Android笔记（上篇）

目录

[目录 1](#_Toc463193134)

[1 APP FOUNDAMENTAL 3](#_Toc463193135)

[2 System Permissions 4](#_Toc463193136)

[3 进程和线程 5](#_Toc463193137)

[4 Handler与Looper… 7](#_Toc463193138)

[5 Intent And IntentFilter 8](#_Toc463193139)

[5.1 基本内容 8](#_Toc463193140)

[5.2 隐式启动组件的安全性分析 8](#_Toc463193141)

[5.3 AIDL学习笔记 9](#_Toc463193142)

[5.3.1 AIDL产生的原因分析 9](#_Toc463193143)

[5.3.2 AIDL的实现 11](#_Toc463193144)

[5.3.3 AIDL的限制 12](#_Toc463193145)

[5.3.4 安卓5.0以后的使用要求 13](#_Toc463193146)

[6 Context 14](#_Toc463193147)

[2. Activity 15](#_Toc463193148)

[2.1 Activity基本概念 15](#_Toc463193149)

[2.2 启动模式 15](#_Toc463193150)

[2.3 任务和返回栈（Tasks and Back Stack） 16](#_Toc463193151)

[2.4 Fragment 18](#_Toc463193152)

[2.4.1 基本概念 18](#_Toc463193153)

[2.4.2 代码实现 19](#_Toc463193154)

[2.4.3 待处理问题 19](#_Toc463193155)

[2.4.4 Analyses for fragment 19](#_Toc463193156)

[2.5 Analyses 19](#_Toc463193157)

[2.5.1 Activity的启动及状态存取分析 20](#_Toc463193158)

[3. Content Provider官方文档笔记（待续） 23](#_Toc463193159)

[3.1 基本内容 23](#_Toc463193160)

[3.2 源码笔记 23](#_Toc463193161)

[4. Service学习笔记 25](#_Toc463193162)

[4.1 基本内容 25](#_Toc463193163)

[4.2 Should you use a service or a thread? 26](#_Toc463193164)

[4.3 unbindService()分析 26](#_Toc463193165)

[4.4 其他 27](#_Toc463193166)

[4.5 疑问 27](#_Toc463193167)

[5. Broadcast Receiver 28](#_Toc463193168)

[5.1 基本概念 28](#_Toc463193169)

[5.2 系统广播的特点 28](#_Toc463193170)

[5.2.1 短信广播 28](#_Toc463193171)

[5.2.2 电话广播 28](#_Toc463193172)

[5.3 Analyses 28](#_Toc463193173)

[5.3.1 Receiver工作进程分析 28](#_Toc463193174)

[6. View 29](#_Toc463193175)

[6.1 基本概念 29](#_Toc463193176)

[6.2 参数 29](#_Toc463193177)

[6.3 事件分发 29](#_Toc463193178)

[6.4 View的布局流程 31](#_Toc463193179)

[6.4.1 基本概念 31](#_Toc463193180)

[6.4.2 执行方法流程表 32](#_Toc463193181)

[6.4.3 View的测量过程 32](#_Toc463193182)

[6.4.4 观察到的现象 33](#_Toc463193183)

[6.4.5 疑问 34](#_Toc463193184)

[7. 设计模式 35](#_Toc463193185)

[7.1 Android程序的基本框架 35](#_Toc463193186)

[7.2 业务逻辑 35](#_Toc463193187)

[7.3 设计模式的类型 35](#_Toc463193188)

[7.4 MVC设计模式 35](#_Toc463193189)

[7.5 MVP设计模式 35](#_Toc463193190)

[8. Manager 37](#_Toc463193191)

[8.1 devicePolicyManager 37](#_Toc463193192)

[8.2 ActivityManager 37](#_Toc463193193)

[9. 重要基本概念 39](#_Toc463193194)

[9.1 回调函数 39](#_Toc463193195)

[10. Analysis 41](#_Toc463193196)

[10.1 界面交互性的交替（待续） 41](#_Toc463193197)

[10.2 混合启动服务 41](#_Toc463193198)

[10.3 0xff与byte的运算分析 41](#_Toc463193199)

1. APP FOUNDAMENTAL
2. The Android operating system is a multi-user Linux system in which each app is a different user. By default, the system assigns each app a unique Linux user ID (the ID is used only by the system and is unknown to the app). The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.

所以在底层，不同用户的代码文件，应该是即使知道包名，也不能够直接拿来使用。因为不同用户之间，除了包名，还有用户ID。一个ID下的文件，只可以这个ID的访问。所以AIDL分析里，接口文件直接放过去不能用。5.0之后全部谷歌要求用显式意图调用Service（实现AIDL），官方示例代码中，虽然表面上使用的是相应Service的包名，但是下面还有一句话，也进行了setAction()，只不过action不是写的Service的action属性，而是需要绑定的接口包名，所以感觉谷歌还是在底层对包名进行了修饰，不是表面上我们看到的以com开头的包名。

1. When the system starts a component, it starts the process for that app (if it's not already running) and instantiates the classes needed for the component. For example, if your app starts the activity in the camera app that captures a photo, that activity runs in the process that belongs to the camera app, not in your app's process.

所以进程之间的通讯（数据传输）也可以通过startActivityForResult()方式实现（已经由代码实现并证实），但是这种方式没有什么意义，因为基本不会存在打开第三方（系统自带应用、支付宝、微信朋友圈等除外）Activity的应用场景，而且即使需要进行该操作，一般也只是发送数据给第三方，获取数据是比较困难的，因为被打开的Activity需要setResult(intent)，第三方应用怎么可能知道不同的调用者需要他返回什么数据，所以这个setResult()不大好实现。

1. Because the system runs each app in a separate process with file permissions that restrict access to other apps, your app cannot directly activate a component from another app. The Android system, however, can. So, to activate a component in another app, you must deliver a message to the system that specifies your intent to start a particular component. The system then activates the component for you.

隐式调用其实是通知系统你要做什么，系统帮你做。

1. asynchronous message called an intent。an intent can be either explicit or implicit。

意图是区分显式和隐式的，显式调用、隐式调用，其实指的是你的意图是显式还是隐式。

1. 广播不区分显式和隐式。
2. 系统通过将接收到的 Intent 与设备上的其他应用的清单文件中提供的 Intent 过滤器进行比较来确定可以响应 Intent 的组件。这么说的话，manifest相当于windows的注册表。
3. Activity、Service、Broadcast的生命周期是一样的，只要进程被销毁，就都不存在了。之所以进程消失了，还可以启动，是因为存在intent-filter，系统检索manifest，发现之后启动进程。没有进程，所有东西无法运行。
4. System Permissions
5. Linux thereby isolates applications from each other and from the system.
6. A central design point of the Android security architecture is that no application, by default, has permission to perform any operations that would adversely impact other applications, the operating system, or the user. This includes reading or writing the user's private data (such as contacts or emails), reading or writing another application's files, performing network access, keeping the device awake, and so on.

按照这句话的意思，你是没法在A应用中直接使用绝对路径访问B应用的文件的。但是这个其实是由B应用创建文件时所设的权限决定的，比如 [MODE\_WORLD\_READABLE](https://developer.android.com/reference/android/content/Context.html#MODE_WORLD_READABLE)权限。

1. applications must explicitly share resources and data
2. In particular the Dalvik VM is not a security boundary, and any app can run native code (see [the Android NDK](https://developer.android.com/tools/sdk/ndk/index.html)). All types of applications — Java, native, and hybrid — are sandboxed in the same way and have the same degree of security from each other.
3. Android gives each package a distinct Linux user ID. You can use the [sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId) attribute in the AndroidManifest.xml's [manifest](https://developer.android.com/reference/android/R.styleable.html#AndroidManifest) tag of each package to have them assigned the same user ID. By doing this, for purposes of security the two packages are then treated as being the same application, with the same user ID and file permissions. Note that in order to retain security, only two applications signed with the same signature (and requesting the same sharedUserId) will be given the same user ID.

两个应用manifest的[sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId) 属性要一样，并且发布的签名要一样才可以在linux系统下被当做一个账户。

这样做很安全，如果只是要求[sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId) ，比如支付宝自己有两个模块，通过[sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId) ，使得在实际运行的时候可以随意共享数据。但是如果有一个应用盗取了支付宝的[sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId)，他就可以也注册这个属性，从而盗取支付宝的数据。

然而实际上安卓系统不但要求[sharedUserId](https://developer.android.com/reference/android/R.attr.html#sharedUserId) ，还要有同一个apk签名。这个签名文件是绝对的机文件，肯定第三方是拿不到的，所以就不会出现上面轻易盗取支付宝数据的情况。

1. permissions that don't pose much risk to the user's privacy or the device's operation), the system automatically grants those permissions. If your app lists dangerous permissions in its manifest (that is, permissions that could potentially affect the user's privacy or the device's normal operation), the system asks the user to explicitly grant those permissions.
2. 6.0以后，应用在运行的时候向用户询问是否承认permission权限，并且用户可以在任何时候撤销相应的权限。

5.1及以前，应用在安装的时候向用户询问是否承认permission权限，并且用户仅可以卸载应用时撤销相应的权限。说白了就是想要修改权限，就得重新安装。官方文档是这么说的，但是魅族5.1的系统是可以在设置里修改的。

1. There are a couple of permissions that don't behave like normal and dangerous permissions. [SYSTEM\_ALERT\_WINDOW](https://developer.android.com/reference/android/Manifest.permission.html#SYSTEM_ALERT_WINDOW) and [WRITE\_SETTINGS](https://developer.android.com/reference/android/Manifest.permission.html#WRITE_SETTINGS) are particularly sensitive。

使用这两个permission的时候需要注意，具体使用方法看文档。

1. Enforcing Permissions in AndroidManifest.xml

通过自定义的permission可以实现某个Activity只允许具有权限的activity开启自己的功能，感觉这个功能在金融应用上应该挺有用，非登录页面没有权限进入涉及到money的activity。

1. 进程和线程
2. All components that run in the same process are instantiated in the UI thread, and system calls to each component are dispatched from that thread.

什么叫做四大组件运行在主线程？

答：1）四大组件对象的创建均是在主线程完成的

2）四大组件的回调方法均是在主线程中执行的

四大组件其实本质都是回调方法的集合，所以组件的运行本质上就是各个回调方法被主线程回调而已。所以完全可能同时运行在主线程。

所谓执行在哪个线程，就是代码在哪个线程执行。一个对象的回调方法可以被主线程调用，也可以被子线程调用。四大组件与其他对象的区别仅仅是，其对象均在主线程中创建，其他线程一般拿不到这些对象，组件的回调方法只能被主线程调用，所以表面上看起来四大组件就只能够运行在主线程。其实如果拿到组件的对象，然后放到子线程中调用其回调方法，也是可以的。

1. Activity运行在主线程，任何其他阻塞、占用主线程的操作都会剥夺该Activity的可交互性。所以说明显示Toast（创建可能是在主线程，但维持显示应该是在子线程）。

（下面这段有待确认，存在错误）

Menu占用主线程，所以剥夺了Activity的可交互性。Menu打开的时候，Activity上的控件无法点击，但是Activity的onPause方法没有被回调（代码实测T03）。

for example, if the foreground activity started a dialog, which allows the previous activity to be seen behind it.

对话框是剥夺Activity交互性的。

onPause() is called when the device goes to sleep or when a dialog appears.(代码实测T03，不会触发onPause()方法)

Dialog就会剥夺Activity的可交互性，从中可以看出dialog工作在主线程。这句话有问题，需要验证。因为代码实测显示当dialog显示在Activity之上时，Activity的onPause()方法并没有被回调。在dialog的setCancelable(false)的情况下，点击dialog边框以外的区域的确是不退出dialog，并且点击不到Activity上的控件，但是此时Activity的onPause()方法同样没有执行。Dialog，Activity，Toast是分布在可以同时存在于屏幕中的三个Window层，所以可能不需要像Activity之间在同一个window层进行切换时，互相之间要让出这一层，一层只能放一个Activity。

1. 需要长时间执行的任务，最好放入Service中。因为当所有Activity销毁之后，该应用在系统中以空进程运行。空进程是最容易在系统内存不足时被销毁的进程。这里可能可以解释Serivce和子线程的一点区别。如果在Activity中创建子线程，当Activity销毁时，子线程应该是保持继续运行的（？），这时当进程被销毁时，子线程也会跟着被销毁。

启动了子线程的进程，会不会被系统回收？应该会，不然就不会有Service

1. 过程式语句执行过程中创建子线程对下语句语句执行的影响。

|  |
| --- |
| 1. 语句A 2. 语句B |

情景：语句A与语句B自上而下顺序执行，并且语句B的顺利执行的前提是语句A顺利执行。

在这种情形下，如果语句A中创建了子线程，语句B怎么需要等到子线程执行完成之后才能开始执行。然后由于语句A在开启子线程之后并没有在主线程发生阻塞，所以语句B会紧接着语句A执行。此时A开启的子线程极有可能没有完成相应的操作，那么语句B就会报错，因为B要在A顺利执行完成后才能够调用。

SurfaceView这个重量级控件的在执行下列语句时，就会发生这个问题。

|  |
| --- |
| 1. SurfaceHolder holder = sfView.getHolder(); 2. imageView.setHolder(); |

所以在遇到这种情况的时候，要注意保证第二句执行之前，判断一下前一句中的子线程完成没有。

1. <activity>, <service>, <receiver>, and <provider>, By default, all components of the same application run in the same process with a single thread of execution.
2. The decision whether to terminate a process, therefore, depends on the state of the components running in that process.
3. 进程的等级划分:
   1. Android ranks a process at the highest level it can, based upon the importance of the components currently active in the process.
   2. a process that is serving another process can never be ranked lower than the process it is serving.
   3. Foregroud Process: It hosts a BroadcastReceiver that's executing its onReceive() method.
4. When a call on a method implemented in an IBinder originates in the same process in which the IBinder is running, the method is executed in the caller's thread. However, when the call originates in another process, the method is executed in a thread chosen from a pool of threads that the system maintains in the same process as the IBinder (it's not executed in the UI thread of the process). For example, whereas a service's onBind() method would be called from the UI thread of the service's process, methods implemented in the object that onBind() returns (for example, a subclass that implements RPC methods) would be called from threads in the pool.

当外部进程通过Ibinder对象调用Service中的方法时，方法运行在什么位置？系统内存的分配和工作原理是什么？

按照这里说的，外部进程远程访问Service的时候，系统会创建线程池，那Service中，我自己还用不用创建子线程？

1. Android offers a mechanism for interprocess communication (IPC) using remote procedure calls (RPCs), in which a method is called by an activity or other application component, but executed remotely (in another process), with any result returned back to the caller. This entails decomposing a method call and its data to a level the operating system can understand, transmitting it from the local process and address space to the remote process and address space, then reassembling and reenacting the call there. Return values are then transmitted in the opposite direction.

IPC通过RPCs技术实现。当在B进程中绑定A进程中的服务时，此时B中的方法如果调用A的Service中的方法，实际是将B中的方法和数据在B进程中拆分，然后通过系统传送至A进程及其相对的地址空间中，然后方法在A进程中执行。当方法执行完毕后，得到的结果再从A进程中返回到B进程中。IPC进程间通讯是效果，具体技术叫PRCs（Remote Procedure Calls）

1. Handler与Looper…
2. 谁收消息，谁就创建Handler对象。
3. 谁发消息，就去接收消息的类找Handler对象，然后拿过来发送消息
4. Intent And IntentFilter
   1. 基本内容
5. fully qualified class name 完全限定类名
6. **显式 Intent**：按名称（完全限定类名）指定要启动的组件。通常，您会在自己的应用中使用显式 Intent 来启动组件，这是因为您知道要启动的 Activity 或服务的类名。
7. **隐式 Intent**：不会指定特定的组件，而是声明要执行的常规操作，从而允许其他应用中的组件来处理它。
8. Intent-filter是应用清单文件中的一个表达式，它指定该组件要接收的 Intent 类型。
9. data的sheme属性用来提供数据的Uri，大量的数据直接放在intent中可能不大合适，另外，如果需要调用的数据是由Content Provider提供的，也没法有intent直接提供，所以sheme属性一般都是uri。
10. data的mimetype用来指定数据的类型。以便告诉系统调用能够处理相关类的组件。
    1. 隐式启动组件的安全性分析

为了确保应用的安全性，启动 [Service](https://developer.android.com/reference/android/app/Service.html) 时，请始终使用显式 Intent，且不要为服务声明 Intent 过滤器。使用隐式 Intent 启动服务存在安全隐患，因为您无法确定哪些服务将响应 Intent，且用户无法看到哪些服务已启动。从 Android 5.0（API 级别 21）开始，如果使用隐式 Intent 调用 [bindService()](https://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int))，系统会抛出异常。

上面是官方文档对隐式意图安全性的描述，下面对这段话做一点分析：

1. 官方文档对显式intent和隐式intent启动Activity和Service的过程做了如下描述：

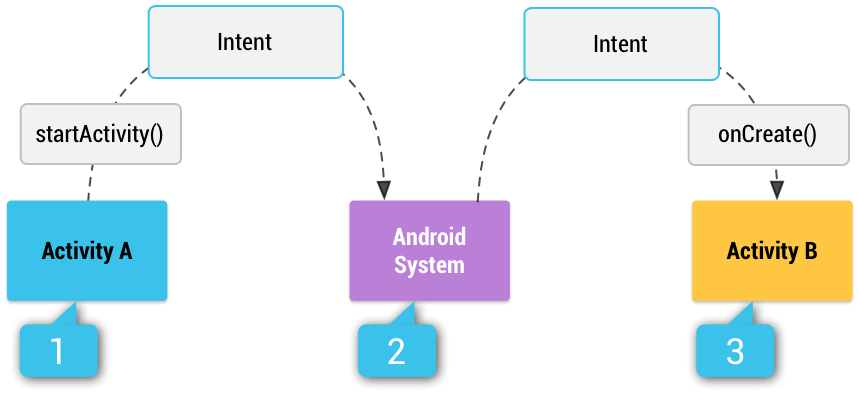
创建显式 Intent 启动 Activity 或服务时，系统将立即启动 [Intent](https://developer.android.com/reference/android/content/Intent.html) 对象中指定的应用组件。

创建隐式 Intent 时，Android 系统通过将 Intent 的内容与在设备上其他应用的[清单文件](https://developer.android.com/guide/topics/manifest/manifest-intro.html)中声明的 Intent 过滤器进行比较，从而找到要启动的相应组件。[Intent](https://developer.android.com/reference/android/content/Intent.html)如果 Intent 与 Intent 过滤器匹配，则系统将启动该组件，并将其传递给对象。

When starting a [Service](file:///C:\\\\\\\\\\\\\\\\Users\\\\\\\\\\\\\\\\cuihaichen\\\\\\\\\\\\\\\\AppData\\\\\\\\\\\\\\\\Local\\\\\\\\\\\\\\\\Android\\\\\\\\\\\\\\\\sdk\\\\\\\\\\\\\\\\docs\\\\\\\\\\\\\\\\reference\\\\\\\\\\\\\\\\android\\\\\\\\\\\\\\\\app\\\\\\\\\\\\\\\\Service.html), you should **always specify the component name**. Otherwise, you cannot be certain what service will respond to the intent, and the user cannot see which service starts.

如果多个 Intent 过滤器兼容，则系统会显示一个对话框，支持用户选取要使用的应用（但是Serivce不需要用户选取，可能只任意启动一个）。

Intent 过滤器是应用清单文件中的一个表达式，它指定该组件要接收的 Intent 类型。例如，通过为 Activity 声明 Intent 过滤器，您可以使其他应用能够直接使用某一特定类型的 Intent 启动 Activity。同样，如果您没有为 Activity 声明任何 Intent 过滤器，则 Activity 只能通过显式 Intent 启动。



**图 1.**隐式 Intent 如何通过系统传递以启动其他 Activity 的图解：

**[1]**Activity A 创建包含操作描述的 [Intent](https://developer.android.com/reference/android/content/Intent.html)，并将其传递给[startActivity()](https://developer.android.com/reference/android/content/Context.html#startActivity(android.content.Intent))。

**[2]** Android 系统搜索所有应用中与 Intent 匹配的 Intent 过滤器。找到匹配项之后，

**[3]** 该系统通过调用匹配 Activity（Activity B）的 [onCreate()](https://developer.android.com/reference/android/app/Activity.html#onCreate(android.os.Bundle)) 方法并将其传递给 [Intent](https://developer.android.com/reference/android/content/Intent.html)，

以此启动匹配 Activity。

1. 隐式意图可以用来启动Activity和Service组件（startService和bindService，以下不再区分start和bind）。
2. 这里的安全性我觉得应该从两个方面分析：
   1. Intent发出者：发送的Intent里存储数据的安全性
   2. Intent接收者：被调用的Activity和Service的安全性
3. Intent发出者：发送的Intent里存储数据的安全性分析（前提，该intent包含数据）：

利用隐式intent启动Activity或者Service时，系统会根据intent的action属性检索所有应用的manifest文件，只要其中某个组件的intent-filter的action属性与隐式intent的相同，就会被调用（多个组件具有相同action属性，系统会弹出窗口让用户确定开启哪个），并且可以获取intent中包含的数据，这样数据就有可能泄露给未知的组件。

当然这种情况发生的前提可能比较偶然：

* 1. 未知组件盗取了相应的action属性值
  2. 发出intent的组件携带的数据比较敏感，涉及到用户某些方面的安全性（金融账户等）。支付宝说是用C编的，所以这些金融软件其实也可能根本就没有这个问题。
  3. 多个组件响应隐式intent调用时，系统弹出对话框供用户选择使用哪个应用，结果用户还选错了

即使如上面分析的三种情况是合理的，并且以上的前提都发生，才会出现intent里携带的数据在隐式启动组件时，发生敏感数据泄露的风险。

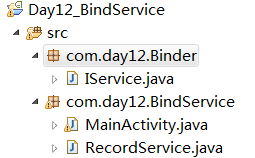
关键的问题在于，敏感数据在发送的时候应该是加密的，所以可能上面说的根本不对。

1. Intent接收者：被调用的Activity和Service的安全性分析：

如果Activity和Service定义了intent-filter，在被第三方盗取到action属性后，就可以通过隐式intent启动相应的组件。Activity和Service被未知应用启动，尤其是Service，应该还是挺不安全的。具体怎么影响安全性，有待深入分析。

但是通过给Activity和Service定义permission属性后，只要拥有相应permission权限的组件才可以开启Activity和Service。这样就增加了敏感Activity和Service的启动安全性。即使未知应用盗取了intent的action属性，也不能启动Activity和Service。

1. 所以说其实即使是隐式intent启动Activity和Service，可能也不是十分的不安全，只不过存在安全隐患。
   1. AIDL学习笔记
      1. AIDL产生的原因分析
2. 谷歌创造AIDL的目的是，简化进程间通信时client和server两端的代码编写量。下面分析到底有多少代码量。
3. 在同一个应用中使用BindService时，有以下两个问题是容易解决的：
4. 分析的前提条件：
5. 提供Service的应用叫A应用，需要调用A所提供服务的应用叫B。
6. 以Day12\_BindService项目进行分析。在MainActivity中绑定RecordService服务，获取RecordBinder对象，从而在MainActivity中调用RecordService中的有关方法。
7. 类包括：MainActivity，RecordService（录音服务），RecordService.RecordBinder（录音服务提供的IBinder实现类）,IService(对外暴露的接口)
8. 程序的包结构



1. 容易解决的问题
2. Service的显式绑定

|  |
| --- |
| Intent intent = **new** Intent(**this**, RecordService.**class**);  bindService(intent, **new** RecordServiceConnection(), *BIND\_AUTO\_CREATE*); |

1. IService对象在MainActivity中声明时，需要倒包。在同一个应用中，直接将com.day12.Binder这个包导入即可。

|  |
| --- |
| import com.day12.Binder.IService; |

1. 当外部应用B需要绑定A的服务时，上面两个在同一个应用中容易实现的问题，变得十分的复杂。
   * + 1. 首先，显式绑定A的Service是不可能了，因为红字部分的qualified类名不知道是什么。这里不知道的原因分两种情况：
          1. RecordService.class的包名就不知道
          2. RecordService.class的包名com.day12.BindService知道，但是当B调用A的这个类文件的时候，只知道A项目中文件的包名是没有意义的。因为B通过这个包名并不能调用到这个类文件，系统可能允许应用程序间互相调用类文件，但是在包名com前可能还需要加一些修饰才能够实现调用。这里并不知道com前的修饰是什么，并且每个品牌的手机前面的修饰可能还不一样，所以即使能够显式调用，也会存在与不同手机适配的问题。（感觉肯定可以跨应用调用，因为隐式调用只是相对于代码来说，底层系统实现调用的时候肯定也是拿到了类文件）。

但是，这里还是可以解决绑定问题的，通过隐式bindService()，依然可以在B应用中绑定A应用提供的RecordServcie（代码已经测试，可以实现），并可以拿到IBinder对象（注意是Ibinder对象，而不是想要的IService对象）。差别就在即使拿到了IBinder对象，父类引用指向子类对象，父类调用方法的时候只能调用父类IBinder中已经存在的方法，而不能调用子类对象IService中特有的方法，所以拿到IBinder对象没有实际意义。

|  |
| --- |
| Intent intent = **new** Intent(**this**, RecordService.**class**);  bindService(intent, **new** RecordServiceConnection(), *BIND\_AUTO\_CREATE*); |

* + - 1. 最难的部分其实是在B应用中将A传回的IBinder对象cast成IService对象。
         1. 首先，需要在B应用中（随便哪，需要bindService的地方）声明一个IService对象。声明对象就需要本包里没有类的对象，就需要导包，这个包怎么导，跟显式bindService一样，跨应用的时候，真正的包名是什么，没法知道。
         2. Aidl是将aidl文件放到B应用中，并且aidl文件的包名要和A应用相同。那IService接口定义文件也这样操作，因为虽然不在同一位置，IService接口的定义文件是同一个。这样在B应用bindService()的地方就可以用这个B项目中的IService接口文件声明IService对象。
         3. 到目前的状态就可以发布两个应用到模拟器，然后实现bindService()操作。
         4. 但结果是当B应用将A传回的IBinder对象强转成IService对象时报错。报错的原因是系统认为A项目中的IService接口和B项目中的IService接口不是一回事，所以不能将IBinder对象强转为B项目里定义的IService对象。
         5. 如何将这两个IService接口定义识别为一个，即相同的定义，就是开发aidl技术的目的。谷歌官方文档中认为这个操作过程过于繁琐，所以官方提供了aidl技术来实现，不需要编程人员自行实现。

1. 关于bindService的其他理解

由于RecordBinder是RecordService的内部类，如果B应用想倒这个内部类的包（假设可能），必须要有RecordService的类文件，等于A应用需要将Serivce服务完整的代码提供给第三方使用，太不安全。

这里需要指出的是，RecordBinder这个实现Binder类的内部类，是不可以从Service中拿出来单独定义的，因为如果从Service类中拿出来之后，这个Binder就跟其他非Service内部类一样，拿不到Service对象，从而无法调用Service中的方法。

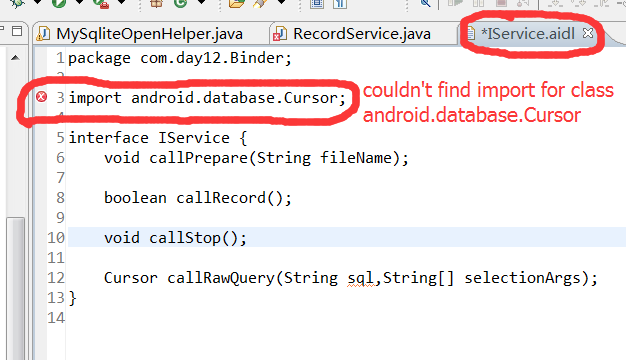
* + 1. AIDL的实现
  1. Service提供者需要执行的操作
     1. Service要提供intent-filter属性，供外部应用隐式调用该Service
     2. IService.java改扩展名为.aidl，这时系统会在gen目录下自动生成一个IService.java文件
     3. Service里的Binder类继承Stub类
  2. bindService的本地服务需要执行的操作
     1. 复制.aidl文件至本地服务的项目文件中，注意包名要和Service提供者的Iservice.aidl文件包名相同。
     2. 在ServiceConnection接口中获取IService对象的方式如下：

|  |
| --- |
| IService iService = Stub.asInterface(IBinder); |

* + 1. 注意一个问题，AIDL的实现过程导致本地服务和远程服务可能要通过Handler进行通信，或者其他，导致Service中不能有执行Toast的代码，不然会导致RemoteException。
    2. AIDL的限制

AIDL在实现IPC的时候，对传递的数据类型有要求，具体事项见后面。也就是说通过服务在线程间传递数据，即通过Service实现ContentProvider的功能，比较繁琐，需要对返回的数据类型进行重新封装（封装过程见后面），不能通过直接导包的方式声明一个类。比如想返回一个Cursor对象，会报异常：couldn't find import for class android.database.Cursor。所以说，应用自身暴露数据给外部，还是采用ContentProvider更简洁。

1. 用IPC实现ContentProvider功能的时候，需要IService接口提供相关的方法返回数据给执行bindService()的外部应用，这个返回数据的类型正常情况下需要import相应的包，但是导入相关包之后，再将.java文件改为.aidl文件时，IServcie接口的定义文件会出现下面的问题，



1. Android 中进程间通信使用了 AIDL 语言，但是支持的数据类型有限：
2. Java的简单类型（int、char、boolean等）。不需要导入（import）。
3. String和CharSequence。不需要导入（import）。
4. List和Map。但要注意，List和Map对象的元素类型必须是AIDL服务支持的数据类型。不需要导入（import）
5. AIDL自动生成的接口。需要导入（import）。
6. 实现android.os.Parcelable接口的类。需要导入（import）。

其中后两种数据类型需要使用import进行导入。

1. 解决这个错误的步骤如下：
2. 确保你的类已经实现了 Parcelable接口
3. 实现writeToParcel方法，将对象的当前状态写入 Parcel
4. 添加一个叫 CREATOR 的静态字段，它实现了 Parcelable.Creator
5. 最后创建一个对应的\*.aidl文件去声明你的Parcelable类
   * 1. 安卓5.0以后的使用要求
6. 5.0以后要求必须显式绑定服务。
7. 但是这里的显式绑定服务和5.0以前的显式绑定还不是一个概念。
   1. 需要setAction，只不过setAction的内容变了，原来是Service的action属性，现在是aidl接口的包名。

|  |
| --- |
| Intent intent = new Intent(Binding.this, RemoteService.class);  intent.setAction(IRemoteService.class.getName());  bindService(intent, mConnection, Context.BIND\_AUTO\_CREATE); |

1. Context
2. Android应用程序窗口在运行的过程中，需要访问一些特定的资源或者类。这些特定的资源或者类构成了Android应用程序的运行上下文环境，Android应用程序窗口可以通过一个Context接口来访问它
3. Activity
   1. Activity基本概念
4. Activity不是UI界面，Activity管理界面
5. Each view controls a particular rectangular space within the activity's window and can respond to user interaction.

也就是说每个控件所占的空间只能是矩形，即使在屏幕上看起来是各种形状。与用户的交互区域由屏幕上能够看到的区域决定；但控件摆放的时候，还是以矩形为边界，也就是说多个圆形（外观）控件应该是做不到相互外切的状态的。

1. you can also create new Views in your activity code and build a view hierarchy by inserting new Views into a ViewGroup, then use that layout by passing the root ViewGroup to setContentView()

setContentView(layout)的参数必须是viewGroup的子类。

1. The <category> element specifies that this activity should be listed in the system's application launcher (to allow users to launch this activity).
2. Your implementation of these lifecycle methods must always call the superclass implementation before doing any work, as shown in the examples above.

声明周期方法的回调是显式Activity目前的状态，而不是通过回调方法决定Activity的生命周期。当然，通过在回调方法中执行可以影响Activity生命周期的方法，也可以影响Activity的生命周期，但是回调函数的主要作用是在相应的生命周期状态到来时，进行相应的操作，而不是去影响生命周期本身。Fragment的生命周期回调方法同样的道理，只是显示生命周期的状态。

1. if your activity has a thread running in the background to download data from the network, it might create that thread in onCreate() and then stop the thread in onDestroy()。

从这里来看，谷歌认为Activity中开启的子线程就应该与该Activity同生命周期。

1. you can register a BroadcastReceiver in onStart() to monitor changes that impact your UI, and unregister it in onStop()
2. Because this state can transition often, the code in these two methods should be fairly lightweight in order to avoid slow transitions that make the user wait.

onStart方法、onStop方法感觉也应该lightweight，因为onStop方法执行完了，才会执行下一个Activity的onCreate方法。而且在界面切换的时候，这两个方法与onPause和onResume的频率也差不了多少。

1. Activity在任务栈中，虽然任务栈遵守“后进先出”的原则，但是系统在内存不足时，还是可以销毁被压在下面的Activity。
2. A paused activity is completely alive (the Activity object is retained in memory, it maintains all state and member information, and remains attached to the window manager)

A stopped activity is also still alive (the Activity object is retained in memory, it maintains all state and member information, but is not attached to the window manager)

是否是说Pause之后可以在window manager里获取activity？

* 1. 启动模式

1. 启动模式的目的不在于怎么启动Activity，而在于Managing Tasks.所以SingleTask是single的task，而不是back stack。Back stack不是直接被管理的，可能也不需要直接被管理，只要管理好task即可。
2. Activity启动模式的两种设置方式
3. 如果 Activity A 启动 Activity B，则 Activity B 可以在其清单文件中定义它应该如何与当前任务关联（如果可能），这种方式可以用来设置Activity默认的启动方式。
4. Activity A 还可以在启动 Activity B的时候动态设置B的启动方式。这种方式适用于根据具体情况临时决定启动方式。

如果以上两种情况均存在，则 Activity A 的请求（如 Intent 中所定义）优先级要高于 Activity B 的请求（如其清单文件中所定义）。

但是两种方式可以使用的启动模式种类不同。

1. Activity的启动模式
2. 概念
3. Single是对于一个Activity对象来说的。
4. Top，同一个Activity的多个对象，只有一个在顶部（启动该Activity的Activity所在的task）。
5. Task，这个Activity只能存在于一个task中，所以其实也是只能有一个对象。
6. Instance，这个Activity只能有一个对象。
7. Manifest能够声明的四种启动模式
8. Standard：

系统在启动 Activity 的任务中创建 Activity 的新实例并向其传送 Intent。Activity 可以多次实例化，而每个实例均可属于不同的任务，并且一个任务可以拥有多个实例。

1. singleTop：

如果当前任务的顶部已存在 Activity 的一个实例，则系统会通过调用该实例的 onNewIntent() 方法向其传送 Intent，而不是创建 Activity 的新实例。Activity 可以多次实例化，而每个实例均可属于不同的任务，并且一个任务可以拥有多个实例（但前提是位于返回栈顶部的 Activity 并不是 Activity 的现有实例）。

系统并没有销毁当前顶部的Activity实例，而是调用了该Activity的onNewIntent()方法，将intent传给Activity而已。

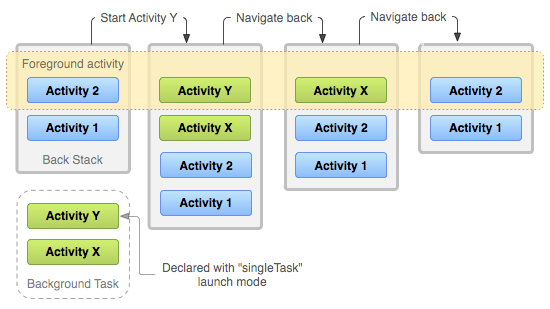
注：为某个 Activity 创建新实例时，用户可以按“返回”按钮返回到前一个 Activity。 但是，当 Activity 的现有实例处理新 Intent 时，则在新 Intent 到达 onNewIntent() 之前，用户无法按“返回”按钮返回到 Activity 的状态。

1. singleTask:

系统创建新任务并实例化位于新任务底部的 Activity。但是，如果该 Activity 的一个实例已存在于一个单独的任务中，则系统会通过调用现有实例的 onNewIntent() 方法向其传送 Intent，而不是创建新实例。一次只能存在 Activity 的一个实例。

尽管 Activity 在新任务中启动，但是用户按“返回”按钮仍会返回到前一个 Activity。

启动原理图



1. singleInstance:

"singleTask" 相同，只是系统不会将任何其他 Activity 启动到包含实例的任务中。该 Activity 始终是其任务唯一仅有的成员；由此 Activity 启动的任何 Activity 均在单独的任务中打开。

* 1. 任务和返回栈（Tasks and Back Stack）

1. 已确定的认知：
2. 启动图标的作用：
3. 没有task存在于background，创建新的task，启动ACTION\_MAIN对应的Activity
4. 有task存在于background，返回task。注意：这里并不去启动ACTION\_MAIN对应的Activity，仅仅是将已有的task转换到foreground。
5. 同一个应用打开的Activity（启动模式默认），不论是自己的还是别的App的，均保存在当前应用的task中。
6. Activity只能运行在自己所在application的进程中，所以考虑到第1）条，会可能出现这样一种情况，activity“保存”在别的App的task中，但是运行在自己所在应用的进程中。

疑问：保存在别的App的Task中的Activity，task中保存的是这个Activity对象的引用？如果是引用，那么等于将一个A应用的对象保存到B应用中，B中有A的Activity的引用，如果这样就需要AIDL了，底层通过AIDL实现的？

1. When the user leaves a task by pressing the Home button. 按“home”键的本质是将task放置于background。
2. 点击App的启动图标，当应用没有被启动的时候，本质是开启一个task；当应用已经启动的时候，本质是恢复task。
3. 一个应用可以同时拥有多个task

疑问：是不是同一时刻只有一个task是foreground，其他都是background？

1. A new activity is, by default, launched into the task of the activity that called startActivity(). It's pushed onto the same back stack as the caller.

默认启动模式下，新开启的Activity运行在开启它的Activity的task和back stack中。即使是A应用开启B应用的Activity，B应用的Activity也是运行在A应用的task和back stack中。

For example, suppose that an activity that reports weather conditions in selected cities is defined as part of a travel application. It has the same affinity as other activities in the same application (the default application affinity) and it allows re-parenting with this attribute. When one of your activities starts the weather reporter activity, it initially belongs to the same task as your activity. However, when the travel application's task comes to the foreground, the weather reporter activity is reassigned to that task and displayed within it.

红字部分证实了A开B，B的Activity的确是运行在A的task中。再考虑到第一段所说的，那么A开B，B的Activity还是被push到A的back stack的。

疑问：

如果B的Activity引用保存在A的task和back stack中，那么task和back stack中是运行在进程中，进程之间通讯是怎么实现的？

1. Task和Back Stack肯定不是一个东西，因为谷歌文档的名称就叫Tasks and Back Stack。
2. Even though the activities may be from different applications, Android maintains this seamless user experience by keeping both activities in the same task.

红字部分应该是说为了用户的无缝体验，即使不同应用的activity，如果是由同一个应用启动的，均会放在同一个task中。

1. A task is a cohesive unit that can move to the "background" when users begin a new task or go to the Home screen

关于这里的new task，singleTask的activity创建的时候，是创建了一个新的task？创建了一个新的back stack？还是其他？

那要是同一个应用，顺序启动activity的过程中，有一个activity的启动模式是singleTask，那这个新创建的task应该会来到foreground，之前的task就去到了background。

1. 任务是指在执行特定作业时与用户交互的一系列 Activity。 这些 Activity 按照各自的打开顺序排列在堆栈（即“返回栈”）中。
2. 返回栈中的 Activity 永远不会重新排列。（官方文档原话）
3. Activities in the stack are never rearranged, only pushed and popped from the stack

所以说不可能出现从back stack中抽取某一个Activity的情况。因为这个原因，single task启动模式，在调用所需的Activity的时候，才会清空其上面的Activity，这里还有一个原因，single task默认clean top，至于clean top怎么实现的，待看。

1. APP进程创建一个task，task中维护一个back stack（所以称作任务栈）。感觉Task是一个抽象的，从全局上控制一个App的Manager，对Activities的暂存是由back stack完成的。应用程序开启的时候，task被启动，task可能是用来维护Activities的，back stack只是存储Activities。

嵌入到Activity中的Fragment有自己的back stack，这个back stack由该Activity管理。

1. 同一个任务中的Activities可能是运行在不同应用的进程中。Activity只能够运行在自己所属应用的进程中。
2. 一个任务可以包含多个堆栈，一个堆栈也可以包含分属于多个任务的Activities。（待确认）
3. 不同启动模式下的task和stack分析
   1. A1和B1启动模式就是默认
      1. 在应用A的Activity A1中开启应用B的Activity B1，B1运行在A1的task和stack中，但是B1是运行在自己的进程中。（貌似不对）
   2. Fragment
      1. 基本概念
4. Fragment可以当成一个个Activity模块，这样当屏幕大小不同时，可以将几个模块放入同一个Activity，也可以将几个模块放入不同的Activities，这样就实现了模块之间的解耦操作。避免都写在一个Activity中，所含功能需要拆开的时候还要拆分代码。
5. Fragment继承的是object。所以没有context所提供的所有内容。Fragment感觉更像是个Manager，所以它继承的并不是View，而是Object，因为它不需要作为组件被显示在Activity上。被显示在Activity上的还是onCreateView方法返回的View对象。

Fragment在layout.xml文件中定义时，虽然它是Object的直接子类，但可以被当做一个View来设置各种属性，本质上还是因为<fragment>只是一个标签，实际运行的时候是由onCreateView创建的View对象来填充view hierarchy中fragment标签所占有的位置。

1. Fragment的主要作用：在同一个Activity中实现界面的动态转换。 there's more room to combine and interchange UI components.
2. Fragment切换的时候Activity重新加载了么？

应该是没有，因为Activity本身是可以动态对组件进行增删改查的。Activity不是UI界面，它管理UI界面，所以组件的变化是Activity正常的工作内容，不可能需要重建Activity。

1. 利用片段实现此类设计时，您无需管理对视图层次结构的复杂更改。
2. share events with the activity。

Activity和fragments共享事件。事件是怎么传递的？

1. When you perform such a fragment transaction, you can also add it to a back stack that's managed by the activity—each back stack entry in the activity is a record of the fragment transaction that occurred. The back stack allows the user to reverse a fragment transaction (navigate backwards), by pressing the Back button.

Fragment的back stack保存的是transaction，而不是fragment。而且可以通过返回键返回上一个transaction。微信就不行，说明可以将这个返回设置成无效。

Fragment的状态随Activity一起到task的back stack中。

1. Fragment的优势：（这里微信的例子有可能是不对的，主界面有可能是用ViewPager实现的）

拿微信主界面来说，完全可以通过多个Activity之间进行切换来实现。但是这样相比fragment实现，就会有几个不方便的地方：

* 1. 在task任务栈中需要压入更多的Activity，占用更多内存，响应速度应该也不如fragment快
  2. 横向滑动屏幕切换界面的时候，也需要重新开启、关闭Activity
  3. 多Activity间传输数据（如果需要）

所以说代码量应该多不少，而且结构不够清晰。

1. 特别注意一点：fragment和当前的Activity同时存在时，两者都是可交互的。Fragment虽然不是继承自View，但是应该进行什么处理了，所以可以实现两者同时可交互。此处的实现原理可以用来实现自定义组件与Activity的同时可交互。
2. Fragment的点击事件，是在fragment自身类内部定义的。listView的item点击事件，是在activity中定义的。
3. 同一个Activity中的多个fragment，通过activity进行交互。
4. Fragment对象运行在Activity中，即主线程。

Each set of changes that you commit to the activity is called a transaction and you can perform one using APIs in [FragmentTransaction](file:///C:\\Users\\cuihaichen\\AppData\\Local\\Android\\sdk\\docs\\reference\\android\\app\\FragmentTransaction.html)。一个事务可能要同时完成多个操作，如果中间断掉了，可能出现屏幕中各部分的内容不相关的情况，因为有可能有的fragment更新了，而有的没有更新，且这些fragment显示的内容是相关的。为了避免上述情况的发生，采用了事务的处理方式，保证一系列改变同生同死。

1. avoid directly manipulating one fragment from another fragment

说明fragment是可以直接互相操作的。怎么操作？

1. Fragment的生命周期和Activity的生命周期在Activity处于foreground的状态时，是相互独立的。Fragment生命周期的开始节点，取决于在哪里调用的Fragment（加载方式）：layout.xml文件中加载，动态加载的代码位置。动态的时候，因为代码有可能在任何Activity回调函数中执行（能不能显示到屏幕上不管），所以fragment的生命周期可以在任何时间节点开始。

At any time while your activity is running, you can add fragments to your activity layout.

* + 1. 代码实现

1. 3.0版本（API-11）之前如果使用fragment，注意两个类：Fragment Activity，Fragment（getSupportFragmentManage）
2. Fragment生命周期的任何回调方法，不会引发Activity的任何声明周期回调方法的执行。
3. FragmentTransaction.addToBackStack()方法是将transaction存入back stack中，这样在按手机的返回键时，不会返回上一个Activity，而是返回上一个fragment时Activity的界面。如果不在commit()前执行这句话，将直接返回上一个Activity。
4. Transaction执行commit()之后，要重新beginTransaction()。当前的transaction已经执行完成，不能再用了。
5. Your implementation of these lifecycle methods must always call the superclass implementation before doing any work。Fragment是这样，但Service里的父类方法可以不调用，目前来看好多都是空的。
   * 1. 待处理问题
6. 概念
   1. Fragment底层实现的原理？估计核心的是transaction的原理
   2. Fragment中的数据，在Fragment自己类中加载还是在Activity中加载？
   3. 嵌套使用Fragment的时候为什么会出现内存溢出？
      1. Analyses for fragment
7. How come transaction is being used for the changes on fragment?

Fragments allow such designs without the need for you to manage complex changes to the view hierarchy.

Fragment切换的时候，实际上是对Activity的view hierarchy进行修改，如果fragment中包含很多组件，更新view hierarchy是需要一个过程的，而且这个过程必须一次性完全完成，而不能执行一部分停掉了，如果中途停掉了，Activity的界面上fragment所在的位置，就会出现一部分组件显示，一部分组件不见了。归根结底，fragment也无非就是Activity上的一些组件，这些组件的替换直接用java代码也可以实现，只不过google帮你完成了transactiion这些比较繁杂的代码，从而简化view hierarchy更迭的实现步骤。

同一个Activity内的组件动态组合（增删改查），这个步骤不使用Fragment直接自己也可以实现（代码已验证）。但是如果当一次性需要替换的组件数量较多，同时要保证这些组件必须一起完成转换，不能一些转换了，一些没有。这时就有问题了，怎么保证所有转换都顺利执行？怎么在一部分没转换成功时，代码实现回滚，恢复原来的界面？Fragment的transaction感觉主要是干了这件很复杂的事儿。

* 1. Analyses
     1. Activity的启动及状态存取分析

1. 回调函数应该是Activity执行过程中设置了多个状态变量，用来表示执行过程的状态，然后在运行到相应状态时将变量赋值，系统随时监测状态值，然后回调相应的方法。
2. 前后两个Activity启动过程中回调函数执行顺序

it's important to understand that the first activity is not completely stopped before the second one is created. Rather, the process of starting the second one overlaps with the process of stopping the first one.

The order of lifecycle callbacks is well defined, particularly when the two activities are in the same process and one is starting the other. Here's the order of operations that occur when Activity A starts Acivity B:

Activity A's onPause() method executes.

Activity B's onCreate(), onStart(), and onResume() methods execute in sequence. (Activity B now has user focus.)

Then, if Activity A is no longer visible on screen, its onStop() method executes.

A应用的Activity调用B应用的Activity，两个进程同时处理各自Activity的生命周期。但执行顺序是相同的。只不过可能不会像上面的顺序那么严格，因为毕竟是两个进程各自执行自己的操作。

A应用的Activity开启另一个自己的Activity，是同一个进程处理的，严格按照上面的顺序执行。

由B返回到A时，也是一样的执行顺序。

1. 代码的执行位置
   1. 尽量使代码被执行的次数尽量少
   2. 代码尽量放在对用户来说看起来像是静态的操作中，用户能够感觉到动态变化的操作中，代码应尽量简洁。
2. 资源的释放
   1. 资源释放的基本原则
      1. when stopped, your activity should release any large objects, such as network or database connections. When the activity resumes, you can reacquire the necessary resources and resume actions that were interrupted.
      2. system can kill the process hosting the activity at any time after the method returns, without executing another line of the activity's code.

onPause() is the last method that's guaranteed to be called before the process can be killed. if the system must recover memory in an emergency, then onStop() and onDestroy() might not be called.

because any blocking procedures in this method block the transition to the next activity and slow the user experience.

the system calls onSaveInstanceState() before onStop() and possibly before onPause(), if user is not explicitly closing the activity.

Almost every widget in the Android framework implements this method as appropriate.

Because onSaveInstanceState() is not guaranteed to be called, you should use it only to record the transient state of the activity (the state of the UI)—you should never use it to store persistent data. Instead, you should use onPause() to store persistent data (such as data that should be saved to a database) when the user leaves the activity.

* + 1. 阻塞Activity生命周期中任意一个回调方法，都会影响Activity的相应速度，所以不仅这里，尽量做到在不易察觉阻塞的生命周期阶段进行稍微耗时的操作。
  1. 各回调方法所执行操作的分析
     1. 在onPause()方法中释放的资源：

这里释放的资源应该是轻量级的。因为下一个Activity的启动过程只有在当前Activity执行完onPause()方法后才会开始，所以onPause()的执行速度，很影响下一个Activity呈现给用户的速度。

onPause()方法执行后，有两种回调方法可能被执行。

1. 马上调用onResume()恢复显示

onPause()后由于用户的主动操作马上执行onResume()方法。这种可操作性极低，计算机运行速度极快。

1. 界面被下一个Activity覆盖，但是仍然可见

onPause()后启动的Activity并没有覆盖掉当前的Activity，当前Activity仍然是可见的。因为获取资源均是有些耗时的操作，这种仍然可见，只是被遮挡的Activity，可能对切换回原Activity速度的要求更高，这种Activity很可能被马上切换回来，所以也没有必要释放资源。

* + 1. 在onStop()方法中释放的资源：

Activities之间进行切换，当Activity变为不可见被压入栈后，如果还持有资源会长期占用资源，而此时这些Activities又不可交互，是对资源的严重浪费。onDestroy()方法只有Activity销毁的时候才回调，所以不能满足尽可能早释放资源的要求。所以说Activity一不可见，就应该立马释放资源，做到尽量少的占用系统内存和资源。

但是onStop()仍然不能操作重量级的资源。

* 1. 特殊情况：前后两个Activity共用的资源

当前后两个Activity共用相同资源时（前一个A，后一个B，A中启动B），比如读取同一个文件。B的onStart如果要想打开这个文件，一定要A先释放对这个文件的占用。所以B需要使用的资源，不能等到A的onPause()方法之后执行，一定要A在onPause()中释放。A的onPause()执行之后，B就会从onCreate~onResume一口气执行完，之后才是A执行onStop()方法，onPause()不释放资源就没机会了。

* 1. onSaveInstanceState()

存储当前Activity的状态（onSaveInstanceState(bundle)回调方法中保存除外）之所以要求在onPause()方法中执行，是因为onPause、onStop、onDestroy三个方法中任何一个执行完成后，系统就重获了销毁该Activity的权利（onCreate、onStart、onResume、onRestart四个方法执行完成后，系统不能销毁Activity）。onPause方法是第一个被回调的，也就是说onPause方法执行完成前，系统保证不会销毁该Activity，而一旦onPause执行完成，系统随时可以销毁该Activity，只有在onPause方法中保存当前Activity状态，才能保证一定可以完成对状态的保存。

系统在销毁Activity的时候，不会再执行任何一句Activity内的代码，即onStop、onDestroy方法有可能就不会被执行，而是进程直接被销毁。

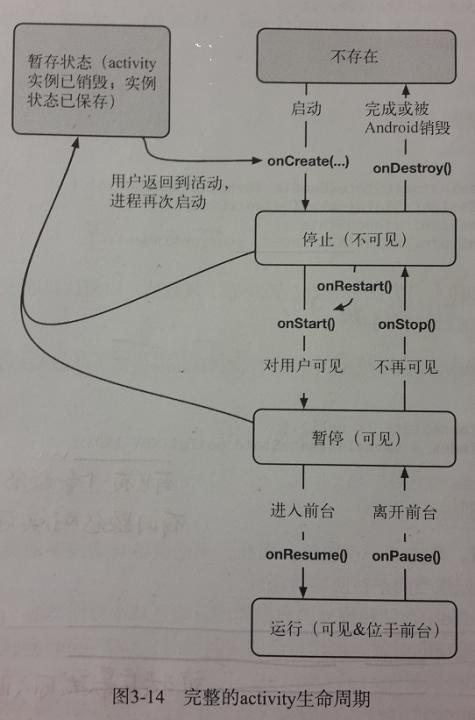
但是保存当前Activity状态时，要尽量控制数据量，因为这些回调方法执行完成后，才会回调下一个Activity的生命周期方法。

保存状态是为了在系统销毁进程，重新调用onCreate方法创建Activity时使用。正常onPause、onStop，Activity的状态随Activity本身一起被放入了任务栈中。所以其实需不需要保存，看实际需求。

如果用户正常销毁Activity，onSaveInstanceState中保存的状态就会被丢弃。如果系统为了腾出内存空间，销毁Activity，保存的状态不会在销毁时执行。在哪个回调方法中被丢弃？

onSaveInstanceState()方法，除了Activity用户主动销毁Activity的情况，均会被调用

之前分析说onPause的时候存储数据，并且此时系统也会调用onSaveInstanceState(Bundle)方法对Activity的状态进行保存。但是看下面这张图，onCreate、onStart两个回调方法执行完成时，用户其实也是可能在此处退出Activity的，此时如果退出Activity，前一次存储的Bundle对象已使用，这次又没有调用onSaveInstanceState(Bundle)方法，再回来的时候，Bundle是空的，就没办法恢复前一次Activity的状态。



但这里要搞清楚两件事：

1. Activity退出的情况
   1. 正常情况
      1. 用户主动退出
   2. 非正常情况
      1. 腾出内存空间，系统杀死进程
      2. 设备配置改变，比如屏幕旋转
2. onCreate和onStart方法执行完成时，为什么Activity会退出？
   1. 用户主动退出：不需要用Bundle存储数据
   2. 设备配置改变：需要用Bundle存储数据
3. 设备配置改变的时候，比如屏幕旋转，怎么存储数据？

在onCreate、onStart中存储数据？

这样可以有待代码测试。

但是如果存储的数据较多，每次启动时感觉会影响Activity的响应速度。而且因为cpu速度很快，一般情况下也不会正巧赶到这两个状态的时候设备配置改变。

|  |
| --- |
| **protected** **void** onCreate(Bundle savedInstanceState) {  **Int currentIndex = 5;**  savedInstanceState.putInt(currentIndex);  **super**.onSaveInstanceState(savedInstanceState);  ……  ……  } |

1. Content Provider官方文档笔记（待续）
   1. 基本内容
2. Two parts: content provider and provider client
3. You don't need to develop your own provider if you don't intend to share your data with other applications. However, you do need your own provider to provide custom search suggestions in your own application. You also need your own provider if you want to copy and paste complex data or files from your application to other applications.

程序内部搜索数据、转移文件给其他应用（不是封装好的对外接口），有可能用到provider。搜索为什么要用到provider？

1. Android itself includes content providers that manage data such as audio, video, images, and personal contact information.

Audio, video, images, personal contact, sms这些数据，Android均提供了ContentProvider。也就是说对于这些数据，系统默认应用也不是直接操作数据，而是通过ContentProvider间接操作。

1. With some restrictions, these providers are accessible to any Android application.
   1. 源码笔记
2. Observable
3. Observer
4. ContentObservable
5. ContentObserver
6. DataSetObservable
7. DataSetObserver
8. ContentResolver
9. 注册observer

|  |
| --- |
| getContentResolver().registerContentObserver(uri, notifyForDescendents, observer); |

这句话的意思就是将observer注册到相应的observable的ArrayList中

1. notifyChanged()方法，实际上并没有使用Handler或其他传递消息的对象来将消息发送给observer。见下面的源码，notifyChanged()方法中，直接调用了observer的onChanged()方法。安卓中很多感觉像是单独启动了一个子线程，持续监听某种东西，然后回调函数的对象，其实都不是，只不过是将注册的对象直接传给了发出消息或接收事件的对象A，当A执行发送消息或处理事件的时候，在自己类的代码中直接调用了注册对象的方法。

回调方法是从被注册对象的角度起的名字，实际回调方法也是主动被调用的，只是在注册对象重写回调方法时，感觉起来像是别人执行了什么代码之后，这个方法被回调了。

回调方法可以理解为B对象执行代码的过程中，需要调用A的方法，但是这个方法并不在B对象所执行的代码所在的位置进行定义，而是在任何地方都可以进行定义，并通过某个方法，将这个A对象传给B，这样B代码中就等同于写了一段A方法的定义。安卓中采用传对象的方式实现上述方法调用的方式比较普遍。

|  |
| --- |
| class DataSetObservable extends Observable{  public void notifyChanged() {  synchronized(mObservers) {  // since onChanged() is implemented by the app, it could do anything, including  // removing itself from {@link mObservers} - and that could cause problems if  // an iterator is used on the ArrayList {@link mObservers}.  // to avoid such problems, just march thru the list in the reverse order.  for (int i = mObservers.size() - 1; i >= 0; i--) {  mObservers.get(i).onChanged();  }  }  }  } |

1. Service学习笔记
   1. 基本内容
2. 进程是否被从内存中清理与与进程当前的级别有关，也就是说后台运行的Service是否会被断掉，跟进程相关。但是Service应该可以通过设置自己的运行状态，间接的影响进程的运行级别，从而达到保证自己不会被中途断掉。
3. A service runs in the main thread of its hosting process—the service does not create its own thread and does not run in a separate process (unless you specify otherwise).

服务工作于当前应用所在进程的主线程中，一般情况下并不会去新建一个进程或线程。也就是说应用程序的进程结束，服务也结束。

1. 服务在manifest中注册的时候，有两个属性

exported：决定外部应用是否可以start或bind该服务；完全阻止发布应用开启该服务，即使显示开启。

isolatedProcess：服务可以通过设置这个属性，单独为自己开启一个进程。

1. To ensure your app is secure, always use an explicit intent when starting or binding your Service and do not declare intent filters for the service.

为了程序的安全，尽量不要去给Servcie声明intent-filter。

1. A services runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.

Service运行在主程序所在的进程的主线程中，Service中的耗时操作或卡顿，会影响Activity与用户的交互体验，这时应在服务中开启子线程。这里也是说明为什么需要放到后台执行的操作没必要均采用子线程，因为非耗时操作完全可以交给Service执行。这样带来的好处是Service可能能够提供更好的应用程序Context。

1. 两种Service实现方式
   1. Service

This is the base class for all services. When you extend this class, it's important that you create a new thread in which to do all the service's work, because the service uses your application's main thread, by default, which could slow the performance of any activity your application is running.

按照这里的说法，如果我们创建Service的时候直接继承Service这个类，上来就要开启子线程。

* 1. IntentService

This is a subclass of Service that uses a worker thread to handle all start requests, one at a time. This is the best option if you don't require that your service handle multiple requests simultaneously. All you need to do is implement onHandleIntent(), which receives the intent for each start request so you can do the background work.

当不需要执行多线程并发操作的时候，继承这个类实现Service更简洁。比如只是执行单一的现在操作。并且在onHandleIntent()方法中代码执行完成后，IntentService会自动关闭服务。

创建线程（worker thread）和关闭服务都由系统自动完成。

onHandleIntent()方法中不需要创建子线程，该方法已经在worker thread中。

1. 同时在多处或在一处多次开启服务，oncreate()方法只调用一次，所以我感觉应该只有一个Service对象，但是onStartCommand()方法会被多次调用，如果里面创建了子线程，那其实就相当于开启了多个服务主体。如果在服务中开启了子线程，那关闭服务并不能关闭子线程，关闭服务就变得没有意义了。

当多个component分别开启同一个服务时，传给Service的intent所携带的数据可以是不相同的，这样如果onStartCommand()中开启子线程，多次调用onStartCommand()方法可以针对不同的intent分别开启线程执行相同的操作。相应的，多线程并发所带来的问题当然也要进行考虑。

1. 官方继承Service实现自定义服务类的示例代码中，通过将用户请求放到Looper对象中，然后通过handleMessage()方法实现多任务队列顺序执行。子线程创建Handler对象，接收主线程的message。好像很多地方都是子线程接收主线程发来的message。

同时官方提到，也可以通过在每次请求服务时，均创建子线程的方式，实现多任务并发执行。

1. If the service does not also provide binding, the intent delivered with startService() is the only mode of communication between the application component and the service. However, if you want the service to send a result back, then the client that starts the service can create a PendingIntent for a broadcast (with getBroadcast()) and deliver it to the service in the Intent that starts the service. The service can then use the broadcast to deliver a result.

不使用bindService()时，intent是Service和开启Service的组件之间唯一的数据传输途径，并且只能是组件将数据传递给Service。如果想要Service返回数据，需要把intent换成pendingIntent，并传送给Service。然后Service通过发广播的形式，将数据返回给组件。

1. 无论调用多少次onStartCommand()方法，关闭服务的时候，均只需要调用一次stopSelf()或stopService()。
   1. Should you use a service or a thread?
2. A service is simply a component that can run in the background even when the user is not interacting with your application. Thus, you should create a service only if that is what you need.

If you need to perform work outside your main thread, but only while the user is interacting with your application, then you should probably instead create a new thread and not a service. For example, if you want to play some music, but only while your activity is running, you might create a thread in [onCreate()](file:///C:UserscuihaichenAppDataLocalAndroidsdkdocsreferenceandroidappActivity.html#onCreate(android.os.Bundle)), start running it in [onStart()](file:///C:UserscuihaichenAppDataLocalAndroidsdkdocsreferenceandroidappActivity.html#onStart()), then stop it in [onStop()](file:///C:UserscuihaichenAppDataLocalAndroidsdkdocsreferenceandroidappActivity.html#onStop()). Also consider using [AsyncTask](file:///C:\\Users\\cuihaichen\\AppData\\Local\\Android\\sdk\\docs\\reference\\android\\os\\AsyncTask.html) or [HandlerThread](file:///C:\\Users\\cuihaichen\\AppData\\Local\\Android\\sdk\\docs\\reference\\android\\os\\HandlerThread.html), instead of the traditional [Thread](file:///C:\\Users\\cuihaichen\\AppData\\Local\\Android\\sdk\\docs\\reference\\java\\lang\\Thread.html) class. See the [Processes and Threading](file:///C:UserscuihaichenAppDataLocalAndroidsdkdocsguidecomponentsprocesses-and-threads.html#Threads) document for more information about threads.

Remember that if you do use a service, it still runs in your application's main thread by default, so you should still create a new thread within the service if it performs intensive or blocking operations.

1. A *service* is a component that runs in the background to perform long-running operations or to perform work for remote processes.
2. Because a process running a service is ranked higher than a process with background activities, an activity that initiates a long-running operation might do well to start a service for that operation, rather than simply create a worker thread—particularly if the operation will likely outlast the activity. For example, an activity that's uploading a picture to a web site should start a service to perform the upload so that the upload can continue in the background even if the user leaves the activity. Using a service guarantees that the operation will have at least "service process" priority, regardless of what happens to the activity. This is the same reason that broadcast receivers should employ services rather than simply put time-consuming operations in a thread.
   1. unbindService()分析
3. 理解这个问题首先要理解以下几点：
   1. Java基础：一个类的对象，只要有变量在引用，就会一直存在于堆内存中，就不会被垃圾回收。
   2. Service类也是类，不论Service对象是系统生成的还是new出来的，也只是符合java语言规则的类对象。仅仅是生成的方式、功能和地位与普通类不同。
   3. unBindService()方法执行之后，Service的onDestroy()方法就会跟着执行（注1）。
   4. 重写onDestroy()方法的时候会调用父类的onDestroy()方法，父类的方法中没有任何代码（因为系统也不清楚你要释放什么资源，Service引用了什么资源只有你自己知道）。
   5. onDestroy()方法的目的：释放该Service对象占用的各种资源。注意，不是使Service这个对象被垃圾回收。被回收的前提是没有任何一个变量指向该Service对象。
4. Service服务里没有引用任何资源

这种情况下，即使unbindService()方法被执行，Service回调了自身的onDestroy()方法，因为onDestroy()方法里什么代码都没有，相当于没有释放任何资源，因为本来什么资源也没引用。

但是这时myBinder变量还在引用Service提供的Binder对象，所以Service对象并没有被垃圾回收。这时用myBinder调用Service里的方法，必然是可以调到的，因为Service对象还存在。

这里之所以出现这个问题，感觉是因为Service就不该被这么使用，这么大一个组件，都不需要用到任何资源，不如直接写个方法算了。

1. Service服务里引用了资源

这种情况下，unbindService()方法被执行，Service回调了自身的onDestroy()方法，释放了Service所引用的资源（注2）。

与“第二个条目”中一样，由于myBinder对象仍然引用着Service提供的Binder对象，Service对象并没有被回收，仍然存在于堆内存中。

不同在于，如果这个时候通过myBinder对象调用Service中的方法，同时这个被调用的方法引用了刚才在onDestroy()方法中释放的资源，那系统就会报错。

所以为了安全，unBindService()的下一行代码，应该将myBinder对象对Service提供的Binder对象的引用注销，myBinder = null，这样才是安全的。

注1：同一个Service可以被绑定数次，只有所有绑定都解除之后，Service才会执行onDestroy()方法。

注2：释放资源的代码应该被放在onDestroy()方法中。

引申：所有对象都没有在堆内存真正销毁自己的能力，将堆内存中的对象销毁，只能由垃圾回收完成。

* 1. 其他

1. getSystemService()是获取了底层的Service对象，不是bindService。
   1. 疑问
2. Activity启动一个Service，然后Activity销毁了，再进来的时候启动的还是同一个Service么？比如播放音乐，如果中途Activity销毁了，再进来重新启动Service（start）的时候，会不会让音乐重新开始播放？

音乐播放是混合启动Service，只看bindService，因为Serivice中途没有停止，所以bind之后，调用相关方法是继续控制当前的音乐，不会带来其他的问题。

1. Broadcast Receiver
   1. 基本概念
   2. 系统广播的特点
      1. 短信广播
2. Action： android.provider.Telephony.SMS\_RECEIVED
3. 系统smsProvider在收到短信时，首先发送广播，在收到最终广播结果后才会将短信保存到数据库中。所以只要在其他广播接收者中将该广播取消，系统就无法收到最终的广播（系统的应该是优先级比较低，不是finalReceiver）。但是如果采用内容观察者监听短信，是在监听到短信数据库改变之后做出反应，所以需要再利用内容观察者对数据库进行需要的操作。
   * 1. 电话广播
4. Action： android.intent.action.NEW\_OUTGOING\_CALL
5. 系统自带拨号软件在用户按下拨打键之后，首先发出一个广播（有序广播），然后自己是finalReceiver，当自己收到最终的广播时，执行拨打电话的代码。所以用户可以在中途修改广播所携带的数据。
   1. Analyses
      1. Receiver工作进程分析

Receiver工作在当前应用的主进程中，而且并没有开启其他分支进程。当主线程被销毁时，Receiver的确不工作了。但是当广播到来时，Receiver会被唤醒，同时所在应用的主进程也会被重新开启。Receiver的生命周期应该是指在接收到广播后，从回调函数开始执行到回调函数结束的这段时期，而不是从接收广播开始。当主进程被销毁后，Receiver的所有生命周期毁掉方法均不能被执行，处于死亡状态。接收广播应该是由系统完成的，所以Receiver并没有工作在其他进程，而是由于Receiver在manifest中注册了，所以当有广播发出时，系统自动检索所有应用的manifest清单，判断是否有人需要接受这个广播。当检测到Receiver接收当前广播时，系统自动开启包含这个Receiver的应用的进程，然后在该进程中，Receiver的各个回调方法被执行。

1. View
   1. 基本概念
2. Window, display, frame , surface, view都是什么？
   1. 相互之间的关系(由底层到顶层)

View->window->window manager->display

* 1. Display：

Provides information about the size and density of a logical display.

The application display area specifies the part of the display that may containan application window, excluding the system decorations. The application display area maybe smaller than the real display area because the system subtracts the space neededfor decor elements such as the status bar. Use the following methods to query theapplication display area: getSize, getRectSize and getMetrics.

The real display area specifies the part of the display that contains contentincluding the system decorations. Even so, the real display area may be smaller than thephysical size of the display if the window manager is emulating a smaller displayusing (adb shell am display-size). Use the following methods to query thereal display area: getRealSize, getRealMetrics.

A logical display does not necessarily represent a particular physical display devicesuch as the built-in screen or an external monitor. The contents of a logicaldisplay may be presented on one or more physical displays according to the devicesthat are currently attached and whether mirroring has been enabled.

Window是在Display中创建，Display又包含在手机真实屏幕里，Display并不是指physical display。

* 1. Window：

1. DecorView和window不是Activity对象创建的，viewRootImpl是Activity对象创建的。
   1. 参数
2. Layout\_\*\*\*的参数，均是限制性参数，表示的是父容器对自己的限制。比如layout\_width，表示的是该控件在这个父容器中，宽度被限制在layout\_width数值之内。
   1. 事件分发
3. dispatchTouchEvent()方法: 实现类ViewGroup, View
   1. 事件分发分发的是MotionEvent，不同的View可以有不同的处理方法，同一个View也可以有多种处理方法。所以ViewGroup和View均实现了该方法。
   2. 系统自带布局使用的是ViewGroup的dispatchTouchEvent()方法。
   3. 事件分发不仅仅指将事件在一堆View中进行分发，也指单个View将事件在多个方法中进行分发。
   4. 即使已经确定是哪个View拦截事件，事件还是需要经过dispatchTouchEvent()方法进行分发，事件被拦截之后，影响的只是onInterceptTouchEvent()方法的调用。
4. RequestDisallowInterceptTouchEvent()方法: 实现类ViewGroup
   1. 这个方法只是去设置ViewGroup的标志位，这个标志位表示ViewGroup拦截不拦截事件。这个方法并不会直接关联到onInterceptTouchEvent()方法。两者的关联是通过dispatchTouchEvent()方法实现的，因为在这个方法中，有如下一段代码：

|  |
| --- |
| final boolean disallowIntercept = (mGroupFlags & FLAG\_DISALLOW\_INTERCEPT) != 0;  if (!disallowIntercept) {  intercepted = onInterceptTouchEvent(ev); |

执行onInterceptTouchEvent方法之前，会判断FLAG\_DISALLOW\_INTERCEPT标志位。

这里有几点需要说明：

1. ViewGroup的标志位mGroupFlags中对应FLAG\_DISALLOW\_INTERCEPT的位，默认是允许ViewGroup去拦截事件的，只不过是允许它拦截，而它在拦截之后执行onInterceptTouchEvent(ev)方法的时候直接返回了false而已。
2. FLAG\_DISALLOW\_INTERCEPT标志位是别人允许不允许你拦截
3. onInterceptTouchEvent(ev)方法是自己要不要拦截
   1. 子 View可以去干涉自己所在的父容器是否拦截事件，子View不允许父容器拦截，父容器就没法拦截。但是有个特例，子View无法干涉父容器拦截down事件，down事件一定能够传入到父容器的onInterceptTouchEvent(ev)方法中。
4. onInterceptTouchEvent()方法: 实现类ViewGroup
   1. ViewGroup：所有的布局均继承了ViewGroup的onInterceptTouchEvent()方法，该方法默认返回false，仅当用户采用mouse类的设备操作屏幕时，返回true。所以正常触摸屏均是返回false。
   2. 不同View拦截的触摸事件类型是不同的，具体拦截什么类型的触摸事件由onInterceptTouchEvent()方法确定。这里的事件类型不是指MotionEvent的事件类型，而是用户的触摸事件，比如FLING、SCROLL等。也就是说父容器所拦截的触摸事件类型是有数的，如果子View所需要处理的触摸事件类型父容器并不拦截，那子View也没有必要去请求父容器不要去拦截事件。只有当两者需要处理相同类型的触摸事件的时候，才有请求的需求。
5. onTouchEvent()方法: 实现类View
   1. 如果返回false，那么这次的一系列事件就不会交给该控件处理。当该控件返回false时，系统会将事件一步一步上传给更高一级的控件，即使最后没有任何控件处理该事件（包括最终的Activity），这些一列事件也就作废了。当下次的另一系列事件到来时，该控件同样能够得到DOWN事件，如果这次的DOWN事件该控件处理了（返回true），那么这一些列事件就会被交给该控件处理。 前一次一系列事件是否处理，与下一次一系列事件到来时该控件会不会被分发到事件没有关系，是相互独立的。而且前一次一系列事件最后没有人处理，系统也不会阻塞在这里，而不进行下一次事件序列的分发。
6. ViewGroup的dispatchTouchEvent()方法的实现
7. 分发的顺序
   1. Window接收到MotionEvent，然后回调CallBack中的dispatchTouchEvent()(该接口由Activity实现)，即表面上事件分发是从Activity的dispatchTouchEvent()开始。
   2. Activity的dispatchTouchEvent()方法调用Activity中的成员变量mWindow的superDispatchTouchEvent()方法，这个方法中将事件传递给了DecorView，由DecorView.dispatchTouchEvent()开始继续分发事件。
8. 事件分发滑动和点击判断的执行逻辑：
   1. View和ViewGroup判断代码执行的位置不同
      1. View是在onTouchEvent或onTouch中，因为它没有onInterceptTouchEvent方法。
      2. ViewGroup是在onInterceptTouchEvent方法中。
   2. 因为判断是否是滑动不可能在down事件这个单独的事件中完成，所以子View的dispatchTouchEvent、onTouchEvent方法一定会被执行。子View的这两个方法直到父容器确定是滑动之后，才会停止执行。
   3. View的布局流程
      1. 基本概念
9. View的MeasureSpec的确定：

这里的View包括View和ViewGroup，但是均把两者看成是最底层的View进行测量。所以MeasureSpec是用来测量最底层View的大小的。

这时的MeasureSpec的创建，关系到父容器的规格（包括父容器的布局参数和大小，特别注意：与父容器的布局参数也有关）、父容器的padding参数、View自己的layout参数。

有一点需要明确，父容器执行getChildMeasureSpec()方法生成传递给child的MeasureSpec的时候，MeasureSpec的mode，由父容器的布局参数和子View的布局参数共同决定。

1. measure方法只有View类有，所以所有控件的measure方法都是执行父类View的方法。ViewGroup类没有重写onMeasure()方法，应该是因为没有直接用基类ViewGroup做容器的，所以并没有提供自身的测量方法。也就是说如果我们需要直接继承ViewGroup类写一个自定义的layout，那么容器自身的测量方法需要重写。
2. View不会去创建MeasureSpec，正常的顺序是ViewGroup创建每个view的MeasureSpec，然后传递给子view，子view拿过来作为参考，去完成自己宽高的测量。MeasureSpec并不是子view的最终宽高，只是一个限制值，所以子view不会自己创建MeasureSpec，限制值应该是父容器来规定。
3. onMeasure()方法执行完，得到的mMeasuredWidth和mMeasuredHeight的确和控件最终的尺寸一样（极大部分的情况）。但是onMeasure测量过程中的widthSize和heightSize的尺寸是实时调整的，直到最终测量完赋值给mMeasuredWidth和mMeasuredHeight。
   * 1. 执行方法流程表

|  |  |  |  |
| --- | --- | --- | --- |
| 步骤(向下执行) | 语句所在位置(类) | 方法执行者(类) | 方法执行者(对象) |
| performTraversals() | ViewRootImpl | ViewRootImpl | ViewRootImpl |
| performMeasure() | ViewRootImpl | ViewRootImpl | ViewRootImpl |
| mView.Measure() | ViewRootImpl | View | DecorView(mView) |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

* + 1. View的测量过程

1. 在测量View的时候，即使View的宽高是Exactly，也会去测量View，然后在resolveSize()方法执行的时候再考虑View本身的布局参数，也就是测量过程是不考虑mode的。
2. onMeasure()方法参数的说明
   1. MeasureSpec：是父容器根据当前调用onMeasure()方法的view的布局参数、父容器的规格尺寸、父容器的padding参数计算得来的。所以这里MeasureSpec.getMode()得到的就是view自身的布局参数。
3. onMeasure()方法的逻辑
   1. 首先根据内容计算控件的期望宽高（三种取最大值）：
      1. Src属性drawable 图片的宽高
      2. Background属性drawable 图片的宽高
      3. minWidth和minHeight属性设置的控件最小宽高
   2. 然后使用父容器传过来的ChildMeasureSpec，对上面生成的期望宽高进行裁剪等操作，得到差不多是最终的尺寸，对应方法：

resolveSizeAndState();

* 1. 将上面确定的尺寸赋值给view的保存尺寸的成员变量。最后这个过程中会考虑viewgroup的bound修饰，对应方法：

setMeasuredDimension();

1. resolveSize()的执行逻辑：

根据自己的MeasureSpec的mode，以及这步之前获取的view实际的宽高，最终确定view的宽高。

还有一个方法resolveSizeAndState()；

1. View的测量过程可能需要重复好几次。
2. This is called to find out how big a view should be. The parent supplies constraint information in the width and height parameters. The actual measurement work of a view is performed in onMeasure(int, int), called by this method. Therefore, only onMeasure(int, int) can and must be overridden by subclasses.

measure()方法只负责view大概的大小，onMeasure()方法用来获取精确的大小，所以需要具体的View重写onMeasure()方法。

父容器提供的参数只是限制性参数。但是父容器提供的参数是限制性的，onMeasure()方法怎么得到精确的测量值？

1. View的测量过程执行顺序（以width测量进行分析）
   1. DecorView的layout\_width属性为MATCH\_PARENT

这种情况下，ViewRootImpl在执行getRootMeasureSpec()方法的时候，获得的是DecorView的最终宽度，所以传给performMeasure()方法的widthMeasureSpec参数是确定的。

* 1. DecorView的layout\_width属性为WRAP\_CONTENT

这种情况下，ViewRootImpl在执行getRootMeasureSpec()方法的时候，获得的不是DecorView的最终宽度，而是告诉系统，DecorView最宽为这个测量值，所以传给performMeasure()方法的widthMeasureSpec参数并不是最终的宽度，也就是说子元素在测量自己的时候，并不能根据这个值确定自己最终的宽度。

执行顺序

|  |
| --- |
| 08-14 03:01:10.752: onCreate  08-14 03:01:10.827: MySwitchButton(Context context, AttributeSet attrs, int defStyleAttr)  08-14 03:01:10.830: MySwitchButton(Context context, AttributeSet attrs)  08-14 03:01:10.830: onStart  08-14 03:01:10.830: onResume  08-14 03:01:10.901: onMeasure  08-14 03:01:10.901: onMeasure  08-14 03:01:11.232: onMeasure  08-14 03:01:11.232: onMeasure  08-14 03:01:11.233: onLayout  08-14 03:01:11.484: onMeasure  08-14 03:01:11.484: onMeasure  08-14 03:01:11.484: onLayout  08-14 03:01:11.485: onDraw |

* + 1. View的工作流程总结

1. ViewGroup：

无Measure方法，直接继承View的，measure方法中调用onMeasure方法。Measure()方法接收到的MeasureSpec是由上一级容器提供的。

1. Layout

确定ViewGroup的位置

1. onLayout

确定ViewGroup子元素的位置，所以具体布局需要重写该方法，从而实现对子View位置的确定。

1. onMeasure（获取当前View或ViewGroup的大小尺寸）

调用measureChildren() --> measureChild() --> child.measure()，在子View均测量完成之后，同时得到了ViewGroup的大小

1. RelativeLayout：
2. 重写了onMeasure方法，用来对自己的大小进行测量
3. View：
4. Measure()：public final修饰，没法重写，可以直接调用测量宽高

对父容器传过来的measureSpec进行了一定的处理，设置了一些标志位。View的测量工作是在onMeasure中完成的。Measure方法中没有进行测量，只是准备工作。

1. onMeasure()

具体测量View的大小

1. ViewGroup:
2. dispatchTouchEvent

调用了如下方法：

1. onInterceptTouchEvent
2. 根据情况调用子View的dispatchTouchEvent或者调用自己父类的dispatchTouchEvent
3. ViewGroup本身没有onTouch和onTouchEvent方法，其继承父类View获得这两个方法
4. View:
5. dispatchTouchEvent

顺序调用了onTouch和onTouchEvent方法。

1. onTouch

监听回调

1. onTouchEvent

事件处理

* + 1. 观察到的现象

1. Button不论宽高设多少，好像系统都会进行过滤，然后只能最大显示到屏幕宽度，控件边界不会出屏幕。但ImageView不会，不仅控件边界，控件内的src图片，也会出边界。
   * 1. 疑问
2. ViewRootImpl, WindowManager, Window, DecorView之间的关系是下面哪一个？

Window -> WindowManager -> ViewRootImpl -> DecorView

WindowManager -> ViewRootImpl -> Window -> DecorView

1. 网络编程
   1. Socket
2. A socket API is an application programming interface (API), usually provided by the operating system, that allows application programs to control and use network sockets.

Socket~~仅仅~~是一个file descriptor，Socket API是操作系统提供给用户的接口函数，用来操作Socket这个文件。

1. Socket是由网络协议抽象定义的，不同的操作系统可以提供不同的API去实现Socket，但因为各家实现的Socket符合网络协议的定义，所以各操作系统之间就可以通过Socket实现通讯。所以Socket在协议中一定是有定义的。
2. They are treated as different sockets by the operating system, since the remote socket address (the client IP address and/or port number) are different; i.e. since they have different socket pair tuples.

操作系统是以socket pair tuples识别本地socket是否相同，所以即使同一个Server产生的sockets的本地IP和port相同，这些socket对于操作系统也是不同的。因为本地socket对应的远程地址不同，即client socket的地址不同。

1. Network Socket是实现进程间通讯的一种方式，只不过这里的进程指的是网络上不同主机上的多个进程之间。本质上是一个file descriptor。

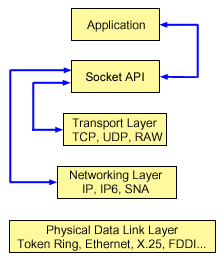
a file descriptor (file handle) in the Unix philosophy that provides a common interface for input and output to streams of data.

1. Socket是在会话层实现的一个接口(见wiki百科OSI模型)，是一个Library，最早的库叫做Berkeley sockets，1983。

The socket is primarily a concept used in the Transport Layer of the Internet model. （wiki，network socket）

但是在这里说socket是传输层的概念。

1. Socket并没有实现OSI模型中的低层（物理层和数据链路层）和高层（表示层、会话层和应用层）。所以基于Socket的编程，开源框架均是在其基础上实现Http协议，即完成应用层的搭建。
2. Client/Server模型中，表面上Server端的Socket地址是IP+Port，但连接到同一个Server的多个client，server提供的socket是不同的，并且每个socket均创建在server的一个子进程中，对于操作系统来说，这个socket是不同的。
3. 一个进程可以创建128个发送socket和128个接收socket。
4. The socket APIs are located in the communications model between the application layer and the transport layer. The socket APIs are not a layer in the communication model. Socket APIs allow applications to interact with the transport or networking layers of the typical communications model. The arrows in the following figure show the position of a socket, and the communication layer that the socket provides.Position of a socket in the communication layer。



1. 设计模式
   1. Android程序的基本框架
2. Andriod应用程序的整体框架就是一个MVC模式。Activity承担了部分Model（业务逻辑，不包括数据）和部分Controler（ListView有自己的controler）的角色。View就是所有界面上的View。
3. ListView自己是一个MVC框架，对于ListView来说，Activity才单纯是model。
4. 在android中，业务是由事件驱动的。
   1. 业务逻辑
5. Activity需要完成的工作：

业务逻辑、数据加载、View之间的协调

1. 联网加载数据，加载后的回调实现
2. 响应事件
3. 作为View之间的桥梁，统一View之间的展示
4. 初始化View
   1. 设计模式的类型
5. 创建型模式

创建对象的同时隐藏创建逻辑

* 1. MVC设计模式

1. Model：业务逻辑和数据，并不只是数据。所以Activity属于model，因为Activity实现业务逻辑的控制。业务逻辑实际上是在Activity中实现的，比如onClick，onItemClick等等。单一的个体View只是提供不同的功能，每个功能对应的业务逻辑最终还是在Activity中去实现，基本上都是通过回调方式。比如onClick，onItemClick，OnRefresh等等。所以对于ListView来说，MVC设计模式中的M其实是Activity，controler是Adapter，view是listview。
   1. MVP设计模式
2. View与Presenter之间的接口一定要在Activity中实现，因为这个接口是用来与Activity中的所有View进行交互的，只有在Activity中实现，才容易直接获取各个View的对象。
3. Presenter在拿到View提供的接口之后，实际上等于从Activity中抽离了其作为controler的功能，以为此时View的控制就由presenter完成了。View提供的接口定义的是presenter作为控制器需要完成的功能。
4. MVC模式中Activity作为控制器需要怎么控制view，是直接写代码实现的，而且是写在各个角落，没有固定位置，所以更不可能需要专门定义一个接口来实现控制功能。但是因为presenter毕竟不是activity，不可能让activity将所有view对象的引用都传给Presenter，所以view需要实现的功能，还是要在Activity中定义，然后将实现功能的方法提供给presenter，由Presenter进行调用。定义接口的好处是，当前界面下，所有需要View实现的功能均被定义到一个代码区域，清晰明了；
5. Manager
   1. devicePolicyManager
6. When a user enables the device admin application, that gives the receiver permission to perform actions in response to the broadcast of particular system events.

注册的广播接收者就是用来接收广播的，这不过这些广播是特殊的系统广播。

1. Device Adminitration
   1. 逻辑：
      1. 应用获得管理员权限之后，可以对系统设置相应的policies。
      2. 为什么要有广播接收者？

获得管理员权限之后，你就可以设置系统的policies，设置了这些policies的目的是什么？一方面是管理员主动去执行一些操作，比如锁屏、设置解锁密码、擦出数据。另一方面是去设置一些requirements。因为现在应用的角色是管理员，相对的其他没有获取管理员权限的应用就是employee app。管理员可以设置一些要求，比如密码需要时多少位的等等。在设置了相应的要求之后，如果有employee app没有遵循这些要求，系统就会发送特定的系统广播给各个管理员（系统可能有好几个管理员，拥有管理员权限即被看做是管理员）。

所以应用需要有一个广播接收者，用来处理这些特定的系统广播。权利越大，义务越多，所以管理员必须有广播接收者。

只不过如果只是想进行一些系统级操作，而并不想设置系统的policies，广播接收者的类文件可以不去重写父类DeviceAdminReceiver的各个回调方法。

* 1. Manifest中receiver的meta-data

meta-data：A name-value pair for an item of additional, arbitrary data that can be supplied to the parent component.

ComponentName：Identifier for a specific application component

DeviceAdminReceiver：Base class for implementing a device administration component.（是不是说拿到权限的其实只是这个Receiver，不是应用？）

根据上面的解释，ComponentName包含应用包名和Receiver的类名，只是标识一个specific application component，并不包含meta-data。Meta-data中的xml数据是怎么传给管理员权限激活页面的，待查？

* 1. DevicePolicyManager执行任何操作，基本都需要传入一个ComponentName，感觉receiver是个标识，告诉系统它是个device administration component。
  2. 如果有多个管理员，当DeviceAdminReceiver的回调方法执行时，哪个管理员是最后被回调的？谁有最终的决定权，去决定怎么处理当前的系统事件？
  3. ActivityManager

1. 主要用来获取动态信息。PackageManager用来获取静态信息。
2. 重要基本概念
   1. 回调函数

|  |  |
| --- | --- |
| 回调方法的定义及实现关系 | |
| A类 | B类 |
| 1.声明IF接口对象的成员变量mIFObject  public IF mIFObject;  2.定义接口:  public interface IF{  public 返回类型 if(参数);  }  3.假设A类中的invoke()方调用了IF接口的if()方法，伪代码如下：  …..  public void invoke(){  …..  mIFObject.if(参数);  …..  }  注意：  1）该方法是在A类中被调用的，A类中的回调接口，一定是在A类中被调用。为什么叫回调？A定义了一个接口和其中的方法，由别的类实现，然后A再拿回来调用，所以才叫回调。相对于A类这个if()方法才叫回调。非接口定义类去调用if()方法，都不叫回调，应该叫调用。  2）A类中通过通过mIFObject对象调用if()方法是有前提的：第一，mIFObject对象要被赋值一个已经创建的IF接口对象（A类中只是声明了这个IF接口对象，并没有创建）;第二，if()方法也要被重写，A类中只是定义了if()方法，并没有方法体。  所以，如果B类中没有实现接口并且没有返回给A中的mIFObject对象，A类中的invoke()方法是没办法执行的。 | 1.创建A类中定义的IF接口的对象ifObject，重写接口内的方法if()  2.将接口对象ifObject返回给A中的mIFObject |
| 回调方法的数据传递 | |
| A类 | B类 |
| 因为接口是在A类中定义  public interface IF{  public 返回类型 if(参数a);  }  A类可以把数据赋值给if()方法的参数a，然后通过参数a将数据给到B类。  但一般来说，这里传递给B类的参数a也仅仅是用在if方法中而已，B类并不会从if方法中拿出来使用。 | 因为接口是在B类中实现  new IF(){  public 返回类型b if(参数a){  …..  return b;  }  }  B类可以数据赋值给返回类型b，然后通过返回值b将数据传递给A类 |
| 回调方法的执行位置 | |
| A类（定义接口） | B类（创建接口对象） |
| 接口的实现代码是在B类中的，回调等于将B中的代码放到了A中执行。等于说回调这种形式，实现的代码表面上是在B类中，但是实际是在A中执行，B在帮A写代码。  实际执行的时候，如果A和B不在一个线程中，回调方法是在哪个线程中被执行？  1.让A在子线程中执行，然后 B的实现代码中弹一个吐司，看看线程之间的关系。 | |

1. Analysis
   1. 界面交互性的交替（待续）

什么操作会夺取当前Activity界面的可交互性：

* 1. Menu会抢夺Activity的交互权
  2. Toast不会
  3. Fragment不会
  4. 下拉菜单会
  5. 遮挡下层Activity的当前Activity会
  6. 混合启动服务

在调用音乐服务的时候，采用了混合启动Service的方式。Why？

假设只startService()，由于无法从系统底层拿到该音乐服务Service的对象，所以不能调用Service中的相关方法对音乐播放进行控制。但好处是即使当前Activity销毁了，Service还能够继续运行。

假设只bindSevice()，并且当前音乐服务只有绑定了这一个对象。此时可以通过binder对象调用Serivce中的相关方法，从而实现对音乐播放的控制。但是当Activity销毁时，需要解绑Service。这时候Serivice会回调onDestroy()方法，释放流对象（网络、本地）。当流对象销毁时，Service也就失去了提供音乐播放的能力。

所以说这里需要结合两种开启方式的优点。

* 1. 0xff与byte的运算分析

1. 不同格式的转换
   1. 规则1：少位数 —> 多位数：我们在进行数据类型转换时，对数据做补齐的操作，保证数据位数发生改变时其值依然不变。正数的在其左边补0，负数则是在其左边补1，而无符号的则也是在其左边补0。
   2. 规则2：多位数 —> 少位数：如果我们需要想将多位数转换为少位数，则只需对数据拦截低位数的长度即可。
2. 移位规则
   1. 当发生左移(<<)时，右边补0；
   2. 当发生有符号的右移(>>)时，左边补符号位，但是符号位不变。
   3. 当发生无符号的右移(>>>)时，无论正负，左边补0。
3. 关键问题
   1. Java中一个数值没有定义类型，默认是int。所以0xff实际是int类型，二进制表示是：00000000 00000000 00000000 11111111
   2. byte（比如-2）肯定是byte类型，根据转换规则1，负数由byte强转为int时，左边补1，所以-2强转为int后，二进制表示为：11111111 11111111 11111111 11111110
   3. 0xff & -2(byte) 计算过程为：（按位与）

00000000 00000000 00000000 11111111

& 11111111 11111111 11111111 11111110

00000000 00000000 00000000 11111110 = 254

1. 所以说&0xff后，不是说将-2转换为了+2，只不过是将负数转换为了正数而已。
2. 特别注意：
   1. 计算的时候不能光看数值表示（不要将0xff想当然的看成二进制数1111 1111），更重要的是看是什么类型，类型才决定了一个数最终的二进制表示形式。