Android面试题

1. Android四大组件

应用组件是Android应用的基本构建基块。每个组件都是一个不同的点，系统可以通过它进入您的应用。并非所有组件都是用户的实际入口点，有些组件相互依赖，但每个组件都以独立实体形式存在，并发挥特定作用----每个组件都是唯一的构建基块，有助于定义应用的总体行为。

共有四种不同的应用组件类型。每种类型都服务于不同的目的，并且具有定义组件的创建和销毁方式的不同生命周期。

以下便是这四种应用组件类型：

1.1. Activity

Activity表示具有用户界面的单一屏幕。例如，电子邮件应用可能具有一个显示新电子邮件列表的Activity、一个用于撰写电子邮件的Activity以及一个用于阅读电子邮件的Activity。尽管这些Activity通过协作在电子邮件应用中形成了一种紧密结合的用户体验，但每一个Activity都独立于其他Activity而存在。因此，其他应用可以启动其中任何一个Activity（如果电子邮件应用允许）。例如，相机应用可以启动电子邮件应用内用于撰写新电子邮件的Activity，以便用户共享图片。

1.2. 服务

服务是一种在后台运行的组件，用于执行长时间运行的操作或为远程进程执行作业。服务不提供用户界面。例如，当用户位于其他应用中时，服务可能在后台播放音乐或者通过网络获取数据，但不会阻断用户与Activity的交互。诸如Activity等其他组件可以启动服务，让其运行或与其绑定以便与其进行交互。

1.3. 内容提供程序

内容提供程序管理一组共享的应用数据。您可以将数据存储在文件系统、SQLite数据库、网络上或您的应用可以访问的任何其他永久性存储位置。其他应用可以通过内容提供程序查询数据，甚至修改数据（如果内容提供程序允许）。例如，Android系统可提供管理用户联系人信息的内容提供程序。因此，任何具有适当权限的应用都可以查询内容提供程序的某一部分（如 `ContactsContract.Data` ），以读取和写入有关特定人员的信息。

内容提供程序也适用于读取和写入您的应用不共享的私有数据。例如，记事本示例应用使用内容提供程序来保存笔记。

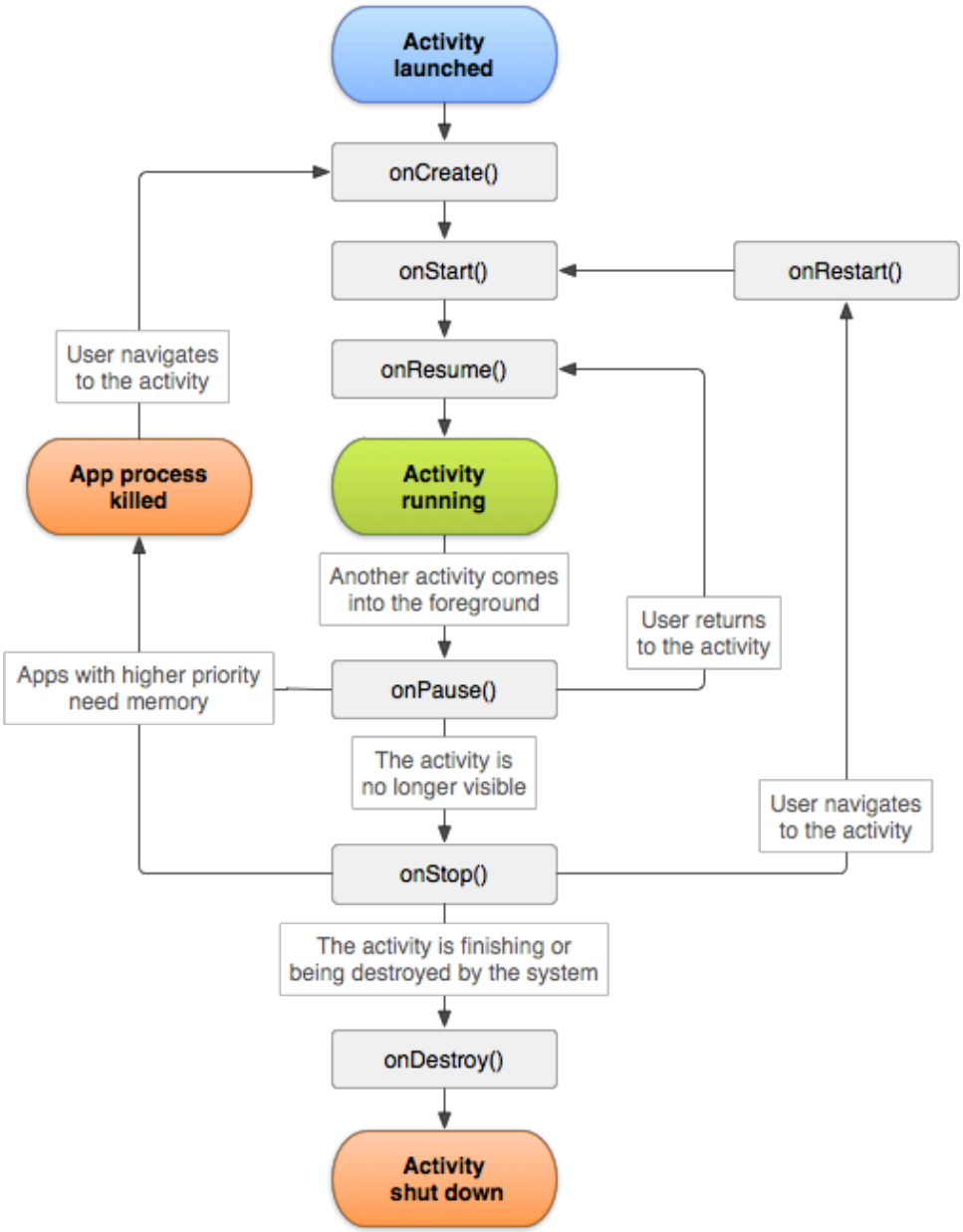
1.4. 广播接收器

广播接收器是一种用于响应系统范围广播通知的组件。许多广播都是由系统发起的----例如，通知屏幕已关闭、电池电量不足或已拍摄照片的广播。应用也可以发起广播----例如，通知其他应用某些数据已下载至设备，并且可供其使用。尽管广播接收器不会显示用户界面，但它们可以创建状态栏通知，在发生广播事件时提醒用户。但广播接收器更常见的用途只是作为通向其他组件的"通道"，设计用于执行极少量的工作。例如，它可能会基于事件发起一项服务来执行某项工作。

2. 四大组件的启动方式

- 您可以通过将Intent传递到 `startActivity()` 或 `startActivityForResult()` （当您想让Activity返回结果时）来启动Activity（或为其安排新任务）。
- 您可以通过将Intent传递到 `startService()` 来启动服务（或对执行中的服务下达新指令）。或者，您也可以通过将Intent传递到 `bindService()` 来绑定到该服务。
- 您可以通过将Intent传递到 `sendBroadcast()` 、 `sendOrderedBroadcast()` 或 `sendStickyBroadcast()` 等方法来发起广播。
- 您可以通过在ContentResolver上调用 `query()` 来对内容提供程序执行查询。

3. 画出Activity的生命周期图



4. 介绍下不同场景下Activity生命周期的变化过程

- 启动Activity：`onCreate()` --> `onStart()` --> `onResume()`，Activity进入运行状态。
- Activity退居后台：当前Activity转到新的Activity界面或按Home键回到主屏：`onPause()` --> `onStop()`，进入停滞状态；这里有一种特殊情况，如果新Activity采用了透明主题，那么当前Activity不会回调`onStop()`。
- Activity返回前台：`onRestart()` --> `onStart()` --> `onResume()`，再次回到运行状态。
- Activity退居后台，且系统内存不足，系统会杀死这个后台状态的Activity，若再次回到这个Activity，则会走`onCreate()` --> `onStart()` --> `onResume()`。
- 锁定屏与解锁屏幕只会调用`onPause()`，而不会调用`onStop()`方法，开屏后则调用`onResume()`。

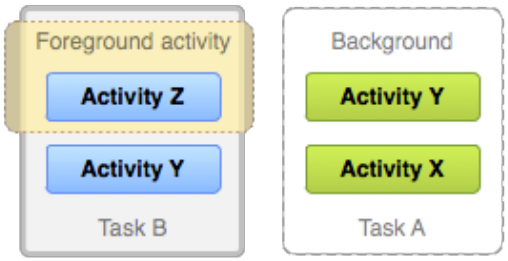
5. 当Activity A启动Activity B时，生命周期执行过程?

`A.onPause()` --> `B.onCreate()`，`B.onStart()`，`B.onResume()` --> `A.onStop()`，如果B是个透明的，或者是对话框的样式，就不会调用`A.onStop()`。

6. 内存不足时系统会杀掉后台的Activity，若需要进行一些临时状态的保存，在哪个方法进行？怎么恢复数据?

- Activity的`onSaveInstanceState()`和`onRestoreInstanceState()`并不是生命周期方法，它们不同于`onCreate()`、`onPause()`等生命周期方法，它们并不一定会被触发。
- 当应用遇到意外情况（如：内存不足、用户直接按Home键）由系统销毁一个Activity，`onSaveInstanceState()`会被调用。但是当用户主动去销毁一个Activity时，例如在应用中按返回键，`onSaveInstanceState()`就不会被调用。除非该activity是被用户主动销毁的，通常`onSaveInstanceState()`只适合用于保存一些临时性的状态，而`onPause()`适合用于数据的持久化保存。
- 重写`onSaveInstanceState()`方法，在此方法中保存需要保存的数据，该方法将会在activity被回收之前调用。通过重写`onRestoreInstanceState()`方法可以从中提取保存好的数据。

7. 什么是任务?



任务是一个有机整体，当用户开始新任务或通过“主页”按钮转到主屏幕时，可以移动到“后台”。尽管在后台时，该任务中的所有Activity全部停止，但是任务的返回栈仍旧不变，也就是说，当另一个任务发生时，该任务仅仅失去焦点而已，如图所示。然后，任务可以返回到“前台”，用户就能够回到离开时的状态。

8. Activity的启动模式?

- standard（默认模式）：系统在启动Activity的任务中创建Activity的新实例并向其传送Intent。Activity可以多次实例化，而

每个实例均可属于不同的任务，并且一个任务可以拥有多个实例。

- singleTop：如果当前任务的顶部已存在Activity的一个实例，则系统会通过调用该实例的onNewIntent()方法向其传送Intent，而不是创建Activity的新实例。Activity可以多次实例化，而每个实例均可属于不同的任务，并且一个任务可以拥有多个实例（但前提是位于返回栈顶部的Activity并不是Activity的现有实例）。例如，假设任务的返回栈包含根Activity A以及Activity B、C和位于顶部的D（堆栈是A-B-C-D；D位于顶部）。收到针对D类Activity的Intent。如果D具有默认的"standard"启动模式，则会启动该类的新实例，且堆栈会变成A-B-C-D-D。但是，如果D的启动模式是"singleTop"，则D的现有实例会通过onNewIntent()接收Intent，因为它位于堆栈的顶部；而堆栈仍为A-B-C-D。但是，如果收到针对B类Activity的Intent，则会向堆栈添加B的新实例，即便其启动模式为"singleTop"也是如此。
- singleTask：系统创建新任务并实例化位于新任务底部的Activity。但是，如果该Activity的一个实例已存在于一个单独的任务中，则系统会通过调用现有实例的onNewIntent()方法向其传送Intent，而不是创建新实例。一次只能存在Activity的一个实例。
- singleInstace：与"singleTask"相同，只是系统不会将任何其他Activity启动到包含实例的任务中。该Activity始终是其任务唯一仅有的成员；由此Activity启动的任何Activity均在单独的任务中打开。

9. 横竖屏切换时候activity的生命周期?

- 不设置Activity的 `android：configChanges` 时，切屏会重新调用各个生命周期，切横屏时会执行一次，切竖屏时会执行两次。
- 设置Activity的 `android：configChanges="orientation"` 时，切屏还是会重新调用各个生命周期，切横、竖屏时只会执行一次。
- 设置Activity的 `android：configChanges="orientation|keyboardHidden"` 时，切屏不会重新调用各个生命周期，只会执行 `onConfigurationChanged()` 方法。

10. 如何将一个Activity设置成窗口的样式?

只需要给我们的Activity配置如下属性即可 `android:theme="@android:style/Theme.Dialog"` 。

11. Activity之间的数据传递有哪些方式?

- `intent.putExtra()` 方法；
- 使用全局变量Application；
- 使用静态变量；
- 剪切板ClipboardManager传递数据；
- 借助Application共享Handler利用消息处理机制；
- 使用Broadcast广播；
- 使用EventBus。

12. Fragment的好处：

- Fragment可以使你能够将activity分离成多个可重用的组件，每个都有它自己的生命周期和UI。
- Fragment可以轻松得创建动态灵活的UI设计，可以适应于不同的屏幕尺寸。从手机到平板电脑。
- Fragment是一个独立的模块，紧紧地与activity绑定在一起。可以运行中动态地移除、加入、交换等。

- Fragment提供一个新的方式让你在不同的安卓设备上统一你的UI。
- Fragment解决Activity间的切换不流畅，轻量切换。
- Fragment替代TabActivity做导航，性能更好。
- Fragment在Android 4.2中新增嵌套fragment使用方法，能够生成更好的界面效果。

13. Intent的原理，作用，可以传递哪些类型的参数?

- Intent是连接Activity、Service、BroadcastReceiver和ContentProvider四大组件的信使，可以传递八种基本数据类型以及 `String` 、 `Bundle` 类型，以及实现了 `Serializable` 或者 `Parcelable` 的类型。
- Intent可以划分成显式意图和隐式意图。
    - 显式意图：调用 `Intent.setComponent()` 或 `Intent.setClass()` 方法明确指定了组件名的Intent为显式意图，显式意图明确指定了Intent应该传递给哪个组件。
    - 隐式意图：没有明确指定组件名的Intent为隐式意图。Android系统会根据隐式意图中设置的动作（ `action` ）、类别（ `category` ）、数据（URI和数据类型）找到最合适的组件来处理这个意图。

14. Intent的主要使用方法

14.1. 启动 Activity

Activity表示应用中的一个屏幕。通过将Intent传递给 `startActivity()` ，您可以启动新的 Activity实例。Intent描述了要启动的Activity，并携带了任何必要的数据。

如果您希望在Activity完成后收到结果，请调用 `startActivityForResult()` 。在 Activity的 `onActivityResult()` 回调中，您的Activity将结果作为单独的Intent对象接收。

14.2. 启动服务

Service是一个不使用用户界面而在后台执行操作的组件。通过将Intent传递给 `startService()` ，您可以启动服务执行一次性操作（例如，下载文件）。Intent描述了要启动的服务，并携带了任何必要的数据。

如果服务旨在使用客户端——服务器接口，则通过将Intent传递给 `bindService(`)，您可以从其他组件绑定到此服务。

14.3. 传递广播

广播是任何应用均可接收的消息。系统将针对系统事件（例如：系统启动或设备开始充电时）传递各种广播。通过将Intent传递给 `sendBroadcast()` 、 `sendOrderedBroadcast()` 或 `sendStickyBroadcast()` ，您可以将广播传递给其他应用。

15. Intent包含哪些信息

- 组件名称：要启动的组件名称。
- 操作：指定要执行的通用操作（例如，“查看”或“选取”）的字符串。
- 数据：引用待操作数据和 / 或该数据MIME类型的URI（Uri对象）。提供的数据类型通常由Intent的操作决定。例如，如果

操作是 `ACTION_EDIT` ，则数据应包含待编辑文档的URI。
- 类别：一个包含应处理Intent组件类型的附加信息的字符串。
- Extra：携带完成请求操作所需的附加信息的键值对。
- 标志：在Intent类中定义的、充当Intent元数据的标志。

16. 什么是Intent过滤器

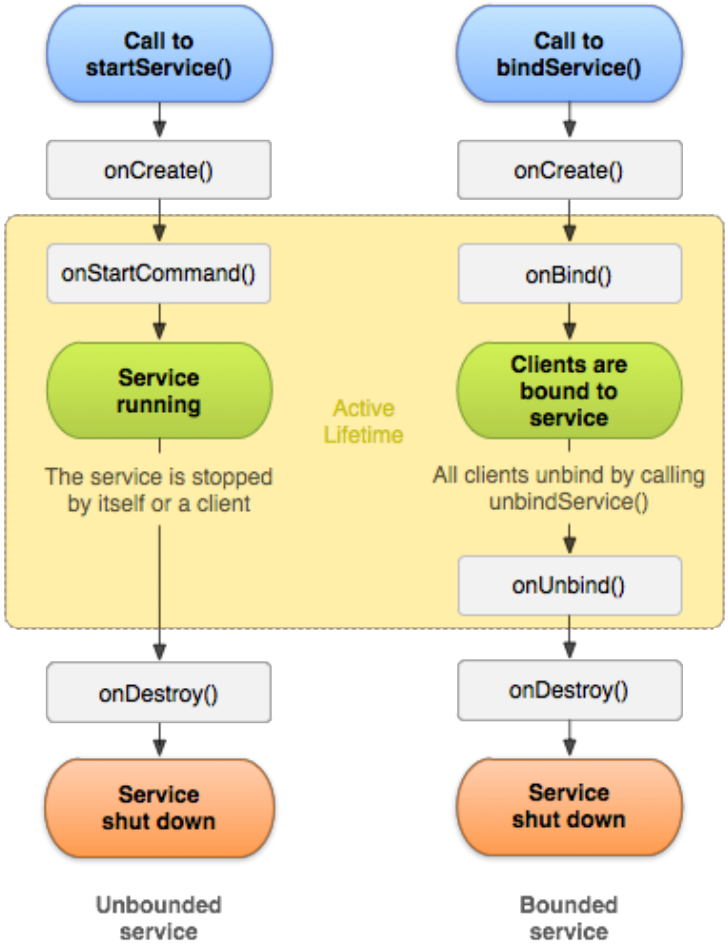
要公布应用可以接收哪些隐式Intent，请在清单文件中使用 `<intent-filter>` 元素为每个应用组件声明一个或多个Intent过滤器。每个Intent过滤器均由应用清单文件中的 `<intent-filter>` 元素定义，并嵌套在相应的应用组件（例如， `<activity>` 元素）中。在 `<intent-filter>` 内部，您可以使用以下三个元素中的一个或多个指定要接受的Intent类型：

- `<action>` ：在 `name` 属性中，声明接受的Intent操作。该值必须是操作的文本字符串值，而不是类常量。
- `<data>` ：使用一个或多个指定数据URI各个方面（scheme、host、port、path等）和MIME类型的属性，声明接受的数据类型。
- `<category>` ：在 `name` 属性中，声明接受的Intent类别。该值必须是操作的文本字符串值，而不是类常量。

17. Service的启动方式

- `startService()` ：只是启动Service，Activity和Service并没有绑定，只有当Service调用 `stopService()` 服务才会终止。
- `bindService()` ：这种启动方式Activity和Service进行了绑定，启动Service的组件可以通过回调获取Service的代理对象和Service交互；当启动方销毁时，Service也会自动进行 `unBind()` 操作，当发现所有绑定都进行了 `unBind()` 时才会销毁Service。

18. Service的生命周期



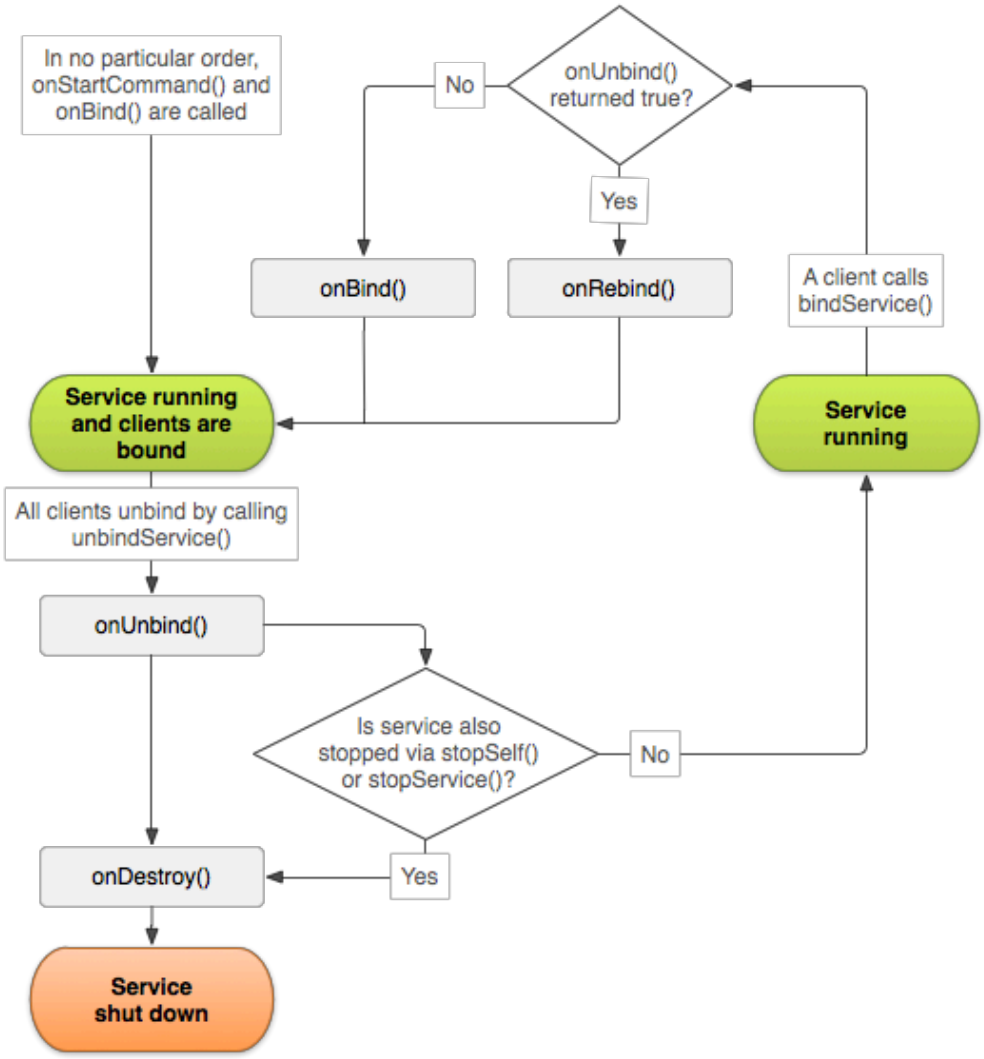
19. Activity怎么和Service绑定，怎么在Activity中启动自己对应的Service?

- Activity通过 `bindService(Intent service, ServiceConnection conn, int flags)` 跟Service进行绑定，当绑定成功的时候Service会将代理对象通过回调的形式传给 `conn` ，这样我们就拿到了Service提供的服务代理对象。
- 在Activity中可以通过 `startService()` 和 `bindService()` 方法启动Service。一般情况下如果想获取Service的服务对象那么肯定需要通过 `bindService()` 方法，比如音乐播放器，第三方支付等。如果仅仅只是为了开启一个后台任务那么可以使用 `startService()` 方法。

20. 允许绑定的已启动服务的生命周期

当服务与所有客户端之间的绑定全部取消时，Android系统便会销毁服务。不过，如果您选择实现 `onStartCommand()` 回调方法，则您必须显式停止服务，因为系统现在已将服务视为已启动。在此情况下，服务将一直运行到其通过 `stopSelf()` 自行停止，或其他组件调用 `stopService()` 为止，无论其是否绑定到任何客户端。

此外，如果您的服务已启动并接受绑定，则当系统调用您的 `onUnbind()` 方法时，如果您想在客户端下一次绑定到服务时接收 `onRebind()` 调用，则可选择返回true。 `onRebind()` 返回空值，但客户端仍在其 `onServiceConnected()` 回调中接收IBinder。



21. Service中可以弹Toast吗?

- 这个问题其实就是问一下Service是执行在UI线程中吗？类似的问题还有"Service的 `onCreate()` 回调函数可以做耗时的操作吗？"，"Service是否在main thread中执行"，"Service和Activity在同一个线程吗？"等；
- 我们要牢记一句真理"默认情况下四大组件都是在UI线程中执行的"，Service本身就是Context的子类，我们可以获取到Context对象，所以Service中当然可以弹Toast，同理，Service的 `onCreate()` 回调函数不可以做耗时的操作。

22. 进程的优先级

22.1. 前台进程

用户当前操作所必需的进程。如果一个进程满足以下任一条件，即视为前台进程：

- 托管用户正在交互的Activity（已调用Activity的 `onResume()` 方法）；
- 托管某个Service，后者绑定到用户正在交互的Activity；
- 托管正在"前台"运行的Service（服务已调用 `startForeground()` ）；

- 托管正执行一个生命周期回调的Service（`onCreate()`、`onStart()`或`onDestroy()`）；
- 托管正执行其`onReceive()`方法的BroadcastReceiver。

通常，在任意给定时间前台进程都为数不多。只有在内存不足以支持它们同时继续运行这一万不得已的情况下，系统才会终止它们。此时，设备往往已达到内存分页状态，因此需要终止一些前台进程来确保用户界面正常响应。

22.2. 可视进程

没有任何前台组件、但仍会影响用户在屏幕上所见内容的进程。如果一个进程满足以下任一条件，即视为可见进程：

- 托管不在前台、但仍对用户可见的Activity（已调用其`onPause()`方法）。例如，如果前台Activity启动了一个对话框，允许在其后显示上一Activity，则有可能会发生这种情况。
- 托管绑定到可见（或前台）Activity的Service。

可见进程被视为是极其重要的进程，除非为了维持所有前台进程同时运行而必须终止，否则系统不会终止这些进程。

22.3. 服务进程

正在运行已使用`startService()`方法启动的服务且不属于上述两个更高类别进程的进程。尽管服务进程与用户所见内容没有直接关联，但是它们通常在执行一些用户关心的操作（例如，在后台播放音乐或从网络下载数据）。因此，除非内存不足以维持所有前台进程和可见进程同时运行，否则系统会让服务进程保持运行状态。

22.4. 后台进程

包含目前对用户不可见的Activity的进程（已调用Activity的`onStop()`方法）。这些进程对用户体验没有直接影响，系统可能随时终止它们，以回收内存供前台进程、可见进程或服务进程使用。通常会有很多后台进程在运行，因此它们会保存在LRU（最近最少使用）列表中，以确保包含用户最近查看的Activity的进程最后一个被终止。如果某个Activity正确实现了生命周期方法，并保存了其当前状态，则终止其进程不会对用户体验产生明显影响，因为当用户导航回该Activity时，Activity会恢复其所有可见状态。

22.5. 空进程

不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。

23. IntentService如何工作?

- 创建默认的工作线程，用于在应用的主线程外执行传递给`onStartCommand()`的所有Intent。
- 创建工作队列，用于将Intent逐一传递给`onHandleIntent()`实现，这样您就永远不必担心多线程问题。
- 在处理完所有启动请求后停止服务，因此您永远不必调用`stopSelf()`。
- 提供`onBind()`的默认实现（返回`null`）。
- 提供`onStartCommand()`的默认实现，可将Intent依次发送到工作队列和`onHandleIntent()`实现。

24. IntentService与Service的区别?

- Service也不是专门一条新线程，因此不应该在Service中直接处理耗时的任务；
- Service不会专门启动一条单独的进程，Service与它所在应用位于同一个进程中；
- IntentService是Service的子类，是一个异步的，会自动停止的服务，很好解决了传统的Service中处理完耗时操作忘记停止并销毁Service的问题；
- IntentService会创建独立的worker线程来处理所有的Intent请求；
- IntentService不会阻塞UI线程，而普通Serveice会导致ANR异常；
- Intentservice若未执行完成上一次的任务，将不会新开一个线程，是等待之前的任务完成后，再执行新的任务，等任务完成后再次调用 `stopSelf()` ；
- 正在运行的IntentService的程序相比起纯粹的后台程序更不容易被系统杀死，该程序的优先级是介于前台程序与纯后台程序之间的。

25. Android Service与Activity之间的通信方式?

- 通过Binder对象：当Activity通过调用 `bindService(Intent service, ServiceConnection conn, int flags)` ，得到一个Service的一个对象，通过这个对象我们可以直接访问Service中的方法。
    - 添加一个继承Binder的内部类，并添加相应的逻辑方法。
    - 重写Service的 `onBind()` 方法，返回我们刚刚定义的那个内部类实。
    - Activity中创建一个ServiceConnection的匿名内部类，并且重写里面的 `onServiceConnected()` 方法和 `onServiceDisconnected()` 方法，这两个方法分别会在活动与服务成功绑定以及解除绑定的时候调用，在 `onServiceConnected()` 方法中，我们可以得到一个刚才那个service的binder对象，通过对这个binder对象进行向下转型，得到我们那个自定义的Binder实例，有了这个实例，做可以调用这个实例里面的具体方法进行需要的操作了。
- 通过Broadcast Receiver：当我们的进度发生变化的时候我们发送一条广播，然后在Activity的注册广播接收器，接收到广播之后更新视图
- EventBus

26. BroadcastReceiver简介

在Android中，Broadcast是一种广泛运用的在应用程序之间传输信息的机制。

26.1. 用途

- 实现了不同的程序之间的数据传输与共享，因为只要是和发送广播的 `action` 相同的接受者都能接受这个广播。典型的应用就是Android自带的短信，电话等等广播，只要我们实现了他们的 `action` 的广播，那么我们就能接收他们的数据了，以便做出一些处理。比如说拦截系统短信，拦截骚扰电话等。
- 起到了一个通知的作用，比如在Service中要通知主程序，更新主程序的UI等。因为Service是没有界面的，所以不能直接获得主程序中的控件，这样我们就只能在主程序中实现一个广播接受者专门用来接受Service发过来的数据和通知了。

26.2. 使用场景

- 同一app内部的同一组件内的消息通信（单个或多个线程之间）；
- 同一app内部的不同组件之间的消息通信（单个进程）；
- 同一app具有多个进程的不同组件之间的消息通信；
- 不同app之间的组件之间消息通信；
- Android系统在特定情况下与App之间的消息通信。

26.3. 实现原理

从实现原理看上，Android中的广播使用了观察者模式，基于消息的发布/订阅事件模型。因此，从实现的角度来看，Android中的广播将广播的发送者和接受者极大程度上解耦，使得系统能够方便集成，更易扩展。具体实现流程要点粗略概括如下：

1. 广播接收者BroadcastReceiver通过Binder机制向AMS（Activity Manager Service)进行注册；
2. 广播发送者通过binder机制向AMS发送广播；
3. AMS查找符合相应条件（IntentFilter/Permission等）的BroadcastReceiver，将广播发送到BroadcastReceiver（一般情况下是Activity）相应的消息循环队列中；
4. 消息循环执行拿到此广播，回调BroadcastReceiver中的 `onReceive()` 方法。

26.4. 注册方式

- 静态注册；
- 动态注册。

27. 为什么要用ContentProvider？它和SQL的实现上有什么差别?

- ContentProvider屏蔽了数据存储的细节，内部实现对用户完全透明，用户只需要关心操作数据的uri就可以了，ContentProvider可以实现不同app之间共享。SQL只能在该工程的内部共享数据，ContentProvider能在工程之间实现数据共享。
- SQL也有增删改查的方法，但是SQL只能查询本应用下的数据库。而ContentProvider还可以去增删改查本地文件.xml文件的读取等。

28. ContentProvider怎么实现数据共享?

一个程序可以通过实现一个ContentProvider的抽象接口将自己的数据完全暴露出去，而且ContentProvider是以类似数据库中表的方式将数据暴露。ContentProvider存储和检索数据，通过它可以让所有的应用程序访问到，这也是应用程序之间唯一共享数据的方法。要想使应用程序的数据公开化，可通过2种方法：创建一个属于你自己的Content provider或者将你的数据添加到一个已经存在的ContentProvider中，前提是有相同数据类型并且有写入ContentProvider的权限。

29. Android如何访问自定义ContentProvider

1. 得到ContentResolver类对象： `ContentResolver cr = getContentResolver()` ；
2. 定义要查询的字段 `String` 数组；
3. 使用 `cr.query()` 返回一个 `Cursor` 对象；
4. 使用 `while` 循环得到 `Cursor` 里面的内容。

30. Android中Activity，Intent，Content Provider，Service各有什么区别。

- Activity：活动，是最基本的Android应用程序组件。一个活动就是一个单独的屏幕，每一个活动都被实现为一个独立的类，并且从活动基类继承而来。
- Intent：意图，描述应用想干什么。最重要的部分是动作和动作对应的数据。
- Content Provider：内容提供器，Android应用程序能够将它们的数据保存到文件、SQLite数据库中，甚至是任何有效的设备中。当你想将你的应用数据和其他应用共享时，内容提供器就可以发挥作用了。
- Service：服务，具有一段较长生命周期且没有用户界面的程序。

31. Android数据存储方式?

- SharedPreferences：以键值对的形式保存少量的数据，且这些数据的格式非常简单：字符串型、基本类型的值。
- 文件存储数据：Context提供了两个方法来打开数据文件里的文件IO流`FileInputStream openFileInput(String name)`，`FileOutputStream(String name, int mode)`，这两个方法第一个参数用于指定文件名，第二个参数指定打开文件的模式；文件默认存储位置：`/data/data/包名/files/文件名`。
- SQLite存储数据。
- 使用ContentProvider存储数据。
- 网络存储数据。

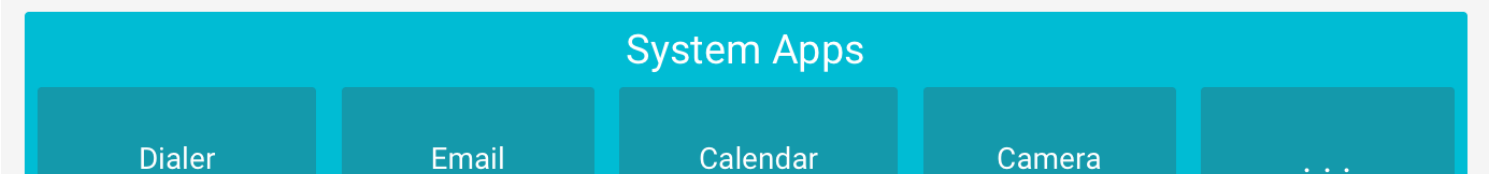
32. Android中常用的布局都有哪些?

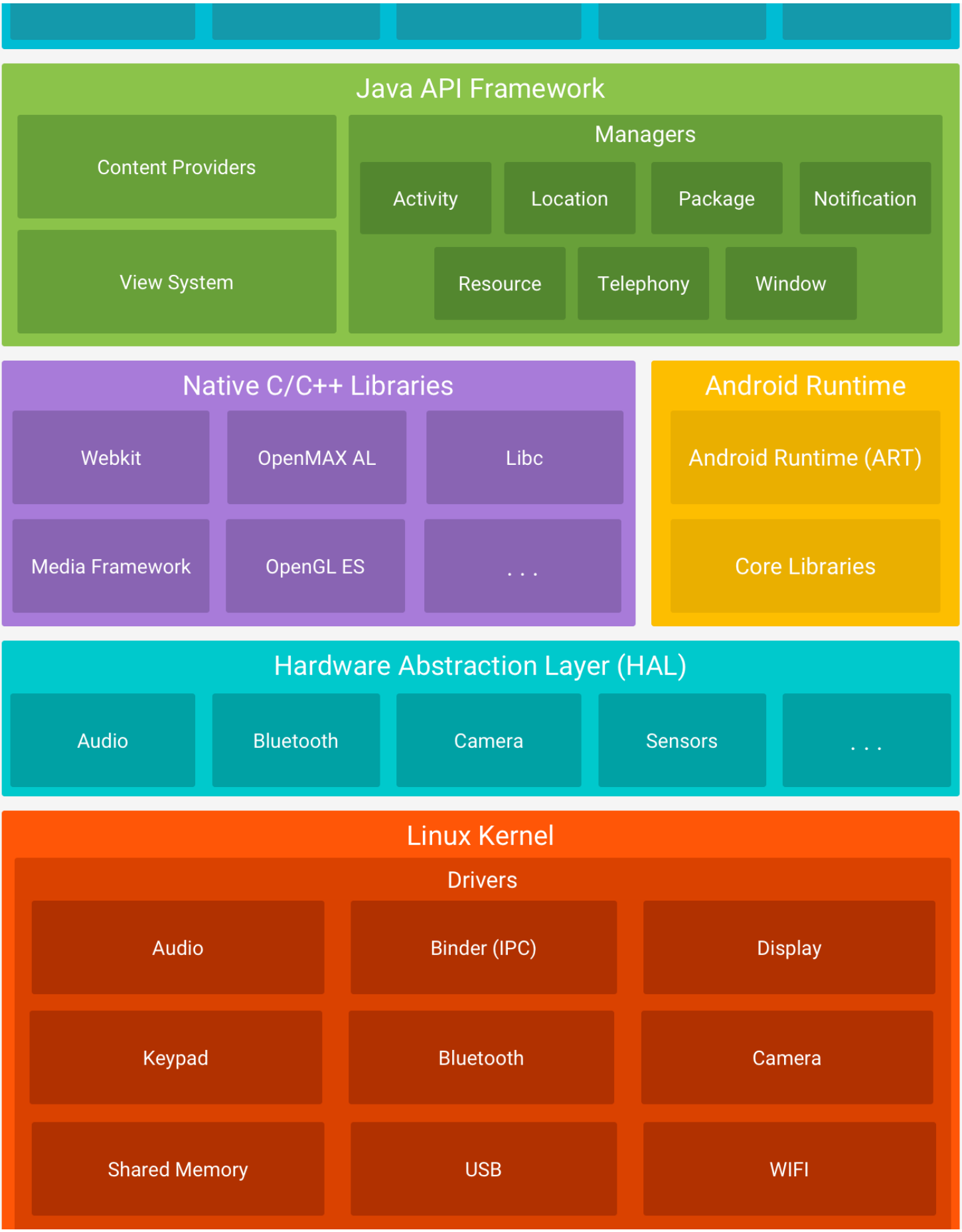
- FrameLayout；
- RelativeLayout；
- LinearLayout；
- AbsoluteLayout；
- TableLayout；
- GridLayout。

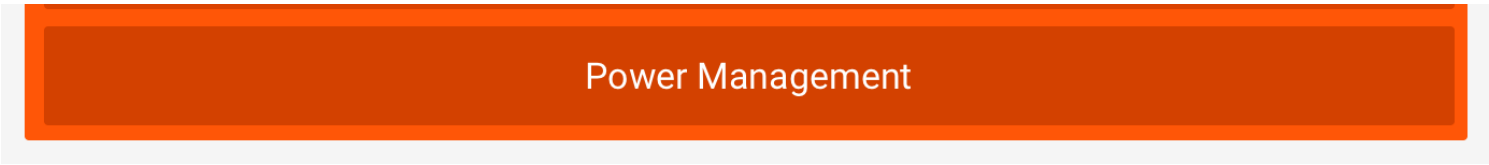
33. `android:layout_gravity`和`android:gravity`的区别?

- `android:layout_gravity`是让该布局在其父控件中的布局方式。
- `android:gravity`是该布局布置其子对象的布局方式。

34. Android平台架构




Java API Framework
Content Providers
View System
Managers
Activity
Location
Package
Notification
Resource
Telephony
Window
Native C/C++ Libraries
Webkit
OpenMAX AL
Libc
Media Framework
OpenGL ES
. . .
Android Runtime
Android Runtime (ART)
Core Libraries
Hardware Abstraction Layer (HAL)
Audio
Bluetooth
Camera
Sensors
. . .
Linux Kernel
Drivers
Audio
Binder (IPC)
Display
Keypad
Bluetooth
Camera
Shared Memory
USB
WIFI



34.1. 系统应用

Android随附一套用于电子邮件、短信、日历、互联网浏览和联系人等的核心应用。平台随附的应用与用户可以选择安装的应用一样，没有特殊状态。因此第三方应用可成为用户的默认网络浏览器、短信Messenger甚至默认键盘（有一些例外，例如系统的"设置"应用）。

系统应用可用作用户的应用，以及提供开发者可从其自己的应用访问的主要功能。例如，如果您的应用要发短信，您无需自己构建该功能，可以改为调用已安装的短信应用向您指定的接收者发送消息。

34.2. Java API框架

您可通过以Java语言编写的API使用Android OS的整个功能集。这些API形成创建Android应用所需的构建块，它们可简化核心模块化系统组件和服务的重复使用，包括以下组件和服务：

- 丰富、可扩展的视图系统，可用以构建应用的UI，包括列表、网格、文本框、按钮甚至可嵌入的网络浏览器；
- 资源管理器，用于访问非代码资源，例如本地化的字符串、图形和布局文件；
- 通知管理器，可让所有应用在状态栏中显示自定义提醒；
- Activity管理器，用于管理应用的生命周期，提供常见的导航返回栈；
- 内容提供程序，可让应用访问其他应用（例如"联系人"应用）中的数据或者共享其自己的数据。

开发者可以完全访问Android系统应用使用的框架API。

34.3. 原生C/C++库

许多核心Android系统组件和服务（例如ART和HAL）构建自原生代码，需要以C和C++编写的原生库。Android平台提供Java框架API以向应用显示其中部分原生库的功能。例如，您可以通过Android框架的Java OpenGL API访问OpenGL ES，以支持在应用中绘制和操作2D和3D图形。

如果开发的是需要C或C++代码的应用，可以使用Android NDK直接从原生代码访问某些原生平台库。

34.4. Android Runtime

对于运行Android 5.0（API级别21）或更高版本的设备，每个应用都在其自己的进程中运行，并且有其自己的Android Runtime(ART)实例。ART编写为通过执行DEX文件在低内存设备上运行多个虚拟机，DEX文件是一种专为Android设计的字节码格式，经过优化，使用的内存很少。编译工具链（例如Jack）将Java源代码编译为DEX字节码，使其可在Android平台上运行。

ART的部分主要功能包括：

- 预先（AOT）和即时（JIT）编译；
- 优化的垃圾回收（GC）；

- 更好的调试支持，包括专用采样分析器、详细的诊断异常和崩溃报告，并且能够设置监视点以监控特定字段；

在Android版本5.0（API级别21）之前，Dalvik是Android Runtime。如果您的应用在ART上运行效果很好，那么它应该也可在Dalvik上运行，但反过来不一定。

Android还包含一套核心运行时库，可提供Java API框架使用的Java编程语言大部分功能，包括一些Java 8语言功能。

34.5. 硬件抽象层（HAL）

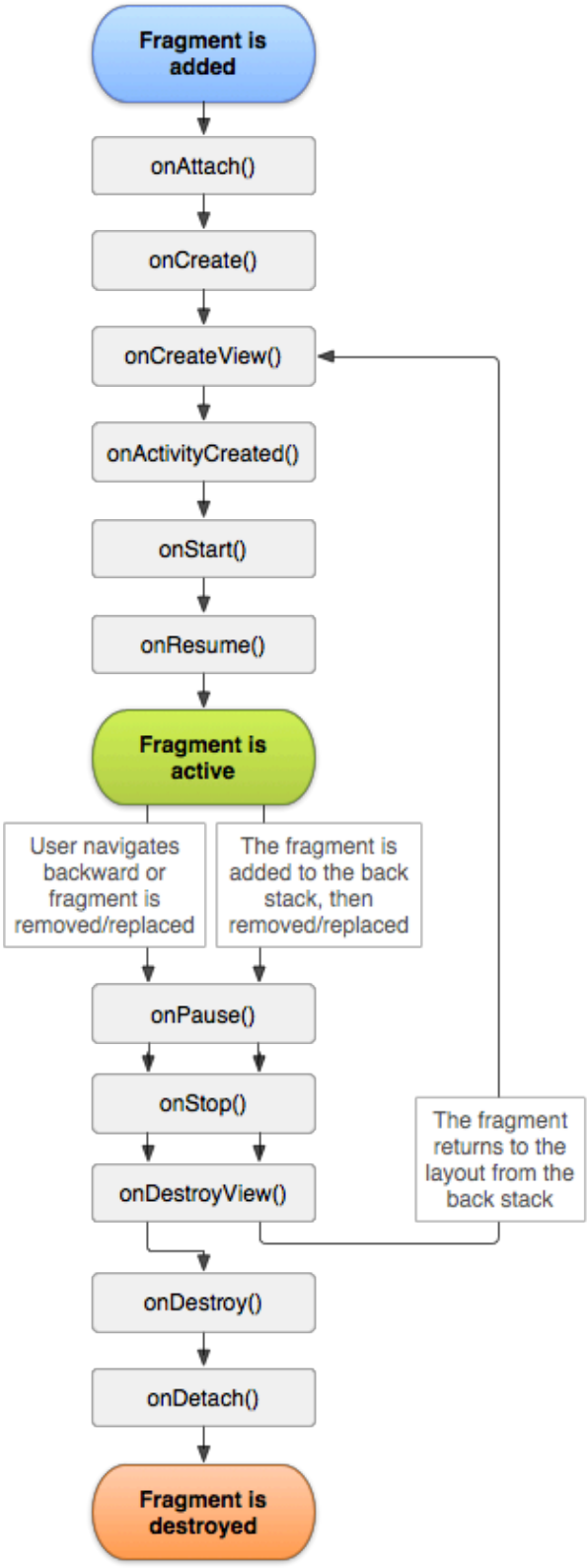
硬件抽象层（HAL）提供标准界面，向更高级别的Java API框架显示设备硬件功能。HAL包含多个库模块，其中每个模块都为特定类型的硬件组件实现一个界面，例如相机或蓝牙模块。当框架API要求访问设备硬件时，Android系统将为该硬件组件加载库模块。

34.6. Linux 内核

Android平台的基础是Linux内核。例如，Android Runtime（ART）依靠Linux内核来执行底层功能，例如线程和低层内存管理。

使用Linux内核可让Android利用主要安全功能，并且允许设备制造商为著名的内核开发硬件驱动程序。

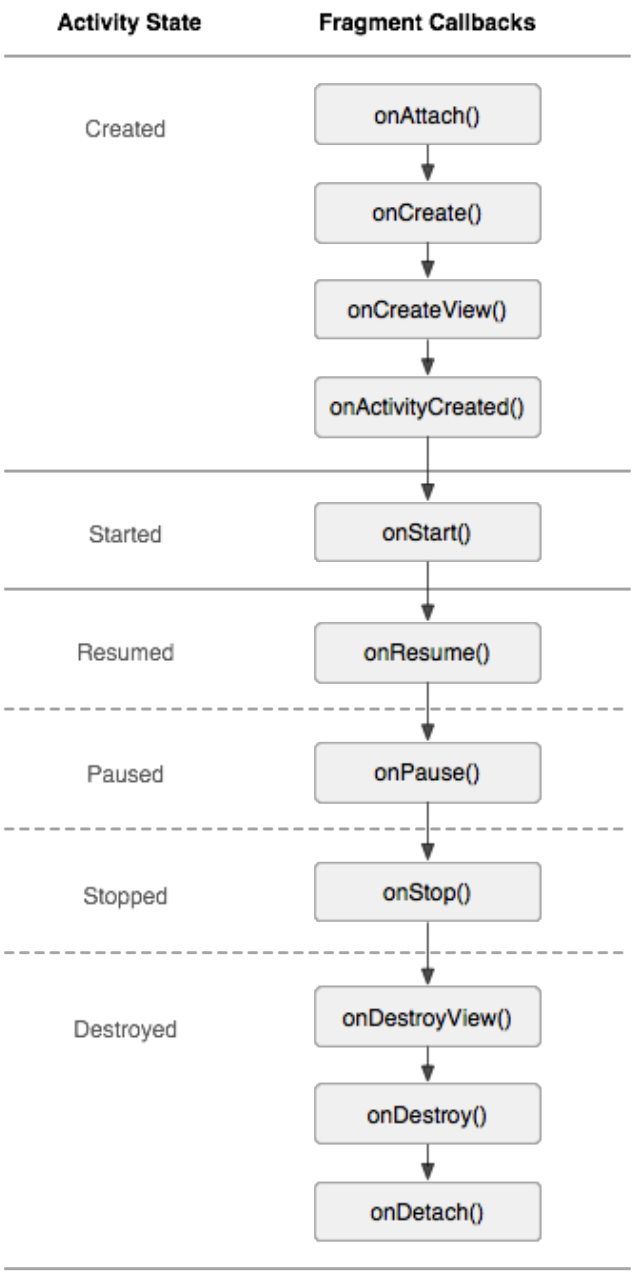
35. Fragment生命周期



- `onCreate()`：系统会在创建片段时调用此方法。您应该在实现内初始化您想在片段暂停或停止后恢复时保留的必需片段组件。
- `onCreateView()`：系统会在片段首次绘制其用户界面时调用此方法。要想为您的片段绘制UI，您从此方法中返回的View必须是片段布局的根视图。如果片段未提供UI，您可以返回 `null` 。

- `onPause()`：系统将此方法作为用户离开片段的第一个信号（但并不总是意味着此片段会被销毁）进行调用。您通常应该在此方法内确认在当前用户会话结束后仍然有效的任何更改（因为用户可能不会返回）。

36. Activity生命周期对片段生命周期的影响



片段所在的Activity的生命周期会直接影响片段的生命周期，其表现为，Activity的每次生命周期回调都会引发每个片段的类似回调。例如，当Activity收到`onPause()`时，Activity中的每个片段也会收到`onPause()`。

不过，片段还有几个额外的生命周期回调，用于处理与Activity的唯一交互，以执行构建和销毁片段UI等操作。这些额外的回调方法是：

- `onAttach()`：在片段已与Activity关联时调用（Activity传递到此方法内）；

- `onCreateView()`：调用它可创建与片段关联的视图层次结构；
- `onActivityCreated()`：在Activity的`onCreate()`方法已返回时调用；
- `onDestroyView()`：在移除与片段关联的视图层次结构时调用；
- `onDetach()`：在取消片段与Activity的关联时调用。

37. Android事件分发

事件相关方法	方法功能	Activity	ViewGroup	View
`dispatchTouchEvent`	事件分发	Yes	Yes	Yes
`onInterceptTouchEvent`	事件拦截	No	Yes	No
`onTouchEvent`	事件消费	Yes	Yes	Yes

37.1. `public boolean dispatchTouchEvent(MotionEvent ev)`

当有监听到事件时，首先由Activity进行捕获，进入事件分发处理流程。（因为activity没有事件拦截，View和ViewGroup有）会将事件传递给最外层View的`dispatchTouchEvent(MotionEvent ev)`方法，该方法对事件进行分发。

- `return true`：表示该View内部消化掉了所有事件。
- `return false`：事件在本层不再继续进行分发，并交由上层控件的`onTouchEvent()`方法进行消费（如果本层控件已经是Activity，那么事件将被系统消费或处理）。
- 如果事件分发返回系统默认的`super.dispatchTouchEvent(ev)`，事件将分发给本层的事件拦截`onInterceptTouchEvent()`方法进行处理。

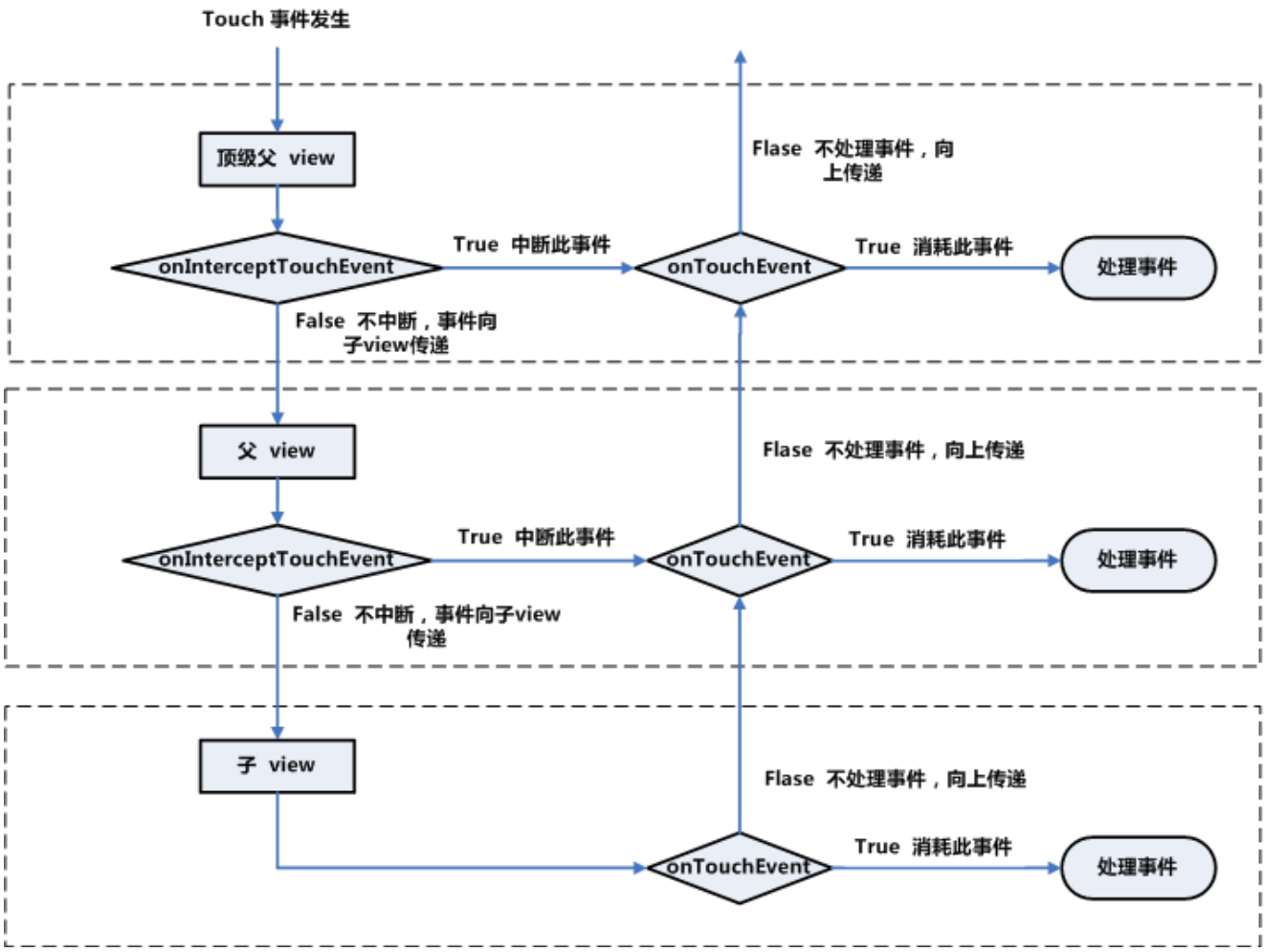
37.2. `public boolean onInterceptTouchEvent(MotionEvent ev)`

- `return true`：表示将事件进行拦截，并将拦截到的事件交由本层控件的`onTouchEvent()`进行处理。
- `return false`：则表示不对事件进行拦截，事件得以成功分发到子View。并由子View的`dispatchTouchEvent()`进行处理。
- 如果返回`super.onInterceptTouchEvent(ev)`，默认表示拦截该事件，并将事件传递给当前View的`onTouchEvent()`方法，和`return true`一样。

37.3. `public boolean onTouchEvent(MotionEvent ev)`

在`dispatchTouchEvent()`（事件分发）返回`super.dispatchTouchEvent(ev)`并且`onInterceptTouchEvent()`（事件拦截返回`true`或`super.onInterceptTouchEvent(ev)`的情况下，那么事件会传递到`onTouchEvent()`方法，该方法对事件进行响应。

- 如果`return true`，表示`onTouchEvent()`处理完事件后消费了此次事件。此时事件终结。
- 如果`return fasle`，则表示不响应事件，那么该事件将会不断向上层View的`onTouchEvent()`方法传递，直到某个View的`onTouchEvent()`方法返回`true`，如果到了最顶层View还是返回`false`，那么认为该事件不消耗，则在同一个事件系列中，当前View无法再次接收到事件，该事件会交由Activity的`onTouchEvent()`进行处理。
- 如果`return super.dispatchTouchEvent(ev)`，则表示不响应事件，结果与`return false`一样。



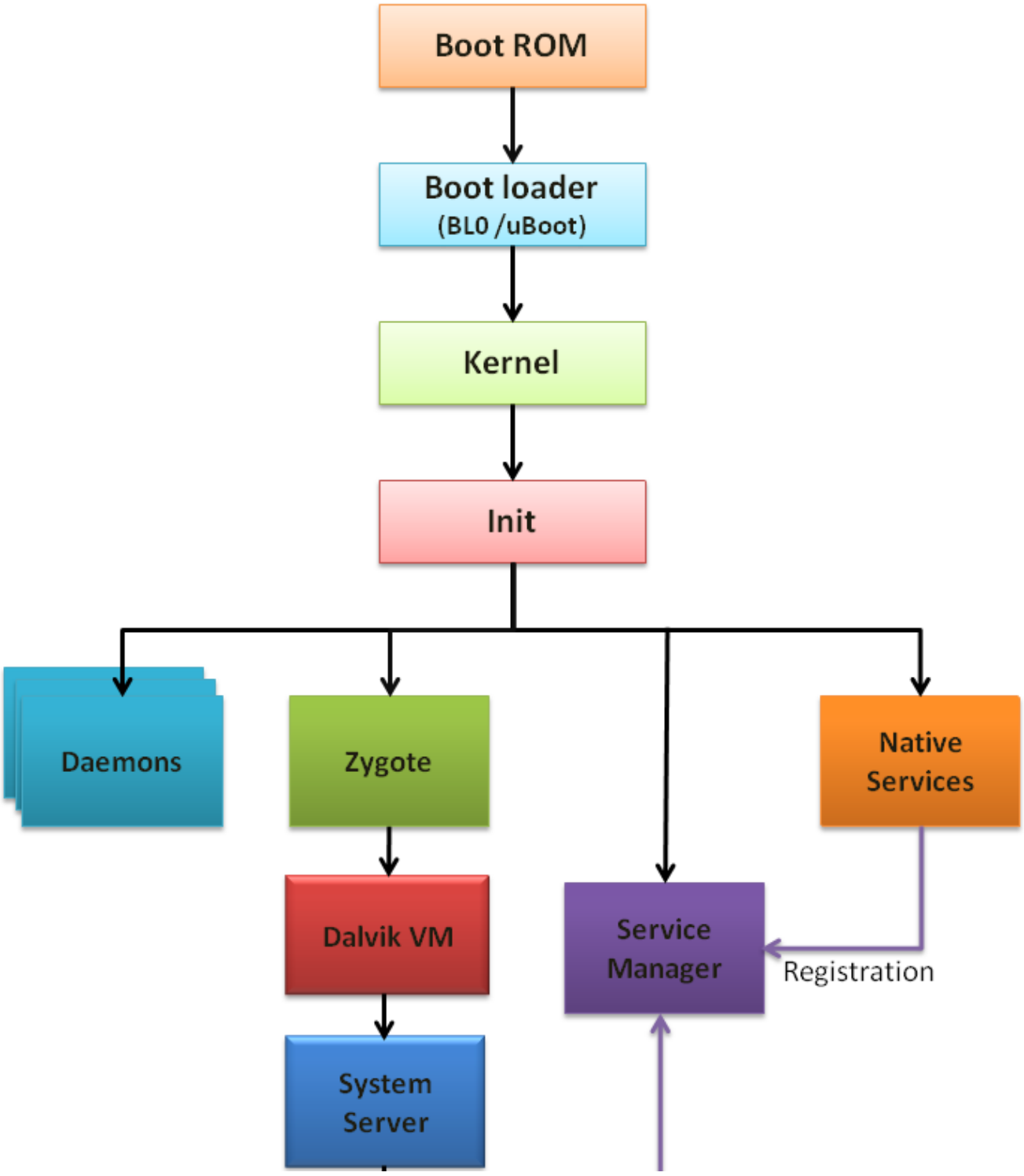
- 如果ViewGroup找到了能够处理该事件的View，则直接交给子View处理，自己的 `onTouchEvent()` 不会被触发。
- 可以通过复写 `onInterceptTouchEvent(ev)` 方法，拦截子View的事件（即 `return true` ），把事件交给自己处理，则会执行自己对应的 `onTouchEvent()` 方法。
- 子View可以通过调用 `getParent().requestDisallowInterceptTouchEvent(true)` 阻止ViewGroup对其 `MOVE` 或者 `UP` 事件进行拦截。
- 一个点击事件产生后，它的传递过程如下：Activity->Window->View。顶级View接收到事件之后，就会按相应规则去分发事件。如果一个View的 `onTouchEvent()` 方法返回 `false` ，那么将会交给父容器的 `onTouchEvent()` 方法进行处理，逐级往上，如果所有的View都不处理该事件，则交由Activity的 `onTouchEvent()` 进行处理。
- 如果某一个View开始处理事件，如果他不消耗 `ACTION_DOWN` 事件（也就是 `onTouchEvent()` 返回 `false` ），则同一事件序列比如接下来进行 `ACTION_MOVE` ，则不会再交给该View处理。
- ViewGroup默认不拦截任何事件。
- 诸如TextView、ImageView这些不作为容器的View，一旦接受到事件，就调用 `onTouchEvent()` 方法，它们本身没有 `onInterceptTouchEvent()` 方法。正常情况下，它们都会消耗事件（返回 `true` ），除非它们是不可点击的（ `clickable` 和 `longClickable` 都为 `false` ），那么就会交由父容器的 `onTouchEvent()` 处理。
- 点击事件分发过程如下 `dispatchTouchEvent()` ---> `OnTouchListener` 的 `onTouch()` 方法 ---> `onTouchEvent()` --> `OnClickListener` 的 `onClick()` 方法。也就是说，我们平时调用的 `setOnClickListener()` ，优先级是最低的，所以， `onTouchEvent()` 或 `OnTouchListener()`

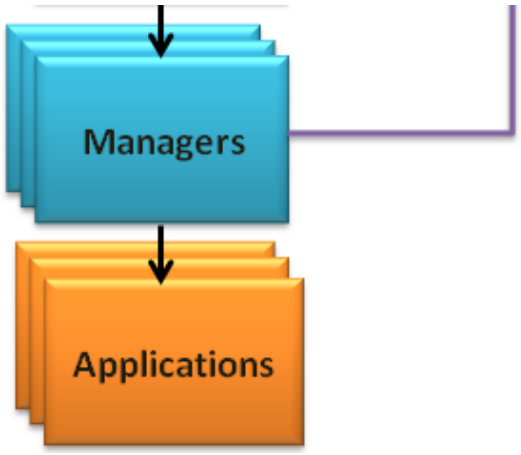
的 `onTouch()` 方法如果返回 `true` ，则不响应 `onClick()` 方法。

参考：ForAndroidInterview/Android View事件分发机制源码分析.md at master · Mr-YangCheng/ForAndroidInterview

参考：Android 编程下 Touch 事件的分发和消费机制 - sunzn - 博客园

38. Android系统启动过程





38.1. Boot ROM

Android设备上电后，首先会从处理器片上ROM的启动引导代码开始执行，片上ROM会寻找Bootloader代码，并加载到内存。

38.2. Boot Loader

BootLoader是在操作系统内核运行之前运行。可以初始化硬件设备、建立内存空间映射图，从而将系统的软硬件环境带到一个合适状态，以便为最终调用操作系统内核准备好正确的环境。

38.3. Kernel

Android内核启动时，会设置缓存、被保护存储器、计划列表，加载驱动。当内核完成系统设置，它首先在系统文件中寻找"init"文件，然后启动root进程或者系统的第一个进程。

38.4. init

init进程，它是一个由内核启动的用户级进程。内核自行启动（已经被载入内存，开始运行，并已初始化所有的设备驱动程序和数据结构等）之后，就通过启动一个用户级程序init的方式，完成引导进程。init始终是第一个进程。

init程序最核心的工作主要有3点：

- 创建和挂载一些系统目录/设备节点，设置权限，如：`/dev`，`/proc`，和`/sys`；
- 解析init.rc，并启动属性服务，以及一系列的服务和进程；
- 显示boot logo，默认是"Android"字样。

第二步的这些服务包含2部分，一部分是本地服务，另一部分是Android服务，所有的这些服务都会向ServiceManager进程注册，由它统一管理，这些服务的启动过程介绍如下：

38.4.1. 本地服务

本地服务是指运行在C++层的系统守护进程，一部分本地服务是init进程直接启动的，它们定义在init.rc脚本中，如ueventd、servicemanager、debuggerd、rild、mediaserver等。还有一部分本地服务，是由这些本地服务进一步创建的，如mediaserver服务会启动AudioFlinger，MediaPlayerService，以及CameraService等本地服务。

注意，每一个由init直接启动的本地服务都是一个独立的Linux进程，在系统启动以后，我们通过 `adb shell` 命令进入手机后，输入 `top` 命令就可以查看到这些本地进程的存在。

38.4.2. Android服务

init进程会执行app_process程序，创建Zygote进程，它是Android系统最重要的进程，所有后续的Android应用程序都是由它 `fork` 出来的。

Zygote进程会首先 `fork` 出SystemServer进程，SystemServer进程的全部任务就是将所有的Android核心服务启动起来。

38.5. Zygote and Dalvik（ART）

Zygote被init进程启动，开始运行和初始化dalvik虚拟机。

38.6. System Server

系统服务是在系统中运行的第一个java组件，它会启动所有的Android服务，比如：电话服务，蓝牙服务，每个服务的启动被直接写在 `SystemServer.java` 这个类的 `run()` 方法里面。

38.7. Boot completed

一旦系统服务启动并运行，Android系统启动就完成了，同时发出 `ACTION_BOOT_COMPLETED` 广播。

39. Android应用启动过程

1. Launcher接收到点击事件，获取应用的信息，向SystemServer（ActivityManagerService简称AMS运行在里面）发起启动应用的请求；
2. SystemServer（AMS）请求Launcher Pause（Launcher需要保存状态进入后台）；
3. LauncherPause，向SystemServer（AMS）发送Pause完毕；
4. SystemServer（AMS）向Zygote请求启动一个新进程（calculator）；
5. Zygote fork出新进程（calculator），在新进程中执行ActivityThread类的 `main()` 方法；
6. calculator向SystemServer（AMS）请求attach到AMS；
7. SystemServer（AMS）请求calculator launch；
8. calculator调用 `onCreate()`， `onResume()` 回调；
9. calculator界面显示自屏幕上（还需细分）。

参考：[Android 应用程序启动过程分析]

40. dp，dip，dpi，ppi区别

px（Pixels，像素）：屏幕上的点。 in（Inch，英寸）：长度单位。 mm（Millimeter，毫米）：长度单位。 pt（Point，磅）：1/72in。 dpi（Dots Per Inch，每英寸所打印的点数）：1in长度的点数。 ppi（Pixels Per Inch，像素密度）：1in长度的像素点数。 dp/dip（Density-independent Pixels，与密度无关的像素）：一种基于屏幕密度的抽象单位。在160dpi的显示器上，1dp = 1px。 sp（Scale-independent Pixels，与刻度无关的像素）：与dp类似，但是可以根据用户的字体大小首选项进行缩放。

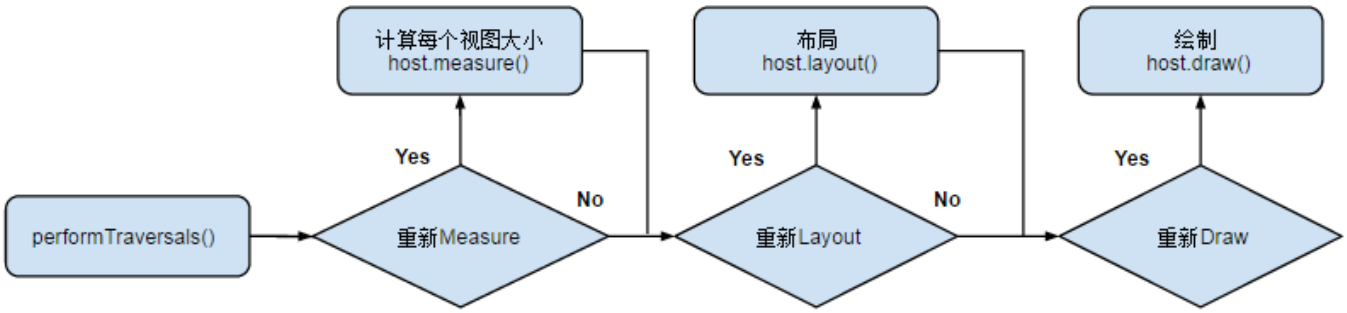
在屏幕密度为160dpi，1dp = 1px，1pt = 160/72sp，1pt = 1/72in。当屏幕密度为240dpi时，1dp = 1.5px。

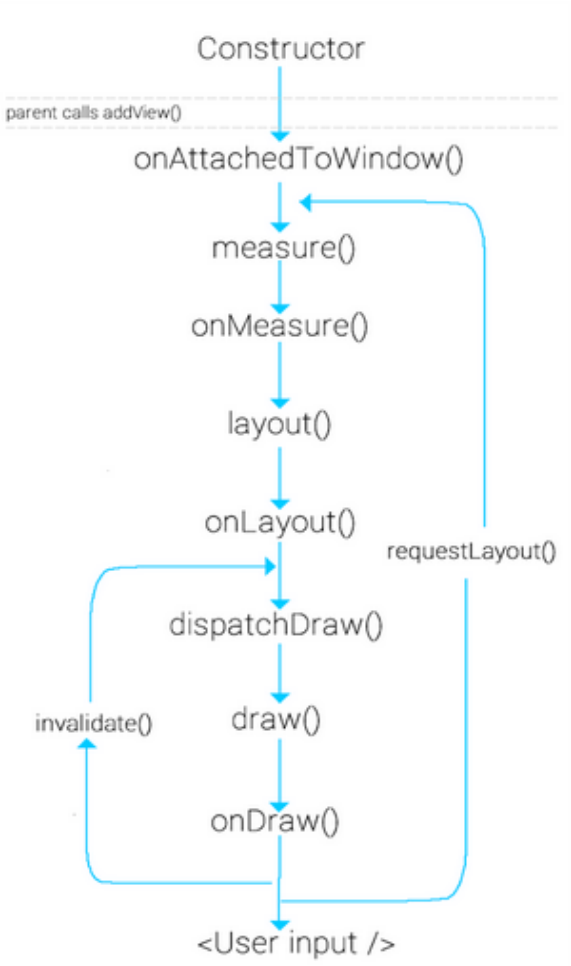
参考：mobile - How do dp, dip, dpi, ppi, pixels and inches relate? - Stack Overflow

41. 长度和字体的推荐单位

长度推荐dp（Density-independent Pixels），字号大小推荐sp（Scale-independent Pixels）。

42. Android View绘制流程





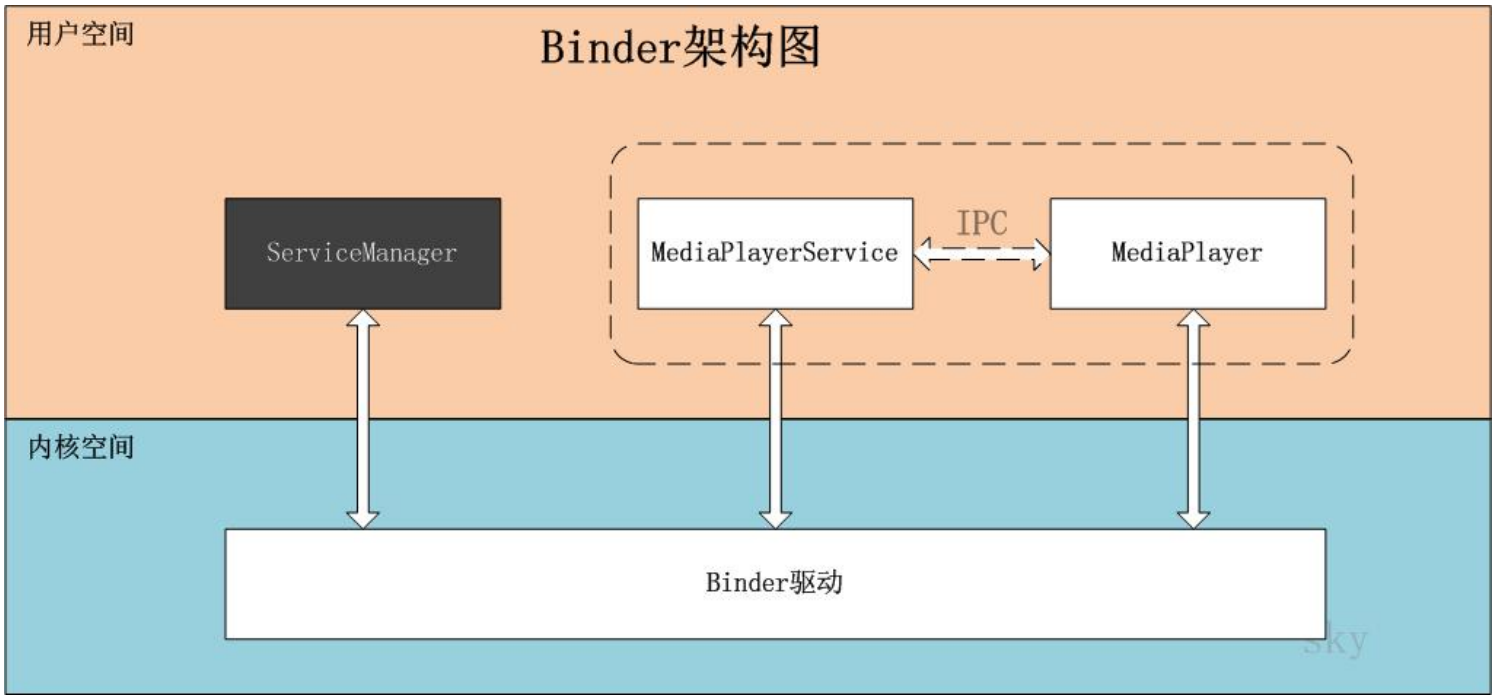
参考：[android-open-project-analysis/tech/viewdrawflow at master · android-cn/android-open-project-analysis]

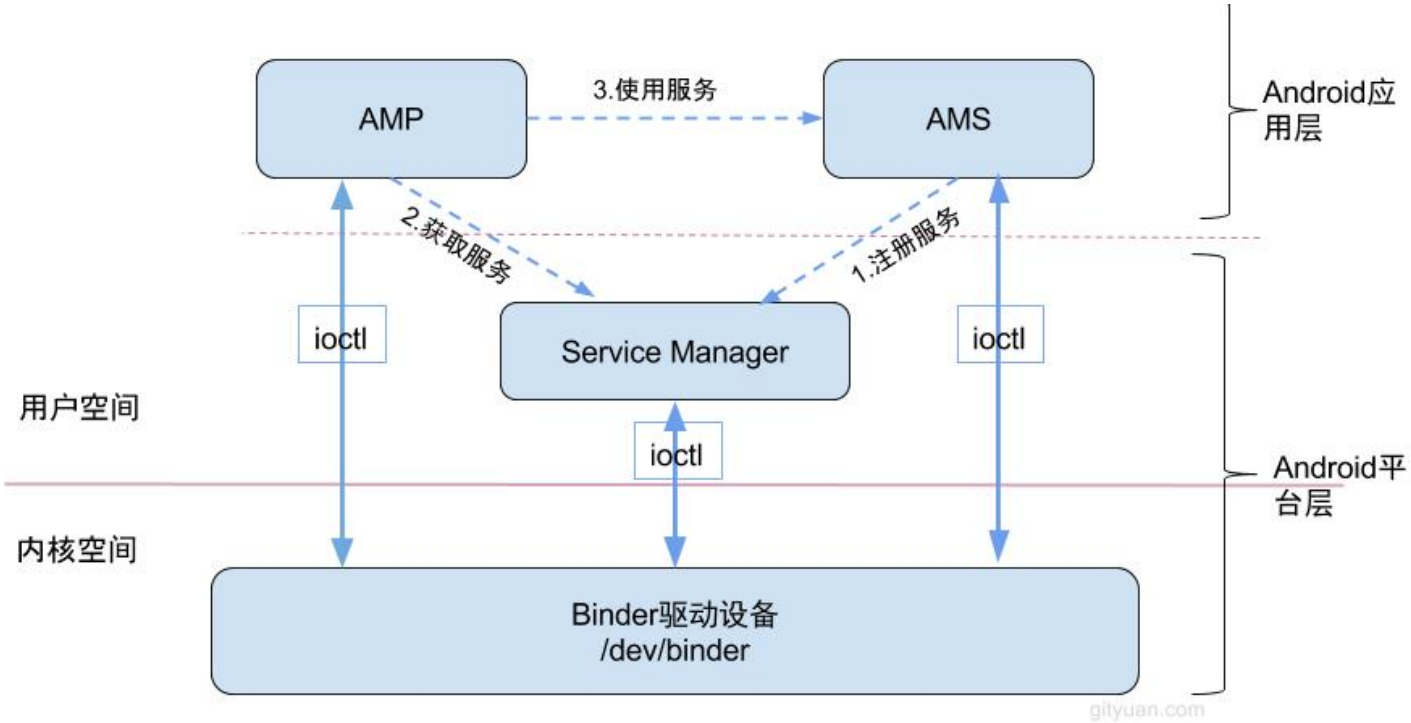
参考：[Android Layout绘制]

43. ListView优化

- 复用convertView：用以避免重复创建View，重复创建View代价较大，而且如果重用view不改变宽高，重用View可以减少重新分配缓存造成的内存频繁分配/回收。
- 使用View Holder模式：findViewById的实现是遍历，如果你定义的View越复杂代价越大。 Google推荐的做法是用ViewHolder，然后保存在view的tag中。现在RecyclerView也是强制使用ViewHolder了。
- 分批加载与分页加载相结合：不需要一次等待好几分钟把数据都加载完再在ListView上显示。
- 使用异步线程加载图片
- 在快速滑动时不要加载图片
- 使用RecyclerView

44. Android Binder机制





1. Server进程启动之后，会进入中断等待状态，等待Client的请求。
2. 当Client需要和Server通信时，会将请求发送给Binder驱动。
3. Binder驱动收到请求之后，会唤醒Server进程。
4. 接着，Binder驱动还会反馈信息给Client，告诉Client：它发送给Binder驱动的请求，Binder驱动已经收到。
5. Client将请求发送成功之后，就进入等待状态。等待Server的回复。
6. Binder驱动唤醒Server之后，就将请求转发给Server进程。
7. Server进程解析出请求内容，并将回复内容发送给Binder驱动。
8. Binder驱动收到回复之后，唤醒Client进程。

9. 接着，Binder驱动还会反馈信息给Server，告诉Server：它发送给Binder驱动的回复，Binder驱动已经收到。
10. Server将回复发送成功之后，再次进入等待状态，等待Client的请求。
11. 最后，Binder驱动将回复转发给Client。

45. Binder机制优点

- 性能：Binder数据拷贝只需要一次，而管道、消息队列、Socket都需要2次，但共享内存方式一次内存拷贝都不需要；从性能角度看，Binder性能仅次于共享内存。
- 稳定性：Binder是基于C/S架构的，Server端与Client端相对独立，稳定性较好。
- 安全性：传统Linux IPC的接收方无法获得对方进程可靠的UID/PID，从而无法鉴别对方身份。Android系统中对外只暴露Client端，Client端将任务发送给Server端，Server端会根据权限控制策略，判断UID/PID是否满足访问权限，目前权限控制很多时候是通过弹出权限询问对话框，让用户选择是否运行。

46. AsyncTask简介

包含4个方法：

- `onPreExecute()`：UI线程；
- `doInBackground(Params...)`：非UI线程；
- `onProgressUpdate(Progress...)`：UI线程；
- `onPostExecute(Result)`：UI线程。

原理：

- 线程池；
- 单例模式；
- `mainLooper()`；
- 串行。

47. 为什么Handler需要声明为static?

所有发送到消息队列的消息Message都会拥有一个对Handler的引用，在java里，非静态内部类和匿名类都会潜在的引用它们所属的外部类。但是，静态内部类却不会。当Activity结束（finish）时，里面的延时消息在得到处理前，会一直保存在主线程的消息队列里持续10分钟。这条消息持有对handler的引用，而handler又持有对其外部类（在这里，即SampleActivity）的潜在引用。这条引用关系会一直保持直到消息得到处理，从而，这阻止了SampleActivity被垃圾回收器回收，同时造成应用程序的泄漏。

48. 广播注册后不解除注册会有什么问题?

内存泄漏。系统会保留Receiver的引用。

49. 自定义View

49.1. 实现步骤

1. 继承View类或其子类；
2. 复写view中的一些函数；
3. 为自定义View类增加属性（两种方式）；
4. 绘制控件（导入布局）；
5. 响应用户事件；
6. 定义回调函数（根据自己需求来选择）。

50. 需要被重写的方法

- `onDraw()`：view中`onDraw()`是个空函数，也就是说具体的视图都要覆写该函数来实现自己的绘制。对于ViewGroup则不需要实现该函数，因为作为容器是"没有内容"的（但必须实现`dispatchDraw()`函数，告诉子view绘制自己）。
- `onLayout()`：主要是为viewGroup类型布局子视图用的，在View中这个函数为空函数。
- `onMeasure()`：用于计算视图大小（即长和宽）的方式，并通过`setMeasuredDimension(width, height)`保存计算结果。
- `onTouchEvent()`：定义触屏事件来响应用户操作。

51. Parcelable和Serializable的区别

Serializable仅需实现Serializable接口。缺点是使用了反射，序列化的过程较慢。这种机制会在序列化的时候创建许多的临时对象，容易触发垃圾回收。

Parcelable需要实现Parcelable接口，但序列化的过程已经提前确定，所以运行速度快。

52. Android中的内存泄漏

1. 查询数据库没有关闭游标。
2. 构造Adapter时，没有使用缓存的convertView。
3. Bitmap对象不再使用时调用`recycle()`释放内存。
4. 无用时没有释放对象的引用。
5. 在Activity中使用非静态的内部类，并开启一个长时间运行的线程，因为内部类持有Activity的引用，会导致Activity本来可以被GC时却长期得不到回收。
6. 使用Handler处理消息前，Activity通过例如`finish()`退出，导致内存泄漏。
7. 动态注册广播在Activity销毁前没有`unregisterReceiver()`。

53. MVC和MVP的区别

我们都知道MVP是从经典的模式MVC演变而来，它们的基本思想有相通的地方：Controller/Presenter负责逻辑的处理，Model提供数据，View负责显示。作为一种新的模式，MVP与MVC有着一个重大的区别：在MVP中View并不直接使用Model，它们之间的通信是通过Presenter（MVC中的Controller）来进行的，所有的交互都发生在Presenter内部，而在MVC中View会直接从Model中读取数据而不是通过Controller。

54. 内存泄露检测有什么好方法?

1. DDMS Heap发现内存泄露dataObject totalSize的大小，是否稳定在一个范围内，如果操作程序，不断增加，说明内存泄露。
2. 使用Heap Tool进行内存快照前后对比BlankActivity手动触发GC进行前后对比，对象是否被及时回收。

55. Android里面为什么要设计出Bundle而不是直接用Map结构

Map里实现了Serializable接口，而在Bundle实现了Parcelable的接口。

56. 在Android的MVP架构中，使用了什么设计模式

- Observer模式：通过EventBus实现订阅者，发布者的功能，实现Model与Presenter的交互。
- Proxy模式：View保持对Presenter的引用，通过Presenter代理，进行交互操作。

57. Android动画类型

- 属性动画（Property Animation）：是Android 3.0之后推出的，其机制不再是针对 View 来设计的，也不限于只能实现移动、缩放、旋转和淡入这几种简单的动画操作，同时也不再只是一种视觉上的动画效果。属性动画实际上是一种在一定时间段内不断修改某个对象的某个属性值的机制。
- 视图动画（View Animation）：
    - 补间动画（Tween animation）：是操作某一个控件让其展现出旋转、渐变、移动、缩放的一种转换过程。是一种视觉上的变化，不是真正位置上的变化。只能运用在 View 对象上，并且功能相对来说较为局限。例如：旋转动画只能够在x、y轴进行，而不能在z轴放心进行旋转。因此，补间动画通常用于执行一些比较简单的动画。
    - 渐变动画（AlphaAnimation）；
    - 缩放动画（ScaleAnimation）；
    - 位移动画（TranslateAnimation）；
    - 旋转动画（RotateAnimation）。
    - 帧动画（Frame animation）：帧动画是一系列图片按照一定的顺序展示的过程，和放电影的机制相似，它的原理是在一定的时间段内切换多张有细微差异的图片从而达到动画的效果。由于是一帧一帧加载，所以需要较多的图片。从而增大 APK 的大小，不过 Frame 动画可以实现一些比较难的效果，例如：等待的环形进度。

58. ANR和FC的区别

- ANR（Application Not Responding）：主线程阻塞。
- FC（Forced Close）：内存耗尽，堆栈溢出，运行时错误等。

59. Android中的菜单

59.1. 选项菜单（Options menu）

在选项菜单中，您应当包括与当前Activity上下文相关的操作和其他选项，如"搜索"、"撰写电子邮件"和"设置"。

- 要为Activity指定选项菜单，请重写 `onCreateOptionsMenu()` 。
- 此外，您还可以使用 `add()` 添加菜单项，并使用 `findItem()` 检索项目，以便使用MenuItem API修改其属性。
- 系统将在启动Activity时调用 `onCreateOptionsMenu()` ，以便向应用栏显示项目。
- 用户从选项菜单中选择项目时，系统将调用Activity的 `onOptionsItemSelected()` 方法。此方法将传递所选的MenuItem。您可以通过调用 `getItemId()` 方法来识别项目，该方法将返回菜单项的唯一ID。
- 系统调用 `onCreateOptionsMenu()` 后，将保留您填充的Menu实例。除非菜单由于某些原因而失效，否则不会再次调用 `onCreateOptionsMenu()` 。
- 如需根据在Activity生命周期中发生的事件修改选项菜单，则可通过 `onPrepareOptionsMenu()` 方法执行此操作。此方法向您传递Menu对象（因为该对象目前存在），以便您能够对其进行修改，如添加、移除或禁用项目。
- 当菜单项显示在应用栏中时，选项菜单被视为始终处于打开状态。 发生事件时，如果您要执行菜单更新，则必须调用 `invalidateOptionsMenu()` 来请求系统调用 `onPrepareOptionsMenu()` 。

59.2. 上下文菜单（Contextual Menus）

59.2.1. 浮动上下文菜单（floating context menu）

用户长按（按住）一个声明支持上下文菜单的视图时，菜单显示为菜单项的浮动列表（类似于对话框）。

- 通过调用 `registerForContextMenu()` ，注册应与上下文菜单关联的View并将其传递给View。
- 在Activity或Fragment中实现 `onCreateContextMenu()` 方法。
- 实现 `onContextItemSelected()` 。

上下文操作模式（contextual action mode）

上下文操作模式是 ActionMode 的一种系统实现，它将用户交互的重点转到执行上下文操作上。用户通过选择项目启用此模式时，屏幕顶部将出现一个"上下文操作栏"，显示用户可对当前所选项执行的操作。 启用此模式后，用户可以选择多个项目（若您允许）、取消选择项目以及继续在 Activity 内导航（在您允许的最大范围内）。

- 实现ActionMode.Callback接口。在其回调方法中，您既可以为上下文操作栏指定操作，又可以响应操作项目的点击事件，还可以处理操作模式的其他生命周期事件。

- 当需要显示操作栏时（例如，用户长按视图），请调用 `startActionMode()` 。

59.3. 弹出菜单（Popup Menu）

PopupMenu 是锚定到 View 的模态菜单。如果空间足够，它将显示在定位视图下方，否则显示在其上方。

- 实例化PopupMenu及其构造函数，该函数将提取当前应用的Context以及菜单应锚定到的View。
- 使用MenuInflater将菜单资源扩充到 `PopupMenu.getMenu()` 返回的Menu对象中。

- 调用 `PopupMenu.show()` 。

60. BaseAdapter中需要重载的方法

最基本的：

- `int getCount ()`：How many items are in the data set represented by this Adapter.
- `Object getItem (int position)`：Get the data item associated with the specified position in the data set.
- `long getItemId (int position)`：Get the row id associated with the specified position in the list.
- `View getView (int position, View convertView, ViewGroup parent)`：Get a View that displays the data at the specified position in the data set.

如果有多种View：

- `int getItemViewType (int position)`：Get the type of View that will be created by getView(int, View, ViewGroup) for the specified item.
- `int getViewTypeCount ()`：Returns the number of types of Views that will be created by getView(int, View, ViewGroup).

61. Android数字签名要点

- 所有的应用程序都必须有数字证书，Android系统不会安装一个没有数字证书的应用程序。
- Android程序包使用的数字证书可以是自签名的，不需要一个权威的数字证书机构签名认证。
- 如果要正式发布一个Android应用，必须使用一个合适的私钥生成的数字证书来给程序签名，而不能使调试证书来发布。
- 数字证书都是有有效期的，Android只是在应用程序安装的时候才会检查证书的有效期。如果程序已经安装在系统中，即使证书过期也不会影响程序的正常功能。

62. 使用相同数字签名的原因

- 应用升级：当系统安装应用的更新时，它会比较新版本和现有版本中的证书。如果证书匹配，则系统允许更新。如果您使用不同的证书签署新版本，则必须为应用分配另一个软件包名称----在此情况下，用户将新版本作为全新应用安装。
- 应用模块化：Android允许通过相同证书签署的多个APK在同一个进程中运行（如果应用请求这样），以便系统将它们视为单个应用。通过此方式，您可以在模块中部署您的应用，且用户可以独立更新每个模块。
- 通过权限共享代码/数据：Android提供基于签名的权限执行，以便应用可以将功能展示给使用指定证书签署的另一应用。通过使用同一个证书签署多个APK并使用基于签名的权限检查功能，您的应用可采用安全的方式共享代码和数据。

63. Theme和Sytle

63.1. Style

样式是指为View或窗口指定外观和格式的属性集合。样式可以指定高度、填充、字体颜色、字号、背景色等许多属性。 样式是在与指定布局的XML不同的XML资源中进行定义。

- 要创建一组样式，请在您的项目的 `res/values/` 目录中保存一个XML文件。
- 该XML文件的根节点必须是 `<resources>` 。
- 对于您想创建的每个样式，向该文件添加一个 `<style>` 元素，该元素带有对样式进行唯一标识的 `name` 属性（该属性为必需属性）。
- 然后为该样式的每个属性添加一个 `<item>` 元素，该元素带有声明样式属性以及属性值的 `name` （该属性为必需属性）。
- 根据样式属性， `<item>` 的值可以是关键字字符串、十六进制颜色值、对另一资源类型的引用或其他值。
- 您可以通过 `<style>` 元素中的 `parent` 属性指定应作为您的样式所继承属性来源的样式。
- 当您对布局中的单个View应用样式时，该样式定义的属性只应用于该View。如果对ViewGroup应用样式，子View元素将不会继承样式属性----只有被您直接应用样式的元素才会应用其属性。

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="CodeFont" parent="@android:style/TextAppearance.Medium">
        <item name="android:layout_width">fill_parent</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:textColor">#00FF00</item>
        <item name="android:typeface">monospace</item>
    </style>
</resources>
```

```
<TextView
    style="@style/CodeFont"
    android:text="@string/hello" />
```

63.2. Theme

主题是指对整个Activity或应用而不是对单个View（如上例所示）应用的样式。以主题形式应用样式时，Activity或应用中的每个视图都将应用其支持的每个样式属性。例如，您可以Activity主题形式应用同一CodeFont样式，之后该Activity内的所有文本都将具有绿色固定宽度字体。

- 在XML中定义您想用作Activity或应用主题的样式与定义视图样式的方法完全相同。
- Activity或应用内的每个View都将应用其支持的每个属性。例如，如果您对某个Activity应用前面示例中的CodeFont样式，则所有支持这些文本样式属性的View元素也会应用这些属性。任何不支持这些属性的View都会忽略这些属性。如果某个View仅支持部分属性，将只应用这些属性。

```
<application android:theme="@style/CustomTheme">

<activity android:theme="@android:style/Theme.Dialog">
```

64. Toast的时长设置

Toast的显示时长仅有两种： `LENGTH_SHORT` 和 `LENGTH_LONG` 。

`Toast makeText (Context context, CharSequence text, int duration)` ： duration `int` : How long to

display the message. Either `LENGTH_SHORT` or `LENGTH_LONG` 。

65. 触发ANR的情况

- KeyDispatchTimeout(5 seconds)：按键或触摸事件在特定时间内无响应；
- BroadcastTimeout(10 seconds)：BroadcastReceiver在特定时间内无法处理完成；
- ServiceTimeout(20 seconds)：Service在特定的时间内无法处理完成

66. ServiceConnection的 `onServiceConnected()` 触发条件

- `bindService()` 方法执行成功；
- `onBind()` 方法返回非空IBinder对象。

67. Android虚拟设备不支持的功能

- WLAN
- 蓝牙
- NFC
- SD 卡插入/弹出
- 连接到设备的耳机
- USB

68. RemoteView的应用

- AppWidget
- Notification

69. Android对HashMap做了优化后推出的新的容器类是什么?

69.1. SparseArray

SparseArray比HashMap更省内存，在某些条件下性能更好，主要是因为它避免了对key的自动装箱（int转为Integer类型），它内部则是通过两个数组来进行数据存储的，一个存储key，另外一个存储value，为了优化性能，它内部对数据还采取了压缩的方式来表示稀疏数组的数据，从而节约内存空间。

69.2. ArrayMap

ArrayMap是一个<key,value>映射的数据结构，它设计上更多的是考虑内存的优化，内部是使用两个数组进行数据存储，一个数组记录key的hash值，另外一个数组记录Value值，它和SparseArray一样，也会对key使用二分法进行从小到大排序，在添加、删除、查找数据的时候都是先使用二分查找法得到相应的index，然后通过index来进行添加、查找、删除等操作。

70. Android安全沙盒

- Android操作系统是一种多用户Linux系统，其中的每个应用都是一个不同的用户；
- 默认情况下，系统会为每个应用分配一个唯一的Linux用户ID（该ID仅由系统使用，应用并不知晓）。系统为应用中的所有文件设置权限，使得只有分配给该应用的用户ID才能访问这些文件；
- 每个进程都具有自己的虚拟机 (VM)，因此应用代码是在与其他应用隔离的环境中运行；
- 默认情况下，每个应用都在其自己的Linux进程内运行。Android会在需要执行任何应用组件时启动该进程，然后在不再需要该进程或系统必须为其他应用恢复内存时关闭该进程。

71. `onStartCommand()`有哪些返回值

`onStartCommand()`的返回值用于描述系统应该如何在服务终止的情况下继续运行服务。其值可以为

- `START_NOT_STICKY`：如果系统在`onStartCommand()`返回后终止服务，则除非有挂起Intent要传递，否则系统不会重建服务。这是最安全的选项，可以避免在不必要时以及应用能够轻松重启所有未完成的作业时运行服务。
- `START_STICKY`：如果系统在`onStartCommand()`返回后终止服务，则会重建服务并调用`onStartCommand()`，但不会重新传递最后一个Intent。相反，除非有挂起Intent要启动服务（在这种情况下，将传递这些Intent），否则系统会通过空Intent调用`onStartCommand()`。这适用于不执行命令、但无限期运行并等待作业的媒体播放器（或类似服务）。
- `START_REDELIVER_INTENT`：如果系统在`onStartCommand()`返回后终止服务，则会重建服务，并通过传递给服务的最后一个Intent调用`onStartCommand()`。任何挂起Intent均依次传递。这适用于主动执行应该立即恢复的作业（例如下载文件）的服务。

72. 如何创建绑定服务

建提供绑定的服务时，您必须提供IBinder，用以提供客户端用来与服务进行交互的编程接口。

72.1. 扩展Binder类

如果服务是供您的自有应用专用，并且在与客户端相同的进程中运行（常见情况），则应通过扩展Binder类并从`onBind()`返回它的一个实例来创建接口。客户端收到Binder后，可利用它直接访问Binder实现中乃至Service中可用的公共方法。

1. 在您的服务中，创建一个可满足下列任一要求的Binder实例：

- 包含客户端可调用的公共方法
- 返回当前Service实例，其中包含客户端可调用的公共方法
- 或返回由服务承载的其他类的实例，其中包含客户端可调用的公共方法

1. 从`onBind()`回调方法返回此Binder实例。
2. 在客户端中，从`onServiceConnected()`回调方法接收Binder，并使用提供的方法调用绑定服务。

72.2. 使用Messenger

1. 服务实现一个Handler，由其接收来自客户端的每个调用的回调

2. Handler用于创建Messenger对象（对Handler的引用）
3. Messenger创建一个IBinder，服务通过 `onBind()` 使其返回客户端
4. 客户端使用IBinder将Messenger（引用服务的Handler）实例化，然后使用后者将Message对象发送给服务
5. 服务在其Handler中（具体地讲，是在 `handleMessage()` 方法中）接收每个 Message。

这样，客户端并没有调用服务的“方法”。而客户端传递的“消息”（Message对象）是服务在其Handler中接收的。

73. 如何绑定到服务

应用组件（客户端）可通过调用 `bindService()` 绑定到服务。Android系统随后调用服务的onBind()方法，该方法返回用于与服务交互的IBinder。

绑定是异步的。 `bindService()` 会立即返回，“不会”使IBinder返回客户端。要接收IBinder，客户端必须创建一个ServiceConnection实例，并将其传递给 `bindService()` 。ServiceConnection包括一个回调方法，系统通过调用它来传递IBinder。

1. 实现ServiceConnection。您的实现必须重写两个回调方法： `onServiceConnected()` ：系统会调用该方法以传递服务的onBind()方法返回的IBinder。 `onServiceDisconnected()` ：Android 系统会在与服务的连接意外中断时（例如当服务崩溃或被终止时）调用该方法。当客户端取消绑定时，系统“不会”调用该方法。
2. 调用 `bindService()` ，传递ServiceConnection实现。
3. 当系统调用您的 `onServiceConnected()` 回调方法时，您可以使用接口定义的方法开始调用服务。
4. 要断开与服务的连接，请调用 `unbindService()` 。如果应用在客户端仍绑定到服务时销毁客户端，则销毁会导致客户端取消绑定。更好的做法是在客户端与服务交互完成后立即取消绑定客户端。

74. Android支持的屏幕密度

- ldpi（低）：120dpi
- mdpi（中）：160dpi
- hdpi（高）：240dpi
- xhdpi（超高）：320dpi
- xxhdpi（超超高）：480dpi
- xxxhdpi（超超超高）：640dpi

75. 如何支持多种屏幕

- 在清单中显式声明您的应用支持哪些屏幕尺寸；
- 为不同屏幕尺寸提供不同的布局；
- 为不同屏幕密度提供不同的位图可绘制对象。

76. 什么是资源ID

所有资源ID都在您项目的R类中定义，后者由aapt工具自动生成。

编译应用时，aapt会生成R类，其中包含您的res/目录中所有资源的资源ID。每个资源类型都有对应的R子类（例如，R.drawable对应于所有可绘制对象资源），而该类型的每个资源都有对应的静态整型数（例如，R.drawable.icon）。这个整型数就是可用来检索资源的资源ID。

资源ID始终由以下部分组成：

- 资源类型：每个资源都被分到一个“类型”组中，例如string、drawable和layout。
- 资源名称：它是不包括扩展名的文件名；或是XML android:name属性中的值，如果资源是简单值的话（例如字符串）。

77. 如何处理运行时变更

有些设备配置可能会在运行时发生变化（例如屏幕方向、键盘可用性及语言）。发生这种变化时，Android会重启正在运行的Activity（先后调用 `onDestroy()` 和 `onCreate()` ）。

77.1. 在配置变更期间保留对象

如果Activity因配置变更而重启，则可通过保留Fragment来减轻重新初始化Activity的负担。此片段可能包含对您要保留的有状态对象的引用。

1. 扩展Fragment类并声明对有状态对象的引用。
2. 在创建片段后调用 `setRetainInstance(boolean)` 。
3. 将片段添加到Activity。
4. 重启Activity后，使用FragmentManager检索片段。

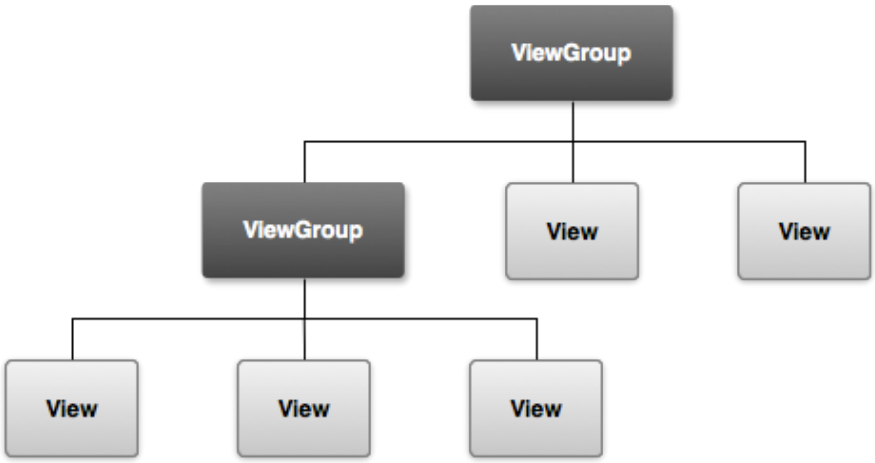
77.2. 自行处理配置变更

要声明由Activity处理配置变更，请在清单文件中编辑相应的 `<activity>` 元素，以包含 `android:configChanges` 属性以及代表要处理的配置的值。 `android:configChanges` 属性的文档中列出了该属性的可能值（最常用的值包括"orientation"和"keyboardHidden"，分别用于避免因屏幕方向和可用键盘改变而导致重启）。

78. AndroidManifest.xml包括哪些内容?

- 为应用的Java软件包命名。软件包名称充当应用的唯一标识符。
- 描述应用的各个组件，包括构成应用的Activity、服务、广播接收器和内容提供程序。它还为实现每个组件的类命名并发布其功能，例如它们可以处理的Intent消息。这些声明向Android系统告知有关组件以及可以启动这些组件的条件的信息。
- 确定托管应用组件的进程。
- 声明应用必须具备哪些权限才能访问API中受保护的部分并与其他应用交互。还声明其他应用与该应用组件交互所需具备的权限。
- 声明应用所需的最低Android API级别。
- 列出应用必须链接到的库。

79. 用户界面如何构成?



Android应用中的所有用户界面元素都是使用View和ViewGroup对象构建而成。View对象用于在屏幕上绘制可供用户交互的内容。ViewGroup对象用于储存其他View（和ViewGroup）对象，以便定义界面的布局。

每个应用组件的用户界面都是使用View和ViewGroup对象的层次结构定义的。每个视图组都是一个用于组织子视图的不可见容器，而子视图可以是输入控件或其他可绘制某一UI部分的小部件。

80. 为什么要回收Bitmap的内存

Bitmap的实例化只能通过BitmapFactory，而Bitmap对象的生成则是通过JNI调用，所以Bitmap包含Java和C两部分内存。Java部分内存可以通过虚拟机自动回收，但C部分则需要手动释放，所以需要显式调用 `recycle()` 方法来释放。

81. 如何优化Bitmap

- 加载合适尺寸的图片；
- 及时回收Bitmap；
- 捕获OOM异常；
- 压缩图片；
- 使用合适的颜色模式。

82. 如何在新进程中创建Activity / Service

在AndroidManifest中，组件元素条目（activity、service、receiver或provider）中设置 `android:process` 属性，此属性可以指定该组件应在哪个进程运行。

每新建一个进程，Application的 `onCreate()` 都将被调用一次。

83. `onActivityResult()` 什么时候会失效?

在 `startActivity()` 为SingleTask时会失效。

> For example, if the activity you are launching uses the singleTask launch mode, it will not run in your task and thus you

> will immediately receive a cancel result.

如果你正加载的activity使用了singleTask的加载模式，它不会在你的栈中运行，而且这样你会马上收到一个取消的结果。即在 `onActivityResult()` 里马上得到一个RESULT_CANCEL。

84. Android崩溃捕获

84.1. Java崩溃捕获

Java提供有UncaughtExceptionHandler接口，该接口含有一个方法： `void uncaughtException(Thread t, Throwable e)` 。

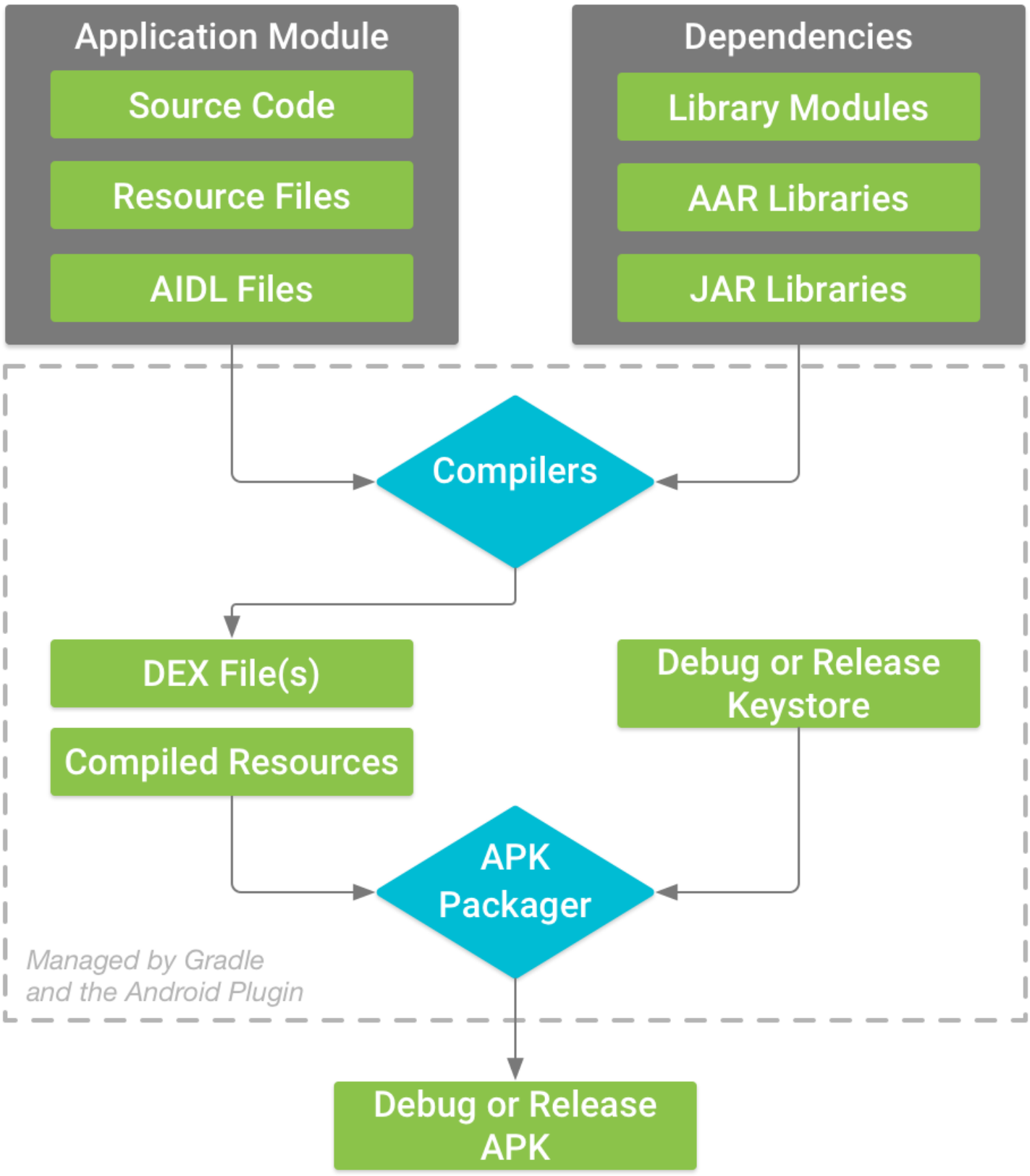
Uncaught异常发生时会终止线程，此时，系统便会通知UncaughtExceptionHandler，调用 `uncaughtException()` 函数。如果该handler没有被显式设置，则会调用对应线程组的默认handler。

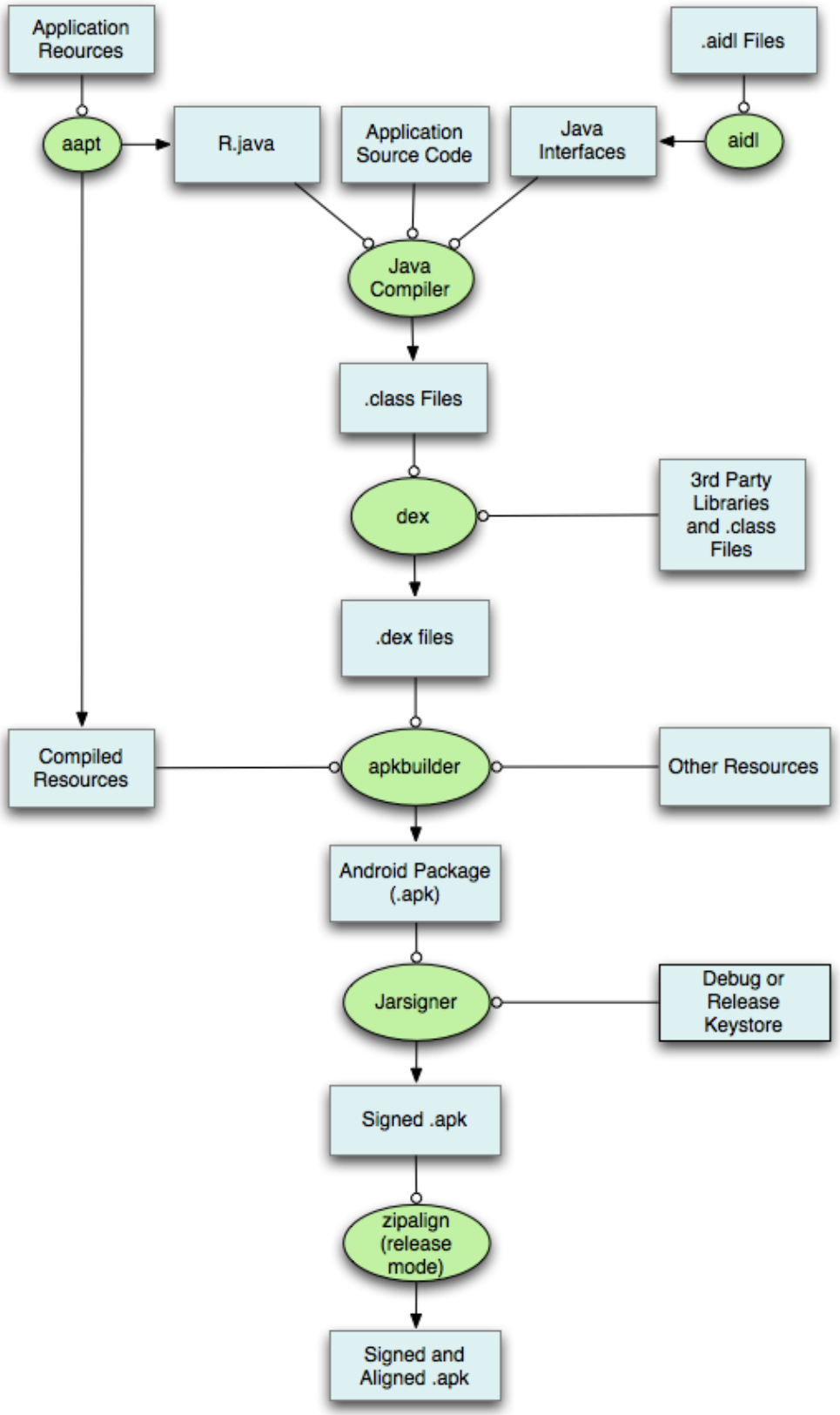
如果要捕获该异常，必须实现UncaughtExceptionHandler，并通过 `public static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler handler)` 进行设置。

84.2. Native崩溃捕获

对Native代码的崩溃，可以通过调用 `int sigaction(int signum,const struct sigaction *act,struct sigaction *oldact))` 注册信号处理函数来完成。

85. Android APP构建流程


Application Module
Source Code
Resource Files
AIDL Files
Dependencies
Library Modules
AAR Libraries
JAR Libraries
Compilers
DEX File(s)
Compiled Resources
Debug or Release Keystore
APK Packager
Managed by Gradle and the Android Plugin
Debug or Release APK



典型的Android APP构建流程：

1. Java编译器对工程本身的java代码进行编译，这些java代码有三个来源：app的源代码，由资源文件生成的R文件（aapt工

具），以及有aidl文件生成的java接口文件（aidl工具）。产出为.class文件。
2. class文件和依赖的三方库文件通过dex工具生成Delvik虚拟机可执行的.dex文件，可能有一个或多个，包含了所有的class信息，包括项目自身的class和依赖的class。产出为.dex文件。
3. apkbuilder工具将.dex文件和编译后的资源文件生成未经签名对齐的apk文件。这里编译后的资源文件包括两部分，一是由aapt编译产生的编译后的资源文件，二是依赖的三方库里的资源文件。产出为未经签名的.apk文件。
4. 分别由Jarsigner和zipalign对apk文件进行签名和对齐，生成最终的apk文件。

86. class文件与.dex文件的区别

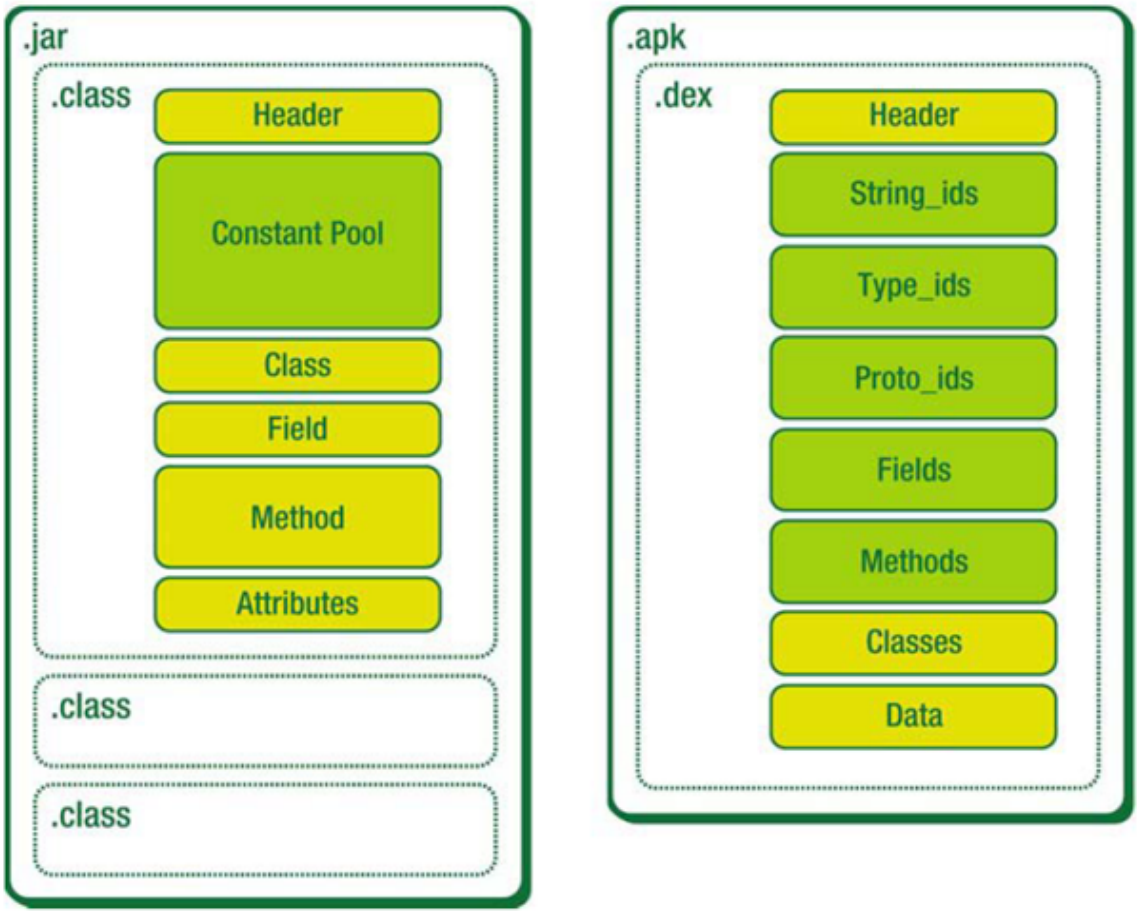


Figure 3-2. *Class file vs DEX file*

87. 65535问题

87.1. 原因

单个dex文件中，method个数采用使用原生类型short来索引，即2个字节最多65536个method，field、class的个数也均有此限制。

87.2. 解决方法

build.gradle中配置 `multiDexEnabled true` ，将dex分包。

88. Dalvik与JVM的区别

- JVM可以执行的文件是.class结尾的字节码文件，而Dalvik执行的是dex文件（不符合JVM规范）。
- Dalvik基于寄存器，而JVM基于栈。
- Dalvik负责进程隔离和线程管理，每一个Android应用在底层都会对应一个独立的Dalvik虚拟机实例，其代码在虚拟机的解释下得以执行。

除此之外：

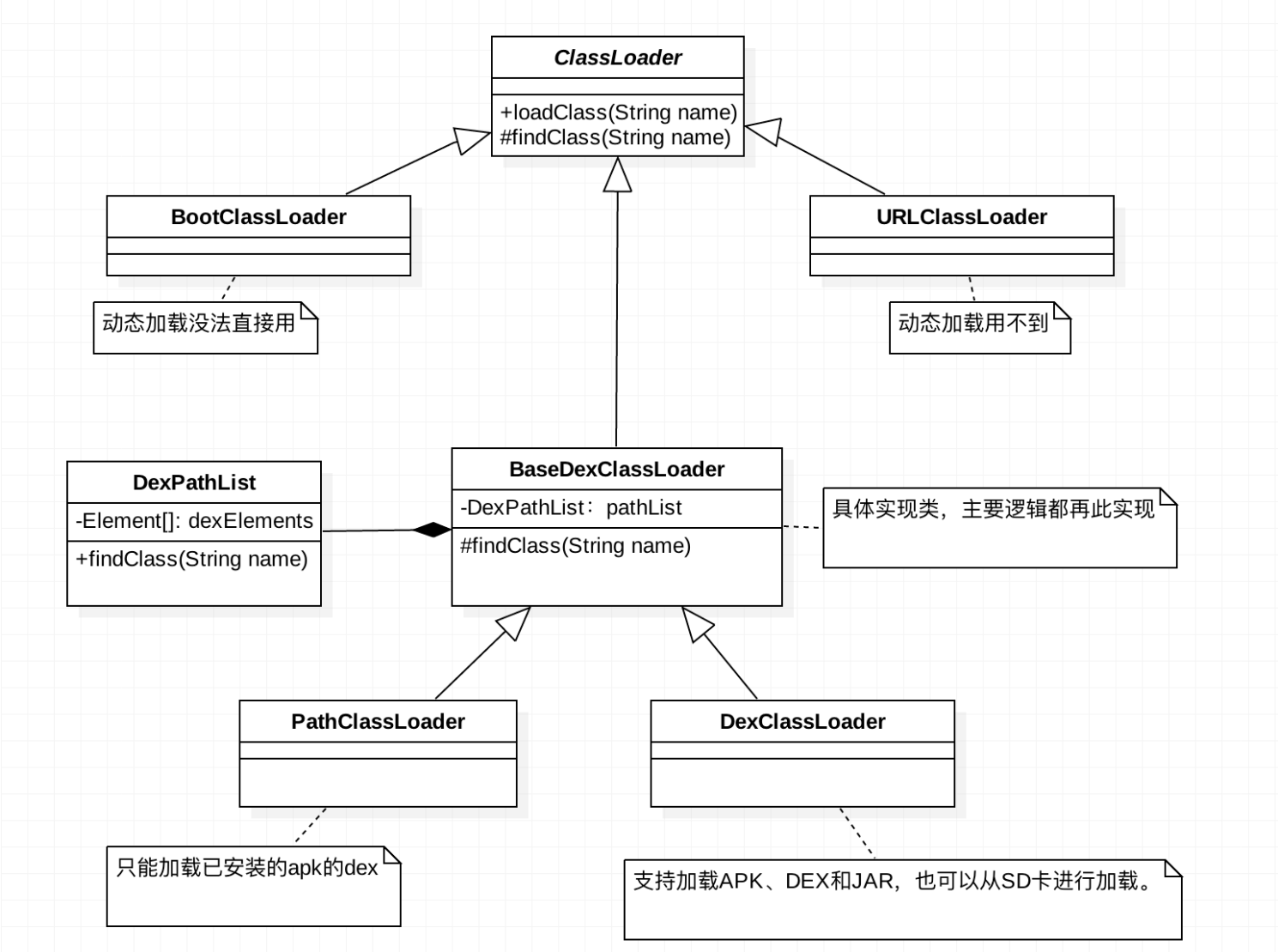
- 有一个特殊的虚拟机进程Zygote，他是虚拟机实例的孵化器。它在系统启动的时候就会产生，它会完成虚拟机的初始化、库的加载、预制类库和初始化的操作。如果系统需要一个新的虚拟机实例，它会迅速复制自身，以最快的速度提供给系统。

89. ART相对Dalvik的优化

- AOT替换JIT：使用AOT直接在安装时用dex2oat将其完全翻译成native代码。
- GC性能提升：并行GC。
- 提升内存效率：专门开辟内存存放large object，因为large object移动成本太大；引入moving collector技术，将不连续的物理内存块对齐，解决内存碎片化问题。

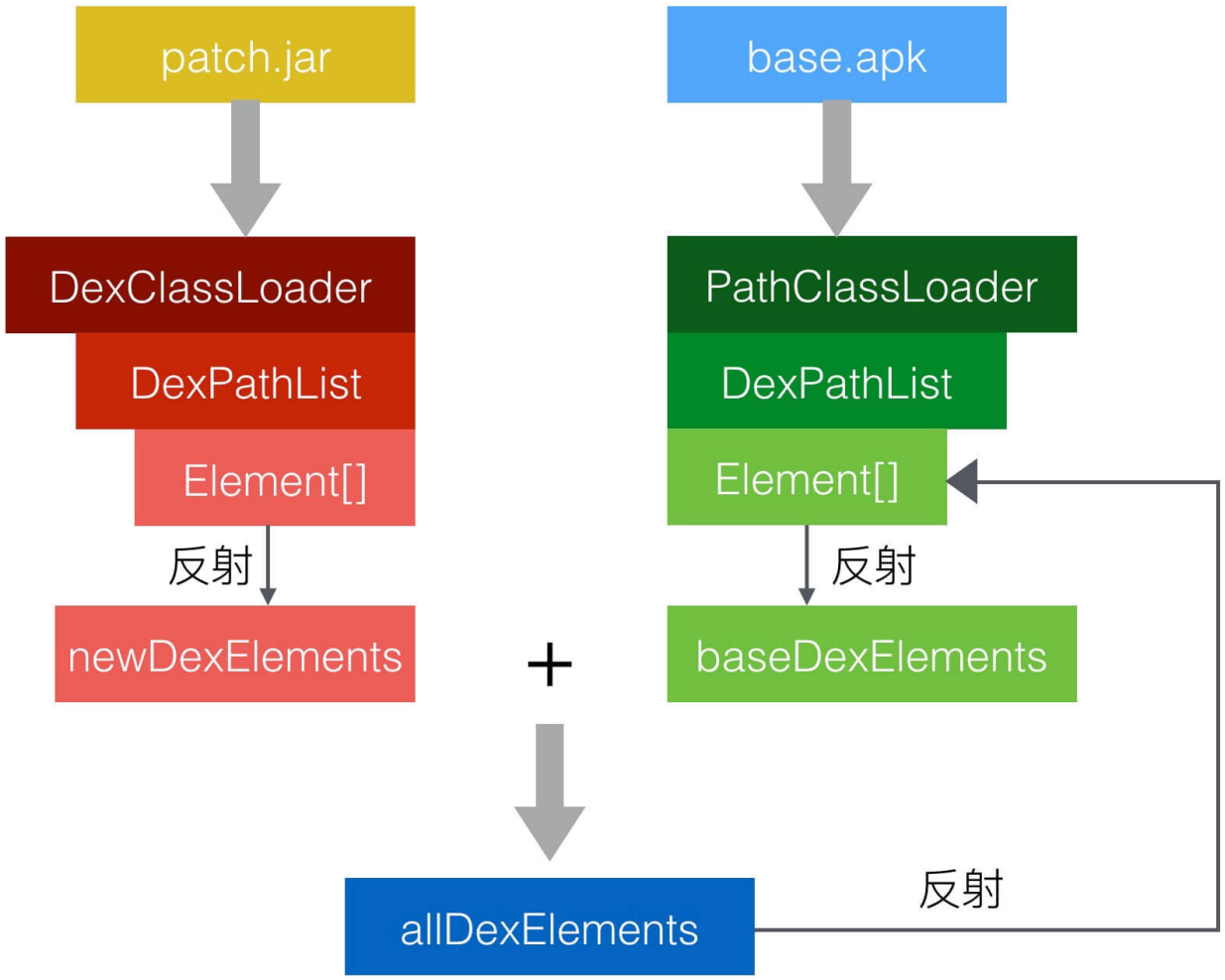
参考：ART and Dalvik | Android Open Source Project

90. Android中的ClassLoader



- 在Android中，App安装到手机后，apk里面的class.dex中的class均是通过PathClassLoader来加载的。
- DexClassLoader可以用来加载SD卡上加载包含class.dex的.jar和.apk文件。
- DexClassLoader和PathClassLoader的基类BaseDexClassLoader查找class是通过其内部的 `DexPathList pathList` 来查找的。
- DexPathList内部有一个 `Element[] dexElements` 数组，其 `findClass()` 方法的实现就是遍历该数组，查找class，一旦找到需要的类，就直接返回，停止遍历。

91. ClassLoader方式实现热修复



主要步骤：

1. 假设MainActivity中有一个方法 `showMsg()` 需要修复。
2. 修复 `showMsg()` 方法，制作补丁包patch.jar，该patch.jar文件中就包含已经修复了的dex文件。
3. 在Application的 `onCreate()` 方法中检测是否已经下载好补丁包，如果存在补丁包，就通过DexClassLoader加载patch.jar，然后通过反射拿到DexClassLoader中的 DexPathList对象，进而拿到 `Element[] dexElements` 数组，这里标记该Element数组为 `newDexElements` 。
4. 还是通过反射，拿到App默认的ClassLoader即PathClassLoader的DexPathList对象，进而拿到Element数组，这里标记下该数组为 `baseDexElements` 。
5. 将 `newDexElements` 和 `baseDexElements` 合成一个新的数组 `allDexElements` ，且保证 `newDexElements` 中的值在 `allDexElements` 数组的最前面。
6. 然后还是通过通过反射，将合成的Element数组设置给PathClassLoader的DexPathList对象。
7. 在Application完成初始化之后，会开始加载MainActivity，加载过程就是通过DexPathList对象的 `findClass()` 方法来完成的，会从头开始遍历其Element数组，会优先查找到之前插入的补丁包中的dexFile，而原apk中的则不会查找到，因此就实现了热修复的目的。

参考：热修复实现：ClassLoader 方式的实现

92. AsyncTask需要在主线程中实例化吗?

92.1. API 16之前

AsyncTask的静态Handler创建和初始化时默认采用的是当前现场的Looper。若子线程无Looper，则会出错；若有Looper则会导致处理消息时无法在主线程执行，出错。所以AsyncTask必须在主线程实例化。

92.2. API 16及之后，API 22之前

在ActivityThread的 `main()` 中直接调用了 `AsyncTask.init()` ，保证Handler在主线程实例化。所以AsyncTask不需要在主线程实例化。

92.3. API 22及之后

不再在ActivityThread的 `main()` 中调用。AsyncTask通过 `getMainLooper()` 获得主线程Looper。所以AsyncTask不需要在主线程实例化。

93. Android消息处理机制

Android消息处理机制主要涉及4个类：Looper、Handler、MessageQueue和Message。

93.1. Looper

Looper的使用：

```
public class LooperThread extends Thread {
    public Handler mHandler;

    public void run() {
        Looper.prepare();

        mHandler = new Handler() {
            public void handleMessage(Message msg) {
                // process incoming messages here
            }
        };

        Looper.loop();
    }
}
```

Looper概览：

```
public class Looper {
    static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<>();
    final MessageQueue mQueue;

    private Looper(boolean quitAllowed) {
        mQueue = new MessageQueue(quitAllowed);
    }

    public static void prepare() {
        prepare(true);
    }

    private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be created per thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));
    }

    public static Looper myLooper() {
        return sThreadLocal.get();
    }

    public static void loop() {
        for (; ; ) {
            Message msg = queue.next(); // might block
            msg.target.dispatchMessage(msg);
            msg.recycleUnchecked();
        }
    }
}
```

`prepare()` 会检查此线程是否已经存在Looper，随后便会实例化一个Looper（创建一个MessageQueue），并将此Looper设置为此线程的ThreadLocal变量，这样完成Looper和线程的绑定。

`loop()` 即进从MessageQueue取消息并处理的死循环。

93.2. Handler

Handler概览：

```
public class Handler {
    final Looper mLooper;
    final MessageQueue mQueue;
    final Handler.Callback mCallback;

    public Handler() {
        mLooper = Looper.myLooper();
        if (mLooper == null) {
```

```
            throw new RuntimeException(
                    "Can't create handler inside thread that has not called Looper.prepare()");
        }
        mQueue = mLooper.mQueue;
    }

    public void handleMessage(Message msg) {
    }

    public void dispatchMessage(Message msg) {
        if (msg.callback != null) {
            handleCallback(msg);
        } else {
            handleMessage(msg);
        }
    }

    public final Message obtainMessage() {
        return Message.obtain(this);
    }

    public final Message obtainMessage(int what, Object obj) {
        return Message.obtain(this, what, obj);
    }

    private static Message getPostMessage(Runnable r) {
        Message m = Message.obtain();
        m.callback = r;
        return m;
    }

    public final boolean post(Runnable r) {
        return sendMessageDelayed(getPostMessage(r), 0);
    }

    public final boolean sendMessage(Message msg) {
        return sendMessageDelayed(msg, 0);
    }

    public final boolean sendMessageDelayed(Message msg, long delayMillis) {
        return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
    }

    public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
        MessageQueue queue = mQueue;
        return enqueueMessage(queue, msg, uptimeMillis);
    }

    private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
        msg.target = this;
        return queue.enqueueMessage(msg, uptimeMillis);
```

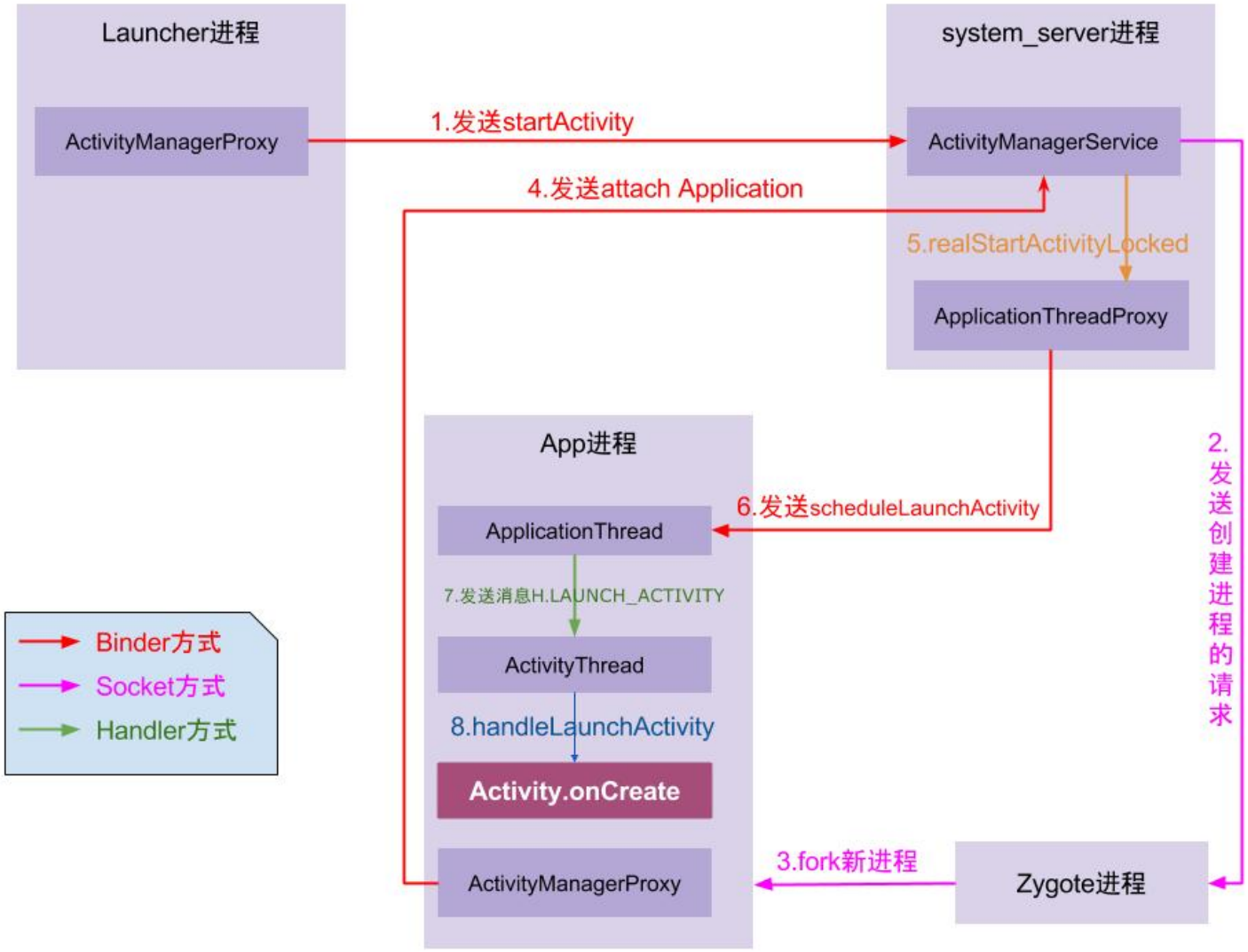
```
    }

    private static void handleCallback(Message message) {
        message.callback.run();
    }
}
```

一般的用法是对 `mHandler` 调用 `obtainMessage()` 获得Message并设置参数后通过 `sendMessage()` 发送出去。在 `obtainMessage()` 和 `sendMessage()` 都会设置Message的Handler为本Handler（Message的 `target` 变量），若需向Message传递Runnable对象，则会在Message的 `callback` 变量中记录。

`sendMessage()` 最终会调用MessageQueue的 `enqueueMessage()` 方法，将此Message绑定到对应Looper对应的MessageQueue上。而Looper中收到Message后，会调用Message的 `target` 变量（即Handler）的 `dispatchMessage()` 方法。对于普通Message， `dispatchMessage()` 又会去调用 `handleMessage()` 方法，而这个方法会被用户重载，所以会执行 `mHandler` 中指定的代码。

参考：android的消息处理机制（图+源码分析）——Looper,Handler,Message - CodingMyWorld - 博客园

94. `startActivity()` 执行流程



- 当Activity的目标进程不存在时，会首先创建进程。
- Activity Manager Service（AMS）向目标进程的主线程发送 `LAUNCH_ACTIVITY`，目标进程通过反射创建目标Activity，然后进入 `onCreate()` 生命周期。

参考：[startActivity启动过程分析 - Gityuan博客 | 袁辉辉博客]