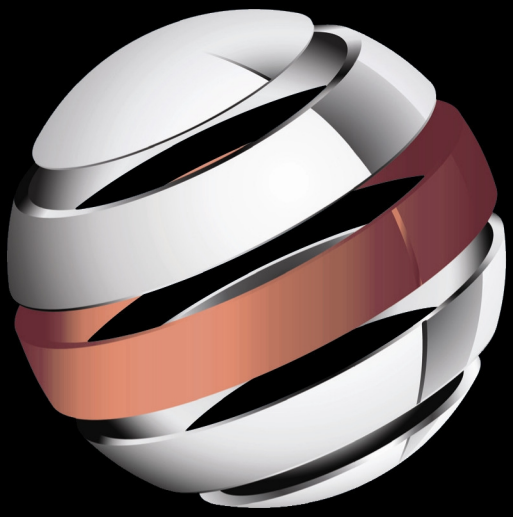
TURING 图灵程序设计丛书 移动开发系列

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



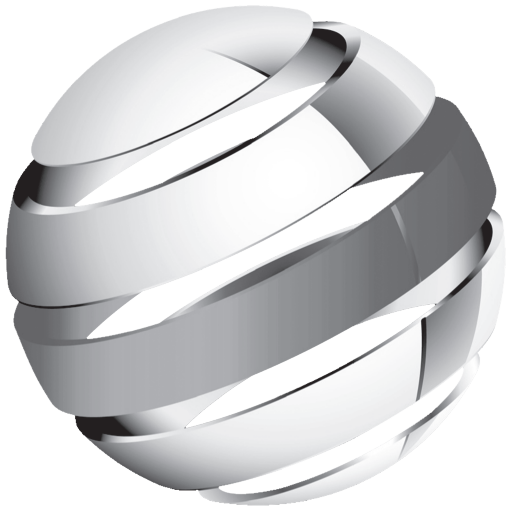
Pro Android Apps
Performance Optimization

Android应用性能优化

[法] Hervé Guihot 著
白龙 译

人民邮电出版社
POSTS & TELECOM PRESS

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

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 印刷

◆ 开本：800×1000 1/16
印张：14.25
字数：335千字 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

版 权 声 明

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

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

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

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，

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！

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。

目　录

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

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
```

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

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;
```

```
                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。

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。

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

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];
```

```
    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

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

```
        }
        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