

深入理解Android网络编程

系统讲解Android网络编程的各项核心技术和应用模块，通过多个案例解析
Android网络编程的方法和技巧
从源码角度深入解析Android核心网络处理方法和关键应用的实现原理，包含
大量最佳实践

移动开发



陈文 郭依正〇著

Understanding Android Network Programming Core Technology and Best Practice

深入理解Android网络编程 技术详解与最佳实践

机械工业出版社
China Machine Press

机械工业出版社

目录

序

第1章 Android网络编程概要

1.1 Android简介

1.1.1 Android的发展

1.1.2 Android功能特性

1.1.3 Android系统构架

1.2 Android网络程序的功能

1.2.1 通信功能

1.2.2 及时分享

1.2.3 个人管理

1.2.4 娱乐游戏

1.2.5 企业应用

1.3 设置Android开发环境

1.3.1 相关下载

1.3.2 安装ADT

1.3.3 Hello World !

1.4 网络应用实战案例

1.4.1 加载一个页面

[1.4.2 下载一个页面](#)

[1.5 小结](#)

[第 2 章 Android基本网络技术和编程实践](#)

[2.1 计算机网络及其协议](#)

[2.1.1 计算机网络概述](#)

[2.1.2 网络协议概述](#)

[2.1.3 IP、 TCP和UDP协议](#)

[2.2 在Android中使用TCP、 UDP协议](#)

[2.2.1 Socket基础](#)

[2.2.2 使用TCP通信](#)

[2.2.3 使用UDP通信](#)

[2.3 Socket实战案例](#)

[2.3.1 Socket聊天举例](#)

[2.3.2 FTP客户端](#)

[2.3.3 Telnet客户端](#)

[2.4 小结](#)

[第 3 章 Android基本Web技术和编程实践](#)

[3.1 HTTP协议](#)

[3.1.1 HTTP简介](#)

[3.1.2 实战案例：基于HTTP协议的文件上传](#)

[3.2 Android中的HTTP编程](#)

[3.2.1 HttpClient和URLConnection](#)

[3.2.2 Post和Get在HttpClient的使用](#)

[3.2.3 实战案例：使用HttpClient和URLConnection访问维基百科](#)

[3.3 Android处理JSON](#)

[3.3.1 JSON简介](#)

[3.3.2 JSON数据解析](#)

[3.3.3 JSON打包](#)

[3.3.4 实战案例：JSON解析wikipedia内容](#)

[3.4 Android处理SOAP](#)

[3.4.1 SOAP简介](#)

[3.4.2 SOAP消息](#)

[3.4.3 实战案例：SOAP解析天气服务](#)

[3.5 Android对HTML的处理](#)

[3.5.1 解析HTML](#)

[3.5.2 HTML适配屏幕](#)

[3.5.3 JavaScript混合编程](#)

[3.5.4 实战案例：Android自定义打开HTML页面](#)

[3.6 小结](#)

[第 4 章 Android常见网络接口编程](#)

[4.1 Android解析和创建XML](#)

[4.1.1 XML简介](#)

[4.1.2 DOM解析XML](#)

[4.1.3 SAX解析XML](#)

[4.1.4 PULL解析XML](#)

[4.1.5 实战案例：Android中创建XML](#)

[4.2 Android订阅RSS](#)

[4.2.1 RSS简介](#)

[4.2.2 实战案例：简单RSS阅读器](#)

[4.3 Android Email编程](#)

[4.3.1 Android发送Email](#)

[4.3.2 实战案例：Android下Email的Base64加密](#)

[4.4 Android网络安全](#)

[4.4.1 Android网络安全简介](#)

[4.4.2 Android加密和解密](#)

[4.4.3 实战案例：Android应用添加签名](#)

[4.5 OAuth认证](#)

[4.5.1 OAuth简介](#)

[4.5.2 实战案例：使用OAuth接口](#)

[4.6 小结](#)

[第 5 章 Android网络模块编程](#)

[5.1 Android地图和定位](#)

[5.1.1 获取map-api密钥](#)

[5.1.2 获取位置](#)

[5.1.3 实战案例：利用MapView显示地图](#)

[5.2 USB编程](#)

[5.2.1 USB主从设备](#)

[5.2.2 USB Accessory Mode](#)

[5.2.3 USB Host Mode](#)

[5.2.4 实战案例：Android和Arduino交互](#)

[5.3 Wi-Fi编程](#)

[5.3.1 Android Wi-Fi相关类](#)

[5.3.2 Android Wi-Fi基本操作](#)

[5.3.3 实战案例：使用Wi-Fi直连方式传输文件](#)

[5.4 蓝牙编程](#)

[5.4.1 蓝牙简介](#)

[5.4.2 Android蓝牙API分析](#)

[5.4.3 Android蓝牙基本操作](#)

[5.4.4 实战案例：蓝牙连接](#)

[5.5 NFC编程简介](#)

[5.5.1 NFC技术简介](#)

[5.5.2 NFC API简介](#)

[5.5.3 NFC处理流程分析](#)

[5.6 小结](#)

第6章 Android线程、数据存取、缓存和UI同步

6.1 Android线程

6.1.1 Android线程模型

6.1.2 异步任务类

6.1.3 实战案例：利用AsyncTask实现多线程下载

6.2 数据存取

6.2.1 Shared Preferences 数据存储

6.2.2 Internal Storage 数据存储

6.2.3 External Storage 数据存储

6.2.4 SQLite Databases 数据存储

6.2.5 实战案例：SQLite数据库操作

6.3 网络判定

6.3.1 判断用户是否连接

6.3.2 判断网络连接的类型

6.3.3 监控网络连接改变

6.3.4 实战案例：根据广播消息判断网络连接情况

6.4 消息缓存

6.4.1 Android中的缓存机制

6.4.2 实战案例：下载、缓存和显示图片

6.5 界面更新

6.5.1 刷新数据时的界面更新

[6.5.2 完成任务时的界面更新](#)

[6.5.3 实战案例：自定义列表显示更新](#)

[6.6 小结](#)

[第 7 章 基于SIP协议的VoIP应用](#)

[7.1 SIP协议简介](#)

[7.2 SIP服务器搭建](#)

[7.2.1 下载安装Brekeke SIP Server](#)

[7.2.2 访问服务器](#)

[7.2.3 启动服务器](#)

[7.3 SIP程序设置](#)

[7.3.1 Android SIP API 中的类和接口](#)

[7.3.2 Android极限列表](#)

[7.3.3 完整的Manifest文件](#)

[7.4 SIP初始化通话](#)

[7.4.1 SipManager对象](#)

[7.4.2 SipProfile对象](#)

[7.5 监听SIP通话](#)

[7.5.1 创建监听器](#)

[7.5.2 拨打电话](#)

[7.5.3 接收呼叫](#)

[7.6 实战案例：SIP通话](#)

7.7 小结

第 8 章 基于XMPP协议的即时通信应用

8.1 XMPP协议简介

8.2 使用Openfire搭建XMPP服务器

8.3 登录XMPP服务器

8.3.1 Asmack相关类

8.3.2 登录XMPP服务器

8.4 联系人相关操作

8.4.1 获取联系人列表

8.4.2 获取联系人状态

8.4.3 添加和删除联系人

8.4.4 监听联系人添加信息

8.5 消息处理

8.5.1 接收消息

8.5.2 发送消息

8.6 实战案例：XMPP多人聊天

8.6.1 创建新多人聊天室

8.6.2 加入聊天室

8.6.3 发送和接收消息

8.7 小结

第 9 章 Android对HTML的处理

[9.1 Android HTML处理关键类](#)

[9.2 HTMLViewer分析](#)

[9.3 浏览器源代码解析](#)

[9.3.1 WebView加载入口分析](#)

[9.3.2 调用JavaScript接口](#)

[9.4 WebKit简单分析](#)

[9.4.1 HTTP Cache管理](#)

[9.4.2 Cookie管理](#)

[9.4.3 处理HTTP认证以及证书](#)

[9.4.4 处理JavaScript的请求](#)

[9.4.5 处理MIME类型](#)

[9.4.6 访问WebView的历史](#)

[9.4.7 保存网站图标](#)

[9.4.8 WebStorage](#)

[9.4.9 处理UI](#)

[9.4.10 Web设置分析](#)

[9.4.11 HTML5音视频处理](#)

[9.4.12 缩放和下载](#)

[9.4.13 插件管理](#)

[9.5 小结](#)

[第 10 章 Android网络处理分析](#)

[10.1 Android网络处理关键类及其说明](#)

[10.2 Android网络处理流程](#)

[10.2.1 监控网络连接状态](#)

[10.2.2 认证类](#)

[10.2.3 DHCP状态机](#)

[10.2.4 LocalServerSocket](#)

[10.2.5 响应邮件请求](#)

[10.2.6 提供网络信息](#)

[10.2.7 Proxy类](#)

[10.2.8 VPN服务](#)

[10.3 Android封装的HTTP处理类](#)

[10.3.1 AndroidHttpClient类和DefaultHttpClient类](#)

[10.3.2 SSL认证信息处理类](#)

[10.3.3 SSL错误信息处理](#)

[10.3.4 AndroidHttpClient](#)

[10.4 Android RTP协议](#)

[10.4.1 传输音频码](#)

[10.4.2 AudioGroup](#)

[10.4.3 语音流RtpStream和AudioStream](#)

[10.5 Android SIP协议](#)

[10.5.1 SIP通话简介](#)

[10.5.2 SIP初始化](#)

[10.5.3 SipProfile](#)

[10.5.4 SipSession](#)

[10.5.5 SIP包错误处理](#)

[10.6 小结](#)

[第 11 章 Android网络应用分析](#)

[11.1 Android中使用SAX解析XML](#)

[11.1.1 几种XML解析方式讨论](#)

[11.1.2 SAX解析XML的原理](#)

[11.1.3 SAX发现XML的根元素](#)

[11.1.4 SAX发现XML的子元素](#)

[11.2 基于位置的服务](#)

[11.2.1 位置服务的基本概念](#)

[11.2.2 位置服务的基本类](#)

[11.2.3 调用Google地图](#)

[11.2.4 根据位置刷新地图显示](#)

[11.3 媒体传输协议](#)

[11.3.1 MTP和PTP简介](#)

[11.3.2 定义MTP和PTP的类型](#)

[11.3.3 封装MTP设备信息](#)

[11.3.4 封装MTP对象的信息](#)

11.3.5 封装MTP设备上存储单元的信息

11.4 小结

本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供

最新最全的优质电子书下载！！！

序

移动互联网发展迅速，既有机会又充满危机。善于学习、勇于挑战者总能在机遇来临时抓住机会迈上一个新的台阶，达到一个新的高度，面对危机时也能从中看到希望从而破茧而出，实现危与机的转换。

和陈文的相识纯属偶然，当时创业不久，没有多少钱可以雇佣高薪员工，所以希望能从应届毕业生中淘些璞玉出来。仅是一个电话，我意识到这就是我要找的人才，思路清晰、有冲劲、知识面开阔，而且对新事物有强烈的兴趣和参与的欲望。

以后的合作如预料中一样愉快。为实现“每个人都有平等学习的机会”的美好愿景，大家通力协作，每一个团队成员都能以主人翁的心态去面对一些挫折并积极考虑解决方案，希望能在移动教育领域书写属于我们自己的传奇。虽然暂时并未如愿，但我相信通过创业实现人生价值和为社会做出贡献的理想已经融入到团队每一位成员的日常行动中。所以陈文后来跟我说他再次创业了，我丝毫不觉得意外，并乐于为其在创业之余挤出时间完成的这部著作作序。

一个好的行知者总喜欢探索新事物，并在实践中不断修炼、感悟，实现理论和实践的良性互动，不断在失败和成功中总结经验教训，只为

下一步能走得更好。这也是德鲁克的自我管理精神和当前热门行业敏捷发展理念的内涵所在。纵观陈文的这本书，覆盖了Android乃至整个移动互联网领域中网络编程的方方面面，从基础的HTTP协议、多线程开发到即时通信、VOIP均有涉及，而且行文之处不时闪现出自己的理解和独到见解，足见作者百忙之余的用心和用功。

人生的起步总是从站在前人的肩膀上开始，愿本书的出版能够对渴望迈入移动互联网行业的读者提供一些帮助，并为其下一步的成长提供知识上的助力和理想的传递，如果有机会，就开始属于自己的传奇！

苗忠良

于福州机场

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第1章 Android网络编程概要

今天，Android上使用网络的应用越来越多，如电子邮件、Web浏览器和IM等传统的应用都是基于网络的程序；微博、微信等大量的新兴应用都是在网络的基础上开发的；音乐播放器、词典等传统的本地应用，在加入在线存储功能、在线推荐、分享等功能后也成为网络应用。

随着Android的发展，其对网络编程的支持也日益强大。Android系统的功能已经远远超过了普通通信手机的功能，更像是有手机功能的PC。Android网络编程将会变得更加简洁和广泛：一方面Android的开源和强大的开发框架大大简化了网络应用的编程；另一方面众多网络服务提供商的开放API也对网络编程提供了极大的便利。

在用Android编写网络程序的时候，需要了解一些Android开发的基础知识。本章将概述Android的发展，讨论Android网络程序的功能，设置Android开发环境。本章最后将用实战案例来具体分析Android网络编程的步骤。

1.1 Android简介

1.1.1 Android的发展

Android纪元正式开始于2008年10月22日。这天，T-Mobile G1正式在美国公开发售。时至今日，Android平台集成了操作系统、中间件、用户界面和应用软件，已经成为开放和完整的移动生态系统，可谓发展飞速。

目前移动终端市场上，随着Android平台的发展以及不断完善，越来越多的厂商开始选择Android系统作为其主要发展方向，自2008年9月Android系统的第一个版本发布至今，Android系统在手机市场大放异彩，已经长期占据市场份额第一的位置。就目前来说，Android手机的统治地位还是无可动摇的。Android 4.0版本发布以来已渐成主力，推动Android手机和平板的份额不断提高，同时也为Android系统“碎片化”的问题提供了可靠的解决方案。

Android系统能够取得今天的成功，最主要的应归功于其开源及免费性。正是在其开源和免费的基础上，各大厂商纷纷在原生系统的基础上进行定制和扩展，植入自身的应用，开发出更多有特色的产品，来满足市场的需求。这种情况在促进全球智能手机产业发展的同时，也使得Android系统的覆盖面积更为广阔。

Android的开源，对于厂家来说可以更好地集成自己的产品和服务；开发者更可以在其开源的基础上进行进一步开发，提供更好的应用；用户能用到免费的Android系统和众多的应用。

目前采用了Android系统的主要的大手机厂商包括：HTC、联想、华为、中兴、魅族、小米、酷派、天语、华硕、OPPO、三星、摩托罗拉、索尼、LG和夏普等。Android已经成为互联网的重要入口和载体。很多互联网企业开始在Android系统上发力，360、盛大、百度、阿里巴巴和网易等互联网巨头，均开始致力于千元左右的智能手机的开发。

Android的未来充满了活力，将给人们的生活带来更加深刻的变革。Google在Android移动平台的基础上推出云音乐服务和电影服务，并与电子书服务相结合，提供更为全面的内容资源。Android TV借助各种应用和游戏，变身成为客厅多媒体娱乐中心的理想将成为现实。

Android开放式配件标准包括第三方配件的硬件设计和系统API。第三方配件将会层出不穷，届时这些配件均可得到Android设备的兼容支持。未来将会有更多的智能设备出现，比如Android音箱、闹钟，甚至电饭锅、电冰箱等。如果有大量的Android第三方配件出现，基于Android的家庭自动化则可以让整个家庭生活都会更方便、更欢乐。

1.1.2 Android功能特性

Android系统在其开放性的基础上，引入了很多由软件和硬件实现的实用功能，在方便人们使用的同时，也给了开发者广阔的空间。下面是一些重要的功能特性。

- 数据存储。Android提供了SharedPreferences、ContentProvider、文件、SQLite数据库和网络等多种方式来存储数据。
- 通信网络。Android操作系统支持所有的网络格式，包括GSM/EDGE、IDEN、CDMA、EV-DO、UMTS、Bluetooth、Wi-Fi、LTE、NFC和WiMAX等。
- 信息。Android操作系统原生支持短信和邮件，并且支持所有的云端信息和服务器信息。
- 语言。Android操作系统支持多语言。
- 浏览器。Android操作系统中内置的网页浏览器基于WebKit内核，并且采用了Chrome V8引擎。在Android 4.0内置的浏览器测试中，HTML5和Acid3故障处理中均获得了满分。
- 支持Java。虽然Android操作系统中的应用程序大部分都是由Java编写，但是Android却需要转换为Dalvik执行文件，在Dalvik虚拟机上运行。由于Android中并不自带Java虚拟机，因此无法直接运行Java程

序。不过Android平台上提供了多个Java虚拟机供用户下载使用，安装了Java虚拟机的Android系统可以运行J2ME的程序。

■多媒体。Android操作系统本身支持以下格式的音频/视频/图片媒体：WebM、H.263、H.264（in 3GP or MP4 container）、MPEG-4 SP、AMR,AMR-WB（in 3GP container）、AAC、HE-AAC（in MP4 or 3GP container）、MP3、MIDI、Ogg Vorbis、FLAC、WAV、JPEG、PNG、GIF、BMP。

■流媒体。Android操作系统支持RTP/RTSP（3GPP PSS,ISMA）的流媒体以及（HTML5<video>）的流媒体，在安装了RealPlayer之后，还支持苹果公司的流媒体。

■外围设备。Android操作系统支持识别并且使用视频/照片摄像头、多点电容/电阻触摸屏、GPS、加速计、陀螺仪、气压计、磁强计、键盘、鼠标、U盘、专用的游戏控制器、体感控制器、游戏手柄、蓝牙设备、无线设备、感应和压力传感器、温度计、2D和3D图形加速等。

■多点触控。Android内核支持原生的多点触控。

■多任务处理。Android操作系统支持原生的多任务处理。

■语音功能。除了支持普通的电话通话之外，Android操作系统从最初版本就支持使用语音进行网页搜索等功能。而从Android 2.2开始，语音

还可以用来输入文本、实现语音导航等功能。

■无线共享功能。Android操作系统支持用户使用本机充当无线路由器，并且将本机的网络共享给其他手机，其他机器只需要通过WiFi寻找找到共享的无线热点，就可以上网。

■截图功能。从Android 4.0版本开始，Android操作系统便支持截图功能，该功能允许用户直接抓取手机屏幕上的任何画面，用户可以通过编辑功能对截图进行处理，还可以通过蓝牙、Email、微博等方式共享给其他用户或者上传到网络上，也可以复制到计算机中。

■Google Now。Google Now是Android 4.1的一个新功能，这个功能可以根据搜索历史或者日历以及其他更多数据来预测出用户想要的到底是什么，并在指定的时间或者地点进行搜索并提出反馈建议。比如当用户有一个新的日历预约，Google将利用各种信息（交通数据、地图、公交换乘）来帮助用户准时到达预约地点；如果用户搜索了一个航班信息，Google将会持续通知这个航班的动态更新；甚至还可以跟踪一个球队的表现情况。

■Android Beam功能。Android Beam优化了近场通信以及蓝牙分享功能。

■Smart App Updates。Smart App Updates是一种智能型的应用更新模式，应用程序在更新时不需要下载整个APK，只需要下载修改的部分

即可，这样更节省流量。

1.1.3 Android系统构架

Android不仅仅局限于操作系统，Android平台由操作系统、中间件、用户友好的界面和应用软件组成。Android核心是经过Google剪裁和调优的Linux Kernel，对于掌上设备的硬件提供了优良的支持；在Dalvik虚拟机上，大部分Java核心类库都已经可以直接运行；拥有大量立即可用的类库和应用软件，可以轻易开发出可媲美桌面应用复杂度的手机软件；基于Android,Google已经开发大量好的应用软件，同时可以直接使用Google很多的在线服务；Google还提供了基于Eclipse的完整开发环境、模拟器、文档、帮助和示例。

Android系统框图如图1-1所示。可以看出Android分为5层，从低到高分别是Linux Kernel内核层、Android系统库、Android运行时、应用程序框架层和应用层。

■Linux Kernel内核层。Linux内核层是硬件和软件层之间的抽象层。其包含了显示驱动、摄像头驱动、蓝牙驱动、闪存驱动、IPC管道通讯驱动、USB驱动、键盘驱动、Wi-Fi无线驱动、音频驱动和电源管理驱动。最下层是Linux系统核心驱动，主要用于协调CPU处理和内存管理。

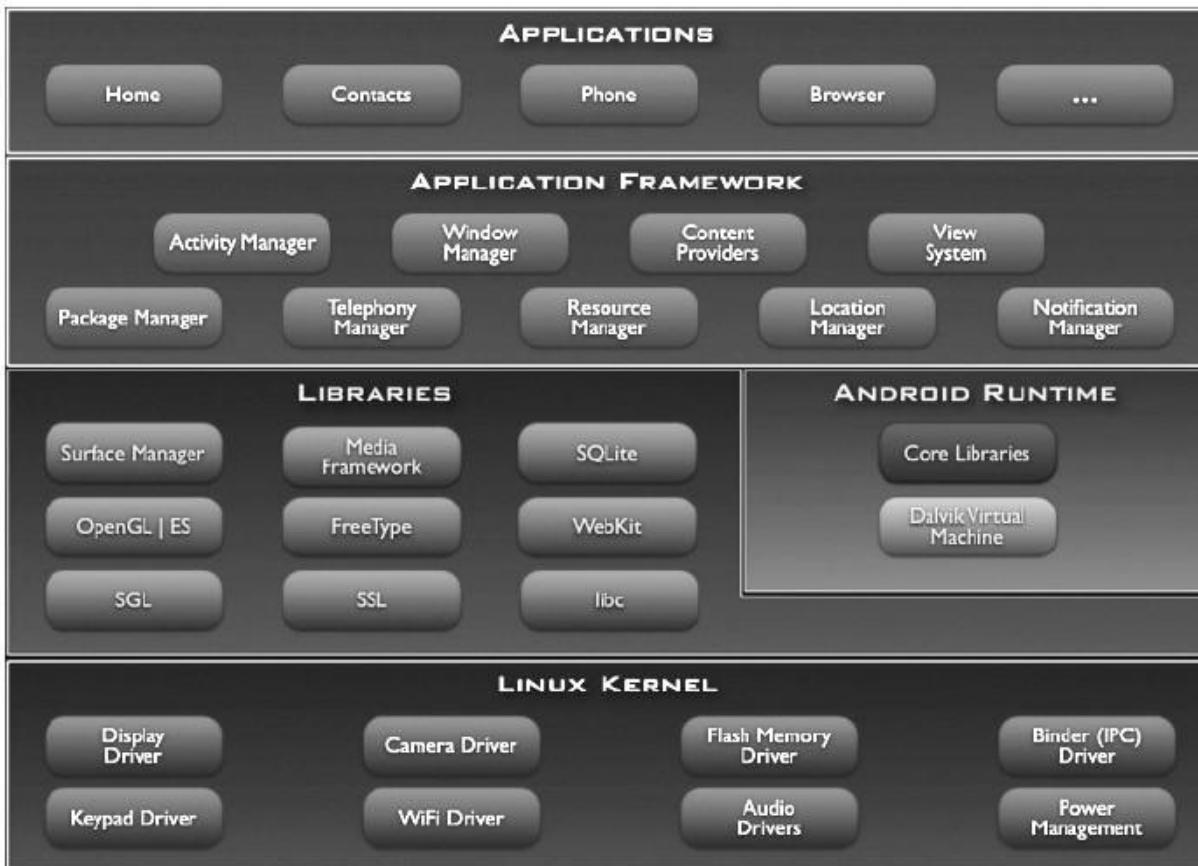


图 1-1 Android系统框图

■Android系统库。Android包含一个C/C++库的集合，供Android系统的各个组件使用。这些功能通过Android的应用程序框架提供给开发者。其核心库包含：SurfaceManager显示系统管理库，负责把2D或3D内容显示到屏幕；Media Framework媒体库，负责支持图像，支持多种视频和音频的录制和回放；SQLite数据库引擎、OpenGL ES图形引擎、FreeType位图和矢量字体渲染引擎、Webkit浏览器引擎、SGL基本的2D图形引擎、SSL安全套接字层引擎、Libc库以及Android Dalvik虚拟机运行库。

- Android运行时。Android包含一个核心库的集合，提供大部分在Java编程语言核心类库中可用的功能。Dalvik被设计成在一个设备可以高效地运行的多个虚拟机，每一个Android应用程序都在它自己的进程中运行，也就是都有一个属于自己的Dalvik虚拟机。这可以让系统在运行时可以优化，从而使程序间的影响大大降低。Dalvik虚拟机并非运行Java字节码，而是运行自己的字节码。Dalvik虚拟机依赖于Linux内核提供基本功能，如线程和底层内存管理。
- 应用程序框架层。应用程序框架层简化了程序开发的架构设计，开发者可以完全使用核心应用程序所使用的框架接口，任何应用程序都能发布它的功能，且任何其他应用程序可以使用这些功能（需要服从框架执行的安全限制）。应用程序框架层主要是系统管理类库，包括Activity管理、窗口管理、内容提供、显示系统基类、消息通知管理、程序包管理、电话管理、资源管理和定位管理。
- 应用层。Android应用层包含核心应用程序，如Home桌面、Contacts联系人、Phone拨打电话、Browser浏览器等，开发者的大部分应用也在这一层。

1.2 Android网络程序的功能

Android的网络程序大大增强了简单程序的功能。通过网络，一个程序可以和成千上万的人进行通信；可以获取世界上联网计算机中存储的信息；可以利用许多计算机的能力来解决一个问题。

Android网络应用程序最基本的形式是作为应用客户端。Android客户端获取服务器的数据并显示。比较复杂的Android网络应用还会对获取的数据进行处理，不断更新数据，向他人和计算机发送数据以实现实时交互。较少的Android网络程序将Android作为服务器来使用，这是Android网络应用程序的另外一种形式，比如作为家庭多媒体中心，为其他设备提供信息服务。Android平台具有的开放特点和其高度集成的开发框架，使其成为个人移动中心、家庭媒体中心，也必将使其在企业应用上大展身手。

1.2.1 通信功能

电话、短信等传统应用在Android系统上得到了加强，Google为电话、短信和联系人提供了强大的管理功能和智能云备份的能力，一些应用也为电话提供了归属地查询和垃圾短信过滤等功能。电话和短信构成了Android最基本的通信应用，满足用户最基本的通信需要。借助于IM、KIK等应用，不仅可以用文本来传输信息，还可以使用图像、语

音、视频等方式交互；可以将通信内容完全保存在服务器上，方便随时查看。

Android网络应用程序丰富了通信的方式，提高了通信的质量，只要在联网的情况下就可以使用，不必通过电信运营商，大大降低了通信的成本。

1.2.2 及时分享

Android应用的社会化已经成为了不可或缺的基本功能，很多应用程序都增加了分享的功能以实现其社会化。通过社会化应用，用户可以将自己的心情、想法随时随地发到微博上；看到有趣的图片也可以直接上传到分享网站；有需要帮助的问题，立刻就可以使用问答应用提问，等待大家的回答。这些分享的信息，不但能充分利用移动的优势，将用户的信息及时发布到互联网上，分享给好友，还能充分利用终端的硬件优势，比如GPS、摄像头等，多维度地分享自己的生活和见闻。用户在分享的过程中，大量的经验和知识被保存起来，为知识化的互联网提供了更多的素材。

Android网络应用程序丰富了社会化交流，用户可以及时发布信息、提供帮助、交流看法，分享身边的一切。

1.2.3 个人管理

个人事务管理应用将Android变成随身的智能管理工具。印象笔记、有道笔记和麦库等云笔记应用以知识管理见长，帮助用户收集随手得来的内容，把碎片知识整理起来，存储到云端，用户可以在多个设备上方便地查看。Doit.im、Remember The Milk和toodledo等时间管理应用重造了任务列表，最大程度地帮助用户管理时间。挖财、随手记等应用帮助用户对个人财务进行分析。

Android网络应用程序加强了用户管理时间、知识、财务的能力，用户把以前记到本子上的内容，通过Android记到云端，可以随时随地查看、学习和管理，用起来更加得心应手。

1.2.4 娱乐游戏

近年来，Android系统软硬件的快速发展（软件方面，Android系统原生支持更多的游戏外设；硬件方面，CPU、内存和屏幕等硬件变得更加强大），使得游戏和娱乐应用不断发展。Android应用商店里面的游戏娱乐的比重在不断增加，更多的大型游戏出现在Android上。同时使用Android进行在线支付、购物变得更加方便。

在线的游戏、视频、音乐、广播已经成为人们生活的一部分，Android已变为娱乐游戏的强大载体。

1.2.5 企业应用

Android作为免费的系统，可以有效降低企业应用的成本；而开放的源代码又为企业应用提供了更多稳定性的保证。众多硬件厂商也根据不同的应用环境开发了不同特性的硬件，比如医疗、军事、运动等不同方面的Android硬件产品。企业软件也在不断发展，比如Epocrates应用可帮助医疗护理专业人员快速和方便地获取可靠的药物信息；SAP的SkyMobile应用可帮助销售和服务人员访问SAP的CRM系统，进而访问客户数据。

Android企业应用将进一步提高企业的生产效率，为企业移动办公提供更加方便、可靠的平台。

1.3 设置Android开发环境

Android应用的开发需要建立一个开发环境，并需要进行一些设置。本节将以Windows平台为例来一步一步地讲解如何设置Android开发环境。

1.3.1 相关下载

1.JDK下载

首先需要下载JDK6。Android SDK需要JDK5或者JDK6，我们使用JDK6来开发本书中的案例。可以从Oracle的官方网站(<http://www.oracle.com/technetwork/java/javase/downloads/index.html>)下载JDK6，然后安装。

安装JDK之后需要设置环境变量，设置JAVA_HOME环境变量指向JDK安装文件夹。在Windows XP系统中，可以右击“我的电脑”图标，在弹出的快捷菜单中选择“属性”命令，然后在弹出的对话框中选择“高级”选项卡，然后单击“环境变量”按钮。“新建”或者“编辑”JAVA_HOME变量，将其设置为上面JDK的安装目录。在Windows Vista和Window 7中，可以选择“开始”→“计算机”，右击选择“属性”，依次单击“高级系

统设置”→“环境变量”，就可以更改环境变量了，将JAVA_HOME变量设置为JDK的安装目录。

通过在命令行里面输入Java-c来验证设置是否正确，设置正确之后会出现图1-2所示的提示（其中版本号可能不同）。

```
java version "1.6.0_30"
Java(TM) SE Runtime Environment (build 1.6.0_30-b12)
Java HotSpot(TM) 64-Bit Server VM (build 20.5-b03, mixed mode)
```

图 1-2 Java-c提示

2.Eclipse下载

安装JDK之后，可以下载Eclipse IDE for JAVA Developer。本书中使用Eclipse 3.7进行示例开发。其下载网址为

<http://www.eclipse.org/downloads/>。下载之后为一个.zip文件，可以解压缩到合适的目录下，目录中的启动图标为eclipse.exe。

3.下载Android SDK

需要使用Android SDK来开发Android应用程序，其下载网址为<http://developer.android.com/sdk/index.html>，根据不同的系统选择相应的版本下载并安装。图1-3所示是SDK安装过程中的一个截图。安装之后启动Android SDK Manager，根据开发需要，选择安装平台版本。本书中如无特别指明，均是使用Android 4.1的平台版本。

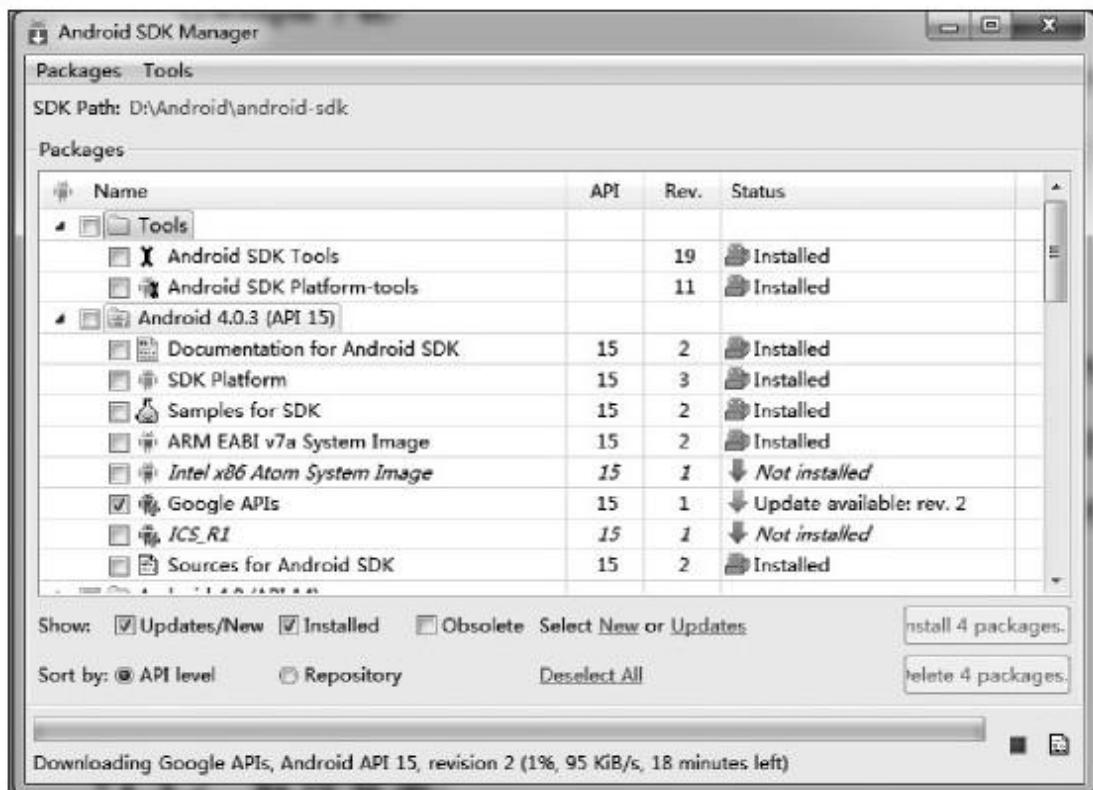


图 1-3 SDK安装过程截图

1.3.2 安装ADT

ADT插件对于开发Android应用程序有非常重要的作用，ADT和Eclipse集成之后，提供了一些工具来自动地创建、检测、测试和调试Android应用程序。可以在Eclipse中的Install New Software处进行安装。启动Eclipse IDE之后，其步骤如下：

步骤1 在Eclipse菜单栏上选择Help，单击下拉菜单Install New Software选项。

步骤2 在Work with字段中，输入https://dl-ssl.google.com/android/eclipse/，回车后Eclipse会链接到该网站，并生成可以下载插件的列表，如图1-4所示。

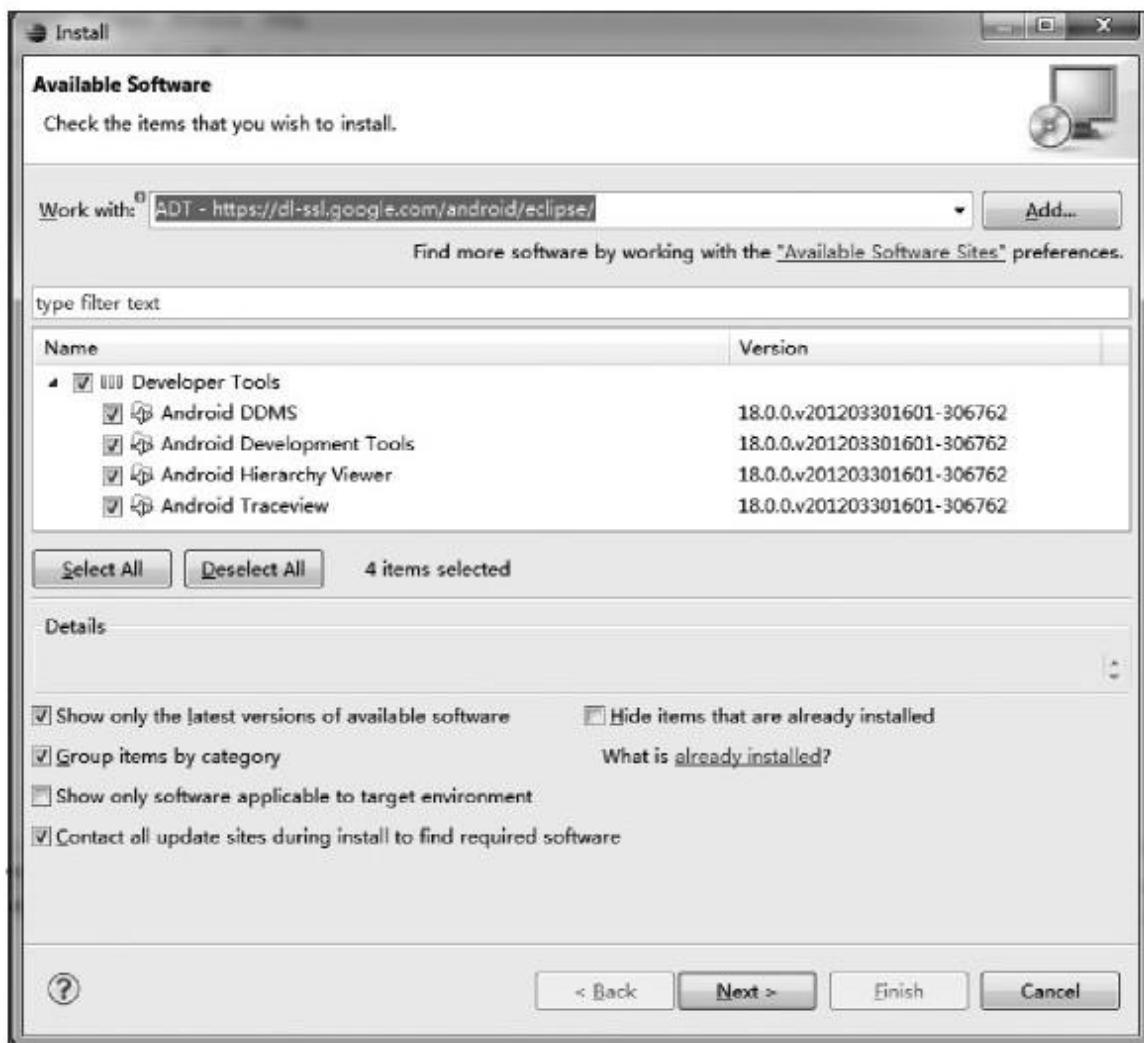


图 1-4 ADT下载插件列表

注意 如果已经安装过会显示需要更新的项目。如输入的链接不能使用，可以使用http://dl-ssl.google.com/android/eclipse/链接，或者下载离

线安装包，单击图中的Add按钮，从Archive里面选择离线安装包的位置。

步骤3 选择Developer Tools，将全部选中其子选项，然后单击Next按钮。

步骤4 Eclipse将要查看ADT和ADT安装所需要工具的许可协议。查看许可协议，单击“*I accept.....*”，然后单击Finish按钮。

步骤5 Eclipse将下载Developer Tools并安装，安装过程中Eclipse需要重新启动才能使用ADT。

步骤6 在Eclipse中安装ADT之后需要配置Android SDK，单击菜单Window->Preferences，进入图1-5所示界面，选择你的Android SDK解压后的目录或者安装目录。

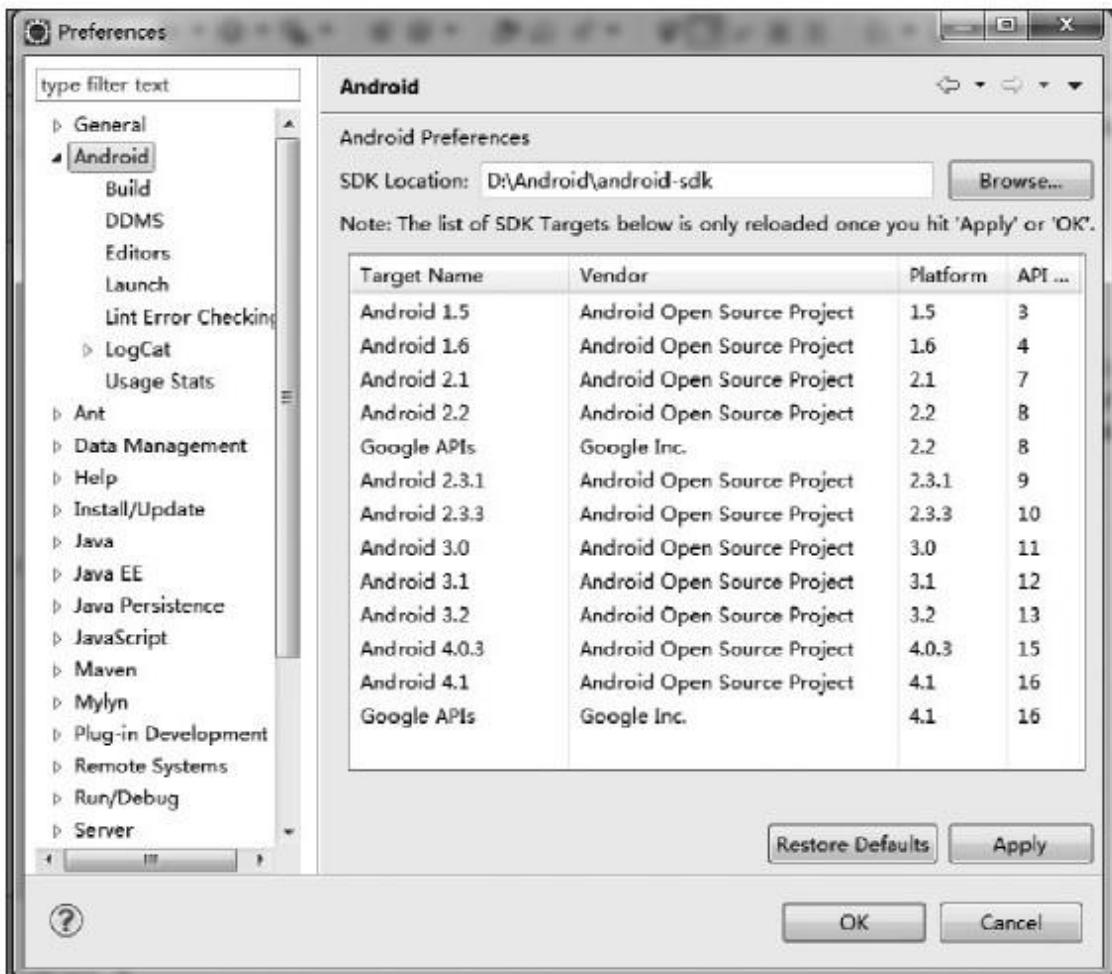


图 1-5 选择目录

步骤7 新建AVD (Android Virtual Device)。单击菜单Window->AVD Manager，进入图1-6所示的AVD管理器。

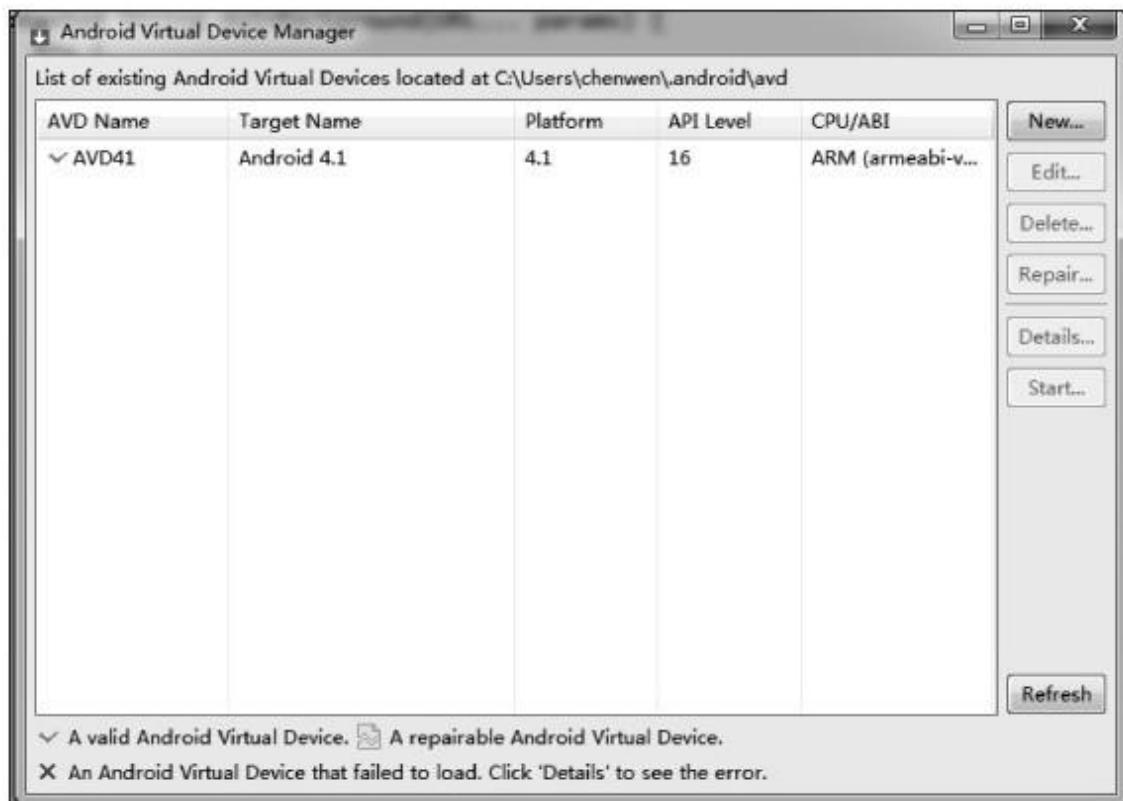


图 1-6 AVD管理器

步骤8 单击New按钮后，进入如图1-7所示的新建AVD详细界面。名称可以随便取，在Target下拉列表中选择你需要的SDK版本，SD卡大小自定义，单击Create AVD按钮，创建AVD完毕。

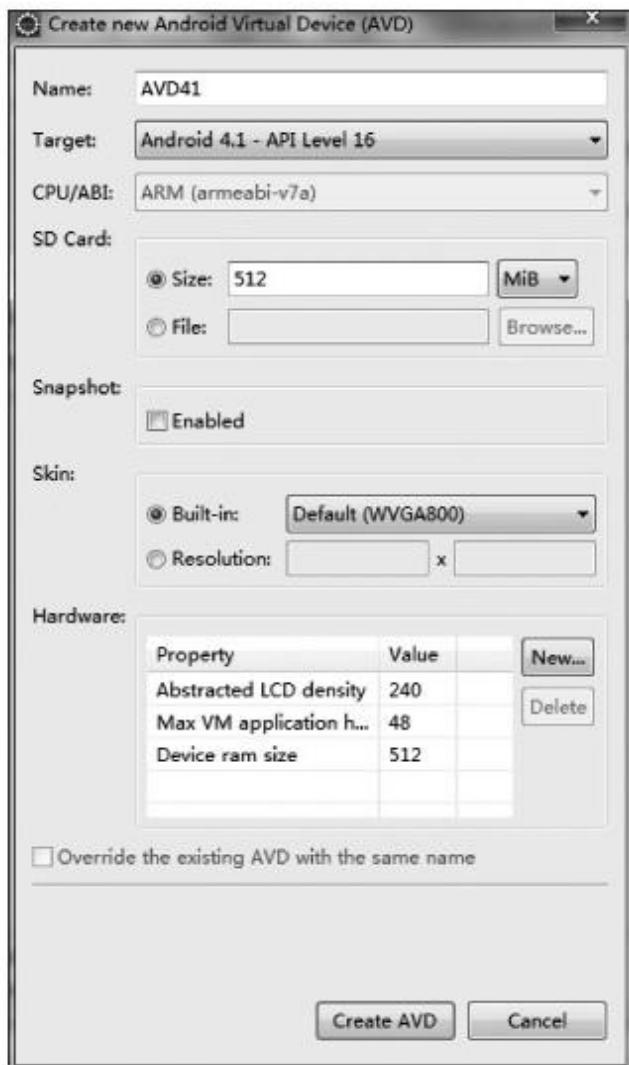


图 1-7 新建AVD详细界面

注意 如果需要添加（如GPS等）其他AVD选项，可以单击New按钮，从下拉菜单里面选择相应选项。

1.3.3 Hello World !

本节通过Hello World程序，介绍创建Android程序的过程。这段程序甚至不需要写一行代码，就可以全自动地创建好。

步骤1 打开Eclipse，选择菜单File->New->Android Application Project，显示如图1-8所示的创建Android程序界面。

在图1-8所示的Application Name文本框中输入应用名称，在Project Name文本框中输入项目名称，在Package Name文本框中输入包名，在Build SDK下拉列表中选择项目目标SDK版本，在Minimum Required SDK下拉列表中选择项目所需要的最小SDK版本。



图 1-8 新建Android界面

注意 如果该项目为其他项目的库，应勾选Mark this project as a library。

步骤2 单击Next按钮之后，显示的界面如图1-9所示，用户可以定义应用显示的图标。Image选择背景图片；Clipart选择系统的切图；Text中输入文字。Trim Surrounding Blank Space选项设置前景图是否自动充满背景图；Additional Padding设置前景图和背景图的显示比例；Foreground Scaling设定背景的尺寸，Crop为拉伸，Center为居中；

Shape设定背景图的形状，None设置背景为空，Square设为方形背景，Circle设为圆形背景；Background Color和Foreground Color分别设置显示图标的背景色和前景色。



图 1-9 选择应用图标

步骤3 单击Next按钮之后，进入的界面如图1-10所示。选择Activity的显示方式：BlankActivity创建空的Acitivity，这个和传统的创建方式类

似；MasterDetailFlow将创建带“碎片”的程序，可以重复利用较大的屏幕。



图 1-10 选择Activity类型

步骤4 单击Next按钮，填写Activity信息，进入的界面如图1-11所示。在Activity Name文本框中输入该Activity的名称；在Layout Name文本框中输入其对应的布局名称；在Navigation Type下拉列表中选择导航类

型，该下拉列表包括的选项有None、Tabs、Tabs+Swipes、Swipe Views+Title Strip和Dropdown，可以根据需要进行选择，本例子中选择None；在Hierarchical Parent中选择父节点，用以指定向上按键指向的界面；在Title文本框中指定显示的名称。

步骤5 单击Finish按钮进入如图1-12所示的布局编辑界面。



图 1-11 填写Activity信息

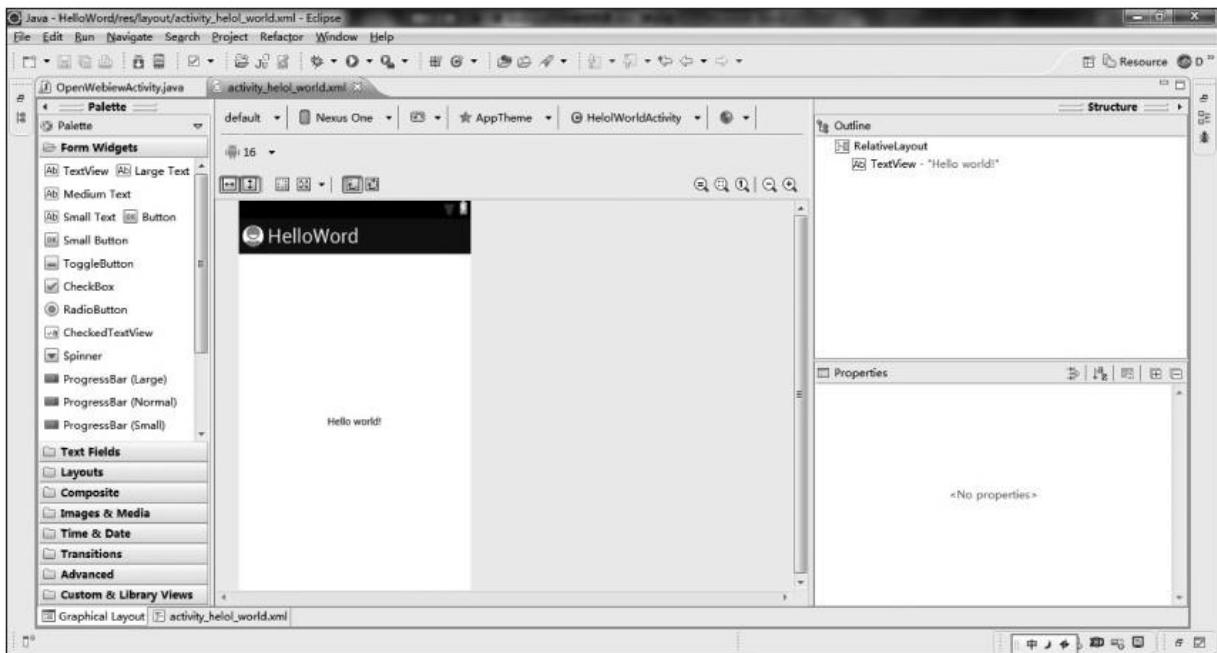


图 1-12 界面编辑

步骤6 切换到代码编辑界面，如图1-13所示。

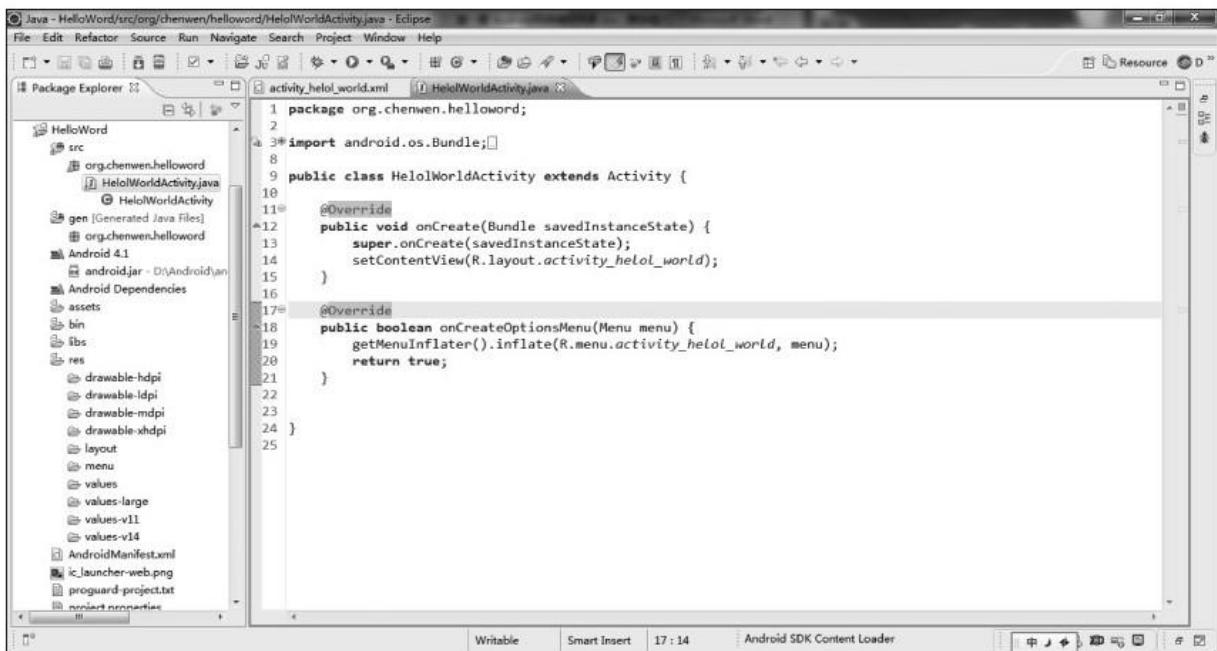


图 1-13 代码编辑界面

注意 若有错误，如Project.....is missing required source folder : 'gen'，则将gen->Android.Test->R.java这个文件删掉，Eclipse会重新生成这个文件，并且不会再报错。

步骤7 运行该项目前，需要进行一些设置，右击项目，在弹出的快捷菜单中依次单击Run as->Run Configuration，进入如图1-14所示界面，选择需要运行的项目。

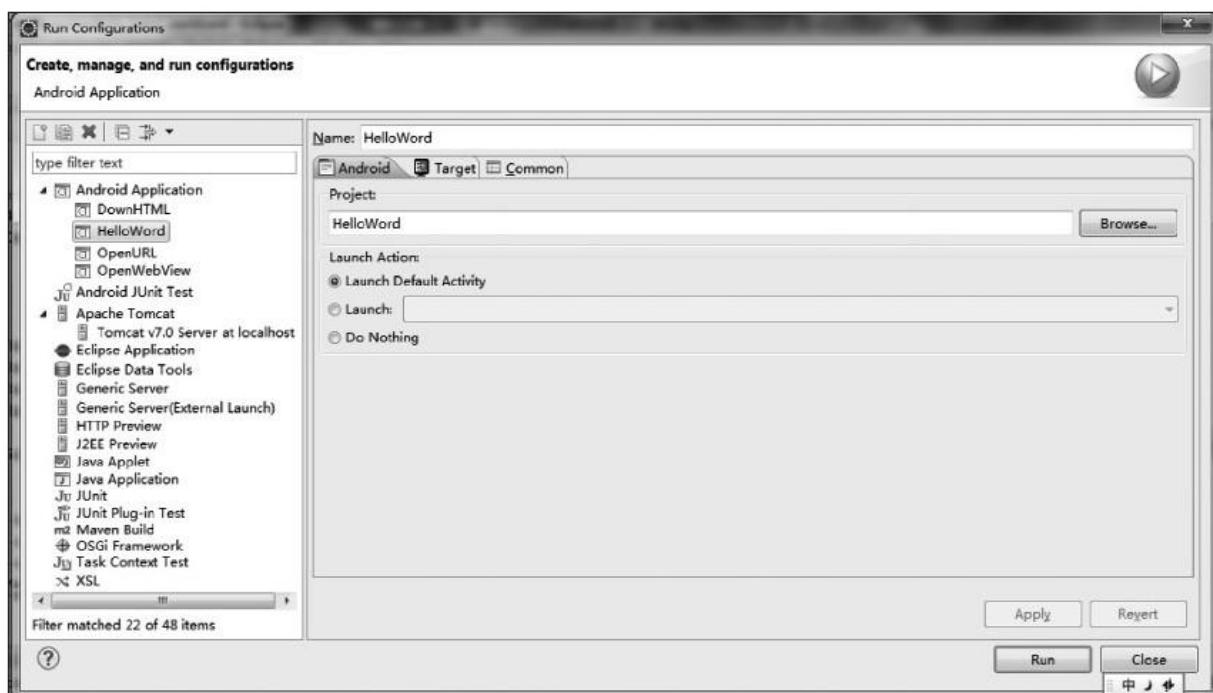


图 1-14 选择项目配置运行

选择Target，切换到如图1-15所示的界面，在对应的复选框中打钩，以选择希望运行的AVD。

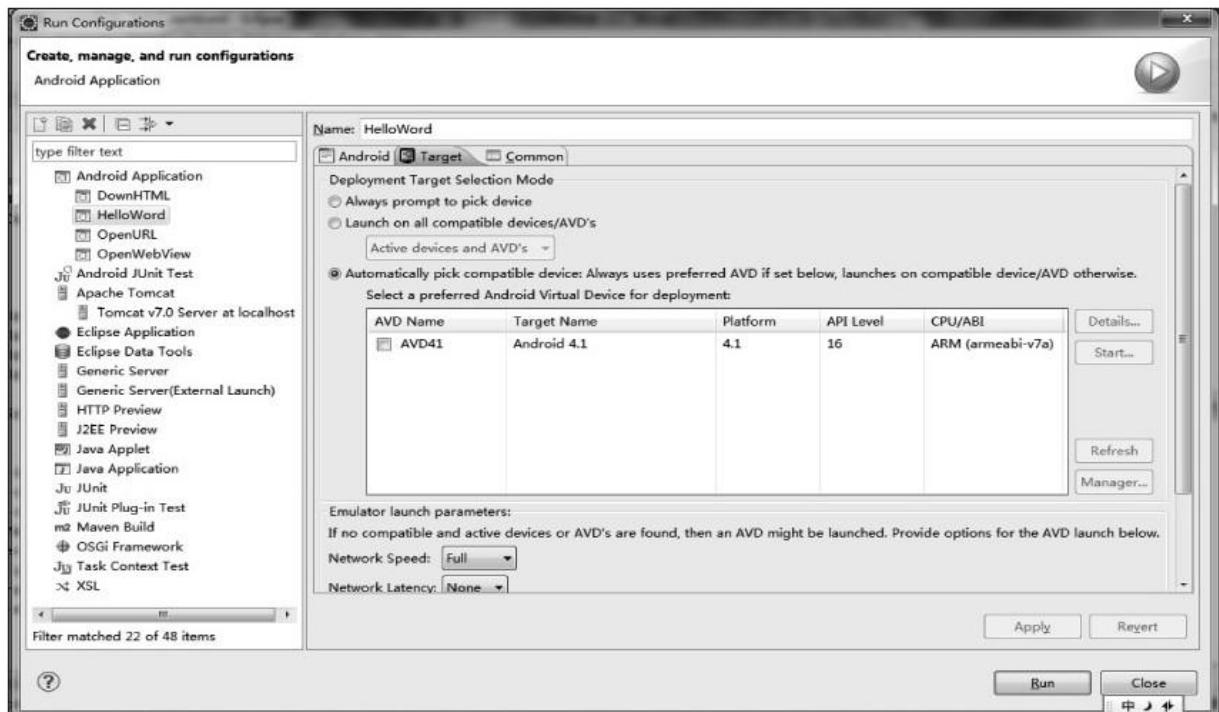


图 1-15 选择AVD运行目标

步骤8 单击Run按钮，就可以运行该项目了。图1-16所示为AVD正在加载的界面，模拟器启动后如图1-17所示。

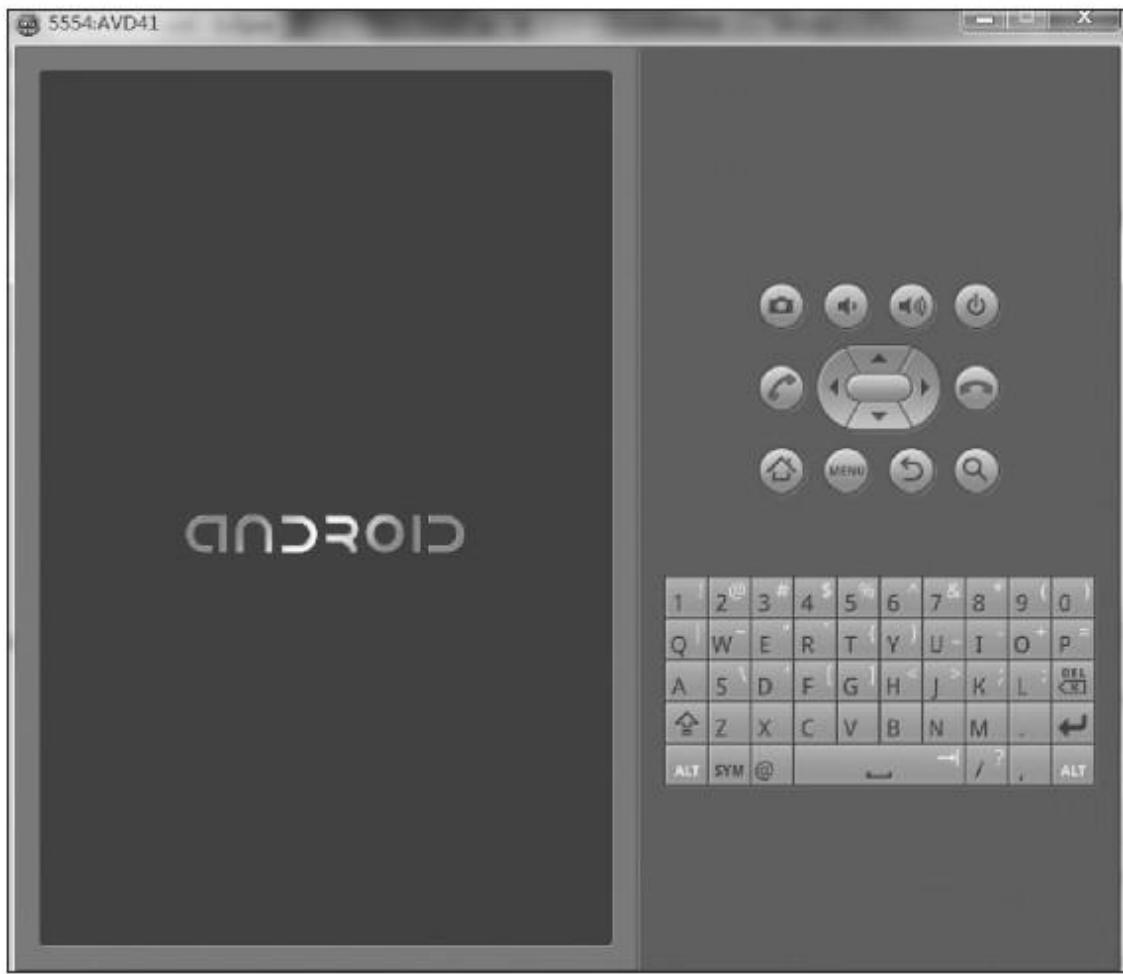


图 1-16 正在加载AVD

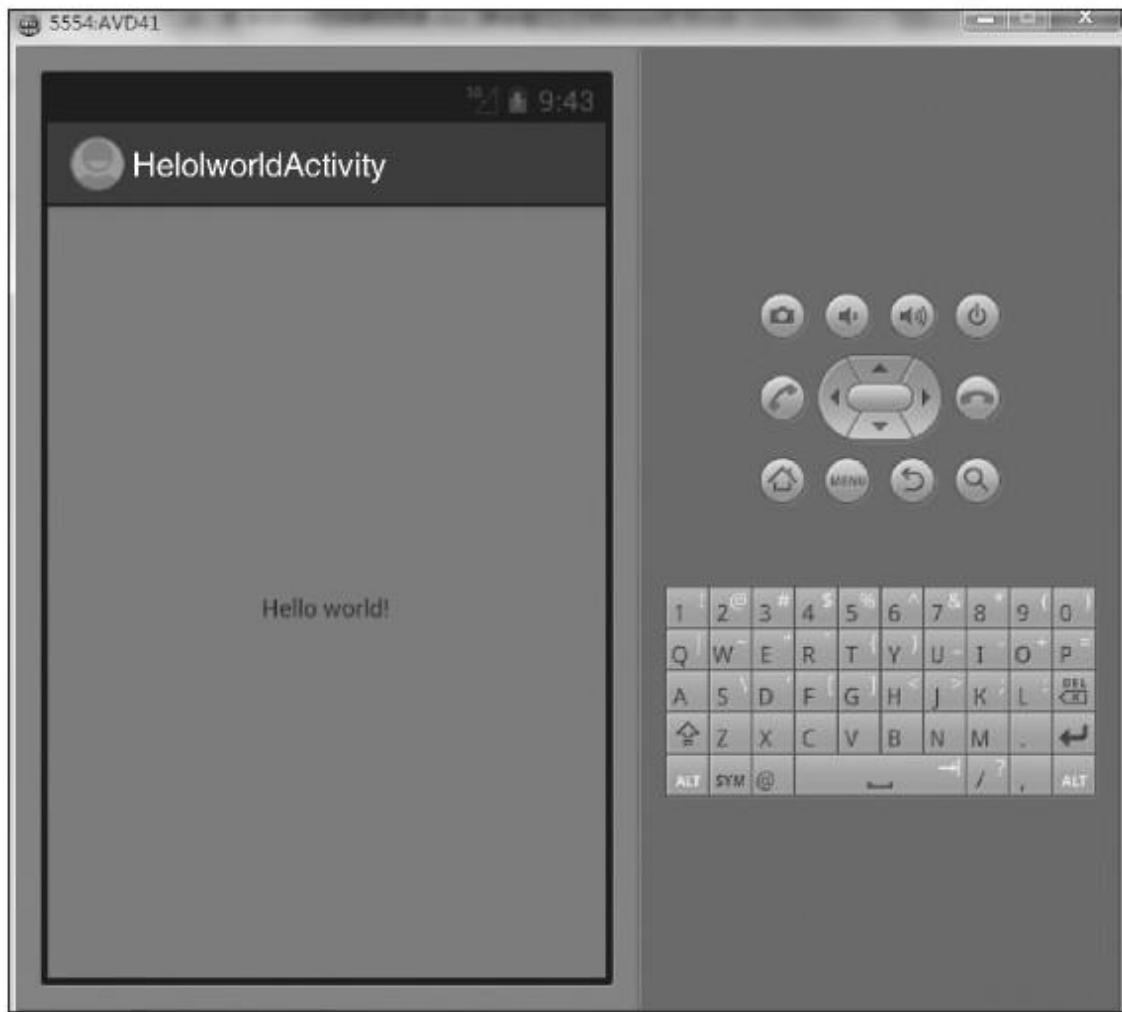


图 1-17 启动成功

1.4 网络应用实战案例

通过加载现有的Web页面来实现Android应用的功能是一种常见的方法。本节通过一个实际案例来展示如何加载一个现有页面，并在此页面的基础上增加需要的功能。

1.4.1 加载一个页面

如需在应用里面提供在线翻译的功能，最简单的方法就是给用户加载一个翻译的网站，用户进入该网站后自行输入想要翻译的单词，并查看翻译的结果。在界面上添加一个按钮，使得用户单击之后自动打开谷歌翻译网站。

步骤1 在Eclipse里面创建项目。

步骤2 添加字符串。在字符串文件\res\values\strings.xml里面（如下代码所示）添加需要的字符串，并指定显示的内容。

```
<string name="open_url">OpenURL</string>
```

步骤3 添加Button控件。打开res\layout\main.xml文件，在布局文件中添加Button按钮。代码如下：

```
<?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"/>  
  
<Button  
  
    android:id="@+id/open_ulr"  
  
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:text="@string/open_ulr"/>  
  
</LinearLayout>
```

步骤4 最简单的加载情况，调用系统的浏览器来打开一个页面，代码如下：

```
package org.chenwen.openurl ;  
  
import android.app.Activity ;  
  
import android.content.Intent ;  
  
import android.net.Uri ;  
  
import android.os.Bundle ;  
  
import android.view.View ;  
  
import android.widget.Button ;  
  
public class OpenURL extends Activity{  
  
    @Override  
  
    public void onCreate (Bundle savedInstanceState) {  
  
        super.onCreate (savedInstanceState) ;  
  
        setContentView (R.layout.main) ;  
  
        Button bt1= (Button) findViewById (R.id.button1) ;
```

```
bt1.setOnClickListener (new View.OnClickListener () {  
  
    @Override  
  
    public void onClick (View arg0) {  
  
        Uri uri=Uri.parse ("http://translate.google.cn/") ;  
  
        Intent intent=new Intent (Intent.ACTION_VIEW,uri) ;  
  
        startActivity (intent) ;  
  
    }  
  
}) ;  
  
}  
  
}
```

Uri.parse () 方法返回的是一个URI类型，通过这个URI可以访问一个网络上的或者本地的资源，Intent () 方法告诉系统调用哪个组件来打开这个URI。这里指明是使用Intent.ACTION_VIEW，这将调用系统里面的浏览器来打开指定的网页，如图1-18所示。这个网页会显示一个翻译网页。

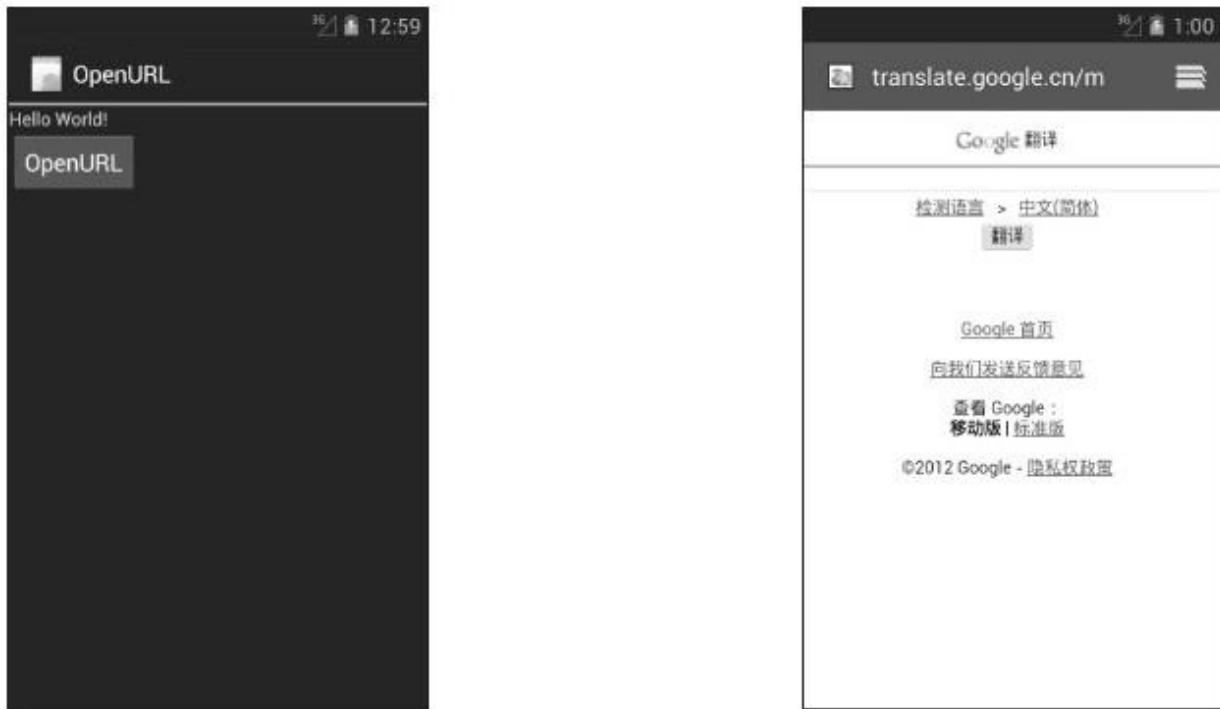


图 1-18 运行OpenURL应用和单击按钮OpenURL之后的效果

有时候，希望页面能够在应用内部打开，以方便添加一些需要的功能，这时可以在应用里面使用WebView控件。WebView是Android里面的浏览器组件，负责打开HTML文件，`setContent ()`方法动态的添加布局，`loadUrl ()`方法从网址加载一个页面，`loadData ()`和`loadDataWithBaseUrl ()`方法都是从字符串来加载一个页面。

重新建立一个项目，添加WebView控件到项目的主界面。

WebView打开页面的方法如下：

```
WebView wv= (WebView) findViewById (R.id.webView1) ;
```

```
wv.loadUrl ("http://translate.google.cn/m") ;
```

loadUrl的原型如下：

```
void loadData (String data,String mimeType,String encoding)
```

如果参数encoding的值为base64，则必须使用base64编码之后的数据。

其各参数含义如下。

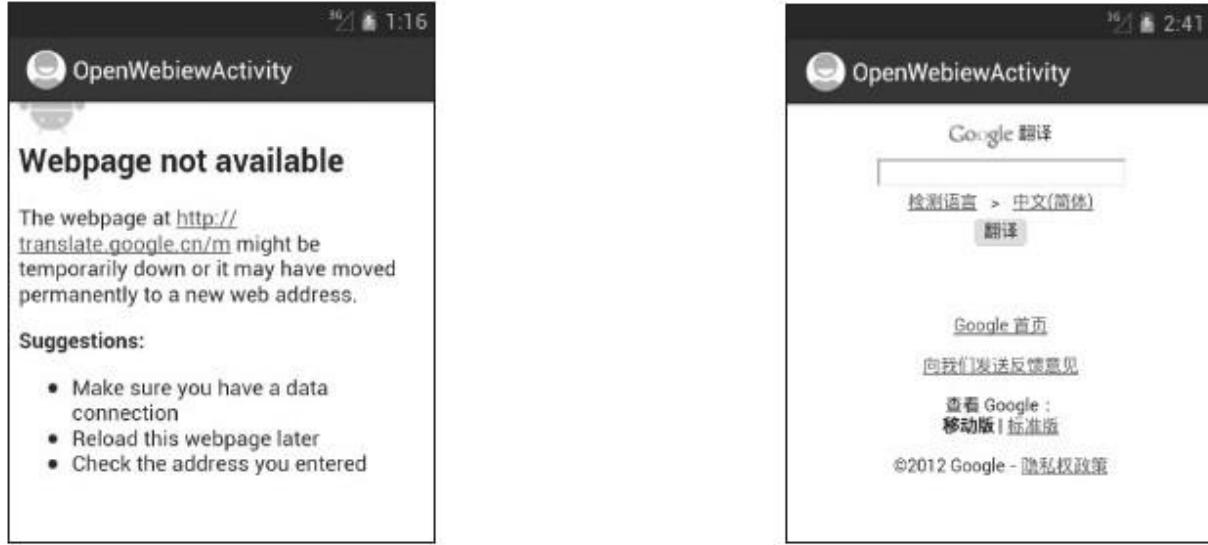
■data参数：数据字符串。

■mimeType参数：表明数据的MIME类型，如text/html。

■Encoding参数：数据的编码。

注意loadUrl方法在遇到错误网页的时候不会报出异常，且loadData方法不能处理js、https等格式的页面特效。如果需要检测页面异常，可以先对页面进行判定，使用loadDataWithBaseUrl可以加载https等特殊页面。

使用WebView控件后，运行应用，其效果如图1-19a所示。这是因为没有添加网络权限，在配置文件中添加权限后运行效果如图1-19b所示。



a) 使用 WebView 控件后

b) 添加权限后

图 1-19 WebView 打开页面

实现上述功能的主要代码如下所示：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="org.chenwen.openwebview"  
    android:versionCode="1"  
    android:versionName="1.0">  
  
<uses-sdk  
    android:minSdkVersion="8"  
    android:targetSdkVersion="15"/>
```

```
<uses-permission android:name="android.permission.INTERNET">
</uses-permission>

<application
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme">

    <activity
        android:name=".OpenWebviewActivity"
        android:label="@string/title_activity_open_webview">

        <intent-filter>

            <action android:name="android.intent.action.MAIN"/>

            <category android:name="android.intent.category.LAUNCHER"/>

        </intent-filter>

    </activity>

</application>
```

</manifest>

注意添加权限的位置，要在application标签之前，否则在某些高版本的编译器中不能通过，或者没有效果。

在翻译输入框里面，输入Android单击“翻译”按钮，其翻译结果如图1-20a所示。其翻译结果显示时，打开的页面已经超出了应用的控制范围了，如果希望其翻译结果仍然显示在该应用内部，可以使用WebView的setWebViewClient () 方法来解决这个问题。修改后翻译效果如图1-20b所示。



图 1-20 翻译效果

实现上述功能的主要代码如下所示：

```
WebView wv= (WebView) findViewById (R.id.webView1) ;  
  
wv.loadUrl ("http://translate.google.cn/m") ;  
  
wv.setWebViewClient (new WebViewClient () {  
  
    public boolean shouldOverrideUrlLoading (WebView view,String url) {  
  
        view.loadUrl (url) ;  
  
        return true ;  
  
    }  
  
}) ;
```

1.4.2 下载一个页面

为了更好地服务使用者，在很多情况下需要保存用户搜索过的翻译页面。要实现这个功能，需要保存加载的页面。首先通过网址生成URL对象，然后打开链接，写入Buffer中，最后写入字符串中，方便进一步的处理。实现相应功能的主要代码如下所示：

```
try{
```

```
URL newUrl=new URL ("http://translate.google.cn/m") ;  
  
URLConnection connect=newUrl.openConnection () ;  
  
DataInputStream dis=new DataInputStream (connect.getInputStream  
() ) ;  
  
BufferedReader in=new BufferedReader (new InputStreamReader  
(dis, "UTF-8") ) ;  
  
String html=""" ;  
  
String readLine=null ;  
  
while ( (readLine=in.readLine () ) !=null) {  
  
    html=html+readLine ;  
  
    Log.d ("OpenWebviewActivity", readLine) ;  
  
}  
  
in.close () ;  
  
}catch (MalformedURLException me) {  
  
}catch (IOException ioe) {
```

}

1.5 小结

本章介绍Android的发展、Android在网络方面的应用、Android开发的设置、如何对现有的页面进行简单的加载下载等内容。接下来我们会围绕Android网络编程从核心概念到实战案例，一步一步地来深入理解Android网络编程。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第2章 Android基本网络技术和编程实践

本章主要介绍计算机网络的相关概念，包括TCP/IP分层模型及IP、TCP、UDP等主要协议。在此基础上，重点阐述在Android中如何使用TCP和UDP进行通信。最后，给出Android Socket编程的一个实际案例。

2.1 计算机网络及其协议

网络原指用一个巨大的虚拟画面，把所有东西连接起来。计算机网络最初的目的也只是将各个独立的计算机连接起来，但是现在的计算机网络所能实现的已远远超出了人们早期的构想。在人们生活中，计算机网络已经无处不在，如网络视频、网络购物、网络教育等。

2.1.1 计算机网络概述

计算机网络就是用物理链路将各个孤立的工作站或主机连接在一起，组成数据链路，从而达到资源共享和通信的目的。凡将地理位置不同且具有独立功能的多个计算机系统通过通信设备和线路连接起来，并以功能完善的网络软件（网络协议、信息交换方式及网络操作系统等）实现网络资源共享的系统，均可称为计算机网络。简单地说，计算机网络即连接两台或多台计算机进行通信的系统。

计算机网络的功能主要表现在硬件资源共享、软件资源共享和用户间信息交换等方面。

- 硬件资源共享**：可以在全网范围内提供对处理资源、存储资源、输入输出资源等设备的共享，使用户节省投资，也便于集中管理和均衡分担负荷。

- 软件资源共享：允许互联网上的用户远程获得各类服务，如网络文件传送服务、远地进程管理服务和远程文件访问服务，从而避免软件研制上的重复劳动及数据资源的重复存储，也便于集中管理。
- 用户间信息交换：计算机网络为分布在各地的用户提供了强有力的通信手段。用户可以通过计算机网络传送电子邮件、发布新闻消息和进行电子商务活动。
- 提高计算机的可靠性和可用性：网络中的每台计算机都可通过网络相互成为后备机。一旦某台计算机出现故障，它的任务就可由其他的计算机代为完成，这样可以避免在单机情况下，一台计算机发生故障引起整个系统瘫痪的现象，从而提高系统的可靠性。而当网络中的某台计算机负担过重时，网络又可以将新的任务交给较空闲的计算机完成，均衡负载，从而提高了每台计算机的可用性。
- 分布式处理：通过算法将大型的综合性问题交给不同的计算机同时进行处理。用户可以根据需要合理选择网络资源，就近快速地进行处理。

计算机网络中用于规定信息的格式以及如何发送和接收信息的一套规则称为网络协议或通信协议。网络协议是为计算机网络中进行数据交换而建立的规则、标准或约定的集合。不同的计算机之间必须使用相同的网络协议才能进行通信。

2.1.2 网络协议概述

大多数网络都采用分层的体系结构，每一层都建立在它的下层之上，同时向它的上一层提供一定的服务，而把如何实现这一服务的细节对上一层加以屏蔽。一台设备上的第n层与另一台设备上的第n层进行通信的规则就是第n层协议。在网络的各层中存在着许多协议，接收方和发送方同层的协议必须一致，否则一方将无法识别另一方发出的信息。网络协议是网络上所有设备（网络服务器、计算机及交换机、路由器、防火墙等）之间通信规则的集合，它规定了通信时信息必须采用的格式和这些格式的意义。通过网络协议，网络上各种设备才能够相互交换信息。

由于网络节点之间联系的复杂性，在制定协议时，通常把复杂成分分解成一些简单成分，然后再将它们复合起来。最常用的复合技术就是采用层次的方法，网络协议的层次结构如下：

- 结构中的每一层都规定有明确的任务及接口标准。
- 把用户的应用程序作为最高层。
- 除了最高层外，中间的每一层都向上一层提供服务，同时又是下一层的用户。

■把物理通信线路作为最低层，它使用从最高层传送来的参数，是提供服务的基础。

为了使不同厂家生产的计算机能够相互通信，以便在更大的范围内建立计算机网络，国际标准化组织（ISO）在1978年提出了“开放系统互连参考模型”，即著名的OSI/RM模型（Open System Interconnection/Reference Model）。它将计算机网络体系结构的通信协议划分为七层，自下而上依次为：物理层、数据链路层、网络层、传输层、会话层、表示层、应用层。

其中低4层完成数据传送服务，上面3层面向用户。对于每一层，至少制定两项标准：服务定义和协议规范。前者给出了该层所提供的服务的准确定义，后者详细描述了该协议的动作和各种有关规程，以保证服务的提供。

■应用层：是开放系统互连环境的最高层。应用层为操作系统或网络应用程序提供访问网络服务的接口。

■表示层：为上层用户提供共同的数据或信息的语法表示变换。为了让采用不同编码方法的计算机在通信中能相互理解数据的内容，可以采用抽象的标准方法来定义数据结构，并采用标准的编码表示形式。表示层管理这些抽象的数据结构，并将计算机内部的表示形式转换成网

络通信中采用的标准表示形式。数据压缩和加密也是表示层可提供的表示变换功能。

- 会话层：也称会晤层，主要功能是组织和同步不同的主机上各种进程间的通信（称为对话），负责在两个会话层实体之间进行对话连接的建立和拆除。会话层还提供在数据流中插入同步点的机制，使得数据传输因网络故障而中断后，可以不必从头开始而是仅重传最近一个同步点以后的数据。
- 传输层：负责数据传送的最高层次。传输层完成同处于资源子网中的两个主机（即源主机和目的主机）间的连接和数据传输，也称为端到端的数据传输。
- 网络层：主要任务就是要选择合适的路由，使网络层的数据传输单元——分组能够正确无误地按照地址找到目的站。
- 数据链路层：负责在两个相邻的节点间的线路上无差错地传送以帧为单位的数据，每一帧包括一定的数据和必要的控制信息，在接收点接收到数据出错时要通知发送方重发，直到这一帧无误地到达接收节点。
- 物理层：定义了为建立、维护和拆除物理链路所需的机械的、电气的、功能的和规程的特性，其作用是使原始的数据比特流能在物理介质上传输。具体涉及接插件的规格、“0”或“1”信号的电平表示、收发双

方的协调等内容。物理层为上一层的数据链路层提供一个物理连接，通过物理连接透明地传输比特流。所谓透明传输是指经实际电路传送后的比特流没有变化，任意组合的比特流都可以在这个电路上传输，物理层并不知道比特的含义。

2.1.3 IP、TCP和UDP协议

由于OSI/RM模型过于复杂也难以实现，现实中广泛应用的是TCP/IP模型。TCP/IP是一个协议集，是由ARPA于1977年到1979年推出的一种网络体系结构和协议规范。随着Internet的发展，TCP/IP也得到进一步的研究开发和推广应用，成为Internet上的“通用语言”。

TCP/IP模型也是分层模型，分为4层。OSI/RM模型与TCP/IP模型的参考层次如图2-1所示。

OSI/RM模型	TCP/IP模型	TCP/IP协议集
应用层		
表示层	应用层	Telnet、FTP、SMTP、DNS、HTTP等
会话层		
传输层	传输层	TCP、UDP
网络层	网络层	IP、ARP、RARP、ICMP
数据链接层		
物理层	网络接口层	各种通信网络接口（以太网等）

图 2-1 OSI/RM模型与TCP/IP模型的参考层次

TCP/IP模型4层分述如下。

- **应用层**：应用层是大多数普通与网络相关的程序为了通过网络与其他程序通信所使用的层。在应用层中，数据以应用内部使用的格式进行传送，然后被编码成标准协议的格式。重要的例子如万维网使用的HTTP协议、文件传输使用的FTP协议、接收电子邮件使用的POP3和IMAP协议、发送邮件使用的SMTP协议，以及远程登录使用的SSH和Telnet等。所以用户通常是与应用层进行交互。

■传输层：传输层响应来自应用层的服务请求，并向网络层发出服务请求。传输层提供两台主机之间透明的数据传输，通常用于端到端连接、流量控制或错误恢复。这一层的两个最重要的协议是TCP（Transmission Control Protocol，传输控制协议）和UDP（User Datagram Protocol，用户数据报协议）。

■网络层：网络层提供端到端的数据包交付，换言之，它负责数据包从源发送到目的地，任务包括网络路由、差错控制和IP编址等。这一层包括的重要协议有IP（版本4和版本6）、ICMP（Internet Control Message Protocol, Internet控制报文协议）和IPSec（Internet Protocol Security, Internet协议安全）。

■网络接口层：是TCP/IP参考模型的最底层，负责通过网络发送和接收IP数据报；允许主机连入网络时使用多种现成的与流行的技术，如以太网、令牌网、帧中继、ATM、X.25、DDN、SDH、WDM等。

一个应用层应用一般都会使用到两个传输层协议之一：面向连接的TCP传输控制协议和面向无连接的UDP用户数据报协议。下面分析TCP/IP协议栈中常用的IP、TCP和UDP协议。

1.IP协议

互联网协议（Internet Protocol, IP）是用于报文交换网络的一种面向数据的协议。IP是在TCP/IP协议中网络层的主要协议，任务是根据源主

机和目的主机的地址传送数据。为达到此目的，IP定义了寻址方法和数据报的封装结构。第一个架构的主要版本，现在称为IPv4，仍然是最主要的互联网协议，如图2-2所示。当前世界各地正在积极部署IPv6。



图 2-2 IPv4封装结构

下面对IPv4协议包的结构进行介绍，包含多个数据域。各个数据域的含义如下。

- 4位版本：表示目前的协议版本号，数值是4表示版本为4，因现在主要使用的还是版本为4的IP协议，所以IP有时也称为IPv4。
- 4位首部长度：头部的长度，它的单位是32位（4字节），数值为5表示IP头部长度为20字节。

- 8位服务类型（TOS）：这个8位字段由3位的优先权子字段（现在已经被忽略）、4位的TOS子字段以及1位的未用字段（现在为0）构成。4位的TOS子字段包含最小延时、最大吞吐量、最高可靠性以及最小费用构成，对应位为1时指出上层协议对处理当前数据报所期望的服务质量。如果都为0，则表示是一般服务。
- 16位总长度（字节数）：总长度字段是指整个IP数据报的长度，以字节为单位。如数值为00 30，换算成十进制为48字节，48字节=20字节的IP头+28字节的TCP头。这个数据报只是传送的控制信息，还没有传送真正的数据，所以目前看到的总长度就是报头的长度。
- 16位标识：标识字段唯一标识主机发送的每一份数据报。
- 3位标志：该字段用于标记该报文是否分片，后面是否还有分片。
- 13位片偏移：指当前分片在原数据报中相对于用户数据字段的偏移量，即在原数据报中的相对位置。
- 8位生存时间（TTL）：TTL（Time-To-Live）生存时间字段设置了数据报可以经过的最多路由器数目。它指定了数据报的生存时间。TTL的初始值由源主机设置，一旦经过一个处理它的路由器，它的值就减去1。可根据TTL值判断服务器是什么系统和经过的路由器。举个例子，TTL的十六进制初始值为80H，换算成十进制为128，Windows操作系统的TTL初始值一般为80H,UNIX操作系统初始值为FFH。

- 8位协议：表示协议类型，6表示传输层是TCP协议。
- 16位头部检验和：当收到一份IP数据报后，同样对头部中的每个16位进行二进制反码的求和。由于接收方在计算过程中包含了发送方存在头部中的检验和，因此，如果头部在传输过程中没有发生任何差错，那么接收方计算的结果应该为全1。如果结果不是全1，即检验和错误，那么IP就丢弃收到的数据报，但是不生成差错报文，而是由上层发现丢失的数据报并进行重传。
- 32位源IP地址和32位目的IP地址：实际这是IPv4协议中核心的部分。32位的IP地址由一个网络ID和一个主机ID组成。源地址是指发送数据的源主机的IP地址，目的地址是指接收数据的目的主机的IP地址。
- 选项：长度不定，如果没有选项就表示这个字节的域等于0。
- 数据：该IPv4协议包负载的数据。

2.TCP协议

传输控制协议（Transmission Control Protocol,TCP）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

注意 IETF RFC 793是TCP的正式规范。其网址为
<http://tools.ietf.org/html/rfc793>。

在Internet协议族中，传输层是位于网络层之上、应用层之下的中间层。不同主机的应用层之间经常需要可靠的、像管道一样的连接，但是网络层不提供这样的流机制，其只能提供不可靠的包交换，所以传输层就自然出现了。

应用层向传输层发送用于网间传输的、用8位字节表示的数据流，然后TCP协议把数据流分成适当长度的报文段（通常受该计算机连接的网络的数据链路层的最大传送单元MTU的限制）。之后TCP协议把结果包传给网络层，由它来通过网络将包传送给接收端实体的传输层。TCP为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的包能按序接收。然后接收端实体为已成功收到的包发回一个相应的确认（ACK）；如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包（假设丢失了）将会被重传。TCP协议用一个校验和（Checksum）函数来检验数据是否有错误，在发送和接收时都要计算校验和。

TCP协议头最少长度为20个字节，其协议包结构及所包含的字段如图2-3所示。

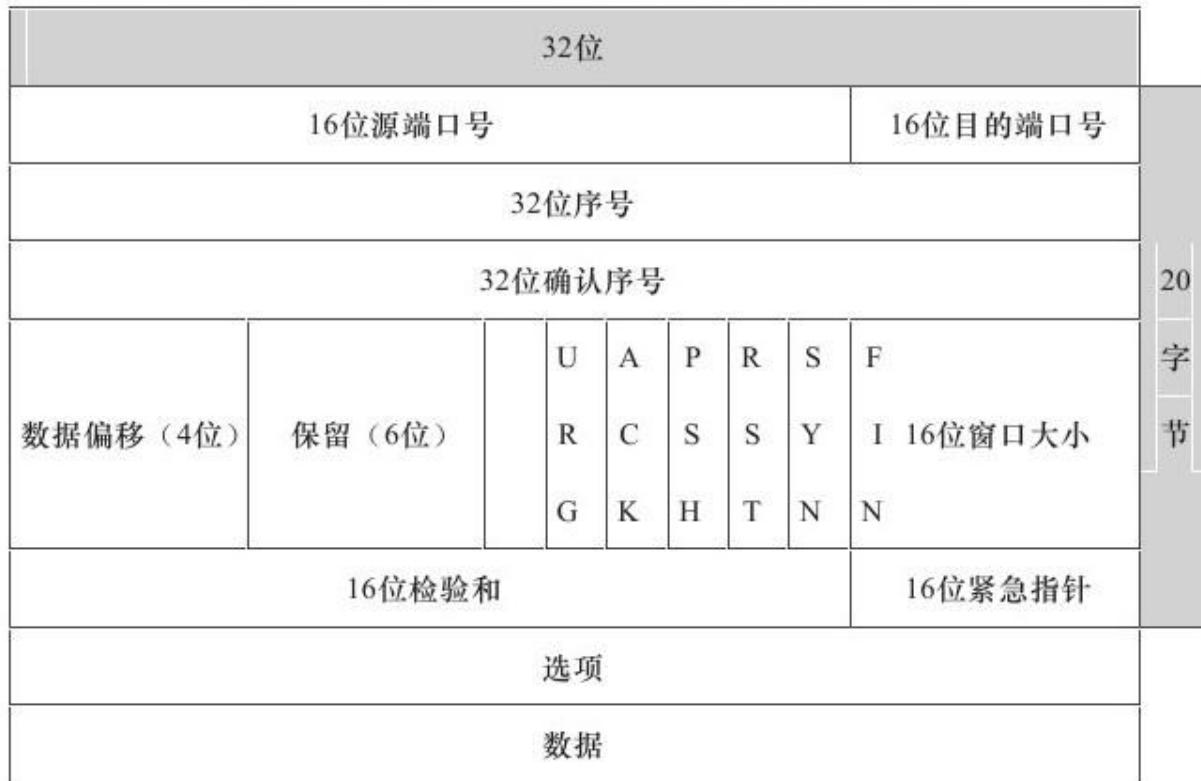


图 2-3 TCP数据包格式

各个字段的含义如下：

- 16位源端口号：源端口号是指发送数据的源主机的端口号，16位的源端口中包含初始化通信的端口。源端口和源IP地址的作用是标识报文的返回地址。
- 16位目的端口号：目的端口号是指接收数据的目的主机的端口号，16位的目的端口域定义传输的目的地。这个端口指明报文接收计算机上的应用程序地址端口。

■32位序号：TCP是面向字节流的，在一个TCP连接中传送的字节流中的每一个字节都按顺序编号。整个要传送的字节流的起始序号必须在连接建立时设置。首部中的序号字段值则指的是本报文段所发送的数据的第一个字节的序号。

■32位确认序号：是期望收到对方下一个报文段的第一个数据字节的序号，若确认号为N，则表明到序号N-1为止的所有数据都已正确收到。

■4位数据偏移：指出TCP报文段的数据起始处距离TCP报文段的起始处有多远，整个字段实际上指明了TCP报文段的首部长度。

■保留（6位）：为了将来定义新的用途而保留的位，但目前应置为0。

■URG、ACK、PSH、RST、SYN、FIN:6位标志域，依次对应为紧急标志、确认标志、推送标志、复位标志、同步标志、终止标志。

■16位窗口大小：指的是发送本报文段的一方的接收窗口，以便告诉对方，从本报文段首部中的确认号算起，接收方目前允许对方发送的数据量。因为接收方的数据缓存空间有限，该窗口值可作为接收方让发送方设置其发送窗口的依据。

■16位校验和：源机器基于数据内容计算一个数值，目的机器根据所接收到的数据内容也要计算出一个数值，这个数值要与源机器数值完全一样，从而证明数据的有效性。检验和字段检验的范围包括首部和数

据两部分。这是一个强制性的字段，一定是由发送端计算和存储，并由接收端进行验证的。

■16位紧急指针：在URG标志为1时其才有效，指出了本报文段中的紧急数据的字节数。

■选项：长度可变，最长可达40字节。当没有使用选项时，TCP首部长度是20字节。

■数据：该TCP协议包负载的数据。

在上述字段中，6位标志域中各标志的功能如下。

■URG：紧急标志。该位为1表示该位有效。

■ACK：确认标志。该位被置位时表示确认序号栏有效。大多数情况下该标志位是置位的。

■PSH：推送标志。该标志位置位时，接收端不将该数据进行队列处理，而是尽可能快地将数据转由应用处理。在处理Telnet或rlogin等交互模式的连接时，该标志位总是置位的。

■RST：复位标志。该位被置位时表示复位相应的TCP连接。

■SYN：同步标志。在连接建立时用来同步序号。当SYN=1时而ACK=0时，表明这是一个连接请求报文段。对方若同意建立连接，则应在响

应的报文段中是SYN=1和ACK=1。即SYN置位时表示这是一个连接请求或者连接接受报文。

■FIN：结束标志，用来释放一个连接。

3. UDP协议

用户数据报协议（UDP）是TCP/IP模型中一种面向无连接的传输层协议，提供面向事务的简单不可靠信息传送服务。UDP协议基本上是IP协议与上层协议的接口。UDP协议适用于端口分别运行在同一台设备上的多个应用程序中。

注意 IETF RFC 768是UDP的正式规范，其网址为
<http://tools.ietf.org/html/rfc768>。

与TCP不同，UDP并不提供对IP协议的可靠机制、流控制以及错误恢复功能等，在数据传输之前不需要建立连接。由于UDP比较简单，UDP头包含很少的字节，所以比TCP负载消耗少。

UDP适用于不需要TCP可靠机制的情形，比如，当高层协议或应用程序提供错误和流控制功能的时候。UDP服务于很多知名应用层协议，包括网络文件系统（Network File System,NFS）、简单网络管理协议（Simple Network Management Protocol,SNMP）、域名系统（Domain

Name System,DNS) 以及简单文件传输系统 (Trivial File Transfer Protocol,TFTP) 。

UDP数据报格式包含的字段如图2-4所示。

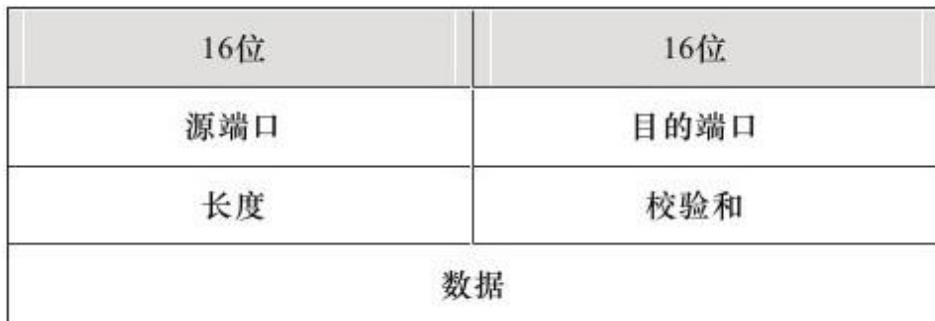


图 2-4 UDP数据报格式

- 源端口：16位。源端口是可选字段。当使用时，它表示发送程序的端口，同时它还被认为是没有其他信息的情况下需要被寻址的答复端口。如果不使用，设置其值为0。
- 目的端口：16位。目标端口在特殊互联网目标地址的情况下具有意义。
- 长度：16位。UDP用户数据报的总长度。
- 校验和：16位。用于校验UDP数据报的UDP头部和UDP数据。
- 数据：包含上层数据信息。

2.2 在Android中使用TCP、 UDP协议

上面介绍了TCP和UDP的报文结构，每段报文里面除了数据本身，还包含了包的目的地址和端口、包的源地址和端口以及其他各种附加的校验信息。这些包的长度是有限的，传输的时候需要将其分解为多个包，在到达传输的目的地址后再组合还原。如包有丢失或者破坏需要重传时，则乱序发送的包在达到时需要重新排序。处理这些过程是一项繁杂的工作，需要大量可靠的代码来完成。为了使程序员不必费心于上述这些底层具体细节，人们通过Socket对网络纠错、包大小、包重传等进行了封装。

2.2.1 Socket基础

Socket通常称为“套接字”。Socket字面上的中文意思为“插座”。一台服务器可能会提供很多服务，每种服务对应一个Socket（也可以这么说，每个Socket就是一个插座，客户若是需要哪种服务，就将插头插到相应的插座上面），而客户的“插头”也是一个Socket。Socket是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。Socket把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。Socket用于描述IP

地址和端口，是一个通信链的句柄。应用程序通常通过套接字向网络发出请求或者应答网络请求。

Socket的基本操作包括：

- 连接远程机器
- 发送数据
- 接收数据
- 关闭连接
- 绑定端口
- 监听到达数据
- 在绑定的端口上接受来自远程机器的连接

服务器要和客户端通信，两者都要实例化一个Socket。服务器和客户端的Socket是不一样的，客户端可以实现连接远程机器、发送数据、接收数据、关闭连接等，服务器还需要实现绑定端口，监听到达的数据，接受来自远程机器的连接。Android在包java.net里面提供了两个类：ServerSocket和Socket，前者用于实例化服务器的Socket，后者用于实例化客户端的Socket。在连接成功时，应用程序两端都会产生一个Socket实例，操作这个实例，完成客户端到服务器所需的会话。

接下来分析一些重要的Socket编程接口。首先是如何构造一个Socket，常用的构造客户端Socket的方法有以下几种。

- `Socket ()` : 创建一个新的未连接的Socket。
- `Socket (Proxy proxy)` : 使用指定的代理类型创建一个新的未连接的Socket。
- `Socket (String dstName,int dstPort)` : 使用指定的目标服务器的IP地址和目标服务器的端口号，创建一个新的Socket。
- `Socket (String dstName,int dstPort,InetAddress localAddress,int localPort)` : 使用指定的目标主机、目标端口、本地地址和本地端口，创建一个新的Socket。
- `Socket (InetAddress dstAddress,int dstPort)` : 使用指定的本地地址和本地端口，创建一个新的Socket。
- `Socket (InetAddress dstAddress,int dstPort,InetAddress localAddress,int localPort)` : 使用指定的目标主机、目标端口、本地地址和本地端口，创建一个新的Socket。

其中，`proxy`为代理服务器地址，`dstAddress`为目标服务器IP地址，`dstPort`为目标服务器的端口号（因为服务器的每种服务都会绑定在一个

端口上面），dstName为目标服务器的主机名。Socket构造函数代码举例如下所示：

```
Socket client=new Socket ("192.168.1.23", 2012) ;
```

//第一个参数是目标服务器的IP地址，2012是目标服务器的端口号

```
Socket sock=new Socket (new Proxy (Proxy.Type.SOCKS,
```

```
new InetSocketAddress ("test.domain.org", 2130) ) ) ;
```

//实例化一个Proxy，以该Proxy为参数，创建一个新的Socket

注意0~1023端口为系统保留，用户设定的端口号应该大于1023。

Socket类的重要方法如表2-1所示。

表 2-1 Socket 类重要方法

方法原型	功能描述
Public InputStream getInputStream()	读出该Socket中的数据
public OutputStream getOutputStream()	向该Socket中写入数据
public synchronized void close()	关闭该Socket

上面是构造Socket客户端的几种常用的方法，构造服务器端的ServerSocket的方法有以下几种。

- ServerSocket ()：构造一个新的未绑定的ServerSocket。

■`ServerSocket (int port)` : 构造一个新的`ServerSocket`实例并绑定到指定端口。如果`port`参数为0，端口将由操作系统自动分配，此时进入队列的数目将被设置为50。

■`ServerSocket (int port,int backlog)` : 构造一个新的`ServerSocket`实例并绑定到指定端口，并且指定进入队列的数目。如果`port`参数为0，端口将由操作系统自动分配。

■`ServerSocket (int port,int backlog,InetAddress localAddress)` : 构造一个新的`ServerSocket`实例并绑定到指定端口和指定的地址。如果`localAddress`参数为null，则可以使用任意地址，如果`port`参数为0，端口将由操作系统自动分配。

下面举例说明`ServerSocket`的构建方法，代码如下所示：

```
ServerSocket socketserver=new ServerSocket (2012) ;
```

//2012表示服务器要监听的端口号

构造完`ServerSocket`之后，需要调用`ServerSocket.accept ()`方法来等待客户端的请求（因为`Socket`都是绑定在端口上面的，所以知道是哪个客户端请求的）。`accept ()`方法就会返回请求这个服务的客户端的`Socket`实例，然后通过返回的这个`Socket`实例的方法，操作传输过来的信息。当`Socket`对象操作完毕之后，使用`close ()`方法将其关闭。

ServerSocket类的重要方法如表2-2所示。

表 2-2 ServerSocket 类重要方法

方法原型	功能描述
public Socket accept ()	等待 Socket 请求，直到连接被打开，该方法返回一个刚刚打开的连接 Socket 对象
public void close()	关闭该服务器 Socket

Socket一般有两种类型：TCP套接字和UDP套接字。

TCP和UDP在传输过程中的具体实现方法不同。两者都接收传输协议数据包并将其内容向前传送到应用层。TCP把消息分解成数据包（数据报，datagrams）并在接收端以正确的顺序把它们重新装配起来，TCP还处理对遗失数据包的重传请求，位于上层的应用层要处理的事情就少多了。UDP不提供装配和重传请求这些功能，它只是向前传送信息包。位于上层的应用层必须确保消息是完整的，并且是以正确的顺序装配的。

接下来我们分别详细讨论使用TCP和UDP通信的一些细节。

2.2.2 使用TCP通信

TCP建立连接之后，通信双方都同时可以进行数据的传输；在保证可靠性上，采用超时重传和捎带确认机制；在流量控制上，采用滑动窗口协议，协议中规定，对于窗口内未经确认的分组需要重传；在拥塞控制上，采用慢启动算法。TCP通信的原理示意图如图2-5所示。

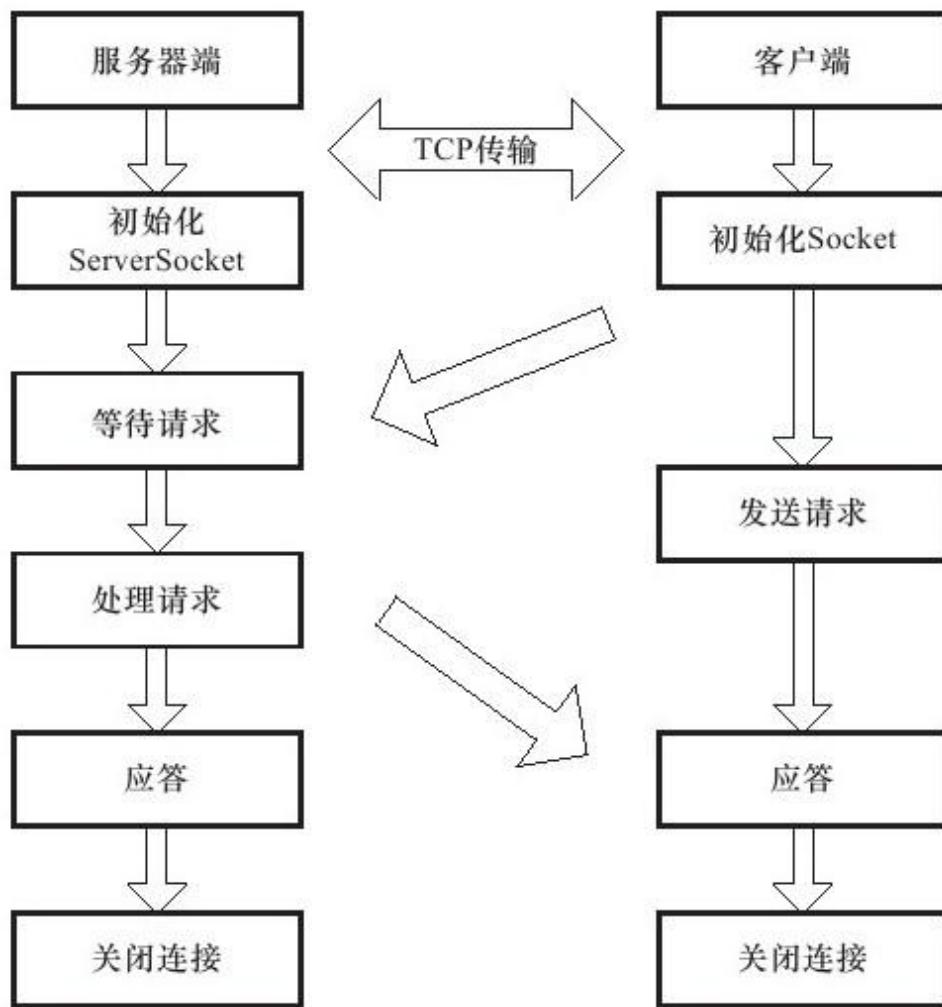


图 2-5 TCP通信原理示意图

TCP服务器端工作的主要步骤如下。

步骤1 调用ServerSocket (int port) 创建一个ServerSocket，并绑定到指定端口上。

步骤2 调用accept ()，监听连接请求，如果客户端请求连接，则接受连接，返回通信套接字。

步骤3 调用Socket类的getOutputStream () 和getInputStream () 获取输出和输入流，开始网络数据的发送和接收。

步骤4 关闭通信套接字。

示例代码如下所示：

```
//创建一个ServerSocket对象  
ServerSocket serverSocket=null ;  
  
try{  
  
    //TCP_SERVER_PORT为指定的绑定端口，为int类型  
    serverSocket=new ServerSocket (TCP_SERVER_PORT) ;  
  
    //监听连接请求  
  
    Socket socket=serverSocket.accept () ;  
  
    //写入读Buffer中  
  
    BufferedReader in=new BufferedReader (new  
  
    //获取输入流  
  
    InputStreamReader (socket.getInputStream () ) ) ;
```

//放到写Buffer中

```
BufferedWriter out=new BufferedWriter (new
```

//获取输出流

```
OutputStreamWriter (socket.getOutputStream () ) ) ;
```

//读取接收信息，转换为字符串

```
String incomingMsg=in.readLine () +System.getProperty  
("line.separator") ;
```

//生成发送字符串

```
String outgoingMsg="goodbye from port"+TCP_SERVER_PORT+  
System.getProperty ("line.separator") ;
```

//将发送字符串写入上面定义的BufferedWriter中

```
out.write (outgoingMsg) ;
```

//刷新，发送

```
out.flush () ;
```

//关闭

```
socket.close () ;  
  
}catch (InterruptedIOException e) {  
  
    //超时错误  
  
    e.printStackTrace () ;  
  
    //IO异常  
  
}catch (IOException e) {  
  
    //打印错误  
  
    e.printStackTrace () ;  
  
}  
  
}finally{  
  
    //判定是否初始化ServerSocket对象，如果初始化则关闭serverSocket  
  
    if (serverSocket !=null) {  
  
        try{  
  
            serverSocket.close () ;  
  
        }catch (IOException e) {  
    }  
}
```

```
e.printStackTrace () ;
```

```
}
```

```
}
```

```
}
```

TCP客户端工作的主要步骤如下。

步骤1 调用Socket () 创建一个流套接字，并连接到服务器端。

步骤2 调用Socket类的getOutputStream () 和getInputStream () 方法获取输出和输入流，开始网络数据的发送和接收。

步骤3 关闭通信套接字。

编写TCP客户端代码如下所示：

```
try{
```

```
//初始化Socket,TCP_SERVER_PORT为指定的端口，int类型
```

```
Socket socket=new Socket ("localhost", TCP_SERVER_PORT) ;
```

```
//获取输入流
```

```
BufferedReader in=new BufferedReader (new
```

```
InputStreamReader (socket.getInputStream () ) ) ;  
  
//生成输出流  
  
BufferedWriter out=new BufferedWriter (new  
OutputStreamWriter (socket.getOutputStream () ) ) ;  
  
//生成输出内容  
  
String outMsg="TCP connecting to"+TCP_SERVER_PORT+  
System.getProperty ("line.separator") ;  
  
//写入  
  
out.write (outMsg) ;  
  
//刷新，发送  
  
out.flush () ;  
  
//获取输入流  
  
String inMsg=in.readLine () +System.getProperty ("line.separator") ;  
  
Log.i ("TcpClient", "received："+inMsg) ;
```

```
//关闭连接  
  
socket.close () ;  
  
}catch (UnknownHostException e) {  
  
e.printStackTrace () ;  
  
}catch (IOException e) {  
  
e.printStackTrace () ;  
  
}  

```

注意 无论是客户端还是服务器端，都要添加UnknownHostException（处理网络异常） 和IOException（处理I/O异常） 异常处理。

在Android配置文件中需要添加下面的权限：

```
<uses-permission android:name="android.permission.INTERNET"/>
```

2.2.3 使用UDP通信

UDP有不提供数据报分组、组装和不能对数据包排序的缺点，也就是说，当报文发送之后，是无法得知其是否安全完整到达的。UDP用来支持那些需要在计算机之间传输数据的网络应用，包括网络视频会议系统在内的众多的客户端/服务器模式的网络应用都需要使用UDP协

议。UDP协议的主要作用是将网络数据流量压缩成数据报的形式。一个典型的数据报就是一个二进制数据的传输单位。UDP传输原理示意
图如图2-6所示。

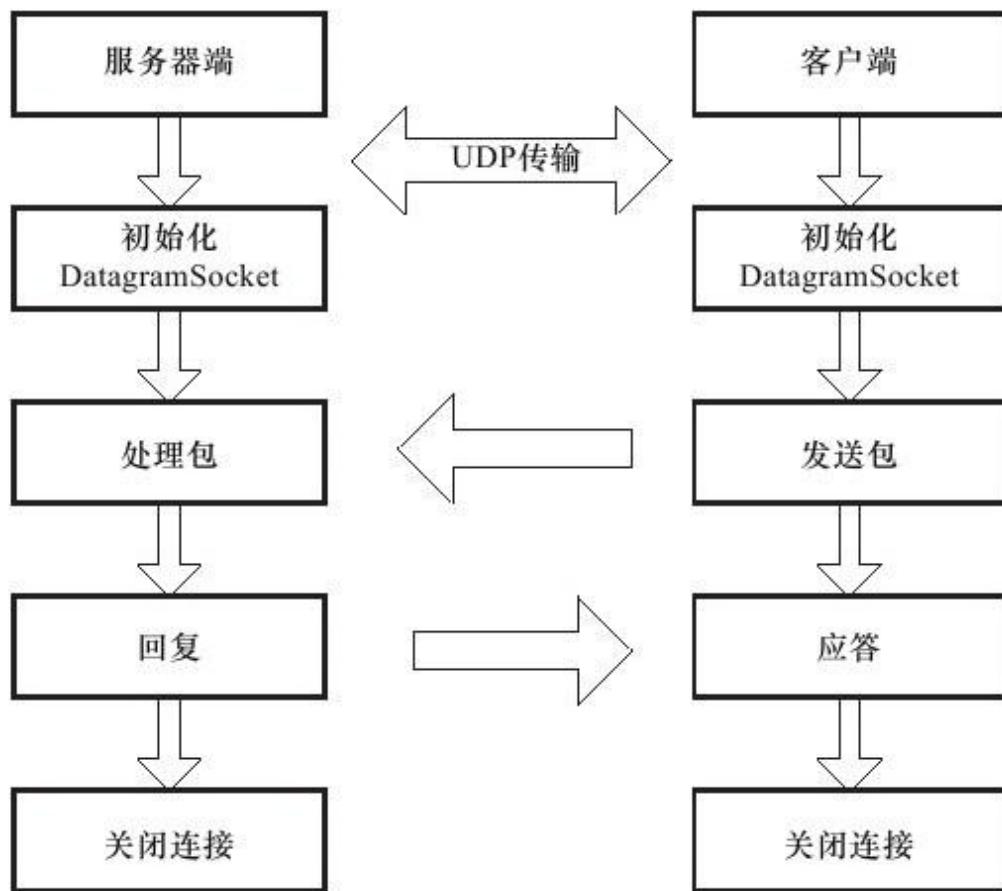


图 2-6 UDP 传输原理示意图

UDP服务器端工作的主要步骤如下。

步骤1 调用DatagramSocket (int port) 创建一个数据报套接字，并绑定到指定端口上。

步骤2 调用DatagramPacket (byte[]buf,int length) , 建立一个字节数组以接收UDP包。

步骤3 调用DatagramSocket类的receive () , 接受UDP包。

步骤4 关闭数据报套接字。

示例代码如下所示：

```
//接收的字节大小，客户端发送的数据不能超过
```

```
MAX_UDP_DATAGRAM_LEN
```

```
byte[]lMsg=new byte[MAX_UDP_DATAGRAM_LEN]；
```

```
//实例化一个DatagramPacket类
```

```
DatagramPacket dp=new DatagramPacket (lMsg,lMsg.length) ；
```

```
//新建一个DatagramSocket类
```

```
DatagramSocket ds=null ；
```

```
try{
```

```
//UDP服务器监听的端口
```

```
ds=new DatagramSocket (UDP_SERVER_PORT) ；
```

```
//准备接收数据

ds.receive (dp) ;

}catch (SocketException e) {

e.printStackTrace () ;

}catch (IOException e) {

e.printStackTrace () ;

}finally{

//如果ds对象不为空，则关闭ds对象

if (ds !=null) {

ds.close () ;

}

}

}
```

UDP客户端工作的主要步骤如下。

步骤1 调用DatagramSocket () 创建一个数据包套接字。

步骤2 调用DatagramPacket (byte[]buf,int offset,int length,InetAddress address,int port) , 建立要发送的UDP包。

步骤3 调用DatagramSocket类的send () 发送UDP包。

步骤4 关闭数据报套接字。

示例代码如下所示：

//定义需要发送的信息

```
String udpMsg="hello world from UDP client"+UDP_SERVER_PORT ;
```

//新建一个DatagramSocket对象

```
DatagramSocket ds=null ;
```

```
try{
```

//初始化DatagramSocket对象

```
ds=new DatagramSocket () ;
```

//初始化InetAddress对象

```
InetAddress serverAddr=InetAddress.getByName ("127.0.0.1") ;
```

```
DatagramPacket dp ;
```

```
//初始化DatagramPacket对象  
  
dp=new Datagram  
  
Packet (udpMsg.getBytes () , udpMsg.length () ,  
serverAddr,UDP_SERVER_PORT) ;
```

//发送

```
ds.send (dp) ;
```

```
}
```

//异常处理

//Socket连接异常

```
catch (SocketException e) {
```

```
    e.printStackTrace () ;
```

//不能连接到主机

```
}catch (UnknownHostException e) {
```

```
    e.printStackTrace () ;
```

//数据流异常

```
 }catch (IOException e) {  
     e.printStackTrace ();  
  
 //其他异常  
  
 }catch (Exception e) {  
     e.printStackTrace ();  
  
 }finally{  
  
 //如果DatagramSocket已经实例化，则需要将其关闭  
  
 if (ds !=null) {  
     ds.close ();  
  
 }  
 }  
 }
```

2.3 Socket实战案例

2.3.1 Socket聊天举例

本节以Socket聊天为例，实现了从服务器端到客户端的聊天功能。

服务器端任务如下：ChatServer类负责开启Server端服务；ReceiveMsg负责接收消息；SendMsg负责发送消息；Server端响应请求，向Client端返回数据。其主要代码如下所示：

```
//Socket聊天服务器端  
  
//继承Thread类，创建一个线程  
  
public class ChatServer extends Thread{  
  
    private ChatServer () throws IOException{  
  
        //创建Socket服务器  
  
        CreateSocket () ;  
  
    }  
  
    //重写run () 方法
```

```
public void run () {  
  
    Socket client ;  
  
    //定义接收的文本  
  
    String txt ;  
  
    try{  
  
        //始终在监听  
  
        while (true) {  
  
            //开启线程， 实时监听socket端口  
  
            //获取应答消息  
  
            client=ResponseSocket () ;  
  
            //响应客户端连接请求  
  
            while (true) {  
  
                //接收客户端消息  
  
                txt=ReceiveMsg (client) ;
```

```
System.out.println (txt) ;  
  
//连接获得客户端发来消息  
  
//并将其显示在Server端的屏幕上  
  
SendMsg (client,txt) ;  
  
//向客户端返回消息  
  
if (true) break ;  
  
//中断，继续等待连接请求  
  
}  
  
//关闭此次连接  
  
CloseSocket (client) ;  
  
}  
  
}  
  
catch (IOException e) {  
  
System.out.println (e) ;
```

}

}

//定义一个ServerSocket对象

private ServerSocket server=null ;

//定义端口号

private static final int PORT=5000 ;

//定义写Buffer

private BufferedWriter writer ;

//定义读Buffer

private BufferedReader reader ;

//实例化ServerSocket

private void CreateSocket () throws IOException{

server=new ServerSocket (PORT, 100) ;

}

//接收消息

```
private Socket ResponseSocket () throws IOException{
```

```
    Socket client=server.accept () ;
```

```
    return client ;
```

```
}
```

//关闭打开的连接和缓存

```
private void CloseSocket (Socket socket) throws IOException{
```

```
    reader.close () ;
```

```
    writer.close () ;
```

```
    socket.close () ;
```

```
}
```

//封装发送消息的方法

```
private void SendMsg (Socket socket,String Msg) throws IOException{
```

//要发送的消息写入BufferedWriter中

```
writer=new BufferedWriter (
    new OutputStreamWriter (socket.getOutputStream () ) ) ;
//添加空行分隔符
writer.write (Msg+"\n") ;
//刷新，发送
writer.flush () ;
}
```

//接收消息的方法

```
private String ReceiveMsg (Socket socket) throws IOException{
```

//保存到读Buffer

```
reader=new BufferedReader (
```

//将接收到的信息写入缓冲区

```
new InputStreamReader (socket.getInputStream () ) ) ;

```

//将接收到的信息保存到字符串中

```
String txt="Sever send :" +reader.readLine () ;
```

//以字符串的方式返回接收到的信息

```
return txt ;
```

```
}
```

```
}
```

启动服务器的代码如下所示：

```
//创建聊天服务器
```

```
ChatServer chatserver=new ChatServer () ;
```

//检测服务器是否已经启动，如果没有则启动服务器

```
if (chatserver !=null) {
```

```
    chatserver.start () ;
```

```
}
```

客户端的任务如下：

客户端发起连接请求，并向服务器端发送数据；客户端接收服务器端的数据。其主要代码如下所示：

//客户端获取消息类

```
private Socket RequestSocket (String host,int port) throws  
UnknownHostException,IOException{
```

//新建Socket类

```
Socket socket=new Socket (host,port) ;
```

```
return socket ;
```

```
}
```

//客户端发送消息类

```
private void SendMsg (Socket socket,String msg) throws IOException{
```

//将要发送的消息写入Buffer中

```
BufferedWriter writer=new BufferedWriter (new
```

```
OutputStreamWriter (socket.getOutputStream () ) ) ;
```

//格式转换

```
writer.write (msg.replace ("\n", "") +"\n") ;
```

//刷新，发送

```
writer.flush () ;  
}  
  
//客户端接收消息  
  
private String ReceiveMsg (Socket socket) throws IOException{  
  
    //写入读Buffer中  
  
    BufferedReader reader=new BufferedReader (new  
  
    //获取接收的消息到数据流中  
  
    InputStreamReader (socket.getInputStream () ) ) ;  
  
    //读取消息到字符串中  
  
    String Msg=reader.readLine () ;  
  
    //以字符串的方式返回消息  
  
    return Msg ;  
}
```

2.3.2 FTP客户端

文件传输协议（File Transfer Protocol,FTP）是用于在网络上进行文件传输的一套标准协议。它属于网络传输协议的应用层。

FTP是一个8位的客户端-服务器协议，能操作任何类型的文件。FTP服务一般运行在20和21两个端口。端口20用于在客户端和服务器之间传输数据流，而端口21用于传输控制流，并且是命令通向FTP服务器的进口。当数据通过数据流传输时，控制流处于空闲状态。运行FTP服务的许多站点都开放匿名服务，在这种设置下，用户不需要账号就可以登录服务器，默认匿名用户的用户名是anonymous，这个账号不需要密码；不开放匿名服务的则要求输入用户名和密码。

在Android中可以使用第三方的库来操作FTP，这里使用Apache的包，其下载地址为http://commons.apache.org/net/download_net.cgi。其文件名为commons-net-3.2-bin.zip，解压之后得到需要使用的包commons-net-3.2.jar。

步骤1 在项目引入包commons-net-3.2.jar之后，导入需要其中的FTPClient类，代码如下：

```
import org.apache.commons.net.ftp.FTPClient ;
```

步骤2 初始化FTPClient，代码如下：

```
mFtp=new FTPClient () ;
```

步骤3 设置登录地址和端口号，代码如下：

```
mFtp.connect (FTP_URL, 21) ;
```

步骤4 设置登录用户名和密码，代码如下：

```
mFtp.login (Name,Password) ;
```

步骤5 设置文件类型和采用被动传输方式，代码如下：

```
mFtp.setFileType (FTP.BINARY_FILE_TYPE) ;
```

```
mFtp.enterLocalPassiveMode () ;
```

步骤6 传输文件，代码如下：

```
boolean aRtn=mFtp.storeFile (remoteFileName,aInputStream) ;
```

//成功返回true

```
aInputStream.close () ;
```

步骤7 关闭连接，代码如下：

```
mFtp.disconnect () ;
```

注意 FTP支持以下两种模式。

主动模式：FTP客户端向服务器的FTP控制端口（默认是21端口）发送请求，服务器接受连接，建立一条命令链路，当需要传送数据时候，客户端在命令链路上用PORT命令通知服务器，服务器从20端口向客户端的该端口发送连接请求，建立一条数据链路来传送数据。在数据链路建立的过程中因为是服务器主动请求的，所以称为主动模式。

被动模式：FTP客户端向服务器的FTP控制端口发送连接请求，服务器接收连接，建立一条命令链路，当需要传送数据时候，服务器在命令链路上用PASV命令通知客户端；客户端向服务器的该端口发送连接请求，建立一条数据链路来传送数据。在数据链路建立的过程中因为是服务器被动等待客户端请求的，所以称为被动模式。

Android FTP客户端核心代码如下：

```
//导入需要的FTPClient类
```

```
import org.apache.commons.net.ftp.FTPClient；
```

```
//初始化FTPClient对象
```

```
FTPClient ftpClient=new FTPClient()；
```

```
try{
```

```
//连接到指定的FTP服务器
```

```
ftpClient.connect (InetAddress.getByName (SERVER) ) ;  
  
//使用用户名和密码登录FTP服务器  
  
ftpClient.login (USERNAME,PASSWORD) ;  
  
//检测返回的字符串里面是否包含250  
  
//250响应代码表示“行为完成”  
  
if (ftpClient.getReplyString () .contains ("250") ) {  
  
//设置文件类型  
  
ftpClient.setFileType (  
  
//默认使用的是ASCII编码的，这里设置为二进制文件  
org.apache.commons.net.ftp.FTP.BINARY_FILE_TYPE) ;  
  
//定义一个输入缓冲区  
  
BufferedInputStream buffIn=null ;  
  
//将文件加载到缓冲区中  
  
buffIn=new BufferedInputStream (new
```

```
FileInputStream (FULL_PATH_TO_LOCAL_FILE) ) ;  
  
//设置客户端PASV模式（客户端主动连服务器端；端口用20）  
  
ftpClient.enterLocalPassiveMode () ;  
  
//存储文件，返回是否成功  
  
boolean result=ftpClient.storeFile (localAsset.  
getFileName () , progressInput) ;  
  
//关闭缓冲区  
  
buffIn.close () ;  
  
//登出  
  
ftpClient.logout () ;  
  
//断开连接  
  
ftpClient.disconnect () ;  
  
}  
  
}catch (SocketException e) {
```

```
 }catch (UnknownHostException e) {  
  
 }catch (IOException e) {  
  
 }
```

2.3.3 Telnet客户端

Telnet协议是TCP/IP协议族中的一员，是Internet远程登录服务的标准协议和主要方式。它为用户提供了在本地计算机上完成远程主机上的工作的能力。在终端使用者的计算机上使用Telnet程序，用它连接到服务器。终端使用者可以在Telnet程序中输入命令，这些命令会在服务器上运行，就像直接在服务器的控制台上输入一样。通过Telnet在本地就能控制服务器。要开始一个Telnet会话，必须输入用户名和密码来登录服务器。Telnet是常用的远程控制Web服务器的方法。

在Android中可以使用第三方的库来操作Telnet，这里使用Apache的包，其下载地址为http://commons.apache.org/net/download_net.cgi。其文件名为commons-net-3.2-bin.zip，解压之后得到需要使用的包commons-net-3.2.jar。

步骤1 在项目引入包commons-net-3.2.jar之后，导入其中的TelnetClient类，代码如下：

```
import org.apache.commons.net.telnet.TelnetClient ;
```

步骤2 初始化TelnetClient， 代码如下：

```
TelnetClient tc=new TelnetClient () ;
```

步骤3 打开连接， 代码如下：

```
tc.connect (remoteip,remoteport) ;
```

步骤4 读写数据， 代码如下：

```
tc.getInputStream () tc.getOutputStream ()
```

步骤5 断开Telnet连接， 代码如下：

```
telnet.disconnect () ;
```

Android Telnet客户端核心代码如下：

```
//定义一个TelnetClient
```

```
TelnetClient telnet ;
```

```
telnet=new TelnetClient () ;
```

```
try{
```

```
//建立连接
```

```
telnet.connect (remoteip,remoteport) ;  
}  
  
catch (IOException e) {  
    e.printStackTrace () ;  
  
    //捕获到异常，停止程序  
    System.exit (1) ;  
}  
  
IOUtil.readWrite (telnet.getInputStream () , telnet.getOutputStream  
() ,  
System.in, System.out) ;  
  
try{  
    //断开连接  
    telnet.disconnect () ;  
}  
  
catch (IOException e) {
```

```
e.printStackTrace () ;
```

```
}
```

```
}
```

其中调用的IOUtil类封装了一些读写操作，其代码如下：

```
//IOUtil类封装了一些读写操作
```

```
public final class IOUtil{
```

```
    public static final void readWrite (final InputStream remoteInput,
```

```
        final OutputStream remoteOutput,
```

```
        final InputStream localInput,
```

```
        final OutputStream localOutput) {
```

```
    //定义读写的线程
```

```
    Thread reader,writer ;
```

```
    //定义读线程的具体操作
```

```
    reader=new Thread () {
```

```
@Override

public void run () {

    int ch ;

    try{

        //判断没有被中断的时候

        while ( ! interrupted ()


            //不是结束标志

            & & (ch=localInput.read () ) !=-1) {



                //写字符到远程输入里面

                remoteOutput.write (ch) ；



                //刷新， 发送

                remoteOutput.flush () ；



            }

    }

}
```

```
        catch (IOException e) {  
            e.printStackTrace () ;  
        }  
    }  
};  
  
//定义写线程的具体操作  
  
writer=new Thread () {  
  
    @Override  
    public void run () {  
        try{  
            //把数据从输入流复制到输出流  
            Util.copyStream (remoteInput,  
                localOutput) ;  
        }  
    }  
}
```

```
        catch (IOException e) {  
            e.printStackTrace () ;  
  
            //发生异常的时候退出该操作  
  
            System.exit (1) ;  
  
        }  
  
    }  
  
    } ;  
  
    //设置writer线程  
  
    writer.setPriority (Thread.currentThread () .getPriority () +1) ;  
  
    //启动writer线程  
  
    writer.start () ;  
  
    //设置reader为后台运行  
  
    reader.setDaemon (true) ;  
  
    //启动reader线程
```

```
reader.start () ;  
  
try{  
    //使得writer线程完成run () 方法后，再执行join () 方法后的代码  
    writer.join () ;  
  
    //中断reader线程  
    reader.interrupt () ;  
  
}  
  
catch (InterruptedException e) {  
  
}  
  
}  
  
}
```

2.4 小结

本章主要讲解了Android的基本网络概念，包括IP、TCP、UDP等相关协议，特别阐述了Android Socket编程的原理和方法，并给出了相关的Android Socket实战案例。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第3章 Android基本Web技术和编程实践

本章主要介绍Android的Web编程，包括Android中的HTTP编程，以及使用Android处理JSON、SOAP、HTML等。本章中每节在详尽而重点地介绍相关概念的基础上，均给出了实战案例，以便读者更好地理解相关概念。

3.1 HTTP协议

HTTP (HyperText Transport Protocol) 协议是互联网上应用最为广泛的一种网络协议标准。所有的www文件都必须遵守这个标准。本节简要介绍HTTP协议的历史、特点，重点介绍HTTP报文结构，最后给出Android中基于HTTP协议的文件上传方法。

3.1.1 HTTP简介

HTTP是一个适用于分布式超媒体信息系统的应用层协议。它于1990年被提出，现已得到广泛使用，并得到不断完善和扩展。HTTP是万维网协会（World Wide Web Consortium）和Internet工作小组（Internet Engineering Task Force）合作的结果，他们最终发布了一系列的RFC:RFC 1945定义了HTTP/1.0版本；RFC 2616定义了今天普遍使用的一个版本——HTTP 1.1。

HTTP协议的主要特点如下：

- 支持C/S（客户端/服务器）模式。
- 简单快速。客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST。每种方法规定了客户与服务器

联系的不同类型。因为HTTP协议比较简单，HTTP服务器的程序规模较小，因而其通信速度很快。

- 灵活。HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。
- 无连接。无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。
- 无状态。HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

绝大多数的Web开发都是构建在HTTP协议之上的Web应用。要想了解Web开发，先要了解HTTP的URL，其一般形式如下：

`http://host[" : "port][abs_path]`

http表示要通过HTTP协议来定位网络资源的；host表示合法的Internet主机域名或者IP地址；port指定一个端口号，为空则使用默认端口80；abs_path指定请求资源的URI（Uniform Resource Identifier，通用资源标志符，指Web上任意的可用资源）。

HTTP报文是面向文本的，报文中的每一个字段都是一些ASCII码串，各个字段的长度是不确定的。

HTTP有两类报文：请求报文和响应报文。

1.HTTP请求报文

一个HTTP请求报文由请求行、请求报头、空行和请求数据4个部分组成，请求报文的一般格式如图3-1所示。

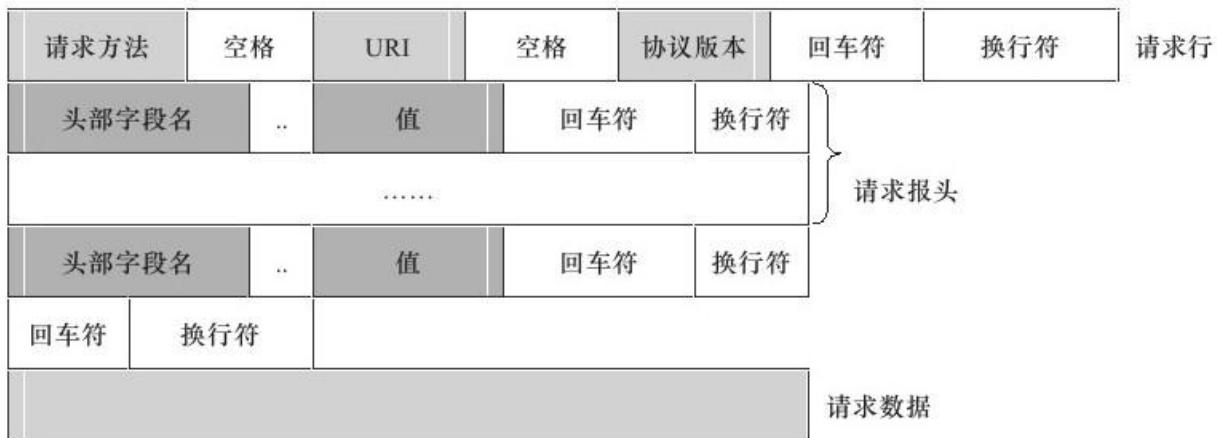


图 3-1 HTTP请求报文的一般格式

(1) 请求行

请求行由请求方法字段、URI字段和HTTP协议版本字段组成，它们之间用空格分隔。格式如下：

Method Request-URI HTTP-Version CRLF

例如：GET/form.html HTTP/1.1 (CRLF)

其中Method表示请求方法；Request-URI是一个统一资源标识符；HTTP-Version表示请求的HTTP协议版本；CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

请求方法有多种，各种方法的解释如表3-1所示。

表 3-1 HTTP 请求报文中请求方法及含义

请求方法	含 义
GET	请求获取 Request-URI 所标识的资源
POST	在 Request-URI 所标识的资源后附加新的数据
HEAD	请求获取由 Request-URI 所标识的资源的响应消息报头
PUT	请求服务器存储一个资源，并用 Request-URI 作为其标识

(续)

请求方法	含 义
DELETE	请求服务器删除 Request-URI 所标识的资源
TRACE	请求服务器回送收到的请求信息，主要用于测试或诊断
CONNECT	保留将来使用
OPTIONS	请求查询服务器的性能，或者查询与资源相关的选项和需求

(2) 请求报头

这部分内容后边会有详细介绍，这里就不赘述了。

(3) 空行

最后一个请求头之后是一个空行，发送回车符和换行符，通知服务器以下不再有请求头。

(4) 请求数据

请求数据不在GET方法中使用，而是在POST方法中使用。POST方法适用于需要客户填写表单的场合。与请求数据相关的、最常使用的请求头是Content-Type和Content-Length。

在接收和解释请求消息后，服务器会返回一个HTTP响应消息。

2. HTTP响应报文

HTTP响应报文也是由4个部分组成的，分别是：状态行、消息报头、空行、响应正文，如图3-2所示。

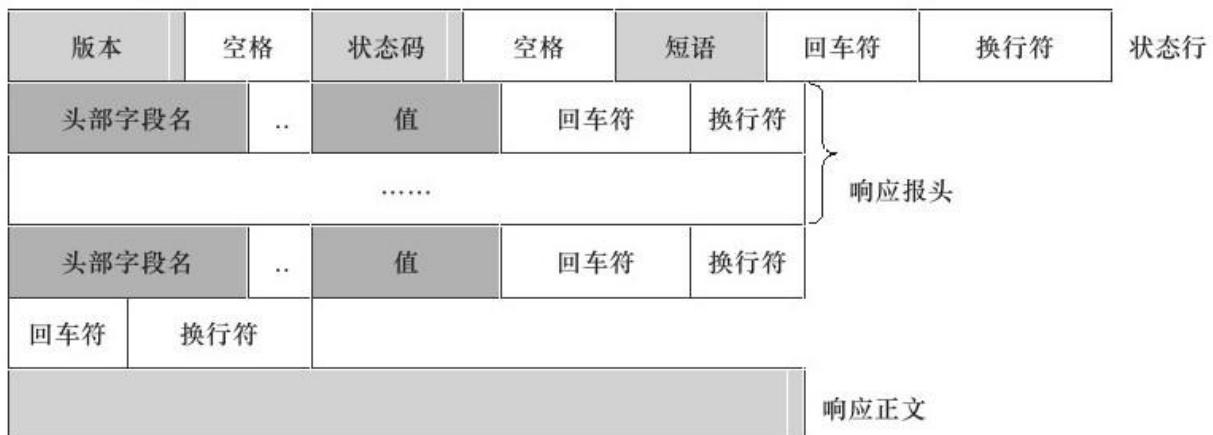


图 3-2 HTTP响应报文

(1) 状态行

状态行的格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

例如：HTTP/1.1 200 OK (CRLF)

其中，HTTP-Version表示服务器HTTP协议的版本；Status-Code表示服务器发回的响应状态代码；Reason-Phrase表示状态代码的文本描述。

状态代码由3位数字组成，第一个数字定义了响应的类别，且有5种可能取值，如表3-2所示。

表 3-2 状态代码可能取值及其含义

状态代码	含 义
1xx	指示信息，表示请求已接受，继续处理
2xx	成功，表示请求已被成功接收、理解、接受
3xx	重定向，要完成请求必须进行进一步的操作
4xx	客户端错误，请求有语法错误或请求无法实现
5xx	服务器端错误，服务器未能实现合法的请求

状态代码具体种类很多，可查阅相关文档，这里列出最常见的状态代码及状态描述，如表3-3所示。

表 3-3 常见状态代码及状态描述

状态代码	状态描述
200 OK	客户端请求成功
400 Bad Request	客户端请求有语法错误，不能被服务器所理解
401 Unauthorized	请求未经授权
403 Forbidden	服务器收到请求，但是拒绝提供服务
404 Not Found	请求资源不存在，比如输入了错误的 URL
500 Internal Server Error	服务器发生不可预期的错误
503 Server Unavailable	服务器当前不能处理客户端的请求，一段时间后可能恢复正常

(2) 响应报头

这部分内容后边会有详细介绍，这里就不赘述了。

(3) 空行

这行是空的。

(4) 响应正文

响应正文就是服务器返回的资源的内容。

3.HTTP消息报头

如前所述，HTTP消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行；对于响应消息，开始行就是状态行）、消息报头（可选）、空行（只有CRLF的行）、消息正文（可选）4部分组成。

其中消息报头包括普通报头、请求报头、响应报头、实体报头。消息报头由“头部字段名/值”对组成，每行一对，头部字段名和值用英文冒号“：“分隔，即形式如下：

头部字段名+"："+"空格+值

其中消息报头头部字段名是大小写无关的。报头描述了客户端或者服务器的属性、被传输的资源以及应该实现的连接。

4种不同类型的消息报头分述如下：

■普通报头：即可用于请求，也可用于响应，不用于被传输的实体，只用于传输消息，是作为一个整体而不是特定资源与事务相关联。比如，Date普通报头表示消息产生的日期和时间；Connection普通报头允许发送指定连接的选项，例如指定连接是连续，或者指定close选项，通知服务器，在响应完成后，关闭连接。

■请求报头：允许客户端传递关于自身的信息和希望的响应形式。请求报头通知服务器有关客户端请求的信息，典型的请求报头有如下几种。

●User-Agent：包含产生请求的操作系统、浏览器类型等信息。

●Accept：客户端可识别的内容类型列表，用于指定客户端接受哪些类型的信息。

●Host：请求的主机名，允许多个域名同处一个IP地址，即虚拟主机。

■响应报头：服务器用于传递自身信息的响应。典型的响应报头有如下两种。

- Location：用于重定向接收者到一个新的位置。Location响应报头常用于更换域名的时候。

- Server：包含了服务器用来处理请求的系统信息，与User-Agent请求报头是相对应的。

- 实体报头：定义被传送资源的信息。既可用于请求，也可用于响应。请求和响应消息都可以传送一个实体。典型的实体报头如下。

- Content-Encoding：被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码。因而要获得Content-Type报头中所引用的媒体类型，必须采用相应的解码机制。

- Content-Language：描述了资源所用的自然语言。没有设置该选项则认为实体内容将提供给所有的语言阅读。

- Content-Length：用于指明实体正文的长度，以字节方式存储的十进制数字来表示。

- Last-Modified：用于指示资源的最后修改日期和时间。

注意 这里只是列出了各种消息报头的典型形式，更多信息大家可以查阅RFC文档。官方的文档地址是<http://www.ietf.org/rfc.html>，用户可以在主页通过RFC文档号或者协议名找到自己想要的RFC文档。如可以输

入2616或者Hypertext Transfer Protocol--HTTP/1.1找到HTTP 1.1的RFC文档。

3.1.2 实战案例：基于HTTP协议的文件上传

在Android中很多时候需要上传文件，这里介绍一种利用HTTP协议的文件上传方法。HTTP协议文件上传的功能是通过RFC1867规范来描述的，Chrome、IE、Mozilla和Opera等浏览器都支持此功能。本案例从搭建服务器开始，分析了文件上传的内容及上传的过程，最后给出Android中实现文件上传的核心代码。具体步骤如下。

步骤1 搭建PHP环境，提供接收上传文件的服务器。

本案例在PHP环境中完成，为此需要搭建PHP环境。为了快速配置PHP环境，这里选用的PHP套件包为PHPnow（流行的PHP套件有多种，读者可以任意选择）。

首先从<http://www.phpnow.org/download.html>下载最新版PHPnow。当前的最新版本为1.5.6，其下载文件名为PHPnow-1.5.6.zip，该包包含的文件如图3-3所示。双击Setup.cmd安装。

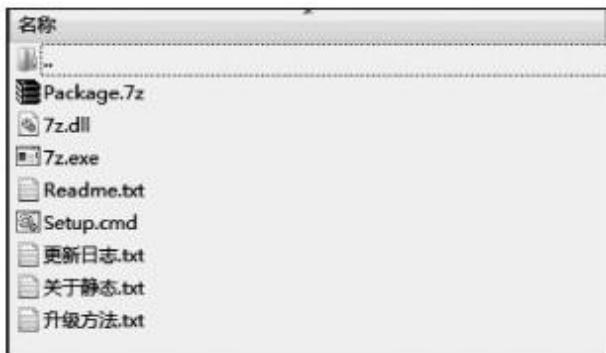


图 3-3 PHPnow安装文件

安装完成之后其目录下所含的文件如图3-4所示。后面编写的PHP代码要放到htdocs目录下面。

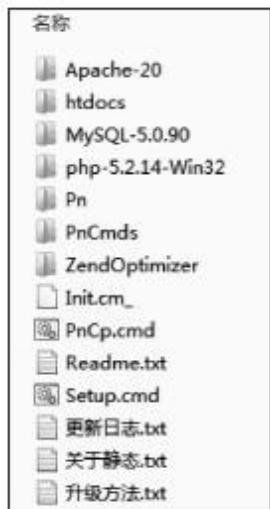


图 3-4 PHPnow安装目录下文件列表

接着编写PHP上传文件的页面。在HTML网页中，写一个如下的form，<form>标签用于为用户输入创建HTML表单。

```
<html>

<body>

<form action="upload_file.php" method="post" enctype="multipart/form-data">

<label for="file">文件名 : </label>

<input type="file" name="file" id="file"/>

<br/>

<input type="submit" name="submit" value="提交"/>

</form>

</body>

</html>
```

代码中<input type="file"/>标签在被浏览器解析后会产生一个文本框和一个“浏览”按钮。单击“浏览”按钮会出现系统的文件选择框，效果如图3-5所示。



图 3-5 标签效果

HTML中<form>标签的属性、取值及其含义如表3-4所示。

表 3-4 HTML 中 <form> 标签的属性

属性	值	描述
action	URL	规定当提交表单时，向何处发送表单数据
accept	MIME_type	规定通过文件上传来提交的文件的类型
enctype	MIME_type	规定表单数据在发送到服务器之前应该如何编码
method	get、post	规定如何发送表单数据
name	name	规定表单的名称
target	_blank、_parent、_self、_top	规定在何处打开 action URL

代码中action="upload_file.php"表示文件提交的时候会执行 upload_file.php中的代码。upload_file.php主要实现了对上传文件进行检查、复制等处理。其代码如下：

```
<?php  
  
//文件大小小于20000个字节  
  
if ($_FILES["file"]["size"]<20000) {  
  
//文件上传相关的错误代码大于0  
  
if ($_FILES["file"]["error"]>0) {  
  
//输出文件上传的错误代码信息  
  
echo"Return Code : ".$_FILES["file"]["error"]."<br/>" ;
```

```
}

else{

//输出文件名

echo"Upload : ".$_FILES["file"]["name"]."<br/>" ;

//输出文件类型

echo"Type : ".$_FILES["file"]["type"]."<br/>" ;

//输出文件大小， 单位为Kb

echo"Size : ". ($_FILES["file"]["size"]/1024) ."Kb<br/>" ;

//输出文件缓存的文件信息

echo"Temp file : ".$_FILES["file"]["tmp_name"]."<br/>" ;

//判断文件是否存在

if (file_exists ("upload/".$_FILES["file"]["name"])) {

//输出提示文件已经存在

echo$_FILES["file"]["name"]."already exists." ;
```

```
}

else{

//将上传的文件移动到新位置

move_uploaded_file ($_FILES["file"]["tmp_name"],

"upload/".$_FILES["file"]["name"]) ;


//输出提示文件存储的位置

echo"Stored in : "."upload/".$_FILES["file"]["name"] ;


}

}

}

else{

//文件大于20000个字节， 提示不合法

echo"Invalid file" ;


}
```

?>

为了测试效果，这里建立一个文本文件，文件名为test.txt，文件内容为“我是上传测试正文”。单击“浏览”按钮选中该文件，然后单击“提交”按钮。成功之后会显示如下信息：

Upload:test.txt

Type:text/plain

Size:0.0263671875 Kb

Temp file:C : \Windows\Temp\php6E50.tmp

Stored in:upload/test.txt

注意 根据RFC1867规范， enctype="multipart/form-data"， method=post,type="file"。这3个属性是必需的。 multipart/form-data用于设置上传文件的编码类型，确保匿名上传文件的正确编码。具体的解释请参阅RFC1867规范。

步骤2 分析HTTP上传过程。

这里使用HTTP协议的分析工具fiddler2来分析文件上传的过程，其下载地址为http://www.fiddler2.com/fiddler2/。安装后运行界面如图3-6所示。

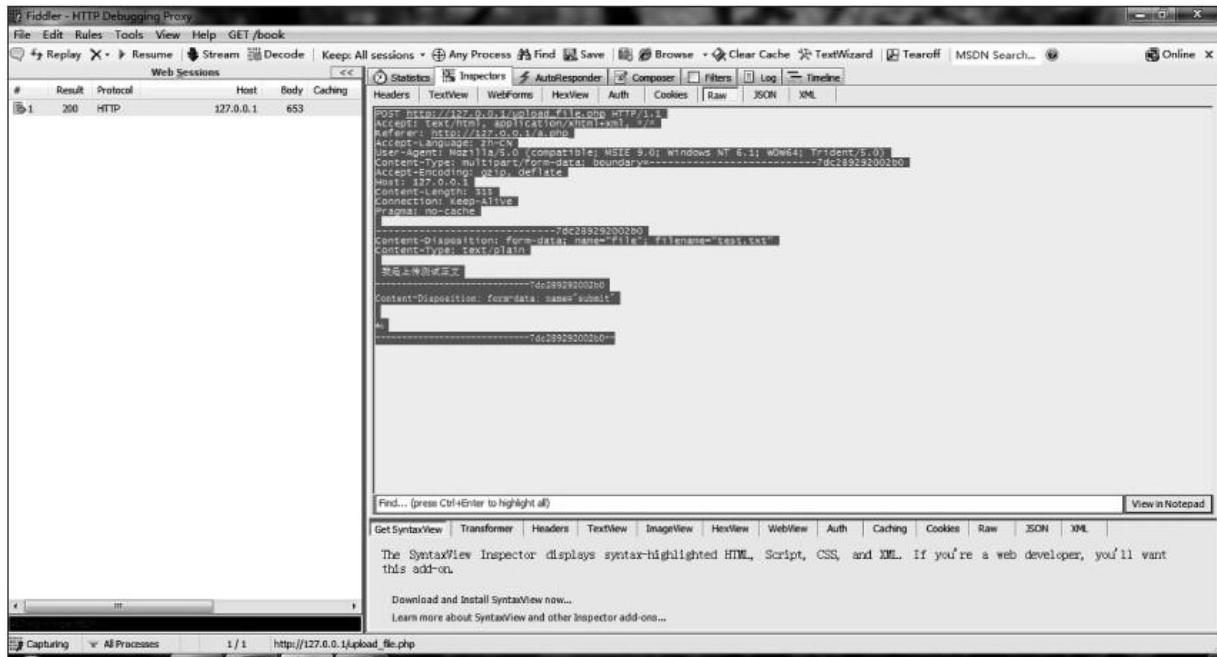


图 3-6 fiddler2运行界面

在fiddler2左侧选中上传文件的一行，右侧单击Inspectors->Raw可以看到如下内容：

POST http://127.0.0.1/upload_file.php HTTP/1.1

Accept:text/html,application/xhtml+xml, */*

Referer:http://127.0.0.1/a.php

Accept-Language:zh-CN

User-Agent:Mozilla/5.0 (compatible ; MSIE 9.0 ; Windows NT 6.1 ;
WOW64 ; Trident/5.0)

Content-Type:multipart/form-data ; boundary=-----
-7dc289292002b0

Accept-Encoding:gzip,deflate

Host:127.0.0.1

Content-Length:315

Connection:Keep-Alive

Pragma:no-cache

-----7dc289292002b0

Content-Disposition:form-data ; name="file" ; filename="test.txt"

Content-Type:text/plain

我是上传测试正文

-----7dc289292002b0

Content-Disposition:form-data ; name="submit"

-----7dc289292002b0--

对照先前对HTTP协议的介绍，分析一下上文传输的原始内容。

RFC1867对HTTP头做了适当的变更，将content-type头由以前的：

content-type:application/x-www-form-urlencoded

变为：

content-type:multipart/form-data ; +空格+boundary=-----

-7dc289292002b0

即增加了boundary，其实就是分隔线。RFC1867利用boundary分隔HTTP实体数据。boundary中数字字符区是随机生成的。因为RFC1867增加了文件上传的功能，而上传文件内容自然也会被加入到HTTP的实体中。因为既有HTTP一般的参数实体，又有上传文件的实体，所以用boundary把两种实体进行了分隔。HTTP的实体内容如下：

-----7dc289292002b0

Content-Disposition:form-data ; name="file" ; filename="test.txt"

Content-Type:text/plain

我是上传测试正文

-----7dc289292002b0

Content-Disposition:form-data ; name="submit"

-----7dc289292002b0--

根据RFC1867协议，在HTTP实体对每个上传的文件必须有说明头，如：

Content-Disposition:form-data ; name="file" ; filename="test.txt"

Content-Type:text/plain

其中Content-Disposition指明内容类型是form-data ; name="file"指明页面上<input>标签的名字是file ; filename="test.txt"指明上传文件在客户端上的路径。其后为空行，文件头说明完毕后，要加一空行，以表示后面的数据是文件的内容。最后是文件内容。

步骤3 编写Android代码实现文件上传。

下面是Android中以HTTP方式上传文件的核心代码。

```
public static String post (String actionUrl,String FileName) throws  
IOException{  
  
//产生随机分隔内容  
  
String BOUNDARY=java.util.UUID.randomUUID () .toString () ;
```

```
String PREFIX="--", LINEND="\r\n" ;  
  
String MULTIPART_FROM_DATA="multipart/form-data" ;  
  
String CHARSET="UTF-8" ;  
  
//定义URL实例  
  
URL uri=new URL (actionUrl) ;  
  
//定义HttpURLConnection实例， 打开连接  
  
HttpURLConnection conn= (HttpURLConnection) uri.openConnection()  
();  
  
//设置从主机读取数据超时（单位：毫秒）  
  
conn.setReadTimeout (5*1000) ;  
  
//设置允许输入  
  
conn.setDoInput (true) ;  
  
//设置允许输出  
  
conn.setDoOutput (true) ;  
  
//设置不允许使用缓存
```

```
conn.setUseCaches (false) ;  
  
//设置为POST发送方法  
  
conn.setRequestMethod ("POST") ;  
  
//设置维持长连接  
  
conn.setRequestProperty ("connection", "keep-alive") ;  
  
//设置文件字符集为UTF-8  
  
conn.setRequestProperty ("Charsert", "UTF-8") ;  
  
//设置文件类型  
  
conn.setRequestProperty ("Content-Type", MULTIPART_FROM_DATA  
+" ; boundary="+BOUNDARY) ;  
  
//创建一个新的数据输出流，将数据写入指定基础输出流  
  
DataOutputStream outStream=new DataOutputStream  
(conn.getOutputStream () ) ;  
  
//发送文件数据  
  
if (FileName != "") {
```

```
//定义StringBuilder对象，构建发送字符串数据

StringBuilder sb1=new StringBuilder () ;

sb1.append (PREFIX) ;

sb1.append (BOUNDARY) ;

sb1.append (LINEND) ;

sb1.append ("Content-Disposition:form-data ;

name=\"file1\" ; filename=\"\""+FileName+"\""+LINEND) ;

sb1.append ("Content-Type:application/octet-stream ; charset="+

CHARSET+LINEND) ;

sb1.append (LINEND) ;

//写入输出流中

outStream.write (sb1.toString () .getBytes () ) ;

//将文件读到输入流中

InputStream is=new FileInputStream (FileName) ;
```

```
byte[]buffer=new byte[1024] ;  
  
int len=0 ;  
  
//写入输出流中  
  
while ( (len=is.read (buffer) ) !=-1) {  
  
    outStream.write (buffer, 0, len) ;  
  
}  
  
//关闭输入流  
  
is.close () ;  
  
//添加换行标志  
  
outStream.write (LINEND.getBytes () ) ;  
  
}  
  
//请求结束标志  
  
byte[]end_data= (PREFIX+BOUNDARY+PREFIX+LINEND) .getBytes  
() ;  
  
outStream.write (end_data) ;
```

```
//刷新，发送数据  
  
outStream.flush () ;  
  
//得到响应码  
  
int res=conn.getResponseCode () ;  
  
InputStream in=null ;  
  
//上传成功则会返回响应码200  
  
if (res==200) {  
  
//读取数据  
  
in=conn.getInputStream () ;  
  
int ch ;  
  
//定义StringBuilder字符串  
  
StringBuilder sb2=new StringBuilder () ;  
  
//保存数据  
  
while ( (ch=in.read ()) !=-1) {
```

```
    sb2.append ( (char) ch) ;  
}  
  
}  
  
//如果数据不为空，则以字符串方式返回数据；否则返回null  
return in==null?null:in.toString () ;  
}
```

3.2 Android中的HTTP编程

3.2.1 HttpClient和URLConnection

HTTP协议是现在Internet上使用得最多、也是最重要的协议之一，越来越多的Android应用程序需要直接通过HTTP协议来访问网络资源。虽然在Android的java.net包中已经提供了访问HTTP协议的基本功能，但是对于大部分应用程序来说，Android原生提供的功能还不够丰富和灵活。HttpClient是Apache Jakarta Common下的子项目，用来提供高效的、最新的、功能丰富的支持HTTP协议的客户端编程工具包，并且支持HTTP协议最新的版本和建议。

一般情况下我们使用浏览器来访问一个Web服务器，用来浏览页面查看信息或者提交一些数据等。所访问的这些页面有的仅仅是一些普通的页面，有的需要用户登录后方可使用，或者需要认证以及通过加密方式传输，如HTTPS。目前我们使用的浏览器处理这些情况都不会构成问题。不过用户可能在某些时候想通过自己的程序来使用别人所提供的服务页面，比如从网页中“抓取”一些数据；利用某些站点提供的页面来完成某种功能等，例如通过已有网站提供的服务去查看某个手机号码的归属地。考虑到一些服务授权的问题，很多公司提供的页面往往并不是可以通过一个简单的URL就可以访问的，而必须经过注册然后登录后方可使用提供服务的页面，这个时候就涉及COOKIE的处理问

题。这些问题有了HttpClient就很容易解决了！HttpClient就是专门设计用来简化HTTP客户端与服务器间各种通信编程的。通过它可以让原来很复杂的事情现在轻松解决。

URL (Uniform Resource Locator) 代表统一资源定位符，Internet上的每个资源都具有一个唯一的名称标识，通常称为URL地址，这种地址可以是本地磁盘，也可以是局域网上的某一台计算机，更多的是Internet上的站点，因此URL是指向互联网“资源”的指针。URLConnection则代表了应用程序和URL之间的通信链接，通过URLConnection类的实例可以读取和写入此URL应用的资源。本节后面会给出使用URLConnection获取网络信息的实例。

3.2.2 Post和Get在HttpClient的使用

HttpClient提供的主要的功能如下：

- 实现了所有HTTP的方法（GET、POST、PUT、HEAD等）
- 支持自动转向
- 支持HTTPS协议
- 支持代理服务器

HTTP请求方法中最常用的是GET方法和POST方法。

1) GET方法

GET方法要求服务器将URL定位的资源放在响应报文的数据部分，回送给客户端。使用GET方法时，请求参数和对应的值附加在URL后面，利用一个问号（“?”）代表URL的结尾与请求参数的开始。

//通过GET方法获取页面信息

//参数为对应页面的URL

```
public static InputStream getInputStreamFromUrl (String url) {
```

//定义输出流变量

```
InputStream content=null ;
```

```
try{
```

//取得默认的HttpClient实例

```
HttpClient httpclient=new DefaultHttpClient () ;
```

//连接到服务器

```
HttpResponse response=httpclient.execute (
```

//创建HttpGet实例

```
new HttpGet (url) ) ;  
  
//获取数据内容  
  
content=response.getEntity () .getContent () ;  
  
}catch (Exception e) {  
  
}  
  
//以InputStream形式返回页面信息  
  
return content ;  
  
}
```

上面的GET方法是以InputStream的形式返回页面的信息，很多情况下需要以String-Builder、 String等字符串的格式。下面的方法把InputStream格式转为StringBuilder和String格式。

```
//将InputStream格式转化为StringBuilder格式  
  
private StringBuilder inputStreamToStringBuilder (InputStream is) {  
  
//定义空字符串  
  
String line="" ;
```

```
//定义StringBuilder的实例total  
  
StringBuilder total=new StringBuilder () ;  
  
//定义BufferedReader，载入InputStreamReader  
  
BufferedReader rd=new BufferedReader (new InputStreamReader  
(is) ) ;  
  
//readLine是一个阻塞的方法，当没有断开连接的时候就会一直等待，  
直到有数据返回  
  
//返回null表示读到数据流最末尾  
  
while ( (line=rd.readLine ()) !=null) {  
  
    total.append (line) ;  
  
}  
  
//以StringBuilder形式返回数据内容  
  
return total ;  
  
}  
  
//将InputStream格式数据流转换为String类型
```

```
private String inputStreamToString (InputStream is) {  
  
    //定义空字符串  
  
    String s="" ;  
  
    String line="" ;  
  
    //定义BufferedReader， 载入InputStreamReader  
  
    BufferedReader rd=new BufferedReader (new InputStreamReader  
        (is) ) ;  
  
    //读取到字符串中  
  
    while ( (line=rd.readLine ()) !=null) {  
  
        s+=line ;  
  
    }  
  
    //以字符串方式返回信息  
  
    return s ;  
  
}
```

2) POST方法

POST方法要求被请求服务器接收附在请求后面的数据，常用于提交表单。当客户端给服务器提供信息较多时可以使用POST方法。POST方法将请求参数封装在HTTP请求数据中，以名称值的形式出现，可以传输大量数据。

```
public void postData () {  
  
    //创建一个新的HttpClient Post头  
  
    HttpClient httpclient=new DefaultHttpClient () ;  
  
    HttpPost httppost=new HttpPost ("http://www.google.com") ;  
  
    try{  
  
        //添加数据  
  
        List<NameValuePair>nameValuePairs=new ArrayList<NameValuePair  
        > (2) ;  
  
        nameValuePairs.add (new BasicNameValuePair ("id", "12345") ) ;  
  
        nameValuePairs.add (new BasicNameValuePair  
        ("stringdata", "myString") ) ;  
  
        //使用utf-8格式对数据进行编码
```

```
httpPost.setEntity (new UrlEncodedFormEntity (nameValuePairs, "UTF-8") ) ;  
  
//执行HTTP Post请求  
  
HttpResponse response=httpclient.execute (httpPost) ;  
  
}catch (ClientProtocolException e) {  
  
}catch (IOException e) {  
  
}  
  
}  
  
}
```

使用HttpClient需要以下6个步骤：

步骤1 创建HttpClient的实例。

步骤2 创建某种连接方法的实例，对于get方法是GetMethod，而对于post方法是PostMethod。

步骤3 调用步骤1中创建好的实例的execute方法来执行步骤2中创建好的method实例。

步骤4 读response。

步骤5 释放连接。

步骤6 对得到的内容进行处理。

3.2.3 实战案例：使用HttpClient和URLConnection访问维基百科

这里介绍使用URLConnection对象和HttpClient组件访问维基百科，本案例要在维基百科里面搜索有关Android的信息。先了解一下维基百科API的构成。

维基百科是一个内容开放、多语言的百科全书协作计划。其内容除了浏览之外，还给出了其自身的API访问接口。

注意 具体的API可以参考<http://zh.wikipedia.org/w/api.php>。

案例中使用的API的格式如下：

`http://zh.wikipedia.org/w/api.php?action=opensearch & search=Android`

其中action=opensearch表示使用OpenSearch协议来访问该API。

注意 OpenSearch是一套基于XML的开放网站搜索协议。OpenSearch其实是一个简单的XML格式，用以分享搜索的结果，或定义该网站搜索

的方法，支持OpenSearch的OpenSearch search clients即可使用。目前支持的浏览器有Internet Explorer 7、Firefox 2.0、Chrome等。

search=Android表示搜索的关键词为Android；前面的&为连接符号。

注意URL中的一些字符有特殊含义，基本编码规则如下：

空格换成加号（+）；正斜杠（/）分隔目录和子目录；问号（?）分隔URL和查询；百分号（%）指定特殊字符；#号指定书签；&号分隔参数，有时也作为连接符号。

维基百科的API访问接口中常用的参数及含义如表3-5所示。

表 3-5 参数及含义

参 数	含 义
search	搜索字符串
limit	返回结果的最大值，默认是10，最大不超过100
namespace	搜索的命名空间，最大值取50，默认取0
suggest	是否打开搜索建议
format	输出格式，可选json、jsonfm、xml、xm(fm，默认为json

（1）使用URLConnection访问维基百科

在项目布局文件里面定义一个TextView用来显示访问的数据内容，代码如下：

```
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
xmlns:tools="http://schemas.android.com/tools"

    android:layout_width="match_parent"

    android:layout_height="match_parent">

<TextView

    android:id="@+id/showWiki"

    android:layout_width="fill_parent"

    android:layout_height="fill_parent"

    android:layout_alignParentLeft="true"

    android:layout_alignParentTop="true"

    android:layout_marginTop="14dp"

    android:text="TextView"/>

</RelativeLayout>
```

使用URLConnection访问维基百科的主要代码如下：

```
//给出访问的URL
```

```
String wikiSearchURL= "http://zh.wikipedia.org/w/api.php?action=opensearch & search=Android" ;  
  
try{  
  
    //初始化URL  
  
    URL url=new URL (wikiSearchURL) ;  
  
    //创建HttpURLConnection，并打开连接  
  
    HttpURLConnection httpconn= (HttpURLConnection)  
url.openConnection () ;  
  
    //判断获取的应答码是否正常  
  
    if (httpconn.getResponseCode () ==HttpURLConnection.HTTP_OK) {  
  
        //给出连接成功的提示  
  
        Toast.makeText (getApplicationContext () , "连接维基百科成功！",  
Toast.LENGTH_SHORT) .show () ;  
  
        //创建InputStreamReader  
  
        InputStreamReader isr=
```

```
//设置字符编码为utf-8

new InputStreamReader (httpconn.getInputStream () , "utf-8") ;

int i ;

String content="" ;

//读取消息到content中

while ( (i=isr.read ()) !=-1) {

content=content+ (char) i ;

}

isr.close () ;

//将获取的内容显示到界面上

showWiki.setText (content) ;

}

//断开连接

httpconn.disconnect () ;
```

```
 }catch (Exception e) {  
  
    //提示连接失败  
  
    //Toast.LENGTH_SHORT设置显示较短的时间  
  
    Toast.makeText(getApplicationContext () ,  
  
        "连接维基百科失败", Toast.LENGTH_SHORT) .show () ;  
  
    e.printStackTrace () ;  
  
}
```

(2) 使用HttpClient访问维基百科

```
//给出访问的URL  
  
String wikiSearchURL=  
  
"http://zh.wikipedia.org/w/api.php?action=opensearch & search=Android" ;  
  
//初始化DefaultHttpClient  
  
DefaultHttpClient httpclient=new DefaultHttpClient () ;  
  
//创建HttpGet
```

```
HttpGet httpget=new HttpGet (wikiSearchURL) ;  
  
//创建ResponseHandler  
  
ResponseHandler<string>responseHandler=new BasicResponseHandler  
() ;  
  
try{  
  
//获取返回的内容  
  
String content=httpclient.execute (httpget,responseHandler) ;  
  
//提示连接成功  
  
Toast.makeText (getApplicationContext () , "连接维基百科成功！",  
Toast.LENGTH_SHORT) .show () ;  
  
//显示到应用界面上  
  
showWiki.setText (content) ;  
  
}catch (Exception e) {  
  
//提示连接失败  
  
Toast.makeText (getApplicationContext () , "连接维基百科失败",
```

```
Toast.LENGTH_SHORT) .show () ;  
e.printStackTrace () ;  
}  
  
//关闭连接  
httpclient.getConnectionManager () .shutdown () ;  
}
```

运行效果如图3-7所示。

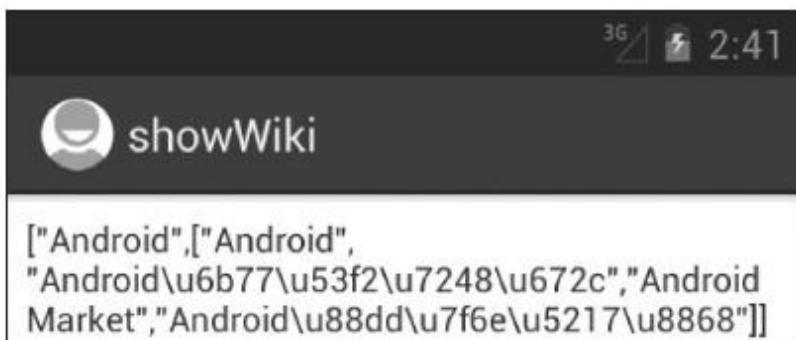


图 3-7 访问维基百科运行效果

图3-7中的\u6b77、\u53f2、\u7248、\u672c、\u88dd、\u7f6e、\u5217、\u8868为转义字符，使用log查看之后，可以得到如下的内容：

[

"Android",

[

"Android",

"Android歷史版本",

"Android Market",

"Android裝置列表"

]

]

3.3 Android处理JSON

3.3.1 JSON简介

JSON指的是JavaScript对象表示法（JavaScript Object Notation），它是一种轻量级的文本数据交换格式，类似于XML，但比XML更小、更快、更易解析。

JSON是基于JavaScript的一个子集，它使用JavaScript语法来描述数据对象，但是JSON仍然独立于语言和平台。JSON解析器和JSON库支持许多不同的编程语言。这些特性都使得JSON成为理想的数据交换语言，使其易于人们阅读和编写，同时也易于机器解析和生成。

JSON的优点如下：

- 数据格式比较简单，易于读写，格式都是压缩的，占用带宽小。
- 易于解析语言，客户端JavaScript可以简单地通过eval（）进行JSON数据的读取。
- 支持多种语言，包括ActionScript、C、C#、ColdFusion、Java、JavaScript、Perl、PHP、Python、Ruby等服务器端语言，便于服务器端的解析。

- 众多服务器端的对象、数组等能够直接生成JSON格式，便于客户端的访问提取。
- 因为JSON格式能够直接为服务器端代码使用，大大简化了服务器端和客户端的代码开发量，但是完成的任务不变，且易于维护。

JSON用于描述数据结构，有以下两种形式。

(1) “名称/值”对的集合 (A collection of name/value pairs)

“名称/值”对的集合形式又称为JSON Object，其名称和值之间使用“：“隔开，一般的形式如下：

{name:value}

例如：{"Width" : "800", "Height" : "600"}

其中名称是字符串；值可以是字符串、数值、对象、布尔值、有序列表或者null值。字符串是以""括起来的一串字符；数值是一系列0~9的数字组合，可以为负数或者小数，还可以用e或者E表示为指数形式；布尔值表示为true或者false。

上述是以“{”开始，并以“}”结束的一系列非排序的“名称/值”对（每个“名称/值”对之间使用“，”分隔）。不同的语言中，这种“名称/值”可以理解为对象（object）、记录（record）、结构（struct）、字典

(dictionary)、哈希表 (hash table)、有键列表 (keyed list) 或者关联数组 (associative array) 等。

(2) 值的有序列表 (An ordered list of values)

值的有序列表形式又称为JSON Array。在大部分语言中，值的有序列表被理解为数组 (array)，一个或者多个值用“，”分隔后，使用“[”和“]”括起来就形成了这样的列表，如下所示：

[collection, collection]

例如：

```
{  
  "employees" : [  
    {"Width" : "800", "Height" : "600"},  
    {"Width" : "700", "Height" : "800"},  
    {"Width" : "900", "Height" : "900"}  
]  
}
```

3.3.2 JSON数据解析

解析JSON数据时，首先需要明确待解析的是JSON Object还是JSON Array，然后再解析。举例如下。

(1) 解析JSON Object之一

下面是一个简单的JSON Object,name为名称，Lili是name的值，将name和Lili用“：“隔开，其文本如下。

```
{"name" : "Lili"}
```

JSONObject.getString ("String") 方法可以得到JSON对象中String名称对应的值。下面是对上面JSON对象的解析方法：

```
//新建JSONObject,jsonString字符串中为上面的JSON对象的文本
```

```
JSONObject demoJson=new JSONObject (jsonString) ;
```

```
//获取name名称对应的值
```

```
String s=demoJson.getString ("name") ;
```

(2) 解析JSON Object之二

下面是一个包含两个“名称/值”对的JSON对象，两个“名称/值”对分别是"name1"："android"和"name2"："iphone"，中间使用“，”隔开，其文

本如下：

```
{"name1": "android", "name2": "iphone"}
```

上面JSON对象的解析方法如下：

```
//新建JSONObject对象，将jsonString字符串转换为JSONObject对象
```

```
//jsonString字符串为上面的文本
```

```
JSONObject demoJson=new JSONObject (jsonString) ;
```

```
//获取名称为name1对应的值
```

```
String name1=demoJson.getString ("name1") ;
```

```
//获取名称为name2对应的值
```

```
String name2=demoJson.getString ("name2") ;
```

(3) 解析JSON Array

下面是一个简单的JSONArray,number为数组名称，[1, 2, 3]为数组的内容，其JSON文本表示如下：

```
{"number": [1, 2, 3]}
```

上面的JSON Array解析方法如下：

```
//新建JSONObject对象，将jsonString字符串转换为JSONObject对象

//jsonString字符串为上面的文本

JSONObject demoJson=new JSONObject (jsonString) ;

//获取number对应的数组

JSONArray numberList=demoJson.getJSONArray ("number") ;

//分别获取numberList中的每个值

for (int i=0 ; i<numberList.length () ; i++) {

//因为数组中的类型为int，所以为getInt，其他getString、getLong具有
类似的用法

System.out.println (numberList.getInt (i) ) ;

}
```

(4) 解析JSON Object和JSON Array混合对象

下面是一个JSON Object和JSON Array的混合文本，mobile为JSON Object名称，其对应的值为JSON Array,JSON Array中包含的对象为JSON Object，其文本表示如下：

```
{"mobile": [{"name": "android"}, {"name": "iphone"}]}
```

上面文本的解析方法如下：

```
//新建JSONObject对象，将jsonString字符串转换为JSONObject对象
```

```
//jsonString字符串为上面的文本
```

```
JSONObject demoJson=new JSONObject (jsonString) ;
```

```
//首先获取名为mobile的对象对应的值
```

```
//该值为JSONArray，这里创建一个JSONArray对象
```

```
JSONArray numberList=demoJson.getJSONArray ("mobile") ;
```

```
//依次取出JSONArray中的值
```

```
for (int i=0 ; i<numberList.length () ; i++) {
```

```
//从第i个取出JSONArray中的值为JSON Object“名称/值”对
```

```
//通过getString ("name") 获取对应的值
```

```
System.out.println (numberList.getJSONObject (i) .getString  
("name") ) ;
```

```
}
```

3.3.3 JSON打包

要想在客户端通过JSON传送对象，需要在JSON把信息全部“打包”之后将JSONObject转换为String。这样JSON就会将“打包”的信息按照特定标准的格式进行“压缩”，之后在服务端进行解析，读取通过JSON传过来的信息。

Android提供的JSON解析类都在包org.json下，主要有以下几个。

- JSONObject：可以看作是一个JSON对象，这是系统中有关JSON定义的基本单元，即前面提到的“名称/值”对。
- JSONStringer:JSON文本构建类，这个类可以帮助快速和方便地创建JSON文本。其最大的优点在于可以减少由于格式的错误导致的程序异常，引用这个类可以自动严格按照JSON语法规则创建JSON文本。每个JSONStringer实体只能对应创建一个JSON文本。
- JSONArray：它代表一组有序的数值。将其转换为String输出(toString) 所表现的形式是用方括号包裹，数值以逗号“，”分隔，即前面提到的值的有序列表。
- JSONTokener:JSON解析类。
- JSONException:JSON中涉及的异常。

下面看一个用JSONObject、JSONArray来构建JSON文本的例子，其需要构建的文本如下：

```
{
```

```
    "Strings" : {"Strings1" : "MyStrings", "Strings2" : "MyStrings"},
```

```
    "Number" : ["987654321", "123456789", "456789123"],
```

```
    "String" : "good",
```

```
    "Int" : 100,
```

```
    "Boolean" : false
```

```
}
```

上面的JSON对象中包含了5个“名称/值”对，其中名称为Strings的对应的值本身也是一个包含2个“名称/值”对的JSON对象；名称为Number的对应的值为一个JSONArray；名称为String的对应的值为一个字符串；名称为Int的对应的值为一个整型；名称为Boolean的对应的值为布尔型。

构建上述文本的主要代码如下：

```
try{
```

//创建JSONObject对象

```
JSONObject mJSONObject=new JSONObject () ;
```

//为Strings创建JSONObject对象

```
JSONObject Strings=new JSONObject () ;
```

//为Strings JSONObject对象添加第一个“名称/值”对

```
Strings.put ("Strings1", "MyStrings") ;
```

//为Strings JSONObject对象添加第二个“名称/值”对

```
Strings.put ("Strings2", "MyStrings") ; //将Strings添加到mJSONObject  
中
```

mJSONObject.put ("Strings", Strings) ; //为Number创建JSONArray对
象

```
JSONArray Number=new JSONArray () ; //将有序列表添加到Number  
中
```

```
Number.put ("987654321") .put ("123456789") .put  
("456789123") ; //将Number添加到mJSONObject中
```

```
mJSONObject.put ("Number", Number) ;
```

```
//将Int“名称/值”对添加到mJSONObject中mJSONObject.put ("Int",
100) ;

//将Boolean“名称/值”对添加到mJSONObject中mJSONObject.put
("Boolean", false) ;

}catch (JSONException ex) {

//进行异常处理

throw new RuntimeException (ex) ;

}
```

3.3.4 实战案例：JSON解析wikipedia内容

下面的实例将解析以JSON表现的wikipedia的内容。 wikipedia的API网址是<http://en.wikipedia.org/w/api.php>。 如果需要查询Android的相关内容，并用JSON格式显示出来，可以使用如下的API：

<http://en.wikipedia.org/w/api.php?action=query&prop=revisions&rvprop=content&titles=Android&format=json>

其显示的内容为：

```

{
  "query": {
    "pages": {
      "8325880": {
        "pageid": 8325880,
        "ns": 0,
        "title": "Android",
        "revisions": [
          {
            "*": "{{wiktionary|Android|android}}\\n'''Android''' commonly-
            refers to:\\n* [[Android (robot)]], designed to resemble a human\\n* [[Android
            (operating system)]], for mobile devices, produced by Google\\n'''Android''' may
            also refer to:\\n* [[Android (board game)|'Android' (board game)]], published
            by Fantasy Flight Games\\n* [[Android (drug)]], brand name for the anabolic
            steroid methyltestosterone\\n* [[Android (film)|'Android' (film)]], directed by
            Aaron Lipstadt\\n* [[Android (song)|"Android\" (song)]], by The Prodigy\\n\\n==See
            also==\\n* [[The Androids]], Australian rock band\\n* [[Droid (disambiguation)]]\\n*
            {{Lookfrom}}\\n* {{Intitle}}\\n\\n{{disambiguation}}\\n\\n[[cs:Android (rozcestník)]]\\
            n[[de:Android]]\\n[[fa: اندروید (مترادف.)]]\\n[[ko: 앤드로이드]]\\n[[id:Android]]\\
            n[[he: אנדรอยד]]\\n[[hu:Android (egyértelműsítő lap)]]\\n[[nl:Android]]\\n[[ru: Андроид
            (значения)]]\\n[[sk:Android (rozlišovacia stránka)]]\\n[[sl:Android (razločitev)]]\\
            n[[th: แอนดรอยด์ ((แก้ความก้า梧) )]]\\n[[tr:Android]]\\n[[uk: Андроїд ( значення )]]\\n[[zh-
            yue:Android (搞清楚)]]"
          }
        ]
      }
    }
  }
}

```

此处以考察Android Dome里面的simplewiktionary为例，其核心代码如下：

//从wiki服务器获取相关页面信息

```

protected static synchronized String getUrlContent (String url) throws
ApiException{

```

//检测是否设置了UserAgent属性

/*User Agent中文名为用户代理，简称UA，
UserAgent是一个特殊字符串头，使得服务器能够识别客户使用的操作
系统及版本、CPU类型、浏览器及版本、浏览器渲染引擎、浏览器语
言、浏览器插件等*/

```
if (sUserAgent==null) {  
  
    //抛出自定义异常，提示没有设置UserAgent属性  
  
    throw new ApiException ("User-Agent string must be prepared") ;  
  
}
```

//新建HttpClient对象

```
HttpClient client=new DefaultHttpClient () ;
```

//新建HttpGet对象

```
HttpGet request=new HttpGet (url) ;
```

//添加用户代理

```
request.setHeader ("User-Agent", sUserAgent) ;
```

```
try{
```

```
//打开连接，获取返回信息

HttpResponse response=client.execute (request) ；

//检测服务器是否可用

StatusLine status=response.getStatusLine () ；

//检测服务器返回的状态码

if (status.getStatusCode () !=HTTP_STATUS_OK) {

//如果服务状态码返回不正常，这里就抛出自定义异常

throw new ApiException ("Invalid response from server :" +
status.toString () ) ；

}

//取回服务器返回的内容

HttpEntity entity=response.getEntity () ；

//将返回的内容保存到InputStream对象中

InputStream inputStream=entity.getContent () ；
```

```
//创建ByteArrayOutputStream对象  
  
//ByteArrayOutputStream捕获内存缓存中的数据  
  
//然后可以将其转换成字节数组或者字符型  
  
ByteArrayOutputStream content=new ByteArrayOutputStream () ;  
  
int readBytes=0 ;  
  
//将缓存中的数据保存到ByteArrayOutputStream对象中  
  
while ( (readBytes=inputStream.read (sBuffer) ) !=-1) {  
  
content.write (sBuffer, 0, readBytes) ;  
  
}  
  
//将内容转化为字符串  
  
return new String (content.toByteArray () ) ;  
  
}catch (IOException e) {  
  
//检测到IO异常的时候，抛出自定义的异常  
  
throw new ApiException ("API返回错误", e) ;
```

```
}
```

```
}
```

//错误检测，自定义异常

```
public static class ApiException extends Exception{
```

//含有2个参数的构造函数

```
    public ApiException (String detailMessage,Throwable throwable) {
```

```
        super (detailMessage,throwable) ;
```

```
}
```

//含有1个参数的构造函数

```
    public ApiException (String detailMessage) {
```

```
        super (detailMessage) ;
```

```
}
```

```
}
```

下面的代码调用了上文定义getUrlContent方法，解析本节开头的那段JSON信息。

/*@param title Wiktionary页面返回的标题

@param expandTemplates设置模板是否可用

@返回解析之后的页面内容*/

```
public static String getPageContent (String title,boolean  
expandTemplates)
```

```
throws ApiException,ParseException{
```

//如有需要对标题和模板进行编码

```
String encodedTitle=Uri.encode (title) ;
```

```
String expandClause=expandTemplates?
```

```
WIKTIONARY_EXPAND_TEMPLATES :"" ;
```

//查询API获取内容

```
String content=getUrlContent (String.format (WIKTIONARY_PAGE,  
encodedTitle,expandClause) ) ;
```

```
try{
```

//解析返回的JSON内容

```
JSONObject response=new JSONObject (content) ;  
  
//获取query对应的JSON对象  
  
JSONObject query=response.getJSONObject ("query") ;  
  
//获取pages对应的JSON对象  
  
JSONObject pages=query.getJSONObject ("pages") ;  
  
//获取page对应的JSON对象  
  
JSONObject page=pages.getJSONObject ((String) pages.keys () .next  
() ) ;  
  
//获取revisions对应的JSON数组  
  
JSONArray revisions=page.getJSONArray ("revisions") ;  
  
//获取版本JSON对象，为revisions对应的JSON数组包含的第一个JSON  
对象  
  
JSONObject revision=revisions.getJSONObject (0) ;  
  
//获取revision JSON对象的值  
  
return revision.getString ("*") ;
```

```
 } catch (JSONException e) {  
     //错误处理  
     throw new ParseException ("API返回错误", e);  
 }  
 }
```

3.4 Android处理SOAP

3.4.1 SOAP简介

简单对象访问协议（Simple Object Access Protocol,SOAP）是一种标准化的通信规范，主要用于Web服务（Web service）。SOAP的出现可以使网页服务器（Web Server）从XML数据库中提取数据时，无需花时间去格式化页面，并能够让不同应用程序之间通过HTTP协议，以XML格式互相交换彼此的数据，使这个交换过程与编程语言、平台和硬件无关。此标准由IBM、Microsoft、UserLand和DevelopMentor在1998年共同提出，并得到IBM、Lotus（莲花）、Compaq（康柏）等公司的支持，于2000年提交给万维网联盟（World Wide Web Consortium,W3C）。目前SOAP 1.1版是业界共同的标准。

SOAP基于XML标准，用于在分布式环境中发送消息，并执行远程过程调用。使用SOAP，不用考虑任何特定的传输协议（尽管通常选用HTTP协议），就能使数据序列化。

SOAP的优点如下：

- SOAP是可扩展的。SOAP无需中断已有的应用程序，SOAP客户端、服务器和协议自身都能发展。而且SOAP能极好地支持中间介质和层次化的体系结构。

- SOAP是简单的。客户端发送一个请求，调用相应的对象，然后服务器返回结果。这些消息是XML格式的，并且封装成符合HTTP协议的消息。因此，它符合任何路由器、防火墙或代理服务器的要求。
- SOAP是完全和厂商无关的。SOAP可以相对于平台、操作系统、目标模型和编程语言独立实现。另外，传输和语言绑定以及数据编码的参数选择都是由具体的实现决定的。
- SOAP与编程语言无关。SOAP可以使用任何语言来完成，只要客户端发送正确SOAP请求（也就是说，传递一个合适的参数给一个实际的远程服务器）。SOAP没有对象模型，应用程序可以捆绑在任何对象模型中。

3.4.2 SOAP消息

1.SOA消息简介

SOAP使用Internet应用层协议作为其传输协议。SMTP以及HTTP协议都可以用来传输SOAP消息，SOAP亦可以通过HTTPS传输。一条SOAP消息就是一个普通的XML文档，包含下列元素：

- 必需的Envelope元素，可把此XML文档标识为一条SOAP消息。
- 可选的Header元素，包含头部信息。

- 必需的Body元素，包含所有的调用和响应信息。
- 可选的Fault元素，提供有关在处理此消息时发生错误的信息。

SOAP消息的重要的语法规则如下：

- SOAP消息必须使用XML来编码。
- SOAP消息必须使用SOAP Envelope命名空间。
- SOAP消息必须使用SOAP Encoding命名空间。
- SOAP消息不能包含DTD引用。
- SOAP消息不能包含XML处理指令。

SOAP消息格式如图3-8所示。

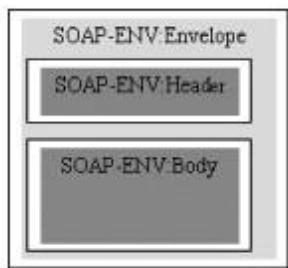


图 3-8 SOAP消息格式

2.SOA消息实例

请求时候发送的消息内容如下：

```
<soapenv:Envelope  
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"  
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  
<soapenv:Body>  
  
<req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">  
  
<req:category>classifieds</req:category>  
  
</req:echo>  
  
</soapenv:Body>  
  
</soapenv:Envelope>
```

响应时候发送的消息内容如下：

```
<soapenv:Envelope  
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing">

<soapenv:Header>

<wsa:ReplyTo><wsa:Address>http://schemas.xmlsoap.org/ws/2004/08/
addressing/role/anonymous</wsa:Address>

</wsa:ReplyTo>

<wsa:From><wsa:Address>http://localhost:8080/axis2/
services/MyService</wsa:Address>

</wsa:From>

<wsa:MessageID>ECE5B3F187F29D28BC11433905662036
</wsa:MessageID>

</soapenv:Header>

<soapenv:Body>

<req:echo xmlns:req="http://localhost:8080/axis2/services/MyService/">

<req:category>classifieds</req:category>

</req:echo>
```

```
</soapenv:Body>  
  
</soapenv:Envelope>
```

3. 调用WebService

SOAP调用WebService的具体步骤如下。

步骤1 添加ksoap2包。WebService是一种基于SOAP协议的远程调用标准，通过WebService可以将不同操作系统平台、不同语言、不同技术整合到一块。在Android SDK中并没有提供调用WebService的库，因此，需要使用第三方的SDK来调用WebService。PC版本的WebService客户端库非常丰富，例如Axis2、CXF等，但这些开发包对于Android系统来说过于庞大，也未必很容易移植到Android系统中。因此，这些开发包并不在我们考虑范围内。适合手机的WebService客户端的SDK有一些，比较常用的是Ksoap2，可以从网址<http://code.google.com/p/ksoap2-android/>下载，然后将下载的ksoap2-android-assembly-2.4-jar-with-dependencies.jar包复制到Eclipse工程的lib目录中，当然也可以放在其他的目录里。在Eclipse工程中引用这个jar包。

步骤2 指定WebService的命名空间和调用的方法名，如：

```
SoapObject request=new SoapObject (http://service, "getName") ;
```

SoapObject类的第一个参数表示WebService的命名空间，可以从WSDL文档中找到WebService的命名空间；第二个参数表示要调用的WebService方法名。

步骤3 设置调用方法的参数值，如果没有参数，可以省略。设置方法的参数值的代码如下：

```
Request.addProperty ("param1", "value") ;
```

```
Request.addProperty ("param2", "value") ;
```

要注意的是，addProperty方法的第一个参数虽然表示调用方法的参数名，但该参数值并不一定与服务端的WebService类中的方法参数名一致，只要设置参数的顺序一致即可。

步骤4 生成调用WebService方法的SOAP请求信息。该信息由SoapSerializationEnvelope对象描述，代码如下：

```
SoapSerializationEnvelope envelope=
```

```
new SoapSerializationEnvelope (SoapEnvelope.VER11) ;
```

```
Envelope.bodyOut=request ;
```

创建SoapSerializationEnvelope对象时需要通过SoapSerializationEnvelope类的构造方法设置SOAP协议的版本号。该版本号需要根据服务端

WebService的版本号设置。在创建SoapSerializationEnvelope对象后，不要忘了设置SOAPSoapSerializationEnvelope类的bodyOut属性，该属性的值就是在步骤2创建的SoapObject对象。

步骤5 创建HttpTransportsSE对象。通过HttpTransportsSE类的构造方法可以指定WebService的WSDL文档的URL。

```
HttpTransportSE ht=new HttpTransportSE
```

```
("http://fy.webxml.com.cn/webservices/EnglishChinese.asmx?wsdl") ;
```

步骤6 使用call方法调用WebService方法，代码如下：

```
ht.call (null,envelope) ;
```

call方法的第一个参数一般为null，第2个参数就是在步骤4创建的SoapSerialization-Envelope对象。

步骤7 使用getResponse方法获得WebService方法的返回结果，代码如下：

```
SoapObject soapObject= (SoapObject) envelope.getResponse () ;
```

步骤8 解析返回的内容。

3.4.3 实战案例：SOAP解析天气服务

以下为一个简单的实现天气查看功能的例子。在这个例子中，用户在文本框中输入城市名之后单击“查询”按钮，查询成功后，会在应用界面上显示所查询城市的天气信息。

其实现的具体过程为：从客户端获取用户输入的城市名称，将城市名称打包成符合SOAP协议的查询消息，把查询信息发送给提供SOAP天气服务的服务器；服务器内部进行操作之后，返回给客户端查询城市的天气信息，该信息以SOAP格式返回，客户端对其进行解析之后显示给用户。

下面是案例的布局文件，给出了使用的控件：

```
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
  
    xmlns:tools="http://schemas.android.com/tools"  
  
    android:layout_width="match_parent"  
  
    android:layout_height="match_parent">  
  
<!--显示控件，用于显示天气情况-->  
  
<TextView  
    android:id="@+id/textView1"
```

```
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:layout_centerHorizontal="true"  
  
    android:layout_centerVertical="true"  
  
    android:padding="@dimen/padding_medium"  
  
    tools:context=".AndroidSoapActivity"/>
```

<!--按钮， 用户提交城市名称时候单击该按钮-->

```
<Button  
  
    android:id="@+id/ok"  
  
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:layout_alignParentTop="true"  
  
    android:layout_toRightOf="@+id/textView1"  
  
    android:text="@string/search"/>
```

<!--输入控件，用户输入城市名称-->

<EditText

 android:id="@+id/cityName"

 android:layout_width="wrap_content"

 android:layout_height="wrap_content"

 android:layout_alignParentLeft="true"

 android:layout_alignParentTop="true"

 android:text="@string/cityName"/>

</RelativeLayout>

应用内部对查询处理的主要代码如下：

```
import java.io.UnsupportedEncodingException ;
```

```
//加入需要使用的ksoap2包中的类
```

```
import org.ksoap2.SoapEnvelope ;
```

```
import org.ksoap2.serialization.SoapObject ;
```

```
import org.ksoap2.serialization.SoapSerializationEnvelope ;  
  
import org.ksoap2.transport.HttpTransportSE ;  
  
import android.os.AsyncTask ;  
  
import android.os.Bundle ;  
  
import android.app.Activity ;  
  
import android.view.Menu ;  
  
import android.view.View ;  
  
import android.widget.Button ;  
  
import android.widget.EditText ;  
  
import android.widget.TextView ;  
  
//SOAP方式查询天气情况  
  
public class AndroidSoapActivity extends Activity{  
  
    //指定命名空间  
  
    private static final String NAMESPACE="http://WebXml.com.cn/" ;
```

```
//给出接口地址  
  
private static String URL=  
"http://www.webxml.com.cn/webservices/weatherwebservice.asmx" ;  
  
//设置方法名  
  
private static final String METHOD_NAME="getWeatherbyCityName" ;  
  
//设置查询接口参数  
  
private static String SOAP_ACTION=  
"http://WebXml.com.cn/getWeatherbyC-ityName" ;  
  
//定义字符串，保存天气信息  
  
private String weatherToday ;  
  
//定义按钮  
  
private Button okButton ;  
  
//定义SoapObject对象  
  
private SoapObject detail ;
```

//定义输入控件

```
private EditText cityNameText ;
```

//定义显示控件，显示天气信息

```
private TextView cityMsgView ;
```

@Override

```
public void onCreate (Bundle savedInstanceState) {
```

```
super.onCreate (savedInstanceState) ;
```

//加载布局文件

```
setContentView (R.layout.activity_android_soap) ;
```

//获取控件

```
cityNameText= (EditText) findViewById (R.id.cityName) ;
```

```
cityMsgView= (TextView) findViewById (R.id.textView1) ;
```

```
okButton= (Button) findViewById (R.id.ok) ;
```

//为按钮添加事件监听

```
okButton.setOnClickListener (new Button.OnClickListener () {  
  
    public void onClick (View v) {  
  
        //执行获取天气信息的操作  
  
        new showWeatherAsyncTask () .execute () ;  
  
    }  
  
}) ;  
  
}  
  
//使用AsyncTask异步方式获取并显示天气信息  
  
private class showWeatherAsyncTask extends AsyncTask<  
String,Integer,String> {  
  
    @Override  
  
    protected String doInBackground (String.....Urls) {  
  
        //获取并显示天气信息  
  
        showWeather () ;  
  
        return null ;
```

```
}
```

```
protected void onPostExecute (String result) {
```

```
}
```

```
} ;
```

```
//获取并显示天气信息
```

```
private void showWeather () {
```

```
//获取需要查询的城市名称
```

```
String city=cityNameText.getText () .toString () .trim () ;
```

```
//检测城市名称是否为空
```

```
if ( ! city.isEmpty () ) {
```

```
//获取指定城市的天气信息
```

```
getWeather (city) ;
```

```
}
```

```
}
```

```
//获取指定城市的天气信息，参数cityName为指定的城市名称

public void getWeather (String cityName) {

try{

//新建SoapObject对象

SoapObject rpc=new SoapObject (NAMESPACE,METHOD_NAME) ;

//给SoapObject对象添加属性

rpc.addProperty ("theCityName", cityName) ;

//创建HttpTransportSE对象，并指定WebService的WSDL文档的URL

HttpTransportSE ht=new HttpTransportSE (URL) ;

//设置debug模式

ht.debug=true ;

//获得序列化的envelope

SoapSerializationEnvelope envelope=

new SoapSerializationEnvelope (SoapEnvelope.VER11) ;
```

```
//设置bodyOut属性的值为SoapObject对象rpc  
  
envelope.bodyOut=rpc ;  
  
//指定webservice的类型为dotNet  
  
envelope.dotNet=true ;  
  
envelope.setOutputSoapObject (rpc) ;  
  
//使用call方法调用WebService方法  
  
ht.call (SOAP_ACTION,envelope) ;  
  
//获取返回结果  
  
SoapObject result= (SoapObject) envelope.bodyIn ;  
  
//使用getResponse方法获得WebService方法的返回结果  
  
detail= (SoapObject) result.getProperty  
("getWeatherbyCityNameResult") ;  
  
System.out.println ("detail"+detail) ;  
  
//解析返回的数据信息为SoapObject对象，对其进行解析  
  
parseWeather (detail) ;
```

return ;

```
}catch (Exception e) {
```

```
e.printStackTrace () ;
```

}

}

//解析SoapObject对象

```
private void parseWeather (SoapObject detail) throws
```

UnsupportedEncodingException{

//获取日期

```
String date=detail.getProperty(6).toString();
```

//获取天气信息

```
weatherToday="今天：" + date.split ("") [0] ;
```

```
weatherToday=weatherToday+"天气："+date.split('') [1];
```

```
weatherToday=weatherToday+"气温：" +detail.getProperty(5).toString()
() ;
```

```
weatherToday=weatherToday+"风力：" +detail.getProperty(7).toString()
() +"" ;
System.out.println ("weatherToday is"+weatherToday) ;
//显示到cityMsgView控件上
cityMsgView.setText (weatherToday) ;
}

//创建Menu菜单
@Override
public boolean onCreateOptionsMenu (Menu menu) {
getMenuInflater ().inflate (R.menu.activity_android_soap,menu) ;
return true ;
}
}
```

运行效果如图3-9所示。



图 3-9 SOAP解析天气服务

3.5 Android对HTML的处理

3.5.1 解析HTML

在Android应用程序开发过程中，经常需要解析HTML文档，特别是那类通过“爬网站”抓取数据的应用，比如天气预报。Java常用的解析HTML文档的方法有以下几种：

- 使用正则表达式来抽取数据。
- 以纯字符串查找定位来实现。
- 使用HTML Parser解析器。
- 使用Jsoup解析器。

在Android平台上推荐使用Jsoup解析器来解析HTML文档。Jsoup既可以
通过一个URL网址，也可以通过存储HTML脚本的文件或者存储HTML
脚本的字符串作为数据源，然后通过DOM、CSS选择器来查找、抽取
数据。

使用Jsoup解析字符串形式的HTML文件的方法如下：

```
//定义需要解析的HTML字符串
```

```
String html="<html><head><title>First parse</title></head>"  
+"<body><p>Parsed HTML into a doc.</p></body></html>" ;  
  
//将字符串解析之后放到Document对象中  
  
Document doc=Jsoup.parse (html) ;  
  
}
```

下面是一个具体的解析例子，使用Jsoup从HTML文件中提取出超链接、超链接文本、页面描述等内容。

```
//需要解析的HTML字符串  
  
String html("<p>An<a href='http://example.com/'><b>example</b></a>link.</p>" ;  
  
//保存到Document对象中  
  
Document doc=Jsoup.parse (html) ;  
  
//得到第一个a标签的超链接  
  
Element link=doc.select ("a") .first () ;  
  
//取出HTML字符串中的文本内容
```

//这里test的值为An example link

```
String text=doc.body().text();
```

//获取属性为href的字符串

//这里linkHref的值为"http://example.com/"

```
String linkHref=link.attr("href");
```

//获取a标签内部的纯文本

//linkText为"example"

```
String linkText=link.text();
```

//获取整个a标签里面的字符串

//这里linkOuterH的值为example

```
String linkOuterH=link.outerHtml();
```

//获取a标签内部（不包含a标签）的全部字符串

//这里linkInnerH的值为example

```
String linkInnerH=link.html();
```

Jsoup还可以使用Whitelist（）方法把不规范的HTML格式整理为规范格式，Whitelist方法定义了哪些HTML的元素和属性可以保留，其他的全部会被删除掉。Whitelist.basic（）方法允许通过的文本节点为：a、b、blockquote、br、cite、code、dd、dl、dt、em、i、li、ol、p、pre、q、small、strike、strong、sub、sup、u、ul，以及相应的属性，不允许图片通过。

具体的使用方法如下：

```
String unsafe=
```

```
"<p><a href='http://example.com/' onclick='stealCookies () '>Link</a></p>" ;
```

//调用clean方法整理不标准的代码

```
String safe=Jsoup.clean (unsafe,Whitelist.basic () ) ;
```

```
//safe为<p><a href="http://example.com/" rel="nofollow">Link</a></p>
```

3.5.2 HTML适配屏幕

Android终端的物理尺寸、分辨率的类别众多，可能你为一种终端设计了一套UI符合要求，但是在另一类大小的物理终端上显示的就完全不

是你想要的。如何将一个应用程序适配在不同的手机上，这就是本节要讲的Android适配屏幕问题。

这里面涉及几个概念，关于像素（pixel）、分辨率（Resolution, width×height）大家已经很清楚了，这里不再强调。要强调的是如下几个：

- 屏幕的尺寸（Screen size），指屏幕对角线长度（单位inch，即英寸）。
- 屏幕密度（Screen density），指单位长度上的像素点数（dots per inches,dpi）。
- 独立像素密度或密度无关像素（Density-independent pixel,dp），标准是160dpi，此时1dp对应1个pixel。dp转换为屏幕像素的计算公式是： $px = dp \times (dpi / 160)$ 。例如根据公式，可知240dpi的屏幕，1dp对应1.5个pixel。屏幕密度越大，1dp对应的像素点数越多。

Android屏幕有两种分类方式，具体如下。

(1) 以总像素数分

每种屏幕都有其最小分辨率：xlarge屏幕最小分辨率为 960×720 dp；large屏幕最小分辨率为 640×480 dp；normal屏幕最小分辨率为

470×320dp；small屏幕最小分辨率为426×320dp。这样通过计算就可以根据分辨率划分种类，如图3-10所示。

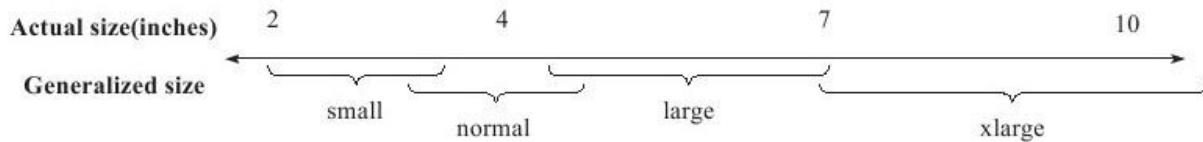


图 3-10 Android 屏幕支持的范围 (以总像素数分)

(2) 以屏幕密度分

Android 屏幕按密度分类，具体如图3-11所示。

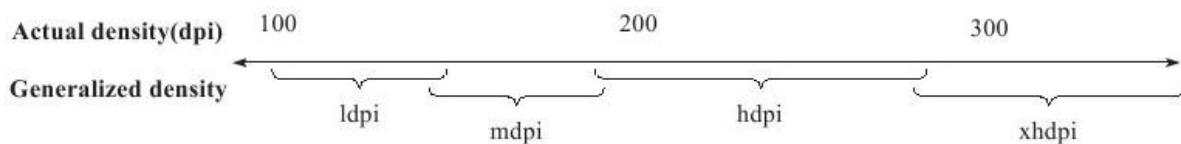


图 3-11 Android 屏幕支持的范围 (以屏幕密度分)

屏幕的匹配要以上面两种方式为参考，开发过程中需要比较两图的对应关系。此外，有时还要考虑屏幕是水平 (landscape) 的还是竖直 (portrait) 的。

当新建一个Android工程后，系统会自动创建3个存放不同分辨率、不同密度UI的文件夹，如图3-12所示。

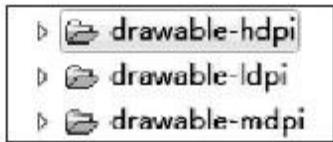


图 3-12 Android工程自动创建的文件夹

3个文件夹下的UI对应不同分辨率及密度的屏幕，对应关系如表3-6所示。

表 3-6 UI 与分辨率等的对应关系

文 件 夹	屏 幕 类 型	分 辨 率	密 度	尺 寸
drawable-hdpi	WVGA	480×800	240	大
drawable-ldpi	QVGA	240×320	120	小
drawable-mdpi	HVGA	320×480	160	中

这样，3个文件夹中的UI就是针对特征屏幕分辨率及密度进行设计的，Android终端会在打开应用的时候自动根据终端类型匹配与文件夹里提供的分辨率及密度相近的UI。

3.5.3 JavaScript混合编程

本节主要介绍在Android中显示HTML代码、添加JavaScript支持，以及通过JavaScript调用Activity。

1. 在Android中显示HTML代码

首先定义布局文件，添加WebView的组件。下面这段代码，展示了如何在布局文件中添加WebView。

```
<?xml version="1.0" encoding="utf-8"?>

<ScrollView xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <!--定义线性布局-->

    <LinearLayout
        android:orientation="vertical"
        android:layout_width="match_parent"
        android:layout_height="wrap_content">

        <!--添加第一个WebView控件-->

        <WebView android:id="@+id/wv1" />
    
```

```
    android:layout_height="wrap_content"

    android:layout_width="match_parent"/>

<!--添加第二个WebView控件-->

<WebView android:id="@+id/wv2"

    android:layout_height="wrap_content"

    android:layout_width="match_parent"/>

<!--添加第三个WebView控件-->

<WebView android:id="@+id/wv3"

    android:layout_height="wrap_content"

    android:layout_width="match_parent"/>

</LinearLayout>

</ScrollView>
```

下面介绍如何在代码中为WebView加载数据：

```
public class WebView1 extends Activity{
```

```
@Override  
  
public void onCreate (Bundle icicle) {  
  
super.onCreate (icicle) ;  
  
setContentView (R.layout.webview_1) ;  
  
//定义数据类型  
  
final String mimeType="text/html" ;  
  
//定义编码类型  
  
final String encoding="utf-8" ;  
  
WebView wv ;  
  
wv= (WebView) findViewById (R.id.wv1) ;  
  
//使用 loadData 方法来加载 HTML 数据  
  
//其第一个参数表示需要加载的数据  
  
//mimeType 表示数据类型  
  
//encoding 表示编码类型
```

```
wv.loadData ("<a href='x'>Hello World ! -1</a>",  
mimeType,encoding) ;
```

```
wv= (WebView) findViewById (R.id.wv2) ;
```

//加载到第二个WebView控件

```
wv.loadData ("<a href='x'>Hello World ! -2</a>",  
mimeType,encoding) ;
```

```
wv= (WebView) findViewById (R.id.wv3) ;
```

//加载到第三个WebView控件

```
wv.loadData ("<a href='x'>Hello World ! -3</a>",  
mimeType,encoding) ;
```

```
}
```

```
}
```

需要添加允许访问网络的权限，如下所示：

```
<uses-permission android:name="android.permission.INTERNET"/>
```

在Android的安全模型中，每一个应用都有自己的Linux用户和群组，在单独的进程和VM上运行，不能影响到其他应用。Android同时也限定

了系统资源的使用，像网络设备、SD卡、录音设备等。如果应用希望去使用指定的系统资源，就必须去申请Android的权限，这就是<uses-permission>元素的作用。

一个权限通常有以下格式，用一个名字为name的字符串去表示希望使用的权限。

```
<uses-permission android:name="string"/>
```

这里使用android.permission.INTERNET表示访问网络的权限。全部的权限列表可以参考网页

<http://developer.android.com/reference/android/Manifest.permission.html>的内容。根据Android版本的不同，其列表各有不同。

2. WebView中添加JavaScript支持

WebView是Android中View的扩展，可以使用WebView作为客户端的一部分，能将Web页面作为显示界面的一部分。WebView组件不能实现一个浏览器的完整功能，比如不能实现导航控制或者地址栏，其默认仅是展现一个Web页面。

在展示终端用户协议或者用户指南等内容的时候，使用WebView可以方便地在应用中提供一些需要及时更新的信息。此时在Android应用

中，需要创建包含WebView的Activity，然后利用它来展现网上的文档。

当展示的数据需要连接网络来获取的时候，也可以在Android应用中构建一个WebView实现。应用这种方法可以更方便地提供相关数据的Web页面，WebView通过解析页面数据并将这些数据重新处理之后，再以更加适合用户使用的方式放到Android应用中。此外，也可以设计一个专供Android设备使用的Web页面，并在Android中通过一个WebView来加载这个页面。

要在应用中加入WebView，只需要在布局文件中加入WebView控件即可。例如，下面是一个布局文件，在这个文件中，通过设置属性使得WebView控件充满屏幕。

```
<?xml version="1.0" encoding="utf-8"?>  
  
<WebView xmlns:android="http://schemas.android.com/apk/res/android"  
        android:id="@+id/webview"  
        //定义WebView在屏幕上可以使用的宽度，fill_parent即填充整个屏幕  
        android:layout_width="fill_parent"  
        //定义WebView在屏幕上可以使用的高度，fill_parent即填充整个屏幕  
        android:layout_height="fill_parent" />
```

```
        android:layout_height="fill_parent"
```

 />

通过使用loadUrl () ， 在WebView中加载页面。

```
WebView myWebView= (WebView) findViewById (R.id.webview) ;
```

```
myWebView.loadUrl ("http://www.example.com") ;
```

3.在WebView中使用JavaScript

如果需要在加载WebView中的Web页面中使用JavaScript，就要在WebView中启用JavaScript。可以通过WebView的WebSettings属性来启用它。先通过getSettings () 来获取WebSettings的值，然后通过setJavaScriptEnabled () 来启用JavaScript。例如：

```
WebView myWebView= (WebView) findViewById (R.id.webview) ;
```

```
WebSettings webSettings=myWebView.getSettings () ;
```

```
webSettings.setJavaScriptEnabled (true) ;
```

4.JavaScript和Android代码相互调用

通过WebView，可以在JavaScript代码和客户端的Android代码间创建接口。例如， JavaScript代码可以调用Android代码中的方法来展示一个

Dialog，而不需要使用JavaScript中的alert（）函数。为了在你的JavaScript和Android代码间绑定一个新的接口，需要调用addJavascriptInterface（），传给它一个类实例来绑定到JavaScript，还需要一个接口名让JavaScript可以调用，以便来访问类。

例如，在Android应用中包括如下类：

```
public class JavaScriptInterface{  
    Context mContext ;  
  
    /*实例化接口*/  
  
    JavaScriptInterface (Context c) {  
  
        mContext=c ;  
  
    }  
  
    public void showToast (String toast) {  
  
        Toast.makeText (mContext,toast,Toast.LENGTH_SHORT) .show () ;  
  
    }  
}
```

在这个例子中， JavaScriptInterface类让Web页面可以使用showToast () 方法来创建一个Toast消息。

可以通过addJavascriptInterface () 绑定JavaScriptInterface类到正在 WebView运行的JavaScript，并将接口命名为Android。例如：

```
WebView webView= (WebView) findViewById (R.id.webview) ;
```

```
webView.addJavascriptInterface (new JavaScriptInterface  
 (this) , "Android") ;
```

这段代码为在WebView上运行的JavaScript创建了一个名为Android的接口。这时候，你的Web app就能访问JavaScriptInterface类了。例如，下面是一些HTML以及JavaScript，在用户单击按钮的时候，它们使用这个新接口创建一个Toast消息。

```
<input type="button" value="Say hello" onClick="showAndroidToast  
('Hello Android ! ') "/>
```

```
<script type="text/javascript">
```

```
function showAndroidToast (toast) {
```

```
    Android.showToast (toast) ;
```

```
}
```

```
</script>
```

不需要从JavaScript初始化Android接口，WebView会自动让它为你的Web页面所用。所以，在单击按钮的时候，`showAndroidToast ()` 函数会用这个Android接口来调用`JavaScriptInterface.showToast ()` 方法。

注意 绑定到你的JavaScript的对象在另一个线程中运行，而不是在创建它的线程中运行。使用`addJavascriptInterface ()` 可以让JavaScript控制你的Android应用。这是一把双刃剑，既有用同时也可能带来安全威胁。当WebView中的HTML不可信时（例如，HTML的部分或者全部都是由一个未知的人或者进程提供的），那么一个攻击者就可能使用HTML来执行客户端的任何他想要的代码。因此，不应该使用`addJavascriptInterface ()`，除非WebView中的所有HTML以及JavaScript都是你自己写的。同样不应该让用户将你的WebView定向到另外一个不是你自己的Web页面上去（相反，让用户的默认浏览器应用打开外部链接——用户浏览器默认打开所有URL链接，因此一定要小心处理页面导航，像下面所描述的那样）。

5. 处理页面导航

当用户单击一个WebView中的页面链接时，默认是让Android启动一个可以处理URL的应用。通常，是由默认的浏览器打开并加载目标URL的。然而，你可以在WebView中覆盖这一行为，那么链接就会在

WebView中打开。这样，你就可以让用户通过保存在WebView中的浏览记录执行前进或者后退操作了。

要想让用户可以通过单击打开链接，只需要使用setWebViewClient () 为WebView提供一个WebViewClient即可。例如：

```
WebView myWebView= (WebView) findViewById (R.id.webview) ;  
myWebView.setWebViewClient (new WebViewClient () ) ;
```

这样，现在所有用户单击的链接都会直接在WebView中加载了。

如果想要对于加载的链接的位置有更多控制，你可以创建自己的WebViewClient，并覆盖shouldOverrideUrlLoading () 方法。例如：

```
private class MyWebViewClient extends WebViewClient{  
  
    @Override  
  
    public boolean shouldOverrideUrlLoading (WebView view,String url) {  
  
        if (Uri.parse (url) .getHost () .equals ("www.example.com") ) {  
  
            //让WebView加载该页面  
  
            return false ;
```

```
}

//设置为在新的页面中打开

Intent intent=new Intent (Intent.ACTION_VIEW,Uri.parse (url) ) ;

startActivity (intent) ;

return true ;

}

}
```

然后为WebView创建一个新的WebViewClient的实例。

```
WebView myWebView= (WebView) findViewById (R.id.webview) ;

myWebView.setWebViewClient (new MyWebViewClient () ) ;
```

现在当用户单击链接的时候，系统会调用shouldOverrideUrlLoading()，通过if (Uri.parse (url) .getHost () .equals ("www.example.com")) 来检查URL host是否和某个特定的域匹配（如上面定义的"www.example.com"）。如果匹配，那么该方法就返回false，不去覆盖URL加载（它仍然让WebView像往常一样加载URL）。

如果不匹配，那么就会创建一个Intent来加载默认活动（default Activity）来处理URL（通过用户默认的Web浏览器解析）。

6.历史记录导航

当WebView覆盖了URL加载时，它会自动生成历史访问记录。可以通过goBack () 或goForward () 向前或向后访问已访问过的站点。

例如，下面的代码实现了通过Activity来利用设备的后退按钮来向后导航。

```
@Override
```

```
public boolean onKeyDown (int keyCode,KeyEvent event) {
```

```
//检测是否是返回键，并且有可以返回的历史记录
```

```
if ((keyCode==KeyEvent.KEYCODE_BACK) & &
```

```
myWebView.canGoBack ()) {
```

```
myWebView.goBack () ;
```

```
return true ;
```

```
}
```

//如果没有可以返回的历史记录则返回给系统处理，此时有可能是退出该界面

```
return super.onKeyDown (keyCode,event) ;  
}
```

如果有历史访问记录可供访问， canGoBack () 方法会返回true。类似地，可以使用canGoForward () 来检查是否有向前访问记录。如果不做这个检查，那么一旦用户访问到历史记录最后一项， goBack () 或 goForward () 就什么都不做。

3.5.4 实战案例：Android自定义打开HTML页面

案例解析http://translate.google.cn/m里面的语言列表页面，取出其文字，同时过滤不需要的部分，即去除“反馈意见”等相关说明，其标识为如下的加粗字体。

```
<html>  
<body dir="ltr">  
<form class="" action="/m?hl=zh-CN">  
<div class="small center">
```


Google首页

向我们发送反馈意见

查看Google：

移动版

|标准版

©2012 Google-

隐私权政策

</div>

</body>

</html>

解析该翻译页面的主要代码如下：

```
import java.io.BufferedReader ;  
  
import java.io.DataInputStream ;  
  
import java.io.IOException ;  
  
import java.io.InputStreamReader ;  
  
import java.net.MalformedURLException ;  
  
import java.net.URL ;  
  
import java.netURLConnection ;  
  
//引入Jsoup解析包  
import org.jsoup.Jsoup ;  
  
import org.jsoup.nodes.Document ;  
  
import org.jsoup.nodes.Element ;
```

```
import org.jsoup.select.Elements ;  
  
import android.os.AsyncTask ;  
  
import android.os.Bundle ;  
  
import android.app.Activity ;  
  
import android.util.Log ;  
  
import android.view.Menu ;  
  
import android.webkit.WebSettings ;  
  
import android.webkit.WebView ;  
  
import android.webkit.WebViewClient ;  
  
public class MainActivity extends Activity{  
  
    //定义调试的标签  
  
    private static final String TAG="ParseHtml" ;  
  
    WebView wv ;  
  
    //设置获取页面的地址，使用移动版的地址可以减少流量，加快速度
```

```
String url="http://translate.google.cn/m" ;  
  
//设置语言选择列表页面  
  
String langListUrl  
="http://translate.google.cn/m?sl=auto & tl=zh-CN & hl=zh-CN & mui=sl" ;  
  
@Override  
  
public void onCreate (Bundle savedInstanceState) {  
  
super.onCreate (savedInstanceState) ;  
  
//加载布局文件  
  
setContentView (R.layout.activity_main) ;  
  
//获取WebView控件  
  
wv= (WebView) findViewById (R.id.webView1) ;  
  
//获取WebView属性  
  
WebSettings webSettings=wv.getSettings () ;  
  
//允许在WebView里面运行JavaScript
```

```
webSettings.setJavaScriptEnabled (true) ;  
  
wv.setWebViewClient (new WebViewClient () {  
  
//在WebView加载的时候运行  
  
public boolean shouldOverrideUrlLoading (WebView view,String url) {  
  
//获取翻译页面  
  
new HTMLAsyncTask ().execute (url) ;  
  
return true ;  
  
}  
  
}) ;  
  
//获取语言列表页面  
  
new HTMLAsyncTask ().execute (langListUrl) ;  
  
}  
  
//创建Menu菜单  
  
@Override
```

```
public boolean onCreateOptionsMenu (Menu menu) {  
    getMenuInflater () .inflate (R.menu.activity_main,menu) ;  
  
    return true ;  
}  
  
//使用异步方法  
  
private class HTMLAsyncTask extends AsyncTask<String,Integer,String>  
{  
  
    @Override  
  
    protected String doInBackground (String.....Urls) {  
  
        String html="" ;  
  
        try{  
  
            //使用参数Url， 新建URL对象  
  
            URL newUrl=new URL (Urls[0]) ;  
  
            //新建URLConnection对象， 打开连接  
  
            URLConnection connect=newUrl.openConnection () ;
```

```
//设置User-Agent的值  
connect.setRequestProperty ("User-Agent", "Mozilla/4.0  
(compatible ; MSIE 5.0 ; Windows NT ; DigExt) ") ;
```

//保存到数据流中

```
DataInputStream dis=new DataInputStream (
```

//获取数据流

```
connect.getInputStream () ) ;
```

//把数据放到Buffer中

```
BufferedReader in=new BufferedReader (
```

//根据实际情况指定编码，防止乱码

```
new InputStreamReader (dis, "GB2312") ) ;
```

//定义字符串

```
String readLine=null ;
```

//将数据读到字符串中

```
while ( (readLine=in.readLine () ) !=null) {  
  
    html=html+readLine ;  
  
}  
  
//关闭Buffer  
  
in.close () ;  
  
//以字符串方式返回页面信息  
  
return html ;  
  
}catch (MalformedURLException me) {  
  
}catch (IOException ioe) {  
  
}  
  
return null ;  
  
}  
  
//异步方法调用对返回的字符串进行处理  
  
protected void onPostExecute (String result) {
```

//对字符串进行处理

```
String html=ModifyingHtml (result) ;
```

//重新加载页面

```
wv.loadDataWithBaseUrl ("http://translate.google.cn/m",
```

```
result, "text/html", "UTF-8", "") ;
```

```
}
```

```
} ;
```

//处理页面过滤掉不需要的部分

```
String ModifyingHtml (String Html) {
```

//定义字符串

```
String html=Html ;
```

//新建Jsoup的Document对象

```
Document doc=Jsoup.parse (html) ;
```

//取出字符串中第3个div标签的内容

```
Element body=doc.select ("div") .get (2) ;
```

```
//将body中的字符串替换为""空内容字符串
```

```
body.text ("") ;
```

```
//解析语言列表页面
```

```
ParseLanguageList (doc.html ()) ;
```

```
//返回解析之后的内容
```

```
return doc.html () ;
```

```
}
```

```
//解析语言列表
```

```
void ParseLanguageList (String Html) {
```

```
//创建Jsoup中的Document对象
```

```
Document doc=Jsoup.parse (Html) ;
```

```
//找出其中的超链接
```

```
Elements links=doc.select ("a") ;
```

```
//对超链接元素的组合进行操作

for (int i=0 ; i<links.size () ; i++) {

//获取第i个超链接元素

Element link=links.get (i) ;

//得到其中的文本

String text=link.text () ;

//得到其中的链接地址

String linkHref=link.attr ("href") ;

//得到其中的文本

String linkText=link.text () ;

//打印得到的内容

Log.d (TAG, "text"+text) ;

Log.d (TAG, "linkHref"+linkHref) ;

Log.d (TAG, "linkText"+linkText) ;
```

}

}

}

从HTML中解析出的语言如图3-13所示。

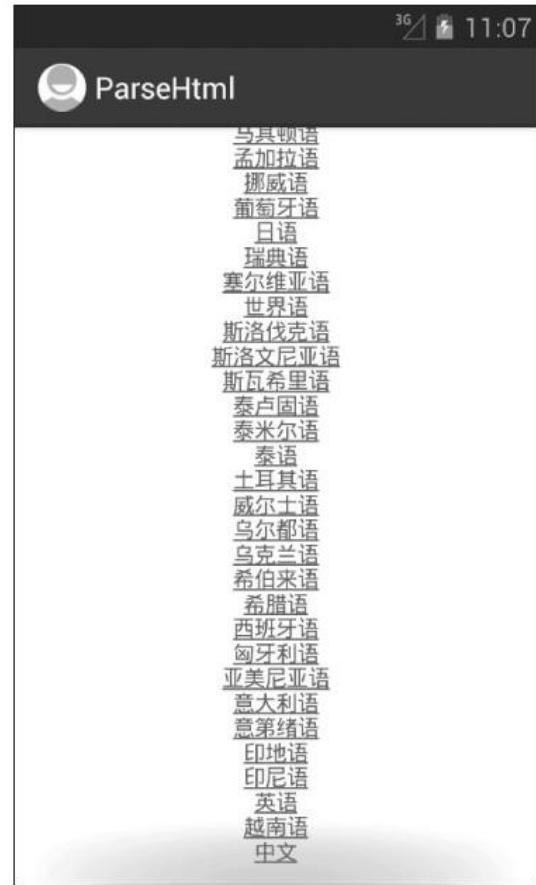
ParseHtml	text	德语
ParseHtml	linkRef	http://translate.google.cn/m?sl=de&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	德语
ParseHtml	text	俄语
ParseHtml	linkRef	http://translate.google.cn/m?sl=ru&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	俄语
ParseHtml	text	法语
ParseHtml	linkRef	http://translate.google.cn/m?sl=fr&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	法语
ParseHtml	text	菲律宾语
ParseHtml	linkRef	http://translate.google.cn/m?sl=tl&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	菲律宾语
ParseHtml	text	芬兰语
ParseHtml	linkRef	http://translate.google.cn/m?sl=fi&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	芬兰语
ParseHtml	text	格鲁吉亚语
ParseHtml	linkRef	http://translate.google.cn/m?sl=ka&tl=zh-CN&hl=zh-CN
ParseHtml	linkText	格鲁吉亚语
ParseHtml	text	古吉拉特语
ParseHtml	linkRef	http://translate.google.cn/m?sl=gu&tl=zh-CN&hl=zh-CN

图 3-13 Android从Google翻译页面解析出的语言

解析前和解析后的效果图比较如图3-14所示。



a) 解析前



b) 解析后

图 3-14 Android解析Google翻译页面

3.6 小结

本章详细介绍了HTTP协议，在此基础上，给出了Android中基于HTTP协议的文件上传案例。随后介绍了HttpClient和URLConnection等相关概念，并使用其访问维基百科。接着介绍了JSON和SOAP等相关概念，分别给出了JSON解析wikipedia及SOAP解析天气的案例。最后介绍了Android解析HTML的相关知识点。本章内容涉及面广，读者需要耐心研读并多加以实践。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第4章 Android常见网络接口编程

本章主要介绍与Android的Web Service、Android解析XML、Android订阅RSS、Android接发Email，以及Android网络安全等相关的编程。

4.1 Android解析和创建XML

4.1.1 XML简介

XML（Extensible Markup Language）即可扩展标记语言，其和HTML很类似，但又和HTML有很多不同之处。XML的设计宗旨是传输和存储数据，其焦点是数据的内容；而HTML被设计用来显示数据，其焦点是数据的外观。XML具有自我描述性，是W3C推荐的标准。

XML可应用于Web开发的许多方面，主要包括以下内容：

- XML把数据从HTML分离。如果需要在HTML文档中显示动态数据，那么每当数据改变时将要花费大量的时间来编辑HTML。通过XML，数据能够存储在独立的XML文件中。这样你就可以专注于使用HTML进行布局和显示，并确保修改底层数据不再需要对HTML进行任何的改变。通过使用几行JavaScript，就可以读取一个外部XML文件，然后更新HTML中的数据内容。
- XML简化数据共享。在真实的世界中，不兼容的计算机系统和应用程序使用不兼容的格式来存储数据。XML数据以纯文本格式进行存储，因此提供了一种独立于软硬件的数据存储方法。这使不同应用程序共享数据变得更加容易。

- XML简化数据传输。对开发人员来说，一项费时的挑战就是在Internet上不兼容的系统之间交换数据。通过XML，可以使不兼容的系统之间交换数据变得轻松。由于可以通过各种不兼容的应用程序来读取XML数据，所以大大降低了数据交换的复杂性。
- XML简化平台的变更。用户硬件或软件平台在升级或扩展过程中，往往需要转换大量的数据，致使不兼容的数据经常会丢失。而XML数据以文本格式存储，故使用XML，可使得各种软件或硬件平台在不损失数据的情况下，更容易扩展或升级。
- XML使数据更有用。由于XML独立于硬件、软件，故不同的软硬件都能够访问XML数据。例如用户不仅能在HTML网页中，也可以从XML数据源中访问XML数据；再如XML数据可供各种阅读设备使用（手持的计算机、语音设备、新闻阅读器等）。
- XML可用于创建新的Internet语言。很多新的Internet语言都是通过XML创建的，如XHTML、WSDL、WAP、WML、RSS、RDF、OWL和SMIL等。

XML最大的特点是标签没有被预定义，用户需要自行定义标签，不像HTML那样必须使用固定的预定义元素集。这就使得XML的语法规则较简单，非常容易学习和使用，人们可以使用XML描述任意类型的文

档。也许有的读者会认为XML是一种相当随意的标准，其实不然，XML有着严格的定义语法。

(1) 关于XML元素的命名规则

XML元素在命名的时候必须遵循以下规则：

- 名称可以含字母、数字以及其他字符；但不能以数字或者标点符号开始，下划线除外。
- 名称不能以xml（XML、Xml、xmL等）开始。
- 名称不能包含空格。
- 可使用任何名称，没有保留的字词。

命名XML元素的时候应尽量避免使用“-”、“.”、“：“等字符，名称应当比较简短，且具有可读性，尽量做到见名知意。

(2) XML文档必须有根元素

XML文档必须有一个元素是所有其他元素的父元素，该元素称为根元素。通常根元素是最外层的标签，XML有且只有一个根元素，其他元素必须嵌入其中。

```
<?xml version="1.0"encoding="gb2312"?>
```

```
<poem lang="chinese">  
  
<title>春晓</title>  
  
<author>孟浩然</author>  
  
<content>春眠不觉晓，处处闻啼鸟，夜来风雨声，花落知多少。  
</content>  
  
</poem>
```

这里<poem>是根元素，其下有<title>、<author>、<content>3个子元素，这样XML文档就形成了一种树结构，父元素下可以有子元素，子元素下还可以有子元素。就像一棵倒置的树从根部开始，并扩展到树的最底端。相同层级上的子元素称为同胞（兄弟或姐妹）。这种树结构可以形象地描述为图4-1所示的形式。



图 4-1 XML文档的树结构示例

(3) 每个开始标签必须要有一个结束标签

在XML中，省略结束标签是非法的。所有元素都必须有结束标签。前面使用的标签<poem>.....</poem>、<title>.....</title>、<author>.....</author>、<content>.....</content>都是成对的。此外，即使是空元素标签也必须要关闭，如<student></student>，亦等价于<student/>。

(4) 所有标签都要区分大小写

例如<title>.....</Title>这样的标签是不合法的。

(5) 所有标签都必须合理嵌套

也就是说，如果一个元素在另一个元素中开始，那么它必须在同一个元素中结束。例如，<poem><title>.....</poem></title>这样的嵌套就是不合法的。

(6) 所有标签的属性值必须用双引号或者单引号括起来

例如<poem lang="chinese">如果写成<poem lang=chinese>就是不合法的。

(7) XML中的空格问题

在XML中，文档中的空格不会被删除，会被原样保留下。

(8) 关于XML实体引用问题

在XML中，一些字符拥有特殊的意义。例如，若把字符“<”放在XML元素中，会发生错误，这是因为解析器会把它当作新元素的开始。比如以下这句：

```
<note>if count<10 then</note>
```

就是非法的。为此，应用实体引用来代替“<”字符，即：

```
<note>if count &lt; 10 then</note>
```

在XML中，有5个预定义的实体引用，如表4-1所示。

表 4-1 XML 中预定义的实体引用

实体引用	符 号	含 义
<	<	小于
>	>	大于
&	&	和号
'	'	单引号
"	"	引号

需要指出的是，在XML中，只有字符<和&是非法的，而>是合法的，但是使用实体引用来代替它是一个好习惯。

习惯上称符合XML语法标准的XML文档为“格式/形式良好”的XML文档（Well-Formed XML Documents）。

XML文档在逻辑上主要由以下5个部分组成。

(1) XML声明

XML文档总是以一个XML声明开始，它位于文档的第一行，且前面不能有任何字符。它指明了所用的XML版本、文档的编码、文档的独立性信息。其格式如下：

```
<?xml版本信息[编码信息][文档独立性信息]?>
```

[]部分表示可选信息。在XML声明中还可以加上文档编码信息，默认是UTF-8，如果要使用中文，可以在声明中加encoding="gb2312"。如果文档不依赖于外部文档，在XML声明中，可以通过standalone="yes"来声明这个文档是独立的文档。如果文档依赖于外部文档，可以通过standalone="no"来声明。例如：

```
<?xml version="1.0"encoding="gb2312"standalone="yes"?>
```

(2) XML文档类型声明

文档类型由DTD (Document Type Definition) 定义，文档类型声明有如下两种形式：

■声明DTD在一个外部文件中。此外部文件是以.dtd为扩展名的文本文件。形式为<!DOCTYPE根元素SYSTEM"文件名">，例如：

```
<!DOCTYPE poem SYSTEM"c.dtd">
```

■直接在XML文档中给出DTD，形式为<！DOCTYPE根元素[元素声明]>，例如：

```
<！DOCTYPE poem[  
    <！ELEMENT poem (title,author,content) >  
  
    <！ELEMENT title (#PCDATA) >  
  
    <！ELEMENT author (#PCDATA) >  
  
    <！ELEMENT content (#PCDATA) >  
]>
```

这部分代码也应该是第一种形式所举例子中c.dtd中的内容。关于DTD的相关语法，限于篇幅这里不再展开叙述。

另外，前面提到符合XML语法标准的XML文档就是“格式/形式良好”的XML文档，如果此XML文档同样遵守文档类型定义的语法规则，即是指通过DTD验证的XML文档，我们就称之为“合法”的XML（有的书上译为“有效”的XML）。

(3) 元素

在XML中，元素由开始标签、元素内容和结束标签构成。而空元素则由空元素标签构成。每一个元素都有一个用名字标识的类型，同时它可以有一个属性说明集，每一个属性说明都有一个名字和一个值。相关例子前面已经有很多了，常见的形式有如下几种。

■空元素，例如：

```
<student/>
```

■带有属性的空元素，例如：

```
<student name="zhang" age="20"/>
```

■带有内容的元素，例如：

```
<student>xueshengxinxi</name>zhang</name><age>20</age>
</student>
```

■带有内容和属性的元素，例如：

```
<student name="zhang"><age>20</age></student>
```

(4) XML注释

在XML中编写注释的语法与HTML的语法很相似，格式如下：

```
<!--This is a comment-->
```

(5) XML处理指令

处理指令的一般形式如下：

```
< ? target instruction ? >
```

其中，target是指令所指向的应用的名称。xml是保留名称，它是处理指令的一种类型，如我们前面看到的指令：`<?xml version="1.0" encoding="gb2312" standalone="yes"?>`。在XML文档中使用的指令取决于读取文档的处理器。通过标准的、预留处理指令来告诉浏览器怎样处理和显示文档。

我们仍以本节开头例子为例，加上文档类型声明，一个完整的“有效”XML文档如下：

```
<?xml version="1.0" encoding="utf-8"?>  
<!DOCTYPE poem[  
    <!ELEMENT poem (title,author,content) >  
    <!ELEMENT title (#PCDATA) >  
    <!ELEMENT author (#PCDATA) >  
    <!ELEMENT content (#PCDATA) >
```

]>

```
<poem lang="chinese">

<title>春晓</title>

<author>孟浩然</author>

<content>春眠不觉晓，处处闻啼鸟，夜来风雨声，花落知多少。
</content>

</poem>
```

在Android中，XML文件解析最常用的有DOM、SAX、PULL3种解析方式。下面就分别来介绍这3种解析方法。

4.1.2 DOM解析XML

DOM解析XML文件时，会将XML文件的所有内容以文档树的方式存放在内存中，然后允许用户使用DOM API遍历XML树、检索所需的数据。使用DOM解析XML的代码看起来是比较直观的，并且在某些方面比基于SAX的实现更加简单。但是，因为DOM需要将XML文件的所有内容以文档树的方式存放在内存中，所以内存的消耗比较大，特别是对于运行Android的移动设备来说，因为设备的资源比较宝贵，所以建

议还是采用SAX或者PULL来解析XML文件。当然，如果XML文件的内容比较小，采用DOM也是可行的。

DOM解析XML文件的基本思路如下：

- 1) 利用DocumentBuilderFactory创建一个DocumentBuilderFactory实例。
- 2) 利用DocumentBuilderFactory创建DocumentBuilder。
- 3) 加载XML文档 (Document) 。
- 4) 获取文档的根节点 (Element) 。
- 5) 获取根节点中所有子节点的列表 (NodeList) 。
- 6) 获取子节点列表中的需要读取的节点。

这里涉及Node接口的几个主要方法如表4-2所示。

表 4-2 Node 接口的主要方法

方 法	释 义
Short getNodeType()	该方法返回一个节点类型的常量，如对于 Element 节点，getNodeType() 方法返回的值为：Node.ELEMENT_NODE
NodeList getChildNodes()	返回一个由当前节点的所有子节点组成的 NodeList 对象
Node getChild()	返回当前节点的第一个子节点
Node getLastChild()	返回当前节点的最后一个子节点
NodeList getTextContent()	返回当前节点及所有子孙节点中的文本内容

在Android平台下，除了可以对应用程序的私有文件夹中的文件进行操作外，还可以从assets中获得输入流以读取数据。放在应用程序assets目录下的文件，在编译的时候会和其他文件一起被打包。这里在assets目录下新建一个名称为city.xml的XML文件，如图4-2所示。

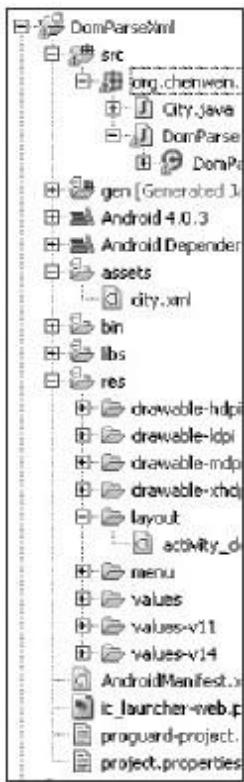


图 4-2 DOM解析XML实例中文件目录结构

city.xml中是本节要解析的XML文件，其文本内容如下：

```
<?xml version="1.0"?>
```

```
<citys>
```

```
<city id="0">  
  
<name>Alaska</name>  
  
<code>907</code>  
  
</city>  
  
<city id="1">  
  
<name>Chicago</name>  
  
<code>312</code>  
  
</city>  
  
<city id="2">  
  
<name>Hawaii</name>  
  
<code>808</code>  
  
</city>  
  
</citys>
```

首先，定义City类，其中包含对应的set和get方法，具体的代码如下：

```
public class City{  
  
    //定义城市名称  
  
    private String name ;  
  
    //定义城市代码  
  
    private String code ;  
  
    //定义城市的id  
  
    private String id ;  
  
    //定义id字符串对应的get方法  
  
    public String getId () {  
  
        return id ;  
  
    }  
  
    //定义id字符串对应的set方法  
  
    public void setId (String id) {  
  
        this.id=id ;  
    }  
}
```

}

//定义name字符串对应的get方法

public String getName () {

return name ;

}

//定义name字符串对应的set方法

public void setName (String name) {

this.name=name ;

}

//定义code字符串对应的get方法

public String getCode () {

return code ;

}

//定义code字符串对应的set方法

```
public void setCode (String code) {  
    this.code=code ;  
  
}  
  
//定义对象的toString方法，这里用作打印该对象的成员  
  
@Override  
  
public String toString () {  
  
    return"City[name='"+name+"， code='"+code+"， id='"+id+"']" ;  
  
}  
  
}
```

下面的代码以DOM方式解析上面的XML文本内容，其中使用到了上面定义的City对象。

```
import java.io.IOException ;  
  
import java.io.InputStream ;  
  
import java.util.ArrayList ;  
  
import java.util.List ;
```

```
import javax.xml.parsers.DocumentBuilder ;  
  
import javax.xml.parsers.DocumentBuilderFactory ;  
  
import javax.xml.parsers.ParserConfigurationException ;  
  
import org.w3c.dom.Document ;  
  
import org.w3c.dom.Element ;  
  
import org.w3c.dom.NodeList ;  
  
import org.xml.sax.SAXException ;  
  
import android.os.Bundle ;  
  
import android.app.Activity ;  
  
import android.util.Log ;  
  
import android.view.Menu ;  
  
//以DOM方式解析XML文本  
  
public class DomParseXml extends Activity{  
  
    //定义调试标签字符串TAG
```

```
private static final String TAG="DomParseXml" ;  
  
//定义静态常量字符串CITY_NAME  
  
private static final String CITY_NAME="name" ;  
  
//定义静态常量字符串CITY_ID  
  
private static final String CITY_ID="id" ;  
  
//定义静态常量字符串CITY_CODE  
  
private static final String CITY_CODE="code" ;  
  
@Override  
  
public void onCreate (Bundle savedInstanceState) {  
  
    super.onCreate (savedInstanceState) ;  
  
    //加载布局文件  
  
    setContentView (R.layout.activity_dom_parse_xml) ;  
  
    //调用解析方法  
  
    parseXml () ;
```

```
}
```

```
//解析XML文件，返回City对象的集合列表
```

```
public List<City>parseXml () {
```

```
//创建Citys对象列表，保存解析对象
```

```
List<City>Citys=new ArrayList<City> () ;
```

```
//定义DocumentBuilder对象
```

```
DocumentBuilder builder ;
```

```
//定义内容为null的DocumentBuilderFactory对象
```

```
DocumentBuilderFactory factory=null ;
```

```
//定义Document对象
```

```
Document document=null ;
```

```
//定义输入信息流
```

```
InputStream inputStream=null ;
```

```
//获取DocumentBuilderFactory类实例
```

```
factory=DocumentBuilderFactory.newInstance () ;  
  
try{  
  
    //获取实例  
  
    builder=factory.newDocumentBuilder () ;  
  
    //打开，并加载XML文件到输入流  
  
    inputStream=getResources () .getAssets () .open ("city.xml") ;  
  
    //将输入流的数据解析之后放到Document对象中  
  
    document=builder.parse (inputStream) ;  
  
    //使用getDocumentElement () 方法获取根节点  
  
    //这里的根Element为Citys  
  
    Element root=document.getDocumentElement () ;  
  
    //获取Citys节点下面的City节点  
  
    //XML文档中存在多个City节点，将其保存到列表中  
  
    NodeList nodes=root.getElementsByTagName ("city") ;
```

```
//声明一个City对象  
  
City city=null ;  
  
//遍历根节点所有子节点， Citys下所有的City  
  
for (int i=0 ; i<nodes.getLength () ; i++) {  
  
    //创建City实例  
  
    city=new City () ;  
  
    //获取city元素节点  
  
    Element cityElement= (Element) (nodes.item (i)) ;  
  
    //获取city中id属性值  
  
    city.setId (cityElement.getAttribute (CITY_ID)) ;  
  
    //获取city下name标签  
  
    Element cityNameElement=  
        (Element) cityElement.getElementsByName (CITY_NAME) .item  
        (0) ;  
  
    city.setName (cityNameElement.getFirstChild () .getNodeValue ()) ;
```

```
//获取city下code标签

Element cityCodeElement=
(Element) cityElement.getElementsByTagName ("CITY_CODE") .item
(0) ;

//获取code标签对应的值

city.setCode (cityCodeElement.

//获取第一个子节点的内容

getFirstChild () .getNodeValue () ) ;

//打印该city节点的信息

Log.d (TAG,city.toString () ) ;

//添加到集合

Citys.add (city) ;

}

}catch (IOException e) {

e.printStackTrace () ;
```

```
 }catch (ParserConfigurationException e) {  
  
    //指示一个严重的配置错误  
  
    e.printStackTrace () ;  
  
 }catch (SAXException e) {  
  
    //SAX错误或警告  
  
    e.printStackTrace () ;  
  
 }finally{  
  
 try{  
  
    //关闭输入流  
  
    inputStream.close () ;  
  
 }catch (IOException e) {  
  
    e.printStackTrace () ;  
  
 }  
 }  
 }
```

```
//返回解析结构

return Citys ;

}

//创建Menu菜单

@Override

public boolean onCreateOptionsMenu (Menu menu) {

getMenuInflater () .inflate (R.menu.activity_dom_parse_xml,menu) ;

return true ;

}

}
```

得到的Log如下：

D/DomParseXml (1027) : City[name=Alaska,code=907, id=0]

D/DomParseXml (1027) : City[name=Chicago,code=312, id=1]

D/DomParseXml (1027) : City[name=Hawaii,code=808, id=2]

4.1.3 SAX解析XML

SAX (Simple API for XML) 是基于事件驱动的，边加载边解析。当然Android的事件机制是基于回调函数的，在用SAX解析XML文档的时候，在读取到文档开始和结束标签时候就会回调一个事件，在读取到其他节点与内容的时候也会回调一个事件。

既然涉及事件，就有事件源和事件处理器。在SAX接口中，事件源是org.xml.sax包中的XMLReader，它通过parser () 方法来解析XML文档，并产生事件。事件处理器是org.xml.sax包中的ContentHandler、DTDHandler、ErrorHandler，以及EntityResolver这4个接口。

XMLReader通过相应事件处理器注册方法setXXXX () 来完成与ContentHandler、DTDHandler、ErrorHandler，以及EntityResolver这4个接口的连接，具体如表4-3所示。

表 4-3 XMLReader 接口中一些重要方法

处理器名称	处理事件	XMLReader 注册方法
ContentHandler	跟文档内容有关的事件：文档的开始与结束、XML 元素的开始与结束、可忽略的实体、名称空间前缀映射开始和结束、处理指令、字符数据和可忽略的空格等	setContentHandler(ContentHandler h)
ErrorHandler	处理 XML 文档时产生的错误	setErrorHandler(ErrorHandler h)
DTDHandler	处理对文档的 DTD 进行解析时产生的相应事件	setDTDHandler(DTDHandler h)
EntityResolver	处理外部实体	setEntityResolver(EntityResolver r)

程序员定义的事件处理类在实现该接口时，必须实现接口中的所有4个方法。为了减少程序员的工作量，SDK提供了DefaultHandler类来做事件处理。DefaultHandler类的一些主要事件的回调方法如表4-4所示。

表 4-4 DefaultHandler 类的主要方法

方 法	含 义
setDocumentLocator(Locator locator)	设置一个可以定位文档位置的对象
startDocument()	用于处理文档解析开始事件
startElement(String uri, String localName, String qName, Attributes attributes)	处理元素开始事件，从参数中可以获取元素所在的URL、元素名称、属性列表等信息
characters(char[] ch, int start, int length)	处理元素的字符内容，从参数中可以获得内容
endElement(String uri, String localName, String qName)	处理元素结束事件，参数中可以获得元素所在 URL、元素名称等
endDocument()	用于处理文档解析的结束事件

因此，可以通过XMLReader及DefaultHandler的配合来解析XML。基本思路如下：

- 1) 创建SAXParserFactory对象；
- 2) 根据SAXParserFactory.newSAXParser () 方法返回一个SAXParser解析器；
- 3) 根据SAXParser解析器获取事件源对象XMLReader；
- 4) 实例化一个DefaultHandler对象；
- 5) 连接事件源对象XMLReader到事件处理类DefaultHandler中；

- 6) 调用XMLReader的parse方法从输入源中获取的XML数据；
- 7) 通过DefaultHandler返回需要的数据集合。

解析的XML文件名称为link.xml，放到项目的raw目录下面。raw目录在Android里面用于存放自定义的资源文件，在编译的时候会打包到项目里面。

XML文件的引用方式如下：

```
getResources () .openRawResource (R.raw.link) ;
```

SAX解析XML实例的相关文件目录如图4-3所示。

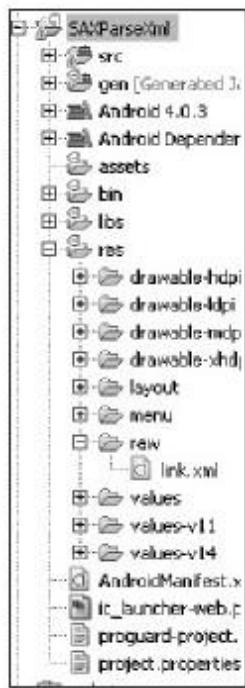


图 4-3 SAX解析XML实例中文件目录结构

在link.xml文件中给出了多个链接以及链接对应的文字说明，其具体的内容如下：

```
<?xml version="1.0"?>

<p>

<a href="http://news.baidu.com"name="tj_news">新闻</a>

<a href="http://www.baidu.com"name="tj_www">网页</a>

<a href="http://tieba.baidu.com"name="tj_tieba">贴吧</a>

<a href="http://zhidao.baidu.com"name="tj_zhidao">知道</a>

<a href="http://mp3.baidu.com"name="tj_mp3">MP3</a>

<a href="http://image.baidu.com"name="tj_img">图片</a>

<a href="http://video.baidu.com"name="tj_video">视频</a>

<a href="http://map.baidu.com"name="tj_map">地图</a>

</p>
```

下面的代码使用SAX的方法解析上面的XML文件，其主要实现代码如下：

```
import java.io.IOException ;  
  
import java.io.InputStream ;  
  
//下面引入SAX解析相关的类  
  
import javax.xml.parsers.ParserConfigurationException ;  
  
import javax.xml.parsers.SAXParser ;  
  
import javax.xml.parsers.SAXParserFactory ;  
  
import org.xml.sax.InputSource ;  
  
import org.xml.sax.SAXException ;  
  
import org.xml.sax.XMLReader ;  
  
import android.os.Bundle ;  
  
import android.app.Activity ;  
  
import android.view.Menu ;  
  
//使用SAX方式解析XML文本  
  
public class SAXParseXml extends Activity{
```

```
//定义调试用字符串TAG
```

```
private static final String TAG="SAXParseXml" ;
```

```
@Override
```

```
public void onCreate (Bundle savedInstanceState) {
```

```
super.onCreate (savedInstanceState) ;
```

```
//加载对应的布局文件
```

```
setContentView (R.layout.activity_saxparse_xml) ;
```

```
//调用解析XML文件的方法
```

```
parseXml () ;
```

```
}
```

```
//用SAX方式解析XML
```

```
private void parseXml () {
```

```
//实例化一个SAXParserFactory对象
```

```
SAXParserFactory factory=SAXParserFactory.newInstance () ;
```

```
//创建一个解析器对象SAXParser  
  
SAXParser parser ;  
  
//创建一个SAX解析的帮助类SAXPraserHelper  
  
//SAXPraserHelper的具体定义在后面的内容里面有介绍  
  
SAXPraserHelper helperHandler=null ;  
  
//实例化SAXParser对象， 创建XMLReader对象， 解析器  
  
try{  
  
    //实例化SAXParserFactory  
  
    parser=factory.newSAXParser () ;  
  
    //调用XMLReader的parse方法从输入源中获取到的XML数据  
  
    XMLReader xmlReader=parser.getXMLReader () ;  
  
    //实例化handler， 事件处理器  
  
    helperHandler=new SAXPraserHelper () ;  
  
    //解析器注册事件
```

```
xmlReader.setContentHandler (helperHandler) ;  
  
//读取XML文件到输入流中  
  
InputStream stream=  
  
//获取XML文件  
  
getResources () .getAssets () .open ("link.xml") ;  
  
InputSource is=new InputSource (stream) ;  
  
//解析文件  
  
xmlReader.parse (is) ;  
  
}catch (ParserConfigurationException e) {  
  
//指示一个严重的配置错误  
  
e.printStackTrace () ;  
  
}catch (SAXException e) {  
  
//SAX错误或警告  
  
e.printStackTrace () ;
```

```
 } catch (IOException e) {  
     e.printStackTrace();  
 }  
  
 //调用帮助类中的获取链接的方法  
 helperHandler.getLinks();  
 }  
  
 //创建menu菜单  
 @Override  
 public boolean onCreateOptionsMenu(Menu menu) {  
     getMenuInflater().inflate(R.menu.activity_saxparse_xml,menu);  
     return true;  
 }  
 }
```

针对XML文件的结构，定义Link类，其成员包含链接、文本、名称，具体的定义代码如下：

```
public class Link{  
  
    //定义成员变量href表示超链接字符串  
  
    private String href ;  
  
    //定义成员变量text表示超链接对应的文本  
  
    private String text ;  
  
    //定义成员变量name表示超链接对应的名称  
  
    private String name ;  
  
    //href变量的get方法， 获取href的值  
  
    public String getHref () {  
  
        return href ;  
  
    }  
  
    //href变量的set方法， 为href变量赋值  
  
    public void setHref (String href) {  
  
        this.href=href ;
```

}

//text变量的get方法， 获取text变量的值

```
public String getText () {
```

```
    return text ;
```

}

//text变量的set方法， 为text变量赋值

```
public void setText (String text) {
```

```
    this.text=text ;
```

}

//name变量的get方法， 获取name变量

```
public String getName () {
```

```
    return name ;
```

}

//name变量的set方法， 为name变量赋值

```
public void setName (String name) {  
    this.name=name ;  
}  
  
//将对象成员变量输出为可读的字符串  
  
@Override  
  
public String toString () {  
    return"Link[href='"+href+"， text='"+text+"， name='"+name+"']" ;  
}  
  
}
```

解析代码如下：

```
import java.util.ArrayList ;  
  
import java.util.List ;  
  
import org.xml.sax.Attributes ;  
  
import org.xml.sax.SAXException ;
```

```
import org.xml.sax.helpers.DefaultHandler ;  
  
import android.util.Log ;  
  
//SAX解析的帮助类  
  
public class SAXPraserHelper extends DefaultHandler{  
  
    //定义调试标签字符串TAG  
  
    private static final String TAG="SAXPraserHelper" ;  
  
    //定义Link对象列表  
  
    List<Link>Links ;  
  
    //新建Link对象  
  
    Link link ;  
  
    //定义当前状态的值  
  
    int currentState=0 ;  
  
    //获取Links  
  
    public List<Link>getLinks () {
```

```
//遍历列表

for (int i=0 ; i<Links.size () ; i++) {

    Log.d (TAG,Links.toString () ) ;

}

//返回Links

return Links ;

}

//接收字符块通知

public void characters (char[]ch,int start,int length) throws
SAXException{

//获取字符数组cn，从第start开始的长度为length的子字符数组

String theString=String.valueOf (ch,start,length) ;

//判断当前状态是否为0

if (currentState !=0) {

//没有完成则将字符串发到Link对象中
```

```
link.setText (theString) ;  
  
//将状态置为0  
  
currentState=0 ;  
  
}  
  
return ;  
  
}  
  
//接收文档结束通知  
  
public void endDocument () throws SAXException{  
  
super.endDocument () ;  
  
}  
  
//接收标签结束通知  
  
public void endElement (String uri,String localName,String qName)  
throws SAXException{  
  
//判断标签是否为a
```

```
if (localName.equals ("a") )  
  
    Links.add (link) ;  
  
}  
  
//处理文档解析开始事件  
  
@Override  
  
public void startDocument () throws SAXException{  
  
    Links=new ArrayList<Link> () ;  
  
}  
  
//处理元素开始事件  
  
public void startElement (String uri,String localName,String  
qName,Attributes attributes) throws SAXException{  
  
    //创建link类  
  
    link=new Link () ;  
  
    //当前为a标签，为链接  
  
    if (localName.equals ("a") ) {
```

```
//为link赋值

for (int i=0 ; i<attributes.getLength () ; i++) {

//如果存在属性href

if (attributes.getLocalName (i) .equals ("href") ) {

//为link对象的href属性赋值

link.setHref (attributes.getValue (i) ) ;

//如果存在属性name

}else if (attributes.getLocalName (i) .equals ("name") ) {

//为link对象的name属性赋值

link.setName (attributes.getValue (i) ) ;

}

}

//将当前状态赋值为1

currentState=1 ;
```

```
    return ; }

//将当前状态赋值为0

currentState=0 ;

return ;

}
```

解析结果如下：

D/SAXPraserHelper (1660) :

```
[Link[href=http://news.baidu.com,text=新闻, name=tj_news],  
Link[href=http://www.baidu.com,text=网页, name=tj_www],  
Link[href=http://tieba.baidu.com,text=贴吧, name=tj_tieba],  
Link[href=http://zhidao.baidu.com,text=知道, name=tj_zhidao],  
Link[href=http://mp3.baidu.com,text=MP3, name=tj_mp3],  
Link[href=http://image.baidu.com,text=图片, name=tj_img],
```

Link[href=http://video.baidu.com, text=视频, name=tj_video],

Link[href=http://map.baidu.com, text=地图, name=tj_map]]

4.1.4 PULL解析XML

PULL解析XML的方式与SAX解析XML的方式一样，它也是基于事件驱动的。使用PULL解析器解析XML文件时应注意以下几点：

- 通过`xml.newPullParser ()` 获得解析器。
- 通过`parser.setInput (in, "UTF-8")` 设置输入流以及编码。
- 通过`parser.next ()` 获取下一个元素并触发相应事件。

`xmlPullParser`中定义了常量来标识各种解析事件，如表4-5所示。

表 4-5 `xmlPullParser` 中定义的常量

常量	常量所标识的解析事件
START_DOCUMENT	读取到 XML 的声明返回
END_DOCUMENT	读取到 XML 的结束返回
START_TAG	读取到 XML 的开始标签返回
END_TAG	读取到 XML 的结束标签返回
TEXT	读取到 XML 的文本返回

李白的《静夜思》可以用XML来表示，其对应的XML内容如下：

```
<?xml version="1.0"?>
```

```
<poem lang="chinese">

<title>静夜思</title>

<author>李白</author>

<content>床前明月光，疑是地上霜。举头望明月，低头思故乡。

</content>

</poem>
```

针对上面的XML文本，使用PULL方式解析，其代码如下：

```
//使用PULL方式解析XML文本
```

```
private void pareXml ()
```

```
{
```

```
//创建XmlPullParserFactory类
```

```
XmlPullParserFactory factory ;
```

```
try{
```

```
//实例化XmlPullParserFactory类
```

```
factory=XmlPullParserFactory.newInstance () ;  
  
//设置支持名称空间  
  
factory.setNamespaceAware (true) ;  
  
//创建XmlPullParser类  
  
XmlPullParser xpp=factory.newPullParser () ;  
  
//将上面的XML文本读入XmlPullParser解析器中  
  
xpp.setInput (new StringReader ("<poem lang=\"chinese\"><title>静  
夜思</title><author>李白</author><content>床前明月光， 疑是  
地上霜。举头望明月， 低头思故乡。</content></poem>") ) ;  
  
//定义Pull解析常用事件类型  
  
int eventType=xpp.getEventType () ;  
  
//查看是否为XML文档的逻辑末尾， 输入流的末尾  
  
while (eventType !=XmlPullParser.END_DOCUMENT) {  
  
//查看是否为文档开始类型， 对应<?xml version="1.0"encoding="UTF-  
8"?>
```

```
if (eventType==XmlPullParser.START_DOCUMENT) {  
  
    Log.d (TAG, "Start document") ;  
  
}else if (eventType==XmlPullParser.START_TAG) {  
  
//查看是否为XML标签的开始  
  
    Log.d (TAG, "Start tag"+xpp.getName () ) ;  
  
}else if (eventType==XmlPullParser.END_TAG) {  
  
//查看是否为XML标签的结束  
  
    Log.d (TAG, "End tag"+xpp.getName () ) ;  
  
}else if (eventType==XmlPullParser.TEXT) {  
  
//查看是否为XML中的文本内容  
  
    Log.d (TAG, "Text"+xpp.getText () ) ;  
  
}  
  
//获得下一个解析事件  
  
eventType=xpp.next () ;
```

```
}

}catch (XmlPullParserException e) {

//解析异常

e.printStackTrace () ;

}catch (IOException e) {

e.printStackTrace () ;

}

Log.d (TAG, "End document") ;
```

解析结果如下：

10-22 03:51:07.832:D/PullParseXml (710) : Start document

10-22 03:51:07.832:D/PullParseXml (710) : Start tag poem

10-22 03:51:07.842:D/PullParseXml (710) : Start tag title

10-22 03:51:07.842:D/PullParseXml (710) : Text 静夜思

10-22 03:51:07.842:D/PullParseXml (710) : End tag title

10-22 03:51:07.842:D/PullParseXml (710) : Start tag author

10-22 03:51:07.842:D/PullParseXml (710) : Text李白

10-22 03:51:07.852:D/PullParseXml (710) : End tag author

10-22 03:51:07.852:D/PullParseXml (710) : Start tag content

10-22 03:51:07.862:D/PullParseXml (710) : Text床前明月光，疑是地
上霜。举头望明月，低头思故乡。

10-22 03:51:07.872:D/PullParseXml (710) : End tag content

10-22 03:51:07.872:D/PullParseXml (710) : End tag poem

10-22 03:51:07.872:D/PullParseXml (710) : End document

4.1.5 实战案例：Android中创建XML

本案例介绍在Android中创建XML文件。案例使用XmlSerializer在
Android中创建XML文件，具体做法是，先在Android的manifest.xml文
档中加入下面的内容，给应用读写SD卡的权限。

<!--允许操作外部存储设备文件-->

```
<uses-permission  
    android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

```
<!--允许挂载和反挂载文件系统可移动存储-->
```

```
<uses-permission  
    android:name="android.permission.MOUNT_UNMOUNT_FILESYSTEMS  
    "/>
```

在Android中使用XmlSerializer创建XML文件的示例代码如下：

```
import java.io.File ;  
  
import java.io.FileNotFoundException ;  
  
import java.io.FileOutputStream ;  
  
import java.io.IOException ;  
  
import org.xmlpull.v1.XmlSerializer ;  
  
import android.os.Bundle ;  
  
import android.os.Environment ;  
  
import android.app.Activity ;
```

```
import android.util.Xml ;  
  
import android.view.Menu ;  
  
//使用XmlSerializer创建XML文件  
  
public class CreateXML extends Activity  
  
{  
  
    @Override  
  
    public void onCreate (Bundle savedInstanceState)  
  
    {  
  
        super.onCreate (savedInstanceState) ;  
  
        //加载布局文件  
  
        setContentView (R.layout.activity_create_xml) ;  
  
        //在SD卡上创建名为poem.xml的文件  
  
        File newxmlfile=new File ('  
  
        //指定文件路径
```

```
Environment.getExternalStorageDirectory () +"/poem.xml") ;  
  
try{  
  
//创建新文件  
  
newxmlfile.createNewFile () ;  
  
}  
}  
  
}catch (IOException e) {  
  
e.printStackTrace () ;  
  
}  
  
//定义文件输出流  
  
FileOutputStream fileos=null ;  
  
try{  
  
//将文件newxmlfile放到文件输出流中  
  
fileos=new FileOutputStream (newxmlfile) ;  
  
}  
}  
  
}catch (FileNotFoundException e) {  
  
//文件不存在的异常
```

```
e.printStackTrace () ;  
}  
  
//创建XmlSerializer对象  
  
XmlSerializer serializer=Xml.newSerializer () ;  
  
try{  
  
    //指定编码为UTF-8  
  
    serializer.setOutput (fileos, "UTF-8") ;  
  
    //写入XML开始的标志  
  
    serializer.startDocument (null,Boolean.valueOf (true) ) ;  
  
    //设置属性  
  
    serializer.setFeature (  
        "http://xmlpull.org/v1/doc/features.html#indent-output", true) ;  
  
    //设置开始标签为poem  
  
    serializer.startTag (null, "poem") ;
```

```
//添加属性值lang="chinese"

serializer.attribute (null, "lang", "chinese") ;

//按照其树形结构， 定义其子标签为title

serializer.startTag (null, "title") ;

//添加标签<title>之间的文本内容

serializer.text ("静夜思") ;

//添加<title>结束标签

serializer.endTag (null, "title") ;

//添加<author>开始标签

serializer.startTag (null, "author") ;

//添加标签<author>之间的文本

serializer.text ("李白") ;

//添加<author>结束标签

serializer.endTag (null, "author") ;
```

```
//添加<content>开始标签  
  
serializer.startTag (null, "content") ;  
  
//在<content>标签之间添加文本内容  
  
serializer.text ("床前明月光，疑是地上霜。举头望明月，低头思故  
乡。") ;  
  
//添加<content>结束标签  
  
serializer.endTag (null, "content") ;  
  
//添加<poem>结束标签  
  
serializer.endTag (null, "poem") ;  
  
//写入XML逻辑结尾  
  
serializer.endDocument () ;  
  
//将XML数据写入FileOutputStream中  
  
serializer.flush () ;  
  
//关闭数据流  
  
fileos.close () ;
```

```
}

catch (Exception e) {

    e.printStackTrace () ;

}

//创建Menu菜单

@Override

public boolean onCreateOptionsMenu (Menu menu)

{

    getMenuInflater () .inflate (R.menu.activity_create_xml,menu) ;

    return true ;

}

}

程序运行后，我们用DDMS查看SD卡中的内容，如图4-4所示。
```

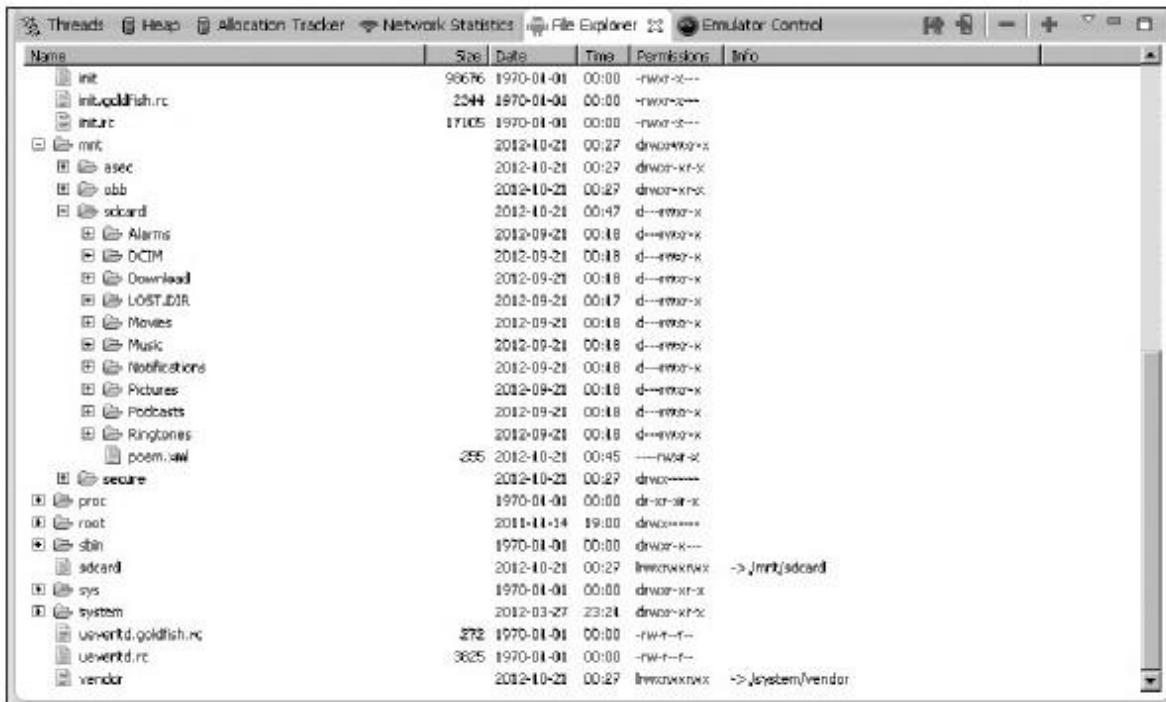


图 4-4 DDMS查看SD卡中内容

从中导出poem.xml文件，打开之后其内容如下：

```
<?xml version='1.0'encoding='UTF-8'standalone='yes'?>

<poem lang="chinese">

<title>静夜思</title>

<author>李白</author>

<content>床前明月光，疑是地上霜。举头望明月，低头思故乡。

</content>
```

</poem>

4.2 Android订阅RSS

4.2.1 RSS简介

1.RSS定义

RSS即Really Simple Syndication，简易信息聚合。RSS是一种描述和同步网站内容的格式，用于与其他站点之间共享内容（也叫聚合内容），发布感兴趣的信息。RSS目前广泛用于网上新闻频道，如BLOG和WIKI等。

信息传播工具多种多样，包括可以免费收听的无线电广播、公共和有线电视、印刷媒体，甚至包括Internet这样颠覆性的技术，以及Internet下的庞大的Web站点和电子邮件订阅。虽然信息传播工具可选择的很多，但是这些工具都存在一个问题：很难在庞大的海量数据中精确查找到用户真正感兴趣的信息。而RSS可以帮助我们解决这个问题。RSS是一种内容发布者用来高效发布信息的XML数据格式，这些信息经过了分类并适合人机阅读。RSS提要通常使用诸如新闻阅读器这种人类可读的友好格式进行处理并显示给用户。

RSS是基于文本格式的，是XML语言的一种形式，数据包含完整信息源的简单摘要。如果对这些摘要感兴趣，用户可以进一步获取信息并获得全部内容。RSS通常被用于新闻和其他按顺序排列的网站。RSS文

件（RSS files，又称RSS feeds或channels）是XML格式的文档，通常只包含简单的项目列表。每一个项目都含有一个标题，一段简单的介绍，还有一个URL链接，亦可包含一些其他的信息，例如日期、创建者等。一段项目的介绍可能包含新闻的全部介绍，或者仅是额外的内容或者简短的介绍。这些项目的链接通常都能链接到全部的内容。网络用户可以在客户端借助于支持RSS的新闻聚合工具软件（例如NewzCrawler、FeedDemon），在不打开网站内容页面的情况下阅读支持RSS输出的网站内容。

2.RSS的使用

通过RSS，可以订阅如下内容：

- 订阅BLOG（在BLOG上，可以订阅工作中所需的技术文章，也可以订阅与你有共同爱好的作者的日志。总之，你对什么感兴趣就可以订阅什么）。
- 订阅新闻（无论是奇闻怪事、明星消息，还是体坛风云，只要你想知道，都可以订阅）。

用了RSS，你再也不用一个网站一个网站、一个网页一个网页地去逛了。只要将你需要的内容订阅在一个RSS阅读器中，这些内容就会自动出现在你的阅读器里。你也不必为了一个急切想知道的消息而不断地

刷新网页了，因为一旦有了更新，RSS阅读器就会主动通知你。网站提供RSS输出，有利于让用户发现网站内容的更新。

如果没有RSS，你就不得不每日都来相关网站查看新的内容。对许多用户来说这样太费时了。通过RSS feeds，用户可以使用RSS聚合器来更快地检查网站更新。由于RSS数据很小巧，并可快速加载，故可轻易地被类似移动电话或PDA等设备的服务使用。拥有相似内容的网站，通过RSS可以轻易地在网站上共享彼此内容，使这些网站更出色，更有价值。

3.RSS文件形式

RSS文件由一个<channel>元素及其子元素组成。除了频道内容本身之外，<channel>还以项的形式包含表示频道元数据的元素，比如<title>、<link>和<description>。项通常是频道的主要部分，包含经常变化的内容。

(1) 频道

频道一般有3个元素，提供关于频道本身的信息。

- <title>：频道或提要的名称。
- <link>：与该频道关联的Web站点或者站点区域的URL。

■<description>：简要介绍该频道是做什么的。

许多频道子元素都是可选的。常用的有<image>、<language>、<copyright>、<managingEditor>、<webMaster>、<pubDate>、<lastBuildDate>、<category>、<generator>、<docs>、<cloud>、<ttl>、<rating>、<textInput>、<skipHours>、<skipDays>等，这些都非常容易理解，不再赘述。

(2) 项

项通常是提要中最重要的部分。每个项都可以是关于某个weblog、完整文档、电影评论、分类广告或者任何希望与频道连锁的内容的记录。频道中的其他元素可能不变，但项经常发生变化。

可以有任意多个项。以前的规范限值为15个项，如果要保持向后兼容的话，这仍然是一个必要的上限。一个新闻项通常包含以下3个元素。

■<title>：这是项的名称，在标准应用中被转换成HTML中的标题。

■<link>：这是项的URL。title通常作为一个链接，指向包含在<link>元素中的URL。

■<description>：通常作为link中所指向的URL的摘要或者补充说明。

所有的元素都是可选的，但是一个项至少包含一个<title>，或包含一个<description>。项还有其他一些可选的元素，如<author>、<category>、<comments>、<enclosure>、<guid>、<pubDate>、<source>等。

下面是一个RSS页面的源代码：

```
<?xml version="1.0"?>

<rss version="2.0">

<channel>

<title>IBM developerWorks中国：文档库</title>

<link>http://www.ibm.com/developerworks/cn/</link>

<description>来自IBM developerWorks中国网站的最新内容
</description>

<pubDate>29 Oct 2012 00:11:08+0000</pubDate>

<language>zh-CN</language>

<copyright>Copyright 2004 IBM Corporation.</copyright>

<image>
```

<title>IBM developerWorks中国</title>

<url><http://www.ibm.com/developerworks/i/dwlogo-small.gif></url>

<link><http://www.ibm.com/developerworks/cn/></link>

</image>

<item>

<title>< ! [CDATA[应用IBM Systems Director可定制的角色实现安全的平台管理]]></title>

<description>< ! [CDATA[本文通过实际场景，介绍如何应用IBM Systems Director Server的RBAC (role-based access control) service创建可定制化的角色，将角色分配给独立的用户和组，并且结合外部的LDAP服务器提供用户身份认证的服务，从而为ISD提供更加灵活和高效的安全管理解决方案。]]></description>

<link>< ! [CDATA[http://www.ibm.com/developerworks/cn/aix/library/1210_fengwei_isdrbac/index.html?ca=drs-]]></link>

<pubDate>25 Oct 2012 04:00:00+0000</pubDate>

</item>

```
<item>.....</item>
```

```
</channel>
```

```
</rss>
```

4.2.2 实战案例：简单RSS阅读器

本节案例介绍如何在Android中创建简单的RSS阅读器。前文已经介绍了RSS是XML格式的文本文件，需要解析，这里采用SAX解析方式。读者也可以根据上一节的内容介绍，将本案例改写为其他解析方式。

RSS阅读器的布局文件如下：

```
<RelativeLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:tools="http://schemas.android.com/tools"
```

```
        android:layout_width="match_parent"
```

```
        android:layout_height="match_parent">
```

```
        <!--使用Listview控件，显示RSS条目-->
```

```
        <!--使用id属性为id/android:list表示使用默认的样式-->
```

```
<ListView android:id="@+id/android:list"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
</ListView>
</RelativeLayout>
```

下面的代码使用SAX方法来解析RSS文件，其具体内容如下：

```
import java.util.List ;
import org.xml.sax.Attributes ;
import org.xml.sax.SAXException ;
import org.xml.sax.helpers.DefaultHandler ;
//使用SAX方法来解析RSS文件
public class RSSHandler extends DefaultHandler
{
    //定义解析名称字符串
```

```
private String XMLParseName ;  
  
//定义RSS的对象列表  
  
private List<RSS>RSSList ;  
  
//定义RSS对象  
  
private RSS rss ;  
  
//定义构造函数  
  
public RSSHandler (List<RSS>NewsList) {  
  
this.RSSList=NewsList ;  
  
}  
  
//处理元素的字符内容，从参数中可以获得内容  
  
@Override  
  
public void characters (char[]ch,int start,int length) throws  
SAXException{  
  
super.characters (ch,start,length) ;  
  
//获取字符数组cn，从第start开始的长度为length的子字符数组
```

```
String value=new String (ch,start,length) ;  
  
//判断字符串是否等于title  
  
if (XMLParseName.equals ("title") ) {  
  
//将值保存到RSS对象中  
  
rss.setTitle (value) ;  
  
}  
else if (XMLParseName.equals ("link") ) {  
  
//判断字符串是否等于link  
  
//将值保存到RSS对象中  
  
rss.setLink (value) ;  
  
}  
else if (XMLParseName.equals ("description") ) {  
  
//判断字符串是否等于description  
  
//将值保存到RSS对象中  
  
rss.setDescription (value) ;  
  
}  
else if (XMLParseName.equals ("pubDate") ) {
```

```
//判断字符串是否等于pubDate  
  
//将值保存到RSS对象中  
  
rss.setPubDate (value) ;  
  
}  
  
}  
  
//处理文档解析开始事件  
  
@Override  
  
public void startDocument () throws SAXException{  
  
super.startDocument () ;  
  
}  
  
//处理元素开始事件  
  
@Override  
  
public void startElement (String uri,String localName,  
String qName,Attributes attributes) throws SAXException{
```

```
super.startElement (uri,localName,qName,attributes) ;  
  
if (localName.equals ("title") ) {  
  
//检测到开始，则新建RSS  
  
rss=new RSS () ;  
  
}  
  
XMLParseName=localName ;  
  
}  
  
//处理元素结束事件  
  
@Override  
  
public void endElement (String uri,String localName,String qName)  
throws SAXException{  
  
super.endElement (uri,localName,qName) ;  
  
//查看是否等于item  
  
if (localName.equals ("item") ) {
```

```
//将对象添加到RSSList

RSSList.add (rss) ;

}

}

//处理文档解析的结束事件

@Override

public void endDocument () throws SAXException

{

super.endDocument () ;

}

}
```

上面的解析里面用到了RSS实体类， RSS实体类负责保存RSS的相关信息， 包含标题、链接、描述、发布日期等， 其代码如下：

```
//RSS实体类

public class RSS{
```

//标题

private String title ;

//链接

private String link ;

//描述

private String description ;

//发布日期

private String pubDate ;

//参数为空的构造函数

public RSS () {}

//含有参数的构造函数

public RSS (String title,String link,String description,String pubDate) {

super () ;

//为成员变量title赋值

```
this.title=title ;  
  
//为成员变量link赋值  
  
this.link=link ;  
  
//为成员变量description赋值  
  
this.description=description ;  
  
//为成员变量pubDate赋值  
  
this.pubDate=pubDate ;  
  
}  
  
//获取title成员变量的值  
  
public String getTitle () {  
  
    return title ;  
  
}  
  
//为成员变量title赋值  
  
public void setTitle (String title) {
```

```
this.title=title ;
```

```
}
```

```
//获取成员变量link的值
```

```
public String getLink () {
```

```
return link ;
```

```
}
```

```
//为成员变量link赋值
```

```
public void setLink (String link) {
```

```
this.link=link ;
```

```
}
```

```
//获取成员变量description的值
```

```
public String getDescription () {
```

```
return description ;
```

```
}
```

//为成员变量description赋值

```
public void setDescription (String description) {  
    this.description=description ;  
}
```

//获取成员变量pubDate的值

```
public String getPubDate () {  
    return pubDate ;  
}
```

//为成员变量pubDate赋值

```
public void setPubDate (String pubDate) {  
    this.pubDate=pubDate ;  
}
```

RSS阅读器的主界面，负责下载RSS文档、解析RSS文档并且显示，其相关代码如下：

```
import java.io.BufferedReader ;  
  
import java.io.IOException ;  
  
import java.io.InputStreamReader ;  
  
import java.io.StringReader ;  
  
import java.net.HttpURLConnection ;  
  
import java.net.MalformedURLException ;  
  
import java.net.URL ;  
  
import java.util.ArrayList ;  
  
import java.util.HashMap ;  
  
import java.util.Iterator ;  
  
import java.util.List ;  
  
import java.util.Map ;  
  
import javax.xml.parsers.ParserConfigurationException ;  
  
import javax.xml.parsers.SAXParser ;
```

```
import javax.xml.parsers.SAXParserFactory ;  
  
import org.xml.sax.InputSource ;  
  
import org.xml.sax.SAXException ;  
  
import org.xml.sax.XMLReader ;  
  
import android.os.AsyncTask ;  
  
import android.os.Bundle ;  
  
import android.app.ListActivity ;  
  
import android.view.Menu ;  
  
import android.widget.SimpleAdapter ;  
  
//简单的RSS阅读器  
  
public class RssReaderActivity extends ListActivity  
  
{  
  
    //新建RSSHandler类  
  
    private RSSHandler rssHandler ;
```

```
//新建RSS对象列表，保存解析得到的RSS对象
```

```
private List<RSS>rssList ;
```

```
//定义加载到界面的数据
```

```
ArrayList<Map<String, Object>>mData=null ;
```

```
public void onCreate (Bundle savedInstanceState) {
```

```
super.onCreate (savedInstanceState) ;
```

```
//设置布局文件
```

```
setContentView (R.layout.activity_rss_reader) ;
```

```
//获取ListView的对象
```

```
getListView () ;
```

```
//异步执行下载任务
```

```
new DownloadFilesTask ().execute
```

```
("http://www.msn.com/rss/msnmoney.aspx") ;
```

```
//初始化列表大小
```

```
rssList=new ArrayList<RSS> (100) ;
```

```
//对rssList进行处理  
  
rssHandler=new RSSHandler (rssList) ;  
  
//初始化mData列表  
  
mData=new ArrayList<Map<String, Object>> () ;  
  
}  
  
//异步执行下载任务  
  
private class DownloadFilesTask extends AsyncTask<String, Integer, String> {  
  
    protected String doInBackground (String.....urls) {  
  
        //定义数据源字符串  
  
        String XmlSourceStr=null ;  
  
        //创建URL对象  
  
        URL url=null ;  
  
        try{  
  
            //初始化URL对象
```

```
//其参数为DownloadFileTask.execute () 的第一个参数
```

```
url=new URL (urls[0]) ;
```

```
}catch (MalformedURLException e) {
```

```
e.printStackTrace () ; }
```

```
//检查输入的url参数是否为null
```

```
if (url !=null) {
```

```
try{
```

```
//打开连接
```

```
HttpURLConnection urlConnection=
```

```
(HttpURLConnection) url.openConnection () ;
```

```
//读取数据到InputStreamReader对象中
```

```
InputStreamReader isReader=new InputStreamReader (
```

```
//设置编码为UTF-8
```

```
urlConnection.getInputStream () , "UTF-8") ;
```

```
//实例化读buffer  
  
BufferedReader br=new BufferedReader (isReader) ;  
  
//实例化StringBuffer  
  
StringBuffer sb=new StringBuffer () ;  
  
String line=null ;  
  
while ( (line=br.readLine ()) !=null) {  
    sb.append (line) ;}  
  
//关闭数据流  
  
isReader.close () ;  
  
//关闭连接  
  
urlConnection.disconnect () ;  
  
//读取StringBuffer数据到字符串中  
  
XmlSourceStr=sb.toString () ;  
  
}catch (IOException e) {
```

```
e.printStackTrace () ; }
```

```
}
```

```
//以字符串方式返回
```

```
return XmlSourceStr ;
```

```
}
```

```
//对返回的结果进行操作
```

```
@Override
```

```
protected void onPostExecute (String result) {
```

```
super.onPostExecute (result) ;
```

```
//检查返回结果是否为null
```

```
//如果不为null则执行接下来的解析工作
```

```
if (result != null) {
```

```
try{
```

```
//创建SAXParserFactory实例
```

```
SAXParserFactory factory=SAXParserFactory.newInstance () ;
```

//新建SAXParser解析类

```
SAXParser parser=factory.newSAXParser () ;
```

//创建XMLReader读取类

```
XMLReader reader=parser.getXMLReader () ;
```

//为reader设置处理器

```
reader.setContentHandler (rssHandler) ;
```

//读取需要处理的数据

```
reader.parse (new InputSource (new StringReader (result) ) ) ;
```

```
}catch (ParserConfigurationException e) {
```

//指示一个严重的配置错误

```
e.printStackTrace () ;
```

```
}catch (SAXException e) {
```

//SAX错误或警告

```
e.printStackTrace () ;
```

```
}catch (IOException e) {
```

```
e.printStackTrace () ;
```

```
}
```

//判断rssList列表对象是否为空

```
if ( ! rssList.isEmpty () ) {
```

```
Iterator<RSS>it=rssList.iterator () ;
```

//填充数据

//判断rssList是否到达结尾

```
while (it.hasNext () ) {
```

//获取rssList下面的一项

```
RSS news= (RSS) it.next () ;
```

//定义HashMap

```
HashMap<String, Object>map=new HashMap<String, Object> () ;
```

//将title字段放入map中

```
map.put ("title", news.getTitle () ) ;
```

//将pubDate字段放入map中

```
map.put ("pubDate", news.getPubDate () ) ;
```

//将map放入mData数据中

```
mData.add (map) ;
```

//绑定到ListView界面

```
SimpleAdapter adapter=new SimpleAdapter (
```

```
RssReaderActivity.this,mData,
```

//定义ListView的布局文件

```
R.layout.rss_item,
```

//定义ListView中的数据项

```
new String[]{"title", "pubDate"},
```

//定义ListView中的数据项对应的控件id

```
new int[]{R.id.title,R.id.pubDate}) ;
```

```
//设置ListView适配器
```

```
setListAdapter (adapter) ; }
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

程序运行后，效果如图4-5所示。



图 4-5 简单RSS阅读器

4.3 Android Email编程

4.3.1 Android发送Email

Android在邮件客户端的设计中，有两种实现方法：

- 调用Android系统自带的邮件服务，这种方法简单易用，但发送邮件的账号必须是gmail账号。
- 采用javamail功能包进行设计，这种方法可以设置邮件服务器地址，不必局限于gmail邮箱，但较第一种方法复杂些。

1. 使用Android自带邮件系统设计

调用Android自带的邮件系统，实际上是使用Intent来调用相关的系统响应应用，其关键代码如下：

```
//建立Intent对象
```

```
Intent intent=new Intent () ;
```

```
//设置对象动作
```

```
intent.setAction (Intent.ACTION_SEND) ;
```

```
//设置对方邮件地址
```

```
intent.putExtra (Intent.EXTRA_EMAIL,new String[]  
{"abc@com.cn", "edf@com.cn"}) ;
```

//设置标题内容

```
intent.putExtra (Intent.EXTRA_SUBJECT, "test") ;
```

//设置邮件文本内容

```
intent.putExtra (Intent.EXTRA_TEXT, "test mail") ;
```

//启动一个新的ACTIVITY， "Sending mail....."是在启动这个
ACTIVITY的等待时间时所显示的文字

```
startActivity (Intent.createChooser (intent, "Sending mail.....") ) ;
```

需要注意的是，这种方法默认为gmail账号，需要在模拟器上配置gmail
邮箱，并输入自己的gmail账号和密码。

2.使用javamail设计

在Android里使用javamail需要依赖3个包：activation.jar、
additionnal.jar、 mail.jar，同时还要开启访问互联网的权限。该权限描述
如下：

```
<uses-permission android:name="android.permission.INTERNET">
</uses-permission>
```

对于javamail实现邮件的发送，还需要确保发方能连接到发方邮件服务器，一般邮件服务器都需要认证，只有认证通过才能连接，从而将邮件发送出去。以http://mail.126.com/免费邮箱为例，下面是其关键的代码：

```
//创建一个身份验证，即定义Authenticator的子类，并重载
getPasswordAuthentication () 方法
```

```
class PopupAuthenticator extends Authenticator{

    public PasswordAuthentication getPasswordAuthentication () {

        String username="from" ; //邮箱登录账号

        String pwd="123456" ; //登录密码

        //返回PasswordAuthentication对象

        return new PasswordAuthentication (username,pwd)  ;

    }

}
```

```
//实例化属性对象Properties  
  
Properties props=new Properties () ;  
  
//设置smtp的服务器地址是smtp.126.com  
  
props.put ("mail.smtp.host", "smtp.126.com") ;  
  
props.put ("mail.smtp.auth", "true") ; //设置smtp服务器要身份验证  
  
PopupAuthenticator auth=new PopupAuthenticator () ; //创建身份验证  
的实例  
  
//创建会话 (Session) , 用它管理连接  
  
Session session=Session.getInstance (props,auth) ;  
  
//实例化MimeMessage对象  
  
MimeMessage message=new MimeMessage (session) ;  
  
//设置发送方邮件地址  
  
Address addressFrom=new InternetAddress  
("from@126.com", "FROM") ;  
  
//设置收件方邮件地址
```

```
Address addressTo=new InternetAddress ("to@126.com", "TO") ;  
  
//收件人地址  
  
Address addressCopy=new InternetAddress  
("abc@gmail.com", "abc") ;  
  
//创建邮件内容  
  
message.setContent ("Hello", "text/plain") ;  
  
//或者使用message.setText ("Hello") ;  
  
message.setSubject ("Title") ;  
  
message.setFrom (addressFrom) ;  
  
message.addRecipient (Message.RecipientType.TO,addressTo) ;  
  
message.addRecipient (Message.RecipientType.CC,addressCopy) ;  
  
message.saveChanges () ;  
  
//发送邮件  
  
Transport transport=session.getTransport ("smtp") ; //创建连接  
  
transport.connect ("smtp.126.com", "from", "123456") ; //连接服务器
```

```
transport.send (message) ;//发送信息
```

```
transport.close () ;//关闭连接
```

4.3.2 实战案例：Android下Email的Base64加密

Base64是一种基于64个可打印字符来表示二进制数据的编码方法。由于 $2^6=64$ ，所以每6比特为一个单元，对应某个可打印字符。3字节为24比特，对应于4个Base64单元，即3字节需要用4个可打印字符来表示。它可用来作为电子邮件的传输编码。在Base64中的可打印字符包括字母A~Z、a~z、数字0~9共62个字符，另两个可打印字符在不同的系统中可能会不同。

Base64常用于处理文本数据的场合，用以表示、传输、存储一些二进制数据，包括MIME的Email、Email via MIME、在XML中存储复杂数据。

在MIME格式的电子邮件中，Base64可以用来将binary的字节序列数据编码成ASCII字符序列构成的文本。使用时，在传输编码方式中指定Base64。使用的字符包括英文大小写字母各26个，加上10个数字和加号“+”、斜杠“/”，一共64个字符，等号“=”用来作为后缀。

注意 完整的Base64定义可以参看RFC 1421和RFC 2045。

Android在API8之后加入Base64的相关类，为了和其他系统兼容，很多时候，需要使用Apache的包来处理Base64的加密和解密。下面的代码根据不同的参数对文件进行Base64的加密和解密处理。

//将文件转成Base64字符串

//path字符串参数为文件路径

```
public static String encodeBase64File (String path) throws Exception{
```

//用path参数创建新的文件

```
File file=new File (path) ;
```

//将文件读入数据流中

```
FileInputStream inputFile=new FileInputStream (file) ;
```

//新建一个和文件长度一样的buffer

```
byte[]buffer=new byte[ (int) file.length () ] ;
```

//从buffer中读取信息

```
inputFile.read (buffer) ;
```

//关闭数据流

```
        inputFile.close () ;  
  
        //调用Android的转换函数  
  
        return android.util.Base64.encodeToString (buffer,Base64.DEFAULT) ;  
  
    }  
  
    //将Base64字符解码保存文件  
  
    public static void decoderBase64File (String base64Code,String  
targetPath)  
throws Exception  
  
{  
  
    //调用Android的转换函数， 将结果保存到baseByte中  
  
    byte[]baseByte=android.util.Base64.decode  
    (base64Code,Base64.DEFAULT) ;  
  
    //实例化一个FileOutputStream对象  
  
    FileOutputStream out=new FileOutputStream (targetPath) ;  
  
    //将baseByte写到out中
```

```
out.write (baseByte) ;  
  
//关闭out  
  
out.close () ;  
  
}  
  
//将Base64字符保存到文本文件  
  
public static void toFile (String base64Code,String targetPath) throws  
Exception{  
  
byte[]buffer=base64Code.getBytes () ;  
  
FileOutputStream out=new FileOutputStream (targetPath) ;  
  
out.write (buffer) ;  
  
out.close () ;  
  
}
```

调用方法如下：

```
String base64Code=encodeBase64File ("D :\\"1.jpg") ;  
  
decoderBase64File (base64Code, "D :\\"2.jpg") ;
```

```
toFile (base64Code, "D:\\three.txt") ;
```

4.4 Android网络安全

4.4.1 Android网络安全简介

Android系统凭借其开源优势，受到市场欢迎，以Android为操作系统的智能手机占有率飞速增长。然而，随之而来的网络安全隐患也日益凸显，Android智能手机也成了手机病毒和恶意程序的“新宠”。

2011年6月，360手机云安全中心拦截到“X卧底”Android版变种，同批发现的还有一系列以监听、窃取用户隐私为目的的木马。瑞星公司也截获了一个Android系统的手机木马——“安卓遗产”木马。2011年7月，国内出现Android系统手机病毒，代号“安卓杀手”，用户手机一旦感染该毒，该病毒将拦截10086发送到手机上的任何短信，并会上传手机设备信息及手机SIM卡信息，最终会定制大量短信增值服务，扣取高额话费。360手机云安全中心还拦截到一批“安卓僵尸”最新变种木马。这批木马会伪装或篡改成Google update服务、俄罗斯方块、九宫格日记等60多款正常软件，以骗取用户下载安装，进而完成回传隐私、卸载特定软件以及安装新木马等一系列恶性操作。2011年8月，360手机云安全中心又截获了第一批利用Android最新GingerMaster漏洞发起攻击的“索马里海盗”木马，它可以像PC木马那样获得管理权限后完全控制用户的手机。

与Android系统架构相对应，Android手机主要涉及的安全问题如图4-6所示。

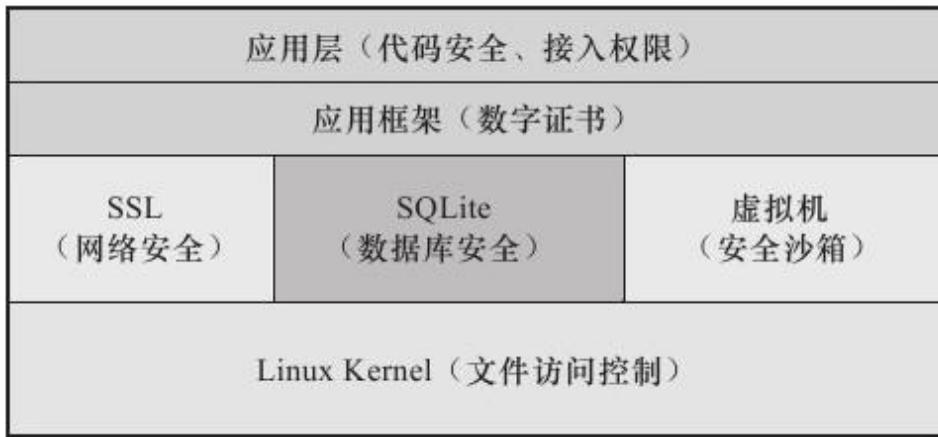


图 4-6 Android手机主要涉及的安全问题

应用层主要涉及代码安全和接入权限；应用框架层涉及数字证书问题；Android系统库与运行时一层主要涉及网络安全、数据库安全和虚拟机安全；Linux内核层涉及文件访问控制等。其中Android最重要的安全设计在于沙箱和权限。

1.代码安全

因为Android的源代码源于Java，因此存在代码反编译的隐患。为了提高代码级的安全性能，程序员应尽量减少客户端代码中的敏感内容，或者使用JNI技术，对敏感代码采用C/C++实现。

2.接入权限

权限主要用来对应用的操作增加限制，这样不但可以防止恶意应用进行非法操作进而造成敏感数据泄露和设备被非法控制，还可以防止恶意收费等问题。Android的接入权限包括如下几种。

- Normal权限：不会给用户带来实质性的伤害，如调整背光。
- Dangerous权限：可能会给用户带来潜在性的伤害，如读取电话簿、联网等，系统在安装应用时提示用户。
- signature权限：具有同一签名的应用才能访问。
- signatureOrSystem权限：主要由设备商使用，不推荐使用。

3.数字证书

Android系统不会安装运行任何一款未经数字签名的apk程序，所有的应用程序都必须有数字证书，无论是在模拟器上还是在实际的物理设备上。数字证书的最大用途是应用升级和设置应用间通信的权限。数字证书具有以下特点：

- 数字证书都是有有效期的，Android只是在应用程序安装的时候才会检查证书的有效期。如果程序已经安装在系统中，即使证书过期也不会影响程序的正常功能。

- Android程序使用的数字证书可以是自签名的，不需要一个权威的数字证书机构签名认证。
- 只有同一包名且采用同一数字证书的应用才被认为是同一个应用。

Android的数字证书有两种模式：调试模式（debug mode）和发布模式（release mode）。

在调试模式下，Android的开发工具会在每次编译时使用调试用的数字证书给程序签名，开发者无须关心，比如我们安装项目bin目录下的apk就是这种情况。但如果要正式发布一个Android应用程序，必须使用一个合适的私钥生成的数字证书来给程序签名。给程序签名有两种方式：一种方式是使用标准的Java工具Keytool and Jarsigner来生成数字证书，并给应用程序包签名；一种方式是使用ADT Export Wizard进行签名。

4.网络安全

网络安全内容非常丰富，各网络层都有安全问题及措施，这里不再展开介绍。需要重点探讨的是数据加密问题，本章下一节会就此做详述。通过加密算法和加密密钥将明文转变为密文进行通信，是对信息进行保护的一种最可靠的办法。

5.数据库安全

Android采用的SQLite目前采用明文存储数据，安全涉及加密、读写、搜索等。AndroidSQLite的权限有：android:permission、
android:readPermission、 android:writePermission。

6.虚拟机（安全沙箱）

Android通过使用沙箱技术来实现应用程序之间的分离和权限，以允许或拒绝一个应用程序访问设备的资源，比如文件、目录、网络、传感器和API等。在Android中，每个应用程序在安装时被分配一个用户ID，应用程序在设备上的存续期间内，用户ID保持不变。系统赋予应用程序里面所有的文件一组权限，以便只有分配用户ID的应用程序可以访问它们。

7.文件访问控制

Android在文件访问权限管理上应用了Linux的ACL（Access Control List）权限机制。

■分区层面：在系统运行时，最外层安全保护是由Linux系统提供的，其中system.img所在的分区是只读的，不允许用户写入，而data.img所在的分区是可读写的，用于存放用户数据。分区的用户权限在init.rc中定义。

■单独文件：单独文件访问权限控制分群组、用户、权限。权限分为可读、可写、可执行。

Android手机操作系统是以Linux为基础的开放源码操作系统，其病毒的猛增也源于此。黑客能够轻松获得系统的基本结构和源代码，从而破坏系统。任何事情都有两面性，正因其系统开发性，广大的软件从业者和使用者才会对代码进行评审、研究，因而漏洞更加容易被发现，出现问题后也能更快、更容易得到补救。

现实生活中，作为用户，需要提高安全防范意识，对来历不明、具有诱导性的彩信或邮件保持警惕，只安装正规商家开发的应用程序，及时安装反病毒工具并定时更新。作为运营商，需要提高反病毒能力，可以在移动通信网络上采取相关防范措施，在网络设备环节对传送的内容进行安全审核，并及时封堵攻击来源，把危害降到最低。此外，防护措施不应仅仅局限于手机终端，而应该是将移动网络、后台服务及个体终端结合起来，从全局角度提出完整的解决方案。

4.4.2 Android加密和解密

加密是通过加密算法和加密密钥将明文转变为密文的过程，解密是其逆过程。加密算法有很多种，一般可以分为对称加密（如DES、AES等）、非对称加密（如RSA等）和单向加密（如MD5等）3类算法。

- 在对称加密算法中，双方使用的密钥相同，要求解密方事先必须知道加密密钥。这类加密算法技术较为成熟，加密效率高。
- 在非对称加密算法中，收发双方使用不同密钥，发方使用公开密钥，收方使用私有密钥，保密性更高，但效率较低。
- 单向加密算法在加密过程中不需要使用密钥，输入明文后由系统直接经过加密算法处理成密文，密文无法解密。只有重新输入明文，并经过同样的加密算法处理，得到相同的密文并被系统重新识别后，才能真正解密。这种算法计算复杂，通常只在数据量有限的情形下使用，如广泛应用在计算机系统中的口令加密。

本小节加解密涉及DES、AES、MD5 3种算法。3种算法的具体加解密过程涉及内容较多，读者可以查阅现代密码学相关的书籍，限于篇幅，本小节对3种算法只做简要介绍，以便读者能读懂程序及程序结果。

- DES是数据加密标准（Data Encryption Standard）的简称，它是第一代公开的、完全说明实现细节的商用密码算法，并在1977年被美国政府正式采纳作为联邦标准。在1998年后，DES不再作为联邦加密算法。它在保护金融数据的安全，如自动取款机中，使用较多。DES算法经过16轮迭代，使用56比特长度密钥加密64比特长度（分组长度）的明文获得64比特长的密文。

■AES是高级加密标准（Advanced Encryption Standard）的简称，用于替代原先的DES，保护敏感信息。该算法由比利时密码学家Joan Daemaen和Vincent Rijmen共同提出，因此也称为Rijndael算法。AES算法的分组长度为128比特，其密钥长度分别为128比特、192比特和256比特。

■MD5全称是Message-digest Algorithm 5（信息-摘要算法），用于确保信息传输完整一致。在20世纪90年代初由Rivest设计，经MD2、MD3、MD4发展而来。MD5用的是散列函数（Hash函数），其典型应用是对一段信息产生信息摘要，从而实现数字签名、登录口令的认证、为文档生成“数字指纹”等。MD5算法的基本思想是以512位分组来处理输入的信息，且每一分组又被划分为16个32位子分组，经过了一系列的处理后，算法的输出由4个32位分组组成，将这4个32位分组级联后将生产一个128位的散列值。

1.DES加密和解密

Android中实现DES加密和解密的主要代码如下：

```
import javax.crypto.Cipher ;  
  
import javax.crypto.spec.IvParameterSpec ;  
  
import javax.crypto.spec.SecretKeySpec ;
```

```
public class DES{  
  
    //初始化向量， 随意填充  
  
    private static byte[]iv={1, 2, 3, 4, 5, 6, 7, 8} ;  
  
    //DES加密  
  
    //encryptString为原文  
  
    //encryptKey为密钥  
  
    //返回加密后的密文  
  
    public static String encryptDES (String encryptString,String encryptKey)  
throws Exception{  
  
    //实例化IvParameterSpec对象， 使用指定的初始化向量  
  
    IvParameterSpec zeroIv=new IvParameterSpec (iv) ;  
  
    //实例化SecretKeySpec类， 根据字节数组来构造SecretKey  
  
    SecretKeySpec key=new SecretKeySpec (encryptKey.getBytes  
() , "DES") ;  
  
    //创建密码器
```

```
Cipher cipher=Cipher.getInstance ("DES/CBC/PKCS5Padding") ;  
  
//用密匙初始化Cipher对象  
  
cipher.init (Cipher.ENCRYPT_MODE,key,zeroIv) ;  
  
//执行加密操作  
  
byte[]encryptedData=cipher.doFinal (encryptString.getBytes ()) ;  
  
//返回加密后的数据  
  
return Base64.encode (encryptedData) ;  
  
}  
  
//DES解密  
  
//decryptString为密文  
  
//decryptKey为密钥  
  
//返回解密后的原文  
  
public static String decryptDES (String decryptString,String decryptKey)  
throws Exception{  
  
//先使用Base64解密
```

```
byte[]byteMi=new Base64 ().decode (decryptString) ;  
  
//实例化IvParameterSpec对象，使用指定的初始化向量  
  
IvParameterSpec zeroIv=new IvParameterSpec (iv) ;  
  
//实例化SecretKeySpec类，根据字节数组来构造SecretKey  
  
SecretKeySpec key=new SecretKeySpec (decryptKey.getBytes  
() , "DES") ;  
  
//创建密码器  
  
Cipher cipher=Cipher.getInstance ("DES/CBC/PKCS5Padding") ;  
  
//用密匙初始化Cipher对象  
  
cipher.init (Cipher.DECRYPT_MODE,key,zeroIv) ;  
  
//获取解密的数据  
  
byte decryptedData[]=cipher.doFinal (byteMi) ;  
  
//解密数据转换为字符串输出  
  
return new String (decryptedData) ;  
  
}
```

```
}
```

下面代码实现的是调用上面DES加密和解密方法：

```
//指定密钥
```

```
String key="12345678" ;
```

```
//指定需要加密的明文
```

```
String text="ABCDEFGHIJKLMNPQRSTUVWXYZ" ;
```

```
try{
```

```
//调用DES加密方法
```

```
String encryptResult=DES.encryptDES (text,key) ;
```

```
//调用DES解密方法
```

```
String decryptResult=DES.decryptDES (encryptResult,key) ;
```

```
//返回DES加密的结果
```

```
Log.d ("DES加密结果为：" , encryptResult) ;
```

```
//返回DES解密的结果
```

```
Log.d ("DES解密结果为：" , decryptResult) ;  
}  
catch (Exception e) {  
e.printStackTrace () ;  
}
```

运行结果如下：

加密结果为：PHRVRP7SYUNVW/r6v65aHpp1aCHaFEelxufJ+oYtNgk=

解密结果为：ABCDEFGHIJKLMNOPQRSTUVWXYZ

2.AES加密和解密

Android中AES加密和解密主要代码如下：

```
import java.security.SecureRandom ;  
  
import javax.crypto.Cipher ;  
  
import javax.crypto.KeyGenerator ;  
  
import javax.crypto.SecretKey ;  
  
import javax.crypto.spec.SecretKeySpec ;
```

```
//AES加密和解密的方法
```

```
public class AES{
```

```
//AES加密
```

```
//cleartext为需要加密的内容
```

```
//seed为密钥
```

```
public static String encrypt (String seed,String cleartext) throws  
Exception{
```

```
//对密钥进行编码
```

```
byte[]rawKey=getRawKey (seed.getBytes () ) ;
```

```
//加密数据
```

```
byte[]result=encrypt (rawKey,cleartext.getBytes () ) ;
```

```
//将十进制数转换为十六进制数
```

```
return toHex (result) ;
```

```
}
```

```
//AES解密
```

//encrypted为需要解密的内容

//seed为密钥

```
public static String decrypt (String seed,String encrypted) throws  
Exception{
```

//对密钥进行编码

```
byte[]rawKey=getRawKey (seed.getBytes () ) ;
```

```
byte[]enc=toByte (encrypted) ;
```

```
byte[]result=decrypt (rawKey,enc) ;
```

```
return new String (result) ;
```

```
}
```

//对密钥进行编码

```
private static byte[]getRawKey (byte[]seed) throws Exception{
```

//获取密匙生成器

```
KeyGenerator kgen=KeyGenerator.getInstance ("AES") ;
```

```
SecureRandom sr=SecureRandom.getInstance ("SHA1PRNG") ;
```

```
sr.setSeed (seed) ;  
  
//生成128位的AES密码生成器  
  
kgen.init (128, sr) ;  
  
//生成密匙  
  
SecretKey skey=kgen.generateKey () ;  
  
//编码格式  
  
byte[]raw=skey.getEncoded () ;  
  
return raw ;  
  
}  
  
//加密  
  
private static byte[]encrypt (byte[]raw,byte[]clear) throws Exception{  
  
//生成一组扩展密钥，并放入一个数组之中  
  
SecretKeySpec skeySpec=new SecretKeySpec (raw, "AES") ;  
  
Cipher cipher=Cipher.getInstance ("AES") ;
```

```
//用ENCRYPT_MODE模式，用skeySpec密码组，生成AES加密方法  
cipher.init (Cipher.ENCRYPT_MODE,skeySpec) ;  
  
//得到加密数据  
  
byte[]encrypted=cipher.doFinal (clear) ;  
  
return encrypted ;  
  
}  
  
//解密  
  
private static byte[]decrypt (byte[]raw,byte[]encrypted) throws Exception{  
  
//生成一组扩展密钥，并放入一个数组之中  
  
SecretKeySpec skeySpec=new SecretKeySpec (raw, "AES") ;  
  
Cipher cipher=Cipher.getInstance ("AES") ;  
  
//用DECRYPT_MODE模式，用skeySpec密码组，生成AES解密方法  
cipher.init (Cipher.DECRYPT_MODE,skeySpec) ;  
  
//得到解密数据
```

```
byte[] decrypted=cipher.doFinal (encrypted) ;  
  
return decrypted ;  
  
}
```

//将十进制数转换为十六进制

```
public static String toHex (String txt) {  
  
return toHex (txt.getBytes ()) ;  
  
}
```

//将十六进制字符串转换为十进制字符串

```
public static String fromHex (String hex) {  
  
return new String (toByte (hex)) ;  
  
}
```

//将十六进制字符串转换为十进制字节数组

```
public static byte[] toByte (String hexString) {  
  
int len=hexString.length () /2 ;
```

```
byte[]result=new byte[len] ;  
  
for (int i=0 ; i<len ; i++)  
  
    result[i]=Integer.valueOf (hexString.  
  
        substring (2*i, 2*i+2) , 16) .byteValue () ;  
  
return result ;  
  
}
```

//把一个十进制字节数组转换成十六进制

```
public static String toHex (byte[]buf) {  
  
if (buf==null)  
  
    return"" ;  
  
StringBuffer result=new StringBuffer (2*buf.length) ;  
  
for (int i=0 ; i<buf.length ; i++) {  
  
    appendHex (result,buf[i]) ;  
  
}  
}
```

```
        return result.toString () ;  
    }  
  
    private final static String HEX="0123456789ABCDEF" ;  
  
    private static void appendHex (StringBuffer sb,byte b) {  
  
        sb.append (HEX.charAt ( (b>>4) & 0x0f) ) .append (HEX.charAt  
        (b & 0x0f) ) ;  
  
    }  
  
}
```

调用方法如下：

```
//设置加密密钥  
  
String masterPassword="Android" ;  
  
//设置原文  
  
String originalText="ABCDEFGHIJKLMNPQRSTUVWXYZ" ;  
  
try{  
  
    //调用AES加密方法
```

```
String encryptingCode=AES.encrypt (masterPassword,originalText) ;  
  
Log.d ("加密结果为："， encryptingCode) ;  
  
//调用AES解密方法  
  
String decryptingCode=AES.decrypt  
(masterPassword,encryptingCode) ;  
  
Log.d ("解密结果为："， decryptingCode) ;  
  
}catch (Exception e) {  
  
e.printStackTrace () ;  
  
}  

```

最后的结果如下：

加密结果为：

374221575F727FA0B57CD5B9E2F83239938FAB4E07CE4657F86B803B8
E2945FC

解密结果为：ABCDEFGHIJKLMNOPQRSTUVWXYZ

3.MD5加密

MD5加密的方法如下：

```
//MD5加密
```

```
private static String toMd5 (byte[]bytes) {
```

```
try{
```

```
//实例化一个指定摘要算法为MD5的MessageDigest对象
```

```
MessageDigest algorithm=MessageDigest.getInstance ("MD5") ;
```

```
//重置摘要以供再次使用
```

```
algorithm.reset () ;
```

```
//使用bytes更新摘要
```

```
algorithm.update (bytes) ;
```

```
//使用指定的byte数组对摘要进行最后更新，然后完成摘要计算
```

```
return toHexString (algorithm.digest () , "") ;
```

```
}catch (NoSuchAlgorithmException e) {
```

```
}
```

```
}
```

```
//将字符串中的每个字符转换为十六进制
```

```
private static String toHexString (byte[]bytes,String separator) {
```

```
    StringBuilder hexString=new StringBuilder () ;
```

```
    for (byte b:bytes) {
```

```
        String hex=Integer.toHexString (0xFF & b) ;
```

```
        if (hex.length () ==1) {
```

```
            hexString.append ('0') ;
```

```
}
```

```
        hexString.append (hex) .append (separator) ;
```

```
}
```

```
    return hexString.toString () ;
```

```
}
```

调用方法如下：

```
String md5=MD5 ("Android".getBytes () ) ;
```

调用结果如下：

e84e30b9390cdb64db6db2c9ab87846d

4.4.3 实战案例：Android应用添加签名

Android应用程序在发布的时候，进行签名打包可以有效地提高其安全性。下面介绍利用Eclipse对Android应用进行签名打包的方法。

步骤1 在项目名称上右击，在弹出的快捷菜单中选择“Android Tools”，在随后弹出的对话框中单击“Export Signed Application Package.....”。



图 4-7 选择需要添加签名的项目

步骤2 如图4-7所示，单击“Browse.....”按钮，选择需要添加签名的项目，然后单击“Next”按钮。

步骤3 再单击“Location”文本框后边的“Browse.....”按钮，选择需要使用的keystore，并在“Password”文本框中输入密码，如图4-8所示。



图 4-8 选择keystore并输入密码

如果希望新建keystore，则需要选中“Create new keystore”单选框，选择其保存的目录，输入密码并确认密码，如图4-9所示。

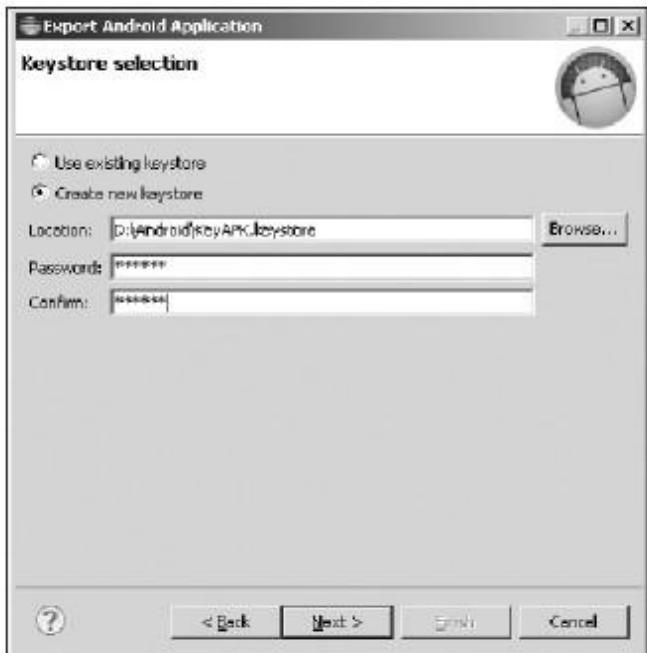


图 4-9 新建keystore并设置密码

都设置好以后，单击“Next”按钮。

步骤4 填写Key的信息，如图4-10所示。Key信息各字段的含义如表4-6所示。全都填写好以后单击“Next”按钮。



图 4-10 填写Key的信息

表 4-6 Key 信息中各字段及含义

字 段	含 义
Alias	私钥的别名
Password	密码
Confirm	确认密码
Validity(years)	设置有效期（推荐 25 年）
First and Last Name	应用开发者姓名
Organizational Unit	开发者组织单位名称
Organization	开发者组织名称
City or Locality	开发者所在的城市或地区
State or Province	开发者所在的州或省
Country Code(XX)	开发者的国家代码

步骤5 选择保存的目录，选好以后单击“Finish”按钮，完成签名的添加，如图4-11所示。



图 4-11 选择保存的目录

4.5 OAuth认证

4.5.1 OAuth简介

1. OAuth认证授权的特点

OAuth是一个开放标准，允许用户让第三方应用访问该用户在某一网站上存储的私密资源（如照片、视频、联系人列表），而无须将用户名和密码提供给第三方应用。

OAuth协议为用户资源的授权提供了一个安全的、开放而又简易的标准。基于OAuth的认证授权具有以下特点。

(1) 安全

OAuth允许用户提供一个令牌（Token），而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的网站（例如视频编辑网站）在特定的时段（例如接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。这样，OAuth允许用户授权第三方网站访问他们存储在另外的服务提供者上的信息，而不需要分享他们的访问许可或他们数据的所有内容。

(2) 开放

OAuth是开放的，任何第三方都可以使用OAuth认证服务，任何服务提供商都可以实现自身的OAuth认证服务。

(3) 简易

不管是客户端还是服务提供方，都很容易理解和使用OAuth认证服务。

2.OAuth认证授权的过程

在认证和授权的过程中涉及以下三方。

- 服务提供方（Service Provider）：用户使用服务提供方来存储受保护的资源（Protected Resources），如照片、视频、联系人列表。
- 用户（User）：存放在服务提供方的受保护的资源的拥有者。
- 客户端（Consumer）：要访问服务提供方资源的第三方应用。

使用OAuth进行认证和授权的过程如图4-12所示。

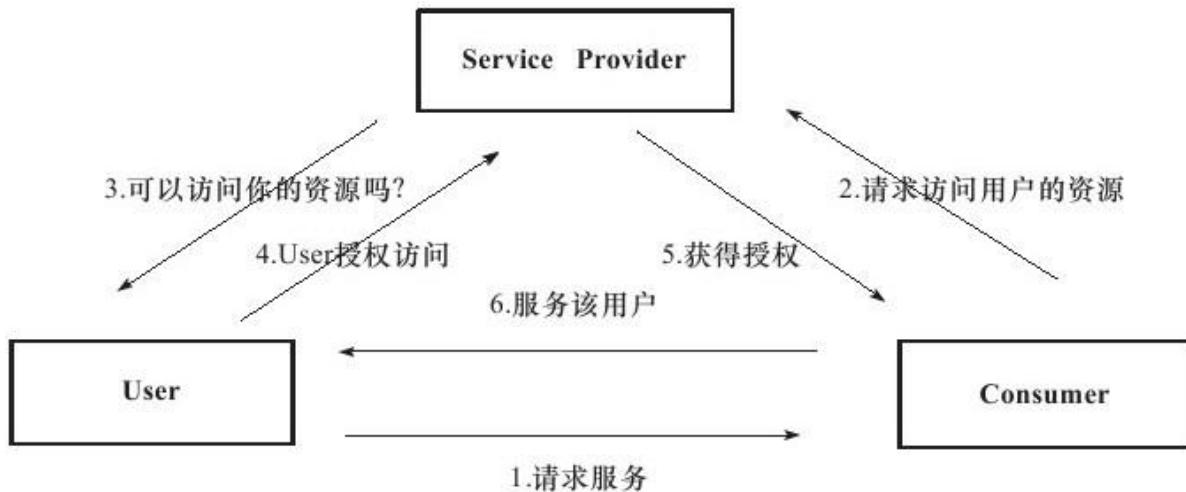


图 4-12 OAuth认证过程

当用户需要客户端为其提供服务时，该服务涉及需要从服务提供方那里获取该用户的保护资源。只有在用户授权后，客户端才可以获取用户的资源。

OAuth协议的一些基本术语定义如下。

- Consumer Key：客户端对于服务提供方的身份唯一标识。
- Consumer Secret：用来确认客户端对于Consumer Key的拥有关系。
- Request Token：获得用户授权的请求令牌，用于交换Access Token。
- Access Token：用于获得用户在服务提供方的受保护资源。
- Token Secret：用来确认客户端对于令牌（Request Token和Access Token）的拥有关系。

OAuth认证授权过程可以简单概括为四步：

步骤1 通过Request Token URL， 获取未授权的Request Token（Request Token URL是获取未授权的Request Token服务的地址）。

步骤2 通过User Authorization URL获取用户授权的Request Token（User Authorization URL是获取用户授权的Request Token服务的地址）。

步骤3 通过Access Token URL用授权的Request Token换取Access Token（Access Token URL是用授权的Request Token换取Access Token服务的地址）。

步骤4 拿到Access Token后，就有权访问用户授权的资源了。

3. 获取OAuth授权

各大公司都推出了自己的开放开发接口，不同程度地对OAuth操作进行了封装和简化。本节以腾讯微博的OAuth接口为例，介绍获取OAuth授权的方法。

首先需要注册成为开发者，提交将要创建的应用的信息，如图4-13所示。具体可以参照网址<http://open.t.qq.com/developer/bedever>。获得App Key和App Secret后，就可以利用腾讯微博开放平台提供的各种API接口开发应用了。



图 4-13 向腾讯提交应用信息

4.5.2 实战案例：使用OAuth接口

获得腾讯App key和App Sercet之后，key和Sercet的使用方法如下：

//认证成功后浏览器会被重定向到这个URL中，必须与注册时填写的一致

```
private String redirectUri="http://www.tencent.com/zh-cn/index.shtml" ;
```

```
//换为用户为自己的应用申请到的APP KEY，笔者申请的时候长度为9位
```

```
private String clientId="123456789" ;
```

//换为用户为自己的应用申请到的APP SECRET，笔者申请的时候长度为32位

```
private String clientSecret="01234567890123456789012345678901" ;
```

案例实现OAuth Version 2.a使用WebView辅助进行ImplicitGrant方式授权。

1) 实例化OAuth授权信息记录实体，代码如下：

```
Private OAuthV2 oAuth ;
```

在授权类中设置成员变量，需将redirectUri、AppKey、AppSecret修改为用户的应用信息。

```
oAuth=new OAuthV2 (redirectUri) ;
```

```
oAuth.setClientId (AppKey) ;
```

```
oAuth.setClientSecret (AppSecret) ;
```

2) 创建Intent，使用WebView让用户授权，代码如下：

```
//请将OAuthV2Activity改为所在类的类名
```

```
Intent intent=new Intent
```

```
(OAuthV2Activity.this,OAuthV2AuthorizeWebView.class) ;
```

```
//将数据放到Intent参数中  
  
intent.putExtra ("oauth", oAuth) ;  
  
//调用相关的Activity  
  
startActivityForResult (intent,myRequestCode) ;  
  
//请设置合适的requestCode
```

重写接回调信息的方法，代码如下：

```
if (requestCode==myRequestCode) {//对应之前设置的myRequestCode  
  
if (resultCode==OAuthV2AuthorizeWebView.RESULT_CODE) {  
  
//取得返回的OAuthV2类实例oAuth  
  
oAuth= (OAuthV2) data.getExtras () .getSerializable ("oauth") ;  
  
}  
  
}
```

完整的代码如下：

```
//OAuth Version 2.a ImplicitGrant方式授权及调用API
```

```
public class TestsdkActivity extends Activity{  
  
    //定义OAuthV2对象  
  
    private OAuthV2 oAuth ;  
  
    public void onCreate (Bundle savedInstanceState) {  
  
        super.onCreate (savedInstanceState) ;  
  
        //添加布局文件  
  
        setContentView (R.layout.main) ;  
  
        //实例化OAuthV2对象  
  
        oAuth=new OAuthV2 ("http://www.tencent.com/zh-cn/index.shtml") ;  
  
        //该处ID为示例内容，实际应用中请替换为申请到的ID  
  
        oAuth.setClientId ("123456789") ;  
  
        //添加客户端密码，该处的密码为示例内容，实际应用中请替换为申请  
        //到的密码  
  
        oAuth.setClientSecret ("01234567890123456789012345678901") ;  
  
        //新建一个Intent对象
```

```
Intent intent=new Intent (TestsdkActivity.this,  
OAuthV2AuthorizeWebView.class) ;  
  
//保存参数  
  
intent.putExtra ("oauth", oAuth) ;  
  
//调用onActivityResult函数  
  
startActivityForResult (intent, 1) ;  
  
}  
  
//定义回调函数  
  
protected void onActivityResult (int requestCode,int resultCode,Intent  
data) {  
  
//检测对应的调用码  
  
if (requestCode==1) {  
  
if (resultCode==OAuthV2AuthorizeWebView.RESULT_CODE) {  
  
oAuth= (OAuthV2) data.getExtras () .getSerializable ("oauth") ;  
  
//调用API获取用户信息
```

```
UserAPI userAPI=new UserAPI  
        (OAuthConstants.OAUTH_VERSION_2_A) ;  
  
try{  
  
    String response=userAPI.info (oAuth, "json") ; //获取用户信息  
  
    (TextView) findViewById (R.id.textView) ) .setText  
    (response+"\n") ;  
  
}catch (Exception e) {  
  
    e.printStackTrace () ;  
  
}  
  
//关闭连接  
  
    userAPI.shutdownConnection () ;  
  
}  
  
}  
  
}  
  
}
```

程序运行效果如图4-14所示。



图 4-14 使用腾讯微博的OAuth接口实例

另外还可以使用TAPI接口进行微博的相关操作，其示例代码如下：

TAPI tAPI ; //TAPI封装了微博相关操作

```
tAPI=new TAPI (OAuthConstants.OAUTH_VERSION_2_A) ;
```

```
try{
```

```
//add接口用于发表普通微博消息
```

```
response=tAPI.add (oAuthV2, "json", "Android客户端文字微博  
2", "127.0.0.1") ;
```

```
//添加换行符  
textResponse.append (response+"\n") ;  
  
}catch (Exception e) {  
  
e.printStackTrace () ;  
  
}  
  
//关闭连接  
tAPI.shutdownConnection () ;
```

4.6 小结

本章首先介绍了Android中解析XML文件的3种方式，即DOM、SAX、PULL。DOM解析XML文件时，会将XML文件的所有内容以文档树方式存放在内存中，内存的消耗比较大，适合XML文件较小的情况；后两种方式是基于事件驱动的，边加载边解析。案例中给出了在Android中创建XML的方法。

接着介绍了RSS，并给出了Android中订阅RSS的实例。其后涉及Android Email编程，在Android邮件客户端的设计中，第一种方法是调用Android系统自带的邮件服务，这种方法虽然简单，但发送邮件的账号必须是gmail账号；第二种方法是采用javamail功能包进行设计，这样不必局限于gmail邮箱，但相当复杂。

后一节文中概述了Android网络安全问题，并重点阐述了Android的加解密，涉及MD5、AES、DES 3种方法，案例中还涉及给Android应用添加签名。文中最后介绍了Android OAuth认证，这一部分的案例介绍了获取腾讯微博的OAuth授权和使用的方法。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第5章 Android网络模块编程

本章涉及的内容包括Android的地图和位置定位编程、USB编程、Wi-Fi编程、蓝牙编程、NFC编程等。除了NFC编程一节外，其他章节均有实用的实战案例，以方便读者掌握相关知识点。

5.1 Android地图和定位

Android SDK提供了相关的API，可帮助应用程序开发人员显示和操作地图，获取实时设备位置信息。地图API和基于位置的API是放在彼此独立的两个包中的，地图包是com.google.android.maps，位置包是android.location。这些API通过Internet从Google服务器调用服务。本节介绍地图和基于位置的服务。

5.1.1 获取map-api密钥

地图包包含在屏幕上显示地图、处理用户与地图的交互（如缩放、平移等）、在地图上显示自定义数据等所需的一切内容。前提是必须安装Google Map API，之后Android中的地图可归结为使用MapView UI控件和MapActivity类。在使用MapView之前，需要从Google获得一个map-api密钥。map-api密钥供Android用于与Google Maps服务交互，以获取地图数据。

关于map-api密钥需要明确的是密钥的证书涉及调试证书和发布证书。我们在4.6节讲过，调试证书用于在模拟器中进行开发，在调试模式下，Android的开发工具会在每次编译时自动使用调试用的数字证书给程序签名。发布证书用于设备的生产。这里获取map-api密钥，要先找

到调试证书，继而获得与调试证书关联的MD5指纹，然后将它输入到Google网站才能生成关联的map-api密钥。

要找到调试证书先要找到调试模式（debug mode）下的keystore，在不同的操作系统中，keystore的位置是不一样的。从Eclipse的Preferences菜单选择Android->Build，就可以发现Default debug keystore字段中显示了keystore的位置。keystore一般默认位置如下：

C:\Documents and Settings\.android\debug.keystore

在模拟器里面，可以在Eclipse下window->Preferences->Android->Build里查看Default debug keystore的位置，如图5-1所示。

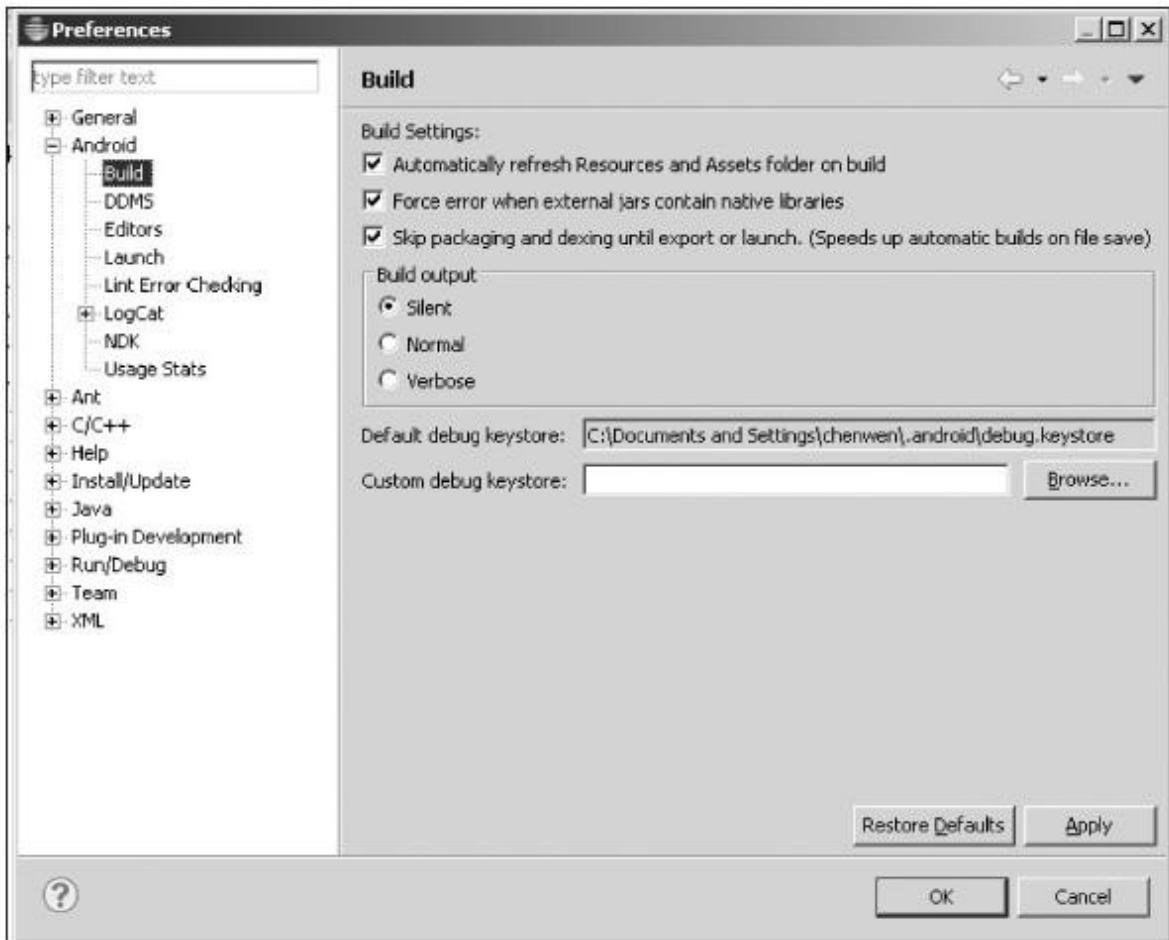


图 5-1 查看调试模式下keystore的位置

接着，使用keytool工具产生MD5指纹，在命令行下输入如下语句：

```
keytool -list -alias androiddebugkey -keystore debug.keystore -storepass  
android-keypass android
```

如果生成的证书指纹的是SHA1的，可以通过如下命令得到MD5的证书指纹：

```
C:\Program Files\Java\jdk1.7.0_07\bin>keytool -list -keystore "C:\\Documents and Settings\\chenwen\\.android\\debug.keystore" -v
```

输入密钥库口令：

```
*****WARNING WARNING
```

```
WARNING*****
```

存储在您密钥库中的信息的完整性

*尚未经过验证！为了验证其完整性， *

*必须提供密钥库口令。 *

```
*****WARNING WARNING
```

```
WARNING*****
```

密钥库类型：JKS

密钥库提供方：SUN

您的密钥库包含1个条目

别名：androiddebugkey

创建日期：2012-9-21

条目类型：PrivateKeyEntry

证书链长度：1

证书[1]：

所有者：CN=Android Debug,O=Android,C=US

发布者：CN=Android Debug,O=Android,C=US

序列号：3edaf99a

有效期开始日期：Fri Sep 21 08:14:16 CST 2012， 截止日期：Sun Sep 14
08:14:16 CST 2042证书指纹：

MD5:51:1D:67:50:7A:9A:60:2B:D5:07:25:9D:39:CF:53:64

SHA1:0C:2A:1B:76:19:16:53:E3:0D:09:C7:87:55:5F:12:B9:3D:67:F4:A7

SHA256:D8:24:9E:A8:45:5E:41:AF:F9:73:7F:3D:9B:07:9E：

AF:02:AA:88:39:77:57:D5:A3:7D:E2:16:48:16:74:3C:09

签名算法名称：SHA256withRSA

版本：3

Windows 7系统中情况如下：

C :\Users\chenwen>keytool -list -keystore

"C :\Users\chenwen\.android\debug.keystore"

输入keystore密码：

*****警告警告警告*****

保存在您的keystore中数据的完整性

*尚未被验证！为了验证其完整性， *

*您必须提供您keystore的密码。 *

*****警告警告警告*****

Keystore类型：JKS

Keystore提供者：SUN

您的keystore包含1输入

androiddebugkey, 2011-8-22, PrivateKeyEntry,

认证指纹 (MD5) :

CC:FA:D3:24:63:A5:A6:FA:96:E8:E9:B2:9B:57:D3:22

这样就可以将证书的MD5指纹粘贴到以下Google网站的MD5 fingerprint字段中：

<http://code.google.com/android/maps-api-signup.html>或者

<https://developers.google.com/android/maps-api-signup?hl=zh-CN>

网页内容如图5-2所示。

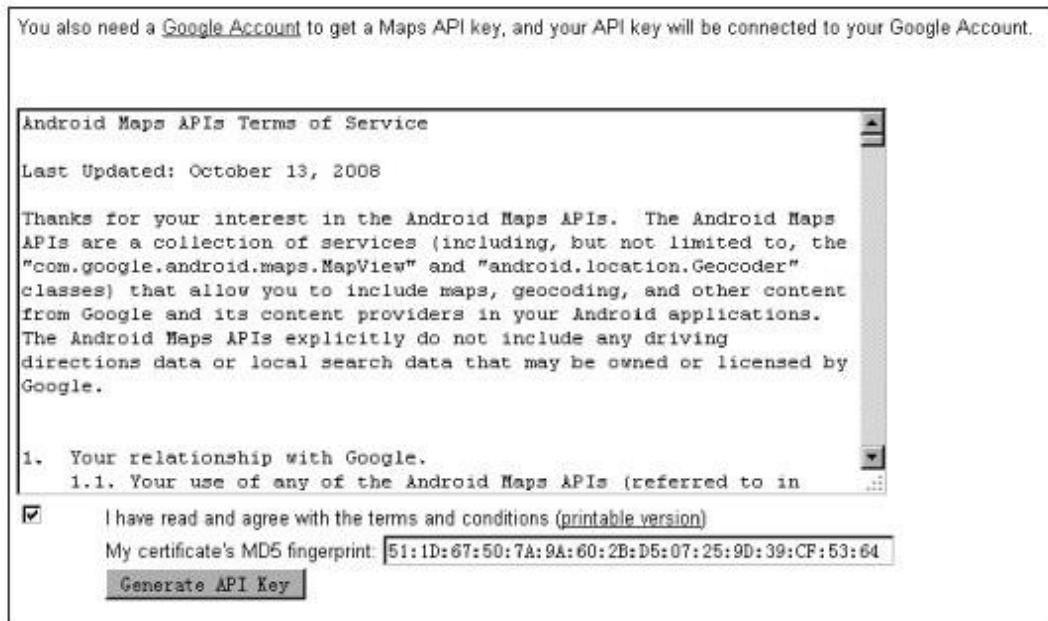


图 5-2 由MD5指纹产生map-api密钥

仔细阅读服务条款。如果同意这些条款，那么单击Generate API Key按钮，就会从Google Maps服务获得相应的map-api密钥了，获取的密钥信息如下：

```
<com.google.android.maps.MapView  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    <!--apiKey的值为获取到的map-api的值-->  
    android:apiKey="0pu0oVIVGQ6yyQjRIADovBf4RPR43gHyfrMzo9g"  
/>
```

map-api密钥会立即被激活，所以这时就可以使用它从Google获得地图数据了。

5.1.2 获取位置

android.location包提供了一些工具来实现基于位置的服务。这个包有两个重要部分：Geocoder类和LocationManager服务。

android.location.Geocoder类可实现地址（或位置）与经纬度的相互转换（称为地理编码）。该类提供了以下两个重要方法：

- getFromLocation方法。该方法通过所给参数经纬度可以获得地方的名称和物理地址，或者该位置的通用的名称。其返回的是一个地址列

表，而不是一个地址，通过参数maxResults可以限制结果集。该方法的声明如下：

```
getFromLocation (double latitude,double longitude,int maxResults)
```

■getFromLocationName方法。它提供了两种重载的调用形式，一种如下：

```
getFromLocationName (String locationName,int maxResults,double lowerLeftLatitude,double lowerLeftLongitude,double upperRightLatitude,double upperRightLongitude)
```

上述调用形式可以获得地方名称的地址描述，位置可以是地址或者众所周知的名称，比如“白宫”。返回的地址信息可以非常多，包括位置的常用名、街道、城市、州、邮编、国家等，程序中应根据需要读取返回值。

另一种调用形式可以获得地方名称的地址描述，形式如下：

```
getFromLocationName (String locationName,int maxResults)
```

对于地理编码，还要强调以下几点：

■返回的地址不一定总是准确的地址。返回的地址列表取决于输入信息的准确度，所以应尽量提供准确的位置信息。

- 尽量将maxResults设置为1~5的值。
- 由于操作很耗时且网络连接未必尽如人意，所以应尽量考虑在不同线程中执行地理编码。

LocationManager服务是android.location包提供的一项重要服务。该服务提供获取设备地理位置的机制，同时在设备进入指定地理位置时可提供通知用户（通过Intent）的功能。

以下代码段给出了LocationManager服务的简单使用：

```
//LocationManager获取位置信息

public class LocationManagerExampleActivity extends Activity

{

protected void onCreate (Bundle savedInstanceState)

{

super.onCreate (savedInstanceState) ;

//获得LocationManager服务

LocationManager locMgr= (LocationManager)
```

```
this.getSystemService (Context.LOCATION_SERVICE) ;  
  
//获取位置信息  
  
Location loc=locMgr.  
  
//getLastKownLocation () 方法获取Loaction对象  
  
getLastKownLocation (LocationManager.GPS_PROVIDER) ;  
  
//获得所有的LocationProvider列表  
  
List<String>providerList=locMgr.getAllProviders () ;  
  
}  
  
}
```

要使用LocationManager服务，必须首先获得它的引用。而
LocationManager服务是一项系统级服务，系统级服务是使用服务名称
从上下文获得的服务，因此不能直接实例化。可以通过
android.app.Activity类提供的getSystemService () 方法获取系统级服
务。在参数中传递服务名称Context.LOCATION_SERVICE。

LocationManager服务使用位置提供程序来提供详细的地理位置信息。
目前，有以下3种类型的位置提供程序：

■GPS提供程序使用全球定位系统获取位置信息。如上述代码所示，就是使用全球定位系统获取的位置信息。参数为 LocationManager.GPS_PROVIDER。

■网络提供程序使用手机信号塔或Wi-Fi网络获取位置信息。此时参数应为LocationManager.NETWORK_PROVIDER。

■被动提供程序类似于位置更新探测器，它向应用程序传递其他应用程序请求的位置更新，应用程序无须专门请求任何位置更新。当然，如果没有其他应用程序请求位置更新，其就不会得到任何更新。此时参数应为LocationManager.PASSIVE_PROVIDER。

5.1.3 实战案例：利用MapView显示地图

com.google.android.maps.MapView控件可以显示地图。既可以通过XML布局，也可以通过代码实例化此控件，但使用它的活动必须扩展MapActivity。MapView和MapActivity类必须协同工作。MapActivity处理加载地图的多线程请求、执行缓存等。

MapView实例化示例如下，示例演示了通过XML布局创建MapView控件。

```
<LinearLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:orientation="vertical" android:layout_width="fill_parent"

    android:layout_height="fill_parent">

    <!--加入MapView控件-->

    <!--android:apiKey对应的值为上文申请到的值-->

    <com.google.android.maps.MapView android:layout_width="fill_parent"

        android:layout_height="fill_parent" android:enabled="true"

        android:clickable="true" android:apiKey="myAPIKey"/>

</LinearLayout>
```

由于地图的基础数据来自GoogleMaps，所以应用程序需要有访问Internet的权限。

```
<!--允许访问网络的权限-->

<uses-permission android:name="android.Permission.INTERNET">

    <!--通过Wi-Fi或移动基站的方式获取用户粗略的经纬度信息，定位精度大概误差在30~1500米-->

    <uses-permission android:name=
```

```
"android.permission.ACCESS_COARSE_LOCATION"/>

<!--通过GPS芯片接收卫星的定位信息，定位精度达10米以内-->

<uses-permission
    android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

- android.permission.INTERNET：允许应用访问网络的权限。
- android.permission.ACCESS_COARSE_LOCATION：允许应用通过网络信息获取当前地理位置信息的权限。
- android.permission.ACCESS_FINE_LOCATION：允许应用通过GPS获取当前地理位置信息的权限，较网络获得的地理位置信息更为精确。

此外，需要添加对Google Map库的引用。

```
<uses-library android:name="com.google.android.maps"/>
```

上例使用XML布局实例化MapView，需要设置android:apiKey属性。如果编程创建MapView，则必须将map-api密钥传递给MapView构造函数。

下面的代码为布局文件：

```
<RelativeLayout  
    xmlns:android="http://schemas.android.com/apk/res/android"  
    xmlns:tools="http://schemas.android.com/tools"  
    android:layout_width="match_parent"  
    android:layout_height="match_parent">  
  
<!--加入MapView控件-->  
  
<!--android:apiKey对应的值为上文申请到的值-->  
  
<com.google.android.maps.MapView  
    android:id="@+id/mapview"  
    android:layout_width="fill_parent"  
    android:layout_height="fill_parent"  
    android:apiKey="0pu0oVIVGQ6yyQjRIADovBf4RPR43gHyfrMzo9g"  
/>  
  
</RelativeLayout>
```

下面的代码是配置文件：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
package="org.chenwen.mapshow" android:versionCode="1" android:version  
Name="1.0">  
  
<uses-sdk android:minSdkVersion="10" android:targetSdkVersion="16"/>  
  
<!--允许应用访问网络-->  
  
<uses-permission android:name="android.permission.INTERNET"/>  
  
<!--允许应用通过网络信息获取当前地理位置信息-->  
  
<uses-permission  
    android:name="android.permission.ACCESS_COARSE_LOCATION"/>  
  
<!--允许应用通过GPS获取当前地理位置信息的权限，较网络获得的  
    地理位置信息更为精确-->  
  
<uses-permission  
    android:name="android.permission.ACCESS_FINE_LOCATION"/>  
  
<application  
    android:icon="@drawable/ic_launcher" android:label="@string/app_name"  
    android:theme="@style/AppTheme" android:allowBackup="true">
```

```
<!--添加对于Google Map库的引用-->

<uses-library android:name="com.google.android.maps"/>

<!--添加启动的Activity-->

<activity
    android:name="org.chenwen.mapshow.MapShowActivity"
    android:label="@string/title_activity_map_show">

    <intent-filter>

        <action android:name="android.intent.action.MAIN"/>

        <category android:name="android.intent.category.LAUNCHER"/>

    </intent-filter>

</activity>

</application>

<!--添加对于Google Map库的引用-->

<uses-library android:name="com.google.android.maps"/>
```

```
</manifest>
```

MapShowActivity用于显示地图的内容，其代码如下：

```
import com.google.android.maps.GeoPoint ;  
  
import com.google.android.maps.MapActivity ;  
  
import com.google.android.maps.MapController ;  
  
import com.google.android.maps.MapView ;  
  
import android.os.Bundle ;  
  
import android.view.KeyEvent ;  
  
import android.view.Menu ;  
  
//显示Google地图  
  
public class MapShowActivity extends MapActivity{  
  
    //新建地图显示控制相关变量  
  
    private MapView map=null ;  
  
    private MapController mapCon ;
```

```
//菜单项

final private int menuMode=Menu.FIRST ;

final private int menuExit=Menu.FIRST+1 ;

@Override

public void onCreate (Bundle savedInstanceState) {

super.onCreate (savedInstanceState) ;

//加载布局文件

setContentView (R.layout.activity_map_show) ;

//获取MapView

map= (MapView) findViewById (R.id.mapview) ;

//设置显示模式

//交通

map.setTraffic (false) ;

//卫星
```

```
map.setSatellite (true) ;  
  
//街景  
  
map.setStreetView (false) ;  
  
//设置可以缩放  
  
map.setBuiltInZoomControls (true) ;  
  
//设置可以点击  
  
map.setClickable (true) ;  
  
map.setActivated (true) ;  
  
//设置初始地图的中心位置  
  
GeoPoint point2=new GeoPoint ( (int) (31.95*1000000) , (int)  
(121.37*1000000) ) ;  
  
mapCon=map.getController () ;  
  
//设置中心点  
  
mapCon.setCenter (point2) ;  
  
//设置放大级数
```

```
mapCon.setZoom (14) ;  
  
}  
  
public boolean onCreateOptionsMenu (Menu menu) {  
  
//建立菜单  
  
menu.add (0, menuMode, 0, "地图模式") ;  
  
menu.add (0, menuExit, 1, "退出") ;  
  
return super.onCreateOptionsMenu (menu) ;  
  
}  
  
public boolean onKeyDown (int keyCode,KeyEvent event) {  
  
return super.onKeyDown (keyCode,event) ;  
  
}  
  
//在每个MapActivity，必须重写isRouteDisplayed () 方法  
  
protected boolean isRouteDisplayed () {  
  
return false ;
```

}

}

以下通过不同的组合来实现不同的视图。

下面是卫星视图组合：

//设置显示模式

//交通

map.setTraffic (false) ;

//卫星

map.setSatellite (true) ;

//街景

map.setStreetView (false) ;

卫星视图效果如图5-3所示。



图 5-3 卫星视图下地图显示效果

下面是交通视图组合：

//获取MapView

```
map= (MapView) findViewById (R.id.mapview) ;
```

//设置显示模式

//交通

```
map.setTraffic (true) ;
```

//卫星

```
map.setSatellite (false) ;  
  
//街景  
  
map.setStreetView (false) ;
```

交通视图效果如图5-4所示。



图 5-4 交通视图下地图显示效果

下面是街景视图组合：

```
//获取MapView  
  
map= (MapView) findViewById (R.id.mapview) ;
```

```
//设置显示模式
```

```
//交通
```

```
map.setTraffic (false) ;
```

```
//卫星
```

```
map.setSatellite (false) ;
```

```
//街景
```

```
map.setStreetView (true) ;
```

街景视图效果如图5-5所示。



图 5-5 街景视图下地图显示效果

5.2 USB编程

USB（Universal Serial Bus，通用串行总线）是一种快速、灵活的总线接口。USB接口因其具有标准统一、支持热插拔、支持即插即用、可连接多个设备等优点，已经在PC机、平板电脑、手机等设备中得到了广泛使用。本节介绍Android USB的相关编程方法。

5.2.1 USB主从设备

USB是一种主从结构。主机称为Host（主机），从机称为Device（设备）。USB的数据交换只能发生在主机和设备之间，主机和主机、设备和设备之间不能互连。所有的数据传输都由主机主动发起，而设备只是被动地负责应答。例如，在读数据时，USB先发出读命令，设备收到该命令后，才返回数据。

在Android系统中，如果以Android手机作为主机，在Android手机上插入USB从设备，则这种模式称为USB主模式（USB Host）；相反则称为USB从模式或副模式（USB Accessory），如图5-6所示。

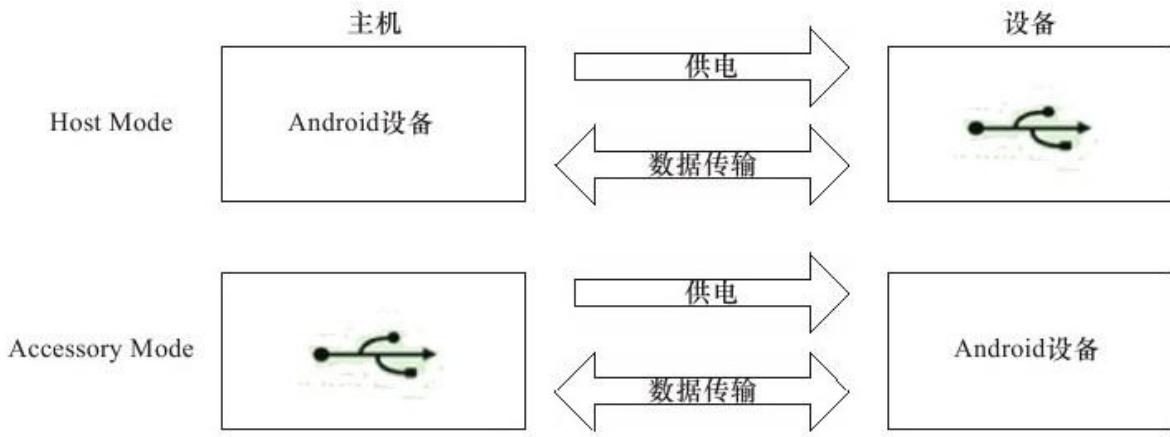


图 5-6 Android USB主从模式示意图

注意 以上两种模式在Android API level-12以下是不支持的，也就是说，只有Android 3.1及更高的版本才支持这两种模式，且只支持一个设备。此外，在Android平台下，关于USB Host和USB Accessory的中文翻译，很多资料上采用直译方式，将USB Host译为主机模式，将USB Accessory译为配件模式，这种翻译方式较直观，所以本书采用这种方式，不再采用前边用的方式。

5.2.2 USB Accessory Mode

在配件模式下，搭载Android系统的设备可以和USB接口的硬件进行交互，被连接的USB硬件（简称配件）作为主机为USB总线提供电源。

注意 配件必须遵守AOA协议（Android Open Accessory protocol,Android开放配件协议），它定义了配件如何检测并与Android设备建立通信等规范。

Android 2.3.4之后已内置AOA协议，而在3.1版本之后，才添加了支持配件模式的相关API，2.3.4版本之后的设备则需要通过附加类库来使用。本文主要介绍API 12之后版本的使用，其主要的USB相关操作都集中在android.hardware.usb这个命名空间中。

1.USB Accessory API简介

配件模式下有两个重要的类：UsbManager和UsbAccessory。其中，通过UsbManager可以获取USB的状态信息，并负责和USB配件进行通信；UsbAccessory代表一个USB配件并且包含获取配件特定信息的方法。

定义UsbManager对象和UsbAccessory对象的方法如下。通过Context.USB_SERVICE可以实例化UsbManager对象。

```
UsbManager manager= (UsbManager) getSystemService  
 (Context.USB_SERVICE) ;
```

通过以下方式获取UsbAccessory对象：

```
UsbAccessory accessory=  
(UsbAccessory) intent.getParcelableExtra  
(UsbManager.EXTRA_ACCESSORY) ;
```

2.Android manifest文件配置

Android系统要想运行一个组件，那么就必须知道组件的相关信息。这些信息就存储在manifest文件中。当发布一个程序时，manifest文件和程序代码、资源等文件一起被打包成一个APK文件。manifest文件是一个XML文件，文件名是AndroidManifest.xml。该文件除了包含组件的相关信息外，还会表明程序需要哪些第三方的组件，以及运行该程序所需要的权限等。

以下描述了在使用USB Accessory API之前，Android manifest文件配置的要求，后面会给出一个Android manifest的例子。

- 因为并不是所有的Android设备都支持USB Accessory API，需要在manifest文件里面使用<uses-feature>元素来声明应用支持它，值为android.hardware.usb.accessory。
- 如果使用android.hardware.usb包，SDK的<version>最小值需要设置为API Level 12级。

如果希望应用在USB配件连接时能接收通知，则在主Activity中需要为 android.hardware.usb.action.USB_ACCESSORY_ATTACHED这个Intent（意图）中指定一对<intent-filter>和<meta-data>元素。<meta-data>元素指向另一个XML资源文件，该文件包含希望得到的配件的一些特定信息。

这个XML资源文件中为希望过滤的配件声明<usb-accessory>元素。每一个<usb-accessory>都可以包含“制造商”、“模式”和“版本”这3个属性。

资源文件要保存在res/xml/目录下。资源文件的名字必须和在<meta-data>元素中指定的名字相同。这个XML资源文件的格式在下面的例子中也会给出。

下面给出一个AndroidManifest.xml和相应的XML资源文件的例子，以便读者更好地理解上述概念。

AndroidManifest.xml中内容如下：

```
<manifest.....>

<uses-feature android:name="android.hardware.usb.accessory"/>

<uses-sdk android:minSdkVersion="
```

.....

```
<intent-filter>

<!--使得应用程序能够发现USB附件设备-->

<action android:name="android.hardware.usb.action.USB_ACCESSORY_

ATTACHED"/>

</intent-filter>

<meta-data

    android:name="android.hardware.usb.action.USB_ACCESSORY_

ATTACHED" android:resource="@xml/accessory_filter"/>

</activity>

</application>

</manifest>
```

在这个例子中，下面的资源文件应该保存在res/xml/accessory_filter.xml中，并且指定只有与其对应的模式、制造商和版本的配件才能够被选择。配件会把这些属性传递给搭载Android系统的设备。相应的XML资源文件如下：

```
<?xml version="1.0" encoding="utf-8"?>

<resources>

<!--model属性表明模式-->

<!--manufacturer表明制造商-->

<!--version表明版本-->

<usb-accessory model="DemoKit" manufacturer="Google" version="1.0"/>

</resources>
```

3. 配件的使用

当用户将USB配件连接到Android设备上时，Android系统会判断该配件是否是应用所感兴趣的配件。如果是，则建立与该配件的通信。当然，要做到这一点，应用还必须完成以下工作：

- 通过一个可以过滤配件附加事件的意图过滤器来找到合适的连接配件，或者枚举所有已连接的配件来找到合适的某个配件。
- 尚未获得许可的用户在与配件通信前需要获得权限。
- 通过合适的端口与配件进行读写通信。

(1) 发现配件

应用可以通过两种方式来发现配件：一种是使用意图过滤器在用户连接配件时对其进行通知；另一种则是通过枚举已经连接的所有配件。如果希望应用能够自动探测到想要的配件，使用一个意图过滤器是非常有用的。但是，如果希望得到一个已连接配件的列表或者不希望通过过滤意图，枚举所有的配件会是一个更好的选择。

使用意图过滤器

为了让应用可以发现一个特定的USB配件，可以使用意图过滤器，即为意图`android.hardware.usb.action.USB_ACCESSORY_ATTACHED`指定一个意图来进行过滤。在这个意图过滤器中，需要指定一个资源文件来特别说明这个USB配件的属性，例如制造商、模式和版本。当连接的配件和意图过滤条件匹配时，应用即会收到通知。

下面的例子展示了如何声明这个意图过滤器：

```
<activity.....>
```

```
.....
```

```
<intent-filter>
```

```
<!--让应用可以发现一个特定的USB配件-->
```

```
<action android:name="android.hardware.usb.action.USB_ACCESSORY_
ATTACHED"/>

</intent-filter>

<!--声明对应的资源文件-->

<meta-data
    android:name="android.hardware.usb.action.USB_ACCESSORY_
ATTACHED" android:resource="@xml/accessory_filter"/>

</activity>
```

下面的例子展示了怎么样声明对应的资源文件，文件指定了希望连接的USB配件的信息。

```
<?xml version="1.0" encoding="utf-8"?>

<resources>

<usb-accessory manufacturer="Google, Inc." model="Demo
Kit" version="1.0"/>

</resources>
```

在activity中，可以通过以下方式获得UsbAccessory，它代表了所连接的配件。

```
UsbAccessory accessory=
```

```
(UsbAccessory) intent.getParcelableExtra  
(UsbManager.EXTRA_ACCESSORY) ;
```

枚举所有支持的配件

当应用在运行的时候，可以在应用中枚举所有能够被识别的配件。通过getAccessoryList () 方法来获得一个包含所有已连接USB配件的数组。

```
UsbManager manager= (UsbManager) getSystemService  
(Context.USB_SERVICE) ;
```

```
UsbAccessory[] accessoryList=manager.getAccessoryList () ;
```

注意 目前一次只能连接一个USB配件，但是这个API设计在未来可用于支持多个配件。

(2) 获得与某个配件通信的权限

在与某个USB配件通信前，应用必须从用户那里获得与之通信的权限。

注意 如果应用在连接配件时是通过一个意图过滤器来发现它们，而用户允许应用来处理这个意图时，它将自动获取通信的权限。如果用户不允许，那么就必须在与配件通信之前明确地在应用中请求获得权限。

一些场合是需要明确的请求权限的，正如“注意”中所提到的那样。再比如用枚举方法获取某个配件时也需要明确的请求通信权限，否则在运行的时候就会报错。

为了获得权限，首先需要创建一个广播接收器（broadcast receiver）。在调用requestPermission（）方法时，这个接收器会监听广播中的意图。通过调用requestPermission（）方法为用户弹出一个获取权限对话框。

下面的例子展示了如何创建一个广播接收器。

```
//定义权限字符串
```

```
private static final String  
ACTION_USB_PERMISSION="com.android.example.USB_PERMISSION"  
N" ;
```

```
//新建广播接收器
```

```
private final BroadcastReceiver mUsbReceiver=new BroadcastReceiver ()  
{  
  
    public void onReceive (Context context,Intent intent) {  
  
        //获取当前Intent的权限字符串  
  
        String action=intent.getAction () ;  
  
        //判断权限是否和定义的相同  
  
        if (ACTION_USB_PERMISSION.equals (action) ) {  
  
            synchronized (this) {  
  
                //声明UsbAccessory对象  
  
                UsbAccessory accessory= (UsbAccessory)  
                    intent.getParcelableExtra  
                        (UsbManager.EXTRA_ACCESSORY) ;  
  
                if (intent.getBooleanExtra  
                    (UsbManager.EXTRA_PERMISSION_GRANTED,false) ) {  
  
                    if (accessory !=null) {
```

```
//调用方法建立与某个配件的通信  
}  
  
}  
  
else{  
  
//没有获取权限  
}  
  
}  
  
}  
  
}  
  
};
```

为了注册广播接收器，需要将下面代码放在activity中的onCreate () 方法中。

```
//获取系统服务对象  
UsbManager mUsbManager= (UsbManager)  
getSystemService (Context.USB_SERVICE) ;
```

```
//定义权限字符串  
  
private static final String ACTION_USB_PERMISSION=  
"com.android.example.USB_PERMISSION" ;  
  
.....  
  
//稍后处理配件通信事件  
  
mPermissionIntent=PendingIntent.getBroadcast (this, 0,  
new Intent (ACTION_USB_PERMISSION) , 0) ;  
  
//新建过滤器  
  
IntentFilter filter=new IntentFilter (ACTION_USB_PERMISSION) ;  
  
//注册接收器  
  
registerReceiver (mUsbReceiver,filter) ;  
  
当需要弹出获取权限的对话框时，则要调用requestPermission () 方  
法。  
  
UsbAccessory accessory ;  
  
.....
```

```
mUsbManager.requestPermission (accessory,mPermissionIntent) ;
```

当用户回应这个对话框时，广播接收器就会收到一个包含boolean值来表示结果的EXTRA_PERMISSION_GRANTED字段的意图。在连接配件之前检查这个字段的值是否为true。

(3) 与配件进行通信

可以通过使用UsbManager这个类与配件进行通信，通过这个类可以获得一个文件描述符，然后可以利用该描述符设置输入和输出流来读取和写入数据。这些流用来代表输入和输出的批量端口。最好另外建立一个线程来让设备与配件进行通信，因为这样就可以不需要将主线程锁起来了。

下面的例子展示了如何与一个配件进行通信。

```
//新建UsbAccessory对象
```

```
UsbAccessory mAccessory ;
```

```
//新建ParcelFileDescriptor对象
```

```
ParcelFileDescriptor mFileDescriptor ;
```

```
FileInputStream mInputStream ;
```

```
FileOutputStream mOutputStream ;  
.....  
  
//打开通信  
  
private void openAccessory () {  
  
//获取设备文件描述  
  
mFileDescriptor=mUsbManager.openAccessory (mAccessory) ;  
  
//设备文件描述不为null  
  
if (mFileDescriptor !=null) {  
  
//获取文件描述  
  
FileDescriptor fd=mFileDescriptor.getFileDescriptor () ;  
  
mInputStream=new FileInputStream (fd) ;  
  
mOutputStream=new FileOutputStream (fd) ;  
  
//打开新的线程  
  
Thread thread=new Thread (null,this, "AccessoryThread") ;
```

```
//启动线程
```

```
    thread.start () ;
```

```
}
```

```
}
```

在这个线程的run () 方法中，可以通过FileInputStream或者 FileOutputStream对象来对配件进行读取和写出数据操作。当通过 FileInputStream对象读取配件中的数据时，请确保所使用的缓存能够存储下USB数据包中的数据。Android配件协议支持高达16384字节的数据包缓存区，所以可以简单地一直设定缓存区为这个大小。

(4) 终止与配件的通信

当已经完成与配件的通信之后，或者该配件被移除了，通过调用close () 方法来关闭已经打开的文件描述符。为了监听分离这样的事件，需要创建如下广播接收器：

```
//新建监听
```

```
BroadcastReceiver mUsbReceiver=new BroadcastReceiver () {
```

```
    public void onReceive (Context context,Intent intent) {
```

```
//获取事件字符串  
  
String action=intent.getAction () ;  
  
//对比是否为需要处理的事件权限  
  
if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals  
(action) ) {  
  
//新建UsbAccessory对象  
  
UsbAccessory accessory= (UsbAccessory) intent.  
getParcelableExtra (UsbManager.EXTRA_ACCESSORY) ;  
  
if (accessory !=null) {  
  
//调用方法终止与配件的通信，清理内存  
  
}  
  
}  
  
};
```

要在应用中创建这个广播接收器，而不是在manifest文件中创建，且允许应用只能在其运行的时候处理这样的配件分离事件。这样的话，配件分离这个事件就只向正在运行的应用广播，而不是向所有的应用进行广播。

5.2.3 USB Host Mode

当Android设备工作在主机模式下的时候，Android设备作为主机，为USB总线供电并与USB设备进行通信。同样，仅Android 3.1及更高版本支持这种模式。主机模式使用步骤与配件模式类似。这里针对主机模式开发，介绍Android USB Host API主要涉及的几个类。

(1) UsbManager

该类负责管理USB设备，一般可以通过以下方法获得此对象的一个实例：

```
UsbManager manager= (UsbManager) getSystemService  
(Context.USB_SERVICE) ;
```

该类提供的主要方法如表5-1所示。

表 5-1 UsbManager 类提供的主要方法

方 法	解 释
getDeviceList()	获得设备列表，返回的是一个 HashMap
hasPermission(UsbDevice device)	判断应用程序是否有接入此 USB 设备的权限。如果有则返回 true，否则返回 false
openDevice(UsbDevice device)	打开 USB 设备，以便向此 USB 设备传输数据。返回一个关于此 USB 设备的连接
requestPermission(UsbDevice device,PendingIntent pi)	向 USB 设备请求临时的接入权限

(2) UsbDevice

该类代表一个USB设备，每个设备都包含了一个或多个接口，每个接口又包含一个或多个端口用来与此设备传输数据。

该类提供的主要方法如表5-2所示。

表 5-2 UsbDevice 类提供的主要方法

方 法	解 释
getDeviceClass()	返回此 USB 设备的类别，用一个整型来表示
getDeviceId()	返回唯一标识此设备的 ID 号，用一个整型来表示
getDeviceName()	返回此设备的名称，用一个字符串来表示
getDeviceProtocol()	返回此设备的协议类别，用一个整型来表示
getDeviceSubclass()	返回此设备的子类别，用一个整型来表示
getVendorId()	返回生产商 ID
getProductId()	返回产品 ID
getInterfaceCount()	返回此设备的接口数量
getInterface(int index)	得到此设备的一个接口，返回一个 UsbInterface

(3) UsbInterface

该类代表USB设备的一个接口。该类提供的主要方法如表5-3所示。

表 5-3 UsbInterface 类提供的主要方法

方 法	解 释
getId()	得到该接口的 ID 号
getInterfaceClass()	得到该接口的类别
getInterfaceSubclass()	得到该接口的子类
getInterfaceProtocol()	得到该接口的协议类别
getEndpointCount()	获得关于此接口的端口数量
getEndpoint(int index)	对于指定的 index 获得此接口的一个端口，返回一个 UsbEndpoint

(4) UsbEndpoint

代表一个接口的某个端口[1]的类。该类提供的主要方法如表5-4所示。

表 5-4 UsbEndpoint 类提供的主要方法

方 法	解 释
getAddress()	获得此端口的地址
getAttributes()	获得此端口的属性
getDirection()	获得此端口的数据传输方向

(5) UsbDeviceConnection

代表USB连接的一个类。用此连接可以向USB设备传输数据，可以通过调用openDevice (UsbDevice) 方法来得到该类的一个实例。

该类提供的主要方法如表5-5所示。

表 5-5 UsbDeviceConnection 类提供的主要方法

方 法	解 释
bulkTransfer(UsbEndpoint endpoint,byte[] buffer,int length,int timeout)	通过给定的端口来进行大量的数据传输，传输的方向取决于该端口的方向，buffer 是要发送或接收的字节数组，length 是该字节数组的长度。传输成功则返回所传输的字节数组的长度，失败则返回一个负数
controlTransfer(int requestType,int request,int value, int index, byte[] buffer, int length, int timeout)	该方法通过端口向此设备传输数据，传输的方向取决于请求的类别，如果 requestType 取值为 USB_DIR_OUT 则为写数据，取值为 USB_DIR_IN 则为读数据

[1]这里端口是指接口电路中的一些寄存器，这些寄存器可以用来存放数据信息、控制信息和状态信息，每个端口使用端口地址唯一标识。

5.2.4 实战案例：Android和Arduino交互

Arduino是一个开源的单片机控制器，采用了基于开放源代码的软硬件平台，并且具有类似Java、C语言的IDE集成开发环境。

构建Arduino开发环境的步骤如下：

步骤1 硬件配置。Arduino提供了多种方案供Arduino和Android之间进行通信，包括Wi-Fi、蓝牙、USB等。Arduino硬件实物如图5-7所示。

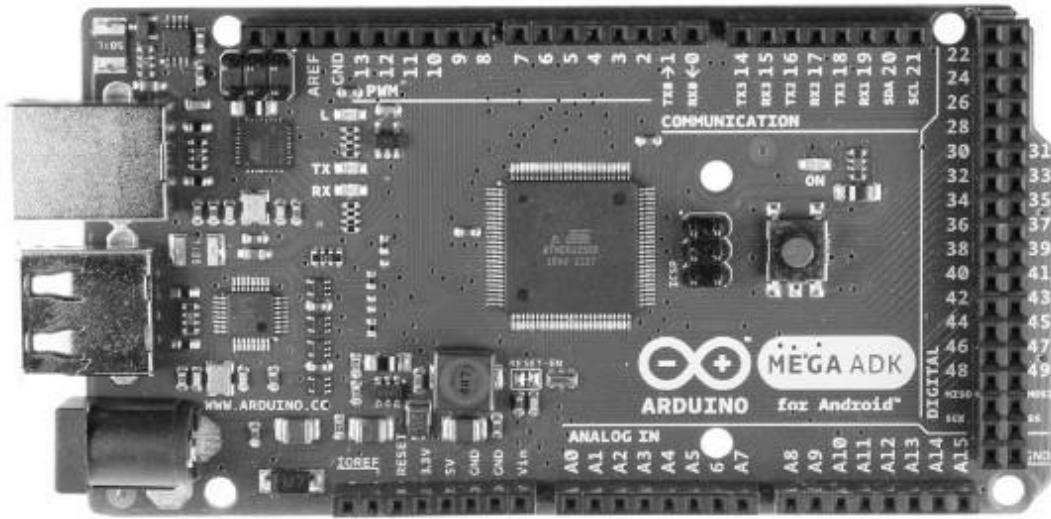


图 5-7 Arduino硬件实物图

步骤2 软件下载。到网址<http://arduino.cc/en/Main/Software>下载适合系统的Arduino IDE，目前支持Windows、Mac OS X、Linux等操作系统。下载后直接解压即完成安装。

步骤3 选择开发板。双击Arduino的图标可以打开Arduino IDE开发界面，Arduino具有多种硬件核心和外围IO设计，在使用Arduino IDE开发的时候需要选择适合自己的开发板。

依次选择Tools → Board → Arduino Mega 2560 or ADK，就可以选择到案例需要的型号，在该界面下可以进行编程、编译、下载等一系列的操作，如图5-8所示。

The screenshot shows the Arduino IDE interface with the title bar "sketch_sep21a | Arduino 1.0.1". The menu bar includes File, Edit, Sketch, Tools, and Help. Below the menu is a toolbar with icons for saving, loading, and uploading sketches. The main code editor window displays the following C++ code:

```
#include <Max3421e.h>
#include <Usb.h>
#include <AndroidAccessory.h>

#define COMMAND_LED 0x2
#define TARGET_PIN_2 0x2
#define VALUE_ON 0x1
#define VALUE_OFF 0x0

#define PIN 13

AndroidAccessory acc("Manufacturer",
                      "Model",
                      "Description",
                      "Version",
                      "URI",
                      "Serial");
```

The status bar at the bottom left shows the page number "17" and at the bottom right shows "Arduino Mega 2560 or Mega ADK on COM12".

图 5-8 Arduino Mega 2560 or ADK示意图

步骤4 硬件连线。将Arduino上标准USB接口连接到Android的USB接口，通过其进行通信；Arduino上的mini母口接到计算机的USB接口上，通过其为Arduino供电，如图5-9所示。

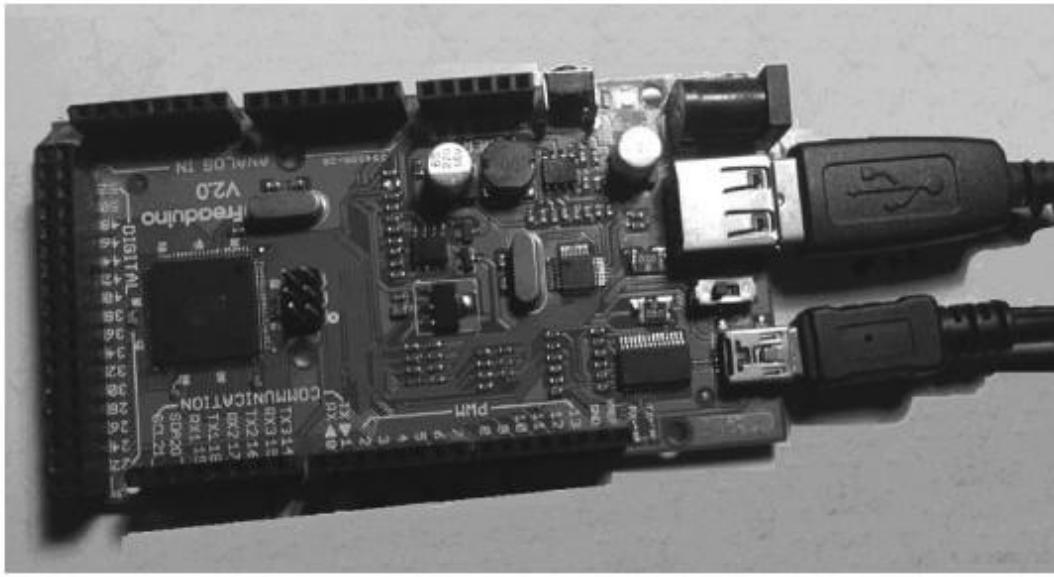


图 5-9 Arduino与USB间的连线

最后，分析一下源代码。manifest文件如下，其中给出了硬件特征要求。

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
package="org.chenwen.android2arduino"  
  
        android:versionCode="1"  
  
        android:versionName="1.0">  
  
<uses-sdk  
  
        android:minSdkVersion="12"
```

```
    android:targetSdkVersion="15"/>

    <!--要求支持USB设备作为主控制器-->

    <uses-feature android:name="android.hardware.usb.accessory"/>

    <!--唤醒锁定允许程序在手机屏幕关闭后后台进程仍然运行-->

    <uses-permission android:name="android.permission.WAKE_LOCK"/>

<application

    android:icon="@drawable/ic_launcher"

    android:label="@string/app_name"

    android:theme="@style/AppTheme">

        <activity

            android:name=".Android2ArduinoActivity"

            android:label="@string/title_activity_project_one">

            <!--定义为启动项-->

            <intent-filter>
```

```
<action android:name="android.intent.action.MAIN"/>

<category android:name="android.intent.category.LAUNCHER"/>

</intent-filter>

</activity>

</application>

</manifest>
```

主要的控制逻辑在Android2ArduinoActivity中，其代码如下：

```
import android.os.Bundle ;

import android.app.Activity ;

import android.view.Menu ;

import java.io.FileDescriptor ;

import java.io.FileInputStream ;

import java.io.FileOutputStream ;

import java.io.IOException ;
```

```
import android.app.PendingIntent ;  
  
import android.content.BroadcastReceiver ;  
  
import android.content.Context ;  
  
import android.content.Intent ;  
  
import android.content.IntentFilter ;  
  
import android.os.AsyncTask ;  
  
import android.os.ParcelFileDescriptor ;  
  
import android.os.PowerManager ;  
  
import android.util.Log ;  
  
import android.widget.Button ;  
  
import android.widget.CompoundButton ;  
  
import android.widget.CompoundButton.OnCheckedChangeListener ;  
  
import android.widget.ToggleButton ;  
  
import android.hardware.usb.UsbAccessory ;
```

```
import android.hardware.usb.UsbManager ;  
  
//使用Android控制Arduino的输出  
  
public class Android2ArduinoActivity extends Activity{  
  
    //定义调试使用的TAG字符串  
  
    private static final String TAG="Android2ArduinoActivity" ;  
  
    //创建PendingIntent对象  
  
    private PendingIntent mPermissionIntent ;  
  
    //定义权限字符串  
  
    private static final String ACTION_USB_PERMISSION=  
        "com.android.example.USB_PERMISSION" ;  
  
    //创建处理权限  
  
    private boolean mPermissionRequestPending ;  
  
    //创建mUsbManager为USB管理类  
  
    private UsbManager mUsbManager ;
```

```
//创建mAccessory为UsbAccessory类  
  
private UsbAccessory mAccessory ;  
  
//创建ParcelFileDescriptor对象， 用于操作原始的文件描述符  
  
private ParcelFileDescriptor mFileDescriptor ;  
  
private FileInputStream mInputStream ;  
  
private FileOutputStream mOutputStream ;  
  
//定义通信格式中的命令类型  
  
private static final byte COMMAND_LED=0x2 ;  
  
//定义通信格式中的命令目标  
  
private static final byte TARGET_PIN_2=0x2 ;  
  
//定义通信命令内容， 0x1为打开命令  
  
private static final byte VALUE_ON=0x1 ;  
  
//定义通信命令内容， 0x0为关闭命令  
  
private static final byte VALUE_OFF=0x0 ;
```

```
//定义ToggleButton  
  
//ToggleButton通过一个带有亮度指示同时默认文本为ON或OFF的按钮  
  
//显示选中/未选中状态  
  
private ToggleButton ledToggleButton ;  
  
//定义电源管理锁PowerManager.WakeLock  
  
PowerManager.WakeLock wl ;  
  
@Override  
  
public void onCreate (Bundle savedInstanceState) {  
  
super.onCreate (savedInstanceState) ;  
  
//USB管理类实例化  
  
mUsbManager= (UsbManager) getSystemService  
(Context.USB_SERVICE) ;  
  
mPermissionIntent=PendingIntent.getBroadcast (this, 0, new Intent (  
ACTION_USB_PERMISSION) , 0) ;  
  
//新建过滤器
```

```
IntentFilter filter=new IntentFilter (ACTION_USB_PERMISSION) ;  
  
//添加过滤器  
  
filter.addAction  
    (UsbManager.ACTION_USB_ACCESSORY_DETACHED) ;  
  
//注册服务，处理标签为  
UsbManager.ACTION_USB_ACCESSORY_DETACHED的广播  
  
registerReceiver (mUsbReceiver,filter) ;  
  
//获取电源管理  
  
PowerManager pm= (PowerManager) getSystemService  
    (Context.POWER_SERVICE) ;  
  
//设置电源管理策略  
  
//保持CPU运转，关闭屏幕和键盘灯  
  
wl=pm.newWakeLock (PowerManager.PARTIAL_WAKE_LOCK, "My  
Tag") ;  
  
//开启电源锁  
  
wl.acquire () ;
```

```
//加载布局

setContentView (R.layout.activity_layout_a2a) ;

ledToggleButton= (ToggleButton) findViewById

(R.id.led_toggle_button) ;

//添加按键监听

ledToggleButton.setOnCheckedChangeListener

(toggleButtonCheckedChangeListener) ;

}

@Override

public void onResume () {

super.onResume () ;

//检测当前是否存在通信，如果有则返回

if (mInputStream !=null & & mOutputStream !=null) {

return ;
```

```
}
```

```
//获取当前连接的USB设备列表
```

```
UsbAccessory[]accessories=mUsbManager.getAccessoryList () ;
```

```
//获取当前连接的第一个USB设备
```

```
UsbAccessory accessory= (accessories==null?null :
```

```
accessories[0]) ;
```

```
//如果存在连接的USB设备
```

```
if (accessory !=null) {
```

```
//是否具有权限
```

```
if (mUsbManager.hasPermission (accessory) ) {
```

```
//打开通信
```

```
openAccessory (accessory) ;
```

```
}else{
```

```
synchronized (mUsbReceiver) {
```

```
//没有获取权限

if ( ! mPermissionRequestPending) {

mUsbManager.requestPermission

(accessory,mPermissionIntent)  ;

mPermissionRequestPending=true ;

}

}

}

}

else{

//没有相关设备

Log.d (TAG, "mAccessory is null")  ;

}

}

@Override
```

```
public void onPause () {  
    super.onPause () ;  
  
    //关闭通信  
  
    closeAccessory () ;  
  
}  
  
@Override  
  
public void onDestroy () {  
    super.onDestroy () ;  
  
    //注销广播接收器  
  
    unregisterReceiver (mUsbReceiver) ;  
  
    //释放电源锁  
  
    wl.release () ;  
  
}  
  
//定义按钮监听
```

```
OnCheckedChangeListener toggleButtonCheckedChangeListener=new  
OnCheckedChangeListener () {  
  
//当按下的时候触发该事件  
  
public void onCheckedChanged (CompoundButton buttonView,  
boolean isChecked) {  
  
//获取到点击按钮的ID值，并检测其是否为LED灯的控制按钮  
  
if (buttonView.getId () ==R.id.led_toggle_button) {  
  
new AsyncTask<Boolean(Void,Void> () {  
  
@Override  
  
protected Void doInBackground (Boolean.....  
params) {  
  
//发送命令  
  
sendLedSwitchCommand (TARGET_PIN_2, params[0]) ;  
  
return null ;
```

```
}

}.execute (isChecked) ;

}

}

};

//定义USB权限的广播接收器

private final BroadcastReceiver mUsbReceiver=new BroadcastReceiver () {

@Override

public void onReceive (Context context,Intent intent) {

String action=intent.getAction () ;

//收到使用USB设备权限的广播

if (ACTION_USB_PERMISSION.equals (action) ) {

synchronized (this) {

UsbAccessory accessory=
```

```
(UsbAccessory) intent.getParcelableExtra  
(UsbManager.EXTRA_ACCESSORY) ;  
  
//允许使用  
  
if (intent.getBooleanExtra  
(UsbManager.EXTRA_PERMISSION_GRANTED,false) ) {  
  
openAccessory (accessory) ;  
  
}  
  
//拒绝使用  
  
Log.d (TAG, "permission denied for accessory"+accessory) ;  
  
}  
  
mPermissionRequestPending=false ;  
  
}  
  
//收到通信中止广播  
  
}else if (UsbManager.ACTION_USB_ACCESSORY_DETACHED.equals  
(action) ) {
```

```
UsbAccessory accessory= (UsbAccessory) intent.  
getParcelableExtra (UsbManager.EXTRA_ACCESSORY) ;  
  
if (accessory !=null & &accessory.equals (mAccessory) ) {  
  
closeAccessory () ;  
  
}  
  
}  
  
}  
  
};  
  
//打开通信  
  
private void openAccessory (UsbAccessory accessory) {  
  
//尝试打开通信  
  
mFileDescriptor=mUsbManager.openAccessory (accessory) ;  
  
//判断是否有相关设备  
  
if (mFileDescriptor !=null) {
```

```
mAccessory=accessory ;  
  
FileDescriptor fd=mFileDescriptor.getFileDescriptor () ;  
  
mInputStream=new FileInputStream (fd) ;  
  
mOutputStream=new FileOutputStream (fd) ;  
  
}else{  
  
//打开通信失败  
  
}  
  
}  
  
//关闭通信  
  
private void closeAccessory () {  
  
try{  
  
//若当前通道没有关闭，则关闭  
  
if (mFileDescriptor !=null) {  
  
mFileDescriptor.close () ;
```

```
}

}catch (IOException e) {

}finally{

mFileDescriptor=null ;

mAccessory=null ;

}

}

//发送开关指令

public void sendLedSwitchCommand (byte target,boolean isSwitchedOn)

{

//定义通信指令的存储字节数组

byte[]buffer=new byte[3] ;



//发送命令的格式

buffer[0]=COMMAND_LED ;




//发送命令的目标
```

```
buffer[1]=target ;  
  
//按钮被按下  
  
if (isSwitchedOn) {  
  
//发送命令：打开  
  
buffer[2]=VALUE_ON ;  
  
}  
  
}else{  
  
//按钮没有被按下  
  
//发送命令：关闭  
  
buffer[2]=VALUE_OFF ;  
  
}  
  
//发送  
  
if (mOutputStream !=null) {  
  
try{  
  
mOutputStream.write (buffer) ;
```

```
 }catch (IOException e) {  
}  
}  
}  
  
}  
  
//创建menu菜单  
  
@Override  
  
public boolean onCreateOptionsMenu (Menu menu) {  
    getMenuInflater () .inflate (R.menu.activity_android_arduino,menu) ;  
  
    return true ;  
}  
}
```

以下是Arduino中的控制代码：

```
#include<Max3421e.h>
```

```
#include<Usb.h>
```

```
#include<AndroidAccessory.h>
```

```
//定义通信格式中的命令类型
```

```
#define COMMAND_LED 0x2
```

```
//定义通信格式中的命令目标
```

```
#define TARGET_PIN_2 0x2
```

```
//定义打开命令
```

```
#define VALUE_ON 0x1
```

```
//定义关闭命令
```

```
#define VALUE_OFF 0x0
```

```
//定义控制的引脚
```

```
#define PIN 13
```

```
AndroidAccessory acc ("Manufacturer",
```

```
"Model",
```

```
"Description",
```

```
"Version",  
"URI",  
"Serial") ;
```

//创建字节数组，用于接收消息

```
byte rcvmsg[3] ;
```

//配置基本通信设置

```
void setup () {
```

```
Serial.begin (19200) ;
```

```
acc.powerOn () ;
```

```
pinMode (PIN,OUTPUT) ;
```

```
}
```

```
void loop () {
```

```
if (acc.isConnected () ) {
```

//读取收到的数据

```
int len=acc.read (rcvmsg,sizeof (rcvmsg) , 1) ;
```

```
if (len>0) {
```

```
//获取命令类型
```

```
if (rcvmsg[0]==COMMAND_LED) {
```

```
//获取命令目标
```

```
if (rcvmsg[1]==TARGET_PIN_2) {
```

```
//获取命令值
```

```
byte value=rcvmsg[2] ;
```

```
//将获取到的值写入引脚中
```

```
if (value==VALUE_ON) {
```

```
//向引脚PIN中写入高电平
```

```
digitalWrite (PIN,HIGH) ;
```

```
}else if (value==VALUE_OFF) {
```

```
//向引脚PIN中写入低电平
```

```
digitalWrite (PIN,LOW) ;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

我们这里选择第13个引脚作为Android控制Arduino的响应引脚，是因为在Arduino板卡上第13引脚有一个对应的LED灯，试验的时候可以方便地看到其状态，如图5-10所示。

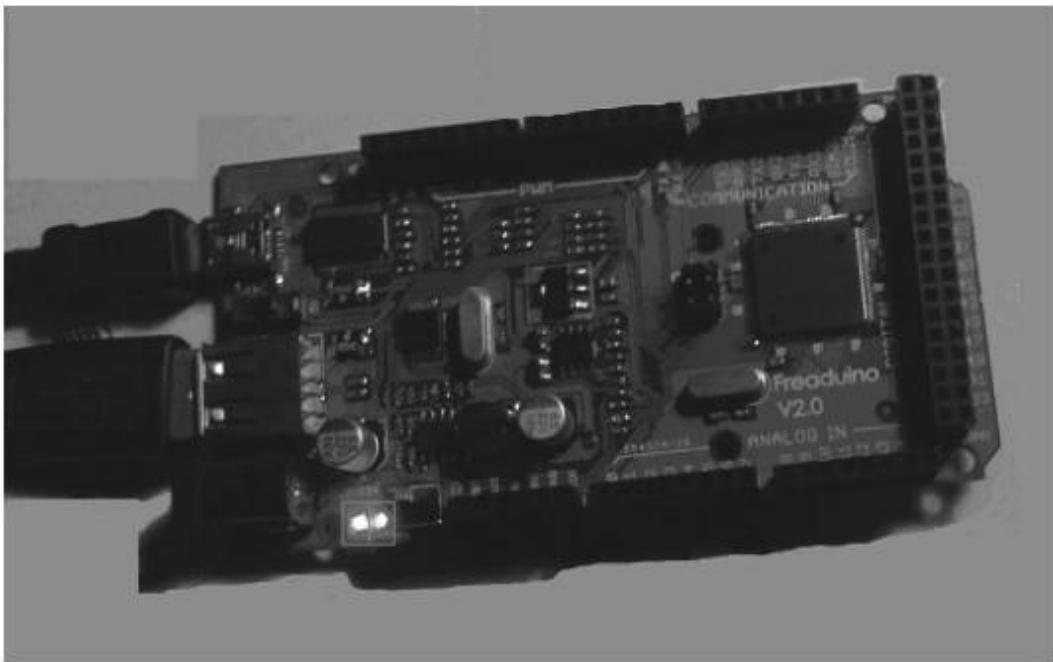


图 5-10 Arduino的响应引脚灯亮效果

在图5-10中，左下方有2个LED灯亮着，右边的那个是与13端口对应的LED灯，左边的是工作信号灯。根据所选择板块的不同，其位置可能有差异。

5.3 Wi-Fi编程

Wi-Fi（Wireless Fidelity）又称802.11b标准，是IEEE定义的一个无线网络通信的工业标准。该技术使用的是2.4GHz附近的频段。

Wi-Fi主要特性如下：

- 速度快，最高带宽为11 Mbps。
- 可靠性高，在信号较弱或有干扰的情况下，带宽可自动调整为5.5Mbps、2Mbps或1Mbps，有效地保障了网络的稳定性和可靠性。
- 距离较远，在开放性区域，通信距离可达305米，在封闭性区域，通信距离为100米左右。
- 方便与现有的有线以太网络整合，组网的成本更低。

5.3.1 Android Wi-Fi相关类

Android中Wi-Fi是按层次结构设计的，Android Wi-Fi模块的层级结构如图5-11所示。

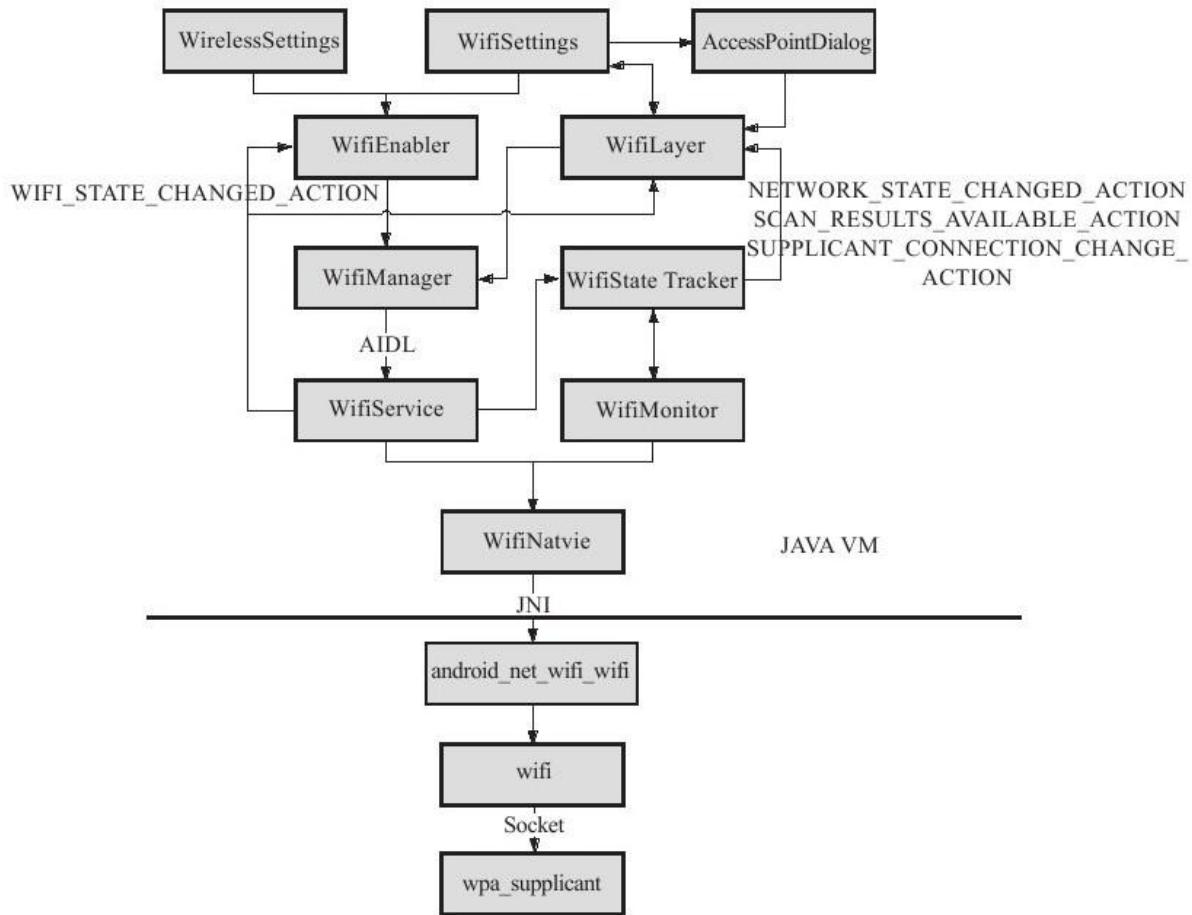


图 5-11 Android Wi-Fi模块层次结构图

从用户的角度看，Android Wi-Fi模块自下向上可以看为5层：硬件驱动程序、wpa_supplicant、JNI、WiFi API、WifiSettings应用程序。

■wpa_supplicant是一个开源库，是Android实现Wi-Fi功能的基础。它从上层接到命令后，通过Socket与硬件驱动进行通信，操作硬件完成需要的操作。

■JNI（Java Native Interface）实现了Java代码与其他代码的交互，使得在Java虚拟机中运行的Java代码能够与用其他语言编写的应用程序和库进行交互。在Android中，JNI可以让Java程序调用C程序。

■WiFi API使应用程序可以使用Wi-Fi功能。

■Wifi Settings应用程序是Android中自带的一个应用程序，选择手机的Settings → Wireless & networks → Wi-Fi，可以让用户手动打开或关闭Wi-Fi功能。当用户打开Wi-Fi功能后，它会自动搜索周围的无线网络，并以列表的形式显示，供用户选择。默认会连接用户上一次成功连接的无线网络。

在android.net.wifi包中提供了一些类管理设备的Wi-Fi功能，主要包括ScanResult、 wifiConfiguration、 WifiInfo和WifiManager。

（1）ScanResult类

ScanResult类主要是通过Wi-Fi硬件的扫描来获取一些周边的Wi-Fi热点（Access Point，如无线路由器等）的信息。该类含以下5个域，如表5-6所示。

表 5-6 ScanResult 类包含的域

返回类型	域 名	解 释
public String	BSSID	接入点的地址
public String	SSID	网络的名字
public String	capabilities	网络性能，包括接入点支持的认证、密钥管理、加密机制等
public int	frequency	以 MHz 为单位的接入频率
public int	level	以 dBm 为单位的信号强度

该类还提供了一个方法——`toString ()` 方法，可将结果转换为简洁、易读的字符串形式。

(2) wifiConfiguration类

通过该类可获取一个Wi-Fi网络的网络配置，包括安全配置等。该类包换6个子类，如表5-7所示。

表 5-7 wifiConfiguration 子类列表

子 类	解 释
WifiConfiguration.AuthAlgorithm	获取 IEEE 802.11 的加密方法
WifiConfiguration.GroupCipher	获取组密钥
WifiConfiguration.KeyMgmt	获取密码管理体制
WifiConfiguration.PairwiseCipher	获取 WPA 方式的成对密钥
WifiConfiguration.Protocol	获取加密协议
WifiConfiguration.Status	获取当前网路状态

每个子类都以常量形式给出相关的配置信息，有些还包含域和方法。内容较多，在此不再一一列举，需要用到的时候可以查Android SDK。

(3) WifiInfo类

通过该类可以获得已经建立的或处于活动状态的Wi-Fi网络的状态信息。该类提供的方法较多，部分常用方法如表5-8所示。

表 5-8 WifiInfo 类中部分方法

方 法	解 释
getBSSID()	获取当前接入点的 BSSID (basic service set identifier)
getIpAddress()	获取 IP 地址
getLinkSpeed()	获得当前连接的速度
getMacAddress()	获得 MAC 地址
getRssi()	获得 802.11n 网络的信号强度指示
getSSID()	获得网络 SSID (service set identifier)
getSupplicantState()	返回客户端状态的信息 (SupplicantState 对象的形式)

(4) wifiManager类

该类用于管理Wi-Fi连接，其定义了26个常量和23个方法。限于篇幅，这里仅列出部分常用方法，如表5-9所示。

表 5-9 wifiManager 类中部分方法

方 法	解 释
addNetwork(WifiConfiguration config)	向设置好的网络里添加新网络
calculateSignalLevel(int rssi , int numLevels)	计算信号的等级
compareSignalLevel(int rssiA, int rssiB)	对比两个信号的强度
createWifiLock(int lockType, String tag)	创建一个 Wi-Fi 锁，锁定当前的 Wi-Fi 连接
disableNetwork(int netId)	让一个网络连接失效
disconnect()	与当前接入点断开连接
enableNetwork(int netId, Boolean disableOthers)	与之前设置好的网络建立连接
getConfiguredNetworks()	客户端获取网络连接的状态
getConnectionInfo()	获取当前连接的信息
getDhcpInfo()	获取 DHCP 的信息
getScanResults()	获取扫描测试的结果
getWifiState()	获取一个 Wi-Fi 接入点是否有效
isWifiEnabled()	判断一个 Wi-Fi 连接是否有效
pingSuplicant()	看看客户后台程序是否响应请求
reassociate()	重连到接入点，即使已经连接上了
reconnect()	如果现在没有连接的话，则重连接
removeNetwork (int netId)	移除某一个特定网络
saveConfiguration()	保留一个配置信息
setWifiEnabled (boolean enabled)	让一个连接有效或失效
startScan()	开始扫描
updateNetwork(WifiConfiguration config)	更新一个网络连接的信息

需要指出的是，Wi-Fi网卡的状态是由一系列的整型常量来表示的。

- WIFI_STATE_DISABLED:Wi-Fi网卡不可用，用整型常量1表示。
- WIFI_STATE_DISABLING:Wi-Fi网卡正在关闭，用整型常量0表示。
- WIFI_STATE_ENABLED:Wi-Fi网卡可用，用整型常量3表示。
- WIFI_STATE_ENABLING:Wi-Fi网正在打开，启动需要一段时间，用整型常量2表示。
- WIFI_STATE_UNKNOWN：未知网卡状态，用整型常量4表示。

此外wifiManager还提供了一个子类wifiManagerLock。 wifiManagerLock 的作用是这样的：在普通的状态下，如果Wi-Fi的状态处于闲置，那么网络将会暂时中断；但是如果把当前的网络状态锁上，那么Wi-Fi连通将会保持在一定状态，在结束锁定之后，才会恢复常态。

5.3.2 Android Wi-Fi基本操作

Android实现Wi-Fi编程步骤包括Wi-Fi启动、AP扫描及配置AP参数、连接AP及配置IP地址等。在上一节介绍Wi-Fi相关类的基础上，本节通过一段代码示例来具体描述Wi-Fi操作的细节。

```
import java.util.ArrayList ;  
  
import java.util.List ;  
  
import android.content.Context ;  
  
import android.net.ConnectivityManager ;  
  
import android.net.NetworkInfo.State ;  
  
import android.net.wifi.ScanResult ;  
  
import android.net.wifi.WifiConfiguration ;  
  
import android.net.wifi.WifiInfo ;
```

```
import android.net.wifi.WifiManager ;  
  
import android.net.wifi.WifiManager.WifiLock ;  
  
import android.util.Log ;  
  
//Wi-Fi相关操作  
  
public class WifiAdmin{  
  
    //定义调试使用的TAG字符串  
  
    private final static String TAG="WifiAdmin" ;  
  
    //新建字符串缓存  
  
    private StringBuffer mStringBuffer=new StringBuffer () ;  
  
    //保存扫描结果列表  
  
    public List<ScanResult>listResult=new ArrayList<ScanResult> () ;  
  
    //创建ScanResult类  
  
    private ScanResult mScanResult ;  
  
    //创建WifiManager对象
```

```
private WifiManager mWifiManager ;  
  
//创建WifiInfo对象  
  
private WifiInfo mWifiInfo ;  
  
//网络连接列表  
  
private List<WifiConfiguration> wifiConfigList=new ArrayList<  
WifiConfiguration> () ;  
  
//创建一个WifiLock  
  
WifiLock mWifiLock ;  
  
//创建连接管理器  
  
private ConnectivityManager connManager ;  
  
private Context mContext ;  
  
//Wi-Fi配置列表  
  
private List<WifiConfiguration> wifiConfigedSpecifiedList=  
new ArrayList<WifiConfiguration> () ;  
  
//利用ArrayList类实例化List集合
```

```
private State state ;  
  
//定义是否已经连接  
  
public boolean isConnected=false ;  
  
//构造方法  
  
public WifiAdmin (Context context) {  
  
    mContext=context ;  
  
    //获取WifiManager系统服务  
  
    mWifiManager= (WifiManager) context.  
        getSystemService (Context.WIFI_SERVICE) ;  
  
    //获取Wi-Fi连接信息  
  
    mWifiInfo=mWifiManager.getConnectionInfo () ;  
  
    //获取连接管理系統服务  
  
    connManager= (ConnectivityManager) mContext  
        .getSystemService (Context.CONNECTIVITY_SERVICE) ;
```

```
}
```

```
//重新获取当前Wi-Fi的连接信息
```

```
public void againGetWifiInfo () {
```

```
mWifiInfo=mWifiManager.getConnectionInfo () ;
```

```
}
```

```
//判断用户是否开启Wi-Fi网卡，true为开启
```

```
public boolean isNetCardFriendly () {
```

```
return mWifiManager.isWifiEnabled () ;
```

```
}
```

```
//判断系统当前是否已连接Wi-Fi,true表示当前已连接Wi-Fi
```

```
public boolean isConnectioning () {
```

```
state=connManager.getNetworkInfo (ConnectivityManager.TYPE_WIFI)
```

```
.getState () ;
```

```
if (State.CONNECTING==state) {
```

```
        return true ;  
  
    }else{  
  
        return false ;  
  
    }  
  
}  
  
//判断系统当前是否已连接Wi-Fi,true表示当前已连接Wi-Fi  
  
public boolean isConnected () {  
  
    //获取连接状态  
  
    state=connManager.getNetworkInfo  
    (ConnectivityManager.TYPE_WIFI) .getState () ;  
  
    if (State.CONNECTED==state) {  
  
        return true ;  
  
    }else{  
  
        return false ;  
    }  
}
```

```
}

}

//得到当前的网络连接状态

public State getCurrentState () {

    state=connManager.getNetworkInfo (ConnectivityManager.TYPE_WIFI)

        .getState () ;

    return state ;

}

//设置配置好的网络（有密码的网络并配置好了密码），指定的

public void setWifiConfigedSpecifiedList (String ssid) {

    wifiConfigedSpecifiedList.clear () ;

    if (wifiConfigList !=null) {

        for (WifiConfiguration item:wifiConfigList) {

            //如果是指定的网络，就加入列表
```

```
if (item.SSID.equalsIgnoreCase ("\""+ssid+"\") )
```

```
& & item.preSharedKey !=null) {
```

```
//添加到列表中
```

```
wifiConfigedSpecifiedList.add (item) ;
```

```
}
```

```
}
```

```
}
```

```
}
```

```
//返回Wi-Fi设置列表
```

```
public List<WifiConfiguration>getWifiConfigedSpecifiedList () {
```

```
return wifiConfigedSpecifiedList ;
```

```
}
```

```
//打开Wi-Fi网卡
```

```
public void openNetCard () {
```

```
if ( ! mWifiManager.isWifiEnabled () ) {
```

```
//设置Wi-Fi可用
```

```
mWifiManager.setWifiEnabled (true) ;
```

```
}
```

```
}
```

```
//关闭Wi-Fi网卡
```

```
public void closeNetCard () {
```

```
if (mWifiManager.isWifiEnabled () ) {
```

```
//设置Wi-Fi不可用
```

```
mWifiManager.setWifiEnabled (false) ;
```

```
}
```

```
}
```

```
//检查当前Wi-Fi网卡状态
```

```
public void checkNetCardState () {
```

```
if (mWifiManager.getWifiState () ==0) {  
  
    Log.i (TAG, "网卡正在关闭") ;  
  
}else if (mWifiManager.getWifiState () ==1) {  
  
    Log.i (TAG, "网卡已经关闭") ;  
  
}else if (mWifiManager.getWifiState () ==2) {  
  
    Log.i (TAG, "网卡正在打开") ;  
  
}else if (mWifiManager.getWifiState () ==3) {  
  
    Log.i (TAG, "网卡已经打开") ;  
  
}  
  
}  
  
//扫描周边网络  
  
public void scan () {
```

```
//开始扫描  
  
mWifiManager.startScan () ;  
  
//获取扫描结果  
  
listResult=mWifiManager.getScanResults () ;  
  
//扫描配置列表  
  
wifiConfigList=mWifiManager.getConfiguredNetworks () ;  
  
if (listResult !=null) {  
  
    Log.i (TAG, "当前区域存在无线网络, 请查看扫描结果") ;  
  
}else{  
  
    Log.i (TAG, "当前区域没有无线网络") ;  
  
}  
  
}  
  
//返回扫描结果  
  
public List<ScanResult>getListResult () {
```

```
return listResult ;
```

```
}
```

```
//得到扫描结果
```

```
public String getScanResult () {
```

```
//每次点击扫描之前清空上一次的扫描结果
```

```
if (mStringBuffer !=null) {
```

```
mStringBuffer=new StringBuffer () ;
```

```
}
```

```
//开始扫描网络
```

```
scan () ;
```

```
//获取扫描结果，保存到列表中
```

```
listResult=mWifiManager.getScanResults () ;
```

```
//将扫描结果列表打印出来
```

```
if (listResult !=null) {
```

```
for (int i=0 ; i<listResult.size () ; i++) {  
  
    mScanResult=listResult.get (i) ;  
  
    //将需要的属性连接到一个字符串里面  
  
    mStringBuffer=  
  
        mStringBuffer.append ("NO.") .append (i+1)  
  
        .append (" : ") .append (mScanResult.SSID) .append ("->")  
  
        .append (mScanResult.BSSID) .append ("->")  
  
        .append (mScanResult.capabilities) .append ("->")  
  
        .append (mScanResult.frequency) .append ("->")  
  
        .append (mScanResult.level) .append ("->")  
  
        .append (mScanResult.describeContents () )  
  
        .append ("\n\n") ;  
  
    }  
}
```

```
Log.i (TAG,mStringBuffer.toString () ) ;
```

```
//返回其结果
```

```
return mStringBuffer.toString () ;
```

```
}
```

```
//连接指定网络
```

```
public void connect () {
```

```
//获取连接信息
```

```
mWifiInfo=mWifiManager.getConnectionInfo () ;
```

```
}
```

```
//断开当前连接的网络
```

```
public void disconnectWifi () {
```

```
//获取网络ID
```

```
int netId=getNetworkId () ;
```

```
//设置网络不可用
```

```
mWifiManager.disableNetwork (netId) ;
```

```
//断开网络
```

```
mWifiManager.disconnect () ;
```

```
//设置Wi-Fi信息为null
```

```
mWifiInfo=null ;
```

```
}
```

```
//检查当前网络状态
```

```
public Boolean checkNetWorkState () {
```

```
if (mWifiInfo !=null) {
```

```
//网络正常工作
```

```
return true ;
```

```
}else{
```

```
//网络已断开
```

```
return false ;
```

```
}
```

```
}
```

//得到连接的ID

```
public int getNetworkId () {
```

```
    return (mWifiInfo==null) ?0:mWifiInfo.getNetworkId () ;
```

```
}
```

//得到IP地址

```
public int getIPAddress () {
```

```
    return (mWifiInfo==null) ?0:mWifiInfo.getIpAddress () ;
```

```
}
```

//锁定WifiLock

```
public void acquireWifiLock () {
```

```
    mWifiLock.acquire () ;
```

```
}
```

```
//解锁WifiLock

public void releaseWifiLock () {
    //判断是否锁定
    if (mWifiLock.isHeld ()) {
        mWifiLock.acquire () ;
    }
}

//创建一个WifiLock

public void creatWifiLock () {
    mWifiLock=mWifiManager.createWifiLock ("Test") ;
}

//得到配置好的网络wpa_supplicant.conf中的内容，不管有没有配置密码

public List<WifiConfiguration>getConfiguration () {
    return wifiConfigList ;
}
```

```
}
```

```
//指定配置好的网络进行连接
```

```
public Boolean connectConfiguration (int index) {
```

```
//索引大于配置好的网络索引返回
```

```
if (index>=wifiConfigList.size () ) {
```

```
return false ;
```

```
}
```

```
//连接配置好的指定ID的网络
```

```
return mWifiManager.enableNetwork (
```

```
wifiConfigedSpecifiedList.get (index) .networkId,true) ;
```

```
}
```

```
//得到MAC地址
```

```
public String getMacAddress () {
```

```
return (mWifiInfo==null) ?"" : mWifiInfo.getMacAddress () ;
```

```
}
```

//得到接入点的BSSID

```
public String getBSSID () {
```

```
    return (mWifiInfo==null) ?"NULL" : mWifiInfo.getBSSID () ;
```

```
}
```

//得到WifiInfo的所有信息包

```
public String getWifiInfo () {
```

```
    return (mWifiInfo==null) ?"NULL" : mWifiInfo.toString () ;
```

```
}
```

//添加一个网络并连接

```
public int addNetwork (WifiConfiguration wcg) {
```

//添加网络

```
    int wcgID=mWifiManager.addNetwork (wcg) ;
```

//设置添加的网络可用

```
mWifiManager.enableNetwork (wcgID,true) ;  
  
//返回添加网络的ID  
  
return wcgID ;  
  
}  
  
}
```

5.3.3 实战案例：使用Wi-Fi直连方式传输文件

首先要清楚什么是Wi-Fi直连。Wi-Fi直连技术允许已经配备了相应硬件并预装了Android 4.0（API level 14）及以上操作系统的设备，在不需要Wi-Fi中间热点的支持下通过Wi-Fi直接互联的技术。Wi-Fi直连让终端之间摆脱了线缆的束缚，并且可以获得高达100Mbps的数据传输速率。

Wi-Fi直连技术的API包含以下主要部分：

- 提供允许用户发现、请求然后连接对等设备的各种方法。这些方法被定义在WifiP2pManager类中。
- 提供允许用户定义收到调用WifiP2pManager类中方法成功或失败的通知监听器。当用户调用WifiP2pManager类中的方法时，每一个方法都可以收到一个以参数形式传过来的特定监听。

■提供通知用户被Wi-Fi直连技术框架检测到的特定事件的意图，比如一个已丢掉的连接或者一个新的对等设备的发现等。

经常会同时使用这3类主要组件的相关功能。例如，可以为调用discoverPeers () 方法提供一个WifiP2pManager.ActionListener的监听器，这样以后可以收到一个ActionListener.onSuccess () 或者 ActionListener.onFailure () 方法的通知。当调用discoverPeers () 方法发现的对等设备列表发生改变时，

WIFI_P2P_PEERS_CHANGED_ACTION意图会发送一个广播。

1.Wi-Fi直连相关的API

WifiP2pManager类提供了很多方法允许用户通过设备的Wi-Fi模块来进行交互，比如做一些如发现、连接其他对等设备的事情。该类提供的方法如表5-10所示。

表 5-10 Wi-Fi 直连方法

方 法 名	详 细 描 述
initialize()	通过 Wi-Fi 框架对应用来进行注册。这个方法必须在任何其他 Wi-Fi 直连方法使用之前调用
connect()	开始一个拥有特定配置的设备的点对点连接
cancelConnect()	取消任何一个正在进行的点对点组的连接
requestConnectInfo()	获取一个设备的连接信息
createGroup()	以当前设备为组拥有者来创建一个点对点连接组
removeGroup()	移除当前的点对点连接组
requestGroupInfo()	获取点对点连接组的信息
discoverPeers()	初始化对等设备的发现
requestPeers()	获取当前发现的对等设备列表

WifiP2pManager的方法可以在一个监听器里传递参数，这样Wi-Fi直连框架就可以通知给activity这个方法调用的状态。可以被使用的监听器接口和WifiP2pManager类中对应方法的关系如表5-11所示。

表 5-11 Wi-Fi 直连监听器

监听器接口	相关联的方法
WifiP2pManager.ActionListener	connect(), cancelConnect(), createGroup(), removeGroup(), and discoverPeers()
WifiP2pManager.ChannelListener	initialize()
WifiP2pManager.ConnectionInfoListener	requestConnectInfo()
WifiP2pManager.GroupInfoListener	requestGroupInfo()
WifiP2pManager.PeerListListener	requestPeers()

Wi-Fi直连技术的API定义了一些当特定的Wi-Fi直连事件发生时（比如当一个新的对等设备被发现或者一个设备的Wi-Fi状态发生改变等）广播的意图。可以在应用里通过创建一个处理这些意图的广播接收器来注册和接收这些意图。Android支持的Wi-Fi直连意图如表5-12所示。

表 5-12 Wi-Fi 直连意图

意 图 名 称	详 细 描 述
WIFI_P2P_CONNECTION_CHANGED_ACTION	当设备的 Wi-Fi 连接信息状态改变时候进行广播
WIFI_P2P_PEERS_CHANGED_ACTION	当调用 discoverPeers() 方法的时候进行广播。在应用里处理此意图时，通常会调用 requestPeers() 获得对等设备列表的更新
WIFI_P2P_STATE_CHANGED_ACTION	当设备的 Wi-Fi 直连功能打开或关闭时进行广播
WIFI_P2P_THIS_DEVICE_CHANGED_ACTION	当设备的详细信息改变的时候进行广播，比如设备的名称

2. 创建一个接收Wi-Fi直连意图的广播接收器

一个广播接收器允许接收由Android系统发布的意图广播，这样应用就可以对那些感兴趣的事件作出响应。创建一个处理Wi-Fi直连意图的广播接收器的基本步骤如下：

步骤1 创建一个继承自BroadcastReceiver的类。对于类的构造，一般最常用的就是以WifiP2pManager、WifiP2pManager.Channel作为参数，同时这个广播接收器对应的窗体也将被注册进来。这个广播接收器可以向窗体发送更新，或者在需要的时候可以访问Wi-Fi硬件或通信通道。

步骤2 在广播接收器里处理onReceive () 方法里感兴趣的意图。执行接收到的意图任何需要的动作。比如，广播接收器接收到一个WIFI_P2P_PEERS_CHANGED_ACTION的意图，就要调用requestPeers () 方法去获得当前发现的对等设备列表。

下面的代码展示了怎样去创建一个典型的广播接收器。

//处理Wi-Fi事件的广播接收器

```
public class WiFiDirectBroadcastReceiver extends BroadcastReceiver{
```

//创建WifiP2pManager对象

```
private WifiP2pManager manager ;
```

//创建Channel对象

```
private Channel channel ;  
  
//创建MyWiFiActivity对象  
  
private MyWiFiActivity activity ;  
  
//构造函数  
  
public WiFiDirectBroadcastReceiver (WifiP2pManager manager,Channel  
channel,MyWiFiActivity activity) {  
  
super () ;  
  
this.manager=manager ;  
  
this.channel=channel ;  
  
this.activity=activity ;  
  
}  
  
@Override  
  
public void onReceive (Context context,Intent intent) {  
  
String action=intent.getAction () ;  
  
//当设备的Wi-Fi直连功能打开或关闭时进行广播
```

```
if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals
    (action) ) {

} else if

//当调用discoverPeers () 方法的时候进行广播

//可以收到该消息，通常会调用requestPeers () 获得设备列表的更新

(WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals
    (action) ) {

} else if

//当设备的Wi-Fi连接信息状态改变时候进行广播

(WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION.equals
    (action) ) {

} else if

//当设备的详细信息改变时进行广播，比如设备的名称

(WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION.equals
    (action) ) {

}
```

}

}

3.Wi-Fi直连的具体应用

创建一个Wi-Fi直连的应用包括创建和注册一个广播接收器、发现对等设备、连接对等设备，然后传输数据等步骤。接下来的几个部分描述了怎么去做这些工作。

(1) 初始化设置

在使用Wi-Fi直连的API之前，必须确保应用可以访问设备的硬件，并且设备要支持Wi-Fi直连的通信协议。如果设备支持Wi-Fi直连技术，可以创建一个WifiP2pManager的实例对象，然后创建并注册广播接收器，其后就可以使用Wi-Fi直连的API方法。

1) 为设备的Wi-Fi硬件获取权限并在Android manifest文件中声明应用正确使用的最低SDK版本。

<!--声明应用使用的最低SDK版本-->

<uses-sdk android:minSdkVersion="14"/>

<!--允许访问Wi-Fi网络状态信息-->

```
<uses-permission  
    android:name="android.permission.ACCESS_WIFI_STATE"/>
```

```
<!--允许改变Wi-Fi连接状态-->
```

```
<uses-permission  
    android:name="android.permission.CHANGE_WIFI_STATE"/>
```

```
<!--允许改变网络连接状态-->
```

```
<uses-permission  
    android:name="android.permission.CHANGE_NETWORK_STATE"/>
```

```
<!--允许打开网络套接字-->
```

```
<uses-permission android:name="android.permission.INTERNET"/>
```

```
<!--允许访问有关GSM网络信息-->
```

```
<uses-permission  
    android:name="android.permission.ACCESS_NETWORK_STATE"/>
```

2) 检查设备是否支持Wi-Fi直连技术。一种好的解决办法是当广播接收器接收到一个WIFI_P2P_STATE_CHANGED_ACTION意图，通知窗体Wi-Fi直连的状态和相应的反应。

```
//定义广播接收器

@Override

public void onReceive (Context context,Intent intent) {

    .....

    //获取意图权限字符串

    String action=intent.getAction () ;

    //Wi-Fi P2P网络是否已经起用

    if (WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION.equals
        (action) ) {

        //查看Wi-Fi P2P网络的状态值

        int state=intent.getIntExtra
            (WifiP2pManager.EXTRA_WIFI_STATE, -1) ;

        //查看Wi-Fi直连状态

        if (state==WifiP2pManager.WIFI_P2P_STATE_ENABLED) {

            //Wi-Fi直连enabled
```

```
 }else{  
  
//Wi-Fi直连not enabled  
  
}  
  
}  
  
.....  
  
}
```

3) 在窗体的onCreate () 方法里，获得一个WifiP2pManager的实例并调用initialize () 方法通过Wi-Fi直连框架去注册应用。这个方法返回一个WifiP2pManager.Channel对象，是被用来连接应用和Wi-Fi直连框架的。应该再创建一个以WifiP2pManager和WifiP2pManager.Channel为参数且关联窗体的广播接收器的实例。这样广播接收器就可以接收到感兴趣的事件通知窗体并更新它。它还可以在需要的时候操作设备的Wi-Fi状态。

```
//创建WifiP2pManager对象  
  
WifiP2pManager mManager ;  
  
//创建Channel对象
```

```
Channel mChannel ;  
  
//创建BroadcastReceiver对象  
  
BroadcastReceiver mReceiver ;  
  
.....  
  
@Override  
  
protected void onCreate (Bundle savedInstanceState) {  
  
.....  
  
//获取WifiP2pManager实例  
  
mManager= (WifiP2pManager) getSystemService  
(Context.WIFI_P2P_SERVICE) ;  
  
//初始化WifiP2pManager  
  
mChannel=mManager.initialize (this,getMainLooper () , null) ;  
  
//获取BroadcastReceiver实例  
  
mReceiver=new WiFiDirectBroadcastReceiver (manager,channel,this) ;  
  
.....
```

}

4) 创建一个意图过滤器并添加广播接收器需要处理的意图。

```
IntentFilter mIntentFilter ;
```

.....

```
@Override
```

```
protected void onCreate (Bundle savedInstanceState) {
```

.....

```
mIntentFilter=new IntentFilter () ;
```

```
mIntentFilter.addAction (
```

```
//Wi-Fi P2P状态改变
```

```
WifiP2pManager.WIFI_P2P_STATE_CHANGED_ACTION) ;
```

```
mIntentFilter.addAction (
```

```
//Wi-Fi P2P设备的列表已经改变
```

```
WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION) ;
```

```
mIntentFilter.addAction ( //Wi-Fi P2P连接状态发生了改变  
    WifiP2pManager.WIFI_P2P_CONNECTION_CHANGED_ACTION) ;  
  
mIntentFilter.addAction ( //Wi-Fi P2P设备的细节已经改变  
    WifiP2pManager.WIFI_P2P_THIS_DEVICE_CHANGED_ACTION) ;  
  
.....  
}
```

5) 在窗体中使用onResume () 方法调用注册广播接收器，调用 onPause () 方法解除注册。

```
@Override  
protected void onResume () {  
    super.onResume () ;  
    //在onResume () 方法中注册广播接收器  
    registerReceiver (mReceiver,mIntentFilter) ;
```

```
}

@Override

protected void onPause () {

super.onPause () ;

//解除广播注册

unregisterReceiver (mReceiver) ;

}
```

当获取到一个WifiP2pManager.Channel对象并且设置好广播接收器时，应用就可以调用Wi-Fi直连的方法，并且可以接收Wi-Fi直连的意图。

(2) 发现对等设备

要发现可以连接的对等设备，需要调用discoverPeers () 方法去检测在范围内的可使用设备。这个方法的调用是异步的，如果创建了一个WifiP2pManager.ActionListener监听器的话，会通过onSuccess () 或者 onFailure () 方法收到发现成功或失败的消息。onSuccess () 方法只能通知是否连接成功而不能提供任何关于所发现的设备的信息。

//发现对等设备

```
manager.discoverPeers (channel,new WifiP2pManager.ActionListener ()  
{  
  
    @Override  
  
    public void onSuccess () {  
  
        //发现成功时候的处理  
  
        .....  
  
    }  
  
    @Override  
  
    public void onFailure (int reasonCode) {  
  
        //发现失败时候的处理  
  
        .....  
  
    }  
}) ;
```

如果成功检测到了对等设备，系统将会广播一个
WIFI_P2P_PEERS_CHANGED_ACTION意图，当应用接收到这个意图

后，就可以调用requestPeers () 方法来获取发现设备的列表。代码如下：

```
//新建一个PeerListListener监听器  
PeerListListener myPeerListListener ;  
.....  
if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals  
(action) ) {  
  
//获取发现设备的列表  
  
//其回调函数为PeerListListener.onPeersAvailable ()  
if (manager !=null) {  
  
manager.requestPeers (channel,myPeerListListener) ;  
  
}  
}
```

通过调用onPeersAvailable () 方法会获得可连接的对等设备的列表 WifiP2pDeviceList，随后就可以选择想连接的对等设备了。

(3) 连接到设备

在获取可连接的对等设备的列表后，调用connect（）方法连接指定设备。这个方法的调用需要一个包含待连接设备信息的WifiP2pConfig对象。可以通过WifiP2pManager.ActionListener接收到连接是否成功的通知。下面的代码展示了怎样去连接一个想连接的对等设备。

```
//从对等设备列表WifiP2pDeviceList中获取对等设备  
  
//创建一个WifiP2pDevice类  
  
WifiP2pDevice device ;  
  
//创建WifiP2pConfig对象  
  
WifiP2pConfig config=new WifiP2pConfig () ;  
  
config.deviceAddress=device.deviceAddress ;  
  
//连接到指定的设备  
  
manager.connect (channel,config,new ActionListener () {  
  
    @Override  
  
    public void onSuccess () {
```

```
//成功连接  
}  
  
@Override  
  
public void onFailure (int reason) {  
  
//连接失败  
}  
  
}) ;
```

(4) 数据传输

一旦连接已经建立，就可以通过套接字来进行数据的传输了。基本的数据传输步骤如下：

步骤1 创建一个ServerSocket对象。这个服务端套接字对象等待一个来自指定地址和端口的客户端的连接且阻塞线程直到连接发生，所以把它建立在一个后台线程里。

步骤2 创建一个客户端Socket。这个客户端套接字对象使用指定IP地址和端口连接服务端设备。

步骤3 从客户端给服务端发送数据。当客户端成功连接服务端设备后，可以通过字节流从客户端给服务端发送数据。

步骤4 服务端等待客户端的连接（使用accept（）方法）。这个调用阻塞服务端线程直到客户端连接上，所以要在一个新的线程中调用。当连接建立时，服务端可以接受来自客户端的数据，执行关于数据的任何动作，比如保存数据或者展示给用户。

下面的代码，展示了怎样创建服务端和客户端的连接和通信，并且使用一个客户端到服务端的服务传输一张JPEG图像。

```
public static class FileServerAsyncTask extends AsyncTask{  
    private Context context ;  
  
    private TextView statusText ;  
  
    public FileServerAsyncTask (Context context,View statusText) {  
        this.context=context ;  
  
        this.statusText= (TextView) statusText ;  
  
    }  
  
    @Override
```

```
protected String doInBackground (Void.....params) {  
  
try{  
  
//创建Socket服务器，等待客户端访问  
  
ServerSocket serverSocket=new ServerSocket (8888) ;  
  
Socket client=serverSocket.accept () ;  
  
//客户端将一个JPEG文件作为数据传送过来  
  
final File f=new File (Environment.getExternalStorageDirectory () +"/"  
+context.getPackageName () +"/wifip2pshared-"+  
  
System.currentTimeMillis () +".jpg") ;  
  
File dirs=new File (f.getParent () ) ;  
  
if ( ! dirs.exists () )  
  
dirs.mkdirs () ;  
  
f.createNewFile () ;  
  
InputStream inputstream=client.getInputStream () ;
```

```
copyFile (inputstream,new FileOutputStream (f) ) ;  
  
serverSocket.close () ;  
  
return f.getAbsolutePath () ;  
  
}catch (IOException e) {  
  
Log.e (WiFiDirectActivity.TAG,e.getMessage () ) ;  
  
return null ;  
  
}  
  
}
```

//处理JPEG图像文件

@Override

```
protected void onPostExecute (String result) {  
  
if (result !=null) {  
  
statusText.setText ("File copied-"+result) ;  
  
Intent intent=new Intent () ;
```

```
        intent.setAction (android.content.Intent.ACTION_VIEW) ;  
  
        intent.setDataAndType (Uri.parse ("file://" +result) , "image/*") ;  
  
        context.startActivity (intent) ;  
  
    }  
  
}  
  
}
```

在客户端，使用客户端套接字连接服务端套接字并传输数据。代码实现了从客户端的文件系统里传输了一张JPEG的图像到服务端。

```
Context context=this.getApplicationContext () ;  
  
String host ;  
  
int port ;  
  
int len ;  
  
Socket socket=new Socket () ;  
  
byte buf[]=new byte[1024] ;  
  
.....
```

```
try{  
  
    //创建Socket客户端  
  
    socket.bind (null) ;  
  
    socket.connect ( (new InetSocketAddress (host,port) ) , 500) ;  
  
    //创建发送数据流  
  
    OutputStream outputStream=socket.getOutputStream () ;  
  
    ContentResolver cr=context.getContentResolver () ;  
  
    InputStream inputStream=null ;  
  
    inputStream=cr.openInputStream (Uri.parse ("path/to/picture.jpg") ) ;  
  
    while ( (len=inputStream.read (buf) ) !=-1) {  
  
        outputStream.write (buf, 0, len) ;  
  
    }  
  
    outputStream.close () ;  
  
    inputStream.close () ;
```

```
 }catch (FileNotFoundException e) {
```

```
 }catch (IOException e) {
```

```
}
```

```
//最后清空Socket
```

```
finally{
```

```
 if (socket !=null) {
```

```
 if (socket.isConnected () ) {
```

```
 try{
```

```
 socket.close () ;
```

```
 }catch (IOException e) {
```

```
}
```

```
}
```

```
}
```

```
}
```

5.4 蓝牙编程

5.4.1 蓝牙简介

蓝牙是一种支持设备短距离通信（一般10米内）的无线数据通信技术，能在包括移动电话、PDA、无线耳机、笔记本电脑、相关外设等众多设备之间进行无线信息交换。Android平台支持蓝牙网络协议栈，可以实现蓝牙设备之间的无线通信。Android蓝牙开发是从Android 2.0版本的SDK才开始支持的，而且模拟器不支持，测试至少需要两部手机。

5.4.2 Android蓝牙API分析

Android支持蓝牙开发的类在android.bluetooth包下。编程主要涉及的类简介如下。

(1) BluetoothAdapter类

该类代表了一个本地的蓝牙适配器。它是所有蓝牙交互的入口点。利用它可以发现其他蓝牙设备，查询绑定了的设备，使用已知的MAC地址实例化一个蓝牙设备和建立一个BluetoothServerSocket（作为服务器端）来监听来自其他设备的连接。

该类提供的主要方法如表5-13所示。

表 5-13 BluetoothAdapter 类中部分方法

方 法	解 释
cancelDiscovery()	取消当前设备的搜索过程
checkBluetoothAddress(String address)	检查蓝牙地址字符串的有效性，如 0:43:A8:23:10:F0，字母必须大写才有效
disable()/enable()	关闭 / 打开本地蓝牙适配器
getAddress()	获取本地蓝牙硬件地址
getDefaultAdapter()	获取默认 BluetoothAdapter
getName()	获取本地蓝牙名称
getRemoteDevice(String address) getRemoteDevice(byte [] address)	根据特定蓝牙地址获取远程蓝牙设备
getState()	获取本地蓝牙适配器当前状态
isDiscovering()	判断当前是否正在查找设备
isEnabled()	判断蓝牙是否打开
listenUsingRfcommWithServiceRecord(String name, UUID uuid)	根据名称，UUID 创建并返回 BluetoothServerSocket
startDiscovery()	开始搜索

(2) BluetoothDevice类

该类代表了一个远端的蓝牙设备，使用它请求远端蓝牙设备连接或者获取远端蓝牙设备的名称、地址、种类和绑定状态（其信息封装在 BluetoothSocket 中）。

该类提供的主要方法如表5-14所示。

表 5-14 BluetoothDevice 类中部分方法

方 法	解 释
createRfcommSocketToServiceRecord(UUID uuid)	根据 UUID 创建并返回一个 BluetoothSocket
getAddress()	返回蓝牙设备的物理地址
getBondState()	返回远端设备的绑定状态
getName()	返回远端设备的蓝牙名称
getUuids()	返回远端设备的 UUID
toString()	返回代表该蓝牙设备的字符串

(3) BluetoothServerSocket类

该类代表打开服务连接来监听可能到来的连接请求（属于server端），为了连接两个蓝牙设备必须有一个设备作为服务器打开一个服务套接字。当远端设备发起连接请求的时候，并且已经连接到了的时候，BluetoothServerSocket类将会返回一个BluetoothSocket。

该类提供的主要方法如表5-15所示。

表 5-15 BluetoothServerSocket 类中部分方法

方 法	解 释
accept()	直到接收到了客户端的请求继而连接建立，否则会一直阻塞线程。因而一般应放在新线程里运行
accept(int timeout)	直到接收到了客户端的请求继而连接建立（或者超时），否则会一直阻塞线程
close()	关闭 Socket，释放所有相关资源

注意 accept方法返回一个BluetoothSocket，服务器端与客户端的连接最后是两个BluetoothSocket间的连接。

(4) BluetoothSocket类

该类代表客户端，跟BluetoothServerSocket相对。代表了一个蓝牙套接字的接口（类似于TCP中的套接字），它是应用程序通过输入、输出流与其他蓝牙设备通信的连接点。

该类提供的主要方法如表5-16所示。

表 5-16 BluetoothSocket 类中部分方法

方 法	解 释
close()	关闭 Socket，释放所有相关资源
connect()	允许连接远端设备
getInputStream()	获取输入流
getOutputStream()	获取输出流
getRemoteDevice()	获取跟这个 Socket 相连的远程设备
isConnected()	得到 Socket 连接状态，判断是否连接

蓝牙支持point-to-point和multipoint两种连接。利用Android Bluetooth API，可以做到：

- 设置本地和搜索其他蓝牙设备；
- 寻找网内匹配的蓝牙设备；
- 建立RFCOMM通道；
- 通过服务发现建立与其他蓝牙设备的连接；
- 设备之间的数据传输；
- 管理多个连接。

5.4.3 Android蓝牙基本操作

Android蓝牙基本操作的具体介绍如下。

1. 声明权限

为了在应用中使用蓝牙功能，要在AndroidManifest.xml中至少声明以下两个权限之一：BLUETOOTH权限和BLUETOOTH_ADMIN权限。

为了能执行任意蓝牙通信，例如请求连接、接收连接和传送数据，必须有BLUETOOTH权限。而启动设备，发现或做蓝牙设置必须要有BLUETOOTH_ADMIN权限。大多数应用程序都仅需要这个权限去发现当地的蓝牙设备。此权限赋予的其他权利不应该被使用，除非应用程序是一个“电源管理”，想根据用户要求去修改蓝牙设置。

注意 如果想有BLUETOOTH_ADMIN权限，必须先有BLUETOOTH权限。

在应用中manifest文件里声明蓝牙权限。示例如下：

```
<manifest.....>
```

```
<!--允许程序连接到已配对的蓝牙设备-->
```

```
<uses-permission android:name="android.permission.BLUETOOTH"/>
```

.....

</manifest>

2. 蓝牙设置

在应用通过蓝牙进行通信之前，需要确认设备是否支持蓝牙，如果支持，则还需要确认它被打开。

如果设备不支持，则不能使用任何蓝牙功能。如果设备支持蓝牙，但没有被打开，可以在应用中请求使用蓝牙。通过BluetoothAdapter打开蓝牙设备分以下两步完成。

(1) 获取BluetoothAdapter

所有的蓝牙activity都需要BluetoothAdapter。为了获取BluetoothAdapter对象，要调用静态的getDefaultAdapter () 方法。这个方法会返回一个BluetoothAdapter对象，代表设备自己的蓝牙适配器。整个系统中只有一个蓝牙适配器，应用可以通过这个对象与它交互。如果getDefaultAdapter () 方法返回null，则表示这个设备不支持蓝牙功能。例如：

```
//获取BluetoothAdapter对象
```

```
BluetoothAdapter mBluetoothAdapter=BluetoothAdapter.getDefaultAdapter()
() ;

if (mBluetoothAdapter==null) {

//设备不支持蓝牙

}
```

(2) 打开蓝牙功能

接着，需要确认蓝牙是否可用。通过调用isEnabled () 方法来检查蓝牙当前是否可用。如果这个方法返回false，则蓝牙不可用。此时，为了打开蓝牙功能，要以ACTION_REQUEST_ENABLE动作意图作为参数调用startActivityForResult () 方法，它将发出一个启用蓝牙的请求。

例如：

```
//检查当前蓝牙是否可用

if ( ! mBluetoothAdapter.isEnabled () ) {

Intent enableBtIntent=new Intent
(BluetoothAdapter.ACTION_REQUEST_ENABLE) ;

startActivityForResult (enableBtIntent,REQUEST_ENABLE_BT) ;
```

}

此时会弹出一个请求使用蓝牙权限的对话框，如图5-12所示。如果用户单击Yes按钮，系统将启动蓝牙功能，一旦这个进程完成（或失败），焦点将重新回到应用。

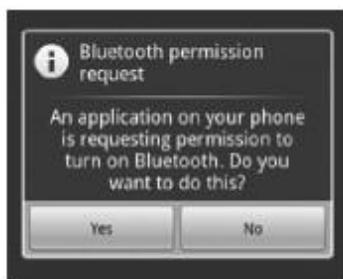


图 5-12 启用蓝牙功能对话框

代码中，方法startActivityForResult（）中的参数

REQUEST_ENABLE_BT是一个局部整型常量，值必须大于0，系统将在onActivityResult（）实现中作为requestCode参数返回。

如果启动蓝牙成功，onActivityResult（）中返回RESULT_OK，如果由于错误（比如用户按了No按钮）蓝牙不能启动，则返回RESULT_CANCELED。

此外，还可以通过监听ACTION_STATE_CHANGED这个广播意图来知道蓝牙状态是否改变。这个Intent包含EXTRA_STATE、EXTRA_PREVIOUS_STATE两个字段，分别代表新旧状态。字段可能

的取值是STATE_TURNING_ON、STATE_ON、STATE_TURNING_OFF以及STATE_OFF。

注意 后面会讲到允许搜索（Enabling discoverability），此时系统将自动启用蓝牙。如果在执行蓝牙操作前，始终使设备可发现，则可以跳过“打开蓝牙功能”这个步骤。

3.发现设备（Finding Devices）

使用BluetoothAdapter，可以通过设备搜索（device discovery）或查询匹配设备（Querying paired devices）找到远端蓝牙设备。

设备搜索是一个搜索本地已开启的蓝牙设备并且从搜索到的设备请求一些信息的过程。但是，搜索到的本地Bluetooth设备只有在打开被搜索功能后才会响应一个discovery请求，响应的信息包括设备名、类、唯一的MAC地址。发起搜寻的设备可以使用这些信息来初始化与被发现的设备的连接。

一旦与远程设备的第一次连接被建立，一个配对请求就会自动提交给用户。如果设备已配对，配对设备的基本信息（设备名称、类、MAC地址）就会被保存下来，能够使用蓝牙API来读取这些信息。使用已知的远程设备的MAC地址，可以在任何时候发起连接而不必再做设备搜索（当然这是假设远程设备是在可连接的空间范围内）。

注意 配对和连接是两个不同的概念。配对意思是两个设备相互意识到对方的存在，共享一个用来鉴别身份的链路键（link-key），能够与对方建立一个加密的连接。连接意思是当前两个设备共享一个RFCOMM信道，能够相互传输数据。目前Android蓝牙API要求设备在建立RFCOMM信道前必须先配对（配对是在使用Bluetooth API建立一个加密连接时自动完成的）。

(1) 查询配对设备 (Querying paired devices)

在搜索设备前，最好先查询一下配对设备集，看需要的设备是否已经存在，可以调用getBondedDevices () 实现。该函数会返回一个已配对的BluetoothDevice集合。例如，可以使用ArrayAdapter查询所有配对的设备，然后显示所有设备名给用户。

//获取已经配对的蓝牙设备集合

```
Set<BluetoothDevice>  
pairedDevices=mBluetoothAdapter.getBondedDevices () ;
```

//判断需要的设备是否存在

```
if (pairedDevices.size () >0) {  
  
for (BluetoothDevice device:pairedDevices) {
```

```
//将设备名字和设备地址加入一个ListView中

mArrayAdapter.add (device.getName () +"\n"+device.getAddress
() ) ;

}

};


```

BluetoothDevice对象中用来初始化一个连接，唯一需要用到的信息就是MAC地址。这个例子中，MAC地址作为ArrayAdapter的一部分显示给用户。后面用户可以提取MAC地址以建立连接，相关知识参看5.4.4节。

(2) 搜索设备 (Discovering devices)

要开始搜索设备，只需简单地调用startDiscovery () 即可。该过程是异步的，调用后会立即返回一个表示搜索是否成功的布尔值。搜索过程通常需要12秒，接着一个页面会显示搜索到的所有蓝牙设备名称。

应用中可以注册一个带ACTION_FOUND意图的广播接收器，以便接收搜索到的设备消息。对于每一个设备，系统都会广播ACTION_FOUND意图，该意图包含字段信息EXTRA_DEVICE和EXTRA_CLASS。

下面的示例显示了如何注册处理设备被搜索后发出的广播。

```
//创建一个广播接收器

private final BroadcastReceiver mReceiver=new BroadcastReceiver () {

    public void onReceive (Context context,Intent intent) {

        String action=intent.getAction () ;

        //发现设备

        if (BluetoothDevice.ACTION_FOUND.equals (action) ) {

            //获取蓝牙设备对象

            BluetoothDevice device=

            intent.getParcelableExtra (BluetoothDevice.EXTRA_DEVICE) ;

            //将设备名字和设备地址加入一个ListView中

            mArrayAdapter.add (device.getName () +"\n"+device.getAddress
                () ) ;

        }

    }

};
```

```
//注册BroadcastReceiver  
  
IntentFilter filter=new IntentFilter  
    (BluetoothDevice.ACTION_FOUND) ;  
  
registerReceiver (mReceiver,filter) ;
```

注意 设备搜索是相当耗费系统资源的，一旦已经找到要与之连接的某个设备，在连接之前要确认已经使用cancelDiscovery () 停止设备搜索。

(3) 允许搜索 (Enabling discoverability)

Android设备默认是不能被搜索的。如果想让本地设备被其他设备搜索到，可以以意图ACTION_REQUEST_DISCOVERABLE为参数去调用startActivityForResult (Intent,int) 方法。该方法会提交一个允许搜索的请求。默认情况下，设备在120秒内是可搜索的。可以通过EXTRA_DISCOVERABLE_DURATION自定义一个间隔时间值，最大值是3600秒，0表示设备总是可以被搜索的（小于0或者大于3600则会被自动设置为120秒）。下面示例设置时间间隔为300秒。

```
//创建允许搜索的意图
```

```
Intent discoverableIntent=new Intent  
    (BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE) ;
```

```
//设置时间间隔为300秒
```

```
discoverableIntent.putExtra  
(BluetoothAdapter.EXTRA_DISCOVERABLE_DURATION, 300) ;  
  
startActivity (discoverableIntent) ;
```

询问用户是否允许打开“设备可以被搜索”功能时会显示一个对话框，如图5-13所示。如果用户单击Yes按钮，设备会在指定时间过后变为可以被搜索的。Activity会接收到onActivityResult () 回调函数的调用结果，结果码等于设备变为可以被搜索所需时长。如果用户单击No按钮或者有错误发生，结果码会是RESULT_CANCELED。



图 5-13 允许搜索对话框

注意 启动蓝牙被搜索功能时，如果蓝牙还没有启用，则会自动开启蓝牙功能。

在规定的时间内，设备会静静地保持可以被搜索模式。如果想在可被搜索模式发生更改时收到通知，可以用ACTION_SCAN_MODE_CHANGED意图注册一个广播接收器，包含额外的字段信息EXTRA_SCAN_MODE和EXTRA_PREVIOUS_SCAN_MODE，这两个字段信息分别表示新旧扫描模式，其可能的值为SCAN_MODE_CONNECTABLE_DISCOVERABLE（可被搜索模式）、SCAN_MODE_CONNECTABLE（非可被搜索模式但可以接受连接）和SCAN_MODE_NONE（非可被搜索模式同时也无法接受连接）。

如果只需要连接远程设备就不需要打开设备的可被搜索功能，只在应用作为一个服务器Socket的宿主用来接收进来的连接时才需要使能可以被搜索功能，因为远程设备在初始化连接前必须先搜索设备。

5.4.4 实战案例：蓝牙连接

为了在两台设备间创建一个连接，必须实现服务器端和客户端的机制，因为一个设备必须打开一个Server Socket，而另一个必须发起连接（使用服务器端设备的MAC地址发起连接）。当服务器端和客户端在同一个RFCOMM信道上都有一个BluetoothSocket时，则两端就建立了连接。此刻，每个设备都能获得一个输入输出流，进行数据传输。下面介绍如何在两个设备之间建立连接。

服务器端和客户端获得BluetoothSocket的方法是不同的，服务器端是在客户端的连接被接受时才产生一个BluetoothSocket，客户端是在打开一个到服务器端的RFCOMM信道时获得BluetoothSocket的。

蓝牙连接的一种实现技术是，每一个设备都自动准备作为一个服务器，所以每个设备都有一个Server Socket并监听连接。然后每个设备都能作为客户端建立一个到另一台设备的连接。另外一种代替方法是，一个设备按需打开一个Server Socket，另外一个设备仅作为客户端建立与这个设备的连接。如果两个设备在建立连接之前并没有配对，那么在建立连接的过程中，Android系统将自动显示一个请求配对的对话框，如图5-14所示。所以，在尝试连接设备时，应用程序无须确保设备之间已经进行了配对。RFCOMM连接将会在用户确认配对之后继续进行，或者因用户拒绝、超时等而失败。



图 5-14 蓝牙配对对话框

1.作为服务器连接

如果要连接两个设备，其中一个必须充当服务器，它拥有 BluetoothServerSocket。服务器Socket的作用是侦听进来的连接，且在一个连接被接受时返回一个BluetoothSocket对象。从 BluetoothServerSocket获取到BluetoothSocket对象之后，BluetoothServerSocket就可以（也应该）丢弃了，除非还要用它来接收更多的连接。

下面是建立服务器Socket和接受一个连接的基本步骤：

步骤1 通过调用listenUsingRfcommWithServiceRecord (String,UUID) 方法得到一个BluetoothServerSocket对象。字符串参数为服务的标识名称，名字是任意的，可以简单地是应用程序的名称。当客户端试图连接本设备时，它将携带一个UUID用来唯一标识它要连接的服务，UUID必须匹配，连接才会被接受。

步骤2 通过调用accept () 来侦听连接请求。这是一个阻塞线程，直到接受一个连接或者产生异常才会返回。当客户端携带的UUID与侦听它Socket注册的UUID匹配，连接请求才会被接受。如果成功，accept () 将返回一个BluetoothSocket对象。

步骤3 除非需要再接受另外的连接，否则的话调用close () 。close () 释放Server Socket及其资源，但不会关闭accept () 返回的BluetoothSocket对象。与TCP/IP不同，RFCOMM同一时刻一个信道只

允许一个客户端连接，因此大多数情况下意味着在 BluetoothServerSocket 接受一个连接请求后应该立即调用 close ()。

accept () 调用不应该在主 Activity UI 线程中进行，因为它是个阻塞线程，会妨碍应用中其他的交互。通常在一个新线程中做 BluetoothServerSocket 或 BluetoothSocket 的所有工作来避免线程阻塞。如果需要放弃阻塞线程，可以调用 close () 方法。

下面是一个服务器组件接受连接的线程示例。

```
//定义接受线程  
  
private class AcceptThread extends Thread{  
  
    //创建BluetoothServerSocket类  
  
    private final BluetoothServerSocket mmServerSocket ;  
  
    public AcceptThread () {  
  
        BluetoothServerSocket tmp=null ;  
  
        try{  
  
            //MY_UUID是应用的UUID标识
```

```
tmp=mBluetoothAdapter.listenUsingRfcommWithServiceRecord  
    (NAME,MY_UUID) ;  
  
}catch (IOException e) {}  
  
mmServerSocket=tmp ;  
  
}  
  
//线程启动时候运行  
  
public void run () {  
  
    BluetoothSocket socket=null ;  
  
    //保持侦听  
  
    while (true) {  
  
        try{  
  
            //接受  
  
            socket=mmServerSocket.accept () ;  
  
        }catch (IOException e) {  
  
            break ;
```

```
}

//连接被接受

if (socket !=null) {

//管理连接

manageConnectedSocket (socket) ;

//关闭连接

mmServerSocket.close () ;

break ;

}

}

}

//关闭连接

public void cancel () {

try{
```

```
//关闭BluetoothServerSocket
```

```
mmServerSocket.close () ;
```

```
}catch (IOException e) {}
```

```
}
```

```
}
```

本例中，只接受一个进来的连接，一旦连接被接受并获取BluetoothSocket，应用就发送获取到的BluetoothSocket给一个单独的线程，然后关闭BluetoothServerSocket并跳出循环。

注意 accept () 返回BluetoothSocket后，Socket就建立了连接，所以在客户端就不应该再调用connect () 。

2.作为客户端连接

为了实现与远程服务器设备的连接，必须首先获得一个代表远程设备BluetoothDevice的对象。然后使用BluetoothDevice对象来获取一个BluetoothSocket以实现连接。

基本步骤如下：

步骤1 使用BluetoothDevice调用方法

createRfcommSocketToServiceRecord (UUID) 获取一个BluetoothSocket 对象。

步骤2 调用connect () 建立连接。当调用这个方法的时候，系统会在远程设备上完成一个SDP查找来匹配UUID。如果查找成功并且远程设备接受连接，就共享RFCOMM信道，connect () 会返回。这个方法也是一个阻塞的调用。如果连接失败或者超时（12秒）都会抛出异常。

注意 要确保在调用connect () 时没有同时做设备搜索，如果在搜索设备，该连接尝试会显著变慢，容易导致连接失败。

下面是一个发起Bluetooth连接的线程示例。

```
//定义Bluetooth连接线程
```

```
private class ConnectThread extends Thread{
```

```
//新建BluetoothSocket对象
```

```
private final BluetoothSocket mmSocket ;
```

```
//新建BluetoothDevice对象
```

```
private final BluetoothDevice mmDevice ;
```

```
public ConnectThread (BluetoothDevice device) {  
  
    BluetoothSocket tmp=null ;  
  
    //赋值给设备  
  
    mmDevice=device ;  
  
    try{  
  
        //根据UUID创建并返回一个BluetoothSocket  
  
        tmp=device.createRfcommSocketToServiceRecord (MY_UUID) ;  
  
    }catch (IOException e) {}  
  
    //赋值给BluetoothSocket  
  
    mmSocket=tmp ;  
  
}  
  
public void run () {  
  
    //取消发现设备  
  
    mBluetoothAdapter.cancelDiscovery () ;
```

```
try{  
  
    //连接到设备  
  
    mmSocket.connect () ;  
  
}catch (IOException connectException) {  
  
    //无法连接，关闭Socket  
  
    try{  
  
        mmSocket.close () ;  
  
    }catch (IOException closeException) {}  
  
    return ;  
  
}  
  
//管理连接  
  
manageConnectedSocket (mmSocket) ;  
  
}  
  
//取消连接
```

```
public void cancel () {  
  
    try{  
  
        //关闭BluetoothSocket  
  
        mmSocket.close () ;  
  
    }catch (IOException e) {}  
  
}  
  
}
```

注意，cancelDiscovery () 应在连接操作前被调用。在连接之前，不管搜索有没有进行，该调用都是安全的，不需要确认（当然如果要确认，可以调用isDiscovering () 方法）。处理完后别忘了调用close () 方法来关闭连接的Socket和释放所有的内部资源。

3.管理连接

如果两个设备成功建立了连接，各自会有一个BluetoothSocket对象，此时可以在设备间共享数据了。使用BluetoothSocket，传输任何数据通常来说都比较容易，通常如下进行：

- 分别使用 `getInputStream ()` 和 `getOutputStream ()` 获取输入输出流来处理传输。
- 调用 `read (byte[])` 和 `write (byte[])` 来实现数据读写。

当然，要注意一些实现细节。比如，需要用一个专门的线程来实现流的读写，因为方法 `read (byte[])` 和 `write (byte[])` 都是阻塞调用。`read (byte[])` 会阻塞，直到流中有数据可读。`write (byte[])` 虽然通常不会阻塞，但是如果远程设备调用 `read (byte[])` 不够快而导致中间缓冲区满，它也可能阻塞。所以线程中的主循环应该用于读取 `InputStream`。线程中也应该有单独的方法用来完成写 `OutputStream`。

请看下面的示例：

```
//连接管理线程
```

```
private class ConnectedThread extends Thread{
```

```
//新建BluetoothSocket对象
```

```
private final BluetoothSocket mmSocket ;
```

```
//新建输入流对象
```

```
private final InputStream mmInStream ;
```

```
//新建输出流对象  
  
private final OutputStream mmOutStream ;  
  
public ConnectedThread (BluetoothSocket socket) {  
  
    //为BluetoothSocket赋初始值  
  
    mmSocket=socket ;  
  
    //输入流赋值为null  
  
    InputStream tmpIn=null ;  
  
    //输出流赋值为null  
  
    OutputStream tmpOut=null ;  
  
    try{  
  
        //从BluetoothSocket中获取输入流  
  
        tmpIn=socket.getInputStream () ;  
  
        //从BluetoothSocket中获取输出流  
  
        tmpOut=socket.getOutputStream () ;
```

```
}catch (IOException e) {}

//为输入流赋值

mmInStream=tmpIn;

//为输出流赋值

mmOutStream=tmpOut;

}

public void run () {

byte[]buffer=new byte[1024]; //流的缓冲大小

int bytes; //用于保存read () 所读取的字节数

//保持侦听

while (true) {

try{

//从输入流中读取数据

bytes=mmInStream.read (buffer) ;
```

//发送数据到界面

```
mHandler.obtainMessage (MESSAGE_READ,bytes, -1,  
buffer) .sendToTarget () ;
```

```
}catch (IOException e) {
```

break ;

}

}

}

//发送数据到远程设备

```
public void write (byte[]bytes) {
```

try{

//写数据到输出流中

```
mmOutStream.write (bytes) ;
```

```
}catch (IOException e) {}
```

}

```
//取消

public void cancel () {

try{

//关闭连接

mmSocket.close () ;

}catch (IOException e) {}

}

}
```

构造函数中得到需要的流，一旦执行，线程会等待从InputStream来的数据。当read (byte[]) 返回从流中读到的字节后，数据通过父类的成员Handler被送到主Activity，然后继续等待读取流中的数据。向外发送数据只需简单地调用线程的write () 方法。线程的cancel () 方法是很重要的，以便连接可以在任何时候通过关闭BluetoothSocket来终止。它总在处理完Bluetooth连接后被调用。

5.5 NFC编程简介

5.5.1 NFC技术简介

近距离无线通信技术（Near Field Communication,NFC），是由飞利浦公司和索尼公司共同开发的一种非接触式识别和互联技术，可以在移动设备、消费类电子产品、PC和智能设备间进行近距离无线通信。

NFC工作频率为13.56MHz，通信距离一般在4cm或更短，传输速率可为106 kbps、212 kbps、424 kbps，甚至可提高到848 kbps。

NFC通信总是由一个发起者（initiator）和一个接收者（target）组成。通常initiator主动发送电磁场（RF）可以为被动接收者（passive target）提供电源。正是由于被动接收者可以使用发起者提供的电源，因此target可以以非常简单的形式存在，比如标签（Tags）、卡等，成本极低。NFC也支持点到点（peer to peer）的通信，此时参与通信的双方都有电源支持。在Android NFC应用中，Android手机通常是作为通信中的发起者，也就是作为NFC的读写器。Android手机也可以模拟作为NFC通信的接收者，且从Android 2.3.3起也支持P2P通信。

这样NFC终端有以下3种工作模式。

- 主动模式：NFC终端作为一个读卡器，主动发出自己的射频场去识别和读写别的NFC设备。

- 被动模式：NFC终端可以模拟成一个智能卡被读写，它只在其他设备发出的射频场中被动响应。
- 双向模式：双方都主动发出射频场来建立点对点的通信。

NFC技术的应用可分以下5类。

- 接触通过（Touch and Go）：如门禁管理、车票和门票等，用户将储存着票证或门控密码的设备靠近读卡器即可，也可用于物流管理。
- 接触支付（Touch and Pay）：如非接触式移动支付，用户将设备靠近嵌有NFC模块的POS机可进行支付，并确认交易。
- 接触连接（Touch and Connect）：如把两个NFC设备相连接进行点对点数据传输，例如在手机和笔记本间下载音乐、图片互传和交换通讯录等。
- 接触浏览（Touch and Explore）：用户可将NFC手机接靠近街头有NFC功能的智能公用电话或海报，来浏览交通信息等。
- 下载接触（Load and Touch）：用户可通过GPRS网络接收或下载信息，用于支付或门禁等功能。

在不久的将来，通过手机和NFC技术的结合，使用户仅仅通过手机就可以实现以下应用：在街边海报上和杂志上下载演唱会时间地点和节

目表；在公园里玩互动的定向越野游戏；在车站实时刷新公交车的到站时间；在办公室发送短信控制家政服务员进出住宅的时间；在学校全面代替现有学生证和学生卡；在遍布市区的智能公用电话亭查询地图、公交线路、餐饮购物等信息；在加油站、超市、银行任何有POS机的地方支付款项，并用手机收取电子发票等。

5.5.2 NFC API简介

Android对NFC的支持主要在android.nfc包中，包括的主要类如下：

- NfcAdapter。它代表了设备上的NFC硬件。NFC的应用场景有很多，但Android 2.3目前API只提供了电子标签（tags）阅读器的功能，因此NfcAdapter可理解为电子标签扫描器。
- NdefMessage。它代表了一个NDEF数据信息，NDEF（NFC Data Exchange Format）是设备与标签间传输数据的标准格式。应用程序可以从ACTION_TAG_DISCOVERED意图中获取这些NdefMessage，NdefMessage中封装了NdefRecord，每个NdefMessage中可以包含多个NdefRecord，通过类NdefMessage的getRecords（）方法可以查询到消息的所有NdefRecord。
- NdefRecord。它是双方传输信息的真正载体。

Android NFC基本工作流程如下：

步骤1 通过`android.nfc.NfcAdapter.getDefaultAdapter ()` 取得手机的`objNfcAdapter`。

步骤2 通过`objNfcAdapter.isEnabled ()` 查询该手机是否支持NFC。

步骤3 如果手机支持NFC， 手机内置的NFC扫描器（相当于`NfcAdapter`）扫描到电子标签后， 就会向应用程序发送`ACTION_TAG_DISCOVERED`的Intent, Intent的extras架构中会包含`NDEF`。

步骤4 如果接收到`ACTION_TAG_DISCOVERED`， 就提取`NdefMessage`， 并在此基础上进而提取`NdefRecord`。

在使用NFC API的时候， 应用必须在`AndroidManifest.xml`中声明获取使用权限， 方式如下：

```
<uses-permission android:name="android.permission.NFC">
```

最小的SDK版本应设置为10， 方式如下：

```
<uses-sdk android:minSdkVersion="10"/>
```

此外， 如果开发者的程序要放到Android Market， 可以申请过滤， 那些不支持NFC的用户就看不到这个发布的程序。 声明内容如下：

```
<uses-feature  
    android:name="android.hardware.nfc" android:required="true">
```

5.5.3 NFC处理流程分析

这里介绍了Android NFC的基本使用，即如何发送和接收NDEF形式的NFC数据。在Android中使用NDFF形式的数据主要有以下两种情况：

- 从一个NFC标签中读取NDFF数据。
- 使用Android Beam从一个设备传输NDFF数据到另一个设备，它允许两个Android设备间的P2P通信。

第一种情况由标签调度系统完成，它会分析发现的NFC标签，并进行适当的数据分类等，这也是本节介绍的重点。

对第二种情况有兴趣的读者请参阅Android API文档。

1. 标签调度系统 (tag dispatch system)

当Android设备搜索到标签，不要让用户去选择要处理的意图，因为设备扫描标签是在一个很短的距离里，如果让用户手动选择执行的Activity，就可能使设备远离了标签，从而打断了连接。所以，应该立刻让Activity去处理所关心的NFC标签。

为此，Android提供了标签调度系统去分析扫描到的NFC标签，然后解析它们，定位处理感兴趣的数据的应用，方法如下：

- 解析NFC标签，识别标签数据中的MIME或者URI。
- 封装MIME或者URI到一个意图。这两点在“NFC标签与MIME及URI建立映射”中介绍。
- 执行一个基于该意图的Activity。这一点在“NFC标签调度到应用”中介绍。

(1) NFC标签与MIME及URI建立映射

在开始写NFC应用程序之前，弄明白如下原理是非常重要的：NFC标签类型间的区别、标签调度系统如何解析NFC标签，以及当接收到NDEF消息时标签调度系统做了哪些特殊工作。NFC标签有很多技术标准，数据写入的方式也很多种。Android支持标准NDEF格式数据。

NDEF数据被封装在NdefMessage消息中，该消息包含一个或多个NdefRecord。每一个NdefRecord都应该是标准的NDEF数据。当然，Android也支持非标准的NDEF数据，这就需要android.nfc.tech包中类的支持。这些不在我们的讨论范围之内。

注意 在网址<http://www.nfc-forum.org/specs/speclicense>中可以下载完整的NDEF规范，在网址

<http://developer.android.com/guide/topics/connectivity/nfc/nfc.html#creating-records>中可以查阅构造标准的NDEF records的例子。

接下来将详细描述Android如何处理NDEF格式的标签。当Android设备扫描到包含NDEF格式数据的NFC标签后，它会解析这个消息，识别出数据的MIME或者URI。系统读取NdefMessage的第一个NdefRecord以决定怎样去解释整个NDEF消息（一个NDEF消息可能有多个NDEF records）。在一个格式良好的NDEF消息中，第一个NdefRecord应包含以下域：

■3位的TNF（Type Name Format）。描述了怎么样解释变长的类型域，有效的值如表5-17所示。

表 5-17 支持的类型名格式及其映射关系

TNF（类型的格式）	映射关系
TNF_ABSOLUTE_URI	类型域为 URI
TNF_EMPTY	回退 ACTION_TECH_DISCOVERED
TNF_EXTERNAL_TYPE	基于 URN 的 URI，URN 的简短形式如下 <domain_name>:<service_name>
TNF_MIME_MEDIA	类型域为 MIME
TNF_UNCHANGED	第一个 record 无效，回退 ACTION_TECH_DISCOVERED
TNF_UNKNOWN	回退 ACTION_TECH_DISCOVERED
TNF_WELL_KNOWN	MIME 或者 URI 由 RTD 决定

■变长的类型。描述了record的类型。如果TNF取值为TNF_WELL_KNOWN，则使用这个域去指明record的类型（Record Type Definition,RTD）。有效的RTD值如表5-18所示。

表 5-18 TNF_WELL_KNOWN 下支持的 RTD 及其映射关系

Record Type Definition (RTD)	映射关系
RTD_ALTERNATIVE_CARRIER	回退 ACTION_TECH_DISCOVERED
RTD_HANDOVER_CARRIER	回退 ACTION_TECH_DISCOVERED
RTD_HANDOVER_REQUEST	回退 ACTION_TECH_DISCOVERED
RTD_HANDOVER_SELECT	回退 ACTION_TECH_DISCOVERED
RTD_SMART_POSTER	解析 payload 的 URI
RTD_TEXT	text/plain 类型的 MIME
RTD_URI	基于 payload 的 URI

- 变长的ID。record的唯一标识。这个域不是经常用到，但是如果需要唯一标识一个标签，可以为标签创建一个ID。
- 变长的payload。是想读取或者写入的实际数据，一个NDEF消息可以包含多个NDEF records，所以不要以为实际数据都在第一个NDEF record中。

标签调度系统使用TNF和类型域去匹配NDEF消息中的MIME或者URI。如果成功，就会将实际传输数据封装在ACTION_NDEF_DISCOVERED意图中。如果由于一些原因，如无法匹配到MIME、URI或者NFC标签中开始本身就不含有NDEF消息，标签调度系统无法从第一个record判断数据类型，此时，标签对象的信息会被封装在ACTION_TECH_DISCOVERED意图中。

表5-17描述了标签调度系统如何根据TNF和类型域匹配MIME或者URI，同时也描述了哪些TNF是无效的，这些情况下，标签调度系统将回退ACTION_TECH_DISCOVERED。比如标签调度系统遇到一个

record类型为TNF_ABSOLUTE_URI，则正好匹配为URI。标签调度系统就会封装URI及其他信息如payload到ACTION_NDEF_DISCOVERED意图中。

(2) NFC标签调度到应用

当标签调度系统创建了一个封装了NFC标签及其他识别信息的意图后，它会将这个意图发给感兴趣的应用去过滤意图。如果超过一个以上的应用能处理这个意图，还需要用户选择用哪个Activity去处理。标签调度系统定义了3个意图，优先级从高到低如下：

- ACTION_NDEF_DISCOVERED：这个意图是当一个标签包含标准的NDEF数据并能被正确识别时被用来启动一个Activity。这个意图优先级最高，标签调度系统在任何情况下，都先试图在其他意图前使用该意图去启动一个Activity。
- ACTION_TECH_DISCOVERED：如果没有Activity注册处理ACTION_NDEF_DISCOVERED意图，标签调度系统就会试图使用该意图去启动一个应用。这个意图也是可以直接启动的，不需要先启动ACTION_NDEF_DISCOVERED。
- ACTION_TAG_DISCOVERED：这个意图在没有任何Activity处理ACTION_NDEF_DISCOVERED或ACTION_TECH_DISCOVERED时启动。

标签调度系统工作的基本方式如下，如图5-15所示。

- 当解析NFC标签时，标签调度系统试图使用所创建的意图去启动一个Activity（ACTION_NDEF_DISCOVERED意图或者ACTION_TECH_DISCOVERED意图）。
- 如果没有Activity过滤这个意图，则试图使用下一个低优先级的意图去启动一个Activity（ACTION_TECH_DISCOVERED或者ACTION_TAG_DISCOVERED），直到标签调度系统找到过滤该意图的应用或者尝试了所有可能情况。
- 如果没有任何过滤该意图的应用，则什么都不做。

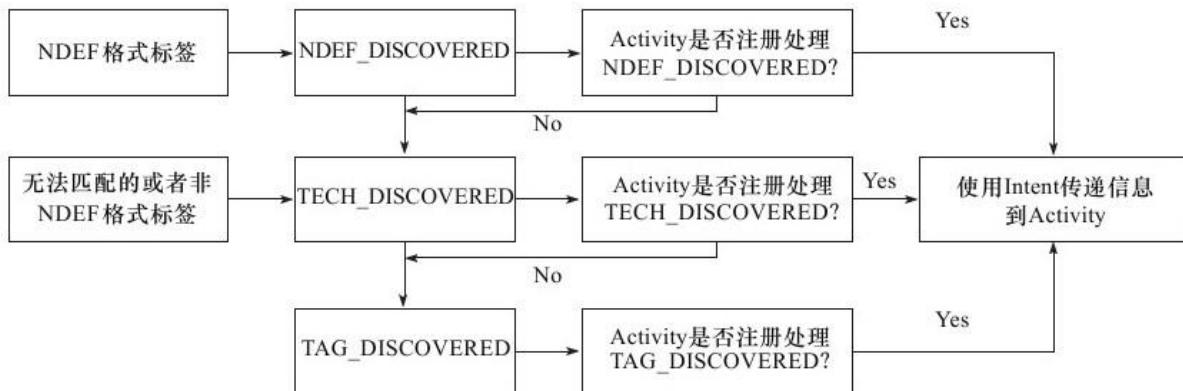


图 5-15 标签调度系统

只要可能，应保证总是NDEF消息及ACTION_NDEF_DISCOVERED意图，因为它在3个中优先级最高。这个意图较其他两个意图而言会在更

合适的时间启动应用。

2.NFC的意图过滤

当希望处理的NFC标签被扫描到了，在Android manifest中可以过滤一种、两种或者所有3种NFC意图。大多数情况下，通常是希望通过ACTION_NDEF_DISCOVERED意图。只有当没有应用过滤ACTION_NDEF_DISCOVERED意图或者传输数据是非NDEF数据时，才回退到过滤ACTION_TECH_DISCOVERED意图。过滤ACTION_TAG_DISCOVERED意图是较普遍的，适合非常多的情形。许多应用在过滤ACTION_TAG_DISCOVERED前都要先检查是否过滤ACTION_NDEF_DISCOVERED或者ACTION_TECH_DISCOVERED。只有当没有任何应用过滤前两个意图的时候，才可能过滤ACTION_TAG_DISCOVERED意图。

因为NFC标签部署多变且很多时候并不在控制之下，当然，这个并不总是发生，必要时可以回退到其他两种意图。当已经掌握了标签类型及数据读写，建议使用NDEF数据格式标签。下面的部分叙述如何过滤每种类型的意图。

(1) ACTION_NDEF_DISCOVERED

为了过滤ACTION_NDEF_DISCOVERED意图，需要声明包含希望过滤的数据的意图过滤器。下面的例子过滤含text/plain类型的MIME数据的

ACTION_NDEF_DISCOVERED意图。

```
<intent-filter>

<!--定义需要过滤的类型-->

<action android:name="android.nfc.action.NDEF_DISCOVERED"/>

<category android:name="android.intent.category.DEFAULT"/>

<data android:mimeType="text/plain"/>

</intent-filter>
```

下面代码是过滤网址http://developer.android.com/index.html所述形式的URI的过滤器。

```
<intent-filter>

<!--定义需要过滤的类型-->

<action android:name="android.nfc.action.NDEF_DISCOVERED"/>

<category android:name="android.intent.category.DEFAULT"/>

<!--过滤如下描述的网址-->

<data android:scheme="http"
```

```
    android:host="developer.android.com"
```

```
    android:pathPrefix="/index.html"/>
```

```
</intent-filter>
```

(2) ACTION_TECH_DISCOVERED

为了过滤ACTION_TECH_DISCOVERED意图，需要创建一个XML资源文件，文件中指明Activity支持的技术列表（tech-list）。可以通过getTechList（）方法获取标签采用的技术，然后跟该列表比对，看是否是应用支持的技术。

例如，如果标签扫描结果支持MifareClassic、NdefFormattable和NfcA，则技术列表必须包含所有3个或者两个或者一个，以便匹配。

下面的例子定义了所有支持的技术，可以删除不需要的项。文件名可以任意，但须保存这个文件到<project-root>/res/xml目录下。

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
```

```
<tech-list>
```

```
<!--NFC的支持列表-->
```

```
<tech>android.nfc.tech.IsoDep</tech>
```

```
<tech>android.nfc.tech.NfcA</tech>

<tech>android.nfc.tech.NfcB</tech>

<tech>android.nfc.tech.NfcF</tech>

<tech>android.nfc.tech.NfcV</tech>

<tech>android.nfc.tech.Ndef</tech>

<tech>android.nfc.tech.NdefFormatable</tech>

<tech>android.nfc.tech.MifareClassic</tech>

<tech>android.nfc.tech.MifareUltralight</tech>

</tech-list>

</resources>
```

也可以指定多个列表集合。每一个技术列表是独立的，使用getTechList()方法获取标签支持的技术后，与任意一个列表匹配都算匹配。还可以定义“与”和“或”的逻辑。下面的例子支持采用NfcA和Ndef技术的标签，也可以支持采用NfcB和Ndef技术的标签。

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
```

```
< ! -定义支持列表一-->
```

```
<tech-list>
```

```
<tech>android.nfc.tech.NfcA</tech>
```

```
<tech>android.nfc.tech.Ndef</tech>
```

```
</tech-list>
```

```
</resources>
```

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
```

```
< ! -定义支持列表二-->
```

```
<tech-list>
```

```
<tech>android.nfc.tech.NfcB</tech>
```

```
<tech>android.nfc.tech.Ndef</tech>
```

```
</tech-list>
```

```
</resources>
```

在AndroidManifest.xml文件中，需要向下面的例子一样在<activity>中使用<meta-data>元素声明刚创建的资源文件。

```
<activity>

.....



<intent-filter>

<action android:name="android.nfc.action.TECH_DISCOVERED"/>

</intent-filter>

<!--声明创建的资源文件-->

<meta-data android:name="android.nfc.action.TECH_DISCOVERED"
    android:resource="@xml/nfc_tech_filter"/>

.....



</activity>
```

(3) ACTION_TAG_DISCOVERED

过滤ACTION_TAG_DISCOVERED需要使用如下的意图过滤器。

```
<intent-filter>

<action android:name="android.nfc.action.TAG_DISCOVERED"/>
```

```
</intent-filter>
```

(4) 从意图获取信息

如果一个NFC意图启动了一个Activity，就可以从Intent中获取被扫描标签的信息。Intent可能包含如下的额外信息，但是具体包含哪个由所扫描的标签决定。

- EXTRA_TAG（必须项）：一个代表所扫描标签的对象。
- EXTRA_NDEF_MESSAGES（可选项）：从标签中解析而来的NDEF消息列表。
- {@link android.nfc.NfcAdapter#EXTRA_ID}（可选项）：标签的ID。

为了获取这些额外信息，首先看一下Activity是否是由NFC意图启动的，这样就可以确认是否已经扫描到了标签，然后在意图之外获取这些额外信息。下面是在ACTION_NDEF_DISCOVERED意图下获取NDEF消息的示例。

```
public void onResume () {  
    super.onResume () ;  
    .....  
}
```

<!--是否为ACTION_NDEF_DISCOVERED意图-->

```
if (NfcAdapter.ACTION_NDEF_DISCOVERED.equals (getIntent  
() .getAction () ) ) {
```

<!--获取从标签中解析而来的NDEF消息列表-->

```
Parcelable[]rawMsgs=
```

```
intent.getParcelableArrayExtra  
(NfcAdapter.EXTRA_NDEF_MESSAGES) ;
```

<!--消息列表不为null-->

```
if (rawMsgs !=null) {
```

<!--保存到msgs中-->

```
msgs=new NdefMessage[rawMsgs.length] ;
```

```
for (int i=0 ; i<rawMsgs.length ; i++) {
```

```
msgs[i]= (NdefMessage) rawMsgs[i] ;
```

```
}
```

```
}
```

}

}

此外，也可以从Intent中获取标签对象，通过该对象得到真正传输的数据或者列举标签支持的技术等，如下面的代码所示：

```
Tag tag=intent.getParcelableExtra (NfcAdapter.EXTRA_TAG) ;
```

5.6 小结

本章首先介绍了Android的地图与定位编程。Android地图API和基于位置的API是放在彼此独立的两个包中的。这些API通过Internet从Google服务器调用服务。要想显示地图、处理用户与地图的交互等，需要先获取map-api密钥。实例部分给出了利用MapView显示地图的方法。

接着介绍了USB主从设备的概念，Android 3.1开始支持USB这两种模式：主机模式（USB Host）、配件模式（USB Accessory）。案例中给出了Android和Arduino的交互实例。

后几节介绍了Android的无线通信技术，涉及Android Wi-Fi编程、蓝牙编程、NFC编程，其他部分高级编程由于篇幅所限没能介绍，如果读者感兴趣，可以查阅相关文档获取更多信息。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第 6 章 Android线程、数据存取、缓存和UI同步

在Android网络应用中，处理网络连接时，对性能也有较高的要求。本章讲解的Android线程、数据存取、缓存和UI同步方面的处理等知识都是与Android网络性能密切相关的。

6.1 Android线程

进程是程序的一次执行过程，而一个进程内部可以有一个或更多个线程在同时运行。进程在执行过程中拥有独立的内存单元，属于它的多个线程共享内存资源，这种多线程程序的高并发性极大地提高了程序的运行效率。

6.1.1 Android线程模型

在Android系统中，如果某个应用程序组件是第一次被启动，且这时应用程序也没有其他的组件在运行，则Android系统会为应用程序创建一个包含单个线程的Linux进程。默认情况下，同一个应用程序的所有组件都运行在同一个进程和线程里（称为main主线程）。如果组件启动时，已经存在应用程序的进程了（因为应用程序的其他组件已经在运行了），则此组件会在已有的进程和线程中启动运行。不过，可以指定组件运行在其他进程里，也可以为任何进程创建额外的线程。

1. 进程

默认情况下，同一个应用程序内的所有组件都是运行在同一个进程中的，大部分应用程序也不会去改变它。不过，如果需要指定某个特定组件所属的进程，则可以利用manifest文件来完成。

manifest文件中的每个组件，比如<activity>、<service>、<receiver>和<provider>等，都支持通过定义android:process属性指定组件运行的进程。设置此属性即可实现每个组件都在各自的进程中运行，或者某几个组件共享一个进程而其他组件运行于独立的进程。设置此属性也可以让不同应用程序的组件运行在同一个进程中，实现多个应用程序共享同一个Linux用户ID、赋予同样的权限。<application>标签也支持android:process属性，用于指定应用程序中所有组件的默认进程。

如果某一时刻内存不足，而此时又恰有其他为用户提供更紧急服务的进程需要更多内存，Android可能会决定关闭一个进程。在此进程中运行着的应用程序组件也会因此被销毁。当需要再次工作时，会为这些组件重新创建一个进程。

在决定关闭哪个进程的时候，Android系统会权衡它们对于用户的相对重要程度。比如，相对于一个拥有可见Activity的进程，更有可能去关闭一个已经在屏幕上看不见的Activity进程。也就是说，是否终止一个进程，取决于运行在此进程中组件的状态。终止进程的判定规则将在后续内容中讨论。

2. 进程的生命周期

Android系统总是试图尽可能长时间地保持应用程序进程，但为了新建或者运行更加重要的进程，总是需要清除过时进程来回收内存。为了

决定保留或终止哪个进程，根据进程中运行的组件及这些组件的状态，系统把每个进程都划入一个“重要性层次结构”中。重要性最低的进程会先被清除，然后是下一个最低的，以此类推。

重要性层次结构中的层次共有5级，以下按照重要程度由高到低列出各类进程（第一类进程是最重要的，将最后一个被终止）。

（1）前台进程

前台进程是用户完成当前操作所必需的进程。满足以下任一条件时，进程被视为处于前台状态：

- 其中运行着正与用户交互的Activity（Activity对象的onResume（）方法已被调用）。
- 其中运行着被正与用户交互的Activity绑定的服务。
- 其中运行着“前台”服务，服务以startForeground（）方式被调用。
- 其中运行着正在执行生命周期回调方法（onCreate（）、onStart（）或onDestroy（））的服务。
- 其中运行着正在执行onReceive（）方法的BroadcastReceiver。

一般而言，任何时刻前台进程的数量都不会很多，只有当内存不足以维持它们同时运行时，作为最后的策略，它们才会被终止。通常，设

备这时候已经到了内存分页状态（memory paging state），终止一些前台进程是为了保证用户界面的及时响应。

(2) 可见进程

可见进程是没有前台组件但仍会影响用户在屏幕上所见内容的进程。

满足以下任一条件时，进程被认为是可见的：

- 其中运行着不在前台的Activity，但用户仍然可见到此Activity（onPause（）方法被调用了）。可能发生这种情况的场合，例如：前台Activity打开了一个对话框，而之前的Activity还允许显示在后面。
- 其中运行着被可见（或前台）Activity绑定的服务。

可见进程是一种非常重要的进程，除非无法维持所有前台进程同时运行了，它们才会被终止。

(3) 服务进程

此进程运行着由startService（）方法启动的服务，它不会升级为上述两种级别。尽管服务进程不直接和用户所见内容关联，但它们通常在执行一些用户关心的操作（比如在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台、可见进程同时运行，否则系统会保持服务进程的运行。

(4) 后台进程

后台进程包含目前用户不可见Activity（Activity对象的onStop（）方法已被调用）的进程。这些进程对用户体验没有直接的影响，系统可以在任意时间终止它们，以回收内存供前台进程、可见进程及服务进程使用。通常会有很多后台进程在运行，所以它们被保存在一个LRU（最近最少使用）列表中，以确保最近被用户使用的Activity最后一个被终止。如果一个Activity正确实现了生命周期方法，并保存了当前的状态，则终止此类进程不会对用户体验产生可见的影响。因为在用户返回时，Activity会恢复所有可见的状态。关于保存和恢复状态的详细信息，请参阅Activities文档。

(5) 空进程

空进程就是不含任何活动应用程序组件的进程。保留这种进程的唯一目的就是用作缓存，以改善下次在此进程中运行组件的启动时间。为了在进程缓存和内核缓存间平衡系统整体资源，系统经常会终止这种进程。

依据进程中目前活跃组件的重要程度，Android会给进程评估一个尽可能高的级别。例如：如果一个进程中运行着一个服务和一个用户可见的Activity，则此进程会被评定为可见进程，而不是服务进程。

此外，一个进程的级别可能会由于其他进程的依赖而被提高，因为为其他进程提供服务的进程级别永远不会低于使用此服务的进程。比如：如果A进程中的content provider为进程B中的客户端提供服务，或进程A中的服务被进程B中的组件所调用，则A进程至少被视为与进程B同样重要。

因为运行服务的进程级别是高于后台Activity进程的，所以，如果Activity需要启动一个长时间运行的操作，则为其启动一个服务会比简单地创建一个工作线程（主线程外的单独线程，也称后台线程）更好些，尤其是在此操作时间比Activity本身存在时间还要长久的情况下。比如，一个Activity要把图片上传至Web网站，就应该创建一个服务来执行之，即使用户离开了此Activity，上传还是会在后台继续运行。不论Activity发生什么情况，使用服务可以保证操作至少拥有“服务进程”的优先级。

3.主线程

应用程序启动时，系统会为它创建一个名为main的主线程。主线程非常重要，因为它负责把事件分发给相应的用户界面Widget，包括屏幕绘图事件。它也是应用程序与Android UI组件包（来自`android.widget`和`android.view`包）进行交互的线程。因此，主线程有时也被称为UI线程。

系统不会为每个组件的实例都创建单独的线程。运行于同一个进程中的所有组件都是在UI线程中实例化的，对每个组件的系统调用也都是由UI线程分发的。因此，对系统回调进行响应的方法（比如报告用户操作的onKeyDown（）或生命周期回调方法）总是运行在UI线程中。

举个例子，当用户触摸屏幕上的按钮时，应用程序的UI线程把触摸事件分发给Widget，Widget先把自己置为按下状态，再发送一个显示区域已失效（invalidate）的请求到事件队列中。UI线程从队列中取出此请求，并通知Widget重绘自己。

如果应用程序在与用户交互的同时需要执行繁重的任务，单线程模式可能会导致运行性能很低下，除非应用程序的执行时机刚好很合适。如果UI线程需要处理每一件事情，那些耗时很长的操作，诸如访问网络或查询数据库等，将会阻塞整个UI线程。一旦线程被阻塞，所有事件都不能被分发，包括屏幕绘图事件。从用户的角度看来，应用程序看上去像是挂起了。更糟糕的是，如果UI线程被阻塞超过一定时间（目前大约是5秒），用户就会被提示“应用程序没有响应”（ANR）。这极有可能引起用户的不满，进而退出并删除这个应用程序。

此外，Android的UI组件包并不是线程安全的，因此不允许从工作线程（后台线程）中操作用户界面，只能从UI线程（主线程）中操作用户界面。于是，Android的单线程模式必须遵守以下两个规则：

- 不要阻塞UI线程。
- 不要在UI线程之外访问Android的UI组件包。

4. 工作线程

根据对以上单线程模式的描述，要想保证程序界面的响应能力，关键是不能阻塞UI线程。如果操作不能很快完成，应该让它们在单独的线程中运行（“后台”或“工作线程”）。

例如，以下响应鼠标单击事件的代码实现了在单独线程中下载图片并在ImageView显示。

//单击事件

```
public void onClick (View v) {
```

//新建线程

```
new Thread (new Runnable () {
```

```
    public void run () {
```

//下载图片

```
        Bitmap b=loadImageFromNetwork ("http://example.com/image.png") ;
```

```
//显示到界面上  
mImageView.setImageBitmap (b) ;  
}  
  
//启动  
} .start () ;  
}
```

这段代码它违反了单线程模式的第二条规则：不要在UI线程之外访问Android的UI组件包。上面这个例子在工作线程里而不是UI线程里修改了ImageView。这可能导致不明确、不可预见的后果，要跟踪这种情况也是很困难很耗时间的。

为了解决以上问题，Android提供了几种途径用来从其他线程中访问UI线程。下面列出了解决问题的几种方法：

- Activity.runOnUiThread (Runnable)

- View.post (Runnable)

- View.postDelayed (Runnable,long)

比如，可以使用View.post (Runnable) 方法修正上面的代码。

//单击事件

```
public void onClick (View v) {
```

//创建线程

```
new Thread (new Runnable () {
```

```
    public void run () {
```

//新建Bitmap对象

```
    final Bitmap bitmap=
```

//从网络下载

```
    loadImageFromNetwork ("http://example.com/image.png") ;
```

//mImageView控件使用View.post (Runnable) 方法运行加载图片

```
    mImageView.post (new Runnable () {
```

```
        public void run () {
```

//加载到界面

```
        mImageView.setImageBitmap (bitmap) ;
```

```
    }  
  
    }) ;  
  
}  
  
//开始运行  
  
}) .start () ;  
  
}
```

通过对代码进行上述的修改，现在它是线程安全的了：网络相关的操作在单独的线程里完成，而ImageView在UI线程里被操纵。

不过，随着操作变得越来越复杂，这类代码也会变得很复杂，很难维护。为了用工作线程完成更加复杂的交互处理，可以考虑在工作线程中用Handler来处理UI线程分发过来的消息。当然，最好的解决方案也许就是继承使用异步任务类AsyncTask，这个类简化了一些工作线程和UI交互的操作。

5.线程使用方法

在某些场合，方法可能会从不止一个线程中被调用，因此这些方法必须是线程安全的。

对于能被远程调用的方法，比如绑定服务（bound service）中的方法，线程安全是理所当然的。如果对IBinder所实现方法的调用发起于IBinder所在进程的内部，那么这个方法是执行在调用者的线程中的。但是，如果调用发起于其他进程，那么这个方法将运行于线程池中选出的某个线程中（而不是运行于进程的UI线程中），该线程池由系统维护且位于IBinder所在的进程中。例如，即使一个服务的onBind（）方法是从服务所在进程的UI线程中调用的，实现了onBind（）的方法对象（比如，实现了RPC方法的一个子类）仍会从线程池中的线程被调用。因为一个服务可以有不止一个客户端，所以可以同时有多个线程池与同一个IBinder方法相关联。因此IBinder方法必须实现为线程安全的。

类似地，内容提供者（content provider）也能接收来自其他进程的数据请求。尽管ContentResolver类、ContentProvider类隐藏了进程间通信管理的细节，ContentProvider中响应请求的方法，如query（）、insert（）、delete（）、update（）和getType（）方法，都是从ContentProvider所在进程的线程池中调用的，而不是进程的UI线程。因为这些方法可能会被很多线程同时调用，它们也必须实现为线程安全的。

6.1.2 异步任务类

异步任务（`AsyncTask`）能够适当地、简单地用于UI线程。这个类不需要操作线程（`Thread`）就可以完成后台任务并将结果返回UI。

异步任务的定义是：一个在后台线程上运行，而其结果是在UI线程上发布的任务。异步任务必须被继承使用。子类至少覆盖一个方法（`doInBackground (Params.....)`），但也经常覆盖另一个方法（`onPostExecute (Result)`）。

一个异步任务要用到以下3个泛型类型。

■`Params`：启动任务执行的输入参数。

■`Progress`：后台任务执行的百分比。

■`Result`：后台计算的结果类型。

在一个异步任务里，不是所有的泛型类型都要被使用。假如其中一个类型不被使用，可以简单地使用`Void`类型。例如以下代码：

```
private class MyTask extends AsyncTask<Void,Void(Void>{.....}
```

执行一个异步任务需要经过以下4个步骤：

步骤1 `onPreExecute ()`，在UI线程上调用任务后立即执行。这步通常用于设置任务，例如在用户界面显示一个进度条。

步骤2 doInBackground (Params.....) , 后台线程执行onPreExecute

() 完后立即调用。这步用于执行较长时间的后台计算。异步任务的参数也被传到这步。计算的结果必须从这步返回到上一步。在执行过程中可以调用publishProgress (Progress.....) 来更新任务的进度。

步骤3 onProgressUpdate (Progress.....) , 一次呼叫publishProgress

(Progress.....) 后调用UI线程。执行时间是不确定的。这个方法用于当后台计算还在进行时在用户界面显示进度。例如：这个方法可以被用于一个进度条动画或在文本域显示记录。

步骤4 onPostExecute (Result) , 当后台计算结束时，调用UI线程。后台计算结果作为一个参数传递到这步。

要想AysncTask类能正确工作，有以下线程规则必须遵守：

AsyncTask类必须在UI线程被中加载。

任务实例必须在UI线程中创建。

execute (Params.....) 必须在UI线程中被调用。

不要手动调用 onPreExecute () 、 onPostExecute (Result) 、
doInBackground (Params.....) 、 onProgressUpdate (Progress.....) 。

一个异步任务只能执行一次（如果执行第二次将会抛出异常）。

AsyncTask类提供的主要方法如表6-1所示。

表 6-1 AsyncTask 类提供的主要方法

方 法	解 释
cancel (boolean mayInterruptIfRunning)	尝试取消一个任务的执行，参数 mayInterruptIfRunning 如果为 true 表示正在执行的线程将会中断；如果为 false，则会允许正在执行的任务线程执行完毕
execute (Params...params)	用指定的参数来执行此任务
get (long timeout, TimeUnit unit)	等待计算结束并返回结果，timeOut 为超时等待时间，unit 为超时的时间单位
get()	等待计算结束并返回结果
getStatus()	获得任务的当前状态
isCancelled()	如果在任务正常结束之前取消任务成功则返回 true，否则返回 false
doInBackground (Params...params)	覆盖此方法以便在后台线程执行计算
onCancelled (Result result)	在 UI 线程中，当 cancel (boolean) 被调用后或者 doInBackground (Object[]) 被调用后调用此方法
onPostExecute (Result result)	在 UI 线程中调用 doInBackground (Params...) 方法之后调用此方法
onPreExecute()	在 doInBackground (Params...) 方法调用之前调用
onProgressUpdate (Progress...values)	在 UI 线程中调用 publishProgress (Progress...) 之后调用该方法

异步任务允许以异步的方式对用户界面进行操作。它先阻塞了工作线程，再在UI线程中呈现结果，在此过程中不需要对线程和handler进行人工干预。

要执行异步任务，必须继承AsyncTask类并实现doInBackground () 回调方法，该AsyncTask类对象将运行于一个后台线程池中。要更新UI时，须使用onPostExecute () 方法来分发doInBackgroundO返回的结果，由于此方法运行在UI线程中，所以UI的更新是安全的了。更新完UI后就可以在UI线程中调用execmeO来执行任务了。

例如，可以利用AsyncTask来实现6.1.1节的那个下载图片并在 ImageView显示的例子。

//单击事件

```
public void onClick (View v) {
```

//开启异步图片下载任务

```
new DownloadImageTask () .execute ("http://example.com/image.png")
```

```
}
```

//以异步方式下载图片

```
private class DownloadImageTask extends AsyncTask<String,Void,Bitmap> {
```

```
protected Bitmap doInBackground (String.....urls) {
```

//调用下载程序

```
return loadImageFromNetwork (urls[0]) ;
```

```
}
```

```
protected void onPostExecute (Bitmap result) {
```

```
//下载完成之后，加载到界面上  
  
mImageView.setImageBitmap (result) ;  
  
}  
  
}
```

现在UI是安全的，代码也得到了简化，因为任务分解成了工作线程内完成的部分和UI线程内完成的部分。

以下是关于AsyncTask工作方式的概述：

- 可以用generics来指定参数、进度值和任务最终值的类型。
- 工作线程中的doInBackground () 方法会自动执行。
- onPreExecute () 、onPostExecute () 和onProgressUpdate () 方法都在UI线程中调用。
- doInBackground () 的返回值会传给onPostExecute () 。
- 在doInBackground () 内的任何时刻，都可以调用publishProgress () 来执行UI线程中的onProgressUpdate () 。
- 可以在任何时刻、任何线程内取消任务。

要全面理解`AsyncTask`类的使用，须阅读`AsyncTask`的参考文档。

注意 在使用工作线程时，可能遇到的另一个问题——由于运行配置的改变（比如用户改变了屏幕方向）导致Activity意外重启，这可能会销毁该工作线程。

6.1.3 实战案例：利用`AsyncTask`实现多线程下载

扩展上小节的例子，本节利用`AsyncTask`实现多线程下载。文件可能较大，为了给JVM充分的空间存储，这里使用弱引用来保存`ImageView`对象。

//多线程下载

```
class BitmapDownloaderTask extends AsyncTask<String,Void,Bitmap>{
```

//定义字符串

```
private String url ;
```

//使用WeakReference解决内存问题

```
private final WeakReference<ImageView>imageViewReference ;
```

```
public BitmapDownloaderTask (ImageView imageView) {
```

```
imageViewReference=new WeakReference<ImageView>
(imageView) ;

}

@Override

protected Bitmap doInBackground (String.....params) {

//实际的下载线程的内部其实是concurrent线程，所以不会阻塞

return downloadBitmap (params[0]) ;

}

//下载完后执行

@Override

protected void onPostExecute (Bitmap bitmap) {

//判断是否取消

if (isCancelled ()) {

//如果取消则赋值为null

bitmap=null ;
```

}

//WeakReference对象不为null

if (imageViewReference != null) {

//获取图像

ImageView imageView = imageViewReference.get();

if (imageView != null) {

//下载完设置imageview为刚才下载的bitmap对象

imageView.setImageBitmap(bitmap);

}

}

}

}

调用方法如下：

//调用下载方法

//参数String url表示下载的地址

//参数ImageView imageView表示显示的控件

```
public void download (String url,ImageView imageView) {
```

//新建下载任务

```
BitmapDownloaderTask task=new BitmapDownloaderTask
```

```
(imageView) ;
```

//传入参数

```
task.execute (url) ;
```

```
}
```

6.2 数据存取

Android提供了若干种用于存储永久性应用程序数据的方法。具体选择何种方案要根据需要而定，比如数据仅为应用私有还是为其他应用（以及用户）所共享、数据需要多少空间等。

数据存储有以下几种选择，如表6-2所示。

表 6-2 数据存储方式

方 式	主 要 应 用
Shared Preferences	以键值对的方式存储私有的原始数据
Internal Storage	在存储器上存储私有的数据
External Storage	在外部存储器上存储公共数据
SQLite Database	在私有数据库中存储结构化的数据
Network Connection	使用网络服务器存储数据

Android提供了在其他应用中访问私有数据的方法，即使用内容提供器(contentprovider)。内容提供器是一个可选的组件，为应用程序数据提供读/写权限，并受制于给定的限制。本书限于篇幅不再涉及，以下几节重点讨论Shared Preferences、Internal Storage、External Storage 及 SQLite Database。

6.2.1 Shared Preferences 数据存储

SharedPreferences类提供了一个通用的框架，允许保存和检索以持久化的键值对形式存储的原始数据。可以使用SharedPreferences保存任意类型的原始数据：布尔型（boolean）、浮点型（float）、整型（int）、长整型（long）和字符串型（string）。即使应用程序已经退出，这些数据仍将会存放在用户会话中。

要在应用程序中获得SharedPreferences对象，可以使用以下两种方法：

- 当需要多个配置文件，使用getSharedPreferences()方法，配置文件由名字标识，名字由第一个参数指定。
- 当仅需要一个配置文件时，使用getPreferences()方法。由于这时只有Activity这唯一一个配置文件，所以不必提供名称。

向SharedPreferences写入值的步骤如下：

步骤1 调用edit()，获得一个 SharedPreferences.Editor。

步骤2 使用putBoolean()或putString()等方法向其添加值。

步骤3 使用commit()提交新值。

可以使用getBoolean()或getString()等方法从SharedPreferences读取值。

下面这个例子实现的是在计算器静音模式下保存一个配置。

```
//计算器静音模式下保存配置

public class Calc extends Activity{

    //定义保存SharedPreferences使用的名称

    public static final String PREFS_NAME="MyPrefsFile" ;

    @Override

    protected void onCreate (Bundle state) {

        super.onCreate (state) ;

        //存储设置

        SharedPreferences settings=getSharedPreferences (PREFS_NAME, 0) ;

        //加入设置变量，设置其值为false

        boolean silent=settings.getBoolean ("silentMode", false) ;

    }

    @Override

    protected void onStop () {
```

```
super.onStop () ;  
  
//修改SharedPreferences中的存储值  
  
SharedPreferences settings=getSharedPreferences (PREFS_NAME, 0) ;  
  
//新建SharedPreferences编辑器  
  
SharedPreferences.Editor editor=settings.edit () ;  
  
//设置Boolean值  
  
editor.putBoolean ("silentMode", mSilentMode) ;  
  
//提交修改  
  
editor.commit () ;  
  
}  
  
}
```

6.2.2 Internal Storage 数据存储

可以直接将文件保存在设备上的内部存储（Internal Storage）中。默认情况下，存放于内部存储的文件为应用程序所私有，其他应用程序和用户不能够访问它们。当用户卸载应用，这些文件也会被删除。

在内部存储中创建并写入一个私有文件的步骤如下：

步骤1 调用openFileOutput () 方法传入文件名和操作模式。方法返回一个FileOutputStream对象。

步骤2 使用write () 写文件。

步骤3 使用close () 关闭文件流。

下面举一个例子，具体如下：

//定义文件名

```
String FILENAME="hello_file" ;
```

//文件内容

```
String string="hello world ! " ;
```

//实例化文件输出流

//Context.MODE_PRIVATE为默认操作模式，代表该文件是私有数据，
只能被应用本身访问

//在该模式下，写入的内容会覆盖原文件的内容

```
FileOutputStream fos=openFileOutput  
    (FILENAME,Context.MODE_PRIVATE) ;  
  
//写入文件内容  
  
fos.write (string.getBytes ()) ;  
  
//关闭文件  
  
fos.close () ;
```

上面的例子中，用到了MODE_PRIVATE模式，其将创建新文件（或替换同名文件），所创建的文件是应用私有的。其他可用的模式有：

MODE_APPEND（检查文件是否存在，存在就往文件追加内容，否则就创建新文件）

MODE_WORLD_READABLE（表示当前文件可以被其他应用读取）

MODE_WORLD_WRITEABLE（表示当前文件可以被其他应用写入）

从内部存储中读取一个文件的步骤如下：

步骤1 调用openFileInputO，并传递需要读取的文件的名称。这个方法返回一个FileInputStream。

步骤2 使用readO从文件中读取数据。

步骤3 使用close()关闭文件流。

注意 如果想在编译时往应用中存入一个静态文件，就得把文件保存到项目的res/raw目录下。可以调用openRawResource () 并传递资源的ID (R.raw.<filename>) 来打开这个静态文件。这个方法返回一个InputStream对象，可以使用它读取文件，但不能向原始文件中写入信息。

和内部存储相关的其他有用的方法如表6-3所示。

表 6-3 内部存储相关方法

方 法	功 能 描 述
getCacheDir()	应用保存临时缓存文件的内部目录
getFilesDir()	取得内部文件在文件系统中保存位置的绝对路径
getDir()	创建（或者打开已存在的）内部存储空间所在的目录
deleteFile()	删除内部存储的一个文件
fileList()	返回当前由应用保存的文件的列表

6.2.3 External Storage 数据存储

所有兼容Android的设备都支持一个可共享的外部存储（External Storage），可用来保存文件。外部存储器可以是一个可移动的存储设备（比如SD卡）或者一个内部的（不可移动的）存储器。保存在外部存储器上的文件是可读的，并且当USB传输启用时，用户能够在计算机上修改它们。

有的时候，设备在使用内部存储的一部分作为外部存储的同时提供SD存储。此时，SD存储就不再是外部存储的一部分，应用将无法访问它。

注意 如果用户挂载外部存储到计算机上，或者移除媒体，外部存储将不可用。对于这些保存在外部存储器上的文件，没有强制的安全措施。所有的应用都可以读/写这些文件。用户也可以删除它们。

1.检测媒体可用性

在对外部存储做任何事情之前，总是应当调用getExternalStorageState()以检测媒体是否可用。媒体可能被计算机挂载、丢失、只读，或者处于某些其他状态，此时媒体就不可用。以下是检测媒体是否可用的示例代码：

```
//新建boolean变量，表示外部存储读操作是否可用
```

```
boolean mExternalStorageAvailable=false；
```

```
//新建boolean变量，表示外部存储写操作是否可用
```

```
boolean mExternalStorageWriteable=false；
```

```
//获取外部存储状态
```

```
String state=Environment.getExternalStorageState () ；
```

```
//获取外部存储路径是否成功

if (Environment.MEDIA_MOUNTED.equals (state) ) {

//可以对外部存储进行读写

mExternalStorageAvailable=mExternalStorageWriteable=true ;

}else if (


//存储媒体已经挂载， 挂载点只读

Environment.MEDIA_MOUNTED_READ_ONLY.equals (state) ) {

//只能对外部存储进行读操作

mExternalStorageAvailable=true ;

//对外部存储不能进行写操作

mExternalStorageWriteable=false ;

}else{

//对外部存储不能进行读和写的操作

mExternalStorageAvailable=mExternalStorageWriteable=false ;
```

}

这个例子检测了外部存储是否可读或可写。getExternalStorageState () 方法返回想要检测的其他状态，比如，媒体是否被共享（已连接到一台计算机）、是否完全丢失、是否被移除等。当应用需要访问媒体时，可以依据这些以更详细的信息通知用户。

2.保存共享文件

如果想要保存与应用没有关联的文件，并且这些文件不在应用卸载时被删除，可以把这些文件保存在外部存储的一个公共目录上。这些目录位于外部存储的根目录下，比如Music/、Pictures/、Ringtones/，以及其他目录。

可以调用getExternalStoragePublicDirectory (String type) 方法，返回为File对象，该方法可以获取type类型的公共目录，该方法位于 android.os.Environment包中，其参数是默认的公共文件夹的属性定义，主要参数有：

- public static String DIRECTORY_ALARMS
- public static String DIRECTORY_DCIM
- public static String DIRECTORY_DOWNLOADS

- public static String DIRECTORY_MOVIES
- public static String DIRECTORY_MUSIC
- public static String DIRECTORY_NOTIFICATIONS
- public static String DIRECTORY_PICTURES
- public static String DIRECTORY_PODCASTS
- public static String DIRECTORY_RINGTONES

其不同参数返回的目录不同，分别对应为下面的目录，其含义如下：

- Music/——媒体扫描器把所有在此发现的媒体归类为用户的音乐。
- Podcasts/——媒体扫描器把所有在此发现的媒体归类为播客。
- Ringtones/——媒体扫描器把所有在此发现的媒体归类为铃声。
- Alarms/——媒体扫描器把所有在此发现的媒体归类为闹铃。
- Notifications/——媒体扫描器把所有在此发现的媒体归类为提示音。
- Pictures/——所有照片（相机拍摄的除外）。
- Movies/——所有影片（摄像机拍摄的除外）。

■Download/——各类下载。

3.保存缓存文件

如果正在使用API Level 8或者更高版本，调用getExternalCacheDir () 打开一个File，它表示应用应当保存文件所在的外部存储目录。如果用户卸载了应用，这些文件会被自动删除。然而，在应用的生命周期中，应当管理这些缓存文件，并且删除其中过期的部分以保证文件空间。

如果正在使用API Level 7或者更低版本，使用 getExternalStorageDirectory () 打开一个File，它表示外部存储的根目录。接下来，应当写入数据，目录为:/Android/data/<package_name>/files/。

其中，<package_name>是Java风格的包名称，例如，com.example.android.app。

6.2.4 SQLite Databases 数据存储

SQLite是一款轻型的关系型数据库。它的设计目标是以嵌入式的方式应用于各相关产品中，目前很多嵌入式产品中都使用了它。它占用资源非常低，在嵌入式设备中，可能只需要几百KB的内存就够了。

Android在运行时（Runtime）集成了SQLite，所以每个Android应用程序都可以使用SQLite数据库。Android提供了一些新的API来使用SQLite数据库，数据库存储在“data/<项目文件夹>/databases/”下。

查看SQLite数据库比较方便的方法是在Firefox（火狐浏览器）中安装SQLite Manager的插件，其下载地址：<https://code.google.com/p/sqlite-manager/>。可以从Android手机中导出SQLite文件，并使用该插件查看、编辑等。SQLite Manager的运行界面如图6-1所示。

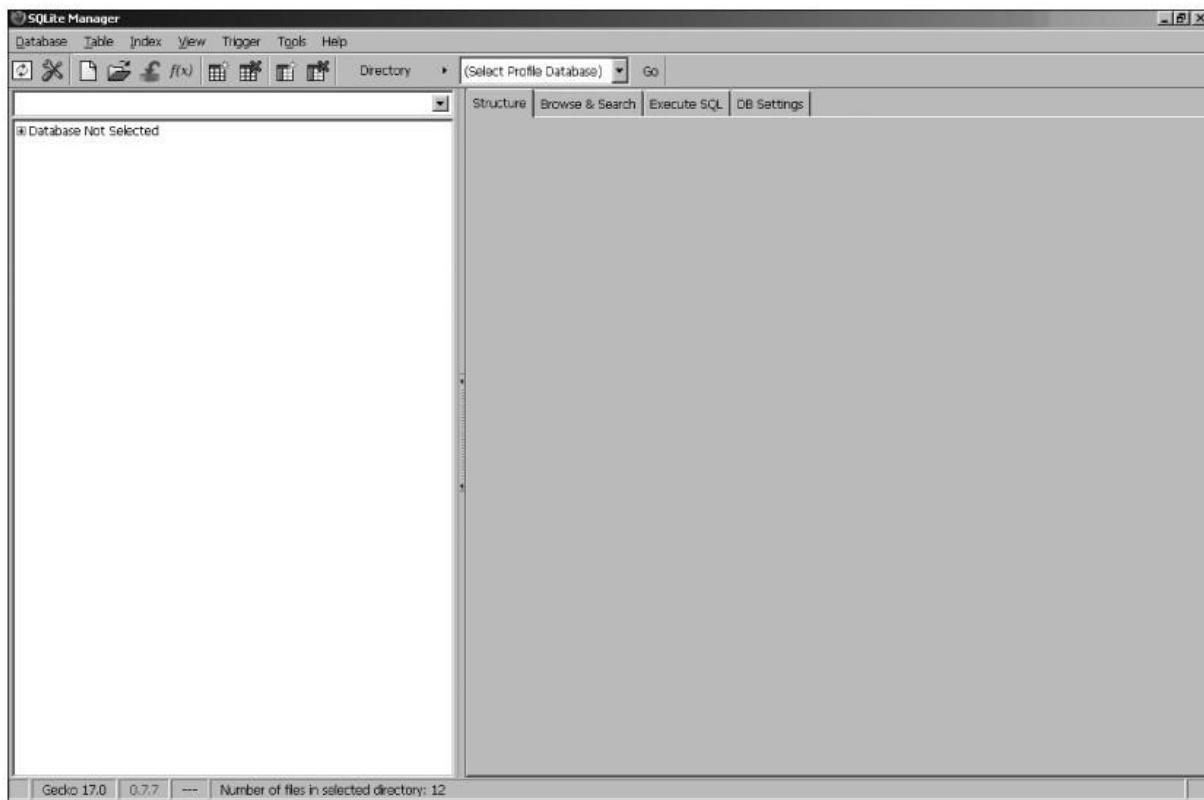


图 6-1 SQLite Manager运行界面

一般数据采用静态数据类型，而SQLite采用的是动态数据类型，会根据存入值自动判断。SQLite具有5种数据类型，如表6-4所示。

表 6-4 SQLite 提供的数据类型

数据类型	解 释
NULL	空类型
INTEGER	带符号的整型，具体取决于存入数字的范围大小
REAL	浮点型
TEXT	字符串文本
BLOB	二进制对象

6.2.5 实战案例：SQLite数据库操作

Android中使用SQLite需要创建一个继承自SQLiteOpenHelper的类，可以通过继承这个类，实现它的一些方法来对数据库进行一些操作。

SQLiteDatabase直接操作数据库的对象，可以通过SQLiteOpenHelper的getWritableDatabase () 和getReadableDatabase () 这两个方法来获得SQLiteDatabase的对象，从而通过该对象对数据库进行操作。

SQLiteOpenHelper是个抽象类，创建的继承自SQLiteOpenHelper的类重写3个方法：构造方法、onCreate () 方法、onUpgrade () 方法。

1.构造方法

构造方法的具体声明如下：

```
public SQLiteOpenHelper (Context context,String  
name,SQLiteDatabase.CursorFactory factory,int version)
```

这里需要4个参数：context表示上下文对象；name表示数据库名称；CursorFactory表示一个可选的游标工厂（通常是Null）；version表示当前数据库的版本，值必须是整数并且是递增的状态。该构造方法提供一套自动执行的机制来帮助开发者创建、更新、打开数据库。

示例如下：

```
public DatabaseHelper (Context context,String name,CursorFactory  
factory,int version) {  
  
    //必须通过super调用父类当中的构造函数  
  
    super (context,name,factory,version) ;  
  
}
```

2.onCreate () 方法

onCreate () 方法的具体声明如下：

```
public abstract void onCreate (SQLiteDatabase db)
```

`onCreate ()` 方法需要一个`SQLiteDatabase`对象作为参数，根据需要对这个对象填充表和初始化数据。`onCreate ()` 函数在第一次创建的时候执行，实际上是在第一次得到`SQLiteDatabase`对象的时候才会调用这个方法。

3.`onUpgrade ()` 方法

`onUpgrade ()` 方法的具体声明如下：

```
public abstract void onUpgrade (SQLiteDatabase db,int oldVersion,int  
newVersion)
```

`onUpgrade ()` 方法需要3个参数：一个`SQLiteDatabase`对象，一个旧的版本号（`oldVersion`）和一个新的版本号（`newVersion`）。

如果当前传入的数据库版本号比上次创建或升级的版本号高，`SQLiteOpenHelper`就会调用`onUpdate ()` 方法。也就是说，当数据库第一次创建时会有一个初始的版本号。当需要对数据库中的表、视图等组建升级时可以增大版本号，再重新创建它们。

示例如下：

```
public void onCreate (SQLiteDatabase db) {
```

```
//创建数据库后，对数据库的操作
```

```
}
```

```
@Override
```

```
public void onUpgrade (SQLiteDatabase db,int oldVersion,int  
newVersion) {
```

```
//更改数据库版本的操作
```

```
}
```

```
@Override
```

```
public void onOpen (SQLiteDatabase db) {
```

```
super.onOpen (db) ;
```

```
//每次成功打开数据库后首先被执行
```

```
}
```

以下给出了对SQLite数据库进行增、删、改、查等操作的完整示例，
运行效果如图6-2所示。



图 6-2 SQLite Databases操作示意图

SQLiteSimpleActivity中代码如下：

```
import android.os.Bundle ;  
  
import android.app.Activity ;  
  
import android.content.ContentValues ;  
  
import android.database.Cursor ;  
  
import android.database.sqlite.SQLiteDatabase ;  
  
import android.view.Menu ;  
  
import android.view.View ;  
  
import android.view.View.OnClickListener ;
```

```
import android.widget.Button ;  
  
//Android对SQLite操作简单示例  
  
public class SQLiteSimpleActivity extends Activity implements  
OnClickListener{  
  
    //新建创建数据库按钮  
  
    private Button createDatabaseButton=null ;  
  
    //新建更新数据库按钮  
  
    private Button updateDatabaseButton=null ;  
  
    //新建插入数据按钮  
  
    private Button insertButton=null ;  
  
    //新建更新数据按钮  
  
    private Button updateButton=null ;  
  
    //新建查询数据按钮  
  
    private Button selectButton=null ;  
  
    //新建删除数据按钮
```

```
private Button deleteButton=null ;  
  
@Override  
  
protected void onCreate (Bundle savedInstanceState) {  
  
    super.onCreate (savedInstanceState) ;  
  
    //加载布局文件  
  
    setContentView (R.layout.activity_sqlite_simple) ;  
  
    //按钮绑定到布局文件中对应的ID上  
  
    createDatabaseButton= (Button)  
  
    findViewById (R.id.createDatabase) ;  
  
    updateDatabaseButton= (Button)  
  
    findViewById (R.id.updateDatabase) ;  
  
    insertButton= (Button) findViewById (R.id.insert) ;  
  
    updateButton= (Button) findViewById (R.id.update) ;  
  
    selectButton= (Button) findViewById (R.id.select) ;
```

```
deleteButton= (Button) findViewById (R.id.delete) ;  
  
//为各个按钮设置事件监听  
  
createDatabaseButton.setOnClickListener (this) ;  
  
updateDatabaseButton.setOnClickListener (this) ;  
  
insertButton.setOnClickListener (this) ;  
  
updateButton.setOnClickListener (this) ;  
  
selectButton.setOnClickListener (this) ;  
  
deleteButton.setOnClickListener (this) ;  
  
}  
  
//单击事件  
  
@Override  
  
public void onClick (View v) {  
  
//获取被单击按钮的ID  
  
switch (v.getId () ) {
```

```
case R.id.createDatabase :  
  
    //创建了一个DatabaseHelper对象  
  
    DatabaseHelper dbHelper=new DatabaseHelper (   
  
        SQLiteSimpleActivity.this, "simle_db") ;  
  
    //创建或打开只读数据库  
  
    dbHelper.getReadableDatabase () ;  
  
    break ;  
  
case R.id.updateDatabase :  
  
    DatabaseHelper dbHelperUpdateDatabase=new DatabaseHelper (   
  
        SQLiteSimpleActivity.this, "simple_db", 2) ;  
  
    dbHelperUpdateDatabase.getReadableDatabase () ;  
  
    break ;  
  
case R.id.insert :  
  
    //创建ContentValues对象
```

```
ContentValues valuesInsert=new ContentValues () ;  
  
//向该对象中插入键值对，其中键是列名  
  
//值是希望插入到这一列的值  
  
//值必须和数据库当中的数据类型一致  
  
valuesInsert.put ("id", 1) ;  
  
valuesInsert.put ("name", "gyz") ;  
  
valuesInsert.put ("sex", "man") ;  
  
valuesInsert.put ("age", "30") ;  
  
//创建DatabaseHelper对象  
  
DatabaseHelper dbHelperInsert=new DatabaseHelper (  
    SQLiteSimpleActivity.this, "simple_db", 2) ;  
  
//得到一个可写的SQLiteDatabase对象  
  
SQLiteDatabase sqliteDatabaseInsert=dbHelperInsert  
.getWritableDatabase () ;
```

```
//调用insert方法，将数据插入数据库当中

//第一个参数为表名称

//第二个参数为可以选参数，SQL不允许一个空列

//如果ContentValues是空的，那么这一列被明确地指明为NULL值

//第三个参数：ContentValues对象

sqliteDatabaseInsert.insert ("person", null,valuesInsert) ；

break ;

case R.id.update :

//创建一个DatabaseHelper对象

DatabaseHelper dbHelperUpdate=new DatabaseHelper (
    SQLiteSimpleActivity.this, "simple_db", 2) ；

//得到一个可写的SQLiteDatabase对象

SQLiteDatabase sqliteDatabaseUpdate=dbHelperUpdate
    .getWritableDatabase () ；
```

```
//创建一个ContentValues对象  
  
ContentValues valuesUpdate=new ContentValues () ;  
  
valuesUpdate.put ("name", "gyz_update") ;  
  
valuesUpdate.put ("sex", "man") ;  
  
valuesUpdate.put ("age", "30") ;  
  
//调用update方法  
  
//第一个参数String：表名  
  
//第二个参数ContentValues:ContentValues对象  
  
//第三个参数String:where子句  
  
//相当于SQL语句where后面的语句， ?号是占位符  
  
//第四个参数String[]：占位符的值  
  
sqliteDatabaseUpdate.update ( "persion", valuesUpdate, "id=?", new String[]{"1"}) ;  
  
break ;
```

```
case R.id.select :  
  
    String id=null ;  
  
    String name=null ;  
  
    //创建DatabaseHelper对象  
  
    DatabaseHelper dbHelperSelect=new DatabaseHelper  
        (SQLiteSimpleActivity.this, "simple_db", 2) ;  
  
    //得到一个只读的SQLiteDatabase对象  
  
    SQLiteDatabase sqliteDatabaseSelect=dbHelperSelect.getReadableDatabase  
        () ;  
  
    //调用SQLiteDatabase对象的query方法进行查询  
  
    //返回一个Cursor对象：由数据库查询返回的结果集对象  
  
    //第一个参数String：表名  
  
    //第二个参数String[]：要查询的列名  
  
    //第三个参数String：查询条件  
  
    //第四个参数String[]：查询条件的参数
```

```
//第五个参数String：对查询的结果进行分组  
  
//第六个参数String：对分组的结果进行限制  
  
//第七个参数String：对查询的结果进行排序  
  
Cursor cursor=sqliteDatabaseSelect.query (   
  
    "persion", new String[]{"id", "name"},  
  
    "id=?", new String[]{"1"}, null,null,null) ;  
  
//将游标移动到下一行，从而判断该结果集是否还有下一条数据  
  
//如果有则返回true，没有则返回false  
  
while (cursor.moveToNext () ) {  
  
    id=cursor.getString (cursor.getColumnIndex ("id") ) ;  
  
    name=cursor.getString (cursor.getColumnIndex ("name") ) ;  
  
}  
  
break ;  
  
case R.id.delete :
```

```
//创建DatabaseHelper对象  
  
DatabaseHelper dbHelperDelete=new DatabaseHelper (   
  
    SQLiteSimpleActivity.this, "simple_db", 2) ;  
  
//获得可写的SQLiteDatabase对象  
  
SQLiteDatabase sqliteDatabaseDelete=dbHelperDelete.getWritableDatabase  
() ;  
  
//调用SQLiteDatabase对象的delete方法进行删除操作  
  
//第一个参数String : 表名  
  
//第二个参数String : 条件语句  
  
//第三个参数String[] : 条件值  
  
sqliteDatabaseDelete.delete ("person", "id=?", new String[]{"1"}) ;  
  
break ;  
  
default :  
  
break ;  
  
}
```

```
}
```

```
//加载Menu菜单
```

```
@Override
```

```
public boolean onCreateOptionsMenu (Menu menu) {
```

```
    getMenuInflater () .inflate (R.menu.activity_sqlite_simple,menu) ;
```

```
    return true ;
```

```
}
```

```
}
```

DatabaseHelper类继承自SQLiteOpenHelper类，其中封装了示例中需要使用的操作，其代码如下：

```
import android.content.Context ;
```

```
import android.database.sqlite.SQLiteDatabase ;
```

```
import android.database.sqlite.SQLiteDatabase.CursorFactory ;
```

```
import android.database.sqlite.SQLiteOpenHelper ;
```

```
//数据库帮助类
```

```
public class DatabaseHelper extends SQLiteOpenHelper{  
  
    //定义版本号  
  
    private static final int VERSION=1 ;  
  
    //含有4个参数的构造函数  
  
    public DatabaseHelper (Context context,String name,  
        CursorFactory factory,int version) {  
  
        super (context,name,factory,version) ;  
  
    }  
  
    //含有3个参数的构造函数  
  
    public DatabaseHelper (Context context,String name,int version) {  
  
        this (context,name,null,version) ;  
  
    }  
  
    //含有两个参数的构造函数  
  
    public DatabaseHelper (Context context,String name) {
```

```
this (context,name,VERSION) ;  
}  
  
//第一次得到SQLiteDatabase对象的时候会调用这个方法  
  
@Override  
  
public void onCreate (SQLiteDatabase db) {  
  
    System.out.println ("create a database") ;  
  
    //execSQL用于执行SQL语句， 创建表person  
  
    db.execSQL ("create table person  
    (id int,name varchar (20) , sex varchar (20) , age int) ") ;  
  
}  
  
@Override  
  
public void onUpgrade (SQLiteDatabase arg0, int arg1, int arg2) {  
  
}  
}
```

在上面的例子中使用到了按钮监听的方法，下面介绍Android提供的监听按钮的几种方法。

方法一 为每一个按钮单独的写一个处理函数。

```
Button button= (Button) findViewById (R.id.button) ;  
  
//监听事件  
  
button.setOnClickListener (new View.OnClickListener () {  
  
    public void onClick (View arg0) {  
  
        //这里写单击按钮的处理  
  
    }  
  
}) ;
```

方法二 可以复用的按钮监听处理方法。

```
Button button1= (Button) findViewById (R.id.button1) ;  
  
Button button2= (Button) findViewById (R.id.button2) ;  
  
button1.setOnClickListener (new MyButtonOnClickListener () ) ;  
  
button2.setOnClickListener (new MyButtonOnClickListener () ) ;
```

```
class MyButtonOnClickListener implements OnClickListener{  
  
    public void onClick (View v) {  
  
        //这里写单击按钮的处理  
  
    }  
  
}
```

方法三 实现OnClickListener监听接口。

```
public class MyActivity extends Activity implements OnClickListener{}
```

在代码里面添加监听：

```
Button button1= (Button) findViewById (R.id.button1) ;
```

```
Button button 2= (Button) findViewById (R.id.button2) ;
```

```
Button button 3= (Button) findViewById (R.id.button3) ;
```

//3个按钮添加监听

```
button1.setOnClickListener (this) ;
```

```
button2.setOnClickListener (this) ;
```

```
button3.setOnClickListener (this) ;
```

下面是单击事件处理的内容：

```
public void onClick (View v) {
```

```
    switch (v.getId ()) {
```

```
        case R.id.button1 :
```

```
            //button1的处理内容
```

```
            break ;
```

```
        case R.id.button2 :
```

```
            //button2的处理内容
```

```
            break ;
```

```
        case R.id.button3 :
```

```
            //button3的处理内容
```

```
            break ;
```

```
    default:break ;
```

}

}

6.3 网络判定

终端应用对网络的连接情况非常敏感，特别是在流量或者网速受到限制的条件下。当需要在线收看视频或者下载大型软件游戏的时候，在Wi-Fi连接下会取得比较好的体验；当使用IM、Weibo等通信工具的时候，并不需要很大的网络流量，在低速收费网络里面也可以流畅使用。如果用户在后一种网络环境中想下载视频，显然等待的时间可能会很长，此时可以提示用户更新网络。这就需要使用ConnectivityManager检查用户实际使用的网络是否连接，以及使用的是什么类型的网络。

6.3.1 判断用户是否连接

使用ConnectivityManager可以判断是否有网络连接，示例代码如下：

```
//获取ConnectivityManager的实例
```

```
ConnectivityManager cm= (ConnectivityManager) context
```

```
.getSystemService (Context.CONNECTIVITY_SERVICE) ;
```

```
//获取NetworkInfo的实例
```

```
NetworkInfo activeNetwork=cm.getActiveNetworkInfo () ; //判断是否有网络连接
```

```
boolean isConnected=activeNetwork.isConnectedOrConnecting () ;
```

6.3.2 判断网络连接的类型

网络连接的类型有多等，为了方便区别各种类型，Android中给出了方便的判断方法。比如判断是否为Wi-Fi类型，代码如下：

```
boolean isWiFi=activeNetwork.getType ()  
==ConnectivityManager.TYPE_WIFI ;
```

boolean变量将赋值是否为Wi-Fi类型，activeNetwork.getType () 方法将获得的类型与系统定义的类型变量ConnectivityManager.TYPE_WIFI做比较，如果比较结果相等则isWiFi赋值为TRUE，比较结果不相等则赋值为FALSE。

6.3.3 监控网络连接改变

对于移动终端设备来说，联网类型、方式的改变是很频繁的事情，故监视网络连接的改变是非常必要的。当网络连接改变的时候，ConnectivityManager会广播一个CONNECTIVITY_ACTION ("android.net.conn.CONNECTIVITY_CHANGE")。可以在manifest中

注册一个广播接收器以便接收该广播。其注册的代码添加到对应的Activity声明目录中，可以在网络连接发生变化的时候通知该Activity，代码如下：

```
<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>
```

6.3.4 实战案例：根据广播消息判断网络连接情况

获取和使用网络状态以及其变化的值不是一个简单分割的过程，很多时候需要整合使用。下面给出了一个综合示例，介绍根据广播消息判断网络连接情况，并给出相应的处理，其主要代码如下：

```
public class ConnectionChangeReceiver extends BroadcastReceiver{
```

```
//定义调试使用的标签变量
```

```
//获取类名
```

```
private static final String
```

```
TAG=ConnectionChangeReceiver.class.getSimpleName () ;
```

```
@Override
```

```
public void onReceive (Context context,Intent intent) {
```

```
Log.e (TAG, "网络状态改变") ;  
  
boolean success=false ;  
  
//获得网络连接服务  
  
ConnectivityManager connManager= (ConnectivityManager)  
context.getSystemService (Context.CONNECTIVITY_SERVICE) ;  
  
//获取Wi-Fi网络连接状态  
  
State state=connManager.getNetworkInfo (   
ConnectivityManager.TYPE_WIFI) .getState () ;  
  
//判断是否正在使用Wi-Fi网络  
  
if (State.CONNECTED==state) {  
  
success=true ;  
  
}  
  
//获取GPRS网络连接状态  
  
state=connManager.getNetworkInfo
```

```
(ConnectivityManager.TYPE_MOBILE) .getState () ;  
  
//判断是否正在使用GPRS网络  
  
if (State.CONNECTED !=state) {  
  
    success=true ;  
  
}  
  
//没有连接成功  
  
if ( ! success) {  
  
    //弹出提示框  
  
    Toast.makeText (context,  
    context.getString (R.string.your_network_has_disconnected) ,  
    Toast.LENGTH_LONG) .show () ;  
  
}  
  
}
```

需要在manifest文件里面进行权限声明和广播接收器注册。

<!--检测网络链接的变化-->

<uses-permission

 android:name="android.permission.ACCESS_NETWORK_STATE"/>

<!--定义接收权限-->

<receiver

 android:name="you_package_name.ConnectionChangeReceiver" android:label="NetworkConnection">

<!--定义过滤器-->

<intent-filter>

<action android:name="android.net.conn.CONNECTIVITY_CHANGE"/>

</intent-filter>

</receiver>

注册和取消监听有以下两种方式。

方式一 在Activity的onCreate () 中注册监听。

//注册网络监听

```
IntentFilter filter=new IntentFilter () ;  
  
//添加过滤事件  
  
filter.addAction (ConnectivityManager.CONNECTIVITY_ACTION) ;  
  
//注册监听  
  
registerReceiver (mNetworkStateReceiver,filter) ;
```

在Activity的onDestroy () 中取消监听。

//取消监听

```
unregisterReceiver (mNetworkStateReceiver) ;
```

方式二 应用启动时，启动Service，在Service的onCreate () 方法中注册网络监听。

//注册网络监听

```
IntentFilter filter=new IntentFilter () ;
```

/添加过滤事件

```
filter.addAction (ConnectivityManager.CONNECTIVITY_ACTION) ;
```

//注册监听

```
registerReceiver (mNetworkStateReceiver,filter) ;
```

应用退出时，Service关闭，在Service的onDestroy () 方法中取消监听。

```
//取消监听
```

```
unregisterReceiver (mNetworkStateReceiver) ;
```

6.4 消息缓存

缓存的目的是避免重复计算，特别是对一些比较耗时间和资源的计算。在移动终端中，由于存在联网下载缓慢、无网络信号、流量计费等情况，需要把一些耗时、耗流量的操作缓存到本地，比如微博里面图片的下载、用户好友信息的更新、浏览器里面Web页面的本地存储等。缓存有利于更好的用户体验，并能节约用户的下载流量和载入时间。

6.4.1 Android中的缓存机制

Android中提供的缓存机制是利用本地存储实现的。

- 新下载数据的时候，将数据缓存到本地。
- 再次下载之前，先判定该资源是否已经被缓存，如果是，则优先使用本地资源；如果没有被缓存，则从网络上下载资源，并进行缓存。

实际上使用缓存机制的时候，还要考虑到额外的两个条件：空间和时间。

对于存储空间的条件限制，处理方法一般是结合应用下载量大小和用户选择来确定，比如以缓存文本为主的应用，由于文本本身占用极小

的空间，其缓存大小可以根据用户的磁盘空间大小来确定；以缓存图片为主应用，由于图片占用空间较大，更加需要用户参与指定空间大小。

对于时间限制，可以通过设定缓存的过期时间来实现，为下载到缓存的数据设定时间戳，在读取该缓存的时候，比较时间戳，超过时间限制的则需要更新该缓存。

在清空应用缓存的时候也需要谨慎，在存储空间已满，用户注销的时候可以考虑清空该用户的整个缓存，而在普通升级应用的情况下并不需要清空整个缓存。

6.4.2 实战案例：下载、缓存和显示图片

在Android网络应用中，下载和显示图片要耗费大量的时间，通过应用缓存机制可以提高用户体验。下面给出一个实战案例。

首先使用6.1节中讨论的AsyncTask方法，在后台下载图片，其次利用缓存机制来显示。图片为地址<http://www.android.com/images/logo.png>，是一张Android的logo图标。

下载之后的文件存放目录如图6-3所示。



图 6-3 图片存放目录

程序运行效果如图6-4所示。



图 6-4 下载、缓存和显示图片运行效果

其核心代码如下：

```
import java.io.File ;  
  
import java.io.FileOutputStream ;  
  
import java.io.InputStream ;  
  
import java.io.OutputStream ;
```

```
import java.net.HttpURLConnection ;  
  
import java.net.URL ;  
  
import java.util.HashMap ;  
  
import android.os.AsyncTask ;  
  
import android.os.Bundle ;  
  
import android.os.Environment ;  
  
import android.app.Activity ;  
  
import android.graphics.drawable.Drawable ;  
  
import android.view.Menu ;  
  
import android.view.View ;  
  
import android.widget.Button ;  
  
import android.widget.EditText ;  
  
import android.widget.ImageView ;  
  
//下载、显示和缓存图片
```

```
public class PictureShowActivity extends Activity{  
  
    //新建URL对象  
  
    URL url ;  
  
    //获取SDCard根目录  
  
    String sdcardPATH=Environment.getExternalStorageDirectory () +"/" ;  
  
    //设置缓存的目录  
  
    String cachePath=sdcardPATH+"pictureCache/" ;  
  
    //新建按钮  
  
    Button showPicture ;  
  
    //新建图片对象  
  
    ImageView pictureView ;  
  
    //新建可编辑文本框  
  
    EditText urlText ;  
  
    HashMap<String,String>cacheMap=new HashMap<String,String>  
    () ;
```

```
//新建File对象
```

```
File file ;
```

```
//新建url字符串
```

```
String urlString ;
```

```
@Override
```

```
protected void onCreate (Bundle savedInstanceState) {
```

```
super.onCreate (savedInstanceState) ;
```

```
//设置布局文件
```

```
setContentView (R.layout.activity_picture_show) ;
```

```
//绑定控件到布局文件的ID上
```

```
pictureView= (ImageView) findViewById (R.id.pictureView) ;
```

```
urlText= (EditText) findViewById (R.id.urlText) ;
```

```
showPicture= (Button) findViewById (R.id.showPicture) ;
```

```
//添加按钮监听
```

```
showPicture.setOnClickListener (new View.OnClickListener () {  
  
    @Override  
  
    public void onClick (View arg0) {  
  
        //获取图片地址  
  
        urlString=urlText.getText ().toString ();  
  
        //输入检测  
  
        if (urlString !=null & & urlString.length () >0) {  
  
            //检测图片地址  
  
            urlString= (urlString.startsWith ("http://"))  
  
?urlString : "http://" +urlString ;  
  
            if (cacheMap !=null & & cacheMap.containsKey (urlString) ) {  
  
                //获取保存的地址  
  
                String cacheFile=cacheMap.get (urlString) .toString ();  
  
                //验证该地址的文档是否存在
```

```
if (checkFileExists (cacheFile) ) {  
  
//存在则显示缓存中的内容  
  
Drawable d=Drawable.createFromPath (cacheFile) ;  
  
pictureView.setImageDrawable (d) ;  
  
}else  
  
{//不存在则显示缓存中的内容  
  
downFile (urlString) ;  
  
}  
  
//存在则直接显示，不存在则先下载  
  
}else{downFile (urlString) ; }  
  
}  
  
}  
  
}) ;  
  
}
```

//下载文件

```
public void downFile (String url)
```

//调用异步下载方法

```
{new DownloadFileTask () .execute (url) ;
```

```
}
```

//判断文件是否已经存在

```
public boolean checkFileExists (String filepath) {
```

//新建文件对象

```
File file=new File (filepath) ;
```

//返回文件是否存在

```
return file.exists () ;
```

```
}
```

//以异步方式下载文件

```
private class DownloadFileTask extends AsyncTask<String,Integer,Integer>{
```

```
//后台下载

@Override

protected Integer doInBackground (String.....sUrl) {

try{

//获取URL地址

String urlString=sUrl[0] ;

//实例化url对象

url=new URL (urlString) ;

//打开连接

HttpURLConnection connection= (HttpURLConnection) url

.openConnection () ;

//获取内容

InputStream istream=connection.getInputStream () ;

//获取文件名称
```

```
String filename=urlString.substring (urlString  
.lastIndexOf ('/') +1) ;  
  
//实例化新的文件目录  
  
File dir=new File (cachePath) ;  
  
//判断文件目录是否存在  
  
if ( ! dir.exists () ) {  
  
//创建新的目录  
  
dir.mkdir () ;  
  
}  
  
//实例化新的文件  
  
file=new File (cachePath+filename) ;  
  
//创建文件  
  
file.createNewFile () ;  
  
//读取信息到文件中
```

```
OutputStream output=new FileOutputStream (file) ;  
  
byte[]buffer=new byte[1024*4] ;  
  
while (istream.read (buffer) !=-1) {  
  
    output.write (buffer) ;  
  
}  
  
//写入文件  
  
output.flush () ;  
  
//关闭输出流  
  
output.close () ;  
  
//关闭数据连接  
  
istream.close () ;  
  
}catch (Exception e) {  
  
    e.printStackTrace () ;  
  
//有异常时返回-1
```

```
        return-1 ;  
  
    }  
  
    //无异常时返回1  
  
    return 1 ;  
  
}  
  
protected void onPostExecute (Integer result) {  
  
    //如果没有异常  
  
    if (result>0) {  
  
        //保存到缓存维护键值对  
  
        cacheMap.put (urlString,file.getPath () ) ;  
  
        Drawable d=Drawable.createFromPath (file.getPath () ) ;  
  
        //显示到界面  
  
        pictureView.setBackgroundDrawable (d) ;  
  
    }  
}
```

```
}
```

```
}
```

//创建Menu菜单

```
@Override
```

```
public boolean onCreateOptionsMenu (Menu menu) {  
    getMenuInflater () .inflate (R.menu.activity_picture_show,menu) ;  
    return true ;
```

```
}
```

```
}
```

配置声明在manifest文件中，其内容如下：

```
<?xml version="1.0"encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
    package="org.chenwen.pictureshow"  
    android:versionCode="1"
```

android:versionName="1.0">

 <!--指定SDK版本的最小值和目标值-->

 <uses-sdk

 android:minSdkVersion="8"

 android:targetSdkVersion="16"/>

 <!--连接网络的权限-->

 <uses-permission android:name="android.permission.INTERNET"/>

 <!--写入存储卡数据的权限-->

 <uses-permission

 android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>

 <application

 android:allowBackup="true"

 android:icon="@drawable/ic_launcher"

 android:label="@string/app_name"

 android:theme="@style/AppTheme">

```
<!--指定启动应用-->
```

```
<activity
```

```
    android:name="org.chenwen.pictureshow.PictureShowActivity"
```

```
    android:label="@string/app_name">
```

```
    <intent-filter>
```

```
        <action android:name="android.intent.action.MAIN"/>
```

```
        <category android:name="android.intent.category.LAUNCHER"/>
```

```
    </intent-filter>
```

```
    </activity>
```

```
    </application>
```

```
</manifest>
```

界面布局如下：

```
<LinearLayout
```

```
    xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    xmlns:tools="http://schemas.android.com/tools"
```

```
    android:layout_width="match_parent"

    android:layout_height="match_parent"

    tools:context=".PictureShowActivity"

    android:orientation="vertical">

<!--使用线性布局-->

<LinearLayout

    android:layout_width="match_parent"

    android:layout_height="wrap_content">

<!--显示文本控件-->

<TextView

    android:id="@+id/textView1"

    android:layout_width="wrap_content"

    android:layout_height="wrap_content"

    android:text="URL :"/>
```

<!--输入控件-->

<EditText

 android:id="@+id/urlText"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"

 android:ems="10"

 android:text="http://www.android.com/images/logo.png"/>

</LinearLayout>

<!--单击按钮-->

<Button

 android:id="@+id/showPicture"

 android:layout_width="match_parent"

 android:layout_height="wrap_content"

 android:text="ShowPicture"/>

```
<!--显示图片控件-->
```

```
<ImageView
```

```
    android:id="@+id/pictureView"
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="50dp"/>
```

```
</LinearLayout>
```

另外显示固定图片的时候，也可以直接指定本地或者系统的图片。

```
<ImageView
```

```
    android:id="@+id/imageView1"
```

```
    android:layout_width="match_parent"
```

```
    android:layout_height="match_parent"
```

```
    android:src="@drawable/ic_launcher"/>
```

这个实战案例比较简单，更加深入的讨论可以参看相关文献。很多时候，应用缓存机制还需要考虑以下几点：缓存的容量不是无限大的，

需要给定一个可调整的值；缓存需要有相应的清理策略；缓存列表不能只放在内存里面，还需要利用数据库等进行长期的有效维护。

6.5 界面更新

应用开始的时候，进入第一个Activity之前，如果这个时候需要显示一些提示，比如提示用户如何使用该软件、如何设置该应用或者显示版本更新信息等，可以考虑使用“开场图片”。此时，通过该图片的视觉效果可以更加有效地提示用户。具体的案例请参考6.1节。接下来两节介绍刷新数据及完成任务时如何进行界面更新。

6.5.1 刷新数据时的界面更新

在同一个Activity内部，当Activity正在刷新的时候，为了避免用户长时间等待，需要给用户一个提示等待的信号，这就是刷新时的提示。

刷新时最常见的提示是标题栏提示，特别适用于需要在主界面显示一些有用信息的场合。常见的方式是使用ProgressBar控件。下面的代码示例显示了用一个进度条，从一个线程来更新进度显示。

//刷新时候以进度条来提示

```
public class MyActivity extends Activity{
```

```
    private static final int PROGRESS=0x1 ;
```

```
    //新建一个ProgressBar控件
```

```
private ProgressBar mProgress ;  
  
//定义状态变量  
  
private int mProgressStatus=0 ;  
  
//新建一个Handler  
  
private Handler mHandler=new Handler () ;  
  
protected void onCreate (Bundle icicle) {  
  
super.onCreate (icicle) ;  
  
//加载布局文件  
  
setContentView (R.layout.progressbar_activity) ;  
  
//绑定进度条到界面ID上  
  
mProgress= (ProgressBar) findViewById (R.id.progress_bar) ;  
  
//后台耗时间操作放到线程里面  
  
new Thread (new Runnable () {  
  
public void run () {
```

```
while (mProgressStatus<100) {  
  
    mProgressStatus=doWork () ;  
  
    //更新ProgressBar  
  
    mHandler.post (new Runnable () {  
  
        public void run () {  
  
            //设置当前进度条的值  
  
            mProgress.setProgress (mProgressStatus) ;  
  
            //开始运行  
  
        }  
  
    }) ;  
  
}  
  
}  
  
}) .start () ;  
  
}
```

}

在布局文件中设置ProgressBar的样式。

```
<ProgressBar
```

```
    android:layout_width="wrap_content"
```

```
    android:layout_height="wrap_content"
```

```
<!--设置样式-->
```

```
    style="@android:style/Widget.ProgressBar.Small"
```

```
    android:layout_marginRight="5dp"/>
```

6.5.2 完成任务时的界面更新

当一个Activity里面刷新或者耗时任务完成之后需要给出完成任务的提示，或者让用户选择接下来做什么。任务完成之后，可以通过调用 startActivityForResult (Intent,int) 启动一个Activity，在结束时返回一些信息，第二个参数int用来标识想要返回的结果。此外，可以通过调用 onActivityResult (int,int,Intent) 来获取返回的信息，第一个int就是定义标识的。在接收返回标识的时候，可以添加界面更新的效果。

示例代码如下：

//完成任务时候给出提示

```
public class MyActivity extends Activity{
```

.....

//定义匹配码

```
static final int PICK_CONTACT_REQUEST=0 ;
```

//监听按钮

```
protected boolean onKeyDown (int keyCode,KeyEvent event) {
```

//按下中间的按钮

```
if (keyCode==KeyEvent.KEYCODE_DPAD_CENTER) {
```

//和onActivityResult配合使用

```
startActivityForResult (
```

//调用联系人相关内容

```
new Intent (Intent.ACTION_PICK,new Uri ("content://contacts") ) ,  
PICK_CONTACT_REQUEST) ;
```

```
return true ;
```

```
}
```

```
return false ;
```

```
}
```

```
//在startActivityForResult调用结果返回之后判断并执行操作
```

```
protected void onActivityResult (int requestCode,int resultCode,Intent  
data) {
```

```
//查看是否匹配
```

```
if (requestCode==PICK_CONTACT_REQUEST) {
```

```
if (resultCode==RESULT_OK) {
```

```
//这里可以添加一些界面提示
```

```
}
```

```
}
```

```
}
```

6.5.3 实战案例：自定义列表显示更新

很多时候我们需要自定义列表显示更新，比如微博里面下拉的时候刷新界面。这里分析一个ListView下拉刷新的案例。给ListView加一个头文件，然后当ListView往下拉，直到头文件出现的时候，就是处理下拉事件的时候。这里我们分析一个广泛使用的开源案例，其开源代码网址：<https://github.com/johannilsson/android-pulltorefresh>。

其运行效果如图6-5所示。

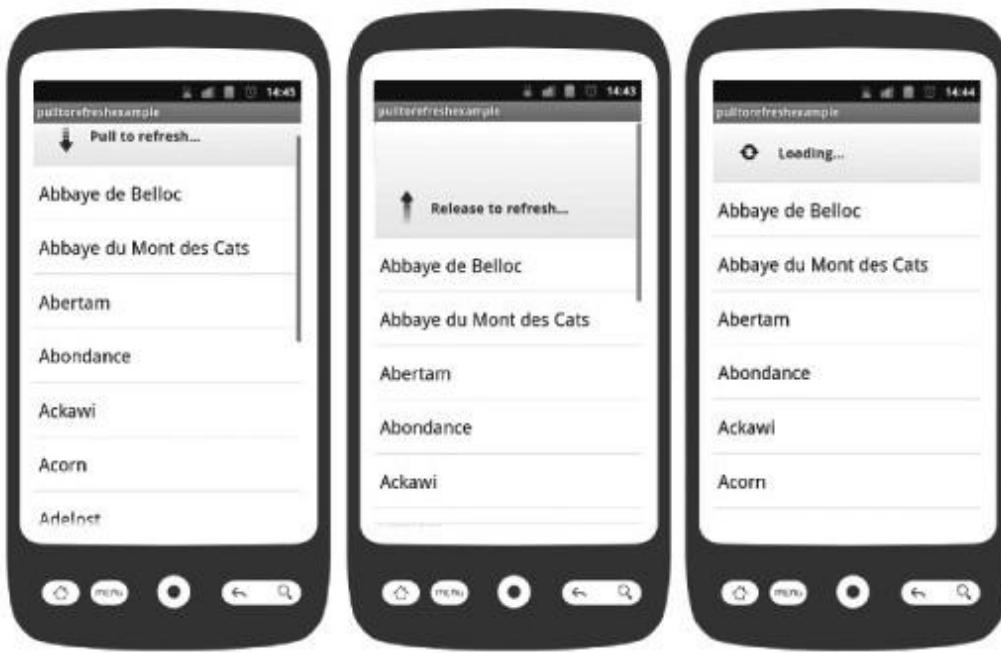


图 6-5 自定义列表显示更新效果图

主要原理为监听触摸和滑动操作，在ListView头部加载一个视图。主要工作如下：

- 写好加载到ListView头部的view。

■重写ListView，实现onTouchEvent（）方法和onScroll（）方法，监听滑动状态。

■计算headview全部显示出来即可实行加载动作，加载完成即刷新列表。

■重新隐藏headview。

其头部的布局文件pull_to_refresh_header.xml的代码如下：

```
<?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:paddingTop="10dip"
    android:paddingBottom="15dip"
    android:gravity="center"

    android:id="@+id/pull_to_refresh_header">
```

<!--定义显示的进度条-->

<ProgressBar

 android:id="@+id/pull_to_refresh_progress"

 android:indeterminate="true"

 android:layout_width="wrap_content"

 android:layout_height="wrap_content"

 android:layout_marginLeft="30dip"

 android:layout_marginRight="20dip"

 android:layout_marginTop="10dip"

 android:visibility="gone"

 android:layout_centerVertical="true"

 style="?android:attr/progressBarStyleSmall"

/>

<!--定义图标控件-->

```
<ImageView  
    android:id="@+id/pull_to_refresh_image"  
  
    android:layout_width="wrap_content"  
  
    android:layout_height="wrap_content"  
  
    android:layout_marginLeft="30dip"  
  
    android:layout_marginRight="20dip"  
  
    android:visibility="gone"  
  
    android:layout_gravity="center"  
  
    android:gravity="center"  
  
    android:src="@drawable/ic_pulltorefresh_arrow"/>  
  
<!--定义显示文本控件-->  
  
<TextView  
    android:id="@+id/pull_to_refresh_text"  
  
    android:text="@string/pull_to_refresh_tap_label"
```

```
    android:textAppearance="?android:attr/textAppearanceMedium"  
  
    android:textStyle="bold"  
  
    android:paddingTop="5dip"  
  
    android:layout_width="fill_parent"  
  
    android:layout_height="wrap_content"  
  
    android:layout_gravity="center"  
  
    android:gravity="center"/>  
  
<!--这个文本显示控件的visibility为gone， 默认不显示也不占用空间-->  
  
<TextView  
  
    android:id="@+id/pull_to_refresh_updated_at"  
  
    android:layout_below="@+id/pull_to_refresh_text"  
  
    android:visibility="gone"  
  
    android:textAppearance="?android:attr/textAppearanceSmall"  
  
    android:layout_width="fill_parent"
```

```
    android:layout_height="wrap_content"

    android:layout_gravity="center"

    android:gravity="center"

/>

</RelativeLayout>
```

其处理用户操作的核心类如下：

```
public class PullToRefreshLayout extends ListView implements
OnScrollListener{

    //初始状态

    private static final int TAP_TO_REFRESH=1 ;

    //拉动刷新

    private static final int PULL_TO_REFRESH=2 ;

    //释放刷新

    private static final int RELEASE_TO_REFRESH=3 ;

    //正在刷新
```

```
private static final int REFRESHING=4 ;  
  
private static final String TAG="PullToRefreshListView" ;  
  
//刷新接口  
  
private OnRefreshListener mOnRefreshListener ;  
  
//监听列表滚动的通知  
  
private OnScrollListener mOnScrollListener ;  
  
private LayoutInflater mInflater ;  
  
private RelativeLayout mRefreshView ;  
  
private TextView mRefreshViewText ;  
  
private ImageView mRefreshViewImage ;  
  
private ProgressBar mRefreshViewProgress ;  
  
private TextView mRefreshViewLastUpdated ;  
  
//当前列表的滚动状态  
  
private int mCurrentScrollState ;
```

//当前列表的刷新状态

private int mRefreshState ;

//动画效果

//变为向下的箭头

private RotateAnimation mFlipAnimation ;

//变为逆向的箭头

private RotateAnimation mReverseFlipAnimation ;

//头视图的高度

private int mRefreshViewHeight ;

//头视图原始的top padding属性值

private int mRefreshOriginalTopPadding ;

private int mLastMotionY ;

//是否反弹标志

private boolean mBounceHack ;

```
public PullToRefreshLayout (Context context) {  
    super (context) ;  
  
    init (context) ;  
  
}  
  
public PullToRefreshLayout (Context context,AttributeSet attrs) {  
    super (context,attrs) ;  
  
    init (context) ;  
  
}  
  
public PullToRefreshLayout (Context context,AttributeSet attrs,int  
defStyle) {  
    super (context,attrs,defStyle) ;  
  
    init (context) ;  
  
}  
  
private void init (Context context) {  
    //初始化动画
```

//从下往上转动的动画

```
mFlipAnimation=new RotateAnimation (0, -180,  
RotateAnimation.RELATIVE_TO_SELF, 0.5f,  
RotateAnimation.RELATIVE_TO_SELF, 0.5f) ;  
  
mFlipAnimation.setInterpolator (new LinearInterpolator () ) ;  
  
mFlipAnimation.setDuration (250) ;  
  
mFlipAnimation.setFillAfter (true) ;
```

//从上往下转动的动画

```
mReverseFlipAnimation=new RotateAnimation (-180, 0,  
RotateAnimation.RELATIVE_TO_SELF, 0.5f,  
RotateAnimation.RELATIVE_TO_SELF, 0.5f) ;  
  
mReverseFlipAnimation.setInterpolator (new LinearInterpolator () ) ;  
  
mReverseFlipAnimation.setDuration (250) ;  
  
mReverseFlipAnimation.setFillAfter (true) ;
```

```
mInflater= (LayoutInflater) context.getSystemService (Context.LAYOUT_INFLATER_SERVICE) ;  
  
//设置头部布局  
  
mRefreshView= (RelativeLayout) mInflater.inflate (R.layout.pull_to_refresh_header,this,false) ;  
  
//初始化头部各控件  
  
mRefreshViewText=  
  
(TextView) mRefreshView.findViewById (R.id.pull_to_refresh_text) ;  
  
mRefreshViewImage=  
  
(ImageView) mRefreshView.findViewById  
(R.id.pull_to_refresh_image) ;  
  
mRefreshViewProgress=  
  
(ProgressBar) mRefreshView.findViewById  
(R.id.pull_to_refresh_progress) ;  
  
mRefreshViewLastUpdated=
```

```
(TextView) mRefreshView.findViewById  
(R.id.pull_to_refresh_updated_at) ;  
  
//单击头部控件刷新  
  
mRefreshViewImage.setMinimumHeight (50) ;  
  
mRefreshView.setOnClickListener (new OnClickListener () ) ;  
  
mRefreshOriginalTopPadding=mRefreshView.getPaddingTop () ;  
  
//状态设置为单击刷新  
  
mRefreshState=TAP_TO_REFRESH ;  
  
//添加头部视图  
  
addHeaderView (mRefreshView) ;  
  
super.setOnScrollListener (this) ;  
  
measureView (mRefreshView) ;  
  
//获取头部的测量高度  
  
mRefreshViewHeight=mRefreshView.getMeasuredHeight () ;  
  
}
```

```
@Override
```

```
protected void onAttachedToWindow () {
```

```
super.onAttachedToWindow () ;
```

```
setSelection (1) ;
```

```
}
```

```
@Override
```

```
public void setAdapter (ListAdapter adapter) {
```

```
super.setAdapter (adapter) ;
```

```
setSelection (1) ;
```

```
}
```

```
//设置列表滚动监听器
```

```
@Override
```

```
public void setOnScrollListener (AbsListView.OnScrollListener l) {
```

```
mOnScrollListener=l ;
```

```
}
```

```
//注册回调函数，当要刷新列表的时候调用
```

```
public void setOnRefreshListener (OnRefreshListener onRefreshListener)
```

```
{
```

```
mOnRefreshListener=onRefreshListener ;
```

```
}
```

```
//设置最后更新时间
```

```
public void setLastUpdated (CharSequence lastUpdated) {
```

```
if (lastUpdated !=null) {
```

```
mRefreshViewLastUpdated.setVisibility (View.VISIBLE) ;
```

```
mRefreshViewLastUpdated.setText (lastUpdated) ;
```

```
}else{
```

```
mRefreshViewLastUpdated.setVisibility (View.GONE) ;
```

```
}
```

```
}
```

```
//实现该方法处理触摸

@Override

public boolean onTouchEvent (MotionEvent event) {

    //当前的y值

    final int y= (int) event.getY () ;

    //设置为不反弹

    mBounceHack=false ;

    switch (event.getAction () ) {

        case MotionEvent.ACTION_UP :

            //将垂直滚动条设置为可用状态

            if ( ! isVerticalScrollBarEnabled () ) {

                setVerticalScrollBarEnabled (true) ;

            }

            //如果头部刷新条出现， 并且不是正在刷新状态
```

```
if (getFirstVisiblePosition () ==0 & &mRefreshState !=REFRESHING) {  
  
//拖动距离达到刷新需要  
  
if ( (mRefreshView.getBottom () >=mRefreshViewHeight  
||mRefreshView.getTop () >=0)  
& &mRefreshState==RELEASE_TO_REFRESH)  
  
//如果头部视图处于拉离顶部的情况  
  
{  
  
//将标量设置为正在刷新  
  
mRefreshState=REFRESHING ;  
  
//准备刷新  
  
prepareForRefresh () ;  
  
//刷新  
  
onRefresh () ;  
  
}else if (mRefreshView.getBottom () <mRefreshViewHeight
```

```
|&gt;mRefreshView.getTop () <=0) {  
    //停止刷新，并且滚动到头部刷新视图的下一个视图  
    resetHeader ();  
    //定位在第二个列表项  
    setSelection (1);  
}  
}  
  
break ;  
  
case MotionEvent.ACTION_DOWN :  
    mLastMotionY=y ; //获得按下y轴位置  
    break ;  
  
case MotionEvent.ACTION_MOVE :  
    applyHeaderPadding (event) ; //计算边距  
    break ;
```

```
}
```

```
return super.onTouchEvent (event) ;
```

```
}
```

```
//更新头视图的边距
```

```
private void applyHeaderPadding (MotionEvent ev) {
```

```
int pointerCount=ev.getHistorySize () ;
```

```
for (int p=0 ; p<pointerCount ; p++) {
```

```
if (mRefreshState==RELEASE_TO_REFRESH) {
```

```
if (isVerticalFadingEdgeEnabled () ) {
```

```
setVerticalScrollBarEnabled (false) ;
```

```
}
```

```
//历史累积的高度
```

```
int historicalY= (int) ev.getHistoricalY (p) ;
```

```
//计算申请的边距，除以1.7使得拉动效果更好
```

```
int topPadding= (int) ( ( (historicalY-mLastMotionY)
-mRefreshViewHeight) /1.7) ;

mRefreshView.setPadding (
mRefreshView.getPaddingLeft () ,
topPadding,
mRefreshView.getPaddingRight () ,
mRefreshView.getPaddingBottom () ) ;

}

}

}

//使头部视图的边距恢复到初始值
```

```
private void resetHeaderPadding () {
mRefreshView.setPadding (
mRefreshView.getPaddingLeft () ,
```

```
mRefreshOriginalTopPadding,  
mRefreshView.getPaddingRight () ,  
mRefreshView.getPaddingBottom () ) ;  
}  
  
//重置头部视图状态  
  
private void resetHeader () {  
if (mRefreshState !=TAP_TO_REFRESH) {  
mRefreshState=TAP_TO_REFRESH ; //初始刷新状态  
  
//使头部视图的边距恢复到初始值  
resetHeaderPadding () ;  
  
//将文字初始化  
mRefreshViewText.setText (R.string.pull_to_refresh_tap_label) ;  
  
//设置初始图片  
mRefreshViewImage.setImageResource  
(R.drawable.ic_pulldownrefresh_arrow) ;
```

```
//清除动画  
mRefreshViewImage.clearAnimation () ;  
  
//隐藏头视图  
mRefreshViewImage.setVisibility (View.GONE) ;  
  
//隐藏进度条  
mRefreshViewProgress.setVisibility (View.GONE) ;  
  
}  
  
}  
  
//计算头部的高度和宽度  
  
private void measureView (View child) {  
    ViewGroup.LayoutParams p=child.getLayoutParams () ;  
    if (p==null) {  
        p=new ViewGroup.LayoutParams (  
            ViewGroup.LayoutParams.FILL_PARENT,  
            ViewGroup.LayoutParams.WRAP_CONTENT);  
        child.setLayoutParams(p);  
    }  
    int width=child.getMeasuredWidth();  
    int height=child.getMeasuredHeight();  
    if (width>parentWidth) parentWidth=width;  
    if (height>parentHeight) parentHeight=height;  
}  
  
//设置头部的宽高  
parentView.setLayoutParams(new ViewGroup.LayoutParams(parentWidth,parentHeight));  
parentView.invalidate();
```

```
        ViewGroup.LayoutParams.WRAP_CONTENT) ;  
  
    }  
  
    int childWidthSpec=ViewGroup.getChildMeasureSpec (0, 0+0,  
p.width) ;  
  
    int lpHeight=p.height ;  
  
    int childHeightSpec ;  
  
    if (lpHeight>0) {  
  
        childHeightSpec=MeasureSpec.makeMeasureSpec  
        (lpHeight,MeasureSpec.EXACTLY) ;  
  
    }else{  
  
        childHeightSpec=MeasureSpec.makeMeasureSpec (0,  
MeasureSpec.UNSPECIFIED) ;  
  
    }  
  
    child.measure (childWidthSpec,childHeightSpec) ;  
  
}
```

//滑动事件

@Override

public void onScroll (AbsListView view,int firstVisibleItem,

int visibleItemCount,int totalItemCount) {

//当刷新视图完全可见的时候，将提示改为“释放来刷新”，并且改变箭头的图标

if (mCurrentScrollState==SCROLL_STATE_TOUCH_SCROLL

& &mRefreshState !=REFRESHING) {

//如果显示出来了第一个列表项

if (firstVisibleItem==0) {

//显示刷新图片

mRefreshViewImage.setVisibility (View.VISIBLE) ;

if ((mRefreshView.getBottom () >=mRefreshViewHeight+20

||mRefreshView.getTop () >=0)

& &mRefreshState !=RELEASE_TO_REFRESH) {

```
//如果下拉了列表，则显示上拉刷新动画  
  
mRefreshViewText.setText (R.string.pull_to_refresh_release_label) ;  
  
mRefreshViewImage.clearAnimation () ;  
  
mRefreshViewImage.startAnimation (mFlipAnimation) ;  
  
mRefreshState=RELEASE_TO_REFRESH ;  
  
}else if (mRefreshView.getBottom () <mRefreshViewHeight+20  
& & mRefreshState !=PULL_TO_REFRESH) {  
  
//如果没有到达下拉刷新距离，则回归原来的状态  
  
mRefreshViewText.setText (R.string.pull_to_refresh_pull_label) ;  
  
if (mRefreshState !=TAP_TO_REFRESH) {  
  
//回弹  
  
mRefreshViewImage.clearAnimation () ;  
  
mRefreshViewImage.startAnimation (mReverseFlipAnimation) ;  
  
}  
}
```

```
mRefreshState=PULL_TO_REFRESH ;  
  
}  
  
}else{  
  
mRefreshViewImage.setVisibility (View.GONE) ; //隐藏刷新图片  
  
resetHeader () ; //初始化头部  
  
}  
  
}else if (mCurrentScrollState==SCROLL_STATE_FLING//快速滑动  
//如果是自己滚动状态+第一个视图已经显示+不是刷新状态  
  
& & firstVisibleItem==0  
  
& & mRefreshState !=REFRESHING) {  
  
setSelection (1) ;  
  
mBounceHack=true ; //状态为回弹  
  
}else if (mBounceHack & &  
mCurrentScrollState==SCROLL_STATE_FLING) {  
  
setSelection (1) ;
```

```
}

if (mOnScrollListener !=null) {

    mOnScrollListener.onScroll (view,firstVisibleItem,
        visibleItemCount,totalItemCount) ;

}

}

//滚动状态改变

@Override

public void onScrollStateChanged (AbsListView view,int scrollState) {

    mCurrentScrollState=scrollState ;

    //当滚动停顿的时候，不回弹

    if (mCurrentScrollState==SCROLL_STATE_IDLE) {

        mBounceHack=false ;

    }

}
```

```
if (mOnScrollListener != null) {  
    mOnScrollListener.onScrollStateChanged (view,scrollState) ;  
}  
  
}  
  
//准备刷新操作  
  
public void prepareForRefresh () {  
  
    //重置头部的边距  
  
    resetHeaderPadding () ;  
  
    mRefreshViewImage.setVisibility (View.GONE) ;  
  
    mRefreshViewImage.setImageDrawable (null) ;  
  
    mRefreshViewProgress.setVisibility (View.VISIBLE) ;  
  
    //显示刷新提示  
  
    mRefreshViewText.setText (R.string.pull_to_refresh_refreshing_label) ;  
  
    mRefreshState=REFRESHING ;
```

}

//刷新

public void onRefresh () {

if (mOnRefreshListener != null) {

mOnRefreshListener.onRefresh () ;

}

}

//刷新完成的回调函数

public void onRefreshComplete (CharSequence lastUpdated) {

setLastUpdated (lastUpdated) ;

onRefreshComplete () ;

}

//刷新完成回调函数

public void onRefreshComplete () {

```
resetHeader () ;  
  
if (mRefreshView.getBottom () >0) {  
  
    invalidateViews () ; //重绘视图  
  
    setSelection (1) ;  
  
}  
  
}  
  
private class OnClickRefreshListener implements OnClickListener{  
  
    @Override  
  
    public void onClick (View v) {  
  
        if (mRefreshState !=REFRESHING) {  
  
            //准备刷新  
  
            prepareForRefresh () ;  
  
            //刷新  
  
            onRefresh () ;  
    }  
}
```

```
}

}

}

//刷新方法接口

public interface OnRefreshListener{

    public void onRefresh ()  ;

}

}
```

下面的代码显示如何刷新调用。

```
public class PullToRefreshActivity extends ListActivity{

    private LinkedList<String>mListItems ;

    @Override

    public void onCreate (Bundle savedInstanceState) {

        super.onCreate (savedInstanceState)  ;
```

```
setContentView (R.layout.pull_to_refresh) ;  
  
//监听列表刷新事件  
  
( (PullToRefreshLayout) getListView () ) .setOnRefreshListener  
(new  
  
nRefreshListener () {  
  
@Override  
  
public void onRefresh () {  
  
//刷新列表  
  
new GetDataTask () .execute () ;  
  
}  
  
}) ;  
  
mListItems=new LinkedList<String> () ;  
  
mListItems.addAll (Arrays.asList (mStrings) ) ;  
  
ArrayAdapter<String>adapter=new ArrayAdapter<String> (this,  
android.R.layout.simple_list_item_1, mListItems) ;
```

```
setListAdapter (adapter) ;  
}  
  
private class GetDataTask extends AsyncTask<Void(Void,Void, String[]> {  
  
    @Override  
  
    protected String[] doInBackground (Void.....params) {  
  
        //此处模拟后台耗时操作  
  
        try{  
  
            Thread.sleep (2000) ;  
  
        }catch (InterruptedException e) {  
  
            ;  
  
        }  
  
        return mStrings ;  
    }  
  
    @Override
```

```
protected void onPostExecute (String[]result) {  
  
    mListItems.addFirst ("Added after refresh.....") ;  
  
    //刷新完成调用回调函数  
  
    ( (PullToRefreshLayout) getListView () ) .onRefreshComplete  
    () ;  
  
    super.onPostExecute (result) ;  
  
}  
  
}  
  
//给出填充数据  
  
private String[]mStrings={  
  
    "Abbaye de Belloc", "Abbaye du Mont des Cats", "Abertam",  
  
    "Abondance", "Ackawi", "Acorn", "Adelost", "Affidelice au  
    Chablis",  
  
    "Afuega'l Pitu", "Airag", "Airedale", "Aisy Cendre",  
  
    "Allgauer Emmentaler"} ;
```

}

在布局文件里面引用上面的控件。

```
<?xml version="1.0" encoding="utf-8"?>

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"

    >

    <com.markupartist.android.widget.PullToRefreshListView
        android:id="@+id/android:list"
        android:layout_height="fill_parent"
        android:layout_width="fill_parent"

        />

    </LinearLayout>
```

6.6 小结

本章涉及Android线程、数据存取、缓存和UI同步等相关知识。在一个进程内部可以有一个或多个线程在同时运行，这种多线程程序的并发性高，可以极大地提高程序的运行效率。本章在介绍Android进程、线程等相关概念的基础上，重点讲解了如何利用AsyncTask实现Android多线程应用开发。Android提供的用于存储永久性应用程序数据的方法主要有5种，即Shared Preferences、Internal Storage、External Storage、SQLite Database及Network Connection。本章详述了前4种方法使用的基本步骤。然后介绍了如何判断用户是否连接、如何判断网络连接的类型以及如何监控网络连接的改变。Android提供的缓存机制是利用本地存储实现的，本章给出了一个下载、缓存和显示图片的示例。本章最后讲解了如何在加载数据前、刷新数据时、完成任务时更新界面的方法，并给出了一个通过自定义列表显示更新界面的案例。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第7章 基于SIP协议的VoIP应用

本章介绍SIP协议的基本知识、SIP服务器的搭建、Android中实现SIP通话的基本方法，并给出了实现SIP通话的案例。

7.1 SIP协议简介

SIP（Session Initiation Protocol，会话发起协议）是一个应用层控制协议，用于建立、修改和终止包括视频、语音、即时通信、在线游戏和虚拟现实等多种多媒体元素在内的交互式用户会话。

SIP最早由Henning Schulzrinne和Mark Handley于1996年设计并提出的。2000年，Internet工程任务组（IETF）发布了第一个SIP规范，即RFC 2543。2001年发布了RFC 3261，即SIP 2.0，对这个协议有了更深入、详细的解释，目前仍然使用这个版本。SIP与H.323一样，是用于VoIP（Voice over Internet Protocol）最主要的信令协议之一。

SIP的设计目标之一是提供类似公用交换电话网（PSTN）中呼叫处理功能的扩展集。在这个扩展集中，实现类似日常电话的操作：拨号、振铃、回铃音或者忙音，只是实现方式和术语有所不同。SIP也实现了许多信令系统7（SS7）中高级的呼叫处理功能，尽管这两个协议相差很远。SS7是一个高度集中处理的协议，其特点表现为高复杂度的中心网络结构和无智能的哑终端（传统的电话机）。SIP则是一个点对点协议，所以它只需要一个相对简单的（因此也高度可扩展的）核心网络，而将处理工作下放给连接在网络边缘的智能端点（装有硬件或软件的终端设备）。SIP的许多功能在端点中实现，这与传统的SS7将其在网络核心设备实现的作法是不同的。

SIP不是万能的。它既不是会话描述协议，也不提供会议控制功能。为了描述消息内容的负载情况和特点，SIP使用Internet的会话描述协议（SDP）来描述终端设备的特点。SIP自身也不提供服务质量（QoS），它与负责语音质量的资源保留设置协议（RSVP）互操作。SIP与若干个其他协议进行协作，包括负责定位的轻型目录访问协议（LDAP）、负责身份验证的远程身份验证拨入用户服务（RADIUS）以及负责实时传输的RTP等多个协议。

Android SDK 2.3开始提供支持SIP的API，可以让用户将基于SIP的网络电话功能添加到其应用程序中。Android包括一个完整的SIP协议栈和集成的呼叫管理服务，让应用轻松设置语音通话等。因此，在开发视频会议、即时消息等应用程序的时候可能会用到Android SIPAPI。

以下是开发一个Android SIP应用程序的必要条件：

- 必须有一个运行Android 2.3或者更高版本的移动设备。
- SIP是通过无线数据连接来运行的，所以设备必须有一个数据连接（通过移动数据服务或者Wi-Fi）。这意味着不能在模拟器（AVD）上进行测试，只能在一个物理设备上测试。
- 每一个参与者在应用程序的通信会话过程中必须有一个SIP账户（有很多不同的SIP服务提供商提供SIP账户）。

7.2 SIP服务器搭建

高效的SIP服务器有很多，我们这里介绍的服务器是Brekeke SIP Server，其下载地址为<http://www.brekeke.com/downloads/sip-server.php>。

下载sip3_0_7_0.exe文件之后，双击弹出如图7-1所示的安装向导对话框。



图 7-1 Brekeke SIP Server的安装向导

7.2.1 下载安装Brekeke SIP Server

单击Next按钮，弹出如图7-2所示对话框，这里我们选择同意相关协议。

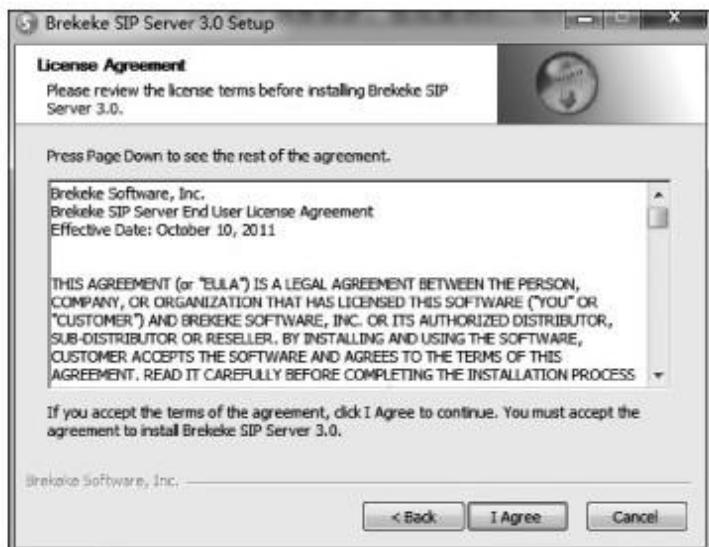


图 7-2 同意协议界面

选择安装默认的组件还是自定义的组件，如图7-3所示。

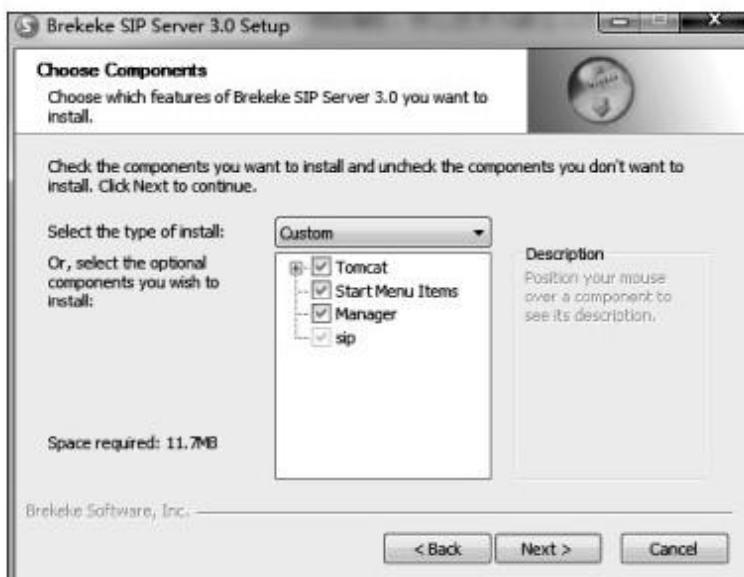


图 7-3 选择安装组件

设置相关的端口，如图7-4所示。



图 7-4 设置相关端口

选择JRE的安装路径，如图7-5所示。

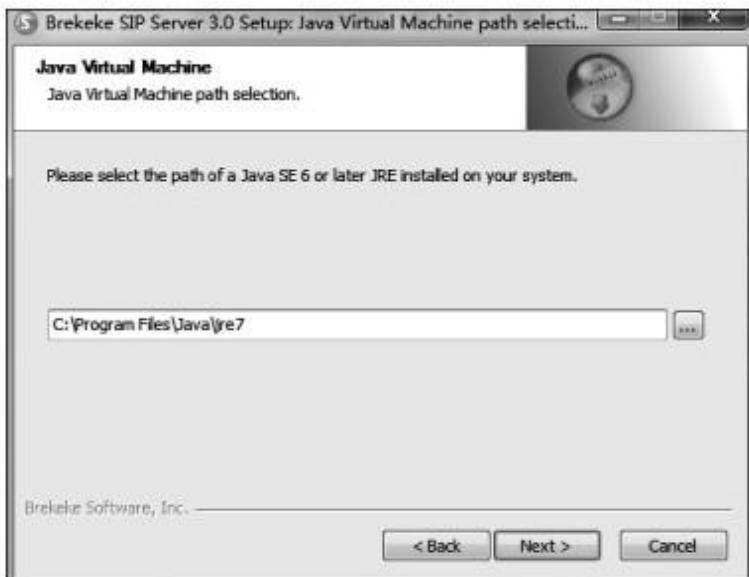


图 7-5 选择JRE安装路径

选择SIP服务器的安装路径，如图7-6所示。



图 7-6 选择SIP服务器安装路径

7.2.2 访问服务器

安装完成之后，从地址http://127.0.0.1:18080/sip/gate?bean=sipadmin.web.Login访问服务器。此时需要产品的ID，单击页面左下角的“Need a Product ID？”链接，如图7-7所示。根据其提示可以获得产品ID，当然这里我们选择的是试用的ID。



图 7-7 获取产品ID

将获得的ID输入之后，可以看到下面的登录界面，用户ID和密码在默认情况下都为sa，如图7-8所示。



图 7-8 登录服务器

成功登录之后，选择Status → Server Status可以看到服务器状态，如图7-9所示。当前为INACTIVE，此时需要启动服务器。

SIP Server Status	
Status	INACTIVE

图 7-9 查看服务器状态

7.2.3 启动服务器

单击Start/Shutdown下面的Start按钮，可以启动服务器，如图7-10所示。



图 7-10 启动服务器

启动成功，界面如图7-11所示。

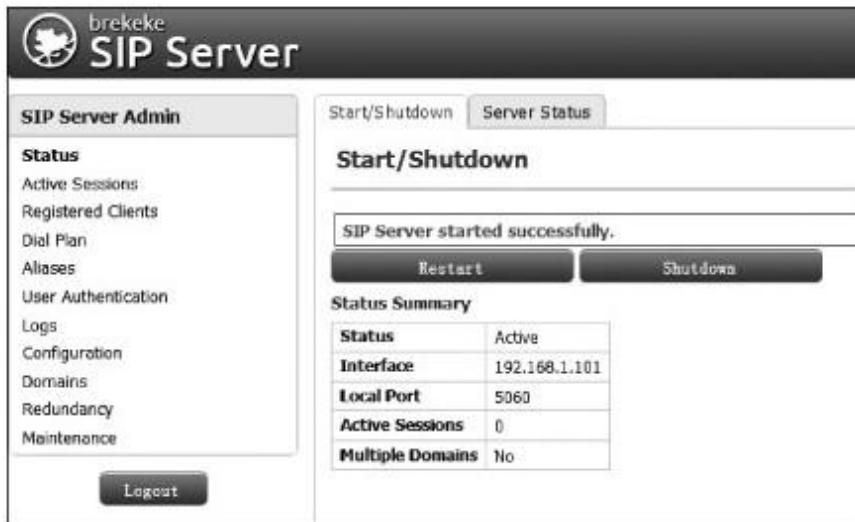


图 7-11 启动服务器成功后的界面

在User Authentication → User Authentication选项下可以添加新用户，如图7-12所示。

The screenshot shows the 'User Authentication' section of the administration interface. Under 'User Authentication', there is a link to 'User Authentication >>'. The main panel is titled 'New User' and contains fields for adding a new user:

User	<input type="text" value="bbb"/>
Password	<input type="password" value="●●●"/>
Confirm Password	<input type="password" value="●●●"/>
Name	<input type="text"/>
Email Address	<input type="text"/>
Description	<input type="text"/>

At the bottom right of the form is a large 'Add' button.

图 7-12 添加新用户

这里我们分别添加用户名为aaa和bbb的两个用户。

7.3 SIP程序设置

Android从SDK 2.3开始包含一个SIP协议栈和API框架，主要类位于 android.net.sip包中，其中最重要的是SipMaimger类。

7.3.1 Android SIP API 中的类和接口

表7-1给出了Android SIP APT中主要的类和接口的概述。

表 7-1 Android SIP API 中主要的类和接口

类 / 接口	功 能 描 述
SipAudioCall	通过 SIP 处理网络音频电话
SipAudioCall.Listener	关于 SIP 电话的事件监听器，比如接到一个电话（on ringing）或者呼出一个电话（on calling）的时候
SipErrorCode	定义在 SIP 活动中返回的错误代码
SipManager	为 SIP 任务提供 API，比如初始化一个 SIP 连接、提供相关 SIP 服务的访问
SipProfile	定义了 SIP 的相关属性，包含 SIP 账户、域名和服务器信息
SipProfile.Builder	创建 SipProfile 的帮助类
SipSession	代表一个 SIP 会话，跟 SIP 对话框或者一个没有对话框的独立事务相关联
SipSession.Listener	关于 SIP 会话的事件监听器，比如注册一个会话（on registering）或者呼出一个电话（on calling）的时候
SipSession.State	定义 SIP 会话的声明，比如“注册”、“呼出电话”、“打入电话”
SipRegistrationListener	一个关于 SIP 注册事件监听器的接口

7.3.2 Android极限列表

下面是需要用到的permission列表，其中后面3个为使用Wi-Fi获取IP地址所要用到的权限。

- 1) 允许程序访问网络连接。

```
<uses-permission android:name="android.permission.INTERNET"/>
```

2) 运行程序使用SIP连接。

```
<uses-permission android:name="android.permission.USE_SIP"/>
```

3) 允许程序访问Wi-Fi网络状态信息。

```
<uses-permission  
    android:name="android.permission.ACCESS_WIFI_STATE"/>
```

4) 允许程序在手机屏幕关闭后后台进程仍然运行。

```
<uses-permission android:name="android.permission.WAKE_LOCK"/>
```

5) 允许通过手机或耳机的麦克录制声音。

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/  
>
```

不是所有设备都提供SIP支持，确保APP只安装在支持SIP的装置上，程序中可以用下面的代码进行检测：

```
//检测当前设备是否支持VOIP通话
```

```
Boolean voipSupported=SipManager.isVoipSupported (this) ;
```

//检测当前设备是否支持SIP的API

```
Boolean apiSupported=SipManager.isApiSupported (this) ;
```

添加<uses-sdk android:minSdkVersion="9"/>到应用程序的manifest文件里，该设置表明应用程序需要Android 2.3或者更高版本的平台，可以确保应用程序能够安装到支持SIP的设备上。

添加<uses-feature android:name="android.hardware.sip.voip"/>到应用程序的manifest文件里，不支持SIP的设备可以在市场里（比如Google Play）过滤掉。

在应用程序的manifest文件里定义一个广播接收器（BroadcastReceiver的子类）接收呼叫。

```
<receiver android:name=".IncomingCallReceiver" android:label="Call  
Receiver"/>
```

7.3.3 完整的Manifest文件

以下是manifest文件完整的内容：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"  
package="com.example.android.sip">
```

<!--设置Activity的
 android:configChanges="orientation|keyboardHidden"时，切屏不会重新
 调用各个生命周期，只会执行onConfigurationChanged方法-->

```
<application android:icon="@drawable/icon" android:label="SipDemo">  
  
    <activity android:name=".WalkieTalkieActivity"  
  
        android:configChanges="orientation|keyboardHidden">  
  
        <intent-filter>  
  
            <action android:name="android.intent.action.MAIN"/>  
  
            <category android:name="android.intent.category.LAUNCHER"/>  
  
        </intent-filter>  
  
    </activity>  
  
    <activity android:name=".SipSettings" android:label="set_preferences"/>  
  
    <receiver android:name=".IncomingCallReceiver" android:label="Call  
    Receiver"/>  
  
</application>
```

<!--指明该应用程序可以运行的API最低版本，这里指定为9，默认是1-->

<uses-sdk android:minSdkVersion="9"/>

<!--允许程序使用SIP视频服务-->

<uses-permission android:name="android.permission.USE_SIP"/>

<!--访问网络连接，可能产生GPRS流量-->

<uses-permission android:name="android.permission.INTERNET"/>

<!--允许振动-->

<uses-permission android:name="android.permission.VIBRATE"/>

<!--允许程序访问Wi-Fi网络状态信息-->

<uses-permission
 android:name="android.permission.ACCESS_WIFI_STATE"/>

<!--允许程序在手机屏幕关闭后后台进程仍然运行-->

<uses-permission android:name="android.permission.WAKE_LOCK"/>

<!--通过手机或耳机的麦克运行录制声音-->

```
<uses-permission android:name="android.permission.RECORD_AUDIO"/>

<!--应用程序需要使用设备上SIP相关的API-->

<uses-feature
    android:name="android.hardware.sip.voip" android:required="true"/>

<!--应用程序要能使用设备上的wi-fi功能。-->

<uses-feature
    android:name="android.hardware.wifi" android:required="true"/>

<!--应用程序要能使用设备上的麦克风-->

<uses-feature
    android:name="android.hardware.microphone" android:required=
    "true"/>

</manifest>
```

7.4 SIP初始化通话

SIP初始化通话的时候需要SipManager对象负责发起SIP呼叫，需要SipProfile对象进行通话配置。下面分别讨论这两个对象。

7.4.1 SipManager对象

SipManager对象可以发起SIP会话、发起和接受呼叫、在SIP provider里进行注册和注销、验证会话的连通性等。

以下代码的功能是实例化一个新的SipManager对象。

```
//创建空的SipManager对象
```

```
public SipManager mSipManager=null ;
```

```
//检测当前SipManager是否为空
```

```
if (mSipManager==null) {
```

```
//实例化SipManager对象
```

```
mSipManager=SipManager.newInstance (this) ;
```

```
}
```

7.4.2 SipProfile对象

一个典型的Android SIP应用中包含一个或多个用户，他们中的每个人都有一个SIP账户。在Android SIP中，每一个SIP账户代表一个SipProfile对象。一个SipProfile对象定义了一个SIP的概要文件，包括SIP账户、域名和服务器信息。在终端设备上运行应用的SIP账户相关的概要文件称为本地配置文件，与会话相连接的概要文件称为对应配置文件。当SIP应用通过本地SipProfile登录SIP服务器的时候，其内容也就是注册当前设备为基站，并发送SIP呼叫到指定的SIP地址。

下面展示了如何创建一个SipProfile，以及如何把创建的SipProfile注册到SIP服务器上，并且跟踪注册事件。

以下代码的功能是创建一个SipProfile对象：

```
//创建SipProfile对象
```

```
public SipProfile mSipProfile=null ;
```

```
//实用用户名和服务器地址做为参数
```

```
SipProfile.Builder builder=new SipProfile.Builder (username, domain) ;
```

```
//给出密码
```

```
builder.setPassword (password) ;
```

```
//实例化SipProfile对象
```

```
mSipProfile=builder.build () ;
```

接下来的代码设置了一个标签为android.SipDemo.INCOMING_CALL的意图，这个意图会被一个意图过滤器使用。

```
//新建Intent对象
```

```
Intent intent=new Intent () ;
```

```
//将android.SipDemo.INCOMING_CALL行为放到Intent中
```

```
intent.setAction ("android.SipDemo.INCOMING_CALL") ;
```

```
//使用PendingIntent对象延后调用
```

```
PendingIntent pendingIntent=PendingIntent.getBroadcast (this, 0,  
intent,
```

```
Intent.FILL_IN_DATA) ;
```

```
//调用方法处理该意图
```

```
mSipManager.open (mSipProfile,pendingIntent,null) ;
```

最后这段代码在SipManager上设置了一个SipRegistrationListener监听器，这个监听器会跟踪SipProfile，来确定是否成功地注册到SIP服务器。

//设置注册监听器

```
mSipManager.setRegistrationListener (mSipProfile.getUriString () , new  
SipRegistrationListener () {
```

```
public void onRegistering (String localProfileUri) {
```

//正在和SIP服务器通信

```
updateStatus ("Registering with SIP Server.....") ;
```

```
}
```

```
public void onRegistrationDone (String localProfileUri,long expiryTime)
```

```
{
```

//在SIP服务器上注册完成

```
updateStatus ("Ready") ;
```

```
}
```

```
public void onRegistrationFailed (String localProfileUri,int errorCode,
```

```
String errorMessage) {  
  
    //在SIP服务器上注册失败  
  
    updateStatus ("Registration failed.Please check settings.") ;  
  
}  
  
}
```

当应用程序使用完一个profile的时候，应该关闭它以便释放相关联的对象，并从服务器上注销当前设备。代码如下：

```
//注销当前设备  
  
public void closeLocalProfile () {  
  
    //检测SipManager对象是否为空  
  
    if (mSipManager==null) {  
  
        //如果为空，则不进行任何操作，直接返回  
  
        return ;  
  
    }  
  
    try{
```

```
//不为空的情况

if (mSipProfile !=null) {

//关闭相关对象

mSipManager.close (mSipProfile.getUriString () ) ;

}

}catch (Exception ee) {

//发现异常

Log.d ("WalkieTalkieActivity/onDestroy", "Failed to close local
profile.", ee) ;

}

}
```

7.5 监听SIP通话

大部分客户与SIP堆栈的交互都是通过监听器来完成的，所以在拨打SIP语音电话的时候，需要建立一个SipAudioCall.Listener监听器。

7.5.1 创建监听器

下面的代码，展示了如何创建一个新的SipAudioCall.Listener监听器。

```
//创建一个新的SipAudioCall.Listener监听器
SipAudioCall.Listener listener=new SipAudioCall.Listener () {
    @Override
    public void onCallEstablished (SipAudioCall call) {
        //对已经建立的呼叫启动音频
        call.startAudio () ;
        //设备设置为扬声器模式
        call.setSpeakerMode (true) ;
        //静音切换
    }
}
```

```
call.toggleMute () ;
```

```
}
```

//会话终止时候调用

```
@Override
```

```
public void onCallEnded (SipAudioCall call) {
```

```
}
```

```
} ;
```

SipAudioCall.Listener监听器中一些重要的处理方法如表7-2所示。

表 7-2 SipAudioCall.Listener 监听器中重要的方法列表

方 法	含 义
onCallBusy (SipAudioCall call)	会话 (session) 建立过程中, 站点忙时被调用
onCallEnded (SipAudioCall call)	会话终止时被调用
onCallEstablished (SipAudioCall call)	会话建立时被调用
onCallHeld (SipAudioCall call)	呼叫保持 (call is on hold) 时被调用
onCalling (SipAudioCall call)	当一个请求被发出用于初始化一个新连接时被调用
onChanged (SipAudioCall call)	当一个事件发生且相应的回调函数未被覆盖时调用
onError (SipAudioCall call, int errorCode, String errorMessage)	错误发生时被调用
onReadyToCall (SipAudioCall call)	呼叫目标准备好建立呼叫时被调用
onRinging (SipAudioCall call, SipProfile caller)	一个新的呼叫进来时被调用
onRingingBack (SipAudioCall call)	INVITE 请求接收到振铃响应

7.5.2 拨打电话

一旦创建了SipAudioCall.Listener监听器，就可以拨打电话了，需要用到SipAudioCall的makeAudioCall方法。

//拨打电话

```
makeAudioCall (SipProfile localProfile,SipProfile  
peerProfile,SipAudioCall.Listener listener,int timeout)
```

其中，参数localProfile代表本地SIP配置文件（呼叫方）；peerProfile为相对应的SIP配置文件（被呼叫方）；listener用来监听从SipAudioCall发出的呼叫事件，这个参数可以为null；timeout是超时时间，以秒为单位。

完整的例子如下：

```
call=mSipManager.makeAudioCall (mSipProfile.getUriString () ,  
sipAddress,listener, 30) ;
```

7.5.3 接收呼叫

为了接收呼叫，SIP应用程序必须包含一个BroadcastReceiver的子类，这个子类要有能力响应一个表明有来电的Intent。

首先是实现BroadcastReceiver的子类。

当Android系统接收到一个呼叫的时候，会处理这个SIP呼叫，然后广播一个来电Intent（这个Intent由系统来定义），以下是示例中实现BroadcastReceiver子类的代码。

```
//监听并拦截SIP电话给WalkieTalkieActivity.
```

```
public class IncomingCallReceiver extends BroadcastReceiver{
```

```
@Override
```

```
//接收到通话
```

```
public void onReceive (Context context,Intent intent) {
```

```
//创建一个空的对象
```

```
SipAudioCall incomingCall=null ;
```

```
try{//新建监听器
```

```
SipAudioCall.Listener listener=new SipAudioCall.Listener () {
```

```
@Override
```

```
//有一个新的呼叫
```

```
public void onRinging (SipAudioCall call,SipProfile caller) {
```

```
try{  
  
    //答复一个来电，参数是超时时间，超时时间以秒为单位  
  
    //如果超时值为零或负数，则使用SIP协议定义的默认值  
  
    call.answerCall (30) ;  
  
}catch (Exception e) {  
  
    //打印异常  
  
    e.printStackTrace () ;  
  
}  
  
}  
  
};  
  
//WalkieTalkieActivity负责处理SIP来电、打电话、UI更新等  
  
WalkieTalkieActivity wtActivity= (WalkieTalkieActivity) context ;  
  
//接听电话  
  
incomingCall=wtActivity.mSipManager.takeAudioCall (intent,listener) ;
```

//回答一个呼叫，参数为超时的时间

```
incomingCall.answerCall (30) ;
```

//为建立连接开启音频

```
incomingCall.startAudio () ;
```

//开启为扬声器模式

```
incomingCall.setSpeakerMode (true) ;
```

//检查是否为静音状态

```
if (incomingCall.isMuted ()) {
```

//切换静音

```
incomingCall.toggleMute () ;
```

```
}
```

//赋值设置参数

```
wtActivity.call=incomingCall ;
```

//更新状态

```
wtActivity.updateStatus (incomingCall) ;  
}  
catch (Exception e) {  
if (incomingCall !=null) {  
//关闭  
incomingCall.close () ;  
}  
}  
}  
}  
}
```

其次，通过挂起一个Intent来初始化本地配置文件（SipProfile），当有人呼叫时候，这个挂起的Intent会调用接收器。

当SIP服务接收到一个新的呼叫时，会发送一个Intent，这个Intent会附带一个由应用程序提供的action。以下代码通过挂起一个基于 android.SipDemo.INCOMING_CALL action的Intent来创建SipProfile对象的。当SipProfile接收到一个呼叫的时候PendingIntent对象将执行一个广播。

//创建一个空的SipManager对象

```
public SipManager mSipManager=null ;
```

//创建一个空的SipProfile对象

```
public SipProfile mSipProfile=null ;
```

//创建Intent，设置行为

```
Intent intent=new Intent () ;
```

```
intent.setAction ("android.SipDemo.INCOMING_CALL") ;
```

//设置延时响应

```
PendingIntent pendingIntent=PendingIntent.getBroadcast (this, 0,
```

```
intent,
```

```
Intent.FILL_IN_DATA) ;
```

//发起一个被动会话请求

```
mSipManager.open (mSipProfile,pendingIntent,null) ;
```

最后，创建一个用来接收呼叫的Intent过滤器。上面的广播消息如果被Intent过滤器拦截的话，这个Intent过滤器将会启动声明过的Receiver

(IncomingCallReceiver)。其代码如下：

//登录到SIP设备供应商，注册device去处理来电，拨打电话，在通话过程中用户界面管理

```
public class WalkieTalkieActivity extends Activity implements
```

```
View.OnTouchListener{
```

//IncomingCallReceiver负责监听传入的SIP电话，

//然后传递这些SIP电话给WalkieTalkieActivity控制

```
public IncomingCallReceiver callReceiver ;
```

```
@Override
```

```
public void onCreate (Bundle savedInstanceState) {
```

//新建一个IntentFilter

```
IntentFilter filter=new IntentFilter () ;
```

//设置需要响应的行为

```
filter.addAction ("android.SipDemo.INCOMING_CALL") ;
```

//新建一个IncomingCallReceiver对象

```
callReceiver=new IncomingCallReceiver () ;
```

```
//注册接收器
```

```
this.registerReceiver (callReceiver,filter) ;
```

```
}
```

```
}
```

7.6 实战案例：SIP通话

本案例中，WalkieTalkieActivity负责处理SIP来电、打电话、UI更新等；SipSettings则负责应用的设置、身份验证等。SipSettings调用res/xml/preferences.xml文件来包含属性定义。

SipSettings类定义代码如下：

//处理SIP认证设置

```
public class SipSettings extends PreferenceActivity{  
  
    @Override  
  
    public void onCreate (Bundle savedInstanceState) {  
  
        super.onCreate (savedInstanceState) ;  
  
        //从资源文件加载设置界面  
  
        addPreferencesFromResource (R.xml.preferences) ;  
  
    }  
  
}
```

res/xml/preferences.xml中内容如下：

<!--定义设置界面-->

<PreferenceScreen

 xmlns:android="http://schemas.android.com/apk/res/android" >

<!--输入SIP用户名-->

<EditTextPreference

 android:name="SIP Username"

 android:summary="Username for your SIP Account"

 android:defaultValue=""

 android:title="Enter Username"

 android:key="namePref"/>

<!--输入SIP服务器地址-->

<EditTextPreference

 android:name="SIP Domain"

 android:summary="Domain for your SIP Account"

```
    android:defaultValue=""  
  
    android:title="Enter Domain"  
  
    android:key="domainPref"/>  
  
<!--输入SIP密码-->  
  
<EditTextPreference  
  
    android:name="SIP Password"  
  
    android:summary="Password for your SIP Account"  
  
    android:defaultValue=""  
  
    android:title="Enter Password"  
  
    android:key="passPref"  
  
    android:password="true"/>  
  
</PreferenceScreen>
```

WalkieTalkieActivity的代码如下：

```
//处理SIP来电，打电话，UI更新
```

```
public class WalkieTalkieActivity extends Activity implements  
View.OnTouchListener{  
  
    //定义sipAddress字符串，保存地址  
  
    public String sipAddress=null ;  
  
    //新建SipManager对象  
  
    public SipManager manager=null ;  
  
    //新建SipProfile对象  
  
    public SipProfile me=null ;  
  
    //新建SipAudioCall对象  
  
    public SipAudioCall call=null ;  
  
    //新建IncomingCallReceiver对象  
  
    public IncomingCallReceiver callReceiver ;  
  
    //状态标志，用于对话框中  
  
    //拨号地址  
  
    private static final int CALL_ADDRESS=1 ;
```

```
//认证信息  
  
private static final int SET_AUTH_INFO=2 ;  
  
//更新设置  
  
private static final int UPDATE_SETTINGS_DIALOG=3 ;  
  
//挂断  
  
private static final int HANG_UP=4 ;  
  
@Override  
  
public void onCreate (Bundle savedInstanceState) {  
    super.onCreate (savedInstanceState) ;  
  
    //加载布局文件  
  
    setContentView (R.layout.walkietalkie) ;  
  
    //谈话按钮  
  
    ToggleButton pushToTalkButton= (ToggleButton)  
        findViewById (R.id.pushToTalk) ;
```

```
pushToTalkButton.setOnTouchListener (this) ;  
  
//设置Intent过滤器。当有广播要求使用SIP地址的应用程序时  
  
//将触发一个IncomingCallReceiver  
  
IntentFilter filter=new IntentFilter () ;  
  
//添加行为  
  
filter.addAction ("android.SipDemo.INCOMING_CALL") ;  
  
//监听传入的SIP电话  
  
callReceiver=new IncomingCallReceiver () ;  
  
//注册接收器  
  
this.registerReceiver (callReceiver,filter) ;  
  
//禁止黑屏进入省电模式  
  
getWindow () .addFlags  
( WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON) ;  
  
//初始化SipManager  
  
initializeManager () ;
```

```
}

@Override

public void onStart () {

super.onStart () ;

//重新进入界面的时候，调用initializeManager () 完成初始化

initializeManager () ;

}

@Override

public void onDestroy () {

super.onDestroy () ;

//销毁的时候关闭通话

if (call !=null) {

//关闭通话

call.close () ;


```

```
}

//关闭本地的配置文件，在服务器上注销该设备

closeLocalProfile () ;

if (callReceiver !=null) {

//取消广播接收器注册

this.unregisterReceiver (callReceiver) ;

}

}

//初始化SipManager

public void initializeManager () {

if (manager==null) {

//实例化SipManager

manager=SipManager.newInstance (this) ;

}

}
```

```
//发送信息

initializeLocalProfile () ;

}

//登录到指定的SIP提供商，发送相关信息给服务器

public void initializeLocalProfile () {

//检测SipManager是否初始化成功，不成功则退出

if (manager==null) {

return ;

}

//如果已经登录，则需要关闭连接

if (me !=null) {

closeLocalProfile () ;

}

//保存用户名和密码、SIP服务器地址
```

```
//获取SharedPreferences对象
```

```
SharedPreferences prefs=
```

```
PreferenceManager.getDefaultSharedPreferences (getBaseContext  
(() ) ;
```

```
//获取namePref对应的值
```

```
String username=prefs.getString ("namePref", "") ;
```

```
//获取domainPref对应的值
```

```
String domain=prefs.getString ("domainPref", "") ;
```

```
//获取passPref对应的值
```

```
String password=prefs.getString ("passPref", "") ;
```

```
//检查用户名、密码和SIP地址
```

```
if (username.length () ==0||domain.length () ==0||
```

```
password.length () ==0) {
```

```
//显示提示信息
```

```
showDialog (UPDATE_SETTINGS_DIALOG) ;
```

```
return ;  
  
}  
  
try{  
    //构造SipProfile  
  
    SipProfile.Builder builder=new SipProfile.Builder (username, domain) ;  
  
    //设置密码  
  
    builder.setPassword (password) ;  
  
    //构建SipProfile对象  
  
    me=builder.build () ;  
  
    //新建意图  
  
    Intent i=new Intent () ;  
  
    //设置行为  
  
    i.setAction ("android.SipDemo.INCOMING_CALL") ;  
  
    //延时处理
```

```
PendingIntent pi=PendingIntent.getBroadcast (this, 0, i,Intent.FILL_IN_DATA) ;  
  
manager.open (me,pi,null) ;  
  
//设置登录监听  
  
manager.setRegistrationListener (me.getUriString () ,  
  
//新建一个注册SIP的监听器  
  
new SipRegistrationListener () {  
  
//在注册过程中  
  
public void onRegistering (String localProfileUri) {  
  
//更新状态信息  
  
updateStatus ("Registering with SIP Server.....") ;  
  
}  
  
//注册完成  
  
public void onRegistrationDone (String localProfileUri,
```

```
long expiryTime) {  
  
    //更新状态信息  
  
    updateStatus ("Ready") ;  
  
}  
  
//注册失败  
  
public void onRegistrationFailed (String localProfileUri,  
int errorCode,String errorMessage) {  
  
    //更新状态信息  
  
    updateStatus ("Registration failed.Please check settings.") ;  
  
}  
});  
  
}catch (ParseException pe) {  
  
    //解析异常，注册失败  
  
    updateStatus ("Connection Error.") ;  
}
```

```
 }catch (SipException se) {  
  
    //SIP连接异常  
  
    updateStatus ("Connection error.");  
  
}  
  
}  
  
//关闭本地的配置文件，在服务器上注销该设备  
  
public void closeLocalProfile () {  
  
    //如果SipManager为空  
  
    if (manager==null) {  
  
        //直接返回  
  
        return ;  
  
    }  
  
    try{  
  
        //如果SipProfile不为空
```

```
if (me !=null) {  
  
    //在服务器上注销对应的设备  
  
    manager.close (me.getUriString () ) ;  
  
}  
  
}  
  
}catch (Exception ee) {  
  
    //错误处理  
  
    Log.d ("WalkieTalkieActivity/onDestroy",  
    "Failed to close local profile.", ee) ;  
  
}  
  
}  
  
}  
  
//拨出一个SIP电话  
  
public void initiateCall () {  
  
    //更新状态，显示拨号地址  
  
    updateStatus (sipAddress) ;
```

```
try{  
  
    //新建监听，进行音频呼叫  
  
    SipAudioCall.Listener listener=new SipAudioCall.Listener () {  
  
        //会话建立  
  
        @Override  
  
        public void onCallEstablished (SipAudioCall call) {  
  
            //对已经建立的呼叫启动音频  
  
            call.startAudio () ;  
  
            //设备设置为扬声器模式  
  
            call.setSpeakerMode (true) ;  
  
            //静音切换  
  
            call.toggleMute () ;  
  
            //更新状态  
  
            updateStatus (call) ;
```

```
}
```

```
//会话终止
```

```
@Override
```

```
public void onCallEnded (SipAudioCall call) {
```

```
    updateStatus ("Ready.") ;
```

```
}
```

```
} ;
```

```
//拨打电话
```

```
call=manager.makeAudioCall (me.getUriString () ,
```

```
sipAddress,listener, 30) ;
```

```
}
```

```
catch (Exception e) {
```

```
//报出异常
```

```
Log.d ("WalkieTalkieActivity/InitiateCall",
```

```
"Error when trying to close manager.", e) ;
```

```
//如果SipProfile不为空

if (me !=null) {

try{

//在服务器上注销对应的设备

manager.close (me.getUriString () ) ;

}catch (Exception ee) {

//报出异常

Log.d ("WalkieTalkieActivity/InitiateCall",

"Error when trying to close manager.", ee) ;

ee.printStackTrace () ;

}

}

//如果电话在通话

if (call !=null) {
```

```
//关闭会话
```

```
call.close () ;
```

```
}
```

```
}
```

```
}
```

```
//更新界面状态
```

```
public void updateStatus (final String status) {
```

```
//在线程中更新界面
```

```
this.runOnUiThread (new Runnable () {
```

```
public void run () {
```

```
//根据内容设置状态信息
```

```
TextView labelView= (TextView) findViewById (R.id.sipLabel) ;
```

```
labelView.setText (status) ;
```

```
}
```

```
}) ;  
  
}  
  
//根据呼叫情况更新界面状态  
  
public void updateStatus (SipAudioCall call) {  
  
    //获取对方显示名称  
  
    String useName=call.getPeerProfile () .getDisplayName () ;  
  
    //如果名称为空，则调用使用者名称  
  
    if (useName==null) {  
  
        useName=call.getPeerProfile () .getUserName () ;  
  
    }  
  
    //更新界面  
  
    updateStatus (useName+"@"+call.getPeerProfile () .getSipDomain  
    () ) ;  
  
}  
  
//用户通过按钮来确定是否在静音状态
```

```
public boolean onTouch (View v,MotionEvent event) {  
  
    //如果不在通话状态，则返回  
  
    if (call==null) {  
  
        return false ;  
  
    }else if (event.getAction () ==MotionEvent.ACTION_DOWN  
  
    //检测按钮按下、正在通话、是否为静音  
  
    & &call !=null& &call.isMuted () ) {  
  
        //静音切换  
  
        call.toggleMute () ;  
  
    }else if (event.getAction () ==MotionEvent.ACTION_UP  
  
    //检测按钮松开、是否为静音  
  
    & & !call.isMuted () ) {  
  
        //静音切换  
  
        call.toggleMute () ;  
    }  
}
```

```
}

return false ;

}

//加载菜单

public boolean onCreateOptionsMenu (Menu menu) {

//添加菜单

/*第一个int类型为group ID参数， 代表的是组概念， 可以将几个菜单项
归为一组， 以便更好地以组的方式管理菜单按钮；第二个int类型为item
ID参数， 代表的是项目编号， 这个参数非常重要， 一个item ID对应一
个menu中的选项，在后面使用菜单的时候， 依据该item ID来判断使
用的是哪个选项；第三个int类型为order ID参数， 代表的是菜单项的显
示顺序， 默认是0， 表示菜单的显示顺序就是按照add的显示顺序来显
示；第四个String类型为title参数， 表示选项中显示的文字*/

menu.add (0, CALL_ADDRESS, 0, "Call someone") ;

menu.add (0, SET_AUTH_INFO, 0, "Edit your SIP Info.") ;

menu.add (0, HANG_UP, 0, "End Current Call.") ;
```

```
    return true ;
```

```
}
```

```
//当Menu有命令被选择的时候，调用此方法
```

```
public boolean onOptionsItemSelected (MenuItem item) {
```

```
//根据选项的ID号判断
```

```
    switch (item.getItemId ()) {
```

```
//显示通话地址
```

```
    case CALL_ADDRESS :
```

```
        showDialog (CALL_ADDRESS) ;
```

```
        break ;
```

```
//设置信息
```

```
    case SET_AUTH_INFO :
```

```
        updatePreferences () ;
```

```
        break ;
```

```
//挂断

case HANG_UP :

//正在通话

if (call !=null) {

try{

//结束通话

call.endCall () ;

}catch (SipException se) {

Log.d ("WalkieTalkieActivity/onOptionsItemSelected",
"Error ending call.", se) ;

}

//释放资源

call.close () ;

}
```

```
        break ;  
  
    }  
  
    return true ;  
  
}  
  
//根据ID显示对话框  
  
@Override  
  
protected Dialog onCreateDialog (int id) {  
  
    switch (id) {  
  
        //显示通话地址  
  
        case CALL_ADDRESS :  
  
            //创建LayoutInflater类  
  
            LayoutInflater factory=LayoutInflater.from (this) ;  
  
            //加载布局  
  
            final View textBoxView=
```

```
factory.inflate (R.layout.call_address_dialog,null) ;  
  
//创建一个新的对话框  
  
return new AlertDialog.Builder (this)  
  
//设置标题  
  
.setTitle ("Call Someone.")  
  
//加载布局  
  
.setView (textBoxView)  
  
//设置确定按钮，并定义响应事件  
  
.setPositiveButton (android.R.string.ok,  
  
new DialogInterface.OnClickListener () {  
  
public void onClick (DialogInterface dialog,int whichButton) {  
  
//定义输入控件  
  
EditText textField= (EditText)  
  
(textBoxView.findViewById (R.id.calladdress_edit) ) ;
```

```
//获取地址  
  
sipAddress=textField.getText () .toString () ;  
  
//初始化呼叫  
  
initiateCall () ;  
  
}  
  
})  
  
//定义取消按钮，并设置监听事件  
  
.setNegativeButton (android.R.string.cancel,new  
DialogInterface.OnClickListener () {  
  
public void onClick (DialogInterface dialog,int whichButton) {  
  
}  
  
}) .create () ;//创建对话框  
  
//显示更新设置对话框  
  
case UPDATE_SETTINGS_DIALOG :  
  
//创建一个新的对话框
```

```
return new AlertDialog.Builder (this)

//设置显示的信息

.setMessage ("Please update your SIP Account Settings.")

//设置确定按钮，并定义响应事件

.setPositiveButton (android.R.string.ok,

new DialogInterface.OnClickListener () {

public void onClick (DialogInterface dialog,int whichButton) {

//更新设置

updatePreferences () ;

}

})

//定义取消按钮，并设置监听事件

.setNegativeButton (android.R.string.cancel,

new DialogInterface.OnClickListener () {
```

```
public void onClick (DialogInterface dialog,int whichButton) {}

}) .create () ;//创建对话框

}

return null ;

}

//更新用户设置

public void updatePreferences () {

Intent settingsActivity=new Intent (getBaseContext () ,

SipSettings.class) ;

//调用SipSettings

startActivity (settingsActivity) ;

}

}
```

在客户端登录之后，可以在服务器界面看到登录的详细信息，如图7-13所示。

The screenshot shows the 'SIP Server Admin' interface for the brekeke SIP Server. The left sidebar contains navigation links: Status, Active Sessions, Registered Clients (selected), Dial Plan, Aliases, User Authentication, Logs, Configuration, Domains, Redundancy, and Maintenance. Below the sidebar is a 'Logout' button. The main area is titled 'Registered Clients' with tabs for 'Registered Clients' (selected) and 'Manual Register'. It includes a 'Show Filter' section and a 'Unregister' button. A message at the top right says 'Registered: 2 Pages: 1'. A table lists two registered clients: 'aaa' and 'bbb'. The table columns are 'User', 'Contact URI', and 'Detail'. The details for 'aaa' show: Expires : 3600, Priority : 1000, User Agent : X-Lite release 5.0.0 stamp 67284, Requester : 192.168.1.101:5061, Time Update : Sat Dec 08 08:35:14 CST 2012. The details for 'bbb' show: Expires : 3600, Priority : 1000, User Agent : SIPAUA/0.1.001, Requester : 192.168.1.101:57250, Time Update : Sat Dec 08 08:51:10 CST 2012.

图 7-13 查看客户端登录服务器的详细信息

输入地址的界面如图7-14所示。

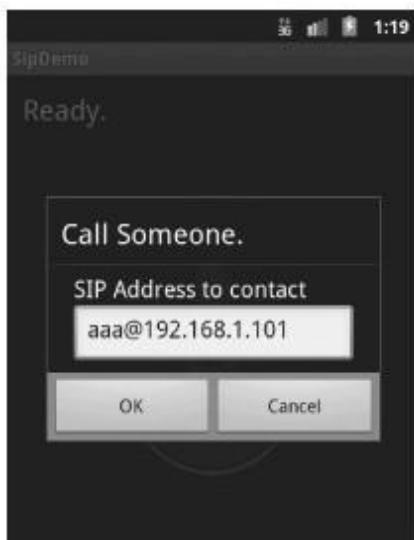


图 7-14 连接SIP服务器

通话中的界面如图7-15所示。

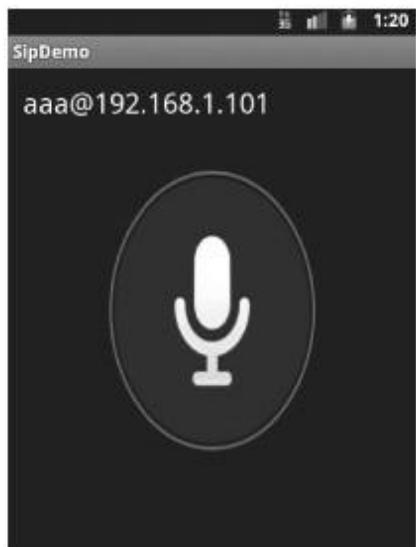


图 7-15 通话中界面

可以使用X-Lite做桌面的客户端进行通信测试，其通话界面如图7-16所示。



图 7-16 使用X-Lite做客户端进行通信测试

7.7 小结

Android SDK 2.3开始提供支持SIP的API，可以添加基于SIP的网络电话功能到应用程序。本章在简介SIP协议的基础上，详细介绍了使用Brekeke SIP Server搭建SIP服务器的方法。后面从设置应用程序的权限到初始化和监听SIP通话，一步步地介绍了Android中实现SIP通话的一般方法和步骤。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第8章 基于XMPP协议的即时通信 应用

本章介绍XMPP的基本知识、Openfire服务器的搭建，以及在Android开发中的使用。

8.1 XMPP协议简介

XMPP（Extensible Messaging and Presence Protocol，可扩展消息处理现场协议）是一种以XML为基础的开放式实时通信协议，是经由互联网工程工作小组（IETF）通过的互联网标准。它主要用于即时消息（IM），允许Internet用户向Internet上的其他任何人发送即时消息，允许二者的操作系统和浏览器不同。

XMPP的前身是Jabber，一个开源的网络即时通信协议。XMPP原本是为即时通信而量身定制，但由于XML Stanza本身是XML元素，XML的迅速发展使得XMPP也可以适用于其他方面。XMPP与IMPP、PRIM、SIP（SIMPLE）被称为四大主流IM协议，在这些协议中，XMPP是最灵活的。XMPP的扩展协议Jingle使得其支持语音和视频。

XMPP的特点如下：

- 开放。XMPP协议是自由、开放、易于理解的。在客户端、服务器、组件、源码库等方面，都已经各自有多种实现。
- 标准。互联网工程工作小组已经将Jabber的核心XML流协定以XMPP之名，正式列为认可的实时通信及Presence技术。而XMPP的技术规格已被定义在RFC 3920及RFC 3921中。任何IM供应商在遵循XMPP协议下，都可与Google Talk实现连接。

■稳定。第一个Jabber（现在XMPP）技术是Jeremie Miller在1998年开发的，现在已经相当稳定，数以百计的开发者为XMPP技术的完善而努力。当今的互联网上有数以万计的XMPP服务器运行着，并有数以百万计的人们使用XMPP实时通信软件。

■分布式。XMPP网络的架构与电子邮件十分相像，XMPP核心协议通信方式是先创建一个stream，以TCP传递XML stream。它没有中央主服务器，任何人都可以运行自己的XMPP服务器，使个人及组织能够掌控他们的实时通信体验。

■安全。任何XMPP协议的服务器可以独立于公众XMPP网络（例如在企业内部网络中），而使用SASL及TLS等技术的可靠安全性，已自带于核心XMPP技术规格中了。

■可扩展。XML命名空间的威力可使任何人在核心协定的基础上建造定制化的功能，为了维持通透性，常见的扩展有XMPP Standards Foundation。

■弹性佳。XMPP除了可用在实时通信的应用程序中外，还能用于网络管理、内容公告、协同工作、文件共享、游戏、远程系统监控等。

■多样性。用XMPP协定来建造及部署实时应用程序及服务的公司及开放源代码计划分布在各种领域。用XMPP技术开发软件，资源及支持的来源是多样的，使其不会陷于被“绑架”的困境。

XMPP网络是基于服务器的（即客户端之间彼此不直接交谈），但是也是分散式的，XMPP没有中央官方服务器，任何人都可以在自己的网域上运行XMPP服务器。

Jabber识别符（JID）是用户登录时所使用的账号，通常看起来像一个电子邮件地址，如someone@example.com。前半部分为用户名，后半部分为XMPP服务器域名，两个字段以@符号间隔。

假设user1（user1@example1.com）想与user2（user2@example2.com）通话，他们两人的账号分别在example1.com及example2.com的服务器上。当user1输入信息并按下传送按钮之后，其传输的步骤如下：

步骤1 user1的XMPP客户端将其信息传送到example1.com的XMPP服务器。

步骤2 example1.com的XMPP服务器打开与example2.com的XMPP服务器的连接，并实现传送。

步骤3 example2.com的XMPP服务器将信息传送给user2。如果其目前不在联机状态，那么存储信息以待稍后传送。

8.2 使用Openfire搭建XMPP服务器

Openfire是一个强大的即时消息和聊天服务器，它使用Java语言编写，实现了XMPP协议，并且是开源的。其官方网址为：

[http://www.igniterealtime.org/projects/openfire/。](http://www.igniterealtime.org/projects/openfire/)下面介绍使用Openfire搭建XMPP服务器的具体步骤。

步骤1 下载最新的Openfire安装文件。官方下载站点为
<http://www.igniterealtime.org/downloads/index.jsp#openfire>，本书下载的
文件名为openfire_3_7_1.exe。

步骤2 安装Openfire。下载完成后，双击安装文件进行安装。默认安装
目录为C：\Program Files\。具体如下。

1) 选择安装语言，如图8-1所示。



图 8-1 选择安装语言

2) 接受许可协议，如图8-2所示。



图 8-2 接受许可协议

步骤3 安装完毕后，弹出如图8-3所示的界面。单击Launch Admin按钮进入`http://127.0.0.1:9090/setup/index.jsp`页面。接下来就要配置Openfire服务器了。



图 8-3 配置服务器

步骤4 配置服务器第一步如图8-4所示，选择语言中文简体。

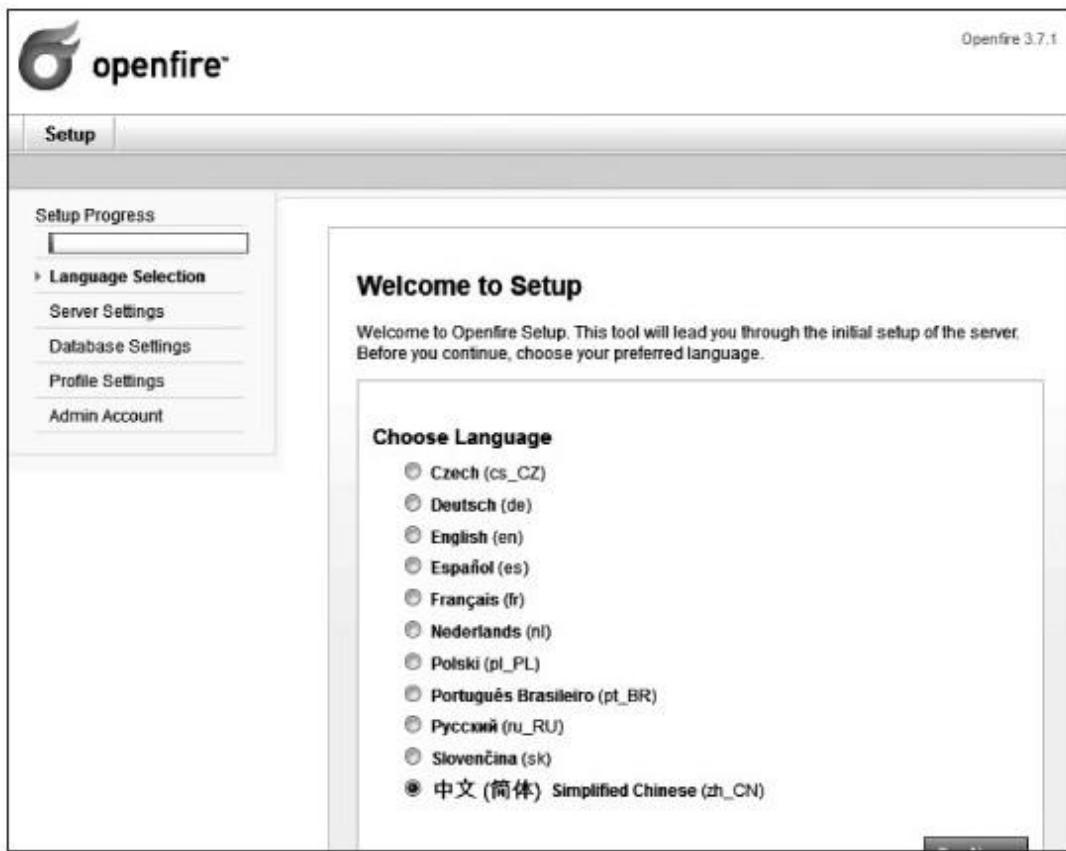


图 8-4 配置服务器-选择语言

步骤5 配置服务器域名，如图8-5所示。



图 8-5 配置域名

如果是本地访问，那么可以不修改或使用localhost、127.0.0.1的方式。如果用于外网或局域网访问，那么地址配置成外网或局域网地址。单击“继续”按钮后显示如图8-6所示界面。

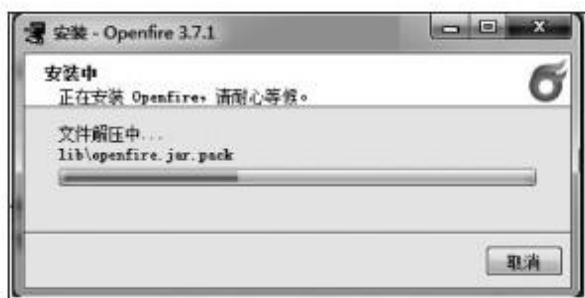


图 8-6 安装过程

步骤6 选择数据库，如图8-7所示。



图 8-7 选择数据库

这里选择Openfire自带的嵌入的数据库。当然也可以选择数据库类型，如Oracle、SQLServer、MySQL等。如果Openfire没有带JDBC的连接驱动，需要添加连接数据库的JDBC驱动，驱动放在C：\Program Files\openfire\lib目录下。

步骤7 选择特性设置，如图8-8所示。



图 8-8 特性设置界面

步骤8 设置管理员账户，如图8-9所示。此步可以跳过。



图 8-9 设置管理员账户

步骤9 安装完成后的效果如图8-10所示。接着单击“登录到管理控制台”按钮可以进入管理员控制台页面。



图 8-10 安装完成

步骤10 进入`http://127.0.0.1:9090/login.jsp`页面后，输入用户名admin、密码admin进行登录，如图8-11所示。



图 8-11 管理控制台登录

步骤11 进入运行界面，如图8-12所示。



图 8-12 运行界面

至此，Openfire的安装和配置已经完成。

8.3 登录XMPP服务器

8.3.1 Asmack相关类

Smack是一个Java语言编写的、开源的、易于使用的XMPP（Jabber）客户端类库。可以把它用于商业或非商业应用程序。

Asmack是Smack在Android上的移植版本，更加适合Android的开发环境。其下载地址为[http://code.google.com/p/asmack/。](http://code.google.com/p/asmack/)

Asmack的相关类简介如下。

- 1) ConnectionConfiguration。通过该类设置用于与XMPP服务建立连接的配置。它能配置连接是否使用TLS、SASL加密等。它包含ConnectionConfiguration.SecurityMode内嵌类。
- 2) XMPPConnection。这个类用来连接XMPP服务。可以使用connect()方法建立与服务器的连接，使用disconnect()方法断开与服务器的连接。

在创建连接前可以设置XMPPConnection.DEBUG_ENABLED为true，使开发过程中可以弹出一个GUI窗口，用于显示连接与发送Packet的信息。

- 3) ChatManager。用于监控当前所有chat。可以使用createChat (String userJID,Message Listener listener) 创建一个聊天。
- 4) Chat。Chat用于监控两个用户间的一系列消息 (message) 。使用addMessageListener (MessageListener listener) , 当有任何消息到达时将会触发listener的processMessage (Chat chat,Message message) 方法。
- 5) Message。Message用于表示一个消息包 (可以用调试工具看到发送包和接收包的具体内容) 。它有以下多种类型，如表8-1所列。

表 8-1 Message 消息包的类型与含义列表

消息包类型	含 义
Message.Type.NORMAL	(默认) 文本消息 (比如邮件)
Message.Type.CHAT	典型的短消息，如 QQ 聊天时一行一行地显示的消息
Message.Type.GROUP_CHAT	群聊消息
Message.Type.HEADLINE	滚动显示的消息
Message.TYPE.ERROR	错误的消息

此外，Message有两个内部类：一个是Message.Body，表示消息体；另一个是Message.Type，表示消息类型。

6) Roster。该类表示存储了一个花名册，其中包含很多RosterEntry。为了易于管理，花名册的项被分配到了各个group中。当建立与XMPP服务的连接后可以使用connection.getRoster () 获取Roster对象。

别的用户可以使用一个订阅请求（相当于QQ加好友）尝试订阅目的用户。可以使用枚举类型Roster.SubscriptionMode的值处理这些请求：

accept_all表示接收所有订阅请求；reject_all表示拒绝所有订阅请求；manual表示手工处理订阅请求。

创建组方法如下：

```
RosterGroup group=roster.createGroup ("大学") ;
```

可以向组中添加RosterEntry对象，方法如下：

```
group.addEntry (entry) ;
```

7) RosterEntry。RosterEntry表示Roster（花名册）中的每条记录。它包含了用户的JID、用户名、用户分配的昵称。

8) RosterGroup。RosterGroup表示RosterEntry的组。可以使用addEntry (RosterEntry entry) 添加，contains (String user) 判断某用户是否在组中，removeEntry (RosterEntry entry) 是从组中移除，getEntries () 获取所有RosterEntry。

9) Presence。Presence表示XMPP状态的packet。每个presence packet都有一个状态，用枚举类型Presence.Type的值表示，可能的取值及含义如表8-2所示。

表 8-2 Presence.Type 的可能取值及含义

可能取值	含 义
available	(默认) 用户空闲状态
unavailable	用户没空看消息
subscribe	请求订阅别人，即请求加对方为好友
subscribed	统一被别人订阅，也就是确认被对方加为好友
unsubscribe	取消订阅别人，请求删除某好友
unsubscribed	拒绝被别人订阅，即拒绝对方的添加请求
error	当前状态 packet 有错误

8.3.2 登录XMPP服务器

首先我们创建一个新用户，然后建立与XMPP服务器的连接，最后使用新用户登录到XMPP服务器。

创建新用户需要引用的类如下：

//管理XMPP账号的包

```
import org.jivesoftware.smack.AccountManager ;
```

//网络连接管理

```
import org.jivesoftware.smack.Connection ;
```

//网络连接设置

```
import org.jivesoftware.smack.ConnectionConfiguration ;
```

//XMPP的连接管理

```
import org.jivesoftware.smack.XMPPConnection ;
```

```
//XMPP异常
```

```
import org.jivesoftware.smack.XMPPEception ;
```

下面是创建新用户的核心代码：

```
//指定连接到服务器的参数：网站和端口
```

```
ConnectionConfiguration mConnectionConfiguration=new
```

```
ConnectionConfiguration ("192.168.1.101", 5222) ;
```

```
//初始化XMPPConnection连接
```

```
XMPPConnection mXMPPConnection=new XMPPConnection
```

```
(mConnectionConfiguration) ;
```

```
//连接上XMPP服务器
```

```
try{
```

```
mXMPPConnection.connect () ;
```

```
//实例化用户管理
```

```
AccountManager
```

```
mAccountManager=mXMPPConnection.getAccountManager () ;
```

```
//用指定的用户名和密码创建用户
```

```
mAccountManager.createAccount ("NewUser", "111111") ;
```

```
}catch (XMPPException e) {
```

```
//打印异常
```

```
e.printStackTrace () ; }
```

```
}
```

新用户创建前的用户列表截图如图8-13所示。

用户摘要		
用户总数: 3 – 按用户名排序 – 用户 / 页		
	在线	用户名
1	是	admin ★
2	是	user1
3	是	user2

图 8-13 用户列表截图（新建用户前）

新用户创建后的用户列表截图如图8-14所示。

用户摘要	
用户总数: 4 – 按用户名排序 – 用户 / 页	
在线 用户名	
1	admin ★
2	newuser
3	user1
4	user2

图 8-14 用户列表截图（新建用户后）

登录之前要和XMPP服务器建立连接。XMPPConnection类用来建立到XMPP服务器的连接。要建立SSL连接，使用SSLXMPPConnection类。几种建立连接的方法如下：

//建立一个到服务器的连接

```
XMPPConnection conn1=new XMPPConnection ("hostname") ;
```

//通过一个特殊的端口建立一个到服务器的连接

```
XMPPConnection conn2=new XMPPConnection ("hostname", 5222) ;
```

//建立一个到服务器的SSL连接

```
XMPPConnection connection=new SSLXMPPConnection  
("hostname") ;
```

登录代码如下：

```
//创建连接参数  
  
ConnectionConfiguration mConnectionConfiguration=  
  
new ConnectionConfiguration ("192.168.1.101", 5222) ;  
  
XMPPConnection mXMPPConnection=new XMPPConnection  
(mConnectionConfiguration) ;  
  
try{  
  
    mXMPPConnection.connect () ;  
  
    //使用指定的用户名和密码登录  
  
    mXMPPConnection.login ("NewUser", "111111") ;  
  
}catch (XMPPException e) {  
  
    e.printStackTrace () ;  
  
}
```

而下面的代码使得该用户下线：

```
mXMPPConnection.disconnect () ;
```

如果考虑登录进去，但是显示不在线，可以用以下方法：

//Presence类可以设置用户的是否在线的类型属性

```
Presence mPresence=new Presence (Presence.Type.unavailable) ;
```

//向服务器提交该属性

```
mXMPPConnection.sendPacket (mPresence) ;
```

用户登录之后的截图如图8-15所示。



图 8-15 用户列表截图（用户登录后）

8.4 联系人相关操作

8.4.1 获取联系人列表

Asmack中的Roster对象能跟踪其他用户的有效性（存在）。联系人列表需要列表引用Roster。一个Roster实例通过XMPPConnection.getRoster()方法获得，需要注意的是只有在成功登录服务器之后才可以用该方法。

在Roster中每个用户用一个RosterEntry表示，它包括了用户的XMPP地址、用户的昵称等信息。下面是获取联系人列表的代码：

//新建Roster对象

```
Roster roster=mXMPPConnection.getRoster();
```

//获取联系人列表的集合

```
Collection<RosterEntry>rg=roster.getEntries();
```

//遍历联系人列表

```
for (RosterEntry rosterEntry:rg) {
```

//获取联系人列表

```
Log.d (TAG, "====rosterEntry.getName ()  
====="+rosterEntry.getUser () ) ;  
}
```

运行结果显示如下：

```
D/XMPPActivity (676) :====rosterEntry.getName () ;  
=====user2@stonechen
```

```
D/XMPPActivity (676) :====rosterEntry.getName () ;  
=====user1@stonechen
```

8.4.2 获取联系人状态

可以通过Presence的getStatus () 来获取好友的状态信息。

//新建Roster对象

```
Roster roster=mXMPPConnection.getRoster () ;
```

//获取联系人列表的集合

```
Collection<RosterEntry>rg=roster.getEntries () ;
```

//遍历联系人列表

```
for (RosterEntry rosterEntry:rg) {  
  
    //获取好友状态对象  
  
    Presence presence=roster.getPresence (rosterEntry.getUser () ) ;  
  
    //获取好友状态  
  
    Log.d (TAG, "====presence.getStatus () ====="+presence.getStatus  
    () ) ;  
  
}
```

而设置状态的方法如下：

```
//新建状态为在线  
  
Presence mPresence=new Presence (Presence.Type.available) ;  
  
//设置状态信息  
  
mPresence.setStatus ("Set My Status here") ;  
  
//发送配置包，设置在线状态  
  
mXMPPConnection.sendPacket (mPresence) ;
```

8.4.3 添加和删除联系人

下面是发出添加联系人请求的代码：

```
//新建Roster对象  
  
Roster roster=mXMPPConnection.getRoster () ;  
  
try{  
  
//添加联系人， 3个参数分别为：用户名（JID） 、 昵称和分组  
  
roster.createEntry ("user1@127.0.0.1", null,new String[]{"friends"}) ;  
  
}catch (XMPPException e) {  
  
//抛出异常  
  
e.printStackTrace () ;  
  
}  
  
}
```

添加成功之后， Openfire后台管理处可以看到其好友状态，如图8-16所示。

The screenshot shows a web-based user roster interface titled "User Roster". A message at the top states: "Below is the list of roster items for user newuser. Shared group may not be deleted via this interface." Below this, it says "Total Items: 2 – Sorted by JID". There are dropdown menus for "Items per page: 15" and "Show: All roster items". A table lists two entries:

JID
1 user1@stonechen
2 user2@stonechen

图 8-16 添加好友成功后的后台显示

下面是删除好友的代码：

```
//新建Roster对象  
Roster roster=mXMPPConnection.getRoster () ;  
  
try{  
    //删除联系人  
  
    roster.removeEntry (roster.getEntry ("user1@stonechen") ) ;  
  
}catch (XMPPException e) {  
  
    e.printStackTrace () ;  
  
}
```

删除好友之后的后台显示，如图8-17所示。

The screenshot shows a web-based user roster interface titled "User Roster". It displays a single contact item for "user2@stonechen". The interface includes a message about shared groups, sorting by JID, and pagination settings (15 items per page). The contact list table has columns for JID and name.

JID	Name
1	user2@stonechen

图 8-17 删除好友后的后台显示

8.4.4 监听联系人添加信息

收到订阅请求时候的XMPP信息如下：

```
<presence
id="rm1952_24"to="newuser@stonechen"from="user1@stonechen"type="subscribed"></presence>
```

收到被删除订阅的时候的XMPP信息如下：

```
<presence
to="newuser@stonechen"from="user1@stonechen"type="unsubscribe">
</presence>
```

下面是添加监听的核心代码：

```
//添加监听联系人信息

private static void addListener () {

    //新建消息过滤器

    PacketFilter filterMessage=new PacketTypeFilter (Presence.class) ;

    //创建监听器

    PacketListener myListener=new PacketListener () {

        public void processPacket (Packet packet) {

            //消息包赋值给一个新的Presence

            final Presence mPresence= (Presence) packet ;

            if (mPresence.getType () .equals (Presence.Type.subscribe) )

            {

                //收到请求订阅的信息，可以在该处处理

            }else if (mPresence.getType () .equals (Presence.Type.unsubscribe) ) {

                //接收到取消订阅的信息，可以在该处处理

            }

        }

    }

}
```

}

//packet.toXML () 可以将接收到的信息转化为XML字符串格式

}

} ;

//在XMPP连接上注册该监听

mXMPPConnection.addPacketListener (myListener,filterMessage) ;

}

8.5 消息处理

发送接收消息处于即时通信的核心地位，类org.jivesoftware.smack.Chat 用于在两个联系人之间发送和接收消息。下面一起看看如何发送和接收消息。

8.5.1 接收消息

下面是接收消息监听的代码：

//接收消息

```
private static void addMessageListener () {
```

//添加消息监听

```
mXMPPConnection.getChatManager () .addChatListener (new  
ChatManagerListener () {
```

@Override

//创建消息

```
public void chatCreated (Chat mChat,boolean createdLocally) {
```

```
//检测消息来源，是否为本地发出的

//这里为接收到的消息

if ( ! createdLocally) {

//监听接收到的消息

mChat.addMessageListener (new MessageListener () {

//添加消息处理

@Override

public void processMessage (Chat mChat,Message mMessage) {

//打印接收到的消息

Log.d (TAG, "接收到的消息为："+mMessage.getBody () ) ;

}

}

}

}

}
```

```
} ) ;
```

```
}
```

8.5.2 发送消息

一个chat在两个用户之间创建一个消息线程（通过线程ID）。下面这段代码演示了怎样和需要对话的用户创建一个新的chat并向他发送一条文本消息。

```
//新建一个消息线程
```

```
Chat chat=mXMPPConnection.getChatManager () .
```

```
//指定接收人，并添加消息监听
```

```
createChat ("user1@stonechen", new MessageListener () {
```

```
//处理消息
```

```
public void processMessage (Chat chat,Message message) {
```

```
//打印接收到的消息
```

```
System.out.println ("Received message :" +message) ;
```

```
}
```

```
} ) ;  
  
//创建新的消息  
  
try{  
  
    Message newMessage=new Message () ;  
  
    //设置消息内容  
  
    newMessage.setBody ("Hello ! ") ;  
  
    //发送消息  
  
    chat.sendMessage (newMessage) ;  
  
} catch (XMPPException e) {  
  
    //打印错误  
  
    e.printStackTrace () ;  
  
}
```

8.6 实战案例：XMPP多人聊天

传统上，即时消息被视为一对一的聊天。在XMPP的xep0045规范中给出了多人聊天（MUC,Multi-User Chat）的方法。查看MUC协议文本的网址为<http://xmpp.org/extensions/xep-0045.html>。

XMPP定义的多人聊天比一些常见多人聊天系统（比如群、组）更有可定制性，也更加复杂。多个XMPP用户可以在一个房间或频道互相交流信息，类似IRC；还有标准聊天室功能，如聊天室的主题和邀请；MUC协议还定义了一个强有力的房间控制模型，包括能够“踢出”和禁止用户、任命主持人和管理员，要求会员或密码才能加入房间等功能。

下面来看看MUC协议中最重要的功能在Android中是如何实现的。

8.6.1 创建新多人聊天室

在XMPP协议中，用户可以通过两种方式创建新的多人聊天室。一种是Instant rooms，适用于立即进入和基于一些默认配置自动创建。另一种是Reserved rooms，允许其他人进入之前由聊天室创建者手动配置。

创建聊天室的步骤如下：

步骤1 创建MultiUserChat的一个实例。聊天室名字通过构造方法传递给要创建的聊天室。

步骤2 调用MultiUserChat的create (String nickname) 方法，在这里nickname是用户加入聊天室用的昵称。

步骤3 创建一个Instant room，调用sendConfigurationForm (Form form) 方法，在这里表单是一个空表单；创建一个Reserved room应该首先获得聊天室的配置表单，完成表单后将它发送回服务器。

下面是创建一个instant room的核心代码：

```
//使用XMPPConnection创建一个MultiUserChat  
  
try{  
  
    //初始化MultiUserChat对象  
  
    MultiUserChat muc=new MultiUserChat (mXMPPConnection,  
  
        "newroom1@conference.stonechen") ;  
  
    //创建聊天室  
  
    muc.create ("newroom1") ;  
  
    //发送一个空表单配置，这显示我们想要一个instant room
```

```
muc.sendConfigurationForm (new Form (Form.TYPE_SUBMIT) ) ;  
}  
catch (XMPPException e) {  
    e.printStackTrace () ;  
}
```

下面的代码功能是创建一个reserved room，其表单使用默认值完成。

```
try{  
    //使用XMPPConnection创建一个MultiUserChat  
    MultiUserChat muc=new MultiUserChat (mXMPPConnection,  
    "newroom2@conference.stonechen") ;  
  
    //创建聊天室  
    muc.create ("newroom2") ;  
  
    //获得聊天室的配置表单  
    Form form=muc.getConfigurationForm () ;  
  
    //根据原始表单创建一个要提交的新表单
```

```
Form submitForm=form.createAnswerForm () ;  
  
//向要提交的表单添加默认答复  
  
for (Iterator fields=form.getFields () ; fields.hasNext () ;) {  
  
    FormField field= (FormField) fields.next () ;  
  
    if ( ! FormField.TYPE_HIDDEN.equals (field.getType ()) )  
        & & field.getVariable () !=null) {  
  
        //设置默认值作为答复  
  
        submitForm.setDefaultAnswer (field.getVariable () ) ;  
  
    }  
  
}  
  
//设置聊天室的新拥有者  
  
List owners=new ArrayList () ;  
  
owners.add ("NewUser@stonechen") ;  
  
submitForm.setAnswer ("muc#roomconfig_roomowners", owners) ;
```

```
//发送已完成的表单（有默认值）到服务器来配置聊天室
```

```
muc.sendConfigurationForm (submitForm) ;  
}  
catch (XMPPException e) {  
    e.printStackTrace () ;  
}
```

在Openfire中也可以创建聊天室，如图8-18所示。



图 8-18 Openfire中创建聊天室

在图8-18所示的界面中填写“房间标识”的内容，这里为newroom1；其他的内容可以视需要填写。全部设定好以后单击“保存更改”按钮，至

此聊天室就创建完成了，如图8-19所示。我们可以看到服务器管理界面的变化。



图 8-19 Openfire中聊天室创建后

在调用不同版本的Asmack的时候，有些版本会在运行上面示例的时候报错，这是由于Asmack的一个Bug引起的。解决方法如下：

//添加属性

```
private void configure (ProviderManager pm) {
```

.....

//聊天室邀请

```
pm.addExtensionProvider ("x", "jabber:x : conference",
```

```
new GroupChatInvitation.Provider () ) ;  
  
//数据格式  
  
pm.addExtensionProvider ("x", "jabber:x : data", new  
DataFormProvider () ) ;  
  
//聊天室用户  
  
pm.addExtensionProvider ("x", "http://jabber.org/protocol/muc#user",  
new MUCUserProvider () ) ;  
  
//聊天室管理员  
  
pm.addIQProvider ("query", "http://jabber.org/protocol/muc#admin",  
new MUCAdminProvider () ) ;  
  
//聊天室拥有者  
  
pm.addIQProvider ("query", "http://jabber.org/protocol/muc#owner",  
new MUCOwnerProvider () ) ;  
  
.....  
}
```

在XMPPConnection实例化之后调用以下语句，添加一些属性。

```
configure (ProviderManager.getInstance () ) ;
```

注意 更多有关这个问题的讨论可以参见官方网站

<http://community.igniterealtime.org/thread/31118>上的介绍。

8.6.2 加入聊天室

加入一个聊天室的步骤如下：

步骤1 创建一个MultiUserChat的实例。把要加入的聊天室的名称传递给构造方法。

步骤2 调用MultiUserChat实例的join (String nickname) 方法。若想要加入聊天室需要密码才能加入，可以使用join (String nickname, String password) 方法。

下面的例子中使用特定昵称加入一个聊天室。

```
//使用XMPPConnection创建一个MultiUserChat
```

```
MultiUserChat muc2=new MultiUserChat
```

```
(conn1, "newroom@conference.stonechen") ;
```

```
//用户2加入新聊天室，参数为用户在聊天室中昵称
```

```
muc2.join ("user2") ;
```

在下面的例子中使用特定昵称和密码加入聊天室。

```
//使用XMPPConnection创建一个MultiUserChat
```

```
MultiUserChat muc2=new MultiUserChat
```

```
(conn1, "myroom@conference.stonechen") ;
```

```
//用户2使用密码加入新聊天室，参数为用户在聊天室中昵称和登录密  
码
```

```
muc2.join ("user2", "password") ;
```

8.6.3 发送和接收消息

发送MUC消息的代码如下：

```
try{
```

```
//使用XMPPConnection创建一个MultiUserChat
```

```
MultiUserChat muc1=new MultiUserChat (mXMPPConnection,
```

```
"newroom1@conference.stonechen") ;
```

```
//用户加入新聊天室，参数为用户在聊天室中的昵称
```

```
muc1.join ("NewMUCuser") ;  
  
Message newMessage=muc1.createMessage () ;  
  
//设置聊天内容  
  
newMessage.setBody ("Hello MUC ! ") ;  
  
//发送聊天信息  
  
muc1.sendMessage (newMessage) ;  
  
}catch (XMPPException e) {  
  
e.printStackTrace () ;  
  
}  

```

接收MUC消息的代码如下：

```
//使用XMPPConnection创建一个MultiUserChat  
  
MultiUserChat muc1=new MultiUserChat  
(mXMPPConnection, "newroom1@conference.stonechen") ;  
  
//用户加入新聊天室，参数为用户在聊天室中的昵称  
  
try{
```

```
muc1.join ("NewMUCuser") ;  
  
}catch (XMPPException e) {  
  
    e.printStackTrace () ;  
  
}  
  
//添加监听  
  
muc1.addMessageListener (new PacketListener () {  
  
    @Override  
  
    public void processPacket (Packet packet) {  
  
        //此处对收到的消息进行处理  
  
        System.out.println (packet.toXML () ) ;  
  
    }  
  
}) ;
```

8.7 小结

本章主要介绍了如何使用Openfire搭建XMPP服务器，客户端如何使用Asmack登录服务器。其中XMPP是一种以XML为基础的开放式实时通信协议，是互联网工程工作小组（IETF）通过的互联网标准。Openfire是一个Java语言编写的、开源的、实现了XMPP协议的即时消息和聊天服务器。Smack是一个Java语言编写的、开源的、易于使用的XMPP（jabber）客户端类库，Asmack是Smack在Android上的移植版本。

在介绍了使用Asmack类库登录Openfire搭建的XMPP服务器后，又介绍了获取、添加、删除联系人，接收、发送消息，以及创建多人聊天室等相关操作的具体步骤。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第9章 Android对HTML的处理

本章从调用接口的层次讲解Android对HTML的处理，以使读者更加深入地了解如何使用现有的API进行Web显示和处理。

9.1 Android HTML处理关键类

Android HTML处理关键类在源代码中的
frameworks/base/core/java/android/webkit目录下，各类的简单说明如
下。

- AccessibilityInjector.java：为WebView注入可以访问的JavaScript和其相
关内容。
- AutoCompletePopup.java：根据输入内容，自动给出补全提示。
- BrowserFrame.java：对WebCore中Frame对象的Java层封装，用于创建
WebCore中定义的Frame，以及为该Frame对象提供Java层回调方法。
- CacheManager.java:Cache管理对象，负责Java层Cache对象管理。
- CallbackProxy.java：用于处理WebCore与UI线程消息的代理类。
- CertTool.java:WebView证书工具。
- ClientCertRequestHandler.java：处理客户端证书请求。
- ConsoleMessage.java:WebCore的JavaScript控制台消息。
- CookieManager.java:Cookies管理对象。

- CookieManagerClassic.java:CookieManager的扩展实现类。
- CookieSyncManager.java:Cookies同步管理对象，该对象负责同步RAM和Flash之间的Cookies数据。
- DateSorter.java：对日期进行排序。
- DebugFlags.java：给出调试标志。
- DeviceMotionAndOrientationManager.java：用于实现DeviceMotion和DeviceOrientation。
- DeviceMotionService.java：实现SensorEventListener接口，处理动作。
- DeviceOrientationService.java：实现SensorEventListener接口，处理方向变化。
- DownloadListener.java：下载侦听器接口。
- FindActionModeCallback.java：搜索动作回调。
- GeolocationPermissions.java：负责WebView的地理位置JavaScript API的权限管理。
- GeolocationPermissionsClassic.java:GeolocationPermissions扩展类，处理从UI线程中调用的WebKit线程。

- GeolocationService.java：实现Java层次的GeolocationServiceAndroid，封装了位置监听。
- HTML5Audio.java:HTML5 Audio支持类。
- HTML5VideoFullScreen.java：浏览器全屏视频视图。
- HTML5VideoInline.java：浏览器内嵌视频视图。
- HTML5VideoView.java：浏览器视频视图。
- HTML5VideoViewProxy.java：视频视图代理类。
- HttpAuthHandler.java：处理HTTP认证请求。
- JWebCoreJavaBridge.java：用于Java与WebCore库中Timer和Cookies对象交互的桥接代码。
- JniUtil.java：供JNI使用的实用类，用于获取cache目录等C代码无法直接获取的信息，以及读取资源包中的文件等。
- JsPromptResult.java:Js结果提示对象，用于向用户提示JavaScript运行结果。
- JsResult.java:Js结果对象，用于用户交互。

- KeyStoreHandler.java : 负责将证书安装到系统密钥存储区，从网络读取证书传给CertTool。
- L10nUtils.java : 负责字符串国际化。
- MimeTypeMap.java : MIME类型映射。
- MockGeolocation.java : 模拟地理位置信息。
- MustOverrideException.java : 扩展运行时错误。
- OverScrollGlow.java : 用于实现OverScroller效果。
- Plugin.java : 定义了插件的类。
- PluginData.java : 插件数据。
- PluginFullScreenHolder.java : 获取插件视图的容器的大小，并负责显示等操作。
- PluginList.java : 维护插件列表。
- PluginManager.java : 插件管理类。
- PluginStub.java : WebView的实现插件的接口。
- QuadF.java : 定义了一个四边形。

- `SearchBox.java`：定义搜索对话框接口。
- `SearchBoxImpl.java`：搜索对话框接口实现。
- `SelectActionModeCallback.java`：选择动作回调。
- `SslCertLookupTable.java`：存储用户是否使用一个证书的决定。
- `SslClientCertLookupTable.java`：保存客户端证书的用户选择。
- `SslErrorHandler.java`：处理SSL报错。
- `URLUtil.java`: URL实用处理类。
- `ValueCallback.java`：用于异步返回数据值的回调接口。
- `ViewManager.java`：子视图管理类，主要用于管理插件视图。
- `ViewStateSerializer.java`: WebView视图序列化和反序列化。
- `WebBackForwardList.java`: WebView对象中显示的历史数据列表。
- `WebBackForwardListClient.java`：浏览历史处理的客户接口类，所有需要接收浏览历史改变的类都需要实现该接口。
- `WebChromeClient.java`: Chrome客户基类， Chrome客户对象在浏览器文档标题、进度条、图标改变时候会得到通知。

- `WebCoreThreadWatchdog.java`: WebCore的“看门狗”。
- `WebHistoryItem.java` : 该对象用于保存一条网页历史数据。
- `WebIconDatabase.java` : 图标数据库管理对象，所有的`WebView`均请求相同的图标数据库对象。
- `WebIconDatabaseClassic.java`: `WebIconDatabase`的扩展类。
- `WebResourceResponse.java` : 封装资源的响应信息。
- `WebSettings.java`: `WebView`的管理设置数据，该对象数据是通过JNI接口从底层获取。
- `WebStorage.java` : 处理`WebStorage`数据库。
- `WebSyncManager.java` : 数据同步对象，用于RAM数据和FLASH数据的同步操作。
- `WebTextView.java` : 在HTML文本输入控件激活时，显示系统原生编辑组件。
- `WebView.java` Web : 视图对象，用于基本的网页数据载入、显示等UI操作。

- WebViewClient.java: Web视图客户对象，在Web视图中有事件产生时，该对象可以获得通知。
- WebViewCore.java：该对象对WebCore库进行了封装，将UI线程中的数据请求发送给WebCore处理，并且通过CallbackProxy的方式，通过消息通知UI线程数据处理的结果。
- WebViewDatabase.java：该对象使用SQLiteDatabase为WebCore模块提供数据存取操作。
- WebViewFactory.java：实现WebView嵌入Fragment中。
- WebViewFragment.java：实现WebView嵌入Fragment中。
- ZoomControlBase.java：缩放控件接口。
- ZoomControlEmbedded.java：内置缩放控件。
- ZoomManager.java：维护WebView的缩放状态。

9.2 HTMLViewer分析

为了使读者更清楚地了解Android API的调用过程，我们着重于Java层的分析，从加载以及调用入口开始，详细介绍重要API的源代码实现和功能，对于一些有启发意义的代码也将进行重点的分析。

在正式开始分析之前，首先需要下载Android源代码。将Android源代码下载好之后，为了阅读的方便，可以使用代码分析工具查看Android源代码。Linux下常用的代码分析工具有cflow、ctags、cscope和KScope等，根据个人习惯可以选择不同的分析工具，这里推荐KScope，其下载网址为：[http://kscope.sourceforge.net/。](http://kscope.sourceforge.net/)

下面我们通过分析源代码里面的一个HTMLViewer案例，来介绍WebView对HTML文档的简单处理。源代码位于目录src/com/android/htmlviewer/下，下面有HTMLViewerActivity和FileContentProvider两个文件。源代码具体如下：

```
//WebView中不支持加载文件，这个类封装了一个ContentProvider  
//HTMLViewer没有联网，也不允许JavaScript运行，基于的HTML内容  
文件加载是安全的  
  
public class FileContentProvider extends ContentProvider{
```

//定义字符串，保存响应行为字符串前面的通用部分

public static final String

BASE_URI="content://com.android.htmlfileprovider" ;

//通过URI获得MIME类型，如果没有则返回null

public String getType (Uri uri) {

//解析uri对象，获取其MIME类型字符串

String mimetype=uri.getQuery () ;

//检测其MIME类型字符串是否为null

return mimetype==null?"":mimetype ;

}

//串行化文件

public ParcelFileDescriptor openFile (Uri uri,String mode)

throws FileNotFoundException{

//安全检测

//检测当前应用的Uid和绑定的Uid是否相同

```
if (Process.myUid () != Binder.getCallingUid () ) {  
  
    //如果不同则抛出安全错误  
  
    throw new SecurityException ("Permission denied") ;  
  
}  
  
//检测文件读写模式  
  
if ( ! "r".equals (mode) ) {  
  
    //如果不为r模式，则抛出异常  
  
    throw new FileNotFoundException ("Bad mode for"+uri+" : "+mode) ;  
  
}  
  
//获取文件路径  
  
String filename=uri.getPath () ;  
  
//返回串行化描述的文件  
  
return ParcelFileDescriptor.open (new File (filename) ,  
  
    //设置为只读模式
```

```
ParcelFileDescriptor.MODE_READ_ONLY) ;  
}  
  
//删除  
  
public int delete (Uri uri,String selection,String[]selectionArgs) {  
  
    //抛出不支持操作的异常  
  
    throw new UnsupportedOperationException () ;  
}  
  
//插入  
  
public Uri insert (Uri uri,ContentValues values) {  
  
    throw new UnsupportedOperationException () ;  
}  
  
@Override  
  
public boolean onCreate () {  
  
    return true ;
```

```
}
```

```
//查询
```

```
public Cursor query (Uri uri,String[]projection,String selection,
```

```
String[]selectionArgs,String sortOrder) {
```

```
throw new UnsupportedOperationException () ;
```

```
}
```

```
//更新
```

```
public int update (Uri uri,ContentValues values,String selection,
```

```
String[]selectionArgs) {
```

```
throw new UnsupportedOperationException () ;
```

```
}
```

```
}
```

在Activity内部加入一个WebView，当启动的时候使用URI来调用
WebView。源代码位置为

src/com/android/htmlviewer/HTMLViewerActivity.java。具体代码如下：

/*HTMLViewerActivity内部包含了一个WebView小部件。当启动的时候，HTMLViewerActivity使用意图中的URI来获取URL，加载到WebView中。HTMLViewerActivity支持所有URL方案和标准的WebView*/

```
public class HTMLViewerActivity extends Activity{
```

```
//声明WebView对象
```

```
private WebView mWebView ;
```

```
//设置文件的最大长度
```

```
static final int MAXFILESIZE=8096 ;
```

```
//定义LOGTAG字符串，方便调试
```

```
static final String LOGTAG="HTMLViewerActivity" ;
```

```
@Override
```

```
protected void onCreate (Bundle savedInstanceState) {
```

```
super.onCreate (savedInstanceState) ;
```

```
//WebView在BrowserFrame中调用CreateInstance ()
```

```
//为了在onResume () 方法中调用getInstance ()  
  
//这里需要先调用CreateInstance ()  
  
CookieSyncManager.createInstance (this) ;  
  
//requestWindowFeature () 的取值  
  
//1.DEFAULT_FEATURES : 系统默认状态，一般不需要指定  
  
//2.FEATURE_CONTEXT_MENU : 启用ContextMenu， 默认该项已启用， 一般无需指定  
  
//3.FEATURE_CUSTOM_TITLE : 自定义标题。当需要自定义标题时必须指定  
  
//4.FEATURE_INDETERMINATE_PROGRESS : 不确定的进度  
  
//5.FEATURE_LEFT_ICON : 标题栏左侧的图标  
  
//6.FEATURE_NO_TITLE : 无标题  
  
//7.FEATURE_OPTIONS_PANEL : 启用“选项面板”功能， 默认已启用  
  
//8.FEATURE_PROGRESS : 进度指示器功能  
  
//9.FEATURE_RIGHT_ICON : 标题栏右侧的图标
```

```
requestWindowFeature (Window.FEATURE_PROGRESS) ;  
  
//初始化WebView对象  
  
mWebView=new WebView (this) ;  
  
//加入到界面中  
  
setContentView (mWebView) ;  
  
//设置标题和进度条  
  
mWebView.setWebChromeClient (new WebChrome () ) ;  
  
//设置WebView属性  
  
WebSettings s=mWebView.getSettings () ;  
  
//设定了WebView的HTML布局方式，其中包含了下面的3个参数值  
  
//NORMAL：正常显示，没有渲染变化  
  
//SINGLE_COLUMN：把所有内容放到WebView组件等宽的一列中  
  
//NARROW_COLUMNS：可能的话，使所有列的宽度不超过屏幕宽度  
  
s.setLayoutAlgorithm  
(WebSettings.LayoutAlgorithm.NARROW_COLUMNS) ;
```

```
//图片自动适应webView的大小  
s.setUseWideViewPort (true) ;  
  
//设置WebView是否保存密码  
s.setSavePassword (false) ;  
  
//设置是否保存表格数据  
s.setSaveFormData (false) ;  
  
//设置WebView是否从网上下载资源  
s.setBlockNetworkLoads (true) ;  
  
//设置JavaScript不可用  
s.setJavaScriptEnabled (false) ;  
  
//恢复WebView状态  
if (savedInstanceState !=null) {  
    mWebView.restoreState (savedInstanceState) ;  
}  
else{
```

```
//获取Intent对象

Intent intent=getIntent ()  ;

//检查Intent是否为null

if (intent.getData ()  !=null) {

//获取数据

Uri uri=intent.getData ()  ;

//检查并获取该Uri的方案名称

String contentUri="file".equals (uri.getScheme ()  )

?FileContentProvider.BASE_URI+

uri.getEncodedPath ()

: uri.toString ()  ;

//获取MIME类型

String intentType=intent.getType ()  ;

//检查MIME类型字符串是否为null
```

```
if (intentType != null) {  
  
    contentUri+="?" + intentType ;  
  
}  
  
//WebView加载对应的Uri  
  
mWebView.loadUrl (contentUri) ;  
  
}  
  
}  
  
}  
  
}  
  
//在onResume的生命周期里面，打开Cookie同步  
  
@Override  
  
protected void onResume () {  
  
    super.onResume () ;  
  
    CookieSyncManager.getInstance () .startSync () ;  
  
}
```

```
//保存程序状态

@Override

protected void onSaveInstanceState (Bundle outState) {

    mWebView.saveState (outState) ;

}

//在onStop生命周期里面，Cookie停止同步

@Override

protected void onStop () {

    super.onStop () ;

    CookieSyncManager.getInstance () .stopSync () ;

}

//WebView停止加载

mWebView.stopLoading () ;

}

//在onDestroy生命周期里面，销毁WebView
```

```
@Override

protected void onDestroy () {

super.onDestroy () ;

//销毁WebView

mWebView.destroy () ;

}

//重写WebChromeClient

class WebChrome extends WebChromeClient{

//当文档标题改变的时候触发

@Override

public void onReceivedTitle (WebView view,String title) {

//设置标题

HTMLViewerActivity.this.setTitle (title) ;

}

}
```

```
//当前加载进程改变的时候触发

@Override

public void onProgressChanged (WebView view,int newProgress) {

//自定义标题

getWindow () .setFeatureInt (

Window.FEATURE_PROGRESS,newProgress*100) ;

//如果进度到达100

if (newProgress==100) {

//获取CookieSyncManager的实例

CookieSyncManager.getInstance () .sync () ;

}

}

}

}
```

9.3 浏览器源代码解析

9.3.1 WebView加载入口分析

初始化WebView最简单的方法如下：

```
WebView webview=new WebView (this) ;
```

下面是其初始化的具体过程。

1) 初始化WebView。

```
//传入上下文参数Context， 初始化WebView
```

```
public WebView (Context context) {this (context,null) ; }
```

2) 增加布局参数。

```
//AttributeSet为布局参数
```

```
public WebView (Context context,AttributeSet attrs) {
```

```
this (context,attrs,com.android.internal.R.attr.webViewStyle) ;
```

```
}
```

3) 增加默认布局参数。

//defStyle为默认布局参数

```
public WebView (Context context,AttributeSet attrs,int defStyle) {  
    this (context,attrs,defStyle,false) ;  
}
```

4) 增加选择参数，决定是否是私有模式。

//privateBrowsing决定是否是私有模式

```
public WebView (Context context,AttributeSet attrs,int defStyle,boolean  
privateBrowsing) {  
    this (context,attrs,defStyle,null,privateBrowsing) ;  
}
```

5) 调用内部保护函数，增加了JS接口。

//javaScriptInterfaces保存JS接口的MAP对象

```
protected WebView (Context context,AttributeSet attrs,int defStyle,  
Map<String,Object> javaScriptInterfaces,boolean privateBrowsing) {  
    super (context,attrs,defStyle) ;
```

```
//检查context

if (context==null) {

throw new IllegalArgumentException ("Invalid context argument") ;
```

```
}
```

//检查当前线程是否为主线程

```
checkThread () ;
```

//保证WebViewProvider对象可以访问Provider对象

```
ensureProviderCreated () ;
```

//初始化Provider对象

```
mProvider.init (javaScriptInterfaces,privateBrowsing) ;
```

```
}
```

WebViewProvider为WebView的后端提供者接口，每一个WebView对象被绑定到一个WebViewProvider，它实现了WebView的运行时行为。

WebViewProvider接口的定义如下：

```
public interface WebViewProvider{
```

```
//定义WebViewProvider对象变量  
  
private WebViewProvider mProvider ;  
  
//保证创建Provider对象  
  
private void ensureProviderCreated () {  
  
    //检查当前线程是否为主线程  
  
    checkThread () ;  
  
    if (mProvider==null) {  
  
        //创建WebView  
  
        mProvider=getFactory () .createWebView (this,new  
PrivateAccess () ) ;  
  
    }  
  
}  
  
}  
  
}
```

上述代码中，getFactory () 的定义如下：

```
//使用工厂类的目的是为了最少地使用WebViewClassic内部绑定

private static synchronized WebViewFactoryProvider getFactory () {

    //检查当前线程是否为主线程

    checkThread () ;

    //返回WebViewFactoryProvider类

    return WebViewFactory.getProvider () ;

}
```

上面是初始化WebView的简单接口调用关系，下面介绍如何加载数据接口，其中加载网址的方法具体代码如下：

```
webview.loadUrl ("http://www.android.com/") ;
```

其直接调用的代码如下：

```
//加载数据

//参数url为网址字符串

public void loadUrl (String url) {

    checkThread () ;
```

```
//调用WebViewProvider接口  
  
mProvider.loadUrl (url) ;  
  
}
```

也可以通过调用loadDataWithBaseURL方法加载数据接口，具体如下：

/*将指定的data加载到WebView中，该方法不能加载来自网络的内容。

参数MimeType，是该资源的媒体类型，可以取值text/html,image/jpeg等。

参数encoding为网页编码，可以取值utf-8，base64等。

参数baseUrl为基础目录，data中的文件路径可以是相对于基础目录的相对目录。 */

```
public void loadDataWithBaseURL (String baseUrl,String data,  
String mimeType,String encoding,String historyUrl) {  
  
checkThread () ;  
  
mProvider.loadDataWithBaseURL (baseUrl,data,  
mimeType,encoding,historyUrl) ;
```

}

或者调用loadData方法加载数据接口，其代码原型如下：

```
public void loadData (String data, String mimeType, String encoding) {  
    checkThread () ;  
  
    mProvider.loadData (data, mimeType, encoding) ;  
  
}
```

其中mimeType为数据的MIME类型，默认为text/html类型；encoding表示数据的编码类型。

如果是加载HTML格式的字符串，调用loadData方法的实现代码如下：

```
String summary = "<html><body>You scored <b>192</b> points.  
</body></html>" ;  
  
webview.loadData (summary, "text/html", null) ;
```

9.3.2 调用JavaScript接口

向WebView中注入指定的Java对象，首先该对象使用对象提供的名字被注入JavaScript中，这将允许JavaScript访问Java对象的公共方法。需要

注意的是，直到页面重载之前，注入的Java对象不会出现在JavaScript中。向WebView注入Java对象的调用方法如下：

```
webView.addJavascriptInterface (new Object () , "injectedObject") ;  
  
webView.loadData ("<title></title>" , "text/html" , null) ;  
  
webView.loadUrl ("javascript:alert (injectedObject.toString () ) ") ;
```

源代码中调用抽象接口的对应方法如下：

```
//添加JS对象  
  
public void addJavascriptInterface (Object object, String name) {  
  
    checkThread () ;  
  
    mProvider.addJavascriptInterface (object, name) ;  
  
}
```

其中，Object参数为JS对象；String参数为JS对象的名称。

移去对JS对象的调用方法如下：

```
//移除JS对象  
  
public void removeJavascriptInterface (String name) {
```

```
checkThread () ;  
  
mProvider.removeJavascriptInterface (name) ;  
  
}
```

注意 这种方法允许JavaScript控制主机应用程序。这是一个比较强大的功能，但也存在一定的安全风险，特别是JavaScript可以使用反射来访问注入对象的公共领域。当在WebView中含有不受信任的内容时，攻击者可能以意想不到的方式使用宿主主机应用程序的权限执行Java代码，所以使用这种方法时要格外小心。

9.4 WebKit简单分析

实现WebKit功能的Java层的代码位于
frameworks/base/core/java/android/webkit目录下，下面内容分析其中一
些比较重要的类和对象。

9.4.1 HTTP Cache管理

HTTP Cache管理对象定义在CacheManager.java类中，其负责对Java层
Cache对象的管理。当WebView被创建的时候，系统会自动的初始化
HTTP缓存。观察其初始化函数，其在初始化的时候通过getCacheFile
() 方法检查Cache文件是否存在，如果不存在则新建Cache文件。

//初始化Cache文件

```
static void init (Context context) {  
  
    //新建File对象  
  
    mBaseDir=new File (context.getCacheDir  
    () , "webviewCacheChromiumStaging") ;  
  
    //检测文件是否存在，如果不存在则创建
```

```
if ( ! mBaseDir.exists () ) {  
  
    mBaseDir.mkdirs () ;  
  
}  
  
}
```

代码中可以看出一部分Cache参数存储在数据库中，一部分不存储在数据库中，并且提供了相应的GET方法来获取其值。

//CacheResult提供了Cache中参数的获取方法

```
public static class CacheResult{
```

//下面的参数存储在数据库中

```
int httpStatusCode ;
```

```
long contentLength ;
```

```
long expires ;
```

```
String expiresString ;
```

```
String localPath ;
```

```
String lastModified ;
```

```
String etag ;  
  
String mimeType ;  
  
String location ;  
  
String encoding ;  
  
String contentdisposition ;  
  
String crossDomain ;  
  
//下面的参数没存储在数据库中  
  
InputStream inStream ;  
  
OutputStream outStream ;  
  
File outFile ;  
  
.....  
}
```

9.4.2 Cookie管理

CookieManager.java类根据RFC2109规范，用于管理Cookie。

CookieSyncManager.java类Cookie同步管理对象，该对象负责同步RAM

和Flash之间的Cookie数据。实际的物理数据操作在基类
WebSyncManager中完成。WebSyncManager的具体实现如下：

```
WebSyncManager implements Runnable{  
    CookieManager cookieManager=CookieManager.getInstance () ;  
    .....  
}
```

值得注意的是，用上面的方法获取CookieManager之前应该首先调用
CookieSyncManager.createInstance (Context) ，保证Cookie同步正常。

CookieSyncManager用于同步存储在RAM和持久存储的浏览器Cookie。
为了获得最佳性能，浏览器的Cookie被保存在RAM中。一个单独的线
程负责Cookie之间的同步，并由一个定时器来驱动。

CookieSyncManager的初始化方法如下：

```
CookieSyncManager.createInstance (context)
```

在Activity.onResume () 的生命周期的时候开始同步，方法如下：

```
CookieSyncManager.getInstance () .startSync ()
```

在Activity.onPause () 的生命周期的时候停止同步，其方法如下：

```
CookieSyncManager.getInstance () .stopSync ()
```

如果不希望等到定时器触发刷新，而是希望即时地调用刷新，可以使用如下的方法：

```
CookieSyncManager.getInstance () .sync ()
```

同步的核心代码如下：

```
protected void syncFromRamToFlash () {
```

```
.....
```

```
CookieManager manager=CookieManager.getInstance () ;
```

```
if ( ! manager.acceptCookie () ) {
```

```
return ;
```

```
}
```

```
manager.flushCookieStore () ;
```

```
.....
```

```
}
```

9.4.3 处理HTTP认证以及证书

HttpAuthHandler.java类负责处理HTTP认证请求。由WebView实例化这个类并传递到onReceivedHttpAuthRequest
(WebView,HttpAuthHandler,String,String) 中。主应用程序可以调用proceed (String,String) 或cancel () 来设置对WebView请求的响应。
HttpAuthHandler.java类的具体实现代码如下：

```
public class HttpAuthHandler extends Handler{  
  
    HttpAuthHandler () {}  
  
    //返回Http的认证结果  
  
    public boolean useHttpAuthUsernamePassword () {  
  
        return false ;  
  
    }  
  
    //取消当前认证  
  
    public void cancel () {}  
  
    //使用给定的认证信息进行认证  
  
    public void proceed (String username,String password) {}  
  
    .....  
}
```

}

ClientCertRequestHandler.java:ClientCertRequestHandler类负责处理客户端的证书请求。这个类作为BrowserCallback.displayClientCertRequestDialog的参数，接受用户响应。

9.4.4 处理JavaScript的请求

JsResult.java类为JS结果对象，用于用户交互。JsResult类的实例被作为WebChromeClient动作通知的一个参数传递。该对象作为JavaScript请求的句柄，在客户端指示此动作是否应继续进行。JsResult.java类的具体实现如下：

```
public class JsResult{  
  
    //回调接口  
  
    public interface ResultReceiver{  
  
        public void onJsResultComplete (JsResult result) ;  
  
    }  
  
    //定义回调接收
```

```
private final ResultReceiver mReceiver ;  
  
//保存确认或提示对话框的结果  
  
private boolean mResult ;  
  
//处理用户取消操作  
  
public final void cancel () {  
  
    mResult=false ;  
  
    wakeUp () ;  
  
}  
  
//处理用户确认操作  
  
public final void confirm () {  
  
    mResult=true ;  
  
    wakeUp () ;  
  
}  
  
public JsResult (ResultReceiver receiver) {
```

```
mReceiver=receiver ;  
}  
  
public final boolean getResult () {  
    return mResult ;  
}  
  
}  
  
//通知调用者JsResult已经完成  
  
private final void wakeUp () {  
    mReceiver.onJsResultComplete (this) ;  
}  
  
}
```

JsPromptResult.java类用于向用户提示JavasCript运行结果。

JsPromptResult.java类的具体实现如下：

```
public class JsPromptResult extends JsResult{  
  
    //定义运行结果提示字符串  
  
    private String mStringResult ;
```

```
//处理用户的确认信息

public void confirm (String result) {

    mStringResult=result ;

    confirm () ;

}

public JsPromptResult (ResultReceiver receiver) {

    super (receiver) ;

}

public String getStringResult () {

    return mStringResult ;

}

}
```

WebChromeClient类中核心的函数如下：

- onCloseWindow () 方法通知主机应用关闭WebView，必要时把它从界面系统移除。从这一点上讲，WebCore已经停止了任何窗体的加载，

并移除了JavaScript中任何跨脚本能力。onCloseWindow () 方法的具体实现如下：

```
//参数window为需要关闭的WebView
```

```
public void onCloseWindow (WebView window) {}
```

■onJsAlert () 方法告诉客户端呈现一个JavaScript的警告对话框。如果客户端返回ture,WebView将认为客户端将处理该对话框。如果客户端返回false，它将继续执行。onJsAlert () 方法的具体实现如下：

```
/*参数含义：view发起回调的WebView
```

```
url请求对话框的页面URL
```

```
message窗体将显示的消息
```

```
result确认用户点击enter的JsResult
```

```
*/
```

```
//返回布尔值，表示客户端是否处理该警告对话框
```

```
public boolean onJsAlert (WebView view,String url,String  
message,JsResult result) {
```

```
return false ;
```

}

■onJsConfirm () 方法告诉客户端给用户呈现一个确认对话框。如果客户端返回true,WebView认为客户端将处理该确认对话框并调用合适的JsResult方法。如果客户端返回false，一个代表false的默认值将返回给JavaScript。默认是返回false。onJsConfirm () 方法的具体实现如下：

//返回布尔值，表示客户端是否处理该确认对话框

```
bpublic boolean onJsConfirm (WebView view,String url,String  
message,JsResult result) {  
  
    return false ;  
  
}
```

■onJsPrompt () 方法告诉客户端给用户呈现一个提示对话框。如果客户端返回true,WebView认为客户端将处理该提示对话框并调用合适的JsPromptResult方法。如果客户端返回false，一个代表false的默认值将返回给JavaScript。默认是返回false。onJsPrompt () 方法的具体实现如下：

//返回布尔值，表示客户端是否处理该提示对话框

```
public boolean onJsPrompt (WebView view,String url,String message,
```

```
String defaultValue,JsPromptResult result) {  
  
    return false ;  
  
}
```

■onJsBeforeUnload () 方法告诉客户端给用户呈现一个确认是否离开当前页面导航的对话框。如果客户端返回true,WebView认为客户端将处理该确认对话框并调用合适的JsResult方法。如果客户端返回false,JavaScript离开当前页面导航。默认是返回false。onJsBeforeUnload () 方法具体实现如下：

//返回布尔值，表示客户端是否处理该确认对话框

```
public boolean onJsBeforeUnload (WebView view,String url,String  
message,JsResult result) {  
  
    return false ;  
  
}
```

9.4.5 处理MIME类型

一些方法是通过libcore.net.MimeUtils类来实现的，MimeUtils类里面定义了相关的MIME键值对和获取方法，其中一种方法是从文件路径里面获取MIME类型，代码实现如下：

//从文件路径获取MIME类型

```
public static String getFileExtensionFromUrl (String url) {
```

//保证路径不为空

```
if ( ! TextUtils.isEmpty (url) ) {
```

//返回字符串中#出现的最后的位置

```
int fragment=url.lastIndexOf ('#') ;
```

//从最后一个#处截取Url

```
if (fragment>0) {
```

```
url=url.substring (0, fragment) ;
```

```
}
```

//查找最后一个？出现的地方

```
int query=url.lastIndexOf ('?') ;
```

//从最后一个？处截取Url

```
if (query>0) {
```

```
url=url.substring (0, query) ;  
}  
  
int filenamePos=url.lastIndexOf ('/') ;  
  
String filename=0<=filenamePos?url.substring (filenamePos+1) : url ;  
  
//如果文件名包含特殊字符则无效  
  
if ( ! filename.isEmpty () & &  
    Pattern.matches ("[a-zA-Z_0-9\\\\.\\\\-\\\\ (\\\\) \\\\%]+", filename) ) {  
  
    int dotPos=filename.lastIndexOf ('.') ;  
  
    if (0<=dotPos) {  
  
        return filename.substring (dotPos+1) ;  
  
    }  
  
}
```

//如果没有发现则返回空字符串

```
return"";  
}  
}
```

9.4.6 访问WebView的历史

WebHistoryItem.java类用于保存网页历史数据，每个WebHistoryItem都是网页访问历史的快照。每个历史项在页面加载时都可能更新。

WebBackForwardList类包含了一个WebView的前后网页访问历史列表。

WebView.copyBackForwardList () 将返回该类的拷贝，用于检查列表中的项目。

WebBackForwarList对象维护用户访问的历史记录，该类为客户端提供操作访问浏览器历史数据的相关方法。从如下代码可以看到，定义了WebHistoryItem类型的列表变量为WebHistoryItem，用来存储访问历史记录，而mCurrentIndex则指向当前的页面在列表中的位置。

WebHistoryItem中存储了网址等相关信息。

```
private int mCurrentIndex ;
```

```
private ArrayList<WebHistoryItem>mArray ;
```

WebBackForwardListClient.java:WebBackForwardListClient对象定义了对访问历史操作时可能产生的事件接口，当用户实现了该接口，则在操

作访问历史时（访问历史移除、访问历史清空等）用户会得到通知。也就是当WebBackForwardList中的项产生变化的时候，通过该类来通知用户。

9.4.7 保存网站图标

WebIconDatabase.java类为图标数据库管理对象，所有的WebView均请求相同的图标数据库对象。WebIconDatabase中定义了WebView操作图标的功能。WebView.getIconDatabase () 将返回一个WebIconDatabase对象。WebIconDatabase对象是一个单例。

WebIconDatabase中定义了IconListener接口，在图标被检索出来的时候使用。

9.4.8 WebStorage

WebStorage是网络应用程序在网络浏览器存储数据的方法和通信协议。网络存储支持持久性数据存储（类似于Cookie）以及本地存储。

WebStorage被W3C标准化，其文本可在<http://www.w3.org/TR/webstorage/>处查阅。WebStorage最初是HTML5规范的一部分，现在已成为一个独立的规范。WebStorage可以被看作改进的Cookie，其可提供更大的存储容量和更好的编程接口。但是，它与Cookie在一些关键的地方并不相同：Cookie可以被客户端和服务器存

取，但WebStorage只限被客户端脚本（如Javascript）控制；WebStorage的资料并不会在每个HTTP请求下传送到服务器，网络服务器也不能直接把资料直接写入WebStorage中，只可以发出读取和写入请求。

WebStorage是用来管理WebView中JavaScript存储API的，包括管理应用程序缓存API、Web SQL数据库API及HTML5 Web存储API。应用程序缓存API提供了一种机制来创建和维护一个应用程序缓存去驱动离线Web应用程序。需要注意的是，每个应用程序只能有一个应用程序缓存。WebStorage封装了主机、模式和URI的端口等信息。

9.4.9 处理UI

BrowserFrame类负责URL资源的载入、访问历史的维护、数据缓存等操作，该类通过JNI接口直接与WebKit C层库交互。BrowserFrame类的具体实现如下：

```
//创建应用中使用的BrowserFrame

public BrowserFrame (Context context, WebViewCore w, CallbackProxy
proxy,
WebSettingsClassic settings, Map<String, Object> javascriptInterfaces) {

    Context appContext=context.getApplicationContext () ;

    //创建全局的JwebCoreJavaBridge处理WebCore线程中的定时器和Cookie
```

```
if (sJavaBridge==null) {  
  
    sJavaBridge=new JWebCoreJavaBridge () ;  
  
    //设置WebCore本地缓存的大小  
  
    ActivityManager am= (ActivityManager) context.getSystemService  
    (Context.ACTIVITY_SERVICE) ;  
  
    if (am.getMemoryClass () >16) {  
  
        sJavaBridge.setCacheSize (8*1024*1024) ;  
  
    }else{  
  
        sJavaBridge.setCacheSize (4*1024*1024) ;  
  
    }  
  
    //初始化CacheManager  
  
    CacheManager.init (appContext) ;  
  
    //创建CookieSyncManager  
  
    CookieSyncManager.createInstance (appContext) ;  
  
    //创建PluginManager
```

```
        PluginManager.getInstance (appContext) ;  
  
    }  
  
.....  
  
}
```

WebViewCore类是Java层与C层WebKit核心库的交互类，客户程序调用WebView的网页浏览相关操作会转发给BrowserFrame对象。当WebKit核心库完成实际的数据分析和处理后会回调WebViewCore中定义的一系列JNI接口，这些接口会通过CallbackProxy将相关事件通知相应的UI对象。WebViewCore类的具体实现如下：

```
//WebViewCore类是Java层与C层WebKit核心库的交互类  
  
public final class WebViewCore{  
  
    private static final String LOGTAG="webcore" ;  
  
    static{  
  
        try{  
  
            //装载库文件  
  
            System.loadLibrary ("webcore") ;
```

```
System.loadLibrary ("chromium_net") ;  
  
}catch (UnsatisfiedLinkError e) {  
  
    Log.e (LOGTAG, "Unable to load native support libraries.") ;  
  
}  
  
}  
  
}
```

CallbackProxy是一个代理类，用于UI线程和WebCore线程交互。该类定义了一系列与用户相关的通知方法，当WebCore完成相应的数据处理时，则调用CallbackProxy类中对应的方法，这些方法通过消息方式间接调用相应处理对象的处理方法。

//绘制窗口

```
public BrowserFrame createWindow (boolean dialog,boolean  
userGesture) {
```

//检测是否已经存在mWebChromeClient

```
if (mWebChromeClient==null) {  
  
    return null ;
```

}

WebView.WebViewTransport transport=mWebView.getWebView () .new
WebViewTransport () ;

final Message msg=obtainMessage (NOTIFY) ;

msg.obj=transport ;

sendMessageToUiThreadSync (obtainMessage
(CREATE_WINDOW,dialog?1:0, userGesture?1:0, msg)) ;

WebViewClassic w=WebViewClassic.fromWebView
(transport.getWebView ()) ;

if (w !=null) {

WebViewCore core=w.getWebViewCore () ;

//如果WebView.destroy () 已经被调用， core可能为空， 此处加入检测

if (core !=null) {

core.initializeSubwindow () ;

return core.getBrowserFrame () ;

```
 }  
  
}  
  
return null ;
```

```
}
```

ViewManager类负责子视图管理类，主要用于管理插件视图。

以下代码定义了mChildren列表来存储子视图。

```
private final ArrayList<ChildView>mChildren=new ArrayList<  
ChildView> () ;
```

子视图的定义如下：

//子窗口结构

```
class ChildView{
```

//边界

```
int x ; int y ;
```

//宽和高

```
int width ; int height ;
```

```
//定义显示的视图
```

```
View mView ;
```

```
ChildView () {}
```

```
//设置赋值函数
```

```
void setBounds (int x,int y,int width,int height) {
```

```
this.x=x ; this.y=y ;
```

```
this.width=width ; this.height=height ;
```

```
}
```

```
void attachView (int x,int y,int width,int height) {
```

```
if (mView==null) {return ;}
```

```
setBounds (x,y, width,height) ;
```

```
mWebView.mPrivateHandler.post (new Runnable () {
```

```
public void run () {
```

```
//该方法可以多次调用，如果该视图已经安装则只设置新的布局参数
```

```
//否则添加视图，并添加到mChildren列表
```

```
requestLayout (ChildView.this) ;
```

```
if (mView.getParent () ==null) {
```

```
attachViewOnUIThread () ;
```

```
}
```

```
}
```

```
) ;
```

```
}
```

```
//添加到界面
```

```
private void attachViewOnUIThread () {
```

```
mWebView.getWebView () .addView (mView) ;
```

```
mChildren.add (this) ;
```

```
if ( ! mReadyToDraw) {
```

```
mView.setVisibility (View.GONE) ;
```

}

}

}

9.4.10 Web设置分析

WebSettings.java类中描述了Web浏览器访问相关的用户配置信息，是WebView管理设置数据的对象，该对象数据是通过JNI接口从底层获取的。

WebSettings类管理WebView的设置状态。当WebView第一次创建时，它获得一组默认设置。WebView.getSettings () 方法可以返回这些默认设置。WebView.getSettings () 调用必须在WebView的生命周期里面。如果一个Webview已被销毁，调用WebSettings的任何方法都会抛出一个IllegalStateException异常。

9.4.11 HTML5音视频处理

HTML5是下一代的HTML，虽然现在HTML5仍处于完善之中，但是大部分现代浏览器已经具备了某些HTML5支持。

HTML5中的一些有趣的新特性如下：

- 用于绘画的Canvas元素。
 - 用于媒介回放的video和audio元素。
 - 对本地离线存储的更好的支持。
 - 新的特殊内容元素，比如article、footer、header、nav、section。
 - 新的表单控件，比如calendar、date、time、email、url、search。
- 最新版本的Safari、Chrome、Firefox、Opera及Internet Explorer 9都已支持某些HTML5特性。HTML5Audio.java是HTML5Audio支持类，HTML5Audio中加入了对MediaPlayer对象的事件监听，并且有相关初始化、缓冲、播放、完成等调用方法的封装。下面的代码中可以看到其扩展的接口。

```
class HTML5Audio extends Handler implements  
    //监听网络上的媒体资源  
    MediaPlayer.OnBufferingUpdateListener,  
    //监听媒体是否播放完成  
    MediaPlayer.OnCompletionListener,  
    //错误监听
```

MediaPlayer.OnErrorListener,

//监听是否准备好随时播放

MediaPlayer.OnPreparedListener,

//监听是否寻道完成

MediaPlayer.OnSeekCompleteListener,

//监听音频系统的焦点是否被更新

AudioManager.OnAudioFocusChangeListener

HTML5VideoView.java负责视频视图，提供HTML5视频处理的核心类，仅提供给浏览器使用。

其中定义了HTML5VideoViewProxy类如下：

protected HTML5VideoViewProxy mProxy ;

使用代理类HTML5VideoViewProxy，完成功能播放、暂停、获取进度等功能。

protected void switchProgressView (boolean playerBuffering) {

//只适用于HTML5VideoFullScreen中

```
}

public boolean surfaceTextureDeleted () {

//只适用于HTML5VideoInline

return false ;

}
```

HTML5VideoFullScreen.java负责全屏视频视图，仅提供给浏览器使用；HTML5VideoInline.java为内嵌视频视图，仅提供给浏览器使用；HTML5VideoViewProxy.java为HTML5视频视图代理类。

```
//HTML5VideoViewProxy实例化，返回HTML5VideoViewProxy对象

public static HTML5VideoViewProxy getInstance (WebViewCore
webViewCore,int nativePtr) {

return new HTML5VideoViewProxy (webViewCore.getWebViewClassic
() , nativePtr) ;

}
```

9.4.12 缩放和下载

ZoomControlBase.java为缩放控件接口；ZoomControlEmbedded.java为内置缩放控件，其实现了ZoomControlBase的接口。下面为ZoomControlBase接口的代码内容，该接口提供了控制当前缩放控件是否可见和控制缩放级别的抽象方法。

```
interface ZoomControlBase{  
  
    //使得屏幕上的缩放控制可见  
  
    public void show () ;  
  
    //隐藏缩放控制  
  
    public void hide () ;  
  
    /*根据页面中缩放级别的变化，可以控制其允许缩放的状态。例如，如果达到的最大放大级别，那么可以禁用放大按键*/  
  
    public void update () ;  
  
    //检查当前缩放控件是否可见  
  
    public boolean isVisible () ;  
  
}
```

ZoomManager.java类维护WebView的缩放状态和WebView当前缩放级别，并负责管理屏幕上的缩放控件和缩放时候的动画。

目前，有以下两种驱动WebView缩放控件的方法：

- 通过startZoomAnimation () 方法触发。其在启动时候就已经知道的最终缩放尺寸。这种类型的缩放在其缩放之前会通知WebKit其最终的大小。
- 由多点触摸来触发。根据用户的手势来决定缩放。这种类型只在手势完成之后通知WebKit新的缩放尺寸。

android.webkit.DownloadListener里面定义了以下接口类：

```
public interface DownloadListener{  
    //通知应用程序，文件将被下载  
    public void onDownloadStart (String url,String userAgent,  
        String contentDisposition,String mimetype,long contentLength) ;  
}
```

9.4.13 插件管理

Plugin.java是定义插件的类，其定义如下：

```
public Plugin (String name,String path,String fileName,String  
description)
```

PluginData.java负责插件数据，封装了一个插件生成的内容，能够构造一个HTTP响应，包含有响应主体、主体的长度、响应头和响应状态码。

PluginData类是插件相关数据的容器，其具体实现如下：

```
public PluginData (InputStream stream,long length,Map<String,String[]>  
headers,int code) {  
  
    mStream=stream ;  
  
    mContentLength=length ;  
  
    mHeaders=headers ;  
  
    mStatusCode=code ;  
  
    .....  
}
```

PluginFullScreenHolder.java负责插件处理相关，获取插件视图的容器的大小，并负责显示等操作。PluginFullScreenHolder的具体实现如下：

```
//定义了插件视图的容器

private static CustomFrameLayout mLayout ;

private View mContentView ;

PluginFullScreenHolder (WebViewClassic webView,int orientation,int
npp) {

mWebView=webView ;

mNpp=npp ;

mOrientation=orientation ;

}

public void setContentView (View contentView) {

//为插件视图创建布局文件

mLayout=new CustomFrameLayout (mWebView.getContext () ) ;

FrameLayout.LayoutParams layoutParams=new
FrameLayout.LayoutParams (
```

```
ViewGroup.LayoutParams.MATCH_PARENT,ViewGroup.LayoutParams.M
ATCH_
PARENT,Gravity.CENTER) ;
mLayout.addView (contentView,layoutParams) ;
mLayout.setVisibility (View.VISIBLE) ;
if (contentView instanceof SurfaceView) {
final SurfaceView sView= (SurfaceView) contentView ;
if (sView.isFixedSize ()) {
sView.getHolder () .setSizeFromLayout () ;
}
}
mContentView=contentView ;
}
```

PluginList.java为一个初始化插件的列表。在浏览器启动的时候，该表填充一个插件列表。该类包含了列表添加、获取、清空等方法。

PluginManager.java负责管理系统中安装的插件和WebViewClassic类之间的关系，其具体实现如下：

//获取插件权限和签名

containsPluginPermissionAndSignatures ()

//获取插件目录

getPluginDirectories ()

//获取插件APK名称

getPluginsAPKName (String)

//获取插件数据目录

getPluginSharedDataDirectory ()

//刷新插件

refreshPlugins ()

PluginStub是WebView实现插件的接口。任何一个插件包都可以继承该类，实现抽象函数来创建嵌入或全屏显示在WebView的视图。

PluginStub的具体实现如下：

```
public interface PluginStub{  
  
    public abstract View getEmbeddedView (int NPP,Context context) ;  
  
    public abstract View getFullScreenView (int NPP,Context context) ;  
  
}
```

9.5 小结

本章首先介绍了Android HTML处理相关的关键类，给出了各类的简要说明。重点讲解了WebView对HTML文档的处理，涉及WebView加载入口的分析及在WebView中如何调用JavaScript等。最后，本章解析了WebKit内核，逐一分析了WebKit下一些比较重要的类。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第 10 章 Android网络处理分析

本章主要分析android.net包下面的源代码：首先分析Android网络处理的流程；然后简介Android封装的HTTP处理类；最后分析与RTP协议和SIP协议相关的源码。

10.1 Android网络处理关键类及其说明

在正式开始本章的分析之前，我们先简单介绍一下Android网络处理关键类。这些类在源代码中的位置为
frameworks/base/core/java/android/net。各类的简要说明如下。

- ConnectivityManager.java：该类用于查询网络连接的状态。
- Credentials.java：一个代表UNIX身份信息（通过UNIX域套接字中辅助数据传送）的类。
- DhcpInfo.java：代表了一个DHCP请求结果的对象。
- LocalServerSocket.java：用于在Android平台上创建下行UNIX域套接字的非标准类，它是在Linux非文件系统命名空间中创建的。
- LocalSocket.java：在UNIX域命名空间中创建一个非服务器socket。
- LocalSocketAddress.java：是一个UNIX域（AF_LOCAL）的socket地址。
- MailTo.java：这个类解析mailto协议格式的URL，然后可以查询解析后的参数。
- NetworkInfo.java：描述了网络接口的状态。

- `Proxy.java`：一个用于访问用户及默认代理设置的类。
- `SSLCertificateSocketFactory.java`：该类继承自`SSLSocketFactory`类。
`SSLSocketFactory`类实现了SSL操作的一些额外细节，比如规定了SSL握手操作超时规范等。
- `SSLSessionCache.java`：代表了SSL会话缓存。
- `TrafficStats.java`：一个提供了网络流量统计的类。
- `Uri.java`：代表一个不可变的URI引用。
- `Uri.Builder.java`：一个用于创建和操作URI引用的辅助类。
- `UrlQuerySanitizer.java`：该类用于过滤URL查询。
- `UrlQuerySanitizer.IllegalCharacterValueSanitizer.java`：用它们包含的字符来过滤非法值。
- `UrlQuerySanitizer.ParameterValuePair.java`：一个包含了参数值对的简单单元组。
- `VpnService.java`：它是应用扩展和建立自己的VPN解决方案的基类。
- `VpnService.Builder.java`：用于创建一个VPN接口的辅助类。

10.2 Android网络处理流程

10.2.1 监控网络连接状态

Java层负责网络连接状态监控的是ConnectivityManager.java类，该类实例化的方法如下：

```
Context context=icontext.getApplicationContext () ;
```

```
ConnectivityManager connectivity= (ConnectivityManager)  
context.getSystemService (Context.CONNECTIVITY_SERVICE) ;
```

ConnectivityManager.java类主要功能如下：

- 监视网络连接状态，如Wi-Fi、GPRS、UMTS等。

- 当网络连接改变的时候发送广播通知。

- 当一个网络连接断开的时候，尝试另外一个连接。

- 为应用程序查询网络状态提供接口。

ConnectivityManager.java类中定义了网络类型，相关源码如下：

```
/*APN (Access Point Name) ，即“接入点名称”，是通过手机上网时必  
须配置的一个参数，
```

它决定了手机通过哪种接入方式来访问网络，用来标识GPRS的业务种类*/

```
public static final int TYPE_NONE=-1 ;
```

//默认的移动数据连接

```
public static final int TYPE_MOBILE=0 ;
```

//默认Wi-Fi数据连接

```
public static final int TYPE_WIFI=1 ;
```

/*TYPE_MOBILE_MMS类型的连接与TYPE_MOBILE有时候相同，有时候不同。

当应用需要和运营商的多媒体信息服务服务器通信时使用。它可能与默认的数据连接共存*/

```
public static final int TYPE_MOBILE_MMS=2 ;
```

/*SUPL（Secure User Plane Location，安全用户平面定位）的移动数据连接。

这个连接可能与TYPE_MOBILE相同，但也可能不同。

当应用需要和运营商的安全用户平面定位服务器通信以定位设备时使用。

它可能与默认的数据连接共存*/

```
public static final int TYPE_MOBILE_SUPL=3 ;
```

/*一个DUN (Dial Up Networking, 拨号网络) 的移动数据连接。

这个连接可能与TYPE_MOBILE相同，但也可能不同。

当应用需要执行一个拨号网络桥接器以便运营商知道网络流量时使用。

它可能与默认的数据连接共存*/

```
public static final int TYPE_MOBILE_DUN=4 ;
```

/*一个高优先级的数据连接。这个连接一般与TYPE_MOBILE相同，但路由设置不同。

仅当请求过程有权访问移动DNS服务器，且必须是通过
requestRouteToHost请求的IP，

在默认路由存在时才能通过该接口路由*/

```
public static final int TYPE_MOBILE_HIPRI=5 ;
```

//默认的WiMAX数据连接。当被激活时，所有数据流将默认使用该连接

```
public static final int TYPE_WIMAX=6 ;
```

//默认的蓝牙连接。当被激活时，所有数据流将默认使用该连接

```
public static final int TYPE_BLUETOOTH=7 ;
```

//虚拟数据连接

```
public static final int TYPE_DUMMY=8 ;
```

//默认的以太网数据连接。当被激活时，所有数据流将默认使用该连接

```
public static final int TYPE_ETHERNET=9 ;
```

//通过无线管理，FOTA（Firmware Over The Air Updates，固件无线更新）

```
public static final int TYPE_MOBILE_FOTA=10 ;
```

//IP多媒体子系统

```
public static final int TYPE_MOBILE_IMS=11 ;
```

//运营商的品牌服务

```
public static final int TYPE_MOBILE_CBS=12 ;
```

//Wi-Fi点对点连接。请求过程必须有权与接入点连接

```
public static final int TYPE_WIFI_P2P=13 ;
```

系统中对网络的判断和选择是在ConnectivityService.java服务中处理的，在系统启动的时候会启动这个系统服务。ConnectivityManager.java类通过aidl访问ConnectivityService.java提供的服务，相关源码如下：

.....

```
private final IConnectivityManager mService ;
```

.....

```
public ConnectivityManager (IConnectivityManager service) {
```

```
    mService=checkNotNull (service, "missing IConnectivityManager") ;
```

```
}
```

.....

/*该方法在当前活动的数据网络流量被测量时返回。

网络根据流量进行分类，如果用户对大量数据传输很敏感，那么在做大量数据传输之前应该检查连接，并告知用户或者延迟操作直到另一个网络是可用的*/

```
public boolean isActiveNetworkMetered () {  
  
try{  
  
return mService.isActiveNetworkMetered () ;  
  
}catch (RemoteException e) {  
  
return false ;  
  
}  
  
}
```

10.2.2 认证类

认证类Credentials.java是一个代表UNIX身份信息（通过UNIX域套接字中辅助数据传送）的类，它包含了进程的pid、uid和gid信息。

Credentials.java类的定义如下：

```
public class Credentials{  
  
//Linux进程的pid信息
```

```
private final int pid ;  
  
//Linux进程的uid信息  
  
private final int uid ;  
  
//Linux进程的gid信息  
  
private final int gid ;  
  
public Credentials (int pid,int uid,int gid) {  
  
    this.pid=pid ;  
  
    this.uid=uid ;  
  
    this.gid=gid ;  
  
}  
  
//获取pid  
  
public int getPid () {  
  
    return pid ;  
  
}
```

```
//获取uid  
  
public int getUid () {  
  
    return uid ;  
  
}  
  
//获取gid  
  
public int getGid () {  
  
    return gid ;  
  
}  
  
}
```

10.2.3 DHCP状态机

动态主机设置协定协议（Dynamic Host Configuration Protocol,DHCP）是一个局域网协议，DHCP通过UDP协议可以为用户自动分配IP地址或者允许内部网络管理员对所有计算机进行集中管理。

Android中使用DHCP状态机来管理DHCP协议，DHCP状态机可以与本地DHCP客户端交互，同时可以与同是状态机的控制器交流。DHCP状态机提供以下特性：

- 使用本地DHCP客户端唤醒和续约。当设备在暂停状态时，它将不续约自己，可能导致设备持有的IP地址超期。
- 在DHCP请求和续约启动前接收到通知。这个可以用于DHCP前任何附加的启动。

DHCP状态机通过DhcpInfo.java类给出一个DHCP请求结果。

DhcpInfo.java类包含了相关的网络信息，其定义如下：

//定义了一个代表DHCP请求结果的对象

```
public class DhcpInfo implements Parcelable{
```

//IP地址

```
public int ipAddress ;
```

//网关

```
public int gateway ;
```

//子网掩码

```
public int netmask ;
```

//首选DNS

```
public int dns1 ;  
  
//备用DNS  
  
public int dns2 ;  
  
//服务器地址  
  
public int serverAddress ;  
  
.....  
}
```

//实现Parcelable接口

```
public static final Creator<DhcpInfo> CREATOR=new Creator<DhcpInfo>()  
> {  
  
    public DhcpInfo createFromParcel (Parcel in) {  
  
        DhcpInfo info=new DhcpInfo () ;  
  
        info.ipAddress=in.readInt () ;  
  
        info.gateway=in.readInt () ;  
  
        info.netmask=in.readInt () ;  
    }  
}
```

```
info.dns1=in.readInt () ;  
  
info.dns2=in.readInt () ;  
  
info.serverAddress=in.readInt () ;  
  
info.leaseDuration=in.readInt () ;  
  
return info ;  
  
}  
  
public DhcpInfo[]newArray (int size) {  
  
return new DhcpInfo[size] ;  
  
}  
  
};
```

10.2.4 LocalServerSocket

LocalServerSocket.java类是用于在Android平台上创建下行UNIX域套接字的非标准类，它是在Linux非文件系统命名空间中创建的。在仿真平台上，它可在文件系统的临时目录中创建。该类的定义如下：

```
public class LocalServerSocket{
```

```
private final LocalSocketImpl impl ;  
  
private final LocalSocketAddress localAddress ;  
  
private static final int LISTEN_BACKLOG=50 ;  
  
//创建一个新的服务器socket监听特定的名字。在Android平台中，名字  
由Linux创建  
  
public LocalServerSocket (String name) throws IOException{  
    impl=new LocalSocketImpl () ;  
  
    impl.create (true) ;  
  
    localAddress=new LocalSocketAddress (name) ;  
  
    impl.bind (localAddress) ;  
  
    impl.listen (LISTEN_BACKLOG) ;  
}  
  
//利用已经创建与绑定的文件描述符创建一个LocalServerSocket。listen  
() 将立刻被调用  
  
//用于文件描述符是通过环境变量传进来的场合
```

//参数fd绑定文件描述符

```
public LocalServerSocket (FileDescriptor fd) throws IOException{
```

```
    impl=new LocalSocketImpl (fd) ;
```

```
    impl.listen (LISTEN_BACKLOG) ;
```

```
    localAddress=impl.getSockAddress () ;
```

```
}
```

//获得socket的本地地址

```
public LocalSocketAddress getLocalSocketAddress () {
```

```
    return localAddress ;
```

```
}
```

//接受一个新的socket连接。它将处于阻塞状态，直到有一个新的连接

```
public LocalSocket accept () throws IOException{
```

```
    LocalSocketImpl acceptedImpl=new LocalSocketImpl () ;
```

```
    impl.accept (acceptedImpl) ;
```

```
        return new LocalSocket (acceptedImpl) ;  
  
    }  
  
    //返回文件描述符或者空 (如果还没有打开或者已经关闭)  
  
    public FileDescriptor getFileDescriptor () {  
  
        return impl.getFileDescriptor () ;  
  
    }  
  
    //关闭服务器socket  
  
    public void close () throws IOException{  
  
        impl.close () ;  
  
    }  
  
}
```

客户端Socket通信的建立需要使用LocalSocket类创建，该类用于在UNIX域命名空间中创建一个非服务器Socket。LocalSocket类的定义如下：

```
public class LocalSocket{
```

```
private LocalSocketImpl impl ;  
  
private volatile boolean implCreated ;  
  
private LocalSocketAddress localAddress ;  
  
private boolean isBound ;  
  
private boolean isConnected ;  
  
//创建一个AF_LOCAL/UNIX域socket  
  
public LocalSocket () {  
  
    this (new LocalSocketImpl () ) ;  
  
    isBound=false ;  
  
    isConnected=false ;  
  
}  
  
//供AndroidServerSocket使用  
  
LocalSocket (LocalSocketImpl impl) {  
  
    this.impl=impl ;
```

```
this.isConnected=false ;
```

```
this.isBound=false ;
```

```
}
```

```
}
```

LocalSocketAddress.java类是一个UNIX域（AF_LOCAL）的Socket地址，供android.net.LocalSocket和android.net.LocalServerSocket使用。

LocalSocketAddress类的定义如下：

```
public class LocalSocketAddress{
```

```
//该地址所在的命名空间
```

```
public enum Namespace{
```

```
//一个Linux抽象命名空间中的socket
```

```
ABSTRACT (0) ,
```

```
/*在Android保留命名空间in/dev/socket中的socket，
```

```
仅仅初始进程可以在这里创建一个socket*/
```

```
RESERVED (1) ,
```

```
/*以正常文件系统路径命名的socket*/
FILESYSTEM (2) ;

/*id跟include/cutils/sockets.h中#define定义的匹配*/
private int id ;

Namespace (int id) {

    this.id=id ;

}

//返回共享的整型常量

int getId () {

    return id ;

}

private final String name ;

private final Namespace namespace ;
```

//通过一个给定的名字创建一个实例

```
public LocalSocketAddress (String name, Namespace namespace) {  
    this.name=name ;  
  
    this.namespace=namespace ;  
  
}
```

//通过一个命名空间中给定的名字创建一个实例

```
public LocalSocketAddress (String name) {  
    this (name,Namespace.ABSTRACT) ;  
  
}
```

//获得地址字符串的名字

```
public String getName () {  
    return name ;  
  
}
```

//返回该地址的命名空间

```
public Namespace getNamespace () {  
    return namespace ;  
}  
  
}
```

10.2.5 响应邮件请求

在Android中，当给出发送电子邮件的URL的时候，会调用MailTo.java这个类来响应请求。MailTo.java类用于解析mailto协议格式的URL，然后可以查询解析后的参数。它在文档RFC 2368中实现。

MailTo.java类在响应邮件请求的时候常用到的方法如下。

■getBody () : 从URL中获取邮件的正文内容。如果没有设置正文内容，则返回null。

■getCc () : 从URL中获取抄送地址。如果是多个电子邮件地址，则以逗号分隔。如果没有指定抄送地址，则返回null。

■getHeaders () : 从URL中获取邮件头。

■getSubject () : 从URL中获取邮件主题。如果没有指定的主题行，则返回null。

- `getTo()`：从URL中获取收信人。如果是多个电子邮件地址，则以逗号分隔。如果没指定，则返回null。
- `isMailTo()`：测试指定的字符串是否是符合mailto协议的URL。
- `parse()`：解析mailto格式的URL字符串。这个解析器的功能在RFC 2368中有描述。解析的参数可以通过返回的对象查询。

MailTo类的定义源码如下：

```
public class MailTo{  
  
    static public final String MAILTO_SCHEME="mailto：" ;  
  
    //所有被解析的内容都被加到headers中  
  
    private HashMap<String,String>mHeaders ;  
  
    //邮件头header的字段  
  
    static private final String TO="to" ;  
  
    static private final String BODY="body" ;  
  
    static private final String CC="cc" ;  
  
    static private final String SUBJECT="subject" ;
```

/*测试指定的字符串是否是符合mailto协议的URL，如果该字符串是符合mailto协议的URL则返回ture*/

```
public static boolean isMailTo (String url) {  
  
    if (url !=null & & url.startsWith (MAILTO_SCHEME) ) {  
  
        return true ;  
  
    }  
  
    return false ;  
  
}
```

/*解析mailto格式的URL字符串。这个解析器的功能在RFC 2368中有描述。

解析的参数可以通过返回的对象查询。参数url包含一个mailto的URL。
返回的是MailTo对象。

如果参数不是一个mailto的URL则会产生ParseException异常*/

```
public static MailTo parse (String url) throws ParseException{  
  
    if (url==null) {
```

```
throw new NullPointerException () ;  
}  
  
if ( ! isMailTo (url) ) {  
  
throw new ParseException ("Not a mailto scheme") ;  
}  
  
//Uri解析器无法处理时则退出  
  
String noScheme=url.substring (MAILTO_SCHEME.length ()) ;  
  
Uri email=Uri.parse (noScheme) ;  
  
MailTo m=new MailTo () ;  
  
//解析查询参数  
  
String query=email.getQuery () ;  
  
if (query !=null) {  
  
String[]queries=query.split (" & ") ;  
  
for (String q:queries) {
```

```
String[]nameval=q.split ("=") ;  
  
if (nameval.length==0) {  
  
continue ;  
  
}  
  
//以小写字母形式插入header以便更容易发现header  
  
m.mHeaders.put (Uri.decode (nameval[0]) .toLowerCase () ,  
  
nameval.length>1?Uri.decode (nameval[1]) : null) ;  
  
}  
  
}  
  
//地址在指定的header和mailto行之后， 把两个连接在一起  
  
String address=email.getPath () ;  
  
if (address !=null) {  
  
String addr=m.getTo () ;  
  
if (addr !=null) {
```

```
address+="," +addr ;  
}  
  
m.mHeaders.put (TO,address) ;  
  
}  
  
return m ;  
}
```

/*从被解析的mailto URL中检索收件人地址行（To address line）。

这可能是几个用逗号分隔的电子邮件地址。如果没有指定收件人地址行，则返回null。

方法返回逗号分隔的电子邮件地址或null*/

```
public String getTo () {  
    return mHeaders.get (TO) ;  
}
```

/*从被解析的mailto URL中检索抄送地址行（CC address line）。

这可能是几个用逗号分隔的电子邮件地址。

如果没有指定抄送行，则返回null。

方法返回逗号分隔的电子邮件地址或null*/

```
public String getCc () {
```

```
    return mHeaders.get (CC) ;
```

```
}
```

/*从被解析的mailto URL中检索标题行 (subject line) 。

如果没有指定标题行，则返回null。

方法返回标题或null*/

```
public String getSubject () {
```

```
    return mHeaders.get (SUBJECT) ;
```

```
}
```

/*从被解析的mailto URL中检索邮件体行 (body line) 。

如果没有指定邮件体行，则返回null。

方法返回邮件体或null*/

```
public String getBody () {  
    return mHeaders.get (BODY) ;  
}  
  
/*从mailto URL中检索所有被解析的邮件头。  
返回包含所有解析值的map*/  
  
public Map<String,String>getHeaders () {  
    return mHeaders ;  
}  
  
.....  
  
/*私有构造函数。建立一个Mailto对象的唯一方法是通过parse () 方法  
*/  
  
private MailTo () {  
    mHeaders=new HashMap<String,String> () ;  
}  
}
```

10.2.6 提供网络信息

NetworkInfo类用于描述网络接口的状态，而其子类

NetworkInfo.DetailedState用于描述网络连接更详细的状态；子类

NetworkInfo.State用于描述较为粗化的网络状态。

使用ConnectivityManager.getActiveNetworkInfo () 可以获得一个代表当前网络连接的实例，返回值为NetworkInfo类，返回当前活动的数据网络的详细信息。当连接时，这个返回的网络是对外连接的默认路由，应该在初始化网络流之前使用isConnected () 进行判断，如果没有网络可用方法将返回null。

NetworkInfo类里面定义的网络状态如表10-1所示。

表 10-1 NetworkInfo 类定义的网络状态

网络状态	含 义
IDLE	准备开始一个数据连接的建立
SCANNING	搜索可用的接入点
CONNECTING	目前正在建立数据连接
AUTHENTICATING	网络连接已经建立，正在执行验证
OBTAINING_IPADDR	等待 DHCP 服务器的响应以便分配 IP 地址信息
CONNECTED	IP 须是可用的
SUSPENDED	IP 被终止使用
DISCONNECTING	目前正在断开数据连接
DISCONNECTED	IP 无法到达
FAILED	试图连接失败
BLOCKED	尝试接入该网络被阻塞
VERIFYING_POOR_LINK	网络连接很差
Message.TYPE.ERROR	错误的消息

NetworkInfo类内部常用到的方法如下：

- getDetailedState () : 通过该方法可以获得当前详细的网络状态信息。
- getExtraInfo () : 通过该方法可以获得网络状态的额外信息，如果低的网络层提供且可获取。
- getReason () : 如果网络连接可用但建立连接失败，可通过该方法获取连接失败原因。
- getState () : 通过该方法可以获得当前粗化的网络状态信息。
- getSubtype () : 该方法返回一个网络类型特定的整型量，以描述网络的子类型。
- getSubtypeName () : 该方法返回描述网络子类型的名字。
- getType () : 通过该方法可以获得网络类型。
- getTypeName () : 该方法返回描述网络类型的名字，比如Wi-Fi或者MOBILE。
- isAvailable () : 指示网络连接是否可用。

- `isConnected ()`：指示网络连接是否存在且能否建立连接并传输数据。
- `isConnectedOrConnecting ()`：指示网络连接是否存在或者正在建立连接过程中。
- `isRoaming ()`：指示设备当前是否在该网络中漫游。

10.2.7 Proxy类

Proxy.java类是一个用于访问用户及默认代理设置的类。其相应的Intent标识字符串如下：

"android.intent.action.PROXY_CHANGE"

该Intent用于通知应用程序在默认连接或其代理发生改变时缓存默认的连接代理，并有一个存储信息的值——`EXTRA_PROXY_INFO`，表示Proxy的属性（proxyproperties）。Intent一般情况下是非空值，但如果代理是未定义的，则主机字符串为空。另外这是一个受保护的Intent，只能由系统发送。

上面的Intent的定义在Proxy.java类中，代码如下：

```
@SdkConstant (SdkConstantType.BROADCAST_INTENT_ACTION)  
public static final String PROXY_CHANGE_ACTION=
```

```
"android.intent.action.PROXY_CHANGE" ;
```

Proxy.java类中一些利用正则表达式解析网址的代码如下：

```
public static final String EXTRA_PROXY_INFO="proxy" ;
```

```
private static ConnectivityManager sConnectivityManager=null ;
```

```
//正则表达式验证主机名/IP
```

```
//匹配空白输入、ip及域名
```

```
private static final String NAME_IP_REGEX=
```

```
"[a-zA-Z0-9]+ (\-[a-zA-Z0-9]+) * (\.[a-zA-Z0-9]+ (\-[a-zA-Z0-9]+)* ) *" ;
```

```
private static final String
```

```
HOSTNAME_REGEXP="^$|^"+NAME_IP_REGEX+"$" ;
```

```
private static final Pattern HOSTNAME_PATTERN ;
```

```
private static final String EXCLLIST_REGEXP=
```

```
"$|^ (.?" + NAME_IP_REGEX + ") + (, (.?" + NAME_IP_REGEX + ")) *$" ;
```

```
private static final Pattern EXCLLIST_PATTERN ;  
  
static{  
  
    HOSTNAME_PATTERN=Pattern.compile (HOSTNAME_REGEX) ;  
  
    EXCLLIST_PATTERN=Pattern.compile (EXCLLIST_REGEX) ;  
  
}
```

10.2.8 VPN服务

VPN服务主要通过VpnService类提供。VpnService类是应用扩展和建立自己的VPN解决方案的基类，其辅助类为VpnService.Builder。一般情况下，它将创建一个虚拟的网络接口、配置地址和路由选择规则，并给应用返回一个文件描述符。从描述符获取接口路由过来的数据包。同样地，将数据包写入描述符就像从接口接收数据一样。接口运行在IP之上，所以数据包通常以IP报头开始。应用接着通过establish () 方法在某个通道上与远程服务器处理和交换数据包来完成VPN连接。

一个VPN应用可以轻易地破坏网络，并且允许应用拦截数据包，这样大大增加了安全风险。此外，两个VPN之间也会有冲突。系统需要采取一些措施解决这些问题，下面列出的是这些措施的要点：

- 在用户有操作需要时才去创建VPN连接。

- 在同一时间只能有一个VPN连接运行，也就是说只有现有的接口被停用时，才能创建一个新的VPN。
- 系统管理公告在VPN连接的整个生命周期中都要显示。
- 系统管理对话框将给出当前VPN连接的信息，并且还提供了一个断开连接的按钮。
- 保证关闭文件描述符时网络能自动恢复。这还包括VPN应用系统崩溃或VPN应用被系统终止的情况。

在VpnService类中有两个主要方法：prepare（Context）和establish（）。前者处理用户操作，终止由其他应用创建的VPN连接；后者使用VpnService.Builder提供的参数创建一个VPN接口。一个应用必须调用prepare（Context）去授予在该类中使用其他方法的权限，并可随时撤销权限。下面是创建一个VPN连接的主要步骤：

步骤1 当用户单击“连接”按钮时，调用prepare（Context）并发起返回的Intent。

步骤2 应用准备好了，启动服务。

步骤3 创建一个连接服务器的通道，为VPN连接协调好网络参数。

步骤4 提供这些参数给VpnService.Builder，并调用establish () 创建VPN接口。

步骤5 通过通道和返回的文件描述符处理和交换数据包。

步骤6 当onRevoke () 触发时，关闭文件描述符并关闭通道。

服务中扩展VpnService类需要声明合适的权限和意图过滤器。下面代码是在AndroidManifest.xml中声明一个VPN服务的例子。

<!--声明服务权限-->

<service android:name=".ExampleVpnService"

 android:permission="android.permission.BIND_VPN_SERVICE">

<intent-filter>

<!--添加过滤器-->

<action android:name="android.net.VpnService"/>

</intent-filter>

</service>

10.3 Android封装的HTTP处理类

10.3.1 AndroidHttpClient类和DefaultHttpClient类

AndroidHttpClient类本质上是Apache DefaultHttpClient类在Android上的实现，并且针对Android的网络环境设置了更加合理的默认配置。在应用中不能直接创建AndroidHttpClient类，只能通过newInstance(String)方法创建。

DefaultHttpClient类实现了一个HTTP客户端。Android中包括2个HTTP客户端：HttpURLConnection和Apache HttpClient（Android官方推荐使用HttpURLConnection）。两者都支持HTTPS、数据流的上传与下载、配置超时设定、IPv6和连接池。在Android 2.2（Froyo）及早一些的版本中，Apache HttpClient中漏洞相对少一些。在Android 2.3（Gingerbread）及之后，HttpURLConnection是最好的选择，因为它简单的API及小容量更适合Android平台，透明的压缩和响应缓冲减小了网络需求，改善了速度并减少电耗。

注意有关Apache DefaultHttpClient的知识，感兴趣的读者请参考<http://hc.apache.org/index.html>。

AndroidHttpClient和DefaultHttpClient的区别如下：

■AndroidHttpClient不能在主线程中执行，若在主线程中运行会抛出异常；而DefaultHttpClient是在主线程中运行。

■AndroidHttpClient通过静态方法newInstance获得实例；而DefaultHttpClient通过new关键字创建对象。

■DefaultHttpClient默认是启用Cookie的，而AndroidHttpClient默认不启用Cookie。AndroidHttpClient要使用的话每次执行时要加一个HttpContext参数，并且添加CookieStore。用完后需要调用close方法，这样才能创建新实例。

AndroidHttpClient客户端处理Cookie，但默认情况下不保留Cookie。如果要保留Cookie，需要增加CookieStore到HttpContext中。实现代码如下：

```
context.setAttribute (ClientContext.COOKIE_STORE,cookieStore) ;
```

AndroidHttpClient类创建一个新的、配置了合理默认设置的HttpClient，其定义如下：

```
/*参数userAgent在HTTP请求中；
```

```
参数context用于缓存SSL会话（无缓存时可能为空）；
```

```
返回类型AndroidHttpClient可用于所有的请求*/
```

```
public static AndroidHttpClient newInstance (String userAgent,Context  
context) {  
  
    HttpParams params=new BasicHttpParams () ;  
  
    HttpConnectionParams.setStaleCheckingEnabled (params,false) ;  
  
    HttpConnectionParams.setConnectionTimeout  
(params,SOCKET_OPERATION_TIMEOUT) ;  
  
    HttpConnectionParams.setSoTimeout  
(params,SOCKET_OPERATION_TIMEOUT) ;  
  
    HttpConnectionParams.setSocketBufferSize (params, 8192) ;  
  
    //不处理重定向——返回给调用者。我们往往希望代码在重定向后重新  
    //发布，这个必须自己来做  
  
    HttpClientParams.setRedirecting (params,false) ;  
  
    //对SSL socket使用session缓存  
  
    SSLSessionCache sessionCache=context==null?null:new SSLSessionCache  
(context) ;  
  
    //设置指定的用户代理，注册标准协议
```

```
HttpProtocolParams.setUserAgent (params,userAgent) ;  
  
SchemeRegistry schemeRegistry=new SchemeRegistry () ;  
  
schemeRegistry.register (new Scheme ("http",  
  
PlainSocketFactory.getSocketFactory () , 80) ) ;  
  
schemeRegistry.register (new Scheme ("https",  
  
SSLCertificateSocketFactory.getHttpSocketFactory (  
  
SOCKET_OPERATION_TIMEOUT,sessionCache) , 443) ) ;  
  
ClientConnectionManager manager=  
  
new ThreadSafeClientConnManager (params,schemeRegistry) ;  
  
//要使用工厂方法以修改基类初始化参数，没有直接调用静态方法实现  
的  
  
return new AndroidHttpClient (manager,params) ;  
  
}
```

10.3.2 SSL认证信息处理类

针对Web的网络安全协议，常见的HTTPS链接采用了SSL（SecureSocketsLayer，安全套接层）技术。SSL协议的实现与数字证书密切相关。在android.net.http包中提供了SslCertificate类和SslError类（详见10.3.3节）用以描述X509数字证书信息。在WebView中通过getCertificate（）方法可以查看当前页面是否拥有SSL证书。

SslCertificate类有一个SslCertificate.Dname嵌套类。该嵌套类成员是一个三元组，包含通用名称（common name,CN）、组织（organization,O）和组织单元（organizational unit,OU）。用户可以通过对应的函数获取对应的名字。

- public String getCName（）：返回通用名称（common name,CN）部分。
- public String getDName（）：返回标识名称（通常包括CN、O、OU）。
- public String getOName（）：返回组织（organization,O）部分。
- public String getUName（）：返回组织单元（organizational unit,OU）部分。

SslCertificate的构造方法为SslCertificate（X509Certificate certificate），它创建一个新的X509证书的SSL证书对象。

SslCertificate类中包含的重要方法如下。

- SslCertificate.DName getIssuedBy () : 返回Issued-by标识名或者在没有设置时返回null。
- SslCertificate.DName getIssuedTo () : 返回Issued-to标识名或者在没有设置时返回null。
- Date getValidNotAfterDate () : 返回不在证书有效期之后的日期或者在没有设置时返回""。
- Date getValidNotBeforeDate () : 返回不在证书有效期之前的日期或者在没有设置时返回""。
- static SslCertificate restoreState (Bundle bundle) : 从保存的bundle中恢复证书。
- static Bundle saveState (SslCertificate certificate) : 保存证书状态到bundle中。

10.3.3 SSL错误信息处理

SslError类表示一个或多个SSL错误及相关联的SSL证书的集合，其处理函数主要有以下几个。

- addError (int error) : 将提供的SSL error加入集合。

- `getCertificate ()` : 获得与该对象相关联的SSL证书。
- `getPrimaryError ()` : 获得错误集合中优先级最高的SSL error。
- `getUrl ()` : 获得与该对象相关联的URL。
- `hasError (int error)` : 检查是否该对象包含提供的error。

SslError类中定义错误类型的代码如下：

```
public class SslError{  
  
    //单独的SSL error (按次要到重要的顺序排列)  
  
    //证书尚未生效  
  
    public static final int SSL_NOTYETVALID=0 ;  
  
    //证书过期  
  
    public static final int SSL_EXPIRED=1 ;  
  
    //主机名不匹配  
  
    public static final int SSL_IDMISMATCH=2 ;  
  
    //证书授权不被信任
```

```
public static final int SSL_UNTRUSTED=3 ;  
  
//证书日期无效  
  
public static final int SSL_DATE_INVALID=4 ;  
  
//发生一般性错误  
  
public static final int SSL_INVALID=5 ;  
  
.....  
}
```

10.3.4 AndroidHttpClient

Android中HTTP协议主要体现在android.net.http和org.apache.http等包中。在android.net.http包中，主要通过AndroidHttpClient类来实现HTTP协议，AndroidHttpClient实际上就是实现了org.apache.http.client.HttpClient，它是DefaultHttpClient的子类，通过AndroidHttpClient newInstance (String userAgent, Context context) 方法获得一个实例。其能够处理Cookie，但是在默认情况下无法维护Cookie。设置Cookie的方法如下：

```
context.setAttribute (ClientContext.COOKIE_STORE,cookieStore) ;
```

AndroidHttpClient通常和HttpHost、HttpUriRequest、HttpContext、
ResponseHandler一起发起HTTP请求以及处理服务器响应。下面是一个
简单例子，向服务器发送GET请求。

```
try{
```

```
//获取AndroidHttpClient实例
```

```
AndroidHttpClient client=
```

```
AndroidHttpClient.newInstance ("user_agent_my_mobile_browser") ;
```

```
//创建HttpGet方法，该方法会自动处理URL地址的重定向
```

```
HttpGet httpGet=new HttpGet ("http://www.android.com/") ;
```

```
HttpResponse response=client.execute (httpGet) ;
```

```
if (response.getStatusLine () .getStatusCode () !=  
HttpStatus.SC_OK) {
```

```
//错误处理
```

```
}
```

```
.....
```

```
//关闭连接  
  
client.close () ;  
  
}catch (Exception ee) {  
  
.....  
  
}
```

10.4 Android RTP协议

实时传输协议（Real-time Transport Protocol,RTP）是一个网络传输协议，它是IETF提出的一个标准，对应的RFC文档为RFC 3550。该文档不仅定义了RTP，而且定义了配套的实时传输控制协议（Real-time Transport Control Protocol,RTCP）。RTP用来为IP网上的语音、图像、传真等多种需要实时传输的多媒体数据提供端到端的实时传输服务。RTP为Internet上端到端的实时传输提供时间信息和流同步，但并不保证服务质量。服务质量由RTCP来保证。

RTP协议详细说明了在互联网上传递音频和视频的标准数据包格式。它一开始被设计为一个多播协议，但后来被用在很多单播应用中。RTP协议常用于流媒体系统（配合RTSP协议）、视频会议和Push to Talk系统（配合H.323或SIP），这使它成为了IP电话产业名副其实的基础技术。

RTP用于在单播或多播网络中传送实时数据，其典型的应用场合如下：

- 简单的多播音频会议。语音通信通过一个多播地址和一对端口来实现。分别用于音频数据（RTP）和控制包（RTCP）。
- 音频和视频会议。如果在一次会议中同时使用了音频和视频会议，这两种媒体将分别在不同的RTP会话中传送，每一个会话使用不同的传输地址（IP地址+端口）。如果一个用户同时使用了两个会话，则每个会

话对应的RTCP包都使用规范化名字CNAME（Canonical Name）。与会者可以根据RTCP包中的CNAME来获取相关联的音频和视频，然后根据RTCP包中的计时信息（Network time protocol）来实现音频和视频的同步。

■翻译器和混合器。翻译器和混合器都是RTP级的中继系统。翻译器用在通过IP多播不能直接到达的用户区，例如发送者和接收者之间存在防火墙。若与会者能接收的音频编码格式不一样，比如有一个与会者通过一条低速链路接入高速会议，这时就要使用混合器。在进入音频数据格式需要变化的网络前，混合器将来自一个源或多个源的音频包进行重构，并把重构后的多个音频合并，采用另一种音频编码进行编码后，再转发这个新的RTP包。从一个混合器出来的所有数据包要用混合器作为它们的同步源（SSRC，见RTP的封装）来识别，可以通过贡献源列表（CSRC表，见RTP的封装）可以确认谈话者。

Android中从API level 12开始含有对RTP协议的支持，相关方法主要包含在android.net.rtp包中。Android内置RTP（实时传输协议）协议栈，开发应用程序的时候可以使用它来管理交互数据流。这样应用程序就可以提供VOIP、一键通、会议和音频流，并且在网络可用的前提下，可以使用这些API来启动会话、传输或接收数据。

若要使用RTP API，应用程序必须添加如下权限：

```
<uses-permission android:name="android.permission.INTERNET">
```

此外，需要添加获取如下麦克风的权限：

```
<uses-permission  
    android:name="android.permission.RECORD_AUDIO">
```

RTP支持包位于目录frameworks/base/voip/java/android/net/rtp下，主要包含4个Java类，分别代表基于RTP协议的流RtpStream、基于RTP协议的语音流AudioStream、描述了语音Codec信息的AudioCodec和语音会话组的AudioGroup。

10.4.1 传输音频码

android.net.rtp.AudioCodec类定义了一个与AudioStream一起使用的音频码的集合，它们的参数使用SDP（Session Description Protocol）协议互换信息。大多数列在这里的值都可以在RFC 3551文档中找到，其他的使用单独的标准描述。

很少一部分简单配置被定义为公有的静态实例以方便直接使用。大多数复杂的配置可以调用方法getCodec (int, String, String) 获得。例如，可以使用以下代码创建一个mode-1-only AMR codec。

```
AudioCodec codec=AudioCodec.getCodec (100, "AMR/8000", "mode-set=1") ;
```

一个`android.net.rtp.AudioStream`需要有一个`android.net.rtp.AudioCodec`为其编解码。Java层的`AudioCodec`只是描述了Codec信息的类，主要包含了以下3类信息：

`public final int type ; //编码的RTP类型`

`public final String rtpmap ; //用于相应的SDP属性的编码参数`

`public final String fmtp ; //用于相应的SDP属性的格式化参数`

下面列举了`AudioCodec`中定义的类型。

- `public static final AudioCodec GSM`:GSM全速率音频编码，又称GSM-FR、GSM 06.10、GSM，或者简称FR。
- `public static final AudioCodec GSM_EFR`：增强的GSM全速率音频编码，又称GSM-EFR、GSM 06.60，或者简称EFR。
- `public static final AudioCodec PCMA`：采用a-law压缩算法的G.711音频编码。
- `public static final AudioCodec PCMU`：采用u-law压缩算法的G.711音频编码。

10.4.2 AudioGroup

一个语音组AudioGroup是可以包括扬声器、麦克风及其他音频流的一个音频中心。这些组件中的每一个逻辑上都可以通过调用setMode (int) 或者setMode (int) 进行打开或关闭。AudioGroup将执行循环逐一检查这些组件，具体包括以下4个步骤：

步骤1 对每个非MODE_SEND_ONLY模式下接收的音频流进行解码，并存储在缓冲区中。

步骤2 如果麦克风被启用，则处理录音并存储在缓冲区。

步骤3 如果扬声器被启用，混合所有的音频流缓冲区并播放。

步骤4 对每个非MODE_SEND_ONLY模式下的音频流，混合所有其他缓冲区并发送编码数据包。如果没有任何音频流，AudioGroup就什么也不做。

在使用这些类时有一些事情必须注意。系统性能、系统负载均与网络带宽密切相关。通常一个简单的编解码器占用很少的CPU周期，但却需要较多的网络带宽。同时使用两个音频流会同时使负载和带宽的需求都加倍。从一个设备转换到另一个设备，条件是变化的，开发者应选择正确的组合，以达到最佳的效果。

有时候同时保持多个AudioGroup也是有用的。例如，一个VoIP应用可能想保持会议电话以便建立新会话，但又想允许开会的人之间相互通

话。这种情况使用两个AudioGroup很容易实现，但存在一定的局限性。由于扬声器和麦克风是全球共享资源，在一段时间内只有一个AudioGroup允许运行在MODE_ON_HOLD模式下，其他的都将因无法获得这些资源而失败。

注意 使用AudioGroup类需要RECORD_AUDIO权限。开发者应该使用setMode (int) 方法设置音频模式为MODE_IN_COMMUNICATION，并且当没有AudioGroup在使用时要变回原来的设置。

android.net.rtp.AudioGroup代表的是一个会话，可能只是两人通话，也可能是多于两人的电话会议。虽然android.net.rtp.AudioGroup允许同时有多组会话，但因为麦克风和扬声器只能是排他性地使用，故只能有一组会话为活动的，其他的必须是HOLD状态。

语音组通过以下映射表来维护加入它里面的语音流。

```
private final Map<AudioStream,Integer>mStreams ;
```

一个AudioStream加入AudioGroup的流程如下：首先是AudioStream调用join加入到某个AudioGroup中。

```
public void join (AudioGroup group)
```

然后调用AudioGroup.add；接着调用下面的native Add本地方法：

```
private native void nativeAdd (int mode,int socket,String remoteAddress,int  
remotePort,String codecSpec,int dtmfType) ;
```

其中前4个参数：mode、socket号、远程地址、远程端口号，来自于语音流的父类RTPStream中的信息，codecSpec来自于语音流对应的Codec信息，最后一个参数dtmf也来自语音流。

10.4.3 语音流RtpStream和AudioStream

语音流android.net.rtp.AudioStream继承自RtpStream，代表着一个建立在 RTP协议之上的与对方通话的语音流。语音流需要使用一个语音Codec 来描述其对应的编解码信息。在建立通话之前，语音流需加入（join）会话组android.net.rtp.AudioGroup中。因此，它包含了所在的语音组、语音Codec以及DTMF（Dual-Tone Multi-Frequency）类型（RFC 2833）等信息。

RTP流RtpStream类是基于RTP协议的数据流。Java层的API类是 android.net.rtp.RtpStream，代表着一个通过RTP协议发送和接收网络多媒体数据包的流。一个流主要包括本机网络地址和端口号、远程主机网络地址和端口号、socket号和流模式。

RtpStream支持3种流模式，可由setMode函数设定。setMode可取3个值，分别代表了以下3种流模式。

■MODE_NORMAL：正常模式，接收和发送数据包。

■MODE_SEND_ONLY：只发送数据包。

■MODE_RECEIVE_ONLY：只接收数据包。

本地主机IP地址（InetAddress，支持IPv4和IPv6）由调用构造函数时传入。在构造函数中，会调用native层实现的create函数获取一个本地主机端口号（依据RFC 3550）。同时，native层的create函数还会得到一个socket连接号，socket号会在native层中更新到该Java类实例中。

函数associate指定远程主机地址和端口号。

```
public void associate (InetAddress address,int port)
```

下面介绍如何获取socket号。

android.net.rtp.RtpStream的一个私有成员整型变量mNative存放Socket号。

```
private int mNative ;
```

由JNI层在调用socket函数后得到一个socket号，存入mNative变量，即在JNI层（frameworks/base/voip/jni/rtp/RtpStream.cpp）中定义标识该socket号的变量。

```
jfieldID gNative ;
```

在函数registerRtpStream中获取具体的值。

```
(gNative=env->GetFieldID (clazz, "mNative", "I") ) ==NULL
```

在JNI层的create函数中，调用socket函数得到一个socket号。

```
int socket= : socket (ss.ss_family,SOCK_DGRAM, 0) ;
```

这个socket号指定给Java层的mNative变量。

```
env->SetIntField (thiz,gNative,socket) ;
```

端口号port则由create函数直接返回，create函数已经支持IPv6。

10.5 Android SIP协议

SIP（Session Initiation Protocol，会话发起协议）是用来帮助提供跨越Internet的高级电话业务。Internet电话（IP电话）正在向一种正式的商业电话模式演进，SIP就是用来确保这种演进的实现而需要的NGN（下一代网络）系列协议中重要的一员。

SIP在Android上已经有多种解决方案，除了Android本身的SIP接口之外，还有一些第三方的解决方案，其中比较成熟的第三方开源方案为Sipdroid。

注意 Sipdroid官方网址为<http://code.google.com/p/sipdroid/>，有兴趣的读者可以参考其方案。

Android官方的SIP处理接口在android.net.sip包中，源代码在frameworks/base/core/java/android/net/sip目录下。

10.5.1 SIP通话简介

当需要创建会话时，使用sipManager.makeAudioCall（String localProfileURI, String peerProfileURI, SipAudioCall.Listener, int timeout）方法来创建一个SipAudioCall，其中timeout单位为秒，定义了打电话超

时时间。需要打给别人时使用makeAudioCall创建SipAudioCall，接听电话用takeAudioCall来创建SipAudioCall。

SipAudioCall中有一个嵌套的类SipAudioCall.Listener，此类主要用于在呼叫电话和接听电话时监听SIP。

以下为makeAudioCall的示例代码：

```
SipAudioCall.Listener listener=new SipAudioCall.Listener () {  
  
    @Override  
  
    //呼叫建立  
  
    public void onCallEstablished (SipAudioCall call) {  
  
        call.startAudio () ;//启动音频  
  
        call.setSpeakerMode (true) ;//设置为可讲话模式  
  
        call.toggleMute () ;//触发无声  
  
        updateStatus (call) ;  
  
    }  
  
} ;
```

```
SipAudioCall call=manager.makeAudioCall (me.getUriString () ,  
sipAddress,listener, 30) ;
```

SIP通话涉及的相关类均在android.net.sip包中，下面列出了该包中的主要类及其作用。

- SipAudioCall：通过SIP处理网络音频电话。
- SipAudioCall.Listener：关于SIP电话的事件监听器，比如接到一个电话（on ringing）或者呼出一个电话（on calling）的时候。
- SipErrorCode：定义在SIP活动中返回的错误代码。
- SipManager：为SIP任务提供APIs，比如初始化一个SIP连接，提供相关SIP服务的访问等。
- SipProfile：定义了SIP的相关属性，包含SIP账户、域名和服务器信息。
- SipProfile.Builder：创建SipProfile的辅助类。
- SipSession：代表一个SIP会话，跟SIP对话框或者一个没有对话框的独立事务相关联。
- SipSession.Listener：关于SIP会话的事件监听器，比如注册一个会话（on registering）或者呼出一个电话（on calling）的时候。

■SipSession.State：定义SIP会话的声明，比如“注册”、“呼出电话”、“接到电话”。

■SipRegistrationListener：一个关于SIP注册事件监听器的接口。

10.5.2 SIP初始化

SipManager类负责管理SIP通话的过程，包括启动SIP会话、启动并接受呼叫、注册和注销SIP服务提供商、验证会话的连通性等。SipManager类提供了SIP工作的API函数，用于初始化SIP连接、提供相关SIP服务等。这个类是任何SIP动作的起始点，可以使用newInstance（）获得它的实例。

```
public SipManager mSipManager=null ;  
  
if （mSipManager==null） {  
  
mSipManager=SipManager.newInstance （this） ；  
  
}
```

SipManager类可以创建一个SipSession，以便打出电话或者接听电话。SipManager类还可以初始化和接收泛化的SIP对话或者仅含语音的SIP对话。泛化的SIP对话可能包含视频、声音或者其他，使用open（）方法初始化。仅含语音的SIP对话应该使用SipAudioCall来处理，

SipAudioCall可以通过makeAudioCall () 和takeAudioCall () 方法获得。

注意 不是所有Android设备都支持使用SIP协议的VOIP对话。应该经常调用isVoipSupported () 方法去验证设备是否支持VOIP对话，调用isApiSupported () 方法去验证设备是否支持SIP APIs。应用必须拥有INTERNET及USE_SIP权限。

SipManager提供的API如下。

- public static boolean isApiSupported (Context context) :用于判断系统是否支持SIP API。
- public static boolean isVoipSupported (Context context) :用于判断系统是否支持基于SIP协议的VOIP API。
- public static boolean isSipWifiOnly (Context context) :用于判断SIP是否仅适用于Wi-Fi。
- public SipAudioCall makeAudioCall (SipProfile localProfile,SipProfile peerProfile,SipAudioCall.Listener listener,int timeout) :创建一个拨打电
- 话。
- public SipAudioCall takeAudioCall (Intent incomingCallIntent,SipAudioCall.Listener listener) :创建一个接听来

电。

- public static boolean isIncomingCallIntent (Intent intent) : 检查是否是呼入广播意图。
- public static String getOfferSessionDescription (Intent incomingCallIntent) : 从指定的来电广播意图获取提供的会话描述。
- public void register (SipProfile localProfile,int expiryTime,SipRegistrationListener listener) : 手动注册的配置文件，相应的SIP提供商接听电话。
- public void unregister (SipProfile localProfile,SipRegistrationListener listener) : 注销相应的SIP。

10.5.3 SipProfile

SipProfile类定义了一个SIP的概况，包括SIP账号、域及服务器等信息。可以使用SipProfile.Builder创建一个SipProfile，也可以从一个SipSession中调用getLocalProfile () 和getPeerProfile () 方法得到。示例如下：

```
SipProfile sipProfile=SipProfile.Builder.build () ;
```

```
//返回一个SipProfile object
```

与该类相关的主要方法如下。

- int describeContents () : 代表了这种Parcelable接口所描述的特殊对象。
- String getAuthUserName () : 获取授权的用户名。
- boolean getAutoRegistration () : 获取自动注册的标志。
- String getDisplayName () : 获得用户显示的名字。
- String getPassword () : 获取密码。
- int getPort () : 获取SIP服务器的端口号。
- String getProfileName () : 获得用户定义的profile名字。
- String getProtocol () : 获得用于连接服务器的协议。
- String getProxyAddress () : 获得服务器境外代理的网络地址。
- boolean getSendKeepAlive () : 获取Sending keep-alive的标志。
- String getSipDomain () : 获取SIP域。
- String getUriString () : 获取该profile的SIP URI字符串。
- String getUserName () : 获取用户名。

■void writeToParcel (Parcel out,int flags) : 将该对象写入Parcel。

SipProfile.Builder类用于辅助构造一个SIP配置文件，其相关方法如下。

■SipProfile build () : 创建及返回SIP profile对象。

■SipProfile.Builder setAuthUserName (String name) : 设置授权的用户名。

■SipProfile.Builder setAutoRegistration (boolean flag) : 设置自动注册。

■SipProfile.Builder setDisplayName (String displayName) : 设置显示的用户名。

■SipProfile.Builder setOutboundProxy (String outboundProxy) : 设置SIP服务器的外网代理。

■SipProfile.Builder setPassword (String password) : 设置SIP账户的密码。

■SipProfile.Builder setPort (int port) : 设置服务器的端口号。

■SipProfile.Builder setProfileName (String name) : 设置profile的名称。

- SipProfile.Builder setProtocol (String protocol) : 设置用于连接SIP服务器的协议。
- SipProfile.Builder setSendKeepAlive (boolean flag) : 设置send keep-alive标志。

10.5.4 SipSession

SipSession类代表了SIP会话信息。可以通过SipManager的createSipSession () 方法（在初始化对话时）或者getSessionFor () 方法（在接收对话过程中）获得一个SipSession。该类中定义了一个状态类State（如“注册”和“呼出”），代码如下：

//当会话已准备好发起呼叫或事务时

```
public static final int READY_TO_CALL=0 ;
```

//当注册请求被发送出去时

```
public static final int REGISTERING=1 ;
```

//当注销请求被发送出去时

```
public static final int Deregistering=2 ;
```

//当INVITE请求被接受时

```
public static final int INCOMING_CALL=3 ;  
  
//当作为INVITE请求返回的OK响应被接受时  
  
public static final int INCOMING_CALL_ANSWERING=4 ;  
  
//当INVITE请求被发送时  
  
public static final int OUTGOING_CALL=5 ;  
  
//当作为INVITE请求返回的RINGING响应被发送时  
  
public static final int OUTGOING_CALL_RING_BACK=6 ;  
  
//当INVITE请求发送一个CANCEL请求时  
  
public static final int OUTGOING_CALL_CANCELING=7 ;  
  
//当一个呼叫建立时  
  
public static final int IN_CALL=8 ;  
  
//当一个OPTIONS请求被发送时  
  
public static final int PINGING=9 ;  
  
//当结束一个呼叫时
```

```
public static final int ENDING_CALL=10 ;
```

//没有定义时

```
public static final int NOT_DEFINED=101 ;
```

SipSession类还定义了一个监听类Listener。SipSession.Listener中和IP会话有关的事件如下。

- public void onCalling (SipSession session) : 正在会话中调用。
- public void onRinging (SipSession session,SipProfile caller,String sessionDescription) : 收到INVITE请求时调用。
- public void onRingingBack (SipSession session) : RINGING收到响应发送的INVITE请求时调用。
- public void onCallEstablished (SipSession session,String sessionDescription) : 在会话建立时调用。
- public void onCallEnded (SipSession session) : 在会话终止时调用。
- public void onCallBusy (SipSession session) : 通话忙时调用。
- public void onCallTransferring (SipSession newSession,String sessionDescription) : 呼叫转移时调用。

- public void onError (SipSession session,int errorCode,String errorMessage) :会话初始化和终止过程中发生错误时调用。
- public void onCallChangeFailed (SipSession session,int errorCode,String errorMessage) :更改会话协商过程中发生错误时调用。
- public void onRegistering (SipSession session) :发送注册请求时调用。
- public void onRegistrationDone (SipSession session,int duration) :注册成功完成时调用。
- public void onRegistrationFailed (SipSession session,int errorCode,String errorMessage) :注册失败时调用。
- public void onRegistrationTimeout (SipSession session) :注册超时调用。

10.5.5 SIP包错误处理

SipErrorCode类定义了一些SIP行为（如调用onRegistrationFailed () 、onError () 、onCallChangeFailed () 或onRegistrationFailed () ）返回的错误代码。

源码中相关错误码定义如下：

```
public class SipErrorCode{  
  
    //没有错误  
  
    public static final int NO_ERROR=0 ;  
  
    //当一些socket错误发生时  
  
    public static final int SOCKET_ERROR=-1 ;  
  
    //当服务器响应错误时  
  
    public static final int SERVER_ERROR=-2 ;  
  
    //当事务意外终止时  
  
    public static final int TRANSACTION_TERMINATED=-3 ;  
  
    //当设备上一些错误发生时，可能的情况是因为一个bug  
  
    public static final int CLIENT_ERROR=-4 ;  
  
    //当事务超时时  
  
    public static final int TIME_OUT=-5 ;  
  
    //当远程URI无效时
```

```
public static final int INVALID_REMOTE_URI=-6 ;
```

//当peer无法到达时

```
public static final int PEER_NOT_REACHABLE=-7 ;
```

//当提供的是无效的证书时

```
public static final int INVALID_CREDENTIALS=-8 ;
```

//客户端已在处理事务而无法发起一个新事务时

```
public static final int IN_PROGRESS=-9 ;
```

//当数据连接丢失时

```
public static final int DATA_CONNECTION_LOST=-10 ;
```

//当需要跨域授权时

```
public static final int CROSS_DOMAIN_AUTHENTICATION=-11 ;
```

//当服务器不可达时

```
public static final int SERVER_UNREACHABLE=-12 ;
```

.....

}

10.6 小结

本章介绍的Android网络处理分析主要涉及的是android.net包，本章首先给出了该包下的关键类及其简要说明。随后，重点分析了Android网络处理的流程，包括网络状态监控、网络认证、DHCP处理、网络代理等相关内容。接着，在Android封装的HTTP处理类一节中，主要介绍了AndroidHttpClient及SSL认证。最后分析了与RTP协议和SIP协议相关的源码。

**本书由“[ePUBw.COM](#)”整理，[ePUBw.COM](#) 提供
最新最全的优质电子书下载！！！**

第 11 章 Android 网络应用分析

本章对Android中几种网络应用源码进行分析，主要讲解Android如何使用SAX解析XML，在Android上如何实现基于位置的服务，以及媒体传输协议的相关内容。

11.1 Android中使用SAX解析XML

在第4章我们曾提到，在Android编程中，XML文档解析最常用的方式有DOM、SAX、PULL三种，并通过实例具体讲述了各种方式的解析方法。本节先对几种XML解析方式进行讨论，然后重点分析SAX解析XML的原理。

11.1.1 几种XML解析方式讨论

DOM、SAX、PULL是现在最常用的3种解析XML文档的方式。这3种方式有什么不同呢？各有什么缺点呢？本小节就和读者一起简单讨论一下。

1.DOM方式解析XML

DOM是用与平台和语言无关的方式解析XML文档的官方W3C标准。这种方式将XML文件的所有内容以文档树的方式存放在内存中，然后允许开发人员使用DOM API遍历XML树，在树中寻找特定信息。但这种基于树的层级结构通常需要加载整个文档，因此有其固有的优缺点。

■由于树在内存中是持久的，因此可以修改它以便应用程序能对数据和结构做出更改。它还可以在任何时候在树中进行上下导航，而不是像SAX那样只是一次性的处理。因而DOM方式使用起来简单直观。

■对于特别大的文档，解析和加载整个文档可能很慢且很耗资源，此时使用其他解析方式（如SAX等）效果会更好。

2.SAX方式解析XML

SAX是基于事件驱动的，这种解析方式是边加载边解析的。这种解析方式的优点类似于流媒体，即使用这种方式对XML文档的解析能够立即开始，而无需等待所有的数据被处理。而且，由于应用程序只是在读取数据时检查数据，因此不需要将数据存储在内存中。因而这种方式特别适合大型文档的解析。事实上，应用程序甚至不必解析整个文档，而在某个条件得到满足时停止解析。

3.PULL方式解析XML

PULL解析方式和SAX解析方式很相似，都是轻量级的解析方式。由于在Android的内核中已经嵌入了PULL解析器，因此不需要再添加第三方Jar包来支持PULL。

PULL解析器小巧轻便，解析速度快，简单易用，非常适合在Android移动设备中使用。Android系统内部解析各种XML文档时使用的也是PULL解析器，Android官方也推荐开发者们使用PULL方式解析XML文档。

11.1.2 SAX解析XML的原理

SAX解析XML文档中的数据是以流的方式来分析，处理过程中每次都读入部分数据进行处理，处理完后再读入后面的数据，SAX处理方法可以有效地节省内存。SAX从机制上并不会对XML数据进行任何保存，在其读入下一部分数据的同时，前面读入的数据会被自动删除。

SAX的工作原理是对文档进行由上到下的顺序扫描，当扫描到文档开始、XML元素开始、XML元素结束、文档结束等地方时通知事件处理函数，由事件处理函数做相应的处理动作。这些事件处理函数在Android中已经定义了，各个函数中相应的动作处理都可以扩展，这样就可以方便地处理XML中每一个元素。

SAX解析XML的步骤如下：

while（文档没有结束） {

 读入一行XML文档；

 If（文件开始、文档开始、元素开始、元素结束、文档结束、文件结束等标志） {

 触发相应的处理函数；

 编写代码处理；

 执行characters方法；

}

}

android.sax包中提供了相关的类与方法，方便了大家编写高效的、健壮的SAX处理程序。SAX采用的是事件驱动，不需要完整解析整个文档，而是按照内容顺序看文档某个部分是否符合XML语法，如果符合就触发相应的事件。这里所谓的事件就是一些回调方法，这些方法定义在ContentHandler中。下面列出了其中的主要方法。

- startDocument () : 用于处理文档解析开始事件，可以在其中做些预处理工作，如申请对象资源等。
- endDocument () : 用于处理文档解析的结束事件，可以在其中做些后继工作，如释放申请的对象资源等。
- startElement (String namespaceURI, String localName, String qName, Attributes attrs) : 处理元素开始事件。
- endElement (String uri, String localName, String name) : 处理元素结束事件。
- Charachers (char[] ch,int start,int length) : 处理元素的字符内容事件，第一个参数为文件的字符串内容，后面两个参数是读到的字符串在这

个数组中的起始位置和长度，使用new String (ch,start,length) 就可以获取内容。

包含的监听事件如下。

■ElementListener：监听元素的开始和结束。

■EndElementListener：监听元素的结束。

■EndTextElementListener：监听文本元素的结束。

■StartElementListener：监听元素的开始。

■TextElementListener：监听文本元素的开始和结束。

11.1.3 SAX发现XML的根元素

XML文档是一种树结构，从“树根”开始定义，然后定义到“树枝”。

XML在定义上都使用简单的具有自我描述性的词汇，比如在下面的例子里面，第一行描述了文档的根元素，feed也是这个XML文档的名称，xmlns为该XML文档的命名空间。entry为feed的子元素，id又为entry的子元素。

在Android平台下，RootElement类代表XML的根元素，是SAX解析的入口，通过RootElement方法可以获取根元素，通过RootElement.getChild() 方法获取子元素。例如，有以下XML文本：

```
<feed xmlns='http://www.w3.org/2005/Atom'>

<entry>

<id>bob</id>

</entry>

</feed>
```

使用SAX方式解析上面的一段XML文档，获取其中的entry id的信息，其实现的关键代码如下：

```
static final String  
ATOM_NAMESPACE="http://www.w3.org/2005/Atom" ;  
  
.....  
  
//获取根元素  
  
RootElement root=new RootElement (ATOM_NAMESPACE, "feed") ;  
  
//获取子元素  
  
Element entry=root.getChild (ATOM_NAMESPACE, "entry") ;  
  
//获取id号
```

```
entry.getChild  
        (ATOM_NAMESPACE, "id") .setEndTextElementListener (   
  
new EndTextElementListener () {  
  
    public void end (String body) {  
  
        System.out.println ("Entry ID : "+body) ;  
  
    }  
  
}) ;
```

.....

//设置处理内容

```
reader.setContentHandler (root.getContentHandler () ) ;
```

//读取解析到的内容

```
reader.parse (.....) ;
```

上一段代码输出内容如下：

Entry ID:bob

下面是RootElement类在Android源码中的实现：

```
public class RootElement extends Element{  
    final Handler handler=new Handler () ;  
  
    //构造一个特定名称的根元素  
  
    public RootElement (String uri,String localName) {  
        super (null,uri,localName, 0) ;  
  
    }  
  
    //使用一个空的字符串作为命名空间，构造一个新的特定名称的根元素  
  
    public RootElement (String localName) {  
        this ("", localName) ;  
  
    }  
  
    //得到SAX的ContentHandler，并传递给SAX解析器  
  
    public ContentHandler getContentHandler () {  
        return this.handler ;  
  
    }  
}
```

```
//扩展默认处理类
```

```
class Handler extends DefaultHandler{
```

```
    Locator locator ;
```

```
    int depth=-1 ;
```

```
    Element current=null ;
```

```
    StringBuilder bodyBuilder=null ;
```

```
    @Override
```

```
    public void setDocumentLocator (Locator locator) {
```

```
        this.locator=locator ;
```

```
}
```

```
    @Override
```

```
    public void startElement (String uri,String localName,String qName,
```

```
        Attributes attributes) throws SAXException{
```

```
        int depth=++this.depth ;
```

```
if (depth==0) {  
  
    //根元素  
  
    startRoot (uri,localName,attributes) ;  
  
    return ;  
  
}  
  
//禁止文本与元素混合使用  
  
if (bodyBuilder !=null) {  
  
    throw new BadXmlException ("Encountered  
mixed content"+"within text element named"+  
current+".", locator) ;  
  
}  
  
//如果有下一层，还要解析当前元素下一层  
  
if (depth==current.depth+1) {  
  
    //找到子元素并压入堆栈  
}
```

```
Children children=current.children ;  
  
if (children !=null) {  
  
Element child=children.get (uri,  
localName) ;  
  
if (child !=null) {  
  
start (child,attributes) ;  
  
}  
  
}  
  
}  
  
}  
  
}  
  
//开始处理根元素  
  
void startRoot (String uri,String localName,Attributes attributes)  
throws SAXException{  
  
Element root=RootElement.this ;
```

```
if (root.uri.compareTo (uri) !=0||root.localName.compareTo
(localName) !=0) {

throw new BadXmlException ("Root element name does"+ "not
match.Expected : "+root+", Got : "+Element.toString
(uri,localName) , locator) ;

}

start (root,attributes) ;

}

void start (Element e,Attributes attributes) {

//将元素压入堆栈

this.current=e ;

if (e.startElementListener !=null) {

e.startElementListener.start (attributes) ;

}

if (e.endTextElementListener !=null) {
```

```
this.bodyBuilder=new StringBuilder () ;  
}  
  
e.resetRequiredChildren () ;  
  
e.visited=true ;  
  
}  
  
@Override  
  
public void characters (char[]buffer,int start,int length)  
throws SAXException{  
  
if (bodyBuilder !=null) {  
  
bodyBuilder.append (buffer,start,length) ;  
  
}  
  
}  
  
@Override  
  
public void endElement (String uri,String localName,String qName)
```

```
throws SAXException{

Element current=this.current ;

//检测是否到了当前层

if (depth==current.depth) {

current.checkRequiredChildren (locator) ;

//调用元素结束监听器

if (current.endElementListener !=null) {

current.endElementListener.end () ;

}

//调用文档结束监听器

if (bodyBuilder !=null) {

String body=bodyBuilder.toString () ;

bodyBuilder=null ;

//假设该监听器存在
```

```
        current.endTextElementListener.end (body) ;  
  
    }  
  
    //将元素弹出堆栈  
  
    this.current=current.parent ;  
  
}  
  
depth-- ;  
  
}  
  
}  
  
}
```

11.1.4 SAX发现XML的子元素

从上面的RootElement类的定义中可以看到，RootElement类是继承自Element类的，这符合一个逻辑，根元素本身也可以是子元素，根元素只是特殊的子元素。子元素更加侧重于其中存储的信息和指示的内容，是XML文档存储中的重要组成部分。Element类定义了一个XML子元素，并提供了访问子元素的方法，还可监听与这个元素相关的事情。下面是Element类的定义：

```
public class Element{  
  
    final String uri ;  
  
    final String localName ;  
  
    final int depth ;  
  
    final Element parent ;  
  
    Children children ;  
  
    ArrayList<Element> requiredChildren ;  
  
    boolean visited ;  
  
    StartElementListener startElementListener ;  
  
    EndElementListener endElementListener ;  
  
    EndTextElementListener endTextElementListener ;  
  
    Element (Element parent,String uri,String localName,int depth) {  
  
        this.parent=parent ;  
  
        this.uri=uri ;
```

```
this.localName=localName ;  
  
this.depth=depth ;  
  
}
```

//获取特定名称的子元素。使用一个空字符串作为命名空间

```
public Element getChild (String localName) {  
  
return getChild ("", localName) ;  
  
}
```

//获取特定名称的子元素

```
public Element getChild (String uri,String localName) {  
  
if (endTextElementListener !=null) {  
  
throw new IllegalStateException ("This element already has an end"  
+"text element listener.It cannot have children.") ;  
  
}  
  
if (children==null) {
```

```
children=new Children () ;  
}  
  
return children.getOrCreate (this,uri,localName) ;  
}
```

/*获取特定名称的子元素。使用一个空字符串作为命名空间。

如果指定的子元素在解析时丢失，将抛出org.xml.sax.SAXException异常。这能保证监听器被调用*/

```
public Element requireChild (String localName) {  
  
return requireChild ("", localName) ;  
}
```

/*获取特定名称的子元素。

如果指定的子元素在解析时丢失，将抛出org.xml.sax.SAXException异常。

这能保证监听器被调用*/

```
public Element requireChild (String uri,String localName) {
```

```
Element child= getChild (uri,localName) ;  
  
if (requiredChilden==null) {  
  
requiredChilden=new ArrayList<Element> () ;  
  
requiredChilden.add (child) ;  
  
}  
  
}else{  
  
if ( ! requiredChilden.contains (child) ) {  
  
requiredChilden.add (child) ;  
  
}  
  
}  
  
return child ;  
  
}  
  
//同时设置元素开始和结束监听器  
  
public void setElementListener (ElementListener elementListener) {  
  
setStartElementListener (elementListener) ;
```

```
        setEndElementListener (elementListener) ;  
  
    }  
  
    //同时设置文档开始和结束监听器  
  
    public void setTextElementListener (TextElementListener  
        elementListener) {  
  
        setStartElementListener (elementListener) ;  
  
        setEndTextElementListener (elementListener) ;  
  
    }  
  
    //设置该元素开始的监听器  
  
    public void setStartElementListener (StartElementListener  
        startElementListener) {  
  
        if (this.startElementListener !=null) {  
  
            throw new IllegalStateException (  
                "Start element listener has already been set." ) ;  
  
        }  
    }
```

```
this.startElementListener=startElementListener ;  
}  
  
//设置该元素结束的监听器  
  
public void setEndElementListener (EndElementListener  
endElementListener) {  
  
if (this.endElementListener !=null) {  
  
throw new IllegalStateException ('  
"End element listener has already been set."') ;  
  
}  
  
this.endElementListener=endElementListener ;  
}  
  
//设置该文档结束的监听器  
  
public void setEndTextElementListener (EndTextElementListener  
endTextElementListener) {  
  
if (this.endTextElementListener !=null) {
```

```
        throw new IllegalStateException ( "End text element listener has already been set." ) ;  
    }  
  
    if (children !=null) {  
  
        throw new IllegalStateException ( "This element already has  
children."+"It cannot have an end text element listener." ) ;  
    }  
  
    this.endTextElementListener=endTextElementListener ;  
  
}  
  
@Override  
  
public String toString () {  
  
    return toString (uri,localName) ;  
}  
  
}  
  
static String toString (String uri,String localName) {
```

```
return"""+ (uri.equals ("") ?localName:uri+" : "+localName) +""";
```

```
}
```

```
void resetRequiredChildren () {
```

```
ArrayList<Element>requiredChildren=this.requiredChildren ;
```

```
if (requiredChildren !=null) {
```

```
for (int i=requiredChildren.size () -1 ;i>=0 ;i--) {
```

```
requiredChildren.get (i) .visited=false ;
```

```
}
```

```
}
```

```
}
```

```
//如果指定的子元素不存在则抛出异常
```

```
void checkRequiredChildren (Locator locator) throws
```

```
SAXParseException{
```

```
ArrayList<Element>requiredChildren=this.requiredChildren ;
```

```
if (requiredChildren !=null) {
```

```
for (int i=requiredChildren.size () -1 ; i>=0 ; i--) {  
  
Element child=requiredChildren.get (i) ;  
  
if ( ! child.visited) {  
  
throw new BadXmlException (  
"Element named"+this+"is missing required"  
+"  
+ "child element named"  
+child+".", locator) ;  
  
}  
  
}  
  
}  
  
}
```

11.2 基于位置的服务

11.2.1 位置服务的基本概念

通过android.location包中提供的类库，Android应用程序可以使用地理位置的相关服务。其中最重要的是LocationManager类，它提供API定位设备所在的位置。LocationManager是个系统服务，不能直接实例化一个LocationManager对象，需要调用方法getSystemService(Context.LOCATION_SERVICE) 来实例化。当应用程序创建了LocationManager对象后，就可以做以下3件事：

- 查询所有的LocationProviders的列表，得到最近已知位置坐标。
- 注册或注销一个位置提供商，以便周期性地更新用户的当前位置。
- 如果设备是在一个给定纬度、经度及距离（指定半径，单位米）范围内，可以注册或者注销一个特定的Intent。

11.2.2 位置服务的基本类

android.location包中提供了一些定义Android基于位置及相关服务的类。主要包括LocationManager类、Location类LocationProvider类和Criteria类，其相关的作用如下：

1) LocationManager类是提供访问系统定位服务的类。这些服务允许应用程序获取该设备地理位置的定期更新，或者在设备进入一个特定的地理位置附近时，对特定应用程序的意图报警。不能直接实例化这个类，而应通过调用Context.getSystemService(Context.LOCATION_SERVICE) 来实例化。

除非特别注明，所有与定位相关的API方法都需要在manifest中设置ACCESS_COARSE_LOCATION或ACCESS_FINE_LOCATION权限。如果应用程序只有ACCESS_COARSE_LOCATION权限，则无法进行GPS定位或无源定位。其他情况仍然会返回定位结果，但更新率将节流，同时准确定位的精度也会降到较低的水平。

ACCESS_FINE_LOCATION权限可以进行更加精确的GPS定位，而且包含了ACCESS_COARSE_LOCATION权限。另外，如果应用中使用了基于网络的定位，则还要声明网络权限。这些权限的声明如下：

```
<uses-permission  
    android:name="android.permission.ACCESS_COARSE_LOCATION"/>  
  
<uses-permission  
    android:name="android.permission.ACCESS_FINE_LOCATION"/>  
  
<uses-permission android:name="android.permission.INTERNET"/>
```

可以通过调用getSystemService () 方法获得一个LocationManager引用。

```
LocationManager locationManager= (LocationManager)  
this.getSystemService (Context.LOCATION_SERVICE) ;
```

2) Location类是代表一个地理位置的数据类，包括纬度、经度、时间戳和其他信息（如方位、高度和速度）。

LocationProvider类是代表位置提供者的一个抽象的超类。位置提供者有周期性报告Android设备地理位置的功能。位置提供者定位的时候会使用不同的定位方式，有的需要GPS硬件支持，有的需要使用蜂窝无线通信，有的通过访问特定运营商的网络，还有的需要Internet网络等。其中一些位置提供者可以在Android设置界面中关闭，比如GPS，编写程序时需要先验证一下目标Location Provider是否使能，这可以通过调用isProviderEnabled () 方法实现。

如果Location Provider是不可用的状态，需要请求用户在设置中打开它，调用一个action为ACTION_LOCATION_SOURCE_SETTINGS的Intent来实现。

```
@Override
```

```
protected void onStart () {
```

```
super.onStart () ;  
  
//获取系统位置服务  
  
LocationManager locationManager=  
    (LocationManager) getSystemService  
    (Context.LOCATION_SERVICE) ;  
  
//检测location provider是否使能  
  
final boolean gpsEnabled=  
    locationManager.isProviderEnabled  
    (LocationManager.GPS_PROVIDER) ;  
  
if ( ! gpsEnabled) {  
  
    //在这里建立一个警告对话框，要求用户使能位置服务，用户单击“确定”按钮  
  
    enableLocationSettings ()  
  
}  
  
}
```

```
//调用设置界面，使用户完成设定
```

```
private void enableLocationSettings () {  
  
Intent settingsIntent=new Intent  
(Settings.ACTION_LOCATION_SOURCE_SETTINGS) ;  
  
startActivity (settingsIntent) ;  
  
}
```

3) Criteria类用于设定规则以获取最合适的位置提供者。可设定的规则包括精度、功耗、成本等，还可报告高度、速度、方位。

此外，其他相关类如下。

- Address：一个代表地址的类，比如，描述一个位置的一组字符串。
- Geocoder：实现地址（或位置）与经纬度的相互转换（称为地理编码）。
- GpsSatellite：该类表示一个GPS卫星当前状态。
- GpsStatus：该类表示GPS引擎当前的状态。

提供的事件监听如下。

- GpsStatus.Listener：在GPS状态发生改变时接收通知。
- GpsStatus.NmeaListener：用于从GPS接收NMEA信息。
- LocationListener：提供定位信息发生改变时的回调功能，用于当位置发生改变时从LocationManager接收通知。

11.2.3 调用Google地图

在本书第5章我们已经了解到，Android SDK提供了相关的API，可帮助应用程序开发人员显示和操作地图，获取实时设备位置信息。地图API和基于位置的API是放在彼此独立的两个包中的，地图包是com.google.android.maps，位置包是android.location。前两节我们介绍了android.location包，下面我们重点分析com.google.android.maps包。为了更容易把强大的绘图功能添加到应用程序里，Google提供了一个地图的外部类库，在包com.google.android.maps中。该包提供下载、渲染和缓存地图功能，以及各种显示选项和集成控制缓存。地图包里的关键类是com.google.android.maps.MapView，它是ViewGroup的子类。一个MapView显示从谷歌地图服务获得的地图数据。当MapView具有焦点时，它将捕捉按键和触摸事件，地图会自动平移和缩放，包括处理网络请求获取更多的地图。它还提供了所有必要的UI元素，以便用户控制地图。应用程序也可以使用MapView类提供的方法编程控制MapView地图上的元素。

google.map.api包中最重要的几个类如下。

- MapActivity：用于显示Map的Activity类，它需要连接底层网络。
- MapView：用于显示地图的View组件，它必须和MapActivity配合使用。
- MapController：用于控制地图的移动。
- Overlay：可显示于地图之上的可绘制的对象。
- GeoPoint：包含经纬度位置的对象。

Android中位置API里面提供了LocationManager类以及Location类。其中 LocationManager用来获得位置服务， Location用来获取位置。具体代码如下：

```
private GeoPoint getGeoPoint () {  
  
    LocationManager locationManager=  
        (LocationManager) getSystemService  
        (Context.LOCATION_SERVICE) ;
```

```
    Location location=
```

```
locationManager.getLastKnownLocation  
    (LocationManager.GPS_PROVIDER) ;  
  
return new GeoPoint ((int) location.getLatitude () , (int)  
    location.getLongitude () ) ;  
  
}
```

一个经纬度对象GeoPoint，接受两个整型的经纬度值，并可以使用GPS获取当前经纬度，这就需要启用GPS模式，而启动GPS模式前先要判断是否启用了GPS，如果没有则跳到Settings中请求用户启用。相关的设定代码如下：

```
boolean flag=locationManager.isProviderEnabled  
    (LocationManager.GPS_PROVIDER) ;  
  
if (!flag) {  
  
Intent intent=new Intent (Settings.ACTION_SECURITY_SETTINGS) ;  
  
}
```

11.2.4 根据位置刷新地图显示

当设备在移动的时候，其位置是不断变化的，要想检测到位置的变化并显示出来，需要用到位置监听器LocationListener，如果要查询出最佳

的数据，需要设置Criteria。

获取最佳位置服务的代码如下：

//获得最佳位置服务

```
private Criteria getCriteria () {  
    Criteria criteria=new Criteria () ;  
  
    criteria.setAccuracy (Criteria.ACCURACY_FINE) ; //高精度  
  
    criteria.setAltitudeRequired (false) ; //海拔  
  
    criteria.setBearingRequired (false) ; //地轴线  
  
    criteria.setCostAllowed (false) ; //付费  
  
    criteria.setPowerRequirement (Criteria.POWER_LOW) ; //电量低  
  
    return criteria ;  
}
```

在实例化LocationListener的时候，其中的回调函数较多。处理位置变化的回调函数为onLocationChanged，具体代码如下：

//位置监听器，监听位置的改变

```
LocationListener locationListener=new LocationListener () {
```

```
//当位置变化时激发
```

```
@Override
```

```
public void onLocationChanged (Location location) {
```

```
//根据位置获取经纬度对象
```

```
getGeoPoint (location) ;
```

```
}
```

```
//GPS或者network可用时激发
```

```
@Override
```

```
public void onProviderDisabled (String provider) {
```

```
}
```

```
//GPS或者network不可用时激发
```

```
@Override
```

```
public void onProviderEnabled (String provider) {
```

```
}
```

```
//GPS或者network状态改变时激发
```

```
@Override
```

```
public void onStatusChanged (String provider,int status,Bundle extras) {
```

```
switch (status) {
```

```
case LocationProvider.OUT_OF_SERVICE :
```

```
//不再服务
```

```
break ;
```

```
case LocationProvider.TEMPORARILY_UNAVAILABLE :
```

```
//暂时不可用
```

```
break ;
```

```
case LocationProvider.AVAILABLE :
```

```
//可用
```

```
break ;
```

```
}
```

```
}
```

```
} ;
```

定义好了监听处理类，就可以在LocationManager中注册这个监听。

```
private void registeListener () {
```

```
    LocationManager locationManager
```

```
    = (LocationManager) getSystemService  
        (Context.LOCATION_SERVICE) ;
```

```
    if ( ! locationManager.isProviderEnabled
```

```
        (LocationManager.GPS_PROVIDER) ) {
```

//如果GPS不可用，则跳转到Settings中进行设置

```
        Intent intent=new Intent (Settings.ACTION_SECURITY_SETTINGS) ;
```

```
        startActivity (intent) ;
```

```
}
```

//查询最适合的服务信息

```
String provider=locationManager.getBestProvider (getCriteria () ,  
true) ;  
  
//设置位置监听器，间隔5秒改变一次  
  
locationManager.requestLocationUpdates (provider, 5000, 0,  
locationListener) ;  
  
}
```

其中，语句locationManager.requestLocationUpdates (provider, 5000, 0, locationListener) 的第一个参数表示位置提供器，第二个参数表示多少毫秒请求一次，第三个参数表示移动最小距离（米），第四个参数表示监听类。

当然，也可以取消位置监听，其调用的方法如下：

```
locationManager.removeUpdates (locationListener) ;
```

把经纬度解析成实际地址，可以采用Geocoder类的getFromLocation方法解析位置信息，然后返回一系列地址信息List<Address>，再根据最符合的Address获取国家、省市、街道等地址信息。实现代码如下：

```
//根据经纬度获得地址（国家、省市、街道等）
```

```
private void showAddressFromGeoPoint (GeoPoint gPoint) throws  
IOException{  
  
StringBuilder sb=new StringBuilder () ;  
  
Geocoder geoCode=new Geocoder (this,Locale.getDefault ()) ;  
  
List<Address>addresses=geoCode.getFromLocation (gPoint.  
getLatitudeE6 () /1E6, gPoint.getLongitudeE6 () /1E6, 1) ;  
  
if (addresses.size () >0) {  
  
Address address=addresses.get (0) ;  
  
for (int i=0 ; i<address.getMaxAddressLineIndex () ; i++) {  
  
sb.append (address.getAddressLine (i) +"\n") ;  
  
}  
  
sb.append (address.getLocality () +"\n") ;  
  
sb.append (address.getPostalCode () +"\n") ;  
  
sb.append (address.getCountryName () +"\n") ;
```

```
        Toast.makeText (this, sb.toString () , Toast.LENGTH_LONG) .show  
        () ;  
  
    }  
  
}
```

以上是把经纬度、位置等信息解析成地址，下面是根据实际地址获取经纬度。跟上述方式一样，也是调用Geocoder的getFromLocationName方法，获得地址列表，然后取出最佳的地址，再获取经纬度。实现代码如下：

```
//根据地址获取经纬度信息GeoPoint  
  
private GeoPoint getGeoPointFromAddressName (String locationName)  
throws IOException{  
  
    Geocoder geoCode=new Geocoder (this, Locale.getDefault ()) ;  
  
    List<Address> addresses=geoCode.getFromLocationName  
    (locationName, 1) ;  
  
    if (addresses !=null) {  
  
        double lati=addresses.get (0) .getLatitude () ;
```

```
        double longi=addresses.get (0) .getLongitude () ;  
  
        return new GeoPoint ( (int) (lati*1E6) , (int) (longi*1E6) ) ;  
  
    }  
  
    return null ;  
  
}
```

11.3 媒体传输协议

11.3.1 MTP和PTP简介

媒体传输协议（Media Transfer Protocol,MTP）是一个基于图片传输协议（Picture Transfer Protocol,PTP）的自定义扩展协议，主要用于传输媒体文件。PTP的设计目的是从数码相机中下载照片，MTP则支持数字音频播放器上的音乐文件、便携式媒体播放器上的媒体文件及个人数字助理的个人信息的传输。

MTP是Windows Media框架的一部分，因此与Windows Media Player密切相关。Windows Vista中内置了对MTP的支持。Windows XP则需要安装Windows Media Player 10或更高版本来获得对MTP的支持。Mac以及Linux系统有相应的软件包提供对MTP的支持。

MTP既可以实现在USB协议上，也可以实现在TCP/IP协议上，目前大部分都是基于USB协议。MTP是微软免费向数码相机、媒体设备等厂商公开的连接技术，这些厂商可以将其写入自己设备的“固件”当中。

Windows内置对MTP的支持后，数码相机用户不用再额外安装驱动程序就能将自己的数码设备连接至微软操作系统的计算机。利用MTP，数码相机将被Windows识别为和USB闪存驱动器一样的设备，用户可以方便地实现媒体文件的传输与共享。

android.mtp包可以使数码相机和其他设备直接使用PTP或MTP与系统保持连接，上层App可以接收到通知，从而管理这些设备的文件和存储传输等。

android.mtp包提供了与PTP或MTP相关的MtpConstants、MtpDevice、MtpDeviceInfo、MtpObjectInfo、MtpStorageInfo等5个类，具体描述如下。

- MtpConstants：该类包含了MTP/PTP规范中定义的一些常量值。
- MtpDevice：该类代表一个连接在USB host总线上的MTP或PTP设备。应用程序可以通过引用UsbDevice实例化该类型的一个对象，然后就可以使用该类中的方法来获取有关设备和存储在设备上的对象信息，以及打开连接和传输数据。
- MtpDeviceInfo：该类封装了一个MTP设备的信息。
- MtpObjectInfo：该类封装了一个MTP设备上的一个对象的信息。
- MtpStorageInfo：该类封装了一个MTP设备上的一个存储单元的信息。

11.3.2 定义MTP和PTP的类型

MtpDevice类代表一个连接在USB host总线上的MTP或PTP设备。应用程序可以通过引用UsbDevice实例化该类型的一个对象，然后就可以使用该类中的方法来获取有关设备和存储在设备上的对象信息，以及打开连接和传输数据。

其中重要的类和方法如下。

- close () : 关闭与MtpDevice对象相关的所有资源。
- deleteObject () : 删除设备上的对象。
- getDeviceId () : 返回该USB设备的USBID号。
- getDeviceInfo () : 返回该设备的MtpDeviceInfo信息。
- getDeviceName () : 返回该USB设备的名称。
- getObject () : 以字节数组形式返回对象中的数据。
- getObjectHandles () : 返回给定的存储单元中所有对象的对象句柄列表。
- getObjectInfo () : 获取一个对象的MtpObjectInfo信息。
- getParent () : 获取设备上的对象的父对象的句柄。
- getStorageId () : 返回MTP对象存储单元的存储ID号。

- `getStorageIds ()`：返回此设备上的所有存储单元的ID列表。
- `getStorageInfo ()`：获取一个存储单元的MtpStorageInfo信息。
- `getThumbnail ()`：以字节数组形式返回对象的粗略数据。
- `importFile ()`：复制对象数据到外部存储文件。
- `open ()`：打开MTP设备。

MtpDevice类的定义代码如下：

```
package android.mtp ;  
  
import android.hardware.usb.UsbDevice ;  
  
import android.hardware.usb.UsbDeviceConnection ;  
  
import android.os.ParcelFileDescriptor ;  
  
import android.util.Log ;  
  
public final class MtpDevice{  
  
    private static final String TAG="MtpDevice" ;  
  
    private final UsbDevice mDevice ;
```

```
static{System.loadLibrary ("media_jni") ;  
}  
  
//MtpClient构造函数， 其中参数device为MTP或PTP设备  
  
public MtpDevice (UsbDevice device) {  
  
    mDevice=device ;  
  
}  
  
/*打开MTP设备。当该方法调用失败或者应用调用close () 时， 连接  
会被关闭。如果设备被成功打开则返回true*/  
  
public boolean open (UsbDeviceConnection connection) {  
  
    boolean result=native_open (mDevice.getDeviceName () ,  
        connection.getFileDescriptor () ) ;  
  
    if ( ! result) {  
  
        connection.close () ;  
  
    }  
  
    return result ;
```

```
}
```

```
/*关闭与MtpDevice对象相关的所有资源。这个方法被调用后，在使用  
open重新打开前，该对象是不能被使用的*/
```

```
public void close () {
```

```
    native_close () ;
```

```
}
```

```
@Override
```

```
protected void finalize () throws Throwable{
```

```
    try{
```

```
        native_close () ;
```

```
    }finally{
```

```
        super.finalize () ;
```

```
}
```

```
}
```

```
//返回该USB设备的名称
```

```
public String getDeviceName () {  
    return mDevice.getDeviceName () ;  
}  
  
//返回该USB设备的USB ID号  
  
public int getDeviceId () {  
    return mDevice.getDeviceId () ;  
}  
  
}  
  
@Override  
  
public String toString () {  
    return mDevice.getDeviceName () ;  
}  
  
}  
  
//返回该设备的MtpDeviceInfo信息  
  
public MtpDeviceInfo getDeviceInfo () {  
    return native_get_device_info () ;
```

}

/*返回此设备上的所有存储单元的ID列表，每个存储单元的信息可通过
getstorageinfo获取*/

```
public int[]getStorageIds () {  
  
    return native_get_storage_ids () ;  
  
}
```

//返回给定的存储单元中所有对象的对象句柄列表

```
public int[]getObjectHandles (int storageId,int format,int objectHandle) {  
  
    return native_get_object_handles (storageId,format,objectHandle) ;  
  
}
```

/*以字节数组形式返回对象中的数据。

这个调用可能因非常耗时而阻塞，具体是否阻塞取决于数据大小及设备速度。

该函数成功返回对象数据，或因读取失败返回NULL*/

```
public byte[]getObject (int objectHandle,int objectSize) {
```

```
    return native_get_object (objectHandle,objectSize) ;  
}  
  
/*以字节数组形式返回对象的粗略数据。
```

数据的大小和格式由getThumbCompressedSize和getThumbFormat决定，
典型设备的格式为JPEG。

该函数返回对象粗略数据，或因读取失败返回NULL*/

```
public byte[]getThumbnail (int objectHandle) {  
  
    return native_get_thumbnail (objectHandle) ;  
  
}
```

/*获取一个存储单元的MtpStorageInfo信息。参数storageID为存储单元
的ID*/

```
public MtpStorageInfo getStorageInfo (int storageId) {  
  
    return native_get_storage_info (storageId) ;  
  
}
```

/*获取一个对象的MtpObjectInfo信息。参数objectHandle为对象的句柄
*/

```
public MtpObjectInfo getObjectInfo (int objectHandle) {  
  
    return native_get_object_info (objectHandle) ;  
  
}
```

/*删除设备上的对象。这个调用可能会阻塞，因为删除某些设备上包含许多文件的目录可能需要很长的时间。参数objectHandle为要删除的对象的句柄，如果删除成功，则返回true*/

```
public boolean deleteObject (int objectHandle) {  
  
    return native_delete_object (objectHandle) ;  
  
}
```

/*获取设备上的对象的父对象的句柄。

参数objectHandle为要查询的对象。函数返回父对象句柄，或因是存储根而返回0*/

```
public long getParent (int objectHandle) {  
  
    return native_get_parent (objectHandle) ;
```

```
}
```

/*获取设备上包含给定对象的存储单元的ID。参数objectHandle为要查询的对象。函数返回该对象的存储单元ID*/

```
public long getStorageId (int objectHandle) {  
  
    return native_get_storage_id (objectHandle) ;  
  
}
```

/* 复制对象数据到外部存储文件。

这个调用可能因非常耗时而阻塞，是否阻塞取决于数据大小及设备速度。

参数objectHandle为要读取的对象的句柄。

参数destPath为文件传输的路径，此路径应该是在外部存储中由getExternalStorageDirectory定义的。

如果文件传输成功则返回true*/

```
public boolean importFile (int objectHandle,String destPath) {  
  
    return native_import_file (objectHandle,destPath) ;
```

}

//由JNI使用

private int mNativeContext ;

private native boolean native_open (String deviceName,int fd) ;

private native void native_close () ;

private native MtpDeviceInfo native_get_device_info () ;

private native int[]native_get_storage_ids () ;

private native MtpStorageInfo native_get_storage_info (int storageId) ;

private native int[]native_get_object_handles (int storageId,int format,int objectHandle) ;

private native MtpObjectInfo native_get_object_info (int objectHandle) ;

private native byte[]native_get_object (int objectHandle,int objectSize) ;

private native byte[]native_get_thumbnail (int objectHandle) ;

private native boolean native_delete_object (int objectHandle) ;

private native long native_get_parent (int objectHandle) ;

```
private native long native_get_storage_id (int objectHandle) ;  
  
private native boolean native_import_file (int objectHandle,String  
destPath) ;
```

11.3.3 封装MTP设备信息

MtpDeviceInfo类封装了一个MTP设备的信息，其中包含了很多MTP设备的基本信息。MtpDeviceInfo类首先声明了一些字符串存储对象，包括MTP设备的制造商名称、MTP设备的型号、MTP设备版本信息和MTP设备的唯一序列号，这些都是比较重要的MTP设备信息，其具体定义代码如下：

```
package android.mtp ;  
  
public class MtpDeviceInfo{  
  
    private String mManufacturer ;  
  
    private String mModel ;  
  
    private String mVersion ;  
  
    private String mSerialNumber ;  
  
    //只能通过JNI实例化
```

```
private MtpDeviceInfo () {  
}  
  
//返回MTP设备的制造商名称  
  
public final String getManufacturer () {  
  
    return mManufacturer ;  
  
}  
  
//返回MTP设备的型号  
  
public final String getModel () {  
  
    return mModel ;  
  
}  
  
//返回MTP设备版本信息字符串  
  
public final String getVersion () {  
  
    return mVersion ;  
  
}
```

```
//返回MTP设备的唯一序列号

public final String getSerialNumber () {

    return mSerialNumber ;

}

}
```

11.3.4 封装MTP对象的信息

MtpObjectInfo类封装了一个MTP设备上的一个对象的信息，MTP设备对象信息比MTP设备信息包含的内容要多很多。不但包含了硬件方面的信息，比如存储ID号、格式信息等，还包含了较多的软件信息，比如MTP对象的关键字、MTP对象的日期等，其具体的定义代码如下：

```
package android.mtp ;

public final class MtpObjectInfo{

    private int mHandle ;

    private int mStorageId ;

    private int mFormat ;
```

```
private int mProtectionStatus ;  
  
private int mCompressedSize ;  
  
private int mThumbFormat ;  
  
private int mThumbCompressedSize ;  
  
private int mThumbPixWidth ;  
  
private int mThumbPixHeight ;  
  
private int mImagePixWidth ;  
  
private int mImagePixHeight ;  
  
private int mImagePixDepth ;  
  
private int mParent ;  
  
private int mAssociationType ;  
  
private int mAssociationDesc ;  
  
private int mSequenceNumber ;  
  
private String mName ;
```

```
private long mDateCreated ;  
  
private long mDateModified ;  
  
private String mKeywords ;  
  
//只能通过JNI实例化  
  
private MtpObjectInfo () {  
  
}  
  
//返回该MTP设备的对象句柄  
  
public final int getObjectHandle () {  
  
    return mHandle ;  
  
}  
  
//返回MTP对象存储单元的存储ID  
  
public final int getStorageId () {  
  
    return mStorageId ;  
  
}
```

//返回MTP对象的格式信息

```
public final int getFormat () {
```

```
    return mFormat ;
```

```
}
```

/*返回MTP对象的保护状态。

可能的值有PROTECTION_STATUS_NONE、

PROTECTION_STATUS_READ_ONLY、

PROTECTION_STATUS_NON_TRANSFERABLE_DATA*/

```
public final int getProtectionStatus () {
```

```
    return mProtectionStatus ;
```

```
}
```

//返回MTP对象的大小 (size)

```
public final int getCompressedSize () {
```

```
    return mCompressedSize ;
```

```
}
```

//返回MTP缩略对象的格式信息，如果对象没有缩略则返回0

```
public final int getThumbFormat () {  
    return mThumbFormat ;  
}
```

//返回MTP缩略对象的大小，如果对象没有缩略则返回0

```
public final int getThumbCompressedSize () {  
    return mThumbCompressedSize ;  
}
```

//以像素为单位返回MTP缩略对象的宽度，如果对象没有缩略则返回0

```
public final int getThumbPixWidth () {  
    return mThumbPixWidth ;  
}
```

//以像素为单位返回MTP缩略对象的高度，如果对象没有缩略则返回0

```
public final int getThumbPixHeight () {
```

```
return mThumbPixHeight ;
```

```
}
```

```
//以像素为单位返回MTP对象的宽度，如果没有图片对象则返回0
```

```
public final int getImagePixWidth () {
```

```
return mImagePixWidth ;
```

```
}
```

```
//以像素为单位返回MTP对象的高度，如果没有图片对象则返回0
```

```
public final int getImagePixHeight () {
```

```
return mImagePixHeight ;
```

```
}
```

```
//以位为单位返回MTP对象的像素深度，如果没有图片对象则返回0
```

```
public final int getImagePixDepth () {
```

```
return mImagePixDepth ;
```

```
}
```

//返回父对象的对象句柄，如果是存储单元的根目录则返回0

```
public final int getParent () {  
    return mParent ;  
}
```

//返回MTP对象相关联的类型，如果不是有效格式则返回0

```
public final int getAssociationType () {  
    return mAssociationType ;  
}
```

//返回MTP对象相关的描述，如果不是有效格式则返回0

```
public final int getAssociationDesc () {  
    return mAssociationDesc ;  
}
```

//返回MTP对象的序列号。该域在MTP设备上一般不用，但有时用于定义PTP相机中的图片序列

```
public final int getSequenceNumber () {
```

```
return mSequenceNumber ;
```

```
}
```

```
//返回MTP对象的名称
```

```
public final String getName () {
```

```
return mName ;
```

```
}
```

```
//返回MTP对象的创建日期
```

```
public final long getDateCreated () {
```

```
return mDateCreated ;
```

```
}
```

```
//返回MTP对象的修改日期
```

```
public final long getDateModified () {
```

```
return mDateModified ;
```

```
}
```

```
//返回以逗号分隔的MTP对象的关键字列表
```

```
public final String getKeywords () {
```

```
    return mKeywords ;
```

```
}
```

```
}
```

11.3.5 封装MTP设备上存储单元的信息

MtpStorage类代表了MTP设备上的一个存储单元，包含存储单元的存储ID、存储单元在文件系统中的文件路径、存储单元的字符串描述等信息。其定义代码如下：

```
package android.mtp ;
```

```
import android.content.Context ;
```

```
import android.os.storage.StorageVolume ;
```

```
//该类代表了MTP设备上的一个存储单元
```

```
public class MtpStorage{
```

```
    private final int mStorageId ;
```

```
private final String mPath ;  
  
private final String mDescription ;  
  
private final long mReserveSpace ;  
  
private final boolean mRemovable ;  
  
private final long mMaxFileSize ;  
  
public MtpStorage (StorageVolume volume,Context context) {  
  
    mStorageId=volume.getStorageId () ;  
  
    mPath=volume.getPath () ;  
  
    mDescription=context.getResources () .  
        getString (volume.getDescriptionId () ) ;  
  
    mReserveSpace=volume.getMtpReserveSpace () ;  
  
    mRemovable=volume.isRemovable () ;  
  
    mMaxFileSize=volume.getMaxFileSize () ;  
  
}
```

//返回存储单元的存储ID

```
public final int getStorageId () {  
    return mStorageId ;  
}
```

//产生给定索引的存储ID

```
public static int getStorageId (int index) {  
    //0x00010001为主存储ID  
    return ( (index+1) <<16) +1 ;  
}
```

//返回存储单元在文件系统中的文件路径

```
public final String getPath () {  
    return mPath ;  
}
```

//返回存储单元的字符串描述

```
public final String getDescription () {  
    return mDescription ;  
}  
  
//返回文件存储系统的预留空间  
  
public final long getReserveSpace () {  
    return mReserveSpace ;  
}  
  
}  
  
//如果是可移动存储则返回true  
  
public final boolean isRemovable () {  
    return mRemovable ;  
}  
  
}  
  
//返回能存储的文件大小上限，如果存储文件大小不受限制则返回0  
  
public long getMaxFileSize () {  
    return mMaxFileSize ;
```

}

}

MtpStorageInfo类封装了一个MTP设备上的一个存储单元的信息，其定义代码如下：

```
package android.mtp ;
```

```
//该类封装了MTP设备上的一个存储单元的信息
```

```
public final class MtpStorageInfo{
```

```
    private int mStorageId ;
```

```
    private long mMaxCapacity ;
```

```
    private long mFreeSpace ;
```

```
    private String mDescription ;
```

```
    private String mVolumeIdentifier ;
```

```
//只能通过JNI实例化
```

```
    private MtpStorageInfo () {
```

}

//返回存储单元的存储ID，该存储ID是存储MTP设备上存储单元的唯一标识

```
public final int getStorageId () {
```

```
    return mStorageId ;
```

```
}
```

//以字节为单位返回存储单元的最大存储容量

```
public final long getMaxCapacity () {
```

```
    return mMaxCapacity ;
```

```
}
```

//以字节为单位返回存储单元的剩余空间

```
public final long getFreeSpace () {
```

```
    return mFreeSpace ;
```

```
}
```

//返回存储单元的字符串描述

```
public final String getDescription () {
```

```
    return mDescription ;  
}  
  
//返回存储单元的卷标识符  
  
public final String getVolumeIdentifier () {  
    return mVolumeIdentifier ;  
}  
}
```

11.4 小结

本章首先在讨论Android中几种XML解析方式各自优缺点的基础上，重点阐述了Android中使用SAX方式解析XML的原理，并对SAX发现XML根元素与子元素的过程做了具体分析；接着讲解了Android中如何实现基于位置的服务，并叙述了Google地图显示的流程；最后讲解了媒体传输协议（MTP）的概念，并对MTP设备、MTP设备上的对象与存储单元等做了具体介绍。

Table of Contents

序

第1章 Android网络编程概要

1.1 Android简介

 1.1.1 Android的发展

 1.1.2 Android功能特性

 1.1.3 Android系统构架

1.2 Android网络程序的功能

 1.2.1 通信功能

 1.2.2 及时分享

 1.2.3 个人管理

 1.2.4 娱乐游戏

 1.2.5 企业应用

1.3 设置Android开发环境

 1.3.1 相关下载

 1.3.2 安装ADT

 1.3.3 Hello World !

1.4 网络应用实战案例

 1.4.1 加载一个页面

 1.4.2 下载一个页面

1.5 小结

第2章 Android基本网络技术和编程实践

2.1 计算机网络及其协议

 2.1.1 计算机网络概述

 2.1.2 网络协议概述

 2.1.3 IP、 TCP和UDP协议

2.2 在Android中使用TCP、 UDP协议

2.2.1 Socket基础

2.2.2 使用TCP通信

2.2.3 使用UDP通信

2.3 Socket实战案例

2.3.1 Socket聊天举例

2.3.2 FTP客户端

2.3.3 Telnet客户端

2.4 小结

第3章 Android基本Web技术和编程实践

3.1 HTTP协议

3.1.1 HTTP简介

3.1.2 实战案例：基于HTTP协议的文件上传

3.2 Android中的HTTP编程

3.2.1 HttpClient和URLConnection

3.2.2 Post和Get在HttpClient的使用

3.2.3 实战案例：使用HttpClient和URLConnection访问维基百科

3.3 Android处理JSON

3.3.1 JSON简介

3.3.2 JSON数据解析

3.3.3 JSON打包

3.3.4 实战案例：JSON解析wikipedia内容

3.4 Android处理SOAP

3.4.1 SOAP简介

3.4.2 SOAP消息

3.4.3 实战案例：SOAP解析天气服务

3.5 Android对HTML的处理

3.5.1 解析HTML

3.5.2 HTML适配屏幕

3.5.3 JavaScript混合编程

3.5.4 实战案例：Android自定义打开HTML页面

3.6 小结

第4章 Android常见网络接口编程

4.1 Android解析和创建XML

4.1.1 XML简介

4.1.2 DOM解析XML

4.1.3 SAX解析XML

4.1.4 PULL解析XML

4.1.5 实战案例：Android中创建XML

4.2 Android订阅RSS

4.2.1 RSS简介

4.2.2 实战案例：简单RSS阅读器

4.3 Android Email编程

4.3.1 Android发送Email

4.3.2 实战案例：Android下Email的Base64加密

4.4 Android网络安全

4.4.1 Android网络安全简介

4.4.2 Android加密和解密

4.4.3 实战案例：Android应用添加签名

4.5 OAuth认证

4.5.1 OAuth简介

4.5.2 实战案例：使用OAuth接口

4.6 小结

第5章 Android网络模块编程

5.1 Android地图和定位

5.1.1 获取map-api密钥

5.1.2 获得位置

5.1.3 实战案例：利用MapView显示地图

5.2 USB编程

- 5.2.1 USB主从设备
- 5.2.2 USB Accessory Mode
- 5.2.3 USB Host Mode
- 5.2.4 实战案例：Android和Arduino交互

5.3 Wi-Fi编程

- 5.3.1 Android Wi-Fi相关类
- 5.3.2 Android Wi-Fi基本操作
- 5.3.3 实战案例：使用Wi-Fi直连方式传输文件

5.4 蓝牙编程

- 5.4.1 蓝牙简介
- 5.4.2 Android蓝牙API分析
- 5.4.3 Android蓝牙基本操作
- 5.4.4 实战案例：蓝牙连接

5.5 NFC编程简介

- 5.5.1 NFC技术简介
- 5.5.2 NFC API简介
- 5.5.3 NFC处理流程分析

5.6 小结

第6章 Android线程、数据存取、缓存和UI同步

6.1 Android线程

- 6.1.1 Android线程模型
- 6.1.2 异步任务类
- 6.1.3 实战案例：利用AsyncTask实现多线程下载

6.2 数据存取

- 6.2.1 Shared Preferences 数据存储
- 6.2.2 Internal Storage 数据存储
- 6.2.3 External Storage 数据存储
- 6.2.4 SQLite Databases 数据存储
- 6.2.5 实战案例：SQLite数据库操作

6.3 网络判定

- 6.3.1 判断用户是否连接
- 6.3.2 判断网络连接的类型
- 6.3.3 监控网络连接改变
- 6.3.4 实战案例：根据广播消息判断网络连接情况

6.4 消息缓存

- 6.4.1 Android中的缓存机制
- 6.4.2 实战案例：下载、缓存和显示图片

6.5 界面更新

- 6.5.1 刷新数据时的界面更新
- 6.5.2 完成任务时的界面更新
- 6.5.3 实战案例：自定义列表显示更新

6.6 小结

第 7 章 基于SIP协议的VoIP应用

7.1 SIP协议简介

7.2 SIP服务器搭建

- 7.2.1 下载安装Brekeke SIP Server
- 7.2.2 访问服务器
- 7.2.3 启动服务器

7.3 SIP程序设置

- 7.3.1 Android SIP API 中的类和接口
- 7.3.2 Android极限列表
- 7.3.3 完整的Manifest文件

7.4 SIP初始化通话

- 7.4.1 SipManager对象
- 7.4.2 SipProfile对象

7.5 监听SIP通话

- 7.5.1 创建监听器
- 7.5.2 拨打电话

7.5.3 接收呼叫

7.6 实战案例：SIP通话

7.7 小结

第 8 章 基于XMPP协议的即时通信应用

8.1 XMPP协议简介

8.2 使用Openfire搭建XMPP服务器

8.3 登录XMPP服务器

 8.3.1 Asmack相关类

 8.3.2 登录XMPP服务器

8.4 联系人相关操作

 8.4.1 获取联系人列表

 8.4.2 获取联系人状态

 8.4.3 添加和删除联系人

 8.4.4 监听联系人添加信息

8.5 消息处理

 8.5.1 接收消息

 8.5.2 发送消息

8.6 实战案例：XMPP多人聊天

 8.6.1 创建新多人聊天室

 8.6.2 加入聊天室

 8.6.3 发送和接收消息

8.7 小结

第 9 章 Android对HTML的处理

9.1 Android HTML处理关键类

9.2 HTMLViewer分析

9.3 浏览器源代码解析

 9.3.1 WebView加载入口分析

 9.3.2 调用JavaScript接口

9.4 WebKit简单分析

- 9.4.1 HTTP Cache管理
- 9.4.2 Cookie管理
- 9.4.3 处理HTTP认证以及证书
- 9.4.4 处理JavaScript的请求
- 9.4.5 处理MIME类型
- 9.4.6 访问WebView的历史
- 9.4.7 保存网站图标
- 9.4.8 WebStorage
- 9.4.9 处理UI
- 9.4.10 Web设置分析
- 9.4.11 HTML5音视频处理
- 9.4.12 缩放和下载
- 9.4.13 插件管理

9.5 小结

第 10 章 Android网络处理分析

- 10.1 Android网络处理关键类及其说明
- 10.2 Android网络处理流程
 - 10.2.1 监控网络连接状态
 - 10.2.2 认证类
 - 10.2.3 DHCP状态机
 - 10.2.4 LocalServerSocket
 - 10.2.5 响应邮件请求
 - 10.2.6 提供网络信息
 - 10.2.7 Proxy类
 - 10.2.8 VPN服务
- 10.3 Android封装的HTTP处理类
 - 10.3.1 AndroidHttpClient类和DefaultHttpClient类
 - 10.3.2 SSL认证信息处理类
 - 10.3.3 SSL错误信息处理
 - 10.3.4 AndroidHttpClient

- 10.4 Android RTP协议
 - 10.4.1 传输音频码
 - 10.4.2 AudioGroup
 - 10.4.3 语音流RtpStream和AudioStream
- 10.5 Android SIP协议
 - 10.5.1 SIP通话简介
 - 10.5.2 SIP初始化
 - 10.5.3 SipProfile
 - 10.5.4 SipSession
 - 10.5.5 SIP包错误处理
- 10.6 小结

第 11 章 Android网络应用分析

- 11.1 Android中使用SAX解析XML
 - 11.1.1 几种XML解析方式讨论
 - 11.1.2 SAX解析XML的原理
 - 11.1.3 SAX发现XML的根元素
 - 11.1.4 SAX发现XML的子元素
- 11.2 基于位置的服务
 - 11.2.1 位置服务的基本概念
 - 11.2.2 位置服务的基本类
 - 11.2.3 调用Google地图
 - 11.2.4 根据位置刷新地图显示
- 11.3 媒体传输协议
 - 11.3.1 MTP和PTP简介
 - 11.3.2 定义MTP和PTP的类型
 - 11.3.3 封装MTP设备信息
 - 11.3.4 封装MTP对象的信息
 - 11.3.5 封装MTP设备上存储单元的信息
- 11.4 小结



Your gateway to knowledge and culture. Accessible for everyone.



z-library.sk

z-lib.gs

z-lib.fm

go-to-library.sk



[Official Telegram channel](#)



[Z-Access](#)



<https://wikipedia.org/wiki/Z-Library>