Android 开发艺术探索

任玉刚　著

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

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。

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：2015 年 11 月第 4 次印刷
印　　数：8001～11000 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。

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

目 录

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。

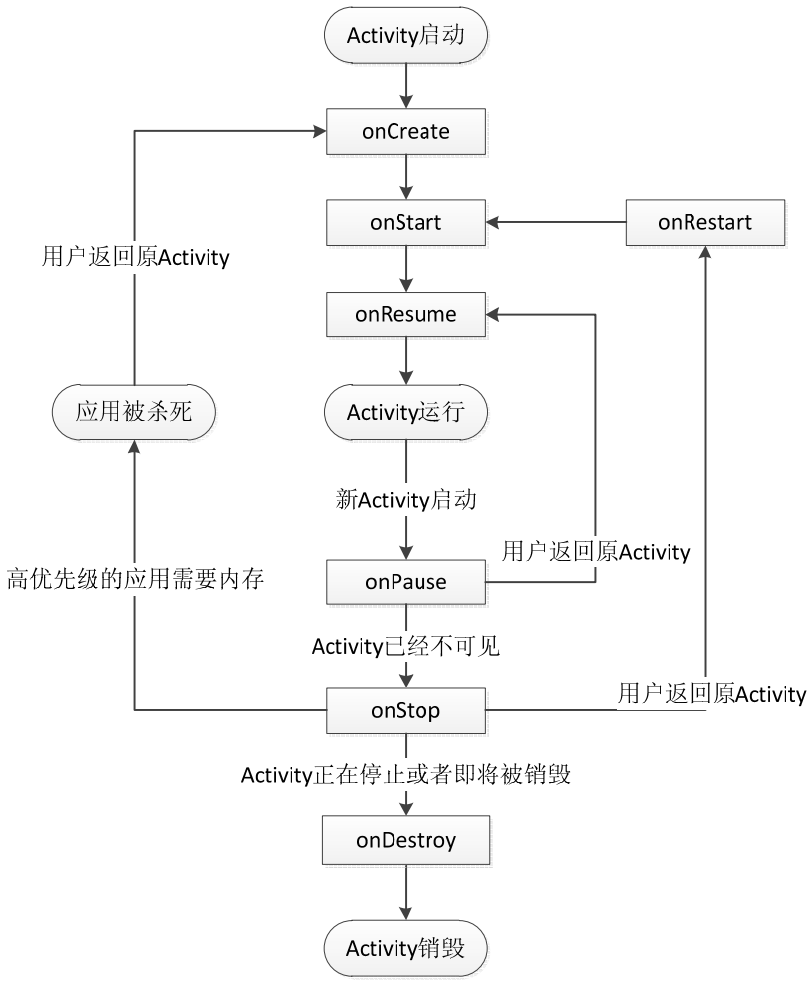

图 1-1　Activity 生命周期的切换过程

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。

Level	Time	PID	TID	Application	Tag	Text
D	02-01 01:37:33.051	724	724	com.ryg.chapter_1	MainActivity	onPause
D	02-01 01:37:33.111	724	724	com.ryg.chapter_1	SecondActivity	onCreate
D	02-01 01:37:33.111	724	724	com.ryg.chapter_1	SecondActivity	onStart
D	02-01 01:37:33.111	724	724	com.ryg.chapter_1	SecondActivity	onResume
D	02-01 01:37:33.431	724	724	com.ryg.chapter_1	MainActivity	onStop

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。

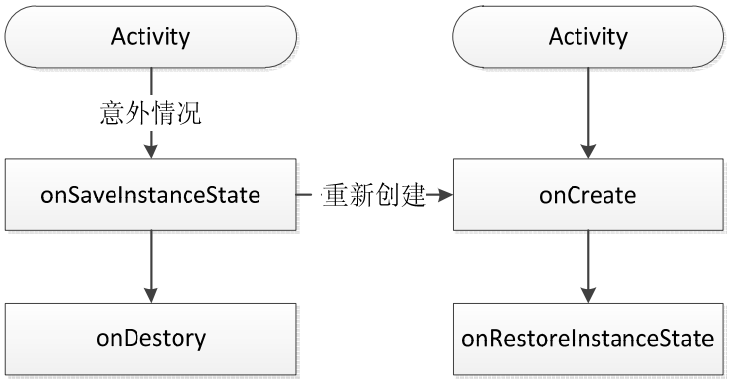

图 1-3 异常情况下 Activity 的重建过程

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，当我们旋转屏幕以后，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。

Level	PID	TID	Application	Tag	Text
D	8534	8534	com.ryg.chapter_1	MainActivity	onPause
D	8534	8534	com.ryg.chapter_1	MainActivity	onSaveInstanceState
D	8534	8534	com.ryg.chapter_1	MainActivity	onStop
D	8534	8534	com.ryg.chapter_1	MainActivity	onDestroy
D	8534	8534	com.ryg.chapter_1	MainActivity	[onCreate]restore extra_test:test
D	8534	8534	com.ryg.chapter_1	MainActivity	[onRestoreInstanceState]restore extra_test:test

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。这个例子很好地证明了上面我们的分析结论。针对 onSaveInstanceState 方法还有一点需要说明，那就是系统只会在 Activity 即将被销毁并且有机会重新显示的情况下才会去调用它。考虑这么一种情况，当 Activity 正常销毁的时候，系统不会调用 onSaveInstanceState，因为被销毁的 Activity 不可能再次被显示。这句话不好理解，但是我们可以对比一下旋转屏幕所造成的 Activity 异常销毁，这个过程和正常停止 Activity 是不一样的，因为旋转屏幕后，Activity 被销毁的同时会立刻创建新的 Activity 实例，这个时候 Activity 有机会再次立刻展示，所以系统要进行数据存储。这里可以简单地这么理解，系统只在 Activity 异常终止的时候才会调用 onSaveInstanceState 和 onRestoreInstanceState 来存储和恢复数据，其他情况不会触发这个过程。

2．情况 2：资源内存不足导致低优先级的 Activity 被杀死

这种情况我们不好模拟，但是其数据存储和恢复过程和情况 1 完全一致。这里我们描述一下 Activity 的优先级情况。Activity 按照优先级从高到低，可以分为如下三种：

（1）前台 Activity——正在和用户交互的 Activity，优先级最高。

（2）可见但非前台 Activity——比如 Activity 中弹出了一个对话框，导致 Activity 可见但是位于后台无法和用户直接交互。

（3）后台 Activity——已经被暂停的 Activity，比如执行了 onStop，优先级最低。

当系统内存不足时，系统就会按照上述优先级去杀死目标 Activity 所在的进程，并在后续通过 onSaveInstanceState 和 onRestoreInstanceState 来存储和恢复数数据。如果一个进程中没有四大组件在执行，那么这个进程将很快被系统杀死，因此，一些后台工作不适合脱离四大组件而独自运行在后台中，这样进程很容易被杀死。比较好的方法是将后台工作放入 Service 中从而保证进程有一定的优先级，这样就不会轻易地被系统杀死。

上面分析了系统的数据存储和恢复机制，我们知道，当系统配置发生改变后，Activity会被重新创建，那么有没有办法不重新创建呢？答案是有的，接下来我们就来分析这个问题。系统配置中有很多内容，如果当某项内容发生改变后，我们不想系统重新创建 Activity，可以给 Activity 指定 configChanges 属性。比如不想让 Activity 在屏幕旋转的时候重新创建，就可以给 configChanges 属性添加 orientation 这个值，如下所示。

```
android:configChanges="orientation"
```

如果我们想指定多个值，可以用“|”连接起来，比如 android:configChanges="orientation|keyboardHidden"。系统配置中所含的项目是非常多的，下面介绍每个项目的含义，如表 1-1 所示。

表 1-1　configChanges 的项目和含义

项　　目	含　　义
mcc	SIM 卡唯一标识 IMSI（国际移动用户识别码）中的国家代码，由三位数字组成，中国为 460。此项标识 mcc 代码发生了改变
mnc	SIM 卡唯一标识 IMSI（国际移动用户识别码）中的运营商代码，由两位数字组成，中国移动 TD 系统为 00，中国联通为 01，中国电信为 03。此项标识 mnc 发生改变
locale	设备的本地位置发生了改变，一般指切换了系统语言
touchscreen	触摸屏发生了改变，这个很费解，正常情况下无法发生，可以忽略它
keyboard	键盘类型发生了改变，比如用户使用了外插键盘
keyboardHidden	键盘的可访问性发生了改变，比如用户调出了键盘
navigation	系统导航方式发生了改变，比如采用了轨迹球导航，这个有点费解，很难发生，可以忽略它
screenLayout	屏幕布局发生了改变，很可能是用户激活了另外一个显示设备
fontScale	系统字体缩放比例发生了改变，比如用户选择了一个新字号
uiMode	用户界面模式发生了改变，比如是否开启了夜间模式（API 8 新添加）
orientation	屏幕方向发生了改变，这个是最常用的，比如旋转了手机屏幕
screenSize	当屏幕的尺寸信息发生了改变，当旋转设备屏幕时，屏幕尺寸会发生变化，这个选项比较特殊，它和编译选项有关，当编译选项中的 minSdkVersion 和 targetSdkVersion 均低于 13 时，此选项不会导致 Activity 重启，否则会导致 Activity 重启（API 13 新添加）
smallestScreenSize	设备的物理屏幕尺寸发生改变，这个项目和屏幕的方向没关系，仅仅表示在实际的物理屏幕的尺寸改变的时候发生，比如用户切换到了外部的显示设备，这个选项和 screenSize 一样，当编译选项中的 minSdkVersion 和 targetSdkVersion 均低于 13 时，此选项不会导致 Activity 重启，否则会导致 Activity 重启（API 13 新添加）
layoutDirection	当布局方向发生变化，这个属性用的比较少，正常情况下无须修改布局的 layoutDirection 属性（API 17 新添加）

从表 1-1 可以知道，如果我们没有在 Activity 的 configChanges 属性中指定该选项的话，当配置发生改变后就会导致 Activity 重新创建。上面表格中的项目很多，但是我们常用的只有 locale、orientation 和 keyboardHidden 这三个选项，其他很少使用。需要注意的是 screenSize 和 smallestScreenSize，它们两个比较特殊，它们的行为和编译选项有关，但和运行环境无关。下面我们再看一个 demo，看看当我们指定了 configChanges 属性后，Activity 是否真的不会重新创建了。我们所要修改的代码很简单，只需要在 AndroidMenifest.xml 中加入 Activity 的声明即可，代码如下：

```
<uses-sdk
    android:minSdkVersion="8"
    android:targetSdkVersion="19" />
<activity
    android:name="com.ryg.chapter_1.MainActivity"
    android:configChanges="orientation|screenSize"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>

@Override
public void onConfigurationChanged(Configuration newConfig) {
    super.onConfigurationChanged(newConfig);
    Log.d(TAG, "onConfigurationChanged, newOrientation:" + newConfig.
    orientation);
}
```

需要说明的是，由于编译时笔者指定的 minSdkVersion 和 targetSdkVersion 有一个大于 13，所以为了防止旋转屏幕时 Activity 重启，除了 orientation，我们还要加上 screenSize，原因在上面的表格里已经说明了。其他代码还是不变，运行后看看 log，如图 1-6 所示。

Level	PID	TID	Application	Tag	Text
D	3565	3565	com.ryg.chapter_1	MainActivity	onConfigurationChanged, newOrientation:2
D	3565	3565	com.ryg.chapter_1	MainActivity	onConfigurationChanged, newOrientation:1
D	3565	3565	com.ryg.chapter_1	MainActivity	onConfigurationChanged, newOrientation:2

图 1-6　系统日志

由上面的日志可见，Activity 的确没有重新创建，并且也没有调用 onSaveInstanceState 和 onRestoreInstanceState 来存储和恢复数据，取而代之的是系统调用了 Activity 的 onConfigurationChanged 方法，这个时候我们就可以做一些自己的特殊处理了。

1.2 Activity 的启动模式

上一节介绍了 Activity 在标准情况下和异常情况下的生命周期，我们对 Activity 的生命周期应该有了深入的了解。除了 Activity 的生命周期外，Activity 的启动模式也是一个难点，原因是形形色色的启动模式和标志位实在是太容易被混淆了，但是 Activity 作为四大组件之首，它的的确确非常重要，有时候为了满足项目的特殊需求，就必须使用 Activity 的启动模式，所以我们必须要搞清楚它的启动模式和标志位，本节将会一一介绍。

1.2.1 Activity 的 LaunchMode

首先说一下 Activity 为什么需要启动模式。我们知道，在默认情况下，当我们多次启动同一个 Activity 的时候，系统会创建多个实例并把它们一一放入任务栈中，当我们单击 back 键，会发现这些 Activity 会一一回退。任务栈是一种“后进先出”的栈结构，这个比较好理解，每按一下 back 键就会有一个 Activity 出栈，直到栈空为止，当栈中无任何 Activity 的时候，系统就会回收这个任务栈。关于任务栈的系统工作原理，这里暂时不做说明，在后续章节会专门介绍任务栈。知道了 Activity 的默认启动模式以后，我们可能就会发现一个问题：多次启动同一个 Activity，系统重复创建多个实例，这样不是很傻吗？这样的确有点傻，Android 在设计的时候不可能不考虑到这个问题，所以它提供了启动模式来修改系统的默认行为。目前有四种启动模式：standard、singleTop、singleTask 和 singleInstance，下面先介绍各种启动模式的含义：

（1）standard：标准模式，这也是系统的默认模式。每次启动一个 Activity 都会重新创建一个新的实例，不管这个实例是否已经存在。被创建的实例的生命周期符合典型情况下 Activity 的生命周期，如上节描述，它的 onCreate、onStart、onResume 都会被调用。这是一种典型的多实例实现，一个任务栈中可以有多个实例，每个实例也可以属于不同的任务栈。在这种模式下，谁启动了这个 Activity，那么这个 Activity 就运行在启动它的那个 Activity 所在的栈中。比如 Activity A 启动了 Activity B（B 是标准模式），那么 B 就会进入到 A 所在的栈中。不知道读者是否注意到，当我们用 ApplicationContext 去启动 standard 模式的

Activity的时候会报错，错误如下：

```
E/AndroidRuntime(674): android.util.AndroidRuntimeException: Calling
startActivity from outside of an Activity context requires the FLAG_
ACTIVITY_NEW_TASK flag. Is this really what you want?
```

相信这句话读者一定不陌生，这是因为standard模式的Activity默认会进入启动它的Activity所属的任务栈中，但是由于非Activity类型的Context（如ApplicationContext）并没有所谓的任务栈，所以这就有问题了。解决这个问题的方法是为待启动Activity指定FLAG_ACTIVITY_NEW_TASK标记位，这样启动的时候就会为它创建一个新的任务栈，这个时候待启动Activity实际上是以singleTask模式启动的，读者可以仔细体会。

（2）singleTop：栈顶复用模式。在这种模式下，如果新Activity已经位于任务栈的栈顶，那么此Activity不会被重新创建，同时它的onNewIntent方法会被回调，通过此方法的参数我们可以取出当前请求的信息。需要注意的是，这个Activity的onCreate、onStart不会被系统调用，因为它并没有发生改变。如果新Activity的实例已存在但不是位于栈顶，那么新Activity仍然会重新重建。举个例子，假设目前栈内的情况为ABCD，其中ABCD为四个Activity，A位于栈底，D位于栈顶，这个时候假设要再次启动D，如果D的启动模式为singleTop，那么栈内的情况仍然为ABCD；如果D的启动模式为standard，那么由于D被重新创建，导致栈内的情况就变为ABCDD。

（3）singleTask：栈内复用模式。这是一种单实例模式，在这种模式下，只要Activity在一个栈中存在，那么多次启动此Activity都不会重新创建实例，和singleTop一样，系统也会回调其onNewIntent。具体一点，当一个具有singleTask模式的Activity请求启动后，比如Activity A，系统首先会寻找是否存在A想要的任务栈，如果不存在，就重新创建一个任务栈，然后创建A的实例后把A放到栈中。如果存在A所需的任务栈，这时要看A是否在栈中有实例存在，如果有实例存在，那么系统就会把A调到栈顶并调用它的onNewIntent方法，如果实例不存在，就创建A的实例并把A压入栈中。举几个例子：

- 比如目前任务栈S1中的情况为ABC，这个时候Activity D以singleTask模式请求启动，其所需要的任务栈为S2，由于S2和D的实例均不存在，所以系统会先创建任务栈S2，然后再创建D的实例并将其入栈到S2。

- 另外一种情况，假设D所需的任务栈为S1，其他情况如上面例子1所示，那么由于S1已经存在，所以系统会直接创建D的实例并将其入栈到S1。

- 如果 D 所需的任务栈为 S1，并且当前任务栈 S1 的情况为 ADBC，根据栈内复用的原则，此时 D 不会重新创建，系统会把 D 切换到栈顶并调用其 onNewIntent 方法，同时由于 singleTask 默认具有 clearTop 的效果，会导致栈内所有在 D 上面的 Activity 全部出栈，于是最终 S1 中的情况为 AD。这一点比较特殊，在后面还会对此种情况详细地分析。

通过上述 3 个例子，读者应该能比较清晰地理解 singleTask 的含义了。

（4）singleInstance：单实例模式。这是一种加强的 singleTask 模式，它除了具有 singleTask 模式的所有特性外，还加强了一点，那就是具有此种模式的 Activity 只能单独地位于一个任务栈中，换句话说，比如 Activity A 是 singleInstance 模式，当 A 启动后，系统会为它创建一个新的任务栈，然后 A 独自在这个新的任务栈中，由于栈内复用的特性，后续的请求均不会创建新的 Activity，除非这个独特的任务栈被系统销毁了。

上面介绍了几种启动模式，这里需要指出一种情况，我们假设目前有 2 个任务栈，前台任务栈的情况为 AB，而后台任务栈的情况为 CD，这里假设 CD 的启动模式均为 singleTask。现在请求启动 D，那么整个后台任务栈都会被切换到前台，这个时候整个后退列表变成了 ABCD。当用户按 back 键的时候，列表中的 Activity 会一一出栈，如图 1-7 所示。如果不是请求启动 D 而是启动 C，那么情况就不一样了，请看图 1-8，具体原因在本节后面会再进行详细分析。

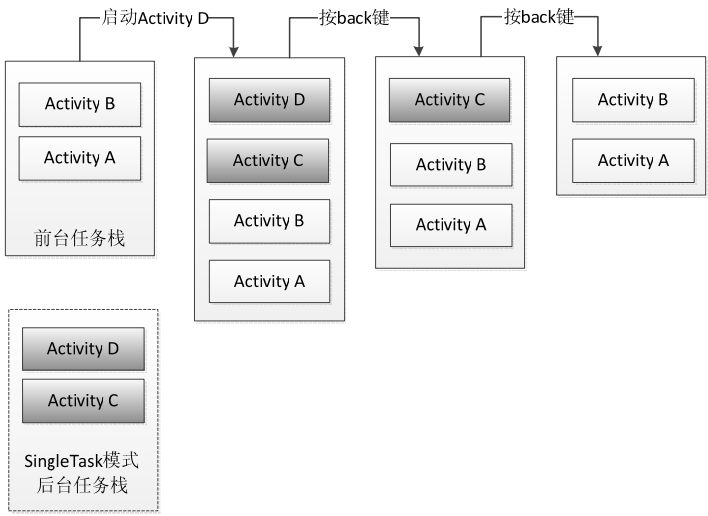

图 1-7　任务栈示例 1

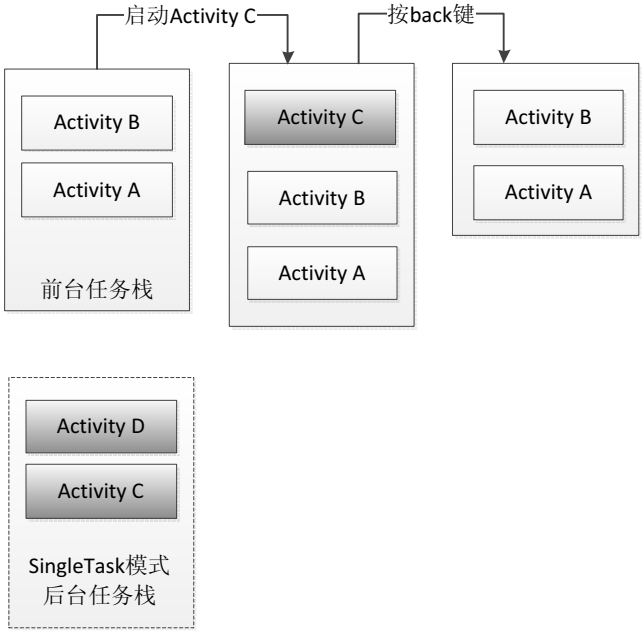

图 1-8　任务栈示例 2

另外一个问题是，在 singleTask 启动模式中，多次提到某个 Activity 所需的任务栈，什么是 Activity 所需要的任务栈呢？这要从一个参数说起：TaskAffinity，可以翻译为任务相关性。这个参数标识了一个 Activity 所需要的任务栈的名字，默认情况下，所有 Activity 所需的任务栈的名字为应用的包名。当然，我们可以为每个 Activity 都单独指定 TaskAffinity 属性，这个属性值必须不能和包名相同，否则就相当于没有指定。TaskAffinity 属性主要和 singleTask 启动模式或者 allowTaskReparenting 属性配对使用，在其他情况下没有意义。另外，任务栈分为前台任务栈和后台任务栈，后台任务栈中的 Activity 位于暂停状态，用户可以通过切换将后台任务栈再次调到前台。

当 TaskAffinity 和 singleTask 启动模式配对使用的时候，它是具有该模式的 Activity 的目前任务栈的名字，待启动的 Activity 会运行在名字和 TaskAffinity 相同的任务栈中。

当 TaskAffinity 和 allowTaskReparenting 结合的时候，这种情况比较复杂，会产生特殊的效果。当一个应用 A 启动了应用 B 的某个 Activity 后，如果这个 Activity 的 allowTaskReparenting 属性为 true 的话，那么当应用 B 被启动后，此 Activity 会直接从应用 A 的任务栈转移到应用 B 的任务栈中。这还是很抽象，再具体点，比如现在有 2 个应用 A

和 B，A 启动了 B 的一个 Activity C，然后按 Home 键回到桌面，然后再单击 B 的桌面图标，这个时候并不是启动了 B 的主 Activity，而是重新显示了已经被应用 A 启动的 Activity C，或者说，C 从 A 的任务栈转移到了 B 的任务栈中。可以这么理解，由于 A 启动了 C，这个时候 C 只能运行在 A 的任务栈中，但是 C 属于 B 应用，正常情况下，它的 TaskAffinity 值肯定不可能和 A 的任务栈相同（因为包名不同）。所以，当 B 被启动后，B 会创建自己的任务栈，这个时候系统发现 C 原本所想要的任务栈已经被创建了，所以就把 C 从 A 的任务栈中转移过来了。这种情况读者可以写个例子测试一下，这里就不做示例了。

如何给 Activity 指定启动模式呢？有两种方法，第一种是通过 AndroidMenifest 为 Activity 指定启动模式，如下所示。

```
<activity
    android:name="com.ryg.chapter_1.SecondActivity"
    android:configChanges="screenLayout"
    android:launchMode="singleTask"
    android:label="@string/app_name" />
```

另一种情况是通过在 Intent 中设置标志位来为 Activity 指定启动模式，比如：

```
Intent intent = new Intent();
intent.setClass(MainActivity.this, SecondActivity.class);
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);
```

这两种方式都可以为 Activity 指定启动模式，但是二者还是有区别的。首先，优先级上，第二种方式的优先级要高于第一种，当两种同时存在时，以第二种方式为准；其次，上述两种方式在限定范围上有所不同，比如，第一种方式无法直接为 Activity 设定 FLAG_ACTIVITY_CLEAR_TOP 标识，而第二种方式无法为 Activity 指定 singleInstance 模式。

关于 Intent 中为 Activity 指定的各种标记位，在下面的小节中会继续介绍。下面通过一个例子来体验启动模式的使用效果。还是前面的例子，这里我们把 MainActivity 的启动模式设为 singleTask，然后重复启动它，看看是否会重复创建，代码修改如下：

```
<activity
    android:name="com.ryg.chapter_1.MainActivity"
    android:configChanges="orientation|screenSize"
    android:label="@string/app_name"
```

```
        android:launchMode="singleTask" >
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />

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

    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        Log.d(TAG, "onNewIntent, time=" + intent.getLongExtra("time", 0));
    }

    findViewById(R.id.button1).setOnClickListener(new OnClickListener() {

        @Override
        public void onClick(View v) {
            Intent intent = new Intent();
            intent.setClass(MainActivity.this, MainActivity.class);
    intent.putExtra("time", System.currentTimeMillis());
            startActivity(intent);
        }
    });
```

根据上述修改，我们做如下操作，连续单击三次按钮启动 3 次 MainActivity，算上原本的 MainActvity 的实例，正常情况下，任务栈中应该有 4 个 MainActivity 的实例，但是我们为其指定了 singleTask 模式，现在来看一看到底有何不同。

执行 adb shell dumpsys activity 命令：

```
    ACTIVITY MANAGER ACTIVITIES (dumpsys activity activities)
      Main stack:
        TaskRecord{41350dc8 #9 A com.ryg.chapter_1}
        Intent { cmp=com.ryg.chapter_1/.MainActivity (has extras) }
          Hist #1: ActivityRecord{412cc188 com.ryg.chapter_1/.MainActivity}
            Intent { act=android.intent.action.MAIN cat=[android.intent.
            category.LAUNCHER] flg=0x
    0 cmp=com.ryg.chapter_1/.MainActivity bnds=[160,235][240,335] }
            ProcessRecord{411e6898 634:com.ryg.chapter_1/10052}
```

```
    TaskRecord{4125abc8 #2 A com.android.launcher}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.
    HOME] flg=0x10000000
m.android.launcher/com.android.launcher2.Launcher }
     Hist #0: ActivityRecord{412381f8 com.android.launcher/com.android.
     launcher2.Launcher}
       Intent { act=android.intent.action.MAIN cat=[android.intent.
       category.HOME] flg=0x1000
p=com.android.launcher/com.android.launcher2.Launcher }
       ProcessRecord{411d24c8 214:com.android.launcher/10013}

  Running activities (most recent first):
    TaskRecord{41350dc8 #9 A com.ryg.chapter_1}
      Run #1: ActivityRecord{412cc188 com.ryg.chapter_1/.MainActivity}
    TaskRecord{4125abc8 #2 A com.android.launcher}
      Run #0: ActivityRecord{412381f8 com.android.launcher/com.android.
      launcher2.Launcher}

  mResumedActivity: ActivityRecord{412cc188 com.ryg.chapter_1/.MainActivity}
  mFocusedActivity: ActivityRecord{412cc188 com.ryg.chapter_1/.MainActivity}

  Recent tasks:
  * Recent #0: TaskRecord{41350dc8 #9 A com.ryg.chapter_1}
  * Recent #1: TaskRecord{4125abc8 #2 A com.android.launcher}
  * Recent #2: TaskRecord{412b60a0 #5 A com.estrongs.android.pop.app.
    InstallMonitorActivity}
```

从上面导出的 Activity 信息可以看出，尽管启动了 4 次 MainActivity，但是它始终只有一个实例在任务栈中.从图 1-9 的 log 可以看出，Activity 的确没有重新创建，只是暂停了一下，然后调用了 onNewIntent，接着调用 onResume 就又继续了。

Level	PID	TID	Application	Tag	Text
D	755	755	com.ryg.chapter_1	MainActivity	onPause
D	755	755	com.ryg.chapter_1	MainActivity	onNewIntent, time=1422898165307
D	755	755	com.ryg.chapter_1	MainActivity	onResume
D	755	755	com.ryg.chapter_1	MainActivity	onPause
D	755	755	com.ryg.chapter_1	MainActivity	onNewIntent, time=1422898166173
D	755	755	com.ryg.chapter_1	MainActivity	onResume
D	755	755	com.ryg.chapter_1	MainActivity	onPause
D	755	755	com.ryg.chapter_1	MainActivity	onNewIntent, time=1422898167429
D	755	755	com.ryg.chapter_1	MainActivity	onResume

图 1-9　系统日志

现在我们去掉 singleTask，再来对比一下，还是同样的操作，单击三次按钮启动 MainActivity 三次。

执行 adb shell dumpsys activity 命令：

```
ACTIVITY MANAGER ACTIVITIES (dumpsys activity activities)
  Main stack:
    TaskRecord{41325370 #17 A com.ryg.chapter_1}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.
    LAUNCHER] flg=0x100000
p=com.ryg.chapter_1/.MainActivity }
      Hist #4: ActivityRecord{41236968 com.ryg.chapter_1/.MainActivity}
        Intent { cmp=com.ryg.chapter_1/.MainActivity (has extras) }
        ProcessRecord{411e6898 803:com.ryg.chapter_1/10052}
      Hist #3: ActivityRecord{411f4b30 com.ryg.chapter_1/.MainActivity}
        Intent { cmp=com.ryg.chapter_1/.MainActivity (has extras) }
        ProcessRecord{411e6898 803:com.ryg.chapter_1/10052}
      Hist #2: ActivityRecord{411edcb8 com.ryg.chapter_1/.MainActivity}
        Intent { cmp=com.ryg.chapter_1/.MainActivity (has extras) }
        ProcessRecord{411e6898 803:com.ryg.chapter_1/10052}
      Hist #1: ActivityRecord{411e7588 com.ryg.chapter_1/.MainActivity}
        Intent { act=android.intent.action.MAIN cat=[android.intent.category.
        LAUNCHER] flg=0x10
0 cmp=com.ryg.chapter_1/.MainActivity }
        ProcessRecord{411e6898 803:com.ryg.chapter_1/10052}
    TaskRecord{4125abc8 #2 A com.android.launcher}
    Intent { act=android.intent.action.MAIN cat=[android.intent.category.
    HOME] flg=0x10000000 c
m.android.launcher/com.android.launcher2.Launcher }
      Hist #0: ActivityRecord{412381f8 com.android.launcher/com.android.
      launcher2.Launcher}
        Intent { act=android.intent.action.MAIN cat=[android.intent.cate-
        gory.HOME] flg=0x100000
p=com.android.launcher/com.android.launcher2.Launcher }
        ProcessRecord{411d24c8 214:com.android.launcher/10013}

  Running activities (most recent first):
    TaskRecord{41325370 #17 A com.ryg.chapter_1}
      Run #4: ActivityRecord{41236968 com.ryg.chapter_1/.MainActivity}
```

```
      Run #3: ActivityRecord{411f4b30 com.ryg.chapter_1/.MainActivity}
      Run #2: ActivityRecord{411edcb8 com.ryg.chapter_1/.MainActivity}
      Run #1: ActivityRecord{411e7588 com.ryg.chapter_1/.MainActivity}
    TaskRecord{4125abc8 #2 A com.android.launcher}
      Run #0: ActivityRecord{412381f8 com.android.launcher/com.android.
      launcher2.Launcher}

  mResumedActivity: ActivityRecord{41236968 com.ryg.chapter_1/.MainAc-
  tivity}
  mFocusedActivity: ActivityRecord{41236968 com.ryg.chapter_1/.MainAc-
  tivity}

  Recent tasks:
  * Recent #0: TaskRecord{41325370 #17 A com.ryg.chapter_1}
  * Recent #1: TaskRecord{4125abc8 #2 A com.android.launcher}
  * Recent #2: TaskRecord{412c8d58 #16 A com.estrongs.android.pop.app.
    InstallMonitorActivity}
```

上面的导出信息很多，我们可以有选择地看，比如就看 Running activities (most recent first)这一块，如下所示。

```
  Running activities (most recent first):
    TaskRecord{41325370 #17 A com.ryg.chapter_1}
      Run #4: ActivityRecord{41236968 com.ryg.chapter_1/.MainActivity}
      Run #3: ActivityRecord{411f4b30 com.ryg.chapter_1/.MainActivity}
      Run #2: ActivityRecord{411edcb8 com.ryg.chapter_1/.MainActivity}
      Run #1: ActivityRecord{411e7588 com.ryg.chapter_1/.MainActivity}
    TaskRecord{4125abc8 #2 A com.android.launcher}
      Run #0: ActivityRecord{412381f8 com.android.launcher/com.android.
      launcher2.Launcher}
```

我们能够得出目前总共有 2 个任务栈，前台任务栈的 taskAffinity 值为 com.ryg.chapter_1，它里面有 4 个 Activity，后台任务栈的 taskAffinity 值为 com.android.launcher，它里面有 1 个 Activity，这个 Activity 就是桌面。通过这种方式来分析任务栈信息就清晰多了。

从上面的导出信息中可以看到，在任务栈中有 4 个 MainActivity，这也就验证了 Activity 的启动模式的工作方式。

上述四种启动模式，standard 和 singleTop 都比较好理解，singleInstance 由于其特殊性也好理解，但是关于 singleTask 有一种情况需要再说明一下。如图 1-7 所示，如果在 Activity B 中请求的不是 D 而是 C，那么情况如何呢？这里可以告诉读者的是，任务栈列表变成了 ABC，是不是很奇怪呢？Activity D 被直接出栈了。下面我们再用实例验证看看是不是这样。首先，还是使用上面的代码，但是我们做一下修改：

```
<activity
    android:name="com.ryg.chapter_1.MainActivity"
    android:configChanges="orientation|screenSize"
    android:label="@string/app_name"
    android:launchMode="standard" >
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
<activity
    android:name="com.ryg.chapter_1.SecondActivity"
    android:configChanges="screenLayout"
    android:label="@string/app_name"
    android:taskAffinity="com.ryg.task1"
    android:launchMode="singleTask" />

<activity
    android:name="com.ryg.chapter_1.ThirdActivity"
    android:configChanges="screenLayout"
    android:taskAffinity="com.ryg.task1"
    android:label="@string/app_name"
    android:launchMode="singleTask" />
```

我们将 SecondActivity 和 ThirdActivity 都设成 singleTask 并指定它们的 taskAffinity 属性为“com.ryg.task1”，注意这个 taskAffinity 属性的值为字符串，且中间必须含有包名分隔符“.”。然后做如下操作，在 MainActivity 中单击按钮启动 SecondActivity，在 SecondActivity 中单击按钮启动 ThirdActivity，在 ThirdActivity 中单击按钮又启动 MainActivity，最后再在 MainActivity 中单击按钮启动 SecondActivity，现在按 2 次 back 键，然后看到的是哪个 Activity？答案是回到桌面。是不是有点摸不到头脑了？没关系，接下来我们分析这个问题。

首先，从理论上分析这个问题，先假设 MainActivity 为 A，SecondActivity 为 B，

ThirdActivity 为 C。我们知道 A 为 standard 模式，按照规定，A 的 taskAffinity 值继承自 Application 的 taskAffinity，而 Application 默认 taskAffinity 为包名，所以 A 的 taskAffinity 为包名。由于我们在 XML 中为 B 和 C 指定了 taskAffinity 和启动模式，所以 B 和 C 是 singleTask 模式且有相同的 taskAffinity 值“com.ryg.task1”。A 启动 B 的时候，按照 singleTask 的规则，这个时候需要为 B 重新创建一个任务栈“com.ryg.task1”。B 再启动 C，按照 singleTask 的规则，由于 C 所需的任务栈（和 B 为同一任务栈）已经被 B 创建，所以无须再创建新的任务栈，这个时候系统只是创建 C 的实例后将 C 入栈了。接着 C 再启动 A，A 是 standard 模式，所以系统会为它创建一个新的实例并将到加到启动它的那个 Activity 的任务栈，由于是 C 启动了 A，所以 A 会进入 C 的任务栈中并位于栈顶。这个时候已经有两个任务栈了，一个是名字为包名的任务栈，里面只有 A，另一个是名字为“com.ryg.task1”的任务栈，里面的 Activity 为 BCA。接下来，A 再启动 B，由于 B 是 singleTask，B 需要回到任务栈的栈顶，由于栈的工作模式为“后进先出”，B 想要回到栈顶，只能是 CA 出栈。所以，到这里就很好理解了，如果再按 back 键，B 就出栈了，B 所在的任务栈已经不存在了，这个时候只能是回到后台任务栈并把 A 显示出来。注意这个 A 是后台任务栈的 A，不是“com.ryg.task1”任务栈的 A，接着再继续 back，就回到桌面了。分析到这里，我们得出一条结论，singleTask 模式的 Activity 切换到栈顶会导致在它之上的栈内的 Activity 出栈。

接着我们在实践中再次验证这个问题，还是采用 dumpsys 命令。我们省略中间的过程，直接看 C 启动 A 的那个状态，执行 adb shell dumpsys activity 命令，日志如下：

```
Running activities (most recent first):
TaskRecord{4132bd90 #12 A com.ryg.task1}
  Run #4: ActivityRecord{4133fd18 com.ryg.chapter_1/.MainActivity}
  Run #3: ActivityRecord{41349c58 com.ryg.chapter_1/.ThirdActivity}
  Run #2: ActivityRecord{4132bab0 com.ryg.chapter_1/.SecondActivity}
TaskRecord{4125a008 #11 A com.ryg.chapter_1}
  Run #1: ActivityRecord{41328c60 com.ryg.chapter_1/.MainActivity}
TaskRecord{41256440 #2 A com.android.launcher}
  Run #0: ActivityRecord{41231d30 com.android.launcher/com.android.launch
  er2.Launcher}
```

可以清楚地看到有 2 个任务栈，第一个（com.ryg.chapter_1）只有 A，第二个（com.ryg.task1）有 BCA，就如同我们上面分析的那样，然后再从 A 中启动 B，再看一下日志：

```
Running activities (most recent first):
TaskRecord{4132bd90 #12 A com.ryg.task1}
```

```
    Run #2: ActivityRecord{4132bab0 com.ryg.chapter_1/.SecondActivity}
  TaskRecord{4125a008 #11 A com.ryg.chapter_1}
    Run #1: ActivityRecord{41328c60 com.ryg.chapter_1/.MainActivity}
  TaskRecord{41256440 #2 A com.android.launcher}
    Run #0: ActivityRecord{41231d30 com.android.launcher/com.android.launch
    er2.Launcher}
```

可以发现在任务栈 com.ryg.task1 中只剩下 B 了，C、A 都已经出栈了，这个时候再按 back 键，任务栈 com.ryg.chapter_1 中的 A 就显示出来了，如果再 back 就回到桌面了。分析到这里，相信读者对 Activity 的启动模式已经有很深入的理解了。下面介绍 Activity 中常用的标志位。

1.2.2 Activity 的 Flags

Activity 的 Flags 有很多，这里主要分析一些比较常用的标记位。标记位的作用很多，有的标记位可以设定 Activity 的启动模式，比如 FLAG_ACTIVITY_NEW_TASK 和 FLAG_ACTIVITY_SINGLE_TOP 等；还有的标记位可以影响 Activity 的运行状态，比如 FLAG_ACTIVITY_CLEAR_TOP 和 FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS 等。下面主要介绍几个比较常用的标记位，剩下的标记位读者可以查看官方文档去了解，大部分情况下，我们不需要为 Activity 指定标记位，因此，对于标记位理解即可。在使用标记位的时候，要注意有些标记位是系统内部使用的，应用程序不需要去手动设置这些标记位以防出现问题。

FLAG_ACTIVITY_NEW_TASK

这个标记位的作用是为 Activity 指定“singleTask”启动模式，其效果和在 XML 中指定该启动模式相同。

FLAG_ACTIVITY_SINGLE_TOP

这个标记位的作用是为 Activity 指定“singleTop”启动模式，其效果和在 XML 中指定该启动模式相同。

FLAG_ACTIVITY_CLEAR_TOP

具有此标记位的 Activity，当它启动时，在同一个任务栈中所有位于它上面的 Activity 都要

出栈。这个标记位一般会和 singleTask 启动模式一起出现，在这种情况下，被启动 Activity 的实例如果已经存在，那么系统就会调用它的 onNewIntent。如果被启动的 Activity 采用 standard 模式启动，那么它连同它之上的 Activity 都要出栈，系统会创建新的 Activity 实例并放入栈顶。通过 1.2.1 节中的分析可以知道，singleTask 启动模式默认就具有此标记位的效果。

FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS

具有这个标记的 Activity 不会出现在历史 Activity 的列表中，当某些情况下我们不希望用户通过历史列表回到我们的 Activity 的时候这个标记比较有用。它等同于在 XML 中指定 Activity 的属性 android:excludeFromRecents="true"。

1.3 IntentFilter 的匹配规则

我们知道，启动 Activity 分为两种，显式调用和隐式调用。二者的区别这里就不多说了，显式调用需要明确地指定被启动对象的组件信息，包括包名和类名，而隐式调用则不需要明确指定组件信息。原则上一个 Intent 不应该既是显式调用又是隐式调用，如果二者共存的话以显式调用为主。显式调用很简单，这里主要介绍一下隐式调用。隐式调用需要 Intent 能够匹配目标组件的 IntentFilter 中所设置的过滤信息，如果不匹配将无法启动目标 Activity。IntentFilter 中的过滤信息有 action、category、data，下面是一个过滤规则的示例：

```
<activity
    android:name="com.ryg.chapter_1.ThirdActivity"
    android:configChanges="screenLayout"
    android:label="@string/app_name"
    android:launchMode="singleTask"
    android:taskAffinity="com.ryg.task1" >
    <intent-filter >
        <action android:name="com.ryg.charpter_1.c"/>
        <action android:name="com.ryg.charpter_1.d"/>
        <category android:name="com.ryg.category.c"/>
        <category android:name="com.ryg.category.d"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
</activity>
```

为了匹配过滤列表，需要同时匹配过滤列表中的 action、category、data 信息，否则匹配失败。一个过滤列表中的 action、category 和 data 可以有多个，所有的 action、category、data 分别构成不同类别，同一类别的信息共同约束当前类别的匹配过程。只有一个 Intent 同时匹配 action 类别、category 类别、data 类别才算完全匹配，只有完全匹配才能成功启动目标 Activity。另外一点，一个 Activity 中可以有多个 intent-filter，一个 Intent 只要能匹配任何一组 intent-filter 即可成功启动对应的 Activity，如下所示。

```
<activity android:name="ShareActivity">
    <!-- This activity handles "SEND" actions with text data -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/>
    </intent-filter>
    <!-- This activity also handles "SEND" and "SEND_MULTIPLE" with media
         data -->
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <action android:name="android.intent.action.SEND_MULTIPLE"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="application/vnd.google.panorama360+jpg"/>
        <data android:mimeType="image/*"/>
        <data android:mimeType="video/*"/>
    </intent-filter>
</activity>
```

下面详细分析各种属性的匹配规则。

1．action 的匹配规则

action 是一个字符串，系统预定义了一些 action，同时我们也可以在应用中定义自己的 action。action 的匹配规则是 Intent 中的 action 必须能够和过滤规则中的 action 匹配，这里说的匹配是指 action 的字符串值完全一样。一个过滤规则中可以有多个 action，那么只要 Intent 中的 action 能够和过滤规则中的任何一个 action 相同即可匹配成功。针对上面的过滤规则，只要我们的 Intent 中 action 值为“com.ryg.charpter_1.c”或者“com.ryg.charpter_ 1.d”都能成功匹配。需要注意的是，Intent 中如果没有指定 action，那么匹配失败。总结一下，action 的匹配要求 Intent 中的 action 存在且必须和过滤规则中的其中一个 action 相同，这里

需要注意它和 category 匹配规则的不同。另外，action 区分大小写，大小写不同字符串相同的 action 会匹配失败。

2．category 的匹配规则

category 是一个字符串，系统预定义了一些 category，同时我们也可以在应用中定义自己的 category。category 的匹配规则和 action 不同，它要求 Intent 中如果含有 category，那么所有的 category 都必须和过滤规则中的其中一个 category 相同。换句话说，Intent 中如果出现了 category，不管有几个 category，对于每个 category 来说，它必须是过滤规则中已经定义了的 category。当然，Intent 中可以没有 category，如果没有 category 的话，按照上面的描述，这个 Intent 仍然可以匹配成功。这里要注意下它和 action 匹配过程的不同，action 是要求 Intent 中必须有一个 action 且必须能够和过滤规则中的某个 action 相同，而 category 要求 Intent 可以没有 category，但是如果你一旦有 category，不管有几个，每个都要能够和过滤规则中的任何一个 category 相同。为了匹配前面的过滤规则中的 category，我们可以写出下面的 Intent，intent.addcategory (“com.ryg.category.c”)或者 Intent. addcategory (“com.ryg.category.d”)亦或者不设置 category。为什么不设置 category 也可以匹配呢？原因是系统在调用 startActivity 或者 startActivityForResult 的时候会默认为 Intent 加上“android.intent.category.DEFAULT”这个 category，所以这个 category 就可以匹配前面的过滤规则中的第三个 category。同时，为了我们的 activity 能够接收隐式调用，就必须在 intent-filter 中指定“android.intent.category.DEFAULT”这个 category，原因刚才已经说明了。

3．data 的匹配规则

data 的匹配规则和 action 类似，如果过滤规则中定义了 data，那么 Intent 中必须也要定义可匹配的 data。在介绍 data 的匹配规则之前，我们需要先了解一下 data 的结构，因为 data 稍微有些复杂。

data 的语法如下所示。

```
<data android:scheme="string"
      android:host="string"
      android:port="string"
      android:path="string"
      android:pathPattern="string"
      android:pathPrefix="string"
      android:mimeType="string" />
```

data 由两部分组成，mimeType 和 URI。mimeType 指媒体类型，比如 image/jpeg、audio/mpeg4-generic 和 video/*等，可以表示图片、文本、视频等不同的媒体格式，而 URI 中包含的数据就比较多了，下面是 URI 的结构：

```
<scheme>://<host>:<port>/[<path>|<pathPrefix>|<pathPattern>]
```

这里再给几个实际的例子就比较好理解了，如下所示。

```
content://com.example.project:200/folder/subfolder/etc
http://www.baidu.com:80/search/info
```

看了上面的两个示例应该就瞬间明白了，没错，就是这么简单。不过下面还是要介绍一下每个数据的含义。

Scheme：URI 的模式，比如 http、file、content 等，如果 URI 中没有指定 scheme，那么整个 URI 的其他参数无效，这也意味着 URI 是无效的。

Host：URI 的主机名，比如 www.baidu.com，如果 host 未指定，那么整个 URI 中的其他参数无效，这也意味着 URI 是无效的。

Port：URI 中的端口号，比如 80，仅当 URI 中指定了 scheme 和 host 参数的时候 port 参数才是有意义的。

Path、pathPattern 和 pathPrefix：这三个参数表述路径信息，其中 path 表示完整的路径信息；pathPattern 也表示完整的路径信息，但是它里面可以包含通配符“*”，“*”表示 0 个或多个任意字符，需要注意的是，由于正则表达式的规范，如果想表示真实的字符串，那么“*”要写成“*”，“\”要写成“\\\\”；pathPrefix 表示路径的前缀信息。

介绍完 data 的数据格式后，我们要说一下 data 的匹配规则了。前面说到，data 的匹配规则和 action 类似，它也要求 Intent 中必须含有 data 数据，并且 data 数据能够完全匹配过滤规则中的某一个 data.这里的完全匹配是指过滤规则中出现的 data 部分也出现在了 Intent 中的 data 中。下面分情况说明。

（1）如下过滤规则：

```
<intent-filter>
    <data android:mimeType="image/*" />
```

```
    ...
</intent-filter>
```

这种规则指定了媒体类型为所有类型的图片，那么 Intent 中的 mimeType 属性必须为“image/*”才能匹配，这种情况下虽然过滤规则没有指定 URI，但是却有默认值，URI 的默认值为 content 和 file。也就是说，虽然没有指定 URI，但是 Intent 中的 URI 部分的 schema 必须为 content 或者 file 才能匹配，这点是需要尤其注意的。为了匹配（1）中规则，我们可以写出如下示例：

```
intent.setDataAndType(Uri.parse("file://abc"),"image/png")。
```

另外，如果要为 Intent 指定完整的 data，必须要调用 setDataAndType 方法，不能先调用 setData 再调用 setType，因为这两个方法彼此会清除对方的值，这个看源码就很容易理解，比如 setData：

```
public Intent setData(Uri data) {
    mData = data;
    mType = null;
    return this;
}
```

可以发现，setData 会把 mimeType 置为 null，同理 setType 也会把 URI 置为 null。

（2）如下过滤规则：

```
<intent-filter>
    <data android:mimeType="video/mpeg" android:scheme="http" ... />
    <data android:mimeType="audio/mpeg" android:scheme="http" ... />
    ...
</intent-filter>
```

这种规则指定了两组 data 规则，且每个 data 都指定了完整的属性值，既有 URI 又有 mimeType。为了匹配（2）中规则，我们可以写出如下示例：

```
intent.setDataAndType(Uri.parse(″http://abc″),"video/mpeg")
```

或者

```
intent. setDataAndType (Uri.parse("http://abc"),"audio/mpeg")
```

通过上面两个示例，读者应该已经明白了 data 的匹配规则，关于 data 还有一个特殊情况需要说明下，这也是它和 action 不同的地方，如下两种特殊的写法，它们的作用是一样的：

```
<intent-filter . . . >
    <data android:scheme="file" android:host="www.baidu.com" />
    . . .
</intent-filter>

<intent-filter . . . >
    <data android:scheme="file" />
    <data android:host="www.baidu.com" />
    . . .
</intent-filter>
```

到这里我们已经把 IntentFilter 的过滤规则都讲解了一遍，还记得本节前面给出的一个 intent-filter 的示例吗？现在我们给出完全匹配它的 Intent：

```
Intent intent = new Intent("com.ryg.charpter_1.c");
intent.addCategory("com.ryg.category.c");
intent.setDataAndType(Uri.parse("file://abc"), "text/plain");
startActivity(intent);
```

还记得 URI 的 schema 是有默认值的吗？如果把上面的 intent.setDataAndType(Uri.parse("file://abc"), "text/plain") 这句改成 intent.setDataAndType(Uri.parse("http://abc"), "text/plain")，打开 Activity 的时候就会报错，提示无法找到 Activity，如图 1-10 所示。另外一点，Intent-filter 的匹配规则对于 Service 和 BroadcastReceiver 也是同样的道理，不过系统对于 Service 的建议是尽量使用显式调用方式来启动服务。

Level	Application	Tag	Text
D	com.ryg.chapter_1	AndroidRuntime	Shutting down VM
W	com.ryg.chapter_1	dalvikvm	threadid=1: thread exiting with uncaught exception (group=0x409c01f8)
E	com.ryg.chapter_1	AndroidRuntime	FATAL EXCEPTION: main
E	com.ryg.chapter_1	AndroidRuntime	android.content.ActivityNotFoundException: No Activity found to handle Intent { act=com.ryg.charpter_1.c cat=[com.ryg.category.c] dat=http://abc typ=text/plain (has extras) }
E	com.ryg.chapter_1	AndroidRuntime	at android.app.Instrumentation.checkStartActivityResult(Instrumentation.java :1512)
E	com.ryg.chapter_1	AndroidRuntime	at android.app.Instrumentation.execStartActivity(Instrumentation.java:1384)
E	com.ryg.chapter_1	AndroidRuntime	at android.app.Activity.startActivityForResult(Activity.java:3190)
E	com.ryg.chapter_1	AndroidRuntime	at android.app.Activity.startActivity(Activity.java:3297)
E	com.ryg.chapter_1	AndroidRuntime	at com.ryg.chapter_1.MainActivity$1.onClick(MainActivity.java:36)
E	com.ryg.chapter_1	AndroidRuntime	at android.view.View.performClick(View.java:3511)

图 1-10　系统日志

最后，当我们通过隐式方式启动一个 Activity 的时候，可以做一下判断，看是否有 Activity 能够匹配我们的隐式 Intent，如果不做判断就有可能出现上述的错误了。判断方法有两种：采用 PackageManager 的 resolveActivity 方法或者 Intent 的 resolveActivity 方法，如果它们找不到匹配的 Activity 就会返回 null，我们通过判断返回值就可以规避上述错误了。另外，PackageManager 还提供了 queryIntentActivities 方法，这个方法和 resolveActivity 方法不同的是：它不是返回最佳匹配的 Activity 信息而是返回所有成功匹配的 Activity 信息。我们看一下 queryIntentActivities 和 resolveActivity 的方法原型：

```
public abstract List<ResolveInfo> queryIntentActivities(Intent intent, int
flags);
public abstract ResolveInfo resolveActivity(Intent intent, int flags);
```

上述两个方法的第一个参数比较好理解，第二个参数需要注意，我们要使用 MATCH_DEFAULT_ONLY 这个标记位，这个标记位的含义是仅仅匹配那些在 intent-filter 中声明了<category android:name="android.intent.category.DEFAULT"/>这个 category 的 Activity。使用这个标记位的意义在于，只要上述两个方法不返回 null，那么 startActivity 一定可以成功。如果不用这个标记位，就可以把 intent-filter 中 category 不含 DEFAULT 的那些 Activity 给匹配出来，从而导致 startActivity 可能失败。因为不含有 DEFAULT 这个 category 的 Activity 是无法接收隐式 Intent 的。在 action 和 category 中，有一类 action 和 category 比较重要，它们是：

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

这二者共同作用是用来标明这是一个入口 Activity 并且会出现在系统的应用列表中，少了任何一个都没有实际意义，也无法出现在系统的应用列表中，也就是二者缺一不可。另外，针对 Service 和 BroadcastReceiver，PackageManager 同样提供了类似的方法去获取成功匹配的组件信息。