面试宝典-Android

Activity

生命周期

生命周期方法	作用	说明
onCreate	表示 Activity 正在被创建	activity 被创建时调用，一般在这个方法中进行活动的初始化工作，如设置布局工作、加载数据、绑定控件等。
onRestart	表示 Activity 正在重新启动	这个回调代表了 Activity 由完全不可见重新变为可见的过程，当 Activity 经历了 onStop() 回调变为完全不可见后，如果用户返回原 Activity，便会触发该回调，并且紧接着会触发 onStart() 来使活动重新可见。
onStart	表示 Activity 正在被启动	经历该回调后，Activity 由不可见变为可见，但此时处于后台可见，还不能和用户进行交互。
onResume	表示 Activity 已经可见	已经可见的 Activity 从后台来到前台，可以和用户进行交互。
onPause	表示 Activity 正在停止	当用户启动了新的 Activity ，原来的 Activity 不再处于前台，也无法与用户进行交互，并且紧接着就会调用 onStop() 方法，但如果用户这时立刻按返回键回到原 Activity ，就会调用 onResume() 方法让活动重新回到前台。而且在官方文档中给出了说明，不允许在 onPause() 方法中执行耗时操作，因为这会影响到新 Activity 的启动。 一般会导致变为 onPause 状态的原因除了 onStop 中描述的四个原因外，还包括当用户按 Home 键出现最近任务列表时。
onStop	表示 Activity 即将停止	这个回调代表了 Activity 由可见变为完全不可见，在这里可以进行一些稍微重量级的操作。需要注意的是，处于 onPause() 和 onStop() 回调后的 Activity 优先级很低，当有优先级更高的应用需要内存时，该应用就会被杀死，那么当再次返回原 Activity 的时候，会重新调用 Activity 的onCreate()方法。 一般会导致变为 stop 状态的原因：1.用户按 Back 键后、用户正在运行 Activity 时，按 Home 键、程序中调用 finish() 后、用户从 A

		启动 B 后，A 就会变为 stop 状态。
onDestroy	表示 Activity 即将被销毁	来到了这个回调，说明 Activity 即将被销毁，应该将资源的回收和释放工作在该方法中执行。 当 Activity 被销毁时，销毁的情况包括：当用户按下 Back 键后、程序中调用 finish() 后。
onNewIntent	重用栈中 Activity	当在 AndroidManifest 里面声明 Activty 的时候设置了 launchMode 或者调用 startActivity 的时候设置了 Intent 的 flag ，当启动 Activity 的时候，复用了栈中已有的 Activity，则会调用 Activity 的该回调。

- 关于生命周期常见的问题：

问题	回调
由活动 A 启动活动 B时，活动 A 的 onPause() 与 活动 B 的 onResume() 哪一个先执行?	活动 A 的 onPause() 先执行，活动 B 的 onResume() 方法后执行
标准 Dialog 是否会对生命周期产生影响	没有影响
全屏 Dialog 是否会对生命周期产生影响	没有影响
主题为 Dialog 的 Activity 是否会对生命周期产生影响	有影响，与跳转 Activity 一样

- 异常状态下活动的生命周期：

在发生异常情况后，用户再次回到 Activity，原 Activity 会重新建立，原已有的数据就会丢失，比如用户操作改变了一些属性值，重建之后用户就看不到之前操作的结果，在异常的情况下如何给用户带来好的体验，有两种办法：1. 系统提供的 **onSaveInstanceState** 和 **onRestoreInstanceState** 方法，onSaveInstanceState 方法会在 Activity 异常销毁之前调用，用来保存需要保存的数据，onRestoreInstanceState 方法在 Activity 重建之后获取保存的数据。2.在默认情况下，资源配置改变会导致活动的重新创建，但是可以通过对活动的 android:configChanges 属性的设置使活动防止重新被创建。

启动模式

- standard(标准模式)：Activity 的默认启动模式，不设置启动模式时，就是标准模式。只要启动 Activity 就会创建一个新实例，并将该 Activity 添加到当前任务栈中。

  标准模式的应用场景：正常打开一个新的页面，这种启动模式使用最多，最普通。一般没有特殊需求都是使用标准模式。
- singleTop(栈顶复用)：在这种启动模式下，首先会判断要启动的活动是否已经存在于栈顶，如果是的话就不创建新实例，直接复用栈顶活动，并且调用 activity 的 onNewIntent() 方法。如果要启动的活动不位于栈顶，则会创建新实例入栈。

栈顶复用模式的应用场景：栈顶复用模式避免了同一个页面被重复打开，应用场景例如一个新闻客户端，在通知栏收到多条推送，点击一条推送就会打开新闻的详情页，如果是默认的启动模式，点击一次将会打开一个详情页，栈中就会有三个详情页，如果使用栈顶复用模式，点击第一条推送之后，接着点击其他的推送，都只会有一个详情页，可以避免重复打开页面。

- singleTask(栈内复用)：singleTask 是一种栈内单例模式，当一个 activity 启动时，如果栈中没有 activity 则会创建 activity 并让它入栈；如果栈中有 activity ，则会将位于 activity 之上的 activities 出栈，然后复用栈中的 activity ，调用 activity 的 onNewIntent() 方法。

  这种模式会保证 Activity 在栈内只有一个或者没有。

  栈内复用模式的应用场景：栈内复用模式适合作为程序的入口。最常用的就是一个 APP 的首页，一般 App 的首页长时间保留在栈内，并且是栈的第一个 activity。例如浏览器的主界面，不管从多少个应用启动浏览器，只会启动主界面一次，并清空主界面上面的其他页面，根据 onNewIntent 方法传递的数值，显示新的界面。

- singleInstance(单例模式)：这种模式是真正的单例模式，以这种模式启动的活动会单独创建一个任务栈，并且依然遵循栈内复用的特性，保证了这个栈中只能存在这一个活动。并且系统不会在这个单例模式的 Activity 的实例所在栈中中启动任何其他的 Activity 。单例模式的 Activity 的实例永远是这个栈中的唯一一个成员。

  单例模式的应用场景：单例模式使用需要与程序分离开的页面。电话拨号页面，通过自己的应用或者其他应用打开拨打电话页面，只要系统的栈中存在该实例，那么就会直接调用，还有闹铃提醒。

Intent 的 flags

- FLAG_ACTIVITY_CLEAR_TOP：设置此标志，如果 activity 已经在栈中，会将栈中 activity 之上的 activities 进行出栈关闭，如果启动模式是默认的（标准模式），设置了 FLAG_ACTIVITY_CLEAR_TOP 标志的 activity 会结束并重新创建；如果是其他模式或者 Intent 设置了 FLAG_ACTIVITY_SINGLE_TOP，则 activity 会将新的 intent 传递给栈中的 activity 的 onNewIntent() 方法。
- FLAG_ACTIVITY_NO_HISTORY：如果这只此 flag，则启动的 activity 将不会保留在历史栈中，一旦用户离开它，activity 将结束。
- FLAG_ACTIVITY_NO_ANIMATION：设置此标签，则跳转启动的 activity 动画不会显示。
- FLAG_ACTIVITY_NEW_TASK：设置 FLAG_ACTIVITY_NEW_TASK 标签后，首先会查找是否存在和被启动的 activity 具有相同亲和性的任务栈，如果没有，则新建一个栈让 activity 入栈；如果有，则保持栈中 activity 的顺序不变，如果栈中没有 activity，将 activity 入栈，如果栈中有 activity，则将整个栈移动到前台。
- FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_CLEAR_TASK：FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_CLEAR_TASK 联合使用时，首先会查找是否存在和被启动的 activity 具有相同亲和性的任务栈，如果有则先将栈清空，将被启动的 activity 会入栈，并将栈整体移动到前台；如果没有，则新建栈来存放被启动的 activity。
- FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_CLEAR_TOP：FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_CLEAR_TOP 联合使用时，首先会查找是否存在和被启动的 activity 具有相同亲和性的任务栈，如果有，栈中如果包含 activity ，则将栈中 activity 之上包括栈中的 activity 移除，将被启动的 activity 入栈，并将栈整体移动到前台，如果栈中没有要启动 activity，则直接将 activity 入栈；如果没有，则新建栈来存放被启动的 activity。

- FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS：如果设置 FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS，则新的 activity 将不会被保留在最近启动 activities 的列表中。

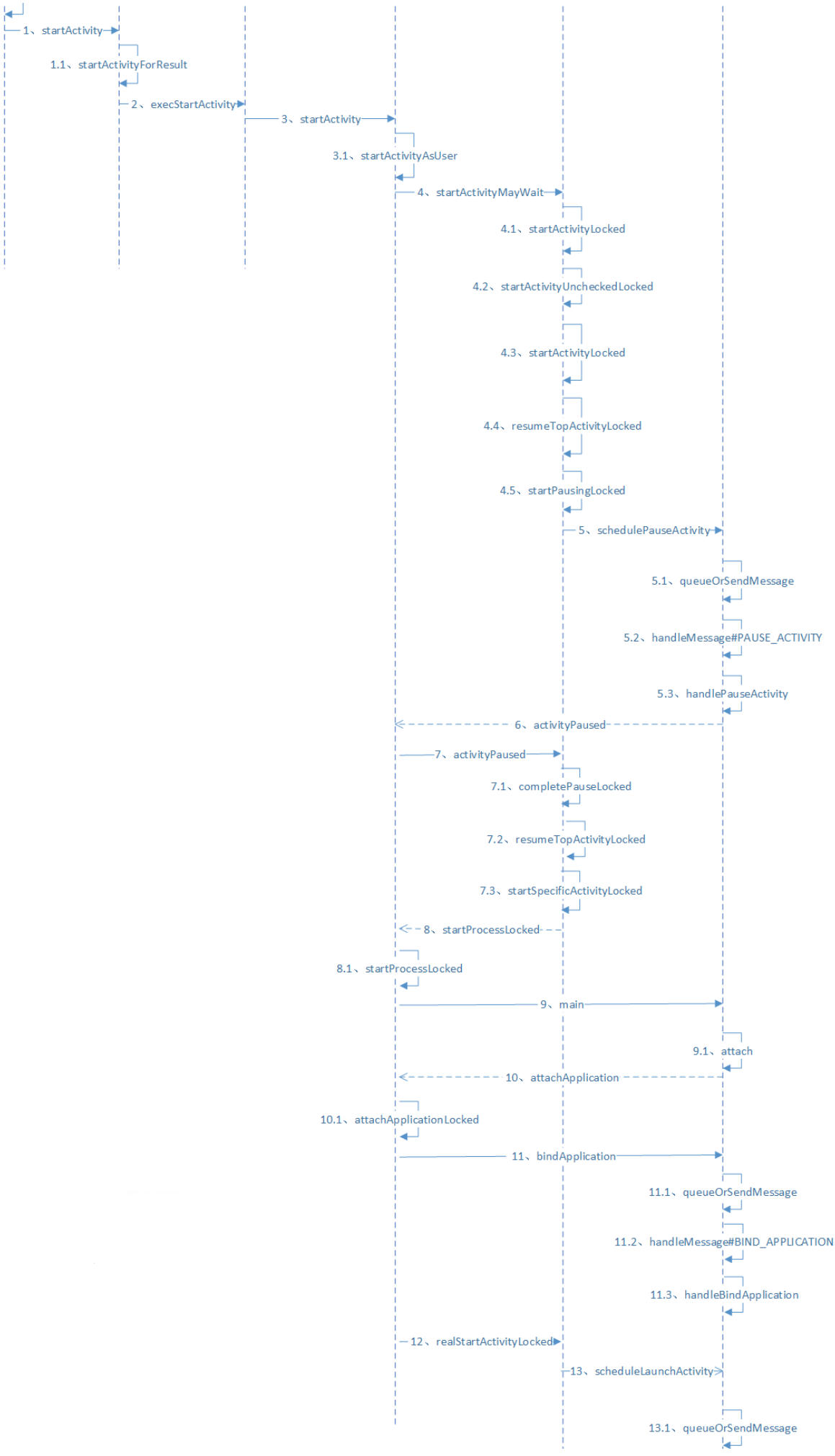
  FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS 与使用 FLAG_ACTIVITY_NO_HISTORY 标志不同，使用 FLAG_ACTIVITY_NO_HISTORY 标志时，在经过 A -> B -> C 的界面跳转后，在 C 界面点击 back 键就会回到 A 界面，而 FLAG_ACTIVITY_NEW_TASK 和 FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS 一起使用时，在经过 A -> B -> C 的界面跳转后 ，在 C 点击 back 返回，还是会回到 B 界面的。
- FLAG_ACTIVITY_REORDER_TO_FRONT：设置此标志，如果 activity 已经在栈中运行，将会把 activity 带到栈的顶部。
- FLAG_ACTIVITY_FORWARD_RESULT：如果设置这个标志并用于启动一个新的 activity，则回复对象从本 activity 移动到新的 activity 上。
- FLAG_ACTIVITY_NEW_DOCUMENT：被用于基于 Intent 的 activity 活动开一个新的任务。同一个 activity 的不同实例将会在最近的任务列表中显示不同的记录。
- FLAG_ACTIVITY_NEW_DOCUMENT 与 FLAG_ACTIVITY_MULITIPLE_TASK：单独使用 FLAG_ACTIVITY_NEW_DOCUMENT 时，会先从存在的任务栈中搜索匹配 Intent 的栈，如果没有任务栈被发现则创建新的任务栈，当与 FLAG_ACTIVITY_MULTIPLE_TASK 配合使用时，会跳过搜索匹配任务栈而是直接开启一个新的任务栈。
- FLAG_ACTIVITY_NEW_TASK 与 FLAG_ACTIVITY_MULITIPLE_TASK：单独使用 FLAG_ACTIVITY_NEW_TASK 时，会先从存在的任务栈中搜索匹配 Intent 的栈，如果没有任务栈被发现则创建新的任务栈，当与 FLAG_ACTIVITY_MULTIPLE_TASK 配合使用时，会跳过搜索匹配任务栈而是直接开启一个新的任务栈。
- FLAG_ACTIVITY_RETAIN_IN_RECENTS：默认情况下，进入最近任务栈的记录由 FLAG_ACTIVITY_NEW_DOCUMENT 创建，当用户关闭 activity 时任务栈就会被移除，如果想要允许任务栈保留方便它能被重新启动，可以使用此标志。
- FLAG_ACTIVITY_NO_USER_ACTION：如果设置此标志，在 activity 被前台的新启动的 activity 造成 paused 之前，将会阻止当前最顶部的 activity 的 onUserLeaveHint 回调。通常，当 activity 在用户的操作下被移除栈顶则会调用 onUserLeaveHint 回调，这个回调标志着 activity 生命周期的一个点，以便隐藏任何 “ 直到用户看到它们 ” 的通知，比如闪烁的 LED 灯。

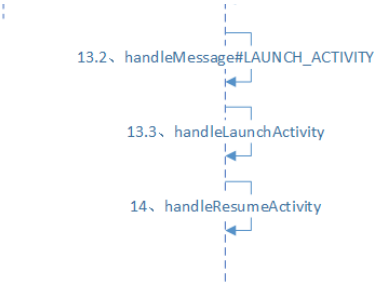
onNewIntent 的回调时机

onNewIntent() 方法会在 activity 复用的时候调用，也就是说调用 activity ，并不会创建 activity 的新实例，而是复用栈中的 activity ，复用时就会调用 onNewIntent() 方法，将新的 Intent 传递给 oNewIntent() 方法。

Activity 的启动流程（重新梳理）

冷启动


1、startActivity
1.1、startActivityForResult
2、execStartActivity
3、startActivity
3.1、startActivityAsUser
4、startActivityMayWait
4.1、startActivityLocked
4.2、startActivityUncheckedLocked
4.3、startActivityLocked
4.4、resumeTopActivityLocked
4.5、startPausingLocked
5、schedulePauseActivity
5.1、queueOrSendMessage
5.2、handleMessage#PAUSE_ACTIVITY
5.3、handlePauseActivity
6、activityPaused
7、activityPaused
7.1、completePauseLocked
7.2、resumeTopActivityLocked
7.3、startSpecificActivityLocked
8、startProcessLocked
8.1、startProcessLocked
9、main
9.1、attach
10、attachApplication
10.1、attachApplicationLocked
11、bindApplication
11.1、queueOrSendMessage
11.2、handleMessage#BIND_APPLICATION
11.3、handleBindApplication
12、realStartActivityLocked
13、scheduleLaunchActivity
13.1、queueOrSendMessage



图中涉及的几个类：

（1）Launcher：Launcher 本质上也是一个应用程序，和一个简单的 App 一样，也继承自 Activity，实现了点击、长按等回调接口，来接收用户的输入。

（2）ActivityManagerServices：简称 AMS，服务端对象，负责系统中所有 Activity 的生命周期。

（3）ActivityThread：App 的真正入口。当开启 App 之后，会调用 main() 开始运行，开启消息循环队列，这就是 UI 线程（主线程）。与 ActivityManagerService 配合，一起完成 Activity 的管理工作。

（4）ApplicationThread：用来实现 ActivityManangerService 与 ActivityThread 之间的交互。在 ActivityManangerService 需要管理相关 Application 中的 Activity 的生命周期时，通过 ApplicationThread 的代理对象与 ActivityThread 通讯。

（5）ApplicationThreadProxy：是 ApplicationThread 在服务器端的代理，负责和客户端的 ApplicationThread 通讯。AMS 就是通过该代理与 ActivityThread 进行通信的。

（6）Instrumentation：每一个应用程序只有一个 Instrumentation 对象，每个 Activity 内都有一个对该对象的引用。Instrumentation 可以理解为应用进程的管家，ActivityThread 要创建或暂停某个 Activity 时，都需要通过 Instrumentation 来进行具体的操作。

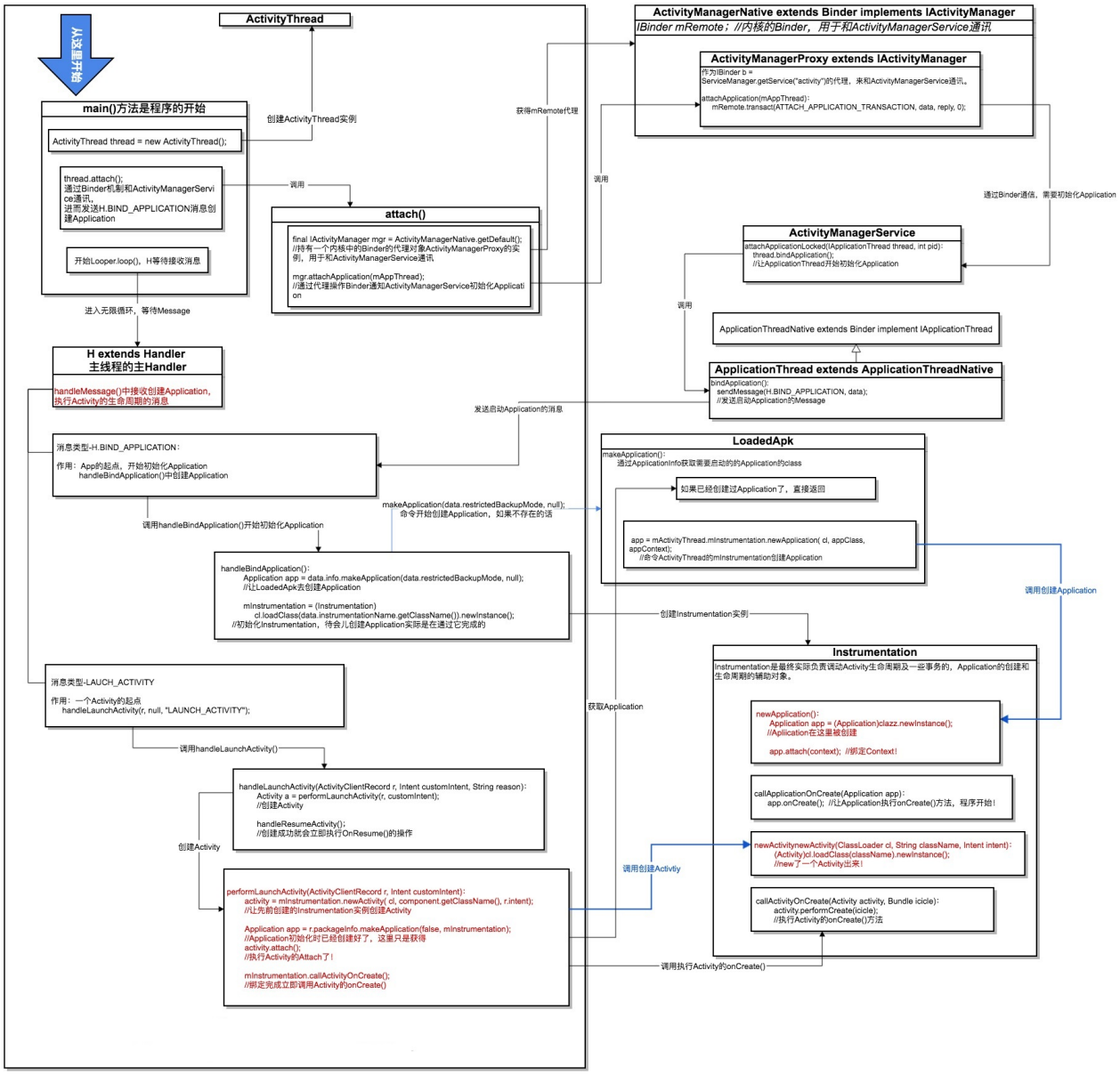
（7）ActivityStack：Activity 在 AMS 的栈管理，用来记录已经启动的 Actiivty 的先后关系，状态信息等。通过 ActivityStack 决定是否需要启动新的进程。

（8）ActivityRecord：ActivityStack 的管理对象，每个 Activity 在 AMS 对应一个 ActivityRecord，来记录 Activity 的状态以及其他的管理信息。其实就是服务端的 Activity 对象的映像。

（9）TaskRecord：AMS 抽象出来的一个“任务”的概念，是记录 ActivityReacord 的栈，一个“Task”包含若干个 ActivityRecord 。AMS 用 TaskRecord 确保 Activity 启动和退出的顺序。

Activity 冷启动过程（app 进程不存在）：在 launch 点击触发了打开应用后，会先通过调用 AMS 的 startActivity() 方法，而 AMS 会调用 ActivityStack 的 resumeTopActivityInnerLocked 方法，在该方法中会先 pause 当前显示的 activity，在处理完 pause activity 之后，会判断启动 app 的进程是否存在，判断如果 Activity 所在进程存在且 Activity 之前启动过，则直接发送 ResumeActivityItem 请求通知 APP 进程进行 resume，否则调用 **ActivityStackSupervisor** 的 **startSpecificActivityLocked方法** 继续执行去启动目标进程（通过 **Process.start** 请求 ZYGOTE 创建子 APP 进程）。

热启动



Activity 热启动过程：ActivityThread 的 main() 方法作为程序的入口，在 main() 方法中，初始化了主线程的 Looper，主 Handler，并使主线程进入等待接收 Message 消息的无限循环状态，调用 attach() 方法，而 attach() 方法通过调用 ActivityManager 的 attachApplication() 方法，最后调用到 ApplicationThread 的 bindApplication() 方法，在 bindApplication() 方法中发送出 BIND_APPLICATION 的消息，ActivityThread 类处理 BIND_APPLICATION 消息，接收到 BIND_APPLICATION 消息之后，创建一个 Application 实例，初始化一个 Instrumentation 对象，通过 Instrumentation 的 callApplicationOnCreate() 方法去调用 Application 的 onCreate() 方法。ApplicationThread 发出 LAUNCH_ACTIVITY 消息来启动 activity，通过反射机制创建 activity 实例，创建完成之后就会调用 onCreate() 方法。

关于 IActivityManager、ActivityManagerNative、ActivityManagerProxy、ActivityManagerService

IActivityManager 是一个接口，用于与活动管理服务通讯。ActivityManagerProxy 实现了 IActivityManager 接口，ActivityManagerProxy 主要代理了内核中与 ActivityManager 通讯的 Binder 实例。ActivityManagerProxy 持有一个 ActivityManagerNative 的对象实例，当调用 IActivityManager 的方法时，调用 ActivityManagerNative 的实例来完成。ActivityManagerNative 是一个抽象类，实现 IActivityManager 接口，并且继承 Binder 类，提供 ActivityManagerProxy 实例供外部使用。

ActivityManagerService 类继承 ActivityManagerNative 类，真正实现 IActivityManager 接口的方法。

很明显 ActivityManager 使用的是代理模式，ActivityManagerProxy 代理了与活动管理服务通讯。

关于 ApplicationThread

ApplicationThread 作为 IApplicationThread 的一个实例，承担了发送 Activity 生命周期以及它一些消息的任务，也就是说发送消息。

至于为什么在 ActivityThread 中已经创建出了 ApplicationThread 了还要绕弯路发消息，是为了让系统根据情况来控制这个过程。

关于 Instrumentation

Instrumentation 会在应用程序的任何代码运行之前被实例化，它能够允许你监视应用程序和系统的所有交互。

在 Application 的创建、Activity 的创建和生命周期过程中都会调用 Instrumentation 的方法，Application 的创建是调用 ActivityManagerService 的方法来实现，而 Activity 的创建是反射实现，Activity 的生命周期调用了 activity 的相关方法。

Instrumentation 是如何实现监视应用程序和系统交互的？Instrumentation 类将 Application 的创建、Activity 的创建以及生命周期这些操作包装起来，通过操作 Instrumentation 进而实现上述操作。

Instrumentation 封装有什么好处？Instrumentation 作为抽象，在约定好需要实现的功能之后，只需要给 Instrumentation 添加这些抽象功能，然后调用就好了。关于怎么实现这些功能，都会交给 Instrumentation 的实现对象就好了。这就是多态的运用，依赖抽象，不依赖具体的实践。就是上层提出需求，底层定义接口，即依赖倒置原则的践行。

1. 先讲下普通的activity的启动流程吧，涉及到应用的本地进程和系统进程（AMS）。

   首先调用startActivity启动新的activity，这个方法其实调用了startActivityForResult方法。里面向instrumentation请求创建，调用execActivity方法，通过AMS在本地进程的IBinder接口（IActivityManager）（AIDL通信，用该AIDL的代理类）向AMS发起远程请求创建Activity，

   AMS首先调用startActivity方法，里面校验了要启动的activity是否注册、确定启动模式等信息，如果启动的activity合法，然后会调用ActivityTaskSuperVisitor，找出对应的activityTask。如果activityTask栈顶上存在处于resume状态的activity，则让该activity调用其onPause方法；接下来判断应用进程是否已启动，如果没有就启动应用的进程，创建ActivityThread对象，并调用其main方法；

   在 ActivityThread 的 main() 方法中会初始化主线程的 Looper，并且发出创建 Application 的消息，最后开启 Looper，等待接收消息。创建 application 是利用本地进程在AMS的IBinder接口（IApplicationThread）直接调用本地的ActivityThread的内部类ApplicationThread对象的ScheduleActivity方法，通过叫H的handler对象，切换到主线程调用handleLaunchActivity方法，最后把逻辑处理交给performLaunchActivity。这个方法主要干了几个事情：（1）获取待启动activity的信息；（2）通过instrumentation利用类加载器创建待启动activity对象；（3）判断是否已经有创建Application对象，没有则创建并调用其onCreate方法。（3）调用Activity对象的attach方法来初始化一些重要数据（创建PhoneWindow和WindowManager、主线程的赋值等）；（4）然后调用activity的onCreate方法。接着跳出performLaunchActivity调用handleStartActivity方法（这里应该调用了activity的onStart方法），然后调用handleResumeActivity方法，通过activity的

performResume方法调用了activity的onResume方法；接着通过WindowManagerImpl里面的WindowGlobal创建要添加的view对应的viewRootImp对象，然后WindowSession将phoneWindow添加到WMS中。WMS把相应的DecorView添加到该window，并用对应的viewRootImp对象完成对view的绘制，然后设置view可见了。最后通知AMS执行前一个activity（如果有的话）的stop方法。

2. 根activity的启动：
   （1）Launcher进程请求AMS创建activity
   （2）AMS请求Zygote创建进程。
   （3）Zygote通过fork自己来创建进程。并通知AMS创建完成。
   （4）AMS通知应用进程创建根Activity。
   （5）创建过程和上面说过的普通activity的流程是一样的

Service

Service 是可以在后台执行长时间运行操作并且不需要和用户交互的应用组件。服务是由其他应用组件启动，依赖于启动服务所在的应用程序进程，服务一旦被启动将在后台一直运行，即使启动服务的组件已销毁也不受影响。此外，服务也可以绑定到组件上，以与之进行交互。

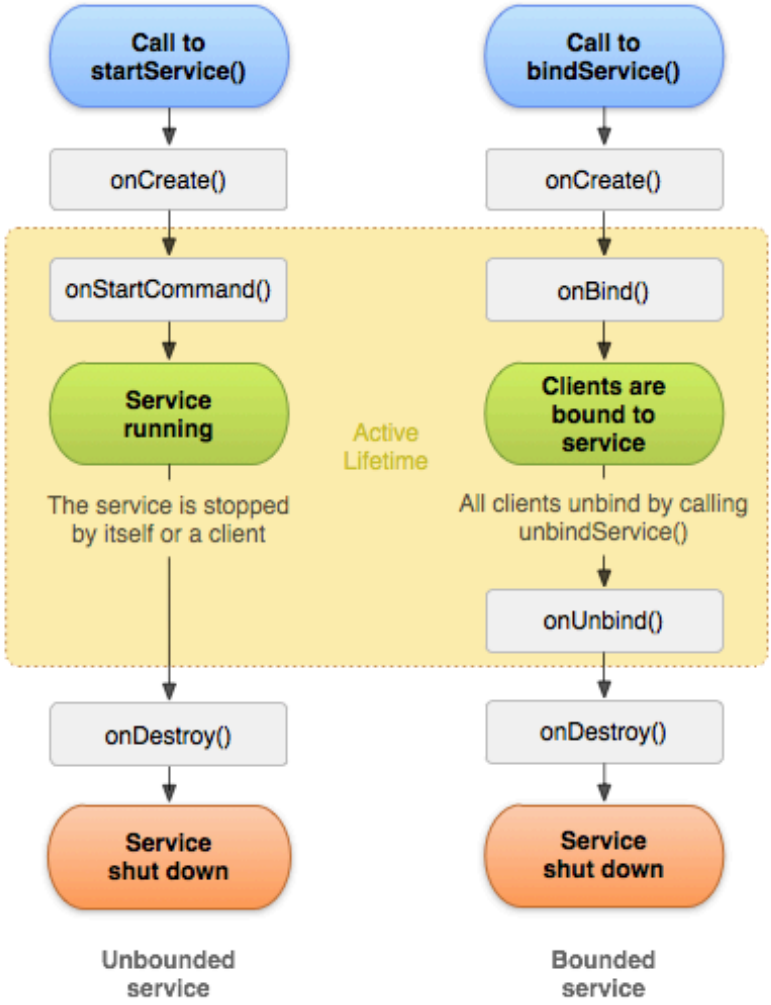
服务可以在很多场合使用，比如播放多媒体的时候用户启动了其他 activity，此时要在后台继续播放；比如检测 sd 卡上文件的变化；比如在后台记录你的地理位置的改变；也可以执行进程间通信（IPC）等等。

两种启动方式

服务有两种启动方式，一种是启动服务，一种就是绑定服务。

1. 启动服务：当应用组件（如 Activity）通过调用 `startService()` 启动服务时，服务即处于 “ 启动 ” 状态。一旦启动，服务即可在后台无限期运行，即使启动服务的组件已被销毁也不受影响，除非手动调用才能停止服务，已启动的服务通常是执行单一操作，而且不会将结果返回给调用方。
2. 绑定服务：当应用组件通过调用 `bindService()` 绑定到服务时，服务即处于“绑定”状态。绑定服务提供了一个客户端 - 服务端接口，允许组件与服务进行交互、发送请求、获取结果，甚至是利用进程间通信（IPC）执行这些操作。仅当与另一个应用组件绑定时，绑定服务才会运行。多个组件可以同时绑定到该服务，但全部取消绑定后，该服务即会被销毁。

生命周期



Service生命周期图

Call to startService()：onCreate() -> onStartCommand()->onDestory()

Call to bindService()：onCreate() -> onBind() -> onUnbind() -> onDestory()

onCreate()：首次创建服务时，系统将调用此方法来执行一次性设置程序（在调用 `onStartCommand()` 或 `onBind()` 之前），如果服务已在运行，则不会调用此方法，该方法只调用一次。

onBind()：当另一个组件想通过调用 `bindService()` 与服务绑定（例如执行 RPC）时，系统将调用此方法。在此方法实现中，必须返回一个 IBinder 接口的实现类，供客户端用来与服务进行通信。无论是启动状态还是绑定状态，此方法必须重写，但在启动状态就会直接返回 null 。

onStartCommand()：当另一个组件（Activity）通过调用 `startService()` 请求启动服务时系统将调用此方法，一旦执行此方法，服务即会启动并可在后台无限期运行。如果自己实现此方法，则需要在服务工作完成后，通过调用 `stopSelf()` 或 `stopService()` 来停止服务（在绑定状态无需实现此方法）。

onUnbind()：当另一个组件通过调用 `unbindServicer()` 与服务解绑时，系统将调用此方法。

onDestroy()：当服务不再使用且被销毁时，系统将调用此方法，服务应该实现此方法来清理所有的资源，如线程、注册的监听器、接收器等，这是服务接收的最后一个调用。

启动服务与绑定服务

启动服务和绑定服务的生命周期

- 启动服务生命周期：第一次调用 `startService()` 启动服务，会调用 `onCreate()` 和 `onStartCommand()` 方法，之后再次调用 `startService()` 启动服务，只会调用 `onStartCommand()` 方法。调用 `stopService()` 方法停止服务，会调用 `onDestory()` 方法。停止服务之后再次 `startService()` 启动服务，会再次调用 `onCreate()` 和 `onStartCommand()` 方法。
- 绑定服务生命周期：第一次调用 `bindService()` 启动服务，调用 `onCreate()` 和 `onBind()` 方法，之后调用 `bindService()` 没有任何方法调用，调用 `unbindService()` 方法解绑服务，会调用 `onUnbind()` 和 `onDestory()` 方法。在 Activity 退出的时候不调用 `unbindService()` 解绑的话会报错。
- 启动并绑定服务生命周期
    - 先绑定服务后启动服务：先调用 `bindService()` 方法，调用 `onCreate()` 和 `onBind()` 方法，再调用 `startService()` 方法，调用 `onStartCommand()` 方法，调用 `unbindService()` 方法解绑，调用 `onUnbind()` 方法，再调用 `stopService()` 方法，调用 `onDestory()` 方法，如果是先调用 `stopService()` 没有方法回调，再调用 `unbindService()` 方法解绑会调用 `onUnbind()` 和 `onDestory()` 方法。
    - 先启动服务后绑定服务：先调用 `startService()` 方法，调用 `onCreate()` 和 `onStartCommand()` 方法，（之后再调用 `startService()` 方法，只会回调 `onStartCommand()` 方法）再调用 `bindService()` 方法，调用 `onBind()` 方法，调用 `unbindService()` 方法解绑，调用 `onUnbind()` 方法，再调用 `stopService()` 方法，调用 `onDestory()` 方法，如果是先调用 `stopService()` 没有方法回调，再调用 `unbindService()` 方法解绑会调用 `onUnbind()` 和 `onDestory()` 方法。

启动服务与绑定服务的区别

区别一：生命周期

通过 start 方式的服务会一直运行在后台，需要由组件本身或外部组件来停止服务才会结束。

bind 方式的服务，生命周期就会依赖绑定的组件。

区别二：参数传递

start 服务可以给启动的服务对象传递参数，但无法获取服务中方法的返回值。

bind 服务可以给启动的服务对象传递参数，也可以用过绑定的业务对象获取返回结果。

IntentService

服务不会自动开启线程，服务中的代码默认是运行在主线程中，如果直接在服务里执行一些耗时操作，容易造成 ANR(Application Not Responding)异常，为了可以简单的创建一个异步的、会自动停止的服务，Android 专门提供了一个 **IntentService** 类。可以启动 IntentService 多次，而每一个耗时操作会以工作队列的方式在 IntentService 的 onHandleIntent() 回调方法中执行，并且每次只会执行一个工作线程，执行完第一个，再执行第二个，以此类推。

前台服务

前台服务被认为是用户主动意识到的一种服务，因此在内存不足时，系统也不会考虑将其终止。前台服务必须为状态栏提供通知，状态栏位于“ 正在进行 ”标题下方，这意味着除非服务停止或从前台删除，否则不能清除通知。例如将从服务播放音乐的音乐播放器设置在前台运行，这是因为用户明确意识到其操作。状态栏中的通知可能表示正在播放的歌曲，并允许用户启动 Activity 来与音乐播放器进行交互。

`startForeground()` 和 `stopForeground()` 方法分别将服务设置为前台服务和从前台删除服务。`startForeground(int id, Notification notification)` 的作用是把当前服务设置为前台服务，其中 `id` 参数代表唯一标识通知的整型数，需要注意的是提供给 `startForeground()` 的整型 ID 不得为 0 ，而 notification 是一个状态栏的通知。`stopForeground(boolean removeNotification)` 用来从前台删除服务，此方法传入一个布尔值，指示是否也删除状态栏通知，true 为删除。注意该方法并不会停止服务，但是，如果在服务正在前台运行时将其停止，则通知也会被删除。

BroadcastReceiver

BroadcastReceiver ，广播接收者，用来接收来自系统和应用中的广播，是 Android 四大组件之一。

BroadcastReceiver 的两种常用类型

- Normalbroadcasts（默认广播）:发送一个默认广播使用 `Context.sendBroadcast()` 方法，普通广播对于多个接受者来说是完全异步的，通常每个接受者都无需等待即可接收到广播，接受者相互之间不会有影响。对于这种广播，接收者无法终止广播，即无法阻止其他接收者的接收动作。
- Orderedbroadcasts（有序广播）： 发送一个有序广播使用 `Context.sendorderedBroadcast()` 方法，有序广播比较特殊，它每次只发送到优先级较高的接受者那里，然后由优先级高的接受者再传播到优先级低的接受者那里，优先级高的接受者有能力终止这个广播。

静态和动态注册方式

构建 Intent ，使用 sendBroadcast 方法发出广播定义一个广播接收器，该广播接收器继承 BroadcastReceiver，并且覆盖 onReceive() 方法来接收事件。注册该广播接收器，可以在代码中注册（动态注册），也可以在 AndroidManifest.xml 配置文件中注册（静态注册）。

两种注册方式区别

广播接收器注册一种有两种形式：静态注册和动态注册。

两者及其接收广播的区别：

（1）动态注册广播不是常驻型广播，也就是说广播跟随 Activity 的生命周期。注意在 Activity 结束前，移除广播接收器。静态注册是常驻型，也就是说当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

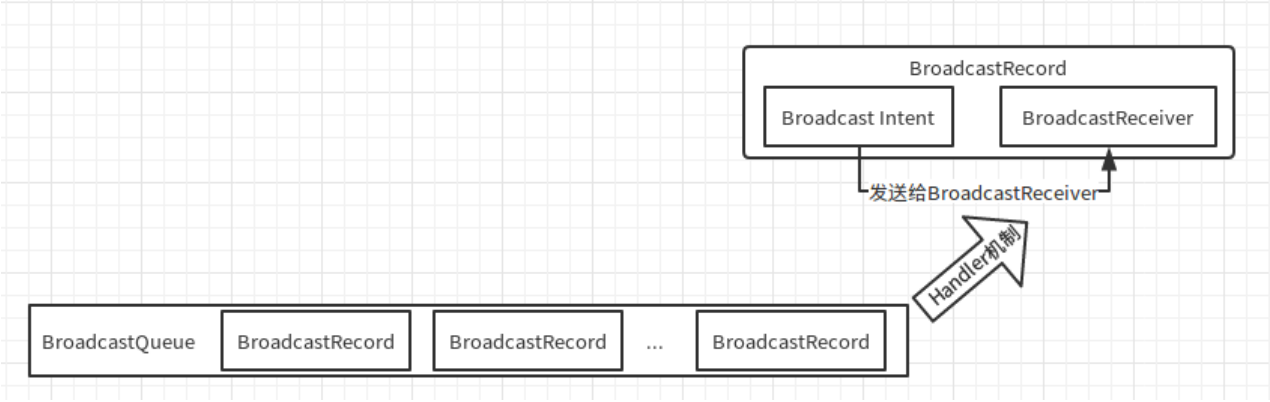
（2）当广播为有序广播时：优先级高的先接收（不分静态和动态）。同优先级的广播接收器，动态优先于静态。当广播为默认广播时：无视优先级，动态广播接收器优先于静态广播接收器。

（3）同优先级的同类广播接收器，静态：先扫描的优先于后扫描的。动态：先注册优先于后注册的。

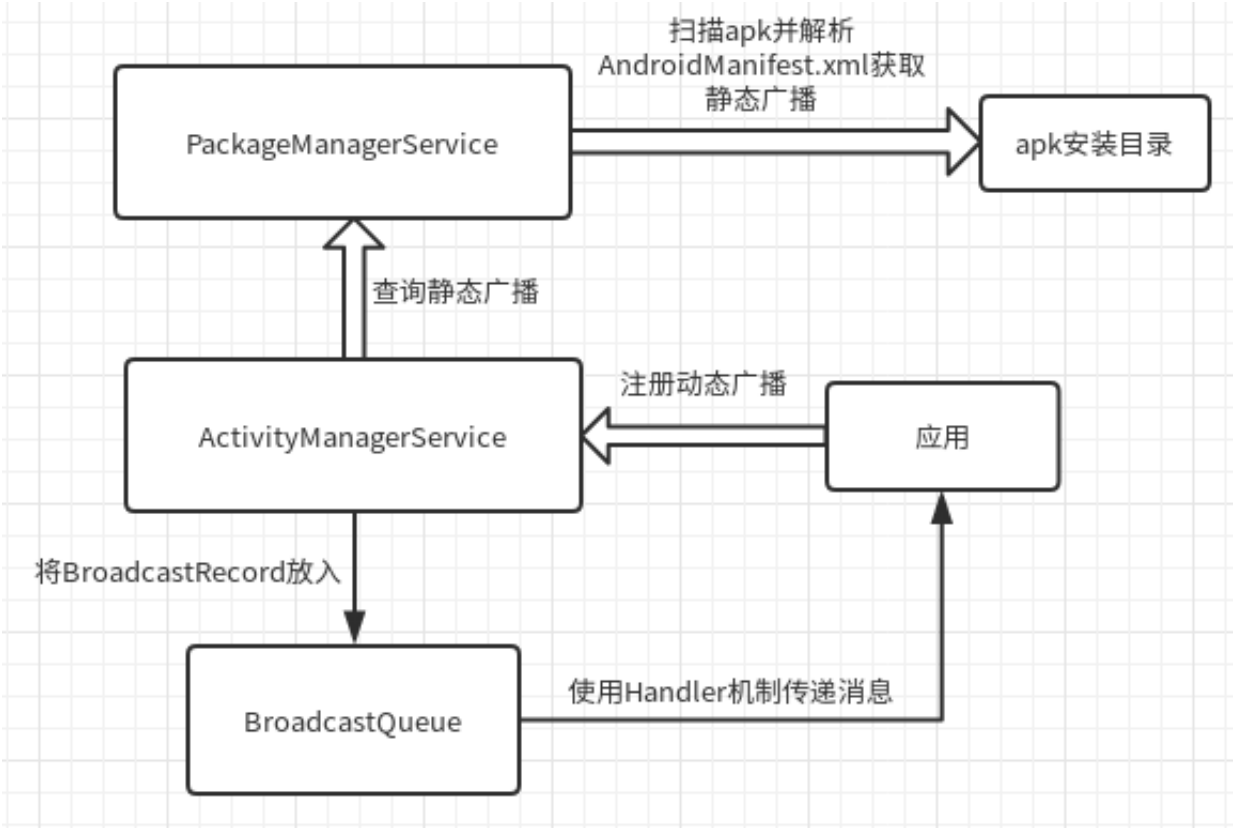
（4）静态注册是在 AndroidManifesy.xml 里通过< receive > 标签声明的。不受任何组件的生命周期影响，缺点是耗电和占内存，适合在需要时刻监听使用。动态注册在代码中调用 `Context.registerReceiver()` 方法注册，比较灵活，适合在需要特定时刻监听使用。

BroadcastReceiver 的实现原理

广播队列传送广播给 Receiver 的原理其实就是将 BroadcastReceiver 和消息都放到 BroadcastRecord 里面，然后通过 Handler 机制遍历 BroadcastQueue 里面的 BroadcastRecord ，将消息发送给 BroadcastReceiver：



整个广播的机制总结成下图：



ContentProvider

ContentProvider 可以实现在应用程序之间共享数据。

Android 为常见的一些数据提供了默认的 ContentProvider（包括音频、视频、图片和通讯录等）。所以可以在其他应用中通过那些 ContentProvider 获取这些数据。

Android 所提供的 ContentProvider 都存放在 android.provider 包中。

为什么要选择 ContentProvider

虽然也可以通过文件等其他方式来达到在不同程序之间共享数据，但是会很复杂，而 ContentProvider 也是可以实现应用程序之间共享数据的，除了可以在不同程序之间共享数据之外，还有其他优点。

ContentProvider 的 特点

1. ContentProvider 为存储和获取数据提供了统一的接口。ContentProvider 对数据进行了封装，不用关心数据存储的细节。统一了数据的访问方式。
2. 使用 ContentProvider 可以在不同的应用程序之间共享数据。
3. Android 为常见的一些数据提供了默认的 ContentProvider（包括音频、视频、图片和通讯录等）。
4. 不同于文件存储和 SharedPreferences 存储中的两种全局可读写操作模式，ContentProvider 可以选择只对哪一部分进行共享，从而保证程序中的隐私数据不会有泄漏的风险。

对 ContentProvider 封装的理解

继承 ContentProvider 的类在 onCreate()、insert()、delete()、update()、query()、getType() 方法中实现对数据增删改查的操作，而数据的存储可以使用文件、数据库、网络等各种方式去实现。而对数据的操作使用的是 ContentResolver 类，不管 ContentProvider 如何对数据进行实质操作，ContentReselver 的使用都是一样的。将实现与使用进行了分割，完成了对数据的封装，也统一了对数据的使用方式。

* ContentProvier 运行过程源码分析

* ContentProvider 的共享数据更新通知机制

IntentFilter 的匹配规则

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，如果匹配不成功就不能启动目标 Actiivty。

IntentFilter 中过滤的信息包括：action、category、data。

关于 IntentFilter 的一些描述：

- 匹配过滤列表时需要同时匹配过滤列表中的 action、category、data。
- 一个 Activity 中可以有多组 intent-filter。
- 一个 intent-filter 可以有多个 action、category、data，并各自构成不同类别，一个 Intent 必须同时匹配 action 类别、category 类别和 data 类别才算完全匹配。
- 一个 Intent 只要能匹配任何一组 intent-filter 就算匹配成功。

action 的匹配规则

1. Intent 中必须存在 action，这一点和 category 不同。
2. action 的字符串严格区分大小写，intent 中的 action 必须和过滤规则中的 action 完全一致才能匹配成功。
3. 匹配规则中可以同时有多个 action，但是 Intent 中的 action 只需与其中一只相同即可匹配成功。

category 匹配规则

1. 匹配规则中必须添加 “action.intent.category.DEFAULT” 这个过滤条件。
2. Intent 中可以不设置 category，系统会自动添加 “action.intent.category.DEFAULT” 这个默认的 category。
3. Intent 中可以同时设置多个 category，一旦设置多个 category，那么每个 category 都必须能够和过滤条件中的某个 category 匹配成功。

category 的第 3 个规则和 action 的匹配规则有所不同，action 有多个的时候，主要其中之一能够匹配成功即可，但是 category 必须是每一个都需要匹配成功。

data 的匹配规则

1. Intent 中必须有 data 数据。
2. Intent 中的 data 必须和过滤规则中的某一个 data 完全匹配。
3. 过滤规则中可以有多个 data 存在，但是 Intent 中的 data 只需匹配其中的任意一个 data 即可。
4. 过滤规则中可以没有指定 URI，但是系统会赋予其默认值：content 和 file，这一点在 Intent 中需要注意。
5. 为 Intent 设定 data 和 type 的时候必须要调用 setDataAndType() 方法，而不能先 setData 再 setType，因为这两个方法是互斥的，都会清除对方的值。
6. 在匹配规则中，data 的 scheme、host、post、path 等属性可以写在同一个 < / > 中，也可以分来单独写，其功效是一样的。

* ActivityManagerService

* WindowManagerService

* PackageManagerService 之启动解析

守护进程

守护进程的实现思想分为两方面：

1. 保活。通过提高进程优先级，降低进程被杀死的概率。当前业界的 Andriod 进程保活手段主要分为黑、白、灰三种。

2. 拉起。进程被杀死后，进行拉起。

守护进程的实现方法如下：

- 黑色保活：不同的 app 进程，用广播相互唤醒（包括利用系统提供的广播进行唤醒）。
- 白色保活：通过启动前台 Service 使得进程优先级提高到前台进程。
- 灰色保活：利用系统的漏洞启动前台 Service。
- 双进程守护：两个进程互相拉起。
- JobService 轮询：关闭后自动拉起。

1. 黑色保活

所谓黑色保活，就是利用不同的 app 进程使用广播来进行相互唤醒。

适用对象：腾讯系全家桶、阿里系全家桶、应用之间互相拉起。

举 3 个比较常见的场景：

1. 开机、网络切换、拍照、拍视频的时候，利用系统产生的广播唤醒 app。

2. 接入第三方 SDK 也会唤醒相应的 app 进程，如微信 sdk 会唤醒微信，支付宝 sdk 会唤醒支付宝。

3. 假如手机里装了支付宝、淘宝、天猫、UC 等阿里系的 app，那么打开任意一个阿里系的 app 后，有可能就顺便把其他阿里系的 app 给唤醒了。

对于场景 1，在最新的 Android N 取消了 ACTION_NEW_PICTURE（拍照）、ACTION_NEW_VIDEO（拍视频）、CONNECTIVITY_ACTION（网络切换）等三种广播。而开机广播，有一些定制 ROM 的厂商会将其去掉。

2. 白色保活

白色保活手段非常简单，就是调用系统 api 启动一个前台的 Service 进程，这样会在系统的通知栏生成一个 Notification，用来让用户知道有这样一个 app 在运行着，哪怕当前的 app 推到了后台。

比如 LBE 和 QQ 音乐就是这样。

优点：写法简单、处理方便。

缺点：前台服务和通知绑定在一起，意味着开启服务要伴随一条通知在通知栏，用户有感知。

3. 灰色保活

灰色保活，这种保活手段是应用范围最广泛。

它是利用系统的漏洞来启动一个前台的 Service 进程，与普通的启动方式区别在于，它不会在系统通知栏处出现一个 Notification，看起来就如同运行着一个后台 Service 进程一样。这样做带来的好处就是，用户无法察觉到应用运行着一个前台进程（因为看不到 Notification），但是应用的进程优先级又是高于普通后台进程的。

大致的实现思路如下：

1. 思路一： API level < 18，启动前台 Service 时直接传入 new Notification。

2. 思路二：API level >= 18，同时启动两个 id 相同的前台 Service，然后再将后启动的 Service 做 stop 处理。。

使用灰色保活并不代表 Service 就永久不死了，只能说是提高了进程的优先级。如果 app 进程占用了大量的内存，按照回收进程的策略，同样会干掉 app。

优点：开启前台服务的情况下，可以去掉通知，使得用户无感知。

缺点：target26 8.0 以上的系统该漏洞已修复，因此不适用。

4. 双进程守护

所谓双进程守护，就是指两个进程互相监视，一旦有一个进程死了，另一个进程监听到就拉起。

依托这个原理，衍生出的双进程守护的方案有很多，比如利用监听 socket 连接中断实现，利用文件锁实现，利用 android 的绑定服务实现。

以服务绑定为例：

```
context.bindService(intent, serviceConnection, flag);
```

这里的 serviceconnection 就是监听回调，回调中有 onServiceConnected 方法和 onServiceDisconnected 这两个方法，通过 onServiceDisconnected 可以监听到另一个服务是否还存活。把两个服务放在两个进程就能够做到监听并拉起进程。

5. JobService

通过定时触发任务，判定进程是否存活，如果不存活了，则拉起。

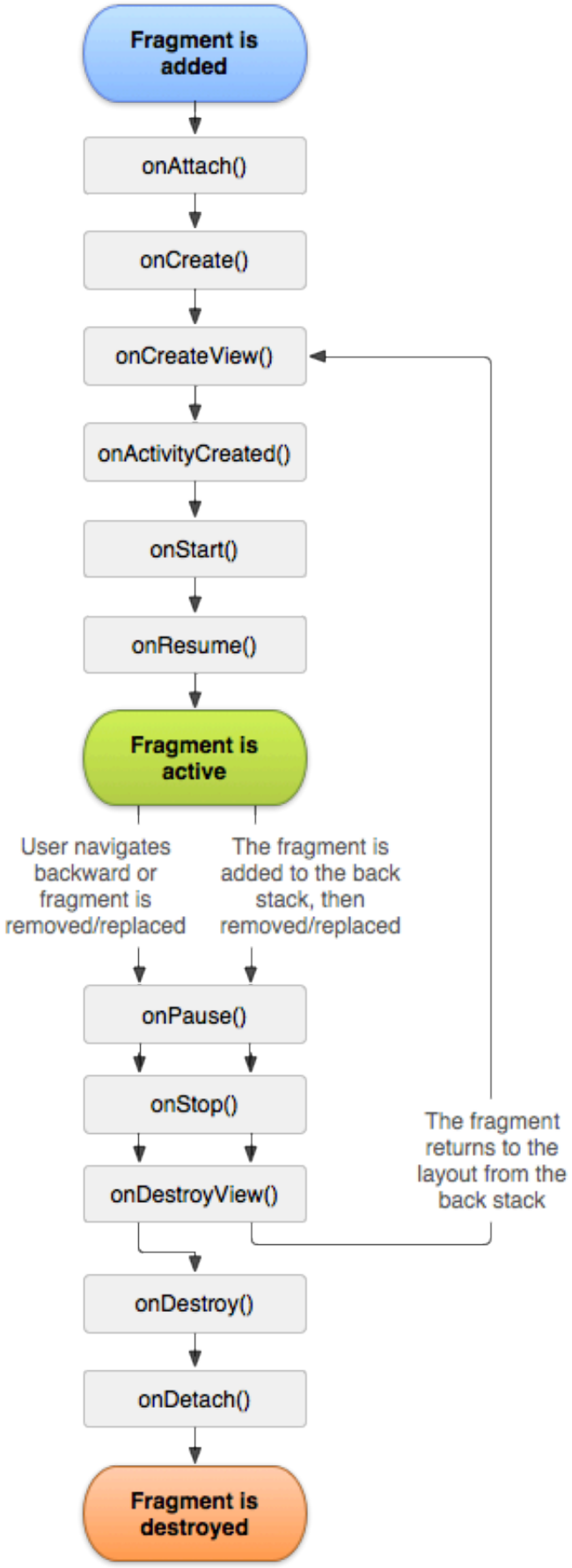
优点：5.0 以后出现的 JobService 是官方推荐的方式，比较稳定。

缺点：触发时机不够实时，JobService 的触发时机会是充电时，闲暇时等特殊时机或者时周期性运行。

Fragment

Fragment 真正的强大之处在于可以动态地添加到 Activity 当中，程序的界面可以定制的更加多样化，更加充分地利用平板的屏幕空间。

Fragment 的生命周期



- onAttach()：Fragment 和 Activity 建立关联的时候调用。需要使用 Activity 的引用或者使用 Activity 作为其操作的上下文，将在此回调方法中实现。

- onCreate(Bundle savedInstanceState)：此时的 Fragment 的 onCreate 回调时，该 fragment 还没有获得 Activity 的 onCreate() 已完成的通知，所以不能将依赖于 Activity 视图层次结构的代码放入此回调方法中。在 OnCreate() 回调方法中，应该尽量避免耗时操作。
- onCreateView(LayoutInflater inflater,ViewGroup container,Bundle savedInstanceState)： 为 Fragment 加载布局时调用。不要将视图层次结构附加到传入的 ViewGroup 父元素中（就是不要把初始化的 view 视图主动添加到 container 里面，不能出现 container.addView(v) 的操作），该关联会自动完成，如果在此回调中将碎片的视图层次结构附加到父元素，很可能会出现异常。
- onActivityCreated()： 当 Activty 中的 onCreate 方法执行完后调用。在调用 onActivityCreated() 之前，Activity 的视图层次结构已经准备好了。如果 Activity 和它的 Fragment 是从保存的状态重新创建的，此回调尤其重要，也可以在这里保证此 Activity 的其他所有 Fragment 已经附加到该 Activity 中了。
- onStart()\onResume()\onPause()\onStop()： 这些回调方法和 Activity 的回调方法进行绑定，也就是说与 Activity 中对应的生命周期相同。
- onDestoryView()：该回调方法在视图层次结构与 Fragment 分离之后调用。
- onDestory()：不再使用 Fragment 时调用。Fragment 仍然附加到 Activity 并依然可以找到，但是不能执行其他操作。
- onDetach()：Fragment 和 Activity 解除关联的时候调用。

setRetainInstance() 方法

此方法可以有效地提高系统的运行效率，对流畅性要求较高的应用可以适当采用此方法进行设置。

Fragment 有一个强大的功能，可以在 Activity 重新创建时可以不完全销毁 Fragment，以便 Fragment 可以恢复。在 onCreate() 方法中调用 setRetainInstance(true/false) 方法是最佳位置。

当在 onCreate() 方法中调用了 setRetainInstance(true) 后，Fragment 恢复时会跳过 onCreate() 和 onDestory() 方法，因此不能在 onCreate() 中放置一些初始化逻辑。

FragmentPagerAdapter 与 FragmentStatePagerAdapter 区别

使用 ViewPager 再结合 FragmentPagerAdapter 或者 FragmentStatePagerAdapter 可以制作一个 App 的主页。

而 FragmentPagerAdapter 和 FragmentStatePagerAdapter 的区别在于对于 Fragment 是否销毁：

- FragmentPagerAdapter：对于不再需要的 Fragment，选择调用 detach() 方法，仅销毁视图，并不会销毁 fragment 实例。
- FragmentStatePagerAdapter：会销毁不再需要的 Fragment，当当前事务提交以后，会彻底的将 fragment 从当前 Activity 的 FragmentManager 中移除，state 标明，销毁时，会将其 onSaveInstanceState(Bundle outState) 中的 bundle 信息保存下来，当用户切换回来，可以通过该 bundle 恢复生成新的 Fragment，也就是说，可以在 onSaveInstanceState(Bundle outState) 方法中保存一些数据，在 onCreate 中进行恢复创建。

使用 FragmentStatePagerAdapter 更省内存，但是销毁新建也是需要时间的。一般情况下，如果是制作主界面，就 3-4 个 Tab，那么可以选择使用 FragmentPagerAdapter，如果是用于 ViewPager 展示数量特别多的条目时，建议使用 FragmentStatePagerAdapter。

replace 与 add 的区别

两个方法不同之处：是否要清空容器再添加 Fragment 的区别，用法上 add 配合 hide 或是 remove 使用，replace 一般单独出现。

添加

一般会配合 hide 使用：transaction.add(R.id.fragment_container, oneFragment).hide(twoFragment).commit();

1. 第一个参数是容器 id，第二个参数是要添加的 fragment，添加不会清空容器中的内容，不停的往里面添加。
2. 不允许添加同一个 fragment 实例，这是非常重要的特点。如果一个 fragment 已经进来的话，再次添加会报异常错误的。
3. 添加进来的 fragment 都是可见的（visible），后添加的 fragment 会展示在先添加的 fragment 上面，在绘制界面的时候会会址所有可见的 view。
4. 所以大多数 add 都是和 hide 或者是 remove 同时使用的。这样可以节省绘制界面的时间，节省内存消耗，是推荐的用法。

替换

transaction.replace(R.id.fragment_container, oneFragment).commit();

1. 替换会把容器中的所有内容全部替换掉，有一些 app 会使用这样的做法，保持只有一个 fragment 在显示，减少了界面的层级关系。

相同之处：每次 add 和 replace 都要走一遍 fragment 的周期。

其实fragment一般不会这么简单使用，replace的使用场景一般不多，大多数是添加（add）和显示（show）配合隐藏（hide）来使用，这样首先避免相同类型的fragment的重复添加，提示开发者使用单例模式，已经添加过的fragment很多情况没有必要再次添加，而且还有把生命周期再走一遍，这是一种比较浪费的做法。

最合适的处理方式是这样的：

1.在add的时候，加上一个tab参数
transaction.add(R.id.content, IndexFragment,”Tab1");
2.然后当IndexFragment引用被回收置空的话，先通过
IndexFragment＝FragmentManager.findFragmentByTag(“Tab1");
找到对应的引用，然后继续上面的hide,show;

Binder 机制

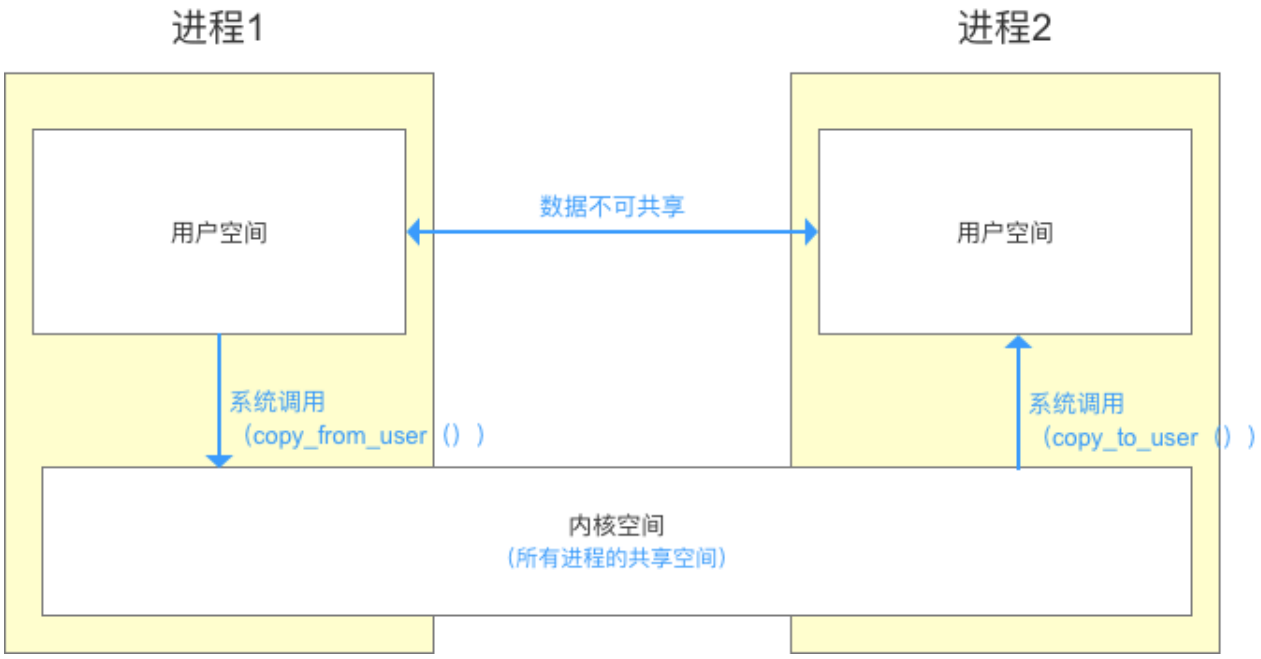
是一种实现 android 跨进程通讯的方式，由物理上的虚拟物理设备驱动，和 Binder 类组成。

android是基于linux内核的。

Linux 进程空间划分

- 一个进程空间分为 用户空间 & 内核空间（`Kernel`），即把进程内用户 & 内核隔离开来，所有进程共用1个内核空间。
- 二者区别：
  1. 进程间，用户空间的数据不可共享，所以用户空间 = 不可共享空间
  2. 进程间，内核空间的数据可共享，所以内核空间 = 可共享空间
- 进程内用户空间 & 内核空间进行交互需通过系统调用，

  主要通过函数：
  copy_from_user（）：将用户空间的数据拷贝到内核空间
  copy_to_user（）：将内核空间的数据拷贝到用户空间

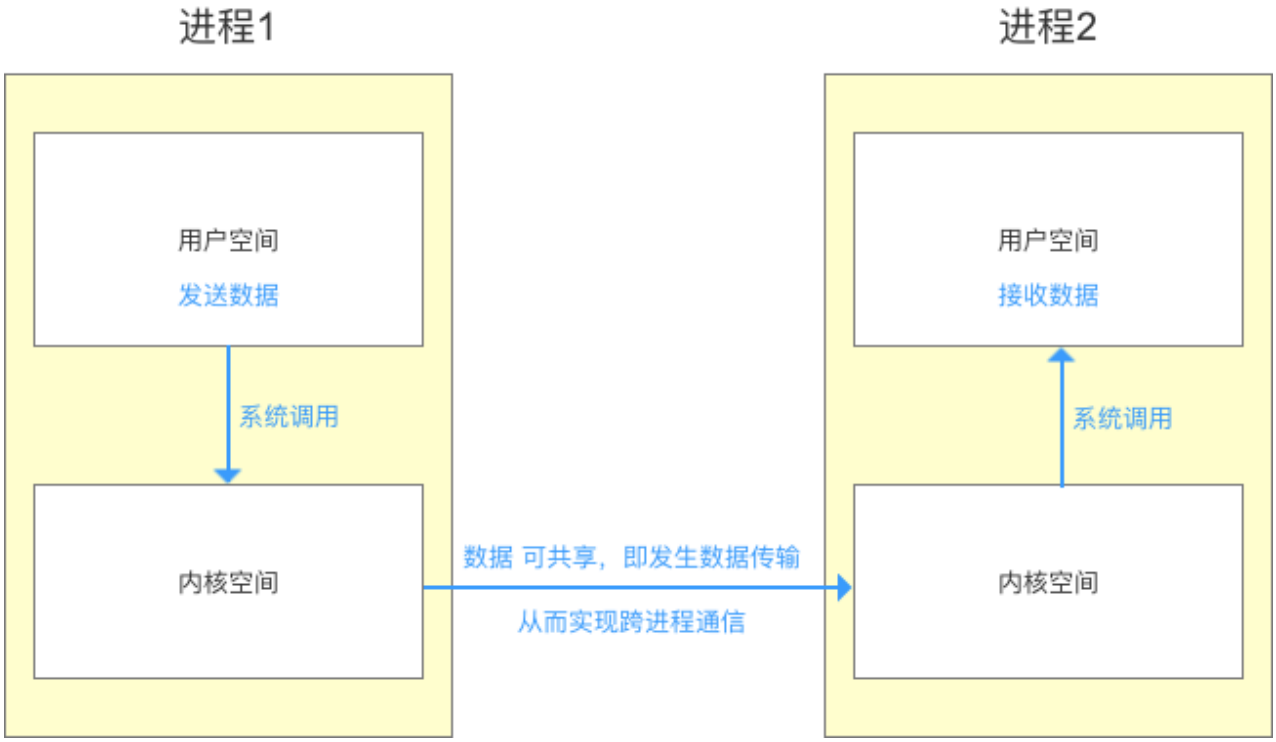


进程隔离

为了保证安全性 & 独立性，一个进程不能直接操作或者访问另一个进程，即 Android 的进程是相互独立、隔离的。

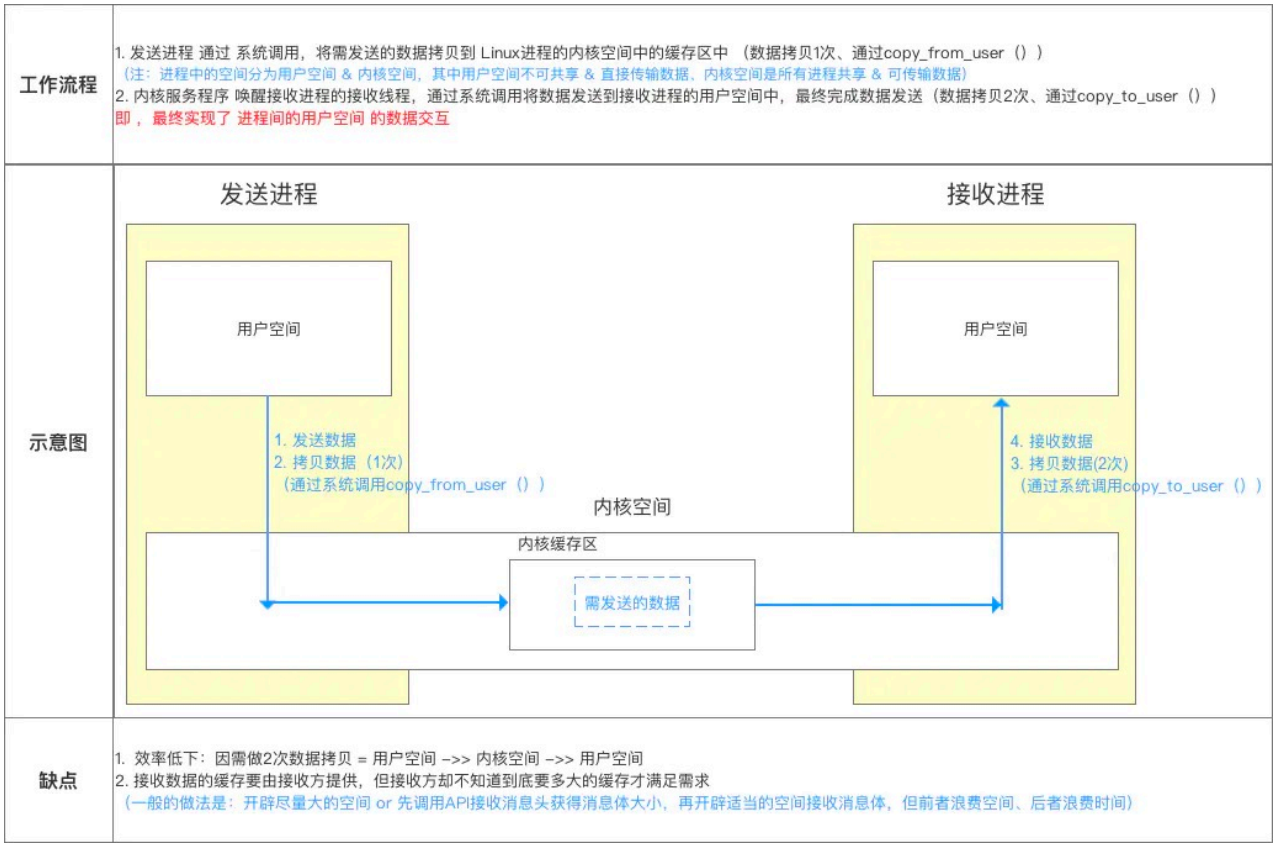
跨进程通信（IPC）

- 隔离后，由于某些需求，进程间需要合作 / 交互
- 跨进程间通信的原理
  1. 先通过进程间的内核空间进行数据交互
  2. 再通过进程内的用户空间 & 内核空间进行数据交互，从而实现进程间的用户空间的数据交互



而`Binder`，就是充当 连接 两个进程（内核空间）的通道。

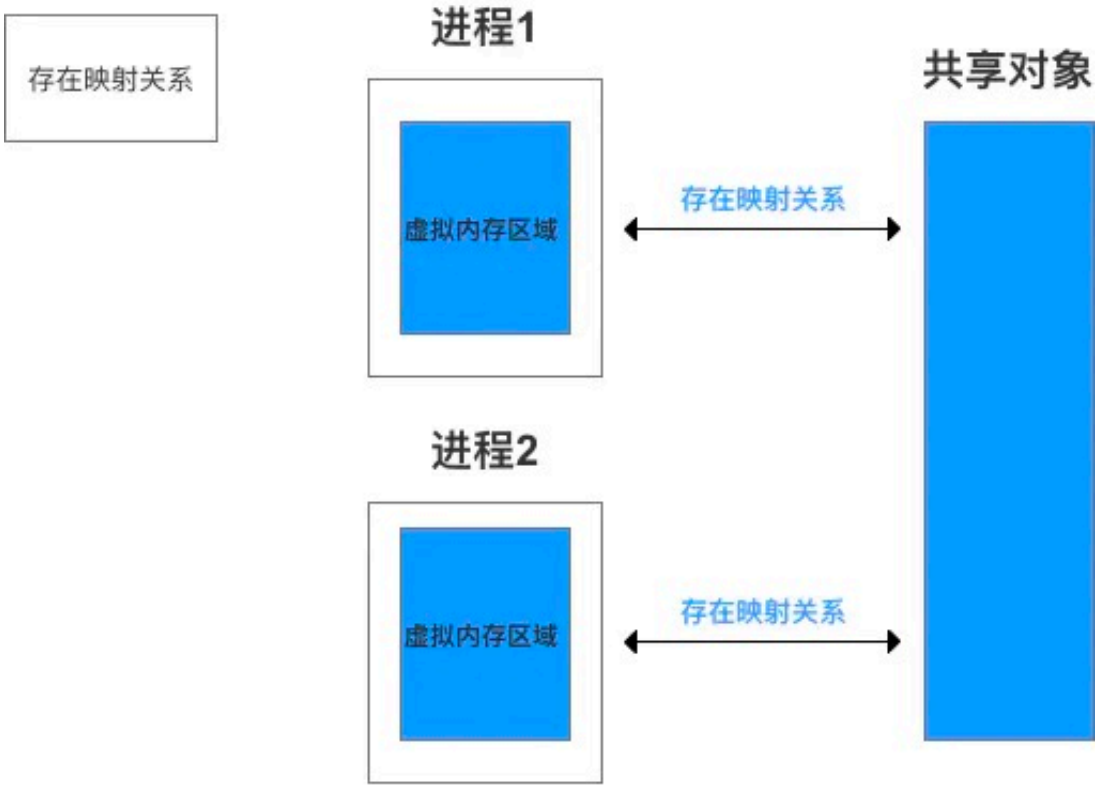
传统跨进程通信的基本原理

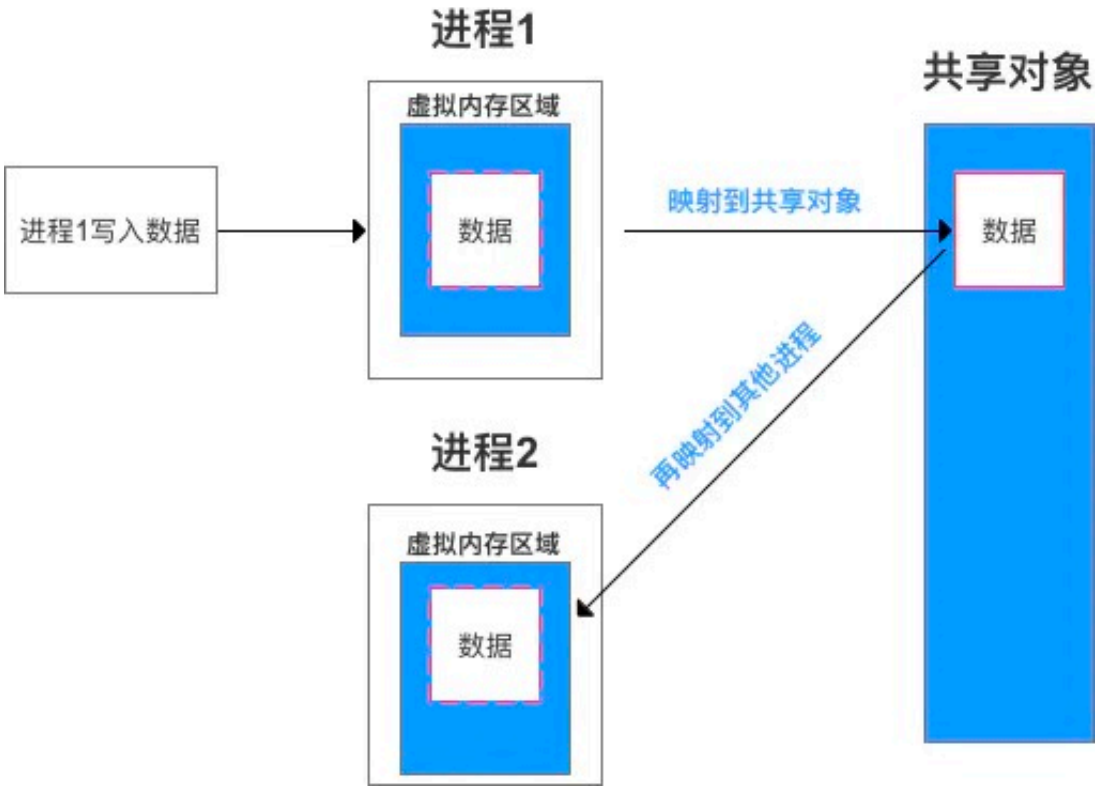


而 Binder 的作用则是：连接两个进程，实现了mmap() 系统调用，主要负责创建数据接收的缓存空间 & 管理数据接收缓存，传统的跨进程通信需拷贝数据 2 次，但 Binder 机制只需 1 次，主要是使用到了内存映射。

什么是内存映射?

内存映射是关联 进程中的1个虚拟内存区域 & 1个磁盘上的对象，使得二者存在映射关系。



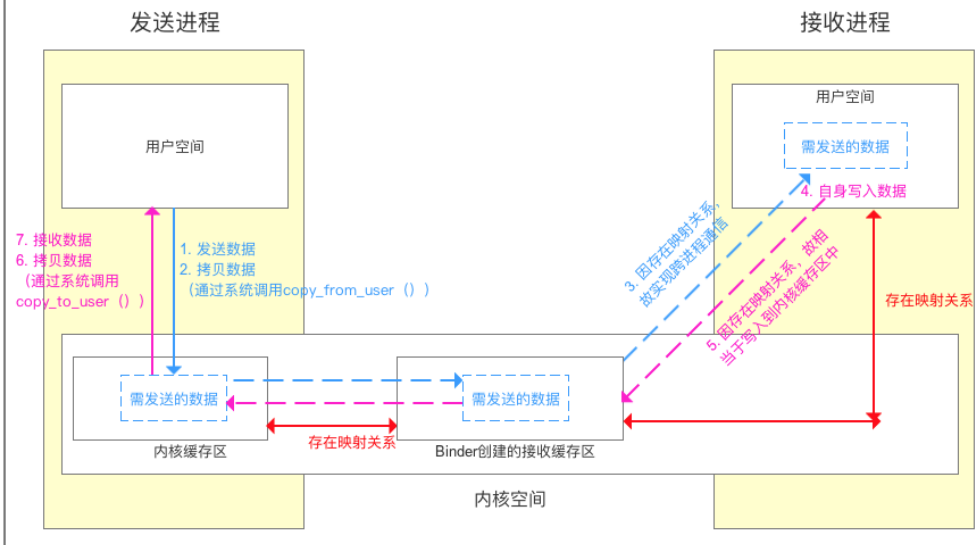


内存映射的实现过程主要是通过 Linux 系统下的系统调用函数： mmap（），该函数的作用 = 创建虚拟内存区域 + 与共享对象建立映射关系。

内存映射的作用

1. 实现内存共享：如跨进程通信
2. 提高数据读 / 写效率 ： 如文件读 / 写操作

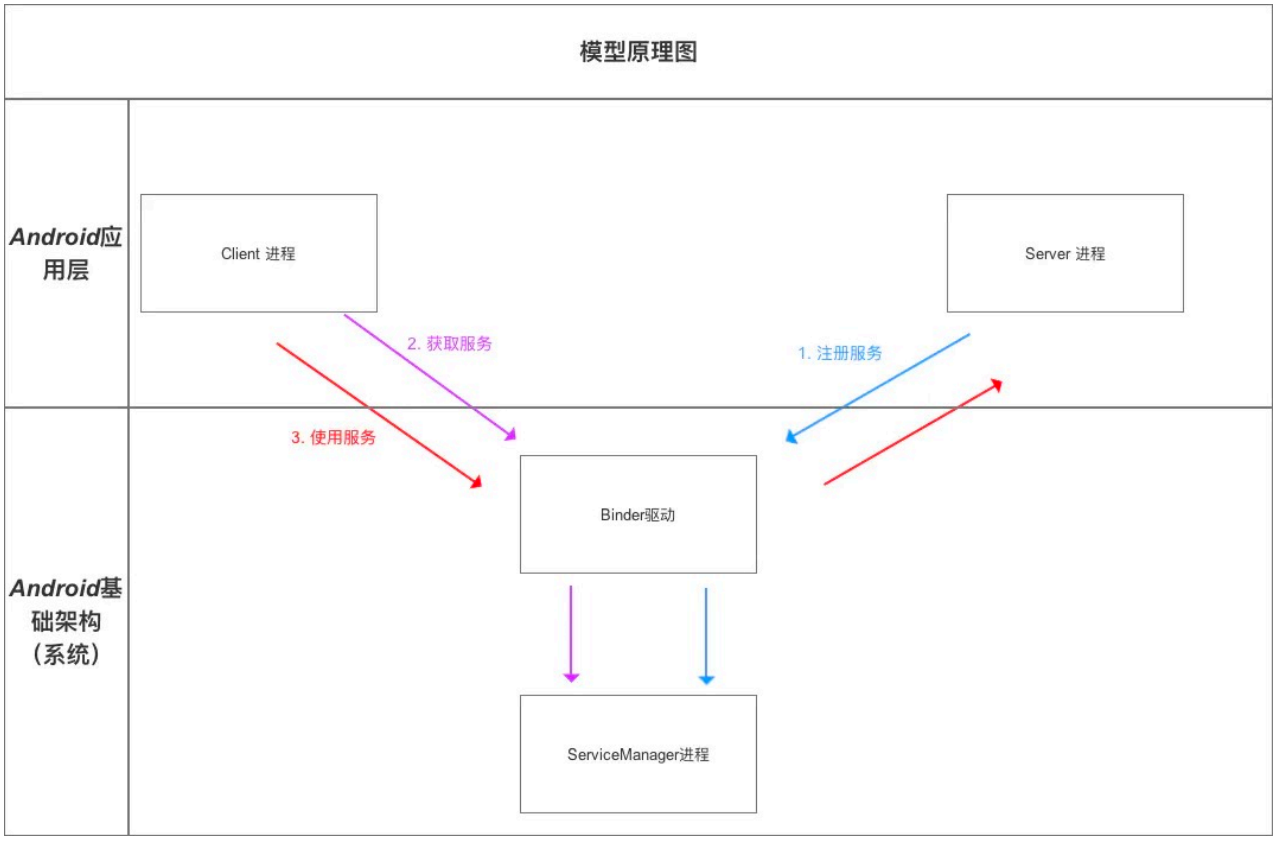
Binder

<table>
<tr><th>步骤</th><th colspan="3">过程描述</th></tr>
<tr><td>1. 注册服务</td><td colspan="2">1. Server进程 向 Binder驱动 发起服务注册请求
2. Binder驱动 将注册请求转发给Service Manager进程
3. Service Manager进程 添加该Server进程（即已注册服务）</td><td>此时，ServiceManager进程拥有了Server进程的信息</td></tr>
<tr><td>2. 获取服务</td><td colspan="2">1. Client 向 Binder 驱动发起获取服务的请求，传递要获取的服务名称
2. Binder 驱动将该请求转发给 ServiceManager 进程
3. ServiceManager 查找到 Client 需要的 Server 对应的服务信息
4. 通过 Binder 驱动将上述服务信息返回给 Client进程</td><td>此时，Client进程与 Server进程已经建立了连接</td></tr>
<tr><td rowspan="6">3. 使用服务</td><td>步骤1： Binder驱动为跨进程通信作准备：实现内存映射
（调用mmap（）系统函数）</td><td>1. Binder驱动 创建一块 接收缓存区
2. 实现地址映射关系：即 根据 ServiceManager进程里的Server信息找到对应的Server 进程
，实现 内核缓存区 和 Server 进程用户空间地址 同时映射到 同1个接收缓存区中
（注：此时仅创建了虚拟区间 & 映射关系，但并无将传输数据）</td></tr>
<tr><td>步骤2： Client进程 将参数数据发送到Server进程</td><td>1. Client进程 通过 系统调用copy_from_user（）发送数据到内核空间中的缓存区；（当前线程被挂起）
（由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射Binder创建的接收缓存区中），故相当于也发送到了Server进程的用户空间地址，即Binder驱动实现了跨进程通信）
3. Binder驱动 通知Server 进程执行 解包</td></tr>
<tr><td>步骤3： Server进程 根据Client进程要求 调用目标方法</td><td>1. 收到Binder驱动通知后，Server 进程从线程池中取出 线程，进行数据解包 & 调用目标方法
2. 将最终执行结果写入到自己的共享内存中</td></tr>
<tr><td>步骤4：Server进程 将目标方法的结果 返回给Client进程</td><td>1. 将最终执行结果写入存在映射的用户空间的内存区域中
（由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射Binder创建的接收缓存区中），故相当于也发送到了内核缓存区中）
2. Binder驱动通知Client进程获得返回结果（此时Client进程之前被挂起的线程被重新唤醒）
3. Client进程 通过 系统调用copy_to_user（）从内核缓存区接收Server进程返回的数据</td></tr>
<tr><td>示意图</td><td colspan="2"> 
</td></tr>
<tr><td>优点</td><td colspan="2">• 传输效率高：每次单向通信数据拷贝次数少（1次）、用户空间 & 内核空间可直接通过共享对象直接交互
• 为接收进程 分配了不确定大小的接收缓存区</td></tr>
</table>

所以 Binder 驱动一共有两个作用

1. 创建接受缓存区
2. 通知 client 和 service 数据准备就绪
3. 管理线程

Binder驱动属于进程空间的内核空间，可进行进程间 & 进程内交互



Binder请求的线程管理

Binder 模型的线程管理采用 Binder 驱动的线程池，并由 Binder 驱动自身进行管理。

一个进程的 Binder 线程数默认最大是 16，超过的请求会被阻塞等待空闲的 Binder 线程。

Android中的Binder实现机制

android 中提供了 Binder 实体类，Binder 实体是 Server进程在 Binder 驱动中的存在形式。

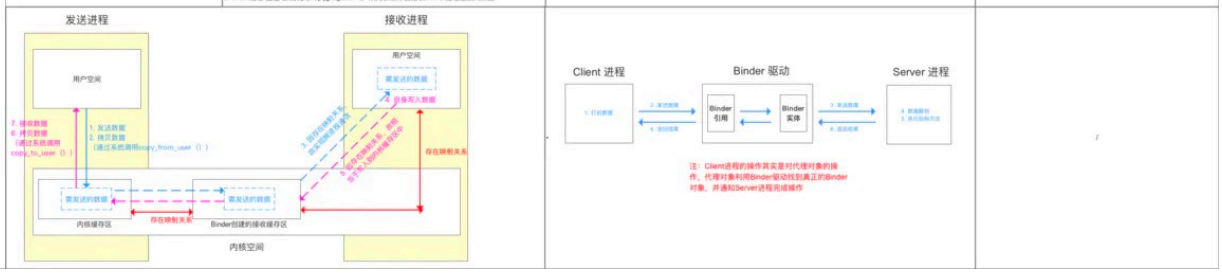
该对象保存 Server 和 ServiceManager 的信息（保存在内核空间中），Binder 驱动通过内核空间的 Binder 实体找到用户空间的 Server 对象，注册服务后，Binder 驱动持有 Server 进程创建的 Binder 实体。

```
public class Binder implement IBinder{
    // Binder 机制在 Android 中的实现主要依靠的是 Binder 类，其实现了 IBinder 接口
    // IBinder 接口：定义了远程操作对象的基本接口，代表了一种跨进程传输的能力
    // 系统会为每个实现了 IBinder 接口的对象提供跨进程传输能力
    // 即 Binder 类对象具备了跨进程传输的能力

        void attachInterface(IInterface plus, String descriptor);
          // 作用：
          // 1. 将（descriptor，plus）作为（key,value）对存入到 Binder 对象中的一个
Map<String,IInterface> 对象中
          // 2. 之后，Binder 对象可根据 descriptor 通过 queryLocalIInterface（） 获
得对应 IInterface 对象（即 plus）的引用，可依靠该引用完成对请求方法的调用

        IInterface queryLocalInterface(Stringdescriptor) ；
        // 作用：根据 参数 descriptor 查找相应的 IInterface 对象（即plus引用）
```

```
        boolean onTransact(int code, Parcel data, Parcel reply, int flags);
        // 定义：继承自 IBinder 接口的
        // 作用：执行 Client 进程所请求的目标方法（子类需要复写）
        // 参数说明：
        // code：Client 进程请求方法标识符。即 Server 进程根据该标识确定所请求的目标方法
        // data：目标方法的参数。（Client 进程传进来的，此处就是整数 a 和 b）
        // reply：目标方法执行后的结果（返回给 Client 进程）
        // 注：运行在 Server 进程的 Binder 线程池中；当 Client 进程发起远程请求时，远程请求会要求系统底层执行回调该方法

        final class BinderProxy implements IBinder {
         // 即 Server 进程创建的 Binder 对象的代理对象类
         // 该类属于 Binder 的内部类
        }
        // 回到分析1原处
}
```

过程描述	代码实现	备注
1. Client 向 Binder 驱动发起获取服务的请求，并传递要获取的服务名称	Client 通过 bindService（）绑定Server进程中注册的Server 进程	
2. Binder 驱动将该请求转发给 ServiceManager 进程	调用Server进程 的onBind（）得到创建的Binder对象的代理对象：BinderProxy对象	• 即Binder对象的引用 • 该对象类即上述提到的Binder类的内部类
3. ServiceManager 查找到 Client 需要的 Server 对应的 Binder 实体的 Binder 引用信息并返回给Binder驱动		
4. Binder 驱动将上述 Binder代理对象返回给 Client	Client进程通过 调用 onServiceConnected（）获得了Server进程创建的Binder对象的代理对象BinderProxy对象	• Client进程获得的不是创建的真正的Binder对象，而是其代理对象 • Client进程的操作其实是对代理对象的操作，代理对象利用Binder驱动最终让真正的Binder对象完成操作

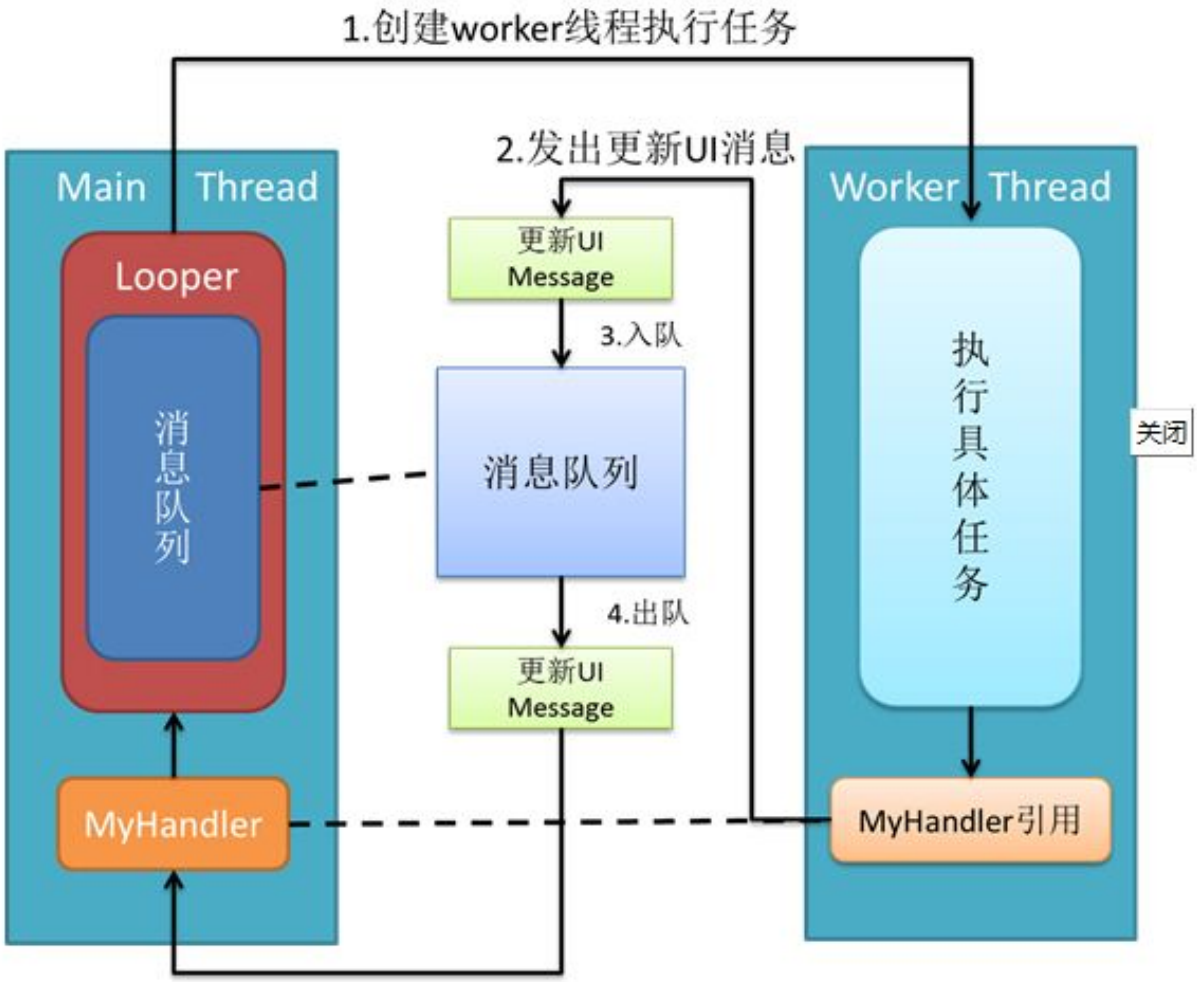
流程总结：客户端通过 bindService，通过 Binder 驱动查询 ServiceManager 是否已经注册该服务，如果没有注册，Service 进程会向 Binder 驱动发起服务注册请求，一旦注册，调用该服务的 onBind 返回一个 Binder 对象到 Binder 驱动，已经注册则意味着 Binder 驱动内包含这个 Binder 对象，Binder 驱动返回一个 BinderProxy 对象，并通过回调，传递给客户端，客户端通过这个 BinderProxy(在 java 层仍然是 Binder 对象)操作 Binder 驱动内的 Binder 对象（transact 方法），Binder 驱动含有很多的 Binder 对象，它们是通过 InterfaceToken 区分不同服务的。

<table>
<tr><th>步骤</th><th>过程描述</th><th>代码实现</th><th>备注</th></tr>
<tr><td>1. 注册服务</td><td colspan="2">1. Server进程 向 Binder驱动 发起服务注册请求
2. Binder驱动 将注册请求转发给Service Manager进程
3. Service Manager进程 添加该Server进程（即已注册服务，此时，ServiceManager进程拥有了Server进程的信息）</td><td>Server进程 创建 一个 Binder 实体对象
（注册服务后，Binder驱动将有 Server进程创建的Binder实体）</td><td>• Binder 实体是 Server进程 在 Binder 驱动中的存在形式
• 该对象保存 Server 和 ServiceManager 的信息（保存在内核空间中）
• Binder 驱动通过 内核空间的Binder 实体 找到用户空间的Server对象</td></tr>
<tr><td rowspan="4">2. 获取服务</td><td colspan="2">1. Client 向 Binder 驱动发起获取服务的请求，并传递要获取的服务名称</td><td>1. Client 通过 bindService（）绑定Server进程中注册的Server 进程</td><td></td></tr>
<tr><td colspan="2">2. Binder 驱动将该请求转发给 ServiceManager 进程</td><td rowspan="2">2. 调用Server进程 的onBind（）得到创建的Binder对象的代理对象：BinderProxy对象</td><td rowspan="2">• 即Binder对象的引用
• 该对象类即上述提到的Binder类的内部类</td></tr>
<tr><td colspan="2">3. ServiceManager 查找到 Client 需要的 Server 对应的 Binder 实体的 Binder 引用信息并返回给Binder驱动</td></tr>
<tr><td colspan="2">4. Binder 驱动将上述 Binder代理对象返回给 Client（此时，Client进程与 Server进程已经建立了连接）</td><td>3. Client进程通过 调用 onServiceConnected（）获得了Server进程创建的Binder对象的代理对象BinderProxy对象</td><td>• Client进程获得的不是创建的真正的Binder对象，而是其代理对象
• Client进程的操作其实是对代理对象的操作，代理对象利用Binder驱动最终让真正的Binder对象完成操作</td></tr>
<tr><td rowspan="5">3. 使用服务</td><td>步骤1： Binder驱动为跨进程通信作准备：实现内存映射
（调用mmap（）系统函数）</td><td>1. Binder驱动 创建一块 接收缓存区
2. 实现地址映射关系：即 根据 ServiceManager进程里的Server信息找到对应的Server 进程
• 实现 内核缓存区 和 Server 进程用户空间地址 同时映射到 同1个接收缓存区中
（注：此时仅创建了虚拟区间 & 映射关系，但并无传输数据）</td><td>mmap（）系统函数调用</td><td>操作系统的底层实现，可不做了解</td></tr>
<tr><td>步骤2： Client进程 将参数数据发送到Server进程</td><td>1. Client进程 通过 系统调用copy_from_user（）发送数据到内核空间中的缓存区（当前线程被挂起）
（由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射到Binder创建的接收缓存区中），故相当于也发送到了Server进程的用户空间地址，即Binder驱动实现了跨进程通信）
3. Binder驱动 通知Server 进程执行 解包</td><td>1. Client进程 将需要传送的数据写入到Parcel对象中
2. 通过 调用代理对象的transact（） 将 上述数据发送到Binder驱动
3. Binder驱动根据 代理对象 找到对应的具备Binder对象所在的Server 进程（系统自动执行）
4. Binder驱动把 数据 发送到Server 进程中，并通知Server 进程执行解包（系统自动执行）</td><td>• 传送的数据包括目标方法的标识符、Parcel对象（存放了目标方法的参数、方法对象的标识符）、存放执行结果的reply参数
• 在发送数据后，Client进程的该线程会暂时被挂起，所以，若Server进程执行的耗时操作，请不要使用主线程，以防止ANR</td></tr>
<tr><td>步骤3： Server进程 根据Client进程要求 调用目标方法</td><td>1. 收到Binder驱动通知后，Server 进程从线程池中取出 线程，进行数据解包 & 调用目标方法
2. 将最终执行结果写入到自己的共享内存中</td><td>收到Binder驱动通知后，Server 进程通过回调Binder对象onTransact()进行数据解包 ，调用目标方法，最终将结果返回(写入reply参数中)</td><td rowspan="2">均在onTransact（）进行</td></tr>
<tr><td>步骤4： Server进程 将目标方法的结果 返回给Client进程</td><td>1. 将最终执行结果写入存在映射的用户空间的内存区域中
（由于 内核缓存区 & 接收进程的用户空间地址 存在映射关系（同时映射到Binder创建的接收缓存区中），故相当于也发送到了内核缓存区中）
2. Binder驱动通知Client进程获得返回结果（此时Client进程之前被挂起的线程被重新唤醒）
3. Client进程 通过 系统调用copy_to_user（）从内核缓存区接收Server进程返回的数据</td><td>1. Binder驱动根据 代理对象 指明路 将结果返回 并通知Client进程获取返回结果
2. 通过代理对象 获取reply参数，从而接收结果（之前被挂起的线程被唤醒）</td></tr>
<tr><td colspan="3"> 
</td></tr>
</table>

Handler

消息机制的架构

消息机制的运行流程：在子线程执行完耗时操作，当 Handler 发送消息时，将会调用 MessageQueue.enqueueMessae，向消息队列中添加消息。当通过 Looper.loop 开启循环后，会不断地从线程池中读取消息，即调用 MessageQueue.next，然后调用目标 Handler (即发送该消息的 Handler) 的 dispatchMessage 方法传递消息，然后返回到 Handler 所在线程，目标 Handler 收到消息，调用 handleMessage 方法，接收消息并处理消息。



Message、Handler 和 Looper 三者之间的关系：每个线程中只能存在一个 Looper，Looper 是保存在 ThreadLocal 中的。

主线程（UI 线程）已经创建了一个 Looper，所以在主线程不需要再创建 Looper，但是在其他线程中需要创建 Looper。

每个线程中可以有多个 Handler，即一个 Looper 可以处理来自多个 Handler 的消息。

Looper 中维护一个 MessageQueue，来维护消息队列，消息队列中的 Message 可以来自不同的 Handler。

1. Handler 的背后有着 Looper 以及 MessageQueue 的协助，三者通力合作。
2. Looper 负责关联线程以及消息的分发。在该线程下循环从 MessageQueue 获取 Message，分发给 Handler。在创建 Handler 之前一定需要先创建 Looper。Looper 有退出的功能，但是主线程的 Looper 不允许退出。异步线程的 Looper 需要自己调用 `Looper.myLooper().quit();` 退出。
3. MessageQueue 负责消息的存储与管理。负责管理由 Handler 发送过来的 Message。
4. Handler 负责发送并处理消息。面向开发者，提供 API，并隐藏背后实现的细节。
5. Looper.loop() 是个死循环，会不断调用 MessageQueue.next() 获取 Message，并调用 msg.target.dispatchMessage(msg) 回到了 Handler 来分发消息，以此来完成消息的回调。
6. Runnable 被封装进了 Messgage，可以说是一个特殊的 Message。

7. Handler 发送的消息由 MessageQueue 存储管理，并由 Looper 负责回调消息到 handleMessage()。
8. 消息处理的方法调用栈：Looper.loop() -> MessageQueue.next() -> Message.target.dispatchMessage() -> Handler.handleMessage() ，所以 Handler.handleMessage() 所在的线程是 Looper.loop() 方法被调用的线程，也可以说成 Looper 所在的线程，并不是创建 Handler 的线程。

   平时使用 Handler 的时候会从异步发送消息到 Handler，而 Handler 的 handleMessage() 方法是在主线程调用的，所以消息就从异步线程切换到了主线程。最终又调用到了重写的 handleMessage(Message msg) 方法来做处理子线程发来的消息或者调用 handleCallback(Message message) 去执行子线程中定义并传过来的操作。
9. 使用内部类的方式使用 Handler 可能会造成内存泄漏，即便在 Activity.onDestory 里移除延时消息，必须要写成静态内部类。

Handler 引起的内存泄漏原因以及最佳解决方案

Handler 允许发送延时消息，如果在延时期间用户关闭了 Activity，那么该 Activity 会泄漏。

这个泄漏是因为 Message 会持有 Handler，而又因为 Java 的特性，内部类会持有外部类，使得 Activity 会被 Handler 持有，这样最终就导致 Activity 泄漏。

解决该问题的最有效的方法是：将 Handler 定义成静态的内部类，在内部持有 Activity 的弱引用，并及时移除所有消息。

而单纯的在 onDestory 中移除消息并不保险，因为 onDestory 并不一定执行。

为什么主线程不会因为 Looper.loop() 里的死循环卡死或者不能处理其他事务?

为什么不会卡死?

handler 机制是使用 pipe 来实现的，主线程没有消息处理时会阻塞在管道的读端。

binder 线程会往主线程消息队列里添加消息，然后往管道写端写一个字段，这样就能唤醒主线程从管道读端返回，也就是说 queue.next() 会调用返回。

主线程大多数都是出于休眠状态，并不会消耗大量 CPU 资源。

既然是死循环又如何去处理其他事务呢?

答案是通过创建新线程的方式。

在 main() 方法里调用了 thread.attach(false)，这里便会创建一个 Binder 线程（具体是指 ApplicationThread，Binder 的服务端，用于接收系统服务 AMS 发送来的事件），该 Binder 线程通过 Handler 将 Message 发送给主线程。

ActivityThread 对应的 Handler 是一个内部类 H，里面包含了启动 Acitivity、处理 Activity 生命周期等方法。

ThreadLocal

ThreadLocal 并不是一个 Thread，而是 Thread 的局部变量，它的作用是可以在每个线程中存储数据。

ThreadLocal 是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，数据存储以后，只是在指定线程中可以获取到存储的数据，对于其他线程来说无法获取到数据。

当使用 ThreadLocal 维护变量时，ThreadLocal 为每个使用该变量的线程提供独立的变量副本，所以每一个线程都可以独立地改变自己的副本，而不会影响其他线程所对应的副本。

AsyncTask 的知识

Android UI 是线程不安全的，如果想要在子线程里进行 UI 操作，就需要借助 Android 的异步消息处理机制 ，为了更加方便在子线程中更新 UI 元素，Android 5.1 版本就引入了一个 AsyncTask 类，使用它就可以非常灵活方便的从子线程切换到 UI 线程。

AsyncTask 内部封装了 Thread 和 Handler ，可以在后台进行计算并且把计算的结果及时更新到 UI 上，而这些正是 Thread + Handler 所做的事情，AsyncTask 的作用就是简化 Thread + Handler，能够通过更少的代码来完成一样的功能。

重写 AsyncTask 中的几个方法才能完成对任务的定制。经常需要去重写的方法有以下四个：

1. onPreExecute

   这个方法会在后台任务开始执行之前调用，用于进行一些界面上的初始化操作，比如显示一个进度条对话框等。

   所在线程：UI 线程

2. doInBackground(Params ...)

   这个方法中的所有代码都会在子线程中运行，应该在这里处理所有的耗时任务。任务一旦完成就可以通过 return 语句来将任务的执行结果进行返回，如果 AsyncTask 的第三个泛型参数执行的 Void，就可以不返回任务执行结果。注意，在这个方法中是不可以进行 UI 操作的，如果需要更新 UI 元素，比如说反馈当前任务的执行进度，可以调用 publishProgress(Progress...) 方法来完成。

   所在线程：后台线程

3. onProgressUpdate(Progress...)

   当在后台任务中调用了 publishProgress(Progress...) 方法后，这个方法就很快会被调用，方法中携带的参数就是在后台任务中传递过来的。这个方法中可以对 UI 进行操作，利用参数中的数值就可以对界面元素进行相应的更新。

   所在线程：UI 线程

4. onPostExecute(Boolean result)

   运行结果。

   所在线程：UI 线程

View

Activity 的布局绘制过程

setContentView 会将整个布局文件都解析完成并形成一个完整的 Dom 结构，并设置最顶部的根布局。在 resume 的时候才会进行视图的绘制操作，通过调用 requestLayout() 最终调用到 performTraversales() 方法，performTraversales() 方法会依次调用 View 的 measure、layout、draw 步骤将视图显示在屏幕上。

View 绘制流程

Android 中的任何一个布局、任何一个控件其实都是直接或间接继承自 View 的，如 TextView、Button、ImageView、ListView 等。

每一个视图的绘制过程都必须经历三个最主要的阶段，即 onMeasure()、onLayout() 和 onDraw()。

onMeasure()

measure 是测量的意思，那么 onMeasure() 方法顾名思义就是用来测量视图的大小的。

View 系统的绘制流程会从 ViewRoot 的 performTravesals() 方法中开始，在其内部调用 View 的 measure() 方法。measure() 方法接收两个参数，widthMeasureSpec 和 heightMeasureSpec，这两个值分别用于确定视图的宽度和高度的规格和大小。

MeasureSpec 的值由 specSize 和 specMode 共同组成的，其中 specSize 记录的是大小，specMode 记录的是规格。

specMode 一共有三种类型：

1. EXACTLY

   表示父视图希望子视图的大小应该是由 specSize 的值来决定的。

   系统默认会按照这个规则来设置子视图的大小，开发人员当然也可以按照自己的意愿设置成任意的大小。

2. AT_MOST

   表示子视图最多只能是 specSize 中指定的大小，开发人员应该尽可能小的去设置这个视图，并且保证不会超过 specSize。

   系统默认会按照这个规则来设置子视图的大小，开发人员当然也可以按照自己的意愿设置成任意的大小。

3. UNSPECIFIED

   表示开发人员可以将视图按照自己的意愿设置成任意的大小，没有任何限制。

   这种情况比较少见，不太会用到。

WRAP_CONTENT 对应的是 AT_MOST，MATCH_PARENT 与具体的数值对应的是 EXACTLY。

ViewRootImpl 的 performTraversals 方法中会performMeasure() 方法，在 performMeasure() 方法中调用了 View 的 measure() 方法。而 View 的 measure() 方法调用了 onMeasure() 方法去真正测量宽高。onMeasure 方法默认会调用 getDefaultSize() 方法来获取视图的大小。之后会在 onMeasure() 方法中调用 setMeasuredDimension() 方法来设定测量出的大小。

视图大小的控制是由父视图、布局文件以及视图本身共同完成的，父视图会提供给子视图参考的大小，而开发人员可以在 XML 文件中指定视图的大小，然后视图本身会对最终的大小进行拍板。

onLayout()

measure 过程结束后，视图的大小就已经测量好了，接下来就是 layout 的过程了。正如其名字所描述的一样，这个方法是用于给视图进行布局的，也就是确定视图的位置。

ViewRoot 的 performTraversals() 方法会在 measure 结束后继续执行，会调用 performLayout() 方法，在 performLayout() 方法中会调用 View 的 layout() 方法来执行此过程。

ViewRootImple 的 performLayout() 方法中调用了 view 的 layout 方法。

在 layout() 方法中，首先会调用 setFrame() 方法来判断视图的大小是否发生过变化，以确定有没有必要对当前的视图进行重绘，同时还会在这里把传递过来的四个参数分别赋值给 mLeft、mTop、mRight 和 mBottom 这几个变量。接下来会调用 onLayout() 方法。

View 的 onLayout 是一个空方法，因为 onLayout() 过程是为了确定视图在布局中所在的位置，而这个操作应该是由布局来完成的，即父视图决定子视图的显示位置，那么就是 ViewGroup 的 onLayout() 方法。

在 FrameLayout 的 onLayout() 方法中，对子视图进行循环处理，调用子视图的 layout() 方法来确定它在 FrameLayout 布局中的位置，传入的 childLeft、childTop、childLeft + width、childTop + height，分别代表着子视图在 FrameLayout 中左上右下四个点的坐标。其中，调用 childView.getMeasuredWidth() 和 childView.getMeasuredHeight() 方法得到的值就是在 onMeasure() 方法中测量出的宽和高。

在 onLayout() 过程结束后，就可以调用 getWidth() 方法和 getHeight() 方法来获取视图的宽高了。

getMeasureWidth() 和 getWidth() 方法的区别

1. 首先 getMeasureWidth() 方法在 measure() 过程结束后就可以获取到了，而 getWidth() 方法要在 layout() 过程结束后才能获取到。
2. 另外，getMeasureWidth() 方法中的值是通过 setMeasuredDimension() 方法来进行设置的，而 getWidth() 方法中值则是通过视图右边的坐标减去左边的坐标计算出来的。

onDraw()

ViewRootImpl 的 performTravers() 方法在调用了 performLayout() 方法之后，会调用 performDraw() 方法。

在 performDraw() 方法中，调用了 ViewRootImpl 的 draw() 方法。

在 draw() 方法中，调用了 ViewRootImpl 的 drawSoftware() 方法。

drawSoftware() 方法中创建出一个 Canvas 对象，然后调用 View 的 draw() 方法来执行具体的绘制工作。

draw() 方法内部的绘制过程总共可以分为六部，其中第二步和第五步在一般情况下很少用到。

- 第一步：绘制背景

  第一步的作用是对视图的背景进行绘制。调用了 drawBackground() 方法来绘制。

- 第二步：如果有必要，保存画布层以准备褪色（不常用）

- 第三步：绘制视图的内容

  第三步的作用是对视图的内容进行绘制。调用了 onDraw() 方法。

  而 onDraw() 是一个空方法，因为每个视图的内容部分肯定都是各不相同的，这部分的功能交给子类来实现是理所当然的。

- 第四步：绘制子视图

  第四步的作用是对当前视图的所有子视图进行绘制。调用了 dispatchDraw() 方法。

  dispatchDraw() 方法也是一个空方法，也是交由子类去实现，如果当前的视图没有子视图，那么就不需要进行绘制了。比如 TextView 继承 View，但是没有重写 dispatchDraw() 方法，它没有子视图，也就没有必要实现这个方法，而 ViewGroup 类就重写了 dispatchDraw() 方法，去实现子视图的绘制。

- 第五步：如果有必要，绘制褪色边缘并恢复层（不常用）

  绘制装饰（例如滚动条）：第六步的作用是对视图的滚动条进行绘制。任何一个视图都是有滚动条的，只是一般情况下都没有让它显示出来。

通过以上流程分析，发现 View 是不会绘制内容部分的，因此需要每个视图根据想要展示的内容来自行绘制。绘制的方式主要是借助 Canvas 这个类，它会作为参数传入到 onDraw() 方法中，供给每个视图使用。

视图状态与重绘流程

invalidate() 方法虽然最终会调用 performTraversals() 方法中，但这时 measure 和 layout 流程是不会重新执行的，因为视图没有强制重新测量的标志位，而且大小也没有发生过变化，所以这时只有 draw 流程可以得到执行。

而如果希望视图的绘制流程可以完完整整地重新走一遍，就不能使用 invalidate() 方法，而应该调用 requestLayout() 了。

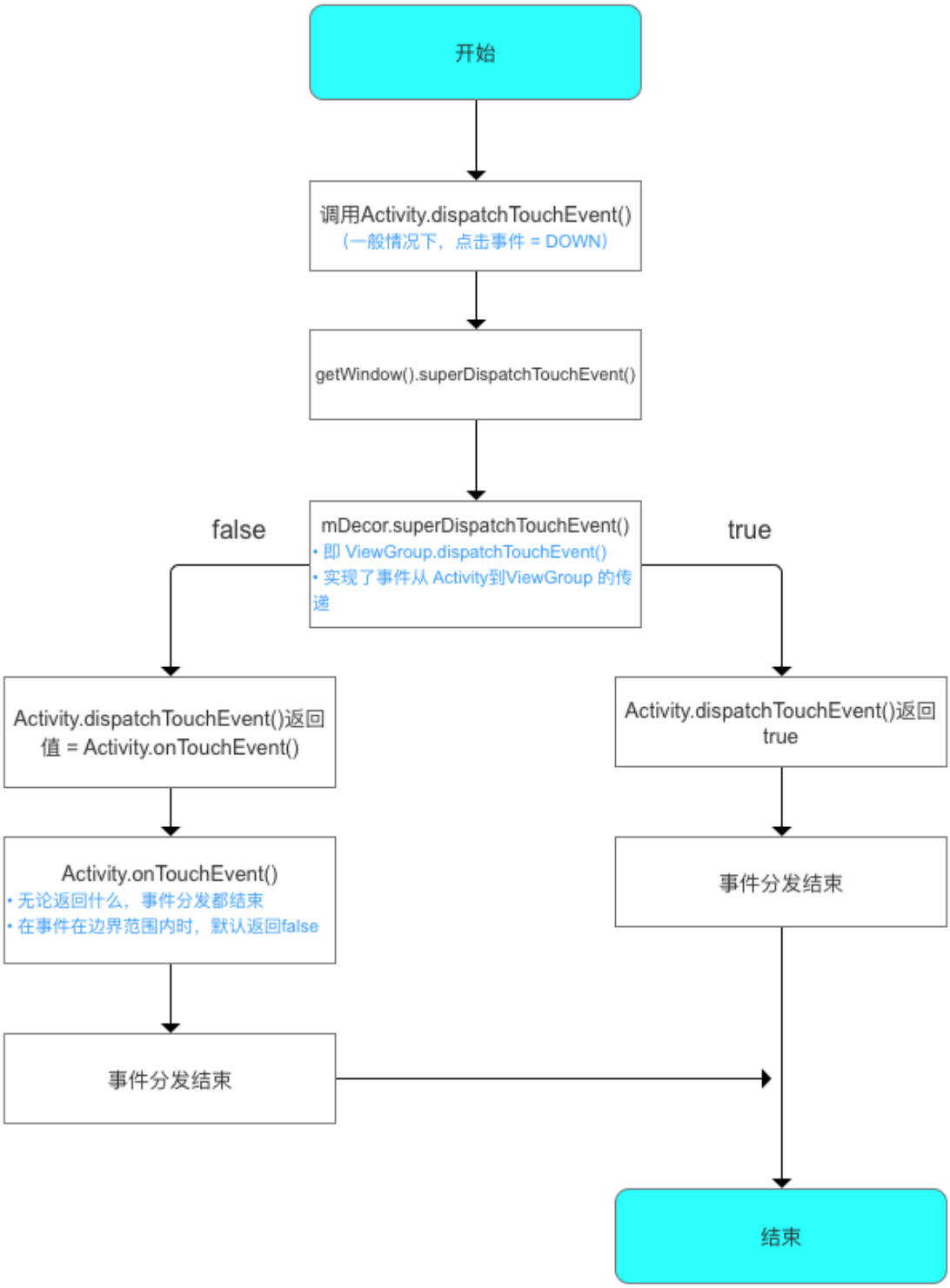
View 事件分发机制

Android 的事件分发机制基本会遵从 Activity -> ViewGroup -> View 的顺序进行事件分发，然后通过调用 onTouchEvent() 方法进行事件的处理。

一般情况下，事件列都是从用户按下（ACTION_DOWN）的那一刻产生的，不得不提到，三个非常重要的于事件相关的方法。

- dispatchTouchEvent() - 分发事件
- onTouchEvent() - 处理事件
- onInterceptTouchEvent() - 拦截事件

Activity 的事件分发



不管是 DOWN、MOVE 还是 UP 都是按照下面的顺序执行：

1. dispatchTouchEvent
2. setOnTouchListener 的 onTouch
3. onTouchEvent

onTouch 方法里能做的事情比 onClick 要多一些，比如判断手指按下、抬起、移动等事件。

那么如果两个事件都注册了，onTouch 是优先于 onClick 执行的，并且 onTouch 执行了两次，一次是 ACTION_DOWN，一次是 ACTION_UP。因此事件传递的顺序是先经过 onTouch，再传递给 onClick。

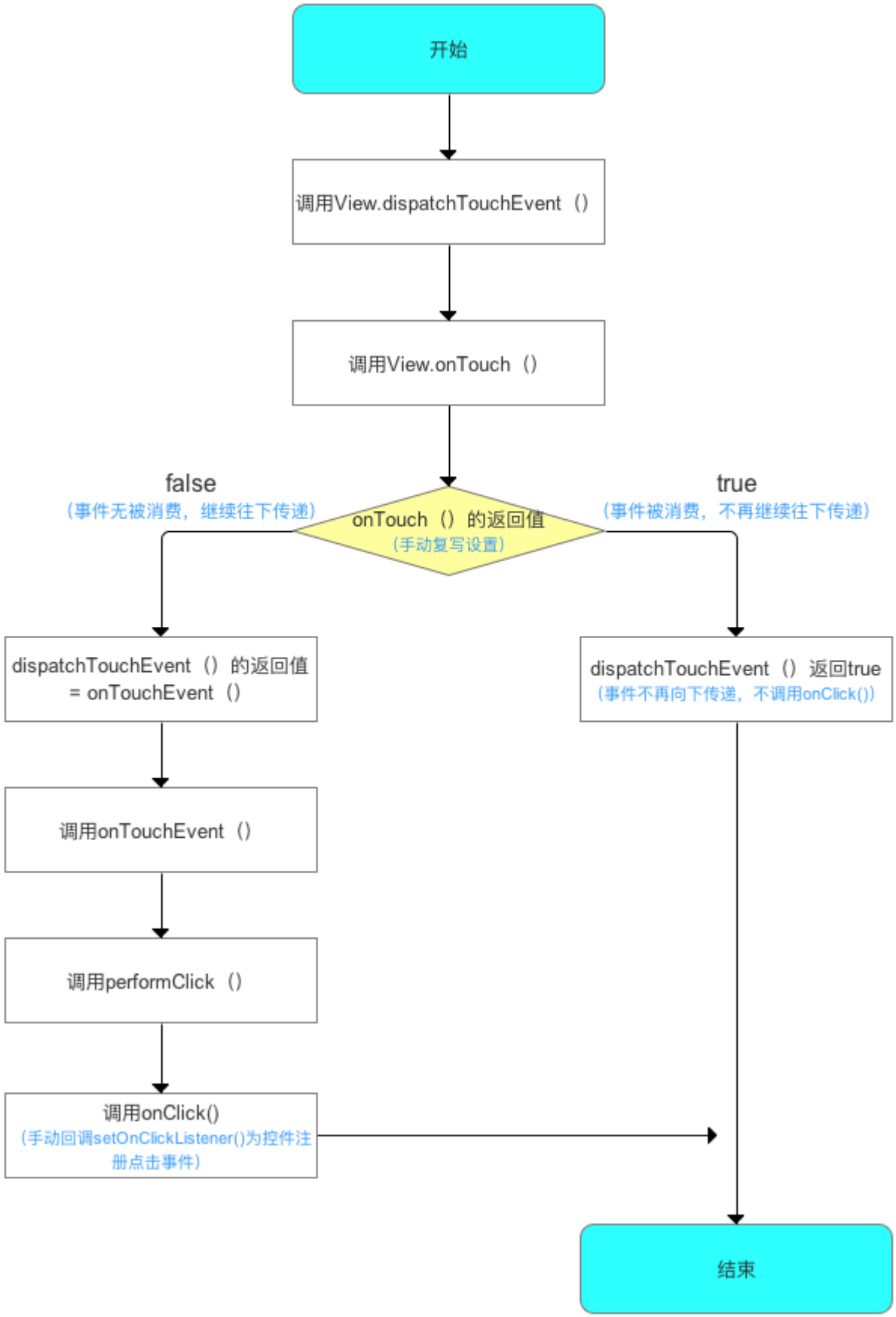
onTouch 方法是有返回值的，如果把 onTouch 方法里的返回值改成 true，onClick 方法不再执行了。

onTouch 和 onTouchEvent 有什么区别，又该如何使用?

这两个方法都是在 View 的 dispatchTouchEvent 中调用的，onTouch 优先于 onTouchEvent 执行。如果在 onTouch 方法中通过返回 true 将事件消费掉，onTouchEvent 将不会再执行。

另外需要注意的是，onTouch 能够得到执行需要两个前提条件，第一 mOnTouchListener 的值不能为空，第二当前点击的控件必须是 enable 的。因此如果有一个控件是非 enable 的，那么给它注册 onTouch 事件将永远得不到执行。对于这一类控件，如果想要监听它的 touch 事件，就必须通过在该控件中重写 onTouchEvent 方法来实现。

View的事件分发示意图：



整个 View 的事件转发流程是View.dispatchTouchEvent -> View.setOnTouchListener -> View.onTouchEvent

在 dispatchTouchEvent 中会进行 OnTouchListener 的判断，如果 onTouchEvent 不为 null 且返回 true，则表示事件被消费，onTouchEvent 不会被执行，否则执行 onTouchEvent。

onTouchEvent 中的 DOWN、MOVE、UP

1. DOWN

如果父控件支持滑动，首先设置标志为 PFLAG_PREPRESSED，设置 mHasPerformedLongPress = false，然后发出了一个 100ms 后的 mPendingCheckForTag。

如果 100ms 内没有触发 UP，则将标志置为 PFLAG_PRESSED，清除 PREPRESSED 标志，同时发出一个延时为 500-100 ms 的检查长按任务的消息。

如果父控件不支持滑动，则是将标记置为 PFLAG_PRESSED，同时发出一个延时为 500ms 的检查长按任务的消息。

检查长按任务的消息时间到了后，则会触发 LongClickListener。

此时如果 LongClickListener 不为 null，则会执行回调，但是如果 LongClickListener.onClick 返回 true，才把 mHasPerformedLongPress 设置为 true，否则 mHasPerformedLongPress 依然为 false。

2. MOVE

主要就是检查用户是否滑出了控件，如果触摸的位置已经不在当前 view 上了，则移除点击和长按的回调。

3. UP

如果 100ms 内，触发 UP，此时标志为 PFLAG_PREPRESSED ，则执行 UnSetPressedState，setPressed(false)，会把 setPress 转发下去，可以在 View 中复写 dispatchSetPressed 方法接收。

如果是 100ms - 500ms 之间，即长按还未发生，则首先移除长按检测，执行 onClick 回调；

如果是 500ms 以后，那么有两种情况：

- 设置了 onLongClickListener，且 onLongClickListener.onClick 返回 true，则点击事件 onClick 无法触发。
- 没有设置 onLongClickListener 或者 onLongClickListener.onClick 返回 false，则点击事件 onClick 事件触发。
- 最后执行 mUnSetPressedState.run()，将 setPressed 传递下去，然后将 PFLAG_PRESSED 标识清除。

ViewGroup 事件分发机制

dispatchTouchEvent 方法：

ACTION_DOWN 总结：ViewGroup 实现捕获 DOWN 事件，如果代码中不做 TOUCH 事件拦截，则判断当前子 View 是否在当前 x,y 的区域内，如果在，将其添加到 mFirstTouchTarget 链表的头部，并且调用子 View 的 dispatchTouchEvent() 方法把事件分发下去。

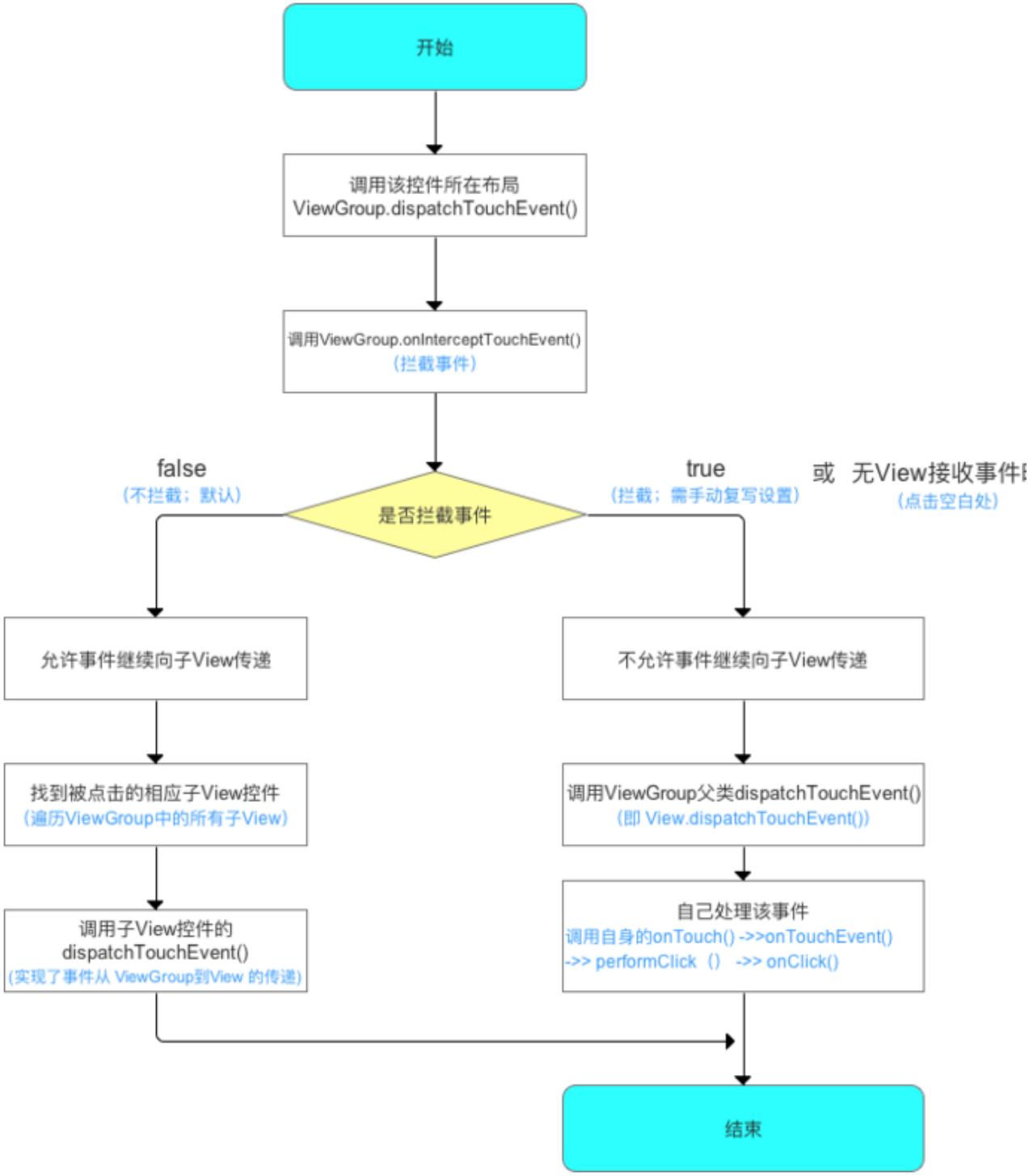
ACTION_MOVE 总结：ACTION_MOVE 在检测完是否拦截以后，直接调用了子 View 的 dispatchTouchEvent，事件分发下去。

ACTION_UP 总结：ACTION_UP 在检测完是否拦截以后，直接调用了子 View 的 dispatchTouchEvent，事件分发下去，最后重置触摸状态，将 mFirstTouchTarget 清空。

1. ViewGroup 实现捕获 DOWN 事件，如果代码中不做 TOUCH 事件拦截，则开始查找当前 x,y 是否在某个子 View 的区域内，如果在，将其添加到 mFirstTouchTarget 链表的头部，并且调用子 View 的 dispatchTouchEvent() 方法把事件分发下去。
2. ACTION_MOVE 中，ViewGroup 捕获到事件，然后判断是否拦截，如果没有拦截，则直接调用子 View 的 dispatchTouchEvent(ev) 将事件分发下去。
3. ACTION_UP 中，ViewGroup 捕获到事件，然后判断是否拦截，如果没有拦截，则直接调用子 View 的 dispatchTouchEvent(ev) 将事件分发小区，最后重置触摸状态，将 mFirstToychTarget 清空。

在分发之前都会修改一下坐标系统，把当前的 x,y 分别减去 child.left 和 child.top，然后传给 child。

ViewGroup事件分发示意图



1. Android 事件分发是先传递给 ViewGroup，再由 ViewGroup 传递给 View 的。
2. 在 ViewGroup 中可以通过 onInterceptTouchEvent() 方法对事件传递进行拦截，onInterceptTouchEvent() 方法返回 true 代表不允许事件继续向子 View 传递，返回 false 代表不对事件进行拦截，默认返回 false。
3. 子 View 中如果将传递的事件消费掉，ViewGroup 中将无法接收到任何事件。
4. 如果 ViewGroup 找到了能够处理该事件的 View，则直接交给子 View 处理，自己的 onTouchEvent() 不会被触发。
5. 可以通过复写 onInterceptTouchEvent(ev) 方法，拦截子 View 的事件（即 return true），把事件交给自己处理，则会执行自己对应的 onTouchEvent() 方法。
6. 子 View 可以通过调用 getParent().requestDisallowInterceptTouchEvent(true); 阻止 ViewGroup 对其 ACTION_MOVE 或者 ACTION_UP 事件进行拦截。

自定义 View 的实现方式

如果要按类型来划分的话，自定义 View 的实现方式大概可以分为三种，自绘控件、组合控件以及继承控件。

- 自绘组件

  自绘控件的意思就是，这个 View 上所展现的内容全部都是自己绘制出来的。

  绘制的代码是写在 onDraw() 方法中的。

- 组合控件

  组合控件的意思就是，并不需要自己去绘制视图上显示的内容，而只是用系统原生的控件就好了，但可以将几个系统原生的控件组合在一起，这样创建出的控件就被称为组合控件。

- 继承控件

  继承控件的意思就是并不需要自己从头去实现一个控件，只需要去继承一个现有的控件，然后在这个空间上增加一些新的功能，就可以形成一个自定义的控件了。

  这种自定义控件的特点就是不仅能够按照需求加入相应的功能，还可以保留原生控件的所有功能。

RecyclerView

RecyclerView 的 getLayoutPosition 和 getAdapterPosition

- getLayoutPosition 和 getAdapterPosition 通常情况下是一样的，只有当 Adapter 里面的内容改变了，而 Layout 还没来得及绘制的这段时间之内才有可能不一样，这个时间小于16ms
- 如果调用的是 notifyDataSetChanged()，因为要重新绘制所有 Item，所以在绘制完成之前 RecyclerView 是不知道 adapterPosition 的，这时会返回-1（NO_POSITION）
- 但如果用的是 notifyItemInserted(0)，那立即就能获取到正确的 adapterPosition，即使新的 Layout 还没绘制完成，比如之前是0的现在就会变成1，因为插入了0, 相当于 RecyclerView 提前帮你计算的，此时getLayoutPosition 还只能获取到旧的值。
- 总的来说，大多数情况下用 getAdapterPosition，只要不用 notifyDataSetChanged() 来刷新数据就总能立即获取到正确 position 值。

设计结构

ViewHolder

对于`Adapter`来说，一个`ViewHolder`就对应一个`data`。它也是`Recycler缓存池`的基本单元。

`ViewHolder`最重要的4个属性:

- itemView : 会被当做`child view`来`add`到`RecyclerView`中。
- mPosition : 标记当前的`ViewHolder`在`Adapter`中所处的位置。
- mItemViewType : 这个`ViewHolder`的`Type`，在`ViewHolder`保存到`RecyclerPool`时，主要靠这个类型来对`ViewHolder`做复用。
- mFlags : 标记`ViewHolder`的状态，比如`FLAG_BOUND(显示在屏幕上)`、`FLAG_INVALID(无效，想要使用必须rebound)`、`FLAG_REMOVED(已被移除)`等。

Adapter

它的工作是把`data`和`View`绑定，即上面说的一个`data`对应一个`ViewHolder`。主要负责`ViewHolder`的创建以及数据变化时通知`RecycledView`。

AdapterDataObservable

`Adapter`是数据源的直接接触者，当数据源发生变化时，它需要通知给`RecyclerView`。这里使用的模式是`观察者模式`。`AdapterDataObservable`是数据源变化时的被观察者。`RecyclerViewDataObserver`是观察者。
在开发中我们通常使用`adapter.notifyXX()`来刷新UI，实际上`Adapter`会调用`AdapterDataObservable`的`notifyChanged()`。

```
    public void notifyChanged() {
        for (int i = mObservers.size() - 1; i >= 0; i--) {
            mObservers.get(i).onChanged();
        }
    }
```

RecyclerViewDataObserver

它是`RecycledView`用来监听`Adapter`数据变化的观察者。

```
    public void onChanged() {
        mState.mStructureChanged = true; // RecycledView每一次UI的更新都会有一个
State
        processDataSetCompletelyChanged(true);
        if (!mAdapterHelper.hasPendingUpdates()) {
            requestLayout();
        }
    }
```

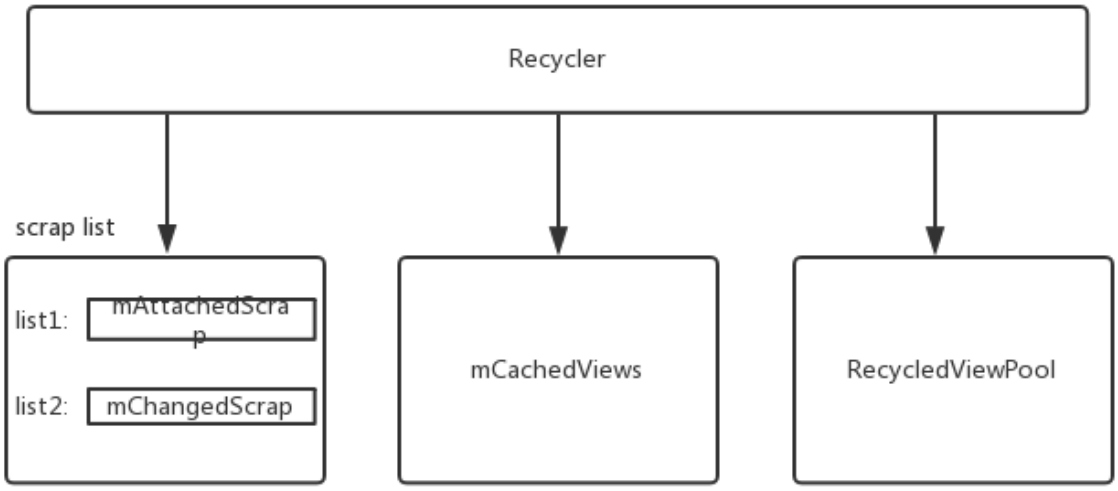
LayoutManager

它是`RecyclerView`的布局管理者，`RecyclerView`在`onLayout`时，会利用它来`layoutChildren`，它决定了`RecyclerView`中的子View的摆放规则。但不止如此， 它做的工作还有:

1. 测量子 View
2. 对子 View 进行布局
3. 对子 View 进行回收
4. 子 View 动画的调度
5. 负责`RecyclerView`滚动的实现

Recycler

对于`LayoutManager`来说，它是`ViewHolder`的提供者。对于`RecyclerView`来说，它是`ViewHolder`的管理者，是`RecyclerView`最核心的实现。下面这张图大致描述了它的组成:



- `mChangedScrap`：用来保存`RecyclerView`做动画时，被 detach 的`ViewHolder`。
- `mAttachedScrap`：用来保存`RecyclerView`做数据刷新(`notify`)，被 detach 的`ViewHolder`
- `mCacheViews`：`Recycler`的一级`ViewHolder`缓存。
- `RecyclerViewPool`：`mCacheViews`集合中装满时，会放到这里。

scrap list

```
final ArrayList<ViewHolder> mAttachedScrap = new ArrayList<>();
ArrayList<ViewHolder> mChangedScrap = null;
```

- view scrap 状态

  它是指`View`在`RecyclerView`布局期间进入分离状态的子视图。即它已经被`deatach`(标记为`FLAG_TMP_DETACHED`状态)了。这种`View`是可以被立即复用的。它在复用时，如果数据没有更新，是不需要调用`onBindViewHolder`方法的。如果数据更新了，那么需要重新调用`onBindViewHolder`。

  `mAttachedScrap`和`mChangedScrap`中的 View 复用主要作用在`adapter.notifyXXX`时。这时候就会产生很多`scrap`状态的`view`。 也可以把它理解为一个`ViewHolder`的缓存。不过在从这里获取`ViewHolder`时完全是根据`ViewHolder`的`position`而不是`item type`。如果在`notifyXX`时 data 已经被移除掉了，那么其中对应的`ViewHolder`也会被移除掉。

- mCacheViews

  可以把它理解为`RecyclerView`的一级缓存。它的默认大小是 3， 从中可以根据`item type`或者`position`来获取`ViewHolder`。可以通过`RecyclerView.setItemViewCacheSize()`来改变它的大小。

- RecyclerViewPool

  它是一个可以被复用的`ViewHolder`缓存池。即可以给多个`RecycledView`来设置统一个`RecycledViewPool`。这个对于`多tab feed流`应用可能会有很显著的效果。它内部利用一个`ScrapData`来保存`ViewHolder`集合：

```
class ScrapData {
    final ArrayList<ViewHolder> mScrapHeap = new ArrayList<>();
    int mMaxScrap = DEFAULT_MAX_SCRAP;   //最多缓存5个
    long mCreateRunningAverageNs = 0;
    long mBindRunningAverageNs = 0;
}

SparseArray<ScrapData> mScrap = new SparseArray<>();  //RecycledViewPool 用来保存ViewHolder的容器
```

一个`ScrapData`对应一种`type`的`ViewHolder`集合。看一下它的获取`ViewHolder`和保存`ViewHolder`的方法：

```
//存
public void putRecycledView(ViewHolder scrap) {
    final int viewType = scrap.getItemViewType();
    final ArrayList<ViewHolder> scrapHeap =
getScrapDataForType(viewType).mScrapHeap;
    if (mScrap.get(viewType).mMaxScrap <= scrapHeap.size())  return; //到最大极限就不能放了
    scrap.resetInternal();  //放到里面，这个view就相当于和原来的信息完全隔离了，只记得他的type，清除其相关状态
    scrapHeap.add(scrap);
}

//取
private ScrapData getScrapDataForType(int viewType) {
    ScrapData scrapData = mScrap.get(viewType);
    if (scrapData == null) {
        scrapData = new ScrapData();
        mScrap.put(viewType, scrapData);
    }
    return scrapData;
}
```

RecyclerView 刷新机制

adapter.notifyDataSetChanged() 引起的刷新

Adapter.notifyDataSetChanged() 方法会引起 RecyclerView 重新布局（requestLayout），因此从 onLayout() 方法开始。

RecyclerView.onLayout

onLayout() 方法调用了 diapatchLayout() 方法：

```
    void dispatchLayout() {
        if (mAdapter == null) {
```

```
            Log.e(TAG, "No adapter attached; skipping layout");
            // leave the state in START
            return;
        }
        if (mLayout == null) {
            Log.e(TAG, "No layout manager attached; skipping layout");
            // leave the state in START
            return;
        }
        mState.mIsMeasuring = false;
        if (mState.mLayoutStep == State.STEP_START) {
            dispatchLayoutStep1();
            mLayout.setExactMeasureSpecsFrom(this);
            dispatchLayoutStep2();
        } else if (mAdapterHelper.hasUpdates() || mLayout.getWidth() !=
getWidth()
                || mLayout.getHeight() != getHeight()) { // 数据变化 || 布局变化
            // First 2 steps are done in onMeasure but looks like we have to
run again due to
            // changed size.
            mLayout.setExactMeasureSpecsFrom(this);
            dispatchLayoutStep2();
        } else {
            // always make sure we sync them (to ensure mode is exact)
            mLayout.setExactMeasureSpecsFrom(this);
        }
        dispatchLayoutStep3();
    }
```

可以看到整个布局过程总共分为 3 步：

- STEP_START -> dispatchLayoutStep1
- STEP_LAYOUT -> dispatchLayoutStep2
- STEP_ANIMATIONS -> dispatchLayoutStep2()、dispatchLayoutStep3()

第一步`STEP_START`主要是来存储当前`子View`的状态并确定是否要执行动画。 而第 3 步`STEP_ANIMATIONS`是来执行动画的。

dispatchLayoutStep2()

```
    /**
     * The second layout step where we do the actual layout of the views for
the final state.
     * This step might be run multiple times if necessary (e.g. measure).
     */
    private void dispatchLayoutStep2() {
        eatRequestLayout();
        onEnterLayoutOrScroll();
```

```
        mState.assertLayoutStep(State.STEP_LAYOUT | State.STEP_ANIMATIONS); //
方法执行期不能重入
        mAdapterHelper.consumeUpdatesInOnePass();
        // 设置到初始状态
        mState.mItemCount = mAdapter.getItemCount();
        mState.mDeletedInvisibleItemCountSincePreviousLayout = 0;

        // Step 2: Run layout
        mState.mInPreLayout = false;
        // 调用布局管理器去布局
        mLayout.onLayoutChildren(mRecycler, mState);

        mState.mStructureChanged = false;
        mPendingSavedState = null;

        // onLayoutChildren may have caused client code to disable item
animations; re-check
        mState.mRunSimpleAnimations = mState.mRunSimpleAnimations &&
mItemAnimator != null;
        mState.mLayoutStep = State.STEP_ANIMATIONS; // 接下来执行布局的第三步
        onExitLayoutOrScroll();
        resumeRequestLayout(false);
    }
```

`mState`是一个`RecyclerView.State`对象。它是用来保存`RecyclerView`状态的一个对象，主要是用在`LayoutManager、Adapter等`组件之间共享`RecyclerView状态`的。可以看到这个方法将布局的工作交给了`mLayout`。这里它的实例是`LinearLayoutManager`，因此接下来看一下`LinearLayoutManager.onLayoutChildren()`。

LinearLayoutManager.onLayoutChildren()

布局逻辑还是很简单的:

1. 确定锚点`(Anchor)View`, 设置好`AnchorInfo`
2. 根据`锚点View`确定有多少布局空间`mLayoutState.mAvailable`可用
3. 根据当前设置的`LinearLayoutManager`的方向开始摆放子View

- 确定锚点 View

  `锚点View`大部分是通过`updateAnchorFromChildren`方法确定的，这个方法主要是获取一个View，把它的信息设置到`AnchorInfo`中：

  ```
  mAnchorInfo.mLayoutFromEnd = mShouldReverseLayout    // 即和你是否在 manifest
  中设置了布局 rtl 有关

  private boolean updateAnchorFromChildren(RecyclerView.Recycler recycler,
  RecyclerView.State state, AnchorInfo anchorInfo) {
      ...
      View referenceChild = anchorInfo.mLayoutFromEnd
  ```

```
                ? findReferenceChildClosestToEnd(recycler, state) //如果是从
end(尾部)位置开始布局，那就找最接近end的那个位置的View作为锚点View
                : findReferenceChildClosestToStart(recycler, state); //如果是从
start(头部)位置开始布局，那就找最接近start的那个位置的View作为锚点View

    if (referenceChild != null) {
        anchorInfo.assignFromView(referenceChild,
getPosition(referenceChild));
        ...
        return true;
    }
    return false;
}
```

如果是`start to end`, 那么就找最接近 start (RecyclerView 头部)的 View 作为布局的锚点 View。如果是`end to start (rtl)`, 就找最接近 end 的 View 作为布局的锚点。

`AnchorInfo`最重要的两个属性是`mCoordinate`和`mPosition`，找到锚点 View 后就会通过`anchorInfo.assignFromView()`方法来设置这两个属性:

```
public void assignFromView(View child, int position) {
    if (mLayoutFromEnd) {
        mCoordinate = mOrientationHelper.getDecoratedEnd(child) +
mOrientationHelper.getTotalSpaceChange();
    } else {
        mCoordinate = mOrientationHelper.getDecoratedStart(child);
    }
    mPosition = position;
}
```

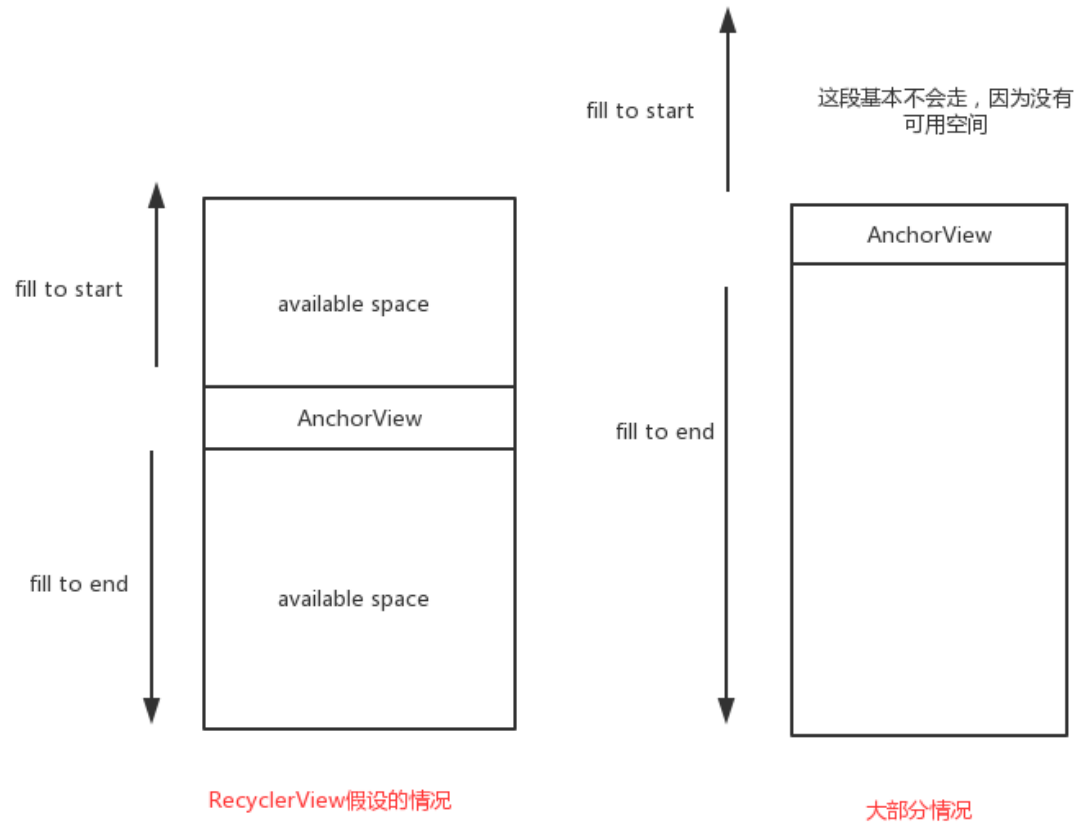
  - `mCoordinate`其实就是`锚点View`的`Y(X)`坐标去掉`RecyclerView`的padding。
  - `mPosition`其实就是`锚点View`的位置。

- 确定有多少布局空间可用并拜访子 view

  当确定好`AnchorInfo`后，需要根据`AnchorInfo`来确定`RecyclerView`当前可用于布局的空间，然后来摆放子View。以布局方向为`start to end (正常方向)`为例，这里的`锚点View`其实是`RecyclerView`最顶部的 View：

```
    // fill towards end  (1)
    updateLayoutStateToFillEnd(mAnchorInfo);      //确定AnchorView到
RecyclerView的底部的布局可用空间
    ...
    fill(recycler, mLayoutState, state, false); //填充view, 从 AnchorView 到
RecyclerView的底部
    endOffset = mLayoutState.mOffset;

    // fill towards start (2)
    updateLayoutStateToFillStart(mAnchorInfo); //确定AnchorView到
RecyclerView的顶部的布局可用空间
    ...
    fill(recycler, mLayoutState, state, false); //填充view,从 AnchorView 到
RecyclerView的顶部
```

上面标注了`(1)和(2)`，1 次布局是由这两部分组成的，具体如下图所示：



- fill towards end
  - 确定可用布局空间

    在`fill`之前，需要先确定`从锚点View`到`RecyclerView底部`有多少可用空间。是通过`updateLayoutStateToFillEnd`方法：

```
updateLayoutStateToFillEnd(anchorInfo.mPosition,
anchorInfo.mCoordinate);

void updateLayoutStateToFillEnd(int itemPosition, int offset) {
    mLayoutState.mAvailable = mOrientationHelper.getEndAfterPadding() -
offset;
    ...
    mLayoutState.mCurrentPosition = itemPosition;
    mLayoutState.mLayoutDirection = LayoutState.LAYOUT_END;
    mLayoutState.mOffset = offset;
    mLayoutState.mScrollingOffset = LayoutState.SCROLLING_OFFSET_NaN;
}
```

`mLayoutState`是`LinearLayoutManager`用来保存布局状态的一个对象。`mLayoutState.mAvailable`就是用来表示`有多少空间可用来布局`。`mOrientationHelper.getEndAfterPadding() - offset`其实大致可以理解为`RecyclerView`的高度。**所以这里可用布局空间`mLayoutState.mAvailable`就是RecyclerView的高度**

- 摆放子 View

  接下来继续看`LinearLayoutManager.fill()`方法，这个方法是布局的核心方法，是用来向`RecyclerView`中添加子View的方法：

```
int fill(RecyclerView.Recycler recycler, LayoutState layoutState,
RecyclerView.State state, boolean stopOnFocusable) {
    final int start = layoutState.mAvailable;  //前面分析，其实就是
RecyclerView的高度
    ...
    int remainingSpace = layoutState.mAvailable + layoutState.mExtra;
 //extra 是你设置的额外布局的范围，这个一般不推荐设置
    LayoutChunkResult layoutChunkResult = mLayoutChunkResult; //保存布局
一个child view后的结果
    while ((layoutState.mInfinite || remainingSpace > 0) &&
layoutState.hasMore(state)) { //有剩余空间的话，就一直添加 childView
        layoutChunkResult.resetInternal();
        ...
        layoutChunk(recycler, state, layoutState, layoutChunkResult);
//布局子View的核心方法
        ...
        layoutState.mOffset += layoutChunkResult.mConsumed *
layoutState.mLayoutDirection; // 一次 layoutChunk 消耗了多少空间
        ...
        // 子 View 的回收工作
    }
    ...
}
```

这个方法的核心是调用`layoutChunk()`来不断消耗`layoutState.mAvailable`，直到消耗完毕。继续看一下`layoutChunk()方法`，这个方法的主要逻辑是:

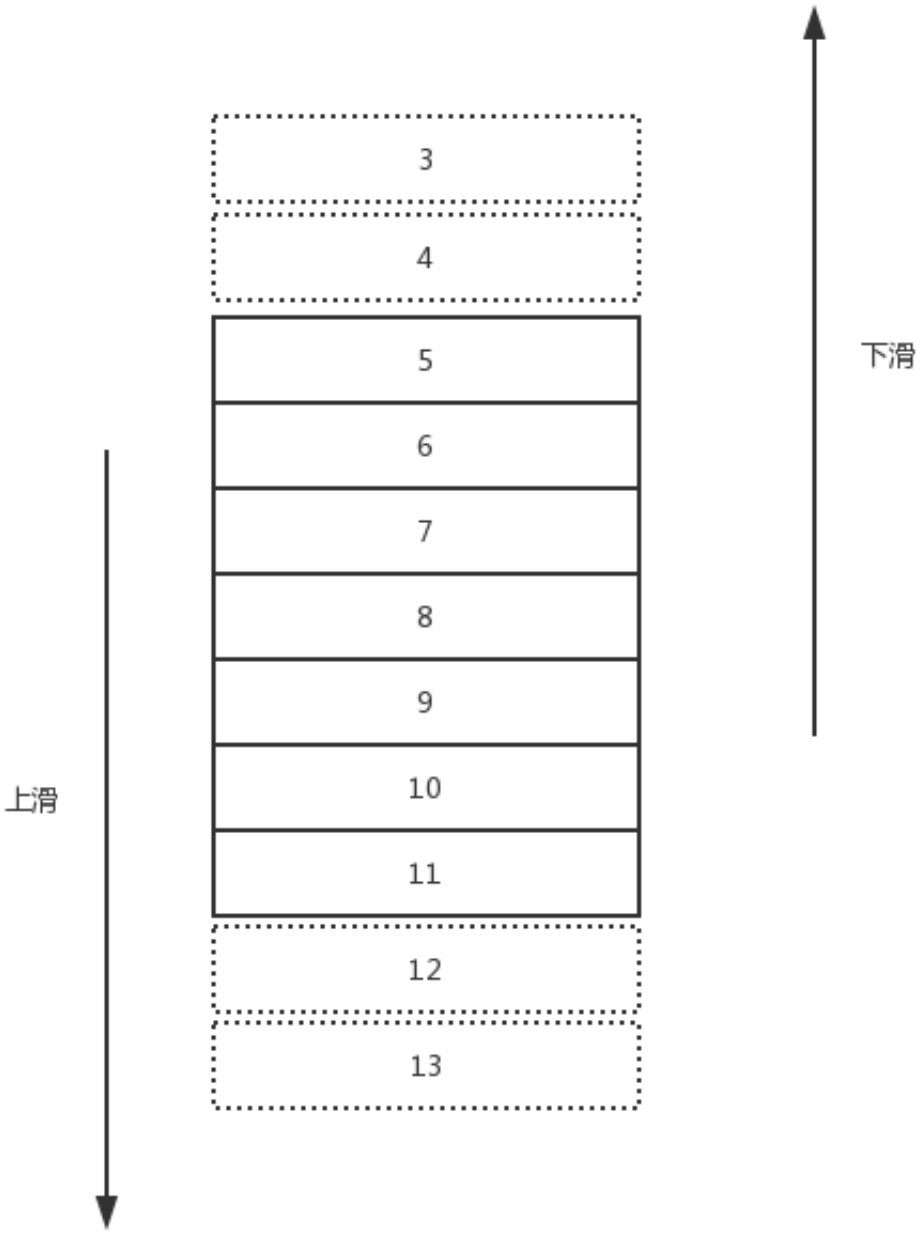
1. 从`Recycler`中获取一个`View`
2. 添加到`RecyclerView`中
3. 调整`View`的布局参数，调用其`measure、layout`方法。

```
void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State
state,LayoutState layoutState, LayoutChunkResult result) {
        View view = layoutState.next(recycler);  //这个方法会向 recycler
view 要一个 holder
        ...
        if (mShouldReverseLayout == (layoutState.mLayoutDirection ==
LayoutState.LAYOUT_START)) { //根据布局方向，添加到不同的位置
            addView(view);
        } else {
            addView(view, 0);
        }
        measureChildWithMargins(view, 0, 0);     //调用view的measure

        ...// measure 后确定布局参数 left/top/right/bottom

        layoutDecoratedWithMargins(view, left, top, right, bottom); //
调用 view 的 layout
        ...
    }
```

到这里其实就完成了上面的`fill towards end`。`fill towards start`就是从`锚点View`向`RecyclerView顶部`来摆放子View，具体逻辑类似`fill towards end`。

RecyclerView 滑动时的刷新逻辑

`RecyclerView`在滑动时是如何展示`子View`的，即下面这种状态：



`RecyclerView`在`OnTouchEvent`对滑动事件做了监听，然后派发到`scrollStep()`方法：

```
void scrollStep(int dx, int dy, @Nullable int[] consumed) {
    startInterceptRequestLayout(); //处理滑动时不能重入
    ...
    if (dx != 0) {
        consumedX = mLayout.scrollHorizontallyBy(dx, mRecycler, mState);
    }
    if (dy != 0) {
        consumedY = mLayout.scrollVerticallyBy(dy, mRecycler, mState);
    }
    ...
    stopInterceptRequestLayout(false);
```

```
    if (consumed != null) { //记录消耗
        consumed[0] = consumedX;
        consumed[1] = consumedY;
    }
}
```

即把滑动的处理交给了`mLayout`，继续看`LinearLayoutManager.scrollVerticallyBy`，它直接调用了`scrollBy()`，这个方法就是`LinearLayoutManager`处理滚动的核心方法。

LinearLayoutManager.scrollBy

```
int scrollBy(int dy, RecyclerView.Recycler recycler, RecyclerView.State state)
{
    ...
    final int layoutDirection = dy > 0 ? LayoutState.LAYOUT_END :
LayoutState.LAYOUT_START;
    final int absDy = Math.abs(dy);
    updateLayoutState(layoutDirection, absDy, true, state); //确定可用布局空间
    final int consumed = mLayoutState.mScrollingOffset + fill(recycler,
mLayoutState, state, false); //摆放子View
    ....
    final int scrolled = absDy > consumed ? layoutDirection * consumed : dy;
    mOrientationHelper.offsetChildren(-scrolled); // 滚动 RecyclerView
    ...
}
```

这个方法的主要执行逻辑是:

1. 根据布局方向和滑动的距离来确定可用布局空间`mLayoutState.mAvailable`
2. 调用`fill()`来摆放子 View
3. 滚动 RecyclerView

根据布局方向和滑动的距离来确定可用布局空间

以向下滚动为为例，看一下`updateLayoutState`方法:

```
// requiredSpace是滑动的距离;  canUseExistingSpace是true
void updateLayoutState(int layoutDirection, int requiredSpace,boolean
canUseExistingSpace, RecyclerView.State state) {

    if (layoutDirection == LayoutState.LAYOUT_END) { //滚动方法为向下
        final View child = getChildClosestToEnd(); //获得RecyclerView底部的View
        ...
        mLayoutState.mCurrentPosition = getPosition(child) +
mLayoutState.mItemDirection; //view的位置
        mLayoutState.mOffset = mOrientationHelper.getDecoratedEnd(child);
//view的偏移 offset
        scrollingOffset = mOrientationHelper.getDecoratedEnd(child) -
mOrientationHelper.getEndAfterPadding();
```

```
    } else {
        ...
    }

    mLayoutState.mAvailable = requiredSpace;
    if (canUseExistingSpace)  mLayoutState.mAvailable -= scrollingOffset;
    mLayoutState.mScrollingOffset = scrollingOffset;
}
```

所以可用的布局空间就是滑动的距离。mLayoutState.mScrollingOffset = `（childView 的 bottom + childView 的 margin） - RecyclerView 的 Padding`。

滚动 RecyclerView

对于`RecyclerView`的滚动，最终调用到了`RecyclerView.offsetChildrenVertical()`：

```
//dy 这里就是滚动的距离
public void offsetChildrenVertical(@Px int dy) {
    final int childCount = mChildHelper.getChildCount();
    for (int i = 0; i < childCount; i++) {
        mChildHelper.getChildAt(i).offsetTopAndBottom(dy);
    }
}
```

可以看到逻辑很简单，就是**改变当前子View布局的top和bottom**来达到滚动的效果。

RecyclerView 复用机制

从 Recycler 中获取一个 ViewHolder 的逻辑

`LayoutManager`会调用`Recycler.getViewForPosition(pos)`来获取一个指定位置（这个位置是子View 布局所在的位置）的`view`。`getViewForPosition()`会调用`tryGetViewHolderForPositionByDeadline(position...)`，这个方法是从`Recycler`中获取一个`View`的核心方法。它就是`如何从Recycler中获取一个ViewHolder`的逻辑，即`怎么取`。

```
ViewHolder tryGetViewHolderForPositionByDeadline(int position, boolean dryRun,
long deadlineNs) {
    ...
    if (mState.isPreLayout()) {     //动画相关
        holder = getChangedScrapViewForPosition(position);  //从缓存中拿吗？不应该
不是缓存?
        fromScrapOrHiddenOrCache = holder != null;
    }
    // 1) Find by position from scrap/hidden list/cache
    if (holder == null) {
        holder = getScrapOrHiddenOrCachedHolderForPosition(position, dryRun);
//从 attach 和 mCacheViews 中获取
        if (holder != null) {
            ... //校验这个 holder 是否可用
```

```
        }
    }
    if (holder == null) {
        ...
        final int type = mAdapter.getItemViewType(offsetPosition); // 获取这个位置的数据的类型。  子 Adapter 复写的方法
        // 2) Find from scrap/cache via stable ids, if exists
        if (mAdapter.hasStableIds()) {    //stable id 就是标识一个 viewholder 的唯一性，  即使它做动画改变了位置
            holder =
getScrapOrCachedViewForId(mAdapter.getItemId(offsetPosition),  // 根据 stable id 从 scrap 和  mCacheViews 中获取
                    type, dryRun);
            ....
        }
        if (holder == null && mViewCacheExtension != null) { // 从用户自定义的缓存集合中获取
            final View view = mViewCacheExtension
                    .getViewForPositionAndType(this, position, type);  //你返回的 View 要是 RecyclerView.LayoutParams  属性的
            if (view != null) {
                holder = getChildViewHolder(view);  //把它包装成一个ViewHolder
                ...
            }
        }
        if (holder == null) { // 从 RecyclerViewPool中获取
            holder = getRecycledViewPool().getRecycledView(type);
            ...
        }
        if (holder == null) {
            ...
            //实在没有就会创建
            holder = mAdapter.createViewHolder(RecyclerView.this, type);
            ...
        }
    }
    ...
    boolean bound = false;
    if (mState.isPreLayout() && holder.isBound()) { //动画时不会想去调用 onBindData
        ...
    } else if (!holder.isBound() || holder.needsUpdate() || holder.isInvalid()) {
        ...
        final int offsetPosition = mAdapterHelper.findPositionOffset(position);
        bound = tryBindViewHolderByDeadline(holder, offsetPosition, position, deadlineNs);  //调用 bindData 方法
    }
```

```
    final ViewGroup.LayoutParams lp = holder.itemView.getLayoutParams();
    final LayoutParams rvLayoutParams;
    ...//调整 LayoutParams
    return holder;
}
```

即大致步骤是:

1. 如果执行了`RecyclerView`动画的话，尝试`根据position`从`mChangedScrap集合`中寻找一个`ViewHolder`
2. 尝试`根据position`从`scrap集合`、`hide的view集合`、`mCacheViews(一级缓存)`中寻找一个`ViewHolder`
3. 根据`LayoutManager`的`position`更新到对应的`Adapter`的`position`。(这两个`position`在大部分情况下都是相等的，不过在`子view删除或移动`时可能产生不对应的情况)
4. 根据`Adapter position`，调用`Adapter.getItemViewType()`来获取`ViewType`
5. 根据`stable id(用来表示ViewHolder的唯一，即使位置变化了)`从`scrap集合`和`mCacheViews(一级缓存)`中寻找一个`ViewHolder`
6. 根据`position和viewType`尝试从用户自定义的`mViewCacheExtension`中获取一个`ViewHolder`
7. 根据`ViewType`尝试从`RecyclerViewPool`中获取一个`ViewHolder`
8. 调用`mAdapter.createViewHolder()`来创建一个`ViewHolder`
9. 如果需要的话调用`mAdapter.bindViewHolder`来设置`ViewHolder`。
10. 调整`ViewHolder.itemview`的布局参数为`Recycler.LayoutPrams`，并返回 Holder。

逻辑还是很简单的，即从几个缓存集合中获取`ViewHolder`，如果实在没有就创建。

情形一：由无到有

即一开始`RecyclerView`中没有任何数据，添加数据源后`adapter.notifyXXX`。

很明显在这种情形下`Recycler`中是不会存在任何可复用的`ViewHolder`。所以所有的`ViewHolder`都是新创建的。即会调用`Adapter.createViewHolder()`和`Adapter.bindViewHolder()`。

这时候新创建的这些`ViewHolder`是不会被缓存起来的。 即在这种情形下: **Recycler只会通过Adapter创建ViewHolder，并且不会缓存这些新创建的ViewHolder。**

情形二：在原有数据的情况下进行整体刷新

其实就是相当于用户在 feed 中做了下拉刷新。实现中的伪代码如下:

```
dataSource.clear()
dataSource.addAll(newList)
adapter.notifyDatasetChanged()
```

`Recycler`肯定复用了老的卡片(卡片的类型不变)，那么问题是 : 在用户刷新时`旧ViewHolder`保存在哪里？ 如何调用`旧ViewHolder`的`Adapter.bindViewHolder()`来重新设置数据的?

`LinearLayoutManager`在确定好`布局锚点View`之后就会把当前`attach`在`RecyclerView`上的`子View`全部设置为`scrap状态`:

```
void onLayoutChildren(RecyclerView.Recycler recycler, RecyclerView.State state)
{
    ...
    onAnchorReady(recycler, state, mAnchorInfo, firstLayoutDirection);  //
RecyclerView 指定锚点，要准备正式布局了
    detachAndScrapAttachedViews(recycler);    // 在开始布局时，把所有的 View 都设置为
scrap 状态
    ...
}
```

什么是 scrap 状态呢？ ViewHolder 被标记为`FLAG_TMP_DETACHED`状态，并且其`itemview`的`parent`被设置为`null`。

`detachAndScrapAttachedViews`就是把所有的 view 保存到`Recycler`的`mAttachedScrap`集合中：

```
public void detachAndScrapAttachedViews(@NonNull Recycler recycler) {
    for (int i = getChildCount() - 1; i >= 0; i--) {
        final View v = getChildAt(i);
        scrapOrRecycleView(recycler, i, v);
    }
}
private void scrapOrRecycleView(Recycler recycler, int index, View view) {
    final ViewHolder viewHolder = getChildViewHolderInt(view);
    ...//删去了一些判断逻辑
    detachViewAt(index);  //设置 RecyclerView 这个位置的 view 的 parent 为 null,
并标记 ViewHolder 为 FLAG_TMP_DETACHED
    recycler.scrapView(view); //添加到 mAttachedScrap 集合中
    ...
}
```

所以在这种情形下`LinearLayoutManager`在真正摆放`子View`之前，会把所有`旧的子View`按顺序保存到`Recycler`的`mAttachedScrap集合`中

`LinearLayoutManager`在布局时如何复用`mAttachedScrap集合`中的`ViewHolder`。

前面已经说了`LinearLayoutManager`会根据当前布局子View的位置向`Recycler`要一个子View，即调用到`tryGetViewHolderForPositionByDeadline(position..)`。上面已经列出了这个方法的逻辑，其实在前面的第二步：

尝试`根据position`从`scrap集合`、`hide的view集合`、`mCacheViews(一级缓存)`中寻找一个`ViewHolder`

即从`mAttachedScrap`中就可以获得一个`ViewHolder`：

```
ViewHolder getScrapOrHiddenOrCachedHolderForPosition(int position, boolean
dryRun) {
    final int scrapCount = mAttachedScrap.size();
    for (int i = 0; i < scrapCount; i++) {
        final ViewHolder holder = mAttachedScrap.get(i);
        if (!holder.wasReturnedFromScrap() && holder.getLayoutPosition() ==
position
                && !holder.isInvalid() && (mState.mInPreLayout ||
!holder.isRemoved())) {
            holder.addFlags(ViewHolder.FLAG_RETURNED_FROM_SCRAP);
            return holder;
        }
    }
    ...
}
```

即如果`mAttachedScrap`中`holder`的位置和`入参position`相等，并且`holder`是有效的话这个`holder`就是可以复用的。所以综上所述，在情形二下所有的`ViewHolder`几乎都是复用`Recycler`中`mAttachedScrap`集合中的。
并且重新布局完毕后`Recycler`中是不存在可复用的`ViewHolder`的。

情形三：滚动复用

这个情形分析是在`情形二`的基础上向下滑动时`ViewHolder`的复用情况以及`Recycler`中`ViewHolder`的保存情况。

在这种情况下滚出屏幕的 View 会优先保存到 `mCacheViews`，如果`mCacheViews`中保存满了，就会保存到`RecyclerViewPool`中。

在`RecyclerView刷新机制`中分析过，`RecyclerView`在滑动时会调用`LinearLayoutManager.fill()`方法来根据滚动的距离来向`RecyclerView`填充子 View，其实在个方法在填充完子 View 之后就会把滚动出屏幕的 View 做回收：

```
int fill(RecyclerView.Recycler recycler, LayoutState
layoutState,RecyclerView.State state, boolean stopOnFocusable) {
    ...
    int remainingSpace = layoutState.mAvailable + layoutState.mExtra;
    ...
    while ((layoutState.mInfinite || remainingSpace > 0) &&
layoutState.hasMore(state)) {
        ...
        layoutChunk(recycler, state, layoutState, layoutChunkResult); //填充一个
子 View

        if (layoutState.mScrollingOffset != LayoutState.SCROLLING_OFFSET_NaN) {
            layoutState.mScrollingOffset += layoutChunkResult.mConsumed;
            if (layoutState.mAvailable < 0) {
                layoutState.mScrollingOffset += layoutState.mAvailable;
            }
```

```
                recycleByLayoutState(recycler, layoutState); //根据滚动的距离来回收
View
            }
        }
    }
```

即`fill`每填充一个`子View`都会调用`recycleByLayoutState()`来回收一个`旧的子View`，这个方法在层层调用之后会调用到`Recycler.recycleViewHolderInternal()`。这个方法是`ViewHolder`回收的核心方法，不过逻辑很简单:

1. 检查`mCacheViews集合`中是否还有空位，如果有空位，则直接放到`mCacheViews集合`。
2. 如果没有的话就把`mCacheViews集合`中最前面的`ViewHolder`拿出来放到`RecyclerViewPool`中，然后再把最新的这个ViewHolder放到`mCacheViews集合`。
3. 如果没有成功缓存到`mCacheViews集合`中，就直接放到`RecyclerViewPool`。

`mCacheViews集合`为什么要这样缓存?

往上滑动一段距离，被滑动出去的`ViewHolder`会被缓存在`mCacheViews集合`，并且位置是被记录的。如果用户此时再下滑的话，从Recycler`中获取 ViewHolder 的逻辑:

1. 先按照位置从`mCacheViews集合`中获取
2. 按照`viewType`从`mCacheViews集合`中获取

上面对于`mCacheViews集合`两步操作，其实第一步就已经命中了缓存的`ViewHolder`。并且这时候都不需要调用`Adapter.bindViewHolder()`方法的。即是十分高效的。

所以在普通的滚动复用的情况下，`ViewHolder`的复用主要来自于`mCacheViews集合`，旧的`ViewHolder`会被放到`mCacheViews集合`， `mCacheViews集合`挤出来的更老的`ViewHolder`放到了`RecyclerViewPool`中。

RecyclerView 动画源码浅析

可以通过下面这两个方法触发`RecyclerView`的删除动画:

```
    //一个item的删除动画
    dataSource.removeAt(1)
    recyclerView.adapter.notifyItemRemoved(1)

    //多个item的删除动画
    dataSource.removeAt(1)
    dataSource.removeAt(1)
    recyclerView.adapter.notifyItemRangeRemoved(1,2)
```

`adapter.notifyItemRemoved(1)`会回调到`RecyclerViewDataObserver`的 onItemRangeRemoved 方法:

```
    public void onItemRangeRemoved(int positionStart, int itemCount) {
        if (mAdapterHelper.onItemRangeRemoved(positionStart, itemCount)) {
            triggerUpdateProcessor();
        }
    }
```

其实按照`onItemRangeRemoved()`这个的执行逻辑方法可以将`Item删除动画`分为两个部分:

1. 添加一个`UpdateOp`到`AdapterHelper.mPendingUpdates`中。
2. `triggerUpdateProcessor()`调用了`requestLayout`,即触发了`RecyclerView`的重新布局。

AdapterHelper

这个类可以理解为是用来记录`adapter.notifyXXX`动作的，即每一个`Operation(添加、删除)`都会在这个类中有一个对应记录`UpdateOp`，`RecyclerView`在布局时会检查这些`UpdateOp`，并做对应的操作。
`mAdapterHelper.onItemRangeRemoved`其实是添加一个`Remove UpdateOp`:

```
mPendingUpdates.add(obtainUpdateOp(UpdateOp.REMOVE, positionStart, itemCount,
null));
mExistingUpdateTypes |= UpdateOp.REMOVE;
```

即把一个`Remove UpdateOp`添加到了`mPendingUpdates`集合中。

RecyclerView.layout

`RecyclerView`的布局一共分为3分步骤:`dispatchLayoutStep1()`、`dispatchLayoutStep2()`、`dispatchLayoutStep3()`。

dispatchLayoutStep1(保存动画现场)

直接从`dispatchLayoutStep1()`开始看，这个方法是`RecyclerView`布局的第一步:

```
    private void dispatchLayoutStep1() {
        ...
        processAdapterUpdatesAndSetAnimationFlags();
        ...
        if (mState.mRunSimpleAnimations) {
            ...
        }
        ...
    }
```

`processAdapterUpdatesAndSetAnimationFlags()`最终其实是调用到`AdapterHelper.postponeAndUpdateViewHolders()`:

```
private void postponeAndUpdateViewHolders(UpdateOp op) {
    mPostponedList.add(op); //op 其实是从 mPendingUpdates 中取出来的
```

```
    switch (op.cmd) {
        case UpdateOp.ADD:
            mCallback.offsetPositionsForAdd(op.positionStart, op.itemCount);
break;
        case UpdateOp.MOVE:
            mCallback.offsetPositionsForMove(op.positionStart, op.itemCount);
break;
        case UpdateOp.REMOVE:

 mCallback.offsetPositionsForRemovingLaidOutOrNewView(op.positionStart,
op.itemCount); break;
        case UpdateOp.UPDATE:
            mCallback.markViewHoldersUpdated(op.positionStart, op.itemCount,
op.payload); break;
        ...
    }
}
```

即这个方法做的事情就是把`mPendingUpdates`中的`UpdateOp`添加到`mPostponedList`中，并回调根据`op.cmd`来回调`mCallback`，其实这个`mCallback`是回调到了`RecyclerView`中：

```
 void offsetPositionRecordsForRemove(int positionStart, int itemCount, boolean
applyToPreLayout) {
        final int positionEnd = positionStart + itemCount;
        final int childCount = mChildHelper.getUnfilteredChildCount();
        for (int i = 0; i < childCount; i++) {
            final ViewHolder holder =
getChildViewHolderInt(mChildHelper.getUnfilteredChildAt(i));
            ...
            if (holder.mPosition >= positionEnd) {
                holder.offsetPosition(-itemCount, applyToPreLayout);
                mState.mStructureChanged = true;
            }
            ...
        }
        ...
    }
```

`offsetPositionRecordsForRemove`方法：**主要是把当前显示在界面上的`ViewHolder`的位置做对应的改变，即如果item位于删除的item之后，那么它的位置应该减一**，比如原来的位置是 3 现在变成了 2。

接下来继续看`dispatchLayoutStep1()`中的操作：

```
    if (mState.mRunSimpleAnimations) {
        int count = mChildHelper.getChildCount();
        for (int i = 0; i < count; ++i) {
            final ViewHolder holder =
getChildViewHolderInt(mChildHelper.getChildAt(i));
            // 根据当前的显示在界面上的 ViewHolder 的布局信息创建一个 ItemHolderInfo
            final ItemHolderInfo animationInfo = mItemAnimator
                    .recordPreLayoutInformation(mState, holder,

 ItemAnimator.buildAdapterChangeFlagsForAnimations(holder),
                            holder.getUnmodifiedPayloads());
            mViewInfoStore.addToPreLayout(holder, animationInfo); // 把 holder
对应的 animationInfo 保存到  mViewInfoStore 中
            ...
        }
    }
```

即就做了两件事:

1. 为当前显示在界面上的每一个`ViewHolder`创建一个`ItemHolderInfo`，`ItemHolderInfo`其实就是保存了当前显示`itemview`的布局的`top、left`等信息。
2. 拿着`ViewHolder`和其对应的`ItemHolderInfo`调用`mViewInfoStore.addToPreLayout(holder, animationInfo)`。

`mViewInfoStore.addToPreLayout()`就是把这些信息保存起来:

```
void addToPreLayout(RecyclerView.ViewHolder holder,
RecyclerView.ItemAnimator.ItemHolderInfo info) {
    InfoRecord record = mLayoutHolderMap.get(holder);
    if (record == null) {
        record = InfoRecord.obtain();
        mLayoutHolderMap.put(holder, record);
    }
    record.preInfo = info;
    record.flags |= FLAG_PRE;
}
```

即把`holder 和 info`保存到`mLayoutHolderMap`中。可以理解为它是**用来保存动画执行前当前界面ViewHolder的信息**一个集合。

执行`Items删除动画`时`AdapterHelper`和`dispatchLayoutStep1()`的执行逻辑:

1. 创建一个 REMOVE OP
2. 当前的 ViewHolder 的位置做相应的变化
3. 保存当前显示的 ViewHolder 的信息

其实这些操作可以简单的理解为**保存动画前View的现场** 。其实这里有一次**预布局**，预布局也是为了保存动画前的 View 信息。

dispatchLayoutStep2

这一步就是摆放当前`adapter`中剩余的 Item。

`LinearLayoutManager`会向`Recycler`要`View`来填充`RecyclerView`，所以`RecyclerView`中填几个`View`，其实和`Recycler`有很大的关系，因为`Recycler`不给`LinearLayoutManager`的话，`RecyclerView`中就不会有`View`填充。那`Recycler`给`LinearLayoutManager``View`的边界条件是什么呢?
来看一下`tryGetViewHolderForPositionByDeadline()`方法:

```
ViewHolder tryGetViewHolderForPositionByDeadline(int position, boolean dryRun,
long deadlineNs) {
        if (position < 0 || position >= mState.getItemCount()) {
            throw new IndexOutOfBoundsException("Invalid item position " +
position
                    + "(" + position + "). Item count:" + mState.getItemCount()
                    + exceptionLabel());
        }
}
```

即如果位置大于`mState.getItemCount()`，那么就不会再向`RecyclerView`中填充子 View。而这个`mState.getItemCount()`一般就是`adapter`中当前数据源的数量。所以经过这一步布局后，View 的状态就成了最终形态。

dispatchLayoutStep3(执行删除动画)

接下来`dispatchLayoutStep3()`就会做删除动画:

```
private void dispatchLayoutStep3() {
    ...
    if (mState.mRunSimpleAnimations) {
        ...
        mViewInfoStore.process(mViewInfoProcessCallback); //触发动画的执行
    }
    ...
}
```

可以看到主要涉及到动画的是`mViewInfoStore.process()`，其实这一步可以分为两个操作:

1. 先把`Item View`动画前的起始状态准备好。
2. 执行动画使`Item View`到目标布局位置。

把 item view 动画前的起始状态准备好

```
 void process(ProcessCallback callback) {
        for (int index = mLayoutHolderMap.size() - 1; index >= 0; index--) { //
对 mLayoutHolderMap 中每一个 Holder 执行动画
            final RecyclerView.ViewHolder viewHolder =
mLayoutHolderMap.keyAt(index);
```

```
            final InfoRecord record = mLayoutHolderMap.removeAt(index);
            if ((record.flags & FLAG_APPEAR_AND_DISAPPEAR) ==
FLAG_APPEAR_AND_DISAPPEAR) {
                callback.unused(viewHolder);
            } else if ((record.flags & FLAG_DISAPPEARED) != 0) {
                callback.processDisappeared(viewHolder, record.preInfo,
record.postInfo);  //被删除的那个 item 会回调到这个地方
            }else if ((record.flags & FLAG_PRE_AND_POST) == FLAG_PRE_AND_POST)
{
                callback.processPersistent(viewHolder, record.preInfo,
record.postInfo);   //需要上移的 item 会回调到这个地方
            }
            ...
            InfoRecord.recycle(record);
        }
    }
```

这一步就是遍历`mLayoutHolderMap`对其中的每一个 ViewHolder 做对应的动画。这里`callback`会调到了`RecyclerView`,`RecyclerView`会对每一个`Item`执行相应的动画：

```
ViewInfoStore.ProcessCallback mViewInfoProcessCallback =
        new ViewInfoStore.ProcessCallback() {
            @Override
            public void processDisappeared(ViewHolder viewHolder, @NonNull
ItemHolderInfo info,@Nullable ItemHolderInfo postInfo) {
                mRecycler.unscrapView(viewHolder);   //从 scrap 集合中移除,
                animateDisappearance(viewHolder, info, postInfo);
            }

            @Override
            public void processPersistent(ViewHolder viewHolder, @NonNull
ItemHolderInfo preInfo, @NonNull ItemHolderInfo postInfo) {
                ...
                if (mItemAnimator.animatePersistence(viewHolder, preInfo,
postInfo)) {
                    postAnimationRunner();
                }
            }
            ...
        }
}
```

先分析被删除那那个`Item`的消失动画：

- 将Item的动画消失动画放入到`mPendingRemovals`待执行队列

```
void animateDisappearance(@NonNull ViewHolder holder, @NonNull
ItemHolderInfo preLayoutInfo, @Nullable ItemHolderInfo postLayoutInfo) {
    addAnimatingView(holder);
    holder.setIsRecyclable(false);
    if (mItemAnimator.animateDisappearance(holder, preLayoutInfo,
postLayoutInfo)) {
        postAnimationRunner();
    }
}
```

先把`Holder`attch到`RecyclerView`上(这是因为在`dispatchLayoutStep1`和`dispatchLayoutStep2`中已经对这个`Holder`做了Dettach)。即它又重新出现在了`RecyclerView`的布局中(位置当然还是未删除前的位置)。然后调用了`mItemAnimator.animateDisappearance()`其执行这个删除动画，`mItemAnimator`是`RecyclerView`的动画实现者，它对应的是`DefaultItemAnimator`。继续看`animateDisappearance()`它其实最终调用到了`DefaultItemAnimator.animateRemove()`:

```
public boolean animateRemove(final RecyclerView.ViewHolder holder) {
    resetAnimation(holder);
    mPendingRemovals.add(holder);
    return true;
}
```

其实并没有执行动画，而是把这个`holder`放入了`mPendingRemovals`集合中，看样是要等下执行。

- 将未被删除的Item的移动动画放入到`mPendingMoves`待执行队列

  其实逻辑和上面差不多`DefaultItemAnimator.animatePersistence()`:

```
public boolean animatePersistence(@NonNull RecyclerView.ViewHolder
viewHolder,@NonNull ItemHolderInfo preInfo, @NonNull ItemHolderInfo
postInfo) {
    if (preInfo.left != postInfo.left || preInfo.top != postInfo.top) {  //
和预布局的状态不同，则执行move动画
        return animateMove(viewHolder,preInfo.left, preInfo.top,
postInfo.left, postInfo.top);
    }
    ...
}
```

`animateMove`的逻辑也很简单，就是根据偏移构造了一个`MoveInfo`然后添加到`mPendingMoves`中，也没有立刻执行。

```
public boolean animateMove(final RecyclerView.ViewHolder holder, int fromX,
int fromY, int toX, int toY) {
    final View view = holder.itemView;
```

```
    fromX += (int) holder.itemView.getTranslationX();
    fromY += (int) holder.itemView.getTranslationY();
    resetAnimation(holder);
    int deltaX = toX - fromX;
    int deltaY = toY - fromY;
    if (deltaX == 0 && deltaY == 0) {
        dispatchMoveFinished(holder);
        return false;
    }
    if (deltaX != 0) {
        view.setTranslationX(-deltaX);  //设置他们的位置为负偏移！！！！！
    }
    if (deltaY != 0) {
        view.setTranslationY(-deltaY);  //设置他们的位置为负偏移！！！！！
    }
    mPendingMoves.add(new MoveInfo(holder, fromX, fromY, toX, toY));
    return true;
}
```

但要注意这一步把要做滚动动画的View的`TranslationX`和`TranslationY`都设置负的被删除的Item 的高度，**即被删除的Item之后的Item都下移了。**

- `postAnimationRunner()`执行所有的 pending 动画

  上面一步操作已经把动画前的状态准备好了，`postAnimationRunner()`就是将上面`pendding`的动画开始执行:

```
//DefaultItemAnimator.java
public void runPendingAnimations() {
        boolean removalsPending = !mPendingRemovals.isEmpty();
        ...
        for (RecyclerView.ViewHolder holder : mPendingRemovals) {
            animateRemoveImpl(holder); //执行pending的删除动画
        }
        mPendingRemovals.clear();

        if (!mPendingMoves.isEmpty()) { //执行pending的move动画
            final ArrayList<MoveInfo> moves = new ArrayList<>();
            moves.addAll(mPendingMoves);
            mMovesList.add(moves);
            mPendingMoves.clear();
            Runnable mover = new Runnable() {
                @Override
                public void run() {
                    for (MoveInfo moveInfo : moves) {
                        animateMoveImpl(moveInfo.holder, moveInfo.fromX,
moveInfo.fromY,
                                moveInfo.toX, moveInfo.toY);
                    }
```

```
                    moves.clear();
                    mMovesList.remove(moves);
                }
            };
            if (removalsPending) {
                View view = moves.get(0).holder.itemView;
                ViewCompat.postOnAnimationDelayed(view, mover,
getRemoveDuration());
            } else {
                mover.run();
            }
        }
        ...
    }
```

`animateRemoveImpl`和`animateMoveImpl`的操作：

1. `animateRemoveImpl` 把这个被Remove的Item做一个透明度由（1~0）的动画
2. `animateMoveImpl`把它们的`TranslationX`和`TranslationY`移动到 0。的位置。

RecyclerView 的使用总结以及常见问题解决方案

RecyclerView使用常见的问题和需求

RecyclerView 设置了数据不显示

没有设置`LayoutManger`。没有`LayoutManger`的话`RecycleView`是无法布局的，即是无法展示数据。

`RecycleView`布局的源码：

```
void dispatchLayout() {  //没有设置 Adapter 和 LayoutManager,  都不可能有内容
    if (mAdapter == null) {
            Log.e(TAG, "No adapter attached; skipping layout");
            // leave the state in START
            return;
    }
    if (mLayout == null) {
            Log.e(TAG, "No layout manager attached; skipping layout");
            // leave the state in START
            return;
    }
}
```

即`Adapter`或`Layout`任意一个为 null，就不会执行布局操作。

RecyclerView 数据多次滚动后出现混乱

`RecycleView`在滚动过程中`ViewHolder`是会不断复用的，因此就会带着上一次展示的 UI 信息 (也包含滚动状态)， 所以在设置一个`ViewHolder`的 UI 时，尽量要做`resetUi()`操作：

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {
        holder.itemView.resetUi()
        ...//设置信息UI
}
```

`resetUi()`这个方法就是用来把 UI 还原为最初的操作。当然如果你的每一次`bindData`操作会对每一个 UI 对象重新赋值的话就不需要有这个操作。就不会出现`itemView`的UI混乱问题。

如何获取当前 ItemView 展示的位置

可能会有这样的需求: 当`RecycleView`中的特定`Item`滚动到某个位置时做一些操作。比如某个`Item`滚动到顶部时，展示搜索框。那怎么实现呢?

可以直接获取这个`Item`的`ViewHolder`：

```
    val holder = recyclerView.findViewHolderForAdapterPosition(speicalItemPos)
?: return

    val offsetWithScreenTop = holder.itemview.top

    if(offsetWithScreenTop <= 0){  //这个ItemView已经滚动到屏幕顶部
        //do something
    }
```

如何在固定时间内滚动一段距离

```
//自定义 LayoutManager,  Hook smoothScrollToPosition 方法
recyclerView.layoutManager = object : LinearLayoutManager(this,
LinearLayoutManager.VERTICAL, false) {
    override fun smoothScrollToPosition(recyclerView: RecyclerView?, state:
RecyclerView.State?, position: Int) {
        if (recyclerView == null) return
        val scroller = get200MsScroller(recyclerView.context, position * 500)
        scroller.targetPosition = position
        startSmoothScroll(scroller)
    }
}

private fun get200MsScroller(context: Context, distance: Int):
RecyclerView.SmoothScroller = object : LinearSmoothScroller(context) {
    override fun calculateSpeedPerPixel(displayMetrics: DisplayMetrics): Float
{
        return (200.0f / distance) //表示滚动 distance 花费200ms
    }
```

```
}
```

如何测量当前 RecyclerView 的高度

需求: `RecyclerView`中的每个`ItemView`的高度都是不固定的。数据源中有 20 条数据，在没有渲染的情况下想知道这个 20 条数据被`RecycleView`渲染后的总共高度。

思路是利用`LayoutManager`来测量，因为`RecycleView`在对`子View`进行布局时就是用`LayoutManager`来测量`子View`来计算还有多少剩余空间可用，源码如下：

```
   void layoutChunk(RecyclerView.Recycler recycler, RecyclerView.State
state,LayoutState layoutState, LayoutChunkResult result) {
        View view = layoutState.next(recycler);    //这个方法会向 recycler要一个
View
        ...
        measureChildWithMargins(view, 0, 0);  //测量这个View的尺寸，方便布局，这个方
法是public
        ...
    }
```

所以也可以利用`layoutManager.measureChildWithMargins`方法来测量，代码如下：

```
    private fun measureAllItemHeight():Int {
        val measureTemplateView = SimpleStringView(this)
        var totalItemHeight =
        dataSource.forEach {  //dataSource当前中的所有数据
            measureTemplateView.bindData(it, 0) //设置好UI数据
            recyclerView.layoutManager.measureChild(measureTemplateView, 0, 0)
//调用源码中的子View的测量方法
            currentHeight += measureTemplateView.measuredHeight
        }
        return totalItemHeight;
    }
```

但要注意的是，这个方法要等布局稳定的时候才可以用，如果你在`Activity.onCreate`中调用，那么应该`post`一下， 即：

```
recyclerView.post{
    val totalHeight = measureAllItemHeight()
}
```

IndexOutOfBoundsException: Inconsistency detected. Invalid item position 5(offset:5).state:9

这个异常通常是由于`Adapter的数据源大小`改变没有及时通知`RecycleView`做 UI 刷新导致的，或者通知的方式有问题。 比如如果数据源变化了(比如数量变少了)，而没有调用`notifyXXX`, 那么此时滚动`RecycleView`就会产生这个异常。

解决办法很简单：**`Adapter的数据源`改变时应立即调用`adapter.notifyXXX`来刷新`RecycleView`** 。

在使用`RecycleView`时一定要注意保证**数据和UI的同步，数据变化，及时刷新RecyclerView**, 这样就能避免很多crash。

RecyclerView 原理

* ListView

* ListView 的优化

ListView 原理解析

布局流程

ListView 通过复写 layoutChildren() 方法进行布局。

1. ListView 的 layoutChildren() 方法

   （1）数据有改变，将所有的 view 全部都添加到 scrapView 集合中（已被回收，等待重用的集合），准备重新设置 item 时复用；数据没有改变，将所有的 view 添加到 activeViews （正在显示的集合）中准备复用。
   （2）将所有 view 从 parent 上 detach 掉。
   （3）根据 mLayoutMode 决定 view 的方向，并布局（一般情况时从顶到底，调用了 fillFromTop()方法进行布局）。

2. ListView 的 fillFromTop() 方法调用了 fillDown() 方法

   循环 item，调用 makeAndAddView() 方法拿到其 view，根据 view 的大小更新 top，如果 top 已经大于 ViewGroup 的 bottom或者全部 item 添加完成则停止。

3. makeAndAddView() 方法就是获取和添加每个 itemView 的方法，也是复用 view 的地方。

   （1）没有数据改变时，从 activeViews（正在展示的 view 集合）中用 position 拿到 view，然后调用 setupChild() 方法添加 view 到层级上。

   （2）有数据变化时，调用 obtainView() 方法获取到复用或者新建的 view，然后调用 setupChild() 方法添加 view 到层级上。

4. obtainView() 实现在 ListView 的父类 AbsListView 中，会拿到复用或者新建的 view。

   （1）用 position 从 scrapViews 中拿到复用的 view scrapeView。

   （2）调用 adapter 的 getView() 方法拿到 view child，并传递了scrapeView 作为参数。

   （3）如果 scrapeView 为 null，则说明是新建的 view，如果 scrapeView 不为 null，并且 scrapeView 等于 child，则说明 view 是复用的，后续不用执行 measure，如果 scrapeView 不等于 child，则将 scrapeView 添加到 scrapeViews （复用 view 集合）中。

5. setupChild() 方法会将视图添加为子视图。

   （1）如果是复用的 view，将 view 重新 attach 到 parent 上即可。

   （2）如果是新建的 view，则调用 addViewInLayout 将 view 添加到 parent 里。

   （3）然后根据需要来 measure 和 layout 该 child 完成添加。

滚动时 view 的展示与复用

拖动滑动

1. 在 AbsListView 的 onTouchEvent() 方法中，当捕获到 MOVE 事件是调用了 onTouchMove() 方法，onTouchMove() 方法判断 touchMode 是 scroll 时调用了 scrollIfNeeded() 方法，计算从开始触摸到此刻的 deltaY 距离以及增量 incrementY，调用 trackMotionScroll 方法进行处理 view。
2. trackMotionScroll()

   （1）trackMotionScroll() 方法中通过 incremetalDeltaY 的正负判断 ListView 是向上还是向下滚动，为正是向下滚动，为负是向上滚动。

   （2）滑出屏幕不可见的 view 放到 scrapViews 中，并将回收的 view 从 parent 上 detach 掉。

   （3）调用 offsetChildrenToAndBottom() 方法通过改变每一个 view 的 top 来移动 view。

   （4）如果有新的 view 可见，调用 fillGap()方法进行添加（fillGap 还是 fillDown 或者 fillup）。

Fling 滚动

1. 在 onTouchEvent() 中，当捕获到 UP 事件（松开手指）是调用 onTouchUp() 方法，在 onTouchUp() 方法中判断 mTouchMode 为 TOUCH_MODE_SCROLL 时，使用 VelocityTracker.computeCurrentVelocity() 方法计算滑动速度，以此速度开启一个 scroll 对象的 fling 状态，然后执行了 FlingRunnable 对象。
2. 在 FlingRunnable 的 run 方法中不断的计算 scroll 当前的滑动两，调用 trackMotionScroll() 方法执行滚动操作。

数据刷新 - AdapterDataSetObserver 观察者模式

Adapter 用于管理一个数据集合及其种种操作，并且还应该可以在数据发生改变的时候得到相应的通知，所以， Adapter 使用了观察者模式，允许向其注册多个 DataSetObserver 对象，当 adapter 的数据发生改变时，通知这些观察者，使其完成自己的操作。

1. Adapter 可以注册/取消观察者

```
void registerDataSetObserver(DataSetObserver observer);
void unregisterDataSetObserver(DataSetObserver observer);
```

2. AdapterView 自己有一个观察者内部类，子类注册到 adapter 上，以便在数据发生改变时做相应处理。

```
// AdapterView 的内部类
class AdapterDataSetObserver extends DataSetObserver {

        @Override
        public void onChanged() {
```

```
            mDataChanged = true; // 数据已更新
            mOldItemCount = mItemCount; // 更新  oldItemCount
            mItemCount = getAdapter().getCount(); // 更新 itemCount

            // Detect the case where a cursor that was previously invalidated
has
            // been repopulated with new data.
            if (AdapterView.this.getAdapter().hasStableIds() && mInstanceState
!= null
                    && mOldItemCount == 0 && mItemCount > 0) {
                AdapterView.this.onRestoreInstanceState(mInstanceState);
                mInstanceState = null;
            } else {
                rememberSyncState();
            }
            checkFocus();
            requestLayout(); // 重新请求布局
        }
        }
```

3. AbsListView 在创建时会主动创建一个继承自上述观察者类的观察者并在 adapter 更新时注册到 adapter 上。

```
        if (mAdapter != null && mDataSetObserver == null) {
            mDataSetObserver = new AdapterDataSetObserver();
            mAdapter.registerDataSetObserver(mDataSetObserver);

            // Data may have changed while we were detached. Refresh.
            mDataChanged = true;
            mOldItemCount = mItemCount;
            mItemCount = mAdapter.getCount();
        }
```

4. 在 BaseAdapter 的 notifyDataSetChanged() 方法中会调用注册的 DataSetObserver 的 change()。

由代码可知，ListView 自己创建一个观察者，并在 adapter 更新的地方保证给其注册一个该观察者，这样，当数据更新时经常调用的 notifyDataSetChanged 方法被调用后，会触发相应观察者的方法，也就是 onChanged 方法，此时会更新状态及数据信息，然后请求重新布局绘制，最终会回到 layoutChildren 方法开始布局。

除此之外，还可以创建自己的观察者注册到 adapter 上，不用担心被覆盖，因为 BaseAdapter 实现 Observable 来注册观察者，该类维护的是一个观察者数组，需要注意的是，notify 的时候会从后往前一次调用观察者的 onChanged 方法。

Adapter 装饰器模式

ListView 是可以添加多个 Header 和 Footer 的，ListView 也会将其作为 adapter 的一部分。

Header 和 Footer 的 adapter

ListView 对其管理的 adapter 使用了装饰器模式，构建 adapter 时，当有 header 或 footer 时，ListView 会将 adapter 作为被装饰者，连同 Header 和 Footer 信息一起创建一个自己维护的带 Header 和 Footer 的 adapter 对象，作为真正管理的 adapter，在管理过程中调用的 adapter 的任何方法都是在这个 adapter 上调用的，只不过该 adapter 的方法又都调用自己的 adapter (被装饰者)的相应方法，所以可以完全通过使用自己的 adapter 的方法达到任何效果。

```
public class HeaderViewListAdapter implements WrapperListAdapter, Filterable {

    private final ListAdapter mAdapter;//被装饰者，也就是自己写的 adapter

    // Header 和 Footer 信息
    ArrayList<ListView.FixedViewInfo> mHeaderViewInfos;
    ArrayList<ListView.FixedViewInfo> mFooterViewInfos;

    public boolean isEmpty() {
        return mAdapter == null || mAdapter.isEmpty();
    }


     public int getCount() {
        if (mAdapter != null) {
            return getFootersCount() + getHeadersCount() + mAdapter.getCount();
        } else {
            return getFootersCount() + getHeadersCount();
        }
    }

      public Object getItem(int position) {
        // Header (negative positions will throw an IndexOutOfBoundsException)
        int numHeaders = getHeadersCount();
        if (position < numHeaders) {
            return mHeaderViewInfos.get(position).data;
        }

        // Adapter
        final int adjPosition = position - numHeaders;
        int adapterCount = 0;
        if (mAdapter != null) {
            adapterCount = mAdapter.getCount();
            if (adjPosition < adapterCount) {
                return mAdapter.getItem(adjPosition);
            }
        }
```

```
        // Footer (off-limits positions will throw an
IndexOutOfBoundsException)
        return mFooterViewInfos.get(adjPosition - adapterCount).data;
    }

      public long getItemId(int position) {
        int numHeaders = getHeadersCount();
        if (mAdapter != null && position >= numHeaders) {
            int adjPosition = position - numHeaders;
            int adapterCount = mAdapter.getCount();
            if (adjPosition < adapterCount) {
                return mAdapter.getItemId(adjPosition);
            }
        }
        return -1;
    }

      public View getView(int position, View convertView, ViewGroup parent) {
        // Header (negative positions will throw an IndexOutOfBoundsException)
        int numHeaders = getHeadersCount();
        if (position < numHeaders) {
            return mHeaderViewInfos.get(position).view;
        }

        // Adapter
        final int adjPosition = position - numHeaders;
        int adapterCount = 0;
        if (mAdapter != null) {
            adapterCount = mAdapter.getCount();
            if (adjPosition < adapterCount) {
                return mAdapter.getView(adjPosition, convertView, parent);
            }
        }

        // Footer (off-limits positions will throw an
IndexOutOfBoundsException)
        return mFooterViewInfos.get(adjPosition - adapterCount).view;
    }

      public int getItemViewType(int position) {
        int numHeaders = getHeadersCount();
        if (mAdapter != null && position >= numHeaders) {
            int adjPosition = position - numHeaders;
            int adapterCount = mAdapter.getCount();
            if (adjPosition < adapterCount) {
                return mAdapter.getItemViewType(adjPosition);
            }
        }
```

```
        return AdapterView.ITEM_VIEW_TYPE_HEADER_OR_FOOTER;
    }
}
```

由源码可知，该 adapter 维护了自己实现的 adapter，在重写的方法里，处理了关于 header 和 footer 的各种情况，并把普通 item 的情况交由自己实现的 adapter 来处理，所以自己实现的 adapter 只需关注自己的数据集合即可。

ListView 创建 adapter

在 4.4 版本之前，通过 ListView 的 addHeaderView 和 addFooterView 添加 header 和 footer，然后在 setAdapter 的时候，会根据有没有 header 或 footer 信息决定要不使用装饰器；如果在 setAdapter 之后再去调用这些方法将会出现问题。

从 4.4 版本开始，ListView 解决了这个问题，在 setAdapter 之后添加 header 或 footer时，增加了对 adapter 的再创建，如果原来没有使用装饰器，则重新构建一个装饰器作为 adapter，并且还会调用上面说到的自带的观察者的 onChanged 方法进行刷新。

- setAdapter

```
    @Override
    public void setAdapter(ListAdapter adapter) {
        if (mAdapter != null && mDataSetObserver != null) { // 取消旧 adapter 的观察者
            mAdapter.unregisterDataSetObserver(mDataSetObserver);
        }

        resetList(); // 重置 view 状态
        mRecycler.clear();

        if (mHeaderViewInfos.size() > 0|| mFooterViewInfos.size() > 0) { // 有 header 或者 footer 则要使用装饰器 adapter
            mAdapter = wrapHeaderListAdapterInternal(mHeaderViewInfos, mFooterViewInfos, adapter);
        } else {
            mAdapter = adapter;
        }

        mOldSelectedPosition = INVALID_POSITION;
        mOldSelectedRowId = INVALID_ROW_ID;

        // AbsListView#setAdapter will update choice mode states.
        super.setAdapter(adapter);

        if (mAdapter != null) {
            mAreAllItemsSelectable = mAdapter.areAllItemsEnabled(); // 更新信息
            mOldItemCount = mItemCount;
            mItemCount = mAdapter.getCount();
            checkFocus();
```

```
            mDataSetObserver = new AdapterDataSetObserver();
            mAdapter.registerDataSetObserver(mDataSetObserver); // 新 adapter 注
册观察者

            mRecycler.setViewTypeCount(mAdapter.getViewTypeCount()); // 更新
RecycleBin 信息

            int position;
            if (mStackFromBottom) {
                position = lookForSelectablePosition(mItemCount - 1, false);
            } else {
                position = lookForSelectablePosition(0, true);
            }
            setSelectedPositionInt(position);
            setNextSelectedPositionInt(position);

            if (mItemCount == 0) {
                // Nothing selected
                checkSelectionChanged();
            }
        } else {
            mAreAllItemsSelectable = true;
            checkFocus();
            // Nothing selected
            checkSelectionChanged();
        }

        requestLayout(); // 刷新布局
    }
```

- addHeaderView

```
    public void addHeaderView(View v, Object data, boolean isSelectable) {
        ...
        final FixedViewInfo info = new FixedViewInfo();
        info.view = v;
        info.data = data;
        info.isSelectable = isSelectable;
        mHeaderViewInfos.add(info); // 添加 header 信息
        mAreAllItemsSelectable &= isSelectable;

        // Wrap the adapter if it wasn't already wrapped.
        if (mAdapter != null) {
            if (!(mAdapter instanceof HeaderViewListAdapter)) { // 如果还没有使用
装饰器则使用装饰器构建 adapter
                wrapHeaderListAdapterInternal();
            }
```

```
            // In the case of re-adding a header view, or adding one later on,
            // we need to notify the observer.
            if (mDataSetObserver != null) { // 通知刷新布局
                mDataSetObserver.onChanged();
            }
        }
    }
```

item 的点击事件

onItemClickListener

ListView 的 item 点击事件可以通过 setOnItemClickListener 设置，该监听器是 AdapterView 维护的一个监听器，可以监听到每个 item 的点击事件，他并不是通过给每个 itemView 设置 OnClickListener 来监听点击事件，否则每次使用 view 都会创建新的监听器对象。

ListView (实际上是 AbsListView 管理) 是在 onTouchEvent 时判断当前触摸的 item 的 position，然后 postDelay 一个 PerformClick 内部类对象，该对象的 run 方法，取到相应 position 的 childView，然后调用 onItemClickListener 回调。

1. 在 AbsListView 的 onTouchEvent() 中触发 MotionEvent.ACTION_DOWN 事件，调用了 onTouchDown() 方法。
2. 在 onTouchDown() 方法中会调用 pointToPosition() 方法获取触摸点 item 的 position。

```
final int x = (int) ev.getX();
final int y = (int) ev.getY();
int motionPosition = pointToPosition(x, y);
```

3. 在 positionToPosition() 方法中会遍历所有的 view，如果触摸点在该 view 内，则返回该点。

```
    public int pointToPosition(int x, int y) {
        Rect frame = mTouchFrame;
        if (frame == null) {
            mTouchFrame = new Rect();
            frame = mTouchFrame;
        }

        final int count = getChildCount();
        // 遍历每一个 view
        for (int i = count - 1; i >= 0; i--) {
            final View child = getChildAt(i);
            if (child.getVisibility() == View.VISIBLE) {
                child.getHitRect(frame);
                if (frame.contains(x, y)) { // 如果触摸点在该 view 内
                    return mFirstPosition + i;
                }
            }
        }
```

```
        return INVALID_POSITION;
    }
```

4. 在 AbsListView 的 onTouchEvent() 中触发 MotionEvent.ACTION_UP 事件，调用了 onTouchUp() 方法。
5. 在 onTouchUp() 方法中通过 handler 发送一个点击事件处理的 runnable

```
                    if (mPerformClick == null) {
                        mPerformClick = new PerformClick();
                    }

                    final AbsListView.PerformClick performClick =
mPerformClick;
                    performClick.mClickMotionPosition = motionPosition;
                    performClick.rememberWindowAttachCount();
performClick.run();
```

6. performClick 的 run 方法中根据 itemPosition 找到 child view，然后调用了 performItemChick 方法处理点击事件。

```
    private class PerformClick extends WindowRunnnable implements Runnable {
        int mClickMotionPosition;

        @Override
        public void run() {
            // The data has changed since we posted this action in the event
queue,
            // bail out before bad things happen
            if (mDataChanged) return;

            final ListAdapter adapter = mAdapter;
            final int motionPosition = mClickMotionPosition;
            if (adapter != null && mItemCount > 0 &&
                    motionPosition != INVALID_POSITION &&
                    motionPosition < adapter.getCount() && sameWindow() &&
                    adapter.isEnabled(motionPosition)) {
                // 根据 itemPosition 找到 child 的 position
                final View view = getChildAt(motionPosition - mFirstPosition);
                // If there is no view, something bad happened (the view
scrolled off the
                // screen, etc.) and we should cancel the click
                if (view != null) {
                    performItemClick(view, motionPosition,
adapter.getItemId(motionPosition));
                }
            }
        }
    }
```

7. AbsListView 的 performItemClick() 方法调用了 AdapterView（AbsListView 的父类）的 performItemClick() 方法，在 AdapterView 的 performItemClick() 方法中调用了 onItemClick() 方法。

```
    public boolean performItemClick(View view, int position, long id) {
        final boolean result;
        if (mOnItemClickListener != null) {
            playSoundEffect(SoundEffectConstants.CLICK);
          // view 是点击的 ViewGroup 的子 View
            mOnItemClickListener.onItemClick(this, view, position, id);
            result = true;
        } else {
            result = false;
        }

        if (view != null) {
            view.sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
        }
        return result;
    }
    // mOnItemClickListener 就是调用方法设置的监听。
    public void setOnItemClickListener(@Nullable OnItemClickListener listener)
{
        mOnItemClickListener = listener;
    }
```

onItemLongClickListener

onItemLongClickListener 是监听 item 的长按事件对象，与 onItemClickListener 类似，他也不是通过给每个 child 设置监听器来实现。

1. 在 AbsListView 的 onTouchDown 方法中，postDelay 一个点击 runnable，在其 run 方法里，找到对应 position 的 child，如果可以长按(isLongClickable)，则 postDelay 一个长按事件 runnable，delay 为系统默认的长按事件的响应时间，在该 run 方法中执行 onItemLongClickListener 回调。

```
    private void onTouchDown(MotionEvent ev) {
                        if (mPendingCheckForTap == null) {
                        mPendingCheckForTap = new CheckForTap();
                    }

                    mPendingCheckForTap.x = ev.getX();
                    mPendingCheckForTap.y = ev.getY();
                    postDelayed(mPendingCheckForTap,
ViewConfiguration.getTapTimeout()); // 执行轻触事件
    }
```

2. mPendingCheckForTap 的 run() 方法中提交了一个长按延时事件，当点击或者长按事件发生后，

这个 runnable 在相应的方法中就会被 removeCallback 调不再执行；在 onTouchMove 时也会取消相应的 runnable 。

```
if (longClickable) {
                            if (mPendingCheckForLongPress == null) {
                                mPendingCheckForLongPress = new
CheckForLongPress();
                            }
                            mPendingCheckForLongPress.setCoords(x, y);

 mPendingCheckForLongPress.rememberWindowAttachCount();
                            postDelayed(mPendingCheckForLongPress,
longPressTimeout);// 提交一个长按事件 runnable，delay 时间为系统默认长按响应时间(500毫秒)
                        } else {
                            mTouchMode = TOUCH_MODE_DONE_WAITING;
                        }
```

3. mPendingCheckForLongPress 的 run 方法处理了长按事件的响应

```
        @Override
        public void run() {
            final int motionPosition = mMotionPosition;
            final View child = getChildAt(motionPosition - mFirstPosition);
            if (child != null) {
                final int longPressPosition = mMotionPosition;
                final long longPressId = mAdapter.getItemId(mMotionPosition);

                boolean handled = false;
                if (sameWindow() && !mDataChanged) {
                    if (mX != INVALID_COORD && mY != INVALID_COORD) {
                        handled = performLongPress(child, longPressPosition,
longPressId, mX, mY);// 处理事件
                    } else {
                        handled = performLongPress(child, longPressPosition,
longPressId);// 处理事件
                    }
                }

                if (handled) {
                    mHasPerformedLongPress = true;
                    mTouchMode = TOUCH_MODE_REST;
                    setPressed(false);// 处理 view 状态
                    child.setPressed(false);
                } else {
                    mTouchMode = TOUCH_MODE_DONE_WAITING;
                }
            }
```

```
        }
```

4. performLongPress() 方法调用了 OnItemLongClickListener 的 onItemLongClick() 方法

```
    boolean performLongPress(final View child,
            final int longPressPosition, final long longPressId, float x, float
y) {
        // CHOICE_MODE_MULTIPLE_MODAL takes over long press.
        if (mChoiceMode == CHOICE_MODE_MULTIPLE_MODAL) {
            if (mChoiceActionMode == null &&
                    (mChoiceActionMode =
startActionMode(mMultiChoiceModeCallback)) != null) {
                setItemChecked(longPressPosition, true);
                performHapticFeedback(HapticFeedbackConstants.LONG_PRESS);
            }
            return true;
        }

        boolean handled = false;
        if (mOnItemLongClickListener != null) {
            handled =
mOnItemLongClickListener.onItemLongClick(AbsListView.this, child,
                    longPressPosition, longPressId);
        }
        if (!handled) {
            mContextMenuInfo = createContextMenuInfo(child, longPressPosition,
longPressId);
            if (x != CheckForLongPress.INVALID_COORD && y !=
CheckForLongPress.INVALID_COORD) {
                handled = super.showContextMenuForChild(AbsListView.this, x,
y);
            } else {
                handled = super.showContextMenuForChild(AbsListView.this);
            }
        }
        if (handled) {
            performHapticFeedback(HapticFeedbackConstants.LONG_PRESS);
        }
        return handled;
    }
```

position 和 itemId 的区别

在设置 onItemClickListener 时，重写的方法里的参数有 position 和 id，一般习惯性的使用 getItem(position) 取得相应的数据，但是这在有 header 和 footer 的时候是有问题的，先来看下 getItemId 的代码：

```
public long getItemId(int position) {
    int numHeaders = getHeadersCount();
    if (mAdapter != null && position >= numHeaders) {
        int adjPosition = position - numHeaders;
        int adapterCount = mAdapter.getCount();
        if (adjPosition < adapterCount) {
            return mAdapter.getItemId(adjPosition);//返回用户定义的id
        }
    }
    return -1;
}
```

这是 HeaderViewListAdapter 的 getItemId 方法，在 AbsListView 里的 PerformClick 里来处理 onItemClick 事件，传入的 position 时触摸点所在的 view 的 position，他是包含了 header 和 footer 的位置的，所以在有 header 和 footer 的情况下通过 getItem(position) 拿数据时就会错位，而且会有越界异常。

而 getItemId() 方法，会将 header 和 footer 的 id 返回 -1，否则会返回自定义 adapter 的 getItemId() 方法返回的 id，此时传入的参数是数据正确的 position，所以只要在自定义的 adapter 中，重写 getItemId() 方法，并直接返回这个 position，就可以保证 OnItemClickListener 中拿到的 id，不是 -1(headers) 就是正确数据的 position，稍加判断即可。

要想使用这个 itemId，需要重写 hasStableIds() 方法并返回 true，否则无效。

缓存

缓存机制

把不需要实时更新的数据缓存下来，通过时间或者其他因素来判别是读缓存还是网络请求，这样可以缓解服务器压力，一定程度上提高应用响应速度，并且支持离线阅读。

访问网络的数据常见返回格式有图片、文件和数据库，因此从这几个方向考虑缓存的实现。

图片缓存

常见的优化就是子项不可见时，所占用的内存会被回收以供正在前台显示子项使用。如果想让 UI 运行流畅的话，就不应该每次显示时都去重新加载图片。保持一些内存和文件缓存就变得很有必要了。

内存缓存

通过预先消耗应用的一点内存来存储数据，便可快速的为应用中的组件提供数据，是一种典型的以空间换时间的策略。

LruCache 类（Android v4 Support Library 类库中开始提供）非常适合来做图片缓存任务，它可以使用一个 LinkedHashMap 的强引用来保存最近使用的对象，并且当它保存的对象占用的内存总和超出了为它设计的最大内存时会把不经常使用的对象踢出以供垃圾回收器回收。

给 LruCache 设置一个合适的内存大小，需考虑如下因素：

- 还剩余多少内存给 activity 或应用使用
- 屏幕上需要一次性显示多少张图片和多少图片在等待显示

- 手机的大小和密度是多少（密度越高的设备需要越大的缓存）
- 图片的尺寸（决定了所占用的内存大小）
- 图片的访问频率（频率高的在内存中一直保存）
- 保存图片的质量（不同像素在不同情况下显示）

磁盘缓存

内存缓存能够快速的获取到最近显示的图片，但不一定就能够获取到需要的图片缓存。当数据集过大时很容易把内存缓存填满（如 GridView）。应用也有可能被其他的任务（比如来电）中断进行到后台，后台应用有可能会被杀死，那么相应的内存缓存对象也会被销毁。当应用重新回到前台显示时，应用又需要一张一张的去加载图片了。

硬盘文件缓存能够用来处理这些情况，保存处理好的图片，当内存缓存不可用的时候，直接读取在硬盘中保存好的图片，这样可以有效的减少图片加载的次数。读取磁盘文件要比直接从内存缓存中读取要慢一些，而且需要在一个 UI 主线程外的线程中进行，因为磁盘的读取速度是不能够保证的，磁盘文件缓存显然也是一种以空间换时间的策略。

DiskLruCache

使用 SQLite 进行缓存

网络请求数据完成后，把文件的相关信息（如 url （一般作为唯一标示）、下载时间、过期时间）等存放到数据库。下次加载的时候根据 url 先从数据库中查询，如果查询到并且时间未过期，就根据路径读取本地文件，从而实现缓存的效果。

注意：缓存的数据库是存放在 /data/data/databases/ 目录下，是占用内存空间的，如果缓存累积，容易浪费内存，需要及时清理缓存。

文件缓存

思路和一般缓存一样，把需要的数据存储在文件中，下次加载时判断文件是否存在和过期（使用 File.lastModified() 方法得到文件的最后修改时间，与当前时间判断），存在并未过期加载文件中的数据，否则请求服务器重新下载。

注意，无网络环境下就默认读取文件缓存中的。

LruCache 知识

LRU 是近期最少使用的算法，它的核心思想是当缓存满时，会优先淘汰那些近期最少使用的缓存对象。采用 LRU 算法的缓存有两种：LruCache 和 DisLruCache，分别用于实现内存缓存和硬盘缓存，其核心思想都是 LRU 缓存算法。

LruCache 的实现原理

LruCache 的核心思想很好理解，就是要维护一个缓存对象列表，其中对象列表的排列方式是按照访问顺序实现的，即一直没访问的对象，将放在队尾，即将被淘汰。而最近访问的对象将放在对头，最后被淘汰。

LruCache 中维护了一个集合 LinkedHashMap，该 LinkedHashMap 是以访问顺序排序的。当调用 put() 方法时，就会在集合中添加元素，并调用 trimToSize() 判断缓存是否已满，如果满了就用 LinkedHashMap 的迭代器删除队尾元素，即近期最少访问的元素。当调用 get() 方法访问缓存对象时，就会调用 LinkedHashMap 的 get() 方法获得对应集合元素，同时会更新该元素到队头。

优化

ANR 相关知识

ANR 全称是 Application Not Responding，意思就是应用程序未响应。如果一个应用无法响应用户的输入，系统就会弹出一个 ANR 对话框，用户可以自行选择继续等待或者是停止当前程序。

ANR 的发生原因

1. 代码自身引起，例如：
    - 主线程阻塞、IOWait 等；

- 主线程进行耗时计算；
- 错误的操作，比如调用了 Thread.wait 或者 Thread.sleep 等。

2. 其他进程间接引起，例如：
    - 当前应用进程进行进程间通信请求其他进程，其他进程的操作长时间没有反馈；
    - 其他进程的 CPU 占用率高，使得当前应用进程无法抢占到 CPU 时间片。

发生 ANR 的条件

Andriod 系统中，ActivityManangerService(简称 AMS) 和 WindowManangerService(简称 WMS) 会检测 App 的响应事件，如果 App 在特定时间无法响应屏幕触摸或键盘输入事件，或者特定事件没有处理完毕，就会出现 ANR。

- InputDispatching Timeout：输入事件分发或屏幕触摸事件超时 5s 未响应完毕；
- BroadcastQueue Timeout：前台广播在 10秒 内、后台广播在 60 秒内未执行完成；
- Service Timeout：前台服务在 20 秒内、后台服务在 200 秒内未执行完成；
- ContentProvider Timeout：内容提供者，在 publish 超时 10s；

分析 ANR 的方法

- ANR 分析方法一：Log

  可以看到 logcat 清晰地记录了 ANR 发生的时间，发生 ANR 所在的报名、类名以及线程的 tid 和一句话概括原因：WaitingInMainSignalCatcherLoop，大概意思为主线程等待异常。

- ANR 分析方法二：traces.txt

  ANR 异常已经输出到 trances.txt 文件，使用 adb 命令把这个文件从手机里导出来。

- ANR 分析方法三：Java 线程调用分析

  通过 JDK 提供的命令可以帮助分析和调试 Java 应用，命令是： jstack {pid}

- ANR 分析方法四：DDMS 分析 ANR 问题
    - 使用 DDMS-----Update Threads 工具
    - 阅读 Update Threads 的输出

如何避免 ANR

不是所有的 ANR 都可找到原因，也受限于当时发生的环境或系统 bug，因此对 ANR ，是避免而不是分析。

基本的思路就是尽量避免在主线程（UI 线程）中做耗时操作，将耗时操作放在子线程中。将 IO 操作在工作线程来处理，减少其他耗时操作和错误操作。

- 使用 AsyncTask 处理耗时 IO 操作。
- 使用 Thread 或者 HandlerThread 时，调用 Process.setThreadPriority(Process.THREAD_PRIORITY_BACKGROUND)设置优先级，否则仍然会降低程序响应，因为默认 Thread 的优先级和主线程相同。
- 使用 Handler 处理工作线程结果，而不是使用 Thread.wait() 或者 Thread.sleep() 来阻塞主线程。
- 四大组件的生命周期方法中尽量避免耗时的代码。
- 如果要在后台进行耗时操作，建议使用 IntentService 处理。
- 在程序启动时，如果要做一些耗时操作，可以选择加上欢迎界面，避免用户察觉卡顿。
- 主程序需要等待其他线程返回结果时，可以加上进度显示，比如使用 ProgressBar 控件，让用户得知进度。
- 使用 Systrace 和 TraceView 找到影响响应的问题，进一步优化。
- 如果是由于内存不足引起的问题，AndroidManifest.xml 文件 < application > 中可以设置 android:largeHeap="true"，以此增大 App 使用内存。不过不建议使用此法，从根本上防止内存泄漏，优化内存使用才是正确的做法。

发生 ANR 条件的源码分析

Service 造成的 Service Timeout

1. Service 创建之前会延迟发送一个消息，而这个消息就是 ANR 的起源；
2. Service 创建完毕，在规定的时间之内执行完毕 onCreate() 方法就移除这个消息，就不会产生 ANR 了；
3. 在规定的时间之内没有完成 onCreate() 的调用，消息被执行，ANR 发生。

BroadcastReceiver 造成的 BroadcastQueue Timeout

1. BroadcastReceiver 在获取广播接收者后，会延迟发送一个消息，而这个消息就是 ANR 的起源，前台广播延时 10 秒，后台广播延时 60 秒；
2. 处理广播消息之后就会移除延时消息，就不会产生 ANR 了；
3. 如果在延时时间之内没有处理完广播消息，延时消息被执行，ANR 发生。

四大组件发生 ANR 的流程基本都是：

1. 发送延时消息
2. 执行相应方法
3. 方法在延时时间内执行完成，取消消息
4. 方法在延时时间内未执行完成，消息触发
5. 接收延时消息，发生 ANR ，调用 AMS.appNotResponding 方法。

性能优化

内存优化

内存泄漏

内存泄漏：内存不在 GC 的掌控范围之内了。

java 的 GC 内存回收机制：某对象不再有任何引用的时候才会进行回收。

GC 的引用点

1. Java 栈中引用的对象
2. 方法静态引用的对象
3. 方法常量引用的对象
4. Native 中 JNI 引用的对象
5. Thread - " 活着的 " 线程

内存溢出

内存泄漏一般导致应用卡顿，极端情况会导致项目 boom。Boom 的原因是因为超过内存的阈值。原因只要有两方面：

1. 代码存在泄漏，内存无法及时释放导致 oom。
2. 一些逻辑消耗了大量内存，无法及时释放或者超过导致 oom。

所谓消耗大量的内存的，绝大多数是因为图片加载。这是 oom 出现最频繁的地方。

如何查看内存溢出

1. 确定是否存在内存溢出的问题

   方法有两种：

- Android -> SystemInformation -> Memory Usage 查看 Object 里面是否有没有被释放的 Views 和 Activity。
- 命令 `adb shell dumpsys meminfo 包名 -d`。

2. 通过看 Memory Montor 工具，检查动作，反复多次执行某一个操作，查看内存的大概变化情况。
3. 使用 Heap SnapShot 工具（堆栈快照）更仔细地查找泄漏的位置。

点击 -> 进行一段时间的监控，生成一个文件 -> package treeview 会出现一些信息，信息内容如下：

- Total count：内存中该类的对象个数
- Heap count：堆内存中该类的对象个数
- Size of：物理大小
- Shallow size：该对象本身占有内存大小
- Retained size：释放该对象后，节省的内存大小
- Depth：深度
- Shadow size：对象本身内存大小
- Domination size：管辖的内存大小

内存分析工具

性能优化工具：

- Heap SnapShot 工具
- Head Views 工具
- LeakCanary 工具
- MAT 工具
- TraceView 工具（Device Monitor）

第三方分析工具：

- Memory Analyzer
- GT Home
- iTest

注意事项

- 长生命周期的对象使用 Application 的上下文，而不是 Activity 的上下文（Context）。
- Animation 会导致内存溢出。 view 显示动画时，View 会持有 Activity 对象，而动画持有 View，动画不 cancel 就会一直去执行 view 的 onDraw 方法，那么 Activity 就会被一直持有，不能被释放，导致内存泄漏。解决方法是在 Activity 的 onDestory() 方法中调用 Animation.cancle() 进行停止，也可以通过自定义 view 来代替动画。

UI 优化

UI 优化主要包括**布局优化**以及 **view 的绘制优化**。

一般会导致卡顿的情况：

1. 人为在 UI 线程中做轻微耗时操作，导致 UI 线程卡顿；
2. 布局 Layout 过于复杂，无法在 16 ms 内完成渲染；
3. 同一时间动画执行的次数过多，导致 CPU 或 GPU 负载过重；
4. View 过度绘制，导致某些像素在同一帧时间内被绘制多次，从而使 CPU 或 GPU 负载过重；
5. View 频繁的触发 measure、layout，导致 measure、layout 累计耗时过多及整个 View 频繁的重新渲染；
6. 内存频繁触发 GC 过多（同一帧中频繁创建内存），导致暂时阻塞渲染操作；
7. 冗余资源及逻辑等导致加载和执行缓慢；
8. ANR；

布局优化

屏幕上的某个像素在同一帧的时间内被绘制了多次，在多层次的 UI 结构里面，如果不可见的 UI 也在做绘制的操作，这就会导致某些像素区域被绘制了多次，这就浪费了大量的 CPU 以及 GPU 资源。

关于布局优化的方法

1. 如果父控件有颜色，也有自己需要的颜色，那么就不必在子控件加背景颜色。
2. 如果每个子控件的颜色不太一样，而且可以完全覆盖父控件，那么就不需要在子控件上加背景颜色。
3. 尽量避免不必要的嵌套。

4. 能用 LinearLayout 和 FrameLayout，就不要用 RelativeLayout，因为 RelativeLayout 控件相对比较复杂，测试也相对耗时。
5. 使用 include 和 merge 增加复用，减少层级。

   针对嵌套布局，google 提出了 include、merge 和 ViewStub 。

   include 可以提高布局的复用性，大大方便开发，include 没有减少布局的嵌套，但是 include 和 merge 联手搭配，还是不错的。

   merge 的布局取决于父控件是哪个布局，使用 merge 相当于减少了自身的一层布局，直接采用父 include 的布局，当然直接在父布局里面使用意义不大，所以会和 include 配合使用，即增加了布局的复用性，也减少了一层布局嵌套。
6. ViewStub 按需加载，更加轻便。

   ViewStub 可以按需加载，在需要的时候 ViewStub 中的布局才加载到内存。对于一些进度条，提示信息等等少次数使用的功能，使用 ViewStub 很合适。
7. 复杂界面可选择 ConstraintLayout，可有效减少层级。

   ConstraintLayout 可以有效地解决布局嵌套过多的问题。ConstraintLayout 使用约束的方式来指定各个控件的位置和关系，有点类似于 RelativeLayout，但远比 RelativeLayout 要更强大。所以简单布局简单处理，复杂布局 ConstraintLayout 很好使。

绘制优化

平时感觉的卡顿问题最主要的原因之一是因为渲染性能，因为越来越复杂的界面交互，其中可能添加了动画、图片等等。希望创造出越炫的交互界面，同时也希望可以流畅显示，但是往往卡顿就会发生在这里。

这个是 Android 的渲染机制造成的，Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，但是渲染未必成功，如果成功了那么代表一切顺利，但是失败了可能就要延误时间，或者直接跳过去，给人视觉上的表现就是要么卡了一会，要么跳帧。

View 的绘制频率保证 60 fps 是最佳的，这就要求每帧绘制时间不超过 16ms (16ms = 1000 / 60) ，虽然程序很难保证 16 ms 这个时间，但是尽量降低 onDraw 方法中的复杂度总是切实有效的。

解决方法：
第一点：onDraw 方法中不要做耗时的任务，也不做过多的循环操作，特别是嵌套循环，虽然每次循环耗时很小，但是大量的循环势必霸占 CPU 的时间片，从而造成 View 的绘制过程不流畅。

第二点：除了循环之外，onDraw() 中不要创建新的布局对象，因为 onDraw() 方法一般都会频繁大量调用，就意味着会产生大量的临时对象，不仅占用内存，而且会导致系统更加频繁的 GC，大大降低程序的执行速度和效率。

关于绘制优化的方法

1. onDraw 中不要创建新的布局对象。
2. onDraw 方法中不要做耗时的任务，少使用循环。

检查 UI 优化的方法

对于 UI 性能的优化可以通过开发者选项中的 GPU 过度绘制工具来进行分析。在设置 -> 开发者选项 -> 调试 GPU 过度绘制中打开调试。

目标就是尽量减少红色 Overdraw，看到更多的蓝色区域。

查看每帧的渲染时长

可以打开设置 -> 开发者选项 -> GPU 呈现模式分析 -> 在屏幕上显示为条形图。

随着界面的刷新，界面上会以实时柱状图来显示每帧的渲染时间，柱状图越高表示渲染时间越长，每个柱状图偏上都有一根表示 16 ms 基准的绿色横线，每一条竖着的柱状线都包含三部分（蓝色代表测量绘制 Display List 的时间，红色代表 OpenGL 渲染 Display List 所需要的时间，黄色代表 CPU 等待 GPU 处理的时间），只要每一帧的总时间低于基准线就不会发生 UI 卡顿问题（个别超出基准线其实也不算什么问题的）。

也可以在执行完 UI 滑动操作后在命令行输入如下命令查看命令行打印的 GPU 渲染数据：

```
adb shell dumpsys gfxinfo [应用包名]
```

分析依据：Draw + Process + Execute = 完整的显示一帧时间 < 16ms。

内存泄漏

内存泄漏：程序不再使用的对象无法被 GC 识别，导致对象一直留在内存当中，占用了内存空间。

内存泄漏分为 4 种：

1. 集合类泄漏
2. 单例 / 静态变量造成的内存泄漏
3. 匿名内部类 / 非静态内部类
4. 资源未关闭造成的内存泄漏

集合类泄漏

集合类（List 等）添加元素后，仍引用着集合元素对象，导致该集合中的元素对象无法被回收，从而导致内存泄漏。

解决方法：使用 clear 清理集合，并释放其引用（list = null）。

单例 / 静态变量造成的内存泄漏

单例模式具有其静态特性，它的生命周期等于应用程序的生命周期，所以容易造成内存泄漏。

当一个生命周期长的对象持有了生命周期短的对象的引用，这就造成了内存泄漏。

Context 和 Toast 是容易出现内存泄漏的，如果生命周期长，可以使用 Application 的 Content。

解决方法：生命周期长的引用生命周期长的，不要引用生命周期短的。

匿名内部类 / 非静态内部类

如果非静态内部类的生命周期长于外部类，再加上自动持有外部类的强引用，就会发生内存泄漏。解决方法就是改为静态内部类。

匿名内部类也会持有外部类的强引用，解决方法也是改为静态内部类。

静态持有非静态类的引用，就会导致内存泄漏，解决方法是使用弱引用。

引用分为强引用、软引用、弱引用、虚引用，强度依次递减。

- 强引用：不做特殊处理的一般都是强引用，如果一个对象具有强引用，GC 绝不会回收它。
- 软引用（SoftReference）：如果内存空间足够，GC 就不会回收它，如果内存空间不足了，就会回收这些对象的内存。
- 弱引用（WeakReference）：弱引用要比软引用更弱一个级别，内存不够回收，GC 的时候不管内存够不够也回收。不过 GC 是一个优先级很低的线程，所以也不会随便就被回收了。
- 虚引用

资源未关闭造成的内存泄漏

资源为关系的情况有：

1. 网络、文件等流忘记关闭
2. 手动注册广播时，退出时忘记 unregisterReceiver()
3. Service 执行完后忘记 stopSelf()
4. EventBus 等观察者模式的框架忘记手动解除注册

检查内存泄漏的工具：1. leakcanary；2.Memory Monitor；3. Android Lint。

启动优化

安卓应用的启动方式分为三种：冷启动（Cold start）、暖启动（Warm start）、热启动（Hot start）。不同的启动方式决定了应用 UI 对用户可见所需要花费的时间长短。冷启动消耗的时间最长，基于冷启动方式的优化工作也是最考验产品用户体验的地方。

启动方式

冷启动

冷启动是指应用程序从头开始，系统没有获取到当前 app 的 activity、Service 等等。例如第一次启动 app 或者杀死进程后第一次启动。那么对比其他两种当时，冷启动是耗时最长的。

应用发生冷启动时，系统一定会执行下面的三个任务：

1. 开始加载并启动应用
2. 应用启动后，显示一个空白的启动窗口（启动闪屏页）
3. 创建应用程序进程

那么创建应用程序进程之后，开始创建应用程序对象：

4. 启动 UI 线程
5. 创建主 Activity
6. 加载布局
7. 屏幕布局
8. 执行初始化绘制

应用程序进程完成第一次绘制后，系统进程会交换当前显示的背景窗口，将其替换为主活动。此时，用户可以开始使用该应用进程，至此启动完成。

在冷启动过程中有两个 creation 工作，分别是 Application 和 Activity creation，它们均在 View 绘制展示之前。所以，在应用自定义的 Application 类和第一个 Activity 类中，onCreate() 方法做的事情越多，冷启动消耗的时间越长。

Application 创建

当 Application 启动时，空白的启动窗口将保留在屏幕上，直到系统首次完成绘制应用程序。此时，系统进程会交换应用程序的启动窗口，允许用户开始与应用程序进行交互。（为什么程序启动时会出现一段时间的黑屏或白屏）。

如果有自己的 Application，系统会调用 Application 对象的 onCreate() 方法，之后，应用程序会生成主线程（UI 线程），并通过创建主要活动来执行任务。

Activity 创建

应用程序进程创建 Activity 后，Activity 将执行以下操作：

1. 初始化值
2. 调用构造函数
3. 调用回调方法（例如 Activity.onCreate），对应 Activity 的当前生命周期状态。

通常，onCreate() 方法对加载时间的影响最大，因为它以最高的开销执行工作：加载和渲染视图以及初始化 Activity 运行所需的对象。

暖启动

当应用中的 Activities 被销毁，但在内存中常驻时，应用的启动方式就会变为暖启动。相比冷启动，暖启动过程减少了对象初始化、布局加载等工作，启动时间更短。但启动时，系统依然会展示闪屏页，直到第一个 Activity 的内容呈现为止。

暖启动的场景有：

1. 用户退出应用，但随后重新启动它。该过程可能已继续运行，但应用程序必须通过调用从头开始重新创建 Activity 的 onCreate()。
2. 系统将您的应用程序从内存中逐出，然后用户重新启动它。需要重新启动进程和活动，但是在调用 onCreate() 的时候可以从 Bundle(savedInstanceState) 获取数据。

热启动

相比暖启动，热启动时应用做的工作更少，启动时间更短。在一个热启动中，系统会把你的 Activity 带到前台，如果应用程序的 Activity 仍然驻留在内存中，那么引用程序可以避免重复对象初始化、布局加载和渲染。产生热启动的场景有：用户使用返回键退出应用，然后马上又重新启动应用。

热启动和冷启动一样系统进程显示空白屏幕，直到应用程序完成呈现活动。

如何优化

1. 利用提前展示出来的 Window，快速展示出来一个界面。

   使用 Activity 的 windowBackground 主题属性来为启动的 Activity 提供一个简单的 drawable。
2. 避免在启动时做密集沉重的初始化

对于一些必须去初始化的，可以使用异步加载。一些初始化的情况：a.比如像友盟，bugly 这样的业务非必要的可以异步加载；b.比如地图、推送等，非第一时间需要的可以在主线程做延时启动。当程序以及启动起来之后，再进行初始化。c.对于图片、网络请求框架必须在主线程初始化。

3. 避免 I/O 操作、反序列化、网络操作、布局嵌套等

网络优化

可以通过 Memory 下面的 Net 进行网络的监听。

线程优化

大多数就是因为线程阻塞导致的 ANR。

ANR：application not responding，应用程序无响应。

ANR 出现的时机：

1. activity 如果在 5s 内无响应事件（屏幕触摸事件或者键盘输入事件）；
2. BroadcastReceiver 如果在 10s 内无法处理完成；
3. Service 如果在 20s 内无法处理完成。

Android 系统提供了一些工具类来解决此问题：

- Asynctask：为 UI 线程与工作线程之间进行快速处理的切换提供一种简单边界的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的场景。
- HandlerThread：为某些回调方法或者等待某些执行任务设置一个专属的线程，并提供线程任务的调度机制。
- ThreadPool：把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。
- IntentService：适合执行由 UI 触发的后台任务。并可以把这些任务执行的情况进行一定的机制反馈给 UI。

一般多线程的情况可以通过 Asynctask 处理，也可以使用 annotation。

网络优化

网络优化的重点：时间、速度和成功率。

图片优化

图片现在在应用中很常见，所以对图片优化也是对网络优化的一种方式。

1. 使用 WebP 格式。同样的图片，采用 WebP 格式可大幅节省流量，相对于 JPG 格式的图片，流量能节省将近 25% 到 35%；相对于 PNG 格式的图片，流量可以节省将近 80%。最重要的是使用 WebP 之后图片质量也没有改变。
2. 使用缩略图。可以控制 inside 和 option，进行图片缩放。

网络请求处理的优化

1. 对服务器返回数据进行缓存，设定有效时间，有效时间之内不走网络请求，读取缓存显示，从而减少流量消耗。可以使用 HttpResponseCache 进行数据缓存。
2. 尽量少使用 GPS 定位，如果条件允许，尽可能使用网络定位。
3. 下载和上传尽可能使用断点。
4. 刷新数据时，尽可能使用局部刷新，而不是全局刷新。第一、界面会闪屏一下，网差的时候界面可能会直接白屏一段时间；第二，节省流量。

包体优化

apk文件：

AndroidManifest.xml：这个文件用来描述 Android 应用的配置信息，一些组件的注册信息、可使用权限等。

assets 文件夹：存放一些配置文件、资源文件，assets 不会自动生成对应的 ID，而是通过 AssetManager 类的接口获取。

classes.dex：Dalvik 字节码程序，让 Dalvik 虚拟机可执行，一般情况下，Android 应用在打包时通过 Android SDK 中的 dx 工具将 Java 字节码转换为 Dalvik 字节码。

META-INF：保存应用的签名信息，签名信息可以验证 APK 文件的完整性。

res 目录：res 是 resource 的缩写，这个目录存放资源文件，会自动生成对应的 ID 并映射到 .R 文件中，访问直接使用资源 ID。

resources.arsc：记录着资源文件和资源 ID 之间的映射关系，用来根据资源 ID 寻找资源。

包体优化的方式

从代码和资源两个方面去减少 APK 的大小。

1. 使用可绘制对象，某些图像不需要静态图片资源；框架可以在运行时动态绘制图像。Drawable 对象（< shape > 以 XML 格式）可以占用 APK 中的少量空间。此外，XML Drawbale 对象产生符合材料设计准则的单色图像。
2. 可以使用自己用 XML 写 Drawable，就使用自己写的，这种图片占用空间会很小。
3. 重用资源。可以使用旋转等方法去重用（比如三角按钮，三角朝上代表收起，三角朝下表示展开，就可以通过旋转重用一张图片）。

   统一图像的着色不同，可以使用 android:tint（将图片渲染成指定的颜色） 和 tintMode 属性，低版本（5.0 以下）可以使用 ColorFilter（图片减半处理，可以修改图片每个像素的颜色）。
4. 压缩 PNG 和 JPEG 文件。可以减少 PNG 文件的大小，而不会丢失图像质量。使用工具如 pngcrush、pngquant 或 zopfliping，这些工具都可以减少 PNG 文件的大小，同时保持感知的图像质量。
5. 使用 WebP 文件格式。可以使用 Webp 文件格式，而不是使用 PNG 或 JPEG 文件。Webp 格式提供有损压缩（如 JPEG）以及透明度（如 PNG），但可以提供比 JPEG 、PNG 更好的压缩。

   可以使用 Android Studio 将现有的 BMP、JPG、PNG 或静态 GIF 图像转换为 WebP 格式。
6. 使用矢量图像。可以使用矢量图形来创建与分辨率无关的图标和其他可伸缩 Image。使用这些图像可以大大减少 APK 大小。一个 100 字节的文件可以生成与屏幕大小相关的清晰图像。

   但是，系统渲染每个 VectorDrawable 对象需要花费大量时间，而较大的图像需要更长的时间才能显示在屏幕上。因此，仅在显示小图像时使用矢量图形。

   不要把 AnimationDrawable 用于创建逐帧动画，因为这样做需要为动画的每个帧包含一个单独的位图文件，这会大大增加 APK 的大小。
7. 代码混淆。使用 proGuard 代码混淆器工具，包括压缩、优化、混淆等功能。
8. 插件化。比如功能模块放在服务器上，按需下载，可以减少安装包大小。

检查包体的工具

lint 工具，没有使用过的资源就会打印信息

```
res/layout/preferences.xml: Warning: The resource R.layout.preferences appears
    to be unused [UnusedResources]
```

开启资源压缩，就会自动删除无用的代码

```
android {
    ...
    buildTypes {
        release {
            shrinkResources true
            minifyEnabled true
            proguardFiles getDefaultProguardFile('proguard-android.txt'),
                    'proguard-rules.pro'
        }
    }
}
```

2. Android Studio 有自带的代码检查工具。打开 Analyze -> Run Inspection by Name... -> unused resource 点击开始检测。
3. 也可以将鼠标放在代码区点击右键 -> Analyze -> Inspect Code -> 界面选择你要检测的模块 ->点击确认开始检测。

上面的方法是最容易找到代码缺陷以及无用代码的地方。

电量优化

Battery Historian 是由 Google 提供的 Android 系统电量分析工具，从手机中导出 bugreport 文件上传至页面，在网页中生成详细的图标数据来展示手机上各模块电量消耗过程，最后通过 App 数据的分析制定出相应的电量优化的方法。

谷歌推荐使用 JobScheduler 来调整任务优先级等策略来达到降低损耗的目的。JobScheduler 可以避免频繁的唤醒硬件模块，造成不必要的电量消耗。避免在不合适的时间（例如低电量情况下、弱网络或者移动网络情况下）执行过多的任务从而消耗电量。

具体实践

1. 可以退出非面向用户的任务（如定期数据库数据更新）；
2. 当充电时才希望执行的工作（如备份数据）；
3. 需要访问网络或 Wi-Fi 连接的任务（如向服务器拉取配置数据）；
4. 零散任务合并到一个批次取定期运行；
5. 当设备空闲时启动某些任务；
6. 只有当条件得到满足，系统才会启动计划中的任务（充电、WIFI）。
7. 需要进行网络请求时，首先判断网络当前的状态。
8. 在同时有 wifi 和移动数据的情况下，应该直接屏蔽移动数据的网络请求，只有当 wifi 断开时再调用，因为 wifi 请求的耗电量远比移动数据的耗电量低的低。
9. 后台任务要尽可能少的唤醒 CPU。

电量优化的法则

谷歌对电量优化提出了一个懒惰第一的法则：

减少：应用可以删除冗余操作吗？例如，是否可以缓存下载的数据而不是重复唤醒无线电以重新下载数据。

推迟：应用是否需要立即执行操作？例如，可以等到设备充电时把数据备份到云端。

合并：可以批处理工作，而不是多次将设备至于活动状态吗？例如，几十个应用程序是否真的有必要在不同时间打开收音机发送邮件，在依次唤醒收音机期间，是否可以传输消息。

电量优化的方法

1. 使用 JobScheduler 调度任务。
2. 使用懒惰法则

其他优化

ListView 优化

1. 复用 `public View getView(int position, View convertView, ViewGroup parent)` 的 convertView，不必每次创建新的 View。
2. 使用 ViewHolder，减少每次初始化子 View。
3. 分段、分页加载。

Bitmap 优化

Bitmap 优化就是压缩。

三种压缩方式：

1. 对图片质量进行压缩。
2. 对图片尺寸进行压缩。
3. 使用 libjpeg.so 库进行压缩。

线程有优化

线程优化的思想是采用线程池，避免在程序中存在大量的 Thread。线程池可以重用内部的线程，从而避免了线程的创建和销毁所带来的性能开销，同时线程池还能有效地控制线程池地最大并发数，避免大量地线程因互相抢占系统资源从而导致阻塞线程发生。

线程池的优点

1. 减少在创建和销毁线程上所花的时间以及系统资源的开销。
2. 如不使用线程池，有可能造成系统创建大量线程而导致消耗完系统内存以及 ” 过度切换 “。

注意

1. 如果线程池中的数量未达到核心线程的数量，则直接会启动一个核心线程来执行任务。
2. 如果线程池中的数量已经达到或超过核心线程的数量，则任务会被插入到任务队列中等待执行。
3. 如果 2 中的任务无法插入到任务队列中，等待任务队列已满，这时候如果线程数量未达到线程池规定最大值，则回启动一个非核心线程来执行任务。
4. 如果 3 中线程数量已经达到线程池最大值，则会拒绝执行此任务，ThreadPoolExecutor 会调用 RejectedExecutionHandler 的 rejectedExecution 方法通知调用者。

高效代码

编写高效代码的有个基本规则：1.不要做不需要做的工作；2.如果可以避免，请不要分配内存。

1. 避免创建不必要的对象。
2. 首选静态。如果不需要访问对象的字段，请使方法保持静态。调用速度将提高约 15% - 20%。
3. 对常量使用 static final，此优化仅适用于基本类型和 String 常量，而不适用于任意引用类型。
4. 使用增加的 for 循环语句。增强 for 循环（for-each）可用于实现 Iterator 接口和数组的集合。
5. 避免使用浮点数。根据经验，浮点数比 Android 设备上的整数慢约 2 倍。

Android 内存知识

Android 应用层是由 java 开发的，Android 的 davlik 虚拟机与 jvm 也类似，只不过它是基于寄存器的。

在 java 中，通过 new 为对象分配内存，所有对象在 java 堆内分配空间，而内存的释放是由垃圾收集器（GC）来回收的。

Java 采用了有向图的原理来判断对象是否可以被回收。Java 将引用关系考虑为图的有向边，有向边从应用者指向引用对象。线程对象可以作为有向图的起始顶点，该图就是从起始顶点（GC roots）开始的一棵树，根顶点可以到达的对象都是有效对象，GC 不会回收这些对象。如果某个对象（连通子图）与这个根顶点不可达，那么认为这个对象不再被引用，可以被 GC 回收。

Android 的内存管理机制

Android 系统的 Dalvik 虚拟机扮演了常规的内存垃圾自动回收的角色，Android 系统没有为内存提供交换区，它使用 paging 与 memory-mapping(mmaping) 的机制来管理内存。

共享内存

Android 系统通过下面几种方式来实现共享内存：

- Android 应用的进程都是从一个叫做 Zygote 的进程 fork 出来的。Zygote 进程在系统启动并且载入通用的 framework 的代码与资源之后开始启动。

  为了启动一个新的程序进程，系统会 fork Zygote 进程生成一个新的进程，然后在新的进程中加载并运行应用程序的代码。这使得大多数的 RAM pages 被用来分配给 framework 的代码，同时使得 RAM 资源能够在应用的所有进程之间进行共享。

- 大多数 static 的数据被 mmapped 到一个进程中。这不仅仅使得同样的数据能够在进程间进行共享，而且使得它能够在需要的时候被 paged out。

  常见的 static 数据包括 Dalvik Code、app resources、so 文件等。

- 大多数情况下，Android 通过显式的分配共享内存数据（例如 ashmem 或者 gralloc）来实现动态 RAM 区域能够在不同进程之间进行共享的机制。

  例如，Window Surface 在 App 与 Screen Compositor 之间使用共享的内存，Cursor Buffers 在 Content Provider 与 Clients 之间共享内存。

分配与回收内存

- 每一个进程的 Dalvik heap 都反映了使用内存的占用范围。这就是通常逻辑意义上提到的 Davlik Heap Size，它可以随着需要进行增长，但是增长行为会有一个系统为它设置的上限。
- 逻辑上讲的 Heap Size 和实际物理意义上使用的内存大小是不对等的，Proportional Set Size(PSS) 记录了应用程序自身占用以及和其他进程进行共享的内存。
- Android 系统并不会对 Heap 中空闲内存区域做碎片整理。

  系统仅仅在新的内存分配之前判断 Heap 的尾端剩余空间是否足够，如果空间不够会触发 gc 操作，从而腾出更多空闲的内存空间。

  在 Android 的高级系统版本里面针对 Heap 空间有一个 Generational Heap Memory 的模型，最近分配的对象会存放在 Young Generation 区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到 Old Generation，最后累积一定时间再移动到 Permanent Generation 区域。

  系统会根据内存中不同的内存数据类型分别执行不同的 gc 操作。例如，刚分配到 Young Generation 区域的对象通常更容易被销毁回收，同时在 Young Generation 区域的 gc 操作速度会比 Old Generation 区域的 gc 操作速度更快。

每个 Generation 的内存区域都有固定的大小，随着新的对象陆续被分配到此区域，当这些对象总的大小快达到这一级别内存区域的阈值时，会触发 GC 的操作，以便腾出空间来存放其他新的对象。

通常情况下，GC 发生的时候，所有的线程都是会被暂停的。执行 GC 所占用的时间和它发生在哪一个 Generation 也有关系，Young Generation 中的每次 GC 操作时间是最短的，Old Generation 其次，Permanent Generation 最长。执行时间的长短也和当前 Generation 中的对象数量有关。

限制应用的内存

- 为了整个 Android 系统的内存控制需要，Android 系统为每一个应用程序都设置了一个硬性的 Dalvik Heap Size 最大限制阈值，这个阈值在不同的设备上会因为 RAM 大小不同而各有差异。

  如果应用占用内存空间已经接近这个阈值，此时再尝试分配内存的话，很容易引起 OutOfMemoryError 的错误。
- ActivityManager.getMemoryClass() 可以用来查询当前应用的 Heap Size 阈值，这个方法会返回一个整数，表明应用的 Heap Size 阈值是多少 Mb(megabates)。

应用切换操作

- Android 系统并不会在用户切换应用的时候做交换内存的操作。

  Android 会把那些不包含 Foreground 组件的应用进程放到 LRU Cache 中。

  例如，当用户开始开启了一个应用，系统会为它创建一个进程，但是当用户离开这个应用，此进程并不会立即被销毁，而是会被放到系统的 Cache 当中，如果用户后台再切换回到这个应用，此进程就能够被马上完整的恢复，从而实现应用的快速切换。
- 如果应用中有一个被缓存的进程，这个进程会占用一定的内存空间，它会对系统的整体性能有影响。

  因此当系统开始进入 Low Memory 的状态时，它会由系统根据 LRU 的规则、应用的优先级、内存占用情况以及其他因素的影响综合评估之后决定是否被杀死。

什么是内存抖动

内存抖动是指在短时间内有大量的对象被创建或者被回收的现象。

内存抖动出现原因主要是频繁创建对象，导致大量对象在短时间内被创建，由于新对象要占用内存空间而且很频繁。

如果一次或者两次创建对象对内存影响不大，不会造成严重内存抖动，这样可以接受也不可避免，频繁的话就会内存抖动严重。

内存抖动的影响是：如果抖动很频繁，会导致垃圾回收机制频繁运行，短时间内产生大量对象，需要大量内存，而且还是频繁抖动，就可能会需要回收内存以用于产生对象，垃圾回收机制就自然会频繁运行了。

所以频繁内存抖动会导致垃圾回收频繁运行。

* 内存泄漏

* 内存溢出

布局优化之 include

概述

include 标签常用于将布局中的公共部分提取出来供其他 layout 共用，以实现布局模块化，也是平常设计布局时用的最多的。

include 就是为了解决重复定义相同布局的问题。

include 使用注意

1. 一个 xml 布局文件有多个 include 标签需要设置 ID，才能找到相应子 View 的控件，否则只能找到第一个 include 的 layout 布局，以及该布局的控件。
2. include 标签如果使用 layout_xx 属性，会覆盖被 include 的 xml 文件根节点对应的 layout_xx 属性，建议在 include 标签调用的布局设置好宽高位置，防止不必要的 bug。
3. 如果要在 include 标签下使用 RelativeLayout，如 layout_margin 等其他属性，记得要同时设置 layout_margin 等其他属性，记得要同时设置 layout_width 和 layout_height，不然其它属性会没反应。
4. include 添加 id，会覆盖被 include 的 xml 文件根节点 id，建议 include 和被 include 覆盖的 xml 文件根节点设置同名的 id，不然有可能会报空指针异常。
5. 如果 include 中设置了 id，那么就通过 include 的 id 来查找被 include 布局根元素的 View；如果 include 中没有设置 id，而被 include 的布局的根元素设置了 id，那么通过该根元素的 id 来查找该 view 即可。拿到根元素后查找其子控件都是一样的。

源码分析

为什么使用根元素的 id 查找 view 就会报空指针呢？分析一下源码来查看。对于布局文件的解析，最终都会调用 LayoutInflater 的 inflater 方法。

1. 在 inflate() 方法中调用了 rInflateChildren() 方法去加载布局的子 View。
2. rInflateChildren 就是调用 rInflate() 方法。
3. rInflate() 方法其实就是遍历 xml 中的所有元素，然后挨个进行解析。

   例如解析到一个标签，那么就根据用户设置的一些 layout_width、layout_height、id 等属性来构造一个 TextView 对象，然后添加到父控件（ViewGroup 类型）中。

   include 标签也是一样的，当遇到 include 标签时，就调用了 parseInclude() 方法，就会对 include 标签进行解析。
4. 整个过程就是根据不同的标签解析不同的元素，首先会解析 include 元素，然后再解析被 include 进来的布局的 root view 元素。然后再解析 root view 下面的所有元素，这个过程是从上面注释的 2-4 的过程，然后是设置布局参数。
5. 注视 5 处解释了为什么 include 标签和被引入的布局的根元素都设置了 id 的情况下，通过被引入的根元素的 id 来查找子控件会找不到的情况。可以看到，注释 5 处会判断 include 标签的 id 如果不是 View.NO_ID 的话会把该 id 设置给被引入的布局根元素的 id，因此再通过引入的根元素的 id 来查找根元素就会找不到了。

所以结论就是：如果 include 中设置了 id，那么就通过 include 的 id 来查找被 include 布局根元素的 View；如果 include 中没有设置 id，而被 include 的布局的根元素设置了 id，那么通过该根元素的 id 来查找该 View 即可。拿到根元素后查找其子控件都是一样的。

布局优化之 merge

概述

merge 标签主要用于辅助 include 标签，在使用 include 后可能导致布局嵌套过多，多余的 layout 节点或导致解析变慢（可通过 hierarchy viewer 工具查看布局的嵌套情况）。

官方文档说明：merge 用于视图层级结构中的冗余视图，例如根布局是 LinearLayout，那么又 include 一个 LinearLayout 布局就没意义了，反而会减慢 UI 加载速度。

merge 就是为了减少在使用 include 布局文件时的层级。

merge 标签常用场景

1. 根布局是 FrameLayout 且不需要设置 background 或 padding 等属性，可以用 merge 代替，因为 Activity 的 ContentView 父元素就是 FrameLayout，所以可以用 merge 消除只剩一个。
2. 某布局作为子布局被其他布局 include 时，使用 merge 当做该布局的顶节点，这样在被引入时顶节点会自动被忽略，而将其子节点全部合并到主布局中。
3. 自定义 View 如果继承 LinearLayout(ViewGroup)，建议让自定义 View 的布局文件根布局设置成 merge，这样能少一层节点。

merge 使用注意

1. 因为 merge 标签并不是 View，所以在通过 LayoutInflate.inflate() 方法渲染的时候，第二个参数必须指定一个父容器，且第三个参数必须为 true，也就是必须为 merge 下的视图指定一个父亲节点。
2. 因为 merge 不是 View，所以对 merge 标签设置的所有属性都是无效的。
3. 注意如果 include 的 layout 用了 merge，调用 include 的根布局也使用了 merge 标签，那么就失去布局的属性了。
4. merge 标签必须使用在根布局。
5. ViewStub 标签中的 layout 布局不能使用 merge 标签。

merge 源码分析

相关的代码还是从 LayoutInflate 的 inflate() 函数中开始，从 include 的源码分析贴出的 inflate() 代码可以看到，如果标签是 merge 的话就会调用 rInflate() 方法解析子 view。

如果是 merge 标签，那么直接将其中的子元素添加到 merge 标签的 parent 中，这样就保证了不会引入额外的层级。

布局优化之 ViewStub

概述

ViewStub 标签最大的优点是当需要时才会加载，所以使用它并不会影响 UI 初始化的性能。各种不常用的布局像加载条、显示错误信息等可以使用 ViewStub 标签，以减少内存使用量，加快渲染速度。

ViewStub 是一个不可见的，实际上是把宽高设置为 0 的 View。效果有点类似普通的 view.setVisible()，但性能体验提高不少。

ViewStub 默认是不可见的，只有通过调用 setVisibility 函数或者 Inflate 函数才会将其要装载的目标布局给加载出来，从而达到延迟加载的效果，这个要被加载的布局通过 android:layout 属性来设置。

第一次初始化时，初始化的是 ViewStub View，当调用 inflate() 或 setVisibility() 后会被 remove 掉，然后再将其中的 layout 加到当前 view hierarchy 中。

判断 ViewStub 是否已经加载过

1. 如果通过 setVisibility 来加载，那么通过判断可见性即可。
2. 如果通过 inflate() 来加载，判断 ViewStub 是否为 null 来判断。

ViewStub 标签使用注意

1. ViewStub 标签不支持 merge 标签
2. ViewStub 的 inflate 只能被调用一次，第二次调用会抛出异常，setVisibility 可以被调用多次，但不建议这么做（ViewStub 调用过后，可能给 GC 掉，再调用 setVisiblity() 会报异常）。
3. 为 ViewStub 赋值的 android:layout_xx 属性会替换待加载布局文件的根节点对应的属性。
4. 判断是否已经加载过，如果通过 setVisibility 来加载，那么通过判断可见性即可；如果通过 inflate() 来加载是不可以通过判断可见性来处理的，而需要判断 view 是否为 null 来进行判断。
5. findViewById 的问题，注意 ViewStub 中是否设置了 inflatedId，如果设置了则需要通过 inflateId 来查找目标布局的根元素。

ViewStub 源码分析

可以看出，其实最终加载目标布局的还是 inflate() 函数，在该函数中将加载目标布局，获取到根元素后，如果 mInflatedId 不为 NO_ID 则把 mInflatedId 设置为根元素的 id，这也是为什么在获取根元素时会使用 ViewStub 的 inflatedId。如果没有设置 inflatedId 的话可以通过根元素的 id 来获取。

然后将 ViewStub 从 parent 中移除，将目标布局的根元素添加到 parent 中。最后会把目标布局的根元素返回。因此在调用 inflate() 函数时可以直接获得根元素，省掉了 findViewById 的过程。

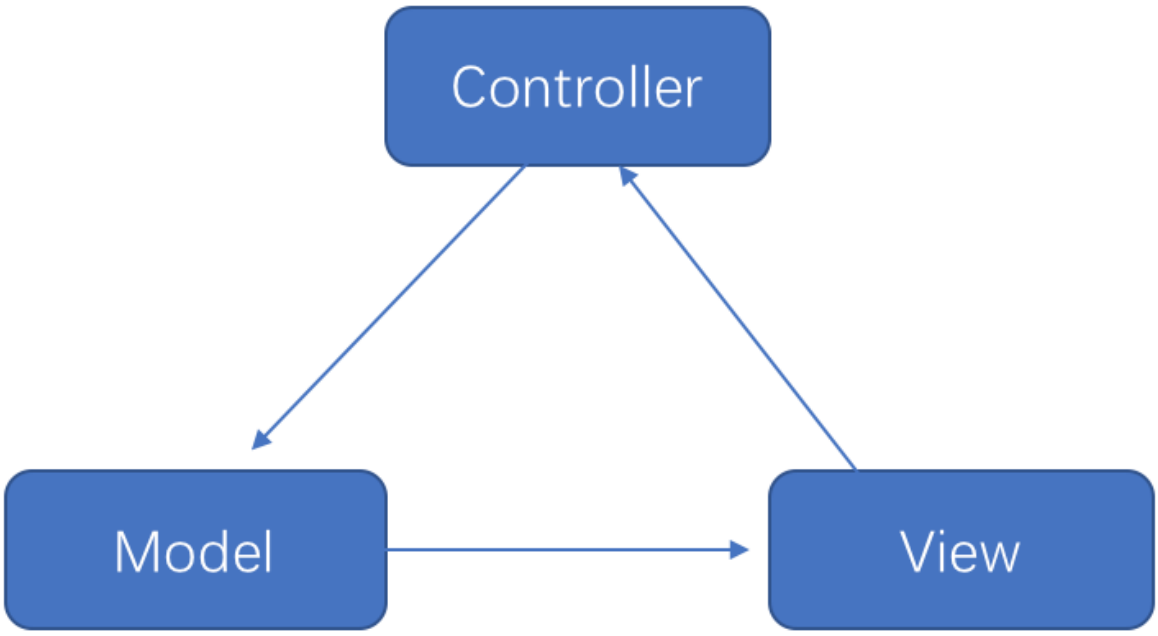
Space 组件

在 ConstaintLayout 出来前，写布局会使用大量的 margin 或 padding，但是这种方式可读性会很差，加一个布局嵌套又会损耗性能，鉴于这种情况，可以使用 space，使用方式和 View 一样，不过主要用来占位置，不会有任何显示效果。

架构与设计模式

设计模式选择

MVC



视图层（View）对应于 xml 布局文件和 java 代码动态 view 部分。

控制层（Controller）MVC 中 Android 的控制层是由 Activity 来承担的，Activity 本来主要是作为初始化页面，展示数据的操作，但是因为 XML 视图功能太弱，所以 Activity 既要负责视图的显示又要加入控制逻辑，承担的功能太多。

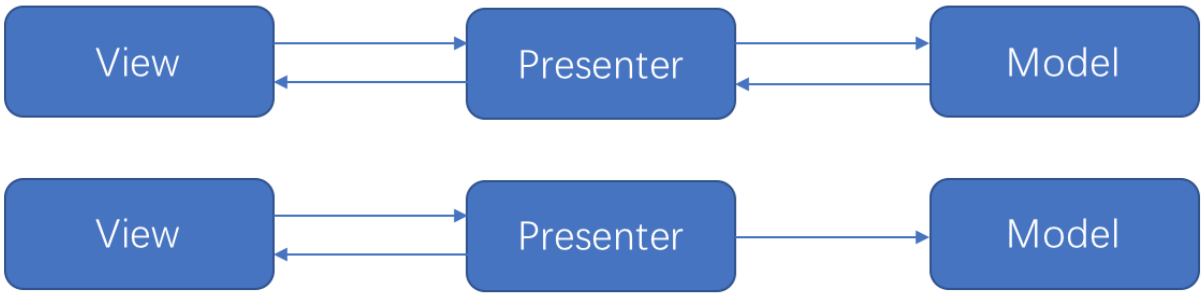
模型层（Model）针对业务模型，建立的数据结构和相关的类，它主要负责网络请求、数据库处理、I/O 操作。

- 总结
    - 具有一定的分层，mode 彻底解耦，controller 和 view 并没有解耦。

- 层与层之间的交互尽量使用回调或者使用消息机制去完成，尽量避免直接持有。
- controller 和 view 在 android 中无法做到彻底分离，但在代码逻辑层面一定要分清。
- 业务逻辑被放置在 mode 层，能够更好的复用和修改增加业务。

MVP

MVP 和 MVC 很像，MVP 也是三层，唯一的差别是 Mode 和 View 之间不进行通讯，都是通过 Presenter 完成。MVC 有一个缺点就是在 Android 中由于 Activity 的存在，Controller 和 View 很难做到完全解耦，但在 MVP 中就可以很好的解决这个问题。



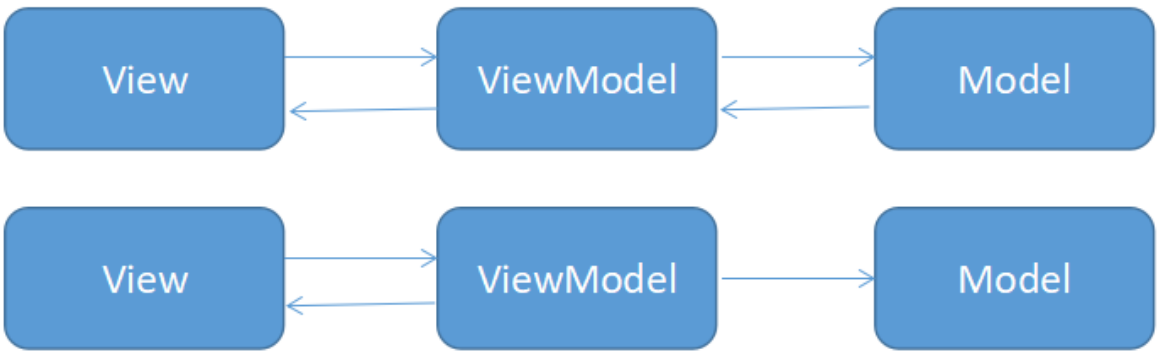
MVP 中也有一个 Contract 类，Contract 在 MVP 中是一个契约类，契约类用于定义同一个界面的 view 接口和 presenter 的具体实现，好处是通过规范的方法命名和注释可以清晰的看到整个页面的逻辑。

- 总结

  通过引入接口 BaseView，让相应的视图组件如 Activity、Fragment 去实现 BaseView，实现了视图层的独立，通过中间层 Preseter 实现了 Model 和 View 的完全解耦。MVP 彻底解决了 MVC 中 View 和 Controller 傻傻分不清楚的问题，但是随着业务逻辑的增加，一个页面可能会非常复杂，UI 的改变是非常多的，会有非常多的 case，这样就会造成 View 接口会很庞大。

MVVM

MVP 会随着业务逻辑的增加、UI 的改变多的情况下，会有非常多的跟 UI 相关的 case，这样就会造成 View 的接口会很庞大。而 MVVM 就解决了这个问题，通过双向绑定的机制，实现数据和 UI 内容，只要想改其中一方，另一方都能及时更新的一种设计理念，这样就省去了很多在 View 层中写很多 case 的情况，只需要改变数据就行。MVVM 的设计图：



一般情况就这两种情况，看起来跟 MVP 好像没什么差别，其实区别还是挺大的，在 MVP 中 View 和 Presenter 要相互持有，方便调用对方，而在 MVP 中 View 和 ViewModel 通过 Binding 进行关联，他们之间的关联处理通过 DataBinding 完成。

- 总结

  看起来 MVVM 很好的解决了 MVC 和 MVP 的不足，但是由于数据和视图的双向绑定，导致出现问题时不太好定位来源，有可能数据问题导致，也有可能业务逻辑中对视图属性的修改导致。如果项目中打算用 MVVM 的话可以考虑使用官方的架构组件 ViewModel、LiveData、DataBinding 去实现 MVVM 。

关于 MVC、MVP、MVVM 如何选择

在 MVP 中要实现根据业务逻辑和页面逻辑做很多 Present 和 View 的具体实现，如果这些 case 太多，会导致代码的可读性变差，但是通过引入 contract 契约类，会让业务逻辑变得清晰许多。因此不管是用哪种设计模式，只要运用得当，都可以达到想要的结果。

简单建议：

1. 如果项目简单，没什么复杂性，未来改动也不大的话，那就不要用设计模式或者架构方法，只需要将每个模块封装好，方便调用即可，不要为了使用设计模式或架构方法而使用。
2. 对于偏向展示型的 app，绝大多数业务逻辑都在后端，app 主要功能就是展示数据、交互等，建议使用 mvvm。
3. 对于工具类或者需要写很多业务逻辑 app，使用 mvp 或者 mvvm 都可。

*** MVI**

MVI：Model-View-Intent

MVI 在隔离 View 和 Model 的基础上，实现了响应式编程。

MVVM 的核心是在 MVC 的思想上实现了数据驱动 UI。

通过 ViewModel 将数据（Model）和UI（View）隔离，再通过 LiveData 将数据和 UI 的绑定，实现数据驱动UI，只需要 liveData 的数据修改 UI 能自动响应更新。

MVI 与 MVVM 很相似，其借鉴了前端框架的思想，更加强调数据的单向流动和唯一数据源。

其主要分为以下几部分：

\1. Model: 与MVVM中的Model不同的是，MVI的Model主要指UI状态（State）。例如页面加载状态、控件位置等都是一种UI状态。

\2. View: 与其他MVX中的View一致，可能是一个Activity或者任意UI承载单元。MVI中的View通过订阅Intent的变化实现界面刷新。（注意：这里不是Activity的Intent）

\3. Intent: 此Intent不是Activity的Intent，用户的任何操作都被包装成Intent后发送给Model层进行数据请求。

MVI强调数据的单向流动，主要分为以下几步：

\1. 用户操作以Intent的形式通知Model。

\2. Model基于Intent更新State。

\3. View接收到State变化刷新UI。

IPC

Android 中的多进程模式

通过给四大组件指定 android:process 属性，就可以轻易地开启多进程模式。

但是有时候通过多进程得到的好处甚至都不足以弥补使用多进程所带来的代码层面的负面影响。

开启多进程模式

首先，在 Android 中使用多进程只有一种方法，那就是给四大组件（Activity、Service、Receiver、ContentProvider）在 AndroidMenifest 中指定 android:process 属性，除此之外没有其他办法，也就是说我们无法给一个线程或者一个实体类指定其运行时所在的进程。

其实还有另一种非常规的多进程方法，那就是通过 JNI 在 native 层去 fork 一个新的进程，但是这种方法属于特殊情况，也不是常用的创建多进程的方式。

下面是一个示例，描述了如何在 Android 中创建多进程：

```
< activity

        android:name="com.ryg.chapter_2.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_2.SecondActivity"

        android:configChanges="screenLayout"

        android:label="@string/app_name"

        android:process=":remote" />
```

```
    <activity
      android:name="com.ryg.chapter_2.ThirdActivity"
      android:configChanges="screenLayout"
      android:label="@string/app_name"
      android:process="com.ryg.chapter_2.remote" />
```

< activity

android:name="com.ryg.chapter_2.MainActivity"

android:configChanges="orientation|screenSize"

android:label="@string/app_name"

android:launchMode="standard" >

</activity

<activity

android:name="com.ryg.chapter_2.SecondActivity"

android:configChanges="screenLayout"

android:label="@string/app_name"

android:process=":remote" />

<activity

android:name="com.ryg.chapter_2.ThirdActivity"

android:configChanges="screenLayout"

android:label="@string/app_name"

android:process="com.ryg.chapter_2.remote" />

默认进程的进程名是包名。

除了在 Eclipse 的 DDMS 视图中查看进程信息，还可以用 shell 来查看，命令为：adb shell ps 或者 adb shell ps | grep com.ryg.chapter_2。其中 com.ryg.chapter_2 是包名，通过 ps 命令也可以查看一个包名中当前所存在的进程信息。

进程分别为 ":remote" 和 "com.ryg.chapter_2.remote" ，那么这两种方式是有区别的，区别有两方面：

1. 首先，":" 的含义是指要在当前的进程名前面附加上当前的包名，这是一种简写的方法。而 "com.ryg.chapter_2.remote" 是一种完整的命名方式，不会附加包名信息。

2. 其次，进程名以 ":" 开头的进程属于当前应用的私有进程，其他应用的组件不可以和它跑在同一个进程中，而进程名不以 ":" 开头的进程属于全局进程，其他应用通过 ShareUID 方式可以和它跑在同一个进程中。

Android 系统会为每个应用分配一个唯一的 UID，具有相同 UID 的应用才能共享数据。这里要说明的是，两个应用通过 ShareUID 跑在同一个进程中是有要求的，需要这两个应用有相同的 ShareUID 并且签名相同才可以。在这种情况下，它们可以互相访问对方的私有数据，比如 data 目录、组件信息等，不管它们是否跑在同一个进程中。当然如果它们跑在同一个进程中，那么除了能共享 data 目录、组件信息，还可以共享内存数据，或者说它们看起来就像是一个应用的两个部分。

多进程模式的运行机制

Android 为每一个应用分配了一个独立的虚拟机，或者说为每个进程都分配一个独立的虚拟机，不同的虚拟机在内存分配上有不同的地址空间，这就导致在不同的虚拟机中访问同一个类的对象会产生多份副本。

所有运行在不同进程中的四大组件，只要它们之间需要通过内存来共享数据，都会共享失败，这也是多进程所带来的主要影响。正常情况下，四大组件中间不可能不通过一些中间层来共享数据，那么通过简单地指定进程名来开启多进程都会无法正确运行。当然，特殊情况下，某些组件之间不需要共享数据，这个时候可以直接指定 android:process 属性来开启多进程，但是这种场景是不常见的，几乎所有情况都需要共享数据。

一般来说，使用多进程会造成如下几方面的问题：

1. 静态成员和单例模式完全失效。

2. 线程同步机制完全失效。

3. SharedPreferences 的可靠性下降。

4. Application 会多次创建。

SharedPreferences 的可靠性下降是因为 SharedPreferences 不支持两个进程同时去执行写操作，否则会导致一定几率的数据丢失，这是因为SharedPreferences 底层是通过读/写 XML 文件来实现的，并发写显然是可能出问题的，甚至并发读/写都有可能出问题。

Application 会多次创建问题也是显而易见的，当一个组件跑在一个新的进程中的时候，由于系统要在创建新的进程同时分配独立的虚拟机，所以这个过程其实就是启动一个应用的过程。因此，相当于系统又把这个应用重新启动了一遍，既然重新启动了，那么自然会创建新的 Application。这个问题其实可以这么理解，运行在同一个进程中的组件是属于同一个虚拟机和同一个 Application 的，同理，运行在不同进程中的组件是属于两个不同的虚拟机和 Application 的。

既然 Parcelable 和 Serializable 都能实现序列化并且都可用于Intent间的数据传递，那么二者该如何选取呢?

1. Serializable 是 Java 中的序列化接口，其使用起来简单但是开销很大，序列化和反序列化过程需要大量 I/O 操作。

2. 而 Parcelable 是 Android 中的序列化方式，因此更适合用在 Android 平台上，它的缺点就是使用起来稍微麻烦点，但是它的效率很高，这是 Android 推荐的序列化方式，因此要首选 Parcelable。

3. Parcelable 主要用在内存序列化上，通过 Parcelable 将对象序列化到存储设备中或者将对象序列化后通过网络传输也都是可以的，但是这个过程会稍显复杂，因此在这两种情况下建议大家使用 Serializable。

以上就是 Parcelable 和 Serializable 的区别。

Android 中的 IPC 方式

1. 使用 Bundle

四大组件中的三大组件（Activity、Service、Receiver）都是支持在 Intent 中传递 Bundle 数据的，由于 Bundle 实现了 Parcelable 接口，所以它可以方便地在不同的进程间传输。基于这一点，当在一个进程中启动了另一个进程的 Activity、Service 和 Receiver，就可以在 Bundle 中附加需要传输给远程进程的信息并通过 Intent 发送出去。当然，传输的数据必须能够被序列化，比如基本类型、实现了 Parcellable 接口的对象、实现了 Serializable 接口的对象以及一些 Android 支持的特殊对象，具体内容可以看 Bundle 这个类，就可以看到所有它支持的类型。

Bundle 不支持的类型无法通过它在进程间传递数据。

这是一种最简单的进程间通信方式。

2. 使用文件共享

共享文件也是一种不错的进程间通信方式，两个进程通过读/写同一个文件来交换数据。

在 Windows 上，一个文件如果被加了排斥锁将会导致其他线程无法对其进行访问，包括读和写，而由于 Android 系统基于 Linux，使得其并发读/写文件可以没有限制地进行，甚至两个线程同时对同一个文件进行写操作都是允许的，尽管这可能出问题。

通过文件交换数据很好使用，除了可以交换一些文本信息外，还可以序列化一个对象到文件系统中的同时从另一个进程中恢复这个对象。

通过文件共享这种方式来共享数据对文件格式是没有具体要求的，比如可以是文本文件，也可以是 XML 文件，只要读/写双方约定数据格式即可。

通过文件共享的方式也是有局限性的，比如并发读/写的问题，如果并发读/写，那么读出的内容就有可能不是最新的，如果是并发写的话那就更严重了。

因此要尽量避免并发写这种情况的发生或者考虑使用线程同步来限制多个线程的写操作。

文件共享方式适合在对数据同步要求不高的进程之间进行通信，并且要妥善处理并发读/写的问题。

当然，SharedPreferences 是个特例，众所周知，SharedPreferences 是 Android 中提供的轻量级存储方案，它通过键值对的方式来存储数据，在底层实现上它采用 XML 文件来存储键值对，每个应用的 SharedPreferences 文件都可以在当前包所在的 data目录 下查看到。一般来说，它的目录位于 /data/data/package name/shared_prefs 目录下，其中 package name 表示的是当前应用的包名。

从本质上来说，SharedPreferences 也属于文件的一种，但是由于系统对它的读/写有一定的缓存策略，即在内存中会有一份 SharedPreferences 文件的缓存，因此在多进程模式下，系统对它的读/写就变得不可靠，当面对高并发的读/写访问，Sharedpreferences 有很大几率会丢失数据，因此，不建议在进程间通信中使用 SharedPreferences。

3. 使用 Messenger

Messenger 可以翻译为信使，顾名思义，通过它可以在不同进程中传递 Message 对象，在 Message 中放入需要传递的数据，就可以轻松地实现数据的进程间传递了。

Messenger 是一种轻量级的 IPC 方案，它的底层实现是 AIDL。

Messenger 的使用方法很简单，它对 AIDL 做了封装，使得可以更简便地进行进程间通信。同时，由于它一次处理一个请求，因此在服务端不用考虑线程同步的问题，这是因为服务端中不存在并发执行的情形。

Message 中所支持的数据类型就是 Messenger 所支持的传输类型。

实际上，通过 Messenger 来传输 Message, Message 中能使用的载体只有 what、arg1、arg2、Bundle 以及replyTo。Message 中的另一个字段 object 在同一个进程中是很实用的，但是在进程间通信的时候，在 Android 2.2 以前 object 字段不支持跨进程传输，即便是 2.2 以后，也仅仅是系统提供的实现了 Parcelable 接口的对象才能通过它来传输。这就意味着自定义的 Parcelable 对象是无法通过 object 字段来传输的。

非系统的 Parcelable 对象的确无法通过 object 字段来传输，这也导致了 object 字段的实用性大大降低，所幸还有 Bundle, Bundle 中可以支持大量的数据类型。

4. 使用 AIDL

Messenger 是以串行的方式处理客户端发来的消息，如果大量的消息同时发送到服务端，服务端仍然只能一个个处理，如果有大量的并发请求，那么用 Messenger 就不太合适了。

Messenger 的作用主要是为了传递消息，很多时候可能需要跨进程调用服务端的方法，这种情形用 Messenger 就无法做到了，但是可以使用 AIDL 来实现跨进程的方法调用。

AIDL 也是 Messenger 的底层实现，因此 Messenger 本质上也是 AIDL，只不过系统做了封装从而方便上层的调用而已。

AIDL 文件支持哪些数据类型呢？如下所示。·

- 基本数据类型（ int、long、char、boolean、double 等）；
- String 和 CharSequence；·
- List：只支持 ArrayList，里面每个元素都必须能够被 AIDL 支持；
- Map：只支持 HashMap，里面的每个元素都必须被 AIDL 支持，包括 key 和 value；
- Parcelable：所有实现了 Parcelable 接口的对象；·
- AIDL：所有的 AIDL 接口本身也可以在 AIDL 文件中使用。

以上 6 种数据类型就是 AIDL 所支持的所有类型，其中自定义的 Parcelable 对象和 AIDL 对象必须要显式 import 进来，不管它们是否和当前的 AIDL 文件位于同一个包内。

如果 AIDL 文件中用到了自定义的 Parcelable 对象，那么必须新建一个和它同名的 AIDL 文件，并在其中声明它为 Parcelable 类型。

除此之外，AIDL 中除了基本数据类型，其他类型的参数必须标上方向：in、out 或者 inout, in 表示输入型参数，out 表示输出型参数，inout 表示输入输出型参数，

要根据实际需要去指定参数类型，不能一概使用 out 或者 inout，因为这在底层实现是有开销的。

AIDL 接口中只支持方法，不支持声明静态常量，这一点区别于传统的接口。

为了方便 AIDL 的开发，建议把所有和 AIDL 相关的类和文件全部放入同一个包中，这样做的好处是，当客户端是另外一个应用时，可以直接把整个包复制到客户端工程中。

AIDL 的包结构在服务端和客户端要保持一致，否则运行会出错，这是因为客户端需要反序列化服务端中和 AIDL 接口相关的所有类，如果类的完整路径不一样的话，就无法成功反序列化，程序也就无法正常运行。

Binder 会把客户端传递过来的对象重新转化并生成一个新的对象。

5. 使用 ContentProvider

ContentProvider 是 Android 中提供的专门用于不同应用间进行数据共享的方式，从这一点来看，它天生就适合进程间通信。

和 Messenger 一样，ContentProvider 的底层实现同样也是 Binder，由此可见，Binder 在 Android 系统中是何等的重要。

虽然 ContentProvider 的底层实现是 Binder，但是它的使用过程要比 AIDL 简单许多，这是因为系统已经做了封装，使得无须关心底层细节即可轻松实现 IPC。

创建一个自定义的 ContentProvider 很简单，只需要继承 ContentProvider 类并实现六个抽象方法即可：onCreate、query、update、insert、delete和getType。这六个抽象方法都很好理解：

1. onCreate 代表 ContentProvider 的创建，一般来说需要做一些初始化工作；

2. getType 用来返回一个 Uri 请求所对应的 MIME 类型（媒体类型），比如图片、视频等，这个媒体类型还是有点复杂的，如果应用不关注这个选项，可以直接在这个方法中返回 null 或者 “/”；

3. 剩下的四个方法对应于 CRUD 操作，即实现对数据表的增删改查功能。

通过 ContentResolver 的 notifyChange 方法来通知外界当前 ContentProvider 中的数据已经发生改变。要观察一个 ContentProvider 中的数据改变情况，可以通过 ContentResolver 的 registerContentObserver 方法来注册观察者，通过 unregisterContentObserver 方法来解除观察者。

6. 使用 Socket

Socket 也称为“套接字”，是网络通信中的概念，它分为流式套接字和用户数据报套接字两种，分别对应于网络的传输控制层中的 TCP 和 UDP 协议。

TCP 协议是面向连接的协议，提供稳定的双向通信功能，TCP 连接的建立需要经过 “三次握手” 才能完成，为了提供稳定的数据传输功能，其本身提供了超时重传机制，因此具有很高的稳定性。

而 UDP 是无连接的，提供不稳定的单向通信功能，当然 UDP 也可以实现双向通信功能。

在性能上，UDP 具有更好的效率，其缺点是不保证数据一定能够正确传输，尤其是在网络拥塞的情况下。

Socket 本身可以支持传输任意字节流。

Binder连接池

使用 AIDL 的流程：首先创建一个 Service 和一个 AIDL 接口，接着创建一个类继承自 AIDL 接口中的 Stub 类并实现 Stub 中的抽象方法，在 Service 的 onBind 方法中返回这个类的对象，然后客户端就可以绑定服务端 Service，建立连接后就可以访问远程服务端的方法了。

多个不同的业务模块都需要使用 AIDL 来进行进程间痛惜，为了减少 Service 的数量，将所有的 AIDL 放在同一个 Service 中去管理。

在这种模式下，整个工作机制是这样的：每个业务模块创建自己的 AIDL 接口并实现此接口，这个时候不同业务模块之间是不能有耦合的，所有实现细节要单独开来，然后向服务端提供自己的唯一标识和其对应的 Binder 对象；对于服务端来说，只需要一个 Service 就可以了，服务端提供一个 queryBinder 接口，这个接口能够根据业务模块的特征来返回相应的 Binder 对象给它们，不同的业务模块拿到所需的 Binder 对象后就可以进行远程方法调用了。

由此可见，Binder 连接池的主要作用就是将每个业务模块的 Binder 请求统一转发到远程 Service 中去执行，从而避免了重复创建 Service 的过程，

选用合适的 IPC 方式

通过下表，可以明确地看出不同 IPC 方式的优缺点和适用场景，那么在实际的开发中，只要选择合适的 IPC 方式就可以轻松完成多进程的开发场景。

名　　称	优　　点	缺　　点	适 用 场 景
Bundle	简单易用	只能传输 Bundle 支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发场景，并且无法做到进程间的即时通信	无并发访问情形，交换简单的数据实时性不高的场景
AIDL	功能强大，支持一对多并发通信，支持实时通信	使用稍复杂，需要处理好线程同步	一对多通信且有 RPC 需求
Messenger	功能一般，支持一对多串行通信，支持实时通信	不能很好处理高并发情形，不支持 RPC，数据通过 Message 进行传输，因此只能传输 Bundle 支持的数据类型	低并发的一对多即时通信，无 RPC 需求，或者无须要返回结果的 RPC 需求
ContentProvider	在数据源访问方面功能强大，支持一对多并发数据共享，可通过 Call 方法扩展其他操作	可以理解为受约束的 AIDL，主要提供数据源的 CRUD 操作	一对多的进程间的数据共享
Socket	功能强大，可以通过网络传输字节流，支持一对多并发实时通信	实现细节稍微有点烦琐，不支持直接的 RPC	网络数据交换

uses-permission 和 permission

uses-permission 和 permission 的区别

- permission 定义权限
- uses-permission 申请权限

用途

首先，什么是权限？我有一个应用A，一个应用B，现在应用 A 想使用应用 B 的一些功能，那么 Android 系统为了安全起见，肯定不能让我们随便就调用了。应用 B 必须先声明一些权限，对应于开放给外界调用的功能。应用 A 要用，必须要先注册这些权限，相当于获取了一块通行证。

权限类型

permission 自定义权限的时候可以指定权限类型。

android:protectionLevel： 说明权限中隐含的潜在风险，并指示系统在确定是否将权限授予请求授权的应用时应遵循的流程。下表列出了所有基本权限类型。

- **normal**：默认值。具有较低风险的权限。系统会自动向在安装时请求授权的应用授予此类权限，**无需征得用户的明确许可**（但用户始终可以选择在安装之前查看这些权限）。
- **dangerous**：具有较高风险的权限。由于此类权限会带来潜在风险，因此系统可能**不会自动向请求授权的应用授予此类权限**。
- **signature**：只有在请求授权的应用使用与声明权限的应用相同的证书进行签名时系统才会授予的权限。**如果证书匹配**，则系统会在不通知用户或征得用户明确许可的情况下自动授予权限。
- **signatureOrSystem**：**不要使用此选项，因为 signature 保护级别应足以满足大多数需求**，无论应用安装在何处，该保护级别都能正常发挥作用。signatureOrSystem 权限适用于以下特殊情况：多个供应商将应用内置到一个系统映像中，并且需要明确共享特定功能，因为这些功能是一起构建的。((除了满足signature的应用可以申请外，存放在系统目录 `/system/app` 目录下也可以申请。))