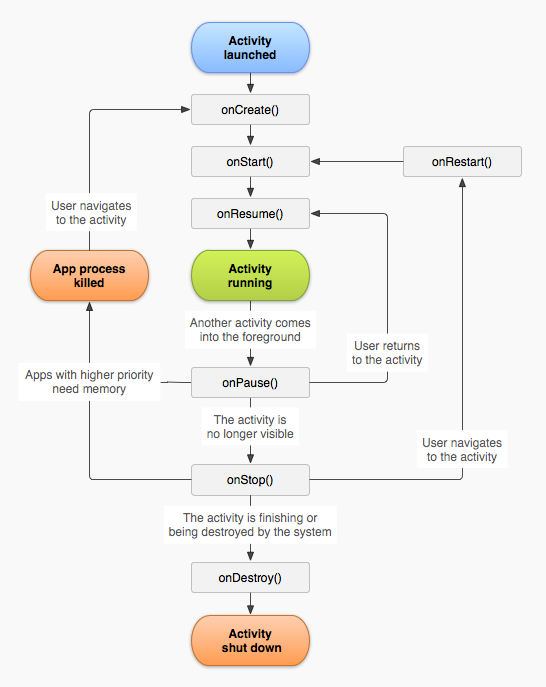
**Android面试题整理**

1. Activity
2. 生命周期



1. 切换屏幕

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

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

1. 定义成对话框模式

第一种方法：<activity android:name=".activity2" android:theme="@android:style/Theme.Dialog"></activity>

第二种方法：

<style name="dialog" parent="@android:style/Theme.Dialog">

<item name="android:windowFrame">@null</item>

<item name="android:windowIsFloating">true</item>

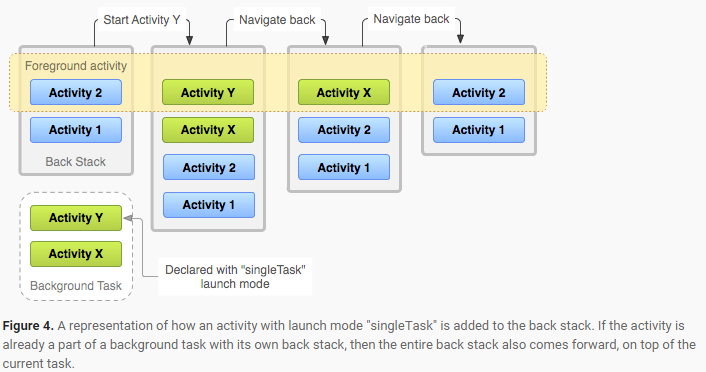
<item name="android:windowIsTranslucent">true</item>

<item name="android:windowNoTitle">true</item>

<itemname="android:windowBackground">@android:color/transparent</item>

</style>

1. 数据存储
2. 一般是在onSaveInstanceState中保存activity的数据，然后在Oncreate或者[onRestoreInstanceState(Bundle)](http://developer.android.com/reference/android/app/Activity.html#onRestoreInstanceState(android.os.Bundle))中获取保存的数据，使用这种方法保存数据的前提是这个Activity必须有机会能够被恢复，假如A启动了B，然后从B返回到A，就可以在A中采用这个方法保存A的运行状态，但是B就不行，因为B没有机会重新加载，因为再次在A中打开B都是一个新的Task。这个方法在onStop()方法之前调用，但是不能确定是在onPause()之前或者之后调用。
3. 在onPause()中保存Activity的状态和其他数据是google官方推荐的方式，因为onPause()之后是onResume(),或者onStop(),如果是onStop()，那么a的存储方式就可行，但是如果是onResume()，那么onSaveInstanceState就不会执行，导致数据丢失。
4. 综上所述，a适合保存视图的状态，b适合保存持久的数据，包括保存关键的数据到数据库等，因为在a中不能保证onSaveInstanceState每次都运行。
5. 退出多个Activity
6. 使用一个容器记住每一个打开的activity，并将其假如到容器中，在每一个activity的onDestory()方法中移除这个activity的实例，在最后退出应用的时候遍历容器并将每一个activity结束掉。
7. 使用广播，在每一个打开的activity中注册一个广播，在程序要退出的时候发送结束的广播，此时每一个打开的activity会自行关闭。
8. 加载模式(LaunchMode)
9. Standard ,默认模式，在一个任务里面为每一个activity创建一个实例，一个activity能够被实例化多次，每一个实例可以属于多个任务，一个任务可以有多个实例。
10. Singletop，单顶模式([FLAG\_ACTIVITY\_SINGLE\_TOP](http://developer.android.com/reference/android/content/Intent.html#FLAG_ACTIVITY_SINGLE_TOP))，即A B C D，此时要添加D，但是在原来的task中D的加载模式为单顶模式，那么原Task中的D就会执行onNewIntent()方法，不会重新创建一个D的实例，而是变成A B C D，但是如果原Task中D的加载模式是默认模式的话就会变成A B C D D。
11. SingleTask,单任务模式([FLAG\_ACTIVITY\_NEW\_TASK](http://developer.android.com/reference/android/content/Intent.html#FLAG_ACTIVITY_NEW_TASK))，例如activity A打开浏览器，之后退出浏览器，此时activity B打开浏览器，这个浏览器在A中打开时相当于SingleTask模式，在B中打开时相当于返回之前已经打开的实例，一个典型的应用关系图如下：



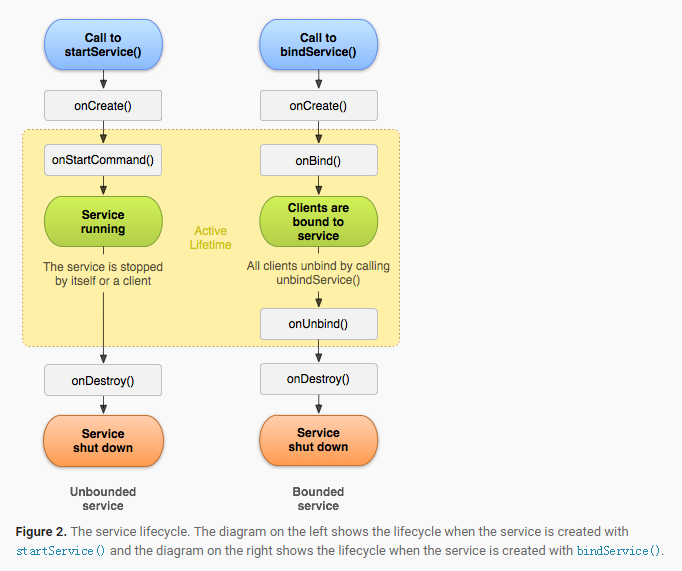
1. singleInstance,单例模式，即在该task中只有一个Activity的实例，任何从该task启动的activity都在另外一个task中。
2. [FLAG\_ACTIVITY\_CLEAR\_TOP](http://developer.android.com/reference/android/content/Intent.html#FLAG_ACTIVITY_CLEAR_TOP),清除该activity顶层的activity，意思是清除在这个实例上面的所有实例，然后使用这个模式的activity进入resume()的状态。

<http://developer.android.com/guide/components/tasks-and-back-stack.html>

<http://developer.android.com/guide/components/activities.html#SavingActivityState>

<http://stackoverflow.com/questions/151777/saving-activity-state-in-android>

<http://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle)>

1. Service
2. 生命周期
3. 启动方式
4. 在activity中startService,如果该Service是第一次创建，那么所经历的生命周期是；onCreate() onStartCommand() onDestory()

如果该Service已经启动，再次启动时的生命周期是：

onStartCommand() onDestory()

1. bindService,绑定Service，如果是第一次绑定这个Service，那么其生命周期是：

onCreate() onBind() onUnbind onDestory

如果该Service已经启动，再次绑定时的生命周期是：

onBind() onUnbind onDestory

1. 在Manifest中注册Service

<manifest ... >  
  ...  
  <application ... >  
      <service android:name=".YourService" />  
      ...  
  </application>  
</manifest>

<http://developer.android.com/reference/android/app/IntentService.html>

<http://developer.android.com/guide/components/services.html>

1. IntentService
2. intentService运行在异步线程中，通过[startService(Intent)](http://developer.android.com/reference/android/content/Context.html#startService(android.content.Intent)) 启动IntentService,其会按照顺序在非主线程中执行每一个请求，并且在工作完成之后自行关闭结束。
3. 工作队列的方式让任务运行于主线程之外，通过继承IntentService,并且重写onHandleIntent0(Intent)方法来使用他，之后其会开启一个工作线程，并且在任务完成之后停止这个Service。
4. 所有的请求会在一个工作线程里面被处理，并且不会阻塞主线程，但是一个请求只会在一个时间点被处理。
5. 提供默认实现onBind()方法，并且返回空
6. 提供默认实现onStartCommand()方法，发送请求的intent到工作线程，之后实现onHandleIntent()方法。
7. 实现关键代码如下：

**public** **abstract** **class** IntentService **extends** Service {

**private** **volatile** Looper mServiceLooper;

**private** **volatile** ServiceHandler mServiceHandler;

**private** String mName;

**private** **boolean** mRedelivery;

**private** **final** **class** ServiceHandler **extends** Handler {//定义一个Handler

**public** ServiceHandler(Looper looper) {

**super**(looper);

}

@Override

**public** **void** handleMessage(Message msg) {

onHandleIntent((Intent)msg.obj);

stopSelf(msg.arg1);

}

}

@Override

**public** **void** onCreate() {

**super**.onCreate();

HandlerThread thread = **new** HandlerThread("IntentService[" + mName + "]");

thread.start();//开始线程

mServiceLooper = thread.getLooper();

mServiceHandler = **new** ServiceHandler(mServiceLooper);

}

@Override

**public** **void** onStart(Intent intent, **int** startId) {

Message msg = mServiceHandler.obtainMessage();

msg.arg1 = startId;

msg.obj = intent;

mServiceHandler.sendMessage(msg);//发送消息

}

@Override

**public** **int** onStartCommand(Intent intent, **int** flags, **int** startId) {

onStart(intent, startId);//调用onStart方法

**return** mRedelivery ? *START\_REDELIVER\_INTENT* : *START\_NOT\_STICKY*;

}

}

1. onStartCommand返回类型

[START\_NOT\_STICKY](http://developer.android.com/reference/android/app/Service.html#START_NOT_STICKY)

如果在onStartCommand返回之后关闭这个Service，其不会重新创建这个Service，除非有intent传递过来。这种返回类型是最安全的模式避免运行

不必要的Service并且能够重启未完成的工作。

[START\_STICKY](http://developer.android.com/reference/android/app/Service.html#START_STICKY)

如果在onStartCommand返回之后关闭这个Service，重新创建这个Service并且调用onStartCommand，但是不会重新传递最后的那个Intent，因为系统会调用onStartCommand，并且传递一个空的intent，除非有intent等着传递过来开始这个Service。这种返回类型适合媒体播放器获其他相似的Service，不会运行指令，但是能够循环运行并等待一个任务。

[START\_REDELIVER\_INTENT](http://developer.android.com/reference/android/app/Service.html#START_REDELIVER_INTENT)

如果在onStartCommand返回之后关闭这个Service，会重新创建Service，并且会调用onStartCommand，并将传递最后的那个intent，其他任何等待传递的intent将会排队等候。这种返回模式适合那些活跃的任务需要立即启动，比如下载一个文件。

1. 运行前台Service

请求运行一个Service在前台可以调用startForeground(),这个方法有两个参数，一个整型是独有的用来辨识notification,一个是显示状态的通知栏对象。

1. Broadcast Receiver(广播)
2. 发送广播方式
3. SendBroadcast()，常规发送广播的方式，这种方式发送广播方式是异步的，基本上没有设置优先级的接受体都是在同一时间接收到广播，这种方式更有效，但是这种方式下接受体无法使用结果并且不能停止广播的传递。
4. [sendOrderedBroadcast](http://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent, java.lang.String))，发送有序广播，在一个时间发送一个广播到接受体，接着往下传递，其传递顺序取决于广播注册时设定的优先级，优先级高的先接受广播，再往优先级低的广播传递，如果接受体优先级相同，则接受顺序是随意的。优先级可以通过android.priority在intent-filter中设定，也可以在传递过程中abort广播的传递。
5. 注册方式
6. 动态注册

[Context.registerReceiver()](http://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter))

1. 静态注册

在AndroidManifest.xml中注册，注册之后跟随程序的生命周期。

注意：在onresume()中注册广播时要在onpause()中注销，因为在onpause状态无法接受广播，这样可以避免系统中不必要的重载。不要在[onSaveInstanceState()](http://developer.android.com/reference/android/app/Activity.html#onSaveInstanceState(android.os.Bundle))中注销广播，因为有可能这个方法不会执行。广播传递过程中的intent不能用于启动一个Activity，因为广播传递intent的时候是在后台，而启动activity的intent过程是在前台进行的。

1. 安全
2. 当在一个程序的manifest定义一个接受体并且指定intent-filter的时候，其他的程序也会也会发送广播到这个接受体，而不受intent-filter的限定。为了避免这种情况应该在定义接受体的时候使用标签android:exported=”false”。
3. 当使用sendbroadcast()或者相关方法的时候，一般情况下别的程序也会接收到这个广播，我们可以控制谁可以接收到这个广播，从三明治系统开始，我们可以发送广播的时候定义Intent.setPackage
4. 可以强制发送的时候设置权限，通过设置sendBroadcast(Intent,String)或者sendOrderedBroadcast(Intent,String,BroadcastReceiver,Handler,int,String Bundle)的参数不为空来实现，在接受方只有在androidmanifest中的<uses-permission>设置了相关标签才会接受到相关广播。
5. 可以强制设置接受的权限，通过在注册广播的时候设置registerReceiver(BroadcastReceiver , intentFilter,String,handler)的参数不为空或者在androidmanifest的<receiver>标签中设置。只有在<users-permission>中设置了对应权限的广播才会被发送到接受体。
6. 生命周期

onReceive被认为是一个前台线程，并且会在系统里面保持运行，除非系统资源紧张，一旦onreceive返回，那么这个BroadcastReceiver就会被认为不再活跃，并且很有可能被杀死，然后释放资源给其他进程。并且在这个方法里面不能运行异步任务，因为广播有可能在任务没有完成之前关闭，但是可以在这个方法里面startService开启Service或者使用[NotificationManager](http://developer.android.com/reference/android/app/NotificationManager.html) 接口打开对话框。

<http://developer.android.com/reference/android/content/BroadcastReceiver.html>

1. Content Provider(内容提供者)
2. 操作ContentProvider的数据

通过获取ContentResolver对象来操作ContentProvider的数据，主要包括增删改查操作。

查找操作如下：

mCursor = getContentResolver().query(  
UserDictionary.Words.CONTENT\_URI,   //相当于数据库的表名

mProjection,                        //返回每一列的名称，相当于表的列名mSelectionClause                    //被选择的每一条数据where col=value

mSelectionArgs,                     //初始化被选择的每一条数据mSortOrder);                        //根据该参数排序每一行的数据order by

1. 识别内容提供者（URI）

URI是由scheme + authority + path组成.

1. 使用查找到的数据

int index = mCursor.getColumnIndex(UserDictionary.Words.WORD);

if (mCursor != null) {

while (mCursor.moveToNext()) {

newWord = mCursor.getString(index);

}

1. 插入数据

Uri mNewUri;

ContentValues mNewValues = new ContentValues();

mNewValues.put(UserDictionary.Words.APP\_ID, "example.user");  
mNewValues.put(UserDictionary.Words.LOCALE, "en\_US");  
mNewValues.put(UserDictionary.Words.WORD, "insert");  
mNewValues.put(UserDictionary.Words.FREQUENCY, "100");

mNewUri = getContentResolver().insert(  
    UserDictionary.Word.CONTENT\_URI,   // the user dictionary content URI  
    mNewValues                          // the values to insert  
 );

1. 更新数据

ContentValues mUpdateValues = new ContentValues();

String mSelectionClause = UserDictionary.Words.LOCALE +  "LIKE ?";  
String[] mSelectionArgs = {"en\_%"};

int mRowsUpdated = 0;

ateValues.putNull(UserDictionary.Words.LOCALE);

mUpd mRowsUpdated = getContentResolver().update(  
    UserDictionary.Words.CONTENT\_URI,   // the user dictionary content URI  
    mUpdateValues                       // the columns to update  
    mSelectionClause                    // the column to select on  
    mSelectionArgs                      // the value to compare to  
);

1. 删除数据

String mSelectionClause = UserDictionary.Words.APP\_ID + " LIKE ?";  
String[] mSelectionArgs = {"user"};

int mRowsDeleted = 0;

mRowsDeleted = getContentResolver().delete(  
    UserDictionary.Words.CONTENT\_URI,   // the user dictionary content URI  
    mSelectionClause                    // the column to select on  
    mSelectionArgs                      // the value to compare to  
 );

1. 支持的数据类型

Int ,long,double,float,blob(binary large object)