Android Coding Assessment Test Prepare

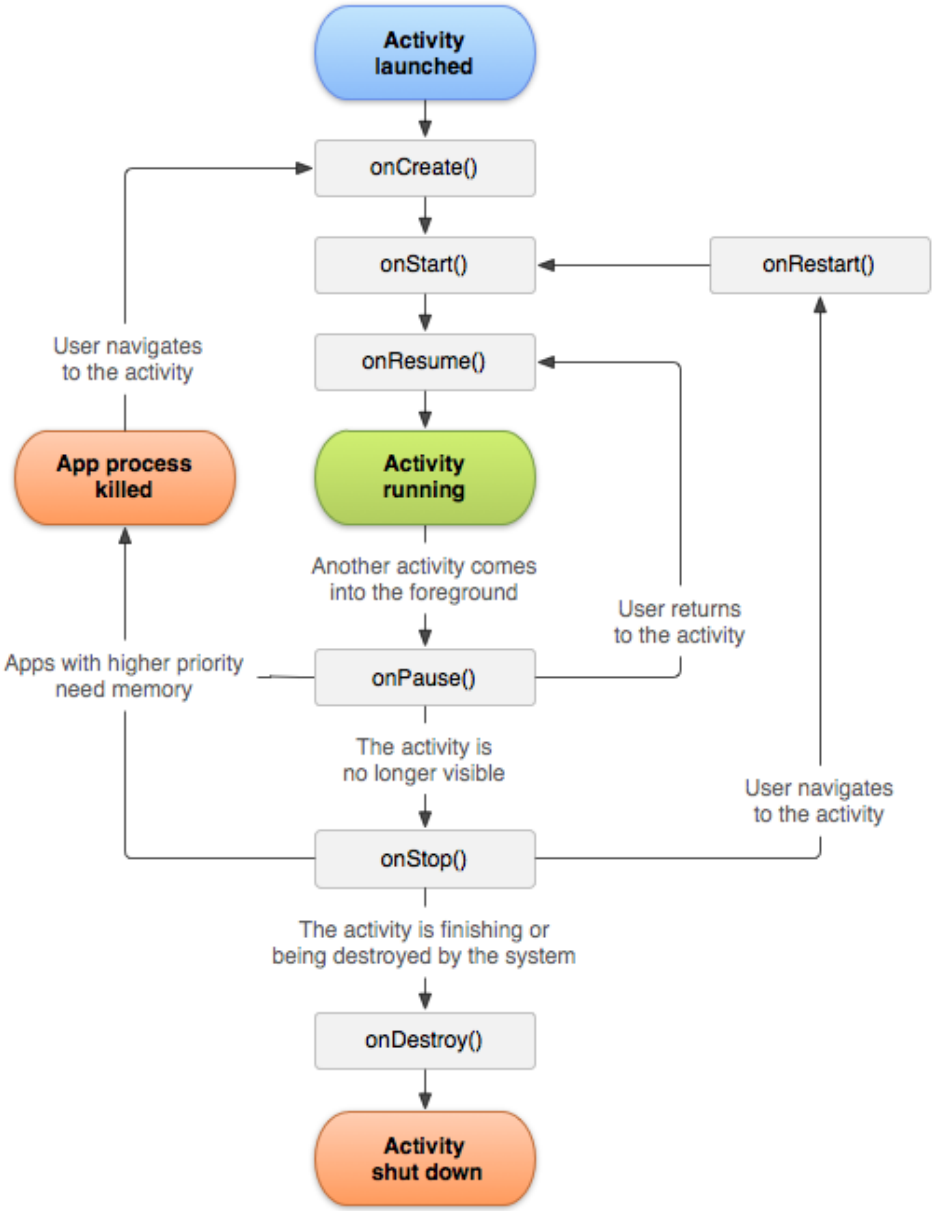
deepwaterooo

August 11, 2022

Contents

1 Activity 生命周期探讨

- https://www.jianshu.com/p/1b3f829810a1



1.1 Activity 生命周期相关函数说明

- (1) onCreate() 是 activity 正在被创建，也就是说此时的 UI 操作不会更新 UI，比如 setText() 操作，所以此时在子线程调用 setText() 不会报线程错误。详解可见 Android 子线程更新 View 的探索，在这个方法内我们可以做一些初始化工作。

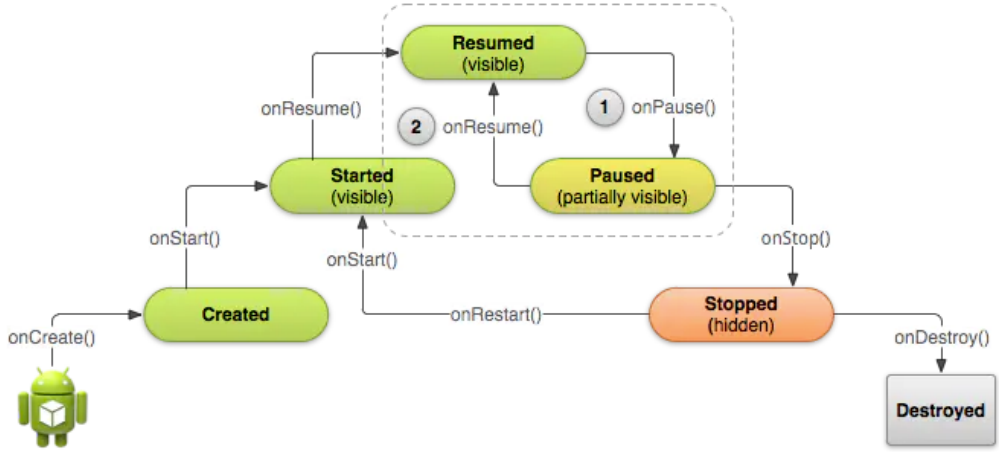
- (2) onRestart() 需要注意的是：activity 正在重新启动，一般情况下，activity 从不可见状态到可见状态，onRestart() 才会被调用，但是一定要注意的是一般来说这是用户行为导致 activity 不可见的时候，此时变为可见的时候才会调用 onRestart()，这里所说的用户行为就是用户按

home 键，或者进入“新”的 activity。这样的操作会使 activity 先执行 onPause()，后执行 onStop()，这样回到这个 activity 会调用 onRestart()。为什么我这里强调说用户行为导致的不可见状态，等下我会说。。。。

- (3) onStart() 的时候，activity 才可见，但是没有出现在前台，无法与用户交互
- (4) onResume() 的时候，activity 已经可见，并且出现在前台开始活动，与 onStart() 相比，activity 都已经可见，但是 onStart() 的时候 activity 还在后台，onResume() 才显示在前台
- (5) onPause() 主要注意的是：此时的 activity 正在被停止，接下来马上调用 onStop()。特殊情况下快速回到该 activity，onStop() 不会执行，会去执行 onResume()。
  - 一般在这个生命周期内做存储数据、停止动画工作，但不能太耗时。
  - 为什么特殊强调呢，因为该 activity 的 onPause() 执行完了，才回去执行新的 activity 的 onResume()，一旦耗时，必然会拖慢新的 activity 的显示。
- (6) onStop()：此时的 activity 即将停止。在这里可以做稍微重量级的操作，同样也不能耗时。
- (7) onDestroy()：此时的 activity 即将被回收，在这里会做一些回收工作和最终资源释放。

1.1.1 在这里着重讲解一下 onStart 与 onResume，onPause 与 onStop 区别

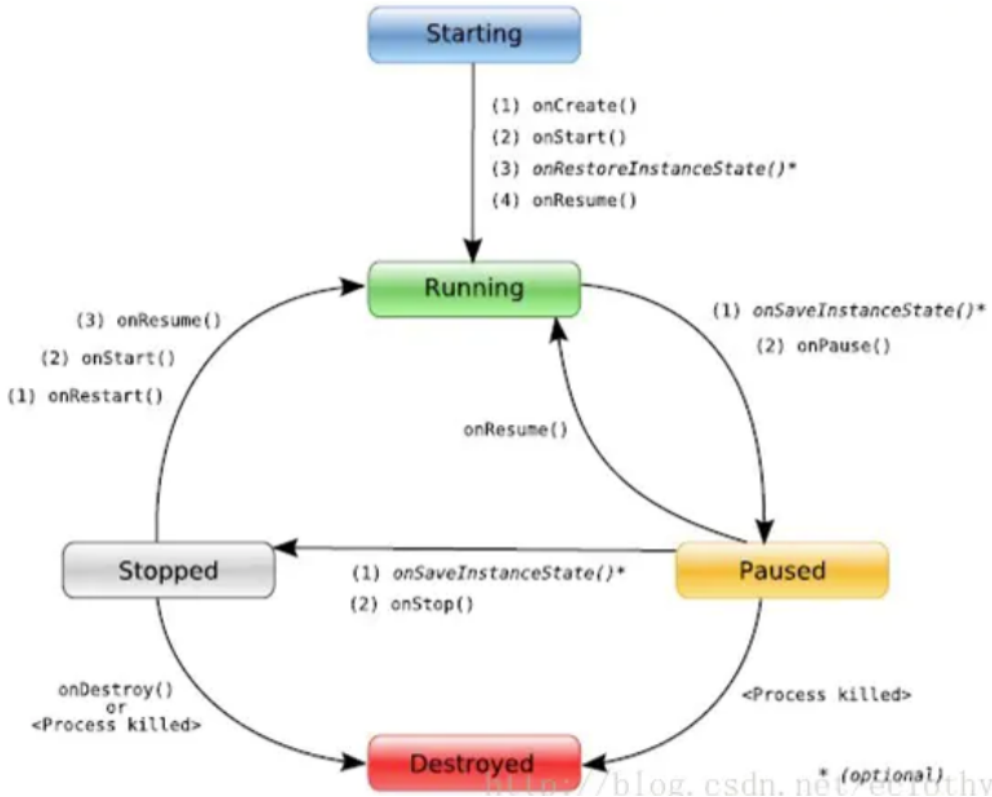
onStart 与 onResume 两种状态虽都可见，但 onStart 时还无法与用户交互，并未获得焦点。onResume 时页面已获得焦点，可与用户交互；onPause 时页面还在前台，只不过页面已失去焦点，无法与用户交互了，onStop 时已不可见了。



activity 四个状态所在的生命周期：

- Running 状态：一个新的 Activity 启动入栈后，它在屏幕最前端，处于栈的最顶端，此时它处于可见并可和用户交互的激活状态。
- Paused 状态：依旧在用户可见状态，但是界面焦点已经失去，此 Activity 无法与用户进行交互。当 Activity 被另一个透明或者 Dialog 样式的 Activity 覆盖时的状态。此时它依然与窗口管理器保持连接，系统继续维护其内部状态，它仍然可见，但它已经失去了焦点，故不可与用户交互。所以就解释为什么启动一个 dialogActivity 或者透明 Activity 时，原 Activity 只执行了 onPause 生命周期，并未执行 onStop
- Stopped 状态：用户看不到当前界面，也无法与用户进行交互完全被覆盖。当 Activity 不可见时，Activity 处于 Stopped 状态。当 Activity 处于此状态时，一定要保存当前数据和当前的 UI 状态，否则一旦 Activity 退出或关闭时，当前的数据和 UI 状态就丢失了。

- Killed 状态：Activity 被杀掉以后或者被启动以前，处于 Killed 状态。这是 Activity 已从 Activity 堆栈中移除，需要重新启动才可以显示和使用。
- 4 种状态中，Running 状态和 Paused 状态是可见的，Stopped 状态和 Killed 状态时不可见的。



1.2 Activity 注意事项

- Activity 中所有和状态相关的回调函数：

InstanceSate(outState)	
onPause()	Activity暂停时被调用。导致暂停的原因除了onStop()中描述四个原因外，还包括一个，即当用户长按"Home"键出现最近任务列表时，此时正在运行的Activity将被执行onPause()
onCreateDescription()	仅在要停止Activity时调用，先于onStop()
onStop()	一般会导致变为stop状态的原因有以下几个： • 用户按"Back"键后 • 用户正在运行Activity时，按"Home"键 • 程序中调用finish()后 • 用户从A启动B后，A就会变为stop状态
onDestroy()	当Activity被销毁时，销毁的情况包括： • 当用户按下"Back"键后 • 程序中调用finish()后

回调函数名称	使用场合
onCreate()	Activity实例被创建后第一次运行时
onNewIntent()	有两种情况会执行该回调： • Intent的Flag中包含CLEAR_TOP，并且目标Activity已经存在当前任务队列中。 • Intent的Flag中包括SINGLE_TOP，并且目标 • Activity已经存在当前任务队列中。 前者和后者的区别在于，举例如下：当前任务队列为ABCD， 此时目标Activity为B，那么前者会把队列改变为AB，而后者则会改变为ABCDB； 但假设当前为ABCD，目标为D，前者则会改变为ABCD，后者则还是ABCD，而不会重新创建D对象，即SINGLE_TOP只对目标在top上时才有效
onStart()	Activity从stop状态重新运行时
onRestore InstanceState(Bundle savedInstance)	与onPostCreate()相同，只是先于onPostCreate()调用
onPostCreate()	如果Activity实例是第一次启动，则不调用，否则，以后的每次重新启动都会调用
onResume()	Activity继续运行时
onSave	与onPause()相同，只是会先于onPause()调用

回调函数名称	使用场合
onCreate()	Activity实例被创建后第一次运行时
onNewIntent()	有两种情况会执行该回调： • Intent的Flag中包含CLEAR_TOP，并且目标Activity已经存在当前任务队列中。 • Intent的Flag中包括SINGLE_TOP，并且目标 • Activity已经存在当前任务队列中。 前者和后者的区别在于，举例如下：当前任务队列为ABCD， 此时目标Activity为B，那么前者会把队列改变为AB，而后者则会改变为ABCDB； 但假设当前为ABCD，目标为D，前者则会改变为ABCD，后者则还是ABCD，而不会重新创建D对象，即SINGLE_TOP只对目标在top上时才有效
onStart()	Activity从stop状态重新运行时
onRestore InstanceState(Bundle savedInstance)	与onPostCreate()相同，只是先于onPostCreate()调用
onPostCreate()	如果Activity实例是第一次启动，则不调用，否则，以后的每次重新启动都会调用
onResume()	Activity继续运行时
onSave	与onPause()相同，只是会先于onPause()调用

- 在这里我会特别提出一个 point，就是异常情况下 activity 被杀死，而后被重新创建的情况。

Figure 1: 异常情况下 activity 的重建过程

- 这张图非常重要，可以帮我们解决异常情况下 activity 如何正常回复的问题
- 当系统停止 activity 时，它会调用 onSaveInstanceState()(过程 1)，如果 activity 被销毁了，但是需要创建同样的实例，系统会把过程 1 中的状态数据传给 onCreate() 和 onRestoreInstanceState()，所以我们要在 onSaveInstanceState() 内做保存参数的动作，在 onRestoreInstanceState() 做获取参数的动作。

```
// Save Activity State
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore); // Save the user's current game state
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);
    super.onSaveInstanceState(savedInstanceState); // Always call the superclass so it can save the view hierarchy st
}
```

- 获取参数操作：

```
// onCreate() 方法
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first
    if (savedInstanceState != null) { // Check whether we're recreating a previously destroyed instance
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE); // Restore value of members from saved state
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else  // Probably initialize members with default values for a new instance
}
```

- 也可以

```
// onRestoreInstanceState() 方法
public void onRestoreInstanceState(Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState); // Always call the superclass so it can restore the view hierar
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE); // Restore state members from saved instance
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

1.3 一些比较重要的回调方法

1.3.1 onWindowFocusChanged()

- 在 Activity 窗口获得或失去焦点时被调用并且当 Activity 被创建时是在 onResume 之后被调用，当 Activity 被覆盖或者退居后台或者当前 Activity 退出时，它是在 onPause 之后被调用（在这个方法中可以 view 已经绘制完成，可以获取 view 的宽高等属性）

1.3.2 onSaveInstanceState()

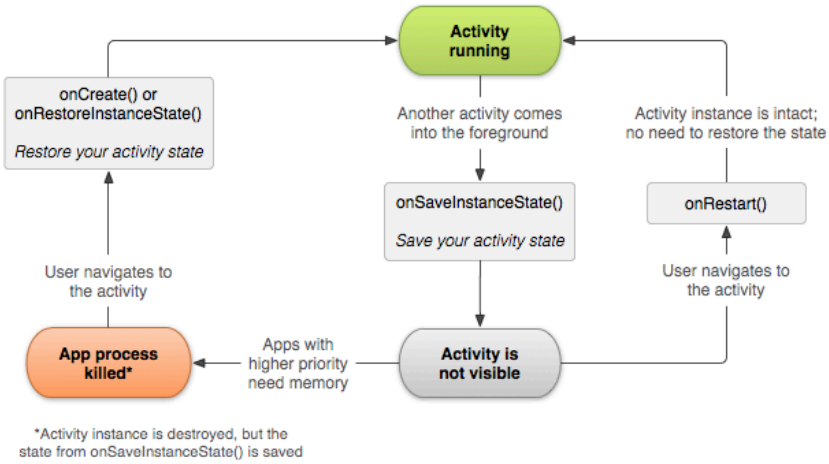
- (1) 在 Activity 被覆盖或退居后台之后，系统资源不足将其杀死，此方法会被调用；
- (2) 在用户改变屏幕方向时，此方法会被调用；
- (3) 在当前 Activity 跳转到其他 Activity 或者按 Home 键回到主屏，自身退居后台时，此方法会被调用。
- 第一种情况我们无法保证什么时候发生，系统根据资源紧张程度去调度；
- 第二种是屏幕翻转方向时，系统先销毁当前的 Activity，然后再重建一个新的，调用此方法时，我们可以保存一些临时数据；
- 第三种情况系统调用此方法是为了保存当前窗口各个 View 组件的状态。
- onSaveInstanceStat()e 的调用顺序是在 onStop() 之前。（android3.0 之前：在 onPause 之前调用，在 3.0 之后，在 onPause 之后调用）
- 下面是其它版本的更为详细一点儿的分析与总结

1.3.3 保存 Activity 的状态

- 当系统为了恢复内存而销毁某项 Activity 时，Activity 对象也会被销毁，因此系统在继续 Activity 时根本无法让其状态保持完好，而是必须在用户返回 Activity 时重建 Activity 对象。但用户并不知道系统销毁 Activity 后又对其进行了重建，因此他们很可能认为 Activity 状态毫无变化。在这种情况下，您可以实现另一个回调方法对有关 Activity 状态的信息进行保存，以确保有关 Activity 状态的重要信息得到保留：**onSaveInstanceState()**。

- 系统会先调用 onSaveInstanceState()，然后再使 Activity 变得易于销毁。

  - 系统会向该方法传递一个 Bundle，您可以在其中使用 putString() 和 putInt() 等方法以名称-值对形式保存有关 Activity 状态的信息。
  - 然后，如果系统终止您的应用进程，并且当用户返回您的 Activity，则系统会重建该 Activity，并将 Bundle 同时传递给 onCreate() 和 onRestoreInstanceState()。
  - 您可以使用上述任一方法从 Bundle 提取您保存的状态并恢复该 Activity 状态。如果没有状态信息需要恢复，则传递给您的 Bundle 是空值（如果是首次创建该 Activity，就会出现这种情况）。



- 在两种情况下，Activity 重获用户焦点时可保持状态完好：系统在销毁 Activity 后重建 Activity，Activity 必须恢复之前保存的状态；系统停止 Activity 后继续执行 Activity，并且 Activity 状态保持完好。

  - 注：无法保证系统会在销毁您的 Activity 前调用 onSaveInstanceState()，因为存在不需要保存状态的情况（例如用户使用“返回”按钮离开您的 Activity 时，因为用户的行为是在显式关闭 Activity）。如果系统调用 onSaveInstanceState()，它会在调用 onStop() 之前，并且可能会在调用 onPause() 之前进行调用。

- 不过，即使您什么都不做，也不实现 onSaveInstanceState()，Activity 类的 onSaveInstanceState() 默认实现也会恢复部分 Activity 状态。具体地讲，默认实现会为布局中的每个 View 调用相应的 onSaveInstanceState() 方法，让每个视图都能提供有关自身的应保存信息。Android 框架中几乎每个小工具都会根据需要实现此方法，以便在重建 Activity 时自动保存和恢复对 UI 所做的任何可见更改。例如，EditText 小工具保存用户输入的任何文本，CheckBox 小工具保存复选框的选中或未选中状态。您只需为想要保存其状态的每个小工具提供一个唯一的 ID（通过 android:id 属性）。如果小工具没有 ID，则系统无法保存其状态。

- 您还可以通过将 android:saveEnabled 属性设置为”false”或通过调用 setSaveEnabled() 方法显式阻止布局内的视图保存其状态。您通常不应将该属性禁用，但如果您想以不同方式恢复 Activity UI 的状态，就可能需要这样做。

- 注：由于无法保证系统会调用 onSaveInstanceState()，因此您只应利用它来记录 Activity 的瞬态（UI 的状态）——切勿使用它来存储持久性数据，而应使用 onPause() 在用户离开 Activity 后存储持久性数据（例如应保存到数据库的数据）。

1.3.4 onRestoreInstanceState()

- 有的人说这个方法和 onSaveInstanceState 是一对，其实不然，
- (1) 在 Activity 被覆盖或退居后台之后，系统资源不足将其杀死，然后用户又回到了此 Activity，此方法会被调用；
- (2) 在用户改变屏幕方向时，重建的过程中，此方法会被调用。
- 我们可以重写此方法，以便可以恢复一些临时数据。
- onRestoreInstanceState 的调用顺序是在 onStart 之后。
- 在当前 Activity 跳转到其他 Activity 或者按 Home 键回到主屏，自身退居后台时：onRestoreInstanceState 不会调用，但是 onSaveInstanceState 会调用，这点就是区别 (验证一下)

1.3.5 注意一下两个函数在生命周期中的调用顺序

```
onCreate()
onStart()
onRestoreInstanceState() // onStart() 之后恢复数据
onResume()
onPause()
onSaveInstanceState()    // onStop() 之前保存数据
onStop()
onDestory()
```

1.4 最后的总结

- 当 Activity 被系统撤销后重新建立时，保存以及恢复数据的函数调用顺序是：
  - onSaveInstanceState(保存数据) —-> onStop() —-> onCreate(恢复数据 allstate) —-> onStart() —-> onRestoryInstanceState(恢复数据 HierarchyState) —> onResume()
- 如果要取消切换屏幕方法重建 activity，可以配置 configChanges 属性：当支持的最小 sdk 版本大于 android4.0 需要设置这个属性）
  - 其中 Bundle 数据会传到 onCreate()（不一定有数据）和 onRestoreInstanceState()（一定有数据）。
  - Activity 在 manifest.xml 文件中可以指定参数 android：ConfigChanges，用于捕获手机状态的改变
  - 在 Activity 中添加了 android:configChanges 属性，在当所指定属性 (Configuration Changes) 发生改变时，通知程序调用 onConfigurationChanged() 函数。
- 设置方法：将下列字段用“|”符号分隔开，例如：“locale|navigation|orientation”
  - “mcc “移动国家号码，由三位数字组成，每个国家都有自己独立的 MCC，可以识别手机用户所属国家。
  - “mnc “移动网号，在一个国家或者地区中，用于区分手机用户的服务商。
  - “locale “所在地区发生变化。
  - “touchscreen “触摸屏已经改变。(这不应该常发生。)

  - “keyboard “键盘模式发生变化，例如：用户接入外部键盘输入。
  - “keyboardHidden “用户打开手机硬件键盘
  - “navigation “导航型发生了变化。(这不应该常发生。)
  - “orientation “设备旋转，横向显示和竖向显示模式切换。
  - “fontScale “全局字体大小缩放发生改变

- 对 android:configChanges 属性，一般认为有以下几点：(这里面的次数是怎么数的，还没有搞懂)
  - 1、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
  - 2、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
  - 3、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
- 自从 Android 3.2(API 13),在设置 Activity 的 android:configChanges="orientation|keyboardHidde
后，还是一样会重新调用各个生命周期的。因为 screen size 也开始跟着设备的横竖切换而改变。所以，在 AndroidManifest.xml 里设置的 MiniSdkVersion 和 TargetSdkVersion 属性大于等于 13 的情况下，如果你想阻止程序在运行时重新加载 Activity，除了设置"orientation"，你还必须设置"ScreenSize"。
  - 解决方法：AndroidManifest.xml 中设置

```
android:configChanges="orientation|screenSize “
```

```
android:configChanges="keyboardHidden|orientation|screenSize (当支持的最小 sdk 版本大于 android4.0 需要设置这个属性) "
```

1.5 他人总结

- 横竖屏切换时 Activity 的生命周期

1、不设置 Activity 的 android:configChanges 时，切屏会重新调用各个生命周期默认首先销毁当前 activity，然后重新加载。如下图，当横竖屏切换时先执行 onPause/onStop 方法

```
onPause()
onStop()
onCreate()
onStart()
onResume()
```

2、设置 Activity 的 android:configChanges="orientation|keyboardHidden|screenSize" 时，切屏不会重新调用各个生命周期，只会执行 onConfigurationChanged 方法。

- https://juejin.im/post/5a18f58651882531bb6c82e2
- 1. 打开一个全新的 activityA：
  - onCreate() —-> onStart() —-> onResume()
- 2. 从 activity A —-> activity B（全屏)：
  - activity A 先执行: onPause()
  - 然后 activity B 执行: onCreate() —-> onStart() —-> onResume()
  - activity A 再执行: onStop()

- 3. 从 activity A —-> activity B（非全屏）:
  - activity A 先执行: onPause()
  - 然后 activity B 执行: onCreate() —-> onStart() —-> onResume()
  - **activity A 不会执行 onStop()**
- 4. activity B（全屏）返回到 activity A:
  - activity B 先执行: onPause()
  - activity A: onRestart —-> onStart() —-> onResume()
  - activity B 再执行: onStop() —-> onDestory()
- 5. activity B（非全屏）返回到 activity A
  - activity B 先执行: onPause()
  - activity A: onResume()
  - activity B 再执行: onStop() —-> onDestory()
- 6. activity B 返回到 activity A:
  - 如果 activityA 已经被销毁，activityA 会重新创建，执行: onCreate() —-> onStart() —-> onResume()
  - activityB 的流程不变
- 7. activity A 按 home 键退居后台:
  - 同 2 的流程: onPause()
- 8. 再从 home 返回到 activity A
  - 同 4 的流程: onRestart —-> onStart() —-> onResume()

1.6 activity 之间的数据传递：6 种方式

1.6.1 使用 Intent 的 putExtra 传递

- 第一个 Activity 中

```
Intent intent = new Intent(this,TwoActivity.class);
intent.putExtra("data",str);
startActivity(intent);
```

- 第二个 Activity

```
Intent intent = getIntent();
String str = intent.getStringExtra("data");
tv.setText(str);
```

1.6.2 使用 Intent 的 Bundle 传递

- 第一个 Activity 中

```
Intent intent = new Intent(MainActivity.this,TwoActivity.class);
Bundle bundle = new Bundle();
bundle.putString("data", str);
intent.putExtra("bun", bundle);
startActivity(intent);
```

- 第二个 Activity

```
Intent intent = getIntent();
Bundle bundle = intent.getBundleExtra("bun");
String str = bundle.getString("data");
tv.setText(str);
```

1.6.3 使用 Activity 销毁时传递数据

- 第一个 Activity 中

```
Intent intent = new Intent(MainActivity.this,TwoActivity.class);
startActivityForResult(intent, 11);
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    String str = data.getStringExtra("data");
    tvOne.setText(str);
}
```

- 第二个 Activity

```
Intent intent = new Intent();
intent.putExtra("data", edtOne.getText().toString().trim());
setResult(3, intent);
finish();
```

1.6.4 SharedPreferences 传递数据

- 第一个 Activity 中

```
SharedPreferences sp = this.getSharedPreferences("info", 1);
Editor edit = sp.edit();
edit.putString("data", str);
edit.commit();
Intent intent = new Intent(MainActivity.this,TwoActivity.class);
startActivity(intent);
```

- 第二个 Activity

```
SharedPreferences sp = this.getSharedPreferences("info", 1);
tv.setText(sp.getString("data", ""));
```

1.6.5 使用序列化对象 Seriazable

- 这里需要建一个工具类

```
import java.io.Serializable;
class DataBean implements Serializable {
    private String name;
    private String sex;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getSex() {
        return sex;
    }
    public void setSex(String sex) {
        this.sex = sex;
    }
}
```

- 第一个 Activity 中

```
Intent intent = new Intent(MainActivity.this,TwoActivity.class);
DataBean bean = new DataBean();
bean.setName(" 啦啦");
bean.setSex(" 男");
intent.putExtra("key", bean);
startActivity(intent);
```

- 第二个 Activity

```
Intent intent = getIntent();
//反序列化数据对象
Serializable se = intent.getSerializableExtra("key");
if (se instanceof DataBean){
    //获取到携带数据的 DataBean 对象 db
    DataBean db = (DataBean) se;
    tv.setText(db.getName()+"==="+db.getSex());
}
```

1.6.6 使用静态变量传递数据

- 第一个 Activity 中

```
Intent intent = new Intent(MainActivity.this,TwoActivity.class);
TwoActivity.name=" 牛逼";
TwoActivity.str=" 你说";
startActivity(intent);
```

- 第二个 Activity

```
protected static String name;
protected static String str;
tv.setText(str+name);
```

1.6.7 剪切板，利用系统剪切板服务，稍偏

- 第一个 Activity 中

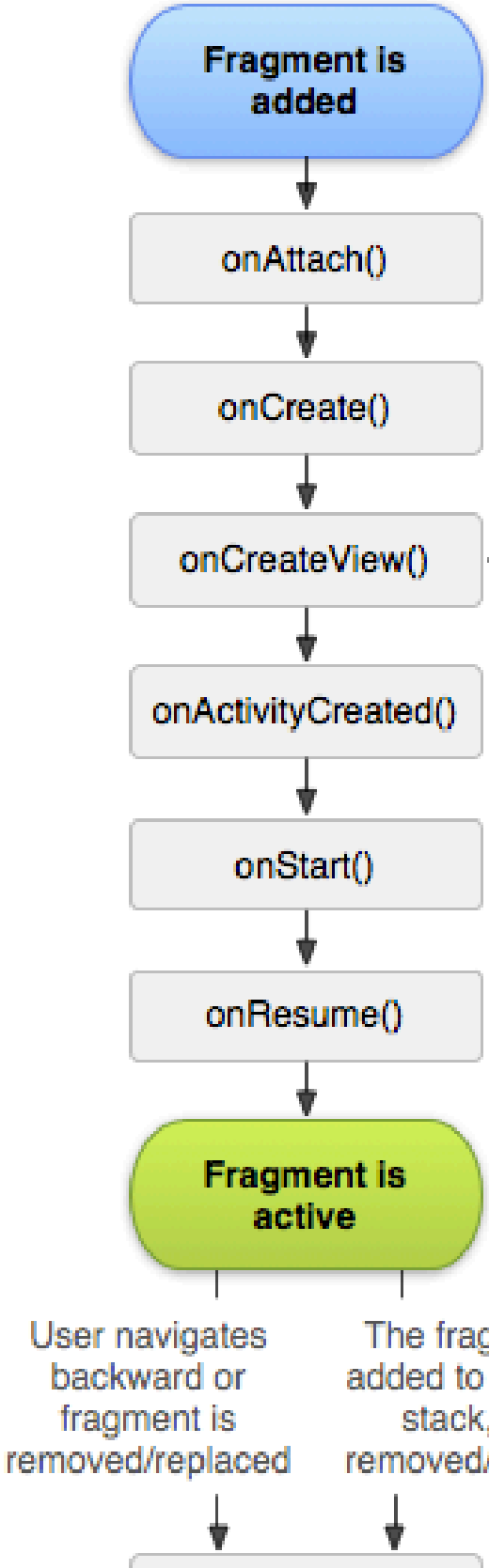
```
ClipboardManager clipboardManager = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);
String name = "AHuier";
clipboardManager.setText(name);
Intent intent = new Intent(IntentDemo.this, Other.class);
startActivity(intent);
```

- 第二个 Activity

```
ClipboardManager clipboardManager = (ClipboardManager) getSystemService(Context.CLIPBOARD_SERVICE);
String msgString = clipboardManager.getText().toString();
textView.setText(msgString);
```

- 利用剪切板传递复杂的数据，如对象，略

2 Fragment 生命周期探讨



- Fragment 基本类，生命周期如下：

```
void onAttach(Context context)
void onCreate(Bundle savedInstanceState)
View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
void onActivityCreated(Bundle savedInstanceState)
void onStart()
void onResume()
void onPause()
void onStop()
void onDestroyView()
void onDestroy()
void onDetach()
```

2.1 首先需要提出的一些 points：

- Fragment 是直接从 Object 继承的，而 Activity 是 Context 的子类。因此我们可以得出结论：Fragment 不是 Activity 的扩展。但是与 Activity 一样，在我们使用 Fragment 的时候我们总会扩展 Fragment(或者是她的子类)，并可以通过子类更改她的行为。
- 使用 Fragment 时，必要构建一个无参构造函数，系统会默认带。但一但写有参构造函数，就必要构建无参构造函数。一般来说我们传参数给 Fragment，会通过 bundle，而不会用构造方法传，代码如下：

```
public static MyFragment newInstance(int index){
    MyFragment mf = new MyFragment();
    Bundle args = new Bundle();
    args.putInt("index",index);
    mf.setArguments(args);
    return mf;
}
```

2.2 生命周期

- onAttach()：onAttach() 回调将在 Fragment 与其 Activity 关联之后调用。需要使用 Activity 的引用或者使用 Activity 作为其他操作的上下文，将在此回调方法中实现。
  - 将 Fragment 附加到 Activity 以后，就无法再次调用 setArguments()--除了在最开始，无法向初始化参数添加内容。
- onCreate(Bundle savedInstanceState)：此时的 Fragment 的 onCreate() 回调时，该 fragmet 还没有获得 Activity 的 onCreate() 已完成的通知，所以不能将依赖于 Activity 视图层次结构存在性的代码放入此回调方法中。在 onCreate() 回调方法中，我们应该尽量避免耗时操作。此时的 bundle 就可以获取到 activity 传来的参数

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        Bundle args = getArguments();
        if (args != null) {
            mLabel = args.getCharSequence("label", mLabel);
        }
    }
```

- onCreateView()

```
onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
```

  - 其中的 Bundle 为状态包与上面的 bundle 不一样。
  - 不要将视图层次结构附加到传入的 ViewGroup 父元素中，该关联会自动完成。如果在此回调中将碎片的视图层次结构附加到父元素，很可能会出现异常。

  - 这句话什么意思呢？就是不要把初始化的 view 视图主动添加到 container 里面，以为这会系统自带，所以 inflate 函数的第三个参数必须填 false，而且不能出现 container.addView(v) 的操作。

```
View v = inflater.inflate(R.layout.hello_world, container, false);
```

- onActivityCreated()
  - onActivityCreated() 回调会在 Activity 完成其 onCreate() 回调之后调用。在调用 onActivityCreated() 之前，Activity 的视图层次结构已经准备好了，这是在用户看到用户界面之前你可对用户界面执行的最后调整的地方。
  - 如果 Activity 和她的 Fragment 是从保存的状态重新创建的，此回调尤其重要，也可以在这里确保此 Activity 的其他所有 Fragment 已经附加到该 Activity 中了
- Fragment 与 Activity 相同生命周期调用：接下来的 onStart(), onResume(), onPause(), onStop() 回调方法将和 Activity 的回调方法进行绑定，也就是说与 Activity 中对应的生命周期相同，因此不做过多介绍。
- onDestroyView(): 该回调方法在视图层次结构与 Fragment 分离之后调用。
- onDestroy()：不再使用 Fragment 时调用。(备注：Fragment 仍然附加到 Activity 并任然可以找到，但是不能执行其他操作)
- onDetach()：Fragment 生命周期最后回调函数，调用后，Fragment 不再与 Activity 绑定，释放资源。

2.3 Fragment 每个生命周期方法的意义、作用

- onAttach()
  - 执行该方法时，Fragment 与 Activity 已经完成绑定，该方法有一个 Activity 类型的参数，代表绑定的 Activity，这时候你可以执行诸如 mActivity = activity 的操作。
- onCreate()
  - 初始化 Fragment。可通过参数 savedInstanceState 获取之前保存的值。
- onCreateView()
  - 初始化 Fragment 的布局。加载布局和 findViewById 的操作通常在此函数内完成，但是不建议执行耗时的操作，比如读取数据库数据列表。
- onActivityCreated()
  - 执行该方法时，与 Fragment 绑定的 Activity 的 onCreate 方法已经执行完成并返回，在该方法内可以进行与 Activity 交互的 UI 操作，所以在该方法之前 Activity 的 onCreate 方法并未执行完成，如果提前进行交互操作，会引发空指针异常。
- onStart()
  - 执行该方法时，Fragment 由不可见变为可见状态。
- onResume()
  - 执行该方法时，Fragment 处于活动状态，用户可与之交互。
- onPause()

  - 执行该方法时，Fragment 处于暂停状态，但依然可见，用户不能与之交互。
- onSaveInstanceState()
  - 保存当前 Fragment 的状态。该方法会自动保存 Fragment 的状态，比如 EditText 键入的文本，即使 Fragment 被回收又重新创建，一样能恢复 EditText 之前键入的文本。
- onStop()
  - 执行该方法时，Fragment 完全不可见。
- onDestroyView()
  - 销毁与 Fragment 有关的视图，但未与 Activity 解除绑定，依然可以通过 onCreateView 方法重新创建视图。通常在 ViewPager+Fragment 的方式下会调用此方法。
- onDestroy()
  - 销毁 Fragment。通常按 Back 键退出或者 Fragment 被回收时调用此方法。
- onDetach()
  - 解除与 Activity 的绑定。在 onDestroy 方法之后调用。
- setUserVisibleHint()
  - 设置 Fragment 可见或者不可见时会调用此方法。在该方法里面可以通过调用 getUserVisibleHint() 获得 Fragment 的状态是可见还是不可见的，如果可见则进行懒加载操作。

2.4 Fragment 生命周期执行流程

- 1、Fragment 创建
  - setUserVisibleHint() —-> onAttach() —-> onCreate() —-> onCreateView() —-> onActivityCreated() —-> onStart() —-> onResume()
- 2、Fragment 变为不可见状态（锁屏、回到桌面、被 Activity 完全覆盖）
  - onPause() —-> onSaveInstanceState() —-> onStop()
- 3、Fragment 变为部分可见状态（打开 Dialog 样式的 Activity）
  - onPause() —-> onSaveInstanceState()
- 4、Fragment 由不可见变为活动状态
  - onStart() —-> OnResume()
- 5、Fragment 由部分可见变为活动状态
  - onResume()
- 6、Fragment 退出
  - onPause() —-> onStop() —-> onDestroyView() —-> onDestroy() —-> onDetach()
  - （注意退出不会调用 onSaveInstanceState 方法，因为是人为退出，没有必要再保存数据）
- 7、Fragment 被回收又重新创建

  - 被回收执行: onPause() —-> onSaveInstanceState() —-> onStop() —-> onDestroyView() —-> onDestroy() —-> onDetach()
  - 重新创建执行: onAttach() —-> onCreate() —-> onCreateView() —-> onActivityCreated() —-> onStart() —-> onResume() —-> setUserVisibleHint()

- 横竖屏切换
  - 与 Fragment 被回收又重新创建一样。

2.5 onHiddenChanged 的回调时机

- 当使用 add()+show()，hide() 跳转新的 Fragment 时，旧的 Fragment 回调 onHiddenChanged()，不会回调 onStop() 等生命周期方法，而新的 Fragment 在创建时是不会回调 onHiddenChanged()，这点要切记。

2.6 FragmentPagerAdapter+ViewPager 的注意事项

- 1、使用 FragmentPagerAdapter+ViewPager 时，切换回上一个 Fragment 页面时（已经初始化完毕），不会回调任何生命周期方法以及 onHiddenChanged()，只有 setUserVisibleHint(boolean isVisibleToUser) 会被回调，所以如果你想进行一些懒加载，需要在这里处理。
- 2、在给 ViewPager 绑定 FragmentPagerAdapter 时，new FragmentPagerAdapter(fragmentManage 的 FragmentManager，一定要保证正确，如果 ViewPager 是 Activity 内的控件，则传递 getSupportFragmentManager()，如果是 Fragment 的控件中，则应该传递 getChildFragmentManager()。只要记住 ViewPager 内的 Fragments 是当前组件的子 Fragment 这个原则即可。
- 3、你不需要考虑在“内存重启”的情况下，去恢复的 Fragments 的问题，因为 FragmentPagerAdapter 已经帮我们处理啦。

2.7 setUserVisibleHint() 不调用的问题

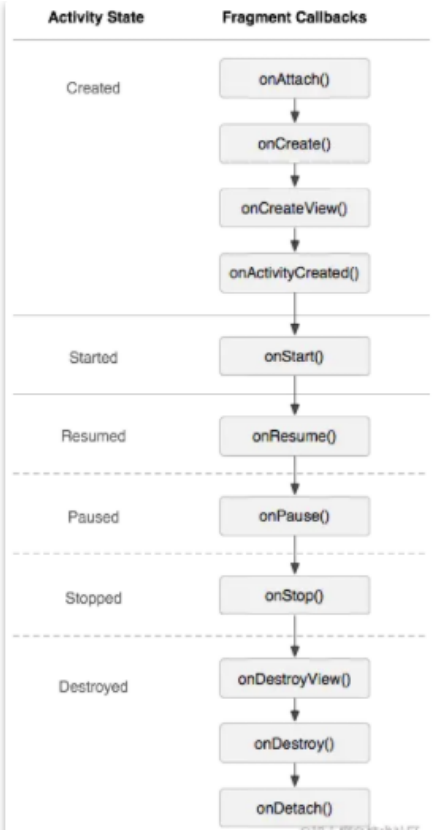
- 通常情况下都是因为 PagerAdapter 不是 FragmentPagerAdapter 造成的，FragmentPagerAdapter 内部实现了对 setUserVisibleHint() 方法的调用，所以需要懒加载的结构最好使用 FragmentPagerAdapter +Fragment 的结构，少用 PagerAdapter。

2.8 详细细节

- 我们这里举个例子来理解 Fragment 生命周期方法。功能如下：共有两个 Fragment：F1 和 F2，F1 在初始化时就加入 Activity，点击 F1 中的按钮调用 replace 替换为 F2。
- 当 F1 在 Activity 的 onCreate() 中被添加时，日志如下：

```
MainActivity: onCreate() BEGIN
MainActivity: onCreate() END

MainActivity: onStart() BEGIN
Fragment1: onAttach() BEGIN
Fragment1: onAttach() END
MainActivity: onAttachFragment() BEGIN
MainActivity: onAttachFragment() END
Fragment1: onCreate() BEGIN
Fragment1: onCreate() END
Fragment1: onCreateView()
Fragment1: onViewCreated() BEGIN
Fragment1: onViewCreated() END
```

```
Fragment1: onActivityCreated() BEGIN
Fragment1: onActivityCreated() END
Fragment1: onStart() BEGIN
Fragment1: onStart() END
MainActivity: onStart() END

MainActivity: onPostCreate() BEGIN
MainActivity: onPostCreate() END
MainActivity: onResume() BEGIN
MainActivity: onResume() END
MainActivity: onPostResume() BEGIN
Fragment1: onResume() BEGIN
Fragment1: onResume() END
MainActivity: onPostResume() END
MainActivity: onAttachedToWindow() BEGIN
MainActivity: onAttachedToWindow() END
```



- 可以看出：
  - Fragment 的 onAttach()->onCreate()->onCreateView()->onActivityCreated()->onStart() 都是在 Activity 的 onStart() 中调用的。
  - Fragment 的 onResume() 在 Activity 的 onResume() 之后调用。
- 接下去分两种情况，分别是不加 addToBackStack() 和加 addToBackStack()。
  - 1、当点击 F1 的按钮，调用 replace() 替换为 F2，且不加 addToBackStack() 时，日志如下：

```
Fragment2: onAttach() BEGIN
Fragment2: onAttach() END
MainActivity: onAttachFragment() BEGIN
MainActivity: onAttachFragment() END
Frag    2: onCreate() BEGIN
```

```
Frag    2: onCreate() END
Fragment1: onPause() BEGIN
Fragment1: onPause() END
Fragment1: onStop() BEGIN
Fragment1: onStop() END
Fragment1: onDestroyView() BEGIN
Fragment1: onDestroyView() END
Fragment1: onDestroy() BEGIN
Fragment1: onDestroy() END
Fragment1: onDetach() BEGIN
Fragment1: onDetach() END
Frag    2: onCreateView()
Frag    2: onViewCreated() BEGIN
Frag    2: onViewCreated() END
Frag    2: onActivityCreated() BEGIN
Frag    2: onActivityCreated() END
Frag    2: onStart() BEGIN
Frag    2: onStart() END
Frag    2: onResume() BEGIN
Frag    2: onResume() END
```

- 可以看到，F1 最后调用了 onDestroy() 和 onDetach()。
- 2、当点击 F1 的按钮，调用 replace() 替换为 F2，且加 addToBackStack() 时，日志如下：

```
Frag    2: onAttach() BEGIN
Frag    2: onAttach() END
MainActivity: onAttachFragment() BEGIN
MainActivity: onAttachFragment() END
Frag    2: onCreate() BEGIN
Frag    2: onCreate() END
Fragment1: onPause() BEGIN
Fragment1: onPause() END
Fragment1: onStop() BEGIN
Fragment1: onStop() END
Fragment1: onDestroyView() BEGIN
Fragment1: onDestroyView() END
Frag    2: onCreateView()
Frag    2: onViewCreated() BEGIN
Frag    2: onViewCreated() END
Frag    2: onActivityCreated() BEGIN
Frag    2: onActivityCreated() END
Frag    2: onStart() BEGIN
Frag    2: onStart() END
Frag    2: onResume() BEGIN
Frag    2: onResume() END
```

- 可以看到，F1 被替换时，最后只调到了 onDestroyView()，并没有调用 onDestroy() 和 onDetach()。当用户点返回按钮回退事务时，F1 会调 onCreateView()->onStart()->onResume()，因此在 Fragment 事务中加不加 addToBackStack() 会影响 Fragment 的生命周期。
- FragmentTransaction 有一些基本方法，下面给出调用这些方法时，Fragment 生命周期的变化：
  - add(): onAttach()->…->onResume()。
  - remove(): onPause()->…->onDetach()。
  - replace(): 相当于旧 Fragment 调用 remove()，新 Fragment 调用 add()。
  - show(): 不调用任何生命周期方法，调用该方法的前提是要显示的 Fragment 已经被添加到容器，只是纯粹把 Fragment UI 的 setVisibility 为 true。
  - hide(): 不调用任何生命周期方法，调用该方法的前提是要显示的 Fragment 已经被添加到容器，只是纯粹把 Fragment UI 的 setVisibility 为 false。
  - detach(): onPause()->onStop()->onDestroyView()。UI 从布局中移除，但是仍然被 FragmentManager 管理。
  - attach(): onCreateView()->onStart()->onResume()。

2.9 Fragment 基础知识

- 核心的类有：
  - Fragment：Fragment 的基类，任何创建的 Fragment 都需要继承该类。
  - FragmentManager：管理和维护 Fragment。他是抽象类，具体的实现类是 FragmentManagerImpl。
  - FragmentTransaction：对 Fragment 的添加、删除等操作都需要通过事务方式进行。他是抽象类，具体的实现类是 BackStackRecord。
- Nested Fragment（Fragment 内部嵌套 Fragment 的能力）是 Android 4.2 提出的，support-fragment 库可以兼容到 1.6。通过 getChildFragmentManager() 能够获得管理子 Fragment 的 FragmentManager，在子 Fragment 中可以通过 getParentFragment() 获得父 Fragment。
- 这里给出 Fragment 最基本的使用方式。首先，创建继承 Fragment 的类，名为 Fragment1：

```
public class Fragment1 extends Fragment {
    private static String ARG_PARAM = "param_key";
    private Activity mActivity; //
    private String mParam;
    public static Fragment1 newInstance(String str) {
        Fragment1 frag = new Fragment1();
        Bundle bundle = new Bundle();
        bundle.putString(ARG_PARAM, str);
        fragment.setArguments(bundle);   //设置参数
        return fragment;
    }
    public void onAttach(Context context) {
        mActivity = (Activity) context;
        mParam = getArguments().getString(ARG_PARAM);  //获取参数
    }
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        View root = inflater.inflate(R.layout.fragment_1, container, false);
        TextView view = root.findViewById(R.id.text);
        view.setText(mParam);
        return root;
    }
}
```

- Fragment 有很多可以复写的方法，其中最常用的就是 onCreateView()，该方法返回 Fragment 的 UI 布局，需要注意的是 inflate() 的第三个参数是 false，因为在 Fragment 内部实现中，会把该布局添加到 container 中，如果设为 true，那么就会重复做两次添加，则会抛如下异常：

```
Caused by: java.lang.IllegalStateException: The specified child already has a parent. You must call removeView() on the chi
```

- 如果在创建 Fragment 时要传入参数，必须要通过 setArguments(Bundle bundle) 方式添加，而不建议通过为 Fragment 添加带参数的构造函数，因为通过 setArguments() 方式添加，在由于内存紧张导致 Fragment 被系统杀掉并恢复（re-instantiate）时能保留这些数据。官方建议如下：

```
It is strongly recommended that subclasses do not have other constructors with parameters, since these constructors will no
```

- 我们可以在 Fragment 的 onAttach() 中通过 getArguments() 获得传进来的参数，并在之后使用这些参数。如果要获取 Activity 对象，不建议调用 getActivity()，而是在 onAttach() 中将 Context 对象强转为 Activity 对象。
- 创建完 Fragment 后，接下来就是把 Fragment 添加到 Activity 中。在 Activity 中添加 Fragment 的方式有两种：
  - 静态添加：在 xml 中通过 <fragment> 的方式添加，缺点是一旦添加就不能在运行时删除。

- 动态添加：运行时添加，这种方式比较灵活，因此建议使用这种方式。
  - 这里只给出动态添加的方式。首先 Activity 需要有一个容器存放 Fragment，一般是 FrameLayout，因此在 Activity 的布局文件中加入 FrameLayout：

```
<FrameLayout
    android:id="@+id/container"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
/>
```

- 然后在 onCreate() 中，通过以下代码将 Fragment 添加进 Activity 中。

```
if (bundle == null) {
    getSupportFragmentManager().beginTransaction()
        .add(R.id.container, Fragment1.newInstance("hello world"), "f1")
        //.addToBackStack("fname")
        .commit();
}
```

- Fragment 有一个常见的问题，即 Fragment 重叠问题，这是由于 Fragment 被系统杀掉，并重新初始化时再次将 fragment 加入 activity，因此通过在外围加 if 语句能判断此时是否是被系统杀掉并重新初始化的情况。

- Fragment 有个常见的异常：

```
java.lang.IllegalStateException: Can not perform this action after onSaveInstanceState
    at android.support.v4.app.FragmentManagerImpl.checkStateLoss(FragmentManager.java:1341)
    at android.support.v4.app.FragmentManagerImpl.enqueueAction(FragmentManager.java:1352)
    at android.support.v4.app.BackStackRecord.commitInternal(BackStackRecord.java:595)
    at android.support.v4.app.BackStackRecord.commit(BackStackRecord.java:574)
```

- 该异常出现的原因是：commit() 在 onSaveInstanceState() 后调用。首先，onSaveInstanceState() 在 onPause() 之后，onStop() 之前调用。onRestoreInstanceState() 在 onStart() 之后，onResume() 之前。

- 因此避免出现该异常的方案有：
  - 不要把 Fragment 事务放在异步线程的回调中，比如不要把 Fragment 事务放在 AsyncTask 的 onPostExecute()，因此 onPostExecute() 可能会在 onSaveInstanceState() 之后执行。
  - 逼不得已时使用 commitAllowingStateLoss()。

2.10 Fragment 注意事项

- 在使用 Fragment 时，我发现了一个金矿，那就是 setRetainInstance() 方法，此方法可以有效地提高系统的运行效率，对流畅性要求较高的应用可以适当采用此方法进行设置。

- Fragment 有一个非常强大的功能--就是可以在 Activity 重新创建时可以不完全销毁 Fragment，以便 Fragment 可以恢复。在 onCreate() 方法中调用 setRetainInstance(true/false) 方法是最佳位置。当 Fragment 恢复时的生命周期如上图所示，注意图中的红色箭头。当在 onCreate() 方法中调用了 setRetainInstance(true) 后，Fragment 恢复时会跳过 onCreate() 和 onDestroy() 方法，因此不能在 onCreate() 中放置一些初始化逻辑.

2.11 回退栈

2.11.1 相关操作下生命周期函数调用顺序

1. 首先点击“ADD”按钮，将 SecondFragment 和 ThirdFragment 动态添加到相应容器

```
Second onAttach()
Second onCreate()
Trd onAttach()
Trd onCreate()
Second onCreateView()
Second onActivityCreated()
Second onStart()
Second onResume()
Trd onCreateView()
Trd onActivityCreated()
Trd onStart()
Trd onResume()
```

2. 然后点击“REMOVE”将 SecondFragment 移除，再点击“REPLACE”按钮将本来加载了 ThirdFragment 的第二个容器替换为 SecondFragment。打开 Logcat 日志可以看到：

```
Second onAttach()
Second onCreate()
Trd onPause()
Trd onStop()
Trd onDestroyView()
Trd onDestroy()
Trd onDetach()
Second onCreateView()
Second onActivityCreated()
Second onStart()
Second onResume()
```

2.11.2 基础原理：这里是否会涉及两套 fragment 的不同？(应该不会，可以证实一下)

- 如果没有加入回退栈，则用户点击返回按钮会直接将 Activity 出栈；如果加入了回退栈，则用户点击返回按钮会回滚 Fragment 事务。
- 默认情况下，Fragment 事务是不会加入回退栈的，如果想将 Fragment 加入回退栈并实现事物回滚，首先需要在 commit() 方法之前调用事务的以下方法将其添加到回退栈中：

```
addToBackStack(String tag) // 标记本次的回滚操作
```

- 在 Fragment 回退时，默认调用 FragmentManager 的 popBackStack() 方法将最上层的操作弹出回退栈。当栈中有多层时，我们可以根据 id 或 TAG 标识来指定弹出到的操作所在层。

```
popBackStack(int id, int flags)      // 其中 id 表示提交变更时 commit() 的返回值。
popBackStack(String name, int flags) // 其中 name 是 addToBackStack(String tag) 中的 tag 值。
```

- 在上面 2 个方法里面，都用到了 flags，其实 flags 有两个取值：0 或 FragmentManager.POP_BACK$\backslash$_STACK$\backslash$_INCLUSIVE。
  - 当取值 0 时，表示除了参数指定这一层之上的所有层都退出栈，指定的这一层为栈顶层；
  - 当取值 POP_BACK_STACK_INCLUSIVE 时，表示连着参数指定的这一层一起退出栈
- 根据之前 fragment 界面的 tag，返回到那个 fragment 界面并关闭当前 Fragment 和要返回 fragment 界面之间的 fragment，包括关闭当前的 fragment

```
/***
 * 返回到指定已经打开的 fragment，并关闭当前及当前与指定 Fragment 间的所有 Fragment
 * @param tag 需要返回到指定 Fragment 的 tag
 *            若你需要返回到 FragmentOne，且 FragmentOne 的 tag 为“one”，则此处 tag 参数填“one”
 * @param context
 */
public void goBackToFragmentByTag(String tag,Context context)
```

- 关闭所有 Fragment

```
public void finishAllFragments(Context context)
```

- 获取回退栈中 fragment 个数

```
public int getFragmentSize(Context context)
```

- 获取回退栈中所有 Fragment 对应的 tag 的集合

```
public List<String> getFragmentTags(Context context)
```

- 如果想要了解回退栈中 Fragment 的情况，可以通过以下 2 个方法来实现：

```
getBackStackEntryCount()       // 获取回退栈中 Fragment 的个数。
getBackStackEntryAt(int index) // 获取回退栈中该索引值下的 Fragment。
```

- 使用 popBackStack() 来弹出栈内容的话，调用该方法后会将事物操作插入到 FragmentManager 的操作队列，只有当轮询到该事物时才能执行。如果想立即执行事物的话，可以使用下面这几个方法：

```
popBackStackImmediate()
popBackStackImmediate(String tag)
popBackStackImmediate(String tag, int flag)
popBackStackImmediate(int id, int flag)
```

- 这里对 popBackStackImmediate 方法参数做一个解释：方法中第二个参数 flag 只能是 0 或者 1(POP_BACK_STACK_INCLUSIVE)
  - 当第一个参数为 null，第二个参数为 0 时，会弹出栈顶的一个 fragment
  - 当第一个参数为 null，第二个参数为 1 (POP_BACK_STACK_INCLUSIVE) 时，会清空回退栈中所有 fragment
  - 当第一个参数为 fragment 的 tag，第二个参数为 1(POP_BACK_STACK_INCLUSIVE) 时，会弹出该状态（包括该状态）以上的所有状态
- 出栈时只是将栈顶的 Fragment 移出了数组，并没有将其销毁。所以当它再次入栈时，便能恢复之前的数据。

```
显示新页面时 hide 旧页面
A 到 B
    A 生命周期无变化
 B(  onAttach()  --> onViewStateRestored( )--> onResume() )
B 返回 A
    A 生命周期无变化
    B(  onPause()  --> onDetach() )
A 再到 B
    A 生命周期无变化
    B(  onAttach()  --> onViewStateRestored( )--> onResume() )
如果在 B 页面输入了数据, B 返回 A 再到 B, B 中数据会恢复

显示新页面时 remove 旧页面
A 到 B
    A(  onPause()  --> onDestroyView() )
    B(  onAttach()  --> onViewStateRestored( )--> onResume() )
B 返回 A
    A(  onCreateView()  --> onViewStateRestored( )--> onResume() )
     B(  onPause()  --> onDetach() )
A 再到 B
    A(  onPause()  --> onDestroyView() )
    B(  onAttach()  --> onViewStateRestored( )--> onResume() )
如果在 B 页面输入了数据, B 返回 A 再到 B, B 中数据也会恢复
```

- androidX 一文读懂 Fragment 的方方面面
  - https://juejin.cn/post/7006970844542926855 androidX 很值得一读

3 Android Fragment 之间数据传递

3.1 获得对方的引用强转：这种方式不推荐

- 通过 findFragmentByTag 或者 getActivity 获得对方的引用（强转）之后，再相互调用对方的 public 方法，但是这样做一是引入了“强转”的丑陋代码，另外两个类之间各自持有对方的强引用，耦合较大，容易造成内存泄漏。

```
// 宿主 activity 中的 getTitles() 方法
public String getTitles(){
    return "hello";
}
// Fragment 中的 onAttach 方法
@Override
public void onAttach(Activity activity) {
    super.onAttach(activity);
    titles = ((MainActivity) activity).getTitles(); // 通过强转成宿主 activity，就可以获取到传递过来的数据
}
```

3.2 通过 Bundle 的方法进行传值，例如以下代码：

```
// 从 Activity 中对 fragment 设置一些参数
MyFragment myFragment = new MyFragment();
Bundle bundle = new Bundle();
bundle.putString("DATA",values); // 这里的 values 就是我们要传的值
myFragment.setArguments(bundle);

// 在 fragment 中，通过 getArguments 获得 Activity 中传来的 bundle
Bundle arguments = getArguments();
```

- 然后在 Fragment 中的 onCreatView() 方法中，通过 getArgments() 方法，获取到 bundle 对象，然后通过 getString 的 key 值拿到我们传递过来的值。

3.3 利用 eventbus 进行通信，这种方法实时性高，而且 Activity 与 Fragment 之间可以完全解耦。

```
// Activity 中的代码
EventBus.getDefault().post(" 消息");
// Fragment 中的代码
EventBus.getDefault().register(this);
@Subscribe
public void test(String text) {
    tv_test.setText(text);
}
```

3.4 其它自定义（? 这里讲得不透彻，寻找更透彻的解释）

- 如果我们不需要传递数值，那就直接可以在宿主 activity 中，跟平常一样创建 fragment，但是如果我们需要传递数据的话，可以使用 newInstance（数据）方法来传递，这个方法是自己定义的，但是是定义在 Fragment 中的一个静态方法。

```
static MyFragment newInstance(String s){
    MyFragment myFragment = new MyFragment();
    Bundle bundle = new Bundle();
    bundle.putString("DATA",s);
    myFragment.setArguments(bundle);
    return myFragment;
}
@Nullable @Override // 同样，在 onCreatView 中直接获取这个值
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
    View view = inflater.inflate(R.layout.layout_fragment,container,false);
    Bundle bundle = getArguments();
    String data = bundle.getString("DATA");
    tv = (TextView) view.findViewById(R.id.id_fm_tv);
    if (data != null)
```

```
        tv.setText(data);
    return view;
}
```

- 在宿主 activity 中，创建 Fragment

```
FragmentTransaction fragmentTransaction = fragmentManager.beginTransaction();
fragmentTransaction.setCustomAnimations(android.R.anim.fade_in,android.R.anim.fade_out);
fragment1 = MyFragment.newInstance(" 这是第一个 fragment");//这里只需要直接调用这个方法，就创建了一个 fragment
fragment2 = MyFragment.newInstance(" 这是第二个 fragment");
fragment3 = MyFragment.newInstance(" 这是第三个 fragment");
```

3.4.1 Fragment 向 Activity 传递数据

- 首先，在 Fragment 中定义接口，并让 Activity 实现该接口（具体实现省略）：

```
public interface OnFragmentInteractionListener {
    void onItemClick(String str);  //将 str 从 Fragment 传递给 Activity
}
```

- 在 Fragment 的 onAttach() 中，将参数 Context 强转为 OnFragmentInteractionListener 对象：

```
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener) context;
    } else {
        throw new RuntimeException(context.toString()
                + " must implement OnFragmentInteractionListener");
    }
}
```

- 并在 Fragment 合适的地方调用 mListener.onItemClick("hello") 将” hello” 从 Fragment 传递给 Activity。

1. FABridge

    - https://github.com/hongyangAndroid/FABridge
    - 由于通过接口的方式从 Fragment 向 Activity 进行数据传递比较麻烦，需要在 Fragment 中定义 interface，并让 Activity 实现该 interface，FABridge 通过注解的形式免去了这些定义。
    - 在 build.gradle 中添加依赖：

    ```
    annotationProcessor 'com.zhy.fabridge:fabridge-compiler:1.0.0'
    compile 'com.zhy.fabridge:fabridge-api:1.0.0'
    ```

    - 首先定义方法 ID，这里为 FAB_ITEM_CLICK，接着在 Activity 中定义接口：

    ```
    @FCallbackId(id = FAB_ITEM_CLICK)
    public void onItemClick(String str) {  //方法名任意
        Toast.makeText(this, str, Toast.LENGTH_SHORT).show();
    }
    ```

    - 最后，在 Fragment 中，通过以下形式调用” ID=FAB_ITEM_CLICK” 的方法（该方法可能在 Activity 中，也可能在任何类中）：

    ```
    Fabridge.call(mActivity,FAB_ITEM_CLICK,"data");  //调用 ID 对应的方法，"data" 为参数值
    ```

    - 后半部分,关于 FragmentViewPager https://github.com/tyzlmjj/PagerBottomTabStrip.git
    - https://cloud.tencent.com/developer/article/1005540
    - http://www.cnblogs.com/purediy/p/3276545.html
    - Fragment 与 Activity 生命周期对比图：

3.5 生命周期分析

3.5.1 当一个 fragment 被创建的时候 (它会经历以下状态)

- onAttach()
- onCreate()
- onCreateView()
- onActivityCreated()

3.5.2 当这个 fragment 对用户可见的时候

- onStart()
- onResume()

3.5.3 当这个 fragment 进入“后台模式”的时候

- onPause()
- onStop()

3.5.4 当这个 fragment 被销毁了（或者持有它的 activity 被销毁了）

- onPause()
- onStop()
- onDestroyView()
- onDestroy() // 本来漏掉类这个回调，感谢 xiangxue336 提出。
- onDetach()

3.5.5 就像 activity 一样，在以下的状态中，可以使用 Bundle 对象保存一个 fragment 的对象。

- onCreate()
- onCreateView()
- onActivityCreated()

3.5.6 fragments 的大部分状态都和 activity 很相似，但 fragment 有一些新的状态。

- onAttached() – 当 fragment 被加入到 activity 时调用（在这个方法中可以获得所在的 activity）。
- onCreateView() – 当 activity 要得到 fragment 的 layout 时，调用此方法，fragment 在其中创建自己的 layout(界面)。
- onActivityCreated() – 当 activity 的 onCreated() 方法返回后调用此方法
- onDestroyView() – 当 fragment 中的视图被移除的时候，调用这个方法。
- onDetach() – 当 fragment 和 activity 分离的时候，调用这个方法。

- Notes:
  - 一旦 activity 进入 resumed 状态（也就是 running 状态），你就可以自由地添加和删除 fragment 了。
  - 因此，只有当 activity 在 resumed 状态时，fragment 的生命周期才能独立的运转，其它时候是依赖于 activity 的生命周期变化的。

3.6 Dialog

- 主要应用于一些临时的对话框，比如向用户询问是否允许开启一些权限 AlertDialog，让用户选择一个时间 TimePickerDialog，或者自定义界面进行选择 DialogFragment。初始化一个简单的 Dialog 的语法是：

```
// 使用 Builder class 来定义 AlertDialog 的属性
val builder = AlertDialog.Builder(this)
builder.setMessage(R.string.your_dialog_message)
    .setPositiveButton(
        R.string.ok,
        DialogInterface.OnClickListener { dialog, id ->
            // 定义用户按下 OK 按钮后的行为
        })
    .setNegativeButton(
        R.string.cancel,
        DialogInterface.OnClickListener { dialog, id ->
            // 定义用户按下 CANCEL 按钮后的行为
        })
// 创建一个 AlertDIalog 实例
builder.create()
```

- 上面的代码会创建出一个带有两个按钮的对话框，并且根据用户的选择来运行相对应的逻辑。

3.7 DialogFragment

- DialogFragment 是 Android 3.0 提出的，代替了 Dialog，用于实现对话框。他的优点是：即使旋转屏幕，也能保留对话框状态。
- 如果要自定义对话框样式，只需要继承 DialogFragment，并重写 onCreateView()，该方法返回对话框 UI。这里我们举个例子，实现进度条样式的圆角对话框。

```
public class ProgressDialogFragment extends DialogFragment {
    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
        getDialog().requestWindowFeature(Window.FEATURE_NO_TITLE); //消除 Title 区域
        getDialog().getWindow().setBackgroundDrawable(new ColorDrawable(Color.TRANSPARENT));  //将背景变为透明
        setCancelable(false);  //点击外部不可取消
        View root = inflater.inflate(R.layout.fragment_progress_dialog, container);
        return root;
    }

    public static ProgressDialogFragment newInstance() {
        return new ProgressDialogFragment();
    }
}
```

- 进度条动画我们使用 Lottie 实现，Lottie 动画从这里找到。使用非常方便，只需要下载 JSON 动画文件，然后在 XML 中写入：

```
<com.airbnb.lottie.LottieAnimationView
    android:layout_width="wrap_content"    // 大小根据 JSON 文件确定
    android:layout_height="wrap_content"
    app:lottie_fileName="loader_ring.json"  /JSON 文件
    app:lottie_loop="true"          // 循环播放
    app:lottie_autoPlay="true" />  // 自动播放
```

然后通过下面代码显示对话框：

```
ProgressDialogFragment fragment = ProgressDialogFragment.newInstance();
fragment.show(getSupportFragmentManager(), "tag");
//fragment.dismiss();
```

- 为了实现圆角，除了在 onCreateView() 中把背景设为透明，还需要对 UI 加入背景：

```
<shape xmlns:android="http://schemas.android.com/apk/res/android">
    <solid android:color="#ffffff"/>
    <corners
        android:radius="20dp"/>
</shape>
```

4 启动模式与任务栈

- 安卓有四种启动模式：standard、singleTop、singleTask 和 singleInstance，想要更改模式可以在 AndroidManifest.xml 中 activity 标签下添加 launchMode 标签。下面是各种模式的详细介绍 (下文中所有的栈，均指任务栈)：
  - Activity 所需的任务栈?
    - 这要从一个参数说起：TaskAffinity(任务相关性)。这个参数标识了一个 activity 所需的任务栈的名字，默认情况下，所有 activity 所需的任务栈的名字为应用的包名。
    - TaskAffinity 参数标识着 Activity 所需要的任务栈的名称，默认情况下，一个应用中所有 Activity 所需要的任务栈名称都为该应用的包名。
    - TaskAffinity 属性一般跟 singleTask 模式或者跟 allowTaskReparenting 属性结合使用，在其他情况下没有实际意义。

4.1 standard：标准模式

- 这也是系统的默认模式。每次启动一个 Activity 都会重新创建一个新的实例，不管实例是否已经存在。被创建的实例的生命周期符合典型的 Activity 的生命周期。在这种模式下，谁启动了这个 Activity，那么这个 Activity 就运行在启动它的那个 Activity 所在的栈中。比如 ActivityA 启动了 ActivityB（B 也是 standard 模式），那么 B 就会进入到 A 所在的栈中。
  - 如果我们用 ApplicationContext 去启动 standard 模式的时候 Activity 的时候会报错，错误的原因是因为 standard 模式的 Activity 默认会进入启动它的 Activity 所属的任务栈中，但是由于非 Activity 类型的 Context（如 ApplicationContext）并没有所谓的任务栈，所有这就有问题了。解决这个问题的方法是为待启动 Activity 指定 FLAG_ACTIVITY_NEW_TASK 标记位，这样启动的时候就会为它创建一个新的任务栈，这时候待启动 Activity 实际上是一 singleTask 模式启动的。后文会再次强调一下

4.2 singleTop：栈顶复用模式

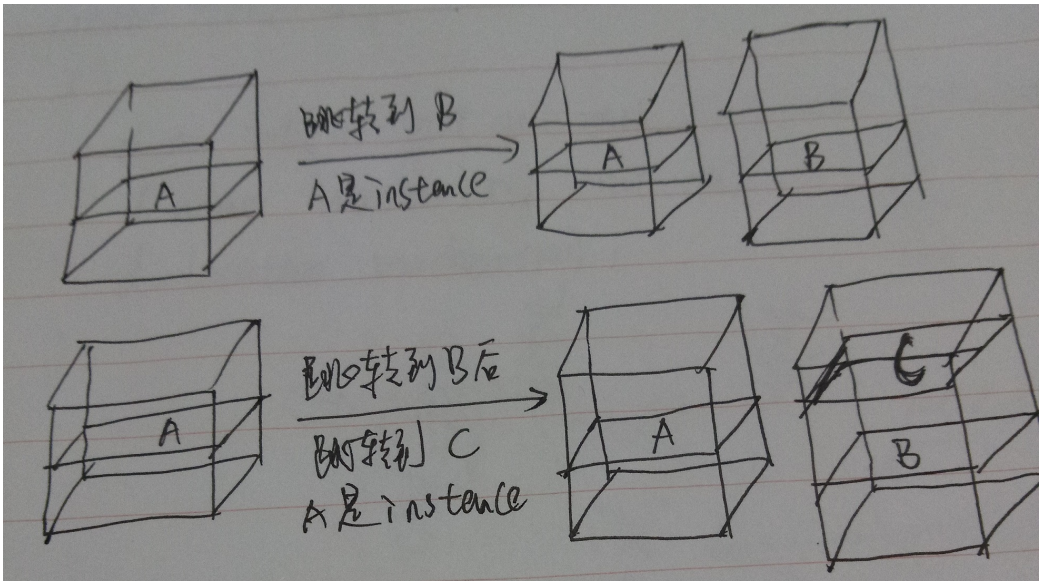
- 在这种模式下，如果新 Activity 已经位于任务栈的栈顶（处于完全可见状态），那么此 Activity 不会被重新创建。但是如果新的 Activity 不是位于栈顶（处于不完全可见状态），那么新 Activity 仍然会重新创建。在 singleTop（栈顶复用模式）下，如果 Activity 位于栈顶，我们再次启动该方法，那么该方法会回调 onNewIntent() 方法 (不会创建新的 instance)，而 onCreate、onStart 方法不会被调用。
- 也允许出现重复的活动实例，分为两种情况，页面不在栈顶和页面已经在栈顶的情况。
  - （1）页面不在栈顶。此时还是和 standard 一样重新生成一个新的 activity 实例。
  - （2）页面在栈顶。此时直接使用栈顶的这个活动实例，不会重新生成新的。

4.3 singleTask：栈内复用模式

- 这是一种单实例模式，在这种模式下，一个 Activity 在一个栈中只能有一个实例，类似单例模式。详细讲解，当一个具有 singleTask 模式的 Activity 请求启动后，例如 Activity A，系统首先会寻找是否存在 A 想要的任务栈，如果不存在，就重新创建一个新的任务栈，然后创建 A 的实例后把 A 放到栈中。如果存在 A 所需要的任务栈，再看 Activity A 是否在栈中有实例存在，如果有实例存在，那么系统就会把 A 调到栈顶并调用 onNewIntent 方法，如果实例不存在，就创建 A 的实例并把 A 压入栈中。
  - 小例子：活动启动的顺序是 ABCDB，第一次启动 B 的时候栈内没有实例，系统会创建新的 B 的实例，但第二次启动 B 的时候，系统查询任务栈，发现有实例，系统就会把活动 B 移到栈顶，并且把 B 之上的所有活动出栈。
- 不会出现重复的活动实例，这种模式下至于两种情形，栈内有这个活动的实例的情形和栈内没有这个实例的情形。
  - （1）假设站内原来有 A、B 两个实例，此时跳转到 A 页面，不管 A 页面位不位于栈顶，只要栈内存在 A 活动的实例，那么就把 A 以上的实例全部销毁出栈，总之让 A 位于栈顶得到用户观看。
  - （2）假设栈内没有 C，此时跳转到 C 页面，会创建新的 C 活动实例。

4.4 singleInstance：单实例模式

- 这是一种加强的 singleTask 模式，它除了具有 singleTask 模式所具有的所有特性外，还加强一点，那就是具有此种模式的 Activity 只能单独的位于一个任务栈中 (具有独占性：独占一个任务栈)。简单而言，如果一个活动是 singleInstance 的启动模式，那么该活动就只能单独位于一个栈中。
  - 小例子：如果一个活动是 singleInstance 模式，那么活动 C 会单独创建一个新的任务栈，而返回栈（这里返回栈没有读懂，是什么意思呢???）中，活动 C 处于的任务栈会先压入返回栈的栈低，再把另外一个活动栈放入返回栈中。
- 不会出现重复的活动实例，此时比较特殊，持有这种模式的活动实例单独有一个栈来存储它，栈内只有它一个实例，如果多个应用启动这个活动，那么他们共同享用这个栈内的唯一实例，不会生成新的。这个模式的应用比如说手机的锁屏页面。



- 这种模式的返回模式，出栈顺序是 C-B-A, 入栈顺序是 A-B-C, 最先出现，最后死亡

4.5 综合比较与汇总

launchMode	简述	详解	主要用途
standard	标准模式、默认模式 android:launchMode =”standard” 或不指定属性	Standard 是默认启动模式，没有指定 android:launchMode 属性时默认是 standard。在启动 activity 时不管栈中是何种状态，都会新生成一个 activity 压入栈中，可以存在多个连续相同的 activity。	最常用的启动模式，适用于没有特殊要求的启动模式。
singleTop	栈顶复用 android:launchMode =”singleTop”	singleTop 模式下启动 activity 时会匹配当前栈顶 activity 与要启动的 activity 是否相同，相同则调用 onNewIntent()方法。如果不同则会 new 一个 activity 并压入栈中。	适合用于接受通知推送启动的界面，如新闻客户端收到 7 条新闻推送，则尽可能复用 activity 去显示推送，如果打开 7 个 activity 是很耗内存和低性能的方法。
singleTask	栈内复用 android:launchMode =”singleTask”	singleTask 模式会确认当前任务栈中是否存在即将新建 activity 的实例，如果存在，则调用其 onNewIntent()方法，并调用 clearTop 方法清理其上面的 activity。如果不存在则会新建一个 task 栈，并把新 new 出的 activity 压入新生成的任务栈中	适合用作程序主入口，比如浏览器主界面，不管多少个应用启动浏览器都只会启动一个主界面，其余情况都使用 onNewIntent，并清空其上的其它界面。
singleInstance	单实例模式 android:launchMode =”singleInstance”	singleInstance 模式不管任务栈中是否存在该 activity，都会新建一个任务栈存放 activity。每个任务栈中只有一个 activity，每个 activity 占用一个任务栈。	在需要系统中只存在一个该 activity 实例时使用，可以保证 activity 的全局唯一性。

4.6 设定方法

- 怎么设定这四种模式，有两种方法，

4.6.1 manifest.xml 文件中设置。

```
<activity android:name=".Activity1"
          android:launchMode="standard"
          android:label="@string/app_name">
  <intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
  </intent-filter>
</activity>
```

4.6.2 Intent 设置标记位

- Intent 设置标记位方式的优先级高于 manifest 中指定 launchMode 的方式.

1. FLAG_ACTIVITY_NEW_TASK:
    - 效果和在 manifest 中设置 launchMode 为 singleTask 相同。
    - 该标志位表示使用一个新的 Task 来启动一个 Activity，相当于在清单文件中给 Activity 指定“singleTask”启动模式。通常我们在 Service 启动 Activity 时，由于 Service 中并没有 Activity 任务栈，所以必须使用该 Flag 来创建一个新的 Task。
2. FLAG_ACTIVITY_SINGLE_TOP:

- 这个 FLAG 就相当于加载模式中的 singletop，比如说原来栈中情况是 A,B,C,D 在 D 中启动 D，栈中的情况还是 A,B,C,D

3. FLAG_ACTIVITY_CLEAR_TOP：
   - 具有此标记的 activity 启动时，在同一任务栈中所有位于它上面的 activity 都要出栈。一般和 FLAG_ACTIVITY_NEW_TASK 配合使用。这种情况下，被启动的 activity 的实例如果已经存在，那么会调用它的 onNewIntent 方法。
   - 这个 FLAG 就相当于加载模式中的 SingleTask，这种 FLAG 启动的 Activity 会把要启动的 Activity 之上的 Activity 全部弹出栈空间。类如：原来栈中的情况是 A,B,C,D 这个时候从 D 中跳转到 B，这个时候栈中的情况就是 A,B 了.
4. FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS：
   - 具有此标记的 activity 不会出现在历史 activity 列表中。
   - 使用该标识位启动的 Activity 不添加到最近应用列表，也即我们从最近应用里面查看不到我们启动的这个 activity。
   - 等同于在 manifest 中设置 activity 属性

```
<android:excludeFromRecents="true"/>
```

5. Intent.FLAG_ACTIVITY_NO_HISTORY -使用该模式来启动 Activity，当该 Activity 启动其他 Activity 后，该 Activity 就被销毁了，不会保留在任务栈中。如 A-B,B 中以这种模式启动 C，C 再启动 D，则任务栈只有 ABD。
6. FLAG_ACTIVITY_CLEAR_TASK
   - 如果在传递给 Context.startActivity() 的意图中设置了该标志，则会导致在启动 activity 之前清除与该 activity 关联的任何现有任务。也就是说，activity 成为一个空任务的新根，任何旧 activity 都 finish 了。
   - 这只能与 FLAG_ACTIVITY_NEW_TASK 一起使用。
7. 注意事项：
   - 当通过非 activity 的 context 来启动一个 activity 时，需要增加 intent flag FLAG_ACTIVITY_NEW

```
Intent i = new Intent(this, Wakeup.class);
i.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
```

   (a) 对 Intent.FLAG_ACTIVITY_NEW_TASK 这个属性，是不是一定新开一个栈？
   - 这个问题的答案是：不一定
   - 假设现在有一个栈 1，里面是 A,B,C。此时，在 C 中启动 D 的时候，设置 FLAG_ACTIVITY_NEW 标记，此时会有两种情况：
     - 1. 如果 D 这个 Activity 在 Manifest.xml 中的声明中添加了 Task Affinity，系统首先会查找有没有和 D 的 Task Affinity 相同的 Task 栈存在，如果有存在，将 D 压入那个栈
     - 2. 如果 D 这个 Activity 在 Manifest.xml 中的 Task Affinity 默认没有设置，则会把其压入栈 1，变成：A B C D，这样就和标准模式效果是一样的了。
   - 我想，这篇里的部分结论，很大一部分结论，还是需要小项目代码再验证一下其正确性的 不敢轻信、没有确信!!!
     - 也就是说，设置了这个标志后，新启动的 Activity 并非就一定在新的 Task 中创建，如果 A 和 B 在属于同一个 package，而且都是使用默认的 Task Affinity，那 B 还是会在 A 的 task 中被创建。所以，只有 A 和 B 的 Task Affinity 不同时，设置了这个标志才会使 B 被创建到新的 Task。

- ！注意如果试图从非 Activity 的非正常途径启动一个 Activity，比如从一个 Receiver 中启动一个 Activity，则 Intent 必须要添加 FLAG_ACTIVITY_NEW_TASK 标记。
- 我们这里之所以会新建一个栈，因为我们的 APP 和系统 Activity 的 Task Affinity 不同

4.6.3 Intent flag 标记位进阶：（这个难度比较高一点儿，改天脑袋清醒的时候再好好理解消化一下）

- https://blog.csdn.net/vshuang/article/details/66472338?spm=1001.2101.3001.6661.1&utm_medium=distribute.pc_relevant_t0.none-task-blog-2%7Edefault%7ECTRLIST%7Edefault-1.no_search_link&depth_1-utm_source=distribute.pc_relevant_t0.none-task-7Edefault%7ECTRLIST%7Edefault-1.no_search_link

1. FLAG_ACTIVITY_CLEAR_TOP　　如果设置，并且这个 Activity 已经在当前的 Task 中运行，因此，不再是重新启动一个这个 Activity 的实例，而是在这个 Activity 上方的所有 Activity 都将关闭，然后这个 Intent 会作为一个新的 Intent 投递到老的 Activity（现在位于顶端）中。例如，假设一个 Task 中包含这些 Activity：A，B，C，D。如果 D 调用了 startActivity()，并且包含一个指向 Activity B 的 Intent，那么，C 和 D 都将结束，然后 B 接收到这个 Intent，因此，目前 stack 的状况是：A，B。上例中正在运行的 Activity B 既可以在 onNewIntent() 中接收到这个新的 Intent，也可以把自己关闭然后重新启动来接收这个 Intent。如果它的启动模式声明为“multiple”(默认值)，并且你没有在这个 Intent 中设置 FLAG_ACTIVITY_SINGLE_TOP 标志，那么它将关闭然后重新创建；对于其它的启动模式，或者在这个 Intent 中设置 FLAG_ACTIVITY_SINGLE_TOP 标志，都将把这个 Intent 投递到当前这个实例的 onNewIntent() 中。这个启动模式还可以与 FLAG_ACTIVITY_NEW_TASK 结合起来使用：用于启动一个 Task 中的根 Activity，它会把那个 Task 中任何运行的实例带入前台，然后清除它直到根 Activity。这非常有用，例如，当从 Notification Manager 处启动一个 Activity

2. FLAG_ACTIVITY_CLEAR_WHEN_TASK_RESET　　如果设置，这将在 Task 的 Activity stack 中设置一个还原点，当 Task 恢复时，需要清理 Activity。也就是说，下一次 Task 带着 FLAG_ACTIVITY_RESET_TASK_IF_NEEDED 标记进入前台时（典型的操作是用户在主画面重启它），这个 Activity 和它之上的都将关闭，以至于用户不能再返回到它们，但是可以回到之前的 Activity。这在你的程序有分割点的时候很有用。例如，一个 e-mail 应用程序可能有一个操作是查看一个附件，需要启动图片浏览 Activity 来显示。这个 Activity 应该作为 e-mail 应用程序 Task 的一部分，因为这是用户在这个 Task 中触发的操作。然而，当用户离开这个 Task，然后从主画面选择 e-mail app，我们可能希望回到查看的会话中，但不是查看图片附件，因为这让人困惑。通过在启动图片浏览时设定这个标志，浏览及其它启动的 Activity 在下次用户返回到 mail 程序时都将全部清除。

3. FLAG_ACTIVITY_RESET_TASK_IF_NEEDED　　If set, and this activity is either being started in a new task or bringing to the top an existing task, then it will be launched as the front door of the task. This will result in the application of any affinities needed to have that task in the proper state (either moving activities to or from it), or simply resetting that task to its initial state if needed.

4. FLAG_ACTIVITY_NEW_TASK　　如果设置，这个 Activity 会成为历史 stack 中一个新 Task 的开始。一个 Task（从启动它的 Activity 到下一个 Task 中的 Activity）定义了用户可以迁移的 Activity 原子组。Task 可以移动到前台和后台；在某个特定 Task 中的所有 Activity 总是保持相同的次序。这个标志一般用于呈现“启动”类型的行为：它们提供用户一系列可以单独完成的事情，与启动它们的 Activity 完全无关。使用这个标志，如果正在启动的 Activity 的 Task 已经在运行的话，那么，新的 Activity 将不会启动；代替的，当前 Task 会简单的移入前

台。参考 FLAG_ACTIVITY_MULTIPLE_TASK 标志，可以禁用这一行为。这个标志不能用于调用方对已经启动的 Activity 请求结果。

5. FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS

   - 如果设置，新的 Activity 不会在最近启动的 Activity 的列表中保存。
   - 参考一个 stackoverflow 的问答 https://stackoverflow.com/questions/7759556/flag-activity-exclude-from-recents-excludes-whole-application-not-only-the-ac
   - I have a Notification which starts an Activity. After a long press on home button and selecting my app, I want to start my main Activity again, and not this Activity started by the Notification. I tried with FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS, but this removed my whole application from the recents, and that's not what I want to achieve. How can I have my app in the recents, but have the main Activity started?
   - Okay, I found the solution to my problem. I started an Activity from a Notification with FLAG_ACTIVITY_NEW_TASK. But it seems to me that this Activity only gets started in an own task if affinity is different from the default affinity. So I had to add a different affinity in the manifest.
   - And it seems that FLAG_ACTIVITY_EXCLUDE_FROM_RECENTS does not (as documented) exlucde the Activity from the recents, rather it excludes the whole task (not the whole application) in which the Activity gets started from the recents. And as I hadn't set a different affinity the Activity which I wanted to exclude was started in the same task (although I had set FLAG_ACTIVITY_NEW_TASK) and so my whole application (as it was running in only one task) was excluded from the recents.
   - Now I've set a different affinity for the Activity that gets started from the Notification and I start it with FLAG_ACTIVITY_NEW_TASK | FLAG_ACTIVITY_EXCLUDE_FROM_RECENT When I leave this Activity and long-press the HOME button I can choose my app and the default task is started or brought to the front.

6. FLAG_ACTIVITY_FORWARD_RESULT

   - 如果设置，并且这个 Intent 用于从一个存在的 Activity 启动一个新的 Activity，那么，这个作为答复目标的 Activity 将会传到这个新的 Activity 中。这种方式下，新的 Activity 可以调用 setResult(int)，并且这个结果值将发送给那个作为答复目标的 Activity。

4.7 启动模式与 startActivityForResult

4.7.1 LaunchMode 与 StartActivityForResult

- 我们在开发过程中经常会用到 StartActivityForResult 方法启动一个 Activity，然后在 onActivityResult() 方法中可以接收到上个页面的回传值，但你有可能遇到过拿不到返回值的情况，那有可能是因为 Activity 的 LaunchMode 设置为了 singleTask。5.0 之后，android 的 LaunchMode 与 StartActivityForResult 的关系发生了一些改变。两个 Activity，A 和 B，现在由 A 页面跳转到 B 页面，看一下 LaunchMode 与 StartActivityForResult 之间的关系：
- 这是为什么呢?
- 这是因为 ActivityStackSupervisor 类中的 startActivityUncheckedLocked 方法在 5.0 中进行了修改。
- 在 5.0 之前，当启动一个 Activity 时，系统将首先检查 Activity 的 launchMode，如果为 A 页面设置为 SingleInstance 或者 B 页面设置为 singleTask 或者 singleInstance，则会在 LaunchFlags 中加入 FLAG_ACTIVITY_NEW_TASK 标志，而如果含有 FLAG_ACTIVITY_NEW_TASK 标志的话，onActivityResult 将会立即接收到一个 cancle 的信息。

```
final boolean launchSingleTop = r.launchMode == ActivityInfo.LAUNCH_SINGLE_TOP;
final boolean launchSingleInstance = r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE;
final boolean launchSingleTask = r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK;
int launchFlags = intent.getFlags();
if ((launchFlags & Intent.FLAG_ACTIVITY_NEW_DOCUMENT) != 0 &&
        (launchSingleInstance || launchSingleTask)) {
    // We have a conflict between the Intent and the Activity manifest, manifest wins.
    Slog.i(TAG, "Ignoring FLAG_ACTIVITY_NEW_DOCUMENT, launchMode is " +
            "\"singleInstance\" or \"singleTask\"");
    launchFlags &=
            ~(Intent.FLAG_ACTIVITY_NEW_DOCUMENT | Intent.FLAG_ACTIVITY_MULTIPLE_TASK);
} else {
    switch (r.info.documentLaunchMode) {
        case ActivityInfo.DOCUMENT_LAUNCH_NONE:
            break;
        case ActivityInfo.DOCUMENT_LAUNCH_INTO_EXISTING:
            launchFlags |= Intent.FLAG_ACTIVITY_NEW_DOCUMENT;
            break;
        case ActivityInfo.DOCUMENT_LAUNCH_ALWAYS:
            launchFlags |= Intent.FLAG_ACTIVITY_NEW_DOCUMENT;
            break;
        case ActivityInfo.DOCUMENT_LAUNCH_NEVER:
            launchFlags &= ~Intent.FLAG_ACTIVITY_MULTIPLE_TASK;
            break;
    }
}
final boolean launchTaskBehind = r.mLaunchTaskBehind
        && !launchSingleTask && !launchSingleInstance
        && (launchFlags & Intent.FLAG_ACTIVITY_NEW_DOCUMENT) != 0;
if (r.resultTo != null && (launchFlags & Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
    // For whatever reason this activity is being launched into a new
    // task...  yet the caller has requested a result back.  Well, that
    // is pretty messed up, so instead immediately send back a cancel
    // and let the new task continue launched as normal without a
    // dependency on its originator.
    Slog.w(TAG, "Activity is launching as a new task, so cancelling activity result.");
    r.resultTo.task.stack.sendActivityResultLocked(-1,
            r.resultTo, r.resultWho, r.requestCode,
            Activity.RESULT_CANCELED, null);
    r.resultTo = null;
}
```

- 而 5.0 之后这个方法做了修改，修改之后即便启动的页面设置 launchMode 为 singleTask 或 singleInstance，onActivityResult 依旧可以正常工作，也就是说无论设置哪种启动方式，StartActivityForResult 和 onActivityResult() 这一组合都是有效的。所以如果你目前正好基于 5.0 做相关开发，不要忘了向下兼容，这里有个坑请注意避让。
- 所以下面的结论可能不对，需要改天代码再好好验证一下

```
if (sourceRecord == null) {
    // This activity is not being started from another...
    // in this case we -ALWAYS- start a new task.
    if ((launchFlags & Intent.FLAG_ACTIVITY_NEW_TASK) == 0) {
        Slog.w(TAG, "startActivity called from non-Activity context; forcing Intent.FLAG_ACTIVITY_NEW_TASK for: "
              + intent);
        launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
    }
} else if (sourceRecord.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE) {
    // The original activity who is starting us is running as a single
    // instance...  this new activity it is starting must go on its
    // own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
} else if (r.launchMode == ActivityInfo.LAUNCH_SINGLE_INSTANCE
        || r.launchMode == ActivityInfo.LAUNCH_SINGLE_TASK) {
    // The activity being started is a single instance...  it always
    // gets launched into its own task.
    launchFlags |= Intent.FLAG_ACTIVITY_NEW_TASK;
}
if (r.resultTo != null && (launchFlags&Intent.FLAG_ACTIVITY_NEW_TASK) != 0) {
    // For whatever reason this activity is being launched into a new
    // task...  yet the caller has requested a result back.  Well, that
    // is pretty messed up, so instead immediately send back a cancel
    // and let the new task continue launched as normal without a
```

```
        // dependency on its originator.
        Slog.w(TAG, "Activity is launching as a new task, so cancelling activity result.");
        sendActivityResultLocked(-1,
                r.resultTo, r.resultWho, r.requestCode,
            Activity.RESULT_CANCELED, null);
        r.resultTo = null;
    }
```

- 也就是说 startActivityForResult 启动的 activity 有 FLAG_ACTIVITY_NEW_TASK，那么就不能返回结果。(这个结论可能太古老了吧?!!!)

4.7.2 启动任务（Task）：这里要再消化一下！

- Intent filter 中有”android.intent.action.MAIN ”action 和”android.intent.category.LAUNCHER ” category 的 activity 将被标记为 task 的入口。带有这两个标记的 activity 将会显示在应用程序启动器（application launcher）中。
- 第二个比较重要的点是，用户必须能够离开 task 并在之后返回 。因为这个原因，singleTask 和 singleInstance 这两种运行模式只能应用于含有 MAIN 和 LAUNCHER 过滤器的 activity 。打个比方，如果不包含带 MAIN 和 LAUNCHER 过滤器，某个 activity 运行了一个 singleTask 模式的 activity，初始化了一个新的 task，当用户按下 HOME 键时，那个 activity 就被主屏幕“挡住”了，用户再也无法返回到那个 activity。(这里读得昏昏乎乎!!!)
- 类似的情况在 FLAG_ACTIVITY_NEW_TASK 标记上也会出现。如果这个标记会新建一个 task，当用户按下 HOME 键时，必须有一种方式能够让用户返回到那个 activity。有些东西（比如 notification manager）总是要求在外部 task 中启动 activity，在传递给 startActivity 的 intent 中总是包含 FLAG_ACTIVITY_NEW_TASK 标记。
- 对于那种不希望用户离开之后再返回 activity 的情况，可将 finishOnTaskLaunch 属性设置为 true。

4.7.3 Activity Stack

- 可以通过 adb shell dumpsys | grep ActivityRecord 来查看 TASKS 的 ActivityStacks
- 可以通过 adb shell dumpsys activity activities | grep packageName| grep Run 来查看某个 packageName 的 ActivityStatcks

4.8 onNewIntent

- 当通过 singleTop/singleTask 启动 activity 时，如果满足复用条件，则不会创建新的 activity 实例，生命周期就变为 onNewIntent()—->onResart()——>onStart()—->onResume()。
  - Activity 第一启动的时候执行 onCreate()—->onStart()—->onResume() 等后续生命周期函数，也就时说第一次启动 Activity 并不会执行到 onNewIntent().;
  - 而后面如果再有想启动 Activity 的时候，那就是执行 onNewIntent()—->onResart()——>onStart()—->onResume();
  - 如果 android 系统由于内存不足把已存在 Activity 释放掉了，那么再次调用的时候会重新启动 Activity 即执行 onCreate()—->onStart()—->onResume() 等。
- 注意:当调用到 onNewIntent(intent) 的时候,需要在 onNewIntent() 中使用 setIntent(intent) 赋值给 Activity 的 Intent. 否则，后续的 getIntent() 都是得到老的 Intent。

4.9 Activity 所需的任务栈与 TaskAffinity

- 这要从一个参数说起：TaskAffinity(任务相关性)。这个参数标识了一个 activity 所需的任务栈的名字，默认情况下，所有 activity 所需的任务栈的名字为应用的包名。
- TaskAffinity 参数标识着 Activity 所需要的任务栈的名称，默认情况下，一个应用中所有 Activity 所需要的任务栈名称都为该应用的包名。
- TaskAffinity 属性一般跟 singleTask 模式或者跟 allowTaskReparenting 属性结合使用，在其他情况下没有实际意义。

4.9.1 TaskAffinity 和 singleTask 启动模式结合使用

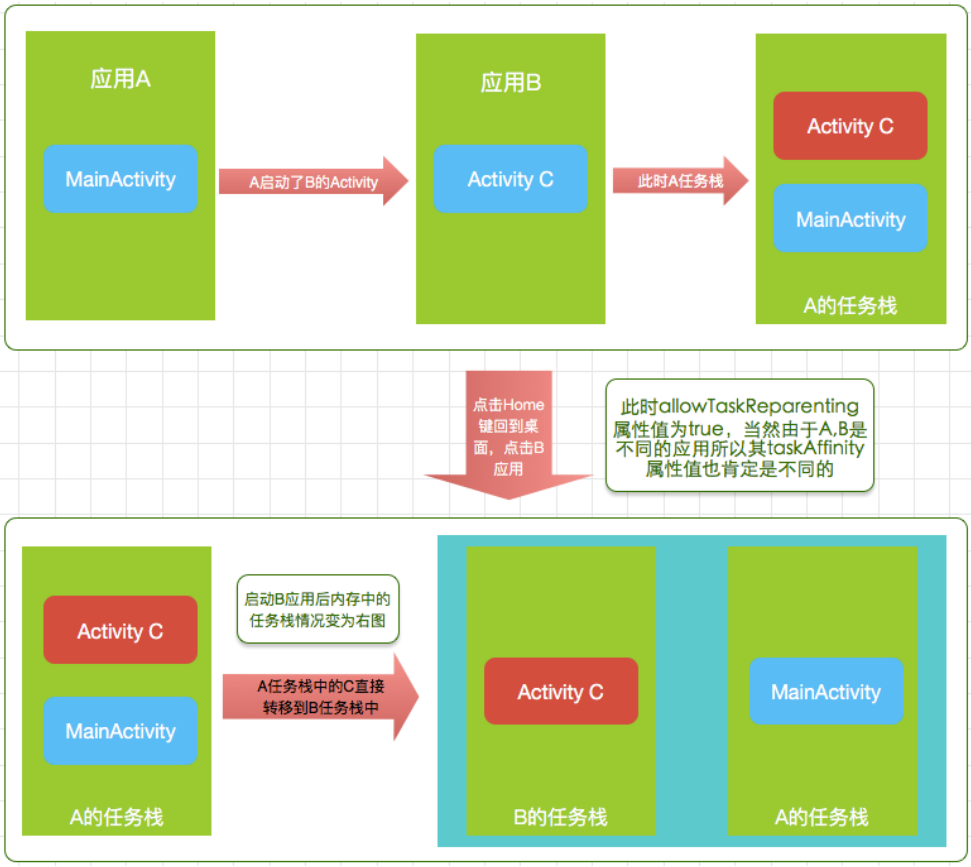
- 当 TaskAffinity 和 singleTask 启动模式结合使用时，当前 Activity 的任务栈名称将与 TaskAffinity 属性指定的值相同，下面我们通过代码来验证，我们同过 MainActivity 来启动 ActivityA，其中 MainActivity 启动模式为默认模式，ActivityA 启动模式为 singleTask，而 TaskAffinity 属性值为 android:taskAffinity="com.zejian.singleTask.affinity"

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="comzejian.myapplication">
  <application
      android:allowBackup="true"
      android:icon="@mipmap/ic_launcher"
      android:label="@string/app_name"
      android:supportsRtl="true"
      android:theme="@style/AppTheme">

    <activity android:name=".MainActivity">
      <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
      </intent-filter>
    </activity>

    <activity android:name=".ActivityA"
              android:launchMode="singleTask"
              android:taskAffinity="com.zejian.singleTask.affinity"
              />

  </application>
</manifest>
```

- 可以通过 singleTask 与 android:taskAffinity 属性相结合的方式来指定我们 Activity 所需要的栈名称，使相应的 Activity 存在于不同的栈中

4.9.2 当 TaskAffinity 和 allowTaskReparenting 结合使用

1. allowTaskReparenting 属性
   - 它的主要作用是 activity 的迁移，即从一个 task 迁移到另一个 task，这个迁移跟 activity 的 taskAffinity 有关。
     - 当 allowTaskReparenting 的值为“true”时，则表示 Activity 能从启动的 Task 移动到有着 affinity 的 Task（当这个 Task 进入到前台时），
     - 当 allowTaskReparenting 的值为“false”，表示它必须呆在启动时呆在的那个 Task 里。如果这个特性没有被设定，元素 (当然也可以作用在每次 activity 元素上) 上的 allowTaskReparenting 属性的值会应用到 Activity 上。默认值为“false”。



- 举个例子，比如现在有两个应用 A 和 B，A 启动了 B 的一个 ActivityC，然后按 Home 键回到桌面，再单击 B 应用时，如果此时，allowTaskReparenting 的值为“true”，那么这个时候并不会启动 B 的主 Activity，而是直接显示已被应用 A 启动的 ActivityC，我们也可以认为 ActivityC 从 A 的任务栈转移到了 B 的任务栈中。

(a) 用代码来码证一下

- ActivityA

```
public class ActivityA extends Activity {
    private Button btnC;
    @Override
        protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_a);
        btnC = (Button) findViewById(R.id.mainC);
        btnC.setOnClickListener(new View.OnClickListener() {
                @Override public void onClick(View v) {
                    Intent intent = new Intent(Intent.ACTION_MAIN);
                    intent.addCategory(Intent.CATEGORY_LAUNCHER); // 去打开 B 应用中的 activity
                    ComponentName cn = new ComponentName("com.cmcm.activitytask2", "com.cmcm.activitytask2.Acti
                    intent.setComponent(cn);
                    startActivity(intent);
                }
            });
    }
}
```

- A 应用的 manifest.xml

```
<activity android:name=".ActivityA">
     <intent-filter>
          <action android:name="android.intent.action.MAIN" />
          <category android:name="android.intent.category.LAUNCHER" />
```

```
    </intent-filter>
</activity>
```

- B 应用中的启动模式以及标志位的设置

```
public class ActivityC extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_c);
    }
}
```

- B 应用的 manifest.xml

```
<activity android:name=".ActivityC" android:exported="true"
      android:allowTaskReparenting="true">
</activity>
```

(b) 查看 Activity 的返回栈

- adb shell dumpsys activity // 找
- ACTIVITY MANAGER RECENT TASKS (dumpsys activity recents)
- ACTIVITY MANAGER ACTIVITIES (dumpsys activity activities)

2. 注意点

- 有点需要说明的是 allowTaskReparenting 仅限于 singleTop 和 standard 模式，这是因为一个 activity 的 affinity 属性由它的 taskAffinity 属性定义（代表栈名），而一个 task 的 affinity 由它的 root activity 定义。所以，一个 task 的 root activity 总是拥有和它所在 task 相同的 affinity。
- 由于以 singleTask 和 singleInstance 启动的 activity 只能是一个 task 的 root activity，因此 allowTaskReparenting 仅限于以 standard 和 singleTop 启动的 activity
- 列一下清单文件中 <activity> 元素的几个关键属性
  - launchMode
  - taskAffinity
  - allowTaskReparenting
  - clearTaskOnLaunch
  - alwaysRetainTaskState
  - finishOnTaskLaunch

3. taskAffinity 在两种情况下起作用：

(a) 当启动 Activity 的 Intent 中带有 FLAG_ACTIVITY_NEW_TASK 标志时。

- 在默认情况下，目标 Activity 将与 startActivity 的调用者处于同一 task 中。但如果用户特别指定了 FLAG_ACTIVITY_NEW_TASK，表明它希望为 Activity 重新开设一个 Task。这时就有两种情况：
  - 假如当前已经有一个 Task，它的 affinity 与新 Activity 是一样的，那么系统会直接用此 Task 来完成操作，而不是另外创建一个 Task；
  - 否则系统需要创建一个 Task。

(b) 当 Activity 中的 allowTaskReparenting 属性设置为 true 时。

- 在这种情况下，Activity 具有" 动态转移" 的能力。举个前面的" 短信" 例子，在默认情况下，该应用程序中的所有 Activity 具有相同的 affinity。
- 当另一个程序启动了" 短信编辑" 时，一开始这个 Activity 和启动它的 Activity 处于同样的 Task 中。但如果" 短信编辑"Activity 指定了 allowTaskReparenting，且后期" 短信" 程序的 Task 转为前台，此时" 短信编辑" 这一 Activity 会被" 挪" 到与它更亲近的" 短信"Task 中。

4.9.3 清空任务栈

- Android 系统除了给我提供了 TaskAffinity 来指定任务栈名称外，还给我提供了清空任务栈的方法，在一般情况下我们只需要在 <activity> 标签中指明相应的属性值即可。

1. clearTaskOnLaunch

   ```
   android:clearTaskOnLaunch
   ```

   - 这个属性用来标记是否从 task 清除除根 Activity 之外的所有的 Activity，“true”表示清除，“false”表示不清除，默认为“false”。这里有点我们必须要注意的，这个属性只对任务栈内的 root Activity 起作用，任务栈内其他的 Activity 都会被忽略。如果 android:clearTaskOnLaunch 属性为“true”，每次我们重新进入这个应用时，我们只会看到根 Activity，任务栈中的其他 Activity 都会被清除出栈。
   - 比如一个应用的 Activity A,B,C，其中 clearTaskOnLaunch 设置为 true，C 为默认值，我们依次启动 A,B,C，点击 HOME, 再在桌面点击图标。启动的是 A，而 B，C 将都被移除当前任务栈。也就是说，当 Activity 的属性 clearTaskOnLaunch 为 true 时将被优先启动，其余的 Activity(B、C) 都被移除任务栈并销毁，除非前面 A 已经 finish 销毁，后面的已注册 clearTaskOnLaunch 为 true 的 activity(B) 才会生效。

   -特别地,如果我们的应用中引用到了其他应用的 Activity,这些 Activity 设置了 android:allowTaskRep 属性为“true”，则它们会被重新宿主到有共同 affinity 的 task 中。

2. android:finishOnTaskLaunch -finishOnTaskLaunch 属性与 clearTaskOnLaunch 有些类似，它们的区别是 finishOnTaskLaunch 是作用在自己身上 (把自己移除任务栈，不影响别的 Activity)，而 clearTaskOnLaunch 则是作用在别人身上 (把别的 Activity 移除任务栈)，如果我们把 Activity 的 android:finishOnTaskLaunch 属性值设置为 true 时，离开这个 Activity 所依赖的任务栈后，当我们重新返回时，该 Activity 将会被 finish 掉，而且其他 Activity 不会受到影响。

3. android:alwaysRetainTaskState -alwaysRetainTaskState 实际上是给了当前 Activity 所在的任务栈一个“免死金牌”，如果当前 Activity 的 android:alwaysRetainTaskState 设置为 true 时，那么该 Activity 所在的任务栈将不会受到任何清理命令的影响，一直保持当前任务栈的状态。

4.10 启动模式的应用场景

4.10.1 SingleTask 模式的运用场景

- 最常见的应用场景就是保持我们应用开启后仅仅有一个 Activity 的实例。最典型的样例就是应用中展示的主页 (Home 页)。
- 假设用户在主页跳转到其他页面，运行多次操作后想返回到主页，假设不使用 SingleTask 模式，在点击返回的过程中会多次看到主页，这明显就是设计不合理了。

4.10.2 SingleTop 模式的运用场景

- 假设你在当前的 Activity 中又要启动同类型的 Activity，此时建议将此类型 Activity 的启动模式指定为 SingleTop，能够降低 Activity 的创建，节省内存!

4.10.3 SingleInstance 模式的运用场景

- SingleInstance 是 activity 启动的一种模式，一般做应用层开发很少用到，我一般用到的 app 定时提醒会用到这个模式吧。这个模式使用起来有很多坑，假设有 activityA，activityB，activityC 这三个 activity，我们将 activityB 设置为 SingleInstance

1. 第一种情况
   - A 开启 B，B 开启 C，如果 finish activityC，那么 activityA 会显示而不是我们想要的 activityB，这是因为 activityB 和 activityA、activityC 所处的栈不同，C 关闭了，就要显示 C 所处栈的下一个 activity，解决这个问题办法很多，我自己用的方法是通过记录开启 activity，在被关闭的 activity 的 finish 方法中重新开启 activityB。
2. 第二种情况 A 开启 B，然后按 home 键，再从左面点开应用，显示的是 A，这是因为 launch 启动我们应用的时候会从默认的栈找到栈顶的 activity 显示，这个解决办法的思路跟第一种差不多，也就不献丑了
3. 第三种情况
   - A 开启 C，C 开启 B，B 开启 A，结果显示的是 C，这还是两个栈造成的，B 开启 A 的时候，其实是到达 A 所处的栈，栈顶是 C，所以就显示 C 了，解决办法是用 flag 把默认栈 activity 清理了，重新开启 A，或者回退到 C 时再开启 A。
4. 总结
   - 三种情况的解决方法都是基于页面少的情况，如果页面多了会产生更多的问题
   - 为了必避免这个问题，最好不用在中间层使用 SingleInstance
   - TIPS:
     - (1) 如果想让 C 和 B 同一个栈，那就使用 taskinfinity，给他俩设置同样的栈名
     - (2)onActivityResult 不能与 SingleInstance 不能一起使用，因为不同栈

4.10.4 standard 运用场景

- Activity 的启动默认就是这种模式。在 standard 模式下，每次启动一个 Activity 都会创建一个新的实例;
- 在正常应用中正常打开和关闭页面就可以了，退出整个 app 就关闭所有的页面

4.11 Activity 时的生命周期不同

- 由于当一个 Activity 设置了 SingleTop 或者 SingleTask 模式或者 SingleInstance 模式后，跳转此 Activity 出现复用原有 Activity 的情况时，此 Activity 的 onCreate 方法将不会再次运行。onCreate 方法仅仅会在第一次创建 Activity 时被运行。
- 而一般 onCreate 方法中会进行该页面的数据初始化、UI 初始化，假设页面的展示数据无关页面跳转传递的参数，则不必操心此问题，若页面展示的数据就是通过 getInten() 方法来获取，那么问题就会出现：getInten() 获取的一直都是老数据，根本无法接收跳转时传送的新数据!
- 这时我们须要另外一个回调 onNewIntent(Intent intent) 方法。此方法会传入最新的 intent，这样我们就能够解决上述问题。这里建议的方法是又一次去 setIntent。然后又一次去初始化数据和 UI

```
/** 复用 Activity 时的生命周期回调 */
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    setIntent(intent);
    initData();
    initView();
}
```

4.12 实际中的栈管理类

- 管理 Activity 的类，一般在 BaseActivity 会调用这个类，然后所有的 Activity 继承 BaseActivity，这样管理好整个项目的 Activity

```
public class ActivityStackManager { // activity 堆栈管理
    private static ActivityStackManager mInstance;
    private static Stack<Activity> mActivityStack;
    public static ActivityStackManager getInstance() {
        if (null == mInstance)
            mInstance = new ActivityStackManager();
        return mInstance;
    }
    private ActivityStackManager() {
        mActivityStack = new Stack<Activity>();
    }
    public void addActivity(Activity activity) {  // 入栈
        mActivityStack.push(activity);
    }
    public void removeActivity(Activity activity) {  // 出栈
        mActivityStack.remove(activity);
    }
    public void finishAllActivity() {  // 彻底退出
        Activity activity;
        while (!mActivityStack.empty()) {
            activity = mActivityStack.pop();
            if (activity != null)
                activity.finish();
        }
    }
    public void finishActivity(Class<?> cls) {  // 结束指定类名的 Activity
        for (Activity activity : mActivityStack) {
            if (activity.getClass().equals(cls)) {
                finishActivity(activity);
            }
        }
    }
    public boolean checkActivity(Class<?> cls) {  // 查找栈中是否存在指定的 activity
        for (Activity activity : mActivityStack) {
            if (activity.getClass().equals(cls)) {
                return true;
            }
        }
        return false;
    }
    public void finishActivity(Activity activity) {  // 结束指定的 Activity
        if (activity != null) {
            mActivityStack.remove(activity);
            activity.finish();
            activity = null;
        }
    }
    public boolean finishToActivity(Class<? extends Activity> actCls, boolean isIncludeSelf) { // finish 指定的 activity 之
        List<Activity> buf = new ArrayList<Activity>();
        int size = mActivityStack.size();
        Activity activity = null;
        for (int i = size - 1; i >= 0; i--) {
            activity = mActivityStack.get(i);
            if (activity.getClass().isAssignableFrom(actCls)) {
                for (Activity a : buf)
                    a.finish();
                return true;
            } else if (i == size - 1 && isIncludeSelf)
                buf.add(activity);
```

```
            else if (i != size - 1)
                buf.add(activity);
        }
        return false;
    }
}
```

5 Content Provider 内存承载器

- ContentProvider 的底层是采用 Android 中的 Binder 机制

5.1 统一资源标识符（URI）

- 定义：Uniform Resource Identifier，即统一资源标识符
- 作用：唯一标识 ContentProvider & 其中的数据
  - 外界进程通过 URI 找到对应的 ContentProvider & 其中的数据，再进行数据操作
- 具体使用
  - URI 分为系统预置 & 自定义，分别对应系统内置的数据（如通讯录、日程表等等）和自定义数据库
    - 关于系统预置 URI 此处不作过多讲解，需要的同学可自行查看
    - 此处主要讲解自定义 URI

- 授权信息（Authority）：ContentProvider 的唯一标识符。所以这个一定得 match 对了

```
Uri uri = Uri.parse("content://com.carson.provider/User/1") // 设置 URI
// 上述 URI 指向的资源是: 名为 `com.carson.provider`的 `ContentProvider` 中表名 为 `User` 中的 `id`为 1 的数据

// 特别注意: URI 模式存在匹配通配符: * # (两个)
//  *: 匹配任意长度的任何有效字符的字符串
//     以下的 URI 表示 匹配 provider 的任何内容
//     content://com.example.app.provider/*
// #: 匹配任意长度的数字字符的字符串
//     以下的 URI 表示 匹配 provider 中的 table 表的所有行
//     content://com.example.app.provider/table/#
```

- uri 的各个部分在安卓中都是可以通过代码获取的，下面我们就以下面这个 uri 为例来说下获取各个部分的方法：
- http://www.baidu.com:8080/wenku/jiatiao.html?id=123456&name=jack

```
getScheme() // 获取 Uri 中的 scheme 字符串部分, 在这里是 http
getHost()   // 获取 Authority 中的 Host 字符串, 即 www.baidu.com
getPost()   // 获取 Authority 中的 Port 字符串, 即 8080
getPath()   // 获取 Uri 中 path 部分, 即 wenku/jiatiao.html
getQuery()  // 获取 Uri 中的 query 部分, 即 id=15&name=du
```

5.2 MIME 数据类型

- 作用：指定某个扩展名的文件用某种应用程序来打开
  - 如指定.html 文件采用 text 应用程序打开、指定.pdf 文件采用 flash 应用程序打开

5.2.1 ContentProvider 根据 URI 返回 MIME 类型

```
ContentProvider.geType(uri) ;
```

5.2.2 MIME 类型组成

- 每种 MIME 类型由 2 部分组成 = 类型 + 子类型
- MIME 类型是一个包含 2 部分的字符串

```
text / html
// 类型 = text、子类型 = html
text/css
text/xml
application/pdf
```

5.2.3 MIME 类型形式

- MIME 类型有 2 种形式：单条记录, 或是多条记录

```
// 形式 1：单条记录
vnd.android.cursor.item/自定义
// 形式 2：多条记录（集合）
vnd.android.cursor.dir/自定义

// 注：
  // 1. vnd：表示父类型和子类型具有非标准的、特定的形式。
  // 2. 父类型已固定好（即不能更改），只能区别是单条还是多条记录
  // 3. 子类型可自定义
```

- 实例说明

```
<-- 单条记录 -->
  // 单个记录的 MIME 类型
  vnd.android.cursor.item/vnd.yourcompanyname.contenttype

  // 若一个 Uri 如下
  content://com.example.transportationprovider/trains/122
  // 则 ContentProvider 会通过 ContentProvider.geType(url) 返回以下 MIME 类型
  vnd.android.cursor.item/vnd.example.rail

<-- 多条记录 -->
  // 多个记录的 MIME 类型
  vnd.android.cursor.dir/vnd.yourcompanyname.contenttype
  // 若一个 Uri 如下
  content://com.example.transportationprovider/trains
  // 则 ContentProvider 会通过 ContentProvider.geType(url) 返回以下 MIME 类型
  vnd.android.cursor.dir/vnd.example.rail
```

5.3 ContentProvider 类

5.3.1 组织数据方式

- ContentProvider 主要以表格的形式组织数据
  - 同时也支持文件数据，只是表格形式用得比较多
  - 每个表格中包含多张表，每张表包含行 & 列，分别对应记录 & 字段, 同数据库

5.3.2 主要方法

- 进程间共享数据的本质是：添加、删除、获取 & 修改（更新）数据 (CRUD: create / read / update / delete)
- 所以 ContentProvider 的核心方法也主要是上述几个作用

```
  // 外部进程向 ContentProvider 中添加数据
  public Uri insert(Uri uri, ContentValues values)

  // 外部进程 删除 ContentProvider 中的数据
  public int delete(Uri uri, String selection, String[] selectionArgs)

  // 外部进程更新 ContentProvider 中的数据
  public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)

  // 外部应用 获取 ContentProvider 中的数据
  public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs,  String sortOrder)

// 注：
  // 1. 上述 4 个方法由外部进程回调，并运行在 ContentProvider 进程的 Binder 线程池中（不是主线程）
  // 2. 存在多线程并发访问，需要实现线程同步
    // a. 若 ContentProvider 的数据存储方式是使用 SQLite & 并且只有一个，则不需要，因为 SQLite 内部实现好了线程同步，若是多个 SQL
    // b. 若 ContentProvider 的数据存储方式是内存，则需要自己实现线程同步

// 2 个其他方法 -->
// ContentProvider 创建后 或 打开系统后其它进程第一次访问该 ContentProvider 时 由系统进行调用
public boolean onCreate()
// 注：运行在 ContentProvider 进程的主线程，故不能做耗时操作

// 得到数据类型，即返回当前 Url 所代表数据的 MIME 类型
public String getType(Uri uri)
```

- Android 为常见的数据（如通讯录、日程表等）提供了内置了默认的 ContentProvider
- 但也可根据需求自定义 ContentProvider，但上述 6 个方法必须重写
- 数据访问的方法 insert，delete 和 update 可能被多个线程同时调用，此时必须是线程安全的。(前面提到过)
- 如果操作的数据属于集合类型，那么 MIME 类型字符串应该以 vnd.android.cursor.dir/ 开头，
  - 要得到所有 tablename 记录：Uri 为 content://com.wang.provider.myprovider/tablename，那么返回的 MIME 类型字符串应该为：vnd.android.cursor.dir/table。
- 如果要操作的数据属于非集合类型数据，那么 MIME 类型字符串应该以 vnd.android.cursor.item/ 开头，
  - 要得到 id 为 10 的 tablename 记录，Uri 为 content://com.wang.provider.myprovider/tablename 那么返回的 MIME 类型字符串为：vnd.android.cursor.item/tablename 。
- ContentProvider 类并不会直接与外部进程交互，而是通过 ContentResolver 类

5.4 ContentResolver 类

- 统一管理不同 ContentProvider 间的操作
  - 即通过 URI 即可操作不同的 ContentProvider 中的数据
  - 外部进程通过 ContentResolver 类从而与 ContentProvider 类进行交互
- 为什么要使用通过 ContentResolver 类从而与 ContentProvider 类进行交互，而不直接访问 ContentProvider 类?

  - 一般来说，一款应用要使用多个 ContentProvider，若需要了解每个 ContentProvider 的不同实现从而再完成数据交互，操作成本高 & 难度大
  - 所以再 ContentProvider 类上加多了一个 ContentResolver 类对所有的 ContentProvider 进行统一管理。

- ContentResolver 类提供了与 ContentProvider 类相同名字 & 作用的 4 个方法

```
// 外部进程向 ContentProvider 中添加数据
public Uri insert(Uri uri, ContentValues values)

// 外部进程 删除 ContentProvider 中的数据
public int delete(Uri uri, String selection, String[] selectionArgs)

// 外部进程更新 ContentProvider 中的数据
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs)

// 外部应用 获取 ContentProvider 中的数据
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder)
```

- 实例说明

```
// 使用 ContentResolver 前，需要先获取 ContentResolver
// 可通过在所有继承 Context 的类中 通过调用 getContentResolver() 来获得 ContentResolver
ContentResolver resolver =  getContentResolver();

// 设置 ContentProvider 的 URI
Uri uri = Uri.parse("content://cn.scu.myprovider/user");

// 根据 URI 操作 ContentProvider 中的数据
// 此处是获取 ContentProvider 中 user 表的所有记录
Cursor cursor = resolver.query(uri, null, null, null, "userid desc");
```

- Android 提供了 3 个用于辅助 ContentProvider 的工具类：
  - ContentUris
  - UriMatcher
  - ContentObserver

5.5 ContentUris 类

- 作用：操作 URI
- 核心方法有两个：withAppendedId（）&parseId（）

```
// withAppendedId() 作用：向 URI 追加一个 id
Uri uri = Uri.parse("content://cn.scu.myprovider/user")
Uri resultUri = ContentUris.withAppendedId(uri, 7);
// 最终生成后的 Uri 为：content://cn.scu.myprovider/user/7

// parseId() 作用：从 URL 中获取 ID
Uri uri = Uri.parse("content://cn.scu.myprovider/user/7")
long personid = ContentUris.parseId(uri);
//获取的结果为:7
```

5.6 UriMatcher 类

- 在 ContentProvider 中注册 URI
- 根据 URI 匹配 ContentProvider 中对应的数据表

```
// 步骤 1：初始化 UriMatcher 对象
    UriMatcher matcher = new UriMatcher(UriMatcher.NO_MATCH);
    // 常量 UriMatcher.NO_MATCH = 不匹配任何路径的返回码
    // 即初始化时不匹配任何东西

// 步骤 2：在 ContentProvider 中注册 URI(addURI())
    int URI_CODE_a = 1;
    int URI_CODE_b = 2;
    matcher.addURI("cn.scu.myprovider", "user1", URI_CODE_a);
    matcher.addURI("cn.scu.myprovider", "user2", URI_CODE_b);
    // 若 URI 资源路径 = content://cn.scu.myprovider/user1 ，则返回注册码 URI_CODE_a
    // 若 URI 资源路径 = content://cn.scu.myprovider/user2 ，则返回注册码 URI_CODE_b

    //如果 match() 方法匹配 content://com.wang.provider.myprovider/tablename/11 路径，返回匹配码为 2
    matcher.addURI("com.wang.provider.myprovider", "tablename/#", 2);

// 步骤 3：根据 URI 匹配 URI_CODE，从而匹配 ContentProvider 中相应的资源（match()）
@Override public String getType(Uri uri) {
      Uri uri = Uri.parse(" content://cn.scu.myprovider/user1");

      switch(matcher.match(uri)) {
      // 根据 URI 匹配的返回码是 URI_CODE_a
      // 即 matcher.match(uri) == URI_CODE_a
      case URI_CODE_a:
          // 如果根据 URI 匹配的返回码是 URI_CODE_a，则返回 ContentProvider 中的名为 tableNameUser1 的表
          return tableNameUser1;
      case URI_CODE_b:
          // 如果根据 URI 匹配的返回码是 URI_CODE_b，则返回 ContentProvider 中的名为 tableNameUser2 的表
          return tableNameUser2;
    }
}
```

- 注意，添加第三个个 URI 时，路径后面的 id 采用了通配符形式“#”，表示只要前面三个部分都匹配上了就 OK。
- 第三步，注册完需要匹配的 Uri 后，可以使用 matcher.match(Uri) 方法对输入的 Uri 进行匹配，如果匹配就返回对应的匹配码，匹配码为调用 addURI() 方法时传入的第三个参数。

5.7 ContentObserver 类

- 定义：内容观察者
- 作用：观察 Uri 引起 ContentProvider 中的数据变化 & 通知外界（即访问该数据访问者）
  - 当 ContentProvider 中的数据发生变化（增、删 & 改）时，就会触发该 ContentObserver 类

```
// 步骤 1：注册内容观察者 ContentObserver
// 通过 ContentResolver 类进行注册，并指定需要观察的 URI
getContentResolver().registerContentObserver(uri);

// 步骤 2：当该 URI 的 ContentProvider 数据发生变化时，通知外界（即访问该 ContentProvider 数据的访问者）
public class UserContentProvider extends ContentProvider {
    public Uri insert(Uri uri, ContentValues values) {
        db.insert("user", "userid", values);
        // 通知访问者
        getContext().getContentResolver().notifyChange(uri, null);
    }
}
// 步骤 3：解除观察者
getContentResolver().unregisterContentObserver(uri);
// 同样需要通过 ContentResolver 类进行解除
```

- 上面说得可能还不是太彻底，下面再重新写一下
- 如果 ContentProvider 的访问者需要知道数据发生的变化，可以在 ContentProvider 发生数据变化时调用 getContentResolver().notifyChange(uri, null) 来通知注册在此 URI 上的访问者。只给出类中监听部分的代码：

```
public class MyProvider extends ContentProvider {
   public Uri insert(Uri uri, ContentValues values) {
      db.insert("tablename", "tablenameid", values);
      getContext().getContentResolver().notifyChange(uri, null);
   }
}
```

- 而访问者必须使用 ContentObserver 对数据（数据采用 uri 描述）进行监听，当监听到数据变化通知时，系统就会调用 ContentObserver 的 onChange() 方法：

```
getContentResolver().registerContentObserver(Uri.parse("content://com.ljq.providers.personprovider/person"),
       true, new PersonObserver(new Handler()));
public class PersonObserver extends ContentObserver{
   public PersonObserver(Handler handler) {
      super(handler);
   }
   public void onChange(boolean selfChange) {
      //to do something
   }
}
```

5.8 优点

5.8.1 安全

- ContentProvider 为应用间的数据交互提供了一个安全的环境：允许把自己的应用数据根据需求开放给其他应用进行增、删、改、查，而不用担心因为直接开放数据库权限而带来的安全问题

5.8.2 访问简单 & 高效

- 对比于其他对外共享数据的方式，数据访问方式会因数据存储的方式而不同：
  - 采用文件方式对外共享数据，需要进行文件操作读写数据；
  - 采用 Sharedpreferences 共享数据，需要使用 sharedpreferences API 读写数据，这使得访问数据变得复杂 & 难度大。
  - 而采用 ContentProvider 方式，其解耦了底层数据的存储方式，使得无论底层数据存储采用何种方式，外界对数据的访问方式都是统一的，这使得访问简单 & 高效
  - 如一开始数据存储方式采用 SQLite 数据库，后来把数据库换成 MongoDB，也不会对上层数据 ContentProvider 使用代码产生影响

5.9 ContentProvider 的工作过程

- ContentProvider 是一种数据共享型组件，用于向其他组件乃至其他应用共享数据。需要注意的是 ContentProvider 内部的 insert、delete、update 和 query 方法需要处理好线程同步，因为这几个方法是在 Binder 线程池中被调用的，另外 ContentProvider 组件不需要手动停止。

5.9.1 一、ContentProvider 的启动过程

- 当 ContentProvider 所在进程启动时，ContentProvider 会同时启动并被发布到 AMS 中，这个时候 ContentProvider 的 onCreate 要先于 Application 的 onCreate 而执行，这在四大组件中是少有的一个现象。
- 当一个应用启动的时候，入口方法为 ActivityThread 的 main 方法，期间 ContentProvider 的启动过程流程如下图。

- 其中，黑色虚线箭头表示调用了重载方法，最终调用箭头指向方法；紫色虚线箭头表示远程调用的 IPC 过程。橙色流程先于绿色流程执行。对于 ContentProvider 的启动过程，需要注意几点。

1. ContentProvider 可以是单实例也可以是多实例

    - 一般来说，ContentProvider 是单实例的。但是这是可以设置的，它的属性 android:multiprocess 为 false 时表示单实例，为 true 表示多实例，其默认值为 false。实际开发中，多实例缺乏使用价值，因此默认 ContentProvider 为单实例。

2. ActivityThread 会先加载 ContentProvider 再调用 Application 的 onCreate 方法

    - ActivityThread 的 attach 方法的部分源码如下。

```
private void handleBindApplication(AppBindData data) {
    // ...
    // 创建 ContextImpl 和 Instrumentation
    final ContextImpl instrContext = ContextImpl.createAppContext(this, pi,
            appContext.getOpPackageName());
    try {
        final ClassLoader cl = instrContext.getClassLoader();
        mInstrumentation = (Instrumentation)
            cl.loadClass(data.instrumentationName.getClassName()).newInstance();
    } catch (Exception e) {
        throw new RuntimeException(
            "Unable to instantiate instrumentation "
            + data.instrumentationName + ": " + e.toString(), e);
    }
    final ComponentName component = new ComponentName(ii.packageName, ii.name);
    mInstrumentation.init(this, instrContext, appContext, component,
            data.instrumentationWatcher, data.instrumentationUiAutomationConnection);
    // ....
    // 创建 Application 对象
    Application app;
    final StrictMode.ThreadPolicy savedPolicy = StrictMode.allowThreadDiskWrites();
    final StrictMode.ThreadPolicy writesAllowedPolicy = StrictMode.getThreadPolicy();
    try {
        // If the app is being launched for full backup or restore, bring it up in
        // a restricted environment with the base application class.
        app = data.info.makeApplication(data.restrictedBackupMode, null);
        // ...
        // 启动当前进程的 ContentProvider 并调用其 onCreate 方法，绿色流程
        if (!data.restrictedBackupMode) {
            if (!ArrayUtils.isEmpty(data.providers)) {
            installContentProviders(app, data.providers);
            }
        }
    // ....
    // 调用 Application 的 onCreate 方法
    try {
        mInstrumentation.callApplicationOnCreate(app);
    } catch (Exception e) {
        if (!mInstrumentation.onException(app, e)) {
            throw new RuntimeException(
              "Unable to create application " + app.getClass().getName()
              + ": " + e.toString(), e);
        }
    }
}
```

    - 从源码可以得知，handleBindApplication 方法的工作主要分为四个：
        - 创建 ContentImpl 和 Instrumentation。
        - 创建 Application 对象。
        - 启动当前进程的 ContentProvider（即调用 installContentProviders 方法）并调用其 onCreate 方法，其对应于上述流程图的绿色流程部分。
        - 调用 Application 的 onCreate 方法

- 因此在一个应用启动的时候，ActivityThread 会先加载 ContentProvider，调用 ContentProvider 的 onCreate 方法，接着才调用 Application 的 onCreate 方法。

3. publicContentProviders 存储 ContentProvider -在 handleBindApplication 调用 installContentProviders 方法加载 ContentProvider 进行启动的同时，通过 AMS 的 publicContentProviders 方法将 ContentProvider 发布到 AMS 并将其对应的 ContentProviderRecode 存储到 ProviderMap 中，这样一来外部调用者就可以直接从 AMS 中获取 ContentProvider。其部分源码如下。

```
public final void publishContentProviders(IApplicationThread caller,
        List<ContentProviderHolder> providers) {
    // ...
    for (int i = 0; i < N; i++) {
        ContentProviderHolder src = providers.get(i);
        if (src == null || src.info == null || src.provider == null) {
            continue;
        }
        ContentProviderRecord dst = r.pubProviders.get(src.info.name);
        if (DEBUG_MU) Slog.v(TAG_MU, "ContentProviderRecord uid = " + dst.uid);
        if (dst != null) {
            ComponentName comp = new ComponentName(dst.info.packageName, dst.info.name);
            // 保存 ContentProvider
            mProviderMap.putProviderByClass(comp, dst);
            String names[] = dst.info.authority.split(";");
            for (int j = 0; j < names.length; j++) {
                mProviderMap.putProviderByName(names[j], dst);
            }
        }
        // ...
    }
    // ....
}
```

-至此 ContentProvider 的启动过程就分析完了。

5.9.2 二、ContentProvider 启动过程的触发

- 当通过 ContentProvider 的四个方法的任何一个调用可以触发 ContentProvider 的启动过程。而当 ContentProvider 所在进程未启动时，第一次访问它就会触发 ContentProvider 的创建，同时伴随着 ContentProvider 所在进程的启动。

- 我们知道，访问 ContentProvider 需要通过 ContentResolver，ContentResolver 是一个抽象类，通过 Context 的 getContentResolver 方法获取的实际上是 ApplicationContentResolver 对象。调用 query 方法触发启动过程的具体流程如下。

- 这个流程看起来相当复杂，实际上总结起来就是：若当前 Application A 有 ContentProvider 对象，用该对象进行 IPC 过程调用 query；否则从 AMS 获取 ContentProvider，AMS 中已经有发布的 ContentProvider 则返回，否则启动其所在进程并加载 ContentProvider，等发布到 AMS 后返回。

- 需要注意的一点是，这里一直提及的 ContentProvider 并非本身，因为其无法进行进程间通信，其指代的是 ContentProvider.Transport。该类是 Binder 对象，实现了 IContentProvider 接口。当 A 获得 ContentProvider 的 Binder 对象即 ContentProvider.Transport 对象，以进程通信的方式调用 B 应用的 query 方法，最终调用子类实现的 query 方法。具体见上述流程图。

- 至于其他的三个方法：insert、update 和 delete，其流程就与之类似了。

5.10 IPC interprocess ContentProvider 实例关键点纪录

5.10.1 app 1: function as server

- AndroidManifest.xml

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="scut.carson_ho.contentprovider">
  <permission
      android:name="scut.carson_ho.PROVIDER"
      android:label="provider permission"
      android:protectionLevel="normal" />
  <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"

        android:supportsRtl="true"
        android:grantUriPermissions="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:name="MyProvider"
                  android:authorities="cn.scu.myprovider"
                  android:enabled="true"
                  android:exported="true"
                  android:permission="scut.carson_ho.PROVIDER"/>
    </application>
</manifest>
```

- the <permissions> section, <provider> section, and authorities, permissions

5.10.2 app 2: functions as client

- AndroidManifest

```
<uses-permission android:name="scut.carson_ho.Read"/>
<uses-permission android:name="scut.carson_ho.Write"/>
<uses-permission android:name="scut.carson_ho.PROVIDER"/>
```

- 客户端的许可要与服务器端的相对应，可细分为读写许可
- MainActivity.java

```
// 设置 URI
Uri uri_user = Uri.parse("content://cn.scu.myprovider/user");
```

  - 这里设置 uri 的“content://AUTHORITY/tablename”需要与服务器端 manifest 中的 <provider>android:authorities 相契合

6 Services: 文件夹里有一个文件总结，但是太杂乱，需要删改和总结

- Service 分为两种：
  - 本地服务，属于同一个应用程序，通过 startService 来启动或者通过 bindService 来绑定并且获取代理对象。如果只是想开个服务在后台运行的话，直接 startService 即可，如果需要相互之间进行传值或者操作的话，就应该通过 bindService。

  - 远程服务（不同应用程序之间），通过 bindService 来绑定并且获取代理对象。

- 对应的生命周期如下：
  - context.startService() -> onCreate()-> onStartCommand()->Service running–调用 context.stopService() ->onDestroy()
  - context.bindService()->onCreate()->onBind()->Service running--调用 >onUnbind() -> onDestroy

- Service 默认是运行在 main 线程的，因此 Service 中如果需要执行耗时操作（大文件的操作，数据库的拷贝，网络请求，文件下载等）的话应该在子线程中完成。
  - ！特殊情况是：Service 在清单文件中指定了在其他进程中运行。

7 性能优化

7.1 内存优化

7.1.1 珍惜 Service，尽量使得 Service 在使用的时候才处于运行状态。尽量使用 IntentService

IntentService 在内部其实是通过线程以及 Handler 实现的，当有新的 Intent 到来的时候，会创建线程并且处理这个 Intent，处理完毕以后就自动销毁自身。因此使用 IntentService 能够节省系统资源。

7.1.2 内存紧张的时候释放资源（例如 UI 隐藏的时候释放资源等）。复写 Activity 的回调方法。

```
@Override public void onLowMemory() {
    super.onLowMemory();
    }
    @Override
    public void onTrimMemory(int level) {
    super.onTrimMemory(level);
    switch (level) {
    case TRIM_MEMORY_COMPLETE:
    //. . .
    break;
    case 其他:
    }
    }
```

7.1.3 通过 Manifest 中对 Application 配置更大的内存，但是一般不推荐

```
android:largeHeap="true"
```

7.1.4 避免 Bitmap 的浪费，应该尽量去适配屏幕设备。尽量使用成熟的图片加载框架，Picasso，Fresco，Glide 等。

7.1.5 使用优化的容器，SparseArray 等

7.1.6 其他建议：尽量少用枚举变量，尽量少用抽象，尽量少增加类，避免使用依赖注入框架，谨慎使用 library，使用代码混淆，时当场合考虑使用多进程等。

7.1.7 避免内存泄漏（本来应该被回收的对象没有被回收）

- 一旦 APP 的内存短时间内快速增长或者 GC 非常频繁的时候，就应该考虑是否是内存泄漏导致的。

  分析方法

  1. 使用 Android Studio 提供的 Android Monitors 中 Memory 工具查看内存的使用以及没使用的情况。
  2. 使用 DDMS 提供的 Heap 工具查看内存使用情况，也可以手动触发 GC。
  3. 使用性能分析的依赖库，例如 Square 的 LeakCanary，这个库会在内存泄漏的前后通过 Notification 通知你。

8 activity lifecycle

8.0.1 Activity 横竖屏切换生命周期变化

1. 新建一个 Activity，并把各个生命周期打印出来 onCreate, 创建 activity 时调用。设置在该方法中，还以 Bundle 中可以提出用于创建该 Activity 所需的信息。onStart, activity 变为在屏幕上对用户可见时，即获得焦点时，会调用。onResume, activity 开始与用户交互时调用（无论是启动还是重新启动一个活动，该方法总是被调用的）onSaveInstanceState onPause, activity 被暂停或收回 cpu 和其他资源时调用，该方法用于保存活动状态的 onStop, activity 被停止并转为不可见阶段及后续的生命周期事件时，即失去焦点时调用 onDestroy, activity 被完全从系统内存中移除时调用，该方法被调用可能是因为有人直接调用 finish() 方法或者系统决定停止该活动以释放资源。onRestoreInstanceState, Android 在横竖排切换时候，将主动销毁 activity 和重新创建一个新的 activity 出来，在此过程中，onRestoreInstanceState 就要被回调 onConfigurationChanged, 配置指定属性后, 屏幕方向发生变化后回调此函数.

2. 运行 Activity，得到如下信息

   ```
   onCreate  -->
   onStart  -->
   onResume  -->
   ```

3. 按 crtl+f12 切换成横屏时

   ```
   onPause  -->
   onStop  -->
   onDestroy  -->
   onCreate  -->
   onStart  -->
   onRestoreInstanceState  -->
   onResume  -->
   ```

4. 再按 crtl+f12 切换成竖屏时，发现又打印了相同的 log

   ```
   onPause  -->
   onStop  -->
   onDestroy  -->
   onCreate  -->
   onStart  -->
   onRestoreInstanceState  -->
   onResume  -->
   ```

5. 修改 AndroidManifest.xml 把该 Activity 添加

   ```
   android:configChanges="orientation",
   ```

   执行步骤 3(切换成横屏时)

```
onPause  -->
onStop  -->
onDestroy  -->
onCreate  -->
onStart  -->
onRestoreInstanceState  -->
onResume  -->
```

6. 再执行步骤 4(切换竖屏)，发现再打印相同信息

```
onPause  -->
onStop  -->
onDestroy  -->
onCreate  -->
onStart  -->
onRestoreInstanceState  -->
onResume  -->
```

8.0.2 Why do developers often put app initialization code in the Application class?

- The Application class is instantiated before any other class when the process for the application is created.

9 fragmnet

9.0.1 What are Retained Fragments?

- By default, Fragments are destroyed and recreated along with their parent Activity’s when a configuration change occurs.
- Calling setRetainInstance(true) allows us to bypass this destroy-and-recreate cycle, signaling the system to retain the current instance of the fragment when the activity is recreated.

9.0.2 How would you communicate between two Fragments?

All Fragment-to-Fragment communication is done either through a shared ViewModel or through the associated Activity. Two Fragments should never communicate directly.

- The recommended way to communicate between fragments is to create a shared ViewModel object. Both fragments can access the ViewModel through their containing Activity. The Fragments can update data within the ViewModel and if the data is exposed using LiveData the new state will be pushed to the other fragment as long as it is observing the LiveData from the ViewModel.

```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData <Item> selected = new MutableLiveData < Item > ();
    public void select(Item item) {
        selected.setValue(item);
    }
    public LiveData <Item> getSelected() {
        return selected;
    }
}
public class MasterFragment extends Fragment {
    private SharedViewModel model;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
                model.select(item);
            });
```

```
    }
}
public class DetailFragment extends Fragment {
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        SharedViewModel model = ViewModelProviders.of(getActivity()).get(SharedViewModel.class);
        model.getSelected().observe(this, {
                item ->
                    // Update the UI.
                    // model.select(item); // 这行充当占位符，复制的上面的
                    });
    }
}
```

- Another way is to define an interface in your Fragment A, and let your Activity implement that Interface. Now you can call the interface method in your Fragment, and your Activity will receive the event. Now in your activity, you can call your second Fragment to update the textview with the received value.

9.1 transaction stack backstack

9.1.1 What is the chief purpose of line five in this code snippet?

```
override fun onCreate(savedInstanceState: Bundle?) { super.onCreate(savedInstanceState) setContentView(R.layout.activity_po
    if (savedInstanceState != null) return
    val fragment = CreatePostFragment()
        supportFragmentManager
        .beginTransaction()
        .add(R.id. fragment_container, fragment)
        .commit()
}
```

- to make sure that the activity creates a new fragment each time it is restored from a previous state

10 View

10.1 What should you use to display a large, scrolling list of elements?

- Recycler View

10.2 Given the fragment below, how would you get access to a TextView with an ID of text_home contained in the layout file of a Fragment class?

```
    private lateinit var textView: TextView
    override fun onCreateView(...): View? {
        val root = inflator.inflator(R>layout.fragment_home, container, false)
        textView = ??
        return root
    }
// root.findViewById(R.id.text_home)
```

11 Intent

11.1 You have created a NextActivity class that relies on a string containing some data that pass inside the intent Which code snippet allows you to launch your activity?

```
Intent(this, NextActivity::class.java).apply {
    putExtra(EXTRA_NEXT, "some data")
}.also { intent ->
    startActivity(intent)
}
```

11.2 You have created an AboutActivity class that displays details about your app. Which code snippet allows you to launch your activity?

```
Intent(this, AboutActivity::class.java).also {
    intent -> startActivity(intent)
}
```

11.3 Which definition will prevent other apps from accessing your Activity class via an intent?

```
<activity android:name=".ExampleActivity" />
```

- Intent filters are used to make activities accessible to other apps using intents. So we have to choose option which have no intent filter to make sure it is not accessible by intent

11.4 You want to allow users to take pictures in your app. Which is not an advantage of creating an appropriate intent, instead of requesting the camera permission directly?

- Users can select their favorite photo apps to take pictures.
- You do not have to make a permission request in your app to take a picture.
- You do not have to design the UI. The app that handles the camera intent will provide the UI.
- You have full control over the user experience. The app that handles the camera intent will respect your design choices. (ANSWER)

11.5 onActivityResult()

- When will an activity's onActivityResult()be called?
  - when calling finish() in the target activity

11.6 startActivityWithResult(): You want to open the default Dialer app on a device. What is wrong with this code?

```
val dialerIntent = Intent()
val et = findViewById(R.id.some_edit_text)
dialerIntent.action = Intent.ACTION_DIAL
dialerIntent.data = Uri.parse("tel:" + et.getText()?.toString())
startActivity(dialerIntent) // <--
```

- startActivityWithResult() should be used instead of startActivity() when using Intent.ACTION_DIA

12 Data

12.1 storage

12.1.1 To persist a small collection of key-value data, what should you use?

- SharedPereferences

12.1.2 What allows you to properly restore a user's state when an activity is restarted?

- the onSaveInstance()method
- persistent storage
- ViewModel objects
- all of these answers (Refrence) (ANSWER)

12.1.3 To preserve on-device memory, how might you determine that the user's device has limited storage capabilities?

```
ActivityManager.isLowRamDevice() ;
```

- Use the ActivityManager.isLowRamDevice() method to find out whether a device defines itself as "low RAM."

12.2 How would you retrieve the value of a user's email from SharedPreferences while ensuring that the returned value is not null?

```
getDefaultSharedPreferances(this).getString(EMAIL,"")
```

- In Method "getDefaultSharedPrefarances(this).getString()" Second parameter is passed so that it can be returned, in case key doesn't exist. So we need to pass an empty string to be returned in case key doesn't exist.

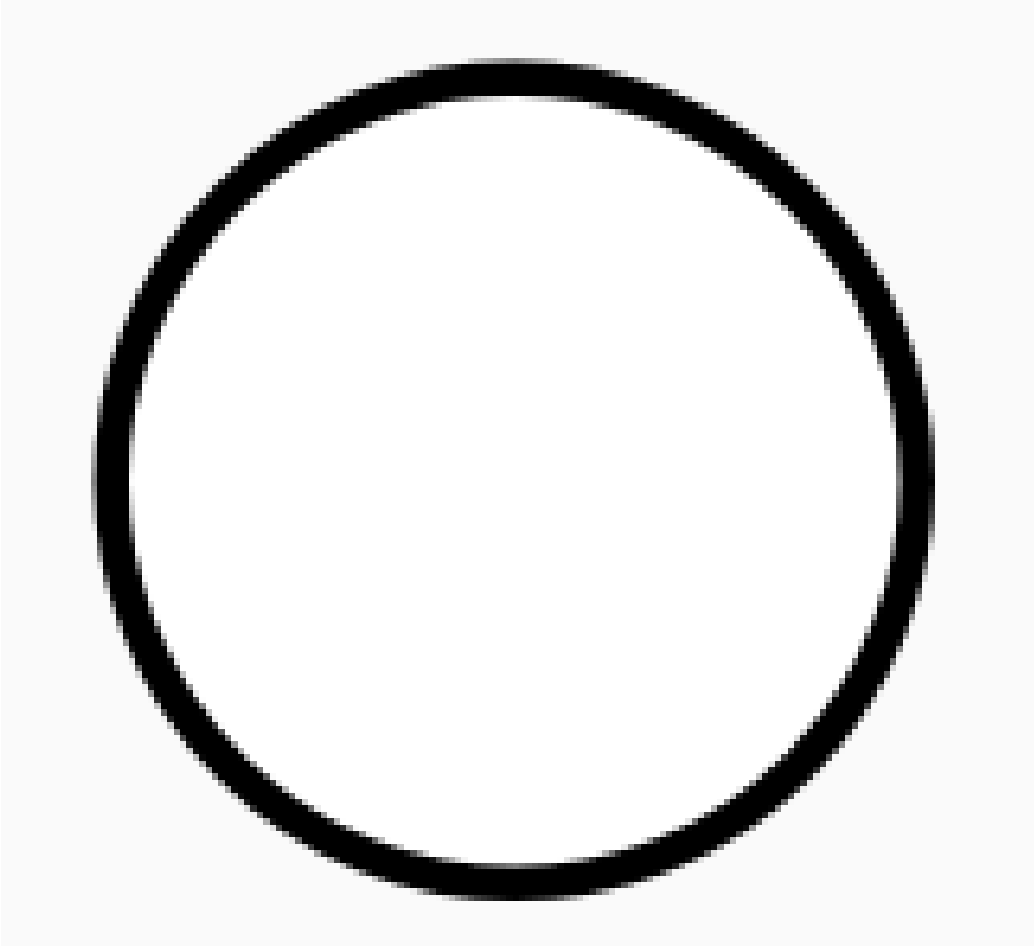
13 xml resource files

13.1 layout

13.1.1 Which layout is best for large, complex hierarchies?

- ConstraintLayout

13.1.2 Which drawable definition allows you to achieve the shape below?



```
<shape xmlns:android="http://schemas.android.com/apk/res/android"
android:shape="oval">
<stroke
    android:width="4dp"
    android:color="@android:color/black" />
<solid android:color="@android:color/white" />
</shape>
```

13.1.3 Which image best corresponds to the following LinearLayout?

```
  <LinearLayout
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:orientation="horizontal"
      android:gravity="center">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button" />
  </LinearLayout>
```

- gravity="center" 是描述的竖直方向上位于中间，而水平方向上同样也是位于中间，每个键的宽度由自身内容的宽度决定

- gravity: todo

13.1.4 Which code snippet would achieve the layout displayed below?

```
<androidx.constraintlayout.widget.ConstraintLayout
    ...>

    <TextView
        android:id="@+id/text_dashboard"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginStart="8dp"
        android:layout_marginEnd="8dp"
        android:textAlignment="center"
        android:text="Dashboard"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

- 左右各留 8dp, 文字中间对齐；文本框高度由自身高度决定；文本框宽度拉伸 match_parent;

13.1.5 实现矩形，右下角白色，左上角黑色的渐变效果

```
  <shape xmlns:android-"http://schemas.android.com/apk/res/android"
  android:shape-"rectangle">
  <gradient
      android:startColor-"@android:color/white"
      android:endColor-"@android:color/black"
      android:angle-"135"/>
    </shape>
```

13.1.6 Given the following dimens.xml file, how would you define an ImageView with medium spacing at the bottom?

```
<?xml version=1.0 encoding="utf-8"?>
<resources>
    <dimen name="spacing_medium">8dp</dimen>
    <dimen name="spacing_large">12dp</dimen>
</resources>

<ImageView
   android:id=@+id/image_map_pin"
   android:layout_width="wrap_content"
   android:layout_heignt="wrap_content"
   android:layout_marginBottom="@dimen/spacing_medium"
   android:src=@drawable/map_pin />
```

13.1.7 You want to provide a different drawable for devices that are in landscape mode and whose language is set to French. which directory is named correctly?

- drawable-fr-land

13.1.8 What folder should you use for your app's launcher icons?

- /mipmap

13.2 permissions

13.2.1 写外部存储权限

```
<uses-permission
     android:name="android.permission.WRITE_EXTERNAL_STORAGE"
     android:maxSdkVersion="18" />
```

13.2.2 When would you use the ActivityCompat.shouldShowRequestPermissionRationale() function?

```
ActivityCompat.shouldShowRequestPermissionRationale();
```

- when a user has previously denied the request for a given permission and selected "Don't ask again," but you need the permission for your app to function

13.2.3 Why might you need to include the following permission to your app?

```
android.permission.ACCESS_NETWORK_STATE
```

- to monitor the network state of the devices so that you don't attempt to make network calls when the network is unavailable

14 annotation

14.1 @VisibleForTesting:

- to denote that a class, methos, or field has its visibility relaxed to make code testable

15 apk

15.1 basic

15.1.1 When would you use a product flavour in your build setup?

when you want to provide different version of your app with custom configuration and resources

15.1.2 To shrink your code in release builds, what tool does Android Studio use?

- R8

15.2 build configuration

15.2.1 Which statement, in build.gradle file, correctly denotes that the corresponding module is an Android library module?

```
apply plugin: 'com.android.library'
```

15.2.2 You would like to enable analytics tracking only in release builds. How can you create a new field in the generated BuildConfig class to store that value?

```
buildTypes {
    debug {
        buildConfigField 'boolean', 'ENABLE_ANALYTICS', 'false'
    }
    release {
        buildConfigField 'boolean', 'ENABLE_ANALYTICS', 'true'
    }
}
```

15.2.3 To optimize your APK size, what image codec should you use?

- WebP (Reference)

15.2.4 Given an APK named app-internal-debug.apk produced from the build process, which statement is likely to be true?

- This APK is created from the debug build type and internal product flavor.

15.3 build errors

15.3.1 When attempting to build your project, what might the following error indicate?

```
Conversion to Dalvik format filed: Unable to execute dex: method ID not in [0, 0xffff]: 65536
```

- You have exceeded the total number of methods that can be referenced within a single DEX file.

16 testing

16.1 Why do you use the AndroidJUnitRunner when running UI tests?

- Notice: AndroidJUnitRunner lets us run JUnit3/4-style tests on Android Devices
  - The test runner facilitates loading your test package and the app under test onto a device or emulator, runs the test, and reports the results.

16.2 Given the test class below, which code snippet would be a correct assertion?

```
assertNotNull(resultAdd)
```

17 debugging

17.1 network

17.1.1 You have built code to make a network call and tested that it works in your development environment. However, when you publish it to the Play console, the networking call fails to work. What will NOT help you troubleshoot this issue?

- checking whether ProGuard -keepclassmembers have been added to the network data transfer objects (DTOs) in question
- checking for exceptions in the server logs or server console
- checking that the network data transfer object has @SerizlizedName applied to its member properties
- using the profiler tools in Android Studio to detect anomalies in CPU, memory, and network usage (ANSWER: this does not help)

18 Frameworks

18.1 Retrofit

18.1.1 You need to remove an Event based on it;s id from your API, Which code snippet defines that request in Retrofit?

- @DELETE("events/{id}") fun deleteEvent(@Path("id") id: Long): Call

18.1.2 You need to retrieve a list of photos from an API. Which code snippet defines an HTML GET request in Retrofit?

```
@GET("photo") fun listPhotos() : Call<List>
```

19 需要分类出去的

19.0.1 What are the permission protection levels in Android?

- Normal —A lower-risk permission that gives requesting applications access to isolated application-level features, with minimal risk to other applications, the system, or the user. The system automatically grants this type of permission to a requesting application at installation, without asking for the user’s explicit approval.
- Dangerous —A higher-risk permission. Any dangerous permissions requested by an application may be displayed to the user and require confirmation before proceeding, or some other approach may be taken to avoid the user automatically allowing the use of such facilities.
- Signature —A permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. If the certificates match, the system automatically grants the permission without notifying the user or asking for the user’s explicit approval.
- SignatureOrSystem —A permission that the system grants only to applications that are in the Android system image or that are signed with the same certificate as the application that declared the permission.

19.0.2 What is Android Data Binding?

The Data Binding Library is a support library that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

Layouts are often defined in activities with code that calls UI framework methods. For example, the code below calls findViewById() to find a TextView widget and bind it to the userName property of the viewModel variable:

```
TextView textView = findViewById(R.id.sample_text);
textView.setText(viewModel.getUserName());
```

The following example shows how to use the Data Binding Library to assign text to the widget directly in the layout file. This removes the need to call any of the Java code shown above.

```
<TextView
    android:text="@{viewmodel.userName}" />
```

- The pros of using Android Data Binding:
  - Reduces boilerplate code which in turns brings
  - Less coupling
  - Stronger readability
  - Powerful, easy to implement custom attribute and custom view
  - Even faster than findViewById - The binding does a single pass on the View hierarchy, extracting the Views with IDs. This mechanism can be faster than calling findViewById for several Views.

19.0.3 What is the ViewHolder pattern? Why should we use it?

Every time when the adapter calls getView() method, the findViewById() method is also called. This is a very intensive work for the mobile CPU and so affects the performance of the application and the battery consumption increases. ViewHolder is a design pattern which can be applied as a way around repeated use of findViewById().

A ViewHolder holds the reference to the id of the view resource and calls to the resource will not be required after you "find" them: Thus performance of the application increases.

```
private static class ViewHolder {
    final TextView text;
    final TextView timestamp;
    final ImageView icon;
    final ProgressBar progress;

    ViewHolder(TextView text, TextView timestamp, ImageView icon, ProgressBar progress) {
        this.text = text;
        this.timestamp = timestamp;
        this.icon = icon;
        this.progress = progress;
    }
}
public View getView(int position, View convertView, ViewGroup parent) {
    View view = convertView;
    if (view == null) {
        view = // inflate new view
        ViewHolder holder = createViewHolderFrom(view);
        view.setTag(holder);
    }
    ViewHolder holder = view.getTag();
    // TODO: set correct data for this list item
    // holder.icon.setImageDrawable(...)
    // holder.text.setText(...)
    // holder.timestamp.setText(...)
    // holder.progress.setProgress(...)
    return view;
}
private ViewHolder createViewHolderFrom(View view) {
    ImageView icon = (ImageView) view.findViewById(R.id.listitem_image);
    TextView text = (TextView) view.findViewById(R.id.listitem_text);
    TextView timestamp = (TextView) view.findViewById(R.id.listitem_timestamp);
    ProgressBar progress = (ProgressBar) view.findViewById(R.id.progress_spinner);
    return new ViewHolder(text, timestamp, icon, progress);
}
```

- View.setTag(Object) allows you to tell the View to hold an arbitrary object. If we use it to hold an instance of our ViewHolder after we do our findViewById(int) calls, then we can use View.getTag() on recycled views to avoid having to make the calls again and again.

19.0.4 What is the difference between Handler vs AsyncTask vs Thread?

Mid Top 113 Android Interview Questions Android 113 Answer The Handler class can be used to register to a thread and provides a simple channel to send data to this thread. A Handler

allows you communicate back with the UI thread from other background thread. The AsyncTask class encapsulates the creation of a background process and the synchronization with the main thread. It also supports reporting progress of the running tasks. And a Thread is basically the core element of multithreading which a developer can use with the following disadvantage: Handle synchronization with the main thread if you post back results to the user interface No default for canceling the thread No default thread pooling No default for handling configuration changes in Android Having Tech or Coding Interview? Check 113 Android Interview Questions Source: stackoverflow.com

19.0.5 What is the difference between compileSdkVersion and targetSdkVersion?

Mid Top 113 Android Interview Questions Android 113 Answer The compileSdkVersion is the version of the API the app is compiled against. This means you can use Android API features included in that version of the API (as well as all previous versions, obviously). If you try and use API 16 features but set compileSdkVersion to 15, you will get a compilation error. If you set compileSdkVersion to 16 you can still run the app on a API 15 device as long as your app's execution paths do not attempt to invoke any APIs specific to API 16.

The targetSdkVersion has nothing to do with how your app is compiled or what APIs you can utilize. The targetSdkVersion is supposed to indicate that you have tested your app on (presumably up to and including) the version you specify. This is more like a certification or sign off you are giving the Android OS as a hint to how it should handle your app in terms of OS features.

19.0.6 What is the difference between a Bundle and an Intent?

Mid Top 113 Android Interview Questions Android 113 Answer A Bundle is a collection of key-value pairs. However, an Intent is much more. It contains information about an operation that should be performed. This new operation is defined by the action it can be used for, and the data it should show/edit/add. The system uses this information for finding a suitable app component (activity/broadcast/service) for the requested action. Think of the Intent as a Bundle that also contains information on who should receive the contained data, and how it should be presented.

19.0.7 What are the wake locks available in android?

A - PARTIAL_WAKE_LOCK B - SCREEN_DIM_WAKE_LOCK C - SCREEN_BRIGHT_WAKE_LOCK D - FULL_WAKE_LOCK E - FULL_WAKE_LOCK Answer : E Explanation When CPU is on mode, PARTIAL_WAKE_LOCK will be active.

When CPU + bright Screen low is on mode, SCREEN_DIM_WAKE_LOCK will be active.

When CPU + bright Screen High is on mode,SCREEN_BRIGHT_WAKE_LOCK will be active.

When CPU, Screen, bright Screen High is on mode, FULL_WAKE_LOCK will be active.

20 项目中用到的小点

20.1 api level 28, androidx 之前的最后一个版本

20.2 android api level 30 androidx 中项目一定需要修改的条款: androidx.fragme 还没有弄通

20.2.1 gradles.propertiess

20.2.2 project build.gragle: gragle versions

20.2.3 app module build.gradle

- 什么是 Jetifier？例如，要使用 androidx 打包的依赖项创建新项目，此新项目需要在 gradle.properties 文件中添加以下行：

20.2.4 JavaVersion.VERSION_1_8

```
java version 8
 compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
```

20.2.5 references

```
import android.content.Context;
import android.os.Bundle;
import android.util.Log;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.view.Menu;
import android.view.MenuItem;
import androidx.fragment.app.Fragment;
import androidx.annotation.Nullable;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import android.support.design.widget.FloatingActionButton;
import android.support.design.widget.Snackbar;
import com.google.android.material.floatingactionbutton.FloatingActionButton;
import com.google.android.material.snackbar.Snackbar;
```

20.2.6 xml 中也还有一些注意事项

```
<com.me.generalprac.CustomTitleView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"/>
<include layout="@layout/custom_title"/>
```