基于Google发布的Jelly Bean原始代码，
讲述Android系统的内部静态结构关系和内部运行机制，
为你呈现原汁原味的Android代码分析大餐！

Broadview®
www.broadview.com.cn

深入剖析Android系统

杨长刚 著

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

深入剖析Android系统

杨长刚 著

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

内 容 简 介

本书以 Android Jelly Bean（4.1）的代码为蓝本，对 Android 的部分关键代码进行了注释分析，并辅以大量插图，讲述了 Android 大部分子系统中的模块和类的静态结构，让读者对 Android 系统的内部静态结构有着“类”粒度这一层级上的认识和了解。同时，也对关键类和函数的代码调用流程、运行时刻所位于的进程和线程上下文等动态运行场景进行了分析讲述，让读者深刻理解 Android 系统内部是如何运行的。本书直接对 Source Insight 进行截图，保留了代码的原始行号、英文注释等信息并进行了高亮显示，方便读者阅读；代码中同时添加了作者所做的中文注释说明。

图书在版编目（CIP）数据

深入剖析 Android 系统/杨长刚著.—北京：电子工业出版社，2013.1
ISBN 978-7-121-19374-3

Ⅰ.①深… Ⅱ.①杨… Ⅲ.①移动终端－应用程序－程序设计 Ⅳ.①TN929.53

中国版本图书馆 CIP 数据核字（2012）第 311962 号

责任编辑：葛　娜
特约编辑：赵树刚
印　　刷：北京中新伟业印刷有限公司
装　　订：北京中新伟业印刷有限公司
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：29　字数：742.4 千字
印　　次：2013 年 1 月第 1 次印刷
印　　数：4000 册　　定价：59.00 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，联系电话：（010）68279077；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

为了让读者可以对 Android 系统进行“有血有肉”的感知，而不是停留在抽象的原理和概念之上，本书对 Android 的代码进行了分析，进而“提炼”出 Android 的全貌。又因为 Android 有着海量代码，故只分析了 Android 系统的一些主要模块和类，不对各个细节进行全面分析，这样可以在有限的篇幅中让读者了解 Android 的内部结构和运行机制，同时避免让读者陷入海量代码的云雾中而不得要领。

由于 Android 系统升级较快，有些代码变动很大。对设计上有重大改变的一些模块，笔者也兼顾提及了 Android 的 2.x 版本和 4.0 版本中的设计，甚至个别模块在未来的版本中的可能的演进方向。

现将本书各章内容介绍如下:

第 1 章介绍了智能指针。在 Android 的 native 层的 C++代码中，存在着大量形如 sp 和 wp 模板的运用，它们都是智能指针模板。通过本章，读者将掌握 native 层的类的对象的生命周期。

第 2 章介绍了 Android 中消息队列和线程处理机制。这涉及 native 层的 Looper 和 Java 层的 Looper、Handler 及 Message 等，让读者更好地理解 Android 中的代码执行流程场景和所在的线程上下文。

第 3 章详细介绍了 Android 中最重要的一种机制 Binder IPC 及其应用。在 Android 中，一个简单的功能或上层 API 的一个简单调用，往往需要跨越多个进程。一个子系统的功能也往往由各个进程中的模块来完成，这就要用到 Android 的核心机制 Binder IPC。Binder IPC 不仅位于 native 代码中，也大量存在于 Java 层的代码中，因此以 Java 层的播放服务（IMediaPlaybackService）和电话状态监听器（PhoneStateListener）为例，详细分析了多个进程之间的相互调用，尤其是后一个例子，双向跨越了多个进程。另外，为了避免大内存数据传输，借助于 Binder IPC 机制，Android 实现了大内存块的跨进程共享。

第 4 章介绍了 HAL 硬件抽象层，让读者了解到 Android 的框架系统如何利用其下面的抽象硬件。

第 5 章介绍了 Android 的启动过程。内容涉及 Android 初始化语言，Linux 系统的第一个进程 init 的启动过程，以及 init 进程如何解析用 Android 初始化语言编写的.rc 脚本文件。最后简要介绍了 Android 系统中的服务所驻留的宿主进程 system_server 的启动过程。

第 6 章介绍了 Android 输入系统，让读者对输入系统的执行过程有一定的了解。

第 7 章介绍了 Android 系统中的大容量存储（MassStorage）系统，让读者熟悉 Android 存储设备的管理机制。

第 8 章介绍了 Android 中的传感器（Sensor）系统，让读者对传感器有所了解。

第 9 章介绍了 RIL，包括 C 语言实现的 RIL 和与其通信的 RILJ（Java 类 RIL）。通过阅

读本章，读者将了解 Java 层上传消息和下送命令的机制，它是 Java 层的电话功能通道。

第 10 章介绍了 Android 系统中的 phone 进程。首先简单介绍了层次状态机，有助于读者了解层次状态机的工作机制，以便于分析使用层次状态机的代码，如 PS 域的数据连接、Wi-Fi 和蓝牙的连接状态等。接着，对 Android 电话功能的实现进行了介绍。

第 11 章对 Android 中的 Graphic 系统进行了分析，其中包括 SurfaceFlinger 中新引入的 VSync 机制。

第 12 章介绍了 Android 中的 OpenGL ES 软件层次栈，让读者了解 OpenGL ES 的层次调用关系，以及如何通过钩子（hook）将库中的 API 关联起来。

第 13 章介绍了 Android 的多媒体系统，主要介绍了播放和录制的过程。通过阅读本章，读者将了解 Java 的 SDK API 层如何调用 native 层的 Service 服务，以及 Service 如何向应用程序发送消息通知。最后，介绍了 Android 中的 Camera。让读者了解硬件抽象层的 Camera 如何向 Java 层发送采样数据。

第 14 章介绍了 Audio 系统的播放和录音过程，让读者了解 Android 中的应用程序进程和 AudioFlinger 之间的数据传输关系。最后，介绍了 AudioFlinger 的工作机理、音效和音频策略服务。通过阅读本章，读者将熟悉 Android 中的音频系统的工作过程。

第 15 章介绍了 Android 的多媒体框架 Stagefright，并提及 Android 版本演进过程中的不同设计理念下的视频帧的渲染输出原理。通过阅读本章，读者将进一步熟悉多媒体系统的处理过程。

第 16 章介绍了 OMXCodec。这将有助于读者了解 Stagefright 如何使用 OMXCodec 进行编解码，以及 OMXCodec 如何使用平台厂家实现的 OMX 插件。

第 17 章介绍了 Android 的 GPS 系统，让读者了解 GPS 部分的工作机制。

第 18 章介绍了 Android 中的 NFC 实现，让读者了解 Android 中的 NFC 的工作机制。

第 19 章介绍了 Android 对 USB 外设的处理过程。

第 20 章简要介绍了 Android 中的蓝牙和 Wi-Fi 的系统架构。

第 21 章介绍了用于生成 tombstone 调试文件的 Debuggerd 守护进程。通过阅读本章，读者将获悉 Android 如何记录 C/C++代码崩溃时的场景信息，以让开发者获取足够多的调试信息。

为了表示对他人劳动成果的尊重和方便读者进行延伸阅读，本书在页脚注释中给出了笔者参阅的文章、文档和书籍的网络链接和说明。对于侧重点不同的文章或书籍，笔者也给读者做了阅读推荐。

由于笔者时间、精力和能力所限，书中涉及的内容定会存在错误之处，还请相关专家及读者批评批正，不胜感激。

在阅读本书前，读者需要具备 C、C++、Java 等编程语言和 Android 的基础知识。读者最好对 Linux 系统、Linux 命令行和 Shell 脚本等有基本了解。若读者对设计模式、Linux 系统编程（推荐伽玛等的《设计模式——可复用面向对象软件的基础》，史蒂文斯和拉戈的《UNIX 环境高级编程》）有所了解，再拥有书中相关子系统的背景知识，那么阅读本书和理解 Android 系统将是件十分轻松的事。

目　录

第 1 章　智能指针

本章描述了智能指针的基本概念，重点分析了强指针和弱指针的构造函数和重载的赋值操作符，让我们了解它们对引用计数和对象生命周期的影响。在系统的服务等 native 代码中，有大量的通过智能指针来使用的 C++对象。掌握本章内容，将有助于我们理解这些 C++对象的生命周期。

1.1　智能指针概述

在 C++中，我们常用指针去使用某个对象实例，但随着代码对该对象使用频率的增加，比如越来越多的其他的类或模块需要使用该对象，那么什么时候安全地释放该对象就是个问题。比如，某处使用完该对象后，认为不再使用，便释放了它，但另外的某些类或模块还需使用，这时，就会引起内存的非法使用问题，在 Linux 下是 signal 11 错误，也就是通常遇到的“段错误”（Segmentation Fault）。为了安全地使用对象指针，可以借助一种封装了普通指针的 C++模板——智能指针来避免这种问题。当使用某对象时，令其引用计数加 1，释放时减 1。当计数减为 0 时，就自动释放指针所指对象。这个引用计数不由开发人员维护，否则又回到直接使用对象指针时的情景。引用计数增减根据智能指针本身这个变量的生命周期决定。

当一个智能指针变量被创建或销毁时，对象的引用计数会自动加 1 或减 1。但是当在两个类中相互引用对方时，即使使用智能指针，当其他地方都释放它们，它们的引用计数也不会变为 0，导致对象不能自动释放。为解决这种问题，引入了弱引用。让两个相互引用的对象一个使用强引用，一个使用弱引用。只要强引用计数为 0（不考虑弱引用计数），就可销毁对象。

Android 代码中随处可见 sp<XXX>和 wp<XXX>。这里的 XXX 是对象的类，sp 和 wp 是类模板，分别实现了强指针（strong pointer）和弱指针（weak pointer）。以类 XXX 具体化强指针模板后，得到一个 sp 模板对象变量，该变量如同普通的变量，若是局部变量，则具有局部变量的生命周期。它又具有指针的属性，可以简单地把它理解成一个指针，但比普通的指针特殊。

如果需要使用智能指针将对象保护起来，只要让其继承引用计数基类 RefBase，并将析构函数声明为 virtual 虚函数，就可以使用智能指针。对于强指针，还可以继承轻量级引用计数模板类 LightRefBase，弱指针则不可以。

1.2 引用计数基类 RefBase

RefBase 是引用计数基类，见文件 frameworks/native/include/utils/RefBase.h 和 frameworks/native/libs/utils/RefBase.cpp（ICS 及以前版本为 frameworks/base/include /utils/RefBase.h 和 frameworks/base/libs/utils/RefBase.cpp），在它里面还定义了一个嵌套类 weakref_type，在其实现文件 RefBase.cpp 里还定义了一个类 weakref_impl。weakref_impl 继承自嵌套类 weakref_type，而 RefBase 中包含一个私有数据成员 mRefs（指向 weakref_impl 的指针）。可以看出，RefBase 中的数据实体就是 weakref_impl，后者继承自其嵌套类 weakref_type，其示意图如图 1-1 所示。

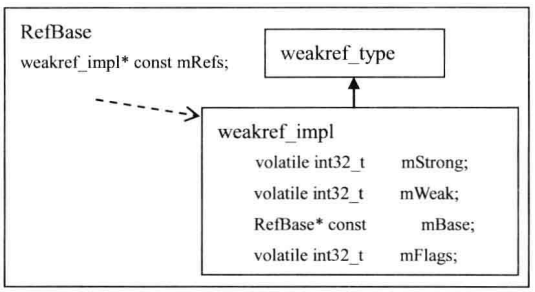

图 1-1 Refbase 结构示意图

这样，相当于将一部分成员函数（weakref_type 的成员函数）和数据成员（weakref_impl 的数据成员）封装起来。它们作为私有的一个逻辑实体，只被 RefBase 内部的实现使用，RefBase 的公有成员才是外界使用的 API。RefBase 通过指针 mRefs 使用内部的引用计数数据。而内部引用计数实体对象也可以通过 mBase 这个成员来使用其“容器”类 RefBase。

weakref_impl 里面的数据成员信息代码如下（见文件 RefBase.cpp）：

```
00052: class RefBase::weakref_impl : public RefBase::weakref_type
00053: {
00054: public:
00055:     volatile int32_t   mStrong;//强引用计数
00056:     volatile int32_t   mWeak;//弱引用计数
00057:     RefBase* const     mBase; //指向其"容器"类RefBase
00058:     volatile int32_t   mFlags;//对象销毁策略标志：若为0，则强引用计数为0时即销毁对象
```

它里面维护着强引用计数和弱引用计数。第四个参数是对象销毁策略标志：若为 0，则强引用计数为 0（OBJECT_LIFETIME_STRONG=0x000，头文件 RefBase.h 的行 0129，见下面的代码）时销毁；若为 1（OBJECT_LIFETIME_WEAK=0x0001），则强、弱引用计数均为 0 时销毁；在较老（ICS 以前）的版本中，还有值为 3（OBJECT_LIFETIME_FOREVER = 0x0003）的选项，这时则由创建者管理，即不依赖 RefBase 引用计数。自 ICS 版本后，已无此选项。定义代码如下：

```
0127:    //! Flags for extendObjectLifetime()
0128:    enum {
0129:      OBJECT_LIFETIME_STRONG  = 0x0000,
0130:      OBJECT_LIFETIME_WEAK    = 0x0001,
0131:      OBJECT_LIFETIME_MASK    = 0x0001
0132:    };
```

可以使用下面的 API 来改变这个策略标志：

```
void RefBase::extendObjectLifetime(int32_t mode)
```

1.3　轻量级引用计数 LightRefBase

在 RefBase.h 中，还定义了一个用于轻引用计数的类模板，它里面包含了一个成员变量，用于标识对象的引用计数（见头文件 RefBase.h），代码如下：

```
00190:  private:
00191:      mutable volatile int32_t mCount;
00192:  };
```

成员函数 incStrong 和 decStrong 用于增加和减少计数。继承该类就可以为自己添加一个轻量级的单纯的引用计数功能，用于强指针对象。

1.4　强指针

在文件 StrongPointer.h（frameworks/native/include/utils 下）中，实现了强指针类模板。该模板实例化后只有一个成员变量，这个变量就是指向被强指针保护起来的类对象的指针，代码如下：

```
00108:      T* m_ptr;
00109:  };
```

重载了“*”和“->”操作符后的代码如下：

```
00091:      inline T&    operator* () const  { return *m_ptr; }
00092:      inline T*    operator-> () const { return m_ptr;  }
00093:      inline T*    get() const         { return m_ptr; }
```

那么，可以像使用指针那样使用强指针变量。如 MyClass（须继承 RefBase 或 LightRefBase 类）有成员变量 func()，则可以这样调用它：

```
mc1->func();或(mc1.get()->func());
```

而使用 get 函数（如 mc1->get();）将返回它包含的指针值。

1.4.1　强指针变量的初始化与生命周期

以上介绍的 m_ptr 指针由构造函数初始化。当不带任何参数时，赋值为 0，可以理解为未指向任何类对象或类对象为空。构造函数见下面的行 00065，代码如下：

```
00061: template <typename T>
00062: class sp
00063: {
00064: public:
00065:     inline sp() : m_ptr(0) { }
```

可以声明一个强指针变量，如：

```
sp<MyClass> mc0;
```

则得到一个模板实例化后的变量 mc0，它里面包含一个指向 MyClass 类对象的指针 m_ptr，其初始值为 0。也可以使用它的其他形式的构造函数，代码如下：

```
00067:     sp(T* other);//使用类对象的指针初始化，直接将other指向的对象的引用计数加1
00068:     sp(const sp<T>& other);//使用类对象的另一个强指针变量初始化，得到指针值后，将自己引用计数加1
00069:     template<typename U> sp(U* other);//使用其它类对象的指针赋值，得到指针值后，将自己引用加1
00070:     template<typename U> sp(const sp<U>& other);//使用其它类对象的强指针变
00071:                         //量other初始化:得到指针值赋给m_prt后，将自己增加1
```

对以上代码的解释如下：

（1）对于第一个构造函数 sp(T* other)，它使用类对象的指针初始化，将对象指针赋值给 m_ptr，然后自身引用计数加 1，如：

```
sp<MyClass> mc1(new MyClass);
```

这时，new MyClass 得到的内存地址赋值给 m_ptr，然后对该新创建出来的对象引用计数加 1。

（2）对于第二个构造函数 sp(const sp<T>& other)，它使用类对象的另一强指针变量进行初始化，如：

```
mc0(mc1);
```

这时强指针变量 mc1 中的指针值赋给 mc0 中的 m_ptr，它们指向的对象引用计数加 1，表示有多个地方引用了该对象。

（3）对于第三个构造函数 template<typename U> sp(U* other)，它使用另一个类的对象指针进行初始化，如（假设 MyClassParent 是 MyClass 的父类）：

```
MyClass* child= new MyClass;sp<MyClassParent> mcp1(child);
```

这时，mcp1 中的 m_ptr 为 MyClassParent 类型的指针，但指向其子类对象 MyClass。指针 m_ptr 指向的子类对象 child 引用计数加 1。

（4）对于第四个构造函数 template<typename U> sp(const sp<U>& other)，它使用另一个类的强指针变量进行初始化，如：

```
sp< MyClassParent>mcp2(mc1);
```

此处的 mc1 为（1）中的 MyClass 的强指针变量，其指针值赋给了 mcp2 中的 m_ptr，该指针指向的子类 MyClass 对象为引用计数加 1。

上面的构造函数初始化实际是得到一个强指针变量，它的生命周期如同普通的变量，如：

```
int a;
```

其位于程序的栈上，如在一个函数中或在一个大括号{}中，当函数结束或大括号结束后，其生命周期结束，此时强指针变量类的析构函数被调用，将引用计数减 1。因此，强指针变量在初始化时其指向的对象计数将被加 1；当其生命周期结束时，引用计数又减 1；若计数为 0，被指向的对象将被自动释放。这就实现了对对象的引用计数的自动增减和自动释放。