目录

1. 四大组件是什么.....4
2. 四大组件的生命周期和简单用法.....4
3. Activity 之间的通信方式.....8
4. Activity 各种情况下的生命周期.....9
5. 横竖屏切换的时候，Activity 各种情况下的生命周期.....10
6. Activity 上有 Dialog 的时候按 Home 键时的生命周期.....12
7. 两个 Activity 之间跳转时必然会执行的是哪几个方法？.....12
8. 前台切换到后台，然后再回到前台，Activity 生命周期回调方法。弹出 Dialog，生命值周期回调方法。.....13
9. Activity 的四种启动模式对比.....14
10. Activity 状态保存于恢复.....15
11. 简述 Activity 启动全部过程.....15
12. Activity 栈（原理）.....18
13. LaunchMode 应用场景(源码).....18
14. Android Service 与 Activity 之间通信的几种方式(源码).....20
15. ApplicationContext 和 ActivityContext 的区别(源码).....22
16. Application 和 Activity 的 Context 对象的区别(应该跟上面是同一个问题).....24
17. Activity 与 Fragment 之间生命周期比较.....25
18. fragment 各种情况下的生命周期.....27
19. Fragment 状态保存 startActivityForResult 是哪个类的方法，

在什么情况下使用？.....34
20. 如何实现 Fragment 的滑动？.....39
21. fragment 之间传递数据的方式？.....40
22. 说说 Activity、Intent、Service 是什么关系(源码).....40
23. IntentService 原理及作用是什么？(源码).....41
24. 怎么在 Activity 中启动自己对应的 Service？.....42
25. service 和 activity 怎么进行数据交互？.....42
26. Service 的开启方式.....43
27. 请描述一下 Service 的生命周期.....45
28. 谈谈你对 ContentProvider 的理解.....46
29. 说说 ContentProvider、ContentResolver、ContentObserver 之间的关系.....46
30. 请介绍下 ContentProvider 是如何实现数据共享的？(源码).47
31. ContentProvider 的权限管理(解答：读写分离，权限控制-精确到表级，URL 控制)（原理）.....47
32. 如何通过广播拦截和 abort 一条短信？（原理）.....47
33. 广播是否可以请求网络？（原理）.....48
34. 请描述一下广 BroadcastReceiver 的理解.....48
35. 广播的分类.....56
36. 广播使用的方式和场景.....56
37. 在 manifest 和代码中如何注册和使用 BroadcastReceiver?..57
38. 本地广播和全局广播有什么差别？.....58

本地广播和全局广播的差别.....58
39. BroadcastReceiver，LocalBroadcastReceiver 区别.....59
40. AlertDialog,popupWindow,Activity 区别.....60

1. 四大组件是什么

① Activity：Activity 就是我们看到的界面显示，一般用于呈现页面内容，处理用户交互等等

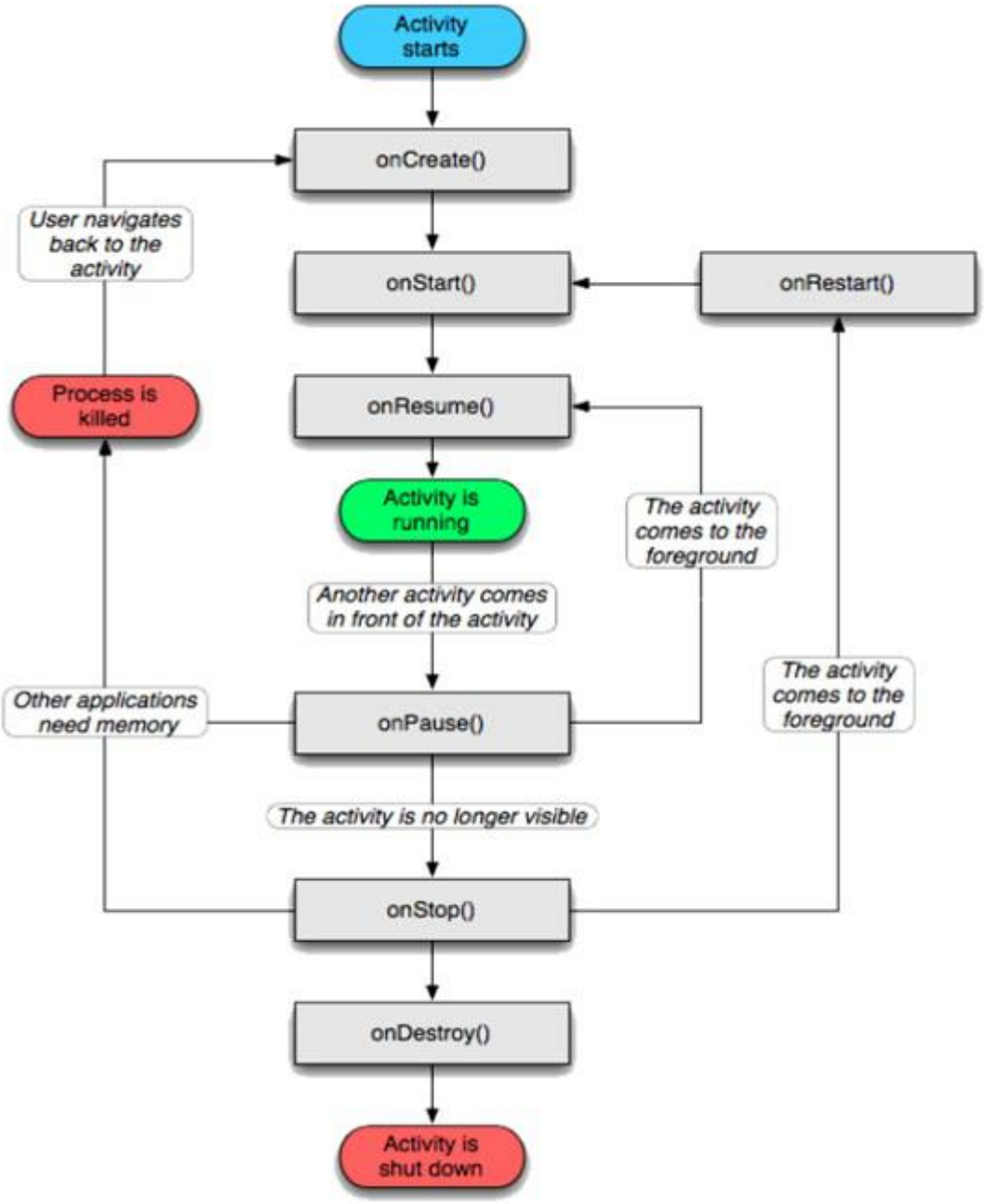
② Service：Service 是一个可以运行于后台的组件，我们一般用于处理一些不需要用户知道，但是又必须执行耗时的操作，比如下载。如果按照运行地点分的话，可以分为本地服务和远程服务，也就是说它的启动是否跟启动他的进程有关系。

③ ContentProvider：内容提供者，主要用来对外共享数据的。主要用于进程间通信，比如暴露某个 APP 的信息内存给予另外一个 APP 获取使用，比如获取联系人等等

④ BroadcastReceiver：广播接收者主要用于接受广播信息。是一个全局的监听器，一般用来传递接受消息的。

2. 四大组件的生命周期和简单用法

Activity 的生命周期：



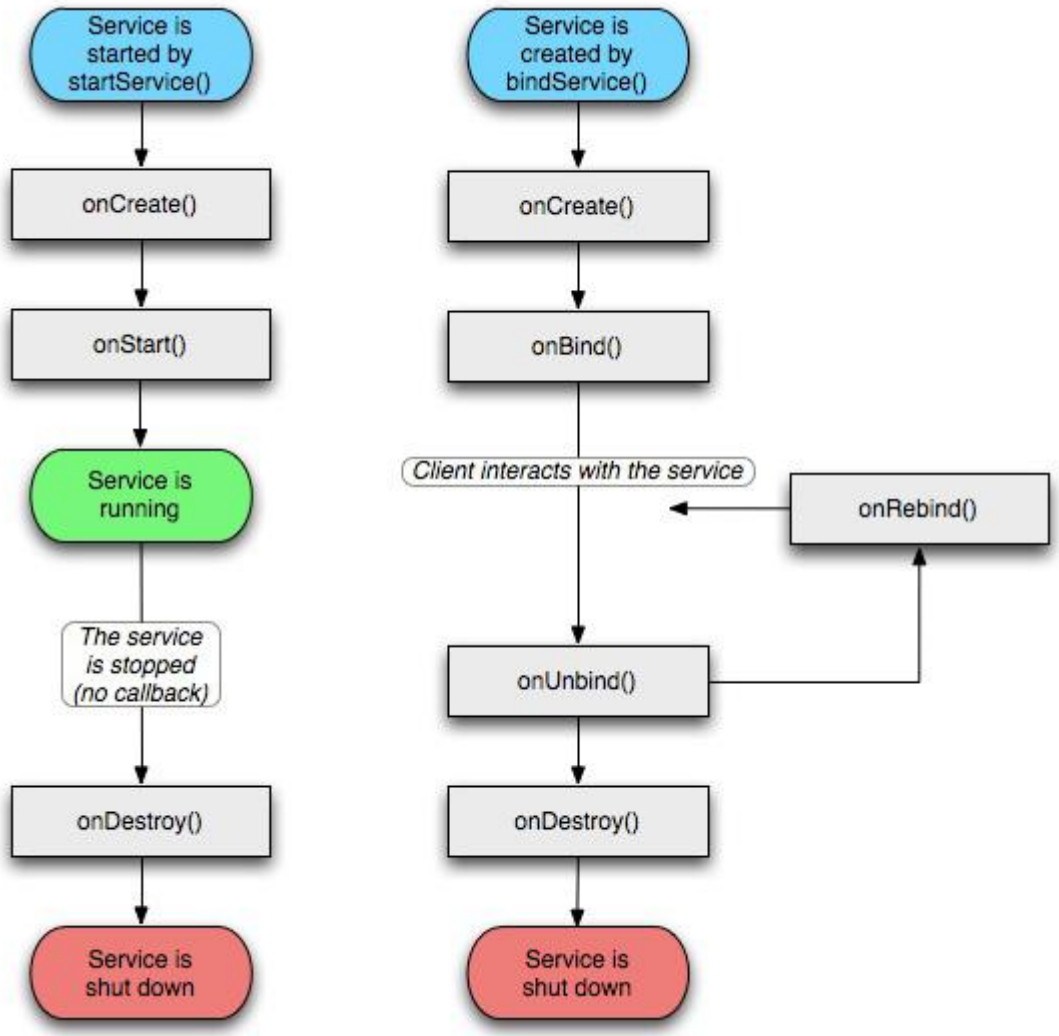
启动方法：

创建一个 Intent 对象,将该 activity 以及将要打开的 activity 传进去，

然后 Intent 对象将代入 startActivity（）即可启动

```
startActivity(Intent(this@someActivity,targetClass::class.java))
```

Service 的生命周期：



使用方法有两种:

startService()

通过简单的 startService()进行 service 启动，此后启动该 Service 的组件无法把控 Service 的生命周期，理论上此后该 Service 可以在后台无期限运行,但根据实际情况该 Service 可能会在任意一个时刻被杀死

我们可以在 onStartCommand()里面做我们要做的操作，虽然运行 anr 时间比 Activity 多了近一倍，但 Service 跟 Activity 一样不可以做耗时操作，

bindService()

通过绑定的方式启动 Service

绑定后，该 Service 与启动绑定操作的组件形成绑定，当组件销毁时，该 Service 也随着销毁。

BroadcastReceive 广播接收器生命周期：

生命周期只有十秒左右，如果在 onReceive() 内做超过十秒内的事情，就会报 ANR(Application No Response) 程序无响应的错误信息

它的生命周期为从回调 onReceive()方法开始到该方法返回结果后结束

用法

静态注册（常驻广播）

在 AndroidManifest.xml 中进行注册,App 启动的时候自动注册到系统中，不受任何组件生命周期影响，（即便应用程序已经关闭），但是 *耗电，占内存*

动态注册（非常驻广播）

在代码中进行注册,通过 IntentFilter 意图过滤器筛选需要监听的广播,记得**注销**（推荐在 onResume()注册，在 onPause()注销），使用灵活，生命周期随组件变化

ContentProvider

1. 提供数据的进程使用 contentProvider 内容提供者
2. 获取数据的进程使用 contentresolver 内容解析器

3. Activity 之间的通信方式

常用的通信方式如下：

1.Intent

在 startActivity()或者 startActivityForResult()时，通过 Intent 携带需要的信息，但是 intent 对携带写信的大小有限制

2.Broadcast 广播

在 A 中发出广播，在 B 中接收广播并解析其中数据

3.用数据存储的方式

理论上凡是数据存储的方式，我们均能在 A 存储信息，并在 B 读取，达到通信的目的，具体方式如 SharedPreference/SQLite/File/Android 剪切板等

4.使用静态变量

在 A 中将静态变量赋值，在 B 中读取并置空

4. Activity 各种情况下的生命周期

1） **启动Activity**：onCreate()—>onStart()—>onResume()，Activity 进入运行状态。

2） **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。

3） **Activity 返回前台**： onRestart()—>onStart()—>onResume()，再次回到运行状态。

4） **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()

5） **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. 横竖屏切换的时候，Activity 各种情况下的生命周期

分两种情况：

(1) 不设置 Activity 的 android:configChanges，或设置 Activity 的 android:configChanges="orientation"，或设置 Activity 的 android:configChanges="orientation|keyboardHidden"，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

行一次。

横竖屏切换造成 activity 的生命周期

onPause()-onSaveInstanceState()-onStop()-onDestroy()-onCreat()-onStart()-onRestoreInstanceState()-onResume()即会导致 activity 的销毁和重建 。

(2) 配置 android:configChanges="orientation|keyboardHidden|screenSize"，才不会销毁 activity，且只调用 onConfigurationChanged 方法。

onSaveInstanceState() 与 onRestoreIntanceState() 资源相关的系统配置发生改变或者资源不足时(例如屏幕旋转)，当前 Activity 会销毁，并且在 onStop 之前回调 onSaveInstanceState 保存数据，在重新创建 Activity 的时候在 onStart 之后回调 onRestoreInstanceState。其中 Bundle 数据会传到 onCreate（不一定有数据）和 onRestoreInstanceState（一定有数据）。

用户或者程序员主动去销毁一个 Activity 的时候不会回调(如代码中 finish（）或用户按下 back，不会回调)，其他情况都会调用，来保存界面信息。

6. Activity 上有 Dialog 的时候按 Home 键时的生命周期

们看到程序的生命周期是：onCreate() -> onStart() -> onResume -> onPause() -> onStop()

我们弹出的 Dialog 实际上是一个布满全屏的 Activity 组件,，因而我们队 Activity 并不是不可见而是被一个布满屏幕的组件覆盖了其他组件。当我们按 Home 键时，Activity 会执行正常的 onPause() -> onStop()操作，使 Activity 真正进入后台。

7. 两个 Activity 之间跳转时必然会执行的是哪几个方法？

两个 Activity 之间跳转必然会执行的是下面几个方法。

(1)onCreate():在 Activity 生命周期开始时调用。

(2)onRestoreInstanceState():用来恢复 UI 状态。

(3)onRestart():当 Activity 重新启动时调用。

(4)onStart():当 Activity 对用户即将可见时调用。

(5)onResume():当 Activity 与用户交互时，绘制界面。

(6)onSaveInstanceState():即将移出栈顶保留 UI 状态时调用。

(7)onPause():暂停当前活动 Activity，提交持久数据的改变，停止动画或其他占用 GPU 资源的东西，由于下一个 Activity 在这个方法返回之前不会 resume，所以这个方法的代码执行要快。

(8)onStop():Activity 不再可见时调用。

(9)onDestroy():Activity 销毁栈时被调用的最后一个方法

8. 前台切换到后台，然后再回到前台，Activity 生命周期回调方法。弹出 Dialog，生命值周期回调方法。

前台切换到后台：onPause()->onStop()

切回来时分两种情况：

如果正常切换回来,那么会走 onRestart()->onStart()->onResume()

如果被系统内存紧急被回收了，那么回来会重新走一次生命周期

弹出 dialog

如果 dialog 是自身 Activity 弹出来的，则不会走生命周期

9. Activity 的四种启动模式对比

（一）standard

这个是 Activity 的默认启动方式，我们不需要额外的配置

在该配置下，启动一个 Activity 就会在该应用的 Activity 栈中压入一个 Activity,返回的时候就直接把该 Activity 弹出栈。

（二）singleTop

这个是栈顶复用模式

在该配置下，如果在 Activity 栈，栈顶是该 Activity，那么会走 onNewIntent()->onResume()

如果不是，那么就走正常的生命周期

（三）singleInstance

这个是栈内复用模式

在该配置下，如果在该 Activity 栈，栈内存在该 Activity(**没有要求是栈顶**)，那么会走 onNewIntent()->onResume()，并且把位于该 Activity 上方的 Activity 全部出栈，使该 Activity 位于栈顶

（四）singleTask

这个配置下，Activity 独享一个 Activity 栈。

10. Activity 状态保存于恢复

重写 onSaveInstanceState(outState: Bundle?){} 将数据数据写进去来保存状态。

重 写 onRestoreInstanceState(savedInstanceState: Bundle?){} 获取数据来恢复状态

11. 简述 Activity 启动全部过程

等待框架文字化以及书本总结

- Launcher 直接启动 app，或者是 activity 调用 startActivity 开启一个新的 activity 均需要通过 Binder 进程间通信进入到 ActivityManagerService 进程中，并且调用 ActivityManagerService.startActivity 接口；

- - ActivityManagerService 调用 ActivityStack.startActivityMayWait 来做准备要启动的 Activity 的相关信息；

- ActivityStack 通知 ApplicationThread 要进行 Activity 启动调度了，这里的 ApplicationThread 代表的是 ActivityManagerService.startActivity 接口的进程里，对于通过点击应用程序图标的场景来说，这个进程就是 Launcher 了。

- ApplicationThread 并不执行真正的启动操作，它通过调用 ActivityManager.activityPaused 接口进入到 ActivityManagerService 进程中，看看是否需要创建新的进程来启动 Activity

- 对于通过点击应用程序图标来启动 Activity 的场景来说，ActivityManagerService 在这一步中，会调用 startProcessLocked 来创建一个新的进程，而对于通过在 Activity 内部调用 startActivity 来启动新的 Activity 来说，这一步是不需要执行的，因为新的 Activity 就是在原来的 Activity 所在的进程中执行的。

- ActivityManagerService 调用 ApplicationThread.scheduleLaunchActivity 接口，通知相应的进程执行启动 Activity 的操作；

- ApplicationThread 把这个启动 Activity 的操作转发给 ActivityThread，ActivityThread 通过 ClassLoader 导入相应的 Activity 类然后启动起来。

·

Activity 的启动完成了以下几件事：

(1)获取 Activity 组件信息：从传入的 ActivityClientRecord 中获

取待启动的 Activity 的组件信息

(2)创建类加载器：使用 Instrumentation#newActivity()加载

Activity 对象

(3)创建 Application：调用 LoadedApk.makeApplication 方法尝试创建 Application，由于单例所以不会重复创建。

(4)创建 Context：创建 Context 的实现类 ContextImpl 对象，并通

过 Activity#attach()完成数据初始化和 Context 建立联系，因为 Activity 是 Context 的桥接类，

(5)创建关联 Window：最后就是创建和关联 window，让 Window 接收

的事件传给 Activity，在 Window 创建过程中会调用 ViewRootImpl 的 performTraversals()初始化 View。

(6)显示界面：

Instrumentation#callActivityOnCreate()->Activity#perform

Create()->Activity#onCreate().onCreate()中会通过 Activity#setContentView()调用 PhoneWindow 的 setContentView()刷新界面

12. Activity 栈（原理）

任务栈是一种后进先出的结构。

位于栈顶的 Activity 处于焦点状态,当按下 back 按钮的时候,栈内的 Activity 会一个一个的出栈,并且调用其 onDestory()方法。如果栈内没有 Activity,那么系统就会回收这个栈,每个 APP 默认只有一个栈,以 APP 的包名来命名

13. LaunchMode 应用场景(源码)

一共有以下四种 launchMode：standard、singleTop、singleTask 、singleInstance

(1)**standard**：系统的默认模式：每次启动都会创建一个新的 activity 对象(一次跳转即会生成一个新的实例),放到目标任务栈中。一次跳转即会生成一个新的实例。平时我们默认使用的就是这种模式。

(2)**singleTop**：判断当前的任务栈顶是否存在相同的 activity 对

象，如果存在，则直接使用，如果系统发现存在有 Activity 实例，但不是位于栈顶，重新生成一个实例。；如果不存在，那么创建新的 activity 对象放入栈中，如果打开的 activity 在栈顶，调用。onNewIntent 方法 ，从 onResume()开始

适用范围：适合接收通知启动的内容显示页面。例如，某个新闻客户端的新闻内容页面，如果收到 10 个新闻推送，每次都打开一个新闻内容页面是很烦人的。

(3)**singleTask**：在任务栈中会判断是否存在相同的 activity，如果存在，那么会清除该 activity 之上的其他 activity 对象显示；如果不存在，则会创建一个新的 activity 放入栈顶。当打开的 activity 存在于栈内，将其上方的 activity 都销毁掉，使此 Activity 实例成为栈顶对象，显示到幕前，调用 onNewIntent 方法 ，从 onResume()开始。

适用范围：适合作为程序入口点。 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，其余情况都会走 onNewIntent，并且会清空主界面上面的其他页面。

(4)**singleIntance**：会在一个新的任务栈中创建 activity，并且该任务栈种只允许存在一个 activity 实例，其他调用该 activity 的组件会直接使用该任务栈中的 activity 对象。与其他 activity 是独立的，它有自己的上下文 activity。将 Acitvity 放置于这个新的栈结构中，并保证不再有其他 Activity 实例进入。

适用范围：界面分离。例如闹铃提醒，将闹铃提醒与闹铃设置分离。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，在此启动，首先打开的是 B。

14. Android Service 与 Activity 之间通信的几种方式(源码)

1. **通过 broadcast:通过广播发送消息到 activitry**
2. **通过 Binder：通过与 activity 进行绑定**

1）添加一个继承 Binder 的内部类，并添加相应的逻辑方法。

2）重写 Service 的 onBind 方法，返回我们刚刚定义的那个内部

类实例。

①Service代码段

```
public class MyService extends Service {

    public MyService() {
    }

    private DownloadBinder mBinder = new DownloadBinder();

    class DownloadBinder extends Binder {

        public void startDownload() {
            Log.d("MyService", "startDownload executed");
        }//在服务中自定义startDownload()方法，待会活动中调用此方法

        public int getProgress() {
            Log.d("MyService", "getProgress executed");
            return 0;
        }//在服务中自定义getProgress()方法，待会活动中调用此方法

    }

    @Override
    public IBinder onBind(Intent intent) {
        return mBinder;
    }//普通服务的不同之处，onBind()方法不在打酱油，而是会返回一个实例
```

3）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）。

②**Activity代码段**

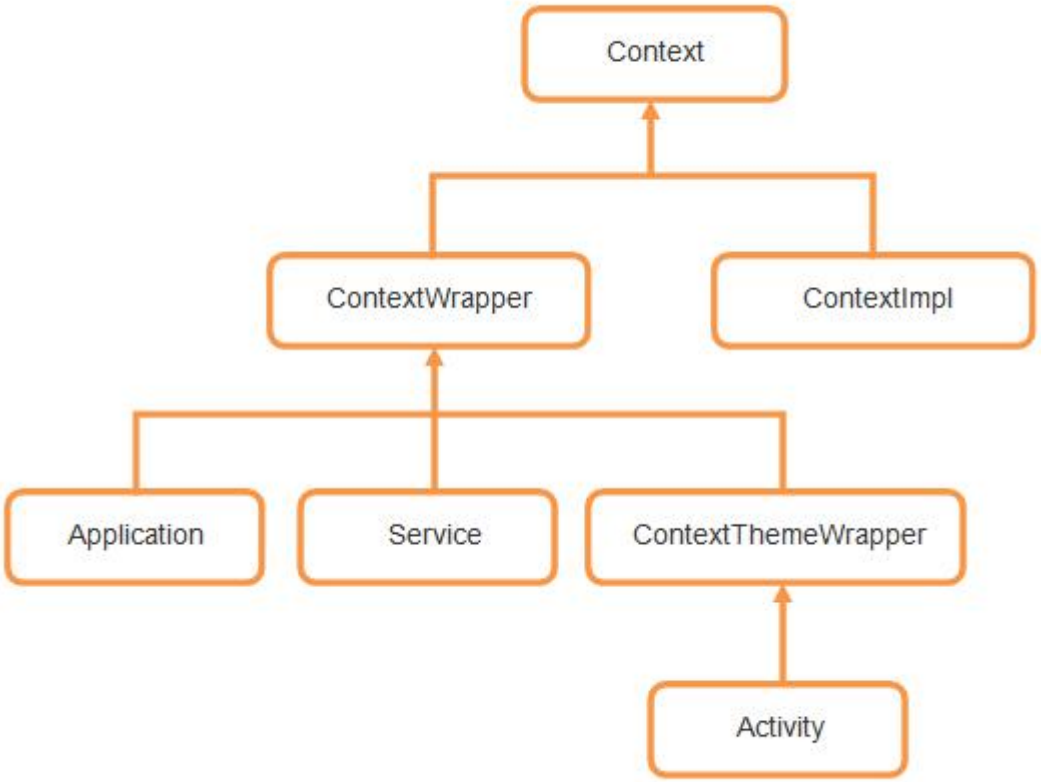
```
public class MainActivity extends AppCompatActivity implements View.OnClickListener{

    private MyService.DownloadBinder downloadBinder;

    private ServiceConnection connection = new ServiceConnection() {
                //可交互的后台服务与普通服务的不同之处，就在于这个connection建立起了两者的联系
        @Override
        public void onServiceDisconnected(ComponentName name) {
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            downloadBinder = (MyService.DownloadBinder) service;
            downloadBinder.startDownload();
            downloadBinder.getProgress();
        }//onServiceConnected()方法关键，在这里实现对服务的方法的调用
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Button bindService = (Button) findViewById(R.id.bind_service);
        Button unbindService = (Button) findViewById(R.id.unbind_service);
        bindService.setOnClickListener(this);
        unbindService.setOnClickListener(this);

    }

    @Override
    public void onClick(View v) {
        switch (v.getId()) {
            case R.id.bind_service:
                Intent bindIntent = new Intent(this, MyService.class);
                bindService(bindIntent, connection, BIND_AUTO_CREATE); // 绑定服务和活动，之后活动就可以去
                break;
            case R.id.unbind_service:
                unbindService(connection); // 解绑服务，服务要记得解绑，不要造成内存泄漏
                break;
            default:
                break;
        }
    }

}
```

15. ApplicationContext 和 ActivityContext 的区别(源码)

ApplicationContext 的生命周期与 Application 的生命周期

相关的，ApplicationContext 随着 Application 的销毁而销毁，伴随 application 的一生，与 activity 的生命周期无关.

ActivityContext 跟 Activity 的生命周期是相关的，但是对一个 Application 来说，Activity 可以销毁几次，那么属于 Activity 的 context 就会销毁多次



Context 一共有 Application、Activity 和 Service 三种类型，因此一个应用程序中 Context 数量的计算公式就可以这样写：

Context 数量 = Activity 数量 + Service 数量 + 1

上面的 1 代表着 Application 的数量，因为一个应用程序中可以有多个 Activity 和多个 Service，但是只能有一个 Application。

和 UI 相关的方法基本都不建议或者不可使用 Application，并且，前三个操作基本不可能在 Application 中出现。实际上，只要把握住一点，**凡是跟 UI 相关的，都应该使用 Activity 做为 Context 来处理；其他的一些操作，Service,Activity,Application 等实例都可以**，当然了，注意 Context 引用的持有，防止内存泄漏。

还有就是，在使用 context 的时候，为防止内存泄露，注意一下几个方面：

(1)不要让生命周期长的对象引用 activity context，即保证引用 activity 的对象要与 activity 本身生命周期是一样的。

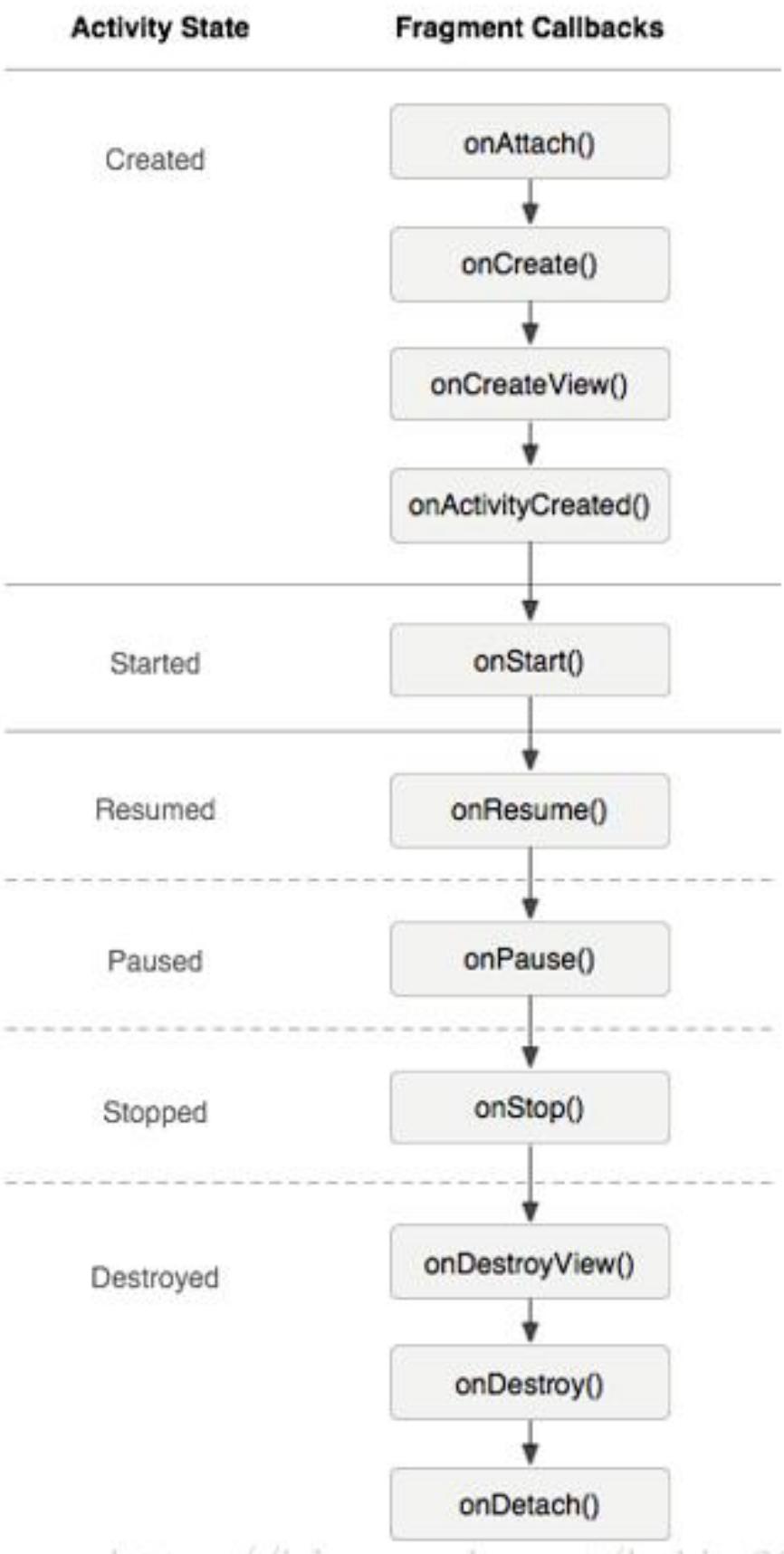
(2)对于生命周期长的对象，可以使用 application context。

(3)避免非静态的内部类，尽量使用静态类，避免生命周期问题，注意内部类对外部对象引用导致的生命周期变化。

16. Application 和 Activity 的 Context 对象的区别(应该跟上面是同一个问题)

17. Activity与Fragment之间生命周期比较

Fragment 是 Activity 的一个组件片段，也就是说他的生命周期是依赖于 Activity 的



但是它比 Activity 多了几个生命步骤，

(1)首先是 Activity#onCreate 中 Fragment 会执行 onAttach（）、onCreate()、onCreateView()、onActivityCreated()

(2)接着 Activity#onStart()中 Fragment 也会执行 onStart（）

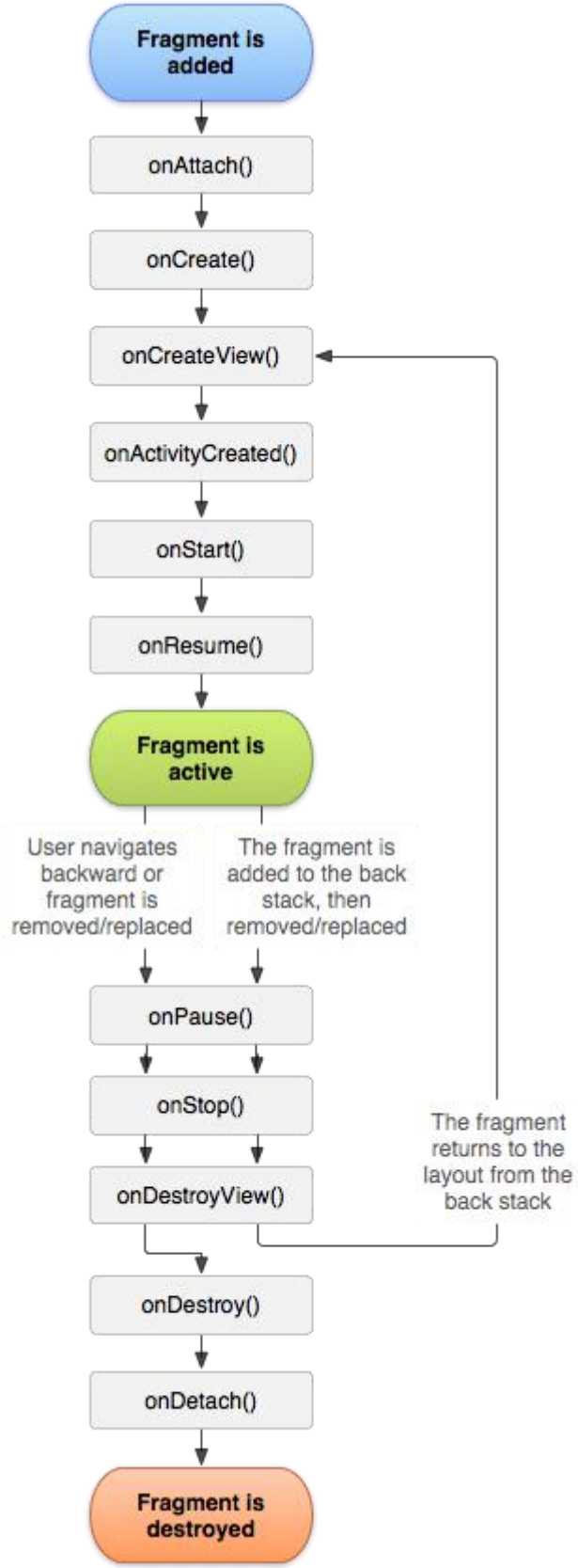
(3)然后是 Activity#onResume()中 Fragment 也会执行 onResume（）

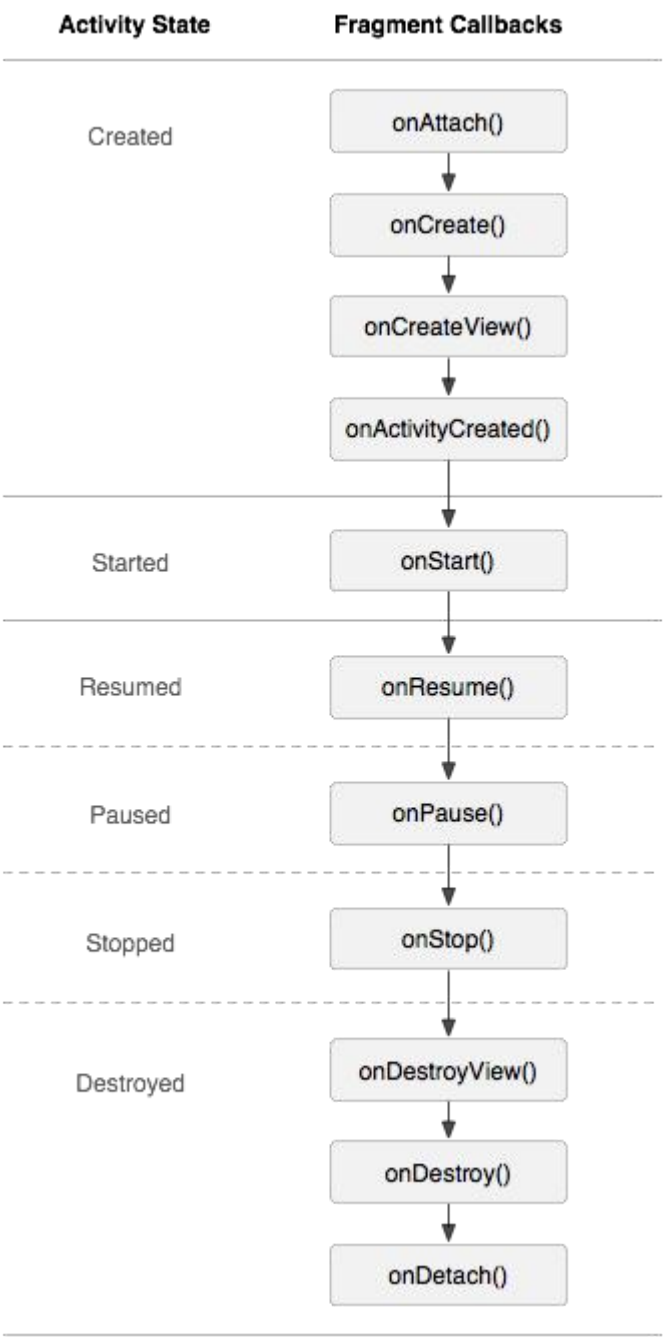
(4)接着是 Activity#onPause()中 Fragment 也会执行 onPause()(先于 Activity 执行)

(5)然后是 Activity#onStop()中 Fragment 也会执行 onStop（）(先于 Activity 执行)

(6)接着是 Activity#onDestroy()中 Fragment 会执行 onDestroyView（）、onDestroy()、onDetach()(先于 Activity 执行)

18. fragment 各种情况下的生命周期


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed



根据使用分五种情况：

(1) Fragment 在 Activity 中 replace 新替换的 Fragment：

onAttach > onCreate > onCreateView > onViewCreated > onActivityCreat

ed > onStart > onResume

被替换的 Fragment：onPause > onStop > onDestroyView > onDestroy > onDetach

备注：addToBackStack（）会在新 Fragment 启动之前保存旧的 Fragment，当新的 Fragment 退出时，旧的 Fragment 就会重新加载

(2) Fragment 在 Activity 中 replace，并 addToBackStack

如果新替换的 Fragment 没有在 BackStack 中：onAttach > onCreate > onCreateView > onViewCreated > onActivityCreated > onStart > onResume

如果新替换的 Fragment 已经在 BackStack 中：onCreateView > onViewCreated > onActivityCreated > onStart > onResume

被替换的 Fragment：onPause > onStop > onDestroyView

(3) Fragment 在 ViewPager 中切换

这里有很多种情况，

切换前的的 Fragment 称为 PreviousFragment，简称 PF；

切换后的 Fragment 称为 NextFragment，简称 NF；

其他 Fragment 称为 OtherFragment，简称 OF。

（在 ViewPager 中 setUserVisibleHint 能反映出 Fragment 是否被切换到后台或前台，所以在这里也当作生命周期）

A、如果相关的 Fragment 没有被加载过：（多 onAttach > onCreate 流程）

NF： setUserVisibleHint(false) > onAttach > onCreate > setUserVisibleHint(true) > onCreateView > onViewCreated > onActivityCreated > onStart > onResume

OF 跟 NF 相邻： setUserVisibleHint(false) > onAttach > onCreate > onCreateView > onViewCreated > onActivityCreated > onStart > onResume

B、**如果相关的 Fragment 已经被加载过：**（少 onAttach > onCreate ）

创建时：

NF 跟 PF 相邻 ：setUserVisibleHint(true)

相邻的会预先加载，所以放置前台即可

NF 跟 PF 不相邻:setUserVisibleHint(true) > onCreateView > onViewCreated > onActivityCreated > onStart > onResume

不相邻则不会预先加载，故要重走一遍创建流程

OF 跟 NF 相邻 ： onCreateView > onViewCreated > onActivityCreated > onStart > onResume

与切换后的界面相邻，需要预先加载，故启动创建流程

销毁时：

PF 跟 NF 相邻 ：setUserVisibleHint(false)

相邻放置后台即可，不必执行销毁

PF 跟 NF 不相邻：setUserVisibleHint(false) > onPause > onStop > onDestroyView

不相邻，销毁即可

OF 跟 PF 相邻：onPause > onStop > onDestroyView

与切换后的页面不相邻，销毁

OF 夹在 PF 和 NF 中间：不调用任何生命周期方法

C、如果重写了 FragmentPagerAdapter 的 destroyItem 方法，并且相关 Fragment 已经加载过：

则相互切换时只会调用 setUserVisibleHint

(4) Fragment 进入了运行状态：

Fragment 在上述的各种情况下进入了 onResume 后，则进入了运行状态，以下 4 个生命周期方法将跟随所属的 Activity 一起被调用：

onPause > onStop > onStart > onResume

(5) 关于 Fragment 的 onActivityResult 方法：

在重写 FragmentActivity 的 onActivityResult 方法时，注意调 super.onActivityResult。因为在使用 Fragment 的 startActivity 方法时，FragmentActivity 的 onActivityResult 方法会回调相应的 Fragment 的 onActivityResult 方法。

19. Fragment 状态保存

startActivityForResult 是哪个类的方法，在什么情况下使用？

Fragment 调用 startActivityForResult --->

HostCallbacks . onStartActivityFromFragment --->

FragmentActivity . startActivityFromFragment

FragmentActivity 的 startActivityFromFragment 方法步骤：

```
public void startActivityFromFragment(Fragment fragment, Intent intent,
        int requestCode, @Nullable Bundle options) {
    mStartedActivityFromFragment = true;
    try {
        if (requestCode == -1) {
            ActivityCompat.startActivityForResult(this, intent, -1, options);
            return;
        }
        checkForValidRequestCode(requestCode);
        int requestIndex = allocateRequestIndex(fragment);
        ActivityCompat.startActivityForResult(
                this, intent, ((requestIndex + 1) << 16) + (requestCode & 0xffff), options);
    } finally {
        mStartedActivityFromFragment = false;
    }
}
```

首先将 FragmentActivity 父类 BaseFragmentActivityJB 的 mStartedActivityFromFragment 设置为 true；

然后根据 requestCode 的值判断，调用 startActivity 默认这里的 requestCode 为 -1 ，如果调用 startActivityForResult 则不会走这步；

第二部通过 checkForValidRequestCode 会查验 requestCode 是否正确

第三步调用 allocateRequestIndex 方法得到请求队列中的索引

第四步调用

ActivityCompat . startIntentSenderForResult(this, intent,((requestIndex + 1) << 16) + (requestCode & 0xffff), fillInIntent,flagsMask, flagsValues, extraFlags, options);

(requestIndex + 1) << 16 代表将请求的索引值左移 16 位,在较高 16 位中保存当前的索引值

requestCode & 0xffff 就是 requestCode,为了屏蔽二进制位,置位 0

ActivityCompat . startIntentSenderForResult（）方法：

```
public static void startIntentSenderForResult(Activity activity, IntentSender intent,
        int requestCode, Intent fillInIntent, int flagsMask, int flagsValues,
        int extraFlags, @Nullable Bundle options) throws IntentSender.SendIntentException {
    if (Build.VERSION.SDK_INT >= 16) {
        ActivityCompatJB.startIntentSenderForResult(activity, intent, requestCode, fillInIntent,
                flagsMask, flagsValues, extraFlags, options);
    } else {
        activity.startIntentSenderForResult(intent, requestCode, fillInIntent, flagsMask,
                flagsValues, extraFlags);
    }
}
```

最后调用 Activity 的 startIntentSenderForResult 方法，ActivityManagerNative.getDefault().startActivityIntentSender 真正启动一个 Activity。

最后接收流程：

FragmentActivity 中接收 setResult 的值的方法 onActivityResult

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    mFragments.noteStateNotSaved();
    int requestIndex = requestCode>>16;
    if (requestIndex != 0) {
        requestIndex--;

        String who = mPendingFragmentActivityResults.get(requestIndex);
        mPendingFragmentActivityResults.remove(requestIndex);
        if (who == null) {
            Log.w(TAG, "Activity result delivered for unknown Fragment.");
            return;
        }
        Fragment targetFragment = mFragments.findFragmentByWho(who);
        if (targetFragment == null) {
            Log.w(TAG, "Activity result no fragment exists for who: " + who);
        } else {
            targetFragment.onActivityResult(requestCode & 0xffff, resultCode, data);
        }
        return;
    }

    super.onActivityResult(requestCode, resultCode, data);
}
```

首先将 requestCode 值右移 16 位计算出 requestIndex：得到他在等待处理 Fragment 数据队列的索引，由上面的 ActivityCompat.startIntentSenderForResult 方法参数得知，requestCode 是左移了 16 位。

第二步根据 requestIndex 的值判断是 Fragment 调用了 startActivityForResult 还是 Activity，

如果 requestIndex 不为 0 代表是 Fragment 触发，下面得到 targetFragment 然后可以通过 targetFragment 的 onActivityResult 方法得到返回值；

如果 requestIndex 为 0 代表是 Activity 调用了

startActivityForResult，那么直接返回的是 Activity 的 onActivityResult 方法。

Frament 中需要重写 onActivityResult，因为原方法是空的

需要注意：

Fragment 和 Activity 都有 startActivityForResult 方法，切记不要调用错误。如果发现 Fragment 的 onActivityResult 拿不到数据，而 FragmentActivity 拿到了数据，并且 requestCode 与 Fragment 中用到的 requestCode 不一致，那么一定是在 Fragment 中调用了 FragmentActivity 的 startActivityForResult 方法。

源码就可以看出 requestCode 不一致的原因了，使用 Fragment 发起的请求，因为一个 Activity 可以包含多个 Fragment，而且都可以发起请求，那么会使用一个容器来保存发起请求的 Fragment 和它对应的 requestCode，mPendingFragmentActivityResults 就是这个容器，发起请求的时候将在容器中的 index 和 requestCode 算在一起，作为最终的 requestCode。

FragmentActivity 和 它的内部类 HostCallbacks 分别展现了两个 startActivityForResult 方法的调用。

FragmentActi 本身是有 startActivityForResult 方法，依附它的 Fragment 也有 startActivityForResult 方法，那么放在 HostCallbacks 里，这就体现了 HostCallbacks 是一座桥梁，链接了 Fragment 和 FragmentActivity 之间的通信。

问题：当一个 Activity 中包含一个 Fragment 时我们使用 startActivityForResult()方法启动一个 Activity，待这个被启动的 Activity 设置了一个返回值然后被销毁，如何在 Activity 和 Fragment 中获得这个返回值呢？

启动 Activity 时，我们一般会设置一个 requestCode 以便在开启了多个具有返回值的 Activity 时进行识别以做出不同的响应对策。但使用不同的启动方式，在 Activity 和 Fragment 中拦截到的结果不同，主要有以下几种情况：

1. **在 Activity 中启动并获得结果：** 可以在 Activity 的 onActivityResult()方法中根据 requestCode 就可以获得返回值，无论是否调用 super.onActivityResult()方法，其中的 Fragment 不会获得任何值

2. **在 Fragment 中启动并在 Activity 中获得结果：**在 Fragment 中使用 getActivity().startActivityForResult()方法启动，可以在 Activity 的 onActivityResult()方法中根据 requestCode 就可以获得返回值，无论是否在 Activity 的 onActivityResult()调用 super.onActivityResult()方法，其中的 Fragment 不会获得任何值

3. **在 Fragment 中启动并在 Fragment 中获得结果：**在 Fragment 中使用 startActivityForResult()方法启动，可以在 Fragment 的 onActivityResult()方法中根据 requestCode 就可以获得返回值，必须调用 super.onActivityResult()方法

注意：对于 3 来说，如果在 Activity 的 onActivityResult()方法中没有调用 super.onActivityResult()方法，那么在 Fragment 中不能获取任何数据，在 Activity 中可以获得数据但是无法根据 requestCode 去识别，传入的是一个 6 位的 requestCode

20. 如何实现 Fragment 的滑动？

1. 把 Fragment 放到 ViewPager 里面去
2. 把 Fragment 放到 RecyclerView/ListView/GridView

21. fragment 之间传递数据的方式？

1. Intent 传值
2. 广播传值
3. 静态调用
4. 本地化存储传值
5. 暴露接口/方法调用
6. eventBus 等之类的

22. 说说 Activity、Intent、Service 是什么关系(源码)

一个 Activity 通常是一个单独的屏幕，每一个 Activity 都被实现为一个单独的类，这些类都是从 Activity 基类中继承而来的。

Activity 类会显示由视图控件组成的用户接口，并对视图控件的事件做出响应。

Intent 的调用是用来进行屏幕之间的切换。Intent 描述应用想要做什么。Intent 数据结构中两个最重要的部分是动作和动作对应的数据，一个动作对应一个动作数据。

Service 是运行在后台的代码，不能与用户交互，可以运行在自己的进程里，也可以运行在其他应用程序进程的上下文里。需要一个 Activity 或者其他 Context 对象来调用。

Activity 跳转 Activity,Activity 启动 Service,Service 打开 Activity 都需要 Intent 表明意图，以及传递参数，Intent 是这些组件间信号传递的承载着

23. IntentService 原理及作用是什么？(源码)

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 的问题生成一个默认的且与线程相互独立的工作线程执行所有发送到 onStartCommand()方法的 Intent,可以在 onHandleIntent()中处理. 串行队列,每次只运行一个任务,不存在线程安全问题,所有任务执行完后自动停止服务,不需要自己手动调

用 stopSelf()来停止

24. 怎么在 Activity 中启动自己对应的 Service?

1、Service 中需要实现 onBind 方法。

2、**创建 ServiceConnection**：需要实现一个新的 ServiceConnection，重现 onServiceConnected 和 OnServiceDisconnected 方法，一旦连接建立，就能得到 Service 实例的引用。

3、**执行绑定**：调用 bindService 方法，传入一个选择了要绑定的 Service 的 Intent(显示或隐式)和一个你实现了的 ServiceConnection 的实例

```
    //绑定服务
    bindService(new Intent(this, MyService.class), new MyServiceConnection(), Context.BIND_AUTO_CREATE);
}

//通过ServiceConnection获得binder对象
public class MyServiceConnection implements ServiceConnection{
    @Override
    public void onServiceConnected(ComponentName name, IBinder service) {
        myBind = (MyService.MyBind) service;
    }

    @Override
    public void onServiceDisconnected(ComponentName name) {

    }
}
```

25. service 和 activity 怎么进行数据

交互？

一、通过 broadcast:通过广播发送消息到 activitry

二、通过 Binder：通过与 activity 进行绑定

1）添加一个继承 Binder 的内部类，并添加相应的逻辑方法。

2）重写 Service 的 onBind 方法，返回我们刚刚定义的那个内部类实例。

3）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）。

26. Service 的开启方式

有两种方法，第一种是

1-1 定义一个类继承 Service，重写其 onBind()的方法

1-2 在清单文件中进行配置

1.3 在 Context 中开启服务 startService(intent)开启服务

1.4 不需要时，关闭服务 stopService(intent)

说明：如果服务已经开启，不会重复的执行 onCreate()， 而是会调用 onStart()和 onStartCommand()。

服务停止的时候调用 onDestory()。服务只会被停止一次。

特点：一旦服务开启跟调用者(开启者)就没有任何关系了。

开启者退出了，开启者挂了，服务还在后台长期的运行。

开启者**不能调用**服务里面的方法。

Service 的第二种启动方法：

2-1 定义一个类，继承 Service

2-2 在清单文件中配置服务

2-3 在 Context 中以 bindService(Intent,ServiceConnection,int)方式开启服务

2-4 不需要时，以 unbindService(ServiceConnection)方式关闭服务

注意：绑定服务不会调用 `onstart()`或者 `onstartcommand()`方法

特点：bind 的方式开启服务，绑定服务，调用者挂了，服务也会跟着挂掉。

绑定者可以调用服务里面的方法。

27. 请描述一下 Service 的生命周期

启动 Service 的方式有两种，各自的生命周期也有所不同。

(1)通过 startService 启动 Service：

onCreate（） >> onStartCommand（） >> onDestory（）

1.如果 Service 还没有运行，则调用 onCreate()然后调用 onStartCommand()；

2.如果 Service 已经运行，则只调用 onStartCommand()，所以一个 Service 的 onStartCommand（）方法可能会重复调用多次。

3.调用 stopService 的时候直接 onDestroy（），

4.生命周期和调用者不同,这时 Service 跟启动的 Activity 没有关联，启动后若调用者未调用 stopService 而直接退出,Service 仍会运行。

(2)通过 bindService 绑定 Service：

onCreate >> onBind >> onUnbind >> onDestory。

1.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。

2.生命周期与调用者绑定,那么这个 Service 就跟启动他的进程有关了调用者一旦退出,Service 就会调用 unBind >> onDestory

3.所以调用 bindService 的生命周期为：onCreate --> onBind(只一次，不可多次绑定) --> onUnbind --> onDestory

28. 谈谈你对 ContentProvider 的理解

是应用程序之间共享数据的接口,如果想将自身的数据共享给其他应用就用这个实现

29. 说说 ContentProvider、ContentResolver、ContentObserver 之间的关系

(1) ContentProvider 内容提供者,用于对外提供数据。

(2) ContentResolver 内容解析者,用于获取内容提供者提供据

(3) ContentResolver.notifyChange(uri)发出消息。

(4) ContentResolver.registerContentObserver()监听消息。

(5) ContentObserver 内容监听器,可以监听数据的改变状

30. 请介绍下 ContentProvider 是如何实现数据共享的？(源码)

使用 ContentProvider 可以将数据共享给其他应用，让除本应用之外的应用也可以访问本应用的数据。它的底层是用 SQLite 数据库实现的，所以其对数据做的各种操作都是以 Sql 实现，只是在上层提供的是 Uri，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。

31. ContentProvider 的权限管理(解答：读写分离，权限控制-精确到表级，URL 控制)（原理）

暂时无法解出

32. 如何通过广播拦截和 abort 一条短信？（原理）

短信接收方式也是通过广播来接收，而且这个广播是有序广播

1. 首先添加接收短信的权限

<uses-permission android:name="android.permission.RECEIVE_SMS"/>

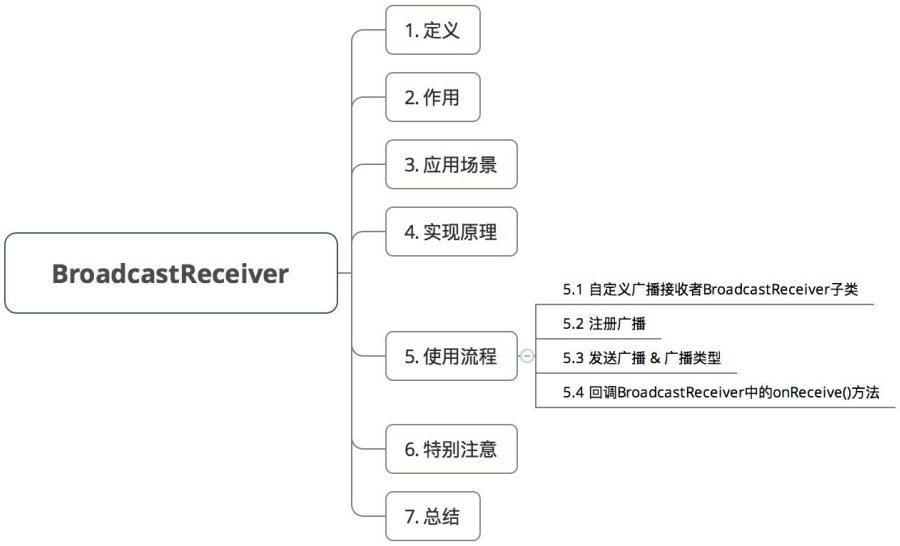
2. 在清单文件中注册广播接收器，设置该广播接收器优先级,尽量设高一点

3. 创建一个 BroadcastReceiver 来实现广播的处理，并设置拦截器 abortBroadcast();

33. 广播是否可以请求网络？（原理）

子线程可以，主线程超过 10s 引起 anr

34. 请描述一下广 BroadcastReceiver 的理解



广播，是一个全局的监听器，属于 Android 四大组件之一。

Android 广播分为两个角色：广播发送者、广播接收者

作用是监听 / 接收 应用 App 发出的广播消息，并 做出响应

可应用在：

1. Android 不同组件间的通信（含 ：应用内 / 不同应用之间）
2. 多线程通信
3. 与 Android 系统在特定情况下的通信

如：电话呼入时、网络可用时

实现原理：

Android 中的广播使用了设计模式中的**观察者模式**：基于消息的发布 / 订阅事件模型。因此，Android 将广播的**发送者 和 接收者 解耦**，使得系统方便集成，更易扩展

有 3 个角色：

1. 消息订阅者（广播接收者）
2. 消息发布者（广播发布者）
3. 消息中心（AMS，即 Activity Manager Service）

(1)广播接收者 通 Binder 机制在 AMS 注册

(2)广播发送者通过 Binder 机制向 AMS 发送广播

(3)AMS 根据广播发送者要求，在已注册列表中，寻找合适的广播接收器（寻找依据：IntentFilter / Permission）

(4)AMS 将广播发送到合适的广播接收者相应的消息循环队列中

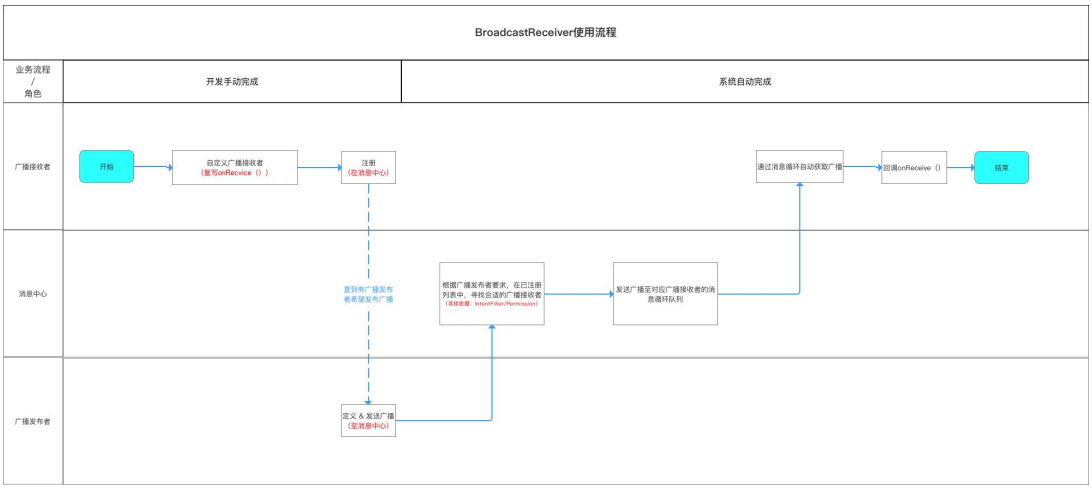
(5)广播接收者通过消息循环拿到此广播，并回调 onReceive（）

（注：广播发送者 和 广播接收者的执行是异步的，即广播发送者不会关心有无接收者接收 &也不确定接收者合适才能接收到）

原理描述

1. 广播接收者 通过 Binder机制在 AMS 注册
2. 广播发送者 通过 Binder 机制向 AMS 发送广播
3. AMS 根据 广播发送者 要求，在已注册列表中，寻找合适的广播接收者
（寻找依据：IntentFilter / Permission）
4. AMS将广播发送到合适的广播接收者相应的消息循环队列中
5. 广播接收者通过 消息循环 拿到此广播，并回调 onReceive()

特别注意：广播发送者 和 广播接收者的执行 是 *异步的，即 广播发送者 不会关心有无接收者接收 & 也不确定接收者何时才能接收到



广播接收器注册

注册的方式分为两种：静态注册、动态注册

静态注册:注册方式:在 AndroidManifest.xml 里通过<receive>标签声明

特点：常驻，不受任何组件的生命周期影响，缺点是耗电、占内存，应用在需要时刻监听广播

动态注册：注册方式：在代码中调用 `Context.registerReceiver()` 方法

特点：非 常驻，灵活，跟随组件的生命周期变化（组件结束 = 广播结束，在组件结束前，必须移除广播接收器）

特别注意：

动态广播最好在 `Activity` 的 `onResume()` 注册、`onPause()` 注销。

原因：对于动态广播，有注册就必然得有注销，否则会导致**内存泄露**

`重复注册、重复注销也不允许`

注册方式	区别		
	使用方式	特点	应用场景
静态注册 （常驻广播）	在AndroidManifest.xml里通过<receive>标签声明	• 常驻，不受任何组件的生命周期影响 （应用程序关闭后，如果有信息广播来，程序依旧会被系统调用） • 缺点：耗电、占内存	需要时刻监听广播
动态注册 （非常驻广播）	在代码中调用Context.registerReceiver（）方法	非常驻，灵活，跟随组件的生命周期变化 （组件结束=广播结束，在组件结束前，必须移除广播接收器）	需要特定时刻监听广播

5.3.1 广播的发送

广播发送 = 广播发送者 将此广播的“意图（Intent）”通过 sendBroadcast（）方法发送出去

5.3.2 广播的类型

广播的类型主要分为 5 类：

1. 普通广播（Normal Broadcast）
2. 系统广播（System Broadcast）
3. 有序广播（Ordered Broadcast）
4. App 应用内广播（Local Broadcast）

1. 普通广播（Normal Broadcast）

即 开发者自身定义 `intent` 的广播（最常用）

2. 系统广播（System Broadcast）

Android 中内置了多个系统广播：只要涉及到手机的基本操作（如开机、网络状态变化、拍照等等），都会发出相应的广播

3. 有序广播（Ordered Broadcast）

定义

发送出去的广播被广播接收者按照先后顺序接收，有序是针对广播接收者而言的

-

广播接受者接收广播的顺序规则（同时面向静态和动态注册的广播接受者）

1. 按照 Priority 属性值从大-小排序；
2. Priority 属性相同者，动态注册的广播优先；

特点：

1. 接收广播按顺序接收
2. 先接收的广播接收者可以对广播进行截断，即后接收的广播接收者不再接收到此广播；
3. 先接收的广播接收者可以对广播进行修改，那么后接收的广播接收者将接收到被修改后的广播

具体使用：

有序广播的使用过程与普通广播非常类似，差异仅在于广播的发送方式：sendOrderedBroadcast(intent);

4. App 应用内广播（Local Broadcast）

使用原因

1. **全局广播容易泄露隐私：**全局的 Broadcast 是不隐私的，其他应用可能获取到应用内部通过 Broadcast 发送的

隐私信息。 通过逆向工程，创建可以收到你的应用广播的BroadcastReceiver，这非常容易导致敏感信息的泄漏。

2. **攻击者可以通过广播来攻击你的应用，从而影响你的安全性能：**你的应也用可以接受其他应用发来的特定IntentFilter 的 Broadcast，这意味着，攻击者可以通过分析构造特定的广播数据，危害根据业务不同和代码的防护性差异而程度不一。比如发送一个空数据的 intent 使你报错。

3. **节省资源，使广播更高效：**如果保需要在应用内通信，全局广播将造成资源上的浪费，Android 为了使得广播更效率，专门设置了优先级，甚至还有阻断拦截。

App 应用内广播可理解为一种局部广播，广播的发送者和接收者都同属于一个 App。相比于全局广播（普通广播），App 应用内广播优势体现在：安全性高 & 效率高

具体使用：

(1) 注册广播时将 exported 属性设置为 *false*，使得非本 App 内部发出的此广播不被接收；

(2) 在广播发送和接收时，增设相应权限 permission，用于权限验证；

(3) 发送广播时指定该广播接收器所在的包名，此广播将只会发送到此包中的 App 内与之相匹配的有效广播接收器中。通过 intent.setPackage(packageName)指定报名

(4) 使用封装好的 LocalBroadcastManager 类，使用方式上与全局广播几乎相同，只是注册/取消注册广播接收器和发送广播时将参数的 context 变成了 LocalBroadcastManager 的单一实例

注：对于 LocalBroadcastManager 方式发送的应用内广播，只能通过 LocalBroadcastManager 动态注册，不能静态注册

35. 广播的分类

https://www.jianshu.com/p/ca3d87a4cdf3

同上，一共四类

36. 广播使用的方式和场景

(1)**静态注册（常驻型广播）**:在清单文件中注册， 常见的有监听设备启动，常驻注册不会随程序生命周期改变，适用于长期监听，

这种常驻型广播当应用程序关闭后，如果有信息广播来，程序也会被系统调用自动运行。

(2)动态注册（非常驻型广播）:在代码中注册，这种方式注册的广播会跟随程序的生命周期。随着程序的结束，也就停止接受广播了。使用与一些需要与生命周期同步的监听。

补充一点:有些广播只能通过动态方式注册，比如时间变化事件、屏幕亮灭事件、电量变更事件，因为这些事件触发频率通常很高，如果允许后台监听，会导致进程频繁创建和销毁，从而影响系统整体性能

37. 在 manifest 和代码中如何注册和使用 BroadcastReceiver?

首先写一个类要继承 BroadcastReceiver

第一种:在清单文件中声明,添加

第二种使用代码进行注册如:

创建 IntentFilter ，并将要广播接收器接收的参数传进去

创建广播接收器，New 出一个广播接收器

接着使用 registerReceiver（）方法，将上述创建的两个参数传进去

```
IntentFilter filter = new IntentFilter("android.p
```

```
rovider.Telephony.SMS_RECEIVED");
IncomingSMSReceiver receiver = new IncomgSMSReceive
r();
registerReceiver(receiver.filter);
```

38. 本地广播和全局广播有什么差别？

本地广播和全局广播的差别

BroadcastReceiver 是针对应用间、应用与系统间、应用内部进行通信的一种方式

LocalBroadcastReceiver 仅在自己的应用内发送接收广播，也就是只有自己的应用能收到，数据更加安全广播只在这个程序里，而且效率更高。

BroadcastReceiver 使用

(1) 制作 intent（可以携带参数）

使用 sendBroadcast()传入 intent;

(2) 制作广播接收器类继承 BroadcastReceiver 重写 onReceive 方法（或者可以匿名内部类啥的）

(3) 在 java 中（动态注册）或者直接在 Manifest 中注册广播接收器（静态注册）使用 registerReceiver()传入接收器和 intentFilter 取消注册可以在 OnDestroy()函数中，unregisterReceiver()传入接收器

LocalBroadcastReceiver 使用

LocalBroadcastReceiver 不能静态注册，只能采用动态注册的方式。在发送和注册的时候采用，LocalBroadcastManager 的 sendBroadcast 方法和 registerReceiver 方法

39. BroadcastReceiver，LocalBroadcastReceiver 区别

一、应用场景不同

1、BroadcastReceiver 用于应用之间的传递消息；

2、而 LocalBroadcastManager 用于应用内部传递消息，比 broadcastReceiver 更加高效。

二、使用安全性不同

1、BroadcastReceiver 使用的 Content API，所以本质上它是跨应用的，所以在使用它时必须要考虑到不要被别的应用滥用；

2、LocalBroadcastManager 不需要考虑安全问题，因为它只在应用内部有效。

40. AlertDialog,popupWindow,Activity 区别

1) AlertDialog ：用来提示用户一些信息,用起来也比较简单,设置标题类容 和按钮即可,如果是加载的自定义的 view ,用 dialog.setView(layout);加载布局即可(其实就是一个布满全屏的空间)

2) popupWindow：就是一个悬浮在 Activity 之上的窗口，可以用展示任意布局文件

3) activity：是 Android 系统中的四大组件之一，可以用于显示 View。Activity 是一个与用户交互的系统模块，几乎所有的 Activity 都是和用户进行交互的

区别：AlertDialog 是非阻塞式对话框：AlertDialog 弹出时，后台还可以做事情；

而 PopupWindow 是阻塞式对话框：PopupWindow 弹出时，程序会等待，在 PopupWindow 退出前，程序一直等待，只有当我们调用了 dismiss 方法的后，PopupWindow 退出，程序才会向下执行。