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

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。

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

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。

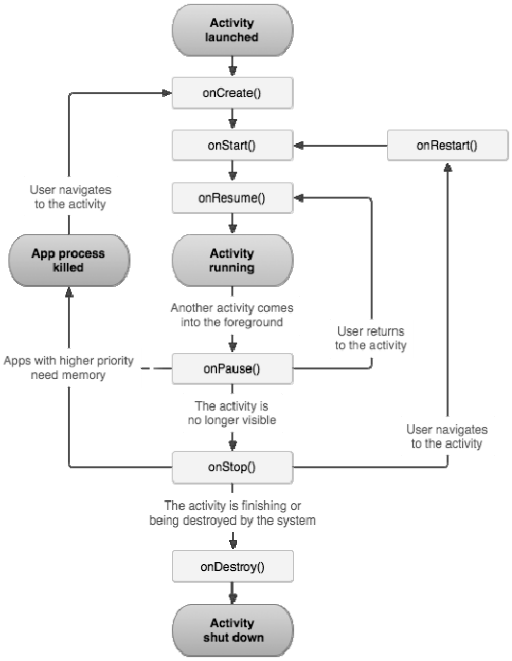

图 13-6 Activity 生命周期

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