手机应用平台软件开发

Intent与Activity

Android Framework构成

- 四大元素
  1. Activities-管理应用程序展示
  2. Services-管理后台服务
  3. Broadcast receivers
  4. Content Provider-管理数据共享
- 沟通的桥梁: **Intent**

活动（Activity）

- 在Android应用程序里，一个Activity就是一个用户界面。用户与程序的交互就是通过该类来实现的。
- 主Activity是程序启动的入口。应用程序成功启动之后，呈献给用户的第一个界面，即为该程序的主Activity。

活动（Activity） 主要特征

主要特征

- 包括UI,以及与user的互动
- 可以放button、list、picture、text等组件
- UI可动态调整 (增加、减少、换位置)
- 利用intent跳转到其他activity
- 超过5秒,会出现ANR (**Android is Not Responding)**
- **Activity是Android程序与用户交互的窗口，可以看成网站的页面。**

Activity的状态变化

- 每个活动（Activity）都处于某个状态。对于开发者来说，是无法控制其应用程序处于某个状态的，这些均由系统来完成。但当一个活动状态发生改变的时，开发者可以通过调用onXXX()方法，获取相关的通知信息。
- **第一次启动MainActivity**：依次执行以下方法：onCreate()→MainActivity created→onStart()→MainActivity started→onResume()→MainActivity actived，进入活动状态。

结束Activity

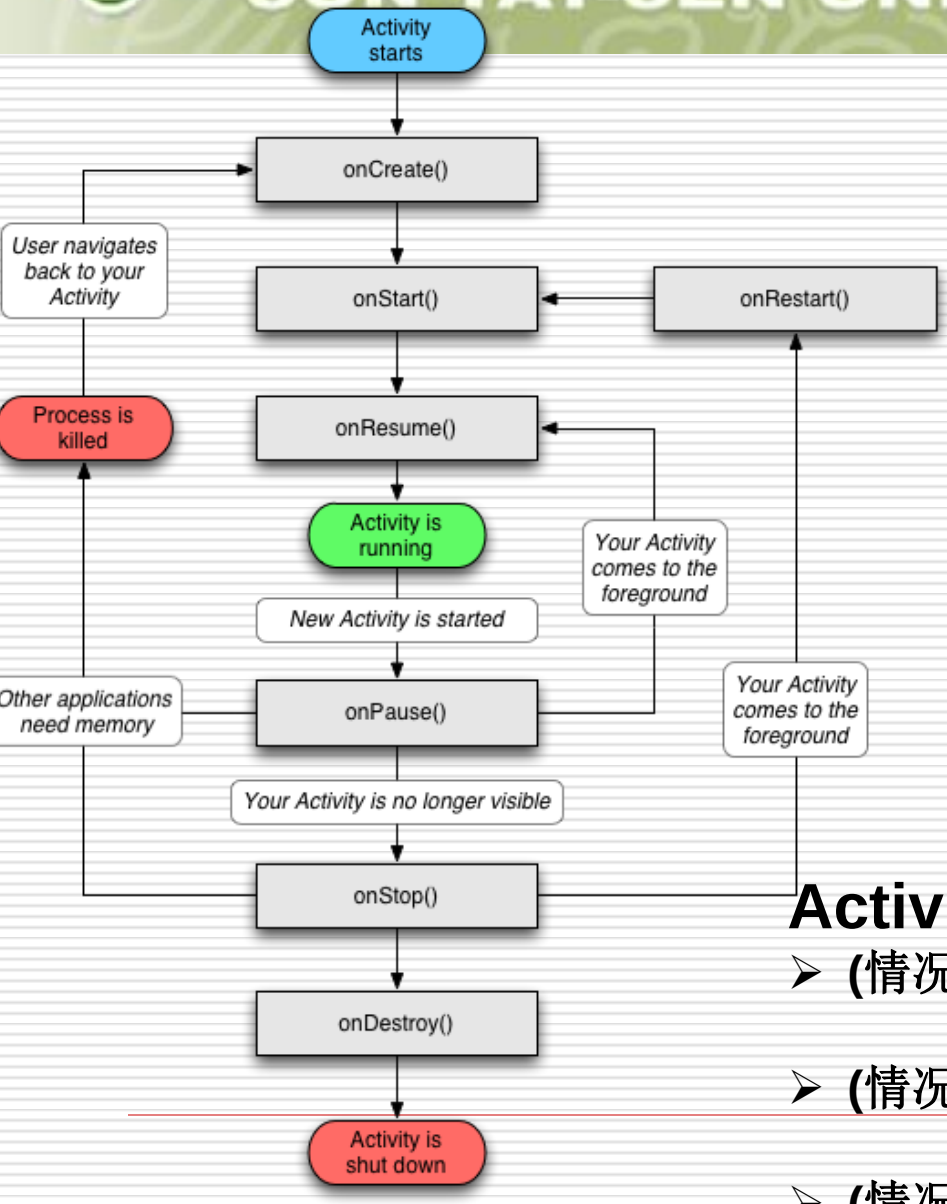
结束**MainActivity**：依次执行以下方法：onPause ()→MainActivity paused→ onStop ()→MainActivity stoped→ onDestory() → MainActivity killed。

1. Activity由活动状态转为暂停状态。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，所以它仍然可见。但已失去了焦点，故不可与用户交互。在极特殊的情况下，Android将会杀死一个暂停的Activity，来为活动的Activity提供充足的资源；
2. Activity被停止，变为完全隐藏，失去焦点，并且不可见。但是系统将仍在内存中保存它所有的状态和信息；
3. Activity被杀死，转为销毁状态。Activity结束，退出当前应用程序

Activity--用户界面屏幕

应用程序可以定义一个或者多个活动，以处理程序不同阶段的任务。作为应用程序生命周期的一部分，每个活动都要保存自己的状态，以便日后还原这些状态。这些活动会被系统的活动管理器记录在**应用程序栈**（**application stack**）中。用户可随时按Back按钮返回到栈中的上一个窗口。从用户的角度看，该工作方式类似于Web浏览器中的历史功能，即按Back返回到上一个页面。

Activity生命周期



若内存紧张，系统会直接结束当前的Activity，而不会触发onStop()方法，没有在前台运行的都会被停止。因此，重要数据都写在onPause()里面，最后一个安全生命周期方法。

Activity的销毁顺序:

- **(情况一)**onPause()—><ProcessKilled>
- **(情况二)**onPause()—>onStop()—><ProcessKilled>
- **(情况三)**onPause()—>onStop()—>onDestroy()

四种状态

Android 的虚拟机(VM)是使用基于栈(Stack based) 管理，主要有四种状态：

1. Active (活动)
2. Paused (暂停)
3. Stopped (停止)
4. Dead (已回收或未启动)

Active (活动)状态

- 使用者启动应用程序或Activity 后，Activity 运行中的状态。
- 在Android 平台上，**同一个时刻只会有一个Activity 处于活动(Active)或运行(Running)状态**。其他的Activity 都处于未启动(Dead)、停止(Stopped)、或是暂停(Pause)的状态。

Paused (暂停)状态

- **当Activity 暂时暗下来，退到背景画面的状态**。
- 当使用Toast、AlertDialog或电话呼入时，都会让原本运行的Activity 退到背景画面。新出现的Toast、AlertDialog 等界面元件盖住了原来的Activity 画面。Activity 处在"Paused"状态时，使用者无法与原Activity互动。

Stopped (停止)状态

- **Activity完全被另一个Activity所覆盖，则其状态为Stopped**
- 有其他Activity 正在执行，而当前Activity 已经离开屏幕，不再动作的状态。通过长按"Home"钮，可以叫出所有处于"Stopped"状态的应用程序列表。在“Stopped”状态的Activity，还可以通过“Notification”来唤醒。

Dead (已回收或未启动)状态

- **Activity** 尚未被启动、已经被手动终止，或已经被系统回收的状态。
- 要手动终止Activity，可以在程序中调用“finish”函数，若是被系统回收，可能是因为内存不足，因此系统根据内存不足时的回收规则，将处于“Stopped”状态的Activity 所占用的内存回收。

Activity 生命周期

- **onCreate()**

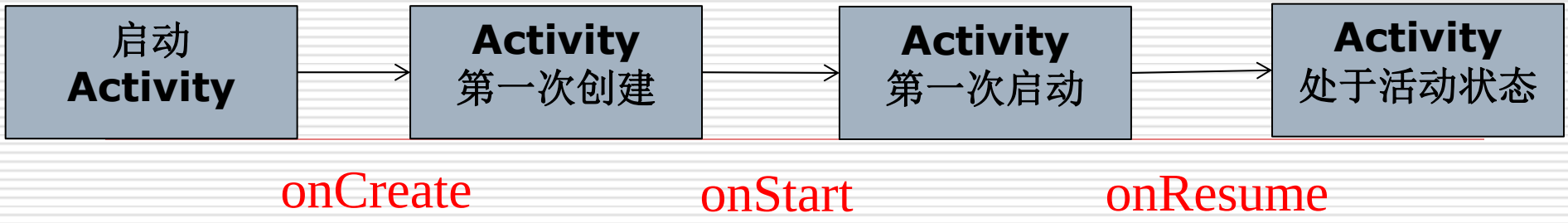
  当活动第一次启动时触发该方法，可以在此时完成活动的初始化工作。onCreate 方法有一个参数，该参数可以为空(null)，也可以是之前调onSaveInstanceState ()方法保存的状态信息。

- **onStart()**

  该方法的触发表示所属活动将被展现给用户。

- **onResume()**

  当一个活动和用户发生交互的时候，触发该方法



Activity 生命周期

- **onPause()**

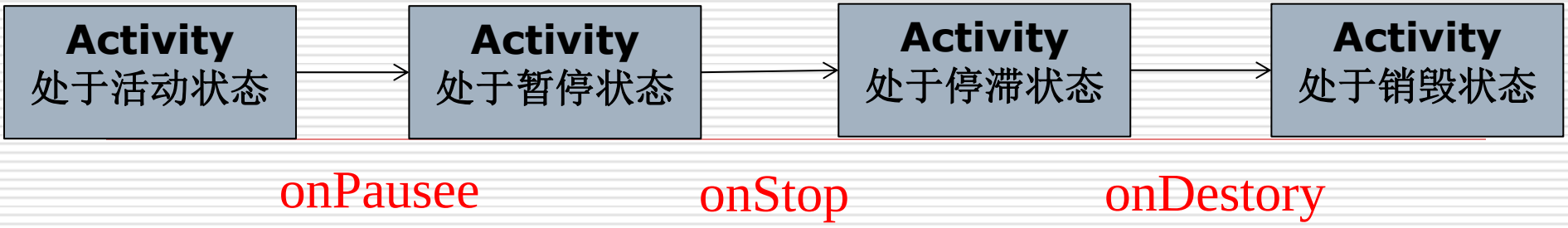
  当一个正在前台运行的活动因为其他的活动需要前台运行而转入后台运行的时候，触发该方法。这时候需要将活动的状态持久化，比如正在编辑的数据库记录等

- **onStop()**

  当一个活动不再需要展示给用户的时候，触发该方法。如果内存紧张，系统会直接结束这个活动，而不会触发onStop 方法。所以保存状态信息是应该在onPause时做，而不是onStop时做.

- **onDestroy()**

  当一个活动不再需要展示给用户的时候，触发该方法。如果内存紧张，系统会直接结束这个活动，而不会触发onStop 方法。所以保存状态信息是应该在onPause时做，而不是onStop时做



Activity 生命周期

- **onRestart()**

  当处于停止状态的活动需要再次展现给用户的时候，触发该方法

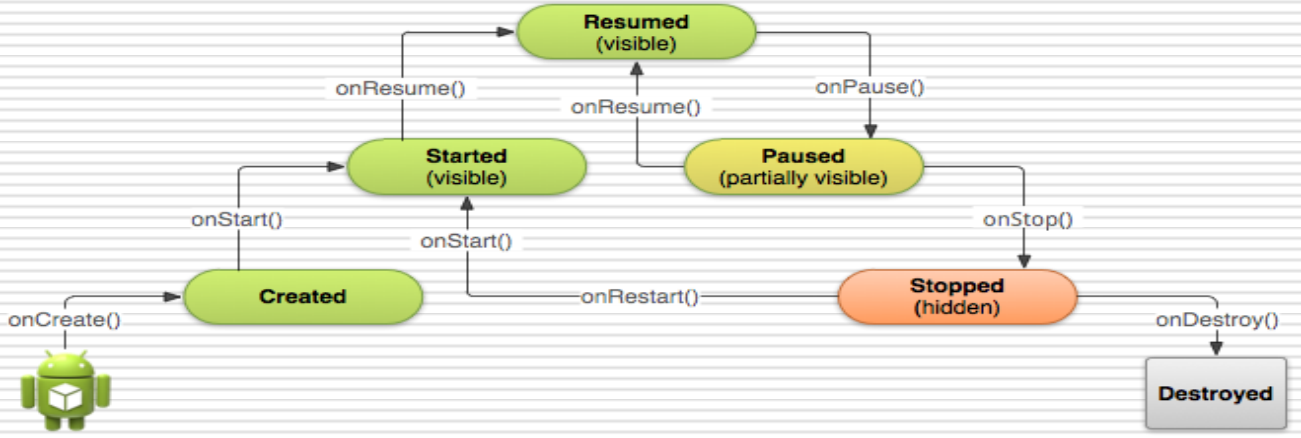
- **onSaveInstanceState**

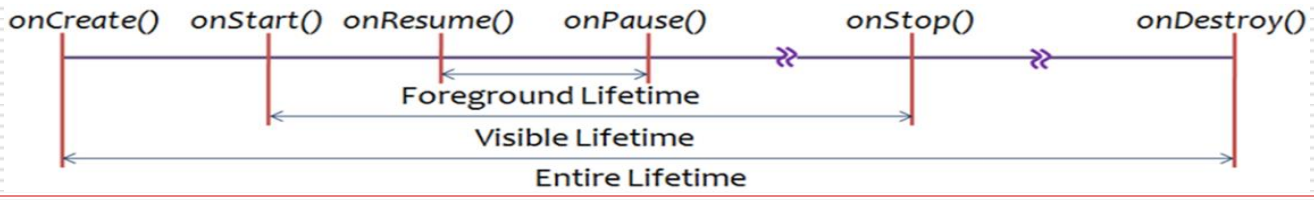
  系统调用该方法，允许活动保存之前的状态，比如说在一串字符串中的光标所处的位置等。当活动销毁的时候，触发该方法。和onStop 方法一样，如果内存紧张，系统会直接结束这个活动而不会触发该方法

Activity完整生命周期

onCreate()/ onDestroy()

完整的Activity 生命周期由"Create"状态开始，由"Destroy"状态结束。建立(Create)时分配资源，销毁(Destroy)时释放资源。

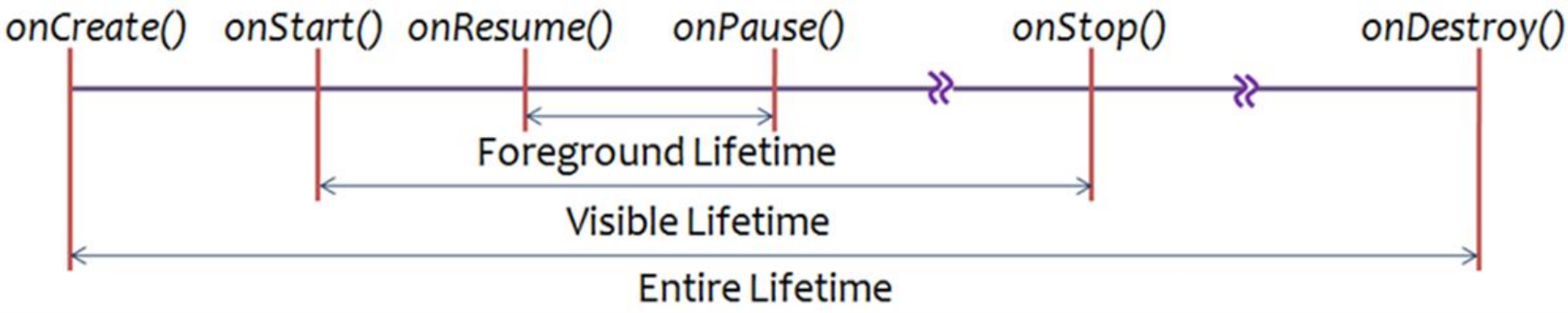




Activity可视生命周期

onStart()/onStop()

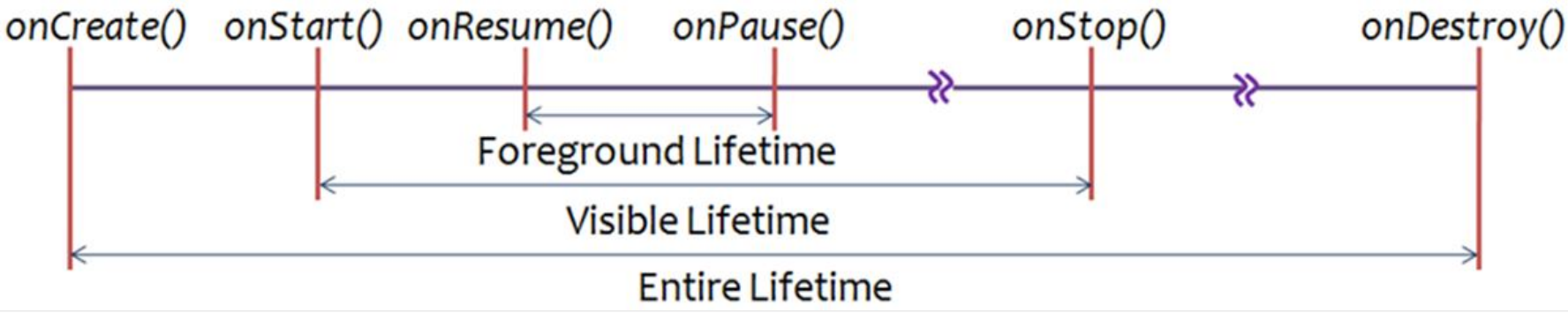
当Activity 运行到“Start”状态时，就可以在屏幕上看到该Activity。相反当Activity 运行到“Stop”状态时，此Activity 就会从屏幕上消失。当使用者按下Back 按钮回到上一个Activity 时，会先到Restart 状态，再到一般的Start 状态。



Activity前台生命周期

onResume()/onPause()

使用者能否直接存取屏幕（Resume/Pause），当有个Toast、AlertDialog、短信、电话等信息发生时，原来的Activity 会处于“Pause”状态，暂时放弃直接存取屏幕的能力，被中断到背景去，将前景交给优先级高的事件。当这些优先级高的事件处理完后，Activity 就改进入"Resume"状态，此时又直接存取屏幕。



多个Activity之间的关系--Intent

意图（Intent）

- Intent是一种运行时绑定机制，能在应用程序运行的过程中连接两个不同的组件，实现组件间的跳转。
- Activity、Service和BroadcastReceiver，都是通过Intent机制激活的，而不同类型的组件在传递Intent时有不同的方式。

意图（Intent）

- An intent is an abstract description of an operation to be performed:一个Intent就是一次对将要执行的操作的抽象描述。
- **Intent 数据结构两个最重要的部分是：**
  - 动作：典型的动作类型有：MAIN（活动的门户）、VIEW、PICK、EDIT等。
  - 动作对应的数据：**以URI 的形式进行表示**

    例如:要查看某个人的联系方式，需要创建一个动作类型为VIEW 的Intent，以及一个表示这个人的URI。

意图（Intent）

- Android 使用了Intent 这个特殊类，实现在屏幕与屏幕之间跳转。Intent 类用于描述一个应用将会做什么事。
- **目前有三种intent**

1. 启动一个新的activity，并可以携带数据；
2. 通过一个intent来启动一个服务；
3. 通过intent来广播一个事件；

Intent样例（页面跳转）


```
import android.content.Intent;
    Intent intent = new Intent(Startup.this, MainList.class);
    startActivity(intent);
```

Intent显式调用，页面就从Startup.java跳转到MainList.java了。

Intent

- 每个**intent**都带有动作（**action**），并根据不同动作去行动。

```
public void onClick()
    { Uri uri = Uri.parse("http://www.sysu.edu.cn/");
    Intent intent = new Intent(Intent.ACTION_VIEW, uri);
    startActivity(intent);  }
    Listener =new OnClickListener(){
        Public void onClick(View v)
        {
            setTitle(i° Test Intenti±);
            Intent int=new Intent(ActivityMain.this,Activity2.class);
    startActivity(intent)
            }
    }
```

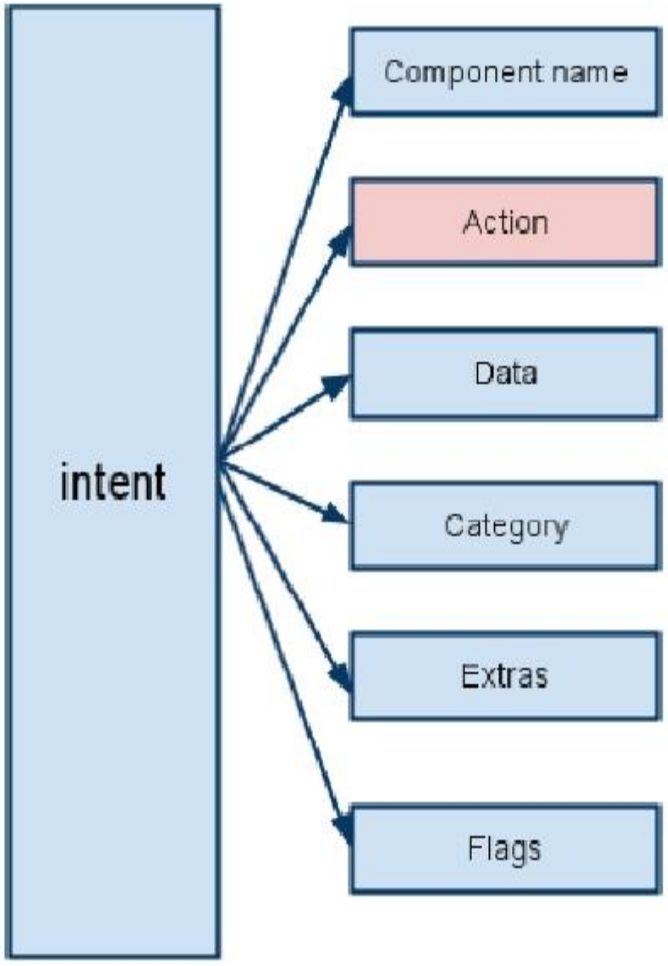
- 要开启一个网页，由**Android**去决定谁要开启网页。

Intent激活方式

- **Activity**：通过调用Context.startActivity()或者Activity.startActivityForResult()方法。
- **Service**：调用Context.startService()或者Context.bindService()方法将调用这方法的上下文与Service绑定。
- **BroadcastReceiver**：通过Context.sendBroadcast()、Context.sendOrderBroadcast()和Context.sendStick
- **Broadcast**()发送BroadcastIntent。 BroadcastIntent发送后，所有已注册的拥有与之匹配的IntentFilter的BroadcastIntent就会被激活。

Intent的组成部分

Intent对象主要包含六类信息。并非每个都需要包含这六类信息。



1. **组件名称**：Intent目标组件的名称。
2. **动作**：描述Intent所触发动作名字的字符串。
3. **Data**：描述Intent要操作的数据URI和数据类型。
4. **Category**：对被请求组件的额外描述信息。
5. **Extra**：当我们使用Intent连接不同的组件时，有时需要在Intent添加额外的信息，以便把数据传递给目标Activity。
6. **Flag**：借助 Intent 中的 flag 和 AndroidMainifest.xml 中 activity 元素的属性，就可以控制到 Task 里 Activity 的关联关系和行为。

Intent的组成部分

1、组件名称：Component name

- 指定Intent的目标组件的类名称
- 通常 Android框架会根据Intents中包含的其它属性的信息，比如action、data/type、category进行查找，最终找到一个匹配的目标组件。但若Component这个属性有指定的话，将直接使用它指定的组件，而不再执行上述查找过程。

指定了这个属性以后，Intent的其它所有属性都是可选的。

Intent的组成部分

2、动作：Action

- 对执行动作的描述：操作(Action)
- 系统自定义了很多Action，其中最熟悉的是 “android.intent.action.MAIN”，可以在每个AndroidManifest.xml文档中找到，标记当前Activity作为一个程序的入口。
- 自己也可以自定义Action

Intent的组成部分

3、数据：Data

- 对于这次动作相关联的数据进行描述，表现为一个URI
- **Action 和 Data 示例**
  1. 打开浏览器显示网页
     - Uri uri = Uri.parse("http://www.sysu.edu.cn");
     - Intent intent = new Intent(Intent.ACTION_VIEW,uri);
     - startActivintenty(intent);
  2. 拨打电话，调用拨号程序
     - Uri uri = Uri.parse("tel:13800138000");
     - Intent intent = new Intent(Intent.ACTION_DIAL, uri);
     - startActivintenty(intent);

Intent的组成部分

Action 和 Data 示例

1. 调用短信程序发送短信

```
Uri uri = Uri.parse("smsto:13800138000");
Intent intent = new Intent(Intent.ACTION_SENDTO, uri);
intent.putExtra("sms_body", "www.sysu.edu.cn");
startActivintenty(intent);
```

2. 从通讯录中选择一个联系人

```
Intent intent = new Intent(Intent.ACTION_PICK,
                    ContactsContract.Contacts.CONTENT_URI);
startActivityForResult(intent, 0);
```

在返回的Intent中getData()可以得到选中的联系人的URI

Intent的组成部分

4、类别：Category

对执行动作的附加信息进行描述，例如：

- CATEGORY_DEFAULT
- CATEGORY_BROWSABLE 可以由浏览器打开
- CATEGORY_TAB 可以嵌入带选项卡的父活动
- CATEGORY_LAUNCHER 顶级活动,在启动屏幕上列出该活动
- CATEGORY_HOME 启动手机时或按下Home键时运行

Intent的组成部分

CATEGORY_HOME 示例

```
<activity android:name=".HelloPicActovity"
            android:label="@string/app_name">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.HOME" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
</activity>
```

按下HOME键时,该应用会被运行

Intent的组成部分

5、附加：Extras

- 附件信息(extras)，用来传递数据。
- 它是一个Bundle类的对象，由一组可序列化的key/value 对组成。

Intent的组成部分

6、标记：Flags

- 指导如何来启动Activity，是用来指明运行的模式。
- 例如，期望这个Intent的执行者和运行在两个完全不同的任务进程中，就需要设置FLAG_ACTIVITY_NEW_TASK 的标志位。

Bundle类型

Bundle是个类型安全的容器，保存的是名值对，值只能是基本类型或基本类型数组。如：String、Int、byte、boolean、char等。

bundle ['bʌndl]
n. 束；捆

为Intent附加数据

Intent提供了各种常用类型重载后的putExtra()方法，如： putExtra(String name, String value)、 putExtra(String name, long value)，在putExtra()方法内部会判断当前Intent对象内部是否已经存在一个Bundle对象，如果不存在就会新建Bundle对象，以后调用putExtra()方法传入的值都会存放于该Bundle对象，下面是Intent的putExtra(String name, String value)方法代码片断：

```
public class Intent implements Parcelable {
    private Bundle mExtras;
    public Intent putExtra(String name, String value) {
        if (mExtras == null) {
            mExtras = new Bundle();
        } mExtras.putString(name, value);
    return this;   }
```

为Intent附加数据

第一种写法：

```
Intent intent = new Intent();
Bundle bundle = new Bundle();
bundle.putString("name","tom");
intent.putExtras(bundle);
```

第二种写法：等价于上面的写法，这种写法使用起来比较方便，而且只需要编写少量的代码。

```
Intent intent = new Intent();
intent.putExtra("name", "tom");
```

IntentFilter

- **则用于描述一个Activity（或IntentReceiver ）能够操作哪些intent。**

  一个activity 如果要显示一个人的联系方式时，需要声明一个IntentFilter，这个IntentFilter 要知道怎么去处理VIEW 动作和表示一个人的URI。

- **IntentFilter 需要在AndroidManifest.xml 中定义。**

IntentFilter

- **Intent Filter：向系统说明该Activity可以做什么？**
- **通过解析各种intent，从一个屏幕导航到另一个屏幕是很简单的。**
  1. 当向前导航时，activity 将会调用startActivity(IntentmyIntent)方法。
  2. 然后，系统会在所有安装的应用程序中定义的IntentFilter 中查找，找到最匹配Intent 对应的activity。
  3. 新的activity 接收到通知后，开始运行。
  4. 当startActivity 方法被调用将触发解析Intent 的动作。
- **该机制提供了两个关键好处：**
  1. Activities 能够重复利用从其它组件中以Intent 的形式产生的一个请求；
  2. Activities 可以在任何时候被一个具有相同IntentFilter 的新的Activity 取代。

IntentReceiver

- IntentReceiver 在AndroidManifest.xml 中注册，也可在代码中使用Context.registerReceiver()进行注册。
- 当一个intentreceiver 被触发时，应用不必对请求调用intentreceiver，系统会在需要的时候启动你的应用。
- 各种应用还可通过Context.broadcastIntent()将自己的intentreceiver 广播给其它应用程序。
- 当希望应用能够对一个外部的事件(如电话呼入，数据网络可用，或者某个定时)做出响应，可以使用一个IntentReceiver。虽然IntentReceiver 在感兴趣的事件发生时，会使用NotificationManager通知用户，但它并不能生成一个UI。

Intent的调用方式

基本调用格式

序号	语　句	说　明
1	Intent intent =new Intent()	声明intent对象
2	Intent.setClass(……, ……)	当前的activity 目标Activity
3	Bundle bundle= new Bundle()	声明Bundle对象
4	Bundle.putString(“ ”,” ”)	设置对象内容
5	Intent.putExtras(bundle)	传递数据
6	startActivity(intent)	启动Activity

Intent的调用方式

Intent分为显式和隐式两种：

- **显式Intent：**明确指出了目标组件名称。 显式Intent更 多用于在应用程序内部传递消息。比如在某应用程序内，一个Activity启动一个Service。
- **隐式Intent：**没有明确指出了目标组件名称。 Android系统使用IntentFilter来寻找与隐式Intent相关的对象。隐式Intent恰恰相反，它不会用组件名称定义需要激活的目标组件，它更广泛地用于在不同应用程序之间传递消息。

自定义Action

- **Explicit Intent**：明确的指定了要启动的Acitivity，比如以下Java代码： Intent intent= new Intent(this, B.class)
- **Implicit Intent**：没有明确的指定要启动哪个Activity，而是通过设置一些Intent Filter来让系统去筛选合适的Acitivity去启动。

自定义Action

- intent到底发给哪个activity，需要进行三个匹配，action，category和data。
- 理论上来说，若intent不指定category，那么无论intent filter的内容是什么都应该是匹配的。但如果是implicit intent，Android默认给加上一个CATEGORY_DEFAULT，因此若intent filter中无android.intent.category.DEFAULT这个category的话，匹配测试就会失败。因此若activity支持接收implicit intent的话就一定要在intent filter中加入android.intent.category.DEFAULT。
- 例外情况是：android.intent.category.MAIN和android.intent.category.LAUNCHER的filter中没有必要加入android.intent.category.DEFAULT，当然加入也没有问题。
- 自定义的activity如果接受implicit intent的话，intent filer就一定要加上android.intent.category.DEFAULT这个category。

拨打电话/发短信（1）


```
Intent intent=new Intent(Intent.ACTION_CALL,Uri.parse("tel:"+telPhone));

Intent intent=new Intent(Intent.ACTION_SENDTO,Uri.parse("smsto:5554"));

intent.putExtra("sms_body", "Hello");

startActivity(intent);
```

拨打电话/发短信（2）

实际无法拨打电话，安全性问题

```
<uses-permission
    android:name="android.permission.CALL_PHONE"/>
 <uses-permission
    android:name="android.permission.SEND_SMS"/>
```

启动新的Activity(一)

通过startActivity(Intent intent)方法打开新的Activity。在打开新的Activity前，**可以决定是否为新的Activity传递参数**。

第一种：打开新的Activity，不传递参数

```
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {.....
            Button button =(Button) this.findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener(){//点击该按钮会打开一个新的Activity
            public void onClick(View v) { //新建一个Intent，第一个参数为当前Activity类对象，第二个参数为你要打开的Activity类
                    startActivity(new Intent(MainActivity.this, NewActivity.class));
        }}); }}
```

启动新的Activity(二)

第二种：打开新的Activity，传递参数

```
public class MainActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
     .......
     button.setOnClickListener(new View.OnClickListener(){//点击该按钮会打开一个新的Activity
            public void onClick(View v) {
            Intent intent = new Intent(MainActivity.this, NewActivity.class)
            Bundle bundle = new Bundle();//该类用作携带数据
            bundle.putString("name","tom");
            bundle.putInt("age",4);
            intent.putExtras(bundle);//附带上额外的数据
            startActivity(intent);
    }}); }}
```

在新的Activity中接收前面Activity传递过来的参数：

```
public class NewActivity extends Activity {
        protected void onCreate(Bundle savedInstanceState) {
         Bundle bundle = this.getIntent().getExtras();
         String name = bundle.getString("name");
                    int age = bundle.getInt("age");
             }
     }
```

获取Activity返回的数据

- 假若从T1Activity跳转到下一个T2Activity，而当这个T2Activity调用了finish()方法以后，程序会自动跳转回T1Activity，并调用前一个T1Activity中的onActivityResult( )方法，必须在前面的Activity中重写onActivityResult(int requestCode, int resultCode, Intent data)方法；
- **相关函数:**
  - startActivityForResult(Intent intent, Int requestCode)
  - setResut(int resultCode, Intent intent)
  - onActivityResult(int requestCode, int resultCode, Intent intent)

获取Activity返回的数据


```
public class MainActivity extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {    .....
    Button button =(Button) this.findViewById(R.id.button);
        button.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {   //第二个参数为请求码，可以根据业务需求自己编号
            startActivityForResult (new Intent(MainActivity.this, NewActivity.class),1);
            }});
    }
   //第一个参数为请求码，即调用startActivityForResult()传递过去的值  第二个参数为结果码
    @Override
    protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    String result = data.getExtras().getString("result"));//得到新Activity 关闭后返回的数据 }}
```

当新Activity关闭后，新Activity返回的数据通过Intent进行传递，Android平台会调用前面Activity 的onActivityResult()方法，把存放了返回数据的Intent作为第三个输入参数传入，在onActivityResult()方法中使用第三个输入参数可以取出新Activity返回的数据。

获得Activity返回的数据

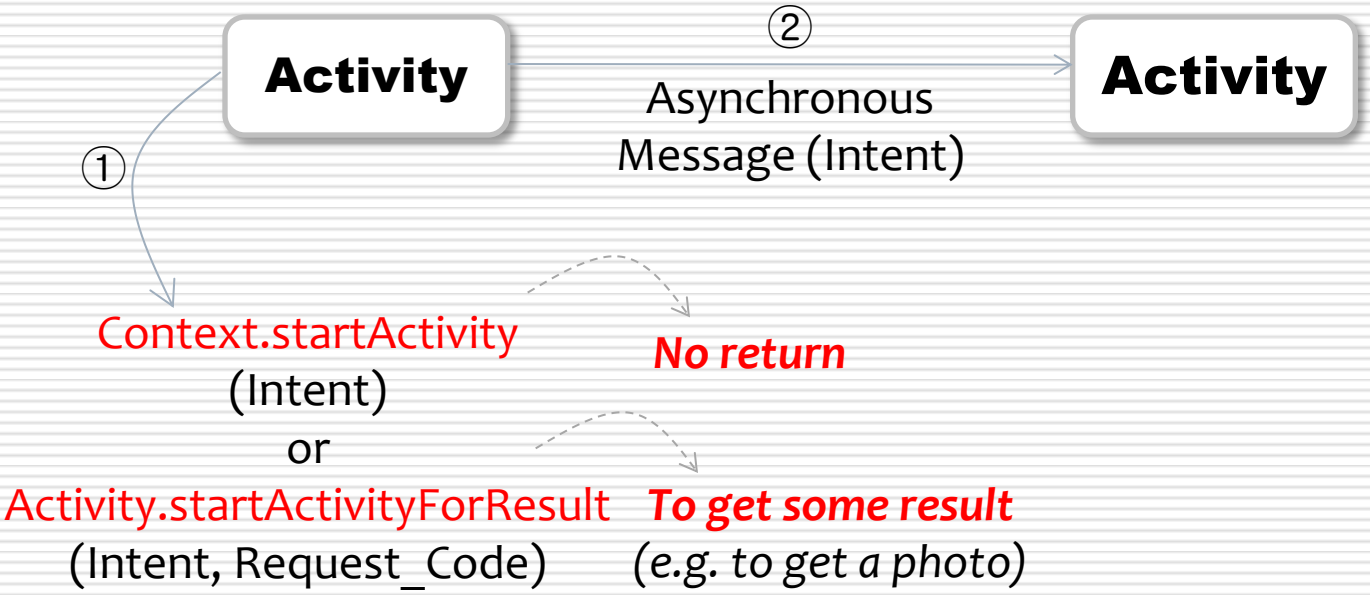
使用startActivityForResult(Intent intent, int requestCode)方法打开新的Activity，新Activity关闭前需要向前面的Activity返回数据需要使用系统提供的**setResult(int resultCode, Intent data)**方法实现：

```
public class NewActivity extends Activity {
    @Override protected void onCreate(Bundle savedInstanceState) {
        ……
        button.setOnClickListener(new View.OnClickListener(){
            public void onClick(View v) {
            Intent intent = new Intent();//数据是使用Intent返回
            intent.putExtra("result", "hello");//把返回数据存入Intent
            NewActivity.this.setResult(RESULT_OK, intent);//设置返回数据
            NewActivity.this.finish();//关闭Activity
        }});
    }
}
```

***setResult()**方法的第一个参数值可以根据业务需要自己定义，上面代码中使用到的RESULT_OK是系统Activity类定义的一个常量，值为-1*

活动（Activities）和任务（ Tasks）

通过Intent，一个活动（Activity）可以启动另一个活动（可以属于不同的应用）



任务栈的概念

- 压栈
- 弹栈

每个Activity的状态是由它在Activity栈(是一个后进先出LIFO，包含所有正在运行Activity的队列)中的位置决定的。

当一个新的Activity启动时，当前的活动的Activity将会移到Activity栈的顶部。

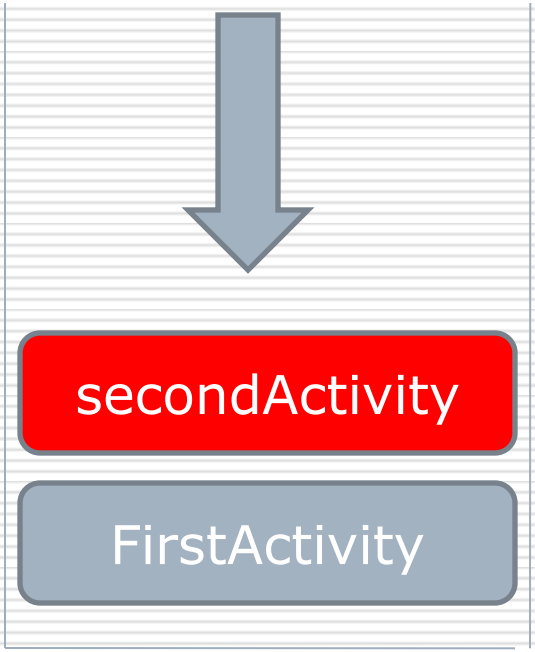
如果用户使用后退按钮返回的话，或者前台的Activity结束，活动的Activity就会被移出栈消亡，而在栈上的上一个活动的Activity将会移上来并变为活动状态。

任务栈运行过程

- 应用程序启动后，运行第一个Activity之后，该Activity对象被压入到Stack之中

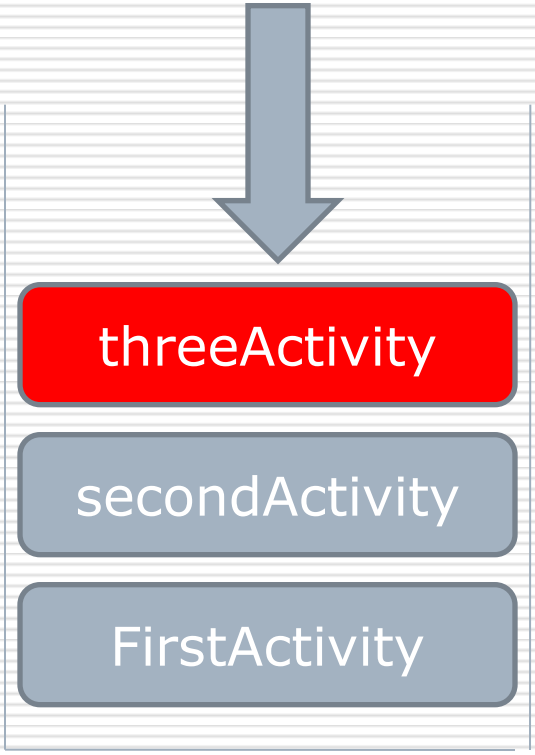
任务栈运行过程

点击按钮后启动第二个Activity，该对象被压入到Stack中



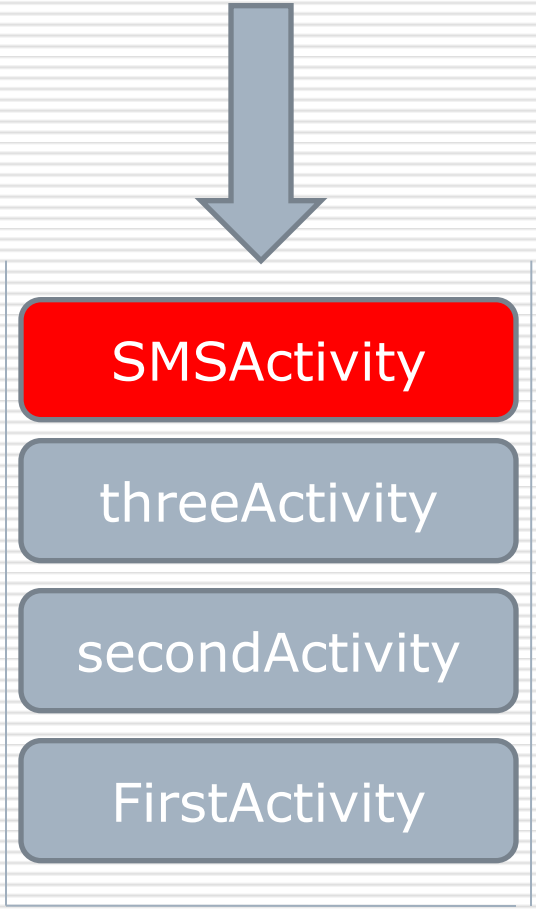
任务栈运行过程

点击第二个Activity按钮启动，该对象被压入到Stack中



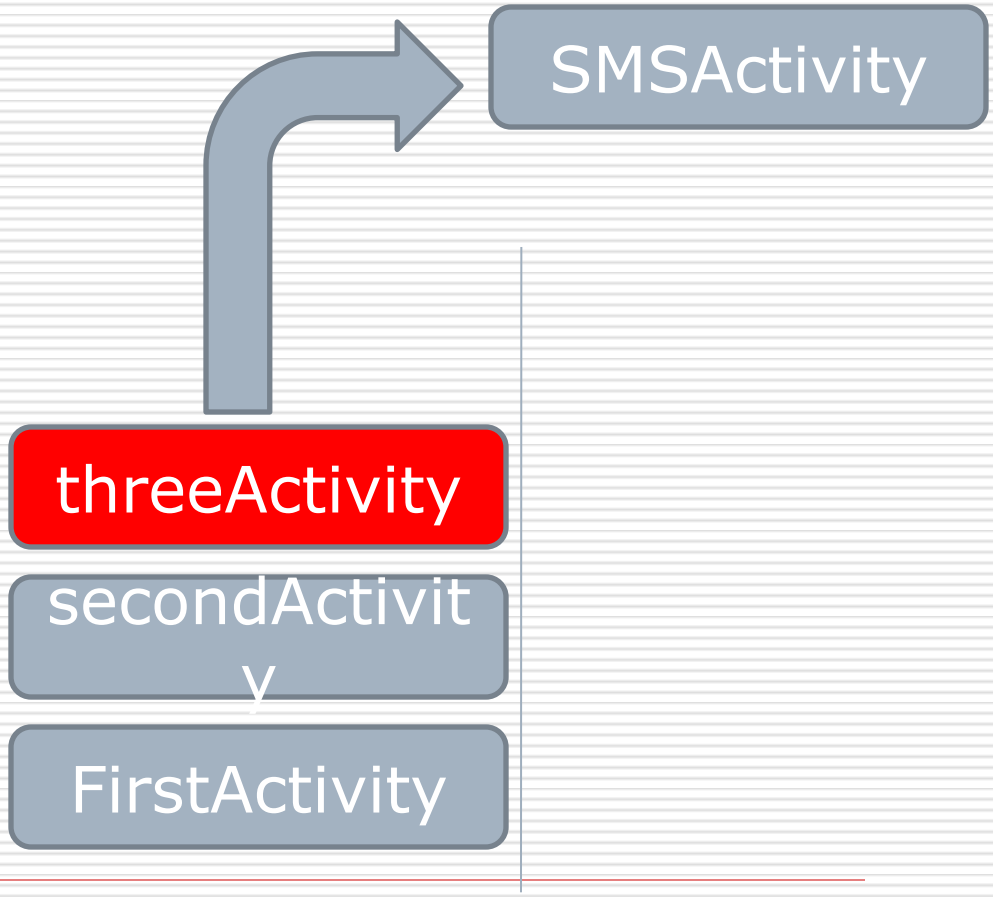
任务栈运行过程

- 当点击第三Activity中的按钮启动之后，启动第四个Activity



任务栈运行过程

- 点击Back之后SMSActivity从堆栈中弹出。
- 后面的对象都是一样。
- 所有的Activity在同一个Task，被组织称同一个单元。



任务栈和返回堆栈（Tasks & Back Stack）

一个应用程序通常包含多个Activity。每个Activity都必须设计成一种特定的操作， 用户可以通过该操作去实现某项功能，并且操作其他的Activity。

例如：一个电子邮件的应用程序可能有一个Activity，用于展现出新的电子邮件列表，当用户选择了一个电子邮件，就打开一个新的Activity以查看该电子邮件。

任务栈和返回堆栈（Tasks & Back Stack）

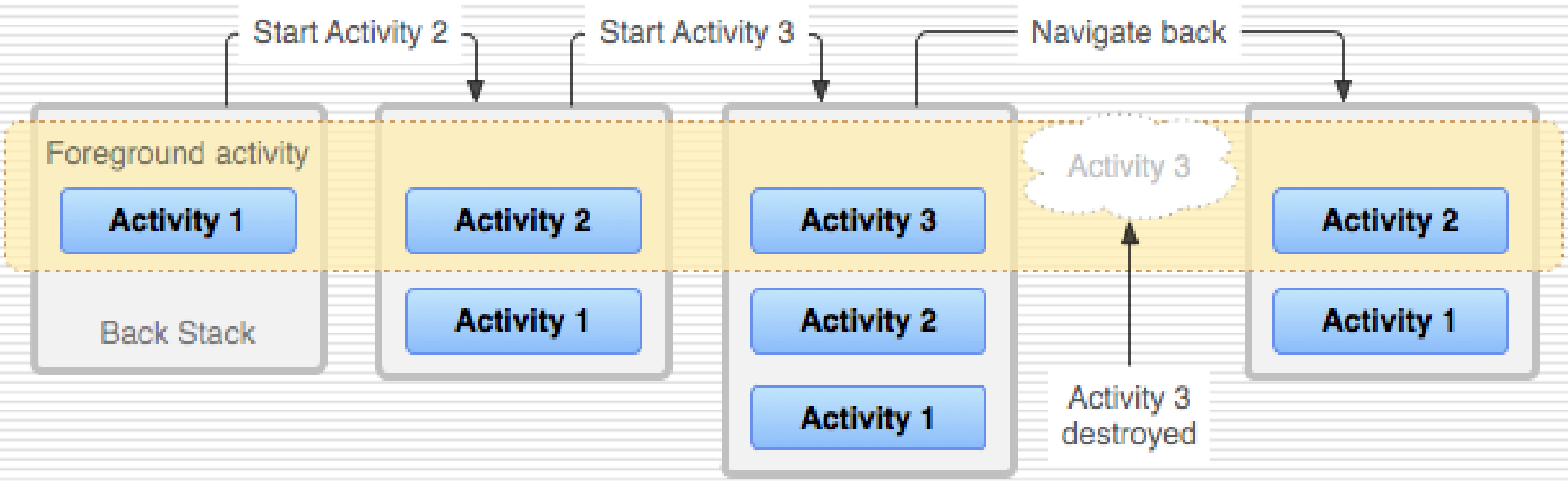
当用户进行某项操作时，任务栈就收集相互交互的Activity。Activity会被安排在堆栈中(返回堆栈)，堆栈中的Activity会按顺序来重新打开。

一个设备的主屏幕是大多数任务栈的起点。当用户触摸图标(或者在屏幕上的快捷方式)时，该应用程序就会到达任务栈的最前面。若该应用在任务栈中不存在(应用在近段时间内没有使用过)就会在任务栈中创建一个新的任务，并将该应用作为"主"Activity放置在根任务栈中。

任务栈和返回堆栈（Tasks & Back Stack）

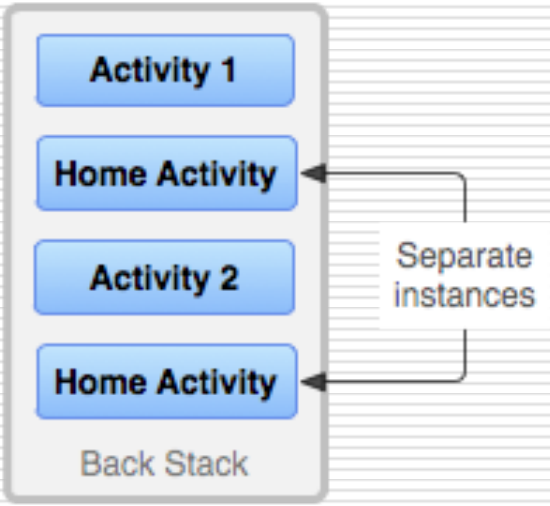
当Activity开始时，新的Activity推入堆栈的顶部和焦点。以前的Activity仍然在堆栈中，但已停止。当Activity停止时，系统会保留其用户界面的当前状态。当用户按下返回按钮时，当前Activity就会从堆栈的顶部(当前的Activity就会被销毁)和之前的一个Activity就会恢复(恢复到之前的UI界面)。 Activity在栈中的顺序永远不会改变，只会压入和弹出——被当前Activity启动时压入栈顶，用户用返回键离开时弹出。 这样，back stack 以“后进先出”的方式运行。下图以时间线表的方式展示了多个Activity切换时对应当前时间点的Back Task状态。

任务栈和返回堆栈（Tasks & Back Stack）



任务栈和返回堆栈（Tasks & Back Stack）

因为Back Stack 中的Activity顺序永远不会改变，如果应用允许某个Activity 可以让用户启动多次，则新的实例会压入栈顶（而不是打开之前位于栈顶的Activity）.于是一个Activity 可能会初始化多次（甚至会位于不同的Task 中），如图所示.



如果用户用返回键键返回时，Activity 的每个实例都会按照原来打开的顺序显示出来（用户界面也都按原来状态显示.当然，如果你不想让Activity 能被多次实例化，你可以改变它.

任务栈和返回堆栈（Tasks & Back Stack）

Activity和Task的默认行为

- 当 Activity A 启动 Activity B 时,Activity A 被停止,但系统仍会保存Activity A 状态（比如滚动条位置和 form 中填入的文字）如果用户在 Activity B 中按下返回键时,Activity A 恢复运行，状态也将恢复。
- 当用户按下Home键离开Task 时，当前 Activity 停止Task 转入后台,系统会保存Task中每个 Activity 的状态。如果用户以后通过选中启动该 Task 的图标来恢复Task,Task 就会回到前台，栈顶的Activity 会恢复运行。
- 如果用户按下返回键,当前Activity 从栈中弹出,并被销毁.栈中前一个Activity恢复运行.当 Activity 被销毁时，系统不会保留Activity 的状态。
- Activity甚至可以在不同的Task中被实例化多次

任务栈和返回堆栈（Tasks & Back Stack）

- 系统默认会在Activity 停止时保存其状态。当用户返回时，用户的界面能与离开时显示得一样。不过也应该使用用回调方法主动地保存Activity的状态,以便应对Activity被销毁并重建的情况；
- 当系统停止一个Activity 运行后（比如启动了一个新Activity 或者Task 转入后台)，系统在需要回收内存的时候有可能会完全销毁该Activity。这时该Activity的状态信息将会丢失。就算这种情况发生,该Activity 仍然会存在于Back Stack中.但是当它回到栈顶时，系统将必须重建它（而不是恢复）。为了避免用户工作内容的丢失，应通过实现Activity 的 onSaveInstanceState() 方法来主动保存这些内容。

onSaveInstanceState

- 在onPause()，onStop()以及onDestroy()中需要保存的是那些需要永久化的数据，而不是保存用于恢复状态的数据；
- onSaveInstanceState()保存状态数据。数据保存在一个Bundle中，Bundle被系统永久化。
- 当系统再次调用Activity的onCreate()时，原先保存的bundle就被传入，以恢复上一次临死时的模样，如果上次死时没有保存Bundle，则为null。

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

onSaveInstanceState

1. 由于Activity类已实现onSaveInstanceState()，在默认实现中，会调用所有控件的相关方法，把它们的状态都保存下来（如EditText中输入的文字）。因此即使程序未实现onSaveInstanceState()，但其上的控件状态可能依然能被保存并恢复。决于layout文件中为控件是否赋了一个名字(android:id)。有名的就存，无名的不管。
2. 开发者是否需要实现onSaveInstanceState()？

   若定制的派生类中有变量影响到UI或程序的行为，必须将此变量保存，就需要自己实现，否则就不需要。

任务栈和返回堆栈（Tasks & Back Stack）

把所有已经启动的Activity 相继放入同一个Task 中以及一个“后入先出”栈，

Android 管理Task和Back Stack 的这种方式适用于大多数应用，一般不用去管理

Activity 如何与Task关联及如何弹出Back Stack 。不过，有时候需要改变这种普通

的运行方式。也许想让某个Activity启动一个新的Task（而不是被放入当前Task中），

或者，想让 activity 启动时只是调出已有的某个实例（而不是在Back Stack 顶创建

一个新的实例）或者，若想在用户离开Task 时只保留根Activity，而Back Stack 中

的其它 Activity 都要清空。

Android事件处理模型

- 基于回调的事件处理
  - 重写Android等组件的特定的回调方法。在Activity和View类中,定义了若干回调方法,可以通过覆盖这些方法来实现事件处理。
- 基于监听接口的事件处理
  - 为Android界面组件绑定特定的事件监听器
- Handler

基于回调的事件处理

- 在Activity和View类中,定义了若干回调方法,可以覆盖这些方法来实现事件处理
- 返回boolean值,true表示完成了事件处理
  - onKeyDown(int keyCode,KeyEvent event)
  - onKeyUp(int keyCode,KeyEvent event)
  - onTouchEvent(MotionEvent event)
    - event.getAction()== MotionEvent.ACTION_DOWN
    - event.getAction()== MotionEvent.ACTION_UP
    - event.getAction()== MotionEvent.ACTION_MOVE

基于监听接口的事件处理

1. 对View调用 setOnXXXListener(listener)
2. View类中若干内部接口：
   - OnClickListener ：点击事件
   - OnLongClickListener: 长时点击
   - OnFocusChangeListener: 焦点改变
   - OnKeyListener 键盘事件
   - OnTouchListener 触摸事件
   - OnCreateContextMenuListener 上下文菜单显示事件

基于监听接口的事件处理

1. **事件源Event Source：**

   产生事件的来源，通常是各种组件，如按钮，窗口等。

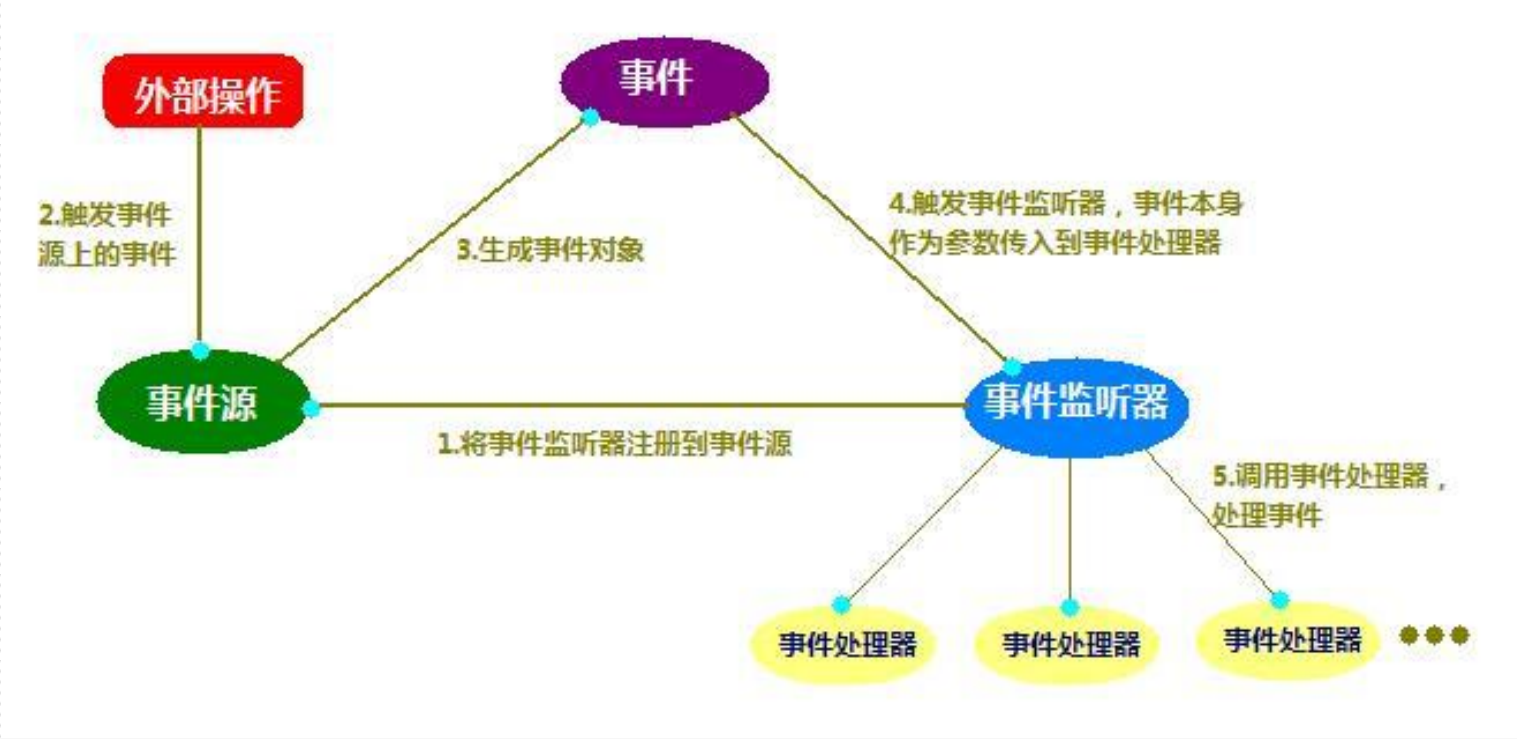
2. **事件Event：**

   事件封装了界面组件上发生的特定事件的具体信息，如果监听器需要获取界面组件上所发生事件的相关信息，一般通过事件Event对象来传递。

3. **事件监听器Event Listener：**

   负责监听事件源发生的事件，并对不同的事件做相应的处理。

基于监听接口的事件处理



基于监听器的事件处理机制是一种委派式Delegation的事件处理方式，事件源将整个事件委托给事件监听器，由监听器对事件进行响应处理。这种处理方式将事件源和事件监听器分离，有利于提供程序的可维护性。

OnClickListener

- OnClickListener接口处理的是点击事件。在触发模式下，是在某个View 上按下并抬起的组合动作，而在键盘模式下，是某个 View 获得焦点后点击确定键或者按下轨迹事件。
- **该接口对应回调方法签名：**
  - Public void onClick（View V）；**说明：**参数 V 便为事件发生的事件源

OnClickListener例程-(1)


```
public class Sample extends Activity implements OnClickListener{
//所有事件监听器方法都必须实现事件接口
Button[] buttons =  new Button[4];
TextView textView;
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        buttons[0] = (Button) this.findViewById(R.id.button01);
        buttons[1] = (Button) this.findViewById(R.id.button02);
        buttons[2] = (Button) this.findViewById(R.id.button03);
        buttons[3] = (Button) this.findViewById(R.id.button04);
        textView = (TextView) this.findViewById(R.id.textView01);
        textView.setTextSize(18);
```

OnClickListener例程-(2)


```
for(Button button : buttons){
        button.setOnClickListener(this);   //给每一个按钮注册监听器
        }
}
@Override
public void onClick(View v) {                    //实现事件监听方法
if(v == buttons[0]){                             //按下第一个按钮时
textView.setText("您按下了"+((Button)v).getText()+", 此时是分开处理
    的! ");
                        }
else{                                            //按下其他按钮时
textView.setText("您按下了" + ((Button)v).getText());
    }
}
```

OnLongClickListener

- **OnLongClickListener 接口和 OnClickListener 接口基本原理是相同的，只是该接口是长按事件的捕捉接口，即当长时间按下某个 View 时触发的事件。**
- **该接口对应方法的签名为：**

  Public boolean onLongClick（View V）；

  说明：

  - 参数 V ：参数 V 为事件源控件，当长时间按下此控件时才会触发该方法
  - 返回值 ： 该方法的返回值为一个 boolean 类型的变量，当返回为 true 时，表示已经完整地处理了这个事件，并不希望其他的回调方法再次进行处理；当返回为 false 时，表示并没有完成处理该事件，更希望其他方法继续对其进行处理。

OnLongClickListener例程

SUN YAT-SEN UNIVERSITY

```
public class Sample_7_5 extends Activity implements OnLongClickListener{
                                                        //实现的接口
Button button;//声明按钮的引用
    public void onCreate(Bundle savedInstanceState) {//重写的onCreate方法
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        button = (Button) this.findViewById(R.id.button);//得到按钮的引用
        button.setTextSize(20);
        button.setOnLongClickListener(this);              //注册监听     }
@Override
public boolean onLongClick(View v) {              //实现接口中的方法
if(v == button){//当按下的是按钮时
Toast.makeText(
this,
"长时间按下了按钮",
Toast.LENGTH_SHORT
).show();           //显示提示
}
return false;
}}
```

OnFocusChangeListener

- **OnFocusChangeListener** 接口用来处理控件焦点发生改变的事件。如果注册了该接口每当某个控件失去焦点或者获得焦点时都会触发该接口中的回调方法。
- 该接口对应的签名方法为：

Public void onFocusChangeListener（View V, Boolean hasFocus）

  - 说明：
  - 参数 V：参数 V便为触发该事件的事件源
  - 参数 hasFocus：参数hasFocus 表示v的新状态，即 v 是否获得焦点。

OnFocusChangeListener例程-(1)

SUN YAT-SEN UNIVERSITY

```
public class TestActivity extends Activity implements OnFocusChangeListener{
TextView myTextView;
ImageButton[] imageButtons = new ImageButton[4];
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        myTextView = (TextView) this.findViewById(R.id.myTextView);
    imageButtons[0] = (ImageButton) this.findViewById(R.id.button01);
     imageButtons[1] = (ImageButton) this.findViewById(R.id.button02);
     imageButtons[2] = (ImageButton) this.findViewById(R.id.button03);
     imageButtons[3] = (ImageButton) this.findViewById(R.id.button04);

        for(ImageButton imageButton : imageButtons){
        imageButton.setOnFocusChangeListener(this);          //添加监听
        }
    }
```

OnFocusChangeListener例程-(2)

SUN YAT-SEN UNIVERSITY

```
public void onFocusChange(View v, boolean hasFocus) {
//实现了接口中的方法
  if(v.getId() == R.id.button01){//改变的是button01时
     myTextView.setText("您选中了羊！");
  }else if(v.getId() == R.id.button02){//改变的是button02时
     myTextView.setText("您选中了猪！");
  }else if(v.getId() == R.id.button03){//改变的是button03时
     myTextView.setText("您选中了牛！");
  }else if(v.getId() == R.id.button04){//改变的是button04时
     myTextView.setText("您选中了鼠！");
  }else{//其他情况
     myTextView.setText("");
  }
 }
}
```

OnKeyListener

- OnKeyListener 是对手机键盘进行监听的接口，通过对某个 View 注册该监听接口，当 View 获得焦点并有键盘事件时，便会触发该接口中的回调方法。该接口中的回调方法如下：

public Boolean onkey（View v，int keyCode，KeyEvent event）；

说明：

- 参数 V：参数 v 为时间按的事件源控件。
- 参数 keyCode：参数 keyCode 为手机键盘的键码。
- 参数 event：参数 event 便为键盘事件封装类的对象，其中包含了事件的详细信息

OnKeyListener例程-(1)

SUN YAT-SEN UNIVERSITY

```
public class Sample_7_7 extends Activity implements
    OnKeyListener,OnClickListener{
ImageButton[] imageButtons = new ImageButton[4];//声明按钮数组
TextView myTextView;//声明TextView的引用
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.setContentView(R.layout.main);
        myTextView = (TextView) this.findViewById(R.id.myTextView);
        imageButtons[0] = (ImageButton) this.findViewById(R.id.button01);
     imageButtons[1] = (ImageButton) this.findViewById(R.id.button02);
     imageButtons[2] = (ImageButton) this.findViewById(R.id.button03);
     imageButtons[3] = (ImageButton) this.findViewById(R.id.button04);
     for(ImageButton imageButton : imageButtons){
        imageButton.setOnClickListener(this);//添加单击监听
        imageButton.setOnKeyListener(this);//添加键盘监听
        }
    }
```

OnKeyListener例程(2)


```
@Override
public void onClick(View v) {//实现了接口中的方法
   if(v.getId() == R.id.button01){//改变的是button01时
      myTextView.setText("您点击了按钮A! ");
   }else if(v.getId() == R.id.button02){//改变的是button02时
      myTextView.setText("您点击了按钮B! ");
   }else if(v.getId() == R.id.button03){//改变的是button03时
      myTextView.setText("您点击了按钮C! ");
   }else if(v.getId() == R.id.button04){//改变的是button04时
      myTextView.setText("您点击了按钮D! ");
   }else{//其他情况
      myTextView.setText("");
   }
}
```

OnKeyListener例程-(3)

SUN YAT-SEN UNIVERSITY

```
@Override
public boolean onKey(View v, int keyCode, KeyEvent event) {//键盘监听
switch(keyCode){//判断键盘码
case 29://按键A
      imageButtons[0].performClick();//模拟单击
      imageButtons[0].requestFocus();//尝试使之获得焦点
      break;
case 30://按键B
      imageButtons[1].performClick();//模拟单击
      imageButtons[1].requestFocus();//尝试使之获得焦点
      break;
case 31://按键C
      imageButtons[2].performClick();//模拟单击
      imageButtons[2].requestFocus();//尝试使之获得焦点
      break;
case 32://按键D
      imageButtons[3].performClick();//模拟单击
      imageButtons[3].requestFocus();//尝试使之获得焦点
      break;}
return false;
}
}
```

OnTouchListener

- **OnTouchListener 接口是用来处理手机屏幕事件的监听接口，当为 View 的范围内触摸按下、抬起或滑下等动作时都会触发该事件。该接口中的监听方法签名如下：**

  Public boolean onTouch（View V ,MotionEvent event）；

  说明：

  - 参数 v:：参数 v 为事件源对象
  - 参数 event：参数 event 为事件封装类的对象，其中封装了触发事件的详细信息，包括事件的类型、触发时间等信息。

OnTouchListener例程-(1)


```
public class Sample_7_8 extends Activity {  //不用实现接口
final static int WRAP_CONTENT=-2;//表示WRAP_CONTENT的常量
final static int X_MODIFY=4;//在非全屏模式下X坐标的修正值
final static int Y_MODIFY=52;//在非全屏模式下Y坐标的修正值
int xSpan;//在触控笔点击按钮的情况下相对于按钮自己坐标系的
int ySpan;//X,Y位置
    public void onCreate(Bundle savedInstanceState) {//重写的onCreate
    方法
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);//设置当前的用户界面
        Button bok=(Button)this.findViewById(R.id.Button01);
        bok.setOnTouchListener(  new OnTouchListener(){  //直接把注册
    事件源，所以不用实现接口并且也不用另外写事件。如果要另外写一个方法，要实现接
    口并且实现的接口方法要在 public voidonCreate（） 方法外。
```

OnTouchListener例程-(2)


```
public boolean onTouch(View view, MotionEvent event) {
switch(event.getAction()){
case MotionEvent.ACTION_DOWN://触控笔按下
     xSpan=(int)event.getX();ySpan=(int)event.getY();
     break;
case MotionEvent.ACTION_MOVE://触控笔移动
     Button bok=(Button)findViewById(R.id.Button01);
     //让按钮随着触控笔的移动一起移动
     ViewGroup.LayoutParams  lp=
     new AbsoluteLayout.LayoutParams(//初始化为一个子位置
     WRAP_CONTENT, WRAP_CONTENT, (int)event.getRawX()-xSpan-X_MODIFY,
     (int)event.getRawY()-ySpan-Y_MODIFY) ;
      bok.setLayoutParams(lp); break; }
return true;
}}); }
```

OnTouchListener例程-(3)


```
public boolean onKeyDown (int keyCode, KeyEvent event){//键盘键按下的方法
    Button bok=(Button)this.findViewById(R.id.Button01);
    bok.setText(keyCode+" Down");
    return true;}
    public boolean onKeyUp (int keyCode, KeyEvent event){//键盘键抬起的方法
    Button bok=(Button)this.findViewById(R.id.Button01);
    bok.setText(keyCode+" Up");
    return true;}
    public boolean onTouchEvent (MotionEvent event){//让按钮随着触控笔的移动一起移动
    Button bok=(Button)this.findViewById(R.id.Button01);
    ViewGroup.LayoutParams  lp=new AbsoluteLayout.LayoutParams(
WRAP_CONTENT, WRAP_CONTENT,
(int)event.getRawX()-xSpan-X_MODIFY, (int)event.getRawY()-ySpan-Y_MODIFY
) ;
bok.setLayoutParams(lp);
    return true;}}
```

Handler的定义

1. **UI控件更新？**
   - 主线程管理UI控件；
   - 耗时操作放在线程中，避免“假死”，导致FC；
   - Android主线程非线程安全
   - 主线程的UI和子线程间通过Handler；
2. **Handler主要接受子线程发送的数据，并用此数据配合主线程更新UI.**

Handler

在android.os包中，用于线程间传递消息，线程间需要共享同一个Handler对象

1. handlerMessage(Message msg)处理消息,需要在接收消息的线程中覆盖该方法
2. sendEmptyMessage(int what)发送空消息,在handlerMessage中,通过msg.what查看what值
3. sendMessage(Message msg) 发送消息

Handler

handler可以分发Message对象和Runnable对象到主线程中, 每个Handler实例,都会绑定到创建他的线程中(一般是位于主线程), 它有两个作用: (1): 安排消息或Runnable 在某个主线程中某个地方执行, (2)安排一个动作在不同的线程中执行

Handler中分发消息的一些方法

post(Runnable)
postAtTime(Runnable,long)
postDelayed(Runnable long)
sendEmptyMessage(int)
sendMessage(Message)
sendMessageAtTime(Message,long)
sendMessageDelayed(Message,long)

以上post类方法允许你排列一个Runnable对象到主线程队列中, sendMessage类方法, 允许你安排一个带数据的Message对象到队列中, 等待更新.

Handler-例程(1)

下面的示例是在一个子线程中定时向主线程发送消息，主线程收到消息后，根据消息携带的数据改变UI界面上色块的颜色。

定义子线程：

```
private class MyThread extends Thread {
    @Override
    public void run() {
        for (int i = 0; i < 20; i++) {  //  循环发送20次消息
            Message message = new Message();  //  构建Message对象
            Bundle bundle = new Bundle();  //  构建Bundle对象
            bundle.putInt("color", colors[i % colors.length]);  //  将数据放入Bundle中
            message.setData(bundle);  //  将Bundle放入Message对象中
            myHandler.sendMessage(message);  //  使用主线程的Handler发送消息
            try {
                sleep(2000);  //  暂停线程2秒
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

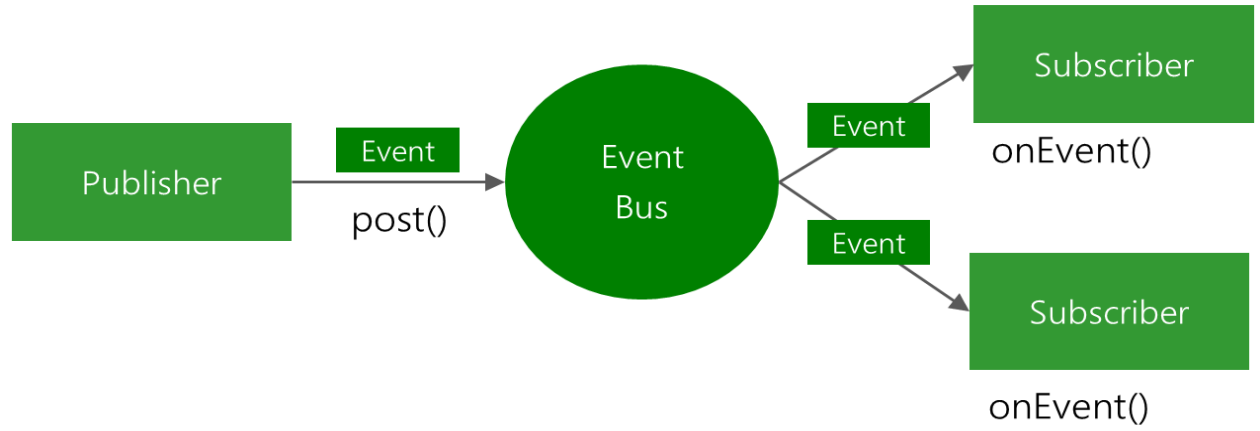
Handler-例程(2)

定义Handler处理消息的方式：

```
private class MyHandler extends Handler {
    @Override
    public void handleMessage(Message msg) {
        Bundle bundle = msg.getData();  //  从Message中取出Bundle
        int colorId = bundle.getInt("color");  //  从Bundle中取出color的Id
        image.setBackground(getResources().getDrawable(colorId));  //  根据id改变色块的颜色
    }
}
```

线程之间-EventBus

EventBus是Android下高效的发布/订阅事件总线机制。可以代替传统的Intent,Handler,Broadcast或接口函数在Fragment ,Activity , Service,线程之间传递数据，执行方法。其特点是代码简洁，是一种发布订阅设计模式（Publish/Subsribe）



EventBus的使用

1. 定义一个类作为事件。下面以MsgEvent为例。

2.添加订阅者：EventBus.getDefault().register(this); 将所在类作为订阅者。订阅者所在类可以定义以下一个或多个方法用以接收事件：

```
public void onEvent(MsgEvent msg)

public void onEventMainThread(MsgEvent msg)

public void onEventBackgroundThread(MsgEvent msg)

    public void onEventAsync(MsgEvent msg)
```

EventBus的使用

3. 发布者发布事件，一旦执行了此方法， 所有订阅者都会执行第二步定义的方法：EventBus.getDefault().post(new MsgEvent("主线程发的消息"));

4.取消订阅：EventBus.getDefault().unregister(this); 当订阅者不再被使用，或者被关闭时，最好进行取消订阅，不再接受事件消息。

Questions?