**1. Activity与Fragment的生命周期。**

答案：

Activity：



Activity生命周期须知：

（1）onStart和onResume的区别：onStart实际上表示Activity已经可见了，只是我们还看不到还不能交互而已，因为它还处在后台。而onResume表示Activity已经显示到前台可见了，并且可以进行交互。

（2）当用户按下home键，Activity经历onPause-onStop的过程，这时重新进入Activity经历onRestart-onStart-onResume过程；如果按下back键，经历onPause-onStop-onDestroy的过程。

（3）当在当前ActivityA中打开一个新的ActivityB，要注意的是只有A的onPause执行完成后B的onCreate-…过程才能开始，而A的onStop则是在之后才会进行的。所以不应该在onPause中做耗时的操作，应该尽快让B显示出来进行操作才行。

Fragment：



对比：



1. onCreate过程
2. 01-22 15:30:28.091: E/HJJ(10315): Activity &&&& onCreate...
3. 01-22 15:30:28.091: E/HJJ(10315): ArrayListFragment \*\*\*\* onAttach...
4. 01-22 15:30:28.091: E/HJJ(10315): ArrayListFragment \*\*\*\* onCreate...
5. 01-22 15:30:28.115: E/HJJ(10315): ArrayListFragment \*\*\*\* onCreateView...
6. 01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onActivityCreated...
8. onStart过程
9. 01-22 15:30:28.123: E/HJJ(10315): Activity &&&& onStart...
10. 01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onStart...
12. onResume过程
13. 01-22 15:30:28.123: E/HJJ(10315): Activity &&&& onResume...
14. 01-22 15:30:28.123: E/HJJ(10315): ArrayListFragment \*\*\*\* onResume...
16. onPause过程
17. 01-22 15:31:26.748: E/HJJ(10315): ArrayListFragment \*\*\*\* onPause...
18. 01-22 15:31:26.748: E/HJJ(10315): Activity &&&& onPause...
20. onStop过程
21. 01-22 15:31:27.638: E/HJJ(10315): ArrayListFragment \*\*\*\* onStop...
22. 01-22 15:31:27.638: E/HJJ(10315): Activity &&&& onStop...
24. onStart过程
25. 01-22 15:31:57.537: E/HJJ(10315): Activity &&&& onStart...
26. 01-22 15:31:57.537: E/HJJ(10315): ArrayListFragment \*\*\*\* onStart...
28. onResume过程
29. 01-22 15:31:57.537: E/HJJ(10315): Activity &&&& onResume...
30. 01-22 15:31:57.537: E/HJJ(10315): ArrayListFragment \*\*\*\* onResume...
32. onPause过程
33. 01-22 15:32:47.412: E/HJJ(10315): ArrayListFragment \*\*\*\* onPause...
34. 01-22 15:32:47.412: E/HJJ(10315): Activity &&&& onPause...
36. onStop过程
37. 01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onStop...
38. 01-22 15:32:47.865: E/HJJ(10315): Activity &&&& onStop...
40. onDestroy过程
41. 01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDestroyView...
42. 01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDestroy...
43. 01-22 15:32:47.865: E/HJJ(10315): ArrayListFragment \*\*\*\* onDetach...
44. 01-22 15:32:47.865: E/HJJ(10315): Activity &&&& onDestroy...

**2. Acitivty的四中启动模式与特点。**

（1）标准模式（Standard）

系统的默认模式，每次启动一个Activity都会重新创建一个新的Activity实例，不管是否已经存在。并且谁启动了这个Activity，那么这个Activity就会运行在启动它的那个Activity的任务栈中，如果我们用ApplicationContext来启动标准模式的Activity就会报错，因为它没有所谓的任务栈，必须给Activity添加FLAG\_ACTIVITY\_NEW\_TASK标志位。

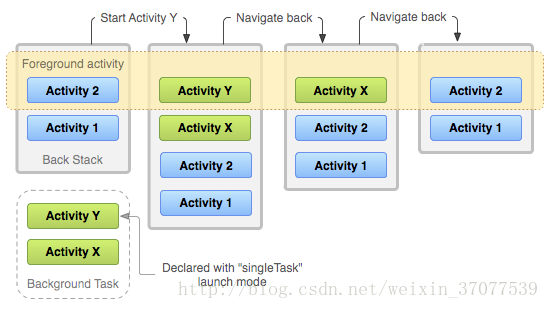
（2）栈顶复用模式（singleTop）

在这个模式中，Activity还是会创建在启动它的Activity的任务栈中，但是如果它已经位于栈顶，那么就不会重复创建，并且它的onNewIntent会被调用，但是要注意它的生命周期方法onCreate等等不会创建。并且如果它并不位于栈顶，如ABC，这时启动B，还是会创建一个新的实例。

（3）栈内复用模式（singleTask）

在这个模式中，比如启动Activity A，首先系统会寻找是否存在A想要的任务栈：

如果存在，就看任务栈中是否有A，如果有就有clearTop的效果，把A推到栈顶，并且调用onNewIntent，即CABD的任务栈，clearTop之后就是CA；如果没有A，就创建A在栈顶。并且要注意一点，如果存在A的任务栈，那么任务栈自动回切换到前台，如下图所示：



Y的任务栈直接被切换到了前台，这点和微信的聊天界面是一样的。

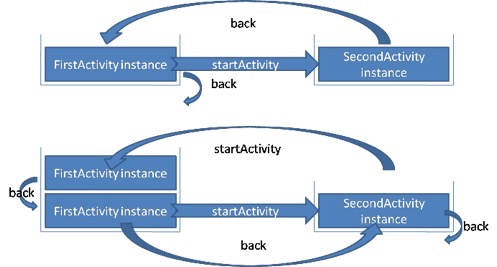
如果不存在，就创建A所需要的任务栈并且创建A入栈。

**使用场景：**

一般来说栈内复用适用于在一个应用里我们希望唯一存在的界面，比如聊天界面，比如微信的聊天界面，我们在一个聊天界面点击了头像进入了新的界面，这个时候来了一条新的消息并且用户点击了消息，这个时候如果是Standard模式就会创建一个新的实例，这样用户在新的界面点击back，用户期望回到的是主界面，而实际上回到了之前的头像界面，这就不符合常规了。如果把聊天设为栈内唯一模式，那么用户点击之后就会回到之前的聊天界面并且做一些改变，保证了聊天界面的唯一性。浏览器的例子也是这样，我们肯定希望浏览器浏览的界面是唯一的，也需要用到这个模式。

（4）单实例模式（singleInstance）

启动时，无论从哪里启动都会给A创建一个唯一的任务栈，后续的创建都不会再创建新的A，除非A被销毁了。



我们看到从MainActivity跳转到SecondActivity时，重新启用了一个新的栈结构，来放置SecondActivity实例，然后按下后退键，再次回到原始栈结构；图中下半部分显示的在SecondActivity中再次跳转到MainActivity，这个时候系统会在原始栈结构中生成一个MainActivity实例，这时点击back发现一个神奇的现象，回到的还是MainActivity，然后再点击一次，**注意，并没有退出，而是回到了SecondActivity**，为什么呢？是因为从SecondActivity跳转到MainActivity的时候，在第一个返回栈中创建了新的实例，而Second所在的成为了后台栈，所以说singleInstance不要用于中间页面，如果用于中间页面，跳转会有问题，比如：A -> B (singleInstance) -> C，完全退出后，再次启动，首先打开的是B。

**使用情况：**

假设，我们希望我们的应用能和另一个应用共享某个Activity的实例。

这样说，可能很难以理解，那么举例来说：   
1.现在我们的手机上有“某某地图”以及我们自己的应用。   
2.我们希望在我们的应用里共享“某某地图”当中的地图功能界面。

那么，假定的操作就应该为，例如：   
1.首先，假设我们在“某某地图”里已经做了一定操作，地图界面被我们定位到了成都市。   
2.我们返回了HOME界面，打开了自己的应用，在自己的应用里打开了”某某地图”的地图界面。   
3.那么，所谓共享该Activity实例，我们要达到的效果就是，当我们打开地图界面，此时地图上显示的位置就应该是我们之前定位到的成都市。而不是地图的初始化显示方位。

那么，显然，通过此前的3种启动模式，我们是实现不了的。因为：   
我们最初在“某某地图”中进行定位时，activity是位于该应用的返回栈里的。   
当我们在自己的应用再次调用地图界面，系统的操作是，在我们自己的应用的返回栈里新建一个地图界面Activity的实例对象。   
所以实际上两个“地图界面”是位于两个不同应用的各自的返回栈里的，两个毫无关联的Activity实例。

MainActivity位于Task“8”当中，当我们在自己的应用MainActivity当中调用SecondActivity，系统新建了返回栈Task“9”，并在该返回栈中放入一个全新的SecondActivity的实例对象。   
此时，当我们再打开SecondActivity本身所在的应用，调用SecondActivity，系统则会复用Task“9”当中的对象，而不会去做其他操作了。   
从而，也就是实现了我们的目的，在两个应用间共享某个Activity。

**3. Activity缓存方法。**

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。

**异常情况的出现分为两种可能：**

（1）资源相关的系统配置发生改变导致Activity被杀死并重新创建

这就是我们所说的横竖屏幕切换的情况，在横竖屏切换时由于系统配置发生了改变，Activity会被销毁，其onPause,onStop,onDestroy均被调用，同时onSaveInstanceState会被调用，它的调用时机发生在onStop之前，和onPause没有时间上的先后关系。

（2）内存不足导致低优先级的Activity被杀死

这种情况onSaveInstanceState会起作用，但是调用的时机不是被杀死时，因为被杀死时怎么会有机会调用呢？注意点2介绍了这个情况。

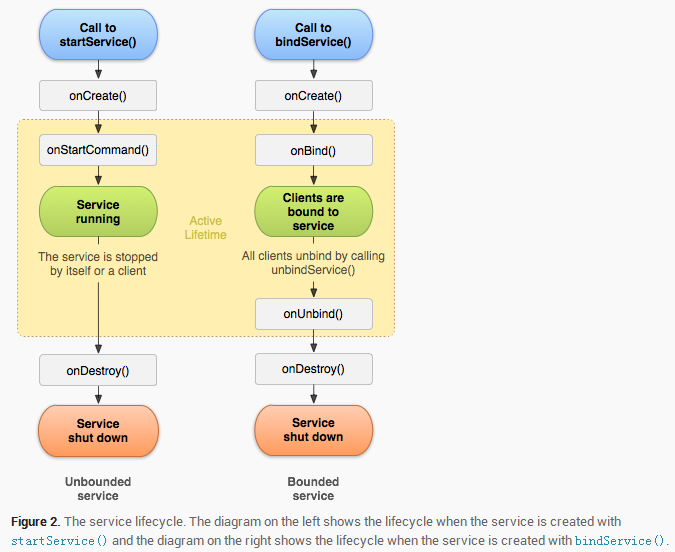
**有两点需要注意：**

（1）使用onRestoreInstanceState来进行之前状态的获取优于onCreate，因为onRestoreInstanceState只要被调用一定说明是有值的（之前已经调用过onSaveInstanceState），但是onCreate无论是否有值一定会调用，所以没有使用onRestoreInstanceState好。

（2）系统正常销毁时，onSaveInstanceState不会被调用，如用户按back键。但是如果Activity有机会重新显示出来，那么onSaveInstanceState一定会调用，因为系统也不知道它是否一定会被杀死。系统不知道你按下HOME后要运行多少其他的程序，**自然也不知道activity A是否会被销毁，因此系统都会调用onSaveInstanceState()，让用户有机会保存某些非永久性的数据。以下几种情况的分析都遵循该原则**

1. 当用户按下HOME键时
2. 长按HOME键，选择运行其他的程序时
3. 锁屏时
4. 从activity A中启动一个新的activity时
5. 屏幕方向切换时

**4. Service的生命周期，两种启动方法，有什么区别。**



**（1）A started service**

　　onCreate, onStartCommand, onBind 和 onDestroy。这几个方法都是回调方法，都是由Android操作系统在合适的时机调用的，并且需要注意的是这几个回调方法都是在主线程中被调用的。

**1、onCreate**: 执行startService方法时，如果Service没有运行的时候会创建该Service并执行Service的onCreate回调方法；如果Service已经处于运行中，那么执行startService方法不会执行Service的onCreate方法。也就是说如果多次执行了Context的startService方法启动Service，**Service方法的onCreate方法只会在第一次创建Service的时候调用一次，以后均不会再次调用！！！！**我们可以在onCreate方法中完成一些Service初始化相关的操作。

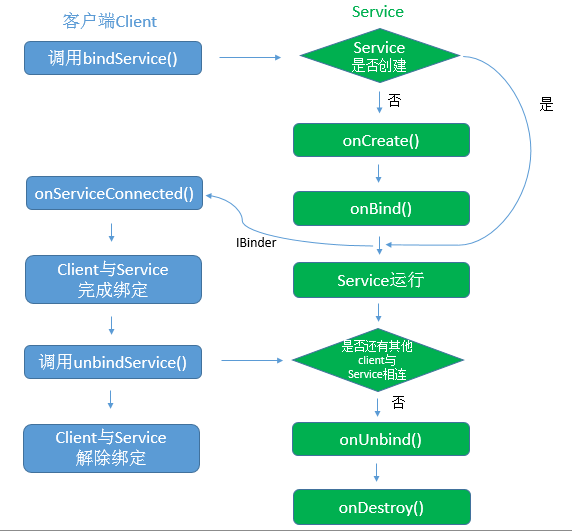
**2、onStartCommand**: 在执行了startService方法之后，有可能会调用Service的onCreate方法，在这之后一定会执行Service的onStartCommand回调方法。也就是说，**如果多次执行了Context的startService方法，那么Service的onStartCommand方法也会相应的多次调用！！！**onStartCommand方法很重要，我们在该方法中根据传入的Intent参数进行实际的操作，比如会在此处创建一个线程用于下载数据或播放音乐等。

**3、onBind**: Service中的onBind方法是抽象方法，所以Service类本身就是抽象类，也就是onBind方法是必须重写的，即使我们用不到。在通过startService使用Service时，我们在重写onBind方法时，只需要将其返回null即可。onBind方法主要是用于给bindService方法调用Service时才会使用到。

**4、onDestroy**: 通过startService方法启动的Service会无限期运行，只有当调用了Context的stopService或在Service内部调用stopSelf方法时，Service才会停止运行并销毁，在销毁的时候会执行Service回调函数。

**（2）A bound service**

被绑定的service是当其他组件（一个客户）调用[bindService()](http://developer.android.com/reference/android/content/Context.html" \l "bindService(android.content.Intent, android.content.ServiceConnection, int))来创建的。客户可以通过一个[IBinder](http://developer.android.com/reference/android/os/IBinder.html)接口和service进行通信。客户可以通过 [unbindService()](http://developer.android.com/reference/android/content/Context.html#unbindService(android.content.ServiceConnection))方法来关闭这种连接。一个service可以同时和多个客户绑定，当多个客户都解除绑定之后，系统会销毁service。context.bindService()->onCreate()->onBind()->Service running-->onUnbind() -> onDestroy() ->Service stop，onBind将返回给客户端一个IBind接口实例，IBind允许客户端回调服务的方法，比如得到Service运行的状态或其他操作。这个时候把调用者（Context，例如Activity）会和Service绑定在一起，当解除绑定时，只有完全没有Client与它绑定时才会调用onUnbind和onDestroy，否则都不会调用。



注意：由于TestService已经处于运行状态，所以ActivityB调用bindService时，不会重新创建TestService的实例，所以也不会执行TestService的onCreate回调方法，由于在ActivityA执行bindService的时候就已经执行了TestService的onBind回调方法而获取IBinder实例，**并且该IBinder实例在所有的client之间是共享的，所以当ActivityB执行bindService的时候，不会执行其onBind回调方法，而是直接获取上次已经获取到的IBinder实例！！！。**并将其作为参数传入ActivityB的ServiceConnection的onServiceConnected方法中，标志着ActivityB与TestService建立了绑定连接，此时有两个客户单client（ActivityA和ActivityB）与TestService绑定。

**（3）注意点：**

**第一：这两条路径并不是完全分开的。**

　　即是说，你可以和一个已经调用了 [startService()](http://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent))而被开启的service进行绑定。

比如，一个后台音乐service可能因调用[startService()](http://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent))方法而被开启了，稍后，可能用户想要控制播放器或者得到一些当前歌曲的信息，可以通过[bindService()](http://developer.android.com/reference/android/content/Context.html#bindService(android.content.Intent, android.content.ServiceConnection, int))将一个activity和service绑定。这种情况下，[**stopService()**](http://developer.android.com/reference/android/content/Context.html#stopService(android.content.Intent))**或**[**stopSelf()**](http://developer.android.com/reference/android/app/Service.html#stopSelf())**实际上并不能停止这个service，除非所有的客户都解除绑定。同样，如果是startService开启的服务及时所有客户解除绑定，如果不调用stopService，也不能停止。**

**第二：除了onStartCommand可以多次调用，其他都不能**

在Service每一次的开启关闭过程中，只有onStart可被多次调用(通过多次startService调用)，其他**onCreate，onBind，onUnbind，onDestory在一个生命周期中只能被调用一次！！！！**

**5. 怎么保证service不被杀死。**

**（1）进程生命周期：**

官方文档告诉我们，Android系统会尽量保持拥有service的进程运行，只要在该service已经被启动(start)或者客户端连接(bindService)到它。当内存不足时，需要保持，拥有service的进程具有较高的优先级。

1． 如果service正在调用onCreate,onStartCommand或者onDestory方法，那么用于当前service的进程则变为前台进程以避免被killed。  
2． 如果当前service已经被启动(start)，拥有它的进程则比那些用户可见的进程优先级低一些，但是比那些不可见的进程更重要，这就意味着service一般不会被killed.  
3． **如果客户端已经连接到service (bindService),那么拥有Service的进程则拥有最高的优先级，可以认为service是可见的。**  
4． 如果service可以使用startForeground(int, Notification)方法来将service设置为前台状态，那么系统就认为是对用户可见的，并不会在内存不足时killed。  
5． 如果有其他的应用组件作为Service,Activity等运行在相同的进程中，那么将会增加该进程的重要性。

（2）

**方法一：**onStartCommand中返回START\_STICKY，如果内存不足被杀死，那么等内存足够是系统会自动重启Service；

**方法二：**提升service优先级，只是降低了被杀死的概率，但如果被杀死不会重启；

**方法三：**Android中的进程是托管的，当系统进程空间紧张的时候，会依照优先级自动进行进程的回收。Android将进程分为6个等级,它们按优先级顺序由高到低依次是:

   1.前台进程( FOREGROUND\_APP)  
   2.可视进程(VISIBLE\_APP )  
   3. 次要服务进程(SECONDARY\_SERVER )  
   4.后台进程 (HIDDEN\_APP)  
   5.内容供应节点(CONTENT\_PROVIDER)  
   6.空进程(EMPTY\_APP)

当service运行在低内存的环境时，将会kill掉一些存在的进程。因此进程的优先级将会很重要，可以使用startForeground将service放到前台状态。这样在低内存时被kill的几率会低一些。

白色方法：放一个可见的Notification，使用startForeground，如网易云音乐。

**灰色方法：**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，用户无法察觉到你运行着一个前台进程（因为看不到Notification）,但你的进程优先级又是高于普通后台进程的。

**方法四：**监听锁屏事件或者主Activity被关闭时，显示一个1像素的透明Activity，让进程成为前台进程。

**方法五：**守护进程（Native层或者Java层），互相监听。

**6. 广播的两种注册方法，有什么区别。**

（1）两种注册方法

BroadcastReceiver分为两类：

* 静态广播接收者：通过AndroidManifest.xml的标签来申明的BroadcastReceiver。
* 动态广播接收者：通过AMS.registerReceiver()方式注册的BroadcastReceiver，动态注册更为灵活，可在不需要时通过unregisterReceiver()取消注册。

（2）区别

1)静态注册：在AndroidManifest.xml注册，android不能自动销毁广播接收器，**也就是说当应用程序关闭后，还是会接收广播。**2)动态注册：在代码中通过registerReceiver()手工注册.当程序关闭时,该接收器也会随之销毁。当然，**也可手工调用unregisterReceiver()进行销毁**。

**7. Intent的使用方法，可以传递哪些数据类型。**

（1）使用方法

显示Intent比较简单，传入类即可，不再赘述。隐式Intent则需要能够匹配目标组件IntentFilter所设置的过滤信息，不匹配则无法进行调用。IntentFilter中的过滤信息有action,category,data。只有这三个都匹配成功才算完全匹配，只有完全匹配才能成功启动。并且一个Activity可以有多个IntentFilter，只需要匹配成功任何一组就可以启动。

**1、action的匹配规则**

action的要求是必须存在，action也是我们创建Intent是传入构造函数的值。action只需要与过滤规则中的任意一个action相同即可，字符串必须完全一样（大小写也要一致）。

**2、category的匹配规则**

也是只需要与其中一个category相同即可，如果不手动添加则默认为DEFAULT，那么想要启动成功IntentFilter中必须要加DEFAULT。

**3、data匹配规则**

data的匹配规则与action类似，如果IntentFilter中定义了data，那么Intent也必须定义可以匹配的data。data由两部分组成，mimeType与URI，mimeType指的是媒体类型，比如image/jpeg等，而URI的组成就是它所定义的结构，分为scheme,host,port以及路径信息。只有mimeType相同并且URI匹配才行，而URI本身是有默认值的，默认值为content和file。

（2）可以传递的数据类型

**1.基本类型**

**2.可序列化的类型（自定义类型需要实现序列化接口）**

**8. ContentProvider使用方法。**

ContentProvider主要是用于给外界提供数据访问的方式，用于在应用间传递数据。使用方式并不复杂，只需要根据需求实现CRUD操作，然后注册在mainfest中即可。主要注意点如下：

（1）如果要通过uri访问不同数据，要做uri的解析工作；

（2）对于更新操作，要使用同步机制防止并发问题

（3）权限控制、防止sql注入等技术

**9. Thread、AsycTask、IntentService的使用场景与特点。**

答案：

**（1）AsyncTask**

**介绍：**

AsyncTask是一种轻量级的异步任务类，可以在后台线程池中执行后台的任务，然后把执行的进度和最终的结果传递给主线程并在主线程中更新UI。从实现上来说，AsyncTask封装了Thread和Handler。但它并不适合特别耗时的任务，对于特别耗时的任务应该使用线程池。

它是一个泛型抽象类，Params表示参数的类型，Progress表示后台任务进度的类型，而Result表示结果的返回类。

**使用特点：**

（1）它必须在主线程中创建，execute方法必须在主线程中调用

（2）execute方法只能执行一次，虽然可以传很多个参数（任务）

**工作原理：**

AsyncTask实际上是对线程池和Handler进行了封装。

**（1）任务执行：**

3.0之前，AsyncTask是并行执行的，而3.0之后就变为了串行执行，并且开发者可以选择进行并行执行。原理是什么呢？实际上它内部有两个线程池，sDefaultExecutor是一个自己实现的串行的线程池，**它是static的，说明一个进程内的所有任务都是它来执行**，它的任务很简单，就是把任务放进一个队列中，然后提醒另一个并行的线程池THREAD\_POOL\_EXECUTOR来取出执行，如果有可取的**并且当前没有任务在执行**就会被这个并行的线程池来执行。如果有任务在执行自然不会执行，当这个任务执行完之后又会重新提醒并行的线程池THREAD\_POOL\_EXECUTOR来取出队列中的任务进行执行。所以从这个原理我们看出来它是串行执行的，原因就是老版本是串行的并且很多代码依赖于这个逻辑。

**（2）任务结果分发**

它的内部有一个static的handler，所以这也是它必须在UI线程中进行初始化的原因，这样可以保证Handler被正常的初始化。当任务执行完成后，就会将结果发送给Handler，使得其在主线程被执行。

**（2）IntentService**

**介绍：**

仅仅是一个封装了HandlerThread的Service而已，由于Service正常来说也是执行在主线程的，所以不能执行耗时的操作。而IntentService在内部维护有个HandlerThread，它拥有自身的Handler，对应于HandlerThread的Looper。当收到一个Intent时，它将Intent包装到Message中直接发送给Handler来处理，从而避免了在主线程中进行操作。

**使用：**

重写onHandleIntent，在其中处理Intent，并且这个是不在主线程中运行的。

10. 五种布局： FrameLayout 、 LinearLayout 、 AbsoluteLayout 、 RelativeLayout 、 TableLayout 各自特点及绘制效率对比。

11. Android的数据存储形式。

（1）使用SharedPreferences存储数据

**特点：**使用简单，应用内数据共享，但只支持基本数据类型。

（2）使用文件存储

（3）SQLite数据库存储

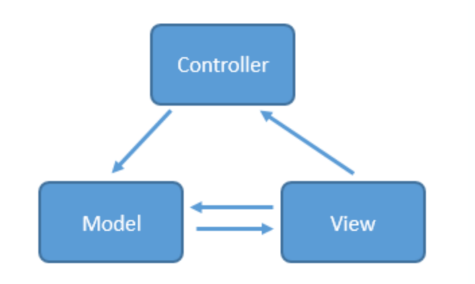
（4）使用ContentProvider存储数据（原理还是123中的一种，只是可以对外共享）

12. Sqlite的基本操作。

**13. Android中的MVC模式以及与MVP的对比。**

（1）Android中的MVC模式

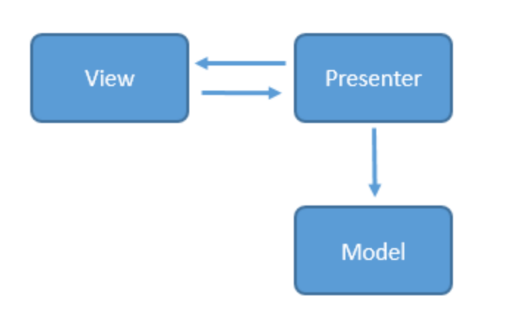
传统的MVC如下图所示：



当用户出发事件的时候，view层会发送指令到controller层，接着controller去通知model层更新数据，model层更新完数据以后直接显示在view层上，这就是MVC的工作原理。

对于原生的Android项目来说，layout.xml里面的xml文件就对应于MVC的view层，里面都是一些view的布局代码，而各种java bean，还有一些类似repository类就对应于model层，至于controller层嘛，当然就是各种activity咯。比如你的界面有一个按钮，按下这个按钮去网络上下载一个文件，这个按钮是view层的，是使用xml来写的，而那些和网络连接相关的代码写在其他类里，比如你可以写一个专门的networkHelper类，这个就是model层，那怎么连接这两层呢？是通过button.setOnClickListener()这个函数，这个函数就写在了activity中，对应于controller层。相较于传统的MVC模式，model层应该要通知View来更新数据才对，而Android中由于View层的控制能力实在太弱，通知view的代码全部放在了activity中，导致activity既算controller又算view，这样activity的代码行数太多，维护起来太过于麻烦。MVC还有一个重要的缺陷，大家看上面那幅图，view层和model层是相互可知的，这意味着两层之间存在耦合，耦合对于一个大型程序来说是非常致命的，因为这表示开发，测试，维护都需要花大量的精力。

（2）Android中的MVP模式



MVP模式中完全把fragment看做是View层，它负责界面的更新和显示操作，而Presenter层则负责了逻辑的实现，从图中就可以看出，最明显的差别就是view层和model层不再相互可知，完全的解耦，取而代之的presenter层充当了桥梁的作用，用于操作view层发出的事件传递到presenter层中，presenter层去操作model层，并且将数据返回给view层，整个过程中view层和model层完全没有联系。一个典型的事件就是点击事件产生，view交由presenter来进行处理，处理完成之后再调用view的界面更新接口进行界面更新。Activity只是作为一个控制器来进行一些全局的控制（因为android的很多操作终究还是依赖于activity的），创建界面的时候activity对view和presenter进行相应的初始化。

看到这里大家可能会问，虽然view层和model层解耦了，但是view层和presenter层不是耦合在一起了吗？其实不是的，对于view层和presenter层的通信，我们是可以通过接口实现的，具体的意思就是说我们的activity，fragment可以去实现实现定义好的接口，而在对应的presenter中通过接口调用方法。不仅如此，我们还可以编写测试用的View，模拟用户的各种操作，从而实现对Presenter的测试。这就解决了MVC模式中测试，维护难的问题。

MVP还有一个巨大优势就是使用面向业务的方式来进行代码编写，代码架构比较清晰。

14. Merge、ViewStub的作用。

15. Json有什么优劣势。

**16. 动画有哪两类，各有什么特点？**

**（1）View动画（补间动画）**

View动画的作用对象是View本身，支持四种动画效果，分别是平移动画、缩放动画、旋转动画和透明度动画。除了这四种，帧动画也属于View动画，是一种特殊的View动画。View动画可以使用xml或者代码来创建，使用set就可以把多个动画组合在一起，调用startAnimation来进行动画的调用。而帧动画就好比是把动画分为许多帧，使用xml来定义一个AnimationDrawable来进行帧动画的使用，**帧动画的问题时图片太多时——OOM。**

View动画的常见作用就是ListView的item的动画，以及Activity之间变换的动画，都是有View动画实现的。而针对View动画的自定义较难，涉及到矩阵变换等知识。

**（2）属性动画**

属性动画相较于View，更加的灵活和多样。属性动画完成的效果其实就是在一个事件间隔内完成对象从一个属性值到另一个属性值的改变，效果更佳更自由。

1、使用注意点

属性动画要求提供属性的get和set方法，可是如果一个控件没有怎么办？最简单的方法就是使用包装类，继承自原始类并重写相对应属性的set和get方法，这是最简单的一种实现方式。

2、原理分析

ObjectAnimator实际上继承自ValueAnimator，主要细节如下：

（1）属性值的设置：使用的是反射来调用，进行属性值的获取和设置，插值器和估值器的作用就在设置之前进行计算，用来确定属性值。

（2）动画效果类似于我们用post实现动画一样，animationHandler实际上是一个Runnable对象，存储在当前线程的ThreadLocal中，start的时候开始执行scheduleAnimation函数， animationHandler一个线程只有一个，用于执行此线程的所有动画。而一个动画过程则是以ValueAnimator的形式存储在animationHandler自己的一个队列Animations中的。当对动画完成了设置只有调用animationHandler的start函数，它里面使用了native的方法来保证UI系统每帧都会执行自己的run函数一次（发送给Looper执行run），run中它取出自己的队列Animations中的ValueAnimator并且执行其中的doAnimationFrame。这就是真正执行动画效果的地方，并且run中有一个FrameTime是当前动画运行的时间，由native函数计算。

（3）doAnimationFrame的作用：它将传入的FrameTime和mStartTime进行比较，因为有可能还未开始执行。之后根据他们的差值，加上插值器和估值器的使用来计算当前应该得到的值，之后利用反射来进行动画的变化调用。

**17. Handler、Looper消息队列模型，各部分的作用。**

答案：

一、消息机制的角色分析   
首先我们介绍一下消息机制的几位主人公，正是有它们的通力合作，消息机制才能正常的运行：   
1、Handler：处理器，负责的内容是消息的发送和具体处理流程，一般使用时由开发者重写handleMessage函数，根据自定义的不同message做不同的UI更新操作；   
2、Message：消息对象，代表一个消息实体，存放消息的基本内容；   
3、MessageQueue：消息队列，按顺序存放消息实体，由单链表实现，被Looper（4）所使用（一个Looper具有一个消息队列）；   
4、Looper：循环器，也是我们的消息机制的主角，一个线程至多具有一个并且与线程一一对应（如何理解等会来说），负责的内容是不断从消息队列取出放入的消息并且交由消息对应的Handler进行处理（Handler的具体handleMessage操作）；   
5、ThreadLocal：线程本地数据，是一个线程内部的数据存储类，为每个线程存储属于自己的独立的数据。

二、消息机制总览   
我们来用最简短的语言说明一下消息循环的整个过程，有个整体性的认识，之后再进行逐一的进行源码分析。   
1、首先，我们知道ThreadLocal是线程内部的数据存储类，一个线程对应一个自己独一无二的数据，而我们的主角Looper就是这样一个对象，每个线程都可以有自己独一无二的Looper，而Looper自己具有一个属于自己的MessageQueue，它不断地从MessageQueue中取Message（注意，Looper的代码是运行在自己对应的线程中的），如果MessageQueue中没有消息，Looper只会不断地循环尝试取消息（阻塞）。   
2、这时，我们在主线程创建了Handler对象，它需要它所在的线程（这里是主线程）所拥有的Looper对象（也就是说，没有Looper的线程创建不了Handler，后面我们也会看到具体代码），一旦我们用Handler发送了消息（可以在别的线程中，这里假设在某个子线程中），Handler就会把这个消息放入自己拥有的Looper对象的属于这个Looper对象的MessageQueue中（这句有点拗口，就是放入Looper的MessageQueue中）。   
3、我们已经知道Looper会不断地尝试从自己的MessageQueue中取出消息，我们刚刚放入的消息立刻被Looper取出来了，它得到了消息就执行了发出消息的Handler（也就是2过程我们所创建的）的消息处理函数handleMessage，我们编写的UI更新操作就在Looper对象的代码中执行了，而我们刚才也说了这个Looper运行在我们创建Handler的线程中的，也就是主线程（UI线程），那这样一来就达到了我们的目标，成功的把代码执行从子线程切换到了主线程中，这个过程我们也就有个总览了，

**底层Android的消息机制见博客。http://blog.csdn.net/ll530304349/article/details/52959415**

**18. 怎样退出终止App。**

建立一个基类Activity，在Activity创建时添加到全局表中，在Activity销毁时移除全局表，调用全局退出时，对全局表Activity进行全部退出，达到完全退出的效果。

**19. Assets目录与res目录的区别。**

assets目录与res下的raw、drawable目录一样，也可用来存放资源文件，但它们三者有区别，对比总结如下表：

|  |  |  |  |
| --- | --- | --- | --- |
|  | **assets** | **res/raw** | **res/drawable** |
| **获取资源方式：** | *文件路径+文件名* | *R.raw.xxx* | *R.drawable.xxx* |
| **是否被压缩：** | *NO* | *NO* | *YES（失真压缩）* |
| **能否获取子目录下的资源：** | *YES* | *NO* | *NO* |

 （1）res/raw和asset中的文件不会被编译成2进制文件，而是原样复制到设备上，可以用文件流来读取原始文件，所以可以用来存放通用的静态文件，比如视频等；

（2）与res/raw不同点在于，Assets支持任意深度的子目录，这是它们的主要区别。这些文件不会生成任何资源ID，必须使用/assets开始（不包含它）的相对路径名，而res中的所有文件都会生成索引来直接访问。

20. Android怎么加速启动Activity。

**21. Android内存优化方法：ListView优化，及时关闭资源，图片缓存等等。**

**（1）Bitmap的高效加载**

核心是利用BitmapFactory加载一个图片时（从文件系统、资源、输入流以及字节数组）中加载一个Bitmap对象的时候，选择合适的采样率进行加载，即Options参数的采样率参数对要加载的图片进行缩放，变成合适ImageView的大小的图片。缩放率是1/采样率的平方。具体做法如下：

**第一：**首先设置Options的inJustDecodeBounds为true并加载图片，这样只会获取图片的参数（长宽）

**第二：**根据需要的长宽对图片的长宽不停做/2操作，计算合适的采样率

**第三：**根据新的采样率，重新加载图片

**（2）ListView（RecyclerView）的优化**

**1、ListView的基础优化方式**

（1）优化加载布局——convertView

通过在getView方法中使用convertView从而来避免View的重复加载，原理是复用已经离开屏幕的View，避免了View的重新加载。

（2）优化加载控件——ViewHolder

通过给复用的view设置Tag，实际上就是避免了重新find控件的过程，将控件的引用提前设置给ViewHolder，让其持有，这样重新设置数据复用是的速度更快。

**2、ListView的加载优化方式**

（1）加载图片时使用压缩方式——Bitmap的高效加载

（2）异步加载过程

（3）缓存加载，利用缓存避免重复加载

优化详情见优化方法整理。

**22. Android中弱引用与软引用的应用场景。**

**（1）弱引用**

主要作用：防止内存泄漏。

使用场景：全局Map用于保存某种映射的时候一定一定使用弱引用来保存对象，因为全局变量一般是static的，它的声明周期一定长于单个对象，如果用弱引用保存对象，当对象被回收时，如果使用强引用，对象就会发生内存泄漏问题。

**（2）软引用**

主要作用：缓存

使用场景：对于Bitmap的加载，非常耗费时间，我们希望把加载过的Bitmap做缓存来节省加载时间，可是Bitmap非常吃内存，我们又不希望发生OOM的问题，所以应该使用软引用来做缓存，这样在系统内存不足时，此部分内存又会重新被回收，避免OOM的问题、

23. Bitmap的四种属性，与每种属性队形的大小。

**24. View与View Group分类。自定义View过程：onMeasure()、onLayout()、onDraw()。**

**View的总体绘制过程：**

当Activity对象被创建完成，会将DecorView添加到Window中（显示），同时创建ViewRoot的实现对象ViewRootImpl与之关联。ViewRootImpl会调用performTraversals来进行View的绘制过程。经过measure,layout,draw三个流程才能完成一个View的绘制过程，分别是用于测量宽、高；确定在父容器中的位置；绘制在屏幕上三个过程。而measure方法会调用onMeasure函数，这其中又会调用子元素的measure函数，如此反复就能完成整个View树的遍历过程。其他两个流程也同样如此。

measure决定了View的宽和高，测量之后就可以根据getMeasuredWidth和getMeasuredHeight来获取View测量后的宽和高，几乎等于最终的宽和高，但有例外；layout过程决定了View四个顶点的位置和实际的宽和高，完成之后可以根据getTop,getBottom,getLeft,getRight来获得四个顶点的位置，并且可以使用getWidth和getHeight来获取实际的宽和高；draw过程就决定了View的显示，完成draw才能真正显示出来。

**1.测量Measure过程：**

MeasureSpec是测量规格，它是系统将View的参数根据父容器的规则转换而成的，之后根据它来测量出View的宽和高。它实际上是一个32位的int值，高二位表示SpecMode，就是测量的模式；低30位表示SpecSize，即在某种测量模式下的规格大小。

MeausureSpec有三种模式，常用的由两种：EXACTLY和AT\_MOST。EXACTLY表示父容器已经检测出View所需要的精确大小（即父容器根据View的参数已经可以确定View的大小了），这时View的最终大小就是SpecSize的值，它对应于View参数中的match\_parent和具体大小数值这两种模式；AT\_MOST表示父容器指定了一个可用大小的数值，记录在SpecSize中，View的大小不能大于它，但具体的值还是看View的具体实现。它对应于View参数中的wrap\_content。

DecorView（顶级View）的测量由窗口的大小和自身的LayoutParams决定，具体逻辑由getRootMeasureSpec决定，如果是具体值或者是match\_parent，就是精确模式；如果是wrap\_content就是最大模式；普通View的measure实际上是由父元素进行调用的（遍历），父元素调用child的measure之前使用getChildMeasureSpec来转换得到子元素的MeasureSpec（具体代码：艺术探索P180-181），总结而来就是与自身的参数以及父元素的SpecMode有关：1、如果View的参数是具体值，那么不管父元素的Mode是什么，子元素的Mode都是精确模式并且大小就是参数的大小；2、如果View的参数是match\_parent，如果父元素的mode是精确模式那么View也是精确模式并且大小是父元素剩余的大小；如果父元素的mode是最大模式，那么View也是最大模式；3、如果View的参数是wrap\_content，那么View的模式一定是最大化模式，并且不能超过父容器的剩余空间。

**View的measure过程：**

View自身的onMeasure方法就是把MeasureSpec的Size设为最终的测量结果，这样的测量问题就是match\_parent和wrap\_content是一样的结果（因为wrap\_content的Size是最大可用Size），所以如果自定义View直接继承自View，就需要对wrap\_content进行处理，ImageView等都对wrap\_content进行了特殊处理。

**ViewGroup的measure过程：**

ViewGroup不同于View，它是一个抽象类，没有实现onMeasure方法（因为具体的Layout布局特性各不相同），但它measure时会遍历children调用measureChild，执行getChildMeasureSpec进行子元素的MeasureSpec创建，创建过程之前已经了解了，就是利用自身的Spec与子元素参数进行创建。

**2.确定位置Layout过程：**

Layout的作用是ViewGroup来确定子元素的位置，当ViewGroup的位置被确定了之后，它就在自己的onLayout函数中遍历所有的子元素并调用其layout方法，确定子元素的位置，对于子元素View，layout中又会调用其onLayout函数。View和ViewGroup中都没有真正实现onLayout方法。但View和ViewGroup的layout方法是一致的，作用都是用于确定自己的位置，layout方法会调用setFrame方法来设定View的四个顶点的位置，即初始化mLeft,mRight,mTop,mBottom四个值，这样就确定了View在父元素中的位置。

**问题：getMeasuredHeight和getHeight有什么区别（同Width）？**

在measure之后就可以使用getMeasuredHeight来进行获取测量的宽和高，而layout过程是晚于measure的，ViewGroup的setChildFrame会调用child的layout来确定child的真实位置，源代码中也可以看出layout的bottom和top就是利用getMeasuredHeight和getMeasuredWidth来计算的，所以说如果child的layout不重写，那么就是一样的！如果child的layout函数被重写，就会有不一样的结果。

**3.绘制draw过程：**

OnDraw中进行绘制自己的操作。使用canvas进行绘制等等，简单来说就是利用代码画自己需要的图形。

**绘制过程中的旋转以及save和restore**

Android中的旋转rotate旋转的是坐标系，也就是说旋转画布实际上画布本身的内容是不动的，不会直接把画布上已经存在的内容进行移动，只是旋转坐标系，这样旋转之后的操作全部是针对这个新的坐标系的。

save就是保存当前的的坐标系，之后再调用restore时，坐标系复原到save保存的状态，这两个函数restore的次数不能大于save，否则会引发异常。

**25. Touch事件分发机制以及滑动冲突的解决。**

答案：

**一、View的事件分发机制**

事件分发机制传递的就是MotionEvent，也就是点击事件，这个传递过程就是分发的过程。

（1）点击事件的传递规则

三大函数：

**public boolean dispatchTouchEvent(MotionEvent ev)**

这个函数用于进行事件的分发，如果这个时间能够传递给当前的View，那么这个方法一定会调用，返回的结果表示是否消耗当前事件，返回的结果受onInterceptTouchEvent和下级View的影响。

**public boolean onInterceptTouchEvent(MotionEvent ev)**

这个函数内部调用，用于判断是否拦截某个事件，如果当前View拦截了某个事件，那么同一事件序列中，此方法不会被再次调用。

**public boolean onTouchEvent(MotionEvent ev)**

在**dispatchTouchEvent**中调用，用于处理点击事件，其返回结果表示是否消耗当前事件，如果不消耗，那么同一事件序列中，当前View无法再接收到事件。

伪代码：

public boolean dispatchTouchEvent(MotionEvent ev){

boolean consume = false;

if(**onInterceptTouchEvent(ev)**){

consume = **onTouchEvent**(ev);

} else {

consume = child. **dispatchTouchEvent(ev);**

}

return consume;

}

通过上述伪代码，我们可以大致得出传递的规则：

（1）对于一个根ViewGroup来说，点击事件产生后，首先会传递给它自己，如果它的**onInterceptTouchEvent**返回true，那么就表示它要拦截当前事件，那么它的**onTouchEvent**函数就会被调用；如果返回false，那么就传递给子元素，直到事件被处理。

（2）当一个View需要进行事件处理时，如果它设置了OnTouchListener，那么它的onTouch方法就会被调用，**这时事件如何处理还要看onTouch的返回值，如果返回false，那么当前View的onTouchEvent就会被调用；如果返回true，那么onTouchEvent方法将不会调用！！！！！！。**由此可见，OnTouchListener的优先级高于onTouchEvent。在onTouchEvent方法中，如果当前设置有OnClickListener，那么它的onClick会被调用，其优先级最低，处于调用的末端。

（3）如果一个事件传递到View，如果此View的onTouchEvent返回false，就是不消耗事件，那么此View的父容器的onTouchEvent就会被调用，也就是说如果事件最终没有View处理，那么处理的人就是Activity，也就是责任链模式。

一些结论：

（1）同一个事件序列指的是从手指接触屏幕开始，到手指离开屏幕的过程，也就是DOWN—MOVE…MOVE—UP，这是一个事件序列。

（2）同一个事件序列只能被同一个View所消耗，因为一旦一个View拦截了某个事件，那么同一序列内的所有事件都会直接交给他处理。但是要注意，如果事件在之前又被别人拦截，根本不交给它处理的情况也会发生——事件拦截。

（3）某个View一旦开始处理事件，如果它不消耗ACTION\_DOWN事件（onTouchEvent返回了false），那么同一事件序列中的其他事件都不会再交给他处理，即父元素的onTouchEvent会被调用，交给父元素来处理（责任链模式）。

如果某个View不处理除了ACTION\_DOWN之外的其他事件，那么这个点击事件就会消失，并且当前View可以持续接收到后续事件（无论你选择不选择处理），最终消失的会被Activity处理。

（4）ViewGroup的onInterceptTouchEvent方法默认返回false，即不拦截任何事件，而View没有onInterceptTouchEvent函数，即不能选择是否拦截，必须拦截，但可以不处理。

（5）View的onTouchEvent默认都会消耗事件，返回true，除非它是不可点击的。

（6）对于onTouch和onClick的总结

**规律（总结）：**

（1）首先没有设置OnClickListener的情况下，onTouch的返回值表示的就是View对点击事件是否消耗，如果在DOWN事件传递过来时返回false，那么剩下的MOVE直到UP的事件都不会被onTouch接收到；如果在DOWN事件返回true，那么剩下的直到UP的事件都会接受到，无论你之后的返回值。   
（2）在同时设置了OnTouchListener与OnClickListener之后，情况就有些复杂了：   
**情况1：**如果onTouch在DOWN时返回了true，那么onTouch就和（1）一样收到剩下的所有事件，但onClick就不会被执行；   
**情况2：**如果onTouch在DOWN时返回了false，与（1）不同的是，onTouch尽管在DOWN时返回了false，但之后的所有事件仍能接受到，并且onClick会在之后被调用。

public boolean dispatchTouchEvent(MotionEvent event){

... ...

if(onFilterTouchEventForSecurity(event)){

ListenerInfo li = mListenerInfo;

if(li != null && li.mOnTouchListener != null && (mViewFlags & ENABLED\_MASK) == ENABLED

&& li.mOnTouchListener.onTouch(this, event)) { //（1）onTouch调用

return true;

}

if(onTouchEvent(event)){ //（2）onTouchEvent调用

return true;

}

}

... ...

return false;

}

分析：

（1）如果没有设置OnClickListener，只设置了OnTouchListener，那么在代码（1）处就会调用onTouch，如果DOWN事件时返回了true，那么剩下的事件都会交由此View进行处理；如果返回了false，那么就会执行代码（2）处的onTouchEvent函数，如果设置了OnClickListener，就会在其中进行调用，如果没有设置，dispatchTouchEvent就会返回false，那么剩下的事件都不会交由此View进行处理；   
（2）如果同时设置了OnTouchListener与OnClickListener，那么我们再按上面的两种情况进行分析：   
**情况1：**onTouch在DOWN时返回了true，那么代码（1）处就得到了真的结果，直接就返回了true，可以知道后面代码（2）处的onTouchEvent函数不会被执行，那么自然你的OnClickListener就不起作用了，onClick就不会被执行；  **情况2：**onTouch在DOWN时返回了false，那么当DOWN事件传递来的时候，代码（1）处就不会得到真的结果，也就是说onTouch中你表示自己不会处理这个事件序列了，后面代码（2）处的onTouchEvent函数就会得到执行，而如果你设置了OnClickListener，View就会处于CLICKABLE状态，那么onTouchEvent函数就会返回true，又表示你可以处理这个点击事件序列了，dispatchTouchEvent就会返回true，那么这时后面的事件由于DOWN时返回true，就会统统交由此View进行处理，自然你的onTouch中也能够监听到后面的所有事件！这样上面的情况就能够得到解释了。

**二、滑动冲突的解决方法**

（1）滑动冲突的类型

滑动冲突分为三种类型，第一类是外部和内部滑动方向不一致，第二类是外部和内部滑动方向一致，第三类是前两种嵌套的模式。

处理这三种类型的规则分为两类，对于第一种类型，我们可以根据滑动方向来处理，符合处理方向的分配给对应的控件；对于2、3种类型，必须根据业务上的区别来处理，某种状态的处理时间分发给对应的控件来处理。

（2）滑动冲突的解决方式

**解决方式一：外部拦截法**

外部拦截法指点击事件首先都会经过父容器的拦截处理，父容器如果需要此事件就进行拦截，如果不需要此事件就不进行拦截，这样就可以解决滑动冲突问题。外部拦截法主要就是重写父容器的onInterceptTouchEvent方法，但是要注意，父容器拦截不能在ACTION\_DOWN中返回true，否则之后的所有事件序列都会交给它处理，无论返回什么，因为不会再调用它的onInterceptTouchEvent函数了。所以父控件应该在ACTION\_MOVE中选择是否拦截。但是这种拦截的问题是，如果拦截了，那么子控件的onClick事件将无法再出发了。

伪代码如下：

public boolean onInterceptTouchEvent(MotionEvent ev) {  
 boolean intercepted = false;  
 switch (ev.getAction()){  
 case MotionEvent.*ACTION\_DOWN*:  
 intercepted = false;  
 break;  
 case MotionEvent.*ACTION\_MOVE*:  
 if(父控件需要处理){  
 intercepted = true;  
 } else{  
 intercepted = false;  
 }  
 break;  
 case MotionEvent.*ACTION\_UP*:  
 intercepted = false;  
 break;  
 }  
  
 return intercepted;  
}

**解决方法二：内部拦截法**

内部拦截法指的是父容器不拦截任何事件，所有事件全部传递给子元素，如果子元素需要就进行消耗，否则交由父容器进行处理。这种方式需要配合ViewGroup的FLAG\_DISALLOW\_INTERCEPT标志位来使用。设置此标志为可以通过requestDisallowIntercept TouchEvent函数来设置，如果设置了此标志位，那么ViewGroup就无法拦截除了ACTION\_DOWN之外的任何事件。这样首先我们保证ViewGroup的onInterceptTouchEvent方法除了DOWN其他都返回true，DOWN返回false，这样保证了不会拦截DOWN事件，交给它的子View进行处理；重写View的dispatchTouchEvent函数，在DOWN中设置parent.requestDisallowInterceptTouchEvent(true)，这样父控件在默认的情况下DOWN之后的所有事件它都拦截不到，交由子View来处理，View在MOVE中判断父控件需要时，调用parent.requestDisallow InterceptTouchEvent(false)，这样父控件的拦截又起作用了，相应的事件交给了父控件进行处理。伪代码如下：

**父控件中：**

@Override  
public boolean onInterceptTouchEvent(MotionEvent ev) {  
 int action = ev.getAction();  
 if(action == MotionEvent.*ACTION\_DOWN*){  
 return false;  
 } else {  
 return true;  
 }  
}

**子View中：**

@Override  
public boolean dispatchTouchEvent(MotionEvent ev) {  
   
 switch (ev.getAction()){  
 case MotionEvent.*ACTION\_DOWN*:  
 getParent().requestDisallowInterceptTouchEvent(true);  
 break;  
 case MotionEvent.*ACTION\_MOVE*:  
 if(父控件需要此点击事件){  
 getParent().requestDisallowInterceptTouchEvent(false);  
 }  
 break;  
 case MotionEvent.*ACTION\_UP*:  
 break;  
 }  
}

**滑动冲突处理案例：下拉刷新实现原理。**

26. Android长连接，怎么处理心跳机制。

27. Zygote的启动过程。

**28. Android IPC:Binder原理。**

见Binder整理文档。

**29. 你用过什么框架，是否看过源码，是否知道底层原理。**

见框架整理文档。

30. Android5.0、6.0新特性。