Android开发艺术探索

作者：任玉刚

目录

内容简介
序言
前言
第1章 Activity的生命周期和启动模式
1.1 Activity的生命周期全面分析
1.1.1 典型情况下的生命周期分析
1.1.2 异常情况下的生命周期分析
1.2 Activity的启动模式
1.2.1 Activity的LaunchMode
1.2.2 Activity的Flags
1.3 IntentFilter的匹配规则
第2章 IPC机制
2.1 Android IPC简介
2.2 Android中的多进程模式
2.2.1 开启多进程模式
2.2.2 多进程模式的运行机制
2.3 IPC基础概念介绍
2.3.1 Serializable接口
2.3.2 Parcelable接口
2.3.3 Binder
2.4 Android中的IPC方式
2.4.1 使用Bundle
2.4.2 使用文件共享
2.4.3 使用Messenger
2.4.4 使用AIDL
2.4.5 使用ContentProvider
2.4.6 使用Socket
2.5 Binder连接池
2.6 选用合适的IPC方式
第3章 View的事件体系
3.1 View基础知识
3.1.1 什么是View
3.1.2 View的位置参数
3.1.3 MotionEvent和TouchSlop
3.1.4 VelocityTracker、GestureDetector和Scroller
3.2 View的滑动
3.2.1 使用scrollTo/scrollBy
3.2.2 使用动画
3.2.3 改变布局参数
3.2.4 各种滑动方式的对比
3.3 弹性滑动
3.3.1 使用Scroller
3.3.2 通过动画
3.3.3 使用延时策略
3.4 View的事件分发机制
3.4.1 点击事件的传递规则
3.4.2 事件分发的源码解析
3.5 View的滑动冲突
3.5.1 常见的滑动冲突场景
3.5.2 滑动冲突的处理规则
3.5.3 滑动冲突的解决方式
第4章 View的工作原理
4.1 初识ViewRoot和DecorView
4.2 理解MeasureSpec
4.2.1 MeasureSpec
4.2.2 MeasureSpec和LayoutParams的对应关系
4.3 View的工作流程
4.3.1 measure过程
4.3.2 layout过程
4.3.3 draw过程
4.4 自定义View
4.4.1 自定义View的分类
4.4.2 自定义View须知
4.4.3 自定义View示例
4.4.4 自定义View的思想
第5章 理解RemoteViews
5.1 RemoteViews的应用
5.1.1 RemoteViews在通知栏上的应用
5.1.2 RemoteViews在桌面小部件上的应用
5.1.3 PendingIntent概述
5.2 RemoteViews的内部机制
5.3 RemoteViews的意义
第6章 Android的Drawable
6.1 Drawable简介
6.2 Drawable的分类
6.2.1 BitmapDrawable
6.2.2 ShapeDrawable
6.2.3 LayerDrawable
6.2.4 StateListDrawable
6.2.5 LevelListDrawable
6.2.6 TransitionDrawable
6.2.7 InsetDrawable
6.2.8 ScaleDrawable
6.2.9 ClipDrawable
6.3 自定义Drawable
第7章 Android动画深入分析
7.1 View动画
7.1.1 View动画的种类
7.1.2 自定义View动画
7.1.3 帧动画

7.2 View动画的特殊使用场景
7.2.1 LayoutAnimation
7.2.2 Activity的切换效果
7.3 属性动画
7.3.1 使用属性动画
7.3.2 理解插值器和估值器
7.3.3 属性动画的监听器
7.3.4 对任意属性做动画
7.3.5 属性动画的工作原理
7.4 使用动画的注意事项
第8章 理解Window和WindowManager
8.1 Window和WindowManager
8.2 Window的内部机制
8.2.1 Window的添加过程
8.2.2 Window的删除过程
8.2.3 Window的更新过程
8.3 Window的创建过程
8.3.1 Activity的Window创建过程
8.3.2 Dialog的Window创建过程
8.3.3 Toast的Window创建过程
第9章 四大组件的工作过程
9.1 四大组件的运行状态
9.2 Activity的工作过程
9.3 Service的工作过程
9.3.1 Service的启动过程
9.3.2 Service的绑定过程
9.4 BroadcastReceiver的工作过程
9.4.1 广播的注册过程
9.4.2 广播的发送和接收过程
9.5 ContentProvider的工作过程
第10章 Android的消息机制
10.1 Android的消息机制概述
10.2 Android的消息机制分析
10.2.1 ThreadLocal的工作原理
10.2.2 消息队列的工作原理
10.2.3 Looper的工作原理
10.2.4 Handler的工作原理
10.3 主线程的消息循环
第11章 Android的线程和线程池
11.1 主线程和子线程
11.2 Android中的线程形态
11.2.1 AsyncTask
11.2.2 AsyncTask的工作原理
11.2.3 HandlerThread
11.2.4 IntentService
11.3 Android中的线程池
11.3.1 ThreadPoolExecutor
11.3.2 线程池的分类
第12章 Bitmap的加载和Cache
12.1 Bitmap的高效加载
12.2 Android中的缓存策略
12.2.1 LruCache
12.2.2 DiskLruCache
12.2.3 ImageLoader的实现
12.3 ImageLoader的使用
12.3.1 照片墙效果
12.3.2 优化列表的卡顿现象
第13章 综合技术
13.1 使用CrashHandler来获取应用的crash信息
13.2 使用multidex来解决方法数越界
13.3 Android的动态加载技术
13.4 反编译初步
13.4.1 使用dex2jar和jd-gui反编译apk
13.4.2 使用apktool对apk进行二次打包
第14章 JNI和NDK编程
14.1 JNI的开发流程
14.2 NDK的开发流程
14.3 JNI的数据类型和类型签名
14.4 JNI调用Java方法的流程
第15章 Android性能优化
15.1 Android的性能优化方法
15.1.1 布局优化
15.1.2 绘制优化
15.1.3 内存泄露优化
15.1.4 响应速度优化和ANR日志分析
15.1.5 ListView和Bitmap优化
15.1.6 线程优化
15.1.7 一些性能优化建议
15.2 内存泄露分析之MAT工具
15.3 提高程序的可维护性
反侵权盗版声明

图书在版编目（CIP）数据

Android开发艺术探索/任玉刚著. —北京：电子工业出版社，2015.9

ISBN 978-7-121-26939-4

I. ①A… II. ①任… 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（2015）第189069号

责任编辑：陈晓猛

印　　刷：中国电影出版社印刷厂

装　　订：三河市华成印务有限公司

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：787×980　1/16

印　　张：32.75

字　　数：733千字

版　　次：2015年9月第1版

印　　次：2016年1月第6次印刷

印　　数：15001～20000册

定　　价：79.00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。

内容简介

本书是一本Android进阶类书籍，采用理论、源码和实践相结合的方式来阐述高水准的Android应用开发要点。本书从三个方面来组织内容。第一，介绍Android开发者不容易掌握的一些知识点；第二，结合Android源代码和应用层开发过程，融会贯通，介绍一些比较深入的知识点；第三，介绍一些核心技术和Android的性能优化思想。

本书侧重于Android知识的体系化和系统工作机制的分析，通过本书的学习可以极大地提高开发者的Android技术水平，从而更加高效地成为高级开发者。而对于高级开发者来说，仍然可以从本书的知识体系中获益。

序言

与玉刚共事两年，其对技术的热情和执著让人敬佩，其技术进步之快又让人惊叹。如今，他把所掌握的知识与经验成书出版，是一件大幸之事：于作者，此书是他的心血所成，可喜可贺；于读者，可解“工作视野”之困与“百思不得其解”之惑，或许有“啊哈，原来如此”之效，又或许有“技能+1”之得意一笑。

玉刚拥有丰富的Android开发经验，对Android开发的很多知识点都有深入研究，我相信此书定能为读者带来惊喜。书的内容，大抵有如下几方面：基础知识点之深入理解（例如，Activity的生命周期和启动模式、Android的消息机制分析、View的事件体系、View的工作原理等章节）；不常见知识点的分析（例如，IPC机制、理解Window和WindowManager等章节）；工程实践中的经验（例如，综合技术、Android性能优化等章节）。因此，此书读者需要有一定的Android开发基础和工程经验，否则读起来会比较吃力或者感觉云里雾里。对于想成长为高级或者资深Android研发的工程师，书中的知识点都是需要掌握的。

最后，希望读者能够从此书获益，接触到一些工作中未曾了解或者思考的知识点。更进一步，希望读者能够活学活用，并学习此书背后的钻研精神。

涂勇策
百度手机卫士　资深工程师

前言

从目前的形势来看，Android开发相当火热，但是高级Android开发人才却比较少，当然在国内，不仅仅是Android，其他技术岗位同样面临这个问题。试想下，如果有一本书能够切实有效地提高开发者的技术水平，那该多好啊！纵观市场上的Android书籍，很多都是入门类书籍，还有一些Android源码分析、系统移植、驱动开发、逆向工程等系统底层类书籍。入门类书籍是目前图书市场中的中坚力量，它们在帮助开发者入门的过程中起到了非常重要的作用，但开发者若想进一步提高技术水平，还需要阅读更深入的书籍。底层书籍包括源码分析、驱动开发、逆向工程等书籍，它们从底层或者某一个特殊的角度来深入地分析Android，这是很值得称赞和学习的，通过这些书可以极大地提高开发者底层或者相关领域的技术水平。但美中不足的是，系统底层书籍比较偏理论，部分开发者阅读起来可能会有点晦涩难懂。更重要的一点，由于它们往往侧重原理和底层机制，导致它们不能直接为应用层开发服务，毕竟绝大多数Android开发岗位都是应用层开发。由于阅读底层类书籍一般只能够加深对底层的认识，而在应用层开发中，还是不能形成直接有效的战斗力，这中间是需要转化过程的。但是，由于部分开发者缺乏相应的技术功底，导致无法完成这个转化过程。

可以发现，目前市场上既能够极大地提高开发者的应用层技术经验，又能够将上层和系统底层的运行机制结合起来的书籍还是比较少的。对企业来说，在业务上有很强的技术能力，同时对Android底层也有一定理解的开发人员，是企业比较青睐的技术高手。为了完成这一愿望，笔者写了这本书。通过对本书的深入学习，开发者既能够极大地提高应用层的开发能力，又能够对Android系统的运行机制有一定的理解，但如果要深入理解Android的底层机制，仍然需要查看相关源码分析的书籍。

本书适合各类开发者阅读，对于初、中级开发者来说，可以通过本书更加高效地达到高级开发者的技术水平。而对于高级开发者，仍然可以从本书的知识体系中获益。本书的书名之所以采用艺术这个词，这是因为在笔者眼中，代码写到极致就是一种艺术。

本文内容

本书共15章，所讲述的内容均基于Android 5.0系统。

第1章介绍Activity的生命周期和启动模式以及IntentFilter的匹配规则。

第2章介绍Android中常见的IPC机制，多进程的运行模式和一些常见的进程间通信方式，包括Messenger、AIDL、Binder以及ContentProvider等，同时还介绍Binder连接池的概念。

第3章介绍View的事件体系，并对View的基础知识、滑动以及弹性滑动做详细的介绍，同时还深入分析滑动冲突的原因以及解决方法。

第4章介绍View的工作原理，首先介绍ViewRoot、DecorView、MeasureSpec等View相关的底层概念，然后详细分析View的测量、布局和绘制三大流程，最后介绍自定义View的分类以及实现思想。

第5章讲述一个不常见的概念RemoteViews，分别描述RemoteViews在通知栏和桌面小部件中的使用场景，同时还详细介绍PendingIntent，最后深入分析RemoteViews的内部机制并探索性地指出RemoteViews在Android中存在的意义。

第6章对Android的Drawable做一个全面性的介绍，除此之外还讲解自定义Drawable的方法。

第7章对Android中的动画做一个全面深入的分析，包含View动画和属性动画。

第8章讲述Window和WindowManager，首先分析Window的内部工作原理，包括Window的添加、更新和删除，其次分析Activity、Dialog等类型的Window对象的创建过程。

第9章深入分析Android中四大组件的工作过程，主要包括四大组件的运行状态以及它们主要的工作过程，比如启动、绑定、广播的发送和接收等。

第10章深入分析Android的消息机制，其中涉及的概念有Handler、Looper、MessageQueue以及ThreadLocal，此外还分析主线程的消息循环模型。

第11章讲述Android的线程和线程池，首先介绍AsyncTask、HandlerThread、IntentService以及ThreadPoolExecutor的使用方法，然后分析它们的工作原理。

第12章讲述的主题是Bitmap的加载和缓存机制，首先讲述高效加载图片的方式，接着介绍LruCache和DiskLruCache的使用方法，最后通过一个ImageLoader的实例来将它们综合起来。

第13章是综合技术，讲述一些很重要但是不太常见的技术方案，它们是CrashHandler、multidex、插件化以及反编译。

第14章的主题是JNI和NDK编程，介绍使用JNI和Android NDK编程的方法。

第15章介绍Android的性能优化方法，比如常见的布局优化、绘制优化、内存泄露优化等，除此之外还介绍分析ANR和内存泄露的方法，最后探讨如何提高程序的可维护性这一话题。

通过这15章的学习，可以让初、中级开发者的技术水平和把控能力提升一个档次，最终成为高级开发者。

本书特色

本书定位为进阶类图书，不会对一些基础知识从头说起，或者说每一章节都不涵盖各种入门知识，但是在向高级知识点过渡的时候，会稍微提及一下基础知识从而做到平滑过渡。开发者在掌握入门知识以后，通过本书可以极大地提高应用层开发的技术水平，同时还可以理解一定的Android底层运行机制，并且能够将它们进行升华从而更好地为应用层开发服务。除了这些，开发者还可以掌握一些核心技术和性能优化思想，本书涉及的知识，都是一个合格的高级工程师所必须掌握的。简单地说，本书的目的就是让初、中级开发者更有针对性地掌握高级工程师所应该掌握的技术，能够让初、中级开发者按照正确的道路快速地成长为高级工程师。

致谢

感谢本书的策划编辑陈晓猛，他的高效率是本书得以及时出版的一个重要原因；感谢我的妻子对我写书的支持，接近1年的写书时光是她一直陪伴在我身边；感谢百度手机卫士这款产品，它是本书的技术源泉；感谢和我一起奋斗的同事们，和你们在一起工作的时光，我不仅提高了技术水平而且还真正感受到了一种融洽的工作氛围；还要感谢所有关注我的朋友们，你们的鼓励和认可是我前进的动力。

由于技术水平有限，书中难免会有错误，欢迎大家向我反馈：singwhatiwanna@ gmail.com，也可以关注我的CSDN博客，我会定期在上面发布本书的勘误信息。

本书互动地址

CSDN博客：http://blog.csdn.net/singwhatiwanna

Github：https://github.com/singwhatiwanna

QQ交流群：481798332

微信公众号：Android开发艺术探索

书中源码下载地址：

https://github.com/singwhatiwanna/android-art-res

或者

www.broadview.com.cn/26939

任玉刚
2015年6月于北京

第1章　Activity的生命周期和启动模式

作为本书的第1章，本章主要介绍Activity相关的一些内容。Activity作为四大组件之首，是使用最为频繁的一种组件，中文直接翻译为“活动”，但是笔者认为这种翻译有些生硬，如果翻译成界面就会更好理解。正常情况下，除了Window、Dialog和Toast，我们能见到的界面的确只有Activity。Activity是如此重要，以至于本书开篇就不得不讲到它。当然，由于本书的定位为进阶书，所以不会介绍如何启动Activity这类入门知识，本章的侧重点是Activity在使用过程中的一些不容易搞清楚的概念，主要包括生命周期和启动模式以及IntentFilter的匹配规则分析。其中Activity在异常情况下的生命周期是十分微妙的，至于Activity的启动模式和形形色色的Flags更是让初学者摸不到头脑，就连隐式启动Activity中也有着复杂的Intent匹配过程，不过不用担心，本章接下来将一一解开这些疑难问题的神秘面纱。

1.1　Activity的生命周期全面分析

本节将Activity的生命周期分为两部分内容，一部分是典型情况下的生命周期，另一部分是异常情况下的生命周期。所谓典型情况下的生命周期，是指在有用户参与的情况下，Activity所经过的生命周期的改变；而异常情况下的生命周期是指Activity被系统回收或者由于当前设备的Configuration发生改变从而导致Activity被销毁重建，异常情况下的生命周期的关注点和典型情况下略有不同。

1.1.1　典型情况下的生命周期分析

在正常情况下，Activity会经历如下生命周期。

（1）onCreate：表示Activity正在被创建，这是生命周期的第一个方法。在这个方法中，我们可以做一些初始化工作，比如调用setContentView去加载界面布局资源、初始化Activity所需数据等。

（2）onRestart：表示Activity正在重新启动。一般情况下，当当前Activity从不可见重新变为可见状态时，onRestart就会被调用。这种情形一般是用户行为所导致的，比如用户按Home键切换到桌面或者用户打开了一个新的Activity，这时当前的Activity就会暂停，也就是onPause和onStop被执行了，接着用户又回到了这个Activity，就会出现这种情况。

（3）onStart：表示Activity正在被启动，即将开始，这时Activity已经可见了，但是还没有出现在前台，还无法和用户交互。这个时候其实可以理解为Activity已经显示出来了，但是我们还看不到。

（4）onResume：表示Activity已经可见了，并且出现在前台并开始活动。要注意这个和onStart的对比，onStart和onResume都表示Activity已经可见，但是onStart的时候Activity还在后台，onResume的时候Activity才显示到前台。

（5）onPause：表示Activity正在停止，正常情况下，紧接着onStop就会被调用。在特殊情况下，如果这个时候快速地再回到当前Activity，那么onResume会被调用。笔者的理解是，这种情况属于极端情况，用户操作很难重现这一场景。此时可以做一些存储数据、停止动画等工作，但是注意不能太耗时，因为这会影响到新Activity的显示，onPause必须先执行完，新Activity的onResume才会执行。

（6）onStop：表示Activity即将停止，可以做一些稍微重量级的回收工作，同样不能太耗时。

（7）onDestroy：表示Activity即将被销毁，这是Activity生命周期中的最后一个回调，在这里，我们可以做一些回收工作和最终的资源释放。

正常情况下，Activity的常用生命周期就只有上面7个，图1-1更详细地描述了Activity各种生命周期的切换过程。

图1-1　Activity生命周期的切换过程

针对图1-1，这里再附加一下具体说明，分如下几种情况。

（1）针对一个特定的Activity，第一次启动，回调如下：onCreate -> onStart -> onResume。

（2）当用户打开新的Activity或者切换到桌面的时候，回调如下：onPause -> onStop。这里有一种特殊情况，如果新Activity采用了透明主题，那么当前Activity不会回调onStop。

（3）当用户再次回到原Activity时，回调如下：onRestart -> onStart -> onResume。

（4）当用户按back键回退时，回调如下：onPause -> onStop -> onDestroy。

（5）当Activity被系统回收后再次打开，生命周期方法回调过程和（1）一样，注意只是生命周期方法一样，不代表所有过程都一样，这个问题在下一节会详细说明。

（6）从整个生命周期来说，onCreate和onDestroy是配对的，分别标识着Activity的创建和销毁，并且只可能有一次调用。从Activity是否可见来说，onStart和onStop是配对的，随着用户的操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次；从Activity是否在前台来说，onResume和onPause是配对的，随着用户操作或者设备屏幕的点亮和熄灭，这两个方法可能被调用多次。

这里提出2个问题，不知道大家是否清楚。

问题1：onStart和onResume、onPause和onStop从描述上来看差不多，对我们来说有什么实质的不同呢？

问题2：假设当前Activity为A，如果这时用户打开一个新Activity B，那么B的onResume和A的onPause哪个先执行呢？

先说第一个问题，从实际使用过程来说，onStart和onResume、onPause和onStop看起来的确差不多，甚至我们可以只保留其中一对，比如只保留onStart和onStop。既然如此，那为什么Android系统还要提供看起来重复的接口呢？根据上面的分析，我们知道，这两个配对的回调分别表示不同的意义，onStart和onStop是从Activity是否可见这个角度来回调的，而onResume和onPause是从Activity是否位于前台这个角度来回调的，除了这种区别，在实际使用中没有其他明显区别。

第二个问题可以从Android源码里得到解释。关于Activity的工作原理在本书后续章节会进行介绍，这里我们先大概了解即可。从Activity的启动过程来看，我们来看一下系统源码。Activity的启动过程的源码相当复杂，涉及Instrumentation、ActivityThread和ActivityManagerService（下面简称AMS）。这里不详细分析这一过程，简单理解，启动Activity的请求会由Instrumentation来处理，然后它通过Binder向AMS发请求，AMS内部维护着一个ActivityStack并负责栈内的Activity的状态同步，AMS通过ActivityThread去同步Activity的状态从而完成生命周期方法的调用。在ActivityStack中的resumeTopActivity-InnerLocked方法中，有这么一段代码：

```
// We need to start pausing the current activity so the top one
// can be resumed...
boolean dontWaitForPause = (next.info.flags&ActivityInfo.FLAG_RESUME_ WHILE_PAUSING) != 0;
boolean pausing = mStackSupervisor.pauseBackStacks(userLeaving,true,dontWaitForPause);
if (mResumedActivity != null) {
    pausing |= startPausingLocked(userLeaving,false,true,dontWait-ForPause);
    if (DEBUG_STATES) Slog.d(TAG,"resumeTopActivityLocked: Pausing " + mResumedActivity);
}
```

从上述代码可以看出，在新Activity启动之前，栈顶的Activity需要先onPause后，新Activity才能启动。最终，在ActivityStackSupervisor中的realStartActivityLocked方法会调用如下代码。

```
app.thread.scheduleLaunchActivity(new Intent(r.intent),r.appToken,
            System.identityHashCode(r),r.info,new Configuration(mService.mConfiguration),
            r.compat,r.task.voiceInteractor,app.repProcState,r.icicle,r.persistentState,
```

```
            results,newIntents,!andResume,mService.isNextTransition-Forward(),
            profilerInfo);
```

我们知道，这个app.thread的类型是IApplicationThread，而IApplicationThread的具体实现是ActivityThread中的ApplicationThread。所以，这段代码实际上调到了ActivityThread的中，即ApplicationThread的scheduleLaunchActivity方法，而scheduleLaunchActivity方法最终会完成新Activity的onCreate、onStart、onResume的调用过程。因此，可以得出结论，是旧Activity先onPause，然后新Activity再启动。

至于ApplicationThread的scheduleLaunchActivity方法为什么会完成新Activity的onCreate、onStart、onResume的调用过程，请看下面的代码。scheduleLaunchActivity最终会调用如下方法，而如下方法的确会完成onCreate、onStart、onResume的调用过程。

源码：ActivityThread#handleLaunchActivity

```
private void handleLaunchActivity(ActivityClientRecord r,Intent custom- Intent) {
    // If we are getting ready to gc after going to the background,well
    // we are back active so skip it.
    unscheduleGcIdler();
    mSomeActivitiesChanged = true;
    if (r.profilerInfo != null) {
            mProfiler.setProfiler(r.profilerInfo);
            mProfiler.startProfiling();
    }
    // Make sure we are running with the most recent config.
    handleConfigurationChanged(null,null);
    if (localLOGV) Slog.v(
            TAG,"Handling launch of " + r);
    //这里新Activity被创建出来，其onCreate和onStart会被调用
Activity a = performLaunchActivity(r,customIntent);
    if (a != null) {
            r.createdConfig = new Configuration(mConfiguration);
            Bundle oldState = r.state;
            //这里新Activity的onResume会被调用
            handleResumeActivity(r.token,false,r.isForward,
                            !r.activity.mFinished && !r.startsNotResumed);
    //省略
}
```

从上面的分析可以看出，当新启动一个Activity的时候，旧Activity的onPause会先执行，然后才会启动新的Activity。到底是不是这样呢？我们写个例子验证一下，如下是2个Activity的代码，在MainActivity中单击按钮可以跳转到SecondActivity，同时为了分析我们的问题，在生命周期方法中打印出了日志，通过日志我们就能看出它们的调用顺序。

代码：MainActivity.java

```
public class MainActivity extends Activity {
     private static final String TAG = "MainActivity";
     //省略
     @Override
     protected void onPause() {
         super.onPause();
         Log.d(TAG,"onPause");
     }
     @Override
     protected void onStop() {
         super.onStop();
         Log.d(TAG,"onStop");
     }
}
```

代码：SecondActivity.java

```
public class SecondActivity extends Activity {
     private static final String TAG = "SecondActivity";
     @Override
     protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_second);
         Log.d(TAG,"onCreate");
     }
     @Override
     protected void onStart() {
         super.onStart();
         Log.d(TAG,"onStart");
     }
     @Override
     protected void onResume() {
         super.onResume();
         Log.d(TAG,"onResume");
     }
}
```

我们来看一下log，是不是和我们上面分析的一样，如图1-2所示。

图1-2　Activity生命周期方法的回调顺序

通过图1-2可以发现，旧Activity的onPause先调用，然后新Activity才启动，这也证实了我们上面的分析过程。也许有人会问，你只是分析了Android5.0的源码，你怎么知道所有版本的源码都是相同逻辑呢？关于这个问题，我们的确不大可能把所有版本的源码都分析一遍，但是作为Android运行过程的基本机制，随着版本的更新并不会有大的调整，因为Android系统也需要兼容性，不能说在不同版本上同一个运行机制有着截然不同的表现。关于这一点我们需要把握一个度，就是对于Android运行的基本机制在不同Android版本上具有延续性。从另一个角度来说，Android官方文档对onPause的解释有这么一句：不能在onPause中做重量级的操作，因为必须onPause执行完成以后新Activity才能Resume，从这一点也能间接证明我们的结论。通过分析这个问题，我们知道onPause和onStop都不能执行耗时的操作，尤其是onPause，这也意味着，我们应当尽量在onStop中做操作，从而使得新Activity尽快显示出来并切换到前台。

1.1.2　异常情况下的生命周期分析

上一节我们分析了典型情况下Activity的生命周期，本节我们接着分析Activity在异常情况下的生命周期。我们知道，Activity除了受用户操作所导致的正常的生命周期方法调度，还有一些异常情况，比如当资源相关的系统配置发生改变以及系统内存不足时，Activity就可能被杀死。下面我们具体分析这两种情况。

1. 情况1：资源相关的系统配置发生改变导致Activity被杀死并重新创建

理解这个问题，我们首先要对系统的资源加载机制有一定了解，这里不详细分析系统的资源加载机制，只是简单说明一下。拿最简单的图片来说，当我们把一张图片放在drawable目录后，就可以通过Resources去获取这张图片。同时为了兼容不同的设备，我们可能还需要在其他一些目录放置不同的图片，比如drawable-mdpi、drawable-hdpi、drawable-land等。这样，当应用程序启动时，系统就会根据当前设备的情况去加载合适的Resources资源，比如说横屏手机和竖屏手机会拿到两张不同的图片（设定了landscape或者portrait状态下的图片）。比如说当前Activity处于竖屏状态，如果突然旋转屏幕，由于系统配置发生了改变，在默认情况下，Activity就会被销毁并且重新创建，当然我们也可以阻止系统重新创建我们的Activity。

在默认情况下，如果我们的Activity不做特殊处理，那么当系统配置发生改变后，Activity就会被销毁并重新创建，其生命周期如图1-3所示。

图1-3　异常情况下Activity的重建过程

当系统配置发生改变后，Activity会被销毁，其onPause、onStop、onDestroy均会被调用，同时由于Activity是在异常情况下终止的，系统会调用onSaveInstanceState来保存当前Activity的状态。这个方法的调用时机是在onStop之前，它和onPause没有既定的时序关系，它既可能在onPause之前调用，也可能在onPause之后调用。需要强调的一点是，这个方法只会出现在Activity被异常终止的情况下，正常情况下系统不会回调这个方法。当Activity被重新创建后，系统会调用onRestoreInstanceState，并且把Activity销毁时onSaveInstanceState方法所保存的Bundle对象作为参数同时传递给onRestoreInstanceState和onCreate方法。因此，我们可以通过onRestoreInstanceState和onCreate方法来判断Activity是否被重建了，如果被重建了，那么我们就可以取出之前保存的数据并恢复，从时序上来说，onRestoreInstanceState的调用时机在onStart之后。

同时，我们要知道，在onSaveInstanceState和onRestoreInstanceState方法中，系统自动为我们做了一定的恢复工作。当Activity在异常情况下需要重新创建时，系统会默认为我们保存当前Activity的视图结构，并且在Activity重启后为我们恢复这些数据，比如文本框中用户输入的数据、ListView滚动的位置等，这些View相关的状态系统都能够默认为我们恢复。具体针对某一个特定的View系统能为我们恢复哪些数据，我们可以查看View的源码。和Activity一样，每个View都有onSaveInstanceState和onRestoreInstanceState这两个方法，看一下它们的具体实现，就能知道系统能够自动为每个View恢复哪些数据。

关于保存和恢复View层次结构，系统的工作流程是这样的：首先Activity被意外终止时，Activity会调用onSaveInstanceState去保存数据，然后Activity会委托Window去保存数据，接着Window再委托它上面的顶级容器去保存数据。顶层容器是一个ViewGroup，一般来说它很可能是DecorView。最后顶层容器再去一一通知它的子元素来保存数据，这样整个数据保存过程就完成了。可以发现，这是一种典型的委托思想，上层委托下层、父容器委托子元素去处理一件事情，这种思想在Android中有很多应用，比如View的绘制过程、事件分发等都是采用类似的思想。至于数据恢复过程也是类似的，这里就不再重复介绍了。接下来举个例子，拿TextView来说，我们分析一下它到底保存了哪些数据。

源码：TextView#onSaveInstanceState

```
@Override
public Parcelable onSaveInstanceState() {
    Parcelable superState = super.onSaveInstanceState();
    // Save state if we are forced to
    boolean save = mFreezesText;
    int start = 0;
    int end = 0;
    if (mText != null) {
            start = getSelectionStart();
            end = getSelectionEnd();
            if (start => 0 || end => 0) {
                    // Or save state if there is a selection
                    save = true;
            }
    }
    if (save) {
            SavedState ss = new SavedState(superState);
            // XXX Should also save the current scroll position!
            ss.selStart = start;
            ss.selEnd = end;
            if (mText instanceof Spanned) {
                    Spannable sp = new SpannableStringBuilder(mText);
                    if (mEditor != null) {
                            removeMisspelledSpans(sp);
                            sp.removeSpan(mEditor.mSuggestionRangeSpan);
                    }
                    ss.text = sp;
            } else {
                    ss.text = mText.toString();
            }
            if (isFocused() && start => 0 && end => 0) {
                    ss.frozenWithFocus = true;
            }
            ss.error = getError();
            return ss;
    }
    return superState;
}
```

从上述源码可以很容易看出，TextView保存了自己的文本选中状态和文本内容，并且通过查看其onRestoreInstanceState方法的源码，可以发现它的确恢复了这些数据，具体源码就不再贴出了，读者可以去看看源码。下面我们看一个实际的例子，来对比一下Activity正常终止和异常终止的不同，同时验证系统的数据恢复能力。为了方便，我们选择旋转屏幕来异常终止Activity，如图1-4所示。

图1-4　Activity旋转屏幕后数据的保存和恢复

通过图1-4可以看出，在我们选择屏幕以后，Activity被销毁后重新创建，我们输入的文本“这是测试文本”被正确地还原，这说明系统的确能够自动地做一些View层次结构方面的数据存储和恢复。下面再用一个例子，来验证我们自己做数据存储和恢复的情况，代码如下：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    if (savedInstanceState != null) {
            String test = savedInstanceState.getString("extra_test");
            Log.d(TAG,"[onCreate]restore extra_test:" + test);
    }
}
@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);
    Log.d(TAG,"onSaveInstanceState");
    outState.putString("extra_test","test");
}
@Override
protected void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    String test = savedInstanceState.getString("extra_test");
    Log.d(TAG,"[onRestoreInstanceState]restore extra_test:" + test);
}
```

上面的代码很简单，首先我们在onSaveInstanceState中存储一个字符串，然后当Activity被销毁并重新创建后，我们再去获取之前存储的字符串。接收的位置可以选择onRestoreInstanceState或者onCreate，二者的区别是：onRestoreInstanceState一旦被调用，其参数Bundle savedInstanceState一定是有值的，我们不用额外地判断是否为空；但是onCreate不行，onCreate如果是正常启动的话，其参数Bundle savedInstanceState为null，所以必须要额外判断。这两个方法我们选择任意一个都可以进行数据恢复，但是官方文档的建议是采用onRestoreInstanceState去恢复数据。下面我们看一下运行的日志，如图1-5所示。

图1-5　系统日志

如图1-5所示，Activity被销毁了以后调用了onSaveInstanceState来保存数据，重新创建以后在onCreate和onRestoreInstanceState中都能够正确地恢复我们之前存储的字符串。这个